Universidad Miguel Hernández

FACULTAD DE CIENCIAS SOCIALES Y JURÍDICAS DE ELCHE

GRADO EN ESTADÍSTICA EMPRESARIAL

ÁRBOLES DE CLASIFICACIÓN: DE LO CLÁSICO A LO ÓPTIMO. DESARROLLO E IMPLEMENTACIÓN DE UNA FORMULACIÓN DE ÁRBOL ÓPTIMO.

Trabajo Fin de Grado

Autora: Ana Navarro Sellés

Tutora: Marina Leal Palazón

Curso 2021/2022

Índice

1. Resumen

En este trabajo, se aborda el desarrollo de un árbol de clasificación óptimo utilizando optimización entera mixta (MIO) y su posterior implementación en R.

Actualmente, se usan lo que se conoce como árboles de clasificación clásicos para hacer predicciones sobre un conjunto de datos que contiene una serie de variables explicativas y una variable que indica la clase de cada individuo del conjunto. Se considera que se puede seguir el proceso de predecir la clase de los individuos mediante un modelo en forma de árbol.

Este tipo de árboles parten de un nodo raíz en el que se concentran todos los individuos, que se van repartiendo a través de lo que se denomina nodos rama dependiendo del valor de las variables predictoras del individuo para, finalmente, acabar en un nodo hoja que definirá la clase que se le asigna, según la clase más común entre los individuos que hayan caído en él.

Sin embargo, estos árboles requieren de podas para conseguir evitar sobreajustes, además, se toman las decisiones de forma local, no teniendo en cuenta cómo estas decisiones influyen en el resto del modelo.

Si se considera el problema de crear un árbol como un problema de optimización, es lógico pensar en utilizar la optimización MIO, ya que se trata de un problema repleto de decisiones discretas (en qué nodo hoja cae cada observación, qué variable se usa para hacer una partición, etc.).

La MIO nos permite desarrollar un árbol en un solo paso y en que se toman las decisiones considerando cómo estas afectan al resto del árbol.

En el trabajo se ha programado un árbol óptimo que ha devuelto unos resultados correctos y que permite tomar todas las decisiones relacionadas con el árbol en un solo paso, sin necesidad de poda. Se han añadido dos nuevas restricciones válidas al planteamiento original del problema en el que se ha basado el trabajo.

2. Introducción

Los árboles de decisión son una de las herramientas más útiles y sencillas de regresión y clasificación. En este trabajo nos centraremos en los árboles de decisión utilizados para clasificación, conocidos también como árboles de clasificación. Para poder entender correctamente su funcionamiento, sus ventajas y desventajas, es importante conocer antes el concepto de 'clasificación supervisada' y todo lo relacionado con ella.

2.1. ¿Qué es la clasificación supervisada?

El aprendizaje supervisado pretende construir un modelo estadístico para predecir el valor o clase de una variable (output) utilizando una o varias variables predictoras o explicativas (inputs). Dependiendo del tipo de output, se puede distinguir entre un problema de clasificación, en caso de hacer predicciones de variables categóricas, y de regresión, si se trata de variables numéricas. Si partimos de un conjunto de individuos u observaciones de los cuales se conocen una serie de características y el valor de una variable que contiene la clase o valor numérico de cada individuo, para crear el modelo, se toma una muestra de esos individuos. Esta muestra se conoce como **"conjunto de entrenamiento"**. Una vez se ha creado el modelo, se realizan predicciones sobre los individuos que no se han incluido en el conjunto de entrenamiento, para comparar el resultado que devuelve el modelo con el dato real. A este subconjunto de datos se le llama **"conjunto de validación"**.

A continuación, se define la notación que se va a utilizar:

- $I = 1, \dots, n$: conjunto de n individuos.
- y: variable objetivo, cuyo valor se pretende predecir.
- Para cada individuo $i \in I$ se tiene un vector (x_i, y_i) donde $x_i = (x_{i1}, x_{i2}, \dots, x_{ip})$, vector de valores de las p variables explicativas e $y_i \in \{1, \dots, K\}$ son cada una de las posibles clases.

Se puede considerar como instrumento de clasificación supervisada todo aquel que ajuste un modelo que permita clasificar a un individuo i en la clase y_i utilizando el grupo de variables predictoras x_i correspondientes a ese individuo.

Un elemento importante en la clasificación supervisada son las conocidas como "**scoring functions**", ya que muchos de los métodos de este tipo de clasificación se basan en estas funciones, que son las encargadas de asignar a los individuos su clase. Por otro lado, para poder comparar las diferentes técnicas y seleccionar la que tenga un mejor rendimiento se utilizan las **técnicas de validación**.

Además, para poder comprobar cómo de bien podría clasificar a los futuros individuos el método finalmente seleccionado, se utiliza lo que conocemos como criterio de evaluación de la precisión.

2.2. Métodos de clasificación supervisada

No existe una técnica de clasificación supervisada que dé los mejores resultados en todos los casos. Por esa razón, en este apartado se presentan de forma breve diferentes métodos de clasificación supervisada y las condiciones que se tienen que dar para poder utilizar cada uno de ellos. Más adelante se explicará de forma más detallada los árboles de clasificación, que son el objeto de estudio de este trabajo. Los diferentes métodos de clasificación supervisada difieren principalmente en las asunciones que se hacen sobre los datos y el tipo de algoritmo que se construye para tratar de aproximar el clasificador.

2.2.1. Regresión Logística

La Regresión Logística modela la probabilidad de que un individuo i pertenezca a una de las dos posibles clases que pueden existir cuando se usa este método y le asigna a este individuo aquella para la cual obtenga una mayor probabilidad. El cálculo exacto de esta probabilidad se conoce como Clasificador de Bayes y la Regresión logística es una aproximación a este clasificador, al igual que otros métodos como el Análisis Discriminante Lineal o K-Nearest Neighbors.

La Regresión logística realiza esta aproximación utilizando la función logística: $\frac{1}{1+e^{-z}}$, que al estar acotada entre 0 y 1 puede ser interpretada como una probabilidad. Se trata de encontrar la función logística que mejor se ajusta a la muestra, de la cual se conoce, como ya se ha indicado anteriormente, para un conjunto de individuos, el valor de los predictores y la clase a la que pertenece cada individuo.

2.2.2. Análisis Discriminante

Usando el Análisis Discriminante nos es posible encontrar lo que llamamos "funciones discriminantes", que permiten dividir a los individuos en t regiones basándose en una serie de variables explicativas. Se asigna a un individuo la clase t si sus x_i se hallan en la región t.

El número de funciones discriminantes se calcula como el mínimo entre K -1 y p, donde K es el número de categorías que tiene la variable dependiente y p el número de variables independientes.

2.2.3. K-Nearest Neighbors

Si partimos del conjunto de variables explicativas (denominado x) asignadas a un individuo, y un número entero positivo K, este método identifica los K puntos del conjunto de entrenamiento más cercanos a x. Estos puntos se representan como N_x.

Estimamos la probabilidad para la clase k como la proporción de puntos en N_x que pertenecen a esa clase.

Finalmente, al igual que ocurría en la Regresión Logística, se asigna al individuo la clase con mayor probabilidad.

2.2.4 Support Vector Machine

El objetivo de este método es separar de forma óptima los puntos de las distintas clases utilizando hiperplanos, de manera que la distancia entre los puntos más cercanos a estos sea la máxima posible. Esta distancia se conoce como margen, y se busca el hiperplano para el que el margen es mayor.

Dependiendo del grado de exigencia, se puede contar con un margen rígido o blando. Si se exige que todas las observaciones del conjunto de entrenamiento estén correctamente clasificadas y fuera del margen, entonces se utilizará un margen rígido. Si, por el contrario, se permite que algunas observaciones estén dentro del margen o mal clasificadas, entonces se usará un margen blando.

Sin embargo, la separación entre varias clases puede no ser lineal. Para abordar este problema se utilizan los kernels, que permiten definir fronteras no lineales entre clases.

Algunos ejemplos de kernels pueden ser los kernels polinomiales de grado d o los kernels radiales.

Aunque este método se platea originalmente para conjuntos de datos en los que la variable a predecir tiene únicamente dos clases, existen extensiones para más clases. Por ejemplo, se pueden ir comparando una a una las clases y generar un hiperplano para cada par o ir comparando una clase con las $k - 1$ clases.

2.2.5 Árboles de clasificación

El conjunto de valores posibles que pueden tomar las variables independientes x_i asociadas a un individuo se divide en t regiones distintas y, a cada región, se le asigna una de las k clases. Si un nuevo individuo cae en la región t, entonces se le asignará la clase k que corresponda a esa región. Esta partición se puede representar en forma de árbol.

Los árboles de clasificación están formados por nodos y ramas. En primer lugar, se parte del nodo raíz, en el que se encuentran todos los individuos. Este nodo se divide en dos ramas que definen unas reglas de partición. Si, por ejemplo, para una variable numérica el valor de una de las variables del individuo es mayor a una cota previamente definida, entonces el individuo se moverá por una de las ramas. Si, por el contrario, el valor es menor a esa cota, entonces se moverá por la otra. Para el caso de variables categóricas, si una variable pertenece a una clase, el individuo se moverá por una de las ramas, si no pertenece a dicha clase, se moverá por la otra.

Cuando se crea el árbol usando el conjunto de entrenamiento, se asocia a cada región o nodo hoja la clase que tengan la mayoría de las observaciones que caen en ella.

Se irán haciendo divisiones con el objetivo de disminuir la "impureza" del nodo padre, considerando que un nodo es totalmente puro en el caso de que contenga observaciones únicamente de una clase.

Sin embargo, un exceso en la búsqueda de pureza de los nodos podría provocar que surgieran nodos con muy pocas observaciones y, por tanto, que se dé un sobreajuste del modelo. Para evitarlo, se definen criterios de parada como, por ejemplo, establecer un mínimo de individuos por nodo.

2.2.6 Random Forests

Los Random Forests son colecciones de árboles de clasificación que usan distintos conjuntos de entrenamiento. Muestreando el conjunto de entrenamiento original se van creando nuevos conjuntos. Se asignará al individuo la clase que se le haya asignado con más frecuencia en grupo de árboles.

Normalmente, no se utilizan las k variables para construir los árboles. Para decidir qué variables se deben usar, se puede medir la importancia de cada variable explicativa a la hora de predecir.

2.3. Scoring functions

Se tiene una scoring function o función de puntuación para cada clase. Estas funciones indican la probabilidad de que un individuo representado por las variables explicativas x_i pertenezca a cada clase.

Las scoring functions se construyen con el conjunto de entrenamiento y los futuros individuos que tengan como vector de predictores xi se clasifican como miembros de la clase en la que $y(x_i)$ tenga una puntuación mayor.

Si se da un empate, los individuos se asocian aleatoriamente a una de las clases con la puntuación máxima.

2.4. Técnicas de validación y evaluación de la precisión

2.4.1 Técnicas de validación

Existen técnicas para evaluar la precisión de predicción de cada clasificador, las que conocemos como técnicas de validación.

La técnica más simple es dividir la muestra de individuos en dos conjuntos, el conjunto de entrenamiento y el conjunto de validación. Por lo general, el primero contiene dos tercios del total de individuos y el segundo el resto de ellos. En el método hold-out, que es como se conoce este método, la evaluación del clasificador depende en gran medida de cómo se dividan los datos.

Para mejorar este método, se puede utilizar la k-folds cross-validation, donde la muestra se divide en k conjuntos de tamaño similar. Cada conjunto se utiliza para comprobar cómo se comporta el clasificador construido con el resto de la muestra. De esta forma, se obtienen k mediciones del rendimiento. Haciendo el promedio de las k mediciones, se consigue el rendimiento estimado con la k-folds cross-validation.

El valor de k dependerá del tamaño de la muestra.

Otra de las técnicas es el bootstrap, que consiste en hacer una muestra de la muestra, es decir, se toma un conjunto de observaciones de la muestra original. Con este conjunto se construye el modelo y se prueba con los datos originales. Aquellas observaciones que no aparecen en el bootstrap son llamadas observaciones OOB (out-of-bag).

Si en lugar de un conjunto de bootstrap creamos múltiples, este procedimiento se conoce como bagging. Cada uno de los bootstraps se usa como conjunto de entrenamiento y el conjunto de validación vuelven a ser los datos de la muestra original. Al igual que en la k-folds cross-validation, se hace un promedio y se obtiene una medida del error, a la que llamamos eboot.

Sin embargo, en este último método, existen observaciones comunes entre el conjunto de entrenamiento y el de validación, por lo que los resultados serán bastante optimistas. De hecho, la probabilidad de que un individuo forme parte de una muestra de bootstrap es aproximadamente de 0,632, o lo que es lo mismo, un tercio de las observaciones son observaciones OOB. Por tanto, si se hace una media de los estimadores de los individuos OOB a la que denominamos eOOB, entonces un cálculo justo del error sería $Error = 0{,}368 \times e_{OOB} + 0{,}632 \times e_{boot}$.

2.4.2 Evaluación de la precisión

La forma de medir lo bien que ha funcionado un modelo más popular es la accuracy o precisión, es decir, la proporción de individuos correctamente clasificados. Además de esta medida, también puede ser interesante conocer la proporción de observaciones cuya clase ha acertado el modelo en cada una de las distintas clases, ya que, algunas veces, el clasificador funciona mejor con una clase concreta.

Para poder saber esto, se mide la sensibilidad y la especificidad, las cuales se pueden deducir mediante una matriz de confusión: una tabla de doble entrada que compara el número de individuos predichos en cada clase con el número real de individuos en cada clase.

Si la forma de una matriz de confusión es la que aparece en la *Figura 1*, entonces las fórmulas de las medidas que se desean obtener son las siguientes:

- Precisión: $\frac{TP+TN}{TP+FN+FP+TN}$
- Sensibilidad: $\frac{TP}{TP+FN}$
- Especificidad: $\frac{TN}{FP+TN}$

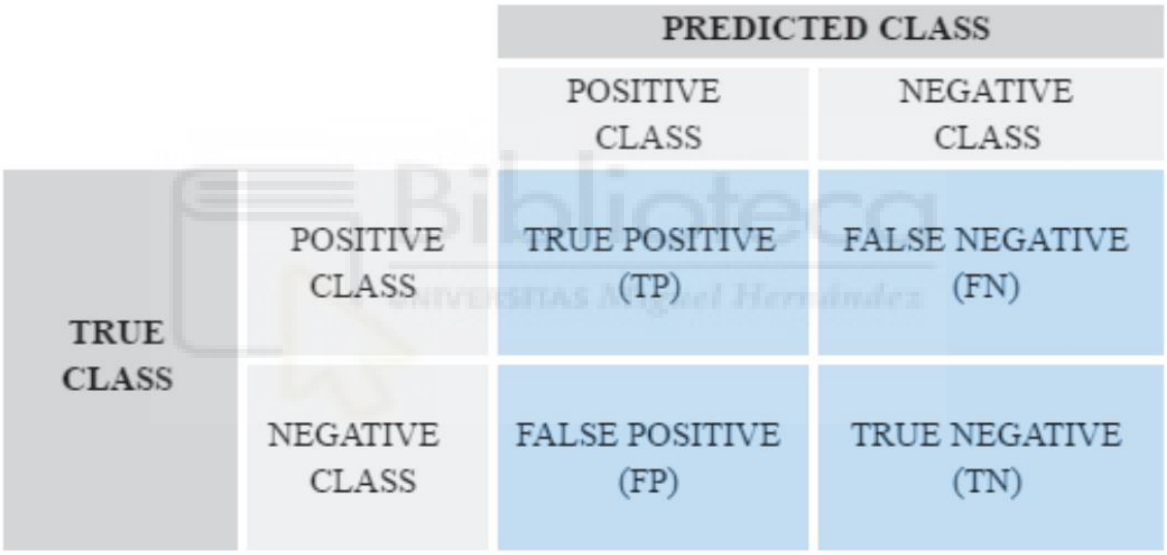

Figura 1: Matriz de confusión (Molero, 2017)

Para los problemas de clasificación binaria, es interesante calcular la F-measure, el coeficiente kappa y lo que se conoce como "Area Under the Receiver Operating Characteristic (ROC) Curve", AUC.

El coeficiente kappa muestra la concordancia entre las predicciones y las observaciones reales, teniendo en cuenta las posibles concordancias como consecuencia del azar. De esta forma, el coeficiente se calcula como $\frac{Precisión-P_{chance}}{1-P_{chance}}$, siendo $P_{chance} = \frac{(TP+FN)(TP+FP)(FP+TN)(TN+FN)}{(TP+FN+FP+TN)^2}$, la probabilidad de que exista concordancia por el azar. Cuando el coeficiente es igual a 1, la concordancia es perfecta, mientras que cuando el

coeficiente es igual a 0, la concordancia se debe al azar. Si toma valores negativos, la concordancia es menor de lo que se esperaría a causa del azar.

La F-measure representa la media armónica ponderada de la sensibilidad y la precisión:

$F_{\propto} = \frac{1}{\propto \frac{1}{Precisión} + (1-\propto)\frac{1}{Sensibilidad}}$, donde α es el peso que toma valores entre 0 y 1.

Por último, otra forma de medir cómo ha funcionado el modelo es con el AUC, o el área que queda por debajo de la curva ROC. Esta curva se genera de la siguiente manera:

$$m_{n,\theta}(x) = \{Positivo\ si\ f_{positivo}(x) \geq \theta, \qquad Negativo\ en\ caso\ contrario\}$$

θ es el umbral de discriminación entre clases. Para los diferentes valores del parámetro θ, la curva ROC muestra la sensibilidad contra (1-especificidad).

3. Estado de la cuestión y marco teórico.

Los árboles de decisión son una de las técnicas más utilizadas para resolver problemas de clasificación. Estos árboles realizan particiones recursivas del espacio de las variables x_i y asignan una etiqueta a cada partición resultante. Posteriormente, se usa el árbol para clasificar futuros puntos de acuerdo con estas particiones y etiquetas (Bertsimas y Dunn, 2017).

Entre las principales ventajas de los árboles, se encuentra su gran interpretabilidad. En ocasiones, se prefiere utilizar este método en lugar de otros más precisos, pero más difíciles de interpretar.

Los árboles de decisión se pueden aplicar a problemas de regresión y de clasificación. Los problemas de regresión buscan respuestas cuantitativas y los de clasificación, respuestas cualitativas. Para los problemas de regresión, la predicción de una nueva observación se hará en base a la media de las observaciones que se encuentren en cada región. Mientras tanto, en los problemas de clasificación, se asignará a la nueva observación la clase predominante en las observaciones que hayan caído en su misma región. En este trabajo, nos vamos a centrar en los problemas de clasificación.

Como se ha comentado anteriormente, se empieza por el nodo raíz, que se parte de forma que se consiga minimizar la impureza de sus nodos hijos. A pesar de que podría parecer lógico que se realizaran las particiones minimizando la tasa de mal clasificados, ya que es el objetivo del árbol y se calcularía simplemente como la fracción de observaciones en el nodo hoja que no pertenecen a la clase más común, esta medida no es lo suficientemente sensible para crear el árbol. Por esta razón se utilizan las medidas de pureza. Se hace una partición del nodo raíz siguiendo este criterio y se forman los nodos ramas, que son nodos intermedios sobre los que se realizan las particiones. Estas particiones vienen dadas por los parámetros a y b Para un determinado punto i, si $a^T x_i < b$, el punto seguirá la rama izquierda del nodo, si no es así, entonces seguirá la rama derecha. Cada partición tiene una sola dimensión, por lo que un elemento de a valdrá uno y todos los demás tomarán el valor cero.

El algoritmo entonces lleva a cabo una partición recursiva de las dos regiones que se crean siguiendo este hiperplano. Las particiones terminan una vez se ha alcanzado alguno de los posibles criterios de parada:

- No se puede hacer una partición sin que uno de los lados tenga al menos un mínimo número de observaciones, $Nmin$.
- Todos los puntos del nodo candidato son de la misma clase.

Por último, se generan los nodos hoja, a los cuales se les asigna una clase, normalmente la clase con mayor ocurrencia en cada nodo. De esta forma, en el futuro, se hará la predicción de la clase de un individuo dependiendo del nodo hoja en el que caiga.

Para medir la impureza se suele recurrir al Índice de Gini. Este índice mide la varianza total entre las K clases. Siendo $\hat{p}_{mk}$ la proporción de observaciones del conjunto de entrenamiento de la región m que son de la clase k. La fórmula para calcular el Índice de Gini es la siguiente:

$$G = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk})$$

Si los valores de $\hat{p}_{mk}$ toman como valor cero o uno, el índice tomará valores pequeños. Valores pequeños implican que un nodo contiene predominantemente observaciones de una sola clase.

El problema de este enfoque viene dado por su naturaleza heurística. Cada partición se realiza de forma local y aislada, sin tener en cuenta el efecto que puede tener en el resto de particiones del árbol. Esto puede provocar que se consigan árboles que no recojan las características de los datos y, por lo tanto, se obtengan unas predicciones en el futuro poco precisas.

Otro de los problemas que encontramos en los árboles clásicos es la necesidad de hacer un "prunning", es decir, una poda al árbol si queremos evitar un sobreajuste y podamos obtener árboles que nos permitan generalizar.

Esto es necesario, ya que, al estar centrado en reducir la impureza, no penaliza la complejidad del árbol. Si se estableciera una penalización de la complejidad del árbol mientras este crece y se situara en un punto demasiado elevado del árbol, se podrían perder particiones útiles a la hora de predecir que provienen de particiones no tan influyentes. El proceso de poda se realiza de abajo hacia arriba, por las particiones desde el fondo del árbol. La decisión de si se debe o no podar, viene dada por el parámetro de complejidad α. Este parámetro permite equilibrar la complejidad que implica añadir una nueva partición con el incremento que supone esta en la precisión del árbol. Cuanto mayor sea el valor del parámetro, más nodos se cortan y más pequeño resulta el árbol.

Estos problemas se podrían resolver si se pudiera crear un árbol de decisión en un solo paso. De esta manera, cada partición se realizaría teniendo en cuenta el resto de las particiones. Este árbol sería un árbol óptimo del conjunto de entrenamiento.

Hasta el día de hoy se ha estado utilizando la poda no porque se piense que se trata del mejor método para encontrar árboles precisos, sino por las dificultades que presentan otros métodos que se han tratado de emplear para crear estos árboles. Existen muchos métodos heurísticos que se han intentado utilizar para desarrollar árboles en un solo paso, como por ejemplo optimización lineal (Bennet, 1992), optimización continua (Bennet y Blue, 1996), etc. Sin embargo, los tiempos de resolución son todavía elevados.

Para poder desarrollar un método que produzca árboles de clasificación óptimos, primero se debe formalizar como problema de optimización el problema que pretenden resolver los árboles de clasificación y regresión (CART) clásicos. De esta forma, este se puede resolver de forma óptima, proporcionando una base para el método que queremos utilizar.

Para poder proponer el problema que buscan resolver los CART, se parte de un conjunto de entrenamiento con n observaciones, cada una de las cuales tiene p variables x_i y una etiqueta de clase $y_i \in \{1, \dots, K\}$, que indica a cuál de las k clases posibles pertenece. Para facilitar el problema, y sin pérdida de generalidad, se va a considerar que las variables predictoras deben ser variables numéricas y los datos están normalizados, por lo que cada $x_i \in [0,1]_p$.

En este problema existen dos parámetros, α, que controla el punto intermedio entre la complejidad y la precisión del árbol, y $Nmin$, que indica el mínimo número de observaciones que se requieren en cada nodo hoja. Buscamos un árbol, al que denominamos T, que resuelva el siguiente problema:

$$\min R_{xy}(T) + \propto |T|$$

$$s.t. N_x(l) \geq Nmin \ \ \forall l \in hojas(T)$$

donde $R_{xy}(T)$es el error de los mal clasificados del árbol T con el conjunto de entrenamiento, $|T|$ es el número de nodos rama de T y $N_x(l)$ es el número de puntos que contiene el nodo l.

Si se resolviera este problema en un solo paso, ya no sería necesaria una medida de impureza mientras construimos el árbol, ni tampoco sería necesario realizar la poda, debido a que se ha tenido ya en cuenta la complejidad formulando el problema.

4. Objetivos

Según Bertsimas y Dunn (Bertsimas y Dunn, 2017), la razón por la que no se ha llegado a desarrollar un algoritmo para árboles óptimos de decisión es que no se ha interpretado correctamente la naturaleza del problema. Construir un árbol implica una serie de decisiones discretas (hacer una partición de un nodo o sobre qué variable hacerla) y también, resultados discretos (en qué nodo hoja cae un punto o si un punto se ha clasificado correctamente).

Por lo tanto, el problema de crear un árbol de decisión óptimo debería considerarse como un problema de optimización entera mixta (MIO, Mixed Integer Optimization). La optimización MIO no ha sido tan utilizada como otros tipos ya que, a pesar de saber que muchos problemas estadísticos tienen formulaciones de este tipo, se tiene la creencia de que son intratables incluso para problemas pequeños. Esta creencia sí era cierta a principio de la década de los 70, cuando se empezaron a desarrollar los primeros métodos estadísticos de optimización continua. Sin embargo, durante los últimos años, la capacidad computacional de los solvers de problemas MIO ha aumentado considerablemente. Algunos como GUROBI (Gurobi Optimization Inc 2015b) o CPLEX (IBM ILOG CPLEX 2014) son capaces de resolver problemas de un tamaño razonable rápidamente. Además, a los avances en los solvers se les debe añadir la mejora en el hardware de los ordenadores en el mismo periodo de tiempo. De esta forma, se puede dar uso a la optimización MIO para poder formular y resolver el problema de los árboles de decisión óptimos.

El uso de un problema MIO nos permite considerar el impacto que tiene cada decisión que se tome en el árbol entero, no como en los CART en los que se toman decisiones que eran óptimas sólo de forma local, sin tener en cuenta cómo estas afectan al resto del árbol.

Dados los inconvenientes ya expuestos sobre los métodos clásicos de generación de árboles de clasificación y la naturaleza de las decisiones discretas de estos árboles, el objetivo del trabajo es la comprensión de un modelo MIO para resolver el problema de creación de un árbol de clasificación, así como su posterior implementación en R.

5. Metodología y resultados

5.1. Desarrollo teórico del problema

5.1.1. Desarrollo

Para empezar, si buscamos construir un árbol óptimo de máxima profundidad D, entonces el número de nodos será igual a $T = 2^{(D+1)} - 1$, a los cuales haremos referencia con el índice $t = 1, \dots, T$. Como ejemplo concreto, si tenemos un árbol como el de la *Figura 3*, con una profundidad máxima de 2, ya que tenemos dos niveles de particiones, entonces su número de nodos según la fórmula debería ser igual a 7, que es exactamente el número de nodos que aparecen.

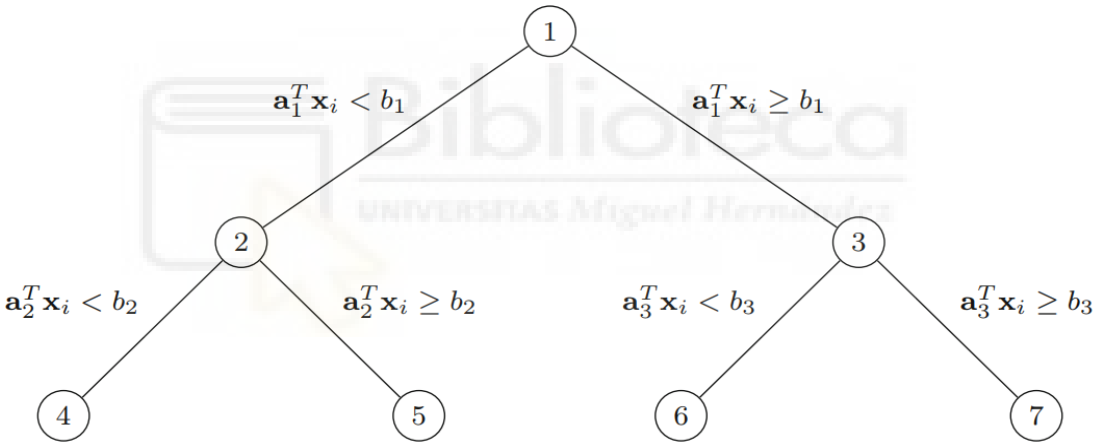

Figura 2: Árbol de máxima profundidad D = 2, (Bertsimas y Dunn, 2017)

Otro parámetro para tener en cuenta es $p(t)$, el nodo padre de t. Por otro lado, definimos $A(t)$, que denota el conjunto de nodos ancestros de t, $A_L(t)$ y $A_R(t)$, que son los conjuntos de nodos ancestros del nodo t que han tenido que seguir su rama izquierda o derecha respectivamente del nodo para poder llegar hasta t. Por tanto, $A(t) = A_L(t) \cup A_R(t)$.

Los nodos, como se ha mencionado anteriormente, se dividen en primer lugar en nodos rama $t \in TB = \{1, \dots, \lfloor \frac{T}{2} \rfloor\}$ que aplican una partición de la forma $a^T x < b$. Aquellos

puntos que cumplan con ella seguirán la rama izquierda del nodo, y los que no la derecha. Cabe tener en cuenta que, al igual que cuando se ha planteado el problema de optimización a resolver por los CART, aquí también vamos a centrarnos únicamente en variables predictoras numéricas y de la forma $x_i \in [0,1]_p$.

Además de los nodos rama, están los nodos hoja $t \in TL = \{\left\lfloor \frac{T}{2} \right\rfloor + 1, \dots, T\}$, que predicen la clase de cada una de las observaciones que caigan en ellos.

Para la primera restricción del problema, se necesitan las variables $a_{jt} \in R_p$ y $b_t \in R$ para los nodos $t \in TB$. Cada una de las particiones que se hagan va a estar formada únicamente por una variable x_i. Esto se consigue definiendo a_{jt} como una variable binaria. De esta forma, cuando a_{jt}=1 significa que se utiliza la variable predictora j al realizar la partición en el nodo t. Asimismo, se debe dar la opción de que un nodo no llegue a partirse. Para ello, se define la variable binaria d_t, que toma el valor uno si se aplica una partición sobre el nodo t. Si para el nodo t, d_t = 0, entonces debe ocurrir que a_{jt} = 0 y b_t = 0, esto también debe quedar indicado en el modelo y obligar a todas las observaciones a tomar la rama derecha del nodo, porque la restricción izquierda 0 < 0 nunca va a ser cierta.

Para conseguir esto, aplicamos las siguientes restricciones:

$$\sum_{j=1}^{p} a_{jt} = d_t, \qquad \forall t \in TB \quad (2)$$

$$0 \leq b_t \leq d_t, \qquad \forall t \in TB \quad (3)$$

$$a_{jt} \in \{0,1\}, \qquad j = 1, \dots, p, \forall t \in TB \quad (4)$$

La primera ecuación permite determinar que si d_t = 0, entonces ninguna de las variables a_{jt} tomará el valor 1. En cambio, si d_t = 1 solo uno de los valores de a_{jt} puede ser igual a uno.

La segunda inecuación es válida para b_t ya que se asume que $x_i \in [0,1]_p$ y sabemos que a_{jt} tiene un elemento que vale 1 si d_t = 1. Por tanto, siempre es verdad que $0 \leq a_{jt}^T x_i \leq d_t$, así que solo debemos considerar valores de b_t en este mismo intervalo.

En la siguiente restricción forzamos a que a un nodo rama no se le pueda aplicar una partición si no se le ha aplicado una a su nodo padre. Esta restricción afecta a todos los nodos $t \in TB$, pero no al nodo raíz debido a que este no tiene padre.

$$d_t \leq d_{p(t)}, \forall t \in TB\ P\backslash\{1\} \quad (5)$$

De esta forma, reforzamos la estructura jerárquica del árbol.

Las próximas restricciones nos van a permitir saber en qué nodos hoja se sitúan las observaciones y, por tanto, los errores en la clasificación.

Para ello, en primer lugar, se deben definir dos nuevas variables: z_{it}, que es una variable binaria que toma el valor uno si el punto i está en el nodo t, y l_t, otra variable binaria que vale uno en caso de que el nodo t contenga algún punto. Ambas se definen para $t \in TL$, porque los nodos hoja son aquellos en los que se clasifican las observaciones una vez han terminado las particiones. Se definen las restricciones de la siguiente manera:

$$z_{it} \leq l_t, \qquad \forall t \in TL, i = 1, \ldots, n \quad (6)$$

$$\sum_{i=1}^{n} z_{it} \geq Nmin \cdot l_t, \qquad \forall t \in TL \quad (7)$$

$Nmin$ es el parámetro que se define para indicar el número mínimo de puntos que debe tener cada nodo hoja. Con la primera restricción, nos aseguramos de que si algún punto i se encuentra en el nodo t y, por lo tanto, z_{it} = 1, entonces l_t tenga que valer uno. Esto sería correcto ya que el nodo t sí tendría un punto. Sin embargo, si solo definimos esta restricción, z_{it} puede tomar el valor 0 y l_t puede valer tanto 0 como 1.

Con la segunda restricción conseguimos que esto no ocurra. Si algún punto cae en el nodo t, entonces debe haber al menos $Nmin$ puntos. Pero, si la suma de las z_{it} para todos los puntos i es igual a cero, entonces l_t = 0.

Por otro lado, se debe forzar a que cada punto quede asignado a una sola hoja:

$$\sum_{t \in TL} z_{it} = 1, i = 1, \ldots, n \quad (8)$$

Si para cada individuo i, solo una de las variables z_{it} = 1 y, por tanto, i queda asignado a un único nodo hoja.

El objetivo del siguiente par de restricciones es forzar al modelo a realizar las particiones de la forma en la que lo requiere la estructura del árbol cuando se asignan individuos a los nodos hoja.

$$a_m^T x_i \; < \; b_m \; + \; M_1 \, (1 \; - \; z_{it}) \, , i \; = \; 1, \ldots , n, \qquad \forall t \in TL, \qquad \forall m \; \in \; A_L(t) \;\; (9)$$

$$a_m^T x_i \; \geq \; b_m \; - \; M_2 \, (1 \; - \; z_{it}) \, , i \; = \; 1, \ldots , n, \qquad \forall t \in TL, \qquad \forall m \; \in \; A_R(t) \;\; (10)$$

La primera restricción está definida para aquellos nodos que forman parte del conjunto de ancestros de t que han seguido su rama izquierda para llegar hasta t y la segunda para los que siguen su rama derecha.

M_1 y M_2 son parámetros que se definen con valores muy grandes. Si nos situamos en la primera restricción y z_{it}= 1, por lo que i está en el nodo t, M_1 queda multiplicado por cero. De esta forma, la restricción queda con la forma $a_m^T x_i \; < \; b_m$ sin ninguna alteración. En cambio, si z_{it}= 0, se le suma un valor muy grande a la parte derecha de la restricción. Esto provoca que la restricción no afecte al modelo, ya que $x_i \; \in [0,1]_p$ y los valores de la parte izquierda de la restricción siempre van a ser menores al valor de b_m más el parámetro M_1. Lo mismo ocurre con la segunda restricción.

La restricción (9) tiene un pequeño inconveniente, se trata de una inecuación estricta, que no se admite en los solvers de MIO. Si se añade una pequeña constante ϵ, se puede cambiar la inecuación para que no sea estricta.

$$a_m^T x_i \; + \; \epsilon \leq \; b_m \; + \; M_1 \, (1 \; - \; z_{it}) \, , i \; = \; 1, \ldots , n, \qquad \forall t \in TL, \qquad \forall m \; \in \; A_L(t) \;\; (11)$$

Se debe encontrar el valor de ϵ que sea lo más grande posible sin afectar a la consistencia del problema. Si definimos un ϵ_j para cada de las j. El mayor valor válido es aquel que viene dado por la distancia más pequeña distinta a cero entre valores adyacentes de cada una misma j.

$$\epsilon_j = \min \left\{ x_j^{(i+1)} - x_j^{(i)} \middle| x_j^{(i+1)} \neq x_j^{(i)}, i = 1, \ldots, n-1 \right\}$$

donde $x_j^{(i)}$ es el i-ésimo mayor valor en la j-ésima característica. Los resultados se pueden usar para dar valor a ϵ en la restricción. El valor de ϵ_j viene determinado por la característica j que se esté utilizando en cada partición.

$$a_m^T(x_i + \epsilon) \leq b_m + M_1(1 - z_{it}), i = 1, \ldots, n, \quad \forall t \in TL, \forall m \in A_L(t) \quad (12)$$

Asimismo, se pueden utilizar los ϵ_j para definir los valores de los parámetros M_1 y M_2. Dado que el máximo valor que puede tomar la expresión $a_t^T(x_i + \epsilon) - b_t$ es $1 + \epsilon_{max}$, donde $\epsilon_{max} = max_j\{\epsilon_j\}$, ponemos definir $M_1 = 1 + \epsilon_{max}$. De la misma forma, el mayor valor que puede tomar $b_t - a_t^T x_i$ es uno, por lo que $M_2 = 1$. Por tanto, las dos restricciones quedan finalmente de la siguiente forma:

$$a_m^T(x_i + \epsilon) \leq b_m + (1 + \epsilon_{max})(1 - z_{it}), i = 1, \ldots, n, \forall t \in TL, \forall m \in A_L(t) \quad (13)$$

$$a_m^T x_i \geq b_m - (1 - z_{it}), i = 1, \ldots, n, \forall t \in TL, \forall m \in A_R(t) \quad (14)$$

Las últimas restricciones guardan relación con el objetivo del problema: minimizar el coste de los mal clasificados, de forma que una predicción incorrecta tenga coste 1 y una correcta, coste 0.

Con el conjunto de entrenamiento, definimos la matriz Y_{ik}, donde:

$$Y_{ik} = \{+1 \; si \; y_i = k, -1 \; en \; caso \; contrario, k = 1, \ldots, K, i = 1, \ldots, n\}$$

Definimos N_{kt} como el número de puntos de clase k en el nodo t y N_t como el número total de puntos del nodo t.

$$N_{kt} = \frac{1}{2}\sum_{i=1}^{n}(1 + Y_{ik})z_{it}, k = 1, \ldots, K, \forall t \in TL \quad (15)$$

$$N_t = \sum_{i=1}^{n} z_{it}, \forall t \in TL \quad (16)$$

En la restricción (15), para que se incluya un punto i dentro del sumatorio, este tiene que pertenecer a la clase k y haber caído en el nodo t, como bien indica la variable N_{kt}. Tanto si Y_{ik} = -1, porque i no pertenece a la clase k, como si z_{it} = 0, porque i no ha caído en el nodo t, el valor en el sumatorio para el individuo i es igual a cero. Para la restricción (16),

se añade un uno al sumatorio por cada individuo i que se asigna al nodo t, resultando el número total en ese mismo nodo.

A continuación, procedemos a asignar una etiqueta a cada nodo hoja del árbol, para posteriormente hacer las predicciones, y lo definimos como $c_t \in \{1, \dots, K\}$.

$$c_t = arg\ max_{k=1,\dots,K}\{N_{kt}\} \quad (17)$$

Como ya se ha mencionado, se suele asignar a cada nodo la etiqueta más común de entre todas las observaciones que contiene el nodo, eso es lo que hace la restricción (17). Sin embargo, esta restricción no se puede incluir de esta forma en un solver, por esa razón, se crean las siguientes restricciones.

Para seguir las predicciones de cada nodo, creamos una nueva variable binaria c_{kt}, que toma valor 1 si $c_t = k$.

$$\sum_{k=1}^{K} c_{kt} = l_t, \forall t \in TL \quad (18)$$

De esta manera, si l_t = 1 y, por lo tanto, el nodo t contiene algún punto, para ese nodo t alguna k hará que c_{kt} = 1.

Por último, se define una variable L_t, que hace referencia a la pérdida óptima por mal clasificados para cada nodo t. L_t se calcula como la diferencia entre el número de puntos que haya en el nodo y el número de puntos de la clase más común en ese nodo.

$$L_t = N_t - max_{k=1,\dots,K}\{N_{kt}\} = min_{k=1,\dots,K}\{N_t - N_{kt}\} \quad (19)$$

Sin embargo, esta no es una ecuación lineal, por lo que se puede linealizar con las últimas restricciones:

$$L_t \geq N_t - N_{kt} - M(1 - c_{kt}), k = 1, \dots, K, \forall t \in TL \quad (20)$$
$$L_t \leq N_t - N_{kt} + Mc_{kt}, k = 1, \dots, K, \forall t \in TL \quad (21)$$
$$L_t \geq 0, \forall t \in TL \quad (22)$$

Como sucedía en las restricciones (9) y (10), M es una constante lo suficientemente grande como para dejar la restricción inactiva dependiendo del valor de c_{kt}. Un valor válido puede ser, en este caso, n.

Por lo tanto, el coste de los mal clasificados es $\sum_{t \in TL} L_t$ y la complejidad del árbol, que viene dada por el número de particiones, es decir, $\sum_{t \in TB} d_t$. Normalizamos la clasificación errónea dividiendo entre $\hat{L}$, que representa el número de puntos que pertenecen a la clase más popular del conjunto de datos. De esta manera, el efecto de α, es decir, del parámetro permite encontrar el equilibrio entre la complejidad que supone agregar una nueva partición y el aumento que implica esta en la precisión del árbol, no depende del número de datos.

El objetivo del problema es minimizar el coste de una clasificación errónea teniendo en cuenta la complejidad y eso se refleja en la función objetivo:

$$\min \ \frac{1}{\hat{L}} \sum_{t \in TL} L_t + \alpha \sum_{t \in TB} d_t \quad (23)$$

La mayor complejidad de este problema recae en el elevado número de variables binarias.

5.1.2. Errores en el artículo original

El artículo del que se ha obtenido el planteamiento teórico del problema ha sido "Optimal classification trees." de Bertsimas, D., & Dunn, J. (2017). Sin embargo, se han localizado una serie de errores que han dificultado la implementación. Se presentan a continuación:

- En las restricciones (6) y (7) se define $t \in TB$ cuando debería ser $t \in TL$, ya que las variables z_{it} y l_t se definen para los nodos hoja. Más tarde, cuando se presenta el problema entero este fallo se corrige.
- En las restricciones (9), (10), (11) y (12) las t deberían ser $t \in TL$, ya que la z_{it} está definida para TL. Este fallo se repite cuando se definen las restricciones (13) y (14) en el modelo completo.
- Tanto en las restricciones (9) y (10) como en la (13) y (14) en el modelo final, se debería usar b_m y no b_t, debido a que b se define para $t \in TB$ y no TL.
- En la restricción (6) falta indicar que la restricción es para todo $i = 1, \ldots, n$.

5.2. Implementación y desarrollo en R

Se va a programar un árbol óptimo en R utilizando la librería lpSolve, por lo tanto, se va a explicar la librería que se va a utilizar, definir la notación y detallar el proceso de implementación.

5.2.1. lpSolve

La función lp del paquete lpSolve permite modelar y resolver problemas de optimización lineal y entera. Para cargar la librería se debe escribir el siguiente código:

```
library("lpSolve")
```

lp se usa de la siguiente manera:

lp (direction = "min", objective.in, const.mat, const.dir, const.rhs, transpose.constraints = TRUE, int.vec, presolve=0, compute.sens=0, binary.vec, all.int=FALSE, all.bin=FALSE, scale = 196, dense.const, num.bin.solns=1, use.rw=FALSE)

A continuación, detallamos los parámetros de la función:

- direction: Cadena de caracteres que describe la dirección del problema. Puede ser “min” o “max”, aunque por defecto es “min”.
- objective.in: Vector de coeficientes de la función objetivo.
- const.mat: Matriz de coeficientes numéricos de las restricciones. Cada fila es una restricción y cada columna, una variable (a no ser que transpose.constraint = FALSE).
- const.dir: Vector de cadena de caracteres que devuelve la dirección de la restricción. Los valores que puede tomar cada elemento del vector son "<", "<=", "=", "==", ">" o ">=".
- const.rhs: Vector de valores numéricos para la parte derecha de las restricciones.
- transpose.constraints: Por defecto, cada fila de la matriz const.mat es una restricción y cada columna, una variable. En el caso de que se construya una matriz donde las restricciones vayan de columna en columna se puede igualar transpose.constraints a FALSE.
- int.vec: Vector que determina los índices de las variables enteras.

- presolve: Por defecto es 0, cualquier otro valor que no sea 0 significa que sí. Hasta el momento se ignora.
- compute.sens: Por defecto es 0, cualquier otro valor que no sea 0 significa que sí.
- binary.vec: Vector que determina los índices de las variables binarias.
- all.int: Valor lógico que indica que todas las variables son enteras. Por defecto es FALSE.
- all.bin: Valor lógico que indica que todas las variables son binarias. Por defecto es FALSE.
- scale: Valor entero que usa lpSolve para escalar. Por defecto es 196 y si se quiere no escalar se pone un 0.
- dense.const: Array de tres columnas de restricciones densas. Este elemento se ignora si se ha añadido un const.mat. Si no, las columnas son número de restricciones, número de columnas y valor. Debe haber una fila por cada entrada en la matriz de restricciones que no sea cero.
- num.bin.solns: Valor entero. En el caso de que all.bin = TRUE, el usuario puede solicitar como mucho num.bin.solns soluciones.
- use.rw: Se utiliza para intentar solucionar un error existente. El valor por defecto es FALSE, sin embargo, si num.bin.solns > 1 se recomienda cambiar el valor a TRUE.

5.2.2. Parámetros y conjuntos

A continuación, se definen los parámetros que se van a emplear para poder resolver el problema.

D: profundidad del árbol.

Tnodos: número de nodos del árbol.

TB: número de nodos rama.

TL: número de nodos hoja.

p: número de variables explicativas.

n: número de observaciones.

Nmin: número mínimo de observaciones que deben caer en cada nodo hoja.

K: número de clases en el conjunto de datos.

AL: conjunto de nodos ancestros de t que han seguido su rama izquierda para poder llegar al nodo t.

AR: conjunto de nodos ancestros de t que han seguido su rama derecha para poder llegar al nodo t.

Las matrices AL y AR se han generado con un bucle.

```
for(i in 1:D){
  aux1=0
  aux2=0
  for(j in 2**i:(2**(i+1)-1)){
    if(aux1==0){ #Su predecesor está en AL
      for(ii in 1:(i-1)){
        AL[j,ii]=AL[2**(i-1)+aux2,ii]
        AL[j,i]=2**(i-1)+aux2

        AR[j,ii]=AR[2**(i-1)+aux2,ii]
      }

    }else{#Su predecesor está en AR
      for(ii in 1:(i-1)){
        AR[j,ii]=AR[2**(i-1)+aux2,ii]
        AR[j,i]=2**(i-1)+aux2

        AL[j,ii]=AL[2**(i-1)+aux2,ii]
      }
    }

    aux1=aux1+1
    if(aux1==2){
      aux2=aux2+1
      aux1=0
    }
```

```
  }
}
```

Para definir las dos matrices anteriores se han determinado fórmulas recursivas que calculan los antecesores izquierda y derecha, respectivamente. Estas fórmulas se pueden observar dentro de los bucles anidados. Cada una de estas matrices tiene tantas filas como nodos tiene el árbol, y en cada fila, es decir, para cada nodo, se indica, por columnas, los números de los nodos predecesores. Si hay menos nodos predecesores que columnas en la matriz, se completa la matriz con ceros.

Y: matriz de n filas y K columnas que toma valor 1 si el individuo $i, i = 1, \ldots, n$, es de la clase $k, k = 1, \ldots, K$.

```
Y = matrix(0,nrow = n,ncol = K)
for (i in 1:n){
  for (k in 0:(K-1)){
    if (datos[i,ncol(datos)]== k){
      Y[i,k+1]=1
    }
    else {
      Y[i,k+1]=-1
    }
  }
}
```

Si se tiene en cuenta que la primera clase toma el valor 0, k debe comenzar desde el 0 y llegar al $K - 1$ para que se cuenten K clases. Con el if se va comprobando si la clase de cada individuo del conjunto de entrenamiento (que se almacena en la última columna) es igual al valor de k. Si es así, a la matriz Y en la fila correspondiente a ese individuo y a la columna correspondiente a esa clase, a la que se le suma 1 porque la clase 0 está en la columna 1 de Y, se le asigna un 1. En caso contrario, se le asigna un -1.

5.2.3. Variables

Como se ha mencionado, uno de los elementos de la función lp del paquete lpSolve es const.mat, una matriz en la que cada fila es una restricción y cada columna, una variable. Por lo tanto, es realmente importante definir el orden en el que van a aparecer las variables en esa matriz para, más tarde, poder interpretar la solución del problema.

Por esa razón, el siguiente paso que se va a dar, consiste en definir las variables en el orden que se va a seguir mientras se explica la forma en la que se han definido.

Cada una de las variables se ha podido definir de forma más sencilla gracias al uso de contadores, que permiten conocer la posición en la matriz de restricciones de la columna en la que se incluye esa variable.

- $a_{jt}\,, j = 1,\ldots,p\,, t \in TB$

Para la variable a_{jt} no ha sido necesario definir un contador, ya que es la primera de las variables que se van a incluir y comienza en la primera posición.

- $b_t\,,\ t \in TB$

Para esta variable, debemos tener en cuenta que el número de variables a_{jt} que hay en el problema es de $p \times TB$. Así, definimos:

```
contb<- p*TB
```

Por tanto, cuando hagamos un bucle que permita calcular los coeficientes de las variables b_t y que comience en $t = 1$, solo habremos de indicar que b_1 está en la columna $p \times TB + 1$, osea, $contb + t$ de la matriz de restricciones.

- $d_t\,,\ t \in TB$

De la misma forma ocurre con el resto de variables. Para alcanzar la variable d_1, tienen que haber pasado las $p \times TB$ variables a_{jt} y las TB variables b_t. Por esa razón, se define el contador de d_t como:

```
contd<-p*TB + TB
```

- $z_{it}\,,\ i = 1,\ldots,n\,,\ t \in TL$

Al igual que con las variables anteriores, se crea un contador que indica en qué columna de la matriz de restricciones empiezan las variables z_{it}. Al contd hay que sumarle las TB variables d_t para obtener el contador:

```
contz<-p*TB + 2*TB
```

- $l_t\ ,\ t \in TL$

Esta vez, para contar cuántas variables z_{it} aparecen y, por tanto, cuándo aparece l_t, se debe multiplicar el número de individuos por el número de nodos hoja.

```
contl<-p*TB + 2*TB + n*TL
```

- $N_{kt}\ , k = 1, \ldots, K\ ,\ t \in TL$

Para poder saber dónde empieza la variable N_{kt} hay que sumar al contador de l_t las TL variables l_t.

```
contNkt <- p*TB + 2*TB + n*TL + TL
```

- $N_t\ ,\ t \in TL$

El siguiente paso es crear el contador de N_t. Se deben tener en cuenta las $K \times TL$ variables N_{kt}.

```
contNt <-p*TB + 2*TB + n*TL + TL + K*TL
```

- $c_{kt}\ ,\ k = 1, \ldots, K\ \ t \in TL$

Creamos el contador de c_t sumando al anterior las TL variables N_t.

```
contc<-p*TB + 2*TB + n*TL + TL + K*TL + TL
```

- $L_t\ ,\ t \in TL$

Para el contador de la última variable, se suman las $K \times TL$ variables c_t anteriores.

```
contLt <- p*TB + 2*TB + n*TL + TL + K*TL + TL + K*TL
```

5.2.4. Variables binarias y enteras

En el apartado en el que se ha explicado el funcionamiento de la librería lpSolve, se ha definido el parámetro binary.vec. Para incluir la posición de las variables binarias en la resolución del problema usando la función lp, se ha creado un vector de la siguiente manera:

```
#ajt y dt
bin1=c(rep(0,p*TB + TB))
for(i in 1:(p*TB)) bin1[i]=i
for(i in 1:TB) bin1[(p*TB)+i]=p*TB + TB + i


#zit y lt
bin2=c(rep(0,n*TL + TL))
for (i in 1:(n*TL)) bin2[i]=p*TB + 2*TB + i
for (i in 1:TL) bin2[(n*TL)+i]=p*TB + 2*TB + n*TL + i

#ckt
bin3=c(rep(0,K*TL))
for (i in 1:(K*TL)) bin3[i]=p*TB + 2*TB + n*TL + TL + K*TL + TL + i


#Vector variables binarias

bin=c(bin1,bin2,bin3)
```

Se ha definido por partes para que fuera más sencillo de entender y, al final, se han unificado los tres vectores en uno. Por ejemplo, para las variables a_{jt} y d_t se ha creado el vector bin1, un vector cuyos elementos toman el valor 0 y cuya dimensión es igual a $p \times TB$ (el número de variables a_{jt}) más TB (el número de variables d_t).

El primer bucle es el asociado con a_{jt}. Es el más sencillo de realizar porque lo que se está indicando es que los primeros $p \times TB$ elementos de bin1 van a tomar valores de 1 hasta $p \times TB$. Sin embargo, se debe recordar que la posición de las variables d_t no es exactamente posterior a a_{jt} sino que está la variable b_t entre ambas.

Lo que se consigue con el segundo bucle es que los TB elementos de bin1 después de $p \times TB$ van a tomar como valor la primera posición que haya después de $p \times TB + TB$, osea, la posición de d_1.

Con el resto de las variables binarias, la lógica sería la misma. Finalmente, se unen los tres vectores en un único vector que será el que se le dé a la función lp.

Además, debemos tener en cuenta N_{kt} y N_t, cuyo valor hace referencia a un número de puntos, por lo que tiene que ser entero. Creamos un vector de la misma forma que hemos hecho con las variables binarias:

```
#Variables enteras Nkt y Nt

enteras=c(rep(0,K*TL + TL))
for (i in 1:(K*TL)) enteras[i]=p*TB + 2*TB + n*TL + TL +i
for(i in 1:TL) enteras[(K*TL)+i]=p*TB + 2*TB + n*TL + TL + K*TL+i
```


5.2.5. Restricciones

Otro de los elementos necesarios para resolver el problema con la función lp es la matriz de restricciones. Esa matriz ha sido nombrada como A y se ha generado de esta forma:

```
A=matrix(0,nrow=TB+TB+TB+ TB-1 + TL*n + TL + n + n*TL*D + K*TL + TL + TL + K*TL + K*TL + TL,ncol=p*TB + 2*TB + n*TL + TL + K*TL + TL + K*TL + TL)
```

Se trata de una matriz que se ha rellenado con ceros, en la que el número de filas es igual al número de restricciones totales del problema y el número de columnas igual al número de variables. En cuanto al número de restricciones, el porqué de esos valores viene dado, en algunos casos, por las restricciones explicadas anteriormente o, en otros casos, por la forma en la que se han programado las restricciones. A medida vayan apareciendo las restricciones, se irán sumando los valores que conforman ese total.

Por otro lado, el número de variables es simple de calcular gracias a los contadores de las variables que se han ido programando. El último contador que se había definido era el de la variable L_t,

```
contLt <- p*TB + 2*TB + n*TL + TL + K*TL + TL + K*TL.
```

Para poder contar el número de variables, solamente es necesario sumar las TL últimas variables.

Esta matriz A se rellena a medida que se programen las restricciones.

Además de esta matriz, lp admite un vector que muestre las direcciones de las diferentes restricciones y otro vector en el que aparezca las partes derechas de las misma.

```
signos=c(rep('<=', TB+TB+TB+ TB-1 + TL*n + TL + n + n*TL*D + K*TL + TL
+ TL + K*TL + K*TL + TL))
b=c(rep(0,TB+TB+TB+ TB-1 + TL*n + TL + n + n*TL*D + K*TL + TL + TL +
K*TL + K*TL + TL))
```

El vector signos contiene tantos símbolos '<=' como restricciones haya. Por defecto, se ha seleccionado el símbolo '<=', que se cambiará si la restricción lo requiere. Lo mismo sucede con el vector b, que está formado por una cantidad de ceros igual a la del número de restricciones.

Si suponemos que existe un vector X que denota todas las variables, entonces las restricciones son de la forma AX # b, donde # es el vector signos.

Para facilitar la programación, se ha definido un contador de restricciones al que inicialmente se ha dado el valor cero.

```
contadorR<-0
```

Este contador permite llevar un seguimiento de las restricciones que se van generando. Además, nos ayuda a la hora de seleccionar en qué fila debe comenzar cada nueva restricción, ya que será la fila siguiente a la que guarde el valor de contadorR.

- *Restricción 2*

Con la finalidad de comprender más fácilmente la programación de las restricciones, estas se van a ir recordando en su formato teórico a la par que se muestra el correspondiente código.

$$\sum_{j=1}^{p} a_{jt} = d_t, \qquad \forall t \in TB \quad (2)$$

Esta restricción se ha definido:

```
for (t in 1:TB){
  contadorR<-contadorR+1
  for (cont in 0:(p-1)){
    A[contadorR,cont*TB + t]=1
  }
  A[contadorR,contd + t]=-1
  b[contadorR]=0
  signos[contadorR]='='
}
```

El primer bucle hace referencia a todas las $t \in TB$ que se recorren y, por tanto, este es el que marca el número de restricciones que van ligadas a la restricción 2. Se suma uno al contador por cada restricción, es decir, por cada t. En el segundo bucle aparece el sumatorio de las $, j = 1, \ldots, p.$ Para definir esta restricción, se debe tener en cuenta que se ha considerado que las variables a_{jt} sigan el orden a_{11}, a_{12}, a_{13}, etc. En cambio, en esta restricción, si $t = 1$, los elementos del sumatorio serán a_{11}, a_{21}, a_{31}, etc. Para poder aplicar esto a la programación, el bucle comienza a contar en cero, para que así la primera columna de la matriz A, a la que se le asigne un uno sea la $0 \times TB + 1$, osea, la primera variable a_{11}. La siguiente columna que se debe seleccionar debe estar $TB + 1$ posiciones por delante de a_{11}, cuando ya han pasado todas las variables con $j = 1$ y comienza $j = 2$. Como el siguiente valor que toma el cont que se ha determinado es el uno, entonces la columna a la que se otorga el coeficiente uno es la $1 \times TB + 1$, en la que se ha posicionado la variable a_{21}. Este razonamiento toma una gran importancia debido a que se ha utilizado en muchos de los planteamientos de las restricciones.

Si salimos del segundo bucle, aparece la asignación que se le hace a las variables d_t. Aunque en la restricción teórica aparecen en el lado derecho de la restricción, para mayor simplicidad se han pasado al lado izquierdo. De esta forma, en lugar de establecer sus correspondientes coeficientes como uno, se han fijado como menos uno.

Como se puede apreciar, se ha usado el contadorR para ir siguiendo el número de fila en el que se encuentra cada restricción y el contd para nada más tener que sumar la t para situarnos en la posición de la d_t que corresponda.

De esta forma, en el lado derecho de la restricción y, por tanto, el valor que se le da al vector b de esas restricciones es cero.

El signo se cambia, ya que en esta restricción es un '='.

- *Restricción 3*

$$0 \leq b_t \leq d_t\,, \qquad \forall t \in TB \quad (3)$$

Esta restricción se puede dividir en dos partes: la primera, $0 \leq b_t$ y la segunda, $b_t \leq d_t$.

Para la primera parte, el código es:

```
for (t in 1:TB){
  contadorR<-contadorR+1
  A[contadorR,contb+t]=1
  b[contadorR]=0
  signos[contadorR]='>='
}
```

Para la segunda:

```
for (t in 1:TB){
  contadorR<-contadorR+1
  A[contadorR,contb +  t]=1
  A[contadorR,contd + t]=-1
  b[contadorR]=0
}
```

Para la primera parte, se crea un bucle con TB iteraciones. Usamos el contador de las variables b_t y en las columnas correspondientes asignamos un coeficiente de valor uno.

Se cambia el signo a '≥'. En la programación de la segunda parte ocurre lo mismo. Además, se pasa la variable d_t al lado izquierdo de la restricción, por lo que el valor que se le adjudica es de menos uno.

- *Restricción 4*

$$a_{jt} \in \{0,1\}, \quad j = 1,\ldots,p\,, \forall t \in TB \quad (4)$$

La restricción 4 determina que las variables a_{jt} deben ser binarias. Esta restricción no es necesario programarla debido a que cuando se ha creado el vector de variables binarias se ha incluido a las a_{jt}. La función lp tendrá en cuenta que estas variables son binarias a la hora de resolver el problema sin tener que programar ninguna restricción que lo señale.

- *Restricción 5*

$$d_t \leq d_{p(t)}, \forall t \in TB\, P\backslash\{1\} \quad (5)$$

```
for (i in 1:(D-1)){
  aux1=0
  aux2=0
  for(t in 2**i:(2**(i+1)-1)){
    contadorR<-contadorR+1
    A[contadorR,contd + t]=1
    A[contadorR,contd+2**(i-1)+aux2]=-1
    b[contadorR]=0
    aux1=aux1+1
    if(aux1==2){
      aux2=aux2+1
      aux1=0
    }
  }
}
```

La dificultad que conlleva esta restricción está en averiguar cuál es el nodo padre de cada nodo $t \in TB$. Imaginemos un árbol cuya máxima profundidad es igual a dos: en el primer nivel de profundidad del árbol, lo consideramos el nivel cero, solo se encontraría el nodo raíz, es decir, habría $2^0 = 1$ nodos.

En el nivel uno, aparecerían desde el nodo número $2^1 = 2$ al nodo número $2^{(1+1)} - 1 = 3$.

De esta forma, se puede crear una regla que nos permita saber que los nodos que se encuentran en el nivel $i = 1, \ldots, D-1$, empezando por el nivel uno ya que en la restricción no se tiene en cuenta el nodo raíz, son los nodos del 2^i al $2^{(i+1)} - 1$.

Asimismo, se crean dos variables auxiliares:

aux1: un nodo solo puede tener dos particiones. Esta variable cuenta cuántas particiones se han efectuado sobre un nodo, en concreto, suma uno por cada nodo hijo que tiene un nodo. En el momento en el que un nodo tiene dos particiones, aux1 vuelve a valer cero y se pasa al nodo siguiente.

aux2: esta variable lleva el seguimiento del nodo padre. Cuando aux1 es igual a dos, entonces a aux2 se le suma un uno y de esta forma pasa el turno al nodo posterior.

Por lo tanto, el primer bucle recorre los distintos niveles de profundidad y el segundo todos los nodos que hay en cada nivel. La primera variable d_t a la que se le fija un coeficiente con valor uno es la que está en la posición contd + t, donde $t = 2^1 = 2$, osea d_2.

Lo que vale esa variable debe ser menor igual a lo que vale su nodo padre, que es aquel que se sitúa en la columna contd + $2^{(i-1)}$ + aux2 de la matriz de restricciones, o lo que es lo mismo, contd + $2^{(1-1)} + 0$, posición que ocupa la variable d_1. d_1 es la $d_{p(2)}$, que se pasa al lado izquierdo de la restricción con el coeficiente cambiado de signo.

- *Restricción 6*

$$z_{it} \leq l_t, \ \forall t \in TL, \forall i = 1, \ldots, n \quad (6)$$

```
for (t in 1:TL){
  for (i in 1:n){
    contadorR<-contadorR+1
    A[contadorR,contz + TL*(i-1) + t]=1
    A[contadorR,contl+t]=-1
    b[contadorR]=0
  }
}
```

De los dos bucles que se programan, el primero recorre todas las $t \in TL$ y el segundo las $i = 1,\ldots,n$.

Para poder fijar el coeficiente de las variables z_{it} se ha de tener en cuenta la i. Si se hubiera calculado la posición de las columnas como contz + t, para todas las i solamente se crearían las restricciones para las TL primeras variables z_{it}. Por esa razón, se añade la suma de $TL \times (i - 1)$, de forma que se lleva un seguimiento de la i en la que se encuentra el bucle.

La variable l_t se desplaza al lado izquierdo de la restricción junto con un cambio de signo.

- *Restricción 7*

$$\sum_{i=1}^{n} z_{it} \geq Nminl_t, \qquad \forall t \in TL \quad (7)$$

```
for (t in 1:TL){
  contadorR<-contadorR+1
  for (i in 1:n){
    A[contadorR,contz + TL*(i-1) + t]=1
  }
  A[contadorR,contl+t]=-Nmin
  b[contadorR]=0
  signos[contadorR]='>='
}
```

Se añade una restricción por cada $t \in TL$ y para el $\sum_{i=1}^{n} z_{it}$ se construye el segundo bucle. Como ocurría con el $\sum_{j=1}^{p} a_{jt}$, el orden establecido para las variables z_{it} en la matriz de restricciones no es el mismo que el que requiere el sumatorio. Por eso, para cada t, las variables z_{it} que se añaden al sumatorio están separadas por TL columnas. Fuera del segundo bucle se define el coeficiente de las l_t , Nmin, con el signo cambiado ya que se lleva al lado izquierdo de la restricción. También se cambia el signo de la restricción por el pertinente.

- *Restricción 8*

$$\sum_{t \in TL} z_{it} = 1, i = 1, \ldots, n \quad (8)$$

```
for (i in 1:n){
  contadorR<-contadorR+1
  for (t in 1:TL){
    A[contadorR, contz + t + TL*(i-1)]=1
  }
  b[contadorR]=1
  signos[contadorR]='='
}
```

Al contrario de lo que sucedía en la restricción anterior, en esta hay tantas iteraciones como individuos i en el conjunto de datos. El $\sum_{t \in TL} z_{it}$ cuenta con TL elementos. Para $i = 1$, se suman las primeras TL variables z_{it}. Seguidamente, con $i = 2$, se multiplica $TL \times (2 - 1)$, por lo que se suman las TL variables siguientes.
El elemento de la derecha de la restricción se fija con un valor de uno.

- *Restricción 9, 10, 11 y 12*

Debido a que estas restricciones son un paso intermedio para llegar a las restricciones definitivas, que son las restricciones 13 y 14, no se han programado. Sin embargo, sí se

han utilizado para definir la restricción 13 y la restricción 14 como se explica a continuación.

- *Restricción 13*

$$a_m^T(x_i + \epsilon) \leq b_m + (1 + \epsilon_{max})(1 - z_{it}), i = 1,\dots,n, \forall t \in TL, \forall m \in A_L(t) \quad (13)$$

```
for(i in 1:n){
  for(t in (TB+1):Tnodos){
    for(mm in 1:(D)){
      if(AL[t,mm]!=0){
        contadorR<-contadorR+1
        m=AL[t,mm]
        for(j in 1:p){
          A[contadorR,(j-1)*TB+m]=datos[i,j]+epsilon[j]
        }
        A[contadorR,contb + m] = -1
        A[contadorR, contz  + (TL)*(i-1) + (t-TL)+1] = 1 + emax
        b[contadorR] = 1 + emax
      }
    }
  }
}
```

El valor de ϵ_j se ha tratado como una constante establecida por nosotros y ϵ_{max} es el mayor de esos valores. El primer bucle recorre todas las $i = 1,\dots,n$ y el segundo las $\forall t \in TL$. Para las m solo se deben incluir aquellos nodos ancestros de t que hayan seguido la rama izquierda de la partición que se ha realizado sobre ellos para llegar hasta t. $A_L(t)$ y $A_R(t)$ son matrices cuyo número de filas es igual al número total de nodos y cuyo número de columnas es igual a la profundidad D del árbol. Por ejemplo, si para $t = 28$ sus nodos ancestros pertenecientes a $A_L(t)$ son el 7 y el 14, que están en el nivel de profundidad 3 y 4 respectivamente, entonces la fila 28 de la matriz $A_L(t)$ tendrá en sus columnas los valores 0, 0 , 7, 14, etc. Lo mismo sucede en la matriz $A_R(t)$.

Por tanto, se añade un tercer bucle que recorra todos los niveles de profundidad del árbol. Si para un nodo t y una profundidad $d = 1,\dots,D$ determinados, el elemento situado en la fila t y columna d de la matriz $A_L(t)$ es distinto a cero, se genera una restricción y a m se le asigna ese valor distinto a cero.

El último bucle es con el que se calcula la $\sum_{j=1}^{p} a_{jm} \times (x_{ij} + \epsilon_j)$, que en la restricción aparece como un producto matricial.

- *Restricción 14*

$$a_m^T x_i \geq b_m - (1 - z_{it}), i = 1,\dots,n, \forall t \in TL, \forall m \in A_R(t) \quad (14)$$

```
for(i in 1:n){
  for(t in (TB+1):Tnodos){
    for(mm in 1:(D)){
      if(AR[t,mm]!=0){
        contadorR<-contadorR+1
        m=AR[t,mm]
        for(j in 1:p){
          A[contadorR,(j-1)*TB+m]=datos[i,j]
        }
        A[contadorR,contb + m] = -1
        A[contadorR, contz  + (TL)*(i-1) + (t-TL)+1] = -1
        b[contadorR] = -1
        signos[contadorR] = '>='
      }
    }
  }
}
```

En esta restricción, el planteamiento es exactamente es mismo que en la restricción anterior, salvo que ahora la matriz que se utiliza es $A_R(t)$. Por otro lado, también cambian los coeficientes de la suma de las a_{jm} y la dirección de la restricción.

- *Restricción 15*

$$N_{kt} = \frac{1}{2}\sum_{i=1}^{n}(1 + Y_{ik})z_{it}, k = 1,\dots,K, \forall t \in TL \quad (15)$$

```
for (k in 0:(K-1)){
  for (t in 1:TL){
    contadorR<-contadorR +1
    A[contadorR,contNkt + k*TL + t]=1
    for (i in 1:n){
      A[contadorR,contz + TL*(i-1) + t]=-(1/2)*(1 + Y[i,k+1])
    }
    b[contadorR]=0
    signos[contadorR]='='
  }
}
```

El primer bucle recorre las k clases del conjunto de datos, mientras que el segundo hace lo mismo con las $t \in TL$. En cuanto a las variables N_{kt}, primero se recorren las TL primeras que son aquellas con $k = 0$. Para saber en qué k se encuentra el bucle se debe sumar al número de columna $k \times TL$. De esta forma, si nos encontramos por ejemplo en $k = 1$, se sumarán las TL variables N_{kt} inmediatamente posteriores a N_{0TL}.

El bucle siguiente calcula el $\frac{1}{2}\sum_{i=1}^{n}(1 + Y_{ik})z_{it}$. Para seleccionar las variables z_{it} que se suman en cada iteración, se siga la misma lógica que en la restricción 7. Por cada t, se van sumando las variables z_{it} que están a TL columnas de distancia en la matriz de restricciones. Se cambia de signo el valor del coeficiente de las z_{it} debido a que se pasa el sumatorio al lado izquierdo de la restricción.

- *Restricción 16*

$$N_t = \sum_{i=1}^{n} z_{it}, \forall t \in TL \quad (16)$$

```
for (t in 1:TL){
  contadorR<-contadorR+1
  A[contadorR,contNt + t ]=1
  for (i in 1:n){
    A[contadorR,contz + TL*(i-1) + t]=-1
  }
  b[contadorR]=0
  signos[contadorR]='='
}
```

La explicación de cómo se ha programado esta restricción es muy similar a la anterior, excepto porque esta vez el número de restricciones es igual al número de $t \in TL$. Asignar los valores de los coeficientes de las variables N_t es sencillo, porque solo se debe fijar uno para cada $t \in TL$. Siguiendo la misma lógica que en la restricción 15 respecto a las variables z_{it} que se suman en cada iteración, se fija el coeficiente que le corresponde al sumatorio, teniendo en cuenta que este se desplaza al lado izquierdo de la restricción.

- *Restricción 17*

La restricción 17 no afecta a la resolución del problema, por lo que no es necesario incluirla en la matriz de restricciones. Los valores de c_t se pueden calcular una vez se haya obtenido la solución.

- *Restricción 18*

$$\sum_{k=1}^{K} c_{kt} = l_t, \forall t \in TL \quad (18)$$

```
for (t in 1:TL){
  contadorR<-contadorR+1
  for (k in 0:(K-1)){
    A[contadorR,contc + k*TL + t]=1
  }
  A[contadorR, contl + t]= -1
```

```
  b[contadorR]=0
  signos[contadorR]='='
}
```

El primer bucle es el que marca el número de restricciones, por eso recorre todas las $t \in TL$. Debido a que el sumatorio es de las $k = 1, \ldots, K$, las variables c_{kt} que se suman para cada t estarán separadas por TL columnas, por eso se añade $k \times TL$, para ir separando los elementos de la suma por TL variables.

- *Restricción 19*

La restricción 19 no es lineal, por lo que se han programado directamente las restricciones que permiten su linealización.

- *Restricción 20*

$$L_t \geq N_t - N_{kt} - M(1 - c_{kt}), k = 1, \ldots, K, \forall t \in TL \quad (20)$$

```
for (k in 0:(K-1)){
  for (t in 1:TL){
    contadorR<-contadorR+1
    A[contadorR,contLt + t]=1
    A[contadorR,contNt + t]=-1
    A[contadorR,contNkt + k*TL + t ]=1
    A[contadorR,contc + k*TL + t]=-M
    b[contadorR]=-M
    signos[contadorR]='>='
  }
}
```

El parámetro M se ha fijado asignándole un valor grande. El primer bucle se encarga de recorrer las clases $k = 1, \ldots, K$ y el segundo los nodos $t \in TL$. Por cada clase y nodo se añade una restricción al contador de restricciones. Con respecto al coeficiente de las variables N_{kt} y c_{kt}, vuelve a ser necesario tener en cuenta la k en la que se encuentra el

bucle a la hora de seleccionar las columnas a las que añadirlo. Se cambia, además, el signo de las variables que ahora han pasado a estar en el lado izquierdo de la restricción, así como el signo correspondiente.

- *Restricción 21*

$$L_t \leq N_t - N_{kt} + Mc_{kt}, k = 1, \ldots, K, \forall t \in TL \quad (21)$$

```
for (k in 0:(K-1)){
  for (t in 1:TL){
    contadorR<-contadorR+1
    A[contadorR,contLt + t]=1
    A[contadorR,contNt + t]=-1
    A[contadorR,contNkt + k*TL + t ]=1
    A[contadorR,contc + k*TL + t]=-M
    b[contadorR]=0

  }

}
```

La restricción 21 se programa de la misma manera que la restricción 20, cambiando únicamente los coeficientes distintos y la dirección de la restricción.

- *Restricción 22*

$$L_t \geq 0, \forall t \in TL \quad (22)$$

```
for (t in 1:TL){
  contadorR<-contadorR+1
  A[contadorR,contLt + t]=1
  signos[contadorR]='>='
}
```

El coeficiente que corresponde a las variables L_t se iguala a uno y se cambia el signo para que este sea '$\geq$'.

5.2.6. Función objetivo

La función objetivo del problema es:

$$\min \frac{1}{\hat{L}} \sum_{t \in TL} L_t + \alpha \sum_{t \in TB} d_t \quad (23)$$

Para resolver el problema, la función lp requiere que se le faciliten los coeficientes de la función objetivo.

En primer lugar, se define un vector de ceros con una longitud igual al número total de variables.

```
f=c(rep(0,p*TB + 2*TB + n*TL + TL + K*TL + TL + K*TL + TL))
```

Como se puede comprobar, la cantidad de variables del vector es la misma que de columna en la matriz de restricciones A.

$\hat{L}$ se calcula como el total de individuos en el conjunto de datos que pertenecen a la clase más común en el mismo conjunto.

```
Lgorro= max(table(datos$y))
```

Si consideramos que 'datos' es el conjunto de datos que se utiliza para crear el árbol y la columna 'y' es aquella en la que se guarda la clase de cada individuo, table() devuelve el número de observaciones por clase. De entre esos valores, se selecciona el máximo y ese es el valor que se asocia a $\hat{L}$.

A la hora de fijar los coeficientes de la función objetivo, se debe tener en consideración que las únicas variables a las que se les asigna un coeficiente son a las L_t y d_t.

```
for (t in 1:TL){
  f[contLt+t]=1/Lgorro
}
```

```
for (t in 1:TB){
  f[contd+t]=alpha
}
```

Cada bucle hace recorre los elementos del sumatorio de las L_t y d_t respectivamente. Los contadores de posición de las variables también nos sirven para la función objetivo porque, como se ha mencionado, el número de componentes de este vector es igual al número de columnas de A.

Para terminar, se otorga a las TL variables L_t y a las TB variables d_t es coeficiente que les corresponde, donde 'alpha' es un parámetro al que se le dota un valor entre cero y uno.

5.2.7 Solución

El paso final para poder obtener la solución del problema de los árboles óptimo es rellenar los parámetros de la función lp de la librería lpSolve con las matrices y vectores que se han calculado.

```
solucion=lp(direction = 'min', objective.in = f,const.mat = A,
const.dir = signos, const.rhs = b, binary.vec=bin, int.vec = enteras)
```

Para poder obtener una matriz con la solución final se aplica el siguiente código:

```
solucion$solution
```

Mientras que, para poder acceder al valor de la función objetivo, este es el código que se utiliza:

```
solucion$objval
```

5.2.8 Ejemplo de un árbol óptimo

Con la intención de comprobar el funcionamiento de comprobar el funcionamiento del problema que se ha programado, se va a presentar un ejemplo sencillo que permita interpretar el resultado de todas las variables.

El ejemplo consta de tres individuos, $i = 1,2,3$, con dos variables x_j para cada i, $x = \{x_{11}, x_{12}, x_{21}, x_{22}, x_{31}, x_{32}\}$ y una variable objetivo $y = \{y_1, y_2, y_3\}$.

Los valores de las x_{ij} se han generado de forma aleatoria como valores entre cero y uno y los de las y_i también se han generado de forma aleatoria, pero de manera que solo puedan tomar valor de cero o de uno.

```
datos=matrix(0,3,2)

set.seed(1)

for (i in 1:3){
  datos[i,]=runif(2,0,1)
}

v.obj<- sample(0:1,3,replace = TRUE)
v.obj

## [1] 0 0 1

datos<-data.frame(x1 = datos[,1],
                  x2 = datos[,2],

                  y = v.obj)

datos

##          x1        x2 y
## 1 0.2655087 0.3721239 0
## 2 0.5728534 0.9082078 0
## 3 0.2016819 0.8983897 1
```

El árbol tiene una profundidad total máxima de tres niveles, aunque a D le tenemos que dar como valor dos, porque cuenta el nivel del nodo raíz como nivel cero. El árbol tiene esta forma:

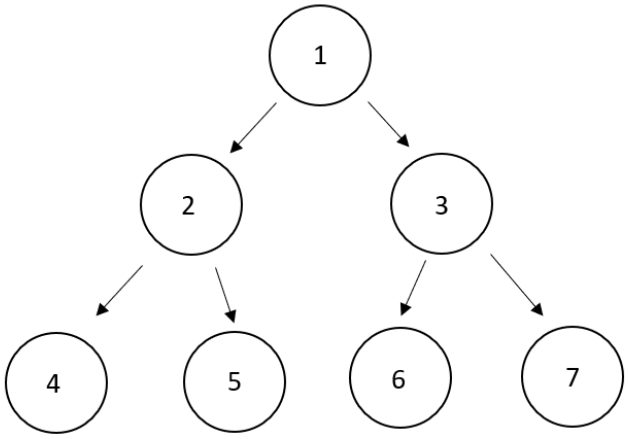

Figura 3. Árbol de ejemplo de profundidad máxima D=2 (Elaboración propia)

Cuando definimos los parámetros en R:

```
D <- 2
Tnodos <- 2^(D+1)-1;Tnodos

## [1] 7
```

El número total de nodos es correcto como se puede comprobar en la *Figura 3*.

```
TB <- floor(Tnodos/2);TB

## [1] 3

TL<-length(c((floor(Tnodos/2)+1):Tnodos));TL

## [1] 4
```

De esos siete nodos, tres son nodos rama (1,2 y 3) y los otros son nodos hoja (4,5,6 y 7).

```
p <- length(datos)-1;p

## [1] 2

n <-nrow(datos);n

## [1] 3

Nmin <- 1

K<- length(unique(datos$y));K
```

```
## [1] 2
```

p es el número de variables explicativas, que es igual a dos y n es el número de filas del conjunto de datos, en este caso, tres. Le damos a $Nmin$ el valor de uno, ya que al haber tan pocas observaciones, se ha intentado que se repartan en más de un nodo hoja. El número de K clases es igual a dos.

Las matrices $A_L(t)$ y $A_R(t)$ dan los siguientes resultados:

```
AL
```

```
##      [,1] [,2]
## [1,]    0    0
## [2,]    1    0
## [3,]    0    0
## [4,]    1    2
## [5,]    1    0
## [6,]    0    3
## [7,]    0    0
```

```
AR
```

```
##      [,1] [,2]
## [1,]    0    0
## [2,]    0    0
## [3,]    1    0
## [4,]    0    0
## [5,]    0    2
## [6,]    1    0
## [7,]    1    3
```

Si nos fijamos, por ejemplo, en el nodo 5, las matrices nos indican que su nodo ancestro por la izquierda es el 1 y por la derecha el 2, lo cual se puede confirmar mirando la Figura 3.

La matriz Y_{ik} resulta:

```
Y
```

```
##      [,1] [,2]
## [1,]    1   -1
## [2,]    1   -1
## [3,]   -1    1
```

Lo cual es correcto, ya que las dos primeras observaciones son de clase cero, que es la primera clase y la última de clase uno. El próximo paso es ejecutar las restricciones que se han programado en el apartado anterior.

En este problema, el número total de restricciones es 90 y el número de variables, 52. Se ha usado $\alpha = 0{,}5$, a M se le ha asignado el valor 1000 y $\epsilon_j = (0'0001, 0'0001)$.

Finalmente, los valores de las variables son:

```
#ajt
solucion$solution[1:contb]

## [1] 0 0 0 0 0 0

#bt
solucion$solution[(contb+1):contd]

## [1] 0 0 0

#dt
solucion$solution[(contd+1):contz]

## [1] 0 0 0
```

Todos los valores de estas tres variables son igual a cero. Tendría sentido que ninguna a_{jt} valiera uno y, por tanto, no se hiciera ninguna partición si, por alguna razón, ningún punto hubiera caído en un nodo hoja. Sin embargo, esto no es así, ya que, por ejemplo z_{it} es igual a:

```
#zit
solucion$solution[(contz+1):contl]

##  [1] 1 0 0 0 1 0 0 0 0 1 0 0
```

Según este resultado, algún punto i sí ha acabado en un nodo hoja $t \in TL$. Supuestamente, no podría llegar ningún punto a un nodo $t \in TL$ a no ser que sobre sus nodos ancestros se hayan hecho particiones.

Por ejemplo, z_{14} no tendría que poder valer uno si no se han partido sus nodos ancestros usando las variables a_{11} o a_{21} y a_{12} o a_{22}, o lo que es lo mismo, que se hayan usado la variable x_1 o x_2 para partir los nodos 1 y 2, que son los ancestros del nodo 4.

Sin embargo, si volvemos a las restricciones 13 y 14 con este supuesto que acabamos de mencionar suponiendo que $i = 1$, $m = 1$ y $t = 4$:

$$(x_{11} + \epsilon_1) \times a_{11} + (x_{12} + \epsilon_2) \times a_{21} \leq b_1 + (1 + \epsilon_{max}) \times (1 - z_{14}) \quad (13)$$
$$(x_{11}) \times a_{11} + (x_{12}) \times a_{21} \geq b_1 - (1 - z_{14}) \quad (14)$$

Si sustituimos los valores que nos ha devuelto el programa de R:

$$(x_{11} + \epsilon_1) \times 0 + (x_{12} + \epsilon_2) \times 0 \leq 0 + (1 + \epsilon_{max}) \times (1 - 1) \quad (13)$$
$$(x_{11}) \times 0 + (x_{12}) \times 0 \geq 0 - (1 - 1) \quad (14)$$

O lo que es lo mismo:

$$0 \leq 0 \quad (13)$$
$$0 \geq 0 \quad (14)$$

Los valores que nos ha devuelto el programa de R cumplen ambas restricciones. Esto quiere decir, que las restricciones 13 y 14 no aseguran que para que un punto llegue a un nodo hoja, sus nodos ancestros se deban haber partido.

Antes de continuar con el ejemplo, se va a comentar la forma en la que se ha decidido solucionar este problema para poder darle al problema un resultado con sentido.

Para poder asegurarnos de que cuando una observación alcance a un nodo hoja, los nodos ancestros de ese nodo se hayan partido se van a definir dos restricciones que se conocen como restricciones válidas del problema, que nos ayudan a reforzar la formulación.

Las restricciones son:

$$\sum_{j=1}^{p} a_{jm} \geq z_{it}, i = 1, \dots, n, \forall t \in TL, \forall m \in A_L(t) \quad (24)$$

$$\sum_{j=1}^{p} a_{jm} \geq z_{it}, i = 1, \dots, n, \forall t \in TL, \forall m \in A_R(t) \quad (25)$$

Con ellas, nos aseguramos de que si $z_{it} = 1$, con alguna de las x_j se han partido sus nodos ancestros, que vienen dados por m.

A continuación, introducimos estas últimas restricciones en nuestra programación del árbol óptimo.

- *Restricción 24*

```
for(i in 1:n){
  for(t in (TB+1):Tnodos){

    for(mm in 1:(D)){

      if(AL[t,mm]!=0){

        contadorR<-contadorR+1
        m=AL[t,mm]
        for(j in 1:p){

          A[contadorR,(j-1)*TB+m]=1

        }


        A[contadorR, contz  + (TL)*(i-1) + (t-TL)+1] = -1
        b[contadorR] = 0
        signos[contadorR]='>='
      }
    }
  }
}
```

El planteamiento de las restricciones es muy similar al de las restricciones 13 y 14, ya que tanto el sumatorio como el número de restricciones son iguales. La única cosa que cambia son los coeficientes que acompañan a las variables.

- *Restricción 25*

```
for(i in 1:n){
  for(t in (TB+1):Tnodos){
    for(mm in 1:(D)){
      if(AR[t,mm]!=0){
        contadorR<-contadorR+1
        m=AR[t,mm]
        for(j in 1:p){

          A[contadorR,(j-1)*TB+m]=1
        }

        A[contadorR, contz  + (TL)*(i-1) + (t-TL)+1] = -1
        b[contadorR] = 0
        signos[contadorR] = '>='

      }
    }
  }
}
```


Ahora, por tanto, a la matriz de restricciones A, al vector de la parte derecha de las restricciones b y al vector de signos, les tendremos que añadir el número de restricciones generadas por estas dos nuevas restricciones.

```
A=matrix(0,nrow=TB+TB+TB+ TB-1 + TL*n + TL + n + n*TL*D + K*TL + TL +
TL + K*TL + K*TL + TL+ n*TL*D,ncol=p*TB + 2*TB + n*TL + TL + K*TL + TL
+ K*TL + TL)
signos=c(rep('<=', TB+TB+TB+ TB-1 + TL*n + TL + n + n*TL*D + K*TL + TL
+ TL + K*TL + K*TL + TL+ n*TL*D))
b=c(rep(0,TB+TB+TB+ TB-1 + TL*n + TL + n + n*TL*D + K*TL + TL + TL +
K*TL + K*TL + TL+ n*TL*D))
```

Si volvemos a ejecutar el programa después de haber añadido las dos nuevas restricciones, estos son los resultados de las variables:

```
#ajt
solucion$solution[1:contb]

## [1] 1 0 1 0 0 0

#bt
solucion$solution[(contb+1):contd]

## [1] 0.0000000 0.0000000 0.2017819

#dt
solucion$solution[(contd+1):contz]

## [1] 1 0 1
```

Según estos resultados $a_{11} = a_{13} = 1$ y $d_1 = d_3 = 1$, por lo que los nodos 1 y 3 son los nodos rama que se parten y lo hacen usando la variable x_1. Fijándonos en los resultados de las b_t, el árbol resultante sería:

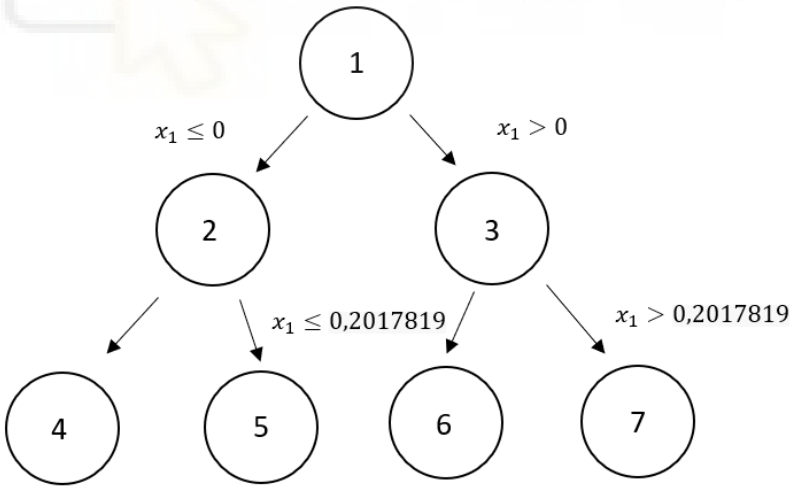

Figura 4. Árbol de ejemplo resuelto (Elaboración propia)

En el nodo 2 no se ha incluido la partición porque no se parte. Las variables z_{it} son:

```
#zit
solucion$solution[(contz+1):contl]
```

```
##  [1] 0 0 0 1 0 0 0 1 0 0 1 0
```

Lo cual significa que el punto 1 está en el nodo 7, al igual que el punto 2 y el punto 3 está en el nodo 6. Tiene sentido, porque el valor de x_1 de la primera observación es 0,2655087 y el de la segunda es 0,5728534, por lo tanto, son valores mayores a 0 y mayores a 0,2017819, que según el árbol deben acabar en el nodo 7. En cambio, la x_1 del tercer punto vale 0,2016819, por lo que al ser menor o igual a 0,2017819, se sitúa en el nodo 6.

Seguimos comprobando el resultado de las variables y las siguientes son las l_t.

```
#lt
solucion$solution[(contl+1):contNkt]
```

```
## [1] 0 0 1 1
```

l_t toma valor uno cuando hay un punto en el nodo $t \in TL$. En este caso, toman valor uno los elementos del vector referentes a los nodos 6 y 7, por lo que el resultado es correcto.

```
#Nkt
solucion$solution[(contNkt+1):contNt]
```

```
## [1] 0 0 0 2 0 0 1 0
```

En cuanto a las variables N_{kt}, el resultado nos indica que hay dos observaciones de clase 0 en el nodo 7 y una observación de clase 1 en el nodo 6. Es correcto, ya que los puntos 1 y 2 son de clase 0 y se han situado en el nodo 7 y el punto restante es de clase 1 y se le ha asignado el nodo 6.

```
#Nt
solucion$solution[(contNt+1):contc]
```

```
## [1] 0 0 1 2
```

N_t vuelve a confirmar que en el nodo 6 hay un punto y en el nodo 7, dos.

```
#ckt
solucion$solution[(contc+1):contLt]
```

```
## [1] 0 0 0 1 0 0 1 0
```

Recordemos que c_{kt} es una variable binaria que toma el valor 1 cuando la clase mayoritaria en el nodo t es k. En caso de que no haya punto en un nodo hoja, también vale 0. Si interpretamos el resultado, este nos indica que la clase predominante en el nodo 7 es la 0 y en el 6 es la clase 1. Otra vez es correcto, ya que los puntos 1 y 2, que han acabado en el nodo 7 son ambos de clase 0 y la observación 3, que es de clase 1, está en el nodo 6.

```
#Lt
solucion$solution[(contLt+1):(contLt+TL)]

## [1] 0 0 0 0
```

L_t en este caso, nos devuelve resultados nulos para todos los $t \in TL$. L_t se calcula como la diferencia entre el número de puntos que haya en el nodo y el número de puntos de la clase más común en ese nodo y en este ejemplo, el total de puntos en el nodo 7 es igual a dos y el número de puntos de la clase más común es también dos. Lo mismo ocurre en el nodo 6, solo hay un punto y, obviamente, ese punto es de la clase predominante del nodo.

Si se utilizara el programa con un conjunto de datos más grande, entonces se podría apreciar mejor el valor de las L_t y, por consecuente, el valor de la función objetivo.

```
solucion$objval

## [1] 1
```

El valor objetivo es 1, ya que, si sustituimos los resultados en la función que se ha definido anteriormente:

$$\min \frac{1}{\hat{L}} \sum_{t \in TL} L_t + \alpha \sum_{t \in TB} d_t$$

$$\frac{1}{2} \times (0 + 0 + 0 + 0) + 0{,}5 \times (1 + 0 + 1) = 1$$

6. Conclusiones

En conclusión, se ha analizado una formulación de árbol de clasificación óptima y se ha implementado correctamente en R. Además, se ha resuelto para un ejemplo concreto obteniendo resultados con sentido. Se han propuesto dos desigualdades válidas adicionales al planteamiento original de las restricciones de Bertsimas y Dunn (Bertsimas y Dunn, 2017). Estas dos nuevas restricciones válidas aseguran que un punto no puede caer en un nodo hoja $t \in TL$ si sus nodos ancestros no se habían partido primero, suceso que ha resultado clave a la hora de desarrollar e implementar el problema.

Este tipo de árbol va a permitir tomar todas las decisiones pertinentes en un solo paso, sin necesidad de hacer podas como ocurría con los árboles clásicos, y asegurando optimalidad global.

Para futuros trabajos, resultaría interesante realizar una comparación computacional entre los árboles clásicos y los árboles óptimos. En este trabajo, se han podido apreciar algunas ventajas de los árboles óptimos sobre los árboles clásicos, pero no se ha alcanzado a abordar las diferencias en cuanto a la precisión de ambos.

7. Bibliografía.

Parte del contenido teórico forma parte del material de la asignatura Técnicas Estadísticas en Análisis de Mercados, que se imparte en el Grado en Estadística Empresarial de la Universidad Miguel Hernández de Elche.

Carrizosa, E., & Romero Morales, D. (2013). Supervised classification and mathematical optimization. In *Computers and Operations Research* (Vol. 40, Issue 1, pp. 150–165). https://doi.org/10.1016/j.cor.2012.05.015

Bertsimas, D., & Dunn, J. (2017). Optimal classification trees. *Machine Learning*, *106*(7), 1039–1082. https://doi.org/10.1007/s10994-017-5633-9

Carrizosa, E., Molero-Río, C., & Romero Morales, D. (2021). Mathematical optimization in classification and regression trees. *TOP*, *29*(1), 5–33. https://doi.org/10.1007/s11750-021-00594-1

James, G. (Gareth M. (n.d.). An introduction to statistical learning : with applications in R.

K.P.Bennett. (1992). Decision Tree Construction Via Linear Programming. Proceedings of the 4th Midwest Artificial Intelligence and Cognitive Science Society Conference, October, 97–101. https://minds.wisconsin.edu/bitstream/handle/1793/59564/TR1067.pdf?sequence=1

Bennett, K. P., & Blue, J. (1996). Optimal decision trees. Rensselaer Polytechnic Institute Math Report No.214.

Michel Berkelaar and others (2020). lpSolve: Interface to 'Lp_solve' v. 5.5 to Solve Linear/Integer Programs. R package version 5.6.15. https://CRAN.R-project.org/package=lpSolve

Manos Papadakis, Michail Tsagris, Marios Dimitriadis, Stefanos Fafalios, Ioannis Tsamardinos, Matteo Fasiolo, Giorgos Borboudakis, John Burkardt, Changliang Zou, Kleanthi Lakiotaki and Christina Chatzipantsiou. (2020). Rfast: A Collection of Efficient andExtremely Fast R Functions. R package versión 2.0.1. https://CRAN.Rproject.org/package=Rfast

Gurobi Optimization Inc. (2015b). Gurobi optimizer reference manual. http://www.gurobi.com.

IBMILOG CPLEX.(2014). V12.1 usersmanual. https://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/.

Molero, M.C. (2017). Aprendizaje Supervisado mediante Random Forests.