**MACHINE LEARNING INTRO PRACTICA**

**FASE INICIAL**

Diagrama

Descripción generada automáticamente

1. Lo primero es saber qué queremos resolver: Problema de negocio vs Problema técnico
2. Lo siguiente es entender que datos tenemos y podemos tener
3. Distinguir entre feature y target
4. Separar entre Train y Test

**1. El problema de "Negocio"**

En nuestro caso la compañía "California Real State Investment Funds" quiere mejorar sus inversiones inmobiliarias para fondos privados. Para ello quiere mejorar su estimación de precios de posibles adquisiciones de viviendas y así escoger mejor sus inversiones y negociar también con ventaja en la adquisición. En concreto le gustaría saber cuál podría ser el precio medio de una casa en una zona de California determinada por una serie de características como su localización, antigüedad media de las casas de la zona, población de la zona, etc.

**Tendremos que traducir nuestro problema de negocio en un problema técnico**

En este caso, parece claro que queremos predecir el precio medio de una casa en una zona determinada. Pero, no vale solo con eso, tendremos que saber en función de qué factores (features) concretos (no sólo una lista "vaga" o "incompleta"), con que margen de error tendríamos un modelo aceptable es decir una métrica de evaluación, tendremos que definir que tipo de aprendizaje (supervisado o no supervisado, por refuerzo...), etc, etc. ¿Cómo se obtiene esto? Preguntando y deduciendo. Lo iremos haciendo poco a poco. Supongamos que ya nos ha contestado al objetivo de negocio y además nos han dicho que nuestro sistema debe ayudar a los analistas de precios que luego introducirán el dato en otro sistema. Estos analistas ya tienen su propia forma de hacer los cálculos, con lo que podrán servirnos como "modelo de referencia" o baseline... Pero son muchas cosas... vayamos por partes...

Hagamos nuestra cheklist:

* Objetivo de negocio [X] check
* Objetivo tecnico [X] check
* Tipo de modelado (supervisado, no supervisado) [] pending
* Features [] pending
* Target, si hay [] pending
* Tipo de problema (clasificación, regresión, etc) [] pending
* Métrica de Evaluacion [] pending
* Separación Train-Test [] pending

Preguntamos y "California Real State Investment Funds" posee un dataset de muchas zonas con sus precios reales. Y además nos dice que las variables de ese dataset puede tenerlas para cualquier zona futura que quiera valorar.

Pues manos a la obra, hay que conseguir los datos...

**2. Obtener los datos y echar "un primer vistazo"**

Para simplificar este ejemplo, suponemos que tenemos acceso directo a los datos que ese encuentra en un fichero. Es hora de leerlo y antes de importar nuestras librerías (observa que hay un nuevo jugador de equipo):

[1]:

**import** matplotlib.pyplot **as** plt

**import** numpy **as** np

**import** pandas **as** pd

**from** sklearn.model\_selection **import** train\_test\_split

df\_inmo **=** pd.read\_csv("./data/ejemplo\_housing.csv")

Como siempre, echaremos un vistazo, pero ojo no podemos hacer más antes de separar en train y test para no "contaminar" nuestro modelo.

df\_inmo.head()

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

En este momento deberíamos hacernos nuestra tabla con definiciones, tipo de columna y prioridades... Si estuvieramos haciendo un EDA. Ahora nos llega con la definición y con el tipo de variable. Por simplicidad y tiempo no lo vamos a hacer aquí con detalle. Simplemente ver que la mayoría de las variables o columnas son de tipo numérico continuo salvo esa última "OCEAN\_PROXIMITY" a la que echaremos un ojo para entenderla mejor.

df\_inmo.info()

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

RangeIndex: 20433 entries, 0 to 20432

Data columns (total 10 columns):

# Column Non-Null Count Dtype

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

0 longitude 20433 non-null float64

1 latitude 20433 non-null float64

2 housing\_median\_age 20433 non-null float64

3 total\_rooms 20433 non-null float64

4 total\_bedrooms 20433 non-null float64

5 population 20433 non-null float64

6 households 20433 non-null float64

7 median\_income 20433 non-null float64

8 median\_house\_value 20433 non-null float64

9 ocean\_proximity 20433 non-null object

dtypes: float64(9), object(1)

memory usage: 1.6+ MB

df\_inmo["ocean\_proximity"].value\_counts()

ocean\_proximity

<1H OCEAN 9034

INLAND 6496

NEAR OCEAN 2628

NEAR BAY 2270

ISLAND 5

Name: count, dtype: int64

Tenemos cinco etiquetas, con frecuencias diferentes. Me lo apunto para luego. Veamos ahora de un golpe las variables numéricas:

df\_inmo.describe()

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza baja

Más que con los valores, con lo que nos quedamos es que son rangos muy variados (de miles, de dolares, en el "median\_house\_value", hasta unidades y decenas en "median\_income"). Esto tendremos que tratarlo como veremos en un par de sesiones, porque a los algoritmos no le suelen sentar bien estas diferencias tan potentes.

Echemos un vistazo, y digo vistazo, porque todavía no podemos tomar números ni hacer análisis univariantes, simplemente veamos la pinta que tienen las variables.

*# Código extra para los tamaños de las fuentes y los "ticks" de los ejes, esto no lo vimos al hablar de matplotlib así que una extra*

plt.rc('font', size**=**14)

plt.rc('axes', labelsize**=**14, titlesize**=**14)

plt.rc('legend', fontsize**=**14)

plt.rc('xtick', labelsize**=**10)

plt.rc('ytick', labelsize**=**10)

df\_inmo.hist(bins**=**50, figsize**=**(12, 8)) *# La forma de generar histogramas de una sola vez para las variables numéricas de un dataset*

plt.show()

Gráfico, Gráfico de cajas y bigotes

Descripción generada automáticamente

Y aquí podemos ver:

1- El ingreso medio ("median\_income") esta claro que no está ni en dólares ni en euros. Tendríamos que preguntar a quien nos dio los datos. En este caso nos dirían que el ingreso se ha traducido a una escala de 0 a 15 (dividiendo por 10000 dolares pero además se han limitado las zonas con ingresos medios por encima de 150000 a 15)

2- Existen más variables donde se han "capado" (puesto límite artificial) como "median\_house\_value" y "housing\_median\_age". Esto puede ser un problema porque el modelo puede aprender que no debe dar valores por encima de estos. Aquí hay que preguntar a los "jefes" que quieren hacer y tendremos que decidir (buscar los datos exactos para los capados o pasar de estos valores). Lo trataremos en la fase de tratamiento de Features.

3- Se ve que las escalas (ejes x) son muy diferentes como ya habíamos observado en los números.

4- Hay muchas distribuciones tipo "precio del billete" del Titanic, es decir desplazadas a la izquierda y con aparente larga cola. Esto también puede sentarle mal a nuestros algoritmos de modelado y en sprints posteriores veremos como "normalizar" las dsitribuciones de datos para que los algoritmos creen mejores modelos.

Pues hasta aquí el vistazo, no podemos hacer más si no queremos hacernos trampas. Es hora de separar los problemas de estudio de los problemas de examen. Es decir el train del test. Pero antes completemos nuestro checklist.

**3. Target, Features, tipo de problema**

En este caso tenemos una variable que es "claramente" nuestro target (es decir lo que querremos predecir para zonas nuevas): "mean\_house\_value".

Además el tener target (la Y del modelo), nos dice que estamos tratando con un modelado supervisado.

En principio las features serán el resto de variables, pero puede que luego nos descontemos alguna o transformemos otras.

Y finalmente, queremos predecir una variable continua con lo que estaremos tratando con un problema de regresión.

En nuestra checklist:

* Objetivo de negocio [X] check
* Objetivo tecnico [X] check
* Tipo de modelado (supervisado, no supervisado): SUPERVISADO [X] check
* Features [] pending (aunque tengamos la base no tenemos las defintivas)
* Target, si hay: "mean\_house\_value" [X] check
* Tipo de problema (clasificación, regresión, etc): Regresión [X] check
* Métrica de Evaluacion [] pending [podríamos elegirla ya pero lo dejamos para la fase de evaluación para no alargar la sesión demasiado]
* Separación Train-Test [] pending

**4. Separar Train y Test**

Necesitamos tener un dataset para entrenar, pero sobre todo necesitamos un dataset para poder tener una idea de como va a funcionar el modelo cuando le pasemos datos que no ha visto.

¿Y como se crea un set de Test? Generalmente generando dos muestras aleatorias del dataset de partida. Las cifras típicas son un 80% para train y un 20% para test y la forma de hacerlo rápidamente es acudir a la fución que nos hemos importado:

train\_set, test\_set **=** train\_test\_split(df\_inmo, test\_size**=**0.2, random\_state**=**42)

len(train\_set)

len(test\_set)

No podemos ver el test desgraciadamente :-).

Y con esto habríamos terminado.

**EDA**

[1]:

*# Recuperemos lo hecho en la sesión anterior*

**import** matplotlib.pyplot **as** plt

**import** numpy **as** np

**import** pandas **as** pd

**import** seaborn **as** sns

**from** bootcampviztools **import** pinta\_distribucion\_categoricas, plot\_categorical\_numerical\_relationship, plot\_grouped\_histograms

**from** sklearn.model\_selection **import** train\_test\_split

*# Código extra para los tamaños de las fuentes y los "ticks" de los ejes, esto no lo vimos al hablar de matplotlib así que una extra*

plt.rc('font', size**=**14)

plt.rc('axes', labelsize**=**14, titlesize**=**14)

plt.rc('legend', fontsize**=**14)

plt.rc('xtick', labelsize**=**10)

plt.rc('ytick', labelsize**=**10)

df\_inmo **=** pd.read\_csv("./data/ejemplo\_housing.csv")

train\_set, test\_set **=** train\_test\_split(df\_inmo, test\_size**=**0.2, random\_state**=**42)

Diagrama

Descripción generada automáticamente

**1. Limpieza y transformación**

Deberíamos acometer la limpieza y transformación, pero como ya vimos no hay valores nulos y los datos parecen bastante limpios

[2]:

df\_inmo.info()

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

RangeIndex: 20433 entries, 0 to 20432

Data columns (total 10 columns):

# Column Non-Null Count Dtype

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

0 longitude 20433 non-null float64

1 latitude 20433 non-null float64

2 housing\_median\_age 20433 non-null float64

3 total\_rooms 20433 non-null float64

4 total\_bedrooms 20433 non-null float64

5 population 20433 non-null float64

6 households 20433 non-null float64

7 median\_income 20433 non-null float64

8 median\_house\_value 20433 non-null float64

9 ocean\_proximity 20433 non-null object

dtypes: float64(9), object(1)

memory usage: 1.6+ MB

Unicamente parece interesante convertir "mean\_income" en una variable categórica, porque ya de por sí se ha intentado hacer un escalado con la misma (dividida entre 10.000). Esto es algo que ya hicimos varias veces cuando analizamos datos, recurriendo a la función cut de pandas. Hagámoslo:

[3]:

train\_set["income\_cat"] **=** pd.cut(train\_set["median\_income"],

bins**=**[0., 1.5, 3.0, 4.5, 6., np.inf],

labels**=**[1, 2, 3, 4, 5])

[4]:

train\_set["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

[NOTA: Importante, toda transformación que se haga en las variables de train habrá que hacerlas en los datos de test antes de poder aplicarles el modelo]

**2. Análisis univariante**

En un proceso estricto y formal de ML ahora haríamos el análisis univariante de todas las variables. Por simplicidad vamos a repetir únicamente el código para ver las distribuciones de las numéricas y las frecuencias de las categóricas:

[5]:

train\_set.hist(bins**=**50, figsize**=**(12, 8)) *# La forma de generar histogramas de una sola vez para las variables numéricas de un dataset*

plt.show()

Gráfico, Gráfico de cajas y bigotes

Descripción generada automáticamente

Y las frecuencias:

[6]:

pinta\_distribucion\_categoricas(train\_set, ["income\_cat","ocean\_proximity"], relativa**=** **True**, mostrar\_valores**=** **True**)

Gráfico, Gráfico de barras

Descripción generada automáticamente

La distribución numérica sigue las mismas pautas, como era de esperar, cuando hicimos el primer vistazo y se mantienen las mismas consideraciones.

Respecto a las frecuencias, el income de nivel 3 es el más frecuente, el income de nivel 1 el menor y que la mayoría de las zonas no son costeras y además que apenas hay zonas en islas.

No podemos forzar más y tampoco lo necesitamos. Lo realmente importante es ver algunas combinaciones y el bivariante con el target "mean\_house\_value", eso sí también tendremos que decidir que hacer con los "capados" de esta y de la variable "housing\_median\_age".

**2. Análisis Multivariante**

**2.1 Variables Categóricas vs Target**

Empecemos con la proximidad al océano:

[7]:

plot\_categorical\_numerical\_relationship(train\_set, categorical\_col**=** "ocean\_proximity", numerical\_col**=**"median\_house\_value")

Gráfico, Gráfico de barras

Descripción generada automáticamente

Claramente parece que hay relación entre esta variable y el target, me la quedaré como feature. Dado que tenemos los valores geográficos (latitud y longitud), podemos comprobar esta relación y además ver el bivariante con esas dos variables haciendo una visualización:

[8]:

plt.figure(figsize**=**(10,7))

plt.scatter(train\_set["longitude"], train\_set["latitude"], c **=** train\_set["median\_house\_value"], cmap **=** "jet")

plt.grid()

plt.xlabel("longitud")

plt.ylabel("latitud")

cbar **=** plt.colorbar(label **=** "Mediana Precios ($)")

Gráfico, Gráfico de dispersión

Descripción generada automáticamente

[EXTRA: Usando pandas directamente y añadiendo la población como tamaño de la burbuja]

[9]:

train\_set.plot(kind**=**"scatter", x**=**"longitude", y**=**"latitude", grid**=True**,

s**=**train\_set["population"] **/** 100, label**=**"population",

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

legend**=True**, sharex**=False**, figsize**=**(10, 7))

plt.show()

Gráfico, Gráfico de dispersión

Descripción generada automáticamente

Seguramente podríamos hasta simplificar la variable en dos o tres valores (costero, no costero, isla). Pero no lo vamos a hacer.

[10]:

plot\_categorical\_numerical\_relationship(train\_set, categorical\_col**=** "income\_cat", numerical\_col**=**"median\_house\_value")

Gráfico, Gráfico de barras

Descripción generada automáticamente

No hace falta decir nada no. Ahora cuando veamos las relaciones numéricas, lo veremos con el "mean\_income". Esta es otra variable que nos quedaremos como feature.

**2.2 Variables numéricas y target**

Por simplificar, directamente iremos a las correlaciones de las variables numéricas con la variable target y pintando las más significativas:

[11]:

corr\_matrix **=** train\_set.corr(numeric\_only**=** **True**)

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

[11]:

median\_house\_value 1.000000

median\_income 0.687342

total\_rooms 0.134610

housing\_median\_age 0.107896

households 0.067969

total\_bedrooms 0.053060

population -0.020492

longitude -0.046563

latitude -0.143382

Name: median\_house\_value, dtype: float64

Como era de esperar "median\_income" es claramente la que tiene más correlación, y podríamos descartar población y las geográficas que hemos visto que hasta cierto punto están contenidas en la categórica de cercanía al mar (quizás la latitud podríamos quedárnosla)

[12]:

columnas **=** corr\_matrix["median\_house\_value"][corr\_matrix["median\_house\_value"] **>** 0.07].index.to\_list()

[13]:

sns.pairplot(train\_set[columnas]);

Imagen que contiene Diagrama

Descripción generada automáticamente

Como era de esperar gráficamente la "posible" correlación más potente es con "median\_income". Pero en este caso no vamos a descartar las otras, y nos quedaremos con la categórica en vez de con "median\_income".

**2.3 Crear otras variables**

En este caso, podríamos pensar que quizás algunas variables combinadas puedan ser interesantes por ejemplo:

1. El número medio de habitaciones por casa ("total\_rooms"/"households")
2. El número de dormitorios en función del total de habitaciones ("total\_bedrooms"/"total\_rooms")
3. O el número medio de personas por casas ("population"/"housholds")

Creemos estas variables y veamos su correlación con el target

[14]:

train\_set["rooms\_per\_house"] **=** train\_set["total\_rooms"] **/** train\_set["households"]

train\_set["bedrooms\_ratio"] **=** train\_set["total\_bedrooms"] **/** train\_set["total\_rooms"]

train\_set["people\_per\_house"] **=** train\_set["population"] **/** train\_set["households"]

[15]:

corr\_matrix **=** train\_set.corr(numeric\_only**=True**)

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

[15]:

median\_house\_value 1.000000

median\_income 0.687342

rooms\_per\_house 0.151574

total\_rooms 0.134610

housing\_median\_age 0.107896

households 0.067969

total\_bedrooms 0.053060

population -0.020492

people\_per\_house -0.021595

longitude -0.046563

latitude -0.143382

bedrooms\_ratio -0.252579

Name: median\_house\_value, dtype: float64

Hmm, interesante porque hemos encontrado una "mejor" correlación en habitaciones por casa "rooms\_per\_house" pero más interesante es la nueva variable ratio de dormitorios, que presenta una contraintuitiva "anticorrelación". Es decir a menor ratio de dormitorios por habitaciones mayor precio.

**3. Lista de Features**

Para terminar, hacemos nuestra lista de features [Nota: A medida que veamos algoritmos y más llegados a las redes, dejaremos que sea el modelo el que se encargue incluso de esta selección pero es bueno que lo practiquemos y lo entendamos]

1. ocean\_proximity
2. income\_cat
3. rooms\_per\_house
4. total\_rooms
5. housing\_median\_age
6. bedrooms\_ratio

[ ]:

features **=** ["ocean\_proximity", "income\_cat", "rooms\_per\_house","total\_rooms","housing\_median\_age","bedrooms\_ratio"]

target **=** "median\_house\_value"

**ML INTRO PRACTICA (III): Preparación de Features Categóricas**

[1]:

*# Recuperemos lo hecho en la sesión anterior*

**import** matplotlib.pyplot **as** plt

**import** numpy **as** np

**import** pandas **as** pd

**import** seaborn **as** sns

**from** bootcampviztools **import** pinta\_distribucion\_categoricas, plot\_categorical\_numerical\_relationship, plot\_grouped\_histograms

**from** sklearn.model\_selection **import** train\_test\_split

*# Código extra para los tamaños de las fuentes y los "ticks" de los ejes, esto no lo vimos al hablar de matplotlib así que una extra*

plt.rc('font', size**=**14)

plt.rc('axes', labelsize**=**14, titlesize**=**14)

plt.rc('legend', fontsize**=**14)

plt.rc('xtick', labelsize**=**10)

plt.rc('ytick', labelsize**=**10)

df\_inmo **=** pd.read\_csv("./data/ejemplo\_housing.csv")

train\_set, test\_set **=** train\_test\_split(df\_inmo, test\_size**=**0.2, random\_state**=**42)

train\_set["income\_cat"] **=** pd.cut(train\_set["median\_income"],

bins**=**[0., 1.5, 3.0, 4.5, 6., np.inf],

labels**=**[1, 2, 3, 4, 5])

train\_set["rooms\_per\_house"] **=** train\_set["total\_rooms"] **/** train\_set["households"]

train\_set["bedrooms\_ratio"] **=** train\_set["total\_bedrooms"] **/** train\_set["total\_rooms"]

features **=** ["ocean\_proximity", "income\_cat", "rooms\_per\_house","total\_rooms","housing\_median\_age","bedrooms\_ratio"]

target **=** "median\_house\_value"

Diagrama

Descripción generada automáticamente

**1. Tratamiento de variables categóricas**

[2]:

train\_set.info()

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

Index: 16346 entries, 17560 to 15795

Data columns (total 13 columns):

# Column Non-Null Count Dtype

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

0 longitude 16346 non-null float64

1 latitude 16346 non-null float64

2 housing\_median\_age 16346 non-null float64

3 total\_rooms 16346 non-null float64

4 total\_bedrooms 16346 non-null float64

5 population 16346 non-null float64

6 households 16346 non-null float64

7 median\_income 16346 non-null float64

8 median\_house\_value 16346 non-null float64

9 ocean\_proximity 16346 non-null object

10 income\_cat 16346 non-null category

11 rooms\_per\_house 16346 non-null float64

12 bedrooms\_ratio 16346 non-null float64

dtypes: category(1), float64(11), object(1)

memory usage: 1.6+ MB

Para poder aplicar los algoritmos de generación de modelos necesitamos convertir las categorías o valores de las variables categóricas en números de alguna forma. Vamos a ver dos para transformar "ocean\_proximity" e "income\_cat" que tiene el tipo category (que algunos algoritmos maneja pero otros no)

**1.1 Ordinal Encoding**

Una primera forma de convertir los valores de una categoría es utilizar un "codificador ordinal" o "Ordinal Encoder" que convertira cada valor de la categoría en un número entero. El problema es que automáticamente se establece una relación de orden que no necesariamente tiene por qué existir:

[3]:

**from** sklearn.preprocessing **import** OrdinalEncoder

ordinal\_encoder **=** OrdinalEncoder()

train\_set["ordinal\_ocean"] **=** ordinal\_encoder.fit\_transform(train\_set[["ocean\_proximity"]])

[4]:

train\_set[["ordinal\_ocean","ocean\_proximity"]].sample(10)

[4]:

|  | **ordinal\_ocean** | **ocean\_proximity** |
| --- | --- | --- |
| **17551** | 1.0 | INLAND |
| **11212** | 0.0 | <1H OCEAN |
| **7482** | 0.0 | <1H OCEAN |
| **3038** | 1.0 | INLAND |
| **2537** | 3.0 | NEAR BAY |
| **9372** | 1.0 | INLAND |
| **16156** | 0.0 | <1H OCEAN |
| **5987** | 1.0 | INLAND |
| **303** | 1.0 | INLAND |
| **19868** | 1.0 | INLAND |

[5]:

ordinal\_encoder.categories\_

[5]:

[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],

dtype=object)]

El caso de "ocean\_proximity" quizá sea de esos en los que establecer un orden no sea ni claro ni oscuro, ni sí ni no, porque si hay una relación de distancia al mar. Pero tendríamos que forzarlo, de más cerca del mar (0) a más lejos del mar (5).

[6]:

categories **=** [["ISLAND","NEAR OCEAN","NEAR BAY","<1H OCEAN","INLAND"]]

ordinal\_encoder **=** OrdinalEncoder(categories**=** categories)

train\_set["ordinal\_ocean"] **=** ordinal\_encoder.fit\_transform(train\_set[["ocean\_proximity"]])

[7]:

train\_set.ordinal\_ocean.value\_counts()

[7]:

ordinal\_ocean

3.0 7219

4.0 5212

1.0 2101

2.0 1810

0.0 4

Name: count, dtype: int64

En el caso de "income\_cat" tiene más sentido porque en sí es una categórica ordinal:

[8]:

train\_set.income\_cat.value\_counts()

[8]:

income\_cat

3 5771

2 5182

4 2892

5 1847

1 654

Name: count, dtype: int64

No vamos a tocarla, si es necesario para los algoritmos que usemos cambiaremos el tipo sobre la marcha.

**1.2 One-Hot Encoding**

Cuando no estamos seguros o no queremos convertir una variable categórica en una variable ordinal numérica, una alternativa es hacer lo que hicimos con los géneros de las películas en el sprint dedicado al Algebra y Numpy [si hace miles de años] que convertíamos cada valor categórico en un 1 o un cero dentro de un vector. Sólo que ahora no es un vector sino que cada valor posible de la variable categórica se va a convertir en una nueva columna que tendrá valor 0 o 1 si la zona (la instancia) cumple con ese valor. Veámoslo

[9]:

pd.get\_dummies(train\_set, columns**=**["ocean\_proximity"]).head()

[9]:

|  | **longitude** | **latitude** | **housing\_median\_age** | **total\_rooms** | **total\_bedrooms** | **population** | **households** | **median\_income** | **median\_house\_value** | **income\_cat** | **rooms\_per\_house** | **bedrooms\_ratio** | **ordinal\_ocean** | **ocean\_proximity\_<1H OCEAN** | **ocean\_proximity\_INLAND** | **ocean\_proximity\_ISLAND** | **ocean\_proximity\_NEAR BAY** | **ocean\_proximity\_NEAR OCEAN** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **17560** | -117.96 | 34.00 | 34.0 | 2777.0 | 540.0 | 1954.0 | 522.0 | 4.5163 | 183800.0 | 4 | 5.319923 | 0.194454 | 3.0 | True | False | False | False | False |
| **2045** | -122.81 | 38.54 | 12.0 | 2289.0 | 611.0 | 919.0 | 540.0 | 1.1553 | 139300.0 | 1 | 4.238889 | 0.266929 | 3.0 | True | False | False | False | False |
| **6383** | -122.25 | 37.77 | 52.0 | 2650.0 | 566.0 | 1468.0 | 567.0 | 3.0161 | 215700.0 | 3 | 4.673721 | 0.213585 | 2.0 | False | False | False | True | False |
| **4574** | -117.98 | 33.86 | 25.0 | 1025.0 | 266.0 | 726.0 | 183.0 | 3.8750 | 137500.0 | 3 | 5.601093 | 0.259512 | 3.0 | True | False | False | False | False |
| **15118** | -119.70 | 34.43 | 35.0 | 1402.0 | 369.0 | 654.0 | 385.0 | 2.6205 | 318800.0 | 2 | 3.641558 | 0.263195 | 3.0 | True | False | False | False | False |

[11]:

pd.get\_dummies(train\_set, columns**=**["ocean\_proximity"], dtype**=** int).head()

[11]:

|  | **longitude** | **latitude** | **housing\_median\_age** | **total\_rooms** | **total\_bedrooms** | **population** | **households** | **median\_income** | **median\_house\_value** | **income\_cat** | **rooms\_per\_house** | **bedrooms\_ratio** | **ordinal\_ocean** | **ocean\_proximity\_<1H OCEAN** | **ocean\_proximity\_INLAND** | **ocean\_proximity\_ISLAND** | **ocean\_proximity\_NEAR BAY** | **ocean\_proximity\_NEAR OCEAN** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **17560** | -117.96 | 34.00 | 34.0 | 2777.0 | 540.0 | 1954.0 | 522.0 | 4.5163 | 183800.0 | 4 | 5.319923 | 0.194454 | 3.0 | 1 | 0 | 0 | 0 | 0 |
| **2045** | -122.81 | 38.54 | 12.0 | 2289.0 | 611.0 | 919.0 | 540.0 | 1.1553 | 139300.0 | 1 | 4.238889 | 0.266929 | 3.0 | 1 | 0 | 0 | 0 | 0 |
| **6383** | -122.25 | 37.77 | 52.0 | 2650.0 | 566.0 | 1468.0 | 567.0 | 3.0161 | 215700.0 | 3 | 4.673721 | 0.213585 | 2.0 | 0 | 0 | 0 | 1 | 0 |
| **4574** | -117.98 | 33.86 | 25.0 | 1025.0 | 266.0 | 726.0 | 183.0 | 3.8750 | 137500.0 | 3 | 5.601093 | 0.259512 | 3.0 | 1 | 0 | 0 | 0 | 0 |
| **15118** | -119.70 | 34.43 | 35.0 | 1402.0 | 369.0 | 654.0 | 385.0 | 2.6205 | 318800.0 | 2 | 3.641558 | 0.263195 | 3.0 | 1 | 0 | 0 | 0 | 0 |

[12]:

train\_set\_bck **=** train\_set.copy()

train\_set **=** pd.get\_dummies(train\_set, columns **=** ["ocean\_proximity"], dtype **=** int)

Y con eso tendríamos preparadas las columnas categóricas. Es hora de "escalar" y convertir las distribuciones de las numéricas

**ML INTRO PRACTICA (IV): Preparación de Features Numéricas**

[1]:

*# Recuperemos lo hecho en la sesión anterior*

**import** matplotlib.pyplot **as** plt

**import** numpy **as** np

**import** pandas **as** pd

**import** seaborn **as** sns

**from** bootcampviztools **import** plot\_combined\_graphs

**from** sklearn.model\_selection **import** train\_test\_split

*# Código extra para los tamaños de las fuentes y los "ticks" de los ejes, esto no lo vimos al hablar de matplotlib así que una extra*

plt.rc('font', size**=**14)

plt.rc('axes', labelsize**=**14, titlesize**=**14)

plt.rc('legend', fontsize**=**14)

plt.rc('xtick', labelsize**=**10)

plt.rc('ytick', labelsize**=**10)

df\_inmo **=** pd.read\_csv("./data/ejemplo\_housing.csv")

train\_set, test\_set **=** train\_test\_split(df\_inmo, test\_size**=**0.2, random\_state**=**42)

train\_set["income\_cat"] **=** pd.cut(train\_set["median\_income"],

bins**=**[0., 1.5, 3.0, 4.5, 6., np.inf],

labels**=**[1, 2, 3, 4, 5])

train\_set["rooms\_per\_house"] **=** train\_set["total\_rooms"] **/** train\_set["households"]

train\_set["bedrooms\_ratio"] **=** train\_set["total\_bedrooms"] **/** train\_set["total\_rooms"]

features **=** ["ocean\_proximity", "income\_cat", "rooms\_per\_house","total\_rooms","housing\_median\_age","bedrooms\_ratio"]

target **=** "median\_house\_value"

Diagrama

Descripción generada automáticamente

Con escasas excepciones, la mayoría de los algoritmos de machine learning para construir modelos no funcionan bien cuando las variables y features numéricas tienen escalas y rangos de valores muy diferentes. Si no "escalamos" (igualamos escalas) lo que suele pasar es que lo algoritmos "pasan" de considerar las variables con menores rangos o valores más pequeños y se centran en las variables numéricas con rangos superiores.

Hay dos formas de "escalar" o asimilar los rangos: escalado min-max (*min-max scaling*) y la estandardización (*standardization*)

**1. Min-Max Scaling**

También se le conoce como normaización es el algoritmo más sencillo y lo que hace es transformar los rangos de las variables numéricas para que estos esten entre 0 y 1 (aunque el método que vamos a usar permite cambiar el rango, por ejemplo entre -1 y 1):

�^�=��−��������−����

[2]:

**from** sklearn.preprocessing **import** MinMaxScaler

features\_num **=** ["rooms\_per\_house","total\_rooms","housing\_median\_age","bedrooms\_ratio"]

min\_max\_scaler **=** MinMaxScaler(feature\_range**=**(**-**1, 1))

resultado\_min\_max **=** pd.DataFrame(min\_max\_scaler.fit\_transform(train\_set[features\_num]), columns**=** features\_num)

resultado\_min\_max

[2]:

|  | **rooms\_per\_house** | **total\_rooms** | **housing\_median\_age** | **bedrooms\_ratio** |
| --- | --- | --- | --- | --- |
| **0** | -0.932682 | -0.859032 | 0.294118 | -0.790101 |
| **1** | -0.949105 | -0.883858 | -0.568627 | -0.629047 |
| **2** | -0.942499 | -0.865493 | 1.000000 | -0.747589 |
| **3** | -0.928410 | -0.948161 | -0.058824 | -0.645528 |
| **4** | -0.958180 | -0.928982 | 0.333333 | -0.637343 |
| **...** | ... | ... | ... | ... |
| **16341** | -0.936574 | -0.947754 | -0.098039 | -0.815639 |
| **16342** | -0.947496 | -0.897746 | 0.372549 | -0.665564 |
| **16343** | -0.952376 | -0.757338 | -0.137255 | -0.609901 |
| **16344** | -0.948739 | -0.933510 | 0.058824 | -0.682322 |
| **16345** | -0.932478 | -0.985654 | 1.000000 | -0.558642 |

16346 rows × 4 columns

[3]:

resultado\_min\_max.describe()

[3]:

|  | **rooms\_per\_house** | **total\_rooms** | **housing\_median\_age** | **bedrooms\_ratio** |
| --- | --- | --- | --- | --- |
| **count** | 16346.000000 | 16346.000000 | 16346.000000 | 16346.000000 |
| **mean** | -0.931223 | -0.865835 | 0.082591 | -0.748855 |
| **std** | 0.034885 | 0.112708 | 0.494526 | 0.128182 |
| **min** | -1.000000 | -1.000000 | -1.000000 | -1.000000 |
| **25%** | -0.946083 | -0.926578 | -0.333333 | -0.831830 |
| **50%** | -0.934000 | -0.892201 | 0.098039 | -0.770856 |
| **75%** | -0.921663 | -0.840578 | 0.411765 | -0.689278 |
| **max** | 1.000000 | 1.000000 | 1.000000 | 1.000000 |

No igualamos todavía porque nos queda un detalle por ver sobre las distribuciones tipo *heavy-tail* (como las que hemos visto para precios, ingresos y demás en bastantes datasets)

**2. Standardization**

Hace la siguiente transformación para cada valor de una variable numérica X:

óáé��=�−�¯�� ,donde �� es la desviación estándar de la variable numérica

[4]:

**from** sklearn.preprocessing **import** StandardScaler

std\_scaler **=** StandardScaler()

resultado\_standardization **=** pd.DataFrame(std\_scaler.fit\_transform(train\_set[features\_num]),columns **=** features\_num)

resultado\_standardization.head()

[4]:

|  | **rooms\_per\_house** | **total\_rooms** | **housing\_median\_age** | **bedrooms\_ratio** |
| --- | --- | --- | --- | --- |
| **0** | -0.041816 | 0.060356 | 0.427749 | -0.321791 |
| **1** | -0.512619 | -0.159918 | -1.316896 | 0.934699 |
| **2** | -0.323245 | 0.003031 | 1.855185 | 0.009874 |
| **3** | 0.080636 | -0.730463 | -0.285969 | 0.806117 |
| **4** | -0.772764 | -0.560292 | 0.507051 | 0.869973 |

[5]:

resultado\_standardization.describe()

[5]:

|  | **rooms\_per\_house** | **total\_rooms** | **housing\_median\_age** | **bedrooms\_ratio** |
| --- | --- | --- | --- | --- |
| **count** | 1.634600e+04 | 1.634600e+04 | 1.634600e+04 | 1.634600e+04 |
| **mean** | 4.042608e-17 | 6.346460e-17 | 1.912632e-17 | 1.753970e-16 |
| **std** | 1.000031e+00 | 1.000031e+00 | 1.000031e+00 | 1.000031e+00 |
| **min** | -1.971583e+00 | -1.190419e+00 | -2.189218e+00 | -1.959350e+00 |
| **25%** | -4.259837e-01 | -5.389643e-01 | -8.410835e-01 | -6.473450e-01 |
| **50%** | -7.961065e-02 | -2.339439e-01 | 3.123857e-02 | -1.716490e-01 |
| **75%** | 2.740394e-01 | 2.240945e-01 | 6.656546e-01 | 4.647986e-01 |
| **max** | 5.536109e+01 | 1.655516e+01 | 1.855185e+00 | 1.364397e+01 |

En este caso lo que se busca es que la media sea 0 o casi y la desviación estándar 1 o casi. Este escalado no impone restricciones al rango y le afectan menos los outliers.

De nuevo no tocamos nada todavía... Veamos por qué:

**3. Transformado de distribuciones**

Si nos fijamos en distribuciones tipo "total\_rooms":

[6]:

train\_set["total\_rooms"].hist(bins **=** 40)

[6]:

<Axes: >

Gráfico, Histograma

Descripción generada automáticamente

Vemos ese efecto *heavy-tail* que hemos contemplado en otras distribuciones y que provoca que cuando escalemos los valores se van a concentrar en rangos muy pequeños:

[7]:

resultado\_min\_max["total\_rooms"].describe()

[7]:

count 16346.000000

mean -0.865835

std 0.112708

min -1.000000

25% -0.926578

50% -0.892201

75% -0.840578

max 1.000000

Name: total\_rooms, dtype: float64

[8]:

resultado\_min\_max.total\_rooms.quantile(0.90)

[8]:

-0.7626545251055603

Hemos concentrado el 90% de los valores entre -1 y -0.76, pero todavía quedan valores entre -0.76 y 1. Esto le sienta mal en general a los algoritmos de Machine Learning y por eso primero hacemos una transformación de las variables tipo "total\_rooms":

[9]:

*# extra code – this cell generates Figure 2–17*

fig, axs **=** plt.subplots(1, 2, figsize**=**(8, 3), sharey**=True**)

train\_set["total\_rooms"].hist(ax**=**axs[0], bins**=**50)

train\_set["total\_rooms"].apply(np.log).hist(ax**=**axs[1], bins**=**50)

axs[0].set\_xlabel("total rooms")

axs[1].set\_xlabel("Log of total rooms")

axs[0].set\_ylabel("Number of districts")

plt.show()

Gráfico, Gráfico de líneas

Descripción generada automáticamente

Y ahora sobre la transformada aplicaríamos el min\_max o el escalado:

[10]:

train\_set["log\_total\_room"] **=** train\_set["total\_rooms"].apply(np.log)

train\_set["total\_room\_tr"] **=** min\_max\_scaler.fit\_transform(train\_set[["log\_total\_room"]])

[11]:

train\_set["total\_room\_tr"].hist()

[11]:

<Axes: >

Gráfico, Histograma

Descripción generada automáticamente

**4. Aplicamos todas las transformaciones**

Primero veamos si es conveniente aplicar la transformación logarítmica en alguna otra variable:

[12]:

f\_num **=** ["rooms\_per\_house","total\_rooms","housing\_median\_age","bedrooms\_ratio"]

train\_set[f\_num].hist()

[12]:

array([[<Axes: title={'center': 'rooms\_per\_house'}>,

<Axes: title={'center': 'total\_rooms'}>],

[<Axes: title={'center': 'housing\_median\_age'}>,

<Axes: title={'center': 'bedrooms\_ratio'}>]], dtype=object)

Gráfico

Descripción generada automáticamente

Es necesario aplicarla a "bedrooms\_ratio", lo que podía ser de esperar porque depende de "total\_rooms". Pero también observamos que hay un outlier o varios en "rooms\_per\_house" que deberíamos tratar:

[13]:

train\_set.rooms\_per\_house.describe()

[13]:

count 16346.000000

mean 5.415940

std 2.296221

min 0.888889

25% 4.437817

50% 5.233142

75% 6.045176

max 132.533333

Name: rooms\_per\_house, dtype: float64

[14]:

plot\_combined\_graphs(train\_set, ["rooms\_per\_house"])

(2,)

Gráfico

Descripción generada automáticamente

Aplicaremos la transformación a todas ellas:

[15]:

**for** col **in** ["rooms\_per\_house","total\_rooms","bedrooms\_ratio"]:

train\_set[f"log\_{col}"] **=** train\_set[col].apply(np.log)

train\_set[col] **=** min\_max\_scaler.fit\_transform(train\_set[[f"log\_{col}"]])

train\_set["housing\_median\_age"] **=** min\_max\_scaler.fit\_transform(train\_set[["housing\_median\_age"]])

[16]:

f\_num **=** ["rooms\_per\_house","total\_rooms","housing\_median\_age","bedrooms\_ratio"]

train\_set[f\_num].hist()

[16]:

array([[<Axes: title={'center': 'rooms\_per\_house'}>,

<Axes: title={'center': 'total\_rooms'}>],

[<Axes: title={'center': 'housing\_median\_age'}>,

<Axes: title={'center': 'bedrooms\_ratio'}>]], dtype=object)

Gráfico, Histograma

Descripción generada automáticamente

[ ]:

*# Features finales*

features **=** ['ocean\_proximity\_<1H OCEAN', 'ocean\_proximity\_INLAND',

'ocean\_proximity\_ISLAND', 'ocean\_proximity\_NEAR BAY',

'ocean\_proximity\_NEAR OCEAN', "income\_cat", "rooms\_per\_house","total\_rooms","housing\_median\_age","bedrooms\_ratio"]

**ML INTRO PRACTICA (V): Modelado**

[1]:

*# Recuperemos lo hecho en la sesión anterior*

**import** matplotlib.pyplot **as** plt

**import** numpy **as** np

**import** pandas **as** pd

**import** seaborn **as** sns

**from** bootcampviztools **import** plot\_combined\_graphs

**from** sklearn.model\_selection **import** train\_test\_split

**from** sklearn.preprocessing **import** MinMaxScaler

*# Código extra para los tamaños de las fuentes y los "ticks" de los ejes, esto no lo vimos al hablar de matplotlib así que una extra*

plt.rc('font', size**=**14)

plt.rc('axes', labelsize**=**14, titlesize**=**14)

plt.rc('legend', fontsize**=**14)

plt.rc('xtick', labelsize**=**10)

plt.rc('ytick', labelsize**=**10)

df\_inmo **=** pd.read\_csv("./data/ejemplo\_housing.csv")

train\_set, test\_set **=** train\_test\_split(df\_inmo, test\_size**=**0.2, random\_state**=**42)

train\_set["income\_cat"] **=** pd.cut(train\_set["median\_income"],

bins**=**[0., 1.5, 3.0, 4.5, 6., np.inf],

labels**=**[1, 2, 3, 4, 5])

train\_set["rooms\_per\_house"] **=** train\_set["total\_rooms"] **/** train\_set["households"]

train\_set["bedrooms\_ratio"] **=** train\_set["total\_bedrooms"] **/** train\_set["total\_rooms"]

target **=** "median\_house\_value"

*# Procesado de variables categóricas*

train\_set **=** pd.get\_dummies(train\_set, columns **=** ["ocean\_proximity"], dtype **=** int)

*# Procesado de variables numéricas*

min\_max\_scaler **=** MinMaxScaler(feature\_range**=**(**-**1, 1))

**for** col **in** ["rooms\_per\_house","total\_rooms","bedrooms\_ratio"]:

train\_set[f"log\_{col}"] **=** train\_set[col].apply(np.log)

train\_set[col] **=** min\_max\_scaler.fit\_transform(train\_set[[f"log\_{col}"]])

train\_set["housing\_median\_age"] **=** min\_max\_scaler.fit\_transform(train\_set[["housing\_median\_age"]])

*# Features finales*

features **=** ['ocean\_proximity\_<1H OCEAN', 'ocean\_proximity\_INLAND',

'ocean\_proximity\_ISLAND', 'ocean\_proximity\_NEAR BAY',

'ocean\_proximity\_NEAR OCEAN', "income\_cat", "rooms\_per\_house","total\_rooms","housing\_median\_age","bedrooms\_ratio"]

Diagrama

Descripción generada automáticamente

Una vez con el dataset de train preparado es hora de escoger algoritmos que se ajusten al problema que tenemos, que recordemos es de regresión. Además, en general, no escogeremos un único algoritmo sino varios para poder encontrar el que mejor se adapte a las características concretas de nuestro problema.

Aunque lo hemos mencionado en la parte de evaluación que veremos en la siguiente sesión, este también es un buen momento para escoger una métrica de comparación y un sistema de validación de cuál de los modelos es mejor.

**1. Métrica de Evaluación**

Dedicaremos su sesión particular a las métricas, en este caso vamos a escoger como métrica una clásica en los problemas de regresión: El error cuadrático medio o MSE (Mean Squared Error).

MSE=1�∑�=1�(��−�^�)2

Donde �� es el valor real de cada zona inspeccionada (en este ejemplo, es decir de cada instancia o fila del datset), �^� es el valor que da el modelo y n el número de instancias que se han evaluado por el modelo.

Utilizaremos la función que nos proporciona sklearn:

[2]:

**from** sklearn.metrics **import** mean\_squared\_error

**2. Sistema de validación/compración: Cross-Validation**

El sitema que vamos a emplear para comparar los modelos creados con los algoritmos escogidos se denomina "validación cruzada" o cross-validation. Este es un método, que iremos viendo repetidas veces, y que es especialmente útil para comparar modelos y para evitar el problema del sobreajuste u overfitting [que veremos en la sesión en vivo]. Ahora mismo no hace falta que entremos en detalles. Lo veremos al comparar los dos modelos escogidos.

**3. Algoritmos o técnicas escogidas**

Aquí vamos a dar un salto de fe y vamos a escoger "Regresión Lineal" y "Arboles de Decisión para Regresión". No le des vueltas, cada uno tendrás su unidad particular. Simplemente mira como se crean los modelos:

[3]:

*# Regresor Lineal*

**from** sklearn.linear\_model **import** LinearRegression

lin\_reg **=** LinearRegression()

*# Arboles de decisión para Regresión*

**from** sklearn.tree **import** DecisionTreeRegressor

tree\_reg **=** DecisionTreeRegressor(random\_state **=** 42) *# Necesita una inicialización aleatoria y la semilla permite que siempre sea la misma*

**4. Entrenando los modelos**

[4]:

*# primero separamos la variable target del dataset de train y además solo nos quedamos con las features:*

X **=** train\_set[features].copy()

y**=** train\_set[target]

Y ahora entrenamos/creamos el modelo para cada algoritmo:

[5]:

lin\_reg.fit(X,y)

tree\_reg.fit(X,y)

[5]:

DecisionTreeRegressor

DecisionTreeRegressor(random\_state=42)

**5. Primera evaluación: Evaluación contra los propios datos de entrenamiento**

Aunque la fase de evaluación la contemplamos en la siguiente sesión, podemos hacer una primera evaluación sobre los mismos datos de entrenamiento. Es decir, generamos las predicciones que haría el modelo para los datos con los que ha entrenado y las comparamos con los datos reales.

Vamos a calcular el MSE de train:

[6]:

*# Primero las predicciones*

train\_pred\_lin **=** lin\_reg.predict(X)

train\_pred\_tree **=** tree\_reg.predict(X)

[7]:

train\_pred\_lin

[7]:

array([301056., 74240., 265728., ..., 167680., 153088., 53760.])

Y ahora MSE:

[8]:

mse\_lin **=** mean\_squared\_error(y, train\_pred\_lin, squared**=False**) *# En realidad damos la raíz cuadrada del error cuadrático medio*

*# que está en las mismas unidades que el target*

print(mse\_lin)

76782.74793065387

[9]:

mse\_tree **=** mean\_squared\_error(y, train\_pred\_tree, squared**=** **False**)

print(mse\_tree)

0.0

Uauhhh, el regresor de árboles lo clava!!!! Ya está tenemos un modelo perfecto... ¿o no? Lo veremos en la siguiente sesión.

**ML INTRO PRACTICA (VI): Evaluación y Ajuste**

[1]:

*# Recuperemos lo hecho en la sesión anterior*

**import** matplotlib.pyplot **as** plt

**import** numpy **as** np

**import** pandas **as** pd

**import** seaborn **as** sns

**from** bootcampviztools **import** plot\_combined\_graphs

**from** sklearn.model\_selection **import** train\_test\_split

**from** sklearn.preprocessing **import** MinMaxScaler

*# Código extra para los tamaños de las fuentes y los "ticks" de los ejes, esto no lo vimos al hablar de matplotlib así que una extra*

plt.rc('font', size**=**14)

plt.rc('axes', labelsize**=**14, titlesize**=**14)

plt.rc('legend', fontsize**=**14)

plt.rc('xtick', labelsize**=**10)

plt.rc('ytick', labelsize**=**10)

df\_inmo **=** pd.read\_csv("./data/ejemplo\_housing.csv")

train\_set, test\_set **=** train\_test\_split(df\_inmo, test\_size**=**0.2, random\_state**=**42)

train\_set["income\_cat"] **=** pd.cut(train\_set["median\_income"],

bins**=**[0., 1.5, 3.0, 4.5, 6., np.inf],

labels**=**[1, 2, 3, 4, 5])

train\_set["rooms\_per\_house"] **=** train\_set["total\_rooms"] **/** train\_set["households"]

train\_set["bedrooms\_ratio"] **=** train\_set["total\_bedrooms"] **/** train\_set["total\_rooms"]

target **=** "median\_house\_value"

*# Procesado de variables categóricas*

train\_set **=** pd.get\_dummies(train\_set, columns **=** ["ocean\_proximity"], dtype **=** int)

*# Procesado de variables numéricas*

min\_max\_scaler\_log **=** MinMaxScaler(feature\_range**=**(**-**1, 1))

min\_max\_scaler **=** MinMaxScaler(feature\_range**=**(**-**1, 1))

**for** col **in** ["rooms\_per\_house","total\_rooms","bedrooms\_ratio"]:

train\_set[f"log\_{col}"] **=** train\_set[col].apply(np.log)

train\_set[col] **=** min\_max\_scaler\_log.fit\_transform(train\_set[[f"log\_{col}"]])

train\_set["housing\_median\_age"] **=** min\_max\_scaler.fit\_transform(train\_set[["housing\_median\_age"]])

*# Features finales*

features **=** ['ocean\_proximity\_<1H OCEAN', 'ocean\_proximity\_INLAND',

'ocean\_proximity\_ISLAND', 'ocean\_proximity\_NEAR BAY',

'ocean\_proximity\_NEAR OCEAN', "income\_cat", "rooms\_per\_house","total\_rooms","housing\_median\_age","bedrooms\_ratio"]

X **=** train\_set[features].copy()

y**=** train\_set[target]

**from** sklearn.linear\_model **import** LinearRegression

lin\_reg **=** LinearRegression()

lin\_reg.fit(X,y)

**from** sklearn.tree **import** DecisionTreeRegressor

tree\_reg **=** DecisionTreeRegressor(random\_state **=** 42)

tree\_reg.fit(X,y)

**from** sklearn.metrics **import** mean\_squared\_error

Diagrama

Descripción generada automáticamente

Comenzamos donde lo dejamos en la sesión anterior con el asombroso resultado de que el decisor de árboles tenia un cero de error. Y ahí va la primera gran advertencia: si el modelo sale perfecto o muy bueno, desconfia. Algo extraño hay. En este caso en concreto lo que ha ocurrido es sencillo de explicar: Sobreajuste u *Overfitting*

**Overfitting**

El sobreajuste o overfitting es la capacidad de los modelos de adaptarse completamente a los datos de entrenamiento (de memorizarlos de alguna forma) de tal manera que el error de train es muy pequeño o muy inferior al error de validación o test.

Estos dos errores son los que se obtienen al aplicar el modelo al set de datos de Validación o de Test.

**Validación Cruzada**

Nosotros no hemos reservado datos para un set de validación porque hemos comentado que usaríamos validación cruzada. Y eso haremos. Básicamente vamos a reentrenar el los modelos con las dos técnicas elegidas con unos datasets de entrenamiento creados a partir del dataset completo de entrenamiento. Luego evaluaremos esos submodelos para cada ténica y la métrica será la media de las métricas para cada modelo.

El proceso es el siguiente:

1. Dividiremos el dataset de train en "k" grupos (depende del total de datos que tengamos).
2. Entrenaremos un modelo para cada combinación de k-1 grupos posibles (si tengo k = 5 por ejemplo tendré 5 posibles combinaciones, ver la tabla más abajo). Evaluaremos con el grupo restante.
3. La métrica a considerar será la media de las obtenidas en cada uno de las evaluaciones hechas.

| **Ronda** | **Fold 1** | **Fold 2** | **Fold 3** | **Fold 4** | **Fold 5** |
| --- | --- | --- | --- | --- | --- |
| 1 | Test | Training | Training | Training | Training |
| 2 | Training | Test | Training | Training | Training |
| 3 | Training | Training | Test | Training | Training |
| 4 | Training | Training | Training | Test | Training |
| 5 | Training | Training | Training | Training | Test |

Usaremos una función de python no tendrás que hacer nada en especial:

[2]:

**from** sklearn.model\_selection **import** cross\_val\_score

**Modelo construido con Regresión Lineal**

[3]:

*#### Error contra todo el train:*

train\_pred\_lin **=** lin\_reg.predict(X)

mse\_lin **=** mean\_squared\_error(y, train\_pred\_lin, squared**=False**) *# En realidad damos la raíz cuadrada del error cuadrático medio*

*# que está en las mismas unidades que el target*

print(mse\_lin)

76782.74793065387

[4]:

lin\_rmses **=** **-**cross\_val\_score(lin\_reg, X, y,

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

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

[4]:

count 10.000000

mean 76827.948320

std 2013.205984

min 73131.895637

25% 75828.759778

50% 77272.967328

75% 78368.576039

max 79041.653234

dtype: float64

Compara el error que teníamos de train con el obtenido mediante Cross Validation:

76780 ��. 76827

El error será mejor o peor en función de las expectativas de "California Real State Investment Funds" pero se puede esperar que generalice bien (es decir que haya poco overfitting). Veamos que tal los árboles.

**Modelo construido con Árboles de regresión**

[5]:

*#### Error contra todo el train:*

train\_pred\_tree **=** tree\_reg.predict(X)

mse\_tree **=** mean\_squared\_error(y, train\_pred\_tree, squared**=False**) *# En realidad damos la raíz cuadrada del error cuadrático medio*

*# que está en las mismas unidades que el target*

print(mse\_tree)

0.0

[6]:

tree\_rmses **=** **-**cross\_val\_score(tree\_reg, X, y,

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

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

[6]:

count 10.000000

mean 98229.945774

std 1731.305230

min 95557.734996

25% 97128.198619

50% 97647.697767

75% 99776.831156

max 100954.557331

dtype: float64

Comparando el error que teníamos de train con el obtenido mediante Cross Validation:

0 ��. 98229

No sólo generaliza mucho peor (el error de validacion es muy superior al de train, hay claramente overfitting) sino además podemos esperarnos que sea peor ese error.

**Escoger Modelo**

Viendo los resultados de la validación cruzada, escogeremos, para este caso **el modelo creado con regresión lineal** (repito, para este caso)

**Ajustar Modelo**

Este es el momento en el que podemos tocar los hiperparámetros del modelo e ir ajustándolos hasta llegar al modelo de regresión lineal (para nuestro problema) con las mejores prestaciones (siempre usando cross-validation)

Como los hiperparámetros de cada modelado o técnica son específicos, trataremos el ajuste de forma específica con cada uno.

**Evaluación contra Test:**

Lo primero que hay que hacer es aplicar las mismas transformaciones al Test que al train, además con rigor (sobre todo en los escalados, aunque aquí no lo vamos a hacer así exactamente):

[7]:

test\_set["income\_cat"] **=** pd.cut(test\_set["median\_income"],

bins**=**[0., 1.5, 3.0, 4.5, 6., np.inf],

labels**=**[1, 2, 3, 4, 5])

test\_set["rooms\_per\_house"] **=** test\_set["total\_rooms"] **/** test\_set["households"]

test\_set["bedrooms\_ratio"] **=** test\_set["total\_bedrooms"] **/** test\_set["total\_rooms"]

target **=** "median\_house\_value"

*# Procesado de variables categóricas*

test\_set **=** pd.get\_dummies(test\_set, columns **=** ["ocean\_proximity"], dtype **=** int)

*# Procesado de variables numéricas*

min\_max\_scaler **=** MinMaxScaler(feature\_range**=**(**-**1, 1)) *# ESTO NO ES FORMALMENTE CORRECTO, lo veremos en la sesión en vivo*

**for** col **in** ["rooms\_per\_house","total\_rooms","bedrooms\_ratio"]:

test\_set[f"log\_{col}"] **=** test\_set[col].apply(np.log)

test\_set[col] **=** min\_max\_scaler.fit\_transform(test\_set[[f"log\_{col}"]])

test\_set["housing\_median\_age"] **=** min\_max\_scaler.fit\_transform(test\_set[["housing\_median\_age"]])

*# Features finales*

features **=** ['ocean\_proximity\_<1H OCEAN', 'ocean\_proximity\_INLAND',

'ocean\_proximity\_ISLAND', 'ocean\_proximity\_NEAR BAY',

'ocean\_proximity\_NEAR OCEAN', "income\_cat", "rooms\_per\_house","total\_rooms","housing\_median\_age","bedrooms\_ratio"]

X\_test **=** test\_set[features].copy()

y\_test **=** test\_set[target]

Y ahora ya predecimos y evaluamos:

[8]:

test\_predict **=** lin\_reg.predict(X\_test)

mse\_lin\_test **=** mean\_squared\_error(y\_test, test\_predict, squared**=** **False**)

print(mse\_lin\_test)

77309.04488580789

Como esperábamos, aunque nos ha salido ligeramente superior al error obtenido con la validación cruzada es muy similar en test al de train. Esto es un caso de alto bias (error en train alto) y low variance (diferencia de error en train y test baja).