Data Science & Machine Learning

Módulo 5: Data Science & Machine Learning (ML)

Impartido por: Carl McBride Ellis

(enlace al pdf: https://github.com/Carl-McBride-Ellis/DSyML/M5.pdf)

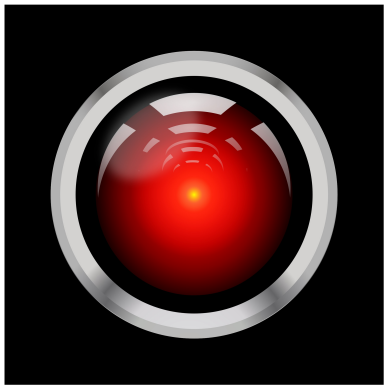
(v.0.0.4)

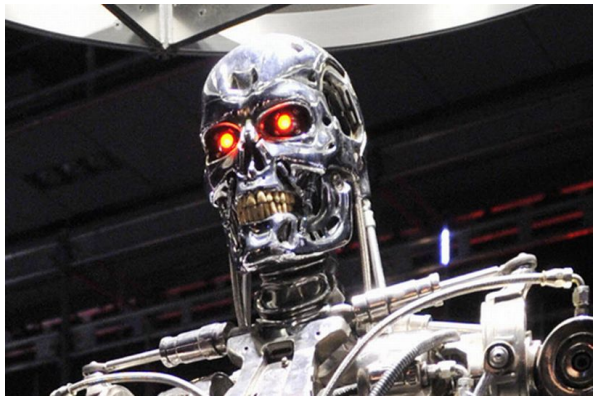
Tres principales áreas de aplicación:

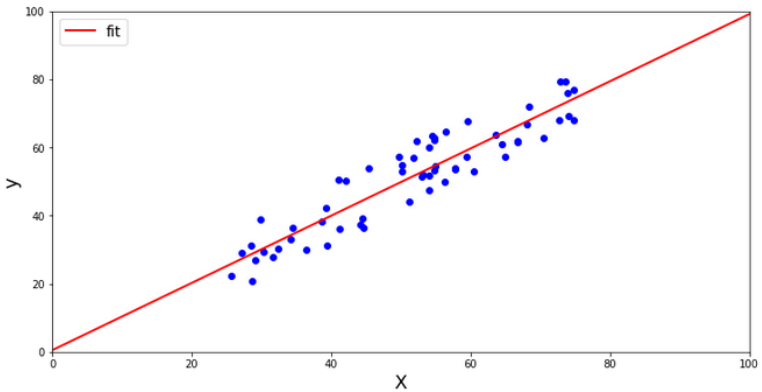
- deep-learning con redes neuronales (NN):
  - Computer Vision (CV) - imágenes
  - procesamiento de lenguaje natural (NLP) - texto
- machine learning (ML) - datos tabulares

Vamos a enfocar en el área más fundamental: ML de datos tabulares

- **ML $\neq$ inteligencia artificial (AI)**
- **ML $=$ ajustes de datos usando la estadística**






fit
y
X
0
20
40
60
80
100

¡Competición!

¡Esta semana vais a ser data scientists!

A lo largo de la semana vais a hacer vuestro propio proyecto de ML. Vais (solos o en parejas) a predecir el numero de 'likes' de páginas Facebook.

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: Facebook competición (Zaragoza)

Introducción

¿Que son "datos tabulares"?

Ejemplo de un `DataFrame` (frecuentemente abreviado a `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)

Un dataframe tiene una fila especial con las nombres de las columnas.
Eso se llama la "header"

También tiene una columna especial para numerar las filas
Eso se llama la "index"

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:

los datos = señal + ruido (Gaussiana)
es decir, $y = f(x) + \epsilon$ donde ϵ es el 'error irreducible'

¿Que es un 'científico de datos'?

"...es alguien que es mejor en estadística que un informático, y mejor en informática que un estadístico"

¿Porque hacemos ML?

- Predecir: crear un modelo predictivo en y
- Entender: llegar a conocer que factores influyen en y

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")
```

Ya una copia del fichero `mis_datos.csv` existe en la memoria del notebook como la entidad `df` en formato "dataframe"

(kaggle memoria: CPU: 30GB RAM, GPU: 13GB RAM)

Nota: Aunque solemos leer ficheros `.csv`, los dataset muy grandes ($>$ 1G) pueden venir en ortos formatos, p.ej:

- `feather`
- `hdf5`
- `jay`
- `parquet`
- `pickle`
- bases de datos + SQL + `pySpark`

También hay alternativas al pandas `dataframe`, por ejemplo:

- `dask`
- `datatable`
- `rapids` (usando los GPU)

Para saber más sobre `pandas`:

manual 'on-line':

- Pandas User Guide

libros:

- "Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython" Wes McKinney
- "LEARNING pandas" Stack Overflow contributors

notebook 'chuleta':

- Pandas one-liners

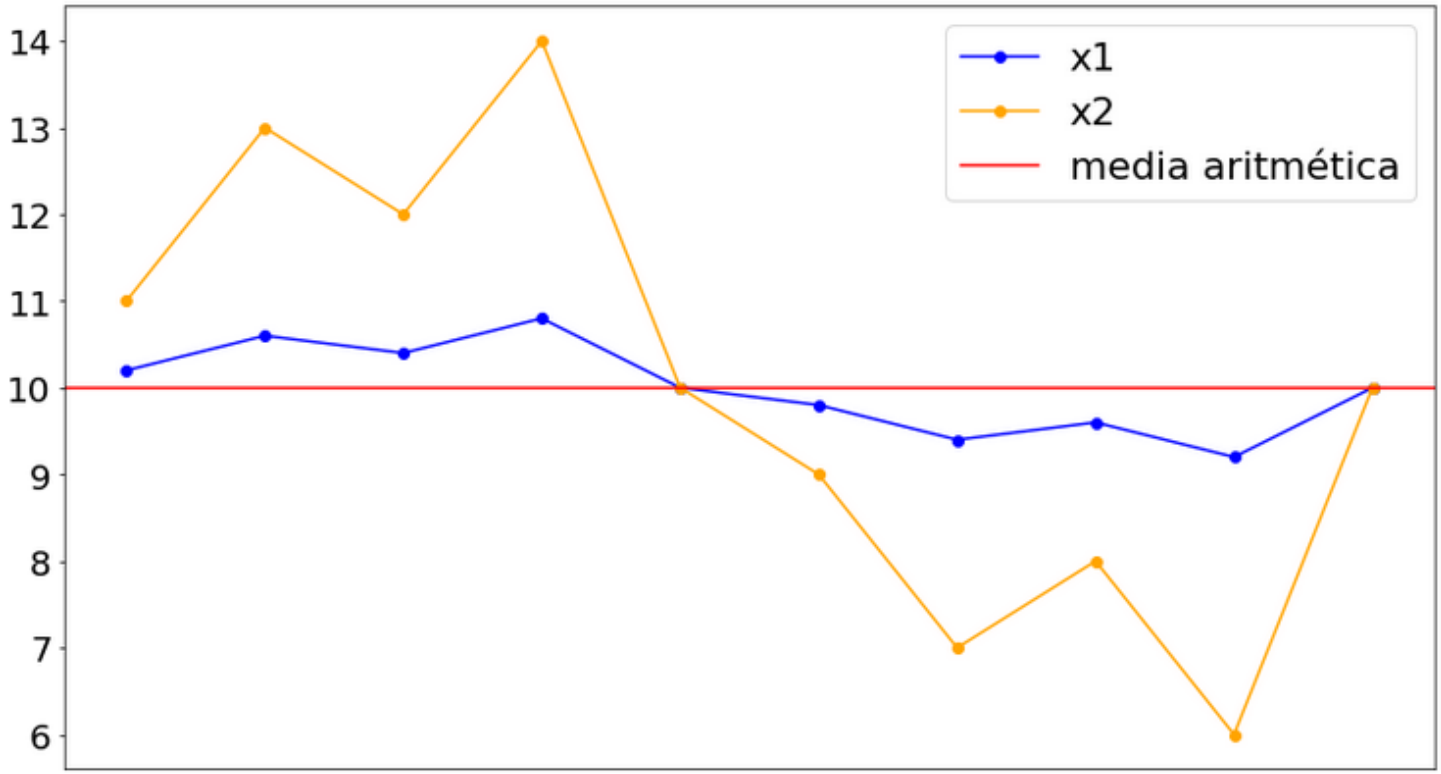
Estadística básica

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

La varianza es una medida de ruido
(más adelante vamos a ver la MSE y la RMSE)

Ejemplo:



media: $\bar{y}(x1) = \bar{y}(x2) = 10$
varianza: $\sigma^2(x1) = 0.26, \sigma^2(x2) = 6.6$
desviación estándar: $\sigma(x1) = 0.52, \sigma(x2) = 2.58$

Vemos que la desviación estándar tiene valores más "intuitivos" que la varianza

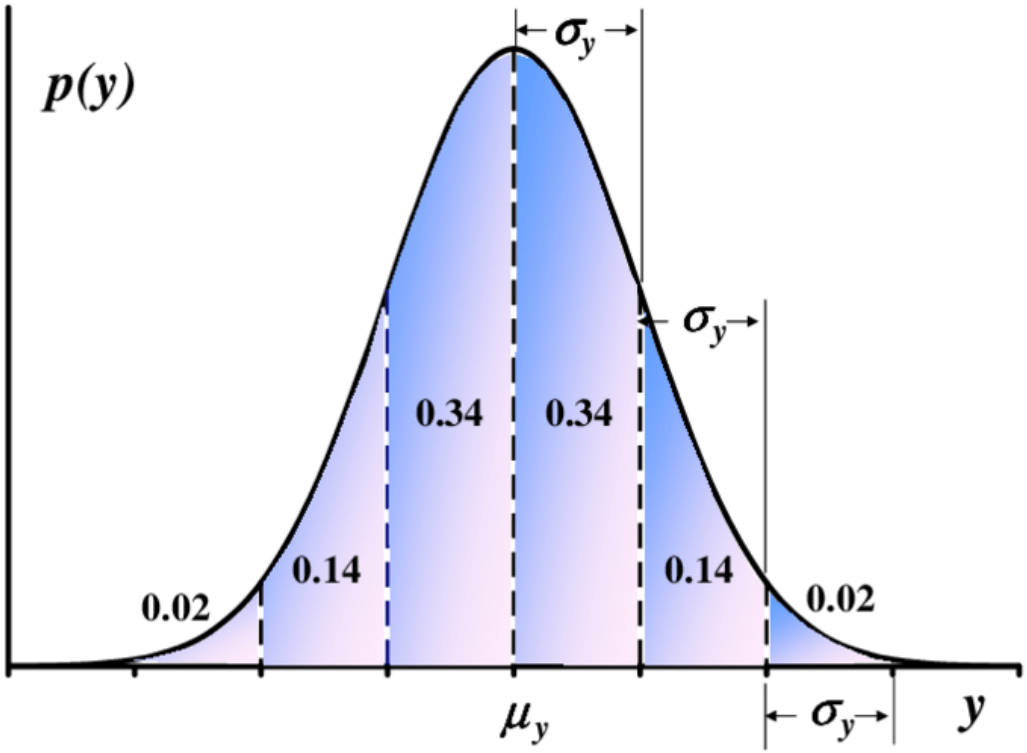
Jupyter notebook: BIG DATA: Media aritmética y varianza

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"

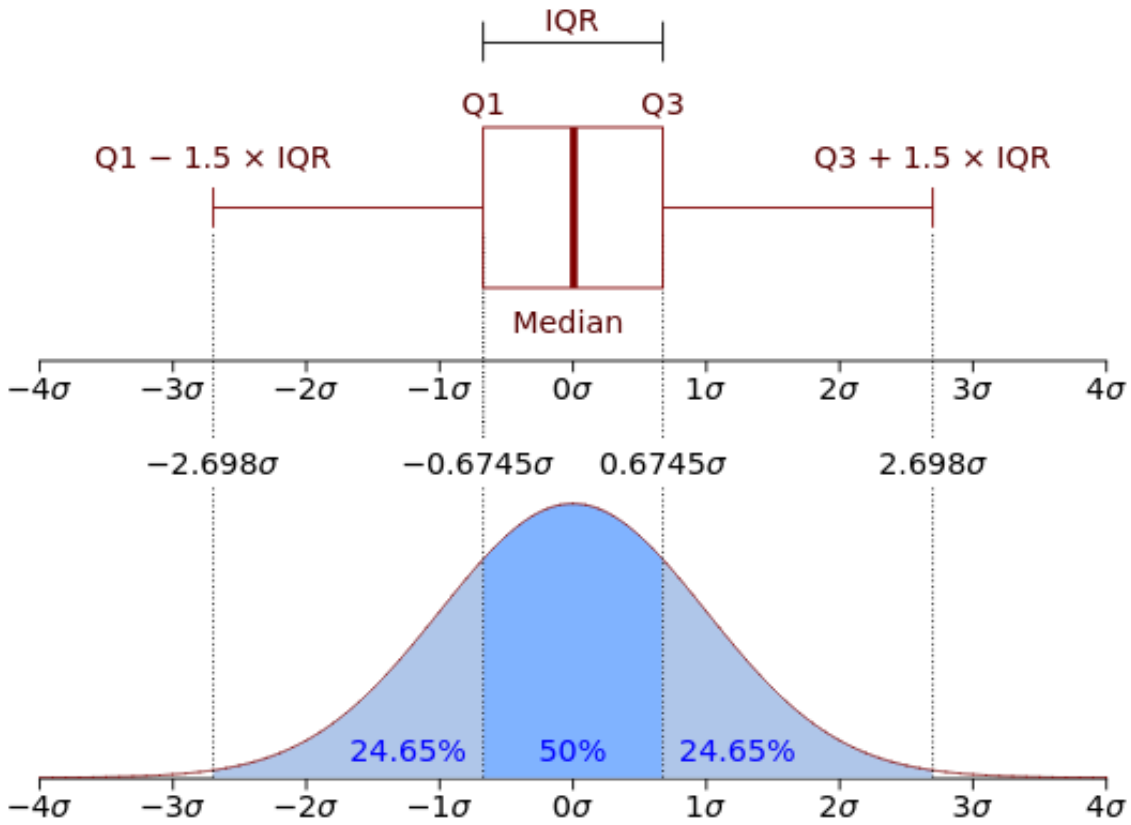
(Notebook: Animated histogram of the central limit theorem)

¿Que pinta tiene una distribución Gaussiana?:



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

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

Análisis exploratorio de datos (EDA)

¿Por qué exploración y análisis (EDA)?

"basura entra, basura sale" (GIGO)

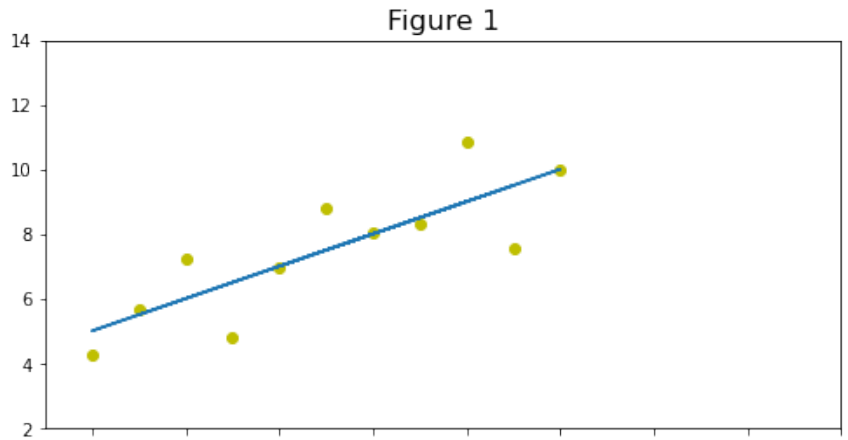
Llegando a conocer a tu dataframe (`df`):

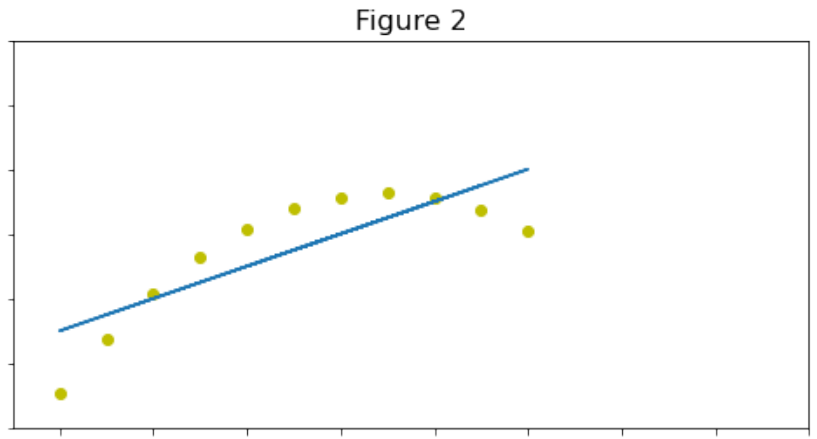
- `df.shape` - da el número de rows y columnas del `df`
- `df.head(n)` - imprime los primeros n rows
- `df.tail(n)` - imprime los últimos n rows
- `df.dtypes` - da los tipos de las columnas (float, integer, *etc.*)
- `df.isna().sum()` - número de valores ausentes en cada columna
- `df.isna().sum().sum()`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

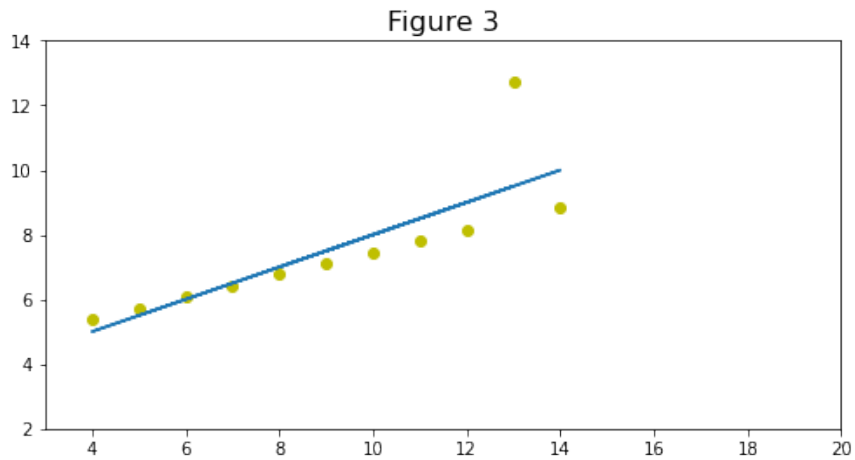
EDA

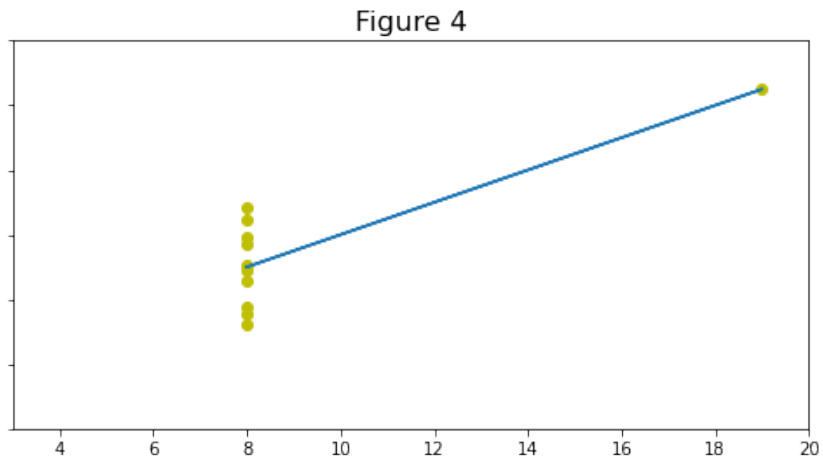
- ¿Por qué?: Anscombe's quartet
- valores ausentes
- outliers
- correlación (Pearson o Spearman)
- pair plots
- feature selección - nos da ideas

Visualización de datos: "El cuarteto de Anscombe":









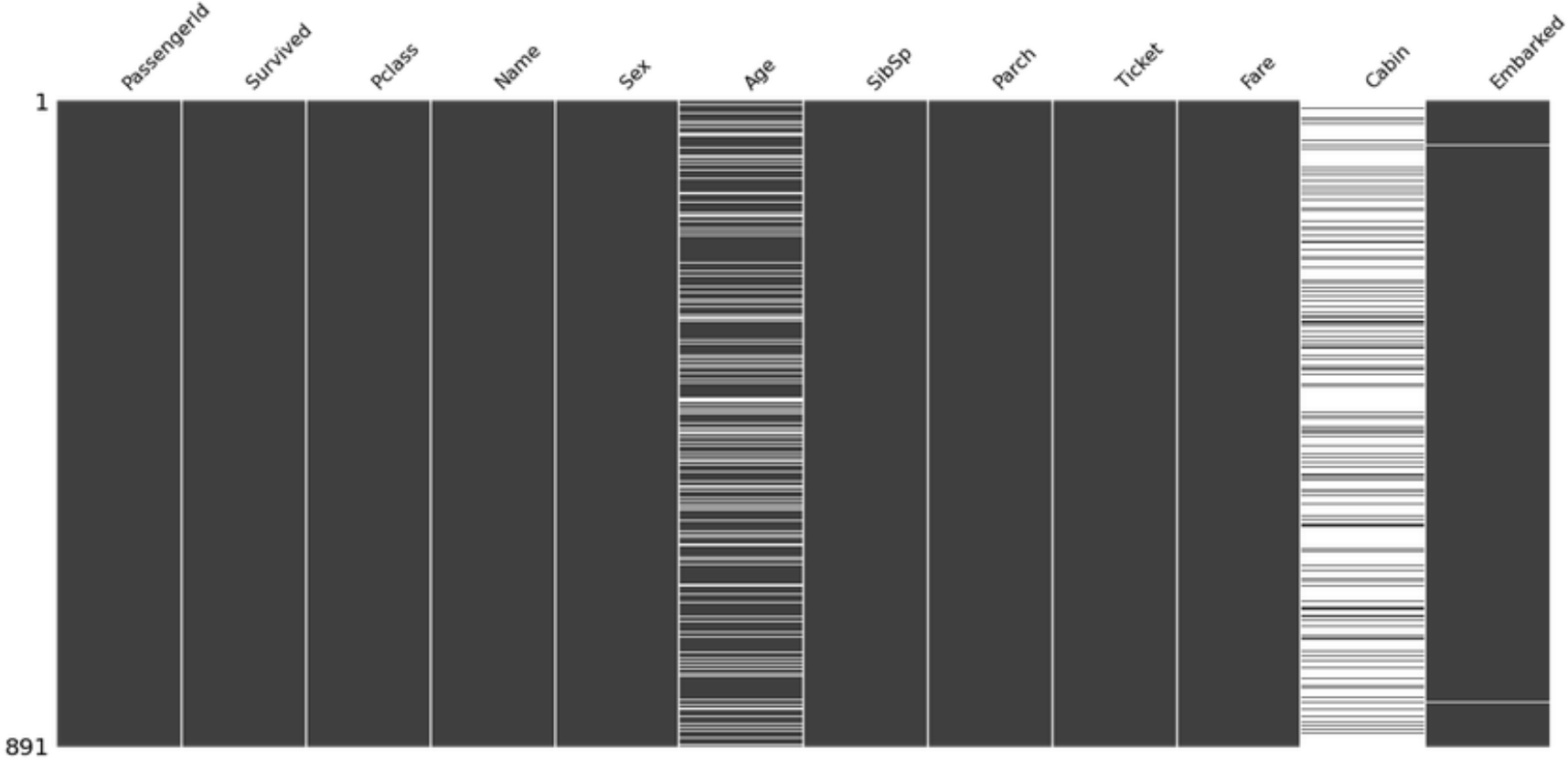
Notebook: Anscombe's quartet and the importance of EDA

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

- 1. un buen ajuste con una linea recta
- 2. 'underfitting'
- 3. tenemos un punto 'outlier'
- 4. tenemos un punto de 'levarage'

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

Visualizar los valores no disponibles ("missing"):



Paquete: `missingno`

¿Que hacer con los datos ausentes (`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. -999)
- otro método...

como: por ejemplo usar el `Simple Imputer` de scikit-learn

Lo más fácil para empezar: reemplazar todos los `NaN` con un `0`:

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

Valores *outlier* (valores atípicos o extremos):
Causas:

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

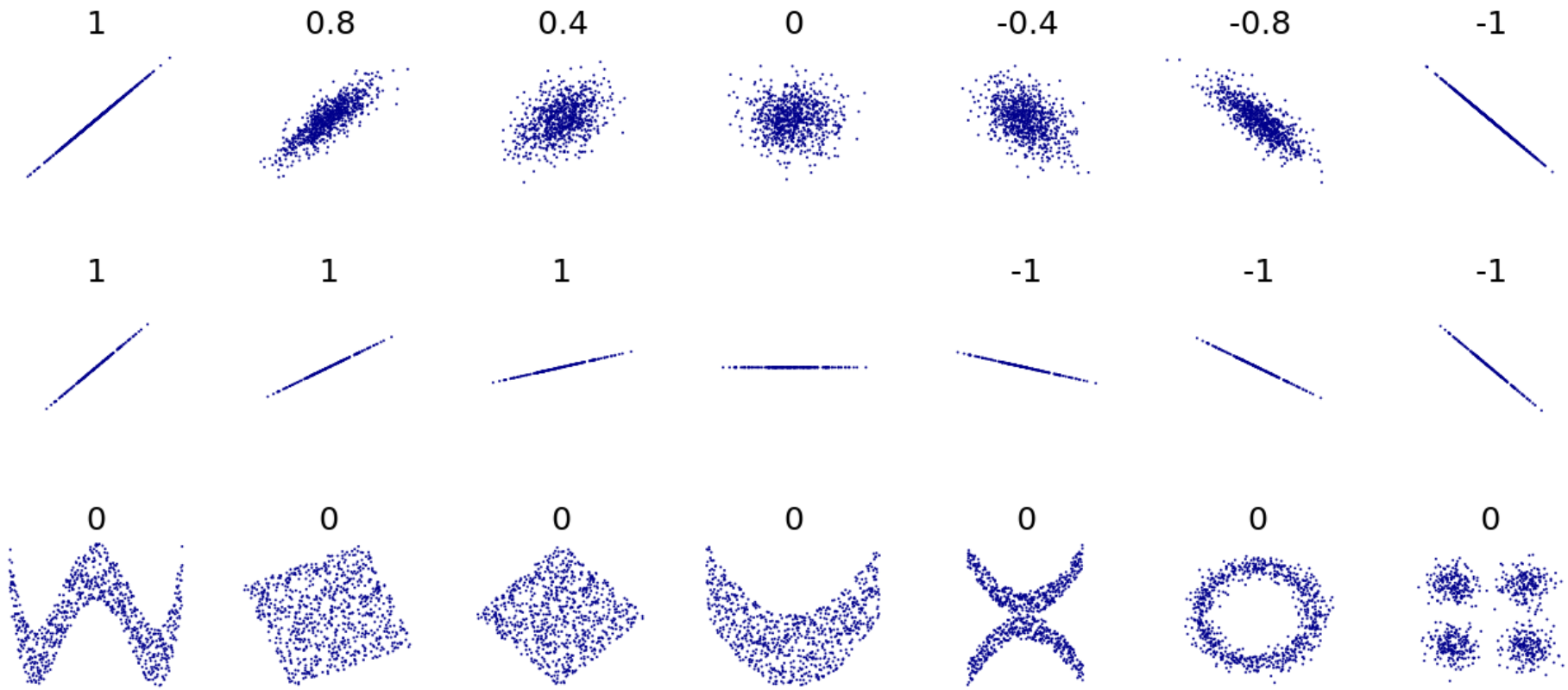
¿Detección de outliers?:

- Z-score $= \frac{X-\bar{x}}{\sigma} =$ `stats.zscore()`
- valores $> |3|$ están más lejos de μ que 99.7% del resto de los datos
- visualizar, p.ej. con un box o raincloud plot: Box+strip+violin = raincloud plot

¿Que hacer con los outliers?:
después de seria consideración puede llegar a cambiar o incluso borrar los outliers

Note: Cuidado con data leakage entre `test` y `train`

Correlación (Pearson):



Recuerda: *"Correlación no implica causalidad"*

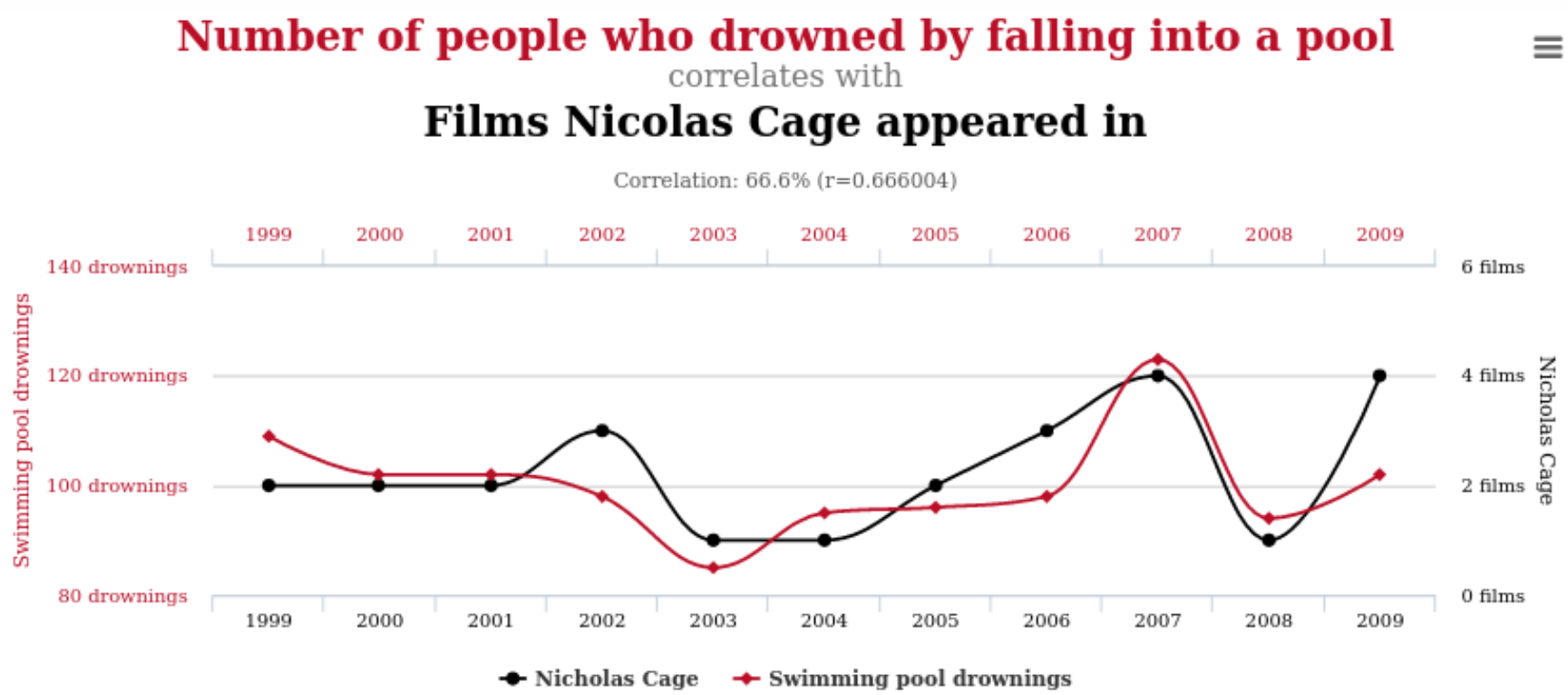
Matriz de correlación:

	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

Nota: columnas con mucho correlación entre si suelen ser malos; complican mucho la interpretación de un ajuste

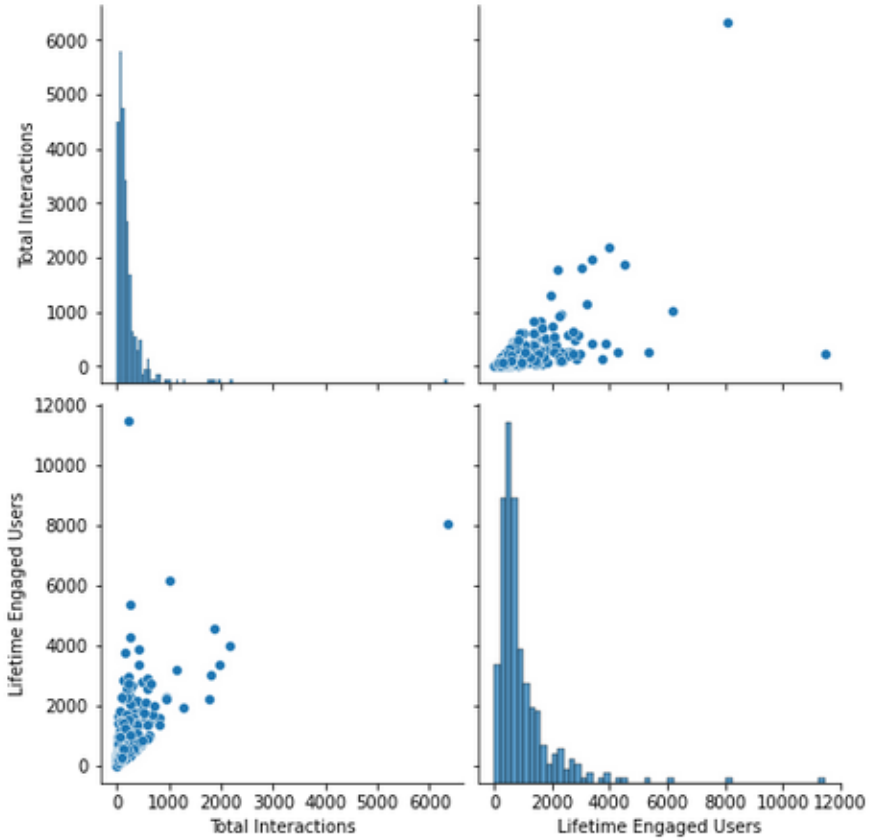
(ver tb. : Explainability, collinearity and the variance inflation factor (VIF))

Correlación espuria:



(fuente: la pagina web Spurious Correlations)

Pairplot:



función: `sns.pairplot(df, height=3)`

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

Visualización de datos: las 3 librerías principales

- matplotlib
- seaborn
- plotly express - interactivo

Vamos a usar principalmente matplotlib:

```
import matplotlib.pyplot as plt
```

Ejemplos de gráficos y su correspondiente código fuente:

- matplotlib gallery
- seaborn gallery
- plotly express gallery

Sesión práctica: EDA

Sesión práctica #1: EDA:

Jupyter notebook: BIG DATA: Práctica 1: Facebook dataset EDA (plantilla)

Vamos a explorar nuestro Facebook dataset

Aprendizaje automático

Fundamentos del aprendizaje automático

Dos tareas principales:

- regresión
- clasificación

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

Miramos al 'target'; a ver si son valores continuos o si son discretos
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

- Para que ML funcione hacen falta, por lo menos, $\texttt{n\_filas} > 20 \times \texttt{n\_columnas}$
- las columnas son las *"features"*
- las filas son las *"feature vectors"*
- hay una columna muchas veces llamada *'target'* para regresión, o *'label'* para clasificación, y suele ser la primera o la última columna

Herramientas esenciales

Entornos:

- Jupyter notebooks / Jupyter Lab / Google Colab

Lenguaje:

- python 3

Librerías:

- numpy
- pandas
- matplotlib
- scikit-learn

Como científico de datos tus notebooks (o scripts) de `python` casi siempre van a empezar así:

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

Cómo instalar paquetes o librerías nuevas:

```
!pip install nombre_del_paquete
```

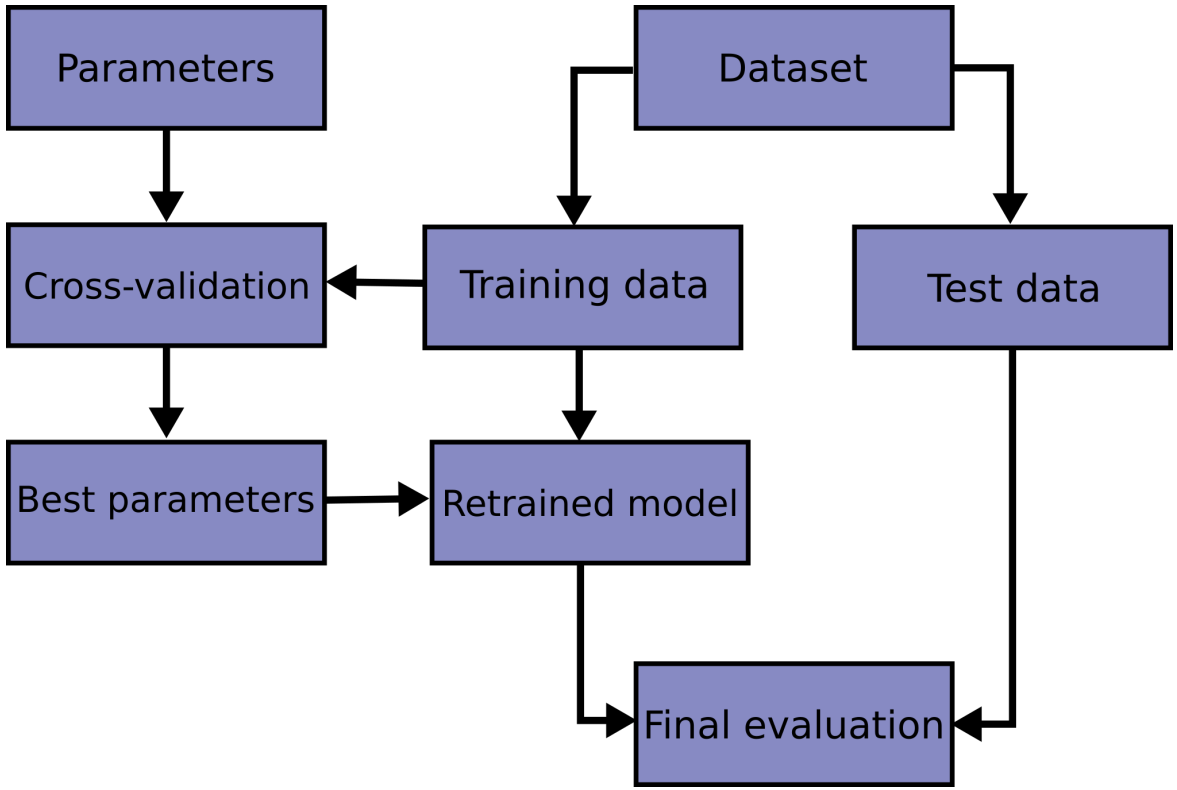
Se pueden encontrar muchos paquetes interesantes en

The Python Package Index (PyPI)

en la sección de

Scientific/Engineering :: Artificial Intelligence

Flujo de trabajo:



Datos de entrenamiento, validación y test

- train (los datos de entrenamiento) para los parámetros
- validation (or hold-out) para los hiper-parámetros
- test - para ver qué tal todo

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: 70:15:15
Nota: Queremos que los tres datasets sean *iid* (suele bastar con un *split* aleatorio)

Materia avanzada: Prueba de Kolmogórov-Smirnov

Materia avanzada: What is Adversarial Validation?

Apartado: Números aleatorios

En ML sueles ver mucho `random_state = 42` o `seed = 42`

Nuestro RNG de juguete: los ultimos dos digitos de $n \times 9$:

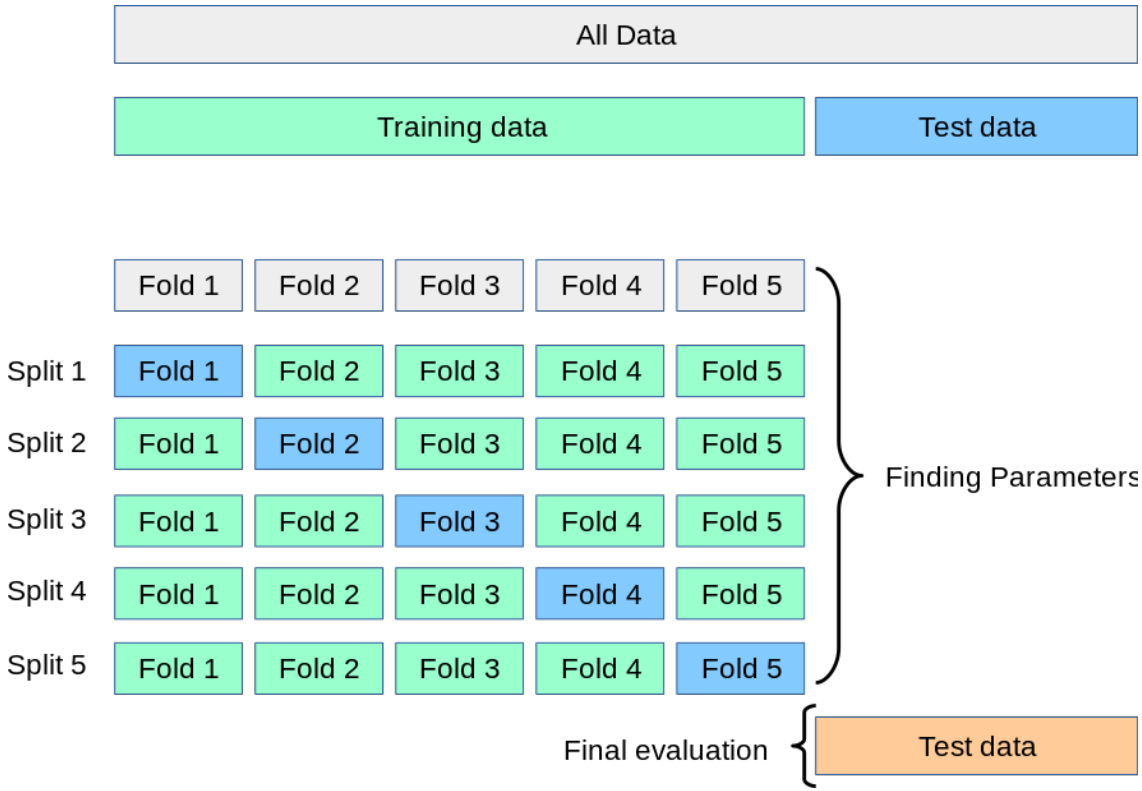
`seed=7`: $7 \to 63 \to 67 \to 03 \to 27 \to 43...$
`seed=2`: $2 \to 18 \to 62 \to 58 \to 22 \to 98...$

Eligimos un `seed` para que nuestros resultados sean *'reproducible'*

¿Por qué 42?

(Ver también: "The Lehmer RNG algorithm for seed=42")

¿Que pasa si tenemos pocos datos?
k-fold cross-validation:



Más información: Cross-validation: evaluating estimator performance

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"*

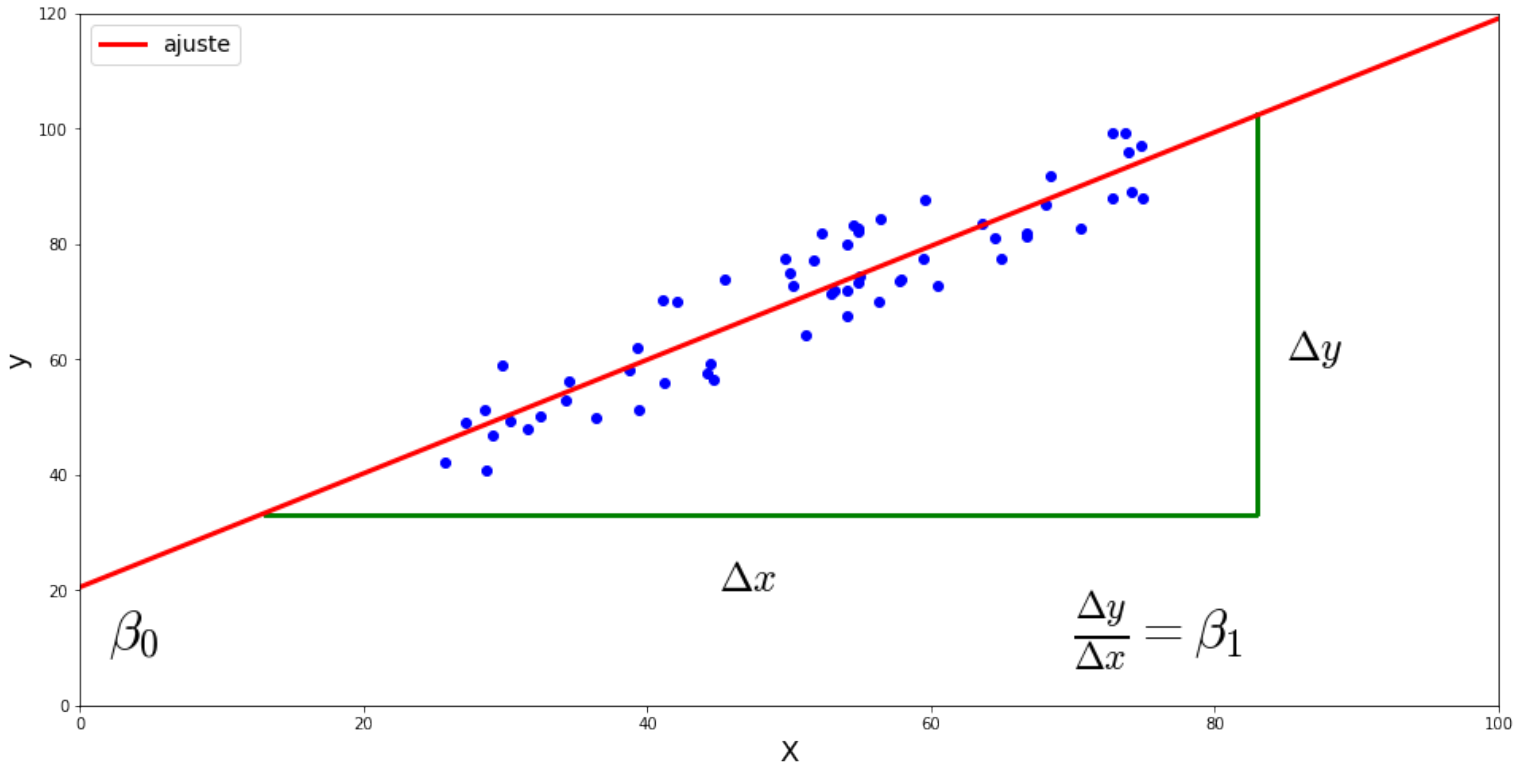
Eso es cuando sabemos qué va a pasar en el futuro

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

Problema: que suele devolver resultados demasiado optimistas

Regresión

Regresión lineal: Ejemplo en 1D (*"univariate linear regresión"*):



El pendiente: nomenclatura

una linea recta:

$$pendiente = \frac{\Delta y}{\Delta x}$$

una curva (la derivada):

$$pendiente = \frac{\mathrm{d}y}{\mathrm{d}x}$$

una curva multi-dimensional (la derivada parcial)

$$pendiente = \frac{\partial y}{\partial x}$$

Modelo de regresión lineal:

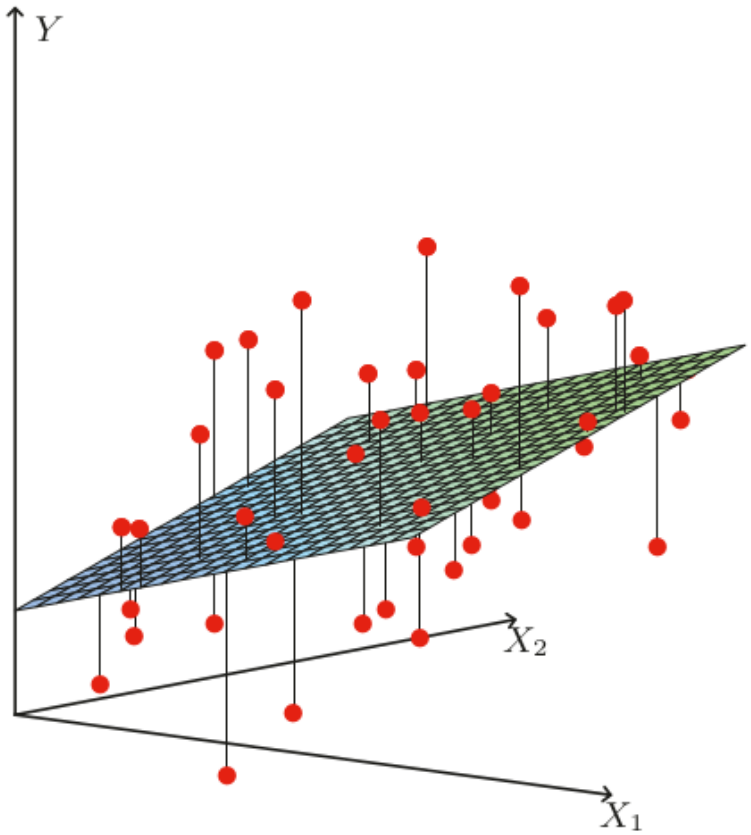
$$\hat{y}(\mathbf{X_1}) = \beta_1 \mathbf{X_1} + \beta_0$$

Para cada *'feature'* $(\mathbf{X})$ hay dos parámetros ajustables: β_1 y β_0

- β_1 es la pendiente
- β_o es la intersección o *'bias'*

Eso es un modelo "paramétrico".

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



Regresión lineal: (univariate) regresión polinómica

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

$$\hat{y}(\mathbf{X_1}) = \beta_2 \mathbf{X_1}^2 + \beta_1 \mathbf{X_1} + \beta_0$$

(Nota: las regresiones polinómicas de grandes potencias tienen colas muy inestables)

Explicabilidad (*explainability*):

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

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

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

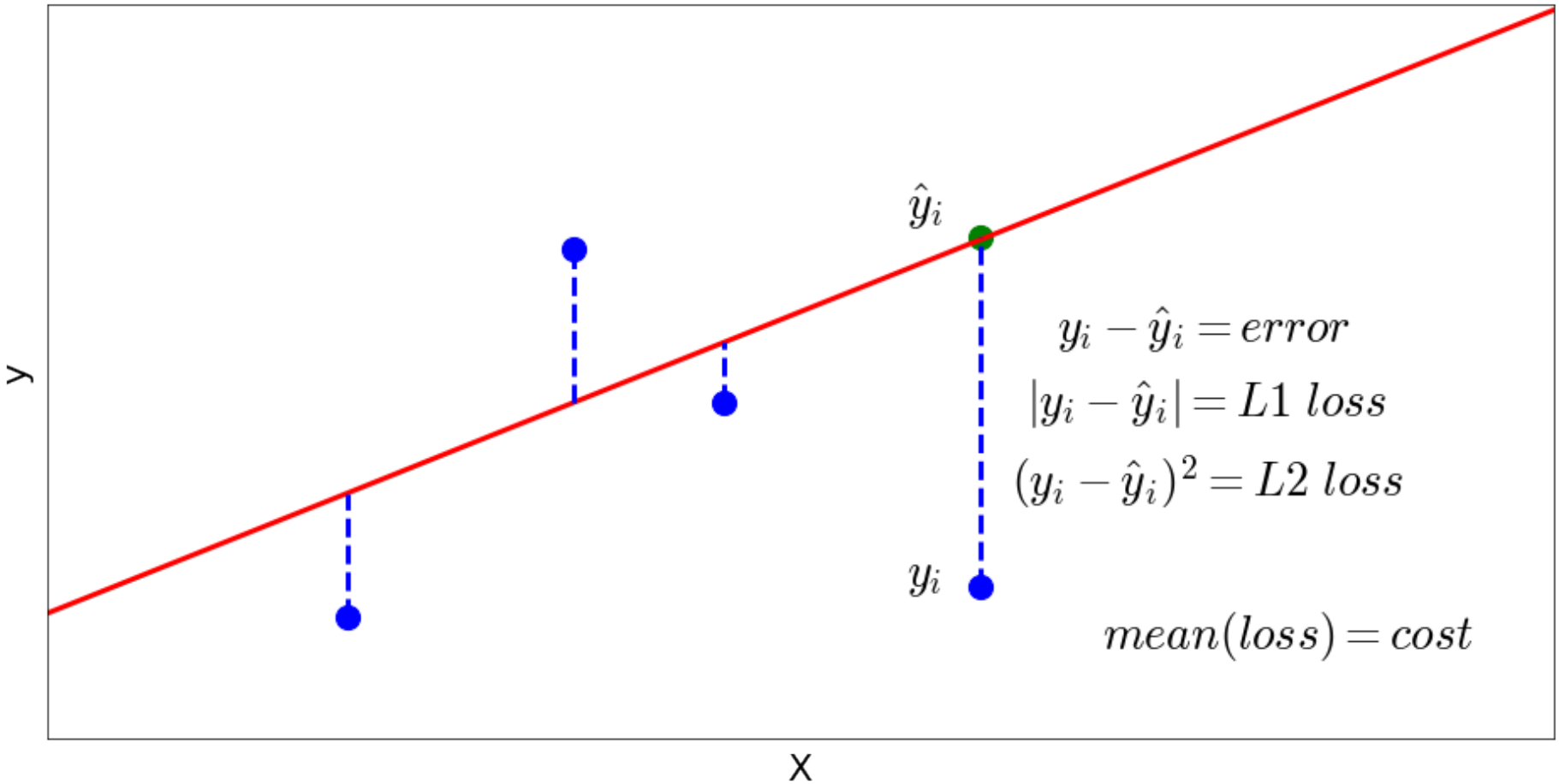
- *loss* es una medida de distancia entre dos puntos $\hat{y}$ y y
- *cost* es la media de todos estos puntos de loss: $\bar{\mathcal{L}}$

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

Regresión: Funciones de *'loss'* y *'cost'*

- L1 loss = `abs(y_true - y_pred)` = AE (absolute error)
- L1 cost fn = MAE (mean absolute error)
- L1 loss no es muy sensible a los outliers
- L2 loss = `(y_true - y_pred)**2` = SE (squared error)
- L2 cost = MSE = $\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$
- L2 loss sí es muy sensible a los outliers debido a las diferencias al cuadrado

En código $\hat{y}$ esta escrito como `y_pred`
`(y_true - y_pred)` se conoce como el error o el "residuo"


y
X
$\hat{y}_i$
y_i
$y_i - \hat{y}_i = error$
$|y_i - \hat{y}_i| = L1\ loss$
$(y_i - \hat{y}_i)^2 = L2\ loss$
$mean(loss) = cost$

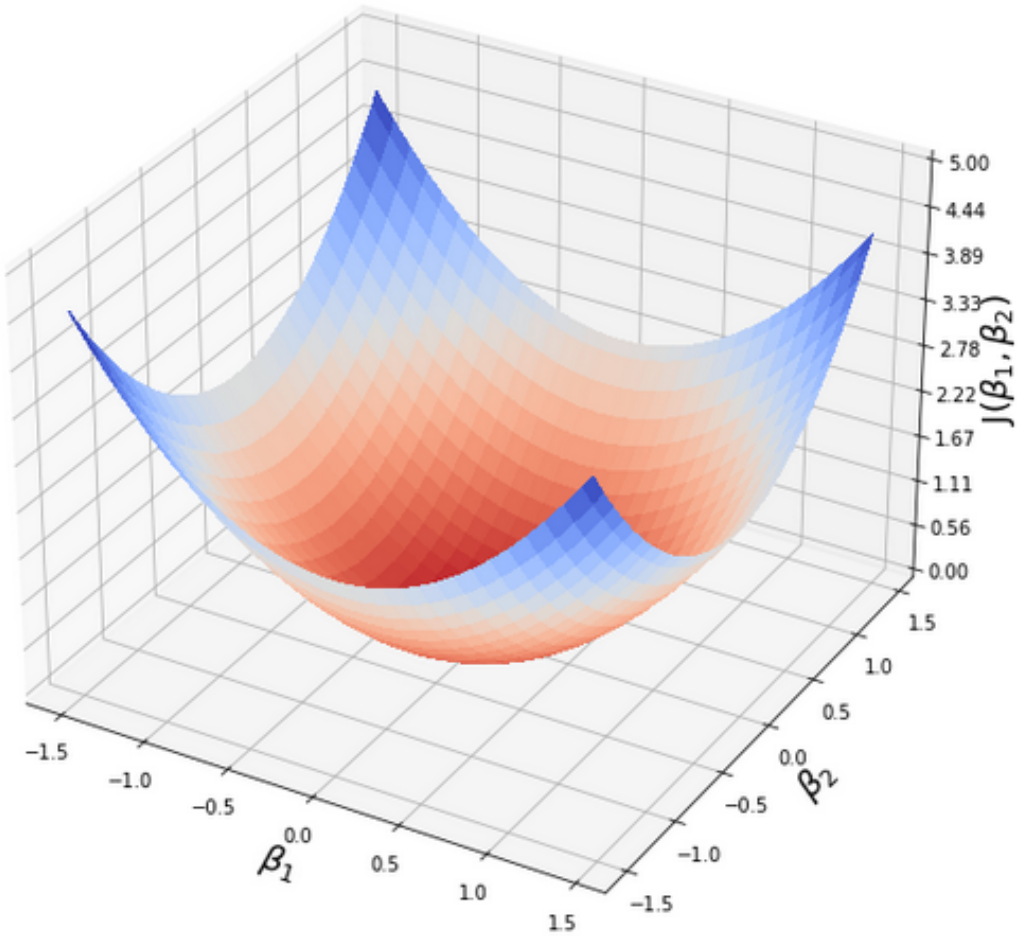
El funcion de cost (o *'objective function'*) J es lo que queremos minimizar

Por ejemplo, en regresión lineal (mínimos cuadrados ordinarios):

$$
\begin{aligned}
J(\beta_1, \beta_0) &= MSE \\
&= \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \\
&= \frac{1}{n} \sum_{i=1}^{n} (y_i - \beta_1 \mathbf{X_1}_i + \beta_0)^2
\end{aligned}
$$

¿Que forma tiene J?

J es un superficie con la mismo dimensión que el numero de parámetros, por ejemplo algo así con dos parámetros:



Solver: Gradient descent
El gradiente (G) de un superficie esta dado por

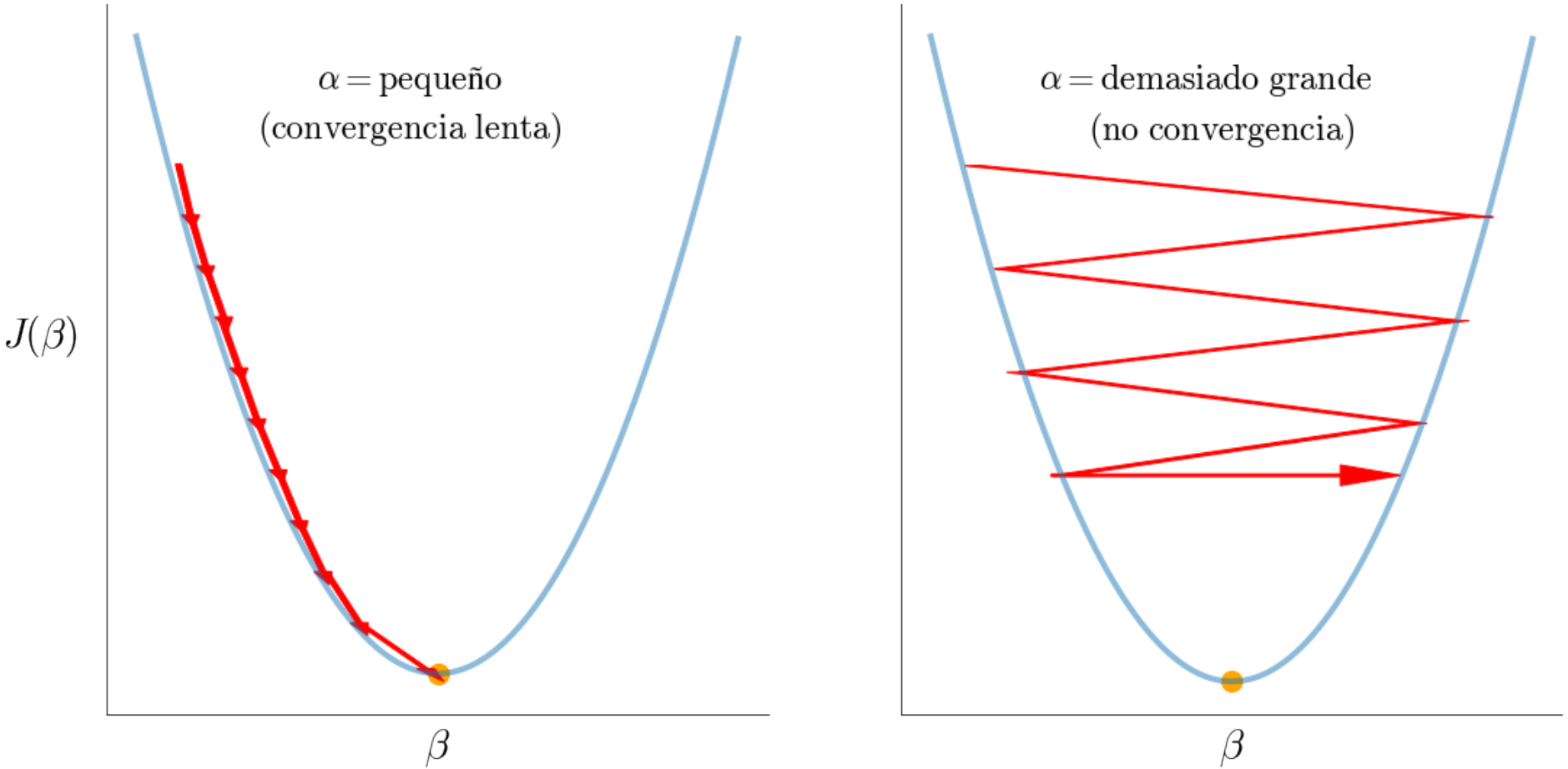
$$G = \frac{\partial J(\beta)}{\partial \beta}$$

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, global

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.

Otro solver, procedente del álgebra lineal: La *"Normal equation"*
Eso es una solución analítica

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

donde $(X^T X)$ se llama la *"normal matrix"*.

Escalado y normalización (*Scaling y 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.RobustScaler` si hay outliers
- `sklearn.preprocessing.StandardScaler` $\mu = 0, \sigma^2 = 1$

Ejemplo de uso:

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

Métricas de regresión: ¿Qué tal lo estamos haciendo?

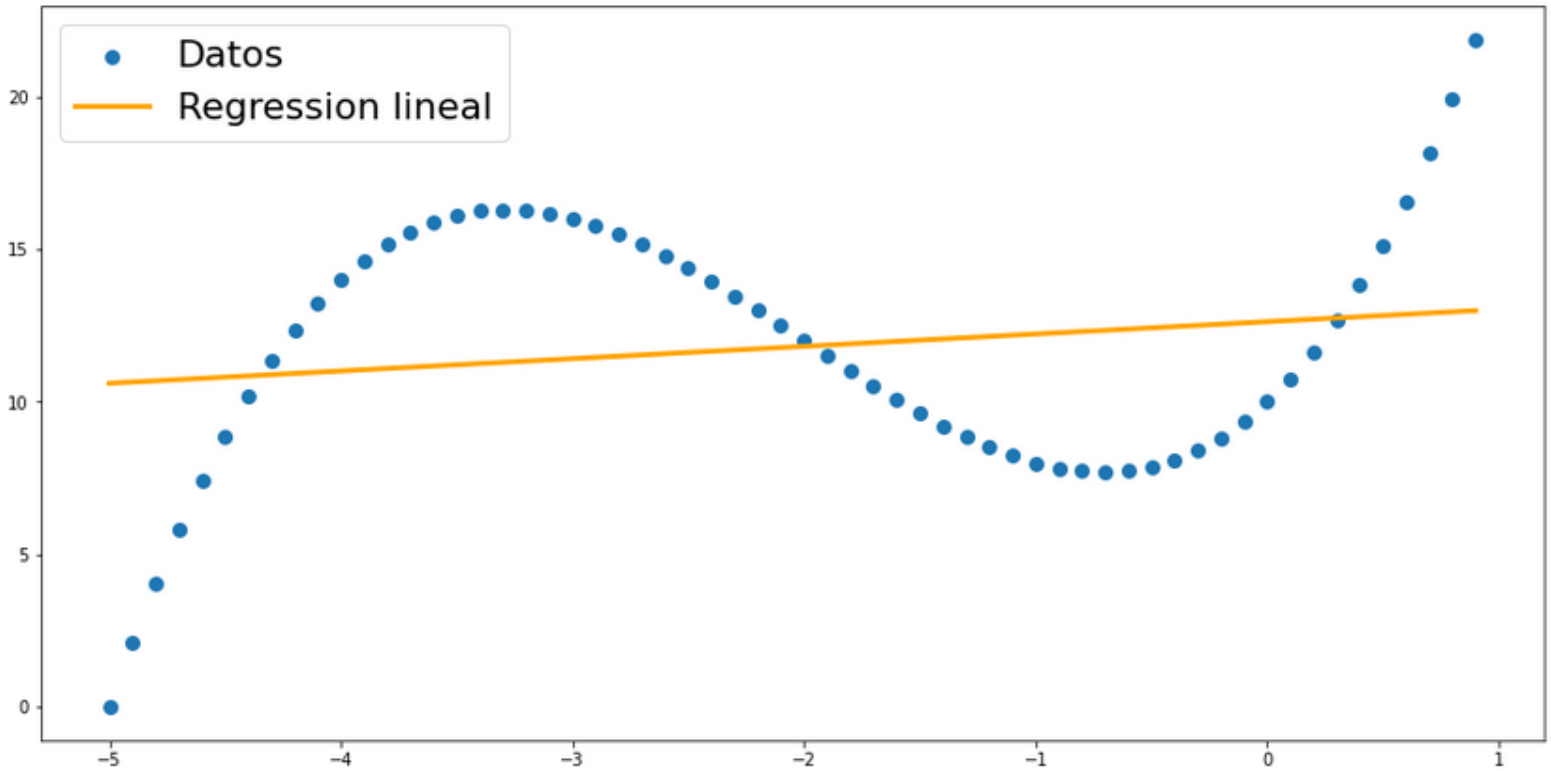
La métrica más común es el *root mean squared error* (RMSE)

`sklearn.metrics.mean_squared_error`

Regresión lineal:

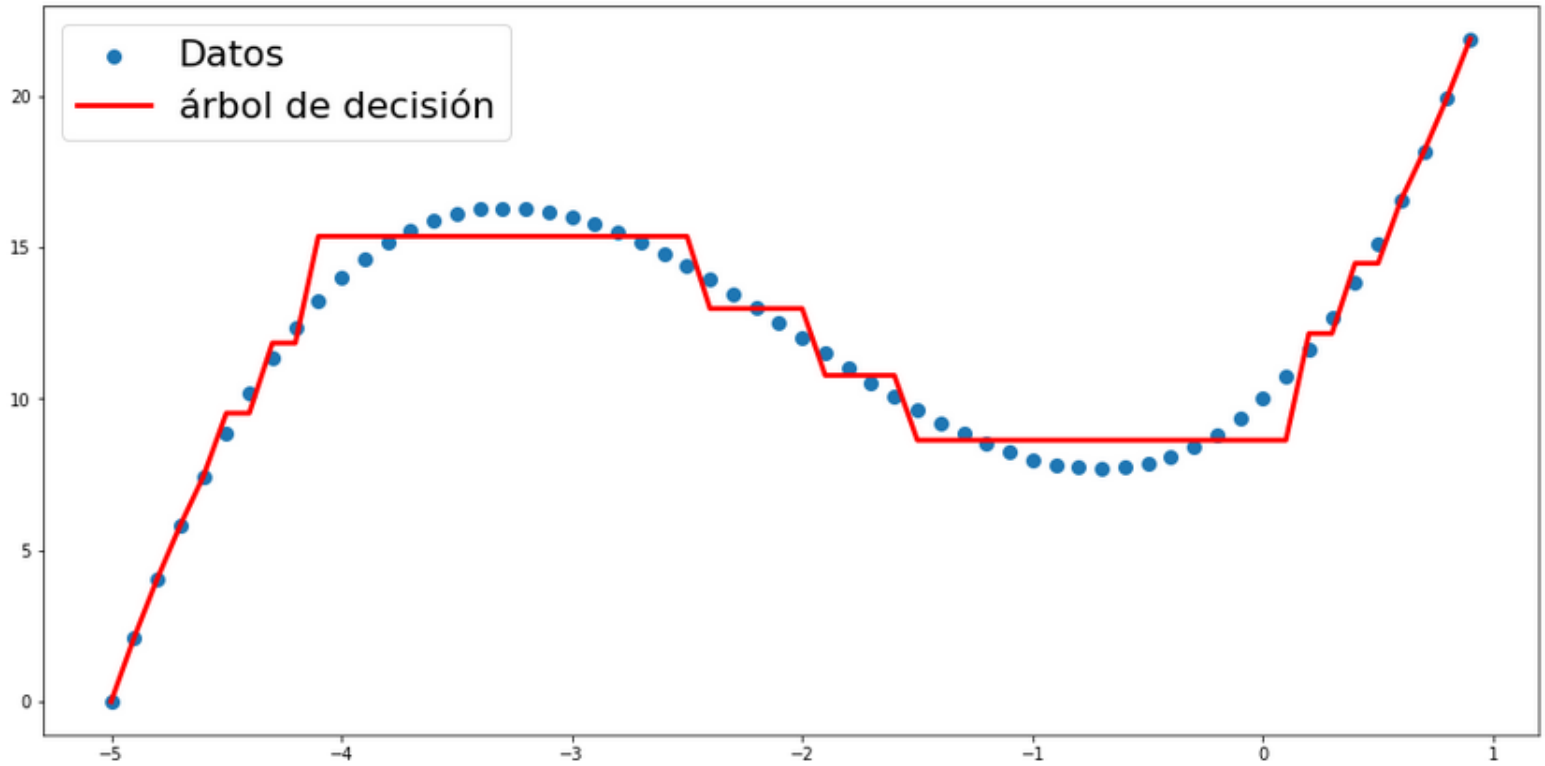
- ventajas: muy fácil de interpretar
- desventajas: solo va bien si las relaciones entre X y y son lineales

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"*).

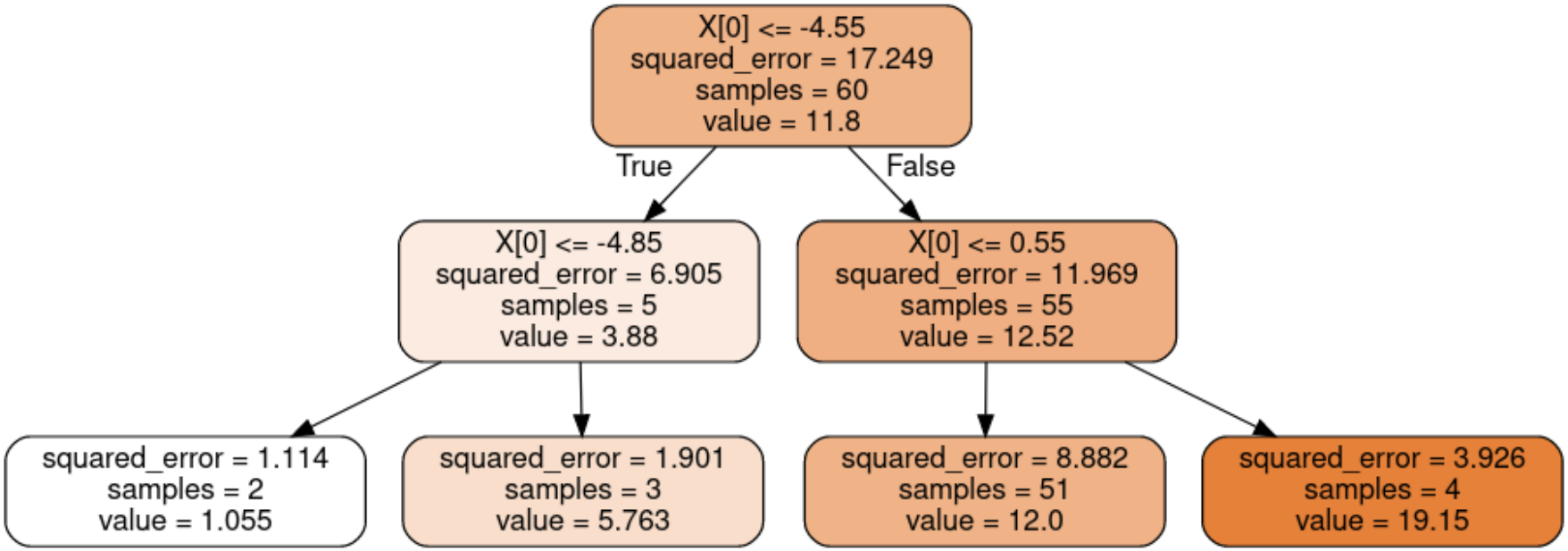
Más complejidad: Árboles de decisión:




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

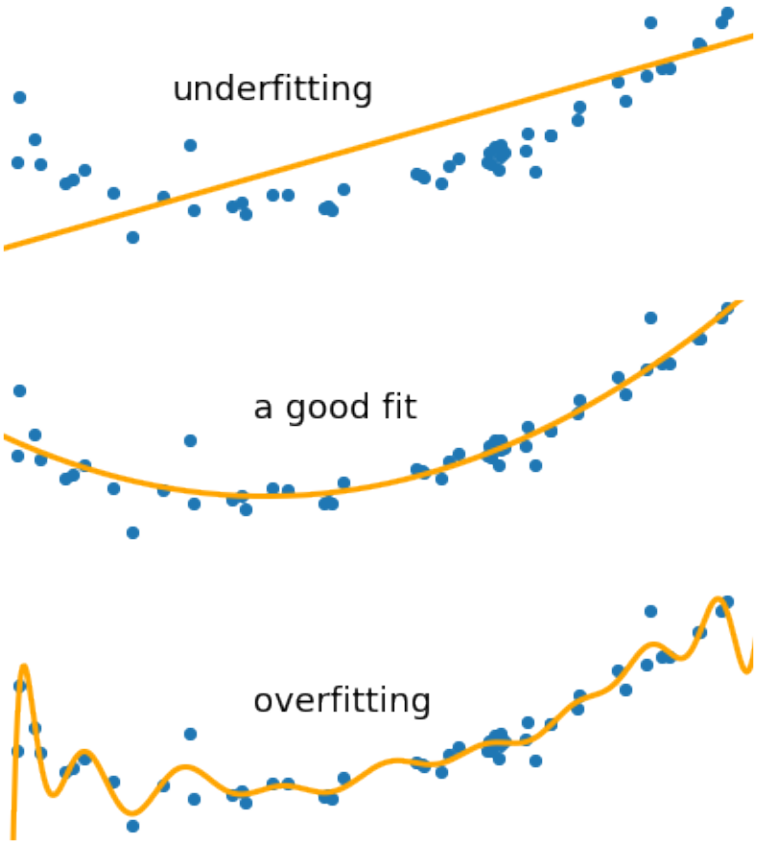
Los árboles usan *"recursive binary splitting"*, y paran de dividir cuando llegan a un numero mínimo de datos en cada 'hoja' (región de espacio)

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
El residual es el L2 (*i.e.* error^2)



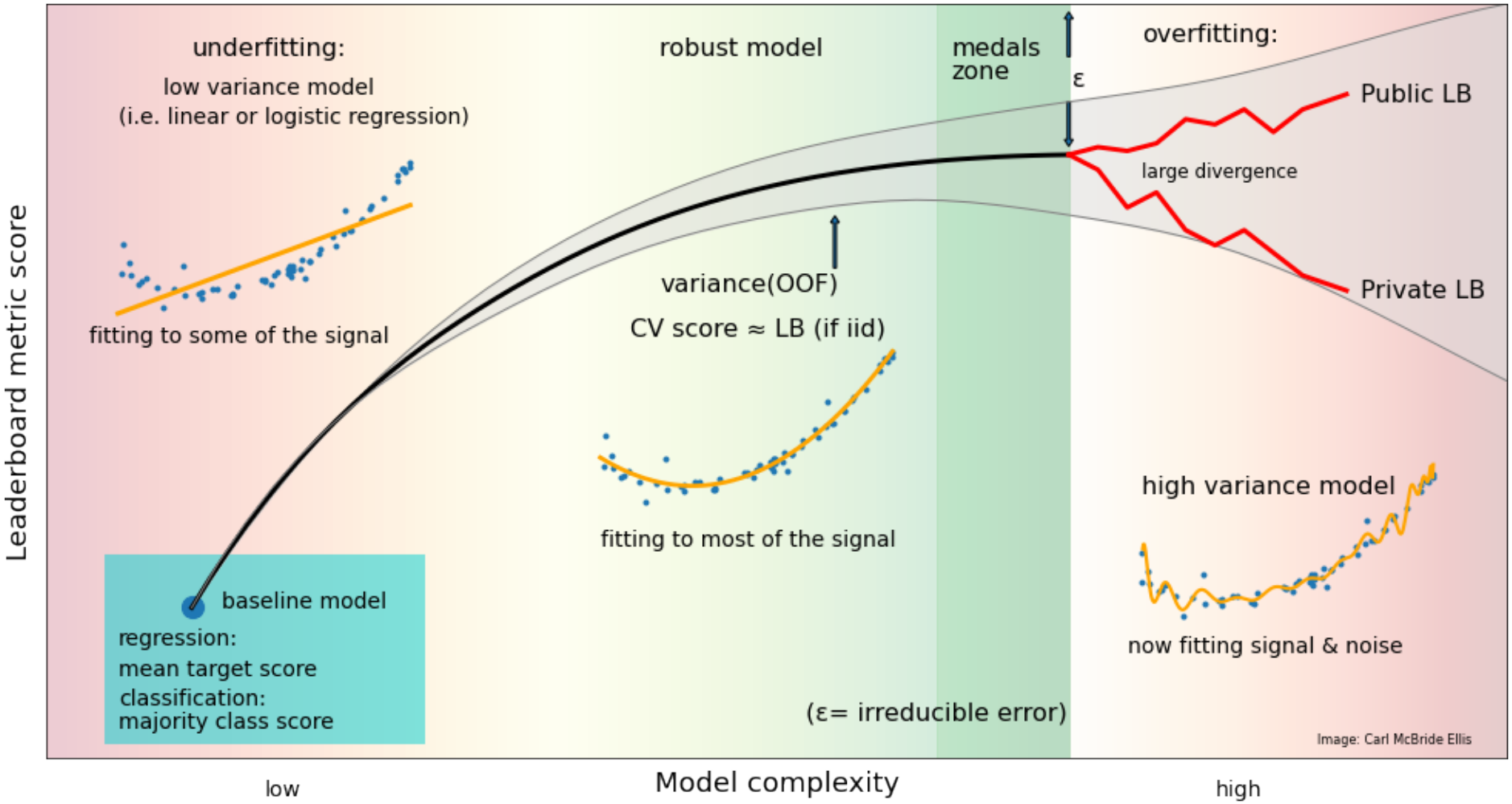
Los estimadores “no-paramétricos” que están basados en árboles de decisión, y que no usan parámetros como los β que usa regresión lineal, y por tanto no necesitan ni escalado ni normalización.

Sobreajuste (*overfitting*): regresión:



Eso pasa cuando nuestro modelo tiene demasiada complejidad.

El compromiso entre bias y varianza:



Como reducir overfitting: *shrinkage*

regularization = Cost function + penalty

- Ridge (o L2) penalty es β^2
- i.e. $J \leftarrow J + \lambda \sum_{j=1}^{p} \beta_j^2$
- LASSO (o L1) penalty es $|\beta|$
- i.e. $J \leftarrow J + \lambda \sum_{j=1}^{p} |\beta_j|$

donde λ es la *"regularization parameter"*

Jupyter notebook: Feature importance using the LASSO

Variables categóricas (*categorical features*) (i.e. strings): en general hay dos tipos

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

¿Que hacer?

- ordinal: Label Encoding: a $\rightarrow$ 1, b $\rightarrow$ 2, c $\rightarrow$ 3
- `sklearn.preprocessing.OrdinalEncoder()`
- nominal: crear un *"mask"*. Conocido como *"One Hot encoding"*, con $n-1$ columnas nuevas
- `sklearn.preprocessing.OneHotEncoder()`

Primer modelo: baseline

- $\hat{y} = \bar{y}$

Los predicciones son la media de los targets

Sesión práctica

Sesión práctica #2: Regresión:

Jupyter notebook: BIG DATA: Práctica 2: Regresión

Clasificación

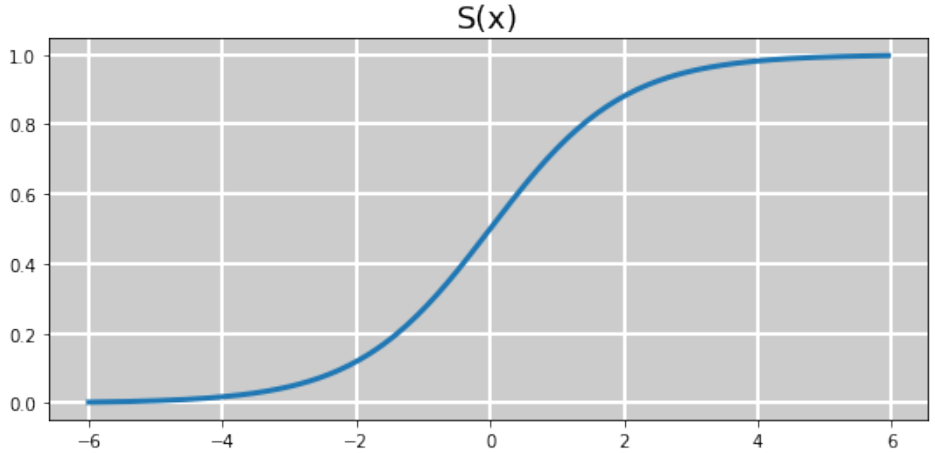
En regresión calculamos un valor real ($\hat{y}$) para cada punto (cada fila en el dataframe).
En cambio, en clasificación asignamos un *'label'* a cada punto. Cada label es un valor del set de clases
Por ejemplo en clasificación binaria los clases suelen ser 0 ó 1
Nota: No siempre queremos las clases; a veces las probabilidades son más útiles (método: `predict_proba`)
(*p.ej.* aunque las dos pertenecen al clase "1", $0.51 \neq 0.99$)

Regresión logística (*Logistic regression*):

$$p_i = \text{sigmoid}\left(\beta_0 + \beta_1 x_{i1} + \cdots + \beta_p x_{ip}\right)$$

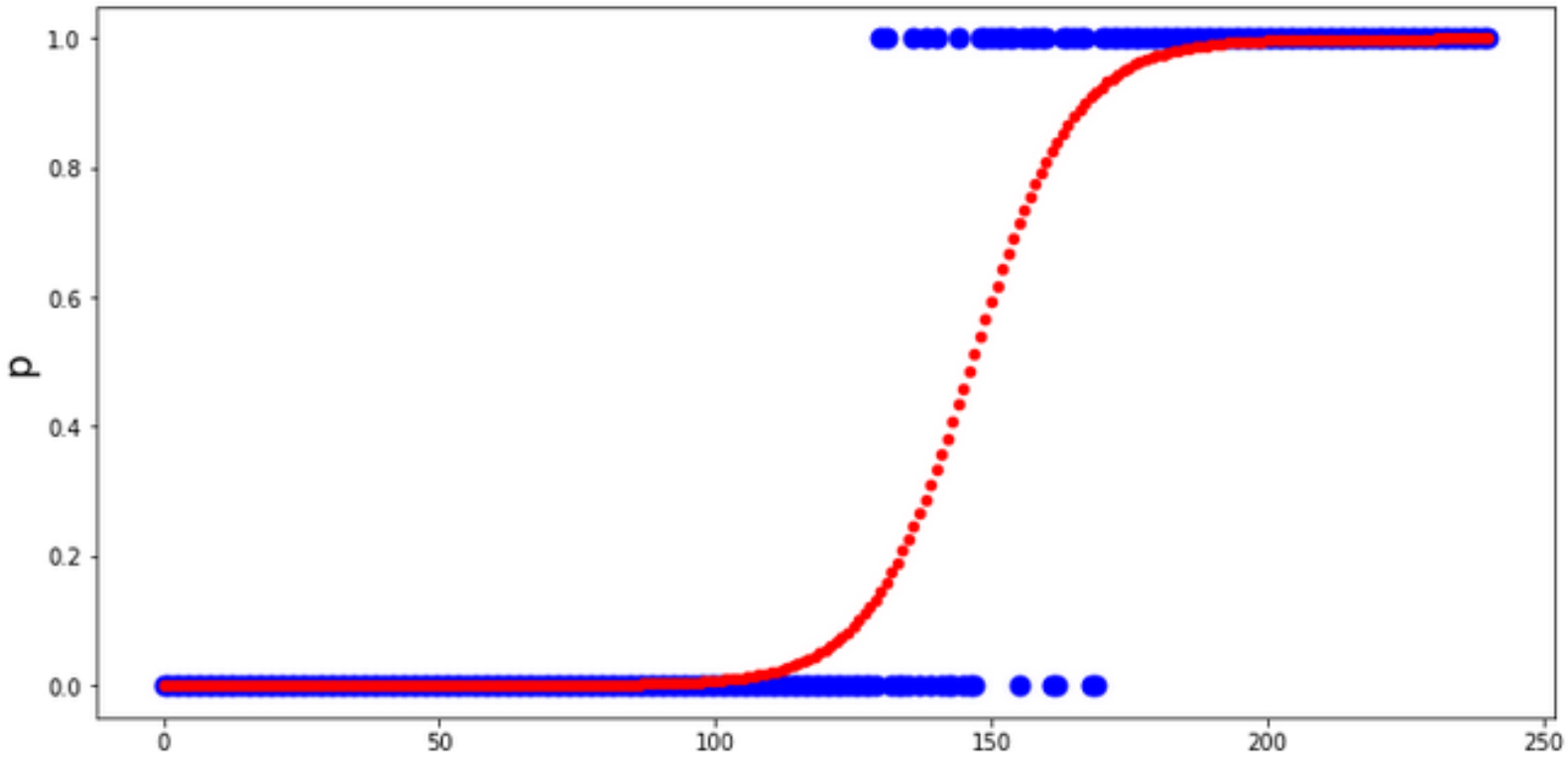
donde

$$\text{sigmoid}(x) = \frac{1}{1+e^{-(x)}}$$



(tb. se llama el función *'sigmiod'* la función *'logisitc'*)
en lugar de calcular el valor $\hat{y}$ ya estamos calculando la probabilidad, p

Regresión logística: Ejemplo:



clasificación loss function = log loss

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

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

Explicabilidad (*explainability*):

Regresión logística, como con su homologo el regresión lineal, tiene un ventaja en que sus predicciones, aunque no son necesariamente los mejores, son relativamente fáciles a entender

p.ej. ver el notebook Titanic explainability: Why me? asks Miss Doyle

Métricas de clasificación: *Accuracy score*

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

Primer modelo de clasificación: crear un baseline

- $\hat{y} = 0$

Es decir, las predicciones son la clase mayoritaria

(Nota: Si tienes un `accuracy_score` < 0.5 algo va mal)

Métricas de clasificación: Accuracy score

- ¡cuidado con datos desbalanceados!
- `from sklearn.metrics import balanced_accuracy_score`

¿Qué es un buen `accuracy_score`?

¿Es un `accuracy_score` de 1.0 es posible?

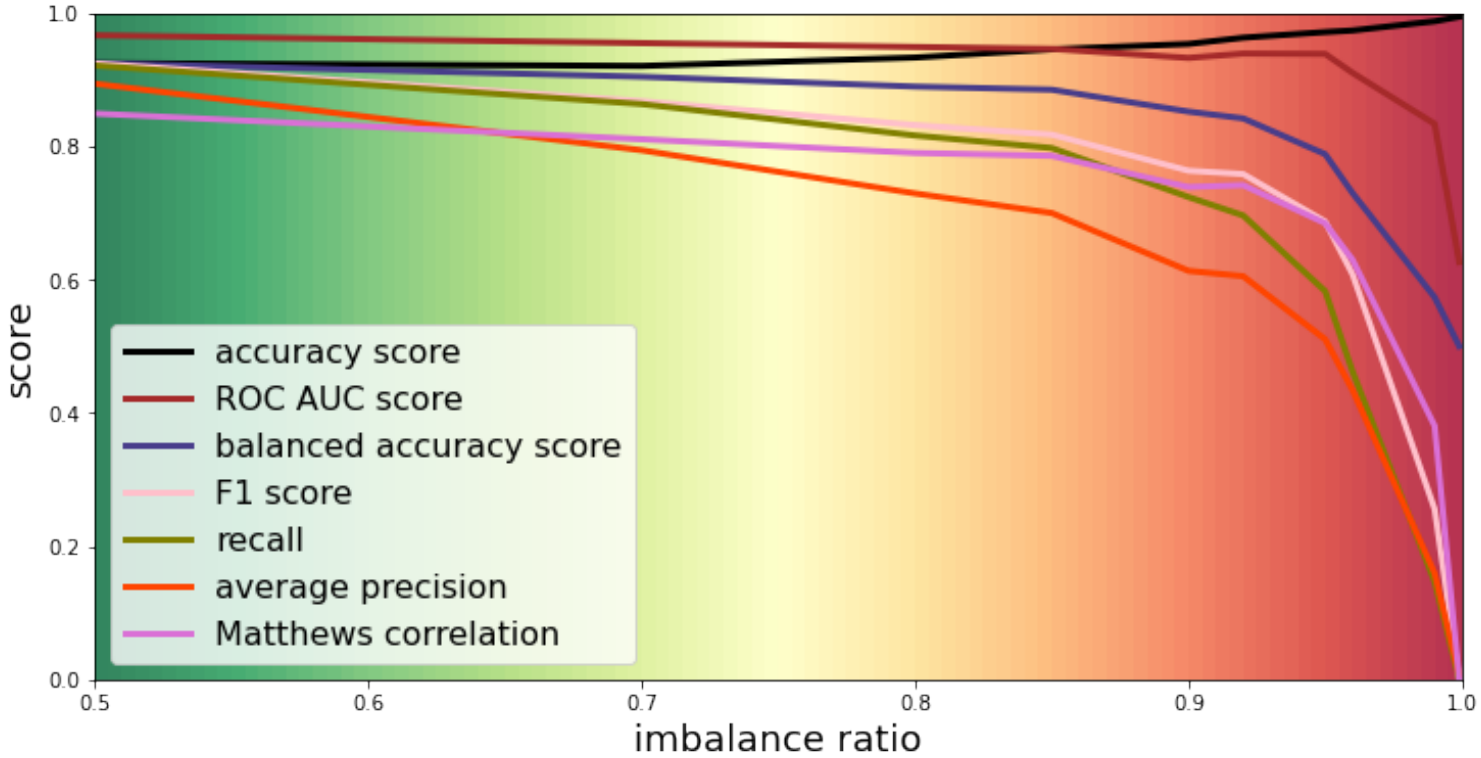
Con datos reales: No

¿Es un `accuracy_score` de 0.51 malo?

¡No necesariamente!

Notebook: El juego de "cara o cruz"

Clasificación con datos desbalanceados: métricas



(fuente: "Classification: How imbalanced is "imbalanced"?)

En más detalle: hay 4 clasificaciones posibles:

- falso positivo (FP)
- falso negativo (FN)
- verdadero positivo (TP)
- verdadero negativo (TN)

Matriz de confusión (*Confusion matrix*):

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

Equations:

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

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

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

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

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

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

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

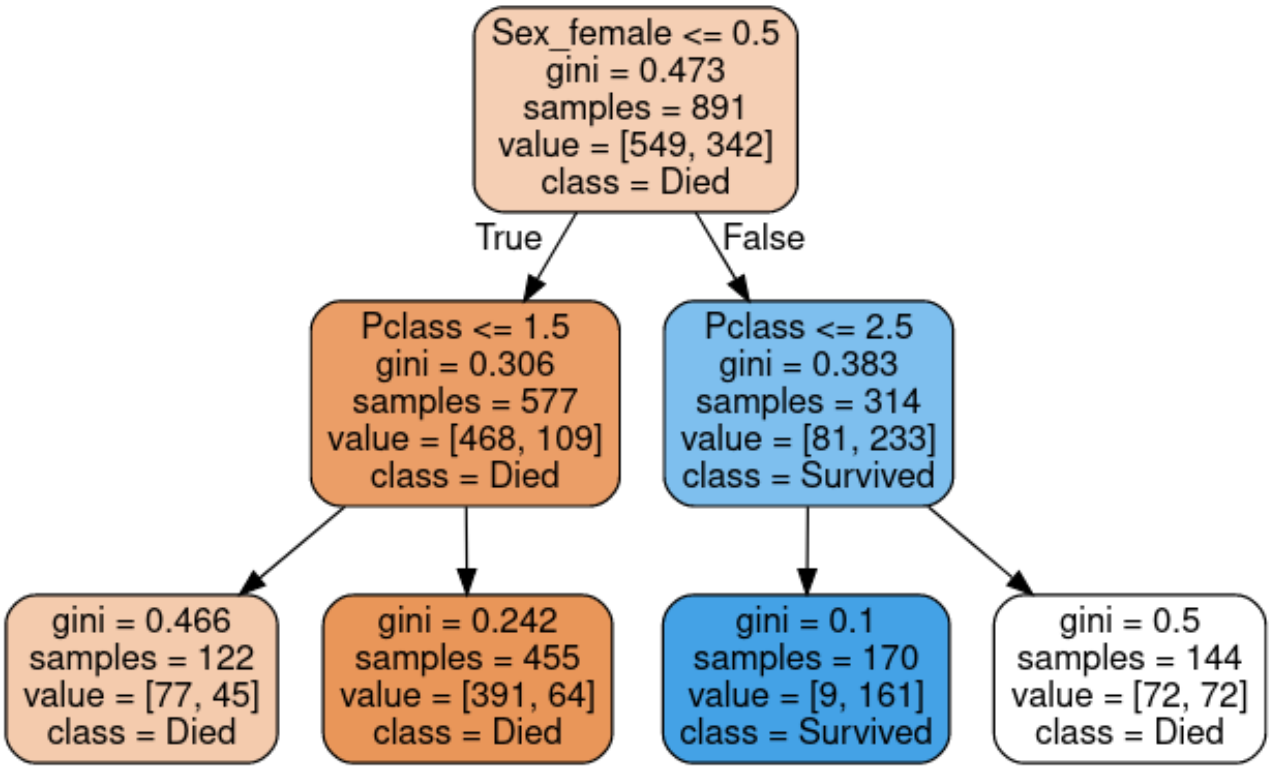
Precision y recall

- $\text{precision} = \frac{TP}{TP+FP}$
- $\text{recall} = \frac{TP}{TP+FN}$

Métricas de clasificación: F_1 score

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

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



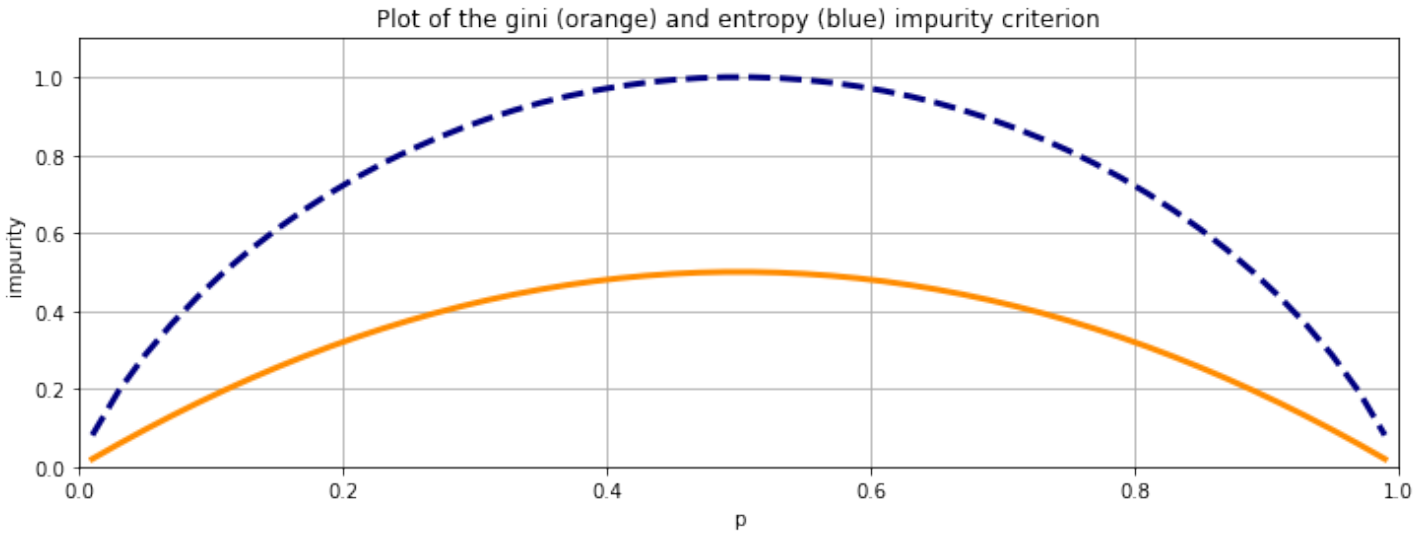
Este vez en lugar del *'squared error'* en regresión, ya se usa el *Gini impurity* (I_G) para splitting

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

donde p_i es la fracción de muestras en clases i.
Cuanto más puro sea el nodo, menor será el valor de Gini.
En el caso de clasificación binaria:

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

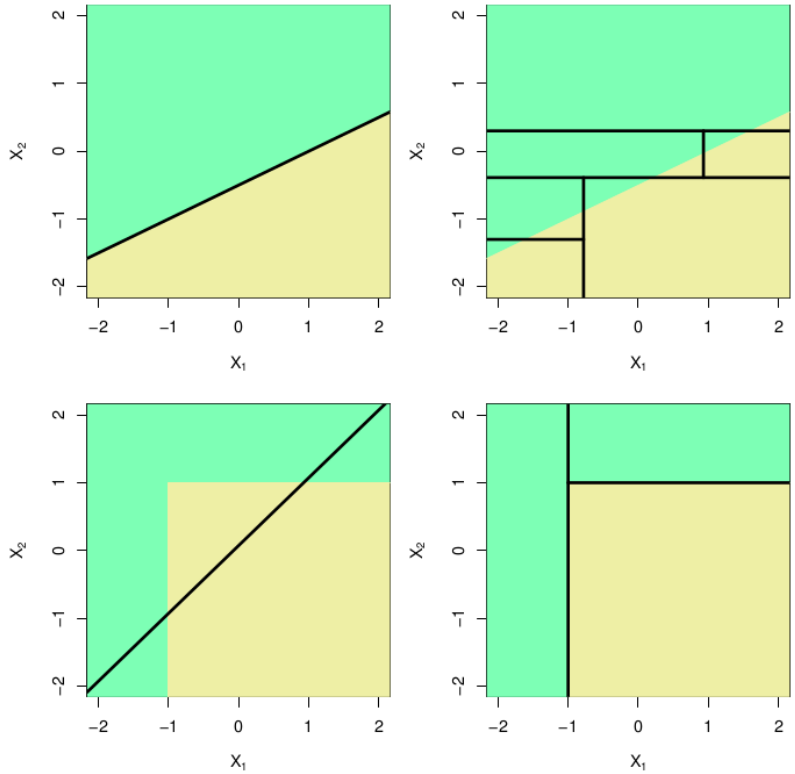
¿Que pinta tiene la Gini (y la entropía)?



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...)

Ajuste lineal Vs árboles:



Si la *"decision boundary"* es lineal, los árboles no van a dar el mejor resultado

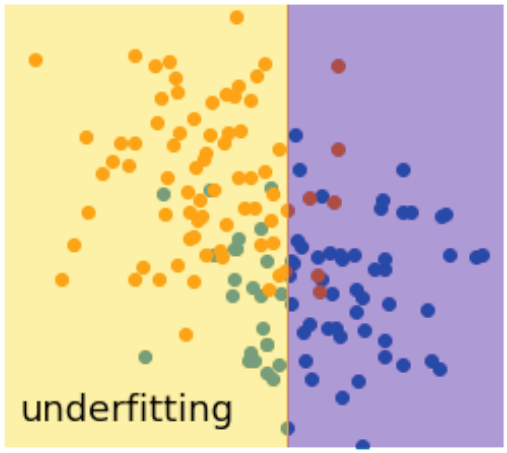
Bagging ensembe: El *"Random Forest"*

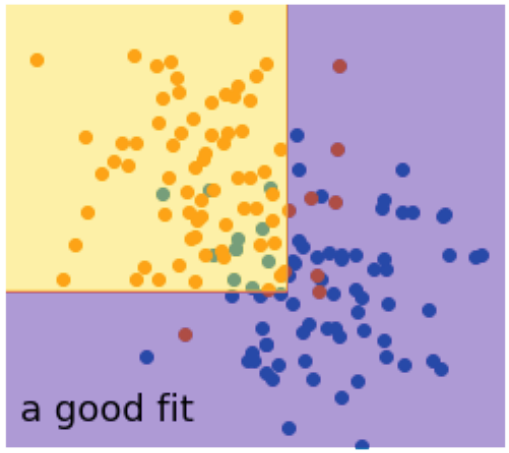
Usa el "boostrap" para fabricar nuevas training datasets (del mismo tamaño) y unos $\sqrt{\texttt{n\_features}}$ en cada árbol:

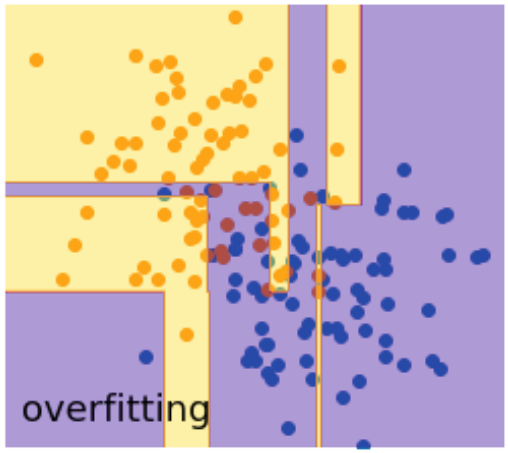
- Regresión: el resultado final es el promedio de los árboles
- Clasificación: el resultado final es el *"majority vote"*

El *bootstrap* es un dataset creado con muestras aleatorias con re-emplazo, y contiene $\approx 63\%$ del dataset `train`.

Sobreajuste (overfitting): clasificación:







Sesión práctica

Sesión práctica #3: Clasificación:

Jupyter notebook: BIG DATA: Práctica 3: Clasificación

Hyperparameters vs. parameters

Hemos visto

- α de gradient descent,
- λ de regularization,
- el grado del polinomio
- la profundidad del árbol
- *etc*

estos son "Hyperparameters"

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

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

Ajustes: Flujo de trabajo

Creando un punto de referencia (*baseline*):
Usa estimadores que no sobre-ajustan, es decir con muy poca varianza p.ej. LR o LR siendo simples, no son propensos a sobre-ajustan, y son computacionalmente baratos, así que son buen 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
- LightGBM muy rápido aunque usa CPU
- CatBoost bueno con datos tipo 'categorical'

Nota: Estos usan 'boosting', y no 'bagging' como el Random Forest

Notebook: Ejemplo de como hacer regresión con XGBoost

Boosting es secuencial

Ajustar el nuevo (menguado) árbol al error del último tree (parámetro: `learning_rate`)

Al contrario que bagging, boosting sí puede sobre-fitear.

Árboles de decisión: Como reducir overfitting: *"jardinería"*

Con modelos no-paramétricos, como los árboles de decisión y sus extensiones el `RandomForest` y `XGBoost` *etc* se puede reducir overfitting al reducir los hyperparameters de `max_depth` y `min_samples_leaf`

Árboles de decisión: hyperparameters

Hay muchísimas hyperparameters. Estos son las más importantes:

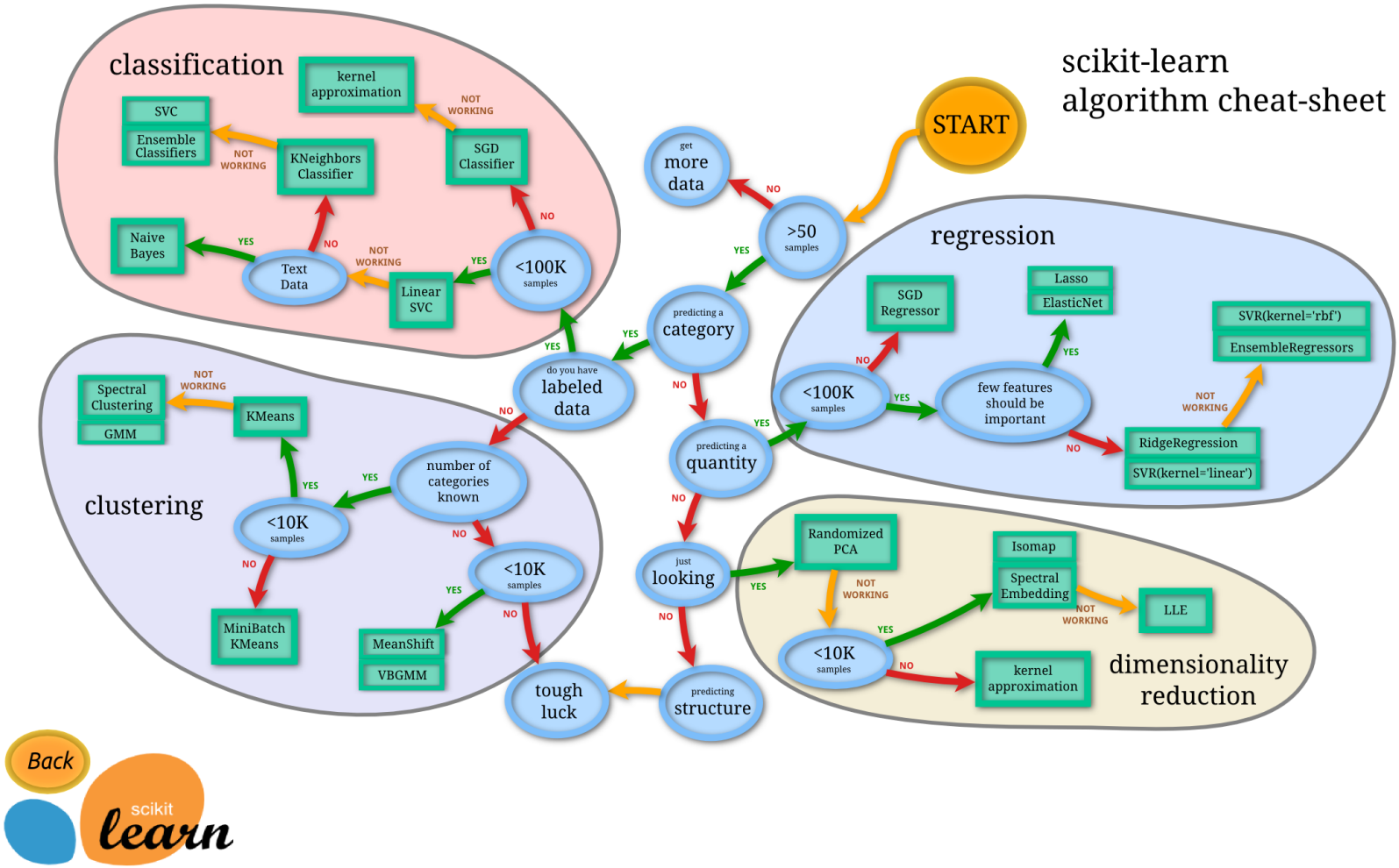
- `DecisionTree`: `max_depth`
- `RandomForest`: `max_depth`, `n_estimators=100`
- `XGBoost`: `max_depth=6`, `n_estimators=100`, `learning_rate=0.3`

Todos: `min_samples_leaf`

Mi consejo de donde empezar tu *hyperparameter optimization*:

- `max_depth = 4`
- `n_estimators = 100`
- `learning_rate = 0.1` (o incluso menos; `0.05`)
- `min_samples_leaf = 10`

Cómo elegir tu estimador scikit-learn:



(fuente: Scikit-learn: Choosing the right estimator)

Feature engineering

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*

Feature engineering es quizás la parte más artesanal de ML

Técnicas:

- `permutation_importance`
- recursive feature elimination: quitar el feature menos predictivo
- feature importance plots: visualización de importancias
- 'recursive' o 'forward' feature addition (`f_regression`)
- fabricar nuevas features, *p.ej.* a, b, $a \times b$, a^2 *etc.*
- transformar features, *p.ej.* $\log(x+1)$
- ...
- Peor caso: ¡obtener más datos!

Permutation Importance:

- 1. Reordenar aleatoriamente una sola columna (feature)
- 2. Re-calcular el metric
- 3. Su importancia es proporcional a cuanto 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

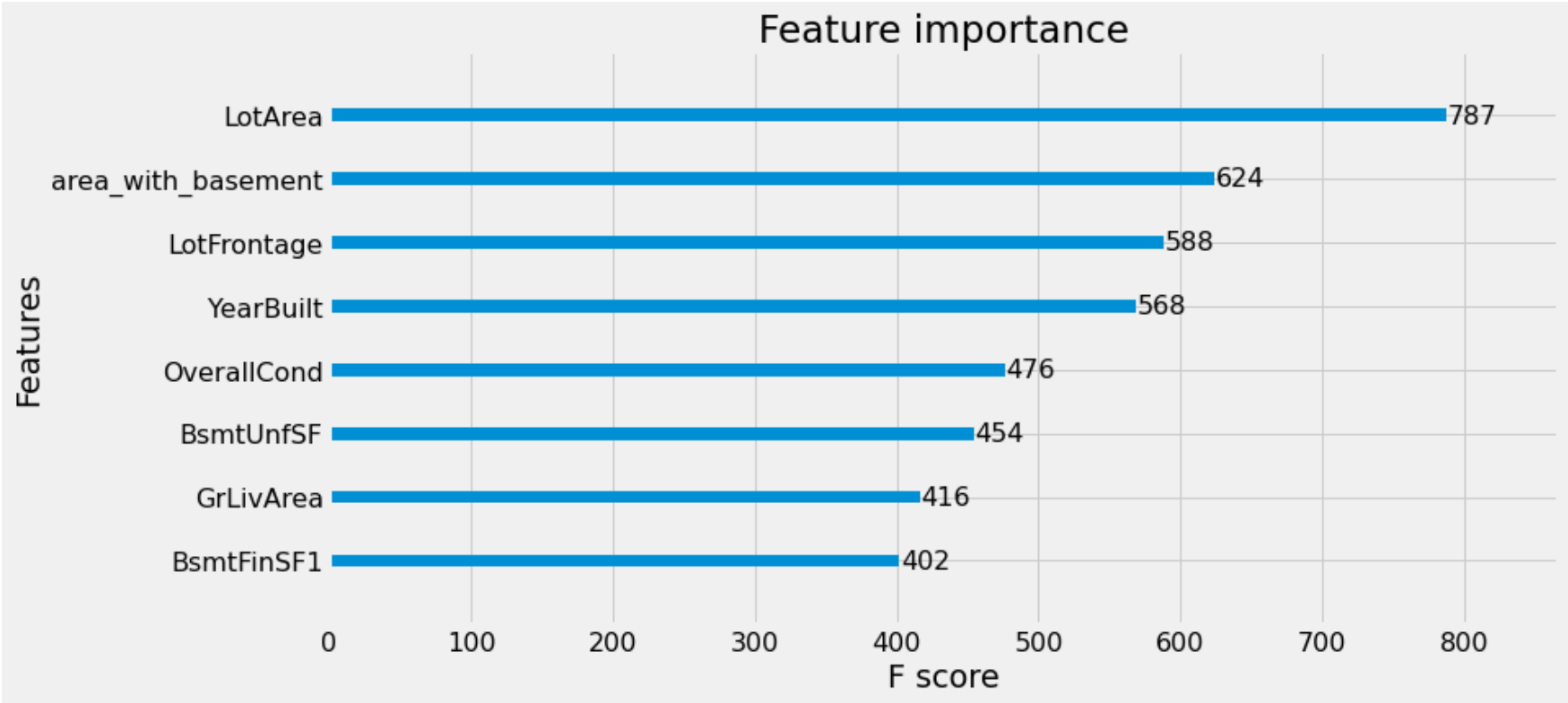
Notebook: "House Prices: Permutation Importance example"

Recursive feature elimination (RFE)

- Eligir el numero de features que quieres (n)
- 1. Calcular la importancia de cada feature
- 2. Eliminar el feature menos importante
- 3. Repetir hasta solo queda n features

Notebook: "Recursive Feature Elimination (RFE) example"

Feature importance plot; Ejemplo:



Notebook: "An introduction to XGBoost regression"

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

Más detalles: "Kaggle Ensembling Guide"

Recursos

Libros:

- "Introduction to Machine Learning with Python" Müller y Guido
- "The Elements of Statistical Learning" Hastie, Tibshirani y Friedman
- "The Hundred-Page Machine Learning Book" Burkov

Blogs:

- Machine Learning Mastery
- Analytics Vidhya
- Towards Data Science

Foros de Q&A:

Informática:

- Stack Overflow

Estadística:

- Cross Validated

BIG DATA Notebooks

"Learning By Doing"

- BIG DATA: Enlace al concurso
- BIG DATA: Media aritmética y varianza
- BIG DATA: Práctica 1: Facebook dataset EDA (plantilla)
- BIG DATA: Práctica 2: Regresión
- BIG DATA: El juego de "cara o cruz"
- BIG DATA: Práctica 3: Clasificación
- BIG DATA: Regresión lineal (plantilla para el concurso)