TidyModels

Carlos Morales Aguilera

29/4/2021

Contents

1 Introducción

Dentro del ámbito de Machine Learning, Data Mining o estadística, **R** es uno de los principales lenguajes de programación, al tratarse de un software orientado a este tipo de labores, y con una gran variedad de bibliotecas que facilitan las funcionalidades deseadas. Por otro lado, la utilización de diferentes librerías requiere una gran curva de aprendizaje ya que cada una funciona de manera diferente.

Existen librerías como **caret** o **tidyverse** que pretenden solventar este problema proporcionando una interfaz que bajo un único marco se unifiquen los procedimientos de diferentes librerías. En este caso hablamos dentro del contexto de **tidyverse**, donde encontramos **Tidymodels**. Esta librería es una interfaz que reúne bajo un marco único funciones de diferentes paquetes que facilitan las diferentes etapas de Preprocesamiento, Entrenamiento de modelos, Optimización y Validación de modelos predictivos.

Por otro lado, se puede encontrar una separación de esta librería en diferentes paquetes como son:

** **rsample** - Operaciones asociadas a muestreos de los datos. ** **recipes** - Operación y organización de las diferentes técnicas de preprocesamiento. ** **parnsip** - Modelado y entrenamiento de diferentes modelos de distintos paquetes. ** **tune** - Operaciones de *tuning* de modelos predictivos. ** **yardstick** - Operaciones de métricas de modelos. ** **workflows** - Combinación de procesos en un único flujo de trabajo.

```
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --

## v ggplot2 3.3.3     v purrr   0.3.4
## v tibble  3.1.1     v dplyr   1.0.5
## v tidyr   1.1.3     v stringr 1.4.0
## v readr   1.4.0     v forcats 0.5.1

## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
library(tidymodels)
```

```
## -- Attaching packages --------------------------------------- tidymodels 0.1.3 --

## v broom        0.7.6      v rsample      0.0.9
## v dials        0.0.9      v tune         0.1.5
## v infer        0.5.4      v workflows    0.2.2
## v modeldata    0.1.0      v workflowsets 0.0.2
## v parsnip      0.1.5      v yardstick    0.0.8
## v recipes      0.1.16

## -- Conflicts ------------------------------------------ tidymodels_conflicts() --
## x scales::discard() masks purrr::discard()
## x dplyr::filter()   masks stats::filter()
## x recipes::fixed()  masks stringr::fixed()
## x dplyr::lag()      masks stats::lag()
## x yardstick::spec() masks readr::spec()
## x recipes::step()   masks stats::step()
## * Use tidymodels_prefer() to resolve common conflicts.
```

```
library(titanic)
library(skimr)
library(DataExplorer)
```

2 Datos

El conjunto de datos del **Titanic** describe el estado de supervivencia de pasajeros individuales en el Titanic. No contiene información de la tripulación, pero contiene edades reales de la mitad de los pasajeros. La principal fuente de datos sobre Pasajeros del Titanic es la denominada Enciclopedia Titanica. Los conjuntos de datos utilizados aquí fueron iniciados por una variedad de investigadores.

Descripción de variables:

** **Pclass** Passenger Class (1 = 1st; 2 = 2nd; 3 = 3rd) ** **survived** Survived (0 = No; 1 = Yes) ** **name** Name ** **sex** Sex ** **age** Age ** **sibsp** Number of Siblings/Spouses Aboard ** **parch** Number of Parents/Children Aboard ** **ticket** Ticket Number ** **fare** Passenger Fare (British pound) ** **cabin** Cabin ** **embarked** Port of Embarkation (C = Cherbourg; Q = Queenstown; S = Southampton)

2.1 Lectura de datos

Se utiliza la librería `titanic`:

```
train <- read_csv("https://raw.githubusercontent.com/agconti/kaggle-titanic/master/data/train.csv")
```

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   PassengerId = col_double(),
##   Survived = col_double(),
##   Pclass = col_double(),
##   Name = col_character(),
##   Sex = col_character(),
##   Age = col_double(),
##   SibSp = col_double(),
##   Parch = col_double(),
##   Ticket = col_character(),
##   Fare = col_double(),
##   Cabin = col_character(),
##   Embarked = col_character()
## )
```

```
train$Survived <- as.factor(train$Survived)
```

3 Análisis exploratorio

Antes de realizar las labores para las que está diseñado **Tidymodels**, se procede a realizar un sencillo análisis exploratorio para poder observar la distribución y relaciones de los datos.

3.1 Resumen inicial

```
skim(train)
```

Table 1: Data summary

Name	train
Number of rows	891
Number of columns	12
Column type frequency:	
character	5
factor	1
numeric	6
Group variables	None

Variable type: character

skim_variable	n_missing	complete_rate	min	max	empty	n_unique	whitespace
Name	0	1.00	12	82	0	891	0
Sex	0	1.00	4	6	0	2	0
Ticket	0	1.00	3	18	0	681	0
Cabin	687	0.23	1	15	0	147	0
Embarked	2	1.00	1	1	0	3	0

Variable type: factor

skim_variable	n_missing	complete_rate	ordered	n_unique	top_counts
Survived	0	1	FALSE	2	0: 549, 1: 342

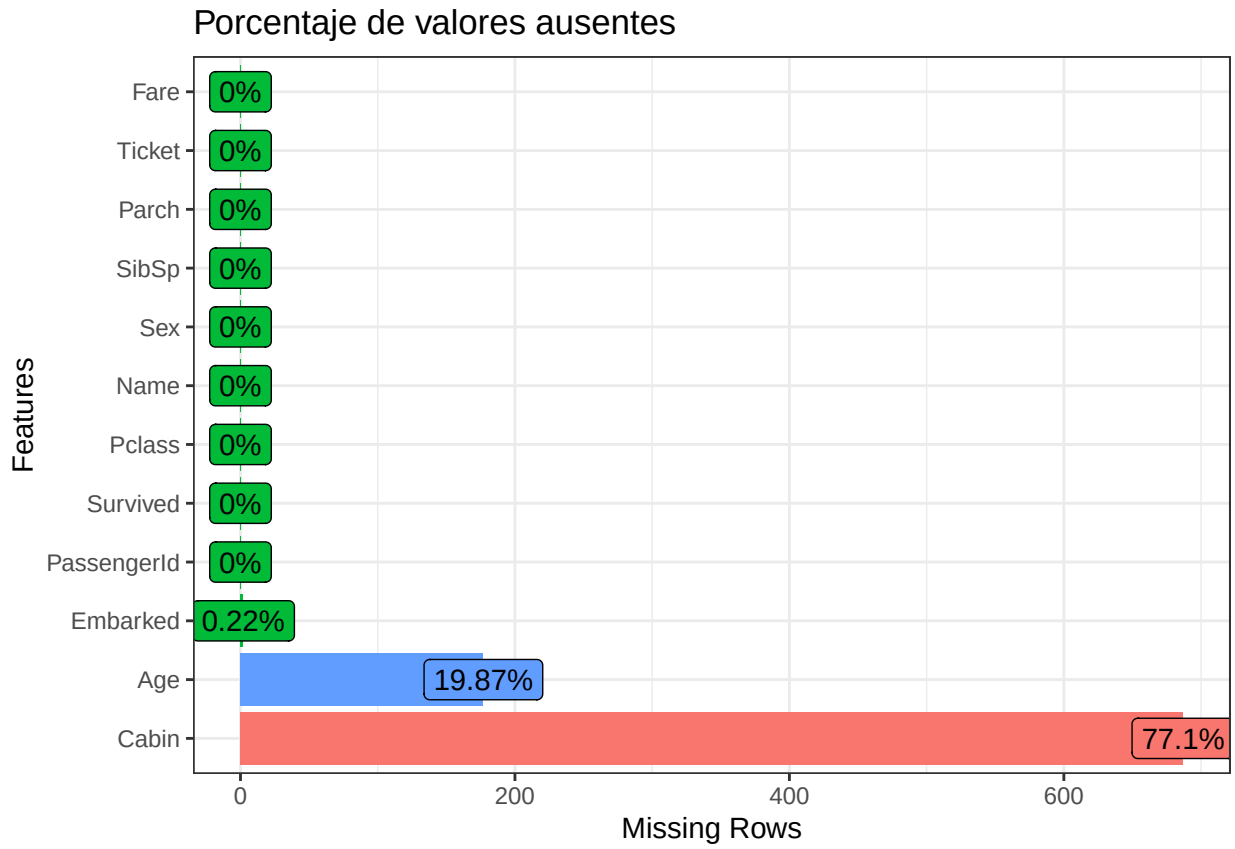
Variable type: numeric

skim_variable	n_missing	complete_rate	mean	sd	p0	p25	p50	p75	p100	hist
PassengerId	0	1.0	446.00	257.35	1.00	223.50	446.00	668.5	891.00	
Pclass	0	1.0	2.31	0.84	1.00	2.00	3.00	3.0	3.00	
Age	177	0.8	29.70	14.53	0.42	20.12	28.00	38.0	80.00	
SibSp	0	1.0	0.52	1.10	0.00	0.00	0.00	1.0	8.00	
Parch	0	1.0	0.38	0.81	0.00	0.00	0.00	0.0	6.00	
Fare	0	1.0	32.20	49.69	0.00	7.91	14.45	31.0	512.33	

3.2 Valores perdidos

```
plot_missing(
  data    = train,
```

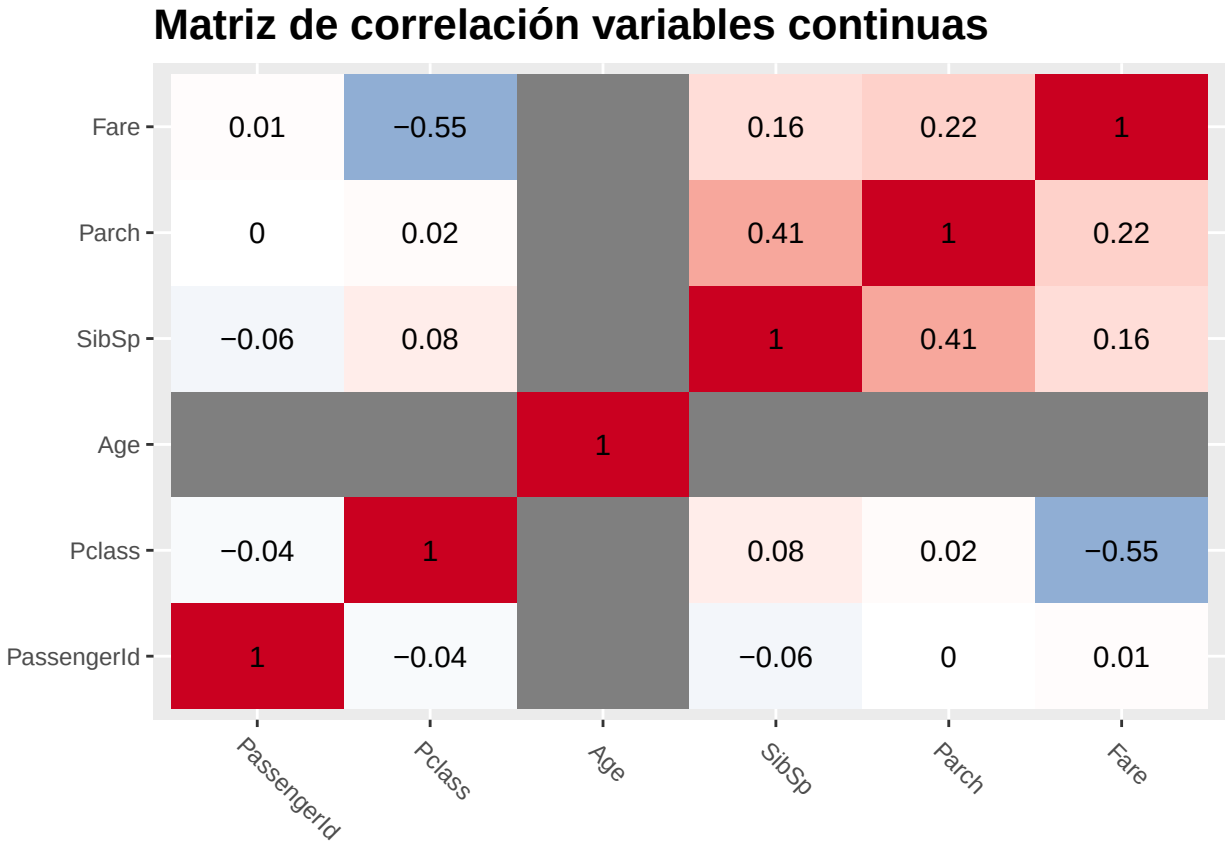
```
  title    = "Porcentaje de valores ausentes",
  ggtheme = theme_bw(),
  theme_config = list(legend.position = "none")
)
```



3.3 Correlación de variables

```
plot_correlation(
  data = train,
  type = "continuous",
  title = "Matriz de correlación variables continuas",
  theme_config = list(legend.position = "none",
                      plot.title = element_text(size = 16, face = "bold"),
                      axis.title = element_blank(),
                      axis.text.x = element_text(angle = -45, hjust = +0.1)
                     )
)
```

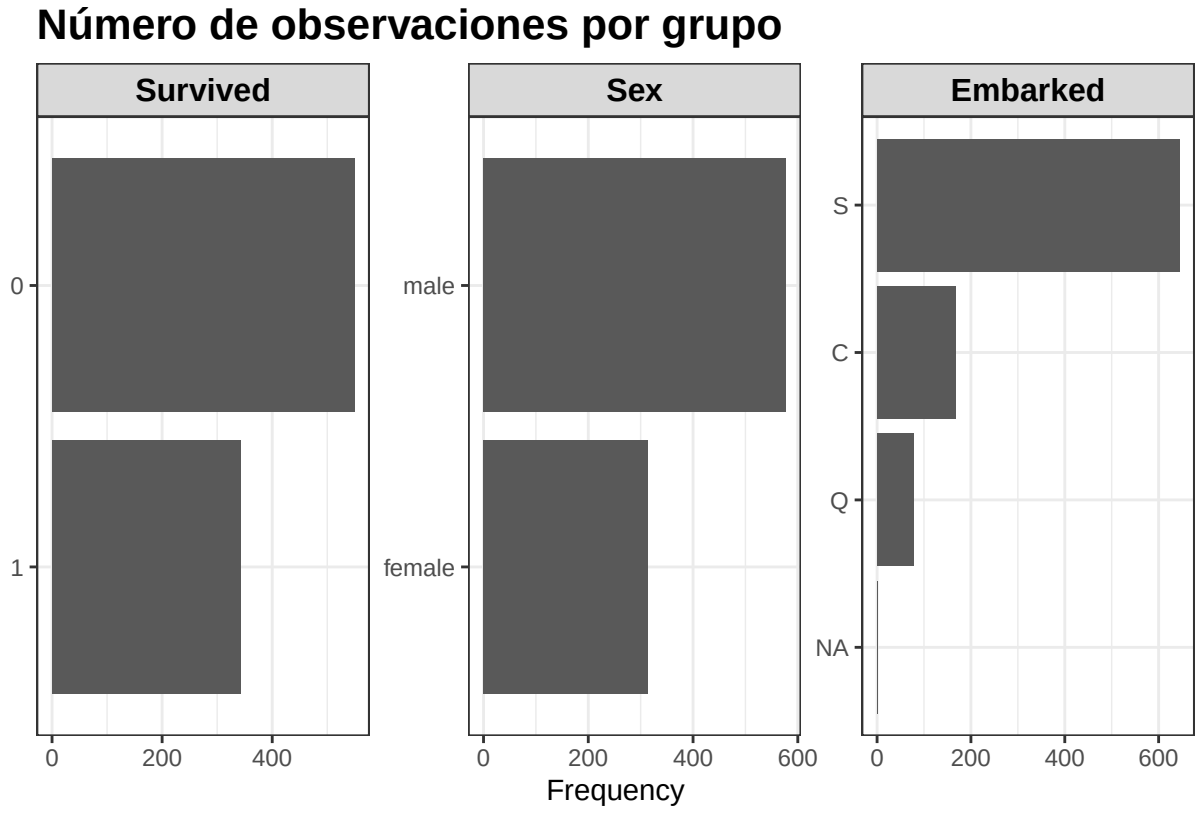
```
## Warning: Removed 10 rows containing missing values (geom_text).
```



3.4 Balanceo de variables

```
plot_bar(
  train,
  ncol    = 3,
  title   = "Número de observaciones por grupo",
  ggtheme = theme_bw(),
  theme_config = list(
                   plot.title = element_text(size = 16, face = "bold"),
                   strip.text = element_text(colour = "black", size = 12, face = 2),
                   legend.position = "none"
                  )
)
```

```
## 3 columns ignored with more than 50 categories.
## Name: 891 categories
## Ticket: 681 categories
## Cabin: 148 categories
```



4 Particionamiento de datos

Como en cualquier problema que se aborda, es necesario tener un conjunto de datos de muestra sobre el que aprender o de **entrenamiento** y otro de comprobación o **test**. Para realizar esta funcionalidad se suelen utilizar diferentes opciones de particionamiento. **Tidymodels** ofrece su propio paquete **rsample**, el cual se encarga de hacer esta partición ajustándose a los datos dada una proporción.

Para esta función se utiliza la función **initial_split()** a la que entre otras partes se puede asignar tanto una proporción como la variable que se utiliza para estratificar los submuestreos. En nuestro caso `Survived`. Vamos a realizar una proporción del 80%-20% para entrenamiento-test.

```
split_data <- initial_split(
                    data   = train,
                    prop   = 0.8,
                    strata = Survived
                  )
data_train <- training(split_data)
data_test  <- testing(split_data)
```

5 Preprocesado

Todas las transformaciones sobre los datos con el objetivo de ser útiles, interpretables y valiosos para el estudio pertenecen a esta fase de preprocesamiento. Para ello existen diferentes operaciones que se pueden

realizar sobre los datos con este fin, y **Tidymodels** aporta una serie de herramientas mediante al paquete **recipes**.

Las recetas se construyen como una seria de pasos de preprocesamiento como pueden ser:

** Convertir predictores cualitativos en variables *dummy*. ** Transformar datos para que estén en una escala diferente. ** Transformar grupos enteros de predictores juntos. ** Extraer características clave de variables sin procesar.

5.1 Omisión de valores perdidos

Una de las técnicas más comunes, consistente en la eliminación de valores perdidos, básicamente obtiene todos los predictores y elimina aquella información donde existe un valor ausente para no introducir información completa y obtener un modelo más fiable. **recipes** define para ello la función **step_naomit()**, la cual funciona de forma sencilla indicando que predictores evaluar.

5.2 Imputación de valores perdidos

A diferencia de otras técnicas ya conocidas consistentes en la eliminación de valores perdidos o de variables que los contengan, una de las principales ventajas de **Tidymodels** en este ámbito es el de poder imputar valores utilizando el paquete **recipes** mediante diferentes métodos de imputación, que permiten coger los datos ya existentes y predecir el valor de las variables ausentes. Aunque conlleva un riesgo, es un enfoque claramente diferentes a los ya existentes y supone una nueva herramienta a considerar.

Algunos de estos métodos son **step_bagimpute()**, **step_impute_knn()**, **step_meaninpute()**, **step_medianimpute()** y otros más hasta un total de 7 diferentes métodos.

5.3 Exclusión variables con varianza cercana a cero

Como es lógico, los predictores con valor único no se incluyen, pero existen casos donde existen varios valores pero la varianza es prácticamente nula, por lo que no aportan demasiada información, para ello se define en **recipes** la función **step_nzv()**, donde se definen tanto el ratio de frecuencias como el porcentaje de valores únicos.

5.4 Normalización de variables

Como se ha visto en trabajos previos, otra de las técnicas más relevantes es la normalización de datos en diferentes rangos para su procesamiento. Dentro de **recipes** se distinguen principalmente dos estrategias:

** **Centrado**: Restar la media a cada uno de los valores de los predictores. Valores centrados en torno al origen. ** **Normalización**: Escalar o estandarizar los datos. Como ya conocemos existen dos técnicas principales como son la **Normalización Z-score** y la **Normalización max-min**.

5.5 Binarización de variables

Este procedimiento más conocido como variables **dummy** es muy común y consiste en distinguir todos los casos de una variable en valores binarias de valor 0 o 1 recogiendo de formás más sencilla esta información. **recipes** aporta la función **step_dummy()** la cual binariza las variables y elimina niveles redundantes (novedad frente a otras librerías similares en esta operación).

5.6 Funcionamiento recipes

¿Qué es entonces **recipes**? Una receta no es más que un modelo sobre un conjunto de datos, al cual se le indica la variable objetivo y se le aplican diferentes transformaciones como las indicadas previamente. Por lo tanto a continuación se aplican algunas de dichas transformaciones:

```
receta <- recipe(
  formula = Survived ~.,
  data = data_train
) %>%
  # Omisión de valores perdidos
  step_naomit(all_predictors()) %>%
  # Exclusión variables varianza cercana a cero
  step_nzv(all_predictors()) %>%
  # Normalización max min
  step_range(all_numeric(), -all_outcomes(), min = 0, max = 1)
```

A continuación, podemos observar que se ha definido un modelo:

```
receta
```

```
## Data Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor         11
##
## Operations:
##
## Removing rows with NA values in all_predictors()
## Sparse, unbalanced variable filter on all_predictors()
## Range scaling to [0,1] for all_numeric(), -all_outcomes()
```

Para poder aplicar dicho modelo, a continuación tenemos que ajsutar dicha receta como *preparada* con la función **prep()** la cual entrena el modelo, y posteriormente pasaríamos a *cocinarla* con **juice()** o **bake()** sobre ambos conjuntos. En este caso se utiliza **bake()** ya que con **prep()** se han aprendido las transformaciones.

```
receta_fit <- prep(receta)

# Aplicar transformaciones
data_train_prep <- bake(receta_fit, new_data = data_train)
data_test_prep <- bake(receta_fit, new_data = data_test)
```

Se puede observar mediante un pequeño análisis que han cambiado las variables:

```
skim(data_train_prep)
```

Table 5: Data summary

Name	data_train_prep
Number of rows	159
Number of columns	12
Column type frequency:	
factor	6
numeric	6
Group variables	None

Variable type: factor

skim_variable	n_missing	complete_rate	ordered	n_unique	top_counts
Name	0	1	FALSE	159	All: 1, All: 1, All: 1, All: 1
Sex	0	1	FALSE	2	mal: 84, fem: 75
Ticket	0	1	FALSE	120	113: 4, 199: 4, 110: 3, 113: 3
Cabin	0	1	FALSE	122	B96: 4, C23: 4, C22: 3, D: 3
Embarked	0	1	FALSE	3	S: 102, C: 56, Q: 1
Survived	0	1	FALSE	2	1: 106, 0: 53

Variable type: numeric

skim_variable	n_missing	complete_rate	mean	sd	p0	p25	p50	p75	p100	hist
PassengerId	0	1	0.52	0.28	0	0.30	0.52	0.78	1	
Pclass	0	1	0.09	0.25	0	0.00	0.00	0.00	1	
Age	0	1	0.50	0.22	0	0.34	0.50	0.67	1	
SibSp	0	1	0.16	0.21	0	0.00	0.00	0.33	1	
Parch	0	1	0.12	0.19	0	0.00	0.00	0.25	1	
Fare	0	1	0.15	0.14	0	0.06	0.11	0.18	1	

6 Modelos de aprendizaje

Como en todo trabajo de Machine Learning, el siguiente paso es el modelado de diferentes herramientas de aprendizaje. La elección de un modelo es una tarea compleja y por eso no se abordará con mayor detenimiento en este trabajo, pero caben destacar una serie de fases a seguir:

1. **Ajuste/Entrenamiento del modelo**: Se aplica un algoritmo de Machine Learning sobre el conjunto de datos de entrenamiento.
2. **Evaluación/validación del modelo**: Mediante diferentes técnicas se pretende observar como de bueno es el modelo sobre muestras del conjunto inicial de entrenamiento.
3. **Optimización de hiperparámetros**: Algunos algoritmos poseen diferentes parámetros que se configurar con el objetivo de refinar el modelo a realizar.
4. **Predicción**: Se utiliza el modelo para predicir datos sobre el conjunto de test.

Aquí es donde se puede percibir la potencia de **Tidymodels** ya que facilita toda esta labor en unas simples operaciones que llevan todo el trabajo por debajo, pero que a su vez permiten ajustar los diferentes modelos

con gran detalle si se desea de una forma muy intuitiva. Para ello posee diferentes paquetes que veremos a continuación.

6.1 Entrenamiento

Para los diferentes modelos dentro de **Tidymodels** existe un paquete único denominado **parsnip**, el cual se abstrae al igual que librerías como **caret** de la customización de cada una de las librerías existentes y facilita una interfaz única de entrenamiento del modelo de forma que se definen tres componentes principales: **modelo**, **engine** (implementación) y **ajuste**.

6.1.1 Modelo

A continuación, con el objetivo de observar un modelo sencillo, se va a utilizar un **árbol de regresión**, el cual permitirá predecir en base a los predictores si un pasajero ha sobrevivido o no al hundimiento del Titanic. Realmente es sencillo y solo cabe señalar que se indica el algoritmo **rpart**:

```
modelo_rpart <- decision_tree(mode = "classification") %>%
                set_engine(engine = "rpart")
```

A continuación se procedería a realizar el ajuste del modelo, para ello **parnsip** ofrece dos funciones de ajuste siendo **fit()** y **fit_xy()** (esta última en lugar de la clásica fórmula define matrices de predictores y vector de variable respuesta). Utilizaremos por simpleza **fit()**.

```
modelo_rpart_fit <- modelo_rpart %>% fit(formula = Survived ~.,
                                         data = data_train_prep)
```

6.2 Validación

Tidymodels ofrece nuevamente un paquete con la finalidad de simplificar el procedimiento de validación de un modelo de forma que recoja diferentes métodos como *Bootsrap*, *validación cruzada* u otros en un mismo paquete denominado **rsampler** (mencionado previamente). Para ello es necesario definir un objeto **resampler** el cual contiene la información asociada los repartos de los conjuntos de datos.

A continuación se muestra con una validación cruzada sencilla mediante la función **vfold_cv()**:

```
fold <- vfold_cv(data = data_train_prep, v = 5, repeats = 10, strata = Survived)
```

Una vez definidas las particiones, se emplea la función **fit_resamples()** para ajustar el modelo:

```
modelo_rpart_val_fit <- fit_resamples(object = modelo_rpart,
                                      preprocessor = receta,
                                      resamples = fold,
                                      metrics = metric_set(roc_auc),
                                      control = control_resamples(
                                        save_pred = TRUE))
```

Los resultados se almacenan en forma de **tibble**, donde las columnas contienen la información sobre cada partición: su id, las observaciones que forman parte, las métricas calculadas, si ha habido algún error o warning durante el ajuste, y las predicciones de validación si se ha indicado y se obtiene la información **collect_predictions()** y **collect_metrics()**. Además también es posible indicarle el preprocesamiento en el mismo ajuste, lo cual resulta finalmente bastante cómodo.

```
modelo_rpart_val_fit %>% collect_metrics(summarize = TRUE)
```

```
## # A tibble: 1 x 6
##   .metric .estimator  mean     n std_err .config
##   <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1 roc_auc binary     0.703    50  0.0106 Preprocessor1_Model1
```

Por último cabría destacar que para todos los métodos de **resampling** es posible una paralelización de los mismos ya que **Tidymodels** a traves de sus paquetes soporta dicha funcionalidad, la cual es una gran ventaja frente a otras librerías parecidas.

6.3 Ajuste de hiperparámetros o Tuning

Muchos de los modelos contienen lo que se denominan **hiperparámetros**, los cuales son parámetros que no puden ser aprendidos sino que deben ser establecidos por un profesional, además es posible definir de forma sencilla estos parámetros gracias a la función **tune()** que ofrece el paquete **tune**, y que permite sin gran esfuerzo explorar una gran cantidad de valores para poder encontrar los mejores parámetros posibles para el modelo.

Una vez definido el modelo y ajustados los parámetros a explorar con la función **tune()** (también es posible detallarla en mayor medida), quedaría definir el modelo con la función **tune_grid()** que permite establecer el número de combinaciones generadas automáticamente (por supuesto también paralelizable).

Los resultados de la búsqueda de hiperparámetros pueden verse con las funciones auxiliares **collect_metrics()**, **collect_predictions()**, **show_best()** y **select_best()**.

```
modelo_rpart_tune_fit %>% show_best(metric = "roc_auc", n = 5)
```

```
## # A tibble: 5 x 8
##   tree_depth min_n .metric .estimator  mean     n std_err .config
##        <int> <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1          7    10 roc_auc binary     0.672    50  0.0166 Preprocessor1_Model01
## 2          4    25 roc_auc binary     0.672    50  0.0166 Preprocessor1_Model02
## 3         13    38 roc_auc binary     0.672    50  0.0166 Preprocessor1_Model03
## 4         12    31 roc_auc binary     0.672    50  0.0166 Preprocessor1_Model04
## 5         10    35 roc_auc binary     0.672    50  0.0166 Preprocessor1_Model05
```

Una vez realizado todo este procedimiento, es tan simple como obtener los mejores hiperparámetros con la función **select_best()** y aplicarlos al modelo con la función **finalize_model()**:

```
hiperpara_finales <- select_best(modelo_rpart_tune_fit, metric = "roc_auc")

modelo_final <- finalize_model(x = modelo_rpart, parameters = hiperpara_finales)

modelo_final_fit <- modelo_final %>% fit(formula = Survived~., data_train_prep)
```

6.4 Predicción

Para realizar las predicciones sobre el conjunto de test es tan simple como emplear la función **predict** que ya conocemos de prácticas previas, por lo que sería de la siguiente forma:

```
predicciones <- modelo_final_fit %>%
                predict(
                  new_data = data_test_prep
                )
```

7 Validación de resultados

Como cabría esperar **Tidymodels** ofrece nuevamente una serie de funciones que permiten mediante el paquete **yardstick** ofrecer una serie de métricas que permitan comprobar el desempeño del modelo en base al conjunto de test que se ha predicho. Para ello utiliza la función **metrics()**, la cual es personalizable y permite definir una serie de medidas que se ajustan dado el tipo de modelo.

Otra de las cosas más interesante de **metrics()** es que permite configurar directamente la fuente de verdad de las clases que se pretenden obtener frente a la clases estimada por lo que resulta realmente cómodo utilizar dicha función para validar.

```
modelo_final_fit %>%
  predict(data_test_prep) %>%
  bind_cols(data_test_prep) %>%
  metrics(truth = Survived, estimate = .pred_class)
```

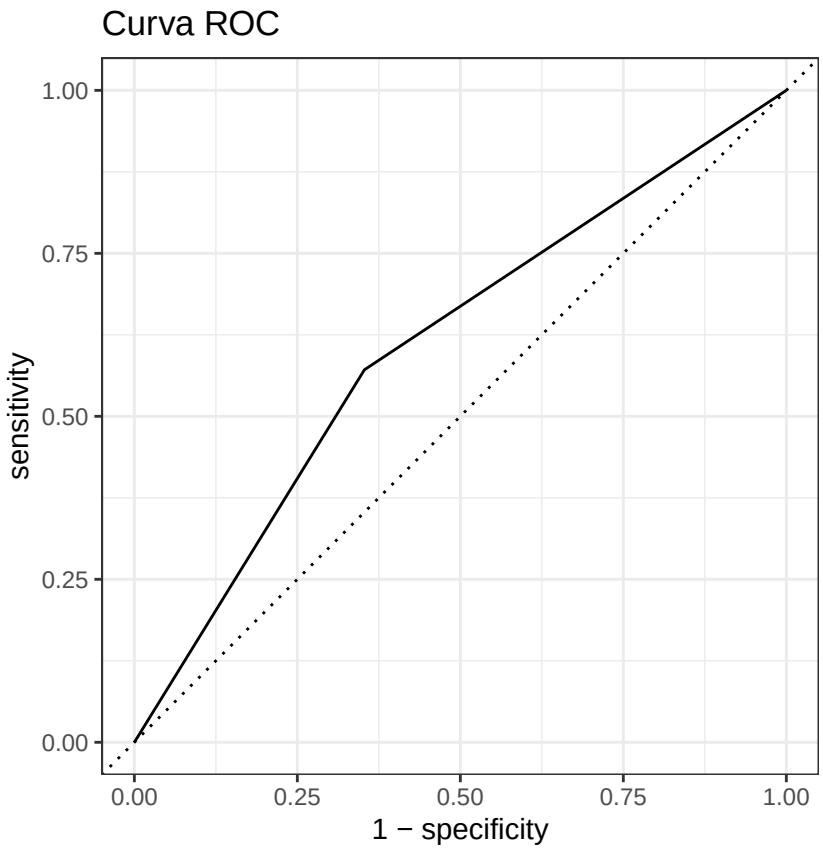
```
## # A tibble: 2 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy binary         0.625
## 2 kap      binary         0.194
```

Como es lógico también podemos obtener las probabilidades de las predicciones en lugar de simplemente una asignación de clase:

```
pred_prob <- modelo_final_fit %>%
  predict(data_test_prep, type = "prob") %>%
  bind_cols(data_test_prep)
```

Y si queremos podemos mostrar la curva **ROC** directamente con la función **roc_curve()** usando **autoplot()**:

```
pred_prob %>%
  roc_curve(Survived, .pred_0) %>%
  autoplot()+
  labs(title = 'Curva ROC')
```



8 Workflows

Los **workflows** permiten combinar en un solo objeto todos los elementos que se encargan del preprocesamiento (**recipes**) y modelado (**parsnip** y **tune**). Para crear el **workflow** se van encadenando los elementos con las funciones **add_*** o bien modificando los elementos ya existentes con las funciones **update_***.

```
modelo_rpart <- decision_tree(
                  mode       = "classification",
                  tree_depth = tune(),
                  min_n      = tune()
                ) %>%
                set_engine(engine = "rpart")

receta <- recipe(
  formula = Survived ~.,
  data = data_train
) %>%
  # Omisión de valores perdidos
  step_naomit(all_predictors()) %>%
  # Exclusión variables varianza cercana a cero
  step_nzv(all_predictors()) %>%
  # Normalización max min
  step_range(all_numeric(), -all_outcomes(), min = 0, max = 1)
```

```
fold <- vfold_cv(data = data_train_prep, v = 5, repeats = 10, strata = Survived)

workflow_modelo <- workflow() %>%
                     add_recipe(receta) %>%
                     add_model(modelo_rpart)

workflow_modelo
```

```
## == Workflow ====================================================================
## Preprocessor: Recipe
## Model: decision_tree()
##
## -- Preprocessor ----------------------------------------------------------------
## 3 Recipe Steps
##
## * step_naomit()
## * step_nzv()
## * step_range()
##
## -- Model -----------------------------------------------------------------------
## Decision Tree Model Specification (classification)
##
## Main Arguments:
##   tree_depth = tune()
##   min_n = tune()
##
## Computational engine: rpart
```

Cabe destacar que se podrían aplicar técnicas de **tuning** con la función **tune_grid()** sobre el modelo del workflow tal y como si de un modelo normal se tratase, por lo que resulta claramente cómodo agrupar funcionalidades en flujos de trabajo. Por otro lado, para seleccionar el mejor modelo funcionará exactamente igual que como si de un modelo normal se tratase, pero en lugar de finalizar el modelo, se finaliza el workflow con **finalize_workflow()** y se obtiene el modelo con **pull_workflow_fit()**.

```
modelo_final_fit <- finalize_workflow(
                        x = workflow_modelo,
                        parameters = hiperpara_finales
                    ) %>%
                    fit(
                      data = data_train_prep
                    ) %>%
                    pull_workflow_fit()

modelo_final_fit
```

```
## parsnip model object
##
## Fit time:  0ms
## n= 159
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
```

```
##
## 1) root 159 53 1 (0.3333333 0.6666667)
##   2) Name=Allison, Miss. Helen Loraine,Allison, Mrs. Hudson J C (Bessie Waldo Daniels),Baxter, Mr. Qu
##   3) Name=Allen, Miss. Elisabeth Walton,Allison, Master. Hudson Trevor,Anderson, Mr. Harry,Andrews, M
```