

FACULTAD DE CIENCIAS
GRADO EN MATEMÁTICAS
TRABAJO FIN DE GRADO
CURSO ACADÉMICO [2021-2022]

TÍTULO:

**MÉTODOS DE MACHINE LEARNING BASADOS EN ÁRBOLES DE
DECISIÓN**

AUTOR:

GINÉS MECA CARBONELL

Resumen

En este trabajo, se ha estudiado el conjunto de algoritmos de Machine Learning que derivan de los árboles de decisión, confeccionando un mapa conceptual de modelos de manera que sea posible clasificarlos en cuanto a aspectos teóricos y reconocer sus principales características. Los árboles de decisión son algoritmos capaces de aprender de variables explicativas de datos conocidos y realizar predicciones sobre datos nuevos. Mediante diferentes tipos de combinaciones, se llegan a dos grandes familias de modelos más complejos: algoritmos de bagging y algoritmos de boosting. Estos modelos de mayor complejidad son capaces de conseguir un significativo aumento de precisión, especialmente cuando se enfrentan a grandes cantidades de datos. Por ello, son muy utilizados e imprescindibles actualmente para trabajar en Big Data.

Para poder profundizar y realizar un correcto análisis de los modelos más complejos, en la primera parte se ha estudiado el proceso de construcción de árboles de decisión. En particular, se ha analizado el algoritmo CART, que es el más utilizado. A lo largo de esta sección, se introducen diferentes nociones como los criterios de escisión, los criterios de parada o la asignación de valores de respuesta.

En la segunda parte del trabajo, se han estudiado los algoritmos de bagging y boosting por separado. Respecto a los algoritmos de bagging, se han explicado las ideas principales de su funcionamiento. En cuanto a los algoritmos de boosting, se ha realizado un detallado análisis de cada uno de ellos, explicando cuáles son las funciones y parámetros implicados así como el proceso exacto que aplican para conseguir disminuir el error de predicción.

Por último, tras la explicación de cada tipo de algoritmo, se ha realizado un ejemplo práctico a fin de ilustrar aquello explicado de manera teórica. También, se han utilizado estos ejemplos para realizar una comparación en términos de precisión y tiempo computacional de todos los modelos explicados, señalando diferencias entre ellos y comprobando que se cumplen ciertas características previamente explicadas. El código utilizado ha sido elaborado en exclusiva para el trabajo y se encuentra en el repositorio de GitHub del autor.

Palabras clave: CART, Bagging, Boosting, Machine Learning

Abstract

In this research, it has been studied the set of Machine Learning algorithms derived from decision trees, making a conceptual map of models so that we can classify it according to their theoretical aspects and we can recognize their main features. Decision trees are algorithms capable of learning from known data explanatory variables and making predictions about new data. Through different types of combinations, we obtain two families of more complex models: bagging algorithms and boosting algorithms. These models are able to increase significantly the accuracy, specially when they have to deal with large amounts of data. That's why they are very useful models and they have become essential in Big Data technologies.

In order to study in more detail and analyze correctly the complex models introduced, in the first part of this work it has studied the decision trees building process. In particular, it has been analyzed the most popular algorithm: CART. Along this section, different notions like split criteria, stop criteria or output assignment are introduced.

In the second part of this work, it has studied bagging and boosting algorithms by separate. In regard to bagging algorithms, it has been explained the main ideas related to their functioning. Referring to boosting algorithms, it has been done a complete analysis of each of them, explaining which are the functions and parameters implied as well as the exact process applied to reduce the prediction error.

Finally, after each type of algorithms explanation, solve a practical example has been solved in order to illustrate what the theory explained. Furthermore, these examples have been used to make a accuracy and running time comparison of all models, showing the differences between them and checking the features explained previously. The code used has been developed exclusively for the research and it's available in the author's GitHub repository.

Key words: CART, Bagging, Boosting, Machine Learning

Índice

1. Introducción	6
1.1. Datos: Lluvea en Australia	7
1.2. Evaluación de modelos	8
2. Árboles de decisión CART	10
2.1. Preliminares	10
2.1.1. Notación y conceptos básicos	11
2.1.2. Objetos de estudio de un CART	12
2.2. Valor o clase asignada a cada nodo hoja	12
2.2.1. Árboles de regresión	13
2.2.2. Árboles de clasificación	14
2.3. Elección de variables y valores asociados a cada nodo interno	14
2.3.1. Árboles de regresión	14
2.3.2. Árboles de clasificación	14
2.4. Criterio de parada de escisión de nodos	15
2.5. Ejemplo	16
2.6. Ventajas y desventajas	17
3. Métodos derivados de los árboles de decisión	19
3.1. Algoritmos de bagging	19
3.1.1. Bagging	19
3.1.2. Random Forest	19
3.1.3. Extra-Trees	20
3.1.4. Error de Out-Of-Bag	20
3.1.5. Ejemplo	21
3.1.6. Ventajas y desventajas	22
3.2. Algoritmos de boosting	23
3.2.1. AdaBoost	23
3.2.2. Gradient Boosting	30

3.2.3. XGBoost	36
3.2.4. LightGBM	42
3.2.5. CatBoost	43
3.2.6. Ejemplo	47
3.2.7. Otros parámetros	49
3.2.8. Ventajas y desventajas	50
 4. Comparación de modelos	 51
 5. Conclusión	 53
 A. Detalles del desarrollo del trabajo	 56
 B. Desarrollos Gradient Boosting para clasificación	 57
 Referencias	 59

1. Introducción

En plena era de la información, cada segundo millones de datos viajan entre diferentes lugares del planeta y se guardan formando enormes conjuntos de datos. Además, cada vez hay más instrumentos capaces de recoger información. Donde antes se necesitaba hacer uso de encuestas, ahora existen dispositivos, como los teléfonos móviles, capaces de escribir texto, recoger audio y realizar fotografías y vídeos. Así, la información al alcance es infinita. Es posible conocer cuáles son los conceptos que son tendencia en los buscadores de internet, o cuáles son aquellos audiovisuales que más éxito están teniendo en las redes sociales. Del mismo modo, se puede acceder al historial de imágenes de la cámara de seguridad de una vivienda o a la tabla que recoge las últimas operaciones de la empresa familiar. Es fácil obtener grandes cantidades de datos de aquello en lo que se está interesado.

Dado que el número de datos manipulados cada vez es mayor, las técnicas utilizadas para analizarlos van evolucionando constantemente. Muchas de las técnicas clásicas del análisis de datos han quedado obsoletas, otras se utilizan para crear algoritmos más complejos. Uno de los algoritmos más conocidos es el de los árboles de decisión. Fueron introducidos por primera vez en 1963 [12], quienes obtuvieron un método muy eficaz a través de un algoritmo muy básico. Actualmente, existen diferentes algoritmos que se pueden emplear a la hora de generar un árbol de decisión: CHAID, C4.5, FACT, QUEST, CRUISE... No obstante, el más conocido, y el que se estudiará en este trabajo, es el algoritmo CART (Classification And Regression Trees), publicado en 1984 [3] por Leo Breiman, entre otros. De hecho, fue el propio Breiman quien, haciendo uso de CART, desarrolló años más tarde algoritmos como Bagging [1] y Random Forest [2]. Paralelamente, la publicación de Freund y Schapire en 1997 [7] de un nuevo método basado en boosting daría lugar a otra serie de algoritmos, donde aparecen algoritmos más complejos, como XGBoost, que ocuparán gran parte del trabajo.

Este TFG se centra en analizar a fondo los algoritmos más complejos y novedosos de árboles de decisión. Para ello, se comenzará repasando, brevemente, el proceso de construcción de los árboles. Después, se dedicará la mayor parte del trabajo a desarrollar los algoritmos derivados de éstos, explicando brevemente los algoritmos de bagging y analizando en profundidad los de boosting, pues emplean técnicas más dificultosas. Para ilustrar los procesos descritos en el trabajo, se han desarrollado diferentes ejemplos. El código empleado para estos ejemplos y para los gráficos y tablas que aparecen a lo largo de la memoria ha sido desarrollado por el autor y se encuentra disponible en GitHub, se

puede acceder a él desde el Anexo (A). Por otro lado, se ha utilizado el mismo conjunto de datos a lo largo del proyecto, que se describen a continuación.

1.1. Datos: Llueve en Australia

El conjunto de datos “Rain in Australia” es un dataset formado por mediciones de 20 variables meteorológicas así como la fecha y localidad correspondientes y la información relativa a si llovió o no el día de la medición y el día siguiente, que hacen un total de 23 variables. Se trata de un conjunto de datos público disponible en Kaggle, el enlace al mismo se encuentra en el Anexo (A). El objetivo será evaluar la capacidad de predicción de lluvia de días futuros de los diferentes algoritmos descritos.

Se dispone de las mediciones de 145.460 días. En la Tabla 1 se muestran los valores de las 23 variables para los 5 primeros casos.

	0	1	2	3	4
Date	2008-12-01	2008-12-02	2008-12-03	2008-12-04	2008-12-05
Location	Albury	Albury	Albury	Albury	Albury
MinTemp	13.4	7.4	12.9	9.2	17.5
MaxTemp	22.9	25.1	25.7	28	32.3
Rainfall	0.6	0	0	0	1
Evaporation	NaN	NaN	NaN	NaN	NaN
Sunshine	NaN	NaN	NaN	NaN	NaN
WindGustDir	W	WNW	WSW	NE	W
WindGustSpeed	44	44	46	24	41
WindDir9am	W	NNW	W	SE	ENE
WindDir3pm	WNW	WSW	WSW	E	NW
WindSpeed9am	20	4	19	11	7
WindSpeed3pm	24	22	26	9	20
Humidity9am	71	44	38	45	82
Humidity3pm	22	25	30	16	33
Pressure9am	1007.7	1010.6	1007.6	1017.6	1010.8
Pressure3pm	1007.1	1007.8	1008.7	1012.8	1006
Cloud9am	8	NaN	NaN	NaN	7
Cloud3pm	NaN	NaN	2	NaN	8
Temp9am	16.9	17.2	21	18.1	17.8
Temp3pm	21.8	24.3	23.2	26.5	29.7
RainToday	No	No	No	No	No
RainTomorrow	No	No	No	No	No

Tabla 1: Encabezado del dataset donde se muestran las variables en filas y los 5 primeros casos en columnas.

En los diferentes ejemplos, se tratará de realizar predicciones de la variable “Rain Tomorrow”.

1.2. Evaluación de modelos

Hoy en día es habitual evaluar un modelo predictivo mediante el método de validación cruzada, proceso que evalúa el error o bien la calidad de un modelo con casos que no han participado en su obtención. A partir de un conjunto de datos, se siguen los siguientes pasos:

1. Se separa, al azar, el conjunto de datos inicial en dos: datos de entrenamiento y datos de testeo.
2. Se estima el modelo con los datos de entrenamiento.
3. Se aplica el modelo entrenado a los datos de testeo.
4. Se evalúa la precisión de las predicciones realizadas.

Habitualmente, la división del conjunto inicial que da lugar a los subconjuntos de entrenamiento y testeo se realiza de manera aleatoria. No obstante, dado que los datos podrían interpretarse como una serie temporal, donde se recogen mediciones diarias a lo largo de 10 años, a la hora de realizar predicciones interesa conocer la precisión del modelo ante datos futuros. Así, realizar una división aleatoria del conjunto inicial conllevaría que los datos de entrenamiento se encontrarían intercalados con los de testeo por lo que no se podría determinar la fiabilidad real del modelo, pues estas predicciones no nos aportarían ningún tipo de información acerca de la precisión del algoritmo a la hora de clasificar datos de fechas posteriores.

Por tanto, para el conjunto de datos introducido, el procedimiento será ligeramente distinto. Se empleará una validación “*Out Of Time*”, basada en separar el conjunto de datos siguiendo un orden cronológico. Dado que el dataset contiene fechas desde el 01/11/2007 hasta el 24/06/2017, el conjunto de entrenamiento estará formado por todos aquellos datos con fecha anterior al 01/01/2015 y los datos de testeo serán los restantes. Así, se entrenará el modelo con datos anteriores a los de testeo para saber cómo van a ser las predicciones futuras y lograr una mayor semejanza respecto a lo que interesaría en un caso real: entrenar el modelo en función de los datos que se poseen para predecir fechas futuras.

Otra posible manera de proceder sería realizando la predicción de cada día en función de todos los días anteriores. Sin embargo, este procedimiento resultaría muy costoso con los recursos disponibles, ya que se deberían construir tantos árboles de decisión como individuos se desee predecir y esto aumentaría exponencialmente el tiempo de ejecución.

Además, proceder de esta forma implicaría que, en un caso real, habría que entrenar el modelo diariamente, puesto que no habría forma de conocer el error cometido en las predicciones hechas con más de un día de diferencia. Así, utilizando la validación propuesta, se pueden calcular predicciones a dos años vista siendo conscientes del error esperado.

Por ultimo, con el fin de analizar los resultados obtenidos por cada modelo, se utilizará el AUC; es decir, el área bajo la curva ROC (“Area Under the Curve”), creada en base a la sensibilidad (tasa de verdaderos positivos) y especificidad (tasa de verdaderos negativos) de las predicciones. El AUC representa la probabilidad de ordenar correctamente dos individuos o datos dada la predicción. Dado que la métrica AUC se encuentra en el intervalo $[0, 1]$, se realiza la transformación $2 \times AUC - 1$ para obtener una nueva métrica en el intervalo $[-1, 1]$. Esta nueva métrica recibe el nombre de coeficiente Gini y permite dar un porcentaje de efectividad del modelo.

2. Árboles de decisión CART

2.1. Preliminares

Los árboles de decisión son muy utilizados como método predictor (tanto de regresión como de clasificación) dada su simpleza, su efectividad y lo visual que resulta su funcionamiento a través de un gráfico, lo cual facilita su comprensión. También, se pueden utilizar como herramienta descriptiva para un entendimiento inicial de los datos de un proyecto. Se trata de un algoritmo que divide sucesivamente de manera lineal el conjunto de datos inicial en diferentes subconjuntos a través de diversas condiciones aplicadas a sus variables explicativas. Véase con un ejemplo la idea básica de los árboles de decisión.

Ejemplo 2.1: *Se pretende estudiar la variable “RainTomorrow” del conjunto de datos dado en función de las variables explicativas “Cloud3pm” y “Humidity3pm”(nivel de nubosidad y de humedad a las 15.00, respectivamente) para predecir si lloverá o no al día siguiente. Así, escogiendo únicamente las variables explicativas indicadas, el esquema que ha aprendido el árbol de clasificación es el que se corresponde a la Figura 1.*

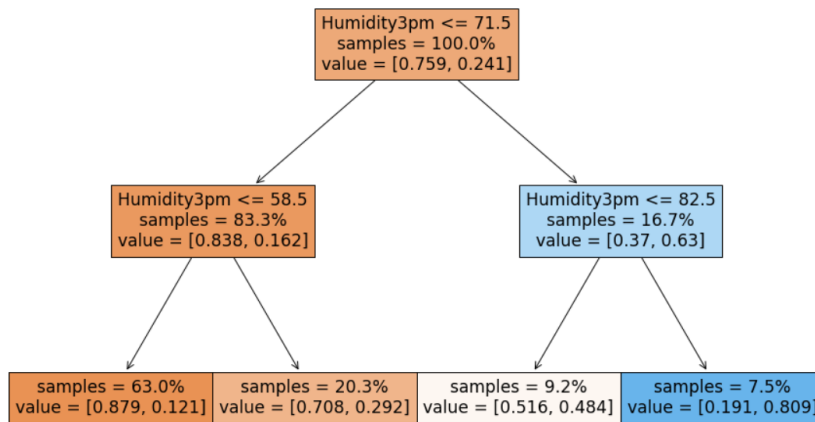


Figura 1: Árbol de decisión del Ejemplo 2.1

Como se puede observar, el algoritmo es muy intuitivo: a partir de la información de un individuo, se considera la primera condición; si la respuesta es afirmativa se sigue el camino de la izquierda y si es negativa el de la derecha. Mediante este proceso, se va comprobando si sus variables explicativas cumplen las diferentes condiciones que se plantean hasta alcanzar un grupo al que no se le aplican condiciones, el cual determinará su predicción.

Por otro lado, es fácil ver que las condiciones del algoritmo se corresponden con particiones del espacio. Como en el ejemplo anterior se han elegido únicamente dos variables explicativas, se puede hacer una representación de los individuos sobre \mathbb{R}^2 y determinar sobre él las diferentes regiones de predicción. En la Figura 2 se puede observar la partición del espacio del ejemplo mencionado. Cabe destacar que esta partición se basa en líneas verticales ya que, como se puede apreciar en la Figura 1, la variable “Cloud3pm” no ha sido seleccionada en el árbol para ninguna de las escisiones.

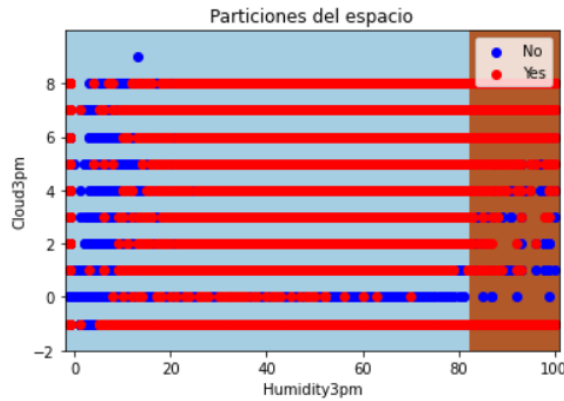


Figura 2: Partición del espacio del Ejemplo 2.1

2.1.1. Notación y conceptos básicos

A continuación, se introducen una serie de términos básicos para poder hacer referencias correctas a los conceptos presentados. Por un lado, nos encontramos con las siguientes definiciones:

Nodo raíz: Primer nodo, contiene a todos los individuos y a partir de él comienzan a realizarse las divisiones.

Nodo interno: Nodos intermedios que provienen de una división y desencadenan otra. Dentro de ellos, se pueden definir:

Nodo padre: Nodo anterior a un nodo interno fijado.

Nodos hijos: Nodos resultantes de la división del nodo interno fijado.

Nodo hoja: Nodos finales que dan lugar a la predicción.

Todos estos elementos son fácilmente identificables en la figura 1 del ejemplo 2.1 . Se tiene un nodo raíz, dos nodos internos y cuatro nodos hoja.

Por otro lado, conviene hablar del concepto de sobreajuste. Se dice que un modelo se encuentra sobreajustado si el algoritmo se ha adaptado en exceso a los individuos del conjunto de entrenamiento. Este hecho provoca que la predicción de los individuos de este conjunto sea muy buena, pero que las predicciones de nuevos individuos presenten grandes errores.

2.1.2. Objetos de estudio de un CART

Una vez explicada, de manera intuitiva, la manera de proceder de los árboles de decisión, corresponde explicar el verdadero funcionamiento de los mismos. Para ello, habrá que tratar los siguientes aspectos:

1. Elección de variables y valores asociados a cada nodo interno.
2. Criterio de parada de división de los nodos
3. Valor o clase asignada a cada nodo

2.2. Valor o clase asignada a cada nodo hoja

Sea un conjunto de datos de n individuos y $p+1$ variables ($n, p \in \mathbb{N}$). Sean X_1, X_2, \dots, X_p las variables explicativas e Y la variable dependiente. En el caso del conjunto de datos de ejemplo:

	X_1	X_2	\dots	X_{22}	Y
1	2008-12-01	Albury	\dots	No	No
2	2008-12-02	Albury	\dots	No	No
3	2008-12-02	Albury	\dots	No	No
\vdots	\vdots	\vdots	\ddots	\vdots	\vdots

Tabla 2: Datos “*Rain in Australia*”

Se denotarán con minúscula los datos o valores concretos de cada individuo, siendo (x_i, y_i) la fila i -ésima de la Tabla 2.

Como se comentaba anteriormente, el resultado de un árbol de decisión es una partición del espacio. Si se obtienen M regiones diferentes, denotadas por R_m con $m = 1, \dots, M$.

Se define como \hat{Y} al árbol de decisión y como c_m , con $m = 1, \dots, M$, a las predicciones (imágenes de \hat{Y}) asociadas a cada región R_m . De este modo:

$$\hat{Y}(x) = \sum_{m=1}^M c_m I(x \in R_m) \quad \forall x \in D$$

donde D es el dominio de la función \hat{Y} (espacio de las variables explicativas) e I es la función indicadora:

$$I(x) = \begin{cases} 1 & \text{si } x \in R_m \\ 0 & \text{si } x \notin R_m \end{cases}$$

La manera de establecer el valor de estas predicciones presenta ligeras diferencias en función del tipo de problema con el que se trate (regresión o clasificación). Por tanto, se analizarán por separado.

2.2.1. Árboles de regresión

En un modelo de regresión, se busca minimizar los errores entre Y e \hat{Y} . Tomando el error mínimo cuadrático, se llega al problema:

$$\min_{\hat{Y}_i} \sum_{i=1}^n (y_i - \hat{Y}_i)^2 \Rightarrow \min_{c_m} \sum_{m=1}^M \sum_{x_i \in R_m} (y_i - c_m)^2$$

donde $\hat{Y}_i = \hat{Y}(x_i) \quad \forall i = 1, \dots, n$. Denotando $f(c_m) = \sum_{x_i \in R_m} (y_i - c_m)^2$, se observa que alcanzar la solución óptima del problema planteado es equivalente a minimizar la función f . Derivando, se obtiene un candidato a óptimo:

$$f'(c_m) = -2 \sum_{x_i \in R_m} (y_i - c_m) = -2 \sum_{x_i \in R_m} y_i + 2n_m c_m = 0 \Leftrightarrow c_m = \frac{1}{n_m} \sum_{x_i \in R_m} y_i \quad (1)$$

y se comprueba que, efectivamente, es el óptimo a través de la segunda derivada:

$$f''(c_m) = -2 \sum_{x_i \in R_m} (-1) = 2n_m > 0$$

donde n_m es el conjunto de datos pertenecientes a la región R_m , $\forall m = 1, \dots, M$.

En conclusión, el valor asignado a cada nodo hoja será la media de las observaciones de los individuos del conjunto de entrenamiento que pertenecen a dicho nodo.

2.2.2. Árboles de clasificación

En el caso de árboles de clasificación, la tarea se simplifica: basta con calcular la proporción de individuos de cada clase en cada región y, así, el valor asignado a cada subconjunto del espacio será la clase a la que le corresponde la mayor proporción en el mismo.

2.3. Elección de variables y valores asociados a cada nodo interno

2.3.1. Árboles de regresión

Se analizará la primera división y el proceso será similar para las siguientes. Al situarse en el nodo raíz, se pretende dividir el conjunto de datos inicial en dos regiones, denotadas por R_1 y R_2 , en función de una de las variables explicativas, X_j , y de un valor respecto de ésta, s , de manera que cada región contendrá a los individuos:

$$R_1(j, s) = \{x_i | x_{ij} \leq s\} \quad \wedge \quad R_2(j, s) = \{x_i | x_{ij} > s\}$$

Así, se generan dos valores de predicción constantes, uno por cada nodo hoja; es decir, se definen c_1 y c_2 tales que c_1 se mantendrá constante sobre R_1 y c_2 constante sobre R_2 , calculados según (1). De esta manera, se buscan j y s que minimicen:

$$\min_{j,s} \left(\sum_{x_i \in R_1} (y_i - c_1)^2 + \sum_{x_i \in R_2} (y_i - c_2)^2 \right)$$

o lo que es lo mismo, se buscan j y s que minimicen la suma de residuos cuadráticos (RSS, del inglés: “*Residuals Sum of Squares*”) en el conjunto de entrenamiento.

Así, a la hora de dividir el nodo raíz, se comprueban todas las posibles divisiones posibles y se escogen j y s óptimos. A continuación, se aplica el mismo proceso recursivamente para los nodos hijos generados.

2.3.2. Árboles de clasificación

En el caso de los árboles de clasificación, la variable dependiente es una clase. Es decir, la situación es equivalente a la recogida en la Tabla 2. En este caso, se utiliza una función de impureza que permita evaluar la homogeneidad de los nodos; o lo que es lo mismo, que permita estudiar qué acciones separan mejor el espacio en función de las diferentes clases.

El criterio más empleado es el índice Gini, que se define de la forma:

$$G_{region_m} = G_m = \sum_{k=1}^K p_{mk}(1 - p_{mk})$$

donde p_{mk} es la proporción de individuos del conjunto de entrenamiento de clase k que se encuentran en la región m y K es el número total de clases. Además, se puede observar que valores de p_{mk} cercanos a 0 o 1 implican que el producto $p_{mk}(1 - p_{mk})$ sea muy pequeño. Por tanto, el índice Gini actúa como medidor de la pureza de una región: un índice Gini de bajo valor significa que los diferentes p_{mk} están cercanos a 0 o 1 y la región es “pura”.

Para evaluar el índice Gini asociado a la división de un nodo se utiliza una media ponderada de los índices de las dos regiones generadas tras la escisión, es decir:

$$G_{nodo} = \frac{n_1}{n_1 + n_2} G_1 + \frac{n_2}{n_1 + n_2} G_2$$

donde n_1 y n_2 son el número de individuos de las regiones R_1 y R_2 , respectivamente, y G_1 y G_2 son los índices Gini de las mismas.

Por tanto, al igual que en los árboles de regresión, la tarea se resume en obtener, mediante comprobación de todas las opciones posibles, la variable y el valor que minimizan el índice Gini del nodo y aplicar el proceso recursivamente.

2.4. Criterio de parada de escisión de nodos

Es lógico pensar que la precisión de un árbol CART aumenta conforme aumenta en profundidad (número de divisiones). No obstante cuantas más particiones del espacio realice el algoritmo, mayor será el ajuste del mismo a los datos de entrenamiento, lo que puede llevar a problemas de *overfitting*. Con tal de evitar estos problemas, se han implementado algunos métodos de “poda” del árbol (eliminación de algunas de las divisiones realizadas). Es evidente que podando el árbol se introduce sesgo, pues no se permite que se ajuste totalmente a los datos; sin embargo, aplicando estos procesos de manera controlada se consigue disminuir la varianza y mejorar las predicciones.

Por un lado, la técnica más simple consiste en establecer una profundidad máxima del árbol, es decir, poner límite a la cantidad de nodos hijo a desarrollar. Con esta condición, se puede asegurar que los nodos dejarán de dividirse una vez alcanzado el nivel indicado. Por ejemplo, en el Ejemplo 2.1 se ha establecido una profundidad máxima igual a 2.

Por otro lado, el árbol determinará un nodo como nodo hoja cuando la separación del mismo no sea conveniente en términos de pureza. El procedimiento es el siguiente:

en primer lugar, se calcula el índice Gini de la región asociada a ese nodo y, después, se realiza la mejor división del nodo y se calcula el índice Gini del mismo (proveniente de la media ponderada de índices de sus regiones). Si el índice de la región sin dividir es inferior al índice del nodo, se deshará la división y se determinará el nodo como nodo hoja, pues su división no mejora la pureza.

2.5. Ejemplo

Se desea obtener una predicción fiable acerca de las precipitaciones del día siguiente en una ciudad de Australia. Para ello, haciendo uso del dataset anteriormente introducido, se dividirá el conjunto de datos en dos subconjuntos: uno de entrenamiento y otro de testeo. Además, se probará con diferentes profundidades máximas permitidas para escoger el modelo más fiable. En particular, se construirán árboles de profundidades 1, 3, 5, 10, 15, 20 y 30.

	Run_Time	Train_Gini%	Test_Gini%	delta%
DT_1	0.3	36.0	32.0	-9.0
DT_3	0.5	58.0	56.0	-2.0
DT_5	0.7	66.0	63.0	-4.0
DT_10	1.2	77.0	61.0	-21.0
DT_15	1.7	91.0	43.0	-52.0
DT_20	2.1	98.0	31.0	-68.0
DT_30	2.1	100.0	36.0	-64.0

Tabla 3: Coeficientes de Gini en 7 árboles de clasificación para los datos “Rain in Australia”

En la Tabla 3 se ven reflejados el tiempo de ejecución de cada árbol, los coeficientes de Gini para los conjuntos de entrenamiento y testeo y el porcentaje de información que se pierde en el conjunto de testeo respecto del de entrenamiento. Como se puede observar, cuando la profundidad máxima permitida es superior o igual a 10, el modelo empieza a sobreajustarse a los datos de entrenamiento, aumentando la desviación entre los conjuntos conforme aumenta la profundidad máxima permitida. Así, los modelos más fiables son el DT_3 y el DT_5 (profundidades 3 y 5) pues en ambos la variación se encuentra por debajo del 5%. En la Figura 3 se puede comprobar que estos modelos son los óptimos ya que permitiendo una mayor profundidad se obtiene una precisión menor con una diferencia mayor entre conjuntos.

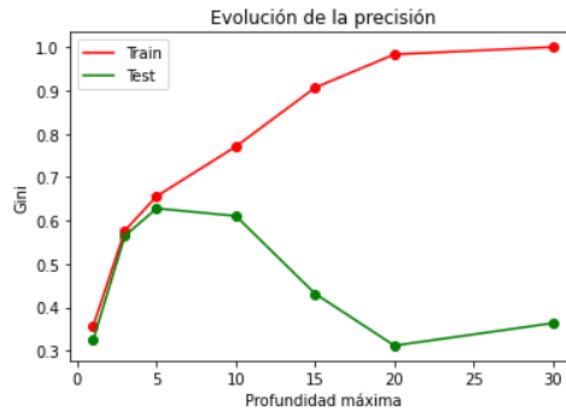


Figura 3: Evolución de los coeficientes Gini en ambos conjuntos

Intuitivamente, el valor $\text{delta}\%$ se puede entender como desviación que sufre el conjunto de testeo con el paso de 2 años (los datos de testeo van de 2015 a 2017). Por tanto, si se escogiese un modelo con un $\text{delta}\% = 20\%$, tendríamos que este 20% se iría incrementando en las predicciones cada dos años, algo que llevaría a un modelo totalmente impreciso. Por ello, se escogen aquellos con variaciones pequeñas.

Por último, nótese que el tiempo empleado en la generación de los árboles presentados es mínimo. Los modelos escogidos (DT_3 y DT_5) apenas tardan medio segundo en construirse, resultado que es prueba de la eficiencia de esta clase de modelos.

2.6. Ventajas y desventajas

Ventajas:

- Válidos para problemas de regresión y de clasificación.
- Válidos para una exploración inicial de los datos.
- Modelos de bajo coste computacional.
- Método muy intuitivo de fácil comprensión, sencillo de comunicar a compañeros no expertos.
- La manera de proceder del modelo se asemeja a cómo lo hace la mente humana: se van estableciendo condiciones que restringen, paso a paso, el conjunto total de datos hasta llegar a la solución.

- Se puede obtener una representación gráfica de los diferentes modelos, lo cual facilita su comprensión a nivel visual.

Desventajas:

- La capacidad predictora de los árboles de decisión es inferior a la que se puede obtener mediante otros métodos. De hecho, podemos observar que los coeficientes de Gini de los modelos del Ejemplo 2.5 en el conjunto de testeo no alcanza el 65 %.
- Se trata de un algoritmo inestable. Es decir, un ligero cambio en los datos de entrenamiento puede conllevar a la creación de un modelo totalmente distinto.
- Con un árbol de decisión es ineficiente modular correlaciones lineales altas entre la variable objetivo y una variable explicativa. Modular esta alta correlación podría ser más eficiente con un modelo clásico de Regresión Lineal o Logística.

No obstante, pese a no poseer una gran capacidad de predicción, existen diferentes maneras de combinar árboles de decisión de forma que se obtengan modelos de mayor precisión y que no se sobreajusten.

3. Métodos derivados de los árboles de decisión

Tanto CART como muchos de los algoritmos utilizados para construir árboles de decisión se desarrollaron a lo largo de la década de los 80. No obstante, ya se ha visto en el Ejemplo 2.5 que estos algoritmos no llegan a alcanzar grandes niveles de precisión; es por eso que a partir de la década de los 90 empiezan a surgir diferentes artículos que introdujeron nuevas técnicas que, mediante distintas combinaciones de árboles de decisión, consiguen modelos de mayor fiabilidad y que incorporan distintas características que los convierte en modelos muy potentes. A la hora de combinar los árboles de decisión para construir modelos más complejos se pueden seguir dos caminos: bagging o boosting. Dedicaremos esta sección al estudio de ambos.

3.1. Algoritmos de bagging

Los algoritmos de bagging (*“Bootstrap Aggregation”*) se caracterizan por tomar muestras aleatorias del conjunto de datos inicial y construir con ellas árboles de decisión diferentes. Se analizarán tres algoritmos de bagging distintos: Bagging, Random Forest y Extra-Trees.

3.1.1. Bagging

Tras participar en la creación del algoritmo CART, Leo Breiman publicó, en 1996, un artículo en el que introducía un nuevo método con el fin de disminuir la varianza de CART: Bagging [1]. Bagging es un algoritmo que toma diferentes muestras aleatorias de individuos con repetición del conjunto de datos y con cada una de ellas construye un árbol de decisión haciendo uso de todas las variables explicativas disponibles. De esta manera, se consiguen diferentes predicciones, todas ellas del mismo peso, a través de las cuales se obtiene una respuesta final mediante el cálculo de la media (regresión) o mediante conteo de clases, escogiendo la clase con más apariciones (clasificación).

3.1.2. Random Forest

El desarrollo de Bagging llevó a Breiman a definir un nuevo algoritmo 5 años más tarde. Con Random Forest [2], Breiman consiguió, además de disminuir la varianza, obtener información más completa de las variables explicativas. El procedimiento es muy parecido al de Bagging: nuevamente, se toman diferentes muestras con repetición de los individuos

y, para cada muestra, también se toma una muestra aleatoria de m variables (con $m \leq p$). Una buena elección de m podría ser: $m = \lfloor \sqrt{p} \rfloor$.

Por tanto, la diferencia con Bagging es que ahora cada árbol se construirá teniendo en cuenta una muestra aleatoria de las variables explicativas. De esta manera, dado que en ocasiones tenemos variables muy correlacionadas con la variable a predecir, se consigue que no siempre sean estas variables las que participen en el modelo y, así, participe el máximo número de variables posibles en la respuesta final.

3.1.3. Extra-Trees

El algoritmo Extra-Trees, o *Extremely Randomized Trees*, aporta aún más margen a la aleatoriedad dentro del modelo. Utiliza como base lo descrito en Random Forest, pero presenta una particularidad en cuanto a la escisión de nodos. Como se explica en la subsección 2.3, en la escisión de nodos se busca la variable, y el valor óptimo asociado, que minimizan una función de error o impureza. En Extra-Trees, en lugar de probar todos los valores posibles de todas las variables disponibles, se generará un valor para cada una de las variables que será el que se evalúe. Así, se probará con m valores aleatorios correspondientes a las m variables explicativas escogidas al azar. En particular, este algoritmo puede resultar muy útil ante problemas de *overfitting* extremos.

3.1.4. Error de Out-Of-Bag

Cabe dedicar un apartado para hablar de la noción del error *Out-Of-Bag* (OOB). Como ya se ha visto, los algoritmos de bagging toman muestras con repetición de los individuos para la construcción de cada uno de los árboles. Aquellos individuos que no participan en la creación de un árbol reciben el nombre de individuos *out-of-bag*. Si se cuenta con n individuos, la probabilidad de que un individuo no se escoja para la creación de un árbol (individuo *out-of-bag*) es:

$$\left(\frac{n-1}{n}\right)^n$$

Por tanto, se puede observar que cuando el número de individuos de la muestra sea elevado (supóngase $n \rightarrow +\infty$):

$$\lim_{n \rightarrow +\infty} \left(\frac{n-1}{n}\right)^n = \lim_{n \rightarrow +\infty} \left(1 - \frac{1}{n}\right)^n = \lim_{n \rightarrow +\infty} \left[\left(1 + \frac{1}{-n}\right)^{-n}\right]^{-1} = e^{-1} \simeq 0,3679$$

Es decir, si se trabaja con un conjunto de datos de gran tamaño, cerca de un 37% de los individuos no se tendrán en cuenta para la creación de cada árbol (en particular).

Así, en bagging, el error del modelo se puede evaluar en el conjunto OOB, que actúa como conjunto test. Para cada árbol del modelo, los individuos que se utilizarán para el testeo serán aquellos individuos *out-of-bag* del árbol en cuestión. Además, una propiedad muy buena del error OOB es que converge a medida que se añaden árboles al modelo. De esta manera, podemos asegurar que los modelos de bagging no se sobreajustan.

No obstante, en los ejemplos se seguirá evaluando la precisión del modelo empleando la validación *Out Of Time*. El motivo es el ya comentado en la subsección 1.2: interesa realizar una separación de “datos pasados” y “datos futuros” y los individuos *out-of-bag* son aleatorios dentro del conjunto de entrenamiento.

3.1.5. Ejemplo

Nuevamente, se desea predecir si lloverá o no el próximo día en una ciudad de Australia haciendo uso, ahora, de los algoritmos de bagging. Además, con tal de evitar sobreajustes, se establecerá el criterio de parada de los árboles generados en 5 (como se ve en el Ejemplo 2.5, es el más fiable junto a 3) y se probará con diferentes números de árboles generados por cada algoritmo.

a)	Run_Time	Train_Gini%	Test_Gini%	delta%
Bag_1	0.6	65.0	63.0	-4.0
Bag_3	1.4	69.0	66.0	-3.0
Bag_5	2.4	69.0	66.0	-4.0
Bag_10	4.6	69.0	67.0	-4.0
Bag_15	6.4	69.0	66.0	-4.0
Bag_20	8.4	69.0	67.0	-4.0
Bag_30	12.6	69.0	67.0	-4.0
Bag_50	21.0	69.0	67.0	-4.0
Bag_100	41.7	70.0	67.0	-4.0
Bag_200	83.0	70.0	67.0	-4.0
Bag_500	208.8	70.0	67.0	-3.0
Bag_1000	425.1	70.0	67.0	-3.0

b)	Run_Time	Train_Gini%	Test_Gini%	delta%
RF_1	0.2	49.0	46.0	-6.0
RF_3	0.3	66.0	62.0	-5.0
RF_5	0.5	65.0	62.0	-4.0
RF_10	0.8	68.0	65.0	-4.0
RF_15	1.0	68.0	66.0	-4.0
RF_20	1.3	69.0	67.0	-4.0
RF_30	2.1	69.0	67.0	-3.0
RF_50	3.3	70.0	67.0	-4.0
RF_100	6.4	70.0	67.0	-3.0
RF_200	11.7	70.0	67.0	-4.0
RF_500	30.0	70.0	67.0	-4.0
RF_1000	56.3	70.0	67.0	-4.0

c)	Run_Time	Train_Gini%	Test_Gini%	delta%
ET_1	0.2	30.0	26.0	-13.0
ET_3	0.3	56.0	55.0	-3.0
ET_5	0.4	60.0	58.0	-3.0
ET_10	0.6	66.0	64.0	-3.0
ET_15	0.8	66.0	63.0	-4.0
ET_20	0.9	64.0	62.0	-4.0
ET_30	1.3	65.0	62.0	-4.0
ET_50	2.0	65.0	63.0	-4.0
ET_100	3.8	65.0	63.0	-4.0
ET_200	7.5	66.0	63.0	-4.0
ET_500	18.2	65.0	63.0	-4.0
ET_1000	35.0	66.0	63.0	-4.0

Tabla 4: Resultados algoritmos de bagging para RainTomorrow: a)Bagging b)Random Forest c)Extra-Trees

Aplicando los diferentes algoritmos se llega a los resultados expuestos en la Tabla 4. Como se puede apreciar, la mejora respecto a CART es evidente: en el ejemplo anterior se llegaba a la conclusión de que una de las mejores opciones era elegir el árbol con profundidad máxima 5 que otorgaba una precisión en Gini del 66% en el conjunto de entrenamiento y del 63% en el de testeo. Ahora, tanto en Bagging como en Random Forest, escogiendo cualquiera de los modelos que realiza al menos 10 árboles, se obtienen

coeficientes de Gini que rozan, en ocasiones, el 70% en ambos conjuntos. En el caso de *Extra-Trees*, la mejora no es tan significativa pero los coeficientes siguen siendo mejores. Además, en los tres modelos se aprecia un sustancial descenso, de aproximadamente 1 punto porcentual, en delta, lo cual refleja una clara disminución de la varianza del modelo.

También, se puede observar que el modelo no se sobreajusta al incrementar el número de estimadores que se generan; y que este incremento no produce grandes cambios en los coeficientes de Gini a partir de un cierto número de árboles. Así, el error converge como se puede comprobar visualmente en la Figura 4.

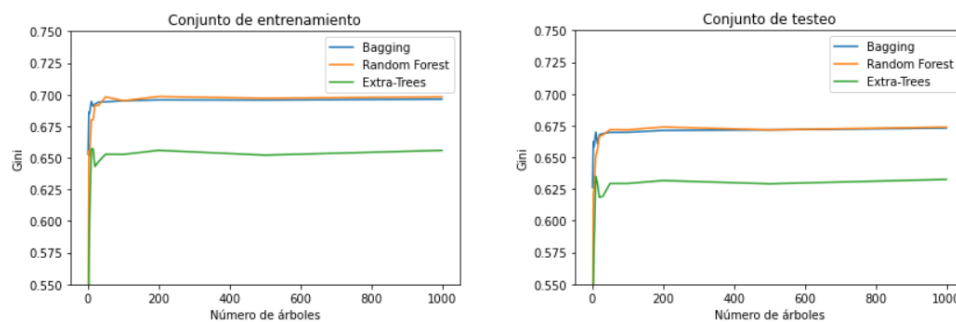


Figura 4: Convergencia coeficientes Gini

Por último, nótese que los algoritmos de bagging son más costosos que *CART* en cuanto al tiempo empleado para su construcción. Además, se puede observar una clara relación entre los diferentes modelos y el tiempo que requieren. A la hora de construir los árboles de decisión, Bagging tiene en cuenta todas las variables, Random Forest solo considera algunas de ellas y *Extra-Trees*, además de no usar todas las variables, evalúa las posibles escisiones de nodos en base a valores aleatorios. Estas características se ven claramente reflejadas en los tiempos de ejecución, pues los tiempos de ejecución más altos corresponden a Bagging y los más bajos se le atribuyen a *Extra-Trees*.

3.1.6. Ventajas y desventajas

Ventajas:

- Algoritmos capaces de disminuir la varianza; lo que implica que las predicciones realizadas serán (casi) igual de fiables que las del conjunto de entrenamiento.
- Aumento de precisión respecto a otros algoritmos más simples (como *CART*).
- Al tratarse de modelos “democráticos”, son buenas alternativas estables en problemas con datos ruidosos, con bajas correlaciones o con datos faltantes.

Desventajas:

- Se pierde la fácil visualización y comprensión de CART.
- Pese a la mejora en la precisión, el nivel de fiabilidad sigue siendo relativamente pobre (en el Ejemplo 3.1.5 no se alcanza el 70 % de precisión en Gini).

3.2. Algoritmos de boosting

Los algoritmos de boosting, al igual que los de bagging, generan diferentes árboles de decisión a través de los cuales se realiza la predicción final. No obstante, mientras los algoritmos de bagging generan árboles independientes y devuelven una respuesta promedio o la respuesta más votada, los algoritmos de boosting generan los árboles secuencialmente, teniendo en cuenta los resultados anteriores. De esta manera, esta clase de métodos consigue un significativo descenso del sesgo ya que la forma en la que se construyen los diferentes árboles permite otorgar pesos diferentes a los individuos, consiguiendo así que el nuevo árbol generado se focalice en predecir bien a aquellos individuos que presentan mayor error (o que se han clasificado erróneamente) en el árbol anterior.

A lo largo de esta subsección trataremos de abordar a fondo los principales algoritmos de boosting: AdaBoost, Gradient Boosting, XGBoost, LightGBM y CatBoost.

3.2.1. AdaBoost

Adaboost (“***A**daptative **B**oosting*”) [7] es un algoritmo que construye árboles de decisión con solo dos nodos hoja; es decir, con una sola escisión, por lo que está pensado para ser ejecutado con un alto número de iteraciones. A estos árboles de decisión se les llamará “tocones”. En la figura 5 se puede observar un ejemplo de estos árboles.

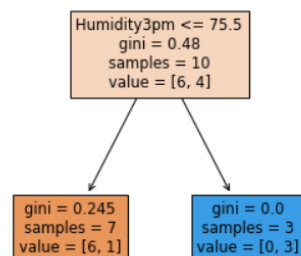


Figura 5: Ejemplo de tocón construido en AdaBoost.

En la práctica, AdaBoost construye un primer tocón a partir de todos los datos disponibles, cuyo resultado es considerado como la predicción inicial del modelo, y a continuación se realiza un cálculo de errores respecto a esta predicción. Así, se genera un nuevo conjunto de datos a través de una muestra con repetición del conjunto inicial, en la que tendrán mayor probabilidad de aparecer aquellos individuos con errores más altos. Después, la siguiente y sucesivas iteraciones consistirán en construir nuevos tocones que clasifiquen la muestra de datos correspondiente a la iteración, calculando los errores pertinentes y los sucesivos conjuntos de datos. De esta manera, AdaBoost construye una cadena de árboles donde se le resta importancia a la predicción de aquellos individuos para los que ya ha conseguido errores bajos mientras que aumenta el peso de los individuos mal predichos para que sean el principal objetivo en los árboles posteriores. Por último, la respuesta final del modelo será la predicción inicial más las predicciones de las sucesivas iteraciones. A continuación, se procederá a explicar de manera detallada el algoritmo “Discrete AdaBoost”. Éste fue el primer algoritmo presentado en el que se hace referencia a este método, y aborda problemas de clasificación binaria.

En el primer paso, se asigna el mismo peso a todos los individuos. Así, en primer lugar:

$$w_i = \frac{1}{n} \quad \forall i \in \{1, \dots, n\}$$

En cada una de las M iteraciones, se crea un tocón, denotado por G_m con $m \in \{1, \dots, M\}$. Además, dado que estos árboles solo poseen dos nodos hoja, en cada tocón únicamente se verá involucrada una variable.

El error de la iteración m -ésima se calcula de la forma:

$$err_m = \frac{\sum_{i=1}^n w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^n w_i}$$

es decir, la suma de los pesos de los individuos mal clasificados en el numerador y la suma total de los pesos en el denominador. Dado que Adaboost tiene diferentes implementaciones, en algunas de ellas el denominador puede ser distinto de 1. No obstante, siguiendo la implementación de Python, siempre se estandarizarán los pesos antes de empezar de nuevo con el bucle, por lo que se puede considerar el denominador igual a 1:

$$err_m = \frac{\sum_{i=1}^n w_i I(y_i \neq G_1(x_i))}{\sum_{i=1}^n w_i} = \frac{\frac{1}{n} \sum_{i=1}^n I(y_i \neq G_1(x_i))}{1} = \frac{num.errores}{n}$$

A continuación, se calcula el coeficiente:

$$\alpha_m = \log \left(\frac{1 - err_m}{err_m} \right)$$

Como se puede observar, si el tocón $G_m(x)$ ha realizado un buen trabajo de clasificación ($err_m \simeq 0$), a α_m le corresponderá un valor positivo. Por el contrario, un mal trabajo de clasificación ($err_m \simeq 1$) se verá traducido en un α_m negativo. Así, se puede entender α_m como un medidor de la importancia del tocón $G_m(x)$ en el modelo: un valor alto implica una buena clasificación general y, por tanto, su respuesta es importante para el resultado final y, por otro lado, un valor bajo implica una clasificación general pobre con un número considerable de errores y conviene que su influencia en el resultado final no sea muy elevada. Esta evolución del α_m en función del error se puede comprobar visualmente en el gráfico de la función expuesto en la Figura 6.

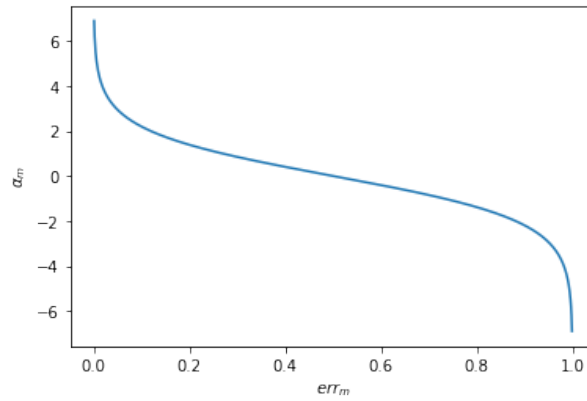


Figura 6: Gráfico de α_m

Al final de cada iteración, se recalculan los pesos a través de la expresión:

$$w_i^{m+1} = \begin{cases} w_i^m & \text{si } y_i = G_m(x_i) \\ w_i^m e^{\alpha_m} & \text{si } y_i \neq G_m(x_i) \end{cases}$$

donde w_i^m es el peso del individuo i en la iteración m -ésima. Nótese que en caso de que $G_m(x)$ haga un buen trabajo de clasificación ($\alpha_m \gg 0$) habrá pocos individuos mal clasificados y a e^{α_m} le corresponderá un valor alto; pues, dado que $G_m(x)$ tendrá una importancia relevante en el modelo, interesa priorizar la correcta clasificación de aquellos que han presentado errores. Conforme α_m se va acercando a 0, la importancia del tocón en el modelo disminuye, efecto que se ve reflejado en la modificación de los pesos pues e^{α_m} se acercará a 1 y no habrá apenas diferencia entre los pesos de la iteración actual y su consecutiva. Por otro lado, en caso de que $G_m(x)$ tenga una eficacia del 50 %, $\alpha_m = 0$,

el modelo no tendrá importancia y, dado que la probabilidad de acertar o no es la misma, no tiene sentido modificar los pesos. Por último, en el caso de obtener un modelo $G_m(x)$ que haga un pésimo trabajo de clasificación (eficacia menor que el 50 %), $\alpha_m \lll 0$, se podría interpretar que el modelo está trabajando a la inversa y que hay que considerar las clasificaciones contrarias que éste proporciona. Por tanto, en este caso, interesa disminuir el peso de los mal clasificados por el modelo, lo que es equivalente a aumentar el peso de los bien clasificados, ya que los que el modelo clasifique correctamente serán los que serán considerados como erróneos dada la eficacia del modelo.

Finalizando con las iteraciones, queda hacer especial hincapié en un par de conceptos. Se puede observar en la sección 2.3 que en la expresión del índice Gini dada para escisión de nodos no intervienen los pesos de los individuos. No obstante, en AdaBoost es necesario que los tocones se construyan teniendo en cuenta los pesos de los individuos. La solución más fácil, y la que se encuentra implementada en la librería *Scikit-learn* de Python, es la siguiente: se estandarizan los pesos, es decir, se divide el peso de cada individuo por la suma total de pesos, y se considera el resultado como las frecuencias relativas de cada individuo. Así, se pueden conseguir también las frecuencias absolutas de la muestra. Se pueden observar los pasos descritos en la Tabla 5.

Individuo	w		Individuo	w*		Individuo	w*
1	w_1		1	$\frac{w_1}{\sum_{i=1}^n w_i}$		1	w_1^*
2	w_2	\Rightarrow	2	$\frac{w_2}{\sum_{i=1}^n w_i}$	\Rightarrow	2	w_2^*
\vdots	\vdots		\vdots	\vdots		\vdots	\vdots
n	w_n		n	$\frac{w_n}{\sum_{i=1}^n w_i}$	\Rightarrow	n	w_n^*

	Individuo	w*	F		Individuo	F
	1	w_1^*	w_1^*		1	F_1
\Rightarrow	2	w_2^*	$\sum_{i=1}^2 w_i^*$	\Rightarrow	2	F_2
	\vdots	\vdots	\vdots		\vdots	\vdots
	n	w_n^*	$\sum_{i=1}^n w_i^* (= 1)$		n	$F_n (= 1)$

Tabla 5: Reasignación de pesos

El siguiente paso es generar n números aleatorios (t_1, t_2, \dots, t_n) pertenecientes al in-

tervalo $[0, 1]$. Entonces, se cumple que:

$$\exists j \in \{1, \dots, n\} / t_i \in [F_{j-1}, F_j] \quad \forall i \in \{1, \dots, n\}$$

donde $F_0 = 0$.

Entonces, la solución propuesta al problema de los pesos consiste en crear un nuevo conjunto de datos utilizando los t_i generados aleatoriamente: si $t_i \in [F_{j-1}, F_j]$, entonces se añade x_j al nuevo conjunto. De esta manera, se obtiene un nuevo conjunto de datos de n individuos en el que los individuos mal clasificados por el árbol anterior tendrán mayor presencia (como sus pesos son más altos, los intervalos $[F_{j-1}, F_j]$ serán más amplios). Por último, se asigna un peso de $1/n$ a cada uno de los individuos del nuevo conjunto de datos y ya se procede a la creación del siguiente tocón.

Ejemplo funcionamiento del bucle:

Supóngase que el conjunto de datos se redujese al de la Tabla 6.

	Humidity3pm	Cloud3pm	RainTomorrow	Weight
0	88.0	-1.0	1.0	0.1
1	27.0	-1.0	0.0	0.1
2	69.0	-1.0	0.0	0.1
3	82.0	8.0	1.0	0.1
4	54.0	-1.0	0.0	0.1
5	36.0	7.0	0.0	0.1
6	36.0	7.0	0.0	0.1
7	92.0	8.0	1.0	0.1
8	66.0	6.0	0.0	0.1
9	52.0	4.0	1.0	0.1

Tabla 6: Submuestra del conjunto de datos para ilustrar el funcionamiento del bucle.

Inicialmente, se crea un árbol de decisión de una única división, obteniendo el tocón y la consiguiente partición del espacio ilustrados en la figura 7.

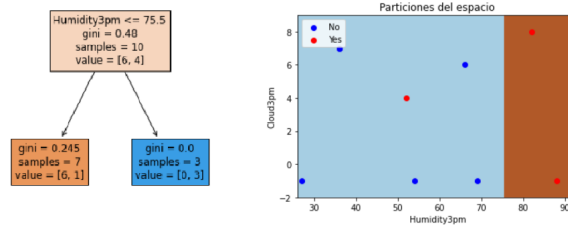


Figura 7: Primer tocón del modelo

Como se puede observar, solo hay un individuo mal clasificado, el número 9. A continuación, se calcula err_1 :

$$err_1 = \frac{\text{num.errores}}{n} = \frac{1}{10} = 0,1$$

y, haciendo uso de él, se obtiene α_1 :

$$\alpha_1 = \log \frac{1 - err_1}{err_1} = \log \frac{1 - 0,1}{0,1} = \log \frac{0,9}{0,1} = \log 9 \simeq 2,1972$$

Una vez hechos estos cálculos, se realiza la reasignación de pesos:

$$w_i^{m+1} = \begin{cases} w_i^m & \text{si } y_i = G_m(x_i) \\ w_i^m e^{\alpha_m} & \text{si } y_i \neq G_m(x_i) \end{cases} \Rightarrow w_i^2 = \begin{cases} w_i^1 & \text{si } i \neq 9 \\ w_i^1 e^{\alpha_1} & \text{si } i = 9 \end{cases} \Rightarrow w_i^2 = \begin{cases} w_i^1 & \text{si } i \neq 9 \\ 9w_i^1 & \text{si } i = 9 \end{cases}$$

donde los superíndices hacen referencia a la iteración. Así, modificados los pesos, se aplica el proceso recogido en la Tabla 5 y se obtienen los resultados de la Tabla 7.

Individuo	w^2		Individuo	w^{2*}		Individuo	w^{2*}
0	0,1		0	$\frac{0,1}{1,8}$		0	0,0556
1	0,1		1	$\frac{0,1}{1,8}$		1	0,0556
2	0,1		2	$\frac{0,1}{1,8}$		2	0,0556
3	0,1		3	$\frac{0,1}{1,8}$		3	0,0556
4	0,1	\Rightarrow	4	$\frac{0,1}{1,8}$	\Rightarrow	4	0,0556
5	0,1		5	$\frac{0,1}{1,8}$		5	0,0556
6	0,1		6	$\frac{0,1}{1,8}$		6	0,0556
7	0,1		7	$\frac{0,1}{1,8}$		7	0,0556
8	0,1		8	$\frac{0,1}{1,8}$		8	0,0556
9	0,9		9	$\frac{0,9}{1,8}$		9	0,5

Individuo	F
0	0,0556
1	0,1111
2	0,1667
3	0,2222
4	0,2778
5	0,3333
6	0,3889
7	0,4444
8	0,5
9	1

Tabla 7: Transformación del peso en frecuencias

Por tanto, solo queda generar 10 números aleatorios en $[0, 1]$ y asociarlos a los individuos en función del intervalo en el que se encuentren, tal y como se muestra en la Tabla 8.

t	0.3954	0.4193	0.0144	0.5211	0.0210	0.9923	0.0953	0.5072	0.7743	0.8238
Fila	7	7	0	9	0	9	1	9	9	9

Tabla 8: Selección de individuos para el nuevo conjunto de datos

Por tanto, el conjunto de datos que utilizaremos para entrenar el segundo tocón será el formado por los individuos que aparecen en la Tabla 8. Como se puede comprobar, el individuo 9, antes mal clasificado, ahora aparece 5 veces, por lo que el algoritmo tendrá que clasificarlo bien para no asumir un error tan elevado.

Una vez terminado el bucle, queda analizar cómo realiza las predicciones AdaBoost. La solución del algoritmo es:

$$G(x) = \text{sign} \left[\sum_{m=1}^M \alpha_m G_m(x) \right]$$

Entonces, dado que el algoritmo “Discrete AdaBoost” se diseñó para resolver problemas de clasificación binaria, a una clase se le asocia el valor -1 y a la otra el valor $+1$. Por tanto, la respuesta será la siguiente: si $\sum_{m=1}^M \alpha_m G_m(x) < 0$, $G(x) = -1$; y en caso de $\sum_{m=1}^M \alpha_m G_m(x) > 0$, $G(x) = 1$. De esta manera, como ya se avanzaba anteriormente, α_m es un medidor de la importancia del tocón dentro del modelo, pues un α_m alto implicará una gran contribución de la respuesta del tocón a la respuesta final.

Consideraciones finales AdaBoost:

Tras la publicación de “Discrete AdaBoost”, se expuso el algoritmo “Real Adaboost” [9], que extendía el funcionamiento del algoritmo visto al caso continuo y a más de dos grupos. A continuación, analicemos qué modificaciones se añaden para afrontar problemas de clasificación multiclase y problemas de regresión.

En el caso de tener más de dos grupos, en el último paso, la respuesta de la función será la clase cuyo sumatorio de α'_m s sea mayor. Es decir, para cada clase k se calcula $\sum_{m/G_m(x)=k} \alpha_m$ y la clase cuyo sumatorio sea mayor, es la que corresponde a la respuesta del algoritmo.

En el caso de un problema de regresión, se utiliza una “función de pérdida” o, en inglés, “*Loss Function*” (a partir de ahora, se hará referencia a ella en su término inglés dado que es como se conoce realmente). Esta función mide la discrepancia o distancia entre el verdadero valor y_i y el valor \hat{y}_i predicho por el modelo. Vendrá dada de la siguiente forma:

$$L_i^{(m)} = L [|\hat{y}^{(m)}(x_i) - y_i|] \quad \forall i \in \{1, \dots, n\}$$

donde $\hat{y}^{(m)}(x_i)$ hace referencia al valor predicho por el m -ésimo árbol al introducir los valores del individuo i -ésimo e y_i es el valor de la variable dependiente en el individuo i -ésimo. Existen diferentes funciones candidatas Loss Function. De entre ellas, destacan las que devuelven valores entre 0 y 1, en concreto:

$$\begin{cases} L_i^{(m)} = \frac{|\hat{y}^{(m)}(x_i) - y_i|}{n} & (lineal) \\ L_i^{(m)} = \frac{|\hat{y}^{(m)}(x_i) - y_i|^2}{n^2} & (cuadrada) \\ L_i^{(m)} = 1 - \exp \left[\frac{-|\hat{y}^{(m)}(x_i) - y_i|}{n} \right] & (exponencial) \end{cases}$$

Una vez elegida la función a emplear, se calcula el error ponderado del tocón realizando la media de las $L_i^{(m)}$'s:

$$L^{(m)} = \frac{1}{n} \sum_{i=1}^n L_i^{(m)}$$

De esta manera, se define el α_m en el caso de regresión de manera análoga a lo explicado anteriormente:

$$\alpha_m = \log \left(\frac{1 - L^{(m)}}{L^{(m)}} \right)$$

Así, una vez solucionado el problema de cómo medir el error, el resto del procedimiento es análogo al de clasificación, tomando como respuesta final: $G(x) = \sum_{m=1}^M \alpha_m G_m(x)$.

3.2.2. Gradient Boosting

Gradient Boosting [8] es un algoritmo que, mediante la construcción secuencial de árboles, consigue identificar cuáles son aquellos individuos cuyos errores asociados son mayores y trata de mejorar las predicciones de los mismos. Al igual que AdaBoost, el primer paso consiste en determinar una predicción inicial. No obstante, mientras AdaBoost prosigue a través de actualizaciones de los pesos y de la generación de nuevos conjuntos de datos, Gradient Boosting construye una secuencia de árboles en la que cada árbol aspira a corregir el error cometido por los árboles anteriores. Para ello, se requiere de una Loss Function a través de la cual obtiene los residuos y el valor asociado a cada nodo hoja de

los árboles generados. El objetivo es minimizar esta Loss Function para, así, minimizar el error cometido. La respuesta final viene dada por la predicción inicial más la suma del resto de árboles que tratan de cuantificar los sucesivos errores. De esta forma, se parte de una predicción inicial con un error alto que se reduce iteración a iteración.

El algoritmo es el mismo tanto para el caso de regresión como para el de clasificación. La diferencia entre estos dos casos reside únicamente en la elección de la Loss Function a emplear. A continuación, se analizarán ambos casos por separado.

Regresión

Para iniciar el algoritmo, se debe establecer una Loss Function $L(y_i, f(x_i))$, donde y_i es el elemento i -ésimo de y y $f(x_i)$ es la predicción de la fila i -ésima de X . Ya se ha visto que existen diferentes funciones que pueden ser candidatas a Loss Function, por ello el algoritmo viene definido para una $L(y_i, f(x_i))$ cualquiera. No obstante, en Gradient Boosting se suele hacer uso de:

$$L(y_i, f(x_i)) = \frac{1}{2}(y_i - f(x_i))^2$$

La elección de esta Loss Function se debe a que va a ser necesario derivarla. Así, derivando respecto de $f(x)$, la expresión resultante no es más que el residuo de la predicción:

$$\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)} = -(y_i - f(x_i)) \quad (2)$$

Una vez establecida la Loss Function a emplear, el primer paso consiste en determinar un valor fijo que representará la “primera predicción”; es decir, aquella que luego será modificada en función de los residuos:

$$f_0(x) = \arg \min_{\gamma} \sum_{i=1}^N L(y_i, \gamma)$$

que sustituyendo por la función fijada queda:

$$f_0(x) = \arg \min_{\gamma} \sum_{i=1}^N L(y_i, \gamma) = \arg \min_{\gamma} \frac{1}{2} \sum_{i=1}^N (y_i - \gamma)^2$$

Ya se vio en la sección 2.2 (ecuación (1)) que el valor que minimiza este valor es la media, por tanto:

$$f_0(x) = \frac{1}{N} \sum_{i=1}^N y_i$$

A continuación, en cada una de las M iteraciones de este bucle se genera un nuevo árbol que añadirá información a la última predicción realizada. Primero, se calcula el residuo de la última predicción de cada uno de los individuos (la $(m-1)$ -ésima) mediante la derivada especificada en 2):

$$r_{im} = - \left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)} \right]_{f=f_{m-1}} = y_i - f_{m-1}(x_i)$$

Después, se genera un árbol cuyo objetivo es realizar una predicción de los r_{im} en función de los x_i . Así, se denota por R_{jm} a los diferentes nodos hoja del árbol m -ésimo y cada uno de ellos llevará asignado el valor de respuesta:

$$\gamma_{jm} = \arg \min_{\gamma} \sum_{x_i \in R_{jm}} L(y_i, f_{m-1}(x_i) + \gamma)$$

De nuevo, el γ que minimizará dicha expresión será la media de los residuos de los individuos pertenecientes a R_{jm} .

Para terminar con el bucle, se define cuál es la m -ésima actualización de la predicción:

$$f_m(x) = f_{m-1}(x) + \nu \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$$

es decir, se le añade el residuo predicho a la última predicción calculada multiplicada por un valor $\nu \in [0, 1]$, donde ν es un parámetro, denominado “tasa de aprendizaje”, que ayuda a evitar el sobreajuste. Como el algoritmo establece una predicción inicial que actualiza iteración a iteración haciendo uso de los residuos, la omisión de la tasa de aprendizaje implicaría un *overfitting* en un número bajo de iteraciones. Cabe mencionar que Gradient Boosting se suele aplicar con M grande y ν pequeño pues, en la práctica, se comprueba que es más fiable utilizar modelos que realicen muchos pasos de pequeñas modificaciones en lugar de pocos pasos de grandes modificaciones, pues tienden a sobreajustarse menos.

Por último, la respuesta del algoritmo es la de $f_M(x)$.

Por último, cabe recordar que se ha desarrollado el algoritmo bajo la hipótesis de que la Loss Function escogida es la más habitual en este caso. Evidentemente, el procedimiento es análogo para cualquier función pero, en ese caso, valores como $f_0(x)$ o γ_{jm} dejarán de ser medias y se tendrán que calcular los valores que minimizan las nuevas expresiones. Además, puntualizar que entonces no sería del todo correcto definir los r_{im} como residuos, ya que la forma de estos podría ser muy diferente (no tendría por qué ser “valor observado - valor predicho”). Es por eso que en Gradient Boosting se denomina a estos valores como “pseudo-residuos”.

Clasificación:

El procedimiento descrito está pensado para clasificación binaria. Mientras que en regresión la estimación proporcionada se corresponde con la variable de interés, en el caso de clasificación estamos interesados en estimar la probabilidad de pertenecer a cierta clase de manera que si la probabilidad es mayor que 0.5 se clasificará en la clase “1” y si la probabilidad es menor que 0.5 se clasificará en la clase “0”. No obstante, Gradient Boosting no aproxima esta probabilidad directamente, si no que aproxima el log-odd-ratio y, posteriormente, obtiene la probabilidad deseada:

$$odd - ratio = \frac{p}{1 - p}$$

En primer lugar, se verá cómo determinar la Loss Function escogida. Nótese que en el caso de clasificación binaria el conjunto de datos se corresponderá con una muestra aleatoria simple de una población con distribución bernoulli. Es decir:

$$f(y, p) = p^y(1 - p)^{1-y}$$

y por tanto, la función de verosimilitud quedará:

$$l(p) = l(p|y_1, y_2, \dots, y_n) = \prod_{i=1}^n f_p(y_i) = \prod_{i=1}^n p^{y_i}(1 - p)^{1-y_i}$$

y la función de soporte:

$$\ln(l(p)) = \ln\left(\prod_{i=1}^n f_p(y_i)\right) = \ln\left(\prod_{i=1}^n p^{y_i}(1 - p)^{1-y_i}\right) = \sum_{i=1}^n y_i \ln(p) + (1 - y_i) \ln(1 - p)$$

Ahora, nótese que cuando se hace uso de esta función en el método de Máxima Verosimilitud, interesa maximizar su valor. Por el contrario, cuando trabajamos con una Loss Function, se trata de minimizarla, pues el modelo será más ajustado conforme menor sea el valor de esta función. Por tanto, si se desea emplear la función de soporte como Loss Function, se tendrá que multiplicar por -1 . También, como se desea aplicar esta función individuo a individuo, se puede eliminar el sumatorio. Así, la Loss Function queda:

$$L(y_i, p_i) = -(y_i \ln(p_i) + (1 - y_i) \ln(1 - p_i))$$

No obstante, ésta es una función de p_i y se desea trabajar con log-odd-ratio (a partir

de ahora, por simplificar, $\ln(\text{odd}_i)$). Entonces, se transforma esta expresión de la forma:

$$\begin{aligned} L(y_i, p_i) &= -(y_i \ln(p_i) + (1 - y_i) \ln(1 - p_i)) = -y_i \ln(p_i) - \ln(1 - p_i) + y_i \ln(1 - p_i) = \\ &= -y_i (\ln(p_i) - \ln(1 - p_i)) - \ln(1 - p_i) = -y_i \ln \left(\frac{p_i}{1 - p_i} \right) - \ln(1 - p_i) = \\ &= -y_i \ln(\text{odd}_i) - \ln(1 - p_i) \end{aligned} \quad (3)$$

Además, se puede comprobar que $p_i = \frac{e^{\ln(\text{odd}_i)}}{1 + e^{\ln(\text{odd}_i)}}$ y así:

$$\ln(1 - p_i) = \ln \left(1 - \frac{e^{\ln(\text{odd}_i)}}{1 + e^{\ln(\text{odd}_i)}} \right) = \ln \left(\frac{1}{1 + e^{\ln(\text{odd}_i)}} \right) = -\ln(1 + e^{\ln(\text{odd}_i)})$$

Por tanto, volviendo a (3):

$$L(y_i, p_i) = -y_i \ln(\text{odd}_i) - \ln(1 - p_i) = -y_i \ln(\text{odd}_i) + \ln(1 + e^{\ln(\text{odd}_i)})$$

y ya se tiene la Loss Function en función de $\ln(\text{odd}_i)$:

$$L(y_i, \ln(\text{odd}_i)) = -y_i \ln(\text{odd}_i) + \ln(1 + e^{\ln(\text{odd}_i)})$$

Así pues, una vez establecida la Loss Function, se deben seguir los mismos pasos que en el caso de regresión pero sustituyendo por la nueva función. Como el procedimiento es, en la gran mayoría, análogo, se suprimirá parte de los desarrollos, disponibles en el Anexo (B).

Ahora, la predicción inicial vendrá dada por:

$$p_0(x) = \arg \min_{\gamma} \sum_{i=1}^N L(y_i, \gamma) = \arg \min_{\gamma} \sum_{i=1}^N (-y_i \gamma + \ln(1 + e^{\gamma})) = \frac{\sum_{i=1}^N y_i}{N} \quad (4)$$

Se puede observar, dado que $y_i \in \{0, 1\} \forall i \in \{1, \dots, n\}$, que la predicción inicial es la proporción de individuos del conjunto de entrenamiento que pertenecen a la clase “1”. No obstante, dado que se quiere trabajar en términos de $\ln(\text{odd}_i)$, se debe escribir esta probabilidad en la forma correspondiente. De esta manera:

$$f_0(x) = \frac{p_0(x)}{1 - p_0(x)} = \frac{\frac{\sum_{i=1}^N y_i}{N}}{1 - \frac{\sum_{i=1}^N y_i}{N}} = \frac{\sum_{i=1}^N y_i}{N - \sum_{i=1}^N y_i}$$

De acuerdo con lo anterior, el siguiente paso consiste en el bucle que calculará en cada iteración (supóngase la iteración m -ésima): los “pseudo-residuos” de la predicción $f_{m-1}(x)$,

un árbol de decisión que clasifique estos residuos, los valores de respuesta de los nodos hoja del árbol generado y la nueva predicción $f_m(x)$.

Los “pseudo-residuos” vendrán dados por:

$$r_{im} = - \left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)} \right]_{f=f_{m-1}} = y_i - \frac{e^{f_{m-1}(x_i)}}{1 + e^{f_{m-1}(x_i)}} \quad (5)$$

Nótese que $f_{m-1}(x_i)$ es, realmente, la última predicción realizada de $\ln(\text{odd}_i)$. Por tanto, r_{im} podría escribirse como:

$$r_{im} = y_i - p_{i,m-1}$$

donde $p_{i,m-1}$ es la predicción $(m-1)$ -ésima de p_i . Es decir, r_{im} es y_i menos la última predicción realizada del mismo. Así, tiene sentido que este valor sea conocido como “pseudo-residuo”.

A continuación, se construye un árbol de decisión que realice una predicción de los r_{im} 's en función de las variables independientes y se denotan sus nodos hoja por R_{jm} con $j \in \{1, \dots, J_m\}$, donde J_m es el número total de nodos hoja del m -ésimo árbol. Dado j , la predicción de los individuos pertenecientes a R_{jm} vendrá dada por:

$$\gamma_{jm} = \arg \min_{\gamma} \sum_{x_i \in R_{jm}} L(y_i, f_{m-1}(x_i) + \gamma)$$

Para obtener el γ que minimiza dicha expresión, se aproxima $L(x_i, f_{m-1}(x_i) + \gamma)$ mediante su desarrollo de Taylor de segundo orden. Se hará para un x_i cualquiera con el fin de no complicar los cálculos con los sumatorios. Es decir, se considera:

$$L(y_i, f_{m-1}(x_i) + \gamma) \approx L(y_i, f_{m-1}(x_i)) + \frac{d}{d(\ln(\text{odd}_i))} L(y_i, f_{m-1}(x_i)) \gamma + \frac{1}{2} \frac{d^2}{d(\ln(\text{odd}_i))^2} L(y_i, f_{m-1}(x_i)) \gamma^2$$

y se deriva respecto de γ igualando a cero puesto que es lo que se busca minimizar:

$$\begin{aligned} \frac{d}{d\gamma} L(y_i, f_{m-1}(x_i) + \gamma) &\approx \frac{d}{d(\ln(\text{odd}_i))} L(y_i, f_{m-1}(x_i)) + \frac{d^2}{d(\ln(\text{odd}_i))^2} L(y_i, f_{m-1}(x_i)) \gamma \Rightarrow \\ \Rightarrow \gamma &= - \frac{\frac{d}{df_{m-1}(x_i)} L(y_i, f_{m-1}(x_i))}{\frac{d^2}{df_{m-1}(x_i)^2} L(y_i, f_{m-1}(x_i))} \end{aligned}$$

Realizando el cálculo de estas derivadas se obtiene:

$$\gamma = \frac{y_i - \frac{e^{f_{m-1}(x_i)}}{1 + e^{f_{m-1}(x_i)}}}{\frac{e^{f_{m-1}(x_i)}}{(1 + e^{f_{m-1}(x_i)})^2}} \quad (6)$$

Además, realizando las transformaciones adecuadas, se llega a que esta expresión es equivalente a:

$$\gamma = \frac{y_i - p_{i,m-1}}{p_{i,m-1}(1 - p_{i,m-1})} \quad (7)$$

Para finalizar con los cálculos, recuérdese que se ha decidido omitir los sumatorios en el desarrollo realizado. Ahora bien, como la derivada de un sumatorio es igual al sumatorio de las derivadas, se puede afirmar que:

$$\gamma_{jm} = \frac{\sum_{x_i \in R_{jm}} y_i - \frac{e^{f_{m-1}(x_i)}}{1 + e^{f_{m-1}(x_i)}}}{\sum_{x_i \in R_{jm}} \frac{e^{f_{m-1}(x_i)}}{(1 + e^{f_{m-1}(x_i)})^2}} = \frac{\sum_{x_i \in R_{jm}} y_i - p_{i,m-1}}{\sum_{x_i \in R_{jm}} p_{i,m-1}(1 - p_{i,m-1})}$$

A partir de aquí el procedimiento es equivalente al visto en el caso de Gradient Boosting para regresión: se actualiza el valor de la predicción $(m - 1)$ -ésima a partir de los γ_{jm} y la predicción final será la predicción M -ésima, es decir, la correspondiente a la última iteración del bucle.

Por último, cabe recordar que en este caso estamos realizando predicciones de log-odd-ratio; por tanto, para obtener la probabilidad deseada y clasificar en una clase o en la otra, habrá que llevar a cabo la transformación:

$$p = \frac{e^{\ln(odd)}}{1 + e^{\ln(odd)}}$$

3.2.3. XGBoost

XGBoost(*'eXtreme Gradient Boosting'*) [4] es un algoritmo que utiliza de base lo explicado en la última sección añadiendo una serie de mejoras que lo convierten en una herramienta muy potente. Actualmente, es uno de los modelos cuyo dominio es imprescindible y su continuada aparición entre los finalistas de concursos de Machine Learning dan fe de su importancia y utilidad.

Este algoritmo utiliza la misma idea de Gradient Boosting: generar árboles de decisión que realicen predicciones secuenciales de los residuos. No obstante, presenta una serie de modificaciones entre las que destacan la manera de construir los árboles y la inclusión de parámetros de regularización, como λ y γ , que ayudan a evitar el sobreajuste restando importancia a los datos aislados y aportando aleatoriedad. Además, XGBoost fue diseñado para trabajar con conjuntos de datos muy grandes, por lo que estos parámetros de

regularización serán de ayuda, sobretodo, en caso de que el conjunto de datos no sea lo suficientemente grande.

A continuación, al igual que se realizó en Gradient Boosting, se explicarán el caso de regresión y el de clasificación por separado.

Regresión

En el caso de regresión, la Loss Function elegida es la misma que se escogió en Gradient Boosting:

$$L(y_i, f(x_i)) = \frac{1}{2}(y_i - f(x_i))^2$$

El primer paso consiste en definir la predicción inicial, denotada por $f^{(0)}$. Como ya se ha visto en cálculos anteriores (sección 3.2.2), esta primera predicción vendrá dada por:

$$f^{(0)}(x) = \frac{1}{N} \sum_{i=1}^N y_i$$

A continuación, interviene un bucle de M iteraciones en el que, en cada iteración, se generará un árbol que prediga los residuos respecto de la última predicción calculada. Así para la m -ésima iteración calculamos, de manera análoga a los r_{im} de Gradient Boosting:

$$g_m(x_i) = \left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)} \right]_{f=f^{(m-1)}} = f^{(m-1)}(x_i) - y_i$$

Por otro lado, se tiene:

$$h_m(x_i) = \left[\frac{\partial^2 L(y_i, f(x_i))}{\partial f(x_i)^2} \right]_{f=f^{(m-1)}} = 1$$

A continuación se genera un árbol de decisión que clasifique los residuos. La respuesta de estos árboles vendrá dada por los valores ω_j (valor asociado al j -ésimo nodo hoja) que minimicen la siguiente función:

$$\mathcal{L}^{(m)} = \sum_{i=1}^n L(y_i, f^{(m-1)}(x_i) + f_m(x_i)) + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^T \omega_j^2$$

donde γ y λ son parámetros de regularización. Esto es así pues se desea minimizar el error cometido por la predicción m -ésima, que vendrá dada por: $f^m(x_i) = f^{(m-1)}(x_i) + f_m(x_i)$, donde f_m hace referencia al m -ésimo árbol generado. Para minimizar esta función se

recurre al desarrollo de Taylor de segundo orden de $L(y_i, f^{(m-1)}(x_i) + f_m(x_i))$ de manera que:

$$\mathcal{L}^{(m)} \approx \sum_{i=1}^n \left(L(y_i, f^{(m-1)}(x_i)) + g_m(x_i)f_m(x_i) + h_m(x_i)f_m(x_i)^2 \right) + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^T \omega_j^2$$

Ahora, teniendo en cuenta que $\sum_{i=1}^n f_m(x_i) = \sum_{j=1}^T \sum_{i \in I_j} \omega_j$ con $I_j = \{i / x_i \in R_j\}$, la expresión anterior se puede reescribir de la forma:

$$\mathcal{L}^{(m)} \approx \sum_{j=1}^T \left(\left(\sum_{i \in I_j} g_m(x_i) \right) \omega_j + \frac{1}{2} \left(\sum_{i \in I_j} h_m(x_i) + \lambda \right) \omega_j^2 \right) + \gamma T$$

Así, derivando respecto de ω_j e igualando a 0 se tiene que:

$$\omega_j^* = - \frac{\sum_{i \in I_j} g_m(x_i)}{\sum_{i \in I_j} h_m(x_i) + \lambda} = - \frac{\sum_{i \in I_j} f^{(m-1)}(x_i) - y_i}{\sum_{i \in I_j} 1 + \lambda}$$

expresión que se puede interpretar como $\frac{\sum_{i \in I_j} \text{Residuos}_i}{\text{Num_Residuos}_j + \lambda}$. De esta manera, se observa que para nodos hoja formados por muchos residuos, el efecto del parámetro λ será mínimo y el ω_j se corresponderá, prácticamente, con la media de los valores del nodo. Esto es porque, para N_j (número de residuos del j -ésimo nodo hoja) grande:

$$\omega_j = \frac{\sum_{i \in I_j} \text{Residuos}_i}{N_j + \lambda} = \frac{\sum_{i \in I_j} \text{Residuos}_i}{N_j} \times \frac{N_j}{N_j + \lambda} \approx \frac{\sum_{i \in I_j} \text{Residuos}_i}{N_j}$$

Así, λ actúa como parámetro de regularización ya que modifica considerablemente la respuesta de aquellos nodos hoja a los que corresponden pocos individuos; es decir, la respuesta de las observaciones aisladas. Aumentar λ implica disminuir los pesos y aumentar su coste en la Loss Function. En resumen, escogiendo un $\lambda \gg 1$ se reduce el *overfitting*.

Una vez visto qué valor asignar a los nodos hoja de los árboles que se construyen en XGBoost, veamos la profundidad que alcanzan estos árboles. Para ello, sustituyendo el valor ω_j en la ecuación de $\mathcal{L}^{(m)}$ se obtiene:

$$\mathcal{L}^{*(m)} = -\frac{1}{2} \sum_{j=1}^T \frac{\left(\sum_{i \in I_j} g_m(x_i) \right)^2}{\sum_{i \in I_j} h_m(x_i) + \lambda} + \gamma T$$

El criterio de escisión de nodos de los árboles se basará en esta última fórmula. Dado un nodo I , se realiza una división de manera que $I = I_L \cup I_R$, donde I_L hace referencia al nodo hijo izquierdo e I_R al derecho. Así, se define una función de ganancia (“*Gain*”) que realiza un trabajo análogo al realizado por la función de impureza explicada en CART (sección 2.3):

$$Gain_I = \frac{G_I}{H_I} = \frac{(\sum_{i \in I} g_m(x_i))^2}{\sum_{i \in I} h_m(x_i) + \lambda}$$

Entonces, las variables y los valores escogidos como referencia para la escisión serán aquellos que maximicen la función:

$$\begin{aligned} \mathcal{L}_{split} &= \frac{1}{2} (Gain_{I_L} + Gain_{I_R} - Gain_I) - \gamma = \frac{1}{2} \left(\frac{G_{I_L}}{H_{I_L}} + \frac{G_{I_R}}{H_{I_R}} - \frac{G_I}{H_I} \right) - \gamma = \\ &= \frac{1}{2} \left(\frac{(\sum_{i \in I_L} g_m(x_i))^2}{\sum_{i \in I_L} h_m(x_i) + \lambda} + \frac{(\sum_{i \in I_R} g_m(x_i))^2}{\sum_{i \in I_R} h_m(x_i) + \lambda} - \frac{(\sum_{i \in I} g_m(x_i))^2}{\sum_{i \in I} h_m(x_i) + \lambda} \right) - \gamma \end{aligned}$$

y el criterio de escisión será dividir en caso de que $\mathcal{L}_{split} > 0$ y no dividir en caso de que $\mathcal{L}_{split} < 0$. En este criterio, se puede apreciar la función de parámetro de regularización que realiza γ , ya que cuanto mayor sea el γ , menor será la probabilidad de dividir el nodo. De esta manera, para $\gamma \gg 0$, los árboles generados no serán muy profundos y se evitará el sobreajuste; y para $\gamma \simeq 0$, los árboles se generarán de manera parecida a CART: si la ganancia de los nodos hijos es mayor que la del nodo padre, se divide.

En particular, dada la Loss Function fijada:

$$\mathcal{L}_{split} = \frac{1}{2} \left(\frac{(\sum_{i \in I_L} f^{(m-1)}(x_i) - y_i)^2}{\sum_{i \in I_L} 1 + \lambda} + \frac{(\sum_{i \in I_R} f^{(m-1)}(x_i) - y_i)^2}{\sum_{i \in I_R} 1 + \lambda} - \frac{(\sum_{i \in I} f^{(m-1)}(x_i) - y_i)^2}{\sum_{i \in I} 1 + \lambda} \right) - \gamma$$

que, de nuevo, se puede entender como el sumatorio de los residuos al cuadrado dividido por el número de residuos menos γ .

Una vez explicado como genera los árboles XGBoost, el resto del procedimiento es análogo a Gradient Boosting. La predicción realizada por el árbol m -ésimo se añade a la última predicción realizada (la $(m-1)$ -ésima) y se toma como predicción final del modelo $f^{(M)}(x)$.

Clasificación

Al igual que en Gradient Boosting, la Loss Function que se suele emplear en XGBoost es:

$$L(y_i, \ln(odd_i)) = -y_i \ln(odd_i) + \ln(1 + e^{\ln(odd_i)})$$

Ahora, las derivadas de la iteración m -ésima serán:

$$g_m(x_i) = -y_i + \frac{e^{f^{(m-1)}(x_i)}}{1 + e^{f^{(m-1)}(x_i)}} \quad \wedge \quad h_m(x_i) = \frac{e^{f^{(m-1)}(x_i)}}{(1 + e^{f^{(m-1)}(x_i)})^2}$$

que se pueden escribir en términos de las probabilidades predichas $p_{i,m-1}$ (visto en Gradient Boosting):

$$g_m(x_i) = -y_i + p_{i,m-1} \quad \wedge \quad h_m(x_i) = p_{i,m-1}(1 - p_{i,m-1})$$

Así, para obtener el criterio de escisión de nodos seguido en la construcción de los árboles se deben sustituir estos valores en:

$$\mathcal{L}_{split} = \frac{1}{2} \left(\frac{(\sum_{i \in I_L} g_m(x_i))^2}{\sum_{i \in I_L} h_m(x_i) + \lambda} + \frac{(\sum_{i \in I_R} g_m(x_i))^2}{\sum_{i \in I_R} h_m(x_i) + \lambda} - \frac{(\sum_{i \in I} g_m(x_i))^2}{\sum_{i \in I} h_m(x_i) + \lambda} \right) - \gamma$$

y para obtener el valor de los ω_j^* basta con sustituir en:

$$\omega_j^* = - \frac{\sum_{i \in I_j} g_m(x_i)}{\sum_{i \in I_j} h_m(x_i) + \lambda}$$

mientras que el resto del procedimiento es análogo.

De nuevo, cabe tener en cuenta que la predicción vendrá dada en términos de log-odd-ratio, por lo que habrá que realizar la transformación:

$$p = \frac{e^{\ln(odd)}}{1 + e^{\ln(odd)}}$$

para obtener la probabilidad deseada y clasificar.

Otras optimizaciones

Además de la particular forma que tiene XGBoost de construir los árboles, también incluye otra serie de mecanismos que hacen de él una herramienta muy potente. A continuación, se explicará, sin entrar en muchos detalles, cuáles son estos métodos que aplica.

Por un lado, como XGBoost se ideó para conjuntos de datos muy grandes, a la hora de realizar las escisiones de los árboles, probar con todos los valores posibles de todas las variables para determinar cuáles maximizan la función \mathcal{L}_{split} conllevaría un gasto computacional inasumible. Así, una manera eficaz de llevar a cabo esta tarea es probar solo con los percentiles de cada variable. Por ejemplo, tomando los percentiles P_{25} , P_{50} y

P_{75} , solo habrá que probar con 3 valores por variable y escoger el óptimo. En la práctica, se suelen usar alrededor de 33 percentiles. Además, estos percentiles están ponderados; esto quiere decir que no todos los intervalos recogen el mismo número de individuos, sino que la suma de los pesos de los individuos es igual en todos los intervalos. El peso de cada individuo en la iteración m -ésima viene dado por $h_m(x_i)$ que, como ya se ha visto:

$$h_m(x_i) = 1 \quad \vee \quad h_m(x_i) = p_{i,m-1}(1 - p_{i,m-1})$$

según se trate de un problema de regresión o de clasificación, respectivamente. Así, en el caso de regresión los percentiles se calculan de la forma habitual y en el caso de clasificación se tendrán en cuenta las últimas predicciones realizadas. La idea intuitiva de por qué los pesos vienen dados por $h_m(x_i)$ pasa por observar que se puede reescribir la ecuación de $\mathcal{L}^{(m)}$ como:

$$\mathcal{L}^{(m)} = \sum_{i=1}^n \frac{1}{2} h_m(x_i) \left(f_m(x_i) - \frac{g_m(x_i)}{h_m(x_i)} \right)^2 + \Omega(f_m) + constant$$

con $\Omega(f_m)$ los términos de regularización, de manera que se puede interpretar considerando $h_m(x_i)$ el peso del individuo y el otro factor como una función de pérdida.

Por otro lado, XGBoost es capaz de trabajar con conjuntos de datos con datos ausentes (o “*Null values*”) sin necesidad de depurarlos previamente. Cuando se construyen los árboles para predecir los residuos, a la hora de separar un nodo, si se tienen variables con datos vacíos, para cada uno de los valores que se le otorguen a la variable para analizar el valor de “*Gain*” se contemplarán dos casos: que los residuos de aquellos datos vacíos se encuentren en el nodo hijo izquierdo y que se encuentren en el derecho. De esta manera, el algoritmo tendrá en cuenta más particiones pero seguirá escogiendo aquella que maximice la ganancia y, por tanto, se podrán predecir resultados sin necesidad de conocer el valor de todas las variables explicativas.

Por último, también se tienen en cuenta ciertas mejoras desde el punto de vista de la implementación. Con la intención de optimizar el tiempo de computación, XGBoost recurre a diferentes técnicas:

- **Paraleliza el conjunto de datos** enviando fragmentos del mismo a diferentes terminales.
- **Calcula $g_m(x_i)$ y $h_m(x_i)$ haciendo uso de la memoria RAM** para realizar las operaciones necesarias en el menor tiempo posible.

- **Realiza una compresión del conjunto de datos** ya que es más eficiente comprimirlos y descomprimirlos que leerlos directamente del disco duro. Además, en caso de disponer de más de un disco duro, separa el conjunto entre los disponibles con la finalidad de ganar velocidad.

3.2.4. LightGBM

LightGBM (del inglés: ***Light Gradient Boosting Machine***) [5] es un algoritmo desarrollado por Microsoft que fue publicado en 2017. Su funcionamiento es similar al de XGBoost pero añadiendo una serie de matices que consiguen obtener modelos muy potentes y eficientes.

La diferencia principal reside en el modo de construir los árboles. Como se puede observar en la Figura 8, en cada capa de nodos, XGBoost realiza la división de todos ellos (*Level-wise tree growth*) y luego evalúa si dicha evaluación se debe llevar a cabo a través del valor de “*Gain*”. Por el contrario, LightGBM no trabaja por capas (*Leaf-wise tree growth*) sino que, a la hora de generar un árbol de decisión, se decide separar el nodo raíz y luego, de los dos nodos resultantes, sólo se separará el nodo que maximice la ganancia. Así, en la siguiente iteración, se contará con tres nodos que podremos dividir y, de nuevo, se valorará cual de los tres maximiza la ganancia y será dividido. Por tanto, en cada iteración de la construcción del árbol, se calculará la ganancia de cada escisión y sólo se realizará aquella que la maximice.

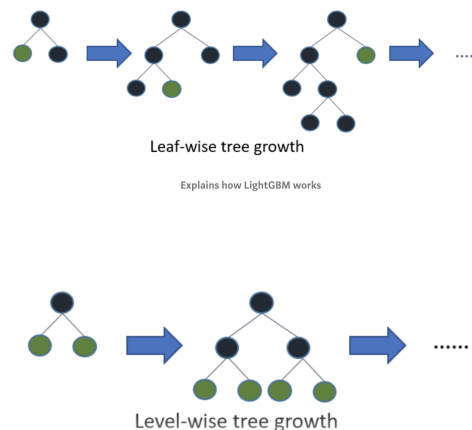


Figura 8: Construcción de árboles en LightGBM vs XGBoost (Fuente: [11])

Además de esta particularidad de LightGBM, también se aplican diferentes modificaciones computacionales que mejoran el tiempo de ejecución considerablemente, consiguiendo a veces una mejor precisión que XGBoost en un tiempo menor. Por ejemplo,

LightGBM está diseñado para ser muy veloz al hacer uso de la CPU, procesador presente en todos los ordenadores. Por el contrario, XGBoost obtiene mejores resultados de tiempo al trabajar con la GPU, procesador gráfico que no poseen todos los equipos. Por tanto, al trabajar en CPU ambos, los tiempos de ejecución de LightGBM son mejores que los de XGBoost.

3.2.5. CatBoost

CatBoost [6] es un modelo desarrollado por Yandex también publicado en 2017. Al igual que LightGBM, guarda gran parecido con XGBoost; no obstante, se trata de una gran alternativa frente a problemas que cuentan con un gran número de variables explicativas categóricas (de ahí su nombre: *Categorical Boosting*).

A la hora de trabajar con modelos de Machine Learning, dado que los ordenadores no son capaces de entender cadenas de texto, es necesario codificar las variables categóricas del conjunto de datos con el que se trabaja. La codificación de una variable categórica binaria es sencilla, pues basta con asignar a las clases los valores 0 y 1 respectivamente. No obstante, cuando la variable categórica posee más de dos clases, la manera de proceder no resulta tan simple e intuitiva. Una solución a este problema es la codificación *one-hot-encoding*. Si se cuenta con m clases diferentes A_1, A_2, \dots, A_m , se trata de crear m nuevas variables V_1, V_2, \dots, V_m de manera que:

$$V_j(x) = \begin{cases} 1 & \text{si } x \in A_j \\ 0 & \text{si } x \notin A_j \end{cases}$$

En la Tabla 9 se muestra un ejemplo de este tipo de codificación.

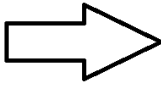
Individuo	Clase		Individuo	ClaseA	ClaseB	ClaseC
1	A	\Rightarrow	1	1	0	0
2	B		2	0	1	0
3	A		3	1	0	0
4	C		4	0	0	1

Tabla 9: *One-hot-encoding*

De esta manera, el problema quedaría resuelto. Cada nueva variable V_j hará referencia a la propiedad de “pertenecer a la clase A_j ” y el conjunto de datos ya estará correctamente codificado para su uso. En este trabajo ha sido necesario realizar un preprocesado de los datos y se ha utilizado esta técnica para codificar variables como “Location”.

No obstante, pese a haber solucionado el problema, la codificación *one-hot-encoding* no resulta del todo eficiente ya que la presencia de muchas variables categóricas implica un aumento significativo del número de variables a considerar en el conjunto de datos, lo que se traduce en un mayor uso de la memoria y en un mayor coste computacional. También, podría provocar problemas de multicolinealidad en algoritmos clásicos como el de Regresión Lineal. Por tanto, la principal innovación de CatBoost respecto al resto de algoritmos es que consigue codificar las variables categóricas sin necesidad de recurrir a la codificación *one-hot-encoding*. Así, se consigue un ahorro de recursos que permite obtener resultados similares en un menor tiempo de ejecución.

En el caso de clasificación, la codificación realizada por CatBoost se basa en intentar codificar cada clase A_j con el valor de $P(Y = 1|X \in A_j)$. Dado que este valor es desconocido, un razonamiento lógico sería proceder a través de proporciones. Se verá con el apoyo de un ejemplo, haciendo uso del conjunto de datos de la Tabla 10, donde “Country” es la variable explicativa que se desea codificar y “Target” la variable binaria a predecir. Entonces, la idea es codificar “Spain” teniendo en cuenta que aparece 5 veces y toma el valor 1 en 3 ocasiones. Así, “Spain” se codificaría como $\frac{3}{5}$. No obstante, un modelo construido de esta manera se sobreajustaría dado que estaríamos prediciendo “Target” en función de “Country” cuando a su vez “Country” ha sido codificada en función de “Target”. Entonces, la manera de proceder no será exactamente la anterior, pero sí muy parecida.



...	Country	...	Target
0	Spain	...	1
1	Spain	...	0
2	France	...	0
3	Australia	...	0
4	France	...	1
5	Spain	...	1
6	Spain	...	0
7	Colombia	...	0
8	Spain	...	1

...	Country	...	Target
0	Spain	...	1
1	Spain	...	0
2	France	...	0
3	Australia	...	0
4	France	...	1
5	Spain	...	1
6	Spain	...	0
7	Colombia	...	0
8	Spain	...	1

Tabla 10: Conjunto de datos del ejemplo teórico

Para evitar el sobreajuste del modelo, basta con añadir parámetros de regularización y técnicas relacionadas con la validación cruzada. En primer lugar, se codifica la variable de la manera propuesta anteriormente, a través de proporciones, tal y como se puede observar en la Tabla 11.

	...	Country	Country_encoded	Target
0	...	Spain	0.8	1
1	...	Spain	0.8	0
2	...	France	0
3	...	Australia	0
4	...	France	1
5	...	Spain	0.8	1
6	...	Spain	0.8	0
7	...	Colombia	0
8	...	Spain	0.8	1
9

Tabla 11: Codificación a través de proporciones de todo el conjunto de datos de ejemplo

A continuación, se separa el conjunto de datos en dos de manera que el primer subconjunto mantendrá el valor asignado y el segundo no. En nuestro ejemplo, dicho proceso se encuentra reflejado en la Tabla 12.

	...	Country	Country_encoded	Target
0	...	Spain	0.8	1
1	...	Spain	0.8	0
2	...	France	0
3	...	Australia	0
4	...	France	1
5	...	Spain	0.8	1
6	...	Spain	???	0
7	...	Colombia	0
8	...	Spain	???	1
9

Tabla 12: Separación del conjunto de datos de ejemplo

Las variables del segundo conjunto se codificarán, secuencialmente, en función de aquellos individuos que ya han sido codificados. La fórmula es la siguiente:

$$\frac{Total_Target_Clase + a \times p}{Total_Clase + a}$$

donde $Total_Target_Clase$ hace referencia al sumatorio de $Target$ de los individuos ya codificados que pertenecen a la clase a codificar, p es el valor de codificación calculado mediante proporción, $Total_Clase$ es el valor de individuos ya codificados que pertenecen

a la clase en cuestión y a es un parámetro de regularización. Es decir, para codificar al sexto individuo del ejemplo, es necesario fijar un a (por ejemplo, $a = 1$) y realizar :

$$\frac{Total_Target_Clase + a \times p}{Total_Clase + a} = \frac{(1 + 0 + 1) + 0,8}{3 + 1} = 0,7$$

De esta manera, se mantiene el valor de los individuos anteriores y se codifica el sexto individuo con el valor obtenido, tal y como se recoge en la Tabla 13.

...	Country	Country_encoded	Target	
0	...	Spain	0.8	1
1	...	Spain	0.8	0
2	...	France	0
3	...	Australia	0
4	...	France	1
5	...	Spain	0.8	1
6	...	Spain	0.7	0
7	...	Colombia	0
8	...	Spain	???	1
9

Tabla 13: Codificación del sexto individuo del ejemplo

Para el octavo individuo, basta con repetir los cálculos:

$$\frac{Total_Target_Clase + p}{Total_Clase + 1} = \frac{(1 + 0 + 1 + 0) + 0,8}{4 + 1} = \frac{2,8}{5} \simeq 0,56$$

Así, el conjunto de datos propuesto queda codificado como se muestra en la Tabla 14.

...	Country	Country_encoded	Target	
0	...	Spain	0.8	1
1	...	Spain	0.8	0
2	...	France	0
3	...	Australia	0
4	...	France	1
5	...	Spain	0.8	1
6	...	Spain	0.7	0
7	...	Colombia	0
8	...	Spain	0.56	1
9

Tabla 14: Codificación del octavo individuo del ejemplo

Con el fin de ajustarse con la mayor exactitud posible al funcionamiento real del modelo, cabe mencionar el hecho de que CatBoost realiza permutaciones del conjunto de datos antes de dividirlo en dos y realiza este proceso varias veces para exhibir finalmente una media del valor obtenido. Además, se ha explicado el proceso para el caso de clasificación, pero el caso de regresión no presenta grandes diferencias; lo único que se debe tener en cuenta es que la p (que antes ha sido definida como la proporción de individuos cuyo valor en la variable dependiente es “1”) ahora será la media (es decir, la media de los valores de la variable dependiente para los individuos que pertenecen a la clase que interesa codificar).

Además, al igual que LightGBM y XGBoost, CatBoost añade mejoras del ámbito informático para ciertas optimizaciones. Por ejemplo, está implementado de manera que es muy eficiente al trabajar con procesadores GPU. También, cabe destacar que este modelo construye árboles de decisión simétricos, como se recoge en la Figura 9. La construcción de árboles simétricos consigue que trabajar con procesadores CPU sea mucho más eficiente, además de actuar como regularizador. De esta manera, CatBoost consigue reducir el coste tanto en CPU como en GPU.

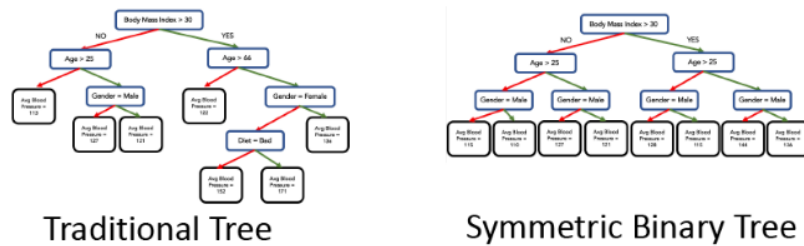


Figura 9: Árboles simétricos. Fuente: [13]

3.2.6. Ejemplo

Una vez más, se hará una predicción de la probabilidad de lluvia del día próximo con nuestro conjunto de datos “Rain in Australia”. En este caso, no es necesario establecer un criterio de parada en la construcción de los árboles, dado que la estructura de los modelos de boosting hace que estos no se sobreajusten pese a construir árboles muy profundos. No obstante, por realizar una comparación justa con los algoritmos de bagging, se fijará la profundidad máxima de cada árbol en 5.

Aplicando los algoritmos estudiados, se llega a los resultados expuestos en la Tabla 15. Como se puede observar, en cualquiera de las tablas se puede escoger un modelo que

mantenga el delta% por debajo del 5% y que a su vez tenga un coeficiente de Gini para el conjunto de testeo superior al 70%. Por tanto, se observa claramente que los modelos de boosting suponen una mejora significativa de la precisión del modelo. No obstante, para alcanzar esta precisión, en ocasiones es necesario realizar un número elevado de iteraciones, con el correspondiente coste computacional que conlleva.

a)	Run_Time	Train_Gini%	Test_Gini%	delta%
AdaB_1	0.7	36.0	32.0	-9.0
AdaB_3	1.2	44.0	39.0	-9.0
AdaB_5	1.8	54.0	52.0	-4.0
AdaB_10	3.0	57.0	54.0	-5.0
AdaB_15	4.9	62.0	57.0	-7.0
AdaB_20	6.7	64.0	61.0	-6.0
AdaB_30	8.9	66.0	63.0	-6.0
AdaB_50	15.4	69.0	66.0	-4.0
AdaB_100	30.4	71.0	69.0	-3.0
AdaB_200	59.4	73.0	71.0	-2.0
AdaB_500	148.4	74.0	73.0	-2.0
AdaB_1000	291.5	75.0	73.0	-2.0

b)	Run_Time	Train_Gini%	Test_Gini%	delta%
GB_1	1.3	66.0	63.0	-4.0
GB_3	2.8	69.0	66.0	-4.0
GB_5	4.3	70.0	67.0	-4.0
GB_10	8.4	72.0	69.0	-4.0
GB_15	12.5	73.0	70.0	-3.0
GB_20	15.7	74.0	71.0	-3.0
GB_30	22.3	75.0	73.0	-4.0
GB_50	38.5	78.0	74.0	-4.0
GB_100	74.6	80.0	76.0	-5.0
GB_200	158.1	83.0	77.0	-7.0
GB_500	402.2	88.0	78.0	-12.0
GB_1000	771.6	92.0	77.0	-16.0

c)	Run_Time	Train_Gini%	Test_Gini%	delta%
XGB_1	0.5	66.0	63.0	-4.0
XGB_3	0.6	69.0	66.0	-5.0
XGB_5	0.6	70.0	67.0	-4.0
XGB_10	0.9	71.0	68.0	-4.0
XGB_15	1.1	73.0	70.0	-3.0
XGB_20	1.3	74.0	71.0	-4.0
XGB_30	1.7	75.0	73.0	-4.0
XGB_50	2.6	78.0	74.0	-4.0
XGB_100	4.8	80.0	76.0	-5.0
XGB_200	10.4	83.0	77.0	-7.0
XGB_500	32.8	88.0	78.0	-11.0
XGB_1000	65.6	92.0	78.0	-15.0

d)	Run_Time	Train_Gini%	Test_Gini%	delta%
LGBM_1	0.5	66.0	63.0	-4.0
LGBM_3	0.5	69.0	66.0	-4.0
LGBM_5	0.5	70.0	68.0	-4.0
LGBM_10	0.5	72.0	70.0	-3.0
LGBM_15	0.6	73.0	71.0	-3.0
LGBM_20	0.6	74.0	71.0	-4.0
LGBM_30	0.7	76.0	73.0	-4.0
LGBM_50	0.8	78.0	74.0	-4.0
LGBM_100	0.9	80.0	76.0	-5.0
LGBM_200	1.4	83.0	77.0	-7.0
LGBM_500	2.7	88.0	78.0	-12.0
LGBM_1000	5.7	93.0	78.0	-16.0

e)	Run_Time	Train_Gini%	Test_Gini%	delta%
CatB_1	1.3	65.0	62.0	-4.0
CatB_3	1.2	70.0	67.0	-4.0
CatB_5	1.4	72.0	71.0	-2.0
CatB_10	1.9	74.0	72.0	-2.0
CatB_15	2.2	75.0	73.0	-3.0
CatB_20	2.8	76.0	73.0	-3.0
CatB_30	3.7	78.0	74.0	-4.0
CatB_50	5.3	79.0	75.0	-5.0
CatB_100	9.1	82.0	75.0	-8.0
CatB_200	24.9	85.0	78.0	-8.0
CatB_500	62.2	86.0	79.0	-8.0
CatB_1000	120.8	86.0	79.0	-8.0

Tabla 15: Resultados modelos de boosting: a)AdaBoost b)Gradient Boosting c)XGBoost d)LightGBM e)CatBoost

Además, no hay que obviar el hecho de que muchos de estos modelos están sobreajustados. Mientras que al trabajar con modelos de bagging no surgen problemas de overfitting, con modelos de boosting habrá que preocuparse por ver que no se están cometiendo errores en la elección del número de iteraciones u otros parámetros de regularización. También, atendiendo a los tiempos de ejecución, se puede observar la eficiencia de LightGBM ya que donde algunos algoritmos tardan entre 1 y 10 minutos en entrenarse, LGBM apenas necesita 5 segundos. En la Figura 15, se pueden comprobar cómo aumenta la diferencia entre conjuntos y el tiempo de ejecución, respectivamente, a medida que se añaden árboles a los modelos.

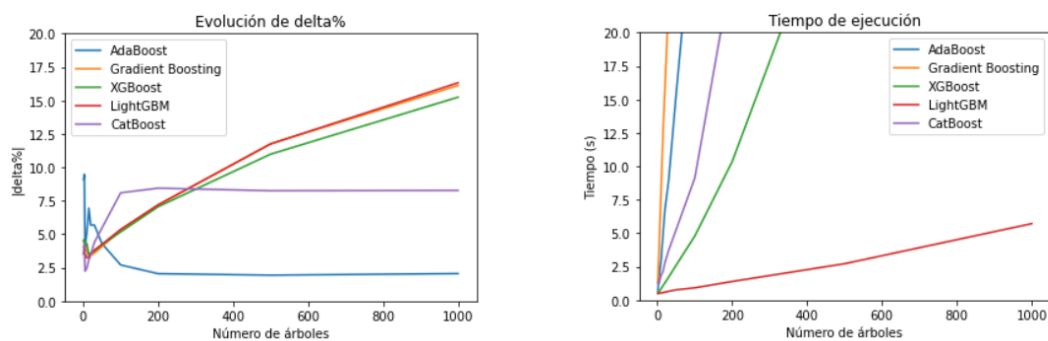


Figura 10: Evolución del $\delta\%$ y del tiempo de ejecución

3.2.7. Otros parámetros

Los diferentes algoritmos de boosting se han desarrollado explicando la base del funcionamiento de los mismos y los principales parámetros a tener en cuenta. No obstante, la implementación de estos algoritmos contiene un gran número de parámetros que se pueden ajustar a gusto del usuario, dando lugar a modelos muy versátiles. Uno de los parámetros a tener en cuenta que incorpora XGBoost y, por extensión, LightGBM y CatBoost es conocido como *early stopping*. Este parámetro permite a los algoritmos mencionados “autoregularse”, de manera que los modelos construidos no se sobreajusten. En la práctica, se evalúa el incremento de precisión que se consigue en cada paso en el conjunto test; si este incremento no supera la cota establecida, el algoritmo terminará con el bucle. Esto es así porque se entiende que el resto de árboles conseguirán aumentar la precisión en el conjunto de entrenamiento y no en el de testeo. Sin embargo, realizar esta evaluación del incremento en cada paso supondría un coste elevado a la hora de entrenar el modelo. Es por eso que la librería propia *XGBoost* de Python contiene también el parámetro *early_stopping_rounds* a través del cual se puede controlar cada cuantas iteraciones se calculará el valor del incremento.

Por otro lado, cabe observar que a lo largo del trabajo se han tratado muchos algoritmos que han resultado ser muy útiles para problemas de regresión y de clasificación binaria. No obstante, muchos de los problemas de clasificación que se presentan no poseen únicamente dos clases posibles. Por ejemplo, si el problema fuese predecir la comunidad autónoma en la que reside un individuo dadas ciertas características del mismo, la respuesta del algoritmo no es binaria. La solución ante estos problemas de clasificación multiclase pasa por aplicar *one-hot-encoding* (visto en CatBoost (sección 3.2.5)) a la variable dependiente. De esta manera, se aplicará el modelo independientemente a cada nueva variable y se escogerá como respuesta final aquella que maximice la probabilidad.

Del mismo modo, en ocasiones las clases no tienen por qué ser exclusivas, si no que un individuo puede pertenecer a dos clases diferentes. Un ejemplo de esto sería el problema de clasificar una película en una determinada plataforma; pues una película puede pertenecer al género “Adulto” y “Comedia” simultáneamente. En este caso, la solución es la misma; se aplica *one-hot-encoding* a cada variable por separado y se seleccionan aquellas cuya probabilidad supera un cierto valor fijo. Además, la librería Scikit-Learn de Python incluye una implementación de este último caso, cuya salida es un vector de probabilidades.

3.2.8. Ventajas y desventajas

Ventajas:

- Herramientas muy potentes que ofrecen muy buenos resultados gracias a su corrección secuencial del error.
- No se sobreajustan ante grandes cantidades de datos gracias a que incluye parámetros de regularización.

Desventajas:

- Modelos de complejidad muy elevada (comprensión más costosa que en CART o algoritmos de bagging)
- Sólo apto para grandes conjuntos de datos, pues la gran potencia de los algoritmos conlleva sobreajustes cuando la cantidad de datos no es suficientemente elevada.
- Aumento del coste computacional.
- Modelos menos robustos ante datos ruidosos que los basados en bagging debido a la alta optimización.

4. Comparación de modelos

A lo largo del trabajo, se han utilizado los diferentes algoritmos explicados para construir modelos que, a partir de los datos de un día dado, predigan si habrá precipitaciones el día próximo. En esta sección, se resumirán los resultados obtenidos en los diferentes ejemplos.

En primer lugar, se realizará una comparación de los mejores modelos de cada categoría. Para cada algoritmo, se escogerá el modelo que maximice el valor del índice Gini para el conjunto de testeo y que, a su vez, no supere el 5% para el valor de *delta*%. De esta manera, los modelos seleccionados son los recogidos en la Tabla 16.

	Run_Time	Train_Gini%	Test_Gini%
DT_5	0.7	66.0	63.0
Bag_1000	425.1	70.0	67.0
RF_1000	56.3	70.0	67.0
ET_1000	35.0	66.0	63.0
AdaB_1000	291.5	75.0	73.0
GB_50	38.5	78.0	74.0
XGB_50	2.6	78.0	74.0
LGBM_50	0.8	78.0	74.0
CatB_30	3.7	78.0	74.0

Tabla 16: Modelos seleccionados

En la Figura 11 se encuentran recogidos los gráficos de la precisión en Gini y el tiempo de ejecución en función del modelo. Es evidente que los modelos que obtienen las mejores predicciones son aquellos que hacen uso de boosting, siendo el modelo más eficiente el *LGBM_50* dado que obtiene una de las mejores precisiones en el menor tiempo registrado. No obstante, los gráficos no son del todo justos con CatBoost ya que, pese a obtener un tiempo de ejecución superior al obtenido el LightGBM o XGBoost, hay que tener en cuenta que no se ha considerado el tiempo invertido en procesar los datos y codificar las variables categóricas para el correcto funcionamiento de los dos últimos.

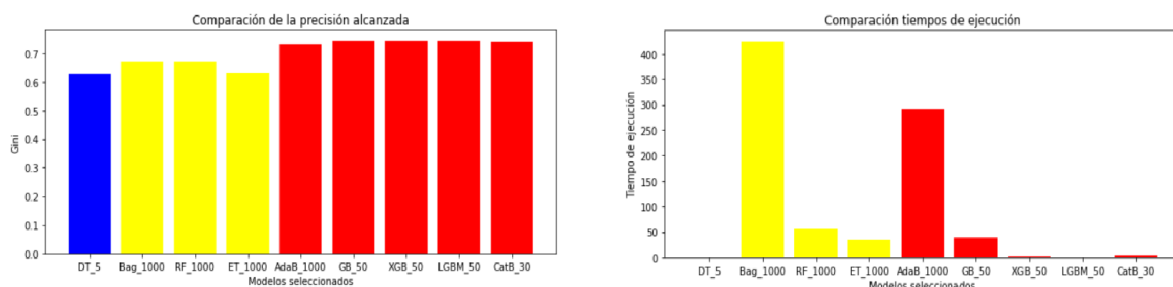


Figura 11: Comparación de precisión y tiempo de ejecución

Otros modelos a los que el gráfico no hace justicia son aquellos de Bagging. Según el criterio establecido, la selección de los modelos es la correcta; no obstante, dada la convergencia del error, se pueden obtener modelos cuya ejecución es mucho más rápida y cuya precisión apenas disminuye. Así, se representará la información en otro gráfico más completo.

En la Figura 12 se muestra la precisión en Gini de cada modelo en función del número de iteraciones e indica, a través del tamaño de los puntos, el tiempo de ejecución registrado; cuanto mayor sea este tiempo, mayor será el tamaño del punto. Con tal de poder observar bien la información, dado que el coeficiente de Gini converge a partir de las 100 iteraciones, no se contemplan las iteraciones posteriores. De esta manera, se comprueba visualmente que, por ejemplo, Gradient Boosting, XGBoost, LightGBM y CatBoost alcanzan un nivel de precisión muy parecido, siendo CatBoost el que antes converge y siendo Gradient Boosting el que conlleva un mayor coste computacional. Del mismo modo, observamos que AdaBoost es el que registra un mayor incremento de precisión; lo cual se corresponde con lo explicado en 3.2.1 ya que las primeras iteraciones sólo recogerán las predicciones realizadas por unos pocos tocones que son muy imprecisos.

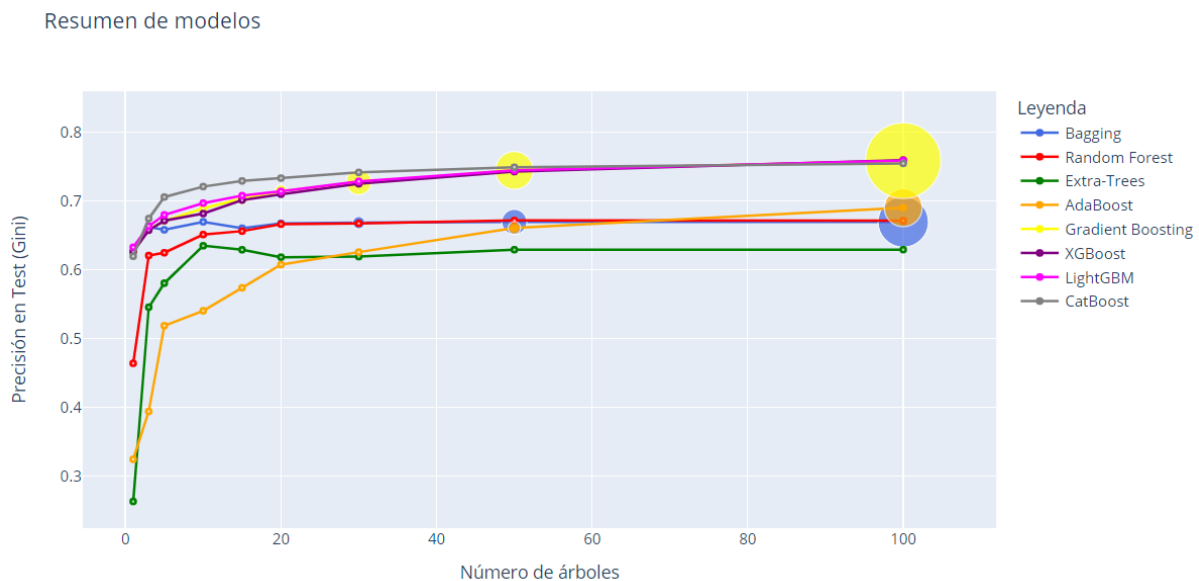


Figura 12: Resumen de modelos

5. Conclusión

A lo largo del trabajo se ha querido profundizar en diferentes modelos de Machine Learning que se construyen utilizando como base el algoritmo de los árboles de decisión. Desde los árboles más simples, como CART, se ha llegado a métodos más sofisticados que consiguen reducir los errores cometidos en las predicciones, e incluso evitar el sobreajuste, mediante diferentes técnicas. Estos métodos se clasifican en bagging y en boosting y, tras su estudio, se ha concluido que los algoritmos de bagging son una gran herramienta cuando los datos son muy ruidosos (pues no se sobreajustan), mientras que los de boosting son capaces de optimizar al máximo la construcción de árboles y dan lugar a modelos muy precisos.

Además, se ha hecho uso de un ejemplo extraído de Kaggle para poner en práctica estos algoritmos. El código se ha desarrollado en una serie de *notebooks* (disponibles en *GitHub* (A)) y, en ellos, se puede comprobar que se cumplen las conclusiones teóricas extraídas anteriormente: la convergencia del error en los algoritmos de bagging, la mejora en precisión de los algoritmos de boosting También se ha extraído otra serie de observaciones, como el hecho de que los algoritmos de boosting son más costosos dada la alta optimización que realizan, aunque algoritmos como XGBoost, LightGBM o CatBoost son capaces de obtener excelentes resultados en tiempos de ejecución bajos dado que son modelos muy bien implementados informáticamente.

Con el fin de sintetizar la información que se ha detallado, en las Tablas 17 y 18 se ha realizado un resumen de los algoritmos estudiados indicando los principales parámetros que intervienen en cada uno de ellos, su coste computacional, cómo controlar el overfitting De esta manera, se aspira a confeccionar un mapa conceptual de modelos, donde se puede realizar una consulta completa y, a su vez, esquemática sobre el funcionamiento de cualquiera de los algoritmos.

Algoritmo	Metodología	Implementación Python	Parámetros principales	Control <i>Overfitting</i>	Coste Computacional	Consideraciones
CART	Simple	Scikit-Learn: - DecisionTreeRegressor - DecisionTreeClassifier	- <i>max_depth</i> : Fija la profundidad máxima del árbol - <i>min_samples_leaf</i> : Fija el número mínimo de individuos en cada nodo hoja - <i>min_samples_split</i> : Fija el número mínimo de individuos que debe haber en un nodo para que sea divisible - <i>max_leaf_nodes</i> : Fija el número máximo de nodos hoja - <i>min_impurity_decrease</i> : Establece una cota al descenso de impureza a la hora de dividir un nodo - <i>n_estimators</i> : Número de árboles - <i>oob_score</i> : Devuelve el error OOB del modelo - <i>bootstrap</i> : Decide si las muestras se toman con repetición o no - <i>n_estimators</i> - <i>oob_score</i> - <i>bootstrap</i> - <i>m</i> : Número de variables que participan en la construcción de cada árbol	- Disminución de la profundidad máxima - Aumento del mínimo número de individuos por hoja - Aumento del número de individuos necesarios para la escisión de un nodo - Disminución del número máximo de nodos hoja - Aumento de la cota de impureza	Bajo	- Útil para exploración inicial de datos - Método sencillo de explicar - Admite representación gráfica - Inestable - Aumentar el <i>min_samples_split</i> o el <i>min_impurity_decrease</i> conlleva árboles menos profundos y con menos nodos hoja - No se puede establecer una relación de igualdad entre la profundidad máxima y el número máximo de nodos hoja dado que los árboles de decisión no tienen por qué ser simétricos.
Bagging	Bagging	Scikit-Learn: - BaggingRegressor - BaggingClassifier	- <i>n_estimators</i> : Número de árboles - <i>oob_score</i> : Devuelve el error OOB del modelo - <i>bootstrap</i> : Decide si las muestras se toman con repetición o no - <i>n_estimators</i> - <i>oob_score</i> - <i>bootstrap</i> - <i>m</i> : Número de variables que participan en la construcción de cada árbol	Control del <i>overfitting</i> de los árboles generados mediante alguna de las técnicas anteriores	Medio-Alto	- Uso de todas las variables explicativas disponibles - Convergencia del error
Random Forest	Bagging	Scikit-Learn: - RandomForestRegressor - RandomForestClassifier	- <i>n_estimators</i> - <i>oob_score</i> - <i>bootstrap</i> - <i>m</i> : Número de variables que participan en la construcción de cada árbol	- Control del <i>overfitting</i> de los árboles generados mediante alguna de las técnicas anteriores - Disminución de <i>m</i>	Medio-Bajo	- Cada árbol utiliza <i>m</i> variables - Disminución de la varianza - Convergencia del error
Extra-Trees	Bagging	Scikit-Learn: - ExtraTreesRegressor - ExtraTreesClassifier	- <i>n_estimators</i> - <i>oob_score</i> - <i>bootstrap</i> - <i>m</i>	- Control del <i>overfitting</i> de los árboles generados mediante alguna de las técnicas anteriores - Disminución de <i>m</i>	Medio-Bajo	- Mayor aleatoriedad - Útil ante problemas de <i>overfitting</i> extremos - Convergencia del error - Cada árbol utiliza <i>m</i> variables y solo chequea un valor por variable para escindir nodos

Tabla 17: Resumen de los algoritmos: CART, Bagging, Random Forest y Extra-Trees

Algoritmo	Metodología	Implementación Python	Parámetros principales	Control <i>Overfitting</i>	Coste Computacional	Consideraciones
AdaBoost	Boosting	Scikit-Learn: - AdaBoostRegressor - AdaBoostClassifier	- $n_estimators$	- Control del <i>overfitting</i> de los árboles generados mediante alguna de las técnicas de CART - Disminución del número de iteraciones	Medio	- Construcción de cadenas de árboles que reducen secuencialmente el error - Generación de árboles de dos nodos hoja
Gradient Boosting	Boosting	Scikit-Learn: - GradientBoostingRegressor - GradientBoostingClassifier	ν : Tasa de aprendizaje (<i>learning_rate</i>)	- Control del <i>overfitting</i> de los árboles generados mediante alguna de las técnicas de CART - Disminución de la tasa de aprendizaje - Disminución del número de iteraciones	Alto	- Construcción de cadenas de árboles que reducen secuencialmente el error - Suele aplicarse con un elevado número de iteraciones - Suele aplicarse con un elevado número de iteraciones
XGBoost	Boosting	XGBoost: - XGBRegressor - XGBClassifier	- η : Tasa de aprendizaje (<i>eta</i> o <i>learning_rate</i>) - λ : Parámetro regularizador que disminuye el peso de las observaciones aisladas (<i>lamda</i>) - γ : Parámetro regularizador que controla la profundidad de los árboles. (<i>gamma</i>)	- Control del <i>overfitting</i> de los árboles generados mediante alguna de las técnicas de CART - Disminución de la tasa de aprendizaje (η) - Aumento de alguno de los parámetros de regularización λ y γ	Medio	- Construcción de cadenas de árboles que reducen secuencialmente el error - Suele aplicarse con un elevado número de iteraciones - Inclusión de parámetros de regularización - Escisión de nodos a través de percentiles - Autoregulación mediante <i>early-stopping</i> - Capacidad de trabajar con <i>null values</i> - Tanto λ como γ influyen en la profundidad de los árboles generados - γ es equivalente al parámetro <i>min_impurity_decrease</i> de CART (influye en la escisión de nodos) - λ no tiene ningún parámetro equivalente en CART (influye en la asignación de valores a los nodos hoja) - XGBoost también incluye otros parámetros que evitan el sobreajuste a través de técnicas de bagging. Por ejemplo: - <i>subsample</i> : Proporción de individuos del conjunto de datos con los que se construye cada árbol - <i>colsample_bytree</i> : Proporción de variables que se utilizan para la construcción de cada árbol
LightGBM	Boosting	LightGBM: - LGBMRegressor - LGBMClassifier	- Parámetros para el control de los árboles generados (como CART): <i>num_leaves</i> y <i>max_depth</i> - Mismos que XGBoost	Equivalente a XGBoost	Medio-Bajo	- Misma base que XGBoost - Árboles que maximizan la ganancia nodo a nodo
CatBoost	Boosting	CatBoost: - CatBoostRegressor - CatBoostClassifier	Mismos que XGBoost	Equivalente a XGBoost	Medio	- Misma base que XGBoost - Codificación de las variables categóricas sin utilizar <i>one-hot-encoding</i> - Generación de árboles simétricos

Tabla 18: Resumen de los algoritmos: AdaBoost, Gradient Boosting, XGBoost, LightGBM y CatBoost

A. Detalles del desarrollo del trabajo

Todo el código empleado a lo largo del trabajo se encuentra en la carpeta de GitHub creada para tal fin (<https://github.com/GinesMeca/TFG>). El código se ha dividido en diferentes notebooks, organizados en función de los distintos apartados del trabajo. Dentro de ellos, se puede encontrar todo el código usado en los ejemplos prácticos, así como el referente a aquellas tablas y gráficos incluidos en la memoria.

Los datos utilizados, como ya se comenta en la introducción (sección 1.1), han sido extraídos de Kaggle. Para acceder a ellos, basta con seguir el siguiente link: <https://www.kaggle.com/jsphyg/weather-dataset-rattle-package>. Además, como también comentamos en el apartado dedicado a CatBoost (sección 3.2.5), ha sido necesario realizar un preprocesamiento de los datos para poder aplicar los diferentes algoritmos. Para ello, se ha seguido la estructura de la Figura 13, la cual se encuentra disponible en la web de Kedro y supone una práctica común en ingeniería de datos dado el interés en mantener una organización ETL (*Extract, Transform and Load*) de los datos con los que se trabaja.

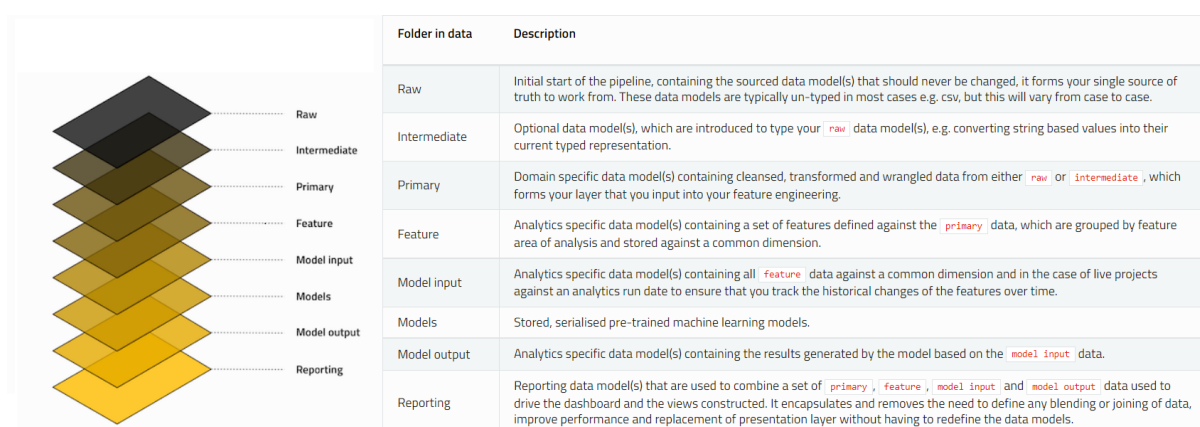


Figura 13: Estructura del procesamiento de los datos. Fuente: https://kedro.readthedocs.io/en/stable/12_faq/01_faq.html

En cuanto al tiempo invertido a cada parte del trabajo, éste se podría repartir, aproximadamente, como se muestra en la Tabla 19.

Además, para el desarrollo de ciertos apartados del trabajo ha sido necesario apoyarse en las asignaturas que se recogen en la Tabla 20.

Tarea	Tiempo (horas)
Recopilación de materiales	10
Estudio de bibliografía	30
Elaboración de resultados gráficos/numéricos	40
Redacción de la memoria	70
Total	150

Tabla 19: Tiempo aproximado de dedicación al trabajo

Asignatura	Páginas	Descripción
Series Temporales	7-8	El tipo de validación que se aplica se encuentra relacionada con la que se explicó en la presentación: <i>Model Assesment and Selection</i>
Análisis de Datos I	24-31	Se debe dominar el concepto de peso.
Análisis de Datos II	35-39	Modelo logit y odd-ratio.
Análisis de Datos I	General	Relación general con diferentes aspectos tratados.
Análisis de Datos II	General	Relación general con diferentes aspectos tratados.

Tabla 20: Asignaturas relacionadas con el trabajo

B. Desarrollos Gradient Boosting para clasificación

Se tiene que $f_0(x)$ viene dado por:

$$f_0(x) = \arg \min_{\gamma} \sum_{i=1}^N L(y_i, \gamma) = \arg \min_{\gamma} \sum_{i=1}^N (-y_i \gamma + \ln(1 + e^{\gamma}))$$

por tanto, para obtener la expresión de la predicción inicial dada en (4) se debe derivar la expresión y ver dónde alcanza el mínimo:

$$\begin{aligned} \frac{\partial}{\partial \gamma} \sum_{i=1}^N (-y_i \gamma + \ln(1 + e^{\gamma})) &= \sum_{i=1}^N \left(-y_i + \frac{1}{1 + e^{\gamma}} e^{\gamma} \right) = \sum_{i=1}^N (-y_i + p) = 0 \Leftrightarrow \\ \Leftrightarrow p &= \frac{\sum_{i=1}^N y_i}{N} \end{aligned}$$

El cálculo de los “pseudo-residuos” dado en (5) es el siguiente:

$$\begin{aligned}
 r_{im} &= - \left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)} \right]_{f=f_{m-1}} = - \left[\frac{\partial L(y_i, \ln(\text{odd}_i))}{\partial \ln(\text{odd}_i)} \right]_{\ln(\text{odd}_i)=f_{m-1}} = \\
 &= - \left[\frac{\partial(-y_i \ln(\text{odd}_i) + \ln(1 + e^{\ln(\text{odd}_i)}))}{\partial \ln(\text{odd}_i)} \right]_{\ln(\text{odd}_i)=f_{m-1}} = - \left[-y_i + \frac{e^{\ln(\text{odd}_i)}}{1 + e^{\ln(\text{odd}_i)}} \right]_{\ln(\text{odd}_i)=f_{m-1}} = \\
 &= y_i - \frac{e^{f_{m-1}(x_i)}}{1 + e^{f_{m-1}(x_i)}}
 \end{aligned}$$

Más tarde, una vez obtenida la expresión de γ , el valor de respuesta de cada nodo hoja:

$$\gamma = - \frac{\frac{d}{df_{m-1}(x_i)} L(y_i, f_{m-1}(x_i))}{\frac{d^2}{df_{m-1}(x_i)^2} L(y_i, f_{m-1}(x_i))}$$

se requiere calcular el valor de las derivadas involucradas. Como se tiene que $L(y_i, \ln(\text{odd}_i)) = -y_i \ln(\text{odd}_i) + \ln(1 + e^{\ln(\text{odd}_i)})$, las derivadas respecto de $\ln(\text{odd}_i)$ quedarán:

$$\frac{d}{d(\ln(\text{odd}_i))} L(y_i, \ln(\text{odd}_i)) = y_i - \frac{e^{\ln(\text{odd}_i)}}{1 + e^{\ln(\text{odd}_i)}}$$

y:

$$\begin{aligned}
 \frac{d^2}{d(\ln(\text{odd}_i))^2} L(y_i, \ln(\text{odd}_i)) &= \frac{d}{d(\ln(\text{odd}_i))} \left(- \frac{e^{\ln(\text{odd}_i)}}{1 + e^{\ln(\text{odd}_i)}} \right) = - \frac{e^{\ln(\text{odd}_i)}(1 + e^{\ln(\text{odd}_i)}) - e^{\ln(\text{odd}_i)}e^{\ln(\text{odd}_i)}}{(1 + e^{\ln(\text{odd}_i)})^2} = \\
 &= - \frac{e^{\ln(\text{odd}_i)}}{(1 + e^{\ln(\text{odd}_i)})^2}
 \end{aligned}$$

y así se obtiene la expresión recogida en (6). Además, esta expresión se puede modificar para obtener la fracción equivalente de la ecuación (7).

Por un lado, se observa que la expresión del numerador es equivalente a la dada para los “pseudo-residuos” r_{im} . Por tanto, el numerador se puede escribir como:

$$y_i - \frac{e^{f_{m-1}(x_i)}}{1 + e^{f_{m-1}(x_i)}} = y_i - p_{i,m-1}$$

Por otro lado, se puede transformar el denominador de manera que:

$$\frac{e^{f_{m-1}(x_i)}}{(1 + e^{f_{m-1}(x_i)})^2} = \frac{e^{f_{m-1}(x_i)}}{(1 + e^{f_{m-1}(x_i)})} \times \frac{1}{1 + e^{f_{m-1}(x_i)}} = p_{i,m-1}(1 - p_{i,m-1})$$

Así:

$$\gamma = \frac{y_i - p_{i,m-1}}{p_{i,m-1}(1 - p_{i,m-1})}$$

Referencias

- [1] Breiman, L. (1996). Bagging predictors. *Machine Learning* **24**, 123-140.
- [2] Breiman, L. (2001). Random Forests. *Machine Learning* **45**, 5-32.
- [3] Breiman, L., Friedman, J.H, Olshen, R.A. and Stone, C.J. (1984). *Classification And Regression Trees*. Routledge.
- [4] Chen, T. and Guestrin, C. (2016). XGBoost: A Scalable Tree Boosting System. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* **1**, 785-794.
- [5] Chen, W., Finley, T., Liu, T.Y., Ke, G., Ma, W., Meng, Q., Wang, T. and Ye, Q. (2017). LightGBM: A Highly Efficient Gradient Boosting Decision Tree. *Proceedings of the 31st Conference on Neural Information Processing Systems*
- [6] Dorogush, A.V., Gulin, A., Gusev, G., Prokhorenkova, L. and Vorobev, A. (2018). CatBoost: unbiased boosting with categorical features. *Proceedings of the 32nd Conference on Neural Information Processing Systems*
- [7] Freund, Y. and Schapire, R.E. (1997). A Decision-Theoretic Generalization of On-Line Learning and a Application to Boosting. *Journal of Computer and System Sciences* **55**, 119-139.
- [8] Friedman, J. (2001). Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, **29(5)**, 1189-1232.
- [9] Friedman, J., Hastie, T. and Tibshirani, R. (2000). Additive logistic regression: a statistical view of boosting. *Annals of Statistics*, **28(2)**, 337-407.
- [10] Jin, Y., Tree Boosting With XGBoost — Why Does XGBoost Win “Every” Machine Learning Competition?. <https://medium.com/syncedreview/tree-boosting-with-xgboost-why-does-xgboost-win-every-machine-learning-competition> (Consultado el 11 de Mayo de 2022).
- [11] Mandot, P., What is LightGBM, How to implement it? How to fine tune the parameters?. <https://medium.com/@pushkarmandot/https-medium-com-pushkarmandot-what-is-lightgbm-how-to-implement-it-how-to-fine-tune-the-parameters> (Consultado el 12 de Mayo de 2022).

- [12] Morgan, J.N., Sonquist, J.A. (1963). Problems in the Analysis of Survey Data, and a proposal. *Journal of the American Statistical Association* **58**, 415-434.
- [13] Segura, T., “CatBoost Algorithm” explained in 200 words. <https://thaddeus-segura.com/catboost/> (Consultado el 12 de Mayo de 2022).