EST-25134: Aprendizaje Estadístico

Profesor: Alfredo Garbuno Iñigo — Primavera, 2023 — Ensambles y *Boosting*.
Objetivo: En esta sección estudiaremos otra estrategia para combinar modelos en uno sólo. La estrategia que estudiaremos en esta sección es un mecanismo altamente flexible para regresión y clasificación y actualmente representa el estado del arte para resolver problemas de predicción con datos tabulares.
Lectura recomendada: Capítulo 10 del libro Hastie et al. [2]. También el capítulo 5 de Greenwell [1].

1. INTRODUCCIÓN

Boosting es otra estrategia de agregación de modelos. Es una herramienta bastante general que puede ser utilizada para problemas de regresión o clasificación [5]. La estrategia de `boosting` es una `estrategia secuencial` que busca mejorar la capacidad predictiva lograda anteriormente.

Usualmente con *boosting* utilizamos árboles pequeños (sesgo alto), al contrario de bosques aleatorios donde se prefieren árboles profundos (sesgo bajo).

En *boosting* el sesgo se disminuye con modelos predictivos que se encargan de distintos grupos en el conjunto de entrenamiento. La varianza se puede controlar con el parámetro de tasa de aprendizaje.

2. MODELOS ADITIVOS SECUENCIALES

En el marco de `modelos aditivos secuenciales` aplicados a regresión, consideremos el contexto de modelado por etapas. En este marco, consideremos un predictor de la forma

$$f(x) = \sum_{k=1}^{M} \beta_k \, b_k(x) = \sum_{k=1}^{M} T_k(x)\,, \tag{1}$$

donde cada término es el resultado de ajustar un árbol de regresión.

Recuerda que un árbol T_k de decisión está definido por sus parámetros θ_k los cuales incluyen las variables que se utilizan para los cortes, el punto de corte y la predicción en cada una de las regiones terminales.

Resolver este problema es dificil pues se tienen que ajustar los coeficientes y los árboles al mismo tiempo. Asi que lo resolveremos de manera secuencial. Es decir, consideremos que estamos en la iteración m y que un modelo pre-entrenado

$$f_{m-1}(x) = \sum_{k=1}^{m-1} T_k(x)\,, \tag{2}$$

donde T_k con $k = 1, \ldots, m-1$ son los árboles individualmente entrenados.

Para la iteración actual, ajustaremos un modelo adicional al resolver

$$\min_{T\in\mathcal{T}} \sum_{i=1}^{n} \ell\left(y_i, f_{m-1}(x_i) + T(x_i)\right) , \tag{3}$$

donde $\ell(y, \hat{y})$ es una función de pérdida adecuada.

En el contexto de regresión, si utilizamos pérdida cuadrática podemos agrupar y reescribir

$$\min_{T\in\mathcal{T}} \sum_{i=1}^{n} \left(r_i^{(m)} - T(x_i)\right)^2 , \tag{4}$$

donde $r_i^{(m)}$ es el residual que enfrentamos al momento de ajustar el m-ésimo modelo.

2.1. Clasificación binaria

Para problemas de clasificación necesitamos resolver en escala logarítmica para después transformar a probabilidades por medio de

$$\mathbb{P}(Y = 1|x) = p(x) = h(f(x)) , \tag{5}$$

donde h es la función logística.

En este caso resolvemos el problema de optimización

$$\min_{T\in\mathcal{T}} \sum_{i=1}^{n} \ell\left(y_i, f_{m-1}(x_i) + T(x_i)\right) , \tag{6}$$

donde ℓ es la `devianza` (- 2 $\times$ log-verosimilitud). Además, usaremos la siguiente codificación de las clases $y \in \{-1, 1\}$. Por lo que la devianza la podemos escribir como

$$\ell(y, \hat{z}) = -\left[(y+1)\log h(\hat{z}) - (y-1)\log(1 - h(\hat{z}))\right] , \tag{7}$$

la cual se puede escribir (utilizando la expresión de la función h) como

$$\ell(y, \hat{z}) = 2\log\left(1 + e^{-y\hat{z}}\right) . \tag{8}$$

> Nota que en el contexto de regresión el producto $y\hat{z}$ tiene un efecto similar al de un residual en el contexto de regresión. ¿Qué pasa con los signos de la categoría observada y y del *score* latente $\hat{z}$ cuando coinciden y cuando no?

Esto nos permite formular el problema de optimización como

$$\min_{T\in\mathcal{T}} \sum_{i=1}^{n} 2\log\left(1 + e^{-y_i\cdot(f_{m-1}(x_i)+T(x_i))}\right) , \tag{9}$$

el cual puede ser re-escrito como

$$\min_{T\in\mathcal{T}} \sum_{i=1}^{n} 2\log\left(1 + d_i^{(m)} e^{-y_i T(x_i)}\right) . \tag{10}$$

2.1.1. Para pensar: ¿Cómo cambia la función objetivo si la función de pérdida es una pérdida exponencial? Es decir, bajo

$$\ell(y, f_m(x)) = \exp\left(-y f_m(x)\right) . \tag{11}$$

En este caso (clasificación binaria) vemos que la formulación no es tan fácil que con pérdida cuadrática en el contexto de regresión. Sin embargo, parece ser que los datos reciben una ponderación $d_i^{(m)}$.

3. GRADIENT BOOSTING

Utilizaremos *gradient boosting* para resolver la formulación anterior. Esta técnica se puede aplicar de manera general para cualquier función de pérdida. Lo que necesitamos es resolver

$$\min_{T \in \mathcal{T}} \sum_{i=1}^{n} \ell\left(y_i, f_{m-1}(x_i) + T(x_i)\right) , \tag{12}$$

donde podemos interpretar como *qué tanto tenemos que movernos a partir de* f_{m-1} *para disminuir la función de pérdida.*

Si nos fijamos en uno de los términos y escribimos

$$\ell(y_i, f_{m-1}(x_i) + z_i) , \tag{13}$$

entonces sabemos que tenemos que modificar de acuerdo a

$$z_i = -\lambda \cdot \frac{\partial \ell}{\partial z_i}\left(y_i, f_{m-1}(x_i) + z\right)\Big|_{z=0} , \tag{14}$$

donde tomamos una `longitud de paso` (λ) para estabilizar el proceso de optimización.

La restricción que tenemos es que $z_i = T(x_i)$ para toda observación. De hecho, lo mejor que podríamos tener es que

$$T(x_i) \approx \frac{\partial \ell}{\partial z_i}\left(y_i, f_{m-1}(x_i)\right) = g_{i,m} . \tag{15}$$

Así que formulamos un nuevo problema de optimización como

$$\min_{T \in \mathcal{T}} \sum_{i=1}^{n} \left(g_{i,m} - T(x_i)\right)^2 , \tag{16}$$

donde $g_{i,m}$ es el gradiente de la función objetivo en la iteración m evaluando en la observación i. Por lo que la actualización sería de la forma

$$f_m(x) = f_{m-1}(x) + \lambda T(x) , \tag{17}$$

donde el parámetro λ se puede interpretar como una longitud de paso o un mecanismo de suavizamiento/regularización.

3.0.1. Para pensar: ¿Tiene sentido utilizar un árbol de clasificación para la formulación anterior?

3.1. Pseudo-código (tarea de regresión)

El caso de regresión con pérdida cuadrática se presenta a continuación como un ejemplo. Nota que en esta formulación el problema de optimización en términos del gradiente de la función de pérdida es equivalente a ajustar los residuales. Esto nos ayuda a conectar la formulación con resolver un problema predictivo. La tasa de aprendizaje en este caso, previene el sobreajuste pues ayuda a reducir la importancia de la solución a los residuales.

1. Definimos $\hat{f}(x) = 0$, y $r_i = y_i$ para toda observación en el conjunto de entrenamiento.
2. Para cada $m = 1, 2, \ldots, M$:
   a) Ajustamos un árbol sencillo $\hat{f}_m$ con J nodos terminales para el conjunto $\{(x_i, r_i)\}_{i=1}^n$.
   b) Actualizamos el predictor $\hat{f}$ al incluir una versión escalada del nuevo árbol
   $$\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}_m(x)\,. \tag{18}$$
   c) Actualizamos los residuales
   $$r_i \leftarrow r_i - \lambda \hat{f}_m(x)\,. \tag{19}$$
3. Regresamos el modelo
   $$\hat{f}(x) = \lambda \sum_{m=1}^{M} \hat{f}_m(x)\,. \tag{20}$$

Hay algunas librerías que no consideran λ como un parámetro de búsqueda. Al contrario, utilizan un mecanismo para ajustar la longitud de paso de manera automática. Le pueden preguntar a su profesor favorito de optimización numérica por *line search* o consultar el libro de Nocedal and Wright [4].

3.2. Funciones de pérdida

La selección de función de pérdida parte crucial del algoritmo y se escoge de acuerdo al problema y al objetivo que se quiera resolver. Por ejemplo, en regresión tenemos (por nombrar un par):

1. **Pérdida cuadrática**:
   $$\ell(y, z) = \frac{1}{2}(y - z)^2, \qquad \frac{\partial \ell}{\partial z} = -(y - z)\,. \tag{21}$$
2. **Pérdida absoluta**:
   $$\ell(y, z) = |y - z|, \qquad \frac{\partial \ell}{\partial z} = \frac{|y - z|}{y - z}\,. \tag{22}$$

En el contexto de clasificación podemos utilizar:

1. **Devianza binomial**:
   $$\ell(y, z) = -\log(1 + e^{-yz}), \qquad \frac{\partial \ell}{\partial z} = I(y = 1) - h(z)\,. \tag{23}$$
2. **Pérdida exponencial**:
   $$\ell(y, z) = e^{-yz}, \qquad \frac{\partial \ell}{\partial z} = -ye^{-yz}\,. \tag{24}$$

3.3. Parámetros a optimizar

Los parámetros que usualmente se ajustan con validación cruzada son:

- La tasa de aprendizaje o tamaño de paso λ.
- El número de términos del modelo M.

Más los adicionales de la familia de árboles:

- Profundidad del árbol.
- Número de observaciones en los nodos terminales.

Se pueden incorporar adicionales:

- El número de predictores a utilizar (como en `RF`).
- Alguna cota de reducción de función objetivo para profundizar el árbol.
- Tamaño de submuestreo (*Stochastic gradient boosting*).

3.4. Observaciones

- El número de arboles en la expansión puede llevar a sobre-ajuste.
- La estrategia de validación cruzada puede ser muy útil para determinar el número de iteraciones en algunas situaciones.
- Se puede utilizar un criterio de paro para evitar sobre-ajuste.
- Los hiper-parámetros M y λ **no** son independientes.

4. APLICACIÓN: ANALÍTICA DEPORTIVA

El objetivo es determinar si un equipo de *volleyball* ganará su siguiente partido tomando en cuenta estadísticas como errores, ataques, bloqueos, etc. por equipo.

Necesitamos los estadísticos por equipo, no por jugador.

```
vb_parsed ← vb_matches ▷
transmute(
  circuit,
  gender,
  year,
  w_attacks = w_p1_tot_attacks + w_p2_tot_attacks,
  w_kills = w_p1_tot_kills + w_p2_tot_kills,
  w_errors = w_p1_tot_errors + w_p2_tot_errors,
  w_aces = w_p1_tot_aces + w_p2_tot_aces,
  w_serve_errors = w_p1_tot_serve_errors + w_p2_tot_serve_errors,
  w_blocks = w_p1_tot_blocks + w_p2_tot_blocks,
  w_digs = w_p1_tot_digs + w_p2_tot_digs,
  l_attacks = l_p1_tot_attacks + l_p2_tot_attacks,
  l_kills = l_p1_tot_kills + l_p2_tot_kills,
  l_errors = l_p1_tot_errors + l_p2_tot_errors,
  l_aces = l_p1_tot_aces + l_p2_tot_aces,
  l_serve_errors = l_p1_tot_serve_errors + l_p2_tot_serve_errors,
  l_blocks = l_p1_tot_blocks + l_p2_tot_blocks,
  l_digs = l_p1_tot_digs + l_p2_tot_digs
) ▷
na.omit()
vb_parsed ▷ print(n = 3, width = 75)
```

```
# A tibble: 14,332 × 17
  circuit gender  year w_attacks w_kills w_errors w_aces w_serve_...¹ w_...²blo
  <chr>   <chr>  <dbl>     <dbl>   <dbl>    <dbl>  <dbl>       <dbl>   <dbl>
```

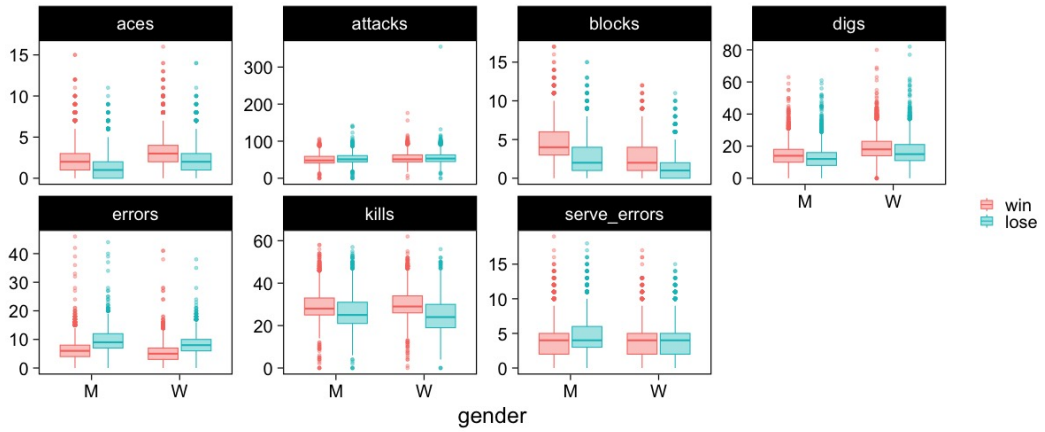
```
1 AVP      M        2004         45       24         7       0           2         5
2 AVP      M        2004         71       31        16       3           8         7
3 AVP      M        2004         43       26         5       2           4         7
# ... with 14,329 more rows, 8 more variables: w_digs <dbl>,
#   l_attacks <dbl>, l_kills <dbl>, l_errors <dbl>, l_aces <dbl>,
#   l_serve_errors <dbl>, l_blocks <dbl>, l_digs <dbl>, and abbreviated
#   variable names ¹w_serve_errors, ²w_blocks
#  Use `print(n = ...)` to see more rows, and `colnames()` to see all variable
    names
```

Separemos por equipos que ganan sus partidos y los que no.

```
winners ← vb_parsed ▷
  select(circuit, gender, year,
         w_attacks:w_digs) ▷
  rename_with(~ str_remove_all(., "w_"), w_attacks:w_digs) ▷
  mutate(win = "win")

losers ← vb_parsed ▷
  select(circuit, gender, year,
         l_attacks:l_digs) ▷
  rename_with(~ str_remove_all(., "l_"), l_attacks:l_digs) ▷
  mutate(win = "lose")

vb_df ← bind_rows(winners, losers) ▷
  mutate_if(is.character, factor) ▷
  mutate(win = factor(win, levels = c("win", "lose")))
```



4.1. Construcción del modelo

```
set.seed(123)
vb_split ← initial_split(vb_df, strata = win)
vb_train ← training(vb_split)
vb_test ← testing(vb_split)
```

```
xgb_spec ← boost_tree(
  trees = 1000,
  tree_depth = tune(), min_n = tune(), ## complexity
  mtry = tune(),         ## randomness
  learn_rate = tune()    ## step size
```

```
) ▷
  set_engine("xgboost") ▷
  set_mode("classification")

xgb_spec
```

```
Boosted Tree Model Specification (classification)

Main Arguments:
  mtry = tune()
  trees = 1000
  min_n = tune()
  tree_depth = tune()
  learn_rate = tune()

Computational engine: xgboost
```

```
xgb_rec ← recipe(win ~ ., vb_train) ▷
  step_dummy(all_nominal_predictors())

xgb_rec
```

```
Recipe

Inputs:

      role #variables
   outcome           1
 predictor          10

Operations:

Dummy variables from all_nominal_predictors()
```

Usaremos un diseño experimental tipo *latin hypercube*. Su característica principal es que es un diseño que tiende a cubrir de manera uniforme el espacio de búsqueda con componentes aleatorios. Se denomina diseño de *cobertura de espacio* (*space filling*) y es uno de los que ya están codificados en `tidymodels`.

```
xgb_grid ← grid_latin_hypercube(
  tree_depth(),
  min_n(),
  finalize(mtry(), vb_train),
  learn_rate(),
  size = 30
)

xgb_grid ▷ print(n = 3)
```

```
# A tibble: 30 × 4
  tree_depth min_n  mtry    learn_rate
       <int> <int> <int>         <dbl>
1         13    29     8 0.00000543
```

```
2           2    11     9 0.00000000312
3           8    35    10 0.0000174
# ... with 27 more rows
#  Use 'print(n = ...)' to see more rows
```

4.2. Entrenamiento del modelo

```
xgb_wf ← workflow() ▷
  add_recipe(xgb_rec) ▷
  add_model(xgb_spec)

xgb_wf
```

```
== Workflow ====================================================================
Preprocessor: Recipe
Model: boost_tree()

-- Preprocessor ----------------------------------------------------------------
1 Recipe Step

- step_dummy()

-- Model -----------------------------------------------------------------------
Boosted Tree Model Specification (classification)

Main Arguments:
  mtry = tune()
  trees = 1000
  min_n = tune()
  tree_depth = tune()
  learn_rate = tune()

Computational engine: xgboost
```

```
set.seed(123)
vb_folds ← vfold_cv(vb_train, strata = win)
vb_folds ▷ print(n = 5)
```

```
#  10-fold cross-validation using stratification
# A tibble: 10 × 2
  splits               id
  <list>               <chr>
1 <split [19348/2150]> Fold01
2 <split [19348/2150]> Fold02
3 <split [19348/2150]> Fold03
4 <split [19348/2150]> Fold04
5 <split [19348/2150]> Fold05
# ... with 5 more rows
#  Use 'print(n = ...)' to see more rows
```

```
all_cores ← parallel::detectCores(logical = TRUE) - 1
library(doParallel)
cl ← makePSOCKcluster(all_cores)
registerDoParallel(cl)
```

```
set.seed(234)
system.time(
xgb_res ← tune_grid(
  xgb_wf,
  resamples = vb_folds,
  grid = xgb_grid,
  control = control_grid(
    save_pred = TRUE, parallel = "everything")
))
```

```
    user   system  elapsed
   5.381    1.946 1255.699
There were 30 warnings (use warnings() to see them)
```

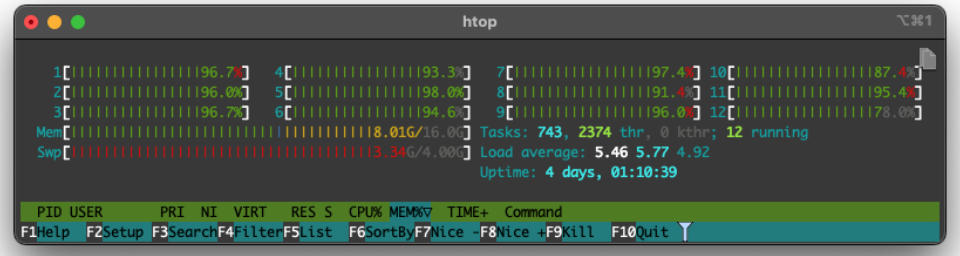


FIGURA 1. *Procesamiento en paralelo utilizando todos los recursos computacionales.*

```
set.seed(234)
system.time(
xgb_res ← tune_grid(
  xgb_wf,
  resamples = vb_folds,
  grid = xgb_grid,
  control = control_grid(
    save_pred = TRUE, parallel = "resamples")
))
```

```
 user   system  elapsed
1.128    0.374 1144.103
```

4.3. Resultados

```
collect_metrics(xgb_res) ▷ print(n = 5, width = 70)
```

```
# A tibble: 60 × 10
   mtry min_n tree_depth learn_...¹r .metric ....²esti   mean     n std_err
  <int> <int>      <int>      <dbl> <chr>   <chr>    <dbl> <int>   <dbl>
1     2     8          8   1.17e-10 ...accura binary   0.614    10 0.00216
```

```
2      2      8           8  1.17e-10 roc_auc binary   0.794     10 0.00252
3     10     13           5  8.28e- 4 ...accura binary   0.795     10 0.00227
4     10     13           5  8.28e- 4 roc_auc binary   0.890     10 0.00223
5      1     32           6  3.68e- 6 ...accura binary   0.797     10 0.00280
# ... with 55 more rows, 1 more variable: .config <chr>, and
#   abbreviated variable names ¹learn_rate, ².estimator
#  Use 'print(n = ...)' to see more rows, and 'colnames()' to see all variable
    names
```

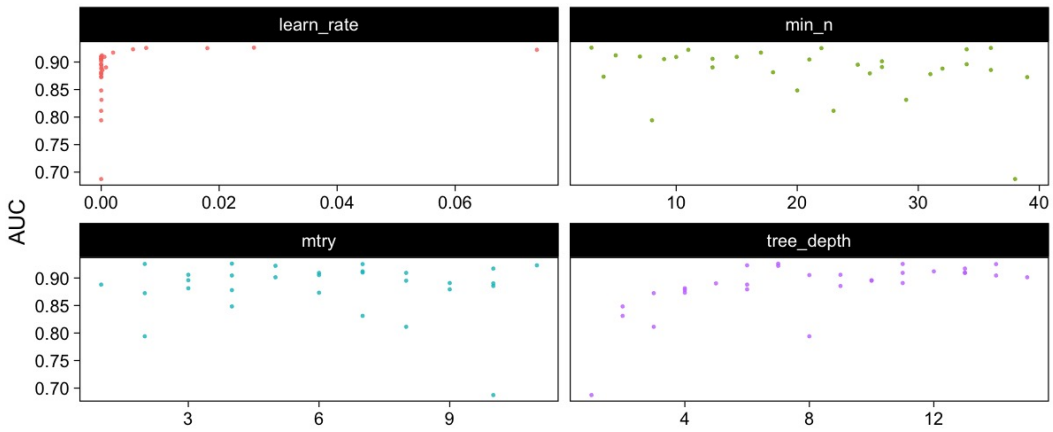


FIGURA 2. *Resultados del ajuste por validación cruzada.*

```
show_best(xgb_res, "roc_auc") ▷
  print(width = 70)
```

```
# A tibble: 5 × 10
   mtry min_n tree_depth learn_...¹r .metric ....²esti  mean     n std_err
  <int> <int>      <int>      <dbl> <chr>   <chr>   <dbl> <int>   <dbl>
1     4     3          7   0.0259   roc_auc binary   0.926    10 0.00190
2     2    36         11   0.00763  roc_auc binary   0.926    10 0.00214
3     7    22         14   0.0180   roc_auc binary   0.925    10 0.00195
4    11    34          6   0.00540  roc_auc binary   0.923    10 0.00213
5     5    11          7   0.0739   roc_auc binary   0.922    10 0.00199
# ... with 1 more variable: .config <chr>, and abbreviated variable
#   names ¹learn_rate, ².estimator
#  Use 'colnames()' to see all variable names
```

```
best_auc ← select_best(xgb_res, "roc_auc")
best_auc
```

```
# A tibble: 1 × 5
   mtry min_n tree_depth learn_rate .config
  <int> <int>      <int>      <dbl> <chr>
1     4     3          7     0.0259 Preprocessor1_Model24
```

4.4. Modelo entrenado

```
final_xgb ← finalize_workflow(
  xgb_wf,
  best_auc
)

final_xgb
```

```
== Workflow ====================================================================
Preprocessor: Recipe
Model: boost_tree()

-- Preprocessor ----------------------------------------------------------------
1 Recipe Step
- step_dummy()

-- Model -----------------------------------------------------------------------
Boosted Tree Model Specification (classification)

Main Arguments:
mtry = 4
trees = 1000
min_n = 3
tree_depth = 7
learn_rate = 0.0258957422005733

Computational engine: xgboost
```

```
final_res ← last_fit(final_xgb, vb_split)
collect_metrics(final_res)
```

```
# A tibble: 2 × 4
  .metric  .estimator .estimate .config
  <chr>    <chr>          <dbl> <chr>
1 accuracy binary         0.835 Preprocessor1_Model1
2 roc_auc  binary         0.924 Preprocessor1_Model1
```

4.5. Post-procesamiento

```
extract_boosted_prediction ← function(dt){
  final_res ▷
    extract_fit_parsnip() ▷
    multi_predict(new_data = prep(xgb_rec) ▷ bake(dt) ▷ select(-win),
                  trees = seq(1,1000, by = 5)) ▷
    mutate(truth = dt$win,
           id = 1:n()) ▷
    unnest(.pred) ▷
    group_by(trees) ▷
    nest(data = c(.pred_class, truth, id)) ▷
    mutate(results = map(data, function(x) {accuracy(x, .pred_class, truth)})) ▷
    ungroup()
}
```

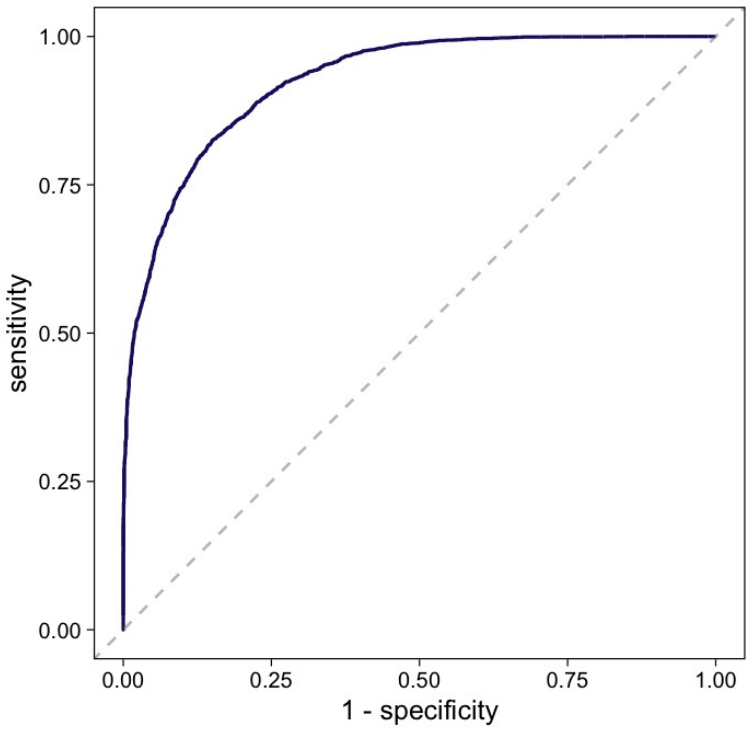
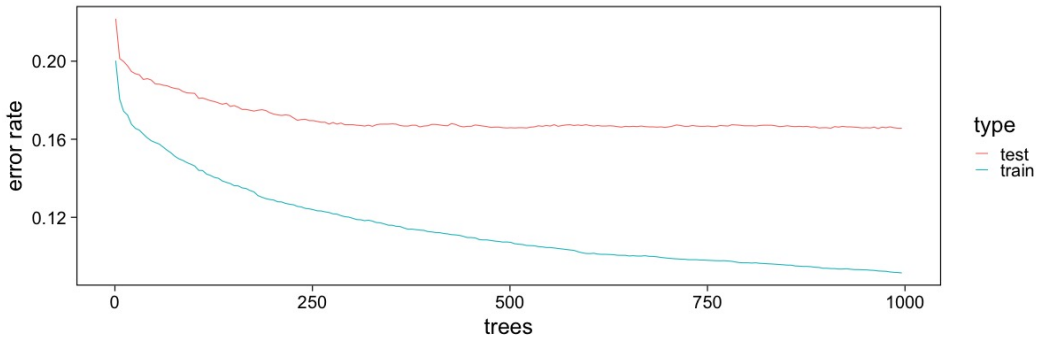


FIGURA 3. *Diagnóstico ROC para conjunto de prueba.*

```
preds_train ← extract_boosted_prediction(vb_train) ▷ mutate(type = "train")
preds_test ← extract_boosted_prediction(vb_test) ▷ mutate(type = "test")
```



5. PROCESAMIENTO EN PARALELO

Hemos utilizado procesamiento en paralelo para poder eficientar el cómputo necesario para el ajuste y comparación de modelos por medio de validación cruzada.

Hemos considerado dos esquemas de procesamiento en paralelo. Un método utiliza un ciclo usando los bloques de validación cruzada (o remuestras) y el otro utiliza las configuraciones distintas que se están considerando.

El esquema es el siguiente

```
for (rs in resamples) {
```

```
  ## Create analysis and assessment sets
  ## Preprocess data (e.g. formula or recipe)
  for (mod in configurations) {
    ## Fit model {mod} to the {rs} analysis set
    ## Predict the {rs} assessment set
  }
}
```

El *default* utiliza el ciclo externo y es lo mas conveniente cuando el pre-procesamiento es computacionalmente costoso.

Las desventajas son:

1. Tiene una limitante si el pre-procesamiento no es costoso.
2. Tiene un límite por el número de remuestras que utilizamos.

Consideremos la situación donde usamos validación cruzada con 5 bloques y estamos comparando 7 configuraciones distintas. En este escenario contamos con 5 *cores* para el procesamiento.

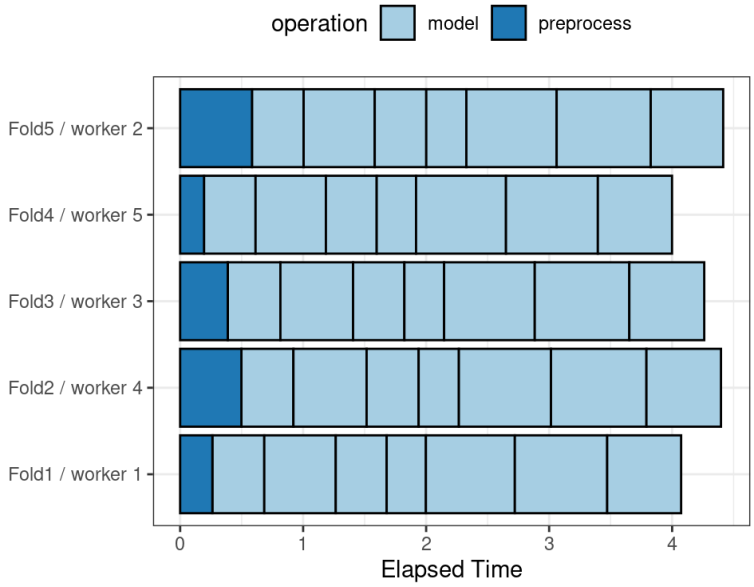


FIGURA 4. *Imagen tomada de [3].*

Por otro lado, podemos considerar todas las combinaciones posibles entre bloques e hiper-parámetros. Algo que esquemáticamente podemos representar como:

```
all_tasks ← crossing(resamples, configurations)

for (iter in all_tasks) {
  ## Create analysis and assessment sets for {iter}
  ## Preprocess data (e.g. formula or recipe)
  ## Fit model {iter} to the {iter} analysis set
  ## Predict the {iter} assessment set
}
```

En este escenario el pre-procesamiento se repite distintas veces. Tantas como número de bloques y combinaciones únicas de hiper-parametros tengamos. El esquema se ve de esta manera si contamos con 10 *cores*

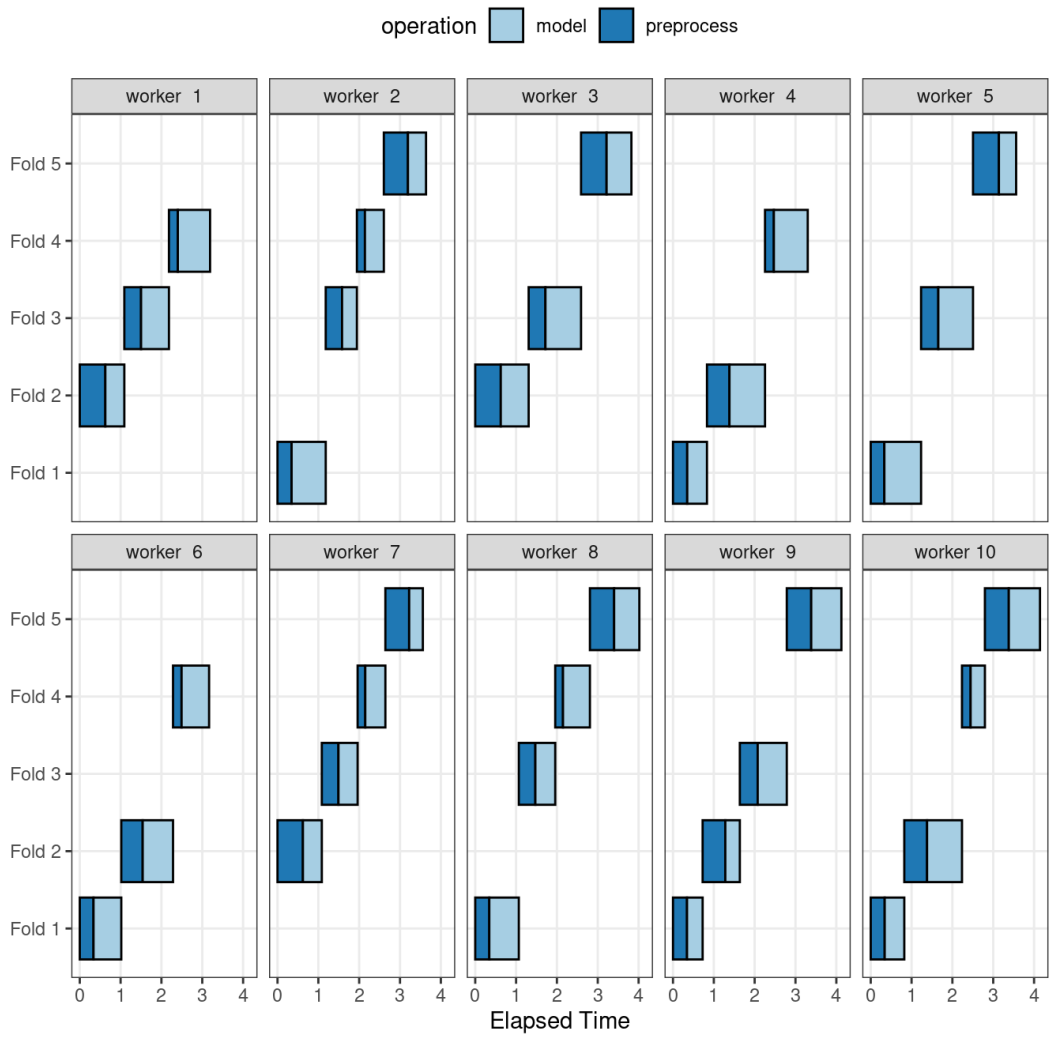


FIGURA 5. *Imagen tomada de [3].*

En la práctica entre mas *cores* tengamos disponibles mejores serán los retornos en eficiencia computacional (escenarios típicos de cómputo remoto).

6. ANOVA: PERFIL RÁPIDO DE DESEMPEÑO

```
library(finetune)
system.time(
  xgb_res ← tune_race_anova(
    xgb_wf,
    resamples = vb_folds,
    grid = xgb_grid,
    control = control_race(
      alpha = .05,
      verbose_elim = TRUE,
      save_pred = TRUE,
      parallel = "everything"))
)
```

```
Racing will maximize the roc_auc metric.
Resamples are analyzed in a random order.
Fold10: 25 eliminated;  5 candidates remain.
Fold02:  2 eliminated;  3 candidates remain.
Fold06:  0 eliminated;  3 candidates remain.
Fold04:  0 eliminated;  3 candidates remain.
Fold03:  0 eliminated;  3 candidates remain.
Fold07:  0 eliminated;  3 candidates remain.
Fold08:  0 eliminated;  3 candidates remain.
   user  system elapsed
  7.832   0.766 517.435
```

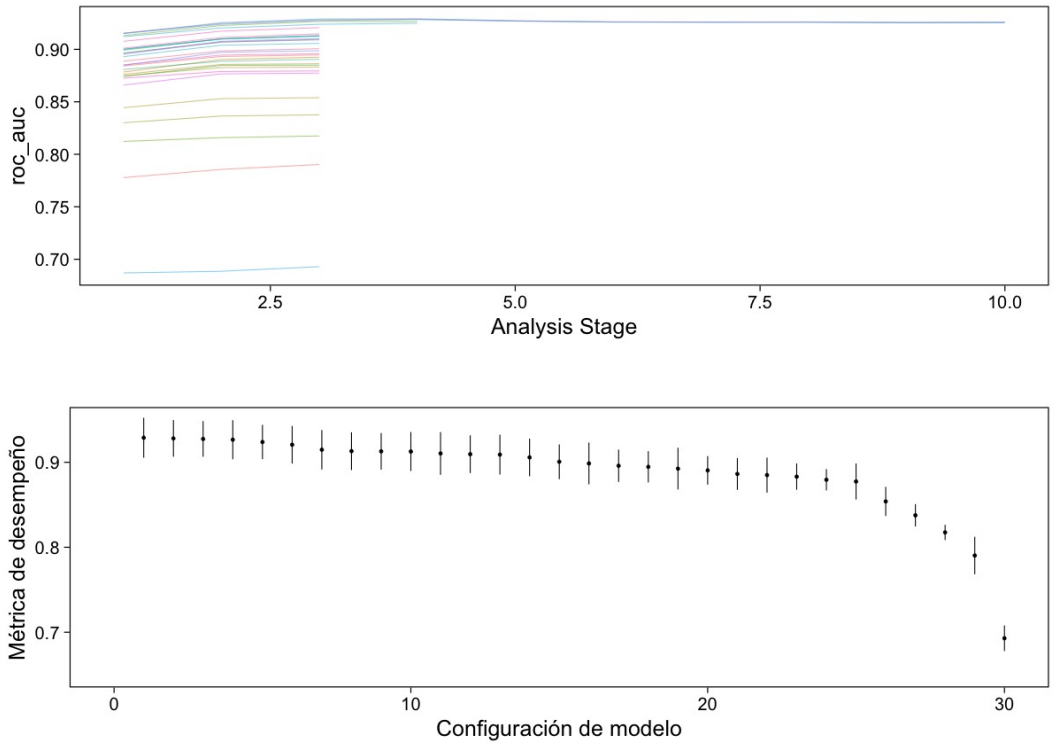


FIGURA 6. *Estimación de desempeño con tres evaluaciones.*

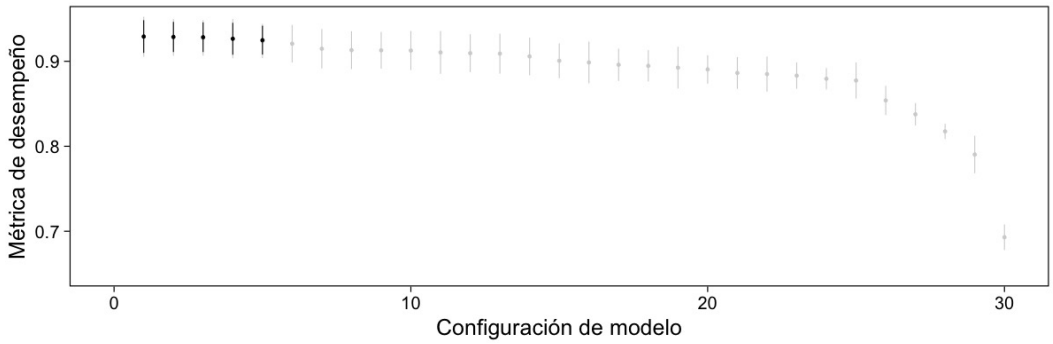


FIGURA 7. *Estimación de desempeño con cuatro evaluaciones.*

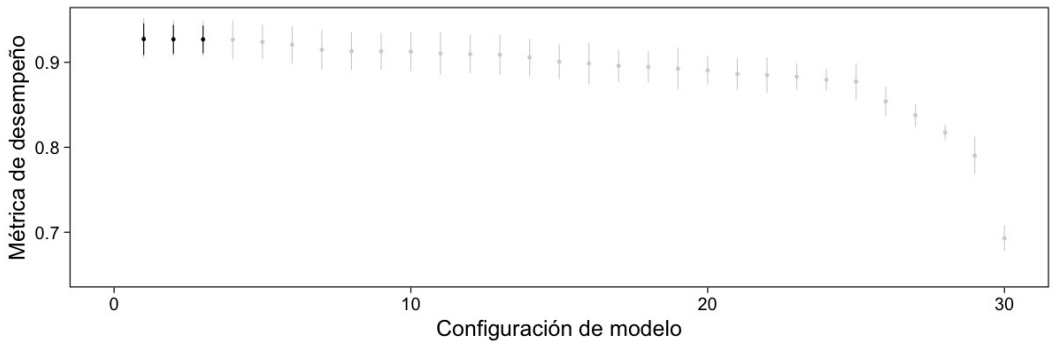


FIGURA 8. *Estimación de desempeño con cinco evaluaciones.*

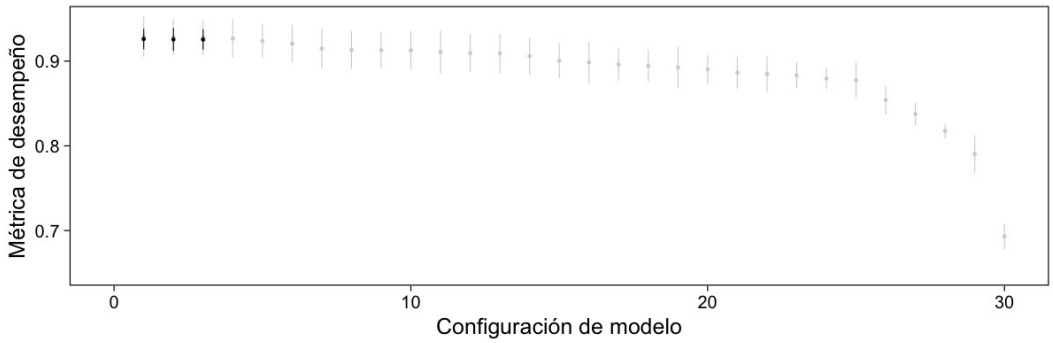


FIGURA 9. *Estimación de desempeño con diez evaluaciones.*

7. MODELOS DE ENSAMBLE (EPILOGO)

Hasta ahora tenemos tres opciones de ensambles: .... Usualmente *boosting* requiere mas cuidado en la selección de hiper-parámetros, mientras que alternativas basadas en remuestreo usualmente funcionan bien sin requerir tanta selección de valores apropiados. ¿Puedes decir por qué? Es por esto que Greenwell [1] tiene en mente su opinión

$$\mathsf{Boosting} \geq \mathsf{RandomForest} > \mathsf{Bagging} > \mathsf{Arbol}\,. \tag{25}$$

7.1. Importancia de variables

Tanto para bosques aleatorios como modelos de *boosting* basados en árboles se puede estimar cuáles fueron las variables (atributos) que mas contribuyeron en la construcción del modelo.

En cada *nodo interno* de los árboles construidos sabemos que la variable y el punto de corte se escogieron de acuerdo a que maximizaban la `mejora` en dicha región.

Para calcular la importancia de la variable j en el árbol T se suman las contribuciones cada vez que esta variable fue utilizada para generar cortes

$$\mathcal{I}_j^2(T) = \sum_{t=1}^{J-1} \hat{\iota}_t^2 I(v(t) = j)\,, \tag{26}$$

donde $\hat{\iota}_t^2$ es el registro de la mejora en `RSS`, `Gini` o `entropía cruzada` por el nodo t cuando este nodo toma el corte utilizando la variable $v(t)$.

La métrica de importancia en un ensamble de modelos considera promediar la mejora en todo el ensamble

$$\mathcal{I}_j^2 = \frac{1}{M} \sum_{m=1}^{M} \mathcal{I}_j^2(T_m)\,. \tag{27}$$

Se acostumbra registrar importancias relativas de manera que la variable con mayor importancia se le asigna un *score* de 100 puntos y las demás se calculan de manera proporcional.

REFERENCIAS

[1] B. M. Greenwell. *Tree-Based Methods for Statistical Learning in R*. Chapman and Hall/CRC, Boca Raton, first edition, may 2022. ISBN 978-1-00-308903-2. . 1, 16

[2] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York, New York, NY, 2009. ISBN 978-0-387-84857-0 978-0-387-84858-7. . 1

[3] M. Kuhn and J. Silge. *Tidy Modeling with R*. .O'Reilly Media, Inc.", 2022. 13, 14

[4] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer Series in Operations Research. Springer, New York, 2nd ed edition, 2006. ISBN 978-0-387-30303-1. 4

[5] R. Schapire. *The Boosting Approach to Machine Learning: An Overview*. MIT press, 2001. 1