FACULTAD DE MATEMÁTICAS

Trabajo fin de Grado

Grado en Matemáticas

Desarrollo de una librería Haskell sobre árboles de decisión

Realizado por
Carlos García Sancho

Dirigido por
Francisco Jesús Martín Mateos

Departamento
Ciencias de la Computación
e Inteligencia Artificial

Sevilla, Diciembre de 2021

Abstract

A Decision Tree is a supervised learning technique used for prediction tasks, which stands out for being able to produce interpretable classification and regression models. This project includes an overview of some Machine Learning, specifically Decision Trees, concepts and algorithms. The aim of this work is to obtain a clear and readable implementation of algorithms C4.5 and CART, pruning and a simple Random Forest in the functional programming language Haskell.

Agradecimientos

A mis padres, Juan Carlos y Nativi, por todo lo que ha hecho que esté aquí ahora; a Ali, mi familia y amigos, por los buenos momentos; a Olga, por ser mi apoyo (aun detrás de la pantalla).

Índice general

Índice de figuras

Índice de código

CAPÍTULO 1

Introducción

En este capítulo se introducirán en primer lugar los conceptos de Aprendizaje Automático Supervisado, modelos predictivos y Árboles de Decisión. A continuación, se expondrán algunos algoritmos de construcción de árboles de decisión: ID3, C4.5 y CART. Posteriormente, se describirán algunos métodos de poda. Por último, se tratará el concepto de Aprendizaje Combinado, algunos métodos y su aplicación a los árboles de decisión en *Random Forest*.

1.1– Introducción al aprendizaje automático supervisado

La **Inteligencia Artificial** es definida por Bellman en [2] como "[La automatización de] actividades asociadas al pensamiento humano, tales como toma de decisiones, resolución de problemas, aprendizaje...". El **Aprendizaje Automático** (*Machine Learning*) es un campo de la Inteligencia Artificial. Pero, ¿en qué consiste el aprendizaje? Según la definición dada por Tom Mitchell en [20], un programa aprende de la experiencia E con respecto a una tarea T y una medida de desempeño P, si su desempeño en T medido por P mejora con la experiencia E.

Por tanto, las secciones sucesivas se centrarán en describir algunas técnicas y conceptos relativos a la resolución de problemas mediante Aprendizaje Automático, en particular, mediante Árboles de Decisión.

A continuación, se indicará qué es un modelo y cómo se pueden aplicar estas técnicas para su obtención.

1.1.1. Modelos

En el contexto de un suceso o problema que se quiere resolver, analizar, explicar, simular o simplemente entender, surge el concepto de **modelo**.

En [1] se encuentra la siguiente definición: "Un modelo constituye una representación abstracta de un cierto aspecto de la realidad, y tiene una estructura que está formada por los elementos que caracterizan el aspecto de la realidad modelada y por las relaciones entre estos elementos".

Para representar la información de una observación del suceso modelado se seleccionan una serie de características o variables medibles y se representa dicha observación mediante sus valores. Estas características pueden ser continuas o discretas.

Este trabajo se centrará en un caso particular de modelo: el modelo predictivo. En él, se intenta predecir una o varias características de un suceso a partir de una serie de características dadas. Estos, a su vez, se clasifican en modelos de clasificación, si la característica a predecir toma valores discretos, y modelos de regresión, si toma valores continuos (i.e. un infinito número de valores).

Para obtener modelos predictivos, se puede hacer uso de técnicas de Aprendizaje Supervisado, un tipo de Aprendizaje Automático que se describirá a continuación.

1.1.2. Aprendizaje Supervisado

Según [21], la tarea del **Aprendizaje Supervisado** consiste en, dado un conjunto de N observaciones formadas por pares

$$(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)$$

de entradas-salidas donde cada y_j está generada por una función $y = f(x)$, encontrar una función h que aproxime a la verdadera función f. Estas observaciones se llamarán ejemplos.

Nótese que x e y pueden ser vectores formados por valores numéricos y/o categóricos. La función h es una hipótesis. El aprendizaje consiste en una búsqueda por el espacio de posibles funciones hipótesis. La hipótesis aprendida será el modelo predictivo dado por el Aprendizaje Supervisado.

Dicho de otra forma, las técnicas de Aprendizaje Automático supervisado proporcionan modelos infiriendo una función (relación) que asocia entradas y salidas a partir de un conjunto de valores de entrada para los que se conoce la salida. Para ello se considera un conjunto de ejemplos de los cuáles se conocen los valores de la característica que queremos predecir. Estos serán la salida del modelo, mientras que las entradas serán los valores de las características que representan a los ejemplos.

El objetivo de estas técnicas es obtener modelos que clasifiquen correctamente no solo los ejemplos del conjunto considerado, sino también otros cuya salida sea desconocida.

Algunos de los modelos más comunes provenientes de estas técnicas son las redes neuronales, k-vecinos más cercanos (KNN) o los árboles de decisión. A partir de ahora, el trabajo se centrará en este último.

1.2– Árboles de Decisión

Un **árbol de decisión** es un modelo predictivo basado en la estructura de árbol dirigido dada por la teoría de grafos. Están formados por nodos internos, a los que se llama nodos decisión, y nodos hoja, llamados nodos respuesta.

Destacan por diversas razones: su simpleza, capacidad de ser interpretable y de ser representado de forma visual. La capacidad de ser interpretable se refiere a la posibilidad de entender el razonamiento que hay detrás de una decisión. A pesar de esto, puede llegar a modelar sucesos y resolver problemas complejos. Estas características hacen que su uso esté muy extendido.

Se supone que tenemos un conjunto de N ejemplos $S = \{e_1, ..., e_N\}$ de un suceso representados por un conjunto de características que se llamarán atributos. Se $e_i = (x_{i1}, ..., x_{ik})$, donde x_{ij} es el valor que toma el atributo j en el ejemplo i. Para cada uno es conocido el valor de la característica que se quiere predecir, llamada clasificación, c_i. A este conjunto S se le llamará conjunto de entrenamiento.

En cada nodo decisión se hace una pregunta sobre el valor de un atributo. Dependiendo de la respuesta, se toma una de las ramas que salen del nodo. Cada nodo respuesta está asociado a una clasificación. Al evaluar un árbol sobre un ejemplo, se seguirá el camino indicado por las respuestas a las preguntas de los nodos de decisión sobre el ejemplo, y se devolverá como salida la clasificación del nodo respuesta que corresponda.

Se debe tener en cuenta que la construcción de un árbol de decisión no es única. Según el orden en el que se consideren los atributos al hacer las preguntas, se obtendrán diferentes árboles. Además, no siempre se puede construir un árbol que clasifique correctamente todos los ejemplos de un conjunto. Por tanto, se debe definir una función o medida de desempeño sobre los árboles para poder compararlos y decidir cuál se elige. Una posible medida es el porcentaje de acierto sobre un conjunto de ejemplos dado, no necesariamente pertenecientes al conjunto de entrenamiento.

Otra cuestión importante es la forma en que se construye el árbol de decisión. Una opción, motivada por el principio de la navaja de Occam según el cuál el modelo más simple es el mejor mientras sea consistente con los datos, consiste en mantener la menor profundidad posible en el árbol generado, que puede indicar cómo de simple es. En general, modelos más complejos se comportarán mejor en el conjunto de entrenamiento usado para crearlo pero generalizarán peor a nuevos ejemplos.

Para la construcción de árboles existen diversos algoritmos que toman como entrada un conjunto de ejemplos clasificados y devuelven un árbol de decisión. Estos algoritmos difieren en la clase de atributos que pueden tratar (continuos o discretos), en las técnicas utilizadas para decidir que atributos usar en los nodos de decisión, en el uso de técnicas que mejoran las predicciones para ejemplos no clasificados, etc.

A continuación, se presentarán algunos algoritmos de construcción de árboles de decisión.

1.2.1. Algoritmo ID3

Desarrollado en [23] por Quinlan, es el algoritmo más utilizado y en el que se basan gran parte de los posteriores.

Se supone que tanto los atributos como la clasificación toman valores discretos y que para todos los ejemplos considerados se conoce el valor de cada atributo y clasificación. Este algoritmo comienza con la pregunta ¿qué atributo debería ocupar el nodo raíz? Para responderla, se podría considerar el atributo que mejor clasifique el conjunto de datos de entrenamiento por sí mismo. Pero, ¿cómo se compara qué atributo clasifica mejor? Esto necesita ser medido de alguna forma. ID3 hace uso de la Teoría de la Información desarrollada por Shannon en [28], introduciendo el concepto de Entropía de Shannon.

Sean S un conjunto de ejemplos, $\{1, ..., C\}$ las distintas clasificaciones, y p_i la proporción de elementos de la clasificación i que hay en S.

Se define la **entropía de Shannon** de S como

$$E(S) = \sum_{i=1}^{C} -p_i \cdot log_2(p_i)$$

La entropía mide el grado de incertidumbre o desorden de un conjunto con respecto a una clasificación. Así $E(S) = 0$ si todos los elementos del conjunto clasifican igual (totalmente ordenado) y $E(S) = 1$ si el conjunto tiene el mismo número de elementos de cada clasificación (totalmente desordenado).

Haciendo uso de esta función, se define la **ganancia de información** $G(S, A)$ de un atributo A asociada a un conjunto de ejemplos S como

$$G(S, A) = E(S) - \sum_{v \in Valores(A)} (\frac{|S_v|}{|S|}) \cdot E(S_v)$$

donde $Valores(A)$ es el conjunto de posibles valores que toma el atributo A y $S_v = \{s \in S | A(s) = v\}$, i.e. el subconjunto de S para el cuál A toma valor v.

La ganancia de información es la reducción esperada de entropía causada por el conocimiento del valor del atributo A. Dicho de otra forma, es la información sobre la clasificación de un conjunto de ejemplos obtenida al conocer el valor del atributo A.

Respondiendo a la pregunta ¿qué atributo clasifica mejor?, será aquel con mayor ganancia de información para el conjunto de ejemplos considerado. Una vez seleccionado el atributo para el nodo decisión, se considera una rama para cada posible valor del atributo. Bajo esa rama, se considera el árbol dado por el algoritmo aplicado a los ejemplos cuyo atributo seleccionado toma dicho valor. No se podrá volver a considerar un atributo que ya ha aparecido en el árbol.

La condición de parada del algoritmo usual es la siguiente: si la proporción de una clasificación en un nodo supera un umbral fijado, se detiene la construcción del árbol en esa rama y se considera un nodo respuesta con dicha clasificación.

Recapitulando, el algoritmo ID3 recibe como entrada un conjunto de ejemplos descritos mediante una serie de pares atributos-valor y una clasificación. ID3 devuelve como salida un árbol de decisión que clasifica los ejemplos en la clase a la que pertenecen.

Escrito en pseudo-código:

ID3$(Ejemplos, Clasificacion, Atributos)$:

- Crear Nodo raíz para el árbol
- **Si** $Ejemplos$ cumple condición de parada:
  - **Entonces** devolver un único nodo respuesta con el valor más común de $Clasificacion$ en $Ejemplos$.
- **Si** $Atributos$ está vacía:
  - **Entonces:** devolver un único nodo respuesta con el valor más común de $Clasificacion$ en $Ejemplos$.
- **En otro caso:**
  - $A \leftarrow$ el atributo de $Atributos$ que tenga la mayor ganancia de información para $Ejemplos$
  - Atributo para nodo decisión $\leftarrow A$
  - **Para cada** valor v_i de A:
    - Añadir nueva rama debajo de raíz correspondiente a $A = v_i$
    - $Ejemplos|_{v_i} \leftarrow$ ejemplos que toman el valor v_i para A
    - **Si** $Ejemplos|_{v_i}$ vacío:
      - **Entonces**, debajo de esta rama añadir un nodo respuesta con el valor más común de $Clasificacion$ en $Ejemplos$.
    - **En otro caso**:
      - **Entonces** debajo de esta rama añadir **ID3**$(Ejemplos|_{v_i}, Clasificacion, Atributos \setminus \{A\})$
- **Fin**
- **Devolver** árbol

Entre los problemas y limitaciones del ID3 se encuentra el hecho de que no siempre proporciona el mejor árbol posible, ya que optimiza cada decisión de qué atributo seleccionar por separado. Puede existir otro orden de preguntas que, aun no optimizando la ganancia de información en cada paso, obtenga un mejor resultado.

1.2.2. Algoritmo C4.5

Es una extensión del algoritmo ID3 desarrollada por el mismo Quinlan en [25]. Tal y como sucede con ID3, **C4.5** solo puede tratar problemas de clasificación discreta, pero introduce algunas mejoras.

Una de las limitaciones de ID3 es que favorece la selección de los atributos que toman mayor número de valores, debido a que dividir el conjunto de ejemplos en más subconjuntos aumenta la ganancia de información. Para resolver este problema, se hace uso de la **ganancia de información normalizada**. Esta se define como

$$G_{norm}(S, A) = \frac{G(S, A)}{-\sum_{v \in Valores(A)} p_v \cdot log_2(p_v)}$$

donde p_v es la proporción de ejemplos de S en los que A toma el valor v.

A mayor número de valores, mayor será el término divisor. De esta forma, se evita que los atributos que toman muchos valores sean beneficiados. En cada nodo se tomará el atributo que tenga mayor ganancia de información normalizada.

A diferencia de ID3, C4.5 introduce una forma de tratar atributos continuos. Esta consiste en dividir el intervalo de valores que toma el atributo a partir de un umbral. Se crean dos clases, una para los valores menores o iguales que el umbral y otra para los mayores. De esta forma, se puede tratar como un atributo discreto.

Ahora la pregunta es, ¿cómo se encuentra el umbral? Primero, se ordenan todos los valores que toma el atributo continuo. Posteriormente, se consideran todos los puntos medios de cada par de valores consecutivos. Para cada valor medio u, se consideran dos clases: los ejemplos con valor del atributo $\leq u$ y aquellos tal que este sea $> u$.

Otra novedad es el tratamiento de ejemplos con algún atributo cuyo valor es desconocido. La solución pasa por estimar el valor del atributo en base al resto de ejemplos cuyo valor sí es conocido. Se puede tomar dicho valor como aquel más común en el nodo actual, el más común en el nodo entre los ejemplos con igual clasificación o asignar probabilidades a partir de las frecuencias con las que aparece cada valor. Posteriormente, se toma como valor de la ganancia de información para los atributos

$$G^*_{norm}(S, A) = F \cdot G_{norm}(S, A)$$

siendo F la proporción de ejemplos en S cuyo valor de A es conocido.

1.2.3. Algoritmo CART

El algoritmo CART, desarrollado por Breiman en [3], es nombrado así por las siglas en inglés de *Classification and Regression Trees* (árboles de clasificación y regresión). La

principal diferencia entre este algoritmo y los introducidos anteriormente es que, en este caso, se pueden tratar tanto problemas de clasificación como de regresión. CART produce árboles de decisión binarios.

Los atributos del problema pueden ser discretos o continuos. En el caso de los atributos discretos, se codifican las clases numéricamente de forma que puedan ser tratados como continuos. Para los atributos continuos, se actúa de la misma forma que en el C4.5. Se busca un valor umbral u que separa los ejemplos en dos clases: ($\leq u$) y ($> u$). Este umbral se busca entre los puntos medios de los valores (ordenados) que toma el atributo continuo. Para cada valor umbral, una función con cada valor evalúa cómo de bien clasifica los ejemplos y se elige el que mejor realice dicha tarea.

La otra principal diferencia es la función utilizada para evaluar qué atributo se elige en cada nodo:

Para los problemas de clasificación, donde la variable a predecir es discreta, se define el **Índice de Gini** de un conjunto de ejemplos S como

$$Gini(S) = 1 - \sum_{i=1}^{C} (p_i)^2$$

donde p_i es la proporción de ejemplos con clasificación c_i y C el número de posibles clasificaciones.

Para los problemas de regresión, donde el atributo objetivo es continuo, se define el **Error Cuadrático Medio** de un conjunto de ejemplos S como

$$ECM(S) = \sum_{i=1}^{N} (y_i - y_i')^2$$

donde N es el número de ejemplos de S, y_i el valor de la predicción del atributo objetivo para el ejemplo i e y_i' el valor real del atributo objetivo para el ejemplo i.

El algoritmo funciona de manera similar a los anteriores. En cada nodo decisión se elige el atributo que mejor clasifica los ejemplos considerados. Se realiza la división de los ejemplos evaluando la condición sobre el atributo elegido, creando una rama para cada clase descrita previamente según el umbral.

Si el problema es de clasificación, se buscará minimizar el Índice de Gini tanto en la selección del umbral para cada atributo como en la selección del propio atributo. Si es de regresión, se tomará como predicción del atributo objetivo la media aritmética de los valores que toman los ejemplos para cada clase. De igual forma, se buscará aquel umbral y atributo que minimice el ECM. Esta función a minimizar será notada como f en el pseudo-código que se mostrará a continuación.

Como condición de parada se considera que la proporción de una clasificación de los

ejemplos de un nodo sea mayor que un umbral fijado (clasificación) o el ECM sea menor que un umbral fijado (regresión).

Por último, se define la predicción de un conjunto de ejemplos como: el valor más común del atributo objetivo en el caso de clasificación; la media aritmética de los valores del atributo objetivo de los ejemplos del conjunto en la regresión.

En pseudo-código:

CART($Ejemplos$, $Atributo\_objetivo$, $Atributos$) :

- Crear Nodo raíz para el árbol
- **Si** $Ejemplos$ cumple condición de parada :
  - **Entonces** devolver un único nodo respuesta con la predicción de $Ejemplos$
- **Si** $Atributos$ está vacía:
  - **Entonces** devolver un único nodo respuesta con la predicción de $Ejemplos$
- **En otro caso:**
  - **Para cada** *atributo* de $Atributos$:
    - $Valores \leftarrow$ lista de valores que toma *atributo* en $Ejemplos$ ordenados de menor a mayor.
    - $Umbrales \leftarrow$ puntos medios de cada pareja consecutiva de $Valores$.
    - **Para cada** *umbral* en $Umbrales$:
      - Considerar las clases ($\leq umbral$), ($> umbral$) en $Ejemplos$.
      - Evaluar la función f para dichas clases.
    - $U \leftarrow umbral$ de $Umbrales$ que minimice f.
    - Considerar *atributo* como un atributo discreto con las clases descritas anteriormente.
  - $A \leftarrow$ el *atributo* de $Atributos$ que tenga menor f para $Ejemplos$.
  - Atributo para nodo decisión $\leftarrow A$
  - **Para cada** clase v_i de A:
    - Añadir nueva rama debajo de raíz correspondiente a $A \in v_i$
    - $Ejemplos|_{v_i} \leftarrow$ Ejemplos que pertenecen a v_i para A
    - **Si** $Ejemplos|_{v_i}$ vacío:
      - **Entonces** debajo de esta rama añadir nodo hoja con predicción de $Ejemplos$.
    - **En otro caso:**
      - **Entonces** debajo de esta rama añadir **CART**($Ejemplos|_{v_i}$, $Atributo\_objetivo$, $Atributos \setminus v_i$)
- **Fin**
- **Devolver** árbol

1.2.4. Sobreajuste

Como se comentó previamente, los modelos que buscan resolver un problema de predicción deben ser capaces de generalizar a partir del conjunto de ejemplos de entrenamiento. Esto no siempre ocurre, de hecho, este problema se presenta de forma común en la práctica.

En primer lugar, como medida del error cometido por un árbol de decisión en un conjunto de ejemplos, se puede considerar, entre otras funciones, el número de ejemplos del conjunto que se clasifican incorrectamente o la media del error cuadrático medio de las predicciones de los ejemplos, según la naturaleza del problema.

Se dice que un árbol t está **sobreajustado** al conjunto de entrenamiento si existe otro árbol t' tal que t tiene un menor error sobre el conjunto de entrenamiento, pero t' tiene un menor error que t sobre el conjunto completo de posibles ejemplos.

Al aplicar algoritmos de construcción de árboles de decisión como los vistos previamente, es común ajustarse demasiado a los ejemplos del conjunto de entrenamiento y que el árbol no realice una predicción correcta con otros ejemplos. Esto puede ser debido a que no haya suficientes ejemplos en el conjunto de entrenamiento o a que exista ruido en los mismos. Especialmente, cuando en algún nodo respuesta haya un número muy pequeño de ejemplos.

Entrenamiento y validación

Poder conocer si un modelo generaliza correctamente y cuál será su rendimiento con nuevos ejemplos no es adecuado considerar el conjunto de entrenamiento con el que ha sido construido. Para resolver este problema se considera la separación del conjunto de todos los ejemplos en dos subconjuntos diferentes: un conjunto de entrenamiento, tal y como hemos visto hasta ahora, usado para construir el modelo; y un conjunto de validación, usado para evaluar la precisión del modelo sobre ejemplos nuevos. Este es uno de los enfoques más comunes en el Aprendizaje Supervisado y en particular en la construcción de árboles.

Poda

Los algoritmos de construcción de árboles pueden proporcionar árboles sobreajustados al conjunto de entrenamiento. Para abordar este problema, se considera el concepto de poda para simplificar el árbol. Siguiendo el principio de la navaja de Occam, se prefieren árboles simples a complejos ya que generalizarán mejor.

Se puede diferenciar entre prepoda y postpoda. La prepoda ocurre durante la construcción del árbol y consiste en detener la subdivisión del árbol según un criterio dado; mientras que la postpoda ocurre tras la construcción del árbol, reemplazando un subárbol por una hoja o una de sus ramas.

Se verán a continuación algunos métodos con más detalle.

Prepoda

La prepoda consiste en detener el crecimiento del árbol, en base a distintos criterios, antes de que clasifique el conjunto de entrenamiento lo mejor posible. Se presentan algunos de esos criterios:

1. Mínimo de ejemplos para dividir un nodo. Se fija un número de ejemplos necesarios en los nodos para seguir subdividiendo el árbol. Si se llega a un nodo con un número menor de ejemplos, se pondrá un nodo respuesta con la clasificación más representada o la media aritmética del atributo objetivo, según sea un problema de clasificación o regresión.
2. Máxima profundidad del árbol. Si se llega a esa profundidad, se detiene el crecimiento de la rama.
3. Máximo número de nodos respuesta. Si se alcanza este límite, se detiene el desarrollo del árbol.
4. Máximo número de atributos para considerar en cada ramificación. Se seleccionan aleatoriamente. Esta técnica es útil cuando hay un gran número de atributos.

Postpoda de error reducido

A partir de un árbol posiblemente sobreajustado, se considera la posibilidad de sustituir algún subárbol por un nodo respuesta, al que se le asigna la clasificación más común entre los ejemplos asociadas a dicho nodo. Los subárboles serán sustituidos si el árbol resultante obtiene un mejor resultado sobre el conjunto de validación que el árbol original. De esta forma, las ramas que sean creadas a partir de particularidades del conjunto de entrenamiento serán probablemente podadas, ya que es improbable que también estén presentes en el conjunto de validación.

Se itera por todos los nodos decisión, comenzando desde abajo hacia arriba. El procedimiento continúa mientras que no se encuentre una poda que mejore el rendimiento del árbol.

Para este método de poda es necesario un conjunto de validación y por tanto, disponer de un número suficiente de ejemplos.

Postpoda de error pesimista

Propuesto por Quinlan en [25] y consultado en [22]. Cuando no se dispone de un conjunto de validación, es optimista usar el conjunto de entrenamiento para probar la tasa de error, ya que el árbol puede estar adaptado a él. Al usar unos ejemplos diferentes el error

podría aumentar drásticamente. Por tanto, se hace uso de la corrección de continuidad estadística de la distribución binomial. De esta manera se obtiene un error más realista. Se consideran las siguientes fórmulas

$$n'(t) = Error(t) + \frac{1}{2}$$
$$n'(T_t) = Error(T_t) + (\frac{N_T}{2})$$

donde

$n'(t)$ el error para el nodo t

$n'(T_t)$ el error para el subárbol T_t

$Error(t)$ el número de ejemplos mal clasificados en el nodo t

$Error(T_t)$ el número de ejemplos mal clasificados en el subárbol T_t

$\frac{1}{2}$ es la constante que penaliza la complejidad del árbol

Postpoda por mínimo error de complejidad

Este método se conoce como el algoritmo de poda de CART. Dado un árbol, se considera la siguiente secuencia $\{T_0, T_1, ..., T_n\}$ de árboles cada vez más podados, donde T_0 es el árbol original. Luego, se selecciona aquel árbol con la tasa de error más baja en un conjunto de validación.

A partir de T_i, el árbol T_{i+1} será el que se obtiene al recortar todos los nodos que tienen el aumento más bajo en la tasa de error por hoja cortada, notado

$$\alpha = \frac{R(t) - R(T_t)}{N_T - 1}$$

siendo

$R(t)$ tasa de error si el nodo se recorta y se convierte en un nodo respuesta.

$R(T_t)$ tasa de error si el nodo no se ha eliminado, calculado como el promedio de las tasas de error en las hojas ponderadas por el número de ejemplos en cada una.

N_T el número de hojas.

Se realiza el método de la siguiente forma. Se calcula α para cada nodo no terminal excepto la raíz en T_i. Se podan aquellos nodos con menor α, obteniendo así T_{i+1}. En pseudocódigo,

Poda Mín Error Complejidad(T) :

- $T_0 \leftarrow T$
- $i \leftarrow 0$.
- **Mientras** $T_i \neq$ árbol formado por la Raíz de T:
  - Calcular α para cada nodo no terminal (excepto raíz) de T_i.
  - $N \leftarrow$ Nodo con menor α. Si hay dos nodos con el mismo valor, se toman ambos.
  - $T_{i+1} \leftarrow$ Árbol resultante de sustituir el nodo N de T_i con menor α por un nodo respuesta.
  - $E_i \leftarrow$ Error del árbol T_i en un conjunto de validación.
  - $i \leftarrow i + 1$.
- **Fin**
- **Devolver** árbol con menor E_i

1.2.5. Validación cruzada

Como se ha visto, una de las limitaciones en estos problemas es el número de ejemplos disponibles. Cuando este es demasiado pequeño para separar en entrenamiento y validación, se puede considerar el método de validación cruzada con k pliegues para obtener una estimación del error.

En este método, se dividen los m ejemplos disponibles en k subconjuntos disjuntos, de tamaño m/k. El proceso de validación cruzada se realiza k veces, en cada una seleccionando un subconjunto distinto como conjunto de validación y los $k-1$ restantes como conjunto de entrenamiento. Por tanto, se obtienen k tasas de error de k modelos sobre k conjuntos de validación diferentes. Computando la media de estos errores se obtiene una mejor estimación del error para futuros ejemplos.

1.2.6. Limitaciones

Entre las principales limitaciones de estos algoritmos de construcción de árboles de decisión se encuentra la naturaleza de los mismos. Son algoritmos voraces, que en cada paso eligen la opción óptima (en este caso, búsqueda entre los posibles atributos para cada nodo decisión), y no se realizan retrocesos. De esta forma pueden caer en mínimos locales y no llegar al global.

1.3– Métodos combinados

Los métodos **combinados** o de *ensemble*, descritos en [7], son una técnica de Aprendizaje Automático que consiste en combinar diversos modelos para producir un nuevo

modelo predictivo. El objetivo es que dicho modelo tenga mejor rendimiento que aquellos que lo componen individualmente. Gracias a este método se pueden conseguir modelos más potentes con los que se puedan tratar problemas más complejos.

Un ejemplo es el voto por mayoría. En un problema de clasificación, se evaluará un conjunto de modelos de predicción para un ejemplo y se tomará como predicción la clasificación más repetida.

En [9] se muestra lo siguiente, que podría considerarse el fundamento por el cuál esta técnica produce resultados satisfactorios. Supongamos un problema de clasificación binaria. Sea p la probabilidad de que un modelo clasificador haga una predicción incorrecta y supongamos que los errores ocurren de forma independiente o aleatoria (i.e, cuando ocurre un error de clasificación, la probabilidad de que se produzca un error en las siguientes predicciones no cambia). La probabilidad de que el modelo produzca k clasificaciones incorrectas en N intentos es

$$\binom{N}{k} p^k (1-p)^{N-k}$$

Si se considera un método combinado de votación por mayoría con N modelos diferentes, todos con la misma probabilidad de error, la probabilidad de k errores en N intentos será

$$\sum_{k>N/2}^{N} \binom{N}{k} p^k (1-p)^{N-k}$$

Si $p < 1/2$, la probabilidad tiende a 0 cuando $N \to \infty$.

Según [9], se puede demostrar también para problemas de clasificación con más de dos clases. De hecho, una condición necesaria y suficiente para que un clasificador por método combinado sea más preciso que cualquiera de los que lo componen es que estos sean suficientemente precisos y diversos. Un clasificador es suficientemente preciso si predice mejor que un clasificador aleatorio; dos clasificadores son diversos si no producen las mismas predicciones para todos los ejemplos.

De igual forma que con voto por mayoría en los problemas de clasificación, se pueden tratar problemas de regresión considerando una media de las predicciones dadas de los árboles.

A continuación se revisarán algunas de las técnicas más destacadas para crear modelos combinados a partir de familias de modelos.

Bagging

El ***Bagging***, o agregación *Bootstrap*, es un método para crear un modelo combinado a partir de una familia de modelos y un conjunto de entrenamiento.

De acuerdo con [26], dado un conjunto de entrenamiento con N ejemplos, se generan m conjuntos de entrenamiento tomando n' elementos del conjunto original cada uno con probabilidad $1/N$ y con remplazamiento. A partir de cada uno de estos nuevos conjuntos de entrenamiento se construye un modelo. Como predicción del modelo final se toma la mayoritaria entre los m modelos, en el caso de clasificación, o la media, en el de regresión.

Boosting

Se dice que un modelo es: **débil** (*weak learner*) si sólo se ajusta bien al conjunto de ejemplos con el que ha sido entrenado y **fuerte** (*strong learner*) si además generaliza bien a nuevos ejemplos. Según [4], un algoritmo de ***Boosting*** hace uso de modelos débiles para obtener un modelo fuerte. A continuación se revisarán dos de los algoritmos más comunes de *Boosting*: *Arcing* y *AdaBoost*.

En general, la construcción de los modelos ocurre de forma secuencial. En cada iteración, se construye un modelo seleccionando el conjunto de entrenamiento en función del rendimiento de los modelos creados en las iteraciones anteriores. Se busca que el nuevo modelo prediga correctamente los ejemplos que los modelos anteriores clasifican incorrectamente y, así, mejorar el rendimiento del modelo combinado en dichos ejemplos.

En [5], Brieman define el ***Arcing***. Este método, al igual que el *Bagging*, toma N ejemplos con remplazamiento del conjunto de entrenamiento original pero modificando las probabilidades. Se asocia un peso a cada ejemplo del conjunto de entrenamiento, que podemos identificar con la probabilidad de ser elegido. Al comienzo todas las probabilidades serán $1/N$. En cada iteración se crea un modelo y se modifican los pesos, aumentando los de aquellos ejemplos que sean clasificados incorrectamente, siendo más probable que el nuevo modelo los clasifique bien.

El algoritmo ***AdaBoost*** (*Adaptative Boosting*), presentado en [8], es un método de *Boosting* que actúa de la siguiente forma. Sea w_x^t el peso del ejemplo x en la iteración t. El peso puede ser interpretado como la probabilidad de seleccionar un ejemplo para el conjunto de entrenamiento de una iteración. Se inicializa $w_x^1 = 1/N \ \forall x$ del conjunto de entrenamiento, con N el número total de ejemplos. En cada iteración $t = 1, 2, ..., T$ se construye un modelo M^t teniendo en cuenta los pesos w_x^t. El error ϵ^t consiste en la suma de los pesos de los ejemplos que clasifica incorrectamente M^t. Si ϵ^t es mayor que $1/2$ se descarta el modelo y termina el proceso tomando los $t-1$ anteriores, o bien se reinicializan los pesos a $1/N$ y se continúa. Si ϵ^t es cero, termina el proceso tomando $M^1, ..., M^t$. En otro caso, se generan los pesos w^{t+1} multiplicando w^t por el factor $\beta^t = (1 - \epsilon^t)/\epsilon^t$ y normalizando para obtener $\sum_x w^{t+1} = 1$. El clasificador final se obtiene considerando los votos de todos los modelos, donde al voto de M^t se le da peso $\log(\beta^t)$.

Subespacios Aleatorios

Este método, consultado en [27], es similar al *Bagging*, pero considerando el conjunto de atributos en vez del conjunto de ejemplos. En cada iteración, se toma aleatoriamente un número dado de atributos. De esta forma, se entrenan diversos modelos con diversos subconjuntos de los atributos y estos pueden luego ser combinados.

1.3.1. Random Forest

Particularizando algunos de los métodos anteriormente descritos a los árboles de decisión se obtienen los métodos de ***Random Forest*** (Bosques Aleatorios).

Breiman en [6] define un modelo de ***Random Forest*** como un clasificador obtenido aplicando la técnica de voto por mayoría a partir de una colección de árboles de decisión (de clasificación) construidos según los métodos de *Bagging* y Subespacios Aleatorios simultáneamente.

Out-of-Bag error

El error de un modelo de Random Forest se puede estimar sin hacer uso de la validación cruzada, que puede ser costosa y para la que hace falta un elevado número de ejemplos. Cada árbol del modelo combinado ha sido creado con un subconjunto del conjunto de entrenamiento total. Por tanto, existen ejemplos que no han sido utilizados en su creación. Si cada ejemplo se elige con la misma probabilidad $1/N$ (N el número total de ejemplos), la probabilidad de no ser elegido en N selecciones con remplazamiento es de $(1 - 1/N)^N$ que converge a $1/e$. Por tanto, aproximadamente un tercio del conjunto de ejemplos no será seleccionado en cada subconjunto. A este conjunto restante se le denomina *Out-of-Bag*.

Se define el ***Out-of-Bag error*** de un ejemplo como el error promedio de aquellos árboles en cuya construcción este no ha sido utilizado.

CAPÍTULO 2

Diseño de la aplicación

En este capítulo se describirá la librería de construcción de árboles de decisión desarrollada para este proyecto. Se especificará qué librerías predefinidas se utilizan, la estructura de los ficheros de la librería creada y los tipos y funciones que la componen.

El código se desarrolla en HASKELL, un lenguaje de programación funcional puro, con evaluación perezosa y con fuerte tipificación estática, i.e, el lenguaje controla que cada operación solo se realice sobre aquellos datos que sean del tipo para el que está definida. Aunque HASKELL calcula los tipos de una expresión por un proceso llamado inferencia de tipos antes de evaluarla, se especificará el tipo de cada función que compone la librería.

Será necesario crear instancias de los nuevos tipos en las clases predefinidas de `Haskell`: `Eq` y `Show`.

2.1– Librerías

Se han usado las siguientes librerías predefinidas de HASKELL: `Data.List` [10], `Data.Maybe` [11], `Data.Either` [12] y `System.Random` [13].

2.2– Estructura

A continuación se describe la estructura de la librería, enumerando sus archivos.

1. `tipos.hs`
2. `utils.hs`
3. `entropia.hs`

4. `ganancianormalizada.hs`
5. `gini.hs`
6. `ecm.hs`
7. `discretizarcontinuo.hs`
8. `mejoratributo.hs`
9. `parada.hs`
10. `construirmodelos.hs`
11. `evaluar.hs`
12. `randomforest.hs`

Su contenido se especificará en las siguientes secciones.

2.3– Tipos

En primer lugar, se definen los tipos `Discreto` y `Continuo`. Los atributos discretos constan de un nombre y una lista de posibles valores.

Código 2.1: Tipo Discreto

```
data Discreto =
    D { nombreD :: String, posiblesvaloresD :: [String] }
       deriving Eq

instance Show Discreto where
  show = nombreD
```

Código 2.1: Tipo Discreto

Por otro lado, los continuos constan de un nombre, un rango de valores que puede tomar y un umbral que se define con el tipo `Maybe`. Este se inicializa como `Nothing` y posteriormente será modificado. No se tendrá en cuenta el umbral en la definición de igualdad entre atributos continuos. De esta manera se podrá modificar en cada paso el umbral sin tener que definir un nuevo atributo. La razón es que HASKELL es un lenguaje funcional puro, no es un lenguaje orientado a objetos. Por tanto, trata con funciones puras en las que no pueden existir variables que puedan cambiar de estado.

Código 2.2: Tipo Continuo

```
data Continuo =
     C { nombreC :: String, rango :: (Double,Double), umbral :: Maybe Double }

instance Eq Continuo where
     continuo == continuo' =
              nombreC continuo == nombreC continuo'
              && rango continuo == rango continuo'

instance Show Continuo where
  show = nombreC
```

Código 2.2: Tipo Continuo

Haciendo uso del tipo `Either` se definen el tipo `Atributo` y el tipo `ValorAtrib`, que representa el valor de un atributo, `String` o `Int` según la naturaleza del atributo. La elección de `Either` viene motivada por la restricción de tipos de HASKELL. Al ser un lenguaje fuertemente tipado, no permite crear listas de elementos de diversos tipos. Por tanto, es necesario crear un tipo que los agrupe.

Código 2.3: Tipo Atributo

```
type ValorAtrib = Either String Double
```

Código 2.3: Tipo Atributo

Se define el tipo `Ejemplo`, compuesto por un par: en el primer elemento se almacena una lista con pares (atributo,valor); en el segundo, el par (atributo objetivo,valor).

Código 2.4: Tipo Ejemplo

```
type Ejemplo = ( [(Atributo, ValorAtrib)], (Atributo, ValorAtrib) )
```

Código 2.4: Tipo Ejemplo

Se define el tipo `Arbol`, formado por Nodos Hoja que contienen una predicción (`ValorAtrib`) y por Nodos Raíz, que contienen un atributo y una función de los valores del atributo en otro árbol. Esta función `hijo` representa las ramas del nodo.

Código 2.5: Tipo Árbol

```
data Arbol = Hoja ValorAtrib
     | Nodo { atrib::Atributo, hijo::String -> Arbol }
```

Código 2.5: Tipo Árbol

Se añaden algunas funciones útiles para la representación de los árboles y se crea una instancia del tipo `Arbol` para la clase `Show`. Se añaden como posibles valores de un atributo continuo las clases "≤" y ">".

Código 2.6: Instancia del tipo `Arbol` para la clase `Show`

```
instance Show Arbol where
     show x = showTree x ""

showTree :: Arbol -> ShowS
showTree (Hoja x)        = shows x
showTree (Nodo atrib hijo) = ('<':).shows atrib.(showUmbral atrib ++).("|\n"++).
    showList [(hijo a,a) | a <- posiblesValores atrib].('>':)

nombre :: Atributo -> String
nombre (Left atributo) = nombreD atributo
nombre (Right atributo) = nombreC atributo

showUmbral :: Atributo -> String
showUmbral (Left atrib) = ""
showUmbral (Right atrib) =
    let u = umbral atrib
    in if isNothing u then ""
    else " {Umbral: " ++ (show $ fromJust $ u) ++ "}"

posiblesValores :: Atributo -> [String]
posiblesValores (Left atributo) = posiblesvaloresD atributo
posiblesValores (Right atributo) = ["<=",">"]
```

Código 2.6: Instancia del tipo `Arbol` para la clase `Show`

Los árboles se muestran siguiendo la estructura de búsqueda en profundidad:

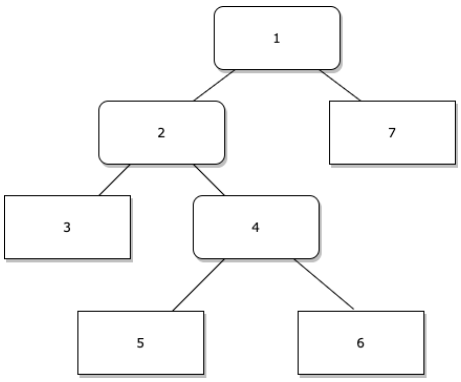

Figura 2.1: Búsqueda en profundidad

2.4– Funciones útiles

Se definen a continuación una serie de funciones que serán utilizadas posteriormente.

Código 2.7: Funciones útiles

```
aleft par = (Left $ fst par, Left $ snd par)
aright par = (Right $ fst par, Right $ snd par)

lengthDouble :: [a] -> Double
lengthDouble = fromIntegral.length

getL :: Either a b -> a
getL (Left x) = x

getR :: Either a b -> b
getR (Right x) = x

ordensindups :: (Ord a) => [a] -> [a]
ordensindups = map head . group . sort

maximo :: [(b,Double)] -> (b,Double) -> b
maximo [] y                = fst y
maximo (x:xs) y
          | (snd x) > (snd y) = maximo xs x
          | otherwise       = maximo xs y

ocurrencia :: (Eq a) => [a] -> a -> Double
ocurrencia [] a    = 0
ocurrencia (x:xs) a
          | a == x   = 1 + ocurrencia xs a
          | otherwise = ocurrencia xs a


elimina _ []              = []
elimina x (y:ys) | x == y = elimina x ys
                 | otherwise = y : elimina x ys

media :: [Double] -> Double
media xs = sum xs / lengthDouble xs

eliminaLista :: (Eq a) => [a] -> [a] -> [a]
eliminaLista [] ys = ys
eliminaLista _ [] = []
eliminaLista (x:xs) ys =
    elimina x (eliminaLista xs ys)

atributoObjetivo :: Ejemplo -> Atributo
atributoObjetivo = fst.snd

valorObjetivo :: Ejemplo -> ValorAtrib
valorObjetivo = snd.snd

valores :: Atributo -> [Ejemplo] -> [ValorAtrib]
valores _ [] = []
valores atributo (e:ejemplos) =
    [ snd x | x <- fst e, fst x == atributo ]
    ++
```

```
53      valores atributo ejemplos
54
55  valorAtributo :: Ejemplo -> Atributo -> ValorAtrib
56  valorAtributo ejemplo atributo =
57      snd $ head (filter (\x -> fst x == atributo) (fst ejemplo))
```

Código 2.7: Funciones útiles

Se define la función `evaluar`: toma una lista de ejemplos, un atributo y un valor de dicho atributo; devuelve otra lista con los ejemplos cuyo atributo toma dicho valor. En el caso de los atributos continuos, evalúa según sea mayor o menor que el umbral fijado.

Para ello, se define de forma separada para atributos discretos y continuos y se combinan haciendo uso de la función `either`. Esta función está definida en `Data.Either` y es de tipo :: $(a \to c) \to (b \to c) \to Either\ a\ b \to c$.

Código 2.8: Evaluar atributo

```
1   evaluarD :: [Ejemplo] -> String -> Discreto -> [Ejemplo]
2   evaluarD ejemplos valor atributo =
3       let atrib = Left atributo
4       in [x | x <- ejemplos, (getL $ valorAtributo x atrib) == valor ]
5
6   evaluarC :: [Ejemplo] -> String -> Continuo -> [Ejemplo]
7   evaluarC ejemplos valor atributo =
8       let u = fromJust $ umbral atributo
9           atrib = (Right atributo)
10      in if valor == "<="
11         then filter (\x -> (getR$valorAtributo x atrib) <= u) ejemplos
12         else if valor == ">"
13         then filter (\x -> (getR$valorAtributo x atrib) > u) ejemplos
14         else []
15
16  evaluar :: [Ejemplo] -> Atributo -> String -> [Ejemplo]
17  evaluar ejemplos atributo valor =
18      either (evaluarD ejemplos valor) (evaluarC ejemplos valor) atributo
```

Código 2.8: Evaluar atributo

2.5– Entropía

Se define la función `entropia` de una lista de ejemplos como:

Código 2.9: Entropia

```
entropia :: [Ejemplo] -> Double
entropia [] = 0
entropia ejemplos =
    let valores = posiblesValores $ atributoObjetivo $ head ejemplos
        n       = lengthDouble ejemplos
        f       = ocurrencia (map (getL.valorObjetivo) ejemplos)
    in foldl (\ac x -> let p = (f x) / n in if p>0 then ac - p * (logBase 2 p)
        else ac )
    0 valores
```

Código 2.9: Entropia

2.6– Ganancia de Información Normalizada

De igual forma que con `evaluar`, se distingue por casos y se hace uso de `either` para definir la ganancia de información normalizada `gananciaNorm`.

Código 2.10: Ganancia de Información

```
gananciaNorm :: [Ejemplo] -> Atributo -> Double
gananciaNorm [] _ = 0
gananciaNorm ejemplos atributo =
    either (gananciaNormD ejemplos) (gananciaNormC ejemplos) atributo
```

Código 2.10: Ganancia de Información

Atributos Discretos

Se define la función para la ganancia de información normalizada para un atributo discreto de la siguiente manera:

Código 2.11: Ganancia de Información (Discreto)

```
gananciaInformacionD ::[Ejemplo] -> Discreto -> Double
gananciaInformacionD [] _ = 0
gananciaInformacionD ejemplos atributo =
    let val      = posiblesvaloresD atributo
        ganancia = foldl (\ac x ->
            let sv = evaluarD ejemplos x atributo
            in ac + (entropia sv) * (lengthDouble sv) / (lengthDouble ejemplos)) 0
                 val
    in entropia ejemplos - ganancia

normaD :: [Ejemplo] -> Discreto -> Double
normaD [] _ = 1
normaD ejemplos atributo =
    let val = posiblesvaloresD atributo
        n  = lengthDouble ejemplos
```

```
15      in foldl (\ac x -> let pc = (lengthDouble $ evaluarD ejemplos x atributo) / n
16                in if pc > 0 then ac - pc * (logBase 2 pc) else ac) 0 val
17
18  gananciaNormD :: [Ejemplo] -> Discreto -> Double
19  gananciaNormD [] _ = 0
20  gananciaNormD ejemplos atributo =
21      let norm = normaD ejemplos atributo
22      in if norm /= 0 then gananciaInformacionD ejemplos atributo / normaD ejemplos
            atributo
23      else gananciaInformacionD ejemplos atributo
```

Código 2.11: Ganancia de Información (Discreto)

Atributos Continuos

En el caso continuo:

Código 2.12: Ganancia de Información (Continuo)

```
1   gananciaNormCUmbral :: [Ejemplo] -> Continuo -> Double -> Double
2   gananciaNormCUmbral [] _ _ = 0
3   gananciaNormCUmbral ejemplos atributo umbral =
4       let atrib = Right atributo
5           s1   = [ x | x <- ejemplos, (getR $ valorAtributo x atrib) <= umbral ]
6           s2   = [ x | x <- ejemplos, (getR $ valorAtributo x atrib) > umbral ]
7           n    = lengthDouble ejemplos
8           (gan,norm) = foldl (\(g,norm) s -> let p = (lengthDouble s) / n
9                       in ( g + (entropia s) * p,
10                          if p /= 0 then norm - p * (logBase 2 p) else norm) )
11                      (0,0) [s1,s2]
12      in if norm /= 0
13         then (entropia ejemplos - gan ) / norm
14         else entropia ejemplos - gan
15
16  gananciaNormC :: [Ejemplo] -> Continuo -> Double
17  gananciaNormC [] _ = 0
18  gananciaNormC ejemplos atributo =
19      let u = fromJust $ umbral atributo
20      in gananciaNormCUmbral ejemplos atributo u
```

Código 2.12: Ganancia de Información (Continuo)

Las funciones que toman como parámetro `umbral` serán necesarias para discretizar los atributos continuas.

2.7– Índice de Gini

Se define el índice de Gini de una lista de ejemplos. Nótese que esta función se usa en CART (clasificación), donde todos los atributos considerados son continuos. Por tanto,

no hay necesidad de definirlo para atributos discretos.

Es necesario definir una versión `giniAtributoUmbral` para la búsqueda del mejor umbral de un atributo continuo.

Código 2.13: Índice de Gini

```
gini :: [Ejemplo] -> Double
gini [] = 0
gini ejemplos =
    let n             = lengthDouble ejemplos
        clasificaciones = map (getL.valorObjetivo) ejemplos
        valores       = posiblesValores $ atributoObjetivo $ head ejemplos
    in 1 - foldl (\ac x -> let p = (ocurrencia clasificaciones x) / n
                  in ac + p^2) 0 valores

giniAtributo :: [Ejemplo] -> Atributo -> Double
giniAtributo ejemplos atributo =
    giniAtributoUmbral ejemplos (getR atributo) (fromJust $ umbral (getR $
        atributo))

giniAtributoUmbral :: [Ejemplo] -> Continuo -> Double -> Double
giniAtributoUmbral ejemplos atributo umbral =
    let atrib = Right atributo
        n     = lengthDouble ejemplos
        s1    = [ x | x <- ejemplos, (getR $ valorAtributo x atrib) <= umbral ]
        s2    = [ x | x <- ejemplos, (getR $ valorAtributo x atrib) > umbral ]
    in foldl (\ac s -> let p = lengthDouble s / n in ac + p * gini s) 0 [s1,s2]
```

Código 2.13: Índice de Gini

2.8– Error Cuadrático Medio

De igual forma que en `gini`, no es necesario definir el Error Cuadrático Medio `ecm` para atributos discretos y se define `ecmAtributoUmbral` para la búsqueda del umbral óptimo de un atributo continuo.

Código 2.14: Error Cuadrático Medio

```
ecm :: [Ejemplo] -> Double
ecm ejemplos =
    let v          = map (getR.valorObjetivo) ejemplos
        prediccion = media v
        ecms       = map (\ x -> (x-prediccion)^2) v
    in media ecms

ecmAtributo :: [Ejemplo] -> Atributo -> Double
ecmAtributo ejemplos atributo =
     ecmAtributoUmbral ejemplos (getR atributo) (fromJust $ umbral $ getR
         atributo)

ecmAtributoUmbral :: [Ejemplo] -> Continuo -> Double -> Double
```

```
ecmAtributoUmbral ejemplos atributo umbral =
    let atrib = Right atributo
        n = lengthDouble ejemplos
        s1 = [ x | x <- ejemplos, (getR $ valorAtributo x atrib) <= umbral ]
        p1 = lengthDouble s1 / n
        s2 = [ x | x <- ejemplos, (getR $ valorAtributo x atrib) > umbral ]
        p2 = lengthDouble s2 / n
    in p1 * (ecm s1) + p2 * (ecm s2)
```

Código 2.14: Error Cuadrático Medio

2.9– Discretizar atributos continuos

Para discretizar un atributo continuo, se consideran todos los posibles umbrales con la función `posiblesParticiones`. La función `mejorUmbral` busca entre todos ellos aquel que maximiza o minimiza una función, datos que se añaden como parámetros `f` y `operador`.

La función `discretizar` actúa sobre una lista de atributos. Se define como la función identidad cuando el atributo es discreto. En el caso de un atributo continuo: se calcula el umbral óptimo haciendo uso de la función definida previamente y se sustituye el atributo por uno con dicho valor en el umbral. Como se definió la igualdad entre atributos sin tener en cuenta `umbral`, las funciones de búsqueda entre los atributos seguirán funcionando correctamente. `discretizar` toma como parámetro una función `f` a optimizar, un `operador` ($\leq$ or ¿) y un atributo.

Código 2.15: Discretizar

```
posiblesParticiones :: Continuo -> [Double] -> [Double]
posiblesParticiones atributo valores =
    let xs = ordensindups ([fst $ rango atributo] ++ valores ++ [snd $ rango
        atributo])
    in [ (x + y) / 2 | (x,y) <- zip xs (tail xs) ]

discretizar :: ([Ejemplo] -> Continuo -> Double -> Double) -> (Double -> Double
    -> Bool) -> [Ejemplo] -> Atributo -> Atributo
discretizar _ _ _ (Left atributo) = (Left atributo)
discretizar f operador ejemplos (Right atributo) =
    let umbral = mejorUmbral f operador atributo ejemplos
    in Right (C (nombreC atributo) (rango atributo) (Just umbral))

mejorUmbral :: ([Ejemplo] -> Continuo -> Double -> Double) -> (Double -> Double
    -> Bool) -> Continuo -> [Ejemplo] -> Double
mejorUmbral f operador atributo ejemplos =
    let v          = ordensindups $ map getR (valores (Right atributo) ejemplos)
        us         =  posiblesParticiones atributo v
        umbral_inic = head us
        f_inic     = f ejemplos atributo umbral_inic
    in fst $ foldl (\ (u, f_u) x ->
                  let f_x = f ejemplos atributo x
                  in if f_x `operador` f_u
```

```
21                      then (x, f_x)
22                      else (u, f_u) )
23              (head us,f_inic) (tail us)
```

Código 2.15: Discretizar

Se particulariza la función `discretizar` para cada función considerada en los algoritmos de construcción.

Código 2.16: Personalizar discretizar

```
1 discretizarGanInfo = discretizar gananciaNormCUmbral (>)
2 discretizarECM = discretizar ecmAtributoUmbral (<)
3 discretizarGini = discretizar giniAtributoUmbral (<)
```

Código 2.16: Personalizar discretizar

2.10– Mejor atributo

Se define la función `mejorClasifica`, que devuelve el atributo que optimiza una función `f` dada como parámetro. El sentido de la optimización lo determina el parámetro `operador`.

Código 2.17: Mejor atributo

```
1 mejorClasifica :: ([Ejemplo] -> Atributo -> Double) -> (Double -> Double -> Bool)
      -> [Atributo] -> [Ejemplo] -> Atributo
2 mejorClasifica f operador atributos ejemplos =
3     let at_inic = head atributos
4         f_inic = f ejemplos at_inic
5     in fst $ foldl (\(at,f_at) x -> let f_x = f ejemplos x
6                     in if f_x `operador` f_at
7                     then (x, f_x)
8                     else (at,f_at) )
9             (head atributos, f_inic) (tail atributos)
```

Código 2.17: Mejor atributo

De igual forma que con `discretizar`, se personalizan la función para cada caso.

Código 2.18: Personalizar mejor atributo

```
1 mejorClasificaGan = mejorClasifica gananciaNorm (>)
2 mejorClasificaECM = mejorClasifica ecmAtributo (<)
3 mejorClasificaGini = mejorClasifica giniAtributo (<)
```

Código 2.18: Personalizar mejor atributo

2.11– Condiciones de parada

Se definen las condiciones de parada, diferenciando entre problemas de clasificación y de regresión. Ambas devuelven un par (`Bool`,`ValorAtrib`) donde el primer elemento es un Booleano que representa si se cumple la condición de parada y el segundo es el valor que tomaría el nodo hoja.

Código 2.19: Condiciones de parada en clasificación

```
masComun :: [Ejemplo] -> ValorAtrib
masComun [] = Left "error: mascomun de lista vacia"
masComun ejemplos =
    let valores_clasificacion = (posiblesValores.atributoObjetivo.head) ejemplos
        clasificaciones     = map (getL.valorObjetivo) ejemplos
    in Left $ maximo [ (x,ocurrencia clasificaciones x) | x <-
        valores_clasificacion ]
    (head clasificaciones,0)

paradaClasificacion :: Double -> [Ejemplo] -> (Bool, ValorAtrib)
paradaClasificacion min ejemplos =
    let n  = lengthDouble ejemplos
        h  = masComun ejemplos
        p_h = (ocurrencia (map valorObjetivo ejemplos) h) / n
    in if p_h >= min
       then (True, h)
       else (False, h)
```

Código 2.19: Condiciones de parada en clasificación

Código 2.20: Condiciones de parada en regresión

```
prediccionHoja :: [Ejemplo] -> ValorAtrib
prediccionHoja ejemplos =
    let v = map (getR.valorObjetivo) ejemplos
    in Right $ media v

paradaRegresion :: Double -> [Ejemplo] -> (Bool, ValorAtrib)
paradaRegresion max ejemplos =
    let n    = lengthDouble ejemplos
        pred = prediccionHoja ejemplos
        error = ecm ejemplos
    in if error <= max
       then (True, pred)
       else (False, pred)
```

Código 2.20: Condiciones de parada en regresión

2.12– Construcción de árboles

A continuación se definirán los algoritmos de construcción de árboles C4.5 y CART, haciendo uso de las funciones ya definidas.

En primer lugar, se define un algoritmo de construcción general, `arbolGenerico`.

Código 2.21: Construcción de un árbol genérico

```
1  arbolGenerico :: ([Ejemplo] -> ValorAtrib) -> ([Ejemplo] -> Atributo -> Atributo)
       -> ([Atributo] -> [Ejemplo] -> Atributo) -> (Double -> [Ejemplo] -> (Bool,
       ValorAtrib)) -> ValorAtrib -> Double -> [Atributo] -> [Ejemplo] -> Arbol
2  arbolGenerico _ _ _ _ hoja_padre _ _ [] = Hoja $ hoja_padre
3  arbolGenerico crea_hoja _ _ _ _ _ [] ejemplos = Hoja $ crea_hoja ejemplos
4  arbolGenerico crea_hoja discretizador mejorClasificador condicionParada
       hoja_padre param_parada atributos ejemplos =
5      let discretizados  = map (discretizador ejemplos) atributos
6          mejor_atributo = mejorClasificador discretizados ejemplos
7          nuevo_atributos = elimina mejor_atributo atributos
8          parada         = condicionParada param_parada ejemplos
9          nueva_hoja_padre = crea_hoja ejemplos
10     in
11     if fst parada
12     then Hoja $ snd parada
13     else Nodo mejor_atributo
14     (creaHijo crea_hoja discretizador mejorClasificador condicionParada
15     nueva_hoja_padre param_parada nuevo_atributos (evaluar ejemplos
           mejor_atributo))
16
17 creaHijo :: ([Ejemplo] -> ValorAtrib) -> ([Ejemplo] -> Atributo -> Atributo) ->
       ([Atributo] -> [Ejemplo] -> Atributo) -> (Double -> [Ejemplo]-> (Bool,
       ValorAtrib)) -> ValorAtrib -> Double -> [Atributo] -> (String -> [Ejemplo])
       -> String -> Arbol
18 creaHijo crea_hoja discretizador mejorClasificador condicionParada hoja_padre
       param_parada atributos evaluarValor valor =
19     let ejemplos = evaluarValor valor
20     in arbolGenerico crea_hoja discretizador mejorClasificador
21     condicionParada hoja_padre param_parada atributos ejemplos
```

Código 2.21: Construcción de un árbol genérico

Especificando las funciones de cada algoritmo, se definen C4.5 y CART.

Código 2.22: Personalizar árbol genérico

```
1  arbolC45 = arbolGenerico masComun discretizarGanInfo mejorClasificaGan
       paradaClasificacion
2  arbolCARTclas = arbolGenerico masComun discretizarGini mejorClasificaGini
       paradaClasificacion
3  arbolCARTreg = arbolGenerico prediccionHoja discretizarECM mejorClasificaECM
       paradaRegresion
4
5  c45 = arbolC45 $ Left "error: Ejemplos vacio."
6  cart "clasificacion" = arbolCARTclas $ Left "error: Ejemplos vacio."
7  cart "regresion" = arbolCARTreg $ Right 0.0
```

Código 2.22: Personalizar árbol genérico

2.13– Error

Clasificación

El error de clasificación de un árbol en una lista de ejemplos es la proporción de ejemplos que este clasifica incorrectamente.

Código 2.23: Error de clasificación

```
bienClasificado :: Arbol -> Ejemplo -> Bool
bienClasificado arbol ejemplo =
   let prediccion  = predice arbol (fst ejemplo)
       clas_correcta = valorObjetivo ejemplo
   in prediccion == clas_correcta

errorClasificacionConjunto :: Arbol -> [Ejemplo] -> Double
errorClasificacionConjunto arbol ejemplos =
    let errores = lengthDouble $ filter (==False) (map (bienClasificado arbol)
        ejemplos)
        n       = lengthDouble ejemplos
    in errores / n
```

Código 2.23: Error de clasificación

Regresión

El error de clasificación de un árbol en una lista de ejemplos es el Error Cuadrático Medio respecto a la predicción de dicho árbol.

Código 2.24: Error de regresión

```
errorRegresion :: Arbol -> Ejemplo -> Double
errorRegresion arbol ejemplo =
   let correcta  = getR $ valorObjetivo ejemplo
       prediccion = getR $ predice arbol (fst ejemplo)
   in (correcta - prediccion) ^ 2

errorRegresionConjunto :: Arbol -> [Ejemplo] -> Double
errorRegresionConjunto arbol ejemplos =
    let error = sum $ map (errorRegresion arbol) ejemplos
        n     = lengthDouble ejemplos
    in error / n
```

Código 2.24: Error de regresión

2.14– Medidas sobre árboles

Se definen algunas funciones sobre árboles, que podrán ser usadas como indicación de su complejidad y como condiciones de parada (prepoda).

Profundidad

Código 2.25: Profundidad

```
profundidad :: Arbol -> Int
profundidad (Hoja x) = 1
profundidad (Nodo atrib hijo) =
    let valores = posiblesValores atrib
    in 1 + maximum ((map profundidad) (map hijo valores))
```

Código 2.25: Profundidad

Número de Hojas

Código 2.26: Número de Hojas

```
numeroHojas :: Arbol -> Int
numeroHojas (Hoja x) = 1
numeroHojas (Nodo atrib hijo) =
    let valores = posiblesValores atrib
    in sum ((map numeroHojas) (map hijo valores))
```

Código 2.26: Número de Hojas

2.15– Evaluar árboles

Se define la función `predice`, que evalúa un árbol sobre un ejemplo para obtener su predicción.

Código 2.27: Evaluar árboles

```
predice :: Arbol -> [(Atributo,ValorAtrib)] -> ValorAtrib
predice (Hoja prediccion) atrib_val = prediccion
predice (Nodo (Left atributo) hijo) atrib_val =
    let v = getL $ snd $ head $ filter (\x -> fst x == (Left atributo)) atrib_val
    in predice (hijo v) atrib_val
predice (Nodo (Right atributo) hijo) atrib_val =
    let v = getR $ snd $ head $ filter (\x -> fst x == (Right atributo))
        atrib_val
        u = fromJust $ umbral atributo
    in if v <= u
       then predice (hijo "<=") atrib_val
       else predice (hijo ">") atrib_val
```

Código 2.27: Evaluar árboles

2.16– Random Forest

Haciendo uso de la librería `System.Random`, se define `subListaRandom`. Esta función selecciona de forma pseudo-aleatoria con reemplazamiento una muestra de una lista. Pseudo, ya que Haskell no puede trabajar con datos aleatorios sin la librería `UnsafeIO`.

Código 2.28: Selección aleatoria de muestras

```
generadores n i j = map mkStdGen (take n [i,j..])

selecRandom :: [a] -> StdGen -> a
selecRandom xs generador = xs !! rand where
    n = length xs
    (rand, _) = randomR (0,(n-1)) generador

subListaRandom n i j xs =
    map (selecRandom xs) (generadores n i j)
```

Código 2.28: Selección aleatoria de muestras

Con la función anterior se implementa `boostTraining`, que selecciona con reemplazamiento k listas de ejemplos. `(i,j)` son los parámetros correspondientes a la pseudo-aleatoriedad.

Código 2.29: Selección aleatoria de ejemplos

```
bootsEjemplos :: Int -> [Ejemplo] -> (Int,Int) -> [ [Ejemplo] ]
bootsEjemplos 0 _ _ = []
bootsEjemplos k ejemplos (i,j) =
    let n = length ejemplos
    in (subListaRandom n i j ejemplos) : bootsEjemplos (k-1) ejemplos (3*i,2*j)
```

Código 2.29: Selección aleatoria de ejemplos

Se realiza el mismo procedimiento sobre la lista de atributos.

Código 2.30: Selección aleatoria de atributos

```
bootsAtribs :: Int -> Int -> [Atributo] -> (Int,Int) -> [ [Atributo] ]
bootsAtribs 0 _ _ _ = []
bootsAtribs _ _ [] _ = []
bootsAtribs k n atributos (i,j) =
    let randomAt = subListaRandom n i j atributos
    in randomAt : bootsAtribs (k-1) n atributos (3*i,2*j)

bootsAtribSinReemplazo :: Int -> Int -> [Atributo] -> (Int,Int) -> [ [Atributo] ]
bootsAtribSinReemplazo 0 _ _ _ = []
bootsAtribSinReemplazo _ _ [] _ = []
bootsAtribSinReemplazo k n atributos (i,j) =
    let randomAt = subListaRandom n i j atributos
        nuevosAt = eliminaLista randomAt atributos
    in randomAt : bootsAtribSinReemplazo (k-1) n nuevosAt (3*i,2*j)
```

Código 2.30: Selección aleatoria de atributos

La función `buildkModels` construye k árboles a partir de las k listas de ejemplos y atributos dados por las funciones anteriores.

Código 2.31: Creación de Random Forest

```
construirkArboles :: Int -> [Ejemplo] -> [Atributo] -> Int -> ([Atributo] -> [
    Ejemplo] -> Arbol) -> [Arbol]
construirkArboles k ejemplos atributos n_atributos modelo =
   let n_e = length ejemplos
       n_a = length atributos
       k_ejemplos = bootsEjemplos k ejemplos (21,13)
       k_atrib = bootsAtribs k n_atributos atributos (13,7)
       zip_ej_at = zip k_atrib k_ejemplos
   in map (\ (at,ej) -> modelo at ej) zip_ej_at
```

Código 2.31: Creación de Random Forest

La función `listaPredicciones` evalúa cada árbol de una lista sobre un ejemplo y devuelve una lista con las predicciones. `prediccionCombinada` devuelve la predicción dada por voto mayoritario (clasificación) o media (regresión) al evaluar una lista de árboles sobre un ejemplo.

Código 2.32: Predicción con Random Forest

```
listaPredicciones :: Ejemplo -> [Arbol] -> [ValorAtrib]
listaPredicciones ejemplo arboles =
    map (\ arbol -> predice arbol (fst ejemplo)) arboles


votoMayoritario :: (Eq a) => [a] -> (a,Double)
votoMayoritario xs =
    let n = lengthDouble xs
        m = maximo [ (x,ocurrencia xs x) | x <- xs ] (head xs,0)
        p = (ocurrencia xs m) / n
    in (m,p)

prediccionCombinada :: Ejemplo -> [Arbol] -> ValorAtrib
prediccionCombinada (at_val,(Left atOb,x)) arboles =
    fst $ votoMayoritario $ listaPredicciones (at_val,(Left atOb,x)) arboles
prediccionCombinada (at_val,(Right atOb,x)) arboles =
    Right (media $ map getR $ listaPredicciones (at_val,(Right atOb,x)) arboles)
```

Código 2.32: Predicción con Random Forest

Se crean dos funciones, una para cada tipo de error usado, que evalúan el error de un conjunto (lista) de árboles sobre una lista de ejemplos.

Código 2.33: Error de Random Forest

```
1
2  errorRFclas :: [Arbol] -> [Ejemplo] -> Double
3  errorRFclas arboles ejemplos =
4      let predicciones    = map (\ x -> prediccionCombinada x arboles) ejemplos
5          correctas       = map valorObjetivo ejemplos
6          bien_clasificados = zipWith (==) predicciones correctas
7          errores         = lengthDouble $ filter (==False) bien_clasificados
8          n               = lengthDouble ejemplos
9      in errores / n
10
11 errorRFreg :: [Arbol] -> [Ejemplo] -> Double
12 errorRFreg arboles ejemplos =
13     let predicciones    = map (\ x -> getR $ prediccionCombinada x arboles)
           ejemplos
14         correctas       = map (getR.valorObjetivo) ejemplos
15         diferencias     = zipWith (-) predicciones correctas
16         ecm             = sum $ map (^2) diferencias
17         n               = lengthDouble ejemplos
18     in ecm / n
```

Código 2.33: Error de Random Forest

Por último, se incluye una función que evalúa todos los árboles de una lista de árboles sobre una lista de ejemplos y elimina aquellos que tengan un error superior a un umbral dado.

Código 2.34: Filtrar árboles de un Random Forest

```
1  filtrarBosque :: (Arbol -> [Ejemplo] -> Double) -> [Arbol] -> [Ejemplo] -> Double
       -> [Arbol]
2  filtrarBosque f arboles ejemplos umbral =
3        foldl (\ac arbol -> if (f arbol ejemplos) < umbral
4              then arbol:ac else ac) [] arboles
```

Código 2.34: Filtrar árboles de un Random Forest

CAPÍTULO 3

Manual de uso

En este capítulo se mostrará el uso de la librería, tomando tres problemas como ejemplos. En primer lugar, con un ejemplo simple se podrá observar la forma en que se representan los ejemplos, los árboles y las predicciones.

Posteriormente, se considerará un problema de clasificación con C4.5 y uno de regresión con CART. También se introducirá un ejemplo simple de modelos combinados de árboles en cada caso.

Se usará la librería `System.IO.Unsafe` [14] para el uso de datos provenientes de archivos `csv`, habiéndose consultado en [16] cómo realizarlo.

3.1– Primer ejemplo

Se considera el problema: dado dos atributos X e Y predecir el atributo objetivo $Color$. $Color$ toma valores $\{Rojo, Verde\}$. X toma valores discretos $\{A, B\}$ mientras que Y toma valores continuos en el intervalo [0,1]. El conjunto de ejemplos de entrenamiento es $= \{((A, 0), Rojo), ((A, 1), Verde), ((B, 1), Rojo)\}$ y el ejemplo a predecir es $(B$,0).

Se considera el algoritmo de construcción de árboles de decisión C4.5, apto para problemas de clasificación con atributos tanto continuos como discretos.

Código 3.1: Ejemplo Color

```
1 module EjemploSimple
2     where
3
4 import Tipos
5 import Utils
6 import ConstruirArbol
7 import Evaluar
8
```

```
-- Atributo Objetivo

objetivo = D "Color" ["Rojo","Azul"] :: Discreto

-- Atributos

x      = D  "X"  ["A","B"]          :: Discreto
y      = C  "Y"    (0,1)    Nothing :: Continuo

atributos = [Left x, Right y] :: [Atributo]

-- Ejemplos

ejemplo1 = ([aleft (x, "B"), aright (y, 1.0)], aleft (objetivo, "Rojo")) ::
    Ejemplo
ejemplo2 = ([aleft (x, "A"), aright (y, 1.0)], aleft (objetivo, "Azul")) ::
    Ejemplo
ejemplo3 = ([aleft (x, "A"), aright (y, 0.0)], aleft (objetivo, "Rojo")) ::
    Ejemplo

ejemplos = [ejemplo1,ejemplo2,ejemplo3] :: [Ejemplo]

ejemploapredecir = [aleft (x,"B"), aright (y, 0.0)] :: [(Atributo, ValorAtrib)]

-- Arbol

arbol = c45 1 atributos ejemplos :: Arbol

-- Prediccion

prediccion = predice arbol ejemploapredecir :: ValorAtrib
```

Código 3.1: Ejemplo Color

La salida de estas funciones es:

Código 3.2: Salida Ejemplo Color

```
*EjemploSimple> arbol
<Left X|
[(<Right Y {Umbral: 0.5}|
[(Left "Rojo","<="),(Left "Azul",">")]>,"A"),(Left "Rojo","B")]>

*EjemploSimple> ejemploapredecir
[(Left X,Left "B"),(Right Y,Right 0.0)]

*EjemploSimple> prediccion
Left "Rojo"
```

Código 3.2: Salida Ejemplo Color

La representación del árbol obtenido es:

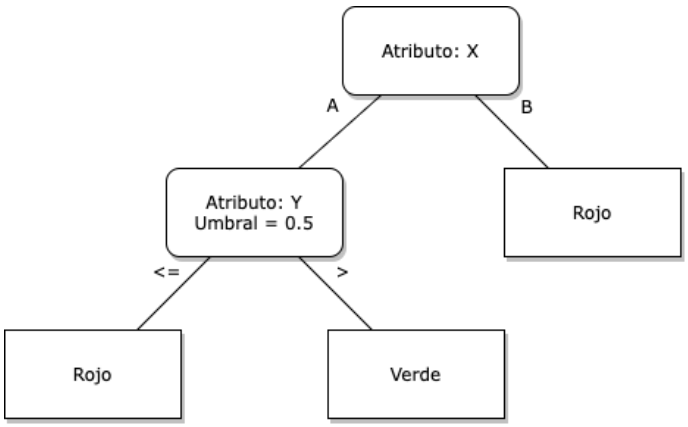

Figura 3.1: Árbol Ejemplo Color

y divide el espacio de ejemplos de la siguiente manera.

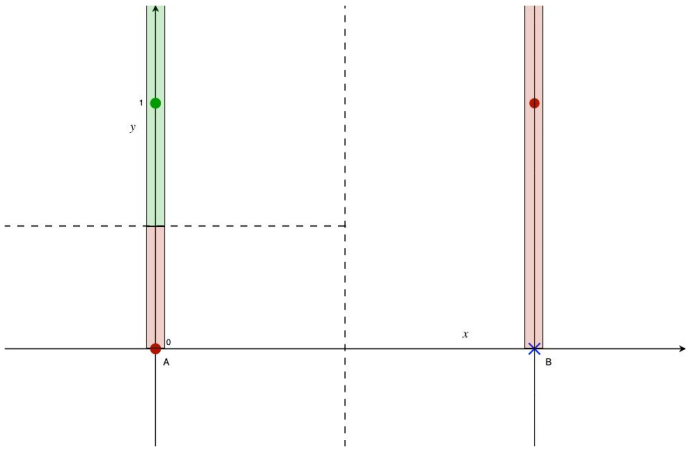

Figura 3.2: Representación ejemplo Color

donde la cruz azul representa el ejemplo a predecir.

3.2– Clasificación: Medicamentos

El siguiente problema se ha obtenido de [17]. En un fichero de tipo `csv` se encuentran datos de 200 pacientes de las siguientes categorías: edad, sexo, presión sanguínea, nivel de colesterol, ratio sodio-potasio, medicamento. El problema consiste en modelar qué medicamento, entre 5 tipos, se debe suministrar a un paciente en función del resto de variables.

En primer lugar, es necesario leer los datos del archivo `csv` y estructurarlos acorde a los tipos anteriormente definidos.

Código 3.3: Leer y preparar ejemplos en Medicamentos

```haskell
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE RecordWildCards #-}

module ReadandPrepareDrugs
    where

-- base
import Control.Exception (IOException)
import qualified Control.Exception as Exception
import qualified System.Exit as Exit
import System.IO.Unsafe

-- bytestring
import Data.ByteString.Lazy (ByteString)
import qualified Data.ByteString.Lazy as ByteString

-- cassava
import Data.Csv
  ( FromField(parseField)
  , FromNamedRecord(parseNamedRecord)
  , (.:)
  )
import qualified Data.Csv as Cassava

-- vector
import Data.Vector (Vector, toList)

-- Libreria arboles
import Tipos
import Utils

data Item =
  Item
    { itemAge :: Double
    , itemSex :: String
    , itemBP :: String
    , itemCholesterol :: String
    , itemNa_to_K :: Double
    , itemDrug :: String
    }
  deriving (Eq, Show)

instance FromNamedRecord Item where
  parseNamedRecord m =
    Item
      <$> m .: "Age"
      <*> m .: "Sex"
      <*> m .: "BP"
      <*> m .: "Cholesterol"
      <*> m .: "Na_to_K"
      <*> m .: "Drug"

```

```haskell
decodeItems
  :: ByteString
  -> Either String (Vector Item)
decodeItems =
  fmap snd . Cassava.decodeByName

decodeItemsFromFile
  :: FilePath
  -> IO (Either String (Vector Item))
decodeItemsFromFile filePath =
  catchShowIO (ByteString.readFile filePath)
    >>= return . either Left decodeItems

catchShowIO
  :: IO a
  -> IO (Either String a)
catchShowIO action =
  fmap Right action
    `Exception.catch` handleIOException
  where
    handleIOException
      :: IOException
      -> IO (Either String a)
    handleIOException =
      return . Left . show

-----------------------------------
------------ Atributos ------------
-----------------------------------

age         = C "age" (10,90) Nothing

sex         = D "sex" ["F", "M"]

bp          = D "BP" ["HIGH","NORMAL","LOW"]

cholesterol = D "cholesterol" ["HIGH","NORMAL"]

na_to_K     = C "na_to_K" (5,40) Nothing

drug        = D "drug" ["DrugX","DrugY","DrugA","DrugB","DrugC"]

-----------------------------------
-----------------------------------

prepareItem :: Item -> Ejemplo
prepareItem item =
    (
    [aright (age,itemAge item),
    aleft (sex,itemSex item),
    aleft (bp, itemBP item),
    aleft (cholesterol,itemCholesterol item),
    aright (na_to_K,itemNa_to_K item)],
```

```
106     aleft (drug,itemDrug item)
107     )
108
109
110 prepare :: Int -> String -> (Item -> Ejemplo) -> IO [Ejemplo]
111 prepare n file prepareItems = do
112     eitherItems <- decodeItemsFromFile file
113
114     case eitherItems of
115       Left reason ->
116         Exit.die reason
117
118       Right items -> do
119         return $ map prepareItems (take n (toList items))
```

Código 3.3: Leer y preparar ejemplos en Medicamentos

Aquí se definen los atributos del problema y haciendo uso de la librería `Data.Csv` [15] se transforman los datos al tipo `Ejemplo`.

Ahora preparamos el fichero que separa los ejemplos en entrenamiento y validación, crea un árbol de decisión, un modelo *Random Forest* y mide sus errores.

Código 3.4: Modelar Medicamentos

```
1  module ModelDrugs
2      where
3
4  import System.IO.Unsafe
5  import ReadandPrepareDrugs
6  import Tipos
7  import ConstruirArbol
8  import Error
9  import RandomForest
10
11 -- Preparar ejemplos
12
13 ejemplos :: [Ejemplo]
14 ejemplos = unsafePerformIO $ prepare 201 "drug200.csv" prepareItem
15
16 splitEjemplos :: ([Ejemplo],[Ejemplo])
17 splitEjemplos = splitAt 150 ejemplos
18
19 entrenamiento = fst splitEjemplos :: [Ejemplo]
20
21 validacion   = snd splitEjemplos :: [Ejemplo]
22
23 -- Atributos
24
25 atributos = [Right age,Left sex,Left bp,Left cholesterol,Right na_to_K]
26
27 atObjetivo = Left drug
28
29 -- Construir árbol
```

```
30
31 param_parada = 0.5
32
33 arbol = c45 param_parada atributos entrenamiento
34
35 -- Random Forest
36
37 n_arboles = 50
38 n_atributos = 4
39 param_parada_bosque = 0.5
40
41 prebosque = (construirkArboles n_arboles entrenamiento atributos n_atributos (c45
       param_parada_bosque))
42
43 bosque = filtrarBosque errorClasificacionConjunto prebosque entrenamiento 0.5
44
45 -- Evaluar error
46
47 errorArbolEntrenamiento = errorClasificacionConjunto arbol entrenamiento
48 errorArbolValidacion  = errorClasificacionConjunto arbol validacion
49
50 errorRFentrenamiento = errorRFclas bosque entrenamiento
51 errorRFvalidacion  = errorRFclas bosque validacion
```

Código 3.4: Modelar Medicamentos

La salida de estas funciones es

Código 3.5: Salida de Modelar Medicamentos

```
1  *ModelDrugs> arbol
2  <Right na_to_K {Umbral: 14.627}|
3  [(<Right age {Umbral: 12.5}|
4  [(Left "drugC","<="),(<Left sex|
5  [(Left "drugX","F"),(<Left BP|
6  [(Left "drugA","HIGH"),(Left "drugX","NORMAL"),(<Left cholesterol|
7  [(Left "drugC","HIGH"),(Left "drugX","NORMAL")]>,"LOW")]>,"M")]>,">")]>,"<="),(
       Left "DrugY",">")]>
8
9  *ModelDrugs> take 3 bosque
10 [<Right na_to_K {Umbral: 14.674}|
11 [(<Left cholesterol|
12 [(Left "drugA","HIGH"),(Left "drugX","NORMAL")]>,"<="),(Left "DrugY",">")]>,<
       Right na_to_K {Umbral: 14.487}|
13 [(<Left BP|
14 [(Left "drugA","HIGH"),(Left "drugX","NORMAL"),(Left "drugC","LOW")]>,"<="),(Left
       "DrugY",">")]>,<Right na_to_K {Umbral: 14.807500000000001}|
15 [(<Left cholesterol|
16 [(<Right age {Umbral: 14.5}|
17 [(Left "drugA","<="),(Left "drugA",">")]>,"HIGH"),(<Right age {Umbral: 14.0}|
18 [(Left "drugB","<="),(Left "drugB",">")]>,"NORMAL")]>,"<="),(Left "DrugY",">")]>]
19 (4.41 secs, 1,339,503,168 bytes)
20
21 *ModelDrugs> length bosque
```

```
22
23
*ModelDrugs> errorRFentrenamiento
0.13333333333333333
26
*ModelDrugs> errorRFvalidacion
0.16
29
*ModelDrugs> errorArbolEntrenamiento
0.17333333333333334
32
*ModelDrugs> errorArbolValidacion
0.14
```

Código 3.5: Salida de Modelar Medicamentos

Se genera el siguiente árbol.

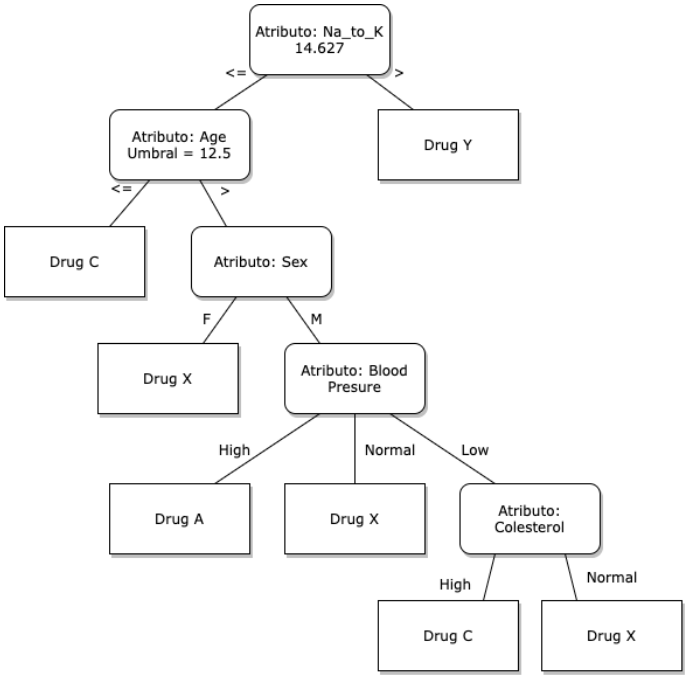

Figura 3.3: Árbol que modela Medicamentos

3.3– Regresión: Calificaciones

En este caso, el problema consiste en predecir la calificación en matemáticas de un alumno a partir de las variables: sexo, raza/etnia, nivel de educación de los padres, almuerzo, curso de preparación, calificación en lectura y calificación en redacción. En este caso el atributo objetivo es continuo por tanto se considera un árbol de decisión producido por el algoritmo CART (regresión).

De igual forma que en el anterior problema, los datos se han obtenido de [18]. Se tiene un archivo `csv` con 1000 filas que debe ser procesado.

En el siguiente fichero, se definen los atributos del problema y se procesan los datos.

Código 3.6: Leer y preparar ejemplos

```haskell
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE RecordWildCards #-}

module ReadAndPrepareMarks
  where

-- base
import Control.Exception (IOException)
import qualified Control.Exception as Exception
import qualified System.Exit as Exit
import System.IO.Unsafe

-- bytestring
import Data.ByteString.Lazy (ByteString)
import qualified Data.ByteString.Lazy as ByteString

-- cassava
import Data.Csv
  ( FromField(parseField)
  , FromNamedRecord(parseNamedRecord)
  , (.:)
  )
import qualified Data.Csv as Cassava

-- vector
import Data.Vector (Vector, toList)

-- Libreria arboles
import Tipos
import Utils


data Item =
  Item
    { itemGender :: String
    , itemRace :: String
    , itemParentsEd :: String
    , itemLunch :: String
    , itemCourse :: String
    , itemMathScore :: Double
    , itemReadingScore :: Double
    , itemWritingScore :: Double
    }
  deriving (Eq, Show)

instance FromNamedRecord Item where
  parseNamedRecord m =
```

```
48      Item
49        <$> m .: "gender"
50        <*> m .: "race/ethnicity"
51        <*> m .: "parental level of education"
52        <*> m .: "lunch"
53        <*> m .: "test preparation course"
54        <*> m .: "math score"
55        <*> m .: "reading score"
56        <*> m .: "writing score"
57
58  decodeItems
59    :: ByteString
60    -> Either String (Vector Item)
61  decodeItems =
62    fmap snd . Cassava.decodeByName
63
64  decodeItemsFromFile
65    :: FilePath
66    -> IO (Either String (Vector Item))
67  decodeItemsFromFile filePath =
68    catchShowIO (ByteString.readFile filePath)
69      >>= return . either Left decodeItems
70
71  catchShowIO
72    :: IO a
73    -> IO (Either String a)
74  catchShowIO action =
75    fmap Right action
76      `Exception.catch` handleIOException
77    where
78      handleIOException
79        :: IOException
80        -> IO (Either String a)
81      handleIOException =
82        return . Left . show
83
84
85  ------------------------------------
86  ------------ Atributos -------------
87  ------------------------------------
88
89  gender = C "gender" (0,1) Nothing -- ["male","female"]
90
91  gendertoC s = case s of
92          "male" -> 0
93          "female" -> 1
94
95  race = C "race" (0,4) Nothing -- ["group A", "group B", "group C", "group D", "
        group E"]
96
97  racetoC s = case s of
98         "group A" -> 0
99         "group B" -> 1
```

```
100         "group C" -> 2
101         "group D" -> 3
102         "group E" -> 4
103
104 parentsed = C "parentsed" (0,5) Nothing -- ["some high school","bachelor's degree
        ","some college","master's degree","associate's degree","high school"]
105
106 parentsedtoC s = case s of
107         "some high school" -> 0
108         "high school" -> 1
109         "some college" -> 2
110         "associate's degree" -> 3
111         "bachelor's degree" -> 4
112         "master's degree" -> 5
113
114 lunch = C "lunch" (0,1) Nothing -- ["standard","free/reduced"]
115
116 lunchtoC s = case s of
117          "standard" -> 0
118          "free/reduced" -> 1
119
120 course = C "course" (0,1) Nothing -- ["none","completed"]
121
122 coursetoC s = case s of
123          "none" -> 0
124          "completed" -> 1
125
126 mathscore = C "mathscore" (0,100) Nothing
127
128 readingscore = C "readingscore" (0,100) Nothing
129
130 writingscore = C "writingscore" (0,100) Nothing
131
132 -----------------------------------
133 -----------------------------------
134
135 prepareItem item =
136     let i = item
137     in
138     (
139     [
140     aright (gender, gendertoC $ itemGender i),
141     aright (race, racetoC $ itemRace i),
142     aright (parentsed, parentsedtoC $ itemParentsEd i),
143     aright (lunch, lunchtoC $ itemLunch i),
144     aright (course, coursetoC $ itemCourse i),
145     aright (readingscore, itemReadingScore i),
146     aright (writingscore, itemWritingScore i)
147     ],
148     aright (mathscore, itemMathScore i)
149     )
150
151 prepare :: Int -> String -> (Item -> Ejemplo) -> IO [Ejemplo]
```

```haskell
prepare n file prepareItems = do
    eitherItems <- decodeItemsFromFile file

    case eitherItems of
      Left reason ->
        Exit.die reason

      Right items -> do
        return $ map prepareItems (take n (toList items))
```

Código 3.6: Leer y preparar ejemplos

En el siguiente fichero, se dividen los ejemplos en entrenamiento y validación, se construye un árbol de decisión CART, un modelo *Random Forest* simple y se evalúan sus errores.

Código 3.7: Modelar Calificaciones

```haskell
module ModelMarks
    where

import System.IO.Unsafe
import ReadAndPrepareMarks
import Tipos
import ConstruirArbol
import Error
import RandomForest

-- Preparar ejemplos

ejemplos :: [Ejemplo]
ejemplos = unsafePerformIO $ prepare 500 "students_performance.csv" prepareItem

splitEjemplos = splitAt 450 ejemplos :: ([Ejemplo],[Ejemplo])

entrenamiento = fst splitEjemplos

validacion   = snd splitEjemplos

-- Atributos

atributos = [Right gender,Right race,Right parentsed,Right lunch,Right course,
    Right readingscore,Right writingscore]

atObjetivo = Right mathscore

-- Construir árbol

param_parada = 200

arbol = cart "regresion" param_parada atributos entrenamiento

-- Random Forest

```

```haskell
n_arboles  = 20
n_atributos = 3
param_parada_bosque = 150

prebosque = (construirkArboles n_arboles entrenamiento atributos n_atributos (
    cart "regresion" param_parada_bosque))

bosque = filtrarBosque errorRegresionConjunto prebosque entrenamiento 400


-- Evaluamos los errores

errorArbolEntrenamiento = errorRegresionConjunto arbol entrenamiento
errorArbolValidacion  = errorRegresionConjunto arbol validacion

errorBosqueEntrenamiento = errorRFreg bosque entrenamiento
errorBosqueValidacion  = errorRFreg bosque validacion
```

Código 3.7: Modelar Calificaciones

La salida de estas funciones es

Código 3.8: Salida de Modelar Calificaciones

```
*ModelMarks> arbol
<Right lunch {Umbral: 0.5}|
[(Right 69.5016835016835,"<="),(<Right parentsed {Umbral: 2.5}|
[(<Right course {Umbral: 0.5}|
[(<Right gender {Umbral: 0.5}|
[(Right 54.51851851851852,"<="),(<Right race {Umbral: 2.5}|
[(<Right readingscore {Umbral: 8.5}|
[(Right 45.44444444444444,"<="),(<Right writingscore {Umbral: 5.0}|
[(Right 45.44444444444444,"<="),(Right 45.44444444444444,">")]>,">")]>,"<="),(<
    Right readingscore {Umbral: 17.0}|
[(Right 53.625,"<="),(<Right writingscore {Umbral: 16.0}|
[(Right 53.625,"<="),(Right 53.625,">")]>,">")]>,">")]>,">")]>,"<="),(Right
    60.225806451612904,">")]>,"<="),(Right 63.766666666666666,">")]>,">")]>
(2.44 secs, 917,868,168 bytes)

*ModelMarks> errorArbolEntrenamiento
181.1992983418412

*ModelMarks> errorArbolValidacion
198.50177074328326

*ModelMarks> take 2 bosque
[<Right gender {Umbral: 0.5}|
[(<Right race {Umbral: 3.5}|
[(<Right parentsed {Umbral: 3.5}|
[(Right 64.25157232704403,"<="),(Right 73.56,">")]>,"<="),(<Right parentsed {
    Umbral: 2.5}|
[(Right 79.61111111111111,"<="),(Right 72.04761904761905,">")]>,">")]>,"<="),(<
    Right race {Umbral: 0.5}|
[(Right 45.5,"<="),(<Right parentsed {Umbral: 2.5}|
```

```
[(Right 57.1965811965812,"<="),(Right 62.03,">")]>,">")]>,">")]>,
<Right lunch {Umbral: 0.5}|
[(Right 72.3177966101695,"<="),(<Right gender {Umbral: 0.5}|
[(Right 66.05172413793103,"<="),(Right 55.816326530612244,">")]>,">")]>]

*ModelMarks> errorBosqueEntrenamiento
205.45360070077692

*ModelMarks> errorBosqueValidacion
206.6371153948943
*ModelMarks>
```

Código 3.8: Salida de Modelar Calificaciones

CAPÍTULO 4

Conclusiones

En el **Capítulo 1** se han tratado algunos conceptos relativos al aprendizaje automático, en particular a los árboles de decisión, permitiendo obtener una idea general de los mismos. Se han presentado algoritmos de construcción de árboles, de poda y de construcción de modelos combinados. Además, se ha visto la aplicación de los modelos combinados al ámbito de los árboles. En el **Capítulo 2** se ha descrito la librería de Árboles de Decisión en Haskell, objetivo fundamental de este trabajo. Se ha incluido la estructura de los archivos, el código de las funciones y breves explicaciones y motivaciones de las mismas. El **Capítulo 3** muestra el uso práctico de la librería sobre algunos problemas seleccionados.

El aspecto más exigente de este proyecto ha sido el desarrollo de la librería: la estructuración de los tipos y de las funciones. Ni la implementación de los algoritmos de construcción de árboles ni la elección de los tipos creados se ha llevado a cabo con el objetivo de la optimización y uso práctico de la librería. La intención ha sido realizarlo de una manera clara y didáctica. Por este motivo, algunas de las funciones y estructuras son poco prácticas e ineficientes. Se realizan algunas búsquedas y cálculos redundantes. Por contraposición, el código es muy sencillo de entender y en muchos casos, similar al pseudocódigo. Gracias a esto la librería puede ser usada como herramienta para comprender cómo funcionan estos algoritmos.

Por tanto, la mejora principal sería relativa a la eficiencia. El uso de tipos definidos por HASKELL como `Data.Vector` y la reestructuración de las búsquedas de los valores de los atributos por posición en vez de por nombre podrían reducir sustancialmente los recursos necesitados.

A pesar de que no se pueden tratar problemas con una elevada cantidad de datos, es más que suficiente para los vistos en el Capítulo 3. Cabe recordar que entre los objetivos de este proyecto no está el de obtener un buen resultado sobre estos problemas.

Una posible futura ampliación sería la introducción de algoritmos de post-poda, que simplificasen los árboles generados. De esta forma, se obtendrían modelos con menor so-

breajuste y más aptos para la resolución de problemas más complejos. En este proyecto se han añadido dos funciones, número de hojas y profundidad de un árbol, que se podrían utilizar en la construcción de árboles con este fin de forma sencilla.

Bibliografía

[1] Aracil, J. *Introducción a la dináhhmica de sistema*, 1983.

[2] Bellman, R. *An introduction to artificial intelligence: can computers think?* , 1978.

[3] Breiman, *Classification Algorithms and Regression Trees*, 1984.

[4] Breiman, *Bagging Predictors*, 1996.

[5] Breiman, *Bias, Variance and Arcing Classifiers*, 1996.

[6] Breiman, *Random Forest*, 2001.

[7] Dietterich, T. G. *Ensemble Methods in Machine Learning*, 2000.

[8] Freund and Schapire, *A decision-theoretic generalization of on-line learning and an application to boosting. Journal of Computer and System Sciences*, 1997.

[9] Hansen and Salamon, *Neural network ensembles*, 1990.

[10] Haskell, *Data.List* . Consultado en `https://hackage.haskell.org/package/base-4.16.0.0/docs/Data-List.html`

[11] Haskell, *Data.Maybe* . Consultado en `https://hackage.haskell.org/package/base-4.16.0.0/docs/Data-Maybe.html`

[12] Haskell, *Data.Either* . Consultado en `https://hackage.haskell.org/package/base-4.16.0.0/docs/Data-Either.html`

[13] Haskell, *System.Random* . Consultado en `https://hackage.haskell.org/package/random-1.2.1/docs/System-Random.html`

[14] Haskell, *System.IO.Unsafe* . Consultado en `https://hackage.haskell.org/package/base-4.16.0.0/docs/System-IO-Unsafe.html`

[15] Haskell, *Data.Csv* . Consultado en `htsvtps://hackage.haskell.org/package/cassava-0.5.2.0/docs/Data-Csv.html`

[16] Haskell with Cassava, Consultado en `https://www.stackbuilders.com/tutorials/haskell/csv-encoding-decoding/`

[17] Kaggle, *Drug classification dataset* Consultado en `https://www.kaggle.com/prathamtripathi/drug-classification`

[18] Kaggle, *Students performance dataset* Consultado en `https://www.kaggle.com/spscientist/students-performance-in-exams/version/1`

[19] Martín Mateos, F.J. *Documentación de un TfE en LaTeX* Personal edition, 2020.

[20] Mitchell, T. M. *Machine Learning*, 1977.

[21] Norvig, P. y Russell, S. *Artificial Intelligence: A Modern Approach.*, 2010.

[22] *Poda de Error Pesimista*, Consultado en `https://numerentur.org/metodos-de-poda/`

[23] Quinlan, J. R. *Discovering rules by induction from large collections of examples*, 1979.

[24] Quinlan, J. R. *Induction of Decision Trees*, 1985.

[25] Quinlan, J. R. *C4.5: Programs for Machine Learning.*, 1993

[26] Quinlan, J. R. *Bagging, Boosting and C4.5*, 2006

[27] Sancho-Caparrini, F. *http://www.cs.us.es/ fsancho/?e=106*

[28] Shannon, C. E. *A Mathematical Theory of Communication*, 1948.