

Data Science & Machine Learning

Módulo 5: Data Science & Machine Learning (ML)

22 al 26 de mayo - Madrid

Escrito e impartido por: Carl McBride Ellis

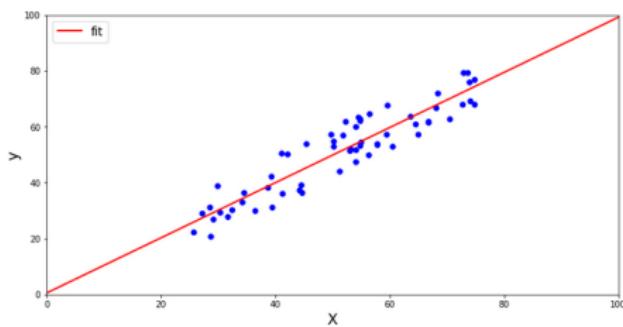
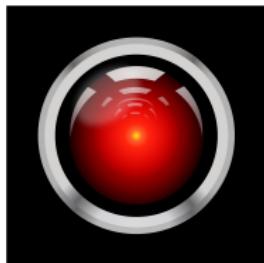
(v 0.14.0)

Principales áreas de “AI”:

- inteligencia artificial (AGI): reinforcement learning
 - juegos ← ajedrez, go, **Gym Retro** (Atari, Sega, etc.),...
- generative AI, con redes neuronales (NN):
 - prompt ⇒ texto, voz, imagen, video
- deep-learning con redes neuronales:
 - computer vision (CV) ← imágenes o video
 - procesamiento de lenguaje natural (NLP y LLM) ← texto

Esta semana nos vamos a enfocar en el área de machine learning (ML)

- machine learning (ML) ← datos tabulares o ‘structured data’
- **ML ≠ “inteligencia artificial” (AI)**
- **ML = ajustes de datos usando la estadística**

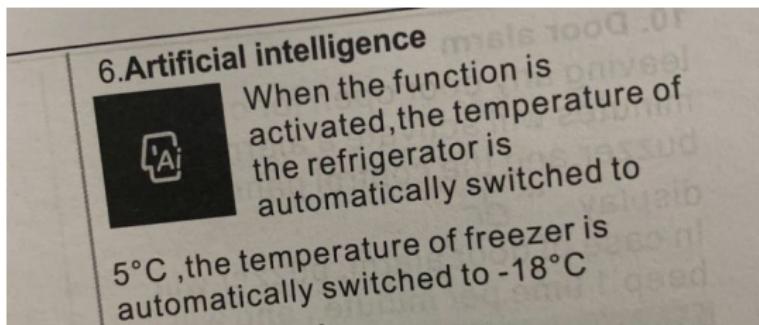


¿Que es un 'científico de datos'?

*“...es alguien que es mejor en estadística que un informático,
y mejor en informática que un estadístico”*

Forty percent of 'AI startups' in Europe don't actually use AI, claims report

"...40 percent of European startups that are classified as AI companies don't actually use artificial intelligence in a way that is "material" to their businesses."



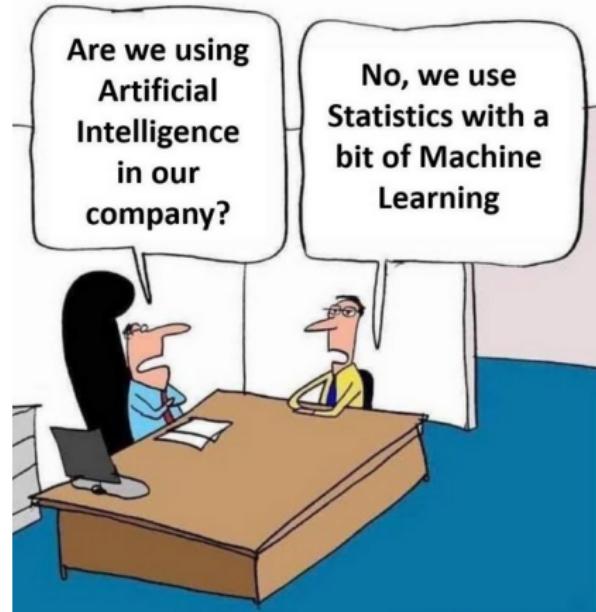
"I invented the term artificial intelligence ...when we were trying to get money"

John McCarthy

Informe:

“Uso de inteligencia artificial y big data en las empresas españolas” (2023)

- El uso de inteligencia artificial en las empresas ha aumentado hasta el 11,8%, 3,5 puntos porcentuales más que el año pasado.
- Se ha producido un aumento del 1,4 al 2,3% en el empleo de especialistas tecnológicos en IA.
- La estrategia España Digital 2026 fija la meta de que para 2025 el 25% de las empresas españolas usen inteligencia artificial y big data.



¿Por qué hacemos ML?

$\hat{\beta}$: **Entender (inferencia)**: llegar a conocer qué factores influyen en y

\hat{y} : **Predecir**: crear un modelo predictivo en y

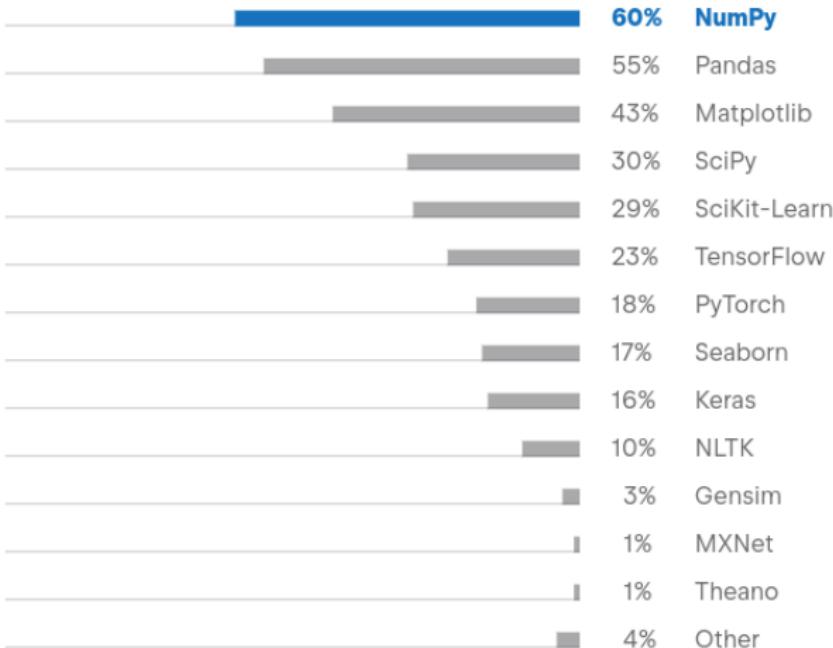
(Leo Breiman "Statistical Modeling: The Two Cultures")

Analytics, key performance indicators (KPI) y dashboards aportan información sobre el estado actual
pero solo un **modelo** puede hacer predicciones!

“Sin datos, no eres más que otra persona con una opinión”
W. Edwards Deming

Data science frameworks and libraries

100+



(Fuente: "Python Developers Survey 2021 Results")

PyData



otros ML frameworks:

-  H2O.ai
-  Spark MLlib
-  RAPIDS (gpu)

para redes neuronales artificiales (ANN) / deep learning

-  TensorFlow
-  keras API
-  PyTorch
-  Hugging Face

cloud end-to-end service

- Google Vertex AI
- Amazon SageMaker
- Microsoft Azure AI

Vamos a usar [python](#) + Jupyter notebooks



alternativas: [R](#) + RStudio IDE (ya llamado *posit*) (¡ojo! "[The R Inferno](#)")



y también: [julia](#) + Pluto notebooks



Herramientas esenciales

Entornos:

- Jupyter notebooks en [Jupyter Lab](#) / [Google Colab](#) / [Kaggle](#) / ...

`(pip3 install jupyterlab)`

Lenguaje:

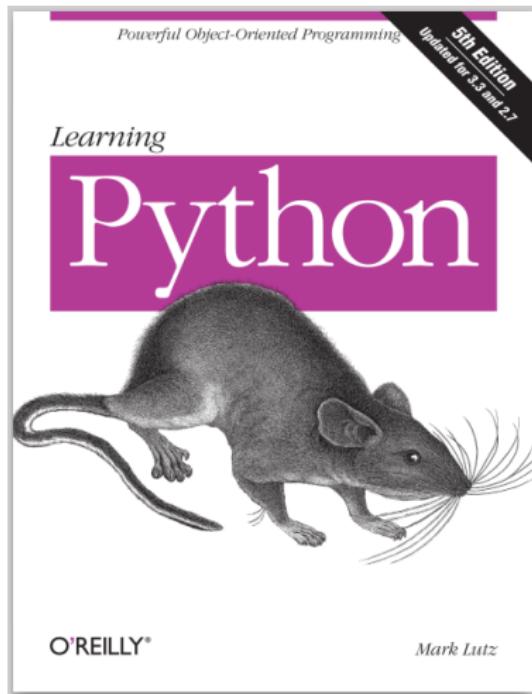
- [python](#) (24 de octubre 2022: versión 3.11)

Librerías:

- [numpy](#) (ver tb. [numba](#), [numexpr](#))
- [pandas](#)
- [matplotlib](#)
- [scikit-learn](#)

(Artículo: “[Why Jupyter is data scientists' computational notebook of choice](#)” [Nature](#) (2018))

Libro: "Learning Python"



(¡ojo: 1.643 paginas!)



- > 227.000 [datasets](#) - la gran mayoría en formato CSV
- ≈ 905.000 [python notebooks](#)
- > 13,5 millones de usuarios
- [Microcursos](#) + certificados
- CPU: 12 horas por notebook, 30GB RAM y 4 cores
- GPU: 2x[T4](#) o [Tesla P100](#), 13GB RAM > 30h/semana
- ...y *todo gratis!*

(Fuente: "[Kaggle in Numbers](#)")

Texto en tu notebook

Título
subtítulo
sub-subtítulo

[enlaces] (<https://www.url.es>)

Para más ver: [Markdown for Jupyter notebooks cheatsheet](#)

Ecuaciones usando L^AT_EX markup *p.ej.*

in-line \$ a=b+c \$ y stand-alone

\$\$ a=b+c \$\$

Para más ver: [Beautiful math in your notebook](#)

¡Competición!

¡Esta semana vais a ser data scientists!

A lo largo de la semana vais a hacer vuestro propio proyecto de ML.
Vais a predecir el precio de un piso.

El ganador será quien tenga la mejor puntuación ('score') el viernes a las 16:50. También habrá una 'mención de honor' al notebook mejor presentado.

Enlace al concurso: [Pisos competición: Madrid](#)

Introducción

ML es, en su esencia, un ajuste de datos: estamos buscando una función (llamada el '*estimator*' o '*learner*') y sus parámetros que constituyan la mejor aproximación a los datos reales.

Supuesto básico de ML:

$$\text{los datos} = \text{señal} + \text{ruido}$$

es decir

$$y = f(x) + \varepsilon$$

donde ε es el '*error irreducible*'

Manipulación de datos:



pandas

([pandas 2.0](#): abril 2023)

¿Que son “datos tabulares” ?

Ejemplo de un DataFrame (frecuentemente abreviado como `df`) :

	Asset_ID	Count	Open	High	Low	Close	Volume	VWAP	Target	Asset name
timestamp										
2018-01-01 00:01:00	1	229.0	13835.194000	14013.800000	13666.11	13850.176000	31.550062	13827.062093	-0.014643	Bitcoin
2018-01-01 00:02:00	1	235.0	13835.036000	14052.300000	13680.00	13828.102000	31.046432	13840.362591	-0.015037	Bitcoin
2018-01-01 00:03:00	1	528.0	13823.900000	14000.400000	13601.00	13801.314000	55.061820	13806.068014	-0.010309	Bitcoin
2018-01-01 00:04:00	1	435.0	13802.512000	13999.000000	13576.28	13768.040000	38.780529	13783.598101	-0.008999	Bitcoin
2018-01-01 00:05:00	1	742.0	13766.000000	13955.900000	13554.44	13724.914000	108.501637	13735.586842	-0.008079	Bitcoin
...

Es un especie de matriz compuesta de datos

Los datos *casi siempre** tienen que ser numéricos (float o integer),

Hablamos de filas (*rows*) y columnas (*columns*)

Las columnas se conocen como *features* (características)

(*) excepciones son, p.ej. timestamps en series temporales ([RFC 3339](#))

Un dataframe tiene una fila especial con los nombres de las columnas.
Eso se llama la “**header**”

También tiene una columna especial para numerar las n filas
 $[0, \dots, n - 1]$. Eso se llama la “**index**”

Los datos suelen venir en formato csv (*comma-separated values*)

Los leemos usando pandas, por ejemplo:

```
import pandas as pd  
df = pd.read_csv("mis_datos.csv")
```

En este momento, ya existe una copia del fichero `mis_datos.csv` en la memoria del notebook, como la entidad `df` en formato “dataframe”

Tb. se puede leer ficheros excel con `pd.read_excel()`

(requisito: `pip3 install xlrd`)

algunos opciones útiles de lectura

- `decimal=", "` – comas decimales en lugar de puntos
- `sep=";"` – si tu csv es un ;sv
- `delim_whitespace=True` – por defecto es False
- `index_col="new_index"` – nominar una columna a ser el índice
- `nrows=100_000` – solo leer las primeras 100k filas

Nota: Aunque solemos leer ficheros .csv, los dataset pueden venir en otros formatos, *p.ej.:*

- `JSON/JSONL`
- `Avro`
- `ORC`
- `feather`
- `hdf5`
- `jay`
- `parquet` (guardado en columnas)
- bases de datos + SQL + `pySpark`

también hay alternativas, mucho más rápidas, al pandas:

- **Modin** (`import modin.pandas as pd`)
- **dask**
- **PyArrow**
- **datatable**
- **rapids** (usando los GPU)
- **vaex** (lazy Out-of-Core DataFrames)
- **polars** (guardado en columnas)

("*Pandas Is Not Enough? A Comprehensive Guide To Alternative Data Wrangling Solutions*")

Para saber más sobre pandas:

manual 'on-line':

- [pandas User Guide](#)

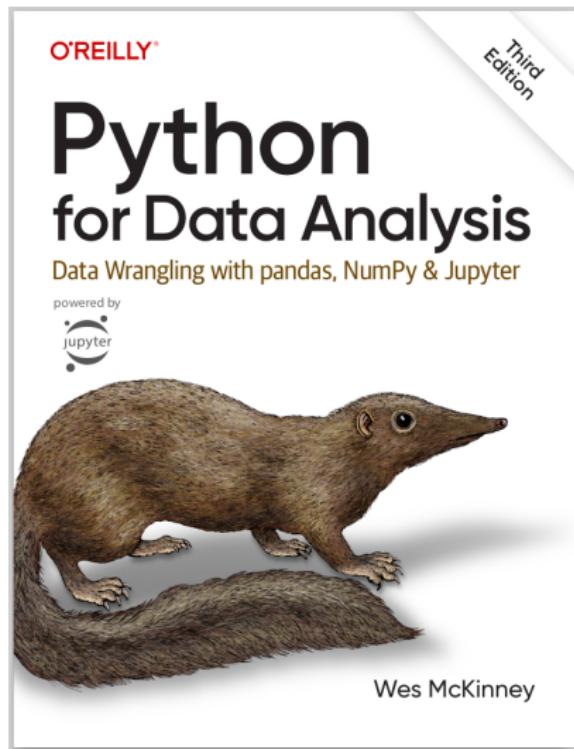
libros:

- “*Python for Data Analysis: Data Wrangling with pandas, NumPy, and Jupyter*” Wes McKinney
- “*LEARNING pandas*” Stack Overflow contributors (gratis)

notebook ‘chuleta’:

- [Pandas one-liners](#)

Libro: "Python for Data Analysis: Data Wrangling with pandas"



Como científico de datos tus notebooks (o scripts) de python casi siempre van a empezar con importando estas tres librerías:

```
import numpy as np  
import pandas as pd  
import matplotlib.pyplot as plt
```

Cómo instalar paquetes o librerías nuevas:

```
!pip install nombre_del_paquete
```

Se pueden encontrar muchos paquetes interesantes en

The Python Package Index (PyPI)

en la sección de

Scientific/Engineering :: Artificial Intelligence

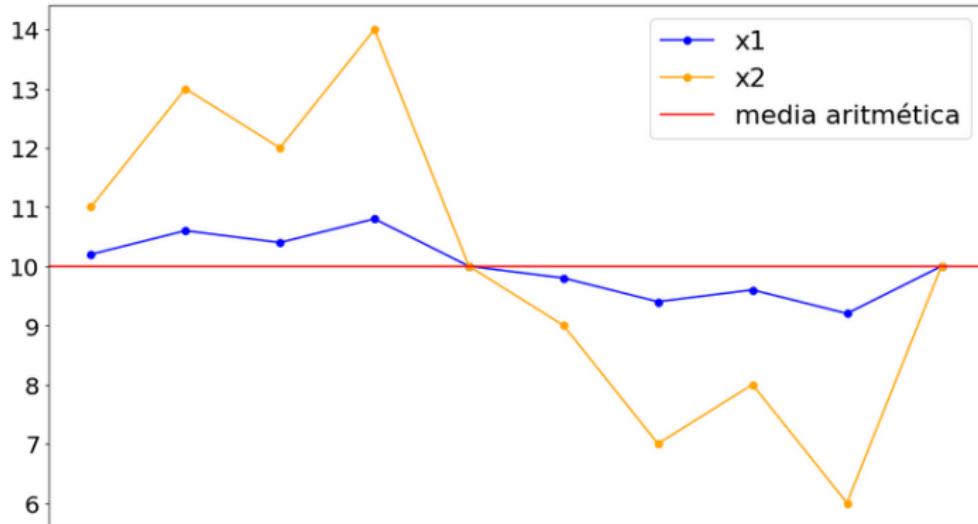
Estadística básica

Aunque para entender en profundidad ML hace falta un buen conocimiento de estadística, en términos prácticos se puede ir bastante lejos solo con saber la media y la varianza:

- media aritmética $\mu = \bar{y} = \frac{1}{n} \sum_{i=1}^n y_i = \text{.mean()}$
- varianza $\sigma^2 = \frac{1}{n} \sum_{i=1}^n (y_i - \bar{y})^2 = \text{.var()}$
- nota que σ es conocida como la “desviación estándar” (`.std()`)

La varianza, y por tanto la desviación estándar, son medidas de dispersión

Ejemplo:



$$\text{media: } \bar{y}(x_1) = \bar{y}(x_2) = 10$$

$$\text{varianza: } \sigma^2(x_1) = 0.26, \sigma^2(x_2) = 6.6$$

$$\text{desviación estándar: } \sigma(x_1) = 0.52, \sigma(x_2) = 2.58$$

Vemos que la desviación estándar tiene valores más “intuitivos” que la varianza

Fuente: [BIG DATA: Media aritmética y varianza](#)

Aunque el ruido (ε) es aleatorio, si se tiene un número grande de muestras finalmente se aproxima a una [distribución Gaussiana](#).

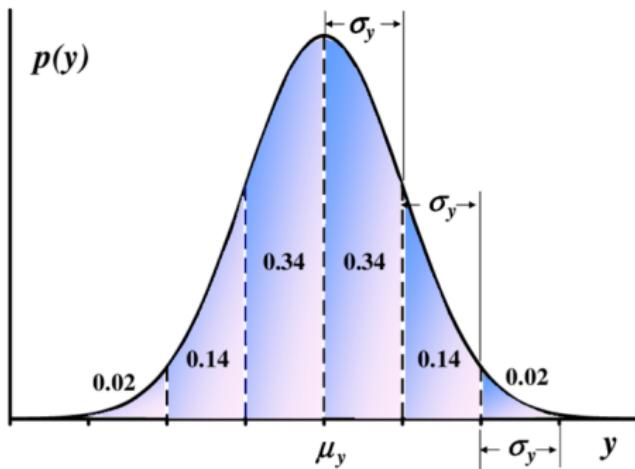
En matemáticas eso se llama el [teorema del límite central](#)

(Pierre-Simon Laplace 1812)

(Notebook: “[Animated histogram of the central limit theorem](#)”)

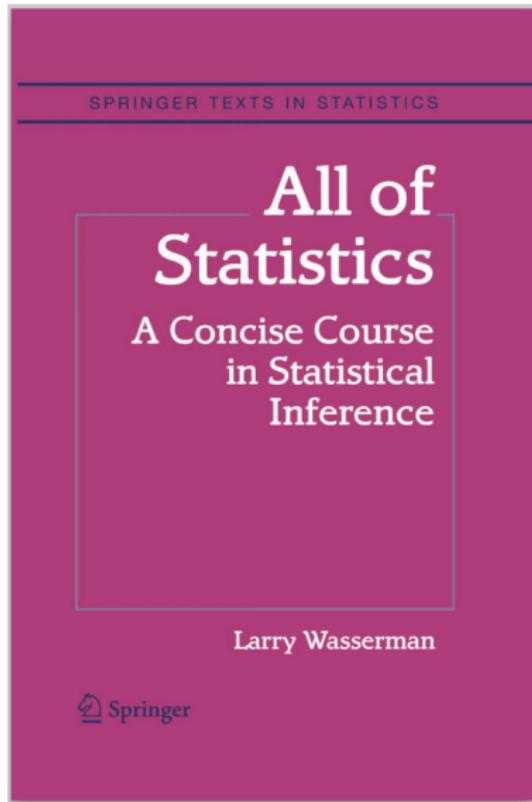
¿Qué distribución tiene la función de densidad de probabilidad Gaussiana?

([scipy.stats.norm.pdf\(\)](#))



donde μ es la media, y σ es la desviación estándar
(ver tb. [la regla 68% - 95% - 99.7%](#))

Libro: "All of Statistics"



Análisis exploratorio de datos (EDA)

¿Por qué exploración y análisis (EDA)?

“Basura entra, Basura sale” (GIGO)

Llegando a conocer a tu dataframe (`df`):

- `df.shape` - da el número de rows y columnas del df
- `df.head(n)` - imprime los primeros n rows
- `df.tail(n)` - imprime los últimos n rows
- `df.dtypes` - da los tipos de las columnas (float, integer, object, etc.)
- `df.describe()` - algunas estadísticas descriptivas de cada columna

EDA ¿Por qué y cómo?

- Anscombe's quartet
- box y raincloud plots
- valores anómalos, outliers y inliers
- normalidad, skewness o curtosis
- correlación (Pearson)
- pair plots
- histogramas
- ayudar con feature selección y engineering - nos da ideas

Visualización de datos: “El cuarteto de Anscombe” (año 1973):

Parameter	Data set 1	Data set 2	Data set 3	Data set 4
\bar{x}	9.0	9.0	9.0	9.0
\bar{y}	7.500909090909093	7.50090909090909	7.5	7.500909090909091
Slope	0.5000909090909094	0.5000000000000003	0.49972727272727313	0.4999090909090908
Intercept	3.000090909090908	3.00090909090909	3.0024545454545453	3.001727272727274
Sum of squares of $x - \bar{x}$	110.0	110.0	110.0	110.0
Regression sum of squares	27.5100009090904	27.500000000000025	27.470008181818226	27.490000909090906
Residual sum of squares of y	13.76269	13.77629181818182	13.756215454545455	13.742493636363639
Standard error of the slope	0.11790550059563408	0.1179637498598266	0.1178777634916929	0.11781895731756249
R^2	0.666542459508775	0.6662420117029543	0.6663234677330083	0.6667071687067612

Figure 1

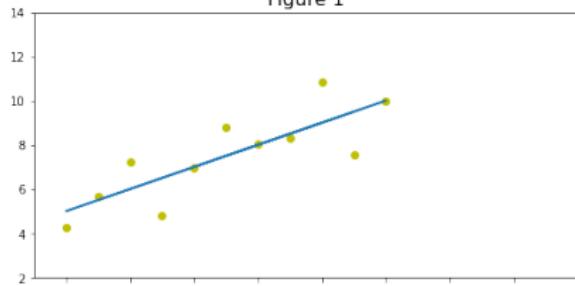


Figure 2

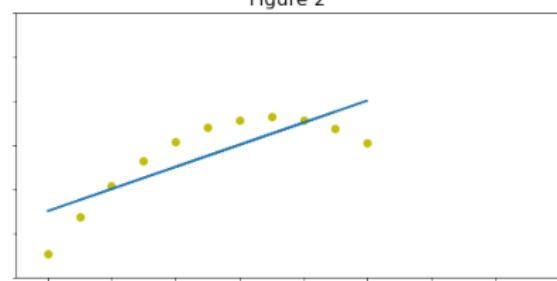


Figure 3

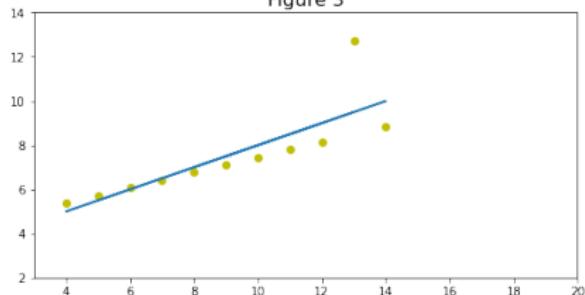
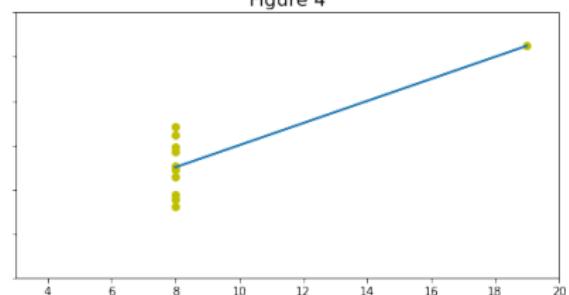


Figure 4



Notebook: [Anscombe's quartet and the importance of EDA](#)

El cuarteto de Anscombe:

Vemos cuatro dataset que tienen los mismos estadísticos descriptivos, pero en realidad son muy distintos:

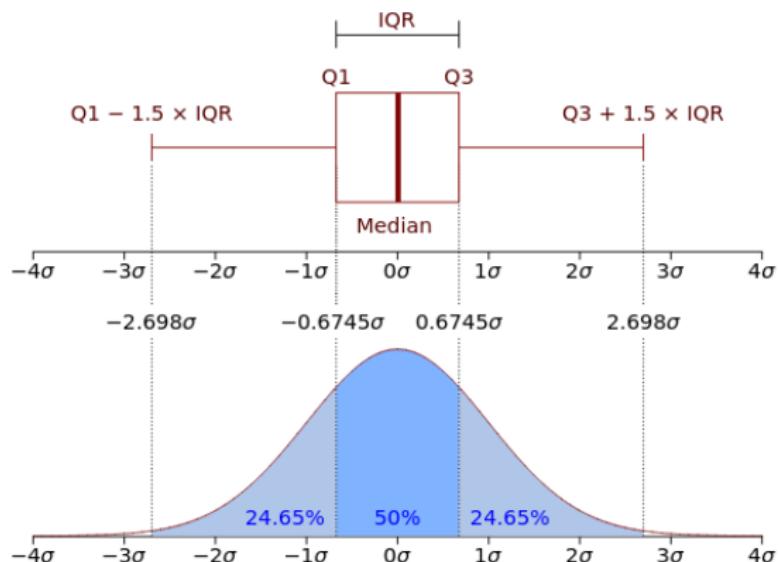
- 1. tenemos un buen ajuste con una linea recta
- 2. hay 'underfitting'
- 3. tenemos un punto 'outlier' (valor anómalo)
- 4. tenemos un punto de 'leverage'

Mensaje: hay que visualizar tus datos; no basta con solo calcular números

Versión moderna: [*The Datasaurus Dozen*](#)

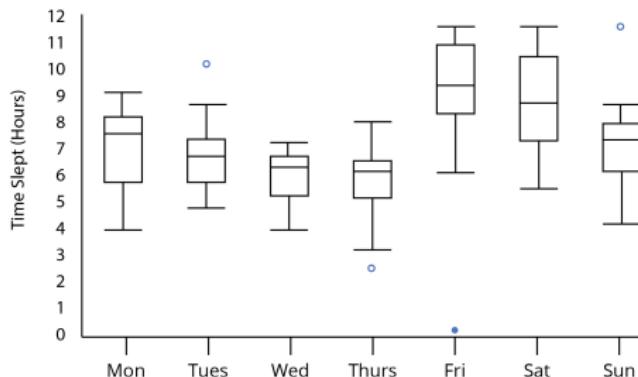
Rango intercuartílico (*interquartile range (IQR)*):

Es el rango entre 25% y 75% de los datos



nota que nuestros datos no (¡básicamente nunca!) son necesariamente Gaussianas

Box plots:



mejor todavía: Box + strip + violin = raincloud plot

(Nick Desbarats "*I've Stopped Using Box Plots. Should You?*")

Valores *outlier* (valores atípicos):

Nota: Es importante a diferenciar entre 'outliers': valores extremos \neq datos malos

causas:

- error humano al anotar un dato
- error de medida
- ...

por ejemplo

- el dato de una persona que mide 6 metros de altura (alguien confundió las unidades de pies frente a metros)
- una temperatura de -150 grados en el Sahara; un termómetro se ha roto

Detección de outliers:

- Visualización *p.ej.* con un box o raincloud plot:
Box+strip+violin = raincloud plot
- $z\text{-score} = \frac{X - \bar{x}}{\sigma} = \text{stats.zscore()}$

Por ejemplo:

```
from scipy.stats import zscore
df_z = df.apply(zscore)
```

valores $> |3|$ están más lejos de μ que 99.7% del resto de los datos

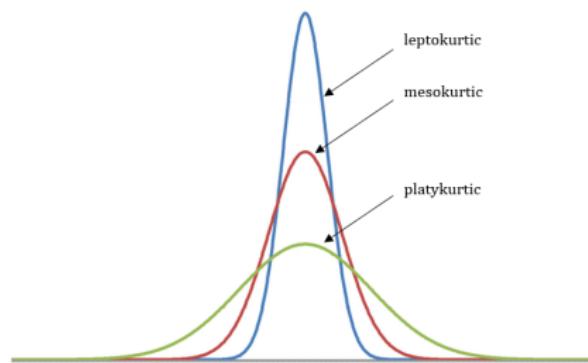
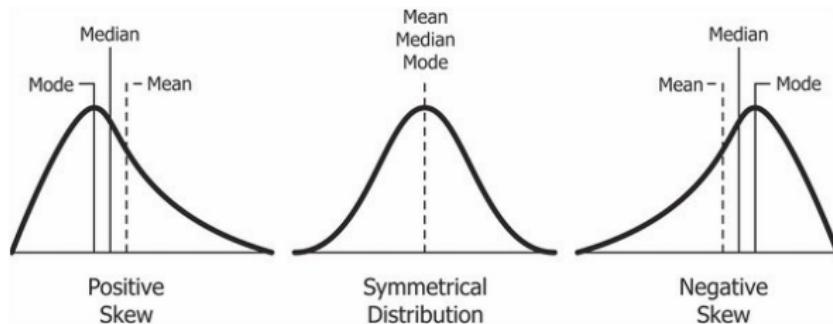
(Nota: El z-score no es capaz de detectar 'inliers')

(ver tb. la [distancia de Mahalanobis](#))

- **sklearn.ensemble.IsolationForest**

(Notebook: [Filtering outliers using the Isolation Forest](#))

EDA: Ver si los features tienen skewness o kurtosis



Eso se puede "corregir" con una transformación de Yeo-Johnson (scikit `power_transform`)

Hipótesis nula (H_0): los datos siguen una distribución normal

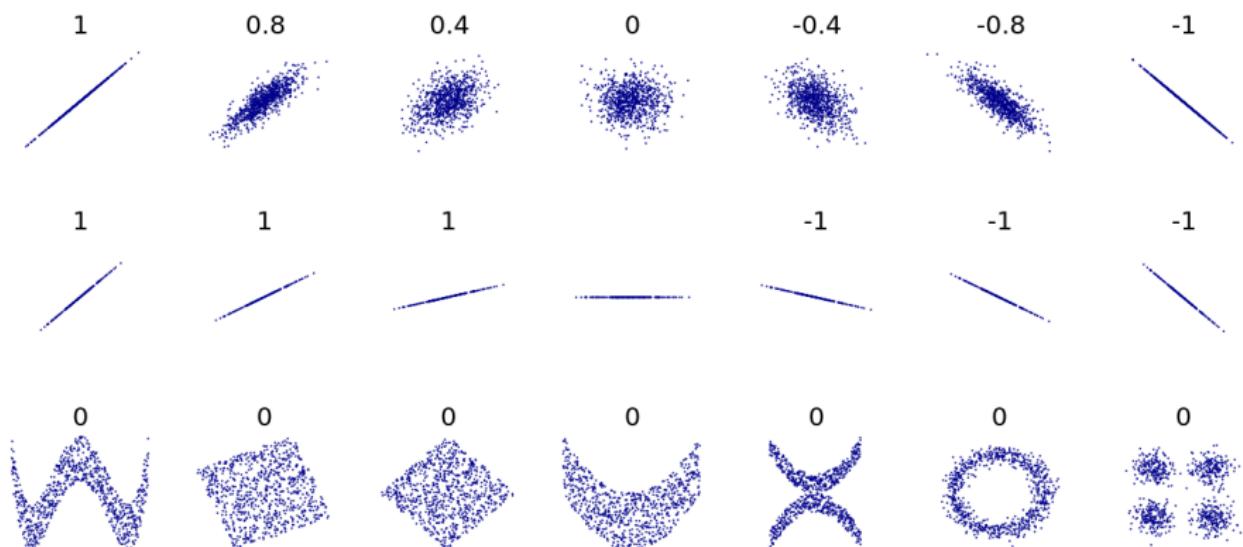
- Prueba de skewness
- Prueba de curtosis

Pruebas de normalidad

- Prueba de Shapiro-Wilk
- Prueba de D'Agostino
- Prueba de Anderson-Darling

(Artículo: "[Comparisons of various types of normality tests](#)")

Correlación de Pearson (rango $[-1, 1]$)
(recuerda: correlación no implica causalidad)



(ver tb.: "Pearson correlation coefficient, mutual information (MI) and Predictive Power Score (PPS)")

Matriz de correlación:

	date	weight	resp_1	resp_2	resp_3	resp_4	resp	feature_0	feature_1	feature_2
date	1.00	0.02	0.02	0.01	0.01	-0.02	-0.03	0.01	-0.05	-0.01
weight	0.02	1.00	-0.01	-0.01	-0.02	-0.02	-0.02	-0.02	-0.05	0.06
resp_1	0.02	-0.01	1.00	0.92	0.80	0.49	0.58	0.04	0.03	0.02
resp_2	0.01	-0.01	0.92	1.00	0.91	0.55	0.68	0.04	0.04	0.01
resp_3	0.01	-0.02	0.80	0.91	1.00	0.77	0.80	0.03	0.04	0.01
resp_4	-0.02	-0.02	0.49	0.55	0.77	1.00	0.95	-0.06	0.00	-0.04
resp	-0.03	-0.02	0.58	0.68	0.80	0.95	1.00	-0.06	0.00	-0.04
feature_0	0.01	-0.02	0.04	0.04	0.03	-0.06	-0.06	1.00	0.03	0.03
feature_1	-0.05	-0.05	0.03	0.04	0.04	0.00	0.00	0.03	1.00	0.84
feature_2	-0.01	0.06	0.02	0.01	0.01	-0.04	-0.04	0.03	0.84	1.00

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

Columnas con mucha correlación entre sí suelen ser malas; complican mucho la interpretación de un ajuste.

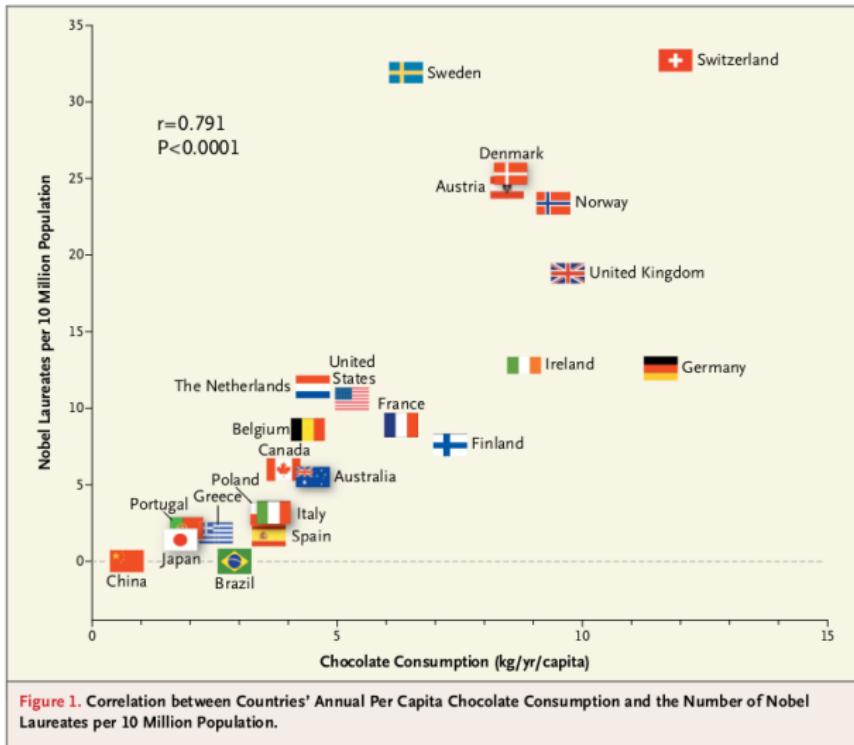
Tambien es posible que columnas con mucha correlación pueden ser básicamente la misma cosa, por ejemplo velocidad en kmh y en millas por hora.

Conviene quizás eliminar (*drop*) una de ellas:

```
df = df.drop(["kmh"], axis=1)
```

(ver tb.: [Explainability, collinearity and the variance inflation factor \(VIF\)](#))

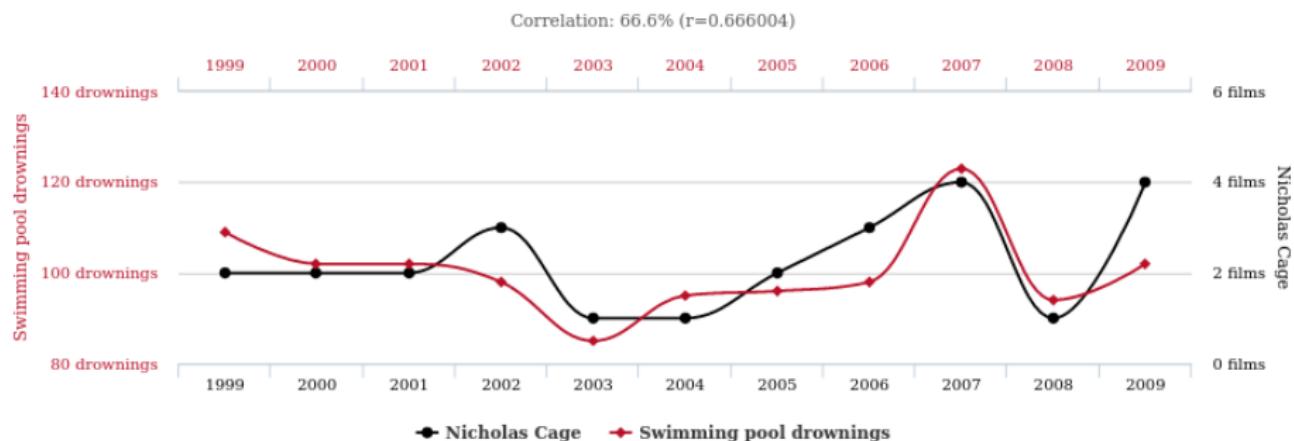
Correlación espuria: Regresión



(Fuente: "Chocolate consumption, cognitive function, and Nobel laureates" Franz H. Messerli, N. Engl. J. Med. (2012))

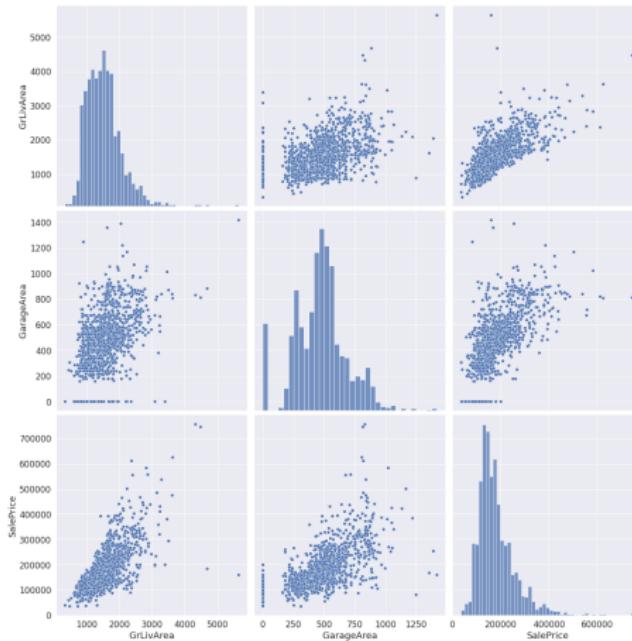
...y también para series temporales

Number of people who drowned by falling into a pool correlates with Films Nicolas Cage appeared in



(fuente: la pagina web [Spurious Correlations](#))

Pairplot:

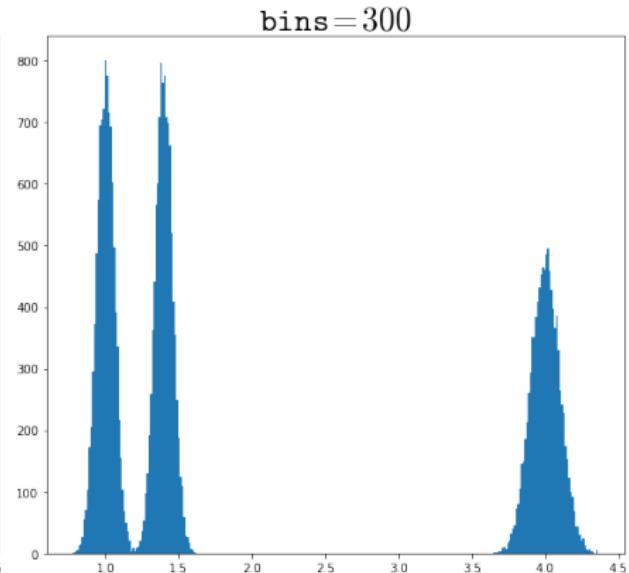
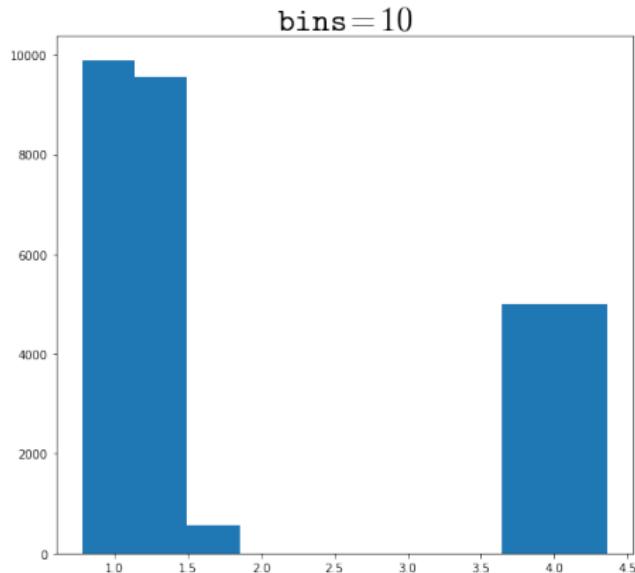


función: `sns.pairplot(df, height=3)`

(desventaja: si hay n features habrá n^2 gráficos)

Histogramas: Cuidado con el numero de '*bins*':

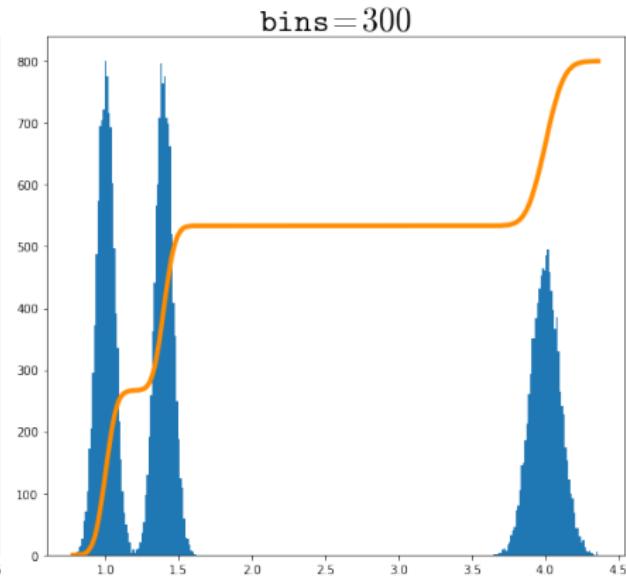
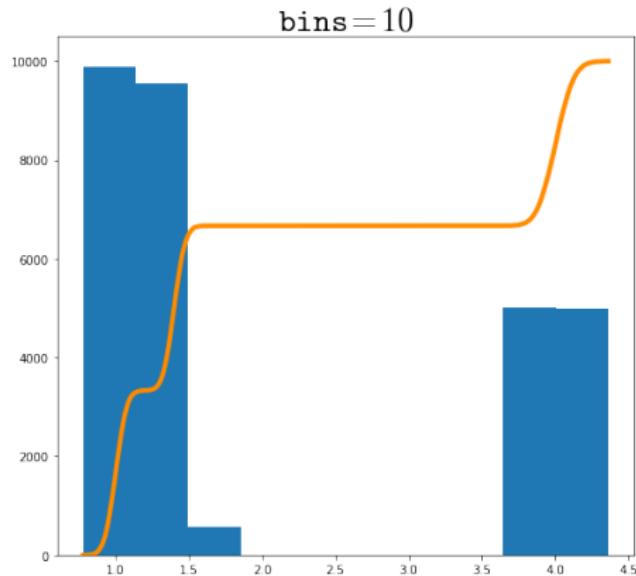
Estos son dos histogramas de **los mismos datos**:



Solución: el *empirical cumulative distribution function* (ECDF)

El ECDF es independiente del numero de bins

(p.ej. en seaborn usa cumulative=True o `seaborn.ecdfplot`)



Visualización de datos: las 3 librerías principales

- `matplotlib` – `import matplotlib.pyplot as plt`
- `seaborn` – `import seaborn as sns`
- `plotly express` (interactivo) – `import plotly.express as px`

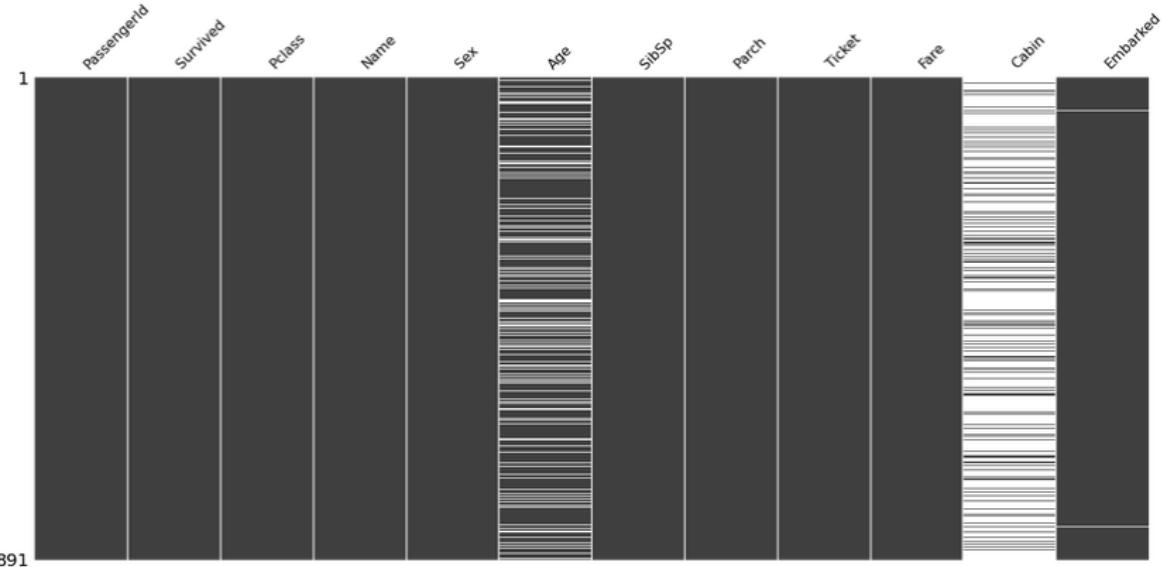
ejemplos de gráficos y su correspondiente código fuente:

- `matplotlib gallery`
- `seaborn gallery`
- `plotly express gallery`

Data cleaning (limpieza de datos)

- valores ausentes (valores 'nulos' o `NaN`)
- filtrando los outliers
- filas duplicadas
- columnas booleanas (`True` o `False`)
- features con cero varianza
- variables categóricas

Visualizar los valores no disponibles (“missing”):



Paquete: [missingno](#)

y en más detalle con Pandas:

- `df.isna().sum()` - número de valores nulos en cada columna
- `df.isna().sum().sum()` - número de valores ausentes en total

(Nota: `df.isnull()` y `df.isna()` son lo mismo)

¿Que hacer con los datos nulos (NaN)?: Imputación

- llenar con un constante
- llenar con la media o la mediana
- llenar con el valor más frecuente
- borrar (columna o fila)
- poner un flag (p.ej. `missing=-999`)
- otro método...

por ejemplo usando el [Simple Imputer](#) de scikit-learn (... si los datos son MCAR)

(Materia avanzada: [Iterative Imputer](#)/ [MissForest](#) en el paquete [missingpy](#))

Artículo: "[Benchmarking missing-values approaches for predictive models on health databases](#)" (2022)

Lo más fácil para empezar: reemplazar todos los NaN con *p.ej.* un 0:

```
df = df.fillna(0)
```

o se puede borrar todas las filas que contienen algún NaN con:

```
df = df.dropna()
```

o las columnas, con:

```
df = df.dropna(axis='columns')
```

Para llenar los NaN con la media:

```
df = df.fillna(df.mean())
```

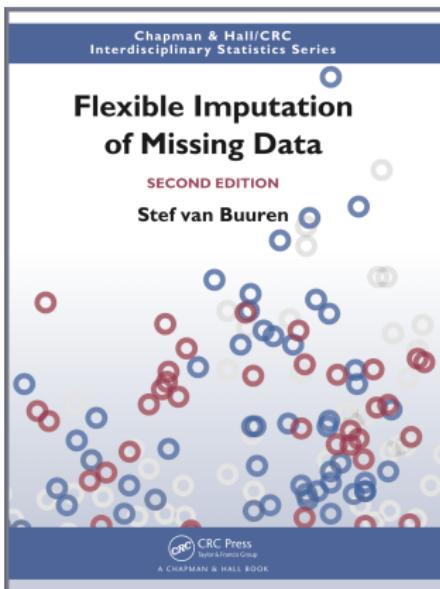
Para llenar los NaN con el valor más frecuente:

```
df['col'] = df['col'].fillna(df['col'].mode()[0])
```

como quitar los infinitos: `df = df.replace([np.inf, -np.inf], np.nan)`

	numero_hijos	edad	y
0	2	30	30000
1	2.0	NaN	42400.0
2	NaN	NaN	30000.0
3	0	28	38100
4	1	25	29700

Libro: "Flexible Imputation of Missing Data" (online)



¿Que hacer con los outliers?:

Se puede poner un '*flag*' feature, cambiar estos valores, o incluso borrar los datos outliers. *P.ej.* borrar los valores debajo de 100 y encima de 900 de la columna a:

```
df = df[ (df["a"] >= 100) & (df["a"] <= 900) ]
```

para ejemplos ver: [BIG DATA: Filtrando datos](#)

(Nota: Resulta más conveniente a filtrar los outlier *antes* de crear X e y, para borrar la fila entera)

(¡Ojo!: NO puedes filtrar/borrar filas del dataset `X_test` del concurso)

...pero a veces los outliers son interesantes, *p.ej.* son indicaciones de casos de fraude etc.

(Materia avanzada: [Python Outlier Detection Library \(PyOD\)](#))

Filas duplicadas

para contar el numero de filas duplicadas:

```
len(df) - len(df.drop_duplicates())
```

y para borrar filas duplicadas

```
df = df.drop_duplicates()
```

Cambiar columnas booleanas (`True` o `False`) a ser numéricas

```
df["col"] = df["col"].astype("int")
```

Eliminar columnas con cero varianza:

```
df = df.loc[:, df.var() != 0]
```

o quizás mejor

```
df = df.drop(df.columns[df.nunique() == 1], axis=1)
```

Categorical features

Variables categóricas (*categorical features*) (i.e. strings, o `dtype` object):
en general hay dos tipos

- **ordinal**: con orden, *p.ej.* $a < b < c$
- **nominal**: sin ningún orden, *p.ej.* Spain, USA, China

¿Qué hacer con los ordinales?

- Label Encoding: a → 1, b → 2, c → 3
- hacer label encoding a mano con un diccionario *p.ej.:*
`df["col"] = df["col"].replace({"Oro":1,"Plata":2,"Bronce":3})`
- `sklearn.preprocessing.OrdinalEncoder()`

¿Qué hacer con los nominales?: “One Hot encoding”

Ejemplo: el *feature* orientación de tipo object:

orientación
norte
este
norte
sur

se convierte en tres nuevas *features*, ya con valores binarios:

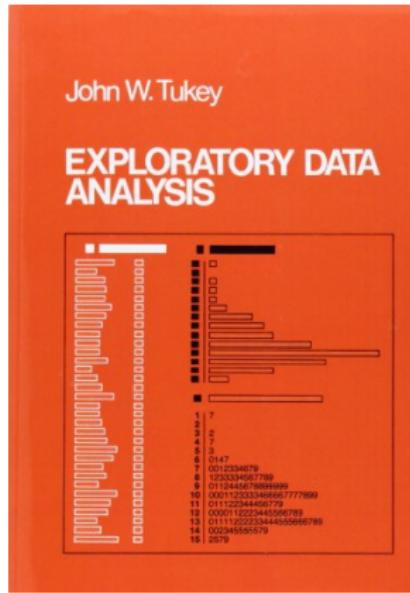
orientación_norte	orientación_este	orientación_sur
1	0	0
0	1	0
1	0	0
0	0	1

Nominal:

- “One Hot encoding”: crear un “bit-mask” con $n - 1$ columnas nuevas
- One hot encoding con la ayuda de `pandas.get_dummies`
`df = pd.get_dummies(df, drop_first=True)`
- `sklearn.preprocessing.OneHotEncoder()`

(Materia avanzada: Ver “Category Encoders”)

Libro (1977):



Sesión práctica: EDA

Sesión práctica: EDA

Jupyter notebook: [BIG DATA: Precio de pisos competición \(plantilla\)](#)

Objetivo: Vamos a explorar nuestro dataset de pisos

Aprendizaje automático (ML)

Fundamentos del aprendizaje automático (machine learning)

Dos tareas principales:

- regresión
- clasificación

¿Como sabemos si tenemos delante un problema de regresión o de clasificación?

Hay una columna muchas veces llamada ‘target’ para regresión, o ‘label’ para clasificación, y suele ser la primera o la última columna

Miramos al ‘target’; a ver si son valores continuos ($y \in \mathbb{R}$) o si son discretos ($y \in \mathbb{Z}$). Por ejemplo, en clasificación los valores suelen ser 0 o 1

```
df["label"].value_counts()
```

```
0    549  
1    342  
Name: label, dtype: int64
```

Nota que también pone: `dtype:int64`

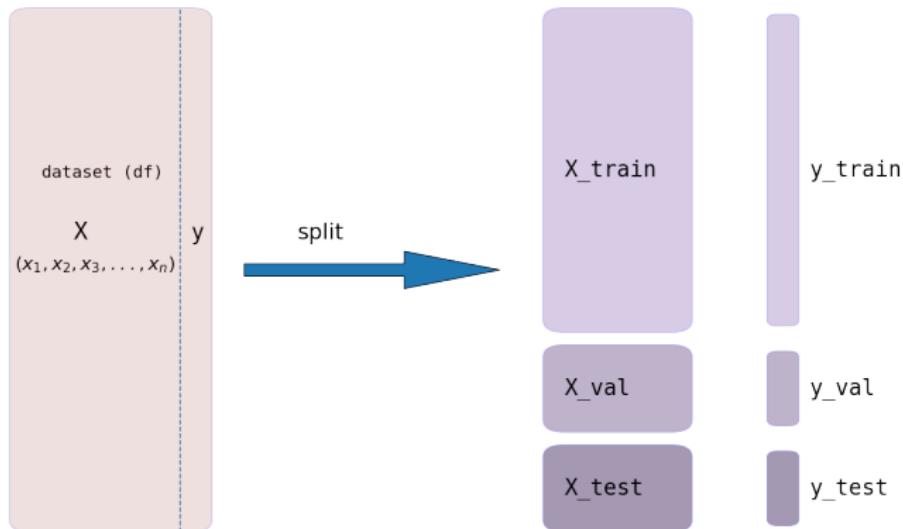
Datos tabulares y el dataframe

- las columnas son las “*features*”
- las filas son las “*feature vectors*”
- Para que ML funcione hacen falta, por lo menos, 1.000 filas, y además $n_filas > 20 \times n_columnas$

(ver: “*Curse of dimensionality*” / “*Hughes phenomenon*”)

Dataset splitting: Entrenamiento, validación y prueba final

`df → X_train, y_train, X_val, y_val, X_test, y_test`



Importante: Los datos `X_test` y `y_test` son de un solo uso!

Datos de entrenamiento, validación y prueba final:

- **train**: (los datos de entrenamiento) son para los parámetros
- **validation**: (or hold-out) es para los hiper-parámetros
- **test**: es para ver qué tal va tu modelo realmente

El set de validation debe tener aproximadamente el mismo tamaño que el set de test para optimizar el rendimiento del *metric*:

p.ej. relación aproximada: 80:10:10

Nota: Queremos que cada columna en los tres datasets sean **independientes e idénticamente distribuidas (iid)**: suele bastar simplemente con un *split* aleatorio.

Materia avanzada: [What is Adversarial Validation?](#)

Materia avanzada: [Prueba de Kolmogórov-Smirnov](#)

Apartado: Números (*pseudo*) aleatorios

En el código fuente de machine learning sueles ver mucho `random_state=42` o `seed=42`. Elegimos una semilla (`seed`) para que nuestros resultados sean '*reproducibles*'.

P.ej. usando el sencillo algoritmo RNG de Lehmer obtenemos:

`seed=1:` 16807 → 282475249 → 1622650073 → 984943658 → 1144108930...

`seed=2:` 33614 → 564950498 → 1097816499 → 1969887316 → 140734213...

`seed=3:` 50421 → 847425747 → 572982925 → 807347327 → 1284843143...

`seed=4:` 67228 → 1129900996 → 48149351 → 1792290985 → 281468426...

¡siempre!

¿y por qué 42?

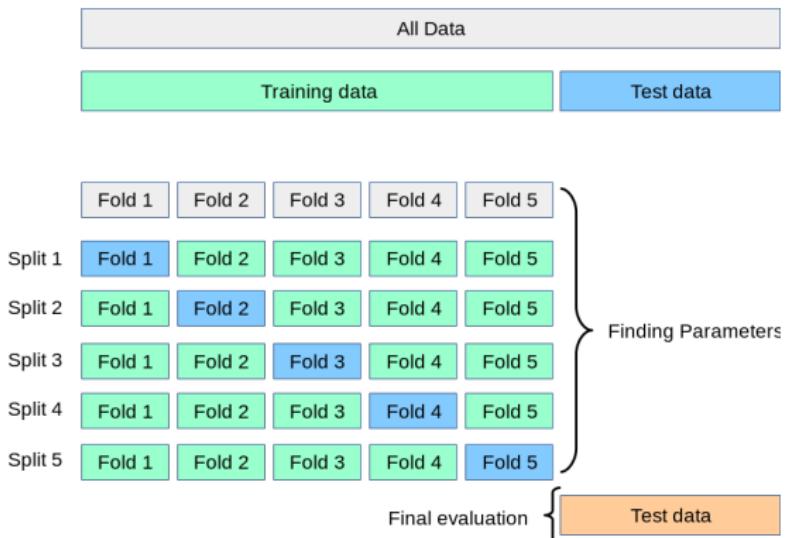
El RNG de la empresa CloudFlare: 100 lámparas de lava



(enlace)

¿Qué pasa si tenemos pocos datos?

k -fold cross-validation (aquí $k = 5$):



Más información: [Cross-validation: evaluating estimator performance](#)

(Materia avanzada: Nested cross-validation en ["Cross-validation: What does it estimate and how well does it do it?"](#))

Hay rutinas para ayudarnos en las tareas de trocear los datasets

- `from sklearn.model_selection import train_test_split`
- y calcular el resultado de cross-validation
- `from sklearn.model_selection import cross_val_score`

Peligro: “*data leakage*”

Eso es cuando nuestro modelo sabe qué datos va a ver en el “futuro”

- transformaciones: *p.ej.* escalado, mean imputation

Problema: que suele devolver resultados demasiado optimistas

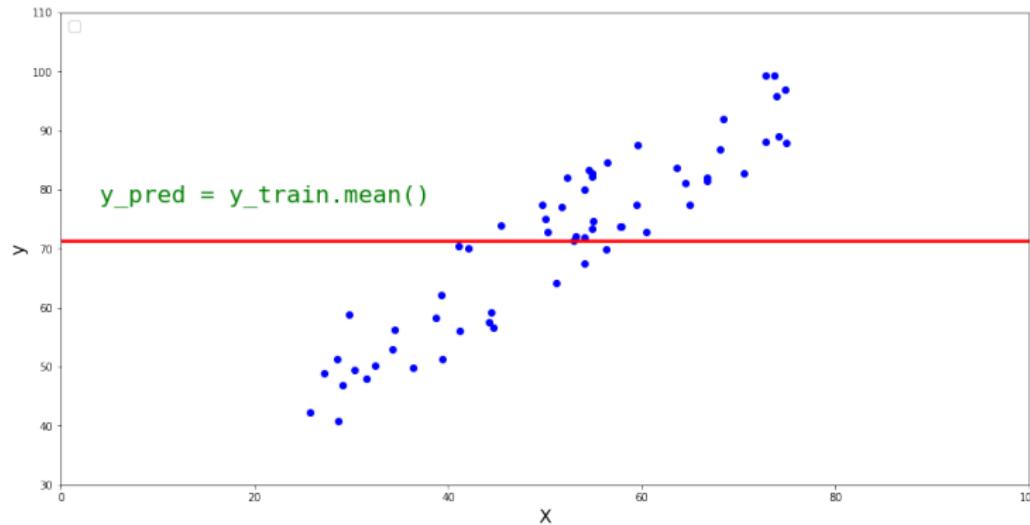
¿Como podemos evitar data leakage? Limpiando los dataframes de **X_train** y **X_test** de **forma igual, pero por separado**.

(Para ayudar hacer eso podemos hacer la limpieza a través de una función *p.ej. cleaning*)

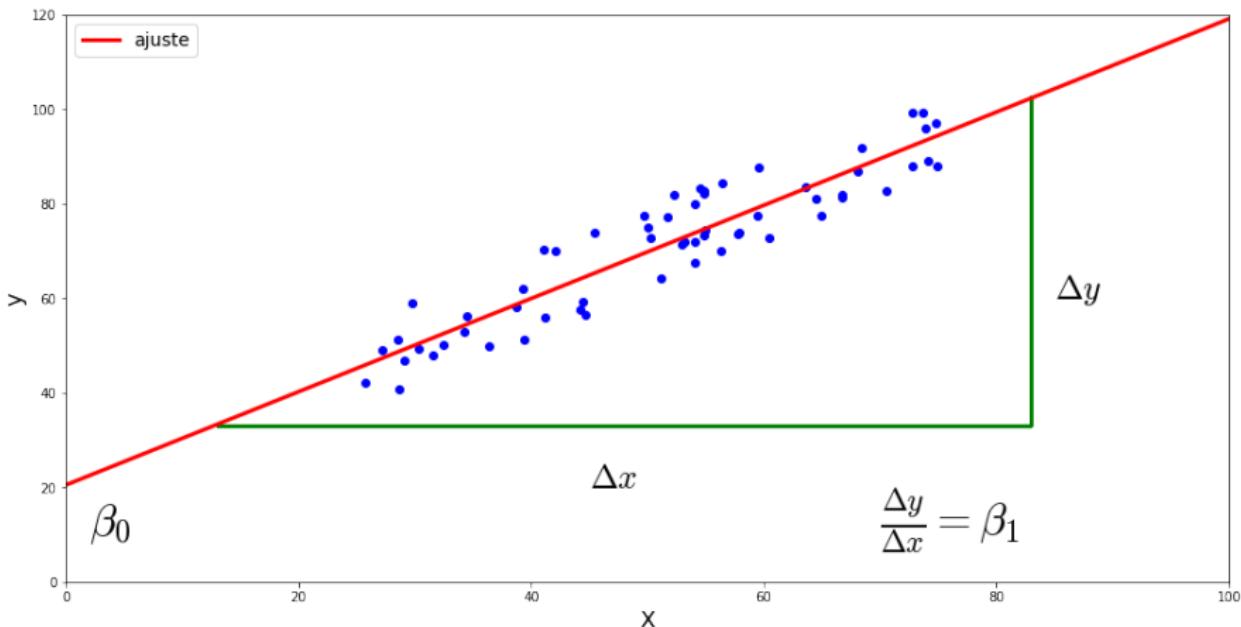
Regresión

Tu primer modelo: un '*baseline*': $\hat{y} = E[f(x)] = \bar{y} \quad \forall \hat{y}$

Los predicciones son la **esperanza** (E), i.e. la media aritmética, de los targets, y



Regresión lineal: Ejemplo en 1D (“*univariate linear regression*”):



La **pendiente** o **gradiente**: nomenclatura

una linea recta:

$$\text{gradiente} = \frac{\Delta y}{\Delta x} = \left. \frac{y(x_2) - y(x_1)}{x_2 - x_1} \right|_{x_2 > x_1}$$

una curva (la **derivada**):

$$\text{gradiente} = \frac{dy}{dx}$$

una curva multi-dimensional (la **derivada parcial**)

$$\text{gradiente} = \frac{\partial y}{\partial x}$$

Modelo de regresión lineal:

$$\hat{y}(x) = \beta_1 x + \beta_0$$

Para cada ‘feature’ x hay un parámetro ajustable, β_1 , que es su pendiente o gradiente

y también hay un término β_0 que es la intersección o ‘bias’

Eso es un modelo “paramétrico”.

Sesión práctica

Sesión práctica: Regresión lineal con python

BIG DATA: Práctica: Regresión lineal con python

Objetivo: Convertir la ecuación de mínimos cuadrados ordinarios en código python, y comprobar visualmente que funciona

(La técnica de mínimos cuadrados fue inventado por [Adrien-Marie Legendre](#) en 1805)

Se puede tratar nuestro DataFrame como si fuera una sistema de ecuaciones lineales donde los valores x_{mn} representan los coeficientes:

$$\begin{cases} \beta_1 \cdot x_{0,1} + \beta_0 \cdot 1 = y_0 \\ \beta_1 \cdot x_{1,1} + \beta_0 \cdot 1 = y_1 \\ \beta_1 \cdot x_{2,1} + \beta_0 \cdot 1 = y_2 \end{cases}$$

donde queremos hallar β_0 y la β_1 asociada con cada columna.

Escrito como matrices:

$$\mathbf{X} = \begin{bmatrix} 1 & x_0 \\ 1 & x_1 \\ \vdots & \vdots \\ 1 & x_m \end{bmatrix}, \quad \boldsymbol{\beta} = \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_m \end{bmatrix}$$

(\mathbf{X} se conoce como el '*design matrix*')

Regresión múltiple (con n features):

$$\mathbf{X} = \begin{bmatrix} 1 & x_{01} & x_{02} & \cdots & x_{0n} \\ 1 & x_{11} & x_{12} & \cdots & x_{1n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{m1} & x_{m2} & \cdots & x_{mn} \end{bmatrix}, \quad \beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_n \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_m \end{bmatrix}$$

y

$$\mathbf{X}\beta = \mathbf{y}$$

y en python (usando `numpy.dot`)

```
y = np.dot(X,beta)
```

Un solver de sistemas de ecuaciones lineales, procedente del [álgebra lineal](#), es la “*Normal equation*”.

Eso es una solución analítica en forma ‘cerrada’ para la ecuación matricial $\mathbf{X}\beta = \mathbf{y}$:

$$\beta = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$$

donde $(\mathbf{X}^T \mathbf{X})$ se llama la “*normal matrix*”.

En código python:

```
X = np.array([x, np.ones(len(x))]).T  
b1, b0 = np.linalg.inv(X.T.dot(X)).dot(X.T).dot(y)
```

(Hay un ejemplo en python en Anexo II en nuestro notebook: [BIG DATA: Práctica: Regresión lineal con python](#))

Regresión lineal: (univariate) regresión polinómica

No siempre (es decir, casi nunca!) los relaciones son puramente lineales.
Podemos añadir features de potencia: *p.ej.*

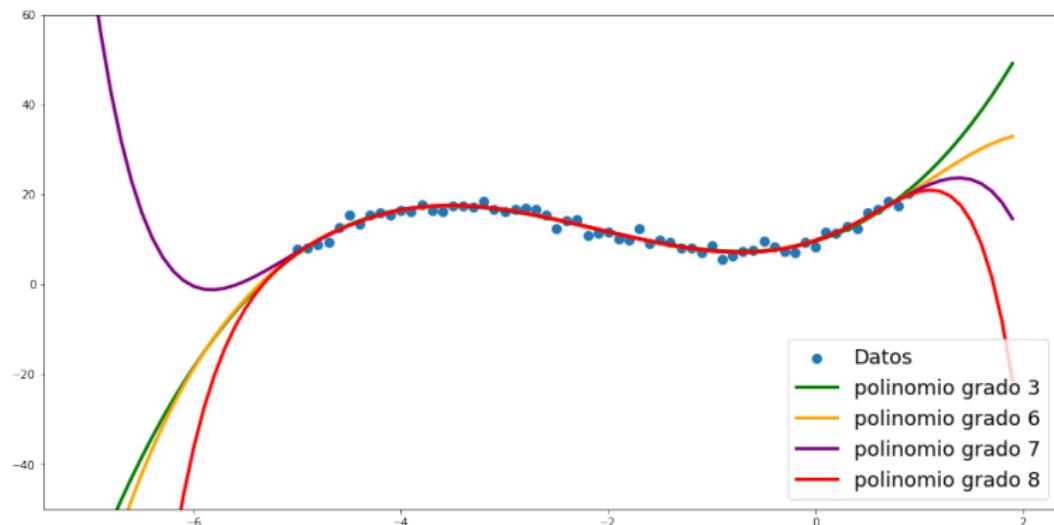
$$\hat{y}(\mathbf{X}) = \beta_2 \mathbf{X}^2 + \beta_1 \mathbf{X} + \beta_0$$

y con esta nueva feature ya introducimos algo de curvatura a nuestro modelo

(Nota: Sigue siendo una función lineal en términos del parámetro β)

Extrapolación

¡Ojo! Las regresiones polinómicas de grandes potencias tienen colas muy inestables:



Explicabilidad (*explainability*):

Los β , es decir las pendientes, dan valiosa información sobre la importancia (o peso) de cada feature.

P.ej. si un feature tiene $\beta = 0$, básicamente este feature no forma parte del modelo, y no tiene ninguna influencia a la hora de predecir \hat{y}

Competición

Según la primera tabla en la plantilla:
¿cuáles son las features más importantes en nuestro modelo lineal?

Weight?	Feature
+390135.335	m2_edificados
+378177.833	<BIAS>
+355886.998	precio_zona
+63058.702	exterior
+16381.696	orientación_norte
+4314.874	planta
+3809.496	año_construcción
+1.025	precio_aparcamiento
-595.693	necesita_reforma
-4027.583	orientación_sur
-4077.580	n_baños
-6322.717	n_habitaciones
-7899.945	m2_útiles
-14058.596	orientación_este
-14204.718	orientación_oeste
-24590.137	aparcamiento
-26983.654	nueva_construcción
-80908.585	ascensor
-118913.261	precio_alquiler

(Nota: aquí <BIAS> es β_0)

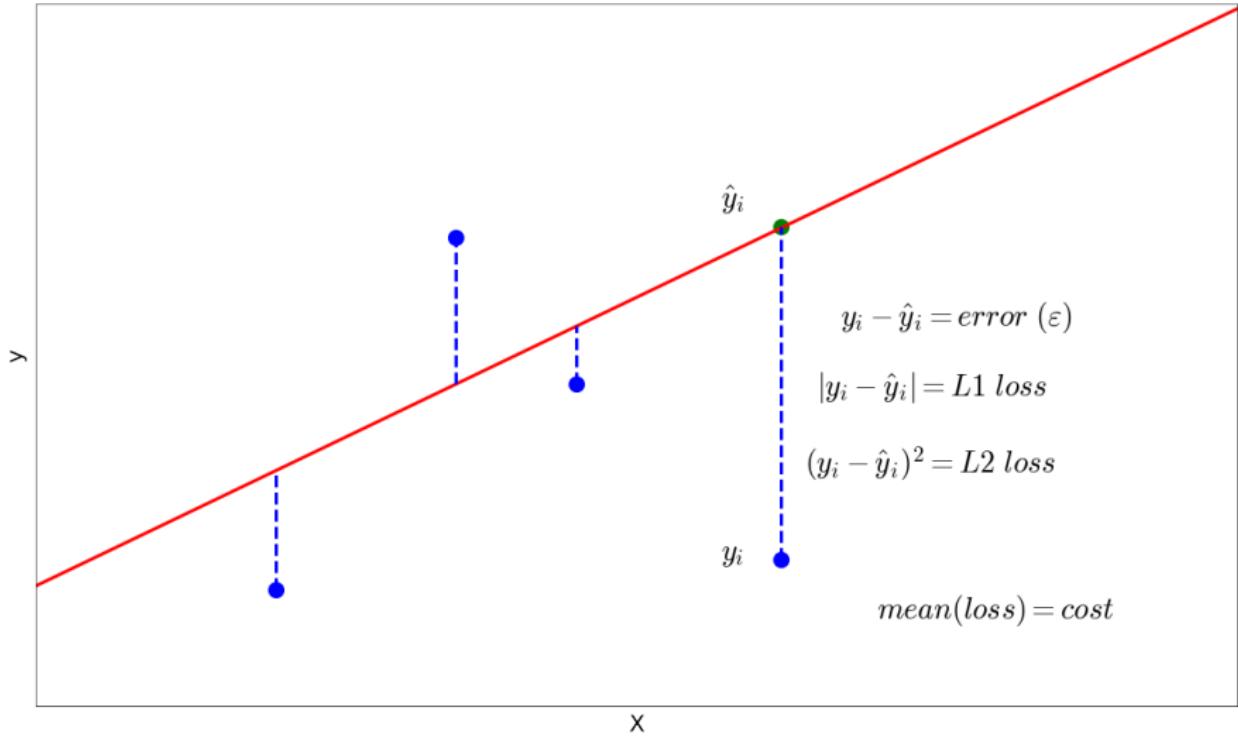
Loss y cost

¿Cómo encuentra la solución óptima un algoritmo?:
Los funciones de *loss* (\mathcal{L}) y *cost* (J)

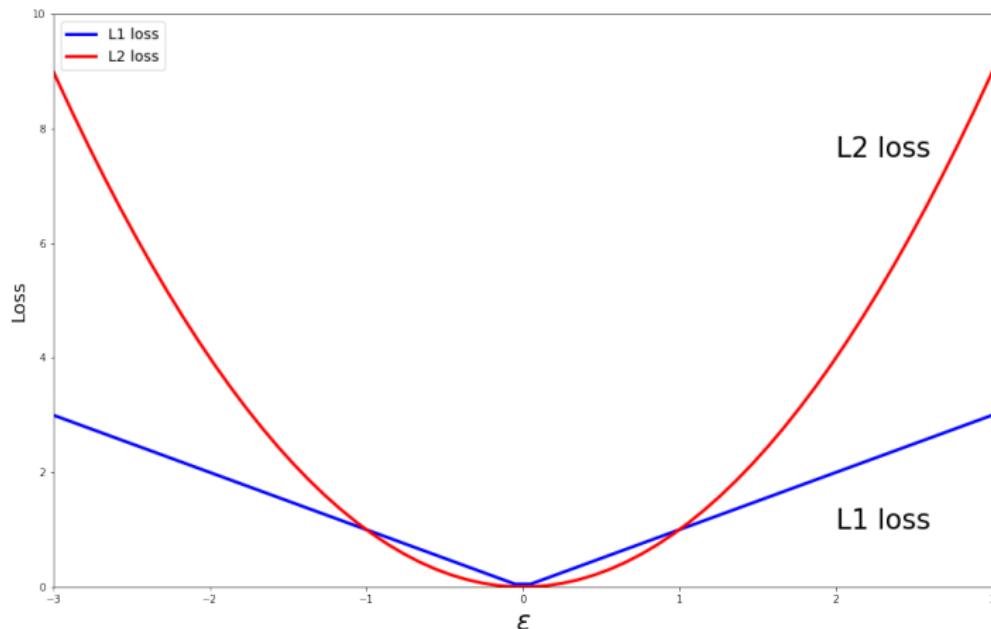
- *loss* es una medida de distancia* entre dos puntos \hat{y} e y
- *cost* es la media de todos estos puntos de *loss*: $\bar{\mathcal{L}}$

(a veces la función de loss es conocido como la *error function*, es decir ¿en cuánto nos hemos equivocado?
y la función de cost es conocido como la *objective function*)

*(Materia (muy) avanzada: [normas: \$p\$ -norm](#) y [espacios de Lebesgue](#))



¿Que pinta tienen las funciones de loss?



- L1 loss no es muy sensible a los outliers
- L2 loss sí es muy sensible a los outliers debido a las diferencias al cuadrado

En código:

$y \rightarrow \text{y_true}$

$\hat{y} \rightarrow \text{y_pred}$

$(\text{y_true} - \text{y_pred})$ se conoce como el error (ε) o el “residuo”.

- L1 loss = AE (absolute error) = `abs(y_true - y_pred)`
- L1 cost = MAE (mean absolute error) = $\frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|$
- L2 loss = SE (squared error) = `(y_true - y_pred)**2`
- L2 cost = MSE (mean squared error) = $\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$

La función de cost (o ‘*objective function*’) J es lo que queremos minimizar

Por ejemplo, en regresión lineal usamos **mínimos cuadrados ordinarios**, es decir, usamos el L2 loss y el L2 cost:

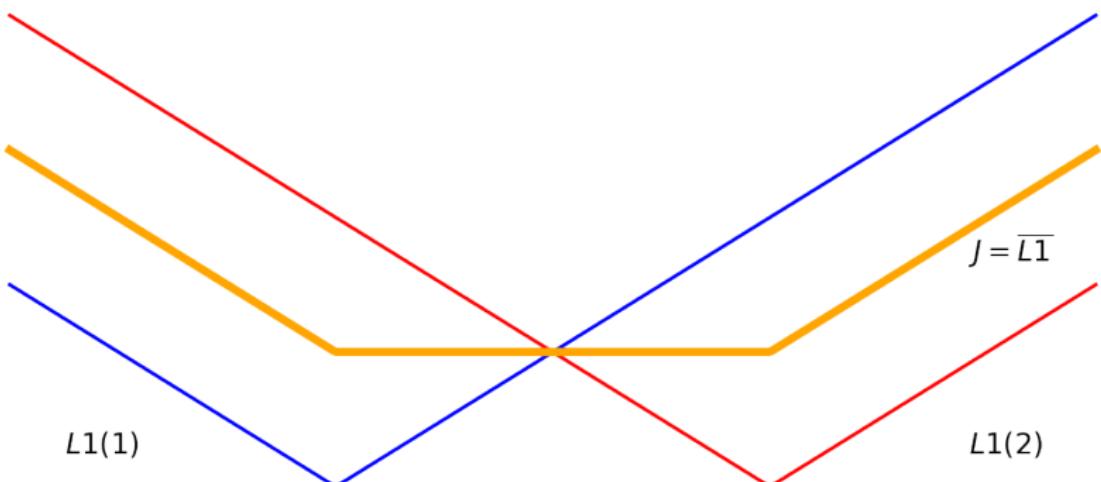
$$J(\beta) = MSE$$

$$= \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

$$J(\beta_1, \beta_0) = \frac{1}{n} \sum_{i=1}^n (y_i - \beta_1 \mathbf{X}_i + \beta_0)^2$$

Los ‘mejores’ valores de β_1 y β_0 son los dados por $\text{argmin}(J)$

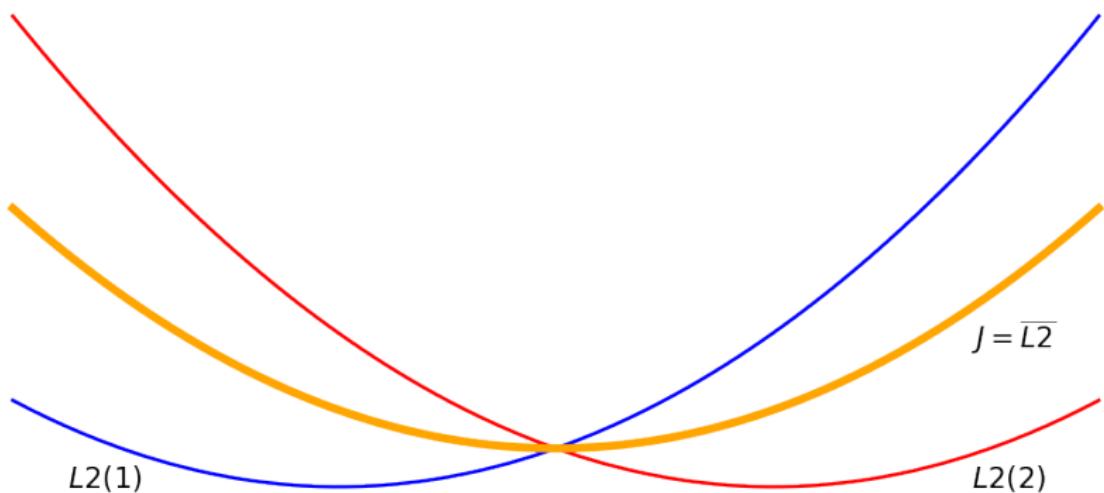
¿Que forma tiene el cost (J) para el L1 loss para dos puntos?



- J es una función definida a trozos
- puede tener un minimo planar, y una segunda derivada de cero

(The mean absolute error is a new member of the collection of objectives in XGBoost (Octubre 2022))

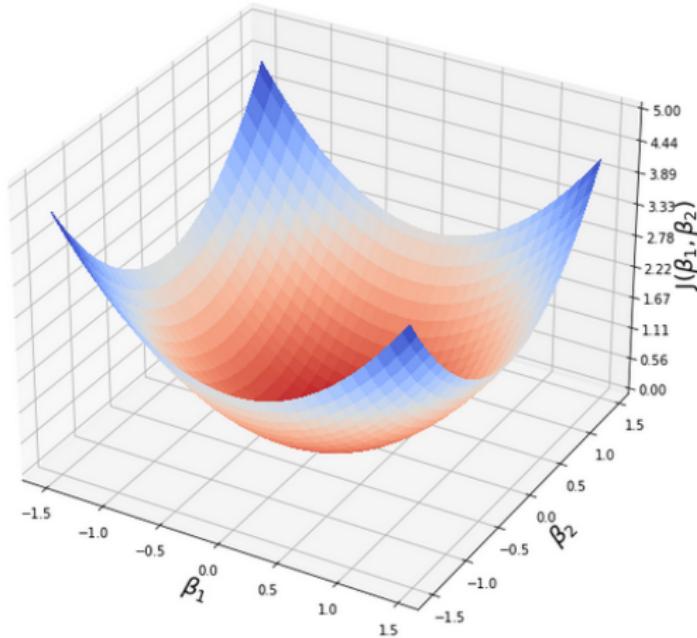
¿Que forma tiene el cost (J) para el L2 loss para dos puntos?



- J es una función suave (infinitamente diferenciable)
- es una función convexa → tiene un único mínimo global

¿Que forma tiene el cost (J) para el L2 loss en 2D?

J ya es una superficie parabólica con la misma dimensión que el número de parámetros, por ejemplo algo así para dos features:



Solver: Gradient descent

El gradiente (G) de una superficie está dado por

$$G = \frac{\partial J(\beta)}{\partial \beta}$$

donde β son los parámetros

Gradient descent '*update rule*'

$$\beta \leftarrow \beta - \alpha \frac{\partial J(\beta)}{\partial \beta}$$

donde α es el parámetro "*learning rate*"

Queremos hallar el mínimo local, o mejor, el mínimo global

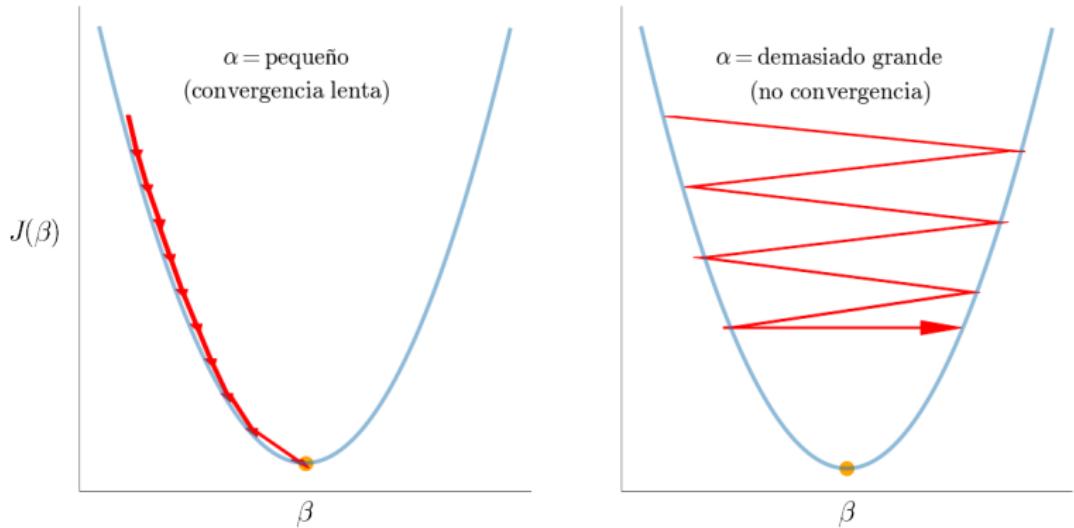
(Artículo: "An overview of gradient descent optimization algorithms")

algoritmo:

primero elegir un valor aleatorio inicial para β

- ➊ calcular el gradiente ($\partial J / \partial \beta$)
- ➋ multiplicar por -1 para cambiar el signo
- ➌ escalar por α
- ➍ eso ya es el nuevo valor de β
- ➎ repetir, y cuando β apenas cambia, has terminado

Efecto del *learning rate*:

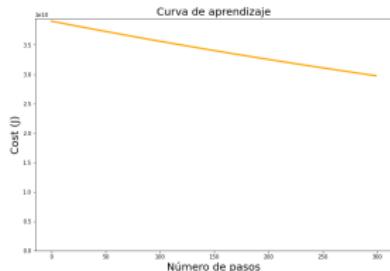


Ajustar a α dibujando cada paso de la función de coste.

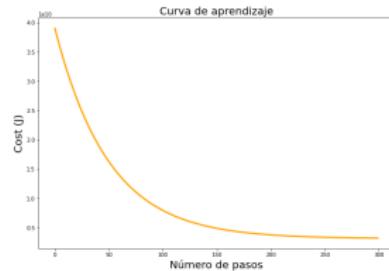
Early stopping: p.ej. si $\Delta J < 0.001$

Si el *cost* sube, hay un problema y hay que usar un α más pequeño.

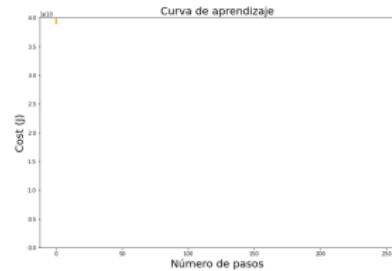
α demasiado pequeño



buen α



α demasiado grande



- Adagrad - Adaptive Gradient Algorithm: cambia α sobre la marcha

Gradient descent para regresión lineal:

$$\beta \leftarrow \beta - \alpha \frac{\partial J(\beta)}{\partial \beta}$$

donde

$$\begin{aligned} J(\beta) &= \text{MSE} = \frac{1}{n} (\varepsilon^T \varepsilon) = \frac{1}{n} (\hat{\mathbf{y}} - \mathbf{y})^T (\hat{\mathbf{y}} - \mathbf{y}) \\ &= \frac{1}{n} (\beta \mathbf{X} - \mathbf{y})^T (\beta \mathbf{X} - \mathbf{y}) \end{aligned}$$

y tenemos (la demostración se deja para el lector interesado!)

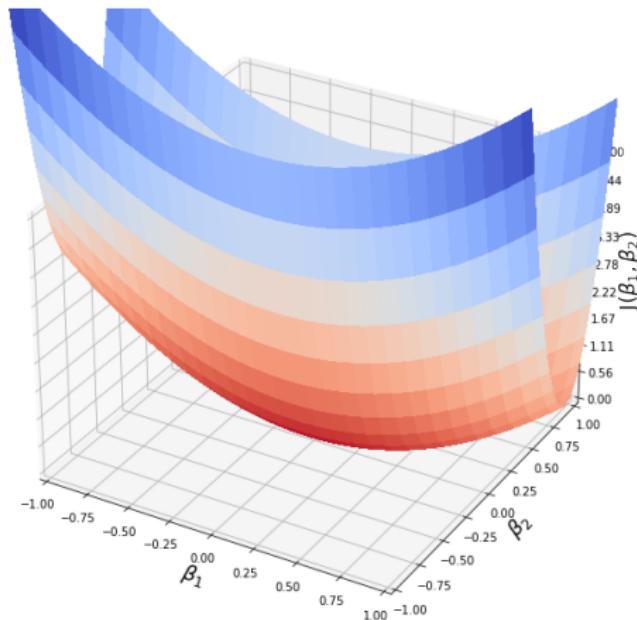
$$\frac{\partial J(\beta)}{\partial \beta} = \frac{1}{n} ((\beta \mathbf{X} - \mathbf{y}) \mathbf{X})$$

dando

$$\beta \leftarrow \beta - \frac{\alpha}{n} (\varepsilon \mathbf{X})$$

(Hay un ejemplo en python en Anexo III en nuestro notebook: [BIG DATA: Práctica: Regresión lineal con python](#))

¿Que pasa si un feature tiene una escala muy distinta?



Escalado y normalización (*feature scaling o normalization*)

Con los estimadores paramétricos los solvers tienen mejor estabilidad numérica si los datos están “*normalized*”

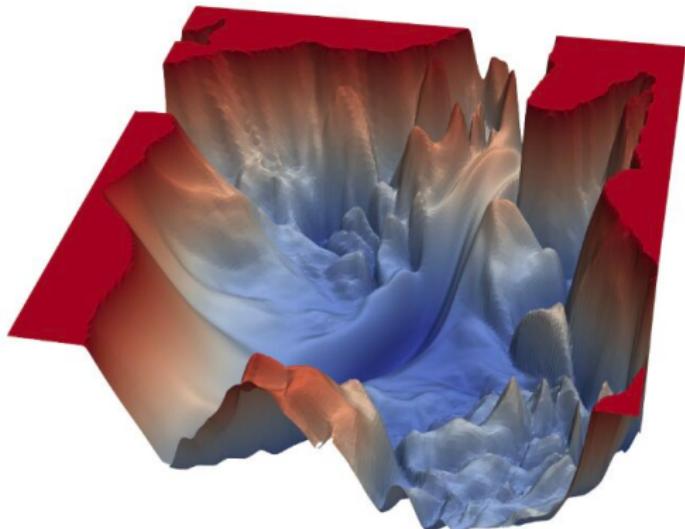
Los siguientes son transformaciones lineales:

- `sklearn.preprocessing.MinMaxScaler` [0, 1]
- `sklearn.preprocessing.StandardScaler` $\rightarrow \mu = 0, \sigma = 1$
- `sklearn.preprocessing.RobustScaler` por si hay outliers

Ejemplo de uso:

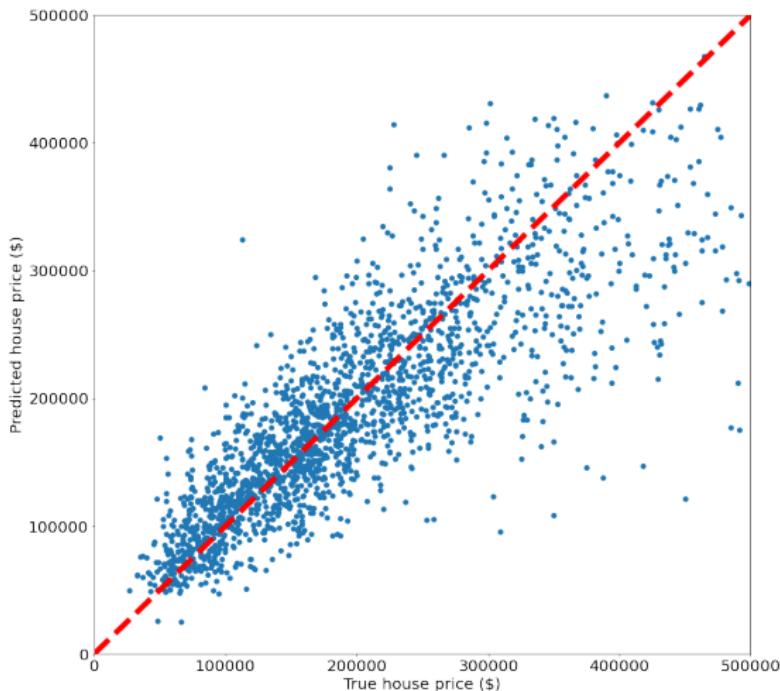
```
from sklearn.preprocessing import RobustScaler  
RS = RobustScaler()  
train_data = RS.fit_transform(train_data)  
test_data = RS.transform(test_data)
```

Ejemplo de una superficie de loss de un red neuronal



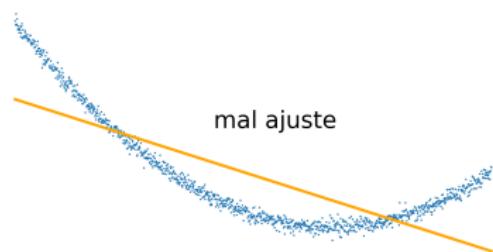
(Fuente: "["Visualizing the Loss Landscape of Neural Nets"](#)")

¿Qué tal lo estamos haciendo?: ‘True’ vs ‘predicted’ plot

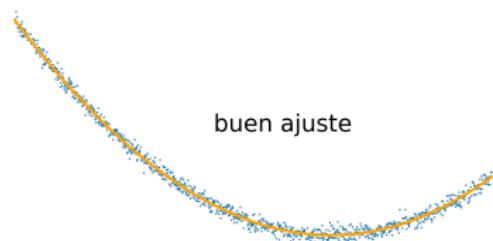


(`sklearn.metrics.PredictionErrorDisplay`)

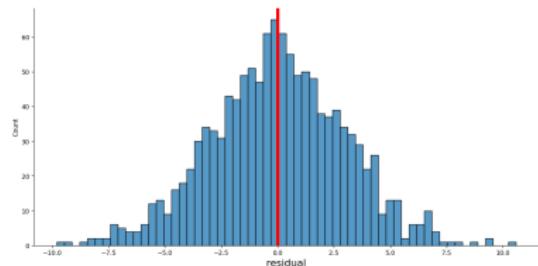
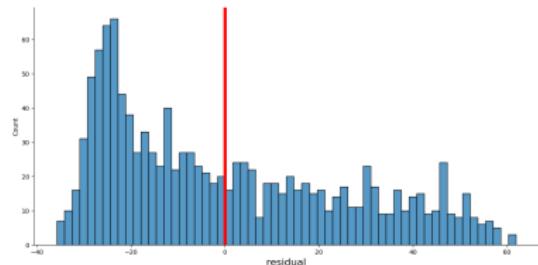
Los residuales de un buen ajuste tendrán una distribución simétrica:



mal ajuste



buen ajuste



Métricas de regresión

La métrica más natural si usamos el L2 loss es el *root mean squared error* (RMSE)

$$\text{RMSE} = \sqrt{\bar{\varepsilon}^2} = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2}$$

`sklearn.metrics.mean_squared_error`

Nota: Métrica de ML \neq KPI de negocio

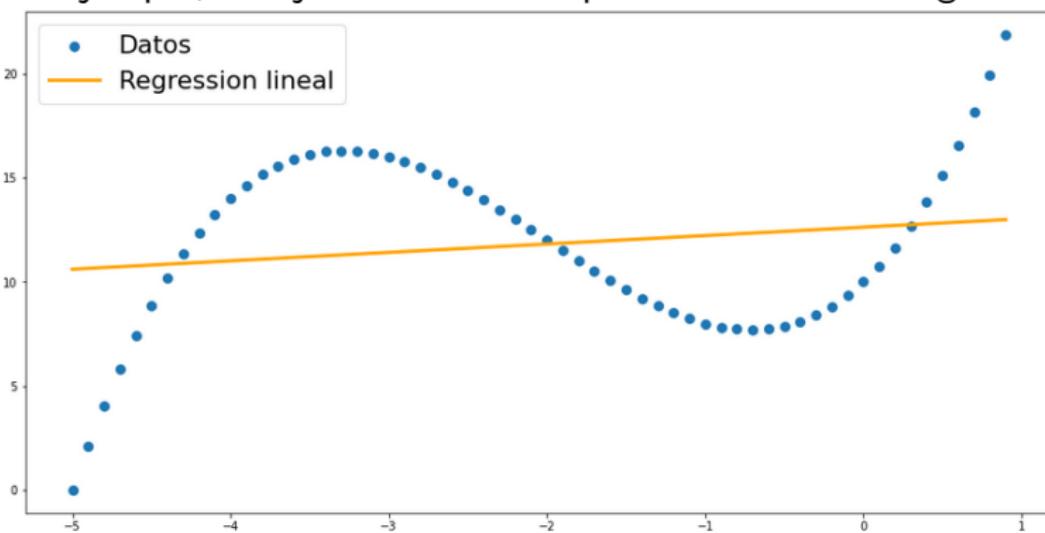
Competición

¿Cual es el RMSE score de nuestro notebook del concurso?

Regresión lineal:

- ventajas: muy fácil de interpretar
- desventajas: solo va bien si las relaciones entre X e y son lineales

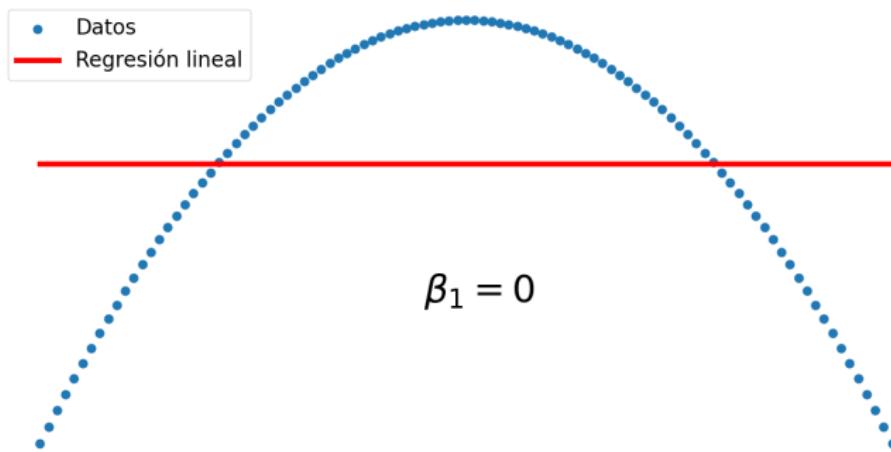
Por ejemplo, un ajuste lineal a un polinomio del tercer grado:



¿Qué ha pasado?: Hemos ajustado la mejor linea recta posible a los datos, pero la regresión lineal no es capaz de capturar lo que está realmente pasando; falta la “complejidad” suficiente (eso se llama “*high bias*”).

Los cuatro supuestos básicos para una ‘buena’ regresión lineal

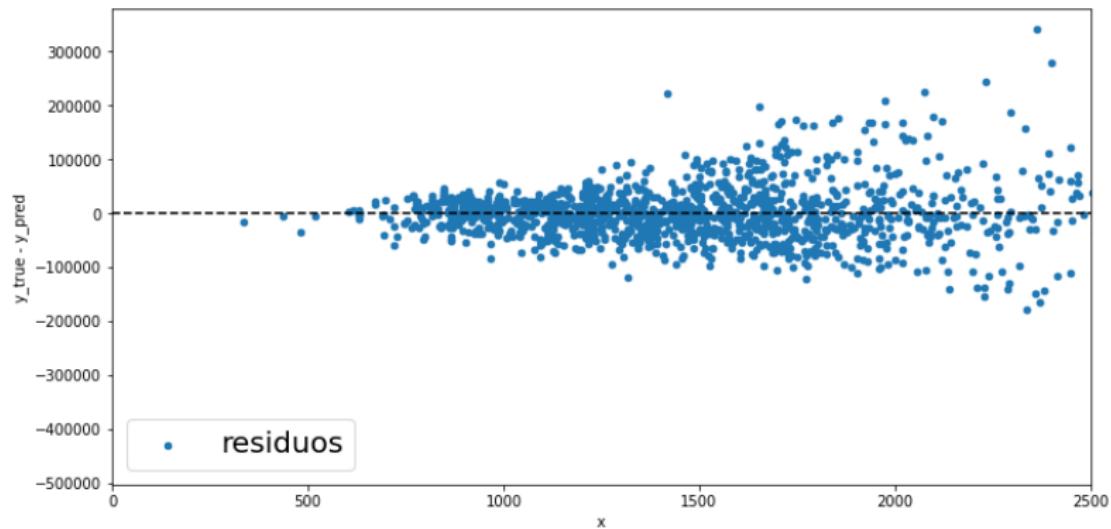
Basta decir que la regresión lineal va mejor cuando existe una relación lineal entre la variable independiente, x , y la variable dependiente, y



Ademas, los residuos (los errores) deben que ser:

- i.i.d
- **distribuidos de forma normal** ($\forall i \in n, \varepsilon_i \sim \mathcal{N}(0, \sigma^2)$)
- **hay homoscedasticidad:** tienen una varianza constante ($\sigma^2 \neq f(x)$)

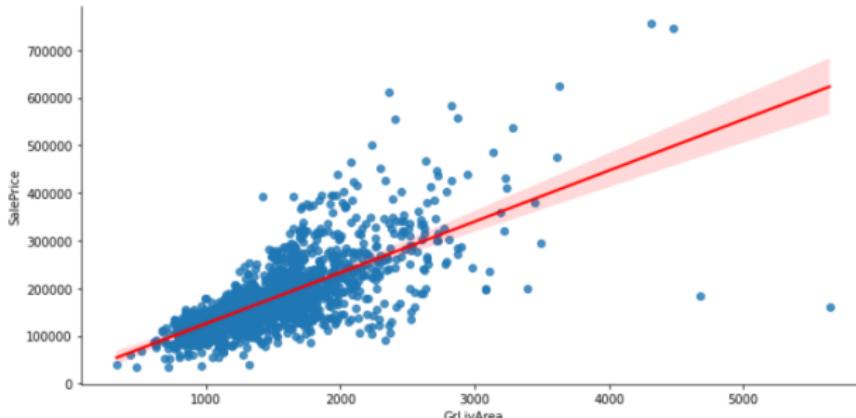
Ejemplo de heterocedasticidad



Possible solución: aplicar una transformación, por ejemplo tomar el logaritmo o raíz cuadrada de la variable dependiente (y)

(Nota: Que no es una buena idea con más de una variable independiente)

Para ayudar con EDA podemos usar `seaborn.lmplot`



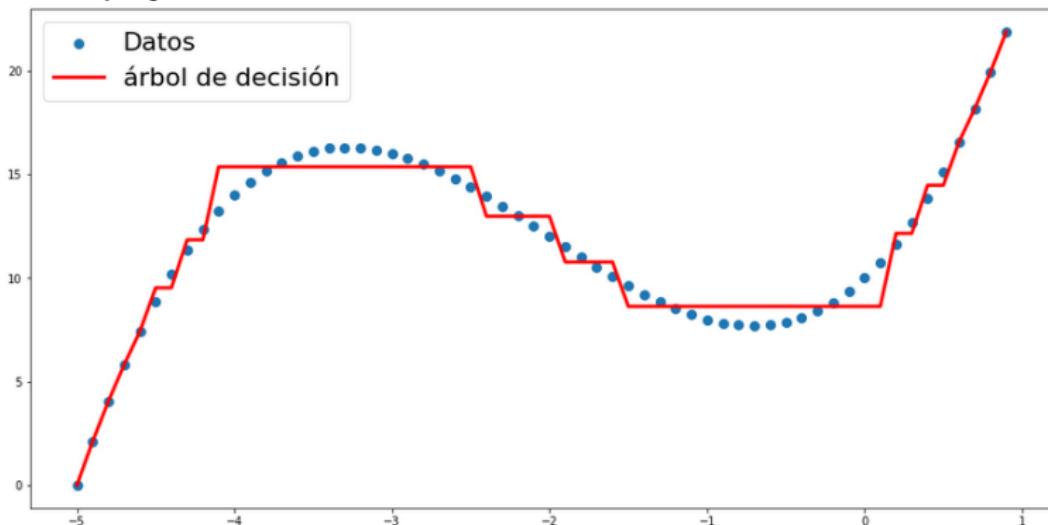
que puede dibujar los `intervalos de confianza` de los parametros

```
sns.lmplot(x=X, y=y, data=data, ci=95)
```

(Más material sobre regresión lineal: "[The Truth about Linear Regression](#)")

Decision Trees

Más complejidad: Árboles de decisión ([1963](#))

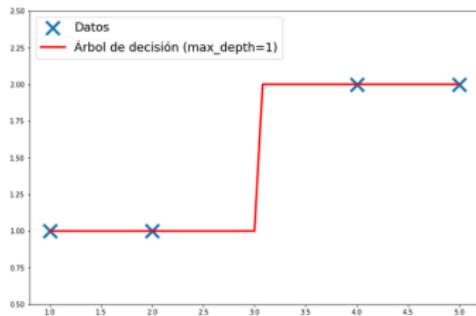


```
from sklearn.tree import DecisionTreeRegressor
```

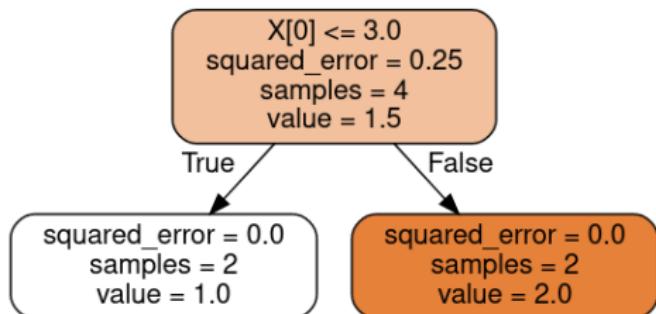
Los árboles usan “*recursive binary splitting*”, y paran de dividir cuando llegan a un numero mínimo de datos en cada ‘hoja’ (región de espacio)

Ejemplo sencillo:

Los datos



el árbol (el ajuste)

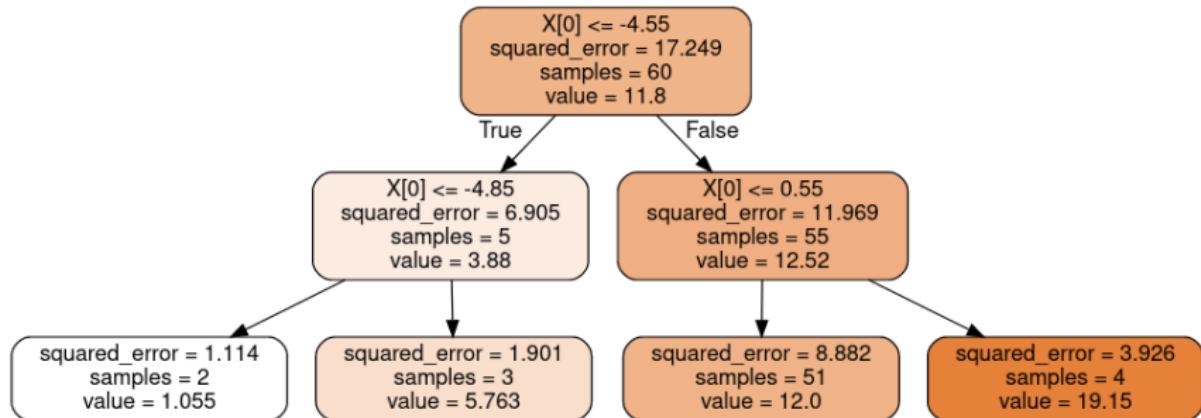


el modelo en palabras:

- Si x es menor que 3, \hat{y} es 1
- Si x es mayor que 3, \hat{y} es 2

(Nota: Un decision tree de `max_depth=1` tb. se conoce como un '[decision stump](#)')

Las hojas del árbol contienen el valor medio del “*split*”
Los splits se calculan de manera que en cada uno se minimiza el ‘*residual*’ global donde el residual es el L2 (*i.e.* error²)



Los estimadores “no-paramétricos” basados en árboles de decisión no usan parámetros como los β que usa regresión lineal, ni gradient descent, y por tanto los features ya **no necesitan ni escalado ni normalización**

...pero sí tienen 'hiper'-parámetros

Regresión lineal tiene cero hiper-parámetros:

```
regressor = LinearRegression()
```

pero el árbol de decisión ya tiene dos hiper-parámetros:

```
regressor = DecisionTreeRegressor(max_depth=5,  
                                   min_samples_leaf=1)
```

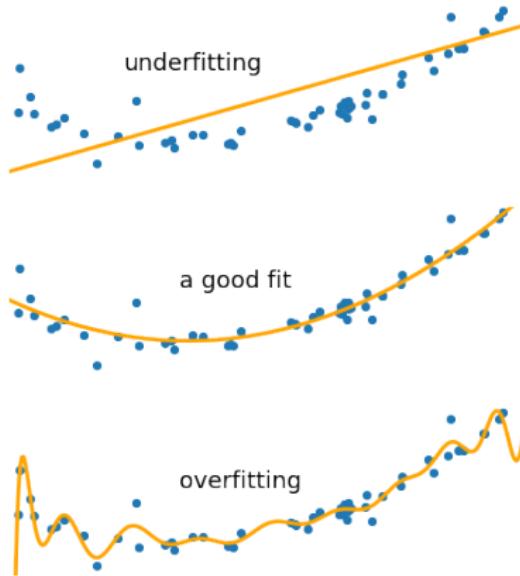
Sesión práctica

Sesión práctica: Regresión con [scikit-learn](#)

- Jupyter notebook: [BIG DATA: Práctica: Regresión con sklearn](#)

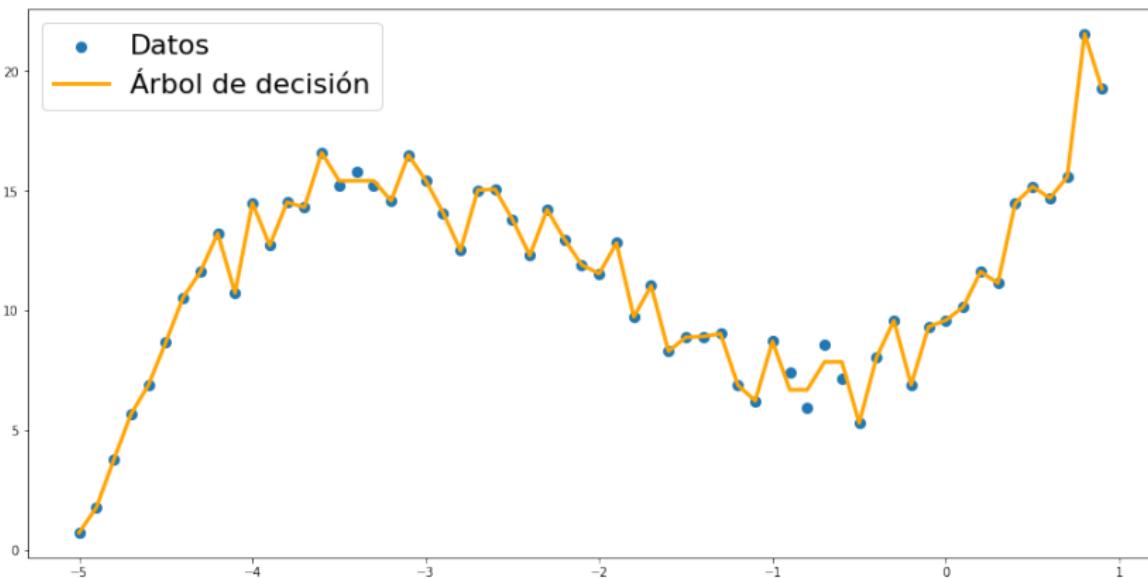
Objetivo: Variar el parametro `max_depth` y dibujar una '*curva de aprendizaje*'

Sobreajuste (*overfitting*): regresión:

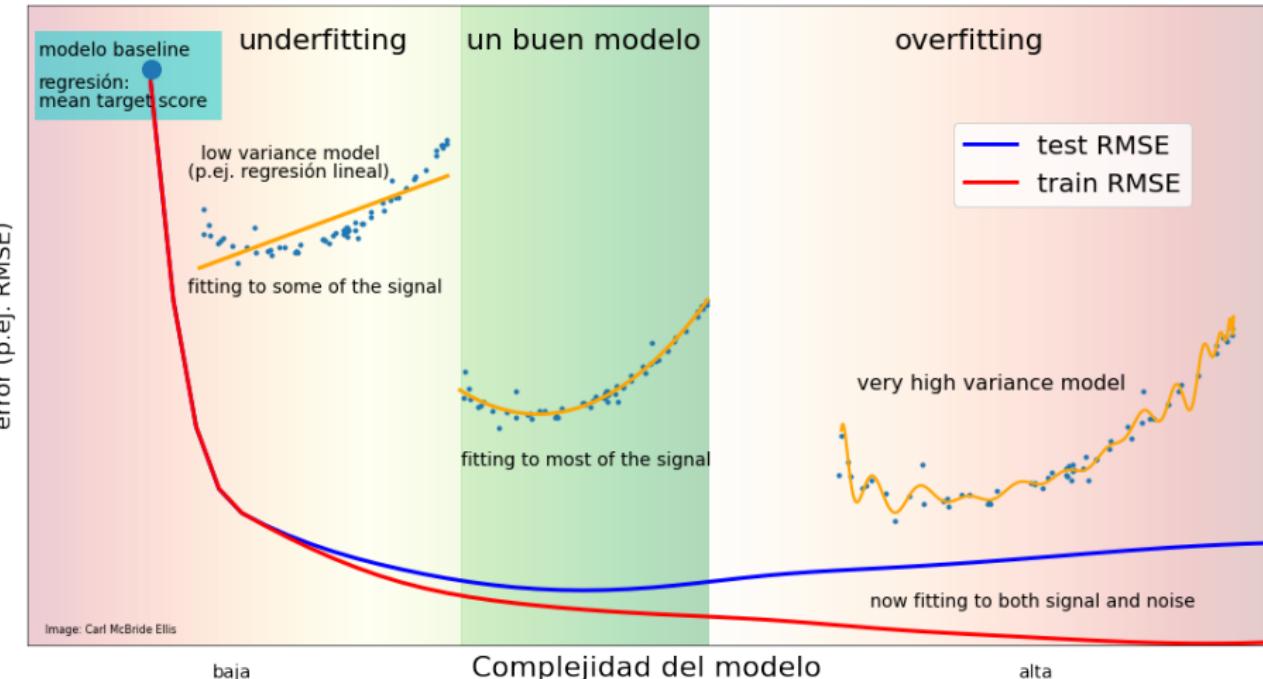


Eso pasa cuando nuestro modelo tiene demasiada complejidad.

por ejemplo, si en nuestro Notebook BIG DATA: Práctica: Regresión con `sklearn` ponemos `max_depth=10`:



La curva de aprendizaje para regresión



(Materia avanzada: Artículo: “Reconciling modern machine-learning practice and the classical bias–variance trade-off”)

¡Ojo! Lo más que interrogamos a nuestro dataset de validación (`X_val`, `y_val`) y volvemos a cambiar nuestro modelo, lo más que, indirectamente, poco a poco estos datos van formando parte de nuestro modelo.

Es por esta razón que hemos guardado otro juego de datos para una única comprobación final (`X_test`, `y_test`)

Un buen modelo, es decir un modelo ‘robusto’, es un modelo cuyo validation score y/o CV score es bastante parecido a su score con los datos de test. (Artículo: “[Model Evaluation, Model Selection, and Algorithm Selection in Machine Learning](#)”)

Competición

¿...y si cambiamos el Linear Regression por el Decision Tree Regressor en nuestro notebook del concurso?

Por ejemplo:

```
from sklearn.linear_model import LinearRegression  
regressor = LinearRegression()
```

por

```
from sklearn.tree import DecisionTreeRegressor  
regressor = DecisionTreeRegressor(max_depth=?)
```

Overfitting

Cómo reducir overfitting en modelos paramétricos:
shrinkage (encogimiento) via regularización

Hemos visto que una regresión lineal suele under-fit una sola **feature**, pero si tenemos demasiados features, sí podemos overfit el **dataset**.

Para quitar features de muy poca importancia (básicamente ruido) podemos usar **regularización**.

(Recuerda, si un $\beta = 0$ este feature no forma parte del modelo)

Regularización = cost function (J) + penalización

- la penalización de L1 (**LASSO**) es $|\beta|$
- i.e. $J \leftarrow J + \lambda \sum |\beta|$ $(\equiv \lambda \|\beta\|_1, \text{ la norma Manhattan})$
- la penalización de L2 (**Ridge**) es β^2
- i.e. $J \leftarrow J + \lambda \sum \beta^2$ $(\equiv \lambda \|\beta\|_2^2, \text{ la norma euclídea})$

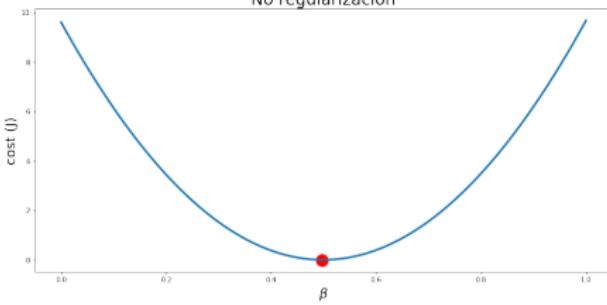
donde λ es el regularization hiper-parámetro.

En scikit-learn:

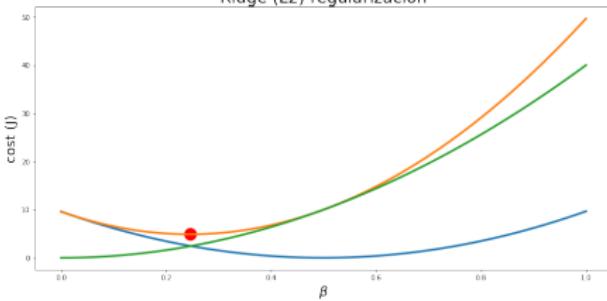
- **LASSO regression** (Linear least squares con L1 regularization)
- **Ridge regression** (Linear least squares con L2 regularization) ([review](#))
- **ElasticNet regression** (Linear least squares con L1 y L2 regularization)

(Ver tambien el notebook: [Feature importance using the LASSO](#))

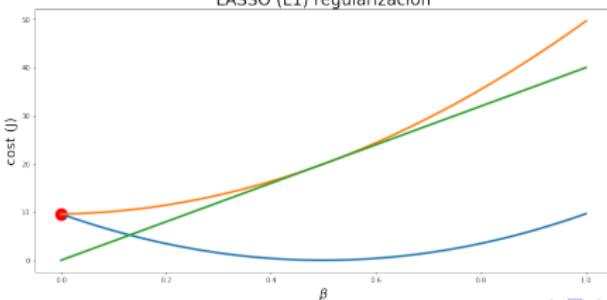
No regularización



Ridge (L2) regularización



LASSO (L1) regularización



Como reducir *overfitting* en los árboles de decisión: “*jardinería*”

Vemos que en un árbol sobre-ajustado hay solo un punto (*sample*) en cada hoja.

Podemos reducir el hiper-parámetro

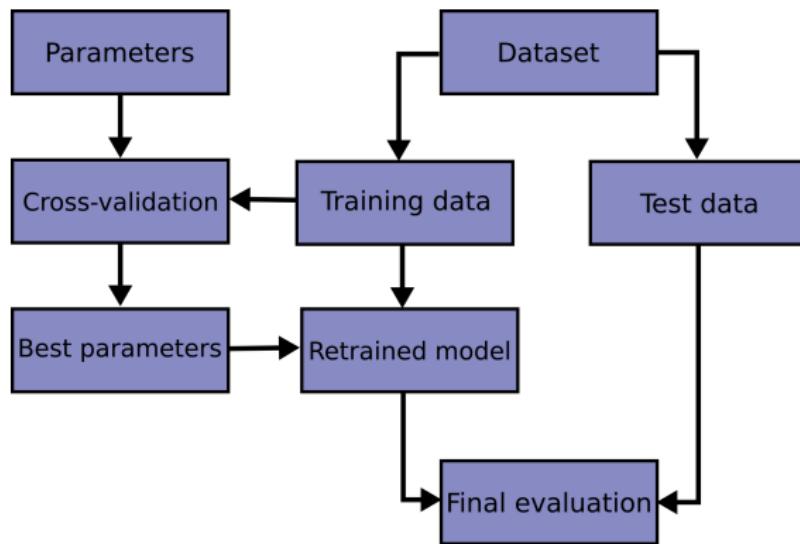
`max_depth`

y/o podemos asignar que cada hoja tenga un número mínimo de muestras con

`min_samples_leaf = 10`

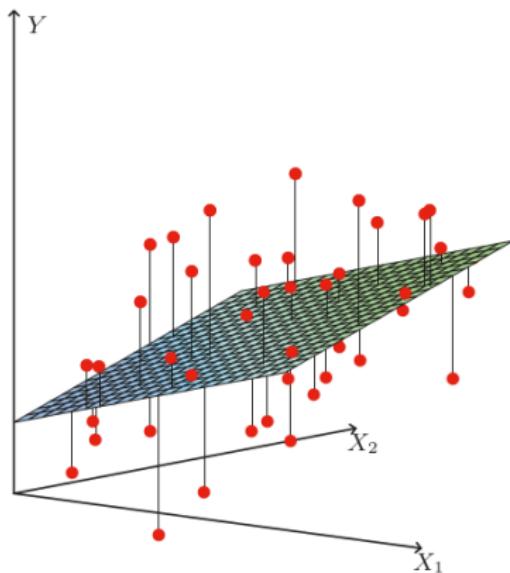
(Materia avanzada: Ver *a posteriori* “*pruning*” con el hiper-parámetro `ccp_alpha`)

Flujo de trabajo:

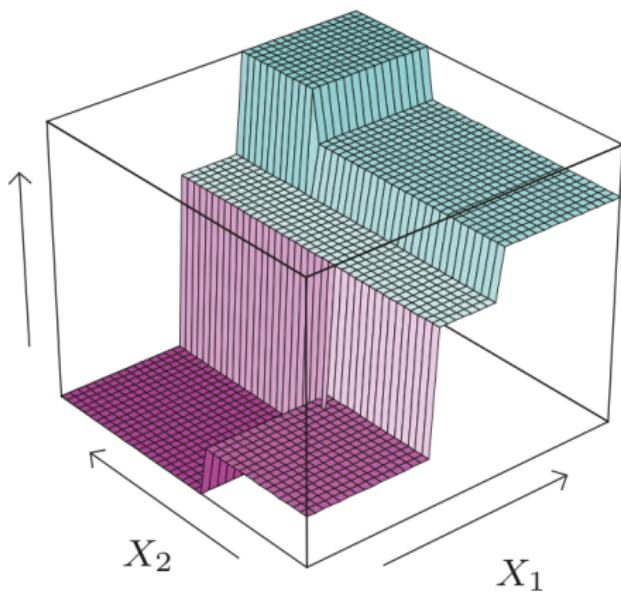


(Fuente)

Regresión lineal: Ejemplo en 2D (es decir, ya con dos 'features'):



Arboles: Ejemplo en 2D



Aquí se ve como los arboles forman trozos continuos de igual valor, un poco como “mesetas”.

Clasificación

En regresión calculamos un valor real (\hat{y}) para cada punto (cada fila en el dataframe).

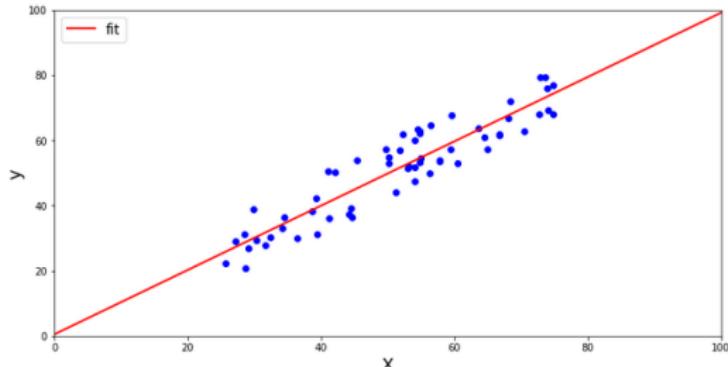
En cambio, en clasificación asignamos una etiqueta o '*label*' a cada punto. Cada label es un valor del set de clases, por ejemplo, en clasificación binaria las clases suelen ser 0 ó 1 *i.e.* $y \in \{0, 1\}$

No siempre queremos las clases, sino más bien las probabilidades *p.ej.* aunque las dos pertenecen a la clase “1”, $0.51 \neq 0.99$

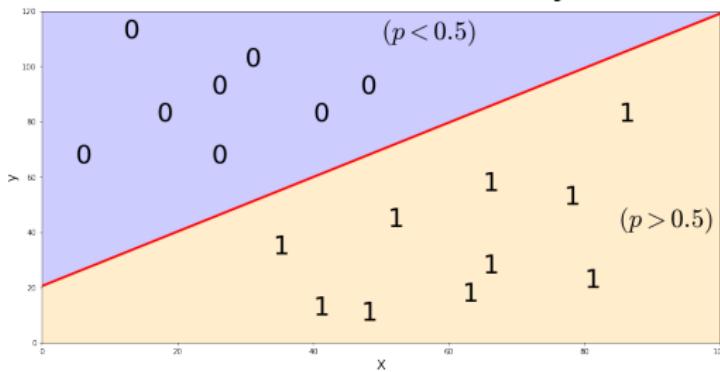
(métodos: `predict` y `predict_proba`)

(Nota: Por convenio la clase mayoritaria es 0, y se llama la '*negative class*')

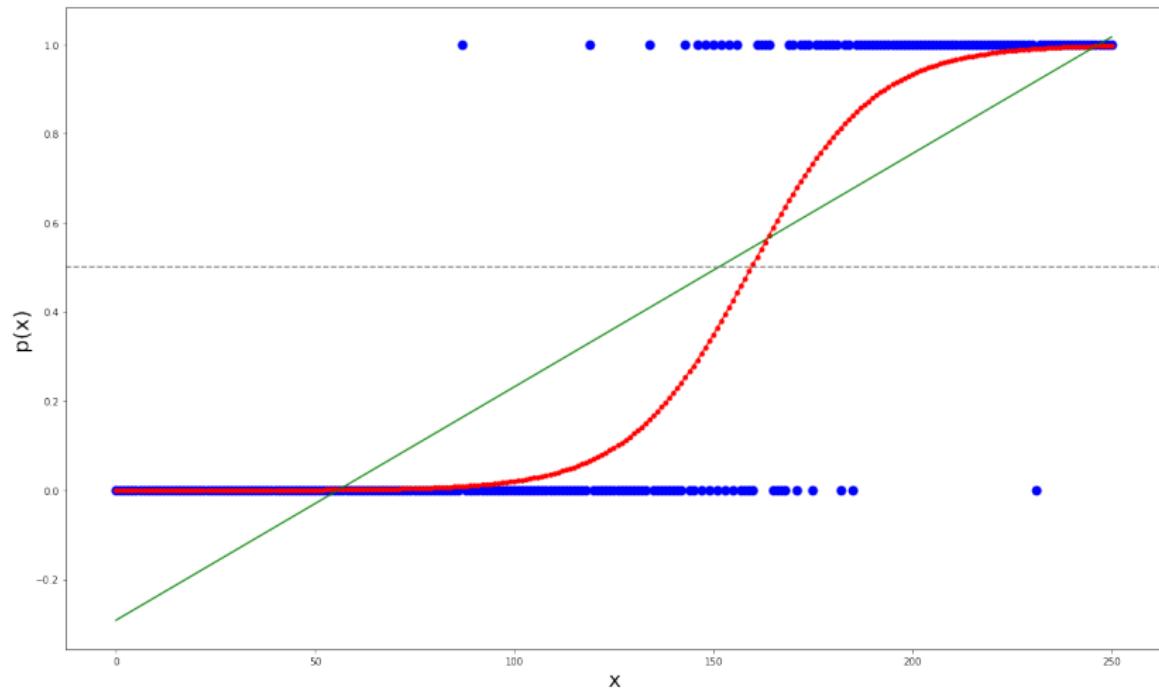
En regresión queremos hallar la línea más cercana a todos los puntos



En cambio, en clasificación queremos hallar la línea que mejor divide las dos clases, llamada la “*decision boundary*”



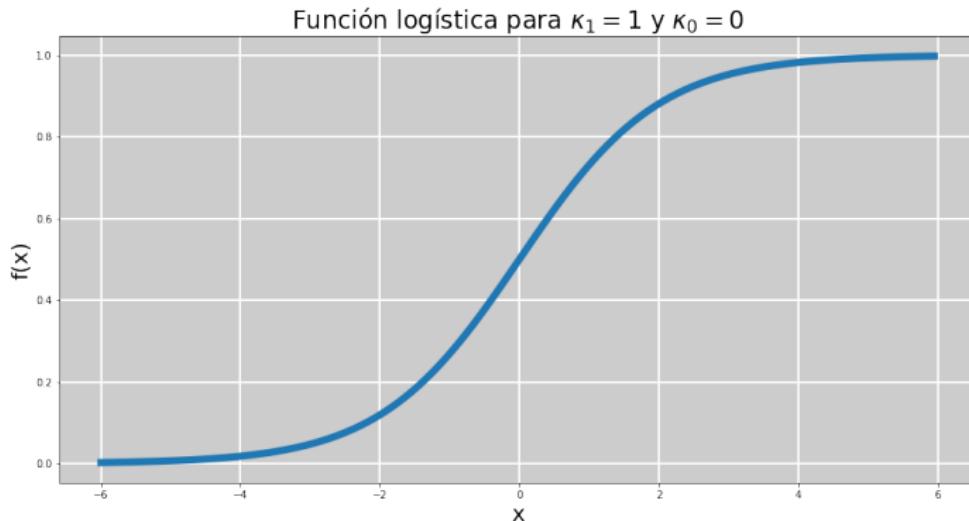
Regresión logística (*Logistic regression*): ejemplo univariante:



(fuente: el notebook [BIG DATA: Práctica: Clasificación](#))

Función logística: $\mathbb{R} \rightarrow (0, 1)$

$$\text{probabilidades}(p) \leftarrow \text{logistic}(x) := \frac{1}{1+e^{-(\kappa_1 x + \kappa_0)}}$$



Los \hat{y} que estamos calculando ya son probabilidades (p)

En regresión lineal nuestra tarea era hallar los parámetros β_1 y β_0 de la línea recta que mejor ajusta a los datos.

$$\hat{y}(x) = \beta_1 x + \beta_0$$

En regresión logística nuestra tarea ya es hallar los parámetros κ_1 y κ_0 de la función logística que mejor ajusta a los datos.

$$\ln\left(\frac{p}{1-p}\right) = \kappa_1 x + \kappa_0$$

(el *log-odds* es conocido como el *logit link function* (g))

En clasificación calculamos la probabilidad de pertenecer a un clase

Notas:

- probabilidad es multiplicativo:
- $p(d_1 \text{ y } d_2) = p(d_1) \cdot p(d_2)$
- propiedades de los logaritmos:
- $\ln(p_1 \cdot p_2) = \ln(p_1) + \ln(p_2) = \sum_i \ln(p_i)$

clasificación loss function = log loss o “cross-entropy” loss

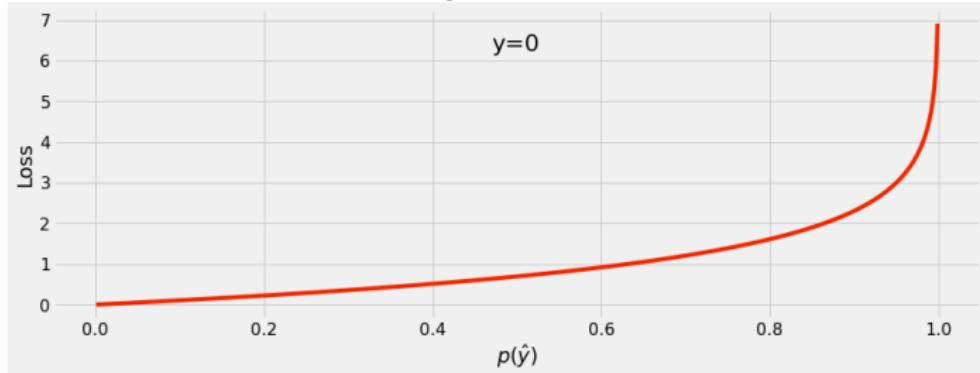
- $\mathcal{L}_{log} = -y \ln(p(\hat{y})) - (1 - y) \ln(1 - p(\hat{y}))$
- $J_{log} = -\frac{1}{N} \sum (y \ln(p(\hat{y})) + (1 - y) \ln(1 - p(\hat{y})))$

donde $p(\hat{y})$ es la probabilidad de pertenecer a la clase 1,
y por tanto $(1 - p(\hat{y}))$ es la probabilidad de pertenecer a la clase 0

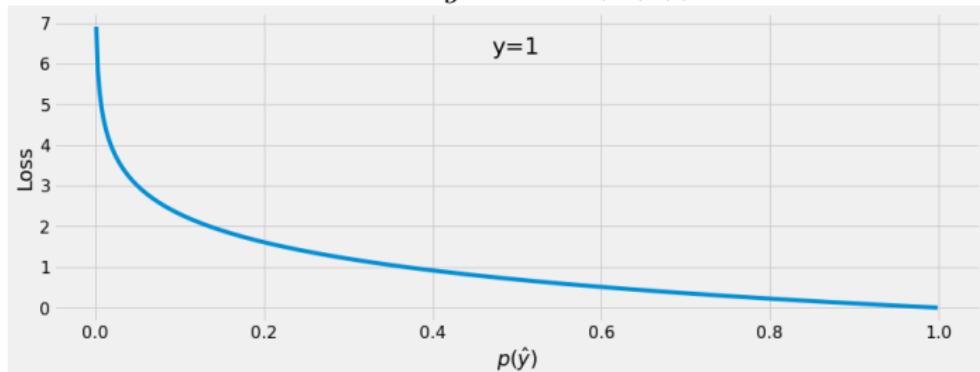
Eso nos da una función de cost convexa, así que siempre podemos hallar el mínimo global

¿Que pinta tiene el loss \mathcal{L} ?

Si la clase de y es 0 el $\mathcal{L}_{log} = -\ln(1 - p(\hat{y}))$



Si la clase de y es 1 el $\mathcal{L}_{log} = -\ln(p(\hat{y}))$



Explicabilidad (*explainability*):

Regresión logística, como con su homóloga la regresión lineal, tiene una ventaja en que sus predicciones, aunque no son necesariamente las mejores, son relativamente fáciles de entender

p.ej. ver el notebook [Titanic explainability: Why me? asks Miss Doyle](#)

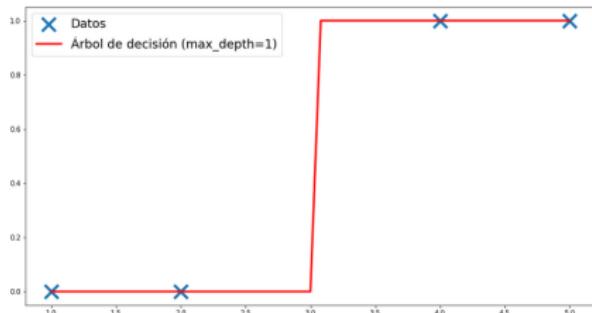
(Ley: [EU General Data Protection Regulation \(GDPR\) 2016/679](#))

```
from sklearn.linear_model import LogisticRegression
```

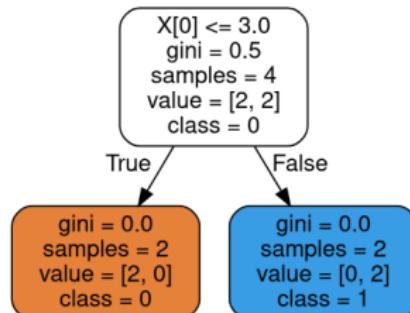
Clasificación no-paramétrica: árboles de decisión (*decision trees*)

Ejemplo sencillo:

Los datos



el árbol (el ajuste)



el modelo en palabras:

- Si x es menor que 3, la clase de \hat{y} es 0
- Si x es mayor que 3, la clase de \hat{y} es 1

Este vez en lugar del 'squared error' en regresión, ya se usa el *Gini impurity* (I_G) para splitting

$$I_G(p) = 1 - \sum_{i=1}^J p_i^2$$

donde p_i es la fracción de muestras en clases i .

Cuanto más puro sea el nodo, menor será el valor de Gini.
(rango [0, 0.5])

En el caso de clasificación binaria:

$$I_G = 1 - p(\text{True})^2 - p(\text{False})^2$$

ejemplos:

$$0 \ 0 \ 0 \ 1 \ 1 \ 1 \quad I_G = 1 - \left(\frac{3}{6}\right)^2 - \left(\frac{3}{6}\right)^2 = 0.5$$

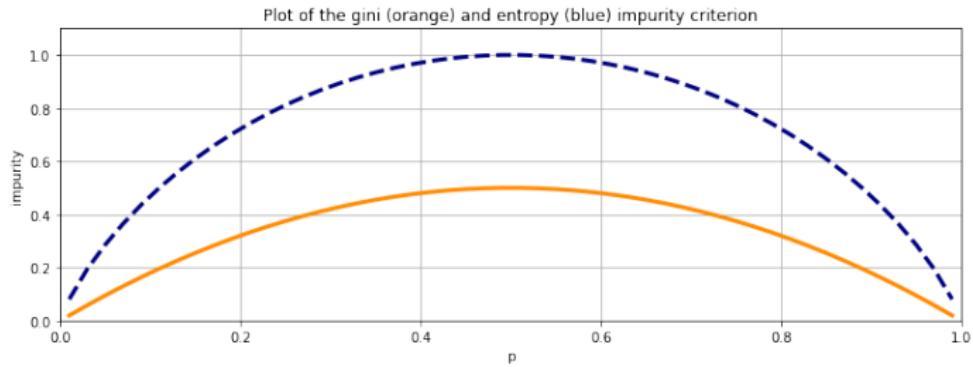
$$0 \ 0 \ 0 \ 1 \ 1 \quad I_G = 1 - \left(\frac{3}{5}\right)^2 - \left(\frac{2}{5}\right)^2 = 0.48$$

$$0 \ 0 \ 0 \ 1 \quad I_G = 1 - \left(\frac{3}{4}\right)^2 - \left(\frac{1}{4}\right)^2 = 0.375$$

$$0 \ 0 \ 0 \quad I_G = 1 - \left(\frac{3}{3}\right)^2 - \left(\frac{0}{3}\right)^2 = 0$$

$$1 \ 1 \ 1 \quad I_G = 1 - \left(\frac{0}{3}\right)^2 - \left(\frac{3}{3}\right)^2 = 0$$

¿Qué pinta tiene la Gini (y la entropía)?



A efectos prácticos son muy parecidos, y hay poco que elegir entre las dos
(fuente: [Titanic: some sex, a bit of class, and a tree...](#))

Primer modelo de clasificación:

crear un baseline análogo a $\hat{y} = \bar{y}$ $\forall \hat{y}$ en regresión:

- $\hat{y} = 0$ $\forall \hat{y}$

Es decir, las predicciones son todas la clase mayoritaria (0 por definición).
Eso se conoce a veces como el modelo **ZeroR**.

Métricas de clasificación: ¡**hay** muchas!

Accuracy score

- Es la fracción de aciertos
- `from sklearn.metrics import accuracy_score`

Si tienes un `accuracy_score < 0.5` sabes que algo va mal!

¿Qué es un buen accuracy_score?

¿Es un accuracy_score de 1.0 es posible?

Con datos reales: No

¿Es un accuracy_score de 0.51 malo?

¡No necesariamente!

Sesión práctica

Notebook: BIG DATA: Práctica: El juego de “cara o cruz”

Objetivo: Variar el parámetro ventaja y ver su efecto en tu balance final

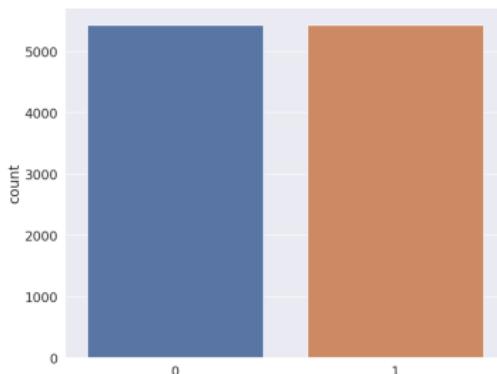
Monte Carlo...



1 en 37 veces gana la casa (el 0 verde). Eso es como ventaja = 0.027
...a lo largo la casa siempre gana su parte! ([La ley de los grandes números](#))

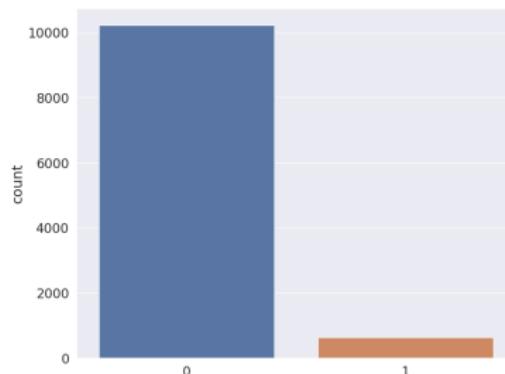
Clasificación binaria ([distribución de Bernoulli](#))

datos 'balanceados'



$$n(0) \approx n(1)$$

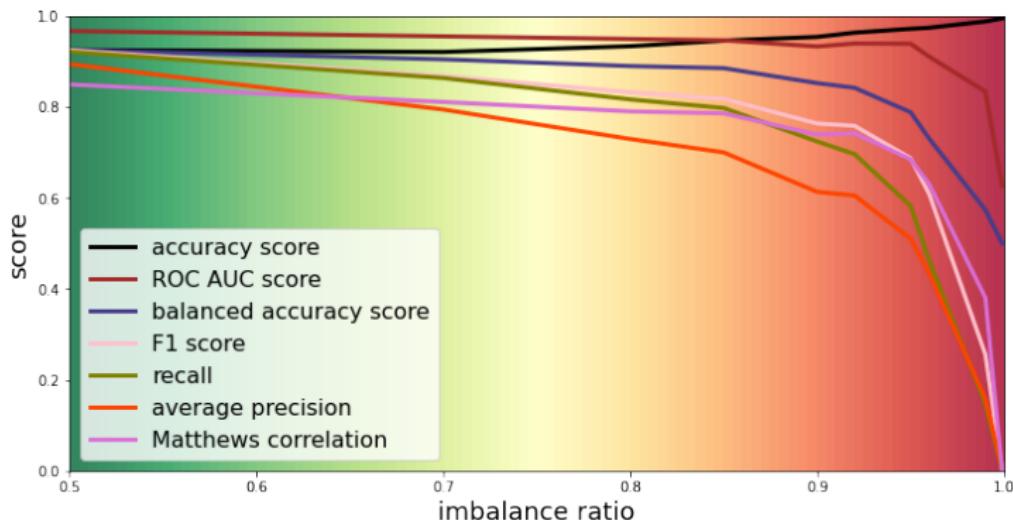
datos 'desbalanceados'



$$n(0) \gg n(1)$$

(gráfica hecho con [sns.countplot](#))

Clasificación con datos desbalanceados: métricas



(fuente: “Classification: How imbalanced is “imbalanced”?)

(Materia avanzada: paquete [imbalanced-learn](#))

Accuracy score: ¡cuidado con datos desbalanceados!

Mucho mejor siempre usar el

```
from sklearn.metrics import balanced_accuracy_score
```

En más detalle: hay 4 clasificaciones posibles:

falso positivo	(FP)	$y = 0, \hat{y} = 1$
falso negativo	(FN)	$y = 1, \hat{y} = 0$
verdadero positivo	(TP)	$y = 1, \hat{y} = 1$
verdadero negativo	(TN)	$y = 0, \hat{y} = 0$

Matriz de confusión (*confusion matrix*):

Machine learning \ Manual counting	True	False
True	True Positive (TP)	False Positive (FP)
False	False Negative (FN)	True Negative (TN)

Equations:

$$\text{False positive rate (FPR)} = \frac{\text{FP}}{\text{FP} + \text{TN}}$$

$$\text{False negative rate (FNR)} = \frac{\text{FN}}{\text{FN} + \text{TP}}$$

$$\text{Sensitivity} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

$$\text{Specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

$$\text{Youden index} = \text{Sensitivity} + \text{Specificity} - 1$$

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

Notebook: "Titanic: In all the confusion..."

...más métricas de clasificación:

Precision y recall

- $\text{precision} = \frac{TP}{TP+FP}$ (para cuando te interesan pocos falsos positivos)
- $\text{recall} = \frac{TP}{TP+FN}$ (para cuando te interesan pocos falsos negativos)

F_1 score (donde el precision es tan importante como el recall)

- $F_1 = 2 \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$
- `from sklearn.metrics import f1_score`

(Artículo: “*Evaluation Methods for Ordinal Classification*”)

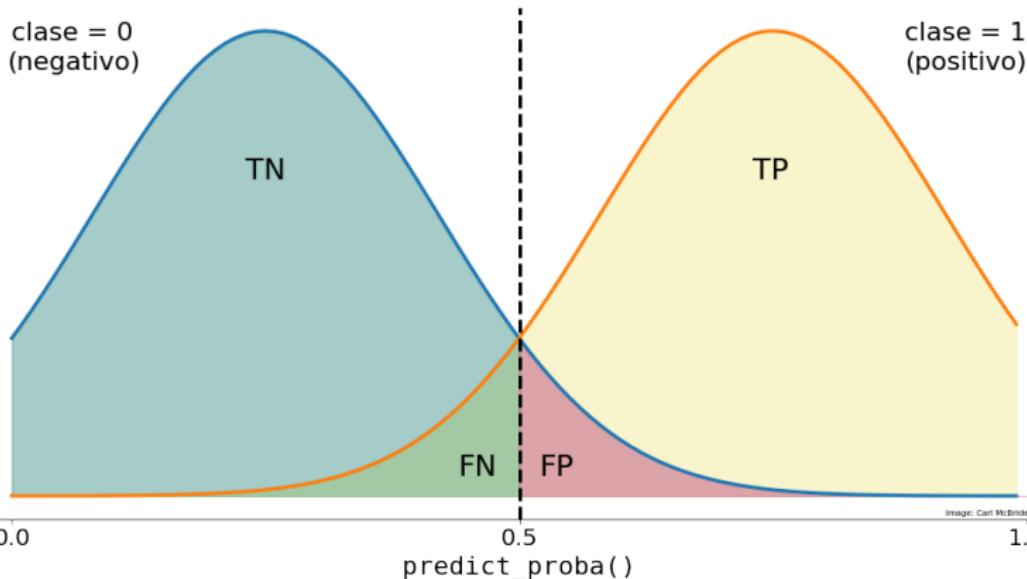
Sesión práctica

Sesión práctica: Clasificación:

Jupyter notebook: [BIG DATA: Práctica: Clasificación](#)

Objetivo: Añadir un cálculo de la métrica F_1 score a mano y con scikit-learn

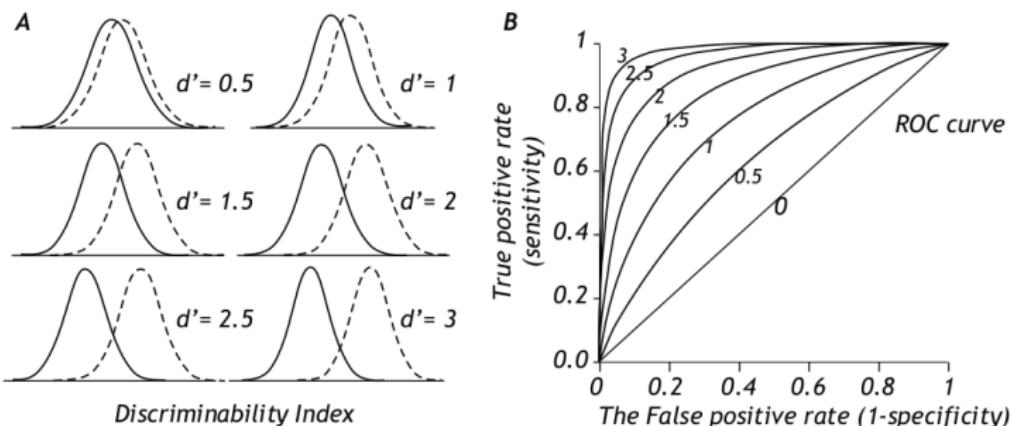
podemos dibujar un histograma de nuestras predicciones (predict_proba), sus clases y también las clases verdaderas:



Notebook: ["False positives, false negatives and the discrimination threshold"](#)

(Materia avanzada: Las probabilidades no están necesariamente bien calibradas. Ver: ["Probability calibration"](#))

El mejor clasificador es el que tiene más área (AUC) bajo la curva ROC

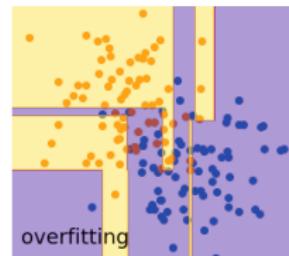
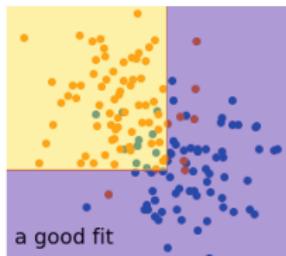
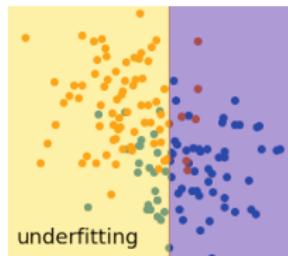
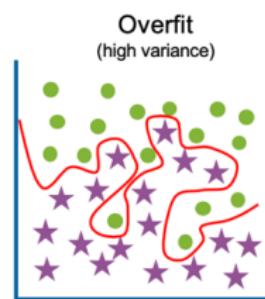
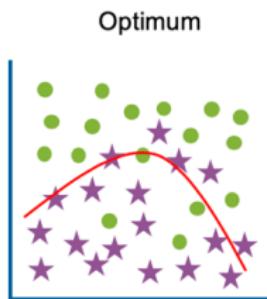
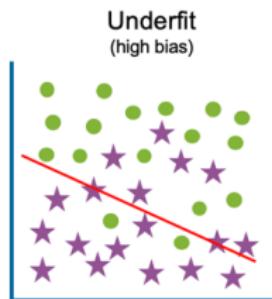


AUC = 1 separación perfecta entre clases

AUC = 0.5 es un desastre, con nada de separación

```
from sklearn.metrics import roc_auc_score
```

Sobreajuste (overfitting): clasificación:



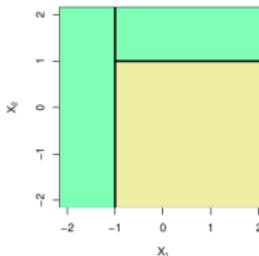
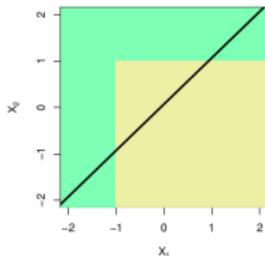
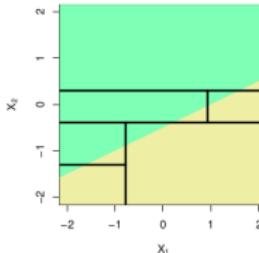
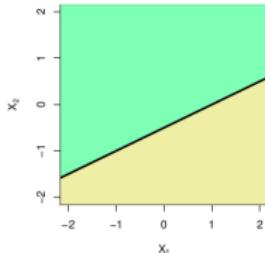
Overfitting

En clasificación, como en regression, en modelos paramétricos se puede aplicar regularización (L1, L2, L1+L2)

y en los árboles puedes usar `min_samples_leaf`

(¡Ojo: en `sklearn LogisticRegression` L2 esta activado por defecto!)

Ajuste lineal Vs árboles:



¿Cual es el mejor “estimator”?

Pues depende de cada dataset. A priori no se sabe cual va a ser el mejor

Eso se llama la “No free lunch theorem”

David Wolpert “*The Lack of A Priori Distinctions Between Learning Algorithms*” (1996)

Multiclass classification

Digamos que ya tenemos 3 classes: $\{0, 1, 2\}$

Técnica “one-vs-rest” (o “one-vs-all”)

Entrenemos 3 clasificadores binarios donde

tratar clase 0 ya como 1, y clases 1 y 2 como clase 0

tratar clase 1 ya como 1, y clases 0 y 2 como clase 0

tratar clase 2 ya como 1, y clases 0 y 1 como clase 0

El resultado es la asignación con más probabilidad de los tres clasificadores para cada punto.

Métricas multiclass

Measures for multi-class classification based on a generalization of the measures of [Table 1](#) for many classes C_i : tp_i are true positive for C_i , and fp_i – false positive, fn_i – false negative, and tn_i – true negative counts respectively. μ and M indices represent micro- and macro-averaging.

Measure	Formula	Evaluation focus
Average Accuracy	$\frac{\sum_{i=1}^I \frac{tp_i + tn_i}{\sum_{j=1}^I fn_j + fp_j + tp_i + tn_i}}{I}$	The average per-class effectiveness of a classifier
Error Rate	$\frac{\sum_{i=1}^I \frac{fp_i + fn_i}{\sum_{j=1}^I fn_j + fp_j + tp_i + tn_i}}{I}$	The average per-class classification error
Precision _{μ}	$\frac{\sum_{i=1}^I \frac{tp_i}{tp_i + fp_i}}{I}$	Agreement of the data class labels with those of a classifiers if calculated from sums of per-text decisions
Recall _{μ}	$\frac{\sum_{i=1}^I \frac{tp_i}{tp_i + fn_i}}{I}$	Effectiveness of a classifier to identify class labels if calculated from sums of per-text decisions
Fscore _{μ}	$\frac{(\beta^2 + 1)Precision_{\mu}Recall_{\mu}}{\beta^2 Precision_{\mu} + Recall_{\mu}}$	Relations between data's positive labels and those given by a classifier based on sums of per-text decisions
Precision _{M}	$\frac{\sum_{i=1}^I \frac{tp_i}{\sum_{j=1}^I fn_j + fp_j + tp_i + tn_i}}{I}$	An average per-class agreement of the data class labels with those of a classifiers
Recall _{M}	$\frac{\sum_{i=1}^I \frac{tp_i}{\sum_{j=1}^I fn_j + fp_j}}{I}$	An average per-class effectiveness of a classifier to identify class labels
Fscore _{M}	$\frac{(\beta^2 + 1)Precision_{\mu}Recall_M}{\beta^2 Precision_M + Recall_M}$	Relations between data's positive labels and those given by a classifier based on a per-class average

(Fuente: "[A systematic analysis of performance measures for classification tasks](#)")

Clustering

Clasificación “no-supervisada”: *Clustering*

Se pueden clasificar puntos aunque no tengas datos de entrenamiento:

- *K-means*
- DBSCAN

K -means (K -medias):

- ① Elige aleatoriamente K puntos (centroides)
- ② asigna cada dato al punto más cercano
- ③ mueve los dos puntos al centro de masa de sus clusters
- ④ repite los pasos 2 y 3 hasta que los puntos no se mueven más

(Nota: Los clusters de K -means forman una [teselación de Voronoi](#))

(Artículo: "[Stop using the elbow criterion for k-means](#)")

DBSCAN

DBSCAN va buscando puntos con `min_samples` de vecinos que estan dentro del radio `eps`. Estos son los puntos “*core*”

- Elegir uno de los puntos ‘core’ para iniciar el primer cluster, e ir propagando core points e incorporándolos al cluster
- Despues añadir los puntos *non-core*, pero dentro de `eps` de un punto core, al cluster
- Ahora, elegir otro core point que no esté en el cluster #1 y repetir los pasos anteriores, para formar el cluster #2
- Repetir, y cuando no quedan core points sin cluster, has terminado

Los puntos no asignados a un cluster son los “*outliers*”

Métricas no-supervisados de clustering (cuando no sabemos labels_true)

- Silhouette Coefficient - Lo mejor es 1 y lo peor es -1
- Calinski-Harabasz Index - Más grande es mejor
- Davies-Bouldin Index - Lo ideal es 0

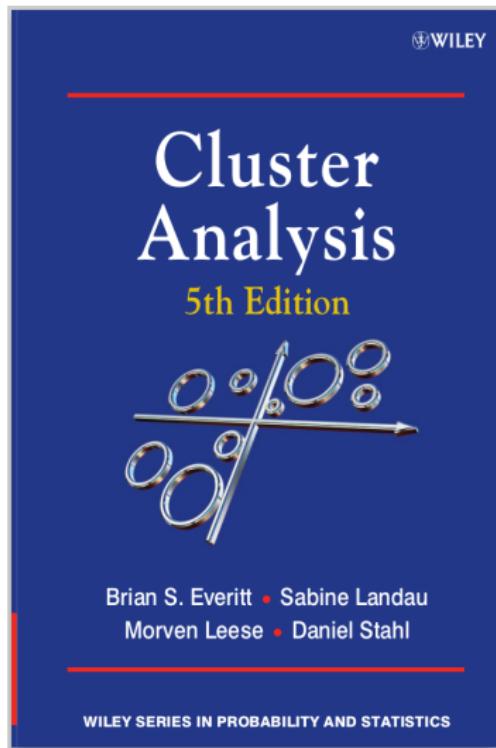
Sesión práctica

Sesión práctica: Clasificación “no-supervisada”:

Notebook: [BIG DATA: Práctica: Clasificación “no-supervisada”: Clustering](#)

Objetivo: Ver el efecto del parámetro `n_clusters` en K-means, y los parámetros `eps` y `min_samples` en DBSCAN, tanto gráficamente como los scores

Libro: "Cluster Analysis"



Meta-learners

Un colectivo de *weak learners* puede formar un *strong learner*

(Artículo: Robert Schapire "The strength of weak learnability" (1990))

Random Forest

Para prevenir el overfitting que hemos visto con un solo árbol
es mejor promediar el resultado de varios árboles

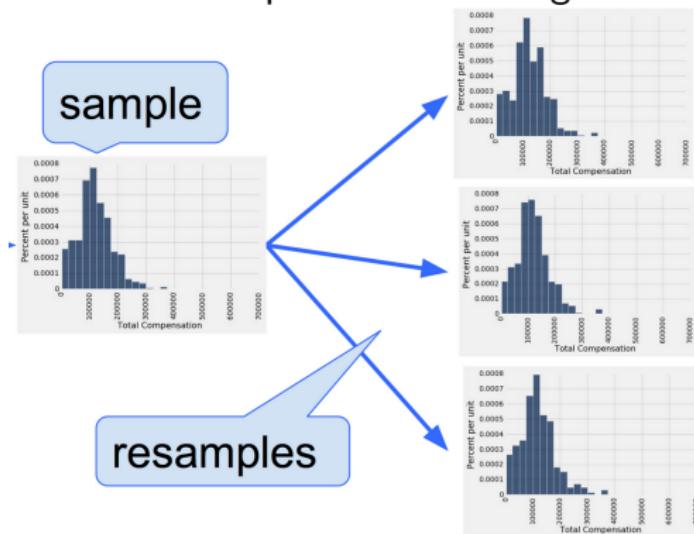
...es decir, ¡un bosque!

(nuevo hyper-parameter: `n_estimators = 100`)

Artículo: Leo Breiman "[Random Forests](#)" (2001)

Bagging ensemble

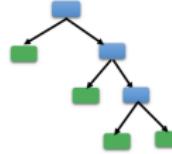
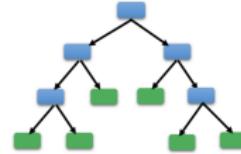
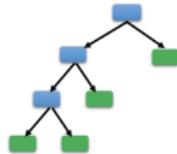
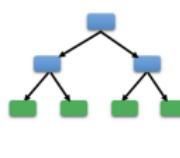
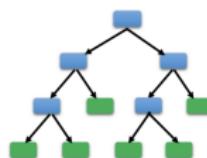
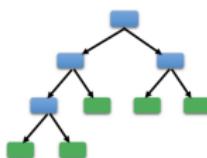
Paso 1: Usa el método *bootstrap* para fabricar n nuevos training datasets del mismo número de filas que el dataset original:



Cada dataset *bootstrap* es creado con muestras aleatorias, con re-emplazo, y contiene $\approx 63\%$ de las filas del dataset *train*.

Paso 2: ademas solo usamos unos $\sqrt{n_features}$, también elegidos al azar, en cada árbol

De este modo, cada árbol ya es un poco distinto de los demás



- Regresión: el resultado final es el promedio de los árboles
- en `sklearn`: `RandomForestRegressor`
- Clasificación: el resultado final es el “*majority vote*”
- en `sklearn`: `RandomForestClassifier`

Sesión práctica

Sesión práctica:

En nuestro notebook **BIG DATA: Práctica: Regresión con sklearn** añadir el **RandomForestRegressor**, y dibujar su curva de aprendizaje encima de la del DecisionTreeRegressor.

Es decir, cambia:

```
from sklearn.tree import DecisionTreeRegressor  
regressor = DecisionTreeRegressor(max_depth=?)
```

por

```
from sklearn.ensemble import RandomForestRegressor  
regressor = RandomForestRegressor(max_depth=?)
```

Explicabilidad

Con el modelo RandomForest ganamos en rendimiento, pero ya no podemos dibujar su árbol

Vamos perdiendo nuestra habilidad de explicar qué hace nuestro modelo

Competición

¿...y si cambiamos el Decision Tree Regressor por el Random Forest Regressor en nuestro notebook del concurso?

Random Forest hiper-parámetros: ejemplos

- `max_depth = 4`
- `min_samples_leaf = 2`
- `n_estimators = 100`
- `random_state = 42`

Boosting

AdaBoost (Adaptive Boosting)

- AdaBoostRegressor
- AdaBoostClassifier

Freund y Schapire "*A desicion-theoretic generalization of on-line learning and an application to boosting*" (1995)

Boosting es secuencial

Ajustar un nuevo decision stump al error del previo decision stump
(hiper-parámetro: `learning_rate`, η)

En regresión primero se resta la media del *target* de la columna *target*.
Después se ajusta un árbol de decisión a este nuevo conjunto de targets
(i.e. una columna compuesta por los ε , formando el *pseudo-residual*). Y ya
se restan las predicciones del árbol de estos targets para otra nueva
columna de targets, cada vez mas cerca de cero. Se sigue así
`n_estimators` veces.

En clasificación reemplazamos los labels por las probabilidades.

Al contrario que bagging, boosting sí puede sobreajustar con más
estimadores.

Gradient boosting

Los 'tres grandes' (SoTA):

- [XGBoost](#) - “eXtreme Gradient Boosting” (2014)
- [LightGBM](#) - “Light Gradient Boosting Machine” (2016)
- [CatBoost](#) - “Categorical Boosting” (2017)

Nota: Estos paquetes no pertenecen a `scikit-learn`, y por tanto tienen un API un poco diferente

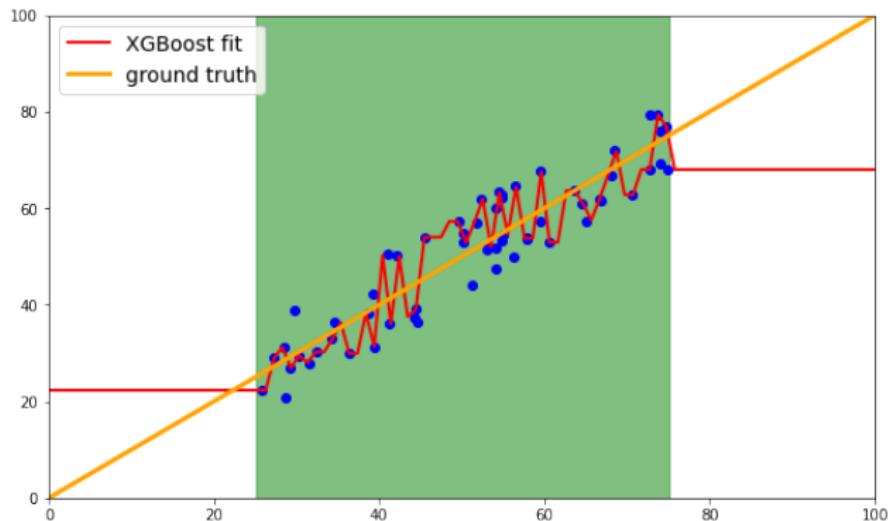
Jerome H. Friedman “*Greedy function approximation: A gradient boosting machine*” (2001)

Artículo: “*Why do tree-based models still outperform deep learning on typical tabular data?*” (Julio 2022)

En cambio de AdaBoost, Gradient boosting usa un función de loss + gradient descent + búsqueda lineal en cada paso

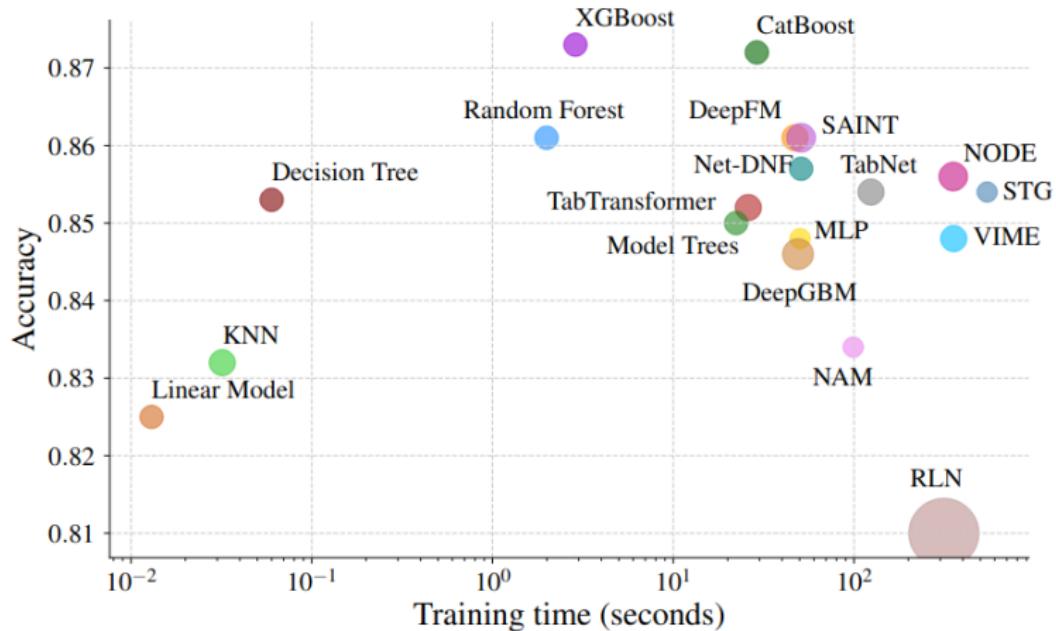
Extrapolación

¡Ojo! Siempre ten cuidado con extrapolación, más aun con estimadores basados en árboles de decisión:



Notebook: [Extrapolation: Do not stray out of the forest!](#)

(Artículo: "[Towards Out-Of-Distribution Generalization: A Survey](#)")



Fuente: “*Deep Neural Networks and Tabular Data: A Survey*” (Junio 2022)
 (ver tb. SoTA benchmarks)

Flujo de trabajo: Estimators

Creando un punto de referencia (*baseline*):

Usa estimadores que no sobreajusten, es decir con muy poca varianza p.ej. LR o LR siendo simples, no son propensos a sobreajustar, y son computacionalmente baratos, así que son buenos puntos de partida.

Modelo #0 Baseline (un modelo “no modelo”):

- Regresión: mean del target
- Clasificación: Accuracy del label mode (0)

Modelo #1 (un modelo con muy poca varianza):

- Regresión: [Linear regression](#)
- Clasificación: [Logistic regression](#)

Modelo #2: Random Forest (`RandomForestRegressor`,
`RandomForestClassifier`)

Modelo #3: SoTA: Gradient boosting ensemble estimators:

Estos dan los mejores resultados para datos tabulares hoy en día:

- `XGBoost` el primero y más famoso (CPU/GPU)
- `LightGBM` muy rápido aunque solo usa CPU
- `CatBoost` bueno con datos tipo 'categorical' (CPU/GPU)

Notebook: Ejemplo de como hacer `regresión con XGBoost`

Hyper-parameters

Hyper-parameters vs. parameters

Hemos visto

- el grado del polinomio
- la profundidad del árbol: `max_depth`
- el numero de arboles: `n_estimators`
- α de gradient descent
- λ de regularization
- *etc.*

estos son ejemplos de “hyper-parameters”

Los parámetros son valores hallados por el modelo que mejor se ajusta al conjunto de datos, por ejemplo los β en regresión lineal

Los hiper-parámetros definen el modelo, y se eligen antes de ajustar a los datos y hallar los parámetros

Árboles de decisión: hyper-parameters

Hay muchos hyper-parameters. Estos son los más importantes:

- `DecisionTree`: `max_depth`
- `RandomForest`: `max_depth`, `n_estimators=100`
- `XGBoost` (valores predeterminados: `max_depth=6`,
`n_estimators=100`, `learning_rate=0.3`)

Todos: `min_samples_leaf`

Mi consejo de donde empezar tu hyper-parameter optimization:

- `max_depth = 4`
- `min_samples_leaf = 5`
- `n_estimators = 100`
- `learning_rate = 0.1` (o incluso menos; 0.05)

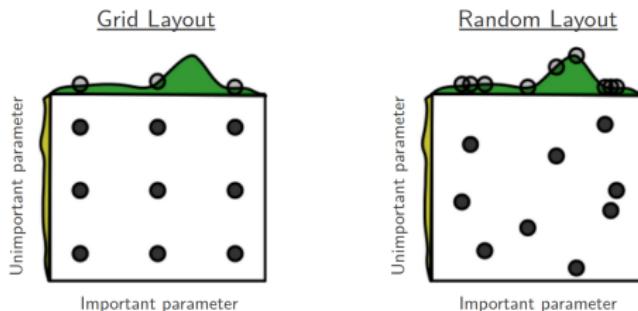
¿Cómo encontrar el mejor conjunto de hiper-parámetros?

Se puede comprobar permutaciones de hiper-parámetros usando:

`sklearn.model_selection.GridSearchCV`

`sklearn.model_selection.RandomizedSearchCV`

`sklearn.model_selection.HalvingGridSearchCV`



ver: “Tuning the hyper-parameters of an estimator”

(Paquetes: [Hyperopt](#), [Optuna](#))

(Artículo: “[Hyper-Parameter Optimization: A Review of Algorithms and Applications](#)”)

Ejemplo:

```
from sklearn.model_selection import GridSearchCV

param_grid = {"max_depth": [3, 4, 5],
              "min_samples_leaf": [2, 3, 4],
              "n_estimators": [200, 400, 600]}

search = GridSearchCV(regressor, param_grid,
                      cv=5).fit(X_train, y_train)

print("The best hyperparameters are ", search.best_params_)
```

Nota: este ejemplo implica hacer un total de $27 \times 5 = 135$ ajustes...

Feature engineering and selection

Mejora tu modelo con '*feature engineering*': selección (*dropping/sparsity*), creación (*adding*)

- un modelo con pocas features es más entendible
- un modelo con muchas features puede dar un mejor *metric score*

Libro (gratis online): "*Feature Engineering and Selection: A Practical Approach for Predictive Models*" Max Kuhn y Kjell Johnson

Feature engineering es quizás la parte más artesanal de ML

- fabricar nuevas features no-lineales, *p.ej.* x^2 , $x^{1/n}$,...
- fabricar nuevas ‘interaction’ features, *p.ej.* $(x_1 + x_2)$, $(x_1 \times x_2)$,...
- transformar features con *p.ej.* $\log(x + 1)$, $\sin(x)$,...
- transformación de Yeo-Johnson (`scikit power_transform`)
- datos “externos”: *p.ej.* fechas de fiestas, el tiempo,...
- latitud y longitud datos a distancias
- ...
- peor caso: ¡obtener más datos!

(Paquetes: [Featuretools](#), [Feature-engine](#))

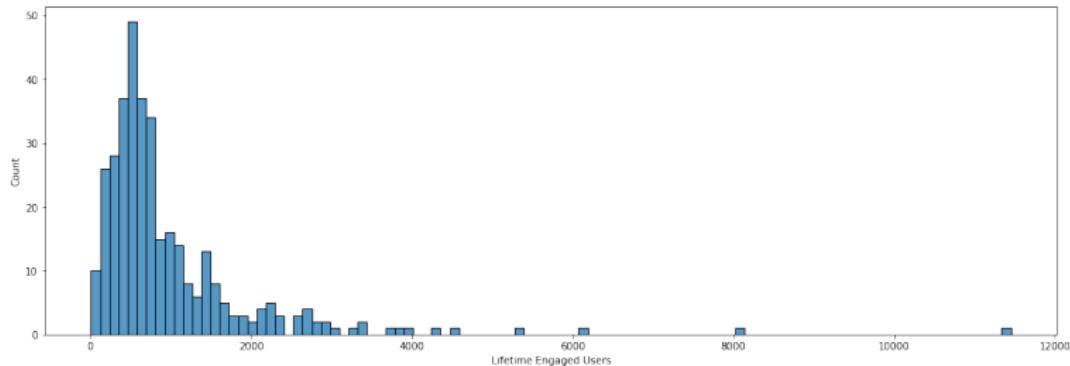
ejemplos:

- `df["x1_por_x2"] = df["x1"] * df["x2"]`
- `df["log_x1"] = np.log(df["x1"]+1)`

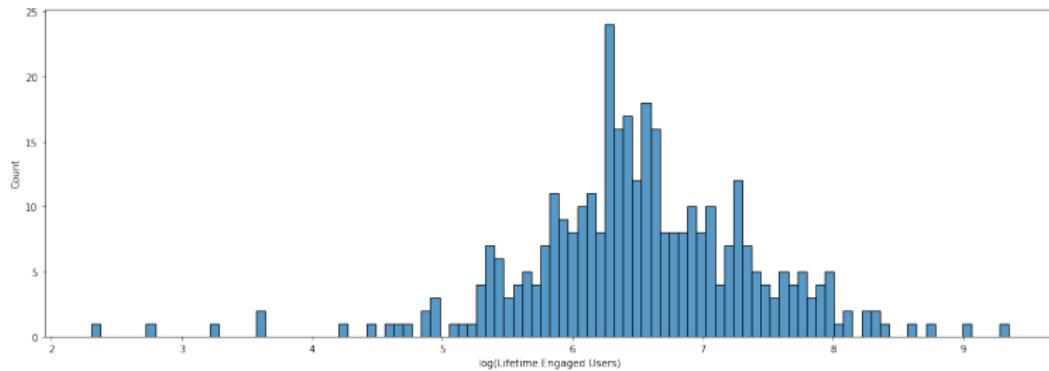
(Nota: $\ln(0) = -\infty$)

- `def fn(x):`
 $y = f(x)$
 `return y`
`df["nueva_col"] = df["col_original"].apply(fn)`

Lognormal \leftrightarrow Gaussian transform: Antes:



despues:



Principal component analysis (PCA)

(Pearson (1901))



(*"Principal Component Analyses (PCA)-based findings... ...must be reevaluated"* (2022))

Feature selection: para encontrar features débiles y eliminarlas del modelo

- eliminar features de cero (o poca) varianza

(p.ej. con: `sklearn.feature_selection.VarianceThreshold`)

- eliminar features con mucha colinealidad o correlación

- `permutation_importance`

- Stepwise regression:

- recursive feature elimination: quitar el feature menos predictivo
 - ‘recursive’ o ‘forward’ feature addition (`f_regression`)

- `feature importance plots`: visualización de importancias

- usar `LASSO`

- el paquete `Boruta-SHAP`

Permutation importance:

- 1. Reordenar aleatoriamente una sola columna (feature)
- 2. Re-calcular el metric
- 3. Su importancia es proporcional a cuánto empeora el resultado
- 4. Repetir para todas las columnas (features)

Height at age 20 (cm)	Height at age 10 (cm)	...	Socks owned at age 10
182	155	...	20
175	147	...	10
...
156	142	...	8
153	130	...	24

(Fuente: "Permutation Importance" por DanB)

Notebook: "[House Prices: Permutation Importance example](#)"

Artículo: "[Unrestricted Permutation forces Extrapolation: Variable Importance Requires at least One More Model](#)"

ejemplo de permutation importance usando ELI5

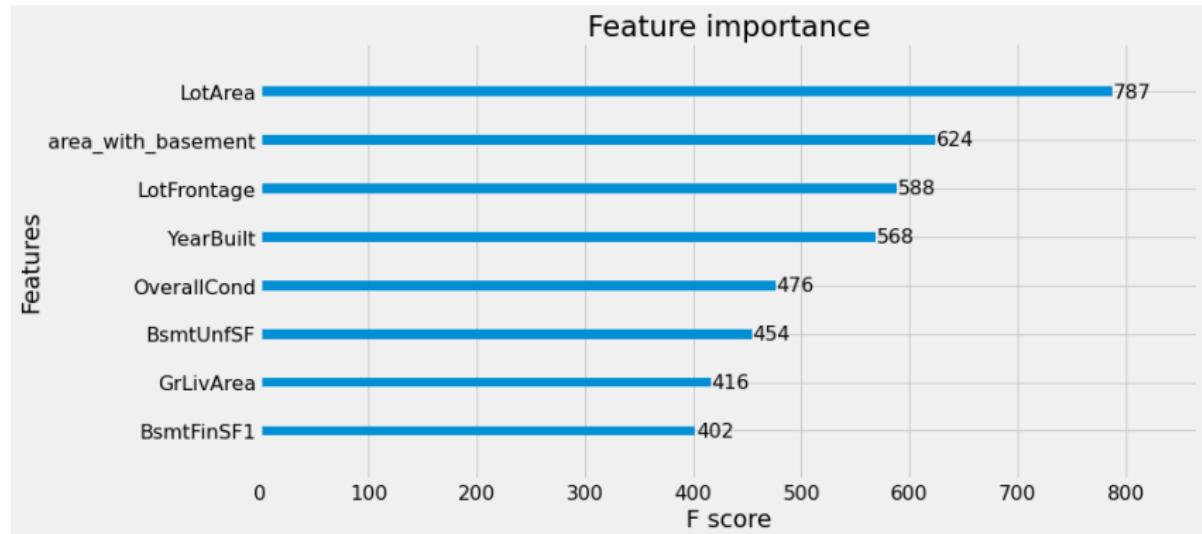
Weight	Feature
1.1113 ± 0.0215	m2_edificados
0.3970 ± 0.0121	precio_zona
0.0194 ± 0.0013	precio_alquiler
0.0065 ± 0.0006	ascensor
0.0019 ± 0.0004	exterior
0.0007 ± 0.0001	aparcamiento
0.0004 ± 0.0001	n_habitaciones
0.0003 ± 0.0002	m2_útiles
0.0002 ± 0.0001	orientaciónEste
0.0002 ± 0.0002	precio_aparcamiento
0.0002 ± 0.0001	nueva_construcción
0.0001 ± 0.0001	orientaciónOeste
0.0001 ± 0.0000	orientaciónNorte
0.0001 ± 0.0000	planta
0.0001 ± 0.0001	n_baños
0.0000 ± 0.0000	orientaciónSur
0.0000 ± 0.0000	año_construcción
0.0000 ± 0.0000	necesita_reforma

Recursive feature elimination (RFE)

- Elegir el numero de features que quieres (n)
 - 1. Calcular la importancia de cada feature (p.ej. con permutation importance)
 - 2. Eliminar el feature menos importante
 - 3. Repetir hasta que solo quedan n features

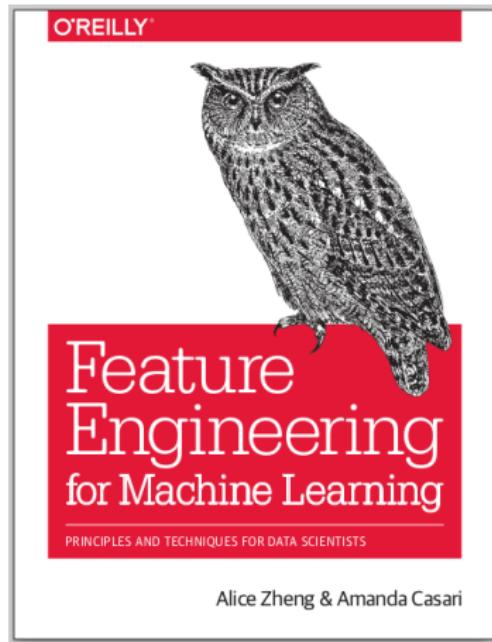
Notebook: “[Recursive Feature Elimination \(RFE\) example](#)”

Feature importance plot; ejemplo:



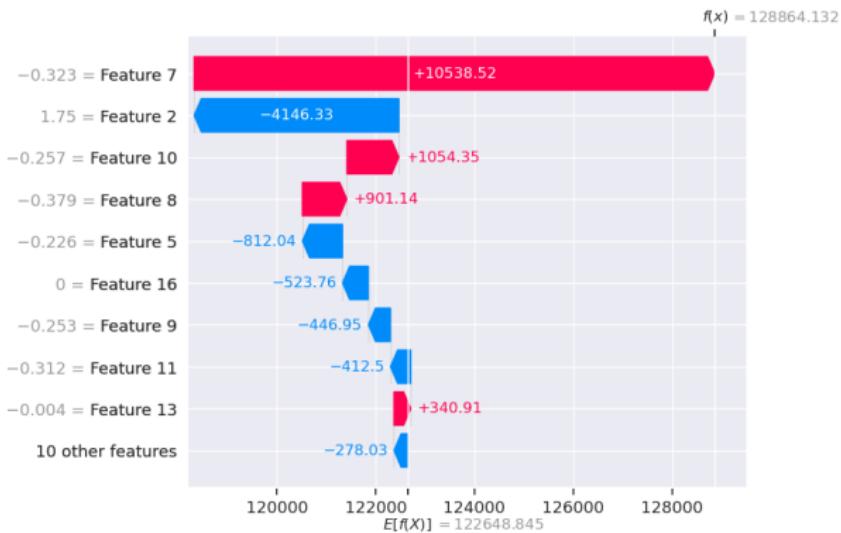
Fuente: notebook: "[An introduction to XGBoost regression](#)"

Libro recomendado: “Feature Engineering for Machine Learning”



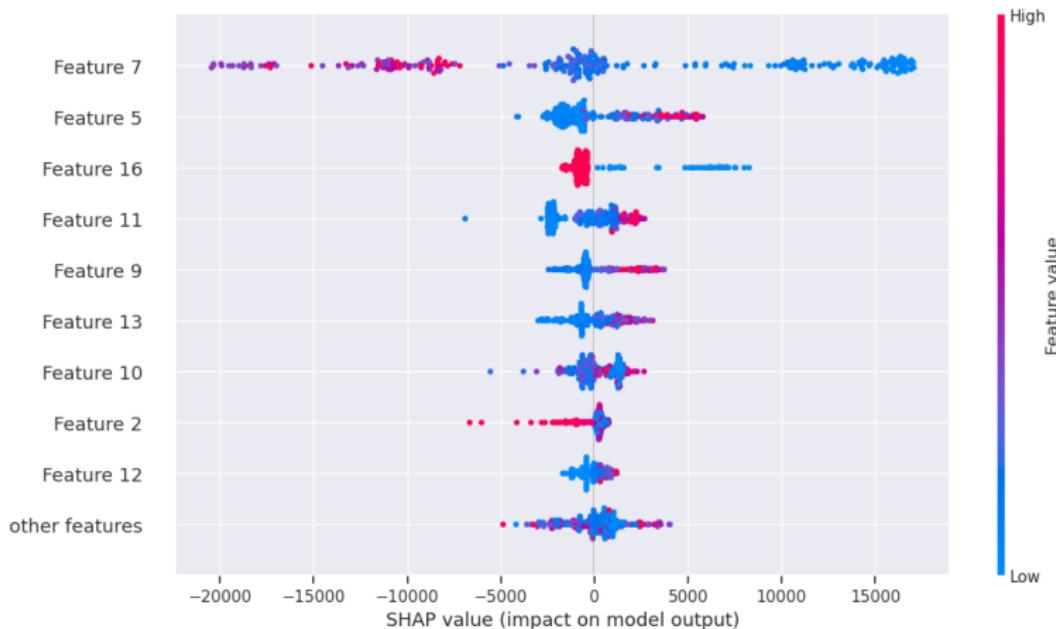
Explicabilidad

SHAP (SHapley Additive exPlanations) - agnóstico de modelo
Análisis local (por fila): 'Waterfall' plot:



($f(x)$ = valor predicho, $E[f(x)]$ = valor esperado)

Análisis global: ‘Beeswarm’ plot:



(Ver tb.: "GPU accelerated SHAP values")

Ensembling

Usando el truco de “ensembling” se puede sacar un poco más rendimiento

Regresión: usar la media aritmética de los resultados de un conjunto de estimadores diversas

Clasificación: “*majority voting*”: usar la moda de un número impar de estimadores

Notebook: [BIG DATA: Ensembling \(Regresión\)](#)

Más detalles: [“Kaggle Ensembling Guide”](#)

Estimators

Regresión

- `LinearRegression`
- `DecisionTreeRegressor`
- `RandomForestRegressor`
- XGBoost: `XGBRegressor`

Clasificación

- `LogisticRegression`
- `DecisionTreeClassifier`
- `RandomForestClassifier`
- XGBoost: `XGBClassifier`

pero hay muchos más...

- Ridge, LASSO, ElasticNet
- Generalized Linear Models (GLM)
- Generalized Additive Models (GAM) ($\text{GAM} \supset \text{GLM}$)
- Support Vector Machines (SVM)
- Gaussian Processes (GP)
- Extremely Randomized Trees (`ExtraTreesRegressor`)
- Regularized Greedy Forest (RGF)
- Linear Trees
- `HistGradientBoostingRegressor`
- ...



“Todos los modelos son incorrectos, pero algunos son útiles”
George Box

'Online' machine learning

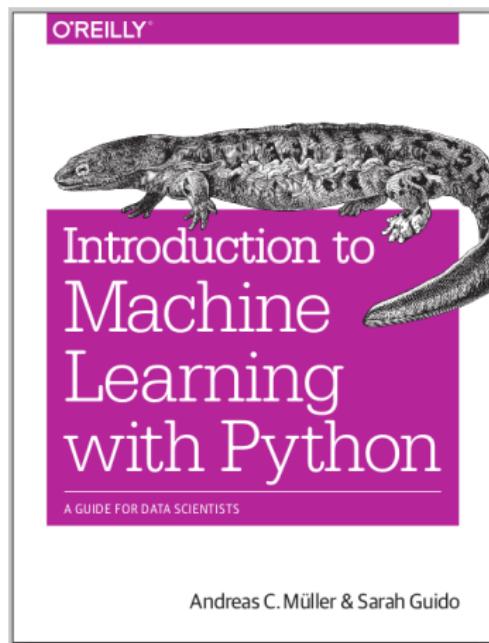
Paquetes para 'incremental', 'stream' o 'online' learning:

- River (antes `scikit-multiflow`)
- Vowpal Wabbit
- Apache Beam (Batch + stream = Beam)

para cuando recibimos nuevos datos constantemente, y/o hay `concept drift` o “covariate shift”.

Recursos

Libro recomendado: “[Introduction to Machine Learning with Python](#)”



Libros:

- “*Introduction to Machine Learning with Python*”, Müller y Guido
- “*The Elements of Statistical Learning*”, Hastie, Tibshirani y Friedman
- “*Pattern Recognition and Machine Learning*”, Bishop
- “*Métodos de Data Science aplicados a la Economía y a la Dirección y Administración de Empresas*” (Editorial UNED)

arXiv review papers:

- “*How to avoid machine learning pitfalls: a guide for academic researchers*”
- “*Model Evaluation, Model Selection, and Algorithm Selection in Machine Learning*”
- “*Hyper-Parameter Optimization: A Review of Algorithms and Applications*”

Blogs:

- [Machine Learning Mastery](#) por Jason Brownlee
- [Analytics Vidhya](#)
- [Towards Data Science](#)
- [Statology](#)

Foros de Q&A:

Informática:

- [Stack Overflow](#)

Estadística:

- [Cross Validated](#)

BIG DATA Notebooks

"Learning By Doing"

- BIG DATA: Filtrando datos
- BIG DATA: Práctica: Regresión lineal con python
- BIG DATA: Práctica: Regresión con sklearn
- BIG DATA: Práctica: El juego de "*cara o cruz*"
- BIG DATA: Práctica: Clasificación
- BIG DATA: Práctica: Clasificación "*no-supervisada*": Clustering
- BIG DATA: Ensembling (Regresión)
- BIG DATA: Plantilla para el concurso