

EST-25134: Aprendizaje Estadístico

Profesor: Alfredo Garbuno Iñigo — Primavera, 2023 — Ensamblados 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/varianza baja), al contrario de bosques aleatorios donde se prefieren árboles profundos (sesgo bajo/varianza alta).

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 un parámetro que llamamos 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

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

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

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

donde T_k con $k = 1, \dots, m-1$ son los árboles entrenados en iteraciones previas.

Para la iteración actual, ajustaremos un modelo adicional al resolver

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

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

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

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

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

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

donde h es la función logística.

En este caso resolvemos el problema de optimización

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

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

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

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

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

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

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

el cual puede ser re-escrito como

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

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

$$\ell(y, f_m(x)) = \exp(-yf_m(x)) . \quad (11)$$

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

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

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

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

entonces sabemos que tenemos que modificar de acuerdo a al negativo del gradiente

$$z_i = - \frac{\partial \ell}{\partial z_i} (y_i, f_{m-1}(x_i) + z) \Big|_{z=0} . \quad (14)$$

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

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

Así que formulamos un nuevo problema de optimización como

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

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

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

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 $\hat{f}_0(x) = 0$, y $r_{i,0} = y_i$ para toda observación en el conjunto de entrenamiento.
2. Para cada $m = 1, 2, \dots, M$:

- a) Ajustamos un árbol sencillo \hat{f}_m con J hojas y datos $\{(x_i, r_{i,m})\}_{i=1}^n$.
- b) Actualizamos el predictor \hat{f} al incluir una versión escalada del nuevo árbol

$$\hat{f}_m(x) = \hat{f}_{m-1}(x) + \lambda \hat{f}_m(x). \quad (18)$$

3. Actualizamos los residuales $r_{i,m} = r_{i,m-1} - \lambda \hat{f}_m(x)$.
4. Regresamos el modelo

$$\hat{f}(x) = \lambda \sum_{m=1}^M \hat{f}_m(x). \quad (19)$$

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 es 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:**

$$\ell(y, z) = \frac{1}{2}(y - z)^2, \quad \frac{\partial \ell}{\partial z} = -(y - z). \quad (20)$$

2. **Pérdida absoluta:**

$$\ell(y, z) = |y - z|, \quad \frac{\partial \ell}{\partial z} = \frac{|y - z|}{y - z}. \quad (21)$$

En el contexto de clasificación podemos utilizar:

1. **Devianza binomial:**

$$\ell(y, z) = -\log(1 + e^{-yz}), \quad \frac{\partial \ell}{\partial z} = I(y = 1) - h(z). \quad (22)$$

2. **Pérdida exponencial:**

$$\ell(y, z) = e^{-yz}, \quad \frac{\partial \ell}{\partial z} = -ye^{-yz}. \quad (23)$$

3.3. Parámetros a optimizar

Los parámetros de *boosting* que usualmente se ajustan 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.

```

1 vb_parsed <- vb_matches >
2   transmute(circuit, gender, year,
3             w_attacks = w_p1_tot_attacks + w_p2_tot_attacks,
4             w_kills = w_p1_tot_kills + w_p2_tot_kills,
5             w_errors = w_p1_tot_errors + w_p2_tot_errors,
6             w_aces = w_p1_tot_aces + w_p2_tot_aces,
7             w_serve_errors = w_p1_tot_serve_errors + w_p2_tot_serve_errors,
8             w_blocks = w_p1_tot_blocks + w_p2_tot_blocks,
9             w_digs = w_p1_tot_digs + w_p2_tot_digs,
10            l_attacks = l_p1_tot_attacks + l_p2_tot_attacks,
11            l_kills = l_p1_tot_kills + l_p2_tot_kills,
12            l_errors = l_p1_tot_errors + l_p2_tot_errors,
13            l_aces = l_p1_tot_aces + l_p2_tot_aces,
14            l_serve_errors = l_p1_tot_serve_errors + l_p2_tot_serve_errors,
15            l_blocks = l_p1_tot_blocks + l_p2_tot_blocks,
16            l_digs = l_p1_tot_digs + l_p2_tot_digs
17          ) >
18   na.omit()
19 vb_parsed > print(n = 3, width = 75)

```

```

1 # A tibble: 14,332 × 17
2   circuit gender  year w_attacks w_kills w_errors w_aces w_serve...1 w...2blo
3   <chr>    <chr>  <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
4 1 AVP      M      2004      45      24      7       0       2       5
5 2 AVP      M      2004      71      31     16       3       8       7
6 3 AVP      M      2004      43      26      5       2       4       7

```

```

7 # ... with 14,329 more rows, 8 more variables: w_digs <dbl>,
8 #   l_attacks <dbl>, l_kills <dbl>, l_errors <dbl>, l_aces <dbl>,
9 #   l_serve_errors <dbl>, l_blocks <dbl>, l_digs <dbl>, and abbreviated
10 #   variable names 1w_serve_errors, 2w_blocks
11 # Use 'print(n = ...)' to see more rows, and 'colnames()' to see all variable
   names

```

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

```

1 winners <- vb_parsed >
2   select(circuit, gender, year,
3         w_attacks:w_digs) >
4   rename_with(~ str_remove_all(., "w_"), w_attacks:w_digs) >
5   mutate(win = "win")
6
7 losers <- vb_parsed >
8   select(circuit, gender, year,
9         l_attacks:l_digs) >
10  rename_with(~ str_remove_all(., "l_"), l_attacks:l_digs) >
11  mutate(win = "lose")

```

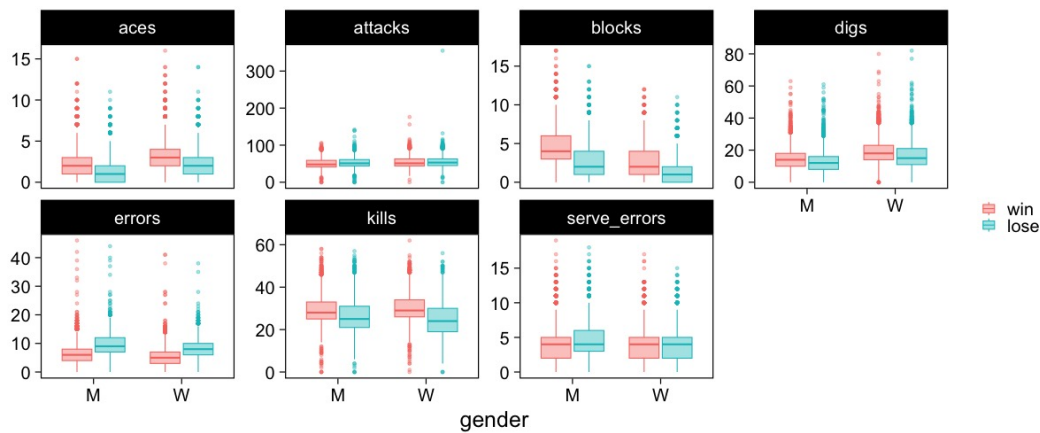


FIGURA 1. Análisis exploratorio inicial.

4.1. Construcción del modelo

```

1 set.seed(123)
2 vb_split <- initial_split(vb_df, strata = win)
3 vb_train <- training(vb_split)
4 vb_test <- testing(vb_split)
5
6 xgb_spec <- boost_tree(
7   trees = 1000,
8   tree_depth = tune(), min_n = tune(), ## complexity
9   mtry = tune(), ## randomness
10  learn_rate = tune() ## step size
11 ) >
12   set_engine("xgboost") >
13   set_mode("classification")
14
15 xgb_spec

```

```

1 Boosted Tree Model Specification (classification)
2
3 Main Arguments:
4   mtry = tune()
5   trees = 1000
6   min_n = tune()
7   tree_depth = tune()
8   learn_rate = tune()
9
10 Computational engine: xgboost

```

```

1 xgb_rec <- recipe(win ~ ., vb_train) >
2   step_dummy(all_nominal_predictors())
3
4 xgb_rec

```

```

1 Recipe
2
3 Inputs:
4
5     role #variables
6   outcome      1
7   predictor     10
8
9 Operations:
10
11 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*.

```

1 xgb_grid <- grid_latin_hypercube(
2   tree_depth(),
3   min_n(),
4   finalize(mtry(), vb_train),
5   learn_rate(),
6   size = 30
7 )
8
9 xgb_grid > print(n = 3)

```

```

1 # A tibble: 30 × 4
2   tree_depth min_n mtry learn_rate
3     <int> <int> <int>     <dbl>
4 1         2    20     6  1.21e- 8
5 2         6    22     7  1.86e-10
6 3         5    30     6  7.34e- 3
7 # ... with 27 more rows
8 # Use 'print(n = ...)' to see more rows

```

4.2. Entrenamiento del modelo

```

1 xgb_wf ← workflow() ▷
2   add_recipe(xgb_rec) ▷
3   add_model(xgb_spec)
4
5 xgb_wf

```

```

1 == Workflow =====
2 Preprocessor: Recipe
3 Model: boost_tree()
4 -- Preprocessor -----
5 1 Recipe Step
6 - step_dummy()
7 -- Model -----
8 Boosted Tree Model Specification (classification)
9 Main Arguments:
10   mtry = tune()
11   trees = 1000
12   min_n = tune()
13   tree_depth = tune()
14   learn_rate = tune()
15 Computational engine: xgboost

```

```

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

```

```

1 # 10-fold cross-validation using stratification
2 # A tibble: 10 × 2
3   splits          id
4   <list>         <chr>
5 1 <split [19348/2150]> Fold01
6 2 <split [19348/2150]> Fold02
7 3 <split [19348/2150]> Fold03
8 4 <split [19348/2150]> Fold04
9 5 <split [19348/2150]> Fold05
10 # ... with 5 more rows
11 # Use 'print(n = ...)' to see more rows

```

```

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

```

```

1 set.seed(234)
2 system.time(
3   xgb_res ← tune_grid(
4     xgb_wf,
5     resamples = vb_folds,
6     grid = xgb_grid,
7     control = control_grid(
8       save_pred = TRUE, parallel = "everything")
9   ))

```



```

1      user      system elapsed
2      5.381      1.946 1255.699
3 There were 30 warnings (use warnings() to see them)

```

```

1 set.seed(234)
2 system.time(
3 xgb_res <- tune_grid(
4   xgb_wf,
5   resamples = vb_folds,
6   grid = xgb_grid,
7   control = control_grid(
8     save_pred = TRUE, parallel = "resamples")
9 ))

```

```

1      user      system elapsed
2      1.439      0.755  820.734

```

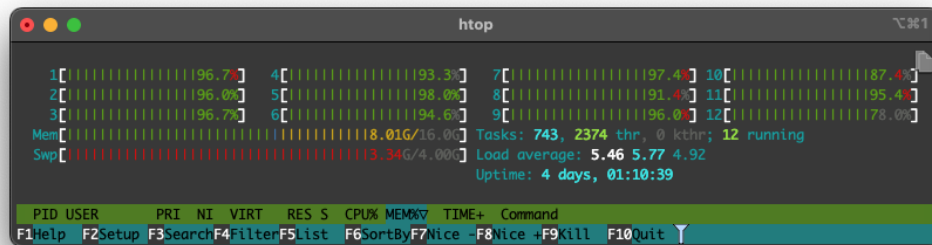


FIGURA 2. Procesamiento en paralelo utilizando todos los recursos computacionales.

4.3. Resultados

```

1 collect_metrics(xgb_res) > print(n = 5, width = 70)

```

```

1 # A tibble: 60 × 10
2   mtry min_n tree_depth learn_...1r .metric ....2esti mean      n std_err
3   <int> <int>      <int>      <dbl> <chr>      <chr>      <dbl> <int>  <dbl>
4 1     6     20          2  1.21e- 8 ...accura binary  0.735     10 0.00289
5 2     6     20          2  1.21e- 8 roc_auc  binary  0.835     10 0.00218
6 3     7     22          6  1.86e-10 ...accura binary  0.738     10 0.00194
7 4     7     22          6  1.86e-10 roc_auc  binary  0.856     10 0.00279
8 5     6     30          5  7.34e- 3 ...accura binary  0.836     10 0.00229
9 # ... with 55 more rows, 1 more variable: .config <chr>, and
10 #   abbreviated variable names 1learn_rate, 2.estimator
11 # Use 'print(n = ...)' to see more rows, and 'colnames()' to see all variable
    names

```

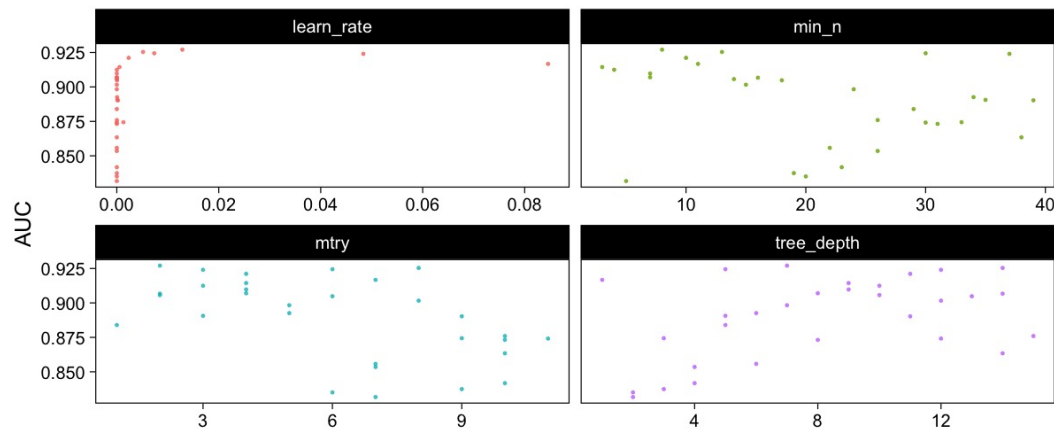


FIGURA 3. Resultados del ajuste por validación cruzada.

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

```
1 # A tibble: 5 × 10
2   mtry min_n tree_depth learn...1r .metric ...2esti mean      n std_err
3   <int> <int>      <int>      <dbl> <chr>   <chr>   <dbl> <int>   <dbl>
4 1     2     8          7    0.0129 roc_auc binary 0.927    10 0.00211
5 2     8    13         14    0.00515 roc_auc binary 0.925    10 0.00209
6 3     6    30          5    0.00734 roc_auc binary 0.924    10 0.00216
7 4     3    37         12    0.0484  roc_auc binary 0.924    10 0.00199
8 5     4    10         11    0.00232 roc_auc binary 0.921    10 0.00222
9 # ... with 1 more variable: .config <chr>, and abbreviated variable
10 #   names 1learn_rate, 2estimator
11 # Use 'colnames()' to see all variable names
```

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

```
1 # A tibble: 1 × 5
2   mtry min_n tree_depth learn_rate .config
3   <int> <int>      <int>      <dbl> <chr>
4 1     2     8          7    0.0129 Preprocessor1_Model13
```

4.4. Modelo entrenado

```
1 final_xgb ← finalize_workflow(
2   xgb_wf,
3   best_auc
4 )
5 final_xgb
```

```
1 == Workflow =====
```

```

2 Preprocessor: Recipe
3 Model: boost_tree()
4 -- Preprocessor -----
5 1 Recipe Step
6 - step_dummy()
7 -- Model -----
8 Boosted Tree Model Specification (classification)
9 Main Arguments:
10 mtry = 2
11 trees = 1000
12 min_n = 8
13 tree_depth = 7
14 learn_rate = 0.0128556104668359
15 Computational engine: xgboost

```

Al medir su capacidad predictiva vemos que no hay evidencia de sobreajuste.

```

1 final_res <- last_fit(final_xgb, vb_split)
2 collect_metrics(final_res)

```

```

1 # A tibble: 2 × 4
2   .metric .estimator .estimate .config
3   <chr>    <chr>        <dbl> <chr>
4 1 accuracy binary      0.833 Preprocessor1_Model1
5 2 roc_auc  binary      0.925 Preprocessor1_Model1

```

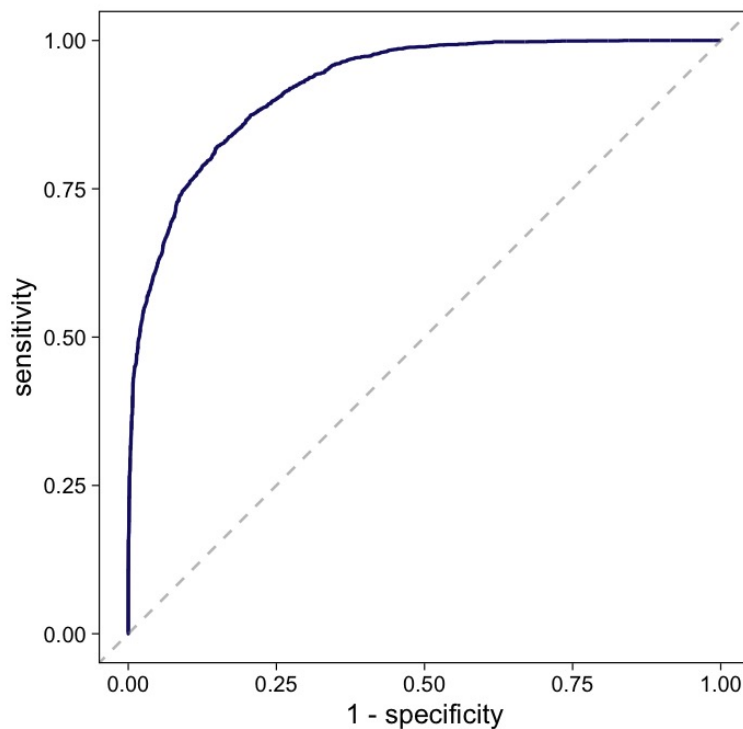


FIGURA 4. Diagnóstico ROC para conjunto de prueba.

4.5. Post-procesamiento

```

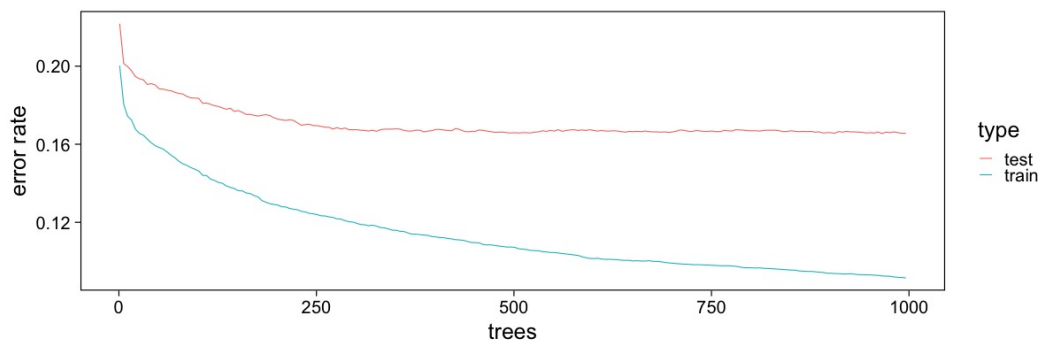
1 extract_boosted_prediction <- function(dt){
2   final_res >
3     extract_fit_parsnip() >
4     multi_predict(new_data = prep(xgb_rec) > bake(dt) > select(-win),
5                   trees = seq(1,1000, by = 5)) >
6     mutate(truth = dt$win,
7            id = 1:n()) >
8     unnest(.pred) >
9     group_by(trees) >
10    nest(data = c(.pred_class, truth, id)) >
11    mutate(results = map(data, function(x) {
12      accuracy(x, .pred_class, truth)})) >
13    ungroup()
14 }

```

```

1 preds_train <- extract_boosted_prediction(vb_train) > mutate(type = "train")
2 preds_test <- extract_boosted_prediction(vb_test) > mutate(type = "test")

```



```

1 extract_boosted_roc <- function(dt){
2   final_res >
3     extract_fit_parsnip() >
4     multi_predict(new_data = prep(xgb_rec) > bake(dt) > select(-win),
5                   trees = seq(1,1000, by = 1),
6                   type = "prob") >
7     mutate(truth = dt$win,
8            id = 1:n()) >
9     unnest(.pred) >
10    group_by(trees) >
11    nest(data = c(.pred_lose, .pred_win, truth, id)) >
12    mutate(results = map(data, function(x) {
13      roc_curve(x, truth, .pred_win)})) >
14    ungroup()
15 }

```

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.

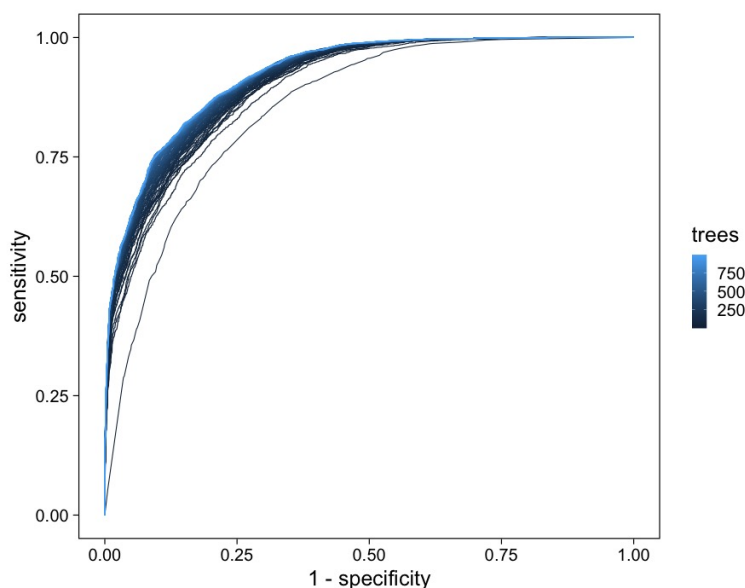


FIGURA 5. Diagnóstico ROC para modelo como función del número de árboles.

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

```

1 for (rs in resamples) {
2   ## Create analysis and assessment sets
3   ## Preprocess data (e.g. formula or recipe)
4   for (mod in configurations) {
5     ## Fit model {mod} to the {rs} analysis set
6     ## Predict the {rs} assessment set
7   }
8 }

```

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.

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

```

1 all_tasks <- crossing(resamples, configurations)
2 for (iter in all_tasks) {
3   ## Create analysis and assessment sets for {iter}
4   ## Preprocess data (e.g. formula or recipe)
5   ## Fit model {iter} to the {iter} analysis set
6   ## Predict the {iter} assessment set

```

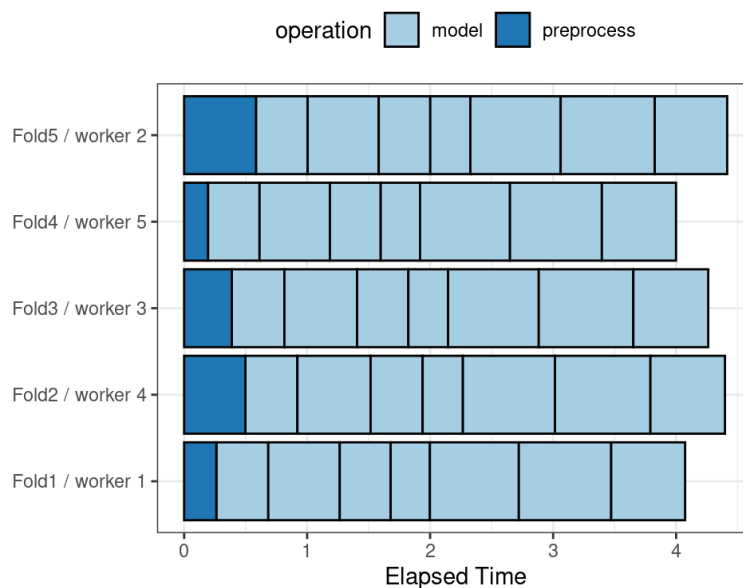


FIGURA 6. Imagen tomada de [3].

7 }

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*

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

```

1 library(finetune)
2 system.time(
3   xgb_res <- tune_race_anova(
4     xgb_wf,
5     resamples = vb_folds,
6     grid = xgb_grid,
7     control = control_race(
8       alpha = .05,
9       verbose_elim = TRUE,
10      save_pred = TRUE,
11      parallel = "everything"))
12 )

```

```

1 Racing will maximize the roc_auc metric.
2 Resamples are analyzed in a random order.
3 Fold10: 26 eliminated; 4 candidates remain.
4 Fold03: 2 eliminated; 2 candidates remain.
5 Fold09: All but one parameter combination were eliminated.
6 user system elapsed
7 6.232 1.516 420.782

```

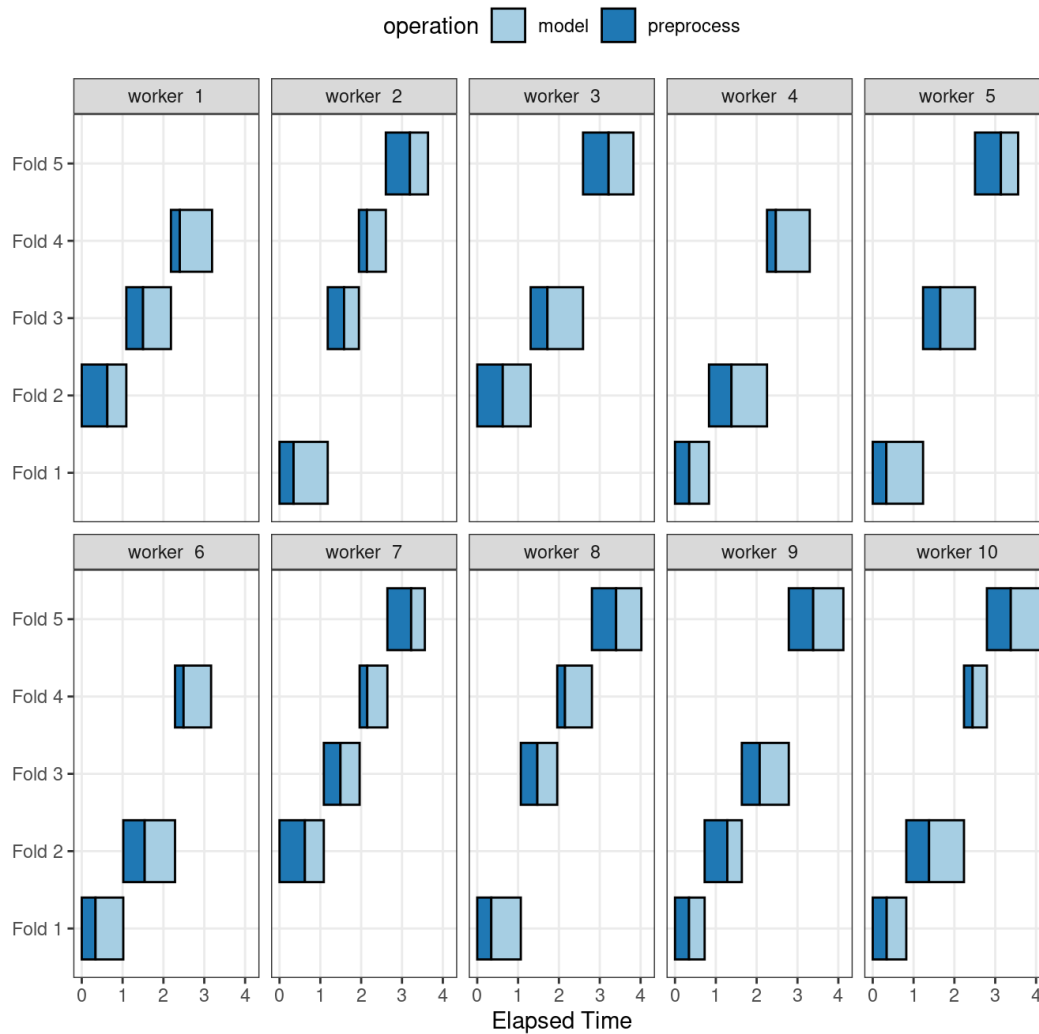
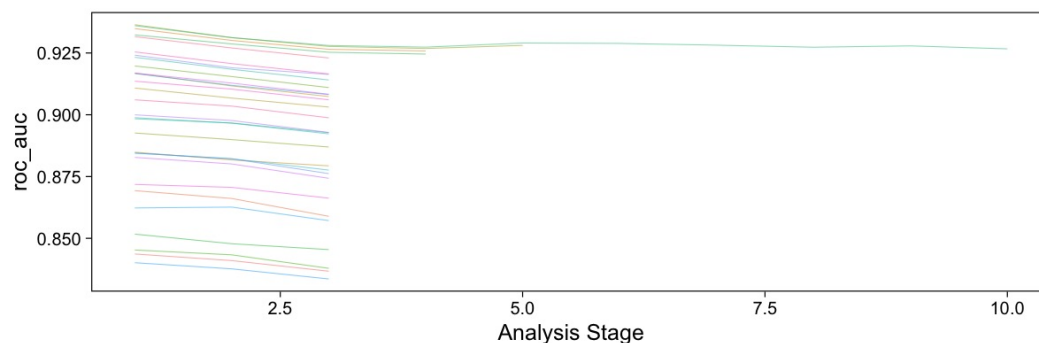


FIGURA 7. Imagen tomada de [3].



7. MODELOS DE ENSAMBLE (EPILOGO)

Hasta ahora tenemos tres opciones de ensambles: Usualmente *boosting* requiere mas cuidado en la selecci3n de hiper-par3metros, mientras que alternativas basadas en remuestreo usualmente funcionan bien sin requerir tanta selecci3n de valores apropiados. ¿Puedes decir

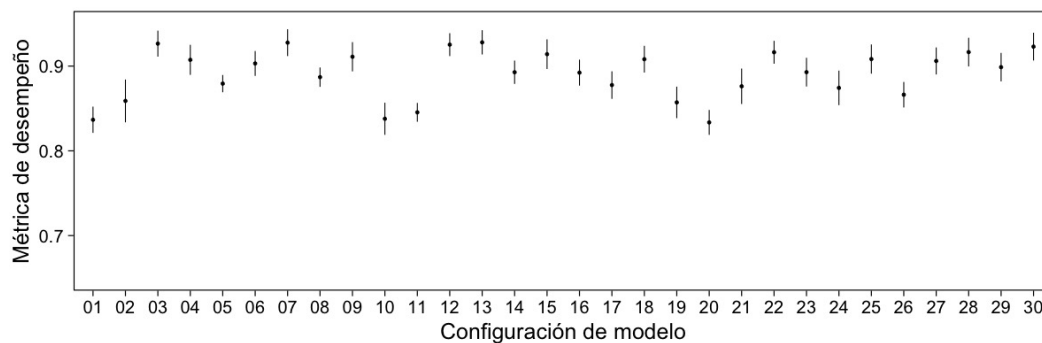


FIGURA 8. Estimación de desempeño con tres evaluaciones.

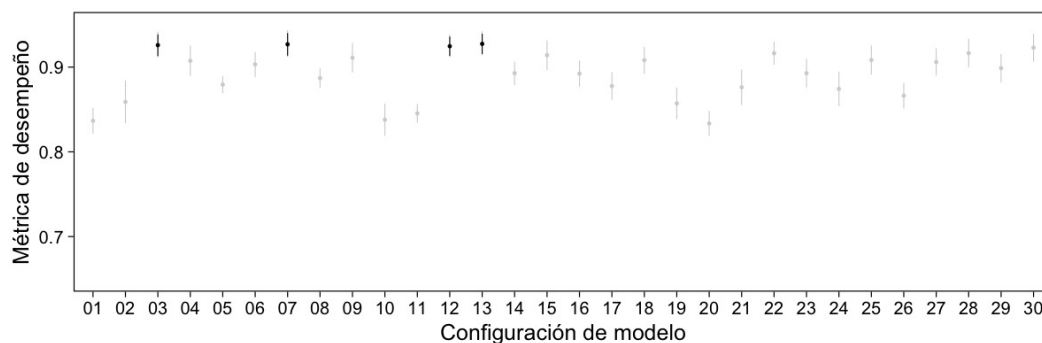


FIGURA 9. Estimación de desempeño con cuatro evaluaciones.

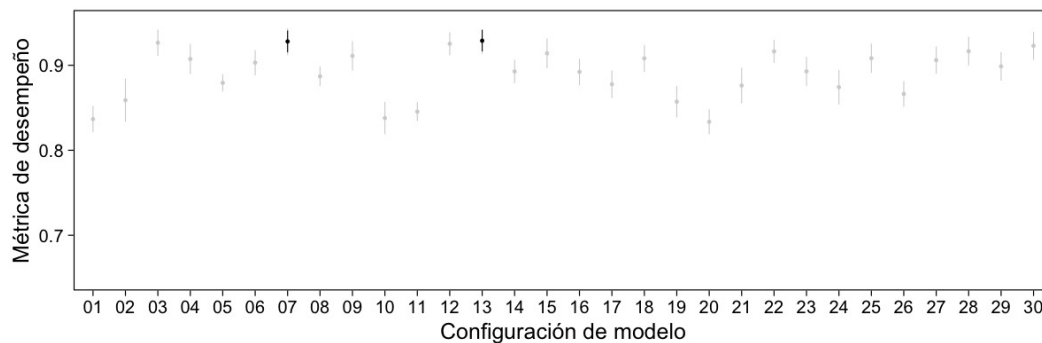


FIGURA 10. Estimación de desempeño con cinco evaluaciones.

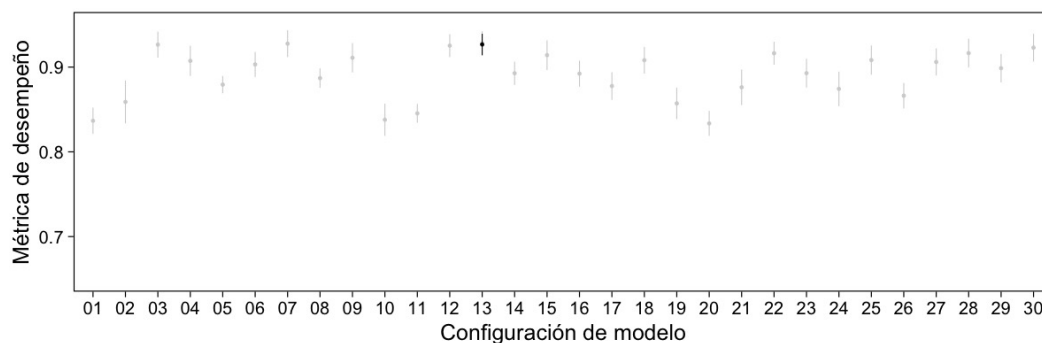


FIGURA 11. Estimación de desempeño con diez evaluaciones.

por qué? Es por esto que Greenwell [1] establece

$$\text{Boosting} \geq \text{RandomForest} > \text{Bagging} > \text{Arbol} . \quad (24)$$

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

$$\mathcal{I}_j^2(T) = \sum_{t=1}^{J-1} \hat{t}_t^2 I(v(t) = j) , \quad (25)$$

donde \hat{t}_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

$$\mathcal{I}_j^2 = \frac{1}{M} \sum_{m=1}^M \mathcal{I}_j^2(T_m) . \quad (26)$$

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](#), [17](#)
- [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. [14](#), [15](#)
- [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](#)