

Práctica Machine Learning

Fernández Hernández, Alberto

4/28/2021

```
[IIII]
 )"""( 
 /      \
 /      \
 |`-...-'|
 |aspirin|
 |`-...-'|
 (\)`-.__.(I) _(/)
 (I)   (/)(I)(\)
 (I)
```

Nota: a lo largo de la práctica, tanto en el tuneo de hiperparámetros como en la comparación de modelos se ha llevado a cabo mediante **validación cruzada repetida**, con la misma semilla (1234) y nº de grupos (5).

1. Introducción y descripción de los datos

El objetivo del presente proyecto consiste en **elaborar un modelo de clasificación binaria que permita predecir si un paciente presentará o será más propenso a padecer una complicación hospitalaria tras una intervención quirúrgica**¹. Originalmente, el fichero (extraído de la plataforma Kaggle) contiene tres variables objetivo, dos continuas:

1. *ccsComplicationRate*: incidencia general de complicaciones hospitalarias por cada tipo de intervención quirúrgica.
2. *complication_rsi*: índice de estratificación de riesgo en complicaciones hospitalarias.

Y una binaria:

3. *complication*: si el paciente ha sufrido una complicación (1) o no (0).

Por tanto, de cara a la práctica tendremos únicamente en cuenta, como variable objetivo, la columna *complication*, descartando las dos variables continuas anteriores.

En relación con las posibles variables *input*, nos encontramos con las siguientes:

CONTINUAS

1. *bmi*: índice de masa corporal.
2. *Age*: edad del paciente.
3. *baseline_charlson*: índice de comorbilidad de Charlson, el cual predice la mortalidad a diez años de un paciente que puede tener una variedad de condiciones comórbidas (como una enfermedad cardíaca, SIDA o cáncer).
4. *ahrq_ccs*: tipo de procedimiento/intervención quirúrgica, etiquetado por la Agencia estadounidense para la Investigación Sanitaria². Dicha variable contiene un total de 22 valores únicos, por lo que se ha decidido mantener la variable como numérica.
5. *ccsMort30Rate*: incidencia general de mortalidad a los 30 días por cada intervención (dato por el código de la columna *ahrq_ccs*).
6. *hour*: hora a la que se realizó la intervención.
7. *mortality_rsi*: índice de estratificación de riesgo en la mortalidad a los 30 días.

CATEGÓRICAS

8. *asa_status*: estado físico del paciente establecido por la Sociedad Americana de Anestesiología³. Contiene tres categorías:
 - 0: **estado I-II** (paciente sano / paciente con enfermedad sistémica leve).
 - 1: **estado III** (paciente con enfermedad sistémica grave).
 - 2: **estado IV-VI** (paciente con enfermedad muy grave / no espera sobrevivir sin la operación / muerte cerebral).
9. *baseline_cancer*: ¿El paciente padece algún cáncer? (1 = Sí; 0 = No)
10. *baseline_cvd*: ¿El paciente sufre alguna enfermedad cardio o cerebrovascular? (1 = Sí; 0 = No)
11. *baseline_dementia*: ¿El paciente sufre algún trastorno por demencia? (1 = Sí; 0 = No)
12. *baseline_diabetes*: ¿El paciente sufre diabetes? (1 = Sí; 0 = No)
13. *baseline_digestive*: ¿El paciente sufre alguna enfermedad gastro-intestinal? (1 = Sí; 0 = No)

¹<https://www.kaggle.com/omnamahshivai/surgical-dataset-binary-classification>

²[https://www.hcup-us.ahrq.gov/toolssoftware/ccs10/CCSCategoryNames\(FullLabels\).pdf](https://www.hcup-us.ahrq.gov/toolssoftware/ccs10/CCSCategoryNames(FullLabels).pdf)

³<https://www.asahq.org/standards-and-guidelines/asa-physical-status-classification-system>

14. *baseline_osteoart*: ¿El paciente padece osteoarthritis⁴? (1 = Sí; 0 = No)
15. *baseline_psych*: ¿El paciente padece algún desorden psiquiátrico? (1 = Sí; 0 = No)
16. *baseline_pulmonar*: ¿El paciente sufre alguna enfermedad pulmonar? (1 = Sí; 0 = No)
17. *dow* o *day of week*: día de la semana en el que se realizó la intervención (0 = Lunes; 1 = Martes; 2 = Miércoles; 3 = Jueves; 4 = Viernes).
18. *month*: mes en el que se realizó la intervención.
19. *moonphase*: fase lunar que tuvo lugar durante la intervención quirúrgica (0 = Luna nueva; 1 = Cuarto creciente; 2 = Luna llena; 3 = Cuarto menguante).
20. *mort30*: ¿El paciente presenta algún riesgo de fallecer a los 30 días? (1 = Sí; 0 = No)
21. *gender*: Sexo del paciente (0 = Hombre; 1 = Mujer)
22. *race*: raza del paciente (0 = Caucásico; 1 = Afroameriano; 2 = Otro)

2. Librerías empleadas

A continuación, se expone un listado de las librerías empleadas en el desarrollo del proyecto:

1. *caret*: tuneo de hipérparámetros de los diferentes algoritmos de clasificación.
2. *data.table*: estructura de datos, similar al *data.frame*, aunque mucho más eficiente en memoria.
3. *ggplot2*: librería gráfica.
4. *scorecard*: cálculo del valor de información (IV), así como el peso de la evidencia (WOE).
5. *dummies*: transformación de variables categóricas a *dummies*.
6. *forcats*: tratamiento de variables categóricas.
7. *inspectdf*: librería para inspeccionar las características principales de un *dataset*, incluyendo variables categóricas, valores *missing* o distribución de las variables continuas.
8. *dplyr*: manipulación de datos.
9. *psych*: información general de data.frames y/o data.tables (media, asimetría, desviación típica, entre otros).
10. *doParallel* y *parallel*: parallelización de funciones.
11. *readxl*: lectura de ficheros *Excel* (.xlsx).
12. *purrr*: herramientas de programación funcional.
13. *visualpred*: visualización de predicciones por diferentes algoritmos de clasificación.
14. *h2o*: auto Machine Learning (*autoML*).
15. **Librerías y funciones proporcionadas por el profesor.**

3. Depuración de los datos

Inicialmente, comenzamos con la lectura del fichero:

⁴<https://dicciomed.usal.es/palabra/osteoartritis>

```

# Lectura del fichero
surgical_dataset <- fread("./data/Surgical-deepnet.csv", data.table = FALSE)

# Eliminamos las dos variables objetivo continuas
surgical_dataset$ccsComplicationRate <- NULL; surgical_dataset$complication_rsi <- NULL

dim(surgical_dataset) # Filas x columnas

## [1] 14635    23

```

Nos encontramos con 14.635 observaciones, junto con las 23 variables descritas anteriormente. Si echamos un vistazo a la variable objetivo, podemos observar el desbalanceo entre ambas categorías:

```

table(surgical_dataset$complication)

##
##      0      1
## 10945 3690

```

3.1 Codificación a *factor*

Tras la lectura del fichero, **codificamos como *factor*** tanto la variable objetivo como el resto de variables categóricas:

```

# Codificamos como factor la variable objetivo...
surgical_dataset$complication <- as.factor(surgical_dataset$complication)
# ...Así como el resto de variables categoricas mencionadas anteriormente
cat_columns <- c("gender", "race", "asa_status", "baseline_cancer", "baseline_cvd",
                 "baseline_dementia", "baseline_diabetes", "baseline_digestive",
                 "baseline_osteoart", "baseline_psych", "baseline_pulmonary",
                 "dow", "month", "moonphase", "mort30")
surgical_dataset[,cat_columns] <- lapply(surgical_dataset[, cat_columns], factor)

```

A continuación, almacenamos los nombres de cada variable en un vector por separado, **en función de si es continua o categórica**:

```

# Separamos las variables en numericas, categoricas y target
# [-16] => Salvo la variable objetivo
cat_columns <- names(Filter(is.factor, surgical_dataset))[-16]
num_columns <- names(Filter(is.numeric, surgical_dataset))
target       <- "complication"

```

3.2 Valores NA

Como se puede comprobar a continuación, el *dataset* **no contiene valores *missing*** en ninguna de las variables:

```

sum(is.na(surgical_dataset))

## [1] 0

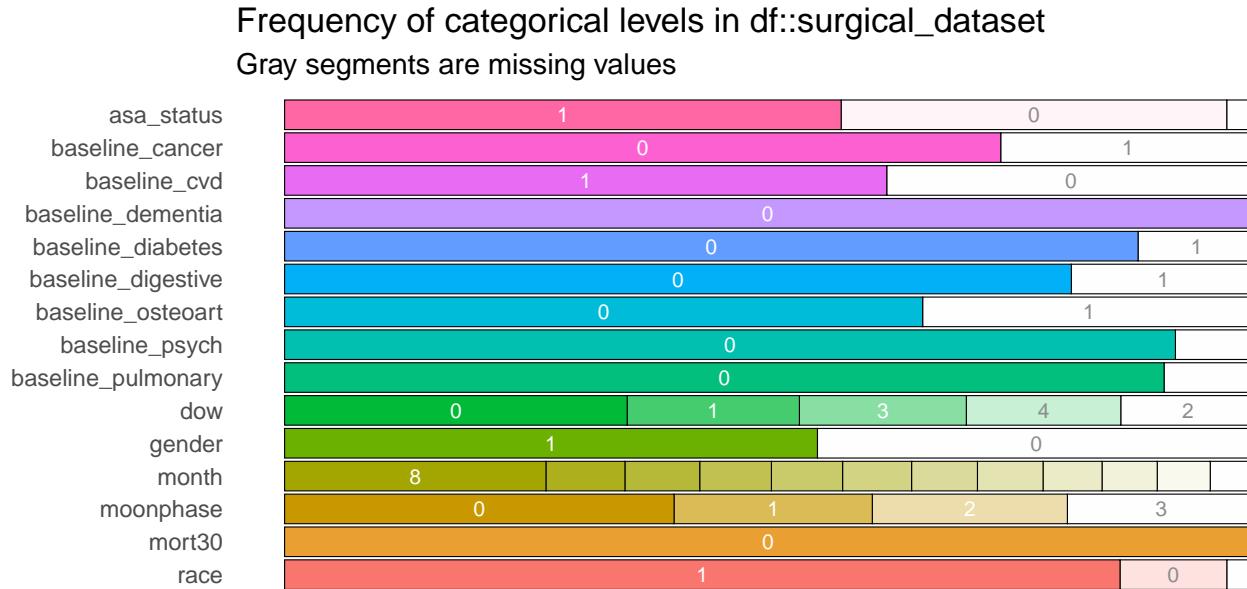
```

3.3 Variables categóricas

Tras almacenar los nombres de cada variable, mediante la librería *inspectdf* se realizó un primer análisis exploratorio de datos automático con el que **analizar el dataset en primera instancia**. Dado que el contenido del informe es muy extenso, se incluirá en la memoria el contenido esencial (el informe completo se incluye, desglosado, en los anexos *00_EDA_report.pdf* y *WOEBIN_factor_variables.pdf*).

Sobre dicho informe, comenzamos remarcando la frecuencia de aparición de los niveles de cada variable categórica:

```
x <- inspectdf::inspect_cat(surgical_dataset[, cat_columns], include_int = TRUE)
show_plot(x)
```



A simple vista, prácticamente todas las categorías presentan una alta frecuencia de aparición, **salvo por baseline_dementia y mort30**, donde el número de observaciones a 1 es de 71 y 58, respectivamente.

```
surgical_dataset[, c("baseline_dementia", "mort30")] %>% map(table)
```

```
## $baseline_dementia      $mort30
##                                0      1
##      0      1                  0      1
## 14564    71                14577  58
```

Es decir, se tratan de variables con pocas observaciones con valor 1. De hecho, si analizamos el valor de información haciendo uso del paquete *scorecard*, la cual nos permite estudiar el “poder predictivo de una variable”, observamos que el valor de información es cero, **dada la poca representatividad de los valores a 1**, de forma que el paquete *scorecard* acaba uniendo ambas categorías, lo que se traduce en un escaso poder de predicción:

Por otro lado, si analizamos la proporción de aparición de la variable objetivo sobre cada categoría:

```
-- baseline_dementia
surgical_dataset %>%
  count(baseline_dementia, complication) %>%
  group_by(complication)

## # A tibble: 4 x 3
## # Groups:   complication [2]
##   baseline_dementia complication     n
##   <fct>              <fct>     <int>
## 1 0                  0          10913
## 2 0                  1          3651
## 3 1                  0            32
## 4 1                  1            39
```

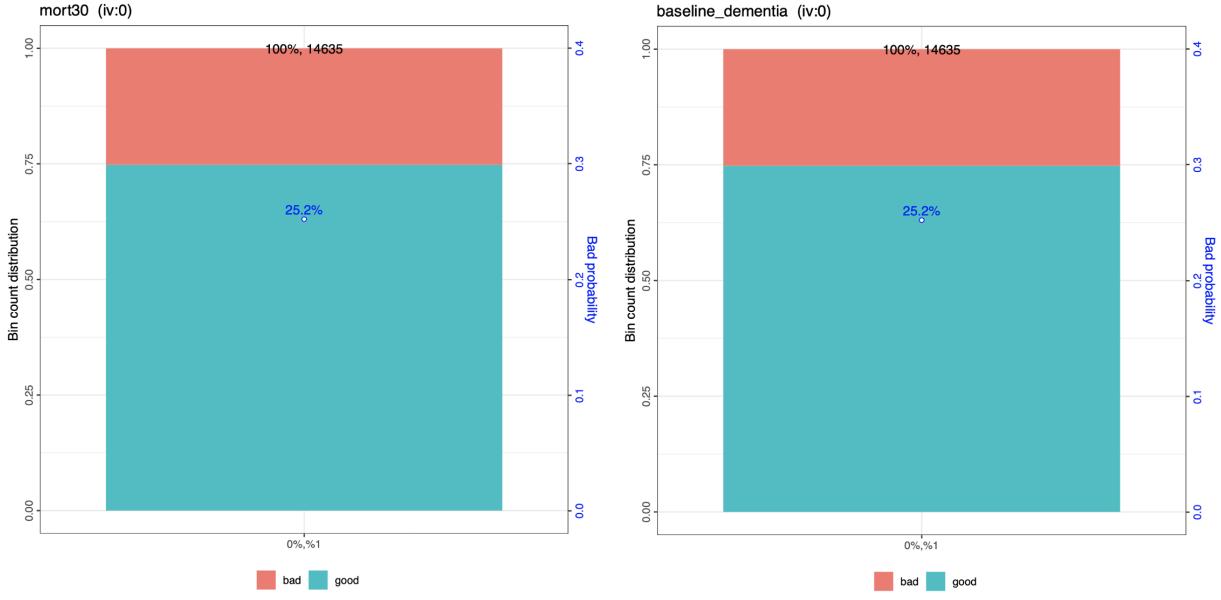


Figure 1: Mort 30 y Baseline dementia (IV)

```
#-- mort30
surgical_dataset %>%
  count(mort30, complication) %>%
  group_by(complication)

## # A tibble: 4 x 3
## # Groups:   complication [2]
##   mort30 complication     n
##   <fct>    <fct>     <int>
## 1 0         0             10924
## 2 0         1             3653
## 3 1         0              21
## 4 1         1              37
```

A simple vista, en ambas variables **no existe una clara separación sobre la variable objetivo**. Por tanto, se ha tomado la decisión de descartar ambas columnas del conjunto de datos.

```
surgical_dataset$baseline_dementia <- NULL; surgical_dataset$mort30 <- NULL

# Actualizamos el vector con las variables categóricas
cat_columns <- setdiff(cat_columns, c("baseline_dementia", "mort30"))
```

3.3.1 Agrupación de variables categóricas

Por otro lado, nos encontramos con dos variables cuyas categorías pueden ser agrupadas, según la información proporcionada por el paquete *scorecard*:

DÍA DE LA SEMANA (dow):

Sobre dicha variable, **observamos una relación “lineal” en la distribución de la variable objetivo a lo largo de los diferentes días de la semana**, comenzando por el Lunes (0), con el menor porcentaje de complicaciones hospitalarias (alrededor del 14 %), seguido de los Martes-Miércoles-Jueves, donde el porcentaje aumenta hasta el 30.4 %, y finalizando con los viernes, donde se alcanza el mayor porcentaje de complicaciones hospitalarias: 34.5 %:

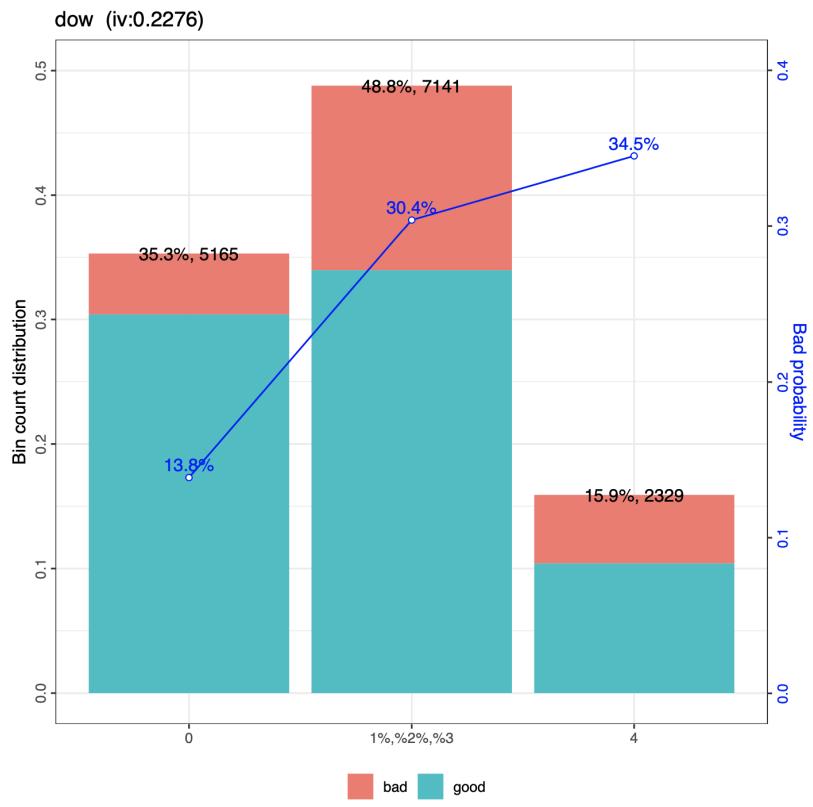
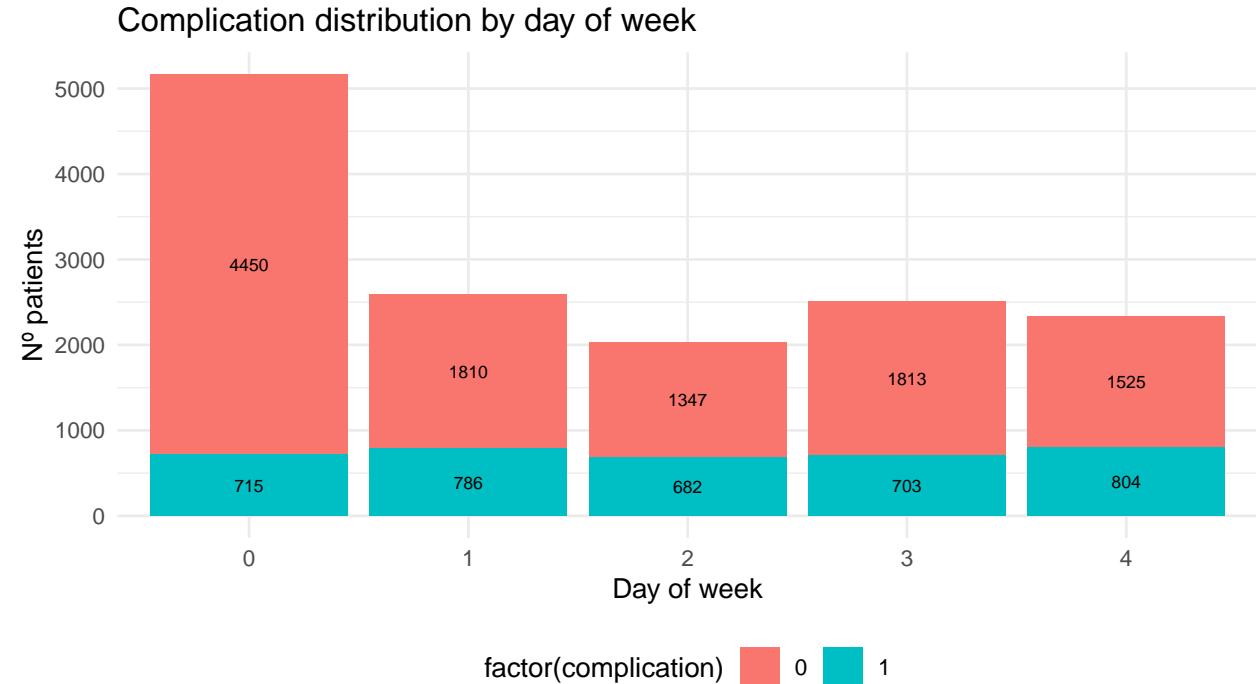


Figure 2: Dia de la semana o dow (IV)

Por otro lado, si analizamos detenidamente el gráfico de distribución:



Observamos que la proporción de aparición de pacientes con complicaciones es muy similar entre los martes, miércoles y jueves:

```

##          dow sin.comp con.comp total prop.compliacion
## 1             1     4450      715  2596        30.3
## 2             2     1810      786  2029        33.6
## 3             3     1347      682  2516        27.9
## 4 En conjunto (1-2-3) 1813      703  7141        30.4
## 5             4     1525      804  2329        34.5

```

En conjunto, acumulan alrededor del 30.4 % de pacientes con complicaciones, mientras que con tan solo el viernes aumenta hasta alcanzar el 34 %. Por tanto, dado que los martes, miércoles y jueves presentan una proporción de aparición similar, las agrupamos en torno a una misma categoría:

1. Lunes (0)
2. Martes-Miercoles-Jueves (1-3)
3. Viernes (4)

MES (month):

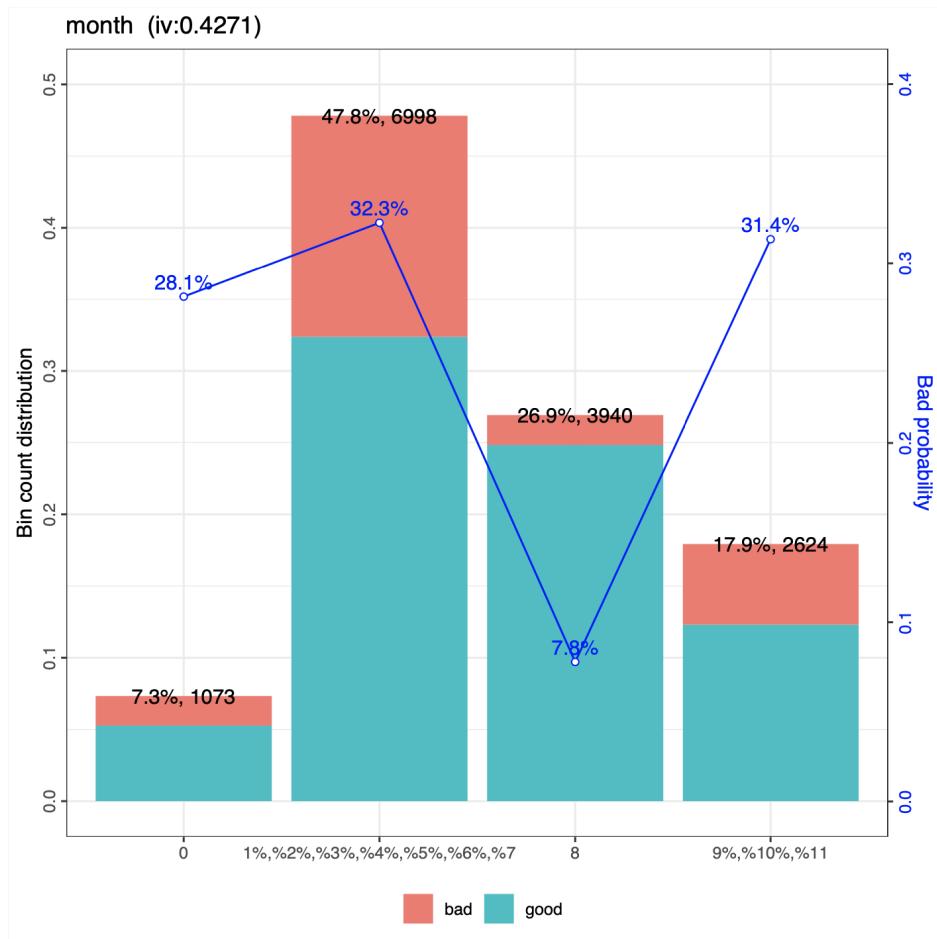
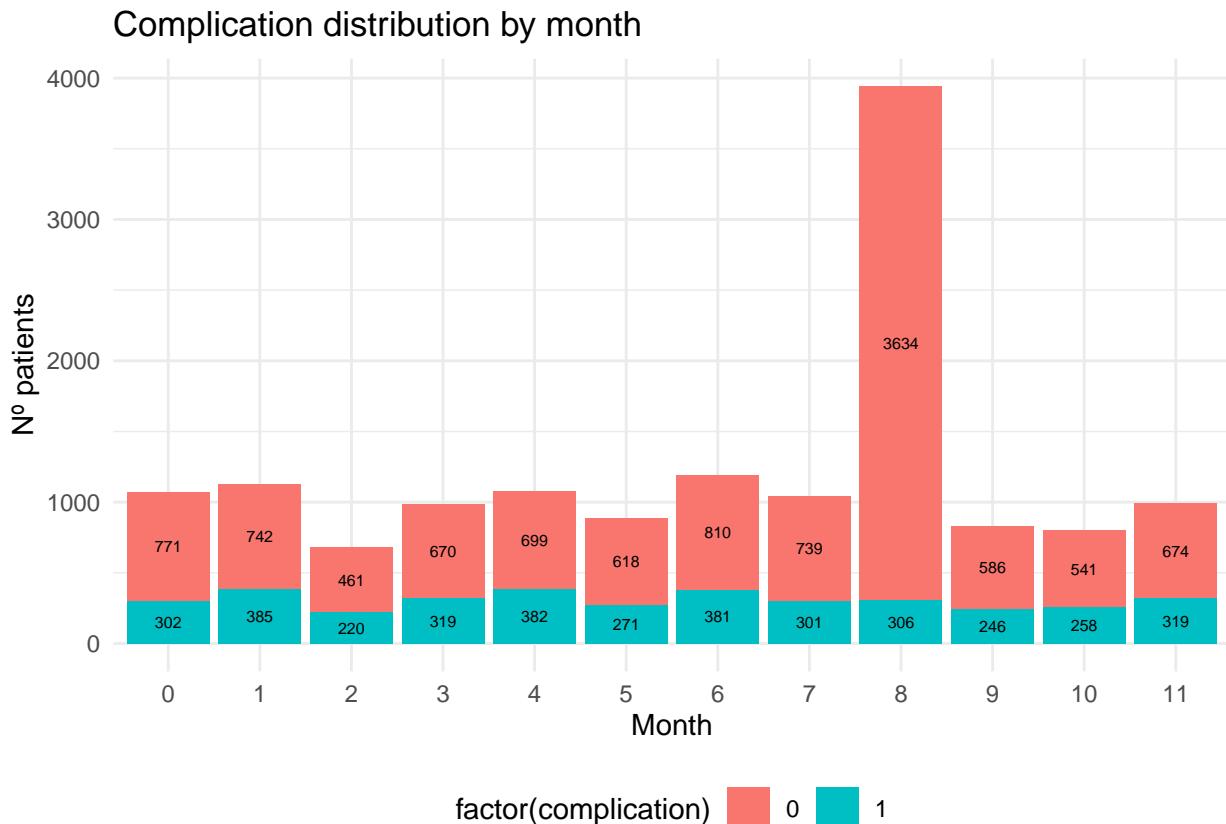


Figure 3: Month (IV)

En este caso, llaman la atención tres principales grupos: en primer lugar el mes de enero (0), con un 28.1 % de las complicaciones hospitalarias, **seguido de los meses de febrero (1) hasta agosto (7) con un total acumulado del 32.3 % de los pacientes con complicaciones**, es decir, el mes de enero tiene un porcentaje similar de pacientes con complicaciones que los siguientes 7 meses en conjunto. Por el contrario, **durante el mes de septiembre (8) el porcentaje se desploma hasta el 7.8 %**, porcentaje que vuelve a aumentar en los tres meses siguientes (octubre, noviembre y diciembre), hasta el 31.4 %.

Por otro lado, si analizamos el gráfico de distribución:



Sucede un comportamiento similar al de la variable *dow*: salvo el mes de septiembre, **la distribución de la variable objetivo sobre cada mes es muy similar, de forma que podemos agrupar varios de los meses en una misma categoría, tal y como hemos comprobado anteriormente.** A modo de ejemplo, analicemos la proporción en el número de complicaciones hospitalarias de cada mes por separado y en conjunto:

	dow	sin.comp	con.comp	total	prop.compliacion
## 1	0	771	302	1073	28.1
## 2	1	742	385	1127	34.2
## 3	2	461	220	681	32.3
## 4	3	670	319	989	32.3
## 5	4	699	382	1081	35.3
## 6	5	618	271	889	30.5
## 7	6	810	381	1191	32.0
## 8	7	739	301	1040	28.9
## 9 Uniendo 1-7	4739	2259	6998		32.3
## 10	8	3634	306	3940	7.8
## 11	9	586	246	832	29.6
## 12	10	541	258	799	32.3
## 13	11	674	319	993	32.1
## 14 Uniendo 9-11	1801	823	2624		31.4

Como podemos comprobar, **la proporción de pacientes con complicaciones hospitalarias, entre los meses de febrero (1) y agosto (7), se sitúa en torno al 32 % si agrupamos dichas categorías.** Del mismo modo, **los meses de octubre, noviembre y diciembre (9, 10 y 11) se sitúan en torno al 31 %.** Como consecuencia, y dado que dichas categorías presentan una proporción similar en cuanto a pacientes con complicaciones se refiere, las agrupamos:

1. Enero (0)
2. Febrero a Agosto (1-7)
3. Septiembre (8)
4. Octubre, Noviembre y Diciembre (9-10-11)

En relación con el resto de variables categóricas, si analizamos su valor de información:

```
##           variable      iv
## 1   baseline_osteoart 0.5246
## 2             month 0.4271
## 3             dow 0.2276
## 4       moonphase 0.2119
## 5   baseline_cancer 0.1358
## 6   baseline_cvd 0.0429
## 7            gender 0.0221
## 8 baseline_digestive 0.0134
## 9      asa_status 0.0062
## 10 baseline_pulmonary 0.0053
## 11 baseline_diabetes 0.0013
## 12            race 0.0002
## 13 baseline_psych 0.0001
```

Por lo general, la mayoría de las variables categóricas presentan, como primera impresión, un buen poder predictivo, con ciertas excepciones como *asa_status*, *baseline_pulmonary*, *baseline_diabetes*, *race* o *baseline_psych*, cuyo IV no alcanza el 0.01 (como normal general, un valor de información inferior a 0.1-0.02 se traduce en un poder predictivo muy bajo o prácticamente nulo)⁵. De este modo, de cara a la selección de variables, **en caso de ser necesario descartar alguna variable categórica, utilizaremos esta tabla como referencia.**

3.4 Variables continuas

En relación con las variables continuas, nos encontramos con algunas variables con un número de valores únicos muy reducidos:

```
-- N° Valores únicos (continuas)
apply(surgical_dataset[, num_columns], 2, function(x) {length(unique(x))})
```

##	bmi	Age	baseline_charlson	ahrq_ccs
##	3095	672	14	22
##	ccsMort30Rate	hour	mortality_rsi	
##	20	725	633	

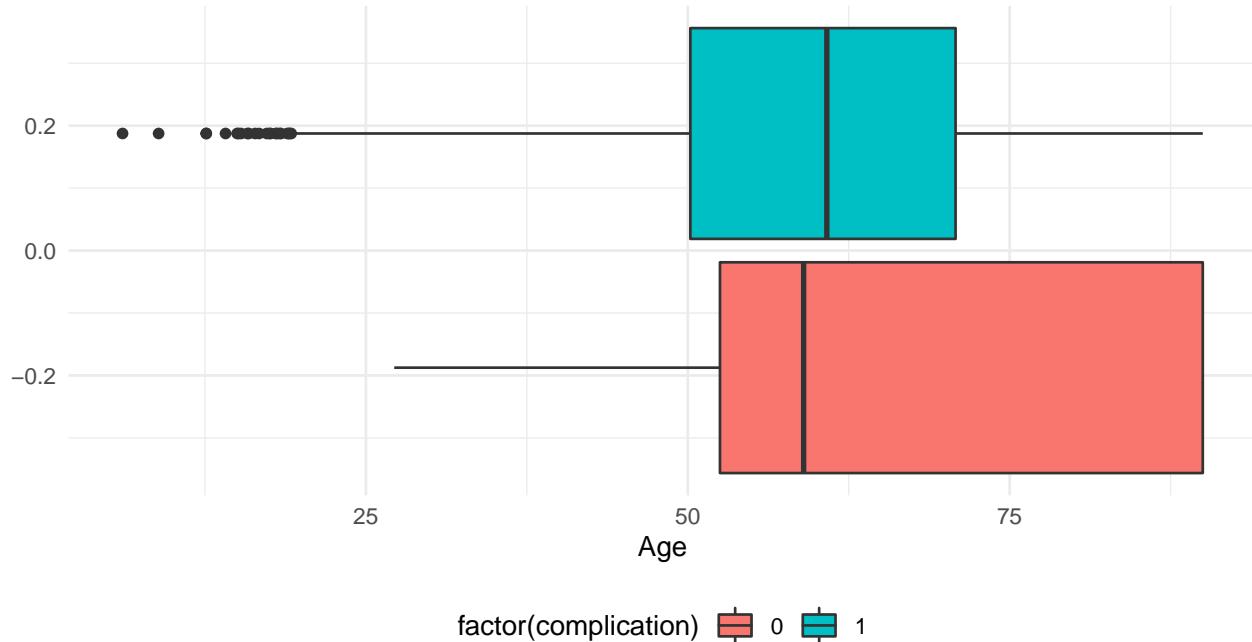
A modo de ejemplo, variables como *baseline_charlson*, *ahrq_ccs* o *ccsMort30Rate* presentan 14, 22 o 20 valores únicos, respectivamente. No obstante, se tomó la decisión de mantener dichas variables como continuas, principalmente para facilitar la selección de variables, evitando con ello crear nuevas variables *dummy* (una por cada categoría), utilizando con ello menos parámetros para describir el conjunto de datos.

Por otro lado, en un primer análisis exploratorio se observó que en las variables continuas, por lo general, **no existe una clara separación entre ambas variables objetivo**. A modo de ejemplo, observemos las dos siguientes variables: *age* y *bmi*:

AGE:

⁵https://docs.tibco.com/pub/sfire-dsc/6.5.0/doc/html/TIB_sfiredsc_user-guide/GUID-07A78308-525A-406F-8221-9281F4E9D7CF.html

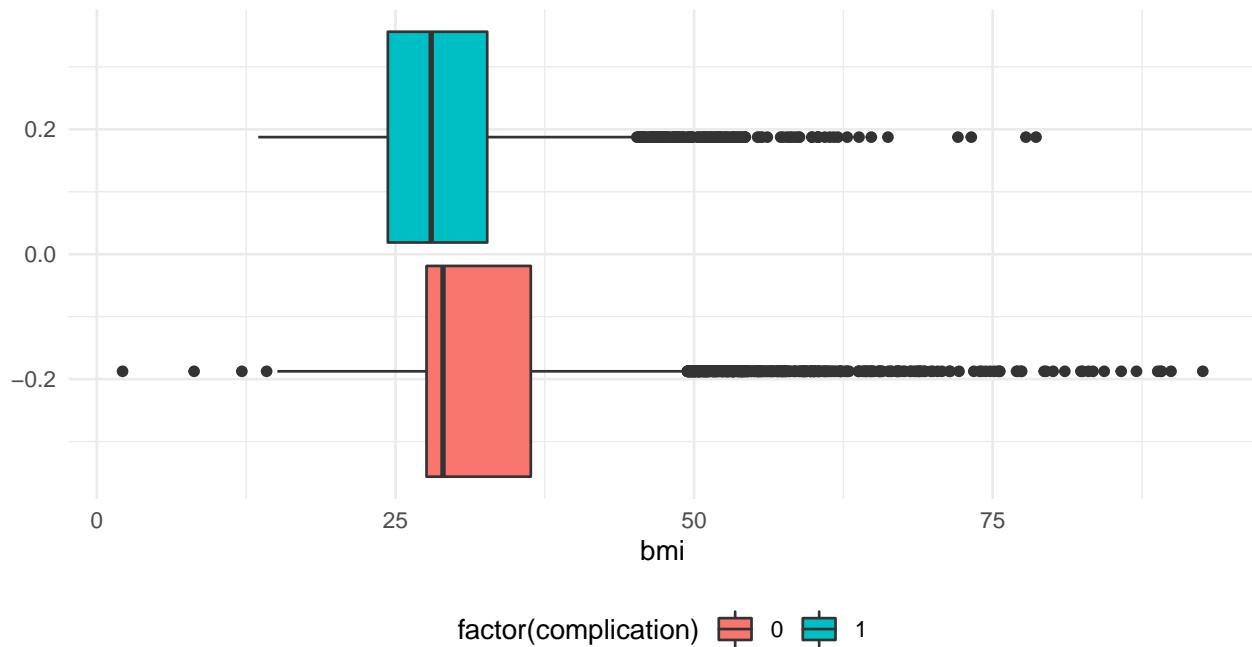
Age boxplot by target



En este caso, bien es cierto que los pacientes que no sufren complicaciones oscilan entre los 25 y los 90 años, aproximadamente, mientras que existe un número de pacientes (menores a 25 años), donde padecen alguna complicación hospitalaria. Pese a ello, la separación no es tan clara.

BMI:

bmi boxplot by target



Incluso con el índice de masa de corporal, a simple vista no existe una clara separación que diferencie a ambas variables objetivo en función de dicho índice.

3.5 Estandarización de variables continuas

Una vez realizado el primer análisis, procedemos a la estandarización de las variables continuas:

```
# --- Estandarizacion de variables
surgical_dataset_stnd <- surgical_dataset

# media
media      <- sapply(surgical_dataset_stnd[, num_columns], mean)

# sd
desv.tipica <- sapply(surgical_dataset_stnd[, num_columns], sd)

surgical_dataset_stnd[, num_columns] <- scale(surgical_dataset_stnd[, num_columns],
                                               center = media,
                                               scale = desv.tipica)
```

3.6 Creacion de variables dummy

A continuación, convertimos las variables categóricas (con más de una categoría) en variables *dummy*, mediante la función *dummy.data.frame*, convirtiendo de este modo todas las variables a formato numérico:

```
columnas_dummy <- c("asa_status", "dow", "month", "moonphase", "race")
surgical_dataset_stnd_dummy <- dummy.data.frame(surgical_dataset_stnd[, columnas_dummy],
                                                sep = ".")
```

Una vez codificadas las variables, comprobamos si existen variables *dummy* con una frecuencia de aparición menor a 100:

```
names(surgical_dataset_stnd_dummy[, colSums(surgical_dataset_stnd_dummy == 0) < 100,
                                    drop = FALSE])
```

```
## character(0)
```

Como podemos comprobar, todas las variables *dummy* codificadas presentan una frecuencia superior. A continuación, unimos en un único *data.table* tanto las variables numéricas, las variables categóricas (binarias), las variables *dummy*, así como la variable objetivo:

```
surgical_dataset_final <- cbind(
  surgical_dataset_stnd[, num_columns],
  surgical_dataset_stnd[, cat_columns[!cat_columns %in% columnas_dummy]],
  surgical_dataset_stnd_dummy,
  surgical_dataset_stnd[, target]
)
```

Finalmente, de cara a la elaboración de los modelos codificamos la variable objetivo como “Yes” / “No”, renombrando la variable como *target*:

```
## Renombramos la variable objetivo como "target"
names(surgical_dataset_final)[33] <- "target"

# Renombramos las columnas para adecuarlas a formulas
names(surgical_dataset_final) <- make.names(names(surgical_dataset_final))

# 1 - "Yes" ; 0 - "No"
surgical_dataset_final$target <- ifelse(
  surgical_dataset_final$target == 1,
  "Yes",
```

```

    "No"
)

surgical_dataset_final$target <- as.factor(surgical_dataset_final$target)

## Numero de variables finales (dummies incluidas): 32

```

4. Selección de variables bajo logística

4.1 Selección de una submuestra

Durante la selección de variables, e incluso con determinados modelos como redes neuronales o SVM, el tiempo de cómputo que requiere era demasiado elevado. Como consecuencia, se planteó utilizar, en lugar del *dataset* completo con 14.000 observaciones, **un subconjunto de menor tamaño**. Dado que la variable objetivo se encuentra desbalanceada, como pudimos comprobar en el apartado de Depuración, recurrimos al **muestreo estratificado**, el cual nos asegura la proporción de la variable objetivo. Para ello, *caret* dispone de la función *createDataPartition*, con el que obtener una submuestra estratificada. Concretamente, para el desarrollo del proyecto empleamos un subconjunto con el **40 % de las observaciones**:

```

-- Muestreo estratificado
set.seed(1234)
partitions <- createDataPartition(surgical_dataset_final$target, p = 0.40, list = FALSE)

surgical_dataset_final_est <- surgical_dataset_final[partitions, ]

# Mantenemos la misma proporcion en la variable objetivo
table(surgical_dataset_final_est$target)

##
##   No   Yes
## 4378 1476

```

No obstante, con un 40 % de los datos ¿Es suficiente? Más importante aún ¿Se obtienen los mismos resultados o similares? Es decir, si nos fijamos en la depuración final, nos encontramos con $14.635 / 32 \sim 457$ observaciones por variable, aproximadamente. Sin embargo, con tan solo el 40 %, obtendríamos $(14.635 * 0.4) / 32 \sim 183$ observaciones por variable. Es decir, con una menor proporción de observaciones por variable, es posible que los modelos no sean exactamente iguales con ambos conjuntos.

Por ello, durante la selección de variables **aplicaremos las mismas técnicas sobre ambos conjuntos**:

1. *stepwise AIC*
2. *stepwise BIC*
3. *RFE o Recursive Feature Elimination (sobre logística)*
4. *RFE (sobre Random Forest)*

A continuación, y en caso de obtener una selección de variables similar, aplicamos un modelo logístico en ambos casos, comprobando de este modo si los resultados son similares, incluso con tan solo el 40 % de las observaciones. De ser así, de cara al resto de modelos trabajaremos con el subconjunto.

4.2 Stepwise AIC

Comenzamos con la selección de variables bajo *stepwise AIC*:

```

-- Stepwise AIC (100 repeticiones)
lista.variables.aic <- steprepetidobinaria(data=surgical_dataset,
                                              vardep=target, listconti=vars,
                                              sinicio=1234, sfinal=1334,

```

```

porcen=0.8,criterio="AIC")
tabla.aic <- lista.variables.aic[[1]]

##      n variable_dataset_original   variable_subset
## 1     1           mortality_rsi   mortality_rsi
## 2     2           ccsMort30Rate  ccsMort30Rate
## 3     3             bmi         bmi
## 4     4        month.8       month.8
## 5     5      baseline_cvd   baseline_cvd
## 6     6            dow.0       dow.0
## 7     7             Age         Age
## 8     8      moonphase.0   moonphase.0
## 9     9        month.0       month.0
## 10    10      asa_status.0  asa_status.0
## 11    11  baseline_osteoart baseline_osteoart
## 12    12  baseline_charlson baseline_charlson
## 13    13          ahrq_ccs      ahrq_ccs
## 14    14  baseline_diabetes baseline_diabetes
## 15    15      baseline_cancer      -
## 16    16      baseline_psych      -
## 17    17      asa_status.1      -

```

Como podemos observar, la diferencia entre el *dataset* original y la submuestra es de tan solo 3 variables: *baseline_cancer*, *baseline_psych* y *asa_status.1*.

4.3 Stepwise BIC

Por otro lado, analicemos la selección de variables bajo *stepwise BIC*:

```

##-- Stepwise BIC (100 repeticiones)
lista.variables.bic <- steprepetidobinaria(data=surgical_dataset,
                                              vardep=target,listconti=vars,
                                              sinicio=1234,sfinal=1334,
                                              porcen=0.8,criterio="BIC")
tabla.bic <- lista.variables.bic[[1]]

##      n variable_dataset_original   variable_subset
## 1     1           mortality_rsi   mortality_rsi
## 2     2           ccsMort30Rate  ccsMort30Rate
## 3     3             bmi         bmi
## 4     4        month.8       month.8
## 5     5            dow.0       dow.0
## 6     6             Age         Age
## 7     7      moonphase.0   moonphase.0
## 8     8      baseline_osteoart baseline_osteoart
## 9     9      asa_status.0   asa_status.0
## 10    10      baseline_cancer      -

```

En este caso, la diferencia en ambas selecciones es de tan solo una única variable: *baseline_cancer*.

4.4 Recursive Feature Elimination (bajo logística)

No obstante, en ambas selecciones de variables el número de parámetros es bastante elevado, en especial con *stepwise AIC*. Como consecuencia, a los métodos anteriores añadimos una selección de variables mediante *Recursive Feature Elimination*, tanto bajo logística como con *Random Forest*:

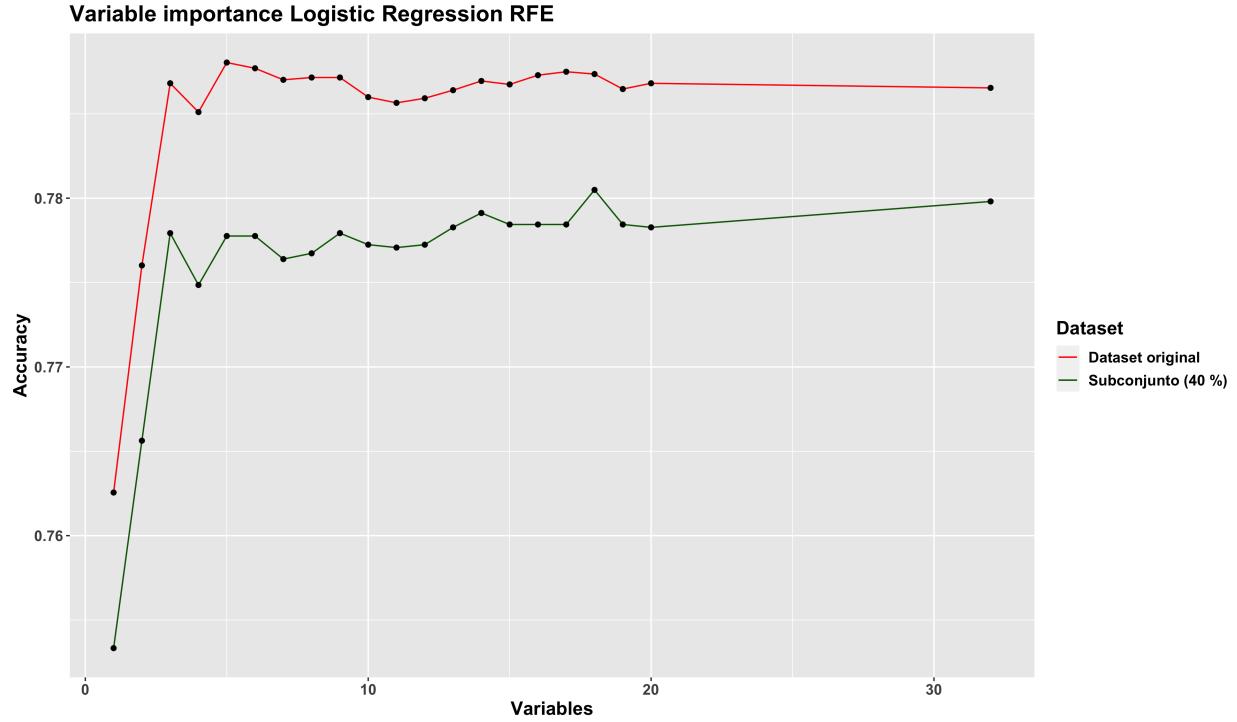


Figure 4: RFE Logistic Regression

Analizando el gráfico resultante, observamos que la diferencia de *Accuracy* en ambos *datasets* no es muy significativa: mientras que con el *dataset* completo se alcanza un valor en torno a 0.78, con el subconjunto la precisión se reduce a tan solo 0.77. De hecho, en ambos casos se obtienen muy buenos resultados con tan solo tres variables, y en ambos casos coinciden:

```
## Top 3 variables:
```

```
## [1] ccsMort30Rate
## [2] mortality_rsi
## [3] bmi
```

4.5 Recursive Feature Elimination (bajo Random Forest)

En el caso de *Random Forest*, con 4-5 parámetros se obtienen resultados muy similares: en el caso el *dataset* original en torno a 0.90 y 0.88 el caso del subconjunto. De hecho, en el punto de máximo *accuracy*, ambos modelos comparten las mismas variables, a excepción de *ahrq_ccs*:

```
##   n variable_dataset_original variable_subset
## 1 1                 Age          Age
## 2 2      ccsMort30Rate    ccsMort30Rate
## 3 3     mortality_rsi   mortality_rsi
## 4 4           bmi         bmi
## 5 5             -       ahrq_ccs
```

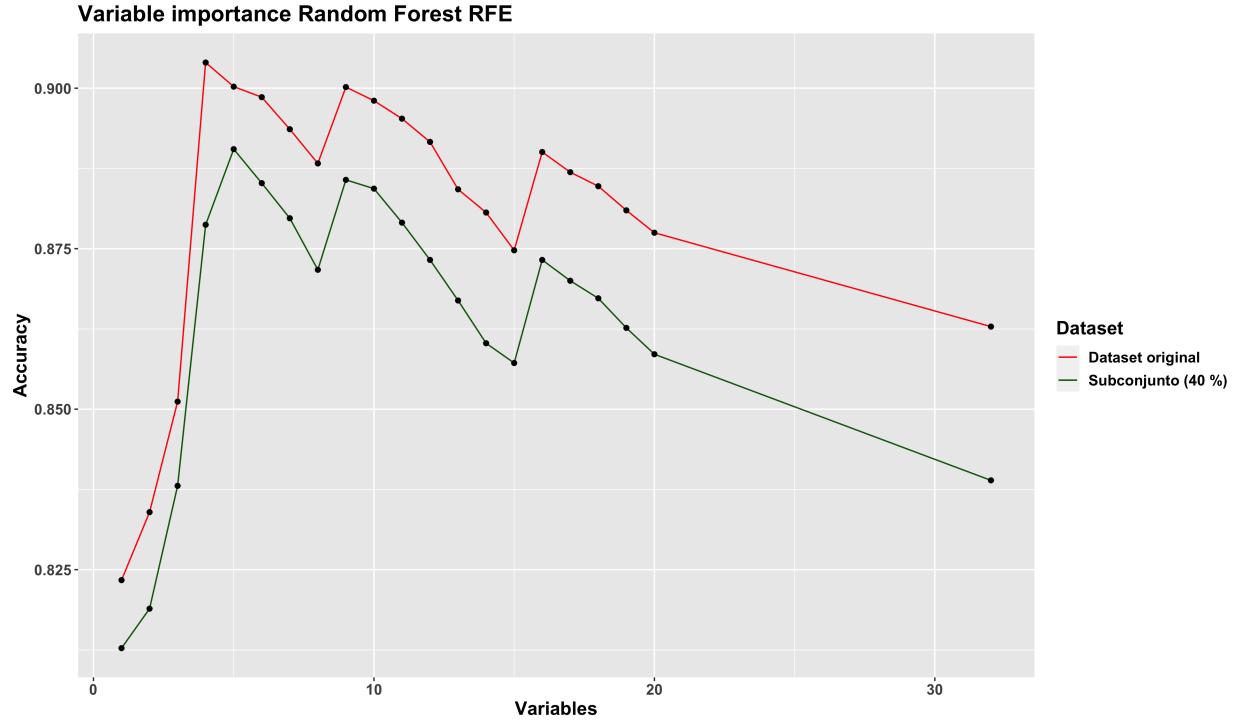


Figure 5: RFE Random Forest

4.6 Selección bajo logística

4.6.1 Comparación datasets

A continuación, y una vez realizada la selección de variables tanto por AIC/BIC como por RFE, **realizamos una primera comparación, bajo logística, de la selección de variables obtenidas en los métodos anteriores, tanto con el dataset original como con el subconjunto:**

Como primera impresión, y dada la escala del eje Y, la diferencia en cuanto a tasa de fallos y AUC se refiere es muy pequeña, concretamente de 0.01, aproximadamente (0.21 en tasa de fallos y 0.79 en AUC frente a 0.22 y 0.78).

Por tanto, dado que la diferencia entre ambos *datasets* es pequeña, **de cara al resto de la práctica se trabajó con el subconjunto del 40 %.** No obstante, en los últimos apartados (y una vez tuneados los modelos), se realiza una última comparación con el fichero original, comprobando de este modo si el orden de los algoritmos se conserva. Además, de cara a la evaluación de los modelos se ha tomado la decisión de utilizar el resto de observaciones del *dataset* original como conjunto *test* con el que evaluar (como primera impresión) la precisión del modelo.

4.6.2 Selección de los mejores sets de variables

Una vez realizada la comparación con el *dataset* original, nos centramos en la selección de variables obtenida por el subconjunto, tanto por *stepwise AIC/BIC* como por RFE. En primer lugar, y remontándonos al daigrama de cajas anterior, debemos recordar la diferencia en el número de variables entre *stepwise AIC* y *BIC*: con 14 y 9 variables, respectivamente, **la tasa de fallos es prácticamente idéntica, y en relación al valor AUC, la diferencia es de tan solo unas milésimas, pues ambos modelos se sitúan en torno a 0.78.** Por tanto, el hecho de incluir demasiadas variables no afecta en gran medida al modelo, lo cual puede traducirse en un sobreajuste en el resto de algoritmos.

En consecuencia, en ambas selecciones **probamos a eliminar las variables categóricas menos relevantes, concretamente aquellas con un menor poder predictivo**, tal y como pudimos observar en el

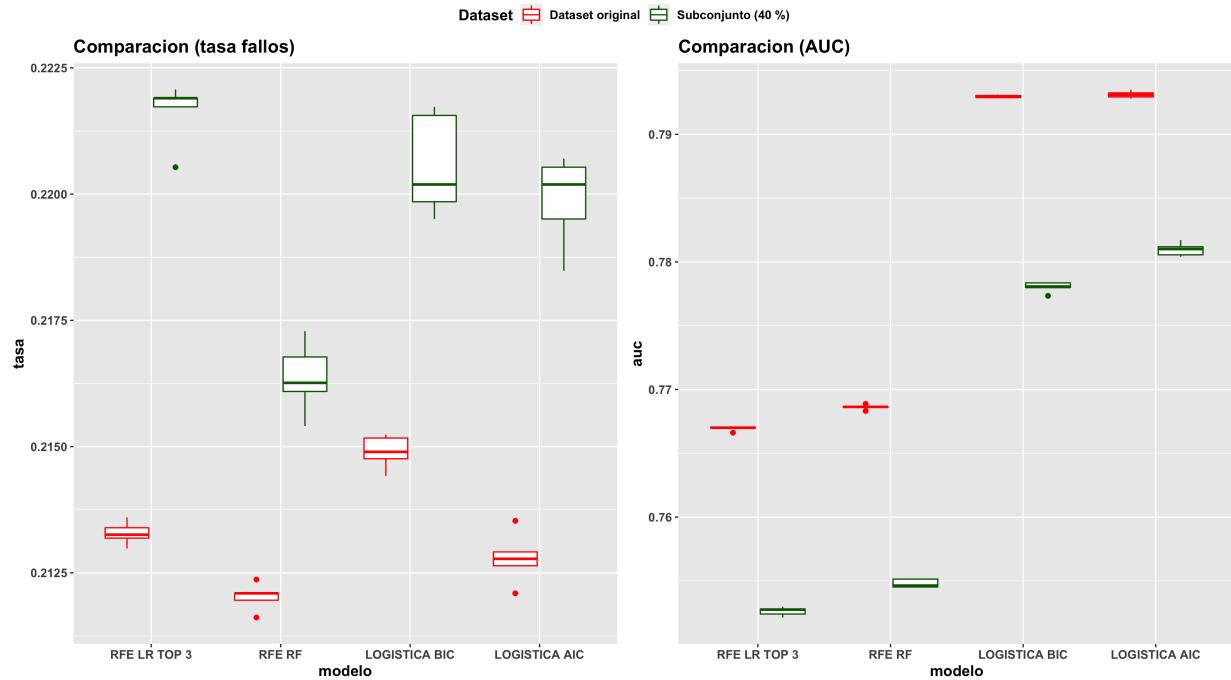


Figure 6: Comparacion bajo logística (I)

apartado de depuración, gracias al **Valor de Información**:

1. En el caso de *stepwise AIC*, eliminamos los campos *baseline_cvd*, *asa_status.0* y *baseline_diabetes* (iv: 0.04, 0.0062 y 0.0013, respectivamente).
2. En el caso de *stepwise BIC*, eliminamos el campo *asa_status.0* (iv: 0.0062).

```
##      n      stepwise_aic      stepwise_bic
## 1    1      mortality_rsi      mortality_rsi
## 2    2      ccsMort30Rate      ccsMort30Rate
## 3    3          bmi          bmi
## 4    4      month.8      month.8
## 5    5      dow.0      dow.0
## 6    6          Age          Age
## 7    7      moonphase.0      moonphase.0
## 8    8      month.0 baseline_osteoart
## 9    9 baseline_osteoart          -
## 10   10 baseline_charlson          -
## 11   11      ahrq_ccs          -
```

Analicemos tanto la tasa de fallos como el valor AUC:

Incluso reduciendo el número de variables en ambos casos, aunque la tasa de fallos aumente o el valor auc disminuya ligeramente, la diferencia no es muy significativa.

No obstante, uno de los aspectos que más ha llamado la atención ha sido la selección de variables RFE con Random Forest, donde con tan solo 5 variables la accuracy aumentaba hasta cerca del 90 %, mucho más alto que un modelo logístico. Por tanto, ¿Y si entrenamos un pequeño modelo random forest para observar la importancia de las variables tanto en *stepwise AIC* como en *BIC*?

```
-- mtry (sqrt(numero de variables), por defecto)
rf_modelo_bic <- train_rf_model(surgical_dataset, formula.candidato.bic.2, ntree = 1000,
```

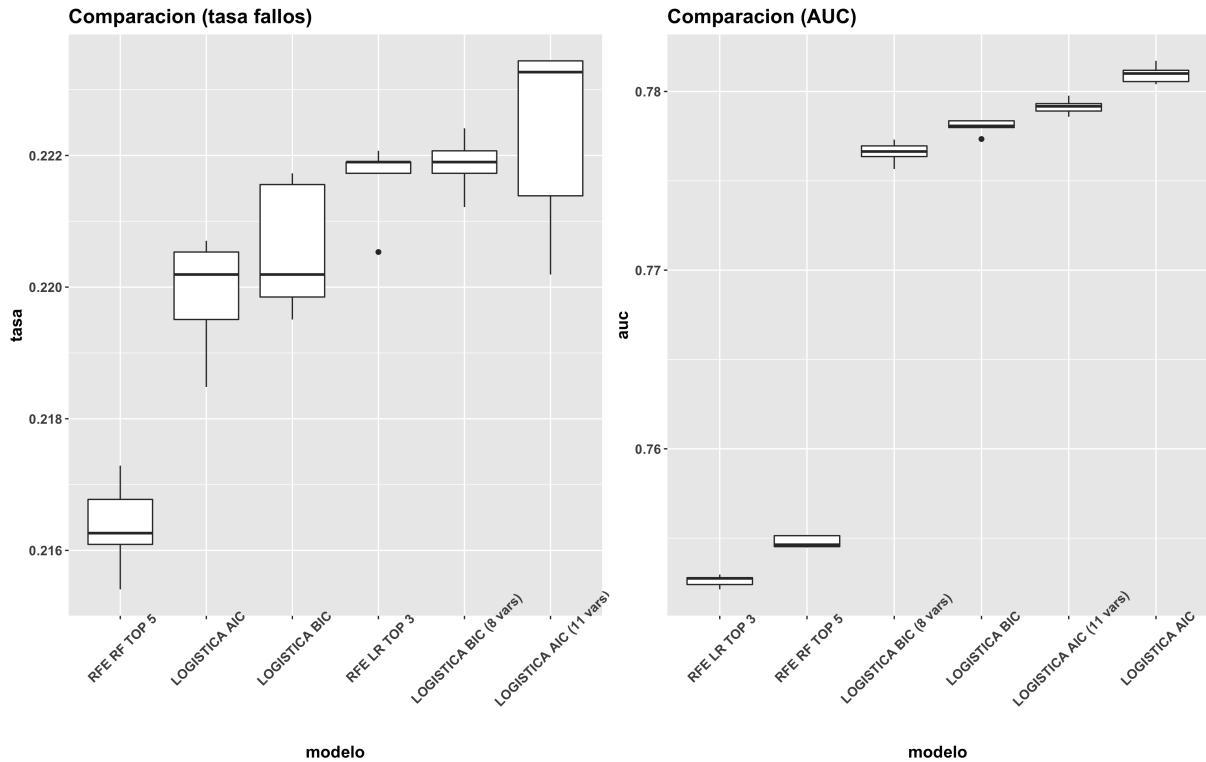


Figure 7: Comparacion bajo logística (II)

```
grupos = 5, repe = 5, nodesize = 10, seed = 1234)
```

Analizando el gráfico de importancia, caben destacar cuatro principales variables, **las cuales coinciden en ambos modelos, tanto en stepwise AIC como BIC**: *Age, mortality_rsi, bmi* y *ccsMort30Rate*. Por otro lado, en el modelo *AIC* cabe destacar, además, la variable *ahrq_ccs*. En relación con el modelo *BIC*, el contraste entre las cuatro primeras variables y el resto de *features* es más significativo, de las cuales caben destacar *baseline_osteart* y *month.8*, con un nivel de importancia similar en el *Random Forest*.

Por tanto, vista la importancia que presentan las variables en el modelo *Random Forest*, cabría preguntarse si realmente es necesario un modelo con 11 u 8 variables, como es el caso de *stepwise AIC* o *BIC*, respectivamente. Es decir, ¿Y si reducimos el modelo a las 4-5 variables más relevantes? Por ejemplo, una última comparación bajo logística consistiría en (y en base al gráfico de importancia anterior) analizar un modelo con las cuatro variables más importantes:

1. *Age, mortality_rsi, bmi* y *ccsMort30Rate*.

Otro modelo logístico con las cinco variables más relevantes del *set* de variables *BIC*:

2. *Age, mortality_rsi, bmi, ccsMort30Rate* y *ahrq_ccs* (coincide con el modelo obtenido en *RFE* con *Random Forest*, por lo que lo denotamos como *RFE RF TOP 5 (AIC TOP 5)*).

Además de otros dos modelos con las cinco variables más relevantes en el *set* de variables *AIC* (dado que *baseline_osteart* y *month.8* son muy similares en cuanto a importancia se refiere, realizamos la prueba con ambos *sets*):

3. *Age, mortality_rsi, bmi, ccsMort30Rate* y *baseline_osteart*.
4. *Age, mortality_rsi, bmi, ccsMort30Rate* y *month.8*.

Como podemos comprobar, aunque la tasa de fallos sea similar tanto con *baseline_osteart* como con *month.8*, el valor AUC es ligeramente superior, incluso cercano a los valores obtenidos por la selección *AIC, BIC*

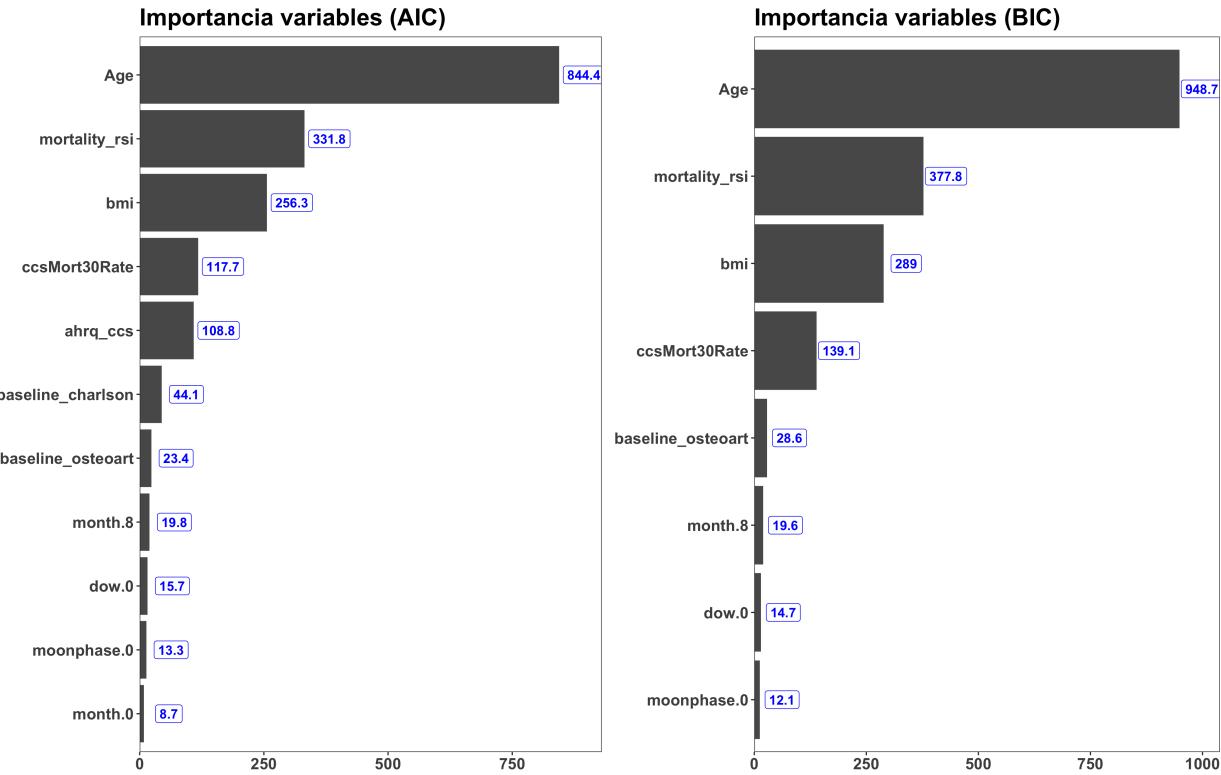


Figure 8: Importancia variables Random Forest (AIC y BIC)

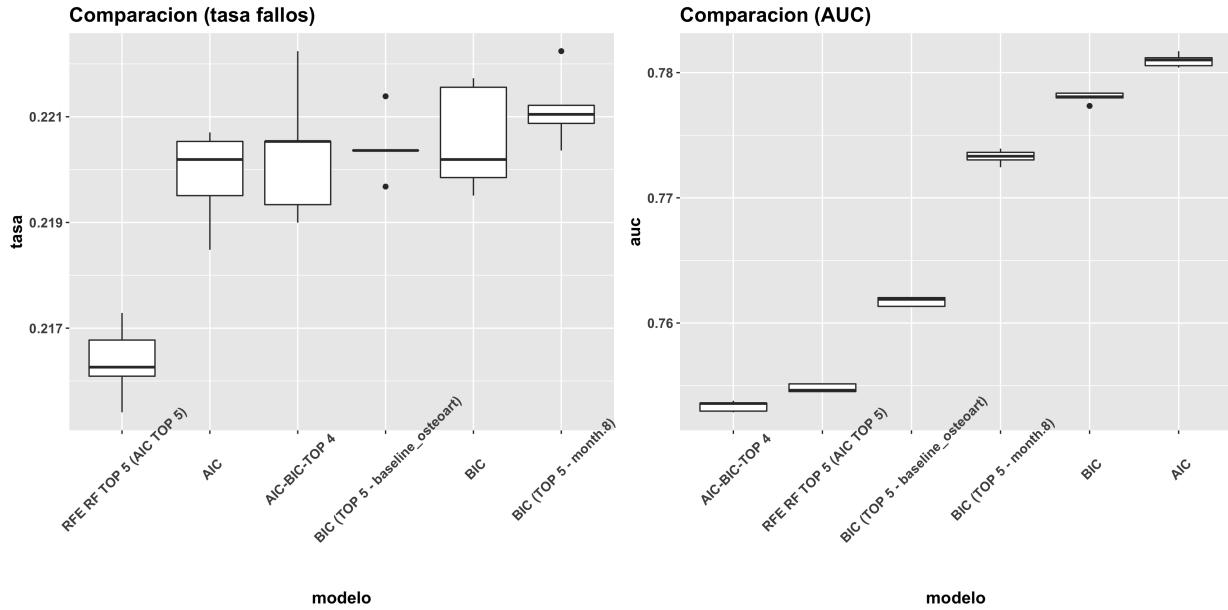


Figure 9: Comparacion bajo logística (III)

original. Por otro lado, con tan solo cuatro variables, aunque el valor AUC sea ligeramente inferior con respecto al resto de *sets*, la diferencia no es tan significativa (0.755 a 0.775, aproximadamente).

Por tanto, de cara al resto de modelos **emplamos dos modelos candidatos**:

```

##      n      modelo1      modelo2
## 1 1      Age      Age
## 2 2 mortality_rsi mortality_rsi
## 3 3      bmi      bmi
## 4 4 ccsMort30Rate ccsMort30Rate
## 5 5 month.8      -
#--- Variables de los modelos candidatos
#-- Modelo 1
var_modelo1 <- c("mortality_rsi", "ccsMort30Rate", "bmi", "month.8", "Age")

#-- Modelo 2
var_modelo2 <- c("mortality_rsi", "bmi", "month.8", "Age")

```

4.7 Tuneo y comparación final

Finalmente, realizamos una última comparación de ambos modelos candidatos, **aumentando la validación cruzada de 5 a 10 repeticiones**:

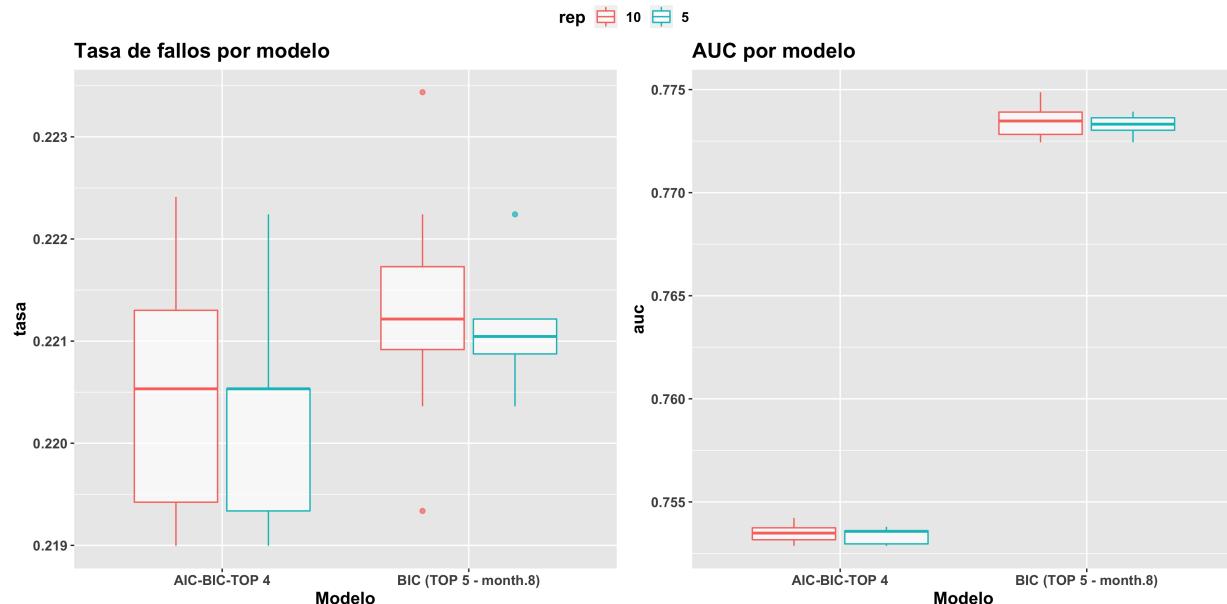


Figure 10: Comparacion final logística (5-10 rep.)

A primera vista, al aumentar el número de repeticiones, la varianza en ambos modelos es muy similar, con una ventaja en el segundo modelo en cuanto a AUC se refiere (0.775 frente a 0.755, aproximadamente). Por otro lado, si nos fijamos en las estadísticas obtenidas a partir de los datos *test*:

```

##              modelo precision sensibilidad especificidad valor_pred_pos
## 1 Modelo 1 (BIC TOP 5)    0.7940      0.67730      0.81011      0.33013
## 2     Modelo 2 (top 4)    0.7824      0.66707      0.79434      0.25194
##   valor_pred_neg
## 1          0.94783
## 2          0.95830

```

Observamos que la sensibilidad de ambos modelos ronda el 66-67 %, mientras que la especificidad aumenta hasta el 80 %, aproximadamente. Es decir, con un modelo lineal como es el caso de la regresión logística conseguimos clasificar la mayoría de los pacientes a 0 (sin complicaciones). Sin embargo, tanto el valor

predictivo positivo como la sensibilidad nos indican justo con lo contrario en relación a los pacientes con complicaciones:

1. **Valor predictivo positivo** (en torno al 25-33 % en ambos modelos), lo que se traduce en un alto número de falsos positivos.
2. **Sensibilidad** (en torno al 66-67 % en ambos modelos), lo que se traduce en un alto número de falsos negativos.

5. Modelos iniciales con H2O

Una realizada la selección de variables y decantarnos por dos posibles *sets* de variables, antes de comenzar con el tuneo de hiperparámetros, **realizamos un modelo *autoML* por cada *set* de variables**. De este modo, podremos comprobar, de forma orientativa:

1. Qué modelo o modelos son los más adecuados, tanto a nivel AUC como en tasa de fallos.
2. Del mejor o mejores modelos, qué parámetros se ha empleado.

5.1 Modelo 1 (BIC TOP 5)

```
#  Modelo 1
aml_1 <- h2o.automl(x = var_modelo1, y = target,
                     training_frame = surgical_dataset_h, nfolds = 5, seed = 1234,
                     balance_classes = TRUE, keep_cross_validation_predictions = TRUE,
                     max_models = 10)

model_id      auc  logloss mean_per_class_error
1           GBM_5_AutoML_20210422_174850 0.9211350 0.2591059          0.1608837
2   StackedEnsemble_BestOfFamily_AutoML_20210422_174850 0.9208431 0.2647496          0.1603029
3           GBM_1_AutoML_20210422_174850 0.9200581 0.2690935          0.1659645
4   StackedEnsemble_AllModels_AutoML_20210422_174850 0.9200033 0.2647962          0.1676764
5           GBM_2_AutoML_20210422_174850 0.9190857 0.2695043          0.1648937
6           XGBoost_3_AutoML_20210422_174850 0.9189663 0.2613378          0.1651901
7   XGBoost_grid__1_AutoML_20210422_174850_model_1 0.9189402 0.2629065          0.1675371
8           GBM_grid__1_AutoML_20210422_174850_model_1 0.9187735 0.2763313          0.1689585
9           GBM_3_AutoML_20210422_174850 0.9175951 0.2725404          0.1757392
10          XGBoost_2_AutoML_20210422_174850 0.9169047 0.2717032          0.1707306
```

En un primer análisis, de todos los modelos *autoML* creados, **los modelos *gradient boosting*, *ensemble* y *xgboost* obtienen los mejores resultados**, con un AUC en torno a 0.92, así como un error medio por cada clase de 0.16-0.17, aproximadamente. Por otro lado, si analizamos los parámetros del mejor modelo, concretamente *gradient boosting*:

```
##             modelo ntrees max_depth sample_rate col_sample_rate
## 1 Modelo 1 (BIC TOP 5)     82        15         0.8            0.8
##   col_sample_rate_per_tree
## 1                      0.8
```

Analizando los parámetros, por lo general *h2o* opta por un modelo *gbm* sencillo, con un número bajo de árboles (inferior a 100), un profundidad moderada en cada árbol (15), además de sortear no solo observaciones, sino además variables (en torno al 80 %).

5.2 Modelo 2 (AIC-BIC top 4)

	model_id	auc	logloss	mean_per_class_error
1	GBM_5_AutoML_20210422_182719	0.9096419	0.2700148	0.1646700

```

2 StackedEnsemble_BestOfFamily_AutoML_20210422_182719 0.9088350 0.2712222 0.1666722
3 StackedEnsemble_AllModels_AutoML_20210422_182719 0.9083655 0.2708531 0.1683963
4 XGBoost_3_AutoML_20210422_182719 0.9083145 0.2683037 0.1735557
5 GBM_1_AutoML_20210422_182719 0.9075638 0.2769396 0.1730899
6 GBM_2_AutoML_20210422_182719 0.9072236 0.2767946 0.1770094
7 XGBoost_grid__1_AutoML_20210422_182719_model_1 0.9065834 0.2696259 0.1678335
8 XGBoost_2_AutoML_20210422_182719 0.9064648 0.2756899 0.1671682
9 XGBoost_1_AutoML_20210422_182719 0.9047755 0.2746863 0.1725274
10 GBM_grid__1_AutoML_20210422_182719_model_1 0.9044822 0.3115999 0.1720491

```

Nuevamente, con el segundo *set* de variables continua optando por un modelo *gradient boosting*, con un AUC muy similar al primer *set* de variables (0.90 frente a 0.92). Además, si analizamos los parámetros del mejor modelo:

```

##           modelo ntrees max_depth sample_rate col_sample_rate
## 1   Modelo 1 (BIC TOP 5)     82        15         0.8          0.8
## 2 Modelo 2 (AIC-BIC-top4)    72        15         0.8          0.8
##   col_sample_rate_per_tree
## 1                      0.8
## 2                      0.8

```

Salvo el número de árboles, el resto de parámetros coinciden. Por tanto, de cara a la elaboración de los modelos, **comprobaremos si el mejor modelo en ambos sets es, efectivamente, un modelo gbm, o si al menos está entre los mejores.**

6. Redes neuronales

6.1 Modelo 1 (BIC TOP 5)

A continuación, procedemos con el tuneado del primer modelo *Machine Learning*: la red neuronal, comenzando con el primer *set* (5 variables input). En primera instancia, analicemos el número de nodos necesario para obtener entre 20 y 30 observaciones por parámetro, teniendo 5 variables *input*:

1. Con 20 observaciones por parámetro:

$$h * (k + 1) + h + 1 = 5854/20 = 292 \text{ parámetros}$$

Con $k = 5$ variables input, entonces: $7 * h + 1 = 292$. Es decir, 41 – 42 nodos

2. Con 30 observaciones por parámetro:

$$h * (k + 1) + h + 1 = 5854/30 = 195 \text{ parámetros}$$

Con $k = 5$ variables input, entonces: $7 * h + 1 = 195$. Es decir, 27 – 28 nodos

En primera instancia, dado que dispone de tan solo cinco variables , probablemente no serán necesarios tantos nodos. No obstante, realizamos un primer tuneo con varios tamaños (desde 5 hasta 40):

```

size.candidato.1 <- c(5, 10, 15, 20, 25, 30, 35, 40)
decay.candidato.1 <- c(0.1, 0.01, 0.001)

cvnnet.candidato.1 <- cruzadaavnnetbin(data=surgical_dataset,vardep=target,
                                         listconti=var_modelo1, listclass=c(""),
                                         grupos=5,sinicio=1234,repe=5, size=size.candidato.1,
                                         decay=decay.candidato.1,repeticiones=5,itera=200)

```

```

##   id size decay Accuracy
## 1  1    25 0.010 0.8950798
## 2  2    30 0.010 0.8947370
## 3  3    20 0.010 0.8945332
## 4  4    35 0.010 0.8939514
## 5  5    30 0.001 0.8934057
## 6  6    40 0.010 0.8921747
## 7  7    25 0.001 0.8919025
## 8  8    35 0.001 0.8914928
## 9  9    15 0.010 0.8905699
## 10 10   20 0.001 0.8899210
## 11 11   40 0.001 0.8892375
## 12 12   25 0.100 0.8862640
## 13 13   35 0.100 0.8855127
## 14 14   30 0.100 0.8847618
## 15 15   20 0.100 0.8841129
## 16 16   40 0.100 0.8837363
## 17 17   15 0.100 0.8829166
## 18 18   15 0.001 0.8794649
## 19 19   10 0.010 0.8793648
## 20 20   10 0.100 0.8653218
## 21 21   10 0.001 0.8557885
## 22 22    5 0.010 0.8237056
## 23 23    5 0.100 0.8129454
## 24 24    5 0.001 0.8113744

```

Analizando la tabla anterior, empleando un *decay* o *learning_rate* de 0.01, el modelo obtiene muy buenos resultados, mejores incluso que la regresión logística, un indicativo de la no linealidad de las variables. Por otro lado, y en relación con el número de nodos ¿Merece aumentar hasta 20, 30 o incluso 40 nodos? En vista a los resultados, no lo parece. A modo de ejemplo, con tan solo 15 nodos y un *decay* de 0.01, el valor de *accuracy* alcanza un 89 %, mientras que con 20, 30 o 40 nodos, la diferencia es de tan solo unas milésimas (no estamos ganando lo suficiente como para decantarnos por un modelo más complejo). Por tanto, una buena alternativa sería emplear 15 nodos (10 tampoco sería una mala opción, aunque comienza a decaer hasta el 87 %).

No obstante, con un *decay* de 0.01, analicemos tanto el sesgo como la varianza en base al número de nodos:

Como primera impresión, y en base al mejor AUC, un modelo con 20, 25 o 30 nodos sería una buena alternativa, de no ser por un aspecto clave: la escala de los ejes. En relación con la tasa de fallos, la diferencia de error entre un modelo con 15 nodos y un modelo con 20 es muy pequeña (de 0.11 a 0.105), además de que la varianza del modelo con 15 nodos (elevada desde un punto de vista gráfico), la escala del eje puede llevar a engaño, dado que sus valores de error oscilan entre 0.1075 y 0.11. Por otro lado, la diferencia del AUC entre un modelo con 15 nodos y un modelo con 20 es muy pequeña (en torno 0.8925-0.895 en el caso de 15 nodos y 0.895-0.90 en el caso de 20 nodos).

De hecho, si aumentamos el número de repeticiones a 10:

Comprobamos que tanto el orden de los diferentes modelos como su varianza se mantienen prácticamente constante. En conclusión, con el primer set de variables nos decantamos por un modelo de red con 15 nodos, dado que la ganancia que supone al aumentar el número de nodos, tanto en AUC como en tasa de fallos, es muy pequeña.

A continuación, y aumentando a 10 el número de repeticiones, tuneamos el número de iteraciones:

Analizando los resultados, bien es cierto que conforme aumenta el número de iteraciones, tanto la tasa de fallos como el AUC comienzan a estabilizarse. No obstante, la ganancia que supone aumentar el número de iteraciones es muy pequeña. A modo de ejemplo, de 200 a 300 iteraciones, la tasa de fallos

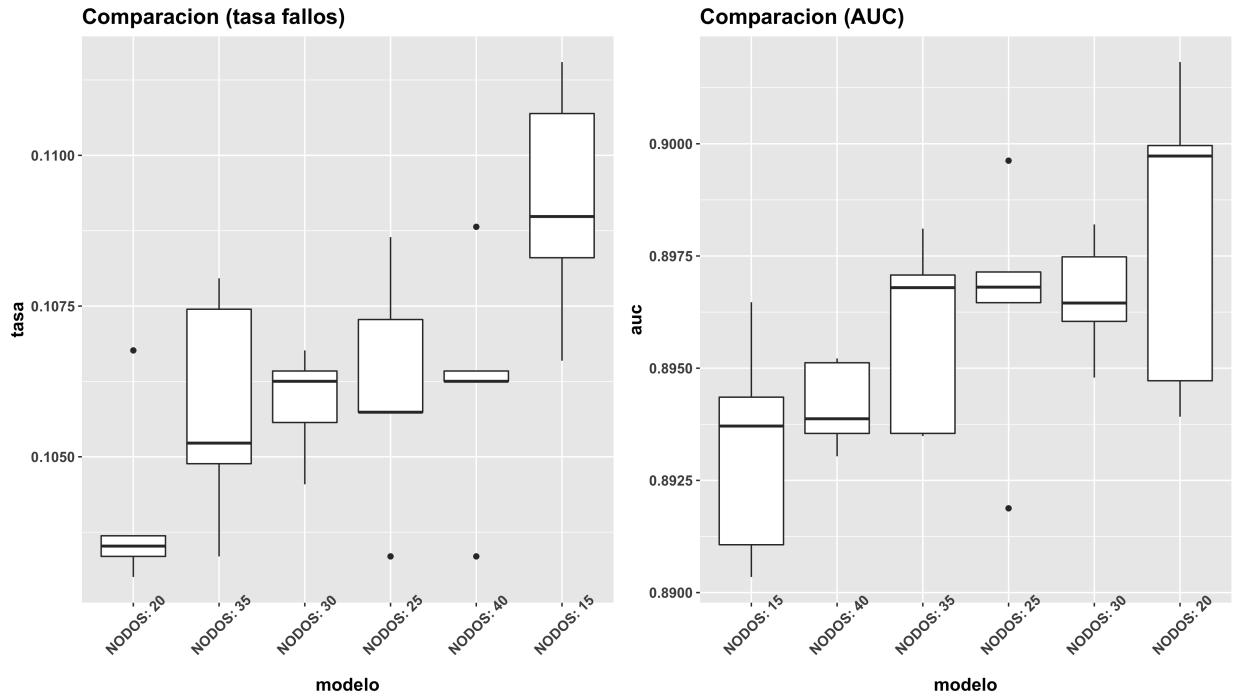


Figure 11: Comparacion avnnet modelo 1

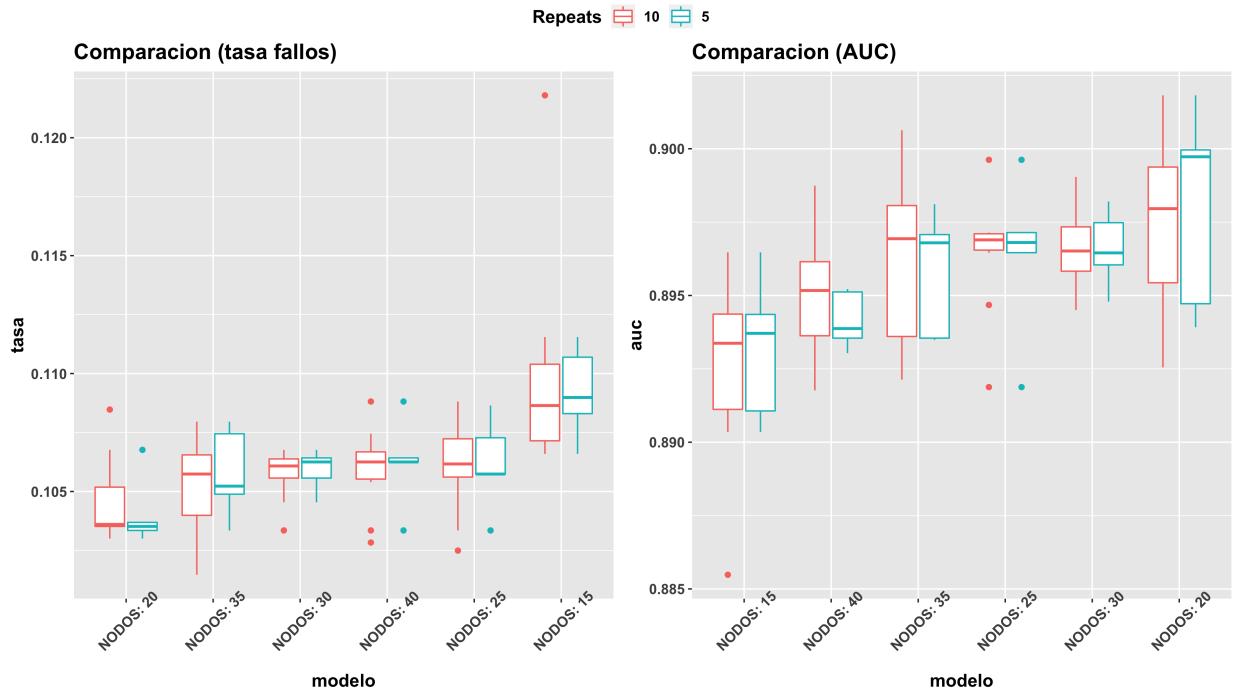


Figure 12: Comparacion avnnet modelo 1 (10 rep.)

mejora de 0.11 a 0.105, aproximadamente, mientras que el valor AUC apenas se ve afectado. Por tanto, mantenemos el número de iteraciones a 200.

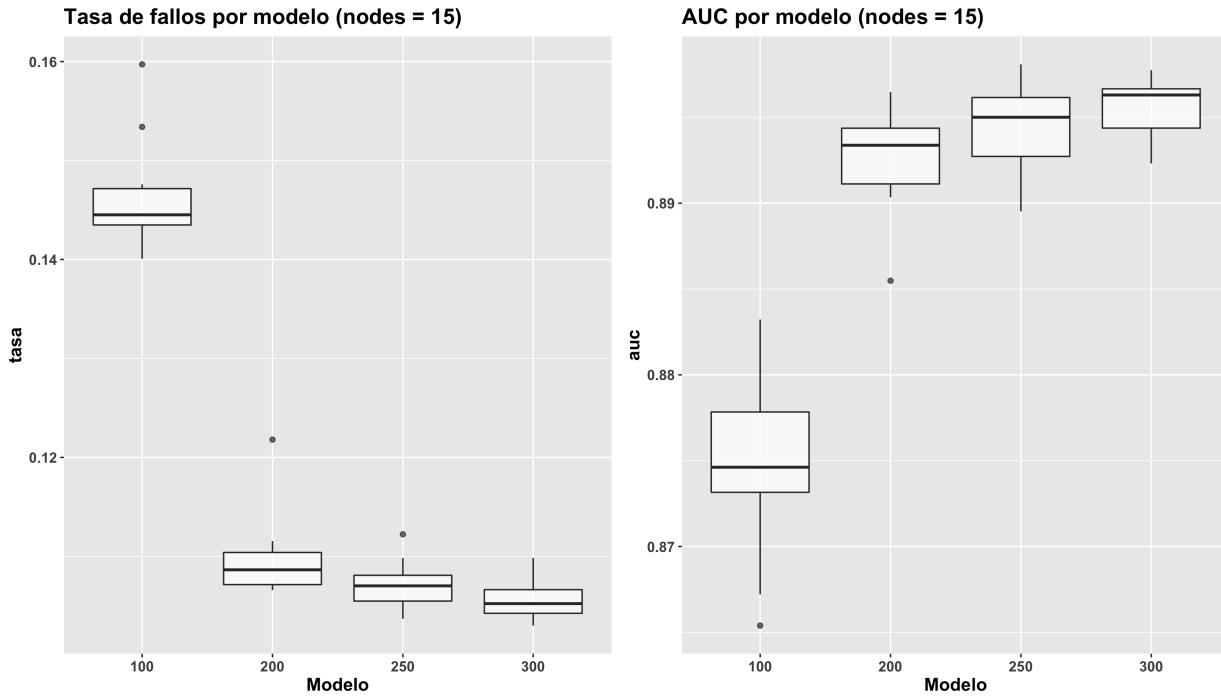


Figure 13: Comparacion avnnet modelo 1 (iteraciones)

6.2 Modelo 2 (AIC-BIC TOP 4)

A continuación, realizamos los mismos pasos con el segundo *set* de variables candidato. En primer lugar, y dado que disponemos de 4 variables *input*, para obtener 20 o 30 observaciones por parámetros necesitamos:

1. Con 20 observaciones por parámetro:

Con $k = 4$ variables input, entonces: $6 * h + 1 = 292$. Es decir, 48 – 49 nodos

2. Con 30 observaciones por parámetro:

Con $k = 4$ variables input, entonces: $6 * h + 1 = 195$. Es decir, 27 – 28 nodos

Con un primer cálculo, necesitaríamos un elevado número de nodos para obtener, al menos, 20 o 30 observaciones por parámetro. Sin embargo, si nos fijamos en lo empírico:

```
size.candidato.2 <- c(5, 10, 15, 20, 25, 30, 35, 40, 45)
decay.candidato.2 <- c(0.1, 0.01, 0.001)

cvnnet.candidato.1 <- cruzadaavnnetbin(data=surgical_dataset, vardep=target,
                                         listconti=var_modelo2, listclass=c(""),
                                         grupos=5, sinicio=1234, repe=5, size=size.candidato.1,
                                         decay=decay.candidato.1, repeticiones=5, itera=200)

##      id size decay Accuracy
## 1    1   20 0.001 0.9026305
## 2    2   35 0.001 0.9021868
## 3    3   25 0.001 0.9020152
## 4    4   30 0.001 0.9017081
## 5    5   25 0.010 0.9007853
## 6    6   30 0.010 0.8998632
```

```

## 7    7    15 0.001 0.8997609
## 8    8    45 0.001 0.8995550
## 9    9    20 0.010 0.8994869
## 10   10   45 0.010 0.8994521
## 11   11   40 0.001 0.8994190
## 12   12   15 0.010 0.8992479
## 13   13   35 0.010 0.8989061
## 14   14   40 0.010 0.8986332
## 15   15   10 0.010 0.8946358
## 16   16   10 0.001 0.8906726
## 17   17   25 0.100 0.8897164
## 18   18   40 0.100 0.8896133
## 19   19   20 0.100 0.8895444
## 20   20   35 0.100 0.8894082
## 21   21   30 0.100 0.8888957
## 22   22   45 0.100 0.8888273
## 23   23   15 0.100 0.8884856
## 24   24   10 0.100 0.8855818
## 25   25    5 0.010 0.8447890
## 26   26    5 0.001 0.8358688
## 27   27    5 0.100 0.8303017

```

En un primer resultado, **con un parámetro de regularización pequeño obtenemos buenos resultados**. A modo de ejemplo, llama la atención modelos de red con 20, 25 o 30 nodos que alcanzan un *Accuracy* de 0.90. Sin embargo, conforme descendemos en la tabla (y con ello, el número de nodos) comprobamos que la diferencia no es muy significativa: con tan solo 15 o 10 nodos y un *decay* de 0.01, el valor de *Accuracy* tan solo empeora en 0.89. Es decir, a simple vista el hecho de aumentar el número de nodos **no compensa la ganancia de Accuracy**.

No obstante, analicemos tanto el sesgo como la varianza con los mejores modelos:

```

# Probamos con 15-20-25-30 nodos y decay 0.001; 10-15 nodos y decay 0.01
size.candidato.2 <- c(10, 15, 15, 20, 25, 30)
decay.candidato.2 <- c(0.01, 0.01, 0.001, 0.001, 0.001, 0.001)

```

En un primer resultado, **la ganancia de error que supone reducir el número de nodos a 10 (y con un decay de 0.01), es muy pequeña: de 0.09 con 25 nodos a 0.105 con tan solo 10 nodos**. Del mismo modo sucede con el área bajo la curva, donde la mejora con 15 o 30 nodos es de tan solo unas milésimas de diferencia, lo cual puede depender de condiciones azarosas como la semilla de aleatoriedad. No obstante, con 10, 15 y 20 nodos, echemos un vistazo al modelo con 10 repeticiones:

Incluso aumentando a 10 repeticiones, el orden de los modelos no cambia. Por tanto, dada su simplicidad y la poca ganancia de error que supone, **elegimos el modelo con 10 nodos y decay 0.01**.

Por último, tuneamos el número de iteraciones:

Del mismo modo que sucedía con el primer *set* de variables, **aumentar el número de iteraciones no supone una mejoría significativa**, por lo que lo mantenemos a 200.

6.3 Comparación final

Una vez obtenidos ambos modelos, con los datos *test* realizamos las primeras predicciones:

```

##           modelo precision sensibilidad especificidad valor_pred_pos
## 1 Modelo 1 (BIC TOP 5)    0.8992        0.9471        0.8897      0.6305
## 2     Modelo 2 (top 4)    0.9032        0.9639        0.8913      0.6351
##   valor_pred_neg
## 1          0.9883

```

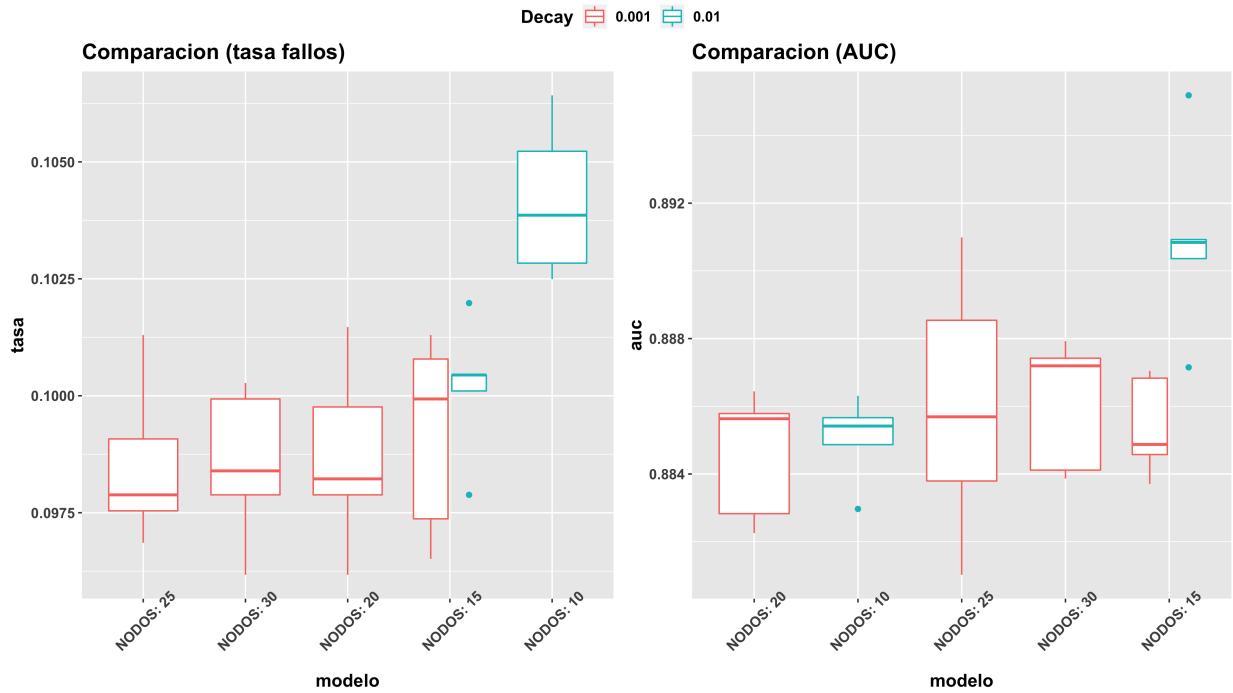


Figure 14: Comparacion avnnet modelo 2

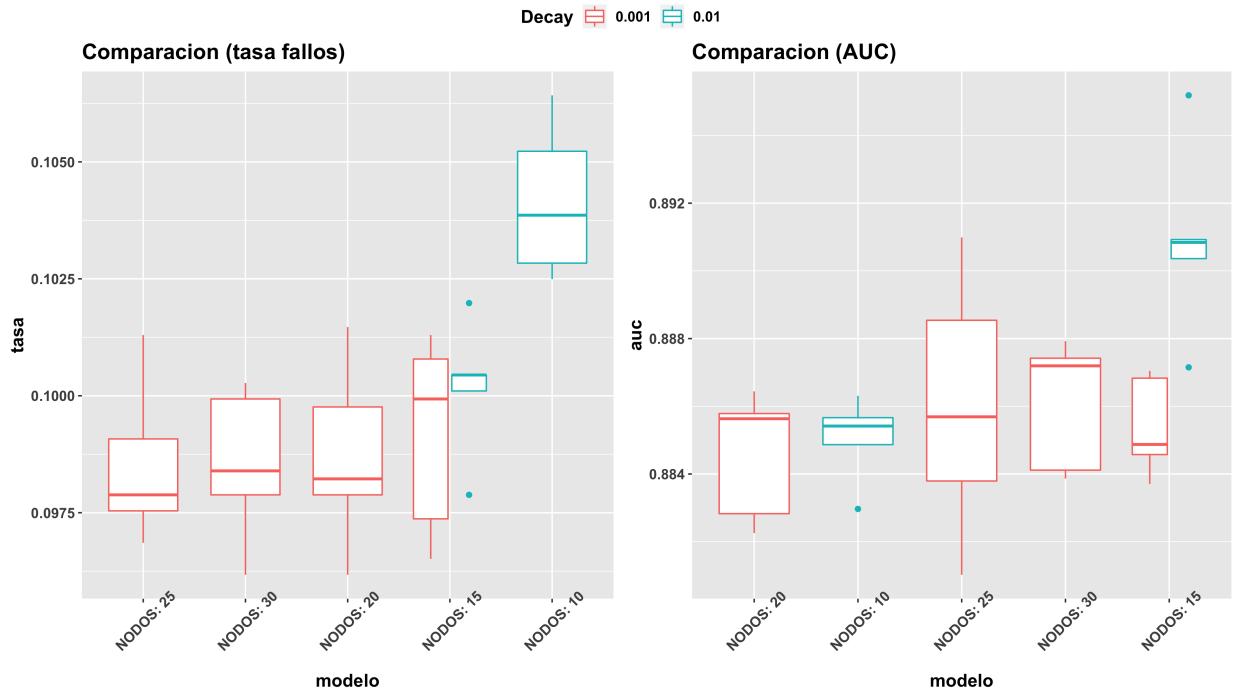


Figure 15: Comparacion avnnet modelo 2 (10 rep.)

```
## 2          0.9921
```

En primera instancia, **con un modelo de red sencillo conseguimos mejorar prácticamente en todos los aspectos**. No obstante, cabe destacar el valor predictivo positivo, que pese a su mejoría (del 20-30 al 63

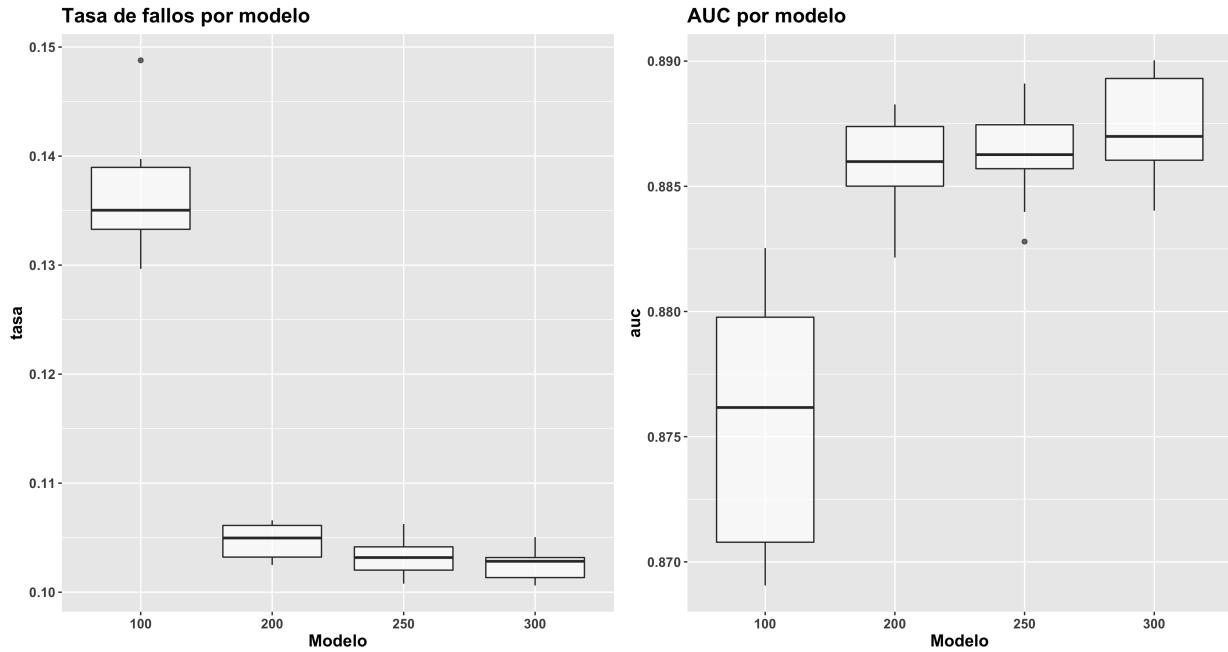


Figure 16: Comparacion avnnet modelo 2 (iteraciones)

%), continua existiendo un alto porcentaje de falsos positivos.

En conclusión, a la vista de los resultados obtenidos, tanto en tasa de fallos como en AUC, **los modelos de red mejoran significativamente los resultados del modelo**, un claro indicio de la **no linealidad con la variable objetivo**. En relación con ambos *set* de variables, el hecho de añadir una variable *input* adicional en el primer modelo **no hace mejorar significativamente sus resultados**:

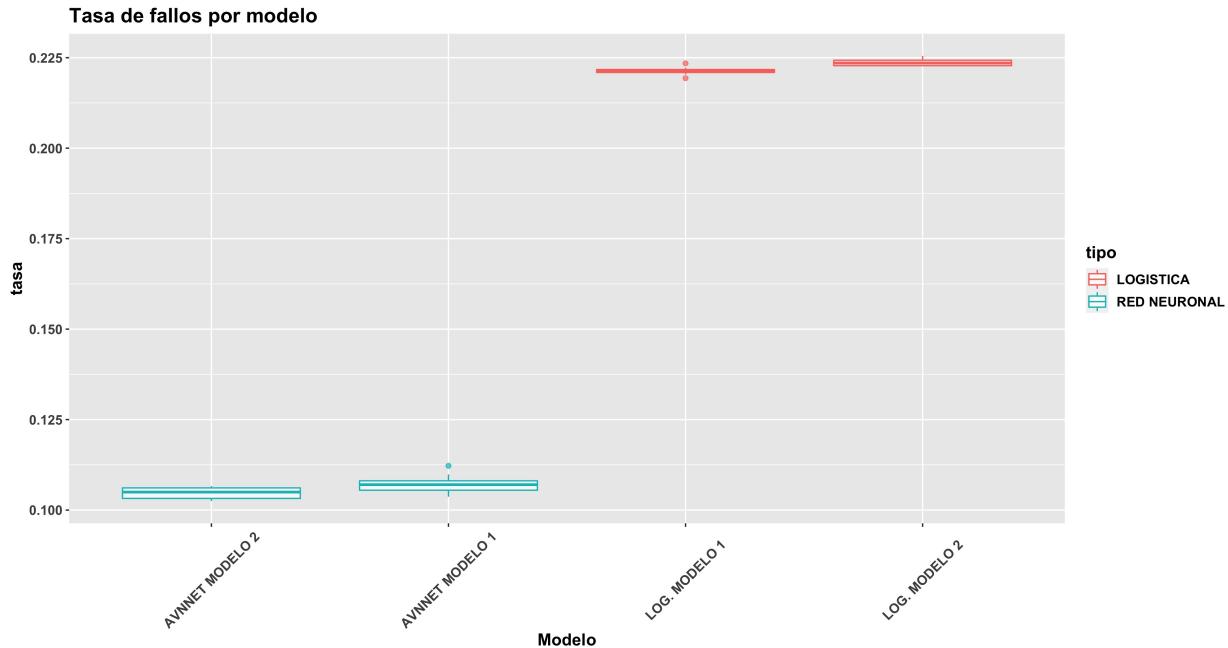


Figure 17: Comparacion tasa fallos log-avnnet

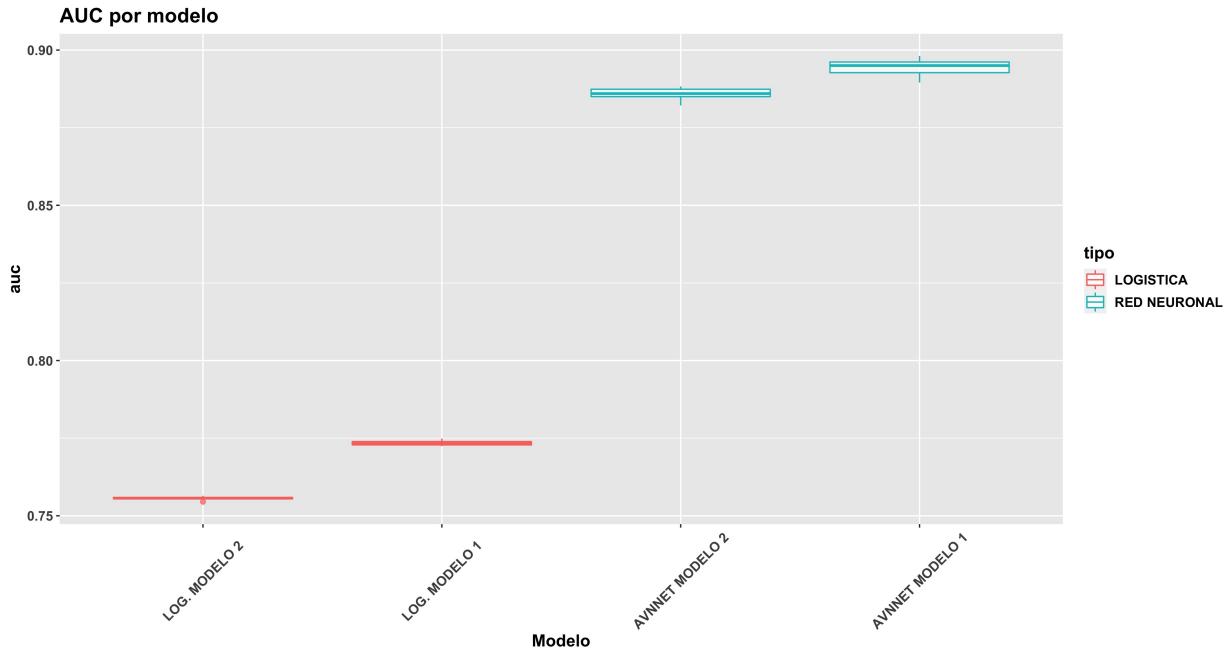


Figure 18: Comparacion AUC log-avnnet

RESUMEN red neuronal:

1. modelo 1: $size = 15$ y $decay = 0.01$.
2. modelo 2: $size = 10$ y $decay = 0.01$.

7. Bagging

7.1 Selección del número de árboles

Continuando con el modelo *bagging*, de forma previa al tuneo de hipéparámetros, debemos fijar un número de árboles (ntrees) para ambos *sets* de variables, un valor mínimo a partir del cual el error OOB (*Out of bag error*) se estabiliza:

```
##-- Seleccion del numero de arboles
set.seed(1234)
##-- Modelo 1
rbfbis.1<-randomForest(factor(target)~mortality_rsi+ccsMort30Rate+bmi+month.8+Age,
                         data=surgical_dataset,
                         mtry=mtry.1,ntree=5000,nodesize=10,replace=TRUE)

##-- Modelo 2
rbfbis.2<-randomForest(factor(target)~Age+mortality_rsi+bmi+month.8,
                         data=surgical_dataset,
                         mtry=mtry.2,ntree=5000,nodesize=10,replace=TRUE)
```

Analizando el error *Out of bag*, en ambos modelos el error se estabiliza con aproximadamente 1000 árboles, obteniendo valores de error inferiores a 0.1, similares a los obtenidos en los modelos de red. Si hacemos *zoom* sobre el gráfico:

Observamos como a partir de 200-250 árboles, el error comienza a situarse por debajo de 0.10. Sin embargo, no es hasta los 900-1000 árboles cuando el error prácticamente se estabiliza, a partir del cual

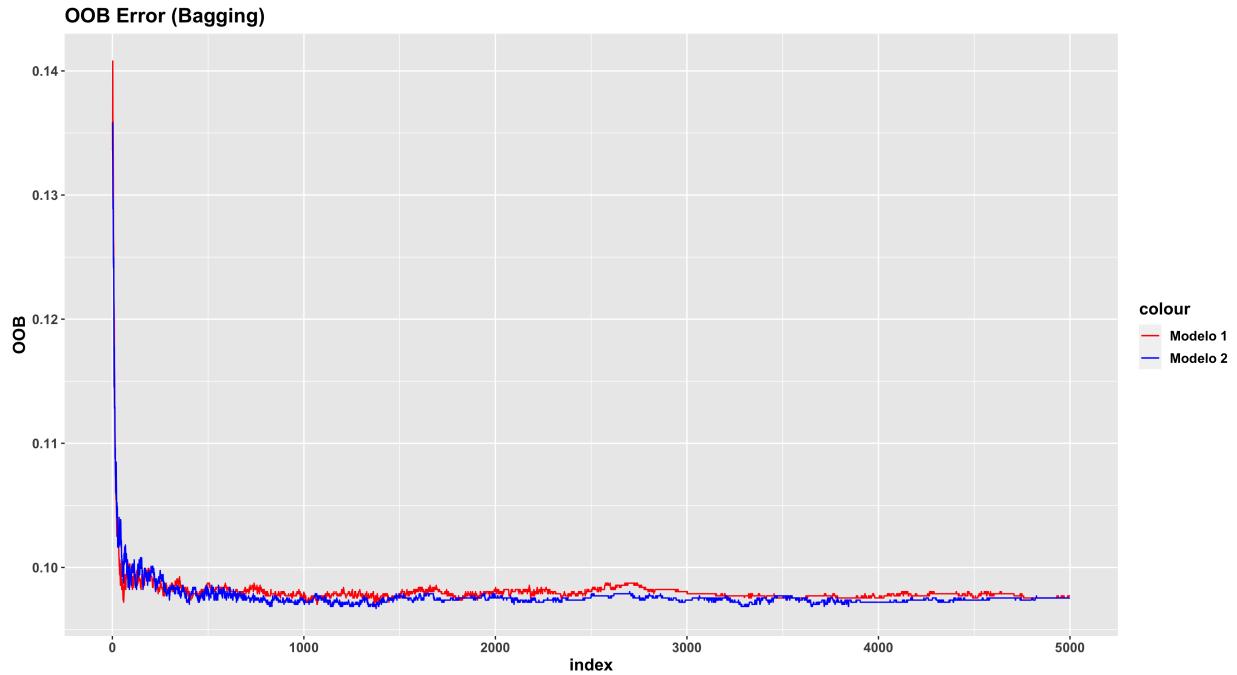


Figure 19: OOB Error (Modelos 1 y 2) (I)

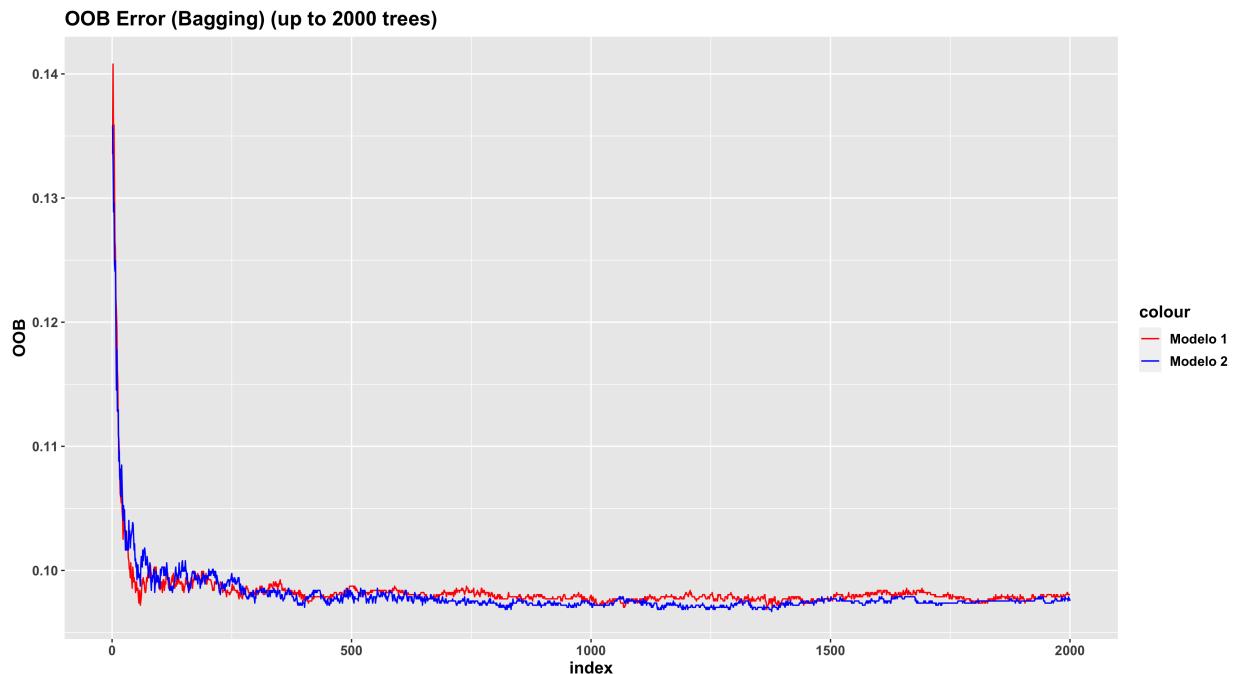


Figure 20: OOB Error (Modelos 1 y 2) (II)

se detectan ciertas fluctuaciones (subidas o bajadas en el error), aunque de forma aleatoria. Por tanto, para ambos modelos escogemos 900 como número de árboles.

```
n.trees.1 <- 900; n.trees.2 <- 900
```

7.2 Modelo 1

En un primer comienzo, hagamos un repaso previo de los parámetros a tunear en un modelo *bagging*, junto con el número de árboles:

mtry: número de variables sorteadas aleatoriamente en cada división del árbol. Dado que se trata de un modelo *bagging*, **establecemos en dicho parámetro el número total de variables independientes del modelo.**

nodesize: tamaño máximo de nodos finales.

sampsizes: número de observaciones seleccionadas aleatoriamente (con o sin reemplazamiento) para la construcción del árbol.

replace: si el sorteo anterior se realiza con o sin reemplazamiento (por defecto, con reemplazamiento).

Dado que *mtry* debe corresponder con el número de variables *input* y el número de árboles ya está definido, quedan por tunear tanto *nodesize* como *sampsizes* y *replace*. Por tanto, **comenzamos ajustando tanto el tamaño de la submuestra como el tamaño máximo de nodos finales** (en el caso del parámetro *replace*, una vez obtenidos los modelos *bagging* para ambos *sets*, con el mejor de ambos probaremos a seleccionar muestras sin reemplazamiento).

Antes de tunear *sampsizes*, y dado que estamos trabajando con validación cruzada, **dispondremos de menos observaciones para construir el modelo, de forma que debemos establecer un tamaño máximo sobre dicho parámetro.** Concretamente, dado que disponemos de 5854 observaciones y 5 grupos de validación cruzada, cada grupo tendrá disponible $5854 * (4/5) \sim 4683$ observaciones, siendo el tamaño máximo de muestra que podemos probar:

```
mtry.1      <- 5
## Redondeamos 4683 a 4600
sampsizes.1 <- list(1, 100, 500, 1000, 2000, 3000, 4600)
nodesizes.1 <- list(5, 10, 20, 30, 40, 50, 100)

bagging_modelo1 <- tuneo_bagging(surgical_dataset, target = target,
                                    lista_continua = var_modelo1,
                                    nodesizes = nodesizes.1,
                                    sampsizes = sampsizes.1, mtry = mtry.1,
                                    ntree = n.trees.1, grupos = 5, repe = 5)
```

A continuación, por cada combinación *nodesize - sampsizes* mostramos el promedio tanto de la tasa de fallos como de AUC:

En un primer análisis, caben destacar fundamentalmente dos aspectos:

1. En relación al *nodesize*, tanto en tasa de fallos como en AUC, se obtienen buenos resultados en torno a un valor de 5-30, aproximadamente, pues a partir de 50-100 (árboles menos complejos), la precisión del modelo comienza a disminuir. Sin embargo, ¿Merece la pena aumentar la complejidad del árbol, disminuyendo el parámetro *nodesize*? A simple vista, la diferencia entre un modelo de mayor complejidad (*nodesize = 5*), y un modelo de menor complejidad (*nodesize = 20*), es muy pequeña (punto rojo y verde). **Por tanto, una posibilidad sería decantarse por un *nodesize* elevado, en torno a 20.**
2. Por otro lado, llama la atención el tuneo sobre el parámetro *sampsizes*. Bien es cierto que conforme aumenta el tamaño de la submuestra, mayor es la precisión. No obstante, ¿Qué diferencia existe entre un modelo en el que se utilizan todas las observaciones (*sampsizes = 1*) y un modelo con tan solo 500 o 1000 submuestras? No hay demasiada diferencia, en especial con un *nodesize* en torno a 20, tal y como comentamos en el apartado anterior. De este modo, y gracias además al elevado número de árboles del que disponemos (900), **un valor *sampsizes* bajo permite no solo reducir el tiempo de cómputo para entrenar el modelo, sino además una mayor variedad en la construcción de cada uno de los árboles.**

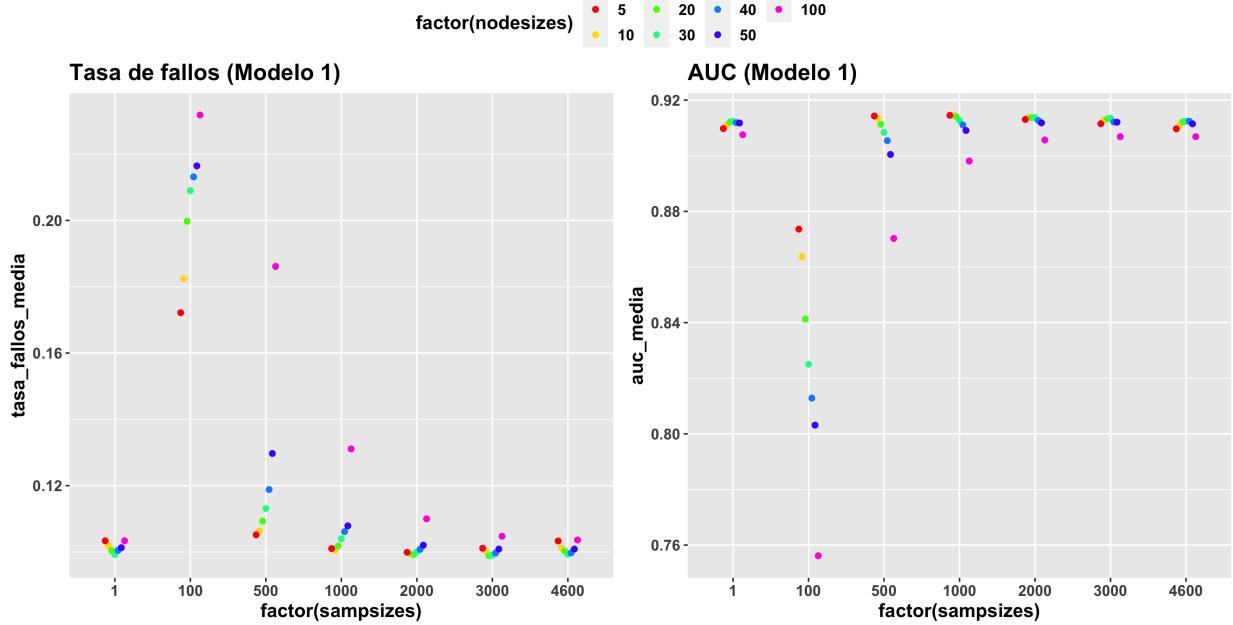


Figure 21: Distribucion de la tasa de fallos y AUC por sampsizes y nodesize (Modelo 1) (I)

Es decir, para obtener un buen modelo *bagging* **no es necesario emplear todas las observaciones, ni tampoco árboles con demasiada complejidad**, sino que con pocas muestras (en torno a 500), y un *nodesize* alto (en torno a 20), se obtienen muy buenos resultados. Por tanto, una vez realizado el primer análisis, realizamos una comparación (tanto del sesgo como de la varianza) sobre modelos *bagging* con un valor *nodesize* pequeño (5), lo que se traduce en árboles más complejos, frente a modelos con un valor alto (en torno a 20). Del mismo modo, analizamos el sesgo y varianza con diferentes valores de *sampsizes*, aumentando a 10 el número de repeticiones para observar mejor su efecto:

```
nodesizes.1 <- list(5, 20)
##-- Probamos con un sampsizes entre 500 y 2000 observaciones (1 = todas las observaciones)
sampsizes.1 <- list(1, 500, 1000, 1500, 2000)
bagging_modelo1_v2 <- tuneo_bagging(surgical_dataset, target = target,
                                         lista_continua = var_modelo1,
                                         nodesizes = nodesizes.1,
                                         sampsizes = sampsizes.1, mtry = mtry.1,
                                         ntree = n.trees.1, grupos = 5, repe = 10,
                                         show_nrnodes = "yes")
```

Nuevamente, sin fijarnos en la escala del eje, una posible opción a decantarse sería elegir un modelo con *nodesize* 5 y *sampsizes* 1000, pues presenta el mayor valor AUC y una tasa de fallos baja. No obstante, los valores del eje pueden llevarnos a engaño. Por ejemplo, la diferencia entre este modelo y uno mucho más sencillo como es el caso de *nodesize* 20 y *sampsizes* en torno a 500-1000 (20 + 500 o 20 + 1000) es muy pequeña (hablamos de una tasa de fallos en torno a 0.10 y un AUC de 0.91, por lo que la diferencia es del orden de milésimas), incluso con un menor número de muestras la varianza de los modelos se ve reducida en comparación con usar todas las observaciones, aunque la diferencia (por las escalas del eje) sea menor.

Sin embargo, ¿Y si aumentamos *nodesize* a 50?

A simple vista, **los resultados obtenidos son muy similares en ambos casos**, especialmente en *AUC*, donde la diferencia entre el mejor modelo (20 + 1500) y el peor (50 + 500) es de tan solo 0.01. Sin embargo, y en relación con la tasa de fallos, para alcanzar un valor en torno al 10 % con un *nodesize* = 50, se requiere por lo general un mayor tamaño de submuestra, dado que la complejidad del árbol es menor. Sin embargo,

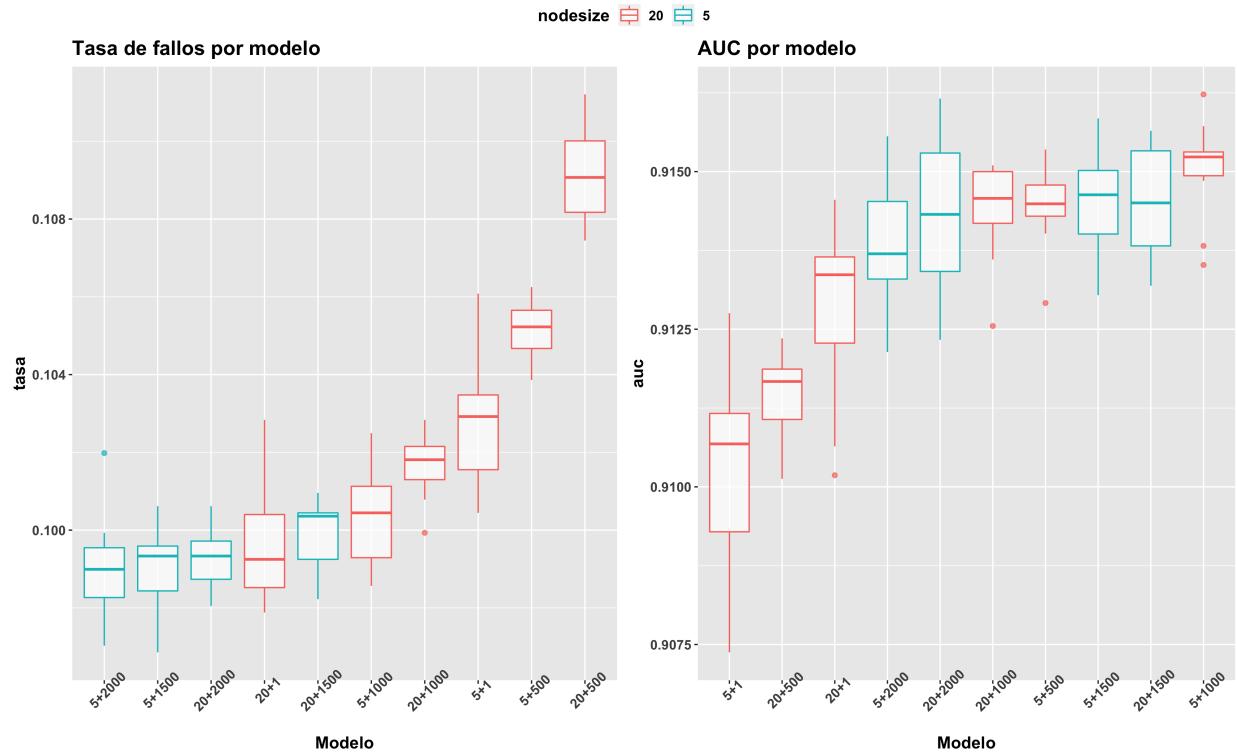


Figure 22: Distribucion de la tasa de fallos y AUC por sampszie y nodesize (Modelo 1) (II)

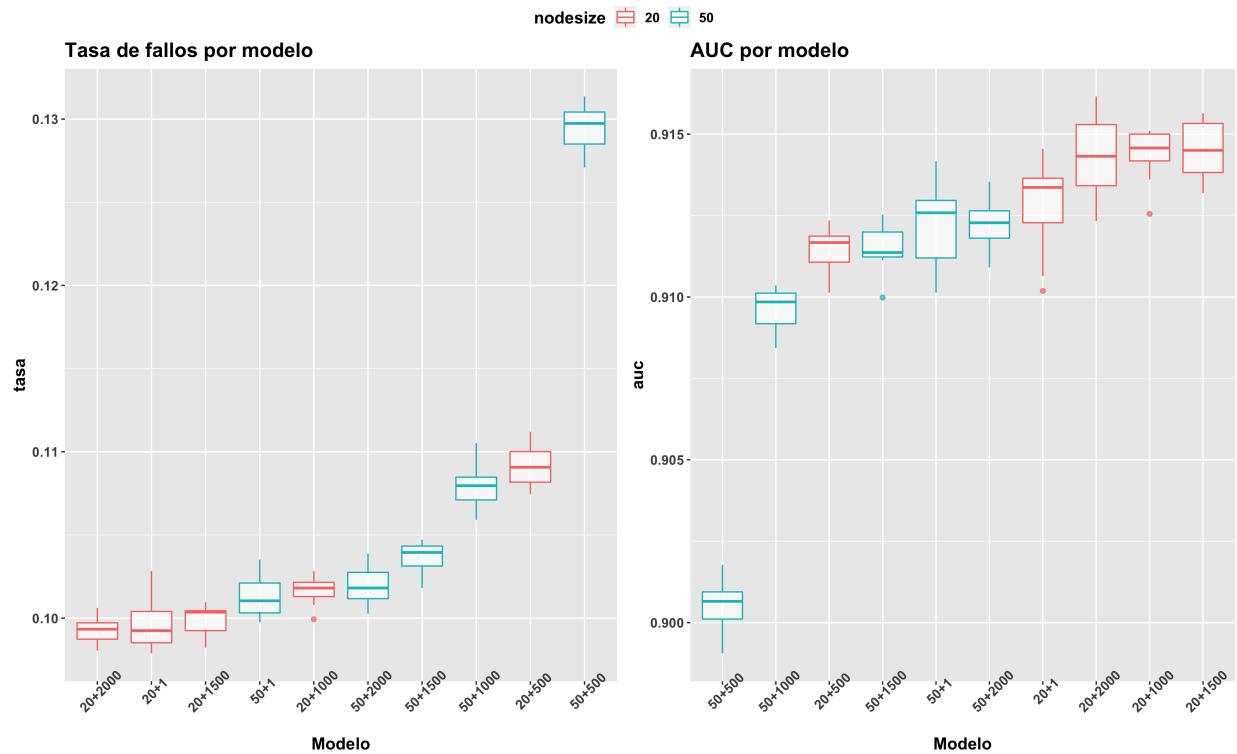


Figure 23: Distribucion de la tasa de fallos y AUC por sampszie y nodesize (Modelo 1) (III)

con un valor *nodesize* menor (20), se requieren menos observaciones.

Por tanto, dada la poca ganancia que presentan los modelos complejos, **nos decantamos por un modelo bagging con *nodesize* = 20 y un tamaño de muestra en torno a 500 y 1000**. Sobre ambos candidatos, para controlar mejor el tamaño óptimo de la muestra, probamos con una validación cruzada de 10 grupos y 20 repeticiones:

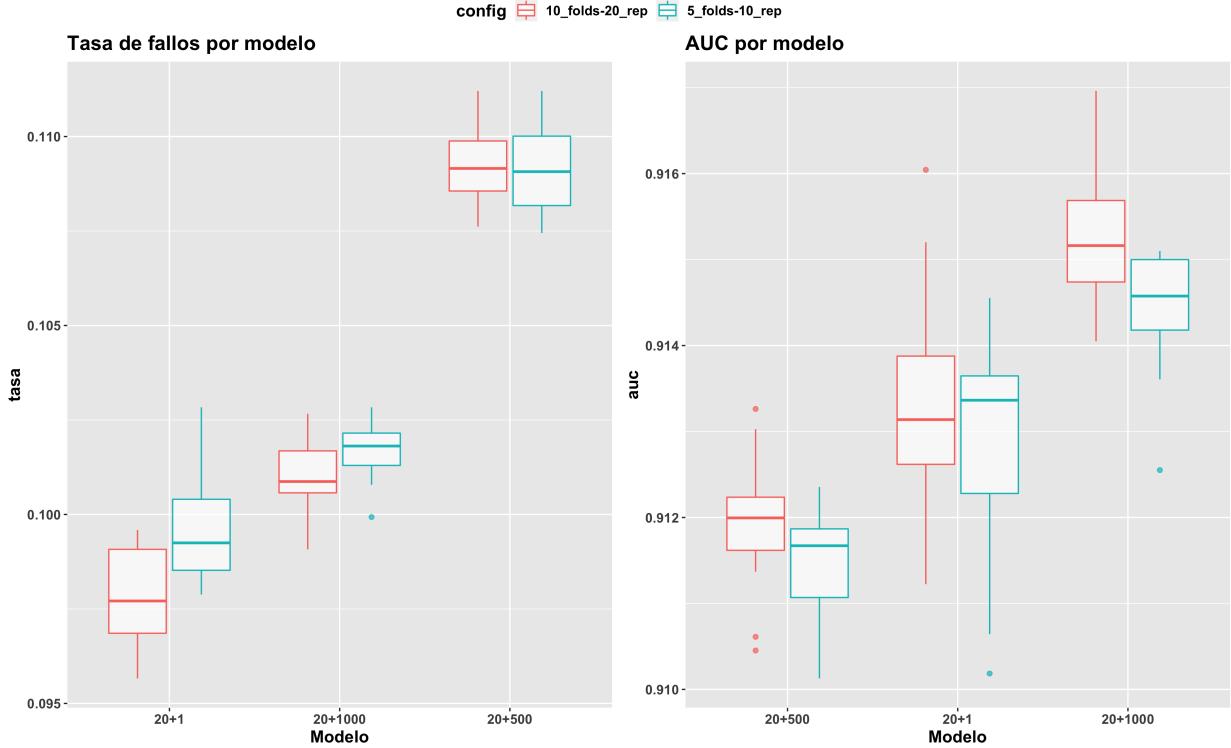


Figure 24: Distribucion de la tasa de fallos y AUC (Modelo 1) aumentando grupos y repeticiones

Por lo general, incluso aumentando el número de grupos y repeticiones, el orden de los modelos se mantiene idéntico, tanto en tasa de fallos como en AUC. Por tanto, de los dos posibles modelos candidatos (con *sampsizes* 500 o 1000), aunque la diferencia entre ambos, así como el sesgo y varianza no sean muy significativas (dada la escala de los ejes), **nos decantamos por un modelo con un tamaño de 1000 submuestras**, es decir, con tan solo el 1000 / 5854 ~ 17 % de las observaciones.

7.3 Modelo 2

Del mismo modo, realizamos los mismos pasos con el segundo *set* de cuatro variables, comenzando con el tuneo tanto de *nodesize* como de *sampsizes*, obteniendo el promedio en tasa de fallos y AUC con 5 repeticiones:

```
mtry.2      <- 4
sampsizes.2 <- list(1, 100, 500, 1000, 2000, 3000, 4600)
nodesizes.2 <- list(5, 10, 20, 30, 40, 50, 100)

bagging_modelo1 <- tuneo_bagging(surgical_dataset, target = target,
                                    lista_continua = var_modelo2,
                                    nodesizes = nodesizes.2,
                                    sampsizes = sampsizes.2, mtry = mtry.2,
                                    ntree = n.trees.2, grupos = 5, repe = 5)
```

Nuevamente, nos encontramos con un comportamiento similar al obtenido con el primer modelo: por un lado,

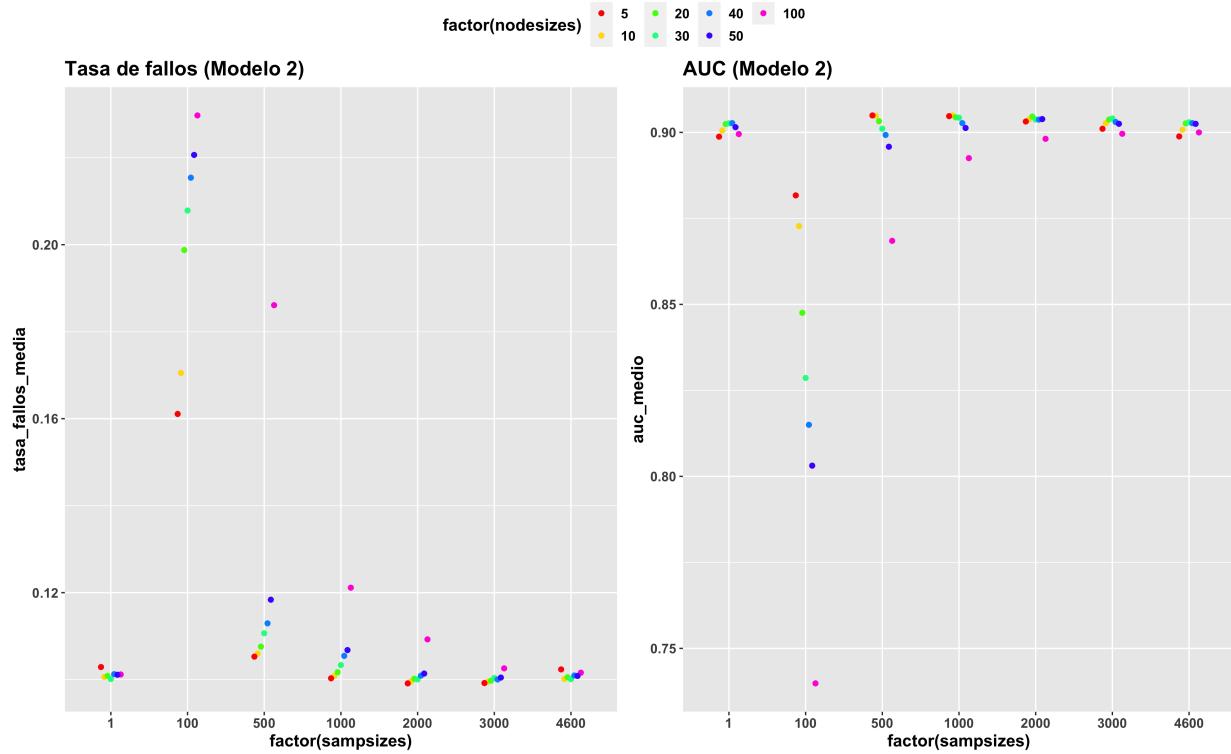


Figure 25: Distribucion de la tasa de fallos y AUC por sampsize y nodesize (Modelo 2) (I)

con un valor *nodesize* moderado/alto (en torno a 20), obtienen buenos resultados, por lo que no es necesario utilizar árboles demasiado complejos, pues la ganancia tanto en tasa de fallos como en AUC es ínfima. Por otro lado, y en relación con *sampsize*, **no es necesario emplear todas las observaciones**, sino que con tan solo 500 o 1000 muestras, el modelo obtiene buenos resultados.

Por tanto, y del mismo modo que en el primer *set* de variables, ___comparamos la diferencia entre un modelo de mayor complejidad (*nodesize* = 5) y de menor complejidad (*nodesize* = 20), utilizando diferentes tamaños de *sampsize*:

```
nodesizes.2 <- list(5, 20)
##-- Probamos con un sampsize entre 500 y 2000 observaciones (1 = todas las observaciones)
sampsizes.2 <- list(1, 500, 1000, 1500, 2000)
bagging_modelo1_v2 <- tuneo_bagging(surgical_dataset, target = target,
                                       lista_continua = var_modelo2,
                                       nodesizes = nodesizes.2,
                                       sampsizes = sampsizes.2, mtry = mtry.2,
                                       ntree = n.trees.2, grupos = 5, repe = 10,
                                       show_nrnodes = "yes")
```

De nuevo, el hecho de aumentar la complejidad de los árboles (reduciendo el valor de *nodesize*), no supone una mejoría relevante al modelo. A modo de ejemplo, la diferencia en la tasa de fallos entre un modelo con *nodesize* 5 (5 + 2000) y un modelo con *nodesize* 20 (20 + 2500), es de apenas unas milésimas de diferencia (del mismo modo sucede con AUC) ¿Y si aumentamos el valor a 50?

En base a la escala de los ejes, la diferencia no es muy significativa. Sin embargo y por lo general, con un *nodesize* = 50 se requiere un mayor número de observaciones, al ser árboles de menor complejidad, mientras que con un *nodesize* menor, en torno a 20, con tan solo 500 o 1000 observaciones se obtienen prácticamente los mismos resultados.

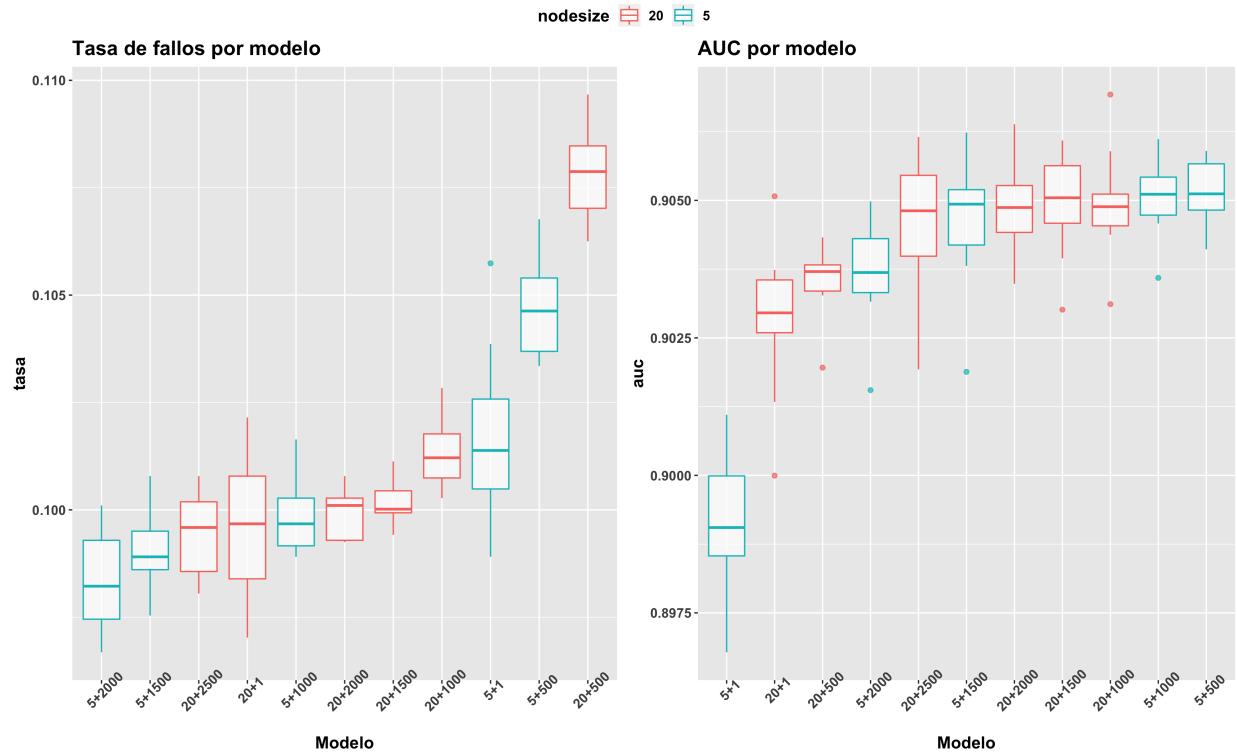


Figure 26: Distribucion de la tasa de fallos y AUC por sampszie y nodesize (Modelo 2) (II)

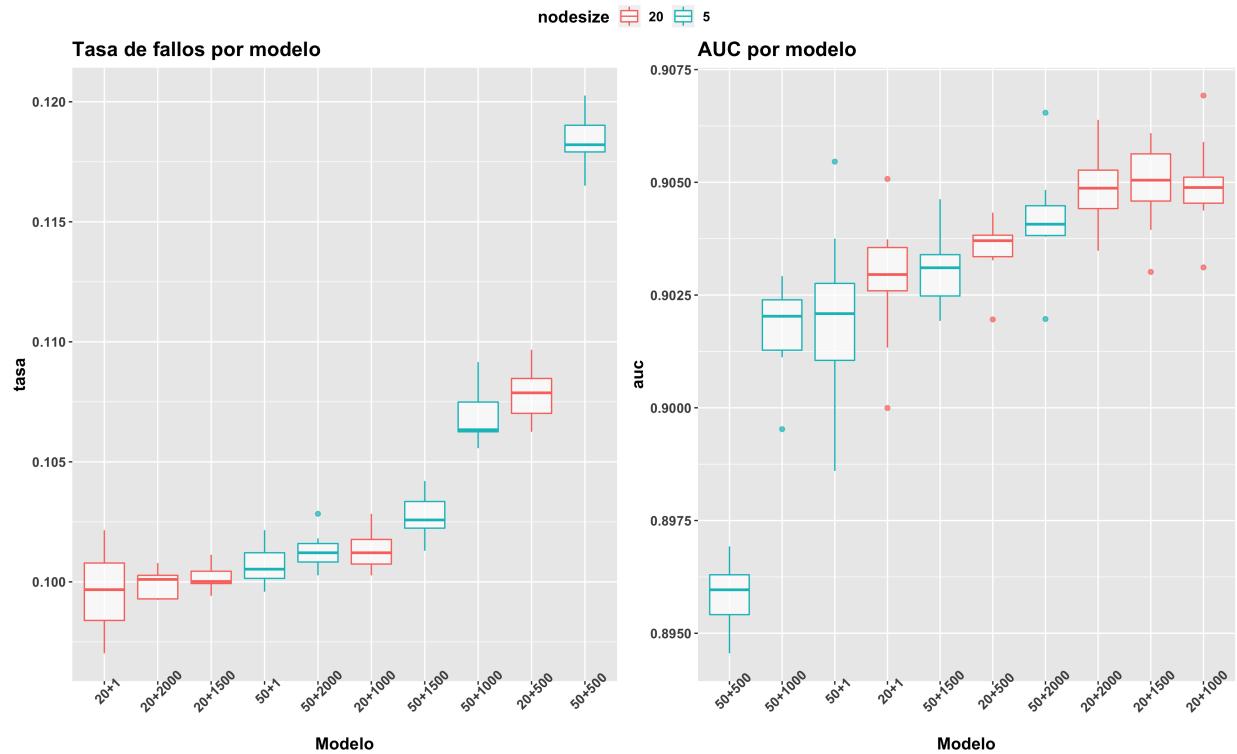


Figure 27: Distribucion de la tasa de fallos y AUC por sampszie y nodesize (Modelo 2) (III)

Por tanto, y del mismo modo que en el primer *set* de variables, nos decantamos por un *nodesize* de 20 y un tamaño de muestra en torno a 500-1000 observaciones. Sobre ambos candidatos, probamos de nuevo con una validación cruzada de 10 grupos y 20 repeticiones:

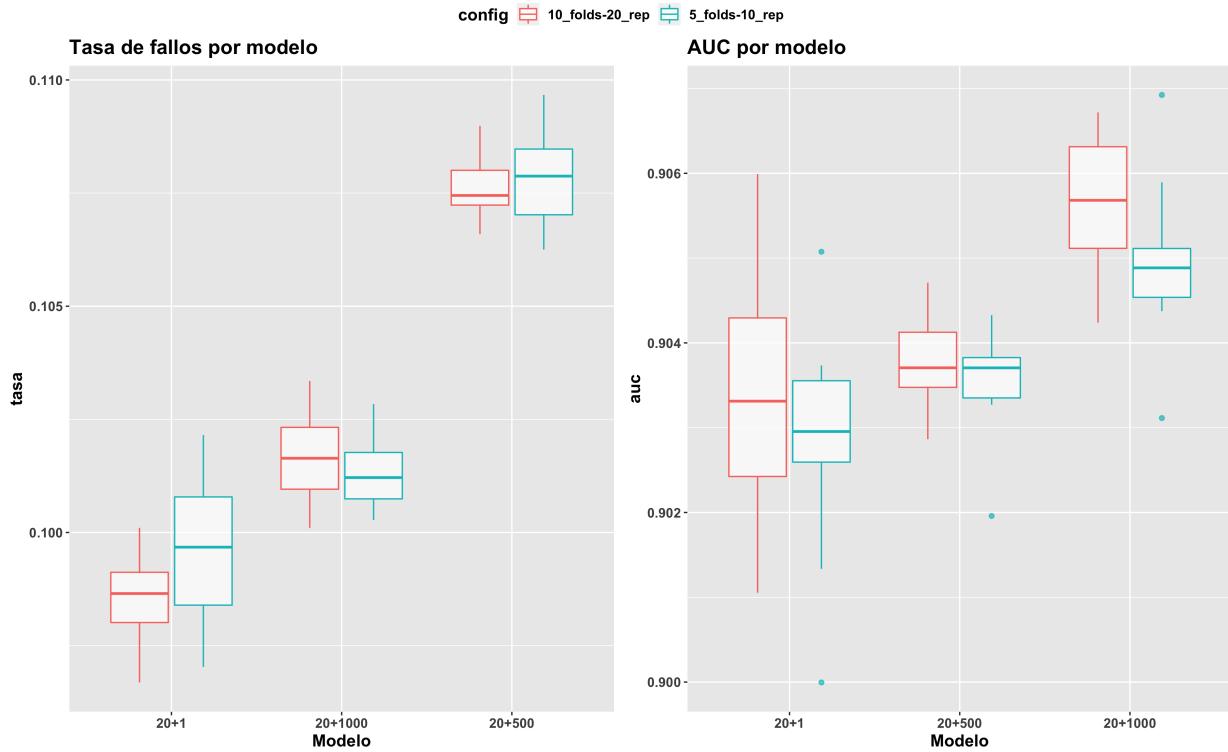


Figure 28: Distribucion de la tasa de fallos y AUC (Modelo 2) aumentando grupos y repeticiones

Incluso aumentando el número de grupos, el orden de los modelos se mantiene prácticamente igual. En conclusión, **nos decantamos por la misma configuración que con el primer set de variables: *nodesize* 20 y *sampszie* 1000.**

RESUMEN *bagging*:

1. modelo 1: *nodesize* = 20 y *sampszie* = 1000.
2. modelo 2: *nodesize* = 20 y *sampszie* = 1000.

7.4 Modelo sin reemplazamiento

Una vez tuneados ambos *sets* de variables, veamos la diferencia entre un modelo con *sampszie* con reemplazamiento y sin reemplazamiento:

```
## ¿Y si lo probamos sin reemplazamiento? Probamos con el mejor modelo en terminos de AUC (modelo 1)
bagging_modelo_sin_reemp <- tuneo_bagging(surgical_dataset, target = target,
                                             lista_continua = var_modelo1, nodesizes = 20,
                                             sampsizes = 1000, mtry = mtry.1, ntree = n.trees.1,
                                             grupos = 5, repe = 10, replace = FALSE)
```

A simple vista, no existe diferencia entre modelos con o sin reemplazamiento en *sampszie*. Por otro lado, y del mismo modo que sucedía con el modelo de red, no existe apenas diferencia entre el primer *set* con 5 variables y el segundo *set* con 4, tan solo de 0.01 en el caso del AUC.

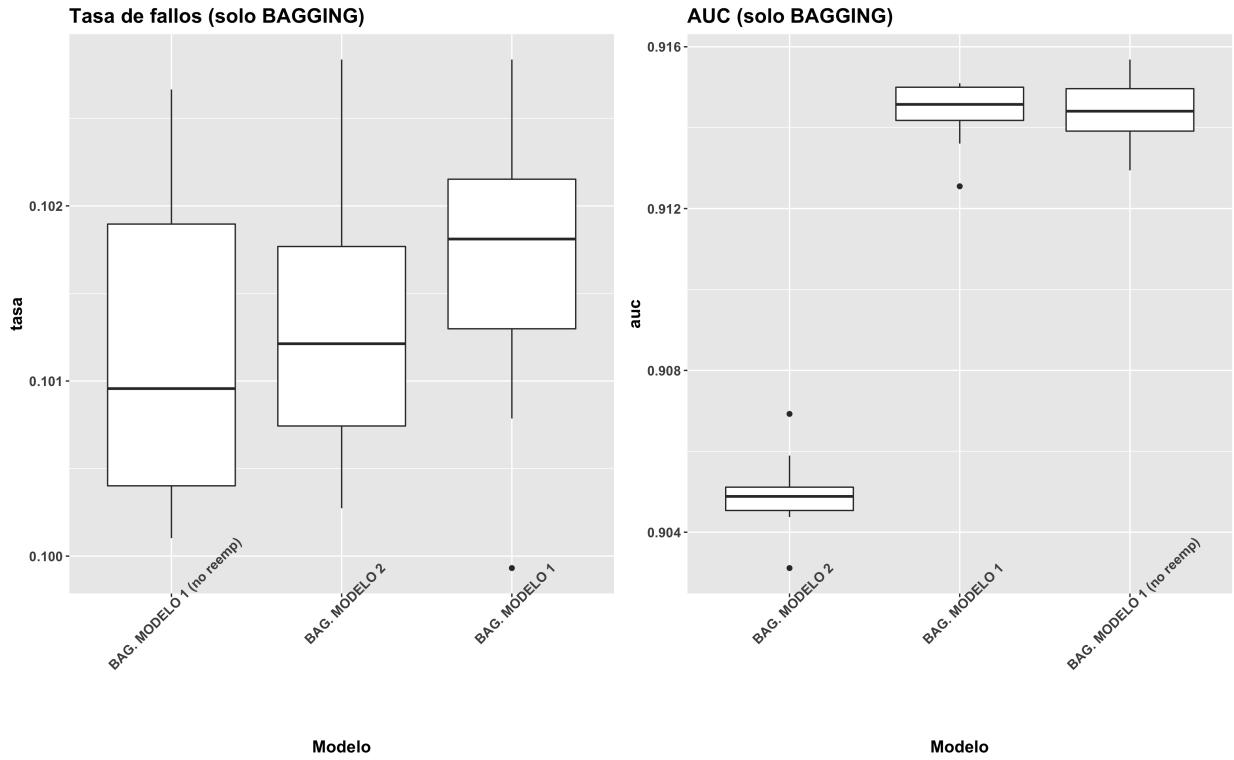


Figure 29: Distribucion de la tasa de fallos y AUC con y sin reemplazamiento

7.5 Comparación final

Finalmente, realizamos la comparación final de ambos modelos *bagging* con los datos *test*:

```
##           modelo precision sensibilidad especificidad valor_pred_pos
## 1 Modelo 1 (BIC TOP 5)    0.9010      0.9214      0.8966      0.6589
## 2     Modelo 2 (top 4)    0.9019      0.9468      0.8928      0.6424
##   valor_pred_neg
## 1          0.9813
## 2          0.9880
```

Por lo general, la estadísticas de las predicciones mejoran ligeramente, aunque se mantienen en la misma línea del modelo de red neuronal: altos porcentajes de sensibilidad, especificidad y valor predictivo negativo, aunque el valor predictivo positivo se sitúa en torno al 64-65 %, aproximadamente, lo cual es un indicativo de la presencia de un alto número de falsos positivos en las predicciones. Además, la diferencia entre el primer modelo *bagging* (con 5 variables *input*) y el segundo (con tan solo 4), no es muy relevante en base a las estadísticas anteriores.

Además, si observamos la distribución de la tasa de fallos y AUC en los modelos:

De nuevo, la relación no lineal entre las variables *input* se ve reflejado en los resultados obtenidos tanto en modelos de red como *bagging*, siendo este último caso el que obtiene mejores resultados tanto en tasa de fallos como en AUC, aunque la diferencia (dada la escala de los ejes) no sea muy relevante.

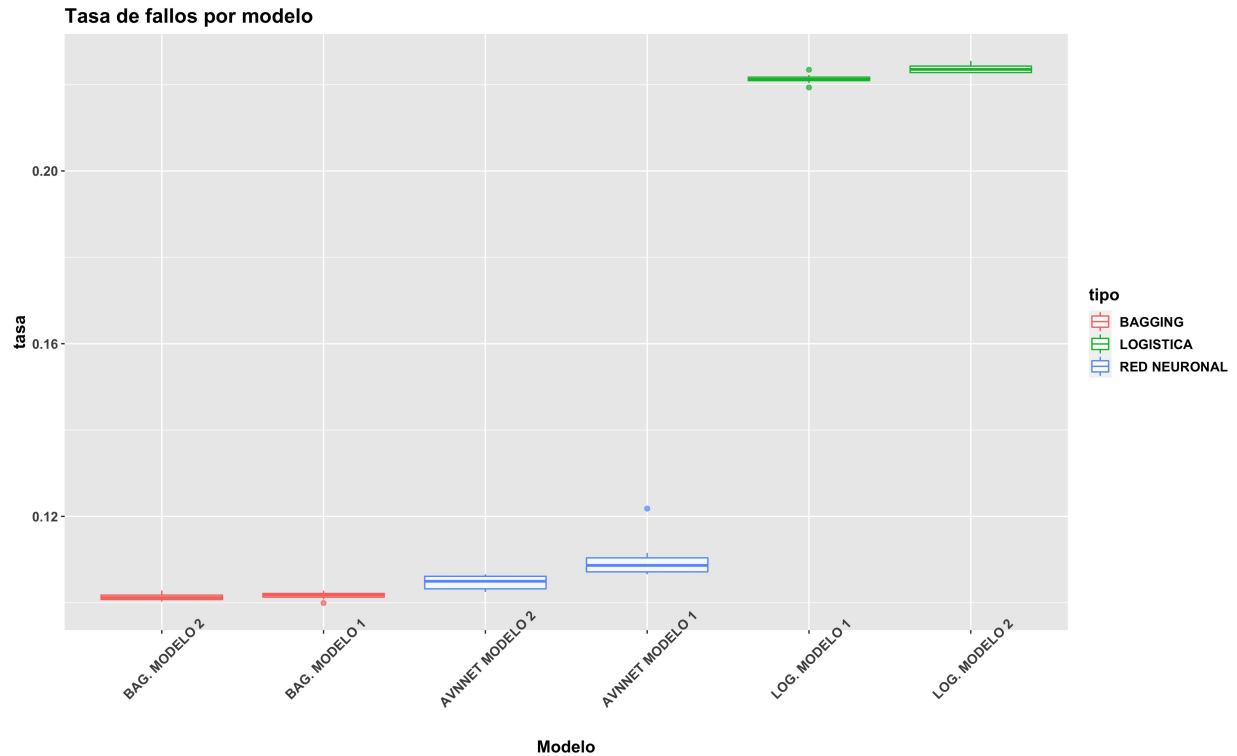


Figure 30: Comparacion tasa fallos log-avnnet-bagging

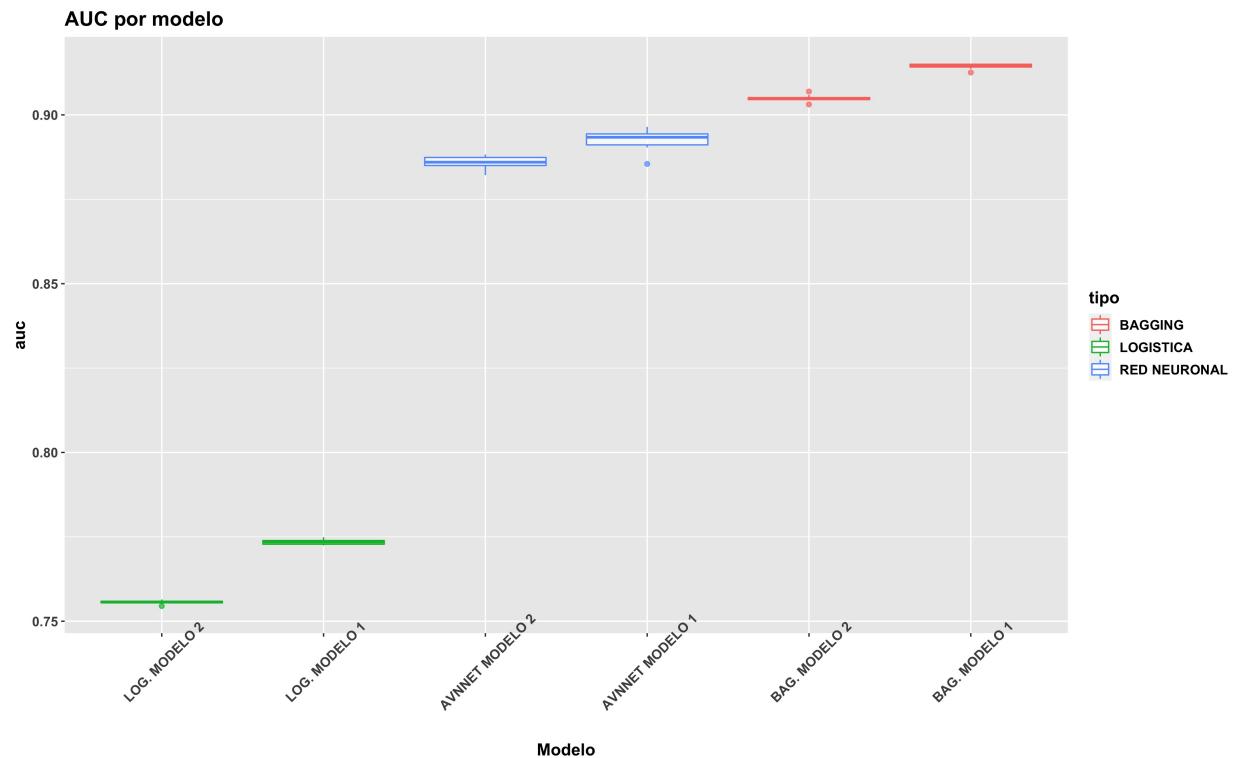


Figure 31: Comparacion AUC log-avnnet-bagging

8. Random Forest

8.1 Selección del número de árboles y mtry

Una vez elaborados los modelos *bagging*, y dado que *Random Forest* si realiza el sorteo de variables por cada división del árbol ¿Con 900 árboles es suficiente? Para comprobarlo, analicemos en ambos *sets* la evolución del *ratio* de error al variar el parámetro *mtry*:

8.1.1 Modelo 1

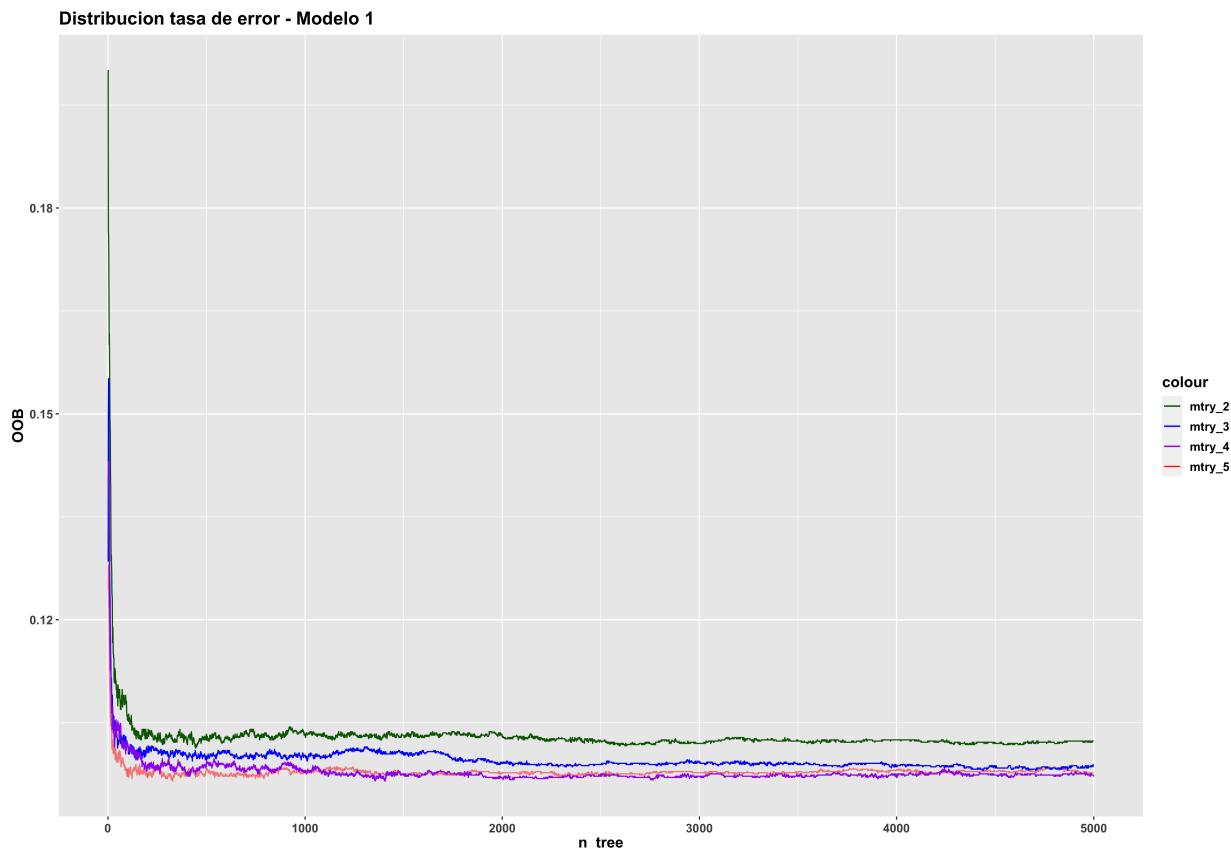


Figure 32: Error rate (Modelo 1) en función de mtry

En primera instancia, observamos que el error (para cualquier *mtry*) se estabiliza prácticamente a partir de 2000 árboles, momento en el que el error *Out of bag* es estable con *mtry* = 2. Sin embargo, con otros valores *mtry*, es decir, **sorteando un mayor número de variables**, el error se estabiliza con antelación:

1. *mtry* = 3: también a partir de 2000 árboles, aproximadamente.
2. *mtry* = 4: a partir de 1500 árboles, aproximadamente.
3. *mtry* = 5: a partir de 900 árboles (caso del modelo *bagging*).

Tras una primera impresión, y utilizando 2000 árboles, **lanzamos los primeros modelos random forest**, utilizando los diferentes valores *mtry*. En relación con el resto de parámetros como *nodesize* o *sampszie*, **utilizamos los empleados en el modelo bagging**: 20 + 1000.

```
##-- 2500: numero de arboles a partir del cual se estabiliza con mtry = 2
# Inicialmente, probamos con 5 repeticiones
mtry.1 <- c(2,3,4,5) # mtry = 5 -> modelo bagging
primera.imp <- tuneo_rf(surgical_dataset, target=target, lista.continua=var_modelo1,
```

```
grupos=5,repe=5,nodesizes=20,mtry=mtry.1,ntree=2000,replace=TRUE,
sampsizes=1000)
```

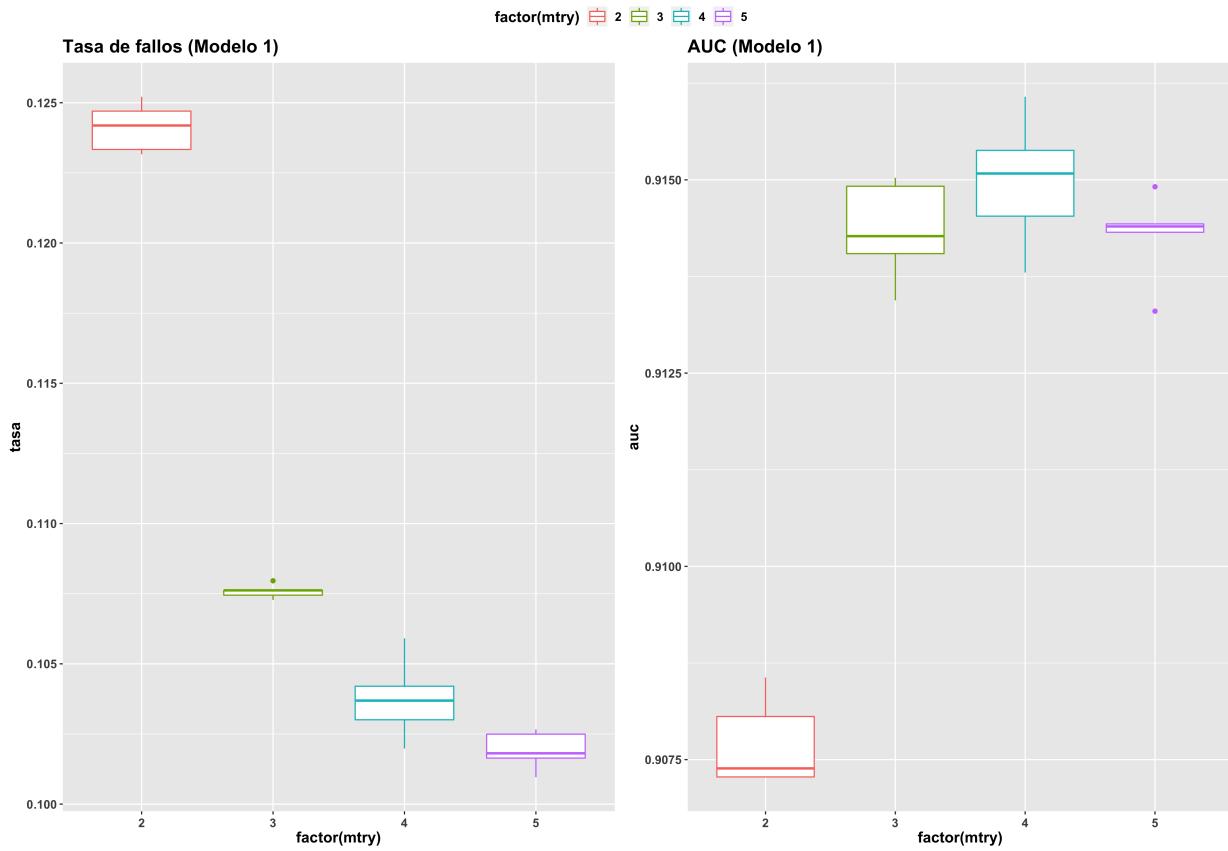


Figure 33: Tasa de fallos y AUC en función de mtry - Modelo 1

A simple vista, podemos comprobar que no es realmente necesario sortear todas las variables en cada nivel del árbol, sino que con un tamaño menor, moderado, en torno a 3, se obtienen los mismos resultados, aunque con una ganancia en la tasa de fallos muy pequeña (0.107-0.108 frente a 0.102 del modelo *bagging*). Bien es cierto que con un $mtry = 2$, la tasa de fallos aumenta (aunque no demasiado) hasta 0.12. Por tanto, y como primera impresión, nos decantamos por un valor $mtry$ moderado (3), pero sin llegar a utilizar todas las variables.

8.1.2 Modelo 2

Del mismo modo, realizamos los mismos pasos para el segundo *set* candidato, comenzando con el número de árboles:

Al igual que en el primer modelo, a excepción de $mtry = 4$ (correspondiente con un modelo *bagging*), para estabilizar el error se requieren un mayor número de árboles, en torno a 2000 aproximadamente tanto para $mtry = 2$ como $mtry = 3$. A continuación, echemos un primer vistazo al comportamiento de los modelos, tanto en sesgo como en varianza, utilizando los diferentes valores $mtry$, así como 2000 árboles y el mejor valor $mtry$ y $nodesize$ obtenidos en *bagging* para el segundo *set* de variables (20 + 1000):

En primera instancia, una posibilidad sería decantarse por un modelo con $mtry = 3$, sin llegar a utilizar todas las variables. Sin embargo, y en base a la escala del eje Y, el error y varianza que presenta el modelo con $mtry = 2$ no es tan pronunciado:

1. En el caso de la tasa de fallos, la diferencia es 0.005 (de 0.1025 en el caso de $mtry = 4$ a 0.1075 en el

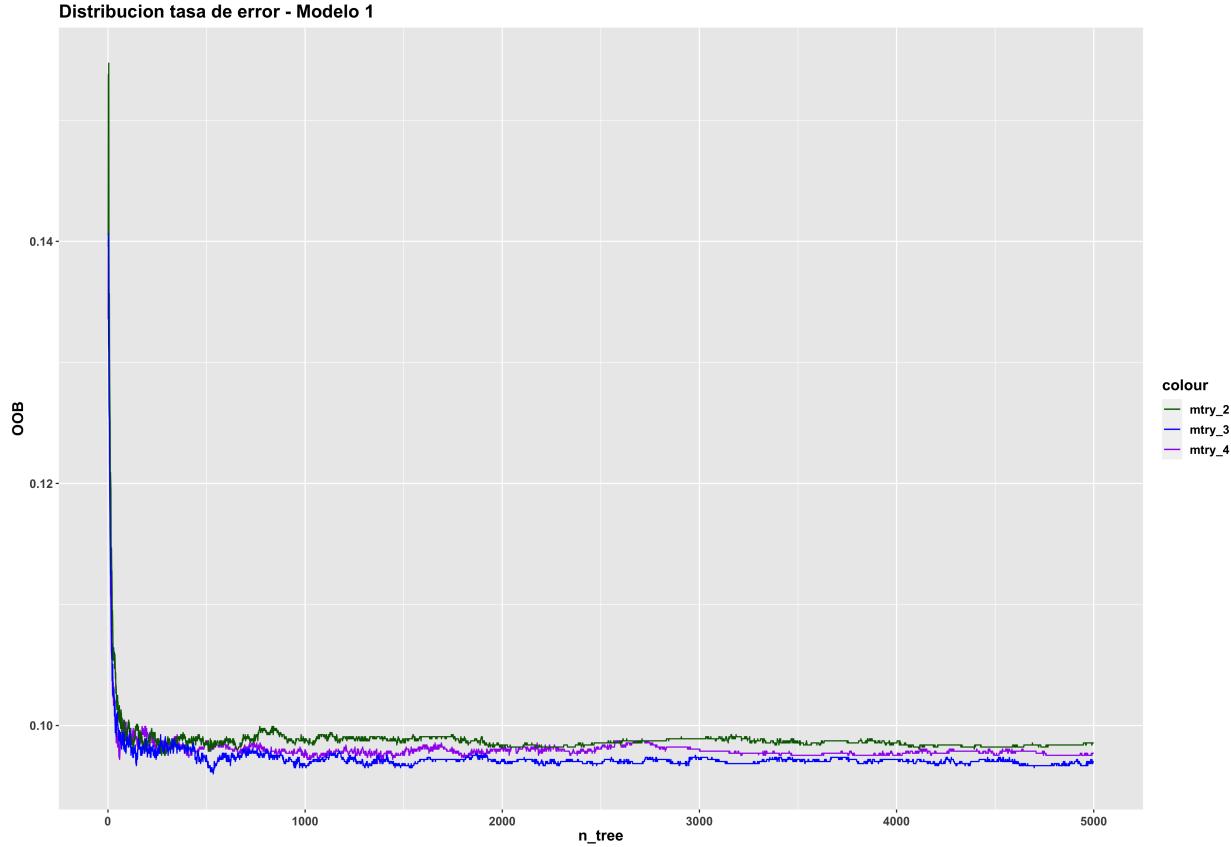


Figure 34: Error rate (Modelo 2) en función de mtry

caso de $mtry = 2$). Además, la varianza que presenta en torno al modelo, teniendo en cuenta la escala del eje, es del orden de milésimas.

2. En el caso del área bajo la curva ROC, la diferencia también se sitúa en torno a las milésimas (de 0.906 en el caso de $mtry = 4$ a 0.904 en el caso de $mtry = 2$).

Dicha diferencia puede ser debida a numerosos factores: estructura de remuestreo, selección de observaciones con *sampsizes*, valor de la semilla etc. Por tanto, dado que la diferencia no es apenas relevante, con el segundo *set* de variables **nos decantamos por un valor $mtry = 2$** .

8.2 Modelo 1

En un primer comienzo, con un valor $mtry = 3$ y el mismo *nodesize* y *sampsizes* que en el modelo *bagging*, obtenemos muy buenos resultados. No obstante, hemos considerado un valor de *nodesize* y *sampsizes* por defecto, esto es, los mejores parámetros obtenidos en *bagging*. Por tanto, ¿Puede llegar a ser influyente el hecho de aumentar o reducir el valor *nodesize*, o más importante aún, el tamaño de la submuestra en *sampsizes*? Del mismo modo que en *bagging*, analicemos la importancia de ambos parámetros, tuneando el modelo con diferentes valores (comenzando con tan solo 5 repeticiones):

```
sampsizes.1 <- list(1, 100, 500, 1000, 2000, 3000, 4600)
nodesizes.1 <- list(5, 10, 20, 30, 40, 50, 100)

# Tuneamos nuevamente tanto nodesizes como sampsizes
# sampsizes maximo: (4/5) * 5854 ~ 4600 observaciones
bagging_modelo1_mtry3 <- tuneo_rf(surgical_dataset, target = target,
```

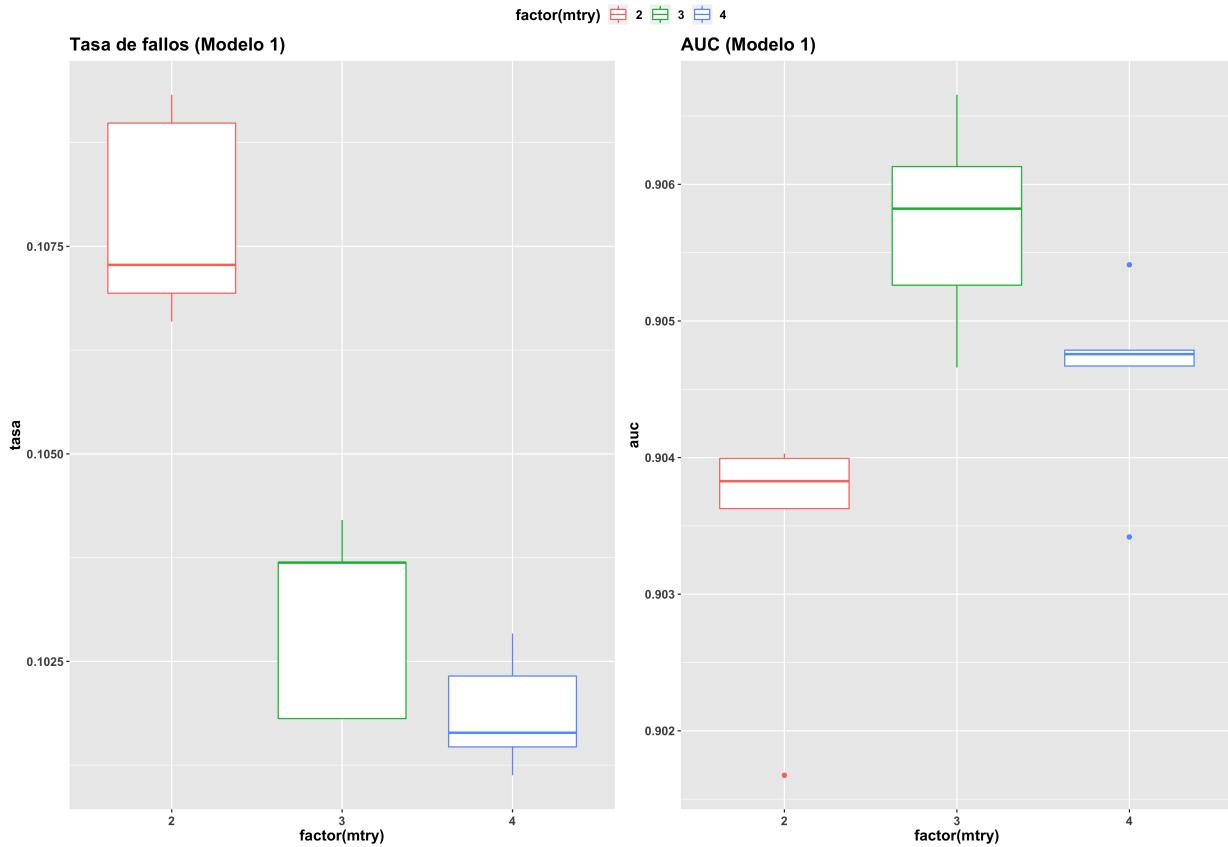


Figure 35: Tasa de fallos y AUC en función de mtry - Modelo 2

```
lista_continua = var_modelo1,
nodesizes = nodesizes.1,
sampsizes = sampsizes.1, mtry = 3,
ntree = 2000, grupos = 5, repe = 5)
```

Analizando el promedio de cada modelo, detectamos prácticamente las mismas características que en el modelo *bagging*:

1. Aumentar la complejidad del árbol, con un *nodesize* menor, la ganancia que supone tanto en tasa de fallos como en AUC no es muy relevante, por lo que podemos mantener 20 como tamaño mínimo de cada nodo.
2. Además, a simple vista un valor *sampsizes* en torno a 500-1000 vuelve a ser una buena opción, obteniendo los mismos resultados que con el conjunto total de las observaciones, lo que permite la construcción de árboles mucho más diferentes entre si gracias al sorteo de menos muestras. No obstante, analicemos más en detalle tanto el sesgo como la varianza al variar el tamaño de la muestra (manteniendo el valor *nodesize* a 20 y aumentando a el número de repeticiones a 10):

```
nodesizes.1 <- list(20)
# Probamos con un tamaño sampsizes entre 500 y 3000 (1 = todas las observaciones)
sampsizes.1 <- list(1, 500, 1000, 2000, 3000)

bagging_modelo1_mtry3 <- tuneo_rf(surgical_dataset, target = target,
                                      lista_continua = var_modelo1,
                                      nodesizes = nodesizes.1,
```

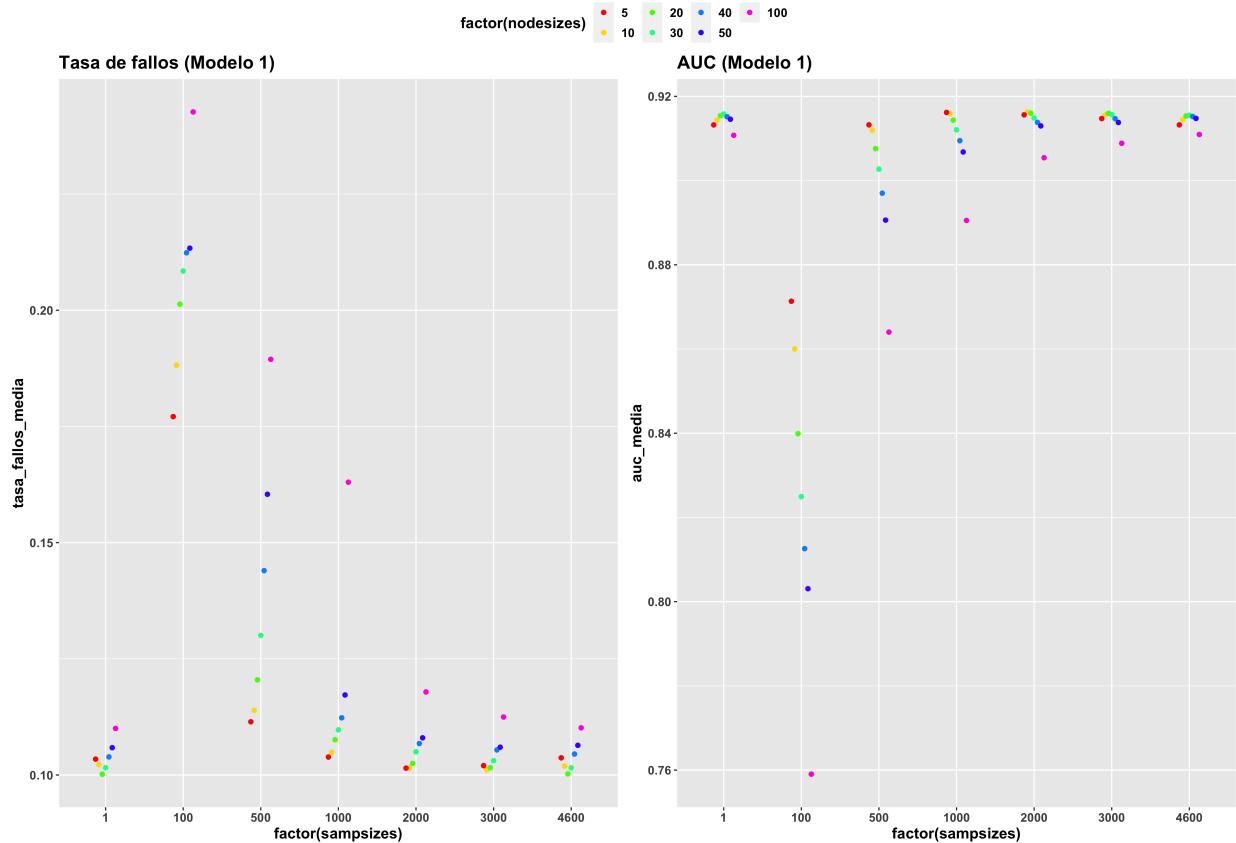


Figure 36: Tasa de fallos y AUC en función de nodesize y sampsize - Modelo 1

```
sampsizes = sampsizes.1, mtry = 3,
ntree = 2000, grupos = 5, repe = 10)
```

Incluso sorteando variables, los resultados obtenido **no difieren demasiado de un modelo bagging**: mientras que con 500 observaciones la tasa de fallos aumenta hasta 0.12 y el AUC se reduce a 0.90, a partir de un tamaño de 1000 muestras (lo que supone alrededor del 17 % del total), **no solo iguala en AUC al resto de modelos con sampsizes mayores (en torno a 0.91)**, sino que además la ganancia de error es de tan solo unas milésimas: entre un sampsize = 1000 y 2000-3000 observaciones, la diferencia es de $0.107 - 0.102 = 0.005$. Por tanto, **aumentar demasiado el tamaño de las submuestras no aporta una mejoría significativa al modelo**, por lo que nos decantamos por un valor de 1000.

Por último, para observar en mejor medida el efecto del parámetro *sampsize*, aumentamos a 10 grupos y 20 repeticiones:

Incluso aumentando a 10 grupos y 20 repeticiones, tanto el sesgo como la varianza se mantienen prácticamente idénticos.

8.2 Modelo 2

A continuación, realizamos los mismos pasos con el segundo *set* de variables, parametrizando en primer lugar tanto *nodesize* como *sampsizes*:

```
sampsizes.2 <- list(1, 100, 500, 1000, 2000, 3000, 4600)
nodesizes.2 <- list(5, 10, 20, 30, 40, 50, 100)
```

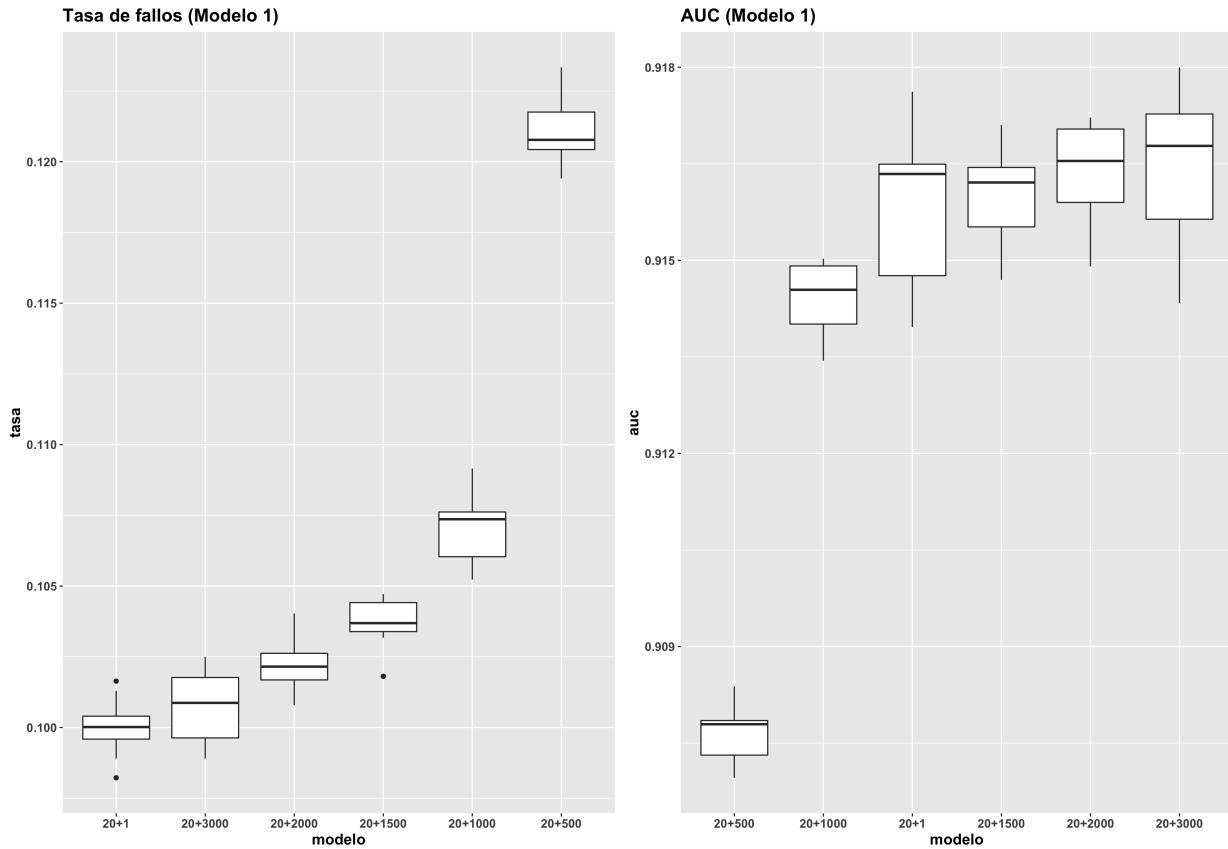


Figure 37: Tasa de fallos y AUC en función de sampsizes - Modelo 1

```
# Tuneamos nuevamente tanto sampsizes como nodesizes
bagging_modelo2_mtry2 <- tuneo_rf(surgical_dataset, target = target,
                                      lista_continua = var_modelo2,
                                      nodesizes = nodesizes.2,
                                      sampsizes = sampsizes.2, mtry = 2,
                                      ntree = 2000, grupos = 5, repe = 5)
```

Del mismo modo que sucede con el primer *set* de variables, tanto con un *nodesize* en torno a 20 como un valor *sampsizes* entre 500 y 1000, parece una buena opción. Además, si analizamos tanto el sesgo como la varianza al variar este último parámetro, aumentando a 10 el número de repeticiones:

Tenemos el mismo comportamiento que el primer *set* de variables: **a partir de un tamaño en torno a 1000 observaciones, la diferencia tanto en la tasa de fallos como en AUC es muy pequeña**, por lo que no merece la pena aumentar aun más el tamaño de la muestra. Incluso si aumentamos el número de grupos y repeticiones a 10 y 20, respectivamente:

El orden de los modelos se mantiene idéntico. Por tanto, dado que el valor AUC es prácticamente el mismo, además de que la ganancia de la tasa de fallos es muy pequeña (dada la escala del eje), nos decantamos nuevamente por un modelo con *sampsizes* = 1000.

RESUMEN *random forest*:

1. modelo 1: *mtry* = 3, *nodesize* = 20 y *sampsizes* = 1000.
2. modelo 2: *mtry* = 2, *nodesize* = 20 y *sampsizes* = 1000.

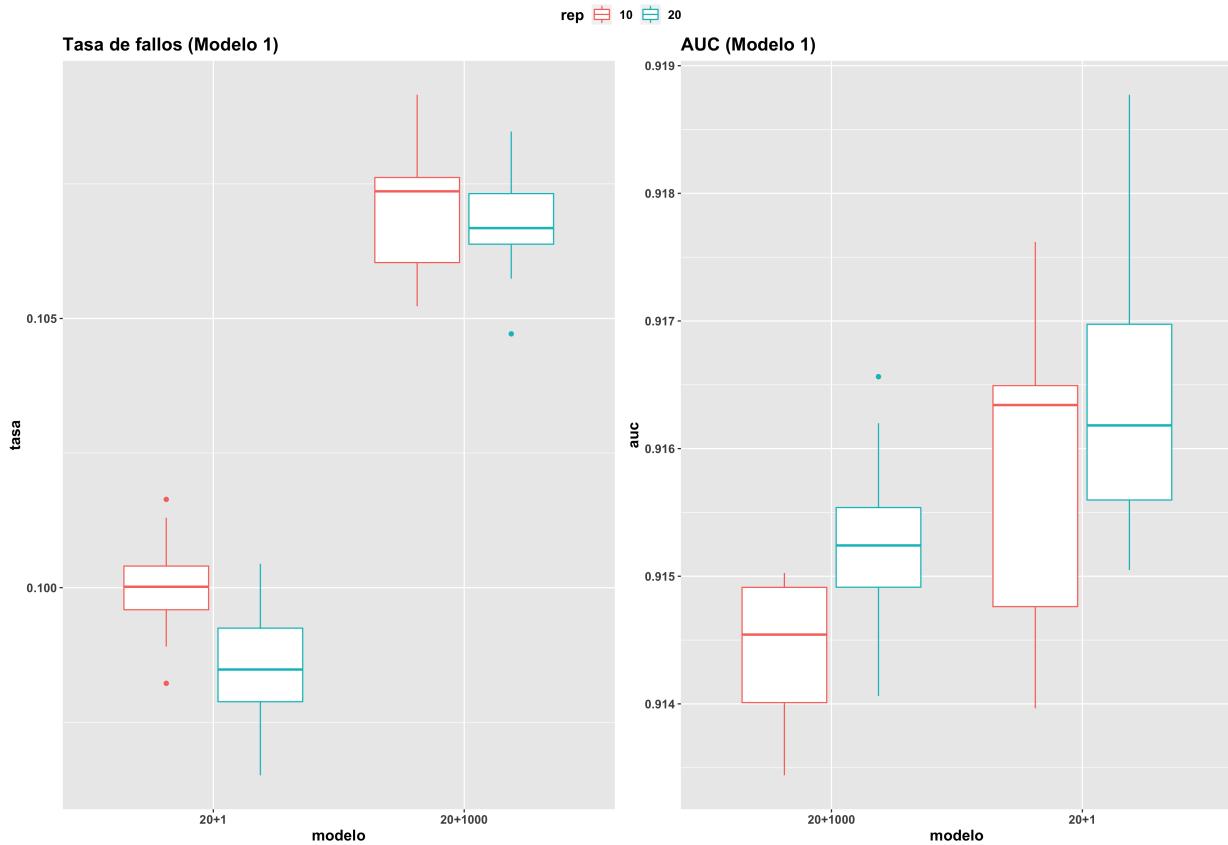


Figure 38: Tasa de fallos y AUC aumentando a 10 grupos y 20 repeticiones - Modelo 1

8.3 Modelo sin reemplazamiento

Una vez tuneados ambos *sets* de variables, veamos la diferencia entre un modelo con *sampsizes* con reemplazamiento y sin reemplazamiento:

```
## ¿Y si lo probamos sin reemplazamiento? Probamos con el mejor modelo en terminos de AUC (modelo 1)
bagging_modelo_sin_reemp <- tuneo_rf(surgical_dataset, target = target,
                                       lista_continua = var_modelo1, nodesizes = 20,
                                       sampsizes = 1000, mtry = 3, ntree = 2000,
                                       grupos = 5, repe = 10, replace = FALSE)
```

Como podemos comprobar, el hecho de seleccionar muestras con o sin reemplazamiento **no aporta mejoría alguna al modelo**.

8.4 Comparación final

Por último, realizamos la comparación final de ambos modelos *random forest* con los datos *test*:

```
##           modelo precision sensibilidad especificidad valor_pred_pos
## 1 Modelo 1 (BIC TOP 5)    0.8990      0.9052      0.8976      0.6639
## 2     Modelo 2 (top 4)    0.8964      0.9117      0.8931      0.6465
##   valor_pred_neg
## 1          0.9769
## 2          0.9792
```

En base a los resultados obtenidos, no se diferencia demasiado de un modelo *bagging*:

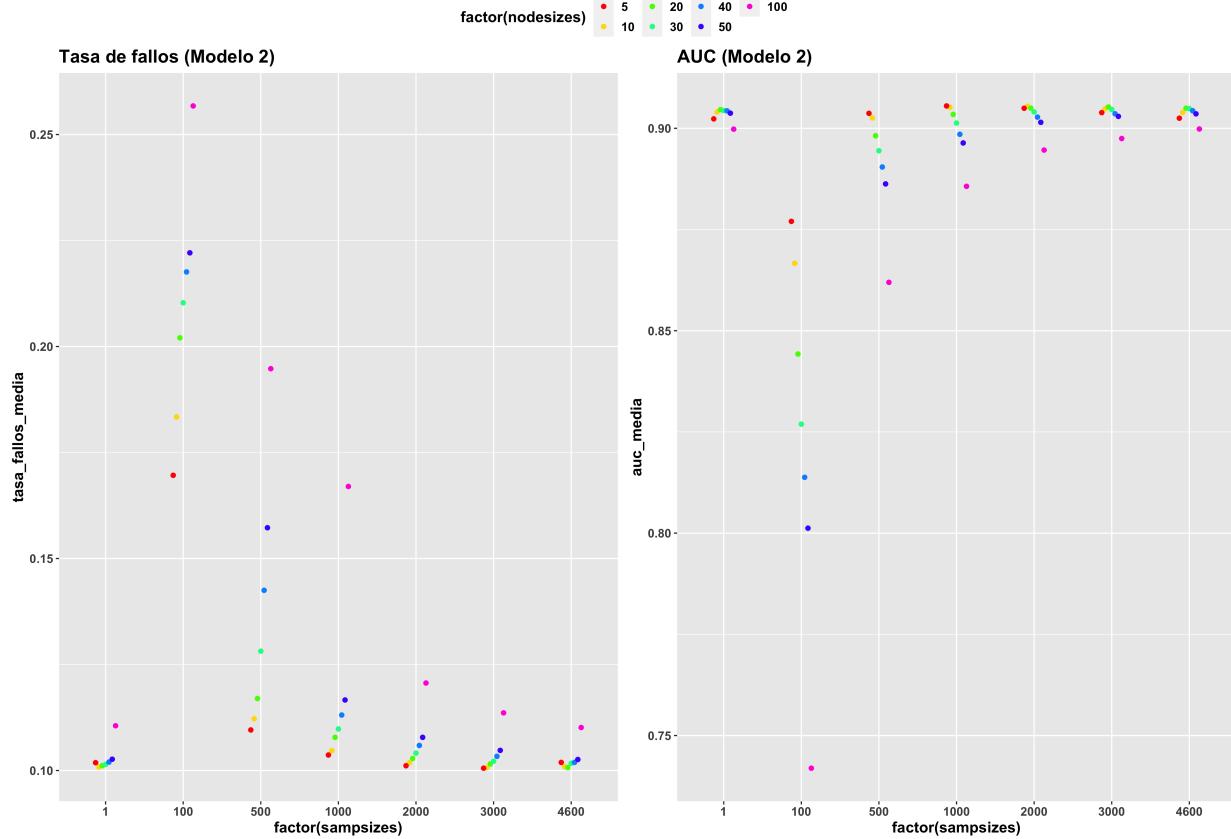


Figure 39: Tasa de fallos y AUC en función de nodesize y sampsizes - Modelo 2

```
##           modelo precision sensibilidad especificidad valor_pred_pos
## 1 Modelo 1 (BIC TOP 5)    0.9010      0.9214      0.8966      0.6589
## 2     Modelo 2 (top 4)    0.9019      0.9468      0.8928      0.6424
##   valor_pred_neg
## 1          0.9813
## 2          0.9880
```

1. **Precisión:** de 0.901 y 0.9019 a 0.899 y 0.8964 en ambos *sets* de variables, respectivamente.
2. **Sensibilidad:** de 0.9214 y 0.9468 a 0.9052 y 0.9117, respectivamente (empeora ligeramente).
3. **Especificidad:** tanto en *bagging* como en *random forest* se sitúa en torno a 0.89.
4. **Valor predictivo positivo:** continúa situándose en torno a 0.64-0.65-0.66.
5. **Valor predictivo negativo:** continúa situándose en torno a 0.97-0.98.

Es decir, añadiendo el sorteo de variables al modelo (*mtry*), la variación que experimenta en el conjunto *test* es muy pequeña. De hecho, si observamos la distribución de la tasa de fallos y AUC:

En relación al resto de modelos, *random forest* se sitúa a la misma altura que modelos como *bagging* o *avnnnet*. De hecho, si hacemos “zoom” sobre los modelos de árbol:

Comprobamos que la diferencia entre ambos no es muy significativa, como es el caso de la tasa de fallos, donde la escala del eje puede llevar a engaño, ya que la diferencia entre ambos modelos es de apenas unas milésimas (de 0.1000 - 0.1025 en el caso de *bagging* a 0.1050-0.1075 en *random forest*). Además, aunque la varianza en este último parezca aumentar, dada la escala del eje no es tan pronunciada.

De nuevo, no existe demasiada diferencia entre ambos *sets* de variables, tanto en tasa de fallos como en AUC, pese a la diferencia de una variable *input*.

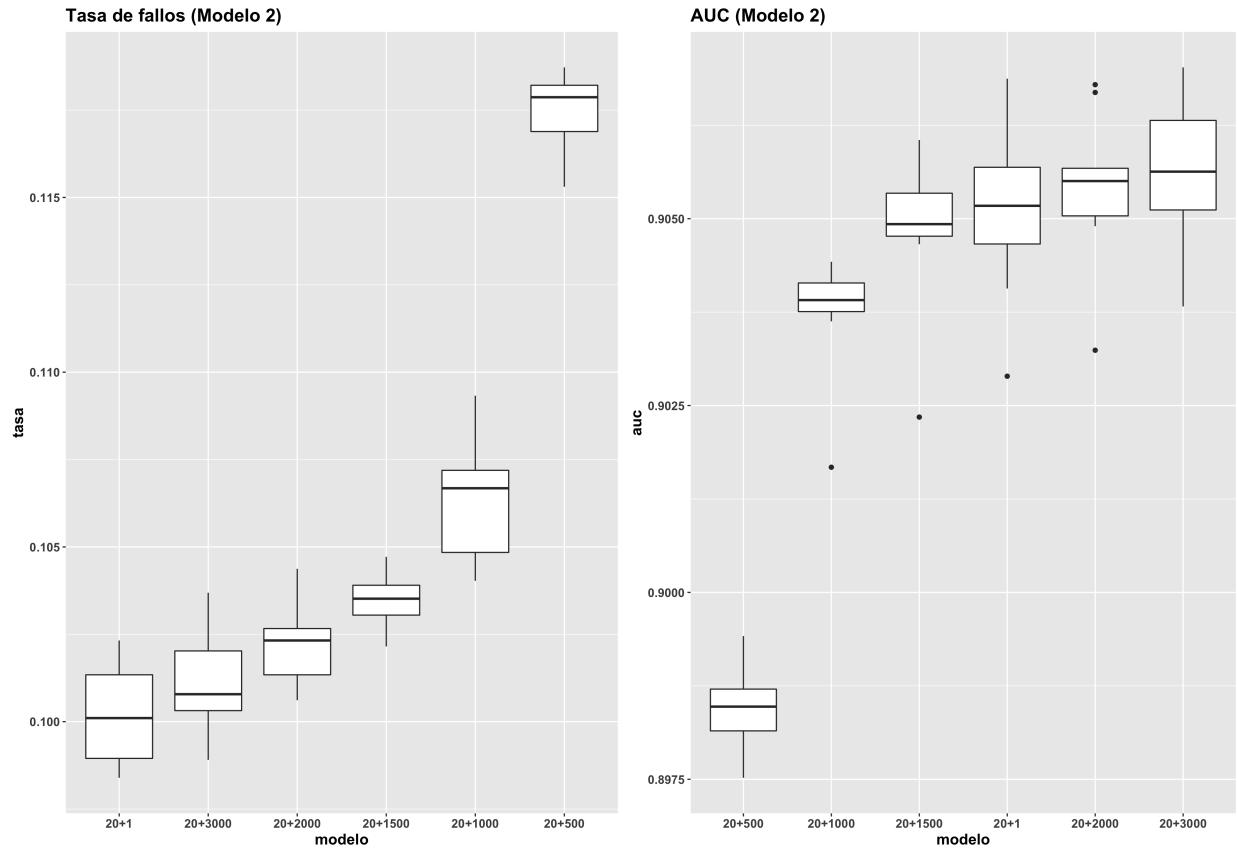


Figure 40: Tasa de fallos y AUC en función de sampszie - Modelo 2

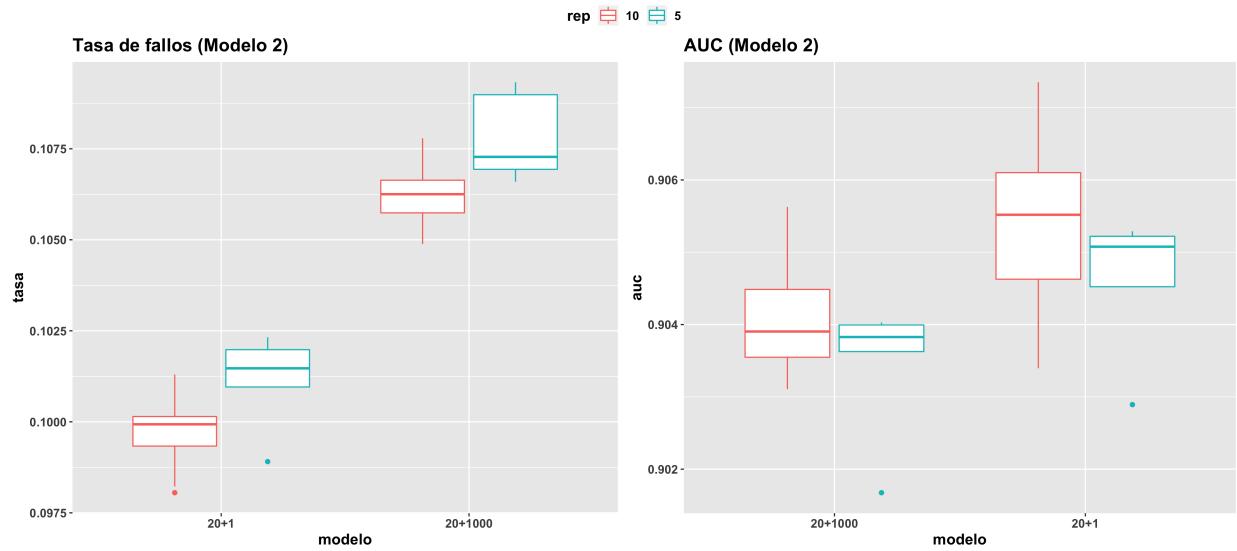


Figure 41: Tasa de fallos y AUC aumentando a 10 grupos y 20 repeticiones - Modelo 2

9. Gradient Boosting

Hasta el momento, hemos comprobado como la variación en el *accuracy*, tasa de error o AUC con los ambos *sets* candidatos no resulta ser relevante, incluso en numerosas ocasiones los parámetros finales acaban siendo

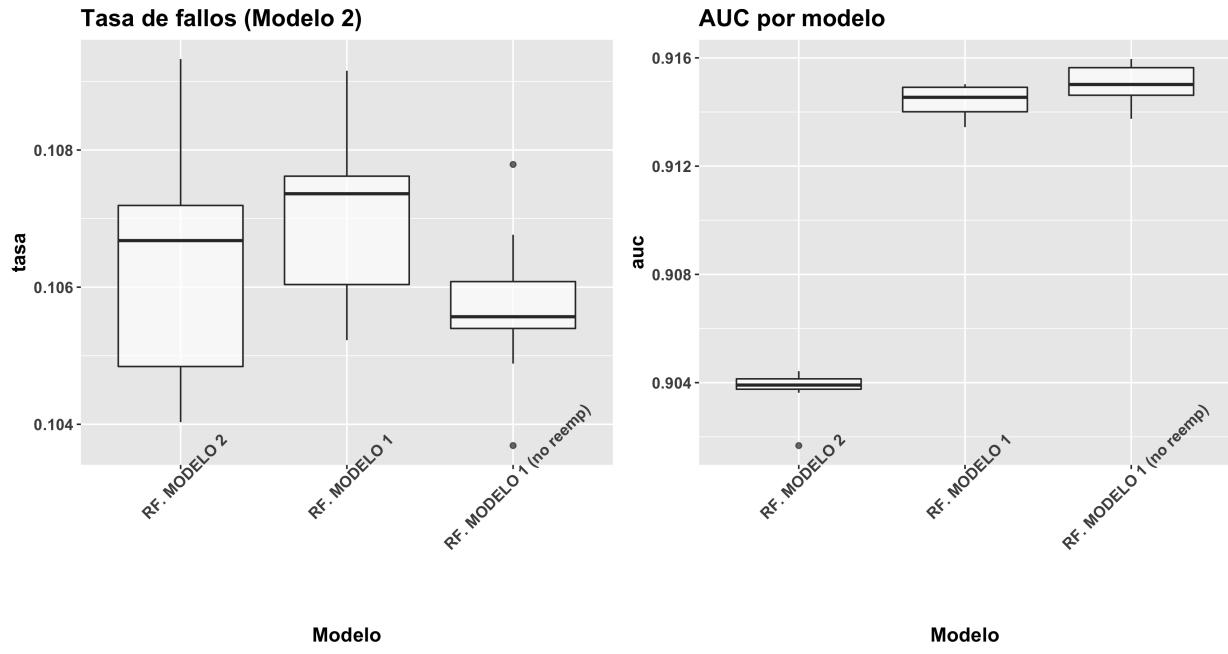


Figure 42: Comparación Random Forest con o sin reemplazamiento

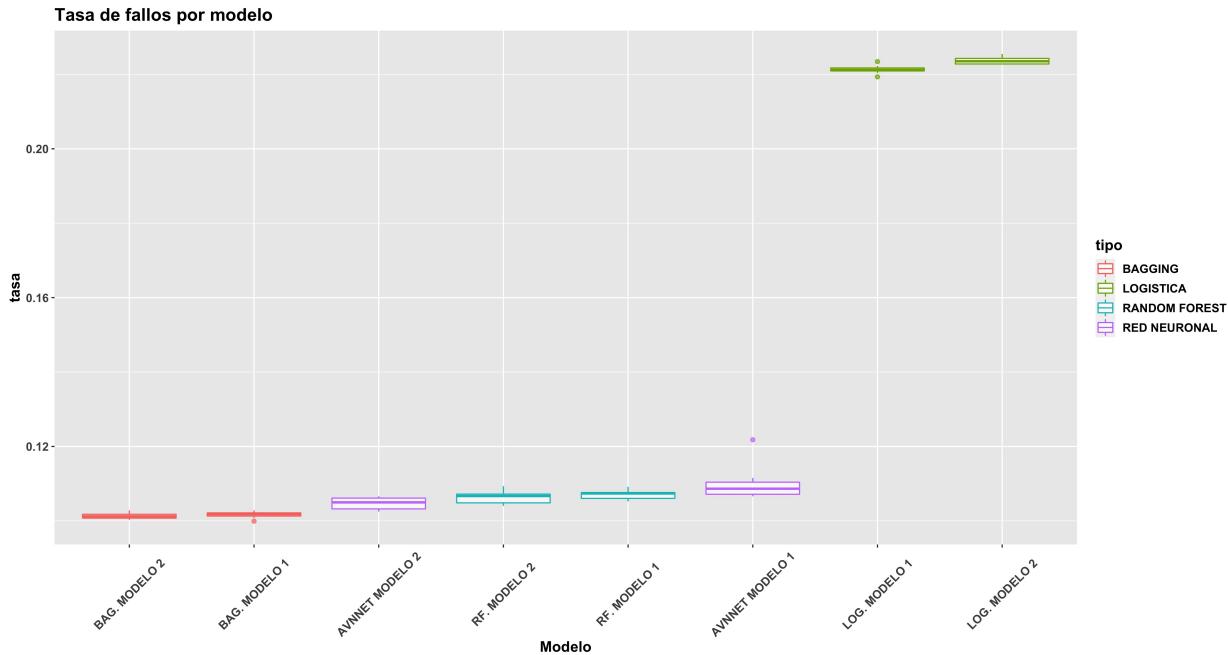


Figure 43: Comparacion tasa fallos log-avnnet-bagging-rf

los mismos para ambos candidatos. Por tanto, de cara al resto de modelos como *gradient boosting*, *xgboost* o *svm*, veamos si es posible poner en común un mismo tuneo de hipérparametros para ambos *sets*, en lugar de “tunear” por separado cada uno de ellos.

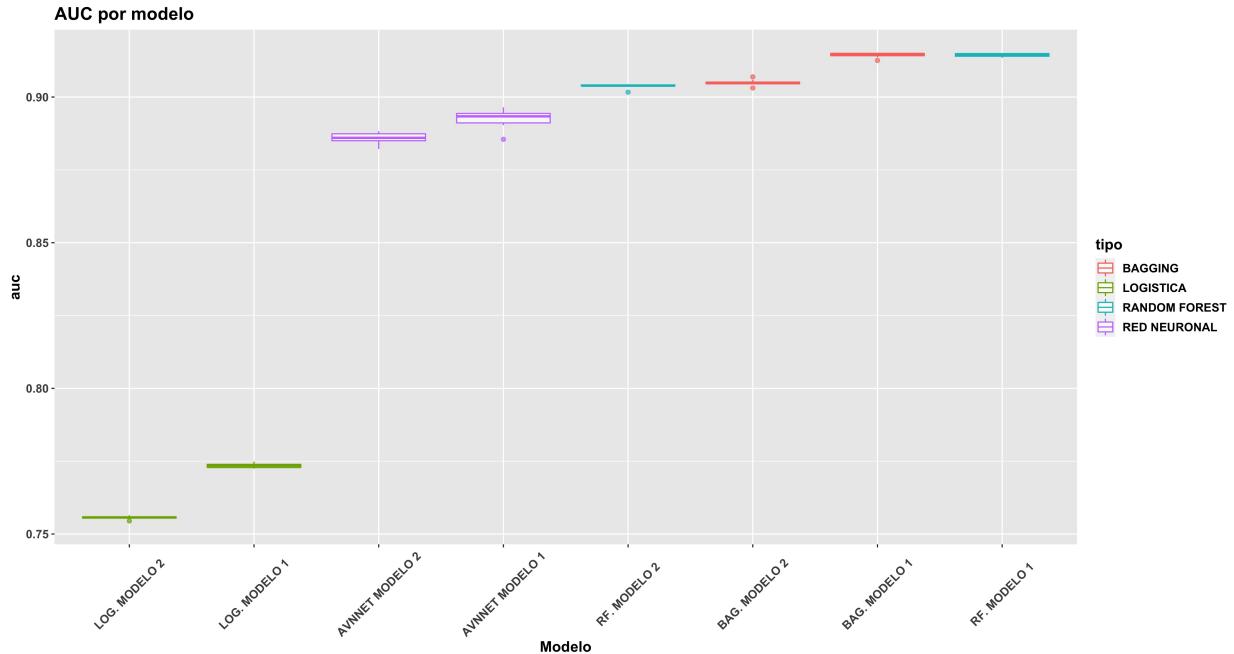


Figure 44: Comparacion AUC log-avnnet-bagging-rf

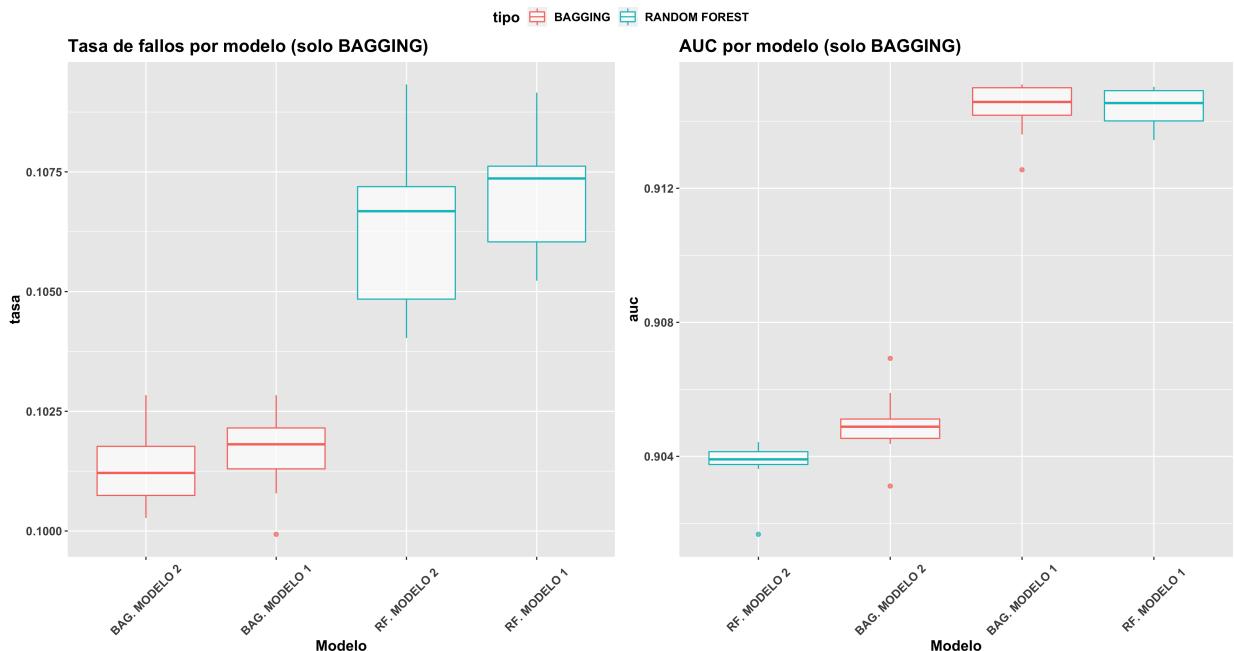


Figure 45: Comparacion modelos bagging y rf

9.1 Tuneo de hipérparámetros

En primer lugar, comenzando con el tuneo de los diferentes hiperparámetros en ambos modelos, con tan solo cinco repeticiones:

1. *shrinkage*: parámetro de regularización.
2. *n.minobsinnode*: tamaño máximo de nodos finales (parámetro encargado de medir la complejidad).

3. *n.trees*: número de iteraciones (árboles).
4. *bag.fraction*: fracción de observaciones del conjunto de entrenamiento seleccionadas aleatoriamente para la construcción del siguiente árbol.

```
-- Ejemplo de tuneo con el primer set de variables
set.seed(1234)
-- El valor maximo de shrinkage suele estar en torno a 0.2 (probamos con 0.3 y 0.4 tambien)
gbmgrid<-expand.grid(shrinkage=c(0.4,0.3,0.2,0.1,0.05,0.03,0.01,0.001),
                      n.minobsinnode=c(5,10,20),
                      n.trees=c(100,500,1000,5000),
                      interaction.depth=c(2))

-- De momento, mantenemos bag.fraction a 1 (todas las observaciones)
control<-trainControl(method="repeatedcv",number=5,savePredictions = "all",
                       repeats=5,classProbs=TRUE)
gbm_modelo1 <- train(formula_modelo_1,data=surgical_dataset,
                      method="gbm",trControl=control,tuneGrid=gbmgrid,
                      distribution="bernoulli", bag.fraction=1,verbose=FALSE)
```

Gradient Boosting Hyperparameters Tuning (Modelo 1)

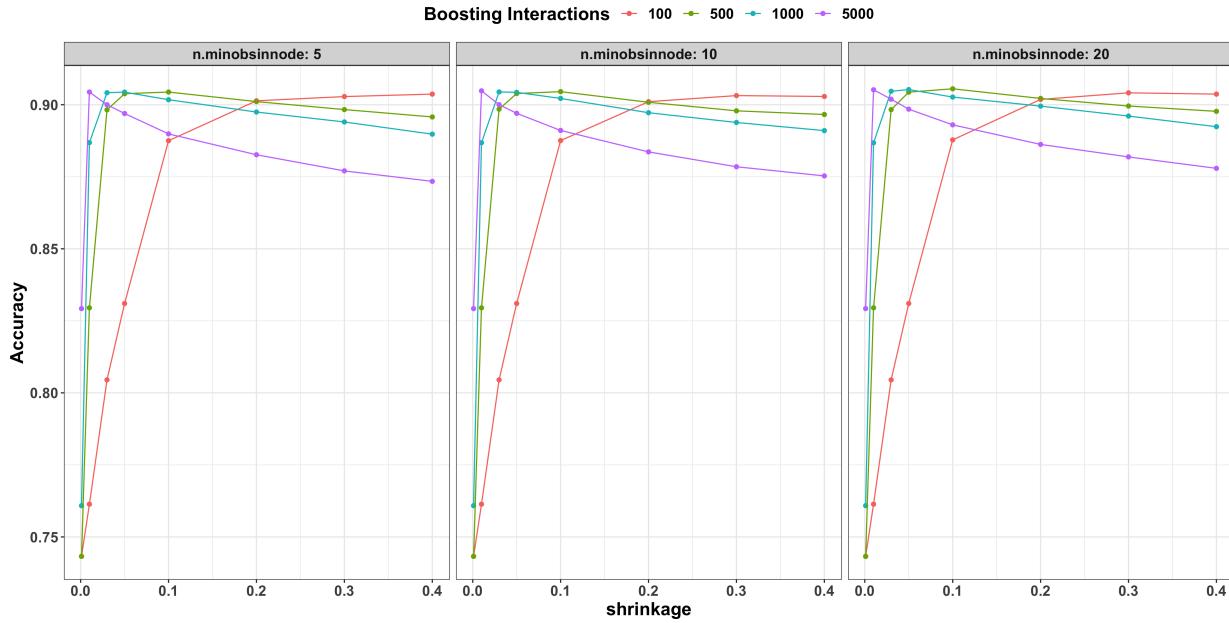


Figure 46: Comparacion hiperparámetros gbm modelo 1

¿Qué sucede en ambos *sets* de variables? En primer lugar, analicemos las recomendaciones de *caret* para ambos *sets*:

Modelo 1: *n.trees* = 500, *shrinkage* = 0.1, *n.minobsinnode* = 20. Accuracy: 0.9055354
 Modelo 2: *n.trees* = 5000, *shrinkage* = 0.01, *n.minobsinnode* = 20. Accuracy: 0.9039292

En función de lo recomendado por el propio paquete, existe un contraste entre ambos modelos: por un lado, con cinco variables *input* recomienda 500 iteraciones, un valor de regularización alto y *n.minobsinnode* alto (20), es decir, modelos más simples. Por el contrario, con una variable *input* menos, recomienda muchos más árboles (5000) y un valor de regularización más bajo.

Desde un punto de vista numérico, con estos parámetros se han obtenido el *accuracy* máximo (incluso también por la estructura de remuestreo, valor de la semilla, etc.). Sin embargo, ¿Es una configuración óptima? O aún más importante ¿Podrían obtenerse resultados similares con modelos más sencillos? Para responder a

Gradient Boosting Hyperparameters Tuning (Modelo 2)

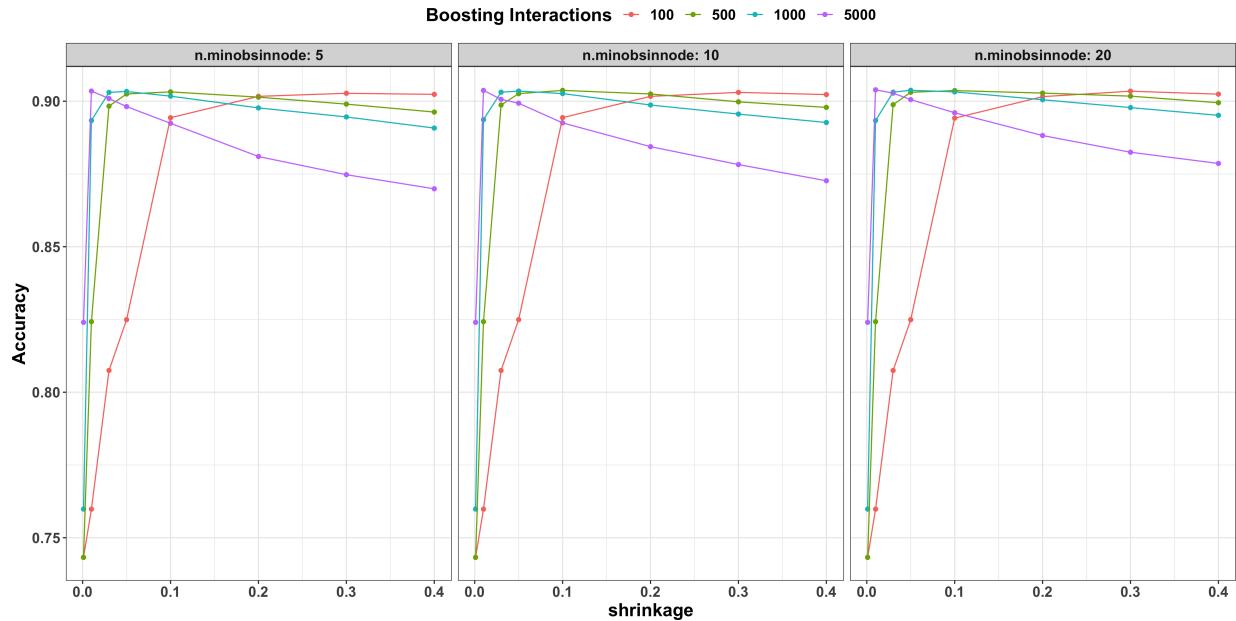


Figure 47: Comparación hiperparámetros gbm modelo 2

ambas preguntas, **debemos analizar los posibles patrones que presenta el modelo en ambos sets**, de forma que podamos decantarnos por una opción mucho más sencilla.

Concretamente, destacamos tres grandes patrones de comportamiento:

1. Por un lado, **un bajo número de iteraciones (en torno a 100)**, junto con **un valor de regularización alto (shrinkage superior o igual a 0.2-0.3, aproximadamente)**.
2. Por el contrario, también es posible obtener un máximo *accuracy* con **un valor shrinkage bajo (inferior a 0.1)**, pero con **un elevado número de iteraciones, en torno a 500, 1000 o 5000**.
3. Además, el comportamiento de *n.minobsinnode* en ambos sets es el mismo tanto para un valor de 5, como de 10 o 20, **por lo que un valor alto (20) es una buena opción a decantarse: a un mayor valor *n.minosinnode*, obtenemos modelos más simples y con menos posibilidad de sobreajuste**.

En conclusión, **no es necesario tunear exageradamente el modelo en ambos casos**, sino que aparentemente con un número de árboles bajo (100), un valor de regularización alto (en torno a 0.2-0.3) y *n.minobsinnode* = 20 se obtienen de por sí buenos resultados, sin necesidad de complicar aún más el modelo.

9.1.1 Estudio del *Early Stopping*

Continuando con la línea anterior, para ambos set de variables **probamos a fijar *n.minobsinnode* = 20 y *shrinkage* = 0.2 y 0.3**, observando no solo qué número de iteraciones/árboles sería adecuado utilizar, sino además si con un valor de regularización de 0.2 es suficiente o si aumentando su valor obtenemos mejores resultados, ya que en el gráfico anterior se detectó un ligero ascenso en el valor de *accuracy* en 0.3 (normalmente el valor máximo suele ser 0.2):

En ambos sets, **observamos que con 100 árboles se obtiene un buen *accuracy***, mientras que a partir de 400-500 iteraciones, el modelo comienza a sobreajustarse y, como consecuencia, disminuye su precisión. Por otro lado, en relación con el parámetro *shrinkage* no existe apenas diferencia entre un valor 0.2 y 0.3, por lo que **nos decantamos por 0.2**.

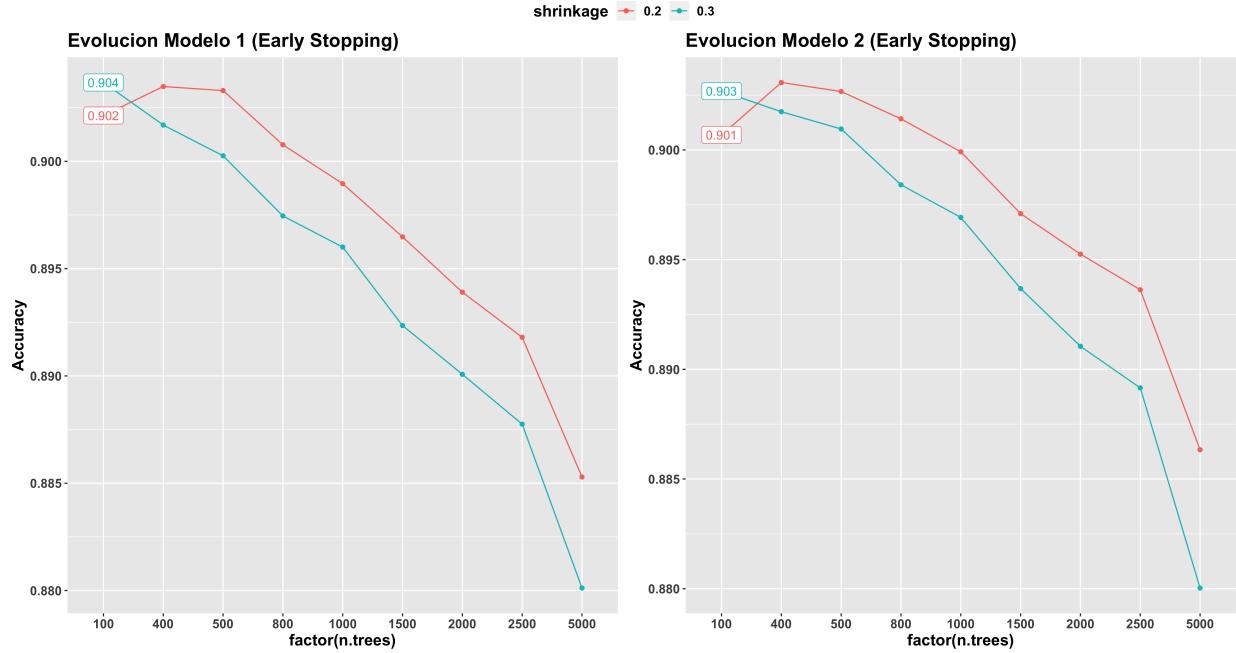


Figure 48: Estudio Early Stopping Modelos 1 y 2

9.1.2 Estudio de *bag.fraction*

Hasta el momento, hemos tuneado los modelos *gbm* con el conjunto total de las observaciones, esto es, con *bag.fraction* = 1. Sin embargo, *caret* por defecto no sorteá todas las observaciones, sino que por defecto escoge el 50 % aleatoriamente (0.5)⁶

Como consecuencia, sobre ambos *sets* estudiamos la variabilidad del modelo con respecto a dicho parámetro, aumentando a 10 el número de repeticiones para observar mejor su efecto. Dado que la función *cruzadagbm* nos dispone de dicho parámetro, se ha incluido internamente en la función *train* de *caret*:

```
#-- bag.fraction no se permite tunear desde expand.grid...
gbmgrid <- expand.grid(n.minobsinnode=n.minobsinnode, shrinkage=shrinkage,
                        n.trees=n.trees, interaction.depth=interaction.depth)
#-- Por lo que lo incluimos dentro de train
gbm     <- train(formu,data=databis, method="gbm",trControl=control,
                  bag.fraction=bag.fraction, tuneGrid=gbmgrid,distribution="bernoulli",
                  verbose=FALSE)
```

En un primer análisis, la diferencia entre *bag.fraction* = 0.5 y *bag.fraction* = 1 es pequeña, ya que tanto en sesgo como en varianza presentan resultados similares, salvo en la tasa de fallos donde aumenta ligeramente, pero no demasiado dada la escala de los ejes (por ejemplo, en el modelo 1 la varianza se sitúa entre 0.099 y 0.104, considerando los *outliers*). Por tanto, no existe apenas diferencia entre utilizar el 100 % de las observaciones y tan solo el 50 %, por lo que podemos mantener dicho parámetro por defecto en *bag.fraction*.

RESUMEN *gradient boosting*:

- modelos 1 y 2: *n.trees* = 100, *shrinkage* = 0.2, *n.minobsinnode* = 20 y *bag.fraction* = 0.5.

⁶*caret* emplea internamente el paquete *gbm*, cuyo valor *bag.fraction* por defecto es 0.5. <https://cran.r-project.org/web/packages/gbm/gbm.pdf>

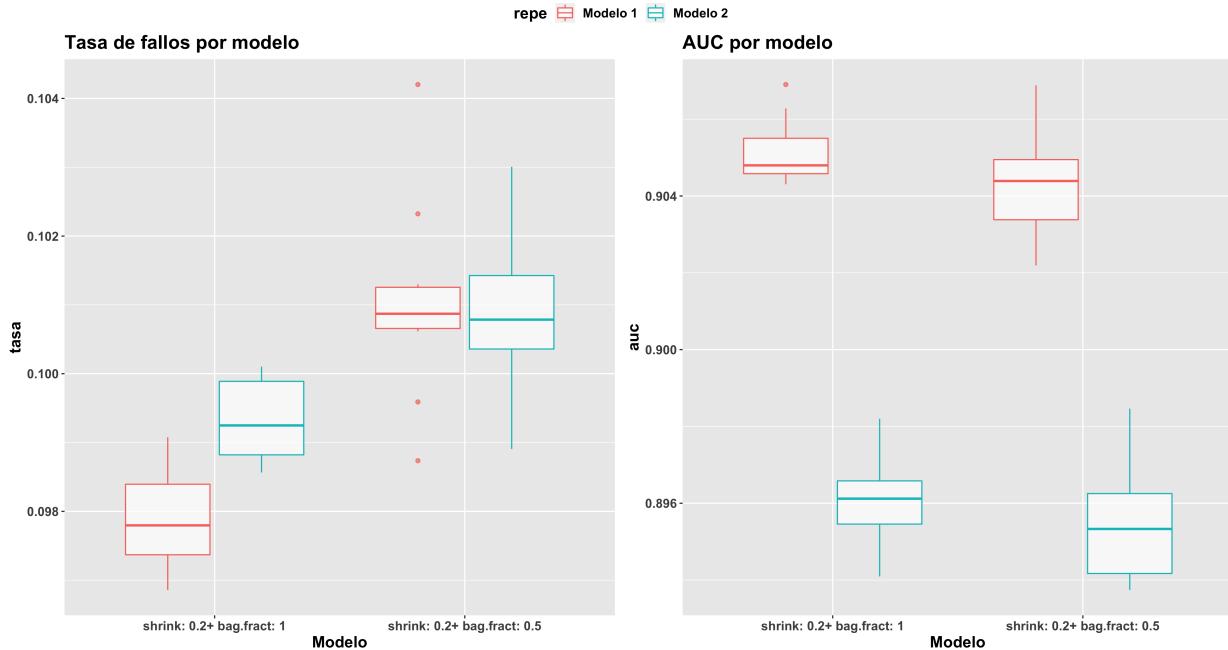


Figure 49: Estudio bag.fraction modelos 1 y 2

9.2 Comparación final

A continuación, con los modelos obtenidos realizamos la predicción sobre el conjunto *test*:

```
##           modelo precision sensibilidad especificidad valor_pred_pos
## 1 Modelo 1 (BIC TOP 5)    0.9030      0.9149      0.9003      0.6731
## 2     Modelo 2 (top 4)    0.9052      0.9264      0.9006      0.6731
##   valor_pred_neg
## 1          0.9792
## 2          0.9823
```

Como podemos comprobar, *gradient boosting* se sitúa prácticamente en la misma línea que el resto de modelos basados en árbol: **alta precisión, sensibilidad, especificidad y valor predictivo negativo, pero con un valor predictivo positivo menor, en torno al 67 %**:

De hecho, aunque la tasa de fallos en *gradient boosting* sea ligeramente menor, **modelos como random forest o bagging continúan siendo una mejor alternativa**.

10. Support Vector Machines

10.1 SVM Lineal

A continuación, continuamos con los modelos *svm*, comenzando por el *kernel* más sencillo: el lineal, del cual solo necesitamos tunear el parámetro de regularización **C** (por el momento, empleamos 5 repeticiones):

```
-- Probamos desde C = 0.01 hasta C = 10
C_binaria <- expand.grid(C=c(0.01,0.05,0.1,0.2,0.5,1,2,5,10))

-- Ejemplo con el primer set de variables (5 variables input)
cruzadaSVMbin(data=surgical_dataset, vardep=target,
               listconti = var_modelo1, listclass=c(""),
               grupos=5,sinicio=1234,repe=5,C=C_binaria)
```

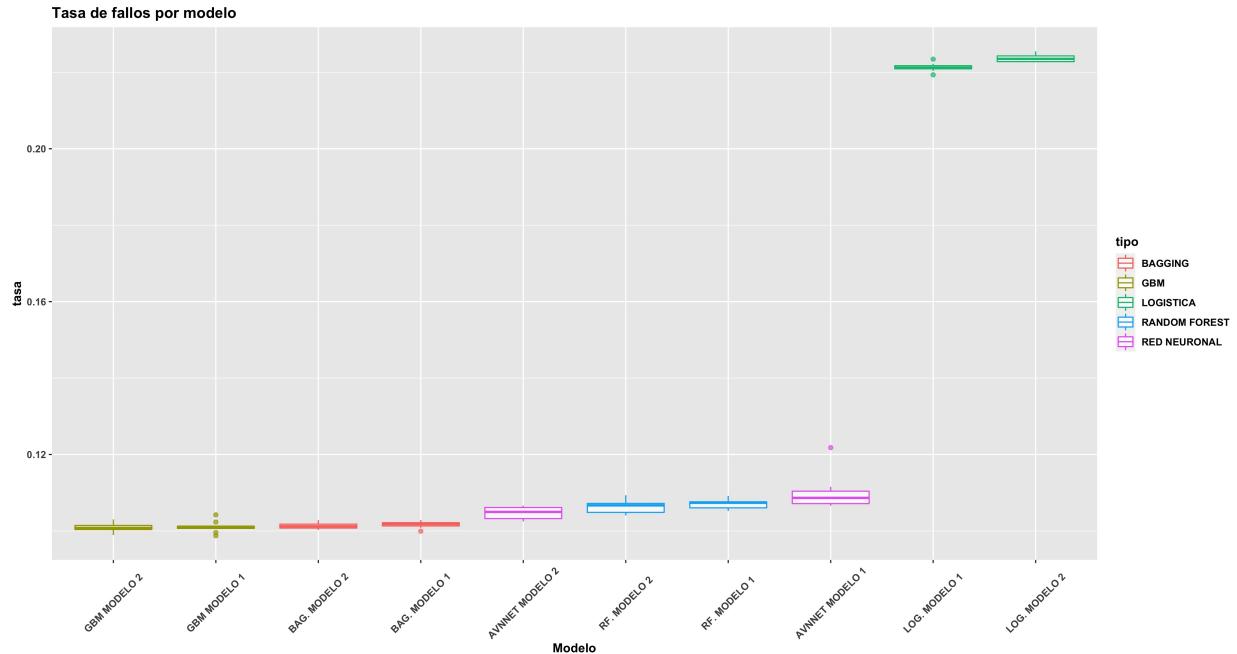


Figure 50: Comparacion tasa fallos log-avnnet-bagging-rf-gbm

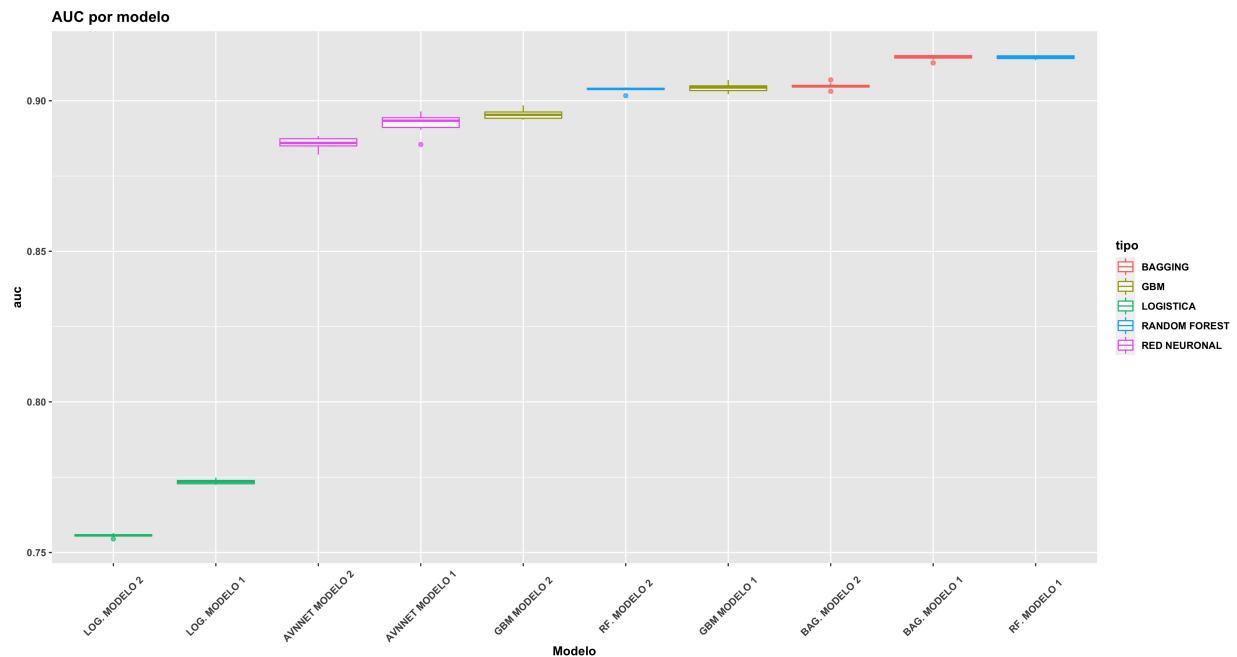


Figure 51: Comparacion AUC log-avnnet-bagging-rf-gbm

Recomendado por caret

Modelo 1: C = 0.01. Accuracy: 0.7746143

Modelo 2: C = 0.01. Accuracy: 0.7466688

A diferencia del resto de modelos no lineales, con el primer *set* de variables (5 variables *input* incluyendo *ccsMort30Rate*), obtenemos un *accuracy* en torno al 77 %, mientras que con tan solo 4 variables desciende al 74 %, de nuevo un indicativo de la relación lineal de dicha variable con la variable objetivo. Pese a ello, los

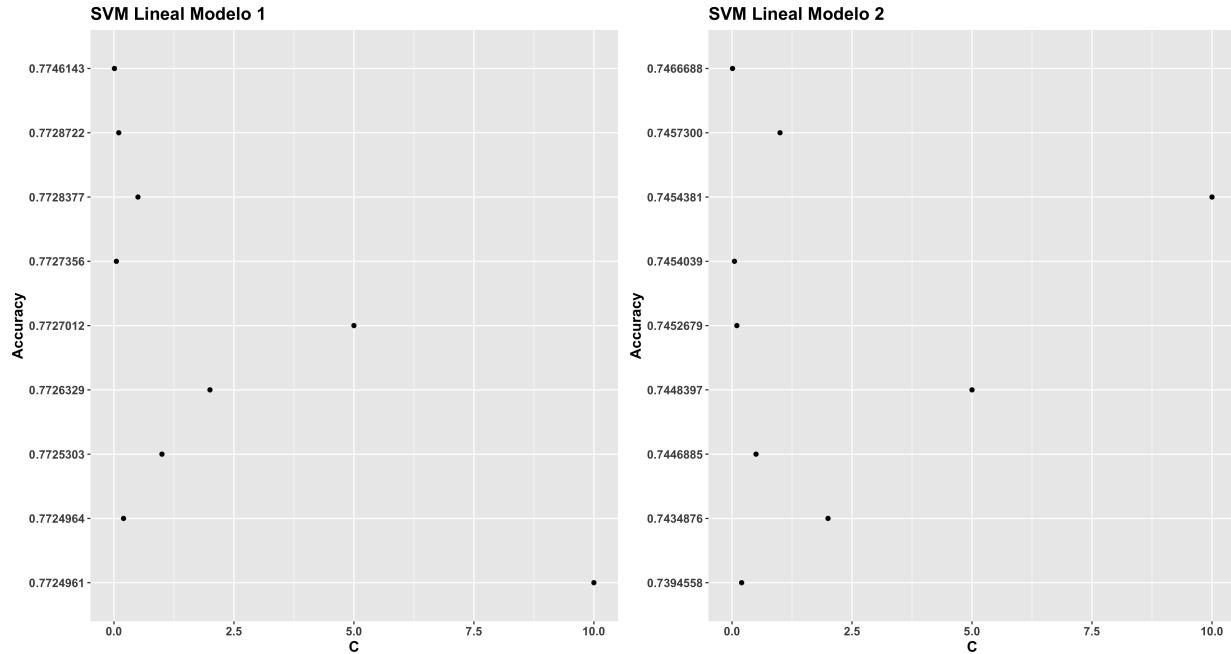


Figure 52: Distribución SVM Lineal Modelos 1 y 2

modelos basados en árbol o *boosting* continúan siendo mejor en términos de *accuracy* (89-90 %).

En relación con la constante C, dada la escala de los ejes **no se aprecia una diferencia relevante entre los diferentes valores**, ya que apenas varían en unas milésimas. Sin embargo, si se tuviera que indica un posible patrón, diríamos que a medida que se reduce la constante C, aumenta ligeramente el *accuracy*. Podríamos incluso probar a reducir aún más el parámetro de regularización:

```
#-- Probamos a reducir C entre 0.1 y 0.001
C_binaria <- expand.grid(C=c(0.001,0.005,0.01))
```

Pero tan solo aumenta en 0.0001 en ambos *sets*:

```
Modelo 1: C: 0.01 -> 0.7746143 ; C: 0.001 -> 0.7749898
Modelo 2: C: 0.01 -> 0.7466688 ; C: 0.001 -> 0.7749898
```

Por tanto, y dado que la diferencia no es muy significativa, **nos decantamos por el máximo *accuracy*, correspondiente con C = 0.01**, recomendado por *caret*, un valor no demasiado alto, lo que se traduce en un menor riesgo por sobreajuste (márgenes de separación mucho mayores).

10.2 SVM Polinomial

A continuación, proseguimos con el *kernel* polinomial, **tuneando tanto la constante C de regularización como la escala** (nuevamente, con cinco repeticiones):

NOTA: dado que los polinomios de grado 3 tardaban demasiado tiempo en procesar (incluso con una sola repetición + *doParallel*), se ha utilizado únicamente grado 2.

```
#-- Probamos con C desde 0.01 hasta 10
C_poly      <- c(0.01, 0.05, 0.1, 0.2, 0.5, 1, 2, 5, 10)
degree_poly <- c(2)
#-- Probamos desde una escala pequeña (0.1) hasta 5
scale_poly  <- c(0.1, 0.5, 1, 2, 5)
```

```

#-- Ejemplo con el primer set de variables (5 variables input)
svm_pol_1 <- cruzadaSVMbinPoly(data=surgical_dataset, vardep=target,
                                listconti = var_modelo1, listclass=c(""),
                                grupos=5,sinicio=1234,repe=5,C = C_poly,
                                degree = degree_poly, scale = scale_poly)

```

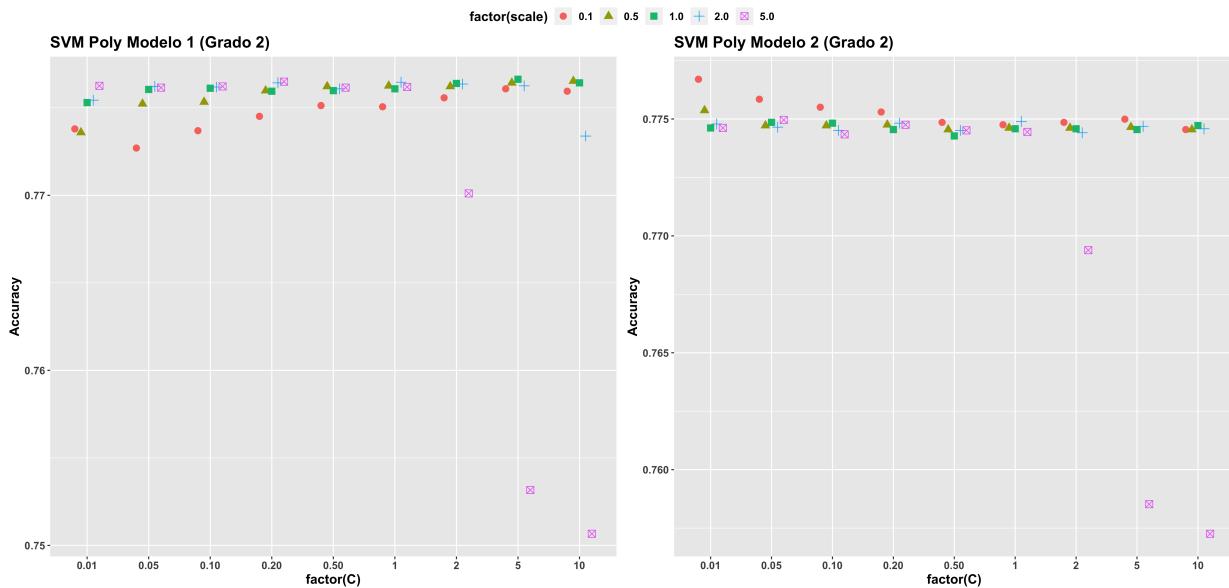


Figure 53: Distribución SVM Polinomial Modelos 1 y 2 (I)

Recomendado por caret

Modelo 1: $C = 5$; $\text{degree} = 2$; $\text{scale} = 0.1$. Accuracy: 0.7766301

Modelo 2: $C = 0.01$; $\text{degree} = 2$; $\text{scale} = 0.1$. Accuracy: 0.7766992

En base a la respuesta del paquete *caret*, con el primer *set* recomienda una escala en torno a 0.1 y una constante C alta (5), mientras que en el caso del segundo *set* de variables, la constante C recomendada es más baja, en torno a 0.01. No obstante, y dado que no podemos dejarnos llevar por valores puntuales (pues puede conducir al sobreajuste), analicemos los patrones existentes:

1. En el caso del primer *set* de variables, nos encontramos con dos posibles extremos: **o bien una constante C alta (entre 5 y 10) y una escala moderada/pequeña (en torno a 0.1-0.5-1) o una constante C pequeña y una escala mayor (1, 2 o 5).**
2. En el segundo *set*, por otro lado, **llama la atención modelos con una constante C pequeña (menor a 0.01) y una escala baja (0.1-0.5, principalmente).**

Por tanto, analicemos si mejoran los resultados en caso de, en el primer *set*, aumentar la constante C a 15 y 20, así como disminuirlo a 0.005 y 0.001. Por otro lado, probamos también a reducir C a 0.005 y 0.001 en el segundo *set* de variables:

1. En el caso del primer *set*, pese a reducir o aumentar la constante C, el valor de *accuracy* se mantiene en torno a 0.77. Por tanto, y dado que la diferencia no es muy significativa entre un valor C alto y pequeño, **podemos decantarnos por un valor en torno a 0.5 (moderado-bajo), así como una escala de 1**, aunque *caret* nos recomienda un valor C = 5, de forma que los márgenes de separación entre ambas clases es mayor, obteniendo un modelo más “general”.

De nuevo, **la decisión esta basada por patrones y no en valores puntuales**, por lo que empleando otra semilla y con otros valores training los parametros serán “aproximadamente” similares, aunque no iguales.

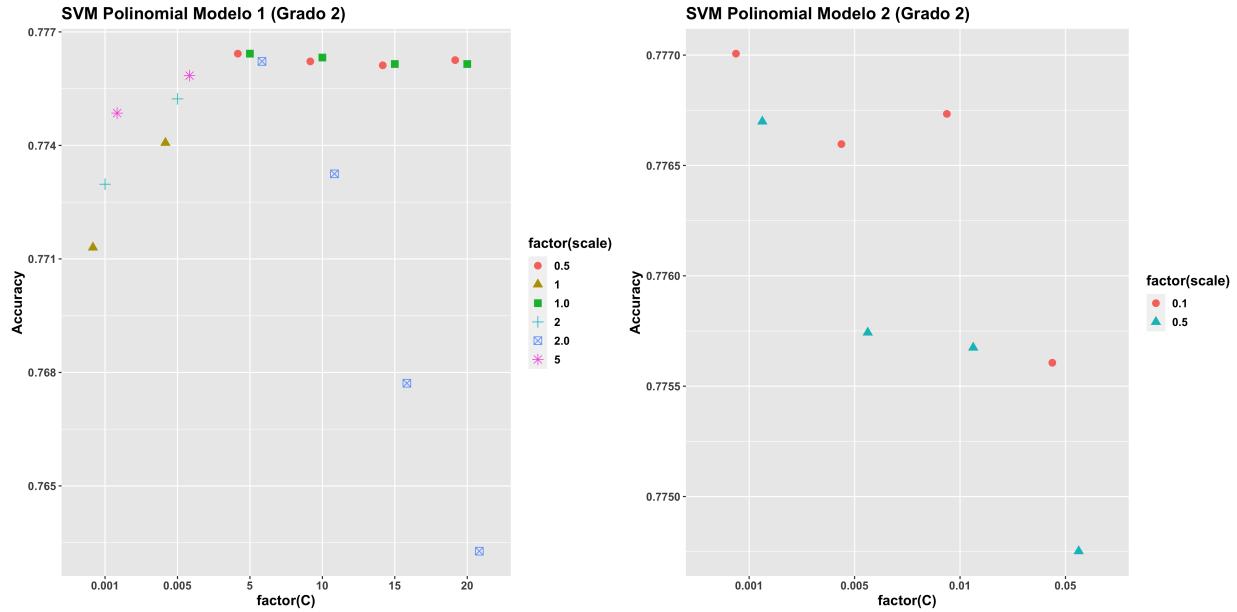


Figure 54: Distribución SVM Polinomial Modelos 1 y 2 (II)

Por tanto, dado que la diferencia entre un modelo SVM más ajustado ($C = 5$) y un modelo más general ($C = 0.5$) es ínfima.

2. Por otro lado, en relación con el segundo *set*, habiendo disminuido el valor C hasta 0.001-0.005, no ha hecho aumentar en gran medida el resultado. Por tanto, dado que con un valor C pequeño no solo obtenemos un mayor margen de separación (maniobrabilidad), sino además el mismo *accuracy* que el de un modelo más “estricto” en la separación, **nos decantamos por la opción de *caret***: C pequeño (0.01) y escala pequeña (0.1).

10.3 SVM RBF

Finalizando con el *kernel* radial, comenzando tuneando los dos parámetros principales: C y σ :

```
#-- Probamos con C entre 0.01 y 10
C_rbf      <- c(0.01,0.05,0.1,0.2,0.5,1,2,5,10)
sigma_rbf  <- c(0.1, 0.5, 1, 2, 5, 10)

## Ejemplo con el primer set variables
svm_rbf_1 <- cruzadaSVMbinRBF(data=surgical_dataset, vardep=target,
                                listconti = var_modelo1, listclass=c(""),
                                grupos=5,sinicio=1234,repe=5,C = C_rbf,
                                sigma = sigma_rbf, label = "Modelo 1")
```

Recomendado por *caret*

Modelo 1: $C = 1$; $\sigma = 5$; Accuracy: 0.8422943

Modelo 2: $C = 1$; $\sigma = 5$; Accuracy: 0.8726666

En base a los resultados devueltos por *caret*, para ambos *sets* de variables recomienda una constante C pequeña/moderada (1), así como un valor $\sigma = 5$. No obstante, y dado que se tratan de valores puntuales, analicemos los patrones existentes:

1. En el caso del primer *set*, los modelos con mayor *accuracy* presentan un valor σ alto (en torno a 5-10 aumenta hasta 0.83). En relación al valor C , prácticamente con cualquier valor el *accuracy* se sitúa en torno a 0.86-0.87, alcanzando su punto máximo en $C = 1$. Es decir, **no es**

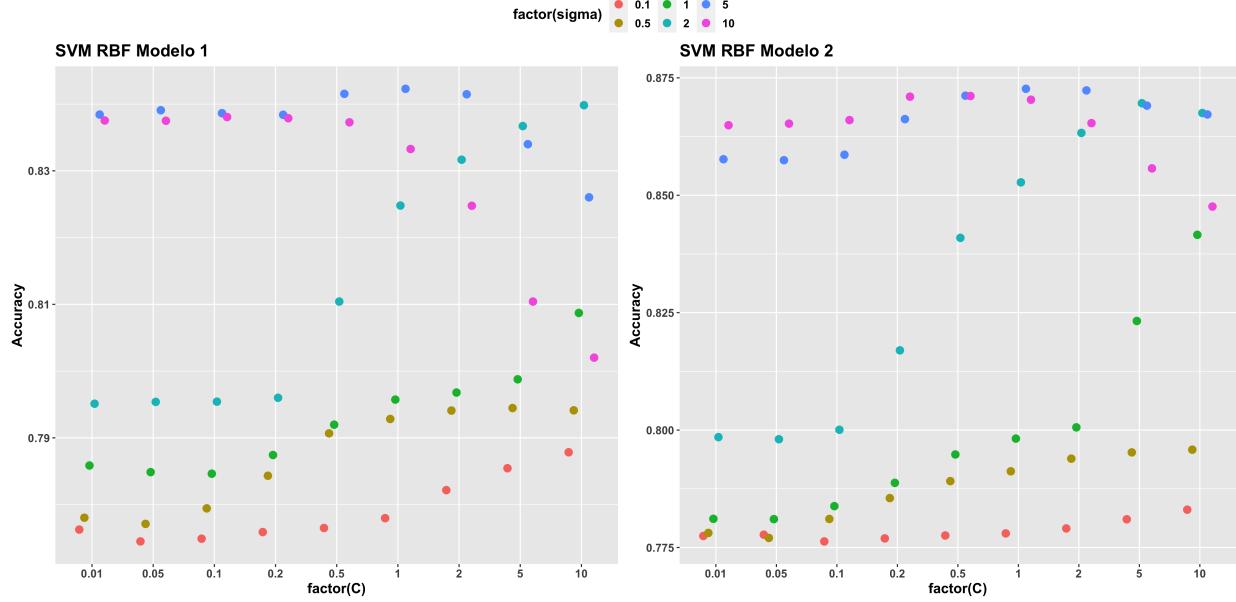


Figure 55: Distribución SVM RBF Modelos 1 y 2 (I)

necesario un valor C alto (márgenes de separación más pequeños), ya que a partir de $C = 2, 5$, el *accuracy* comienza a disminuir. Por otro lado, otro posible patrón de comportamiento sería con un valor *sigma* menor (2) y C alto (a partir de 5-10), donde comienza a aumentar el *accuracy*.

- En el caso del segundo modelo, el comportamiento es bastante similar: o bien escogemos valores *sigma* altos (5-10) y C no demasiado alto (menor a 1), o bien un valor *sigma* menor (2) y una constante C alta (en torno a 5, 10).

En ambos *sets*, podemos comprobar cómo evoluciona el modelo si aumentamos el valor de C con $\sigma = 2$, dado que a simple vista comienza a aumentar conforme incrementamos C (de 0.775 a 0.875):

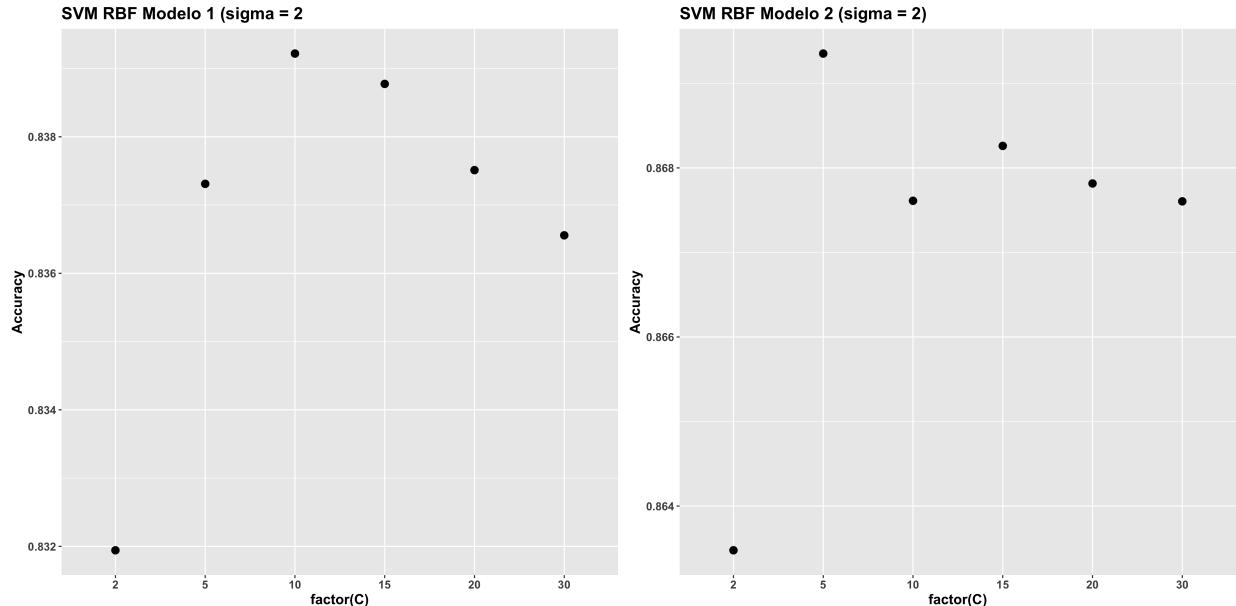


Figure 56: Distribución SVM RBF Modelos 1 y 2 (II)

Incluso aumentando C , se obtienen prácticamente los mismos resultados que con un σ y C menor.

En conclusión, para ambos *sets* de variables **nos decantamos por un valor σ alto (escogemos 5), y una constante C no demasiado alta, entre 0.5 y 1:**

Recomendado por caret

```
Modelo 1: C = 1 ; sigma = 5 ; Accuracy: 0.8422943
          C = 0.5 ; sigma = 5 ; Accuracy: 0.8415420
```

```
Modelo 2: C = 1 ; sigma = 5 ; Accuracy: 0.8726666
          C = 0.5 ; sigma = 5 ; Accuracy: 0.8711983
```

Dado que la diferencia entre $C = 1$ y $C = 0.5$ no es muy relevante (aunque *caret* recomienda 1), resulta indiferente escoger uno de ellos (por ejemplo, escogemos 0.5).

RESUMEN SVM Lineal:

- modelos 1 y 2: $C = 0.01$.

RESUMEN SVM Polinomial:

- modelo 1: $C = 0.01$; $scale = 0.1$.
- modelo 2: $C = 0.5$; $scale = 1$.

RESUMEN SVM RBF:

- modelo 1: $C = 0.5$; $\sigma = 5$.
- modelo 2: $C = 0.5$; $\sigma = 5$.

10.4 Comparación modelos SVM

Una vez elaborados los modelos, **empleando validación cruzada repetida de 5 grupos y 10 repeticiones, analizamos tanto la tasa de fallos como el AUC (en ambos *sets* de variables):**

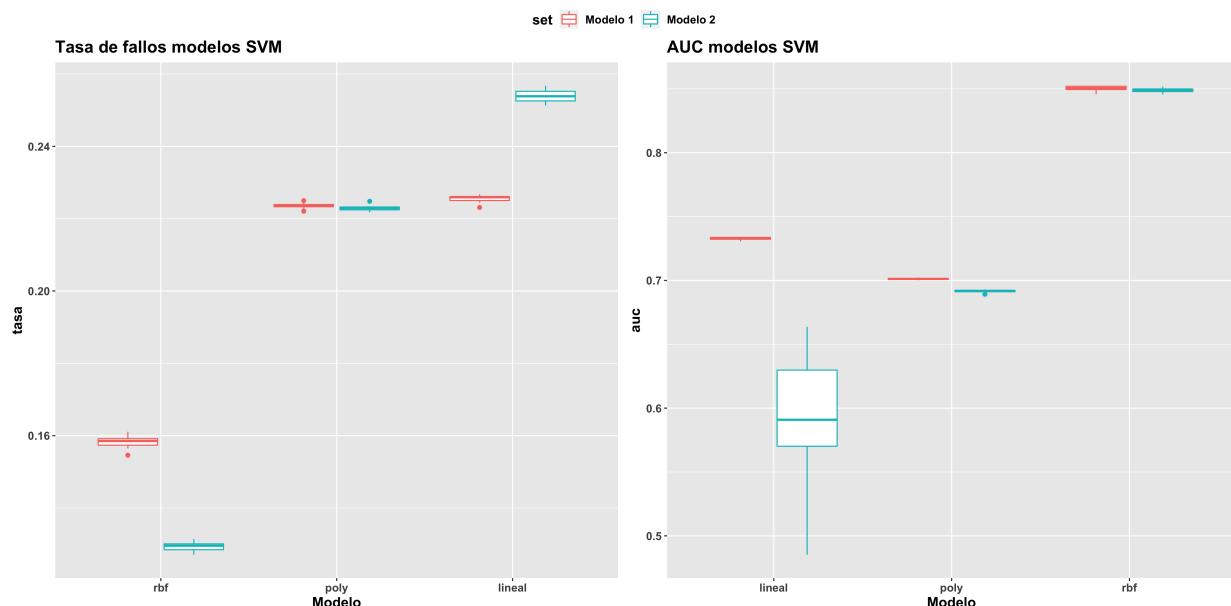


Figure 57: Comparación tasa de fallos y AUC modelos SVM

Sin duda alguna, **la mejor alternativa tanto en tasa de fallos como en AUC es el modelo SVM con kernel radial (RBF)**. De hecho, quisiera remarcar dos detalles fundamentales, relacionados con los *sets* de variables:

1. Por un lado, en relación con el *kernel* lineal, observamos una considerable diferencia entre el primer *set* con cinco variables (junto con *ccsMort30Rate*) y el segundo *set*, tanto en AUC, sesgo y varianza: incluyendo dicha variable (relación lineal), el modelo mejora hasta 0.74-0.75 de AUC, aproximadamente, además de reducir la tasa de error hasta 0.22 (a diferencia del primer *set*, que alcanza 0.26).
2. Por el contrario, con el *kernel* radial, añadir dicha variable perjudica al modelo en cuanto a la tasa de fallos (de 0.16 con cinco variables a 0.12-0.13 con tan solo cuatro), aunque en AUC se mantienen prácticamente idénticos.

Por otro lado, analicemos las predicciones obtenidas con el conjunto *test*:

SVM Lineal:

```
##           modelo precision sensibilidad especificidad valor_pred_pos
## 1 Modelo 1 (BIC TOP 5)      0.789      0.65844      0.80678      0.31733
## 2     Modelo 2 (top 4)      0.549      0.49980      0.76550      0.17310
##   valor_pred_neg
## 1            0.9454
## 2            0.7823
```

SVM Polinomial:

```
##           modelo precision sensibilidad especificidad valor_pred_pos
## 1 Modelo 1 (BIC TOP 5)      0.7940     0.71977      0.80205      0.28304
## 2     Modelo 2 (top 4)      0.7826     0.63188      0.80316      0.30453
##   valor_pred_neg
## 1            0.96345
## 2            0.94116
```

SVM RBF:

```
##           modelo precision sensibilidad especificidad valor_pred_pos
## 1 Modelo 1 (BIC TOP 5)      0.8517     0.8000      0.8622      0.5396
## 2     Modelo 2 (top 4)      0.8804     0.8897      0.8786      0.5935
##   valor_pred_neg
## 1            0.9553
## 2            0.9756
```

Al igual que en la comparación, con el conjunto *test* los modelos SVM RBF resultan ser claramente los ganadores, aunque con unos porcentajes inferiores con respecto a los modelos basados en árbol o gradient boosting:

Antes de continuar con *XGboost* y Ensamblado, debemos detenernos a analizar si realmente es necesario continuar con ambos *sets* de variables o no. A lo largo de la práctica (logística, redes, modelos basados en árbol, *boosting*, *svm*) se ha podido comprobar que la diferencia entre ambos candidatos, por tan solo una variable *input*, no ha sido especialmente relevante, pese a lo observado en la regresión logística donde si existía cierta diferencia entre ambos.

Por tanto, de cara a *XGboost* pero especialmente a los modelos de Ensamblado continuaremos con únicamente con el segundo *set* de variables *input* (4 variables), a excepción del modelo *svm* lineal, donde utilizaremos el modelo obtenido con el primer *set*.

11. XGboost

11.1 Tuneo de hiperparámetros

En primer lugar, debemos recordar los parámetros a tunear en un modelo *XGboost*:

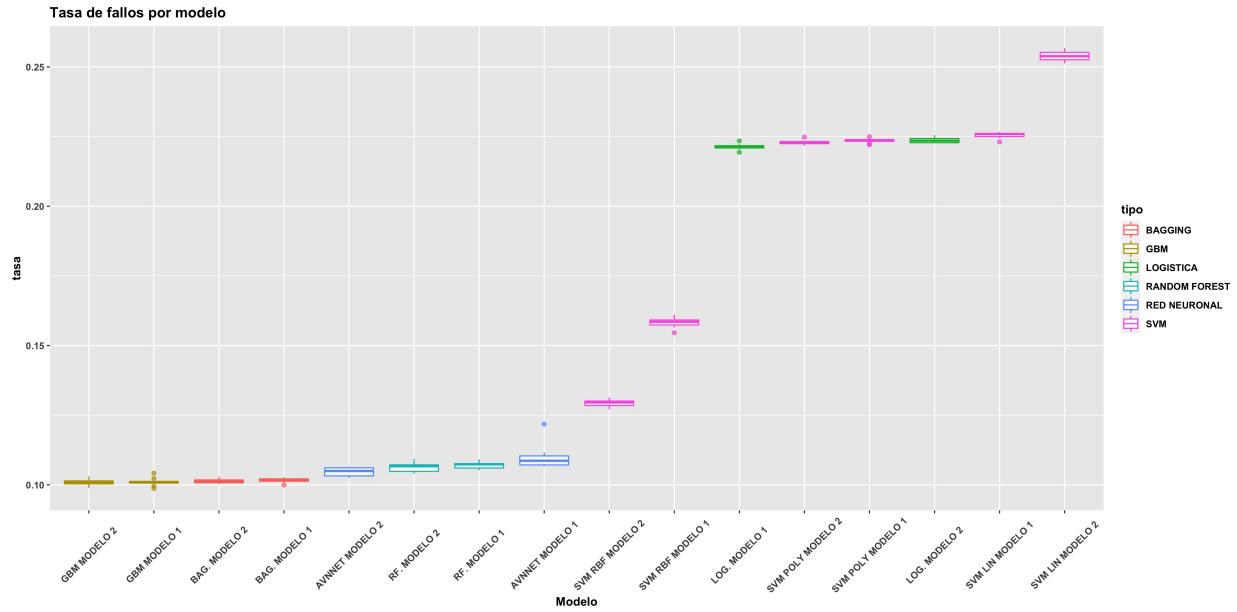


Figure 58: Comparacion tasa fallos log-avnnet-bagging-rf-gbm-svm

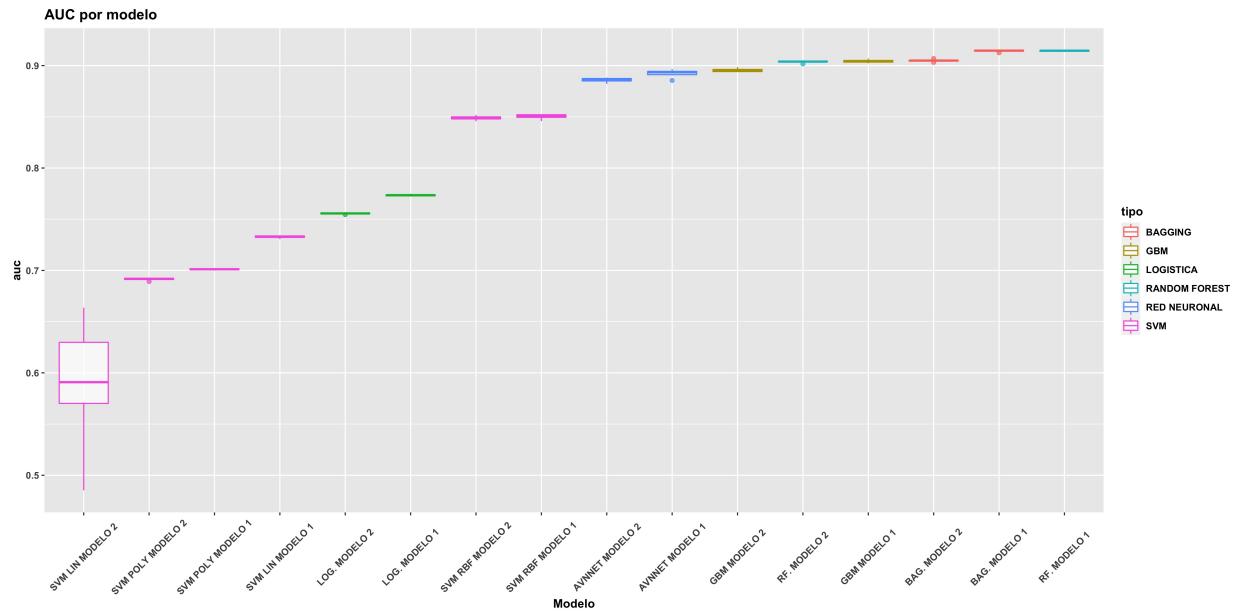


Figure 59: Comparacion AUC log-avnnet-bagging-rf-gbm-svm

1. *min_child_weight*: número de observaciones mínimas en el nodo final. Similar a *n.minobsinnode* de *gbm*.
2. *eta*: parámetro de regularización.
3. *rounds*: número de iteraciones.
4. *max_depth*: profundidad máxima de los árboles.
5. *gamma*: constante de regularización (lo mantenemos a 0).
6. *colsample_bytree*: porcentaje de sorteo de variables en cada árbol (lo mantenemos a 1).

7. *subsample*: sorteo de observaciones antes de cada árbol (lo mantenemos a 1).

```
xgbmgrid <- expand.grid(min_child_weight=c(5,10,20),
                         eta=c(0.1,0.05,0.03,0.01,0.001),
                         nrounds=c(100,500,1000,5000),
                         max_depth=6,gamma=0,
                         colsample_bytree=1,subsample=1)

##-- Probamos al comienzo con 5 repeticiones
set.seed(1234)
control<-trainControl(method = "repeatedcv",number=5,repeats = 5,
                      savePredictions = "all",classProbs=TRUE)

xgbm_ <- train(formula_modelo2,data=surgical_dataset,
                 method="xgbTree",trControl=control,
                 tuneGrid=xgbmgrid,verbose=FALSE)
```

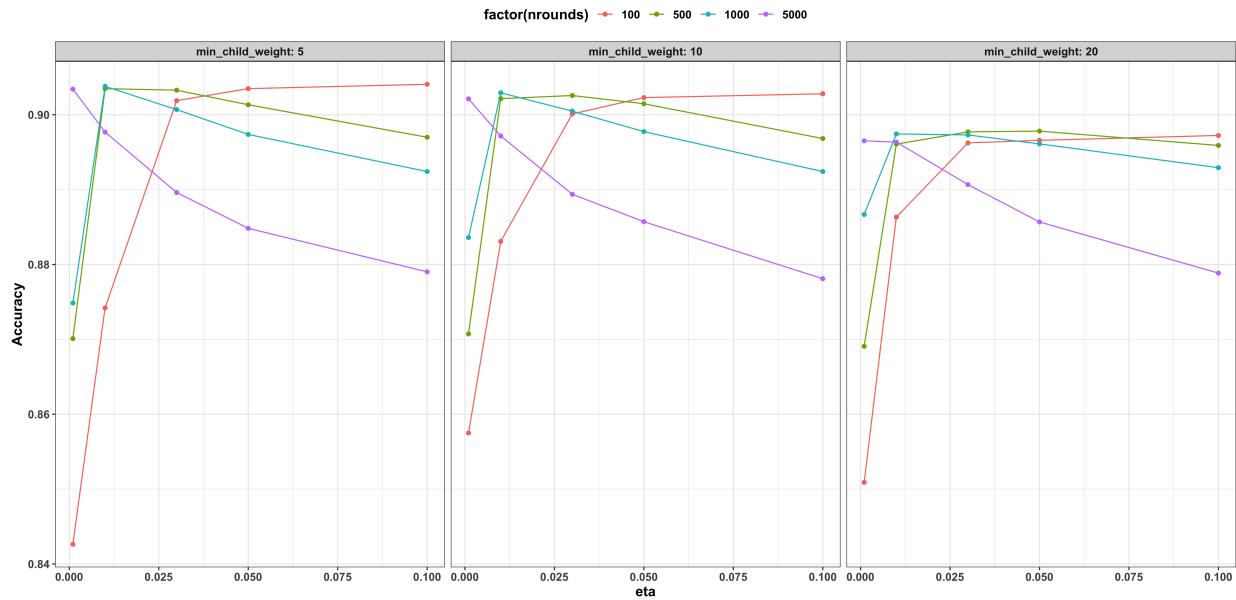


Figure 60: Tuneo hiperparámetros XGboost

De los gráficos anteriores, se deduce que se debe tomar un valor *eta* alto (en torno a 0.1) y pocas iteraciones, alrededor de 100. Además, si nos fijamos en la escala de los ejes, no existe demasiada diferencia entre un valor *min_child_weight* bajo (5) y alto (20), tan solo de apenas unas décimas (0.90 y 0.89, respectivamente). Por tanto, escogemos un valor de 20 al ser las diferencias tan pequeñas, permitiendo con ello modelos más simples y con menos sobreajuste.

11.2 Estudio *Early Stopping*

A continuación, probamos a fijar algunos parámetros para analizar cómo evoluciona el modelo en función del número de iteraciones:

Como podemos comprobar en el gráfico anterior, 100 iteraciones es un valor óptimo, en términos generales.

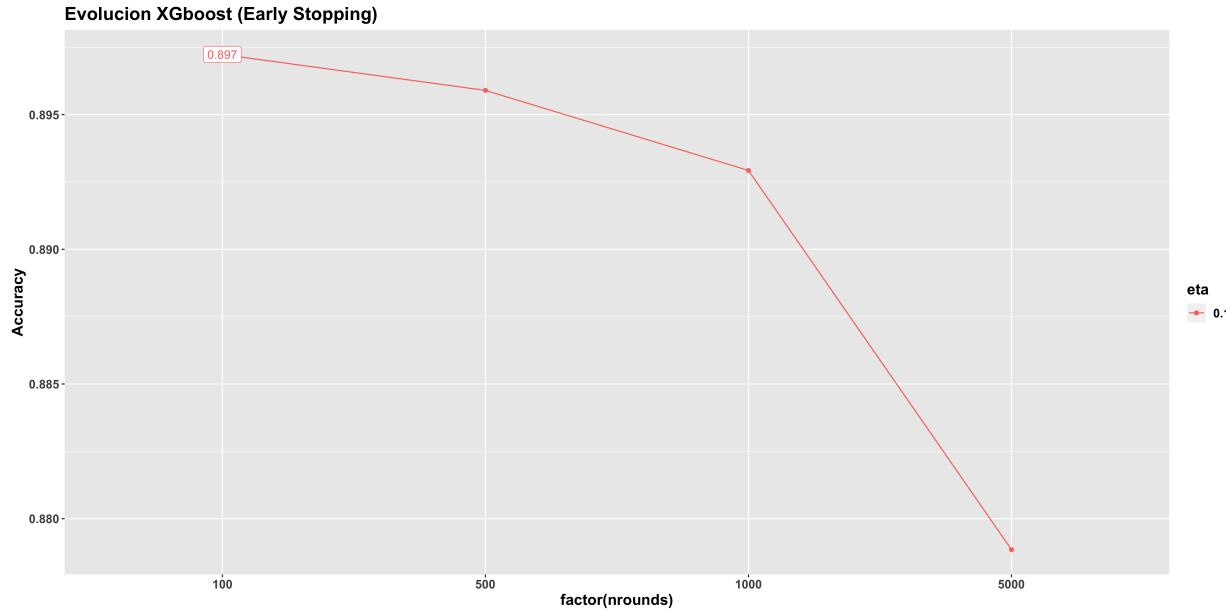


Figure 61: Evolucion Early Stopping XGboost

11.3 Tuneo *max_depth*

Hasta el momento, se ha considerado una profundidad máxima de 6 en cada árbol del modelo. No obstante, **analicemos la evolución del modelo, tanto en sesgo como en varianza, al variar dicho parámetro:**

```
##-- Tuneamos max_depth (minimo 1, maximo 20)
xgbm_mo <- cruzadaxgbmbin(data=surgical_dataset, vardep=target,
                           listconti=var_modelo2, listclass=c(""),
                           grupos=5, sinicio=1234, repe=5,
                           nrounds=100, eta=0.01, min_child_weight=20,
                           gamma=0, colsample_bytree=1, subsample=1,
                           max_depth=c(1,3,6,10,15,20)
                           )
```

Analizando los resultados, **una profundidad máxima de 6 parece ser una buena opción, ya que a partir de dicho valor tanto el sesgo como el AUC se estabilizan.**

11.4 Tuneo *subsample*

Como último parámetro a tunear, probamos con el parámetro *subsample* con el que sortear el número de observaciones, especialmente para comprobar cómo afecta a la varianza, más que al sesog. Concretamente, y del mismo modo que lo realizado con *gradient boosting*, **probaremos sorteando el 50 % de las observaciones:**

Nota: para observar mejor el efecto del sorteo de observaciones, probamos tanto con 5 como 10 repeticiones.

En términos de varianza, al ser datos tan secillos no se observa mejora alguna. No obstante, **no hay demasiada diferencia entre sortear el 100 % de las observaciones y tan solo la mitad.** Por tanto, mantenemos el parámetro *subsample* a 1 (valor por defecto en muchos paquetes *XGboost*).

RESUMEN *XGboost*:

1. *nrounds* = 100
2. *eta* = 0.1
3. *min_child_weight* = 20

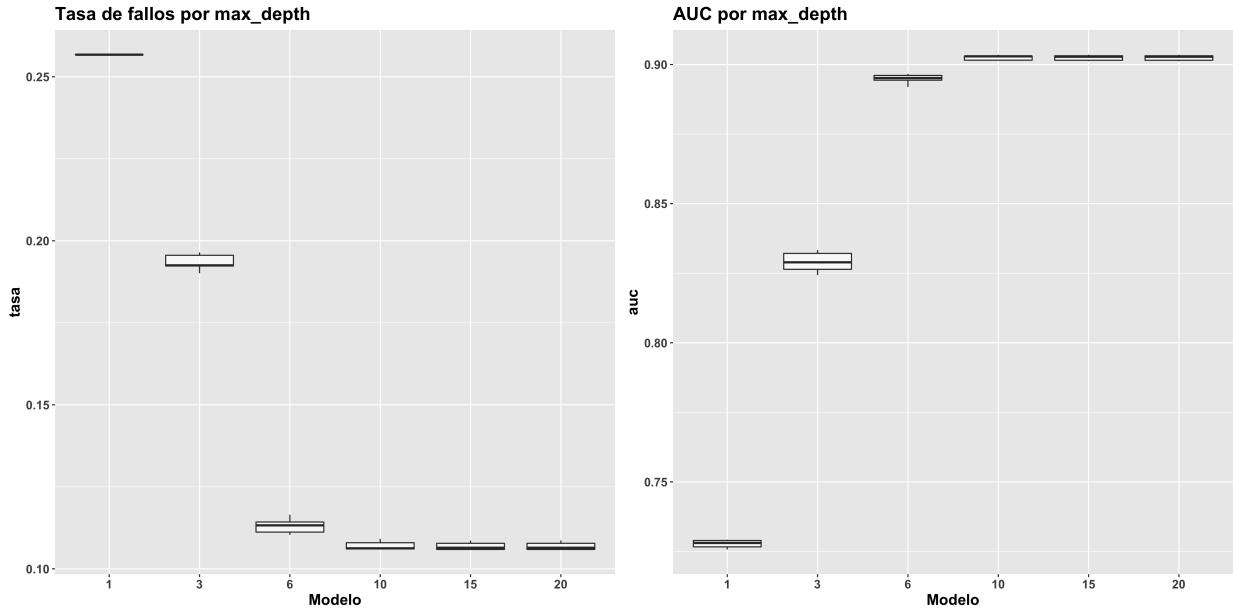


Figure 62: Tasa de fallos y AUC en función de max depth

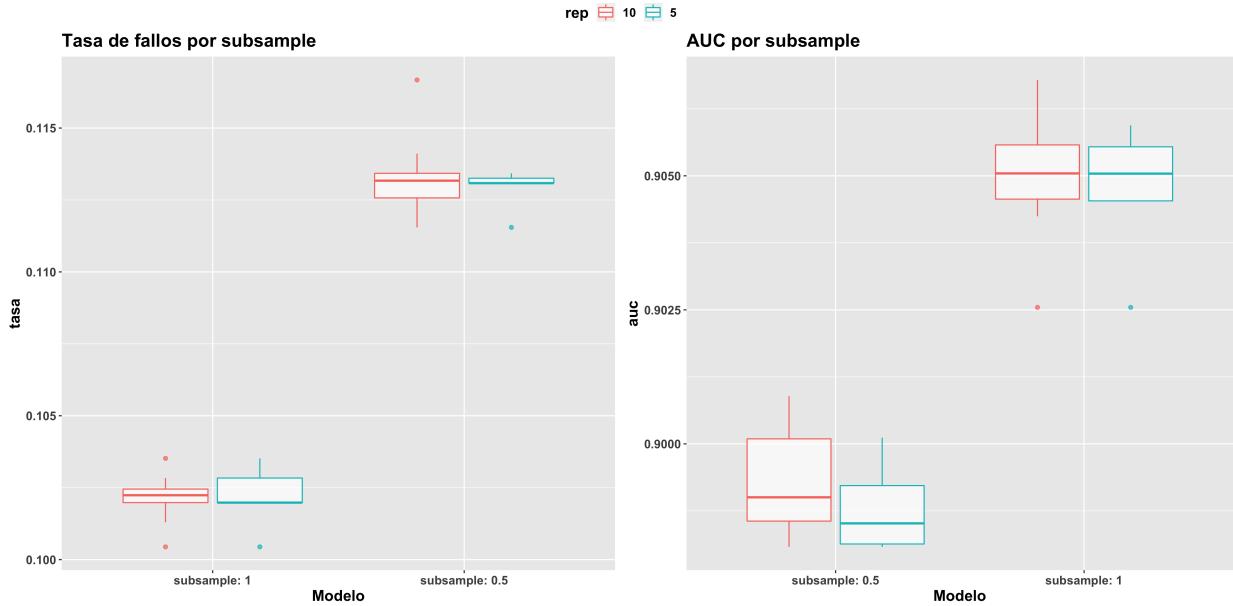


Figure 63: Tasa de fallos y AUC en función de subsample

4. `colsample_bytree = 1`,
5. `subsample = 1`
6. `max_depth = 6`

11.5 Comparación final

Una vez elaborado el modelo, realizamos una primera predicción con los datos *test*:

```
##           modelo precision sensibilidad especificidad valor_pred_pos
## 1 Modelo 2 (top 4)      0.9091        0.9333       0.9037        0.684
```

```
##     valor_pred_neg
## 1          0.9838
```

Del mismo modo, se mantiene en la “línea” de lo obtenido de otros modelos no lineales como *random forest*, *bagging* o *gradient boosting* (aunque si mejora los resultados con respecto a la *svm radial*).

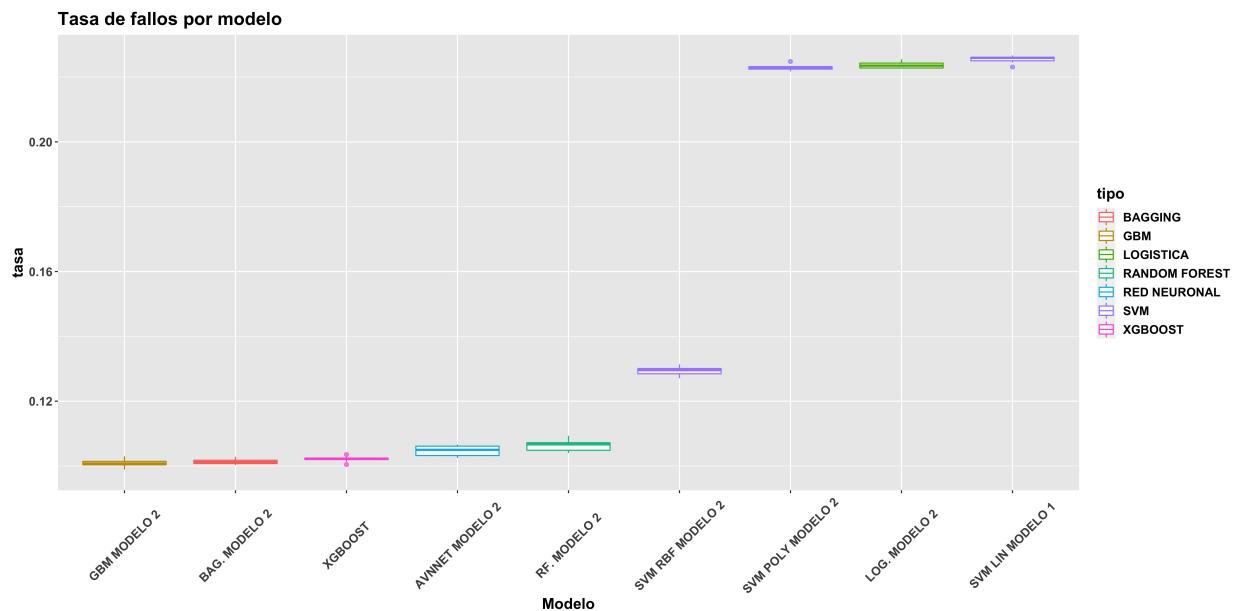


Figure 64: Comparacion tasa fallos log-avnnet-bagging-rf-gbm-svm-xgboost

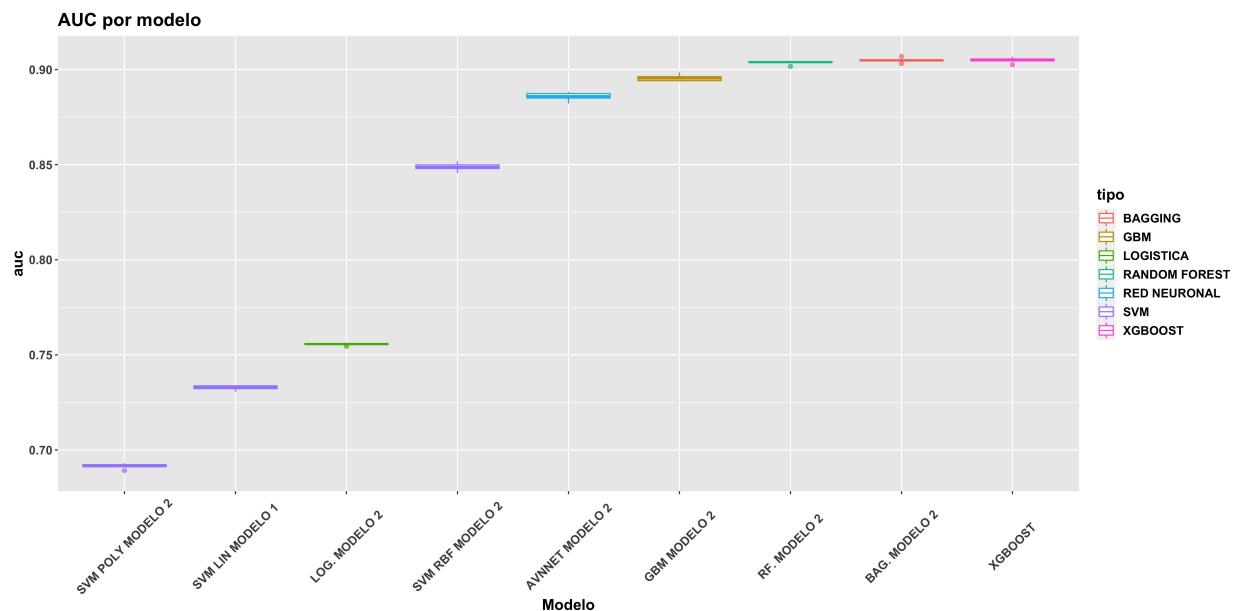


Figure 65: Comparacion AUC log-avnnet-bagging-rf-gbm-svm-xgboost

12. Ensamblado

12.1 Correlación entre los modelos

Inicialmente, de forma previa al ensamblado analicemos la correlación entre cada uno de los modelos, entrenados mediante validación cruzada repetida del mismo modo que lo hecho hasta el momento (5 grupos y 10 repeticiones):

```
#-- Correlacion entre los diferentes modelos
mat <- unigraf[,modelos]
matrizcorr <- cor(mat)
corrplot(matrizcorr, method="color",
          type="upper", order="hclust",
          addCoef.col = "black",
          tl.col="black", tl.srt=45,
          p.mat = matrizcorr, sig.level = 0.99,
          diag=FALSE
)
```

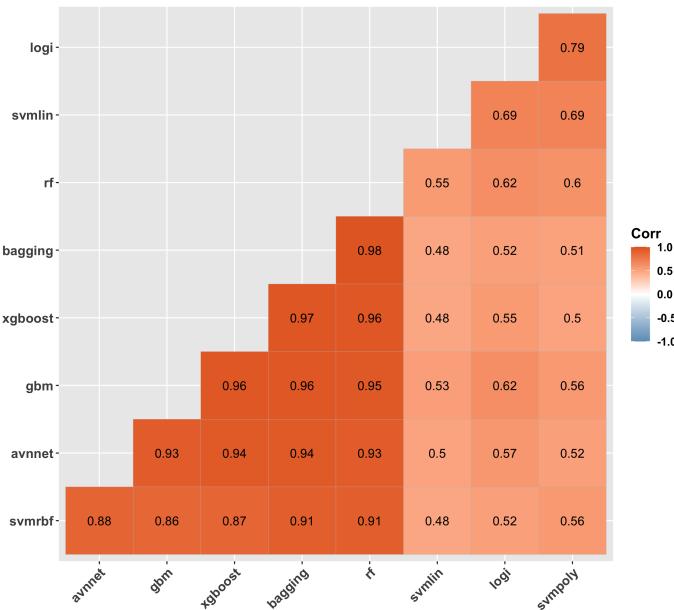


Figure 66: Correlación entre modelos

Podemos comprobar cómo las diferentes familias tienen alta correlación entre sí, detectando dos grandes grupos de alta correlación:

1. **Modelos no lineales basados en árbol, como *bagging* o *random forest*, junto con los modelos basados en *boosting* como *gbm* y *xgboost*, así como la red y el *svm* con *kernel radial*.** En ninguno de los casos la correlación disminuye de 0.8.
2. **El modelo logístico y las *svm* con *kernel* lineal y polinomial.** En cualquiera de los casos, la correlación no disminye de 0.6.

Como era de esperar, se encuentra generalmente una alta correlación entre modelos no lineales y lineales por separado. Por tanto, ¿Cómo influye de cara al ensamblado? Por lo general, cuanto menor sea la correlación entre los clasificadores, más se reducirá el sesgo en los nuevos ensamblados, o en el peor de los casos se mantendrá o no subirá demasiado.

12.2 Ensamblado con dos clasificadores

Por tanto, el objetivo será unir clasificadores que presenten un sesgo lo suficientemente bajo y similar, con poca correlación entre ellos. De hecho, en base al gráfico anterior, existen varias alternativas, uniendo principalmente modelos lineales y no lineales entre sí:

1. *svm* lineal, polinomial o logística con *Random forest*, *bagging*, *gradient boosting*, *Xgboost*, *avnnet* e incluso *svm* con *kernel radial*, **modelos cuya correlación es moderada, en torno a 0.5-0.6, aproximadamente.**

Sin embargo, que el sesgo mejore o no es algo que **no se puede conocer a priori**, por lo que es necesario realizar pruebas empíricas. Como consecuencia, realizamos una primera aproximación, comenzando con un ensamblado entre dos modelos, realizando todas las posibles combinaciones entre sí (mediante el promedio de resultados):

```
nipredi[, paste0("en-", i)] <- (unipredi[, modelo] + unipredi[, modelo_aux]) / 2
```

Además, dado que la regresión logística no obtuvo buenos resultados, remarcamos con un color diferente cualquier ensamblado en el que intervenga dicho modelo, observando con ello si mejora o no:

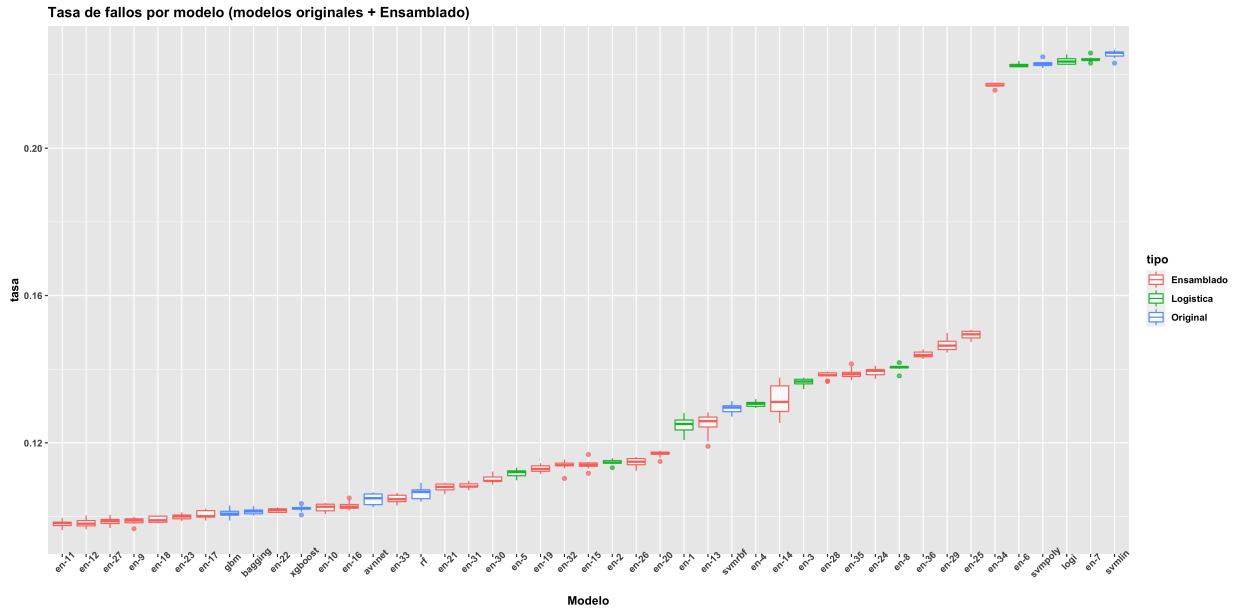


Figure 67: Comparacion tasa fallos modelos + ensamblado

En primer lugar, cabe destacar que en conjunto con otros modelos, **la regresión logística mejora tanto en tasa de fallos como en AUC**:

1. *en-8*: logística + *svm* polinomial
2. *en-1*: logística + *avnnet*
3. *en-4*: logística + *gbm*
4. *en-3*: logística + *Random Forest*
5. *en-2*: logística + *bagging*
6. *en-5*: logística + *Xgboost*

Con ellos, la tasa de fallos se reduce por debajo de 0.20 y el AUC aumenta hasta 0.85. Sin embargo, pese a su mejoría continua siendo preferible modelos originales como *random forest*, *bagging* o *gbm*, dado al juntarlo con

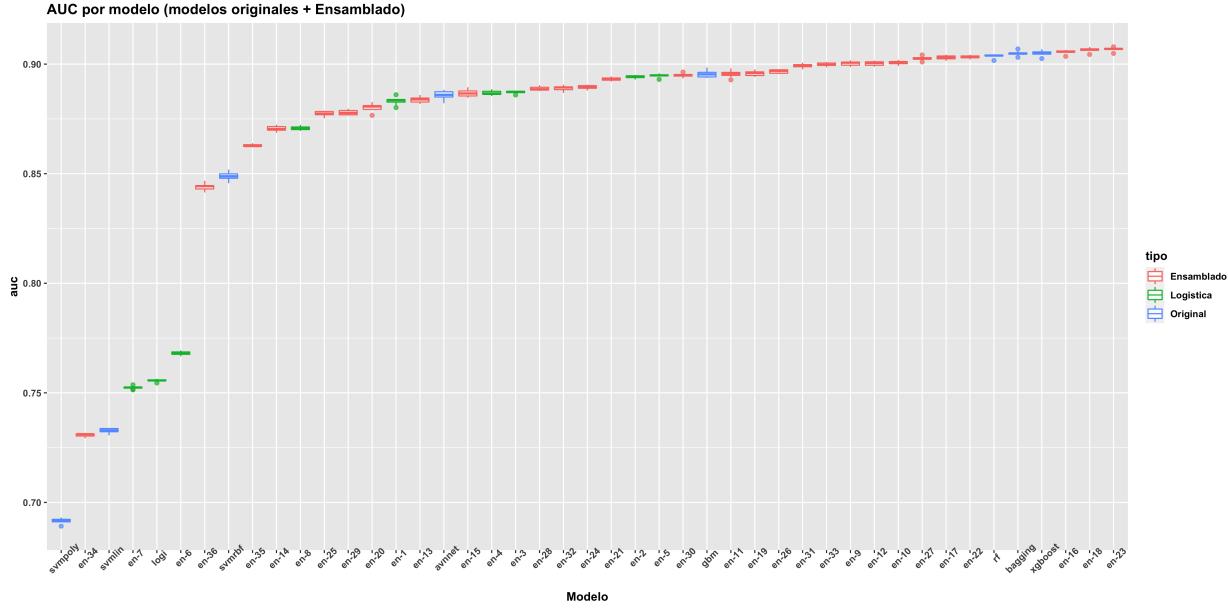


Figure 68: Comparacion AUC modelos + ensamblado

un modelo con peores resultados (como es el caso de la logística), provocan que el resultado mejore aunque sigue siendo preferible los modelos originales.

Por otro lado, debemos destacar los mejores modelos de ensamblado, tanto en tasa de fallos como en AUC:

1. Con menor tasa de fallos:

- *en-11: avnnnet + gbm*
- *en-12: avnnnet + Xgboost*
- *en-27: gbm + Xgboost*

2. Con mayor AUC:

- *en-23: Random Forest + Xgboost*
- *en-18: bagging + Xgboost*
- *en-16: bagging + Random Forest*

Llama la atención como, tanto en tasa de fallos como en AUC, los mejores ensamblados corresponden con modelos similares (estadísticamente), pero con alta correlación entre si, consiguiendo mejorar sus correspondientes tasas de fallos y AUC, aunque muy ligeramente, dada la escala de los ejes. Por ejemplo, uniendo *avnnnet* y *gbm* (*en-11*) solo conseguimos mejorar la tasas de fallos en 0.005 (lo cual incluso puede ser debido a la estructura de remuestreo). Por otro lado, modelos como *bagging* + *XGboost* apenas mejoran el AUC (en torno a milésimas).

De hecho, ¿Qué sucede con el resto de ensamblados con poca correlación? Si filtramos del gráfico anterior, descartando ensamblados con modelos de alta correlación (> 0.8):

En cualquiera de los casos obtenidos, sigue siendo preferible los modelos originales. Esto es debido a que el *dataset* obtiene mejores resultados con la misma familia de modelos, es decir, mientras que con modelos lineales como logística o *svm* lineal apenas alcanzan el 77-78 % de *Accuracy*, con modelos no lineales como redes pero en especial árboles y *boosting*, la precisión llega a aumentar hasta rozar el 90 % de precisión. Por ello, **los únicos modelos con un sesgo especialmente bajo son justamente aquellos con alta correlación entre sí**, por lo que si juntamos un modelo lineal con otro no lineal (por ejemplo, *avnnnet* con logística), hará que el modelo mejore con respecto a la logística, pero seguirá siendo preferible un modelo no lineal.

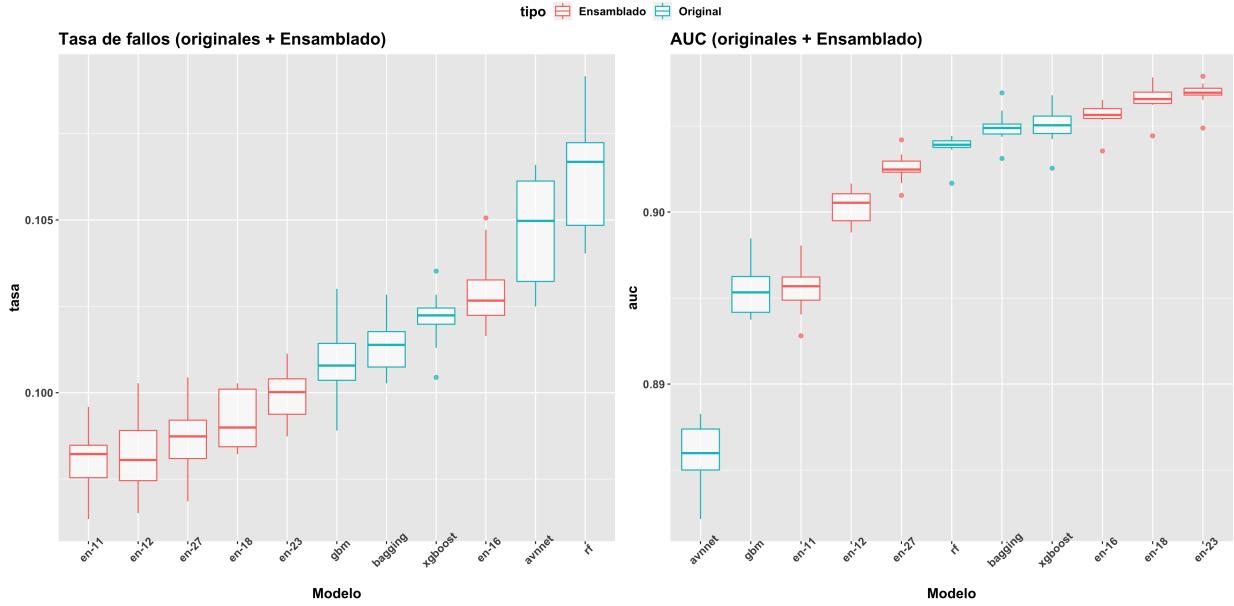


Figure 69: Comparacion tasa fallos y AUC modelos + ensamblado (II)

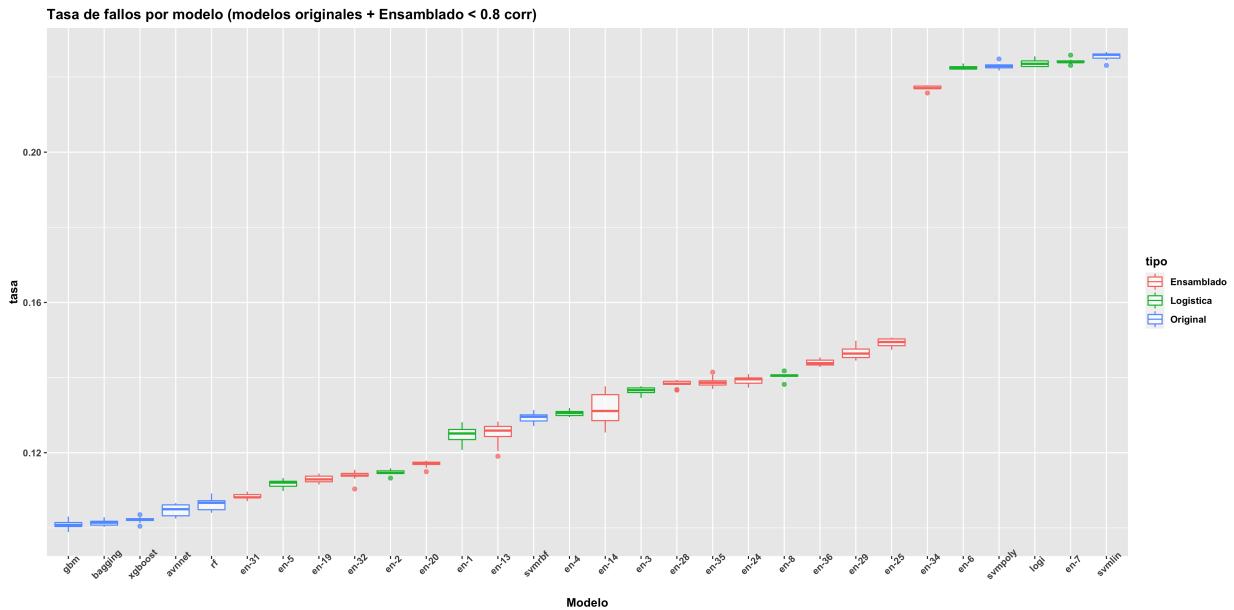


Figure 70: Comparacion tasa fallos modelos + ensamblado (III)

A simple vista, **uniendo dos clasificadores no se consigue mejorar significativamente los resultados del modelo**, en especial cuando los mejores modelos presentan una alta correlación entre sí, por lo que la mejoría es poco o nada apreciable.

12.3 Ensamblado con tres clasificadores

No obstante, podríamos probar a los mejores clasificadores en ensamblados de tres modelos, concretamente aquellos con alta correlación, es decir, *random forest*, *svm rbf*, *avnnet*, *gbm* y *xgboost* (*bagging* lo descartamos ya que con *random forest* se obtienen prácticamente los mismo resultados):

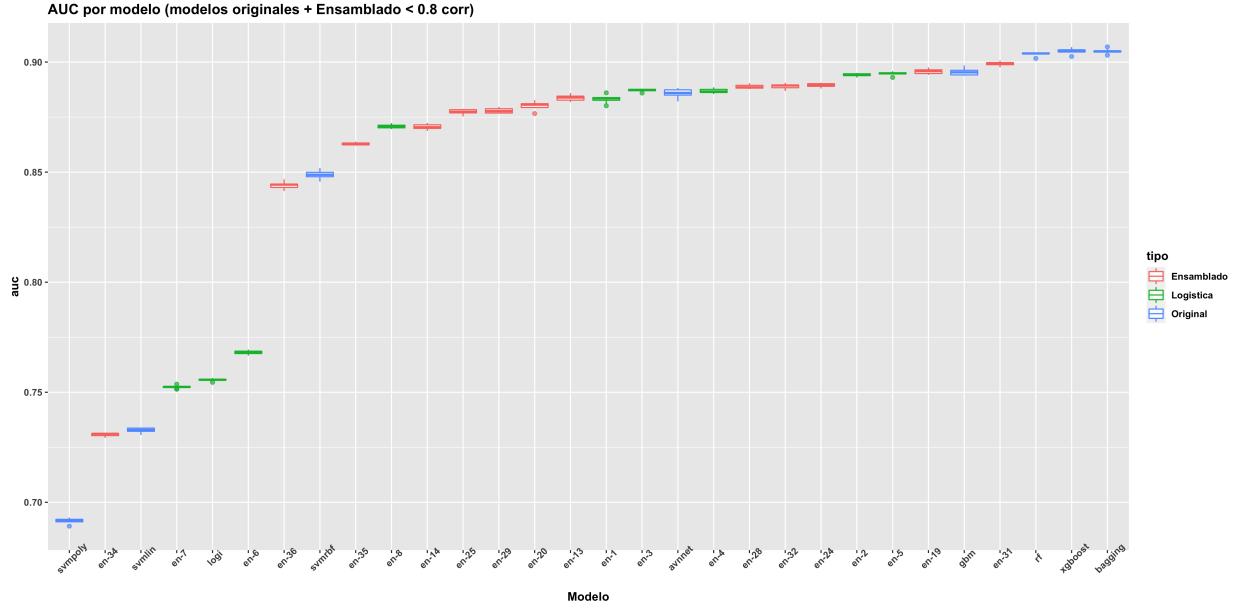


Figure 71: Comparacion AUC modelos + ensamblado (III)

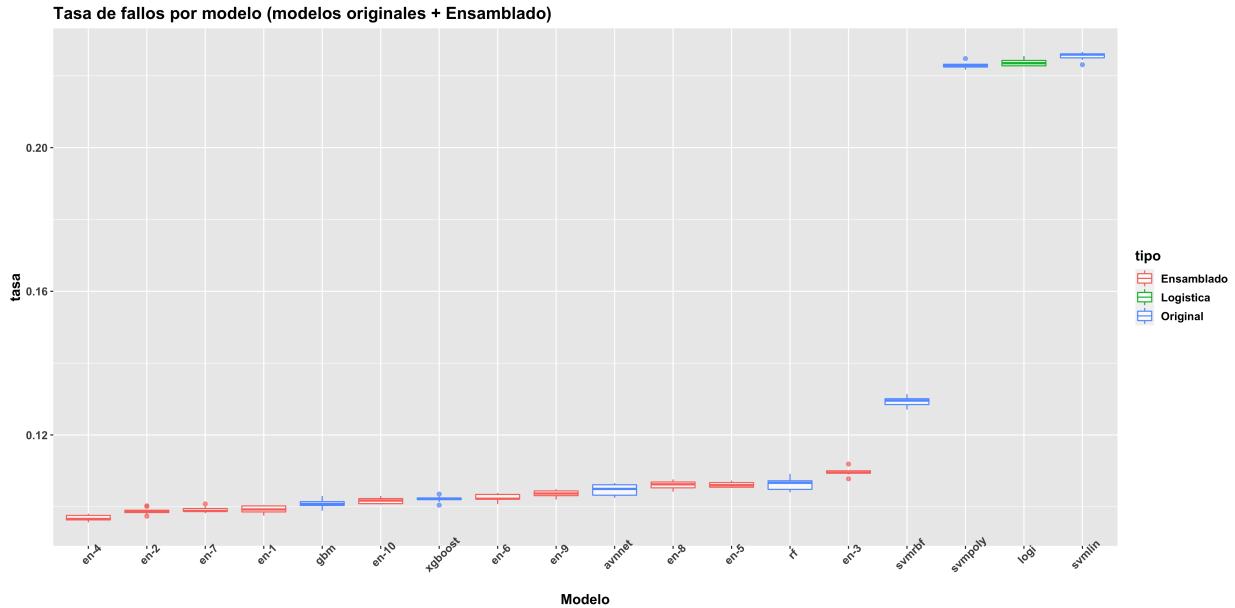


Figure 72: Comparacion tasa fallos modelos + ensamblado (IV)

Pese a añadir tres clasificadores, la **mejora continua siendo insignificante**.

12.4 Conclusiones ensamblado

A la vista de los resultados obtenidos, ¿Merece la pena aplicar ensamblado? La respuesta es no, principalmente por dos motivos:

1. Los modelos originales que obtienen mejores resultados corresponden con aquellos con alta correlación, esto es, modelos de árbol, red y *boosting*.
2. Incluso ensamblando dichos modelos, la mejoría es poco o nada relevante.

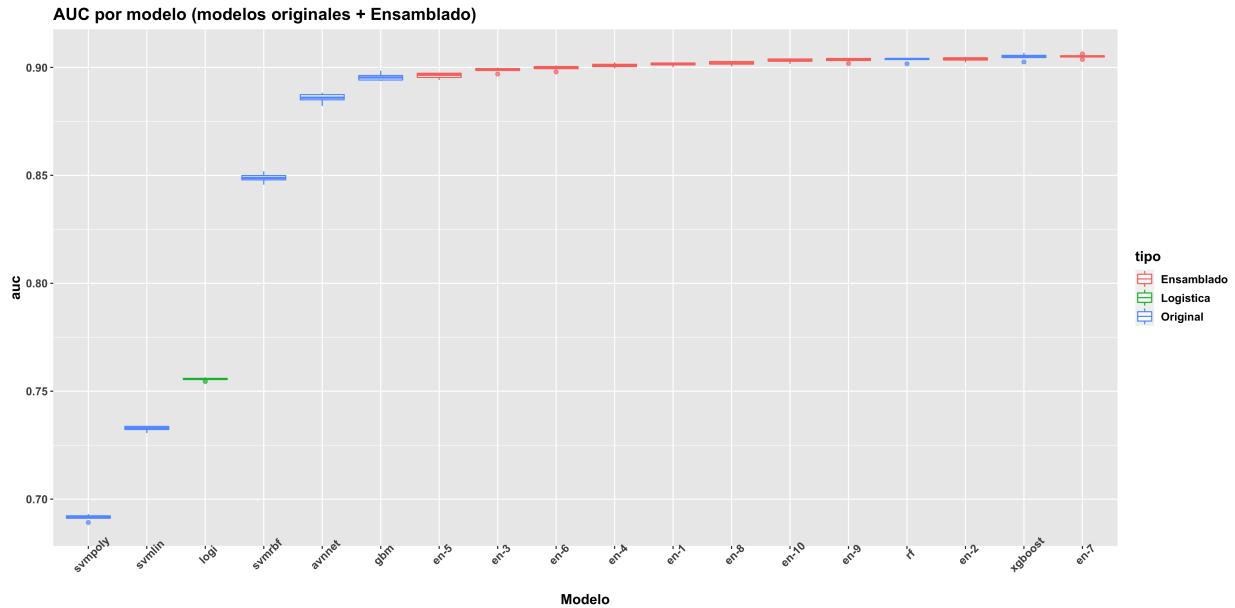


Figure 73: Comparacion AUC modelos + ensamblado (IV)

Por tanto, está claro que los modelos no lineales con alta correlación son los algoritmos dominante para este conjunto de datos, por lo que no merece la pena el ensamblado.

13. Comparación con *h2o*

Una vez elaborados todos los modelos, Es momento de preguntarnos ¿Se cumplen las expectativas obtenidas con *h2o*? Es decir, con *autoML* obtuvimos inicialmente (como mejor modelo) *gradient boosting* con los siguientes valores AUC:

```
##                   modelo      auc
## 1 GBM_5_AutoML_set1_variables 0.9211350
## 2 GBM_5_AutoML_set2_variables 0.9096419
```

Si los comparamos con el AUC de los mejores modelos obtenidos:

```
##                   modelo auc_medio
## 1      bagging set 1  0.9144
## 2      bagging set 2  0.9049
## 3 Random Forest set 1  0.9144
## 4 Random Forest set 2  0.9038
## 5      gbm set 1    0.9043
## 6      gbm set 2    0.8954
```

En relación a *gradient boosting*, las diferencias son pequeñas entre ambos paquetes (de 0.92 a 0.90 con el primer *set* y de 0.90 a 0.89 en el segundo), lo cual puede ser debido a la estructura del remuestreo o la optimización de los algoritmos en los diferentes paquetes. Por tanto, lo obtenido por *h2o* se aproxima a los resultados de *caret*. No obstante, también se han obtenido buenos resultados con diferentes modelos como es el caso de *random forest* o *bagging*, cuyos valores AUC son prácticamente iguales.

14. Probando con el *dataset* completo

A continuación, y utilizando únicamente el segundo *set* de variables (salvo con el modelo *svm* lineal, con el que empleamos el primer *set*), probamos los mismos algoritmos “tuneados” con el conjunto de datos original, con el objetivo de comprobar si el orden entre algoritmos se conserva:

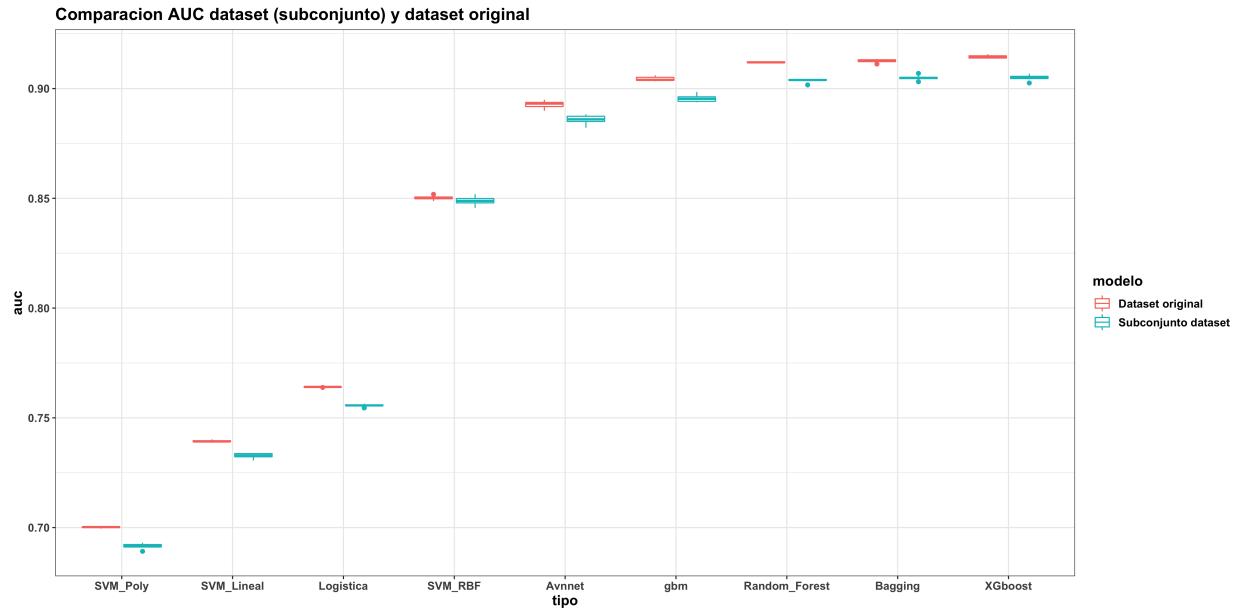


Figure 74: Comparacion tasa fallos dataset original y subconjunto

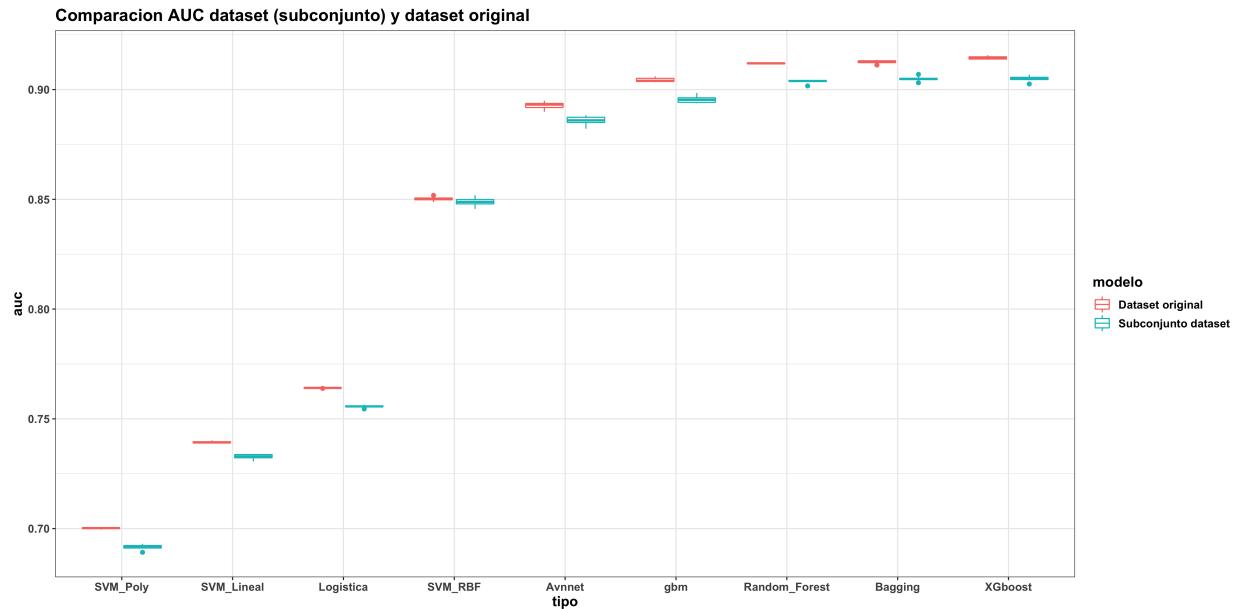


Figure 75: Comparacion AUC dataset original y subconjunto

De forma general, el orden de los modelos, tanto con el *dataset* original como con el subconjunto utilizado durante la práctica, se conserva, a excepción de la tasa de fallos, siendo *Xgboost* el mejor modelo con el *dataset* original, mientras que con el subconjunto la menor tasa se obtiene con *gbm*; aunque la diferencia no sea muy relevante entre ambos.

15. Aumento del número de grupos y repeticiones

Tras comprobar que el orden de los modelos se conserva con el conjunto original, nos queda una última prueba: con los mejores modelos (*bagging*, *gbm*, *random forest* y *XGboost*), realizar una última comparación bajo un número alto y diferente de semillas y un número mayor de grupos de CV. De este modo, lo que se pretende buscar es que el orden entre las “cajas” y la varianza de los modelos se mantenga similar a lo obtenido originalmente.

Dado que durante la práctica se ha trabajado con 5 grupos y un máximo de 10 repeticiones, aumentamos a 10 grupos y 20 repeticiones, además de aumentar la semilla inicial a 123456:

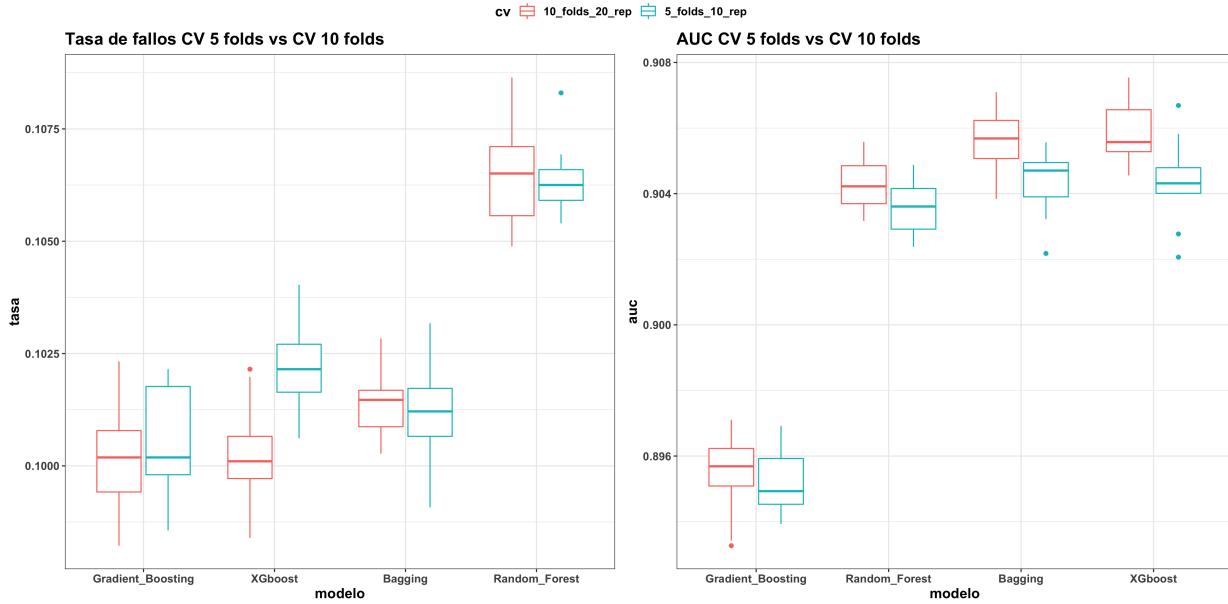


Figure 76: Comparacion AUC y tasa de fallos con 5 y 10 grupos

A la vista de los resultados, tanto el orden como la varianza se mantiene más o menos similar al aumentar el número de grupos, aunque en algunas “cajas” observemos una mayor varianza con 10 grupos, como es el caso de *Random Forest* o *Bagging* en AUC. No obstante, y dada la escala de los ejes, la diferencia de la varianza es muy pequeña, por lo que consideramos que se mantienen prácticamente idénticos.

16. Variación del punto de corte

Como última sección, con los mejores modelos obtenidos analizaremos los valores especificidad-sensibilidad con el punto de corte actual (0.5), además de comprobar como varían dichos algoritmos bajo diferentes puntos.

En primer lugar, de todos los mejores modelos obtenidos, ¿Por cual o cuales nos decantamos? Echemos un vistazo al AUC y tasa de fallos de los cuatro más importantes: *gradient_boosting*, *bagging*, *Random Forest* y *Xgboost*.

De todos los modelos presentes, y dejándonos guiar por los gráficos, diríamos que el mejor en cuanto a AUC y tasa de fallos serían modelos basados en *boosting* como *gbm* y *XGboost*, pues no solo presentan bajas tasas de error, sino que además (y en especial con *XGboost*, un alto valor en el área bajo la curva).

De nuevo, la escala del gráfico puede llevar a engaño, y es que la diferencia entre los modelos basados en árboles y *boosting* es ínfima. A modo de ejemplo, entre la tasa de fallos de *gbm* y un modelo *Random Forest* es de tan solo 0.007. En relación a la variancia, sucede lo mismo: aunque *Random Forest* aperente tener

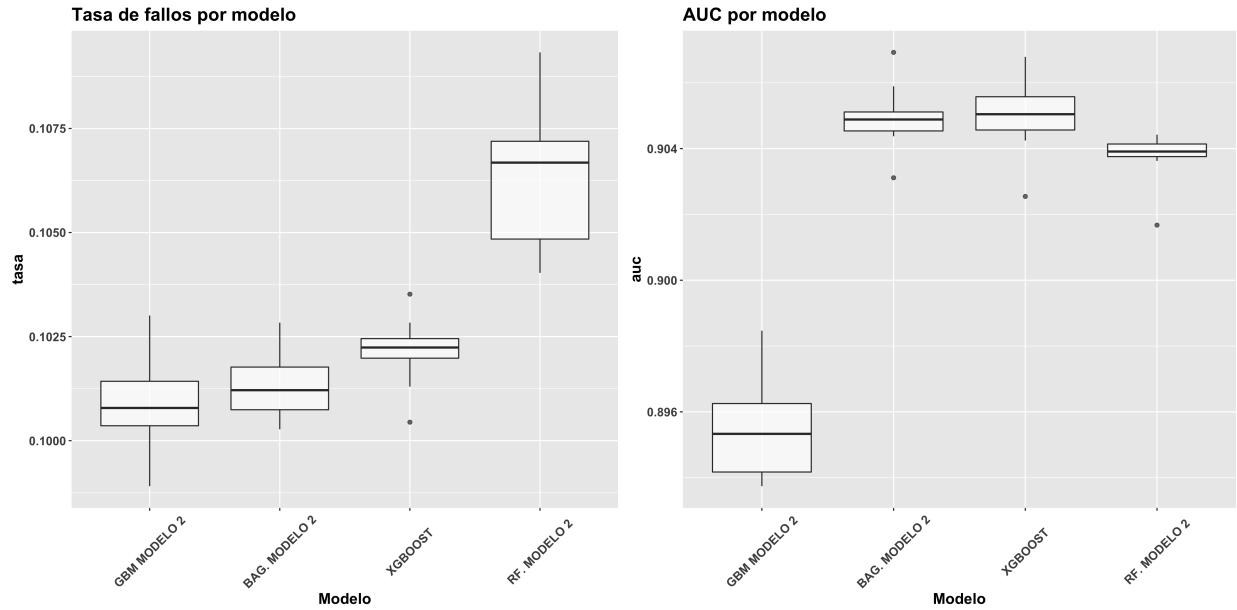


Figure 77: Comparación modelos finales

la mayor variabilidad en cuanto a tasa de fallos, lo cierto es que la escala del eje puede conducir a engaño, puesto que la amplitud es de apenas