Aprendizaje Estadístico Supervisado

Bruno Tancredi

2024

Laboratorio remuestreo

- Este laboratorio vamos a ver como aplicar distintos métodos de remuestreo.
- Usaremos varios paquetes, rsample para crear conjuntos de datos basados en remuestreo y yardstick para calcular métricas de performance.
- Usaremos rsample y dials para la selección de los valores de los parámetros de tuneo.

Laboratorio remuestreo

Vamos a seguir un ejemplo del blog de Julia Silge **link blog**

¿Conocen a Julia?

> **Importante**
>
> Julia Silge va a participar en esta edición de LatinR!

Paquetes que usaremos

```
library(tidymodels) #Recordar que al importar tidymodels estamos im
library(schrute) #Contiene los datos que vamos a utilizar, es necesari
library(vip) #Importancia de variables
set.seed(1234)
```

Datos

Qué son los datos

- Utilizaremos el conjunto de datos theoffice del paquete schrute. Cada fila del conjunto de datos es una linea de diálogo.
- Queremos predecir la calificación del episodio obtenida en la plataforma **IMDB** en base a predictores que están presentes en el conjunto de datos y que iremos construyendo.

Datos de calificación

```
office_info <- theoffice |>
    select(season, episode_name, director, writer,
character, text, imdb_rating)
office_info |> head()
```

```
# A tibble: 6 × 7
  season episode_name director   writer          character text  imdb_rating
   <int> <chr>        <chr>      <chr>           <chr>     <chr>       <dbl>
1      1 Pilot        Ken Kwapis Ricky Gervais;Step… Michael   All …       7.6
2      1 Pilot        Ken Kwapis Ricky Gervais;Step… Jim       Oh, …       7.6
3      1 Pilot        Ken Kwapis Ricky Gervais;Step… Michael   So y…       7.6
4      1 Pilot        Ken Kwapis Ricky Gervais;Step… Jim       Actu…       7.6
5      1 Pilot        Ken Kwapis Ricky Gervais;Step… Michael   All …       7.6
6      1 Pilot        Ken Kwapis Ricky Gervais;Step… Michael   Yes,…       7.6
```

Creación de atributos

- Cantidad de lineas que tiene cada personaje en el episodio.
- Guionistas/Directores (no diferenciaremos el rol) que participan en el episodio.

Lineas en el capítulo por personaje.

```
characters <- office_info |>
  count(episode_name, character) |> # Cantidad de lineas por cap
  add_count(character, wt = n, name = "character_count") |> #Ca
  filter(character_count > 800) |> #Obtenemos los personajes que
  select(-character_count) |>
  pivot_wider(
    names_from = character,
    values_from = n,
    values_fill = list(n = 0)
  ) #Obtenemos una fila sola por capitulo, ponemos 0 si el person
```

Directores/Esritores del capítulo.

```
creators <- office_info |>
  distinct(episode_name, director, writer) |>
  pivot_longer(director:writer, names_to = "role", values_to = "per
  separate_rows(person, sep = ";") |>
  add_count(person) |>
  mutate(person = case_when(
    n <= 10 ~ 'Guest',
    n > 10 ~ person
  )) |>
  distinct(episode_name, person) |>
  mutate(person_value = 1) |>
  pivot_wider(
    names_from = person,
    values_from = person_value,
    values_fill = list(person_value = 0)
  )
```

Conjunto final

```
office <- office_info |>
  distinct(season, episode_name, imdb_rating) |>
  inner_join(characters) |>
  inner_join(creators) |>
  mutate_at("season", as.factor)
```

Exploración

```
1  office |>
2    ggplot(aes(season, imdb_rating, fill = as.factor(season))) +
3    geom_boxplot(show.legend = FALSE)
```

División de datos

- Dividimos los datos en train/test. Tratando de conservar la proporción de datos por temporada.

```
office_split <- initial_split(office,
                              strata = season,
                              prop = 3/4)
office_train <- training(office_split)
office_test <- testing(office_split)
```

Modelo

recipe

Permite especificar una serie de pasos de preprocesamiento de datos, facilitando la creación de flujos de trabajo reproducibles.

```
office_rec <- recipe(imdb_rating ~ ., data = office_train) |>
  update_role(episode_name, new_role = "ID") |> #Nos permite c
  step_dummy(season) |> #codificamos categorias a columnas bi
  step_normalize(all_numeric(), -all_outcomes()) #Normalizamos
```

Más info de step_functions **link**

parsnip

Proporciona una interface ordenada y unificada de modelos que puede ser usada para aplicar una amplia variedad de modelos sin preocuparnos por las variaciones en la sintaxis de los distintos paquetes.

- mixture=1 para Lasso, mixture=0 para Ridge.
- La implementación de glmnet en parsnip tiene ciertas peculiaridades, consultar **documentación**
- Con show_engines("linear_reg") vemos todas los posibles paquetes para implementar el tipo de modelo de parsnip linear_reg

```
1  tune_spec <- linear_reg(penalty = tune(), mixture = 1) |> #Con tu
2    set_engine("glmnet")
```

workflow

Proporciona una manera de combinar una especificación de modelo, una receta de preparación de datos y un conjunto de parámetros de ajuste del modelo en un solo objeto. Esto facilita la gestión de todo el proceso de modelado.

- Cuando usamos workflow la llamada a fit() realiza el preprocesamiento de los datos y el ajuste del modelo.

```
tune_wf <- workflow() |>
  add_recipe(office_rec) |>
  add_model(tune_spec)
```

Ajuste de hiperparámetros

- Si queremos encontrar el mejor valor para \(\lambda\) implica el tuneo de este hiperparámetro y podemos usar k-Fold Cross-Validation.

  Para poder hacer validación cruzada necesitamos usar el paquete tune y tres cosas para que funcione:

- Un objeto parsnip / workflow con uno o más argumentos indicados para ser tuneados.
- Un objeto vfold_cv rsample con las particiones de validación cruzada.
- Una tibble que denote los valores de los hiperparámetros a ser explorados.

Ajuste de hiperparámetros

```
office_cv <- vfold_cv(office_train, v = 5) #5 particiones debido a c
lambda_grid <- grid_regular(penalty(c(-10,-1)),  levels = 50) #Def



lasso_tune<- tune_grid( #Realizamos el ajuste, pensar que estar
  tune_wf,
  resamples = office_cv,
  grid = lambda_grid
)
```

Ajuste de hiperparámetros

```
lasso_tune |>
  collect_metrics()
```

```
# A tibble: 100 × 7
    penalty .metric .estimator  mean     n std_err .config
      <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
 1 1    e-10 rmse    standard   0.482     5  0.0315 Preprocessor1_Model01
 2 1    e-10 rsq     standard   0.341     5  0.0400 Preprocessor1_Model01
 3 1.53e-10 rmse    standard   0.482     5  0.0315 Preprocessor1_Model02
 4 1.53e-10 rsq     standard   0.341     5  0.0400 Preprocessor1_Model02
 5 2.33e-10 rmse    standard   0.482     5  0.0315 Preprocessor1_Model03
 6 2.33e-10 rsq     standard   0.341     5  0.0400 Preprocessor1_Model03
 7 3.56e-10 rmse    standard   0.482     5  0.0315 Preprocessor1_Model04
 8 3.56e-10 rsq     standard   0.341     5  0.0400 Preprocessor1_Model04
 9 5.43e-10 rmse    standard   0.482     5  0.0315 Preprocessor1_Model05
10 5.43e-10 rsq     standard   0.341     5  0.0400 Preprocessor1_Model05
# ℹ 90 more rows
```

Ajuste de hiperparámetros

```
lasso_tune |>
  collect_metrics() |>
  ggplot(aes(penalty, mean, color = .metric)) +
  geom_errorbar(aes(
    ymin = mean - std_err,
    ymax = mean + std_err
  ),
  alpha = 0.5
  ) +
  geom_line(size = 1.5) +
  facet_wrap(~.metric, scales = "free", nrow = 2) +
  scale_x_log10() +
  theme(legend.position = "none")
```

Ajuste de hiperparámetros

Ajuste de hiperparámetros

Seleccionamos el modelo cuyo RMSE es más bajo. Podríamos haber utilizado otra métrica (**métricas disponibles**). Existen otros criterios, para la selección como select_by_one_std_err() que selecciona el modelo más simple cuyo error este dentro de un desvío estándar del modelo de mejor resultado.

```
lowest_rmse <- lasso_tune |>
  select_best(metric = "rmse")

final_lasso <- finalize_workflow(tune_wf, lowest_rmse)
```

Resultados de test.

```
last_fit(
  final_lasso,
  office_split
) |>
  collect_metrics()
```

```
# A tibble: 2 × 4
  .metric .estimator .estimate .config
  <chr>   <chr>          <dbl> <chr>
1 rmse    standard       0.393 Preprocessor1_Model1
2 rsq     standard       0.376 Preprocessor1_Model1
```

Podríamos haber hecho lo mismo con bootstrap!

```
office_boot <- bootstraps(office_train, strata = season)
lambda_grid <- grid_regular(penalty(c(-10,-1)), levels = 50) #Defi

lasso_tune <- tune_grid(
  tune_wf,
  resamples = office_boot,
  grid = lambda_grid
)
```

Resumen del flujo de trabajo.

Diagnóstico de modelo

Importancia de variable

Queremos ver que variables son más importantes a la hora de predecir la calificación. vi toma al valor absoluto del coeficiente asociado a la variable (recordar que las variables están normalizadas) como indicador de la importancia.

- Para calcular la importancia de variable fue necesario extraer el modelo y trabajar con el objeto glmnet.

```
final_lasso |>
  fit(office_train) |>
  extract_fit_engine() |>
  vi(lambda = lowest_rmse$penalty) |> #Es muy importante marca
  ggplot(aes(x = Importance, y = reorder(Variable, Importance), fil
  geom_col() +
  scale_x_continuous(expand = c(0, 0)) +
  labs(y = NULL)
```

Importancia de variable

Tu turno!

Haremos el ejercicio 9 del capítulo 6 del libro ISL. Predicaremos el número de aplicaciones recibidas usando las otras variables del conjunto de datos College como predictores.

1. Cargar el conjunto de datos College, del paquete ISLR2. Dividir estos datos en entrenamiento/testeo.
2. Ajustar un modelo lineal utilizando mínimos cuadrados. Reportar el error obtenido en el conjunto de testeo.
3. Ajustar un modelo lineal utilizando Ridge. Escoger \(\lambda\) con validación cruzada. Reportar el error obtenido en el conjunto de testeo.

Tu turno!

4. Ajustar un modelo lineal utilizando Lasso. Escoger \(\lambda\) con validación cruzada. Reportar el error obtenido en el conjunto de testeo.
5. Ajustar un modelo de vecinos más cercanos. Escoger \(k\) con validación cruzada. Reportar el error obtenido en el conjunto de testeo.
6. Comentar los resultados obtenidos. ¿Con cuanta precisión podemos predecir el numero de aplicaciones?. ¿Hay un mejor modelo? ¿Hay mucha diferencia en el error de testeo de los modelos?