

Tabla de contenidos

Scikit-learn
Introducción
Librerías
Datos
▼ Análisis exploratorio
Tipo de cada columna
Número de observaciones y v
Variable respuesta
Variables numéricas
Correlación variables numéric
Variables cualitativas
División train y test
▼ Preprocesado
Imputación de valores ausent
Exclusión de variables con va
Estandarización y escalado d
Binarización de las variables c
Pipeline y ColumnTransformer
▼ Crear un modelo
Entrenamiento
Validación
Predicción
Error de test
▼ Hiperparámetros (tuning)
Grid search
Random grid search
Optimización bayesiana
Tuning del preprocesado
▼ Algoritmos
K-Nearest Neighbor (kNN)
Regresión lineal (Ridge y Las
Random Forest
Gradient Boosting Trees
▼ Stacking
Algoritmo Super Learner
Comparación
▼ Anexos
Anexo1: Métodos de validació
Anexo2: Métricas
Bibliografía

Machine learning con Python y Scikit-learn

Joaquín Amat Rodrigo

Agosto, 2020 (actualizado Octubre 2020)

Más sobre ciencia de datos: cienciadedatos.net (<https://cienciadedatos.net>)

- [Machine learning con H2O y Python](https://www.cienciadedatos.net/documentos/py04_machine_learning_con_h2o_y_python) (https://www.cienciadedatos.net/documentos/py04_machine_learning_con_h2o_y_python)
- [Random Forest con Python](https://www.cienciadedatos.net/documentos/py08_random_forest_python.html) (https://www.cienciadedatos.net/documentos/py08_random_forest_python.html)
- [Gradient Boosting con Python](https://www.cienciadedatos.net/documentos/py09_gradient_boosting_python.html) (https://www.cienciadedatos.net/documentos/py09_gradient_boosting_python.html)

Scikit-learn

Python es uno de los lenguajes de programación que domina dentro del ámbito de la estadística, *data mining* y *machine learning*. Al tratarse de un software libre, innumerables usuarios han podido implementar sus algoritmos dando lugar a un número muy elevado de librerías donde encontrar prácticamente todas las técnicas de *machine learning* existentes. Sin embargo, esto tiene un lado negativo, cada paquete tiene una sintaxis, estructura y implementación propia, lo que dificulta su aprendizaje. [Scikit-learn](https://scikit-learn.org/stable/index.html) (<https://scikit-learn.org/stable/index.html>), es una librería de código abierto que unifica bajo un único marco los principales algoritmos y funciones, facilitando en gran medida todas las etapas de preprocesado, entrenamiento, optimización y validación de modelos predictivos.

Scikit-learn ofrece tal cantidad de posibilidades que, difícilmente, pueden ser mostradas con un único ejemplo. En este documento, se emplean solo algunas de sus funcionalidades. Si en algún caso se requiere una explicación detallada, para que no interfiera con la narrativa del análisis, se añadirá un anexo. Aun así, para conocer bien todas las funcionalidades de Scikit-learn se recomienda leer su documentación.

Una de las características a destacar de esta librería es su elevado grado de madurez, lo que la hace adecuada para crear modelos predictivos que se quieren poner en producción.

Este documento está reproducido también con [tidymodels](https://www.cienciadedatos.net/documentos/59_machine_learning_con_r_y_tidymodels) (https://www.cienciadedatos.net/documentos/59_machine_learning_con_r_y_tidymodels), [mlr3](https://www.cienciadedatos.net/documentos/60_machine_learning_con_r_y_mlr3) (https://www.cienciadedatos.net/documentos/60_machine_learning_con_r_y_mlr3). Otros proyectos similares son [caret](https://www.cienciadedatos.net/documentos/41_machine_learning_con_r_y_caret) (https://www.cienciadedatos.net/documentos/41_machine_learning_con_r_y_caret) y [H2O](https://www.cienciadedatos.net/documentos/44_machine_learning_con_h2o_y_r) (https://www.cienciadedatos.net/documentos/44_machine_learning_con_h2o_y_r), todos ellos basados en el lenguaje de programación R.

Tabla de contenidos ↗

Scikit-learn
Introducción
Librerías
Datos
▼ Análisis exploratorio
Tipo de cada columna
Número de observaciones y v
Variable respuesta
Variables numéricas
Correlación variables numéric
Variables cualitativas
División train y test
▼ Preprocesado
Imputación de valores ausent
Exclusión de variables con va
Estandarización y escalado d
Binarización de las variables
Pipeline y ColumnTransformer
▼ Crear un modelo
Entrenamiento
Validación
Predicción
Error de test
▼ Hiperparámetros (tuning)
Grid search
Random grid search
Optimización bayesiana
Tuning del preprocesado
▼ Algoritmos
K-Nearest Neighbor (kNN)
Regresión lineal (Ridge y Las
Random Forest
Gradient Boosting Trees
▼ Stacking
Algoritmo Super Learner
Comparación
▼ Anexos
Anexo1: Métodos de validaci
Anexo2: Métricas
Bibliografía

Introducción

Durante los últimos años, el interés y la aplicación de *machine learning* ha experimentado tal expansión que se convertido en una disciplina aplicada en prácticamente todos los ámbitos de investigación académica e industria creciente número de personas dedicadas a esta disciplina ha dado como resultado todo un repertorio herramientas con las que acceder a métodos predictivos potentes. El lenguaje de programación **Python** es ejemplo de ello.

El término *machine learning* engloba al conjunto de algoritmos que permiten identificar patrones presentes en datos y crear con ellos estructuras (modelos) que los representan. Una vez que los modelos han sido generados pueden emplear para predecir información sobre hechos o eventos que todavía no se han observado. Es importante recordar que, los sistemas de *machine learning*, "aprender" patrones que estén presentes en los datos con los que entran, por lo tanto, solo pueden reconocer escenarios similares a lo que han visto antes. Al emplear sistemas entrenados con datos pasados para predecir futuros se está asumiendo que, en el futuro, el comportamiento será similar, cosa que no siempre ocurre.

Aunque con frecuencia, términos como *machine learning*, *data mining*, inteligencia artificial, *data science*... utilizados como sinónimos, es importante destacar que los métodos de *machine learning* son solo una parte de muchas estrategias que se necesita combinar para extraer, entender y dar valor a los datos. El siguiente documento pretende ser un ejemplo del tipo de problema al que se suele enfrentar un analista: partiendo de un conjunto de datos más o menos procesado (la preparación de los datos es una etapa clave que precede al *machine learning*), se debe crear un modelo que permita predecir con éxito el comportamiento o valor que toman nuevas observaciones.

Etapas de un problema de *machine learning*

- Definir el problema: ¿Qué se pretende predecir? ¿De qué datos se dispone? o ¿Qué datos es necesario conseguir?
- Explorar y entender los datos que se van a emplear para crear el modelo.
- Métrica de éxito: definir una forma apropiada de cuantificar cómo de buenos son los resultados obtenidos.
- Preparar la estrategia para evaluar el modelo: separar las observaciones en un conjunto de entrenamiento, un conjunto de validación (o validación cruzada) y un conjunto de test. Es muy importante asegurar que ninguna información del conjunto de test participa en el proceso de entrenamiento del modelo.
- Preprocesar los datos: aplicar las transformaciones necesarias para que los datos puedan ser interpretados por el algoritmo de *machine learning* seleccionado.
- Ajustar un primer modelo capaz de superar unos resultados mínimos. Por ejemplo, en problemas de clasificación el mínimo a superar es el porcentaje de la clase mayoritaria (la moda). En un modelo de regresión, la media de variable respuesta.
- Gradualmente, mejorar el modelo incorporando-creando nuevas variables u optimizando los hiperparámetros.
- Evaluar la capacidad del modelo final con el conjunto de test para tener una estimación de la capacidad que tiene el modelo cuando predice nuevas observaciones.
- Entrenar el modelo final con todos los datos disponibles.

A diferencia de otros [documentos](https://www.cienciadedatos.net) (<https://www.cienciadedatos.net>), este pretende ser un ejemplo práctico con menos desarrollo teórico. El lector podrá darse cuenta de lo sencillo que es aplicar un gran abanico de métodos predictivos con **python** y **Scikit-learn**. Sin embargo, es crucial que cualquier analista entienda los fundamentos teóricos en los que se basa cada uno de ellos para que un proyecto de este tipo tenga éxito. Aunque aquí solo se describen brevemente, estarán acompañados de links donde encontrar información detallada.

Tabla de contenidos ↴

Scikit-learn
Introducción
Librerías
Datos
▼ Análisis exploratorio
Tipo de cada columna
Número de observaciones y v
Variable respuesta
Variables numéricas
Correlación variables numéric
Variables cualitativas
División train y test
▼ Preprocesado
Imputación de valores ausent
Exclusión de variables con va
Estandarización y escalado d
Binarización de las variables c
Pipeline y ColumnTransformer
▼ Crear un modelo
Entrenamiento
Validación
Predicción
Error de test
▼ Hiperparámetros (tuning)
Grid search
Random grid search
Optimización bayesiana
Tuning del preprocesado
▼ Algoritmos
K-Nearest Neighbor (kNN)
Regresión lineal (Ridge y Las
Random Forest
Gradient Boosting Trees
▼ Stacking
Algoritmo Super Learner
Comparación
▼ Anexos
Anexo1: Métodos de validació
Anexo2: Métricas
Bibliografía

Librerías

Las librerías utilizadas en este documento son:

```
In [1]: # Tratamiento de datos
# =====
import numpy as np
import pandas as pd
from tabulate import tabulate

# Gráficos
# =====
import matplotlib.pyplot as plt
from matplotlib import style
import matplotlib.ticker as ticker
import seaborn as sns

# Preprocesado y modelado
# =====
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RepeatedKFold
from sklearn.neighbors import KNeighborsClassifier
from sklearn.datasets import make_blobs
from sklearn.metrics import euclidean_distances
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import Ridge

from skopt.space import Real, Integer
from skopt.utils import use_named_args
from skopt import gp_minimize
from skopt.plots import plot_convergence

# Varios
# =====
import multiprocessing
import random
from itertools import product
from fitter import Fitter, get_common_distributions
```

```
In [2]: # Configuración matplotlib
# =====
plt.rcParams['image.cmap'] = "bwr"
#plt.rcParams['figure.dpi'] = "100"
plt.rcParams['savefig.bbox'] = "tight"
style.use('ggplot') or plt.style.use('ggplot')

# Configuración warnings
# =====
import warnings
warnings.filterwarnings('ignore')
```

Tabla de contenidos ↴

Scikit-learn
Introducción
Librerías
Datos
▼ Análisis exploratorio
Tipo de cada columna
Número de observaciones y v
Variable respuesta
Variables numéricas
Correlación variables numéric
Variables cualitativas
División train y test
▼ Preprocesado
Imputación de valores ausent
Exclusión de variables con va
Estandarización y escalado d
Binarización de las variables
Pipeline y ColumnTransformer
▼ Crear un modelo
Entrenamiento
Validación
Predicción
Error de test
▼ Hiperparámetros (tuning)
Grid search
Random grid search
Optimización bayesiana
Tuning del preprocesado
▼ Algoritmos
K-Nearest Neighbor (kNN)
Regresión lineal (Ridge y Las
Random Forest
Gradient Boosting Trees
▼ Stacking
Algoritmo Super Learner
Comparación
▼ Anexos
Anexo1: Métodos de validaci
Anexo2: Métricas
Bibliografía

Datos

El set de datos `SaratogaHouses` del paquete `mosaicData` de `R` contiene información sobre el precio de 1 viviendas situadas en Saratoga County, New York, USA en el año 2006. Además del precio, incluye 15 variables adicionales:

- `price`: precio de la vivienda.
- `lotSize`: metros cuadrados de la vivienda.
- `age`: antigüedad de la vivienda.
- `landValue`: valor del terreno.
- `livingArea`: metros cuadrados habitables.
- `pctCollege`: porcentaje del vecindario con título universitario.
- `bedrooms`: número de dormitorios.
- `firplaces`: número de chimeneas.
- `bathrooms`: número de cuartos de baño (el valor 0.5 hace referencia a cuartos de baño sin ducha).
- `rooms`: número de habitaciones.
- `heating`: tipo de calefacción.
- `fuel`: tipo de alimentación de la calefacción (gas, electricidad o diesel).
- `sewer`: tipo de desague.
- `waterfront`: si la vivienda tiene vistas al lago.
- `newConstruction`: si la vivienda es de nueva construcción.
- `centralAir`: si la vivienda tiene aire acondicionado.

Pueden descargarse los datos en formato csv de [SaratogaHouses](https://github.com/JoaquinAmatRodrigo/Estadistica-machine-learning-python/blob/master/data/SaratogaHouses.csv) (<https://github.com/JoaquinAmatRodrigo/Estadistica-machine-learning-python/blob/master/data/SaratogaHouses.csv>)

El objetivo es obtener un modelo capaz de predecir el precio del alquiler.

```
In [3]: url = "https://raw.githubusercontent.com/JoaquinAmatRodrigo/Estadistica-machine-learning-python
ster/data/SaratogaHouses.csv"
datos = pd.read_csv(url, sep=",")

# Se renombran las columnas para que sean más descriptivas
datos.columns = ["precio", "metros_totales", "antiguedad", "precio_terreno",
"metros_habitable", "universitarios", "dormitorios",
"chimenea", "banyos", "habitaciones", "calefaccion",
"consumo_calefaccion", "desague", "vistas_lago",
"nueva_construccion", "aire_acondicionado"]
```

Análisis exploratorio

Antes de entrenar un modelo predictivo, o incluso antes de realizar cualquier cálculo con un nuevo conjunto de datos, es muy importante realizar una exploración descriptiva de los mismos. Este proceso permite entender mejor la información que contiene cada variable, así como detectar posibles errores. Algunos ejemplos frecuentes son:

- Que una columna se haya almacenado con el tipo incorrecto: una variable numérica está siendo reconocida como texto o viceversa.
- Que una variable contenga valores que no tienen sentido: por ejemplo, para indicar que no se dispone del precio de una vivienda se introduce el valor 0 o un espacio en blanco.
- Que en una variable de tipo numérico se haya introducido una palabra en lugar de un número.

Además, este análisis inicial puede dar pistas sobre qué variables son adecuadas como predictores en un modelo (más sobre esto en los siguientes apartados).

Tabla de contenidos

Scikit-learn
Introducción
Librerías
Datos
▼ Análisis exploratorio
Tipo de cada columna
Número de observaciones y v
Variable respuesta
Variables numéricas
Correlación variables numéric
Variables cualitativas
División train y test
▼ Preprocesado
Imputación de valores ausent
Exclusión de variables con va
Estandarización y escalado d
Binarización de las variables c
Pipeline y ColumnTransformer
▼ Crear un modelo
Entrenamiento
Validación
Predicción
Error de test
▼ Hipérparámetros (tuning)
Grid search
Random grid search
Optimización bayesiana
Tuning del preprocesado
▼ Algoritmos
K-Nearest Neighbor (kNN)
Regresión lineal (Ridge y Las
Random Forest
Gradient Boosting Trees
▼ Stacking
Algoritmo Super Learner
Comparación
▼ Anexos
Anexo1: Métodos de validació
Anexo2: Métricas
Bibliografía

In [4]: `datos.head(4)`

Out[4]:

	precio	metros_totales	antiguedad	precio_terreno	metros_habitable	universitarios	dormitorios	chimenea	ban
0	132500	0.09	42	50000	906	35	2	1	
1	181115	0.92	0	22300	1953	51	3	0	
2	109000	0.19	133	7300	1944	51	4	1	
3	155000	0.41	13	18700	1944	51	3	1	

Tipo de cada columna

In [5]: `# Tipo de cada columna
======
En pandas, el tipo "object" hace referencia a strings
datos.dtypes
datos.info()`

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1728 entries, 0 to 1727
Data columns (total 16 columns):
 #   Column            Non-Null Count  Dtype  
--- 
 0   precio            1728 non-null   int64  
 1   metros_totales    1728 non-null   float64
 2   antiguedad        1728 non-null   int64  
 3   precio_terreno    1728 non-null   int64  
 4   metros_habitable  1728 non-null   int64  
 5   universitarios    1728 non-null   int64  
 6   dormitorios       1728 non-null   int64  
 7   chimenea          1728 non-null   int64  
 8   banyos             1728 non-null   float64 
 9   habitaciones       1728 non-null   int64  
 10  calefaccion        1728 non-null   object  
 11  consumo_calefacion 1728 non-null   object  
 12  desague            1728 non-null   object  
 13  vistas_lago         1728 non-null   object  
 14  nueva_construcion 1728 non-null   object  
 15  aire_acondicionado 1728 non-null   object  
dtypes: float64(2), int64(8), object(6)
memory usage: 216.1+ KB
```

Todas las columnas tienen el tipo adecuado.

Número de observaciones y valores ausentes

Junto con el estudio del tipo de variables, es básico conocer el número de observaciones disponibles y si todas están completas. Los valores ausentes son muy importantes a la hora de crear modelos, la mayoría de algoritmos aceptan observaciones incompletas o bien se ven muy influenciados por ellas. Aunque la imputación de valores ausentes es parte del preprocesado y, por lo tanto, debe de aprenderse únicamente con los datos de entrenamiento, su identificación se tiene que realizar antes de separar los datos para asegurar que se establecen todas las estrategias de imputación necesarias.

In [6]: `# Dimensiones del dataset
======
datos.shape`Out[6]: `(1728, 16)`

Tabla de contenidos

- Scikit-learn
 - Introducción
 - Librerías
 - Datos
- ▼ Análisis exploratorio
 - Tipo de cada columna
 - Número de observaciones y v
 - Variable respuesta
 - Variables numéricas
 - Correlación variables numéric
 - Variables cualitativas
- División train y test
- ▼ Preprocesado
 - Imputación de valores ausent
 - Exclusión de variables con va
 - Estandarización y escalado d
 - Binarización de las variables c
 - Pipeline y ColumnTransformer
- ▼ Crear un modelo
 - Entrenamiento
 - Validación
 - Predicción
 - Error de test
- ▼ Hiperparámetros (tuning)
 - Grid search
 - Random grid search
 - Optimización bayesiana
 - Tuning del preprocesado
- ▼ Algoritmos
 - K-Nearest Neighbor (kNN)
 - Regresión lineal (Ridge y Las
 - Random Forest
 - Gradient Boosting Trees
- ▼ Stacking
 - Algoritmo Super Learner
 - Comparación
- ▼ Anexos
 - Anexo1: Métodos de validaci
 - Anexo2: Métricas
- Bibliografía

```
In [7]: # Número de datos ausentes por variable
# =====
datos.isna().sum().sort_values()

Out[7]: precio          0
metros_totales        0
antiguedad            0
precio_terreno         0
metros_habitable       0
universitarios         0
dormitorios            0
chimenea               0
banyos                 0
habitaciones           0
calefaccion            0
consumo_calefacion     0
desague                0
vistas_lago             0
nueva_construccion     0
aire_acondicionado     0
dtype: int64
```

Ninguna variable contiene valores ausente. En el apartado **imputación de valores ausentes** se muestra va estrategias de imputación cuando el set de datos está incompleto.

Variable respuesta

Cuando se crea un modelo, es muy importante estudiar la distribución de la variable respuesta, ya que, a fincuentas, es lo que interesa predecir. La variable **precio** tiene una distribución asimétrica con una cola pos debido a que, unas pocas viviendas, tienen un precio muy superior a la media. Este tipo de distribución si visualizarse mejor tras aplicar el logarítmica o la raíz cuadrada.

Tabla de contenidos

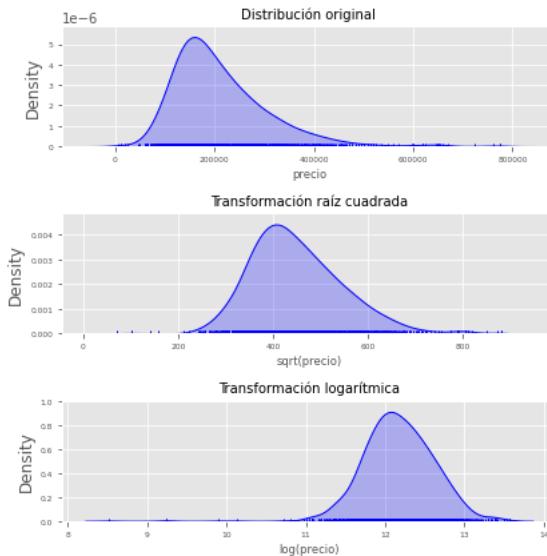
Scikit-learn
Introducción
Librerías
Datos
▼ Análisis exploratorio
Tipo de cada columna
Número de observaciones y v
Variable respuesta
Variables numéricas
Correlación variables numéric
Variables cualitativas
División train y test
▼ Preprocesado
Imputación de valores ausent
Exclusión de variables con va
Estandarización y escalado d
Binarización de las variables c
Pipeline y ColumnTransformer
▼ Crear un modelo
Entrenamiento
Validación
Predicción
Error de test
▼ Hiperparámetros (tuning)
Grid search
Random grid search
Optimización bayesiana
Tuning del preprocesado
▼ Algoritmos
K-Nearest Neighbor (kNN)
Regresión lineal (Ridge y Las
Random Forest
Gradient Boosting Trees
▼ Stacking
Algoritmo Super Learner
Comparación
▼ Anexos
Anexo1: Métodos de validació
Anexo2: Métricas
Bibliografía

```
In [8]: fig, axes = plt.subplots(nrows=3, ncols=1, figsize=(6, 6))
sns.distplot(
    datos.precio,
    hist = False,
    rug = True,
    color = "blue",
    kde_kws = {'shade': True, 'linewidth': 1},
    ax = axes[0]
)
axes[0].set_title("Distribución original", fontsize = 'medium')
axes[0].set_xlabel('precio', fontsize='small')
axes[0].tick_params(labelsize = 6)

sns.distplot(
    np.sqrt(datos.precio),
    hist = False,
    rug = True,
    color = "blue",
    kde_kws = {'shade': True, 'linewidth': 1},
    ax = axes[1]
)
axes[1].set_title("Transformación raíz cuadrada", fontsize = 'medium')
axes[1].set_xlabel('sqrt(precio)', fontsize='small')
axes[1].tick_params(labelsize = 6)

sns.distplot(
    np.log(datos.precio),
    hist = False,
    rug = True,
    color = "blue",
    kde_kws = {'shade': True, 'linewidth': 1},
    ax = axes[2]
)
axes[2].set_title("Transformación logarítmica", fontsize = 'medium')
axes[2].set_xlabel('log(precio)', fontsize='small')
axes[2].tick_params(labelsize = 6)

fig.tight_layout()
```



Algunos modelos de *machine learning* y aprendizaje estadístico requieren que la variable respuesta se distribuya de una forma determinada. Por ejemplo, para los modelos de regresión lineal (*LM*), la distribución tiene que ser de tipo normal. Para los modelos lineales generalizados (*GLM*), la distribución tiene que ser de la [familia exponencial](#) (https://en.wikipedia.org/wiki/Exponential_family).

Existen varias librerías en **python** que permiten identificar a qué distribución se ajustan mejor los datos, una de ellas es [fitter](#). Esta librería permite ajustar cualquiera de las 80 distribuciones implementadas en [scipy](#).

Tabla de contenidos

- Scikit-learn
- Introducción
- Librerías
- Datos
- Analís exploratorio
 - Tipo de cada columna
 - Número de observaciones y v
 - Variable respuesta
 - Variables numéricas
 - Correlación variables numéric
 - Variables cualitativas
- División train y test
- Preprocesado
 - Imputación de valores ausent
 - Exclusión de variables con va
 - Estandarización y escalado d
 - Binarización de las variables
 - Pipeline y ColumnTransformer
- Crear un modelo
 - Entrenamiento
 - Validación
 - Predicción
 - Error de test
- Hiperparámetros (tuning)
 - Grid search
 - Random grid search
 - Optimización bayesiana
 - Tuning del preprocesado
- Algoritmos
 - K-Nearest Neighbor (kNN)
 - Regresión lineal (Ridge y Las
 - Random Forest
 - Gradient Boosting Trees
- Stacking
 - Algoritmo Super Learner
 - Comparación
- Anexos
 - Anexo1: Métodos de validaci
 - Anexo2: Métricas
 - Bibliografía

```
In [9]: distribuciones = ['cauchy', 'chi2', 'expon', 'exponpow', 'gamma',
                      'norm', 'powerlaw', 'beta', 'logistic']

fitter = Fitter(datos.precio, distributions=distribuciones)
fitter.fit()
fitter.summary(Nbest=10, plot=False)
```

Out[9]:

	sumsquare_error	aic	bic	kl_div
beta	2.497420e-11	3068.852573	-55037.908642	inf
logistic	4.913880e-11	3147.967019	-53883.297831	inf
cauchy	5.221450e-11	2956.669693	-53778.388707	inf
chi2	5.776892e-11	3321.818880	-53596.249282	inf
norm	6.947514e-11	3324.534158	-53284.856663	inf
expon	2.915346e-10	2824.103160	-50806.577128	inf
powerlaw	3.078034e-10	2741.669837	-50705.287086	inf
exponpow	4.841645e-10		inf -49922.566370	NaN
gamma	4.841645e-10		inf -49922.566370	3.958212

Variables numéricas

```
In [10]: # Variables numéricas
# =====
datos.select_dtypes(include=['float64', 'int']).describe()
```

Out[10]:

	precio	metros_totales	antiguedad	precio_terreno	metros_habitable	universitarios	dormitorios	
count	1728.000000	1728.000000	1728.000000	1728.000000	1728.000000	1728.000000	1728.000000	17
mean	211966.705440	0.500214	27.916088	34557.187500	1754.975694	55.567708	3.154514	
std	98441.391015	0.698680	29.209988	35021.168056	619.935553	10.333581	0.817351	
min	5000.000000	0.000000	0.000000	200.000000	616.000000	20.000000	1.000000	
25%	145000.000000	0.170000	13.000000	15100.000000	1300.000000	52.000000	3.000000	
50%	189900.000000	0.370000	19.000000	25000.000000	1634.500000	57.000000	3.000000	
75%	259000.000000	0.540000	34.000000	40200.000000	2137.750000	64.000000	4.000000	
max	775000.000000	12.200000	225.000000	412600.000000	5228.000000	82.000000	7.000000	

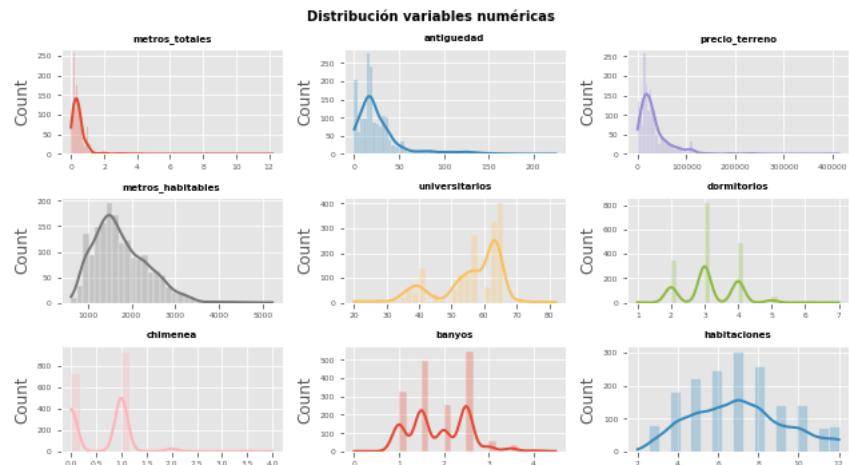
Tabla de contenidos

- Scikit-learn**
- Introducción
- Librerías
- Datos
- ▼ Análisis exploratorio
 - Tipo de cada columna
 - Número de observaciones y v
 - Variable respuesta
 - Variables numéricas
 - Correlación variables numéricas
 - Variables cualitativas
- División train y test
- ▼ Preprocesado
 - Imputación de valores ausent
 - Exclusión de variables con va
 - Estandarización y escalado d
 - Binarización de las variables c
 - Pipeline y ColumnTransformer
- ▼ Crear un modelo
 - Entrenamiento
 - Validación
 - Predicción
 - Error de test
- ▼ Hipérparámetros (tuning)
 - Grid search
 - Random grid search
 - Optimización bayesiana
 - Tuning del preprocesado
- ▼ Algoritmos
 - K-Nearest Neighbor (kNN)
 - Regresión lineal (Ridge y Las
 - Random Forest
 - Gradient Boosting Trees
- ▼ Stacking
 - Algoritmo Super Learner
 - Comparación
- ▼ Anexos
 - Anexo1: Métodos de validaci
 - Anexo2: Métricas
 - Bibliografía

```
In [11]: # Gráfico de distribución para cada variable numérica
# =====#
# Ajustar número de subplots en función del número de columnas
fig, axes = plt.subplots(nrows=3, ncols=3, figsize=(9, 5))
axes = axes.flat
columnas_numeric = datos.select_dtypes(include=['float64', 'int']).columns
columnas_numeric = columnas_numeric.drop('precio')

for i, colum in enumerate(columnas_numeric):
    sns.histplot(
        data = datos,
        x = colum,
        stat = "count",
        kde = True,
        color = (list(plt.rcParams['axes.prop_cycle'])[i]["color"]),
        line_kws= {'linewidth': 2},
        alpha = 0.3,
        ax = axes[i]
    )
    axes[i].set_title(colum, fontsize = 7, fontweight = "bold")
    axes[i].tick_params(labelsize = 6)
    axes[i].set_xlabel("")

fig.tight_layout()
plt.subplots_adjust(top = 0.9)
fig.suptitle('Distribución variables numéricas', fontsize = 10, fontweight = "bold");
```



La variable `chimenea`, aunque es de tipo numérico, apenas toma unos pocos valores y la gran mayoría de observaciones pertenecen a solo dos de ellos. En casos como este, suele ser conveniente tratar la variable como cualitativa.

```
In [12]: # Valores observados de chimenea
# =====#
datos.chimenea.value_counts()
```

```
Out[12]: 1    942
0    740
2     42
4     2
3     2
Name: chimenea, dtype: int64
```

```
In [13]: # Se convierte la variable chimenea tipo string
# =====#
datos.chimenea = datos.chimenea.astype("str")
```

Tabla de contenidos

Scikit-learn
Introducción
Librerías
Datos
▼ Análisis exploratorio
Tipo de cada columna
Número de observaciones y v
Variable respuesta
Variables numéricas
Correlación variables numéricas
Variables cualitativas
División train y test
▼ Preprocesado
Imputación de valores ausent
Exclusión de variables con va
Estandarización y escalado d
Binarización de las variables c
Pipeline y ColumnTransformer
▼ Crear un modelo
Entrenamiento
Validación
Predicción
Error de test
▼ Hipérparámetros (tuning)
Grid search
Random grid search
Optimización bayesiana
Tuning del preprocesado
▼ Algoritmos
K-Nearest Neighbor (kNN)
Regresión lineal (Ridge y Las
Random Forest
Gradient Boosting Trees
▼ Stacking
Algoritmo Super Learner
Comparación
▼ Anexos
Anexo1: Métodos de validació
Anexo2: Métricas
Bibliografía

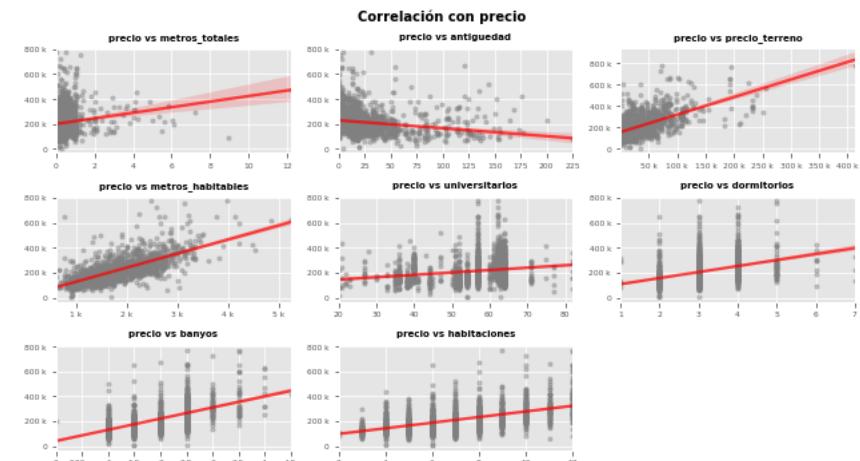
Como el objetivo del estudio es predecir el precio de las viviendas, el análisis de cada variable se hace también relación a la variable respuesta **precio**. Analizando los datos de esta forma, se pueden empezar a extraer ideas sobre qué variables están más relacionadas con el precio y de qué forma.

```
In [14]: # Gráfico de distribución para cada variable numérica
# =====
# Ajustar número de subplots en función del número de columnas
fig, axes = plt.subplots(nrows=3, ncols=3, figsize=(9, 5))
axes = axes.flat
columnas_numeric = datos.select_dtypes(include=['float64', 'int']).columns
columnas_numeric = columnas_numeric.drop('precio')

for i, colum in enumerate(columnas_numeric):
    sns.regplot(
        x          = datos[colum],
        y          = datos['precio'],
        color      = "gray",
        marker     = '.',
        scatter_kws = {"alpha":0.4},
        line_kws   = {"color":"r", "alpha":0.7},
        ax         = axes[i]
    )
    axes[i].set_title(f"precio vs {colum}", fontsize = 7, fontweight = "bold")
    #axes[i].ticklabel_format(style='sci', scilimits=(-4,4), axis='both')
    axes[i].yaxis.set_major_formatter(ticker.EngFormatter())
    axes[i].xaxis.set_major_formatter(ticker.EngFormatter())
    axes[i].tick_params(labelsize = 6)
    axes[i].set_xlabel("")
    axes[i].set_ylabel("")

# Se eliminan los axes vacíos
for i in [8]:
    fig.delaxes(axes[i])

fig.tight_layout()
plt.subplots_adjust(top=0.9)
fig.suptitle('Correlación con precio', fontsize = 10, fontweight = "bold");
```

**Correlación variables numéricas**

Algunos modelos (*LM*, *GLM*, ...) se ven perjudicados si incorporan predictores altamente correlacionados. Por este motivo, es conveniente estudiar el grado de correlación entre las variables disponibles.

Tabla de contenidos

- Scikit-learn
- Introducción
- Librerías
- Datos
- Analís exploratorio
 - Tipo de cada columna
 - Número de observaciones y v
 - Variable respuesta
 - Variables numéricas
 - Correlación variables numéricas
 - Variables cualitativas
- División train y test
- Preprocesado
 - Imputación de valores ausent
 - Exclusión de variables con va
 - Estandarización y escalado d
 - Binarización de las variables c
 - Pipeline y ColumnTransformer
- Crear un modelo
 - Entrenamiento
 - Validación
 - Predicción
 - Error de test
- Hiperparámetros (tuning)
 - Grid search
 - Random grid search
 - Optimización bayesiana
 - Tuning del preprocesado
- Algoritmos
 - K-Nearest Neighbor (kNN)
 - Regresión lineal (Ridge y Las
 - Random Forest
 - Gradient Boosting Trees
- Stacking
 - Algoritmo Super Learner
 - Comparación
- Anexos
 - Anexo1: Métodos de validaci
 - Anexo2: Métricas
 - Bibliografía

In [15]: # Correlación entre columnas numéricas

```
# =====
def tidy_corr_matrix(corr_mat):
    ...
    Función para convertir una matriz de correlación de pandas en formato tidy
    ...
    corr_mat = corr_mat.stack().reset_index()
    corr_mat.columns = ['variable_1', 'variable_2', 'r']
    corr_mat = corr_mat.loc[corr_mat['variable_1'] != corr_mat['variable_2'], :]
    corr_mat['abs_r'] = np.abs(corr_mat['r'])
    corr_mat = corr_mat.sort_values('abs_r', ascending=False)

    return(corr_mat)
```

```
corr_matrix = datos.select_dtypes(include=['float64', 'int']).corr(method='pearson')
tidy_corr_matrix(corr_matrix).head(10)
```

Out[15]:

	variable_1	variable_2	r	abs_r
44	metros_habitable	habitaciones	0.733666	0.733666
76	habitaciones	metros_habitable	0.733666	0.733666
67	banyos	metros_habitable	0.718564	0.718564
43	metros_habitable	banyos	0.718564	0.718564
36	metros_habitable	precio	0.712390	0.712390
4	precio	metros_habitable	0.712390	0.712390
78	habitaciones	dormitorios	0.671863	0.671863
62	dormitorios	habitaciones	0.671863	0.671863
42	metros_habitable	dormitorios	0.656196	0.656196
58	dormitorios	metros_habitable	0.656196	0.656196

In [16]: # Heatmap matriz de correlaciones

```
# =====
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(4, 4))

sns.heatmap(
    corr_matrix,
    annot = True,
    cbar = False,
    annot_kws = {"size": 6},
    vmin = -1,
    vmax = 1,
    center = 0,
    cmap = sns.diverging_palette(20, 220, n=200),
    square = True,
    ax = ax
)
ax.set_xticklabels(
    ax.get_xticklabels(),
    rotation = 45,
    horizontalalignment = 'right',
)
ax.tick_params(labelsize = 8)
```

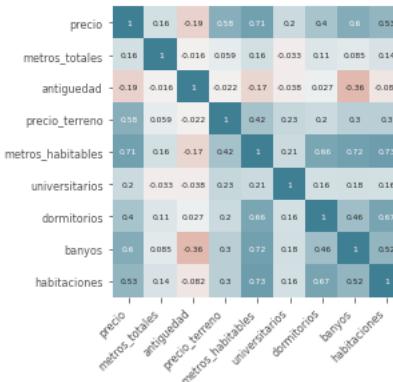


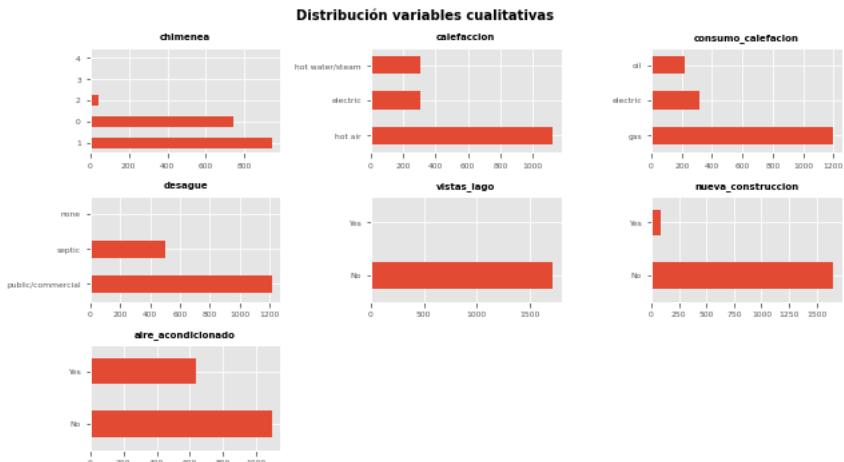
Tabla de contenidos ↗

Scikit-learn
Introducción
Librerías
Datos
▼ Análisis exploratorio
Tipo de cada columna
Número de observaciones y v
Variable respuesta
Variables numéricas
Correlación variables numéric
Variables cualitativas
División train y test
▼ Preprocesado
Imputación de valores ausent
Exclusión de variables con va
Estandarización y escalado d
Binarización de las variables c
Pipeline y ColumnTransformer
▼ Crear un modelo
Entrenamiento
Validación
Predicción
Error de test
▼ Hiperparámetros (tuning)
Grid search
Random grid search
Optimización bayesiana
Tuning del preprocesado
▼ Algoritmos
K-Nearest Neighbor (kNN)
Regresión lineal (Ridge y Las
Random Forest
Gradient Boosting Trees
▼ Stacking
Algoritmo Super Learner
Comparación
▼ Anexos
Anexo1: Métodos de validació
Anexo2: Métricas
Bibliografía

Variables cualitativas

	chimenea	calefaccion	consumo_calefacion	desague	vistas_lago	nueva_construcion	aire_acondicionado
count	1728	1728	1728	1728	1728	1728	1728
unique	5	3	3	3	3	2	2
top	1	hot air	gas	public/commercial	No	No	No
freq	942	1121	1197	1213	1713	1647	1647

In [18]:	# Gráfico para cada variable cualitativa
	# =====
	# Ajustar número de subplots en función del número de columnas
	fig, axes = plt.subplots(nrows=3, ncols=3, figsize=(9, 5))
	axes = axes.flat
	columnas_object = datos.select_dtypes(include=['object']).columns
	for i, colum in enumerate(columnas_object):
	datos[colum].value_counts().plot.barh(ax = axes[i])
	axes[i].set_title(colum, fontsize = 7, fontweight = "bold")
	axes[i].tick_params(labelsize = 6)
	axes[i].set_xlabel("")
	# Se eliminan los axes vacíos
	for i in [7, 8]:
	fig.delaxes(axes[i])
	fig.tight_layout()
	plt.subplots_adjust(top=0.9)
	fig.suptitle('Distribución variables cualitativas',
	fontsize = 10, fontweight = "bold");



Si alguno de los niveles de una variable cualitativa tiene muy pocas observaciones en comparación a los otros niveles, puede ocurrir que, durante la validación cruzada o *bootstrapping*, algunas particiones no contengan ninguna observación de dicha clase (varianza cero), lo que puede dar lugar a errores. En estos casos, suele ser conveniente:

- Eliminar las observaciones del grupo minoritario si es una variable multiclas.
- Eliminar la variable si solo tiene dos niveles.
- Agrupar los niveles minoritarios en un único grupo.
- Asegurar que, en la creación de las particiones, todos los grupos estén representados en cada una de ellas.

Para este caso, hay que tener precaución con la variable `chimenea`. Se unifican los niveles de 2, 3 y 4 en un nuevo nivel llamado "2_mas".

Tabla de contenidos

Scikit-learn
Introducción
Librerías
Datos
▼ Análisis exploratorio
Tipo de cada columna
Número de observaciones y variables
Variable respuesta
Variables numéricas
Correlación variables numéricas
Variables cualitativas
División train y test
▼ Preprocesado
Imputación de valores ausentes
Exclusión de variables con varianza cero
Estandarización y escalado de variables
Binarización de las variables categóricas
Pipeline y ColumnTransformer
▼ Crear un modelo
Entrenamiento
Validación
Predicción
Error de test
▼ Hiperparámetros (tuning)
Grid search
Random grid search
Optimización bayesiana
Tuning del preprocesado
▼ Algoritmos
K-Nearest Neighbor (kNN)
Regresión lineal (Ridge y Least Squares)
Random Forest
Gradient Boosting Trees
▼ Stacking
Algoritmo Super Learner
Comparación
▼ Anexos
Anexo1: Métodos de validación cruzada
Anexo2: Métricas
Bibliografía

In [19]: `datos.chimenea.value_counts().sort_index()`

```
Out[19]: 0    740
         1    942
         2     42
         3      2
         4      2
Name: chimenea, dtype: int64
```

```
In [20]: dic_replace = {'2': "2_mas",
                     '3': "2_mas",
                     '4': "2_mas"}

datos['chimenea'] = datos['chimenea'] \
    .map(dic_replace) \
    .fillna(datos['chimenea'])
```

In [21]: `datos.chimenea.value_counts().sort_index()`

```
Out[21]: 0        740
         1       942
         2_mas     46
Name: chimenea, dtype: int64
```

```
In [22]: # Gráfico relación entre el precio y cada cada variables cualitativas
# =====
# Ajustar número de subplots en función del número de columnas
```

```
fig, axes = plt.subplots(nrows=3, ncols=3, figsize=(9, 5))
axes = axes.flat
columnas_object = datos.select_dtypes(include=['object']).columns

for i, colum in enumerate(columnas_object):
    sns.violinplot(
        x = colum,
        y = 'precio',
        data = datos,
        color = "white",
        ax = axes[i]
    )
    axes[i].set_title(f"precio vs {colum}", fontsize = 7, fontweight = "bold")
    axes[i].yaxis.set_major_formatter(ticker.EngFormatter())
    axes[i].tick_params(labelsize = 6)
    axes[i].set_xlabel("")
    axes[i].set_ylabel("")

# Se eliminan los axes vacíos
for i in [7, 8]:
    fig.delaxes(axes[i])

fig.tight_layout()
plt.subplots_adjust(top=0.9)
fig.suptitle('Distribución del precio por grupo', fontsize = 10, fontweight = "bold");
```

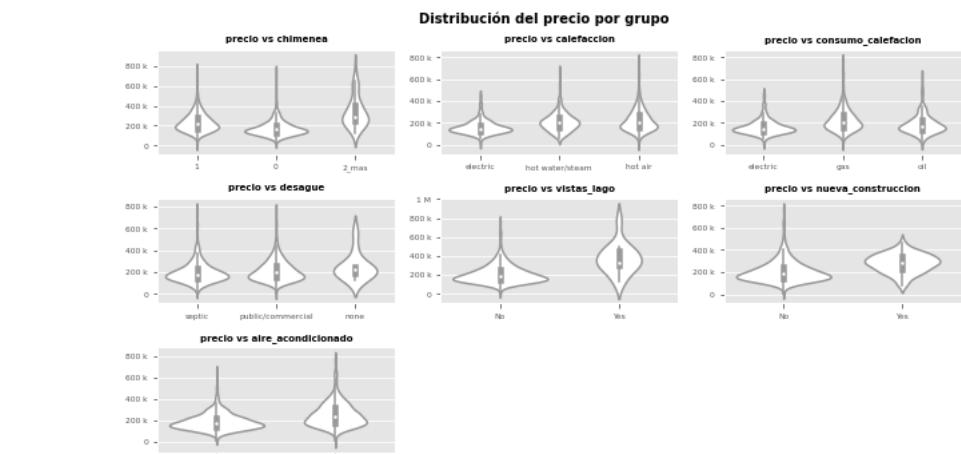


Tabla de contenidos ↗

Scikit-learn
Introducción
Librerías
Datos
▼ Análisis exploratorio
Tipo de cada columna
Número de observaciones y v
Variable respuesta
Variables numéricas
Correlación variables numéric
Variables cualitativas
División train y test
▼ Preprocesado
Imputación de valores ausent
Exclusión de variables con va
Estandarización y escalado d
Binarización de las variables c
Pipeline y ColumnTransformer
▼ Crear un modelo
Entrenamiento
Validación
Predicción
Error de test
▼ Hiperparámetros (tuning)
Grid search
Random grid search
Optimización bayesiana
Tuning del preprocesado
▼ Algoritmos
K-Nearest Neighbor (kNN)
Regresión lineal (Ridge y Las
Random Forest
Gradient Boosting Trees
▼ Stacking
Algoritmo Super Learner
Comparación
▼ Anexos
Anexo1: Métodos de validació
Anexo2: Métricas
Bibliografía

División train y test

Evaluar la capacidad predictiva de un modelo consiste en comprobar cómo de próximas son sus predicciones a verdaderos valores de la variable respuesta. Para poder cuantificarlo de forma correcta, se necesita disponer de conjunto de observaciones, de las que se conozca la variable respuesta, pero que el modelo no haya "visto", es decir que no hayan participado en su ajuste. Con esta finalidad, se dividen los datos disponibles en un conjunto de entrenamiento y un conjunto de test. El tamaño adecuado de las particiones depende en gran medida de la cantidad de datos disponibles y la seguridad que se necesite en la estimación del error, 80%-20% suele dar buenos resultados. El reparto debe hacerse de forma aleatoria o aleatoria-estratificada.

```
In [23]: # Reparto de datos en train y test
# =====
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    datos.drop('precio', axis = 'columns'),
    datos['precio'],
    train_size = 0.8,
    random_state = 1234,
    shuffle = True
)
```

Es importante verificar que la distribución de la variable respuesta es similar en el conjunto de entrenamiento y el de test. Para asegurar que esto se cumple, la función `train_test_split()` de `scikit-learn` permite, **problemas de clasificación**, identificar con el argumento `stratify` la variable en base a la cual hacer el reparto.

Este tipo de reparto estratificado asegura que el conjunto de entrenamiento y el de test sean similares en cuanto a la variable respuesta, sin embargo, no garantiza que ocurra lo mismo con los predictores. Por ejemplo, en un set de datos con 100 observaciones, un predictor binario que tenga 90 observaciones de un grupo y solo 10 de otro, tiene un alto riesgo de que, en alguna de las particiones, el grupo minoritario no tenga representantes. Si esto ocurre en el conjunto de entrenamiento, algunos algoritmos darán error al aplicarlos al conjunto de test, ya que no entenderán el valor que se les está pasando. Este problema puede evitarse eliminando variables con varianza próxima a cero (más adelante).

```
In [24]: print("Partición de entrenamiento")
print("-----")
print(y_train.describe())

Partición de entrenamiento
-----
count      1382.000000
mean     211436.516643
std      96846.639129
min     18300.000000
25%    145625.000000
50%    190000.000000
75%    255000.000000
max     775000.000000
Name: precio, dtype: float64
```

```
In [25]: print("Partición de test")
print("-----")
print(y_test.describe())

Partición de test
-----
count      346.000000
mean     214084.395954
std      104689.155889
min      5000.000000
25%    139000.000000
50%    180750.000000
75%    271750.000000
max     670000.000000
Name: precio, dtype: float64
```

Tabla de contenidos ↗

Scikit-learn
Introducción
Librerías
Datos
▼ Análisis exploratorio
Tipo de cada columna
Número de observaciones y v
Variable respuesta
Variables numéricas
Correlación variables numéric
Variables cualitativas
División train y test
Preprocesado
Imputación de valores ausent
Exclusión de variables con va
Estandarización y escalado d
Binarización de las variables
Pipeline y ColumnTransformer
Crear un modelo
Entrenamiento
Validación
Predicción
Error de test
▼ Hiperparámetros (tuning)
Grid search
Random grid search
Optimización bayesiana
Tuning del preprocesado
Algoritmos
K-Nearest Neighbor (kNN)
Regresión lineal (Ridge y Las
Random Forest
Gradient Boosting Trees
▼ Stacking
Algoritmo Super Learner
Comparación
Anexos
Anexo1: Métodos de validaci
Anexo2: Métricas
Bibliografía

Preprocesado

El preprocesado engloba todas aquellas transformaciones realizadas sobre los datos con el objetivo que puedan interpretados por el algoritmo de *machine learning* lo más eficientemente posible. Todo preprocesado de datos debe aprenderse con las observaciones de entrenamiento y luego aplicarse al conjunto de entrenamiento y al de test. Es muy importante para no violar la condición de que ninguna información procedente de las observaciones de participe o influya en el ajuste del modelo. Este principio debe aplicarse también si se emplea validación cruzada más adelante). En tal caso, el preprocesado debe realizarse dentro de cada iteración de validación, para que estimaciones que se hacen con cada partición de validación no contengan información del resto de partició Aunque no es posible crear un único listado, a continuación se resumen algunos de los pasos de preprocesado más se suelen necesitar.

Tabla de contenidos ↴

Scikit-learn
Introducción
Librerías
Datos
▼ Análisis exploratorio
Tipo de cada columna
Número de observaciones y variables
Variable respuesta
Variables numéricas
Correlación variables numéricas
Variables cualitativas
División train y test
▼ Preprocesado
Imputación de valores ausentes
Exclusión de variables con variancia cero
Estandarización y escalado de variables
Binarización de las variables
Pipeline y ColumnTransformer
▼ Crear un modelo
Entrenamiento
Validación
Predicción
Error de test
▼ Hiperparámetros (tuning)
Grid search
Random grid search
Optimización bayesiana
Tuning del preprocesado
▼ Algoritmos
K-Nearest Neighbor (kNN)
Regresión lineal (Ridge y Least Squares)
Random Forest
Gradient Boosting Trees
▼ Stacking
Algoritmo Super Learner
Comparación
▼ Anexos
Anexo1: Métodos de validación cruzada
Anexo2: Métricas
Bibliografía

Imputación de valores ausentes

La gran mayoría de algoritmos no aceptan observaciones incompletas, por lo que, cuando el set de datos contiene valores ausentes, se puede:

- Eliminar aquellas observaciones que estén incompletas.
- Eliminar aquellas variables que contengan valores ausentes.
- Tratar de estimar los valores ausentes empleando el resto de información disponible (imputación).

Las primeras dos opciones, aunque sencillas, suponen perder información. La eliminación de observaciones puede aplicarse cuando se dispone de muchas y el porcentaje de registros incompletos es muy bajo. En el caso de eliminar variables, el impacto dependerá de cuánta información aporten dichas variables al modelo. Cuando se emplea la imputación, es muy importante tener en cuenta el riesgo que se corre al introducir valores en predictores que tengan mucha influencia en el modelo. Supóngase un estudio médico en el que, cuando uno de los predictores es positivo, el modelo predice casi siempre que el paciente está sano. Para un paciente cuyo valor de este predictor se desconoce, el riesgo de que la imputación sea errónea es muy alto, por lo que es preferible obtener una predicción basada únicamente en la información disponible. Esta es otra muestra de la importancia que tiene que el analista conozca el problema al que se enfrenta y pueda así tomar la mejor decisión.

El módulo `sklearn.impute` incorpora varios métodos de imputación distintos:

- `SimpleImputer`: permite imputaciones empleando un valor constante o un estadístico (media, mediana, o más frecuente) de la misma columna en la que se encuentra el valor ausente.
- `IterativeImputer`: permite imputar el valor de una columna teniendo en cuenta el resto de columnas. De forma concreta, se trata de un proceso iterativo en el que, en cada iteración, una de las variables se emplea como variable respuesta y el resto como predictores. Una vez obtenido el modelo, se emplea para predecir las posiciones vacías de esa variable. Este proceso se lleva a cabo con cada variable y se repite el ciclo `max_iter` veces para ganar estabilidad. La implementación de `sklearn.impute.IterativeImputer` permite que se emplee casi cualquier algoritmo para crear los modelos de imputación (KNN, Random Forest, GradientBoosting...).
- `KNNImputer`: es un caso concreto de `IterativeImputer` en el que se emplea *k*-Nearest Neighbors como algoritmo de imputación.

A pesar de ser un método muy utilizado, imputar utilizando KNN presenta dos problemas: su coste computacional elevado hace que solo sea aplicable en conjuntos de datos de tamaño pequeño o moderado. Si hay variables categóricas, debido a la dificultad de medir "distancias" en este contexto, puede dar lugar a resultados poco realistas. Por estas dos razones, es más recomendable utilizar un modelo tipo Random Forest `IterativeImputer(predictors=RandomForestRegressor())`.

Con el argumento `add_indicator=True` se crea automáticamente una nueva columna en la que se indica con valor 1 qué valores han sido imputados. Esto puede ser útil tanto para identificar las observaciones en las que se ha realizado alguna imputación como para utilizarla como un predictor más en el modelo.

Tabla de contenidos ↴

Scikit-learn
Introducción
Librerías
Datos
▼ Análisis exploratorio
Tipo de cada columna
Número de observaciones y v
Variable respuesta
Variables numéricas
Correlación variables numéric
Variables cualitativas
División train y test
▼ Preprocesado
Imputación de valores ausent
Exclusión de variables con va
Estandarización y escalado d
Binarización de las variables
Pipeline y ColumnTransformer
▼ Crear un modelo
Entrenamiento
Validación
Predicción
Error de test
▼ Hiperparámetros (tuning)
Grid search
Random grid search
Optimización bayesiana
Tuning del preprocesado
▼ Algoritmos
K-Nearest Neighbor (kNN)
Regresión lineal (Ridge y Las
Random Forest
Gradient Boosting Trees
▼ Stacking
Algoritmo Super Learner
Comparación
▼ Anexos
Anexo1: Métodos de validació
Anexo2: Métricas
Bibliografía

Exclusión de variables con varianza próxima a cero

No se deben incluir en el modelo predictores que contengan un único valor (cero-varianza) ya que no aportan información. Tampoco es conveniente incluir predictores que tengan una varianza próxima a cero, es decir, predictores que toman solo unos pocos valores, de los cuales, algunos aparecen con muy poca frecuencia. El problema con estos últimos es que pueden convertirse en predictores con varianza cero cuando se dividen las observaciones por validación cruzada o *bootstrap*.

La clase `VarianceThreshold` del módulo `sklearn.feature_selection` identifica y excluye todos aquellos predictores cuya varianza no supera un determinado *threshold*. En el caso de variables cualitativas, cabe recordar que `scikitlearn` requiere que se binaricen (*one hot encoding* o *dummy*) para poder entrenar los modelos. La variable booleana sigue una distribución de Bernoulli, por lo que su varianza puede ser calculada como:

$$\text{Var}[X] = p(1 - p)$$

Si bien la eliminación de predictores no informativos podría considerarse un paso propio del proceso de *selección de predictores*, dado que consiste en un filtrado por varianza, tiene que realizarse antes de estandarizar los datos porque después, todos los predictores tienen varianza 1.

Estandarización y escalado de variables numéricas

Cuando los predictores son numéricos, la escala en la que se miden, así como la magnitud de su varianza puede influir en gran medida en el modelo. Muchos algoritmos de *machine learning* (SVM, redes neuronales, *lasso*...) son sensibles a esto, de forma que, si no se igualan de alguna forma los predictores, aquellos que se midan en una escala mayor o que tengan más varianza dominarán el modelo aunque no sean los que más relación tienen con la variable respuesta. Existen principalmente 2 estrategias para evitarlo:

- **Centrado:** consiste en restarle a cada valor la media del predictor al que pertenece. Si los datos están almacenados en un `dataframe`, el centrado se consigue restándole a cada valor la media de la columna en la que se encuentra. Como resultado de esta transformación, todos los predictores pasan a tener una media de cero, es decir, los valores se centran en torno al origen. `StandardScaler(with_std=False)`
- **Normalización (estandarización):** consiste en transformar los datos de forma que todos los predictores estén aproximadamente en la misma escala. Hay dos formas de lograrlo:
 - Normalización Z-score (`StandardScaler`): dividir cada predictor entre su desviación típica después de haber sido centrado, de esta forma, los datos pasan a tener una distribución normal.

$$z = \frac{x - \mu}{\sigma}$$

- Estandarización max-min (`MinMaxScaler`): transformar los datos de forma que estén dentro del rango [0, 1].

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

Nunca se deben estandarizar las variables después de ser binarizadas (ver a continuación).

Tabla de contenidos ↴

Scikit-learn
Introducción
Librerías
Datos
▼ Análisis exploratorio
Tipo de cada columna
Número de observaciones y v
Variable respuesta
Variables numéricas
Correlación variables numéric
Variables cualitativas
División train y test
▼ Preprocesado
Imputación de valores ausent
Exclusión de variables con va
Estandarización y escalado d
Binarización de las variables c
Pipeline y ColumnTransformer
▼ Crear un modelo
Entrenamiento
Validación
Predicción
Error de test
▼ Hiperparámetros (tuning)
Grid search
Random grid search
Optimización bayesiana
Tuning del preprocesado
▼ Algoritmos
K-Nearest Neighbor (kNN)
Regresión lineal (Ridge y Las
Random Forest
Gradient Boosting Trees
▼ Stacking
Algoritmo Super Learner
Comparación
▼ Anexos
Anexo1: Métodos de validació
Anexo2: Métricas
Bibliografía

Binarización de las variables cualitativas

La binarización (*one-hot-encoding*) consiste en crear nuevas variables *dummy* con cada uno de los niveles de variables cualitativas. Por ejemplo, una variable llamada *color* que contenga los niveles *rojo*, *verde* y *azul*, convertirá en tres nuevas variables (*color_rojo*, *color_verde*, *color_azul*), todas con el valor 0 excepto la que coincide con la observación, que toma el valor 1.

Por defecto, la clase `OneHotEncoder` binariza todas las variables, por lo que hay que aplicarlo únicamente a variables cualitativas (ver como hacerlo en el apartado *ColumnTransformer*). Con el argumento `drop='first'` elimina uno de los niveles para evitar redundancias. Volviendo al ejemplo anterior, no es necesario almacenar las variables, ya que, si *color_rojo* y *color_verde* toman el valor 0, la variable *color_azul* toma necesariamente el valor 1. Si *color_rojo* o *color_verde* toman el valor 1, entonces *color_azul* es necesariamente 0. Esto es importante para modelos que sufren problemas si los predictores están perfectamente correlacionados (modelos de regularización, redes neuronales...).

En ciertos escenarios puede ocurrir que, en los datos de test, aparezca un nuevo nivel que no estaba en los datos de entrenamiento. Si no se conoce de antemano cuáles son todos los posibles niveles, se puede evitar errores en las predicciones indicando `OneHotEncoder(handle_unknown='ignore')`.

La forma de preprocesar los datos dentro del ecosistema **scikit-learn** es empleando los `ColumnTransformer` y `pipeline`. Además de las ya mencionadas, pueden encontrarse muchas más transformaciones de preprocesado en el módulo `sklearn.preprocessing`.

Pipeline y ColumnTransformer

Las clases `ColumnTransformer` y `make_column_transformer` del módulo `sklearn.compose` permiten combinar múltiples transformaciones de preprocesado, especificando a qué columnas se aplica cada una. Como `ColumnTransformer`, tiene un método de entrenamiento (`fit`) y otro de transformación (`transform`). Esto permite que el aprendizaje de las transformaciones se haga únicamente con observaciones de entrenamiento, y se puedan aplicar después a cualquier conjunto de datos. La idea detrás de este módulo es la siguiente:

1. Definir todas las transformaciones (escalado, selección, filtrado...) que se desea aplicar y a qué columnas (`ColumnTransformer()`). La selección de columnas puede hacerse por: nombre, índice, máscara booleana, patrón `regex`, por tipo de columna o con las funciones de selección `make_column_selector`.
2. Aprender los parámetros necesarios para dichas transformaciones con las observaciones de entrenamiento (`.fit()`).
3. Aplicar las transformaciones aprendidas a cualquier conjunto de datos (`.transform()`).

```
In [26]: # Selección de las variables por tipo
# =====
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.compose import make_column_selector

# Se estandarizan las columnas numéricas y se hace one-hot-encoding de las
# columnas cualitativas. Para mantener las columnas a las que no se les aplica
# ninguna transformación se tiene que indicar remainder='passthrough'.
numeric_cols = X_train.select_dtypes(include=['float64', 'int']).columns.to_list()
cat_cols = X_train.select_dtypes(include=['object', 'category']).columns.to_list()

preprocessor = ColumnTransformer(
    [('scale', StandardScaler(), numeric_cols),
     ('onehot', OneHotEncoder(handle_unknown='ignore'), cat_cols)],
    remainder='passthrough')
```

Tabla de contenidos ↴

Scikit-learn
Introducción
Librerías
Datos
▼ Análisis exploratorio
Tipo de cada columna
Número de observaciones y v
Variable respuesta
Variables numéricas
Correlación variables numéric
Variables cualitativas
División train y test
▼ Preprocesado
Imputación de valores ausent
Exclusión de variables con va
Estandarización y escalado d
Binarización de las variables c
Pipeline y ColumnTransformer
▼ Crear un modelo
Entrenamiento
Validación
Predicción
Error de test
▼ Hipérparámetros (tuning)
Grid search
Random grid search
Optimización bayesiana
Tuning del preprocesado
▼ Algoritmos
K-Nearest Neighbor (kNN)
Regresión lineal (Ridge y Las
Random Forest
Gradient Boosting Trees
▼ Stacking
Algoritmo Super Learner
Comparación
▼ Anexos
Anexo1: Métodos de validació
Anexo2: Métricas
Bibliografía

Una vez que se ha definido el objeto `ColumnTransformer`, con el método `fit()` se aprenden las transformaciones con los datos de entrenamiento y se aplican a los dos conjuntos con `transform()`.

```
In [27]: X_train_prep = preprocessor.fit_transform(X_train)
X_test_prep = preprocessor.transform(X_test)
```

El resultado devuelto por `ColumnTransformer` es un `numpy array`, por lo que se pierden los nombres de columnas. Suele ser interesante poder inspeccionar cómo queda el set de datos tras el preprocesado en forma de dataframe. Por defecto, `OneHotEncoder` ordena las nuevas columnas de izquierda a derecha por orden alfabético

```
In [28]: # Convertir el output en dataframe y añadir el nombre de las columnas
# =====
encoded_cat = preprocessor.named_transformers_['onehot'].get_feature_names(cat_cols)
labels = np.concatenate([numeric_cols, encoded_cat])
datos_train_prep = preprocessor.transform(X_train)
datos_train_prep = pd.DataFrame(datos_train_prep, columns=labels)
datos_train_prep.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1382 entries, 0 to 1381
Data columns (total 26 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   metros_totales  1382 non-null   float64
 1   antiguedad      1382 non-null   float64
 2   precio_terreno  1382 non-null   float64
 3   metros_habitable 1382 non-null   float64
 4   universitarios  1382 non-null   float64
 5   dormitorios     1382 non-null   float64
 6   banyos          1382 non-null   float64
 7   habitaciones    1382 non-null   float64
 8   chimenea_0      1382 non-null   float64
 9   chimenea_1      1382 non-null   float64
 10  chimenea_2_mas  1382 non-null   float64
 11  calefaccion_electric 1382 non-null   float64
 12  calefaccion_hot_air 1382 non-null   float64
 13  calefaccion_hot_water/steam 1382 non-null   float64
 14  consumo_calefaccion_electric 1382 non-null   float64
 15  consumo_calefaccion_gas 1382 non-null   float64
 16  consumo_calefaccion_oil 1382 non-null   float64
 17  desague_none    1382 non-null   float64
 18  desague_public/commercial 1382 non-null   float64
 19  desague_septic   1382 non-null   float64
 20  vistas_lago_No  1382 non-null   float64
 21  vistas_lago_Yes 1382 non-null   float64
 22  nueva_construcción_No 1382 non-null   float64
 23  nueva_construcción_Yes 1382 non-null   float64
 24  aire_acondicionado_No 1382 non-null   float64
 25  aire_acondicionado_Yes 1382 non-null   float64
dtypes: float64(26)
memory usage: 280.8 KB
```

`ColumnTransformer` aplica las operaciones de forma paralela, no de forma secuencial, esto significa que no permite aplicar más de una transformación a una misma columna. En el caso de que sea necesario hacerlo, hay que recurrir a los `pipeline`, que también agrupan operaciones pero las ejecutan de forma secuencial, de forma que la salida de una operación es la entrada de la siguiente. **Si se quieren aplicar varias transformaciones de preprocesamiento sobre una misma columna, es necesario agruparlas primero en un `pipeline`**

En el siguiente ejemplo se combinan las transformaciones:

- Columnas numéricas: se imputan los valores ausentes con la mediana y a continuación se estandarizan.
- Columnas categóricas (cualitativas): se imputan los valores ausentes con el valor más frecuente y a continuación se aplica `one hot encoding`.

Tabla de contenidos

Scikit-learn
Introducción
Librerías
Datos
▼ Análisis exploratorio
Tipo de cada columna
Número de observaciones y variables
Variable respuesta
Variables numéricas
Correlación variables numéricas
Variables cualitativas
División train y test
▼ Preprocesado
Imputación de valores ausentes
Exclusión de variables con varianza cero
Estandarización y escalado de los datos
Binarización de las variables categóricas
Pipeline y ColumnTransformer
▼ Crear un modelo
Entrenamiento
Validación
Predicción
Error de test
▼ Hipérparámetros (tuning)
Grid search
Random grid search
Optimización bayesiana
Tuning del preprocesado
▼ Algoritmos
K-Nearest Neighbor (kNN)
Regresión lineal (Ridge y Least Squares)
Random Forest
Gradient Boosting Trees
▼ Stacking
Algoritmo Super Learner
Comparación
▼ Anexos
Anexo1: Métodos de validación cruzada
Anexo2: Métricas
Bibliografía

```
In [29]: # Selección de las variables por tipo
# =====
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.compose import make_column_selector

numeric_cols = X_train.select_dtypes(include=['float64', 'int']).columns.to_list()
cat_cols = X_train.select_dtypes(include=['object', 'category']).columns.to_list()

# Transformaciones para las variables numéricas
numeric_transformer = Pipeline(
    steps=[
        ('imputer', SimpleImputer(strategy='median')),
        ('scaler', StandardScaler())
    ]
)

# Transformaciones para las variables categóricas
categorical_transformer = Pipeline(
    steps=[
        ('imputer', SimpleImputer(strategy='most_frequent')),
        ('onehot', OneHotEncoder(handle_unknown='ignore'))
    ]
)

preprocessor = ColumnTransformer(
    transformers=[
        ('numerico', numeric_transformer, numeric_cols),
        ('categorico', categorical_transformer, cat_cols)
    ],
    remainder='passthrough'
)
```

```
In [30]: X_train_prep = preprocessor.fit_transform(X_train)
X_test_prep = preprocessor.transform(X_test)
```

```
In [31]: # Convertir el output en dataframe y añadir el nombre de las columnas
# =====

encoded_cat = preprocessor.named_transformers_['cat']['onehot'].get_feature_names(cat_cols)
labels = np.concatenate([numeric_cols, encoded_cat])
datos_train_prep = preprocessor.transform(X_train)
datos_train_prep = pd.DataFrame(datos_train_prep, columns=labels)
datos_train_prep.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1382 entries, 0 to 1381
Data columns (total 26 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   metros_totales  1382 non-null   float64
 1   antiguedad      1382 non-null   float64
 2   precio_terreno  1382 non-null   float64
 3   metros_habitable 1382 non-null   float64
 4   universitarios  1382 non-null   float64
 5   dormitorios     1382 non-null   float64
 6   banos           1382 non-null   float64
 7   habitaciones    1382 non-null   float64
 8   chimenea_0      1382 non-null   float64
 9   chimenea_1      1382 non-null   float64
 10  chimenea_2_mas  1382 non-null   float64
 11  calefaccion_electric 1382 non-null   float64
 12  calefaccion_hot_air 1382 non-null   float64
 13  calefaccion_hot_water/steam 1382 non-null   float64
 14  consumo_calefaccion_electric 1382 non-null   float64
 15  consumo_calefaccion_gas 1382 non-null   float64
 16  consumo_calefaccion_oil 1382 non-null   float64
 17  desague_none    1382 non-null   float64
 18  desague_public/commercial 1382 non-null   float64
 19  desague_septic   1382 non-null   float64
 20  vistas_lago_No  1382 non-null   float64
 21  vistas_lago_Yes 1382 non-null   float64
 22  nueva_construccion_No 1382 non-null   float64
 23  nueva_construccion_Yes 1382 non-null   float64
 24  aire_acondicionado_No 1382 non-null   float64
 25  aire_acondicionado_Yes 1382 non-null   float64
dtypes: float64(26)
memory usage: 280.8 KB
```

Tabla de contenidos ↗

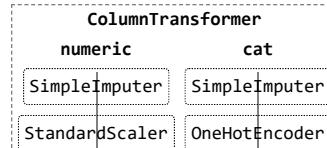
Scikit-learn
Introducción
Librerías
Datos
▼ Análisis exploratorio
Tipo de cada columna
Número de observaciones y v
Variable respuesta
Variables numéricas
Correlación variables numéric
Variables cualitativas
División train y test
▼ Preprocesado
Imputación de valores ausent
Exclusión de variables con va
Estandarización y escalado d
Binarización de las variables c
Pipeline y ColumnTransformer
▼ Crear un modelo
Entrenamiento
Validación
Predicción
Error de test
▼ Hiperparámetros (tuning)
Grid search
Random grid search
Optimización bayesiana
Tuning del preprocesado
▼ Algoritmos
K-Nearest Neighbor (kNN)
Regresión lineal (Ridge y Las
Random Forest
Gradient Boosting Trees
▼ Stacking
Algoritmo Super Learner
Comparación
▼ Anexos
Anexo1: Métodos de validació
Anexo2: Métricas
Bibliografía

A partir de las versión `scikit-learn 0.23` se puede crear una representación interactiva de un objeto `pipeline`

In [32]: `from sklearn import set_config
set_config(display='diagram')`

preprocessor

Out[32]:



In [33]: `set_config(display='text')`

Crear un modelo

El siguiente paso tras definir los datos de entrenamiento, es seleccionar el algoritmo que se va a emplear. En `scikit-learn`, esto se hace mediante la creación de un objeto `estimator`. En concreto, este objeto almacena el nombre del algoritmo, sus parámetros e hiperparámetros y contiene los métodos `fit(X, y)` y `predict(T)` que le permiten aprender de los datos y predecir nuevas observaciones. El siguiente [listado](https://scikit-learn.org/stable/user_guide.html) (https://scikit-learn.org/stable/user_guide.html) contiene todos los algoritmos implementados en `scikit-learn`.

Entrenamiento

Se ajusta un primer modelo de regresión lineal con regularización `ridge` para predecir el precio de la vivienda a función de todos los predictores disponibles. Todos los argumentos de `sklearn.linear_model.Ridge` se dejan en defecto.

Es importante tener en cuenta que, cuando un modelo de regresión lineal incluye regularización en los coeficientes (`ridge`, `lasso`, `elasticnet`), deben estandarizarse los predictores. Para asegurar que el preprocesado se realiza únicamente con los datos de entrenamiento, se combinan las transformaciones y el entrenamiento en un mismo `pipeline`.

Tabla de contenidos

Scikit-learn
Introducción
Librerías
Datos
Analís exploratorio
Tipo de cada columna
Número de observaciones y v
Variable respuesta
Variables numéricas
Correlación variables numér
Variables cualitativas
División train y test
Preprocesado
Imputación de valores ausent
Exclusión de variables con va
Estandarización y escalado d
Binarización de las variables
Pipeline y ColumnTransformer
Crear un modelo
Entrenamiento
Validación
Predicción
Error de test
Hiperparámetros (tuning)
Grid search
Random grid search
Optimización bayesiana
Tuning del preprocesado
Algoritmos
K-Nearest Neighbor (kNN)
Regresión lineal (Ridge y Las
Random Forest
Gradient Boosting Trees
Stacking
Algoritmo Super Learner
Comparación
Anexos
Anexo1: Métodos de validaci
Anexo2: Métricas
Bibliografía

```
In [34]: from sklearn.linear_model import Ridge

# Preprocedado
# =====

# Identificación de columnas numéricas y categóricas
numeric_cols = X_train.select_dtypes(include=['float64', 'int']).columns.to_list()
cat_cols = X_train.select_dtypes(include=['object', 'category']).columns.to_list()

# Transformaciones para las variables numéricas
numeric_transformer = Pipeline(
    steps=[('scaler', StandardScaler())]
)

# Transformaciones para las variables categóricas
categorical_transformer = Pipeline(
    steps=[('onehot', OneHotEncoder(handle_unknown='ignore'))]
)

preprocessor = ColumnTransformer(
    transformers=[
        ('numeric', numeric_transformer, numeric_cols),
        ('cat', categorical_transformer, cat_cols)
    ],
    remainder='passthrough'
)

# Pipeline
# =====

# Se combinan los pasos de preprocesado y el modelo en un mismo pipeline
pipe = Pipeline([('preprocessing', preprocessor),
                 ('modelo', Ridge())])

# Train
# =====

# Se asigna el resultado a _ para que no se imprima por pantalla
_ = pipe.fit(X=X_train, y=y_train)
```

Validación

La finalidad última de un modelo es predecir la variable respuesta en observaciones futuras o en observaciones el modelo no ha "visto" antes. El error mostrado por defecto tras entrenar un modelo suele ser el error entrenamiento, el error que comete el modelo al predecir las observaciones que ya ha "visto". Si bien estos errores son útiles para entender cómo está aprendiendo el modelo (estudio de residuos), no es una estimación realista cómo se comporta el modelo ante nuevas observaciones (el error de entrenamiento suele ser demasiado optimista). Para conseguir una estimación más certera, y antes de recurrir al conjunto de test, se pueden emplear estrategias de validación basadas en *resampling*. **Scikit-learn** incorpora en el módulo `sklearn.model_selection` varias estrategias de validación.

- Todas ellas reciben como primer argumento un `estimator` que puede ser directamente un modelo o `pipeline`.
- Las métricas de error de regresión se devuelven siempre en negativo de forma que, cuantos más próximos a 0 el valor, mejor el ajuste. Esto es así para que, los procesos de optimización siempre sean de maximización.

La forma más sencilla es emplear la función `cross_val_score()`, que utiliza por defecto *KFold*.

Tabla de contenidos

Scikit-learn
Introducción
Librerías
Datos
▼ Análisis exploratorio
Tipo de cada columna
Número de observaciones y v
Variable respuesta
Variables numéricas
Correlación variables numéric
Variables cualitativas
División train y test
▼ Preprocesado
Imputación de valores ausent
Exclusión de variables con va
Estandarización y escalado d
Binarización de las variables
Pipeline y ColumnTransformer
▼ Crear un modelo
Entrenamiento
Validación
Predicción
Error de test
▼ Hipérparámetros (tuning)
Grid search
Random grid search
Optimización bayesiana
Tuning del preprocesado
▼ Algoritmos
K-Nearest Neighbor (kNN)
Regresión lineal (Ridge y Las
Random Forest
Gradient Boosting Trees
▼ Stacking
Algoritmo Super Learner
Comparación
▼ Anexos
Anexo1: Métodos de validaci
Anexo2: Métricas
Bibliografía

In [35]: # Validación cruzada

```
# =====
from sklearn.model_selection import cross_val_score

cv_scores = cross_val_score(
    estimator = pipe,
    X        = X_train,
    y        = y_train,
    scoring   = 'neg_root_mean_squared_error',
    cv       = 5
)

print(f"Metrics validación cruzada: {cv_scores}")
print(f"Media métricas de validación cruzada: {cv_scores.mean()}")

Metrics validación cruzada: [-53936.97817183 -53076.74364513 -62746.15219054 -65963.09754244
-48929.66260476]
Media métricas de validación cruzada: -56930.526830940995
```

También es posible utilizar otras estrategias de validación cruzada (*KFold*, *RepeatedKFold*, *LeaveOneOut*, *LeavePOut*, *ShuffleSplit*) realizando previamente el reparto con las funciones auxiliares `sklearn.model_selection` y pasando los índices al argumento `cv`.

Cada método funciona internamente de forma distinta, pero todos ellos se basan en la idea: ajustar y evaluar modelo de forma repetida, empleando cada vez distintos subconjuntos creados a partir de los datos de entrenamiento y obteniendo en cada repetición una estimación del error. El promedio de todas las estimaciones tiende a converger en el valor real del error de test. Anexo 1

In [36]: # Validación cruzada repetida

```
# =====
from sklearn.model_selection import RepeatedKFold

cv = RepeatedKFold(n_splits=5, n_repeats=5, random_state=123)
cv_scores = cross_val_score(
    estimator = pipe,
    X        = X_train,
    y        = y_train,
    scoring   = 'neg_root_mean_squared_error',
    cv       = cv
)

print(f"Metrics de validación cruzada: {cv_scores}")
print("")
print(f"Media métricas de validación cruzada: {cv_scores.mean()}")

Metrics de validación cruzada: [-66487.72274814 -56333.87119132 -48992.85039086 -55030.0092338
-58790.48472729 -62651.67080028 -55779.2103529 -61820.5882113
-57243.1819794 -48910.70385765 -50033.46929956 -51586.8240037
-62478.2206031 -65111.86204296 -54900.98944223 -64220.39882006
-53694.93490461 -64690.99877883 -51242.27642488 -52330.80605289
-56777.05219272 -58644.60923875 -55573.25900549 -64350.25804164
-50578.15339623]

Media métricas de validación cruzada: -57130.17622962478
```

La función `cross_validate` es similar a `cross_val_score` pero permite estimar varias métricas a la vez, tanto para test como para train, y devuelve los resultados en un diccionario.

Tabla de contenidos

Scikit-learn
Introducción
Librerías
Datos
▼ Análisis exploratorio
Tipo de cada columna
Número de observaciones y v
Variable respuesta
Variables numéricas
Correlación variables numéric
Variables cualitativas
División train y test
▼ Preprocesado
Imputación de valores ausent
Exclusión de variables con va
Estandarización y escalado d
Binarización de las variables c
Pipeline y ColumnTransformer
▼ Crear un modelo
Entrenamiento
Validación
Predicción
Error de test
▼ Hiperparámetros (tuning)
Grid search
Random grid search
Optimización bayesiana
Tuning del preprocesado
▼ Algoritmos
K-Nearest Neighbor (kNN)
Regresión lineal (Ridge y Las
Random Forest
Gradient Boosting Trees
▼ Stacking
Algoritmo Super Learner
Comparación
▼ Anexos
Anexo1: Métodos de validaci
Anexo2: Métricas
Bibliografía

In [37]:

```
# Validação cruzada repetida con múltiples métricas
# =====
from sklearn.model_selection import cross_validate

cv = RepeatedKFold(n_splits=3, n_repeats=5, random_state=123)
cv_scores = cross_validate(
    estimator = pipe,
    X         = X_train,
    y         = y_train,
    scoring   = ('r2', 'neg_root_mean_squared_error'),
    cv        = cv,
    return_train_score = True
)

# Se convierte el diccionario a dataframe para facilitar la visualización
cv_scores = pd.DataFrame(cv_scores)
cv_scores
```

Out[37]:

	fit_time	score_time	test_r2	train_r2	test_neg_root_mean_squared_error	train_neg_root_mean_squared_error
0	0.022077	0.011237	0.632359	0.679758	-64605.636258	-51789.256510
1	0.020592	0.010952	0.718708	0.636812	-48322.428406	-59983.019196
2	0.020795	0.010700	0.595874	0.689939	-58434.926618	-55213.669996
3	0.020548	0.010870	0.672969	0.655316	-58179.173708	-55331.536238
4	0.020579	0.011165	0.615521	0.683390	-60148.310664	-54366.426683
5	0.021876	0.011730	0.650829	0.664771	-53812.579173	-57561.184588
6	0.020461	0.011136	0.698178	0.647441	-49813.241120	-59221.407345
7	0.020722	0.011314	0.646554	0.673016	-61749.477863	-53221.315710
8	0.020936	0.011223	0.615151	0.682999	-59192.994923	-54895.180113
9	0.020982	0.011474	0.645685	0.670425	-61061.033083	-53843.654748
10	0.020997	0.011071	0.564282	0.696123	-59084.326066	-55269.382399
11	0.021377	0.011461	0.709130	0.637747	-52788.788472	-57942.367392
12	0.022120	0.011601	0.663592	0.652572	-59372.918876	-55251.685765
13	0.021590	0.012014	0.637408	0.671598	-54925.502000	-57014.192113
14	0.022014	0.011815	0.624017	0.679811	-58863.107740	-54881.130021

Tabla de contenidos

- Scikit-learn
- Introducción
- Librerías
- Datos
- ▼ Análisis exploratorio
 - Tipo de cada columna
 - Número de observaciones y v
 - Variable respuesta
 - Variables numéricas
 - Correlación variables numéric
 - Variables cualitativas
- División train y test
- ▼ Preprocesado
 - Imputación de valores ausent
 - Exclusión de variables con va
 - Estandarización y escalado d
 - Binarización de las variables c
 - Pipeline y ColumnTransformer
- ▼ Crear un modelo
 - Entrenamiento
 - Validación
 - Predicción
 - Error de test
- ▼ Hiperparámetros (tuning)
 - Grid search
 - Random grid search
 - Optimización bayesiana
 - Tuning del preprocesado
- ▼ Algoritmos
 - K-Nearest Neighbor (kNN)
 - Regresión lineal (Ridge y Las
 - Random Forest
 - Gradient Boosting Trees
- ▼ Stacking
 - Algoritmo Super Learner
 - Comparación
- ▼ Anexos
 - Anexo1: Métodos de validaci
 - Anexo2: Métricas
- Bibliografía

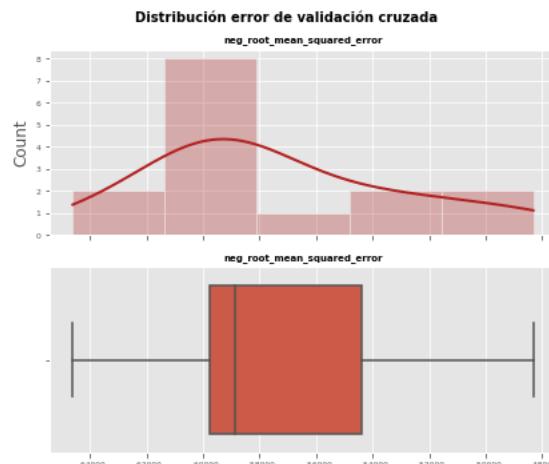
```
In [38]: # Distribución del error de validación cruzada
# =====
fig, axes = plt.subplots(nrows=2, ncols=1, figsize=(6, 5), sharex=True)

sns.histplot(
    data      = cv_scores['test_neg_root_mean_squared_error'],
    stat     = "count",
    kde      = True,
    line_kws = {'linewidth': 2},
    color   = "firebrick",
    alpha   = 0.3,
    ax      = axes[0]
)

axes[0].set_title('neg_root_mean_squared_error', fontsize = 7, fontweight = "bold")
axes[0].tick_params(labelsize = 6)
axes[0].set_xlabel("")

sns.boxplot(
    cv_scores['test_neg_root_mean_squared_error'],
    ax      = axes[1]
)
axes[1].set_title('neg_root_mean_squared_error', fontsize = 7, fontweight = "bold")
axes[1].tick_params(labelsize = 6)
axes[1].set_xlabel("")

fig.tight_layout()
plt.subplots_adjust(top=0.9)
fig.suptitle('Distribución error de validación cruzada', fontsize = 10,
             fontweight = "bold");
```



La función `cross_val_predict`, en lugar de devolver la métrica de cada partición, devuelve las predicciones de cada partición. Esto es útil para poder evaluar los residuos del modelo y diagnosticar su comportamiento. Si se emplea validación cruzada repetida o *bootstrapping*, una misma observación puede formar parte de la partición de validación varias veces.

Tabla de contenidos

Scikit-learn
Introducción
Librerías
Datos
▼ Análisis exploratorio
Tipo de cada columna
Número de observaciones y v
Variable respuesta
Variables numéricas
Correlación variables numéric
Variables cualitativas
División train y test
▼ Preprocesado
Imputación de valores ausent
Exclusión de variables con va
Estandarización y escalado d
Binarización de las variables c
Pipeline y ColumnTransformer
▼ Crear un modelo
Entrenamiento
Validación
Predicción
Error de test
▼ Hiperparámetros (tuning)
Grid search
Random grid search
Optimización bayesiana
Tuning del preprocesado
▼ Algoritmos
K-Nearest Neighbor (kNN)
Regresión lineal (Ridge y Las
Random Forest
Gradient Boosting Trees
▼ Stacking
Algoritmo Super Learner
Comparación
▼ Anexos
Anexo1: Métodos de validació
Anexo2: Métricas
Bibliografía

```
In [39]: # Diagnóstico errores (residuos) de las predicciones de validación cruzada
# =====
from sklearn.model_selection import cross_val_predict
from sklearn.model_selection import KFold
import statsmodels.api as sm

# Validación cruzada
# =====
cv = KFold(n_splits=5, random_state=123, shuffle=True)
cv_prediccones = cross_val_predict(
    estimator = pipe,
    X         = X_train,
    y         = y_train,
    cv       = cv
)

# Gráficos
# =====
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(9, 5))

axes[0, 0].scatter(y_train, cv_prediccones, edgecolors=(0, 0, 0), alpha = 0.4)
axes[0, 0].plot([y_train.min(), y_train.max()], [y_train.min(), y_train.max()],
               'k--', color = 'black', lw=2)
axes[0, 0].set_title('Valor predicho vs valor real', fontsize = 10, fontweight = "bold")
axes[0, 0].set_xlabel('Real')
axes[0, 0].set_ylabel('Predicción')
axes[0, 0].tick_params(labelsize = 7)

axes[0, 1].scatter(list(range(len(y_train))), y_train - cv_prediccones,
                  edgecolors=(0, 0, 0), alpha = 0.4)
axes[0, 1].axhline(y = 0, linestyle = '--', color = 'black', lw=2)
axes[0, 1].set_title('Residuos del modelo', fontsize = 10, fontweight = "bold")
axes[0, 1].set_xlabel('id')
axes[0, 1].set_ylabel('Residuo')
axes[0, 1].tick_params(labelsize = 7)

sns.histplot(
    data      = y_train - cv_prediccones,
    stat     = "density",
    kde      = True,
    line_kws= {'linewidth': 1},
    color   = "firebrick",
    alpha   = 0.3,
    ax      = axes[1, 0]
)

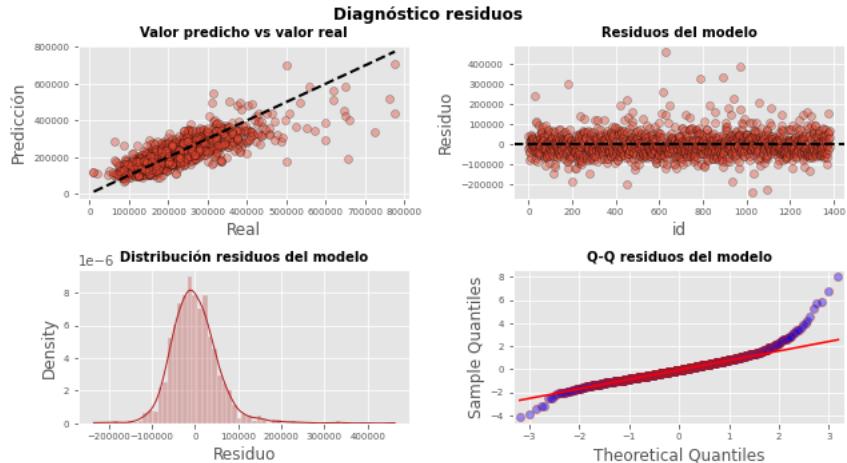
axes[1, 0].set_title('Distribución residuos del modelo', fontsize = 10,
                     fontweight = "bold")
axes[1, 0].set_xlabel("Residuo")
axes[1, 0].tick_params(labelsize = 7)

sm.qqplot(
    y_train - cv_prediccones,
    fit   = True,
    line  = 'q',
    ax   = axes[1, 1],
    color = 'firebrick',
    alpha = 0.4,
    lw    = 2
)
axes[1, 1].set_title('Q-Q residuos del modelo', fontsize = 10, fontweight = "bold")
axes[1, 1].tick_params(labelsize = 7)

fig.tight_layout()
plt.subplots_adjust(top=0.9)
fig.suptitle('Diagnóstico residuos', fontsize = 12, fontweight = "bold");
```

Tabla de contenidos

Scikit-learn
Introducción
Librerías
Datos
▼ Análisis exploratorio
Tipo de cada columna
Número de observaciones y v
Variable respuesta
Variables numéricas
Correlación variables numéric
Variables cualitativas
División train y test
▼ Preprocesado
Imputación de valores ausent
Exclusión de variables con va
Estandarización y escalado d
Binarización de las variables c
Pipeline y ColumnTransformer
▼ Crear un modelo
Entrenamiento
Validación
Predicción
Error de test
▼ Hiperparámetros (tuning)
Grid search
Random grid search
Optimización bayesiana
Tuning del preprocesado
▼ Algoritmos
K-Nearest Neighbor (kNN)
Regresión lineal (Ridge y Las
Random Forest
Gradient Boosting Trees
▼ Stacking
Algoritmo Super Learner
Comparación
▼ Anexos
Anexo1: Métodos de validaci
Anexo2: Métricas
Bibliografía



A efectos prácticos, cuando se aplican métodos de *resampling* para validar un modelo hay que tener en cuenta cosas: el coste computacional que implica ajustar múltiples veces un modelo, cada vez con un subconjunto de datos distinto, y la reproducibilidad en la creación de las particiones. Las funciones `cross_val_score()` y `cross_val_predict()` permiten parallelizar el proceso mediante el argumento `n_jobs`.

```
In [40]: # Validación cruzada repetida paralelizada (multicore)
# =====
from sklearn.model_selection import RepeatedKFold

cv = RepeatedKFold(n_splits=10, n_repeats=5, random_state=123)
cv_scores = cross_val_score(
    estimator = pipe,
    X         = X_train,
    y         = y_train,
    scoring   = 'neg_root_mean_squared_error',
    cv        = cv,
    n_jobs    = -1 # todos los cores disponibles
)

print(f"Média métricas de validación cruzada: {cv_scores.mean()}")
```

Média métricas de validación cruzada: -56735.77500472445

El `root_mean_squared_error` promedio estimado mediante validación cruzada para el modelo `ridge` es de 56735. El valor será contrastado más adelante cuando se calcule el error del modelo con el conjunto de test.

Predicción

Una vez que el modelo ha sido entrenado, bien empleando directamente un `estimator` o un `pipeline`, con su método `.predict()` se pueden predecir nuevas observaciones. Si se emplea un `pipeline`, se aplicarán automáticamente las transformaciones aprendidas durante el entrenamiento.

```
In [41]: predicciones = pipe.predict(X_test)

In [42]: # Se crea un dataframe con las predicciones y el valor real
df_predicciones = pd.DataFrame({'precio' : y_test, 'prediccion' : predicciones})
df_predicciones.head()
```

```
Out[42]:
      precio     prediccion
903  105000  112491.665253
208  113000  185142.195564
358  110500  168900.516039
1187 159000  139333.302565
319  215000  236967.476884
```

Tabla de contenidos ↴

Scikit-learn
Introducción
Librerías
Datos
▼ Análisis exploratorio
Tipo de cada columna
Número de observaciones y v
Variable respuesta
Variables numéricas
Correlación variables numéric
Variables cualitativas
■ División train y test
▼ Preprocesado
Imputación de valores ausent
Exclusión de variables con va
Estandarización y escalado d
Binarización de las variables c
Pipeline y ColumnTransformer
▼ Crear un modelo
Entrenamiento
Validación
Predicción
Error de test
▼ Hiperparámetros (tuning)
Grid search
Random grid search
Optimización bayesiana
Tuning del preprocesado
▼ Algoritmos
K-Nearest Neighbor (kNN)
Regresión lineal (Ridge y Las
Random Forest
Gradient Boosting Trees
▼ Stacking
Algoritmo Super Learner
Comparación
▼ Anexos
Anexo1: Métodos de validació
Anexo2: Métricas
Bibliografía

Error de test

Aunque mediante los métodos de validación (*Kfold*, *LeaveOneOut*) se consiguen buenas estimaciones del error tiene un modelo al predecir nuevas observaciones, la mejor forma de evaluar un modelo final es prediciendo conjunto test, es decir, un conjunto de observaciones que se ha mantenido al margen del proceso de entrenamiento optimización. Dependiendo del problema en cuestión, pueden ser interesantes unas métricas^{Anexo 2} u otras módulo [sklearn.metrics](https://scikit-learn.org/stable/modules/classes.html#module-sklearn.metrics) (<https://scikit-learn.org/stable/modules/classes.html#module-sklearn.metrics>) incorpora variedad considerable de métricas para evaluar la calidad de las predicciones.

```
In [43]: # neg_root_mean_squared_error de test
# =====
from sklearn.metrics import mean_squared_error

rmse = mean_squared_error(
    y_true = y_test,
    y_pred = predicciones,
    squared = False
)
rmse

Out[43]: 65372.916389239625
```

En el apartado de validación, se estimó, mediante validación cruzada repetida, que el *rmse* del modelo era de 56: un valor próximo al obtenido con el conjunto de test 65372.

Hiperparámetros (tuning)

Muchos modelos, entre ellos la regresión lineal con regularización *Ridge*, contienen parámetros que no pueden aprenderse a partir de los datos de entrenamiento y, por lo tanto, deben de ser establecidos por el analista. A esto: les conoce como hiperparámetros. Los resultados de un modelo pueden depender en gran medida del valor tomen sus hiperparámetros, sin embargo, no se puede conocer de antemano cuál es el adecuado. Aunque en práctica, los especialistas en *machine learning* ganan intuición sobre qué valores pueden funcionar mejor en el problema, no hay reglas fijas. La forma más común de encontrar los valores óptimos es probando diferentes posibilidades.

1. Escoger un conjunto de valores para el o los hiperparámetros.
 - *grid search*: se hace una búsqueda exhaustiva sobre un conjunto de valores previamente definidos por el usuario.
 - *random search*: se evalúan valores aleatorios dentro de unos límites definidos por el usuario.
2. Para cada valor (combinación de valores si hay más de un hiperparámetro), entrenar el modelo y estimar su error mediante un método de validación^{Anexo 1}.
3. Finalmente, ajustar de nuevo el modelo, esta vez con todos los datos de entrenamiento y con los mejores hiperparámetros encontrados.

Scikitlearn permite explorar diferentes valores de hiperparámetros mediante [model_selection.GridSearchCV\(\)](#) y [model_selection.RandomizedSearchCV\(\)](#)

Tabla de contenidos

Scikit-learn
Introducción
Librerías
Datos
▼ Análisis exploratorio
Tipo de cada columna
Número de observaciones y variables
Variable respuesta
Variables numéricas
Correlación variables numéricas
Variables cualitativas
División train y test
▼ Preprocesado
Imputación de valores ausentes
Exclusión de variables con varianza cero
Estandarización y escalado de los datos
Binarización de las variables categóricas
Pipeline y ColumnTransformer
▼ Crear un modelo
Entrenamiento
Validación
Predicción
Error de test
▼ Hiperparámetros (tuning)
Grid search
Random grid search
Optimización bayesiana
Tuning del preprocesado
▼ Algoritmos
K-Nearest Neighbor (kNN)
Regresión lineal (Ridge y Least Squares)
Random Forest
Gradient Boosting Trees
▼ Stacking
Algoritmo Super Learner
Comparación
▼ Anexos
Anexo1: Métodos de validación cruzada
Anexo2: Métricas
Bibliografía

Grid search

El modelo `Ridge` empleado hasta ahora tiene un hiperparámetro llamado `alpha`, que por defecto tiene el valor 1.0. Este hiperparámetro controla la penalización que se aplica a los coeficientes del modelo. Cuanto mayor es su valor, más restricción se impone sobre los coeficientes, reduciendo así la varianza, atenuando el efecto de la correlación entre los predictores y minimizando el riesgo de *overfitting*.

Se vuelve a ajustar un modelo `Ridge` con diferentes valores de `alpha` empleando validación cruzada repetida para identificar con cuál se obtienen mejores resultados.

```
In [44]: from sklearn.model_selection import GridSearchCV, RepeatedKFold
from sklearn.linear_model import Ridge

# Pipe: preprocesado + modelo
# =====
# Identificación de columnas numéricas y categóricas
numeric_cols = X_train.select_dtypes(include=['float64', 'int']).columns.to_list()
cat_cols = X_train.select_dtypes(include=['object', 'category']).columns.to_list()

# Transformaciones para las variables numéricas
numeric_transformer = Pipeline(
    steps=[('scaler', StandardScaler())]
)

# Transformaciones para las variables categóricas
categorical_transformer = Pipeline(
    steps=[('onehot', OneHotEncoder(handle_unknown='ignore'))]
)

preprocessor = ColumnTransformer(
    transformers=[
        ('numeric', numeric_transformer, numeric_cols),
        ('cat', categorical_transformer, cat_cols)
    ],
    remainder='passthrough'
)

# Se combinan los pasos de preprocesado y el modelo en un mismo pipeline
pipe = Pipeline([('preprocessing', preprocessor),
                 ('modelo', Ridge())])

# Grid de hiperparámetros
# =====
param_grid = {'modelo_alpha': np.logspace(-5, 3, 10)}

# Búsqueda por validación cruzada
# =====
grid = GridSearchCV(
    estimator = pipe,
    param_grid = param_grid,
    scoring = 'neg_root_mean_squared_error',
    n_jobs = multiprocessing.cpu_count() - 1,
    cv = RepeatedKFold(n_splits = 5, n_repeats = 5),
    verbose = 0,
    return_train_score = True
)

# Se asigna el resultado a _ para que no se imprima por pantalla
_ = grid.fit(X = X_train, y = y_train)
```

Tabla de contenidos

- Scikit-learn
- Introducción
- Librerías
- Datos
- Analís exploratorio
 - Tipo de cada columna
 - Número de observaciones y v
 - Variable respuesta
 - Variables numéricas
 - Correlación variables numéric
 - Variables cualitativas
- División train y test
- Preprocesado
 - Imputación de valores ausent
 - Exclusión de variables con va
 - Estandarización y escalado d
 - Binarización de las variables
 - Pipeline y ColumnTransformer
- Crear un modelo
 - Entrenamiento
 - Validación
 - Predicción
 - Error de test
- Hiperparámetros (tuning)
 - Grid search
 - Random grid search
 - Optimización bayesiana
 - Tuning del preprocesado
- Algoritmos
 - K-Nearest Neighbor (kNN)
 - Regresión lineal (Ridge y Las
 - Random Forest
 - Gradient Boosting Trees
- Stacking
 - Algoritmo Super Learner
 - Comparación
- Anexos
 - Anexo1: Métodos de validaci
 - Anexo2: Métricas
 - Bibliografía

In [45]: # Resultados del grid

```
# =====
resultados = pd.DataFrame(grid.cv_results_)
resultados.filter(regex = '(param.*|mean_t|std_t)')\
    .drop(columns = 'params')\
    .sort_values('mean_test_score', ascending = False)
```

Out[45]:

	param_modelo_alpha	mean_test_score	std_test_score	mean_train_score	std_train_score
6	2.15443	-57099.413791	5668.187123	-55893.567417	1452.746991
5	0.278256	-57147.600152	5601.004732	-55880.800914	1451.294269
4	0.0359381	-57157.010601	5590.778652	-55880.538163	1451.263268
3	0.00464159	-57158.293262	5589.427039	-55880.533650	1451.262732
2	0.000599484	-57158.460083	5589.251944	-55880.533575	1451.262723
1	7.74264e-05	-57158.481648	5589.229321	-55880.533573	1451.262723
0	1e-05	-57158.484434	5589.226399	-55880.533573	1451.262723
7	16.681	-57160.291817	5904.441472	-56150.834484	1478.756590
8	129.155	-57974.435159	6289.662299	-57333.788083	1561.883313
9	1000	-62481.463223	6888.526723	-62364.796213	1620.278470

In [46]: # Mejores hiperparámetros

```
# =====
print("-----")
print("Mejores hiperparámetros encontrados")
print("-----")
print(grid.best_params_, ":", grid.best_score_, grid.scoring)

-----
Mejores hiperparámetros encontrados
-----
{'modelo_alpha': 2.154434690031882} : -57099.41379098262 neg_root_mean_squared_error
```

Tabla de contenidos

Scikit-learn
Introducción
Librerías
Datos
▼ Análisis exploratorio
Tipo de cada columna
Número de observaciones y v
Variable respuesta
Variables numéricas
Correlación variables numéric
Variables cualitativas
División train y test
▼ Preprocesado
Imputación de valores ausent
Exclusión de variables con va
Estandarización y escalado d
Binarización de las variables c
Pipeline y ColumnTransformer
▼ Crear un modelo
Entrenamiento
Validación
Predicción
Error de test
▼ Hiperparámetros (tuning)
Grid search
Random grid search
Optimización bayesiana
Tuning del preprocesado
▼ Algoritmos
K-Nearest Neighbor (kNN)
Regresión lineal (Ridge y Las
Random Forest
Gradient Boosting Trees
▼ Stacking
Algoritmo Super Learner
Comparación
▼ Anexos
Anexo1: Métodos de validació
Anexo2: Métricas
Bibliografía

```
In [47]: # Gráfico resultados validación cruzada para cada hiperparámetro
# =====
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(10, 3.84), sharey=True)

# Gráfico 1
# -----
resultados.plot('param_modelo_alpha', 'mean_train_score', ax=axes[0])
resultados.plot('param_modelo_alpha', 'mean_test_score', ax=axes[0])
axes[0].fill_between(resultados.param_modelo_alpha.astype(np.float),
                     resultados['mean_train_score'] + resultados['std_train_score'],
                     resultados['mean_train_score'] - resultados['std_train_score'],
                     alpha=0.2)
axes[0].fill_between(resultados.param_modelo_alpha.astype(np.float),
                     resultados['mean_test_score'] + resultados['std_test_score'],
                     resultados['mean_test_score'] - resultados['std_test_score'],
                     alpha=0.2)
axes[0].legend()
axes[0].set_xscale('log')
axes[0].set_title('Evolución del error CV')
axes[0].set_ylabel('neg_root_mean_squared_error');

# Gráfico 2
# -----
numero_splits = grid.n_splits_

resultados.plot(
    x      = 'param_modelo_alpha',
    y      = [f'split{i}_train_score' for i in range(numero_splits)],
    alpha  = 0.3,
    c      = 'blue',
    ax     = axes[1]
)

resultados.plot(
    x      = 'param_modelo_alpha',
    y      = [f'split{i}_test_score' for i in range(numero_splits)],
    alpha  = 0.3,
    c      = 'red',
    ax     = axes[1]
)

axes[1].legend(
    (axes[1].get_children()[0], axes[1].get_children()[numero_splits]),
    ('training scores', 'test_scores')
)
axes[1].set_xscale('log')
axes[1].set_title('Evolución del error CV')
axes[1].set_ylabel('neg_root_mean_squared_error');

fig.tight_layout()
```

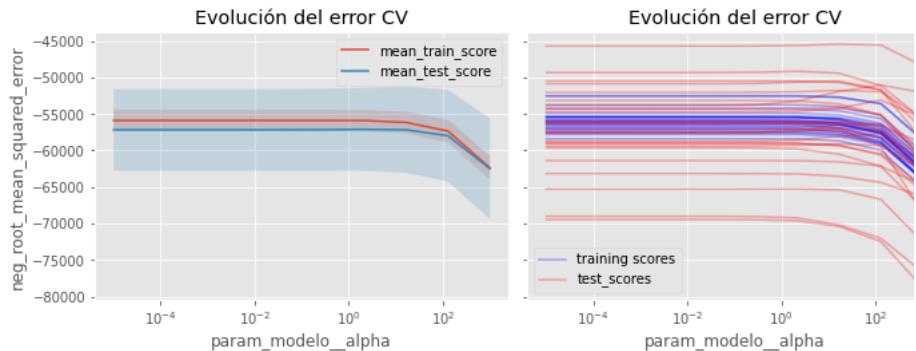
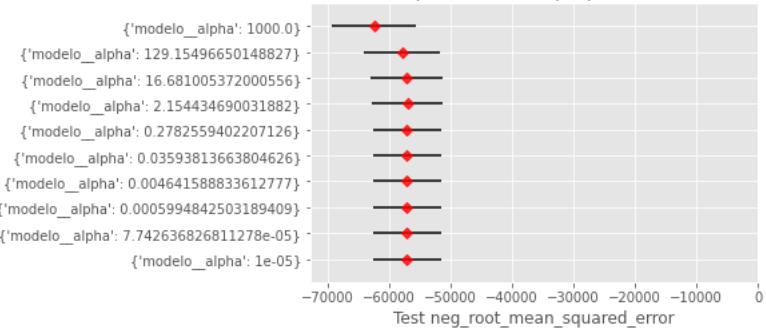


Tabla de contenidos

- Scikit-learn
- Introducción
- Librerías
- Datos
- ▼ Análisis exploratorio
 - Tipo de cada columna
 - Número de observaciones y variables
 - Variable respuesta
 - Variables numéricas
 - Correlación variables numéricas
 - Variables cualitativas
- División train y test
- ▼ Preprocesado
 - Imputación de valores ausentes
 - Exclusión de variables con varianza cero
 - Estandarización y escalado de los datos
 - Binarización de las variables categóricas
 - Pipeline y ColumnTransformer
- ▼ Crear un modelo
 - Entrenamiento
 - Validación
 - Predicción
 - Error de test
 - ▼ Hiperparámetros (tuning)
 - Grid search
 - Random grid search
 - Optimización bayesiana
 - Tuning del preprocesado
- ▼ Algoritmos
 - K-Nearest Neighbor (kNN)
 - Regresión lineal (Ridge y Least Squares)
 - Random Forest
 - Gradient Boosting Trees
- ▼ Stacking
 - Algoritmo Super Learner
 - Comparación
- ▼ Anexos
 - Anexo1: Métodos de validación cruzada
 - Anexo2: Métricas
 - Bibliografía

```
In [48]: fig, ax = plt.subplots(figsize=(6, 3.84))
ax.barh(
    [str(d) for d in resultados['params']],
    resultados['mean_test_score'],
    xerr=resultados['std_test_score'],
    align='center',
    alpha=0
)
ax.plot(
    resultados['mean_test_score'],
    [str(d) for d in resultados['params']],
    marker="D",
    linestyle="",
    alpha=0.8,
    color="r"
)
ax.set_title('Comparación de Hiperparámetros')
ax.set_xlabel('Test neg_root_mean_squared_error');
```

Comparación de Hiperparámetros



Si en `GridSearchCV()` se indica `refit=True`, tras identificar los mejores hiperparámetros, se reentrena el modelo con ellos y se almacena en `.best_estimator_`.

Random grid search

`GridSearchCV()` hace una búsqueda exhaustiva evaluando todas las combinaciones de parámetros. Esta estrategia tiene el inconveniente de que se puede invertir mucho tiempo en regiones de poco interés antes de evaluar otras combinaciones. Una alternativa es hacer una búsqueda aleatoria, de esta forma, se consigue explorar el espacio de búsqueda de una forma más distribuida. `RandomizedSearchCV()` permite este tipo de estrategia, únicamente requiere que se le indique el espacio de búsqueda de cada hiperparámetro (lista de opciones o una distribución) y el número de combinaciones aleatorias a evaluar.

Tabla de contenidos

- Scikit-learn
- Introducción
- Librerías
- Datos
- ▼ Análisis exploratorio
 - Tipo de cada columna
 - Número de observaciones y v
 - Variable respuesta
 - Variables numéricas
 - Correlación variables numéric
 - Variables cualitativas
- División train y test
- ▼ Preprocesado
 - Imputación de valores ausent
 - Exclusión de variables con va
 - Estandarización y escalado d
 - Binarización de las variables c
 - Pipeline y ColumnTransformer
- ▼ Crear un modelo
 - Entrenamiento
 - Validación
 - Predicción
 - Error de test
- ▼ Hipérparámetros (tuning)
 - Grid search
 - Random grid search
 - Optimización bayesiana
 - Tuning del preprocesado
- ▼ Algoritmos
 - K-Nearest Neighbor (kNN)
 - Regresión lineal (Ridge y Las
 - Random Forest
 - Gradient Boosting Trees
- ▼ Stacking
 - Algoritmo Super Learner
 - Comparación
- ▼ Anexos
 - Anexo1: Métodos de validaci
 - Anexo2: Métricas
- Bibliografía

```
In [49]: from itertools import product
import random

fig, axs = plt.subplots(nrows = 1, ncols = 2, figsize=(8, 4),
                      sharex = True, sharey = False)

# GRID EXHAUSTIVO
# =====
hyperparametro_1 = np.linspace(start = 0, stop = 100, num=20)
hyperparametro_2 = np.linspace(start = 0, stop = 10, num=20)

# Lista con todas las combinaciones
combinaciones = [list(x) for x in product(hyperparametro_1, hyperparametro_2)]
combinaciones = pd.DataFrame.from_records(
    combinaciones,
    columns=['hyperparametro_1', 'hyperparametro_2'])

combinaciones.plot(
    x = 'hyperparametro_1',
    y = 'hyperparametro_2',
    kind = 'scatter',
    ax = axs[0]
)
axs[0].set_title('Distribución uniforme')

# RANDOM GRID
# =====
hyperparametro_1 = np.random.uniform(low = 0, high = 100, size = 400)
hyperparametro_2 = np.random.uniform(low = 0, high = 10, size = 400)

combinaciones = pd.DataFrame(
    {
        'hyperparametro_1': hyperparametro_1,
        'hyperparametro_2': hyperparametro_2,
    }
)
combinaciones.plot(
    x = 'hyperparametro_1',
    y = 'hyperparametro_2',
    kind = 'scatter',
    ax = axs[1]
)
axs[1].set_title('Distribución aleatoria');
```

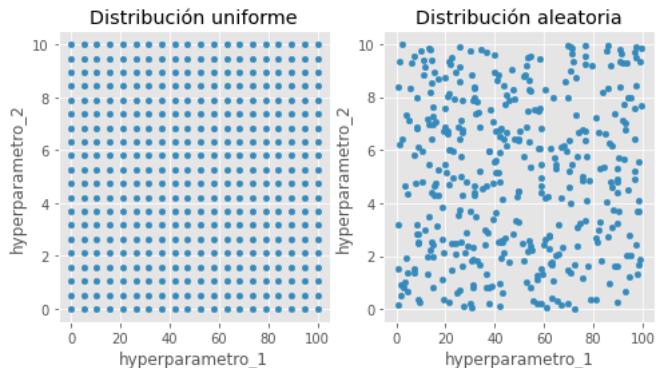


Tabla de contenidos

- Scikit-learn
- Introducción
- Librerías
- Datos
- Analís exploratorio
 - Tipo de cada columna
 - Número de observaciones y v
 - Variable respuesta
 - Variables numéricas
 - Correlación variables numéric
 - Variables cualitativas
- División train y test
- Preprocesado
 - Imputación de valores ausent
 - Exclusión de variables con va
 - Estandarización y escalado d
 - Binarización de las variables c
 - Pipeline y ColumnTransformer
- Crear un modelo
 - Entrenamiento
 - Validación
 - Predicción
 - Error de test
- Hiperparámetros (tuning)
 - Grid search
 - Random grid search
 - Optimización bayesiana
 - Tuning del preprocesado
- Algoritmos
 - K-Nearest Neighbor (kNN)
 - Regresión lineal (Ridge y Las
 - Random Forest
 - Gradient Boosting Trees
- Stacking
 - Algoritmo Super Learner
 - Comparación
- Anexos
 - Anexo1: Métodos de validaci
 - Anexo2: Métricas
 - Bibliografía

```
In [50]: from sklearn.model_selection import RandomizedSearchCV, RepeatedKFold
# Espacio de búsqueda de cada hiperparámetro
# =====
param_distributions = {'modelo_alpha': np.logspace(-5, 3, 100)}

# Búsqueda por validación cruzada
# =====
grid = RandomizedSearchCV(
    estimator = pipe,
    param_distributions = param_distributions,
    n_iter = 50,
    scoring = 'neg_root_mean_squared_error',
    n_jobs = multiprocessing.cpu_count() - 1,
    cv = RepeatedKFold(n_splits = 5, n_repeats = 5),
    verbose = 0,
    random_state = 123,
    return_train_score = True
)

grid.fit(X = X_train, y = y_train)

# Resultados del grid
# =====
resultados = pd.DataFrame(grid.cv_results_)
resultados.filter(regex = '(param.*|mean_t|std_t)')\
    .drop(columns = 'params')\
    .sort_values('mean_test_score', ascending = False)\
    .head(1)
```

Out[50]:

	param_modelo_alpha	mean_test_score	std_test_score	mean_train_score	std_train_score
33	3.76494	-56944.569878	4085.458893	-55956.220656	1042.20213

Optimización bayesiana

Aunque estas tres estrategias son totalmente válidas y generan buenos resultados, sobretodo cuando se tiene crit para acotar el rango de búsqueda, comparten una carencia común: ninguna tiene en cuenta los resultados obten hasta el momento, lo que les impide focalizar la búsqueda en las regiones de mayor interés y evitar regio innecesarias.

Una alternativa es la búsqueda de hiperparámetros con métodos de optimización bayesiana. En términos genera la optimización bayesiana de hiperparámetros consiste en crear un modelo probabilístico en el que la función obje es la métrica de validación del modelo (rmse, auc, precisión..). Con esta estrategia, se consigue que la búsqueda vaya redirigiendo en cada iteración hacia las regiones de mayor interés. El objetivo final es reducir el número combinaciones de hiperparámetros con las que se evalúa el modelo, eligiendo únicamente los mejores candida. Esto significa que, la ventaja frente a las otras estrategias mencionadas, se maximiza cuando el espacio de búsqu es muy amplio o la evaluación del modelo es muy lenta.

La librería `scikit-optimize` implementa varias estrategias de optimización Bayesiana, incluida una adaptación p modelos de `scikitlearn`.

Tabla de contenidos

- Scikit-learn
- Introducción
- Librerías
- Datos
- Analís exploratorio
 - Tipo de cada columna
 - Número de observaciones y v
 - Variable respuesta
 - Variables numéricas
 - Correlación variables numéric
 - Variables cualitativas
- División train y test
- Preprocesado
 - Imputación de valores ausent
 - Exclusión de variables con va
 - Estandarización y escalado d
 - Binarización de las variables c
 - Pipeline y ColumnTransformer
- Crear un modelo
 - Entrenamiento
 - Validación
 - Predicción
 - Error de test
- Hiperparámetros (tuning)
 - Grid search
 - Random grid search
 - Optimización bayesiana
 - Tuning del preprocesado
- Algoritmos
 - K-Nearest Neighbor (kNN)
 - Regresión lineal (Ridge y Las
 - Random Forest
 - Gradient Boosting Trees
- Stacking
 - Algoritmo Super Learner
 - Comparación
- Anexos
 - Anexo1: Métodos de validaci
 - Anexo2: Métricas
- Bibliografía

```
In [51]: from skopt.space import Real, Integer
from skopt.utils import use_named_args
from skopt import gp_minimize
from skopt.plots import plot_convergence

espacio_busqueda = [Real(1e-6, 1e+3, "log-uniform", name='modelo_alpha')]

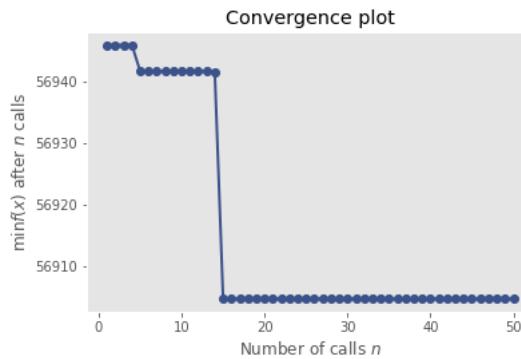
@use_named_args(espacio_busqueda)
def objective(**params):
    pipe.set_params(**params)
    return -np.mean(cross_val_score(pipe, X_train, y_train, cv=5, n_jobs=-1,
                                    scoring="neg_root_mean_squared_error"))

resultados_opt = gp_minimize(
    func          = objective,
    dimensions   = espacio_busqueda,
    n_calls       = 50,
    random_state = 0
)

print(f"Mejor score validación: {resultados_opt.fun}")
print(f"Mejores hiperparámetros: {list(zip([x.name for x in espacio_busqueda], resultados_opt.x))}")

Mejor score validación: 56904.58147502595
Mejores hiperparámetros: [('modelo_alpha', 4.8556084011694125)]
```

```
In [52]: # Evolución de la optimización
# =====
fig, ax = plt.subplots(figsize=(6, 3.84))
plot_convergence(resultados_opt, ax=ax);
```



Tuning del preprocesado

En la mayoría de casos, el proceso de optimización se centra en los hiperparámetros del modelo. Sin embargo también puede ser muy interesante comparar diferentes transformaciones de preprocesado. Gracias a los `pipelines` esto puede hacerse de la misma forma que con los hiperparámetros. Véase el siguiente ejemplo en el que compara el incorporar interacciones entre predictores.

Tabla de contenidos

- Scikit-learn
- Introducción
- Librerías
- Datos
- ▼ Análisis exploratorio
 - Tipo de cada columna
 - Número de observaciones y v
 - Variable respuesta
 - Variables numéricas
 - Correlación variables numéric
 - Variables cualitativas
- División train y test
- ▼ Preprocesado
 - Imputación de valores ausent
 - Exclusión de variables con va
 - Estandarización y escalado d
 - Binarización de las variables c
 - Pipeline y ColumnTransformer
- ▼ Crear un modelo
 - Entrenamiento
 - Validación
 - Predicción
 - Error de test
- ▼ Hiperparámetros (tuning)
 - Grid search
 - Random grid search
 - Optimización bayesiana
 - Tuning del preprocesado
- ▼ Algoritmos
 - K-Nearest Neighbor (kNN)
 - Regresión lineal (Ridge y Las
 - Random Forest
 - Gradient Boosting Trees
- ▼ Stacking
 - Algoritmo Super Learner
 - Comparación
- ▼ Anexos
 - Anexo1: Métodos de validaci
 - Anexo2: Métricas
 - Bibliografía

```
In [53]: from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import PolynomialFeatures
from sklearn.compose import make_column_selector

# Pipe: preprocesado + modelo
# =====
# Transformaciones para las variables numéricas
numeric_transformer = Pipeline(
    steps=[('scaler', StandardScaler())]
)

# Transformaciones para las variables categóricas
categorical_transformer = Pipeline(
    steps=[('onehot', OneHotEncoder(handle_unknown='ignore'))]
)

preprocessor = ColumnTransformer(
    transformers=[
        ('numeric', numeric_transformer, make_column_selector(dtype_include=np.number)),
        ('cat', categorical_transformer, make_column_selector(dtype_include='ot'))
    ],
    remainder='passthrough'
)

# Se combinan los pasos de preprocesado y el modelo en un mismo pipeline.
pipe = Pipeline([
    ('preprocessing', preprocessor),
    ('interactions', PolynomialFeatures(degree=2)),
    ('modelo', Ridge())
])

# Grid de hiperparámetros
# =====
param_grid = {'interactions': [PolynomialFeatures(degree=2), 'passthrough'],
              'modelo_alpha': np.logspace(-5, 3, 10)}

# Búsqueda por validación cruzada
# =====
grid = GridSearchCV(
    estimator = pipe,
    param_grid = param_grid,
    scoring = 'neg_root_mean_squared_error',
    n_jobs = multiprocessing.cpu_count() - 1,
    cv = RepeatedKFold(n_splits = 5, n_repeats = 5),
    verbose = 0,
    return_train_score = True
)

grid.fit(X = X_train, y = y_train)

# Resultados del grid
# =====
resultados = pd.DataFrame(grid.cv_results_)
resultados.filter(regex = '(param.*|mean_t|std_t)')\
.drop(columns = 'params')\
.sort_values('mean_test_score', ascending = False)\n.head(1)
```

Out[53]:

	param_interactions	param_modelo_alpha	mean_test_score	std_test_score	mean_train_score	std_train_score
16	passthrough		2.15443	-56924.695311	5225.795835	-55919.676221

Algoritmos

En los siguientes apartados se entranan diferentes modelos de *machine learning* con el objetivo de compararlos y identificar el que mejor resultado obtiene prediciendo el precio de las viviendas.

Tabla de contenidos ↴

Scikit-learn
Introducción
Librerías
Datos
▼ Análisis exploratorio
Tipo de cada columna
Número de observaciones y v
Variable respuesta
Variables numéricas
Correlación variables numéric
Variables cualitativas
División train y test
▼ Preprocesado
Imputación de valores ausent
Exclusión de variables con va
Estandarización y escalado d
Binarización de las variables c
Pipeline y ColumnTransformer
▼ Crear un modelo
Entrenamiento
Validación
Predicción
Error de test
▼ Hipérparámetros (tuning)
Grid search
Random grid search
Optimización bayesiana
Tuning del preprocesado
▼ Algoritmos
K-Nearest Neighbor (kNN)
Regresión lineal (Ridge y Las
Random Forest
Gradient Boosting Trees
▼ Stacking
Algoritmo Super Learner
Comparación
▼ Anexos
Anexo1: Métodos de validació
Anexo2: Métricas
Bibliografía

K-Nearest Neighbor (kNN)

K-Nearest Neighbor es uno de los algoritmos de *machine learning* más simples. Su funcionamiento es el siguiente: para predecir una observación se identifican las K observaciones del conjunto de entrenamiento que más se asemejan a ella (en base a sus predictores) y se emplea como valor predicho el promedio de la variable respuesta de dichas observaciones. Dada su sencillez, suele dar peores resultados que otros algoritmos, pero es un buen referente como *baseline*.

```
In [54]: from sklearn.model_selection import RandomizedSearchCV, RepeatedKFold
from sklearn.neighbors import KNeighborsRegressor

# Pipeline: preprocesado + modelo
# =====
# Identificación de columnas numéricas y categóricas
numeric_cols = X_train.select_dtypes(include=['float64', 'int']).columns.to_list()
cat_cols = X_train.select_dtypes(include=['object', 'category']).columns.to_list()

# Transformaciones para las variables numéricas
numeric_transformer = Pipeline(
    steps=[('scaler', StandardScaler())]
)

# Transformaciones para las variables categóricas
categorical_transformer = Pipeline(
    steps=[('onehot', OneHotEncoder(handle_unknown='ignore'))]
)

preprocessor = ColumnTransformer(
    transformers=[
        ('numerico', numeric_transformer, numeric_cols),
        ('cat', categorical_transformer, cat_cols)
    ],
    remainder='passthrough'
)

# Se combinan los pasos de preprocesado y el modelo en un mismo pipeline.
pipe = Pipeline([('preprocessing', preprocessor),
                 ('modelo', KNeighborsRegressor())])

# Optimización de hipérparámetros
# =====
# Espacio de búsqueda de cada hipérparámetro
param_distributions = {'modelo_n_neighbors': np.linspace(1, 100, 500, dtype=int)}


# Búsqueda random grid
grid = RandomizedSearchCV(
    estimator = pipe,
    param_distributions = param_distributions,
    n_iter      = 20,
    scoring     = 'neg_root_mean_squared_error',
    n_jobs      = multiprocessing.cpu_count() - 1,
    cv          = RepeatedKFold(n_splits = 5, n_repeats = 3),
    refit      = True,
    verbose     = 0,
    random_state = 123,
    return_train_score = True
)

grid.fit(X = X_train, y = y_train)

# Resultados del grid
# =====
resultados = pd.DataFrame(grid.cv_results_)
resultados.filter(regex = '(param.*|mean_t|std_t)')\
    .drop(columns = 'params')\
    .sort_values('mean_test_score', ascending = False)\n    .head(1)
```

Out[54]:

param_modelo_n_neighbors	mean_test_score	std_test_score	mean_train_score	std_train_score
12	7	-61045.466942	6751.618191	-52833.941909

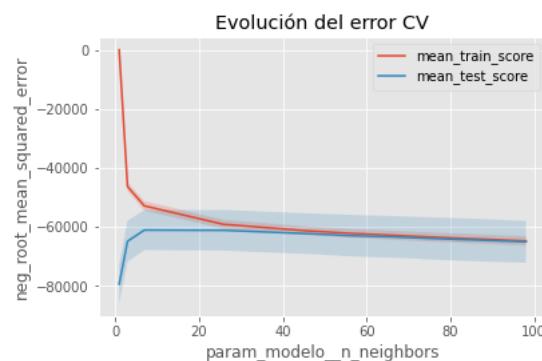
Tabla de contenidos

- Scikit-learn
- Introducción
- Librerías
- Datos
- Analís exploratorio
 - Tipo de cada columna
 - Número de observaciones y v
 - Variable respuesta
 - Variables numéricas
 - Correlación variables numéric
 - Variables cualitativas
- División train y test
- Preprocesado
 - Imputación de valores ausent
 - Exclusión de variables con va
 - Estandarización y escalado d
 - Binarización de las variables c
 - Pipeline y ColumnTransformer
- Crear un modelo
 - Entrenamiento
 - Validación
 - Predicción
 - Error de test
- Hiperparámetros (tuning)
 - Grid search
 - Random grid search
 - Optimización bayesiana
 - Tuning del preprocesado
- Algoritmos
 - K-Nearest Neighbor (kNN)
 - Regresión lineal (Ridge y Las
 - Random Forest
 - Gradient Boosting Trees
- Stacking
 - Algoritmo Super Learner
 - Comparación
- Anexos
 - Anexo1: Métodos de validaci
 - Anexo2: Métricas
 - Bibliografía

In [55]: # Gráfico resultados validación cruzada para cada hiperparámetro

```
# -----
fig, ax = plt.subplots(figsize=(6, 3.84))
hiperparametro = 'param_modelo_n_neighbors'
resultados = resultados.sort_values(hiperparametro, ascending = False)
metrica     = grid.scoring

resultados.plot(hiperparametro, 'mean_train_score', ax=ax)
resultados.plot(hiperparametro, 'mean_test_score', ax=ax)
ax.fill_between(resultados[hiperparametro].astype(np.int),
                resultados['mean_train_score'] + resultados['std_train_score'],
                resultados['mean_train_score'] - resultados['std_train_score'],
                alpha=0.2)
ax.fill_between(resultados[hiperparametro].astype(np.int),
                resultados['mean_test_score'] + resultados['std_test_score'],
                resultados['mean_test_score'] - resultados['std_test_score'],
                alpha=0.2)
ax.legend()
ax.set_title('Evolución del error CV')
ax.set_ylabel(metrica);
```



In [56]: fig, ax = plt.subplots(figsize=(8, 5))

```
resultados = resultados.sort_values('mean_test_score', ascending = True)

ax.barh(
    [str(d) for d in resultados['params']],
    resultados['mean_test_score'],
    xerr=resultados['std_test_score'],
    align='center',
    alpha=0
)
ax.plot(
    resultados['mean_test_score'],
    [str(d) for d in resultados['params']],
    marker="D",
    linestyle="",
    alpha=0.8,
    color="r"
)
ax.set_title('Comparación de Hiperparámetros')
ax.set_ylabel(metrica);
```

Comparación de Hiperparámetros

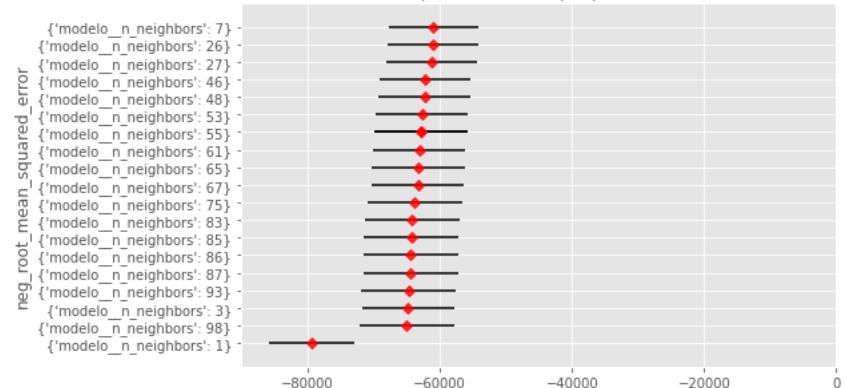


Tabla de contenidos ↴

- ⚙️ Scikit-learn
 - Introducción
 - Librerías
 - Datos
- ▼ Análisis exploratorio
 - Tipo de cada columna
 - Número de observaciones y v
 - Variable respuesta
 - Variables numéricas
 - Correlación variables numéric
 - Variables cualitativas
- División train y test
- ▶ Preprocesado
 - Imputación de valores ausent
 - Exclusión de variables con va
 - Estandarización y escalado d
 - Binarización de las variables c
 - Pipeline y ColumnTransformer
- ▼ Crear un modelo
 - Entrenamiento
 - Validación
 - Predicción
 - Error de test
- ▼ Hiperparámetros (tuning)
 - Grid search
 - Random grid search
 - Optimización bayesiana
 - Tuning del preprocesado
- ▼ Algoritmos
 - K-Nearest Neighbor (kNN)
 - Regresión lineal (Ridge y Las
 - Random Forest
 - Gradient Boosting Trees
- ▼ Stacking
 - Algoritmo Super Learner
 - Comparación
- ▼ Anexos
 - Anexo1: Métodos de validació
 - Anexo2: Métricas
- Bibliografía

Una vez identificados los mejores hiperparámetros, se reentrena el modelo indicando los valores óptimos en argumentos. Si en el `GridSearchCV()` se indica `refit=True`, este reentrenamiento se hace automáticamente y el modelo resultante se encuentra almacenado en `.best_estimator_`.

```
In [57]: # Error de test del modelo final
# =====
modelo_final = grid.best_estimator_
predicciones = modelo_final.predict(X = X_test)
rmse_knn = mean_squared_error(
    y_true = y_test,
    y_pred = predicciones,
    squared = False
)
print(f"El error (rmse) de test es: {rmse_knn}")

El error (rmse) de test es: 67303.99640453797
```

Tabla de contenidos ↴

Scikit-learn
Introducción
Librerías
Datos
▼ Análisis exploratorio
Tipo de cada columna
Número de observaciones y v
Variable respuesta
Variables numéricas
Correlación variables numéric
Variables cualitativas
División train y test
▼ Preprocesado
Imputación de valores ausent
Exclusión de variables con va
Estandarización y escalado d
Binarización de las variables c
Pipeline y ColumnTransformer
▼ Crear un modelo
Entrenamiento
Validación
Predicción
Error de test
▼ Hiperparámetros (tuning)
Grid search
Random grid search
Optimización bayesiana
Tuning del preprocesado
▼ Algoritmos
K-Nearest Neighbor (kNN)
Regresión lineal (Ridge y Lass
Random Forest
Gradient Boosting Trees
▼ Stacking
Algoritmo Super Learner
Comparación
▼ Anexos
Anexo1: Métodos de validació
Anexo2: Métricas
Bibliografía

Regresión lineal (Ridge y Lasso)

La regresión lineal es un método estadístico que trata de modelar la relación lineal entre una variable continua (variable dependiente, respuesta o dependiente) y una o más variables independientes (regresores o predicto) mediante el ajuste de una ecuación lineal. Se llama regresión lineal simple cuando solo hay una variable independiente y regresión lineal múltiple cuando hay más de una.

El modelo de regresión lineal (Legendre, Gauss, Galton y Pearson) considera que, dado un conjunto de observaciones, la media μ de la variable respuesta Y se relaciona de forma lineal con la o las variables regresoras acorde a la ecuación:

$$\mu_Y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_p x_p$$

o en notación matricial (incorporando β_0 en el vector β):

$$\mu_Y = \mathbf{X}^T \boldsymbol{\beta}$$

Además, el modelo lineal puede incluir regularización durante su ajuste, por lo que también incluye los modelos [ridge regression](#)

(https://joquinamatrodrigo.github.io/documentos/31_Seleccion_de_predictores_subset_selection_Ridge_Lasso_d_lasso)

(https://joquinamatrodrigo.github.io/documentos/31_Seleccion_de_predictores_subset_selection_Ridge_Lasso_d_elastic)

(https://joquinamatrodrigo.github.io/documentos/31_Seleccion_de_predictores_subset_selection_Ridge_Lasso_d)

Scikit-Learn incorpora 3 tipos de regularización para los modelos lineales con el objetivo de evitar *overfitting*, reducir la varianza y atenuar el efecto de la correlación entre predictores. Por lo general, aplicando regularización se consigue modelos con mayor poder predictivo (generalización).

- El modelo **lasso** es un modelo lineal por mínimos cuadrados que incorpora una regularización que penaliza la suma de los valores absolutos de los coeficientes de regresión ($||\beta||_1 = \sum_{k=1}^p |\beta_k|$). A esta penalización se le conoce como *L1* y tiene el efecto de forzar a que los coeficientes de los predictores tiendan a cero. Dado que un predictor con coeficiente de regresión cero no influye en el modelo, *lasso* consigue seleccionar los predictores más influyentes. El grado de penalización está controlado por el hiperparámetro λ . Cuando $\lambda = 0$, el resultado es equivalente al de un modelo lineal por mínimos cuadrados ordinarios. A medida que λ aumenta, mayor es la penalización y más predictores quedan excluidos.
- El modelo **ridge** es un modelo lineal por mínimos cuadrados que incorpora una regularización que penaliza la suma de los cuadrados de los coeficientes ($||\beta||_2^2 = \sum_{k=1}^p \beta_k^2$). A esta penalización se le conoce como *L2* y tiene el efecto de reducir de forma proporcional el valor de todos los coeficientes del modelo pero sin que estos lleguen a cero. Al igual que *lasso*, el grado de penalización está controlado por el hiperparámetro λ .

La principal diferencia práctica entre *lasso* y *ridge* es que el primero consigue que algunos coeficientes son exactamente cero, por lo que realiza selección de predictores, mientras que el segundo no llega a excluir ninguno. Esto supone una ventaja notable de *lasso* en escenarios donde no todos los predictores son importantes para el modelo y se desea que los menos influyentes queden excluidos. Por otro lado, cuando existen predictores altamente correlacionados (linealmente), *ridge* reduce la influencia de todos ellos a la vez y de forma proporcional, mientras que *lasso* tiende a seleccionar uno de ellos, dándole todo el peso y excluyendo al resto. En presencia de correlación, esta selección varía mucho con pequeñas perturbaciones (cambios en los datos de entrenamiento), por lo que, las soluciones de *lasso*, son muy inestables si los predictores están altamente correlacionados.

Para conseguir un equilibrio óptimo entre estas dos propiedades, se puede emplear lo que se conoce como penalización *elastic net*, que combina ambas estrategias.

El modelo *elastic net* incluye una regularización que combina la penalización *L1* y *L2* ($\alpha\lambda||\beta||_1 + \frac{1}{2}(1-\alpha)||\beta||_2^2$) en un solo término. El grado en que influye cada una de las penalizaciones está controlado por el hiperparámetro α . Su valor debe estar comprendido en el intervalo [0,1], cuando $\alpha = 0$, se aplica *ridge regression* y cuando $\alpha = 1$ se aplica *lasso*. La combinación de ambas penalizaciones suele dar lugar a buenos resultados. Una estrategia frecuentemente utilizada es asignarle casi todo el peso a la penalización *L1* (α muy próximo a 1) para conseguir seleccionar predictores y asignarles poco a la *L2* para dar cierta estabilidad en el caso de que algunos predictores estén altamente correlacionados.

Tabla de contenidos ↴

Scikit-learn
Introducción
Librerías
Datos
Analís exploratorio
Tipo de cada columna
Número de observaciones y v
Variable respuesta
Variables numéricas
Correlación variables numéric
Variables cualitativas
División train y test
Preprocesado
Imputación de valores ausent
Exclusión de variables con va
Estandarización y escalado d
Binarización de las variables
Pipeline y ColumnTransformer
Crear un modelo
Entrenamiento
Validación
Predicción
Error de test
Hiperparámetros (tuning)
Grid search
Random grid search
Optimización bayesiana
Tuning del preprocesado
Algoritmos
K-Nearest Neighbor (kNN)
Regresión lineal (Ridge y Las
Random Forest
Gradient Boosting Trees
Stacking
Algoritmo Super Learner
Comparación
Anexos
Anexo1: Métodos de validaci
Anexo2: Métricas
Bibliografía

Encontrar el mejor modelo implica identificar los valores óptimos de los hiperparámetros de regularización α y λ .

```
In [58]: from sklearn.model_selection import RandomizedSearchCV, RepeatedKFold
from sklearn.linear_model import Ridge

# Pipeline: preprocesado + modelo
# =====
# Identificación de columnas numéricas y categóricas
numeric_cols = X_train.select_dtypes(include=['float64', 'int']).columns.to_list()
cat_cols = X_train.select_dtypes(include=['object', 'category']).columns.to_list()

# Transformaciones para las variables numéricas
numeric_transformer = Pipeline(
    steps=[('scaler', StandardScaler())]
)

# Transformaciones para las variables categóricas
categorical_transformer = Pipeline(
    steps=[('onehot', OneHotEncoder(handle_unknown='ignore'))]
)

preprocessor = ColumnTransformer(
    transformers=[
        ('numeric', numeric_transformer, numeric_cols),
        ('cat', categorical_transformer, cat_cols)
    ],
    remainder='passthrough'
)

# Se combinan los pasos de preprocesado y el modelo en un mismo pipeline.
pipe = Pipeline([('preprocessing', preprocessor),
                 ('modelo', Ridge())])

# Optimización de hiperparámetros
# =====
# Espacio de búsqueda de cada hiperparámetro
param_distributions = {'modelo_alpha': np.logspace(-5, 5, 500)}

# Búsqueda random grid
grid = RandomizedSearchCV(
    estimator = pipe,
    param_distributions = param_distributions,
    n_iter = 20,
    scoring = 'neg_root_mean_squared_error',
    n_jobs = multiprocessing.cpu_count() - 1,
    cv = RepeatedKFold(n_splits = 5, n_repeats = 3),
    refit = True,
    verbose = 0,
    random_state = 123,
    return_train_score = True
)

grid.fit(X = X_train, y = y_train)

# Resultados del grid
# =====
resultados = pd.DataFrame(grid.cv_results_)
resultados.filter(regex = '(param.*|mean_t|std_t)')\
    .drop(columns = 'params')\
    .sort_values('mean_test_score', ascending = False)\
    .head(1)
```

Out[58]:

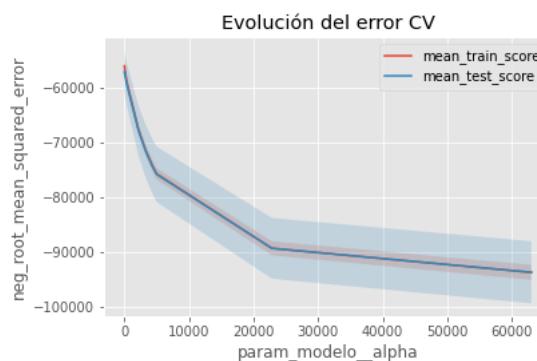
	param_modelo_alpha	mean_test_score	std_test_score	mean_train_score	std_train_score
10	3.39674	-56942.702294	3937.521061	-55951.098314	988.015086

Tabla de contenidos

Scikit-learn
Introducción
Librerías
Datos
▼ Análisis exploratorio
Tipo de cada columna
Número de observaciones y v
Variable respuesta
Variables numéricas
Correlación variables numéric
Variables cualitativas
División train y test
▼ Preprocesado
Imputación de valores ausent
Exclusión de variables con va
Estandarización y escalado d
Binarización de las variables c
Pipeline y ColumnTransformer
▼ Crear un modelo
Entrenamiento
Validación
Predicción
Error de test
▼ Hipérparámetros (tuning)
Grid search
Random grid search
Optimización bayesiana
Tuning del preprocesado
▼ Algoritmos
K-Nearest Neighbor (kNN)
Regresión lineal (Ridge y Las
Random Forest
Gradient Boosting Trees
▼ Stacking
Algoritmo Super Learner
Comparación
▼ Anexos
Anexo1: Métodos de validació
Anexo2: Métricas
Bibliografía

```
In [59]: # Gráfico resultados validación cruzada para cada hiperparámetro
# =====
fig, ax = plt.subplots(figsize=(6, 3.84))
hiperparametro = 'param_modelo_alpha'
resultados = resultados.sort_values(hiperparametro, ascending = False)
metrica      = grid.scoring

resultados.plot(hiperparametro, 'mean_train_score', ax=ax)
resultados.plot(hiperparametro, 'mean_test_score', ax=ax)
ax.fill_between(resultados[hiperparametro].astype(np.int),
                resultados['mean_train_score'] + resultados['std_train_score'],
                resultados['mean_train_score'] - resultados['std_train_score'],
                alpha=0.2)
ax.fill_between(resultados[hiperparametro].astype(np.int),
                resultados['mean_test_score'] + resultados['std_test_score'],
                resultados['mean_test_score'] - resultados['std_test_score'],
                alpha=0.2)
ax.legend()
ax.set_title('Evolución del error CV')
ax.set_ylabel(metrica);
```



```
In [60]: # Error de test del modelo final
# =====
modelo_final = grid.best_estimator_
predicciones = modelo_final.predict(X = X_test)
rmse_lm = mean_squared_error(
    y_true = y_test,
    y_pred = predicciones,
    squared = False
)
print(f"El error (rmse) de test es: {rmse_lm}")
```

El error (rmse) de test es: 65406.82894279458

Random Forest

Un modelo **Random Forest** está formado por un conjunto de árboles de decisión (https://www.cienciadedatos.net/documentos/py07_arboles_decision_python.html) individuales, cada uno entrenado con una muestra ligeramente distinta de los datos de entrenamiento generada mediante bootstrap ([https://en.wikipedia.org/wiki/Bootstrapping_\(statistics\)](https://en.wikipedia.org/wiki/Bootstrapping_(statistics))). La predicción de una nueva observación se obtiene agregando las predicciones de todos los árboles individuales que forman el modelo. Para conocer más sobre este tipo de modelo, visitar **Random Forest con Python** (https://www.cienciadedatos.net/documentos/py08_random_forest_python.html).

Tabla de contenidos

- Scikit-learn
- Introducción
- Librerías
- Datos
- Analís exploratorio
 - Tipo de cada columna
 - Número de observaciones y v
 - Variable respuesta
 - Variables numéricas
 - Correlación variables numéricas
 - Variables cualitativas
- División train y test
- Preprocesado
 - Imputación de valores ausent
 - Exclusión de variables con va
 - Estandarización y escalado d
 - Binarización de las variables c
 - Pipeline y ColumnTransformer
- Crear un modelo
 - Entrenamiento
 - Validación
 - Predicción
 - Error de test
- Hiperparámetros (tuning)
 - Grid search
 - Random grid search
 - Optimización bayesiana
 - Tuning del preprocesado
- Algoritmos
 - K-Nearest Neighbor (kNN)
 - Regresión lineal (Ridge y Las
 - Random Forest
 - Gradient Boosting Trees
- Stacking
 - Algoritmo Super Learner
 - Comparación
- Anexos
 - Anexo1: Métodos de validaci
 - Anexo2: Métricas
 - Bibliografía

```
In [61]: from sklearn.model_selection import RandomizedSearchCV, RepeatedKFold
from sklearn.ensemble import RandomForestRegressor

# Pipeline: preprocesado + modelo
# =====
# Identificación de columnas numéricas y categóricas
numeric_cols = X_train.select_dtypes(include=['float64', 'int']).columns.to_list()
cat_cols = X_train.select_dtypes(include=['object', 'category']).columns.to_list()

# Transformaciones para las variables numéricas
numeric_transformer = Pipeline(
    steps=[('scaler', StandardScaler())]
)

# Transformaciones para las variables categóricas
categorical_transformer = Pipeline(
    steps=[('onehot', OneHotEncoder(handle_unknown='ignore'))]
)

preprocessor = ColumnTransformer(
    transformers=[
        ('numeric', numeric_transformer, numeric_cols),
        ('cat', categorical_transformer, cat_cols)
    ],
    remainder='passthrough'
)

# Se combinan los pasos de preprocesado y el modelo en un mismo pipeline.
pipe = Pipeline([('preprocessing', preprocessor),
                 ('modelo', RandomForestRegressor())])

# Optimización de hiperparámetros
# =====
# Espacio de búsqueda de cada hiperparámetro

param_distributions = {
    'modelo__n_estimators': [50, 100, 1000, 2000],
    'modelo__max_features': ["auto", 3, 5, 7],
    'modelo__max_depth' : [None, 3, 5, 10, 20]
}

# Búsqueda random grid
grid = RandomizedSearchCV(
    estimator = pipe,
    param_distributions = param_distributions,
    n_iter = 20,
    scoring = 'neg_root_mean_squared_error',
    n_jobs = multiprocessing.cpu_count() - 1,
    cv = RepeatedKFold(n_splits = 5, n_repeats = 3),
    refit = True,
    verbose = 0,
    random_state = 123,
    return_train_score = True
)

grid.fit(X = X_train, y = y_train)

# Resultados del grid
# =====
resultados = pd.DataFrame(grid.cv_results_)
resultados.filter(regex = '(param.*|mean_t|std_t)')\
    .drop(columns = 'params')\
    .sort_values('mean_test_score', ascending = False)\
    .head(1)
```

Out[61]:

param_modelo__n_estimators	param_modelo__max_features	param_modelo__max_depth	mean_test_score	std_te
17	1000	7	20	-54107.705167 4151

In [62]: # Error de test del modelo final

```
# =====
modelo_final = grid.best_estimator_
predicciones = modelo_final.predict(X = X_test)
rmse_rf = mean_squared_error(
    y_true = y_test,
    y_pred = predicciones,
    squared = False
)
print(f"El error (rmse) de test es: {rmse_rf}")
```

El error (rmse) de test es: 61240.775327629955

Tabla de contenidos ↗

- ⚙️
 - Scikit-learn
 - Introducción
 - Librerías
 - Datos
 - Analís exploratorio
 - Tipo de cada columna
 - Número de observaciones y v
 - Variable respuesta
 - Variables numéricas
 - Correlación variables numéric
 - Variables cualitativas
 - División train y test
 - ▶ Preprocesado
 - Imputación de valores ausent
 - Exclusión de variables con va
 - Estandarización y escalado d
 - Binarización de las variables c
 - Pipeline y ColumnTransformer
 - ▶ Crear un modelo
 - Entrenamiento
 - Validación
 - Predicción
 - Error de test
 - ▶ Hiperparámetros (tuning)
 - Grid search
 - Random grid search
 - Optimización bayesiana
 - Tuning del preprocesado
 - ▶ Algoritmos
 - K-Nearest Neighbor (kNN)
 - Regresión lineal (Ridge y Las
 - Random Forest
 - Gradient Boosting Trees
 - ▶ Stacking
 - Algoritmo Super Learner
 - Comparación
 - ▶ Anexos
 - Anexo1: Métodos de validació
 - Anexo2: Métricas
 - Bibliografía

Gradient Boosting Trees

Un modelo **Gradient Boosting Trees** está formado por un conjunto de árboles de decisión (https://www.cienciadedatos.net/documentos/py07_arboles_decision_python.html) individuales, entrenados de forma secuencial, de forma que cada nuevo árbol trata de mejorar los errores de los árboles anteriores. La predicción de una nueva observación se obtiene agregando las predicciones de todos los árboles individuales que forman el modelo. Para conocer más sobre este tipo de modelo visitar + Gradient Boosting con Python (https://www.cienciadedatos.net/documentos/py09_gradient_boosting_python.html).

Tabla de contenidos

Scikit-learn
Introducción
Librerías
Datos
▼ Análisis exploratorio
Tipo de cada columna
Número de observaciones y v
Variable respuesta
Variables numéricas
Correlación variables numéric
Variables cualitativas
División train y test
▼ Preprocesado
Imputación de valores ausent
Exclusión de variables con va
Estandarización y escalado d
Binarización de las variables c
Pipeline y ColumnTransformer
▼ Crear un modelo
Entrenamiento
Validación
Predicción
Error de test
▼ Hipérparámetros (tuning)
Grid search
Random grid search
Optimización bayesiana
Tuning del preprocesado
▼ Algoritmos
K-Nearest Neighbor (kNN)
Regresión lineal (Ridge y Las
Random Forest
Gradient Boosting Trees
▼ Stacking
Algoritmo Super Learner
Comparación
▼ Anexos
Anexo1: Métodos de validació
Anexo2: Métricas
Bibliografía

```
In [63]: from sklearn.model_selection import RandomizedSearchCV, RepeatedKFold
from sklearn.ensemble import GradientBoostingRegressor

# Pipeline: preprocesado + modelo
# =====
# Identificación de columnas numéricas y categóricas
numeric_cols = X_train.select_dtypes(include=['float64', 'int']).columns.to_list()
cat_cols = X_train.select_dtypes(include=['object', 'category']).columns.to_list()

# Transformaciones para las variables numéricas
numeric_transformer = Pipeline(
    steps=[('scaler', StandardScaler())]
)

# Transformaciones para las variables categóricas
categorical_transformer = Pipeline(
    steps=[('onehot', OneHotEncoder(handle_unknown='ignore'))]
)

preprocessor = ColumnTransformer(
    transformers=[
        ('numeric', numeric_transformer, numeric_cols),
        ('cat', categorical_transformer, cat_cols)
    ],
    remainder='passthrough'
)

# Se combinan los pasos de preprocesado y el modelo en un mismo pipeline.
pipe = Pipeline([('preprocessing', preprocessor),
                 ('modelo', GradientBoostingRegressor())])

# Optimización de hipérparámetros
# =====
# Espacio de búsqueda de cada hipérparámetro

param_distributions = {
    'modelo_n_estimators': [50, 100, 1000, 2000],
    'modelo_max_features': ["auto", 3, 5, 7],
    'modelo_max_depth' : [None, 3, 5, 10, 20],
    'modelo_subsample' : [0.5, 0.7, 1]
}

# Búsqueda random grid
grid = RandomizedSearchCV(
    estimator = pipe,
    param_distributions = param_distributions,
    n_iter = 20,
    scoring = 'neg_root_mean_squared_error',
    n_jobs = multiprocessing.cpu_count() - 1,
    cv = RepeatedKFold(n_splits = 5, n_repeats = 3),
    refit = True,
    verbose = 0,
    random_state = 123,
    return_train_score = True
)

grid.fit(X = X_train, y = y_train)

# Resultados del grid
# =====
resultados = pd.DataFrame(grid.cv_results_)
resultados.filter(regex = '(param.*|mean_t|std_t)')\
    .drop(columns = 'params')\
    .sort_values('mean_test_score', ascending = False)\
    .head(1)
```

Out[63]:

param_modelo_subsample	param_modelo_n_estimators	param_modelo_max_features	param_modelo_max_depth
12	1	2000	3

In [64]: # Error de test del modelo final

```
# =====
modelo_final = grid.best_estimator_
predicciones = modelo_final.predict(X = X_test)
rmse_gbm = mean_squared_error(
    y_true = y_test,
    y_pred = predicciones,
    squared = False
)
print(f"El error (rmse) de test es: {rmse_gbm}")
```

El error (rmse) de test es: 61582.49312509251

Privacy

Tabla de contenidos ↴

Scikit-learn
Introducción
Librerías
Datos
▼ Análisis exploratorio
Tipo de cada columna
Número de observaciones y variables
Variable respuesta
Variables numéricas
Correlación variables numéricas
Variables cualitativas
División train y test
▼ Preprocesado
Imputación de valores ausentes
Exclusión de variables con variancia cero
Estandarización y escalado de variables
Binarización de las variables categóricas
Pipeline y ColumnTransformer
▼ Crear un modelo
Entrenamiento
Validación
Predicción
Error de test
▼ Hiperparámetros (tuning)
Grid search
Random grid search
Optimización bayesiana
Tuning del preprocesado
▼ Algoritmos
K-Nearest Neighbor (kNN)
Regresión lineal (Ridge y Least Squares)
Random Forest
Gradient Boosting Trees
Stacking
Algoritmo Super Learner
Comparación
▼ Anexos
Anexo1: Métodos de validación cruzada
Anexo2: Métricas
Bibliografía

Stacking

A modo general, el término *model ensembling* hace referencia a la combinación de las predicciones de dos o más modelos distintos con el objetivo de mejorar las predicciones finales. Esta estrategia se basa en la asunción de que los distintos modelos entrenados independientemente, emplean distintos aspectos de los datos para realizar predicciones, es decir, cada uno es capaz de identificar parte de la “verdad” pero no toda ella. Combinando la perspectiva de cada uno de ellos, se obtiene una descripción más detallada de la verdadera estructura subyacente de los datos. A modo de analogía, imagínese un grupo de estudiantes que se enfrentan a un examen multidisciplinar. Aunque todos obtengan aproximadamente la misma nota, cada uno de ellos habrá conseguido más puntos con respuestas que tratan sobre las disciplinas en las que destacan. Si en lugar de hacer el examen de forma individual, los estudiantes hacen en grupo, cada uno podrá contribuir en los aspectos que más domina, y el resultado final será probablemente superior a cualquiera de los resultados individuales.

La clave para que el *ensembling* consiga mejorar los resultados es la diversidad de los modelos. Si todos los modelos combinados son similares entre ellos, no podrán compensarse unos a otros. Por esta razón, se tiene que intentar combinar modelos que sean lo mejor posible a nivel individual y lo más diferentes entre ellos.

Las formas más simples de combinar las predicciones de varios modelos es emplear la media para problemas de regresión y la moda para problemas de clasificación. Sin embargo existen otras aproximaciones más complejas capaces de conseguir mejores resultados:

- Ponderar las agregaciones dando distinto peso a cada modelo, por ejemplo, en proporción al accuracy que han obtenido de forma individual.
- *Super Learner* (stacked regression): emplear un nuevo modelo para que decida la mejor forma de combinar las predicciones de los otros modelos.

Algoritmo Super Learner

La implementación de *Super Learner* disponible en **scikit-learn** en las clases **StackingRegressor** y **StackingClassifier** sigue el siguiente algoritmo:

Definición del ensemble

1. Definir un listado con los algoritmos base (cada uno con los hiperparámetros pertinentes).
2. Seleccionar el algoritmo de *metalearning* que defina cómo se entrena en modelo superior. Por defecto, se emplea **RidgeCV** para regresión y **LogisticRegression** para clasificación.

Entrenamiento del ensemble

1. Entrenar cada uno de los algoritmos base con el conjunto de entrenamiento.
2. Realizar *k-fold cross-validation* con cada uno de los algoritmos base y almacenar las predicciones hechas por cada una de las *k* particiones.
3. Combinar las predicciones del paso 2 en una única matriz $N \times L$ (N = número de observaciones en el conjunto de entrenamiento, L = número de modelos base).
4. Entrenar el *metalearner* con la variable respuesta y la matriz $N \times L$ como predictores.
5. El *Super learner* final está formado por los modelos base y el modelo *metalearning*.

Predecir

1. Predecir la nueva observación con cada uno de los modelos base.
2. Emplear las predicciones de los modelos base como input del *metalearner* para obtener la predicción final.

Tabla de contenidos ↴

Scikit-learn
Introducción
Librerías
Datos
▼ Análisis exploratorio
Tipo de cada columna
Número de observaciones y v
Variable respuesta
Variables numéricas
Correlación variables numéric
Variables cualitativas
■ División train y test
▼ Preprocesado
Imputación de valores ausent
Exclusión de variables con va
Estandarización y escalado d
Binarización de las variables
Pipeline y ColumnTransformer
▼ Crear un modelo
Entrenamiento
Validación
Predicción
Error de test
▼ Hiperparámetros (tuning)
Grid search
Random grid search
Optimización bayesiana
Tuning del preprocesado
▼ Algoritmos
K-Nearest Neighbor (kNN)
Regresión lineal (Ridge y Las
Random Forest
Gradient Boosting Trees
▼ Stacking
Algoritmo Super Learner
Comparación
▼ Anexos
Anexo1: Métodos de validaci
Anexo2: Métricas
Bibliografía

Se procede a crear un *stacking* con los modelos *Ridge* y *RandomForest*, empleando en cada caso lo mejor hiperparámetros encontrados en los apartados anteriores.

In [65]:

```
from sklearn.linear_model import Ridge
from sklearn.linear_model import RidgeCV
from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import StackingRegressor

# Pipeline: preprocessado + modelos para el stacking
# =====
# Identificación de columnas numéricas y categóricas
numeric_cols = X_train.select_dtypes(include=['float64', 'int']).columns.to_list()
cat_cols = X_train.select_dtypes(include=['object', 'category']).columns.to_list()

# Transformaciones para las variables numéricas
numeric_transformer = Pipeline(
    steps=[('scaler', StandardScaler())]
)

# Transformaciones para las variables categóricas
categorical_transformer = Pipeline(
    steps=[('onehot', OneHotEncoder(handle_unknown='ignore'))]
)

preprocessor = ColumnTransformer(
    transformers=[
        ('numeric', numeric_transformer, numeric_cols),
        ('cat', categorical_transformer, cat_cols)
    ],
    remainder='passthrough'
)

# Se combinan los pasos de preprocessado y los modelos creando varios pipeline.
pipe_ridge = Pipeline([('preprocessing', preprocessor),
                      ('ridge', Ridge(alpha=3.4))])

pipe_rf = Pipeline([('preprocessing', preprocessor),
                    ('random_forest', RandomForestRegressor(
                        n_estimators = 1000,
                        max_features = 7,
                        max_depth = 20
                    ))])
])
```

In [66]:

```
# Definición y entrenamiento del StackingRegressor
# =====
estimators = [('ridge', pipe_ridge),
               ('random_forest', pipe_rf)]

stacking_regressor = StackingRegressor(estimators=estimators,
                                         final_estimator=RidgeCV())
# Se asigna el resultado a _ para que no se imprima por pantalla
_ = stacking_regressor.fit(X = X_train, y = y_train)
```

In [67]:

```
# Error de test del stacking
# =====
modelo_final = stacking_regressor
predicciones = modelo_final.predict(X = X_test)
rmse_stacking = mean_squared_error(
    y_true = y_test,
    y_pred = predicciones,
    squared = False
)
print(f"El error (rmse) de test es: {rmse_stacking}")
```

El error (rmse) de test es: 60541.046284245975

Comparación

Se compara el error de test de todos los modelos entrenados.

Tabla de contenidos

- Scikit-learn
- Introducción
- Librerías
- Datos
- Analís exploratorio
 - Tipo de cada columna
 - Número de observaciones y v
 - Variable respuesta
 - Variables numéricas
 - Correlación variables numéric
 - Variables cualitativas
- División train y test
- Preprocesado
 - Imputación de valores ausent
 - Exclusión de variables con va
 - Estandarización y escalado d
 - Binarización de las variables c
 - Pipeline y ColumnTransformer
- Crear un modelo
 - Entrenamiento
 - Validación
 - Predicción
 - Error de test
- Hiperparámetros (tuning)
 - Grid search
 - Random grid search
 - Optimización bayesiana
 - Tuning del preprocesado
- Algoritmos
 - K-Nearest Neighbor (kNN)
 - Regresión lineal (Ridge y Las
 - Random Forest
 - Gradient Boosting Trees
- Stacking
 - Algoritmo Super Learner
 - Comparación
- Anexos
 - Anexo1: Métodos de validaci
 - Anexo2: Métricas
- Bibliografía

```
In [68]: error_modelos = pd.DataFrame({
    'modelo': ['knn', 'lm', 'random forest', 'gradient boosting',
               'stacking'],
    'rmse': [rmse_knn, rmse_lm, rmse_rf, rmse_gbm, rmse_stacking]
})
error_modelos = error_modelos.sort_values('rmse', ascending=False)

fig, ax = plt.subplots(figsize=(6, 3.84))
ax.hlines(error_modelos.modelo, xmin=0, xmax=error_modelos.rmse)
ax.plot(error_modelos.rmse, error_modelos.modelo, "o", color='black')
ax.tick_params(axis='y', which='major', labelsize=12)
ax.set_title('Comparación de error de test modelos'),
ax.set_xlabel('Test rmse');
```

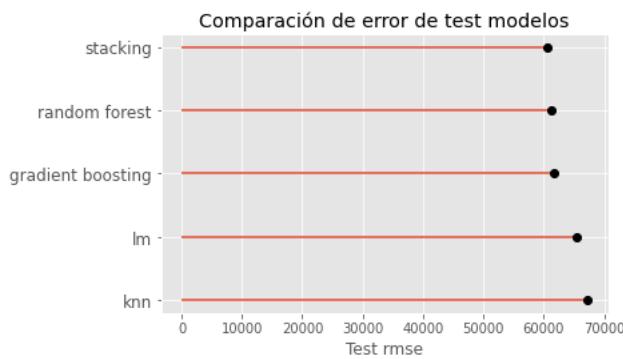
**Anexos**

Tabla de contenidos ↗

Scikit-learn
Introducción
Librerías
Datos
▼ Análisis exploratorio
Tipo de cada columna
Número de observaciones y v
Variable respuesta
Variables numéricas
Correlación variables numéric
Variables cualitativas
División train y test
▼ Preprocesado
Imputación de valores ausent
Exclusión de variables con va
Estandarización y escalado d
Binarización de las variables
Pipeline y ColumnTransformer
▼ Crear un modelo
Entrenamiento
Validación
Predicción
Error de test
▼ Hiperparámetros (tuning)
Grid search
Random grid search
Optimización bayesiana
Tuning del preprocesado
▼ Algoritmos
K-Nearest Neighbor (kNN)
Regresión lineal (Ridge y Las
Random Forest
Gradient Boosting Trees
▼ Stacking
Algoritmo Super Learner
Comparación
▼ Anexos
Anexo1: Métodos de validaci
Anexo2: Métricas
Bibliografía

Anexo1: Métodos de validación

Los métodos de validación, también conocidos como *resampling*, son estrategias que permiten estimar la capacidad predictiva de los modelos cuando se aplican a nuevas observaciones, haciendo uso únicamente de los datos de entrenamiento. La idea en la que se basan todos ellos es la siguiente: el modelo se ajusta empleando un subconjunto de observaciones del conjunto de entrenamiento y se evalúa (calcular una métrica que mida cómo de bueno es el modelo, por ejemplo, *accuracy*) con las observaciones restantes. Este proceso se repite múltiples veces y los resultados se agregan y promedian. Gracias a las repeticiones, se compensan las posibles desviaciones que pueden surgir por el reparto aleatorio de las observaciones. La diferencia entre métodos suele ser la forma en la que generan los subconjuntos de entrenamiento/validación.

K-Fold-Cross-Validation (CV)

Las observaciones de entrenamiento se reparten en *k folds* (conjuntos) del mismo tamaño. El modelo se ajusta a todas las observaciones excepto las del primer *fold* y se evalúa prediciendo las observaciones del *fold* que quedaron excluidos, obteniendo así la primera métrica. El proceso se repite *k* veces, excluyendo un *fold* distinto en cada iteración. Al final, se generan *k* valores de la métrica, que se agregan (normalmente con la media y la desviación típica) generando la estimación final de validación.

Leave-One-Out Cross-Validation (LOOCV)

LOOCV es un caso especial de *k-Fold-Cross-Validation* en el que el número *k* de *folds* es igual al número de observaciones disponibles en el conjunto de entrenamiento. El modelo se ajusta cada vez con todas las observaciones excepto una, que se emplea para evaluar el modelo. Este método supone un coste computacional muy elevado, ya que el modelo se ajusta tantas veces como observaciones de entrenamiento, por lo que, en la práctica, suele compensar emplearlo.

Repeated k-Fold-Cross-Validation (repeated CV)

Es exactamente igual al método *k-Fold-Cross-Validation* pero repitiendo el proceso completo *n* veces. Por ejemplo, *10-Fold-Cross-Validation* con 5 repeticiones implica a un total de 50 iteraciones ajuste-validación, pero no equivale a *50-Fold-Cross-Validation*.

Leave-Group-Out Cross-Validation (LGOCV)

LGOCV, también conocido como *repeated train/test splits* o *Monte Carlo Cross-Validation*, consiste simplemente en generar múltiples divisiones aleatorias entre entrenamiento-test (solo dos conjuntos por repetición). La proporción de observaciones que va a cada conjunto se determina de antemano, 80%-20% suele dar buenos resultados. Este método, aunque más simple de implementar que CV, requiere de muchas repeticiones (>50) para obtener estimaciones estables.

Bootstrapping

Una muestra *bootstrap* es una muestra obtenida a partir de la muestra original por muestreo aleatorio con reposición y del mismo tamaño que la muestra original. Muestreo aleatorio con reposición (*resampling with replacement*) significa que, después de que una observación sea extraída, se vuelve a poner a disposición para las siguientes extracciones. Como resultado de este tipo de muestreo, algunas observaciones aparecerán múltiples veces en la muestra *bootstrap* y otras ninguna. Las observaciones no seleccionadas reciben el nombre de *out-of-bag* (OOB). En cada iteración de *bootstrapping* se genera una nueva muestra *bootstrap*, se ajusta el modelo con ella y se evalúa las observaciones *out-of-bag*.

1. Obtener una nueva muestra del mismo tamaño que la muestra original mediante muestreo aleatorio con reposición.
2. Ajustar el modelo empleando la nueva muestra generada en el paso 1.

Tabla de contenidos

Scikit-learn
Introducción
Librerías
Datos
▼ Análisis exploratorio
Tipo de cada columna
Número de observaciones y v
Variable respuesta
Variables numéricas
Correlación variables numéric
Variables cualitativas
División train y test
▼ Preprocesado
Imputación de valores ausent
Exclusión de variables con va
Estandarización y escalado d
Binarización de las variables c
Pipeline y ColumnTransformer
▼ Crear un modelo
Entrenamiento
Validación
Predicción
Error de test
▼ Hiperparámetros (tuning)
Grid search
Random grid search
Optimización bayesiana
Tuning del preprocesado
▼ Algoritmos
K-Nearest Neighbor (kNN)
Regresión lineal (Ridge y Las
Random Forest
Gradient Boosting Trees
▼ Stacking
Algoritmo Super Learner
Comparación
▼ Anexos
Anexo1: Métodos de validaci
Anexo2: Métricas
Bibliografía

3. Calcular el error del modelo empleando aquellas observaciones de la muestra original que no se han incluido en la nueva muestra. A este error se le conoce como error de validación.
4. Repetir el proceso n veces y calcular la media de los n errores de validación.
5. Finalmente, y tras las n repeticiones, se ajusta el modelo final empleando todas las observaciones de entrenamiento originales.

La naturaleza del proceso de *bootstrapping* genera cierto bias en las estimaciones que puede ser problemático cuando el conjunto de entrenamiento es pequeño. Existen ciertas modificaciones del algoritmo original para corregir este problema, algunos de ellos son: *632 method* y *632+ method*.

No existe un método de validación que supere al resto en todos los escenarios, la elección debe basarse en varios factores.

- Si el tamaño de la muestra es pequeño, se recomienda emplear *repeated k-Fold-Cross-Validation*, ya que consigue un buen equilibrio bias-varianza y, dado que no son muchas observaciones, el coste computacional es excesivo.
- Si el objetivo principal es comparar modelos más que obtener una estimación precisa de las métricas, se recomienda *bootstrapping* ya que tiene menos varianza.
- Si el tamaño muestral es muy grande, la diferencia entre métodos se reduce y toma más importancia la eficiencia computacional. En estos casos, *10-Fold-Cross-Validation* simple es suficiente.

Puede encontrarse un estudio comparativo de los diferentes métodos en [Comparing Different Species of Cross Validation](http://www.appliedpredictivemodeling.com/blog/2014/11/27/vpuig01pqbk1mi72b8lcl3ij5hj2qm) (<http://www.appliedpredictivemodeling.com/blog/2014/11/27/vpuig01pqbk1mi72b8lcl3ij5hj2qm>).

Tabla de contenidos ↗

Scikit-learn
Introducción
Librerías
Datos
▼ Análisis exploratorio
Tipo de cada columna
Número de observaciones y v
Variable respuesta
Variables numéricas
Correlación variables numéric
Variables cualitativas
División train y test
▼ Preprocesado
Imputación de valores ausent
Exclusión de variables con va
Estandarización y escalado d
Binarización de las variables c
Pipeline y ColumnTransformer
▼ Crear un modelo
Entrenamiento
Validación
Predicción
Error de test
▼ Hiperparámetros (tuning)
Grid search
Random grid search
Optimización bayesiana
Tuning del preprocesado
▼ Algoritmos
K-Nearest Neighbor (kNN)
Regresión lineal (Ridge y Las
Random Forest
Gradient Boosting Trees
▼ Stacking
Algoritmo Super Learner
Comparación
▼ Anexos
Anexo1: Métodos de validació
Anexo2: Métricas
Bibliografía

Anexo2: Métricas

Existe una gran variedad de métricas que permiten evaluar cómo de bueno es un algoritmo realizando predicciones. La idoneidad de cada una depende completamente del problema en cuestión, y su correcta elección dependerá de cómo de bien entienda el analista el problema al que se enfrenta. A continuación, se describen algunas de las más utilizadas.

Accuracy y Kappa

Estas dos métricas son las más empleadas en problemas de clasificación binaria y multiclas. *Accuracy* es el porcentaje de observaciones correctamente clasificadas respecto al total de predicciones. *Kappa* o *Cohen's Kappa* es el valor de *accuracy* normalizado respecto del porcentaje de acierto esperado por azar. A diferencia de *accuracy*, el rango de valores puede ser [0, 1], el de *kappa* es [-1, 1]. En problemas con clases desbalanceadas, donde el grupo mayoritario supera por mucho a los otros, *Kappa* es más útil porque evita caer en la ilusión de creer que un modelo es bueno cuando realmente solo supera por poco lo esperado por azar.

MSE, RMSE, MAE

Estas son las métricas más empleadas en problemas de regresión.

MSE (Mean Squared Error) es la media de los errores elevados al cuadrado. Suele ser muy buen indicativo de cómo funciona el modelo en general, pero tiene la desventaja de estar en unidades cuadradas. Para mejorar la interpretación, suele emplearse *RMSE (Root Mean Squared Error)*, que es la raíz cuadrada del *MSE* y por lo tanto sus unidades son las mismas que la variable respuesta.

MAE (Mean Absolute Error) es la media de los errores en valor absoluto. La diferencia respecto a *MSE* es que, en último, eleva al cuadrado los errores, lo que significa que penaliza mucho más las desviaciones grandes. A modo general, *MSE* favorece modelos que se comportan aproximadamente igual de bien en todas las observaciones, mientras que *MAE* favorece modelos que predicen muy bien la gran mayoría de observaciones aunque en unas pocas se equivoque por mucho.

Tabla de contenidos ↗

Scikit-learn
Introducción
Librerías
Datos
▼ Análisis exploratorio
Tipo de cada columna
Número de observaciones y v
Variable respuesta
Variables numéricas
Correlación variables numéric
Variables cualitativas
División train y test
▼ Preprocesado
Imputación de valores ausent
Exclusión de variables con va
Estandarización y escalado d
Binarización de las variables c
Pipeline y ColumnTransformer
▼ Crear un modelo
Entrenamiento
Validación
Predicción
Error de test
▼ Hiperparámetros (tuning)
Grid search
Random grid search
Optimización bayesiana
Tuning del preprocesado
▼ Algoritmos
K-Nearest Neighbor (kNN)
Regresión lineal (Ridge y Las
Random Forest
Gradient Boosting Trees
▼ Stacking
Algoritmo Super Learner
Comparación
▼ Anexos
Anexo1: Métodos de validació
Anexo2: Métricas
Bibliografía

Bibliografía

Introduction to Machine Learning with Python: A Guide for Data Scientists |
https://www.amazon.es/gp/product/1449369413/ref=as_li_qf_asin_il_tl?ie=UTF8&tag=cienciadedato-21&creative=24630&linkCode=as2&creativeASIN=1449369413&linkId=e071892d9e2c458e8144303901ea9580

Introduction to Statistical Learning, Gareth James, Daniela Witten, Trevor Hastie and Robert Tibshirani |
https://www.amazon.es/gp/product/1461471370/ref=as_li_qf_asin_il_tl?ie=UTF8&camp=3638&creative=24630&creativeASIN=1461471370&linkCode=as2&tag=cienciadedato-21&linkId=64752a80078f4f017e81874b8cb355b7

Applied Predictive Modeling by Max Kuhn and Kjell Johnson |
https://www.amazon.es/gp/product/1461468485/ref=as_li_qf_asin_il_tl?ie=UTF8&tag=cienciadedato-21&creative=24630&linkCode=as2&creativeASIN=1461468485&linkId=f014c46d1f6c670f31a2b4cff724e190

The Elements of Statistical Learning by T.Hastie, R.Tibshirani, J.Friedman |
https://www.amazon.es/gp/product/B00M0R6ZJG/ref=as_li_qf_asin_il_tl?ie=UTF8&tag=cienciadedato-21&creative=24630&linkCode=as2&creativeASIN=B00M0R6ZJG&linkId=7d710cfcd19d2d77cbf9eac37b41acf8

Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011.

API design for machine learning software: experiences from the scikit-learn project, Buitinck et al., 2013.

Uso de pipes y ColumnTransformer: <https://www.dataschool.io/encoding-categorical-features-in-python>

¿Cómo citar este documento?

Machine learning con Python y Scikit-learn by Joaquín Amat Rodrigo, available under a Attribution 4.0 International (CC BY 4.0) at https://www.cienciadedatos.net/documentos/py06_machine_learning_python_scikitlearn.html

¿Te ha gustado el artículo? Tu ayuda es importante

Mantener un sitio web tiene unos costes elevados, tu contribución me ayudará a seguir generando contenido divulgativo gratuito. ¡Muchísimas gracias! ☺



[\(http://creativecommons.org/licenses/by/4.0/\)](http://creativecommons.org/licenses/by/4.0/)

This work by Joaquín Amat Rodrigo is licensed under a [Creative Commons Attribution 4.0 International License](http://creativecommons.org/licenses/by/4.0/) (<http://creativecommons.org/licenses/by/4.0/>).