EST-25134: Aprendizaje Estadístico

Profesor: Alfredo Garbuno Iñigo — Primavera, 2023 — Ensamble de modelos.
Objetivo: En competencias de modelado predictivo es usual observar que combinaciones de modelos logran los mejores resultados. En esta sección del curso explicaremos por qué y presentaremos una implementación de esto de la librería `tidymodels`.
Lectura recomendada: Capítulo 10 de [2] y capítulo 15 de [3]. La última sección del libro de [1] menciona las garantías estadísticas que presentamos en esta clase.

1. INTRODUCCIÓN

Hemos discutido ya sobre distintos modelos y cómo cada modelo tiene distintas necesidades para pre-procesar los datos antes de realizarse el ajuste. En el capítulo de 10 de Kuhn and Johnson [2] se ajustan varios modelos para predecir la capacidad de compresión de mezclas de concreto en función de los ingredientes que se utilizan para cada mezcla. Las preguntas que resolveremos en esta sección son:

> ¿Cómo podemos comparar distintos modelos entre si? ¿Cómo podemos utilizar un flujo de trabajo que nos ayude a hacerlo de manera eficiente?

Los datos que usaremos para ilustrar estos conceptos son los mismos que usan [2] donde lo que nos interesa es predecir `compressive_strength` y las unidades son kilogramos por metro cúbico.

```
library(tidymodels)
data(concrete, package = "modeldata")
concrete ▷ print(n = 3, width = 70)
```

```
# A tibble: 1,030 × 9
  cement blast_...¹f fly_ash water ...²super ...³coars fine_...   age ...compr
   <dbl>     <dbl>   <dbl> <dbl>   <dbl>   <dbl>   <dbl> <int>   <dbl>
1   540         0       0   162     2.5    1040     676    28    80.0
2   540         0       0   162     2.5    1055     676    28    61.9
3   332.      142.      0   228     0       932     594   270    40.3
# ... with 1,027 more rows, and abbreviated variable names
# ¹  blast_furnace_slag, ²superplasticizer, ³coarse_aggregate,
#   fine_aggregate, compressive_strength
#  Use ‘print(n = ...)‘ to see more rows
```

En particular para estos datos tenemos mezclas que se probaron varias veces por lo tanto reduciremos un poco esta multiplicidad.

```
concrete ←
  concrete ▷
  group_by(across(-compressive_strength)) ▷
  summarize(compressive_strength = mean(compressive_strength),
            .groups = "drop")
```

Prepararemos nuestros conjuntos de entrenamiento, prueba y agendamos nuestro presupuesto de datos en un proceso de validación cruzada.

```
set.seed(1501)
concrete_split ← initial_split(concrete, strata = compressive_strength)
concrete_train ← training(concrete_split)
concrete_test  ← testing(concrete_split)

set.seed(1502)
concrete_folds ←
  vfold_cv(concrete_train, strata = compressive_strength, repeats = 5)
```

Usaremos algunas instrucciones de pre-procesamiento de datos, pues hay modelos (no todos) que las requieren

```
normalized_rec ←
  recipe(compressive_strength ~ ., data = concrete_train) ▷
  step_normalize(all_predictors())

poly_recipe ←
  normalized_rec ▷
  step_poly(all_predictors()) ▷
  step_interact(~ all_predictors():all_predictors())
```

Preparemos nuestras especificaciones de modelos

```
knn_spec ←
  nearest_neighbor(neighbors = tune(),
                   dist_power = tune(),
                   weight_func = tune()) ▷
  set_engine("kknn") ▷
  set_mode("regression")
```

```
linear_reg_spec ←
  linear_reg(penalty = tune(), mixture = tune()) ▷
  set_engine("glmnet")
```

```
library(baguette)
cart_spec ←
  decision_tree(cost_complexity = tune(), min_n = tune()) ▷
  set_engine("rpart") ▷
  set_mode("regression")

bag_cart_spec ←
  bag_tree() ▷
  set_engine("rpart", times = 50L) ▷
  set_mode("regression")
```

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

xgb_spec ←
```

```
  boost_tree(tree_depth = tune(), learn_rate = tune(),
             loss_reduction = tune(),
             min_n = tune(), sample_size = tune(),
             trees = tune()) ▷
  set_engine("xgboost") ▷
  set_mode("regression")
```

2. SELECCIÓN Y COMPARACIÓN

La idea de poder comparar distintos modelos es poder tener una idea de qué estructura de modelos tienen un mejor desempeño. Para después hacer más optimizaciones en las estructuras que funcionan.

Usaremos la infraestuctura de `tidymodels` para poder utilizar una interfase para ejecutar un flujo de trabajo ordenado.

2.1. Flujo de procesamiento

Empezamos combinando la *receta* estandarizadora con el modelo adecuado.

```
normalized ←
  workflow_set(
    preproc = list(normalized = normalized_rec),
    models = list(KNN = knn_spec)
  )
normalized
```

```
# A workflow set/tibble: 1 × 4
  wflow_id       info             option    result
  <chr>          <list>           <list>    <list>
1 normalized_KNN <tibble [1 × 4]> <opts[0]> <list [0]>
```

Podemos corroborar que tenemos lo usual

```
normalized ▷ extract_workflow(id = "normalized_KNN")
```

```
== Workflow ====================================================================
Preprocessor: Recipe
Model: nearest_neighbor()
-- Preprocessor ----------------------------------------------------------------
1 Recipe Step
- step_normalize()
-- Model -----------------------------------------------------------------------
K-Nearest Neighbor Model Specification (regression)
Main Arguments:
  neighbors = tune()
  weight_func = tune()
  dist_power = tune()
Computational engine: kknn
```

Para los demás modelos podemos utilizar `dplyr` para definir `respuesta` y `atributos`.

```
model_vars ← workflow_variables(
  outcomes = compressive_strength,
  predictors = everything()
)
```

```
no_pre_proc ← workflow_set(
  preproc = list(simple = model_vars),
  models = list(CART = cart_spec,
                CART_bagged = bag_cart_spec,
                RF = rf_spec,
                boosting = xgb_spec)
)
no_pre_proc
```

```
# A workflow set/tibble: 4 × 4
  wflow_id           info             option    result
  <chr>              <list>           <list>    <list>
1 simple_CART        <tibble [1 × 4]> <opts[0]> <list [0]>
2 simple_CART_bagged <tibble [1 × 4]> <opts[0]> <list [0]>
3 simple_RF          <tibble [1 × 4]> <opts[0]> <list [0]>
4 simple_boosting    <tibble [1 × 4]> <opts[0]> <list [0]>
```

Agregamos otro conjunto de modelos que usen términos no lineales e interacciones.

```
with_features ←
  workflow_set(
    preproc = list(fullquad = poly_recipe),
    models = list(linear_reg = linear_reg_spec, KNN = knn_spec)
  )
```

Finalmente, creamos el conjunto completo de procesamiento (preparación, entrenamiento, evaluación)

```
all_workflows ←
  bind_rows(no_pre_proc, normalized, with_features) ▷
  ## Make the workflow ID's a little more simple:
  mutate(wflow_id = gsub("(simple_)|(normalized_)", "", wflow_id))
all_workflows
```

```
# A workflow set/tibble: 7 × 4
  wflow_id            info             option    result
  <chr>               <list>           <list>    <list>
1 CART                <tibble [1 × 4]> <opts[0]> <list [0]>
2 CART_bagged         <tibble [1 × 4]> <opts[0]> <list [0]>
3 RF                  <tibble [1 × 4]> <opts[0]> <list [0]>
4 boosting            <tibble [1 × 4]> <opts[0]> <list [0]>
5 KNN                 <tibble [1 × 4]> <opts[0]> <list [0]>
6 fullquad_linear_reg <tibble [1 × 4]> <opts[0]> <list [0]>
7 fullquad_KNN        <tibble [1 × 4]> <opts[0]> <list [0]>
```

2.2. Ajuste y evaluación de modelos

Casi todos los modelos tienen parámetros que se tienen que ajustar. Podemos utilizar los métodos de ajuste que ya hemos visto (`tune_grid()`, etc.). Con la función `workflow_map()` se aplica la misma función para **todos** los flujos de entrenamiento.

Usaremos las mismas opciones para cada uno. Es decir, 25 candidatos en cada modelo para validación cruzada, utilizando la misma separación en bloques.

La idea de este proceso es ilustrar un mecanismo para condensar en una misma ejecución lo que hemos visto a lo largo de todo el curso. Si, cada modelo tiene distintos hiperparámetros pero de momento nos concentraremos en explorar capacidades predictivas.

```
grid_ctrl ←
  control_grid(
    save_pred = TRUE,
    parallel_over = "everything",
    save_workflow = TRUE
  )
```

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

```
system.time(
  grid_results ← all_workflows ▷
    workflow_map(
      seed = 1503,
      resamples = concrete_folds,
      grid = 25,
      control = grid_ctrl
    )
)
```

```
i Creating pre-processing data to finalize unknown parameter: mtry
    user   system  elapsed
  16.785    2.903 1418.576
```

El `tibble` del flujo se actualiza con las leyendas en `option` y `results`. Los indicadores `tune[+]` y `rsmp[+]` significan que no hubo problemas para procesar ese modelo.

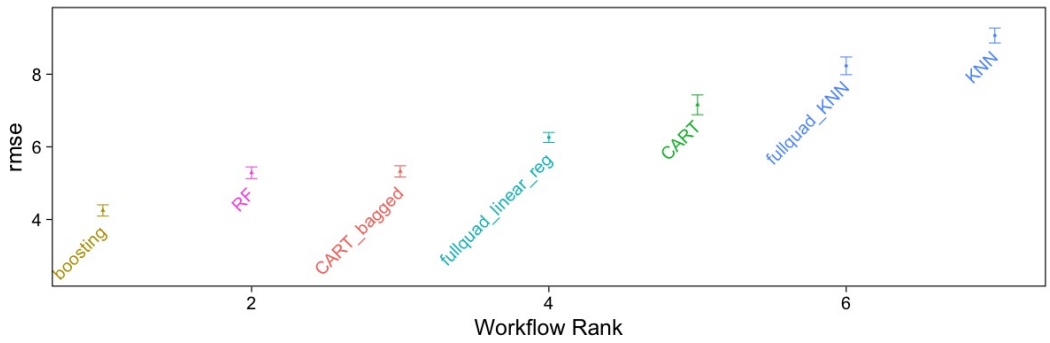
```
grid_results
```

```
# A workflow set/tibble: 7 × 4
  wflow_id             info             option    result
  <chr>                <list>           <list>    <list>
1 CART                 <tibble [1 × 4]> <opts[3]> <tune[+]>
2 CART_bagged          <tibble [1 × 4]> <opts[3]> <rsmp[+]>
3 RF                   <tibble [1 × 4]> <opts[3]> <tune[+]>
4 boosting             <tibble [1 × 4]> <opts[3]> <tune[+]>
5 KNN                  <tibble [1 × 4]> <opts[3]> <tune[+]>
6 fullquad_linear_reg  <tibble [1 × 4]> <opts[3]> <tune[+]>
7 fullquad_KNN         <tibble [1 × 4]> <opts[3]> <tune[+]>
```

Por último, con la función `rank_results()` ordenamos los modelos de acuerdo a su capacidad predictiva. La opción `select_best` nos muestra dentro de cada familia el mejor modelo para ordenar por capacidad predictiva.

```
grid_results ▷
 rank_results(select_best = TRUE) ▷
 filter(.metric == "rmse") ▷
 select(model, .config, rmse = mean, rank)
```

```
# A tibble: 7 × 4
  model             .config                rmse  rank
  <chr>             <chr>                 <dbl> <int>
1 boost_tree        Preprocessor1_Model04  4.25     1
2 rand_forest       Preprocessor1_Model18  5.29     2
3 bag_tree          Preprocessor1_Model1   5.32     3
4 linear_reg        Preprocessor1_Model16  6.26     4
5 decision_tree     Preprocessor1_Model19  7.16     5
6 nearest_neighbor  Preprocessor1_Model18  8.23     6
7 nearest_neighbor  Preprocessor1_Model16  9.07     7
```



2.3. Ajuste y comparación eficiente

Utilizaremos el mismo proceso eficiente de comparación de modelos para determinar la mejor configuración dentro de cada uno.

```
library(finetune)

race_ctrl ←
  control_race(
    save_pred = TRUE,
    parallel_over = "everything",
    save_workflow = TRUE
  )
```

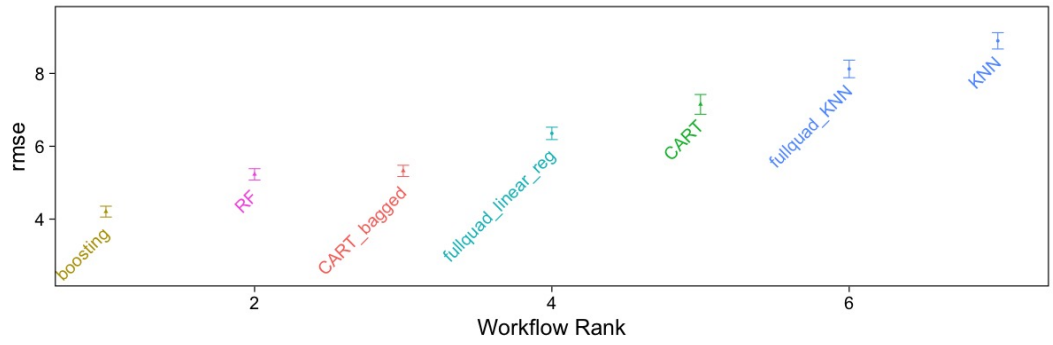
```
system.time(
  race_results ←
    all_workflows ▷
    workflow_map(
      "tune_race_anova",
      seed = 1503,
      resamples = concrete_folds,
      grid = 25,
      control = race_ctrl
    ))
```

```
i Creating pre-processing data to finalize unknown parameter: mtry
   user  system elapsed
 89.465   1.612 364.972
```

El método ajusta 11 modelos de los 151 posibles. Es decir, sólo requiere ajustar el 7.3 % .

```
race_results
```

```
# A workflow set/tibble: 7 × 4
  wflow_id             info             option    result
  <chr>                <list>           <list>    <list>
1 CART                 <tibble [1 × 4]> <opts[3]> <race[+]>
2 CART_bagged          <tibble [1 × 4]> <opts[3]> <rsmp[+]>
3 RF                   <tibble [1 × 4]> <opts[3]> <race[+]>
4 boosting             <tibble [1 × 4]> <opts[3]> <race[+]>
5 KNN                  <tibble [1 × 4]> <opts[3]> <race[+]>
6 fullquad_linear_reg  <tibble [1 × 4]> <opts[3]> <race[+]>
7 fullquad_KNN         <tibble [1 × 4]> <opts[3]> <race[+]>
```



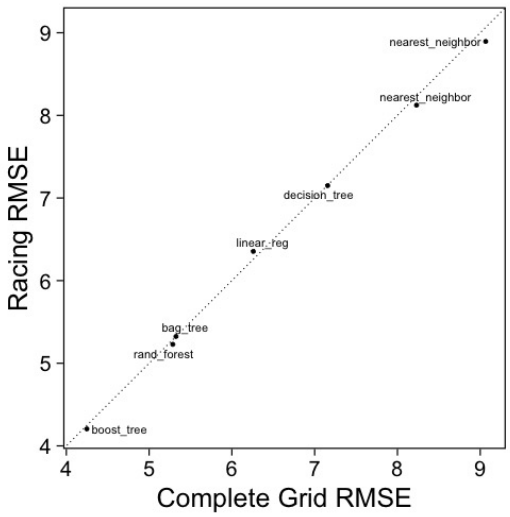


FIGURA 1. *Comparación de procedimiento completo contra paro acelerado.*

2.4. Finalizar modelo

```
best_results ←
  race_results ▷
  extract_workflow_set_result("boosting") ▷
  select_best(metric = "rmse")
best_results
```

```
# A tibble: 1 × 7
  trees min_n tree_depth learn_rate loss_reduction sample_size .config
  <int> <int>      <int>      <dbl>          <dbl>       <dbl> <chr>
1  1957     8          7     0.0756    0.000000145       0.679 Preprocessor1_
    Model04
```

```
boosting_test_results ←
  race_results ▷
  extract_workflow("boosting") ▷
  finalize_workflow(best_results) ▷
  last_fit(split = concrete_split)
```

```
collect_metrics(boosting_test_results)
```

```
# A tibble: 2 × 4
  .metric .estimator .estimate .config
  <chr>   <chr>          <dbl> <chr>
1 rmse    standard        3.52  Preprocessor1_Model1
2 rsq     standard        0.951 Preprocessor1_Model1
```

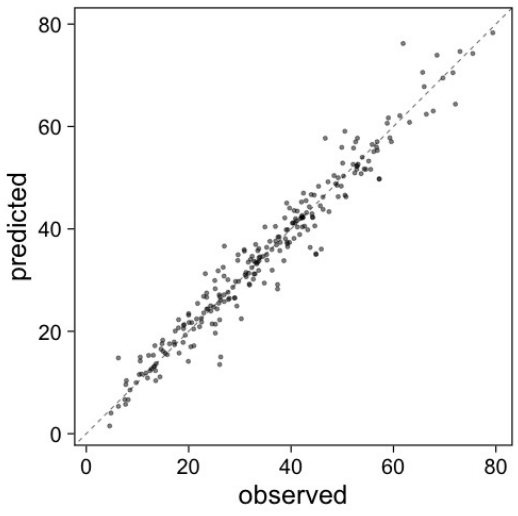


FIGURA 2. *Predicciones contra datos reales.*

3. ENSAMBLE DE MODELOS

Consideremos que tenemos una colección de modelos entrenados con los cuales podemos obtener predicciones $\hat{f}_1(x), \ldots, \hat{f}_M(x)$. Considerando pérdida cuadrática podemos buscar pesos $\omega = (\omega_1, \ldots, \omega_M)$ tales que

$$\hat{\omega} = \arg \min_{\omega \in \mathbb{R}_+^M} \mathbb{E}_{\mathcal{D}_n, Y} \left(Y - \sum_{m=1}^{M} \omega_m \hat{f}_m(x) \right)^2 , \tag{1}$$

donde el valor esperado se calcula considerando la variabilidad en conjuntos de entrenamiento y la aleatoriedad en la respuesta para un atributo fijo.

La solución de este problema será la solución **poblacional** considerando el vector que acopla las predicciones de una familia de modelos $\hat{F}^\top = [\hat{f}_1(x), \ldots, \hat{f}_M(x)]$:

$$\hat{\omega} = \mathbb{E}[\hat{F}(x)\hat{F}^\top(x)]^{-1}\, \mathbb{E}[\hat{F}(x)Y]\,. \tag{2}$$

La combinación lineal, sabemos, tiene mejor capacidad predictiva que cualquier modelo

$$\mathbb{E}\left(Y - \sum_{m=1}^{M} \hat{\omega}_m \hat{f}_m(x) \right)^2 \leq \mathbb{E}\left(Y - \hat{f}_m(x) \right)^2 \quad \forall m\,. \tag{3}$$

El problema es que no tenemos acceso a la solución poblacional del problema de regresión. Asi que podemos usar el concepto de `stacking` para construir un estimador basado en predicciones fuera de muestra.

3.1. Stacking de modelos

Una manera que tenemos para darle la vuelta al problema de antes es utilizar el mecanismo de `validación cruzada` para ajustar los pesos de la siguiente manera.

Denotemos por $\hat{f}_m^{-i}(x)$ la salida del m-ésimo modelo en x el cual fue entrenado sin la observación i-ésima. Los coeficientes se ajustan por medio de mínimos cuadrados utilizando la regresión de respuestas y_i con atributos $\hat{f}_m^{-i}(x)$. De tal forma que tenemos

$$\hat{\omega}^{\mathsf{stack}} = \arg \min_{\omega \in \mathbb{R}^M} \sum_{i=1}^{N} \left(y_i - \sum_{m=1}^{M} \omega_m \hat{f}_m^{-i}(x) \right)^2 . \tag{4}$$

Las predicciones se realizan por medio de

$$\sum_{m=1}^{M} \hat{\omega}_m^{\mathsf{stack}}\, \hat{f}_m^{-i}(x)\,. \tag{5}$$

El resultado es un combinación lineal de predicciones. En la práctica se obtienen los mejores resultados si se restringen los pesos de la combinación lineal a ser no-negativos y que sumen 1.

4. ILUSTRACIÓN NUMÉRICA

Retomaremos nuestra colección de modelos que hemos ajustado. La intención es poder crear una combinación de éstos para mejorar nuestra capacidad predictiva. Para estos usaremos `stacks` de `tidymodels`.

```
1 library(stacks)
```

4.0.1. Para pensar:

1. En el contexto de validación cruzada con K bloques: ¿cuántas predicciones fuera de muestra tenemos para cada observación con el conjunto de entrenamiento?
2. En el contexto de validación cruzada con K bloques: ¿cuántas predicciones fuera de muestra tenemos para cada observación con el conjunto de entrenamiento si repetimos validación cruzada B veces?
3. En el contexto de los modelos que hemos usado: ¿cuántas predicciones fuera de muestra tenemos para *bagging*?

4.1. Construyendo nuestra colección de predicciones

Le pasamos nuestros modelos entrenados a un *stack* vacío. La función `add_candidates()` se encarga de filtrar modelos tienen **todo** el perfil predictivo para cada observación en el conjunto de entrenamiento.

```
concrete_stack ←
  stacks() ▷
  add_candidates(race_results)

concrete_stack
```

```
# A data stack with 7 model definitions and 13 candidate members:
#   CART: 1 model configuration
#   CART_bagged: 1 model configuration
#   RF: 1 model configuration
#   boosting: 1 model configuration
#   KNN: 3 model configurations
#   full_quad_linear_reg: 5 model configurations
#   full_quad_KNN: 1 model configuration
# Outcome: compressive_strength (numeric)
```

4.1.1. Para pensar: ¿Qué pasaría si en lugar de pasar resultados de ANOVA usamos los resultados de la función `tune_grid()`?

4.2. Mezcla de predicciones

Entrenamos nuestro *meta*-modelo utilizando las predicciones fuera de muestra. En esta situación debemos de considerar:

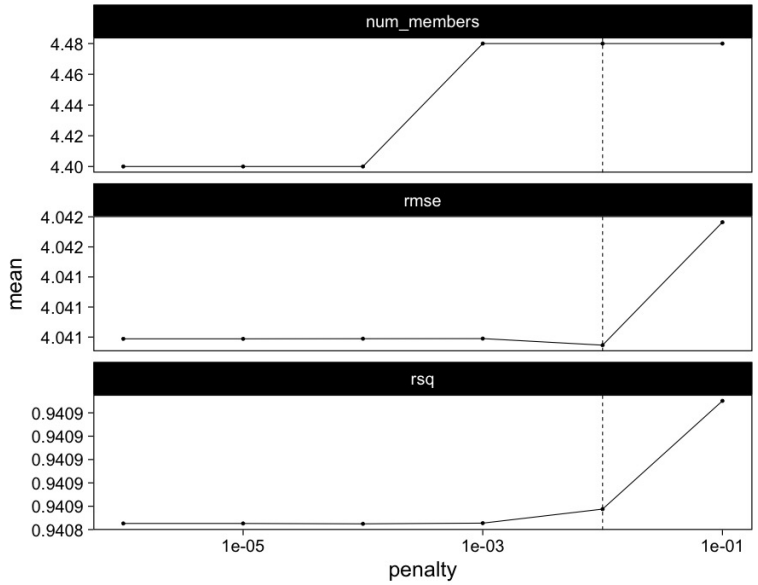
1. Las predicciones entre familias de predictores pueden estar altamente correlacionadas.
2. Habrá predictores que no son necesarios si ya hay algún elemento de la misma familia.

4.2.1. Para pensar: ¿Qué estrategia de regularización hemos visto para resolver estos problemas?

4.2.2. Implementación La función `stacks::blend_predictions()` nos permite ajustar un modelo lineal regularizado que es capaz de evitar estos problemas.

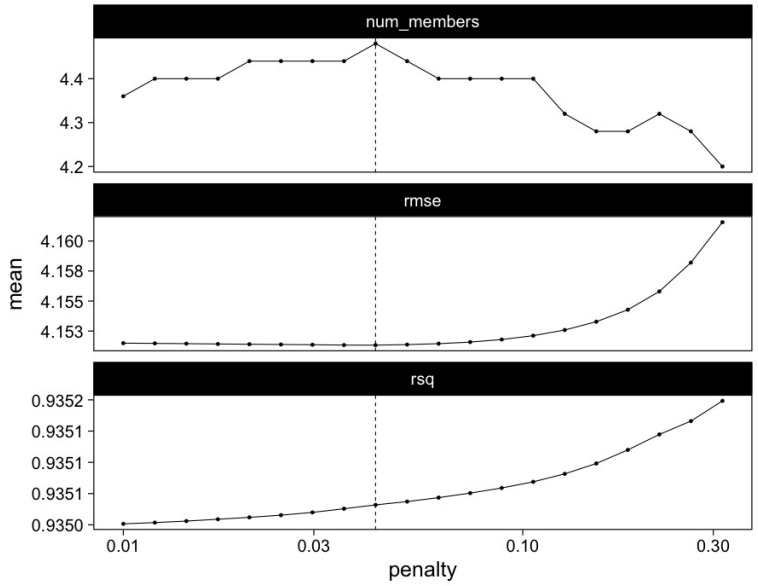
```
set.seed(2001)
ens ← blend_predictions(concrete_stack)
```

El procedimiento realiza un ajuste interno con remuestreo que permite construir trazar la curva de error predictivo como función de factor de penalización.



Al igual que en validación cruzada los resultados obtenidos nos pueden ayudar a concentrar nuestros esfuerzos computacionales en zonas de mayor interés.

```
set.seed(2002)
ens ← blend_predictions(concrete_stack, penalty = 10^seq(-2, -0.5, length =
    20))
```



La combinación final queda construida de la siguiente manera. x

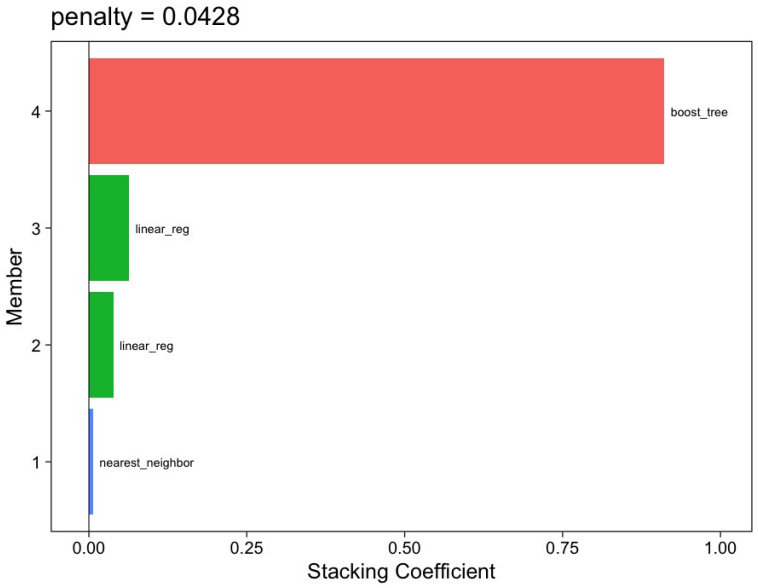
```
ens
```

```
-- A stacked ensemble model -------------------------------------------------
Out of 13 possible candidate members, the ensemble retained 4.
Penalty: 0.0428133239871939.
Mixture: 1.
```

```
The 4 highest weighted members are:
# A tibble: 4 × 3
member                    type              weight
<chr>                     <chr>              <dbl>
1 boosting_1_04             boost_tree        0.911
2 fullquad_linear_reg_1_17 linear_reg        0.0638
3 fullquad_linear_reg_1_16 linear_reg        0.0387
4 KNN_1_12                  nearest_neighbor 0.00704

Members have not yet been fitted with ‘fit_members()‘.
```

Las contribuciones de cada modelo se pueden resumir de manera gráfica con la función de `autoplot()` †.



4.2.3. Para pensar: Por último hay que ajustar los modelos finales a todo el conjunto de entrenamiento. ¿En nuestro ejemplo cuántos modelos se entrenan? ¿Cómo cambia esto cuando lo comparamos con el contexto de validación cruzada?

```
ens ← fit_members(ens)
```

```
reg_metrics ← metric_set(rmse, rsq)
ens_test_pred ←
  predict(ens, concrete_test) ▷
  bind_cols(concrete_test)

ens_test_pred ▷
  reg_metrics(compressive_strength, .pred)
```

```
# A tibble: 2 × 3
  .metric .estimator .estimate
  <chr>   <chr>          <dbl>
1 rmse    standard       3.36
2 rsq     standard       0.956
```

5. CONCLUSIONES

- Las mejoras pueden ser marginales comparadas contra las del mejor modelo individual.
- En la práctica la mezcla de modelos vuelve el resultado muy poco interpretable.
- Para las tareas de interés predictivo (sin explicaciones y sin restricciones computacionales) son lo mejor.
- Una combinación lineal de predictores es sólo una forma de combinar.

REFERENCIAS

[1] 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

[2] M. Kuhn and K. Johnson. *Applied Predictive Modeling*. Springer New York, New York, NY, 2013. ISBN 978-1-4614-6848-6 978-1-4614-6849-3. 1

[3] M. Kuhn and J. Silge. *Tidy Modeling with R*. O'Reilly Media, Inc., 2022. 1