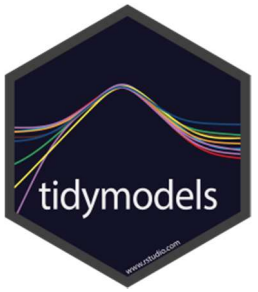


Guía Rápida para el uso de *tidymodels*

Grupo de Usuarios de R
Madrid

24 de septiembre de 2025

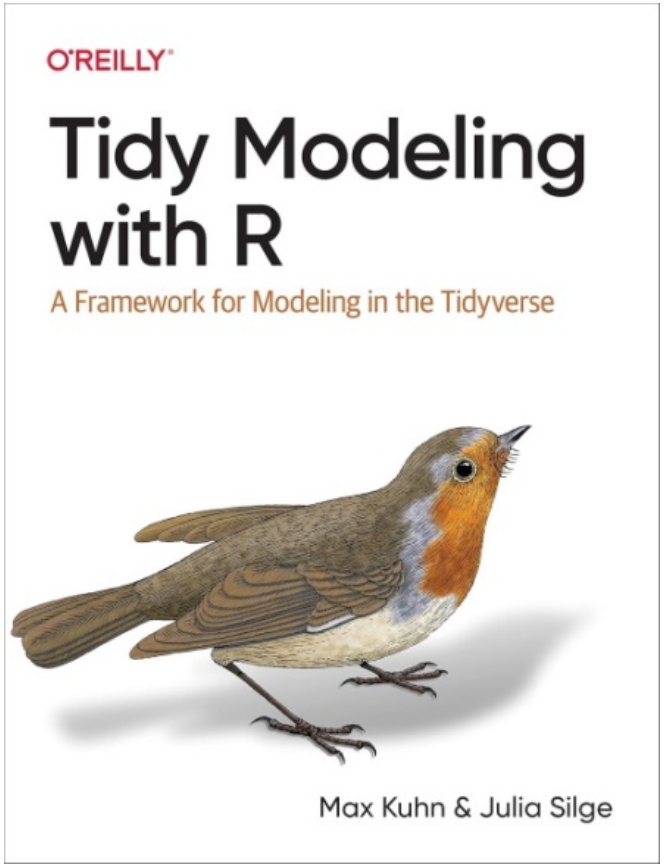
Jesús Herranz Valera
jherranzvalera@gmail.com
Bioestadístico GEICAM

Recursos

- Max Kuhn & Julia Silge. ***"Tidy Modeling with R".*** O'Reilly, 2022 (***tidymodels)*** https://www.tmwr.org/

- Página web de *tidymodels*: https://www.tidymodels.org/

- Hadley Wickham & Garrett Grolemund. ***"R for Data Science".*** O'Reilly, 2017 (***tidyverse)*** https://r4ds.had.co.nz/



De qué va esta presentación ?

- Se pretende mostrar, de una forma sencilla y resumida, las principales **claves** para poder **construir un modelo predictivo** con ***tidymodels*** incluyendo las etapas más importantes
  - **Pre-procesamiento** de datos
  - Partición de los datos en una muestra de **training** y otra de **testing**
  - **Técnicas de remuestreo**. Validación cruzada
  - **Optimización de parámetros** de un modelo
  - Construcción del **modelo final** con los parámetros óptimos
  - **Predicciones** y **evaluación** del modelo en la muestra de testing

- Ejemplo con **Regresión Penalizada**, **Elastic Net.** Se optimizarán los parámetros alpha (compromiso entre lasso y ridge), y la penalización lamba
- Problema de **Regresión**, con una variable respuesta continua

Introducción: cómo surge *tidymodels* ?

- Una de las principales fortalezas de R como software de análisis de datos es la enorme **cantidad de paquetes** que se han desarrollado, pero una de sus principales dificultades es la **poca estandarización** que se ha seguido en el desarrollo de esos paquetes
  - Llamadas a las funciones. Nombres de parámetros
  - Entrada de los datos a analizar: fórmulas, *data frame*, matrices y vectores, …
  - Diferentes formas de presentar los resultados. Función *predict()*

- ***caret*** ("*Classification And REgression Training*") es un paquete que se diseñó para entrenar modelos de machine learning en R de manera sencilla y estandarizada
- Otras iniciativas como ***mlr*** o ***h2o***

- ***tidymodels*** es un conjunto de paquetes en R que sirve para **construir modelos** de machine learning de forma más **ordenada** y **sistemática**, siguiendo la filosofía y sintaxis de ***tidyverse***

tidyverse

- **TIDY** significa limpio, ordenado, metódico
- ***tidyverse*** es un **conjunto de paquetes de R** diseñados para **Ciencia de Datos**, que comparten filosofía, gramática y estructura de datos
- ***tidyverse*** es un **dialecto de R** que facilita la comprensión del código, y lo hace más intuitivo

- Está enfocado en “verbos”, que indican qué es lo que se desea hacer. Usa **nombres sencillos** de funciones
- Es **simple**: cada función tiene un único objetivo
- Es **consistente**: los datos van siempre primero
- Uso del operador **pipe %>% , tubo,** para encadenar funciones de R

- Uso de ***tibble***, una versión modernizada de los *data frames*

Paquetes de *tidyverse*

Paquete	Descripción
readr	Importación y exportación de ficheros
dplyr	Manipulación y transformación de datos
tidyr	Conversión y organización de datos
tibble	Manejo de los *data frames* de tipo *tibble*
stringr	Manejo de variables de tipo *string*
forcats	Manejo de factores
magrittr	Manejo del operador *pipe*
purrr	Herramientas para programación, basadas en funciones de la familia *map()*
ggplot2	Construcción de gráficos de alta calidad

Operador pipe %>%

- El operador **pipe**, del paquete ***magrittr***, es una herramienta para encadenar una **secuencia de funciones de R**
- La idea es crear una **tubería de operaciones y funciones** (**pipeline**), de forma secuencial
- La **salida** de cada función, es decir lo que retorna, es la **entrada** de la siguiente función

- Es fácil de leer
- El foco está en cómo **los usuarios** interaccionan con el software

- Los **pipelines** son muy útiles para crear ***workflows***, porque establecen muy bien el orden, la **secuencia**, en la que se hacen las tareas

tidymodels

- ***tidymodels*** es un conjunto de **paquetes de R**, diseñados especialmente para la **construcción de modelos predictivos**
- El objetivo principal es producir **modelos estadísticos** y de **machine learning** de **alta calidad**

- Sigue la filosofía y el dialecto de ***tidyverse***, es decir, aplica sus principios para la construcción de modelos
- Al igual que ***tidyverse***, recoge el objetivo general de estar "**diseñado para humanos**", en el sentido de estar diseñado para usuarios, no para los desarrolladores de paquetes

- Cada paquete recoge **funciones específicas** de **cada paso** de la construcción de los modelos, lo que facilita el **mantenimiento**

- Web: https://www.tidymodels.org/

tidymodels

- ***tidymodels*** proporciona un **interfaz único** para llamar a distintas funciones y **paquetes de R**, que son los que **construyen los modelos**, y también, proporciona un interfaz único para **explorar los resultados**

- Usa **nombres de funciones** conocidos y reconocibles

- Los **valores por defecto** de algunos parámetros son muy importantes. Algunos no deben tener valores por defecto, para forzar al usuario a elegir
- Otros parámetros pueden ser **derivados de los datos**. Por ejemplo, si un modelo es de clasificación o regresión, según si la variable respuesta es un factor o no

- Algunos paquetes requieren **estructuras de datos** concretas (matrices, fórmulas, …). Se debe ser versátil para poder trabajar con cualquiera de ellas

Paquetes de *tidymodels*

Paquetes de *tidymodels*

Paquete	Descripción
rsample	División de los datos en submuestras, y control de las técnicas de remuestreo
parsnip	Construcción de los modelos
recipes	Pre-procesamiento de los datos
workflows	Integra el preprocesamiento, el modelado y el post-procesamiento
yardstick	Medidas de rendimiento predictivo de los modelos
tune	Optimización de los parámetros del modelo
dials	Tratamiento de los parámetros de tuning en rejillas
broom	Convierte la información en formatos más manejables

Ejemplo: *tidymodels*

```
> library(tidymodels)
-- Attaching packages --------------------------------------- tidymodels 1.0.0 --
v broom        1.0.4     v recipes      1.0.5
v dials        1.2.0     v rsample      1.2.1
v dplyr        1.1.1     v tibble       3.2.1
v ggplot2      3.4.1     v tidyr        1.3.0
v infer        1.0.4     v tune         1.1.1
v modeldata    1.1.0     v workflows    1.1.4
v parsnip      1.2.1     v workflowsets 1.0.1
v purrr        1.0.1     v yardstick    1.1.0
-- Conflicts ------------------------------------------ tidymodels_conflicts() --
x purrr::discard() masks scales::discard()
x dplyr::filter()  masks stats::filter()
x dplyr::lag()     masks stats::lag()
x recipes::step()  masks stats::step()
* Dig deeper into tidy modeling with R at https://www.tmwr.org
There were 15 warnings (use warnings() to see them)
> tidymodels_prefer()
```

- Al cargar la librería ***tidymodels***, se informa de todas las librerías cargadas, que incluye también algunos de los paquetes de ***tidyverse*** como ***dplyr*** o ***ggplot2***
- Se informa de los conflictos, es decir, funciones que están repetidas en varios paquetes con el mismo nombre
- La función ***tidymodels_prefer()*** resuelve los conflictos a favor de los paquetes de ***tidymodels*** y de ***tidyverse***

Pre-procesamiento

- Las **técnicas de pre-procesamiento** de datos se refieren al **tratamiento previo** de las observaciones y de las variables en el conjunto de datos antes de la elaboración de un modelo
  - Suelen venir determinado por la técnica que se utilice para la construcción del modelo predictivo

- Algunas de las **técnicas** más usadas de **pre-procesamiento** de datos:
  - Creación de variables dummy
  - Estandarización de las variables continuas a media 0 y desviación típica 1
  - Eliminar algunas de las variables que estén muy correlacionadas
  - Eliminar variables donde casi todos los valores son iguales (desbalanceadas)
  - Unir categorías con frecuencia muy baja
  - . . . .

Paquete *recipes* de *tidymodels*

- El paquete ***recipes*** puede combinar en un **único objeto** distintas técnicas de **tratamiento** de variables y **tareas de pre-procesamiento** de datos
- Un ***recipe***, **"receta",** es un objeto que incluye la definición de los **pasos del pre-procesamiento de datos**, y el **orden** en el que se procesan
  - Es **una especificación** de los pasos a seguir, pero **no se ejecutan** en el momento de definirlos
  - Este objeto puede ser aplicado después a diferentes **conjuntos de datos**

- Los pasos del pre-procesamiento se definen con **funciones** de tipo ***step_*()***

- A las variables implicadas se les llama **"ingredientes"** de la receta, y pueden ser la variable respuesta, "outcome", o las variables predictoras, variables que tienen **diferentes roles**
  - Las variables pueden ser referenciados en conjunto, con funciones del tipo: ***all_predictors(), all_numeric_predictors(), all_nominal_predictors(), all_numeric(), all_nominal(), all_outcomes(), ….***

Funciones *step_*()*

Función	Descripción
step_normalize()	Normaliza las variables numéricas a media 0 y desviación estándar 1
step_BoxCox() ***step_YeoJohnson()***	Transformaciones de Box y Cox, y de Yeo-Johnson, para conseguir simetría
step_corr()	Borra variables con fuertes correlaciones con las demás
step_pca() ***step_pls()***	Sustituye las variables originales por las componentes principales o componentes PLS (feature extraction)
step_interact()	Crea términos de interacción entre las variables predictoras
step_ns()	Crea términos del tipo *spline*, para relaciones no lineales

Las funciones de tipo ***step_*()*** están en: https://www.tidymodels.org/find/recipes/

Funciones *step_*()*

Función	Descripción
step_zv() ***step_nzv()***	Elimina las variables con varianza cero y varianza "casi cero"
step_dummy()	Crea variables dummy para las variables categóricas
step_unknow()	Crea un nivel para los NAs en una variable categórica
step_other()	Crea una categoría "other" uniendo las categorías que tienen poca frecuencia
step_impute_XX()	Diferentes métodos de imputación de missings (knn, mean, median, mode, linear, …)

Funciones *step_*()* del paquete *themis*

Función	Descripción
step_downsample()	Submuestreo. Toma una submuestra aleatoria de la clase más numerosa para balancear las categorías
step_upsample()	Sobremuestreo. Replica observaciones de la clase menos numerosa para balancear las categorías
step_smote()	Método híbrido

- El paquete ***themis*** tiene funciones del tipo ***step_*()*** especializadas en **muestreo** para tratar variables con **clases desbalanceadas**
- Todas estas funciones tienen un parámetro***, skip=TRUE***, que indica que solo se aplican a la muestra de training. Es decir, estos procesamientos serán ignorados cuando se usa la función ***predict()***, ya que no tiene sentido hacer un muestreo en los datos donde se desea predecir

Paquete *recipes* de *tidymodels*

- Las **operaciones básicas** que hay que hacer con un objeto ***recipe*** son las siguientes:
  - La función ***recipe()*** crea el objeto, **especificando los pasos** que se desean aplicar. Es dónde se asignan los **roles de las variables**, indicando cuál es la variable respuesta y cuáles son las variables predictoras, lo que se puede hacer con una fórmula
  - La función ***prep()*** prepara el objeto para poder aplicarlo. **Calcula las operaciones** que son necesarias. Hay que indicar el *data frame* dónde se deben hacer los cálculos
  - La función ***bake()*** **aplica** este último objeto a un ***data frame*** concreto, creando **otro *data frame*** con los datos procesados

Fichero de datos: Solubility

- **Objetivo del estudio:** estudiar la relación entre **la estructura** y **propiedades** de un conjunto de **compuestos químicos** y su **solubilidad**

Nombre	Descripción	Categorías / Observaciones
FP001 – FP208	208 fingerprints	Variables binarias que indica la presencia o ausencia de una subestructura química
MolWeight	Peso molecular	
NumAtoms - NumRings		16 descriptores de la molécula, de tipo “conteo” (número de átomos, de anillos, …)
HydrophilicFactor SurfaceArea1 SurfaceArea2		3 Descriptores continuos
Solubility	**Solubilidad**	**Solubilidad del compuesto**

- Problema de **Regresión**, usando la variable respuesta continua “**Solubility**”
- Ejemplo extraído de *“Applied Predictive Modeling”. Max Kuhn*

Pre-procesamiento con un *recipe*

```
> load("D://Solubility Data.RData")
> dim(xx_sob_train)
[1] 884 229
> dim(xx_sob_test)
[1] 383 229
> ## 1.- Se crea la receta
> sol_rec <-
+   recipe( Solubility ~ . , data=xx_sob_train ) %>%
+   step_nzv(all_predictors(), freq_cut = 95/5, unique_cut = 10) %>%
+   step_corr(all_predictors(), threshold = 0.8 ) %>%
+   step_normalize(all_numeric_predictors())
>
```

- Se lee un fichero **RData** que contiene los *data frames* de **training** y **testing**
- Se define una **receta** con la función ***recipe*** donde se indica, con una **fórmula**, que la variable "*Solubility*" es la **variable respuesta** y el resto de variables del *data frame* "*xx_sob_train*" son **variables predictoras**. Además, deduce de este data *frame* qué variables son continuas y categóricas
- La receta incluy las **especificaciones** del **pre-procesamiento**, en tres pasos:
  - se quitan las variables con varianza casi zero, variables desbalanceadas
  - se quitan algunas de las variables muy correlacionadas
  - se estandarizan las variables predictoras continuas

Pre-procesamiento con un *recipe*

```
> sol_rec
-- Recipe

-- Inputs
Number of variables by role
outcome:     1
predictor: 228

-- Operations
* Sparse, unbalanced variable filter on: all_predictors()
* Correlation filter on: all_predictors()
* Centering and scaling for: all_numeric_predictors()

> ## 2.- Se prepara la receta
> sol_obj <- prep(sol_rec, training = xx_sob_train)
>
> ## 3.- Se aplica la receta a los dos data frames
> xx_sob_proc_train <- bake(sol_obj, xx_sob_train)
> xx_sob_proc_test <- bake(sol_obj, xx_sob_test)
>
```

- Se usa la función ***prep()*** para preparar la "receta", y se indica **el *data frame* de training,** dónde se deben hacer los **cálculos** de las medias y desviaciones típicas
- Se usa la función ***bake()*** para aplicar los cálculos a los dos *data frames*, creando otros dos ***data frames*** con los **datos procesados**

Pre-procesamiento con un *recipe*

```
> ## Chequeamos
> dim(xx_sob_train)
[1] 884 229
> dim(xx_sob_proc_train)
[1] 884 156
> dim(xx_sob_proc_test)
[1] 383 156
>
> ## Chequeamos
> mean(xx_sob_train$MolWeight)
[1] 200.4881
> sd(xx_sob_train$MolWeight)
[1] 97.55953
>
> mean(xx_sob_proc_train$MolWeight)
[1] -1.340124e-16
> sd(xx_sob_proc_train$MolWeight)
[1] 1
> mean(xx_sob_proc_test$MolWeight)
[1] -0.02420139
> sd(xx_sob_proc_test$MolWeight)
[1] 0.9486329
```

- Se comprueba que se han **quitado algunas variables** de los dos *dataframes*
- Se comprueba que los **cálculos de las estandarización** se han hecho en la muestra de training

Paquete *recipes* de *tidymodels*

- Es muy importante entender que el **pre-procesamiento** de datos forma parte del **proceso de modelado**

- La **opción recomendable** es incluir directamente el objeto ***recipe*** dentro de un ***workflow*** sin tener que crear *data frames* con los datos procesados
  - En este caso, no es necesario usar las funciones ***prep()*** y ***bake(),*** ya que se ejecutan internamente cuando se ejecutan los distintos pasos que se han integrado en el ***workflow***

- Es una buena opción usar el paquete ***recipes***, aunque no se desee usar ***tidymodels*** para modelar. En este caso, sí se salvarían los *data frames* con la función ***bake()***
  - Una de las principales ventajas de este paquete, es que un **único objeto** contiene **todos los pasos** del pre-procesamiento

Recomendaciones de Pre-procesamiento

Table A.1: Preprocessing methods for different models.

model	dummy	zv	impute	decorrelate	normalize	transform
C5_rules()	×	×	×	×	×	×
bag_mars()	✓	×	✓	o	×	o
bag_tree()	×	×	×	o[1]	×	×
bart()	×	×	×	o[1]	×	×
boost_tree()	×[2]	o	✓[2]	o[1]	×	×
cubist_rules()	×	×	×	×	×	×
decision_tree()	×	×	×	o[1]	×	×
discrim_flexible()	✓	×	✓	✓	×	o
discrim_linear()	✓	✓	✓	✓	×	o
discrim_regularized()	✓	✓	✓	✓	×	o
gen_additive_mod()	✓	✓	✓	✓	×	o
linear_reg()	✓	✓	✓	✓	×	o
logistic_reg()	✓	✓	✓	✓	×	o
mars()	✓	×	✓	o	×	o
mlp()	✓	✓	✓	✓	✓	✓
multinom_reg()	✓	✓	✓	✓	×[2]	o
naive_Bayes()	×	✓	✓	o[1]	×	×
nearest_neighbor()	✓	✓	✓	o	✓	✓
pls()	✓	✓	✓	×	✓	✓
poisson_reg()	✓	✓	✓	✓	×	o
rand_forest()	×	o	✓[2]	o[1]	×	×
rule_fit()	✓	×	✓	o[1]	✓	×
svm_*()	✓	✓	✓	✓	✓	✓

https://www.tmwr.org/pre-proc-table

Paquete *parsnip*

- El paquete ***parsnip*** proporciona una **interface estandarizada** para una gran variedad de **modelos predictivos**

- Todas las funciones de ***parsnip*** que construyen modelos tienen **dos parámetros** fundamentales:
- ***set_engine()*** es el "motor", dónde se especifica la **librería o función de R** con la que se va a construir el modelo
- ***set_mode()*** es dónde se especifica **el tipo de variable respuesta**, si es un problema de **clasificación** o **regresión**
  - No es necesario especificarlo si el modelo trabaja solo con un tipo de variable respuesta. Por ejemplo, regresión lineal o regresión logística
  - Si es un problema de **clasificación**, ***parsnip*** necesita que la variable sea definida como un **factor**

- Los **nombres de los parámetros** de las funciones de ***parsnip*** están unificados, para todos los paquetes que los usan

Paquete *parsnip*

- Hay paquetes de R que necesitan una **fórmula** para especificar el modelo, y otros requieren una **interface de tipo X / Y**, para indicar los predictores y la variable respuesta
- Las funciones de ***parsnip*** para **ajustar los modelos** pueden usar las dos maneras, independiente del paquete al que se llame (*set_engine*)
  - ***fit()*** construye un modelo especificando una fórmula
  - ***fit_xy()*** construye un modelo especificando las variables X / Y

- Ambas funciones devuelven un objeto de tipo ***model_fit***, sobre el que se pueden usar varias funciones:
  - ***extract_fit_engine()*** es la función para **extraer el objeto** del paquete de R (“motor”) que se ha especificado en el ***engine***
  - ***predict()*** es la función que se usa para obtener las **predicciones**

Predicciones con el paquete *parsnip*

- La función ***predict()*** de ***tidymodels*** proporciona una **forma única** de obtener las **predicciones**, independientemente del paquete con el que se ha creado el modelo
  - La mayoría de los paquetes de R tienen una función ***predict()***, pero la forma de llamarla, y la forma en la que devuelve las predicciones son muy diferentes entre paquetes

- La función ***predict()*** devuelve siempre un ***tibble***, con nombres de columnas reconocibles, y tantas **filas**, y en el **mismo orden,** como las del *data frame* donde se ha pedido que se hagan las predicciones (***new_data=***)

- Si se realizó algún **pre-procesamiento de datos** en el conjunto con el que se construyó el modelo, la función ***predict()*** lo aplica también al conjunto donde se hacen las predicciones

Predicciones con el paquete *parsnip*

- Los **tipos de predicción** se ponen en el parámetro ***type*=**
  - En modelos de **regresión**: ***"numeric",*** que proporciona el valor predicho de la variable respuesta continua
  - En modelos de **clasificación**: ***"class"*** y ***"prob"***, que proporcionan respectivamente, las clases y las probabilidades predichas de cada categoría, de la variable respuesta categórica

Modelos en el paquete *parsnip*

Modelo	Función	engine	Parámetros
Regresión Lineal	linear_reg	lm glmnet	penalty mixture
Regresión Logística	logistic_reg	glm glmnet	penalty mixture
Análisis Discriminante Lineal	discrim_linear	MASS	penalty regularization_method
Análisis Discriminante Cuadrático	discrim_quad	MASS	regularization_method
Partial Least Squares	pls	mixOmics	predictor_prop num_comp
Naive Bayes	naive_Bayes	klaR naivebayes	smoothness

Los modelos de ***parsnip*** están en: https://www.tidymodels.org/find/parsnip/

Modelos en el paquete *parsnip*

Modelo	Función	engine	Parámetros
K-NN	nearest_neighbor	kknn	neighbors weight_func dist_power
Red Neuronal	mlp	nnet h2o keras	hidden_units, penalty, dropout, epochs, activation, learn_rate
SVM Lineal	svm_linear	LiblineaR kernlab	cost margin
SVM Polinómico	svm_poly	kernlab	cost, margin degree, scale_factor
SVM Funciones Radial	svm_rbf	kernlab	cost , margin rbf_sigma

Modelos en el paquete *parsnip*

Modelo	Función	engine	Parámetros
Árboles de decisión	decision_tree	rpart C5.0 partykit	cost_complexity tree_depth min_n
Bagging	bag_tree	rpart C5.0	cost_complexity tree_depth, class_cost, min_n
Random Forest	rand_forest	ranger randomForest	mtry trees min_n
Boosting	boost_tree	xgboost	mtry, trees, min_n, tree_depth, learn_rate, loss_reduction, sample_size, stop_iter

Ejemplo: Regresión lineal con *linear_reg()*

```
> library(tidymodels)
> ## 1.- Fichero Datos: Solubility
> load("D://Solubility Data.RData")
>
> ## Regresión lineal con R base
> lm_out <- lm(Solubility ~ FP001 + FP004 + MolWeight + NumAtoms, data=xx_sob_train)
> lm_out
Coefficients:
(Intercept)         FP001         FP004     MolWeight      NumAtoms
 -0.5005724     0.1735312     1.3385897    -0.0153786    -0.0007695
> ## Regresión lineal con parsnip
> ## Especificación del modelo
> lm_model <-
+   linear_reg() %>%
+   set_engine("lm")
> lm_model
Linear Regression Model Specification (regression)

Computational engine: lm
```

- La función ***linear_reg()*** especifica el tipo de modelo de **regresión lineal** que se desea construir. En este caso, se desea usar la función **"*lm*"** del paquete ***stats*** (incluido en R base), lo que se especifica en el ***set_engine()***
- Son las **especificaciones del modelo**, sin referenciar ningún conjunto de datos
- No se especifica el ***set_mode()*** ya que la regresión lineal se usa solo para una variable respuesta continua. Se podría poner ***set_mode("regression")***

Ejemplo: *linear_reg()*

```
> ## Construcción del modelo usando fórmulas
> lm_form_fit <-
+   lm_model %>%
+   fit( Solubility ~ FP001 + FP004 + MolWeight + NumAtoms, data=xx_sob_train )
> lm_form_fit
parsnip model object

Call:
stats::lm(formula = Solubility ~ FP001 + FP004 + MolWeight +
    NumAtoms, data = data)

Coefficients:
(Intercept)        FP001        FP004    MolWeight     NumAtoms
 -0.5005724    0.1735312    1.3385897   -0.0153786   -0.0007695

> class(lm_form_fit)
[1] "_lm"       "model_fit"
```

- La función ***fit()*** permite construir el modelo con el objeto que contiene las especificaciones (regresión lineal con la función ***"lm"***), usando **una fórmula** y especificando también el *data frame*
- Se comprueba que se ha ajustado el mismo modelo de regresión lineal
- Es un objeto de clase ***model_fit***, que es distinto de un objeto de la clase *"lm"*, que es el que se obtuvo con la función *lm()*

Ejemplo: *linear_reg()*

```
> ## Construcción del modelo usando x / y
> lm_xy_fit <-
+   lm_model %>%
+   fit_xy(x = xx_sob_train %>% select(FP001, FP004, MolWeight, NumAtoms),
+          y = xx_sob_train %>% select(Solubility) )
> lm_xy_fit
parsnip model object

Call:
stats::lm(formula = ..y ~ ., data = data)

Coefficients:
(Intercept)        FP001        FP004    MolWeight     NumAtoms
 -0.5005724    0.1735312    1.3385897   -0.0153786   -0.0007695

> class(lm_xy_fit)
[1] "_lm"       "model_fit"
> class(lm_form_fit %>% extract_fit_engine())
[1] "lm"
```

- La función ***fit_xy()*** permite construir el modelo con el objeto que contiene las especificaciones (regresión lineal con la función ***"lm"***), usando la **interface X / Y**
- Se especifica en el parámetro ***x=*** los predictores, y en ***y=*** la variable respuesta, usando la funcion ***select()*** de ***tidyverse*** para seleccionar las columnas del *data frame*
- La función ***extract_fit_engine()*** permite acceder al objeto "lm", el que genera la función de R especificada en el *set_engine()*

Ejemplo: *linear_reg()*

```
> ## Coeficientes del modelo
> tidy(lm_form_fit)
# A tibble: 5 x 5
  term          estimate std.error statistic  p.value
  <chr>            <dbl>     <dbl>     <dbl>    <dbl>
1 (Intercept) -0.501     0.117       -4.27  2.21e- 5
2 FP001        0.174     0.177        0.981 3.27e- 1
3 FP004        1.34      0.182        7.34  4.74e-13
4 MolWeight   -0.0154    0.000708   -21.7   3.58e-84
5 NumAtoms    -0.000769  0.00576     -0.133 8.94e- 1
>
> ## Predicción en Testing
> predict(lm_form_fit, new_data=xx_sob_test)
# A tibble: 383 x 1
   .pred
   <dbl>
 1 -4.82
 2 -2.19
 3 -2.62
 4 -3.17
. . . .
```

- La función ***tidy()*** del paquete ***broom*** aplicada directamente al objeto ***model_fit***, permite obtener los coeficientes en un ***tibble*** con nombres de columnas más claros y estandarizados
- La función ***predict()*** sobre el objeto ***model_fit*** de ***parsnip*** devuelve las predicciones en un ***tibble*** en el mismo orden en el que están las observaciones en el *data frame*

Paquete *workflow*

- El paquete ***workflow*** permite **integrar** las principales **operaciones computacionales del proceso** de construcción de modelos predictivos
- La creación de ***workflows*** ayuda a organizar mejor los proyectos

- El paquete ***workflow*** permite unir en un **único objeto** las especificaciones del **pre-procesamiento,** del **modelado** y del **post-procesamiento**
  - Se entiende que “construir un modelo” no es solo el “ajuste matemático”, como sería por ejemplo, el cálculo de los coeficientes de un modelo de regresión

- Una de las principales importancias de los ***workflow*** es que se pueden usar cuando se están **optimizando los parámetros** de un modelo, mediante **técnicas de remuestreo**
  - Todas las tareas incluidas en el ***workflow*** se realizan, de forma independiente, en cada una de las **particiones** donde se están evaluando los parámetros

Paquete *workflow*

- El objeto ***workflow*** se crea con la función ***workflow()*** a la que se le van añadiendo especificaciones:
  - ***add_model()*** para añadir las especificaciones del modelo
  - ***add_formula()*** para añadir una fórmula
  - ***add_variables(outcome= , predictors=)*** para añadir la variable respuesta y las variables predictoras, en lugar de incluir una fórmula
  - ***add_recipe()*** para añadir tareas de pre-procesamiento, las recetas

- Hay funciones del tipo ***update_xxx()*** y ***remove_xxx(),*** que permiten cambiar el *workflow*, modificando o borrando algunos de sus componentes

Paquete *workflow*

- Normalmente, un ***workflow*** se construye con las especificaciones de un **modelo** y un ***recipe***, que incluye todas las tareas del **pre-procesamiento**

- La función ***fit(workflow, data)*** aplica, en primer lugar, el **pre-procesamiento** al *data frame* especificado, que es donde se realizan los cálculos que se necesitan para ese pre-procesamiento. A continuación, ajusta **el modelo** especificado en el *workflow* en ese *data frame* ya procesado

- La función ***predict(workflow, new_data)*** aplica al nuevo *data frame* el pre-procesamiento que fue definido anteriormente, sobre el *data frame* donde se construyó el modelo. Después, se calculan las **predicciones**

- También hay funciones de tipo ***extract_()*** para extraer los modelos o los *recipes* que contiene un ***workflow***

Paquete *yardstick*

- El paquete ***yardstick*** de ***tidymodels*** contiene el cálculo de las **medidas de capacidad predictiva**
  - Es un paquete útil, aunque no se use ***tidymodels*** para generar los modelos

- Todas las funciones se llaman igual: ***function(data, truth, …),*** donde ***data*** es un *data frame* que hay que crear previamente, que contiene los **valores observados**, que se indican en ***truth***, y también los **valores predichos**

- Se pueden calcular varias medidas a la vez creando un ***metric_set()***

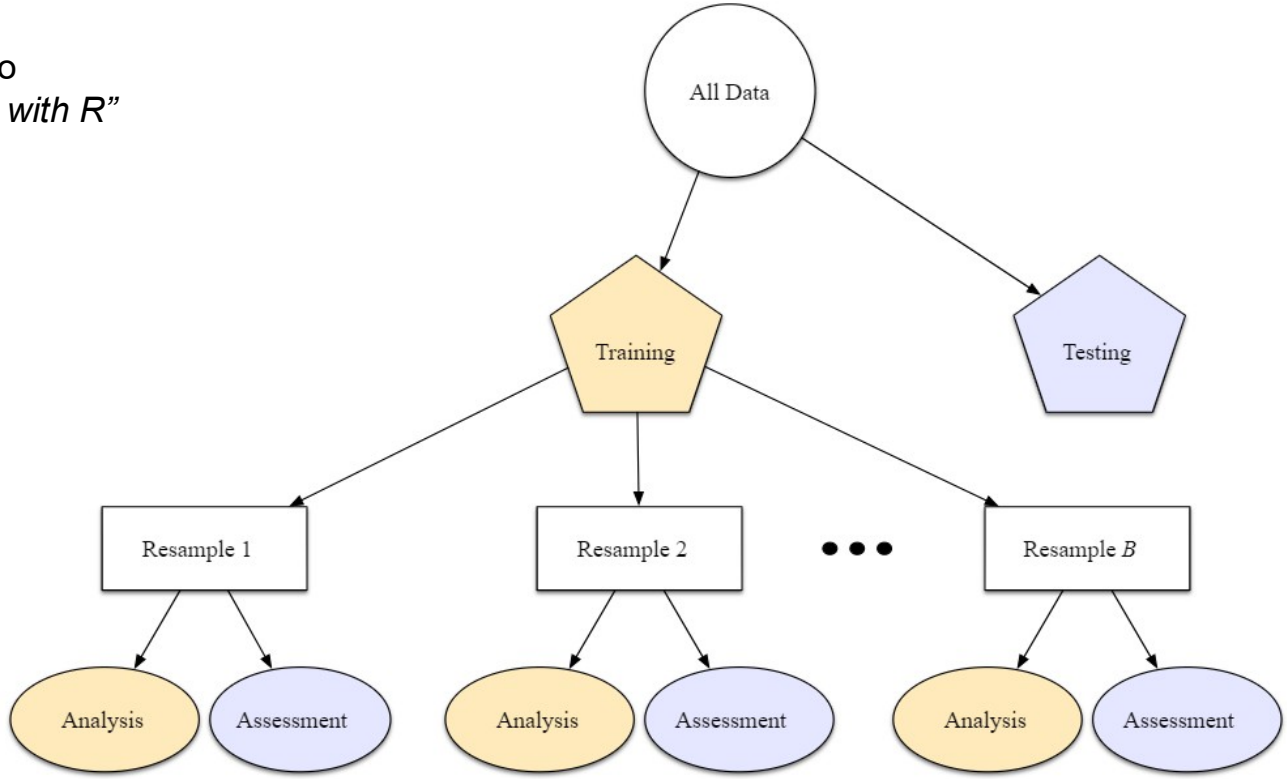
- Algunas de estas funciones para clasificación con una **variable respuesta binaria**, requieren que se indique la categoría con el **evento de interés**, que por defecto es la primera, ***event_level = "first"***
  - Hay que tener cuidado, porque la **regresión logística** codifica la respuesta como 0/1, siendo 1 el evento de interés que asigna al segundo nivel. En este caso, se debería poner el evento como segunda categoría, ***event_level = "second"***

Funciones del paquete *yardstick*

Función	Resp.	Descripción
accuracy()	Class	Calcula la tasa de precisión, accuracy
sens()	Class	Calcula la sensibilidad
spec()	Class	Calcula la especificidad
kap()	Class	Calcula el índice Kappa
roc_curve() ***roc_auc()***	Class	Calcula la curva ROC y el AUC, área bajo la curva ROC
brier_score()	Class	Brier score
mcc()	Class	Calcula el coeficiente de correlación de Mattews
f_means()	Class	Calcula la métrica F1
rmse()	Regr	Calcula la raíz cuadrada del error cuadrático medio
mae()	Regr	Calcula el error absoluto medio
rsq()	Regr	Calcula el coeficiente de determinación R2

Métodos de Estimación. Técnicas de remuestreo

Imagen del libro
"Tidy Modeling with R"



- Se obtienen **B submuestras**, y se repite el proceso B veces, construyendo **B modelos** en los **conjuntos de análisis,** y obteniendo **B medidas** de capacidad predictiva en los **conjuntos de evaluación**. Estas medidas son finalmente **promediadas**
- Todo este proceso se hace en la **muestra de training**

Paquete *rsample*

- El paquete ***rsample*** de ***tidymodels*** contiene las funciones de **creación de las muestras** necesarias para usar las **técnicas de remuestreo**
  - Es un paquete útil, aunque no se use ***tidymodels*** para generar los modelos

- Todas las funciones de remuestreo permiten incluir **todas las fases del proceso** (pre-procesamiento, optimización de parámetros, construcción, …), usando ***workflows***

- Todas las funciones de remuestreo contienen un parámetro que es ***strata=*** que permite realizar las particiones manteniendo las proporciones de las categorías de una variable categórica (**remuestreo estratificado**)
  - Si la variable por la que se desea estratificar es **continua**, se usan sus **cuartiles**

- La mayoría de los procesos de remuestreo permiten **paralelización**, ya que los modelos construidos son independientes unos de otros. ***tidymodels*** lo integra de una forma sencilla

Funciones del paquete *rsample*

Función	Descripción
initial_split()	Divide la muestra en dos submuestras, training y testing
training() ***testing()***	Construye los *data frames* conteniendo las muestras de training y testing, especificadas por *initial_split()*
vfold_cv()	Validación cruzada
loo_cv()	Validación cruzada leave-one-out
mc_cv()	Validación cruzada Monte Carlo (MCCV)
bootstraps()	Bootstrapping
analysis() ***assessment()***	Construye los *data frames* conteniendo los conjuntos de análisis y evaluación, dentro de las técnicas de remuestreo

- Estas funciones devuelven las **especificaciones** de los ***data frames*** de las particiones creadas, indicando qué observaciones pertenecen a cada una
- Con las funciones ***training/testing*** o ***analysis/assessment*** se crean los *data frames*, que se pueden salvar o integrar dentro de un ***workflow***

Muestras de training y testing

```
> library(tidymodels)
> library(tidyverse)
>
> ## Lectura del fichero con read_delim del paquete readr
> xx_sob <- read_delim("D://Solubility.csv", delim = ";")

Rows: 1267 Columns: 229
── Column specification ────────────────────────────────────────────────────────
Delimiter: ";"
dbl (229): FP001, FP002, FP003, FP004, FP005, FP006, FP007, FP008, FP009, FP010, FP...

> ## Especificaciones de las Muestras de Training y Testing
> set.seed(222) ## Se fija una semilla, para reproducir los datos
>
> sob_split <- initial_split(xx_sob, prop = 0.70, strata = Solubility)
> sob_split
<Training/Testing/Total>
<884/383/1267>
```

- Se usa la función ***read_delim()*** del paquete ***readr (tidyverse)*** para leer el fichero
- Se usa la función ***initial_split()*** del paquete ***rsample*** para crear las **especificaciones** de la partición, el 70% de las observaciones serán de training y el 30% de testing
- Se usa la variable respuesta continua como ***strata***, para que se repartan en las dos muestras, de una forma proporcional, los valores de la variable, usando los cuartiles

Muestras de training y testing

```
> ## Se crean las muestras de Training y Testing
> xx_sob_train <- training(sob_split)
> xx_sob_test <- testing(sob_split)
>
> dim(xx_sob_train)
[1] 884 229
> dim(xx_sob_test)
[1] 383 229
>
> quantile(xx_sob$Solubility)
     0%      25%      50%      75%     100%
-11.620  -3.955  -2.490  -1.360   1.580
> quantile(xx_sob_train$Solubility)
      0%       25%       50%       75%      100%
-11.6200  -3.9525  -2.4850  -1.3575   1.2200
> quantile(xx_sob_test$Solubility)
    0%    25%    50%    75%   100%
-9.030 -3.905 -2.510 -1.365  1.580
>
> ## Salva los data frame en un RData
> save( xx_sob_train, xx_sob_test, file = "D://Solubility Data.RData" )
```

- Las funciones ***training()*** y ***testing()*** generan las muestras, que son *data frames*
- La mediana y los cuartiles Q1 y Q3 de las muestras de training y testing son muy parecidos, porque se utilizó la opción ***strata = Solubility***
- Se salvan los dos *data frames* en un fichero **RData**

Paquetes *rsample* y *tune*

- La función ***fit_resamples()*** del paquete ***tune*** es la que **ajusta todos los modelos** en los diferentes subconjuntos de análisis, definidos en el objeto dónde se ha establecido el **tipo de remuestreo** (***resamples=****)*
  - Se le puede pasar un ***workflow***, una **fórmula** o un ***recipe***

- Todos los pasos definidos, se realizan en **cada conjunto de análisis,** de forma independiente. El **proceso** durante el **remuestreo** tiene 2 pasos, que se repiten en cada submuestra
  1. El **conjunto de análisis** es usado para los cálculos necesarios para el **pre-procesamiento** de datos. Se aplica en este mismo conjunto de análisis, y sobre el conjunto procesado, se **ajusta el modelo**
  2. El **pre-procesamiento** anterior, calculado sobre el conjunto de análisis, es aplicado al **conjunto de evaluación**, donde se obtienen las **predicciones**, con las que se calcula el **rendimiento predictivo**

Paquetes *rsample* y *tune*

- Hay un parámetro de control para elegir las métricas (***metrics***). Por defecto:
  - AUC y accuracy en clasificación
  - RMSE y R2 en regresión

- Hay una función ***control_resamples()*** dónde se puede indicar si se quieren salvar los objetos de **todos los modelos** construidos en el remuestreo (las métricas, las predicciones, los propios modelos, etc …)

- Para acceder a toda esta información, hay funciones como ***collect_metrics()*** y ***collect_predictions()***
  - Por defecto tienen un parámetro, ***summarize=TRUE***, que se usa para **promediar** las métricas o las predicciones
  - Si se usa ***summarize=FALSE*** recopila **todas las métricas o predicciones** de todos los modelos del remuestreo

Paquete *tune*. Paralelización

- Todos los **procesos de remuestreo** pueden ser **paralelizados**, ya que los cálculos y modelos en cada conjunto de análisis son **independientes**

- El paquete ***tune*** usa internamente el paquete ***foreach*** de R para la **computación en paralelo**

- Para determinar el **número de cores** se puede usar el paquete ***parallel***
- En Linux y macOS se pueden declarar también con el paquete ***doMC***

- Las funciones de remuestreo del paquete ***tune***, como ***fit_resamples()*** o ***tune_xxx()***, gestionan **automáticamente** la paralelización, una vez que los cores han sido declarados

Paquete *dials*

- El paquete ***dials*** contiene funciones que facilitan la creación de **conjuntos de parámetros** a explorar en los modelos

- A los parámetros que se desean optimizar se les asigna un valor ***=tune()*** en las especificaciones del modelo

- Los **parámetros** tienen nombres genéricos, y unificados, para todos los modelos que lo usan

- Cada parámetro tiene **una función** que indica la **escala** y el **rango** a explorar
  - Por ejemplo: *mixture(), penalty(), hidden_units(), learn_rate(), cost_complexity(), mtry(), trees(), min_n(), ….*
  - Se pueden cambiar estos rangos por defecto de cada parámetro, con las funciones ***extract_parameter_set_dials()*** y ***update()***

Paquete *dials*

- Las funciones ***grid_*()*** facilitan la creación de los conjuntos de parámetros a explorar en un proceso de optimización, con diferentes tipos de rejillas

- La función ***grid_regular()*** crea un número de valores para cada parámetro, con ***levels=*** y crea **todas las combinaciones** entre ellos

- Si se desea especificar los valores de los parámetros que se desean explorar, se puede usar la función ***crossing()*** del paquete ***tidyr***

- Una alternativa a usar ***grid*** regulares es usar una función como ***grid_random(size=)*** que selecciona combinaciones de valores aleatorios de cada parámetro, repartidos en el rango de cada uno de ellos

Paquete *dials*

- La función ***tune_grid()*** es la que ejecuta todos los modelos del *grid*. Su funcionamiento es similar a ***fit_resamples()*** con un parámetro adicional ***grid=*** que se puede rellenar de dos formas:
  - un *data frame* con todas las combinaciones, que se hayan elegido
  - un número entero, en cuyo caso, se crean automáticamente ese número de combinaciones, eligiendo aleatoriamente candidatos entre los parámetros a optimizar

- Las funciones ***tune_*()*** no ajustan el modelo final, con los parámetros óptimos. Para hacer esto, hay que usar las funciones ***finalize_workflow()*** y ***fit()*** donde se especifican los valores de los parámetros elegidos

Búsquedas iterativas

- La función ***tune_grid()*** usa un ***grid*** dónde se definen todas las combinaciones de parámetros para testear
- Otras alternativas establecen un proceso de **búsqueda iterativa**, donde se va prediciendo qué parámetros van a ser testeados en cada paso

- Las técnicas de **optimización bayesianas** crean un modelo predictivo donde los parámetros son los predictores y la medida de la capacidad predictiva es la respuesta. Se inicializa con unas pocas combinaciones, se crea un modelo y se predice la capacidad de otras combinaciones, de las que se selecciona la mejor, que se van incorporando al proceso iterativo
  - En ***tidymodels*** se puede usar con la función ***tune_bayes()***

- Las técnicas de **alineamiento simulado** parten de una combinación inicial que van cambiado levemente en cada paso, sustituyendo cada vez por la mejor
  - En ***tidymodels*** se puede usar con la función ***tune_sim_anneal()***

Construcción de un modelo con *tidymodels*

- La **construcción de un modelo predictivo** con ***tidymodels*** se puede estructurar en los siguientes pasos:
  - Carga de las librerías y lectura de los datos. Training y Testing
  - Especificaciones del **pre-procesamiento**
  - Especificaciones del **modelo** a ajustar, con los parámetros a optimizar
  - Integrar el pre-procesamiento y el modelo en un ***workflow***
  - Especificaciones de la **técnica de remuestreo**
  - Especificaciones del conjunto de **parámetros** a explorar
  - Ejecución de la función de **optimización de parámetros**
  - Se finaliza el *workflow* y se ajusta el **modelo final** con los **parámetros óptimos**
  - Se evalúa el modelo en la **muestra de testing**: obtención de las **predicciones** y cálculo de las **medidas de capacidad predictiva**

Paralelización

```
> library(tidymodels)
> library(tidyverse)
> ## Se cargan los datos, muestras de training y testing
> load("D://Solubility Data.RData")
>
> ## Paralelización
> num_cores <- parallel::detectCores()
> num_cores
[1] 12
>
> if (!grepl("mingw32", R.Version()$platform)) {
+    library(doMC)
+    registerDoMC(cores = num_cores - 1)
+ } else {
+    library(doParallel)
+    cl <- makePSOCKcluster(num_cores - 1)
+    registerDoParallel(cl)
+ }
Cargando paquete requerido: foreach

.  .  .
Cargando paquete requerido: iterators
Cargando paquete requerido: parallel
```

- Puesto que se incluye un paso de optimización de parámetros, se **paraleliza** con varios cores. Al declarar el número de cores, se cargan las distintas librerías de paralelización que son necesarias
- Al final del script, hay que liberar los cores con ***stopCluster(cl)***

Especificaciones del modelo

```
> ## 1.- Especificaciones del recipe
> normalized_rec <-
+    recipe(Solubility ~ ., data = xx_sob_train) %>%
+    step_normalize(all_numeric_predictors())
>
> ## 2.- Especificaciones modelo lineal con elastic net
> enet_spec <-
+    linear_reg(penalty = tune(), mixture = tune()) %>%
+    set_engine("glmnet") %>%
+    set_mode("regression")
>
```

- Se crea un ***recipe*** con el **pre-procesamiento**, con un paso de **normalización** de todos los predictores numéricos. En la fórmula se está indicando que **Solubility** es la **variable respuesta**, y el resto de variables que hay en la muestra de training, son **predictores**

- Se especifica el **modelo** con la función ***linear_reg()*** y se indica que se desea modelar una **regresión lineal penalizada** con el paquete ***glmnet*** *(set_engine=)*, donde se van a optimizar (=***tune()***) los parámetros ***penalty,*** que se refiere a *lambda*, y ***mixture*** que se refiere al *alpha* de elastic net (compromiso entre ridge y lasso)
- Con ***set_mode("regression")*** se indica que la variable respuesta es continua

Especificaciones del modelo

```
> penalty()
Amount of Regularization (quantitative)
Transformer: log-10 [1e-100, Inf]
Range (transformed scale): [-10, 0]
> mixture()
Proportion of Lasso Penalty (quantitative)
Range: [0, 1]
>
```

- Con las funciones ***penalty()*** y ***mixture()*** se pide información sobre estos parámetros, incluidos sus rangos y escala
- Lasso es *mixture = 1*

Especificaciones del modelo

```
> ## 3.- Se crea el workflow
> wflow <- workflow() %>%
+   add_model(enet_spec) %>%
+   add_recipe(normalized_rec)
>
> wflow
══ Workflow ════════════════════════════════════════════════════════════════════
Preprocessor: Recipe
Model: linear_reg()

── Preprocessor ────────────────────────────────────────────────────────────────
1 Recipe Step

• step_normalize()

── Model ───────────────────────────────────────────────────────────────────────
Linear Regression Model Specification (regression)

Main Arguments:
  penalty = tune()
  mixture = tune()

Computational engine: glmnet
```

- Se crea el ***workflow***, integrando las especificaciones del pre-procesamiento y el modelo

Especificaciones del modelo

```
> ## 4.- Especificaciones de la técnica de remuestreo
> cv_split <- vfold_cv(xx_sob_train, strata = Solubility, v = 10, repeats = 10)
>
> cv_split
#  10-fold cross-validation repeated 10 times using stratification
# A tibble: 100 × 3
   splits             id       id2
   <list>             <chr>    <chr>
 1 <split [792/92]> Repeat01 Fold01
 2 <split [796/88]> Repeat01 Fold02
 3 <split [796/88]> Repeat01 Fold03
 4 <split [796/88]> Repeat01 Fold04
 5 <split [796/88]> Repeat01 Fold05
 6 <split [796/88]> Repeat01 Fold06
 7 <split [796/88]> Repeat01 Fold07
 8 <split [796/88]> Repeat01 Fold08
 9 <split [796/88]> Repeat01 Fold09
10 <split [796/88]> Repeat01 Fold10
# ℹ 90 more rows
```

- Se especifica la **técnica de remuestreo** con la función ***vfold_cv()*** del paquete ***rsample*** que es 10 times 10-fold CV (***v=10*** por defecto), **estratificado** por la variable respuesta
- Las **100 particiones** están ya preparadas, indicando en cada una, qué observaciones forman parte de la **muestra de análisis** y cuáles de la **muestra de evaluación**

Especificaciones del modelo

```
> ## 5.- Se crea un grid
> enet_grid <- grid_regular(penalty(), mixture(),
+                           levels = list(penalty = 100, mixture = 11) )
> enet_grid
# A tibble: 1,100 x 2
    penalty mixture
<dbl>   <dbl>
 1 1   e-10       0
 2 1.26e-10       0
 3 1.59e-10       0
 4 2.01e-10       0
 5 2.54e-10       0
 6 3.20e-10       0
 7 4.04e-10       0
 8 5.09e-10       0
 9 6.43e-10       0
10 8.11e-10       0
# i 1,090 more rows
```

- Con la función ***grid_regular()*** del paquete ***dials*** se crea el **grid** de parámetros a explorar con todas las combinaciones de 100 lambdas y 11 parámetros de mixture (0, 0.1, 0.2, …., 0.9, 1). En total, son 1100 combinaciones

Optimización de parámetros

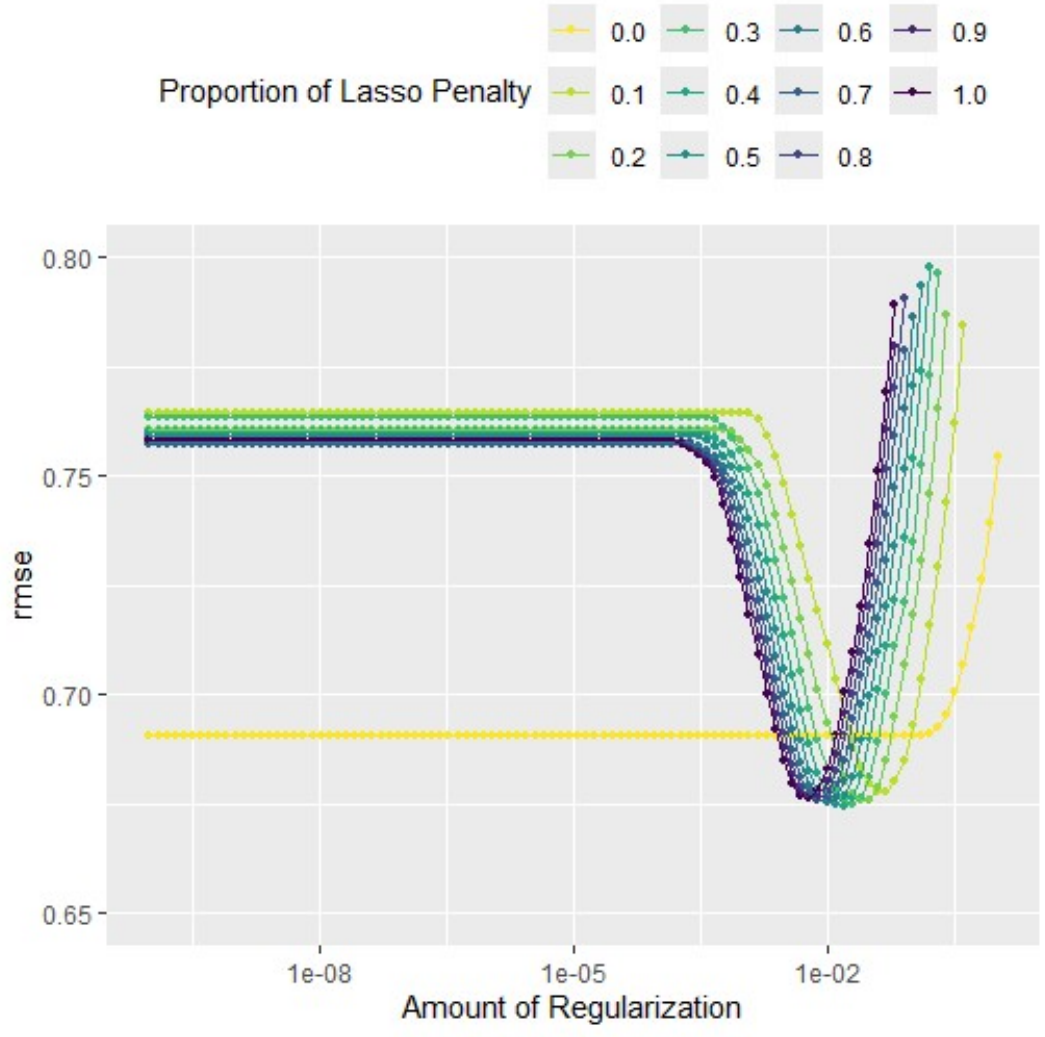
```
> ## Se ejecuta la optimización de parámetros
> keep_pred <- control_grid(save_pred = TRUE) ## salva las predicciones
>
> tune_result <- wflow %>%
+   tune_grid(cv_split, grid = enet_grid, control = keep_pred,
+             metrics = metric_set(rmse, rsq))
>
```

- Se ejecuta la **optimización de parámetros** con la función ***tune_grid()*** del paquete ***tune*** especificando el *workflow*, el remuestreo, el grid y las métricas a evaluar, el RMSE y el R2
- El pre-procesamiento y el modelo ya estaban integrados en el *workflow*
- Además, usando la función ***control_grid(),*** se indica también que se desean salvar **las predicciones** en todas las particiones de evaluación, de todo el remuestreo

- El **pre-procesamiento** de los datos, estandarización de las variables en este caso, se hace en cada **muestra de análisis** de la validación cruzada
- Es decir, en cada **partición** de la validación cruzada, se hacen los **cálculos** necesarios (medias y desviaciones estándar de las variables) en **la muestra de análisis**, y se **aplica** a las **muestras de análisis y de evaluación** de esa partición

Optimización de parámetros

```
> ## Plot
> autoplot(tune_result, metric = "rmse") +
+   scale_color_viridis_d(direction = -1) +
+   ylim(0.65, 0.80) +
+   theme(legend.position = "top")
```



Optimización de parámetros

```
> ## Se analizan los resultados
> tune_result %>%
+   collect_metrics()
# A tibble: 2,200 × 8
    penalty mixture .metric .estimator  mean     n std_err .config
      <dbl>   <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
 1 1   e-10       0 rmse    standard   0.689   100 0.00657 Preprocessor1_Model0001
 2 1   e-10       0 rsq     standard   0.889   100 0.00252 Preprocessor1_Model0001
 3 1.26e-10       0 rmse    standard   0.689   100 0.00657 Preprocessor1_Model0002
 4 1.26e-10       0 rsq     standard   0.889   100 0.00252 Preprocessor1_Model0002
 5 1.59e-10       0 rmse    standard   0.689   100 0.00657 Preprocessor1_Model0003
 6 1.59e-10       0 rsq     standard   0.889   100 0.00252 Preprocessor1_Model0003
 7 2.01e-10       0 rmse    standard   0.689   100 0.00657 Preprocessor1_Model0004
 8 2.01e-10       0 rsq     standard   0.889   100 0.00252 Preprocessor1_Model0004
 9 2.54e-10       0 rmse    standard   0.689   100 0.00657 Preprocessor1_Model0005
10 2.54e-10       0 rsq     standard   0.889   100 0.00252 Preprocessor1_Model0005
```

- Con la función ***collect_metrics()*** se muestran los **resultados** de la optimización de parámetros
- Para cada una de las combinaciones de **penalty** y **mixture**, se han evaluado 100 particiones de la CV (10 times 10-fold CV), y se muestran la **medias** y **errores estándar** del **RMSE** y del R2 de las 100 medidas obtenidas

Optimización de parámetros

```
> ## Los mejores modelos
> show_best(tune_result, metric="rmse")
# A tibble: 5 × 8
  penalty mixture .metric .estimator  mean     n std_err .config
    <dbl>   <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
1 0.0152      0.4 rmse    standard   0.674   100 0.00514 Preprocessor1_Model0482
2 0.0120      0.4 rmse    standard   0.675   100 0.00523 Preprocessor1_Model0481
3 0.0120      0.5 rmse    standard   0.675   100 0.00516 Preprocessor1_Model0581
4 0.0192      0.3 rmse    standard   0.675   100 0.00515 Preprocessor1_Model0383
5 0.00955     0.6 rmse    standard   0.675   100 0.00521 Preprocessor1_Model0680
> ## El mejor modelo
> select_best(tune_result, metric="rmse")
# A tibble: 1 × 3
  penalty mixture .config
    <dbl>   <dbl> <chr>
1  0.0152     0.4 Preprocessor1_Model0482
> ## Parámetros del modelo con mínimo RMSE
> tune_best <- tune_result %>% select_best(metric = "rmse")
> tune_best$penalty
[1] 0.01519911
> tune_best$mixture
[1] 0.4
```

- La función ***show_best()*** muestra los **mejores modelos**, y la función ***select_best()*** selecciona los parámetros del mejor modelo. Se especifica la ***metric="rmse"***
- El **mejor modelo** tiene un **RMSE** de **0.674** (con un error estándar de 0.00514), y corresponde al modelo de **elastic net** con **alpha=0.4** y **lambda = 0.0152**

Modelo final

```
> ## Se crea al workflow final
> final_wflow <-
+   wflow %>%
+   finalize_workflow( select_best(tune_result, metric="rmse") )
> ## Se crea al model final
> enet_fit <-
+   final_wflow %>%
+   fit(xx_sob_train)
> enet_fit
    Df  %Dev Lambda
1    0  0.00 3.3450
2    1  3.83 3.0480
3    2  7.53 2.7770
4    3 12.44 2.5300
5    8 18.99 2.3050
.   .   .   .
```

- La función ***tune_grid()*** no construye el modelo final con los parámetros óptimos, y hay que crear un nuevo ***workflow*** con el **modelo final**, seleccionando los **parámetros óptimos** por la **métrica** deseada, lo que se hace con la función ***finalize_workflow()***
- El modelo especificado en este *workflow*, se ajusta con la función ***fit()*** detallando el *data frame* sobre el que se desea hacer
- El modelo final proporcionado realmente se ajusta con varios lambdas (***glmnet*** lo hace siempre así, por defecto), pero hay un **control** del parámetro de penalización **óptimo**

Modelo final

```
> ## Coeficientes del modelo final
> tidy(enet_fit) %>% print(n=4)
# A tibble: 229 × 3
  term        estimate penalty
  <chr>          <dbl>   <dbl>
1 (Intercept)  -2.76    0.0152
2 FP001         0       0.0152
3 FP002         0.106   0.0152
4 FP003        -0.0155  0.0152
# ℹ 225 more rows
> tidy(enet_fit) %>% filter( estimate != 0 ) %>% print(n=4)
# A tibble: 138 × 3
  term        estimate penalty
  <chr>          <dbl>   <dbl>
1 (Intercept)  -2.76    0.0152
2 FP002         0.106   0.0152
3 FP003        -0.0155  0.0152
4 FP004        -0.0439  0.0152
# ℹ 134 more rows
> ## Modelo final "glmnet"
> out_glmnet <- extract_fit_engine(enet_fit)
> class(out_glmnet)
[1] "coxnet" "glmnet"
```

- Con la función ***tidy()*** se muestran los **coeficientes del modelo** (con penalty = 0.0152), pero hay que seleccionar los de las **138 variables** con coeficientes distintos de 0
- Con la función ***extract_fit_engine()*** se puede extraer el objeto del paquete ***glmnet***

Evaluación en la muestra de testing

```
> ## Predicciones y evaluación en Testing
> pred_enet_df <- predict(enet_fit, new_data = xx_sob_test)
> pred_enet_df %>% print(n=4)
# A tibble: 383 × 1
  .pred
  <dbl>
1 -4.24
2 -3.56
3 -3.62
4 -3.45
>
> pred_enet_df <- bind_cols(pred_enet_df, xx_sob_test %>% select(Solubility))
> pred_enet_df %>% print(n=4)
# A tibble: 383 × 2
  .pred Solubility
  <dbl>      <dbl>
1 -4.24      -3.98
2 -3.56      -3.99
3 -3.62      -4
4 -3.45      -4.08
```

- Se usa la función ***predict()*** de ***tidymodels*** para obtener las **predicciones** en la **muestra de testing**. Esta función aplica el **pre-procesamiento** especificado
- Las predicciones están en el orden en el que estaban en el *data frame*
- Se crea un *data frame* con las predicciones y los valores observados, para poder usar las funciones del paquete ***dials*** de ***tidymodels***

Evaluación en la muestra de testing

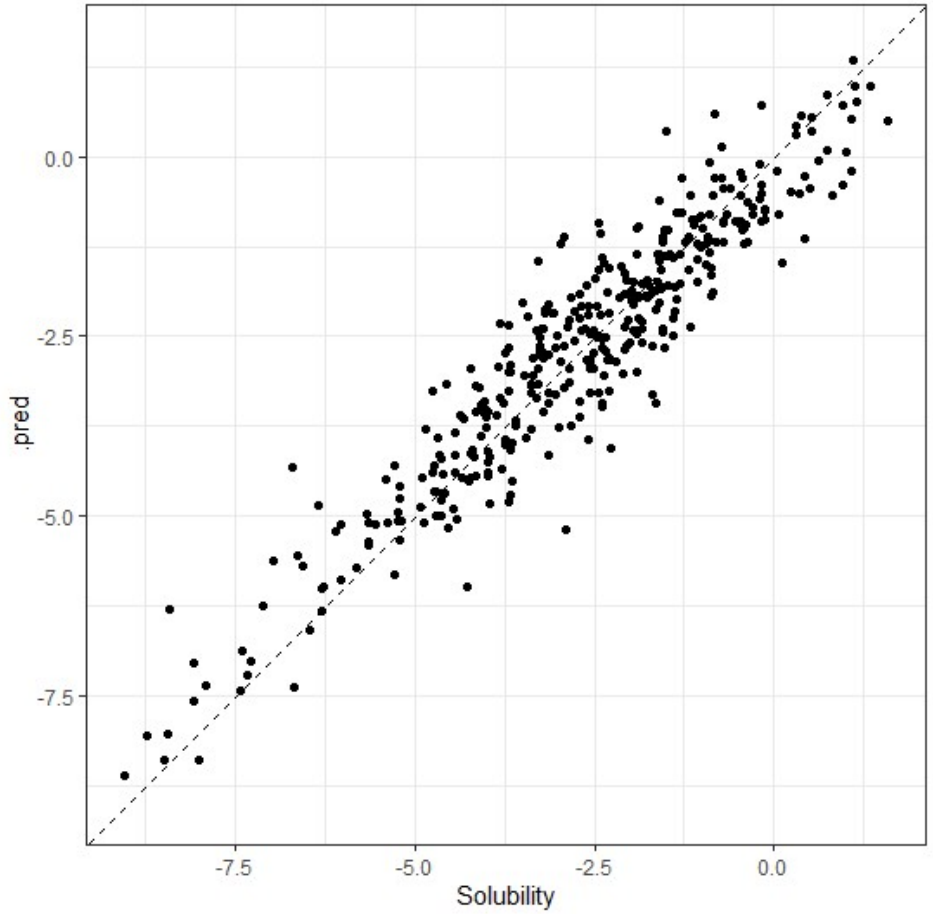
```
> rmse(pred_enet_df, truth = Solubility, estimate = .pred )
# A tibble: 1 × 3
  .metric .estimator .estimate
1 rmse    standard       0.686
>
> rsq(pred_enet_df, truth = Solubility, estimate = .pred )
# A tibble: 1 × 3
  .metric .estimator .estimate
1 rsq     standard       0.883

> all_metrics <- metric_set(rmse, rsq, mae)
> all_metrics(pred_enet_df, truth = Solubility, estimate = .pred )
# A tibble: 3 × 3
  .metric .estimator .estimate
  <chr>   <chr>          <dbl>
1 rmse    standard       0.686
2 rsq     standard       0.883
3 mae     standard       0.536
```

- El **RMSE** en la muestra de testing es **0.686** y el **R2** es **0.883**
- Se crea un ***metric_set()*** con las tres medidas, **RMSE, R2 y MAE**

Evaluación en la muestra de testing

```
> ## Gráfico de diagnóstico, en la muestra de testing
> dev.new()
> ggplot( pred_enet_df, aes(x = Solubility, y = .pred)) +
+   geom_abline(lty = 2) +
+   geom_point() +
+   coord_obs_pred() +
+   theme_bw()
```



Cross-validated predictions

- En la **optimización de parámetros**, con **validación cruzada**, es posible almacenar **todas las predicciones** que se han realizado en las **muestras de evaluación,** con los modelos construidos en las muestras de análisis
- Para **cada observación de training**, dónde se ha ejecutado la optimización de parámetros, hay **una predicción.** Se llaman **cross-validated predictions**
  - Si la validación cruzada se ha **repetido varias veces**, se calcula **la media** de las predicciones, para cada observación

- Estas **predicciones de la validación cruzada** pueden ser usadas de forma semejante a las predicciones de una muestra de testing. Se pueden obtener **medidas de capacidad predictiva** y los **gráficos** derivados
  - Clasificación: AUC, accuracy, curvas ROC, gráficos de calibración, …
  - Regresión: RMSE, R2, gráficos de diagnóstico, …

- Esto tiene especial interés en **muestras pequeñas**, donde no se ha podido hacer una partición inicial, y por tanto, **no hay muestra de testing**

Cross-validated predictions

```
> ## Extraer las Cross-Validated Predictions
> assess_res <- collect_predictions(tune_result)
> assess_res %>% print(n=4)
# A tibble: 9,724,000 × 8
  .pred id       id2      .row      penalty mixture Solubility .config
1 -3.81 Repeat01 Fold01     2 0.0000000001       0      -4.06 Preprocessor1_Model0001
2 -5.43 Repeat01 Fold01     3 0.0000000001       0      -4.08 Preprocessor1_Model0001
3 -4.41 Repeat01 Fold01     8 0.0000000001       0      -4.14 Preprocessor1_Model0001
4 -4.36 Repeat01 Fold01    36 0.0000000001       0      -4.46 Preprocessor1_Model0001
> 884 * 1100 * 10    ## 884 obs. en training, 1100 parámetros, 10 repeticiones de CV
[1] 9724000
> ## Predicciones con summarize ( 1 valor por observación )
> assess_res_summ <- collect_predictions(tune_result, summarize=TRUE)
> assess_res_summ %>% print(n=4)
# A tibble: 972,400 × 6
  .pred  .row      penalty mixture Solubility .config
1 -3.90     1 0.0000000001     0        -3.97 Preprocessor1_Model0001
2 -3.89     1 0.0000000001     0.1      -3.97 Preprocessor1_Model0101
3 -3.89     1 0.0000000001     0.2      -3.97 Preprocessor1_Model0201
4 -3.89     1 0.0000000001     0.3      -3.97 Preprocessor1_Model0301
```

- La función ***collect_predictions()*** permite recuperar todas las predicciones del proceso de la validación cruzada. El parámetro es el objeto de la optimización de parámetros, que se hizo con la función ***tune_grid()*** con la opción ***save_pred = TRUE***
- Si esta función se usa con el parámetro ***summarize=TRUE*** calcula las medias de las 10 predicciones por observación (10 repeats), proporcionando **un valor por observación**

Cross-validated predictions

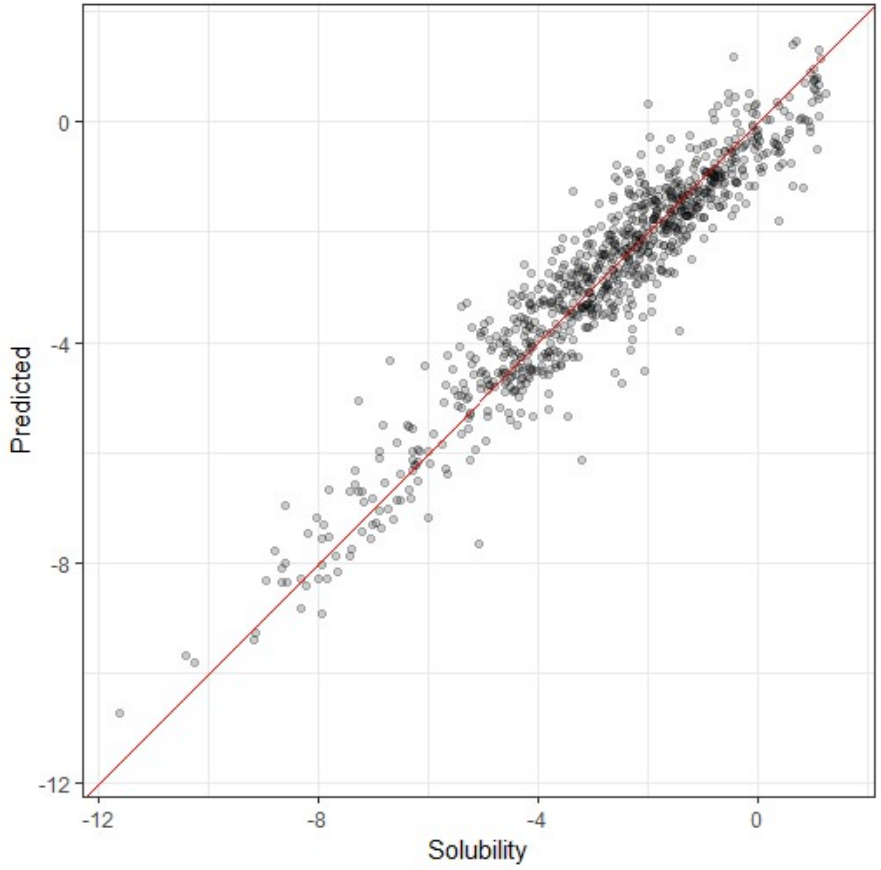
```
> ## Cross-validated predictions del modelo con parámetros óptimos
> assess_res_summ_best <-
+   assess_res_summ %>%
+   filter( penalty == tune_best$penalty,
+           mixture == tune_best$mixture )
> assess_res_summ_best %>% print(n=4)
# A tibble: 884 × 6
  .pred  .row penalty mixture Solubility .config
  <dbl> <int>   <dbl>   <dbl>      <dbl> <chr>
1 -3.83     1  0.0120     0.4      -3.97 Preprocessor1_Model0481
2 -3.62     2  0.0120     0.4      -4.06 Preprocessor1_Model0481
3 -5.36     3  0.0120     0.4      -4.08 Preprocessor1_Model0481
4 -4.49     4  0.0120     0.4      -4.1  Preprocessor1_Model0481
>
> ## RMSE
> rmse(assess_res_summ_best, truth = Solubility, .pred )
1 rmse    standard       0.673
```

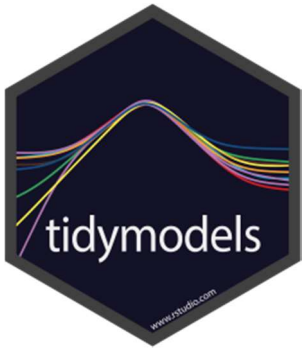
- Se seleccionan las filas de los **parámetros óptimos**, y ahora se tiene **una única predicción por observación,** del proceso de validación cruzada que se hizo en la **muestra de training**, que son a las que se les llama **Cross-validated predictions** del modelo con parámetros óptimos
- Se calcula el **RMSE** con esta predicción, que es **0.673**, muy parecido al que se obtuvo en la optimización de parámetros, 0.678, pero no tienen por qué ser iguales, ya que este cálculo se obtiene de la **media de las predicciones**, y el primero era una media de los 100 RMSEs, obtenidos en las 100 particiones del proceso de optimización

Cross-validated predictions

```
> ## Gráfico de diagnóstico
> dev.new()
> assess_res_summ_best %>%
+   ggplot(aes(x = Solubility, y = .pred)) +
+   geom_point(alpha=0.2) +
+   geom_abline(color="red") +
+   coord_obs_pred() +
+   ylab("Predicted") +
+   theme_bw()
```

- De este proceso lo más interesante es que se puede mostrar **el gráfico de diagnóstico** entre los valores observados y los valores predichos, obtenidos en el proceso de la validación cruzada





GRACIAS !!!!