Big Data y Minería de Datos

Práctica 4. Aprendizaje supervisado para problemas de regresión

Autores
Mariano Albaladejo González
José Antonio Ruipérez Valiente
Manuel Jesús Gómez Moratilla

Grado en Gestión de Información y Contenidos Digitales

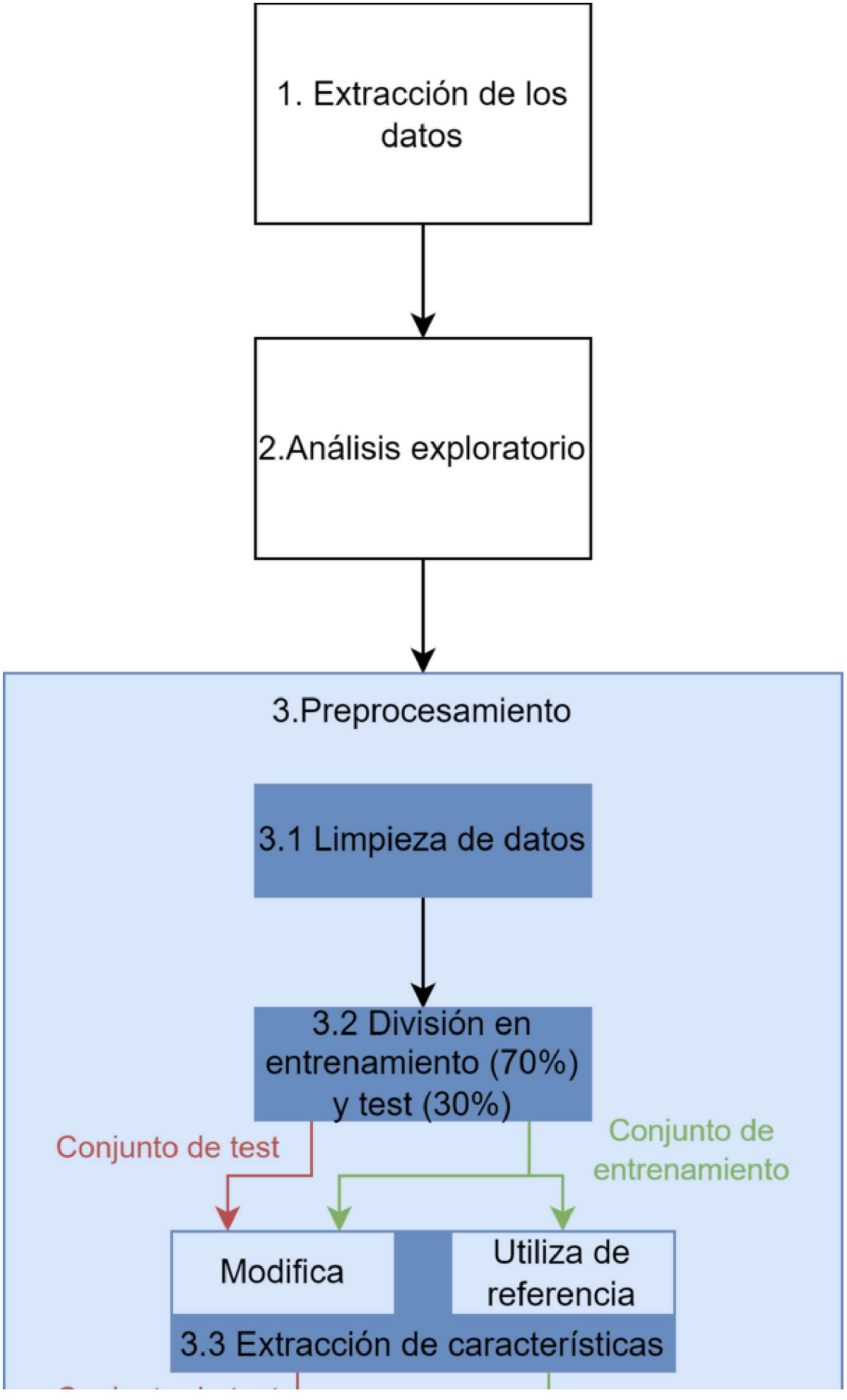
Big Data y Minería de Datos

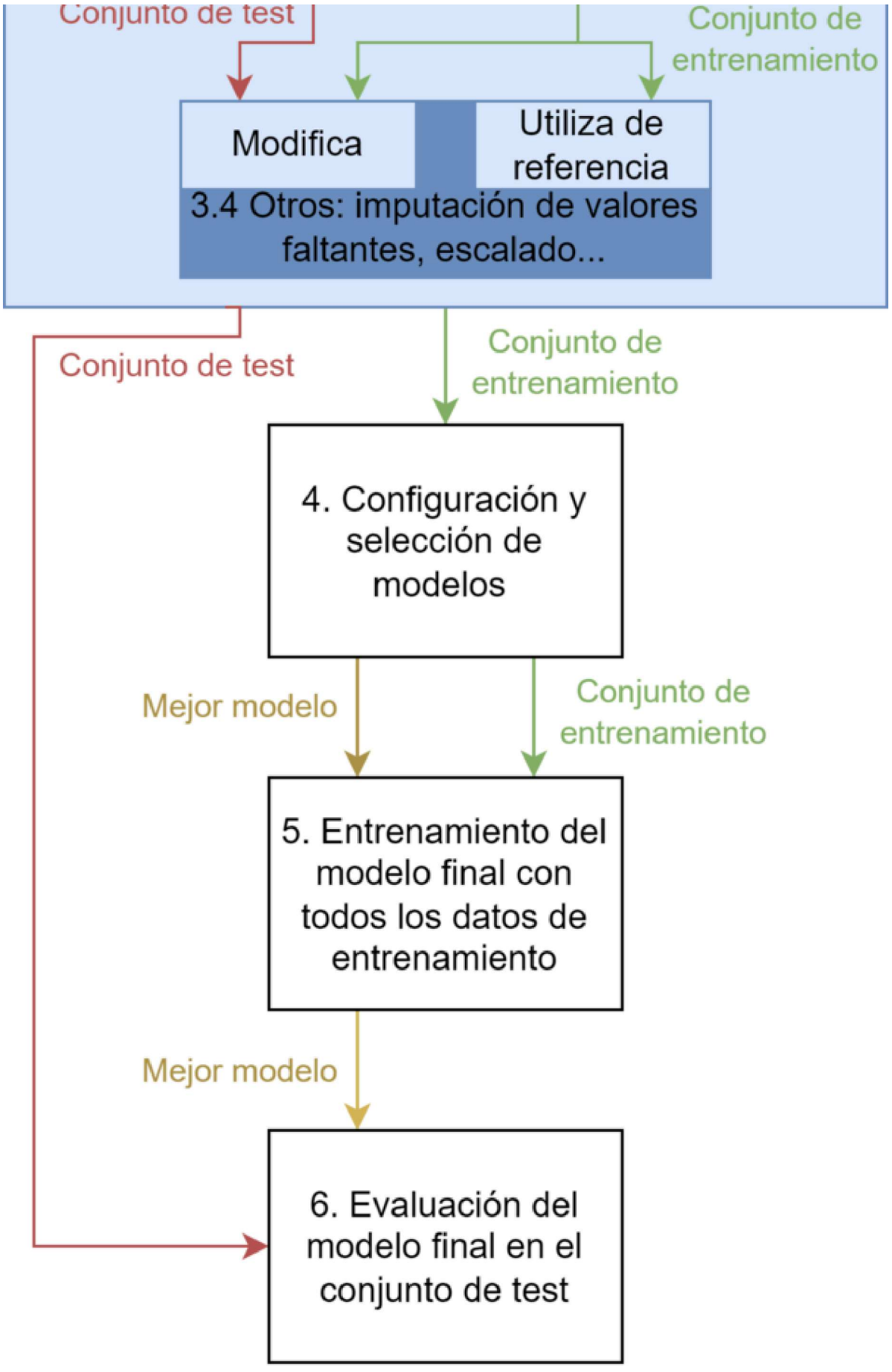
Práctica 4. Aprendizaje supervisado para problemas de regresión

Autores: Mariano Albaladejo González, José Antonio Ruipérez Valiente y Manuel Jesús Gómez Moratilla

Esta práctica consiste en una introducción al machine learning **supervisado** utilizando de ejemplo un problema de regresión. En los problemas de regresión nuestra variable a predecir será una variable numérica continua mientras que en los problemas de clasificación será una variable que represente categorías discretas.

Antes de comenzar revisa la metodología clásica de machine learning:


1. Extracción de los datos
2.Análisis exploratorio
3.Preprocesamiento
3.1 Limpieza de datos
3.2 División en entrenamiento (70%) y test (30%)
Conjunto de test
Conjunto de entrenamiento
Modifica
Utiliza de referencia
3.3 Extracción de características


Conjunto de test
Conjunto de entrenamiento
Modifica
Utiliza de referencia
3.4 Otros: imputación de valores faltantes, escalado...
Conjunto de test
Conjunto de entrenamiento
4. Configuración y selección de modelos
Mejor modelo
Conjunto de entrenamiento
5. Entrenamiento del modelo final con todos los datos de entrenamiento
Mejor modelo
6. Evaluación del modelo final en el conjunto de test

1.Extracción de datos

Este primer paso consiste en la extracción de datos para obtener el conjunto de datos con el que vamos a trabajar. Esto puede suponer desde cargar un fichero que tengamos almacenado a acceder a múltiples bases de datos para extraer nuestro conjunto de datos. En nuestro caso consistirá en descargar el fichero *housing.csv* con el conjunto de datos del aula virtual, guárdarlo en tu drive personal y acceder al conjunto de datos mediante la función `read_csv` de Pandas.

1.1. Descarga el fichero *housing.csv* del aula virtual y súbelo a tu cuenta de Google Drive

1.2. Monta el acceso a tus archivos de Google Drive a tu entorno de ejecución actual. Para ello, ejecuta la siguiente línea de código.

In [65]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.m
ount("/content/drive", force_remount=True).
```

Carga el dataframe utilizando la función `read_csv` de la librería Pandas. Debes introducir la ruta al fichero csv. Esta ruta tendrá el formato `/content/drive/MyDrive/RUTA_DE_CARPETAS_DE_TU_DRIVE/housing.csv`. También puedes guiarte con el explorador de archivos situado en la barra izquierda. Si tienes dudas consulta la práctica inicial de repaso.

In [66]:
```
import pandas as pd
```

In [67]:
```
df = pd.read_csv("/content/drive/MyDrive/df_MD/housing.csv")
```

2.Análsis exploratorio

El siguiente paso consiste en analizar y entender qué datos tenemos y qué información representa cada una de las columnas de nuestro conjunto de datos.

Primero vamos a mostrar el conjunto de datos. Para ello basta con introducir la variable que contiene nuestro conjunto de datos en una celda de código y ejecutarla.

In [68]:
```
df
```

Out[68]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	household
0	-122.23	37.88	41.0	880.0	129.0	322.0	126.(
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0	1138.(
2	-122.24	37.85	52.0	1467.0	190.0	496.0	177.(
3	-122.25	37.85	52.0	1274.0	235.0	558.0	219.(
4	-122.25	37.85	52.0	1627.0	280.0	565.0	259.(
...	...	...	...	...	...	...	..
20635	-121.09	39.48	25.0	1665.0	374.0	845.0	330.(
20636	-121.21	39.49	18.0	697.0	150.0	356.0	114.(
20637	-121.22	39.43	17.0	2254.0	485.0	1007.0	433.(
20638	-121.32	39.43	18.0	1860.0	409.0	741.0	349.(
20639	-121.24	39.37	16.0	2785.0	616.0	1387.0	530.(

20640 rows × 10 columns

En nuestro caso tenemos 20.640 entradas y 10 columnas.

Para entender más información sobre qué contiene nuestro conjunto de datos es necesario leer la documentación disponible. En nuestro caso, esta documentación se encuentra en el siguiente enlace: https://www.kaggle.com/datasets/camnugent/california-housing-prices/.

Aquí tienes una descripción de las variables:

1. longitude: medida de la distancia al oeste de una casa, un valor más alto significa que está más al oeste.
2. latitude: medida de la distancia al norte de una casa, un valor más alto significa que está más al norte.
3. housing_median_age: edad media de una casa dentro de un bloque, un número más bajo es un edificio más nuevo.
4. total_rooms: número total de habitaciones de un bloque.
5. total_bedrooms: número total de dormitorios de un bloque.
6. population: número total de personas que residen en un bloque.
7. households: número total de hogares, un grupo de personas que residen en una unidad doméstica, de un bloque.
8. median_income: mediana de los ingresos de los hogares de un bloque de viviendas (medida en decenas de miles de dólares estadounidenses).
9. median_house_value: valor medio de la vivienda de los hogares de un bloque (medido en dólares estadounidenses).
10. ocean_proximity: ubicación de la casa con respecto al océano/mar.

Podemos listar el nombre de las columnas mediante el atributo `variable_con_dataframe.columns`

In [69]:
```
df.columns
```

Out[69]:
```
Index(['longitude', 'latitude', 'housing_median_age', 'total_rooms',
       'total_bedrooms', 'population', 'households', 'median_income',
       'median_house_value', 'ocean_proximity'],
      dtype='object')
```

También podemos ver los tipos asociados a cada una de las columnas mediante el atributo `variable_con_dataframe.dtypes`

In [70]:
```
df.dtypes
```

Out[70]:

	0
longitude	float64
latitude	float64
housing_median_age	float64
total_rooms	float64
total_bedrooms	float64
population	float64
households	float64
median_income	float64
median_house_value	float64
ocean_proximity	object

dtype: object

Podemos obtener estadísticas básicas de todas las variables numéricas mediante la función `variable_con_dataframe.describe()`

In [71]:
```
df.describe()
```

Out[71]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000000	20640.000000
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553	1425.476744
std	2.003532	2.135952	12.585558	2181.615252	421.385070	1132.462122
min	-124.350000	32.540000	1.000000	2.000000	1.000000	3.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000	787.000000
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000	1166.000000
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000	1725.000000
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000	35682.000000

La variable *ocean_proximity* aparece del tipo "*object*" y en el describe no parece haberse mostrado. Esto puede indicar que no es una variable numérica. Vamos a mostrar esa columna para ver algunos ejemplos de sus valores posibles:

In [72]:
```
df["ocean_proximity"]
```

Out[72]:

	ocean_proximity
0	NEAR BAY
1	NEAR BAY
2	NEAR BAY
3	NEAR BAY
4	NEAR BAY
...	...
20635	INLAND
20636	INLAND
20637	INLAND
20638	INLAND
20639	INLAND

20640 rows × 1 columns

dtype: object

Podemos observar que esta variable contiene cadenas de caracteres. Para ver los posibles valores de las cadenas de caracteres y sus frecuencias podemos utilizar la función `variable_con_dataframe["nombre_variable"].value_counts()`

In [73]:
```
df["ocean_proximity"].value_counts()
```

Out[73]:

ocean_proximity	count
<1H OCEAN	9136
INLAND	6551
NEAR OCEAN	2658
NEAR BAY	2290
ISLAND	5

dtype: int64

Por último, vamos a detectar la presencia de valores faltantes, es decir, valores no rellenados en nuestro conjunto de datos. Es necesario solucionar de alguna forma estos datos faltantes ya que los modelos de IA no podrán ser ejecutados si en nuestro conjunto de datos tenemos valores faltantes.

Mediante la función df.isna() podemos conocer si un valor en una fila y columna es faltante o no.

In [74]:
```
df.isna()
```

Out[74]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	household
0	False	False	False	False	False	False	False
1	False	False	False	False	False	False	False
2	False	False	False	False	False	False	False
3	False	False	False	False	False	False	False
4	False	False	False	False	False	False	False
...	...	...	...	...	...	...	...
20635	False	False	False	False	False	False	False
20636	False	False	False	False	False	False	False
20637	False	False	False	False	False	False	False
20638	False	False	False	False	False	False	False
20639	False	False	False	False	False	False	False

20640 rows × 10 columns

Con el código que tienes a continuación podemos automatizar la detección de las columnas con valores faltantes y su recuento:

In [75]:
```
# Calculamos el número de valores faltantes por columna
na_por_columna = df.isna().sum()

# Filtramos las columnas con valores faltantes
columnas_con_na = na_por_columna[na_por_columna > 0]

# Mostramos las columnas con valores faltantes y el número de NA
for columna, na_count in columnas_con_na.items():
    print("Columna: "+str(columna)+ " Número de NA: "+str(na_count))
```

```
Columna: total_bedrooms Número de NA: 207
```

3.Preprocesamiento

La fase de preprocesamiento contiene todos los pasos necesarios para preparar nuestros datos para que puedan ser utilizados por nuestros modelos de machine learning.

Antes de comenzar con el preprocesamiento vamos a establecer una semilla de aleatoriedad para que cuando ejecutemos procesos en los que haya aleatoriedad siempre se generen los mismos números aleatorios. Lo correcto es almacenar la semilla de aleatoriedad en una variable, establecer la semilla de aleatoriedad al inicio del notebook y cada vez que vayamos a usar alguna función o método que tenga aleatoriedad debemos volver a establecer la semilla de aleatoriedad o introducir la semilla como un argumento más a la función (esto no siempre es posible).

In [76]:
```
SEMILLA_ALEATORIEDAD = 123
```

In [77]:
```
import numpy as np
```

In [78]:
```
np.random.seed(SEMILLA_ALEATORIEDAD)
```

3.1.Limpieza de datos

Este primer paso consiste en eliminar datos erróneos, columnas que no necesitemos, eliminar audios corruptos... Si es necesario aplicar alguna técnica de limpieza de datos debemos darnos cuenta en el análisis exploratorio.

En esta práctica no es necesario aplicar ninguna limpieza de datos.

3.2.División del conjunto de datos en entrenamiento y test

En este paso debemos dividir nuestro conjunto de datos de forma aleatoria en entrenamiento y test, para ello utilizaremos la función `train_test_split` de sklearn. Esta función recibe como primer argumento el conjunto de datos a dividir, en `test_size` la proporción de datos a dejar en test y en `random_state` la semilla de aleatoriedad a utilizar para que siempre que

ejecutemos esta línea de código se realice la misma división. Esta función nos devuelve primero el conjunto de entrenamiento y el conjunto de test.

In [79]:
```
from sklearn.model_selection import train_test_split
```

In [80]:
```
dataset_train, dataset_test = train_test_split(df, test_size=0.3, random_state=SEMILLA
```

3.3. Extracción de características

Esta fase se centra en extraer más variables (columnas) para añadirlas a nuestro conjunto de datos.

En esta práctica no necesitamos extraer características adicionales.

3.4.Eliminación de valores faltantes

Durante el preprocesamiento vimos que teníamos valores faltantes en la columna "total_bedrooms". Esto no siempre es así, pero en este conjunto de datos será necesario que solucionemos la presencia de valores faltantes.

La librería de sklearn nos ofrece diferentes formas de imputar los valores faltantes. Dentro de los imputadores, utilizaremos el algoritmo de imputación más sencillo llamado *SimpleImputer* que según el campo *strategy* podemos:

- *mean*: remplazamos los valores faltantes por la media de cada columna.
- *median*: reemplazamos los valores faltantes por la mediana de cada columna.
- *most_frequent*: reemplazamos los valores faltantes por el valor más frecuente.
- *constant*: remplazamos los valores faltantes por el valor que establezcamos en el argumento *fill_value*.

Existen otras opciones externas a esta librería como por ejemplo eliminar todas las filas con valores faltantes.

Referencias:

https://scikit-learn.org/stable/modules/classes.html#module-sklearn.impute

Lo importante a la hora de aplicar un imputador de valores faltantes es que se entrene o utilice de referencia datos del conjunto de entrenamiento y luego se aplique tanto al conjunto de entrenamiento y de test. A continuación, tienes un ejemplo sustituyendo los valores faltantes por la media. Esta media de cada columna es extraída de solo del conjunto de entrenamiento.

In [81]:
```
import numpy as np
from sklearn.impute import SimpleImputer
```

In [82]:
```
# Creamos el imputador a utilizar
imp_media = SimpleImputer(missing_values=np.nan, strategy='mean')
```

```
# Entrenamos el imputador con el conjunto de entrenamiento y con únicamente la columna
imp_media.fit(dataset_train[["total_bedrooms"]])

# Imputamos los valores faltantes tanto en el conjunto de entrenamiento como en el de
dataset_train[["total_bedrooms"]] = imp_media.transform(dataset_train[["total_bedrooms
dataset_test[["total_bedrooms"]] = imp_media.transform(dataset_test[["total_bedrooms"]
```

3.5.One-hot encoding

Durante el análisis exploratorio hemos visto que la variable *ocean_proximity* era categórica o cualitativa. Por ello, necesitamos transformar esta variable antes de poder introducirla en nuestros modelos. Existen dos tipos de variables categóricas:

- Variables categóricas ordinales: existe un orden entre las distintas categorías, por ejemplo, una variable categórica que representa evaluaciones y tome los valores "excelente", "bueno" y "malo". En este caso puede aplicarse un OneHotEncoding o asignarle directamente un valor numérico a cada categoría puesto que existe un orden entre estas.
- Variables categóricas nominales: no existe un orden entre las distintas categorías, por ejemplo, una variable que contenga marcas de coches, tipos de rotuladores o los colores de las flores. A estas variables solo puede aplicarse un OneHotEncoding.

La estrategia consiste en crear una columna para cada valor distinto que exista en la característica que estamos codificando y, para cada entrada de la base de datos, marcar con un 1 la columna a la que pertenezca dicha entrada y dejar las demás con 0. Aquí tienes un ejemplo:

id	color
1	red
2	blue
3	green
4	blue

One Hot Encoding

id	color_red	color_blue	color_green
1	1	0	0
2	0	1	0
3	0	0	1
4	0	1	0

A continuación tienes el código que realiza esta transformación a la variable "ocean_proximity". La transformación utiliza un *OneHotEncoder* de la librería de *sklearn*. De nuevo debemos utilizar de referencia los datos del conjunto de entrenamiento y aplicarlo tanto al conjunto de entrenamiento como al de test.

Referencias: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html

In [83]:
```
from sklearn.preprocessing import OneHotEncoder
```

In [84]:
```
# Creamos una instancia del OneHotEncoder
encoder = OneHotEncoder(sparse_output=False)
```

```
# Entrenamos el encoder con el conjunto de entrenamiento y la variable a codificar
encoder.fit(dataset_train[['ocean_proximity']])

# Transformos el conjunto de entrenamiento utilizando el encoder entrenado en el conju
train_encoded = encoder.transform(dataset_train[['ocean_proximity']])
train_encoded_df = pd.DataFrame(train_encoded, columns=encoder.get_feature_names_out([

# Ajustmos y transformamos el conjunto de test utilizando el encoder entrenado en el c
test_encoded = encoder.transform(dataset_test[['ocean_proximity']])
test_encoded_df = pd.DataFrame(test_encoded, columns=encoder.get_feature_names_out(['c

# Combinamos los DataFrames codificados con los conjuntos de datos originales
dataset_train_encoded = pd.concat([dataset_train.reset_index(drop=True), train_encoded
dataset_test_encoded = pd.concat([dataset_test.reset_index(drop=True), test_encoded_df

# Eliminamos la columna original 'ocean_proximity'
dataset_train_encoded.drop(['ocean_proximity'], axis=1, inplace=True)
dataset_test_encoded.drop(['ocean_proximity'], axis=1, inplace=True)
```

A continuación puedes ver el resultado final una vez aplicada la codificación.

In [85]:
```
dataset_train_encoded
```

Out[85]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	household
0	-116.46	33.82	6.0	4863.0	920.000000	3010.0	828.(
1	-117.04	34.00	21.0	4624.0	852.000000	2174.0	812.(
2	-121.03	37.55	32.0	946.0	198.000000	624.0	173.(
3	-117.80	33.68	8.0	2032.0	349.000000	862.0	340.(
4	-122.26	37.83	52.0	1656.0	420.000000	718.0	382.(
...	...	...	...	...	...	...	..
14443	-118.10	33.91	36.0	726.0	541.574843	490.0	130.(
14444	-117.24	33.37	14.0	4687.0	793.000000	2436.0	779.(
14445	-121.76	37.33	5.0	4153.0	719.000000	2435.0	697.(
14446	-122.44	37.78	44.0	1545.0	334.000000	561.0	326.(
14447	-119.08	36.21	20.0	1911.0	389.000000	1241.0	348.(

14448 rows × 14 columns

3.6. Otros

Además de las técnicas de preprocesamiento que hemos visto, con algunos conjuntos de datos o con algunos modelos es posible que nos veamos obligados a aplicar más técnicas de preprocesamiento. De nuevo, independientemente de la técnica y lo que haga recuerda siempre

que solo utilice de referencia los datos del conjunto de entrenamiento y se aplique tanto al conjunto de entrenamiento como al de test. Un ejemplo es el escalado de variables.

3.7. Selección de las variables predictoras

Este paso no es un paso como tal de una metodología de machine learning, sin embargo, es habitual que los modelos necesiten que les introduzcas de forma separada las variables predictores y la variable a predecir.

Lo habitual es utilizar la nomenclatura *x* para las variables predictores o variables independientes y la nomenclatura *y* para la variable dependiente o variable a predecir. A continuación, tienes un ejemplo de cómo hacer esta división.

In [86]:
```
train_x = dataset_train_encoded.loc[:, dataset_train_encoded.columns != "median_house_
train_y = dataset_train_encoded.loc[:, "median_house_value"]


test_x = dataset_test_encoded.loc[:, dataset_test_encoded.columns != "median_house_val
test_y = dataset_test_encoded.loc[:, "median_house_value"]
```

En nuestro caso, la variable a predecir es el valor medio de una vivienda (*median_house_value*). Al ser una variable cuantitativa (es numérica) continua tenemos un problema de regresión. Sin embargo, si la variable fuese cualitativa que tomase valores discretos sería un problema de clasificación.

4.Configuración y selección de modelos

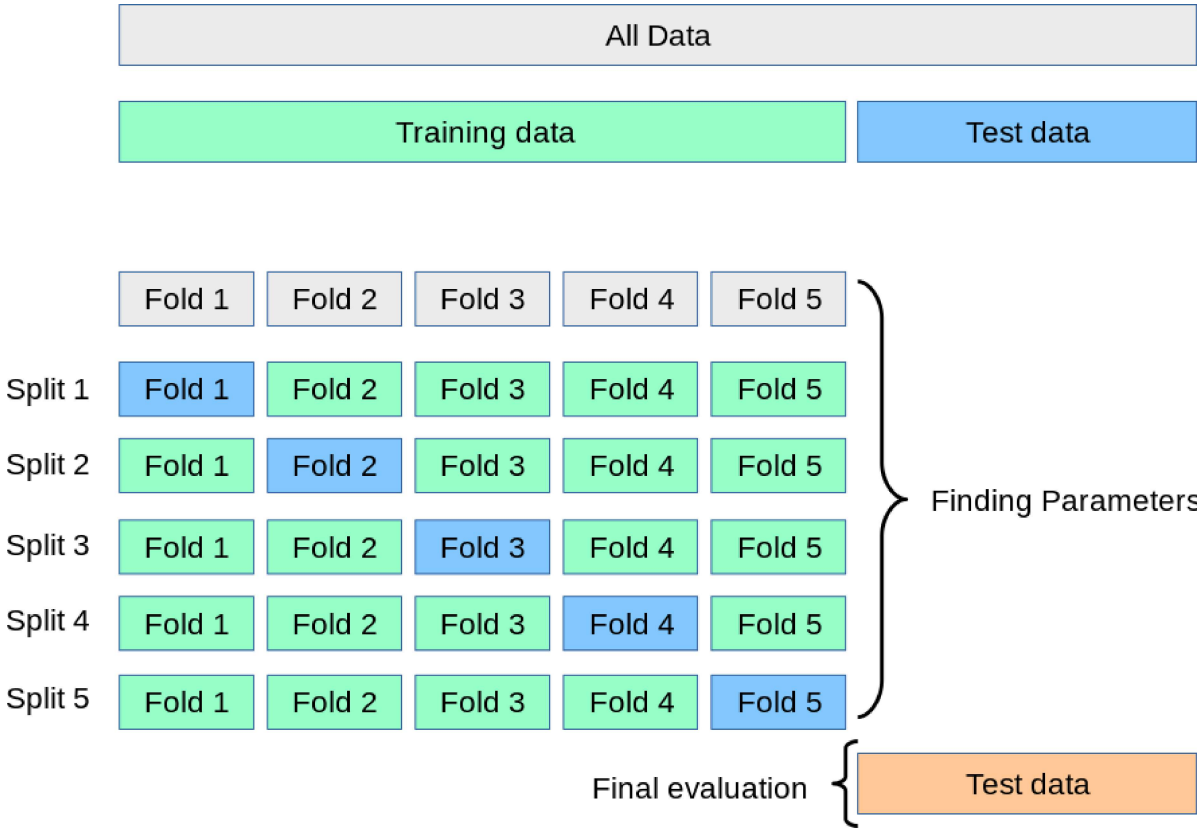
Este paso se centra en probar diferentes modelos y configuraciones (hiperparámetros) de cada modelo. El objetivo es que al final de esta fase se seleccione el mejor modelo con su mejor combinación de hiperparámetros. Una de las técnicas más comunes es realizar una búsqueda grid search (búsqueda en cuadrícula) que consiste en entrenar y evaluar todas las posibles combinaciones de hiperparámetros. Sin embargo, si queremos entrenar y evaluar nuestro modelo utilizando sólo el conjunto de entrenamiento será necesario realizar una nueva división del conjunto de entrenamiento en entrenamiento y evaluación, en este caso lo llamaremos entrenamiento y validación.

Sin embargo, como vamos a estar ejecutando un montón de combinaciones de hiperparámetros es habitual caer en unos hiperparámetros que funcionen bien "casi por casualidad" con un conjunto de validación específico. Por ello, en machine learning clásico es muy habitual de en lugar de dividir el conjunto de entrenamiento en entrenamiento y validación una vez, lo que hacemos es repetir esta división n veces, es decir, aplicaremos una validación cruzada.

La validación cruzada a N pliegues implica dividir el conjunto de datos en N particiones (pliegues) del mismo tamaño.

El proceso de validación cruzada a N pliegues suele seguir estos pasos:

1. Se divide el conjunto de datos en N particiones o pliegues.
2. Se utilizan N-1 de los pliegues para entrenar el modelo y se reserva un pliegue para evaluar el modelo. Este paso se repite N veces, haciendo que cada pliegue se utilice una vez como conjunto de validación.
3. Una vez realizada la validación cruzada tenemos N rendimientos en validación y lo que hacemos es calcular el rendimiento promedio en los pliegues.



¿Cómo evaluamos el rendimiento de los modelos? Para ello utilizaremos métricas de error. Las métricas de error nos indicarán cuánto se acerca o aleja los valores predichos por un modelo de los valores reales o esperados. A continuación, tienes tres métricas de error muy comunes para problemas de regresión (problemas en los que predecimos una variable cuantitativa continua), si la variable fuera cualitativa utilizaríamos métricas de clasificación.

yi es el valor real.

ŷi es la predicción del modelo para el valor yi.

n es el número total de observaciones en el conjunto de datos de evaluación.

- Error Cuadrático Medio (Mean Squared Error, MSE): Se calcula como la media de las diferencias al cuadrado entre las predicciones y los valores reales. MSE = (Σ(yi - ŷi)^2)/n

- Raíz del Error Cuadrático Medio (Root Mean Squared Error, RMSE): El RMSE es simplemente la raíz cuadrada del MSE. Proporciona una medida del error en la misma unidad que los datos originales. RMSE = √MSE
- Error Absoluto Medio (Mean Absolute Error, MAE): El MAE mide la magnitud promedio de los errores sin tener en cuenta su dirección (positiva o negativa). MAE = (Σ|yi - ŷi|)/n

A continuación, tienes un ejemplo para aplicar una búsqueda de hiperparámetros a un modelo llamado K-Vecinos. Este modelo se basa fundamentalmente en la utilización de ejemplos "vecinos" al dato que hay que procesar (por lo tanto, la distancia entre cada ejemplo). En el código se configuran dos hiperparámetros:

- n_neighbors: número de vecinos
- weights: peso aplicado a los vecinos (relevancia de los vecinos)

En la búsqueda de hiperparámetros se realiza una validación cruzada a 5 pliegues calculando MSE, RMSE y el MAE, siendo el MAE la métrica a utilizar para elegir el mejor modelo. Para la búsqueda de hiperparámetros utilizaremos la función GridSearchCV de sklearn.

Referencias:

https://scikit-learn.org/1.5/modules/generated/sklearn.neighbors.KNeighborsRegressor.html

https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

In [87]:
```
from sklearn.neighbors import KNeighborsRegressor
from sklearn.model_selection import GridSearchCV
```

In [88]:
```
# Definimos la métrica de error a optimizar
OPT = "MAE"

# Crear un modelo KNN Regressor
model = KNeighborsRegressor()

# Definimos la grid search con los hiperparámetros que deseamos explorar
param_grid = {
    'n_neighbors': [3, 6, 9, 12, 15, 18, 21],  # Número de vecinos
    'weights': ['uniform', 'distance'],  # Peso aplicado a los vecinos
}

# Realizamos la búsqueda de hiperparámetros utilizando validación cruzada
grid_search = GridSearchCV(model, param_grid, cv=5, refit=OPT, n_jobs=-1,
                           scoring={"MAE": "neg_mean_absolute_error",
                                    "RMSE": "neg_root_mean_squared_error",
                                    "MSE": "neg_mean_squared_error"
                                    }
                           )

np.random.seed(SEMILLA_ALEATORIEDAD)
grid_search.fit(train_x, train_y)

# Imprimimos los mejores hiperparámetros y su rendimiento
```

```
print("Mejores hiperparámetros:", grid_search.best_params_)
print("Mejor puntuación (" + OPT + "):", -grid_search.best_score_)
```

```
Mejores hiperparámetros: {'n_neighbors': 21, 'weights': 'distance'}
Mejor puntuación (MAE): 74609.01120088436
```

También podemos mostrar estos rendimientos en un dataframe:

In [89]:
```
model_gscv_df = pd.DataFrame(grid_search.cv_results_).sort_values(by="mean_test_"+OPT,

model_gscv_df["mean_test_MAE"] = -model_gscv_df["mean_test_MAE"]
model_gscv_df["mean_test_MSE"] = -model_gscv_df["mean_test_MSE"]
model_gscv_df["mean_test_RMSE"] = -model_gscv_df["mean_test_RMSE"]


model_gscv_df[["params", "mean_test_MAE", "mean_test_MSE", "mean_test_RMSE"]]
```

Out[89]:

	params	mean_test_MAE	mean_test_MSE	mean_test_RMSE
0	{'n_neighbors': 21, 'weights': 'distance'}	74609.011201	9.128833e+09	95542.878640
1	{'n_neighbors': 18, 'weights': 'distance'}	74618.354100	9.147443e+09	95639.961277
2	{'n_neighbors': 15, 'weights': 'distance'}	74703.142120	9.185759e+09	95840.315772
3	{'n_neighbors': 12, 'weights': 'distance'}	74898.459655	9.251505e+09	96183.688441
4	{'n_neighbors': 18, 'weights': 'uniform'}	74995.347436	9.244599e+09	96146.405522
5	{'n_neighbors': 15, 'weights': 'uniform'}	75023.050070	9.268683e+09	96271.744559
6	{'n_neighbors': 21, 'weights': 'uniform'}	75032.109009	9.236173e+09	96102.896106
7	{'n_neighbors': 12, 'weights': 'uniform'}	75186.513461	9.317245e+09	96524.830009
8	{'n_neighbors': 9, 'weights': 'distance'}	75278.879264	9.399795e+09	96951.102199
9	{'n_neighbors': 9, 'weights': 'uniform'}	75481.077188	9.441201e+09	97164.224885
10	{'n_neighbors': 6, 'weights': 'distance'}	76152.655340	9.722395e+09	98600.864450
11	{'n_neighbors': 6, 'weights': 'uniform'}	76223.768337	9.742037e+09	98700.163055
12	{'n_neighbors': 3, 'weights': 'uniform'}	79732.168473	1.086179e+10	104217.512474
13	{'n_neighbors': 3, 'weights': 'distance'}	79793.525861	1.090922e+10	104444.336031

In [90]:
```
model_gscv_df.columns
```

Out[90]:
```
Index(['mean_fit_time', 'std_fit_time', 'mean_score_time', 'std_score_time',
       'param_n_neighbors', 'param_weights', 'params', 'split0_test_MAE',
       'split1_test_MAE', 'split2_test_MAE', 'split3_test_MAE',
       'split4_test_MAE', 'mean_test_MAE', 'std_test_MAE', 'rank_test_MAE',
       'split0_test_RMSE', 'split1_test_RMSE', 'split2_test_RMSE',
       'split3_test_RMSE', 'split4_test_RMSE', 'mean_test_RMSE',
       'std_test_RMSE', 'rank_test_RMSE', 'split0_test_MSE', 'split1_test_MSE',
       'split2_test_MSE', 'split3_test_MSE', 'split4_test_MSE',
       'mean_test_MSE', 'std_test_MSE', 'rank_test_MSE'],
      dtype='object')
```

In [91]:
```
grid_search.best_params_
```

Out[91]:
```
{'n_neighbors': 21, 'weights': 'distance'}
```

Es posible que hayas observado que a los resultados les ha cambiado el signo, esto es porque sklearn optimiza los errores maximizando y nuestro objetivo es minimiza los errores, por ello, sklearn calcula una versión negativa del RMSE, MSE y MSE a la que debemos invertir el signo. Un ejemplo es la línea de código: `-grid_search.best_score_`

5.Entrenamiento del mejor modelo

A continuación entrenamos la configuración y modelo que obtuvo mejor rendimiento en la búsqueda de hiperparámetros con todo el conjunto de entrenamiento, sin aplicar validaciones cruzadas. Para ello es necesario que obtengamos los hiperparámetros del mejor modelo de la búsqueda de hiperparámetros mediante `grid_search.best_params_`

In [92]:
```
# Creamos un modelo con los hiperparámetros de la mejor configuración de la búsqueda a
best_model = KNeighborsRegressor(**grid_search.best_params_)

# Entrenamos el modelo con todo el conjunto de datos
np.random.seed(SEMILLA_ALEATORIEDAD)
best_model.fit(train_x, train_y)
```

Out[92]:

KNeighborsRegressor

```
KNeighborsRegressor(n_neighbors=21, weights='distance')
```

6.Evaluación del mejor modelo en test

Para calcular el error en test con el modelo ya entrenado utilizamos la función `mi_modelo.predict()` introduciéndole de los datos de test sólo las variables predictoras. Una vez obtenidas las predicciones podemos utilizar las funciones que ofrece sklearn para calcular las métricas de error. A estas funciones debemos introducirlos los valores reales y los valores predichos.

Referencias:

https://scikit-learn.org/stable/modules/classes.html#module-sklearn.metrics

In [93]:
```
from sklearn.metrics import mean_squared_error, mean_absolute_error
from math import sqrt
```

In [94]:
```
# Obtenemos las predicciones del modelo en test
predictions = best_model.predict(test_x)

# Calculamos el MSE, RMSE y MAE en el conjunto de test a partir de las predicciones de
mse = mean_squared_error(test_y, predictions)
rmse = sqrt(mse)
mae = mean_absolute_error(test_y, predictions)
```

```
print("MSE en test: "+str(mse))
print("RMSE en test: "+str(rmse))
print("MAE en test: "+str(mae))
```

```
MSE en test: 9077236205.1214
RMSE en test: 95274.53072632477
MAE en test: 74794.27019410505
```

Podemos reducir el código necesario para entrenar el mejor modelo de la búsqueda de hiperparámetros con todos los datos de entrenamiento y predecir su rendimiento en test, es decir, los pasos 4 y 5. Al haber puesto GridSearchCV() el atributo refit=OPT además de servir para que la función conozca cómo ordenar los resultados, también deja entrenado al mejor modelo con el 100% de los datos de entrenamiento y solo hace falta obtenerlo del atributo `mi_grid_search.best_estimator_`. Aquí tienes un ejemplo:

In [95]:
```
best_model = grid_search.best_estimator_

# Calculamos MSE, RMSE y MAE en el conjunto de test con el mejor modelo ya entrenado
predictions = best_model.predict(test_x)

mse = mean_squared_error(test_y, predictions)
rmse = sqrt(mse)
mae = mean_absolute_error(test_y, predictions)

print("MSE en test: "+str(mse))
print("RMSE en test: "+str(rmse))
print("MAE en test: "+str(mae))
```

```
MSE en test: 9077236205.1214
RMSE en test: 95274.53072632477
MAE en test: 74794.27019410505
```

En ocasiones también es habitual calcular el error en el 100% del conjunto de entrenamiento. Es decir, ver el rendimiento con los datos que ha visto durante su entrenamiento. Aquí tienes un ejemplo, la diferencia es que en lugar de predecir el conjunto de test predice el conjunto de entrenamiento.

In [96]:
```
best_model = grid_search.best_estimator_

# Calculamos MSE, RMSE y MAE en el conjunto de entrenamiento utilizando el mejor model
predictions = best_model.predict(train_x)

mse = mean_squared_error(train_y, predictions)
rmse = sqrt(mse)
mae = mean_absolute_error(train_y, predictions)

print("MSE en entrenamiento: "+str(mse))
print("RMSE en entrenamiento: "+str(rmse))
print("MAE en entrenamiento: "+str(mae))
```

```
MSE en entrenamiento: 0.0
RMSE en entrenamiento: 0.0
MAE en entrenamiento: 0.0
```

¡Cuidado! No confundas el error de entrenamiento del modelo del paso 5 con los errores en la búsqueda de hiperparámetros del paso 4. Piensa que en el paso 4 hemos aplicado una

validación cruzada, por ello, no se ha entrenado el modelo con el 100% de los datos de entrenamiento. Además, el error que reportamos en el paso 4 es el error promedio de las validaciones cruzadas (porque repetíamos el proceso n veces, siendo n el número de pliegues).

Ejercicios para casa

1.1 Modifica los hiperparámetros de K-Vecinos añadiendo más opciones en el hiperparámetro n_neighbors.

1.2 Realiza la búsqueda de hiperparámetros de otro modelo, concretamente de un regresor lineal que en sklearn recibe el nombre LinearRegression. Este regresor lineal sólo tiene un único hiperparámetro que es *fit_intercept*. Este hiperaprámetro toma los valores *True* o *False*. Compara los resultados con K-Vecinos. En el caso en el que el regresor lineal haya obtenido un mejor rendimiento en las validaciones cruzadas de la búsqueda de hiperparámetros cambia de mejor modelo y predice con este modelo el conjunto de test.

1.3 Calcula en test la métrica de error r2_score.

Referencias:

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html

https://scikit-learn.org/stable/modules/generated/sklearn.metrics.r2_score.html#sklearn.metrics.r2_score