Máster Universitario en Ingeniería Informática

SISTEMAS INTELIGENTES PARA LA GESTIÓN EN LA EMPRESA

Trabajo téorico:
Programación con TidyModels

UNIVERSIDAD DE GRANADA

Carlos Morales Aguilera
75925767-F
carlos7ma@correo.ugr.es

Curso Académico 2020-2021

Índice

1. Introducción **3**

1.1. Paquetes 4

1.1.1. tidymodels 4

1.1.2. rsample 4

1.1.3. recipes 5

1.1.4. parsnip 5

1.1.5. tune 6

1.1.6. yardstick 6

1.1.7. broom 7

1.1.8. dials 7

2. Particionamiento de datos **8**

3. Preprocesamiento **9**

3.1. Técnicas de preprocesamiento 9

3.1.1. Omisión de valores perdidos 9

3.1.2. Imputación de valores perdidos 9

3.1.3. Exclusión variables con varianza cercana a cero 9

3.1.4. Normalización de variables 10

3.1.5. Binarización de variables 10

3.2. ¿Cómo funciona recipes? 10

4. Modelado **12**

4.1. Entrenamiento de modelos 12

4.2. Validación 13

4.3. Ajuste de hiperparámetros 14

4.4. Predicción 15

5. Validación resultados **16**

6. Workflows **18**

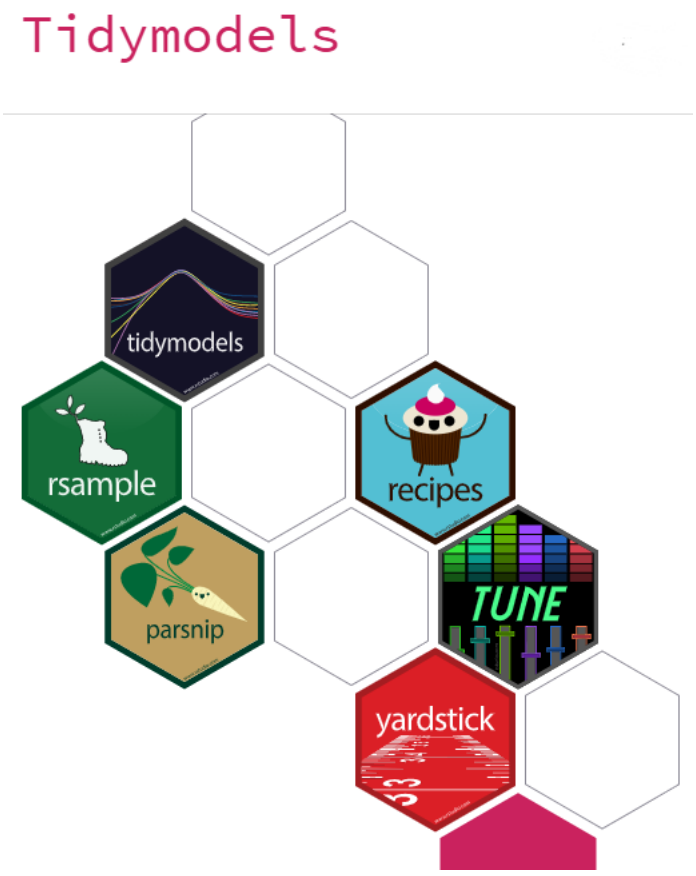
7. Conclusiones **20**

8. Bibliografía **21**

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.

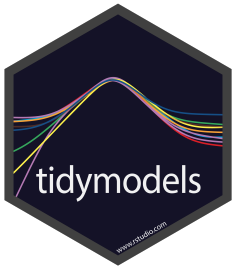


1.1. Paquetes

Por lo tanto cabe destacar una serie de paquetes que componen toda la funcionalidad que compone **tidymodels**.

1.1.1. tidymodels

Como se ha definido previamente, es un *meta-paquete* que instala y carga las dependencias de los diferentes paquetes que componen su núcleo. Una vez cargado este paquete quedarían disponibles las funcionalidades de los siguientes paquetes listados.



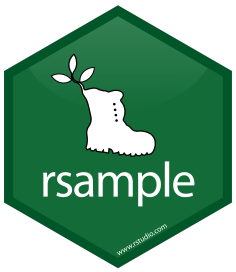
1.1.2. rsample

rsample contiene un conjunto de funciones para crear diferentes tipos de remuestreos y las clases correspondientes para su análisis.

Al final, las principales funcionalidades con las que cumple son:

- Ofrecer técnicas tradicionales de remuestreo que permiten estimar la distribución muestral de una estadística.
- Estimar el rednimiento de un modelo utilizando conjuntos de reserva (particionamiento de datos).

El objetivo proporcionar los bloques de construcción básicos para crear y analizar muestras repetidas de un conjunto de datos, pero no incluye código para modelar o calcular estadísticas.



1.1.3. recipes

Es un método alternativo para crear y preprocesar matrices de diseño que se pueden utilizar para modelado y considerando que en R existen algunas limitaciones a lo que puede hacer la infraestructura existente este paquete resulta realmente útil.

La idea del paquete es definir una **receta** o modelo que se pueda utilizar para definir secuencialmente las codificaciones y el preprocesamiento de los datos, al igual que una receta de cocina.

1.1.4. parsnip

El objetivo es proporcionar una interfaz unificada y ordenada a los modelos que se puede utilizar para probar una variedad de modelos sin atascarse en las minucias sintácticas de los paquetes subyacentes. Básicamente es un competidor directo de **caret**.

Los objetivos son:

- Separar la definición de un modelo de su evaluación.
- Separar la especificación de un modelo de la implementación (engine).
- Unificar los nombres de los parámetros de cada implementación utilizando nombres comunes en base a la especificación del modelo.

1.1.5. tune

El objetivo de **tune** es facilitar el ajuste de los **hiperparámetros** de los modelos definidos en los paquetes de **tidymodels**. Para ello incorpora funcionalidades de *recipes*, *parsnip* y *dials*.

Además, al incorporarse con estos paquetes es capaz de crear ajustes de modelos utilizando directamente objetos definidos con los paquetes anteriores, resumiendo la cantidad de operaciones enormemente.

1.1.6. yardstick

Este paquete utiliza una serie de herramientas sencillas para la estimación de como de bien se ajustan/trabajan los modelos utilizando los principios de *tidy data*. Para ello utiliza objetos como pueden ser *matrices de confusión*, *curvas ROC* o métricas de regresión.

Permite además mantener ajustes y métricas de diferentes modelos por lo que resulta realmente interesante frente a otros paquetes con funcionalidades similares.

1.1.7. broom

Este paquete resume la información clave de los modelos en objetos *tibble*. Proporciona tres verbos principales:

- **tidy()** resume la información de los componentes de un modelo.
- **glance()** reporta la información completa del modelo.
- **augment()** añade información sobre observaciones a un dataset.

1.1.8. dials

Este paquete contiene herramientas para crear y administrar valores de parámetros de ajuste y está diseñado para integrarse bien con el paquete de **parsnip**.

El nombre refleja la idea de que ajustar modelos predictivos puede ser como girar un conjunto de diales en una máquina compleja bajo presión.

2. Particionamiento de datos

Uno de los principales puntos a la hora de trabajar con grandes conjuntos de datos y evaluar modelos es realizar una partición de los datos para obtener conjuntos de muestra sobre el que aprender o también llamado conjunto de **entrenamiento**, mientras que por otro lado deben existir conjuntos de comprobación o de **test**.

En R existen diversas herramientas para realizar esta labor, pero **tidymodels** a través del paquete **rsample** ofrece diversas funcionalidades con este fin, encargándose de hacer las particiones de los datos dada una proporción.

Una de las principales funciones es `initial_split()`, la cual obtiene un conjunto de datos, junto a la proporción. Otra de las ventajas más destacables sería la posibilidad de indicar una variable sobre la que estratificar los submuestreos (tratar de balancear ambos conjuntos).

Un ejemplo de implementación con una proporción sería 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)
```

Evidentemente existen diferentes configuraciones y personalizaciones de las funciones, pero esto requiere un aprendizaje más profundo y con esta funcionalidad mostrada se podría cumplir con la mayoría de particiones que se llegan a realizar en un problema de *Machine Learning*.

Se observa que destaca no solo por la claridad del ejemplo, sino por la facilidad a la hora de emplear dichas funciones.

3. Preprocesamiento

La principal fase de cara a obtener unos datos fiables, con alto interés y que realmente resulten útiles, interpretables y valiosos para un estudio, pertenecen a la fase de **preprocesamiento**. Para esta fase existen diferentes técnicas y operaciones que se pueden realizar sobre los datos, existiendo para ello una infinidad de paquetes en R.

Tidymodels ofrece para ello una interfaz unificada con una serie de herramientas de manejo muy similar dentro del paquete **recipes**. Siguiendo la lógica de una *receta*, esta es una construcción mediante una serie de pasos, entre los que se podrían incluir entre otros:

- 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.

3.1. Técnicas de preprocesamiento

3.1.1. 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.

3.1.2. 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.

3.1.3. 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.

3.1.4. 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**.

3.1.5. 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).

3.2. ¿Cómo funciona 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.

Comparando con su homólogo en la cocina, para realizar una receta se posee una base o ingredientes (siendo estos el conjunto de datos) y una serie de pasos o procedimientos para elaborar la receta (transformaciones o preprocesamientos), obteniendo un plato final (conjunto de datos preprocesado).

Por lo tanto a continuación se aplican algunas de dichas transformaciones a modo de ejemplo:

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

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)
```

A continuación se muestra que contendría realmente la *receta* que se ha definido:

```
Data Recipe

Inputs:


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()
```

4. Modelado

En este ámbito es donde principalmente destaca **tidymodels**, donde se compara con otras herramientas como **caret**. Pretende ofrecer una interfaz que se abstrae de implementaciones y pretenda facilitar esta labor lo máximo posible al usuario.

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 hiperprámetros**: 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.

4.1. Entrenamiento de modelos

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**.

1. **Modelo**: Modelo escogido, donde de forma abstraida únicamente se indica el tipo, como puede ser por ejemplo `decision_tree()` para un árbol de decisión.
2. **Engine**: En este punto se define la implementación exacta a utilizar del modelo indicado previamente. En el caso del árbol de decisión, un ejemplo sería `rpart`.
3. **Ajuste**: Se ajusta el modelo mediante configuraciones sobre el mismo y sus atributos, siendo estos genéricos e independientes de la implementación, ya que hace la asignación por debajo y el programador se despreocupa de dicha labor.

Un ejemplo de implementación de un árbol de decisión con `rpart` sería:

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

A continuación **parsnip** 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). En caso de ajustar el modelo anteriormente descrito ser utilizaría `fit()` de la siguiente forma:

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

4.2. Validación

Otra de las labores más conocidas de cara al refinamiento de un modelo es la realización de pruebas de validación del modelo antes de probarlo directamente sobre el conjunto de prueba.

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, siendo nuevamente **rsampler**. Para ello es necesario definir un objeto **resampler** el cual contiene la información asociada los repartos de los conjuntos de datos.

En el caso de querer realizar una validación cruzada se utilizaría por ejemplo la función `vfold_cv()`:

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

Otra de las novedades de **tidymodels** es la posibilidad de realizar un ajuste parecido a la función `fit()` pero con la posibilidad de aplicar la validación mediante la función `fit_resamples()`.

```
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)
```

Un ejemplo de como se verían dichas métricas sería:

A tibble: 1 x 6

.metric <chr>	.estimator <chr>	mean <dbl>	n <int>	std_err <dbl>	.config <chr>
roc_auc	binary	0.644746	50	0.01573879	Preprocessor1_Model1

1 row

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.

4.3. Ajuste de hiperparámetros

Una de las siguientes tareas que se resuelven es el ajuste de hiperparámetros o **tuning** de modelos. 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).

Al igual que `fit_resamples()`, la función `tune_grid()` también recibe parámetros como preprocesamiento, conjuntos de validación, métricas y evidentemente el número de combinaciones.

Un ejemplo de tuning sobre el modelo anterior sería:

```
modelo_rpart_tune_fit <- tune_grid(object = modelo_rpart,
                                   preprocessor = receta,
                                   resamples = fold,
                                   metrics = metric_set(roc_auc),
                                   control = control_resamples(
                                     save_pred = TRUE),
                                  grid = 10)
```

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

4.4. Predicción

Este probablemente sea el procedimiento más sencillo, ya que **tidymodels** es compatible con la función de R `predict()`, por lo que simplemente habría que indicarle el modelo para obtener las predicciones.

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

5. Validación resultados

Al igual que en las tareas previas, para la labor de validar finalmente los resultados y el rendimiento de los modelos, **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)
```

Y podríamos observar los resultados de la siguiente forma:

A tibble: 2 x 3

.metric <chr>	.estimator <chr>	.estimate <dbl>
accuracy	binary	0.6857143
kap	binary	0.3529412

2 rows

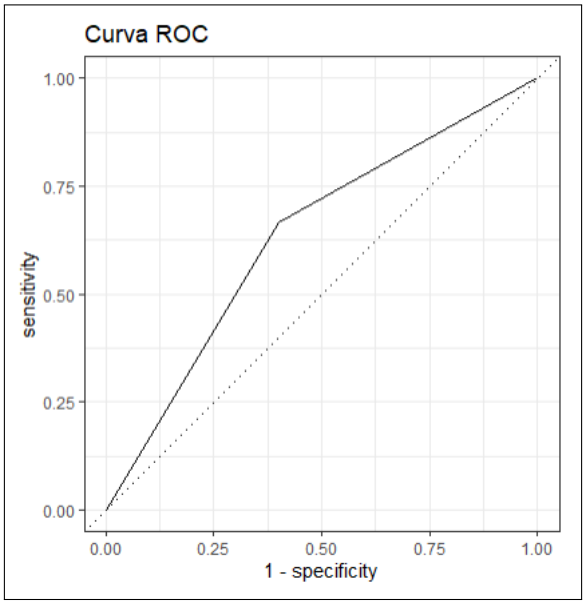
Como es lógico también podemos obtener las probabilidades de las predicciones en lugar de simplemente una asignación de clase:

```
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')
```

Y se visualizaría de la siguiente forma:



6. Workflows

Una de las principales ventajas de **tidymodels** frente a otras librerías con funcionalidades similares es que ofrece la posibilidad de definir flujos de trabajo o **workflows**. Estos flujos de trabajo consisten en la combinación de diferentes modelos u objetos de los mencionados previamente como preprocesamiento (**recipes**) o modelado (**parsnip** y **tune**).

La definición de un **workflow** es una tarea sencilla consistente en la encadenación de los elementos con las funciones que empiezan por la nomenclatura `add_*` o bien modificando elementos ya existentes con las funciones `update_*`.

Un ejemplo de un **workflow** con toda la funcionalidad descrita anteriormente sería:

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

receta <- recipe(
  formula = Survived ~.,
  data = data_train
) %>%
  # Omision de valores perdidos
  step_naomit(all_predictors()) %>%
  # Exclusion variables varianza cercana a cero
  step_nzv(all_predictors()) %>%
  # Normalizacion 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)
```

Y se define el **workflow** con `add_recipe()` y `add_model()`:

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

Y el objeto **workflow** contendría la siguiente información:

```
== 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 (ya que se comporta como tal), 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()
```

7. Conclusiones

A lo largo de este trabajo teórico-práctico se puede apreciar (de una forma bastante superficial, ya que es una herramienta mucho más potente) el alcance básico de **tidymodels**. Como se puede observar es una herramienta que compite directamente con **caret**, pero que a su vez engloba una mayor funcionalidad al recoger tareas como preprocesamiento, métricas o particiones de datos, entre otros.

Se puede observar que **tidymodels** funciona claramente como un marco de trabajo sencillo que requiere una baja curva de aprendizaje ya que facilita todas las funcionalidades lo máximo posible pero sin llegar a limitar, y permitiendo profundizar en las mismas.

Entre las principales ventajas de **tidymodels** se encuentran:

- La facilitación de una interfaz única sobre la que trabajar todos los procesos correspondientes a un trabajo de *Machine Learning*.
- La baja curva de aprendizaje que requiere dicha herramienta.
- La modularización del paquete y la permisibilidad a la hora de llamar a métodos de diferentes paquetes como si se comprendieran en uno mismo (se puede apreciar por ejemplo con la función `tune_grid()`.
- La facilitación de una herramienta que admite paralelismo.
- No se trata de una herramienta con alta verborrea o con dificultad de comprensión, ya que las funciones utilizan verbos comprensibles y los paquetes siguen una lógica en base a lo que los representa (recetas por ejemplo).

Aunque existen una gran seria de ventajas adicionales, resulta imposible nombrarlas todas en un trabajo tan breve, pero sirve de introducción a esta poderosa herramienta.

Como conclusiones personales añadiría que es una herramienta bastante actualizada y moderna, que resulta interesante aprender y que tiene un futuro prometedor, por lo que junto con otras librerías como **caret** tiene un gran sentido que sean explicadas.

Tras haber trabajado con **caret** y **tidymodels**, queda claro el amplio abanico de herramientas para tareas de *Machine Learning* y lo interesante que resulta este ámbito, cada vez con herramientas que facilitan el diseño y desarrollo y facilitan al analista dichas tareas para que pueda centrarse en mayor medida en el problema.

Me gustaría finalmente concluir con que es una herramienta muy potente, interesante y útil, que creo que merece ser estudiada y que puede ser útil en nuestro futuro profesional como *Data Scientist*.

8. Bibliografía

[1] Tidymodels - Página oficial

[2] Tidymodels Packages - Página oficial

[3] Tidymodels GitHub - Repositorio oficial del proyecto

[4] Tidymodels R - Paquete oficial de R

[5] Tidy Models, una introducción - Diego Kozlowski y Juan Manuel Barriola

[6] Machine Learning con R y tidymodels - Joaquín Amat Rodrigo