# PROYECTO DE MACHINE LEARNING

Vamos a trabajar en un proyecto de ejemplo de principio a fin.

Estas son las etapas por las que pasaremos:

* Definición del problema
* Recopilación y preparación de los datos
* Entrenamiento y evaluación cruzada del modelo (definir la medida de evaluación)
* Optimización del modelo (ajustar los hiperparámetros) y evaluación en el conjunto de prueba (si los resultados no son buenos tendremos que volver a replantear nuestro modelo).
* Presentación de la solución

## DEFINICIÓN DEL PROBLEMA

Tenemos que crear un modelo para predecir los precios de las casas de California (los datos son de 1990, un poco desfasado, pero el conjunto de datos tiene muchas cualidades para el aprendizaje y es muy típico para aprender machine learning).

Vamos a usar el censo de California cuyos datos incluyen métricas como la población, los ingresos medianos y los precios medianos da las casas para cada distrito.

El modelo debería aprender a partir de estos datos y ser capaz de predecir el precio mediano de las casas en cualquier distrito si se le dan todas las demás métricas.

Gráfico, Mapa

Descripción generada automáticamente

Precios de las casas de California

Lo primero es saber cuál es exactamente el objetivo empresarial: en este caso la salida del modelo (una predicción del precio mediano de las casas de un distrito) se introducirá en otro sistema de machine learning, junto con muchas otras variables para determinar si merece la pena invertir en un área determinada.

Diagrama

Descripción generada automáticamente

Lo segundo es saber cómo es la solución actual (si la hay). A menudo, la situación actual te dará perspectivas sobre cómo resolver el problema. En este caso, los precios de las casas del distrito los estiman a mano un grupo de expertos. Esto sale caro y lleva mucho tiempo, además, las estimaciones no son muy buenas, a menudo descubren que sus cálculos se desvían más de un 30%.

Ya estamos listos para empezar a diseñar el sistema, pero primero hay que determinar qué tipo de entrenamiento necesitará el modelo: **¿Es una tarea de aprendizaje supervisado, no supervisado o por refuerzo? ¿Es una tarea de clasificación, de regresión…?:**

* Está claro que se trata de una tarea de aprendizaje supervisado típica, puesto que el modelo puede entrenarse con ejemplos etiquetados (cada instancia viene con la salida esperada, es decir, el precio mediano de las casas del distrito).
* Es una tarea de regresión típica, ya que pedirá al modelo que prediga un valor. De manera más específica, se trata de un problema de regresión múltiple, puesto que el sistema utilizará múltiples características para hacer una predicción (la población del distrito, los ingresos medianos, etc.). También es un problema de regresión univariante, porque solo se quiere predecir un valor para cada distrito, el precio mediano de las casas.

## RECOPILACIÓN Y PREPARACIÓN DE LOS DATOS

### OBTENER LOS DATOS

En entornos típicos, los datos estarán disponibles en una base de datos relacional o algún otro almacén de datos común y se extenderían por múltiples tablas/documentos/archivos. Para acceder a ellos, se necesitan credenciales y autorizaciones de acceso.

Además hay que comprobar las restricciones legales, como campos privados que nunca deberían copiarse en almacenes de datos no seguros.

En este proyecto los datos están en un archivo CSV llamado housing.csv.

import pandas as pd

def load\_housing\_data():

try:

return pd.read\_csv("datos/housing.csv")

except:

print("Fichero no encontrado")

housing = load\_housing\_data()

### EXPLORAR LOS DATOS

Vamos a empezar por fijarnos en las cinco filas superiores de tos datos utilizando el método **head()** de Dataframe.

Tabla

Descripción generada automáticamente

Cada fila representa un distrito.

Hay 10 atributos (no se muestran todos en la captura de pantalla): longitude, latitude, housing\_median\_age, total\_rooms, total\_bedrooms, population, households, median\_income, ocean\_proximity y median\_house\_value.

El método **info()** es útil para obtener una descripción rápida de los datos, en particular el número total de filas, el tipo de cada atributo y el número de valores no nulos:

housing.info()

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 20640 entries, 0 to 20639

Data columns (total 10 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 longitude 20640 non-null float64

1 latitude 20640 non-null float64

2 housing\_median\_age 20640 non-null float64

3 total\_rooms 20640 non-null float64

4 total\_bedrooms 20433 non-null float64

5 population 20640 non-null float64

6 households 20640 non-null float64

7 median\_income 20640 non-null float64

8 median\_house\_value 20640 non-null float64

9 ocean\_proximity 20640 non-null object

dtypes: float64(9), object(1)

memory usage: 1.6+ MB

Podemos observar que:

* Hay 20.640 instancias en el conjunto de datos, lo que significa que es bastante pequeño según los estándares del machine learning, pero es perfecto para empezar.
* El atributo total\_bedrooms solo tiene 20.433 valores no nulos, lo que significa que a 207 distritos les falta esta característica.
* Todos los atributos son numéricos, excepto ocean\_proximity. Su tipo es object, así que podría albergar cualquier tipo de objeto de Python, pero, como hemos cargado estos datos desde un archivo CSV, tiene que ser un atributo de texto.
* Los valores en la columna ocean\_proximity son repetitivos, lo que significa que se trata, probablemente, de un atributo categórico. Podemos averiguar qué categorías existen y cuántos distritos pertenecen a cada categoría utilizando el método **value\_counts()**:

housing["ocean\_proximity"].value\_counts()

<1H OCEAN 9136 I

NLAND 6551

NEAR OCEAN 2658

NEAR BAY 2290

ISLAND 5

Name: ocean\_proximity, dtype: int64

El método **describe()** muestra un resumen de los atributos numéricos.

Tabla

Descripción generada automáticamente

Los valores nulos se ignoran (así, por ejemplo, en count de total\_bedrooms tenemos 20.433, no 20.640).

La fila std muestra la desviación estándar, que mide cómo de dispersos están los valores.

Las filas 25%, 50% y 75% muestran los percentiles correspondientes: un percentil indica el valor por debajo del cual queda un porcentaje de observaciones dado en un grupo de observaciones. Por ejemplo, el 25% de los distritos tienen housing\_median\_age inferior a 18, mientras que el 50% están por debajo de 29 y el 75% están por debajo de 37.

Otra manera rápida de hacerse una idea del tipo de datos que estamos manejando es trazar un histograma para cada atributo numérico. Un histograma muestra el número de instancias (en el eje vertical) que tiene un atributo dado (en el eje horizontal). Podemos trazar esto de atributo en atributo o podemos llamar al método **hist()** en el conjunto de datos completo y eso dibujará un histograma para cada atributo numérico:

housing.hist(bins=50, figsize=(12, 8))

plt.show()

Interfaz de usuario gráfica, Gráfico

Descripción generada automáticamente

Podemos observar varias cosas:

* El atributo de los ingresos medianos no parece estar expresado en dólares estadounidenses. Después de hablar con el equipo que recopiló los datos, se sabe que el rango está limitado a 15 (en realidad, 15,0001) para ingresos medianos elevados y a 0,5 (en realidad, 0,4999) para ingresos medianos bajos. Los números representan, de forma aproximada, decenas de miles de dólares (por ejemplo, 3 en realidad significa 30.000 dólares). Trabajar con atributos preprocesados es habitual en el machine learning y no tiene por qué ser un problema, hay que entender cómo se han computado los datos.
* La edad mediana de la casa y el valor mediano de la casa también están limitados. Este último puede ser un problema serio, puesto que es el atributo objetivo. Hay que hablar con el cliente para ver si eso es un problema o no. Si se necesitan predicciones precisas incluso más allá de 500.000 dólares, entonces hay dos opciones:
  + Recopilar etiquetas adecuadas para los distritos cuyas etiquetas estaban limitadas.
  + Eliminar esos distritos del conjunto de entrenamiento (y también del conjunto de prueba, ya que el sistema no debería evaluarse mal si predice valores más allá de 500.000).
* Estos atributos tienen escalas muy diferentes. Hablaremos sobre esto más adelante, cuando exploremos el escalado de características.
* Muchos histogramas están sesgados a la derecha (se extienden mucho más hacia la derecha de la mediana que hacia la izquierda). Esto puede dificultar un poco la detección de patrones por parte de algunos algoritmos de machine learning. Probaremos a transformar estos atributos más adelante para tener distribuciones más simétricas y con forma más acampanada.

### DIVIDIR LOS DATOS

Antes de seguir explorando los datos, vamos a dividir los datos en dos conjuntos: entrenamiento y prueba.

Podemos dividir los datos manualmente con Python, pero Scikit-Learn (librería de Python que vamos a usar en machine learning) nos ofrece la función es **train\_test\_split()**:

* test\_size: indicamos el porcentaje de los datos de prueba (en vez de porcentaje se puede indicar el tamaño exacto de registros). En vez de test\_size se puede especificar train\_size.
* random\_state: permite establecer la semilla aleatoria del generador. Podemos pasar múltiples conjuntos de datos con un número idéntico de filas y los dividirá en los mismos índices (esto es muy útil, por ejemplo, si tenemos un dataframe aparte para las etiquetas).

train\_set, test\_set = train\_test\_split(housing, test\_size=0.2, random\_state=42)

Hasta ahora, hemos considerado métodos de muestreo puramente aleatorios. Por lo general, esto suele servir si el conjunto de datos es lo bastante grande, pero, si no lo es, corremos el riesgo de introducir un sesgo muestral significativo.

Cuando los empleados de una empresa de realización de encuestas deciden llamar a 1.000 personas para hacerles unas preguntas, no eligen a 1.000 personas al azar en la guía telefónica. Intentan asegurarse de que estas 1.000 personas son representativas de la población global, con respecto a las preguntas que quieren hacer. Si las personas que llevan a cabo la encuesta utilizasen solo muestreos aleatorios, los resultados de la encuesta tendrían bastantes probabilidades de estar distorsionados. Por ejemplo, el 51,1% de la población de EE.UU. son mujeres y el 48,9% son hombres, así que una encuesta bien realizada en EE.UU. debe mantener esa proporción en la muestra: 511 mujeres y 489 hombres (si las respuestas pueden variar según el género). Esto se llama **"muestreo estratificado"**, la población se divide en subgrupos homogéneos llamados "estratos" y en cada estrato se recoge como muestra el número adecuado de instancias para garantizar que el conjunto de prueba es representativo de la población total.

Supongamos que hemos hablado con algunos expertos que nos dicen que los ingresos medianos son un atributo muy importante para predecir los precios medianos de las casas. Deberíamos asegurarnos de que los conjuntos de datos son representativos de las diferentes categorías de ingresos en el conjunto de datos completo.

Puesto que los ingresos medianos son un atributo numérico continuo, primero hay que crear un atributo de categorías de ingresos.

Vamos a fijarnos con más detenimiento en el histograma de ingresos medianos:

Gráfico, Histograma

Descripción generada automáticamente

La mayoría de los valores de ingresos medianos se agrupan en torno al 1,5-6 (es decir, 15.000-60.000 dólares). Es importante tener una cantidad suficiente de instancias en cada estrato o, de lo contrario, la estimación de la importancia de un estrato podría distorsionarse.

El siguiente código utiliza la función **pd.cut()** para crear un atributo de categorías de ingresos con cinco categorías (etiquetadas del 1 al 5); la categoría 1 va de 0 a 1,5 (es decir, menos de 15.000 dólares), la categoría 2 va de 1,5 a 3, y así sucesivamente:

housing["income\_cat"] = pd.cut(housing["median\_income"],

bins=[0., 1.5, 3.0, 4.5, 6., np.inf], labels=[1, 2, 3, 4, 5])

Veamos como ha quedado la distribución de la nueva variable:

housing["income\_cat"].value\_counts().sort\_index().plot.bar(rot=0, grid=True)

plt.xlabel("Income category")

plt.ylabel("Number of districts")

plt.show()

Gráfico

Descripción generada automáticamente

Ahora ya podemos hacer un muestreo estratificado basado en la categoría de ingresos, simplemente tenemos que usar el argumento **stratify** en train\_test\_split:

strat\_train\_set, strat\_test\_set = train\_test\_split(

housing, test\_size=0.2, stratify=housing["income\_cat"], random\_state=42)

Vamos a ver si esto funciona como se esperaba.

Nos fijamos en las proporciones de la categoría de ingresos en el conjunto de prueba y en el conjunto de entrenamiento:

strat\_test\_set["income\_cat"].value\_counts() / len(strat\_test\_set)

3 0.350533

2 0.318798

4 0.176357

5 0.114341

1 0.039971

Name: income\_cat, dtype: float64

strat\_train\_set["income\_cat"].value\_counts() / len(strat\_train\_set)

3 0.350594

2 0.318859

4 0.176296

5 0.114462

1 0.039789

Name: income\_cat, dtype: float64

No vamos a volver a usar la columna income\_cat, así que podemos olvidarla y revertir los datos a su estado original:

for set\_ in (strat\_train\_set, strat\_test\_set):

set\_.drop("income\_cat", axis=1, inplace=True)

### EXPLORAR LOS DATOS

Vamos a profundizar un poco más en la exploración de los datos, pero como ya hemos apartado el conjunto de prueba, vamos a trabajar sólo con el conjunto de entrenamiento.

Vamos a experimentar con varias transformaciones del conjunto de entrenamiento completo por lo tanto debemos hacer una copia del original para poder volver a él después:

housing = strat\_train\_set.copy()

Puesto que el conjunto de datos incluye información geográfica (latitud y longitud), es buena idea crear un diagrama de dispersión de todos los distritos para visualizar los datos:

housing.plot(kind="scatter", x="longitude", y="latitude", grid=True)

plt.show()

Gráfico, Gráfico de dispersión

Descripción generada automáticamente

Vale, esto parece California, pero, aparte de eso, es difícil distinguir un patrón particular. Configurar la opción alpha como 0.2 hace que sea mucho más fácil visualizar los lugares donde hay una densidad alta de puntos:

housing.plot(kind="scatter", x="longitude", y="latitude", grid=True, alpha=0.2)

plt.show()

Gráfico, Gráfico de dispersión

Descripción generada automáticamente

Ahora podemos ver con claridad las áreas de alta densidad, concretamente el Área de la Bahía y los alrededores de Los Ángeles y San Diego, además de una línea larga de áreas de densidad bastante alta en el Valle Central (en particular en Sacramento y Fresno).

Ahora vamos a hacer que el radio de cada círculo represente la población del distrito (opción s) y el color represente el precio (opción c):

housing.plot(kind="scatter", x="longitude", y="latitude", grid=True,

s=housing["population"] / 100, label="population",

c="median\_house\_value", cmap="jet", colorbar=True,

legend=True)

plt.show()

Gráfico, Gráfico de dispersión

Descripción generada automáticamente

Esta imagen nos indica que los precios de las casas están muy relacionados con la ubicación (por ejemplo, cerca del océano) y con la densidad de población, como probablemente ya sabías.

#### BUSCAR CORRELACIONES

Puesto que el conjunto de datos no es demasiado grande, podemos calcular con facilidad el coeficiente de correlación estándar (también llamado coeficiente r de Pearson, el cual mide correlaciones lineales) entre cada par de atributos numéricos utilizando el método **corr()**:

corr\_matrix = housing.corr(numeric\_only=True)

Vamos a fijarnos en cuánto se correlaciona cado atributo con el valor mediano de las casas:

corr\_matrix["median\_house\_value"].sort\_values(ascending=False)

median\_house\_value 1.000000

median\_income 0.688380

total\_rooms 0.137455

housing\_median\_age 0.102175

households 0.071426

total\_bedrooms 0.054635

population -0.020153

longitude -0.050859

latitude -0.139584

Name: median\_house\_value, dtype: float64

El coeficiente de correlación va de -1 a 1. Cuando se acerca a 1, significa que hay una correlación positiva fuerte; por ejemplo, el valor mediano de las casas tiende a subir cuando los ingresos medianos suben. Cuando el coeficiente se acerca a -1, significa que hay una correlación negativa fuerte; podemos ver una pequeña correlación negativa entre la latitud y el valor mediano de la casa (es decir, los precios tienen una ligera tendencia a bajar cuando avanzamos hacia el norte). Por último, los coeficientes que se acercan a 0 significan que no hay una correlación lineal.

Otra forma de ver la matriz de correlación es mediante una gráfica:

corr\_matrix.style.background\_gradient()

**Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente**

Parece que el atributo más prometedor para predecir el valor mediano de las casas es el de los ingresos medianos, así que nos centramos en su diagrama de dispersión:

housing.plot(kind="scatter", x="median\_income", y="median\_house\_value", alpha=0.1, grid=True)

plt.show()

Gráfico, Gráfico de dispersión

Descripción generada automáticamente

Este gráfico revela varias cosas:

* La correlación es bastante fuerte: se puede ver con claridad la tendencia ascendente y los puntos no están demasiado dispersos.
* El límite de precio que habíamos observado antes es claramente visible como una línea horizontal en 500.000 dólares. Pero el gráfico también revela atrás líneas rectas menos evidentes: una línea horizontal alrededor de los 450.000 dólares, otra alrededor de 350.000 dólares, quizá otra alrededor de 280.000 dólares y algunas más por debajo de eso. Convendría intentar eliminar los distritos correspondientes para evitar que los algoritmos aprendan a reproducir estas singularidades de los datos.

### PREPARAR LOS DATOS

Es el momento de preparar los datos para los algoritmos de machine learning.

#### COMBINACIONES DE ATRIBUTOS

Podemos probar combinaciones de atributos.

Por ejemplo, el número total de habitaciones en un distrito no resulta muy útil si no sabemos cuántas casas hay. Lo que de verdad nos interesa es el número de habitaciones por casa.

De manera similar, el número total de dormitorios en sí no es muy útil: conviene compararlo con el número de habitaciones.

También el número de personas por casa parece una combinación de atributos interesante.

Estos nuevos atributos se crean de la siguiente manera:

housing["rooms\_per\_house"] = housing["total\_rooms"] / housing["households"]

housing["bedrooms\_ratio"] = housing["total\_bedrooms"] / housing["total\_rooms"]

housing["people\_per\_house"] = housing["population"] / housing["households"]

Y ahora vamos a volver a fijarnos en la matriz de correlaciones:

corr\_matrix = housing.corr(numeric\_only=True)

corr\_matrix["median\_house\_value"].sort\_values(ascending=False)

median\_house\_value 1.000000

median\_income 0.688380

rooms\_per\_house 0.143663

total\_rooms 0.137455

housing\_median\_age 0.102175

households 0.071426

total\_bedrooms 0.054635

population -0.020153

people\_per\_house -0.038224

longitude -0.050859

latitude -0.139584

bedrooms\_ratio -0.256397

Name: median\_house\_value, dtype: float64

No está mal, el nuevo atributo bedrooms\_ratio está más correlacionado con el valor mediano de las casas que el número total de habitaciones o dormitorios. Parece que las casas con una proporción dormitorios/habitaciones más baja tienden a ser más caras.

El número de habitaciones por casa también es más informativo que el número total de habitaciones en un distrito; es evidente que cuanto más grandes sean las casas, más caras serán.

#### SEPARAR CARACTERÍSTICAS Y ETIQUETAS

housing = strat\_train\_set.drop("median\_house\_value", axis=1)

housing\_labels = strat\_train\_set["median\_house\_value"].copy()

#### LIMPIAR DATOS

La mayoría de los algoritmos de machine learning no pueden funcionar si faltan valores en alguno de los atributos.

Antes hemos visto que al atributo total\_bedrooms le faltan algunos valores. Tenemos tres opciones para solucionarlo:

1. Deshacernos de los distritos correspondientes.
2. Deshacernos de todo el atributo.
3. Establecer algún valor para esos valores que faltan (cero, la media, la mediana, etc.). Esto se denomina "**imputación**".

Esto puede hacerse con facilidad utilizando los métodos **dropna()**, **drop()** y **fillna()** de Pandas:

housing.dropna(subset=["total\_bedrooms"], inplace=True) # option 1

housing.drop("total\_bedrooms", axis=1) # option 2

median = housing["total\_bedrooms"].median() # option 3

housing["total\_bedrooms"].fillna(median, inplace=True)

Vamos a utilizar la opción 3, puesto que es la menos destructiva, pero, en vez del código anterior, vamos a usar una clase de Scikit-Learn: **SimpleImputer**:

imputer = SimpleImputer(strategy="median")

Hay más estrategias a parte de median: most\_frequent, constant…

Puesto que la mediana solo puede calcularse en atributos numéricos, tenemos que crear una copia de los datos solo con los atributos numéricos (eso excluirá el atributo de texto ocean\_proximity):

housing\_num = housing.select\_dtypes(include=[np.number])

Ahora podemos ajustar la instancia imputer a los datos de entrenamiento utilizando el método **fit()**

imputer.fit(housing\_num)

Ahora podemos utilizar esta instancia imputer "entrenada" para transformar los datos de entrenamiento sustituyendo valores que faltan por las medianas aprendidas:

X = imputer.transform(housing\_num)

La salida de imputer.**transform**(housing\_num) es una matriz NumPy y por lo tanto X no tiene ni nombres de columnas ni índice. Por suerte, no es demasiado difícil envolver X en un dataframe y recuperar los nombres de columnas y el índice desde housing\_num:

housing\_tr = pd.DataFrame(X, columns=housing\_num.columns, index=housing\_num.index)

#### ATRIBUTOS DE TEXTO Y CATEGÓRICOS

Hasta ahora, solo hemos tratado con atributos numéricos, pero los datos también pueden contener atributos de texto. En este conjunto de datos, solo hay uno: ocean\_proximity.

Echemos un vistazo a su valor para las primeras instancias:

housing\_cat = housing[["ocean\_proximity"]]

housing\_cat.head(8)

ocean\_proximity

13096 NEAR BAY

14973 <1H OCEAN

3785 INLAND

14689 INLAND

20507 NEAR OCEAN

1286 INLAND

18078 <1H OCEAN

4396N EAR BAY

No se trata de un texto arbitrario: hay un número limitado de valores posibles, cada uno de las cuales representa una categoría. Por tanto, este atributo es categórico.

Scikit-Learn ofrece una clase **OneHotEncoder** para convertir valores categóricos en vectores one-hot. Los vectores one-hot son una forma de representar variables categóricas como vectores numéricos.Un vector one-hot para una variable categórica tiene una longitud igual al número de categorías (o clases) posibles. El vector se compone completamente de 0s excepto por un 1 en la posición que representa la categoría a la que pertenece el dato.

cat\_encoder = OneHotEncoder()

housing\_cat\_1hot = cat\_encoder.fit\_transform(housing\_cat)

Por defecto, la salida de OneHotEncoder es una matriz dispersa SciPy, en vez de una matriz NumPy (a no ser que añadas el parámetro sparse=False)

Una matriz dispersa es una representación muy eficiente para matrices que contienen sobre todo ceros. En realidad, a nivel interno solo almacena valores distintos de cero y sus posiciones.

Si queremos convertir una matriz dispersa en una matriz NumPy (densa), tenemos el método toarray():

housing\_cat\_1hot.toarray()

array([[0., 0., 0., 1., 0.],

[1., 0., 0., 0., 0.],

[0., 1., 0., 0., 0.],

...,

[1., 0., 0., 0., 0.],

[0., 0., 0., 0., 1.]])

Podemos obtener la lista de categorías utilizando la variable de instancia **categories\_** del codificador:

cat\_encoder.categories\_

[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'], dtype=object)]

La salida de one-hot no es un dataframe pero podemos fácilmente envolverlo:

df\_output = pd.DataFrame(cat\_encoder.transform(housing\_cat).toarray(),

columns=cat\_encoder.get\_feature\_names\_out(),

index=housing\_cat.index)

df\_output

#### ESCALADO DE CARACTERÍSTICAS Y TRANSFORMACIÓN

Una de las transformaciones más importantes que hay que aplicar a los datos es el escalado de características. Salvo algunas excepciones, los algoritmos de machine learning no tienen un buen rendimiento cuando los atributos numéricos de entrada tienen escalas muy diferentes. Eso es lo que ocurre con los datos de las casas: el número total de habitaciones va de 6 a 39.320, mientras que los ingresos medianos solo van de 0 a 15. Sin ningún escalado, la mayoría de los modelos se inclinarán por ignorar los ingresos medianos y centrarse más en el número de habitaciones. Hay dos formas habituales de conseguir que todos los atributos tengan la misma escala: el **escalado mín./máx.** y la **estandarización**.

Como con todos los estimadores, es importante ajustar los escaladores solo a los datos de entrenamiento: nunca se debe utilizar **fit()** o **fit\_transform()** para nada que no sea el conjunto de entrenamiento. Una vez que un escalador está entrenado (fit), podemos utilizarlo para aplicarlo (**transform**) a cualquier otro conjunto.

El **escalado mín./máx.** (al que mucha gente Ilama "**normalización**") es el más sencillo: para cada atributo, los valores se trasladan y vuelven a escalarse para acabar en un rango entre 0 y 1. Esto se hace restando el valor mínimo y dividiendo por la diferencia entre el mínimo y el máximo. Scikit-Learn tiene un transformador llamado **MinMaxScaler** para realizar este escalado.

min\_max\_scaler = MinMaxScaler()

housing\_num\_min\_max\_scaled = min\_max\_scaler.fit\_transform(housing\_num)

La **estandarización** es diferente: primero se resta el valor medio (de manera que los valores normalizados siempre tienen una media de cero) y luego se divide el resultado por la desviación estándar (así que los valores normalizados tienen una desviación estándar igual a 1).

()

A diferencia del escalado min./máx., la estandarización no limita los valores a un rango especifico y se ve mucho menos afectada por los valores atípicos.

Scikit-Learn tiene un transformador llamado **StandardScaler** para la estandarización:

std\_scaler = StandardScaler()

housing\_num\_std\_scaled = std\_scaler.fit\_transform(housing\_num)

Cuando la distribución de una característica es de cola pesada (alargada), deberíamos transformar característica para **reducir la cola pesada** y, si es posible, hacer que la distribución sea más o menos simétrica.

Por ejemplo, una manera común de hacer esto para características positivas con una cola pesada hacia la derecha es sustituir la característica por su raíz cuadrada (o elevar la característica a una potencia entre 0 y 1). Si la cola es muy pesada, puede que resulte de ayuda sustituir la característica por su logaritmo.

Por ejemplo, en la siguiente figura podemos ver que la característica population queda mejor cuando se calcula su logaritmo: se acerca mucho a una distribución gaussiana (con forma de campana).

Gráfico

Descripción generada automáticamente

#### TRANSFORMADORES PERSONALIZADOS

Aunque Scikit-Learn ofrece muchos transformadores útiles, tendremos que escribir algunos propios para transformaciones personalizadas, las operaciones de limpieza personalizadas o la combinación de atributos específicos.

Para transformaciones que no requieren ningún entrenamiento, podemos escribir sin más una función que tome una matriz como entrada y genere como salida la matriz transformada (**FunctionTransformer**). Por ejemplo, como ya hemos visto, a menudo es buena idea transformar características con distribuciones de cola pesada sustituyéndolas por su logaritmo. Vamos a crear un transformador de logaritmos y aplicarlo a la característica population:

log\_transformer = FunctionTransformer(np.log)

log\_pop = log\_transformer.transform(housing[["population"]])

Los transformadores personalizados también son útiles para combinar características. Por ejemplo, aquí tenemos un FunctionTransformer que calcula la relación entre las características de entrada 0 y 1:

ratio\_transformer = FunctionTransformer(lambda X: X[:, [0]] / X[:, [1]])

ratio\_transformer.transform(np.array([[1., 2.], [3., 4.]]))

array([[0.5 ], [0.75]])

FunctionTransformer resulta muy útil, pero también podemos crear nuestro propio transformador. Para ello necesitamos una clase que herede de BaseEstimator y TransformerMixin y crear para ella los métodos fit y transform. El método fit\_transform() se consigue al heredar de TransformerMixin. Al heredar de BaseEstimator tenemos los métodos get\_params() y set\_params(). Esto será útil para el ajuste automático de hiperparámetros.

Por ejemplo, vamos a crear un estimador personalizado que actúa de forma muy similar a StandardScaler:

class StandardScalerClone(BaseEstimator, TransformerMixin):

def \_\_init\_\_(self, with\_mean=True):

self.with\_mean = with\_mean

def fit(self, X, y=None): # y is required even though we don't use it

X = check\_array(X) # checks that X is an array with finite float values

self.mean\_ = X.mean(axis=0)

self.scale\_ = X.std(axis=0)

self.n\_features\_in\_ = X.shape[1] # every estimator stores this in fit()

return self # always return self!

def transform(self, X):

check\_is\_fitted(self) # looks for learned attributes (with trailing \_)

X = check\_array(X)

assert self.n\_features\_in\_ == X.shape[1]

if self.with\_mean:

X = X - self.mean\_

return X / self.scale\_

Hay varias cosas en las que fijarse:

* Las pipelines de Scikit-Learn requieren que el método fit() tenga dos argumentos X e y, por lo que necesitamos el argumento y=None incluso aunque no usemos y.
* Todos los estimadores de Scikit-Learn establecen **n\_features\_in\_** en el método fit(), y garantizan que los datos pasados a transform() o predict() tienen este número de características.
* El método fit() debe devolver self.
* Esta implementación no está completa al 100%:
  + Todos los transformadores deberían proporcionar un método **get\_feature\_names\_out()**
  + Todos los transformadores deberían proporcionar un método **inverse\_transform()** si la transformación puede revertirse.

#### PIPELINES DE TRANSFORMACIÓN

Es el momento de preparar los datos para los algoritmos de machine learning, en vez de hacerlo a mano debemos escribir secuencias de transformaciones para este fin por varias razones:

* Reproducir estas transformaciones con facilidad en cualquier conjunto de datos (por ejemplo, la próxima vez que se tenga un conjunto de datos nuevo).
* Ir creando una biblioteca de funciones de transformación para reutilizar en futuros proyectos.
* Probar con facilidad varias transformaciones y ver qué combinación de transformaciones funciona mejor.

Scikit-Learn ofrece la clase Pipeline para ayudar con estas secuencias de transformaciones. Esta es una pequeña pipeline para los atributos numéricos, que primero imputará y, después, escalará las características de entrada:

num\_pipeline = Pipeline([

("impute", SimpleImputer(strategy="median")),

("standardize", StandardScaler()),

])

El constructor Pipeline toma una lista de pares nombre/estimador (tuplas de 2) que define una secuencia de pasos. Excepto el último, todos los demás deben tener un método fit\_transform().

Si no quieres poner nombre a los estimadores, puedes usar la función make\_pipeline() que crea una Pipeline usando los nombres de las clases de los estimadores, en minúsculas y sin guiones bajos (por ejemplo, "simpleimputer"). Si hay múltiples estimadores con el mismo nombre, se añade un índice a sus nombres como un apéndice (por ejemplo, "foo-1", "foo-2", etc.).

num\_pipeline = make\_pipeline(SimpleImputer(strategy="median"), StandardScaler())

En un cuaderno de Jupyter, si ejecutas sklearn.set\_config(display="diagram"), todos los estimadores de Scikit‑Learn se representarán como diagramas interactivos.

set\_config(display="diagram")

num\_pipeline

La pipeline expone los mismos métodos que el estimador final:

* Si llamamos al método fit() de la pipeline, se llama secuencialmente al método fit\_transform() en todos los estimadores, pasando la salida de cada estimador como parámetro de entrada para el siguiente hasta llegar al último, para el que llama al método fit().
* Si llamamos al método transform() de la pipeline, aplicará de manera secuencial todas las transformaciones a los datos.
* Si llamamos al método predict() de la pipeline (porque el último estimador es un predictor), se llama secuencialmente al método fit\_transform() en todos los estimadores, pasando la salida de cada estimador como parámetro de entrada para el siguiente hasta llegar al último, para el que llama al método predict().

Vamos a llamar al método fit\_transform() de la pipeline y a fijarnos en las dos primeras filas de la salida, redondeadas a dos decimales:

housing\_num\_prepared = num\_pipeline.fit\_transform(housing\_num)

housing\_num\_prepared[:2].round(2)

array([[-1.42, 1.01, 1.86, 0.31, 1.37, 0.14, 1.39, -0.94], [ 0.6 , -0.7 , 0.91, -0.31, -0.44, -0.69, -0.37, 1.17]])

Como hemos visto antes, si queremos recuperar un dataframe tenemos que envolver los datos devueltos por la pipeline:

df\_housing\_num\_prepared = pd.DataFrame(

housing\_num\_prepared, columns=num\_pipeline.get\_feature\_names\_out(),

index=housing\_num.index)

df\_housing\_num\_prepared.head(2)

Las pipelines soportan la indexación; por ejemplo, pipeline[1] devuelve el segundo estimador de la pipeline, y pipeline[:-1] devuelve un objeto Pipeline que contiene todos los estimadores, salvo el último. También podemos acceder a los estimadores a través del atributo steps, que es una lista de pares nombre/estimador, o a través del atributo de diccionario named\_steps, que asigna los nombres a los estimadores.

num\_pipeline.steps

num\_pipeline[1]

num\_pipeline[:-1]

num\_pipeline.named\_steps["simpleimputer"]

Hasta ahora, hemos tratado las columnas categóricas y las numéricas por separado. Sería más cómodo tener un solo transformador capaz de manejar todas las columnas, aplicando las transformaciones adecuadas a cada una. Para ello, podemos usar **ColumnTransformer**. Por ejemplo, el siguiente ColumnTransformer aplicará num\_pipeline a los atributos numéricos y cat\_pipeline al atributo categórico:

num\_attribs = ["longitude", "latitude", "housing\_median\_age", "total\_rooms",

"total\_bedrooms", "population", "households", "median\_income"]

cat\_attribs = ["ocean\_proximity"]

cat\_pipeline = make\_pipeline(

SimpleImputer(strategy="most\_frequent"),

OneHotEncoder(handle\_unknown="ignore"))

preprocessing = ColumnTransformer([

("num", num\_pipeline, num\_attribs),

("cat", cat\_pipeline, cat\_attribs),

])

Definimos la lista de nombres de las columnas numéricas y categóricas y construimos una pipeline simple para los atributos categóricos. Después, construimos un ColumnTransformer. Su constructor requiere una lista de tripletas (tuplas de 3), donde cada una contenga un nombre, un transformador y una lista de nombres (o índices) de columnas a las que debería aplicarse el transformador.

Por defecto, las columnas restantes (es decir, las que no estaban en la lista) se dejan fuera, pero puedes configurar el hiperparámetro **remainder** con cualquier transformador si quieres que estas columnas se manejen de manera diferente.

Como hacer una lista con todos los nombres de las columnas no resulta muy conveniente, Scikit-Learn ofrece una función **make\_column\_selector()** que devuelve una función de selector que puedes utilizar para seleccionar de manera automática todas las características de un tipo determinado, como numéricas o categóricas. Puedes pasar esta función de selector a ColumnTransformer en vez de nombres de columnas o índices. Además, si te da igual poner nombre a los transformadores, puedes utilizar **make\_column\_transformer()**, que elige los nombres por ti, al igual que make\_pipeline().

Por ejemplo, el siguiente código crea el mismo ColumnTransformer que antes, salvo porque los transformadores reciben automáticamente los nombres "pipeline-1" y "pipeline-2" en vez de "num" y "cat":

preprocessing = make\_column\_transformer(

(num\_pipeline, make\_column\_selector(dtype\_include=np.number)),

(cat\_pipeline, make\_column\_selector(dtype\_include=object)),

)

Ahora estamos listos para aplicar este ColumnTransfomer a los datos de las casas:

housing\_prepared = preprocessing.fit\_transform(housing)

Tenemos una pipeline de preprocesamiento que toma un conjunto de datos de entrenamiento completo y aplica cada transformador a las columnas apropiadas, después concatena las columnas transformadas en horizontal. De nuevo, esto devuelve una matriz NumPy, pero podemos envolver los datos en dataframe.

housing\_prepared\_fr = pd.DataFrame(

housing\_prepared,

columns=preprocessing.get\_feature\_names\_out(),

index=housing.index)

housing\_prepared\_fr.head(2)

Ahora, nos interesa crear una sola pipeline que realice todas las transformaciones con las que hemos experimentado hasta ahora.

Vamos a repasar qué hará pipeline y por qué:

* Los valores que faltan en las características numéricas se imputarán sustituyéndolos por la mediana, ya que la mayoría de los algoritmos de machine learning no esperan que falten valores.
* En las características categóricas, los valores que faltan se sustituirán por la categoría más frecuente.
* La característica categórica tendrá una codificación one-hot, ya que la mayoría de los algoritmos de machine learning solo aceptan entradas numéricas.
* Se calcularán y añadirán algunas características: bedrooms\_ratio, rooms\_per\_house y people\_per\_house.
* Se añadirán también algunas características de similitud de grupos. Es probable que resulten más útiles para el modelo que la latitud y la longitud.
* Las características con una cola larga se sustituirán por su logaritmo, ya que la mayoría de los modelos prefieren características con distribuciones más o menos uniformes o gaussianas.
* Todas las características numéricas se estandarizarán, puesto que la mayoría de los algoritmos de machine learning prefieren que todas las características tengan más o menos la misma escala.

def column\_ratio(X):

return X[:, [0]] / X[:, [1]]

def ratio\_name(function\_transformer, feature\_names\_in):

return ["ratio"] # feature names out

def ratio\_pipeline():

return make\_pipeline(

SimpleImputer(strategy="median"),

FunctionTransformer(column\_ratio, feature\_names\_out=ratio\_name),

StandardScaler())

log\_pipeline = make\_pipeline(

SimpleImputer(strategy="median"),

FunctionTransformer(np.log, feature\_names\_out="one-to-one"),

StandardScaler())

default\_num\_pipeline = make\_pipeline(SimpleImputer(strategy="median"),

StandardScaler())

cat\_pipeline = make\_pipeline(

SimpleImputer(strategy="most\_frequent"),

OneHotEncoder(handle\_unknown="ignore"))

preprocessing = ColumnTransformer([

("bedrooms", ratio\_pipeline(), ["total\_bedrooms", "total\_rooms"]),

("rooms\_per\_house", ratio\_pipeline(), ["total\_rooms", "households"]),

("people\_per\_house", ratio\_pipeline(), ["population", "households"]),

("log", log\_pipeline, ["total\_bedrooms", "total\_rooms", "population",

"households", "median\_income"]),

("geo", cluster\_simil, ["latitude", "longitude"]),

("cat", cat\_pipeline, make\_column\_selector(dtype\_include=object)),

],

remainder=default\_num\_pipeline) # one column remaining: housing\_median\_age

Si ejecutas este ColumnTransformer, realiza todas las transformaciones y genera como salida una matriz NumPy con 24 características:

housing\_prepared = preprocessing.fit\_transform(housing)

housing\_prepared.shape

(16512, 24)

preprocessing.get\_feature\_names\_out()

array(['bedrooms\_\_ratio', 'rooms\_per\_house\_\_ratio', 'people\_per\_house\_\_ratio', 'log\_\_total\_bedrooms', 'log\_\_total\_rooms', 'log\_\_population', 'log\_\_households', 'log\_\_median\_income', 'geo\_\_Cluster 0 similarity', 'geo\_\_Cluster 1 similarity', 'geo\_\_Cluster 2 similarity', 'geo\_\_Cluster 3 similarity', 'geo\_\_Cluster 4 similarity', 'geo\_\_Cluster 5 similarity', 'geo\_\_Cluster 6 similarity', 'geo\_\_Cluster 7 similarity', 'geo\_\_Cluster 8 similarity', 'geo\_\_Cluster 9 similarity', 'cat\_\_ocean\_proximity\_<1H OCEAN', 'cat\_\_ocean\_proximity\_INLAND', 'cat\_\_ocean\_proximity\_ISLAND', 'cat\_\_ocean\_proximity\_NEAR BAY', 'cat\_\_ocean\_proximity\_NEAR OCEAN', 'remainder\_\_housing\_median\_age'], dtype=object)

## ENTRENAMIENTO Y EVALUACIÓN DEL MODELO

Ya estamos listos para seleccionar y entrenar un modelo de machine learning.

Vamos a entrenar un modelo de regresión lineal muy básico para empezar.

lin\_reg = make\_pipeline(preprocessing, LinearRegression())

lin\_reg.fit(housing, housing\_labels)

Una vez tenemos el modelo entrenado podemos hacer predicciones:

housing\_predictions = lin\_reg.predict(housing)

housing\_predictions[:5].round(-2)

array([260100., 357900., 125800., 110000., 298900.])

Podemos comparar los valores predichos con los reales de la y (puesto que hemos hecho predicciones sobre el conjunto de entrenamiento podemos comparar los valores predichos con los valores reales).

housing\_labels[:5].values

array([458300., 483800., 101700., 96100., 361800.])

Vamos ahora a evaluar el modelo para ver lo bueno (no sólo comparando las primeras 5 filas). Para poder evaluar un modelo hemos tenido que decidir qué medida vamos a usar para ver la bondad de nuestro modelo. Una medida de rendimiento típica para los problemas de regresión es **la raíz del error cuadrático medio (RECM)** y la función **mean\_squared\_error()** de Scikit-Learn, con el argumento **squared** establecido como False mide este error:

lin\_rmse = mean\_squared\_error(housing\_labels, housing\_predictions, squared=False)

lin\_rmse

70630.22169174394

Esto es mejor que nada, pero está claro que no es un resultado genial: los valores median\_housing\_values de la mayoría de los distritos están entre 120.000 y 265.000 dólares, así que un error de predicción de 70.630 dólares no es muy satisfactorio.

Este es un ejemplo de modelo que subajusta los datos de entrenamiento y puede ser por dos razones:

* Las características no proporcionan suficiente información para hacer buenas predicciones
* El modelo no es lo bastante potente.

Vamos a ver que pasa si evaluamos el modelo en el conjunto de prueba:

X\_test = strat\_test\_set.drop("median\_house\_value", axis=1)

y\_test = strat\_test\_set["median\_house\_value"].copy()

housing\_test\_predictions = lin\_reg.predict(X\_test)

lin\_test\_rmse = mean\_squared\_error(y\_test, housing\_test\_predictions, squared=False)

lin\_test\_rmse

76862.27404904472

El resultado es aún pero (lógico que el resultado con el conjunto con el que no hemos entrenado sea pero).

Podríamos intentar añadir más características, pero, primero, vamos a probar un modelo más complejo para ver qué tal lo hace.

Vamos a entrenar un **DecisionTreeRegressor**, ya que se trata de un modelo bastante potente, capaz de encontrar relaciones no lineales complejas en los datos:

tree\_reg = make\_pipeline(preprocessing, DecisionTreeRegressor(random\_state=42))

tree\_reg.fit(housing, housing\_labels)

Ahora que el modelo está entrenado, vamos a evaluarlo con el conjunto de entrenamiento:

housing\_predictions = tree\_reg.predict(housing)

tree\_rmse = mean\_squared\_error(housing\_labels, housing\_predictions, squared=False)

tree\_rmse

0.0

¿Qué? ¿Ni un solo error? ¿Podría ser este modelo absolutamente perfecto? Por supuesto, pero es mucho más probable que el modelo haya sobreajustado mucho los datos.

Veamos que ocurre con el conjunto de prueba

housing\_test\_predictions = tree\_reg.predict(X\_test)

tree\_test\_rmse = mean\_squared\_error(y\_test, housing\_test\_predictions, squared=False)

tree\_test\_rmse

70075.85803300602

Efectivamente el modelo está sobreajustando (es bueno para los datos de entrenamiento pero no generaliza bien a otros datos).

#### VALIDACIÓN CRUZADA

Al estar usando el conjunto de test para decidir qué modelo cogemos estamos condicionando nuestra decisión al conjunto de prueba y puede que elijamos un conjunto que se adapte bien al conjunto de entrenamiento y al conjunto de prueba pero no generalice bien a otros conjuntos.

Estaría bien poder decidir cuál es el mejor modelo sin tener que usar el conjunto de prueba hasta el final. Una alternativa muy buena es utilizar la característica de **validación cruzada de k iteraciones** de Scikit-Learn.

El siguiente código divide de manera aleatoria el conjunto de entrenamiento en 10 subconjuntos no solapados llamados iteraciones y, a continuación, entrena y evalúa el modelo del árbol de decisiones 10 veces, eligiendo una iteración diferente cada vez para la evaluación y usando las otras 9 iteraciones para el entrenamiento. El resultado es una matriz que contiene las 10 puntuaciones de evaluación:

tree\_rmses = -cross\_val\_score(tree\_reg, housing, housing\_labels,

scoring="neg\_root\_mean\_squared\_error", cv=10)

Las características de validación cruzada de Scikit‑Learn esperan una función de utilidad (cuanto mayor sea, mejor), más que una función de pérdida (cuanto menor sea, mejor), así que, en realidad, la función de puntuación es lo contrario a la RECM. Es un valor negativo, así que necesitamos cambiar el signo de la salida para obtener las puntuaciones de RECM.

Fíjate en los resultados:

pd.Series(tree\_rmses).describe()

count 10.000000

mean 70042.976853

std 2211.223510

min 67197.302832

25% 68117.719418

50% 70520.112088

75% 71261.203808

max 73874.610882

dtype: float64

El árbol de decisiones tiene un RECM de aproximadamente 70.041, con una desviación estándar de alrededor de 2.211. Si calculamos la misma métrica para el modelo de regresión lineal, veremos que el RECM medio es 71.190 y la desviación estándar es 3.986. Así pues, parece que el rendimiento del modelo del árbol de decisiones es un poco mejor que el del modelo lineal, pero la diferencia es mínima.

Vamos a probar un último modelo: el **RandomForestRegressor**. Los random forests funcionan entrenando muchos árboles de decisiones en subconjuntos aleatorios de las características y haciendo un promedio de sus predicciones:

forest\_reg = make\_pipeline(preprocessing, RandomForestRegressor(random\_state=42))

forest\_rmses = -cross\_val\_score(forest\_reg, housing, housing\_labels,

scoring="neg\_root\_mean\_squared\_error", cv=10)

Veamos las puntuaciones:

pd.Series(forest\_rmses).describe()

count 10.000000

mean 49632.089212

std 1020.064249

min 47620.869586

25% 49099.175089

50% 49749.949265

75% 50183.743922

max 51263.448191

dtype: float64

Esto está mucho mejor. Sin embargo, si entrenamos un RandomForest y medimos el RECM en el conjunto de entrenamiento, el resultado será aproximadamente 18.469 que es mucho más bajo, lo que significa que todavía hay mucho sobreajuste.

Las posibles soluciones son simplificar el modelo, restringirlo (es decir, regularizarlo) o conseguir muchos más datos de entrenamiento.

Antes de pasar a la optimización del modelo deberíamos probar otros muchos modelos de distintas categorías de algoritmos de machine learning (por ejemplo, varias máquinas de vectores soporte con diferentes kernels y, posiblemente, una red neuronal), sin dedicar demasiado tiempo a ajustar los hiperparámetros. El objetivo es seleccionar algunos modelos (entre dos y cinco) prometedores.

## OPTIMIZACIÓN DEL MODELO

Vamos a suponer que ya tenemos una selección de modelos prometedores. Ahora necesitamos optimizarlos, es decir, ajustar sus hiperparámetros para conseguir los mejores resultado posibles.

Una opción consiste en juguetear con los hiperparámetros a mano, hasta encontrar una combinación genial de valores de hiperparámetros. Sería una tarea muy tediosa.

Vamos a probar la búsqueda exhaustiva y la búsqueda aleatoria.

#### BÚSQUEDA EXHAUSTIVA

**GridSearchCV** de Scikit-Learn puede buscar los mejores parametros por nosotros. Lo único que tenemos que hacer es decirle con qué hiperparámetros queremos experimentar y qué valores probar y utilizará validación cruzada para evaluar todas las combinaciones posibles de valores de hiperparámetros.

Por ejemplo, el siguiente código busca las mejores combinaciones de valores de hiperparámetros para RandomForestRegressor:

full\_pipeline = Pipeline([

("preprocessing", preprocessing),

("random\_forest", RandomForestRegressor(random\_state=42)),

])

param\_grid = [

{

"preprocessing\_\_log\_\_functiontransformer\_\_func": [np.log, np.log1p],

"random\_forest\_\_max\_features": [4, 6, 8, 10],

}

]

grid\_search = GridSearchCV(full\_pipeline, param\_grid, cv=3, scoring='neg\_root\_mean\_squared\_error')

grid\_search.fit(housing, housing\_labels)

Observa que podemos referirnos a cualquier hiperparámetro de cualquier estimador en una pipeline, incluso si este estimador está anidado a un nivel profundo dentro de varias pipelines y transformadores de columnas. Por ejemplo, cuando Scikit-Learn ve "preprocessing\_\_log\_\_functiontransformer\_\_func", divide esta cadena por los dobles guiones bajos y, después, busca un hiperparámetro llamado "func" en el transformador functiontransformer dentro de la pipeline log dentro de la pipeline preprocessing. De manera similar, random\_forest\_\_max\_features remite al hiperparámetro max\_features del estimador llamado "random\_forest".

Tenemos un diccionario en este param\_grid, así que GridSearchCV las 2 × 4 = 8 combinaciones de los valores de los hiperparámetros func y max\_features especificados y entrenará la pipeline 3 veces por combinación (cv=3), porque estamos usando validación cruzada de 3 iteraciones. Eso significa que habrá un total de 8 × 3 = 24 rondas de entrenamiento. Puede que lleve un tiempo, pero, cuando acabe, podemos obtener la mejor combinación de parámetros así:

grid\_search.best\_params\_

{'preprocessing\_\_log\_\_functiontransformer\_\_func': <ufunc 'log1p'>, 'random\_forest\_\_max\_features': 6}

Podemos ver que el mejor modelo se obtiene estableciendo func como log1p y configurando max\_features como 6.

Podemos acceder al mejor modelo accediendo a la variable **best\_estimator\_**.

Las puntuaciones de la evaluación están disponibles accediendo a la variable **cv\_results\_**. Esto es un diccionario, pero si lo envolvemos en un dataframe obtenemos una lista de todas las puntuaciones de prueba para cada combinación de hiperparámetros y para cada división de la validación cruzada, además de la puntuación de prueba media de todas las divisiones:

cv\_res = pd.DataFrame(grid\_search.cv\_results\_)

cv\_res.sort\_values(by="mean\_test\_score", ascending=False, inplace=True)

cv\_res = cv\_res[[

"param\_preprocessing\_\_log\_\_functiontransformer\_\_func",

"param\_random\_forest\_\_max\_features",

"split0\_test\_score",

"split1\_test\_score",

"split2\_test\_score",

"mean\_test\_score",

]]

score\_cols = ["split0", "split1", "split2", "mean\_test\_rmse"]

cv\_res.columns = ["log\_func", "max\_features"] + score\_cols

cv\_res[score\_cols] = -cv\_res[score\_cols].round().astype(np.int64)

cv\_res

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

La puntuación mejor puntuación del RECM es 49.462, que es mejor que la puntuación obtenida antes usando los valores de hiperparámetros predeterminados (que era 49.632).

#### BÚSQUEDA ALEATORIZADA

El enfoque de la búsqueda exhaustiva está bien cuando se explora una cantidad relativamente pequeña de combinaciones, como en el ejemplo anterior, pero, a menudo, es preferible utilizar **RandomizedSearchCV**, sobre todo cuando el espacio de búsqueda de hiperparámetros es grande. Esta clase puede utilizarse casi del mismo modo que la clase GridSearchCV, pero, en vez de probar todas las combinaciones posibles, evalúa un número fijo de combinaciones, seleccionando un valor aleatorio para cada hiperparámetro en cada iteración. Puede parecer sorprendente, pero este enfoque tiene varios beneficios:

* Si algunos de los hiperparámetros son continuos (o discretos, pero con muchos valores posibles) y dejas que la búsqueda aleatorizada se ejecute durante, pongamos, 1.000 iteraciones, explorará 1.000 valores diferentes para cada uno de estos hiperparámetros, mientras que una búsqueda exhaustiva solo exploraría los pocos valores listados para cada uno.
* Supongamos que un hiperparámetro no supone en realidad mucha diferencia, pero todavía no lo sabes. Si tienes 10 valores posibles y lo añades a la búsqueda exhaustiva, el entrenamiento durará 10 veces más, pero, si lo añades a una búsqueda aleatoria, no marcará ninguna diferencia.
* Si hay 6 hiperparámetros para explorar, cada uno con 10 posibles valores, la búsqueda exhaustiva no ofrece otra opción que entrenar el modelo un millón de veces, mientras que la búsqueda aleatoria siempre puede ejecutarse para cualquier número de iteraciones que elijas.

Para cada hiperparámetro, debes proporcionar, bien una lista de valores posibles, bien una distribución de probabilidad:

param\_distribs = { 'preprocessing\_\_log\_\_functiontransformer\_\_func': [np.log, np.log1p],

'random\_forest\_\_max\_features': randint(low=2, high=20)}

rnd\_search = RandomizedSearchCV(

full\_pipeline, param\_distributions=param\_distribs, n\_iter=10, cv=3,

scoring='neg\_root\_mean\_squared\_error', random\_state=42)

rnd\_search.fit(housing, housing\_labels)

#### OTROS

A menudo, conseguiremos un mayor entendimiento del problema inspeccionando los mejores modelos. Por ejemplo, RandomForestRegressor puede indicar la importancia relativa de cada atributo para hacer predicciones exactas:

final\_model = rnd\_search.best\_estimator\_

feature\_importances = final\_model["random\_forest"].feature\_importances\_

feature\_importances.round(2)

array([0.06, 0.06, 0.11, 0.02, 0.02, 0.02, 0.01, 0.35, 0.01, 0.15, 0., 0., 0., 0.08, 0.07, 0.04])

Vamos a ordenar estas puntuaciones de importancia en orden descendente y a mostrarlas junto a sus nombres de atributos correspondientes:

sorted(zip(feature\_importances,

final\_model["preprocessing"].get\_feature\_names\_out()),

reverse=True)

[(0.3524278549190886, 'log\_\_median\_income'),

(0.15430857024783234, 'cat\_\_ocean\_proximity\_INLAND'),

(0.11152207012042085, 'people\_per\_house\_\_ratio'),

(0.07677638156445867, 'remainder\_\_longitude'),

(0.06719647370408954, 'remainder\_\_latitude'),

(0.060639471465569535, 'bedrooms\_\_ratio'),

(0.05578209809208504, 'rooms\_per\_house\_\_ratio'),

(0.041608155790607664, 'remainder\_\_housing\_median\_age'),

(0.016317692505479004, 'log\_\_total\_rooms'),

(0.015915318115305437, 'log\_\_population'),

(0.015468882724054852, 'log\_\_total\_bedrooms'),

(0.014327519624211221, 'log\_\_households'),

(0.01065287031994883, 'cat\_\_ocean\_proximity\_<1H OCEAN'),

(0.0045723629694189635, 'cat\_\_ocean\_proximity\_NEAR OCEAN'),

(0.0022902311652259065, 'cat\_\_ocean\_proximity\_NEAR BAY'),

(0.00019404667220360706, 'cat\_\_ocean\_proximity\_ISLAND')]

Con esta información, convendría intentar dejar fuera algunas de las características menos útiles (por ejemplo, parece que solo una categoría de ocean\_proximity es realmente útil, así que podríamos dejar fuera las otras).

#### EVALUACIÓN EN EL CONJUNTO DE PRUEBA

Después de optimizar los modelos un tiempo, al final deberíamos tener un sistema que funcione bastante bien. Ahora, es el momento de evaluar el modelo final en el conjunto de prueba. No hay nada de especial en este proceso; solo tenemos que coger los predictores y las etiquetas del conjunto de prueba y ejecutar el final\_model para transformar los datos y hacer predicciones y, después, evaluar estas predicciones:

X\_test = strat\_test\_set.drop("median\_house\_value", axis=1)

y\_test = strat\_test\_set["median\_house\_value"].copy()

final\_predictions = final\_model.predict(X\_test)

final\_rmse = mean\_squared\_error(y\_test, final\_predictions, squared=False)

final\_rmse

49110.42998521564

## PRESENTACIÓN DE LA SOLUCIÓN

Ahora tenemos que presentar nuestra solución (resaltando lo que hemos aprendido, lo que ha funcionado y lo que no, qué conjeturas se habían hecho y cuáles son las limitaciones de nuestro sistema), documentar todo y crear presentaciones atractivas con visualizaciones claras y afirmaciones fáciles de recordar (por ejemplo, "los ingresos medianos son el predictor número uno de los precios de las casas.").

En este ejemplo de las casas de California, el rendimiento final del sistema no es mucho mejor que las estimaciones de precios de los expertos, que se desviaban en torno a un 30%, pero puede que siga siendo buena idea lanzarlo, sobre todo si deja algo de tiempo extra a los expertos para que puedan trabajar en tareas más interesantes y productivas.

¡Perfecto, ya tenemos la aprobación para el lanzamiento! Ahora tenemos que preparar nuestra solución para desplegar el modelo en el entorno de producción. La manera más básica de hacerlo es guardar el mejor modelo que hayamos entrenado, transferir el archivo al entorno de producción y cargarlo. Para guardar el modelo, podemos utilizar la biblioteca joblib así:

joblib.dump(final\_model, "Recursos/my\_california\_housing\_model.pkl")

Una vez que el modelo se transfiere a producción, podemos cargarlo y utilizarlo. Para eso, primero debemos importar cualquier clase personalizada y función de las que dependa el modelo, después cargar el modelo usando joblib y utilizarlo para hacer predicciones:

def column\_ratio(X):

return X[:, [0]] / X[:, [1]]

# [...] Todas las funciones definidas por nosotros

final\_model\_reloaded = joblib.load("Recursos/my\_california\_housing\_model.pkl")

new\_data = housing[:5] # vamos a suponer que son los datos de 5 distritos nuevos

predictions = final\_model\_reloaded.predict(new\_data)

Puede que el modelo vaya a utilizarse en un sitio web: el usuario escribirá unos datos sobre un distrito nuevo y hará clic en un botón para estimar el precio. Eso enviará una consulta con los datos al servidor web, y, por último, el código simplemente llamará al método predict() del modelo.

Como alternativa, podemos envolver el modelo dentro de un servicio web dedicado al que pueda consultar nuestra aplicación web mediante una API REST. Esto hace que sea más fácil actualizar el modelo a versiones nuevas sin interrumpir la aplicación principal. Además simplifica el escalado y permite que nuestra aplicación web utilice cualquier lenguaje de programación, no solo Python.

Imagen de la pantalla de un celular con letras

Descripción generada automáticamente con confianza baja

Pero el despliegue no es el final de la historia. También es necesario escribir código de monitorización para comprobar el rendimiento en vivo del sistema en intervalos regulares y activar alertas cuando baje. Pero ¿cómo se hace eso? Bueno, depende. En algunos casos, el rendimiento del modelo puede deducirse a partir de métricas descendentes. Por ejemplo, si el modelo es parte de un sistema de recomendaciones y sugiere productos en los que los usuarios podrían estar interesados, entonces es fácil monitorizar el número de productos recomendados que se venden cada día. Si este número baja (en comparación con los productos no recomendados), entonces el principal sospechoso es el modelo. Esto puede deberse a que el modelo necesita volver a entrenarse con datos nuevos.

Sin embargo, puede que también se necesite un análisis humano para evaluar el rendimiento del modelo. Por ejemplo, supongamos que hemos entrenado un modelo de clasificación de imágenes para detectar varios defectos en productos de una cadena de producción. ¿Cómo podemos recibir una alerta si el rendimiento del modelo baja, antes de que miles de productos defectuosos se envíen a los clientes? Una solución es enviar a evaluadores humanos una muestra de todas las imágenes que ha clasificado el modelo (sobre todo imágenes sobre las que el modelo no estaba seguro).

Si los datos siguen evolucionando, tenemos que actualizar los conjuntos de datos y volver a entrenar el modelo con regularidad. Probablemente, deberíamos automatizar todo el proceso lo máximo posible.

También hay que asegurarse de que se evalúa la calidad de los datos de entrada del modelo.

Por último, hay que asegurarnos de tener copias de seguridad de todos los modelos que creamos y tener los procesos y herramientas preparados para volver a un modelo anterior con rapidez, en caso de que el modelo nuevo empiece a fallar mucho por alguna razón.

Ahora es un buen momento para coger un ordenador, seleccionar un conjunto de datos que nos parezca interesante y probar a pasar por todo el proceso, de principio a fin. Un buen punto para empezar es el sitio web <https://kaggle.com/> donde encontraremos conjuntos de datos con el que jugar con un objetivo claro.