EST-25134: Aprendizaje Estadístico

Profesor: Alfredo Garbuno Iñigo — Primavera, 2023 — Modelos en ensamble.
Objetivo: En esta sección estudiaremos una forma de incorporar varios modelos para crear un modelo predictivo mas fuerte que un modelo individual. La estrategia está basada en una técnica de remuestreo que ya hemos estudiado previamente.
Lectura recomendada: Sección 8.2 de James et al. [3]. Capítulo 15 de Hastie et al. [2]. Capítulo 5 (*bagging*) y 7 (*random forest*) de Greenwell [1].

1. INTRODUCCIÓN

Anteriormente, vimos los modelos predictivos basados en `árboles de decisión` (regresión y clasificación). En esta sección del curso estudiaremos `distintas estrategias` para combinarlos para obtener un mejor modelo predictivo al costo de `interpretabilidad`. Algunos de estos modelos continúan representando el estado del arte en competencias como Kaggle para `datos tabulares`.

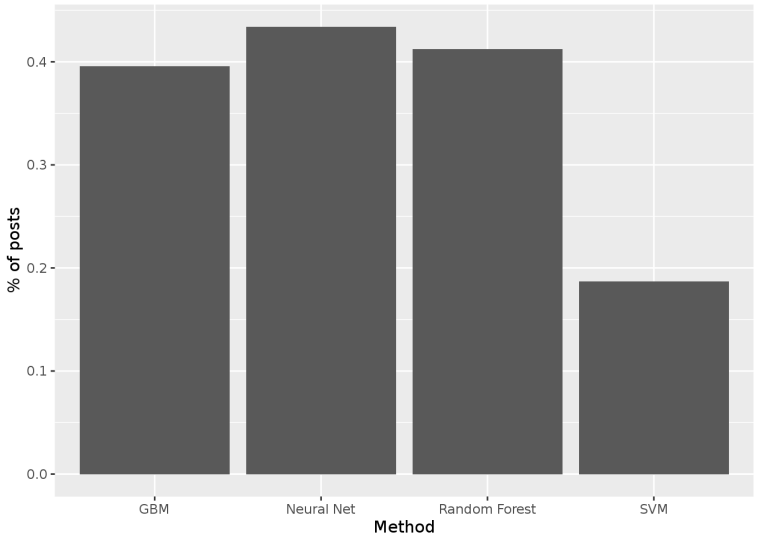


FIGURA 1. *Algoritmos que tienden a quedar en los primeros lugares en las competencias de Kaggle. Tomado de aquí.*

De igual manera se han publicado algunos reportes donde se confirma que en particular para `datos tabulares` modelos de ensamble basados en árboles tienen mejor capacidad predictiva.

[Submitted on 18 Jul 2022]

Why do tree-based models still outperform deep learning on tabular data?

Léo Grinsztajn (SODA), Edouard Oyallon (ISIR, CNRS), Gaël Varoquaux (SODA)

While deep learning has enabled tremendous progress on text and image datasets, its superiority on tabular data is not clear. We contribute extensive benchmarks of standard and novel deep learning methods as well as tree-based models such as XGBoost and Random Forests, across a large number of datasets and hyperparameter combinations. We define a standard set of 45 datasets from varied domains with clear characteristics of tabular data and a benchmarking methodology accounting for both fitting models and finding good hyperparameters. Results show that tree-based models remain state-of-the-art on medium-sized data (~10K samples) even without accounting for their superior speed. To understand this gap, we conduct an empirical investigation into the differing inductive biases of tree-based models and Neural Networks (NNs). This leads to a series of challenges which should guide researchers aiming to build tabular-specific NNs: 1. be robust to uninformative features, 2. preserve the orientation of the data, and 3. be able to easily learn irregular functions. To stimulate research on tabular architectures, we contribute a standard benchmark and raw data for baselines: every point of a 20 000 compute hours hyperparameter search for each learner.

[Submitted on 6 Jun 2021 (v1), last revised 23 Nov 2021 (this version, v2)]

Tabular Data: Deep Learning is Not All You Need

Ravid Shwartz-Ziv, Amitai Armon

A key element in solving real-life data science problems is selecting the types of models to use. Tree ensemble models (such as XGBoost) are usually recommended for classification and regression problems with tabular data. However, several deep learning models for tabular data have recently been proposed, claiming to outperform XGBoost for some use cases. This paper explores whether these deep models should be a recommended option for tabular data by rigorously comparing the new deep models to XGBoost on various datasets. In addition to systematically comparing their performance, we consider the tuning and computation they require. Our study shows that XGBoost outperforms these deep models across the datasets, including the datasets used in the papers that proposed the deep models. We also demonstrate that XGBoost requires much less tuning. On the positive side, we show that an ensemble of deep models and XGBoost performs better on these datasets than XGBoost alone.

2. REMUESTREO O BOOTSTRAP

Utilizar técnicas de remuestreo nos permite cuantificar la variabilidad de un estimador estadístico sin necesidad de invocar un régimen asintótico para el procedimiento. Asimismo, nos permite controlar, hasta cierto punto, la variabilidad de nuestros estimadores.[†]

[†]: Mas información de esto en el curso de `EST-24107: Simulación`.

Por ejemplo, consideremos la situación en donde tenemos una muestra de n observaciones $Z_1, \ldots, Z_n$ las cuales provienen de una distribución con varianza σ^2. Es fácil demostrar que la varianza de la media $\bar{Z}_n$ tiene una varianza σ^2/n.

2.0.1. Importante: Esto quiere decir, que podemos 1) generar muestras, 2) promediar y, entonces, reducimos la varianza estimada!

2.0.2. Para pensar: Usualmente no tenemos acceso al proceso generador de datos (ya sea $\mathbb{P}_{X,Y}$ ó $\mathbb{P}_X$). ¿Qué estrategia podemos utilizar?

2.1. Bootstrap

Podemos utilizar la muestra $z_1, \ldots, z_n \overset{\text{iid}}{\sim} \pi$ como un *proxy* de la población de la cual queremos generar observaciones. En este sentido, consideramos que la función de acumulación empírica (`ECDF`, por sus siglas en inglés) es un *buen* estimador de la función de probabilidad (ó `CDF` por sus siglas en inglés)

$$\pi[X \leq x] \approx \hat{\pi}_n[X \leq x] = \frac{1}{n}\sum_{i=1}^{n} I_{[z_i \leq x]}\,. \tag{1}$$

Con mi muestra, entonces, podemos calcular algún estimador de un característica poblacional de interés

$$\hat{\theta}_n = t(z_1, \ldots, z_n)\,. \tag{2}$$

Con este procedimiento podemos generar B conjuntos de datos

$$z_1^{(b)}, \ldots, z_n^{(b)} \overset{\text{iid}}{\sim} \hat{\pi}_n\,, \qquad b = 1, \ldots, B\,, \tag{3}$$

para obtener una colección de estimadores $\hat{\theta}_n^{(b)} = t(z_1^{(b)}, \ldots, z_n^{(b)})$ y, a través de un promedio, obtener un estimador

$$\bar{\theta}_{B,n}^{(\mathsf{bag})} = \frac{1}{B}\sum_{b=1}^{B} \hat{\theta}_n^{(b)}, \tag{4}$$

con varianza que se reduce a una tasa $1/B$.

El muestreo $z_1^{(b)}, \ldots, z_n^{(b)} \overset{\mathsf{iid}}{\sim} \hat{\pi}_n$ implica tomar muestras **con** reemplazo del conjunto de datos observado. Nota que las remuestras son del mismo tamaño que la muestra original. Es decir, cada remuestra b tiene n observaciones. Como el procedimiento es con reemplazo, esto puede ocasionar que pueda haber algunas observaciones que se repitan en la remuestra.

2.2. Ejemplo: Suavizadores

La estrategia de remuestreo nos puede ayudar a cuantificar la estabilidad de ciertos estimadores. Por ejemplo, consideremos los datos que teníamos sobre el ingreso para un conjunto de 150 observaciones. El interés es construir un suavizador que relacione `Edad` con `Ingreso`.

```
# A tibble: 150 × 5
   year   age  wage education        hi.income
  <int> <dbl> <dbl> <fct>                <dbl>
1  2003    53  81.6 4. College Grad          0
2  2008    50  82.7 4. College Grad          0
3  2006    35 155.  4. College Grad          0
# ... with 147 more rows
#  Use 'print(n = ...)' to see more rows
```

Utilizaremos un suavizador de *splines* con 15 grados de libertad, ver Fig. 2.

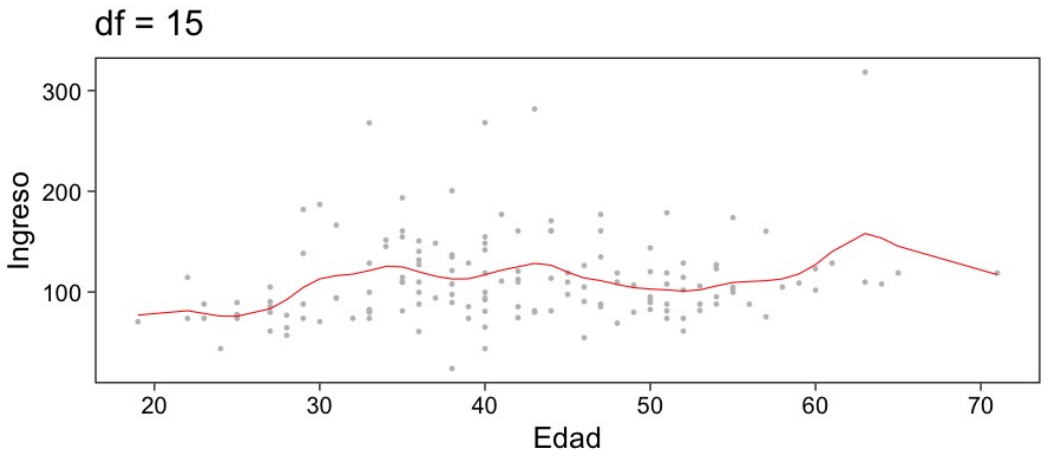


FIGURA 2. *Suavizador por* splines *con 15 grados de libertad.*

A través de remuestreo podemos cuantificar la estabilidad de dicha estimación, ver Fig. 3. La estabilidad también la podemos graficar por medio de intervalos de confianza. Ver Fig. 4.

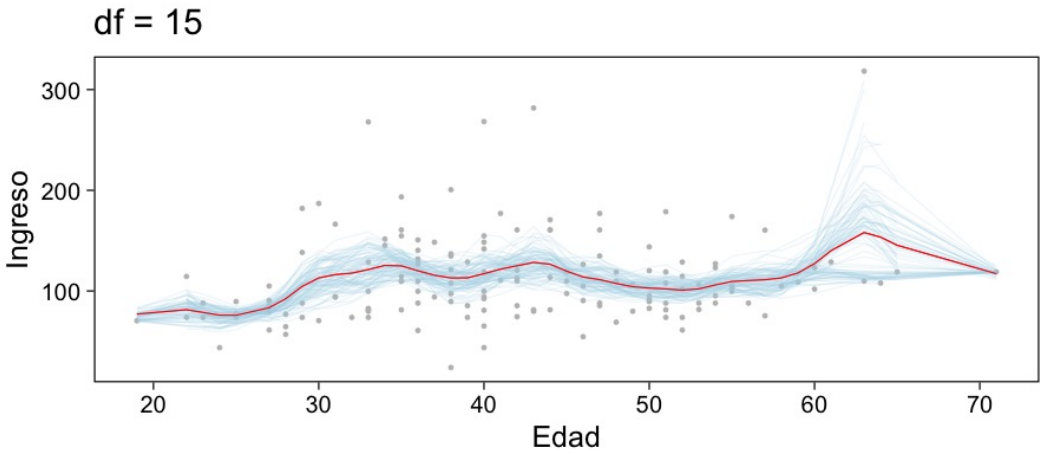


FIGURA 3. *Suavizador por* splines *con 15 grados de libertad, réplicas con remuestreo.*

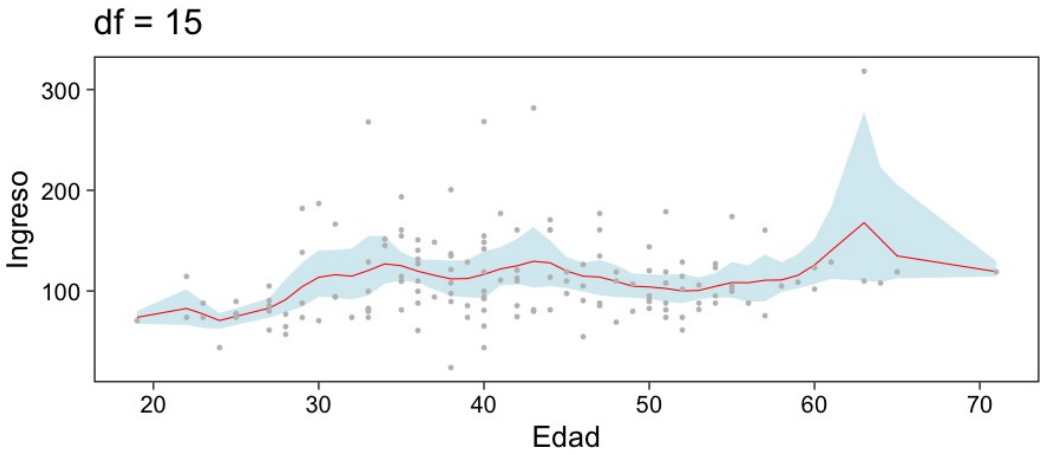


FIGURA 4. *Suavizador por* splines *con 15 grados de libertad, réplicas con remuestreo.*

3. BOOTSTRAPPED AGGREGATION: BAGGING

En el contexto de modelado predictivo nos interesa `estimar la relación` que existe entre atributos x y una respuesta de interés y por medio de una función $f : \mathcal{X} \mapsto \mathcal{Y}$. Dicho estimador, lo denotamos por $\hat{f}_n$ haciendo énfasis en que ha sido construido con una muestra de tamaño n. Recordemos que este estimador en el contexto de modelado predictivo es resultado de un problema de optimización con una función de pérdida adecuada.

Si utilizamos *bootstrap*, para cada uno de los B conjuntos de entrenamiento estimamos $\hat{f}_n^{(b)}$ con $b = 1, \ldots, B$ para poder hacer predicciones por medio de

$$\hat{f}_{B,n}^{(\mathsf{bag})}(x) = \frac{1}{B} \sum_{b=1}^{B} \hat{f}_n^{(b)}(x)\,, \tag{5}$$

esto lo llamamos `bootstrap aggregation` o `bagging`.

3.1. En problemas de clasificación

Para problemas de clasificación podemos considerar las predicciones de cada uno de los modelos $\hat{f}_n^{(b)}$ y tomar la clase con `más votos` dentro del conjunto de B predictores.

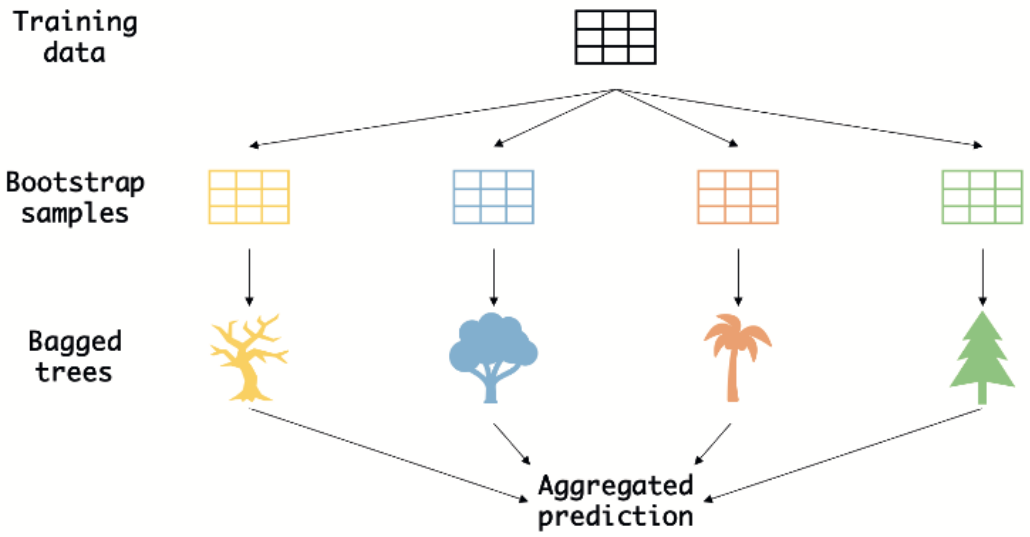


FIGURA 5. *Ilustración esquemática de* bagging *por medio de árboles de decisión. Tomada de [1].*

3.2. Error de generalización

- Usando *bootstrap* entrenamos con cada uno de los conjuntos de datos remuestreados.
- Cada conjunto remuestreado utiliza, en promedio, 2/3 de los datos originales.
- El conjunto no utilizado lo llamamos `conjunto fuera de bolsa` (*out-of-bag*, `OOB`).
- Podemos obtener predicciones para cada observación $i = 1, \ldots, n$ cuando se encuentra en algún conjunto `OOB`. En promedio, tenemos $B/3$ predicciones para cada observación, las cuales podemos promediar para obtener la predicción final.
- Esto es un estimador de `LOO-CV` utilizando `bagging`.

3.3. Observaciones

- El estimador $\hat{f}_{B,n}^{(\text{bag})}$ es un estimador Monte Carlo. ¿De qué?
- El estimador $\hat{f}_{B,n}^{(\text{bag})} \to \hat{f}_n$ con $B \to \infty$ en cada uno de los puntos a evaluar x.
- Cuando los atributos están altamente correlacionados los árboles de decisión pueden presentar varianza alta. `Ventaja:` en esta situación `bagging` puede suavizar la varianza y reducir el error de generalización.

3.4. Bagging, regresión y `MSE`

- Si estamos en tareas de regresión y medimos el error de generalización por medio de pérdida cuadrática obtenemos lo siguiente

$$\mathbb{E}\left(y - \hat{f}^*(x)\right)^2 \geq \mathbb{E}\left(y - \mathbb{E}\hat{f}^*(x)\right)^2, \tag{6}$$

  donde $\hat{f}^*$ es una estimación por medio de una remuestra y $\mathbb{E}\hat{f}^*$ es el valor esperado de las estimaciones de f utilizando remuestras.
- Por lo tanto, *bagging* podrá disminuir el `MSE`.

3.5. Bagging y clasificación

- En problemas de clasificación, no tenemos descomposición aditiva de sesgo y varianza. A menos que ...

- El uso de `bagging` puede hacer de un mal clasificador, algo todavía peor. Consideremos el caso siguiente.

3.5.1. Bagging *y clasificadores:* Supongamos que tenemos un clasificador binario que asigna $Y = 1$ para todo x con probabilidad 0.4. ¿Cuál es el la tasa de error de clasificación de este modelo? ¿Cuál sería la tasa de error de clasificación de un consenso con este modelo?

3.5.2. Bagging *y la sabiduría de las masas:* Supongamos que tenemos una colección de clasificadores independientes donde cada uno tiene una tasa de error de $\varepsilon < 0.5$, y sea

$$S_1(x) = \sum_{b=1}^{B} I[G^{(b)}(x) = 1]\,, \tag{7}$$

el voto por consenso de que la instancia x pertenezca a la clase 1. Dado que los clasificadores son independientes entonces

$$S_1(x) \sim \mathsf{Binomial}(B, 1-\varepsilon)\,, \tag{8}$$

donde

$$\mathbb{P}(\text{ clasificación correcta }) = \mathbb{P}(S_1 > B/2) \approx 1\,, \tag{9}$$

con B suficientemente grande.

El resultado anterior se conoce como `Sabiduría de las masas` en donde se asume que cada clasificador es un clasificador `débil`. Con tasa de error ligeramente menor al azar. Para que el consenso de dichos clasificadores tenga buenos resultados se necesita, además, que los clasificadores sean `independientes`.

3.6. Observaciones

- Utilizar `bagging` en un problema de clasificación con árboles no es un procedimiento que utilice árboles independientes. Por lo tanto no hay garantía de que el consenso mejore el error de clasificación.
- En general para `bagging` estamos dispuestos a usar modelos de `alta varianza` puesto que el remuestreo se encarga de ayudarnos a contenerla.

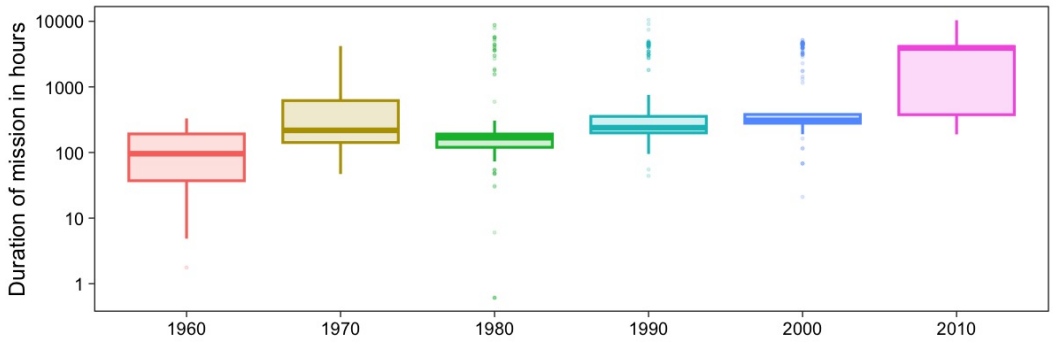
4. APLICACIÓN: MISIONES DE ASTRONAUTAS

Ejemplo tomado de: Bagging with tidymodels and #TidyTuesday astronaut missions. Cargamos la información y exploramos cuál es la nave (*¿spacecraft?*) que mas se utiliza en misiones espaciales.

```
astronauts ▷
  count(in_orbit, sort = TRUE) ▷
  print(n = 3)
```

```
# A tibble: 289 × 2
  in_orbit      n
  <chr>     <int>
1 ISS         174
2 Mir          71
3 Salyut 6     24
# ... with 286 more rows
#  Use 'print(n = ...)' to see more rows
```

Podemos explorar cómo ha cambiado la duración de las misiones a lo largo del tiempo.



Objetivo: predecir la duración de una misión espacial en función de algunas características de la misión.

```
# A tibble: 1,270 × 5
  hours_mission military_civilian occupation year_of_mission in_orbit
          <dbl> <chr>             <chr>                 <dbl> <chr>
1         0.571 military          pilot                  1961 Vostok 2
2         3.22  military          pilot                  1961 Vostok 2
3         1.61  military          pilot                  1962 MA-6
4         5.36  military          psp                    1998 STS
5         1.61  military          pilot                  1962 Mercury-...Atlas
# ... with 1,265 more rows
#  Use 'print(n = ...)' to see more rows
```

4.1. Proceso de modelado

```
set.seed(123)
astro_split ← initial_split(astronauts_df, strata = hours_mission)
astro_train ← training(astro_split)
astro_test ← testing(astro_split)
```

```
astro_recipe ← recipe(hours_mission ~ ., data = astro_train) ▷
  step_other(occupation, in_orbit,
             threshold = 0.005, other = "Other"
             ) ▷
  step_dummy(all_nominal_predictors())
```

```
astro_wf ← workflow() ▷
  add_recipe(astro_recipe)

astro_wf
```

```
 Workflow
Preprocessor: Recipe
Model: None

 Preprocessor
2 Recipe Steps•

```

```
 step_other()•
 step_dummy()
```

4.2. Ajuste de bagging con árboles de clasificación

Especificamos el modelo que utilizaremos

```
library(baguette)

tree_spec ← bag_tree() ▷
  set_engine("rpart", times = 100) ▷
  set_mode("regression")

tree_spec
```

```
Bagged Decision Tree Model Specification (regression)

Main Arguments:
  cost_complexity = 0
  min_n = 2

Engine-Specific Arguments:
  times = 100

Computational engine: rpart
```

Ajustamos el modelo a nuestros de datos de entrenamiento.

```
tree_rs ← astro_wf ▷
  add_model(tree_spec) ▷
  fit(astro_train)

tree_rs
```

```
== Workflow [trained] ==========================================================
Preprocessor: Recipe
Model: bag_tree()

-- Preprocessor ----------------------------------------------------------------
2 Recipe Steps

- step_other()
- step_dummy()

-- Model -----------------------------------------------------------------------
Bagged CART (regression with 100 members)

Variable importance scores include:

# A tibble: 13 × 4
term                              value std.error  used
<chr>                             <dbl>     <dbl> <int>
1 year_of_mission                   872.      12.0    100
2 in_orbit_Other                    592.      27.9    100
3 in_orbit_STS                      339.      13.3    100
4 occupation_flight.engineer        233.      13.5    100
```

```
5 in_orbit_Mir                      133.       9.35   100
6 occupation_pilot                  130.       9.02   100
7 in_orbit_Salyut                   103.       4.64   100
8 occupation_msp                     99.4      4.63   100
9 occupation_other..space.tourist.   44.2      2.24   100
10 military_civilian_military        39.6      1.96   100
11 occupation_psp                    22.3      2.57   100
12 in_orbit_Mir.EP                   18.8      1.57    95
13 occupation_Other                  17.5      1.17    88
```

Realizamos predicciones en nuestro conjunto de prueba para poder reportar capacidad predictiva.

```
test_rs ← astro_test ▷
  bind_cols(predict(tree_rs, astro_test)) ▷
  rename(.pred_tree = .pred)

test_rs ▷ print(n = 5, width = 73)
```

```
# A tibble: 318 × 6
  hours_mission military_civilian occupation year_of_...¹mi in_...²or ....³pred
          <dbl> <chr>             <chr>              <dbl> <chr>     <dbl>
1          3.22 military          pilot               1961 ...Vostok    1.76
2          1.61 military          pilot               1962 MA-6         3.19
3          5.36 military          psp                 1998 STS          5.62
4          6.05 military          pilot               1970 Soyuz 9      5.34
5          5.93 military          commander           1974 Soyuz ...    6.08
# ... with 313 more rows, and abbreviated variable names ¹year_of_mission,
# ²  in_orbit, ³.pred_tree
#  Use `print(n = ...)` to see more rows
```

Por ejemplo, podemos reportar un conjunto de métricas

```
test_rs ▷
  metrics(hours_mission, .pred_tree)
```

```
# A tibble: 3 × 3
  .metric .estimator .estimate
  <chr>   <chr>          <dbl>
1 rmse    standard       0.682
2 rsq     standard       0.756
3 mae     standard       0.360
```

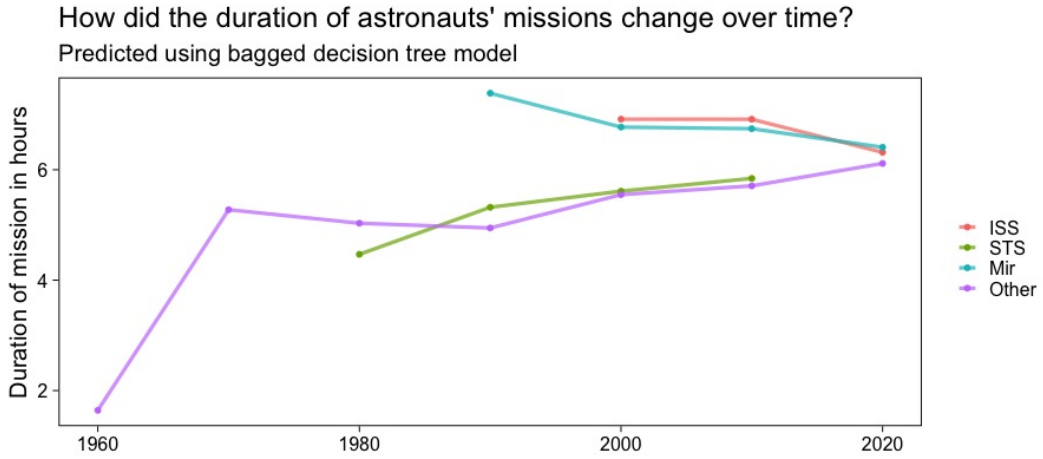
Podemos jugar con el modelo para tratar de entender la respuesta del modelo a situaciones de interés.

```
new_astronauts ← crossing(
  in_orbit = fct_inorder(c("ISS", "STS", "Mir", "Other")),
  military_civilian = "civilian",
  occupation = "Other",
  year_of_mission = seq(1960, 2020, by = 10)
) ▷
  filter(
    !(in_orbit == "ISS" & year_of_mission < 2000),
    !(in_orbit == "Mir" & year_of_mission < 1990),
```

```
    !(in_orbit == "STS" & year_of_mission > 2010),
    !(in_orbit == "STS" & year_of_mission < 1980)
  )

new_astronauts
```

```
# A tibble: 18 × 4
   in_orbit military_civilian occupation year_of_mission
   <fct>    <chr>             <chr>                 <dbl>
 1 ISS      civilian          Other                  2000
 2 ISS      civilian          Other                  2010
 3 ISS      civilian          Other                  2020
 4 STS      civilian          Other                  1980
 5 STS      civilian          Other                  1990
 6 STS      civilian          Other                  2000
 7 STS      civilian          Other                  2010
 8 Mir      civilian          Other                  1990
 9 Mir      civilian          Other                  2000
10 Mir      civilian          Other                  2010
11 Mir      civilian          Other                  2020
12 Other    civilian          Other                  1960
13 Other    civilian          Other                  1970
14 Other    civilian          Other                  1980
15 Other    civilian          Other                  1990
16 Other    civilian          Other                  2000
17 Other    civilian          Other                  2010
18 Other    civilian          Other                  2020
```



5. BOSQUES ALEATORIOS

El modelo propuesto de Bosques aleatorios (`RF` por sus siglas en inglés) ayuda a decorrelacionar un conjunto de árboles. Para lograr esto seguimos utilizando remuestreo para seleccionar conjuntos de datos de entrenamiento. Al mismo tiempo, con cada conjunto de remuestras, utilizamos un conjunto de m predictores al azar para entrenar. Esto es, utilizamos para cada remuestra, un subconjunto distinto de predictores para entrenar un árbol.

Usualmente consideramos $m \approx \sqrt{p}$. Esto permite restringir el espacio de búsqueda y dejar de utilizar consistentemente los mismos predictores en cada remuestra.

5.1. Motivación

Si consideramos la situación donde tenemos B variables iid cada una con varianza σ^2 entonces el promedio tendrá varianza igual σ^2/B. Si las variables son sólo id (números aleatorios de la misma población) con correlación positiva ρ , entonces el promedio tendrá varianza igual a

$$\rho\sigma^2 + \frac{1-\rho}{B}\sigma^2 \,. \tag{10}$$

Incluso aunque tomemos un número suficiente de árboles para controlar el segundo término, el primer término no desvanece con $B \to \infty$. Es por esto que bosques aleatorios busca reducir la correlación entre árboles al permitir que se ajusten a conjuntos aleatorios (en observaciones y predictores) por medio de remuestreo.

5.2. Sobre-ajuste

- El consenso de votos tiende a ser robusto contra sobre-ajustar y si utilizamos una B (el número de árboles) suficientemente grande estabilizamos la variabilidad del error de generalización.
- Usualmente tenemos problemas de sobre-ajuste cuando el número de predictores es alto y el número de predictores relevantes para la predicción es pequeño.

5.3. Análisis de ajuste

La predicción de un bosque aleatorio se realiza por medio de

$$\hat{f}_{\mathsf{RF}}(x) = \frac{1}{B}\sum_{b=1}^{B} \hat{f}^{(b)}(x) = \frac{1}{B}\sum_{b=1}^{B} T\left(x; \Theta(\mathcal{D}_n^{(b)})\right) \,, \tag{11}$$

donde $T(x;\Theta)$ denota la predicción de un árbol utilizando los parámetros (variables de selección, puntos de corte) Θ. La notación $\Theta(\mathcal{D}_n)$ hace énfasis en que los parámetros que gobiernan el árbol fueron escogidos utilizando el conjunto de datos $\mathcal{D}_n$. El término $\mathcal{D}_n^{(b)}$ hace énfasis en que el conjunto de entrenamiento es una remuestra del conjunto original.

El predictor tiende a satisfacer la siguiente igualdad (ley de los grandes números, $B \to \infty$)

$$\hat{f}_{\mathsf{RF}}(x) = \mathbb{E}_{\Theta|\mathcal{D}_n} T\left(x; \Theta(\mathcal{D}_n)\right) \,, \tag{12}$$

donde hacemos énfasis en que es un valor esperado condicional en los datos de entrenamiento.

Nos interesa evaluar el `error estándar` de dicho estimador. Lo cual escribimos como

$$\mathsf{SE}\left(\hat{f}_{\mathsf{RF}}(x)\right)^2 = \mathbb{V}\left(\hat{f}_{\mathsf{RF}}(x)\right) = \rho(x)\cdot\sigma^2(x) \,, \tag{13}$$

donde:

- $\rho(x)$ es la correlación entre dos árboles

$$\rho(x) = \mathsf{Corr}\left[T\left(x;\Theta_i(\mathcal{D}_n)\right), T\left(x;\Theta_j(\mathcal{D}_n)\right)\right] \,. \tag{14}$$

- $\sigma^2(x)$ es la varianza de cualquier árbol

$$\sigma^2(x) = \mathbb{V}\left(T\left(x;\Theta(\mathcal{D}_n)\right)\right) \,. \tag{15}$$

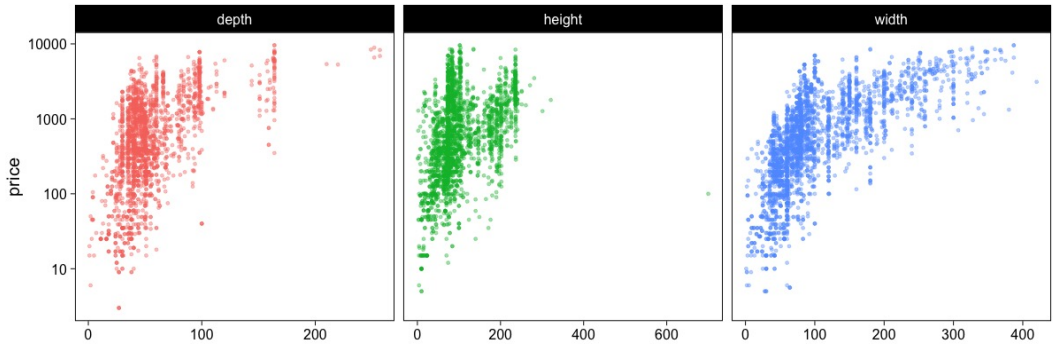


FIGURA 6. *Relación del precio con las dimensiones del producto.*

6. APLICACIÓN: PREDICCIÓN DE PRECIOS IKEA

Ejemplo tomado de: Tune random forests for #TidyTuesday IKEA prices.
Los datos que tenemos disponibles son los siguientes.

```
ikea_df ← ikea ▷
  select(price, name, category, depth, height, width) ▷
  mutate(price = log10(price)) ▷
  mutate_if(is.character, factor)

ikea_df ▷ print(n = 5)
```

```
# A tibble: 3,694 × 6
  price name                  category      depth height width
  <dbl> <fct>                 <fct>         <dbl>  <dbl> <dbl>
1  2.42 FREKVENS              Bar furniture    NA     99    51
2  3.00 NORDVIKEN             Bar furniture    NA    105    80
3  3.32 NORDVIKEN / NORDVIKEN Bar furniture    NA     NA    NA
4  1.84 STIG                  Bar furniture    50    100    60
5  2.35 NORBERG               Bar furniture    60     43    74
# ... with 3,689 more rows
#  Use 'print(n = ...)' to see more rows
```

6.1. Preparación de datos

```
set.seed(123)
ikea_split ← initial_split(ikea_df, strata = price)
ikea_train ← training(ikea_split)
ikea_test ← testing(ikea_split)

set.seed(234)
ikea_folds ← vfold_cv(ikea_train, strata = price)
```

```
library(textrecipes)
ranger_recipe ←
  recipe(formula = price ~ ., data = ikea_train) ▷
  step_other(name, category, threshold = 0.01) ▷
  step_clean_levels(name, category) ▷
  step_impute_knn(depth, height, width)
```

6.2. Especificación del modelo

```
ranger_spec ←
  rand_forest(mtry = tune(), min_n = tune(), trees = 1000) ▷
  set_mode("regression") ▷
  set_engine("ranger")

ranger_workflow ←
  workflow() ▷
  add_recipe(ranger_recipe) ▷
  add_model(ranger_spec)
```

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

```
set.seed(8577)
system.time(
ranger_tune ←
  tune_grid(ranger_workflow,
            resamples = ikea_folds,
            grid = 11,
            control = control_grid(
              parallel_over = "resamples",
              verbose = TRUE))
)
```

```
 user  system elapsed
0.776   0.026  40.034
```

```
show_best(ranger_tune, metric = "rmse")
```

```
# A tibble: 5 × 8
   mtry min_n .metric .estimator  mean     n std_err .config
  <int> <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
1     2     4 rmse    standard   0.323    10 0.00602 Preprocessor1_Model10
2     5     6 rmse    standard   0.331    10 0.00562 Preprocessor1_Model06
3     4    10 rmse    standard   0.332    10 0.00570 Preprocessor1_Model05
4     3    18 rmse    standard   0.339    10 0.00569 Preprocessor1_Model01
5     2    21 rmse    standard   0.343    10 0.00561 Preprocessor1_Model08
```

```
show_best(ranger_tune, metric = "rsq")
```

```
# A tibble: 5 × 8
   mtry min_n .metric .estimator  mean     n std_err .config
  <int> <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
1     2     4 rsq     standard   0.752    10  0.0106 Preprocessor1_Model10
2     5     6 rsq     standard   0.740    10  0.0100 Preprocessor1_Model06
```

```
3     4    10 rsq     standard   0.738    10  0.0101 Preprocessor1_Model05
4     3    18 rsq     standard   0.728    10  0.0104 Preprocessor1_Model01
5     2    21 rsq     standard   0.723    10  0.0107 Preprocessor1_Model08
```

```
final_rf ← ranger_workflow ▷
  finalize_workflow(select_best(ranger_tune))

final_rf
```

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

-- Preprocessor ----------------------------------------------------------------
3 Recipe Steps

- step_other()
- step_clean_levels()
- step_impute_knn()

-- Model -----------------------------------------------------------------------
Random Forest Model Specification (regression)

Main Arguments:
  mtry = 2
  trees = 1000
  min_n = 4

Computational engine: ranger
```

La función de `last_fit` nos permite ajustar el modelo con todo el conjunto de entrenamiento y evaluar métricas de desempeño en el conjunto de prueba.

```
ikea_fit ← last_fit(final_rf, ikea_split)
ikea_fit ▷ select(c(-id, -.notes)) ▷ print(width = 75)
```

```
# A tibble: 1 × 4
  splits             .metrics         .predictions       .workflow
  <list>             <list>           <list>             <list>
1 <split [2770/924]> <tibble [2 × 4]> <tibble [924 × 4]> <workflow>
```

```
collect_metrics(ikea_fit)
```

```
# A tibble: 2 × 4
  .metric .estimator .estimate .config
  <chr>   <chr>          <dbl> <chr>
1 rmse    standard       0.318 Preprocessor1_Model1
2 rsq     standard       0.752 Preprocessor1_Model1
```

Utilizando la librería de `vip` podemos explorar cuáles son las variables mas importantes (mas delante hablaremos de esto) del modelo.

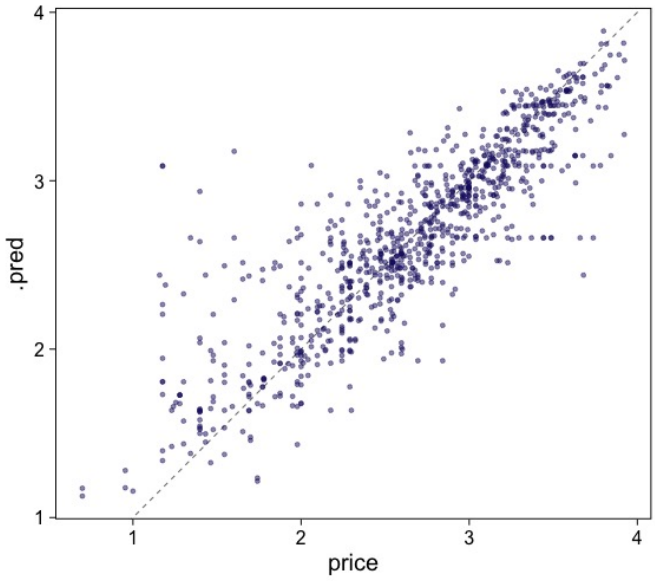
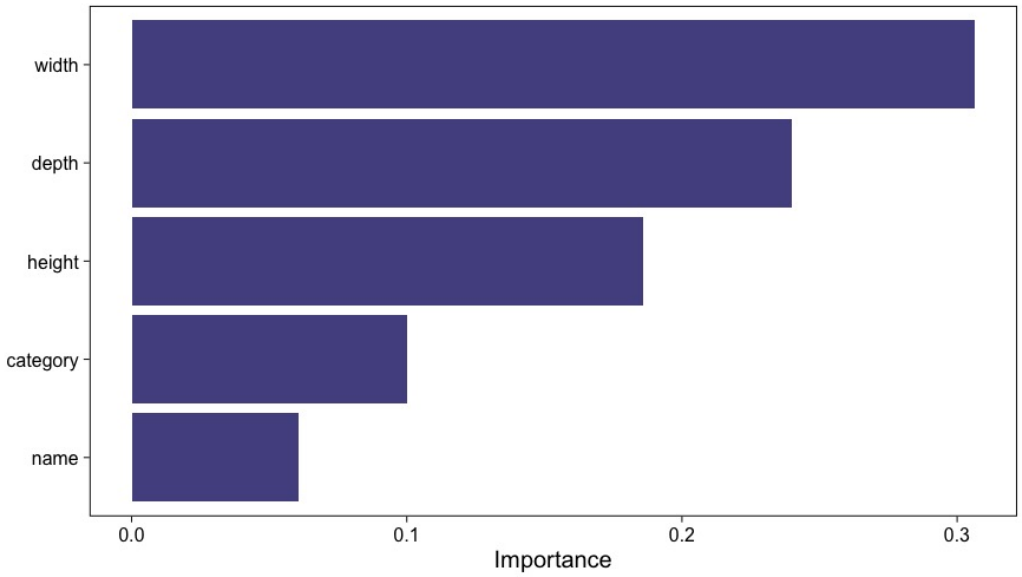


FIGURA 7. *Diagrama de dispersión entre predicciones y datos reales en el conjunto de prueba.*



6.3. Post-procesando el bosque

Lo que queremos es explorar los gráficos de desempeño conforme el tamaño del bosque aumenta. Pero la librería de `tidymodels` no nos permite acceso a estos elementos. Lo tenemos que tratar de manera especializada.

```
ranger_prep ← prep(ranger_recipe, training = ikea_train)
rf_model ← randomForest::randomForest(
                            price ∼., bake(ranger_prep, ikea_train),
                            mtry = 2, ntree = 1000, nodesize = 4)
```

¿Por qué necesitamos la receta de preparación de datos?

```
collect_forest_predictions ← function(model, data){
```

```
  ## Obten predicciones por arbol
  predictions ← predict(model, bake(ranger_prep, data), predict.all = TRUE)
  predictions$individual ▷
    ## obten tabla nrows x ncols = nobs x ntrees
    as_tibble() ▷
    mutate(observation = 1:nrow(data),
           ## agrega response
           truth = data$price) ▷
    ## obten tabla de nrows = nobs x ntrees
    pivot_longer(cols = 1:1000,
                 values_to = ".prediction", names_to = ".tree")  ▷
    group_by(observation) ▷
    ## reordena los arboles de manera aleatoria
    sample_frac(1, replace = FALSE) ▷
    ## calcula prediccion del bosque
    mutate(.estimate = cummean(.prediction),
           .order = 1:n()) ▷
    ungroup() ▷ select(c(-.prediction)) ▷
    nest(data = c(observation, truth, .estimate, .tree)) ▷
    ## calcula metrica de error
    mutate(results = map(data, function(x) { x ▷ rmse(truth, .estimate) })) ▷
    select(-data)
}
```

```
nexp ← 100; set.seed(108)
predictions_train ← tibble(.expid = 1:nexp) ▷
  mutate(.performance = map(## realiza remuestreo de orden de arboles
           .expid,
           ~collect_forest_predictions(rf_model, ikea_train)),
         .type = "training")
predictions_test ← tibble(.expid = 1:nexp) ▷
  mutate(.performance = map(## realiza remuestreo de orden de arboles
           .expid,
           ~collect_forest_predictions(rf_model, ikea_test)),
         .type = "testing")
```

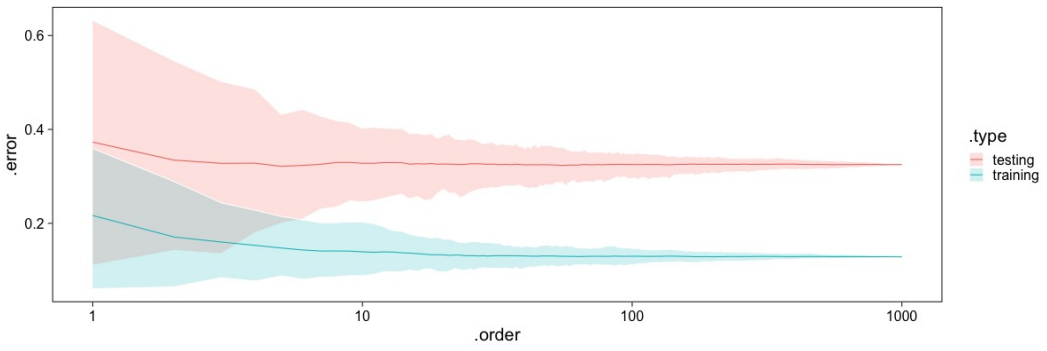


FIGURA 8. *Error predictivo utilizando bosque aleatorio.*

6.4. Post-procesamiento predictivo

¿Y si queremos encontrar un subconjunto de árboles del bosque para mejorar las predicciones en conjuntos de datos nuevos? ¿Qué estrategia hemos visto que nos puede ayudar con este objetivo?

```
predictions ← predict(rf_model, bake(ranger_prep, ikea_train), predict.all =
    TRUE)
```

```
trees_train ▷ print(n = 3, width = 75)
```

```
# A tibble: 924 × 1,001
     V1    V2    V3    V4    V5    V6    V7    V8    V9   V10   V11   V12
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1  2.75  2.18  2.69  3.22  2.78  2.43  2.58  1.84  3.16  2.12  2.69  2.42
2  2.47  2.11  2.44  2.06  2.87  2.80  2.32  2.70  2.67  1.40  2.67  2.54
3  2.11  2.11  2.11  2.06  2.29  2.14  2.15  2.18  2.20  2.11  2.42  2.14
# ... with 921 more rows, and 989 more variables: V13 <dbl>, V14 <dbl>,
#   V15 <dbl>, V16 <dbl>, V17 <dbl>, V18 <dbl>, V19 <dbl>, V20 <dbl>,
#   V21 <dbl>, V22 <dbl>, V23 <dbl>, V24 <dbl>, V25 <dbl>, V26 <dbl>,
#   V27 <dbl>, V28 <dbl>, V29 <dbl>, V30 <dbl>, V31 <dbl>, V32 <dbl>,
#   V33 <dbl>, V34 <dbl>, V35 <dbl>, V36 <dbl>, V37 <dbl>, V38 <dbl>,
#   V39 <dbl>, V40 <dbl>, V41 <dbl>, V42 <dbl>, V43 <dbl>, V44 <dbl>,
#   V45 <dbl>, V46 <dbl>, V47 <dbl>, V48 <dbl>, V49 <dbl>, V50 <dbl>, ...
#  Use `print(n = ...)` to see more rows, and `colnames()` to see all variable
    names
```

```
lasso_spec ← linear_reg(penalty = tune(), mixture = 1) ▷
  set_engine("glmnet") ▷
  set_mode("regression")
```

```
lasso_rec ← recipe(price ~ ., data = trees_train)
```

```
set.seed(108727)
forest_boot ← vfold_cv(trees_train, v = 10)
```

```
lasso_wf ← workflow() ▷
  add_recipe(lasso_rec) ▷
  add_model(lasso_spec)

lasso_grid ← lasso_wf ▷
  tune_grid(
    resamples = forest_boot,
    grid = 50,
    control = control_grid(verbose = FALSE)
  )
```

```
lasso_grid ▷
  collect_metrics() ▷
  print(n = 3, width = 75)
```

```
# A tibble: 100 × 7
   penalty .metric .estimator  mean     n std_err .config
     <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
1 1.26e-10 rmse    standard   0.312    10  0.0137 Preprocessor1_Model01
2 1.26e-10 rsq     standard   0.762    10  0.0105 Preprocessor1_Model01
3 1.90e-10 rmse    standard   0.312    10  0.0137 Preprocessor1_Model02
# ... with 97 more rows
#  Use `print(n = ...)` to see more rows
```

```
lowest_rmse ← lasso_grid ▷
  select_best("rmse")

final_lasso ← finalize_workflow(
  lasso_wf,
  lowest_rmse
)
```

```
active_trees ← final_lasso ▷
  fit(trees_train) ▷
  broom::tidy() ▷
  filter(estimate != 0) ▷
  mutate(.tree = term, beta = estimate) ▷
  select(c(.tree, beta))

active_trees ▷ print(n = 5)
```

```
# A tibble: 65 × 2
  .tree           beta
  <chr>          <dbl>
1 (Intercept) -0.0356
2 V7           0.0471
3 V13          0.0344
4 V17          0.0502
5 V27          0.00622
# ... with 60 more rows
#  Use `print(n = ...)` to see more rows
```

```
collect_dforest_predictions ← function(model, data, active_trees){
  intercept ← active_trees$beta[1]
  predictions ← predict(model, bake(ranger_prep, data), predict.all = TRUE)
  predictions$individual ▷
    as_tibble() ▷
    mutate(observation = 1:nrow(data),
           truth = data$price) ▷
    pivot_longer(cols = 1:1000,
                 values_to = ".prediction", names_to = ".tree") ▷
    right_join(active_trees ▷ filter(.tree != "(Intercept)"), by = ".tree") ▷
    filter(complete.cases(beta)) ▷
    group_by(observation) ▷
    sample_frac(1, replace = FALSE) ▷
    mutate(.estimate = intercept + cumsum(beta * .prediction),
           .order = 1:n()) ▷
    ungroup() ▷ select(c(-.prediction, -beta)) ▷
    nest(data = c(observation, truth, .estimate, .tree)) ▷
```

```
    mutate(results = map(data, function(x) { x ▷ rmse(truth, .estimate) })) ▷
    select(-data)
}
```

```
dpredictions_train ← collect_dforest_predictions(rf_model, ikea_train, active_
    trees)
dpredictions_test  ← collect_dforest_predictions(rf_model, ikea_test, active_
    trees)
```

```
nexp ← 100; set.seed(108)
dpredictions_train ← tibble(.expid = 1:nexp) ▷
  mutate(.performance = map(## realiza remuestreo de orden de arboles
           .expid,
           ~collect_dforest_predictions(rf_model, ikea_train, active_trees)),
         .type = "dtraining")
dpredictions_test ← tibble(.expid = 1:nexp) ▷
  mutate(.performance = map(## realiza remuestreo de orden de arboles
           .expid,
           ~collect_dforest_predictions(rf_model, ikea_test, active_trees)),
         .type = "dtesting")
```

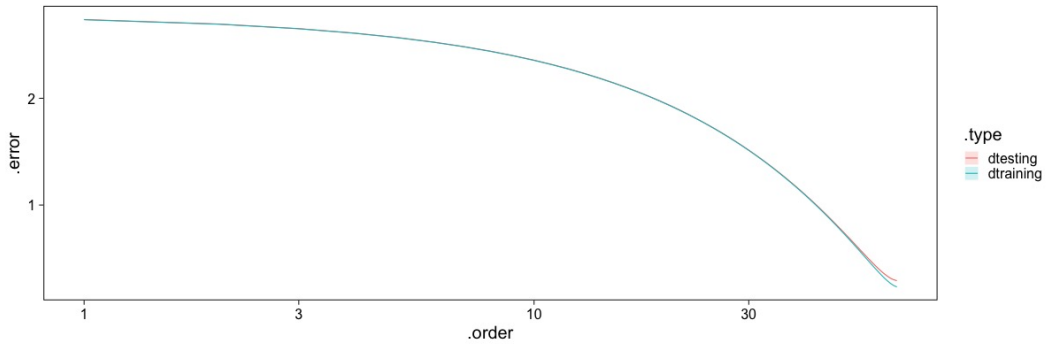
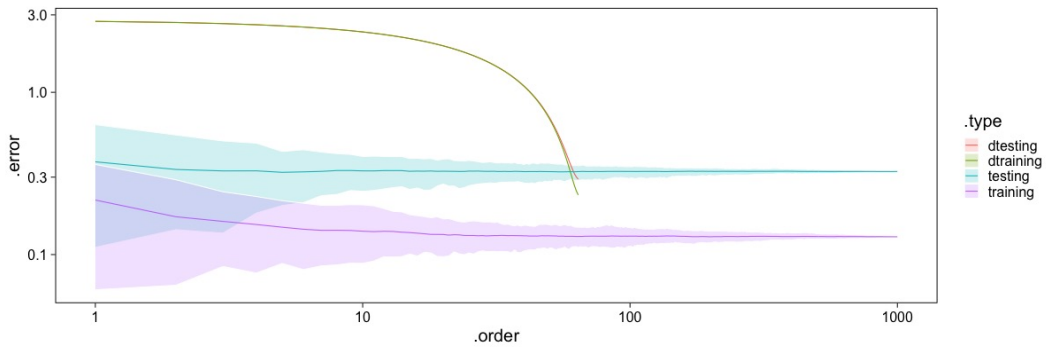


FIGURA 9. *Error predictivo utilizando bosque aleatorio.*



7. CONCLUSIONES

- Los bosques aleatorios son uno de los métodos más generales de predicción.
- Son fáciles de entrenar, usualmente ajustando dos parámetros por validación cruzada.

- Heredan ventajas de los árboles. Por ejemplo, las predicciones siempre se encuentran en el rango de las observaciones.
- Pueden ser lentos en predicción.
- Tienen capacidad de extrapolación limitada.

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, 5

[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] G. James, D. Witten, T. Hastie, and R. Tibshirani. *An Introduction to Statistical Learning: With Applications in R.* Springer Texts in Statistics. Springer US, New York, NY, 2021. 1