# **Capítulo 2. Proyecto End-to-End**

* **Introducción**

En este capítulo se trabajará un ejemplo de un proyecto de machine Learning (ML), tal y como lo haría un científico de datos real.

En este caso vamos a fingir que somos gente que trabaja en el departamento de planeación e infraestructura de Londres, Reino Unido.

Para empezar, estos son los pasos que se deben de seguir:

1. Ver el panorama completo
2. Obtener los datos
3. Descubrir y visualizar los datos para tener una mejor perspectiva
4. Preparar los datos para los algoritmos de Machine Learning
5. Seleccionar un modelo y entrenarlo
6. Afinar el modelo
7. Presentar una solución
8. Correr, monitorear y mantener el sistema.

* **Trabajar con Datos de la Vida Real**

Cuando se está aprendiendo Machine Learning siempre es una buena idea practicar con datos extraídos de situaciones reales, y no con sets de datos artificiales. Afortunadamente hay una gran lista de sitios web y repositorios disponibles, donde se pueden descargar miles de data sets. Estos son algunos de los más populares:

* [Kaggle](https://www.kaggle.com/datasets)
* [Banco Mundial](https://data.worldbank.org/)
* [US Census](https://data.census.gov/cedsci/)
* [NYC Open Data](https://opendata.cityofnewyork.us/)
* [Amazon AWS](https://registry.opendata.aws/)
* [Nasdaq](https://data.nasdaq.com/)
* [INEGI](https://www.inegi.org.mx/datos/)
* [London Datastore](https://data.london.gov.uk/)

Para este proyecto vamos a usar el set de datos Housing in London, el cual se puede descargar de los servidores de Kaggle con este [link](https://www.kaggle.com/datasets/justinas/housing-in-london). Este data set fue actualizado por última vez en el año 2020, por lo que es bastante reciente y se podrían comparar nuestros resultados finales, con los actuales en la ciudad de Londres.

* **Ver el panorama completo**

Para empezar el proyecto imaginemos que te piden que tu primera tarea como científico de datos es construir un modelo de precios para las viviendas en Londres. Nuestros datos incluyen métricas como la población, la media salarial, los precios de las viviendas, entre otros.

El data set nos da información que puede ser divida en grupos geográficos de Londres, en este caso el grupo más pequeño serían los diferentes distritos municipales o igualmente conocidos como municipios de Londres. Por lo que el modelo debería poder aprender de estos datos y poder predecir la media de costos de vivienda en cada municipio.

[](https://www.britain-visitor.com/london/london-boroughs)

* **Formular el Problema**

Como científico de datos podrías preguntarte, ¿Cómo es que mi modelo puede ser de utilidad para mi organización y sacar algún beneficio? Sabiendo que el objetivo es importante y que este determinara como se va a formular el problema, que algoritmos usar, con qué medida de desempeño evaluar el modelo, y cuánto tiempo puedes dedicar a ajustarlo.

Para conseguir este tipo de respuestas, supongamos que tu jefe te indica que a tu modelo de predicción de la media de precios para las viviendas se le va a introducir otro sistema de Machine Learning, por lo que junto con otros datos este sistema nuevo podría determinar si vale la pena invertir en cierto municipio de Londres o no. Sabiendo que este sistema sería de beneficio a la organización, ya que tendrá un impacto directo en sus ganancias.

Si quisiéramos visualizar como funcionaria este conjunto de sistemas de Machine Learning propuesto por el jefe, se vería de esta manera:

Diagram

Description automatically generated

A este tipo de gráficos en el mundo del ML, se les conoce como pipeline, el cual consiste en una secuencia de componentes para el proceso de datos. Cada componente recibe cierta cantidad de datos y produce otro set de datos como resultado, después este nuevo set de datos es procesado por el siguiente componente y así sucesivamente.

Después de saber qué es lo que necesita la organización, podrías preguntarte ¿Cuál es la situación actual en cuanto a la estimación de la media de precios por vivienda? Esto debido a que la situación actual podría darte una referencia del desempeño convencional y algunas ideas sobre cómo resolver el problema. Supongamos que investigas la respuesta y encuentras que, actualmente los precios son estimados por expertos que reúnen información manualmente de los municipios, y cuando no la pueden obtener usan un complejo sistema de reglas para estimarla.

Por lo tanto, te enteras de que el método actual es costoso y toma bastante tiempo, además estas estimaciones no son muy buenas, se ha demostrado que en diversas ocasiones sus predicciones estaban alejadas de la realidad por más de un 25%. En ese momento te das cuenta porque tu proyecto es tan importante para la organización y cómo afectaría en sus ganancias si funciona.

Con toda esta información obtenida, llega el momento de empezar a diseñar tu sistema, por lo que tienes que formular el problema, por ejemplo, ¿será un algoritmo supervisado o no supervisado?, ¿será una regresión, una clasificación una red neuronal o algo más?, ¿el algoritmo tendrá aprendizaje online o aprendizaje batch? Como científico de datos, deberías ser capaz de contestar estas preguntas para tener una idea clara de cómo empezar a trabajar con los datos.

Para contestar estas preguntas tenemos que analizar nuestros datos, al hacer esto se puede ver claramente que los datos están *etiquetados o clasificados* y por lo tanto tienen una categoría en específico, dándonos a entender que este será un algoritmo de aprendizaje supervisado (recordando que por ejemplo en un algoritmo no supervisado, el algoritmo clasifica los datos por sí mismo).

También sabemos que la principal función de nuestro modelo será predecir valores futuros, para lo que comúnmente se usa una regresión. Y como vimos el capítulo anterior una regresión lineal, contiene la variable a predecir Y, y la variable independiente X. en este caso usaremos diferentes variables para hacer la predicción (población, media de ingresos, entre otras), por lo que este modelo se tendría que hacer con una ***regresión múltiple***.

Igualmente será una ***regresión univariante***, ya que solo estamos intentando encontrar un valor por municipio. Si quisiéramos predecir más de un valor o variable, entonces estaríamos hablando de una ***regresión multivariante***.

Finalmente, como no hay una necesidad especifica de ajustar o cargar nuevos datos muy rápidamente, y no habrá un flujo de datos continuo para el sistema (sabiendo que los censos gubernamentales actualizan estas bases de datos en prolongados lapsos de tiempo). El tipo de ***aprendizaje batch*** se puede ajustar al modelo bastante bien.

* **Seleccionar una Medida de Desempeño**

El siguiente paso es seleccionar una medida de desempeño. Una de las medidas más populares para una regresión es el llamado error cuadrático medio o también conocido por sus siglas en inglés RMSE (root mean square error). El RMSE nos da una idea de cuanto error tiene nuestro sistema a la hora de hacer sus predicciones, dándole un mayor peso a los errores más grandes. La fórmula matemática para el RMSE es la siguiente:

A picture containing text

Description automatically generated

* Donde N es el número de muestras que se están evaluando, por ejemplo, si tenemos 200 municipios, entonces N = 200.
* **x** es un vector que contiene valores de todas las variables que se están usando para la regresión, por ejemplo, población, media de ingresos, entre otras. Cuando y es el valor de la etiqueta deseada, en este caso, la media de precio para las viviendas. Por ejemplo, se tiene un municipio con una población de 8500, un ingreso medio de 9600 y un costo de vivienda medio de 3200, entonces:

Text

Description automatically generated

* **X** es una matriz conteniendo las mismas instancias que **x**, pero de manera transpuesta, por lo que X = (x)T.



* **q** es conocida como la función de predicción o hipótesis y se usa para complementar la predicción deseada predicción = q(x)., por ejemplo, supongamos que el modelo predice una media de costo de vivienda de 3550, entonces Y = q(x) = 3550.

Cabe mencionar que, aunque el RMSE es la medida de desempeño preferida para muchos, en algunos casos puede ser aconsejable usar otra función, por ejemplo, si se tiene un gran número de municipios en condiciones fuera de lo normal o outliers, será mejor utilizar el error absoluto medio o MAE (mean absolute error).

Text

Description automatically generated

Tanto el RMSE como el MAE son formas de medir la distancia entre 2 vectores, el vector de predicciones y el vector de valores deseados. Para esto se pueden aplicar varias medidas de distancia o normas:

* A la suma de cuadrados o RMSE le corresponde le corresponde la norma euclidiana, la cual consiste en una serie de reglas para operaciones con vectores y se puede señalizar de esta manera ||-||2. (Si no se recuerdan estas reglas o se quieren repasar, este [video](https://www.youtube.com/watch?v=HXG0XtM1kmM) lo plasma de manera sencilla).
* La suma de los absolutos o MAE utiliza la conocida norma Manhattan para las operaciones de vectores, también señalizada de esta manera ||-||1 (para repasar los fundamentos de esta norma se recomienda ir a este [enlace](https://iq.opengenus.org/manhattan-distance/)).
* En resumen, la norma ***k*** de un vector ***v*** que contiene ***n*** elementos se puede definir 
* Gracias a estas propiedades se puede inferir que entre mayor sea el índice de la norma, más será su enfoque en los valores grandes, es por eso por lo que un RMSE con índice 2, se vería más afectado que un MAE con índice 1 al tener outliers.
* **Validar los Supuestos**

Antes de empezar a trabajar con los datos, es una buena práctica validar todos los supuestos que se están tomando por hecho. Por ejemplo, se asume que se espera que el resultado de las predicciones del modelo esté en formato numérico. Pero qué tal si el segundo componente del pipeline de la organización, el cual este encargado de evaluar las inversiones espera que los datos de entrada sean categóricos, por ejemplo, en vez de necesitar el valor de una casa, solo saber si es caro, normal o barato. Para este tipo de salida nos vendría mejor un modelo de clasificación, no uno de regresión.

Después de haber validado esto con tu jefe o tu equipo de trabajo, y todas aquellas suposiciones de las que no se esté muy seguro, es tiempo de pasar al manejo de datos.

* **Obtener los Datos**

Aquí es donde empieza la parte emocionante, ya que es la hora de tomar la computadora y empezar a programar, para este proyecto utilizaremos Jupyter Notebook.

* **Crear un Espacio de Trabajo**

Para el proyecto hay que asegurarse de tener las librerías principales, que en este caso son: pandas, numpy, matplotlib y scikit-learn.

Si aún no se tienen, para instalarlos se tiene que ejecutar el siguiente comando en Jupyter:

pip install pandas numpy matplotlib scikit-learn

* **Descargar los datos**

Una vez teniendo nuestro espacio de trabajo en Jupyter Notebook listo, vamos a descargar la base de datos con la que trabajaremos.

Podrías descargar un CSV desde la página web y luego cargarla a Python desde nuestra propia computadora, aunque esto no es recomendable porque no es automático.

Lo recomendable seria que nuestro algoritmo descargara esta base de datos directamente del sitio web sin tener que entrar nosotros. Esto también ayuda si son varias las personas que necesitan descargar la base de datos.

En este caso todas las bases de datos para este curso se pueden consultar en nuestro GitHub en este [enlace](https://github.com/a2Proyectos/MachineLearning_Data).

Ahora vamos a descargar la base de datos directamente desde un repositorio de GitHub en formato csv. Esto se hace de la siguiente manera.

Importamos nuestras librerías principales:

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import os

Definimos una función para extraer datos. **DOWNLOAD\_ROO**T es la base del GitHub donde vamos a estar descargando las bases de datos.

DOWNLOAD\_ROOT = "https://raw.githubusercontent.com/a2Proyectos/MachineLearning\_Data/main/"

Complementos con la dirección específica de la base de datos que queremos.

LONDON\_SALARY = "Capitulo\_2/housing\_in\_london\_yearly\_variables.csv"

LONDON\_HOUSING = "Capitulo\_2/housing\_in\_london\_monthly\_variables.csv"

LONDON\_MAP = os.path.abspath("") + "\map\London\_Borough\_Excluding\_MHW.shp"

Creamos una función que una nuestros 2 parámetros, importe los datos CSV.

def extraer\_datos(root,database):

csv\_path = root + database

return pd.read\_csv(csv\_path)

* **Un Vistazo Rápido a La Base de Datos**

Hay que combinar nuestra columna de media salarial, con nuestro dataframe salarial, por lo que usaremos el método de Pandas merge() y creamos un DataFrame con nuestra base de datos.

df1 = extraer\_datos(DOWNLOAD\_ROOT, LONDON\_HOUSING)

df2 = extraer\_datos(DOWNLOAD\_ROOT, LONDON\_SALARY)

df1 = df1.filter(items=["median\_salary","area","date"])

data = pd.merge(df2,df1)

data.head()

Table

Description automatically generated

Para obtener más información de los datos podemos utilizar este método.

data.info()

Text

Description automatically generated

Como se puede observar tenemos 5 columnas diferentes, de las cuales 4 son numéricas y 3 son de otro tipo de datos. También se puede observar que el rango de datos es de 1071, lo cual es considerado bajo para un estándar de Machine Learning, aunque es perfecto para nuestro primer proyecto.

Una forma muy común para saber qué tipo de datos contiene alguna variable de tipo objeto, es contar sus valores, ejemplo:

data["area"].value\_counts()

Texto

Descripción generada automáticamente

Aquí rápidamente se puede apreciar que en esta columna tenemos los valores asociados a los nombres de municipios en Londres.

Una forma muy utilizada para obtener información de nuestros datos numéricos es el método ***describe().*** Este método suele retornar valores con notación científica por lo que, podemos cambiar el formato con las opciones de pandas.

pd.options.display.float\_format = '{:,.2f}'.format

Esto lo que hace es pedirle a pandas que mantenga un formato para los datos de tipo float en específico. Aquí le pedimos separación por comas, y 2 decimales, es por eso por lo que escribimos el 2f.

data.describe()

Table

Description automatically generated

Este método nos regresa los cuantos objetos tiene cada columna, sin contar los objetos vacíos, a diferencia del método info (). También nos da la media, la desviación estándar, el valor mínimo y el valor máximo de cada columna. Por último, nos da los percentiles al 25%, 50% y 75%, los cuales indican que valor está por debajo de ese porcentaje de datos, por ejemplo, el 50% de las casas vendidas es inferior a 351 y el 50% de datos de la media salarial es menor a 28284.

Otra forma rápida de visualizar los datos numéricos, pero gráficamente es utilizando una librería para crear histogramas. Un histograma muestra el número de muestras (en el eje vertical) y su valor dado (eje horizontal). Se puede graficar cada columna a la vez o todas al mismo tiempo.

Ya que estamos utilizando Jupyter Notebook y vamos a graficar con la librería matplotlib, debemos tomar en cuenta que esta librería necesita producir sus graficas en un back-end especifico de nuestro dispositivo.

Es por esto que es una buena idea especificarle que use un back-end específico para nuestro Jupyter Notebook utilizando el comando ***%matplotlib inline***.

%matplotlib inline

data.hist(bins=50,figsize=(15,10))

plt.show()

Chart, bar chart

Description automatically generated

Hay algunas cosas que hay que resaltar en estos histogramas:

1. La columna de numero de crímenes tiene varios valores vacíos o null values, es por eso por lo que tiene un pequeño salto en el número 0. Hay que tener este tipo de detalles en cuenta ya que serán importantes a la hora de entrenar el modelo.
2. La columna llamada borough\_flag consiste en una columna que solo tiene valores 0 y 1, por lo que probablemente no sea muy relevante para este modelo, también es importante tomar esto en cuenta, además sus valores nos son representativos para la predicción de precios.
3. Todas estas variables tienen diferentes escalas, por lo que puede ser recomendable abordar este tema, aunque será más adelante en este capítulo.
4. Como se puede observar los histogramas tienden a cargarse a un lado del cuadro completo, esto puede hacer la tarea más difícil a ciertos modelos de ML, por lo que intentaremos transformar las variables más adelante, para lograr normalizarlas.

* **Crear un Set de Prueba**

Puede que suene un poco extraño querer poner parte de los datos fuera en esta primera instancia del proyecto, después de todo solo hemos echado un pequeño vistazo a los datos y se siente como si quisiéramos saber mucho más antes de emplear un algoritmo.

Hay que tomar en cuenta que cuando dividimos nuestros datos nos quedan 2 diferentes sets, uno de prueba y uno de entrenamiento. El set de entrenamiento es el que vamos a usar durante todo el proceso de entrenamiento de nuestro modelo y el set de prueba normalmente solo se utiliza como un calificador, para saber que tan bueno fue nuestro modelado al final de todo el proceso.

¿Por qué no se utiliza el set de pruebas para entrenar el modelo?

Pues, nuestro cerebro es un perfecto sistema de detección de patrones, por lo que es bastante propenso a cometer overfitting.

Si solo se mira al set de prueba puede que te encuentres con un patrón interesante que te lleve a seleccionar cierto modelo de ML. Cuando se estima el error de generalización usando el set de prueba, la estimación va a ser bastante optimista, pero cuando se corra el sistema completo probablemente no sea tan optimista. A esto se le llama sesgo de espionaje de datos.

Crear un set de prueba es relativamente simple, solo necesitas tomar un porcentaje del total de tus datos.

Una de las formas más sencillas y eficientes de crear un set de prueba y por lo tanto un set de entrenamiento, es usar una función incluida en la paquetería scikit-learn que se llama train\_test\_split.

Scikit-learn es una librería de Python que sirve para hacer un análisis predictivo de datos simple y eficiente, contiene herramientas para hacer tantos modelos de clasificación, regresión, clustering, algoritmos de reducción de dimensiones, preprocesamiento de datos, entre muchos otros.

La función train\_test\_split nos ayuda a separar nuestros datos utilizando un porcentaje de división deseado en un set de entrenamiento y un set de pruebas. La diferencia principal entre usar esta función y separarlos manualmente es que, la función toma los índices de nuestro DataFrame aleatoriamente por lo que al usarla tendremos partes de todo el data set original tanto en el set de prueba como en el de entrenamiento.

Imaginemos que tenemos un pastel de 5 sabores repartidos uniformemente, lo normal sería que tomáramos un poco de cada sabor aleatoriamente y no solo una parte grande de 2 sabores que estaban al inicio o al final.

Esta es la diferencia entre solo recortar un set de datos, a tomar pequeñas partes de todo el set completo. Y es a lo que nos ayuda esta función.

Importamos la función

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

Seleccionamos, por ejemplo, el 30% de los datos para el set de prueba. Random\_state es la semilla que se usa para generar números aleatorios.

set\_ent, set\_prueba = train\_test\_split(data, test\_size=0.3, random\_state=45)

Confirmamos la división

print(len(set\_ent),len(set\_prueba))



Como se puede observar efectivamente el set completo de datos se dividió en, 70% el set de entrenamiento y 30% el set de prueba, esto utilizando índices aleatorios, generados por la misma función.

Es importante considerar como se dijo antes, que esta función distribuye las divisiones tomando los índices aleatoriamente del conjunto de datos completo, esto funciona bien si tu set de datos es lo suficientemente grande, pero si no lo es, se corre el riesgo de tener un gran sesgo de datos.

Por ejemplo, supongamos que una empresa hace una encuesta a 2,000 personas.

Normalmente las empresas que hacen encuestas profesionalmente hacen que la encuesta sea comparable a todas las categorías de la población encuestada.,

Por ejemplo, en la población de la ciudad encuestada hay 60% hombres y 40% mujeres, entonces la casa encuestadora deberá de seleccionar una muestra con los mismos parámetros de hombres vs mujeres

Supongamos que este es el caso de la media salarial que tienes en tu set de datos. deberías asegurarte de que tus sets de datos para prueba y entrenamiento tienen una distribución igual de todas las categorías de salarios.

Como la media salarial es una variable numérica, primero se tiene que crear una división categórica de la media salarial, si vemos el histograma antes mostrado, se puede notar que la mayoría de los salarios van de 20000 a 40000, por lo que sería buena idea basarnos en estos valores para categorizar esta variable, esto se puede hacer utilizando el método de pandas **.cut()**.

La función **cut** se utiliza para segmentar un set de datos en pequeños contenedores, para utilizarla hay que ingresar el set de datos que se quiere segmentar, después el rango de los contenedores o bins y por ultimo las etiquetas o los nombres que se le quiere dar a cada contenedor o categoría. Ejemplo:

Para categorizar una variable con 5 categorías.

data["salary\_cat"] = pd.cut(data["median\_salary"],

bins=[0., 10000, 20000, 30000, 40000,

np.inf],

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

Después podemos hacer un histograma de las categorías.

data["salary\_cat"].hist()

Chart, histogram

Description automatically generated

En este caso en la categoría 1 tenemos todos los menores a 10000, en la categoría 2 tenemos todos los valores de 10000 a 20000 en la 3 de 20000 a 30000 (es la más grande del set de datos), en la 4 de 30000 a 40000 y en la 5 todos los valores mayores a 50000.

Si visualmente no es suficiente para saber la magnitud de las categorías siempre podemos usar este método:

data["salary\_cat"].value\_counts()

Table

Description automatically generated

Como la categoría 1 tiene 0 valores, podemos volver a categorizar en nuestras otras 4 categorías.

data = data.dropna(subset=['median\_salary'])

data = data.reset\_index()

data["salary\_cat"] = pd.cut(data["median\_salary"],

bins=[10000., 20000., 30000., 40000.,

np.inf],

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

data["salary\_cat"].hist()

Chart

Description automatically generated

Para comprobar que nuestras categorías se redefinieron hay que volver a usar el código anterior:

data["salary\_cat"].value\_counts()

Graphical user interface

Description automatically generated

Ahora que sabemos cómo categorizar nuestras variables, podemos volver a definir nuestros sets de entrenamiento y pruebas. Esto con la clase de **scikit-learn StratifiedShuffleSplit**, aquí un ejemplo:

from sklearn.model\_selection import StratifiedShuffleSplit

Generamos nuestro objeto para que lo divida en 30% y solo haga una división

split = StratifiedShuffleSplit(splits=1, test\_size=0.3, random\_state=45)

Creamos nuestras variables basándonos en nuestras categorías.

for ent\_index, prueba\_index in split.split(data,data["salary\_cat"]):

cat\_set\_ent = data.loc[ent\_index]

cat\_set\_prueba = data.loc[prueba\_index]

Comprobación en porcentajes:

cat\_set\_prueba["salary\_cat"].value\_counts() / len(cat\_set\_prueba)

A picture containing chart

Description automatically generated

Como podemos ver los porcentajes de los datos se distribuyeron de manera correcta.

Se ha dedicado algo de tiempo a la generación de el set de prueba y set de entrenamiento, y esto es con justa razón: a esta parte mucha gente la omite y en realidad es una pieza critica para un proyecto de Machine Learning.

En los siguientes capítulos será bastante útil sobre todo al ver la llamada validación cruzada.

Hora de pasar a la exploración de datos.

* **Descubrir Y Visualizar Los Datos Para Tener Una Mejor Perspectiva**

Hasta ahora solo hemos echado un vistazo rápido a los datos para si obtener una idea de que tipo de información estamos manipulando. Ahora la tarea es ir más allá de ese vistazo rápido.

Primero, vamos a dejar el set de prueba a un lado y vamos a enfocarnos en el set de entrenamiento (Es el que contiene el 70% de nuestros datos). Si este es demasiado grande, podrías tomar un set de exploración, para hacer manipulaciones fáciles y rápidas.

Para no trabajar directamente en nuestro set de entrenamiento y mantenerlo sin cambios haremos un data frame. Esto lo hacemos con la función [copy](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.copy.html), la cual nos proporciona un DataFrame exactamente igual al que seleccionemos:

df = cat\_set\_ent.copy()

df.head()



* **Visualizar Datos Geográficamente**

Cuando se trabaja con datos enfocados a una zona geográfica en específico, es una buena idea visualizar los datos con un mapa de la zona.

Para hacer esto se tiene que contar con las coordenadas de cada región o descargar alguna paquetería que ya las incluya, y de esta forma nuestro algoritmo va a ser capaz de graficarlas.

Para esto vamos a necesitar, en conjunto con nuestro set de datos LONDON\_MAP, una nueva librería que se llama geopandas, la cual es una extensión de la librería pandas, para trabajar con datos geoespaciales.

Geopandas construye dataframes con columnas geométricas para hacer operaciones de espacio, en vez de trabajar con texto, booleanos, enteros, entre otros como lo hace pandas. Geopandas trabaja con puntos, polígonos, rombos y muchos más, así nos permite graficar figuras como mapas cuando no se tienen las coordenadas exactas, se puede encontrar más información en este [link](https://geopandas.org/getting_started/introduction.html).

Para instalarlo:

1. Buscar anaconda prompt en tu dispositivo.

Graphical user interface, text, application

Description automatically generated

1. Ejecutar el comando **conda install geopandas**

Text

Description automatically generated

Una vez tenemos instalada nuestra paquetería podemos importar nuestra base de datos con las coordenadas de nuestro mapa:

import geopandas as gpd

londres\_map = gpd.read\_file(LONDON\_MAP)

londres\_map.head()

Graphical user interface

Description automatically generated with medium confidence

En la columna geometry podemos observar cómo es que se ven los datos espaciales. En este caso vamos a graficar los municipios con polígonos y sus coordenadas.

londres\_map.plot()

Chart, map

Description automatically generated

Así es como se ve Londres y la división de sus municipios

Este es el poder de las librerías geoespaciales, en internet se pueden encontrar miles de ejemplos de figuras con las que se puede interactuar en Python, ya que lo único que se tiene que hacer es incluir estos datos al data set con el que estamos trabajando.

Vamos a probar estos métodos, combinando el mapa con nuestros datos sobre el precio medio de las casas en Londres y las casas que se han vendido en Londres.

Antes de unir nuestro data set de entrenamiento y nuestro data set del mapa, tenemos que cambiar los nombres de las columnas de este último, ya que la función que vamos a utilizar para hacerlo va a buscar que el nombre de la columna para identificar los patrones de unión.

En este caso las columnas que queremos que identifique son las de name y gss\_code, que en el data set de entrenamiento están como área y code.

Para esto utilizaremos la función de pandas [rename](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.rename.html), la cual nos permite renombrar algún valor en el DataFrame. Aunque es importante tomar en cuenta que la sustitución tiene que estar introducida en forma de diccionario.

Nos aseguramos de que los nombres estén en minúsculas:

londres\_map.columns = londres\_map.columns.str.lower()

londres\_map = londres\_map.rename({'name': 'area', 'gss\_code': 'code'}, axis=1)

londres\_map["area"] = londres\_map["area"].str.lower()

Renombramos:

También hay que seleccionar las columnas que necesitamos para unir a nuestro data set, ya que no todas son importantes.

Esto lo haremos con la función de pandas [filter](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.filter.html).

La función **filter** nos permite seleccionar columnas o índices específicos en un DataFrame, ya sea por el nombre o por su número de índice. Importante hay que mencionar que los datos tienen que ser introducidos en forma de lista.

En el caso de nuestro set de datos para el mapa nos vamos a quedar con las columnas área, code, hectares y geometry.

londres\_map = londres\_map.filter(items=["area","code","hectares","geometry"])

londres\_map.head()

Table

Description automatically generated

Ahora tenemos que hacer lo mismo con nuestro set de entrenamiento, ya que solo graficaremos la parte de la media del costo de casas y la venta de casas totales en cada municipio.

Para graficar esto solo necesitamos un valor por municipio, aunque nosotros tenemos un valor por cada año, hay que decidir como lo vamos a representar en uno solo.

Lo mejor será agarrar la media anual con la que ya contamos, promediarla, y obtener un número final que no esté dividido por años.

Aquí nos preguntaremos:

**¿Cómo hacer este tipo de operaciones agrupándolas por municipio?**

La respuesta es muy sencilla, recordemos que pandas tiene una función llamada [groupby](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.groupby.html).

La función **groupby** es una combinación que divide los datos y te permite aplicar operaciones sobre estas divisiones.

Por ejemplo, vamos a utilizar esta función para crear un nuevo DataFrame donde vamos a separar los datos por la columna área (municipios), y a la columna de municipios le vamos a agregar la media del precio de las casas y la suma de las casas vendidas.

df\_m = df.groupby('area').agg({'average\_price': ['mean'], 'houses\_sold': 'sum'})

Le asignamos nombre a las columnas del nuevo dataframe y nos aseguramos de que el índice tenga una distribución normal, ya que hay veces que se queda con los índices de el dataframe base.

df\_m.columns = ['average\_price', 'houses\_sold']

df\_m.reset\_index(inplace = True)

df\_m.head()

Table

Description automatically generated

Ha llegado el momento de unir nuestros 2 set de datos, y para esto utilizaremos una nueva función que se llama **merge**.

La función **merge** une 2 dataframes con un parámetro en común, en este caso nuestro parámetro en común es el área.

Lo que va a suceder es que basándose en el área esta función agregara las columnas en el índice del municipio que tenían en su respectivo set de datos, por ejemplo:

londres\_map = pd.merge(londres\_map,df\_m,on="area")

londres\_map.head()

Text

Description automatically generated

Como podemos observar obtuvimos justo el resultado que necesitábamos, por lo que es hora de graficar.

Podemos practicar con algunos atributos de la función plot de matplotlib, por ejemplo:

* Column: para especificar la columna de donde queremos tomar los valores.
* Cmap: para indicar la escala de colores que queremos utilizar al rellenar los municipios.
* Edgecolor: para especificar el color de las líneas divisorias.
* Legend: para agregar una barra que mida los datos.

El parámetro legend lo podemos personalizar, por ejemplo:

* Label: para darle titulo
* Orientation: para decirle en que zona de la gráfica queremos la barra.

Y por último para ponerle un título a nuestra grafica utilizamos set\_title.

plt = londres\_map.plot(column = 'average\_price', cmap = 'Reds', edgecolor = 'maroon',

legend = True, legend\_kwds = {'label': 'Precio', 'orientation' : 'horizontal'})

plt.set\_title('Media de los precios en las casas')

plt.axis(‘off’)

Diagram

Description automatically generated

Y ahora repetimos lo mismo, pero para el número de casas vendidas.

plt = londres\_map.plot(column = 'houses\_sold', cmap = 'Blues', edgecolor = 'maroon',

legend = True, legend\_kwds = {'label': 'Precio', 'orientation' : 'horizontal'})

plt.set\_title('Total de casas vendidas')

plt.axis('off')

Chart

Description automatically generated

Como podemos observar, los precios de las casas son bastante más altos en los municipios del centro, que si saben algo de la geografía de Londres es donde está la City of London.

Es el distrito financiero de Londres y ahí están todos los bancos multibillonarios, así que claro, estará carísima esa zona.

Mientras que en el número de casas vendidas está un poco más distribuido por toda la ciudad, por lo que no podríamos asumir que hay un sector exacto donde se concentren más ventas, como con el rango de precios.

* **Buscar Correlaciones**

Como nuestro data set no es muy grande, no es necesario hacer modificaciones antes de buscar el coeficiente de relación estándar, también conocido como la r de Pearson.

El coeficiente de correlación es una medida que se enfoca en cuantificar la relación lineal entre una variable y otra. la fórmula compara la distancia de cada dato puntual respecto a la media de la variable y utiliza esta comparación para decirnos hasta qué punto la relación entre las variables se ajusta a una línea imaginaria trazada entre los datos.

Por ejemplo, supongamos que tenemos 2 variables, la inflación y el precio de la comida. El precio de la comida normalmente tiende a aumentar, si la inflación aumenta por lo que estos tienen una correlación positiva.

Si una variable se disminuye mientras la otra aumenta, significa que tienen una correlación negativa.

Para crear una matriz de correlación de las variables de nuestro set de entrenamiento utilizaremos la función **corr**.

Esta función crea una matriz de correlación con las variables de nuestro set de datos, hay varias formas de medir la correlación, pero en este caso utilizaremos la antes vista, la de Pearson.

matriz = df.corr(method=’pearson’)

Ahora como ejemplo vamos a medir la correlación de la media de precios en las casas, con todas las otras variables.

matriz["average\_price"].sort\_values(ascending=False)

Al método **sort\_values** lo usamos para que nos ordene los valores, y le señalamos que los ponga de manera descendente, ya que la forma predeterminada es ascendente.

Text

Description automatically generated

Este tipo de correlación se mide de -1 a 1, donde -1 es una correlación 100% negativa y 1 es una correlación 100% positiva.

En la tabla de correlaciones podemos ver que obviamente al medir la media de precios contra su misma variable la correlación es de 1.

Lo que es bastante interesante es que la media salarial tiene una correlación positiva algo alta, por lo que se puede asumir que es la variable con el comportamiento más parecido a la de la media de precios.

Otra forma bastante interesante de visualizar las correlaciones es con la paquetería seaborn.

[Seaborn](https://seaborn.pydata.org/) es una librería especializada en visualizaciones la cual tiene base en matplotlib, y nos provee una red de gráficos más avanzados para medidas estadísticas.

Para importarla:

import seaborn as sns

En este caso utilizaremos su función heatmap, la cual grafica los datos en un rectángulo de colores, diferenciados por su valor.

Solo queremos graficar la mitad del rectángulo, entonces crearemos una matriz de booleanos, del tamaño de la mitad de nuestros datos.

Para utilizar esta función requerimos de un vector que se conoce como el vector del triángulo superior ya que los valores por debajo de la diagonal central los vuelve ceros o booleanos.

Para esto utilizaremos la función de numpy np.triu:

Esta función [triu](https://numpy.org/doc/stable/reference/generated/numpy.triu.html) toma nuestros datos y crea una matriz, como si fuera la matriz de correlación, y separa los valores mediante la diagonal. Esta función en específico trabaja con los valores por debajo de la diagonal, aunque numpy tiene otra función que se llama **tril** la cual trabaja con los valores por encima de la diagonal.

mask = np.triu(np.ones\_like(matriz, dtype = bool))

Text

Description automatically generated

Ahora ya Podemos graficarlo.

plt = sns.heatmap(matriz, mask = mask, annot = True, cmap = 'YlGnBu\_r')

plt



De esta manera es más sencillo analizar cada correlación en comparación con cada variable.

Por último, una manera que también se utiliza mucho en cuanto a la correlación, es graficarla con la función de pandas **scatter\_matrix**.

Esta función grafica nuestra matriz de correlación elemento por elemento, mientras que en la diagonal de la matriz crea un histograma por cada variable numérica de nuestro set de datos.

Importamos la función

from pandas.plotting import matrix\_scatter

Seleccionamos los atributos que queremos analizar:

columns = ['average\_price', 'median\_salary', 'mean\_salary', 'number\_of\_jobs']

scatter\_matrix(df[columns], figsize = (12, 12), color = '#D52B06', alpha = 0.3,

hist\_kwds = {'color':['bisque'], 'edgecolor': 'firebrick'});

Donde el parámetro *alpha*, es que tan transparente queremos que sean los datos en la gráfica, y los otros son argumentos de forma y color como los vistos en las gráficas anteriores.

<https://pandas.pydata.org/docs/reference/api/pandas.plotting.scatter_matrix.html>

Chart

Description automatically generated

Como habíamos observado antes, la media del precio de las casas y la media salarial tienen una correlación positiva algo alta. Es por eso que en su grafica pareciera que intentan formar una diagonal.

Vamos a analizarlo más a fondo con la función **plot** de matplotlib, la cual ya hemos usado antes.

Utilizamos el tipo scatter, el cual nos grafica cada punto de información en la gráfica y un *alpha* de 0.1.

df.plot(kind="scatter",y="median\_salary",x="average\_price",alpha=0.1)

Chart, scatter chart

Description automatically generated

Ahora Podemos confirmar que tienen una correlación positiva ya que tienen una tendencia claramente creciente, y parecen tener una correlación más fuerte sobre todo en los valores más bajos.

* **Experimentos Con Combinaciones De Atributos**

La sección pasada debió haberte dado una idea de cómo puedes visualizar datos para generar indicios e identificar algunas cosas útiles.

Por ejemplo, descubrimos que la variable de media de precio por casa tiende a cambiar por localización y que esta misma está bastante correlacionada con la variable de media salarial

Una forma de darle más importancia a las otras variables es combinando algunas de estas.

Por ejemplo, podemos combinar la variable de casas vendidas con la de tamaño de población para ver si así podemos aumentar su correlación.

Para ello ejecutaremos el siguiente algoritmo:

df["vendidas\_poblacion"] = df["population\_size"] / df["houses\_sold"]

Y ahora ejecutamos la matriz de correlación una vez más.

matriz = df.corr(method='pearson')

matriz["average\_price"].sort\_values(ascending=False)

Text

Description automatically generated

Como Podemos observar, nuestras 2 variables con la menor correlación positiva pasaron a tener una correlación de 0.28 estando combinadas. Lo que podría volverlas más significativas para nuestro modelo.

Siempre podremos usar este método y repetirlo hasta encontrar una correlación que se acerque a lo deseado.

* **Preparar Los Datos Para Un Algoritmo De Machine Learning (Transformación de datos)**

Llego la hora de preparar nuestros datos para nuestro algoritmo de Machine Learning, en vez de hacer esto manualmente, es una buena idea crear un grupo de funciones para ello. Estas son las razones:

* Esto te permitirá hacer las transformaciones fácilmente con cualquier data set, no solo los de este proyecto.
* Eventualmente tendrás tu librería de funciones de transformación, la cual se puede reutilizar en cualquier proyecto.
* Esto facilitara la tarea de probar diferentes tipos de transformación para decidir cuál es la que más te conviene.

En este caso vamos a crear 2 nuevos dataframes, uno con la variable a predecir (media de precio en las casas) y otro con sus predictores (el resto de las variables).

El primero lo vamos a crear utilizando nuestro set de datos original y copiando solo esa variable.

df\_label = cat\_set\_ent['average\_price']

Para obtener solo los predictores vamos a eliminar la variable a predecir con la función [drop](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.drop.html).

La función de pandas **drop** remueve columnas o filas, mientras se especifique su nombre o su índice. Esta función tiene un parámetro que se llama index, donde 0 son filas y 1 son columnas.

df = cat\_set\_ent.drop('average\_price', axis=1)

### **Limpieza de Datos**

La mayoría de los algoritmos de Machine Learning no pueden trabajar óptimamente con valores faltantes o vacíos, para esto vamos a crear un grupo de funciones.

Como pudimos observar en nuestros histogramas anteriores, una variable con bastantes valores vacíos es la de casas vendidas.

Table

Description automatically generated

Es por eso por lo que tiene tantos ceros. Tenemos 3 opciones para arreglar este problema:

1. Eliminar cada línea que tenga un valor vacío, aunque esto implique eliminar todas las variables de esa línea, incluyendo los municipios.
2. Eliminar la variable de casas vendidas por completo.
3. Llenar esos valores con algún valor, ejemplo, la media, ceros, la mediana, etcétera.

Nuestra librería de pandas tiene una función para cada una de estas opciones:

1. [Dropna](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.dropna.html): esta función remueve la fila de cada valor vacío en la posición en el que se lo indiquemos ya sea columna, fila o el set de datos completo.
2. **Drop**: como lo mencionamos antes, remueve columnas o filas, mientras se especifique su nombre o su índice. Esta función tiene un parámetro que se llama index, donde 0 son filas y 1 son columnas.
3. [Fillna](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.fillna.html): esta función lo que hace es rellenar los valores vacíos con un valor que le especifiquemos, se puede seleccionar el tipo de valor, el método, en que columna o en que fila rellenar.

Un ejemplo para cada opción:

df.dropna(subset=["houses\_sold"]) #opción 1

df.drop("houses\_sold",axis=1) # opción 2

Tomamos la media para opción 3

median = df["houses\_sold"].median()

Llenamos los valores con la media

df['houses\_sold'].fillna(median,inplace=True) #opción 3

la función **fillna**, no calcula la media automáticamente, es por eso por lo que la calculamos primero con el método median, y después se la introdujimos a la función para que la usara de valor de relleno.

Otra forma de rellenar datos es usando la paquetería de scikit-learn.

Con la clase SimpleImputer primero creas el objeto, especificando con que valor quieres llenar los valores.

from sklearn.impute import SimpleImputer

imputer = SimpleImputer(strategy="median")

Después usamos la función **fit** con nuestras variables numéricas.

Recordemos que con la función **info** podemos ver qué tipo de datos tienen nuestras variables.

Text

Description automatically generated

Ahí podemos ver que tenemos un dataframe con 3 variable tipo objeto, por lo que procederemos a crear un data set sin estas.

df\_num = df.drop(["area","date","code"],axis=1)

Ahora utilizaremos una función de este método la cual calcula la media de todas nuestras variables. Esta función es la función **fit**.

imputer.fit(df\_num)

Una vez que la función **fit** calculó todas nuestras medias, utilizamos la función **transform**, que transforma los valores vacíos en valores con la media respectiva.

X = imputer.transform(df\_num)

De esta manera tenemos ya nuestros valores vacíos transformados en su media, el único inconveniente es que esta función nos regresa nuestros datos en forma de lista.

Text

Description automatically generated

Para transformar estos datos de vuelta a un dataframe, solo tenemos que crear un pandas dataframe, al que le vamos a ingresar estos datos y le vamos a asignar el nombre de las columnas y los índices del dataframe original.

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

De esta manera tenemos nuestro dataframe completo y sin datos vacíos.

Table

Description automatically generated

Como resumen, lo que hicimos fue:

1. Importar la clase de scikit-learn [**SimpleImputer**](https://scikit-learn.org/stable/modules/generated/sklearn.impute.SimpleImputer.html), la cual sirve específicamente para tratar con valores vacíos.
2. Creamos nuestro objeto con el parámetro de estrategia **median**, aunque este nos ofrece más opciones como **mean**, **most\_frequent** y **constant**.
3. Utilizamos su función **fit**, para calcular las medias.
4. Transformamos los valores vacíos a los valores calculados con la media, gracias a la funcion **transform**.
5. Creamos un dataframe insertando estos valores, y reasignando los nombres de columnas e índices del data set original.

* **Manejo de texto y valores categóricos**

En este caso en nuestro set de datos de entrenamiento, solo tenemos una variable de texto, la cual es la de área, y nos indica en que municipio estamos trabajando.

Vamos a crear un dataframe que contenga solo esta variable.

df\_cat = df[["area"]]

df\_cat.head()

como la mayoría de los algoritmos de Machine Learning prefiere trabajar con números que, con palabras, podemos convertir cada uno de los municipios en números.

Para esto utilizaremos la clase de scikit-learn que se llama **OrdinalEncoder**, este es un transformador, el cual recibe valores categóricos, normalmente de texto y los convierte en valores numéricos que denotan categorías.

Primero importamos la clase y creamos el objeto:

from sklearn.preprocessing import OrdinalEncoder

ordinal\_encoder = OrdinaEncoder()

Una vez creamos el objeto, utilizamos la función **fit\_transform**, la cual toma nuestros datos y los transforma en numéricos.

df\_oe = ordinal\_encoder.fit\_transform(df\_cat)

Ahora tenemos una lista con nuestras variables de forma numérica.

Table

Description automatically generated with medium confidence

Para revisar cuales son las categorías que se transformaron solo utilizamos el método **categories\_**

ordinal\_encoder.categories\_

Text, letter

Description automatically generated

Un problema con este tipo de transformación es que los algoritmos de ML pueden entender que 2 valores que están uno al otro son similares, lo cual podría ser bueno se fueran por ejemplo calificaciones, donde 10 y 8 son valores altos.

En nuestro caso, no nos ayuda porque solo son nombres de municipios. Por lo que una forma para resolver esto es crear los llamados atributos binarios.

Un ejemplo de atributos binarios, si tenemos por ejemplo a la ciudad de Londres, entonces cierto atributo va a ser igual a 1, y todos los demás 0. De esta forma cada municipio tendrá su atributo en un vector, y cuando sea llamado ese atributo será igual a 1.

Por ejemplo, si estamos hablando del vecindario de Merton, entonces todos los demás vecindarios tendrán 0 de valor, y Merton tendrá 1. Nosotros como humanos entendemos Merton, la computadora entiende un 1 en el lugar donde iría Merton.

A este método se le llama **encriptación caliente**, o **One-Hot Encoding** en inglés ya que cuando se encuentra el atributo 1 es caliente y cuando es 0 es que esta frio, como en el juego tradicional.

Scikit-learn tiene una clase específica para encriptamiento caliente que se llama [**OneHotEncoder**](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html)**.** Esta clase sirve para transformar nuestras variables categóricas en vectores de unos y ceros.

Primero importamos la clase y creamos el objeto.

from sklearn.preprocessing import OneHotEncoder

encoder = OneHotEncoder()

Después de crear el objeto utilizamos su variable de transformación, como lo hemos estado haciendo con las otras clases de scikit-learn.

La función de transformación para esta clase es **fit\_transform**.

df\_1hot = encoder.fit\_transform(df\_cat)

df\_1hot

De esta forma ya tenemos una matriz con nuestras variables categóricas en forma binaria.

Al llamar esta matriz nos damos cuenta de que es un tipo de matriz de scipy, el cual no podemos ver, esto se hace porque normalmente son matrices tan grandes que sería mucha memoria consumida para almacenar un gran número de ceros.

Por lo que la función la regresa con este tipo de encriptación, que consume menos memoria a nuestro dispositivo.



Para ver la matriz únicamente tenemos que utilizar la función toarray(), la cual nos generara un array con la matriz.

df\_1hot.toarray()

A picture containing text

Description automatically generated

* **Transformadores personalizados**

Aunque la librería scikit-learn ya nos da una serie de elementos que son bastante útiles. Hay ocasiones en las que tendrás que crear tus propias clases y funciones para facilitarte el trabajo.

Para hacer esto vamos a tener que recurrir a la programación orientada a objetos. Hay que repasar temas como las clases, los atributos, los métodos, el constructor, la herencia y el duck typing.

Al crear nuestras propias clases, la principal prioridad es que funcionen con las funcionalidades de Scikit-learn. Ahora, recordemos que Scikit learn **NO FUNCIONA** con herencia. En vez de eso utiliza una técnica que se llama duck typing.

Gracias al duck typing solo necesitaremos implementar tres métodos, **fit()**, **transform()** y **fit\_transform()**, justo cómo funcionan las de scikit-learn.

El método de **fit-transform()** se puede importar “de a gratis” simplemente agregando **TransformerMixin** como una clase base.

Como ejemplo vamos a crear una clase que nos ayude a reemplazar los valores anormales o outliers de un set de datos por un valor vacío o NaN. Esto puede ser bastante útil, para disminuir el ruido en los datos y obtener predicciones as acertadas.

Recordemos que **fit\_transform()**, prepara y transforma nuestros datos, mientras que **fit()** solo prepara y **transform()** solo transforma.

Otra clase base que podemos importar es la de **BaseEstimator**, esta nos ayudará a no tener que proveer de argumentos nuestro constructor y además nos dará 2 métodos predefinidos, **get\_params** y **set\_params**, los cuales nos permiten obtener e introducir parámetros y a perfeccionar nuestra afinación de hiperparámetros.

Primero importamos nuestras clases base:

from sklearn.base import BaseEstimator, TransformerMixin

Creamos nuestra clase:

class RemoverOutliers(BaseEstimator,TransformerMixin):

def \_\_init\_\_(self,factor=1.5):

self.factor = factor

def outlier\_detector(self,X,y=None):

X = pd.Series(X).copy()

q1 = X.quantile(0.25)

q3 = X.quantile(0.75)

iqr = q3 - q1

self.lower\_bound.append(q1 - (self.factor \* iqr))

self.upper\_bound.append(q3 + (self.factor \* iqr))

def fit(self,X,y=None):

self.lower\_bound = []

self.upper\_bound = []

X.apply(self.outlier\_detector)

return self

def transform(self,X,y=None):

X = pd.DataFrame(X).copy()

for i in range(X.shape[1]):

x = X.iloc[:, i].copy()

x[(x < self.lower\_bound[i]) | (x > self.upper\_bound[i])] = np.nan

X.iloc[:, i] = x

return X

Este es un ejemplo de cómo se vería una clase, vamos a repasarla un poco parte por parte:

* El nombre de la clase es remover outliers y está recibiendo las 2 clases base antes explicadas de la paquetería scikit-learn.
* Empezamos con el constructor, este inicializa los atributos de la clase y tiene un atributo llamado factor.
* Factor es la medida que utilizaremos como hiperparámetro para controlar el proceso de eliminación de outliers, mientras más grande sea el factor, más extremo será el proceso de eliminación.
* Outlier\_detector es una función que utiliza los cuantiles para separar los datos de los extremos, en este caso usaremos los cuantiles del 25% y del 75%, ya que tomaran los datos de los extremos y nos quedaremos con los datos más comunes que están en la media. Esta medida de cuantil se verá afectada por nuestro hiperparámetro factor, ya que se va a multiplicar por este.
* La función **fit** va a heredar las características de la función outlier\_detector y va a almacenar los datos cambiados en nuestro objeto. Es importante recalcar que la función fit no va a aplicar los cambios, solo los va a almacenar.
* Después tenemos la función transform, la cual va a revisar todo nuestro set de datos 1 por 1 y si es outlier lo va a cambiar por un valor NaN, como su nombre lo dice va a transformar los datos y nos va a regresar un dataframe con los datos transformados.

Vamos a utilizar nuestra nueva clase con un ejemplo muy sencillo:

Creamos un dataframe de prueba:

prueba = pd.DataFrame({'col1':[100,200,300,999],'col2':[0,0,1,2],'col3':[-10,0,1,2]})

prueba

Table

Description automatically generated

Este es un dataframe muy sencillo donde a simple vista vemos que el valor 999, está muy por fuera de la media de valores de la columna 1, y el valor -10 está fuera de la media en la columna 3.

Para utilizar nuestra clase primero creamos nuestro objeto.

remover = RemoverOutliers()

Después utilizamos la función **fit**, para que nuestro objeto localice todos los outliers en nuestro dataframe.

remover.fit(prueba)

Recordemos que esta función no regresa ningún valor, simplemente se adapta a nuestros datos.

Por último, utilizamos la función **transform** para transformar los datos.

remover.transform(prueba)

Table

Description automatically generated

Como podemos observar nuestros outliers ahora están en formato NaN, por lo que ahora sería útil utilizar alguna función de las que analizamos en la sección anterior para la limpieza de datos, como sustituirlos por la media.

Otra observación importante es que al haber utilizado la clase base **TransformerMixin**, también podemos utilizar la función **fit\_transform**, la cual hace la tarea de las 2 funciones anteriores.

remover.fit\_transform(prueba)

Table

Description automatically generated

* **Escalado de Variables**

Una de las transformaciones más importantes para el procesamiento de datos en ML es la *escalación de variables*. Con muy pocas excepciones los algoritmos de ML no trabajan bien cuando se les ingresan datos con diferentes escalas.

Un ejemplo de diferentes escalas en nuestro set de datos es nuestra variable de casas vendidas con un rango numérico con media de 3800 y nuestra variable de tamaño de población con una media en sus valores de más de 2 millones. Claramente estas 2 variables tienen un rango de valores numéricos bastantes diferentes.

Hay dos maneras bastante comunes de lidiar con el escalado, una es el escalado min-max, también conocido como normalización y otra es la estandarización.

La normalización es la más simple, los valores son cambiados y rescaldados para que todos entren en un rango de 0 a 1. Esto se hace obteniendo el valor mínimo y el valor máximo de los datos y se efectúa la siguiente operación.

Text

Description automatically generated with medium confidence

La paquetería scikit-learn tiene una clase para implementar este método de escalado. La clase se llama **MinMaxScaler** y funciona como las que ya hemos visto anteriormente con las funciones **fit** y **transform**.

Para esto vamos a utilizar el dataframe de prueba que utilizamos para remover los outliers.

A screenshot of a phone

Description automatically generated with medium confidence

Importamos la clase y transformamos el dataframe:

from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()

pd.DataFrame(scaler.fit\_transform(prueba),columns=prueba.columns)

Table

Description automatically generated with medium confidence

Como resultado tenemos el mismo dataframe, pero con la misma escala en todos sus valores.

La estandarización es diferente, primero se le resta a cada valor la media de los datos y luego se divide entre la desviación estándar para que la distribución de los datos tenga una varianza unitaria.

A picture containing text

Description automatically generated

A diferencia de la normalización este método no le asigna un rango numérico especifico a los datos, lo que puede convertirse en un problema para algunas redes neuronales que necesitan un rango de 0 a 1.

La ventaja de la estandarización es que tiene menos sensibilidad a los outliers.

Un ejemplo de estandarización se puede lograr con la clase de scikit-learn **StandardScaler**.

Si tenemos este dataframe:

A screenshot of a phone

Description automatically generated with medium confidence

Con esta clase quedaría de la siguiente forma:

Table

Description automatically generated

Como conclusión la normalización útil en muchos algoritmos de ML, sobre todo redes neuronales que requieren de un rango numérico. Aunque si se tiene un set de datos con muchos outliers o mucho ruido la estandarización puede ayudar a reducirlo.

* **Pipelines De Transformación**

Un pipeline de transformación o tubería en español nos permite juntar varios métodos de transformación y ejecutarlos en una sola función.

Como hemos visto hay varios tipos de transformación y de limpieza de datos lo que puede volverse un poco tedioso de implementar si se trabaja con varios data sets al mismo tiempo.

Para este ejemplo vamos a usar la clase de scikit-learn Pipeline, y vamos a crear una función que nos encuentre los outliers, rellene esos outliers con la media y use el método de estandarización para escalar los datos:

Para esto solo necesitamos asignarle a la clase Pipeline los parámetros de el nombre que le queremos asignar a cada transformación y su respectiva clase:

pipeline = Pipeline([("remover",RemoverOutliers()),

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

("escalar",StandardScaler())])

Una vez que tenemos nuestra función pipeline lista, vamos a utilizarla con nuestro set de prueba.

Table

Description automatically generated

pipeline.fit\_transform(prueba)

Table

Description automatically generated

Aquí tenemos el resultado de las 3 operaciones ejecutado con una sola función, es por eso por lo que la creación de pipelines de transformación puede ser muy útil y puede ahorrar bastante tiempo.

Hasta ahora hemos aprendido a manejar, tanto variables categóricas como variables numéricas por separado, pero que pasa si quereos manejar estos 2 tipos de variables al mismo tiempo.

Para solucionar esto scikit-learn creo la clase llamada **ColumnTransformer**.

Esta clase funciona de la siguiente manera:

1. Se crea una variable con las columnas numéricas y se crea una variable con las columnas de categoría.
2. Al objeto se le introducen como parámetros 2 tuplas en forma de lista.
3. Las tuplas deben contener 3 cosas, el nombre que le asignemos, el método de transformación que queramos y la variable que creamos en el paso 1.
4. Se aplica la función al data set.

Vamos a hacer un ejemplo:

from sklearn.compose import ColumnTransformer

num = list(df\_num)

cat = ["area"]

pipline\_completo = ColumnTransformer([

("num", pipeline, num),

("cat", OneHotEncoder(), cat)

])

Para este ejemplo como podemos ver, tomamos los valores numéricos de nuestro data set de entrenamiento y la categoría de área (municipios).

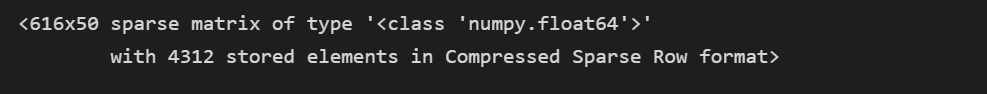
Utilizamos nuestro pipeline creado anteriormente para los valores numéricos y la clase one hot encoder para la variable de clase.

Ahora solo tenemos que ejecutar la función.

df\_preparado = pipeline\_completo.fit\_transform(df)

Como vimos anteriormente la clase OneHotEncoder regresa una matriz encriptada, la clase ColumnTransformer detecta esto y dependiendo el tamaño de la matriz decide si la deja encriptada o la ejecuta de la forma regular.

En nuestro caso la variable área tiene bastantes tipos de municipios por lo que la matriz es grande y la mantendrá encriptada, no hay que preocuparse por esto ya que podemos seguir trabajando con nuestro set de datos sin problemas.



* **Seleccionar y Entrenar un Modelo**

Ahora que hemos explorado los datos, hemos creado nuestro set de entrenamiento y de pruebas, hemos creado nuestros pipelines de transformación para preparar nuestros datos. estamos listos para entrenarlos en un modelo de ML.

### **Entrenamiento y Evaluación del Set de Entrenamiento**

La buena noticia es que, gracias a todos los pasos anteriores, el proceso empieza a ser más simple a partir de aquí.

El primer modelo que vamos a probar es el que utilizamos en el capítulo pasado, la regresión lineal.

Para esto vamos a importar la clase de LinearRegression de Scikit-learn y vamos a entrenar el modelo.

Primero tenemos que crear el objeto y luego tenemos que adaptarlo con nuestro dataframe preparado en la sección anterior y nuestra variable a predecir que en este caso va a ser la de precio medio de las viviendas.

from sklearn.linear\_model import LinearRegression

reg\_lin = LinearRegression()

reg\_lin.fit(df\_preparado, df\_label)

Ahora que tenemos el modelo preparado:

1. Primero tomamos algunos datos de nuestro set de entrenamiento.
2. Tomamos también algunos datos de nuestro dataframe con la variable a predecir.
3. Transformamos con nuestro pipeline los datos que tomamos en el paso 1.
4. Utilizamos la función predict, que intenta predecir la variable seleccionada con los datos transformados.
5. Lo comparamos con los datos de la variable a predecir, en este caso la de media de precio en viviendas.

algunos\_datos = df.iloc[:5]

datos\_predecir = df\_label.iloc[:5]

datos\_transformados = pipeline\_completo.transform(algunos\_datos)

print("Predicción:",reg\_lin.predict(datos\_transformados))

print("Originales:",list(datos\_predecir))

Text

Description automatically generated with low confidence

El modelo funciona, aunque las predicciones no son muy acertadas, la primera predicción tiene 40,000 de diferencia.

Siempre es una buena idea utilizar una medida de error, por lo que podemos empezar con el RMSE como lo vimos antes en este capítulo.

Para esto vamos a utilizar la función de scikit-learn que se llama mean\_squared\_error, tomando en cuenta que nosotros queremos la raíz del error medio cuadrado y esta función regresa solo el error medio cuadrado, tendremos que aplicar también la raíz cuadrada con la función de numpy sqrt.

from sklearn.metrics import mean\_squared\_error

prediccion = reg\_lin.predict(df\_preparado)

error = mean\_squared\_error(df\_label,prediccion)

error = np.sqrt(error)

error



Como podemos ver este modelo tiene un error de costo de más de 69,000 que considerando que son casas en un rango de precio de 50,000 a 1 millón, es bastante alto.

Este es un claro caso de underfitting lo que significa que, las variables no proveen suficiente información sobre la variable o, el modelo no es lo suficientemente poderoso.

Para resolver esto podríamos agregar más variables, aunque primero intentaremos con un nuevo modelo.

En este caso vamos a intentar usar un modelo de regresión con árboles de decisión, este es un modelo muy complejo que es capaz de encontrar relaciones no lineales en los datos, a diferencia de la regresión lineal que intenta encontrar relaciones lineales.

Sobre este modelo profundizaremos en detalle más adelante en el capítulo 6.

Por ahora vamos a usarlo con la clase de scikit-learn **DecisionTreeRegressor**.

El procedimiento va a ser el mismo que con la regresión lineal:

* Creamos el objeto.
* Lo adaptamos con la función fit.
* Creamos las predicciones con la funcion predict.
* Calculamos el RSME

from sklearn.tree import DecisionTreeRegressor

reg\_arbol = DecisionTreeRegressor()

reg\_arbol.fit(df\_preparado,df\_label)

prediccion = reg\_arbol.predict(df\_preparado)

error = mean\_squared\_error(df\_label,prediccion)

error = np.sqrt(error)

error



Como podemos ver el error con este modelo es de 0, pero ¿esto significa que el modelo es perfecto?, ¿será producto de un overfitting del modelo?

Como vimos antes no podemos tocar el set de pruebas hasta que el de entrenamiento este completamente listo, por lo que tendremos que usar parte del set de entrenamiento para la validación del modelo.

### **Mejorar La Evolución Usando Cross-Validation**

Una forma rápida y efectiva de validar el set de entrenamiento es con el método **K-fold** **cross-validation**, el cual consiste en crear diferentes sub-sets llamados folds en inglés, y evaluar cada uno por separado para después comparar los resultados de la evaluación.

Para este ejemplo vamos a usar la función de scikit-learn que se llama **cross\_val\_score**. La cual va a dividir nuestro set de entrenamiento en 10 folds. Después va a entrenar y evaluar el modelo 10 veces, escogiendo un fold diferente por evaluación cada vez y entrenándolo sobre los otros 9 folds.

Como resultado, nos regresará un arreglo con las 10 calificaciones de evaluación.

Los parámetros que tenemos que ingresar a la función son:

* El modelo que usaremos, ya adaptado a nuestros datos.
* Nuestro set de entrenamiento.
* Nuestra variable por predecir.
* La unción de utilidad
* El número de sub-sets que queremos evaluar

Esta función al ser de utilidad y no de error, nos dice que entre más alto sea el valor es mejor, por lo que al sacar la raíz cuadrada la vamos a multiplicar por un valor negativo para revertir ese efecto.

From sklearn.model\_selection import cross\_val\_score

resultados = cross\_val\_score(reg\_arbol, df\_preparado, df\_label, scoring=”neg\_mean\_squared\_error”,cv = 10)

rmse = np.sqrt(-resultados)

rmse

Text

Description automatically generated

Después de hacer esta cross-validation nos podemos dar cuenta de que los resultados son incluso peores que con la regresión lineal, con la regresión lineal teníamos 69000 de error, aquí tenemos errores de más de 100,000.

Por esta razón hemos confirmado que el modelo de regresión de árboles de decisión está haciendo overfitting.

Por último, vamos a probar con un modelo llamdo **random fores**t o **bosque aleatorio.**

Los Bosques aleatorios funcionan porque entrenan muchos ARBOLES de decisión en diferentes subsets de las variables, y luego sacan un promedio de sus predicciones. (¿Entienden el porqué del BOSQUE? ¿Por los árboles de decisión?)

Armar un modelo que se compone de otros modelos se llama Ensemble Learning y es una genial manera de empujar a los algoritmos de Aprendizaje de Máquina al límite.

Vamos a profundizar por completo en este tema en el capítulo 7, por lo que solo vamos a probarlo con nuestros datos.

Usaremos la clase de **RandomForestRegresor** y repetiremos los mismos pasos que con los otros modelos.

from sklearn.ensemble import RandomForestRegressor

reg\_forest = RandomForestRegressor()

reg\_forest.fit(df\_preparado,df\_label)

prediccion = reg\_forest.predict(df\_preparado)

error = mean\_squared\_error(df\_label,prediccion)

error = np.sqrt(error)

error



En el RMSE tenemos un error de 24000, el cual ya es mejor resultado que la regresión lineal y no parece señal de overfitting.

Vamos a hacer la cross-validation para confirmar que no haya overfitting.

resultados = cross\_val\_score(reg\_forest, df\_preparado, df\_label, scoring="neg\_mean\_squared\_error",cv = 10)

rmse = np.sqrt(-resultados)

rmse

Text

Description automatically generated

Después de hacer esta validación nos damos cuenta de que, si tiene mejores resultados, aunque sigue habiendo algo de overfitting.

Algunas formas de solucionar el overfitting, puede ser simplificar el modelo, obtener más variables, obtener más datos.

Es recomendable también probar más modelos de otra rama de ML, como redes neuronales, máquinas de vectores, entre otros.

Lo normal es tener una lista de 2 a 5 modelos que sean prometedores y probarlos.

* **Afinar Tu Modelo**

Supongamos que ya tienes tu lista de modelos prometedores. Ahora necesitas afinarlos.

### **Grid Search**

A esta opción se le conoce como grid search, que en español significa búsqueda en la cuadricula.

Una opción para reducir el overfitting es probar manualmente añadiendo y modificando hiperparámetros, aunque normalmente hay tantas combinaciones posibles que puede resultar lento y tedioso.

Aquí es donde entra el **grid search**, este es una función, donde tienes que ingresar con que hiperparámetros quieres experimentar y la función experimenta con cross-validation para decirte cual es el que funciona mejor.

No hay que preocuparse mucho si todavía no se entiende bien exactamente para que funcionen los hiperparámetros, ya que se explicaran a fondo en el capítulo 7.

Solo hay que saber que son valores que modificamos nosotros para ver si nos ayudan a que el modelo se ajuste mejor.

Un ejemplo es el factor que usamos en nuestra función personalizada para remover los outliers, donde si lo incrementabas, la función eliminaba más valores, y si lo disminuías eliminaba menos.

En este caso, para calcularlo vamos a usar la función de scikit-learn **GridSearchCV.**

from sklearn.model\_selection import GridSearchCV

param\_grid = [{

'n\_estimators': [3,10,30], 'max\_features': [2,4,6,8]

}]

grid\_search = GridSearchCV(reg\_forest,param\_grid,cv=5,scoring='neg\_mean\_squared\_error',return\_train\_score=True)

grid\_search.fit(df\_preparado,df\_label)

Vamos a explicar paso a paso cómo funciona:

* Primero se tiene que crear la ventana de parámetros, estos se dividen en 2, los estimadores y el máximo de variables. Lo que hicimos en esta funcion fue darle 3 estimadores y cada uno de esos estimadores, se va a probar con cada número de variables, hasta que se encuentre la mejor combinación.
* El objeto necesita introducir varios parámetros.
* El modelo que queremos afinar, en este caso random forest.
* Nuestros parámetros
* El tipo de medida de error
* Preparamos los datos con nuestro data set de predictores y la variable a predecir
* Revisamos los resultados con los metodos best\_parameter\_ y best\_score\_





Lo que el resultado es que el estimador 30 con el máximo de variables 8 nos dio el mejor resultado con 78557, lo cual no es muy bueno, es parecido a los otros modelos, por lo que en este caso no valdría tanto la pena usar hiperparámetros.

* **Búsqueda aleatoria**

La grid search es buena cuando se tiene un grado bajo de hiperparámetros por probar, pero cuando el rango de hiperparámetros por probar es más grande.

Es conveniente usar un tipo de búsqueda aleatoria, ya que esta toma valores aleatorios y hace un numero alto de combinaciones hasta encontrar el mejor resultado.

Por ejemplo, si dejas que esta búsqueda se itere 1000 veces, el algoritmo habrá buscado con mil diferentes combinaciones de hiperparámetros, lo que te da muchas más probabilidades de obtener un buen resultado.

Este método se puede utilizar con la clase de scikit-learn **RandomSearchCV**.

* **Métodos de Ensamble**

Otra forma de afinar tu modelo es combinar modelos que tienen un buen desempeño, especialmente si tienen errores diferentes. Este tema lo cubriremos a fondo en el capítulo 7.

* **Analizar Los Mejores Modelos Y Sus Errores**

Otra forma de obtener ideas sobre cómo mejorar tu modelo es inspeccionar a estos. La mayoría de las clases de scikit-learn tienen métodos para observar cuanto pesa cada variable en el modelo.

Por ejemplo, si tu modelo arroja que hay una variable es menos útil para las predicciones, una buena idea sería eliminar esa variable

### **Evaluar Tu Sistema Con El Set De Prueba**

Después de experimentar con tus modelos eventualmente vas a encontrar uno que tenga un buen desempeño. Ahora llego el momento de evaluar el modelo final con el set de pruebas.

Lo único que tienes que hacer es separar las variables numéricas de la variable a predecir, ejecutar tu pipeline y transformar los datos.

Cuidado porque el set de prueba no se adapta, no hay que usar la función fit, solo la de transform.

modelo\_final = grid\_search.best\_estimator\_

Y = cat\_set\_prueba["average\_price"].copy()

X = cat\_set\_prueba.drop("average\_price",axis=1)

X\_preparada = pipeline\_completo.transform(X)

prediccion\_final = modelo\_final.predict(X\_preparada)

mse\_final = mean\_squared\_error(Y, prediccion\_final)

rmse = np.sqrt(-mse\_final)

np.sqrt(mse\_final)



Si en tu set de entrenamiento se hizo mucha modificación de hiperparámetros, muy probablemente el desempeño del set de prueba sea un poco peor.

Cuando esto sucede es importante resistir la tentación de agregar hiperparámetros al set de prueba, ya que estos cambios no se generalizarán en datos nuevos.

Ahora viene la parte de antes del lanzamiento, donde necesitas presentar tu solución, tienes que presentar que hiciste, que aprendiste, que funciono y que no funciono.

Hay que documentar todo, crear una presentación con imágenes claras y sencillas.

En este ejemplo de viviendas en Londres, aunque logramos un desempeño regular, todavía es una buena usarlo, especialmente si le ahorramos algo de tiempo a los expertos que pueden mejorarlo aún más.

* **Ejecutar, Monitorear Y Mantener tu Sistema**

Una vez que tienes aprobación para ejecutar tu sistema. Puede adaptar tu modelo a la producción que se tenga en la empresa. Una forma de hacer esto es incluir todo el preprocesamiento de datos y tu pipeline completo.

Por ejemplo, el modelo podría ser usado en una página web, donde el usuario presionara un botón para estimar el precio de una vivienda en cierto municipio.

En una aplicación primero se mandará una solicitud al servidor, tu modelo lanzará una predicción y la regresará a la aplicación.

Este proceso se hace por medio de funciones llamada API’s y son bastante utilizadas en el diseño web.

Otra forma de implementarlo puede ser, ejecutándolo en algún servicio de nube, en donde las predicciones pueden ser solicitadas en cualquier momento.

Hay un grupo de aplicaciones que pueden ayudarte a implementar estas herramientas web, y que están diseñadas específicamente para Python.

Por ejemplo:

* Flask
* Django
* Joblib

Entre bastantes otras.

Hay que tomar en cuenta que el lanzamiento no es el final de la historia. También hay que desarrollar un código para monitorear que tu algoritmo sigue funcionando y esta hacendo predicciones dentro de tus estándares de desempeño.

Por ejemplo, en este caso, si se combina tu algoritmo con el de inversiones para la empresa, y al final resulta que las inversiones no fueron buenas y se está perdiendo mucho dinero.

Pues probablemente el problema este en alguno de los modelos, por lo que es una buena idea desarrollar un mecanismo que te alerte cuando las inversiones no estén funcionando.

* **¡Sigue practicando!**

Esperemos que este proyecto te haya dado una buena idea de cómo funciona un proyecto de ML.

Como pudiste observar, mucho del trabajo consiste en el procesamiento de datos, creación de pipelines, elección de modelos.

Algo muy importante a tomar en cuenta en ML, es que claro, los tipos de modelos son importantes. Pero es mejor sentirse cómodo sabiendo cómo funciona todo el proceso de un proyecto e ML, aunque solo conozcas 3 modelos. A conocer 100 modelos y o estar seguro de cómo es todo el proceso de antes de eso.

Es buena idea explorar nuevos proyectos en sitios web como Kaggle y veras como poco a poco, iras mejorando bastante.