Machine Learning con R y caret

Joaquín Amat Rodrigo

Abril, 2018

Tabla de contenidos

Versión PDF: Github

Introducción

Durante los últimos años, el interés y la aplicación de *machine learning* ha experimentado tal expansión, que se ha convertido en una disciplina aplicada en prácticamente todos los ámbitos de investigación académica e industrial. El creciente número de personas dedicadas a esta disciplina ha dado como resultado todo un repertorio de herramientas con las que, perfiles con especialización media, consiguen acceder a métodos predictivos potentes. El lenguaje de programación R es un ejemplo de ello.

El término *machine learning* engloba al conjunto de algoritmos que permiten identificar patrones presentes en los datos y crear con ellos estructuras (modelos) que los representan. Una vez que los modelos han sido generados, se pueden emplear para predecir información sobre hechos o eventos que todavía no se han observado. Es importante recordar que los sistemas de *machine learning* solo son capaces de memorizar patrones que estén presentes en los datos con los que se entrenan, por lo tanto, solo pueden reconocer lo que han visto antes. Al emplear sistemas entrenados con datos pasados para predecir futuros se está asumiendo que, en el futuro, el comportamiento será el mismo, cosa que no siempre ocurre.

Aunque con frecuencia, términos como *machine learning*, *data mining*, inteligencia artificial, *data science*... son utilizados como sinónimos, es importante destacar que los métodos de *machine learning* son solo una parte de las muchas estrategias que se necesita combinar para extraer información, entender y dar valor a los datos. El siguiente documento pretende ser un ejemplo del tipo de problema al que se suele enfrentar un analista: partiendo de un conjunto de datos más o menos procesado (la preparación de los datos es una etapa crítica que precede al *machine learning*), se desea crear un modelo que permita predecir con éxito el comportamiento o valor que toman nuevas observaciones.

A diferencia de otros documentos, este pretende ser un ejemplo práctico con menos desarrollo teórico. El lector podrá darse cuenta de lo sencillo que es aplicar un gran abanico de métodos predictivos con R y sus librerías. Sin embargo, es crucial que cualquier analista entienda los fundamentos teóricos en los que se basa cada uno de ellos para que un proyecto de este tipo tenga éxito. Aunque aquí solo se describan brevemente, estarán acompañados de links donde encontrar información detallada.

Etapas de un problema de *machine learning*

El siguiente es un listado de las etapas que suelen formar parte de la mayoría de problemas de *machine learning*.

- Definir el problema: ¿Qué se pretende predecir? ¿De qué datos se dispone? ¿Qué datos es necesario conseguir?
- Explorar y entender los datos que se van a emplear para crear el modelo.
- Métrica de existo: definir una forma apropiada de cuantificar cómo de buenos son los resultados obtenidos.
- Preparar la estrategia para evaluar el modelo: separar las observaciones en un conjunto de entrenamiento, un conjunto de validación (este último suele ser un subconjunto del de entrenamiento) y un conjunto de test. Ninguna información del conjunto de test debe participar en el proceso de entrenamiento del modelo.
- Preprocesar los datos: aplicar las transformaciones necesarias para que los datos puedan ser interpretados por el algoritmo de *machine learning* seleccionado.
- Ajustar un primer modelo capaz de superar unos resultados mínimos. Por ejemplo, en problemas de clasificación, el mínimo a superar es el porcentaje de la clase mayoritaria (la moda).
- Gradualmente, mejorar el modelo optimizando sus hiperparámetros.
- Evaluar la capacidad del modelo final con el conjunto de test para tener una estimación de la capacidad que tiene el modelo cuando predice nuevas observaciones.

Paquete caret

R es uno de los lenguajes de programación que domina dentro del ámbito de la estadística, *data mining* y *machine learning*. Al tratarse de un software libre, innumerables usuarios han podido implementar sus algoritmos, dando lugar a un número muy elevado de paquetes/librerías donde encontrar prácticamente todas las técnicas de *machine learning* existentes. Sin embargo, esto tiene un lado negativo, cada paquete tiene una sintaxis, estructura e implementación propia, lo que dificulta su aprendizaje. El paquete caret, desarrollado por *Max Kuhn*, es una interfaz que unifica bajo un único marco cientos de funciones de distintos paquetes, facilitando en gran medida todo el proceso de preprocesado, entrenamiento, optimización y validación de modelos predictivos. Existen otros proyectos similares y muy prometedores como mlr pero, para este ejemplo, se emplea únicamente caret.

El paquete caret ofrece tal cantidad de posibilidades que, difícilmente, pueden ser mostradas con un único ejemplo. En este documento, se emplean solo algunas de sus funcionalidades. Si

en algún caso se requiere una explicación detallada, para que no interfiera con la narrativa del análisis, se añadirá un anexo. Aun así, para conocer bien todas las funcionalidades de caret se recomienda leer su documentación.

```
# Instalación de los paquetes que unifica caret. Esta instalación puede tardar.
# Solo es necesario ejecutarla si no se dispone de los paquetes.
install.packages("caret", dependencies = c("Depends", "Suggests"))

library(caret)
```

Dataset Titanic

A poco que el lector haya buscado ejemplos sobre *machine learning* o análisis predictivo, seguro que ha oído hablar del dataset Titanic disponible en la plataforma Kaggle. Este set de datos contiene información sobre los pasajeros del RMS Titanic, el transatlántico británico que se hundió en abril de 1912 durante su viaje inaugural desde Southampton a Nueva York. Entre la información almacenada se encuentran la edad, género, características socio-económicas de los pasajeros y si sobrevivieron o no al naufragio. Aunque pueda resultar un clásico poco original, estos datos tienen una serie de características que los hacen idóneos para ser utilizados como ejemplo introductorio al *machine learning*:

- Contiene suficientes observaciones para entrenar modelos que consigan un poder predictivo alto.
- Incluye tanto variables continuas como cualitativas, lo que permite mostrar diferentes análisis exploratorios.
- La variable respuesta es binaria. Aunque la mayoría de los algoritmos de clasificación mostrados en este capítulo se generalizan para múltiples clases, su interpretación, suele ser más sencilla cuando solo hay dos.
- Contiene valores ausentes. La forma en que se manejan estos registros (eliminación o imputación) influye en gran medida en el modelo final.
- Aunque a primera vista no lo parezca, las variables contienen información adicional que puede ser extraída mediante *feature engineering*.
- Requiere ciertos pasos de limpieza y conversión de datos.

y lo más importante de todo, se trata de un problema y de unos datos cuyas variables pueden entenderse de forma sencilla. Es intuitivo comprender el impacto que puede tener la edad, el sexo, la localización del camarote... en la supervivencia de los pasajeros. Aunque no lo parezca, comprender a fondo el problema que se pretende modelar es lo más importante para lograr buenos resultados.

En el paquete titanic se pueden encontrar todos los datos proporcionados por *kaggle*. Están divididos en dos partes, un conjunto de entrenamiento y un conjunto de test. En los datos de test, se desconoce si el pasajero sobrevivió o no, esto es así porque la plataforma *Kaggle* evalúa con ellos la capacidad predictiva de los modelos presentados, es decir, el competidor crea un modelo con los datos de entrenamiento, predice la variable respuesta (superviviente o no superviviente) en el conjunto de test y envía sus predicciones a la plataforma. La plataforma contrasta las predicciones con los verdaderos resultados y devuelve una puntuación. En este ejemplo, para evitar tener que estar enviado los resultados de cada modelo, se emplea únicamente el conjunto de entrenamiento. Además, esto refleja mejor lo que ocurre en la práctica cuando un analista se enfrenta a un nuevo set de datos.

```
library(tidyverse)
library(titanic)
datos <- titanic_train
```

Análisis exploratorio de los datos

Antes de entrenar un modelo predictivo, o incluso antes de realizar cualquier cálculo con un nuevo conjunto de datos, es muy importante realizar una exploración descriptiva de los mismos. Este proceso permite entender mejor que información contiene cada variable, así como detectar posibles errores. Algunos ejemplos frecuentes son:

- Que una columna se haya almacenado con el tipo incorrecto: una variable numérica está siendo reconocida como texto.
- Que una variable contenga valores que no tienen sentido: para indicar que no se dispone de la altura de una persona se introduce el valor cero o un espacio en blanco. No existe nadie cuya altura sea cero.
- Que en una variable de tipo numérico se haya introducido una palabra en lugar de un número.

Además, puede dar pistas sobre qué variables no son adecuadas como predictores en un modelo (más sobre esto en los siguientes apartados).

Acorde a la información facilitada por *Kaggle*, las variables disponibles son:

- PassengerId: identificador único del pasajero.
- Survived: si el pasajero sobrevivió al naufragio, codificada como 0 (no) y 1 (si). Esta es la variable respuesta que interesa predecir.
- Pclass: clase a la que pertenecía el pasajero: 1, 2 o 3.

- Name: nombre del pasajero.
- Sex: sexo del pasajero.
- Age: edad del pasajero.
- SibSp: número de hermanos, hermanas, hermanastros o hermanastras en el barco.
- Parch: número de padres e hijos en el barco.
- Ticket: identificador del billete.
- Fare: precio pagado por el billete.
- Cabin: identificador del camarote asignado al pasajero.
- Embarked: puerto en el que embarcó el pasajero.

Tipo de variables

Una de las primeras comprobaciones que hay que hacer tras cargar los datos, es verificar que cada variable se ha almacenado con el tipo de valor que le corresponde, es decir, que las variables numéricas sean números y las cualitativas *factor*, *character* o *booleanas*. En el lenguaje de programación R, cuando la variable es cualitativa, conviene almacenarla con el tipo *factor*.

```
# Resumen del set de datos
glimpse(datos)
```

```
## Observations: 891
## Variables: 12
## $ PassengerId <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,...
## $ Survived    <int> 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0,...
## $ Pclass      <int> 3, 1, 3, 1, 3, 3, 1, 3, 3, 2, 3, 1, 3, 3, 3, 2, 3,...
## $ Name        <chr> "Braund, Mr. Owen Harris", "Cumings, Mrs. John Bra...
## $ Sex         <chr> "male", "female", "female", "female", "male", "mal...
## $ Age         <dbl> 22, 38, 26, 35, 35, NA, 54, 2, 27, 14, 4, 58, 20, ...
## $ SibSp       <int> 1, 1, 0, 1, 0, 0, 0, 3, 0, 1, 1, 0, 0, 1, 0, 0, 4,...
## $ Parch       <int> 0, 0, 0, 0, 0, 0, 0, 1, 2, 0, 1, 0, 0, 5, 0, 0, 1,...
## $ Ticket      <chr> "A/5 21171", "PC 17599", "STON/O2. 3101282", "1138...
## $ Fare        <dbl> 7.2500, 71.2833, 7.9250, 53.1000, 8.0500, 8.4583, ...
## $ Cabin       <chr> "", "C85", "", "C123", "", "", "E46", "", "", "", ...
## $ Embarked    <chr> "S", "C", "S", "S", "S", "Q", "S", "S", "S", "C", ...
```

En la tabla anterior puede verse el tipo de cada variable, así como un ejemplo de los valores que toma. A priori, el único caso en el que el tipo de valor no se corresponde con la naturaleza de la variable es *Survived*. Aunque esta variable está codificada como 1 si el pasajero sobrevivió y 0 si murió, no conviene almacenarla en formato numérico, ya que esto puede llevar a errores como el de tratar de calcular su media. Para evitar este tipo de problemas, se recodifica la variable para que sus dos posibles niveles sean "Si"-"No" y se convierte a *factor*.

```
datos$Survived <- if_else(datos$Survived == 1, "Si", "No")
datos$Survived <- as.factor(datos$Survived)
```

La variable *Pclass* es cualitativa ordinal, es decir, toma distintos valores cualitativos ordenados siguiendo una escala establecida, aunque no es necesario que el intervalo entre mediciones sea uniforme. Por ejemplo, se asume que la diferencia entre primera y segunda clase es menor que la diferencia entre primera y tercera, sin embargo, las diferencias entre primera-segunda y segunda-tercera no tiene por qué ser iguales. Dado que las propiedades matemáticas no se cumplen $(2 - 1 \neq 3 - 2)$ es preferible no almacenarlas como números.

```
datos$Pclass <- as.factor(datos$Pclass)
```

Las variables *SibSp* y *Parch* son cuantitativas discretas, pueden tomar únicamente determinados valores numéricos. En este caso, al tratarse de número de personas (familiares e hijos), solo pueden ser números enteros. No existe una norma clara sobre como almacenar estas variables. Para este estudio exploratorio, dado que solo toman unos pocos valores, se decide almacenarlas como *factor*.

```
datos$SibSp <- as.factor(datos$SibSp)
datos$Parch <- as.factor(datos$Parch)
```

Las variables *Sex* y *Embarked* también se convierten a tipo *factor*.

```
datos$Sex      <- as.factor(datos$Sex)
datos$Embarked <- as.factor(datos$Embarked)
```

El análisis exploratorio de variables suele caracterizarse por el cálculo de sumatorios, reestructuración de los datos y representaciones gráficas. Los paquetes dplyr, tidyr, ggplot... englobados dentro de la filosofía tydiverse facilitan en gran medida todos estos pasos. Muchos de los cálculos y representaciones que se realizan a lo largo de este capítulo se consiguen de forma más rápida si los datos están almacenados siguiendo la estructura: *observación, variable, valor*. Como el set de datos no es lo suficientemente grande como para dar problemas de memoria, se crea un segundo *dataframe* con esta estructura.

```
datos_long <- datos %>% gather(key = "variable", value = "valor", -PassengerId)
head(datos_long)
```

```
##   PassengerId variable valor
## 1           1 Survived    No
## 2           2 Survived    Si
## 3           3 Survived    Si
## 4           4 Survived    Si
## 5           5 Survived    No
## 6           6 Survived    No
```

Número de observaciones y valores ausentes

Junto con el estudio del tipo de variables, es básico conocer el número de observaciones disponibles y si todas ellas están completas, es decir, verificar si para cada observación se ha registrado el valor de cada una de las variables.

```
# Número de observaciones del set de datos
nrow(datos)
```

```
## [1] 891
```

```
# Detección si hay alguna fila incompleta
any(!complete.cases(datos))
```

```
## [1] TRUE
```

Una vez detectado que existen valores ausentes, se estudia la distribución de los mismos.

```
# Número de datos ausentes por variable
map_dbl(datos, .f = function(x){sum(is.na(x))})
```

```
## PassengerId    Survived      Pclass        Name         Sex         Age
##           0           0           0           0           0         177
##       SibSp       Parch      Ticket        Fare       Cabin    Embarked
##           0           0           0           0           0           0
```

Viendo esta tabla, aparece una contradicción con el glimse() mostrado en el apartado anterior. El primer valor de la variable *Cabin* es “”, es decir, no se dispone del valor, sin embargo, el contaje de los valores ausentes no lo muestra. Esto ocurre porque R interpreta el valor “” como un *character*, no como valor ausente NA. Se procede a identificar qué variables contienen valores “”.

```
datos %>% map_lgl(.f = function(x){any(!is.na(x) & x == "")})
```

```
## PassengerId    Survived      Pclass        Name         Sex         Age
##       FALSE       FALSE       FALSE       FALSE       FALSE       FALSE
##       SibSp       Parch      Ticket        Fare       Cabin    Embarked
##       FALSE       FALSE       FALSE       FALSE        TRUE        TRUE
```

Las variables *Cabin* y *Embarked* contienen al menos un valor "", se sustituyen por NA.

```
# La variable Cabin está almacenada como character
datos$Cabin[datos$Cabin == ""] <- NA
```

La sustitución de valores en *factors* debe de hacerse con más cautela que cuando se hace en vectores de otro tipo. Esto se debe a que los *factors* almacenan los niveles originales y no se eliminan, aunque el *factor* ya no contenga ese valor. Una forma sencilla de evitar este problema es convertir el *factor* a *character*, hacer la modificación y volver a convertirlo a *factor*.

```
levels(datos$Embarked)
```

```
## [1] ""  "C" "Q" "S"
```

```
datos$Embarked <- as.character(datos$Embarked)
datos$Embarked[datos$Embarked == ""] <- NA
datos$Embarked <- as.factor(datos$Embarked)
levels(datos$Embarked)
```

```
## [1] "C" "Q" "S"
```

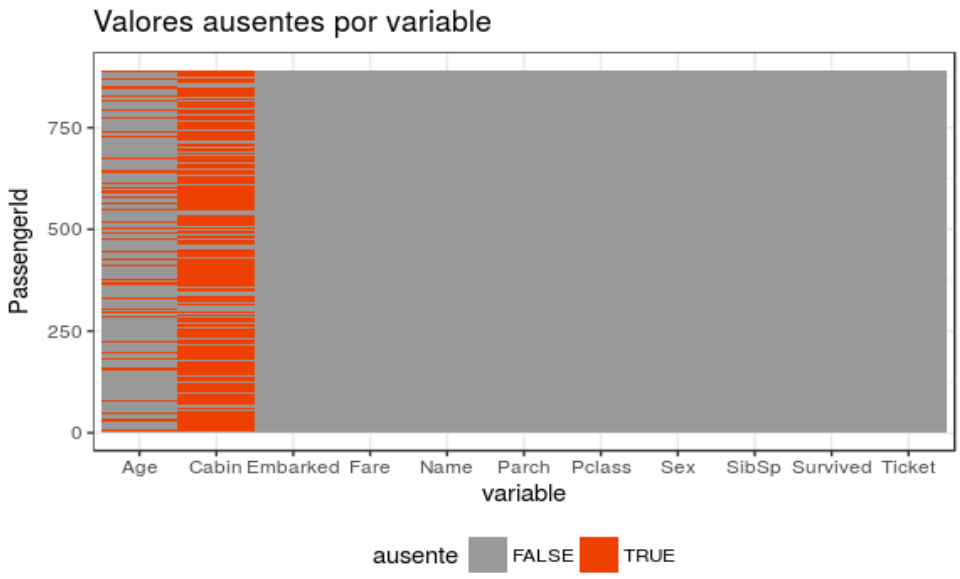
```
# Este cambio también se aplica al dataframe datos_long
datos_long$valor[datos_long$valor == ""] <- NA
```

Una vez sustituidos se repite el contaje de valores ausentes.

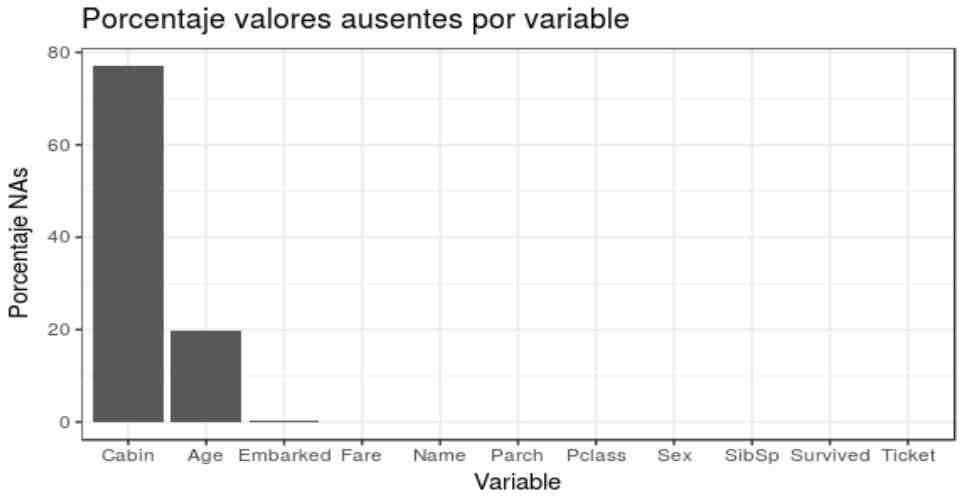
```
# Número de datos ausentes por variable
map_dbl(datos, .f = function(x){sum(is.na(x))})
```

```
## PassengerId    Survived      Pclass        Name         Sex         Age
##           0           0           0           0           0         177
##       SibSp       Parch      Ticket        Fare       Cabin    Embarked
##           0           0           0           0         687           2
```

```
# Representación gráfica de los datos ausentes
datos_long <- datos_long %>%  mutate(ausente = is.na(valor))
ggplot(data = datos_long, aes(x = variable, y = PassengerId, fill = ausente)) +
  geom_raster() +
  scale_fill_manual(values = c("gray60", "orangered2")) +
  theme_bw() +
  labs(title = "Valores ausentes por variable") +
  theme(legend.position = "bottom")
```




```
datos_long %>%
  group_by(variable) %>%
  summarize(porcentaje_NA = 100 * sum(is.na(valor)) / length(valor)) %>%
  ggplot(aes(x = reorder(variable, desc(porcentaje_NA)), y = porcentaje_NA)) +
    geom_col() +
    labs(title = "Porcentaje valores ausentes por variable",
         x = "Variable", y = "Porcentaje NAs") +
    theme_bw()
```



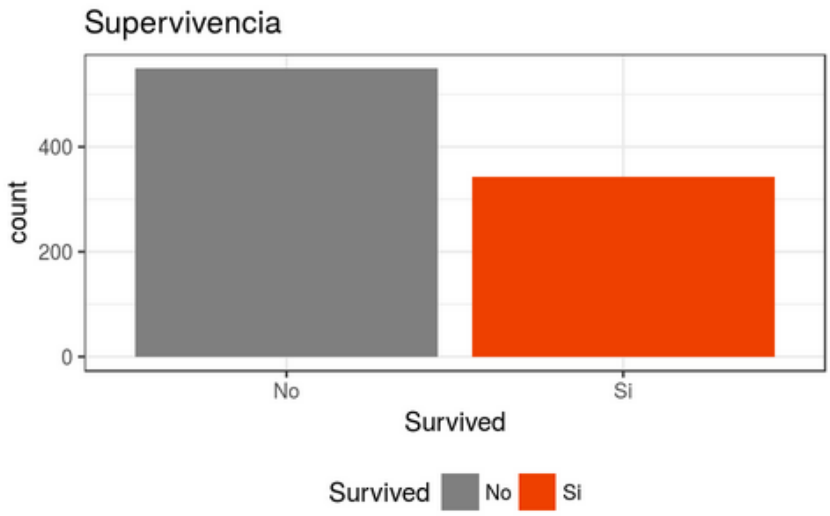
La distribución de los valores ausentes muestra que, para poco más del 20% de los pasajeros, se dispone de información sobre la cabina. La variable *Age* también está ausente en un número considerable de pasajeros. Solo para dos pasajeros se desconoce el puerto desde el que embarcaron.

Los valores ausentes son muy importantes a la hora de crear modelos, algunos algoritmos no aceptan observaciones incompletas o bien se ven muy influenciados por ellas. Aunque la imputación de valores ausentes es parte del preprocesado y, por lo tanto, debe de aprenderse únicamente con los datos de entrenamiento, su identificación se tiene que realizar antes de separar los datos para asegurar que se establecen todas las estrategias de imputación necesarias. Por ejemplo, solo dos observaciones tienen ausente la variable *Embarked*, si esas dos observaciones caen en el conjunto de test, al estudiar el conjunto de entrenamiento, no se identificaría la necesidad de imputar esta variable.

Distribución de variables respuesta

Cuando se crea un modelo, es muy importante estudiar la distribución de la variable respuesta, ya que, a fin de cuentas, es lo que nos interesa predecir.

```
ggplot(data = datos, aes(x = Survived, y = ..count.., fill = Survived)) +
  geom_bar() +
  scale_fill_manual(values = c("gray50", "orangered2")) +
  labs(title = "Supervivencia") +
  theme_bw() +
  theme(legend.position = "bottom")
```




```
# Tabla de frecuencias
table(datos$Survived)
```

```
##
##  No  Si
## 549 342
```

```
prop.table(table(datos$Survived)) %>% round(digits = 2)
```

```
##
##   No   Si
## 0.62 0.38
```

Para que un modelo predictivo sea útil, debe de tener un porcentaje de acierto superior a lo esperado por azar o a un determinado nivel basal. En problemas de clasificación, el nivel basal es el que se obtiene si se asignan todas las observaciones a la clase mayoritaria (la moda). En el naufragio del Titanic, dado que el 62% de los pasajeros fallecieron, si siempre se predice *Survived = No*, el porcentaje de aciertos será aproximadamente del 62%. Este es el porcentaje mínimo que hay que intentar superar con los modelos predictivos. (Siendo estrictos, este porcentaje tendrá que ser recalculado únicamente con el conjunto de entrenamiento).

```
# Porcentaje de aciertos si se predice para todas las observaciones que no
sobrevivieron.
n_observaciones <- nrow(datos)
predicciones    <- rep(x = "No",  n_observaciones)
mean(predicciones == datos$Survived) * 100
```

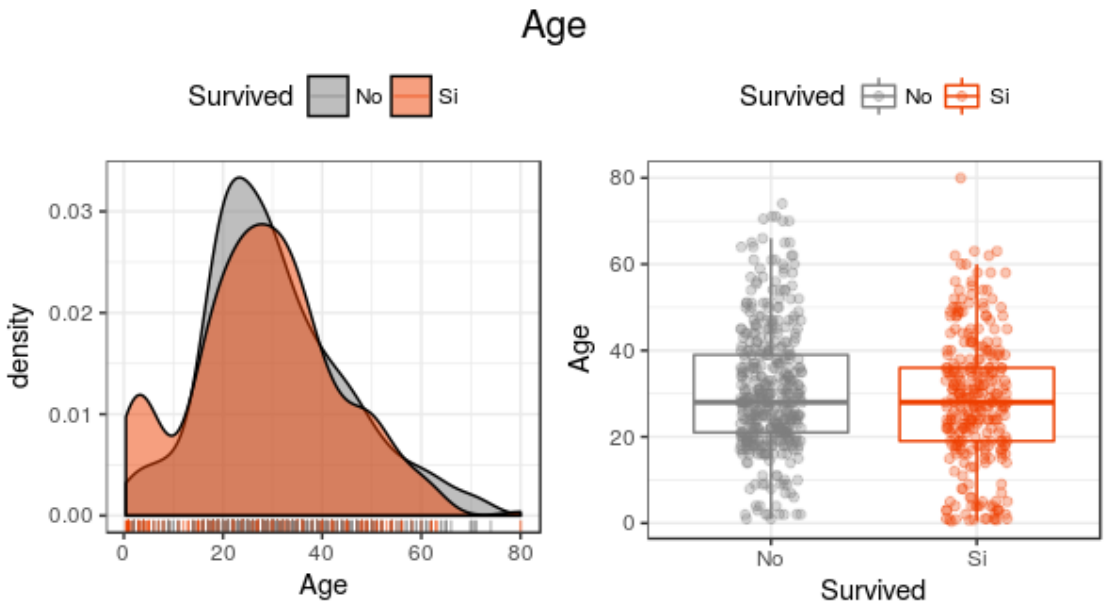
```
## [1] 61.61616
```

Distribución de variables continuas

Como el objetivo del estudio es predecir qué pasajeros sobrevivieron y cuáles no, el análisis de cada variable se hace en relación a la variable respuesta *Survived*. Analizando los datos de esta forma, se pueden empezar a extraer ideas sobre qué variables están más relacionadas con la supervivencia.

```
library(ggpubr)
p1 <- ggplot(data = datos, aes(x = Age, fill = Survived)) +
      geom_density(alpha = 0.5) +
      scale_fill_manual(values = c("gray50", "orangered2")) +
      geom_rug(aes(color = Survived), alpha = 0.5) +
      scale_color_manual(values = c("gray50", "orangered2")) +
      theme_bw()
p2 <- ggplot(data = datos, aes(x = Survived, y = Age, color = Survived)) +
      geom_boxplot(outlier.shape = NA) +
      geom_jitter(alpha = 0.3, width = 0.15) +
      scale_color_manual(values = c("gray50", "orangered2")) +
      theme_bw()
```

```
final_plot <- ggarrange(p1, p2, legend = "top")
final_plot <- annotate_figure(final_plot, top = text_grob("Age", size = 15))
final_plot
```




```
# Estadísticos de la edad de los supervivientes y fallecidos
datos %>% filter(!is.na(Age)) %>% group_by(Survived) %>%
          summarise(media = mean(Age),
                    mediana = median(Age),
                    min = min(Age),
                    max = max(Age))
```

```
## # A tibble: 2 x 5
##   Survived media mediana   min   max
##   <fct>    <dbl>   <dbl> <dbl> <dbl>
## 1 No        30.6     28. 1.00    74.
## 2 Si        28.3     28. 0.420   80.
```

La distribución de la edad de los pasajeros parece ser muy similar entre el grupo de supervivientes y fallecidos, con dos excepciones: en el rango de edad aproximado de 0 a 10 años, el porcentaje de supervivencia es mucho mayor, mientras que, en el extremo opuesto, a partir de los 60 años, la tendencia se invierte. Dos hipótesis que podrían explicar estos patrones son: que, según los registros, en el protocolo de evacuación del Titanic se priorizó que mujeres y niños subiesen a los botes salvavidas, y que los ancianos tuviesen menor movilidad para alcanzar las zonas de evacuación.

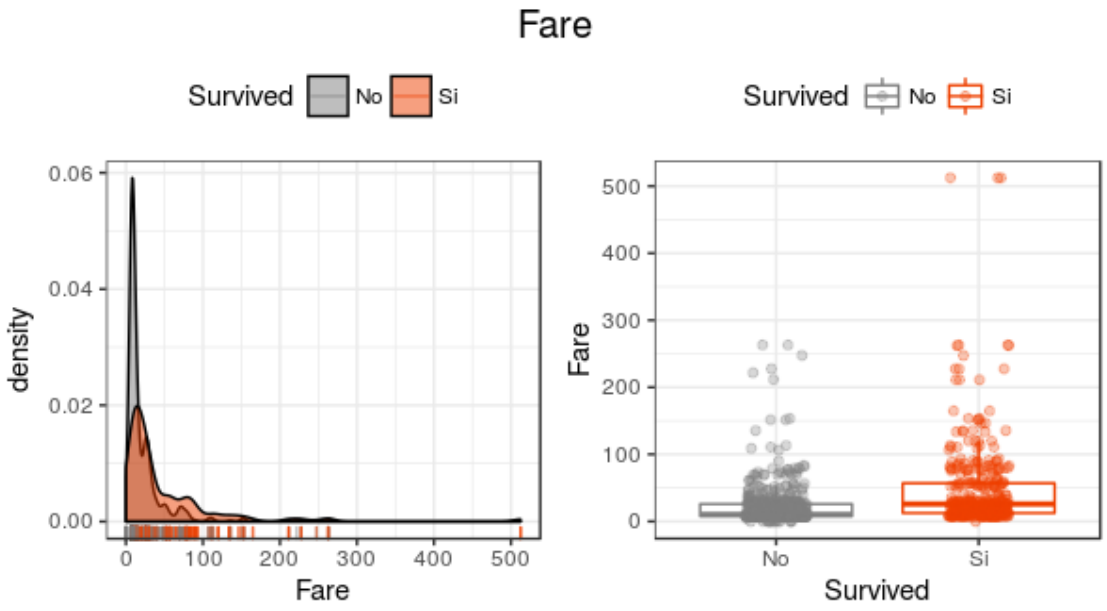
Cuando la información de una variable continua reside en si se superan o no determinados límites, los modelos predictivos suelen conseguir mejores resultados si la variable se discretiza

en intervalos. En este caso, una posible aproximación es crear una nueva variable que clasifique a los pasajeros en niño, adulto o anciano.

```
datos <- datos %>%
         mutate(Age_grupo = case_when(Age <= 10  ~ "niño",
                                      Age > 10 & Age <= 60  ~ "adulto",
                                      Age > 60 ~ "anciano"))
datos$Age_grupo <- as.factor(datos$Age_grupo)
```

Esta nueva variable se analizará junto con el resto de variables cualitativas.

```
p1 <- ggplot(data = datos, aes(x = Fare, fill = Survived)) +
      geom_density(alpha = 0.5) +
      scale_fill_manual(values = c("gray50", "orangered2")) +
      geom_rug(aes(color = Survived), alpha = 0.5) +
      scale_color_manual(values = c("gray50", "orangered2")) +
      theme_bw()
p2 <- ggplot(data = datos, aes(x = Survived, y = Fare, color = Survived)) +
      geom_boxplot(outlier.shape = NA) +
      geom_jitter(alpha = 0.3, width = 0.15) +
      scale_color_manual(values = c("gray50", "orangered2")) +
      theme_bw()
final_plot <- ggarrange(p1, p2, legend = "top")
final_plot <- annotate_figure(final_plot, top = text_grob("Fare", size = 15))
final_plot
```

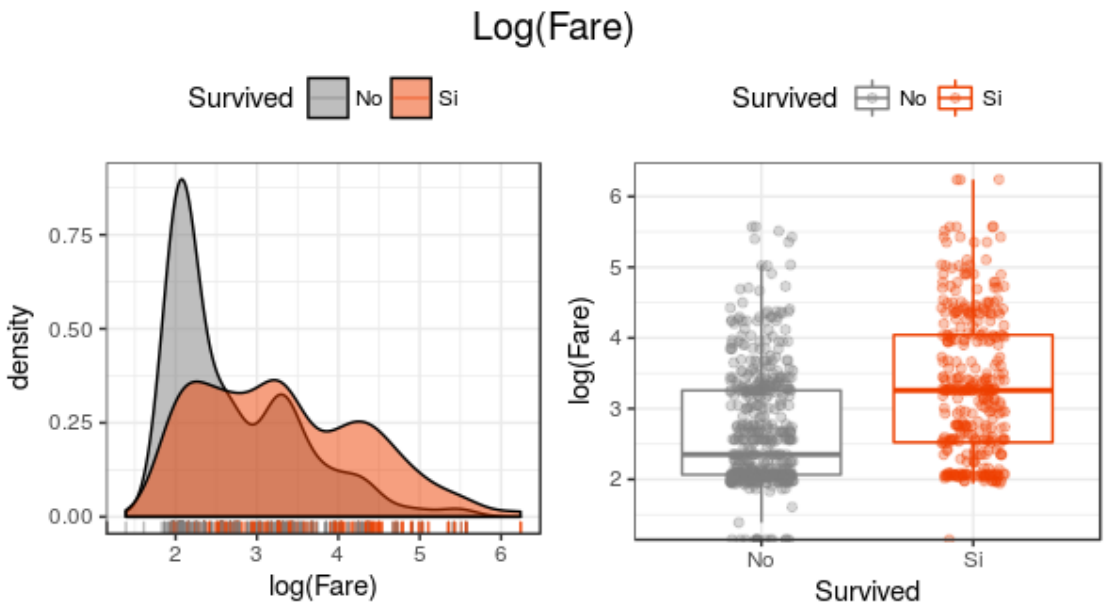


```
# Estadísticos del precio del billete de los supervivientes y fallecidos
datos %>% filter(!is.na(Fare)) %>% group_by(Survived) %>%
          summarise(media = mean(Fare),
                    mediana = median(Fare),
                    min = min(Fare),
                    max = max(Fare))
```

```
## # A tibble: 2 x 5
##   Survived media mediana   min   max
##   <fct>    <dbl>   <dbl> <dbl> <dbl>
## 1 No        22.1    10.5    0.  263.
## 2 Si        48.4    26.0    0.  512.
```

La variable *Fare* tiene una distribución asimétrica, muchos billetes tenían un coste bajo y unos pocos un coste alto. Este tipo de distribución suele visualizarse mejor tras una trasformación logarítmica.

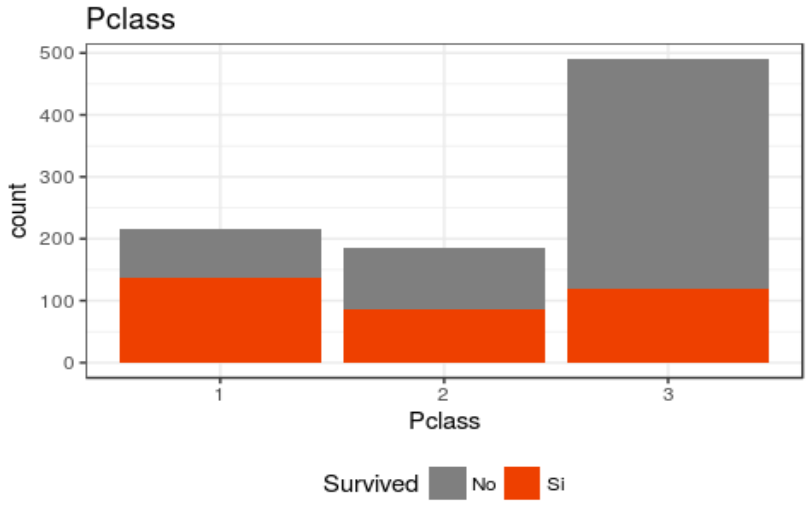
```
p1 <- ggplot(data = datos, aes(x = log(Fare), fill = Survived)) +
      geom_density(alpha = 0.5) +
      scale_fill_manual(values = c("gray50", "orangered2")) +
      geom_rug(aes(color = Survived), alpha = 0.5) +
      scale_color_manual(values = c("gray50", "orangered2")) +
      theme_bw()
p2 <- ggplot(data = datos, aes(x = Survived, y = log(Fare), color = Survived)) +
      geom_boxplot(outlier.shape = NA) +
      geom_jitter(alpha = 0.3, width = 0.15) +
      scale_color_manual(values = c("gray50", "orangered2")) +
      theme_bw()
final_plot <- ggarrange(p1, p2, legend = "top")
final_plot <- annotate_figure(final_plot, top = text_grob("Log(Fare)", size =15))
final_plot
```



Los datos indican que el precio medio de los billetes de las personas que sobrevivieron era superior al de los que fallecieron. ¿Significa esto que, haber pagado más por el billete, garantizaba una mayor probabilidad de supervivencia? No parece muy verosímil que esto fuese así. Este es un caso típico en el que cabe recordar que correlación/asociación no son sinónimos de causalidad. Es probable que exista otra variable subyacente que sí tenga un vínculo de causalidad con la probabilidad de supervivencia y que, a su vez, esté correlacionada con el precio del billete. Una hipótesis podría ser: los pasajeros de primera clase tuvieron ciertas facilidades para acceder a los botes de evacuación por lo que consiguieron salvarse con más éxito que los de segunda y tercera clase. Los billetes de primera clase costaban mucho más dinero que los de las clases inferiores, de ahí que el precio medio de billetes entre los supervivientes fuese superior.

Distribución de variables cualitativas

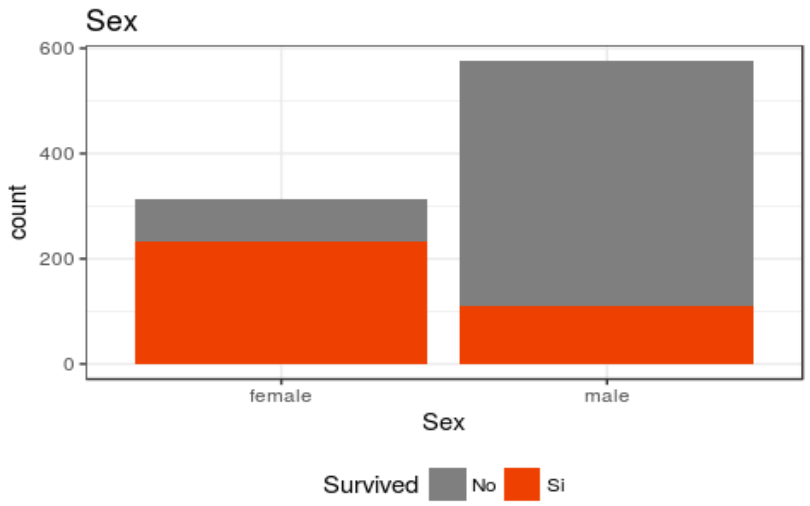
```
ggplot(data = datos, aes(x = Pclass, y = ..count.., fill = Survived)) +
  geom_bar() +
  scale_fill_manual(values = c("gray50", "orangered2")) +
  labs(title = "Pclass") +
  theme_bw() +
  theme(legend.position = "bottom")
```




```
# Tabla de frecuencias relativas de supervivientes por clase
prop.table(table(datos$Pclass, datos$Survived), margin = 1) %>% round(digits = 2)
```

```
##
##       No   Si
##   1 0.37 0.63
##   2 0.53 0.47
##   3 0.76 0.24
```

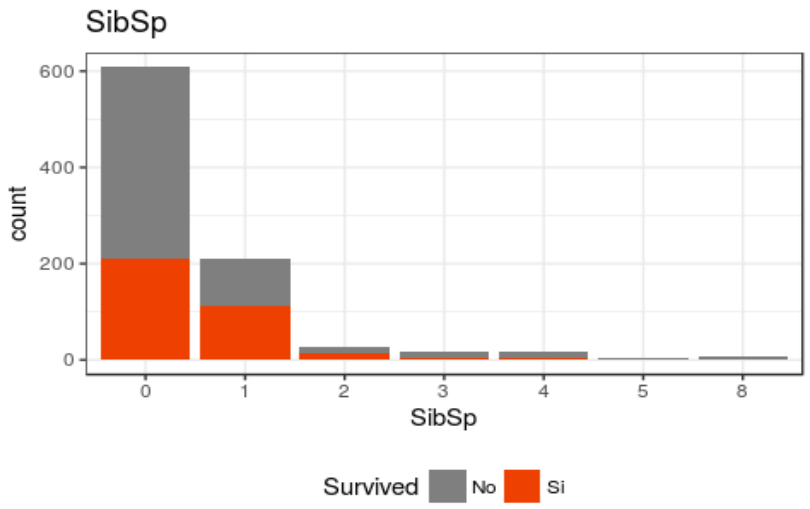
```
ggplot(data = datos, aes(x = Sex, y = ..count.., fill = Survived)) +
  geom_bar() +
  labs(title = "Sex") +
  scale_fill_manual(values = c("gray50", "orangered2")) +
  theme_bw() +
  theme(legend.position = "bottom")
```




```
# Tabla de frecuencias relativas de supervivientes por sexo
prop.table(table(datos$Sex, datos$Survived), margin = 1) %>% round(digits = 2)
```

```
##
##            No   Si
##   female 0.26 0.74
##   male   0.81 0.19
```

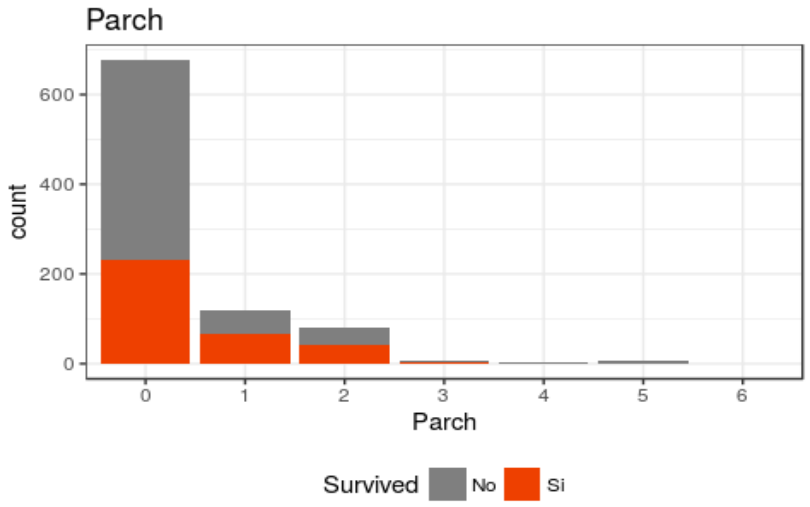
```
ggplot(data = datos, aes(x = SibSp, y = ..count.., fill = Survived)) +
  geom_bar() +
  scale_fill_manual(values = c("gray50", "orangered2")) +
  labs(title = "SibSp") +
  theme_bw() +
  theme(legend.position = "bottom")
```




```r
# Tabla de frecuencias relativas de supervivientes por número de familiares
prop.table(table(datos$SibSp, datos$Survived), margin = 1) %>% round(digits = 2)
```

```
##
##        No   Si
##   0 0.65 0.35
##   1 0.46 0.54
##   2 0.54 0.46
##   3 0.75 0.25
##   4 0.83 0.17
##   5 1.00 0.00
##   8 1.00 0.00
```

```r
ggplot(data = datos, aes(x = Parch, y = ..count.., fill = Survived)) +
  geom_bar() +
  scale_fill_manual(values = c("gray50", "orangered2")) +
  labs(title = "Parch") +
  theme_bw() +
  theme(legend.position = "bottom")
```




```
# Tabla de frecuencias relativas de supervivientes por Parch
prop.table(table(datos$Parch, datos$Survived), margin = 1) %>% round(digits = 2)
```

```
##
##       No   Si
##   0 0.66 0.34
##   1 0.45 0.55
##   2 0.50 0.50
##   3 0.40 0.60
##   4 1.00 0.00
##   5 0.80 0.20
##   6 1.00 0.00
```

Las variables *SibSp* y *Parch*, ambas relacionadas con el número de familiares a bordo, se han tratado como una variable cualitativa para el análisis exploratorio. Al agrupar las observaciones en los diferentes niveles, algunos de ellos apenas contienen unas pocas (*Parch = 4, 5, 6*). Que niveles de una variable cualitativa tengan muy poca representación (distribución desbalanceada) puede suponer un problema a la hora de ajustar los modelos u optimizarlos mediante validación cruzada. Por esta razón, a la hora de ajustar el modelo, conviene seguir una de estas dos estrategias:

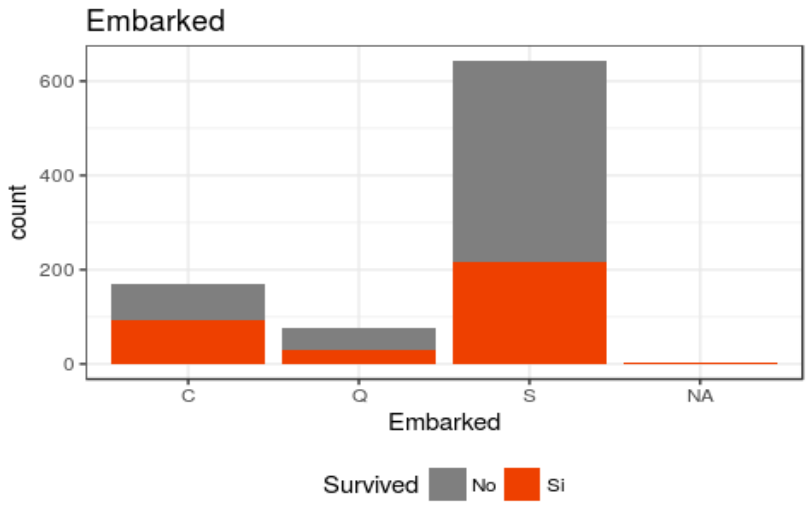
- Agrupar todos los grupos minoritarios en uno solo, por ejemplo, combinar los grupos *Parch = 4, 5, 6* en uno nuevo que sea *Parch > 3*.
- Tratar el predictor como una variable continua.

En este caso, se opta por la segunda opción.

```
# Para pasar de factor a numeric primero se convierte a character
datos$SibSp <- as.character(datos$SibSp)
datos$SibSp <- as.numeric(datos$SibSp)
```

```
datos$Parch <- as.character(datos$Parch)
datos$Parch <- as.numeric(datos$Parch)
```

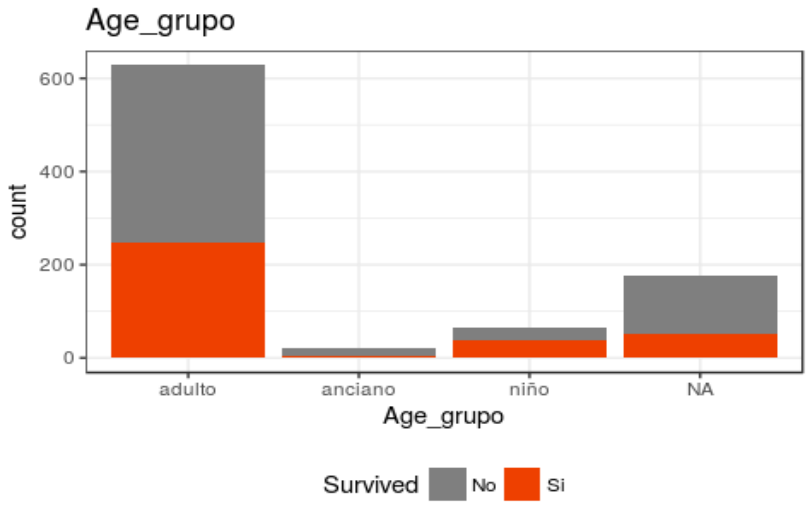
```
ggplot(data = datos, aes(x = Embarked, y = ..count.., fill = Survived)) +
  geom_bar() +
  scale_fill_manual(values = c("gray50", "orangered2")) +
  labs(title = "Embarked") +
  theme_bw() +
  theme(legend.position = "bottom")
```




```
# Tabla de frecuencias relativas de supervivientes por puerto de embarque
prop.table(table(datos$Embarked, datos$Survived), margin = 1) %>% round(digits = 2)
```

```
##
##        No   Si
##   C 0.45 0.55
##   Q 0.61 0.39
##   S 0.66 0.34
```

```
ggplot(data = datos, aes(x = Age_grupo, y = ..count.., fill = Survived)) +
  geom_bar() +
  scale_fill_manual(values = c("gray50", "orangered2")) +
  labs(title = "Age_grupo") +
  theme_bw() +
  theme(legend.position = "bottom")
```




```
# Tabla de frecuencias relativas de supervivientes por grupo de edad
prop.table(table(datos$Age_grupo, datos$Survived), margin =1) %>% round(digits = 2)
```

```
##
##              No   Si
##   adulto   0.61 0.39
##   anciano  0.77 0.23
##   niño     0.41 0.59
```

Las variables *PassengerId*, *Name* y *Cabin* no se han incluido en la exploración porque, a priori, no aportan información relevante sobre la supervivencia. En la sección *feature engineering* se analizan con más detenimiento.

Importancia de las variables

La representación gráfica de la distribución de las variables en función de si los pasajeros sobrevivieron o no, ayuda a tener una idea de qué variables pueden ser buenos predictores para el modelo y cuales no aportan información o la que aportan es redundante. Aunque la creación de un buen modelo debe entenderse como un proceso iterativo, en el que se van ajustando y probando distintos modelos, existen ciertas pistas que pueden ayudar a realizar una selección inicial adecuada.

- Si dos variables numéricas están muy correlacionadas, añaden información redundante al modelo, por lo tanto, no conviene incorporar ambas. Si esto ocurre, se puede: excluir aquella que, acorde al criterio del analista, no está realmente asociada con la variable

respuesta; o combinarlas para recoger toda su información en una única nueva variable, por ejemplo, con un PCA.

- Si una variable tiene varianza igual o próxima a cero (su valor es el mismo o casi el mismo para todas las observaciones) añade al modelo más ruido que información, por lo que suele ser conveniente excluirla.

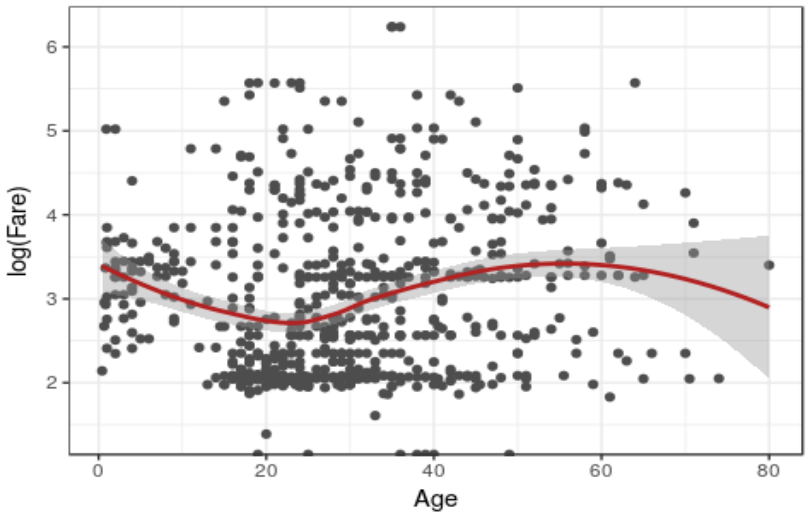
- Si alguno de los niveles de una variable cualitativa tiene muy pocas observaciones en comparación a los otros niveles, puede ocurrir que, durante la validación cruzada o *bootstrapping*, algunas particiones no contengan ninguna observación de dicha clase (varianza cero), lo que puede dar lugar a errores. En estos casos, suele ser conveniente eliminar las observaciones del grupo minoritario (si es una variable multiclase), eliminar la variable (si solo tiene dos niveles) o asegurar que, en la creación de las particiones, se garantice que todos los grupos estén representados en cada una de ellas.

Correlación entre variables continuas

```
cor.test(x = datos$Age, y = datos$Fare, method = "pearson")
```

```
##
##  Pearson's product-moment correlation
##
## data:  datos$Age and datos$Fare
## t = 2.5753, df = 712, p-value = 0.01022
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.02285549 0.16825304
## sample estimates:
##        cor
## 0.09606669
```

```
ggplot(data = datos, aes(x = Age, y = log(Fare))) +
  geom_point(color = "gray30") +
  geom_smooth(color = "firebrick") +
  theme_bw()
```

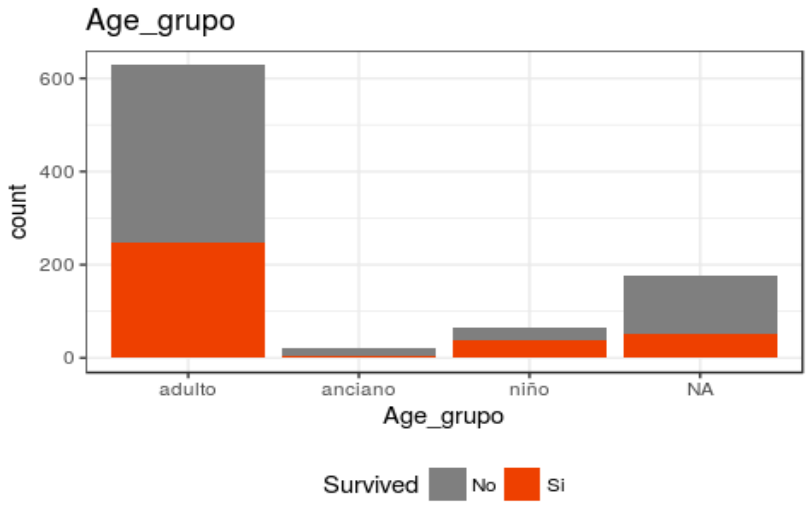


La correlación lineal entre la edad del pasajero y el precio del billete, aun cuando significativa (*p-value = 0.010*), es mínima *(cor = 0.096)*. El diagrama de dispersión tampoco apunta a ningún tipo de relación no lineal evidente. Las variables no contienen información redundante.

Contraste de proporciones

Para la identificación de potenciales predictores cualitativos, es interesante encontrar las variables y niveles de las mismas que muestran una proporción de supervivientes alejada de lo esperado por el nivel basal, en este caso el 38.38%. Este porcentaje se corresponde con la proporción de supervivientes respecto al total de pasajeros, es decir, el valor esperado si no existiese relación entre la variable y la supervivencia. Estas diferencias no siempre son fáciles de apreciar en una gráfica, sobre todo, cuando el número de observaciones es distinto en cada grupo. Por ejemplo, en la siguiente imagen, difícilmente se puede determinar si la proporción de supervivientes en los grupos de ancianos y niños se aleja mucho del 38.38%.

```
ggplot(data = datos, aes(x = Age_grupo, y = ..count.., fill = Survived)) +
  geom_bar() +
  scale_fill_manual(values = c("gray50", "orangered2")) +
  labs(title = "Age_grupo") +
  theme_bw() +
  theme(legend.position = "bottom")
```



Para facilitar este tipo de análisis, resulta útil crear variables *dummy* con todos los niveles de las variables cualitativas (proceso conocido como *binarización* o *one hot encoding*) y aplicar un test de contraste de proporciones. Una de las ventajas de este tipo de test es que tiene en cuenta el número de observaciones, no es lo mismo que de 10 pasajeros, 3 sobrevivan y 7 no, que de 1000 pasajeros, 300 sobrevivan y 700 no. En el segundo caso, se tiene mucha más seguridad de que la verdadera proporción de supervivencia tiende al 30%. Este mismo análisis podría hacerse empleando la proporción de fallecidos.

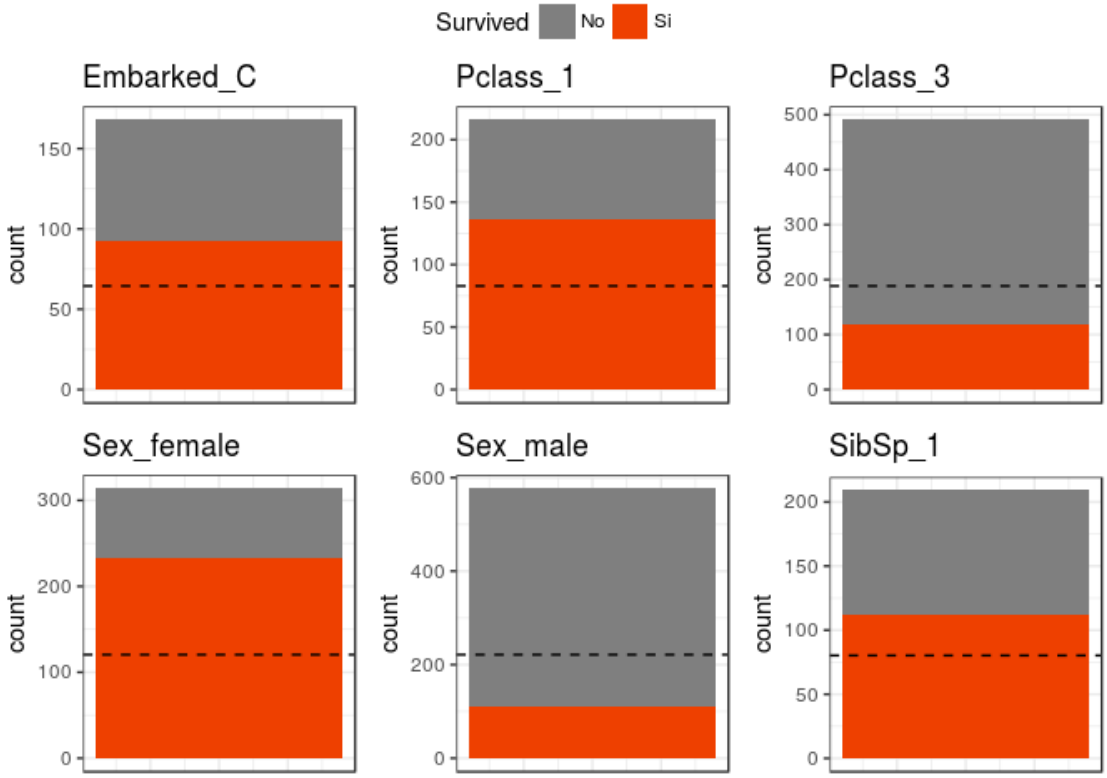
```
# Se excluyen las variables continuas y las cualitativas que no agrupan a los
# pasajeros. También la variable Cabin por su alto % de valores NA.
datos_cualitativos <- datos %>% select(-Age, -Fare, -Name, -Ticket, -Cabin,
                                       -PassengerId)

datos_cualitativos_tidy <- datos_cualitativos %>%
                           gather(key = "variable", value = "grupo",-Survived)

# Se eliminan los valores NA para que no se interpreten como un grupo
datos_cualitativos_tidy <- datos_cualitativos_tidy %>% filter(!is.na(grupo))

# Se añade un identificador formado por el nombre de la variable y el grupo
datos_cualitativos_tidy <- datos_cualitativos_tidy %>%
                           mutate(variable_grupo = paste(variable, grupo, sep="_"))
# Función que calcula el test de proporciones para la columna "Survived" de un df
test_proporcion <- function(df){
  n_supervivientes <- sum(df$Survived == "Si")
  n_fallecidos     <- sum(df$Survived == "No")
  n_total <- n_supervivientes + n_fallecidos
  test <- prop.test(x = n_supervivientes, n = n_total, p = 0.3838)
  prop_supervivientes <- n_supervivientes / n_total
  return(data.frame(p_value = test$p.value, prop_supervivientes))
}
```

```
# Se agrupan los datos por "variable_grupo" y se aplica a cada grupo la función
# test_proporcion()
analisis_prop <- datos_cualitativos_tidy %>%
                 group_by(variable_grupo) %>%
                 nest() %>%
                 arrange(variable_grupo) %>%
                 mutate(prop_test = map(.x = data, .f = test_proporcion)) %>%
                 unnest(prop_test) %>%
                 arrange(p_value) %>%
                 select(variable_grupo,p_value, prop_supervivientes)
analisis_prop
```

```
## # A tibble: 25 x 3
##    variable_grupo  p_value prop_supervivientes
##    <chr>             <dbl>               <dbl>
##  1 Sex_female     1.30e-38               0.742
##  2 Sex_male       9.36e-22               0.189
##  3 Pclass_1       1.85e-13               0.630
##  4 Pclass_3       1.57e-10               0.242
##  5 SibSp_1        8.59e- 6               0.536
##  6 Embarked_C     8.77e- 6               0.554
##  7 Parch_1        2.76e- 4               0.551
##  8 Age_grupo_niño 8.83e- 4               0.594
##  9 Pclass_2       1.61e- 2               0.473
## 10 Embarked_S     1.62e- 2               0.337
## # ... with 15 more rows
```

```
# Representación gráfica de la distribución de los 6 grupos con menor p-value
top6_grupos <- analisis_prop %>% pull(variable_grupo) %>% head(6)

# Se crea una función que, dados un dataframe y el nombre de un grupo, genere la
# representación gráfica de supervivientes y no supervivientes.
plot_grupo <- function(grupo, df, threshold_line = 0.3838){

  p <- ggplot(data = df, aes(x = 1, y = ..count.., fill = Survived)) +
            geom_bar() +
            scale_fill_manual(values = c("gray50", "orangered2")) +
            # Se añade una línea horizontal en el nivel basal
            geom_hline(yintercept = nrow(df) * threshold_line,
                       linetype = "dashed") +
            labs(title = grupo) +
            theme_bw() +
            theme(legend.position = "bottom",
                  axis.text.x = element_blank(),
                  axis.title.x = element_blank(),
                  axis.ticks.x = element_blank())
  return(p)
}
```

```
datos_graficos <- datos_cualitativos_tidy %>%
                  filter(variable_grupo %in% top6_grupos) %>%
                  group_by(variable_grupo) %>%
                  nest() %>%
                  arrange(variable_grupo)

plots <- map2(datos_graficos$variable_grupo, .y = datos_graficos$data,
              .f = plot_grupo)

ggarrange(plotlist = plots, common.legend = TRUE)
```



El listado obtenido muestra, ordenados de menor a mayor *p-value*, cada uno de los posibles grupos simples en los que se puede diferenciar a los pasajeros. Hay que tener precaución a la hora de interpretarlo, la idea es la siguiente: cuanto menor es el *p-value* de un grupo, mayor la evidencia de que la proporción de supervivientes en dicho grupo se aleja de lo esperado (0.38), tanto por encima como por debajo, si no existiese relación entre esa variable y la supervivencia. Ahora bien:

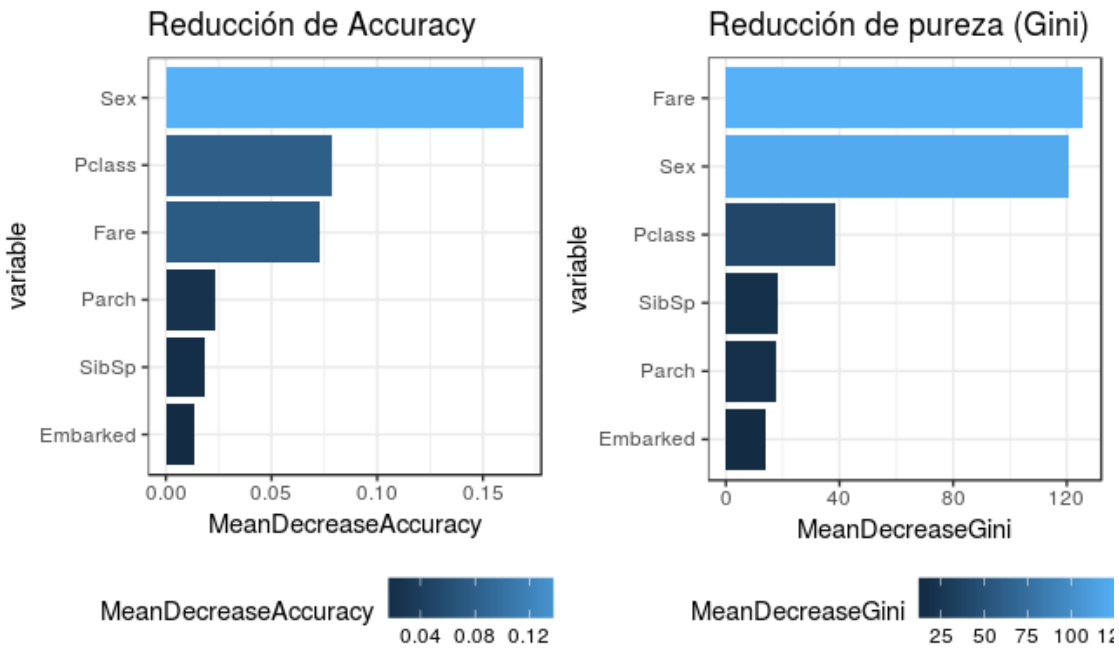
- Al tratarse de comparaciones múltiples sin corrección del error tipo I, no se debe de interpretar el valor del *p-value* más allá de como una forma de ordenación y no para determinar si las variables están significativamente asociadas.
- El estudio aísla cada variable por separado, no tiene en cuenta posibles combinaciones, por ejemplo, ser mujer con hijos frente a mujer sin hijos.

Este análisis tiene como objetivo sacar a la luz posibles relaciones entre las variables disponibles y la supervivencia de los pasajeros, sin embargo, no debe de emplearse como una selección de predictores (al menos no por si sola). Que una variable aparezca al final del listado no garantiza que no sea importante en el modelo de clasificación final.

Random forest

Otra estrategia ampliamente extendida para estudiar la importancia de variables es el empleo de *Random Forest*. El problema de aplicarlo a este ejemplo es que no acepta valores ausentes, tema que todavía no se ha tratado (visto más adelante). Por el momento, se excluyen las variables incompletas. Para información detallada sobre los fundamentos de este método, leer la sección *Importancia de los predictores* del capítulo Árboles de predicción.

```
library(randomForest)
datos_rf <- datos %>%
            select(-PassengerId, -Name, -Ticket, -Cabin, -Age, -Age_grupo) %>%
            na.omit()
datos_rf <- map_if(.x = datos_rf, .p = is.character, .f = as.factor) %>%
            as.data.frame()
modelo_randforest <- randomForest(formula = Survived ~ . ,
                                  data = na.omit(datos_rf),
                                  mtry = 5,
                                  importance = TRUE,
                                  ntree = 1000)
importancia <- as.data.frame(modelo_randforest$importance)
importancia <- rownames_to_column(importancia,var = "variable")

p1 <- ggplot(data = importancia, aes(x = reorder(variable, MeanDecreaseAccuracy),
                                     y = MeanDecreaseAccuracy,
                                     fill = MeanDecreaseAccuracy)) +
      labs(x = "variable", title = "Reducción de Accuracy") +
      geom_col() +
      coord_flip() +
      theme_bw() + theme(legend.position = "bottom")

p2 <- ggplot(data = importancia, aes(x = reorder(variable, MeanDecreaseGini),
                                     y = MeanDecreaseGini,
                                     fill = MeanDecreaseGini)) +
      labs(x = "variable", title = "Reducción de pureza (Gini)") +
      geom_col() +
      coord_flip() +
      theme_bw() + theme(legend.position = "bottom")

ggarrange(p1, p2)
```



Ambos análisis apuntan a que las variables *Sex*, *Class* y *Fare* tienen una influencia alta sobre las probabilidades de supervivencia.

Conclusión análisis exploratorio

La exploración de los datos, el estudio de su distribución, y su posible relación con la variable respuesta parecen indicar que los factores que más influyeron en la supervivencia de los pasajeros fueron: el sexo, la clase a la que pertenecían y si tenían o no al menos un hijo a bordo. También se ha detectado que las variables continuas no están correlacionadas y que las variables *Age*, *Cabin* y *Embarked* tienen valores ausentes.

División de los datos en entrenamiento y test

Evaluar la capacidad predictiva de un modelo consiste en comprobar cómo de próximas son sus predicciones a los verdaderos valores de la variable respuesta. Para poder cuantificar de forma correcta este error, se necesita disponer de un conjunto de observaciones, de las que se conozca la variable respuesta, pero que el modelo no haya "visto", es decir, que no hayan participado en su ajuste. Con esta finalidad, se dividen los datos disponibles en un conjunto de entrenamiento y un conjunto de test. El tamaño adecuado de las particiones depende en gran medida de la cantidad de datos disponibles y la seguridad que se necesite en la estimación del error, 80%-20% suele dar buenos resultados. El reparto debe hacerse de forma aleatoria o aleatoria-estratificada. Evaluación de modelos.

```
set.seed(123)
# Se crean los índices de las observaciones de entrenamiento
train <- createDataPartition(y = datos$Survived, p = 0.8, list = FALSE, times = 1)
datos_train <- datos[train, ]
datos_test  <- datos[-train, ]
```

Es importante verificar que la distribución de la variable respuesta es similar en el conjunto de entrenamiento y en el de test. Por defecto, la función createDataPartition() garantiza una distribución aproximada.

```
prop.table(table(datos_train$Survived))
```

```
##
##        No        Si
## 0.6162465 0.3837535
```

```
prop.table(table(datos_test$Survived))
```

```
##
##        No        Si
## 0.6158192 0.3841808
```

Este tipo de reparto estratificado asegura que el conjunto de entrenamiento y el de test sean similares en cuanto a la variable respuesta, sin embargo, no garantiza que ocurra lo mismo con los predictores. Por ejemplo, en un set de datos con 100 observaciones, un predictor binario que tenga 90 observaciones de un grupo y solo 10 de otro, tiene un alto riesgo de que, en alguna de las particiones, el grupo minoritario no tenga representantes. Si esto ocurre en el conjunto de entrenamiento, algunos algoritmos darán error al aplicarlos al conjunto de test, ya que no entenderán el valor que se les está pasando. Este problema puede evitarse eliminando variables con varianza próxima a cero.

Preprocesado de los datos

El preprocesado de datos engloba aquellas transformaciones de los datos hechas con la finalidad de que puedan ser aceptados por el algoritmo de *machine learning* o que mejoren sus resultados. Todo preprocesado de datos debe aprenderse de las observaciones de entrenamiento y luego aplicarse al conjunto de entrenamiento y al de test. Esto es muy importante para no violar la condición de que ninguna información procedente de las observaciones de test puede participar o influir en el ajuste del modelo. Aunque no es posible crear un único listado, algunos de los pasos de preprocesado que más suelen aplicarse en la práctica son:

- Imputación de valores ausentes
- Exclusión de variables con varianza próxima a cero
- Reducción de dimensionalidad
- Estandarización de las variables numéricas
- Binarización de las variables cualitativas

El paquete caret incorpora muchas funciones para preprocesar los datos. Sin embargo, para facilitar todavía más el aprendizaje de las transformaciones, únicamente con las observaciones de entrenamiento, y poder aplicarlas después a cualquier conjunto de datos, el mismo autor ha creado el paquete recipes.

La idea detrás de este paquete es la siguiente:

1. Definir cuál es la variable respuesta, los predictores y el set de datos de entrenamiento, recipe().
2. Definir todas las transformaciones (escalado, selección, filtrado…) que se desea aplicar, step_().
3. Aprender los parámetros necesarios para dichas transformaciones con las observaciones de entrenamiento rep().
4. Aplicar las trasformaciones aprendidas a cualquier conjunto de datos bake().

En los siguientes apartados, se almacenan en un objeto recipe todos los pasos de preprocesado y, finalmente, se aplican a los datos.

Imputación de valores ausentes

Tal y como se ha identificado en la exploración de datos, las variables *Cabin*, *Age* y *Embarked* contienen valores ausentes. La gran mayoría de algoritmos no aceptan observaciones incompletas, por lo que, cuando el set de datos contiene valores ausentes, se puede:

- Eliminar aquellas observaciones que estén incompletas.
- Eliminar aquellas variables que contengan valores ausentes.
- Tratar de estimar los valores ausentes empleando el resto de información disponible (imputación).

Las primeras dos opciones, aunque sencillas, suponen perder información. La eliminación de observaciones solo puede aplicarse cuando se dispone de muchas y el porcentaje de registros incompletos es muy bajo. En el caso de eliminar variables, el impacto dependerá de cuanta información aporten dichas variables al modelo.

Cuando se emplea imputación, es muy importante tener en cuenta el riesgo que se corre al introducir valores en predictores que tengan mucha influencia en el modelo. Supóngase un estudio médico en el que, cuando uno de los predictores es positivo, el modelo predice casi siempre que el paciente está sano. Para un paciente cuyo valor de este predictor se desconoce, el riesgo de que la imputación sea errónea es muy alto, por lo que es preferible obtener una predicción basada únicamente en la información disponible. Esta es otra muestra de la importancia que tiene que el analista conozca el problema al que se enfrenta y pueda así tomar la mejor decisión.

En el set de datos Titanic, si se eliminan las observaciones incompletas, se pasa de 891 observaciones a solo 183, por lo que esta no es una opción.

```
nrow(datos)
```

```
## [1] 891
```

```
nrow(na.omit(datos))
```

```
## [1] 183
```

La variable *Cabin* está ausente para casi un 80% de las observaciones, con un porcentaje tan alto de valores ausentes, no es conveniente imputarla, se excluye directamente del modelo. La variable *Age_grupo* se ha obtenido discretizando en intervalos la variable *Age*, por lo que contiene el mismo número de valores ausentes que esta última. Esto deja al modelo con dos variables que requieren de imputación: *Age_grupo* y *Embarked*.

La imputación es un proceso complejo que debe de realizarse con detenimiento, identificando cuidadosamente qué variables son las adecuadas para cada imputación. Para más detalles sobre por qué se realizan de la siguiente forma, ver Anexo 7.

El paquete recipes permite 4 métodos de imputación distintos:

step_bagimpute(): imputación vía *Bagged Trees*.

step_knnimpute(): imputación vía *K-Nearest Neighbors*.

step_meanimpute(): imputación vía media del predictor (predictores continuos).

step_modeimpute(): imputación vía moda del predictor (predictores cualitativos).

Cabe destacar también los paquetes Hmisc missForest y MICE que permiten aplicar otros métodos.

Se imputa la variable *Embarked* con el valor *C*. Como no existe una función step() que haga sustituciones por valores concretos, se realiza una sustitución de forma externa al recipe.

```
datos_train <- datos_train %>%
               mutate(Embarked = replace(Embarked, is.na(Embarked), "C"))
datos_test <-  datos_test %>%
               mutate(Embarked = replace(Embarked, is.na(Embarked), "C"))
```

La variable *Age_grupo* se imputa con el método *bagging* empleando todos los otros predictores.

```
library(recipes)
# Se crea un objeto recipe() con la variable respuesta y los predictores.
# Las variables *PassengerId*, *Name*, *Ticket* no parecen aportar información
# relevante sobre la supervivencia de los pasajeros. Excluyendo todas estas
# variables, se propone como modelo inicial el formado por los predictores:
#  Pclass + Sex + SibSp + Parch + Fare + Embarked + Age_grupo.

objeto_recipe <- recipe(formula = Survived ~ Pclass + Sex + SibSp + Parch +
                                  Fare + Embarked + Age_grupo,
                        data =  datos_train)
objeto_recipe
```

```
## Data Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor          7
```

```
objeto_recipe <- objeto_recipe %>% step_bagimpute(Age_grupo)
objeto_recipe
```

```
## Data Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor          7
##
## Operations:
##
## Bagged tree imputation for Age_grupo
```

Variables con varianza próxima a cero

No se deben incluir en el modelo predictores que contengan un único valor (cero-varianza) ya que no aportan información. Tampoco es conveniente incluir predictores que tengan una varianza próxima a cero, es decir, predictores que toman solo unos pocos valores, de los cuales, algunos aparecen con muy poca frecuencia. El problema con estos últimos es que pueden convertirse en predictores con varianza cero cuando se dividen las observaciones por validación cruzada o *bootstrap*.

La función nearZeroVar() del paquete caret y step_nzv() del paquete recipe identifican como predictores potencialmente problemáticos aquellos que tienen un único valor (cero varianza) o que cumplen las dos siguientes condiciones:

- Ratio de frecuencias: ratio entre la frecuencia del valor más común y la frecuencia del segundo valor más común. Este ratio tiende a 1 si las frecuencias están equidistribuidas y a valores grandes cuando la frecuencia del valor mayoritario supera por mucho al resto (el denominador es un número decimal pequeño). Valor por defecto freqCut = 95/5.
- Porcentaje de valores únicos: número de valores únicos dividido entre el total de muestras (multiplicado por 100). Este porcentaje se aproxima a cero cuanto mayor es la variedad de valores. Valor por defecto uniqueCut = 10.

```
datos %>% select(Pclass, Sex, SibSp, Parch, Fare, Embarked, Age_grupo) %>%
          nearZeroVar(saveMetrics = TRUE)
```

```
##           freqRatio percentUnique zeroVar   nzv
## Pclass     2.273148     0.3367003   FALSE FALSE
## Sex        1.837580     0.2244669   FALSE FALSE
## SibSp      2.909091     0.7856341   FALSE FALSE
```

```
## Parch      5.745763     0.7856341   FALSE FALSE
## Fare       1.023810    27.8338945   FALSE FALSE
## Embarked   3.833333     0.3367003   FALSE FALSE
## Age_grupo  9.812500     0.3367003   FALSE FALSE
```

Entre los predictores incluidos en el modelo, no se detecta ninguno con varianza cero o próxima a cero.

```
objeto_recipe <- objeto_recipe %>% step_nzv(all_predictors())
```

Si bien la eliminación de predictores no informativos podría considerarse un paso propio del proceso de *selección de predictores*, dado que consiste en un filtrado por varianza, tiene que realizarse antes de estandarizar los datos, ya que después, todos los predictores tienen varianza 1.

Estandarización y escalado

Cuando los predictores son numéricos, la escala en la que se miden, así como la magnitud de su varianza pueden influir en gran medida en el modelo. Muchos algoritmos de *machine learning* (*SVM*, redes neuronales, *lasso*...) son sensibles a esto, de forma que, si no se igualan de alguna forma los predictores, aquellos que se midan en una escala mayor o que tengan más varianza, dominarán el modelo aunque no sean los que más relación tienen con la variable respuesta. Existen principalmente 2 estrategias para evitarlo:

Centrado: consiste en restarle a cada valor la media del predictor al que pertenece. Si los datos están almacenados en un *dataframe*, el centrado se consigue restándole a cada valor la media de la columna en la que se encuentra. Como resultado de esta transformación, todos los predictores pasan a tener una media de cero, es decir, los valores se centran en torno al origen.

Normalización (estandarización): consiste en transformar los datos de forma que todos los predictores estén aproximadamente en la misma escala. Hay dos formas de lograrlo:

Normalización Z-score: dividir cada predictor entre su desviación típica después de haber sido centrado, de esta forma, los datos pasan a tener una distribución normal.

$$z = \frac{x - \mu}{\sigma}$$

Estandarización max-min: transformar los datos de forma que estén dentro del rango [0, 1].

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

Para este análisis se normalizan todas las variables numéricas.

```
objeto_recipe <- objeto_recipe %>% step_center(all_numeric())
objeto_recipe <- objeto_recipe %>% step_scale(all_numeric())
```

Nunca se debe estandarizar las variables después de ser binarizadas (ver siguiente sección).

Binarización de variables cualitativas

La binarización consiste en crear nuevas variables *dummy* con cada uno de los niveles de las variables cualitativas. A este proceso también se le conoce como *one hot encoding*. Por ejemplo, una variable llamada *color* que contenga los niveles *rojo*, *verde* y *azul*, se convertirá en tres nuevas variables (*color_rojo*, *color_verde*, *color_azul*), todas con el valor 0 excepto la que coincide con la observación, que toma el valor 1.

Por defecto, la función step_dummy(all_nominal()) binariza todas las variables almacenadas como tipo factor o character. Además, elimina uno de los niveles para evitar redundancias. Volviendo al ejemplo anterior, no es necesario almacenar las tres variables, ya que, si *color_rojo* y *color_verde* toman el valor 0, la variable *color_azul* toma necesariamente el valor 1. Si *color_rojo* o *color_verde* toman el valor 1, entonces *color_azul* es necesariamente 0.

```
objeto_recipe <- objeto_recipe %>% step_dummy(all_nominal(), -all_outcomes())
```

Una vez que se ha creado el objeto recipe con todas las transformaciones de preprocesado, se aprenden con los datos de entrenamiento y se aplican a los dos conjuntos.

```
# Se entrena el objeto recipe
trained_recipe <- prep(objeto_recipe, training = datos_train)
trained_recipe
```

```
## Data Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor          7
##
## Training data contained 714 data points and 149 incomplete rows.
##
```

```
## Operations:
##
## Bagged tree imputation for Age_grupo [trained]
## Sparse, unbalanced variable filter removed no terms [trained]
## Centering for SibSp, Parch, Fare [trained]
## Scaling for SibSp, Parch, Fare [trained]
## Dummy variables from Pclass, Sex, Embarked, Age_grupo [trained]
```

```
# Se aplican las transformaciones al conjunto de entrenamiento y de test
datos_train_prep <- bake(trained_recipe, newdata = datos_train)
datos_test_prep  <- bake(trained_recipe, newdata = datos_test)

glimpse(datos_train_prep)
```

```
## Observations: 714
## Variables: 11
## $ Survived          <fct> No, Si, Si, Si, No, No, No, No, Si, Si, Si, ...
## $ SibSp             <dbl> 0.4388679, 0.4388679, -0.4693978, 0.4388679,...
## $ Parch             <dbl> -0.4699865, -0.4699865, -0.4699865, -0.46998...
## $ Fare              <dbl> -0.48548267, 0.74031175, -0.47256109, 0.3922...
## $ Pclass_X2         <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1,...
## $ Pclass_X3         <dbl> 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0,...
## $ Sex_male          <dbl> 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1,...
## $ Embarked_Q        <dbl> 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,...
## $ Embarked_S        <dbl> 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1,...
## $ Age_grupo_anciano <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...
## $ Age_grupo_niño    <dbl> 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0,...
```

Tras el preprocesado de los datos, se han generado un total de 11 variables (10 predictores y la variable respuesta).

Selección de predictores

Cuando se entrena un modelo, es importante incluir como predictores únicamente aquellas variables que están realmente relacionadas con la variable respuesta, ya que son estas las que contienen información útil para el modelo. Incluir un exceso de variables suele conllevar una reducción de la capacidad predictiva del modelo cuando se expone a nuevos datos (*overfitting*). Aunque para este caso de estudio, el número de predictores es considerablemente reducido, la selección de predictores puede suponer la diferencia entre un modelo normal y uno muy bueno, por lo tanto, conviene conocer las herramientas internas de caret para la selección de predictores relevantes.

Algunos algoritmos de *machine learning* (*random forest*, *lasso*, *boosting*...) contienen sus propias estrategias para seleccionar predictores, de ahí que sean modelos tan versátiles. A parte de estos, los métodos para reducir el número de predictores (previo ajuste del modelo) pueden agruparse en dos categorías: métodos *wrapper* y métodos de filtrado.

Métodos wrapper

Los métodos *wrapper* evalúan múltiples modelos, generados mediante la incorporación o eliminación de predictores, con la finalidad de identificar la combinación óptima que consigue maximizar la capacidad del modelo. Pueden entenderse como algoritmos de búsqueda que tratan a los predictores disponibles como valores de entrada y utilizan una métrica del modelo, por ejemplo, su error de predicción, como objetivo de la optimización. El paquete caret incorpora métodos *wrapper* basados en eliminación recursiva, algoritmos genéticos y *simulated annealing*.

Eliminación recursiva de variables

La eliminación recursiva es una estrategia muy práctica para evitar comprobar todas las posibles combinaciones de variables (búsqueda exhaustiva), que es muy costosa computacionalmente. La idea detrás de este proceso puede resumirse con el siguiente algoritmo:

1. Se selecciona un modelo que se empleará para evaluar cada conjunto de predictores. Son muchos los tipos de modelo que se pueden utilizar, la condición necesaria es que ofrezcan la posibilidad de ordenar las variables de mayor a menor importancia. Por ejemplo, si se emplea un modelo lineal, el criterio suele ser el estadístico *t* de los predictores y, si se emplea *random forest*, la reducción en *MSE* conseguida por cada predictor Anexo 1.

2. Ajustar el modelo con el conjunto de entrenamiento incorporando todas las variables disponibles como predictores.

3. Calcular el error del modelo completo con el conjunto de test.

4. Obtener un ranking con la importancia de todos los predictores. El criterio del ranking dependerá del modelo seleccionado en el paso 1.

5. Seleccionar los tamaños de modelo (número de predictores) que se quieren evaluar.

6. Para cada tamaño *n* definido en el paso 5:
   6.1 Seleccionar las top *n* variables del ranking generado en el paso 4.
   6.2 Reajustar el modelo empleando únicamente las top *n* variables.
   6.3 Calcular el error del nuevo modelo con el conjunto de test.

7. Seleccionar el grupo de predictores de tamaño *n* que haya conseguido menor error.

Dado que la selección de predictores es parte del proceso de ajuste de un modelo, el algoritmo anterior viola la norma de que los datos de test no participen en ningún momento en la creación del modelo. Al hacerlo, aumenta el riesgo de *overfitting*. Para excluir los datos de test del proceso de selección de predictores, el algoritmo anterior se puede incluir dentro de un bucle de validación cruzada o *bootstrapping* que solo emplee los datos de entrenamiento. El error final de cada modelo se obtiene agregando los errores obtenidos para cada conjunto de validación o *resampling* Anexo 4. Por ejemplo, si se hace una validación cruzada con 10 *folds*, el bucle exterior tendrá 10 iteraciones, en cada una, un *fold* se emplea como test y el resto como entrenamiento. Si se utiliza *bootstrapping*, por ejemplo 25 *resamples*, el bucle exterior tendrá 25 iteraciones y en cada una de ellas los conjuntos de entrenamiento y test se crean por *bootstrapping*.

El paquete caret automatiza la eliminación recursiva basada en modelos (*recursive feature elimination*) mediante la función rfe(). Existen 4 modelos predefinidos para evaluar cada conjunto de predictores: regresión lineal (lmFuncs), *random forests* (rfFuncs), *naive Bayes* (nbFuncs) y *bagged trees* (treebagFuncs), aunque puede aceptar cualquiera de los algoritmos de *machine learning* que engloba caret Anexo 1

```
# ELIMINACIÓN RECURSIVA MEDIANTE RANDOM FOREST Y BOOTSTRAPPING
# ==============================================================================

# Se paraleliza el proceso para que sea más rápido. El número de cores debe
# seleccionarse en función del ordenador que se está empleando.
library(doMC)
registerDoMC(cores = 4)

# Tamaño de los conjuntos de predictores analizados
subsets <- c(3:11)

# Número de resamples para el proceso de bootstrapping
repeticiones <- 30

# Se crea una semilla para cada repetición de validación. Esto solo es necesario si
# se quiere asegurar la reproducibilidad de los resultados, ya que la validación
# cruzada y el bootstrapping implican selección aleatoria.

# El número de semillas necesarias depende del número total de repeticiones:
# Se necesitan B+1 elementos donde B es el número total de particiones (CV) o
# resampling (bootstrapping). Los primeros B elementos deben ser vectores formados
# por M números enteros, donde M es el número de modelos ajustados, que en este
# caso se corresponde con el número de tamaños. El último elemento solo necesita un
# único número para ajustar el modelo final.
set.seed(123)
seeds <- vector(mode = "list", length = repeticiones + 1)
for (i in 1:repeticiones) {
  seeds[[i]] <- sample.int(1000, length(subsets))
}
seeds[[repeticiones + 1]] <- sample.int(1000, 1)

# Se crea un control de entrenamiento donde se define el tipo de modelo empleado
# para la selección de variables, en este caso random forest, la estrategia de
# resampling, en este caso bootstrapping con 30 repeticiones, y las semillas para
# cada repetición. Con el argumento returnResamp = "all" se especifica que se
# almacene la información de todos los modelos generados en todas las repeticiones.
ctrl_rfe <- rfeControl(functions = rfFuncs, method = "boot", number = repeticiones,
                       returnResamp = "all", allowParallel = TRUE, verbose = FALSE,
                       seeds = seeds)
```

```
# Se ejecuta la eliminación recursiva de predictores
set.seed(342)
rf_rfe <- rfe(Survived ~ ., data = datos_train_prep,
              sizes = subsets,
              metric = "Accuracy",
              # El accuracy es la proporción de clasificaciones correctas
              rfeControl = ctrl_rfe,
              ntree = 500)
# Dentro de rfe() se pueden especificar argumentos para el modelo empleado, por
# ejemplo, el hiperparámetro ntree=500.

# Se muestra una tabla resumen con los resultados
rf_rfe
```

```
##
## Recursive feature selection
##
## Outer resampling method: Bootstrapped (30 reps)
##
## Resampling performance over subset size:
##
##  Variables Accuracy  Kappa AccuracySD KappaSD Selected
##          3   0.7913 0.5475    0.02120 0.05095
##          4   0.8178 0.6027    0.02519 0.05283
##          5   0.8190 0.6060    0.02405 0.05156
##          6   0.8188 0.6055    0.02425 0.05216
##          7   0.8210 0.6096    0.02009 0.04377
##          8   0.8160 0.5985    0.02045 0.04330
##          9   0.8217 0.6139    0.02205 0.05090
##         10   0.8247 0.6199    0.02252 0.05006        *
##
## The top 5 variables (out of 10):
##    Sex_male, Fare, Age_grupo_niño, Pclass_X3, SibSp
```

```
# El objeto rf_rfe almacena en optVariables las variables del mejor modelo.
rf_rfe$optVariables
```

```
##  [1] "Sex_male"          "Fare"              "Age_grupo_niño"
##  [4] "Pclass_X3"         "SibSp"             "Parch"
##  [7] "Embarked_S"        "Pclass_X2"         "Age_grupo_anciano"
## [10] "Embarked_Q"
```

El proceso de eliminación recursiva de variables basada en *random forest* identifica como mejor modelo el formado por los 10 predictores (todos). Además, muestra las 5 variables con mayor influencia. Para interpretar correctamente la información mostrada por pantalla, conviene analizar con detalle todas las métricas calculadas en el proceso de rfe().

Para cada uno de los tamaños analizados (*subsets*), el modelo se ha ajustado y evaluado 30 veces, cada vez con unos conjuntos de entrenamiento y validación distintos creados mediante

bootstrapping. Se dispone por lo tanto de 30 valores de *accuracy* y *kappa* por tamaño. Por defecto, se emplea el promedio de *accuracy* para identificar el mejor conjunto de predictores.

```
# Valores de accuracy y kappa para cada tamaño de modelo en cada resample.
rf_rfe$resample %>% select(1, 2, 3, 8) %>% head(8)
```

```
##   Variables  Accuracy     Kappa   Resample
## 1         3 0.8294574 0.6152908 Resample01
## 2         4 0.8565891 0.6805007 Resample01
## 3         5 0.8449612 0.6502643 Resample01
## 4         6 0.8294574 0.6209936 Resample01
## 5         7 0.8294574 0.6228571 Resample01
## 6         8 0.8410853 0.6511444 Resample01
## 7         9 0.8294574 0.6265298 Resample01
## 8        10 0.8333333 0.6359042 Resample01
```

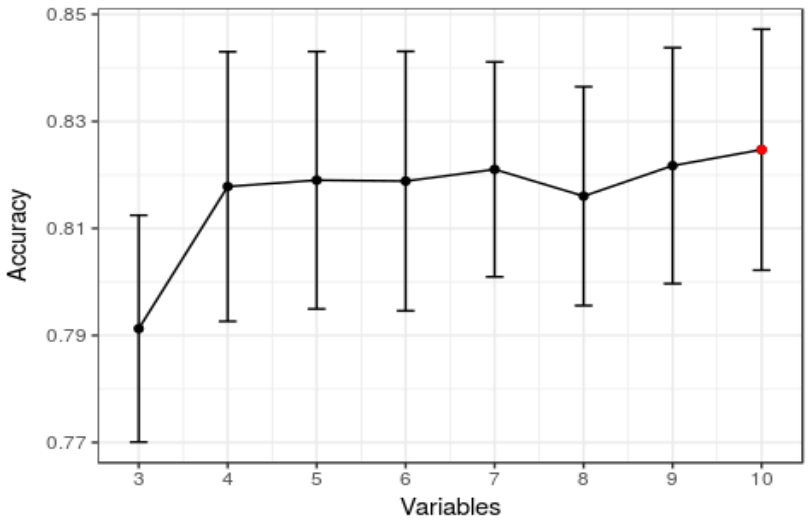
```
# Métricas promedio de cada tamaño
rf_rfe$resample %>% group_by(Variables) %>%
                    summarise(media_accuracy = mean(Accuracy),
                              media_kappa = mean(Kappa)) %>%
                    arrange(desc(media_accuracy))
```

```
## # A tibble: 8 x 3
##   Variables media_accuracy media_kappa
##       <int>          <dbl>       <dbl>
## 1        10          0.825       0.620
## 2         9          0.822       0.614
## 3         7          0.821       0.610
## 4         5          0.819       0.606
## 5         6          0.819       0.606
## 6         4          0.818       0.603
## 7         8          0.816       0.599
## 8         3          0.791       0.548
```

Los valores coinciden con los mostrados al imprimir el objeto rf_rfe.

La siguiente imagen representa la evolución del *accuracy* estimado en función del número de predictores incluido en el modelo.

```
ggplot(data = rf_rfe$results, aes(x = Variables, y = Accuracy)) +
  geom_line() +
  scale_x_continuous(breaks  = unique(rf_rfe$results$Variables)) +
  geom_point() +
  geom_errorbar(aes(ymin = Accuracy - AccuracySD, ymax = Accuracy + AccuracySD),
                width = 0.2) +
  geom_point(data = rf_rfe$results %>% slice(which.max(Accuracy)),
             color = "red") +
  theme_bw()
```



Aunque el máximo *accuracy* se consigue con los 10 predictores, la mejora a partir de 4 predictores es mínima. Acorde al principio de parsimonia, según el cual, de entre un conjunto de modelos con la misma capacidad predictiva, el mejor es el más simple, solo son necesarios 4 predictores.

Además, tras ajustar cada modelo, se recalcula la influencia de cada variable. De esta forma, para cada tamaño de modelo, se obtiene un ranking de la importancia promedio de las variables. Todos los resultados del proceso se almacenan dentro de un *dataframe* llamado *variables*.

```
head(rf_rfe$variables, 10)
```

```
##           No        Si   Overall              var Variables   Resample
## 1  73.919395 73.919395 73.919395         Sex_male        10 Resample01
## 2  33.862203 33.862203 33.862203             Fare        10 Resample01
## 3  26.260147 26.260147 26.260147    Age_grupo_niño        10 Resample01
## 4  24.644747 24.644747 24.644747         Pclass_X3        10 Resample01
## 5  19.812851 19.812851 19.812851             SibSp        10 Resample01
## 6  19.211171 19.211171 19.211171             Parch        10 Resample01
## 7  12.894667 12.894667 12.894667        Embarked_S        10 Resample01
## 8  10.640907 10.640907 10.640907         Pclass_X2        10 Resample01
## 9   9.697091  9.697091  9.697091 Age_grupo_anciano        10 Resample01
## 10  6.392278  6.392278  6.392278        Embarked_Q        10 Resample01
```

A partir de estos datos, se puede obtener el *ranking* de importancia promedio de los predictores para cualquiera de los tamaños, por ejemplo, véase el *ranking* para el modelo con 10 predictores.

```
rf_rfe$variables %>% filter(Variables == 10) %>% group_by(var) %>%
                     summarise(media_influencia = mean(Overall),
                               sd_influencia = sd(Overall)) %>%
                     arrange(desc(media_influencia))
```

```
## # A tibble: 10 x 3
##    var               media_influencia sd_influencia
##    <chr>                        <dbl>         <dbl>
##  1 Sex_male                      66.4          5.80
##  2 Fare                          30.8          2.65
##  3 Age_grupo_niño                26.4          3.61
##  4 Pclass_X3                     23.9          2.23
##  5 SibSp                         19.6          2.53
##  6 Parch                         16.2          2.18
##  7 Embarked_S                    15.6          3.07
##  8 Pclass_X2                     11.3          1.37
##  9 Age_grupo_anciano              7.77         4.12
## 10 Embarked_Q                     7.11         1.72
```

Las primeras 5 variables se corresponden con las Top 5 variables mostradas en la tabla del rf_rfe.

Algoritmo genético

Los algoritmos genéticos son métodos de optimización que pueden emplearse para encontrar la combinación de predictores que consigue maximizar la capacidad predictiva de un modelo. Su funcionamiento se basa en la teoría evolutiva de selección natural propuesta por Darwin: los individuos de una población se reproducen generando nuevos descendientes cuyas características son combinación de las de los progenitores, más ciertas mutaciones. De todos ellos, únicamente los individuos con mejores características sobreviven y pueden reproducirse de nuevo, transmitiendo así sus características. Tras múltiples generaciones, se alcanza un equilibrio en el que se han seleccionado los mejores individuos. Esta idea puede aplicarse a la selección de predictores mediante el siguiente algoritmo:

---

1. Establecer el tamaño de la población inicial m, el número de generaciones, criterios de parada, la probabilidad de mutación p_m y el tipo de modelo empleado para evaluar las combinaciones de predictores.

2. Se crea una población inicial de tamaño m en la que los individuos son vectores binarios que representan diferentes combinaciones aleatorias de predictores, es decir, vectores con una posición para cada predictor disponible, cuyo valor puede ser 1/0.

3. Para cada generación, si no se alcanza un criterio de parada, repetir:

   3.1 Ajustar un modelo por cada individuo de la población y, mediante validación, estimar una métrica que permita cuantificar como de bueno es el modelo, por ejemplo, *accuracy*. Este valor se considera como la fortaleza o *fitness* del individuo.

   3.2 Para *m/2* cruces (se realizan solo *m/2* cruces para mantener constante el tamaño de la población en cada generación) repetir:

      3.2.1 Seleccionar dos individuos de la población, donde la probabilidad de selección es proporcional al valor de la métrica calculada en el paso 3.1.

      3.2.2 Cruce de los individuos: se intercambia la primera mitad del vector de un individuo con la segunda mitad del otro, y viceversa. De esta forma, se generan dos nuevos individuos con nuevas combinaciones que pasan a la siguiente generación.

      3.2.3 Mutaciones: aleatoriamente, modificar los valores binarios de cada posición en cada nuevo individuo, con una probabilidad p_m. Introducir esta aleatoriedad es importante para evitar caer en óptimos locales, normalmente se emplea un valor de *0.05*.

El paquete caret automatiza la selección de variables por algoritmo genético mediante la función gafs(). A continuación, se aplica esta función empleando 10 generaciones, un tamaño poblacional de 10 individuos, un modelo *random forest* como método de evaluación y validación cruzada con 5 particiones. A pesar de lo pequeños que son los números, el proceso completo implica ajustar *10 x 10 x 5 = 500* modelos. Esto pone de manifiesto una de las limitaciones de los algoritmos genéticos, su alto coste computacional. A pesar de ello, pueden lograr resultados muy buenos si se dispone de capacidad para ejecutarlos.

```
# Paralelización
# ==============================================================================
library(doMC)
registerDoMC(cores = 4)

# Control de entrenamiento
# ==============================================================================
```

```
ga_ctrl <- gafsControl(functions = rfGA,
                       method = "cv",
                       number = 5,
                       allowParallel = TRUE,
                       verbose = FALSE)

# Selección de predictores
# ================================================================================
set.seed(10)
rf_ga <- gafs(x = datos_train_prep[, -1], y = datos_train_prep$Survived,
              iters = 10,
              popSize = 10,
              gafsControl = ga_ctrl)

rf_ga
```

```
##
## Genetic Algorithm Feature Selection
##
## 714 samples
## 10 predictors
## 2 classes: 'No', 'Si'
##
## Maximum generations: 10
## Population per generation: 10
## Crossover probability: 0.8
## Mutation probability: 0.1
## Elitism: 0
##
## Internal performance values: Accuracy, Kappa
## Subset selection driven to maximize internal Accuracy
##
## External performance values: Accuracy, Kappa
## Best iteration chose by maximizing external Accuracy
## External resampling method: Cross-Validated (5 fold)
##
## During resampling:
##   * the top 5 selected variables (out of a possible 10):
##     Age_grupo_niño (100%), Pclass_X3 (100%), Sex_male (100%), SibSp (100%),
Age_grupo_anciano (80%)
##   * on average, 8.2 variables were selected (min = 7, max = 9)
##
## In the final search using the entire training set:
##    * 10 features selected at iteration 1 including:
##      SibSp, Parch, Fare, Pclass_X2, Pclass_X3 ...
##    * external performance at this iteration is
##
##    Accuracy       Kappa
##      0.8333      0.6338
```

```
rf_ga$optVariables
```

```
##  [1] "SibSp"             "Parch"             "Fare"
##  [4] "Pclass_X2"         "Pclass_X3"         "Sex_male"
##  [7] "Embarked_Q"        "Embarked_S"        "Age_grupo_anciano"
## [10] "Age_grupo_niño"
```

La selección por algoritmo genético identifica como mejor modelo el formado por los 10 predictores. Estos resultados coinciden con los obtenidos mediante el método de *eliminación recursiva*.

La tabla de resultados `rf_ga$external` muestra información detallada sobre el mejor modelo de cada generación (*Iter*), para cada una de las repeticiones.

```
# Accuracy media en cada generación
rf_ga$external %>% group_by(Iter) %>% summarize(accuracy_media = mean(Accuracy))
```

```
## # A tibble: 10 x 2
##     Iter accuracy_media
##    <int>          <dbl>
##  1     1          0.833
##  2     2          0.824
##  3     3          0.821
##  4     4          0.818
##  5     5          0.817
##  6     6          0.822
##  7     7          0.819
##  8     8          0.819
##  9     9          0.819
## 10    10          0.822
```

Métodos de filtrado

Los métodos basados en filtrado evalúan la relevancia de los predictores fuera del modelo para, posteriormente, incluir únicamente aquellos que pasan un determinado criterio. Se trata por lo tanto de analizar la relación que tiene cada predictor con la variable respuesta. Por ejemplo, en problemas de clasificación con predictores continuos, se puede aplicar un ANOVA a cada predictor para identificar aquellos que varían dependiendo de la variable respuesta. Finalmente, se incorporan al modelo aquellos predictores con un *p-value* inferior a un determinado límite o los *n* mejores. Al igual que con los métodos *wrapper*, para evitar que la selección esté excesivamente influenciada por los datos de entrenamiento (*overfitting*), es conveniente repetir el proceso varias veces mediante validación cruzada o *bootstrapping*.

Al igual que con los métodos de eliminación recursiva, caret incorpora modelos predefinidos para la evaluación de los predictores filtrados (lmSBF, rfSBF, treebagSBF, ldaSBF y nbSBF), además, ofrece la de la posibilidad de emplear cualquier otro modelo (caretFuns). Anexo 2

Los test estadísticos que sbf() emplea por defecto para cuantificar la relación entre los predictores y la variable respuesta son: *ANOVA* (cuando la variable respuesta es cualitativa) y *GAMS* (cuando la variable respuesta es continua). En ambos casos, el límite para la selección es *p-value = 0.05*. Anexo 3

```
# FILTRADO DE PREDICTORES MEDIANTE ANOVA, RANDOM FOREST Y CV-REPETIDA
# ==============================================================================

# Se paraleliza para que sea más rápido
library(doMC)
registerDoMC(cores = 4)

# Se crea una semilla para cada partición y cada repetición: el vector debe
# tener B+1 semillas donde B = particiones * repeticiones.
particiones = 10
repeticiones = 5
set.seed(123)
seeds <- sample.int(1000, particiones * repeticiones + 1)

# Control del filtrado
ctrl_filtrado <- sbfControl(functions = rfSBF, method = "repeatedcv",
                            number = particiones, repeats = repeticiones,
                            seeds = seeds, verbose = FALSE,
                            saveDetails = TRUE, allowParallel = TRUE)
set.seed(234)
rf_sbf <- sbf(Survived ~ ., data = datos_train_prep,
              sbfControl = ctrl_filtrado,
              # argumentos para el modelo de evaluación
              ntree = 500)

rf_sbf
```

```
##
## Selection By Filter
##
## Outer resampling method: Cross-Validated (10 fold, repeated 5 times)
##
## Resampling performance:
##
##  Accuracy  Kappa AccuracySD KappaSD
##    0.8188 0.5968    0.03979 0.09121
##
```

```
## Using the training set, 7 variables were selected:
##    Parch, Fare, Pclass_X2, Pclass_X3, Sex_male...
##
## During resampling, the top 5 selected variables (out of a possible 7):
##    Embarked_S (100%), Fare (100%), Pclass_X3 (100%), Sex_male (100%),
Age_grupo_niño (98%)
##
## On average, 6.9 variables were selected (min = 5, max = 7)
```

```
rf_sbf$optVariables
```

```
## [1] "Parch"          "Fare"           "Pclass_X2"      "Pclass_X3"
## [5] "Sex_male"       "Embarked_S"     "Age_grupo_niño"
```

Empleando la selección por filtrado, se identifican como óptimos 7 predictores: Parch, Fare, Pclass_X2, Pclass_X3, Sex_male, Embarked_S y Age_grupo_niño. El porcentaje que acompaña a las *top 5 selected variables* se corresponde con el porcentaje de repeticiones (*resamples*) en las que la variable ha sido seleccionada.

Se almacenan en una variable los predictores filtrados para emplearlos posteriormente en el ajuste de los modelos.

```
predictores_filtrados <- rf_sbf$optVariables
```

Comparación de métodos

Ambas estrategias, *wrapper* y filtrado, tienen ventajas y desventajas. Los métodos de filtrado son computacionalmente más rápidos por lo que suelen ser la opción factible cuando hay cientos o miles de predictores, sin embargo, el criterio de selección no está directamente relacionado con la efectividad del modelo. Además, en la mayoría de casos, los métodos de filtrado evalúan cada predictor de forma individual, por lo que no contemplan interacciones y pueden incorporar predictores redundantes (correlacionados). Los métodos *wrapper*, además de ser computacionalmente más costosos, para evitar *overfitting*, necesitan recurrir a validación cruzada o *bootstrapping*, por lo que requieren un número alto de observaciones. A pesar de ello, si se cumplen las condiciones, suelen conseguir una mejor selección.

Creación de un modelo predictivo

Una vez que los datos han sido preprocesados y los predictores seleccionados, el siguiente paso es emplear un algoritmo de *machine learning* que permita crear un modelo capaz de representar los patrones presentes en los datos de entrenamiento y generalizarlos a nuevas observaciones. Encontrar el mejor modelo no es fácil, existen multitud de algoritmos, cada uno con unas características propias y con distintos parámetros que deben ser ajustados. Por lo general, las etapas seguidas para obtener un buen modelo son:

Ajuste/entrenamiento: consiste en aplicar un algoritmo de *machine learning* a los datos de entrenamiento para que el modelo aprenda.

Evaluación/validación: el objetivo de un modelo predictivo no es ser capaz de predecir observaciones que ya se conocen, sino nuevas observaciones que el modelo no ha visto. Para poder estimar el error que comete un modelo es necesario recurrir a estrategias de validación, entre las que destacan conjunto de test, bootstrap y validación cruzada. Anexo 4

Optimización de hiperparámetros: muchos algoritmos de *machine learning* contienen en sus ecuaciones uno o varios parámetros que no se aprenden con los datos, a estos se les conoce como hiperparámetros. Por ejemplo, SVM lineal tiene el hiperparámetro de coste *C*. No existe forma de conocer de antemano cuál es el valor exacto de un hiperparámetro que da lugar al mejor modelo, por lo que se tiene que recurrir a estrategias de validación para comparar distintos valores.

Predicción: una vez creado el modelo, este se emplea para predecir nuevas observaciones.

Es a lo largo de todo este proceso donde más destacan las funcionalidades ofrecidas por caret, permitiendo emplear la misma sintaxis para ajustar, optimizar, evaluar y predecir un amplio abanico de modelos variando únicamente el nombre del algoritmo. Aunque caret permite todo esto con apenas unas pocas líneas de código, son muchos los argumentos que pueden ser adaptados, cada uno con múltiples posibilidades. Con el objetivo exponer de mejor cada una de las opciones, en lugar de crear directamente un modelo final, se muestran ejemplos de menor a mayor complejidad.

Entrenamiento del modelo

Todos los modelos incorporados en el paquete caret se entrenan con la función train(). Entre los argumentos de esta función destacan:

- formula: la fórmula del modelo que se quiere crear.
- x, y: en lugar de una fórmula, se pueden pasar por separado los valores de los predictores y de la variable respuesta.
- method: el nombre del algoritmo que se desea emplear (listado).
- metric: las métricas empleadas para evaluar la capacidad predictiva del modelo. Por defecto, se emplea *accuracy* para problemas de clasificación y *RMSE* para regresión.[Anexo 5]
- trControl: especificaciones adicionales sobre la forma de llevar a cabo el entrenamiento del modelo.
- ...: argumentos propios del algoritmo empleado.

En primer lugar, se ajusta un modelo basado en una máquina vector soporte lineal que prediga la supervivencia del pasajero en función de todos los predictores disponibles tras el preprocesado. A excepción del nombre del algoritmo, todos los demás argumentos de la función train() y los propios de svmLinear se dejan por defecto.

```
# Predictores: Survived, SibSp, Parch, Fare, Pclass_X2, Pclass_X3, Sex_male,
# Embarked_Q, Embarked_S, Age_grupo_anciano, Age_grupo_niño
modelo_svmlineal <- train(Survived ~ .,
                          method = "svmLinear",
                          data = datos_train_prep)
```

El objeto devuelto por train() contiene múltiples elementos y mucha información. Por el momento, se analiza únicamente $finalModel, que es el modelo final obtenido tras el ajuste.

```
modelo_svmlineal$finalModel
```

```
## Support Vector Machine object of class "ksvm"
##
## SV type: C-svc  (classification)
##  parameter : cost C = 1
##
## Linear (vanilla) kernel function.
##
## Number of Support Vectors : 311
##
## Objective Function Value : -300.1059
## Training error : 0.203081
```

Al imprimir el objeto $finalModel se muestra el tipo de modelo creado, el valor de los hiperparámetros (más detalles sobre esto en la siguiente sección) e información adicional, entre ella, el error de entrenamiento (*training error*). Este error se corresponde con el error que comete el modelo al predecir las mismas observaciones con las que se ha entrenado, un 20% en este caso.

Evaluación del modelo mediante resampling

La finalidad última de un modelo es predecir la variable respuesta en observaciones futuras o en observaciones que el modelo no ha "visto" antes. El error mostrado en $finalModel se corresponde con el error que comete el modelo al predecir observaciones que sí ha "visto", por lo tanto, no es una estimación realista de cómo se comporta el modelo ante nuevas observaciones (el error de entrenamiento suele ser demasiado optimista). Para conseguir una estimación más certera, y antes de recurrir al conjunto de test, se pueden emplear estrategias de validación basadas en *resampling*. caret incorpora los métodos *boot*, *boot632*, *optimism_boot*, *boot_all*, *cv*, *repeatedcv*, *LOOCV*, *LGOCV* *Anexo4*. Cada uno funciona internamente de forma distinta, pero todos ellos se basan en la idea de ajustar y evaluar el modelo múltiples veces con distintos subconjuntos creados a partir de los datos de entrenamiento, obteniendo en cada repetición una estimación del error. El promedio de todas las estimaciones tiende a converger en el valor real del error de test.

Para especificar el tipo de validación, así como el número de repeticiones, se crea un control de entrenamiento mediante la función trainControl(), que se pasa al argumento trControl de la función train(). A efectos prácticos, cuando se aplican métodos de *resampling* hay que tener en cuenta dos cosas: el coste computacional que implica ajustar múltiples veces un modelo, cada vez con un subconjunto de datos distinto, y la reproducibilidad en la creación de las particiones. caret permite paralelizar el proceso para que sea más rápido y establecer semillas para asegurar que cada *resampling* o partición pueda crearse de nuevo con exactamente las mismas observaciones.

Se ajusta de nuevo una máquina vector soporte lineal, esta vez con validación cruzada repetida para estimar su error.

```
# PARALELIZACIÓN DE PROCESO
#==============================================================================
# Se paraleliza para que sea más rápido. El número de cores activados depende del
# las características del ordenador donde se ejecute el código.
library(doMC)
registerDoMC(cores = 4)
```

```r
# NÚMERO DE REPETICIONES Y SEMILLAS PARA CADA REPETICIÓN
#==============================================================================
# Es este caso se recurre a validación cruzada repetida como método de validación.
# Número de particiones y repeticiones
particiones  <- 10
repeticiones <- 5

# Las semillas solo son necesarias si se quiere asegurar la reproducibilidad de
# los resultados, ya que la validación cruzada y el bootstrapping implican
# selección aleatoria. Las semillas se almacenan en una lista con B+1 elementos
# donde B depende del método de validación empleado:
#
#   "cv": B es el número total de particiones
#   "repeatedcv": B es el número de particiones por el número de repeticiones.
#   "boot": B es el número de resamples.
#   "LOGOCV":   B es el número de repeticiones.
#
# Los primeros B elementos deben ser vectores formados por M números enteros,
# donde M es el número de modelos ajustados en cada partición o repetición,
# es decir, el total de hiperparámetros comparados. El último elemento (B+1) solo
# necesita un único número para ajustar el modelo final.
set.seed(123)
seeds <- vector(mode = "list", length = (particiones * repeticiones) + 1)
for (i in 1:(particiones * repeticiones)) {
  # Cada elemento de la lista, excepto el último tiene que tener tantas semillas
  # como hiperparámetros analizados. En este caso, se emplea el valor por
  # defecto C = 1, por lo que cada elemento de seeds está formada por un
  # único valor.
  seeds[[i]] <- sample.int(1000, 1)
}
# La última semilla se emplea para ajustar el modelo final con todas las
observaciones.
seeds[[(particiones * repeticiones) + 1]] <- sample.int(1000, 1)

# DEFINICIÓN DEL ENTRENAMIENTO
#==============================================================================
control_train <- trainControl(method = "repeatedcv", number = particiones,
                              repeats = repeticiones, seeds = seeds,
                              returnResamp = "all", verboseIter = FALSE,
                              allowParallel = TRUE)

# AJUSTE DEL MODELO
# =============================================================================
set.seed(342)
modelo_svmlineal <- train(Survived ~ ., data = datos_train_prep,
                          method = "svmLinear",
                          metric = "Accuracy",
                          trControl = control_train)
modelo_svmlineal
```

```
## Support Vector Machines with Linear Kernel
##
## 714 samples
##  10 predictors
##   2 classes: 'No', 'Si'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 5 times)
## Summary of sample sizes: 642, 643, 642, 643, 642, 643, ...
## Resampling results:
##
##   Accuracy  Kappa
##   0.795849  0.5579386
##
## Tuning parameter 'C' was held constant at a value of 1
```

Una validación cruzada con 10 particiones y 5 repeticiones, implica ajustar y evaluar el modelo *10 x 5 = 50* veces, cada vez con una partición distinta, más un último ajuste con todos los datos de entrenamiento para crear el modelo final. Al imprimir el objeto devuelto por train() se muestra información sobre el algoritmo empleado, el método de *resampling* y el valor promedio de las métricas de validación, en este caso, al ser un problema de clasificación, se emplea por defecto *accuracy*. Además, si se indica en el control de entrenamiento returnResamp = "all", se almacenan los resultados obtenidos en cada una de las iteraciones, lo que permite un análisis más detallado. Véase la distribución de los valores de *accuracy*.

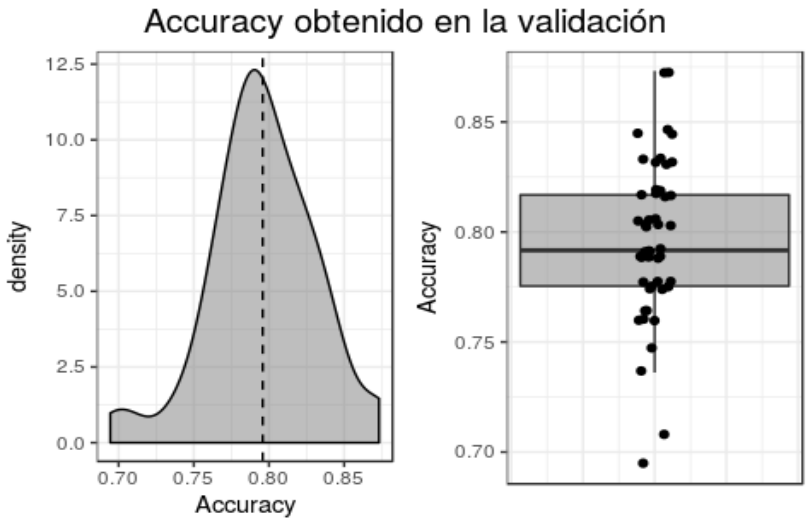
```
# Valores de validación (Accuracy, Kappa) obtenidos en cada partición y repetición.
modelo_svmlineal$resample %>% head(10)
```

```
##     Accuracy     Kappa C    Resample
## 1  0.7777778 0.5135135 1 Fold01.Rep1
## 2  0.8028169 0.5628848 1 Fold02.Rep1
## 3  0.7777778 0.5135135 1 Fold03.Rep1
## 4  0.7887324 0.5485375 1 Fold04.Rep1
## 5  0.7638889 0.5064516 1 Fold05.Rep1
## 6  0.7887324 0.5351375 1 Fold06.Rep1
## 7  0.8732394 0.7291225 1 Fold07.Rep1
## 8  0.7887324 0.5351375 1 Fold08.Rep1
## 9  0.7746479 0.5149445 1 Fold09.Rep1
## 10 0.7777778 0.5200000 1 Fold10.Rep1
```

```
summary(modelo_svmlineal$resample$Accuracy)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.6944  0.7754  0.7917  0.7958  0.8169  0.8732
```

```
p1 <- ggplot(data = modelo_svmlineal$resample, aes(x = Accuracy)) +
      geom_density(alpha = 0.5, fill = "gray50") +
      geom_vline(xintercept = mean(modelo_svmlineal$resample$Accuracy),
                 linetype = "dashed") +
      theme_bw()
p2 <- ggplot(data = modelo_svmlineal$resample, aes(x = 1, y = Accuracy)) +
      geom_boxplot(outlier.shape = NA, alpha = 0.5, fill = "gray50") +
      geom_jitter(width = 0.05) +
      labs(x = "") +
      theme_bw() +
      theme(axis.text.x = element_blank(), axis.ticks.x = element_blank())

final_plot <- ggarrange(p1, p2)
final_plot <- annotate_figure(
                final_plot,
                top = text_grob("Accuracy obtenido en la validación", size = 15))
final_plot
```



El *accuracy* promedio estimado mediante validación cruzada repetida es de *0.79*, el modelo predice correctamente la supervivencia de los pasajeros un 79% de las veces. Este valor será contrastado más adelante cuando se calcule el *accuracy* del modelo con el conjunto de test.

Optimización de hiperparámetros (parameter tuning)

Muchos modelos, entre ellos *SVM*, contienen parámetros que no pueden aprenderse a partir de los datos de entrenamiento y, por lo tanto, deben de ser establecidos por el analista. A estos se les conoce como hiperparámetros. Los resultados de un modelo pueden depender en gran medida del valor que tomen sus hiperparámetros, sin embargo, no se puede conocer de antemano cuál es el adecuado. Aunque con la práctica, los especialistas en *machine learning* ganan intuición sobre qué valores pueden funcionar mejor en cada problema, no hay reglas fijas. La forma más común de encontrar los valores óptimos es probando diferentes posibilidades.

---

1. Escoger un conjunto de valores para el o los hiperparámetros.

2. Para cada valor (combinación de valores si hay más de un hiperparámetro), entrenar el modelo y estimar su error mediante un método de validación.[Anexo 4]

3. Finalmente, se ajusta de nuevo el modelo, esta vez con todos los datos de entrenamiento y con los mejores hiperparámetros encontrados.

El modelo *svmLinear* empleado hasta ahora tienen un hiperparámetro llamado coste (*C*). Este hiperparámetro controla la penalización que se aplica a las clasificaciones erróneas cuando se entrena el modelo. Si su valor es alto, el modelo resultante es más flexible y se ajusta mejor a las observaciones de entrenamiento, pero con el riesgo de *overfitting*. caret permite explorar diferentes valores de hiperparámetros en el momento de entrenar el modelo. Los valores de los hiperparámetros se pasan en formato de *dataframe* al argumento tuneGrid de la función train().

Se vuelve a ajustar un modelo *svmLinear* con diferentes valores de *C* y se emplea validación cruzada repetida para identificar con cuál de ellos se obtienen mejores resultados.

```
# PARALELIZACIÓN DE PROCESO
#==============================================================================
library(doMC)
registerDoMC(cores = 4)

# HIPERPARÁMETROS, NÚMERO DE REPETICIONES Y SEMILLAS PARA CADA REPETICIÓN
#==============================================================================
particiones  <- 10
repeticiones <- 5
```

```r
hiperparametros <- data.frame(C = c(0.001, 0.01, 0.1, 0.5, 1, 10))

set.seed(123)
seeds <- vector(mode = "list", length = (particiones * repeticiones) + 1)
for (i in 1:(particiones * repeticiones)) {
  seeds[[i]] <- sample.int(1000, nrow(hiperparametros))
}
seeds[[(particiones * repeticiones) + 1]] <- sample.int(1000, 1)

# DEFINICIÓN DEL ENTRENAMIENTO
#==============================================================================
control_train <- trainControl(method = "repeatedcv", number = particiones,
                              repeats = repeticiones, seeds = seeds,
                              returnResamp = "all", verboseIter = FALSE,
                              allowParallel = TRUE)

# AJUSTE DEL MODELO
# =============================================================================
set.seed(342)
modelo_svmlineal <- train(Survived ~ ., data = datos_train_prep,
                          method = "svmLinear",
                          tuneGrid = hiperparametros,
                          metric = "Accuracy",
                          trControl = control_train)
modelo_svmlineal
```

```
## Support Vector Machines with Linear Kernel
##
## 714 samples
##  10 predictors
##   2 classes: 'No', 'Si'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 5 times)
## Summary of sample sizes: 642, 643, 642, 643, 642, 643, ...
## Resampling results across tuning parameters:
##
##   C      Accuracy   Kappa
##   1e-03  0.6316745  0.05024495
##   1e-02  0.7862911  0.53795356
##   1e-01  0.7964006  0.55904668
##   5e-01  0.7969562  0.56043954
##   1e+00  0.7958490  0.55793859
##   1e+01  0.7958451  0.55803897
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was C = 0.5.
```

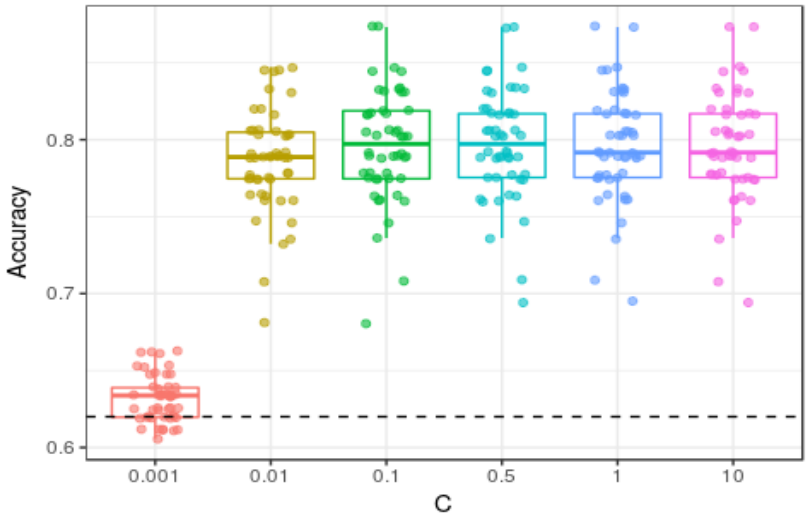
Como se ha empleado validación cruzada con 10 particiones y 5 repeticiones, `caret` ajusta el modelo *10 x 5 = 50* veces por cada valor de *C*. Promediando los 50 valores de *accuracy* obtenidos para cada valor del hiperparámetro, se identifica cual es el mejor (valores mostrados en la tabla). Finalmente, se reajusta el modelo empleando todas las observaciones de entrenamiento y el mejor valor del hiperparámetro, este modelo se almacena en `$finalModel`. En este caso, de entre los valores de *C* probados, *C = 0.5* consigue los mejores resultados con un *accuracy* de 0.797. Si varios valores consiguen el mismo resultado, se selecciona el que da lugar al modelo más sencillo.

Si se especifica en el control de entrenamiento `returnResamp = "all"`, se almacenan los resultados obtenidos en cada iteración para cada valor del hiperparámetro, lo que permite un análisis más detallado.

```
modelo_svmlineal$resample %>% head()
```

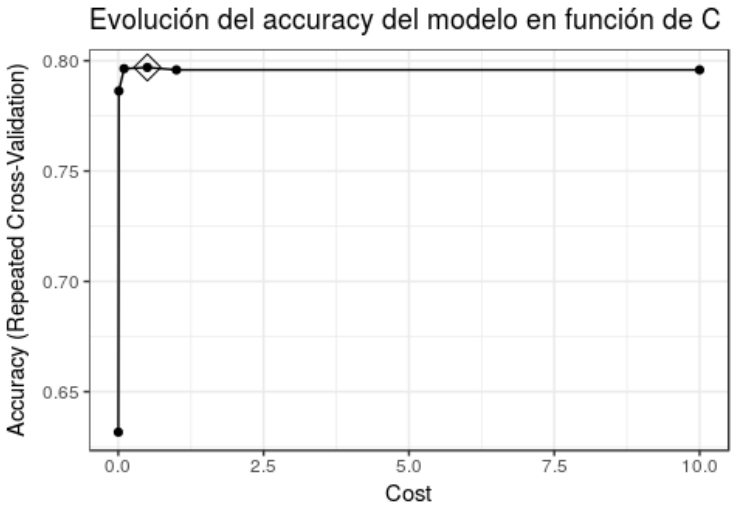
```
##    Accuracy      Kappa     C    Resample
## 1 0.6250000 0.04330709 1e-03 Fold01.Rep1
## 2 0.7777778 0.51351351 1e-02 Fold01.Rep1
## 3 0.7777778 0.51351351 1e-01 Fold01.Rep1
## 4 0.7777778 0.51351351 5e-01 Fold01.Rep1
## 5 0.7777778 0.51351351 1e+00 Fold01.Rep1
## 6 0.7777778 0.51351351 1e+01 Fold01.Rep1
```

```
ggplot(data = modelo_svmlineal$resample,
       aes(x = as.factor(C), y = Accuracy, color = as.factor(C))) +
  geom_boxplot(outlier.shape = NA, alpha = 0.6) +
  geom_jitter(width = 0.2, alpha = 0.6) +
  # Línea horizontal en el accuracy basal
  geom_hline(yintercept = 0.62, linetype = "dashed") +
  labs(x = "C") +
  theme_bw() + theme(legend.position = "none")
```



Además, caret incorpora un método basado en ggplot2 para visualizar la evolución de los modelos según el valor de los hiperparámetros.

```
ggplot(modelo_svmlineal, highlight = TRUE) +
  labs(title = "Evolución del accuracy del modelo en función de C") +
  theme_bw()
```



Búsqueda de hiperparámetros

A la hora de explorar los posibles valores de los hiperparámetros, se pueden seguir 3 estrategias distintas:

Grid search

Se especifican los valores exactos de los hiperparámetros que se quieren estudiar. Esto es lo que se ha hecho en el ejemplo anterior.

Random search

Si no se tiene ninguna idea sobre qué valores pueden ser adecuados, se puede hacer una búsqueda aleatoria dentro de un determinado rango. Algunos estudios parecen indicar que esta estrategia es superior a *grid search* cuando el modelo tiene múltiples hiperparámetros. caret incorpora esta opción mediante los argumentos search = "random" y tuneLength, este último determina la cantidad de valores generados. Sin embargo, en mi experiencia, se tiene más control creando un *dataframe* con valores aleatorios y pasándolo como argumento a tuneGrid. De esta forma, se evita que se generen automáticamente valores que no tienen sentido o que disparan el tiempo de computación necesario. Por ejemplo, si para *svmLinear* se genera un *C = 1000* el modelo tarda minutos en ser ajustado.

```r
# PARALELIZACIÓN DE PROCESO
#==============================================================================
library(doMC)
registerDoMC(cores = 4)

# HIPERPARÁMETROS, NÚMERO DE REPETICIONES Y SEMILLAS PARA CADA REPETICIÓN
#==============================================================================
particiones  <- 10
repeticiones <- 5

# Hiperparámetros aleatorios
set.seed(123)
hiperparametros <- data.frame(C = runif(n = 5, min = 0.001, max = 20))

set.seed(123)
seeds <- vector(mode = "list", length = (particiones * repeticiones) + 1)
for (i in 1:(particiones * repeticiones)) {
  seeds[[i]] <- sample.int(1000, nrow(hiperparametros))
}
seeds[[(particiones * repeticiones) + 1]] <- sample.int(1000, 1)

# DEFINICIÓN DEL ENTRENAMIENTO
#==============================================================================
control_train <- trainControl(method = "repeatedcv", number = particiones,
                              repeats = repeticiones, seeds = seeds,
                              returnResamp = "all", verboseIter = FALSE,
                              allowParallel = TRUE)

# AJUSTE DEL MODELO
# =============================================================================
set.seed(342)
modelo_svmlineal_random <- train(Survived ~ ., data = datos_train_prep,
                                 method = "svmLinear",
                                 tuneGrid = hiperparametros,
                                 metric = "Accuracy",
                                 trControl = control_train)

modelo_svmlineal_random
```

```
## Support Vector Machines with Linear Kernel
##
## 714 samples
##  10 predictors
##   2 classes: 'No', 'Si'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 5 times)
## Summary of sample sizes: 642, 643, 642, 643, 642, 643, ...
## Resampling results across tuning parameters:
##
```

```
##   C          Accuracy   Kappa
##    5.752263  0.7955673  0.5573988
##    8.180129  0.7955673  0.5573988
##   15.766314  0.7952856  0.5566921
##   17.660465  0.7947222  0.5555782
##   18.809405  0.7958490  0.5579386
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was C = 18.80941.
```

Adaptive Resampling

En las dos estrategias anteriores, todos los hiperparámetros son evaluados en todas las repeticiones antes de poder decidir qué valores son buenos y cuáles no. caret incorpora una estrategia llamada Adaptive Resampling que guía la búsqueda hacia los valores óptimos a medida que se van ajustando los modelos, evitando así muchos ajustes innecesarios y reduciendo el coste computacional. Esta estrategia, aunque prometedora, no siempre funciona.

```
# PARALELIZACIÓN DE PROCESO
#==============================================================================
library(doMC)
registerDoMC(cores = 4)

# HIPERPARÁMETROS, NÚMERO DE REPETICIONES Y SEMILLAS PARA CADA REPETICIÓN
#==============================================================================
particiones  <- 10
repeticiones <- 5

# Número de hiperparámetros aleatorios
n_hiperparametros <- 10

set.seed(123)
seeds <- vector(mode = "list", length = (particiones * repeticiones) + 1)
for (i in 1:(particiones * repeticiones)) {
  seeds[[i]] <- sample.int(1000, n_hiperparametros)
}
seeds[[(particiones * repeticiones) + 1]] <- sample.int(1000, 1)

# DEFINICIÓN DEL ENTRENAMIENTO
#==============================================================================
control_train <- trainControl(method = "adaptive_cv", number = particiones,
                              repeats = repeticiones,
                              adaptive = list(min = 5, alpha = 0.05,
                                              method = "gls", complete = TRUE),
                              returnResamp = "all", verboseIter = FALSE,
                              allowParallel = TRUE, search = "random")
```

```
# AJUSTE DEL MODELO
# ========================================================================
set.seed(342)
modelo_svmlineal_adaptative <- train(Survived ~ ., data = datos_train_prep,
                                     method = "svmLinear", trControl = control_train,
                                     tuneLength = 5)

modelo_svmlineal_adaptative
```

```
## Support Vector Machines with Linear Kernel
##
## 714 samples
##  10 predictors
##   2 classes: 'No', 'Si'
##
## No pre-processing
## Resampling: Adaptively Cross-Validated (10 fold, repeated 5 times)
## Summary of sample sizes: 642, 643, 642, 643, 642, 643, ...
## Resampling results across tuning parameters:
##
##   C            Accuracy   Kappa      Resamples
##     0.2060751  0.7966784  0.5597499  50
##    22.9018044  0.7961379  0.5584516   7
##   337.3203713  0.7786255  0.5208073   6
##   357.7548716  0.7739306  0.5105531   6
##   940.0455030  0.7808751  0.5249279   6
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was C = 0.2060751.
```

Predicción

Una vez que el modelo ha sido ajustado, con la función predict() del paquete caret, se pueden predecir nuevos datos utilizando el objeto devuelto por train(). En concreto, la función predict() utiliza el modelo almacenado en $finalModel, que es el mejor modelo de entre los ajustados en el proceso de optimización. Los argumentos de la función son:

models: un modelo o lista de modelos creados mediante train().

newdata: un *dateframe* con nuevas observaciones.

type: tipo de predicción. En modelos de clasificación puede ser *"raw"* para obtener directamente la clase predicha o *"prob"* para obtener la probabilidad de cada clase. Algunos

algoritmos no incorporan de forma nativa un cálculo de probabilidades, para asegurar que caret las calcula, se debe indicar en el control de entrenamiento classProbs = TRUE.

```
predicciones_raw <- predict(modelo_svmlineal, newdata = datos_test_prep,
                            type = "raw")
predicciones_raw
```

```
##   [1] Si No No Si No No Si No No No No Si Si No Si No No No No No No No No
##  [24] No Si Si No Si Si No No No Si No No No Si No Si No Si Si No No Si No
##  [47] Si No No No Si No No Si No No Si No Si No Si Si Si Si No Si No Si Si
##  [70] Si No Si No Si Si No No Si No No Si No Si No Si No No Si No No Si No
##  [93] Si Si No No Si No Si No No No Si No No Si No No No No No No Si No Si
## [116] No Si Si No Si Si Si No No No No Si No No No No Si Si Si No No Si No
## [139] No No No Si No No Si No Si No Si No Si Si No Si No Si No No No No Si
## [162] No No Si No No No Si Si No No Si No No Si Si No
## Levels: No Si
```

```
# El algoritmo svmLinear no calcula de forma nativa probabilidades, para obtenerlas
# se reajusta el modelo indicando `classProbs = TRUE`.
particiones  <- 10
repeticiones <- 5
hiperparametros <- expand.grid(C = c(1))

set.seed(123)
seeds <- vector(mode = "list", length = (particiones * repeticiones) + 1)
for (i in 1:(particiones * repeticiones)) {
  seeds[[i]] <- sample.int(1000, nrow(hiperparametros))
}
seeds[[(particiones * repeticiones) + 1]] <- sample.int(1000, 1)

control_train <- trainControl(method = "repeatedcv", number = particiones,
                              repeats = repeticiones, seeds = seeds,
                              returnResamp = "all", verboseIter = FALSE,
                              classProbs = TRUE, allowParallel = TRUE)

set.seed(342)
modelo_svmlineal <- train(Survived ~ ., data = datos_train_prep,
                          method = "svmLinear",
                          tuneGrid = hiperparametros,
                          metric = "Accuracy",
                          trControl = control_train)

predicciones_prob <- predict(modelo_svmlineal, newdata = datos_test_prep,
                             type = "prob")
predicciones_prob %>% head()
```

```
##          No         Si
## 1 0.2636170 0.73638297
## 2 0.9284028 0.07159723
## 3 0.8225891 0.17741094
## 4 0.2880606 0.71193942
## 5 0.8027662 0.19723383
## 6 0.8173368 0.18266320
```

Obtener las probabilidades para cada clase es más informativo, ya que, además de conocer la clase predicha (la que tiene mayor probabilidad), se puede cuantificar la confianza de dicha predicción. Se tiene mucha más seguridad de que la predicción es correcta si las probabilidades son 0.9-0.1, que si son 0.49-0.51. Para problemas de clasificación binaria, se suele emplear un *cutoff* de probabilidad del 0.5, aquella clase que supera este límite es a la que se asigna la clasificación. Sin embargo, en determinados escenarios en los que se quiere minimizar el riesgo de falsos positivos o falsos negativos, este *cutoff* puede ser modificado. Esta es otra ventaja notable de predecir probabilidades en lugar de clases.

Tal y como se describe en secciones posteriores, caret permite obtener las predicciones de un listado de modelos a la vez. En estos casos, es preferible emplear la función extractPrediction(), ya que devuelve los resultados en formato de *dataframe* en lugar de *list*.

```
predicciones <- extractPrediction(
                  models = list(svm = modelo_svmlineal),
                  testX = datos_test_prep[, -1],
                  testY = datos_test_prep$Survived
                  )
predicciones %>% head()
```

```
##   obs pred     model dataType object
## 1  No   No svmLinear Training    svm
## 2  Si   Si svmLinear Training    svm
## 3  Si   Si svmLinear Training    svm
## 4  Si   Si svmLinear Training    svm
## 5  No   No svmLinear Training    svm
## 6  No   No svmLinear Training    svm
```

Error de test

Aunque mediante los métodos de validación (*CV*, *bootstrapping*...) se consiguen buenas estimaciones del error que tiene un modelo al predecir nuevas observaciones, la mejor forma de evaluar un modelo final es prediciendo un conjunto test, es decir, un conjunto de observaciones que se ha mantenido al margen del proceso de entrenamiento y optimización. Dependiendo del problema en cuestión, pueden ser interesantes unas métricas u otras, la función confusionMatrix() calcula las principales. También hay funciones específicas para cada métrica: sensitivity(), specificity(), posPredValue(), negPredValue(), precision(), recall(), y F_meas().

```
confusionMatrix(data = predicciones_raw, reference = datos_test_prep$Survived,
                positive = "Si")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction No Si
##         No 89 17
##         Si 20 51
##
##                Accuracy : 0.791
##                  95% CI : (0.7236, 0.8483)
##     No Information Rate : 0.6158
##     P-Value [Acc > NIR] : 4.548e-07
##
##                   Kappa : 0.5619
##  Mcnemar's Test P-Value : 0.7423
##
##             Sensitivity : 0.7500
##             Specificity : 0.8165
##          Pos Pred Value : 0.7183
##          Neg Pred Value : 0.8396
##              Prevalence : 0.3842
##          Detection Rate : 0.2881
##    Detection Prevalence : 0.4011
##       Balanced Accuracy : 0.7833
##
##        'Positive' Class : Si
##
```

En el apartado de optimización, se estimó, mediante validación cruzada repetida, que el *accuracy* de modelo *svmLinear* con un *C=0.5* era de *0.797*, es decir, el modelo predice correctamente el 79.7% de las observaciones, un valor casi idéntico al obtenido con el conjunto de test (*0.791*).

En problemas de clasificación, se emplea con frecuencia el porcentaje de predicciones erróneas (1-*accuracy*), también llamado error de clasificación. Esta medida no se muestra en el resultado de confusionMatrix(), pero que es muy sencilla de calcular.

```
# Error de test
error_test <- mean(predicciones_raw != datos_test_prep$Survived)
paste("El error de test del modelo:", round(error_test*100, 2), "%")
```

```
## [1] "El error de test del modelo: 20.9 %"
```

Modelos

En los siguientes apartados se entrenan diferentes modelos de *machine learning* con el objetivo de compararlos e identificar el que mejor resultado obtiene prediciendo la supervivencia de los pasajeros. Tal y como se mencionó al inicio del documento, caret hace una llamada a los modelos implementados en otros paquetes, por lo que, para conocer en detalle cómo funciona cada uno, se debe consultar la documentación del paquete correspondiente. Se puede encontrar un listado de todos los modelos disponibles, así como el paquete de origen y los hiperparámetros en este link.

K-Nearest Neighbor (kNN)

K-Nearest Neighbor es uno de los algoritmos de *machine learning* más simples. Se fundamenta en la idea de identificar observaciones en el conjunto de entrenamiento que se asemejen a la observación de test (observaciones vecinas) y asignarle como valor predicho la clase predominante entre dichas observaciones. A pesar de su sencillez, en muchos escenarios consigue resultados aceptables.

Ajuste, optimización y validación del modelo

El método *knn* de caret emplea la función knn3() con un único hiperparámetro:

- k: número de observaciones vecinas empleadas.

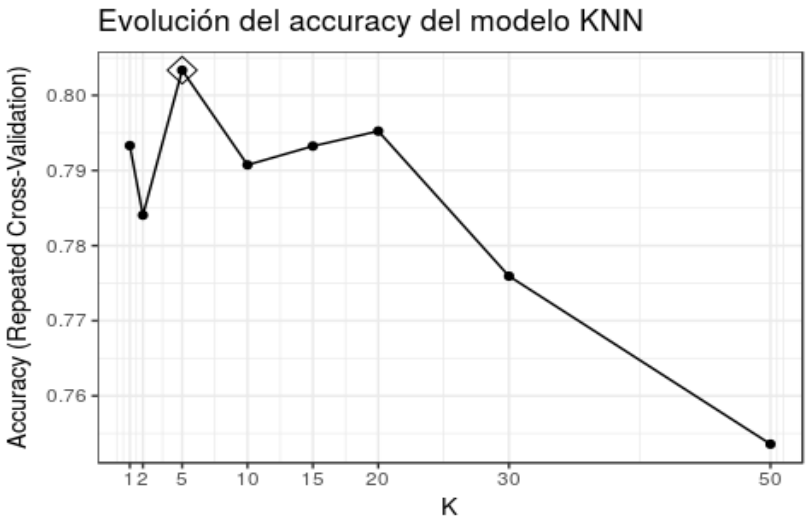
```
# PARALELIZACIÓN DE PROCESO
#==============================================================================
library(doMC)
registerDoMC(cores = 4)

# HIPERPARÁMETROS, NÚMERO DE REPETICIONES Y SEMILLAS PARA CADA REPETICIÓN
#==============================================================================
particiones  <- 10
repeticiones <- 5

# Hiperparámetros
hiperparametros <- data.frame(k = c(1, 2, 5, 10, 15, 20, 30, 50))

set.seed(123)
seeds <- vector(mode = "list", length = (particiones * repeticiones) + 1)
```

```
for (i in 1:(particiones * repeticiones)) {
  seeds[[i]] <- sample.int(1000, nrow(hiperparametros))
}
seeds[[(particiones * repeticiones) + 1]] <- sample.int(1000, 1)

# DEFINICIÓN DEL ENTRENAMIENTO
#==============================================================================
control_train <- trainControl(method = "repeatedcv", number = particiones,
                              repeats = repeticiones, seeds = seeds,
                              returnResamp = "final", verboseIter = FALSE,
                              allowParallel = TRUE)
# AJUSTE DEL MODELO
# ==============================================================================
set.seed(342)
modelo_knn <- train(Survived ~ ., data = datos_train_prep,
                    method = "knn",
                    tuneGrid = hiperparametros,
                    metric = "Accuracy",
                    trControl = control_train)
modelo_knn
```

```
## k-Nearest Neighbors
##
## 714 samples
##  10 predictors
##   2 classes: 'No', 'Si'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 5 times)
## Summary of sample sizes: 642, 643, 642, 643, 642, 643, ...
## Resampling results across tuning parameters:
##
##   k   Accuracy   Kappa
##    1  0.7933138  0.5602402
##    2  0.7840571  0.5409828
##    5  0.8033490  0.5820327
##   10  0.7907473  0.5523747
##   15  0.7932512  0.5567901
##   20  0.7952426  0.5571446
##   30  0.7759390  0.5085140
##   50  0.7535759  0.4472254
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was k = 5.
```

```
# REPRESENTACIÓN GRÁFICA
# ==============================================================================
ggplot(modelo_knn, highlight = TRUE) +
  scale_x_continuous(breaks = hiperparametros$k) +
```

```
  labs(title = "Evolución del accuracy del modelo KNN", x = "K") +
  theme_bw()
```



Con un modelo *kNN* con *k=5* se consigue un *accuracy* de validación promedio del 80.3%.

Naive Bayes

Empleando el teorema de Bayes, el algoritmo de *Naive Bayes* calcula las probabilidades condicionales de que una observación pertenezca a cada una de las clases dadas unas evidencias (valores de los predictores). El término *naive* (ingenuo en inglés) se debe a que el algoritmo asume que las variables son independientes, es decir, que el valor que toman no está influenciado por las demás. Aunque en la práctica, esta asunción raramente es cierta, permite calcular la probabilidad cuando hay múltiples predictores simplemente multiplicando las probabilidades individuales de cada uno (regla de eventos independientes). A pesar de que es solo una aproximación, puede resultar muy efectiva. Más información sobre *Naive Bayes* en este link.

Ajuste, optimización y validación del modelo

El método *nb* de caret emplea la función NaiveBayes() del paquete klaR con tres hiperparámetros:

- usekernel: TRUE para emplear un kernel que estime la densidad o FALSE para asumir una distribución de densidad gaussiana.
- fL: factor de corrección de *Laplace*, o para no aplicar ninguna corrección.
- adjust: parámetro pasado a la función density si usekernel = TRUE.

```r
# PARALELIZACIÓN DE PROCESO
#==============================================================================
library(doMC)
registerDoMC(cores = 4)

# HIPERPARÁMETROS, NÚMERO DE REPETICIONES Y SEMILLAS PARA CADA REPETICIÓN
#==============================================================================
particiones  <- 10
repeticiones <- 5

# Hiperparámetros
hiperparametros <- data.frame(usekernel = FALSE, fL = 0 , adjust = 0)

set.seed(123)
seeds <- vector(mode = "list", length = (particiones * repeticiones) + 1)
for (i in 1:(particiones * repeticiones)) {
  seeds[[i]] <- sample.int(1000, nrow(hiperparametros))
}
seeds[[(particiones * repeticiones) + 1]] <- sample.int(1000, 1)

# DEFINICIÓN DEL ENTRENAMIENTO
#==============================================================================
control_train <- trainControl(method = "repeatedcv", number = particiones,
                              repeats = repeticiones, seeds = seeds,
                              returnResamp = "final", verboseIter = FALSE,
                              allowParallel = TRUE)
# AJUSTE DEL MODELO
# ============================================================================
set.seed(342)
modelo_nb <- train(Survived ~ ., data = datos_train_prep,
                   method = "nb",
                   tuneGrid = hiperparametros,
                   metric = "Accuracy",
                   trControl = control_train)
modelo_nb
```

```
## Naive Bayes
##
## 714 samples
##  10 predictors
##   2 classes: 'No', 'Si'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 5 times)
## Summary of sample sizes: 642, 643, 642, 643, 642, 643, ...
## Resampling results:
##
##   Accuracy   Kappa
##   0.7980869  0.5688785
```

```
##
## Tuning parameter 'fL' was held constant at a value of 0
## Tuning
##  parameter 'usekernel' was held constant at a value of FALSE
##
## Tuning parameter 'adjust' was held constant at a value of 0
```

Empleando un modelo *Naive Bayes* con *usekernel = FALSE, fL = 0, adjust = 0*, se consigue un *accuracy* promedio de validación del 79.8%.

Regresión logística

Información detallada sobre regresión logística en Regresión logística simple y múltiple.

Ajuste, optimización y validación del modelo

El método *glm* de caret emplea la función glm() del paquete básico de R. Este algoritmo no tiene ningún hiperparámetro pero, para que efectúe una regresión logística, hay que indicar family = "binomial".

```
# PARALELIZACIÓN DE PROCESO
#==============================================================================
library(doMC)
registerDoMC(cores = 4)

# HIPERPARÁMETROS, NÚMERO DE REPETICIONES Y SEMILLAS PARA CADA REPETICIÓN
#==============================================================================
particiones  <- 10
repeticiones <- 5

# Hiperparámetros
hiperparametros <- data.frame(parameter = "none")
set.seed(123)
seeds <- vector(mode = "list", length = (particiones * repeticiones) + 1)
for (i in 1:(particiones * repeticiones)) {
  seeds[[i]] <- sample.int(1000, nrow(hiperparametros))
}
seeds[[(particiones * repeticiones) + 1]] <- sample.int(1000, 1)
```

```
# DEFINICIÓN DEL ENTRENAMIENTO
#==============================================================================
control_train <- trainControl(method = "repeatedcv", number = particiones,
                              repeats = repeticiones, seeds = seeds,
                              returnResamp = "final", verboseIter = FALSE,
                              allowParallel = TRUE)
# AJUSTE DEL MODELO
# ==============================================================================
set.seed(342)
modelo_logistic <- train(Survived ~ ., data = datos_train_prep,
                         method = "glm",
                         tuneGrid = hiperparametros,
                         metric = "Accuracy",
                         trControl = control_train, family = "binomial")
modelo_logistic
```

```
## Generalized Linear Model
##
## 714 samples
##  10 predictors
##   2 classes: 'No', 'Si'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 5 times)
## Summary of sample sizes: 642, 643, 642, 643, 642, 643, ...
## Resampling results:
##
##   Accuracy   Kappa
##   0.7994914  0.5673381
```

```
summary(modelo_logistic$finalModel)
```

```
##
## Call:
## NULL
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.8827  -0.5503  -0.4331   0.5798   2.3939
##
## Coefficients:
##                   Estimate Std. Error z value Pr(>|z|)
## (Intercept)        2.55793    0.34586   7.396 1.41e-13 ***
## SibSp             -0.53019    0.15708  -3.375 0.000737 ***
## Parch             -0.20261    0.11101  -1.825 0.067984 .
## Fare               0.19888    0.14648   1.358 0.174537
## Pclass_X2         -0.74873    0.32233  -2.323 0.020185 *
## Pclass_X3         -1.90843    0.30619  -6.233 4.58e-10 ***
```

```
## Sex_male          -2.83340    0.23483 -12.066  < 2e-16 ***
## Embarked_Q         0.03173    0.42675   0.074 0.940726
## Embarked_S        -0.38463    0.27078  -1.420 0.155472
## Age_grupo_anciano -0.48896    0.59820  -0.817 0.413700
## Age_grupo_niño     2.16357    0.46090   4.694 2.68e-06 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 950.86  on 713  degrees of freedom
## Residual deviance: 619.70  on 703  degrees of freedom
## AIC: 641.7
##
## Number of Fisher Scoring iterations: 5
```

Empleando un modelo de regresión logística se consigue un *accuracy* promedio de validación del 80.0%.

LDA

Información detallada sobre *LDA* en Análisis discriminante lineal (LDA).

Ajuste, optimización y validación del modelo

El método *lda* de caret emplea la función lda() del paquete MASS. Este algoritmo no tiene ningún hiperparámetro.

```
# PARALELIZACIÓN DE PROCESO
#==============================================================================
library(doMC)
registerDoMC(cores = 4)

# HIPERPARÁMETROS, NÚMERO DE REPETICIONES Y SEMILLAS PARA CADA REPETICIÓN
#==============================================================================
particiones  <- 10
repeticiones <- 5

# Hiperparámetros
hiperparametros <- data.frame(parameter = "none")
```

```
set.seed(123)
seeds <- vector(mode = "list", length = (particiones * repeticiones) + 1)
for (i in 1:(particiones * repeticiones)) {
  seeds[[i]] <- sample.int(1000, nrow(hiperparametros))
}
seeds[[(particiones * repeticiones) + 1]] <- sample.int(1000, 1)

# DEFINICIÓN DEL ENTRENAMIENTO
#==============================================================================
control_train <- trainControl(method = "repeatedcv", number = particiones,
                              repeats = repeticiones, seeds = seeds,
                              returnResamp = "final", verboseIter = FALSE,
                              allowParallel = TRUE)

# AJUSTE DEL MODELO
# =============================================================================
set.seed(342)
modelo_lda <- train(Survived ~ ., data = datos_train_prep,
                    method = "lda",
                    tuneGrid = hiperparametros,
                    metric = "Accuracy",
                    trControl = control_train)
modelo_lda
```

```
## Linear Discriminant Analysis
##
## 714 samples
##  10 predictors
##   2 classes: 'No', 'Si'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 5 times)
## Summary of sample sizes: 642, 643, 642, 643, 642, 643, ...
## Resampling results:
##
##   Accuracy   Kappa
##   0.8008803  0.5692523
```

```
modelo_lda$finalModel
```

```
## Call:
## lda(x, grouping = y)
##
## Prior probabilities of groups:
##        No        Si
## 0.6162465 0.3837535
```

```
##
## Group means:
##          SibSp       Parch       Fare Pclass_X2 Pclass_X3  Sex_male
## No  0.01776287 -0.08146164 -0.2057724 0.1636364 0.6931818 0.8613636
## Si -0.02852432  0.13081432  0.3304375 0.2408759 0.3394161 0.3357664
##    Embarked_Q Embarked_S Age_grupo_anciano Age_grupo_niño
## No 0.08636364  0.7750000        0.02727273     0.05454545
## Si 0.08759124  0.6532847        0.01824818     0.11313869
##
## Coefficients of linear discriminants:
##                           LD1
## SibSp              -0.2853455
## Parch              -0.1380122
## Fare                0.1115099
## Pclass_X2          -0.5491125
## Pclass_X3          -1.2960061
## Sex_male           -2.1605930
## Embarked_Q          0.0288948
## Embarked_S         -0.2392385
## Age_grupo_anciano  -0.4049878
## Age_grupo_niño      1.3971200
```

Empleando como modelo un *LDA*, se consigue un *accuracy* promedio de validación del 80.0%.

Árbol de clasificación simple

Información detallada sobre árboles de clasificación en Árboles de predicción: bagging, random forest, boosting y C5.0.

Ajuste, optimización y validación del modelo

El método *C5.0Tree* de caret emplea la función C5.0.default() del paquete C50 para crear un árbol de clasificación simple. Este algoritmo no tiene ningún hiperparámetro.

```
# PARALELIZACIÓN DE PROCESO
#==============================================================================
library(doMC)
registerDoMC(cores = 4)
```

```
# HIPERPARÁMETROS, NÚMERO DE REPETICIONES Y SEMILLAS PARA CADA REPETICIÓN
#==============================================================================
particiones  <- 10
repeticiones <- 5

# Hiperparámetros
hiperparametros <- data.frame(parameter = "none")

set.seed(123)
seeds <- vector(mode = "list", length = (particiones * repeticiones) + 1)
for (i in 1:(particiones * repeticiones)) {
  seeds[[i]] <- sample.int(1000, nrow(hiperparametros))
}
seeds[[(particiones * repeticiones) + 1]] <- sample.int(1000, 1)

# DEFINICIÓN DEL ENTRENAMIENTO
#==============================================================================
control_train <- trainControl(method = "repeatedcv", number = particiones,
                              repeats = repeticiones, seeds = seeds,
                              returnResamp = "final", verboseIter = FALSE,
                              allowParallel = TRUE)
# AJUSTE DEL MODELO
# =============================================================================
set.seed(342)
modelo_C50Tree <- train(Survived ~ ., data = datos_train_prep,
                    method = "C5.0Tree",
                    tuneGrid = hiperparametros,
                    metric = "Accuracy",
                    trControl = control_train)
modelo_C50Tree
```

```
## Single C5.0 Tree
##
## 714 samples
##  10 predictors
##   2 classes: 'No', 'Si'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 5 times)
## Summary of sample sizes: 642, 643, 642, 643, 642, 643, ...
## Resampling results:
##
##   Accuracy   Kappa
##   0.8131925  0.5929933
```

```
summary(modelo_C50Tree$finalModel)
```

```
## Call:
## C50:::C5.0.default(x = x, y = y, weights = wts)
##
## C5.0 [Release 2.07 GPL Edition]      Mon Apr 23 16:37:55 2018
## -------------------------------
##
## Class specified by attribute `outcome'
##
## Read 714 cases (11 attributes) from undefined.data
##
## Decision tree:
##
## Sex_male <= 0:
## :...Pclass_X3 <= 0: Si (133/7)
## :   Pclass_X3 > 0:
## :   :...Embarked_S <= 0: Si (46/13)
## :       Embarked_S > 0:
## :       :...Fare <= -0.2911804: Si (38/18)
## :           Fare > -0.2911804: No (26/3)
## Sex_male > 0:
## :...Age_grupo_niño > 0:
##     :...SibSp <= 1.347134: Si (16)
##     :   SibSp > 1.347134: No (13)
##     Age_grupo_niño <= 0:
##     :...Fare <= -0.1217642: No (332/33)
##         Fare > -0.1217642:
##         :...Parch > -0.4699865: No (27/5)
##             Parch <= -0.4699865:
##             :...Pclass_X3 > 0: Si (6/1)
##                 Pclass_X3 <= 0:
##                 :...Fare <= -0.119132: Si (4)
##                     Fare > -0.119132: No (73/29)
## Evaluation on training data (714 cases):
##
##      Decision Tree
##    ----------------
##    Size      Errors
##
##      11  109(15.3%)   <<
##
##
##     (a)   (b)    <-classified as
##    ----  ----
##     401    39    (a): class No
##      70   204    (b): class Si
##
##
##  Attribute usage:
##
##  100.00% Sex_male
```

```
##   70.87% Fare
##   65.97% Age_grupo_niño
##   45.66% Pclass_X3
##   15.41% Parch
##   15.41% Embarked_S
##    4.06% SibSp
##
##
## Time: 0.0 secs
```

Empleando como modelo un árbol simple C5.0, se consigue un *accuracy* promedio de validación del 81.3%.

RandomForest

Información detallada sobre *random forest* en Árboles de predicción: bagging, random forest, boosting y C5.0

Ajuste, optimización y validación del modelo

El método *ranger* de caret emplea la función ranger() del paquete ranger. Este algoritmo tiene 3 hiperparámetros:

- mtry: número predictores seleccionados aleatoriamente en cada árbol.
- min.node.size: tamaño mínimo que tiene que tener un nodo para poder ser dividido.
- splitrule: criterio de división.

Aunque caret también incluye el método *rf* con la función rf() del paquete randomForest, este último solo permite optimizar el hiperparámetro mtry.

```
# PARALELIZACIÓN DE PROCESO
#==============================================================================
library(doMC)
registerDoMC(cores = 4)

# HIPERPARÁMETROS, NÚMERO DE REPETICIONES Y SEMILLAS PARA CADA REPETICIÓN
#==============================================================================
particiones  <- 10
repeticiones <- 5
```

```r
# Hiperparámetros
hiperparametros <- expand.grid(mtry = c(3, 4, 5, 7),
                               min.node.size = c(2, 3, 4, 5, 10, 15, 20, 30),
                               splitrule = "gini")
set.seed(123)
seeds <- vector(mode = "list", length = (particiones * repeticiones) + 1)
for (i in 1:(particiones * repeticiones)) {
  seeds[[i]] <- sample.int(1000, nrow(hiperparametros))
}
seeds[[(particiones * repeticiones) + 1]] <- sample.int(1000, 1)

# DEFINICIÓN DEL ENTRENAMIENTO
#==============================================================================
control_train <- trainControl(method = "repeatedcv", number = particiones,
                              repeats = repeticiones, seeds = seeds,
                              returnResamp = "final", verboseIter = FALSE,
                              allowParallel = TRUE)
# AJUSTE DEL MODELO
# =============================================================================
set.seed(342)
modelo_rf <- train(Survived ~ ., data = datos_train_prep,
                   method = "ranger",
                   tuneGrid = hiperparametros,
                   metric = "Accuracy",
                   trControl = control_train,
                   # Número de árboles ajustados
                   num.trees = 500)
modelo_rf
```

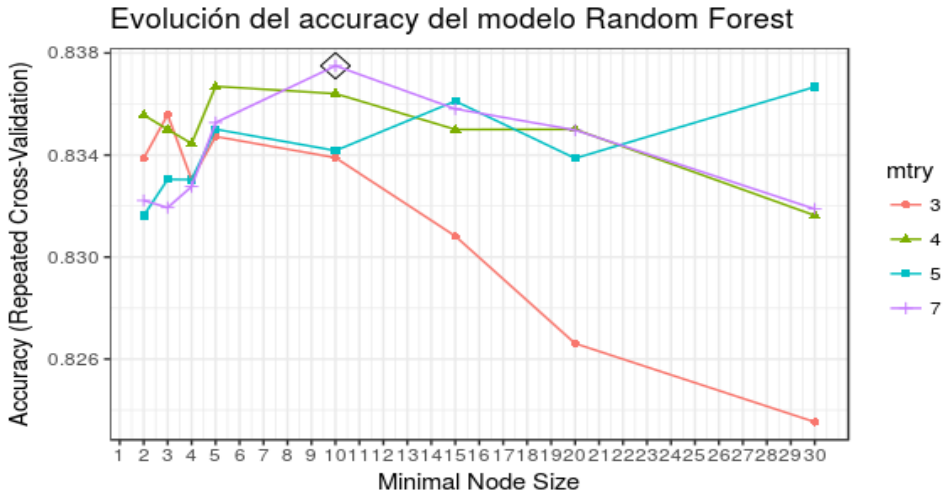
```
## Random Forest
##
## 714 samples
##  10 predictors
##   2 classes: 'No', 'Si'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 5 times)
## Summary of sample sizes: 642, 643, 642, 643, 642, 643, ...
## Resampling results across tuning parameters:
##
##   mtry  min.node.size  Accuracy   Kappa
##   3      2             0.8338732  0.6338623
##   3      3             0.8355829  0.6377688
##   3      4             0.8330321  0.6325584
##   3      5             0.8347261  0.6360054
##   3     10             0.8338967  0.6344485
##   3     15             0.8308216  0.6278330
##   3     20             0.8266119  0.6178842
##   3     30             0.8235329  0.6086024
##   4      2             0.8355712  0.6414031
```

```
##   4      3             0.8350039  0.6401333
##   4      4             0.8344523  0.6388404
##   4      5             0.8366941  0.6434940
##   4     10             0.8364085  0.6430724
##   4     15             0.8350039  0.6405249
##   4     20             0.8350078  0.6401796
##   4     30             0.8316315  0.6304811
##   5      2             0.8316236  0.6346441
##   5      3             0.8330477  0.6378208
##   5      4             0.8330399  0.6375516
##   5      5             0.8350117  0.6421278
##   5     10             0.8341745  0.6402521
##   5     15             0.8361150  0.6437347
##   5     20             0.8338732  0.6392069
##   5     30             0.8366745  0.6439343
##   7      2             0.8322183  0.6395312
##   7      3             0.8319405  0.6379434
##   7      4             0.8327660  0.6393822
##   7      5             0.8352739  0.6442237
##   7     10             0.8375000  0.6489871
##   7     15             0.8358099  0.6446792
##   7     20             0.8349804  0.6433562
##   7     30             0.8318858  0.6354208
##
## Tuning parameter 'splitrule' was held constant at a value of gini
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were mtry = 7, splitrule = gini
##  and min.node.size = 10.
```

```
modelo_rf$finalModel
```

```
## Ranger result
##
## Call:
##  ranger::ranger(dependent.variable.name = ".outcome", data = x,      mtry =
param$mtry, min.node.size = param$min.node.size, splitrule =
as.character(param$splitrule),      write.forest = TRUE, probability = classProbs,
...)
##
## Type:                             Classification
## Number of trees:                  500
## Sample size:                      714
## Number of independent variables:  10
## Mtry:                             7
## Target node size:                 10
## Variable importance mode:         none
## OOB prediction error:             16.39 %
```

```
# REPRESENTACIÓN GRÁFICA
# ==============================================================================
ggplot(modelo_rf, highlight = TRUE) +
  scale_x_continuous(breaks = 1:30) +
  labs(title = "Evolución del accuracy del modelo Random Forest") +
  guides(color = guide_legend(title = "mtry"),
         shape = guide_legend(title = "mtry")) +
  theme_bw()
```



Empleando un modelo *random forest* con mtry = 7, min.node.size = 10 y splitrule = "gini", se consigue un *accuracy* promedio de validación del 83.8%.

Gradient Boosting

Información detallada sobre *boosting* en Árboles de predicción: bagging, random forest, boosting y C5.0

Ajuste, optimización y validación del modelo

El método *gbm* de caret emplea la función gbm() del paquete gbm. Este algoritmo tiene 4 hiperparámetros:

- n.trees: número de iteraciones del algoritmo de *boosting*, es decir, número de modelos que forman el *ensemble*. Cuanto mayor es este valor, más se reduce el error de entrenamiento, pudiendo llegar generarse *overfitting*.

- interaction.depth: complejidad de los árboles empleados como *weak learner*, en concreto, el número total de divisiones que tiene el árbol. Emplear árboles con ente 1 y 6 nodos suele dar buenos resultados.
- shrinkage: este parámetro, también conocido como *learning rate*, controla la influencia que tiene cada modelo sobre el conjunto del *ensemble*. Es preferible mejorar un modelo mediante muchos pasos pequeños que mediante unos pocos grandes. Por esta razón, se recomienda emplear un valor de *shrinkage* tan pequeño como sea posible, teniendo en cuenta que, cuanto menor sea, mayor el número de iteraciones necesarias. Por defecto es de 0.001.
- n.minobsinnode: número mínimo de observaciones que debe tener un nodo para poder ser dividido. Al igual que interaction.depth, permite controlar la complejidad de los *weak learners* basados en árboles.

Además de los hiperparámetros que permite controlar caret, la función gbm() tiene otros dos más que hay que tener en cuenta:

- distribution: determina la función de coste (*loss function*). Algunas de las más utilizadas son: *gaussian* (*squared loss*) para regresión, *bernoulli* para respuestas binarias, *multinomial* para variables respuesta con más de dos clases y *adaboost* para respuestas binarias y que emplea la función exponencial del algoritmo original *AdaBoost*.
- bag.fraction (*subsampling fraction*): fracción de observaciones del set de entrenamiento seleccionadas de forma aleatoria para ajustar cada *weak learner*. Si su valor es de 1, se emplea el algoritmo de *Gradient Boosting*, si es menor que 1, se emplea *Stochastic Gradient Boosting*. Por defecto su valor es de 0.5.

```
# PARALELIZACIÓN DE PROCESO
#==============================================================================
library(doMC)
registerDoMC(cores = 4)

# HIPERPARÁMETROS, NÚMERO DE REPETICIONES Y SEMILLAS PARA CADA REPETICIÓN
#==============================================================================
particiones  <- 10
repeticiones <- 5
# Hiperparámetros
hiperparametros <- expand.grid(interaction.depth = c(1, 2),
                               n.trees = c(500, 1000, 2000),
                               shrinkage = c(0.001, 0.01, 0.1),
                               n.minobsinnode = c(2, 5, 15))
set.seed(123)
seeds <- vector(mode = "list", length = (particiones * repeticiones) + 1)
for (i in 1:(particiones * repeticiones)) {
  seeds[[i]] <- sample.int(1000, nrow(hiperparametros))
}
seeds[[(particiones * repeticiones) + 1]] <- sample.int(1000, 1)
```

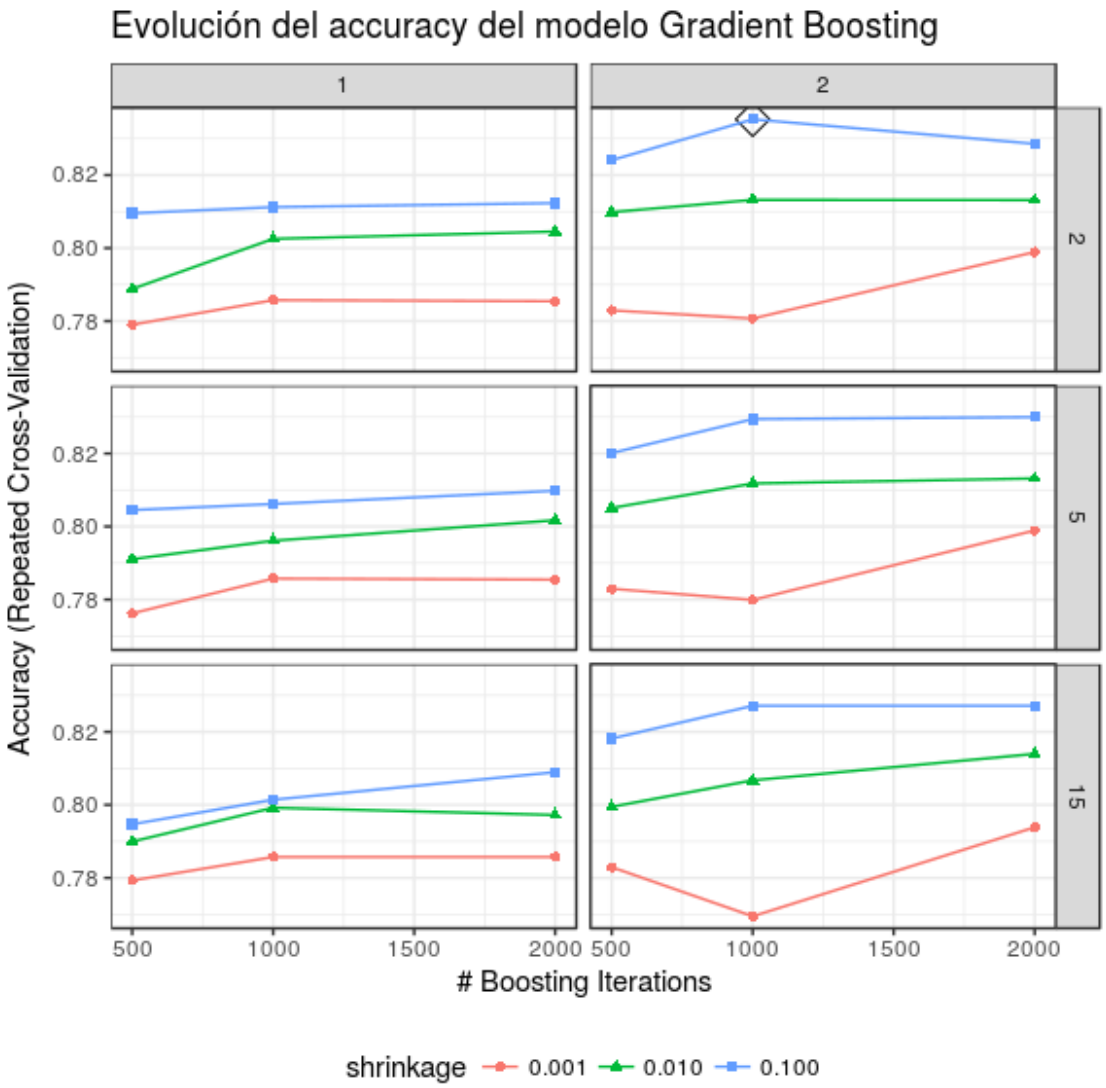
```r
# DEFINICIÓN DEL ENTRENAMIENTO
#=============================================================================
control_train <- trainControl(method = "repeatedcv", number = particiones,
                              repeats = repeticiones, seeds = seeds,
                              returnResamp = "final", verboseIter = FALSE,
                              allowParallel = TRUE)
# AJUSTE DEL MODELO
# ============================================================================
set.seed(342)
modelo_boost <- train(Survived ~ ., data = datos_train_prep,
                      method = "gbm",
                      tuneGrid = hiperparametros,
                      metric = "Accuracy",
                      trControl = control_train,
                      # Número de árboles ajustados
                      distribution = "adaboost",
                      verbose = FALSE)
modelo_boost
```

```
## Stochastic Gradient Boosting
##
## 714 samples
##  10 predictors
##   2 classes: 'No', 'Si'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 5 times)
## Summary of sample sizes: 642, 643, 642, 643, 642, 643, ...
## Resampling results across tuning parameters:
##
##   shrinkage  interaction.depth  n.minobsinnode  n.trees  Accuracy
##   0.001      1                   2               500     0.7790297
##   0.001      1                   2              1000     0.7857316
##   0.001      1                   2              2000     0.7854499
##   0.001      1                   5               500     0.7762128
##   0.001      1                   5              1000     0.7857316
##   0.001      1                   5              2000     0.7854499
##   0.001      1                  15               500     0.7793075
##   0.001      1                  15              1000     0.7857316
##   0.001      1                  15              2000     0.7857316
##   0.001      2                   2               500     0.7829343
##   0.001      2                   2              1000     0.7807081
##   0.001      2                   2              2000     0.7989280
##   0.001      2                   5               500     0.7829343
##   0.001      2                   5              1000     0.7798865
##   0.001      2                   5              2000     0.7989280
##   0.001      2                  15               500     0.7829343
##   0.001      2                  15              1000     0.7694875
##   0.001      2                  15              2000     0.7938850
##   0.010      1                   2               500     0.7888067
```

```
##   0.010      1                   2                1000      0.8025352
##   0.010      1                   2                2000      0.8045266
##   0.010      1                   5                 500      0.7910642
##   0.010      1                   5                1000      0.7961033
##   0.010      1                   5                2000      0.8017058
##   0.010      1                  15                 500      0.7899570
##   0.010      1                  15                1000      0.7992097
##   0.010      1                  15                2000      0.7972379
##   0.010      2                   2                 500      0.8098552
##   0.010      2                   2                1000      0.8132003
##   0.010      2                   2                2000      0.8131573
##   0.010      2                   5                 500      0.8050822
##   0.010      2                   5                1000      0.8117840
##   0.010      2                   5                2000      0.8131573
##   0.010      2                  15                 500      0.7994757
##   0.010      2                  15                1000      0.8067371
##   0.010      2                  15                2000      0.8139828
##   0.100      1                   2                 500      0.8095462
##   0.100      1                   2                1000      0.8112207
##   0.100      1                   2                2000      0.8123200
##   0.100      1                   5                 500      0.8045070
##   0.100      1                   5                1000      0.8061698
##   0.100      1                   5                2000      0.8097926
##   0.100      1                  15                 500      0.7947066
##   0.100      1                  15                1000      0.8014163
##   0.100      1                  15                2000      0.8089789
##   0.100      2                   2                 500      0.8240806
##   0.100      2                   2                1000      0.8352660
##   0.100      2                   2                2000      0.8285446
##   0.100      2                   5                 500      0.8201252
##   0.100      2                   5                1000      0.8294053
##   0.100      2                   5                2000      0.8299570
##   0.100      2                  15                 500      0.8181768
##   0.100      2                  15                1000      0.8271557
##   0.100      2                  15                2000      0.8271674
##   Kappa
##   0.5166983
##   0.5367502
##   0.5360863
##   0.5090923
##   0.5367502
##   0.5360863
##   0.5172698
##   0.5367502
##   0.5367502
##   0.4899315
##   0.5050352
##   0.5590570
##   0.4899315
##   0.5031992
```

```
##   0.5590348
##   0.4899315
##   0.4748830
##   0.5460600
##   0.5417927
##   0.5734055
##   0.5793882
##   0.5461891
##   0.5605769
##   0.5729184
##   0.5452591
##   0.5681149
##   0.5641804
##   0.5871693
##   0.5966833
##   0.5956843
##   0.5776974
##   0.5936203
##   0.5954147
##   0.5650111
##   0.5809224
##   0.5946832
##   0.5888231
##   0.5922447
##   0.5942755
##   0.5777136
##   0.5818648
##   0.5887275
##   0.5572185
##   0.5709685
##   0.5867740
##   0.6187227
##   0.6435155
##   0.6307433
##   0.6109414
##   0.6309865
##   0.6334463
##   0.6034724
##   0.6233648
##   0.6261475
##
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were n.trees = 1000,
##  interaction.depth = 2, shrinkage = 0.1 and n.minobsinnode = 2.
```

```
# REPRESENTACIÓN GRÁFICA
# ==============================================================================
ggplot(modelo_boost, highlight = TRUE) +
  labs(title = "Evolución del accuracy del modelo Gradient Boosting") +
    guides(color = guide_legend(title = "shrinkage"),
           shape = guide_legend(title = "shrinkage")) +
  theme_bw() +
  theme(legend.position = "bottom")
```



Empleando un modelo *boosting* con n.trees = 1000, minobsinnode = 2, interaction.depth = 2 y shrinkage = 0.1 se consigue un *accuracy* promedio de validación del 83.5%.

SVM

Información detallada sobre *SVM* en Máquinas de Vector Soporte (Support Vector Machines, SVMs)

Ajuste, optimización y validación del modelo

El método *svmRadial* de caret emplea la función ksvm() del paquete kernlab. Este algoritmo tiene 2 hiperparámetros:

- sigma: coeficiente del kernel radial.
- C: penalización por violaciones del margen del hiperplano.

```
# PARALELIZACIÓN DE PROCESO
#==============================================================================
library(doMC)
registerDoMC(cores = 4)

# HIPERPARÁMETROS, NÚMERO DE REPETICIONES Y SEMILLAS PARA CADA REPETICIÓN
#==============================================================================
particiones  <- 10
repeticiones <- 5

# Hiperparámetros
hiperparametros <- expand.grid(sigma = c(0.001, 0.01, 0.1, 0.5, 1),
                               C = c(1 , 20, 50, 100, 200, 500, 700))

set.seed(123)
seeds <- vector(mode = "list", length = (particiones * repeticiones) + 1)
for (i in 1:(particiones * repeticiones)) {
  seeds[[i]] <- sample.int(1000, nrow(hiperparametros))
}
seeds[[(particiones * repeticiones) + 1]] <- sample.int(1000, 1)

# DEFINICIÓN DEL ENTRENAMIENTO
#==============================================================================
control_train <- trainControl(method = "repeatedcv", number = particiones,
                              repeats = repeticiones, seeds = seeds,
                              returnResamp = "final", verboseIter = FALSE,
                              allowParallel = TRUE)
```

```r
# AJUSTE DEL MODELO
# ========================================================================
set.seed(342)
modelo_svmrad <- train(Survived ~ ., data = datos_train_prep,
                       method = "svmRadial",
                       tuneGrid = hiperparametros,
                       metric = "Accuracy",
                       trControl = control_train)
modelo_svmrad
```

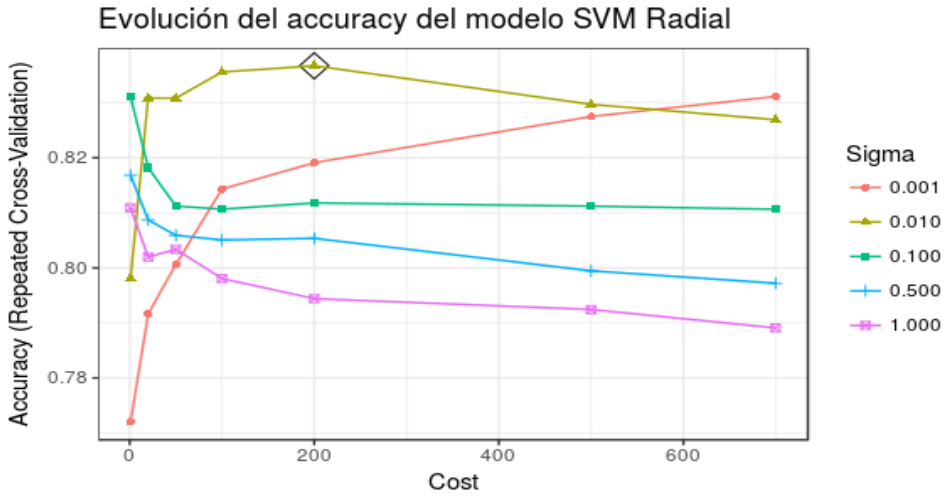
```
## Support Vector Machines with Radial Basis Function Kernel
##
## 714 samples
##  10 predictors
##   2 classes: 'No', 'Si'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 5 times)
## Summary of sample sizes: 642, 643, 642, 643, 642, 643, ...
## Resampling results across tuning parameters:
##
##   sigma  C    Accuracy   Kappa
##   0.001    1  0.7719992  0.4781624
##   0.001   20  0.7916275  0.5480256
##   0.001   50  0.8006103  0.5688424
##   0.001  100  0.8143232  0.5996531
##   0.001  200  0.8191041  0.6090938
##   0.001  500  0.8274961  0.6264161
##   0.001  700  0.8311424  0.6337104
##   0.010    1  0.7981064  0.5631093
##   0.010   20  0.8308490  0.6317733
##   0.010   50  0.8308294  0.6249590
##   0.010  100  0.8355829  0.6328420
##   0.010  200  0.8367058  0.6349823
##   0.010  500  0.8296987  0.6213148
##   0.010  700  0.8269092  0.6160803
##   0.100    1  0.8310994  0.6237852
##   0.100   20  0.8182277  0.6008914
##   0.100   50  0.8112207  0.5865945
##   0.100  100  0.8106612  0.5857181
##   0.100  200  0.8117958  0.5879891
##   0.100  500  0.8112167  0.5873951
##   0.100  700  0.8106416  0.5863056
##   0.500    1  0.8168232  0.5975612
##   0.500   20  0.8087089  0.5836183
##   0.500   50  0.8059155  0.5795853
##   0.500  100  0.8050704  0.5781242
##   0.500  200  0.8053599  0.5792918
##   0.500  500  0.7994523  0.5664956
##   0.500  700  0.7972066  0.5610682
```

```
##   1.000    1  0.8109116  0.5887298
##   1.000   20  0.8019679  0.5733500
##   1.000   50  0.8033685  0.5770781
##   1.000  100  0.7980556  0.5648517
##   1.000  200  0.7944014  0.5553323
##   1.000  500  0.7924296  0.5486441
##   1.000  700  0.7890806  0.5413609
##
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were sigma = 0.01 and C = 200.
```

```
modelo_svmrad$finalModel
```

```
## Support Vector Machine object of class "ksvm"
##
## SV type: C-svc  (classification)
##  parameter : cost C = 200
##
## Gaussian Radial Basis kernel function.
##  Hyperparameter : sigma =  0.01
##
## Number of Support Vectors : 286
##
## Objective Function Value : -46114.84
## Training error : 0.155462
```

```
# REPRESENTACIÓN GRÁFICA
# ==============================================================================
ggplot(modelo_svmrad, highlight = TRUE) +
  labs(title = "Evolución del accuracy del modelo SVM Radial") +
  theme_bw()
```



Empleando un modelo *SVM Radial* con sigma = 0.01 y C = 200, se consigue un *accuracy* promedio de validación del 83.7%.

Redes neuronales (NNET)

Ajuste, optimización y validación del modelo

El método *nnet* de caret emplea la función nnet() del paquete nnet para crear redes neuronales con una capa oculta. Este algoritmo tiene 2 hiperparámetros:

- size: número de neuronas en la capa oculta.
- decay: controla la regularización durante el entrenamiento de la red.

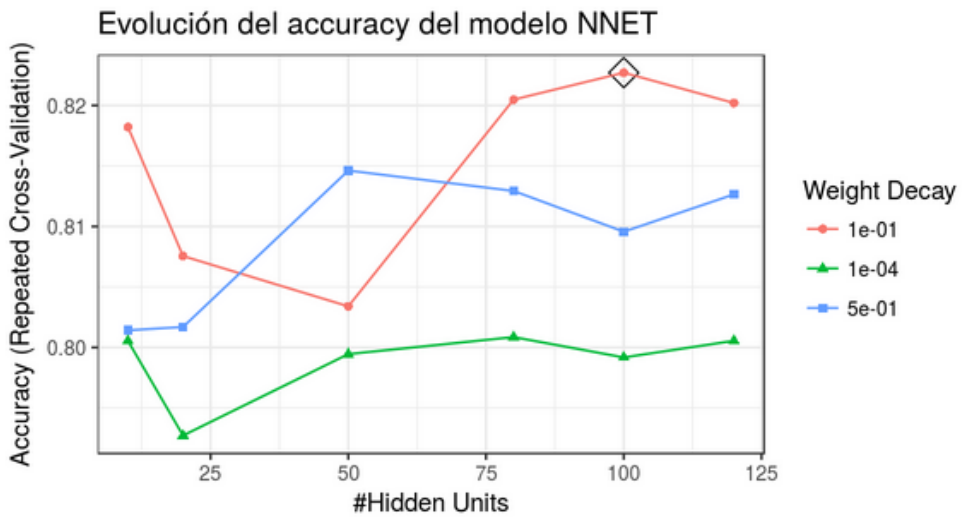
Además de estos hiperparámetros, la función nnet() tiene muchos otros argumentos que controlan el proceso de aprendizaje de la red. Entre ellos cabe destacar MaxNWts, que determina el número máximo de pesos. Por defecto, su valor es 1000, pero puede ocurrir que, si la red tiene muchas neuronas, se necesiten más pesos.

```
# PARALELIZACIÓN DE PROCESO
#==============================================================================
library(doMC)
registerDoMC(cores = 4)

# HIPERPARÁMETROS, NÚMERO DE REPETICIONES Y SEMILLAS PARA CADA REPETICIÓN
#==============================================================================
particiones  <- 10
repeticiones <- 5
# Hiperparámetros
hiperparametros <- expand.grid(size = c(10, 20, 50, 80, 100, 120),
                               decay = c(0.0001, 0.1, 0.5))
set.seed(123)
seeds <- vector(mode = "list", length = (particiones * repeticiones) + 1)
for (i in 1:(particiones * repeticiones)) {
  seeds[[i]] <- sample.int(1000, nrow(hiperparametros))
}
seeds[[(particiones * repeticiones) + 1]] <- sample.int(1000, 1)

# DEFINICIÓN DEL ENTRENAMIENTO
#==============================================================================
control_train <- trainControl(method = "repeatedcv", number = particiones,
                              repeats = repeticiones, seeds = seeds,
                              returnResamp = "final", verboseIter = FALSE,
                              allowParallel = TRUE)
# AJUSTE DEL MODELO
# =============================================================================
set.seed(342)
modelo_nnet <- train(Survived ~ ., data = datos_train_prep,
                     method = "nnet", tuneGrid = hiperparametros,
                     metric = "Accuracy",
```

```
                  trControl = control_train,
                  # Rango de inicialización de los pesos
                  rang = c(-0.7, 0.7),
                  # Se aumenta el número máximo de pesos
                  MaxNWts = 2000,
                  # Para que no se muestre cada iteración por pantalla
                  trace = FALSE)
modelo_nnet
```

```
## Neural Network
## 714 samples
##  10 predictors
##   2 classes: 'No', 'Si'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 5 times)
## Summary of sample sizes: 642, 643, 642, 643, 642, 643, ...
## Resampling results across tuning parameters:
##
##   size  decay  Accuracy   Kappa
##    10   1e-04  0.8005516  0.5693811
##    10   1e-01  0.8182199  0.6020419
##    10   5e-01  0.8014241  0.5665801
##    20   1e-04  0.7927269  0.5505401
##    20   1e-01  0.8075548  0.5795228
##    20   5e-01  0.8016941  0.5666546
##    50   1e-04  0.7994562  0.5683570
##    50   1e-01  0.8033998  0.5726934
##    50   5e-01  0.8146205  0.5986775
##    80   1e-04  0.8008607  0.5701937
##    80   1e-01  0.8204851  0.6069849
##    80   5e-01  0.8129343  0.5941470
##   100   1e-04  0.7991784  0.5677853
##   100   1e-01  0.8227074  0.6121366
##   100   5e-01  0.8095696  0.5870379
##   120   1e-04  0.8005556  0.5652136
##   120   1e-01  0.8202034  0.6063607
##   120   5e-01  0.8126526  0.5940560
##
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were size = 100 and decay = 0.1.
```

```
# REPRESENTACIÓN GRÁFICA
# ==============================================================================
ggplot(modelo_nnet, highlight = TRUE) +
  labs(title = "Evolución del accuracy del modelo NNET") + theme_bw()
```



Empleando un modelo *NNET* con size = 100 y decay = 0.1, se consigue un *accuracy* promedio de validación del 82.3%.

Comparación de modelos

Una vez que se han entrenado y optimizado distintos modelos, se tiene que identificar cuál de ellos consigue mejores resultados para el problema en cuestión, en este caso, predecir la supervivencia de los pasajeros. Con los datos disponibles, existen dos formas de comparar los modelos. Si bien las dos no tienen por qué dar los mismos resultados, son complementarias a la hora de tomar una decisión final.

Métricas de validación

Las métricas obtenidas mediante validación (*cv, bootstrapping*...) son estimaciones de la capacidad que tiene un modelo al predecir nuevas observaciones. Como toda estimación, tiene asociada una varianza. Para poder determinar si un método es superior a otro, no es suficiente con comparar los mínimos (o máximos dependiendo de la métrica) que ha conseguido cada uno, sino que hay que tener en cuenta sus varianzas para determinar si existen evidencias suficientes de superioridad.

Al tratarse de modelos entrenados y validados sobre los mismos datos, mismas particiones y en el mismo orden (siempre que se haya asegurado la reproducibilidad mediante semillas), se pueden emplear métodos estadísticos para datos dependientes. Determinar si uno de los métodos supera al otro equivale a responder a la pregunta de si las diferencias entre pares de datos se alejan significativamente de cero. Cuando se comparan solo dos modelos, algunos contrastes de hipótesis aplicables son:

- Método paramétrico: t-test de datos dependientes.
- Método no paramétrico: test de McNemar o el test de la suma de rangos de Wilcoxon para muestras pareadas.

Si la comparación se hace entre más de dos modelos, entonces pueden emplearse los contrastes de hipótesis:

- Método paramétrico: ANOVA de datos dependientes.
- Método no paramétrico: test de Friedman.

La elección de un método u otro depende de si se satisfacen o no las condiciones necesarias. Por lo general, los test paramétricos requieren de más condiciones, pero, en caso de cumplirse, consiguen mayor poder estadístico (capacidad para detectar diferencias significativas).

La función resamples() permite extraer, de uno o varios modelos creados con train(), las métricas obtenidas para cada repetición del proceso de validación. Es importante tener en cuenta que esta función recupera todos los resultados, por lo que, si no se especifica en el control de entrenamiento returnResamp = "final", se devuelven los valores para todos los hiperparámetros, no solo los del modelo final.

```
modelos <- list(KNN = modelo_knn, NB = modelo_nb, logistic = modelo_logistic,
                LDA = modelo_lda, arbol = modelo_C50Tree, rf = modelo_rf,
                boosting = modelo_boost, SVMradial = modelo_svmrad,
                NNET = modelo_nnet)
resultados_resamples <- resamples(modelos)
resultados_resamples$values %>% head(10)
```

```
##       Resample KNN~Accuracy KNN~Kappa NB~Accuracy  NB~Kappa
## 1  Fold01.Rep1    0.8333333 0.6447368   0.7916667 0.5469799
## 2  Fold01.Rep2    0.8309859 0.6414141   0.7746479 0.5149445
## 3  Fold01.Rep3    0.8591549 0.6968403   0.9154930 0.8126649
## 4  Fold01.Rep4    0.7361111 0.4411765   0.6250000 0.1953642
## 5  Fold01.Rep5    0.8194444 0.6125828   0.7638889 0.4865772
## 6  Fold02.Rep1    0.8309859 0.6414141   0.8028169 0.5693241
## 7  Fold02.Rep2    0.7464789 0.4543126   0.7887324 0.5485375
## 8  Fold02.Rep3    0.7605634 0.4731558   0.7464789 0.4543126
## 9  Fold02.Rep4    0.8028169 0.5875519   0.8309859 0.6414141
## 10 Fold02.Rep5    0.8750000 0.7352941   0.8333333 0.6538462
##    logistic~Accuracy logistic~Kappa LDA~Accuracy LDA~Kappa arbol~Accuracy
## 1          0.7916667      0.5529801    0.7777778 0.5135135      0.8333333
## 2          0.8169014      0.6030108    0.8169014 0.6030108      0.8309859
## 3          0.8591549      0.6877748    0.9014085 0.7830642      0.8732394
## 4          0.6805556      0.2959184    0.6666667 0.2702703      0.7361111
## 5          0.8055556      0.5684932    0.7916667 0.5344828      0.8333333
## 6          0.8028169      0.5628848    0.8169014 0.5971192      0.8309859
## 7          0.7605634      0.4652193    0.7605634 0.4652193      0.7605634
## 8          0.7323944      0.4197849    0.7464789 0.4462738      0.7887324
## 9          0.7887324      0.5549519    0.8169014 0.6087325      0.8450704
## 10         0.7916667      0.5529801    0.7916667 0.5529801      0.8333333
##    arbol~Kappa rf~Accuracy  rf~Kappa boosting~Accuracy boosting~Kappa
## 1    0.6493506   0.8750000 0.7281879         0.8611111      0.7000000
## 2    0.6464730   0.8309859 0.6308492         0.8450704      0.6640860
## 3    0.7251613   0.8732394 0.7251613         0.8732394      0.7291225
## 4    0.3936170   0.7361111 0.4337748         0.7083333      0.3657718
## 5    0.6400000   0.8194444 0.6073826         0.8333333      0.6351351
## 6    0.6137806   0.8309859 0.6308492         0.8591549      0.6923744
## 7    0.4486067   0.8028169 0.5755764         0.7887324      0.5351375
## 8    0.5485375   0.8028169 0.5693241         0.7887324      0.5419355
## 9    0.6591008   0.9014085 0.7862366         0.8169014      0.6142917
## 10   0.6447368   0.8750000 0.7352941         0.9027778      0.7941176
##    SVMradial~Accuracy SVMradial~Kappa NNET~Accuracy NNET~Kappa
## 1           0.8194444       0.5965517     0.8055556  0.5743243
## 2           0.8309859       0.6253298     0.8169014  0.6030108
```

```
## 3           0.8873239       0.7464286     0.8873239  0.7502199
## 4           0.7361111       0.3936170     0.7222222  0.3750000
## 5           0.8055556       0.5563380     0.7916667  0.5344828
## 6           0.8450704       0.6486730     0.8028169  0.5755764
## 7           0.8028169       0.5562500     0.8028169  0.5628848
## 8           0.7746479       0.4850408     0.7746479  0.5077990
## 9           0.8873239       0.7538995     0.9154930  0.8154246
## 10          0.8611111       0.7000000     0.8750000  0.7352941
```

```
# Se trasforma el dataframe devuelto por resamples() para separar el nombre del
# modelo y las métricas en columnas distintas.
metricas_resamples <- resultados_resamples$values %>%
                        gather(key = "modelo", value = "valor", -Resample) %>%
                        separate(col = "modelo", into = c("modelo", "metrica"),
                                 sep = "~", remove = TRUE)
metricas_resamples %>% head()
```

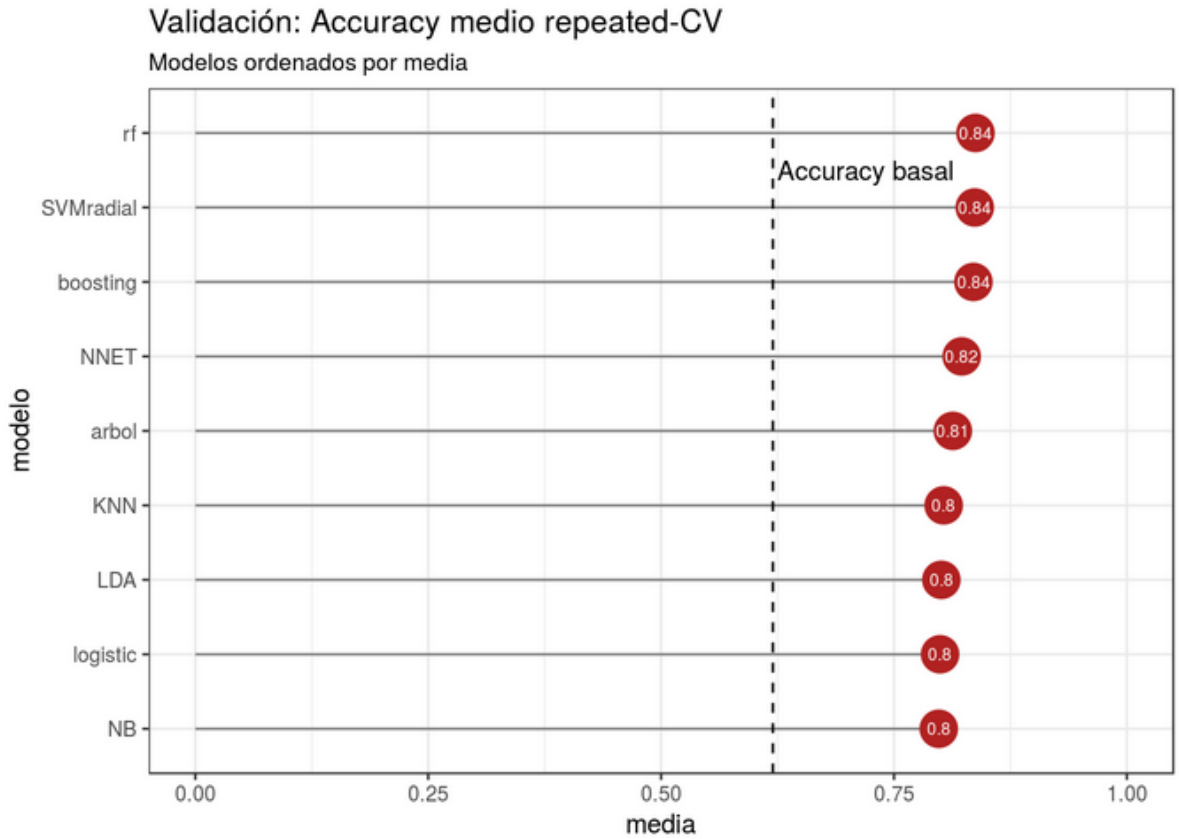
```
##      Resample modelo  metrica     valor
## 1 Fold01.Rep1    KNN Accuracy 0.8333333
## 2 Fold01.Rep2    KNN Accuracy 0.8309859
## 3 Fold01.Rep3    KNN Accuracy 0.8591549
## 4 Fold01.Rep4    KNN Accuracy 0.7361111
## 5 Fold01.Rep5    KNN Accuracy 0.8194444
## 6 Fold02.Rep1    KNN Accuracy 0.8309859
```

Accuracy y Kappa promedio de cada modelo

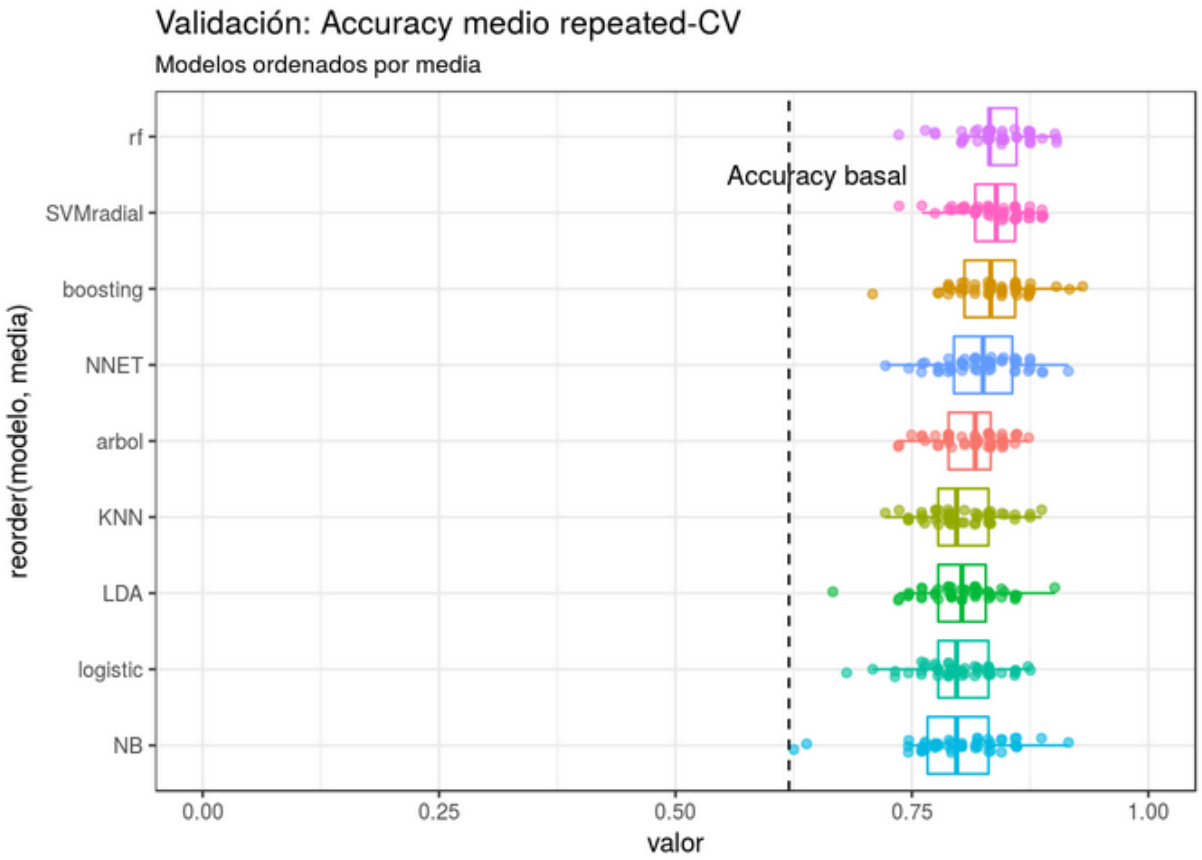
```
metricas_resamples %>%
  group_by(modelo, metrica) %>%
  summarise(media = mean(valor)) %>%
  spread(key = metrica, value = media) %>%
  arrange(desc(Accuracy))
```

```
## # A tibble: 9 x 3
## # Groups:   modelo [9]
##   modelo    Accuracy Kappa
##   <chr>        <dbl> <dbl>
## 1 rf           0.838 0.649
## 2 SVMradial    0.837 0.635
## 3 boosting     0.835 0.644
## 4 NNET         0.820 0.607
## 5 arbol        0.813 0.593
## 6 KNN          0.803 0.582
## 7 LDA          0.801 0.569
## 8 logistic     0.799 0.567
## 9 NB           0.798 0.569
```

```r
metricas_resamples %>%
  filter(metrica == "Accuracy") %>%
  group_by(modelo) %>%
  summarise(media = mean(valor)) %>%
  ggplot(aes(x = reorder(modelo, media), y = media, label = round(media, 2))) +
    geom_segment(aes(x = reorder(modelo, media), y = 0,
                     xend = modelo, yend = media),
                     color = "grey50") +
    geom_point(size = 7, color = "firebrick") +
    geom_text(color = "white", size = 2.5) +
    scale_y_continuous(limits = c(0, 1)) +
    # Accuracy basal
    geom_hline(yintercept = 0.62, linetype = "dashed") +
    labs(title = "Validación: Accuracy medio repeated-CV",
         subtitle = "Modelos ordenados por media",
         x = "modelo") +
    coord_flip() +
    theme_bw()
```



```
metricas_resamples %>% filter(metrica == "Accuracy") %>%
  group_by(modelo) %>%
  mutate(media = mean(valor)) %>%
  ungroup() %>%
  ggplot(aes(x = reorder(modelo, media), y = valor, color = modelo)) +
    geom_boxplot(alpha = 0.6, outlier.shape = NA) +
    geom_jitter(width = 0.1, alpha = 0.6) +
    # Accuracy basal
    geom_hline(yintercept = 0.62, linetype = "dashed") +
    annotate(geom = "text", y = 0.66, x = 8.5, label = "Accuracy basal") +
    theme_bw() +
    labs(title = "Validación: Accuracy medio repeated-CV",
         subtitle = "Modelos ordenados por media") +
    coord_flip() +
    theme(legend.position = "none")
```



Todos los modelos ajustados tienen un acierto en la predicción superior al nivel basal (0.62, línea discontinua). El modelo *random forest* consigue el *accuracy* promedio más alto, seguido

muy de cerca por *SVM radial* y *Boosting*. Para determinar si las diferencias entre ellos son significativas, se recurre a test estadísticos.

Test de Friedman para comparar el accuracy de los modelos

```
matriz_metricas <- metricas_resamples %>% filter(metrica == "Accuracy") %>%
                   spread(key = modelo, value = valor) %>%
                   select(-Resample, -metrica) %>% as.matrix()
friedman.test(y = matriz_metricas)
```

```
##
##  Friedman rank sum test
##
## data:  matriz_metricas
## Friedman chi-squared = 124.02, df = 8, p-value < 2.2e-16
```

Para un nivel de significancia ($\alpha = 0.05$), el test de *Friedman* sí encuentra evidencias para rechazar la hipótesis nula de que los 9 clasificadores consiguen la misma precisión, sin embargo, no determina que par o pares son diferentes. Para identificarlos, se recurre a contrastes *post HOC*.

La función diff() del paquete caret recibe como argumento los resultados de validación de dos o más modelos extraídos con resample() y hace comparaciones por pares aplicando un *t-test pareado* con correcciones por comparaciones múltiples. Esta función no permite mucha flexibilidad en cuanto a las comparaciones, por lo que, una vez extraídos los datos con resample(), suele ser preferible emplear otras funciones disponibles en R.

```
# Comparaciones múltiples con un test suma de rangos de Wilcoxon
# ==============================================================================

metricas_accuracy <- metricas_resamples %>% filter(metrica == "Accuracy")
comparaciones  <- pairwise.wilcox.test(x = metricas_accuracy$valor,
                                       g = metricas_accuracy$modelo,
                                       paired = TRUE,
                                       p.adjust.method = "holm")

# Se almacenan los p_values en forma de dataframe
comparaciones <- comparaciones$p.value %>%
  as.data.frame() %>%
  rownames_to_column(var = "modeloA") %>%
  gather(key = "modeloB", value = "p_value", -modeloA) %>%
  na.omit() %>%
  arrange(modeloA)
```

```
comparaciones
```

```
##      modeloA  modeloB      p_value
## 1   boosting    arbol 3.907518e-04
## 2        KNN    arbol 7.946071e-01
## 3        KNN boosting 8.021198e-06
## 4        LDA    arbol 4.018370e-01
## 5        LDA boosting 7.401186e-05
## 6        LDA      KNN 1.000000e+00
## 7   logistic    arbol 1.806585e-01
## 8   logistic boosting 7.793821e-05
## 9   logistic      KNN 1.000000e+00
## 10  logistic      LDA 1.000000e+00
## 11        NB    arbol 2.729060e-01
## 12        NB boosting 1.427666e-04
## 13        NB      KNN 1.000000e+00
## 14        NB      LDA 1.000000e+00
## 15        NB logistic 1.000000e+00
## 16      NNET    arbol 7.946071e-01
## 17      NNET boosting 4.897953e-02
## 18      NNET      KNN 4.897953e-02
## 19      NNET      LDA 4.080234e-03
## 20      NNET logistic 2.449764e-03
## 21      NNET       NB 1.451528e-03
## 22        rf    arbol 1.284270e-04
## 23        rf boosting 1.000000e+00
## 24        rf      KNN 5.885873e-06
## 25        rf      LDA 6.109140e-05
## 26        rf logistic 7.044149e-06
## 27        rf       NB 2.638262e-05
## 28        rf     NNET 3.157707e-02
## 29 SVMradial    arbol 6.109140e-05
## 30 SVMradial boosting 1.000000e+00
## 31 SVMradial      KNN 8.623374e-06
## 32 SVMradial      LDA 2.790278e-05
## 33 SVMradial logistic 1.122541e-05
## 34 SVMradial       NB 1.032439e-05
## 35 SVMradial     NNET 2.449764e-03
## 36 SVMradial       rf 1.000000e+00
```

Acorde a las comparaciones por pares, no existen evidencias suficientes para considerar que la capacidad predictiva de los modelos *Random Forest*, *SVM radial* y *Boosting* es distinta.

Error de test

Aunque está demostrado que los métodos de validación tipo *CV*, *bootstrapping*, *LOOCV*... consiguen estimaciones muy buenas del error que comente un modelo, es conveniente hacer una medición final con nuevas observaciones para asegurar que, durante la optimización, no se haya generado *overfitting*. Esta es la razón por la que, al inicio de un análisis, se separa un conjunto de test que se mantiene aislado de todo el proceso de transformaciones, entrenamiento y optimización.

Tal y como se describió anteriormente, si se desea obtener predicciones para varios modelos, es conveniente emplear la función extractPrediction(). Esta función devuelve un *dataframe* con las predicciones de cada uno de los modelos, tanto para las observaciones de entrenamiento como para las de test. Además, muestra el verdadero valor de cada observación.

```
predicciones <- extractPrediction(
                  models = modelos,
                  testX = datos_test_prep[, -1],
                  testY = datos_test_prep$Survived
                  )
predicciones %>% head()
```

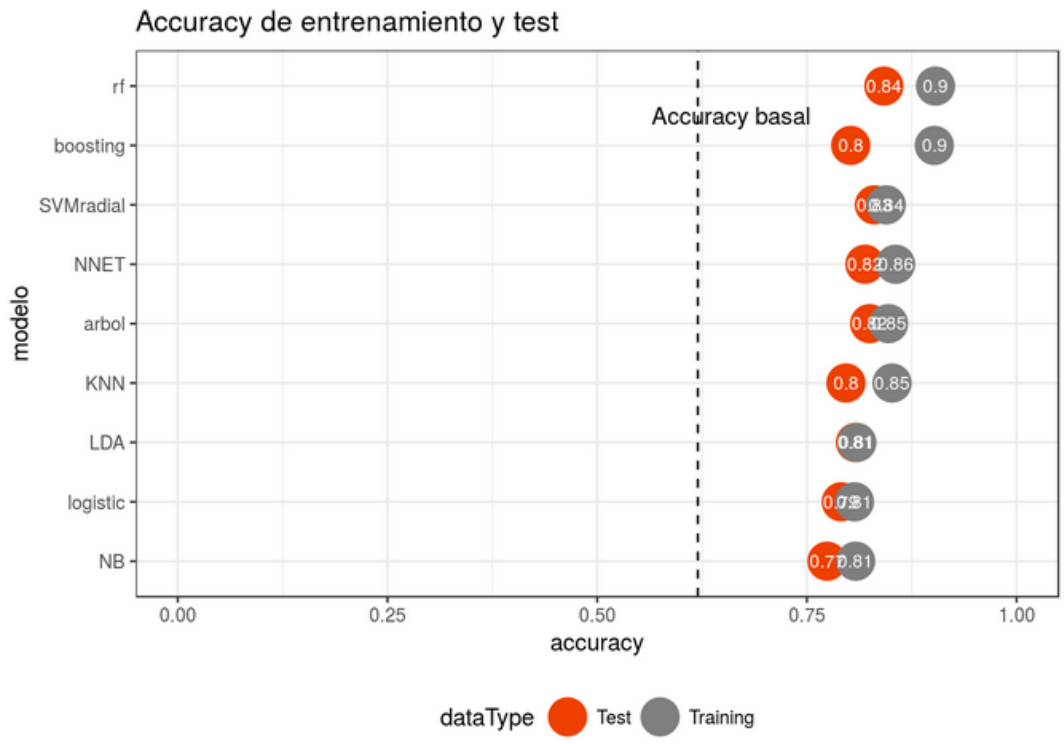
```
##   obs pred model dataType object
## 1  No   No   knn Training    KNN
## 2  Si   Si   knn Training    KNN
## 3  Si   No   knn Training    KNN
## 4  Si   Si   knn Training    KNN
## 5  No   No   knn Training    KNN
## 6  No   No   knn Training    KNN
```

Con toda esta información, es sencillo comparar los resultados de predicción entre modelos y las diferencias entre conjunto de entrenamiento y test.

```
metricas_predicciones <- predicciones %>%
                         mutate(acierto = ifelse(obs == pred, TRUE, FALSE)) %>%
                         group_by(object, dataType) %>%
                         summarise(accuracy = mean(acierto))

metricas_predicciones %>%
  spread(key = dataType, value = accuracy) %>%
  arrange(desc(Test))
```

```
## # A tibble: 9 x 3
## # Groups:   object [9]
##   object     Test Training
##   <fct>     <dbl>    <dbl>
## 1 rf        0.842    0.903
## 2 SVMradial 0.831    0.845
## 3 arbol     0.825    0.847
## 4 NNET      0.819    0.856
## 5 LDA       0.808    0.810
## 6 boosting  0.802    0.902
## 7 KNN       0.797    0.853
## 8 logistic  0.791    0.807
## 9 NB        0.774    0.808
```

```
ggplot(data = metricas_predicciones,
       aes(x = reorder(object, accuracy), y = accuracy,
           color = dataType, label = round(accuracy, 2))) +
  geom_point(size = 8) +
  scale_color_manual(values = c("orangered2", "gray50")) +
  geom_text(color = "white", size = 3) +
  # Accuracy basal
  geom_hline(yintercept = 0.62, linetype = "dashed") +
  annotate(geom = "text", y = 0.66, x = 8.5, label = "Accuracy basal") +
  coord_flip() +
  labs(title = "Accuracy de entrenamiento y test", x = "modelo") +
  theme_bw() +  theme(legend.position = "bottom")
```



Puede verse que, todos los modelos, consiguen más predicciones correctas en el conjunto de entrenamiento que en el de test, de ahí que las métricas obtenidas en el entrenamiento no deban utilizarse para evaluar los modelos, son excesivamente optimistas. El modelo *random forest* consigue el *accuracy* de test más alto.

Conclusión

El modelo basado en *random forest* es el que mejores resultados obtiene (acorde a la métrica *accuracy*) tanto en el conjunto de test como en la validación (*repeated CV*). Los modelos basados *SVM Radial*, árbol simple y *NNET* consiguen valores de test muy similares, sin embargo, acorde a los contrastes de hipótesis con los resultados de la validación, los modelos de árbol simple y *NNET* son inferiores a *random forest*.

```
comparaciones %>% filter((modeloA == "rf") | (modeloA == "SVMradial" &
modeloB == "rf"))
```

```
##     modeloA  modeloB      p_value
## 1        rf    arbol 1.284270e-04
## 2        rf boosting 1.000000e+00
## 3        rf      KNN 5.885873e-06
## 4        rf      LDA 6.109140e-05
## 5        rf logistic 7.044149e-06
## 6        rf       NB 2.638262e-05
## 7        rf     NNET 3.157707e-02
## 8 SVMradial       rf 1.000000e+00
```

Teniendo en cuenta toda esta información, si la prioridad es maximizar la capacidad predictiva del modelo, como primera opción se debería seleccionar el modelo de *random forest* o *SVMradial*. Si por el contrario, la interpretabilidad del modelo es importante, el modelo basado en árboles simples es más adecuado.

Model ensembling

A modo general, el término *model ensembling* hace referencia a la combinación de las predicciones de dos o más modelos distintos, con el objetivo de mejorar las predicciones finales. Esta estrategia se basa en la asunción de que, distintos modelos entrenados independientemente, emplean distintos aspectos de los datos para realizar las predicciones, es decir, cada uno es capaz de identificar parte de la "verdad" pero no toda ella. Combinando la perspectiva de cada uno de ellos, se obtiene una descripción más detallada de la verdadera estructura subyacente en los datos. A modo de analogía, imagínese un grupo de estudiantes que se enfrentan a un examen multidisciplinal. Aunque todos obtengan aproximadamente la misma nota, cada uno de ellos habrá conseguido más puntos con las preguntas que tratan sobre las disciplinas en las que destacan. Si en lugar de hacer el examen de forma individual, lo hacen en grupo, cada uno podrá contribuir en los aspectos que más domina, y el resultado final será probablemente superior a cualquiera de los resultados individuales.

La clave para que el *ensembling* consiga mejorar los resultados es la diversidad de los modelos. Si todos los modelos combinados son similares entre ellos, no podrán compensarse unos a otros. Por esta razón, se tiene que intentar combinar modelos que sean lo mejor posible a nivel individual y lo más diferentes entre ellos.

Las formas más simples de combinar las predicciones de varios modelos son: emplear la media para problemas de regresión y la moda para problemas de clasificación. También es posible ponderar estas agregaciones dando distinto peso a cada modelo, por ejemplo, en proporción al *accuracy* que han obtenido de forma individual.

A continuación, se combinan las predicciones de tres modelos utilizando la moda y se comprueba si se consigue superar al mejor de ellos (*random forest*).

```
moda <- function(x){
  return(names(which.max(table(x))))
}


predicciones_ensemble <- predicciones %>%
  filter(dataType == "Test", object %in% c("rf", "SVMradial", "NNET")) %>%
  select(pred, modelo = object) %>%
  group_by(modelo) %>%
  # Se añade un id único por observación para poder pivotar
  mutate(id = 1:n()) %>%
  ungroup() %>%
  spread(key = modelo, value = pred)
```

```
predicciones_ensemble <- predicciones_ensemble %>%
                         mutate(moda = apply(X = predicciones_ensemble,
                                             MARGIN = 1,
                                             FUN = moda))
predicciones_ensemble %>% head()
```

```
## # A tibble: 6 x 5
##      id NNET  rf    SVMradial moda
##   <int> <fct> <fct> <fct>     <chr>
## 1     1 Si    Si    Si        Si
## 2     2 No    No    No        No
## 3     3 No    No    No        No
## 4     4 No    No    No        No
## 5     5 No    No    No        No
## 6     6 No    No    No        No
```

```
# Accuracy de test del ensemble
mean(datos_test_prep$Survived == predicciones_ensemble$moda)
```

```
## [1] 0.8305085
```

En este caso, el *ensemble* no consigue mejorar al modelo de *ranfom forest*.

Entrenar múltiples modelos simultáneamente

Encontrar el mejor modelo para un problema en particular requiere comparar múltiples modelos, cada uno ajustado con diferentes hiperparámetros. Este proceso implica repetir una cantidad notable de código, lo que no suele ser una estrategia eficiente. Lars Kjeldgaard ha creado un paquete llamado modelgrid que permite definir, modificar y entrenar un conjunto de modelos caret de forma simultánea.

La idea detrás de este paquete es crear un *grid* con múltiples modelos, cada uno con su propio *grid* de hiperparámetros, pero que a la vez comparten una serie de elementos comunes (datos de entrenamiento, estrategia de validación...). A continuación, se muestra un ejemplo en el que se entrenan 3 modelos (*regresión logística*, *RandomForest* y *SVM*) cada uno con su propio proceso de optimización de hiperparámetros.

Definir el grid de modelos

El primer paso es crear un objeto model_grid vacío donde se irán definiendo cada uno de los elementos y acciones.

```
library(modelgrid)

grid_modelos <- model_grid()
grid_modelos
```

```
## $shared_settings
## list()
##
## $models
## list()
##
## $model_fits
## list()
##
## attr(,"class")
## [1] "model_grid"
```

El objeto model_grid contiene 3 elementos, en los que se almacenan 3 tipos de información:

- `shared_settings`: almacena información de configuración que se aplica de la misma forma a todos los modelos que forman el *grid*. Generalmente suele ser el nombre de la

variable respuesta y de los predictores, el tipo de estrategia de validación, transformaciones de preprocesado, etc.

- `models`: almacena el nombre de los modelos que forman el *grid*, así como elementos de configuración individuales. Si un mismo parámetro se encuentra definido tanto en `$shared_settings` como en el modelo individual, este último toma preferencia.

- `model_fits`: almacena los modelos ajustados (uno por cada modelo definido en models) después de entrenar el `model_grid`.

Se definen los shared_settings.

```
grid_modelos <- grid_modelos %>%
                share_settings(
                  y = datos_train_prep$Survived,
                  x = datos_train_prep %>% select(-Survived),
                  metric = "Accuracy",
                  trControl = trainControl(method = "repeatedcv",
                                           number = 10,
                                           repeats = 5,
                                           returnResamp = "final",
                                           verboseIter = FALSE,
                                           allowParallel = TRUE
                                           )
                )
```

Una vez definida la configuración común, se tienen que añadir modelos al *grid* especificando, si es necesario, configuraciones individuales para cada uno. En este caso, para cada modelo se comparan diferentes hiperparámetros, por lo que, dentro de cada uno, se define un tuneGrid distinto. Es conveniente darle un nombre identificativo único a cada modelo.

```
grid_modelos <- grid_modelos %>%
                  add_model(
                    model_name = "Reg_logistica",
                    method     = "glm",
                    family     = binomial(link = "logit")
                  ) %>%
                  add_model(
                    model_name = "RandomForest",
                    method     = "ranger",
                    num.trees  = 500,
                    tuneGrid   = expand.grid(
                                   mtry = c(3, 4, 5, 7),
```

```
                                    min.node.size = c(2, 3, 4, 5, 10, 15, 20, 30),
                                    splitrule = "gini"
                                   )
                  ) %>%
                  add_model(
                    model_name = "SVM",
                    method = "svmRadial",
                    tuneGrid   = expand.grid(
                                    sigma = c(0.001, 0.01, 0.1, 0.5, 1),
                                    C = c(1 , 20, 50, 100, 200, 500, 700)
                                   )
                  )
grid_modelos$models
```

```
## $RandomForest
## $RandomForest$method
## [1] "ranger"
##
## $RandomForest$num.trees
## [1] 500
##
## $RandomForest$tuneGrid
##    mtry min.node.size splitrule
## 1     3             2      gini
## 2     4             2      gini
## 3     5             2      gini
## 4     7             2      gini
## 5     3             3      gini
## 6     4             3      gini
## 7     5             3      gini
## 8     7             3      gini
## 9     3             4      gini
## 10    4             4      gini
## 11    5             4      gini
## 12    7             4      gini
## 13    3             5      gini
## 14    4             5      gini
## 15    5             5      gini
## 16    7             5      gini
## 17    3            10      gini
## 18    4            10      gini
## 19    5            10      gini
## 20    7            10      gini
## 21    3            15      gini
## 22    4            15      gini
## 23    5            15      gini
## 24    7            15      gini
## 25    3            20      gini
## 26    4            20      gini
## 27    5            20      gini
```

```
## 28    7            20      gini
## 29    3            30      gini
## 30    4            30      gini
## 31    5            30      gini
## 32    7            30      gini
##
##
## $Reg_logistica
## $Reg_logistica$method
## [1] "glm"
##
## $Reg_logistica$family
##
## Family: binomial
## Link function: logit
##
##
##
## $SVM
## $SVM$method
## [1] "svmRadial"
##
## $SVM$tuneGrid
##    sigma   C
## 1  0.001   1
## 2  0.010   1
## 3  0.100   1
## 4  0.500   1
## 5  1.000   1
## 6  0.001  20
## 7  0.010  20
## 8  0.100  20
## 9  0.500  20
## 10 1.000  20
## 11 0.001  50
## 12 0.010  50
## 13 0.100  50
## 14 0.500  50
## 15 1.000  50
## 16 0.001 100
## 17 0.010 100
## 18 0.100 100
## 19 0.500 100
## 20 1.000 100
## 21 0.001 200
## 22 0.010 200
## 23 0.100 200
## 24 0.500 200
## 25 1.000 200
## 26 0.001 500
```

```
## 27 0.010 500
## 28 0.100 500
## 29 0.500 500
## 30 1.000 500
## 31 0.001 700
## 32 0.010 700
## 33 0.100 700
## 34 0.500 700
## 35 1.000 700
```

Entrenar el grid de modelos

Con la función `train()` se entrenan uno a uno los modelos del `model_grid` (internamente, se llama a la función `train()` de `caret`) y los resultados se almacenan en `model_fits`. Esta función tiene únicamente 3 argumentos:

- `mg`: el objeto `model_grid` que se va a entrenar.
- `train_all`: por defecto, solo se entrenan aquellos modelos que no han sido ajustados anteriormente, es decir, los que no tienen un objeto correspondiente en `$model_fits`. Esto es muy útil ya que permite incluir nuevos modelos en el *grid* sin necesidad de reentrenar los anteriores. Si se indica `train_all = TRUE`, todos los modelos se ajustan de nuevo.
- `resample_seed`: semilla que permite hacer reproducibles (y comparables) las particiones de validación en todos los modelos.

```
# Se emplean 4 cores en paralelo.
library(doMC)
registerDoMC(cores = 4)

grid_modelos <- train(grid_modelos, train_all = FALSE, resample_seed = 123)
grid_modelos$model_fits
```

```
## $RandomForest
## Random Forest
##
## 714 samples
##  10 predictors
##   2 classes: 'No', 'Si'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 5 times)
```

```
## Summary of sample sizes: 643, 643, 642, 643, 642, 643, ...
## Resampling results across tuning parameters:
##
##   mtry  min.node.size  Accuracy   Kappa
##   3      2             0.8291275  0.6250619
##   3      3             0.8302543  0.6277596
##   3      4             0.8277621  0.6220904
##   3      5             0.8263302  0.6192581
##   3     10             0.8274452  0.6214896
##   3     15             0.8274452  0.6210561
##   3     20             0.8254930  0.6161823
##   3     30             0.8204225  0.6024449
##   4      2             0.8266315  0.6230079
##   4      3             0.8302739  0.6309452
##   4      4             0.8280360  0.6260669
##   4      5             0.8291471  0.6287076
##   4     10             0.8263341  0.6223256
##   4     15             0.8274570  0.6250015
##   4     20             0.8249257  0.6192187
##   4     30             0.8237911  0.6149316
##   5      2             0.8246909  0.6205996
##   5      3             0.8272027  0.6261158
##   5      4             0.8277700  0.6273269
##   5      5             0.8314045  0.6348759
##   5     10             0.8294327  0.6306401
##   5     15             0.8269171  0.6253728
##   5     20             0.8285994  0.6287113
##   5     30             0.8282981  0.6270372
##   7      2             0.8263576  0.6266467
##   7      3             0.8277504  0.6284919
##   7      4             0.8283177  0.6297316
##   7      5             0.8272066  0.6273342
##   7     10             0.8330751  0.6391715
##   7     15             0.8277504  0.6278014
##   7     20             0.8252230  0.6226089
##   7     30             0.8235250  0.6189563
##
## Tuning parameter 'splitrule' was held constant at a value of gini
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were mtry = 7, splitrule = gini
##  and min.node.size = 10.
##
## $Reg_logistica
## Generalized Linear Model
##
## 714 samples
##  10 predictors
##   2 classes: 'No', 'Si'
##
## No pre-processing
```

```
## Resampling: Cross-Validated (10 fold, repeated 5 times)
## Summary of sample sizes: 643, 643, 642, 643, 642, 643, ...
## Resampling results:
##
##   Accuracy   Kappa
##   0.7985446  0.5650226
##
##
## $SVM
## Support Vector Machines with Radial Basis Function Kernel
##
## 714 samples
##  10 predictors
##   2 classes: 'No', 'Si'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 5 times)
## Summary of sample sizes: 643, 643, 642, 643, 642, 643, ...
## Resampling results across tuning parameters:
##
##   sigma  C    Accuracy   Kappa
##   0.001    1  0.7702504  0.4750233
##   0.001   20  0.7904343  0.5457904
##   0.001   50  0.7996674  0.5666844
##   0.001  100  0.8136815  0.5988627
##   0.001  200  0.8201252  0.6113697
##   0.001  500  0.8299335  0.6319814
##   0.001  700  0.8313380  0.6346488
##   0.010    1  0.7974178  0.5614500
##   0.010   20  0.8305203  0.6314211
##   0.010   50  0.8271518  0.6186354
##   0.010  100  0.8299726  0.6224362
##   0.010  200  0.8307981  0.6231936
##   0.010  500  0.8235485  0.6092191
##   0.010  700  0.8187833  0.5998291
##   0.100    1  0.8252034  0.6120609
##   0.100   20  0.8151291  0.5946652
##   0.100   50  0.8092527  0.5828385
##   0.100  100  0.8092527  0.5828531
##   0.100  200  0.8084155  0.5810931
##   0.100  500  0.8064515  0.5779792
##   0.100  700  0.8042175  0.5732928
##   0.500    1  0.8145579  0.5932243
##   0.500   20  0.8036659  0.5738816
##   0.500   50  0.8000078  0.5693739
##   0.500  100  0.8011307  0.5722887
##   0.500  200  0.8033646  0.5772883
##   0.500  500  0.8003052  0.5708128
##   0.500  700  0.7989006  0.5668251
##   1.000    1  0.8070070  0.5811630
```

```
##   1.000   20  0.7969405  0.5647608
##   1.000   50  0.7985994  0.5685710
##   1.000  100  0.7974844  0.5659971
##   1.000  200  0.7910524  0.5507616
##   1.000  500  0.7888224  0.5428908
##   1.000  700  0.7857512  0.5360285
##
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were sigma = 0.001 and C = 700.
```

Extracción de información

El elemento `$model_fits`, creado tras entrenar el *grid*, no es más que una lista de modelos caret, por lo que se pueden aplicar todas las funciones vistas a lo largo del documento.

```
# Extracción de las métricas de validación cruzada.
metricas_cv <- caret::resamples(x = grid_modelos$model_fits)
metricas_cv$values %>% head()
```

```
##      Resample RandomForest~Accuracy RandomForest~Kappa
## 1 Fold01.Rep1             0.7887324          0.5419355
## 2 Fold01.Rep2             0.8591549          0.6830357
## 3 Fold01.Rep3             0.8309859          0.6362084
## 4 Fold01.Rep4             0.8450704          0.6539654
## 5 Fold01.Rep5             0.8450704          0.6591008
## 6 Fold02.Rep1             0.8450704          0.6640860
##   Reg_logistica~Accuracy Reg_logistica~Kappa SVM~Accuracy SVM~Kappa
## 1              0.8591549           0.6877748    0.8450704 0.6539654
## 2              0.8028169           0.5562500    0.8873239 0.7502199
## 3              0.8591549           0.6923744    0.8732394 0.7210825
## 4              0.7746479           0.4850408    0.7746479 0.4928571
## 5              0.8028169           0.5562500    0.8309859 0.6253298
## 6              0.8450704           0.6782035    0.8873239 0.7574722
```

```
# Predicción
predicciones <- extractPrediction(
                  models = grid_modelos$model_fits,
                  testX = datos_test_prep[, -1],
                  testY = datos_test_prep$Survived
                  )
predicciones %>% head()
```

```
##   obs pred  model dataType       object
## 1  No   No ranger Training RandomForest
## 2  Si   Si ranger Training RandomForest
## 3  Si   No ranger Training RandomForest
## 4  Si   Si ranger Training RandomForest
## 5  No   No ranger Training RandomForest
## 6  No   No ranger Training RandomForest
```

Modificar o eliminar modelos

`modelgrid` ofrece una serie de funciones para modificar o eliminar modelos presentes en un `model_grid` sin alterar el resto.

Con `edit_model()` se modifican modelos identificándolos por su nombre. Todo modelo existente modificado es eliminado de `$model_fits` automáticamente, por lo que será entrenado de nuevo al aplicar `train()`.

```
grid_modelos <- grid_modelos %>%
                edit_model(model_name = "RandomForest",
                           num.trees = 200)
grid_modelos$models$RandomForest$num.trees
```

```
## [1] 200
```

Con remove_model se elimina un modelo del *grid*.

```
grid_modelos <- grid_modelos %>%
                remove_model(model_name = "RandomForest")
names(grid_modelos$models)
```

```
## [1] "Reg_logistica" "SVM"
```

Anexos

Anexo1: Otros modelos para rfe()

Además de los modelos predefinidos (*lmFuncs, treebagFuncs, rfFuncs* y *nbFuncs*), la función rfe() acepta cualquiera de los modelos que recoge *caret*. Para utilizar estos últimos, hay que especificar en el control de entrenamiento el argumento functions = caretFuncs y en train() el nombre del modelo. Si el algoritmo seleccionado no incorpora una estrategia propia para cuantificar la importancia de los predictores, como es el caso de las maquinas vector soporte, caret cuantifica el área bajo la curva *ROC* conseguida por cada predictor y la emplea como criterio de importancia. Para más detalles sobre las métricas propias de cada modelo consultar Model Specific Metrics.

Véase como hacer una eliminación recursiva empleando como modelo una máquina de vector soporte lineal.

```
# ELIMINACIÓN RECURSIVA MEDIANTE RANDOM FOREST Y BOOTSTRAPPING
# ==============================================================================

# Se paraleliza el proceso para que sea más rápido. El número de cores debe
# seleccionarse en función del ordenador que se está empleando.
library(doMC)
registerDoMC(cores = 4)

# Tamaño de los conjuntos de predictores analizados
subsets <- c(3:11)

# Número de resamples creados por bootstrapping
repeticiones <- 5

# Hiperparámetros del modelo
hiperparametros <- data.frame(C = 1)

# Se crea una semilla para cada repetición de validación.
set.seed(123)
seeds <- vector(mode = "list", length = repeticiones + 1)
for (i in 1:repeticiones) {
  seeds[[i]] <- sample.int(1000, length(subsets))
}
seeds[[repeticiones + 1]] <- sample.int(1000, 1)
```

```
# Control de entrenamiento
ctrl_rfe <- rfeControl(functions = caretFuncs, method = "boot",
                       number = repeticiones, returnResamp = "all",
                       allowParallel = TRUE, verbose = FALSE,
                       seeds = seeds)

# Se ejecuta la eliminación recursiva de predictores
set.seed(342)
rf_rfe <- rfe(Survived ~ ., data = datos_train_prep,
              sizes = subsets,
              method = "svmLinear",
              tuneGrid = hiperparametros,
              metric = "Accuracy",
              rfeControl = ctrl_rfe)

# Se muestra una tabla resumen con los resultados
rf_rfe
```

```
##
## Recursive feature selection
##
## Outer resampling method: Bootstrapped (5 reps)
##
## Resampling performance over subset size:
##
##  Variables Accuracy  Kappa AccuracySD KappaSD Selected
##          3   0.8098 0.5807    0.02368 0.04579
##          4   0.8098 0.5807    0.02368 0.04579
##          5   0.8098 0.5807    0.02368 0.04579
##          6   0.8098 0.5807    0.02368 0.04579
##          7   0.8090 0.5796    0.01650 0.03231
##          8   0.8217 0.6074    0.02420 0.05028        *
##          9   0.8195 0.6033    0.02462 0.05141
##         10   0.8180 0.5997    0.02696 0.05624
##
## The top 5 variables (out of 8):
##    Sex_male, Fare, Pclass_X3, Parch, Embarked_S
```

```
# El objeto rf_rfe almacena en optVariables las variables del mejor modelo.
rf_rfe$optVariables
```

```
## [1] "Sex_male"       "Fare"           "Pclass_X3"      "Parch"
## [5] "Embarked_S"     "Pclass_X2"      "SibSp"          "Age_grupo_niño"
```

Anexo2: Otros modelos para sbf()

Los mismos pasos mostrados en el anexo 1 para la función rfe() pueden emplearse para modificar el modelo empleado por la función sbf().

Anexo3: Modificación test estadísticos de sbf()

Los test estadísticos que sbf() emplea por defecto para cuantificar la relación entre los predictores y la variable respuesta son:

- ANOVA: cuando la variable respuesta es cualitativa (modelo de clasificación).
- GAMS: cuando la variable respuesta es continua (modelo de regresión).

y el *p-value* empleado como límite para la selección es 0.05.

Todas las subfunciones que emplea sbf() para la selección de los predictores (filtrado, test estadístico, score, ranking…) están almacenadas dentro de los modelos predefinidos (*caretSBF, lmSBF, rfSBF, treebagSBF, ldaSBF, nbSBF*) y pueden ser sobrescritas para emplear cualquier otra opción que el usuario considere. Por ejemplo, para modificar el límite de *p-value* que se emplea para seleccionar los predictores con *rfSBF*, se sobrescribe la función rfSBF$filter.

```
# Por defecto, el límite de filtrado es 0.05
rfSBF$filter

## function (score, x, y)
## score <= 0.05
## <environment: namespace:caret>

# Se crea una nueva función filter en la que el límite es 0.01
custom_filter <- function(score, x, y){
  score <= 0.01
}

# Se sobreescribe la función existente
rfSBF$filter <- custom_filter
rfSBF$filter
```

```
## function(score, x, y){
##   score <= 0.01
## }
```

```
# El resto de los pasos son exactamente igual

# Se paraleliza para que sea más rápido
library(doMC)
registerDoMC(cores = 4)

# Se crea una semilla para cada partición y cada repetición: el vector debe
# tener B+1 semillas donde B es particiones * repeticiones.
particiones = 10
repeticiones = 5
set.seed(123)
seeds <- sample.int(1000, particiones * repeticiones + 1)

ctrl_filtrado <- sbfControl(functions = rfSBF, method = "repeatedcv",
                            number = particiones, repeats = repeticiones,
                            seeds = seeds, verbose = FALSE,
                            saveDetails = TRUE, allowParallel = TRUE)
set.seed(234)
rf_sbf <- sbf(Survived ~ ., data = datos_train_prep,
              sbfControl = ctrl_filtrado,
              # argumentos para el modelo de evaluación
              ntree = 500)
rf_sbf
```

```
##
## Selection By Filter
##
## Outer resampling method: Cross-Validated (10 fold, repeated 5 times)
##
## Resampling performance:
##
##  Accuracy  Kappa AccuracySD KappaSD
##    0.8149 0.5873    0.04212 0.09737
##
## Using the training set, 6 variables were selected:
##    Parch, Fare, Pclass_X3, Sex_male, Embarked_S...
##
## During resampling, the top 5 selected variables (out of a possible 7):
##    Embarked_S (100%), Fare (100%), Pclass_X3 (100%), Sex_male (100%),
Age_grupo_niño (68%)
##
## On average, 5.5 variables were selected (min = 4, max = 7)
```

```
rf_sbf$optVariables
```

```
## [1] "Parch"          "Fare"           "Pclass_X3"      "Sex_male"
## [5] "Embarked_S"     "Age_grupo_niño"
```

También se puede modificar el test estadístico empleado para la calcular los *p-values*. Para ello se tiene que sobrescribir la función $score.

```
# Por defecto, si la variable respuesta es factor, se llama a la función
anovaScores
rfSBF$score
```

```
## function (x, y)
## {
##     if (is.factor(y))
##         anovaScores(x, y)
##     else gamScores(x, y)
## }
## <environment: namespace:caret>
```

```
anovaScores
```

```
## function (x, y)
## {
##     if (is.factor(x))
##         stop("The predictors should be numeric")
##     pv <- try(anova(lm(x ~ y), test = "F")[1, "Pr(>F)"], silent = TRUE)
##     if (any(class(pv) == "try-error") || is.na(pv) || is.nan(pv))
##         pv <- 1
##     pv
## }
## <environment: namespace:caret>
```

```
# Se crea una nueva función que emplee el test de Kruskal-Wallis (alternativa no
# paramétrica al test anova)
kruskalScores <- function(x,y){
  if (is.factor(x)){
        stop("The predictors should be numeric")
  }
  pv <- try(kruskal.test(x = x, g = y)[["p.value"]], silent = TRUE)
  if (any(class(pv) == "try-error") || is.na(pv) || is.nan(pv)){
    pv <- 1
  }
  pv
}

custom_score <- function(x, y){
    if (is.factor(y))
        kruskalScores(x, y)
    else gamScores(x, y)
}
```

```
# Se sobreescribe la función score existente
rfSBF$score <- custom_score

# El resto de los pasos son exactamente igual

# Se paraleliza para que sea más rápido
library(doMC)
registerDoMC(cores = 4)

# Se crea una semilla para cada partición y cada repetición: el vector debe
# tener B+1 semillas donde B es particiones * repeticiones.
particiones = 10
repeticiones = 5
set.seed(123)
seeds <- sample.int(1000, particiones * repeticiones + 1)

ctrl_filtrado <- sbfControl(functions = rfSBF, method = "repeatedcv",
                            number = particiones, repeats = repeticiones,
                            seeds = seeds, verbose = FALSE,
                            saveDetails = TRUE, allowParallel = TRUE)
set.seed(234)
rf_sbf <- sbf(Survived ~ ., data = datos_train_prep,
              sbfControl = ctrl_filtrado,
              # argumentos para el modelo de evaluación
              ntree = 500)
rf_sbf
```

```
##
## Selection By Filter
##
## Outer resampling method: Cross-Validated (10 fold, repeated 5 times)
##
## Resampling performance:
##
##  Accuracy  Kappa AccuracySD KappaSD
##    0.8124 0.5851    0.04407 0.09945
##
## Using the training set, 7 variables were selected:
##    SibSp, Parch, Fare, Pclass_X3, Sex_male...
##
## During resampling, the top 5 selected variables (out of a possible 8):
##    Embarked_S (100%), Fare (100%), Parch (100%), Pclass_X3 (100%), Sex_male
(100%)
##
## On average, 6.7 variables were selected (min = 5, max = 8)
```

```
rf_sbf$optVariables
```

```
## [1] "SibSp"          "Parch"          "Fare"           "Pclass_X3"
## [5] "Sex_male"       "Embarked_S"     "Age_grupo_niño"
```

Anexo4: Métodos de validación

Los métodos de validación, también conocidos como *resampling*, son estrategias que permiten estimar la capacidad predictiva de los modelos cuando se aplican a nuevas observaciones, haciendo uso únicamente de los datos de entrenamiento. La idea en la que se basan todos ellos es la siguiente: el modelo se ajusta empleando un subconjunto de observaciones del conjunto de entrenamiento y se evalúa (calculando una métrica que mida como de bueno es el modelo, por ejemplo, *accuracy*) con las observaciones restantes. Este proceso se repite múltiples veces y los resultados se agregan y promedian. Gracias a las repeticiones, se compensan las posibles desviaciones que puedan surgir por el reparto aleatorio de las observaciones. La diferencia entre métodos suele ser la forma en la que se generan los subconjuntos de entrenamiento/validación.

k-Fold-Cross-Validation (CV)

Las observaciones de entrenamiento se reparten en *k folds* (conjuntos) del mismo tamaño. El modelo se ajusta con todas las observaciones excepto las del primer *fold* y se evalúa prediciendo las observaciones del *fold* que ha quedado excluido, obteniendo así la primera métrica. El proceso se repite *k* veces, excluyendo un *fold* distinto en cada iteración. Al final, se generan *k* valores de la métrica, que se agregan (normalmente con la media y la desviación típica) generando la estimación final de validación.

Leave-One-Out Cross-Validation (LOOCV)

LOOCV es un caso especial de *k-Fold-Cross-Validation* en el que el número *k* de *folds* es igual al número de observaciones disponibles en el conjunto de entrenamiento. El modelo se ajusta cada vez con todas las observaciones excepto una, que se emplea para evaluar el modelo. Este método supone un coste computacional muy elevado, el modelo se ajusta tantas veces como observaciones de entrenamiento, por lo que, en la práctica, no suele compensar emplearlo.

Repeated k-Fold-Cross-Validation (repeated CV)

Es exactamente igual al método *k-Fold-Cross-Validation* pero repitiendo el proceso completo *n* veces. Por ejemplo, *10-Fold-Cross-Validation* con 5 repeticiones implica a un total de 50 iteraciones ajuste-validación, pero no equivale a un *50-Fold-Cross-Validation*.

Leave-Group-Out Cross-Validation (LGOCV)

LGOCV, también conocido como *repeated train/test splits* o *Monte Carlo Cross-Validation*, consiste simplemente en generar múltiples divisiones aleatorias entrenamiento-test (solo dos conjuntos por repetición). La proporción de observaciones que va a cada conjunto se determina de antemano, 80%-20% suele dar buenos resultados. Este método, aunque más simple de implementar que *CV*, requiere de muchas repeticiones (>50) para generar estimaciones estables.

Bootstrapping

Una muestra *bootstrap* es una muestra obtenida a partir de la muestra original por muestreo aleatorio con reposición, y del mismo tamaño que la muestra original. Muestreo aleatorio con reposición (*resampling with replacement*) significa que, después de que una observación sea extraída, se vuelve a poner a disposición para las siguientes extracciones. Como resultado de este tipo de muestreo, algunas observaciones aparecerán múltiples veces en la muestra *bootstrap* y otras ninguna. Las observaciones no seleccionadas reciben el nombre de *out-of-bag (OOB)*. Por cada iteración de *bootstrapping* se genera una nueva muestra *bootstrap*, se ajusta el modelo con ella y se evalúa con las observaciones *out-of-bag*.

---

1. Obtener una nueva muestra del mismo tamaño que la muestra original mediante muestro aleatorio con reposición.

2. Ajustar el modelo empleando la nueva muestra generada en el paso 1.

3. Calcular el error del modelo empleando aquellas observaciones de la muestra original que no se han incluido en la nueva muestra. A este error se le conoce como error de validación.

4. Repetir el proceso *n* veces y calcular la media de los *n* errores de validación.

5. Finalmente, y tras las *n* repeticiones, se ajusta el modelo final empleando todas las observaciones de entrenamiento originales.

La naturaleza del proceso de *bootstrapping* genera cierto bias en las estimaciones que puede ser problemático cuando el conjunto de entrenamiento es pequeño. Existen ciertas

modificaciones del algoritmo original para corregir este problema, algunos de ellos son: *632 method* y *632+ method*.

No existe un método de validación que supere al resto en todos los escenarios, la elección debe basarse en varios factores.

- Si el tamaño de la muestra es pequeño, se recomienda emplear *repeated k-Fold-Cross-Validation*, ya que consigue un buen equilibrio bias-varianza y, dado que no son muchas observaciones, el coste computacional no es excesivo.
- Si el objetivo principal es comparar modelos más que obtener una estimación precisa de las métricas, se recomienda *bootstrapping* ya que tiene menos varianza.
- Si el tamaño muestral es muy grande, la diferencia entre métodos se reduce y toma más importancia la eficiencia computacional. En estos casos, *10-Fold-Cross-Validation* simple es suficiente.

Puede encontrarse un estudio comparativo de los diferentes métodos en Comparing Different Species of Cross-Validation.

Anexo5: Métricas

Existe una gran variedad de métricas que permiten evaluar como de bueno es un algoritmo realizando predicciones. La idoneidad de cada una depende completamente del problema en cuestión, y su correcta elección dependerá de cómo de bien entienda el analista el problema al que se enfrenta. A continuación, se describen algunas de las más utilizadas.

Accuracy y Kappa

Estas dos métricas son las más empleadas en problemas de clasificación binaria y multiclase. *Accuracy* es el porcentaje de observaciones correctamente clasificadas respecto al total de predicciones. *Kappa* o *Cohen's Kappa* es el valor de *accuracy* normalizado respecto del porcentaje de acierto esperado por azar. A diferencia de *accuracy,* cuyo rango de valores puede ser [0, 1], el de *kappa* es [-1, 1]. En problemas con clases desbalanceadas, donde el grupo mayoritario supera por mucho a los otros, *Kappa* es más útil porque evita caer en la ilusión de creer que un modelo es bueno cuando realmente solo supera por poco lo esperado por azar.

MSE, RMSE, MAE

Estas son las métricas más empleadas en problemas de regresión.

MSE (Mean Squared Error) es la media de los errores elevados al cuadrado. Suele ser muy buen indicativo de cómo funciona el modelo en general, pero tiene la desventaja de estar en unidades cuadradas. Para mejorar la interpretación, suele emplearse *RMSE (Root Mean Squared Error),* que es la raíz cuadrada del *MSE* y por lo tanto sus unidades son las mismas que la variable respuesta.

MAE (Mean Absolute Error) es la media de los errores en valor absoluto. La diferencia respecto a *MSE* es que, este último, eleva al cuadrado los errores, lo significa que penaliza mucho más las desviaciones grandes. A modo general, *MSE* favorece modelos que se comportan aproximadamente igual de bien en todas las observaciones, mientras que *MAE* favorece modelos que predicen muy bien la gran mayoría de observaciones aunque en unas pocas se equivoque por mucho.

Por defecto, caret emplea *RMSE*, R^2 y *MAE* en problemas de regresión, y *accuracy* y *Kappa* para problemas de clasificación. Durante la optimización de hiperparámetros, la selección de mejor modelo se hace, en base a *RMSE* o *accuracy*, aunque puede ser modificada mediante el argumento metric de la función train(). Además de estas métricas, caret permite emplear otras métricas como el *AUC* de la curva *ROC* *Anexo 6*.

Anexo6: Curva ROC

Tal como se explicó anteriormente, algunos modelos de clasificación, además de predecir la clase a la que pertenece una observación, calculan la probabilidad. Conocer las probabilidades aporta mayor información, puesto que a partir de ellas no solo se conoce la clase a la que pertenece cada observación (la que mayor probabilidad tiene), sino la confianza con la que se hace dicha asignación. Además, controlando el *cutoff* de probabilidad empleado para asignar las observaciones a cada clase, se puede controlar el riesgo de falsos positivos y falsos negativos.

Las curvas *ROC* (*Receiver Operating Characteritic curve*) permiten evaluar, en problemas de clasificación binaria, cómo varia la proporción de verdaderos positivos (sensibilidad o *recall*) y la de falsos positivos (1-especificidad) dependiendo del *cutoff* de probabilidad empleado en las asignaciones. El gráfico resultante es muy útil para identificar el *cutoff* que consigue un mejor equilibrio sensibilidad-especificidad. Además de esto, la curva ROC, en concreto el área bajo la curva (AUC), puede emplearse como métrica para evaluar modelos. Un modelo que clasifica perfectamente las dos clases tiene un 100% de sensibilidad y especificidad, por lo que el área

bajo la curva es de 1. Un modelo que predice por debajo de lo esperado por azar, tiene un AUC menor de 0.5. Una condición necesaria para crear una curva ROC es disponer de la probabilidad de clases en las predicciones.

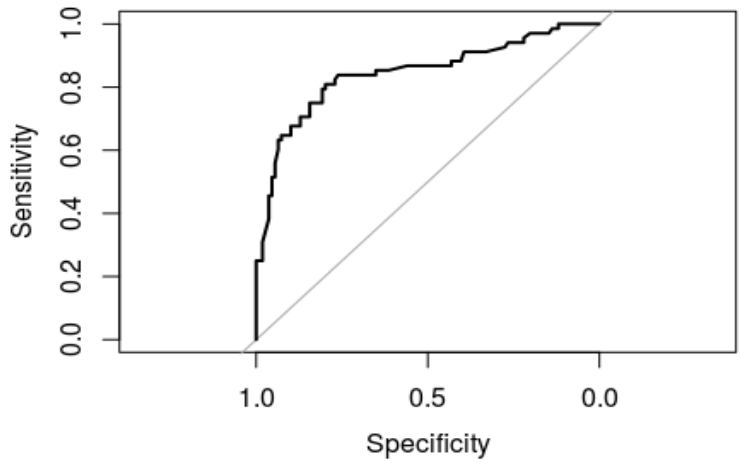
En caret, se puede sustituir la métrica *accuracy* empleada por defecto en problemas de clasificación y calcular en su lugar el AUC. Para ello, se tienen que indicar los argumentos summaryFunction = twoClassSummary y classProbs = TRUE en el control de entrenamiento. El segundo argumento es necesario porque el cálculo de la curva ROC requiere las probabilidades predichas para cada clase. Además del área bajo la curva, se calcula la sensibilidad y la especificidad para un *cutoff* de 0.5.

```
# PARALELIZACIÓN DE PROCESO
#==============================================================================
library(doMC)
registerDoMC(cores = 4)

# HIPERPARÁMETROS, NÚMERO DE REPETICIONES Y SEMILLAS PARA CADA REPETICIÓN
#==============================================================================
particiones  <- 10
repeticiones <- 5

# Hiperparámetros
hiperparametros <- data.frame(parameter = "none")

set.seed(123)
seeds <- vector(mode = "list", length = (particiones * repeticiones) + 1)
for (i in 1:(particiones * repeticiones)) {
  seeds[[i]] <- sample.int(1000, nrow(hiperparametros))
}
seeds[[(particiones * repeticiones) + 1]] <- sample.int(1000, 1)

# DEFINICIÓN DEL ENTRENAMIENTO
#==============================================================================
control_train <- trainControl(method = "repeatedcv", number = particiones,
                              repeats = repeticiones, seeds = seeds,
                              returnResamp = "final", verboseIter = FALSE,
                              summaryFunction = twoClassSummary,
                              classProbs = TRUE,
                              allowParallel = TRUE)

# AJUSTE DEL MODELO
# =============================================================================
set.seed(342)
modelo_logistic <- train(Survived ~ ., data = datos_train_prep,
                         method = "glm", tuneGrid = hiperparametros,
                         trControl = control_train, family = "binomial")
modelo_logistic
```

```
## Generalized Linear Model
##
## 714 samples
##  10 predictors
##   2 classes: 'No', 'Si'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 5 times)
## Summary of sample sizes: 642, 643, 642, 643, 642, 643, ...
## Resampling results:
##
##   ROC        Sens       Spec
##   0.8541901  0.8695455  0.6871958
```

El paquete pROC contiene múltiples funciones para crear, representar y obtener métricas a partir de curvas ROC. Como argumentos se necesitan únicamente las probabilidades predichas para cada clase y la clase verdadera a la que pertenece cada observación.

```
library(pROC)
# Se obtienen las probabilidades predichas para cada clase
predicciones <- predict(object = modelo_logistic,
                        newdata = datos_test_prep,
                        type = "prob")
# Cálculo de la curva
curva_roc <- roc(response = datos_test_prep$Survived,
                 predictor = predicciones$Si)

# Gráfico de la curva
plot(curva_roc)
```



```
# Área bajo la curva AUC
auc(curva_roc)
```

```
## Area under the curve: 0.8447
```

```
# Intervalo de confianza de la curva
ci.auc(curva_roc, conf.level = 0.95)
```

```
## 95% CI: 0.7806-0.9088 (DeLong)
```

Anexo7: Imputación

A la hora de imputar una variable, es muy importante analizar qué otras variables se van a emplear para predecir los valores ausentes. Si se emplean variables que no guardan ninguna relación con la variable de interés, probablemente se imputen valores que no se corresponden con la realidad y que, por lo tanto, solo hacen que añadir ruido al modelo. En el set de datos titanic, dos variables necesitan ser imputadas: *Age_grupo* y *Embarked*.

Imputación de Embarked

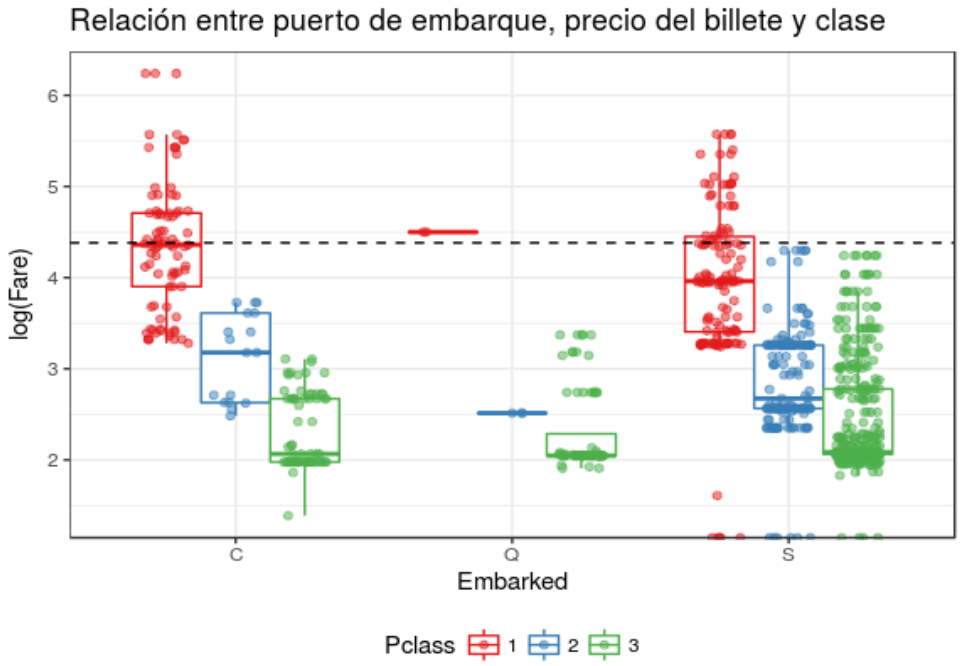
La variable *Embarked* está ausente en dos observaciones:

```
datos %>% filter(is.na(Embarked)) %>% select(-PassengerId, -Name, -Ticket, -
Survived )
```

```
##   Pclass    Sex Age SibSp Parch Fare Cabin Embarked Age_grupo
## 1      1 female  38     0     0   80   B28     <NA>    adulto
## 2      1 female  62     0     0   80   B28     <NA>   anciano
```

La información de las otras variables indica que ambas personas, una adulta y otra anciana, compartían cabina de primera clase y pagaron exactamente la misma cantidad, lo que maximiza la probabilidad de que existiese algún vínculo entre ellas y que, por lo tanto, embarcasen juntas. Véase como se relacionan las variables *Pclass* y *Fare* con el puerto de embarque.

```
datos %>% filter(!is.na(Embarked)) %>%
ggplot(aes(x = Embarked, y = log(Fare), color = Pclass)) +
  geom_boxplot(outlier.shape = NA) +
  geom_point(position = position_jitterdodge(), alpha = 0.5) +
  # Línea horizontal en el precio que pagaron por el billete las dos pasajeras
  geom_hline(yintercept = log(80), linetype = "dashed") +
  scale_colour_brewer(palette = "Set1") +
  theme_bw() +
  labs(title = "Relación entre puerto de embarque, precio del billete y clase") +
  theme(legend.position = "bottom")
```



Con solo un gráfico se puede ver que, la mayoría de pasajeros de clase 1, embarcaron en los puertos *C* y *S*. La probabilidad de que lo hicieran en el puerto *Q* es mínima, por lo que queda descartado. El precio pagado por los billetes (*Fare = 80*) es muy próximo a la mediana de los precios pagados por billetes de primera clase en el puerto *C*, mientras que está casi por encima del 75% (tercer cuartil) de los precios pagados en el puerto *S*. Con toda esta información, es razonable concluir que las dos pasajeras embarcaron en el puerto *C*.

Imputación de Age_grupo

La variable edad no parece estar relacionada con una única variable, por lo que, en este caso, se emplean todos los otros predictores para la imputación.

Bibliografía

Applied Predictive Modeling By Max Kuhn and Kjell Johnson

http://topepo.github.io/caret/index.html

An Introduction to Statistical Learning by James, Gareth et al.

Top 10 algorithms in data mining by Xindong Wu, Vipin Kumar, J. Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J. McLachlan, Angus Ng, Bing Liu, Philip S. Yu, Zhi-Hua Zhou, Michael Steinbach, David J. Hand, Dan Steinberg