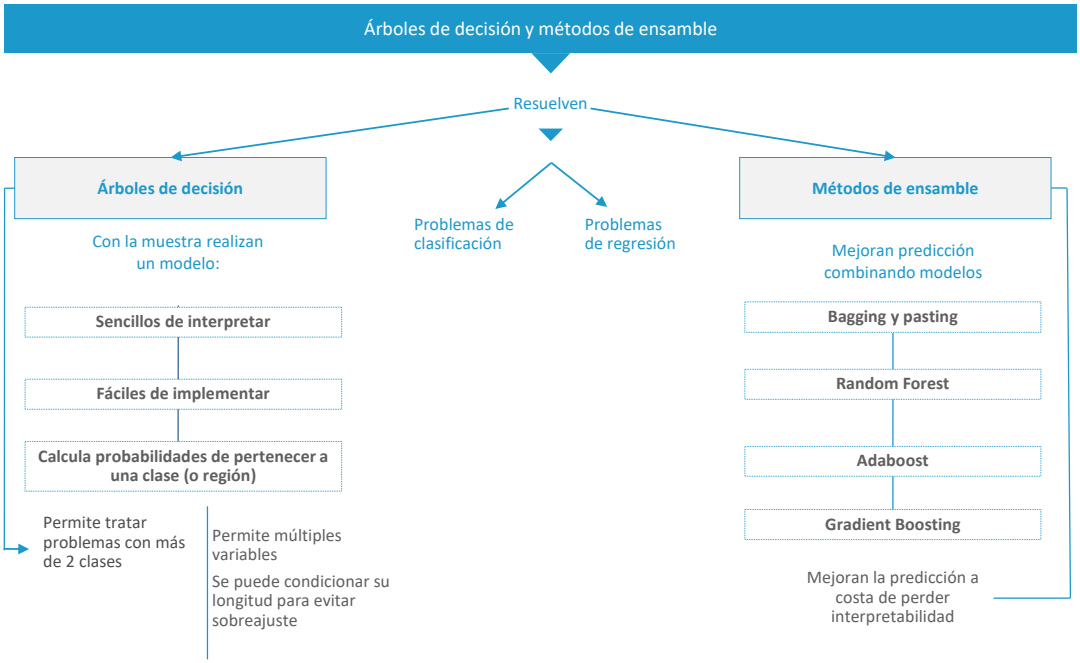
Técnicas Multivariantes

Árboles de decisión y métodos de ensamble

Índice

Esquema



Ideas clave

8.1 Introducción y objetivos

En este tema estudiaremos los árboles de decisión y los métodos de ensamble, centrándonos principalmente en los métodos de ensamble derivados de árboles. Estos métodos de ensamble, como los bosques aleatorios o el ensamble de árboles mediante *boosting*, se han desarrollado a partir de los árboles de decisión valiéndose de las ventajas que se obtienen de promediar los resultados de diferentes modelos.

Los métodos que se van a tratar en este tema, tanto los árboles como las técnicas de ensamble, pueden emplearse del mismo modo para resolver problemas de regresión o para resolver problemas de clasificación. Para desarrollar la explicación sobre su funcionamiento, veremos algunas librerías concretas y cómo emplearlas en python para resolver problemas reales con dichos métodos. Además, se resaltarán cuales son las principales ventajas de emplear estos métodos y se comentarán algunas de sus limitaciones.

8.2 Árboles de decisión

Un árbol de decisión consiste en una secuencia de condiciones sobre las distintas variables predictoras y su relación con la variable de respuesta. Estas condiciones se van sucediendo en distintos caminos del árbol, y por tanto, se van dividiendo las distintas posibilidades en ramas. Estas ramas, así pues, representan divisiones del espacio de los predictores en diferentes regiones.

Una vez definido el árbol, para poder realizar predicciones, habitualmente se emplea la media o la moda de la región a la cual pertenece dicha observación. Los métodos basados en árboles de decisión son simples y son fáciles de interpretar, aunque suelen tener unas capacidades de ajuste y de predicción peores que otro tipo de métodos como pueden ser la regresión penalizada. Sin embargo, para mejorar esta capacidad predictiva han aparecido los métodos de ensamble, en los que se combinan muchos de estos árboles, obteniéndose resultados de predicción mucho mejores (aunque a expensas de la interpretabilidad del modelo).

Funcionamiento

Para explicar el funcionamiento de un árbol de decisión, empezaremos utilizando un problema de clasificación, y posteriormente veremos su contrapartida en el modelo de regresión. En concreto, se va a emplear el conjunto de datos **Iris**, que es un conjunto de datos muy empleado en multitud de manuales y softwares estadísticos. El conjunto de datos **Iris** consiste en diferentes medidas sobre 150 flores de 3 especias distintas: *Iris Setosa*, *Iris versicolor* y *Iris virginica*. Este conjunto de datos se puede encontrar en el paquete Scikit-learn y para acceder a su contenido debemos importar la función *load_iris* del modulo *sklearn.datasets*.

Para construir un árbol de decisión sobre este conjunto de datos, se van a emplear únicamente dos variables predictoras: la **longitud del pétalo** y la **anchura del pétalo** y se va a construir el árbol mediante un objeto *DecisionTreeClassifier* del modulo *sklearn.tree*. Cuando realizamos el ajuste de los datos con el objeto *DecisionTreeClassifier* debemos fijar el nivel máximo de profundidad del árbol e introducir un valor de semilla para obtener siempre los mismos resultados (argumentos *max_depth* y *random_state*, respectivamente). Para construir este árbol se han utilizado los argumentos *max_depth=2* y *random_state=3* tal y como se muestra en el siguiente fragmento de código.

```
# cargar librerias-----------------------------------------------
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
```

```
# cargar dataset-----------------------------------------------
iris = load_iris()
# seleccionar longitud y anchura del petalo---------------------
X = iris.data[:, 2:]
# especie de la planta
y = iris.target
# crear el objeto de clase arbol--------------------------------
tree_clf = DecisionTreeClassifier(max_depth=2, random_state=3)
# ajustar el arbol----------------------------------------------
tree_clf.fit(X, y);
```

Una vez que se construye, se puede visualizar el árbol de decisión empleando el método *export_graphviz()* para obtener un archivo que hemos llamado *"arbol_iris.dot"*.

```
from graphviz import Source
from sklearn.tree import export_graphviz

# definir donde vamos a guardar la figura-----------------------
ROOT_DIR = "."
PATH_FIGURAS = os.path.join(ROOT_DIR, "images")
os.makedirs(PATH_FIGURAS, exist_ok=True)

export_graphviz(
        tree_clf,
        out_file = os.path.join(PATH_FIGURAS, "arbol_iris.dot"),
        feature_names = iris.feature_names[2:],
        class_names = iris.target_names,
        rounded = True,
        filled = True
    )
```

Posteriormente, se transforma este archivo a un archivo de formato gráfico, como

png (también es posible transformar el archivo a *pdf*), mediante el software gratuito *Graphviz*, que podemos instalar siguiendo las intrucciones que se encuentran en el siguiente link:

https://forum.graphviz.org/t/new-simplified-installation-procedure-on-windows/224

Una vez instalado y activado desde la consola de comando, podemos crear un archivo de tipo *png* a partir del archivo *.dot* yendo al directorio donde se encuentra dicho fichero e introduciendo en la consola de comando:

$ dot -Tpng arbol_iris.dot -o arbol_iris.png

obteniéndose la representación del árbol de decisión que se muestra en la Figura 1.

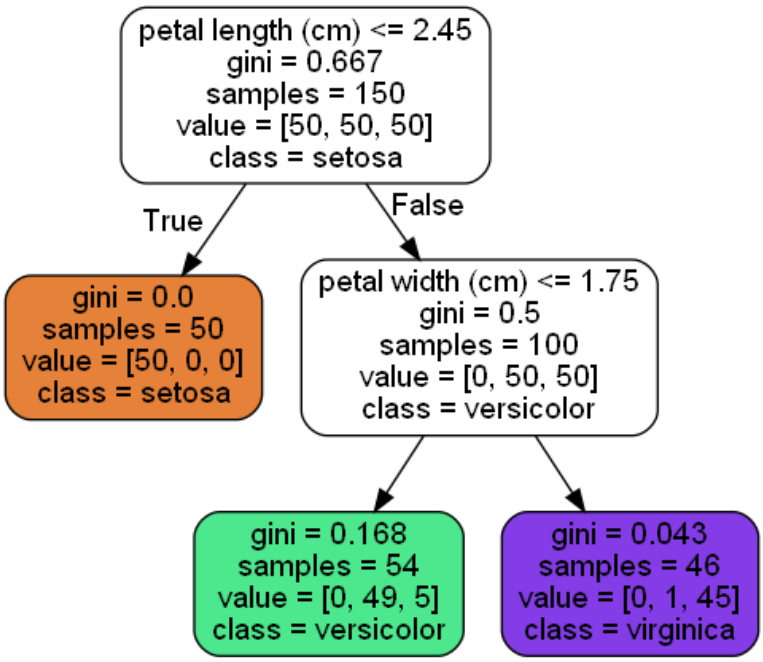


Figura 1: Árbol de decisión sobre el conjunto de datos Iris. Figura adaptada de (Geron, 2019)

En esta figura se muestran las decisiones que se siguen para poder clasificar las observaciones organizadas en nodos. Estos nodos se pueden identificar por la profundidad (*depth*) que ocupan (en este ejemplo tenemos los niveles de profundidad 0 ó raíz, 1 y 2) y por su posición (izquierda o derecha).

Supongamos que nos encontramos ante una nueva flor y la queremos clasificar en alguna de las 3 categorías. Entonces, se empezaría en el primer nodo, el nodo raíz, en la parte superior del árbol de decisión. Este nodo pregunta si la longitud del pétalo es menor que 2.45 cm. En caso afirmativo nos desplazaríamos hacia el nodo de abajo a la izquierda. Este nodo es una hoja, o más comúnmente *leaf*, ya que no tienen ningún otro nodo que descienda de él. En este caso veríamos qué clase predice el nodo y se habría terminado el recorrido en el árbol de decisión, obteniéndose que la clase predicha es: *Iris setosa*.

Ahora, supongamos que encontramos otra flor, y en esta ocasión la longitud del pétalo es mayor que 2.45 cm. Entonces, nos desplazamos en el árbol al nodo de abajo a la derecha del nodo raíz, el cual no es un nodo hoja, si no que es un nodo que pregunta otra condición sobre las variables predictoras: ¿es la anchura del pétalo más pequeña de 1.75 cm? En caso afirmativo, lo más probable es que esta flor pertenezca a la categoría *Iris versicolor* (nodo con profundidad 2 a la izquierda). Si es más grande de 1.75 cm entonces sería más probable que la flor sea *Iris virginica* (nodo con profundidad 2 a la derecha).

Es interesante hacer notar que una de las cualidades de los árboles de decisión es que no es necesario escalar las variables predictoras (por ejemplo, estandarizándolas), con lo que se simplifica el proceso de resolución de los problemas.

Cada nodo cuenta cuántas observaciones de la muestra de entrenamiento pertenecen a él: parámetro *sample*. En este ejemplo se tiene que 100 observaciones poseen una longitud del pétalo superior a 2.45 cm (nodo de profundidad 1, derecha, sample = 100), de las cuales, 46 tienen un ancho de pétalo superior a 1.75 cm (nodo de profundidad 2, derecha, sample = 46). La siguiente característica que aparece en cada nodo es el parámetro *value*. En este ejemplo, en el nodo de profundidad 2, derecha, se tiene un *value* =[0, 1, 45], que está indicando que en este nodo tenemos 0 observaciones *Iris setosa*, 1 observación *Iris versicolor* y 45 observaciones *Iris virginica*.

Medidas de impureza

Gini

Por último, en cada nodo se tiene la medida de impureza de Gini. Se dice que un nodo es "puro" si todas las observaciones del nodo son de la misma clase (gini=0). En nuestro ejemplo, el nodo de profundidad 1, izquierda, cumple con esta condición, ya que todas las observaciones que pertenecen a él son del tipo *Iris setosa*. La Ecuación 1 muestra como se obtiene la medida de impureza de gini para cada uno de los nodos en un árbol de decisión.

$$G_i = 1 - \sum_{k=1}^{n} P_{i,k}^2, \tag{1}$$

donde $P_{i,k}$ es la proporción de observaciones de clase k respecto a todas las observaciones del nodo. En el nodo de profundidad 2, derecha, el índice de gini se calcularía como:

$$G_{2d} = 1 - \left(\left(\frac{0}{46}\right)^2 + \left(\frac{1}{46}\right)^2 + \left(\frac{45}{46}\right)^2\right) = 1 - 0.957 = 0.043$$

Entropía

La medida que viene por defecto para calcular los árboles de decisión en *Scikit-Learn* es la medida de impureza de Gini, pero también se puede seleccionar la entropía, que

se define como se muestra en la Ecuación 2.

$$H_i = -\sum_{k=1}^{n} P_{i,k} \cdot \log_2(p_{i,k}). \quad (2)$$

Si aplicamos la entropía al nodo anterior, se tiene:

$$H_{2d} = -\left(0 + \frac{1}{46} \cdot \log_2\left(\frac{1}{46}\right) + \frac{45}{46} \cdot \log_2\left(\frac{45}{46}\right)\right) = -(-0.120 - 0.031) = 0.151$$

En la mayoría de las ocasiones, ambas medidas arrojan árboles de decisión muy similares. Gini tiene un coste computacional algo menor, por lo que parece una buena decisión por defecto. Sin embargo, en algunas ocasiones en que nos interese obtener nodos más balanceados (la medida de Gini tiende a aislar las observaciones de la clase más frecuente en su propia rama del árbol), el uso de la entropía puede ser más conveniente.

Visualización del árbol de decisión

La figura 2 muestra cómo se dividen las distintas regiones asociadas a los nodos del árbol de decisión del ejemplo. La linea vertical sólida representa la decisión del nodo raíz (profundidad 0): longitud del pétalo = 2.45 cm. La región que queda limitada a su izquierda es pura (únicamente contiene flores del tipo *Iris setosa*), y por tanto, no puede dividirse más. Sin embargo, la región que queda a la derecha es impura, y el nodo de profundidad 1 asociado a esta región genera dos nodos descendientes separados por la condición: anchura del pétalo = 1.75 cm (representado por una línea discontinua). Ya que para este ejemplo la profundidad máxima se ha fijado en 2 (*max_depth = 2*) el árbol de decisión acaba aquí. En cambio, si se hubiese fijado el límite en 3 (*max_depth = 3*) se generaría, para cada uno de los dos nodos de profundidad 2, dos descendientes

de profundidad 3, representados por las líneas punteadas. Como acabamos de ver, los árboles de decisión son intuitivos y la clasificación en los diferentes grupos es fácil de interpretar.

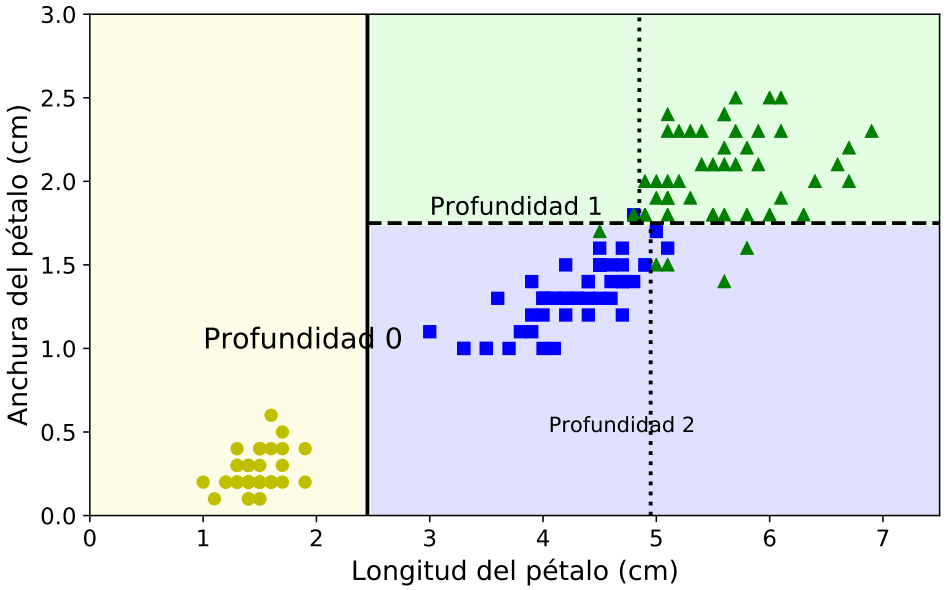


Figura 2: Regiones del árbol de decisión. Figura adaptada de (Geron, 2019)

Estimación de probabilidades

Un árbol de decisión puede estimar la probabilidad de que una observación pertenezca a una determinada clase *k*. Primero, se evalúa en qué nodo hoja se cumplen las condiciones de la nueva observación y después se puede obtener la proporción del número de muestras de entrenamiento de cada clase *k* que se encuentran en dicho nodo hoja. En el ejemplo anterior, si la nueva observación fuese una hoja con una longitud de pétalo de 5 cm y una anchura de pétalo de 1.5 cm, esta observación pertenecería al nodo de profundidad 2, izquierda. Así pues, las probabilidades de pertenecer a cada clase son las siguientes:

- *Iris setosa* $= 0/54 = 0\,\%$.
- *Iris versicolor* $= 49/54 = 90.7\,\%$.

- *Iris virginica* $= 5/54 = 9.3\,\%$.

Por lo tanto, podemos asignar a la nueva observación la predicción de la clase *Iris versicolor*, que es en la que se ha obtenido una mayor probabilidad de pertenencia según el árbol de decisión.

El algoritmo CART

Los árboles de decisión se pueden generar siguiendo distintos algoritmos. En este tema vamos a profundizar en el algoritmo que utiliza la librería *Scikit-Learn* en python, que es el que hemos empleado en el ejemplo anterior. La librería *Scikit-Learn* emplea el método CART, que es un acrónimo de las siglas en inglés *Classification and Regression Tree* para construir los árboles de decisión. En este algoritmo, los nodos que no son hojas siempre generan 2 nodos descendientes, ya que la pregunta que hace sobre la condición de alguna variable predictora es binaria. Sin embargo, existen otros métodos, como el algoritmo ID3, donde existen nodos que pueden generar más de 2 descendientes. El modo en el que el algoritmo CART funciona es el siguiente:

- Escoge una de las variables predictoras del modelo *k*.

- Fija un valor umbral t_k que se va a utilizar como frontera entre las regiones creadas.

La variable y el valor del umbral se escogen de modo que se consiga obtener una región que sea lo más pura posible según la medida de impureza de Gini (o de mínima entropía). Esto quiere decir que el algoritmo trata de obtener los valores de k y de t_k que minimicen la función de coste que se muestra en la Ecuación 3.

$$J(k, t_k) = \frac{m_{izquierda}}{m} \cdot G_{izquierda} + \frac{m_{derecha}}{m} \cdot G_{derecha}, \tag{3}$$

Donde $G_{izquierda/derecha}$ es la medida de impureza de Gini de la región izquierda o derecha y $m_{izquierda/derecha}$ es el número de observaciones que se encuentran en la región izquierda o derecha.

Una vez que el algoritmo CART ha dividido el conjunto de datos de entrenamiento en dos regiones, se repite el proceso y se divide cada una de estas regiones siguiendo el mismo mecanismo, y así sucesivamente hasta que se alcanza la profundidad máxima fijada o no se encuentra una división que consiga reducir más la pureza. Además, existen otros hiperparámetros, como veremos en el siguiente apartado, que permiten añadir condiciones de parada del algoritmo adicionales: *min_samples_split*, *min_samples_leaf*, *min_weight_fraction_leaf* y *max_leaf_nodes*.

A la vista del funcionamiento del algoritmo CART, se puede comprobar que aplicando dicho algoritmo no se tiene necesariamente que encontrar la solución óptima global, ya que se comprueba la impureza para cada nivel de profundidad, pero no comprueba que esa solución dé lugar a la solución óptima varios niveles de profundidad por debajo, pudiéndose alcanzar óptimos locales.

Hiperparámetros del modelo

Los árboles de decisión no necesitan realizar muchos suposiciones sobre la muestra de entrenamiento (en contraposición con, por ejemplo, los modelos lineales donde se asume que las relaciones son lineales). Si no se le añade ninguna restricción, la estructura del árbol se adaptará a la muestra de entrenamiento realizando un ajuste muy exacto. Normalmente, demasiado exacto, lo que va a producir problemas de sobreajuste.

Para evitar el sobreajuste en la muestra de entrenamiento (que conllevará a una mala predicción de observaciones no empleadas para el entrenamiento del modelo) es necesario restringir la libertad del árbol de decisión durante su creación. Una manera sencilla de evitar el posible sobreajuste del árbol es fijar un número de profundidad

máxima, es decir, del número de divisiones sucesivas que se realizan. En *Scikit-Learn*, se puede emplear el argumento *max_depth*, como hemos visto anteriormente, siendo conscientes de que por defecto no lleva ninguna restricción. Otras maneras de actuar sobre los parámetros del modelo y tratar de reducir un posible sobreajuste son:

- *min_samples_split* : El mínimo número de muestras que puede tener un nodo para que se pueda dividir.
- *min_samples_leaf*: El mínimo número de muestras que puede tener un nodo hoja.
- *min_weight_fraction_leaf*: Como el anterior, pero cuando se utilizan pesos en las observaciones.
- *max_leaf_nodes*: Máximo número de nodos hoja.
- *max_features*: El máximo número de variables que son consideradas por el algoritmo para decidir cada división.

Otra estrategia para evitar el sobreajuste, empleada por otros algoritmos, consiste en realizar el árbol de decisión completo sin restricciones y posteriormente recortar los nodos innecesarios.

La Figura 3 muestra dos árboles de decisión entrenados en el conjunto de datos **Moons**. **Moons** se ha generado mediante la función *make_moons* de la librería *Scikit-Learn*, que permite realizar conjuntos de datos aptos para probar algoritmos de clasificación. En particular, **Moons** se ha generado con un tamaño de 100 observaciones, donde se obtienen valores de dos variables predictoras, X_1 y X_2 con dos categorías de clasificación. En el siguiente trozo de código se muestra como se ha obtenido:

```
# crear dataset artificial moons--------------------------------
from sklearn.datasets import make_moons
Xm, ym = make_moons(n_samples=100, noise=0.25, random_state=53)
```

En el panel de la izquierda el árbol es entrenado con los hiperparámetros por defecto, es decir sin restricciones. En cambio, en el panel de la derecha el árbol es entrenado con el argumento *min_samples_leaf = 4*. Se observa que el árbol obtenido en el panel de la izquierda presenta sobreajuste.

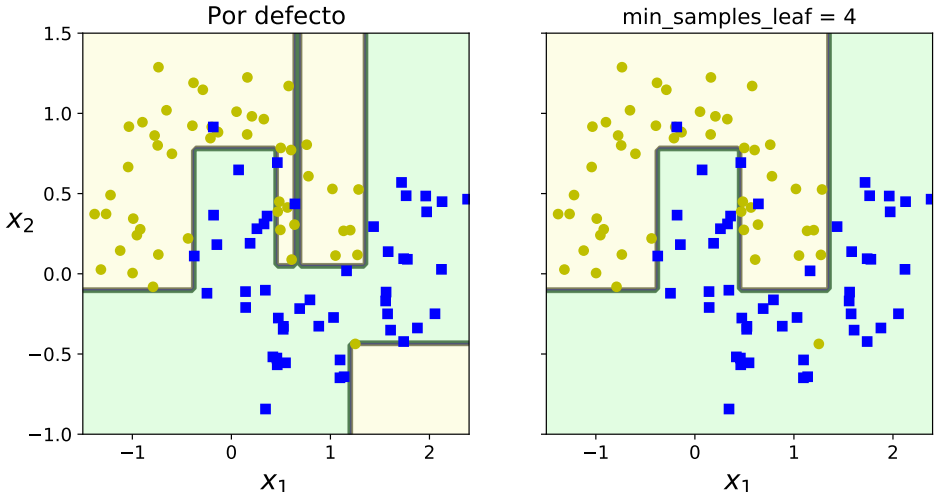


Figura 3: Árboles de decisión sobre el conjunto de datos Moon. Figura adaptada de (Geron, 2019)

Regresión

Hasta ahora hemos visto cómo funcionan los árboles de decisión para problemas de clasificación. No obstante, también pueden ser empleados para resolver problemas de regresión. Para este fin, podemos emplear la clase *DecissionTreeRegressor* del paquete Scikit-Learn. Se realiza, a modo de ejemplo, el entrenamiento de un conjunto de datos consistentes en una regresión cuadrática con algo de ruido. Además, se ha fijado la restricción de nivel de profundidad máximo igual a 2, es decir, *max_depth = 2*.

En la Figura 4 se muestra el árbol de decisión de manera similar al árbol que habíamos creado para el problema de clasificación. La diferencia principal radica de que en lugar de predecir la clase de cada nodo, predice un valor. Por ejemplo, supongamos que queremos realizar una predicción de una nueva observación con $x_1 = 0.6$. En el nodo raíz escogeríamos el descendiente de la izquierda ($x \leq 0.844$) y en el nodo de profundidad 1 escogeríamos el descendiente de la derecha ($x \geq 0.22$). En este caso, el

valor predicho para nuestra observación sería de 0.148, que es el valor medio de todas las observaciones de entrenamiento que pertenecen a ese nodo. El algoritmo que se emplea para generar las regiones es idéntico al que se empleaba para la clasificación, pero se sustituye en la función coste a la medida de impureza de Gini (o a la entropía) por el error cuadrático medio (mse).

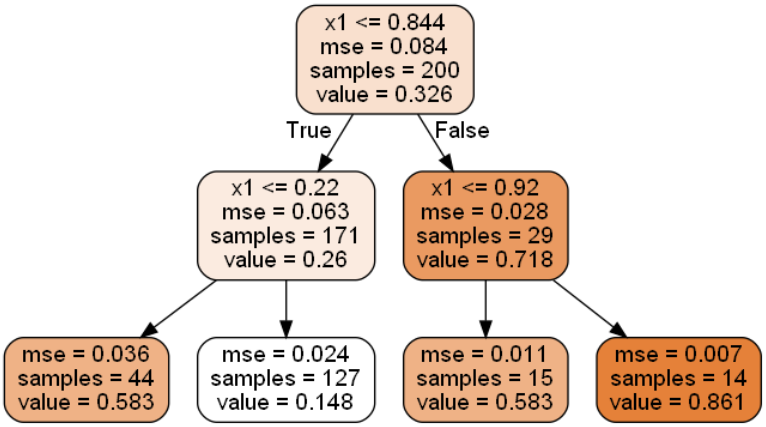


Figura 4: Árbol de decisión sobre un problema de regresión

Las predicciones del modelo se muestran en el panel izquierdo de la Figura 5. En el panel de la derecha, se ha añadido el árbol de decisión que se obtendría si se cambia el valor de la restricción a *max_depth = 3*. En ambos casos se observa que en cada región, el valor predicho es la media de las observaciones de la muestra de entrenamiento.

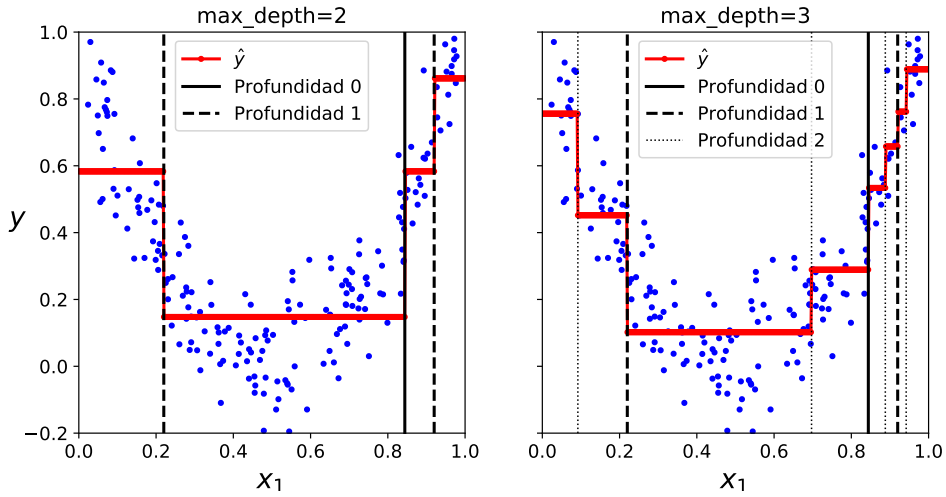


Figura 5: Regiones de los árboles de decisión con max_depth=2 y max_depth=3

Inconvenientes de los árboles de decisión.

Los árboles de decisión tienen muchas ventajas: son sencillos de implementar, fáciles de entender e interpretar, versátiles y permiten resolver tanto problemas de clasificación como de regresión. Sin embargo, también tiene algunas limitaciones:

- Los árboles funcionan con divisiones de las regiones perpendiculares a los ejes de las variables, por lo cual, son **sensibles ante rotaciones de los datos** de la muestra de entrenamiento. En la Figura 6 se muestra en el panel de la izquierda a un conjunto de datos sencillamente separable, y en el panel de la derecha el mismo conjunto de datos en el cual se ha introducido una rotación de 45º. Para obtener la misma solución se necesitan más divisiones en el segundo caso. Una manera de solventar este problema puede ser aplicar previamente a los datos un ánalisis por componentes principales, que veremos en el Tema 9, y que permitirá obtener una mejor orientación de los datos.

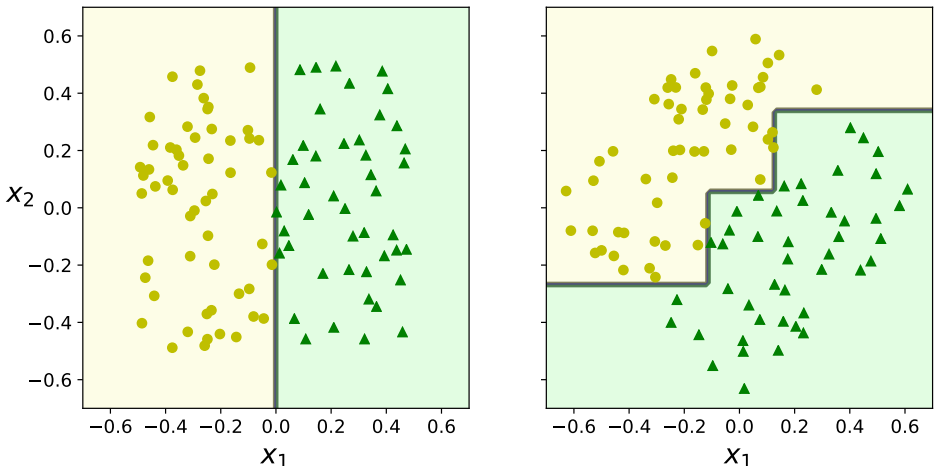


Figura 6: Sensibilidad de los árboles de decisión a la rotación. Figura adaptada de (Geron, 2019)

- Son **sensibles ante pequeñas variaciones en las observaciones de la muestra de entrenamiento**. Por ejemplo, si en el ejemplo que hemos venido trabajando eliminamos la observación que tenía un valor más grande de la anchura de pétalo de la especie *Iris versicolor* (4.8 cm de largo y 1.8 cm de ancho) y entrenamos un

nuevo árbol de decisión se obtiene un modelo (ver Figura 7) muy diferente al que habíamos obtenido sin eliminar esta observación (ver Figura 2). De hecho, el algoritmo para realizar el entrenamiento del árbol de decisión que emplea Scikit-Learn es estocástico, por lo que para obtener siempre el mismo resultado es necesario fijar el argumento *random_state_hyperparameter*.

```
## array([[4.8, 1.8]])
```

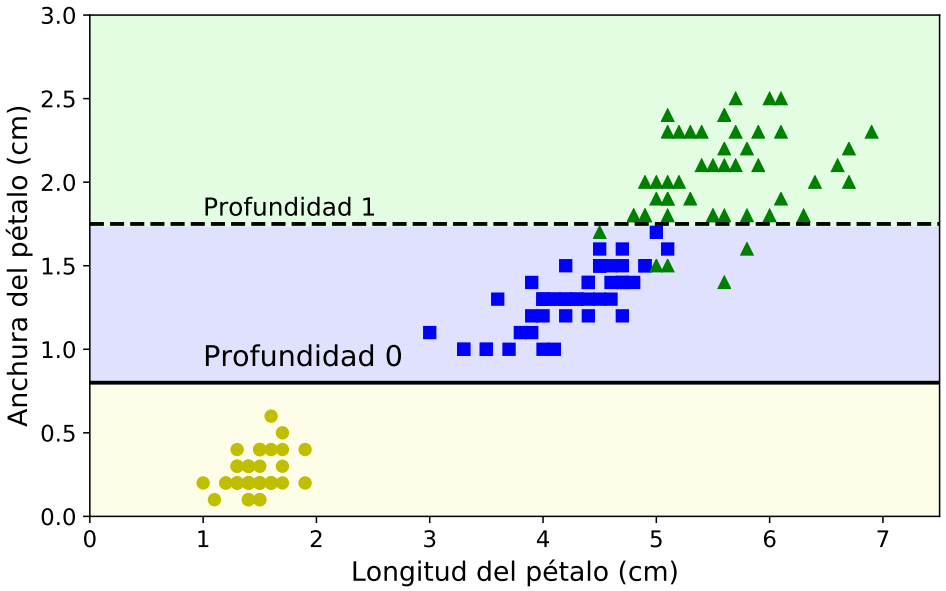


Figura 7: Sensibilidad de los árboles de decisión ante variaciones de la muestra de entrenamiento. Figura adaptada de (Geron, 2019)

En el próximo apartado veremos qué métodos podemos utilizar para tratar de corregir esta inestabilidad de los árboles de decisión.

8.3 Métodos de ensamble

Si se agrupan las predicciones de un grupo de predictores, en muchas ocasiones se tendrán mejores predicciones que las que se obtienen de estos predictores por separado. A un grupo de predictores se le suele llamar un ensamble y por lo tanto a los métodos

que agrupan estas predicciones se les conoce como métodos de ensamble. Dentro de los métodos de ensamble, principalmente podemos encontrar dos estrategias:

1. Escoger métodos muy diferentes entre sí (por ejemplo, podemos promediar las clasificaciones obtenidas mediante una regresión logística, un árbol de decisión y por el método KNN).

2. Emplear el mismo método de predicción pero entrenándolo con diferentes muestras de entrenamiento. Cuando esta muestra se obtiene con remplazamiento, el método de ensamble se conoce como *bagging*, que es la abreviatura de bootstrap aggregating. Cuando el muestreo se realiza sin remplazamiento, el método de ensamble se conoce como *pasting*.

En cualquiera de los casos, tanto en el *bagging* como en el *pasting*, se permite que las mismas observaciones se empleen para entrenar distintos predictores, pero sólo el *bagging* permite que una misma observación se utilice varias veces para el mismo modelo predictor.

Una vez que todos los modelos predictores son entrenados, el ensamble permite realizar predicciones de una nueva observación agregando las predicciones de cada uno de los predictores que lo conforman. Esta agregación, suele ser la moda (es decir, la predicción más frecuente) para los ensambles de clasificación y la media para los problemas de regresión. Cada predictor, de forma individual, tiene un mayor sesgo que si se hubiera entrenado con la muestra de entrenamiento original, pero al producir el ensamble, se consigue reducir tanto el sesgo como la varianza de manera simultánea. Generalmente, el ensamble tendrá un sesgo similar al que se obtendría de entrenar un único predicto con la muestra de entrenamiento original pero una menor varianza.

Evaluación fuera de la muestra

Cuando realizamos un ensamble de tipo *bagging*, para cada predictor tendremos algunas observaciones de la muestra de entrenamiento repetidas mientras que otras observaciones no aparecerán ninguna vez. En particular, el número de observaciones diferentes de la muestra original en cada réplica de Bootstrap es aproximadamente el 63 %. El 37 % de observaciones restantes están, por tanto, fuera de la muestra o *out of bag* (de aquí, en adelante, nos refereriremos como **oob**) y pueden emplearse para evaluar la capacidad predictiva de cada predictor. Para obtener esta evaluación en Scikit-Learn, se puede seleccionar el argumento *oob_score = True* cuando se crea el objeto de clase *BaggingClassifier*. El resultado de dichas evaluaciones queda registrado en la variable *oob_score*.

Además de permitir obtener distintas réplicas de la muestra, el clasificador *BaggingClassifier* también permite seleccionar las variables predictoras que se van a emplear para realizar cada predictor mediante los hiperparámetros *max_features* y *bootstrap_features*. Esta técnica es especialmente útil en problemas de alta dimensionalidad. Muestrear al mismo tiempo las observaciones y las variables predictoras se conoce como el método de los caminos aleatorios (*Random Patches*). En cambio, entrenar por por todas las observaciones y únicamente muestrear las variables predictoras se conoce como el método de los subespacios aleatorios (*Random Subspaces*). Realizar un muestreo sobre las variables predictoras de cada uno de los predictores conlleva una mayor diversidad en la predicción, reduciendo la varianza en los resultados a costa de aumentar un ligeramente el sesgo.

Una vez que ya se han comentado cómo se realizan los diferentes métodos de ensamble estamos en disposición de utilizar un método de ensamble basado en árboles de decisión: los bosques aleatorios o *Random Forests*.

8.4 Random Forests

Los bosques aleatorios, más comúnmente conocidos como *Random Forests*, es un método de ensamble empleando *bagging* (o menos frecuentemente *pasting*) en el cual cada uno de los predictores es un árbol de decisión. Estos árboles de decisión se generan fijando el número máximo de nodos hoja a un valor razonablemente pequeño y en vez de escoger la mejor variable para realizar la división, se escoge la mejor variable de una selección aleatoria de todas las variables disponibles.

El método *Random Forest* introduce una aleatoreidad adicional cuando construye los distintos árboles de decisión, con lo cual se genera un mayor sesgo en las predicciones en beneficio de disminuir la varianza, dando como resultado, de forma general, un mejor modelo. En *Scikit-Learn*, podemos encontrar las clases *RandomForestClassifier* y *RandomForestRegressor*, las cuales, nos permiten obtener estos modelos. En la figura 8 se muestra la diferencia del modelo de predicción de un árbol de decisión y de un modelo predictivo mediante *Random Forest* sobre un conjunto de datos construido con la función *make_moons*, observándose que el método de *Random Forest* presenta un menor sobreajuste.

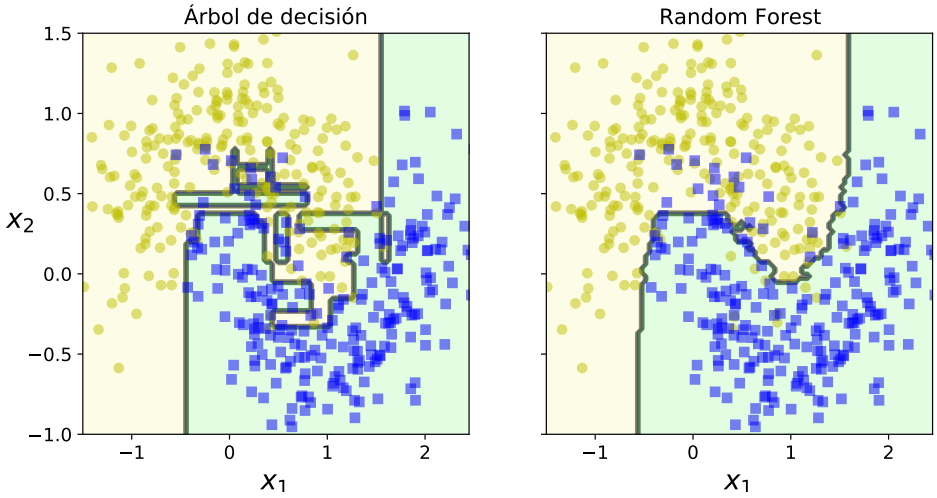


Figura 8: Modelo predictivo con un árbol de decisión vs modelo predictivo con Random Forest. Figura adaptada de (Geron, 2019)

Aún más, en vez de buscar el mejor corte de entre el subconjunto de variables para cada predictor, también se pueden realizar estas divisiones de forma también aleatoria. En este caso estaríamos ante los *Extremely Radomized Trees* (también conocidos como *Extra-trees*). Estos modelos consiguen reducir todavía más la varianza, a costa de introducir más sesgo en las predicciones.

Una de las ventajas que se obtienen de los métodos *Random forest* (sean Extra o no) es que es posible evaluar la importancia de las variables predictoras en el modelo de ensamble cuantificando cómo contribuyen a reducir la impureza en promedio entre los diferentes árboles de decisión. Esta medida es muy útil para ganar algo de interpretabilidad de los modelos y sirve para poder tomar decisiones sobre que variables es importante seguir midiendo en el futuro. En Scikit-Learn, se puede extraer esta información del ajuste de un objeto *RandomForestClassifier/Regressor* con la variable *feature_importances_*.

8.5 Boosting

Los métodos de *boosting* o de impulso son métodos que consisten en el ensamble de predictores débiles para lograr un predictor fuerte. La idea es entrenar estos predictores de forma secuencial, tratando de que cada nuevo predictor vaya mejorando a su predecesor. Entre los métodos más empleados de boosting podemos encontrar el boosting adaptativo (Adaboost) y el boosting por gradiente (Gradient Boosting).

Adaboost

Una manera de corregir al predictor anterior es prestar más atención a las observaciones de entrenamiento que dicho predecesor a ajustado erróneamente. Así, los sucesivos predictores van ajustando a las observaciones más difíciles de clasificar. Esta es la estrategia que se emplea en el método de Adaboost. Esto se puede realizar ajustando

pesos a las observaciones, que en un primer momento tendrán el mismo peso, y estos pesos se irán aumentando para las observaciones incorrectamente clasificadas, de modo que los siguientes predictores tenderán a fijarse más en ellas (por ejemplo, para realizar las divisiones de las regiones en el caso de los árboles de decisión). Una vez que todos los predictores han sido entrenados, el ensamble permite realizar una predicción agregando los resultados, de modo similar al bagging o al pasting, pero ponderando a estos predictores según su desempeño en la clasificación que han obtenido en su muestra de clasificación.

Gradient Boosting

Otro algoritmo muy popular es el *Gradient Boosting*, que del mismo modo que el *Adaboost* ajusta los predictores de forma secuencial corrigiendo cada nuevo predictor a su predecesor. Sin embargo, en vez de ir ajustando los pesos de las observaciones incorrectamente clasificadas, ajusta cada nuevo predictor con los residuos del predictor previo.

Por ejemplo, en un modelo de regresión simple se entrenaría al primer predictor con un árbol de decisión y se obtendrían los residuos de los verdaderos valores de la muestra de entrenamiento con los valores predichos. Estos residuos, ponderados con una constante llamada tasa de aprendizaje (*learning rate* = λ) se van a emplear para actualizar los valores de las observaciones. A continuación se entrena un segundo árbol de decisión con estas observaciones actualizadas y se obtendrían unos segundos residuos. Este proceso se repite hasta alcanzar el número de predictores de nuestro ensamble del modo que se muestra en el siguiente algoritmo.

1. definir $\hat{f}(x) = 0$ y $r_i = y_i$ para todas las observaciones i de la muestra de entrenamiento.

2. Para $n = 1, 2, ..., N$, repetir;

a) Ajustar un árbol $\hat{f}^b$ con d divisiones para los datos de entrenamiento (X, r).

b) Actualizar $\hat{f}$ añadiendo una versión encogida del nuevo árbol:

$$\hat{f}(x) = \hat{f}(x) + \lambda \hat{f}^b(x)$$

c) Actualizar los residuos:

$$r_i = r_i - \lambda \hat{f}^b(x_i)$$

3. Agregar los predictores en el ensamble:

$$\hat{f}(x) = \sum_{n=1}^{N} \lambda \hat{f}^b(x)$$

En el método del Gradient Boosting es necesario seleccionar, entre otros hiperparámetros:

- La profundidad máxima permitida, d, de cada uno de los árboles de decisión.
- El número máximo de árboles, N, es decir, de predictores en el ensamble.
- La tasa de aprendizaje (*learning rate*), λ, que va a determinar la velocidad a la cual se van a ir incorporando los nuevos predictores a los residuos.

Por último, es interesante hacer notar que además de las clases *GradientBoostingRegressor* y *GradientBoostingClassifier* disponibles en la librería Scikit-Learn, existe la librería *XGBoost*, que permite obtener este tipo de ensambles de manera mucho más optimizada, aunque su funcionamiento queda fuera del alcance de esta asignatura.

Material audiovisual

Accede al vídeo: Importancia de las variables y gráficos parciales

8.6 Referencias bibliográficas

Geron, A. (2019). *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems.* 2 edition.

Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The elements of statistical learning: data mining, inference and prediction*. Springer, 2 edition.

James, G., Witten, D., Hastie, T., and Tibshirani, R. (2013). *An introduction to statistical learning : with applications in R*. New York : Springer, [2013] ©2013.

Peña, D. (1987). *Regresión y diseño de experimentos*.

Tibshirani, R. (1996). Regression Selection and Shrinkage via the Lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, 58(1):267–288.

8.7 Ejercicios resueltos

Ejercicio 1.

Realiza los siguientes árboles de decisión sobre el conjunto de datos **Wine** y comenta las diferencias que se observan entre ellos.

a) Sin ningún tipo de restricciones.

b) Indicando que el nivel de profundidad máximo permitido es de 3.

c) Indicando que el mínimo número de observaciones que debe contener cada nodo hoja es de 4.

Solución

Wine es un conjunto de datos que se puede encontrar en *scikit-learn*. Consiste en las medidas químicas de varios tipos de vinos de una misma región de Italia producida por tres fabricantes distintos. En concreto, contiene 13 variables predictoras y una variable respuesta categórica con tres categorías: 0, 1 y 2 (que corresponden con los distintos productores del vino). Para cargar el conjunto de datos se debe emplear la función *load_wine* del modulo *sklearn.datasets*. Posteriormente cargamos el conjunto de datos empleando dicha función. Para saber más sobre las variables que componen el conjunto de datos, podemos modificar el argumento *as_frame = True*, para que se cargue como un data.frame y así poder emplear los métodos de la librería pandas tal y como se ha realizado en temas anteriores.

```
import pandas as pd
from sklearn.datasets import load_wine
# cargar dataset-----------------------------------------------
```

```
wine = load_wine(as_frame = True)
# ver variables predictoras wine----------------------------------
print(wine.data.info())
# ver variable respuesta-----------------------------------------
```

```
## <class 'pandas.core.frame.DataFrame'>
## RangeIndex: 178 entries, 0 to 177
## Data columns (total 13 columns):
##  #   Column                        Non-Null Count  Dtype
## ---  ------                        --------------  -----
##  0   alcohol                       178 non-null    float64
##  1   malic_acid                    178 non-null    float64
##  2   ash                           178 non-null    float64
##  3   alcalinity_of_ash             178 non-null    float64
##  4   magnesium                     178 non-null    float64
##  5   total_phenols                 178 non-null    float64
##  6   flavanoids                    178 non-null    float64
##  7   nonflavanoid_phenols          178 non-null    float64
##  8   proanthocyanins               178 non-null    float64
##  9   color_intensity               178 non-null    float64
##  10  hue                           178 non-null    float64
##  11  od280/od315_of_diluted_wines  178 non-null    float64
##  12  proline                       178 non-null    float64
## dtypes: float64(13)
## memory usage: 18.2 KB
## None
```

```
print(wine.target.value_counts())
```

```
## 1    71
## 0    59
## 2    48
```

```
## Name: target, dtype: int64
```

Se observa que el conjunto de datos **Wine** contiene 178 observaciones de 13 variables predictoras y que la variable respuesta categórica contiene 59 veces la categoría 0, 71 veces la categoría 1, y 48 veces la categoría 2. Una vez que ya se tiene una primera idea de las variables que forman el conjunto de datos **Wine** se procede a realizar los árboles de decisión (antes de realizar los árboles de decisión sería recomendable analizar cada una de las variables por separado, o por parejas, realizar descriptivos como histogramas, estudiar si existen variables que están correlacionadas, etc.).

a) Sin ningún tipo de restricciones

El primer árbol de decisión que se va a ajustar va a realizarse sin ningún tipo de restricciones. Para ello, dejaremos todos los argumentos de la función en sus valores por defecto, exceptuando el argumento *random_state*, que hace las funciones de fijar una semilla para que obtengamos siempre el mismo árbol, y que se va a fijar en 3.

```
from sklearn.tree import DecisionTreeClassifier
# seleccionar todas las variables--------------------------------
X = wine.data
# especie de la planta
y = wine.target
# crear el objeto de clase arbol---------------------------------
tree_clf = DecisionTreeClassifier(random_state = 3)
# ajustar el arbol-----------------------------------------------
tree_clf.fit(X, y);
```

Una vez que se ha ajustado el árbol, se genera el archivo *.dot* empleando la librería graphviz.

```
from graphviz import Source
from sklearn.tree import export_graphviz
```

```
# definir donde vamos a guardar la figura-----------------------
ROOT_DIR = "."
PATH_FIGURAS = os.path.join(ROOT_DIR, "images")
os.makedirs(PATH_FIGURAS, exist_ok=True)

export_graphviz(
        tree_clf,
        out_file = os.path.join(PATH_FIGURAS, "arbol_wine.dot"),
        feature_names = wine.feature_names,
        class_names = wine.target_names,
        rounded = True,
        filled = True
    )
```

Posteriormente, en la consola de comando se ejecuta la siguiente línea de comando con la que se obtiene la Figura 9.

$ dot -Tpng arbol_wine.dot -o arbol_wine.png

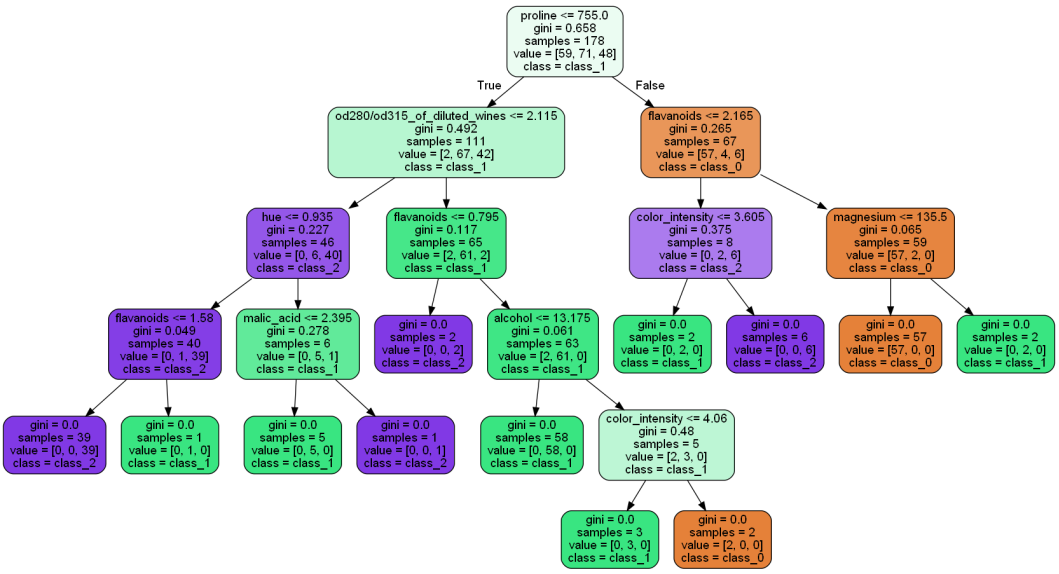


Figura 9: Árbol de decisión sobre el conjunto de datos Wine

Podemos ver que este árbol de decisión es muy largo y seguramente presentará un

sobreajuste si se emplease ante observaciones que no se han utilizado para la construcción del modelo.

b) Indicando que el nivel de profundidad máximo permitido es de 3.

Ahora, se va a restringir la longitud de las ramas del árbol fijando el número máximo de profundidad *max_depth = 3*.

```
# crear el objeto de clase arbol fijando profundidad maxima a 3--
tree_clf_b = DecisionTreeClassifier(random_state=3, max_depth=3)
# ajustar el arbol-----------------------------------------------
tree_clf_b.fit(X, y);
```

Una vez que se ha ajustado el árbol, se genera el archivo *.dot* empleando la librería graphviz.

```
# generar el .dot------------------------------------------------
export_graphviz(
        tree_clf_b,
        out_file = os.path.join(PATH_FIGURAS, "arbol_wine_b.dot"),
        feature_names = wine.feature_names,
        class_names = wine.target_names,
        rounded = True,
        filled = True
    )
```

Posteriormente, en la consola de comando se ejecuta la siguiente línea de comando con la que se obtiene la Figura 10.

$ dot -Tpng arbol_wine_b.dot -o arbol_wine_b.png

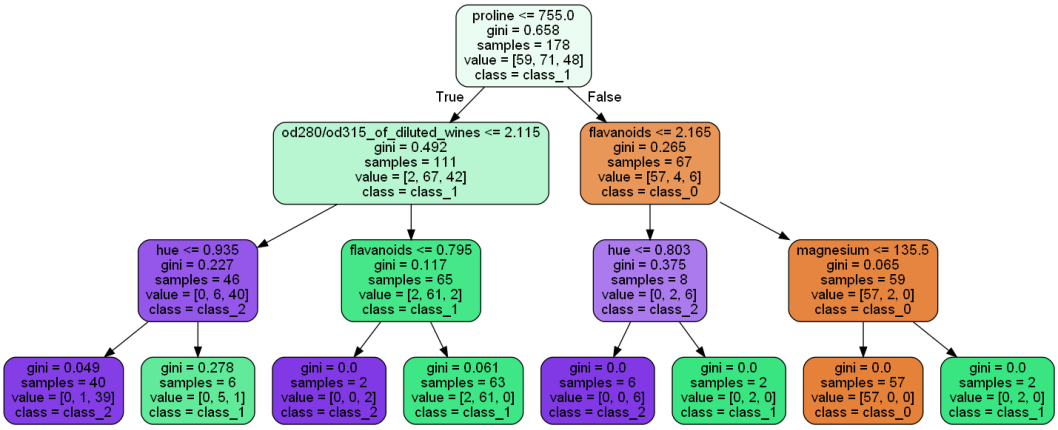


Figura 10: Árbol de decisión con profundidad máxima restringida

En este caso, el árbol tiene una longitud más razonable.

c) Indicando que el mínimo número de observaciones que debe contener cada nodo hoja es de 4.

Otra manera de restringir la longitud del árbol de decisión sin fijar la profundidad máxima es indicando el número mínimo de observaciones que debe contener cada nodo hoja. En este caso se fijará el valor de *min_samples_leaf = 4*.

```
# crear el objeto de clase arbol fijando min_leaf a 4------------
tree_clf_c = DecisionTreeClassifier(random_state=3,
min_samples_leaf=4)
# ajustar el arbol-----------------------------------------------
tree_clf_c.fit(X, y);
```

Una vez que se ha ajustado el árbol, se genera el archivo *.dot* empleando la librería graphviz.

```
# generar el .dot------------------------------------------------
export_graphviz(
        tree_clf_c,
        out_file = os.path.join(PATH_FIGURAS, "arbol_wine_c.dot"),
        feature_names = wine.feature_names,
```

```
        class_names = wine.target_names,
        rounded = True,
        filled = True
    )
```

Por último, en la consola de comando se ejecuta la siguiente línea de comando con la que se obtiene la Figura 11.

$ dot -Tpng arbol_wine_c.dot -o arbol_wine_c.png

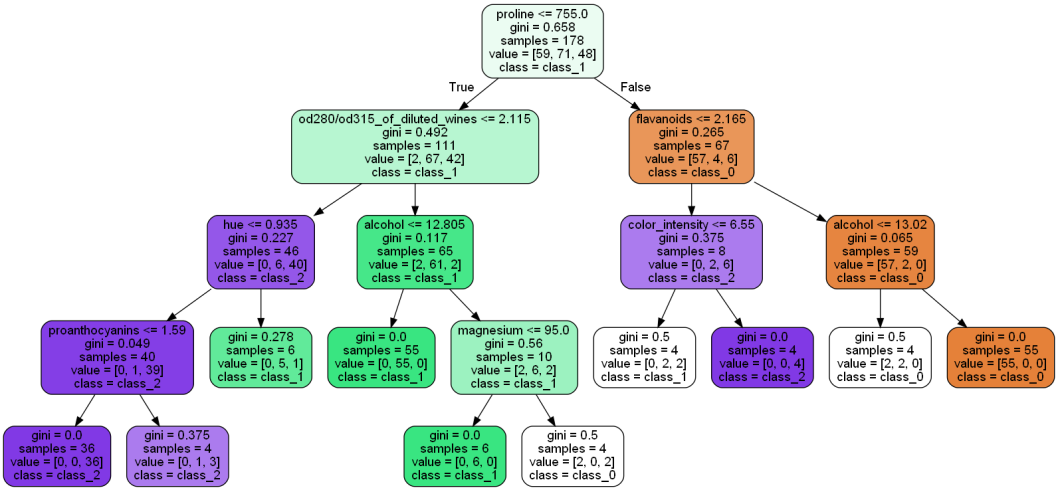


Figura 11: Árbol de decisión condicionado al número de observaciones en cada nodo hoja

En este caso, la longitud del árbol es superior a la que se obtenía en el apartado b. Además, algunos nodos hoja tienen un valor del índice de impureza de gini bastante elevados, por lo que a falta de realizar alguna comprobación mediante validación cruzada de los árboles de decisión realizados, se escogería el árbol obtenido en el apartado b.

Ejercicio 2.

Para el conjunto de datos del apartado anterior (el conjunto de datos *Wine*), realiza un modelo de ensamble de árboles de decisión donde agrupes 500 modelos predictores entrenados con 75 observaciones de la muestra de entrenamiento cada uno. Para poder validar el modelo, divide el conjunto de datos en 128 observaciones para la muestra de entrenamiento y 50 para la muestra de validación.

a) Con reemplazamiento (bagging).

b) Sin reemplazamiento (pasting).

c) Realizando Random Forest (fijando el número de nodos hoja máximo a 4).

¿Con qué método se obtiene una mejor predicción?

Solución

En este ejercicio se pide que realicemos distintos modelos de ensamble. También se indica que el número de predictores del ensamble debe ser 500 y que se deben emplear 128 observaciones para conformar la muestra de entrenamiento (por lo tanto se validará el ensamble con las 50 observaciones restantes) y que cada modelo predictor del ensamble debe emplear 75 de las 128 observaciones.

El primer paso consiste en dividir el conjunto de datos en la muestra de entrenamiento (X_train, y_train) y en la muestra de validación. Para ello definirermos una función, tal y como se vió en el tema 3. En la función particiones es necesario introducir la proporción de observaciones que se van a emplear para la muestra de entrenamiento. En nuestro caso 128/178 es aproximadamente el 71.9 % de las observaciones totales (por lo tanto, el test será el 28.1 %).

```
# cargar numpy-------------------------------------------------
import numpy as np
# definir semilla para que la particion sea la misma-------------
np.random.seed(3)
# definir funcion particiones-----------------------------------
def particiones(target, dataset, test_part):
    test_part_size = int(len(dataset) * test_part)
    mezclar_indices = np.random.permutation(len(dataset))
    test_indices = mezclar_indices[:test_part_size]
    train_indices = mezclar_indices[test_part_size:]
    return dataset.iloc[train_indices], dataset.iloc[test_indices],
 target.iloc[train_indices], target.iloc[test_indices]
# usar funcion particiones con test_part 0.28--------------------
X_train, X_test, y_train, y_test = particiones(wine.target,
wine.data, 0.281)
```

a) Con reemplazamiento (bagging).

Scikit_Learn dispone de la clase *BaggingClassifier* (o *BaggingRegressor* para los problemas de regresión) que permite crear ensambles. El modo de indicar si se requiere emplear el método de *bagging* o *pasting* es mediante el argumento *bootstrap*, que por defecto es igual a *True*, que significa *bagging* y para indicar que el tipo de ensamble es *pasting* es necesario modificar el argumento y pasarlo a *False*.

En este primer apartado se va a realizar el ensamble empleando *bagging*. Para ello, empleamos la función *BaggingClassifier* de *scikit-learn*. Se define el número de modeles predictores (*n_estimators = 500*), el número de observaciones de cada modelo (*max_samples = 75*) y como se está muestreando con reemplazamiento se define la opción (*bootstrap = True*). Una vez que se ajusta el modelo de ensamble, se puede estimar la bondad del mismo mediante la *accuracy* sobre la muestra de test.

```
# cargar librerias-----------------------------------------------
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

# crear objeto de la clase BaggingClassifier---------------------
bag_clf = BaggingClassifier(
    DecisionTreeClassifier(random_state = 3), n_estimators = 500,
    max_samples = 75, bootstrap=True, random_state = 3);

# ajustar el modelo----------------------------------------------
bag_clf.fit(X_train, y_train);

# obtener estimaciones del modelo sobre la muestra de test-------
```

```
## BaggingClassifier(base_estimator=DecisionTreeClassifier
##                   (random_state=3),
##                   max_samples=75, n_estimators=500, random_state=3)
```

```
y_pred = bag_clf.predict(X_test)

# comprobar resultados-------------------------------------------
from sklearn.metrics import accuracy_score
print(accuracy_score(y_test, y_pred))
```

```
## 0.94
```

Se obtienen un valor muy alto de la *accuracy*: 0.94.

b) Sin reemplazamiento (Pasting).

De modo similar, se realiza el ensamble empleando *pasting*. Para ello, empleamos la misma función que en el apartado anterior (*BaggingClassifier* de *scikit-learn*), y

mantenemos todos los argumentos igual (*n_estimators = 500*, *max_samples = 75*) exceptuando el que hace referencia al tipo de muestreo, que en este caso será (*bootstrap = False*). Una vez que se ajusta el modelo de ensamble, se puede estimar la bondad del mismo mediante la *accuracy* sobre la muestra de test del mismo modo que en el apartado anterior.

```
# crear objeto de la clase BaggingClassifier---------------------
pas_clf = BaggingClassifier(
    DecisionTreeClassifier(random_state = 3), n_estimators = 500,
    max_samples = 75, bootstrap=False, random_state = 3);

# ajustar el modelo----------------------------------------------
pas_clf.fit(X_train, y_train);

# obtener estimaciones del modelo sobre la muestra de test-------
```

```
## BaggingClassifier(base_estimator=DecisionTreeClassifier
##                   (random_state=3),
##                   bootstrap=False, max_samples=75,
##                   n_estimators=500, random_state=3)
```

```
y_pred_pas = pas_clf.predict(X_test);

# comprobar resultados-------------------------------------------
print(accuracy_score(y_test, y_pred_pas))
```

```
## 0.94
```

La medida del *accuracy* es la misma tanto con el método de *bagging* que con el de *pasting*.

c) Por último, se realiza un modelo de ensamble de tipo *Random Forest*.

Para ello se emplea la clase *RandomForestClassifier* del modulo *sklearn.ensemble* fijando los argumentos del mismo modo que en los apartados anteriores (*n_estimators = 500, max_samples = 75*) e indicando que el número máximo de hojas nodo en cada árbol de los que se compone el Random Forest es 4 (*max_leaf_nodes = 4*).

```
# cargar librerias-----------------------------------------------
from sklearn.ensemble import RandomForestClassifier

# crear objeto de la clase RandomForestClassifier----------------
rnd_clf = RandomForestClassifier(n_estimators = 500,
max_leaf_nodes = 4, random_state = 3, max_samples = 75);

# ajustar el modelo----------------------------------------------
rnd_clf.fit(X_train, y_train);

# obtener estimaciones del modelo sobre la muestra de test-------
```

```
## RandomForestClassifier(max_leaf_nodes=4, max_samples=75,
##                                     n_estimators=500, random_state=3)
```

```
y_pred_rf = rnd_clf.predict(X_test);

# comprobar resultados-------------------------------------------
print(accuracy_score(y_test, y_pred_rf))
```

```
## 0.96
```

Con el método de *Random Forest* se obtiene el valor de *accuracy* más alto. Por lo tanto, de entre los tres métodos propuestos este sería el método escogido para realizar el ajuste.

```
# comparar con los resultados de un arbol de decision------------
tree_clf = DecisionTreeClassifier(random_state = 3, max_depth = 3);
tree_clf.fit(X_train, y_train);
```

```
y_pred_tree = tree_clf.predict(X_test);
print(accuracy_score(y_test, y_pred_tree))
```

```
## 0.9
```

Con un árbol de decisión, se obtiene un *accuracy* de 0.9, que aunque también es un valor muy alto, es menor que cuando se realiza un método de ensamble.

Ejercicio 3.

Ajusta el mismo modelo pero empleando Gradient Boosting. Obtén la importancia de las variables ajustadas tanto para el modelo de Random Forest empleado en el ejercicio anterior como para el obtenido en el modelo de Gradient Boosting.

Solución

Para realizar el modelo de ensamble empleando Gradient Boosting se va a utilizar el método *GradientBoostingClassifier* del módulo *sklearn.ensemble*. Para obtener una primera aproximación sobre la bondad de la predicción del modelo se dejan los valores por defecto (exceptuando el número de estimadores, que se va a fijar en 500).

```
# cargar librerias-----------------------------------------------
from sklearn.ensemble import GradientBoostingClassifier
# crear objeto de la clase BaggingClassifier---------------------
gb_clf = GradientBoostingClassifier(n_estimators = 500,
random_state = 3);

# ajustar el modelo----------------------------------------------
gb_clf.fit(X_train, y_train);
```

```
# obtener estimaciones del modelo sobre la muestra de test-------
```

```
## GradientBoostingClassifier(n_estimators=500, random_state=3)
```

```
y_pred_gb = gb_clf.predict(X_test);

# comprobar resultados----------------------------------------
print(accuracy_score(y_test, y_pred_gb))
```

```
## 0.92
```

Se obtiene un resultado de *accuracy* de 0.92. Ahora vamos a ajustar los hiperparámetros del ensamble para obtener un mejor resultado en la predicción. En concreto se van a probar distintos valores de *learning_rate* (dejando fijo *n_estimators*) y de *max_depth*.

```
# crear un conjunto de posibles valores-------------------------
learning_rates = [1, 0.5, 0.25, 0.1, 0.05, 0.01]
max_depths = [1, 2, 3, 4]
# inicializamos los vectores de resultados----------------------
resultados_test = []
# bucle para extraer resultados---------------------------------
for eta in learning_rates:
  for d in max_depths:
   gb_clf_i = GradientBoostingClassifier(learning_rate = eta,
   n_estimators = 500, random_state = 3, max_depth = d);
   # entrenamos al modelo-----------------------------------------
   gb_clf_i.fit(X_train, y_train);
   # prediccion sobre la muestra de validacion--------------------
   y_pred_test = gb_clf_i.predict(X_test);
   # accuracy de entrenamiento------------------------------------
   acc_test = accuracy_score(y_test, y_pred_test);
   # guardar resultados en el vector------------------------------
   resultados_test.append(acc_test);
```

Una vez que se han obtenido los modelos para cada una de las combinaciones de hiperparámetros podemos representar los valores de accuracy obtenidos, tal y como se muestra en la figura 12.

```
# resultados_test[0:6] # max_depth = 1
# resultados_test[6:12] # max_depth = 2
# resultados_test[12:18] # max_depth = 3
# resultados_test[18:24] # max_depth = 4
# learning_rates

# pintamos los resultados de entrenamiento y validacion----------
import matplotlib as mpl
import matplotlib.pyplot as plt
from matplotlib.legend_handler import HandlerLine2D

plt.figure(figsize=(8, 4.75))


line1, = plt.plot(learning_rates, resultados_test[0:6], "b",
label = "max_depth = 1")
line2, = plt.plot(learning_rates, resultados_test[6:12], "r",
label = "max_depth = 2")
line2, = plt.plot(learning_rates, resultados_test[12:18], "g",
label = "max_depth = 3")
line2, = plt.plot(learning_rates, resultados_test[18:24], "y",
label = "max_depth = 4")
plt.legend(handler_map={line1: HandlerLine2D(numpoints = 4)})
plt.ylabel("Accuracy")
plt.xlabel("learning rate")
plt.show()
```

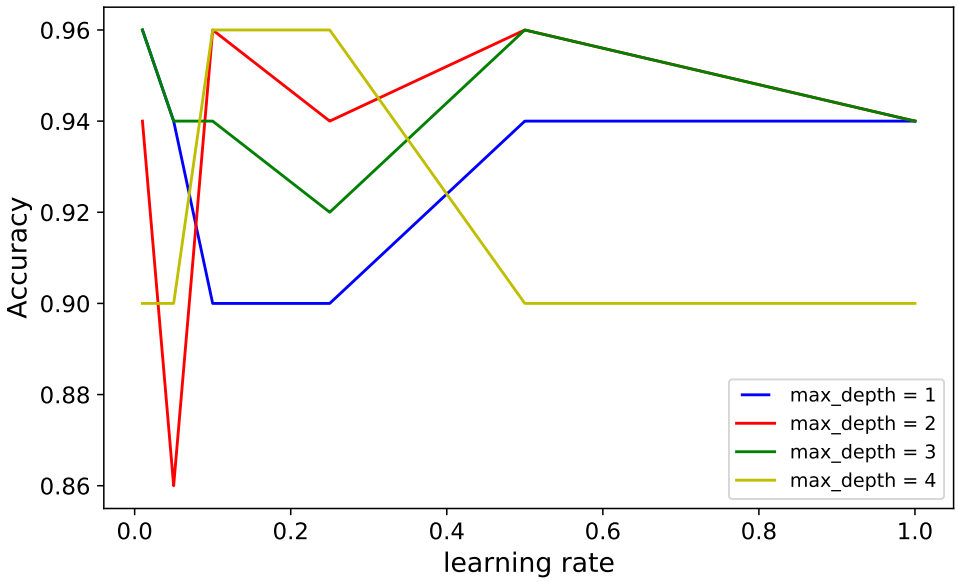


Figura 12: Resultados de los modelos de gradient boosting

Los mejores resultados en cuanto al *accuracy* en la predicción se obtienen con las combinaciones:

- max_depth = 1 y learning_rate = 0.01.
- max_depth = 2 y learning_rate = 0.5 ó 0.1.
- max_depth = 3 y learning_rate = 0.5 ó 0.01.
- max_depth = 4 y learning_rate = 0.25 ó 0.1.

Con todas las combinaciones anteriores se obtendría un valor en la *accuracy* sobre la muestra de validación de 0.96, que es el mejor que se ha conseguido de entre todos los distintos métodos de ensamble.

Por último, en la Figura 13 se muestra la importancia de las variables de dos de los modelos que obtienen la mejor puntuación en cuanto a *accuracy*. Se observa que las variables más importantes son las mismas para ambos modelos, pero que en el modelo

que se permite el ensamble de árboles de profundidad 4 aparecen más variables con importancia no nula que en el modelo que se permite una profundidad máxima de 1. Esto puede ser debido a que dichas variables no son importantes para el modelo por sus efectos directos, pero si a través de interacciones con otras variables. La codificación de las variables se puede obtener a través del método *.info* del objeto *X_train*. Se observa que las variables más importantes para los dos modelos son: *12. proline, 9. color_intensity y 6. flavanoids*.

```
# importancia1
gb_clf_i1 = GradientBoostingClassifier(learning_rate = 0.01,
   n_estimators = 500, random_state = 3, max_depth = 1)

gb_clf_i1.fit(X_train, y_train)
```

```
## GradientBoostingClassifier(learning_rate=0.01, max_depth=1, n_estimators=500,
##                            random_state=3)
```

```
importancias = gb_clf_i1.feature_importances_
indices = np.argsort(importancias)[::-1]

# importancia2
gb_clf_i2 = GradientBoostingClassifier(learning_rate = 0.1,
   n_estimators = 500, random_state = 3, max_depth = 4)

gb_clf_i2.fit(X_train, y_train)
```

```
## GradientBoostingClassifier(max_depth=4, n_estimators=500, random_state=3)
```

```
importancias2 = gb_clf_i2.feature_importances_
indices2 = np.argsort(importancias2)[::-1]

# dibujar_importancias
plt.figure(figsize = (12, 13))

plt.subplot(2, 1, 1)
```

```
plt.title("Importancia de las variables d=1 lr=0.01")
plt.bar(range(X.shape[1]), importancias[indices],
        color="r",  align="center")
```

```
## <BarContainer object of 13 artists>
```

```
plt.xticks(range(X.shape[1]), indices)
```

```
plt.xlim([-1, X.shape[1]])
```

```
## (-1.0, 13.0)
```

```
plt.subplot(2, 1, 2)
plt.title("Importancia de las variables d=4 lr=0.1")
plt.bar(range(X.shape[1]), importancias2[indices2],
        color="r",  align="center")
```

```
## <BarContainer object of 13 artists>
```

```
plt.xticks(range(X.shape[1]), indices)
```

```
plt.xlim([-1, X.shape[1]])
```

```
## (-1.0, 13.0)
```

```
plt.show()
```

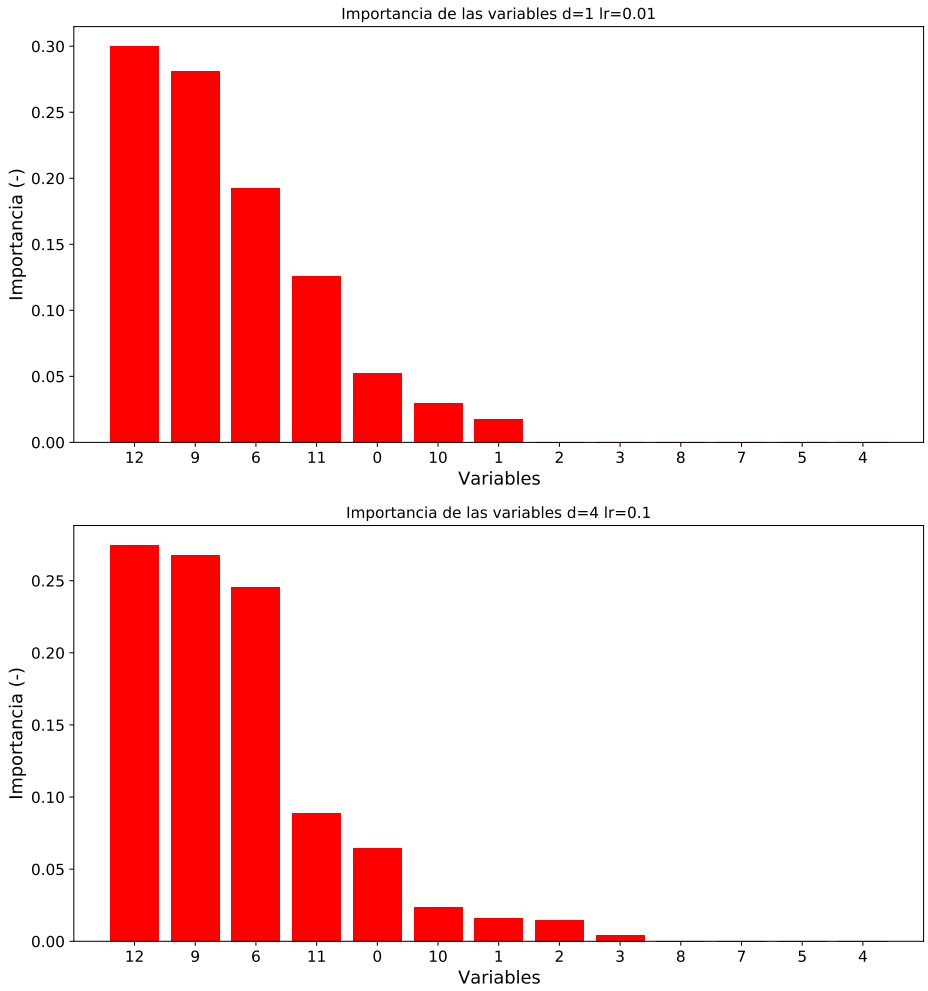


Figura 13: Importancia de las variables de los modelos de gradient boosting

```
X_train.info()
```

```
## <class 'pandas.core.frame.DataFrame'>
## Int64Index: 128 entries, 8 to 106
## Data columns (total 13 columns):
##  #   Column                        Non-Null Count  Dtype
## ---  ------                        --------------  -----
##  0   alcohol                       128 non-null    float64
##  1   malic_acid                    128 non-null    float64
```

```
##  2   ash                          128 non-null    float64
##  3   alcalinity_of_ash            128 non-null    float64
##  4   magnesium                    128 non-null    float64
##  5   total_phenols                128 non-null    float64
##  6   flavanoids                   128 non-null    float64
##  7   nonflavanoid_phenols         128 non-null    float64
##  8   proanthocyanins              128 non-null    float64
##  9   color_intensity              128 non-null    float64
##  10  hue                          128 non-null    float64
##  11  od280/od315_of_diluted_wines 128 non-null    float64
##  12  proline                      128 non-null    float64
## dtypes: float64(13)
## memory usage: 14.0 KB
```