Métodos de Aprendizaje en Inteligencia Artificial. Bloque 1, Tema 1: Introducción

1. Introducción

El objetivo del presente bloque consiste en familiarizarnos con los conceptos básicos del área.

Este bloque se estructura del siguiente modo:

- Representación habitual de la experiencia como ejemplos/casos de entrenamiento (2 horas).
- Introducción a Scheme (dialecto de Lisp), lenguaje escogido para implementar los algoritmos de estas prácticas (5 horas).
- Scheme en métodos de aprendizaje para la IA: orientando Scheme hacia los problemas concretos de esta materia (7 horas).
- Construcción del primer algoritmo de aprendizaje de conceptos (8 horas).
- Evaluación del aprendizaje (4 horas).

Note que estas prácticas deben realizarse secuencialmente, ya que muchos ejercicios y conceptos están basados en otros anteriores. De hecho, es importante dedicar tiempo suficiente a cada uno de los ejercicios, ya que no sólo el conocimiento adquirido sino el propio código implementado para muchos de ellos se utilizarán a lo largo de todo el curso.

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

2. Representación de la experiencia mediante ejemplos de entrenamiento

En este texto limitaremos la representación de las experiencias para el aprendizaje (ver sección 1.3.1 del texto base [EML]) a diferentes ejemplificaciones de un cierto concepto. Denominaremos a cada ejemplificación *caso* o *ejemplo de entrenamiento*.

Cada ejemplo o caso de entrenamiento se representa mediante la descripción de cada uno de los atributos que lo forman. Como se puede ver en el texto base [EML] (ver sección 1.3.1), los dos tipos centrales de atributo son los *atributos nominales* y los *atributos numéricos*. De hecho, en esta asignatura no profundizaremos en formas de representar casos de entrenamiento más complejas.

Como ejemplo, situémonos en un contexto en el que deseamos aprender el concepto "buen día para salir al campo". Para representar las experiencias relacionadas con este concepto utilizaremos distintas ejemplificaciones (o instanciaciones) de la clase "día". De esta forma, los ejemplo de entrenamiento (o instancias) serán *día1*, *día2*, *día3*, etc. Finalmente, la representación explícita de cada caso se realizará mediante la descripción de sus atributos/valor.

Por ejemplo:

	perspectiva	temperatura	humedad	viento
día1	nublado	25º	80 %	no
día2	soleado	25º	70 %	sí
día3	lluvioso	20º	86 %	no
...				

Note que la forma de representar las experiencias tendrá gran incidencia en lo que puedan llegar a aprender los algoritmos. Por ejemplo, en vez de viento sí/no, podríamos representar cada caso de entrenamiento mediante viento fuerte/ligero/no. Incluso podríamos prescindir del atributo "viento", si lo consideramos irrelevante para este concepto; o añadir un nuevo atributo que sí consideremos relevante. Veremos en qué medida afecta este hecho al aprendizaje en temas sucesivos.

Por último, en el caso de aprendizaje con tutor o *supervisado* (ver sección 1.2.2 del texto base [EML]), que será en el que centraremos casi toda la asignatura, debemos representar la opinión de dicho tutor para cada caso. Para ello se añade un nuevo atributo, normalmente denominado "clase", empleado para asignar cada ejemplo de entrenamiento a una clase o concepto, y por tanto es un atributo que representa la opinión (o asignación) dada por el tutor para cada ejemplo de entrenamiento. Normalmente, la clase varía entre + y −, indicando que el ejemplo concuerda o no respectivamente con el concepto a aprender.

Así, por ejemplo, en el caso del concepto "buen día para salir al campo", como tutores podríamos realizar las siguientes asignaciones:

	perspectiva	temperatura	humedad	viento	clase
día1	nublado	25º	80 %	no	+
día3	soleado	25º	75 %	sí	+
día2	lluvioso	20º	86 %	no	−
...					

Ejercicio 1. *Siguiendo con el concepto "buen día para salir al campo", reconsidere los atributos y valores que se muestran en los ejemplos anteriores y proponga los suyos propios. Tras ello, construya su propio conjunto de 20 ejemplos de entrenamiento diferentes. Finalmente, en el papel de tutor, asigne la clase (+ ó −) para cada uno de estos ejemplos.*

Debido a que los casos generados van a ser utilizados en sucesivos ejercicios, se recomienda escribir los ejemplos en un archivo de texto ".txt" (p.ej. en un editor como el "block de notas/notepad"). Escriba cada ejemplo de entrenamiento en una línea *de texto diferente separando cada atributo/valor por medio de* espacios*. Utilice las primeras líneas del archivo para describir cada atributo: si es numérico, indicando este hecho; si es nominal, indicando los posibles valores que éste puede adquirir.*

3. Introducción a Scheme

El lenguaje Scheme es un dialecto sencillo del lenguaje Lisp. Este lenguaje está especializado en el procesamiento de listas (Lisp proviene de LISt Processing), una estructura de datos muy conveniente para representaciones simbólicas como las requeridas en esta asignatura. Existen varias implementaciones de este lenguaje, pero durante este curso utilizaremos `DrScheme` [DRS] una implementación dirigida al aprendizaje de este lenguaje. Esto permitirá hacer más suave el escalón que supone aprender Scheme, como será el caso de algunos de vosotros.

Daremos unos primeros consejos para comenzar cuanto antes a practicar este lenguaje. Para empezar se recomienda empezar por el documento Teach Yourself Scheme in Fixnum Days [TYSIFD]. Para detalles más concretos del lenguaje, poseen el documento de referencia [R5RS: Revised 5 Report on the Algorithmic Language Scheme] (puede encontrarlo en la plataforma). Dicho documento tiene escasamente 50 páginas, pero es referencia fundamental para programar en Scheme.

Ejercicio 2. *Definir la función* `(siguiente <lista>)`*, la cual recibe una lista de números enteros y devuelve una lista con cada uno de sus sucesores. Por ejemplo,* `(siguiente '(1 3 5)) => (2 4 6)`*.*

Ejercicio 3. *Definir la función* `(sumas <lista1><lista2>)`*, la cual recibe dos listas de números enteros y devuelve una lista con la suma de los elementos de las dos listas sumados uno a uno. Por ejemplo,* `(sumas '(1 3 5) '(2 4 6)) => (3 7 11)`*.*

Los siguiente ejercicios se pueden realizar mediante un vistazo rápido al documento [R5RS].

Ejercicio 4. *Construya la función* `(factorial <número>)` *de manera recursiva. Por ejemplo,* `(factorial 3) =>6`

Ejercicio 5. *Construya la función* `obtener-al-azar0` *que admita una lista cuyos elementos sean pares o* `cons`*es (ver sección 6.3.2 de [R5RS], página 25) que asocian un elemento a un número entero. La función debe devolver uno de estos elementos al azar. El número entero se debe relacionar con una frecuencia relativa: la frecuencia con debe aparecer su elemento asociado respecto al resto de elementos cuando se escogen al azar. Por ejemplo:*

```
# (obtener-al-azar0 '((a . 1)(b . 2)(c . 3)))
b
# (define elementos ())
# (do ((x 6000 (- x 1)))
           ((= x 0) elementos)
           (set! elementos
                (cons (obtener-al-azar0
                       '((a . 1)(b . 2)(c . 3)))
                )))
# (map (lambda(x)
                (cons x (count (lambda(y) (eq? x y)) elementos)))
             '(a b c))
((a . 998)(b . 2003)(c . 2999))
```

Para añadir azar a Scheme, utilice la función `random` *que devuelve un número aleatorio entre 0 y 1. El contenido teórico para desarrollar la función* `obtener-al-azar` *se denomina Método de la Transformada Inversa, y se puede encontrar en cualquier buen libro de probabilidad. En cualquier caso, en la plataforma del master se puede obtener el documento [ITM] que explica dicho método.*

Ejercicio 6. *Haga más potente la función* `obtener-al-azar` *permitiendo que admita también una lista simple de elementos, comportándose como si cada elemento de esta lista tuviera asociada una misma frecuencia. Para ello, la función debe detectar si los elementos de la lista son pares (*`cons`*es) elemento-número. Si lo son, ésta debe continuar con el código del ejercicio anterior. Si no, transformar la lista de elementos a una lista de pares elemento-numero, con dicho numero constante para todos los pares, y continuar con el código del ejercicio anterior. Por ejemplo:*

```
# (obtener-al-azar '(a b c))
c
# (define elementos ())
# (do ((x 6000 (- x 1)))
           ((= x 0) elementos)
           (set! elementos
                (cons (obtener-al-azar
                       '((a . 1)(b . 2)(c . 3)))
                )))
# (map (lambda(x)
                (cons x (count (lambda(y) (eq? x y)) elementos)))
             '(a b c))
((a . 1998)(b . 2003)(c . 1999))
```

4. Scheme como herramienta en métodos de aprendizaje para la IA

En la sección anterior empleamos Scheme como un lenguaje de propósito general. En esta sección, enfocaremos este lenguaje a los problemas particulares que nos conciernen en esta asinatura.

Una de las cualidades más importantes del lenguaje Scheme es la facilidad para manipulación de símbolos y listas de objetos. Esto nos permite manipular (obtener, soltar y mezclar) eficientemente tanto ejemplos de entrenamiento como sus atributos, haciendo que este lenguaje esté muy indicado para este tipo de algoritmos. En esta sección concretaremos funciones para resolver estos problemas.

La estructura que utilizaremos para escribir ejemplos de entrenamiento de ahora en adelante será la siguiente: el conjunto de ejemplos estará formado por una lista de listas, donde el primer objeto de esta lista será la descripción de los atributos (con su nombre y tipo); el resto de objetos serán los ejemplos de entrenamiento en sí, descritos como listas de valores. Veamos el ejemplo de la sección 2 en este formato.

```
(;principio de la lista de ejemplos.
 (;definición de los atributos
  (perspectiva (soleado nublado lluvioso));nominal
  (temperatura numerico);numerico
  (humedad numerico);numerico
  (viento (si no));nominal
  (clase  (+ -));nominal
  );fin definición de atributos

;;Ejemplos
```

```
(nublado 25 80 no +)
(soleado  25 70 si +)
(lluvioso 20 86 no -)
);fin de la lista de ejemplos
```

Ejercicio 7. *Actualice su archivo "ejemplos.txt" procedente del Ejercicio 1 con este nuevo formato. Para ello, genere un nuevo archivo denominado* ***''ejemplos.scm''****.*

Ejercicio 8. *Construya una función* ***(leer-ejemplos <archivo>)*** *que lea del archivo "ejemplos.scm" del ejercicio anterior y devuelva la lista de ejemplos. De hecho, se pide que dicha lista pueda quedar almacenada en la variable* ***ejemplos*** *para su uso en ejercicios posteriores. Utilice la función* ***with-input-from-file*** *(repasar la sección 6.6 del texto [R5RS]). Por ejemplo:*

```
# (define ejemplos (leer-ejemplos "ejemplos.scm"))
# ejemplos
((perspectiva (soleado ...)) ...)(nublado 25 80 no +) ...)
```

Ejercicio 9. *Construya la función* ***anadir-ejemplo*** *que admita como parámetros una listas de ejemplos más un nuevo ejemplo y devuelva una lista de ejemplos con el nuevo ejemplo incorporado. Por ejemplo:*

```
# (anadir-ejemplo ejemplos (soleado 5 90 si -))
((perspectiva (soleado ...)) ...)
             (nublado 25 80 no +) ... (soleado 5 90 si -))
```

Ejercicio 10. *Construya la función* ***atributo*** *que admita 2 parámetros: el primero será el nombre del atributo pedido y el segundo la lista de ejemplos. La función debe devolver una lista con el valor que posee cada uno de los ejemplos en dicho atributo. Por ejemplo, con la variable* ***ejemplos*** *del ejercicio anterior, una llamada con el parámetro* ***perspectiva*** *devolvería lo siguiente:*

```
# (atributo 'perspectiva ejemplos)
(nublado soleado lluvioso)
```

Pregunta: ¿Por qué debemos escribir ***'perspectiva*** *y no simplemente* ***perspectiva****?.*

Ejercicio 11. *Construya la función* ***mezclar*** *que admita como parámetros dos listas de ejemplos y devuelva otra con la mezcla de las estas dos.*

Ejercicios Avanzados

Ejercicio 12. *Construya la función* ***separar*** *que admita como parámetros una lista de ejemplos junto con una proporción y devuelva una lista con*

dos listas de ejemplos. La primera lista debe tener el porcentaje de ejemplos indicado por el parámetro, la segunda el resto de los ejemplos disponibles. Añada azar a la composición de las sublistas. Para ello, utilice la función `obtener-al-azar` *del ejercicio 6. Por ejemplo:*

```
# (separar 0.67 ejemplos)
(((nublado 25 80 no +)(soleado 25 70 si +))((lluvioso 20 86 no -)))
```

Ejercicio 13. *Construya la función* `folds` *que admita como parámetros una lista de ejemplos junto con un número natural y devuelva una lista formada por tantas sublistas (o folds) de ejemplos de tamaño similar (*similar *significa que varíen en tamaño en 1 elemento como máximo) como el parámetro indique. Añada azar a la composición de las sublistas. Para ello, utilice la función* `obtener-al-azar` *del ejercicio 6. Por ejemplo:*

```
# (folds 3 ejemplos)
(((nublado 25 80 no +))((soleado 25 70 si +))((lluvioso 20 86 no -)))

# (folds 2 ejemplos)
(((nublado 25 80 no +)(soleado 25 70 si +))((lluvioso 20 86 no -)))

# (folds 1 ejemplos)
(((nublado 25 80 no +)(soleado 25 70 si +)(lluvioso 20 86 no -)))
```

Ejercicio 14. *Construya la función* `stratify` *basada en la anterior función* `folds`*, pero restringida a que la proporción de ejemplos de cada clase se mantenga similar en cada* fold *o sublista de ejemplos y además similar a la existente en el conjunto total de ejemplos. Para ello, utilice la función* `obtener-al-azar` *de los ejercicios 5 y 6. Por ejemplo:*

```
# (stratify 2 ejemplos)
(((nublado 25 80 no +)(lluvioso 20 86 no -))((soleado 25 70 si +)))
```

5. Construcción del primer algoritmo de aprendizaje de conceptos

En esta sección daremos directrices generales para poder codificar algoritmos que *inducen* conceptos a partir de ejemplos de entrenamiento. No obstante, infinitos conceptos pueden ser inducidos de un mismo conjunto de ejemplos de entrenamiento. Es por esto que dedicaremos la próxima sección (sección 6) a medir la capacidad de un algoritmo para generar conceptos *precisos*.

Antes de pasar a codificar un algoritmo de aprendizaje, el primer paso debe ser establecer la manera de representar el concepto inducido a devolver. En otras palabras, debemos escoger el *lenguaje de descripción de conceptos*

del algoritmo (en ingles, *concept description language*, ver sección 1.3.2 del texto base [EML], página 12). De hecho, a alto nivel (al nivel del conocimiento) veremos que la forma de representar este conocimiento incide de manera definitiva en la eficiencia del algoritmo. Esto es debido al *sesgo/bias de representación* (en inglés *representational bias*) que lleva implícito cada tipo de representación (ver sección 1.4.2 del texto base [EML], página 18).

En cualquier caso, bajando al nivel del lenguaje de programación (al nivel simbólico), en Scheme utilizaremos básicamente las listas. Para ello, recomendamos tener a mano el documento [SRFI-1].

Como primer ejemplo de algoritmo de aprendizaje, implementaremos la función `A0`. El concepto devuelto por `A0` será simplemente la clase mayoritaria entre los ejemplos de entrenamiento. Así, el concepto inducido será representado por un solo símbolo que se corresponderá con uno de los valores posibles de clase. Por tanto, siguiendo nuestro ejemplo habitual, el concepto "buen día para salir al campo" será representado por uno de los símbolos $+$ ó $-$, que forman el *lenguaje de decripción de conceptos*[2] de `A0` para este dominio.

Veamos el código de `A0`:

```
(define (A0 ejemplos)
 (let*
   (;;Asignación de VARIABLES locales.
    ;;================================
    (atributos (car ejemplos))

    (casos (cdr  ejemplos))

    (indice-clase ;el índice de 'clase en la lista atributos.
      (list-index (lambda(x) (eq? x 'clase)) (map first atributos)))

    (clases-posibles
      (second ;sólo interesa el conjunto de valores.
        (list-ref atributos indice-clase)))

    ;;variable que mantiene la cuenta de las apariciones de cada clase.
    (clases-contabilizadas
     ;;Como primer paso, se inicializa a 0 la cuenta de cada clase.
     (map (lambda(clase) (cons clase 0)) clases-posibles))
```

[2]Note que a nivel del conocimiento $+$ significa que siempre hace buen tiempo (siempre es un buen día para salir al campo) y $-$ significa que nunca hace buen tiempo (nunca es un buen día para salir al campo). Observe que este lenguaje es muy limitado, lo que provoca que el concepto aprendido no resulte muy preciso en muchos dominios. No obstante, este hecho también permite que sea muy eficiente de aprender.

```
    (concepto ());variable sin asignación, de momento

    ;;Asignación de FUNCIONES locales.
    ;;================================
    ;;función que admite como parámetro un ejemplo, el cual utiliza
    ;;para actualizar la contabilización de clases.
    (actualizar-contabilizacion
      (lambda(ejemplo)
        (let ((clase-del-ejemplo (list-ref ejemplo indice-clase)))
          (set! clases-contabilizadas
            (map (lambda(x)
                 (if (eq? (car x) clase-del-ejemplo)
                  ;then
                  (cons (car x) (+ (cdr x) 1))
                  ;else
                   x))
              clases-contabilizadas
            )))))

     );fin de las asignaciones let*

  ;;Ahora, por cada ejemplo de entrenamiento,
  ;; se actualiza la contabilización de clases.
  (for-each actualizar-contabilizacion ejemplos)

  ;;Finalmente se escoge la clase que más veces ha aparecido:
   ;;;primero, se obtiene el número máximo;
  (set! concepto (apply max (map cdr clases-contabilizadas)))
   ;;;segundo, se obtiene la clase con ese número máximo
   ;;; (almacenado temporalmente en la variable concepto).
  (set! concepto
    (first (find (lambda(x) (= (cdr x) concepto)) clases-contabilizadas)))

  ;;Y por último se devuelve el concepto inducido por el algoritmo.
  concepto
))
```

Una vez ha sido inducido un concepto mediante `A0`, podemos considerar que se ha extraído (o aprendido) de alguna manera la "esencia" de los ejemplos que se le han pasado como parámetro. Esta *esencia* es lo que el texto base [EML] (ver sección 1.3.2, página 12) denomina *definición intensiva* (en inglés, *intensional definition*). Como allí se indica, recuperar la clase de los ejemplos de nuevo o *Extensión* a partir de su "esencia" o definición intensiva requiere la introducción de un *Intérprete*.

A continuación vamos a codificar el *Intérprete* necesario para obtener la *Extensión* a partir de los conceptos que devuelve la función `A0`. Denominaremos a dicho Intérprete `A0i`, el cual requiere dos parámetros: (1) el concepto o "esencia" y (2) el ejemplo que se desea recuperar, en otras palabras, el ejemplo (sin el atributo "clase") para el que se desea conocer su pertenencia o no al concepto aprendido[3].

```
(define (A0i concepto ejemplo-sin-clase)
  (append ejemplo-sin-clase (list concepto)))
```

Como ejemplo, ejecute lo siguiente a partir de sus propios ejemplos (los procedentes del ejercicio 7):

```
# (define esencia (A0 ejemplos))
# (define ejemplos-sin-clase (map (lambda(x) (drop-right x 1)) ejemplos))
# (define extension (map (lambda(x) (A0i esencia x)) ejemplos-sin-clase))
```

Ejercicio 15. *Tras ejecutar los comandos anteriores, comente las diferencias existentes entre el contenido de la variable* `ejemplos` *y el contenido de la variable* `extension`*. ¿Existen errores? ¿es* natural *que existan?.*

Ejercicio 16. *El número de aciertos o errores depende del tamaño de la muestra de ejemplos existentes. Por tanto, una buena medida de lo ajustado que resulta un determinado algoritmo a los casos que se le presentan es la relación de estos respecto al número de casos presentados. Se denomina* precision *a la relación de* aciertos *respecto al número de* casos *que se le han presentado* precision= $\frac{aciertos}{casos}$*. Se denomina error a su opuesto:* error= $\frac{errores}{casos}$*. En nuestro ejemplo, si existen 2 aciertos respecto a 3 casos presentados, la precisión sería de* $\frac{2}{3} = 0{,}666$*=67 %. El error sería el contrario,* $\frac{1}{3} = 0{,}333$*=33 %. Como ejercicio, calcule la precisión y el error de* `A0/A0i` *en su propia muestra de ejemplos de entrenamiento (procedente de los ejercicios 7 y 8).*

En el ejemplo anterior hemos utilizado el algoritmo `A0/A0i` como un *compresor* de los ejemplos de entrenamiento. Sin embargo, la utilidad más importante de un algoritmo de aprendizaje consiste en *predecir* la clase de posibles casos futuros. Por tanto, situándonos en el ejemplo habitual de estas prácticas, si introducimos *días* que nunca han sido considerados, en otras palabras, ejemplos sin el atributo "clase", el intérprete `A0i` servirá para *predecir* los días que se ajustan o no al concepto "buen día para salir al campo", incluso *antes* de que el tutor (que en estos ejemplos somos nosotros mismos) haya realizado ninguna asignación.

[3]Para facilitar la codificación de las prácticas, utilizaremos la siguiente convención: el atributo "clase" se localizará siempre en la última posición

Ejercicio 17. *Haga responder al intérprete* `A0i` *a partir de 5 días que usted imagine y que no estén presentes en el archivo de ejemplos del ejercicio 7. Como tutor que es usted, quien conoce con la máxima precisión el concepto "buen día para salir al campo", reflexione acerca de la respuesta dada por el par* `A0/A0i` *al tratar de clasificar estos 5 nuevos días imaginarios. Observe que estos nuevos días no están incorporados al concepto aprendido, por tanto ¿es comparable esta* precisión *con la que aparece en el ejercicio 16?.*

Como ayuda, cree el archivo "ejemplos2.scm" con estos 5 nuevos ejemplos (así podrá utilizarlos en ejercicios posteriores). Tras ello, utilice la función del ejercicio 8 para cargarlos en una nueva variable.

Ejercicio 18. *Codifique el par de funciones* `A1/A1i`*, donde* `A1` *es similar a* `A0` *exceptuando que el concepto devuelto debe ser el contenido de la variable* `clases-contabilizadas` *en vez del contenido de la variable* `concepto`*. Por ejemplo, teniendo en cuenta los ejemplos de este documento,* `A1` *devolvería* `((+ . 2)(- . 1))` *en vez de devolver únicamente* `+` *como hace* `A0`*. Además,* `A1i` *deberá clasificar cada nuevo ejemplo de entrenamiento en una de las dos clases (+ ó −) con* probabilidad *proporcional a la frecuencia de cada clase. Para ello, utilice la función* `obtener-al-azar` *del ejercicio 5.*

Ejercicio 19. *Mediante los 5 ejemplos (días) anteriores compare la precisión de* `A0/A0i` *respecto a la precisión de* `A1/A1i`*. ¿Qué algoritmo considera que induce mejores conceptos? ¿Considera que ha sido rentable aumentar el lenguaje de descripción de conceptos de* `A0/A0i` *a* `A1/A1i` *(haciendo el algoritmo* `A1/A1i` *más complejo)?.*

Algoritmos incrementales

Una de las características del componente aprendizaje (sección 1.4 del texto base [EML]) aparece al considerar si el aprendizaje es incremental o no (en concreto, ver sección 1.4.3, página 19 del texto base [EML]). Hasta ahora, tanto el algoritmo `A0` como el `A1` construyen su concepto en un solo paso, teniendo para ello a su disposición todos los ejemplos de entrenamiento. Sin embargo, un algoritmo incremental construye un concepto desde el primer ejemplo de entrenamiento, refinándolo según va encontrando más y más ejemplos.

Ejercicio 20. (Opcional) *A partir del algoritmo* `A1` *construya el algoritmo incremental* `IIA1` *que admite 3 parámetros: la descripción de los atributos, el concepto a refinar y un solo un ejemplo de entrenamiento. La función debe devolver un nuevo concepto que* refina *aquel introducido como parámetro mediante el nuevo ejemplo. Por ejemplo:*

```
# (define concepto-inicial '((+ . 0)(- . 0)))
# (IIA1 (car ejemplos) concepto-inicial '(nublado 25 80 no +))
((+ . 1)(- . 0))
```

Ejercicio 21. (Opcional) *En temas sucesivos nos va a interesar comparar algoritmos incrementales respecto a sus compañeros no incrementales. Para realizar esta comparación de manera cómoda, su llamada debería comportarse de manera similar. Para lograrlo, en este ejercicio se debe construir la función externa* `IA1` *de manera que admita los mismos parámetros que sus compañeras no incrementales y, al igual que ellas, devuelva un solo concepto final. Para ello,* `IA1` *debe llamar a* `IIA1` *tantas veces como ejemplos de entrenamiento se le hayan pasado como parámetro, devolviendo como resultado el concepto de la última llamada a* `IIA1`*. Fíjese que este ejercicio conecta con lo expresado en el libro base (sección 1.4.3, página 20, primer párrafo) en el que explica que un método incremental puede ser utilizado como si fuera no-incremental "by iteratively running it through the same training set".*

6. Evaluación del aprendizaje

Al igual que el número de curvas que pueden pasar a través de un número finito de puntos es infinito, así lo es el número de conceptos que pueden ser inducidos a partir de un conjunto finito de ejemplos de entrenamiento. Esto significa que no existe el algoritmo de aprendizaje perfecto que induzca el mejor de los conceptos para un determinado conjunto de ejemplos. Además, ni siquiera existe un algoritmo que induzca buenos conceptos para todo tipo de dominios o ejemplos. Por ejemplo, un algoritmo extremadamente bueno para inducir conceptos lineales en ciertos dominios numéricos, puede ser nefasto en ciertos dominios simbólicos, y al contrario.

Por tanto, para evaluar el aprendizaje realizado sobre determinado conjunto de casos de entrenamiento, tan solo podemos comparar empíricamente entre varios algoritmos y escoger el considerado mejor. Existen otras posibilidades, aunque menos comunes por no ser fácilmente aplicables de manera general (para ampliar, ver sección 2.7, página 84, del texto AA).

Para realizar esta evaluación empírica, se realizan comparaciones de *precisión* o *error* (ver Ejercicio 16) a partir de los ejemplos disponibles. Debido al tamaño finito y normalmente escaso de estos casos disponibles, se debe explotar al máximo cada uno de ellos. Para ello, se han desarrollado las técnicas siguientes.

Resustitución. Resustitution. En esta técnica se utilizan los mismos datos para entrenar y para evaluar (al igual que en los ejercicios 15 y 16). No cabe duda que ésta sería la forma más sencilla de evaluar explotando al máximo los ejemplos (cada ejemplo se utiliza para dos tareas: entrenar y evaluar). Sin embargo, es precisamente esta ventaja su máxima desventaja, ya que mediante esta técnica evaluamos principalmente la precisión tras *comprimir* los casos, pero no para *predecir* casos futuros.

De hecho, un algoritmo trivial que guarde la clase de cada caso para asignársela cuando vuelva a presentarse tendría una precisión del 100 % mediante esta prueba. Por tanto, esta prueba sobrevalora los algoritmos que se *sobreajustan* (en inglés, *overfit*) a los ejemplos de entrenamiento, y por tanto infravalora aquellas que se ajustan correctamente a los casos futuros no vistos previamente.

Dejar algunos ejemplos fuera. Holdout. Si queremos evaluar con mejor criterio la *predicción* de casos futuros, debemos mantener separados los ejemplos utilizados para entrenar de aquellos utilizados para evaluar. Algo parecido hemos realizado en los ejercicios finales (ejercicios 17 y 19) de la sección anterior. No obstante, normalmente no se crean nuevos ejemplos de entrenamiento para evaluar el algoritmo, sino que los ejemplos de entrenamiento existentes se dividen en dos subconjuntos cuyo tamaño se ajusta habitualmente siguiendo la relación $\frac{2}{3}$ para entrenar y $\frac{1}{3}$ para evaluar.

Sin embargo, surge una pregunta ¿no podría variar el resultado dependiendo de los ejemplos utilizados escogidos (normalmente al azar) para formar parte de cada subconjunto?. La respuesta resulta afirmativa, por lo que se ha desarrollado la siguiente técnica.

Dejar uno fuera. Leave one out. En efecto, la separación arbitraria del conjunto completo de ejemplos en dos subconjuntos puede dar lugar a falsas medidas. Por tanto, una técnica menos basada en la arbitrariedad consiste en hacer N separaciones distintas, una por cada uno de los N ejemplos de entrenamiento disponibles, mediante el procedimiento de coger un ejemplo para evaluar y el resto para entrenar. Así, tras realizar N pruebas en las que se entrena y prueba, conseguimos evaluar y entrenar sobre cada uno de los posibles ejemplos de entrenamiento, explotando al máximo su tamaño finito.

No obstante, este método tiene un problema. Por ejemplo, en el caso de tener a disposición la *razonable* cantidad de 1000 ejemplos de entrenamiento, habría que correr cada algoritmo 1000 veces. Imagine que el algoritmo tardara 1 segundo en entrenar $N-1$ ejemplos más clasificar el restante, entonces evaluar el algoritmo llevaría unos 17 minutos. Imagine, además, que deseamos medir su precisión en varios dominios. Por ello, podemos entender que se hayan desarrollado procedimientos intermedios entre la arbitrariedad de *Holdout* y el coste computacional de *Leave-one-out*. Los veremos a continuación.

Validación cruzada. Cross-validation. Esta técnica es una generalización del método *Leave-one-out*, en la cual N deja de asociarse al número de ejemplos total, determinándose como parámetro externo. Así, mediante esta técnica se divide el conjunto total de ejemplos en N

pedazos (denominados *fold* en inglés), utilizando, para cada una de las N posibles iteraciones, uno de estos pedazos para evaluar y el resto para entrenar. Finalmente, se suman el número de aciertos (o errores) procedentes de cada iteración y se calcula su precisión (o error). Habitualmente, dicho parámetro se ajusta a 10 debido a que empíricamente ha demostrado buen comportamiento, llegando incluso a tener denominación propia: *ten-fold-cross-validation*.

Note que, debido a la arbitrariedad asociada a la elección de cada subconjunto de ejemplos, podría resultar que los estos estuvieran algo sesgados en cuanto a su clase. Por ejemplo, podría ocurrir que uno de los subconjuntos contuviera únicamente ejemplos de clase $-$, lo que implicaría que un algoritmo como `A0/A0i` fuera totalmente nefasto o perfecto en uno de estos subconjuntos, sesgando así la evaluación. Es por ello que habitualmente se trata de conservar la proporción de clases del conjunto total de ejemplos en cada uno de los pedazos o folds, lo que se denomina *validación cruzada estratificada* (en inglés, *stratified crossvalidation*).

Ejercicio 22. *Construya las funciones* `resustitution` *y* `leave-one-out` *que admitan 3 parámetros: la función de entrenamiento, la función intérprete y un conjunto de ejemplos de entrenamiento. Dichas funciones deben devolver la precisión total del algoritmo de aprendizaje sobre dichos ejemplos. Compruebe mediante estas funciones la precisión de los algoritmos* `A0/A0i`*,* `A1/A1i` *e* `IA1/A1i` *mediante los 25 ejemplos creados entre los ejercicios 7 y 17. Para ello, recomendamos utilizar las funciones creadas en los ejercicios 8 y 11. Por ejemplo, una llamada correcta sería la siguiente:*

```
# (resustitution A0 A0i ejemplos)
0.6666666
```

Ejercicio 23. *Construya la función* `holdout` *que admita 4 parámetros: la función de entrenamiento, la función intérprete, un conjunto de ejemplos de entrenamiento y un conjunto de ejemplos de evaluación. La función debe devolver la precisión total del algoritmo de aprendizaje sobre los ejemplos de evaluación. Compruebe mediante esta función la precisión de los algoritmos* `A0/A0i`*,* `A1/A1i` *e* `IA1/A1i` *utilizando la relación dos tercios/un tercio para entrenar y evaluar mediante los 25 ejemplos creados entre los ejercicios 7 y 17. Para ello, recomendamos utilizar las funciones creadas en los ejercicios 8, 11 y 12. Por ejemplo, una llamada correcta sería la siguiente:*

```
# (holdout A0 A0i ejemplos '((soleado 30 50 si +)(lluvioso 4 90 -)))
0.50
```

Ejercicio 24. *A partir de los ejercicios 13 y 14, construya las funciones* `cross-validation` *y* `stratified-cross-validation` *que admitan 4*

parámetros: la función de entrenamiento, la función intérprete, un conjunto de ejemplos de entrenamiento y el número de folds. La función debe devolver la precisión total del algoritmo de aprendizaje sobre dichos ejemplos. Compruebe mediante estas funciones, con el parámetro folds a 10, la precisión de los algoritmos `A0/A0i`, `A1/A1i` *e* `IA1/A1i` *mediante los 25 ejemplos creados entre los ejercicios 7 y 17. Por ejemplo, una llamada correcta (respecto a los ejemplos de este documento) sería la siguiente:*

```
# (= (cross-validation A0 A0i ejemplos 3)(leave-one-out A0 A0i ejemplos))
 #t
```