uc3m | Universidad Carlos III de Madrid

OPENCOURSEWARE

APRENDIZAJE AUTOMÁTICO PARA EL ANÁLISIS DE DATOS

GRADO EN ESTADÍSTICA Y EMPRESA

Ricardo Aler

Tutorial MLR

Ricardo Aler

Julio de 2019

Introducción a mlR

mlR significa Machine Learning with R. Por un lado, simplifica el uso de las distintas librerías de aprendizaje automático (**C50**, **Cubist**, **part**, **knn**, etc.). Por otro, automatiza algunas tareas típicas de aprendizaje automático, tales como normalización, imputación, ajuste de hiper-parámetros, etc. Primero, vamos a cargar unos datos de un problema de regresión que nos servirán para hacer las pruebas. Los datos son **BostonHousing** (descritos aquí: [http://www.cs.toronto.edu/~delve/data/boston/bostonDetail.html]), cuyo objetivo es predecir el precio de la vivienda (situado en la última columna, **medv**) en función del resto de atributos de entrada (crime per cápita, habitaciones por vivienda, ...).

```
# Primero activamos los datos Boston Housing
library(mlbench)
data(BostonHousing)
head(BostonHousing)
```

```
##      crim zn indus chas   nox    rm  age    dis rad tax ptratio      b
## 1 0.00632 18  2.31    0 0.538 6.575 65.2 4.0900   1 296    15.3 396.90
## 2 0.02731  0  7.07    0 0.469 6.421 78.9 4.9671   2 242    17.8 396.90
## 3 0.02729  0  7.07    0 0.469 7.185 61.1 4.9671   2 242    17.8 392.83
## 4 0.03237  0  2.18    0 0.458 6.998 45.8 6.0622   3 222    18.7 394.63
## 5 0.06905  0  2.18    0 0.458 7.147 54.2 6.0622   3 222    18.7 396.90
## 6 0.02985  0  2.18    0 0.458 6.430 58.7 6.0622   3 222    18.7 394.12
##   lstat medv
## 1  4.98 24.0
## 2  9.14 21.6
## 3  4.03 34.7
## 4  2.94 33.4
## 5  5.33 36.2
## 6  5.21 28.7
```

El esquema básico de uso de **mlR** es el siguiente:

1. Se crea una "tarea" (task). La tarea puede ser de clasificación (**makeClassifTask**), de regresión (**makeRegrTask**), o de clustering (entre otras). Crear una tarea consiste en crear una variable que contendrá mis datos y el identificador de la variable **target**, o sea, la variable de respuesta o dependiente, en mi problema.

2. Se crea un "learner". Esto consiste en crear una variable que contiene qué algoritmo de aprendizaje se va a usar, y sus hiper-parámetros, en caso de que queramos usar valores de hiper-parámetros distintos a los valores por omisión (o por defecto).

3. Normalmente, no entrenaremos el modelo con todos los datos de la "task", sino que guardaremos algunos para entrenar el modelo y otros para evaluarlo. Una manera de hacerlo es guardar en una variable (**train_indices**) los índices de los datos que usaremos para entrenar (o sea, los números de fila de la matriz de datos). El resto de índices son los que usaremos para evaluar el modelo (**test_indices**).

4. Entrenamos (**train**) el "learner" con los datos contenidos en la "task", aunque no todos, sino sólo aquellos cuyos índices están en **train_indices**. Esto se indica con **subset**.

5. Hacemos predicciones del modelo para los datos de test con **predict**

6. Evaluamos el modelo con **performance**. Esto último me permite obtener el error sobre los datos de test.

```
# Primero activamos los datos Boston Housing
library(mlr) # Activamos la librería mlR. Si no está instalada, usad install.packages("mlR")
```

```
## Warning: package 'mlr' was built under R version 3.5.3
```

```
## Loading required package: ParamHelpers
```

```
## Warning: package 'ParamHelpers' was built under R version 3.5.3
```

```
# Primero, definimos una tarea de regresión, cuyos datos están contenidos en el data.frame BostonHousin
task_bh <- makeRegrTask(data= BostonHousing, target="medv")

# Segundo, definimos el algoritmo de aprendizaje ("learner"). Vamos a usar los árboles de regresión de
learner_rpart <- makeLearner("regr.rpart")

# Tercero, decidmos qué datos se usan para train y cuales para test (holdout)
# n es el número de líneas del data.frame. También valdría n <- nrow(BostonHousing)
n = getTaskSize(task_bh)
train_indices <- sample(1:n, round(2/3*n), replace = FALSE)
# setdiff = diferencia de conjuntos. setdiff(c(1,2,3,4), c(2,4)) == c(1,3)
test_indices <- setdiff(1:n, train_indices) # para test ponemos el resto de los índices

# Cuarto, entrenamos el modelo con los datos de train de la tarea
model_rpart <- train(learner_rpart, task_bh, subset = train_indices)

# Quinto, hacemos predicciones del modelo para los datos de test
preds_bh_rpart <- predict(model_rpart, task_bh, subset = test_indices)
```

Aquí debajo podemos ver las 10 primeras predicciones del modelo

```
# Las predicciones se almacenan en $data$response. Aquí podemos ver las 10  primeras
preds_bh_rpart$data$response[1:10]
```

```
##  [1] 21.20000 32.72000 17.57458 21.20000 17.57458 21.20000 25.92692
##  [8] 21.20000 17.57458 17.57458
```

y también están almacenados las salidas reales. Veamos las 10 primeras

```
# También podemos ver las salidas reales en $data$truth
preds_bh_rpart$data$truth[1:10]
```

```
##  [1] 21.6 36.2 18.9 18.9 21.7 19.9 23.1 19.6 16.6 12.7
```

Si las ponemos juntas (predicciones y valores reales), podemos ver si el modelo está aproximando bien. El error es la diferencia entre la predicción y el valor real. **numDato** es el identificador del dato (o sea, su número de final en el data.frame original).

```
comparacion <- data.frame(
  numDato = preds_bh_rpart$data$id,
  predicciones = preds_bh_rpart$data$response,
  reales = preds_bh_rpart$data$truth,
  error = abs(preds_bh_rpart$data$response - preds_bh_rpart$data$truth))
head(comparacion)
```

```
##   numDato predicciones reales    error
## 1       2     21.20000   21.6 0.400000
## 2       5     32.72000   36.2 3.480000
```

```
## 3      10     17.57458   18.9 1.325424
## 4      12     21.20000   18.9 2.300000
## 5      13     17.57458   21.7 4.125424
## 6      16     21.20000   19.9 1.300000
```

En sexto y último lugar, podemos pedirle a **mlR** que nos calcule alguna medida de error del modelo, teniendo en cuenta todos los datos de test. Podemos hacerlo con **performance**. En concreto, vamos a pedir el root mean squared error (raíz del error cuadrático medio) y el mean absolute error (error absoluto medio).

```
performance(preds_bh_rpart, measures = list(rmse, mae))
```

```
##     rmse      mae
## 4.355222 3.176910
```

A continuación vemos el código completo que hemos venido viendo. Nótese que usamos **set.seed(0)** en dos ocasiones. Lo que hace **set.seed** es fijar la semilla del generador de números aleatorios. Véase que se usa antes de **sample**, el cual genera una secuencia aleatoria de índices. Dado que es aleatoria, cada vez que ejecutemos **sample**, obtendremos una secuencia de índices distinta. Si queremos que siempre se genere la misma secuencia, tenemos que fijar la semilla al mismo número (cero, en este caso). Veamos un ejemplo:

```
"TRES SECUENCIAS DISTINTAS"
```

```
## [1] "TRES SECUENCIAS DISTINTAS"
```

```
(primera <- sample(1:5, 3, replace = FALSE))
```

```
## [1] 5 1 4
```

```
(segunda <- sample(1:5, 3, replace = FALSE))
```

```
## [1] 1 2 4
```

```
(tercera <- sample(1:5, 3, replace = FALSE))
```

```
## [1] 4 5 3
```

```
"TRES SECUENCIAS IGUALES"
```

```
## [1] "TRES SECUENCIAS IGUALES"
```

```
set.seed(0)
(primera <- sample(1:5, 3, replace = FALSE))
```

```
## [1] 5 2 4
```

```
set.seed(0)
(segunda <- sample(1:5, 3, replace = FALSE))
```

```
## [1] 5 2 4
```

```
set.seed(0)
(tercera <- sample(1:5, 3, replace = FALSE))
```

```
## [1] 5 2 4
```

Y ahora ya el código completo. Nótese que también usamos **set.seed(0)** antes de **train**, debido a que la construcción de un árbol tiene aspectos aleatorios (ej: si dos atributos tienen la misma entropía mientras se construye el árbol, se elige uno de ellos aleatoriamente). Esto no es necesario hacerlo en principio, pero es conveniente si queremos que cada vez que ejecutemos nuestro código obtengamos los mismos resultados (reproducibilidad de resultados).

```
# Primero activamos los datos Boston Housing
library(mlbench)
```

```r
data(BostonHousing)

# Primero activamos los datos Boston Housing
library(mlr) # Activamos la librería mlR. Si no está instalada, usad install.packages("mlR")

# Primero, definimos una tarea de regresión, cuyos datos están contenidos en el data.frame BostonHousin
task_bh <- makeRegrTask(data= BostonHousing, target="medv")

# Segundo, definimos el algoritmo de aprendizaje ("learner"). Vamos a usar los árboles de regresión de
learner_rpart <- makeLearner("regr.rpart")

# Tercero, decidmos qué datos se usan para train y cuales para test
# n es el número de líneas del data.frame. También valdría n <- nrow(BostonHousing)
n = getTaskSize(task_bh)
set.seed(0)
train_indices <- sample(1:n, round(2/3*n), replace = FALSE)
# setdiff = diferencia de conjuntos. setdiff(c(1,2,3,4), c(2,4)) == c(1,3)
test_indices <- setdiff(1:n, train_indices)

# Cuarto, entrenamos el modelo con los datos de train de la tarea
set.seed(0) # De esta manera, siempre se aprenderá el mismo modelo
model_rpart <- train(learner_rpart, task_bh, subset = train_indices)

# Quinto, hacemos predicciones del modelo para los datos de test
preds_bh_rpart <- predict(model_rpart, task_bh, subset = test_indices)

# Sexto, calculamos el error sobre el conjunto de test
performance(preds_bh_rpart, measures = list(rmse, mae))
```

```
##     rmse      mae
## 3.879949 2.781932
```

Ejercicio de uso básico de mlr

Usando la estructura que hemos visto hasta ahora, escribir código mlR para aplicar otras dos técnicas de aprendizaje en regresión: **Cubist** (árboles de modelos) y **knn** (vecino más cercano para regresión). En **mlR** reciben el nombre de **regr.cubist** y **regr.kknn** (aunque existen más librerías, como **regr.km** y **regr.fnn**, ambas de k-vecinos y regresión). En cualquier caso, tenéis la lista completa de métodos de aprendizaje automático integrados en **mlR** aquí: [https://mlr-org.github.io/mlr/articles/tutorial/integrated_learners.html]. También podéis usar la función **listLearners()** para ver todos los posibles métodos y sus propiedades. Por ejemplo, **listLearners("regr")** muestra todos los métodos disponibles para hacer regresión.

Nuestro objetivo es ver cual de los tres algoritmos: **regr.rpart**, **regr.cubist** o **regr.kknn** es el mejor en términos de MAE (el que tiene el menor MAE en test). Podéis hacer copy paste de código de las secciones anteriores.

Visualización de modelos

En ocasiones puede ser interesante visualizar el modelo construido. En el caso de Cubist, el modelo se puede ver e incluso interpretar. Se trata de reglas **if-then**, con condiciones en la parte del **if** y modelos lineales en la parte del **then**. Para cada condición, hay un modelo lineal a usar.

```r
learner_cubist <-  makeLearner("regr.cubist")
model_cubist <-  train(learner_cubist, task_bh, subset = train_indices)
summary(getLearnerModel(model_cubist))
```

```
##
## Call:
## cubist.default(x = d$data, y = d$target, control = ctrl)
##
##
## Cubist [Release 2.07 GPL Edition]  Thu Aug 01 16:56:05 2019
## ---------------------------------
##
##     Target attribute `outcome'
##
## Read 337 cases (14 attributes) from undefined.data
##
## Model:
##
##   Rule 1: [70 cases, mean 14.32, range 5 to 27.5, est err 2.20]
##
##     if
##  nox > 0.668
##     then
##  outcome = -5.45 + 3.05 dis + 23.6 nox - 0.35 lstat + 0.01 b
##
##   Rule 2: [136 cases, mean 19.50, range 7 to 31, est err 2.05]
##
##     if
##  nox <= 0.668
##  lstat > 9.59
##     then
##  outcome = 26.52 + 3.7 rm - 0.89 ptratio - 0.8 dis - 0.048 age
##            - 0.16 lstat - 6.7 nox - 0.0036 tax + 0.07 rad + 0.003 b
##
##   Rule 3: [119 cases, mean 28.27, range 11.9 to 50, est err 2.49]
##
##     if
##  nox <= 0.573
##  lstat <= 9.59
##     then
##  outcome = -7.58 + 9.1 rm - 0.61 lstat - 0.0214 tax + 0.24 rad
##            - 0.054 age - 0.62 ptratio + 0.21 crim - 0.47 dis - 1 nox
##
##   Rule 4: [16 cases, mean 39.97, range 21.9 to 50, est err 5.51]
##
##     if
##  nox > 0.573
##  lstat <= 9.59
##     then
##  outcome = 81.32 + 2.43 crim - 60 nox - 0.67 lstat - 0.0213 tax + 2.5 rm
##            - 0.62 ptratio
##
##
## Evaluation on training data (337 cases):
```

```
##
##     Average  |error|               2.13
##     Relative |error|               0.33
##     Correlation coefficient        0.95
##
##
##  Attribute usage:
##    Conds  Model
##
##    100%   100%    nox
##     79%   100%    lstat
##            95%    dis
##            79%    rm
##            79%    tax
##            79%    ptratio
##            75%    age
##            75%    rad
##            60%    b
##            40%    crim
##
##
## Time: 0.0 secs
```

Ejercicio de una tarea de clasificación y visualización del modelo

Antes hemos trabajado con un problema de regresión. Ahora probaremos con el dominio de clasificación que vimos en clase (¿es un buen día para jugar al tenis?). Se pide:

- Descargar los "datos del tenis" de Aula Global. Es un fichero .txt
- Leerlo en un data.frame con **read.csv**
- Crear una tarea de clasificación con **makeClassifTask**
- Construir CON TODOS LOS DATOS, un modelo de árboles de decisión con el algoritmo **C5.0** (es una versión mejorada del C4.5, que es el que vimos en clase para el ejemplo del tenis.). Tendréis que buscar en [https://mlr-org.github.io/mlr/articles/tutorial/integrated_learners.html] como se llama C5.0 en mlR
- Visualizar el árbol de decisión que se obtiene

Preproceso simple en mlR

En clases anteriores vimos cómo llevar a cabo ciertas transformaciones de los datos (preprocesado) usando **lapply**, entre ellas la normalización y la imputación. Con **mlR** el pre-proceso es incluso más sencillo, y podemos hacerlo así, antes de crear la tarea:

```
# Ejemplo de imputación de variables numéricas (enteras y reales)
# (nota: en este caso no se produce ningún efecto dado que BostonHousing no tiene NAs)
# consultar la documentación para otros ejemplos: ?impute
# bh_imputado es una lista. bh_imputado$data es quien contiene realmente el data.frame imputado
bh_imputado <- impute(BostonHousing, classes = list(numeric = imputeMean()))

# "range" normaliza al rango 0-1. target idenfitica la varible de respuesta,
# que normalmente, no queremos normalizar
bh_normalizado <- normalizeFeatures(bh_imputado$data, method="range", target="medv")
```

```
# Por último, creamos la tarea con los datos transformados
task_bh <- makeRegrTask(data= bh_normalizado, target="medv")
```

Evaluando modelos en mlR

En el apartado anterior vimos como evaluar los modelos sobre un conjunto de test, que estaba definido con una lista de índices. Sabemos que se puede evaluar un modelo de otras maneras. Por ejemplo, validación cruzada (crossvalidation). **mlR** permite automatizar también el proceso de evaluación con **makeResampleDesc** y **makeResampleInstance**. **makeResampleDesc** permite definir qué tipo de evaluación queremos (train/test o holdout, validación cruzada, etc.). **makeResampleInstance** crea un caso concreto con este tipo de evaluación. Recordemos que cuando usamos evaluación train/test, 2/3 de los datos originales van a parar al conjunto de train y 1/3 al de test. Pero esto se hace aleatoriamente. Es decir, los datos se pueden dividir de muchas maneras distintas en entrenamiento y test. **makeResampleDesc** crea una partición concreta (aleatoriamente) de los datos originales en entrenamiento y test. Si volvieramos a llamar a **makeResampleInstance**, se crearía otra partición concreta (pero distinta, al ser el reparto aleatorio) de los datos originales.

A continuación vamos a ver cómo se harían los pasos básicos que ya conocemos de los apartados anteriores, pero usando **makeResampleDesc** y **makeResampleInstance**. Lo único que cambia es el tercer paso (recordad que en apartados anteriores usabamos **sample**). Nótese que en **mlR**, el método de evaluación train/test se llama **holdout**. También cambian ligeramente los pasos cuarto (train) y quinto (performance), porque ahora los uniremos en un solo paso con **resample**. **resample** directamente entrena al modelo y lo evalúa con los datos de test (en lugar de tener que hacer las predicciones y evaluarlas después). Así, obtendremos directamente el error en test.

```
# Primero, definimos una tarea de regresión, cuyos datos están contenidos en el data.frame BostonHousin
task_bh <- makeRegrTask(data= BostonHousing, target="medv")

# Segundo, definimos el algoritmo de aprendizaje ("learner"). Vamos a usar los árboles de regresión de
learner_rpart <- makeLearner("regr.rpart")

# Tercero, creamos una partición de los datos en train y test para evaluar el modelo
# Este es el único paso distinto con respecto a lo que hemos visto
metodo_evaluacion <- makeResampleDesc("Holdout") # En este caso, también podríamos usar metodo_evaluaci
set.seed(0)
particion_datos <- makeResampleInstance(metodo_evaluacion, task_bh)

# Cuarto, entrenamos el modelo con los datos de train
set.seed(0)
errores_rpart <- resample(learner_rpart, task_bh, particion_datos, measures = list(rmse, mae))
```

```
## Resampling: holdout
## Measures:             rmse      mae
## [Resample] iter 1:    3.8799488 2.7819322
##
## Aggregated Result: rmse.test.rmse=3.8799488,mae.test.mean=2.7819322
##
```

```
# Podemos ver los errores del modelo sobre el conjunto de test aquí:
errores_rpart$aggr
```

```
## rmse.test.rmse  mae.test.mean
```

```
##       3.879949       2.781932
```

El código anterior es interesante, porque si queremos cambiar el método de evaluación de train/test (o holdout) a validación cruzada de 5 folds, basta que cambiemos ligeramente el tercer paso, como vemos a continuación (**CV** significa CrossValidation):

```
# Tercero, creamos una partición de los datos en train y test para evaluar el modelo
# Este es el único paso distinto con respecto a lo que hemos visto
metodo_evaluacion <- makeResampleDesc("CV", iters=5) # iters es el número de iteraciones o folds de la
set.seed(0)
particion_datos <- makeResampleInstance(metodo_evaluacion, task_bh)

# Cuarto, entrenamos el modelo con los datos de train
# Abajo en rojo se verá una serie de mensajes con el rmse y el mae para cada iteración o fold de crossv
set.seed(0)
errores_rpart <- resample(learner_rpart, task_bh, particion_datos, measures = list(rmse, mae))
```

```
## Resampling: cross-validation
## Measures:             rmse      mae
## [Resample] iter 1:    5.1697128 3.4852900
## [Resample] iter 2:    3.7125325 2.6243096
## [Resample] iter 3:    4.7276499 3.2211837
## [Resample] iter 4:    4.0275458 2.9916950
## [Resample] iter 5:    4.9970364 3.2370391
##
## Aggregated Result: rmse.test.rmse=4.5618198,mae.test.mean=3.1119035
##
```

```
# Al final, Podemos ver la media de los errores (la media de los cinco folds)
errores_rpart$aggr
```

```
## rmse.test.rmse  mae.test.mean
##       4.561820       3.111903
```

Ejercicio

Vamos a aplicar validación cruzada de 10 folds al problema de clasficación de iris con C50. Escribid el código necesario, teniendo en cuenta que la medida de error típica en clasificación es "misclassification rate", o "mmce", que es 1-tasa de aciertos. Tened cuidado de llamar a las variables **metodo_evaluacion_iris**, **particion_datos_iris**, para que no machaquen a los valores que ya tenían para el problema de BostonHousing.

El dataset de clasificación de plantas **iris** está accesible en R en la variable **iris**. La variable de respuesta es "Species" (la especie de la planta)

Cambiando los hiper-parámetros de los algoritmos de aprendizaje

Sabemos que el comportamiento de los distintos algoritmos de aprendizaje depende mucho de los valores que le demos a sus hiper-parámetros. De momento, hemos usado los hiper-parámetros por omisión (por defecto). Para saber qué hiper-parámetros tiene un método de aprendizaje, podemos usar **getParamSet** así:

```
getParamSet("regr.kknn")
```

```
## Loading required package: kknn
```

```
##              Type len     Def                                    Constr Req
## k         integer   -       7                                 1 to Inf   -
## distance  numeric   -       2                                 0 to Inf   -
## kernel   discrete   - optimal rectangular,triangular,epanechnikov,b...   -
## scale     logical   -    TRUE                                         -   -
##          Tunable Trafo
## k           TRUE     -
## distance    TRUE     -
## kernel      TRUE     -
## scale       TRUE     -
```

Por ejemplo, en k-vecinos, el valor por omisión de k es 1. ¿Cómo cambiarlo?. Debajo se ve el código necesario. Lo único que cambia es el segundo paso. Como el resto de pasos ya los hemos ejecutado más arriba, y no cambian, ya no los pondremos en el siguiente código. Es decir, la tarea sigue siendo la de Boston Housing, y el método de evaluación sigue siendo la validación cruzada con 5 folds.

```
# Segundo, definimos el algoritmo de aprendizaje ("learner"). Vamos a usar rpart con minsplit = 40 y kk
learner_rpart <- makeLearner("regr.rpart", minsplit = 40)
learner_kknn <- makeLearner("regr.kknn", k=3)

# Cuarto, entrenamos el modelo con los datos de train
set.seed(0)
errores_rpart <-  resample(learner_rpart, task_bh, particion_datos, measures = list(rmse, mae))
```

```
## Resampling: cross-validation
## Measures:             rmse      mae
## [Resample] iter 1:    5.1231778 3.2888608
## [Resample] iter 2:    3.8470897 2.8271194
## [Resample] iter 3:    5.4614753 3.7536486
## [Resample] iter 4:    3.8972777 2.8002465
## [Resample] iter 5:    5.3560460 3.3903913
##
## Aggregated Result: rmse.test.rmse=4.7906318,mae.test.mean=3.2120533
##
```

```
set.seed(0)
errores_kknn <- resample(learner_kknn, task_bh, particion_datos, measures = list(rmse, mae))
```

```
## Resampling: cross-validation
## Measures:             rmse      mae
## [Resample] iter 1:    5.2456740 2.9580920
## [Resample] iter 2:    3.1483036 2.2693337
## [Resample] iter 3:    5.6137005 3.1508917
## [Resample] iter 4:    3.3906593 2.3576125
## [Resample] iter 5:    3.9731050 2.2226252
```

```
##
## Aggregated Result: rmse.test.rmse=4.3869050,mae.test.mean=2.5917110
##
```

```
# Podemos ver los errores del modelo sobre el conjunto de test aquí:
errores_rpart$aggr
```

```
## rmse.test.rmse  mae.test.mean
##       4.790632       3.212053
```

```
errores_kknn$aggr
```

```
## rmse.test.rmse  mae.test.mean
##       4.386905       2.591711
```

Ejercicio

En la tarea de BostonHousing, probad distintos valores de k (k=1, k=2, k=3, k=4, por ejemplo), entrenad un modelo con cada uno, y evaluadlo con el conjunto de test (usad. ¿Cuál funciona mejor (en test)? ¿Podéis automatizar el proceso con un bucle? ¿Seriais capaces de automatizarlo con **lapply** en lugar de un bucle?

Ejercicio

En el conjunto de datos **airquality** (accesible mediante data(airquality)), vamos a predecir la cantidad de **Ozone** en el aire, teniendo en cuenta la radiación Solar (**Solar.R**), la temperatura (**Temp**) y el viento (**Wind**).

Se pide:

- Eliminar del data.frame las columnas **Month** y **Day**
- Eliminar las filas que contengan **NA** en la variable **Ozone** (la variable de respuesta no puede contener NA). Ojo, aquí no podemos usar **complete.cases**, porque elimina todas las filas que contienen algún NA en cualquier atributo. En este caso, sólo queremos eliminar las filas que tengan NAs en la variable Ozone. Se puede hacer así, suponiendo que **aq** contiene el data.frame airquality: aq <- aq[!is.na(aq$Ozone),]
- Usad **summary** para comprobar si algún otro atributo todavía tiene NAs (vereis que **Solar.R** si que tiene NAs)
- Cread una tarea mlR para predecir la variable **Ozone**. Llamad a la tarea **task_aq**
- Usad mlR para imputar las variables que tengan NAs (en este caso, ya sabemos que **Solar.R** los tiene)
- Usad validación cruzada de 10 folds con rpart para estimar el MAE. Llamad a las variables **metodo_evaluacion_aq** y **particion_datos_aq** para que los nombres de las variables no machaquen a las que hemos definido en apartados anteriores.

Ejercicio: ajuste de k (de KNN) a mano

Vamos a intentar determinar el mejor valor de **k** para el método de k-vecinos. Aunque más adelante veremos como **mlR** puede automatizar este proceso, en este ejercicio lo vamos a hacer a mano, para practicar y entender conceptos. Lo haremos de esta manera: los datos disponibles los dividiremos aleatoriamente en **train_indices** y **test_indices**, con **sample**, como ya sabemos hacer (2/3 para train y 1/3 para test). Y después, los **train_indices** los volvemos a dividir en otros dos conjuntos: **train_train_indices** (2/3) y **validation_indices** (1/3), también con **sample**. Habrá que usar también **setdiff**.

Para ello seguiremos los siguientes pasos:

- Definimos **kknn** como algoritmo de aprendizaje, con **makeLearner**

- Creamos dos variables **train_indices** y otra **test_indices**, que contenga 2/3 de los datos para entrenar y 1/3 para evaluar. Tenemos que usar **sample** y **setdiff** como en apartados anteriores.
- Ahora separamos aleatoriamente los índices de **train_indices** en otros dos conjuntos: **train_train_indices** (2/3) y **validation_indices** (1/3). Usamos también **sample** y **setdiff**.
- Entrenamos un modelo knn con **train_train_indices** y k=1, y lo evaluamos con **validation_indices**. Para entrenar usaremos la función **train**. Recordar que habrá que usar **subset** para decirle que queremos usar los **train_train_indices** para entrenar. Para evaluar, tendremos que obtener las predicciones con **predict** y computar el error con **performance**.
- Hacemos lo mismo que en el punto anterior, pero con k=2
- Hacemos lo mismo que en el punto anterior, pero con k=3
- ¿Cuál de los tres errores obtenidos sobre el conjunto de validación es menor? Apuntamos el k para el que el error sea el menor.
- Ahora construimos un modelo usando todos los datos de entrenamiento (**train_indices**) y el k óptimo anterior
- Y dicho modelo lo evaluamos con los datos de test (**test_indices**)

k = 3 parecer ser el óptimo.

Ajuste óptimo de hiper-parámetros con mlR

Vamos a ajustar de manera óptima algunos de los parámetros de **regr.rpart**, en concreto **minsplit** (cuantos datos necesita una hoja para continuar la subdivisión del árbol) y **maxdepth** (máxima profundidad del árbol). Para ello, el primer paso es definir los posibles valores que pueden tomar los hiper-parámetros. Para **minsplit** probaremos con 10 y 20 datos. Para **maxdepth** probaremos con profundidades de 2, 4, y 6.

```
# simple mini grid
ps = makeParamSet(
  makeDiscreteParam("minsplit", values = c(10, 20)),
  makeDiscreteParam("maxdepth", values = c(2, 4, 6))
)
```

Lo segundo que tenemos que hacer es definir una estrategia de búsqueda de hiper-parámetros. La más sencilla de entender, pero también la más costosa en tiempo es "gridsearch", la cual prueba todas las posibles combinaciones de hiper-parámetros. En nuestro caso, probará con **minsplit==10** y **maxdepth==2**, con **minsplit==10** y **maxdepth==4**, y así con todas las posibles combinaciones de valores de ambos hiper-parámetros.

```
control_grid <- makeTuneControlGrid()
```

Para cada combinación de valores de hiper-parámetros, se entrenará un modelo y se evaluará. Tenemos que definir por tanto como la "gridsearch" va a evaluar los modelos. Podemos usar una estrategia sencilla y rápida como **Holdout**.

```
evaluacion_grid <- makeResampleDesc("Holdout")
```

A continuación vamos a definir una "secuencia de métodos" (Wrapped learner), mediante **makeTuneWrapper** que hace la siguiente secuencia:

1. Entrena todos los modelos necesarios para evaluar a cada posible combinación de hiper-parámetros. En nuestro caso, como hay tres posibles valores para **maxdepth** y dos para **minsplit**, se entrenarán 6 modelos distintos.
2. Evalúa todos esos modelos según el método elegido en **evaluacion_grid**. Quédate con la mejor combinación de hiper-parámetros.

3. Construye el modelo final con esos hiper-parámetros óptimos y todos los datos.

```
# Aquí estamos construyendo un nuevo "learner", el cual es una secuencia de ajuste de hiper-parámetros y
# learner_rpart: es el algoritmo de aprendizaje que queremos usar
# evaluacion_grid: es la manera en la que vamos a evaluar los distintos hiper-parámetros
# par.set: define todas las posibles combinaciones de hiper-parámetros
# control: define que vamos a usar la estrategia de grid-search, o sea, vamos a probar con todas las co

learner_ajuste_rpart <- makeTuneWrapper(learner_rpart, resampling = evaluacion_grid, par.set = ps, cont
```

Pero además de evaluar varios modelos para elegir el mejor conjunto de hiper-parámetros, también queremos evaluar el modelo final construido con los mejores hiper-parámetros. Esta idea es complicada de entender y es conveniente haber comprendido la clase de teoría relacionada con este tema. Pero en el fondo, lo que estamos haciendo es construir y evaluar un modelo, como hicimos en apartados anteriores. La única diferencia es que usamos **learner_ajuste_rpart** como algoritmo de aprendizaje, en lugar de usar **learner_rpart** (esto último es lo que hicimos en apartados anteriores).

Se haría así:

```
metodo_evaluacion <- makeResampleDesc("CV", iters=2)
particion_datos <- makeResampleInstance(metodo_evaluacion, task_bh)

# Cuarto, entrenamos el modelo con los datos de train
errores_ajuste_rpart <- resample(learner_ajuste_rpart, task_bh, particion_datos, measures = list(rmse, m
```

```
## Resampling: cross-validation

## Measures:             rmse      mae

## [Tune] Started tuning learner regr.rpart for parameter set:

##              Type len Def Constr Req Tunable Trafo
## minsplit discrete   -   -  10,20   -    TRUE     -
## maxdepth discrete   -   -  2,4,6   -    TRUE     -

## With control class: TuneControlGrid

## Imputation value: Inf

## [Tune-x] 1: minsplit=10; maxdepth=2

## [Tune-y] 1: mae.test.mean=3.9592317; time: 0.0 min

## [Tune-x] 2: minsplit=20; maxdepth=2

## [Tune-y] 2: mae.test.mean=3.7375978; time: 0.0 min

## [Tune-x] 3: minsplit=10; maxdepth=4

## [Tune-y] 3: mae.test.mean=3.1859448; time: 0.0 min

## [Tune-x] 4: minsplit=20; maxdepth=4

## [Tune-y] 4: mae.test.mean=3.2092128; time: 0.0 min

## [Tune-x] 5: minsplit=10; maxdepth=6

## [Tune-y] 5: mae.test.mean=3.1859448; time: 0.0 min

## [Tune-x] 6: minsplit=20; maxdepth=6

## [Tune-y] 6: mae.test.mean=3.2092128; time: 0.0 min

## [Tune] Result: minsplit=10; maxdepth=4 : mae.test.mean=3.1859448
```

```
## [Resample] iter 1:    4.8066210 3.0071876

## [Tune] Started tuning learner regr.rpart for parameter set:

##              Type len Def Constr Req Tunable Trafo
## minsplit discrete   -   -  10,20   -    TRUE     -
## maxdepth discrete   -   -  2,4,6   -    TRUE     -

## With control class: TuneControlGrid

## Imputation value: Inf

## [Tune-x] 1: minsplit=10; maxdepth=2

## [Tune-y] 1: mae.test.mean=4.2747225; time: 0.0 min

## [Tune-x] 2: minsplit=20; maxdepth=2

## [Tune-y] 2: mae.test.mean=4.2747225; time: 0.0 min

## [Tune-x] 3: minsplit=10; maxdepth=4

## [Tune-y] 3: mae.test.mean=3.1057538; time: 0.0 min

## [Tune-x] 4: minsplit=20; maxdepth=4

## [Tune-y] 4: mae.test.mean=3.0262408; time: 0.0 min

## [Tune-x] 5: minsplit=10; maxdepth=6

## [Tune-y] 5: mae.test.mean=2.8499666; time: 0.0 min

## [Tune-x] 6: minsplit=20; maxdepth=6

## [Tune-y] 6: mae.test.mean=2.7704535; time: 0.0 min

## [Tune] Result: minsplit=20; maxdepth=6 : mae.test.mean=2.7704535

## [Resample] iter 2:    4.5418554 3.4421668

##

## Aggregated Result: rmse.test.rmse=4.6761125,mae.test.mean=3.2246772

##
```

```
# Podemos ver los errores del modelo sobre el conjunto de test aquí:
errores_ajuste_rpart$aggr
```

```
## rmse.test.rmse  mae.test.mean
##       4.676112       3.224677
```

Y aquí podemos ver la mejor combinación de hiper-parámetros que se encontró. Nótese que como nuestro método de evaluación es validación cruzada con dos folds, el ajuste de parámetros se ha llevado a cabo dos veces, una en cada fold. Y para cada fold sale un conjunto de valores de hiper-parámetros distinto (de manera similar a como en cada fold sale un modelo distinto).

```
errores_ajuste_rpart$extract
```

```
## [[1]]
## Tune result:
## Op. pars: minsplit=10; maxdepth=4
## mae.test.mean=3.1859448
##
## [[2]]
## Tune result:
```

```
## Op. pars: minsplit=20; maxdepth=6
## mae.test.mean=2.7704535
```

Y aquí podemos ver la media (haciendo la media de los dos errores anteriores):

```
errores_ajuste_rpart$aggr
```

```
## rmse.test.rmse  mae.test.mean
##       4.676112       3.224677
```

Una vez que tenemos la evaluación, podemos construir el modelo final con **train** con todos los datos. Así, tendremos nuestro modelo final, y su estimación de comportamiento futuro, que es lo que obtuvimos antes con **errores_ajuste_rpart$aggr**.

```
final_model_ajuste_rpart <- train(learner_ajuste_rpart, task_bh)
```

```
## [Tune] Started tuning learner regr.rpart for parameter set:
```

```
##              Type len Def Constr Req Tunable Trafo
## minsplit discrete   -   -  10,20   -    TRUE     -
## maxdepth discrete   -   -  2,4,6   -    TRUE     -
```

```
## With control class: TuneControlGrid
```

```
## Imputation value: Inf
```

```
## [Tune-x] 1: minsplit=10; maxdepth=2
```

```
## [Tune-y] 1: mae.test.mean=4.2080072; time: 0.0 min
```

```
## [Tune-x] 2: minsplit=20; maxdepth=2
```

```
## [Tune-y] 2: mae.test.mean=4.2080072; time: 0.0 min
```

```
## [Tune-x] 3: minsplit=10; maxdepth=4
```

```
## [Tune-y] 3: mae.test.mean=3.1929958; time: 0.0 min
```

```
## [Tune-x] 4: minsplit=20; maxdepth=4
```

```
## [Tune-y] 4: mae.test.mean=3.3848530; time: 0.0 min
```

```
## [Tune-x] 5: minsplit=10; maxdepth=6
```

```
## [Tune-y] 5: mae.test.mean=3.1100675; time: 0.0 min
```

```
## [Tune-x] 6: minsplit=20; maxdepth=6
```

```
## [Tune-y] 6: mae.test.mean=3.2830200; time: 0.0 min
```

```
## [Tune] Result: minsplit=10; maxdepth=6 : mae.test.mean=3.1100675
```