

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 podamos 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. No obstante, dado que esta sección ya se ha tratado en varios trabajos de años anteriores, se ha decidido no profundizar en exceso para poder invertir un mayor tiempo en aquello que resulta novedoso.

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, ya que estos algoritmos también han sido tratados en cursos anteriores. 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 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.

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. Nevertheless, since this topic has been explained in previous researches we have not focused on it, in order to spend more time on the new algorithms 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, since these algorithms have been treated in former works too. 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.

Key words: CART, Bagging, Boosting, Machine Learning

Índice

1. Introducción	6
1.1. Datos: Llueve en Australia	7
1.2. Metodología	7
2. Árboles de decisión CART	9
2.1. Preliminares	9
2.1.1. Notación y conceptos básicos	10
2.1.2. Objetos de estudio de un CART	11
2.2. Valor o clase asignada a cada nodo hoja	11
2.3. Elección de variables y valores asociados a cada nodo interno	13
2.4. Criterio de parada de escisión de nodos	14
2.5. Ejemplo	15
2.6. Ventajas y desventajas	16
3. Métodos derivados de los árboles de decisión	18
3.1. Algoritmos de bagging	18
3.1.1. Bagging	18
3.1.2. Random Forest	19
3.1.3. Extra-Trees	19
3.1.4. Error de Out-Of-Bag	19
3.1.5. Ejemplo	20
3.1.6. Ventajas y desventajas	22
3.2. Algoritmos de boosting	22
3.2.1. AdaBoost	23
3.2.2. Gradient Boosting	31
3.2.3. XGBoost	38
3.2.4. LightGBM	45
3.2.5. CatBoost	46
3.2.6. Ejemplo	50

3.2.7. Otros parámetros	51
3.2.8. Ventajas y desventajas	52
4. Comparación de modelos	54
5. Conclusión	56
A. Detalles del desarrollo del trabajo	57
Referencias	58

1. Introducción

Vivimos en la 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 nos encontramos con dispositivos, como los teléfonos móviles, capaces de escribir texto, recoger audio y realizar fotografías y vídeos. Así, la información a nuestra alcance es infinita. Podemos 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, podemos acceder al historial de imágenes de la cámara de seguridad de nuestra vivienda o a la tabla que recoge las últimas operaciones de nuestra empresa. Es fácil obtener grandes cantidades de datos de aquello que nos interesa.

Dado que el número de datos que manipulamos 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 por James N. Morgan y John A. Sonquist [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 estudiaremos en este trabajo, es el algoritmo CART (Classification And Regression Trees), creado por Breiman, Friedman, Olshen y Stone en 1984 [13]. De hecho, fue el propio Leo Breiman quien, haciendo uso de CART, desarrolló años más tarde algoritmos como Bagging [1] 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 nos encontramos algoritmos más complejos, como XGBoost, que ocuparán gran parte del trabajo.

Dado que en los últimos años varios alumnos han realizado su TFG sobre árboles de decisión explicando su estructura más básica, resumiré las secciones dedicadas a ello con el fin de centrarme en analizar a fondo los algoritmos más complejos y novedosos que no se han tratado en estos trabajos y, así, lograr elaborar un mapa conceptual de modelos lo más completo posible.

1.1. Datos: Llueve en Australia

Para hacernos una idea de cómo funciona cada uno de los algoritmos explicados a lo largo del trabajo, haremos uso del conjunto de datos “Rain in Australia”, de Kaggle. Se trata de un dataset en el que podemos encontrar mediciones de diferentes 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. Así, el objetivo será averiguar si con los datos del día actual se puede predecir si lloverá el día siguiente utilizando los diferentes algoritmos descritos.

	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

Figura 1: Encabezado del dataset

En resumen, se tienen 23 columnas que recogen información de un total de 145.460 días. En los diferentes ejemplos, se tratará de realizar predicciones de la variable “Rain Tomorrow”.

1.2. Metodología

A la hora de aplicar los diferentes métodos al conjunto de datos se procederá mediante validación cruzada. Es decir:

1. Se selecciona el modelo correspondiente.

2. Se separa, al azar, el conjunto de datos inicial en dos: datos de entrenamiento y datos de testeo.
3. Se entrena el modelo con los datos de entrenamiento.
4. Se aplica el modelo entrenado a los datos de testeo.
5. Se comprueba 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 diarios a lo largo de 10 años, a la hora de realizar predicciones sobre las posibles variables de interés nos interesa conocer la precisión de nuestro 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 sabríamos 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, el procedimiento será ligeramente distinto. Emplearemos una validación “*Out Of Time*”, basada en separar el conjunto de datos siguiendo un orden cronológico. Dado que nuestro dataset contiene fechas desde el 01/11/2007 hasta el 24/06/2017, nuestro 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, entrenamos el modelo con datos anteriores a los de testeo para saber cómo van a ser las predicciones futuras y asemejarnos más a lo que nos interesaría en un caso real: entrenar el modelo en función de los datos que poseemos para predecir fechas futuras.

Además, los cálculos se realizarán en el lenguaje de programación Python. Todo el código se encuentra disponible en el Anexo (A), publicado en GitHub.

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. Por ejemplo:

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 siguiente:*

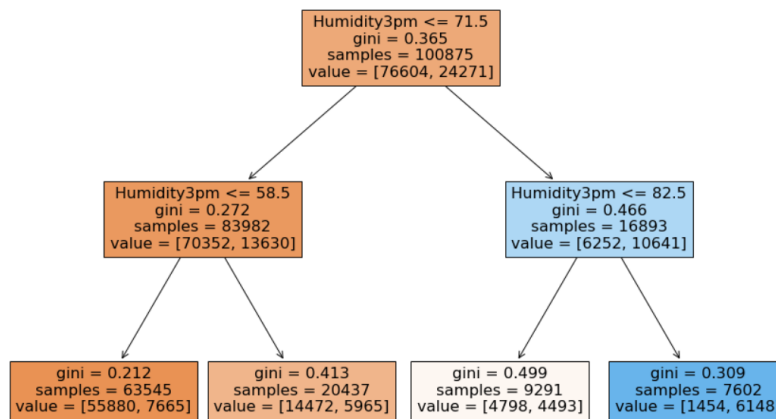


Figura 2: Árbol de decisión del Ejemplo 2.1

Como se puede observar, el algoritmo es muy intuitivo: se introduce un individuo y se le aplica 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 par-

ticiones 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.

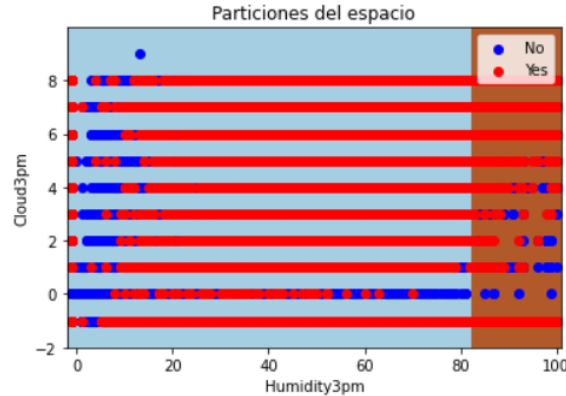


Figura 3: Partición del espacio del Ejemplo 2.1

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 2 del ejemplo 2.1 . Se tiene un nodo raíz, seis nodos internos y 8 nodos hoja.

Por otro lado, conviene hablar del concepto de sobreajuste (en inglés: *overfitting*). 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.

Para lidiar con el sobreajuste, interesa manejar otras dos nociones: el sesgo y la varianza. El sesgo (en inglés: bias) hace referencia a los errores cometidos en las predicciones de los individuos de nuestro conjunto de entrenamiento. Por contra, la varianza es la medida del error cometido en los individuos que no pertenecen a este conjunto; en nuestro caso, lo mediremos en los individuos del conjunto de testeo. No obstante, estos conceptos plantean un problema: interesa tener valores bajos de ambos indicadores, pero disminuir el sesgo conlleva un aumento de la varianza. Así, a la hora de construir un modelo de Machine Learning, interesa estudiar que el sesgo no sea excesivamente bajo, para evitar el sobreajuste, pero también que no sea excesivamente alto para que las predicciones sean fiables.

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 ($n, p \in \mathbb{N}$). De este modo, sean X_1, X_2, \dots, X_p las variables explicativas e y la variable dependiente. En el caso de nuestro conjunto de datos de ejemplo:

Como se comentaba anteriormente, el resultado de un árbol de decisión es una partición del espacio. Supongamos que se obtienen M regiones diferentes, denotadas por R_m con $m = 1, \dots, M$. Por tanto, definamos como Y al árbol de decisión y como c_m , con $m = 1, \dots, M$, a las predicciones (imágenes de Y) asociadas a cada región R_m . De este modo:

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

	X_1	X_2	\dots	X_{22}	y
x_1	2008-12-01	Albury	\dots	No	No
x_2	2008-12-02	Albury	\dots	No	No
x_3	2008-12-02	Albury	\dots	No	No
\vdots	\vdots	\vdots	\ddots	\vdots	\vdots

Tabla 1: Datos “*Rain in Australia*”

donde D es el dominio de la función (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 tratemos (regresión o clasificación). Por tanto, lo analizaremos por separado.

A la hora de entrenar un modelo de regresión, se busca minimizar los errores entre y e Y . Tomando el error mínimo cuadrático, llegamos al problema:

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

donde $Y_i = Y(x_i) \forall i = 1, \dots, n$. Si denotamos $f(c_m) = \sum_{x_i \in R_m} (y_i - c_m)^2$, observamos que alcanzar la solución óptima del problema planteado es equivalente a minimizar la función f . Derivando, obtenemos un candidato a óptimo:

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

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.

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

Pese a las similitudes existentes entre los árboles de regresión y los de clasificación, conviene dividir este apartado en dos con tal de mostrar el funcionamiento exacto de cada uno de ellos.

Árboles de regresión

Se busca conocer cómo realizar las divisiones. En particular, nos centraremos en la primera división y el proceso será similar para las siguientes. Al situarnos en el nodo raíz, nos encontramos con el conjunto de datos al completo y pretendemos dividirlo en dos regiones, que denotaremos R_1 y R_2 , en función de una de las variables explicativas, X_j , y de un valor respecto de esta, 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 e c_2 constante sobre R_2 . Ya hemos visto que estos c_m se corresponderán con la media de los individuos del nodo hoja:

$$c_m = \frac{1}{n_m} \sum_{x_i \in R_m} y_i$$

donde n_m es el número de individuos que pertenecen a dicho nodo. De esta manera, busquemos 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, busquemos 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.

Árboles de clasificación

En el caso de los árboles de clasificación, la variable dependiente es una clase. Es decir, nos encontramos en la situación recogida en la Tabla 1. Ahora, en lugar de calcular el

RSS, introduciremos una función de impureza que nos permita evaluar la homogeneidad de los nodos; o lo que es lo mismo, que nos 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”.

Sabiendo esto, solo queda evaluar el índice Gini asociado a la división de un nodo. Esto se hace mediante 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. No obstante, como ya se ha introducido, 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. Es evidente que podando el árbol se introduce sesgo, pues no dejamos 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, probaremos con diferentes profundidades máximas permitidas para escoger el modelo más fiable.

Con el fin de analizar los resultados y llevar a cabo esta elección, utilizaremos 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 $[-1, 1]$, se realiza la transformación $2 \times AUC - 1$ para obtener una nueva métrica en el intervalo $[0, 1]$. Esta nueva métrica recibe el nombre de coeficiente Gini (no confundir con el índice Gini de escisión de nodos) y permite dar un porcentaje de efectividad del modelo.

Con esto, aplicando los diferentes árboles de clasificación, llegamos al siguiente resultado (redondeado):

	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

Figura 4: Resultados árboles de clasificación para RainTomorrow

En la tabla se ven reflejados el tiempo de ejecución de cada modelo, 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 podemos 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. Es decir, el bias se reduce pero la varianza se dispara. 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%.

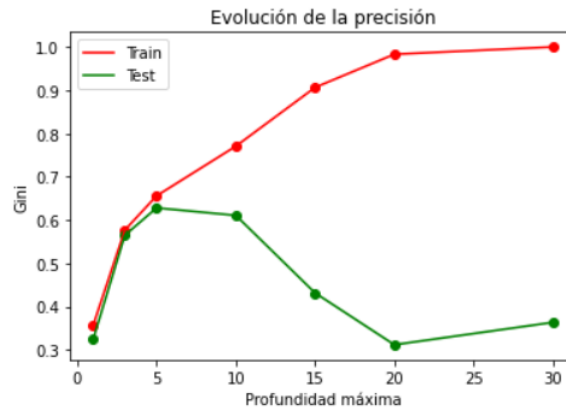


Figura 5: Evolución de los coeficientes Gini en ambos conjuntos

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

Por último, notemos que el tiempo empleado en la generación de los árboles presentados es mínimo. Los modelos escogidos (DT₃ y DT₅) apenas tardan medio segundo en construirse, resultado que es prueba de la eficiencia de esta clase de modelos.

2.6. Ventajas y desventajas

Como ya avanzábamos, y como hemos podido observar a través del ejemplo, los árboles de decisión son una herramienta muy potente para el análisis de datos. No obstante, pese a presentar muchos puntos a favor, también hay una serie de inconvenientes a tener en cuenta. En esta subsección, nos encargaremos de analizarlos.

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 la precisión del modelo 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.

3. Métodos derivados de los árboles de decisión

Los árboles de decisión son una gran herramienta predictiva. 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 hemos visto 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 combinación 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. De esta manera, generan tantos árboles, independientes entre sí, como muestras hayan tomado. Así, la respuesta de un algoritmo de bagging en un problema de regresión será el promedio de las respuestas de cada uno de los árboles, y en un problema de clasificación será la clase que más veces se ha repetido.

Analizaremos 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 o mediante el conteo de clases.

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, conseguimos 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 y, a la hora de realizar las escisiones de nodos en cada árbol, asigna un valor aleatorio a cada variable. Así, para realizar una partición del espacio no se debe probar con todos los valores posibles para cada una de las p variables, se probará con m valores generados aleatoriamente 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 hemos 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*. Es decir, si tenemos 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, como podemos observar, cuando el número de individuos de nuestra muestra

sea elevado (supongamos $n \rightarrow +\infty$):

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

Es decir, sabemos que, si contamos 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).

Gracias al *Out-Of-Bag* se pueden desarrollar conceptos interesantes. Uno de estos conceptos es el error OOB, que se basa en la idea de la validación cruzada para medir la fiabilidad del modelo. En lugar de separar previamente el conjunto de datos en conjunto de entrenamiento y conjunto de testeo, para cada árbol, los individuos que se utilizarán para el test 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, pese a que el error OOB es muy cómodo, dado que no hace falta separar el conjunto de datos en dos, nosotros seguiremos evaluando la precisión del modelo mediante validación cruzada. El motivo es el que ya comentamos en la introducción 1.2 nos interesa aplicar una validación “*Out Of Time*” y el error OOB no respeta esta validación pues, evidentemente, 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. Para ello, usaremos el conjunto de datos que ya hemos utilizado con anterioridad y aplicaremos los diferentes algoritmos de bagging vistos en la sección con el fin de compararlos y comentar sus mejoras. Además, con tal de evitar sobreajustes, estableceremos el criterio de parada de los árboles generados en 5 (como se ve en el ejemplo de CART, es el más fiable junto a 3) y probaremos con diferentes números de árboles generados por cada algoritmo.

Al igual que en el ejemplo de CART, calcularemos el coeficiente Gini haciendo uso del AUC para elegir los modelos más precisos y realizar correctamente las comparaciones.

Aplicando los diferentes algoritmos llegamos 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

	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

	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

Figura 6: Resultados algoritmos de bagging para RainTomorrow

donde “Bag” se corresponde con Bagging, “RT” con Random Forest y “ET” con Extra-Trees. Como se puede apreciar, la mejora respecto a CART es evidente: en el ejemplo anterior llegamos a la conclusión de que una de las mejores opciones era elegir el árbol con profundidad máxima 5 que nos 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, obtenemos 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. Esto se debe a la convergencia del error OOB:

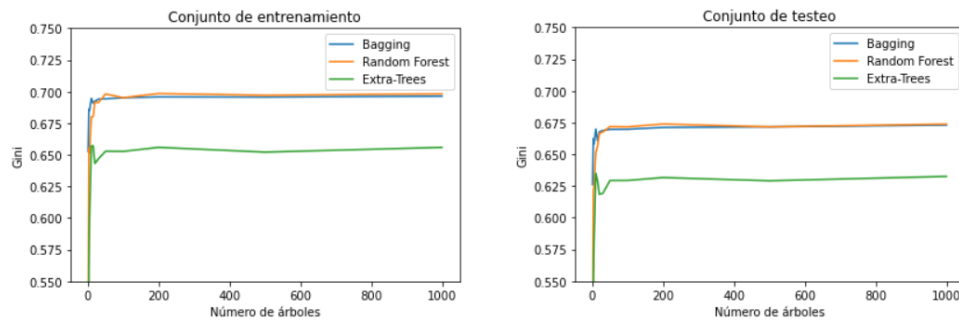


Figura 7: Convergencia coeficientes Gini

Por último, notemos que los algoritmos de bagging son más costosos que CART en cuanto al tiempo empleado para su construcción. Además, podemos 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

Al igual que se ha hecho con CART, a continuación se realizará un análisis de pros y contras de los algoritmos de bagging.

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).

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 (*Adaptive Boosting*) fue publicado en 1997 por Freund y Schapire [7]. Se trata de 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 les llamaremos “tocones”. En la práctica, AdaBoost construye un primer tocón a partir de todos los datos disponibles, cuyo resultado consideraremos 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 un 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, formalizaremos este proceso. El algoritmo es el siguiente:

Algorithm 10.1 *AdaBoost.M1.*

1. Initialize the observation weights $w_i = 1/N$, $i = 1, 2, \dots, N$.
 2. For $m = 1$ to M :
 - (a) Fit a classifier $G_m(x)$ to the training data using weights w_i .
 - (b) Compute

$$\text{err}_m = \frac{\sum_{i=1}^N w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^N w_i}.$$
 - (c) Compute $\alpha_m = \log((1 - \text{err}_m)/\text{err}_m)$.
 - (d) Set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))]$, $i = 1, 2, \dots, N$.
 3. Output $G(x) = \text{sign} \left[\sum_{m=1}^M \alpha_m G_m(x) \right]$.
-

Figura 8: Algoritmo de AdaBoost. Fuente: [14]

Partimos de una matriz de datos X compuesta por n individuos y el valor que toman para p variables. Además, disponemos de la variables dependiente, que denotaremos por

y. Como ya avanzábamos, AdaBoost construye los árboles secuencialmente modificando los pesos de los individuos en cada iteración; por tanto, también habrá que tener en cuenta la matriz de pesos. Es decir:

$$X = \begin{pmatrix} x_{11} & x_{12} & \dots & x_{1p} \\ x_{21} & x_{22} & \dots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \dots & x_{np} \end{pmatrix} \wedge y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} \wedge P = \begin{pmatrix} w_1 & 0 & \dots & 0 \\ 0 & w_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & w_n \end{pmatrix}$$

Como podemos ver en la Figura 8, el primer paso consiste en asignar el mismo peso a todos los individuos; ya que no se ha realizado ningún árbol previo y, por tanto, nos interesa calcular buenas predicciones del máximo número de individuos posibles. Así, en primer lugar:

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

A continuación, siguiendo el esquema de la Figura 8, se crea un árbol de decisión de dos nodos hoja, un tocón. De esta manera, cada tocón construido (denotados por G_m) realiza una única división del espacio, obteniendo dos subconjuntos. Además, es evidente que en cada tocón únicamente se verá involucrada una variable.

Una vez construido el tocón, definimos:

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

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 nos encontramos con que el denominador puede ser distinto de 1. No obstante, nosotros siempre estandarizaremos los pesos antes de empezar de nuevo con el bucle, por lo que podemos 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, siguiendo el apartado 2b de la Figura 8, definimos:

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

Como podemos 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, podemos 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, por tanto, conviene que su influencia en el resultado final no sea muy elevada.

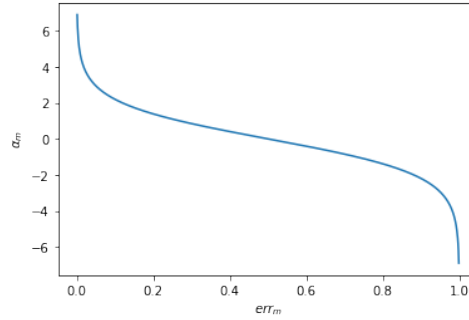


Figura 9: Gráfico de α_m

Para cerrar la segunda parte del algoritmo, nos queda analizar la expresión:

$$(d) \text{ Set } w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))], i = 1, 2, \dots, n$$

que se encarga de modificar los pesos para el paso siguiente. Definimos:

$$P_m = \begin{pmatrix} w_1^m & 0 & \dots & 0 \\ 0 & w_2^m & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & w_n^m \end{pmatrix}$$

donde w_i^m es el peso del individuo i en la iteración m -ésima. Así, la expresión (d) viene a decir:

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

Notemos 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. Así, conforme α_m se va acercando

a 0, la importancia del tocón en el modelo y el aumento del peso de los individuos mal clasificados disminuyen. 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 %), $\alpha_m \ll 0$, podríamos interpretar que el modelo está trabajando a la inversa y que hay que considerar las clasificaciones contrarias que éste nos da. 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 nosotros consideraremos como erróneos dada la eficacia del modelo.

Para finalizar con el bucle, queda hacer especial hincapié en un par de conceptos. Si $m = 1$, el proceso explicado no genera ninguna complicación: todos los individuos tienen el mismo peso, se genera un árbol de solo dos nodos hoja, se realiza el cálculo de err_1 y α_1 y se calcula P_2 (matriz de pesos a utilizar en $m=2$). No obstante, para $m > 1$, $P_m \neq I$; es decir, los individuos tendrán pesos diferentes. Esto se tiene que ver reflejado a la hora de generar $G_m(x)$. El índice Gini de una región, involucrado en el criterio de escisión de nodos (2.3), es el siguiente:

$$G_{region} = \sum_{k=1}^K p_{rk}(1 - p_{rk}) = 1 - \sum_{k=1}^K p_{rk}^2$$

donde p_{rk} era la proporción de individuos del conjunto de entrenamiento de clase k que se encuentran en la región r y K era el número total de clases (en la expresión original de la sección 2.3, r viene denotado por m , notación que hemos cambiado aquí para evitar confusiones). Como podemos observar, esta definición del índice Gini no hace referencia a los pesos de los individuos en ningún momento, por lo que tendremos que encontrar una manera de conseguir que estos pesos sean considerados.

La solución más fácil, y la que se encuentra implementada en la librería *Scikit-learn* de Python, es la siguiente: estandarizamos los pesos, es decir, dividimos el peso de cada individuo por la suma total de pesos, y consideramos el resultado como las frecuencias relativas de cada individuo. Así, podemos conseguir también las frecuencias absolutas de la muestra:

El siguiente paso es generar n números aleatorios (t_1, t_2, \dots, t_n) pertenecientes al intervalo $[0, 1]$. Entonces, se cumple que:

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

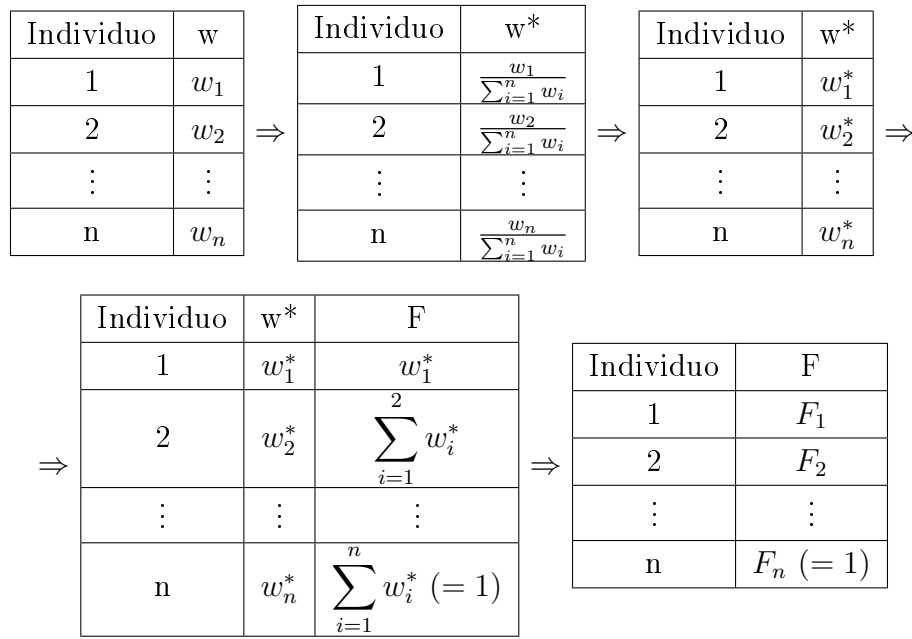


Tabla 2: Reasignación de pesos

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 añadimos x_j a nuestro nuevo conjunto. De esta manera, obtendremos 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, asignamos un peso de $1/n$ a cada uno de los individuos del nuevo conjunto de datos y ya podemos proceder a la creación del siguiente tocón.

Ejemplo funcionamiento del bucle:

Supongamos que nuestro conjunto de datos se redujese a:

	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

Figura 10: Datos para el ejemplo teórico de AdaBoost

Por tanto, creamos un árbol de decisión de una única división y obtenemos:

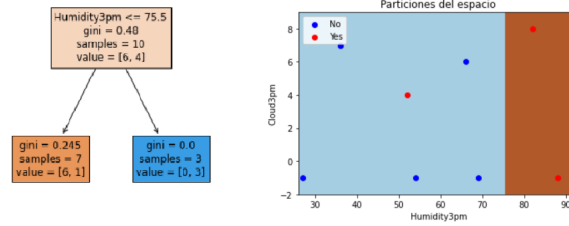


Figura 11: Primer tocón del modelo

Como podemos observar, solo hay un individuo mal clasificado, el número 9. A continuación, calculamos err_1 :

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

y, haciendo uso de él, obtenemos 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, podemos realizar la reasignación de pesos:

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

donde los superíndices hacen referencia a la iteración en la que nos encontramos. Así, modificamos los pesos y aplicando el proceso recogido en la Tabla 2:

Individuo	w		Individuo	w*		Individuo	w*	
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	\Rightarrow
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 3: 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:

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 4: 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 4. Como podemos 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, nos queda analizar cómo realiza las predicciones AdaBoost. Como podemos ver en el pseudocódigo, la solución de la función es:

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

Entonces, dado que el algoritmo AdaBoost publicado en 1997 estaba diseñado 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 avanzábamos antes, 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:

Por último, comentar que este algoritmo es conocido como “Discrete AdaBoost”. Como podemos ver, es un algoritmo de respuesta simple, dado que solo sirve para clasificación binaria. No obstante, 3 años más tarde se expuso el algoritmo “Real Adaboost” [8], que extendía el funcionamiento del algoritmo visto al caso continuo y a más de dos grupos.

En el caso de tener más de dos grupos, la única modificación a realizar se encuentra 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, la solución no es tan inmediata. Resumiendo, habrá que modificar la manera en la que se miden los errores dado que ahora la variable dependiente no es binaria, sino que es una variable continua. La solución que se ofrece pasa por definir una “función de pérdida” o, en inglés, “*Loss Function*” (a partir de ahora, nos referiremos a ella en su término inglés dado que es como se conoce realmente). Esta función nos devolverá un valor entre 0 y 1, al igual que lo hacían anteriormente las funciones err_m o α_m , y su objetivo es dar una medida del error. Vendrá dada de la siguiente forma:

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

donde $y^{(m)}(x_i)$ hace referencia al valor devuelto 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.

De esta manera, se definen diferentes candidatas a loss function. Destacan:

$$L_i^{(m)} = \frac{|y^{(m)}(x_i) - y_i|}{D} \quad (lineal) \quad \Bigg| \quad L_i^{(m)} = \frac{|y^{(m)}(x_i) - y_i|^2}{D^2} \quad (cuadrada)$$

$$L_i^{(m)} = 1 - \exp \left[\frac{-|y^{(m)}(x_i) - y_i|}{D} \right] \quad (exponencial)$$

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, ya podemos definir 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

Siguiendo la línea de AdaBoost, en 2001 se publica un nuevo algoritmo: Gradient Boosting [9]. De nuevo, se trata de un método 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. Así, 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 va reduciendo iteración a iteración.

Además, Gradient Boosting requiere de una Loss Function a través de la cual obtiene los residuos y el valor asociado a cada nodo hoja de los árboles predictores de residuos. Como ya se introdujo en AdaBoost (3.2.1), el objetivo de la Loss Function es evaluar el error cometido entre la predicción realizada y el valor observado. Por ello, interesa minimizarla para así minimizar el error cometido.

Algorithm 10.3 Gradient Tree Boosting Algorithm.

1. Initialize $f_0(x) = \arg \min_{\gamma} \sum_{i=1}^N L(y_i, \gamma)$.
 2. For $m = 1$ to M :
 - (a) For $i = 1, 2, \dots, N$ compute

$$r_{im} = - \left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)} \right]_{f=f_{m-1}}.$$
 - (b) Fit a regression tree to the targets r_{im} giving terminal regions R_{jm} , $j = 1, 2, \dots, J_m$.
 - (c) For $j = 1, 2, \dots, J_m$ compute

$$\gamma_{jm} = \arg \min_{\gamma} \sum_{x_i \in R_{jm}} L(y_i, f_{m-1}(x_i) + \gamma).$$
 - (d) Update $f_m(x) = f_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$.
 3. Output $\hat{f}(x) = f_M(x)$.
-

Figura 12: Algoritmo de Gradient Boosting. Fuente: [14]

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, analizaremos estos dos casos por separado.

Regresión

En primer lugar, para iniciar el algoritmo se debe contar con una matriz de datos X (matriz de variables explicativas), un vector y (vector de la variable dependiente a predecir) y 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 hemos 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 $f(x)$:

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

la expresión resultante no es más que el residuo de la predicción.

Una vez establecida la Loss Function a emplear, veamos cómo funciona el algoritmo al sustituirla. Como podemos observar en la Figura 12, el primer paso consiste en determinar un valor fijo que representará la “primera predicción”; es decir, aquella que luego modificaremos en función de los residuos:

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

que sustituyendo por la función que hemos fijado 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 vimos en la sección 2.2 que el valor que minimiza este valor es la media, por tanto:

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

En el paso 2, se construye un bucle. Cada iteración de este bucle 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 que ya hemos especificado anteriormente:

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

Los dos apartados siguientes hacen referencia a los nodos hoja del nuevo árbol generado. Este árbol tendrá como objetivo realizar una predicción de los r_{im} en función de los x_i . Es decir, lo que se estará prediciendo ahora serán los residuos. 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 γ_{jm} . 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:

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

es decir, se le añade el residuo predicho a la última predicción calculada.

Por último, el algoritmo devuelve como respuesta la respuesta de $f_M(x)$.

Antes de pasar al caso de clasificación, se hará especial hincapié en un par de observaciones. Por un lado, el algoritmo establece una “predicción inicial” que se corresponde con la media de los y'_i s y, actualiza esta predicción iteración a iteración haciendo uso de los residuos. No obstante, este proceso implica que los modelos se sobreajustarían con facilidad en un número bajo de iteraciones. Así, para evitar el sobreajuste, se establece una tasa de aprendizaje que no aparece en el algoritmo descrito y que modifica la definición de las f'_m s de la siguiente manera:

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

donde $\nu \in (0, 1)$ es esta tasa de aprendizaje comentada y hace que las modificaciones que se van realizando sean menores. Por ello, Gradient Boosting se suele aplicar con un M grande, 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 otro lado, concluiremos recordando que hemos 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 tendremos que calcular los valores que minimizan las nuevas expresiones. Además, puntualizar que en este caso no sería del todo correcto definir los r_{im} como residuos, ya que la forma de estos puede ser, ahora, muy diferente (ya no tiene por qué ser “valor observado - valor predicho”). Es por eso que en Gradient Boosting se denomina a estos valores como “pseudo-residuos” (*pseudo-residuals* en inglés).

Clasificación:

Como avanzábamos, el caso de clasificación volverá a basarse en el algoritmo de la Figura 12; pero hará uso de una Loss Function diferente a la de regresión.

El procedimiento descrito está pensado para clasificación binaria. Así, igual que en regresión la estimación que se da es la que corresponde a 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 propuesta (“1”) y si la probabilidad es menor que 0.5 se clasificará en la clase opuesta (“0”). No obstante, Gradient Boosting no aproxima esta probabilidad directamente, si no que aproxima el log-odd-ratio y, posteriormente, obtiene la probabilidad deseada:

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

Veamos en primer lugar cuál es la Loss Function escogida. Notemos que en el caso de clasificación binaria nuestro 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, notemos que cuando utilizamos esta función en el método de Máxima Verosimilitud, estamos interesados en maximizar su valor. Por el contrario, cuando trabajamos con una Loss Function, buscamos minimizarla, pues el modelo será más ajustado conforme menor sea el valor de esta función. Por tanto, si deseamos emplear la función de soporte como Loss Function, tendremos que multiplicarla por -1. También, como deseamos aplicar esta función individuo a individuo, podemos eliminar el sumatorio. Así, la Loss Function que emplearemos 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, como hemos avanzado antes, deseamos trabajar con log-odd-ratio. Entonces, transformamos esta expresión de manera que las p_i pasen a ser log-odd-ratio (a partir de ahora, por simplificar, $\ln(odd_i)$):

$$\begin{aligned} L(y_i, p_i) &= -(y_i \ln(p_i) + (1 - y_i) \ln(1 - p_i)) = -y_i \ln(p_i) - \ln(1 - p_i) + y_i \ln(1 - p_i) = \\ &= -y_i (\ln(p_i) - \ln(1 - p_i)) - \ln(1 - p_i) = -y_i \ln\left(\frac{p_i}{1 - p_i}\right) - \ln(1 - p_i) = \\ &= -y_i \ln(odd_i) - \ln(1 - p_i) \end{aligned}$$

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

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

Por tanto:

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

y ya tenemos la Loss Function en función de $\ln(odd_i)$:

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

Así pues, una vez establecida la Loss Function, podemos abordar el primer paso del algoritmo. Veamos cuál es ahora el valor de la predicción inicial. Recordemos 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 debemos derivar la expresión y ver dónde alcanza el mínimo:

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

Podemos 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 queremos trabajar en términos de $\ln(\text{odd}_i)$, debemos escribir esta probabilidad en la forma correspondiente. De esta manera:

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

Ahora, a través de predicciones secuenciales de los “pseudo-residuos”, debemos afinar esta predicción inicial. De acuerdo con lo anterior, el siguiente paso consiste en el bucle que nos calculará en cada iteración (supongamos 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” los calcularemos al igual que en el apartado anterior teniendo en cuenta que la Loss Function escogida ahora es diferente. Se tiene:

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

Notemos 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 denotamos 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:

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

Ahora, para obtener el γ que minimiza dicha expresión, aproximaremos $L(x_i, f_{m-1}(x_i) + \gamma)$ mediante su desarrollo de Taylor de segundo orden. Lo haremos para un x_i cualquiera con el fin de no complicar los cálculos con los sumatorios. Es decir, consideramos:

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

y derivamos respecto de γ e igualamos a cero puesto que es lo que buscamos minimizar:

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

Por tanto, como tenemos 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:

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

En resumen:

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

Notemos que la expresión del numerador es equivalente a la que hemos dado 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, podemos transformar el denominador de manera que:

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

Es decir, la expresión del γ que minimiza el sumatorio anterior queda:

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

Para finalizar con los cálculos, recordemos que hemos decidido omitir los sumatorios en el desarrollo realizado. Ahora bien, como la derivada de un sumatorio es igual al sumatorio de las derivadas, podemos 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} tal y como se puede ver en el algoritmo (Figura 12) y la predicción final será la predicción M -ésima, es decir, la correspondiente a la última iteración del bucle. Además, en el caso de clasificación también se incorporará una tasa de aprendizaje a la hora de calcular los f_m 's.

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'*) es un algoritmo que utiliza de base lo explicado en la última sección y, además, añade una serie de mejoras que lo convierten en una herramienta muy potente. Fue publicado en 2014 [3] y su repercusión fue inmediata. 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.

Como ya avanzábamos, XGBoost 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. Estos parámetros son ciertos valores que se incluyen al modelo para tratar de evitar el sobreajuste. Además, XGBoost fue diseñado para trabajar con conjuntos de datos muy grandes, por lo que estos parámetros de regularización nos ayudarán, sobretodo, en caso de que el conjunto de datos no sea lo suficientemente grande.

El algoritmo XGBoost es el siguiente:

Algorithm 3: Newton tree boosting

Input : Data set \mathcal{D} .
 A loss function L .
 The number of iterations M .
 The learning rate η .
 The number of terminal nodes T_n

- 1 Initialize $\hat{f}^{(0)}(x) = \hat{f}_0(x) = \hat{\theta}_0 = \arg \min_{\theta} \sum_{i=1}^n L(y_i, \theta)$;
- 2 **for** $m = 1, 2, \dots, M$ **do**
- 3 $\hat{g}_m(x_i) = \left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)} \right]_{f(x)=\hat{f}^{(m-1)}(x)}$;
- 4 $\hat{h}_m(x_i) = \left[\frac{\partial^2 L(y_i, f(x_i))}{\partial f(x_i)^2} \right]_{f(x)=\hat{f}^{(m-1)}(x)}$;
- 5 Determine the structure $\{\hat{R}_{jm}\}_{j=1}^{T_n}$ by selecting splits which maximize
 $Gain = \frac{1}{2} \left[\frac{G_L^2}{H_L} + \frac{G_R^2}{H_R} - \frac{G_{jm}^2}{H_{jm}} \right]$;
- 6 Determine the leaf weights $\{\hat{w}_{jm}\}_{j=1}^{T_n}$ for the learnt structure by
 $\hat{w}_{jm} = -\frac{G_{jm}}{H_{jm}}$;
- 7 $\hat{f}_m(x) = \eta \sum_{j=1}^{T_n} \hat{w}_{jm} \mathbf{I}(x \in \hat{R}_{jm})$;
- 8 $\hat{f}^{(m)}(x) = \hat{f}^{(m-1)}(x) + \hat{f}_m(x)$;
- 9 **end**

Output: $\hat{f}(x) \equiv \hat{f}^{(M)}(x) = \sum_{m=0}^M \hat{f}_m(x)$

Figura 13: Algoritmo de XGBoost. Fuente: [17]

Podemos observar que el número de inputs es elevado. No obstante, ninguno es nuevo para nosotros. El algoritmo requiere de un conjunto de datos de entrenamiento, una Loss Function, el número de iteraciones M (equivalente a la M del bucle de Gradient Boosting), una tasa de aprendizaje y el número máximo de nodos hoja que le vamos a permitir a cada árbol.

A continuación, al igual que realizamos en Gradient Boosting, explicaremos el caso de regresión y el de clasificación por separado.

Regresión

En el caso de regresión, la Loss Function elegida será la misma que escogimos en Gradient Boosting:

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

El primer paso consistirá en definir la predicción inicial, que en el algoritmo de la Figura 13 viene denotada por $f^{(0)}$. Como ya hemos visto en cálculos anteriores (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:

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

cuyo cálculo ya fue realizado en Gradient Boosting para regresión (3.2.2) y es:

$$g_m(x_i) = f^{(m-1)}(x_i) - y_i$$

A continuación, calculamos:

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

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 queremos 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. La función a minimizar es el sumatorio de las Loss Function (al igual que en Gradient Boosting) junto con una serie

de parámetros de regularización. 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 $\sum_{i=1}^n f_m(x_i) = \sum_{j=1}^T \left(\sum_{i \in I_j} \omega_j \right)$ con $I_j = \{i / x_i \in R_j\}$, podemos reescribir la expresión anterior 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, observamos que para nodos hoja donde encontramos 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, observamos que 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 , realizamos 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 (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 con la que estamos trabajando:

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

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 tomamos como predicción final del modelo $f^{(M)}(x)$.

Clasificación

El desarrollo realizado en el caso de regresión es igualmente válido para el caso de clasificación. La única diferencia reside en la Loss Function escogida. Al igual que en Gradient Boosting, XGBoost suele utilizar:

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

Por tanto, los valores calculados anteriormente cambiarán. 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-odds-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, trataremos de explicar, sin entrar en muchos detalles, cuáles son estos métodos que aplica.

Por un lado, como ya se ha comentado, XGBoost se ideó para conjuntos de datos muy grandes. Por ello, 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, si tomamos los percentiles P_{25} , P_{50} y P_{75} , solo tendremos que probar con 3 valores por variable y escoger el óptimo. Cabe citar que, 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 hemos 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 forma habitual y en el caso de clasificación se tendrán las últimas predicciones realizadas. La idea intuitiva de por qué los pesos vienen dados por $h_m(x_i)$ pasa por observar que podemos 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*) es un modelo desarrollado por Microsoft que fue publicado en 2017 [4]. Su funcionamiento es similar al de XGBoost pero añadiendo una serie de matices que consiguen que el algoritmo sea muy potente y eficiente.

La diferencia principal reside en el modo de construir los árboles. En cada capa de nodos, XGBoost realiza la división de todos los nodos y luego evalúa si dicha evaluación se debe llevar a cabo a través del valor de “*Gain*”. Por el contrario, se podría decir que LightGBM no trabaja por capas. 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, nos encontraremos 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.

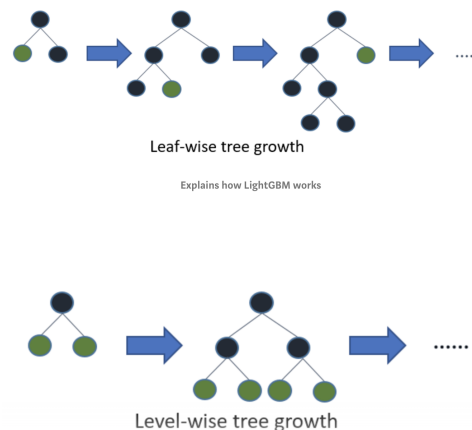


Figura 14: Construcción de árboles en LightGBM vs XGBoost (Fuente: [18])

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.

3.2.5. CatBoost

CatBoost es un modelo desarrollado por Yandex también publicado en 2017 [5]. 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 una clase el valor 0 y a la otra clase el valor 1. 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:

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

Por ejemplo:

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 5: *One-hot-encoding*

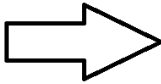
De esta manera, el problema quedaría resuelto. Cada nueva variable V_j hará referencia a la propiedad de “pertenecer a la clase A_j ” y el conjunto de datos ya estará correctamente codificado para su uso. En este trabajo ha sido necesario realizar un preprocesado de los datos y se ha utilizado la técnica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



...	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

Figura 15: Conjunto de datos del ejemplo teórico

donde “Country” es la variable explicativa que deseamos 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.

Para evitar el sobreajuste del modelo, bastaría 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:

	...	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

Figura 16: Codificación a través de proporciones

A continuación, se separa el conjunto de datos en dos de manera que el primer subconjunto de datos mantendrá el valor asignado y el segundo subconjunto no. Es decir:

	...	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

Figura 17: Separación del conjunto de datos

Así, 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, en nuestro ejemplo, para codificar al sexto individuo, necesitamos fijar un a (por ejemplo, tomaremos $a = 1$) y realizar :

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

y por tanto, el valor de la codificación para el sexto individuo queda:

	...	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

Figura 18: Codificación sexto individuo

Para el octavo individuo:

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

y:

	...	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

Figura 19: Codificación octavo individuo

Con el fin de ajustarnos con la mayor exactitud posible al funcionamiento real del modelo, 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, hemos explicado el proceso para el caso de clasificación, pero el caso de regresión no presenta grandes diferencias; lo único que debemos tener en cuenta es que la p (que antes hemos definido 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 estamos interesados en codificar).

Además, al igual que LightGBM y XGBoost, CatBoost añade mejoras del ámbito informático para ciertas optimizaciones. También, cabe destacar que este modelo construye árboles de decisión simétricos:

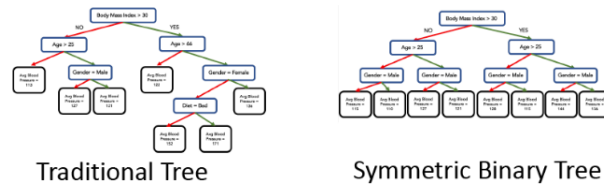


Figura 20: Árboles simétricos. Fuente: [19]

3.2.6. Ejemplo

Una vez más, haremos una predicción de la probabilidad de lluvia del día próximo con nuestro conjunto de datos “Rain in Australia”. Ahora, aplicaremos los algoritmos de boosting estudiados a lo largo de la sección para observar sus resultados, compararlos entre ellos y comprobar que se trata de modelos muy potentes que, a su vez, evitan el “overfitting”. En este caso, no es necesario establecer un criterio de parada en la construcción de los árboles, dado que la estructura de los modelos hace que estos no se sobreajusten pese a construir árboles muy profundos. No obstante, por realizar una comparación justa con los algoritmos de bagging, fijaremos la profundidad máxima de cada árbol en 5.

Al igual que en los ejemplos anteriores, utilizaremos la métrica de Gini para realizar las comparaciones. Los resultados son los siguientes:

Como podemos 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.

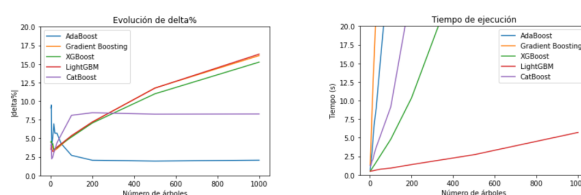
Además, no hay que obviar el hecho de que muchos de estos modelos están sobreajustados. Mientras que al trabajar con modelos de bagging no surgen problemas de overfitting, con modelos de boosting tendremos que preocuparnos por ver que no estamos cometiendo errores en la elección del número de iteraciones 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,

	Run_Time	Train_Gini%	Test_Gini%	delta%		Run_Time	Train_Gini%	Test_Gini%	delta%		Run_Time	Train_Gini%	Test_Gini%	delta%			
	AdaB_1	0.7	36.0	32.0	-9.0		GB_1	1.3	66.0	63.0	-4.0		XGB_1	0.5	66.0	63.0	-4.0
	AdaB_3	1.2	44.0	39.0	-9.0		GB_3	2.8	69.0	66.0	-4.0		XGB_3	0.6	69.0	66.0	-5.0
	AdaB_5	1.8	54.0	52.0	-4.0		GB_5	4.3	70.0	67.0	-4.0		XGB_5	0.6	70.0	67.0	-4.0
	AdaB_10	3.0	57.0	54.0	-5.0		GB_10	8.4	72.0	69.0	-4.0		XGB_10	0.9	71.0	68.0	-4.0
	AdaB_15	4.9	62.0	57.0	-7.0		GB_15	12.5	73.0	70.0	-3.0		XGB_15	1.1	73.0	70.0	-3.0
	AdaB_20	6.7	64.0	61.0	-6.0		GB_20	15.7	74.0	71.0	-3.0		XGB_20	1.3	74.0	71.0	-4.0
	AdaB_30	8.9	66.0	63.0	-6.0		GB_30	22.3	75.0	73.0	-4.0		XGB_30	1.7	75.0	73.0	-4.0
	AdaB_50	15.4	69.0	66.0	-4.0		GB_50	38.5	78.0	74.0	-4.0		XGB_50	2.6	78.0	74.0	-4.0
	AdaB_100	30.4	71.0	69.0	-3.0		GB_100	74.6	80.0	76.0	-5.0		XGB_100	4.8	80.0	76.0	-5.0
	AdaB_200	59.4	73.0	71.0	-2.0		GB_200	158.1	83.0	77.0	-7.0		XGB_200	10.4	83.0	77.0	-7.0
	AdaB_500	148.4	74.0	73.0	-2.0		GB_500	402.2	88.0	78.0	-12.0		XGB_500	32.8	88.0	78.0	-11.0
	AdaB_1000	291.5	75.0	73.0	-2.0		GB_1000	771.6	92.0	77.0	-16.0		XGB_1000	65.6	92.0	78.0	-15.0

	Run_Time	Train_Gini%	Test_Gini%	delta%		Run_Time	Train_Gini%	Test_Gini%	delta%		
	LGBM_1	0.5	66.0	63.0	-4.0		CatB_1	1.3	65.0	62.0	-4.0
	LGBM_3	0.5	69.0	66.0	-4.0		CatB_3	1.2	70.0	67.0	-4.0
	LGBM_5	0.5	70.0	68.0	-4.0		CatB_5	1.4	72.0	71.0	-2.0
	LGBM_10	0.5	72.0	70.0	-3.0		CatB_10	1.9	74.0	72.0	-2.0
	LGBM_15	0.6	73.0	71.0	-3.0		CatB_15	2.2	75.0	73.0	-3.0
	LGBM_20	0.6	74.0	71.0	-4.0		CatB_20	2.8	76.0	73.0	-3.0
	LGBM_30	0.7	76.0	73.0	-4.0		CatB_30	3.7	78.0	74.0	-4.0
	LGBM_50	0.8	78.0	74.0	-4.0		CatB_50	5.3	79.0	75.0	-5.0
	LGBM_100	0.9	80.0	76.0	-5.0		CatB_100	9.1	82.0	75.0	-8.0
	LGBM_200	1.4	83.0	77.0	-7.0		CatB_200	24.9	85.0	78.0	-8.0
	LGBM_500	2.7	88.0	78.0	-12.0		CatB_500	62.2	86.0	79.0	-8.0
	LGBM_1000	5.7	93.0	78.0	-16.0		CatB_1000	120.8	86.0	79.0	-8.0

Figura 21: Resultados modelos de boosting

Figura 22: Evolución del $\delta\%$ y del tiempo de ejecución

LGBM apenas necesita 5 segundos.

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 podemos controlar cada cuantas iteraciones se calculará el valor del incremento.

Por otro lado, cabe observar que a lo largo del trabajo hemos 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(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 en XGBoost una implementación de este último caso, cuya salida es un vector de probabilidades.

3.2.8. Ventajas y desventajas

Tras la debida explicación del funcionamiento de los algoritmos de boosting, cabe realizar una lista de aquellas propiedades que los hacen preferibles respecto a otros algoritmos y aquellas que resultan desfavorables.

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 paráme-

tros 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 actual, predigan si habrá precipitaciones el día próximo. En esta sección, resumiremos los resultados obtenidos en los diferentes ejemplos.

En primer lugar, realizaremos una comparación de los mejores modelos de cada categoría. El criterio que estableceremos para la selección de modelos será el siguiente: para cada algoritmo, escogeremos 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 serán los siguientes:

	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

Figura 23: Modelos seleccionados

A través de estos modelos, se pueden generar los siguientes gráficos:

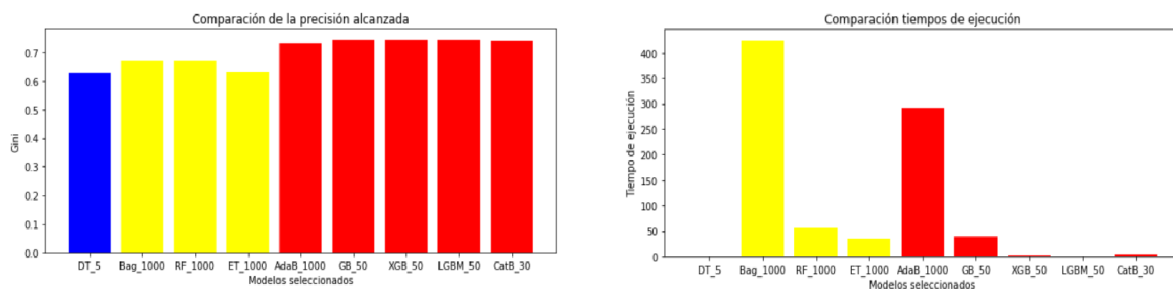


Figura 24: Comparación de precisión y tiempo de ejecución

En vista de los gráficos, es evidente que los modelos que obtienen las mejores predicciones son aquellos que hacen uso del 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.

Otros modelos a los que el gráfico no hace justicia son los modelos 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, tratemos de representar la información en otro gráfico más completo.

El nuevo gráfico que generaremos nos mostrará la precisión de cada modelo en función del número de iteraciones y nos indicará, a través del tamaño de los puntos, el tiempo de ejecución registrado. Con tal de poder observar bien la información, dado que el coeficiente de Gini converge a partir de las 100 iteraciones, no contemplaremos las iteraciones posteriores en el gráfico:

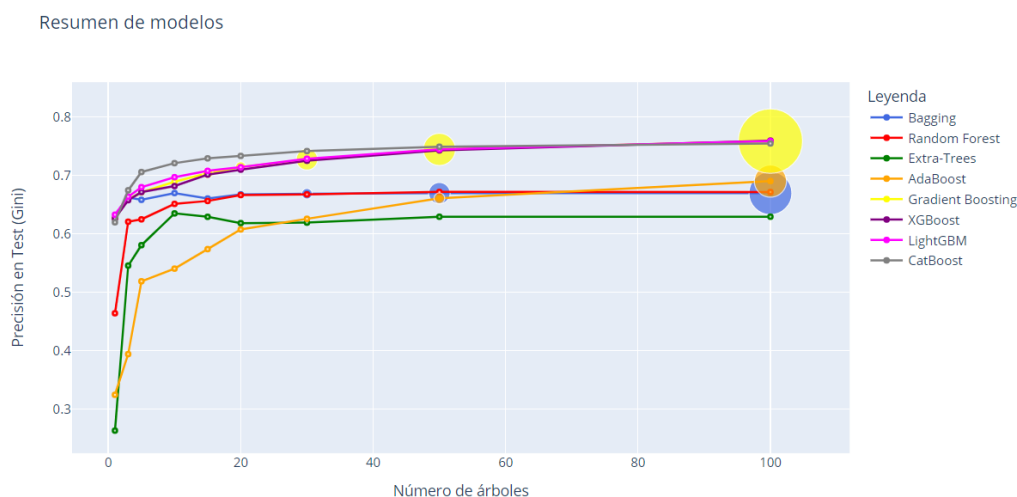


Figura 25: Resumen de modelos

De esta manera, comprobamos 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, en particular, son muy imprecisos.

Por tanto, a través de este gráfico se puede obtener una información muy completa acerca del nivel de precisión alcanzable con cada modelo y del coste implicado.

5. Conclusión

A lo largo del trabajo se han visto 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, hemos llegado a métodos más sofisticados que consiguen reducir los errores cometidos en las predicciones, e incluso evitar el sobreajuste, mediante diferentes técnicas. Con el fin de sintetizar la información que se ha detallado, a continuación se realizará una tabla resumen que contendrá, de manera esquemática, la información más importante de cada algoritmo:

Algoritmo	Metodología	Python	Parámetros	Control <i>Overfitting</i>	Coste Computacional	Consideraciones
CART	Simple	Scikit-Learn: - DecisionTreeRegressor - DecisionTreeClassifier		Profundidad máxima	Bajo	- Exploración inicial de datos - Método sencillo - Admite representación gráfica - Inestable
Bagging	Bagging	Scikit-Learn: - BaggingRegressor - BaggingClassifier		Profundidad máxima de los árboles generados	Alto	- Uso de todas las variables explicativas disponibles - Convergencia del error
Random Forest	Bagging	Scikit-Learn: - RandomForestRegressor - RandomForestClassifier	m	Profundidad máxima de los árboles generados	Medio	- Cada árbol utiliza m variables - Disminución de la varianza - Convergencia del error
Extra-Trees	Bagging	Scikit-Learn: - ExtraTreesRegressor - ExtraTreesClassifier	m	Profundidad máxima de los árboles generados	Medio	- Mayor aleatoriedad - Útil ante problemas de <i>overfitting</i> extremos - Convergencia del error
AdaBoost	Boosting	Scikit-Learn				
Gradient Boosting	Boosting	Scikit-Learn				
XGBoost	Boosting	XGBoost				
Light GBM	Boosting	Light GBM				
CatBoost	Boosting	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 (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(3.2.5), ha sido necesario realizar un preprocesamiento de los datos para poder aplicar los diferentes algoritmos. Para ello, se ha seguido la siguiente estructura, la cual se encuentra disponible en la web de Kedro (https://kedro.readthedocs.io/en/stable/12_faq/01_faq.html) y supone una práctica común en el mundo laboral dado el interés en mantener una organización para poder acceder a versiones anteriores siempre que sea necesario:

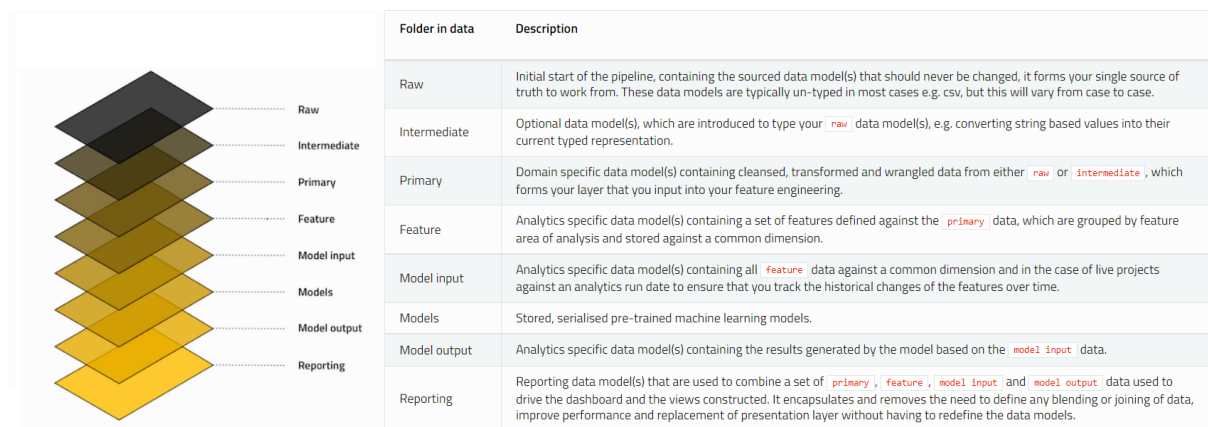


Figura 26: Figuras disponibles en el link de Kedro.

En cuanto al tiempo invertido a cada parte del trabajo, este se podría repartir, aproximadamente, de la siguiente manera:

Además, para el desarrollo de ciertos apartados del trabajo ha sido necesario apoyarse en

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 6: 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 7: Asignaturas relacionadas con el trabajo

Referencias

- [1] Breiman, L. (1996). Bagging predictors. *Machine Learning* **24**, 123-140.
- [2] Breiman, L. (2001). Random Forests. *Machine Learning* **45**, 5-32.
- [3] 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.
- [4] 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*
- [5] 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*

- [6] Drucker, H. (1997). Improving Regressors using Boosting Techniques. *Proceedings of the 14th International Conference on Machine Learning*.
- [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., Hastie, T. and Tibshirani, R. (2000). Additive logistic regression: a statistical view of boosting. *Annals of Statistics*, **28(2)**, 337-407.
- [9] Friedman, J. (2001). Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, **29(5)**, 1189-1232.
- [10] Ho, T.K. (1995). Random Decision Forests. *Proceedings of 3rd International Conference on Document Analysis and Recognition* **1**, 278-282.
- [11] Loh, W.Y. (2014). Fifty Years of Classification and Regression Trees. *International Statistical Review* **82(3)**, 329-348.
- [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] Breiman, L., Friedman, J.H, Olshen, R.A. and Stone, C.J. (1984). *Classification And Regression Trees*. Routledge.
- [14] Friedman, J.H, Hastie, T. and Tibshirani, R. (2001). *The Elements of Statistical Learning: data mining, inference and prediction*. Springer.
- [15] Hastie, T., James, G., Tibshirani, R. and Witten, D. (2013). *An Introduction to Statistical Learning with Applications in R*. Springer
- [16] Moisen, G. G. (2008). Classification and regression trees. In: Jorgensen, Sven Erik; Fath, Brian D., eds. Encyclopedia of Ecology, Volume 1. Oxford, U.K.: Elsevier. p. 582-588.
- [17] 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).

- [18] 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-7d7d7d7d7d7d> (Consultado el 12 de Mayo de 2022).
- [19] Segura, T., “CatBoost Algorithm” explained in 200 words. <https://thaddeus-segura.com/catboost/> (Consultado el 12 de Mayo de 2022).