**MARCO GONZÁLEZ, JUAN CARLOS**

**MARTÍNEZ DOTOR, JESÚS**

**PICADO ÁLVAREZ, MARÍA**

**PRÁCTICA 2 – INGENIERÍA DEL CONOCIMIENTO**

**Árboles de decisión (ID3)**

**4º Curso Grado en Ingeniería del Software**

Índice

[MEMORIA DE LA PRÁCTICA 2](#_Toc406165902)

[**Estructura de Clases** 2](#_Toc406165903)

[**Descripción de la parte obligatoria** 3](#_Toc406165904)

[**Implementación de la recursión completa** 3](#_Toc406165905)

[**Impresión del árbol de decisión final** 4](#_Toc406165906)

[**Tabla de reglas** 4](#_Toc406165907)

[**Clasificación de ejemplos de Test** 5](#_Toc406165908)

[MANUAL 6](#_Toc406165909)

CommonKADS 7

# MEMORIA DE LA PRÁCTICA

Esta práctica consiste en implementar una versión reducida del algoritmo ID3. Bastaría con implementar el primer nivel de recursividad, y estructurar los datos para aplicar la segunda recursividad.

A pesar de que la propuesta era implementar la versión reducida del algoritmo, nosotros **hemos decidido implementar la versión completa**, en la que generamos todos los niveles de recursión. Además, hemos decidido **imprimir por consola el árbol de decisión final**, estructurado para su correcta visualización, **además de la tabla re reglas** que nos servirá para la toma de decisiones.

Por último, hemos decidido dar la posibilidad al usuario de llevar a la práctica esta toma de decisiones, **empleando las reglas generadas para clasificar uno o varios ejemplos de Test**.

Como parte extra hemos decidido realizar la aplicación de manera que **acepte cualquier conjunto de datos** (atributos y ejemplos) y se puedan realizar pruebas de test sobre los mismos.

Para ello hemos empleado el lenguaje de programación Java, y el entorno de desarrollo Eclipse.

## **Estructura de Clases**

Nuestra práctica se compone de las siguientes clases:

* ***Principal.java* 🡪** Clase que contiene el “main”, desde el que se ejecuta la aplicación. Se encarga de llamar al resto de clases; primero cargando los datos desde fichero, y luego llamando a la clase *AlgoritmoID3.java,* que es la que se encarga de realizar el algoritmo.
* ***AlgoritmoID3.java* 🡪** Compone la parte principal de la práctica, en la que se realiza todo el proceso recursivo de clasificación de los atributos, además de la generación de reglas y posterior utilización de las mismas para clasificar los “test” de prueba. Esta clase es la que contiene la lista de Ejemplos y de Atributos.

La descripción concreta de la implementación se dará más adelante.

* ***Ejemplo.java* 🡪** Clase que almacena un ejemplo (cada fila del fichero de entrada) para su posterior utilización. Se compone de un Array con un String por cada atributo y un resultado booleano (que en nuestro ejemplo corresponde a “si” o “no”).
* ***Fichero.java* 🡪** Se encarga de realizar todas las operaciones con ficheros. Lectura de ficheros y formateo de los mismos.
* ***Constantes.java* 🡪** Contiene las constantes de nuestra aplicación.

## **Descripción de la parte obligatoria**

Para la parte obligatoria de la práctica, hemos partido del algoritmo ID3, y hemos tratado implementarlo de la forma más genérica posible, empleando estructuras de datos como la clase “Ejemplo.java”, para que así la ampliación de la práctica fuera más sencilla.

Dentro de la clase AlgoritmoID3 tenemos un método “aprenderID3”, que es el que se encarga de realizar el proceso.

Lo primero que hace este algoritmo es comprobar si la lista de ejemplos o de atributos que se pasa es una lista vacía (si es así acaba con una “excepción”). Después se evalúa si todos los ejemplos tienen el mismo resultado (todos positivos o todos negativos), ya que si esto fuera así, la regla ya estaría acabada, y concluiría el proceso.

Si hay ejemplos positivos y negativos, tenemos que calcular cuál es el atributo con menor mérito, esto es, cuál de los atributos divide mejor todos los ejemplos. Esta es la parte más complicada de la práctica, por su complejidad en el cálculo. Para calcular esto tenemos que hallar la entropía de cada una de las opciones posibles de cada atributo, que se hace mediante ña cuenta de ejemplos positivos y negativos de cada opción de atributo, y aplicando la fórmula:

***infor(p,n) = p log2(p)* n log2(n)**

Una vez calculada la entropía de cada opción, podemos calcular el mérito de cada atributo, y de todos ellos nos quedaremos el de menor mérito.

Este atributo seleccionado será el que nos servirá para “dividir” el árbol, así que lo que debemos hacer es añadir a la regla actual las distintas opciones de este atributo. Además, si queremos preparar los datos para la recursión, deberemos crear tantas tablas auxiliares como opciones tenga el atributo seleccionado. Estas talas serán las que emplearemos para la recursión. Además, deberemos eliminar de la tabla de atributos el atributo seleccionado.

Mediante este proceso hemos conseguido realizar el primer nivel de recursividad, por lo que habríamos creado la primera división del árbol, y habríamos preparado los datos para la siguiente recursión.

## **Implementación de la recursión completa**

La implementación de la recursión completa no es muy complicada si en la implementación normal ya se contaba con ello.

La única diferencia con el primer nivel de recursividad es que hay que tener más cuidado con qué datos modificamos, y que tablas enviamos a cada nivel de recursión, ya que serán estas tablas la que sigan generando las reglas sucesivas.

Si se ha generado un buen código, crear todos los niveles de recursión es tan fácil como llamar a

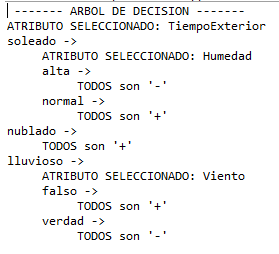
aprenderID3(ejemplos\_aux, atributos\_aux, atributos\_originales, nivel+1, regla\_aux);

por cada opción posible del atributo seleccionado.

Si hemos planteado bien los casos base, la recursión será automática y generará todas las reglas sucesivas hasta llegar al punto en que todos los ejemplos sean positivos o negativos (caso base).

## **Impresión del árbol de decisión final**

Como parte extra hemos decidido pintar el árbol de decisión de la siguiente forma:



Para poder imprimir el árbol de decisión del algoritmo, hemos decidido generar el árbol según se va creando, por lo que en cada paso imprimiremos qué decisión se toma. Por ejemplo, si se elige “x” atributo, o si se termina la recursión porque todos los ejemplos son positivos.

La única complejidad de este punto ha sido que para pintar el árbol, teníamos que llevar de alguna forma la cuenta de en qué profundidad se encontraba la recursión (porque si no nos pintaría todos los nodos al mismo nivel). Esto lo hemos solucionado añadiendo un parámetro “nivel” en el algoritmoID3 (inicializado a 0), y vamos aumentando en 1 el nivel cada vez que llamamos a la recursión.

Para generar el efecto de árbol imprimimos tantas tabulaciones como sea el valor del nivel al que nos encontramos.

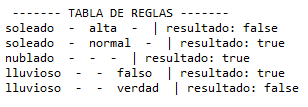
## **Tabla de reglas**

Para luego poder recuperar los valores de decisión, y poder implementar los test con ejemplos, teníamos que ir almacenando en alguna estructura las distintas reglas que se fueran generando.

Para ello hemos creado un Array de reglas en la clase *AlgoritmoID3.java*, y en ella guardaremos la regla resultante cada vez que lleguemos a un caso base (final de la recursión).

Además, la forma de generar cada regla es añadiendo a la regla la opción del atributo seleccionado, y pasando a la recursión una copia de esa regla “a medio hacer”.

Podemos ver un ejemplo de la tabla de reglas en la siguiente imagen:



## **Clasificación de ejemplos de Test**

Por último, hemos decidido implementar los test con ejemplos, para comprobar si nuestro algoritmo está bien.

Para ello bastaba con crear un nuevo ejemplo con los datos del test e ir comprobando parámetro por parámetro a cuál de las reglas correspondía.

Por ejemplo:

El test1, que contenía: nublado,frio,normal,verdad,??

Correspondía con la regla: nublado, - , - , - | resultado: true

Y el ej. de entrenamiento era: nublado,frio,normal,verdad,si

Por lo que podemos deducir que el resultado de este test será positivo, dato que concuerda con los ejemplos de entrenamiento.

De esta forma podemos confirmar que nuestra aplicación funciona correctamente, y puede clasificar cada uno de los ejemplos que se le faciliten.

# MANUAL

Para ejecutar la práctica lo primero es ejecutar el fichero “practica2.class”.

Para que la aplicación funcionara con muchos tipos de ejemplos, se ha decidido pasar como parámetros la ruta de los siguientes ficheros (en este orden):

<fichero atributos, fichero ejemplos, ficheros test (tantos como se desee)>

Por lo que un ejemplo de ejecución desde consola sería (si los ficheros estuvieran en el mismo directorio que la aplicación):

“java –jar Practica2.jar AtributosJuego.txt Juego.txt Test1Juego.txt Test2Juego.txt Test3Juego.txt”

Es importante que la separación entre elementos (ejemplos atributos, etc…) sea siempre el delimitador coma (,) ya que es este el que se utiliza para dividir los campos del fichero.

Una vez ejecutamos la aplicación pasándole por parámetro los ficheros, la aplicación se encarga de realizar todas las operaciones, y de ejecutar cada uno de los test de manera automática, por lo que el resultado visible por consola será el siguiente:

------- ARBOL DE DECISION -------

ATRIBUTO SELECCIONADO: TiempoExterior

soleado ->

ATRIBUTO SELECCIONADO: Humedad

alta ->

TODOS son '-'

normal ->

TODOS son '+'

nublado ->

TODOS son '+'

lluvioso ->

ATRIBUTO SELECCIONADO: Viento

falso ->

TODOS son '+'

verdad ->

TODOS son '-'

------- TABLA DE REGLAS -------

soleado - alta - | resultado: false

soleado - normal - | resultado: true

nublado - - - | resultado: true

lluvioso - - falso | resultado: true

lluvioso - - verdad | resultado: false

------- TEST DE CORRECION -------

RESULTADO Test1Juego.txt : true

RESULTADO Test2Juego.txt : false

RESULTADO Test3Juego.txt : true

La imagen nos muestra la salida por consola de ejecutar la aplicación