CLASIFICACIÓN CON ÁRBOLES DE DECISIÓN, II

Abraham Sánchez López
FCC/BUAP
Grupo MOVIS

MOVIS

Facultad de Ciencias de la Computación

Construyendo tu primer modelo

- En esta sección, aprenderás cómo construir un árbol de decisiones con rpart y cómo ajustar sus hiperparámetros.
- Imagina que trabajas en la participación pública en un santuario de vida silvestre. Tienes la tarea de crear un juego interactivo para niños para enseñarles sobre diferentes clases de animales.
- El juego pide a los niños que piensen en cualquier animal del santuario y luego les haces preguntas sobre las características físicas de ese animal. Según las respuestas que dé el niño, el modelo debe decirle a qué clase pertenece su animal (mamífero, ave, reptil, etc.).
- Es importante que tu modelo sea lo suficientemente general como para poder usarse en otros santuarios de vida silvestre. Comencemos cargando los paquetes mlr y tidyverse:

```
> library(mlr)

> library(tidyverse)
```

Cargando y explorando el dataset zoo, I

- Cargamos el conjunto de datos del zoológico integrado en el paquete mlbench, convirtámoslo en un tibble y exploremos.

- Contamos con un tibble que contiene 101 casos y 17 variables de observaciones realizadas en varios animales; 16 de estas variables son lógicas, indican la presencia o ausencia de alguna característica, y la variable tipo es un factor que contiene las clases de animales que deseamos predecir.

```
> data(Zoo, package = "mlbench")

> zooTib <- as_tibble(Zoo)

> zooTib
```

```
# A tibble: 101 × 17
   hair  feathers eggs  milk  airborne aquatic predator toothed backbone breathes venomous fins   legs tail  domestic
   <lgl> <lgl>    <lgl> <lgl> <lgl>    <lgl>   <lgl>    <lgl>   <lgl>    <lgl>    <lgl>    <lgl> <int> <lgl> <lgl>
 1 TRUE  FALSE    FALSE TRUE  FALSE    FALSE   TRUE     TRUE    TRUE     TRUE     FALSE    FALSE     4 FALSE FALSE
 2 TRUE  FALSE    FALSE TRUE  FALSE    FALSE   FALSE    TRUE    TRUE     TRUE     FALSE    FALSE     4 TRUE  FALSE
 3 FALSE FALSE    TRUE  FALSE FALSE    TRUE    TRUE     TRUE    TRUE     FALSE    FALSE    TRUE      0 TRUE  FALSE
 4 TRUE  FALSE    FALSE TRUE  FALSE    FALSE   TRUE     TRUE    TRUE     TRUE     FALSE    FALSE     4 FALSE FALSE
 5 TRUE  FALSE    FALSE TRUE  FALSE    FALSE   TRUE     TRUE    TRUE     TRUE     FALSE    FALSE     4 TRUE  FALSE
 6 TRUE  FALSE    FALSE TRUE  FALSE    FALSE   FALSE    TRUE    TRUE     TRUE     FALSE    FALSE     4 TRUE  FALSE
 7 TRUE  FALSE    FALSE TRUE  FALSE    FALSE   FALSE    TRUE    TRUE     TRUE     FALSE    FALSE     4 TRUE  TRUE
 8 FALSE FALSE    TRUE  FALSE FALSE    TRUE    FALSE    TRUE    TRUE     FALSE    FALSE    TRUE      0 TRUE  TRUE
 9 FALSE FALSE    TRUE  FALSE FALSE    TRUE    TRUE     TRUE    TRUE     FALSE    FALSE    TRUE      0 TRUE  FALSE
10 TRUE  FALSE    FALSE TRUE  FALSE    FALSE   FALSE    TRUE    TRUE     TRUE     FALSE    FALSE     4 FALSE TRUE
# ℹ 91 more rows
# ℹ 2 more variables: catsize <lgl>, type <fct>
# ℹ Use `print(n = ...)` to see more rows
```

Cargando y explorando el dataset zoo, II

- Desafortunadamente, mlr no nos permitirá crear una tarea con predictores lógicos, así que, en su lugar, los convertiremos en factores. Hay algunas formas de hacer esto, pero la función mutate_if() de dplyr resulta útil aquí.
- Esta función toma los datos como primer argumento (o podríamos haberlo canalizado con %>%). El segundo argumento es nuestro criterio para seleccionar columnas, por lo que aquí hemos usado is.logic para considerar solo las columnas lógicas.
- El argumento final es qué hacer con esas columnas, así que usamos as.factor para convertir las columnas lógicas en factores. Esto dejará intacto el tipo de factor existente.

```
> zooTib <- mutate_if(zooTib, is.logical, as.factor)
```

- *Sugerencia*. Alternativamente, podrías haber usado mutate_all(zooTib, as.factor), porque la columna de tipo ya es un factor.

Entrenamiento del modelo, I

- En esta sección, te guiaremos en el entrenamiento de un modelo de árbol de decisión utilizando el algoritmo rpart.
- Ajustaremos los hiperparámetros del algoritmo y entrenaremos un modelo utilizando la combinación óptima de hiperparámetros.
- Definamos nuestra tarea y nuestro aprendizaje, y construyamos un modelo como de costumbre. Esta vez, proporcionamos "classif.rpart" como argumento para makeLearner() para especificar que vamos a utilizar rpart.

```
> zooTask <- makeClassifTask(data = zooTib, target = "type")

Warning in makeTask(type = type, data = data, weights = weights, blocking = blocking,  :

  Provided data is not a pure data.frame but from class tbl_df, hence it will be converted.

> tree <- makeLearner("classif.rpart")
```

- A continuación, debemos realizar un ajuste de hiperparámetros. Recuerda que el primer paso es definir un espacio de hiperparámetros sobre el cual queremos buscar.

Entrenamiento del modelo, II

- Veamos los hiperparámetros disponibles para el algoritmo rpart, en el listado que se muestra más adelante. Ya hemos analizado los hiperparámetros más importantes para el ajuste: minsplit, minbucket, cp y maxdepth.
- Hay algunos otros que pueden resultarte útil conocer. El hiperparámetro maxcompete controla cuántas divisiones candidatas se pueden mostrar para cada nodo en el resumen del modelo.
- El resumen del modelo muestra las divisiones de los candidatos en orden de cuánto mejoraron el modelo (ganancia de Gini). Puede resultar útil comprender cuál fue la siguiente mejor división después de la que realmente se usó, pero ajustar maxcompete no afecta el rendimiento del modelo, solo su resumen.
- El hiperparámetro maxsurrogate es similar a maxcompete pero controla cuántas divisiones sustitutas se muestran. Una división sustituta es una división que se utiliza si a un caso particular le faltan datos para la división real.
- De esta manera, rpart puede manejar los datos faltantes mientras aprende qué divisiones se pueden usar en lugar de las variables faltantes.

Entrenamiento del modelo, III

- El hiperparámetro maxsurrogate controla cuántos de estos sustitutos se retendrán en el modelo (si a un caso le falta un valor para la división principal, se pasa a la primera división sustituta, luego al segundo sustituto si también le falta un valor para la primera madre sustituta, etc.).
- Aunque no nos falta ningún dato en nuestro conjunto de datos, los casos futuros que deseamos predecir podrían hacerlo.
- Podríamos *establecer* esto en cero para ahorrar algo de tiempo de cálculo, lo que equivale a no utilizar variables sustitutas, pero hacerlo podría reducir la precisión de las predicciones realizadas en casos futuros con datos faltantes.
- El valor predeterminado de 5 suele estar bien.
- *Consejo*: recuerda que podemos contar rápidamente el número de valores faltantes por columna de un data.frame o tibble ejecutando map_dbl(zooTib,~sum(is.na(.))).
- El hiperparámetro usesurrogate controla cómo el algoritmo utiliza divisiones sustitutas.

Entrenamiento del modelo, IV

- Un valor de cero significa que no se utilizarán sustitutos y los casos en los que falten datos no se clasificarán. Un valor de 1 significa que se utilizarán sustitutos, pero si a un caso le faltan datos para la división real y para todas las divisiones de sustitutos, ese caso no se clasificará.
- El valor predeterminado de 2 significa que se utilizarán sustitutos, pero un caso al que le falten datos para la división real y para todas las divisiones sustitutas se enviará a la rama que contenía la mayor cantidad de casos.
- El valor predeterminado de 2 suele ser apropiado.
- **Nota**: Si tienes casos en los que faltan datos para la división real y todas las divisiones sustitutas de un nodo, probablemente deberías considerar el impacto que tienen los datos faltantes en la calidad de su conjunto de datos.

```
> getParamSet(tree)
```

Entrenamiento del modelo, V

	Type	len	Def	Constr	Req	Tunable	Trafo
minsplit	integer	-	20	1 to Inf	-	TRUE	-
minbucket	integer	-	-	1 to Inf	-	TRUE	-
cp	numeric	-	0.01	0 to 1	-	TRUE	-
maxcompete	integer	-	4	0 to Inf	-	TRUE	-
maxsurrogate	integer	-	5	0 to Inf	-	TRUE	-
usesurrogate	discrete	-	2	0,1,2	-	TRUE	-
surrogatestyle	discrete	-	0	0,1	-	TRUE	-
maxdepth	integer	-	30	1 to 30	-	TRUE	-
xval	integer	-	10	0 to Inf	-	FALSE	-
parms	untyped	-	-	-	-	TRUE	-

- Ahora, definamos el espacio de hiperparámetros en el que queremos buscar. Vamos a ajustar los valores de minsplit (un número entero), minbucket (un número entero), cp (un numérico) y maxdepth (un número entero).

- **Nota**: Recuerda que usamos makeIntegerParam() y makeNumericParam() para definir los espacios de búsqueda para hiperparámetros enteros y numéricos, respectivamente.

```
> treeParamSpace <- makeParamSet(
+   makeIntegerParam("minsplit", lower = 5, upper = 20),
+   makeIntegerParam("minbucket", lower = 3, upper = 10),
+   makeNumericParam("cp", lower = 0.01, upper = 0.1),
+   makeIntegerParam("maxdepth", lower = 3, upper = 10))
```

Entrenamiento del modelo, VI

- A continuación, podemos definir cómo vamos a buscar el espacio de hiperparámetros que definimos en el listado anterior. Debido a que el espacio de hiperparámetros es bastante grande, utilizaremos una búsqueda aleatoria en lugar de una búsqueda en rejilla.
- Recuerda que una búsqueda aleatoria no es exhaustiva (no probará todas las combinaciones de hiperparámetros), sino que seleccionará combinaciones aleatoriamente tantas veces (iteraciones) como le indiquemos. Usaremos 200 iteraciones.
- En el listado siguiente también definimos nuestra estrategia de validación cruzada para el ajuste. Aquí, vamos a utilizar una validación cruzada ordinaria de 5 veces.
- Recuerda que esto dividirá los datos en cinco pliegues (folds) y utilizará cada pliegue como conjunto de prueba una vez. Para cada conjunto de prueba, se entrenará un modelo con el resto de los datos (el conjunto de entrenamiento). Esto se realizará para cada combinación de valores de hiperparámetros probados por la búsqueda aleatoria.

Entrenamiento del modelo, VII

- **Nota**: Normalmente, si las clases están desbalanceadas, utilizaríamos un muestreo estratificado. Aquí, sin embargo, debido a que tenemos muy pocos casos en algunas de las clases, no hay suficientes casos para estratificar (pruébalo: obtendrás un error). Para este ejemplo, no estratificaremos; pero en situaciones en las que tienes muy pocos casos en una clase, debes considerar si hay datos suficientes para justificar mantener esa clase en el modelo.

```
> randSearch <- makeTuneControlRandom(maxit = 200)

> cvForTuning <- makeResampleDesc("CV", iters = 5)
```

- Finalmente, ¡realicemos nuestro ajuste de hiperparámetros!

```
> library(parallel)   # está ya instalado, es parte de la base de R

> library(parallelMap)

> parallelStartSocket(cpus = detectCores())

Starting parallelization in mode=socket with cpus=4.
```

Entrenamiento del modelo, VIII

```
> tunedTreePars <- tuneParams(tree, task = zooTask,
+  resampling = cvForTuning,
+  par.set = treeParamSpace,
+  control = randSearch)
[Tune] Started tuning learner classif.rpart for parameter set:
          Type len Def      Constr Req Tunable Trafo
minsplit  integer   -   -    5 to 20   -    TRUE    -
minbucket integer   -   -    3 to 10   -    TRUE    -
cp        numeric   -   - 0.01 to 0.1   -    TRUE    -
maxdepth  integer   -   -    3 to 10   -    TRUE    -
With control class: TuneControlRandom
Imputation value: 1
Exporting objects to slaves for mode socket: .mlr.slave.options
Mapping in parallel: mode = socket; level = mlr.tuneParams; cpus = 4; elements = 200.
[Tune] Result: minsplit=5; minbucket=3; cp=0.0246; maxdepth=5 : mmce.test.mean=0.0990476
> parallelStop()
Stopped parallelization. All cleaned up.
```

Entrenamiento del modelo, IX

```
> tunedTreePars

Tune result:

Op. pars: minsplit=5; minbucket=3; cp=0.0246; maxdepth=5

mmce.test.mean=0.0990476
```

- Para acelerar las cosas, primero comenzamos la paralelización ejecutando paraleloStartSocket(), estableciendo el número de CPU igual al número que tenemos disponibles.
- *Consejo*: Si deseas utilizar tus computadoras para otras cosas mientras se realiza el ajuste, es posible que desees establecer la cantidad de CPU utilizadas en menos que el máximo disponible para tí.
- Luego usamos la función tuneParams() para iniciar el proceso de ajuste. Los argumentos son los mismos que hemos usado anteriormente: el primero es el aprendizaje, el segundo es la tarea, el remuestreo es el método de validación cruzada, par.set es el espacio de hiperparámetros y el control es el método de búsqueda.

Entrenamiento del modelo, X

- Una vez que se completa, detenemos la paralelización e imprimimos nuestros resultados de ajuste.
- ADVERTENCIA Esto tarda unos 30 segundos en ejecutarse en mi máquina de cuatro núcleos.
- El algoritmo rpart no es tan costoso computacionalmente como el algoritmo de máquinas de soporte vectorial (SVM) que utilizamos para la clasificación a menudo.
- Por lo tanto, a pesar de ajustar cuatro hiperparámetros, el proceso de ajuste no lleva tanto tiempo (lo que significa que podemos realizar más iteraciones de búsqueda).

Entrenamiento con hiperparámetros ajustados, I

- Ahora que hemos ajustado nuestros hiperparámetros, podemos entrenar nuestro modelo final usándolos.
- Usamos la función setHyperPars() para crear un aprendizaje usando los hiperparámetros sintonizados, a los que accedemos usando tunedTreePars$x. Luego podemos entrenar el modelo final usando la función train(), como de costumbre.

```
> tunedTree <- setHyperPars(tree, par.vals = tunedTreePars$x)

> tunedTreeModel <- train(tunedTree, zooTask)
```

- Una de las cosas maravillosas de los árboles de decisión es lo interpretables que son. La forma más sencilla de interpretar el modelo es dibujar una representación gráfica del árbol.
- Hay varias formas de trazar modelos de árboles de decisión en R, pero mi favorita es la función rpart.plot() del paquete del mismo nombre. Primero instalemos el paquete rpart.plot y luego extraigamos los datos del modelo usando la función getLearnerModel().

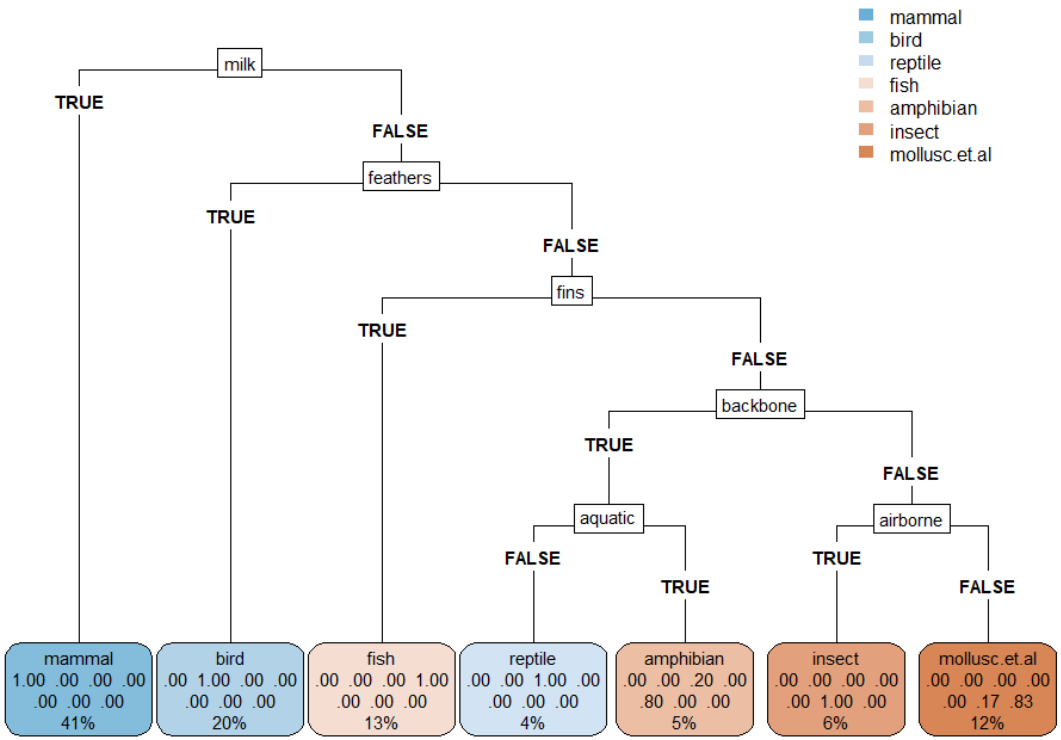
Entrenamiento con hiperparámetros ajustados, II

```
> install.packages("rpart.plot")

> library(rpart.plot)

> treeModelData <- getLearnerModel(tunedTreeModel)

> rpart.plot(treeModelData, roundint = FALSE,
+   box.palette = "BuBn",
+   type = 5)
```



Entrenamiento con hiperparámetros ajustados, III

- El primer argumento de la función rpart.plot() son los datos del modelo. Debido a que entrenamos este modelo usando mlr, la función nos advertirá que no puede encontrar los datos utilizados para entrenar el modelo.
- Podemos ignorar esta advertencia con seguridad, pero si te irrita tanto como a mí, puedes evitarlo proporcionando el argumento roundint =FALSE. La función también se quejará si tenemos más clases que su paleta de colores predeterminada (¡la función más necesaria jamás!).
- Ignora esto o solicita una paleta diferente estableciendo el argumento box.palette igual a una de las paletas predefinidas (ejecuta ?rpart.plot para obtener una lista de paletas disponibles).
- El argumento *type* cambia la forma en que se muestra el árbol.
- Me gusta bastante la simplicidad de la opción 5, pero consulta ?rpart.plot para experimentar con las otras opciones.

Entrenamiento con hiperparámetros ajustados, IV

- El gráfico generado por el listado anterior se muestra en el acetato 16. ¿Puedes ver lo simple e interpretable que es el árbol?
- Al predecir las clases de casos nuevos, comienzan en la parte superior (la raíz) y siguen las ramas según el criterio de división en cada nodo. El primer nodo pregunta si el animal produce leche o no.
- Se eligió esta división porque tiene la ganancia de Gini más alta de todas las divisiones candidatas (discrimina inmediatamente a los mamíferos, que constituyen el 41% del conjunto de entrenamiento de las otras clases).
- Los nodos hoja nos dicen qué clase está clasificada por ese nodo y las proporciones de cada clase en ese nodo. Por ejemplo, el nodo de la hoja que clasifica los casos como moluscos y otros contiene un 83 % de casos de moluscos y un 17 % de casos de insectos.
- El porcentaje en la parte inferior de cada hoja indica el porcentaje de casos en el conjunto de entrenamiento en esta hoja.

Entrenamiento con hiperparámetros ajustados, V

- Para inspeccionar los valores de cp para cada división, podemos usar la función printcp(). Esta función toma los datos del modelo como primer argumento y un argumento de dígitos opcional que especifica cuántos decimales imprimir en la salida.
- Hay información útil en la salida, como las variables realmente utilizadas para dividir los datos y el error del nodo raíz (el error antes de cualquier división). Finalmente, el resultado incluye una tabla de los valores de cp para cada división.

```
> printcp(treeModelData, digits = 3)

Classification tree:
rpart::rpart(formula = f, data = d, xval = 0, minsplit = 5, minbucket = 3,
    cp = 0.0245749159995466, maxdepth = 5)

Variables actually used in tree construction:
[1] airborne aquatic  backbone feathers fins     milk

Root node error: 60/101 = 0.594

n= 101

      CP nsplit rel error
1 0.3333      0     1.000
2 0.2167      1     0.667
3 0.1667      2     0.450
4 0.0917      3     0.283
5 0.0500      5     0.100
6 0.0246      6     0.050
```

Entrenamiento con hiperparámetros ajustados, VI

- Recuerda que en las secciones anteriores se mostró cómo se calculaban los valores de cp:

$$cp = \frac{p(\text{incorrecto}_{l+1}) - p(\text{incorrecto}_{l})}{n(splits_{l}) - n(splits_{l+1})}$$

- Para que puedas comprender mejor lo que significa el valor de cp, analicemos cómo se calcularon los valores de cp en la tabla del acetato 19.

- El valor de cp para la primera división es

$$cp = \frac{1.00 - 0.667}{1 - 0} = 0.333$$

- El valor de cp para la segunda división es

$$cp = \frac{0.667 - 0.450}{2 - 1} = 0.217$$

- y así. Si cualquier división candidata arrojaría un valor de cp inferior al umbral establecido mediante el ajuste, el nodo no se dividirá más.

Entrenamiento con hiperparámetros ajustados, VII

- *Sugerencia*: Para obtener un resumen detallado del modelo, ejecuta summary(treeModelData).
- El resultado es bastante largo (y se alarga cuanto más profundo sea el árbol), por lo que no lo imprimiremos aquí. Incluye la tabla cp, ordena los predictores según su importancia y muestra las divisiones primarias y sustitutas de cada nodo.

Validación cruzada del modelo, I

- En esta sección, validaremos de forma cruzada nuestro proceso de creación de modelos, incluido el ajuste de hiperparámetros. Ya hemos hecho esto varias veces, pero es tan importante que vamos a reiterarlo: debes incluir el preprocesamiento dependiente de los datos en tu validación cruzada.
- Esto incluye el ajuste de hiperparámetros que realizamos en el listado del acetato 11.
- Primero, definimos nuestra estrategia de validación cruzada externa. Esta vez estamos usando una validación cruzada quíntuple como nuestro ciclo externo de validación cruzada. Usaremos la descripción de remuestreo cvForTuning que hicimos anteriormente para el ciclo interno.
- A continuación, creamos nuestro contenedor “envolviendo” nuestro proceso de ajuste de hiperparámetros y de aprendizaje.
- Proporcionamos nuestra estrategia interna de validación cruzada, espacio de hiperparámetros y método de búsqueda a la función makeTuneWrapper().

Validación cruzada del modelo, II

- Finalmente, podemos iniciar la paralelización con la función parallelStartSocket() e iniciar el proceso de validación cruzada con la función resample().
- La función resample() toma como argumentos nuestro aprendizaje, la tarea y la estrategia de validación cruzada externa.
- *Advertencia*: esto lleva aproximadamente 2 minutos en mi máquina de cuatro núcleos.

```
> outer <- makeResampleDesc("CV", iters = 5)

> treeWrapper <- makeTuneWrapper("classif.rpart", resampling = cvForTuning,

+    par.set = treeParamSpace,

+    control = randSearch)

> parallelStartSocket(cpus = detectCores())

Starting parallelization in mode=socket with cpus=4.

> cvWithTuning <- resample(treeWrapper, zooTask, resampling = outer)
```

Validación cruzada del modelo, III

```
Exporting objects to slaves for mode socket: .mlr.slave.options

Resampling: cross-validation

Measures:             mmce

Mapping in parallel: mode = socket; level = mlr.resample; cpus = 4; elements = 5.

Aggregated Result: mmce.test.mean=0.0900000

> parallelStop()

Stopped parallelization. All cleaned up.
```

- Ahora veamos el resultado de la validación cruzada y veamos cómo se desempeñó nuestro proceso de creación de modelos.

```
> cvWithTuning

Resample Result

Task: zooTib

Learner: classif.rpart.tuned

Aggr perf: mmce.test.mean=0.0900000

Runtime: 32.3238
```

Validación cruzada del modelo, IV

- Hmm, eso es un poco decepcionante, ¿no? Durante el ajuste de hiperparámetros, la mejor combinación de hiperparámetros nos dio un error medio de clasificación errónea (MMCE) de 0.09 (probablemente obtendrás un valor diferente). Pero nuestra estimación con validación cruzada del rendimiento del modelo nos da un MMCE de 0.09. ¿Qué está sucediendo?
- Bueno, este es un ejemplo de sobreajuste. Nuestro modelo funciona mejor durante el ajuste de hiperparámetros que durante la validación cruzada. Este también es un buen ejemplo de por qué es importante incluir el ajuste de hiperparámetros dentro de nuestro procedimiento de validación cruzada.
- Acabamos de descubrir el principal problema del algoritmo rpart (y de los árboles de decisión en general): tienden a producir modelos sobreajustados. ¿Cómo superamos este problema?
- La respuesta es utilizar un método conjunto, un enfoque en el que utilizamos múltiples modelos para hacer predicciones para una sola tarea.

Fortalezas y debilidades

- Si bien a menudo no es fácil saber qué algoritmos funcionarán bien para una tarea determinada, aquí hay algunas fortalezas y debilidades que te ayudarán a decidir si los árboles de decisión funcionarán bien para tí.
- Los puntos fuertes de los algoritmos basados en árboles son los siguientes:
  - La intuición detrás de la construcción de árboles es bastante simple y cada árbol individual es muy interpretable.
  - Puede manejar variables predictoras categóricas y continuas.
  - No hace suposiciones sobre la distribución de las variables predictoras.
  - Puede manejar valores faltantes de manera sensible.
  - Puede manejar variables continuas en diferentes escalas.
- La debilidad de los algoritmos basados en árboles es la siguiente:
  - Los árboles individuales son muy susceptibles al sobreajuste, hasta el punto de que rara vez se utilizan.

Resumen

- El algoritmo rpart es un aprendizaje supervisado para problemas de clasificación y regresión.
- Los aprendizajes basados en árboles comienzan con todos los casos en el nodo raíz y encuentran divisiones binarias secuenciales hasta que los casos se encuentran en los nodos hoja.
- La construcción de árboles es un proceso voraz y puede limitarse estableciendo criterios de detención (como el número mínimo de casos requeridos en un nodo antes de que se pueda dividir).
- La ganancia de Gini es un criterio utilizado para decidir qué variable predictiva dará como resultado la mejor división en un nodo en particular.
- Los árboles de decisión tienden a sobre ajustarse al conjunto de entrenamiento.