

# UNIDAD 2

## SISTEMAS DE APRENDIZAJE AUTOMÁTICO

IES SERRA PERENXISA (Torrent) Valencia

# APRENDIZAJE SUPERVISADO I

Profesor: José Rosa Rodríguez  
Curso 2023-2024

## OBJETIVOS:

- Aprendizaje supervisado.
- Los modelos lineales.
- Predecir valores continuos: regresión numérica.
  - Regresión lineal.
  - Regresión lineal restringida.
  - Regresión polinómica.
  - K-NN
- Clasificación binaria y multiclasa:
  - perceptrón
  - adaline,
  - regresión logística,
  - Softmax. KNN
- Algoritmos y técnicas de mejora de datos.
- Algoritmos y métodos de ingeniería de características.
- Métricas y técnicas medición de desempeño o rendimiento.
- Técnicas de mejora de hiperparámetros: k-fold cross validation y GridSearch.
- Métodos ensamble: bagging, boosting, random forest, stacking.

## RECURSOS y BIBLIOGRAFÍA:

- Repaso de numpy: [appendix\\_f\\_numpy-intro.pdf \(sebastianraschka.com\)](#)
- Guía de pandas: [10 minutes to pandas — pandas 2.2.2 documentation \(pydata.org\)](#)
- Guía de matplotlib: [Quick start guide — Matplotlib 3.9.0 documentation](#)
- Vídeos de codificandobits.com
- Hands of Machine Learning with scikit-learn, keras and TensorFlow, Ed. O'Reilly.

## Sumario

1. APRENDIZAJE SUPERVISADO.....	5
2. EL PERCEPTRÓN.....	6
3. ADALINE Y LA FUNCIÓN DE COSTE.....	10
4. INGENIERÍA DE CARACTERÍSTICAS.....	13
4.1. DATOS AUSENTES.....	13
4.2. DATOS CATEGÓRICOS.....	14
4.3. DETECCIÓN Y ELIMINACIÓN DE OUTLIERS.....	14
4.4. HOMOGENEIZAR VALORES.....	18
4.5. CONSTRUIR O GENERAR CARACTERÍSTICAS.....	19
4.6. ANÁLISIS EXPLORATORIO DE DATOS: EDA.....	19
5. ALGORITMOS DE REGRESIÓN LINEAL.....	24
5.1. MEDIR EL ERROR NUMÉRICO.....	28
5.2. DESCENSO POR GRADIENTE.....	30
5.3. CANTIDAD DE EJEMPLOS Y RATIO DE CONVERGENCIA.....	33
5.4. MEJORAS PARA COSTES NO CONVEXOS.....	36
6. PROBLEMAS Y MEJORAS.....	42
6.0. PIPELINES DE OPERACIONES.....	42
6.0.1. PIPELINES DE OPERACIONES.....	42
6.0.2. TRANSFORMAR DATOS DE COLUMNAS CON ColumnTransformer.....	43
6.0.3. TRANSFORMAR LA COLUMNA TARGET EN REGRESIONES.....	45
6.0.4. COMPONER DIFERENTES ESPACIOS.....	45
6.1. CURVAS DE ENTRENAMIENTO/APRENDIZAJE.....	46
6.2. MODELOS REGULARIZADOS.....	53
6.3. ANÁLISIS CON ESTADÍSTICA Y GRÁFICOS.....	57
6.3.1. SIGNIFICANCIA DEL MODELO Y LAS PREDICTORAS.....	58
6.3.2. CONDICIONES PARA LA REGRESIÓN LINEAL.....	60
7. ALGORITMOS LINEALES GENERALIZADOS.....	70
7.1. REGRESIÓN POLINÓMICA.....	71
7.2. REGRESIÓN LOGÍSTICA.....	74
7.2.1. SOLVERS DE SCIKIT-LEARN.....	77
7.3. REGRESIÓN SOFTMAX.....	79
8. ALGORITMO K-NEAREST NEIGHBORS (KNN).....	82
9. EVALUAR MODELOS PARA CLASIFICACIÓN.....	87
9.1 LA MATRIZ DE CONFUSIÓN Y MÉTRICAS.....	88
9.2. CURVAS ROC.....	92
10. SELECCIÓN DE MODELOS.....	94
10.1. MÉTODOS HOLDOUT Y RANDOM SUBSAMPLING.....	94
10.2. VALIDACIÓN CRUZADA (CROSS-VALIDATION).....	95
10.3. USANDO TEST ESTADÍSTICOS.....	98
10.4. PROCEDIMIENTO GRID-SEARCH.....	100
10.5. TEST DE PERMUTACIONES ALEATORIAS.....	102
10.6. MÉTODOS BOOTSTRAP.....	103
11. MEJORAR LA EFICIENCIA DE LOS MODELOS.....	105
11.1. BAGGING.....	105
11.2. BOOSTING.....	106
11.3. RANDOM FOREST.....	111
11.4. MODELOS ENSEMBLE POR VOTACIÓN.....	112
11.5. STACKING.....	114
12. EJERCICIOS.....	115

En esta unidad comenzaremos a trabajar con los primeros algoritmos de aprendizaje que han dado lugar al **Machine Learning** y al **Deep Learning**. Luego continuaremos con otros algoritmos usados en aprendizaje supervisado para predecir valores numéricos y para clasificar. Primero nos centramos en los algoritmos más simples y dejaremos para una próxima unidad algunos otros. También veremos como medir lo bien o mal que funcionan, como mejorar su funcionamiento corrigiendo algunos de sus problemas y aprenderemos la mecánica de trabajo.

Al principio programaremos algo de código en Python desde cero, pero también comenzaremos a utilizar y a conocer funcionalidades del paquete **scikit-learn** de Python, además de **numpy** y **matplotlib** para visualizar gráficos y otros como **statsmodel** o **scipy**.

## 1. APRENDIZAJE SUPERVISADO

Para definir el modelo los algoritmos de aprendizaje usarán datos estructurados formados por n datos (filas) que tienen p columnas o características predictoras o independientes y una columna etiqueta o target. La construcción y uso de un modelo de regresión<sup>1</sup> con algoritmos de aprendizaje supervisado aparece en esta figura 1.

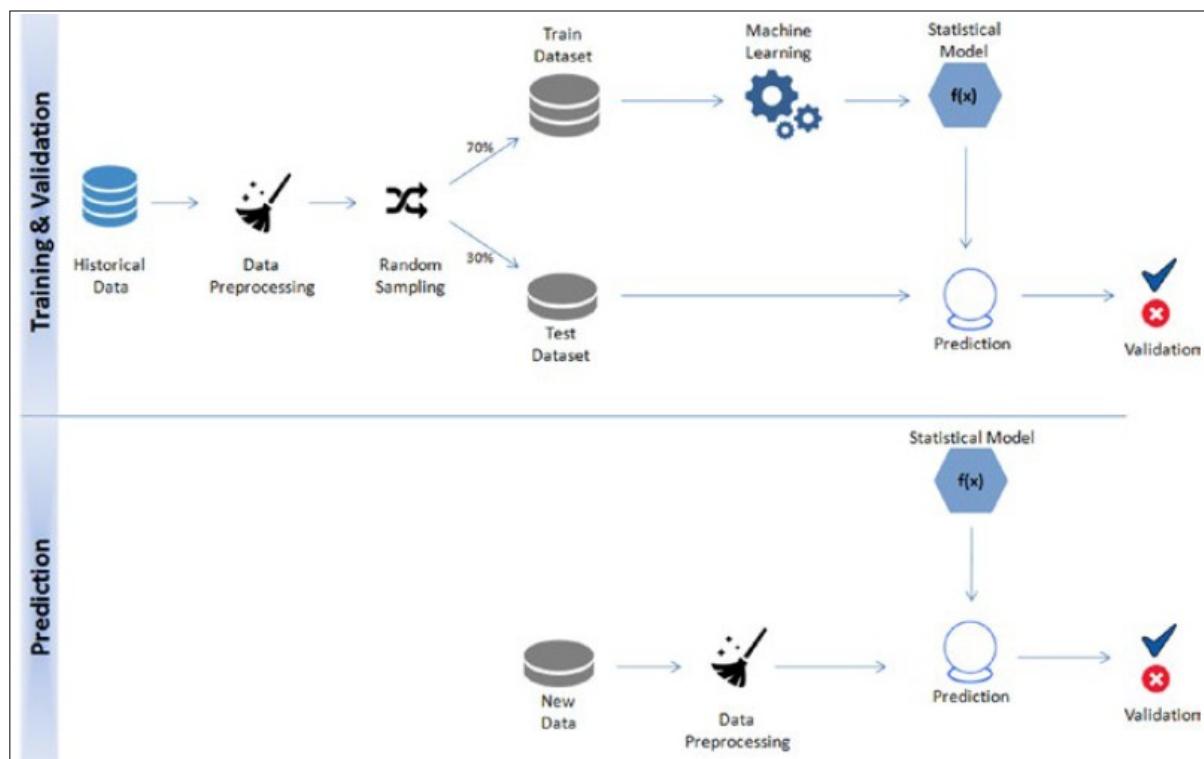


Figura 1. Flujo del proceso de aprendizaje supervisado.

Primero es necesario entrenar y validar el modelo utilizando algoritmos y técnicas de machine learning con datos históricos. Se estudian y preparan los datos para optimizar el trabajo de los algoritmos de aprendizaje, se dividen de alguna forma en datos de aprendizaje y datos de test, se entrena al modelo y luego se valida y se vuelve a entrenar hasta que la validación alcanza el nivel de buen funcionamiento que necesitemos. Una vez definido el modelo, se utiliza para trabajar con futuros valores en la fase de predicción. En todos estos procesos intervienen algoritmos, incluso para decidir el modelo a utilizar o ajustar los hiperparámetros del que hayamos escogido.

### FRONTERAS DE DECISIÓN Y REGRESIÓN LINEAL

Una frontera de decisión es una división que separa una zona en al menos dos partes. Estas diferentes zonas definidas por esas fronteras nos permiten clasificar ejemplos en diferentes tipos de elementos o

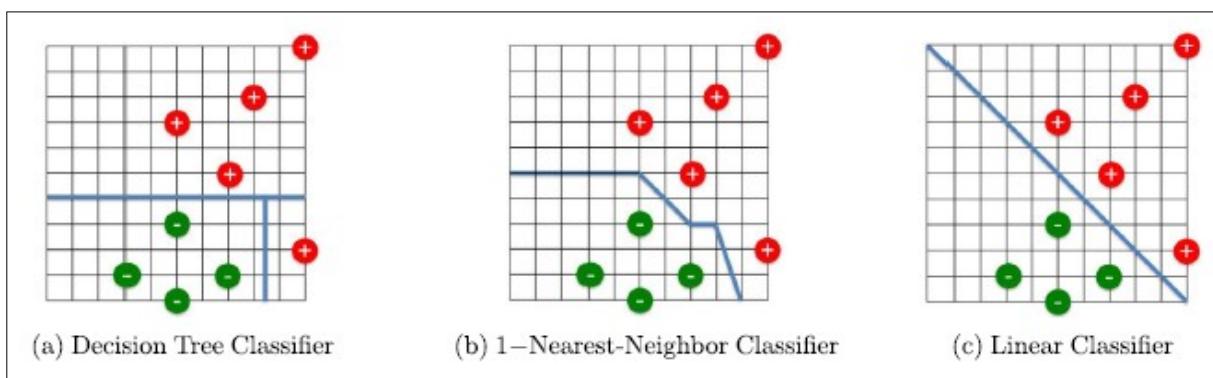
<sup>1</sup> El término **regresión** proviene de Francis Galton, un estadístico del siglo XIX que estudiaba la relación entre la altura de padres e hijos y observó que los hijos de padres altos tenían una altura menor que se acercaba a la media (regresaba). Actualmente se usa para describir el objetivo de predecir un valor numérico o una clase (clasificar).

clases. Por ejemplo: un mail es spam o no; un cliente que ha solicitado un préstamo es solvente o no; un tumor es maligno o no, etc.

Hay algoritmos que son capaces de definir fronteras de decisión muy complejas como los árboles de decisión **CART** que acaban definiendo estructuras poligonales, o un algoritmo clasificador como **k-nearest-neighbor (K-NN)** que define una frontera de decisión *hiperpoligonal*. ¿Y una línea recta? En la figura 2, el gráfico de la izquierda muestra como un **CART** podría definir estas fronteras. En el gráfico del centro, aparece la frontera que crearía el algoritmo K-NN. Y el gráfico de la derecha muestra como una línea recta podría también separar los mismos ejemplos en dos clases distintas.

Además, una frontera lineal podría tener ventajas adicionales como un cálculo más eficiente para entrenar al clasificador, más capacidad de generalización (de trabajar bien con datos nuevos o desconocidos) y más fácil de interpretar.

Comenzamos con algoritmos que definen modelos lineales que realizan regresión y clasificación usando líneas. Y comenzaremos por los primeros algoritmos que se utilizaron en Machine Learning.



*Figura 2. Fronteras de decisión creadas por diferentes tipos de algoritmos.*

## 2. EL PERCEPTRÓN.

Antes de entrar en detalles sobre el perceptrón explicaremos los orígenes del machine learning. Intentando comprender como trabaja el cerebro humano para diseñar inteligencias artificiales, **Warren McCulloch** y **Walter Pitts** publicaron en 1943 el primer concepto simplificado de una célula del cerebro que se llamó **neurona de McCulloch-Pitts (MCP)**.

Las neuronas biológicas están interconectadas en el cerebro y en los nervios y se ocupan de la transmisión y procesamiento de señales electroquímicas. Cada neurona crea muchas conexiones con muchísimas otras neuronas.



*Figura 3. Fotografía de neuronas biológicas.*

McCulloch y Pitts describieron una neurona como una simple puerta lógica con salidas binarias. Muchas señales entran en la neurona a través de sus dendritas, se integran en el interior de la neurona y se suman de forma que si la suma sobrepasa cierto umbral, se genera una señal de respuesta que recorre su axón.

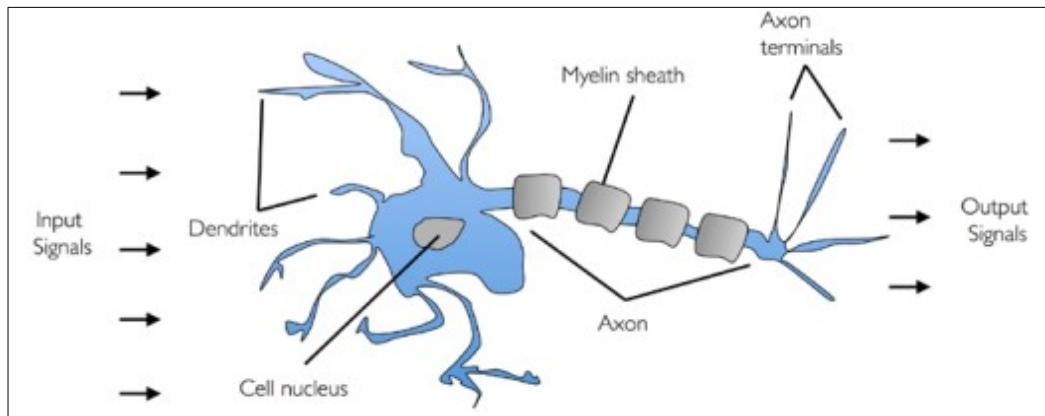


Figura 4. Esquema de una neurona biológica.

Unos años más tarde, en 1957, **Frank Rosenblatt** publica la primera regla de aprendizaje automática llamada **perceptrón**, basada en el modelo de neurona MCP. En la regla del *perceptrón*, Rosenblatt propone un algoritmo que puede aprender automáticamente los valores óptimos de unos pesos para que la neurona tome cierta decisión (en cuyo caso genera una salida). En el contexto del aprendizaje supervisado aplicado a una tarea de clasificación, el *perceptrón* hace aprender a la neurona como clasificar un dato, es decir, decidir a cuál de 2 posibles clases pertenece.

Formalmente, se puede describir una neurona artificial de tipo **perceptrón** en el contexto de una tarea de **clasificación binaria**, cuando la neurona debe decidir a cuál de 2 clases pertenece un ejemplo que se le presenta. Cada una de estas clases se representan como **+1 (clase positiva)** y **-1 (clase negativa)**.

A la neurona entran los datos del  $i$ -ésimo ejemplo  $\mathbf{x}^{(i)}$  que tiene  $m$  datos numéricos denominados  $x_j$  ( $\mathbf{x}^{(i)}$  es un vector de  $m$  números). La neurona almacena un vector  $\mathbf{w}$  de  $m$  valores numéricos llamados pesos. **Estos pesos representan lo que "sabe" la neurona**. La neurona multiplica cada dato del ejemplo de entrada  $x_j^{(i)}$  por uno de sus pesos  $w_j$  y suma estos productos dando como resultado un valor numérico  $z$ . Esta operación se puede describir como un **producto dot** de los vectores  $\mathbf{w}^T \mathbf{x}$  y genera el valor escalar  $z = w_1 x_1 + w_2 x_2 + \dots + w_m x_m$ . Luego aplica una función de activación  $\Phi(z)$  a este valor. Descripto en forma de diagrama:

$$\mathbf{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$$

Si el valor obtenido al recibir un ejemplo  $\mathbf{x}^{(i)}$  de entrada es mayor o igual que un valor límite prefijado llamado  $\theta$ , predice la clase **+1** y en otro caso predice la clase **-1**. En el algoritmo de decisión del *perceptrón* es:

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq \theta, \\ -1 & \text{otherwise.} \end{cases}$$

Para simplificar, ese valor límite  $\theta$  que utiliza la neurona podemos introducirlo en la ecuación que calcula el escalar como su peso  $w_0$  y los datos de entrada tendrán un  $x_0$  adicional siempre a 1 para dejar el límite fijo. Así la ecuación que mezcla el dato de entrada con lo que sabe la neurona es:

$$z = w_0 x_0 + w_1 x_1 + \dots + w_m x_m = \mathbf{w}^T \mathbf{x}$$

Y la función de activación utiliza ahora el cero como umbral:

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq 0, \\ -1 & \text{otherwise.} \end{cases}$$

En machine learning al límite negativo o peso  $w_0 = -\theta$  se le llama unidad **bias**<sup>2</sup>.

### ÁLGEBRA LINEAL: PRODUCTO DOT

Producto de un vector transpuesto por otro vector o una matriz. La T indica la operación traspuesta.

$$z = w_0x_0 + w_1x_1 + \dots + w_mx_m = \sum_{j=0}^m x_j w_j = \mathbf{w}^T \mathbf{x}$$

Por ejemplo:

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \times \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} = 1 \times 4 + 2 \times 5 + 3 \times 6 = 32$$

La traspuesta también se aplica a matrices. De hecho, solo está definida para matrices. Pero en el contexto del machine learning, se puede considerar un vector de n valores como una matriz  $n \times 1$  y un vector traspuesto como una matriz  $1 \times n$ . Ejemplo:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

La siguiente figura ilustra como los datos de entrada en la neurona  $z = \mathbf{w}^T \mathbf{x}$  se transforman en una salida binaria (-1 o bien +1) por la acción de la función de decisión del *perceptrón* que vemos representada en la figura 5. Además vemos en la figura como se usa para diferenciar a elementos de una clase de elementos de otra clase siempre que los elementos de ambas sean linealmente separables:

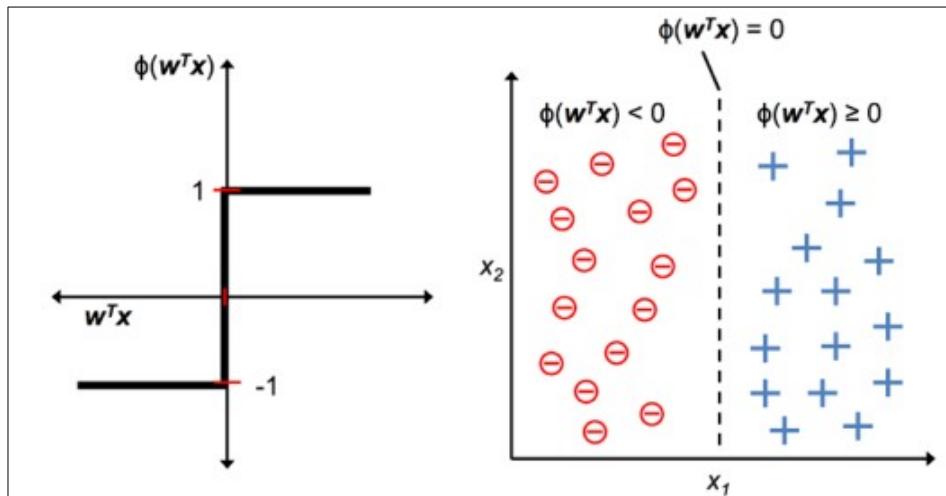


Figura 5. Función de decisión y frontera de decisión.

### ALGORITMO DE APRENDIZAJE DEL PERCEPTRÓN ¿CÓMO APRENDE?

La idea del modelo MCP de neurona y la regla *perceptrón* de Rosenblatt es utilizar una aproximación reduccionista para imitar el funcionamiento de una única neurona del cerebro: dispara una respuesta o no. Así que la idea inicial era bastante simple y puede resumirse así:

1. Inicializar los pesos de la neurona a pequeños valores aleatorios.
2. Para cada ejemplo de entrenamiento  $\mathbf{x}^{(i)}$ :
  - 2.1. Calcular el valor de salida  $\hat{y}$  que propone la neurona.
  - 2.2. Actualizar los pesos si da una respuesta equivocada.

2 **Bias:** sesgo en castellano. Vendría a significar un rumbo fijo o tendencia que tiene algo. En este contexto es un valor constante que usa la neurona. Por ejemplo si trucas una moneda para que salga más veces cara que cruz, la moneda tendría un sesgo, un bias (una tendencia). O si en una línea de corriente eléctrica siempre hay un nivel de voltaje (componente continua), ese valor sería un sesgo.

El valor de salida de la neurona es una de las dos clases usadas para clasificar los ejemplos y que resulta de aplicar la función de decisión. Y la actualización simultánea de los pesos  $w_j$  del vector de pesos  $w$  puede escribirse así:

$$w_j = w_j + \Delta w_j$$

El valor  $\Delta w_j$  representa la variación de valor de  $w_j$  y se usa para actualizar el peso  $w_j$  y se calcula mediante la regla de aprendizaje del perceptrón:

$$\Delta w_j = \eta [y^{(i)} - \hat{y}^{(i)}] x_j^{(i)}$$

Donde:

- $\eta^3$  es el **ratio de aprendizaje (learning rate)**, un valor escalar entre 0.0 y 1.0.
- $y^{(i)}$  es el valor real, valor de la característica label o target. La etiqueta del i-ésimo ejemplo de entrenamiento, la clase real a la que pertenece el ejemplo.
- $\hat{y}^{(i)}$  es la clase que la neurona ha calculado para el i-ésimo ejemplo usando su vector de pesos  $w$  y aplicando su función de activación.

Es importante observar que todos los pesos del vector  $w$  deben actualizarse a la vez, en el sentido de que no podemos volver a calcular otro  $\hat{y}^{(i)}$  hasta que a todos los pesos se les aplique su  $\Delta w_j$  porque si solamente modificamos algunos y no otros, la neurona no aprenderá de manera correcta.

Para unos datos de dos dimensiones (cada ejemplo tiene 2 características o columnas además de las columnas target y bias). El perceptrón tendrá pesos para el bias ( $w_0$ ) y para cada característica que se usa para predecir la columna target ( $w_1$  y  $w_2$ ). Podríamos escribir el proceso de actualizar cada peso de esta forma, si se actualizan sus pesos en cada ejemplo de entrenamiento que usa:

$$\Delta w_0 = \eta(y^{(i)} - \text{salida}^{(i)}) 1$$

$$\Delta w_1 = \eta(y^{(i)} - \text{salida}^{(i)}) x_1^{(i)}$$

$$\Delta w_2 = \eta(y^{(i)} - \text{salida}^{(i)}) x_2^{(i)}$$

Para comprender bien lo que hace el algoritmo de aprendizaje y su ingeniosa definición analizamos los dos escenarios posibles si la neurona predice correctamente la clase del ejemplo (es decir cuando  $y^{(i)}$  es igual que  $\hat{y}^{(i)}$ ) y en ese caso los pesos permanecen sin modificar porque las variaciones quedan a 0:

$$\text{Si el ejemplo es de La clase -1: } \Delta w_j = \eta(-1 - (-1)) x_j^{(i)} = 0$$

$$\text{Si el ejemplo es de La clase +1: } \Delta w_j = \eta(+1 - (+1)) x_j^{(i)} = 0$$

Sin embargo cuando la neurona da una respuesta equivocada, los pesos se van modificando para compensar la diferencia entre lo que calcula y la respuesta correcta (subiendo pesos si se queda corto o bajándolos si se pasa por exceso en el cálculo). Hay dos posibilidades:

$$\text{El ejemplo es +1 pero predice -1: } \Delta w_j = \eta(+1 - (-1)) x_j^{(i)} = 2\eta x_j^{(i)}$$

$$\text{El ejemplo es -1 pero predice +1: } \Delta w_j = \eta(-1 - (+1)) x_j^{(i)} = -2\eta x_j^{(i)}$$

Para ponerlo a prueba de nuevo, supongamos el caso de un factor multiplicativo no entero y donde la neurona falla al calcular la clase:

$$\eta = -1, \quad y^{(i)} = +1, \quad x_j^{(i)} = 0.5, \quad \hat{y}^{(i)} = -1 \quad -2\eta x_j^{(i)}$$

En este caso, se incrementa el peso en 1 para que el cálculo de la neurona sea más positivo la próxima vez y así de una respuesta más cercana a la solución correcta (+1) en próximas ocasiones.

$$\Delta w_j = (1 - (-1)) 0.5 = (2) 0.5 = 1$$

---

3  $\eta$ : es la letra griega conocida como eta.

El peso  $w_j$  se incrementa proporcionalmente al valor de  $x_j^{(i)}$ . Si aparece otro ejemplo donde  $x_j^{(i)}$  es 2 y el resultado de la neurona es erróneo al clasificarlo porque da como resultado -1, incrementará más el peso para poder aproximarse la próxima vez al valor correcto:

$$\Delta w_j = (1 - (-1))2 = (2)2 = 4$$

**La convergencia hacia un aprendizaje correcto en el perceptrón solo se puede garantizar si las dos clases son linealmente separables y el ratio de aprendizaje es suficientemente pequeño.** Si las clases no pueden separarse mediante una frontera que sea una línea, habrá que poner un límite en el número de épocas que repasar o un límite a la cantidad de errores que damos por buenos (dos condiciones de parada) o nunca deja de actualizar sus pesos porque siempre comete errores.

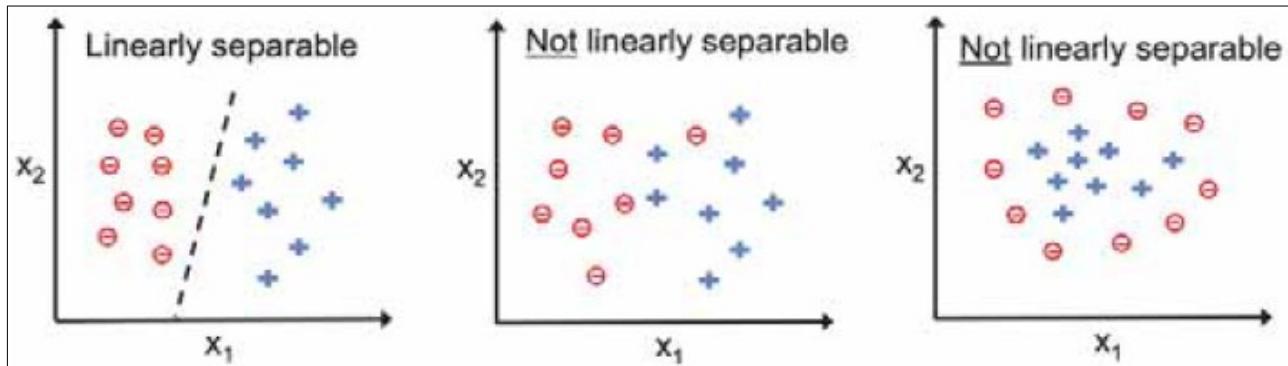


Figura 6. Situaciones linealmente separables y no separables.

El siguiente diagrama ilustra el funcionamiento del perceptrón:

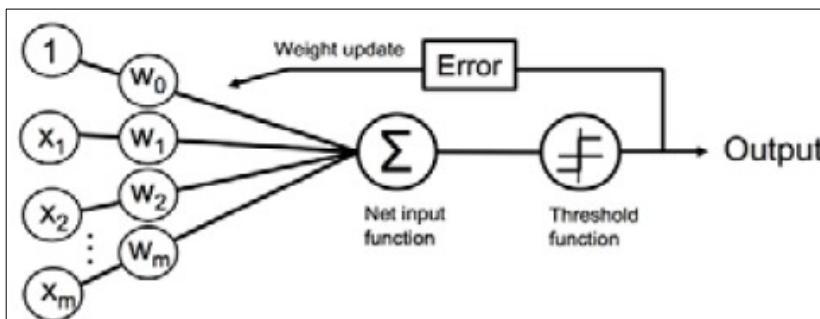


Figura 7. Diagrama de funcionamiento del perceptrón.

Aunque el perceptrón clasifica bien, la convergencia es uno de sus mayores problemas. Frank Rosenblatt demostró matemáticamente que el perceptrón converge si las dos clases están separadas por un hiperplano lineal. Sin embargo, si no están perfectamente separadas por una frontera de decisión lineal, los pesos nunca paran de actualizarse a no ser que se introduzcan condiciones de parada adicionales como un número máximo de épocas o una cantidad de errores que consideremos aceptable.

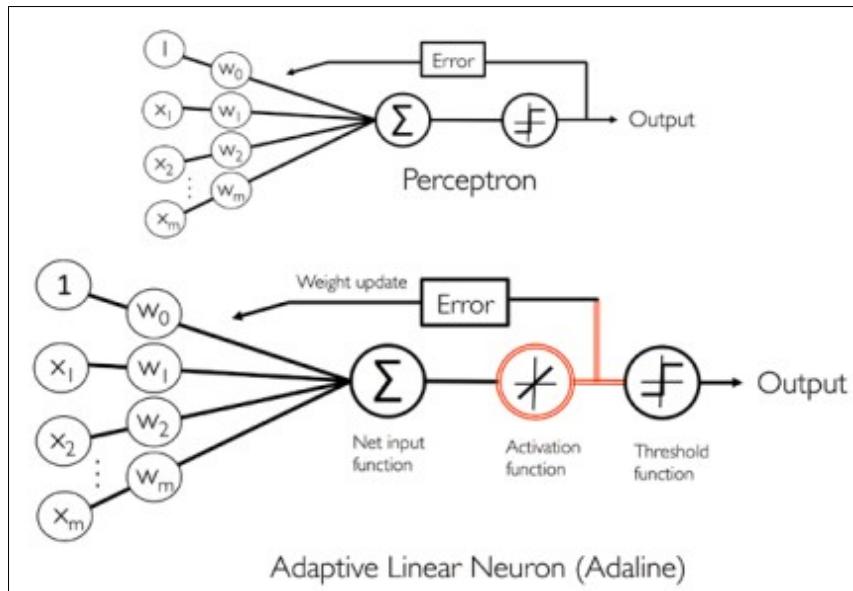
### PRÁCTICA 1. REALIZA LA PRÁCTICA 1 DE ESTA UNIDAD.

## 3. ADALINE Y LA FUNCIÓN DE COSTE.

El algoritmo de aprendizaje **ADALINE** (ADaptive LINEar NEuron) fue publicada por **Bernard Widrow** y su estudiante de doctorado **Tedd Hoff** solo unos pocos años después de aparecer el algoritmo del perceptrón de Frank Rosenblatt y podría considerarse como una mejora. La diferencia clave entre ADALINE y el perceptrón es que en ADALINE los pesos se actualizan basándose en cualquier función de activación en vez de en una función de salto unidad. En ADALINE esta función de activación lineal  $\phi(z)$  es simplemente la función identidad de la entrada en la neurona:

$$\phi(\mathbf{w}^T \mathbf{x}) = \mathbf{w}^T \mathbf{x}$$

Aunque la función de activación se usa en el aprendizaje de los pesos correctos, después se sigue utilizando la función de decisión para poner un límite en la respuesta de la neurona cuando lanza una predicción, lo que es parecido a lo que hace el perceptrón. La siguiente figura muestra esquemáticamente las principales diferencias entre el *perceptrón* y *ADALINE*:



**Figura 8.** Esquemas de funcionamiento de perceptrón y Adaline.

Como se ve en la figura (en rojo), *ADALINE* compara la clase real de los ejemplos de entrada con la función de activación continua para calcular el error y actualizar los pesos. Al contrario de lo que hace el *perceptrón* que compara las clases reales de los ejemplos con la clase que predice la neurona.

### MINIMIZAR FUNCIONES DE COSTE CON DESCENSO POR GRADIENTE

Uno de los ingredientes clave de los algoritmos de machine learning es definir una función objetivo que sea minimizada durante el proceso de aprendizaje. Esta función objetivo es con frecuencia una **función de coste que mide el error que comete el modelo**. También se la conoce como **función Loss** y sus variables independientes dependen de los parámetros que actualmente tiene el modelo:  $\text{Loss}(w) = \text{objetivo}(w) = \text{Error}(w) = \text{cuanto error comete el modelo cuando intenta realizar su trabajo}$ . El **algoritmo de aprendizaje intenta minimizar esta función, es decir, encontrar los parámetros w que hacen que el error sea el mínimo**.

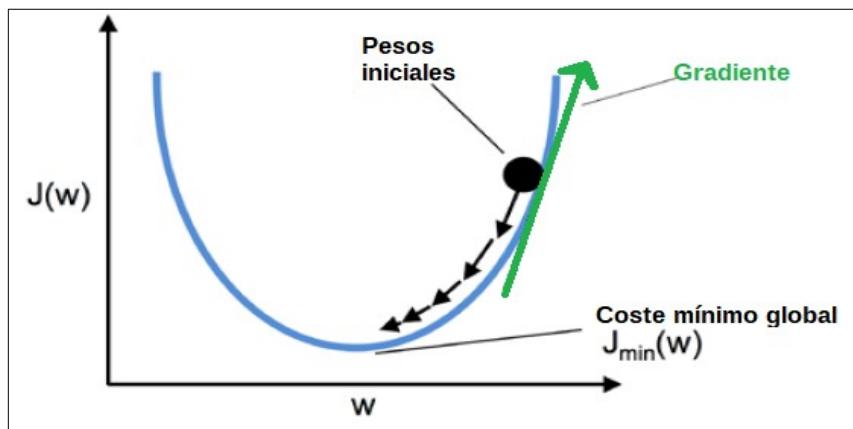
En el caso de *ADALINE*, define la función de coste  $J(w)$ , llamada **SSE (Sum Squared Error)** para encontrar los pesos (aprender) que hagan mínima la suma del cuadrado de los errores (diferencias entre lo que calcula y los valores reales de los ejemplos):

$$J(w) = \frac{1}{2} \sum_i (y^{(i)} - \phi(z^{(i)}))^2$$

El término  $\frac{1}{2}$  se añade para hacer más sencillo el cálculo de la derivada parcial de la función (en realidad no altera el resultado: si minimizas algo, minimizas su mitad,  $\frac{1}{2}$  de algo) con respecto a los pesos. La principal ventaja de esta función de activación lineal con respecto de la función de salto es que ahora la **función de error es diferenciable**. Otra buena propiedad es que es **convexa** (un cuadrado tiene forma de parábola, como una copa, así que tendrá un mínimo estupendo en algún lugar cuando las derivadas parciales con respecto a los parámetros sean cero). El **algoritmo de descenso por gradiente** es un sencillo y potente algoritmo iterativo (por pasos) para encontrar los parámetros que hagan mínimo el error).

El **gradiente es una dirección** (un vector) que tiene la mayor pendiente en cierto punto. Si te mueves un poco desde ese punto actual en dirección contraria al gradiente, irás a otro lugar más bajo, un lugar con un error más pequeño que donde estabas antes. No puedes dar saltos muy grandes porque

igual te pasas el mínimo y acabas en otro lugar más alto que el actual. Tampoco es buena idea dar pasos muy, muy cortos, porque tardarás mucho en bajar. La longitud de los pasos que da el algoritmo lo controla el hiperparámetro **learning rate** (el ratio de aprendizaje).



**Figura 9.** Algoritmo Descenso por Gradiente (GD).

Usando el descenso por gradiente podemos actualizar los pesos de manera iterativa: comenzamos en un lugar aleatorio (pesos aleatorios), medimos el error, calculamos el gradiente del lugar y damos un paso de cierta longitud en la dirección contraria (actualizamos los pesos). **El gradiente de la función de coste  $J(w)$  se escribe  $\nabla J(w)$** . Así que en cada paso actualizaremos los pesos en cierta cantidad que podemos escribir como variación de pesos  $\Delta w$  de esta forma:

$$w \leftarrow w + \Delta w$$

La variación de pesos  $\Delta w$ , se define como el gradiente negativo multiplicado por el learning rate  $\eta$ :

$$\Delta w = -\eta \nabla J(w)$$

Para calcular el gradiente de la función de coste, hay que calcular las derivadas parciales de la función de coste con respecto a cada peso  $w_j$  del vector de pesos  $w$ . El resultado será otro vector de tantos elementos como elementos tiene  $w$ :  $[\partial J / \partial w_0, \partial J / \partial w_1, \dots, \partial J / \partial w_m]^T$

$$\frac{\partial J}{\partial w_j} = -\sum_i (y^{(i)} - \phi(z^{(i)})) x_j^{(i)}$$

Así que se puede escribir el cambio que hay que apagar a cada peso  $w_j$  como:

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_i (y^{(i)} - \phi(z^{(i)})) x_j^{(i)}$$

Si actualizamos todos los pesos de manera simultánea, **la regla Adaline es:**

$$w \leftarrow w + \Delta w$$

### DERIVADA DE LA FUNCIÓN DE ERROR SSE (SUM SQUARED ERROR)

Si estás familiarizado con el cálculo de derivadas parciales, la derivada parcial del SSE con respecto al  $j$ -ésimo peso puede obtenerse con estos cálculos:

$$\frac{\partial J}{\partial w_j} = \frac{\partial}{\partial w_j} \frac{1}{2} \sum_i (y^{(i)} - \phi(z^{(i)}))^2 = \frac{1}{2} \frac{\partial}{\partial w_j} \sum_i (y^{(i)} - \phi(z^{(i)}))^2$$

$$= \frac{1}{2} \sum_i 2(y^{(i)} - \phi(z^{(i)})) \frac{\partial}{\partial w_j} (y^{(i)} - \phi(z^{(i)})) = \sum_i (y^{(i)} - \phi(z^{(i)})) \frac{\partial}{\partial w_j} \left( y^{(i)} - \sum_i (w_j^{(i)} x_j^{(i)}) \right)$$

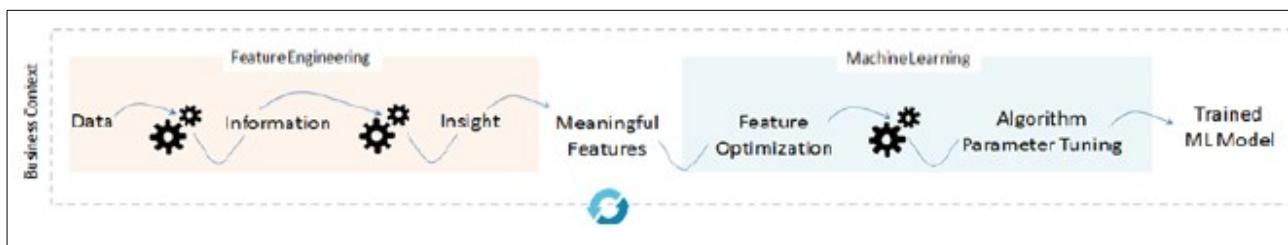
$$= \sum_i (y^{(i)} - \phi(z^{(i)})) (-x_j^{(i)}) = -\sum_i (y^{(i)} - \phi(z^{(i)})) x_j^{(i)}$$

Aunque ADALINE parece muy similar al *perceptrón*, debes fijarte en que  $\phi(z^{(i)})$  con  $z^{(i)} = w^T x^{(i)}$  es un número real y no un valor entero. Por eso al predecir, se vuelve a enviar a la función de salto. Si antes de actualizar los pesos, se mide el error en todos los ejemplos de entrenamiento, en vez de actualizar los pesos cada vez que se mide el error cometido en cada ejemplo individual, se llama **descenso por gradiente batch** (por lotes).

## PRÁCTICA 2. REALIZA LA PRÁCTICA 2 DE ESTA UNIDAD.

## 4. INGENIERÍA DE CARACTERÍSTICAS.

La calidad de las predicciones de salida de cualquier algoritmo de machine learning dependen en gran medida de la calidad de los datos que se le proporcionan para aprender. El proceso de crear las características adecuadas de los datos en el contexto de la tarea que se necesita es a lo que se denomina *ingeniería de características* y es una de las claves para construir de forma eficiente sistemas de aprendizaje automático. Cada algoritmo tiene diferentes necesidades y debilidades que pueden afectarle en mayor o menor medida. En este apartado vamos a ver algunas de ellas.



*Figura 10. Flujo lógico de datos en Machine Learning*

Los datos en el flujo del proceso comienzan como datos crudos (raw) que provienen de diferentes fuentes y tipos. Estos datos se transforman en información (datos procesados) para que describan aspectos interesantes del negocio. A continuación se crean las categorías que pueden ser útiles para resolver las tareas que se necesitan o para proporcionárselos de manera adecuada a los algoritmos que construyen un modelo de machine learning para que puedan funcionar de manera eficiente.

La salida de este proceso de ingeniería de características implica completar datos que faltan, o reducir la cantidad de características para aumentar la eficiencia de construcción del modelo o mejorar su calidad. Este flujo no es lineal, es iterativo. Es decir, tras construir el modelo otros algoritmos lo validarán y ajustarán sus hiperparámetros, y esto quizás exija volver a aplicar mejoras a los datos que se usan. Comenzamos viendo algunas técnicas básicas.

### 4.1. DATOS AUSENTES.

Los datos ausentes pueden falsear o crear problemas cuando se analizan o procesan datos. Para evitar estos problemas necesitas evitar que falten datos en las características. Otra solución es llenar de alguna manera el dato que falta. Se suelen utilizar estas estrategias:

- **Borrar el ejemplo:** se elimina el ejemplo al que le falte algún dato en alguna de sus características. En ese caso, también pierdes la valiosa información que sí tiene en el resto de sus características. Puedes usar esta técnica si la cantidad de ejemplos que no tienen datos es insignificante (digamos < 5%) respecto del total de filas. La función de Pandas **dropna()**.
- **Rellenarlo con un valor resumen:** es la técnica de **imputación de valor** que más se utiliza. El resumen puede ser la media, la moda o la mediana según sea categórica o numérica la característica. Mira la función de Pandas **fillna()**.
- **Imputar un valor aleatorio:** escoger aleatoriamente uno de los valores de la columna y reemplazar el valor ausente. Puede ser una opción si la cantidad de valores ausentes es insignificante.
- **Usar un modelo predictivo:** una solución más avanzada consiste en entrenar un modelo de regresión con las filas que tienen valor para predecir qué valor tendría la columna en esta fila.

## 4.2. DATOS CATEGÓRICOS.

La mayoría de algoritmos trabajan con datos numéricos. Así que las variables originales que tienen datos simbólicos o categóricos no pueden utilizarse directamente con estos algoritmos. Para hacerlo posible hay diferentes métodos.

### CREAR COLUMNAS DUMMIES (ONE HOT ENCODER)

Es una característica booleana que indica la presencia de una categoría con el valor numérico 1 y la ausencia de esa categoría con el valor numérico 0. Si una característica categórica tiene k valores simbólicos diferentes, debes crear k-1 características dummies. Scikit-learn tiene la función **get\_dummies()** para realizar transformaciones '*One Hot Encoder*' creando columnas dummy.

```
import pandas as pd
from patsy import dmatrices
df = pd.DataFrame( {'A': ['alto', 'medio', 'bajo'],
                     'B': [10, 20, 30]}, index=[0, 1, 2])
print(df)
# usar get_dummies()
df_con_dummies= pd.get_dummies(df, prefix='A', columns=['A'])
print(df_con_dummies)
```

	A	B
0	alto	10
1	medio	20
2	bajo	30
	B	A_alto A_bajo A_medio
0	10	1 0 0
1	20	0 0 1
2	30	0 1 0

### CONVERTIR A NUMÉRICA

Otra alternativa es representar mediante un código numérico cada valor simbólico de una característica. Puedes usar la función **LabelEncoder()** de Scikit-learn. Si la cantidad de diferentes valores es alta, puedes aplicar la lógica de negocio para cambiar valores por grupos de valores (por ejemplo códigos postales por provincia), aunque con este método puedes perder información. Otro método puede ser crear valores que describan la frecuencia: alta, media, baja. Puedes usar la función **factorize()** de Pandas.

```
import pandas as pd
df['A_pd_factorizada'] = pd.factorize(df['A'])[0]
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
df['A_skl_Convertida'] = le.fit_transform(df.A)
print(df)
```

	A	B	A_pd_factorizada	A_skl_Convertida
0	alto	10	0	0
1	medio	20		1
2	bajo	30	2	2
PS C:\Users\Jose>				

## 4.3. DETECCIÓN Y ELIMINACIÓN DE OUTLIERS

Un dato outlier podría traducirse como un dato anómalo, es decir, un valor que se desvía significativamente del resto de valores. A los valores que no son outliers los llamaremos normales. Los valores outlier no son exactamente datos ruidosos.

A algunos algoritmos de aprendizaje automático les afecta mucho la presencia de outliers y provocan que no funcionen bien. Para detectarlos y eliminarlos hay muchos métodos diferentes y algunos no son nada triviales, porque hay muchos tipos de outliers (globales, contextuales, temporales, ....). Por ejemplo para detectarlos tenemos métodos supervisados, semisupervisados y no supervisados. Y para eliminarlos tenemos métodos estadísticos, métodos basados en proximidad y otros basados en la reconstrucción.

Considera este pequeño vector que podría contener los valores de una característica numérica de un dataset.

```
columna= [15, 101, 18, 7, 13, 16, 11, 21, 5, 15, 10, 9]
```

Como tiene pocos datos, observándolo a simple vista vemos que tiene un valor que es muy diferente al resto, el segundo valor comenzando por la izquierda es 101 y es mucho mayor que el resto. Si calculamos la media con él y sin él vemos que hay mucha diferencia.

with outlier	without outlier
Mean: 20.08	Mean: 12.72
Median: 14.0	Median: 13.0
Mode: 15	Mode: 15
Variance: 614.74	Variance: 21.28
Std dev: 24.79	Std dev: 4.61

Figura 11. parámetros estadísticos de datos con y sin outliers.

## DETECTANDO OUTLIER

Vamos a utilizar 3 métodos sencillos para detectarlos usando visualización y estadística, son:

- Diagramas Boxplot.
- Z-score.
- Rango inter quartílico: IQR

## DETECTAR OUTLIERS GENERANDO DIAGRAMAS BOXPLOT

Generamos un diagrama Bloxplot de los datos y veremos como aparecen fuera de la caja los outliers.

```
import matplotlib.pyplot as plt
columna= [15, 101, 18, 7, 13, 16,11, 21, 5, 15, 10, 9]
plt.boxplot(columna, vert=False)
plt.title("Detectando outliers usando Boxplot")
plt.xlabel('columna')
plt.show()
```

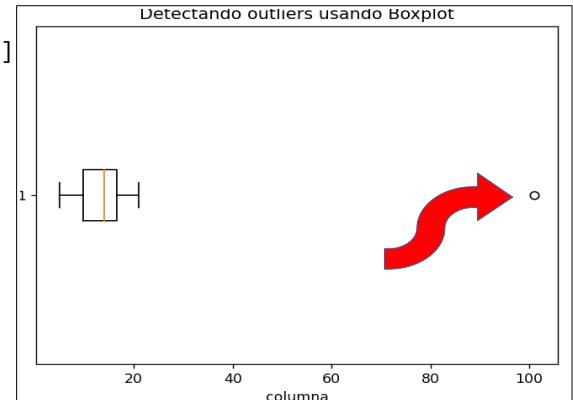


Figura 12. Visualizando los datos y su BoxPlot aparecen los outliers alejados de la caja.

## DETECTAR OUTLIERS CON Z-SCORES

En estadística hay un importante teorema que dice que cuando la cantidad de datos crece, acaban por formar una distribución normal (campana de Gauss). Convertimos los datos de la característica para que sigan una distribución normal de media 0 y desviación estándar 1. A estos valores les llamamos **z-scores**. Y adoptamos el criterio según el cual, consideraremos *outlier* a todo z-score que esté a una distancia de la media mayor (por arriba o por abajo) de 3 veces su desviación estándar.

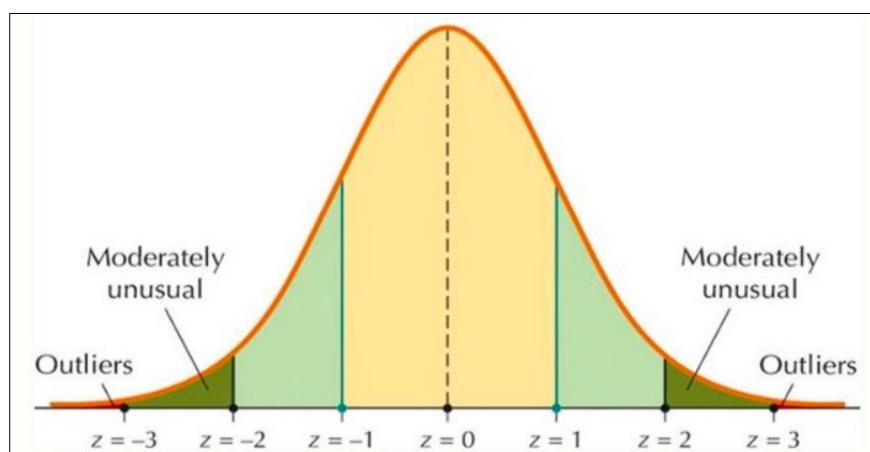


Figura 13. Distribución Normal(0,1) y posición de la media (en cero) y los z-scores.

Los  $z_i$  los podemos calcular como la versión estandarizada de los datos. Para detectarlos habría que dar estos pasos:

- Calcular la media y la desviación estándar de los datos.
- Calcular para cada valor  $x_i$  su *z-score*  $z_i$ :  $z_i = (x_i - \text{media}) / \text{desviación}$
- Si  $|z_i| > 3$  entonces el  $i$ -ésimo dato de X es un outlier.

Código en Python:

```
import numpy as np
columna= [15, 101, 18, 7, 13, 16, 11, 21, 5, 15, 10, 9]

def detecta_outliers_zscore(datos):
    outliers= []
    limite = 3
    media= np.mean(datos)
    desviacion_estandar = np.std(datos)
    print('Media:', media, 'desviación estándar:', desviacion_estandar)
    for x in datos:
        z_score = (x - media) / desviacion_estandar
        if (np.abs(z_score) > limite):
            outliers.append(x)
    return outliers

outliers = detecta_outliers_zscore(columna)
print("Outliers usando método Z-score:", outliers)
```

### DETECTANDO OUTLIERS USANDO IQR.

El fundamento es el mismo que en el método de z-scores pero usando la mediana y los cuartiles 1 y 3 en vez de la media. La idea es que son outliers los datos que están más lejos de la mediana que 1.5 veces el IQR. El IQR es la diferencia entre el percentil 75 (cuartil Q3) y el percentil 25 (cuartil Q1). Pasos para detectarlos:

- Ordenar los datos en orden ascendente (de menor a mayor).
- Calcula el primer y tercer cuartil: Q1 y Q3
- Calcular el  $IQR = Q3 - Q1$
- Calcular el límite de normalidad inferior como  $Q1 - 1.5 \cdot IQR$
- Calcular el límite de normalidad superior como  $Q3 + 1.5 \cdot IQR$
- Examinar todos los valores de la característica y marcar como outlier el que esté por debajo del límite inferior o por encima del límite superior.

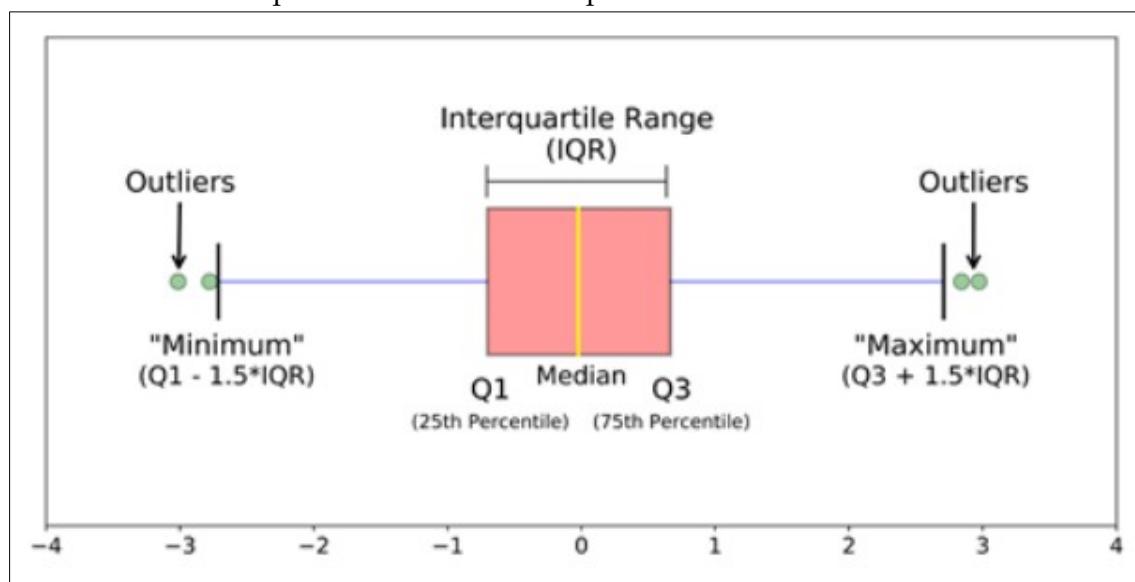


Figura 14. BoxPlot de la mediana, el IQR y posicionamiento de los outliers.

El código en Python que muestra como lo realiza:

```
import numpy as np
columna= [15, 101, 18, 7, 13, 16, 11, 21, 5, 15, 10, 9]

def detecta_outliers_iqr(datos):
    outliers = []
    datos= sorted(datos)
    q1 = np.percentile(datos, 25)
    q3 = np.percentile(datos, 75)
    # print("Q1:", q1, "Q3:", q3)
    IQR = q3 - q1
    limite_inferior = q1 - 1.5 * IQR
    limite_superior = q3 + 1.5 * IQR
    # print("Normalidad en [", limite_inferior, ", ", limite_superior, "]")
    for x in datos:
        if (x < limite_inferior or x > limite_superior):
            outliers.append(x)
    return outliers

outliers_de_columna = detecta_outliers_iqr(columna)
print("Outliers usando método IQR:", outliers_de_columna)
```

## COMO GESTIONAR LOS OUTLIERS

Tenemos varias posibilidades, no todas igual de buenas.

**POSIBILIDAD 1: Eliminar los valores.** No es una buena opción eliminar los valores, porque al final tendrías que deshacerte de todo el ejemplo y pierdes información. Así que habría que hacer algo más.

```
# Borrar el valor '101' y copiar resto de valores a otro array a.
outliers = detecta_outliers_zscore(columna)
a = np.copy(columna)
for i in outliers:
    a = a[a != i]
print("Datos originales:", columna)
print("Outliers:", outliers)
print("Datos sin outliers:", a)
print("Originals:", len(columna), "Sin outliers:", len(a))
```

**POSIBILIDAD 2: Limitar en un rango los valores usando el IQR.** Con este método, cambiamos el valor del outlier limitándolo a [Q1-1.5 IQR, Q3+1.5 IQR].

**EJEMPLO 1:** Este código Python detecta y cambia los *outliers* usando *IQR*:

```
import numpy as np
columna= [15, 101, 18, 7, 13, 16, 11, 21, 5, 15, 10, 9]
# Calcular el percentil 25 y 75 y reemplazar los outliers
Q1 = np.percentile(columna,25)
Q3 = np.percentile(columna, 75)
IQR = Q3 - Q1
limite_inferior = Q1 - 1.5 * IQR
limite_superior = Q3 + 1.5 * IQR
print("Datos no anómalos en [", limite_inferior, ", ", limite_superior, "] con IQR:", IQR)
b = np.where(columna < limite_inferior, limite_inferior, columna)
b = np.where(b > limite_superior, limite_superior, b)
print("Original: ", columna)
print("Sin outliers:", b)
```

**POSIBILIDAD 3: imputar un valor con la media o la mediana.** Como la media está muy influenciada por los outliers, es más conveniente usar la mediana.

**EJEMPLO 2:** Este código Python detecta *outliers* usando *z-scores* y los imputa con la mediana:

```
import numpy as np
import matplotlib.pyplot as plt
# ... función detectar_outliers_zscore()
columna = [15, 101, 18, 7, 13, 16, 11, 21, 5, 15, 10, 9]
```

```

outliers = detecta_outliers_zscore(columna)
mediana = np.median(columna)
# Cambiar outliers por la mediana
c = np.copy(columna)
for i in outliers:
    c = np.where(c == i, mediana, c)
print("Original: ", columna)
print("Mediana: ", mediana)
print("Sin outliers:", c)
plt.boxplot(c, vert= False)
plt.title("Boxplot Tras Eliminar Outliers")
plt.xlabel("columna")
plt.show()

```

## 4.4. HOMOGENEIZAR VALORES.

Cada columna (variable, atributo o característica) numérica tiene valores en unidades o escalas diferentes. Por tanto un análisis con diferentes medidas podría dar información sesgada en aquellas que tengan valores absolutos altos. Además algunos algoritmos pierden efectividad y eficiencia para realizar su trabajo. Por ejemplo, en un dataset de venta de casas, la columna `num_piscinas` tendrá valores 0, 1 o como mucho 2. Mientras que la columna `superficie_parcela` tendrá valores de centenares de metros cuadrados a miles. Para unificar las escalas de todas las columnas se suelen utilizar principalmente dos métodos: normalización o estandarización y el escalado.

### ESCALAR LOS DATOS

Escalar los datos de una columna consiste en cambiarles la escala para dejarlos en un intervalo `[min, max]`. Normalmente se elige el intervalo `[0, 1]`. El algoritmo y la fórmula para escalar cada valor de la columna `x` al intervalo `[a, b]` consiste en:

```

x_min ← el mínimo valor del vector x
x_max ← el máximo valor del vector x
para i desde la primera hasta la última posición de los valores de x hacer
    x[i] ← a + (x[i] - x_min) / (x_max - x_min) * (b - a)

```

$$\text{nuevoX} = a + \frac{(x - x_{\min})}{(x_{\max} - x_{\min})} (b - a)$$

### NORMALIZAR O ESTANDARIZAR LOS DATOS

Esta transformación necesita calcular la media ( $\mu$ ) y la desviación estándar ( $\sigma$ ) de los datos. Consiste en cambiar cada valor por su z-score para que los datos formen una distribución Normal de media 0 y desviación típica 1:

$$Z = \frac{(X - \mu)}{\sigma}$$

Con mucha frecuencia es el método elegido para preparar los datos para que los usen muchos algoritmos de machine learning.

**Nota:** Asegurate de detectar y eliminar posibles outliers antes de estandarizar los datos o tendrás un importante sesgo o datos apretados en un pequeño intervalo muy próximos todos entre sí y esto perjudica a algunos algoritmos por ejemplo a los que hacen aprender a las redes neuronales.

### EJEMPLO 3: Escalar datos estandarizándolos o normalizándolos usando Z-Scores.

```

from sklearn import datasets
import numpy as np
from sklearn import preprocessing

iris = datasets.load_iris()
X = iris.data[:, [2, 3]]
y = iris.target
std_escala = preprocessing.StandardScaler().fit(X)

```

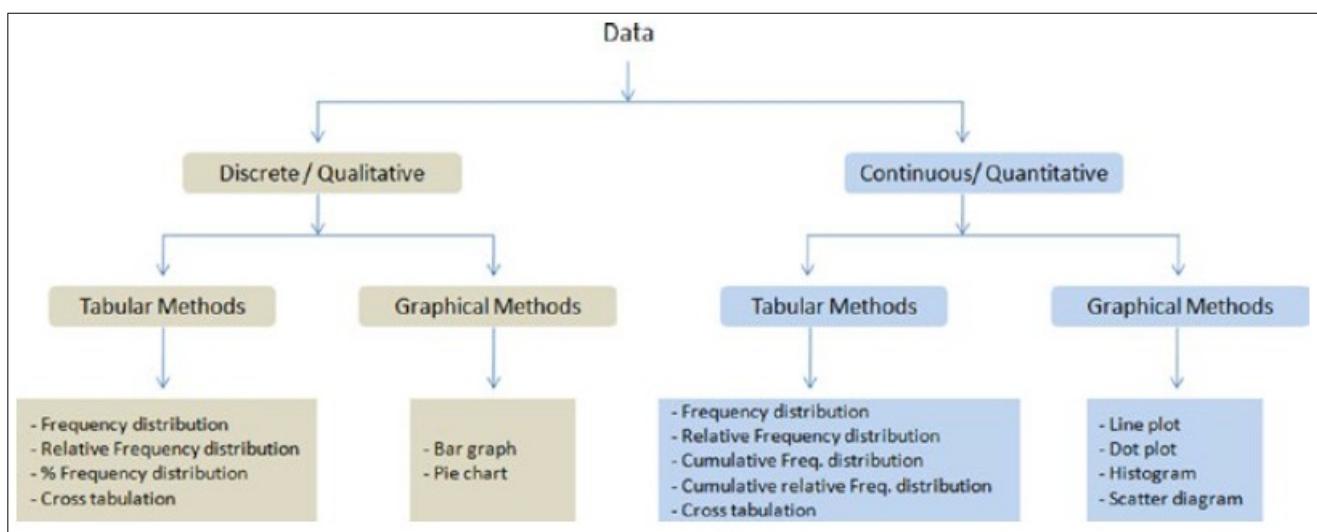
```

X_std = std_escala.transform(X)
minmax_escala = preprocessing.MinMaxScaler().fit(X)
X_mm = minmax_escala.transform(X)
print('Antes de estandarizar y Escalar:')
print('Medias:      long pétalos={:.1f}, ancho pétalos={:.1f}'.format(X[:,0].mean(), X[:,1].mean()))
print('Std:        long pétalos={:.1f}, ancho pétalos={:.1f}'.format(X[:,0].std(), X[:,1].std()))
print('Después de estandarizar:')
print('Medias:      long pétalos={:.1f}, ancho pétalos={:.1f}'.format(X_std[:,0].mean(), X_std[:,1].mean()))
print('Desviaciones: long pétalos={:.1f}, ancho pétalos={:.1f}'.format(X_std[:,0].std(), X_std[:,1].std()))
print('Después de escalar en [0,1]:')
print('Medias:      long pétalos={:.1f}, ancho pétalos={:.1f}'.format(X_mm[:,0].mean(), X_mm[:,1].mean()))
print('Desviaciones: long pétalos={:.1f}, ancho pétalos={:.1f}'.format(X_mm[:,0].std(), X_mm[:,1].std()))

```

## 4.5. CONSTRUIR O GENERAR CARACTERÍSTICAS.

Los algoritmos de ML solo pueden fabricar buenos modelos si les damos datos que tengan la estructura adecuada de características para el tipo de tarea que deban realizar. Muchas veces hay que crear manualmente características a partir de los datos crudos que tenemos y también a menudo hay que dedicar tiempo a estudiar qué relaciones guardan unas características con otras.



*Figura 15. Técnicas más comunes para estudiar los datos y características.*

Esto significa agregar, dividir o combinar características, crear otras nuevas o descomponerlas. Estas transformaciones están en estado de arte. Aplicarlas de manera manual es lento y requiere la experiencia de un experto en la materia de que se trate para crear las características que sean significativas para los algoritmos y el modelo que crean.

Resumir los datos es una técnica que nos ayuda a comprender su calidad, características, fortalezas y debilidades. La figura muestra los diferentes métodos que podemos utilizar según sea el tipo de valores de cada característica. No es un diagrama exhaustivo, aparecen los más comunes.

## 4.6. ANÁLISIS EXPLORATORIO DE DATOS: EDA.

Tiene que ver con todo lo que supone comprender los datos empleando técnicas de resumen y visualización. A alto nivel, EDA puede realizarse a dos niveles: análisis univariante y análisis multivariante.

### ANÁLISIS UNIVARIADO

Cada variable (columna o característica) se analiza de manera aislada e independiente para comprenderla. La librería *Pandas* ofrece la función **describe()** para crear un informe estadístico tabular de todas las columnas. Este informe indica información estadística, cantidad de valores ausentes y presencia de outliers.

```

from sklearn import datasets
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

```

```

iris = datasets.load_iris()
# Lo convertimos en un dataframe de Pandas
iris = pd.DataFrame(data= np.c_[ iris['data'], iris['target'] ],
                      columns= iris['feature_names'] + ['species'])
# Seleccionamos datos de versicolor y virginica
iris.species = np.where(iris.species == 0.0, 'setosa',
                        np.where(iris.species==1.0,'versicolor', 'virginica'))
iris.columns = iris.columns.str.replace(' ', '') # Eliminar espacios en los nombres de columnas
print(iris.describe())

```

	sepallength(cm)	sepalwidth(cm)	petallength(cm)	petalwidth(cm)
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.057333	3.758000	1.199333
std	0.828066	0.435866	1.765298	0.762238
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

Figura 16. La función Pandas.DataFrame.describe().

La columna 'species' es categórica, así que podemos comprobar la distribución de frecuencias de cada categoría con `value_counts()`.

```
print(iris['species'].value_counts())
```

setosa	50
versicolor	50
virginica	50
Name: species, dtype:	int64

Pandas también ofrece la visualización de distribución de valores de cada atributo `pd.DataFrame.hist()`.

```

# ... continúa desde arriba
# Fijar el tamaño del gráfico
iris.hist() # dibuja histogramas
plt.suptitle("Histogramas", fontsize=16)
plt.show()

```

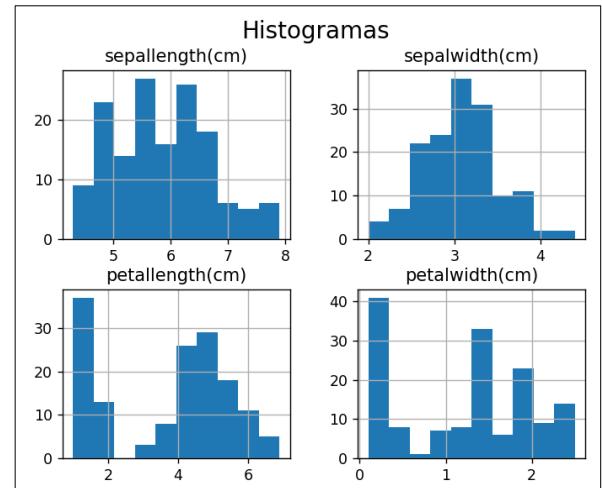
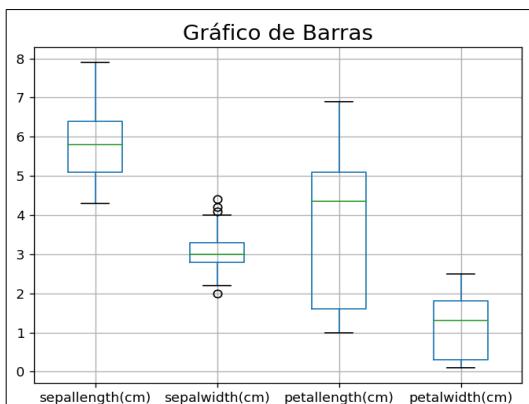


Figura 17. Histogramas de la función pd.DataFrame.hist()

También podemos visualizar gráficos box de cada columna con `pd.DataFrame.boxplot()`.



```

# ... continúa desde arriba
iris.boxplot() # Dibujar los boxplot
plt.title("Gráfico boxplot", fontsize=16)
plt.show()

```

Figura 18. Gráficos Boxplot de cada columna.

## ANÁLISIS MULTIVARIADO

En este análisis lo que intentas es comprender las relaciones de cada variable con el resto.

### PARÁMETROS POR CLASE

Podemos sacar parámetros agrupados por la clase y así poder comparar la diferencia del parámetro según las clases a las que pertenecen los ejemplos.

```
# Imprimir la media de cada columna según la especie
print(iris.groupby(by = "species").mean())
```

	sepallength(cm)	sepalwidth(cm)	petallength(cm)	petalwidth(cm)
species				
setosa	5.006	3.428	1.462	0.246
versicolor	5.936	2.770	4.260	1.326
virginica	6.588	2.974	5.552	2.026

```
# Dibujar media de característica por clase
iris.groupby(by="species").mean().plot(kind="bar")
plt.title('Medias por Clase')
plt.ylabel('Medias (cm)')
plt.xticks(rotation=0)
plt.grid(True)
# bbox_to_anchor: leyenda fuera
plt.legend(loc="upper left",
bbox_to_anchor=(1,1))
plt.show()
```

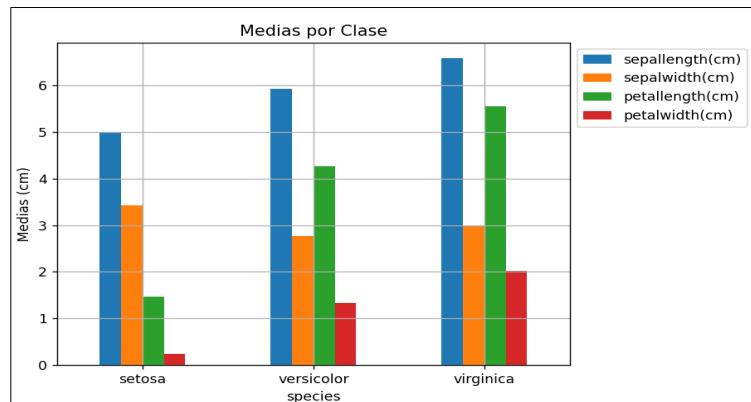


Figura 19. Medias por especie.

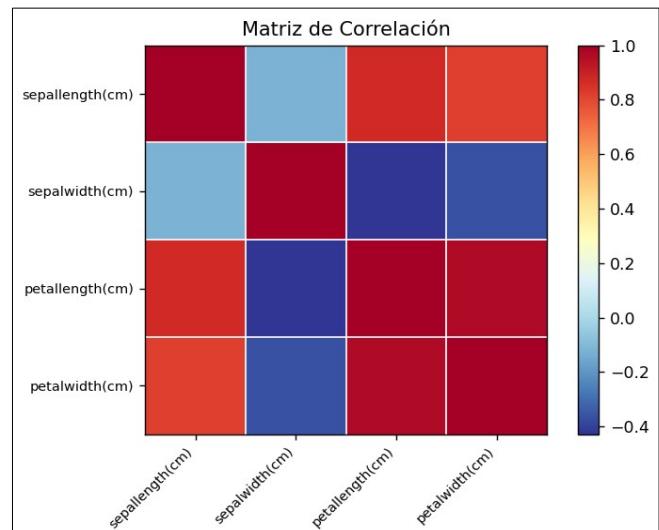
## MATRIZ DE CORRELACIÓN

La función `DataFrame.corr()` utiliza el *coeficiente de correlación de Pearson* cuyos valores van de -1 a +1. Una fuerte correlación inversa implica tener un coeficiente cercano a -1. De igual modo, una fuerte correlación directa supondrá tener un coeficiente cercano a +1. Si el coeficiente está cerca de 0, significa que no hay correlación lineal.

```
# Crea matriz de correlaciones
corr = iris.corr()
print(corr)
```

	sepallength(cm)	sepalwidth(cm)	petallength(cm)	petalwidth(cm)
sepallength(cm)	1.000000	-0.117570	0.871754	0.817941
sepalwidth(cm)	-0.117570	1.000000	-0.428440	-0.366126
petallength(cm)	0.871754	-0.428440	1.000000	0.962865
petalwidth(cm)	0.817941	-0.366126	0.962865	1.000000

```
# Quizás tengas que intalar el paquete: python -m pip install statsmodels
import statsmodels.api as sm
sm.graphics.plot_corr(corr, xnames=list(corr.columns))
plt.title("Matriz de Correlación")
plt.show()
```



*Nota: la diagonal empareja cada columna consigo misma y por tanto esta completamente correlacionada. Lo interesante de la matriz es fijarse en los colores rojos y azules más oscuros. Mira la escala de la derecha. A más oscuro más correlación hay. Correlación significa influencia estadística, no causalidad.*

Figura 20. Matriz de Correlación (coefs. de Pearson).

## GRÁFICO DE PARES DE PANDAS

Para visualizar las relaciones de cada característica con el resto podemos ver la distribución de los datos cada dos características en un gráfico de nube de puntos (scatter) cuando las características son distintas y la distribución cuando es la misma característica.

```
from pandas.plotting import scatter_matrix
scatter_matrix(iris, figsize=(10, 10))
plt.suptitle("Pair Plot", fontsize=10)
plt.show()
```

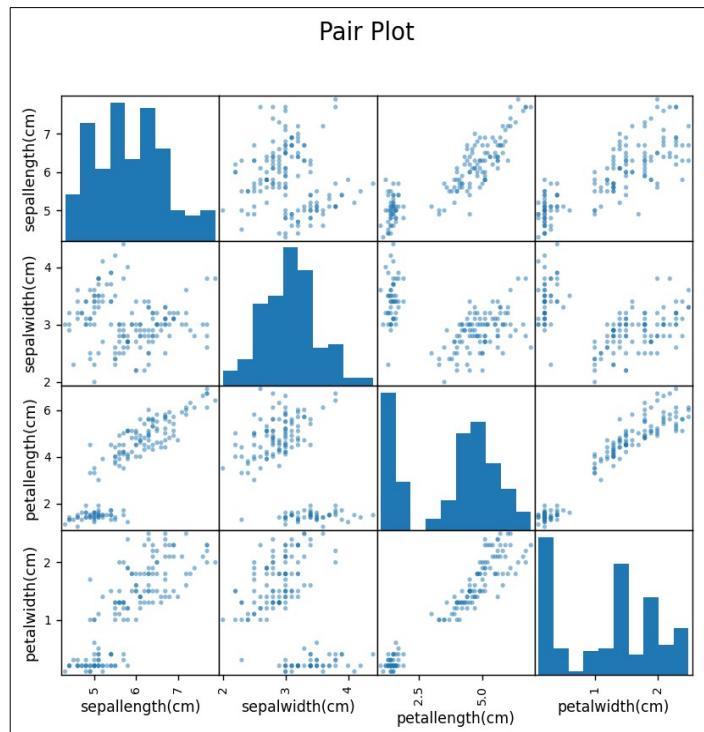


Figura 21-a. Gráfico de pandas.scatter\_matrix().

### GRÁFICO DE REGRESIÓN CON SEABORN

En *seaborn* podemos realizar un gráfico regresión donde consideramos dos características y podemos visualizar la recta de regresión (u otro tipo de relaciones: logarítmica, polinomial, logística, etc.). Podemos crearnos un gráfico similar al pair plot de *matplotlib* creando una rejilla de subgráficos. La única diferencia es que ahora mostramos también la línea de regresión entre cada dos variables:

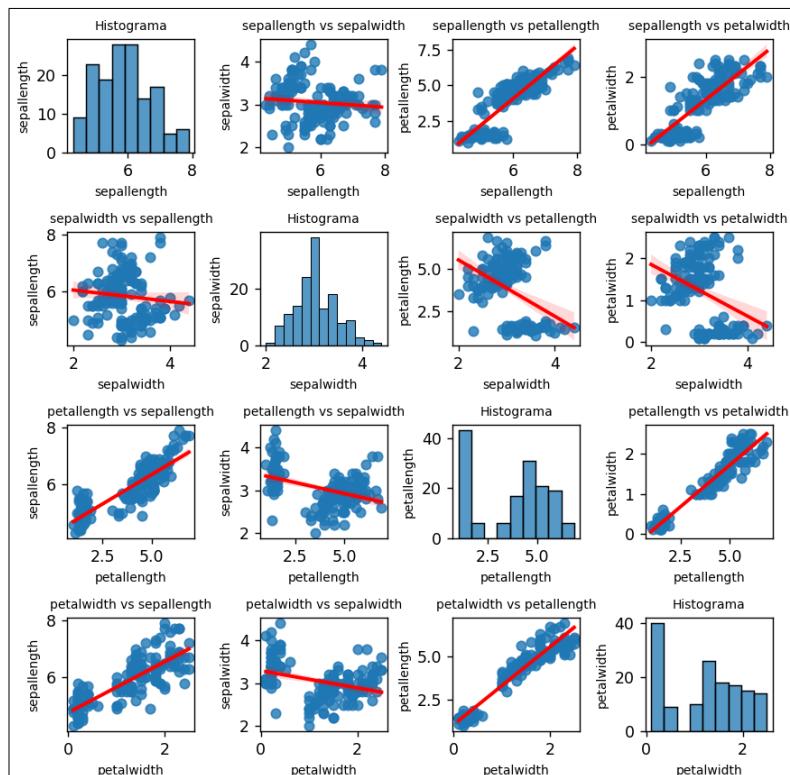


Figura 21-b. Gráfico realizado con seaborn.regplot(), seaborn.histo() y matplotlib.

```
# coding: utf8
import pandas as pd
```

```

iris = pd.read_csv("../iris.data")
iris.columns= ['sepallength', 'sepalwidth', 'petallength', 'petalwidth', 'tipo']
print(iris.head())
cols_numericas = ['sepallength', 'sepalwidth', 'petallength', 'petalwidth']
n_cols_numericas = len(cols_numericas) # Gráficos de regresión simple con target
import matplotlib.pyplot as plt
import seaborn as sns
import math
fig, ejes = plt.subplots( n_cols_numericas, n_cols_numericas, figsize=(7, 7))
ejes = ejes.flatten()
for i, col1 in enumerate(cols_numericas):
    for j, col2 in enumerate(cols_numericas):
        subgrafico = i * n_cols_numericas + j
        if i != j:
            sns.regplot(data=iris, x=col1, y=col2, ax=ejes[subgrafico],
                        line_kws={'color':'red'})
            ejes[subgrafico].set_title(col1 + " vs " + col2, fontsize=8)
        else:
            sns.histplot(data=iris, x=col1, ax=ejes[subgrafico])
            ejes[subgrafico].set_title("Histograma", fontsize=8)
            ejes[subgrafico].set_xlabel(col1, fontsize=8)
            ejes[subgrafico].set_ylabel(col2, fontsize=8)
plt.tight_layout()
plt.show()

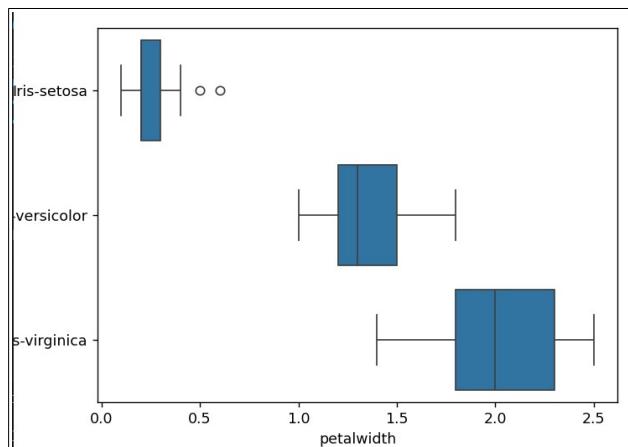
```

Lo que podemos descubrir analizando el dataset Iris con la ayuda de EDA:

- No tiene valores ausentes.
- Los sépalos son mayores que los pétalos. Los rangos de longitudes de los sépalos están entre 4.3 y 7.9 con media 5.8, mientras que los pétalos están entre 1 y 6.9 con media 3.7.
- Los sépalos también son más anchos que los pétalos. La anchura de los sépalos está entre 2 y 4.4 con una media de 3.05, mientras que la de los pétalos va desde 0.1 hasta 2.5 con media 1.19.
- La longitud media de los pétalos de setosa es mucho más pequeña que la de versicolor y virginica. Sin embargo la media del ancho de sépalos de setosa es mayor que la de versicolor y virginica.
- La longitud y la anchura de los pétalos están altamente correlacionadas, el 96% de las veces que se incrementa el ancho se incrementa también la longitud.
- La longitud de los pétalos es inversamente proporcional a la anchura de los sépalos. El 42% de las veces que se incrementa el ancho del sépalo, se decremente la longitud del pétalo.

**La conclusión inicial de los datos:** Se puede identificar versicolor y virginica con la longitud y ancho de los sépalos y pétalos. Sin embargo, las características de setosa no parecen muy diferentes.

En el caso de que una de las dos variables sea categórica podemos utilizar boxplots para saber si la distribución es similar o si cada *boxplot* tiene muchas diferencias podemos asegurar que si influye. Por ejemplo, observamos como según el tipo de flor, los valores del ancho de los pétalos se distribuyen de forma diferente y por tanto el ancho del pétalo influye en el tipo de flor.



**Consejo:** Mira la documentación online de scikit learn en su apartado [preprocessing](#) para conocer estas técnicas con mayor profundidad.

**Figura 21-c.** Gráfico boxplot categórica y numérica o numérica y numérica.

```
# Gráficos boxplot de categóricas con el target
categorica = 'tipo'
numerica = 'petalwidth'
sns.boxplot(data=iris, x=numerica, y=categorica)
plt.show()
```

## 5. ALGORITMOS DE REGRESIÓN LINEAL.

La regresión lineal predice un valor continuo utilizando uno o más valores de características independientes o predictoras. Por ejemplo podría interesarnos predecir el precio de venta de una casa usando su superficie, número de habitaciones, número de baños, ubicación, servicios cercanos, etc. La columna *target* debe ser numérica y las predictoras también.

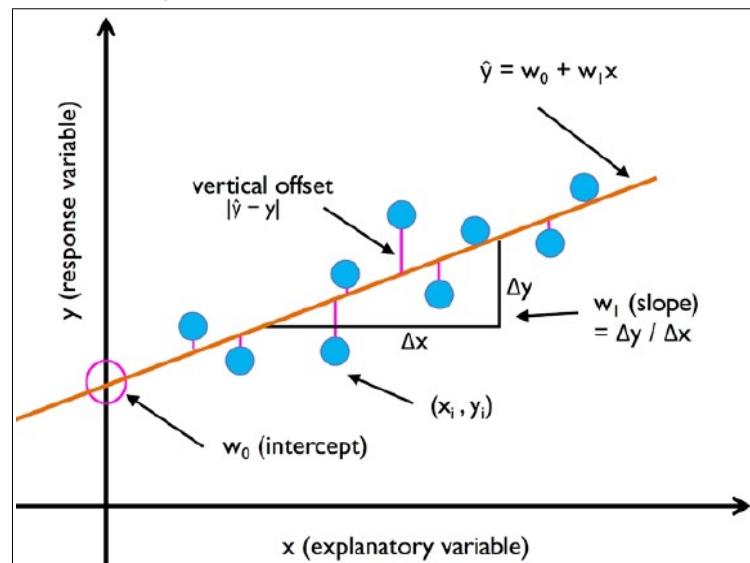


Figura 22. Línea de regresión cuando hay una sola predictora.

El algoritmo que define el modelo supone que hay cierta dependencia lineal entre la variable *target* y las predictoras. El modelo es una combinación lineal de las predictoras, la línea recta más cercana a todos los datos.

En el caso de dos variables predictoras ya no hablaríamos de línea recta sino de un plano (una superficie).

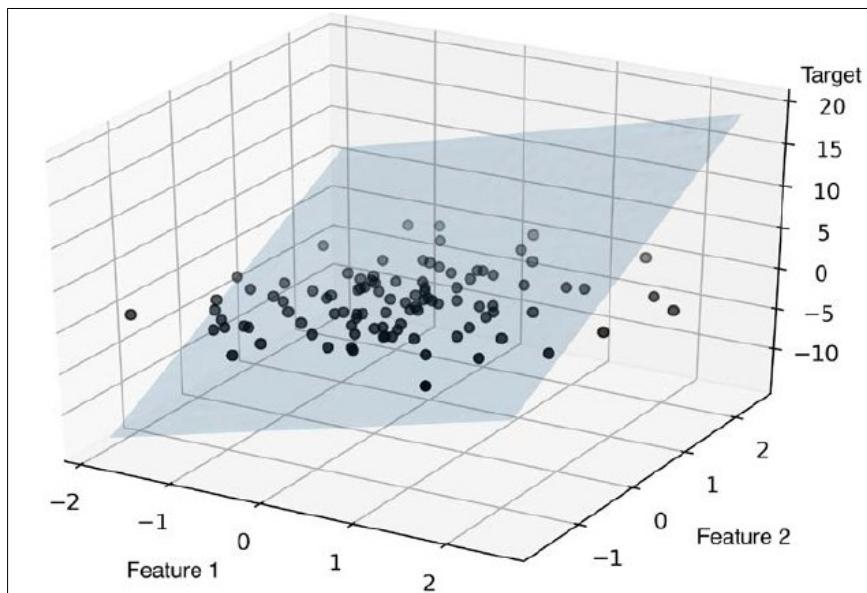


Figura 23. Plano de regresión cuando hay dos predictoras.

En el caso de 3 o más variables hablaríamos de *hiperplano*. En cualquier caso la línea recta que busca el modelo queda descrita por una ecuación que tiene un peso o parámetro que multiplica a cada variable predictora. Este peso indica la pendiente o lo inclinado que está el hiperplano respecto de cada dimensión. Además se suma a otro único valor numérico que se denomina **bias** o **término de intercepción** o corte y que define la altura por la que corta al eje *target*.

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_p x_p$$

- $\hat{y}$  es el valor que predice el modelo.
- $p$  es el número de variables predictoras.
- $x_j$  es el valor de la  $j$ -ésima característica predictora del ejemplo.
- $\theta_j$  es el  $j$ -ésimo peso o parámetro del modelo (incluyendo el término bias que es  $\theta_0$ ).

Esta fórmula puede reescribirse en forma vectorial y matricial de manera más concisa:

$$y = h_{\theta} x = \theta \cdot x$$

- $\theta$  es el vector de parámetros del modelo, que contiene el bias  $[\theta_0, \theta_1, \dots, \theta_p]^T$ .
- $X$  es la matriz  $n \times p$  ( $n$  ejemplos de  $p$  características cada uno) y una primera columna ( $x_0$ ) toda a valores 1 para poder realizar la multiplicación.
- $\theta \cdot X$  es el producto dot de los vectores  $\theta$  y  $X$ .
- $h_{\theta}$  es la función hipótesis que usa los parámetros del modelo.

**¿Cómo encontrar los parámetros del hiperplano?** La manera es minimizar las distancias (*offsets*) de los puntos a la recta. Para ello se calcula la suma de distancias (*SSE*) o la media (*MSE*) o la raíz de la media (*RMSE*). Es decir, se mide como de lejos se queda la recta de los puntos utilizando una métrica y se minimiza esa métrica.

Uno de los métodos más usados (hay más) para minimizar funciones se llama **OLS (Ordinary Less Squared)** que deriva la función de coste (o **función Loss**  $L(\theta)$ ). Si el **Loss** usado es el *SSE* (la suma de los cuadrados del error):

$$L(\theta) = SSE(\theta) = \sum_{i=1}^n (y^{(i)} - \theta^T x^{(i)})^2 = (y - X\theta)^T (y - X\theta) = \|y - X\theta\|_2^2$$

Minimizando esta fórmula se puede obtener una solución analítica (una fórmula llamada **La ecuación normal**) o se puede calcular el gradiente de la función de coste para emplearlo en un método iterativo (el descenso por gradiente):

$$\theta_{OLS} = (X^T X)^{-1} X^T y \quad (\text{Ecuación normal})$$

El gradiente (un vector de derivadas parciales) que se usa en el método iterativo:

$$\frac{\partial L(\theta, X, y)}{\partial \theta} = 2 \sum_{i=1}^n x^{(i)} (x^{(i)} \theta - y^{(i)})^2$$

*Scikit learn* ofrece muchos modelos para aprendizaje supervisado y todos ellos tienen la misma API, la misma manera de usarlos desde programas. Ofrecen estos métodos:

- `modelo = Estimador()` # Nombre del objeto que queremos usar
- `modelo.fit(X, y)` # Entrenar pasando predictoras (X) y etiquetas (y)
- `prediccion = modelo.predict(X)` # Realizar una predicción de los ejemplos X

## LA ECUACIÓN NORMAL

Derivando el *SSE* e igualando a 0, podemos despejar el vector  $\theta$  para tener una fórmula denominada la ecuación normal que nos dice como calcular los parámetros que minimizan el error:

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

donde:

- $\theta$  es el vector de pesos o parámetros que minimiza la función de coste (el SSE).
- $y$  es el vector de valores *target* con valores  $y^{(1)}$  hasta  $y^{(n)}$ .

**EJEMPLO 4:** generar datos de 2 dimensiones relacionados linealmente y calcular la recta usando la ecuación normal.

```
import numpy as np
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)
```

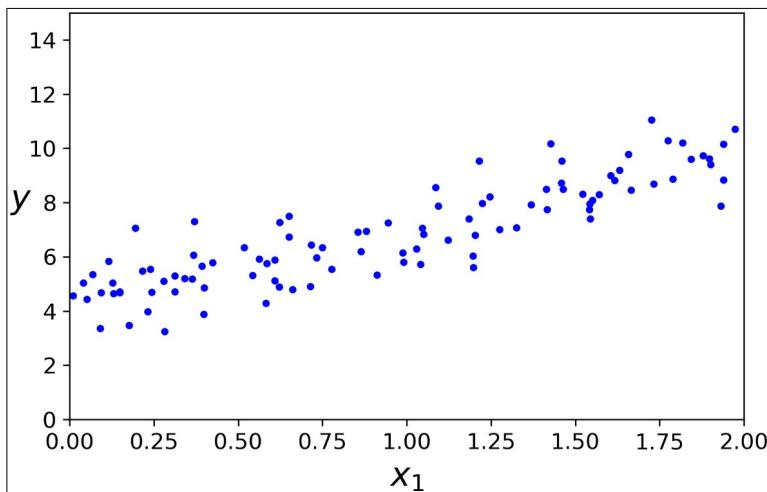


Figura 24: 100 datos aleatorios para calcular la recta de regresión lineal.

Usamos la función `inv()` del módulo de álgebra lineal de numpy (`np.linalg`) para calcular la inversa de la matriz y el método `dot()` para multiplicar las matrices y vectores:

```
X_b = np.c_[np.ones((100, 1)), X]           # añadir una columna x0 = 1 a cada ejemplo
pesos = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
print(pesos)
```

Los datos de  $y$  se han generado como  $y = 4 + 3x_1 + \text{ruido}$ . Y la ecuación ha encontrado:

```
array([[4.21509616],
       [2.77011339]])
```

No podemos encontrar  $\theta_0 = 4$  y  $\theta_1 = 3$  que son los valores exactos usados porque al introducir ruido se hace imposible, aunque nos hemos quedado bastante cerca ( $\theta_0 = 4.215$  y  $\theta_1 = 2.770$ ). Ahora vamos a realizar predicciones usando  $\theta$ :

```
X_nueva = np.array([[0], [2]])
X_nueva_b = np.c_[np.ones((2, 1)), X_nueva] # añadir x0 = 1 a cada ejemplo
y_prediccion = X_nueva_b.dot(pesos)
print(y_prediccion)

array([[4.21509616],
       [9.75532293]])
```

Dibujamos:

```
plt.plot(X_nueva, y_prediccion, "r-")
plt.plot(X, y, "b.")
plt.axis([0, 2, 0, 15])
plt.show()
```

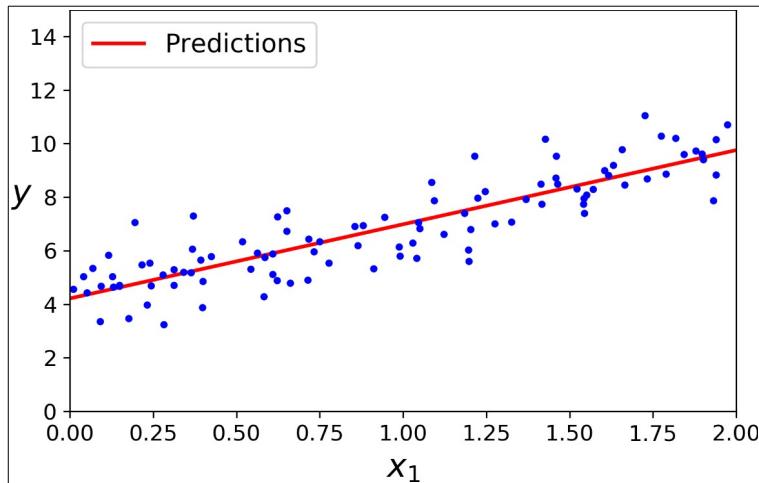


Figura 25: Predicciones de la Regresión lineal.

Como casi siempre, `scikit-Learn` también implementa el algoritmo de regresión lineal de fábrica y usarlo es bastante sencillo. El objeto separa el término **bias** (`intercept_`) del resto de pesos (`coef_`).

```
from sklearn.linear_model import LinearRegression
rl = LinearRegression()
rl.fit(X, y)
print(f"y={rl.intercept_} + {rl.coef_}x")
print(f"Predicción de {X_nueva} = {rl.predict(X_nueva)}")
```

La clase `LinearRegression` se basa en la función `scipy.linalg.lstsq()` (mínimos cuadrados) que también podemos usar directamente:

```
pesos_svd, residuos, rank, s = np.linalg.lstsq(X_b, y, rcond=1e-6)
print(pesos_svd)
```

Esta función calcula  $\theta = \mathbf{X}^+ \mathbf{y}$ , donde  $\mathbf{X}^+$  es la *pseudoinversa* de  $\mathbf{X}$  (la inversa de Moore-Penrose). Puedes usar `np.linalg.pinv()` para calcularla directamente:

```
np.linalg.pinv(X_b).dot(y)
```

La propia *pseudoinversa* se calcula usando una técnica de factorización de matrices llamada descomposición de valores singulares (**SVD**) que rompe la matriz  $\mathbf{X}$  en una multiplicación de 3 matrices llamadas  $\mathbf{U} \Sigma \mathbf{V}^T$  (mira la documentación de `numpy.linalg.svd()`). La pseudoinversa se calcula como  $\mathbf{X}^+ = \mathbf{V} \Sigma^+ \mathbf{U}^T$ . Para calcular la matriz  $\Sigma^+$ , el algoritmo usa  $\Sigma$  y deja a cero todos los valores más pequeños de un límite y reemplaza los valores no 0 por su inversa, finalmente transpone la matriz resultante. Esta aproximación es más eficiente que calcular la ecuación normal y además soluciona el problema de que la matriz  $\mathbf{X}^T \mathbf{X}$  no sea invertible (si es singular), como ocurre cuando  $n < p$  o si alguna característica es redundante, porque la pseudoinversa siempre está definida.

La ecuación normal necesita calcular la inversa de la matriz  $\mathbf{x}'\mathbf{x}$ , que es una matriz  $(p + 1) \times (p + 1)$  (donde  $p$  es el número de predictoras). La complejidad computacional<sup>4</sup> de invertir una matriz normalmente está entre  $O(n^{2.4})$  y  $O(n^3)$  (según la implementación). Es decir, si doblas la cantidad de características, multiplicas por  $2^{2.4} = 5.3$  o por  $2^3 = 8$  el esfuerzo en realizarla.

La aproximación basada en **SVD** que usa la clase `LinearRegression` de Scikit-Learn tiene  $O(n^2)$ . Si doblas las características multiplicas el esfuerzo por 4. Pero cuando hay muchas predictoras (más de 2000), tanto la ecuación normal o incluso la aproximación basada en **SVD** son muy lentas. Por otro lado, que haya muchos ejemplos no les afecta negativamente  $O(n)$  si pueden estar en RAM.

Una vez entrenado el modelo, las predicciones si son muy rápidas. La otra alternativa para calcular el modelo es usar el algoritmo del gradiente para minimizar el **MSE**. Pero eso lo reservaremos para su propio apartado.

<sup>4</sup> **Complejidad computacional:** esfuerzo en consumo de recursos (operaciones o memoria) necesarios para realizar una tarea.

## 5.1. MEDIR EL ERROR NUMÉRICO.

Si un modelo realiza una tarea de regresión numérica, el target tiene valores continuos. Por tanto, durante su entrenamiento y evaluación se puede medir la cantidad numérica de error que comete al predecir cada ejemplo ya etiquetado, calculando el error cometido como la diferencia entre su valor real (la etiqueta) y el valor que predice el modelo. Uniendo esos errores se calculan estas métricas:

- **Coeficiente de determinación R-cuadrado  $R^2$**  (métrica usada en fase de evaluación)
- **MSE y RMSE** (usadas en fase de entrenamiento y evaluación)
- **MAE** (usada en fase de entrenamiento y evaluación)

### MEDIA DE CUADRADOS DE ERROR (MSE) Y SU RAÍZ (RMSE)

MSE es la media de los cuadrados del error cometido por el modelo al predecir y RMSE es su raíz cuadrada. Un valor bajo de MSE o de RMSE significa que el funcionamiento del modelo con esos datos es bueno. La ventaja del RMSE es que las unidades coinciden con la del target. La desventaja es que hay que hacer una operación más para calcularla.

$$MSE(X, h_{\theta}) = \frac{1}{n} \sum_{i=1}^n (\theta^T x^{(i)} - y^{(i)})^2$$

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

Realiza el [ejercicio 2](#) de la relación de problemas.

### MEDIA DEL ERROR ABSOLUTO (MAE)

Es la media del valor absoluto de los errores cometidos al predecir con los datos usados.

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

### COEFICIENTE DE DETERMINACIÓN $R^2$ (R-CUADRADO)

Se utiliza mucho el  $R^2$  para evaluar como de bien hace su trabajo el modelo. Nos da una medida numérica de lo bien que ajusta los datos el modelo. El valor  $R^2$  nos indica que proporción de la varianza de la variable independiente (la columna target) ha aprendido a explicar el modelo. Es un valor entre 0 y 1. Cuanto más cercano a 1 esté, mejor funciona el modelo.

Para calcularlo, primero calculamos el **SST** (**suma total de cuadrados**). Se suman las diferencias de los cuadrados de cada valor *target* con la media de la columna *target*.

$$SST = \sum_{i=1}^n (y^{(i)} - \bar{y})^2$$

A continuación se calcula el **SSR** (**suma de cuadrados de residuos**), la suma del cuadrado de las diferencias entre lo que predice el modelo en cada ejemplo y el valor real del ejemplo:

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

Y el  $R^2$  se calcula como:

$$R^2 = 1 - \frac{SSR}{SST}$$

También tenemos **SSE** (**suma de cuadrados de la varianza explicada**) que también es conocido como suma de cuadrados de varianza registrada. Es la suma de los cuadrados de la diferencia entre cada predicción del modelo con respecto a la media del target:

$$SSE = \sum_{i=1}^n (\hat{y}^{(i)} - \bar{y})^2$$

La relación entre el SST, el SSR y el SSE viene dada por esta otra fórmula:

$$SST = SSE + SSR$$

Esta siempre se cumple (puede servir además de para saber lo bien que funciona el modelo, en caso de que programemos nosotros el código que hace los cálculos, si los hemos programado bien o no).

También nos da una forma alternativa de calcular el  $R^2$ :

$$R^2 = 1 - \frac{SSR}{SST} = 1 - \frac{SST - SSE}{SST} = 1 - \frac{SST}{SST} + \frac{SSE}{SST} = 1 - 1 + \frac{SSE}{SST} = \frac{SSE}{SST}$$

**EJEMPLO 5:** Hemos fabricado un modelo de regresión lineal que intenta predecir la nota que obtienen los alumnos en un control según las horas que hayan estudiado para realizarlo.

	y	$\hat{y}$	$(y_i - \bar{y})^2$	$\sum (\hat{y}_i - \bar{y})^2$	
Hours_Studied	Test_Grade	Test_Grade_Pred	SST	SSR	
2	57	59.71111	518.8272	402.6711	
3	66	64.72778	189.8272	226.5025	
4	73	69.74444	45.93827	100.6678	
5	76	74.76111	14.2716	25.16694	
6	79	79.77778	0.604938	0	
7	81	84.79444	1.493827	25.16694	
8	90	89.81111	104.4938	100.6678	
9	96	94.82778	263.1605	226.5025	
10	100	99.84444	408.9383	402.6711	

**Nota:** la columna SSR que aparece en la imagen se refiere al SSRegistrado, lo que yo he denominado SSExplicado para evitar la confusión. En otros textos llaman a SSR (residuos) errores (SSE) y al explicado (SSE) lo llaman SSR. Por tanto tened cuidado con la nomenclatura que se utilice.

media de puntuaciones (Test\_grade) = 79.77

$SSE = 1510.01$   
 $R^2 = \frac{SSE}{SST} = \frac{1510.01}{1547.55} = 0.97$  Lo que significa que el modelo ha aprendido el 97% de la varianza → horas estudiadas explican el 97% de la nota obtenida

**EJEMPLO 6:** Calcular métricas de funcionamiento a un modelo de regresión lineal:

```
# -*- coding: utf-8 -*-
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import r2_score, mean_absolute_error, mean_squared_error
from sklearn.linear_model import LinearRegression

# Generar datos lineales: y = 2 + 3x + epsilon
np.random.seed(0)
n = 100
x = np.sort(5 * np.random.rand(n,1), axis=0)
y = 2 + 3 * x + np.random.randn(n, 1)
# Crear un modelo lineal
modelo = LinearRegression().fit(x,y)
y_hat = modelo.predict(x)
plt.scatter(x,y)
```

```

plt.plot(x, y_hat, color='b')
plt.show()
# Calcular errores manualmente
sst = np.square( y - y.mean() ).sum()
ssr = np.square( y_hat - y ).sum()
sse = np.square( y_hat - y.mean() ).sum()
print('----Manualmente:\nSST: ', sst)
print('SSR: ', ssr)
print('SSE: ', sse)
print('R2: ', sse / sst)
# Usando funciones preconstruidas
print('----Preconstruidas\nMAE: ', mean_absolute_error(y, y_hat))
mse = mean_squared_error(y, y_hat)
print('MSE: ', mse)
print('RMSE:', np.sqrt(mse))
print('R2: ', r2_score(y, y_hat) )

```

## 5.2. DESCENSO POR GRADIENTE.

Es un algoritmo de optimización muy genérico capaz de encontrar soluciones óptimas a un gran abanico de problemas. Su enfoque es ir cambiando parámetros de manera iterativa para conseguir minimizar o maximizar una función de coste (también llamada a veces de **Loss** o de error).

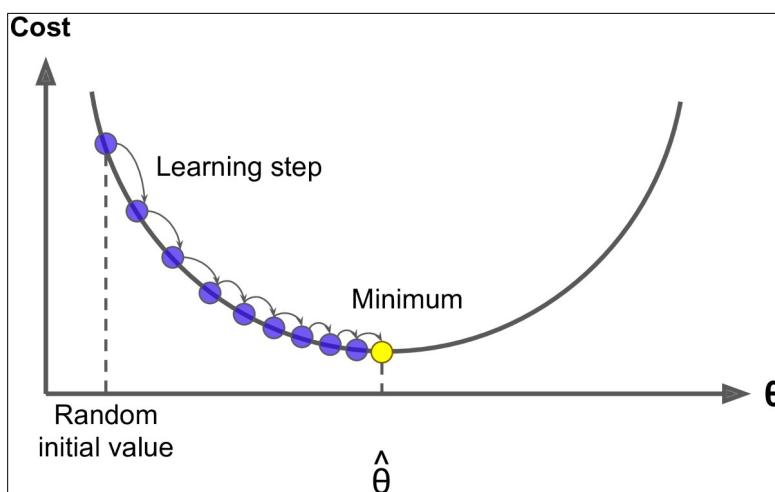


Figura 26: Pasos realizados por el algoritmo descenso por gradiente.

Para comprender como funciona, imagina que estás en una zona montañosa donde hay una densa niebla que no te permite ver más allá de 5 metros de donde estás. Si quieres salir de la montaña una buena estrategia es bajar lo más rápidamente posible. Para hacerlo, donde estás observas donde está la mayor pendiente y vas en dirección contraria, en el nuevo lugar repites la operación una y otra vez, y en cada paso posiblemente vayas descendiendo.

Pues esto es lo que hace el gradiente: la zona montañosa la define la función de coste, en la posición actual (la posición son ciertos valores de los parámetros, el vector  $\theta$ ) el algoritmo calcula la dirección de mayor pendiente (el gradiente) y da un paso en dirección contraria. Y vuelve a repetir una y otra vez el proceso hasta que alcanza un número máximo de pasos, alcanza una altura razonable o los pasos que da ya no bajan lo suficiente como para seguir caminando. Estas 3 situaciones suelen utilizarse como condición de parada de forma aislada o combinada.

Un parámetro muy importante en el funcionamiento del algoritmo es el tamaño de los pasos que da, es lo que controla el hiperparámetro **learning rate**. Si es demasiado pequeño el algoritmo tardará mucho hasta encontrar el mínimo, puede que agote su cantidad máxima de pasos antes de poder tener la oportunidad de encontrarlo tal y como ocurre en la figura 27.

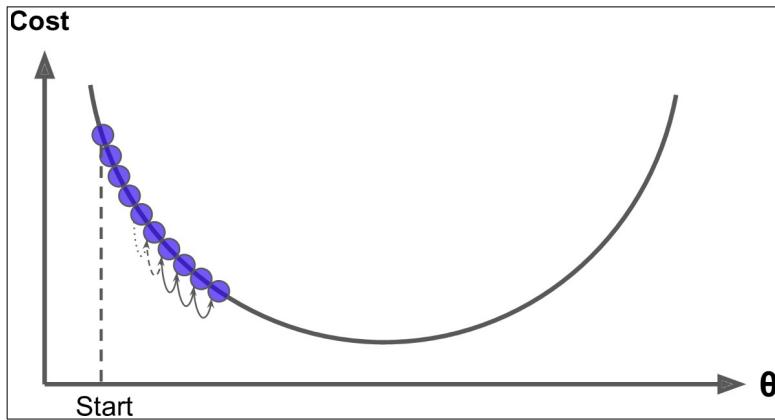


Figura 27: Learning rate demasiado pequeño.

Por otro lado, si el *learning rate* es demasiado grande puede ocurrir que cuando esté cerca del mínimo, de un paso tan grande que vuelva a alejarse de él y en vez de descender haga justo lo contrario, o a veces descienda y a veces ascienda. Y finalmente agote sus máximos pasos sin encontrar un lugar bajo. Es lo que ocurre en la figura 28.

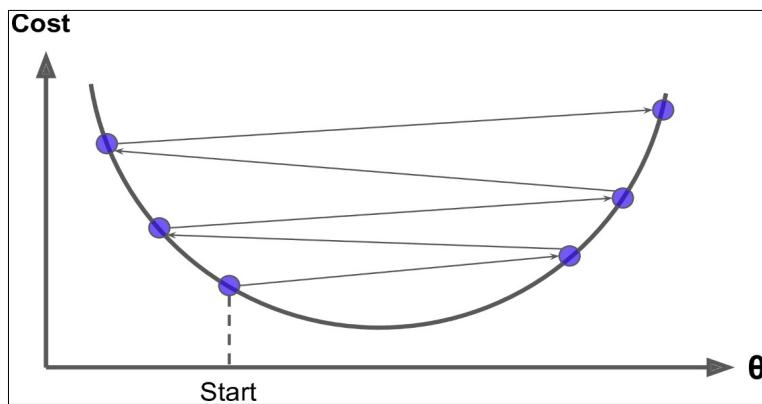


Figura 28: Learning rate demasiado grande.

Finalmente, no todas las funciones de coste tienen superficies que faciliten alcanzar un mínimo. Las mejores son las denominadas convexas, como la que aparece en las figuras 25 a 28, que son paráolas, y comiences donde comiences si vas hacia abajo, vas hacia un mínimo global. Es lo que ocurre con el SSE, MSE y RMSE, que al aplicar un cuadrado, son paráolas.

Sin embargo otras, pueden tener valles, crestas, mínimos locales, o zonas muy llanas y con poca o ninguna pendiente, de forma que si el algoritmo se sitúa en una zona complicada quizás quede atrapado y no sea capaz de encontrar una buena solución como se muestra en la figura 29. Pero como hemos mencionado, la función de coste SSE y MSE son convexas y no presentan estas irregularidades, solo hay un mínimo local, son continuas y derivables y siempre tienen gradiente en cualquier lugar.

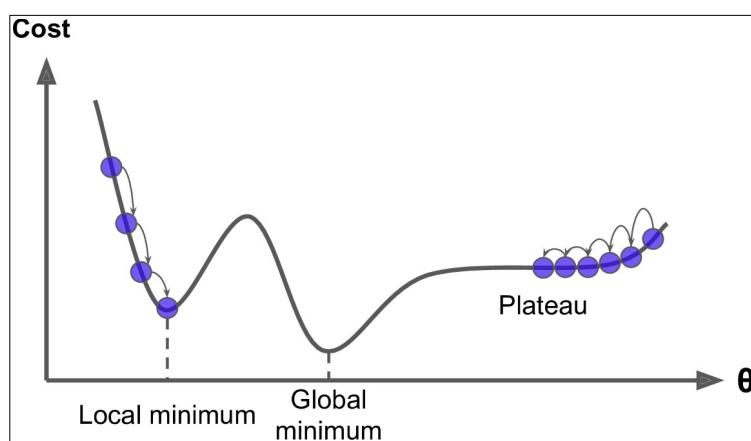
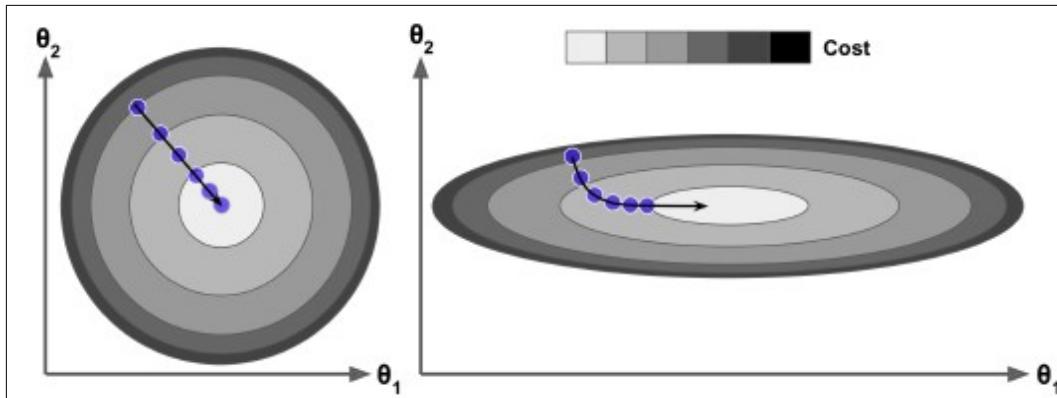


Figura 29: Fallos del descenso por gradiente debido al terreno (función de coste).

Si trazamos isobaras de coste tendría la apariencia de una diana circular o en caso de que haya una dimensión con valores muy diferentes al resto, parecería una diana estirada en esa dimensión. Estos estiramientos también ponen en dificultad al algoritmo de descenso por gradiente, porque aunque en unos pasos en muchas variables avanza veloz hacia una zona con bajo error, en otras le costará más y quizás pueda incluso agotar sus pasos sin bajar del todo en estas dimensiones. La figura 30 tiene el gráfico de la derecha donde los valores de la característica  $\theta_1$  tienen un rango más alargado que los de  $\theta_2$  y esto hace que el gradiente falle o no consiga tan buen resultado como en el de la izquierda. O incluso aunque consiga alcanzar el mínimo, le costará más pasos y por tanto más tiempo conseguirlo.



*Figura 30: Descenso por gradiente con y sin escalado de características.*

Por este motivo, es conveniente escalar o estandarizar los valores de las variables cuando se utilice este algoritmo iterativo. Ten en cuenta que no es raro entrenar modelos con miles de dimensiones.

Una vez comprendido el funcionamiento, examinamos varias modalidades de aplicarlo, cada una con sus ventajas e inconvenientes.

### DESCENSO POR GRADIENTE BATCH

En esta variante del algoritmo, antes de dar un paso, es necesario hacer una predicción para cada ejemplo de entrenamiento y medir el error cometido. Una vez acumulados todos los errores se calcula la métrica SSE o MSE y con ella se calcula el gradiente en los parámetros actuales y se decide la dirección a tomar en el paso actual.

$$\frac{\partial}{\partial \theta_j} \text{MSE}(\theta) = \frac{2}{m} \sum_{i=1}^m (\theta^T \mathbf{x}^{(i)} - y^{(i)}) x_j^{(i)}$$

El gradiente es un valor para cada parámetro  $\theta_j$  del modelo. Todos estos valores forman un vector. Para calcular cada uno se utiliza la derivada parcial de la función de coste usada. En la fórmula de abajo aparece la fórmula de la derivada parcial del parámetro  $\theta_j$  si usamos el MSE.

En vez de calcular cada derivada parcial de manera individual, puedes usar la ecuación de la derecha para calcularlas todas a la vez mediante una fórmula usando matrices. El vector gradiente  $\nabla_{\theta} \text{MSE}(\theta)$ , contiene todas las derivadas parciales de la función de coste (una por parámetro).

Esta fórmula implica realizar cálculos con todo el conjunto de datos de entrenamiento  $X$ . **!En cada paso del descenso por gradiente!** Por eso esta variante recibe el nombre de batch.

$$\nabla_{\theta} \text{MSE}(\theta) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} \text{MSE}(\theta) \\ \frac{\partial}{\partial \theta_1} \text{MSE}(\theta) \\ \vdots \\ \frac{\partial}{\partial \theta_n} \text{MSE}(\theta) \end{pmatrix} = \frac{2}{m} X^T (X\theta - y)$$

Como consecuencia, si hay muchos datos es muy lento. Sin embargo, soporta bien aumentar el número de características, por lo que es más rápido que usar la ecuación normal o la descomposición SVD cuando hay muchas predictoras.

Una vez que tienes el vector gradiente, sus coordenadas apuntan cuesta arriba, basta con invertir su dirección con un signo para que señalen hacia abajo. Así que hay que restar a  $\theta$  (la posición actual) el gradiente ( $\nabla_{\theta} \text{MSE}(\theta)$ ) multiplicado por el learning rate ( $\eta$ ) quedando la ecuación como:

$$\theta^{(\text{next step})} = \theta - \eta \nabla_{\theta} \text{MSE}(\theta)$$

El learning rate determina a los parámetros a los que se mueve el algoritmo en cada nuevo paso. En la figura 30 vemos la evolución del modelo cuando solamente cambiamos el learning rate. La línea punteada representa el valor inicial de los parámetros. Una implementación rápida del algoritmo asumiendo que tenemos los datos preparados en  $X_b$  y las etiquetas en  $y$ :

```
eta = 0.1 # Learning rate
n_iteraciones = 1000
n = 100
theta = np.random.randn(2,1) # inicialización aleatoria de theta
for iteration in range(n_iteraciones):
    gradientes = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradientes
print(theta) #
```

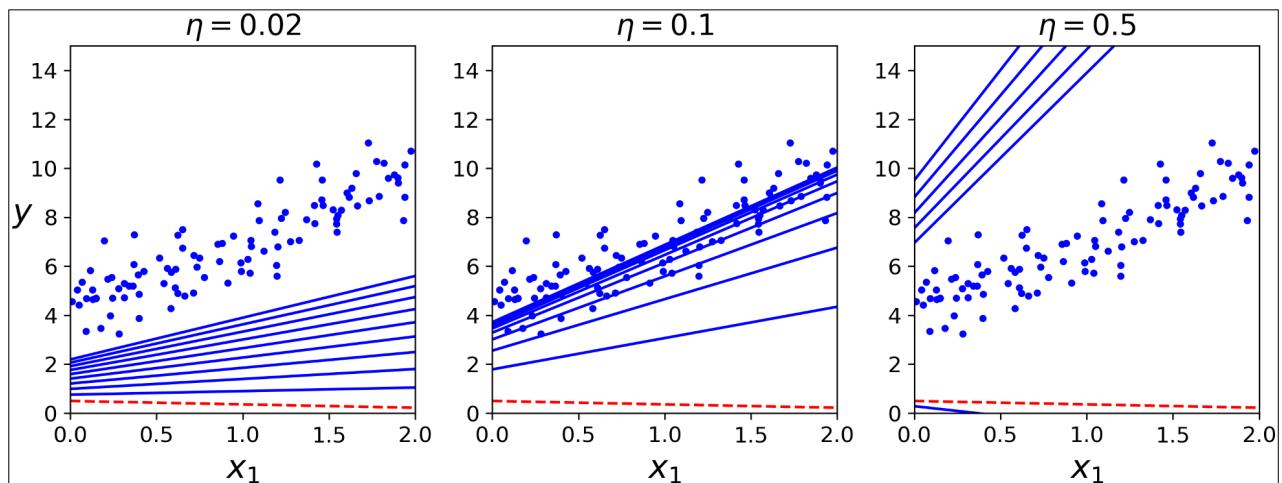


Figura 31: Descenso por Gradiente con diferentes valores de learning rate.

A la izquierda el *learning rate* es demasiado bajo y el algoritmo tarda mucho en encontrar una solución. El caso del centro parece bueno, en pocas iteraciones se acerca a la solución. El caso de la derecha tiene un *learning rate* muy alto y salta la solución en uno de los pasos.

Para encontrar un buen valor para el learning rate puedes usar el método *grid search* y también puedes limitar el número de iteraciones para que deseche modelos que tarden demasiado en converger.

Una solución más sencilla para encontrar un valor adecuado es comenzar con un valor alto e ir bajándolo buscando manualmente el valor que consiga bajar el error y que tarde lo mínimo posible.

### 5.3. CANTIDAD DE EJEMPLOS Y RATIO DE CONVERGENCIA

Cuando la función de coste es convexa y la pendiente no cambia abruptamente (como usando *MSE*), la variante *Batch* del descenso por gradiente con un *learning rate* fijo tiene posibilidades de converger a una solución óptima. Si necesita un  $O(1/\epsilon)$  de iteraciones para alcanzar la solución en el rango de  $\epsilon$  y lo divides por 10 para encontrar una mejor solución, tendrás que multiplicar por 10 las iteraciones.

## DESCENSO POR GRADIENTE ESTOCÁSTICO

El principal problema de la variante batch es que necesita procesar todos los datos de entrenamiento en cada paso y eso lo ralentiza si hay muchos ejemplos. La variante estocástica<sup>5</sup> es el otro extremo, en cada paso solamente escoge un ejemplo aleatorio para calcular el gradiente con él. Esto hace que sea muy rápido porque procesa la mínima cantidad de datos posibles en cada iteración. Esto permite utilizarlo en conjuntos de datos de entrenamiento enormes porque solo se necesita un ejemplo en la RAM para cada iteración (se puede implementar como un **algoritmo out-of-core**<sup>6</sup>).

Este algoritmo también es más irregular debido a su naturaleza aleatoria, cambia de dirección de manera más errática que la variante batch, cuando se acerca a la solución continúa acercándose y alejándose y difícilmente acaba en una solución tan óptima como el batch, así que será una solución buena pero no óptima.

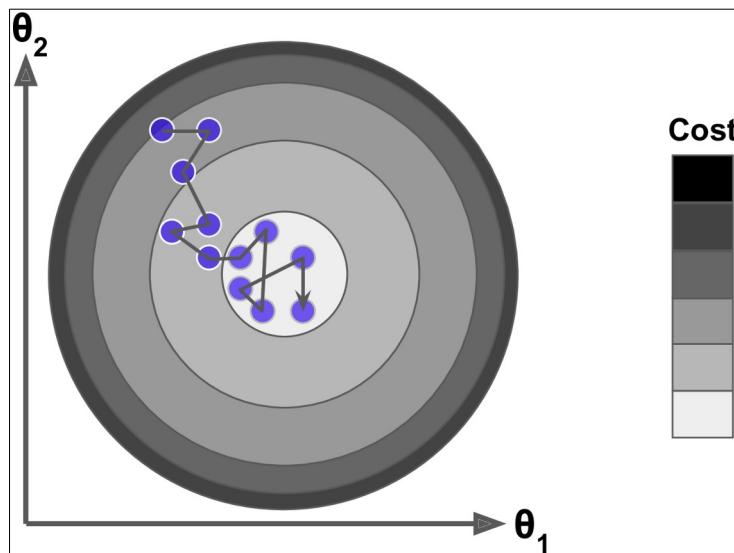


Figura 32: Descenso por gradiente estocástico.

Cuando la función de coste es muy irregular, su naturaleza puede ayudarle a salir de un mínimo local, en estos casos tiene mejores expectativas de alcanzar una solución cercana al mínimo global. Así que su naturaleza le permite acercarse al mínimo global pero le impide quedarse cerca de él. La solución a este comportamiento es no usar un *learning rate* fijo sino que vaya disminuyendo a medida que se van dando pasos, con cada nuevo paso decrece, de forma que inicialmente los pasos son grandes y van haciéndose más pequeños cada vez. Este proceso se denomina *simulated annealing* y la función que determina como va cambiando el *rate learning* se denomina **learning schedule**.

```

n_epocas = 50
t0, t1 = 5, 50          # hiperparámetro Learning schedule
def learning_schedule(t):
    return t0 / (t + t1)
theta = np.random.randn(2,1)      # inicialización aleatoria
for epoca in range(n_epocas):
    for i in range(n):
        indice_random = np.random.randint(n)
        xi = X_b[indice_random:indice_random + 1]
        yi = y[indice_random:indice_random + 1]
        gradientes = 2 * xi.T.dot(xi.dot(theta) - yi)
        eta = learning_schedule(epoca * n + i)
        theta = theta - eta * gradientes
print(theta)

```

5 Estocástico: sinónimo de aleatorio.

6 Algoritmos out-of-core: cargan en RAM parte de los datos, procesan algunos y vuelven a repetir el proceso.

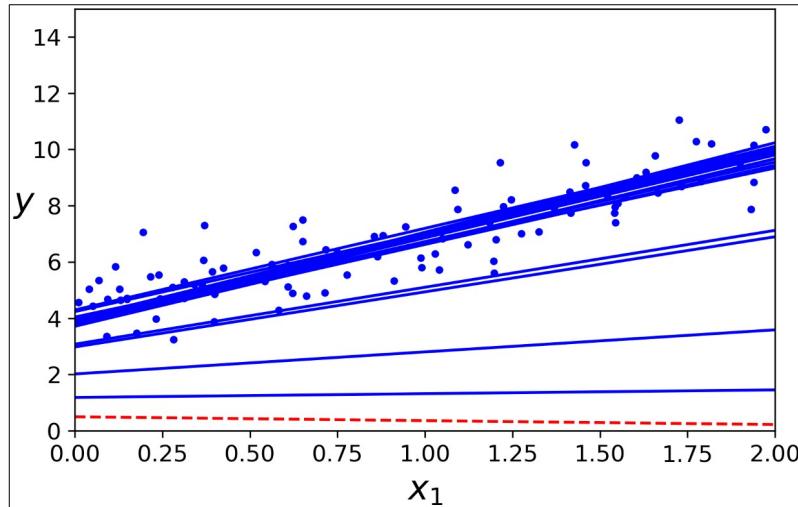


Figura 33: descenso por gradiente estocástico tras 20 pasos.

Como los ejemplos a procesar se escogen aleatoriamente puede ocurrir que en la misma época se escoja el mismo varias veces y otros nunca se usen. Si quieres evitarlo puedes desordenar los datos y sus etiquetas en cada época e ir usando uno a uno, aunque esto ralentiza el funcionamiento. Si se usa esta variante los ejemplos de entrenamiento deben ser independientes y estar distribuidos de manera idéntica.

Para realizar regresión lineal con Scikit-Learn, con la variante estocástica puedes usar la clase **SGDRegressor**, que por defecto optimiza la función de coste SSE. El siguiente código ejecuta un máximo de 1000 épocas y también para cuando no baja el coste durante una época más de 1e-3 (`tol=1e-3`), comenzando con un learning rate de 0.1 (`eta0=0.1`) y un learning schedule y no usa ninguna regularización (`penalty=None`):

```
from sklearn.linear_model import SGDRegressor
sr = SGDRegressor(max_iter=1000, tol=1e-3, penalty=None, eta0=0.1)
sr.fit(X, y.ravel())
print(sr.intercept_, sr.coef_)
```

La fórmula de la variante batch (usa todos los ejemplos):

$$\Delta \mathbf{w} = \eta \sum_i (y^{(i)} - \phi(z^{(i)})) \mathbf{x}^{(i)}$$

Y la de la variante estocástica (solamente usa un ejemplo escogido al azar):

$$\eta (y^{(i)} - \phi(z^{(i)})) \mathbf{x}^{(i)}$$

El estocástico puede utilizarse en aprendizaje *online*.

### DESCENSO POR GRADIENTE MINI-BATCH

La última variante es una mezcla de las dos anteriores, ni usa todos los datos, ni solamente usa uno. Para calcular la dirección del paso procesa un grupo de ejemplos llamado mini-batch, es decir, crea pequeños grupos de ejemplos de manera aleatoria llamados mini-batch y los procesa para calcular el gradiente.

La principal ventaja sobre el estocástico es que se aprovecha de mejoras del hardware para procesar matrices especialmente las GPU. Además es menos errático cuando los mini-batch son suficientemente grandes. También le ayuda usar learning schedule. La figura 34 compara las trayectorias seguidas por las 3 variantes para el mismo problema de regresión lineal.

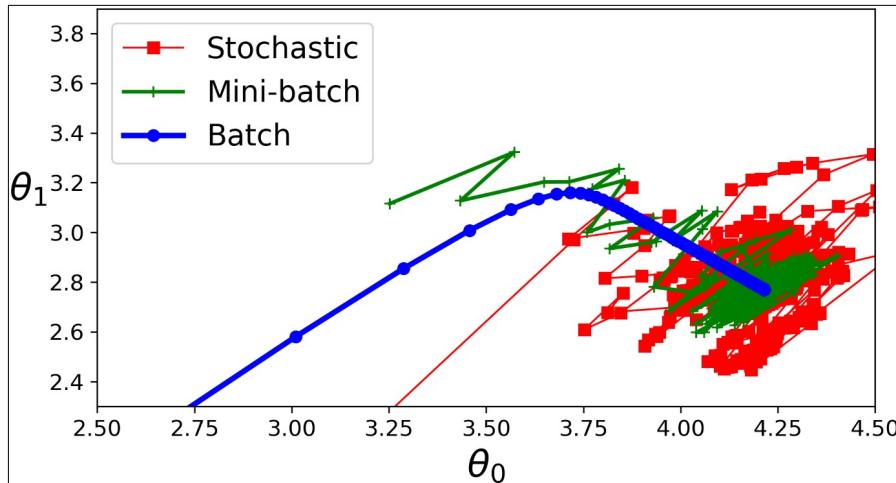


Figura 34: Direcciones usadas por las 3 variantes del descenso por gradiente.

En esta tabla se recogen las características de cada algoritmo que realiza regresión lineal numérica. La  $m$  representa la cantidad de ejemplos y la  $n$  la cantidad de predictoras. La última columna contiene el nombre de la clase que implementa el algoritmo en *scikit-learn*.

Algorithm	Large $m$	Out-of-core support	Large $n$	Hyperparams	Scaling required	Scikit-Learn
Normal Equation	Fast	No	Slow	0	No	n/a
SVD	Fast	No	Slow	0	No	LinearRegression
Batch GD	Slow	No	Fast	2	Yes	SGDRegressor
Stochastic GD	Fast	Yes	Fast	$\geq 2$	Yes	SGDRegressor
Mini-batch GD	Fast	Yes	Fast	$\geq 2$	Yes	SGDRegressor

## 5.4. MEJORAS PARA COSTES NO CONVEXOS.

Utilizar la variante mini-batch del descenso por gradiente tampoco garantiza una buena convergencia porque mantiene algunos peligros que debemos solucionar:

- **Escoger un valor apropiado del *learning rate* puede ser difícil.** Si es demasiado pequeño la convergencia será lenta mientras que si es demasiado grande la función de coste fluctúa (sube y baja) o incluso diverge (crece).
- **Utilizar *learning rate schedules* para cambiar el *learning rate* durante el entrenamiento** usando *annealing* por ejemplo, para reducir el *learning rate* con una variación predefinida o cuando cambie el objetivo entre épocas usando valores límite. Estas modificaciones y límites se definen antes y esto hace que no se puedan adaptar a las características del dataset.
- **El mismo *learning rate* se aplica a la actualización de todos los parámetros.** Si nuestros datos están muy esparcidos (*sparse*) y las características tienen frecuencias muy distintas, sería interesante no actualizarlas todas la misma extensión, porque realizar una actualización importante raramente ocurrirá en estas características.
- **Minimizar error en funciones de coste no convexas:** la función de coste *SSE* y *MSE* son convexas, pero en redes neuronales aparecen funciones de coste que no lo son. Esto hace que tengan irregularidades mínimos y máximos locales aparte del mínimo global, mesetas, etc.

En este apartado comentamos algunas mejoras que se aplican sobre todo en deep learning para solucionar estos desafíos.

### OPTIMIZACIÓN DE MOMENTUM

Imagina un balón que se deja rodar colina abajo por una suave pendiente: comienza a moverse lentamente pero rápidamente comienza a ganar momento (en física sería masa \* velocidad) hasta que

alcanza una velocidad terminal (la resistencia al aire y el rozamiento con el suelo impiden que aumente indefinidamente su velocidad). Esta es la idea básica de esta optimización propuesta por Boris Polyak en 1964.

El descenso por gradiente regular da pequeños pasos superficie abajo actualizando sus pesos  $\theta$  restando el gradiente ( $\nabla J(\theta)$ ) de la función de coste  $J(\theta)$  en ese lugar multiplicado por el *learning rate*  $\eta$ . Recordamos la ecuación de la siguiente posición o paso que da el algoritmo:  $\theta \leftarrow \theta - \eta \nabla J(\theta)$ .

No tiene en cuenta los anteriores gradientes, solo utiliza el gradiente del lugar actual. Esto hace que si el gradiente actual es pequeño (poca pendiente) el paso será muy pequeño. La *optimización del momentum* se fija en los gradientes anteriores al actual porque mantiene un **vector momentum**  $m$  al que resta el gradiente actual antes de multiplicarlo por el *learning rate*. Es decir, en cada actualización de la posición se tiene en cuenta la inercia que se arrastra de las actualizaciones anteriores. Tiene en cuenta la aceleración histórica o la inercia de los movimientos anteriores para cambiar la posición. La ecuación sería:

1.  $m \leftarrow \theta m - \eta \nabla J(\theta)$
2.  $\theta \leftarrow \theta + m$

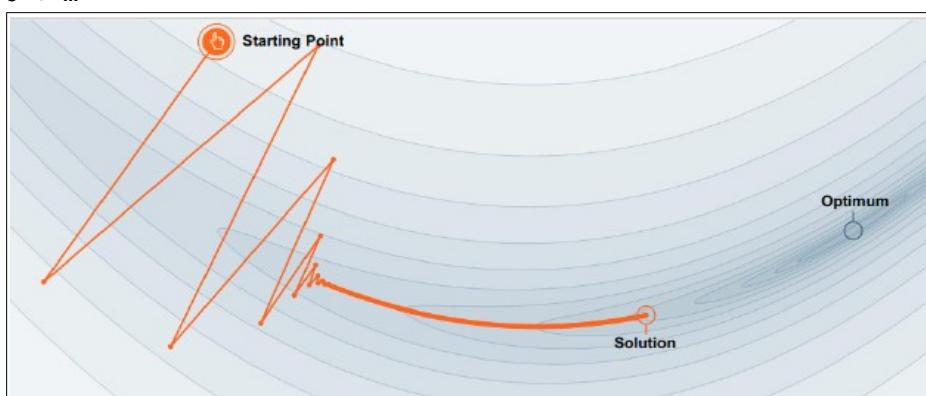


Figura 33: Trayectoria de gradiente sin momento.

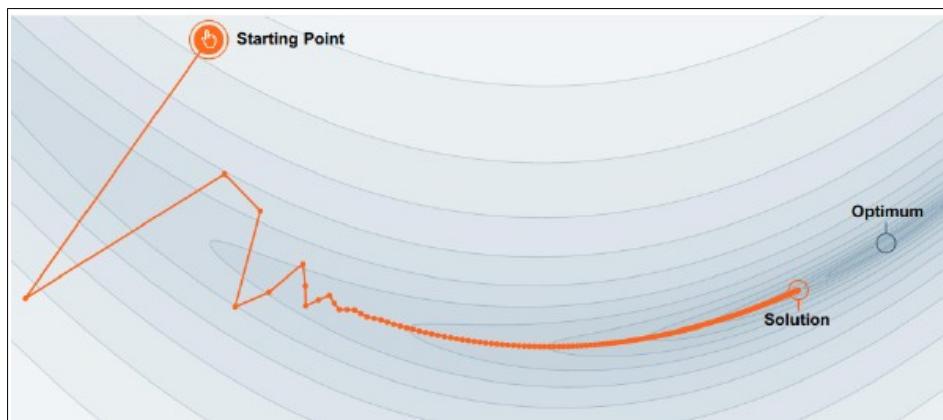


Figura 34: Con momento se reducen las oscilaciones y se acerca más al valor óptimo.

Para simular algún tipo de rozamiento aparece un nuevo hiperparámetro llamado  $\beta$ , que debe tomar valores entre 0 (alta fricción) y 1 (sin fricción). Un valor típico es 0.9. En las figuras 33 y 34 vemos trayectorias del algoritmo con los mismos datos y de fondo las isobaras de error que al no estar separadas por la misma distancia ya dan idea de que la superficie es irregular.

```

vx = 0
while True:
    dx = gradiente(J, x)
    vx = rho * vx + dx
    x -= learning_rate * vx
  
```

Si el gradiente permanece constante, la velocidad terminal (tamaño del paso) es el gradiente multiplicado por el *learning rate*  $\eta$  multiplicado por  $1/(1 - \beta)$ . Por ejemplo si  $\beta = 0.9$ , la velocidad

terminal será 10 veces el gradiente por el learning rate, así que se mueve 10 veces más rápido que el algoritmo normal. Y esto le permite escapar más rápidamente de mesetas y mínimos locales.

Implementar esta optimización en la práctica es tan sencillo como inicializar el hiperparámetro en la llamada al método de *Keras*<sup>7</sup> que usa como optimizador al algoritmo SGD:

```
optimizador = keras.optimizers.SGD(lr=0.001, momentum=0.9)
```

### OPTIMIZACIÓN GRADIENTE ACELERADO DE NESTEROV (NAG)

Es una variante de *optimización de momentum* propuesta por **Yuriii Nesterov** en 1983 que mejora aún más la velocidad de convergencia. También se conoce como *Nesterov Accelerated Gradient*, y consiste en medir el gradiente de la función de coste no en la posición actual sino ligeramente desplazado hacia la dirección del momento. La única diferencia con respecto a la *optimización de momentum* es que el gradiente se mide en la posición  $\theta + \beta m$  en vez de hacerlo en  $\theta$ .

1.  $m \leftarrow \theta - \eta \nabla \theta J(\theta + \beta m)$
2.  $\theta \leftarrow \theta + m$

Este truco da buen resultado porque el vector momento está apuntando a la dirección del paso a dar, así que medir el gradiente en esa dirección es más preciso que hacerlo en la posición actual. En la figura 35 puedes ver una representación donde  $\nabla_1$  representa el gradiente medido en la posición actual  $\theta$  y  $\nabla_2$  representa el gradiente calculado en la posición  $\theta + \beta m$ . NAG da más velocidad y consigue valores más cercanos al óptimo. Para usarlo simplemente activamos con el parámetro `nesterov=True` al crear el optimizador SGD que use *momentum*:

```
optimizador = keras.optimizers.SGD(lr=0.001, momentum=0.9, nesterov=True)
```

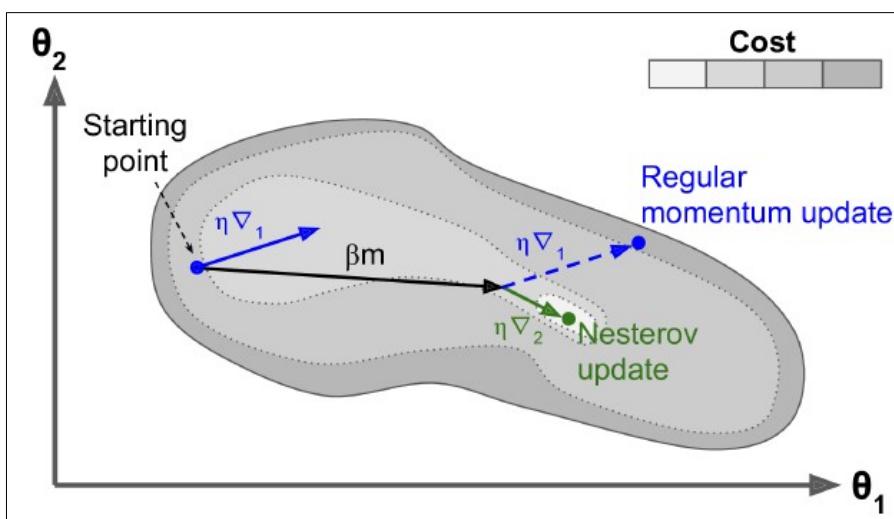


Figura 35: NAG es un poco más rápido que optimización de momento y da valores más óptimos.

### OPTIMIZACIÓN ADAGRAD: ADAPTATIVE LEARNING RATES

El descenso por gradiente comienza bajando a mucha velocidad pos las empinadas pendientes y lentamente cuando alcanza los valles. Estaría muy bien que el algoritmo pudiese detectar esto antes de que entre en un valle y corregir su dirección para dirigirse al mínimo global óptimo. El algoritmo **Adagrad** consigue esto escalando hacia abajo el vector del gradiente de manera diferente en cada dimensión antes de dar un paso.

Su ecuación:

1.  $s \leftarrow s + \nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$
2.  $\theta \leftarrow \theta - \eta \nabla_\theta J(\theta) \odot \sqrt{s + \epsilon}$

<sup>7</sup> **Keras**: librería de Python usada en deep learning (redes neuronales profundas).

El primer paso acumula los cuadrados de los gradientes en el vector  $\mathbf{s}$  (recuerda que el operador  $\otimes$  es la operación multiplicación elemento a elemento). Esta fórmula es la versión vectorizada de calcular  $s_i \leftarrow s_i + (\partial J(\theta) / \partial \theta_i)^2$  para cada elemento  $s_i$  del vector  $\mathbf{s}$ , es decir, cada elemento  $s_i$  del vector  $\mathbf{s}$  acumula los cuadrados de las derivadas parciales de la función de coste que pertenece al parámetro  $\theta_i$ . Si la función de coste es más pendiente en la  $i$ -ésima dimensión, entonces  $s_i$  será mayor en cada iteración.

El segundo paso es idéntico al descenso por gradiente pero con una gran diferencia: el vector gradiente es escalado en un factor de  $\sqrt{s + \epsilon}$  (el símbolo  $\oslash$  indica división elemento a elemento y el término  $\epsilon$  es un mecanismo de seguridad para evitar la división por cero normalmente establecido a un valor de  $10^{-10}$ ). Esta fórmula vectorizada equivale a calcular  $\theta_i \leftarrow \theta_i - \eta \partial J(\theta) / \partial \theta_i / \theta_i \leftarrow \theta_i - \eta \partial J(\theta) / \partial \theta_i / \sqrt{s + \epsilon}$  para todos los parámetros  $\theta_i$  (simultáneamente).

Si la pendiente es pequeña agranda el paso, si es grande lo reduce y otro beneficio adicional que tiene es que necesita menos pruebas para elegir un buen valor del *learning rate*  $\eta$  porque su funcionamiento compensa no elegir uno muy bueno.

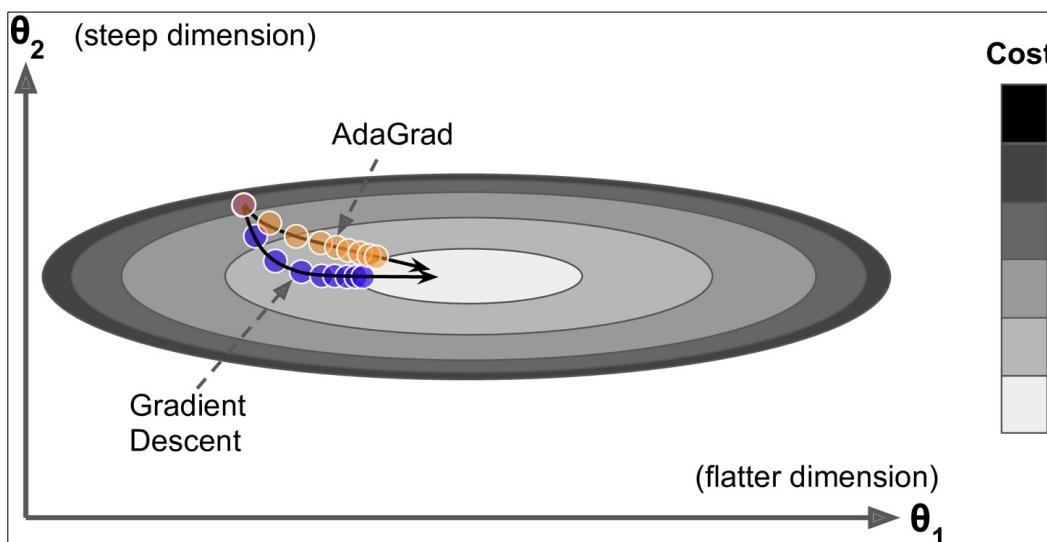


Figura 36: AdaGrad contra descenso por gradiente.

Funciona bien en problemas cuadráticos pero a veces es demasiado lento para entrenar a redes neuronales porque baja demasiado el learning rate y no da tiempo a alcanzar una solución antes de agotar los pasos. Además necesita memoria para guardar el acumulado de cada parámetro, si hay muchos parámetros... Keras tiene un optimizador *Adagrad* pero no deberías usarlo para entrenar redes neuronales profundas (puede ser eficiente para tareas más simples como una regresión lineal). Pero comprenderlo te ayudará a comprender otros optimizadores. Ejemplo de uso con *keras*:

```
optimizador = keras.optimizers.Adagrad(lr=0.001, epsilon=1e-07)
modelo = keras.models.Sequential([...])
```

El algoritmo sería algo así:

```
learning_rate = 0.001          # parámetros más importantes
epsilon = 1e-07
grad_cuadrados = 0            # vector de tantos elementos como parámetros
while True:
    dx = gradiente(J, x)
    grad_cuadrados += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_cuadrados) + epsilon)
```

### OPTIMIZACIÓN RMSPROP: LEAKY ADAGRAD

Aunque *AdaGrad* baja el paso demasiado rápido y nunca converge al óptimo global, el algoritmo *RMSProp* creado por Geoffrey Hinton y Tijmen Tieleman en 2012, fija esto acumulando solamente los gradientes de las iteraciones más recientes y esto lo hace aplicando un decrecimiento exponencial al primer paso.

$$1. \quad s \leftarrow \beta s + (1 - \beta) \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$$

$$2. \quad \theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta) \oslash \sqrt{s + \epsilon}$$

El ratio de decaimiento  $\beta$  normalmente se fija a 0.9. Suele ser el hiperparámetro ***decay\_rate*** y su valor por defecto funciona generalmente bien. Ejemplo de uso en Keras:

```
optimizador = keras.optimizers.RMSprop(lr=0.001, rho=0.9)
```

Excepto en problemas muy simples, funciona mejor que *AdaGrad*. Ejemplo de algoritmo:

```
decay_rate = 0.9      # Si es 0 es AdaGrad y si es 1 es el DGS
grad_cuadrados = 0
epsilon = 1e-7
while True:
    dx = gradiente(J, x)
    grad_cuadrados += decay_rate * grad_cuadrados + (1 - decay_rate) * dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_cuadrados) + 1e-7)
```

### OPTIMIZACIÓN ADAM: ADAPTATIVE MOMENT ESTIMATION

Fue creado en 2015 por D. Kingma y J. Ba. Combina las ideas de la ***optimización del momentum*** y ***RMSProp***: va calculando una media de decaimiento exponencial de los gradientes pasados y de los cuadrados de los gradientes. Son como estimaciones de la media (momento 1) y de la varianza (segundo momento). La fórmula que usa:

$$1. \quad m \leftarrow \beta_1 m - (1 - \beta_1) \nabla_{\theta} J(\theta)$$

$$2. \quad s \leftarrow \beta_2 s + (1 - \beta_2) \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$$

$$3. \quad \hat{m} \leftarrow \frac{m}{1 - \beta_1^t} \quad t \text{ representa el número de iteración.}$$

$$4. \quad \hat{s} \leftarrow \frac{s}{1 - \beta_2^t}$$

$$5. \quad \theta \leftarrow \theta + \eta \hat{m} \oslash \sqrt{\hat{s} + \epsilon}$$

Si te fijas en los pasos 1, 2 y 5 se parece a la optimización de momentum y a RMSProp. La única diferencia es que en el paso 1 calcula un decaimiento exponencial de la media en vez de un decaimiento exponencial de la suma (el decaimiento de la media es  $1 - \beta_1$  veces del de la suma). Los pasos 3 y 4 son algo más técnicos: una vez que **m** y **s** son inicializados a 0, son sesgados hacia 0 al principio del entrenamiento, así que estos dos pasos ayudan a comenzar **m** y **s** al principio del entrenamiento.

El decaimiento del momentum  $\beta_1$  es inicializado a 0.9, mientras que el decaimiento del escalado  $\beta_2$  es inicializado con frecuencia a 0.999. Como antes, el término  $\epsilon$  es inicializado a un valor muy pequeño como  $10^{-7}$ . Estos son los valores por defecto de la clase *Adam* (aunque *epsilon* está a *None*, para indicar a Keras que use *keras.backend.epsilon()* por defecto a  $10^{-7}$  y puedes cambiarlo con *keras.backend.set\_epsilon()*).

```
optimizador = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999)
```

Como *Adam* es un algoritmo adaptativo del *learning rate* (como *AdaGrad* y *RMSProp*), no necesita tanto ajuste del hiperparámetro  $\eta$ . Puedes usar el valor por defecto de  $\eta = 0.001$  porque *Adam* lo adaptará. El algoritmo sería:

```
primer_momento = segundo_momento = 0
while True:
    dx = gradiente(J, x)
```

```

primer_momento = beta1 * primer_momento + (1 - beta1) * dx
segundo_momento = beta2 * segundo_momento + (1 - beta2) * dx * dx
x -= learning_rate * primer_momento / (np.sqrt(segundo_momento) + 1e-7)

```

## LEARNING RATES SCHEDULES PERSONALIZADOS

Las optimizaciones que hemos visto hasta ahora usan derivadas parciales de primer orden (Jacobianas) que es lo que utiliza el gradiente. También existen algoritmos basados en las segundas derivadas (la derivada parcial de la Jacobiana, llamadas Hessianas) pero estos algoritmos son muy difíciles de aplicar en el entrenamiento de redes neuronales profundas porque hay  $n^2$  Hessianas mientras que hay  $n$  Jacobianas. Como hay muchos parámetros y  $n$  es el número de parámetros sería muy lento calcularlas.

La opción de usar un learning rate fijo es comenzar con un primer valor alto y luego ir probando nuevos entrenamientos con valores más bajos hasta alcanzar resultados satisfactorios.

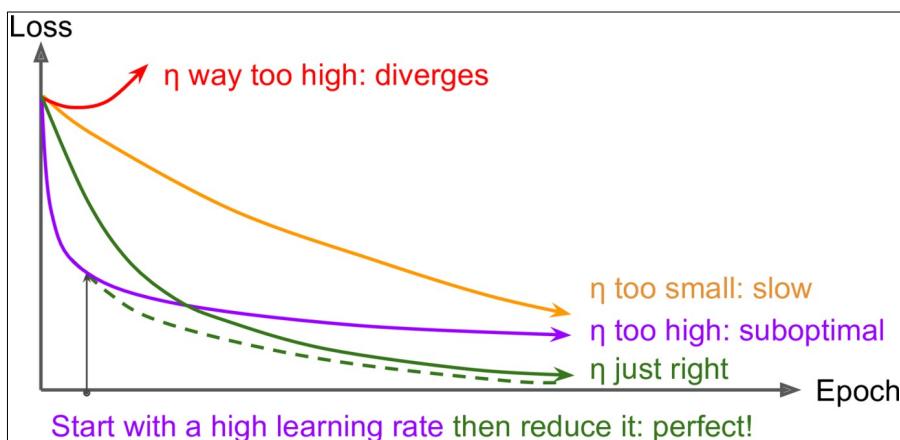


Figura 37: Buscar un valor fijo de learning rate.

También tenemos la opción de usar un optimizador de los vistos anteriormente. Y por último tenemos la opción de fabricarnos nosotros mismos un método de ir variando el valor, personalizar un *learning scheduler*. Algunas posibilidades:

- **Power scheduling** usa el nº de iteración  $t$  para cambiar el valor:  $\eta(t) = \eta_0 / (1 + t/k)^c$ . El valor inicial  $\eta_0$ , la potencia  $c$  (normalmente a 1) y los pasos  $s$  son hiperparámetros. El learning rate es borrado en cada paso y después de  $s$  pasos se baja a  $\eta_0/2$ . Tras  $s$  pasos más se baja a  $\eta_0/3$ . Y así sucesivamente.
- **Exponential scheduling**:  $\eta(t) = \eta_0 \cdot 0.1^{t/s}$ . Reduce el learning rate en un factor de 10 cada  $s$  pasos.
- **Piecewise constant scheduling**: Usa un valor constante cierto número de épocas (por ejemplo  $\eta_0=0.1$  durante 5 épocas) y otro valor más pequeño otro cierto número de épocas, etc...
- **Performance scheduling** mide el error de validación cada  $N$  pasos (como en early stopping) y reduce el valor por un factor  $\lambda$  cuando para.

Andrew Senior et al. Realizó un estudio en 2022 y los autores concluyeron que con optimización de momentum y tanto performance scheduling como exponencial funcionan bien. Ejemplos en keras:

```
optimizador = keras.optimizers.SGD(lr=0.01, decay=1e-4)
```

**Exponential scheduling:**

```
def exponential_decay_fn(epoca):
    return 0.01 * 0.1**((epoca / 20))
```

Si quieres dejar hard-code  $\eta_0$  y  $s$ :

```
def exponencial_decay(lr0, s):
    def exponencial_decay_fn(epoca):
        return lr0 * 0.1**((epoca / s))
    return exponencial_decay_fn
exponencial= exponencial_decay(lr0=0.01, s=20)
```

```
lrs = keras.callbacks.LearningRateScheduler(exponential_decay_fn)
historico = model.fit(X_train_scaled, y_train, [...], callbacks=[lr_scheduler])
```

## **6. PROBLEMAS Y MEJORAS.**

Hay varias técnicas para detectar problemas. Las más básicas las vemos en los siguientes apartados.

## **6.0. PIPELINES DE OPERACIONES.**

Pero antes de comenzar a ver estas técnicas, vamos a introducir una característica de *Scikit-Learn* que consiste en permitirnos encadenar transformaciones a las columnas de entrenamiento, operaciones de selección de características, estimadores y transformaciones a las columnas target dentro de un solo objeto, de manera que al final ahorraremos esfuerzo y errores.

La clase más usada normalmente es `Pipeline` y se utiliza a menudo en combinación con las clases `ColumnTransformer`, `FeatureUnion` y `TransformedTargetRegressor`.

## **6.0.1. PIPELINES DE OPERACIONES.**

Podemos encadenar diferentes operaciones dentro de un objeto **Pipeline** como operaciones de preprocessamiento, selección de características, normalización de datos y clasificación. La ventaja que nos aportan usarlos:

- **Ahorran esfuerzo de desarrollo y facilitan la utilización:** con una llamada a `fit()` y `predict()` desencadenas muchas operaciones que se ejecutan de forma automática.
  - **Unificas la selección de parámetros:** puedes aplicar pruebas para buscar los mejores hiperparámetros de cada estimador del proceso en una misma operación.
  - **Seguridad:** evitan perder información estadística de tus datos de test en el modelo entrenado en un proceso de validación cruzada, asegurando que los mismos datos se usan con los transformadores y los predictores.

Todas las operaciones del *pipeline* salvo la última deben ser transformadores (deben tener el método **`transform()`**) y la última puede ser cualquier cosa: otro transformador, un clasificador, un regresor, etc.

El **Pipeline** se construye usando una lista de parejas (**clave, valor**) donde la clave es un texto con el nombre que tu quieras que identifica la operación y el valor es el objeto que representa la operación. El constructor es **Pipeline(Lista\_operaciones)**:

```
from sklearn.pipeline import Pipeline          # La clase Pipeline
from sklearn.svm import SVC                  # La clase de un clasificador
from sklearn.preprocessing import StandardScaler # Una clase para normalizar datos
operaciones = [('normalizar', StandardScaler()), ('clasificar', SVC())]
pipe = Pipeline(operaciones)
```

Otra forma más abreviada de crear el `Pipeline` es usar el método `sklearn.pipeline.make_pipeline(*steps, memory=None, verbose=False)` que no permite indicar un nombre a los pasos u operaciones que forman parte de la cadena de operaciones porque los nombres se asignan directamente con versiones en minúscula de las operaciones que usas:

El atributo **steps** del objeto que se crea guarda la lista de operaciones que forman la cadena de operaciones. Una de las operaciones podría perfectamente ser otro objeto *Pipeline*. Puede extraerse un *sub-pipeline* usando la notación *slicing* de Python (aunque solo está permitido un paso de 1):

```
p1= pipe[:1] # Pipeline(steps=[('normalizar', StandardScaler())])
p2= pipe[-1:] # Pipeline(steps=[('clasificar', SVC())])
```

Es normal cambiar el valor de un hiperparámetro de alguna operación dentro de un *pipeline*. Este parámetro se denomina anidado porque pertenece a un paso concreto. Se puede acceder a ellos usando la sintaxis **<estimador>\_<parámetro>** (doble subrayado de separación):

```
pipe = Pipeline(steps=[("reduce_dim", PCA()), ("clf", SVC())])
pipe.set_params(clf__C=10)
Pipeline(steps=[('reduce_dim', PCA()), ('clf', SVC(C=10))])
```

## 6.0.2. TRANSFORMAR DATOS DE COLUMNAS CON ColumnTransformer.

Muchos datasets contienen características de diferentes tipos de datos como texto, números, fechas, etc. A veces los algoritmos de aprendizaje solo pueden trabajar con datos numéricos (eso depende de cada algoritmo), o bien funcionan mejor si los valores de las diferentes características tienen escalas similares, etc. En estos y otros casos es necesario aplicar transformaciones a las columnas o características, lo que se conoce como preprocesamiento. Con frecuencia estos cambios son necesarios antes de que los datos lleguen a los algoritmos de aprendizaje y por tanto se pueden aplicar al margen de *scikit-learn* usando operaciones de *numpy* o de *pandas*, pero hacerlo así tiene algunos inconvenientes que solucionan los objetos de *scikit-learn*:

1. Incorporar las estadísticas desde los datos de test a los preprocesadores hacen que los scores (mediciones) de la validación cruzada dejen de estar disponibles (*pérdida de datos*).
2. Si escalas los datos o imputas valores ausentes o outliers durante el entrenamiento cuando hagas uso del modelo debes usar estas mismas transformaciones para realizar predicciones, no solo debes guardar el modelo sino también estas transformaciones y luego aplicarlas con las mismas configuraciones o el modelo no se podrá utilizar (nueva pérdida de información, esta vez de los parámetros).

La clase **ColumnTransformer()** ayuda a realizar diferentes transformaciones a diferentes columnas de los datos, dentro de un Pipeline se asegura que los datos no se pierden y puede parametrizarse. Puede trabajar tanto con **arrays**, matrices dispersas (**sparse matrices**) y **DataFrames** de pandas.

Para cada columna puede aplicarse una transformación diferente, como algún tipo de procesamiento o método de extracción:

```
import pandas as pd
X = pd.DataFrame(
    {'zona': ['Valdepeñas', 'Vinalopó', 'Ribera del Duero', 'Rioja'],
     'bodega': ["San Ricardo", "Las virtudes", "Portia", "Marqués de Riscal"],
     'puntuacion_experto': [5, 3, 4, 5],
     'puntuacion_usuario': [4, 5, 4, 3]})
```

Para estos datos queremos codificar la columna *zona* como una variable categórica usando la clase **OneHotEncoder()** y aplicar una transformación con la clase **CountVectorizer()** a la columna *bodega*. Como nos puede interesar aplicar varias transformaciones a una misma columna como técnicas de extracción de características, vamos a dar a cada transformación un nombre único, por ejemplo '*zona\_categorica*' y '*bodega\_vector*'. Por defecto, el resto de columnas se ignoran que sería el argumento *remainder='drop'*:

```

from sklearn.compose import ColumnTransformer
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.preprocessing import OneHotEncoder
ct = ColumnTransformer([
    ('zona_categorica', OneHotEncoder(dtype='int'), ['zona']),
    ('bodega_vector', CountVectorizer(), 'bodega')],
    remainder='drop', verbose_feature_names_out=False)
ct.fit(X) # Entrenar significa aprender lo necesario para transformar
print(ct.get_feature_names_out()) # Muestra las nuevas columnas que genera
ct.transform(X).toarray()

```

La clase `CountVectorizer()` espera un array 1D como entrada y en este caso un *String* que es el nombre de la columna donde se aplica: `'bodega'`. Sin embargo `OneHotEncoder()` al igual que la mayoría de transformaciones esperan datos 2D, así que en este caso es necesario encerrar el título de la columna en una lista: `['zona']`.

Además de con un escalar o una lista de elementos, las columnas donde aplicar la transformación pueden indicarse con un array de enteros, un operador *slice*, una máscara de booleanos o con una clase selectora de columnas. El método `make_column_selector()` selecciona columnas basándose en sus tipos de datos o en el nombre de las columnas:

```

from sklearn.preprocessing import StandardScaler
from sklearn.compose import make_column_selector
ct = ColumnTransformer([
    ('escalado', StandardScaler(), make_column_selector(dtype_include=np.number)),
    ('categoria', OneHotEncoder(), make_column_selector(pattern='zona', dtype_include=object))])
ct.fit_transform(X)

```

Los *Strings* pueden referenciar columnas si la entrada es un *DataFrame* de pandas, los enteros se interpretan siempre como posiciones de columnas.

Podemos mantener el resto de las columnas y no perderlas si fijamos el parámetro `remainder` al valor `'passthrough': remainder='passthrough'`. Los valores se añaden en ese caso al final de las columnas generadas en la transformación:

```

ct = ColumnTransformer(
    [('categorias', OneHotEncoder(dtype='int'), ['zona']),
     ('vector_bodegas', CountVectorizer(), 'bodega')],
    remainder='passthrough')
print( ct.fit_transform(X) )

```

El parámetro `remainder` puede ser también un estimador para aplicar otra transformación distinta al resto de columnas. Las columnas transformadas se añaden al final de las previamente transformadas:

```

from sklearn.preprocessing import MinMaxScaler
ct = ColumnTransformer(
    [('city_category', OneHotEncoder(), ['city']),
     ('title_bow', CountVectorizer(), 'title')],
    remainder=MinMaxScaler())
ct.fit_transform(X)[:, -2:]

```

La función `make_column_transformer` puede utilizarse para crear más rápidamente un objeto `ColumnTransformer` con la diferencia de que los nombres de las transformaciones se asignan automáticamente. Ejemplo:

```

from sklearn.compose import make_column_transformer
ct = make_column_transformer( OneHotEncoder(), ['zona']),

```

```

        (CountVectorizer(), 'bodega'),
        remainder=MinMaxScaler())
print(ct)

```

Si el **ColumnTransformer** se entrena con un *dataframe* que solamente tiene nombres de columnas, se usan los nombres de las columnas para seleccionar otras:

```

ct = ColumnTransformer([('escalar', StandardScaler(), ["puntuacion_experto"])]).fit(X)
X_nuevo = pd.DataFrame({"puntuacion_experto": [5, 6, 1], "columna_ignorada": [1.2, 0.3, -0.1]})
print(ct.transform(X_nuevo))

```

### 6.0.3. TRANSFORMAR LA COLUMNA TARGET EN REGRESIONES.

La clase **TransformedTargetRegressor()** transforma las columnas *y* antes de realizar el entrenamiento de un modelo de regresión. Las predicciones se mapean al espacio de valores original realizando la transformación inversa. Recibe de argumentos el regresor usado en la predicción y la transformación que se aplica a la columna *target*:

```

import numpy as np
from sklearn.datasets import fetch_california_housing
from sklearn.compose import TransformedTargetRegressor
from sklearn.preprocessing import QuantileTransformer
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
X, y = fetch_california_housing(return_X_y=True)
X, y = X[:2000, :], y[:2000]          # seleccionar un subconjunto de datos
qt = QuantileTransformer(output_distribution='normal')
rl = LinearRegression()
ttr = TransformedTargetRegressor(regressor=rl, transformer=qt)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
ttr.fit(X_train, y_train)
print('R2: {:.2f}'.format(ttr.score(X_test, y_test)))
raw_target = LinearRegression().fit(X_train, y_train)
print('R2: {:.2f}'.format(raw_target.score(X_test, y_test)))

```

Para transformaciones sencillas en vez de un objeto **Transformer** se pueden pasar un par de funciones que definan la transformación y su inversa:

```

def funcion1(x):
    return np.log(x)
def funcion1_inversa(x):
    return np.exp(x)

```

Y el objeto se puede crear como:

```

ttr = TransformedTargetRegressor(regressor=rl, func=funcion1, inverse_func=funcion1_inversa)
ttr.fit(X_train, y_train)
print('R2: {:.2f}'.format(ttr.score(X_test, y_test)))

```

Por defecto se comprueba que las funciones realmente son la inversa de la otra, si quieres que no se realice esta comprobación puedes pasar el argumento **check\_inverse** a **False**.

### 6.0.4. COMPONER DIFERENTES ESPACIOS.

La clase **FeatureUnion** agrupa diferentes transformaciones en un nuevo transformador que combina sus salidas. Recibe una lista de objetos transformadores y durante el entrenamiento cada uno de ellos es entrenado con los datos que se le pasan de manera independiente. Las operaciones se realizan en paralelo sobre los datos y los resultados se unen unos al lado de otros en una matriz mayor. Si quieres

aplicar diferentes transformaciones una después de otra a la misma característica debes usar **ColumnTransformer**.

El objeto **FeatureUnion** se define con una lista de parejas (clave, valor) donde la clave es el nombre que se le da a la transformación y el valor es un objeto estimador:

```
from sklearn.pipeline import FeatureUnion
from sklearn.decomposition import PCA
from sklearn.decomposition import KernelPCA
estimadores = [('linear_pca', PCA()), ('kernel_pca', KernelPCA())]
fu = FeatureUnion(estimadores)
```

Igual que los *pipelines* tienen una función que abrevia la definición que es `make_union()` y no se le pasan los nombres de las transformaciones.

Como los *pipelines*, los pasos individuales pueden cambiarse usando `set_params` e ignorados usando '`drop`':

```
fu.set_params(kernel_pca='drop')
```

## 6.1. CURVAS DE ENTRENAMIENTO/APRENDIZAJE.

Una vez que has entrenado un modelo podría no ser el adecuado para los datos por diferentes razones y en ese caso o cambiamos los datos o cambiamos de modelo. Pero aunque funcione bien con los datos de entrenamiento, es necesario validarla porque podría funcionar mal con otros datos distintos (no generaliza bien) también por diferentes razones. Los problemas más habituales se conocen con los nombres *underfitting* (subajuste) y *overfitting* (sobreajuste).

Para detectar estas situaciones además de observar las métricas, puedes hacerlo visualmente observando las curvas de aprendizaje (o curvas de entrenamiento) que ya hemos visto y que visualizan como va cambiando la función de coste a medida que el algoritmo de entrenamiento va definiendo el modelo. Estos gráficos suelen representar en cada etapa del entrenamiento tanto el error que comete el modelo con los datos de entrenamiento como con otros datos que hemos apartado y reservado para testear el modelo. También ayudan a saber como impacta la cantidad de datos.

**EJEMPLO 7:** Define una función para generar un gráfico de la curva de entrenamiento o curva de aprendizaje. Le pasamos de parámetros el modelo, el dataset X y sus etiquetas y:

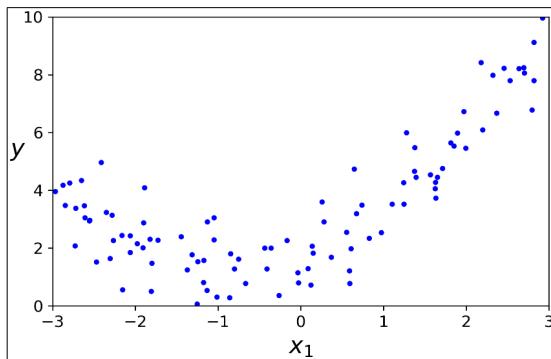
```
import numpy as np
import matplotlib as plt
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn import LinearRegression

def plot_curvas_entrenamiento(modelo, X, y):
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
    train_errores, test_errores = [], []
    for n in range(1, len(X_train)):
        modelo.fit(X_train[:n], y_train[:n])
        y_train_predicciones = modelo.predict(X_train[:n])
        y_prediccion = modelo.predict(X_test)
        train_errores.append(mean_squared_error(y_train[:n], y_train_predicciones))
        test_errores.append(mean_squared_error(y_test, y_prediccion))
    plt.plot(np.sqrt(train_errores), "r+", linewidth=2, label="train")
    plt.plot(np.sqrt(test_errores), "b-", linewidth=3, label="test")
```

Si generamos datos que sigan una parábola e intentamos describirlos con un modelo lineal el resultado no será bueno. Vamos a intentar detectar esta situación con curvas de aprendizaje:

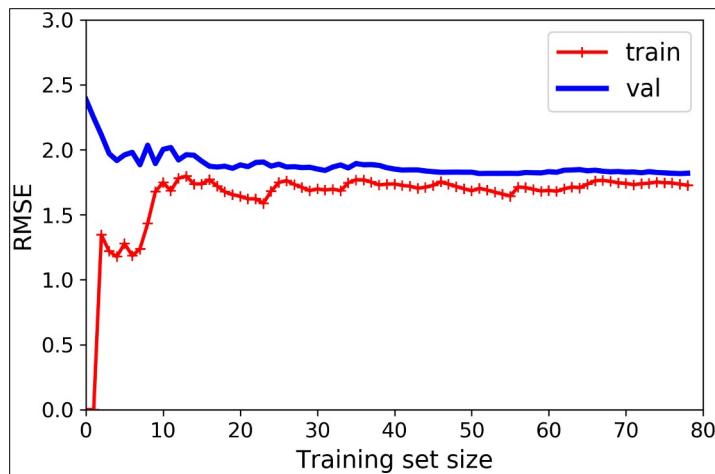
```
m = 100
X = 6 * np.random.rand(m, 1) - 3
reg_lin = LinearRegression()
```

```
plot_curvas_entrenamiento(reg_lin, X, y)
plt.show()
```



*Figura 38: Datos difíciles de describir con una recta.*

La figura 38 muestra el gráfico de curvas de aprendizaje anterior. Esto nos da una idea de como influye la cantidad de datos usados en el entrenamiento en el error que comete el modelo tanto con datos de entrenamiento como de test (en los que debe saber generalizar, aplicar lo que ha aprendido con los de entrenamiento). Si observamos la curva en los datos de entrenamiento, vemos que con uno o dos datos el error parte de 0. Con más de dos ejemplos desalineados en el dataset es imposible que el error sea 0 porque no hay una recta que pueda pasar por 3 diferentes puntos no alineados. El error en los datos de entrenamiento por tanto alcanza una meseta que presenta ciertas subidas y bajadas pero que básicamente se ha mantenido.



*Figura 39: curvas de entrenamiento o aprendizaje.*

Ahora vamos con el error en los datos de test (o validación, en azul). El modelo entrenado con pocos datos es incapaz de generalizar por lo que el error de validación comienza siendo bastante alto. A medida que el modelo va entrenando con más ejemplos, el error va bajando suavemente. Sin embargo como una línea no es un buen modelo para los datos, llega un momento que aparece de nuevo una meseta. Estas curvas de entrenamiento presentan el patrón de un modelo demasiado simple o no adecuado para los datos, es una situación conocida como **underfitting**.

Añadir más ejemplos no ayudará. Las posibles soluciones serían cambiar de modelo, añadir más características que aporten mayor información, o quitar restricciones si las tuviese. Vamos a optar por cambiar de modelo, vamos a usar un modelo más complejo que pueda ajustar más, usaremos un modelo polinomial de grado 10 (en vez de una recta usamos un polinomio, lo veremos más adelante).

```
from sklearn.pipeline import Pipeline
reg_poli = Pipeline([('poly_features', PolynomialFeatures(degree=10, include_bias=False)),
                     ('lin_reg', LinearRegression()),
                     ])
plot_curvas_entrenamiento(reg_poli, X, y)
plt.show()
```

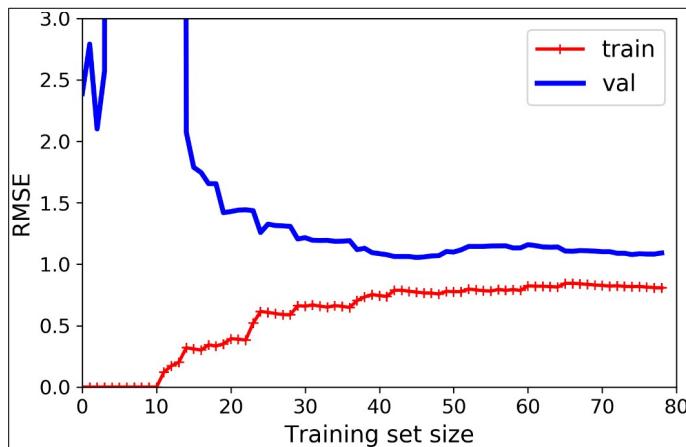


Figura 40: curvas de entrenamiento del modelo polinomial de grado 10.

Las curvas del modelo polinomial se parecen a las del modelo lineal salvo por dos diferencias importantes:

- El error en los datos de entrenamiento es mucho más bajo.
- Hay un hueco entre las curvas. El modelo funciona mejor con los datos de entrenamiento que con los datos de validación, sin embargo, el error de entrenamiento sigue creciendo a partir del entrenamiento 10, esto significa que si continuamos entrenando las dos curvas acabarían por acercarse más, la de test va bajando y la de entrenamiento va subiendo.

Este es el aspecto de un modelo que tiene *overfitting* (sobreajuste). En esta situación, añadir más datos de entrenamiento ayudaría a bajar este problema.

En general, durante el entrenamiento del modelo podemos generar las curvas de aprendizaje o curvas de entrenamiento y muchas veces, su propia forma nos avisará de si hay problemas.

#### EJEMPLO 8: Ejemplos de algunas curvas:

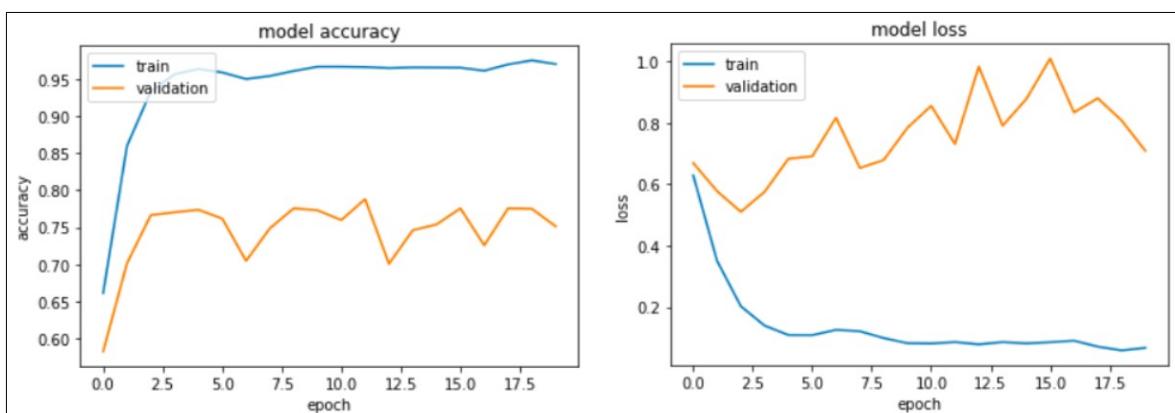


Figura 41: curvas de entrenamiento que indican *overfitting*.

En estos gráficos (es el entrenamiento del mismo modelo pero a la izquierda representamos la eficiencia y a la derecha el error de la función de coste). En este caso el error de entrenamiento es bajo y evoluciona bien pero el de test es alto, por tanto el modelo no generaliza bien, está detectando como patrones de los datos de entrenamiento que no existen en la vida real, es decir, tiene *overfitting*. Esto lo provocan: pocos datos, modelo demasiado complejo, características demasiado influyentes o entrenamiento excesivo. Las soluciones serían según el origen del problema: aumentar los datos de entrenamiento, eliminar características problemáticas, simplificar el modelo, sus capas, incorporar regularización o estrategias de parada temprana.

#### EJEMPLO 9 Curva de aprendizaje con buen entrenamiento pero el test estancado en una meseta:

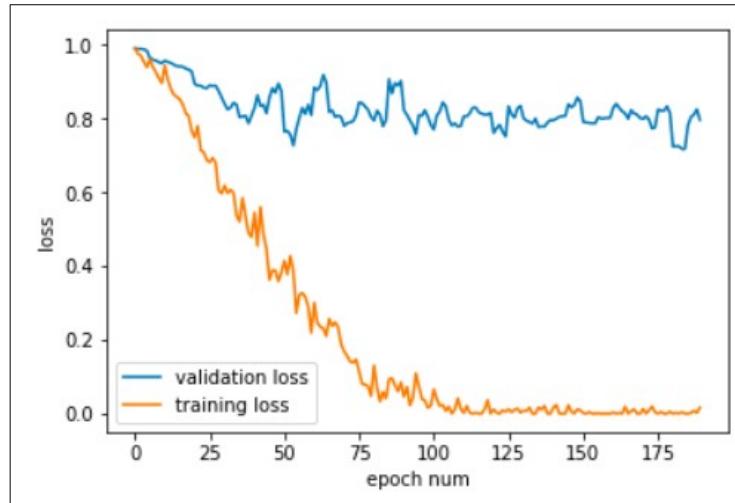


Figura 42: Bien el train y estancado el test.

El error en el train va bajando a medida que progresa el entrenamiento, pero en el test sube y baja quedándose en una especie de meseta (*tableau*). Si el error de train baja pero el de test cambia a un ritmo diferente o no baja, el modelo tiene *overfitting*, no generaliza bien, sigue aprendiendo patrones que no existen.

**EJEMPLO 10.** Curva de aprendizaje con buen entrenamiento y buen test pero mucha diferencia:

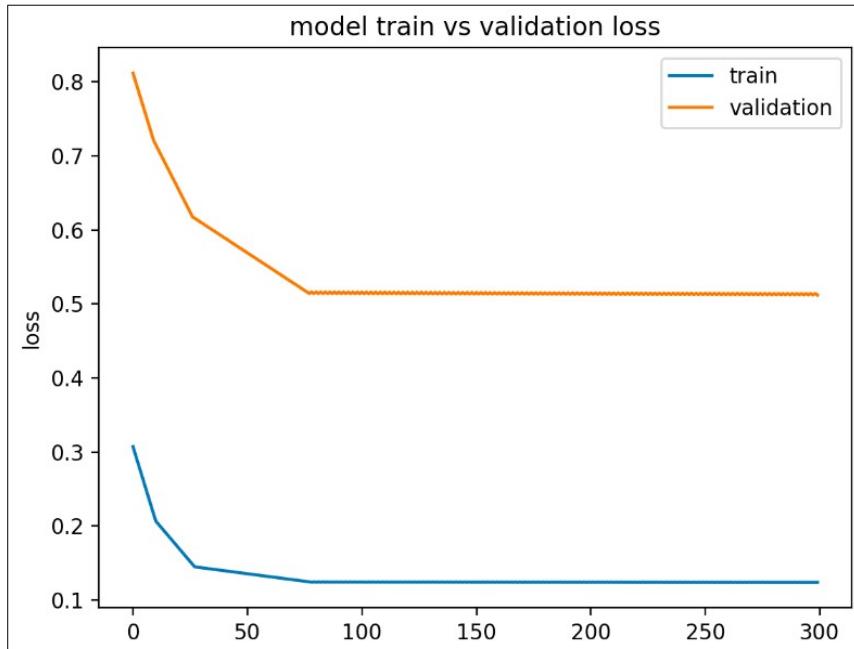


Figura 43: Bien el train y estancado el test.

Un gran hueco entre las curvas de train y test sugieren *overfitting* provocado por datos defectuosos, o porque los datos tienen mucho bias, están mal preprocessados, están imbalanceados o el modelo utiliza una regularización incorrecta.

**EJEMPLO 11.** Curva de aprendizaje errática en train y test. Puede deberse a mal preprocessamiento de datos, mal valor de hiperparámetros como *learning rate*, los datos pueden tener mucho bias, mala elección del algoritmo de aprendizaje, etc.

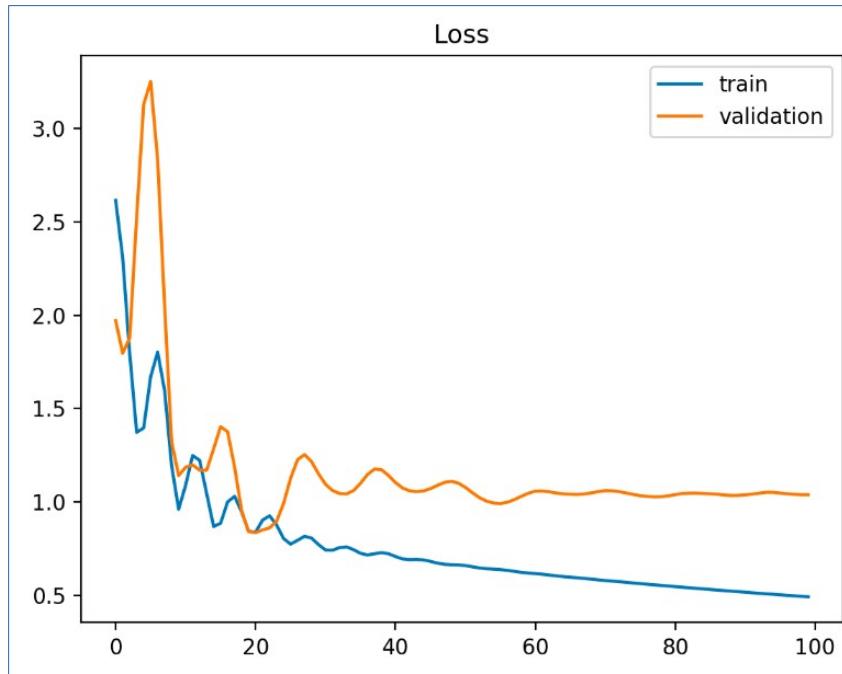


Figura 44: Evolución errática o se incrementa el error.

EJEMPLO 12: Dibujar curvas de aprendizaje o entrenamiento usando scikit.

```
from sklearn.datasets import load_digits
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
# cargar los datos
X, y = load_digits(return_X_y=True)
nb = GaussianNB() # Un modelo que vemos en la siguiente unidad
svc = SVC(kernel="rbf", gamma=0.001) # Otro modelo de la siguiente unidad
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import LearningCurveDisplay, ShuffleSplit
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 6), sharey=True)
parametros = {
    "X": X,
    "y": y,
    "train_sizes": np.linspace(0.1, 1.0, 5),
    "cv": ShuffleSplit(n_splits=50, test_size=0.2, random_state=0),
    "score_type": "both",
    "n_jobs": 4,
    "line_kw": {"marker": "o"},
    "std_display_style": "fill_between",
    "score_name": "Accuracy",
}
for ax_idx, modelo in enumerate([nb, svc]):
    LearningCurveDisplay.from_estimator(modelo, **parametros, ax=ax[ax_idx])
    handles, label = ax[ax_idx].get_legend_handles_labels()
    ax[ax_idx].legend(handles[:2], ["Training Score", "Test Score"])
    ax[ax_idx].set_title(f"Curva de Aprendizaje para {modelo.__class__.__name__}")
```

## BALANCEAR ERRORES DE BIAS Y VARIANZA

Un resultado teórico de estadística y machine learning es que el error de generalización de modelo es la suma de 3 errores diferentes:

- **Error de Bias:** es el error que se comete al asumir como ciertas cosas que no lo son y se introduce un error. Por ejemplo asumimos que los datos son lineales cuando en realidad son cuadráticos. Un modelo con alto *bias* tiene tendencia a tener *underfitting*. Si lo vemos desde la perspectiva de la estadística es la **tendencia a sobrestimar o subestimar un parámetro**. Si el

modelo tiene un **Bias alto significa que le presta poca atención a los datos y sobre simplifica el modelo**. Esto nos lleva a tener un **error alto tanto en train como en test**.

- **Error de Varianza:** lo sensible que es el modelo a pequeñas variaciones de los datos de entrenamiento. Un modelo con muchos grados de libertad (un modelo polinomial de muchos grados por ejemplo) tendrá alta varianza y tendencia a padecer sobreajuste (*overfitting*). Si tenemos exceso de sensibilidad a cambios, el modelo puede creer ver patrones que realmente no existen (porque captura errores en los datos y los asume como patrones de la realidad). En el contexto de estadística es una medida de dispersión de los datos, es la distancia de cada variable a la media de todas las variables. Si tenemos un valor alto de varianza significa que el **modelo le presta demasiada atención a los datos de entrenamiento y no va a generalizar bien con datos que no ha visto**. Esto lo podemos ver cuando **en el entrenamiento el modelo funciona muy bien pero tiene mucho error en el test**.
- **Error Irreducible:** se debe al ruido de los propios datos. La única manera de bajar este ruido es mejorar su calidad realizando limpieza (mejorar la fuente de datos, como sensores defectuosos y detectar y eliminar *outliers*) o escoger mejores datos.

$$\text{Error}(x) = \text{Bias}^2 + \text{Variance} + \text{ErrorIrreducible}$$

Si tenemos un modelo con alto *bias* o alta varianza, nuestro modelo va a tener problemas. La única opción que tenemos es encontrar el punto donde ambos puedan ser lo más pequeños posible. Una imagen clásica para explicar estos conceptos consisten en 4 dianas cuyo centro representa donde están las respuestas donde podemos ver el efecto de *Bias* y varianza.

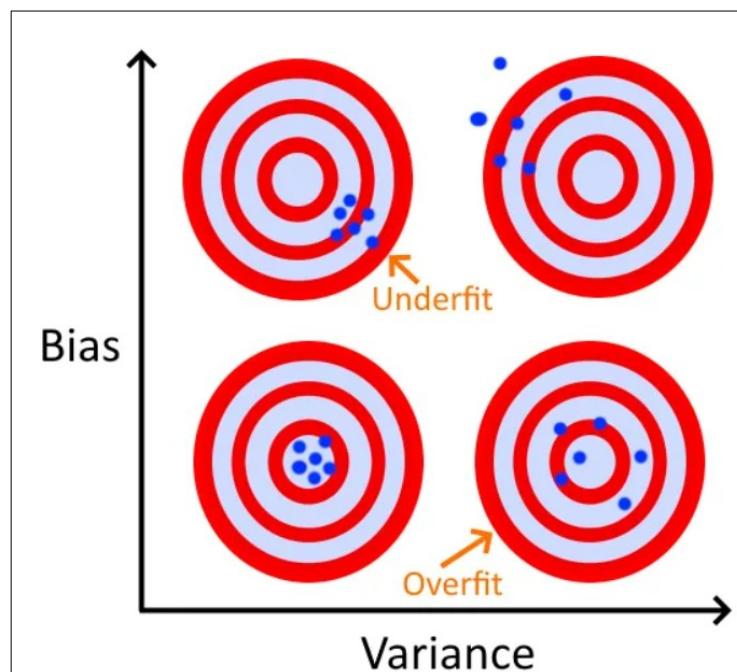
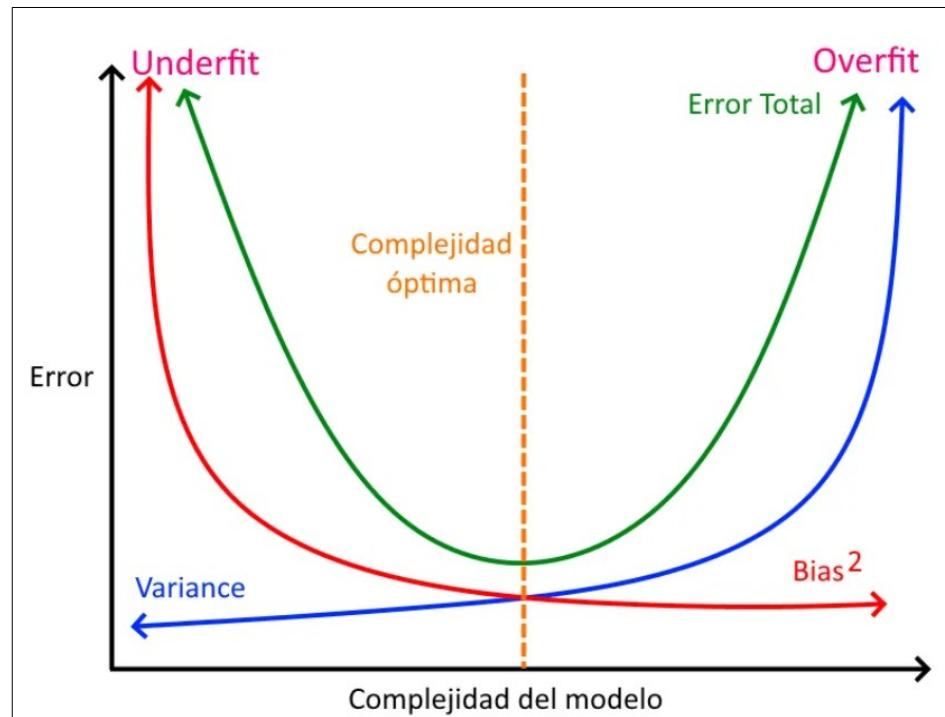


Figura 45: Dianas que representan el efecto de tener alto bias o alta varianza en un modelo.

Si el *bias* es pequeño nos encontramos cerca del centro, pero si se incrementa nos alejamos de él. Con la varianza podemos observar que si el valor es pequeño los puntos (las predicciones) se encuentran cercanos entre sí, pero al aumentar la varianza, los puntos se encuentran dispersos. Solamente si bajamos el error en ambos podemos tener los mejores resultados (diana inferior izquierda).



*Figura 46: Relación entre bias, varianza, complejidad del modelo, underfitting y overfitting.*

Incrementar la complejidad de un modelo normalmente aumenta su varianza y reduce el bias. De la misma manera, bajar la complejidad de un modelo incrementa su bias y baja su varianza.

### OVERFITTING (SOBREAJUSTE)

Ocurre cuando un modelo estadístico o de machine learning aprende de errores aleatorios o del ruido en vez de aprender de las verdaderas relaciones que mantienen los datos entre sí y en las que estamos interesados.

Suele aparecer cuando usamos un modelo demasiado complejo, por ejemplo que tenga muchos parámetros en relación con la cantidad de datos de los que aprende. Se puede detectar este problema porque cometerá muy pocos errores con los datos de entrenamiento pero se comportará muy mal (comete muchos errores) con los datos de test o con los datos de trabajo.

Puede deberse principalmente a 3 factores:

- **Modelos demasiado complejos.** Demasiados parámetros para la cantidad de datos con los que ha sido entrenado. O los datos son ruidosos. Soluciones:
  - Aumentar la cantidad de datos de entrenamiento.
  - Reducir la cantidad de predictoras.
  - Cambiar a un modelo con menos parámetros puede reducir este problema.
  - Poner restricciones al modelo (regularizar)
- **Multicolinealidad:** si los predictores usados están correlacionados linealmente entre sí de una manera importante, los coeficientes del modelo linea pueden cambiar de manera errática en cada entrenamiento. El problema es que si un predictor puede deducirse a partir de otro (está correlacionado), en realidad aporta información redundante que no es útil. Soluciones:
  - **Regularización l2:** Introduce un bias en la solución haciendo que  $(X^T X)^{-1}$  no sea singular.
  - **Selección de características:** bajar la complejidad del modelo detectando y eliminando las características que estén correlacionadas. O usar regularización l1 para detectar cuales son y eliminarlas.
  - Reducir las características con **PCA** (análisis de componentes principales) o **PLS-R** (regresión parcial de mínimos cuadrados): permiten eliminar predictoras correlacionadas.
- **Alta dimensionalidad:** cuando estamos usando demasiadas predictoras (llamamos  $p$  a esta cantidad), cada una da lugar a un parámetro del modelo. Aunque no haya colinealidad,

quizás muchas de ellas no aportan excesiva información para explicar el target en el que estamos interesados. Si  $p$  en relación con la cantidad de ejemplos que tiene nuestros datos de entrenamiento (lo llamamos  $n$ ) es alto, nuestro modelo tiende a sobreajustar. De hecho el tamaño de P/N determinará muchas veces el algoritmo de aprendizaje que se elige.

## 6.2. MODELOS REGULARIZADOS.

Una buena forma de reducir el *overfitting* es regularizar el modelo, es decir, aplicarle alguna restricción para quitarle libertad al algoritmo cuando calcula los parámetros. Si le impones alguna condición, ya no podrá ajustarse tan perfectamente a los datos de entrenamiento. Si estos son pocos o ruidosos, le dificultas que los asimile y le obligas a generalizar, a fabricar un modelo más general.

En un modelo polinomial, si reduces el grado del polinomio simplificas el modelo. En un modelo lineal, se suelen usar las restricciones **Ridge**, **LASSO** y **Elastic Net** que dan lugar a variaciones del modelo lineal. Estas son aportadas por **Statsmodel** y por **scikit-learn**. Cuando se incrementa la complejidad del modelo, el tamaño de los coeficientes sube exponencialmente, así que *Ridge* y *LASSO* penalizan el aumento de valor de los coeficientes.

### REGRESIÓN RIDGE (REGULARIZACIÓN L2 O DE TIKHONOV)

Añade un término para intentar que la suma de los cuadrados de los parámetros sea cercana a 0, pero no 0. Fuerza a los parámetros del modelo a quedar cerca de cero. Así que la función de coste  $L(\theta)$  se mezcla con una función de restricción o penalización  $\Omega(\theta)$ . Aparece un nuevo hiperparámetro  $\lambda$  que es la fuerza con la que se penaliza, un valor entre 0 y 1. Si es 0 es como no aplicar la penalización.

$$\text{Penalización}(\theta) = L(\theta) + \lambda \Omega(\theta)$$

La regresión **Ridge** impone una penalización  $\ell_2$  a los coeficientes, es decir, impone que la norma Euclídea de los coeficientes sea mínima cuando se minimiza **SSE**. La función objetivo es:

$$\text{Ridge}(X, \theta) = \frac{1}{n} \sum_{i=1}^n (y^{(i)} - x^{(i)T} \theta)^2 + \lambda \|\theta\|_2^2 = \|y - X\theta\|_2^2 + \lambda \|\theta\|_2^2$$

Podemos obtener la nueva ecuación normal minimizando la función (llamamos  $\beta$  a  $\theta$ ):

$$\begin{aligned} \nabla_{\beta} \text{Ridge}(\beta) &= 0 \\ \nabla_{\beta} ((y - X\beta)^T (y - X\beta) + \lambda \beta^T \beta) &= 0 \\ \nabla_{\beta} ((y^T y - 2\beta^T X^T y + \beta^T X^T X \beta + \lambda \beta^T \beta)) &= 0 \\ -2X^T y + 2X^T X \beta + 2\lambda \beta &= 0 \\ -X^T y + (X^T X + \lambda I)\beta &= 0 \\ (X^T X + \lambda I)\beta &= X^T y \end{aligned}$$

$$\theta = (X^T X + \lambda A)^{-1} X^T y \quad (\text{Ecuación normal regularizada})$$

Donde la matriz  $A$  (la  $I$  del desarrollo) de esta ecuación es como la matriz identidad de dimensiones  $(p+1) \times (p+1)$  que tiene toda la diagonal a 1 menos el elemento superior izquierdo que lo tiene a 0 para no anular el bias. La solución añade una constante positiva a la diagonal de  $(X^T X)$  antes de invertirla, por lo que soluciona a veces el problema de que sea una matriz singular (no invertible). El gradiente de la función sería:

$$\frac{\partial L(\theta, X, y)}{\partial \theta} = 2 \left( \sum_{i=1}^n x^{(i)} (x^{(i)} \theta - y^{(i)}) + \lambda \theta \right)$$

La figura 47 muestra la solución **OLS** a la izquierda. Las penalizaciones  $\ell_1$  y  $\ell_2$  en el centro y la derecha como las penalizaciones limitan los coeficientes hacia el 0. Los puntos negros son el mínimo encontrado y los puntos blancos son la solución real de los datos generados.

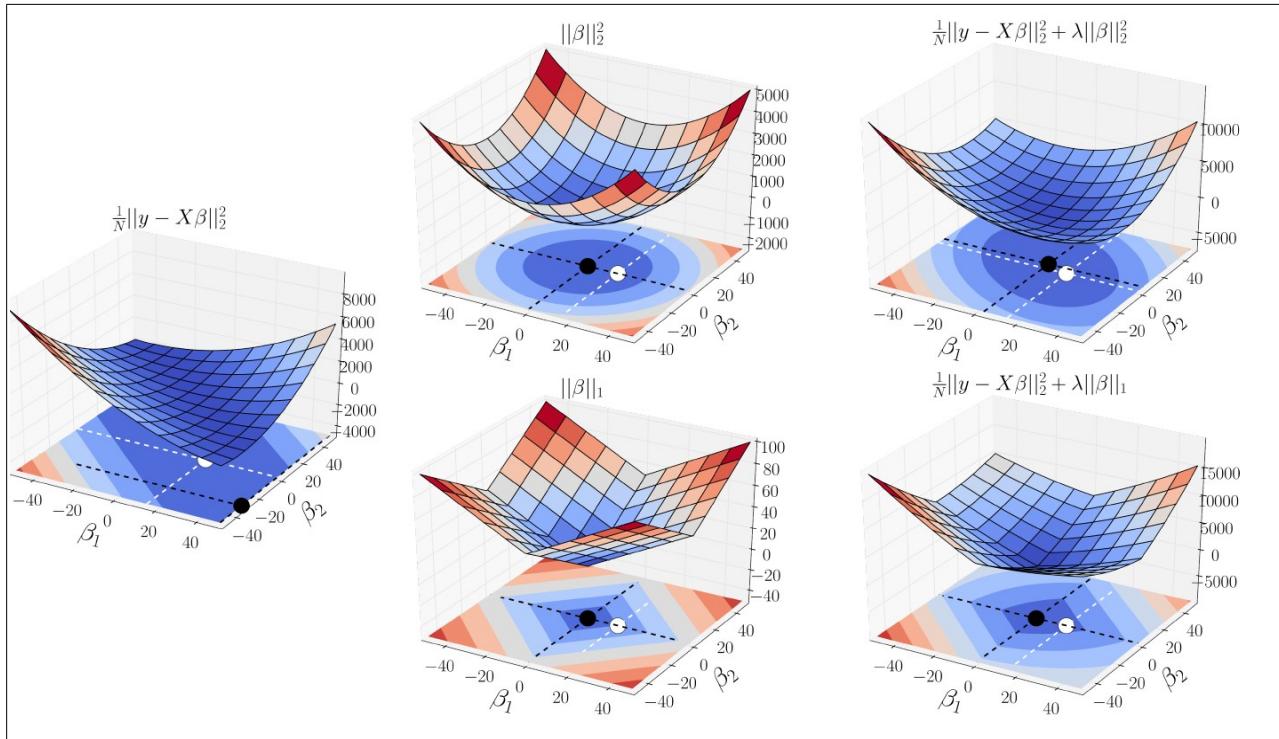


Figura 47: restricciones  $\ell_1$  y  $\ell_2$ .

Ten en cuenta que:

- El término del bias  $\theta_0$  no se regulariza (la suma de los parámetros comienza en  $i=1$ ).
- Es importante escalar los datos de entrenamiento cuando utilices cualquier regularización.

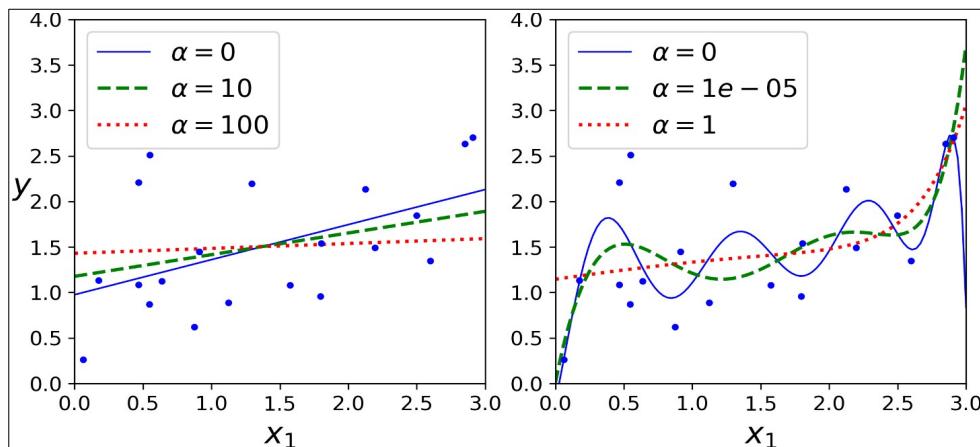


Figura 48: Regresión Ridge.

**EJEMPLO 13:** A la derecha los datos se expanden usando `PolynomialFeatures(degree=10)`, y escalados usando un `StandardScaler` y finalmente se crea el modelo de regresión Ridge. Al incrementar el parámetro de regularización (`lambda` en las fórmulas y alfa en el gráfico) la pendiente de la recta baja en el modelo lineal y el modelo polinómico se simplifica. **El modelo reduce su varianza e incrementa su bias.** Para calcular el modelo podemos usar la ecuación o usar el SGD, a continuación tienes ejemplos de como hacerlo de las dos formas:

```
# Variante usando factorización de Andre-Louis Cholesky
from sklearn.linear_model import Ridge
rl_ridge = Ridge(alpha=1, solver="cholesky")
rl_ridge.fit(X, y)
ridge_reg.predict([[1.5]]) # devuelve array([1.55071465])
# Usando DGS
rl_dgs = SGDRegressor(penalty="l2")
rl_dgs.fit(X, y.ravel())
rl_dgs.predict([[1.5]]) # devuelve array([1.47012588])
```

## REGRESIÓN LASSO (REGULARIZACIÓN L1)

LASSO (*Least Absolute Shrinkage and Selection Operator Regression*) es otra versión regularizada de la regresión lineal: añade a la función de coste un término usando norma  $\ell_1$  al vector de parámetros del modelo:

$$J(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + \alpha \sum_{i=1}^n |\theta_i|$$

Penaliza a los coeficientes de manera que obliga a algunos a ser exactamente 0. Esto es interesante porque detecta las variables que no aportan nada para explicar el *target* y que por tanto se pueden eliminar. Es decir, te ofrece un **método de selección de características**. La función objetivo se convierte en:

$$\text{Lasso}(\boldsymbol{\beta}) = \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|_2^2 + \lambda \|\boldsymbol{\beta}\|_1.$$

Obtener la ecuación analítica es complicado porque el valor absoluto (la norma  $\ell_1$ ) no es derivable, así que es más sencillo de implementar usando algoritmos iterativos como el DGS.

Aquí tienes un código que usa y ten en cuenta que también podrías utilizar un `SGDRegressor(penalty="l1")`.

```
from sklearn.linear_model import Lasso
rl_lasso = Lasso(alpha=0.1)
rl_lasso.fit(X, y)
rl_lasso_reg.predict([[1.5]])      # Resultado: array([1.53788174])
```

**EJEMPLO 14:** el mismo ejemplo que se usó para Ridge pero con LASSO.

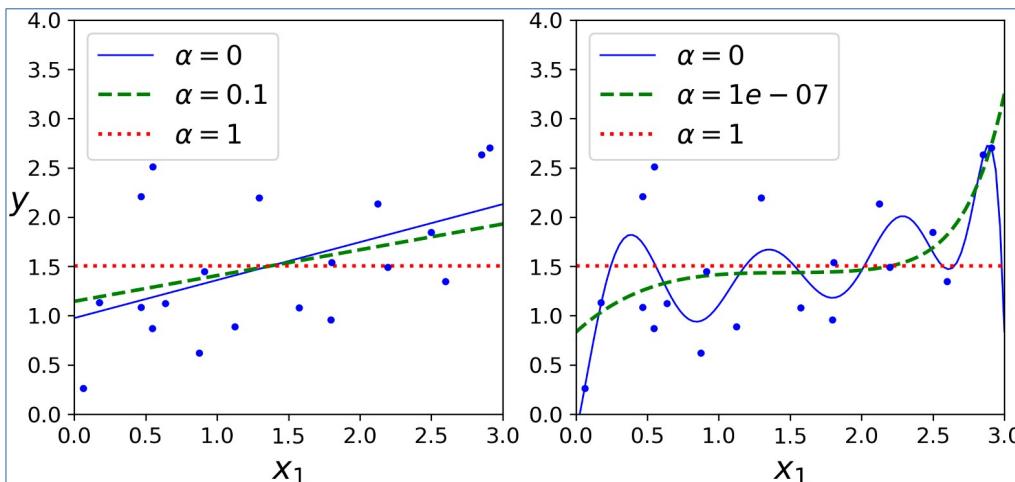


Figura 49: Regresión Lasso.

## REGRESIÓN ELASTIC NET

Es una mezcla de las dos anteriores. El término de regularización es una mezcla de *Ridge* y *LASSO* controlados por un factor  $r$ . Cuando  $r = 0$ , *Elastic Net* equivale a *Ridge* y cuando es 1 equivale a *LASSO*. La función de coste en forma iterativa y vectorial:

$$J(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + r\alpha \sum_{i=1}^n |\theta_i| + \frac{1-r}{2} \alpha \sum_{i=1}^n \theta_i^2$$

$$\text{Enet}(\boldsymbol{\beta}) = \|\mathbf{y} - \mathbf{X}^T \boldsymbol{\beta}\|_2^2 + \alpha (\rho \|\boldsymbol{\beta}\|_1 + (1-\rho) \|\boldsymbol{\beta}\|_2^2)$$

¿Es bueno usar regularización? ¿Cuál de ellas es aconsejable? En la mayoría de los casos siempre es bueno tener un poco de regularización así que podrías descartar usar regresión lineal normal. Ridge suele funcionar bien. Salvo que sospeches que tienes características poco útiles en cuyo caso la opción es usar Lasso o Elastic Net para descubrirlas. En general, Elastic Net es mejor que Lasso porque este se mueve de manera errática si tienes varias características fuertemente correlacionadas.

#### EJEMPLO 15: usar *Elastic Net*:

```
from sklearn.linear_model import ElasticNet
rl_en = ElasticNet(alpha=0.1, l1_ratio=0.5)
rl_en.fit(X, y)
rl_en.predict([[1.5]])      # Resultado: array([1.54333232])
```

#### PARADA TEMPRANA (EARLY STOPPING)

Una forma completamente diferente de regularizar los algoritmos iterativos como los de descenso por gradiente es parar el entrenamiento tan pronto como la validación alcance un mínimo razonable. A esta acción de detener el entrenamiento es a lo que se denomina *early stopping*. La figura 50 muestra la curva de entrenamiento de un modelo complejo (un polinomio de alto grado) que ha comenzado a entrenarse con el descenso por gradiente Batch. Cuando las épocas de entrenamiento van pasando, el algoritmo va aprendiendo de la función de coste *RMSE* que como es de esperar va descendiendo a medida que avanza el aprendizaje.

Sin embargo, alrededor de la época 240, el error que comete vuelve a subir y mantiene esta tendencia a medida que continúa el aprendizaje. En ese punto, el modelo ha comenzado a sobreajustar los datos de entrenamiento, comienza a aumentar su *overfitting*.

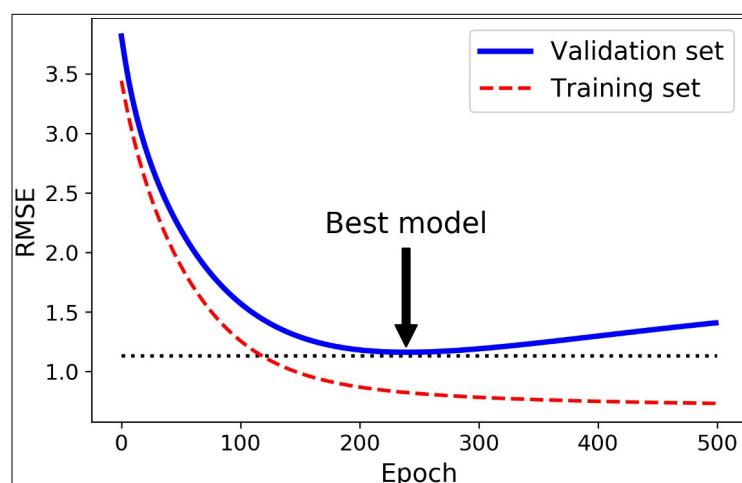


Figura 50: Regularización Early stopping.

Cuando usas las variantes estocástica y *mini-batch* del algoritmo las curvas no son suaves y en estos casos es difícil detectar si se produce esta situación. La solución es parar solamente cuando se alcance el mínimo error con los datos de validación. En ese momento los parámetros del modelo vuelven a fijarse con los valores que tenían cuando se alcanzó el mínimo error.

#### EJEMPLO 16: Implementar de forma básica la regularización early stopping:

```
from sklearn.base import clone
# preparar los datos
poly_scaler = Pipeline([
    ("poly_features", PolynomialFeatures(degree=90, include_bias=False)),
    ("std_scaler", StandardScaler())
])
X_train_poly_scaled = poly_scaler.fit_transform(X_train)
X_val_poly_scaled = poly_scaler.transform(X_val)
sgd_reg = SGDRegressor(max_iter=1, tol=-np.inf, warm_start=True, penalty=None,
                      learning_rate="constant", eta0=0.0005)
minimo_val_error = float("inf")
```

```

mejor_epoca = None
mejor_modelo = None
for epoca in range(1000):
    sgd_reg.fit(X_train_poly_scaled, y_train)      # continua mientras esté por debajo
    y_val_predict = sgd_reg.predict(X_val_poly_scaled)
    val_error = mean_squared_error(y_val, y_val_predict)
    if val_error < minimo_val_error:
        minimo_val_error = val_error
        mejor_epoca = epoca
        mejor_modelo = clone(sgd_reg)

```

Observa que con `warm_start=True`, cuando el método `fit()` se llame, solamente continúa entrenando donde se quedó, no comienza desde el principio de nuevo.

## 6.3. ANÁLISIS CON ESTADÍSTICA Y GRÁFICOS.

Una vez ajustado el modelo, es necesario verificar su utilidad ya que, aun siendo la línea que mejor se ajusta a las observaciones de entre todas las posibles, puede tener un gran error. Las métricas más utilizadas para medir la calidad del ajuste son: el error estándar de los residuos y el coeficiente de determinación  $R^2$  que ya hemos comentado pero que volvemos a recordarlos:

- **Error estándar de los residuos (RSE):** Mide la desviación promedio de las predicciones del modelo respecto de la recta de regresión. Tiene las mismas unidades que target. Una forma de saber si el RSE es elevado consiste en dividirlo entre el valor medio de target obteniendo así un % de la desviación.

$$RSE = \sqrt{\frac{1}{n-p-1} RSS}$$

En modelos lineales simples, dado que hay una sola predictora  $(n-p-1) = (n-2)$ .

- **Coeficiente de determinación  $R^2$ :** describe la proporción de varianza de target explicada por el modelo con respecto a su varianza total. Su valor está acotado entre 0 y 1. Al ser adimensional, presenta la ventaja frente al RSE de ser más fácil de interpretar.

$$\begin{aligned}
R^2 &= \frac{\text{Suma de cuadrados totales} - \text{Suma de cuadrados residuales}}{\text{Suma de cuadrados totales}} = \\
&= 1 - \frac{\text{Suma de cuadrados residuales}}{\text{Suma de cuadrados totales}} = \\
&= 1 - \frac{\sum (\hat{y}_i - y_i)^2}{\sum (y_i - \bar{y})^2}
\end{aligned}$$

En los modelos de regresión lineal simple el valor de  $R^2$  es el cuadrado del *coeficiente de correlación de Pearson (r)* entre  $x$  e  $y$ , lo que no ocurre en la regresión múltiple. En los modelos lineales múltiples, cuantas más predictoras se incluyan en el modelo, mayor es el valor de  $R^2$ . Esto es así ya que, por poco que sea, cada predictor va a explicar una parte de la variabilidad observada en  $y$ . Por eso  $R^2$  no puede utilizarse para comparar modelos con distinto número de predictoras.

- **$R^2$  ajustado:** una penalización al valor de  $R^2$  por cada predictora que se añade al modelo. El valor de la penalización depende del número de predictoras utilizadas y del tamaño de la muestra, es decir, del número de grados de libertad. Cuanto mayor es el tamaño de la muestra, más predictoras se pueden incorporar en el modelo.  $R^2$  ajustado permite encontrar el mejor modelo, aquel que consigue explicar mejor la variabilidad de  $y$  con el menor número de predictoras.

$$R_{ajustado}^2 = 1 - \frac{SSE}{SST} x \frac{n-1}{n-p-1} = R^2 - (1-R^2) \frac{n-1}{n-p-1} = 1 - \frac{SSE/df_e}{SST/df_t}$$

Siendo  $SSE$  la suma de cuadrados de la variabilidad explicada por el modelo,  $SST$  la variabilidad total de  $y$ ,  $n$  el tamaño de la muestra y  $p$  el número de predictoras usadas en el modelo.

### SIGNIFICANCIA DEL MODELO F-test

Uno de los primeros resultados que hay que evaluar al ajustar un modelo es el resultado del **test de significancia F**. Este contraste responde a la pregunta de si el modelo en su conjunto es capaz de predecir la variable respuesta mejor de lo esperado por azar, o lo que es equivalente, si al menos una de las predictoras que forman el modelo contribuye de forma significativa. Para realizar este contraste se compara la suma de residuos cuadrados del modelo de interés con la del modelo sin predictores, formado únicamente por la media (también conocido como suma de cuadrados corregidos por la media,  $TSS$ ).

$$F = \frac{(TSS - RSS)/(p-1)}{RSS/(n-p)}$$

Con frecuencia, la hipótesis nula y alternativa de este test se describen como:

- **$H_0$** :  $\beta_1 = \dots = \beta_{p-1} = 0$
- **$H_a$** : al menos un  $\beta_i \neq 0$

Si el test  $F$  resulta significativo, implica que el modelo es útil, pero no que sea el mejor. Puede ocurrir que algunos de sus predictores no sean necesarios y aun así funcione bien.

### 6.3.1. SIGNIFICANCIA DEL MODELO Y LAS PREDICTORAS.

#### SIGNIFICANCIA DE LAS PREDICTORAS

En la mayoría de casos, aunque el estudio de regresión se aplica a una muestra, el objetivo último es obtener un modelo lineal que explique la relación entre las variables en toda la población. Esto significa que, el modelo generado, es una estimación de la relación poblacional a partir de la relación que se observa en la muestra  $y$ , por lo tanto, está sujeta a variaciones. Para cada uno de los coeficientes de la ecuación de regresión lineal ( $\beta_j$ ) se puede calcular su significancia ( $p$ -value) y su intervalo de confianza. El test estadístico más empleado es el **t-test** (existen alternativas no paramétricas). El test de significancia de los coeficientes  $\beta_j$  del modelo considera como hipótesis:

- **$H_0$** : la predictora  $x_j$  no contribuye al modelo ( $\beta_j=0$ ) en presencia del resto de predictoras. En el caso de regresión lineal simple, se puede interpretar también como que no existe relación lineal entre ambas variables por lo que la pendiente del modelo es cero  $\beta_j=0$ .
- **$H_a$** : la predictora  $x_j$  sí contribuye al modelo ( $\beta_j\neq 0$ ), en presencia del resto de predictoras. En el caso de regresión lineal simple, se puede interpretar también como que sí existe relación lineal entre ambas variables por lo que la pendiente del modelo es distinta de cero  $\beta_j\neq 0$ .

Cálculo del estadístico  $T$  y del  $p$ -value:

$$t = \frac{\hat{\beta}_j}{se(\hat{\beta}_j)}$$

donde

$$SE(\hat{\beta}_j)^2 = \frac{\sigma^2}{\sum_{i=1}^n (x_{ji} - \bar{x})^2}$$

La varianza del error  $\sigma^2$  se estima a partir del *Residual Standard Error (RSE)*, que puede entenderse como la diferencia promedio que se desvía *target* de la verdadera línea de regresión. En el caso de regresión lineal simple, RSE equivale a:

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

Grados de libertad ( $df$ ) = número observaciones - 2 = número observaciones - número predictores - 1

- $p\text{-value} = P(|t| > \text{valor calculado de } t)$

## INTERVALOS DE CONFIANZA Y PREDICCIÓN

Cuanto menor es el número de observaciones  $n$ , menor la capacidad para calcular el error estándar del modelo. Como consecuencia, la exactitud de los coeficientes de regresión estimados se reduce. Esto tiene importancia sobre todo en la regresión múltiple.

En los modelos generados con **statsmodels** se devuelve, junto con el valor de los coeficientes de regresión, el valor del estadístico  $t$  obtenido para cada uno, los  $p\text{-value}$  y los intervalos de confianza correspondientes. Esto permite saber, además de las estimaciones, si son significativamente distintos de 0, es decir, si contribuyen al modelo.

Para que los cálculos anteriores sean válidos, se tiene que asumir que los residuos son independientes y que se distribuyen de forma normal con media 0 y varianza  $\sigma^2$ . Cuando la condición de normalidad no se satisface, existe la posibilidad de recurrir a los test de permutación para calcular significancia ( $p\text{-value}$ ) y al *bootstrapping* para calcular intervalos de confianza.

$$\hat{\beta}_j \pm t_{df}^{\alpha/2} SE(\hat{\beta}_1)$$

Dado que el modelo se ha obtenido a partir de una muestra, las estimaciones de los coeficientes de regresión tienen un error asociado y por lo tanto, también lo tienen los valores de las predicciones generadas con ellos. Existen dos formas de medir la incertidumbre asociada con una predicción:

- **Intervalos de confianza:** intervalo del valor **promedio** esperado de la variable target y para un determinado valor  $x$ .

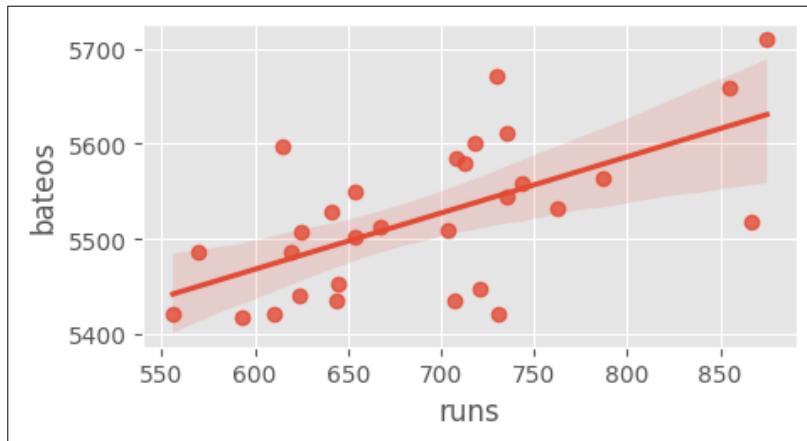
$$\hat{y} \pm t_{\alpha/2,n-2} \sqrt{MSE \left( \frac{1}{n} + \frac{(x_k - \bar{x})^2}{\sum(x_i - \bar{x})^2} \right)}$$

- **Intervalos de predicción:** intervalo del valor esperado de la variable target y para un determinado valor  $x$ .

$$\hat{y} \pm t_{\alpha/2,n-2} \sqrt{MSE \left( 1 + \frac{1}{n} + \frac{(x_k - \bar{x})^2}{\sum(x_i - \bar{x})^2} \right)}$$

Ambos parecen similares, pero la diferencia se encuentra en que los intervalos de confianza se aplican al valor **promedio** que se espera de  $y$  para un determinado valor  $x$ , mientras que los intervalos de predicción no se aplican al promedio. Por esta razón, los segundos siempre son más amplios que los primeros.

Una característica que proviene de la forma en que se calcula el margen de error en los intervalos de confianza y predicción, es que el intervalo se ensancha a medida que los valores de  $x$  se aproximan a los extremos del rango observado.



¿Por qué ocurre esto? Prestando atención a la ecuación del error estándar del intervalo de confianza, el numerador contiene el término  $(x_k - \bar{x})^2$  (lo mismo ocurre para el intervalo de predicción).

$$\sqrt{MSE \left( \frac{1}{n} + \frac{(x_k - \bar{x})^2}{\sum_{i=1}^n (x_i - \bar{x})^2} \right)}$$

Este término se corresponde con la diferencia al cuadrado entre el valor  $x_k$  para el que se hace la predicción y la media  $\bar{x}$  de los valores observados del predictor  $x$ . Cuanto más se aleje  $x_k$  de  $\bar{x}$  mayor es el numerador y por lo tanto el error estándar.

### 6.3.2. CONDICIONES PARA LA REGRESIÓN LINEAL.

Para que un modelo de regresión lineal por mínimos cuadrados y las conclusiones derivadas de él, sean completamente válidas, se deben verificar que se cumplen las suposiciones sobre las que se basa su desarrollo matemático. En la práctica, rara vez se cumplen, o se puede demostrar que se cumplen todas, sin embargo esto no significa que el modelo no sea útil. Lo importante es ser consciente de ellas y del impacto que esto tiene en las conclusiones que se extraen del modelo. Veamos algunas de ellas.

#### NI COLINEALIDAD NI MULTICOLINEALIDAD

En los modelos lineales múltiples, las predictoras deben ser independientes, no debe de haber colinealidad entre ellas. Las predictoras deberían tener una fuerte relación con la variable dependiente. Sin embargo, una predictora no debería tener una fuerte correlación con otra predictora (colinealidad) ni varias predictoras correlacionarse con otra (multicolinealidad). La consecuencia de la colinealidad y la multicolinealidad, es que no se puede identificar de forma precisa el efecto individual que tiene cada predictora sobre target, lo que se traduce en un incremento de la varianza de los coeficientes de regresión estimados hasta el punto de que resulta imposible establecer su significancia estadística. Además, pequeños cambios en los datos, provocan grandes cambios en las estimaciones de los coeficientes. Si bien la colinealidad propiamente dicha existe solo si el coeficiente de correlación simple o múltiple entre predictores es 1, cosa que raramente ocurre en la realidad, es frecuente encontrar la llamada **casi-colinealidad** o **multicolinealidad no perfecta**.

No existe un método estadístico concreto para determinar la existencia de colinealidad o multicolinealidad entre las predictoras de un modelo de regresión, sin embargo, se han desarrollado

numerosas reglas prácticas que tratan de determinar en qué medida afectan al modelo. Los pasos recomendados a seguir son:

- Si el coeficiente de determinación  $R^2$  es alto pero ninguna de las predictoras resulta significativa, hay indicios de colinealidad.
- **Crear una matriz de correlación** en la que se calcula la relación lineal entre cada par de predictoras. Es importante tener en cuenta que, a pesar de no obtenerse ningún coeficiente de correlación alto, no está asegurado que no exista multicolinealidad. Se puede dar el caso de tener una relación lineal casi perfecta entre 3 o más variables y que las correlaciones simples entre pares de estas mismas variables no sean mayores que 0.5.
- **Generar un modelo de regresión lineal simple para que cada una de las predictoras con el target del resto.** Si en alguno de los modelos el coeficiente de determinación  $R^2$  es alto, estaría señalando a una posible colinealidad.
- **Tolerancia (TOL) y Factor de Inflación de la Varianza (VIF).** Se trata de dos parámetros que vienen a cuantificar lo mismo (uno es el inverso del otro).

**VIF (Variance Inflation Factor)** es un indicador de su existencia y la librería **Statsmodels** tiene una función para calcular el **VIF** de cada variable independiente (predictora) con cada otra.

- Un valor mayor de 10 es el límite que indica una alta **colinealidad**.
- Si **VIF** es 1 significa que no hay correlación.
- Si  $1 < VIF < 5$  hay una correlación moderada.

$$VIF_i = \frac{1}{1 - R_i^2}$$

Donde  $R_i^2$  es el coeficiente de determinación de la variable  $X_i$ . El  $TOL_i = 1 / VIF_i$

**EJEMPLO 17:** Generar la matriz de correlación de un modelo:

```
import pandas as pd
datos = pd.read_csv('BMSI.csv')
from statsmodels.stats.outliers_influence import variance_inflation_factor

datos['Gender'] = datos['Gender'].map({'Male':0, 'Female':1})
X = datos[['Gender', 'Height', 'Weight']]
vif_datos = pd.DataFrame()
vif_datos["feature"] = X.columns
vif_datos["VIF"] = [variance_inflation_factor(X.values, i) for i in range(len(X.columns))]
print(vif_datos) # height y weight tienen correlaciones. Si usas los dos tienes colinealidad
```

También podemos generar un gráfico que muestre la **matriz de correlaciones** para detectarlas visualmente. Vamos a usar ahora un dataset que nos obligue a codificar algunas características para poder hacer los cálculos.

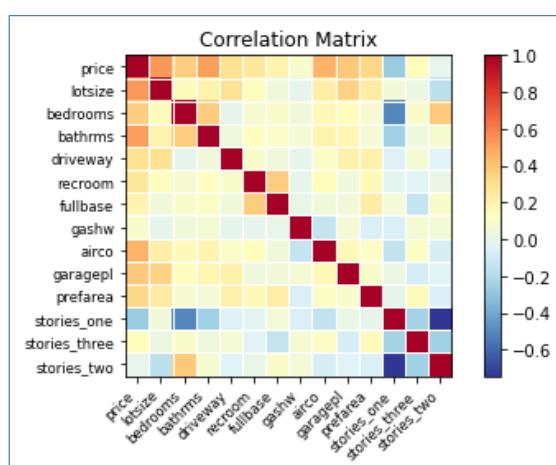


Figura 51: Matriz de correlaciones del ejemplo.

**EJEMPLO 18:** Generar la matriz de correlación de un modelo:

```
# -*- coding: utf-8 -*-
import statsmodels.graphics.api as smg
import pandas as pd
from sklearn import preprocessing
import matplotlib.pyplot as plt
# Cargar datos
df = pd.read_csv('Housing_Modified.csv')
# Convertir campos binarios a campos booleanos numéricos
lb = preprocessing.LabelBinarizer()
df.driveway = lb.fit_transform(df.driveway)
df.recroom = lb.fit_transform(df.recroom)
df.fullbase = lb.fit_transform(df.fullbase)
df.gashw = lb.fit_transform(df.gashw)
df.airco = lb.fit_transform(df.airco)
df.prefarea = lb.fit_transform(df.prefarea)
# Crea variables dummy para stories
df_stories = pd.get_dummies(df['stories'], prefix='stories', drop_first=True)
# Unir las variables dummy al dataframe principal
df = pd.concat([df, df_stories], axis=1)
del df['stories']
# Dibujar la matriz de correlaciones usando plot_corr() de Statsmodel
corr = df.corr()           # crea matriz de correlaciones
sm.graphics.plot_corr(corr, xnames=list(corr.columns))
plt.show()
```

Podemos observar que `stories_one` y `stories_two` tienen una alta correlación y si usamos las dos el modelo tendrá multicolinealidad.

**EJEMPLO 19:** Calcular y eliminar predictoras que tengan un **VIF** que hagan propenso a tener colinealidad al modelo:

```
# -*- coding: utf-8 -*-
# Medir el VIF de cada predictora y eliminar la mayor que esté por encima del límite
import numpy as np
import pandas as pd
from statsmodels.stats.outliers_influence import variance_inflation_factor
from sklearn import preprocessing

df = pd.read_csv('Housing_Modified.csv')          # Cargar datos
# Convertir campos binarios a campos booleanos numéricos
lb = preprocessing.LabelBinarizer()
df.driveway = lb.fit_transform(df.driveway)
df.recroom = lb.fit_transform(df.recroom)
df.fullbase = lb.fit_transform(df.fullbase)
df.gashw = lb.fit_transform(df.gashw)
df.airco = lb.fit_transform(df.airco)
df.prefarea = lb.fit_transform(df.prefarea)
# Crea variables dummy para stories
df_stories = pd.get_dummies(df['stories'], prefix='stories', drop_first=True)
df = pd.concat([df, df_stories], axis=1)      # Unir variables dummy al dataframe
del df['stories']                                # Borrar la original
# Codificar las dummies como numéricas 1 = True, 0=False para hacer cálculos
df['stories_one'] = df['stories_one'].map({True:1, False:0})
df['stories_two'] = df['stories_two'].map({True:1, False:0})
df['stories_three'] = df['stories_three'].map({True:1, False:0})
# Definir la lista de columnas predictoras a través de sus nombres
preds = ['lotsize', 'bedrooms', 'bathrms', 'driveway', 'recroom', 'fullbase', 'gashw',
         'airco', 'garagepl', 'prefarea', 'stories_one', 'stories_two', 'stories_three']
X = df[preds]
limite = 10
for i in np.arange(0, len(preds)):
    vif= [variance_inflation_factor(X[preds].values,i) for ix in range(X[preds].shape[1])]
    maxloc = vif.index(max(vif))
    if max(vif) > limite:
        print("vif :", vif)
```

```

print('Borrar \' + X[predictoras].columns[maxloc] + '\' en índice: ' + str(maxloc))
del predictoras[maxloc]
else:
    break
print('Variables Finales:', predictoras)

```

Si ejecutamos el análisis *VIF* al mismo modelo veremos que también indica las mismas columnas y sugiere eliminar *bedrooms* y mantiene *stories\_one* e *stories\_two*.

- De igual manera, como tolerancia es  $1 / VIF$ , los límites recomendables están entre 1 y 0.1.

En caso de encontrar colinealidad entre predictoras, hay dos posibles soluciones:

- **La primera:** excluir una de las predictoras problemáticas intentando conservar la que a juicio del investigador está influyendo realmente en el *target*. Esta medida no suele tener mucho impacto en la capacidad predictiva del modelo porque al existir colinealidad, la información que aporta una de las predictoras es redundante y sigue estando presente en la otra.
- **La segunda:** consiste en combinar las variables colineales en una única predictora, aunque con el riesgo de perder su interpretación.

Cuando se intenta establecer relaciones causa-efecto, la colinealidad puede llevar a conclusiones muy erróneas, haciendo creer que una variable es la causa de un resultado cuando, en realidad, es otra la que está influenciando sobre esa predictora. Es decir, puede desviar la atención de las verdaderas causas de algo.

### Relación lineal entre las predictoras numéricas y target

Cada predictora numérica tiene que estar linealmente relacionada con target mientras los demás predictores se mantienen constantes, de lo contrario, no se deben usar en el modelo. La forma más recomendable de comprobarlo es representando los residuos del modelo con cada una de las predictoras. Si la relación es lineal, los residuos se distribuyen de forma aleatoria en torno a cero. Estos análisis son solo aproximados, ya que no hay forma de saber si realmente la relación es lineal cuando el resto de predictoras se mantienen constantes.

### Distribución normal de target

La variable target se tiene que distribuir de forma normal. Para comprobarlo se recurre a histogramas, a los cuantiles normalizados o a test de hipótesis de normalidad.

### Varianza constante de target (homocedasticidad)

**La varianza de target debe ser constante en todo el rango de las predictoras.** Para comprobarlo suelen representarse los residuos del modelo con cada predictora. Si la varianza es constante, se distribuyen de forma aleatoria manteniendo una misma dispersión y sin ningún patrón específico. Una distribución cónica es un claro identificador de falta de homocedasticidad. También se puede recurrir a contrastes de homocedasticidad como *el test de Breusch-Pagan*.

La razón de esta condición reside en lo siguiente: los modelos lineales asumen que el target  $y$  sigue una distribución normal, cuya media  $\mu$  puede ser modelada en función de otras variables (predictoras) y cuya varianza  $\sigma^2$  se calcula mediante una constante de dispersión y una función  $v(\mu)$ . Esto último significa que la varianza no se modela directamente en función de las variables predictoras sino de forma indirecta a través de su relación con la media y es un valor único.

$$\hat{\sigma}^2 = \frac{\sum_{i=1}^n \hat{\epsilon}_i^2}{n - p} = \frac{\sum_{i=1}^n (y_i - \hat{\mu})^2}{n - p}$$

¿Qué impacto puede tener esto en la práctica? Aunque las estimación de las medias obtenidas por estos modelos sean buenas, no lo son tanto las incertidumbres asociadas y en consecuencia los intervalos de confianza y de predicción que se puedan calcular.

### No autocorrelación (Independencia)

Los valores de cada observación son independientes de los otros. Esto es especialmente importante de comprobar cuando se trabaja con mediciones temporales. Se recomienda representar los residuos ordenados acorde al tiempo de registro de las observaciones, si existe un cierto patrón hay indicios de autocorrelación. También se puede emplear el *test de hipótesis de Durbin-Watson*.

### Valores atípicos, con alto leverage o influyentes

Es importante identificar observaciones que sean atípicas o que puedan estar influenciando al modelo. La forma más fácil de detectarlos es a través de los residuos.

### Tamaño de la muestra

No se trata de una condición de por sí pero, si no se dispone de suficientes observaciones, predictores que no son realmente influyentes podrían parecerlo. Una recomendación frecuente es que el número de ejemplos sea como mínimo entre 10 y 20 veces el número de predictoras del modelo.

### Parsimonia

Este término hace referencia a que, el mejor modelo, es aquel capaz de explicar con mayor precisión la variabilidad observada en el target empleando el menor número de predictoras, por lo tanto, con menos suposiciones.

La gran mayoría de condiciones se verifican utilizando los residuos, por lo tanto, se suele generar primero el modelo y posteriormente validar las condiciones. De hecho, el ajuste de un modelo debe verse como un proceso iterativo en el que se ajusta el modelo, se evalúan sus residuos y se mejora. Así hasta llegar a un modelo óptimo. En este ejemplo vamos a construir un modelo de regresión usando **statsmodels** y le vamos a pedir que nos resuma sus características. Despues las explicaremos y comentamos algunos parámetros que aparecen.

**EJEMPLO 20:** construir el modelo de regresión lineal multivariado con **statsmodels** y analizar sus parámetros estadísticos (se explican a continuación).

```
# -*- coding: utf-8 -*-
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn import preprocessing

df = pd.read_csv('Housing_Modified.csv') # Cargar datos
# Convertir campos binarios a campos booleanos numéricos
lb = preprocessing.LabelBinarizer()
df.driveway = lb.fit_transform(df.driveway)
df.recroom = lb.fit_transform(df.recroom)
df.fullbase = lb.fit_transform(df.fullbase)
df.gashw = lb.fit_transform(df.gashw)
df.airco = lb.fit_transform(df.airco)
df.prefarea = lb.fit_transform(df.prefarea)
# Crea variables dummy para stories
df_stories = pd.get_dummies(df['stories'], prefix='stories', drop_first=True)
df = pd.concat([df, df_stories], axis=1)    # Unirlas al dataset
del df['stories']
# Codificar las dummies como numéricas 1 = True, 0=False para hacer cálculos
df['stories_one'] = df['stories_one'].map({True:1, False:0})
df['stories_two'] = df['stories_two'].map({True:1, False:0})
df['stories_three'] = df['stories_three'].map({True:1, False:0})
# Definir el problema: target ~ predictoras
```

```

predictoras = ['lotsize', 'bathrms','driveway', 'recroom', 'fullbase', 'gashw', 'airco',
'garagepl', 'prefarea', 'stories_one','stories_two','stories_three']
X = df[predictoras]
y = df['price']
# Dividir datos en 80/20 para train/test
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=.80, random_state=1)
# crea y entrena un modelo de mínimos cuadrados ordinario e imprimir resumen
import statsmodels.api as sm
x = X_train
y = y_train
x = sm.add_constant(x)
ols = sm.OLS(y,x).fit()
print(ols.summary()) # Resumen estadístico del modelo
# Hacer predicciones sobre el dataset test
y_train_pred = ols.predict(x)
X_test = sm.add_constant(X_test)
y_test_pred = ols.predict(X_test)
print("Train MAE: ", metrics.mean_absolute_error(y_train, y_train_pred))
print("Train RMSE: ", np.sqrt(metrics.mean_squared_error(y_train, y_train_pred)))
print("Test MAE: ", metrics.mean_absolute_error(y_test, y_test_pred))
print("Test RMSE: ", np.sqrt(metrics.mean_squared_error(y_test, y_test_pred)))

```

Hay como 3 zonas en los resultados. La primera describe datos del modelo como la fecha en que se genera, cual es la columna target, cuántos ejemplos se han usado para entrenar, los grados de libertad (Df Model) es decir, la cantidad de parámetros, y a la derecha algunos parámetros estadísticos. Por ejemplo el R-squared es el  $R^2$ , recuerda que indica que porcentaje de la varianza del target explica el modelo. No está mal, explica el 95.4%.

OLS Regression Results									
Dep. Variable:	price	R-squared:	0.954						
Model:	OLS	Adj. R-squared:	0.953						
Method:	Least Squares	F-statistic:	731.3						
Date:	Mon, 05 Dec 2016	Prob (F-statistic):	1.46e-274						
Time:	23:39:09	Log-Likelihood:	-4828.1						
No. Observations:	436	AIC:	9680.						
Df Residuals:	424	BIC:	9729.						
Df Model:	12								
Covariance Type:	nonrobust								
	coef	std err	t	P> t	[95.0% Conf. Int.]				
lotsize	3.7626	0.401	9.619	0.000	3.074	4.651			
bathrms	2.005e+04	1417.881	14.119	0.000	1.73e+04	2.28e+04			
driveway	1.198e+04	2146.338	5.581	0.000	7759.759	1.62e+04			
recroom	3285.2328	2188.248	1.501	0.134	-1016.010	7586.476			
fullbase	4775.3171	1831.641	2.607	0.009	1175.090	8375.544			
gashw	1.42e+04	3554.187	3.995	0.000	7214.613	2.12e+04			
airco	1.425e+04	1791.906	7.950	0.000	1.07e+04	1.78e+04			
garagepl	4652.5499	983.036	4.735	0.000	2720.319	6584.781			
prefarea	8080.7376	2026.326	3.988	0.000	4097.842	1.21e+04			
stories_one	-5273.4580	2227.269	-2.368	0.018	-9651.321	-895.595			
stories_two	469.8222	228.422	2.025	0.037	-4028.241	4967.886			
stories_three	1.072e+04	3519.387	3.047	0.002	3804.264	1.76e+04			
Omnibus:		22.770	Durbin-Watson:		1.964				
Prob(Omnibus):		0.000	Jarque-Bera (JB):		35.535				
Skew:		0.383	Prob(JB):		1.92e-08				
Kurtosis:		4.170	Cond. No.		3.10e+04				

Parameter estimates  
Standard error of the Parameter estimates  
T value for null hypothesis  
exact probability of t  
95% confidence interval of parameters  
Value around 2 indicates no multicollinearity

Figura 52: Resumen de un modelo ofrecido OLS por la función `summary()`.

Debajo está el **Adjusted R-squared**: El  **$R^2$  ajustado** es una modificación del anterior, en el que introduce una penalización para el caso de la regresión multivariada, de forma que si se añaden columnas a un modelo es de esperar que aumente su  $R^2$ . Pero si esto no ocurre, se penaliza que sea más complejo que la versión con menos predictoras. Nos sirve para penalizar la complejidad si no se obtienen mejoras, así que es una forma de comparar diferentes modelos con más o menos predictoras. La fórmula es esta, donde  $N$  es el número de ejemplos y  $p$  el número de predictoras:

$$\text{Adjusted } R^2 = 1 - \frac{(1 - R^2)(N - 1)}{N - p - 1}$$

La figura 53 muestra como cambia el  $R^2$  y el  $R^2$  ajustado cuando se van incrementando la cantidad de predictoras que se utilizan. El  $R^2$  aumenta siempre que se añaden predictoras, incluso cuando estas no aportan ninguna explicación útil al modelo. Sin embargo el  $R^2$  ajustado se desploma en ese caso.

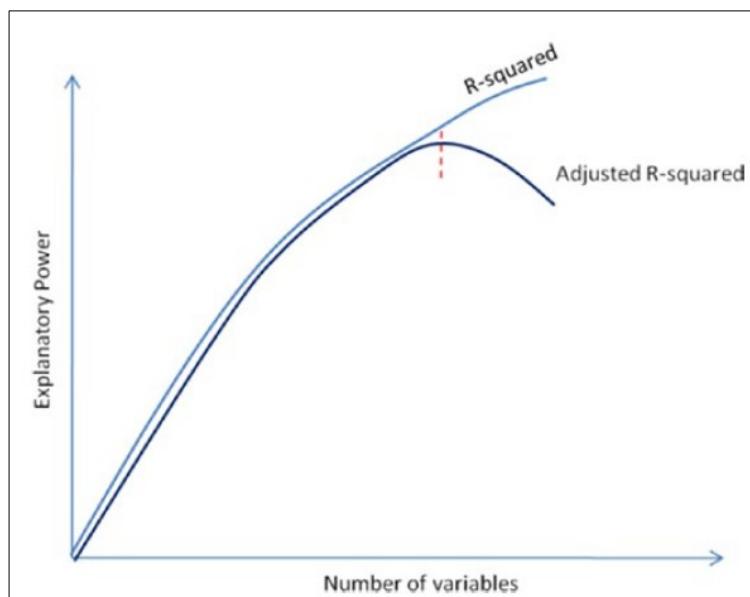


Figura 53:  $R^2$  contra  $R^2$  ajustado.

La segunda zona describe propiedades estadísticas de cada parámetro o característica. Algunos ya los conocemos, por ejemplo la columna *coef* define los parámetros del modelo (son los valores de  $\theta_1$ ,  $\theta_2$ , ...,  $\theta_p$ ). Explicamos algunos de ellos.

- **Standard error:** La distancia media desde la recta a los valores reales de las etiquetas. Cuanto más pequeño mejor.
- **t y p-value:** *p-value* es un estadístico importante. Para comprenderlo mejor necesitamos comprender el concepto de contraste de hipótesis y la distribución normal. El contraste de hipótesis es una afirmación que se comprueba basándonos en la observación de la distribución normal. Se realiza en varios pasos:
  - Se expresa una hipótesis.
  - Se comprueba la validez de la hipótesis. Si se encuentra que es cierta, se acepta. Si se encuentra que es falsa se rechaza. La hipótesis que es testeada para posible rechazo se llama hipótesis nula y se expresa como  $H_0$ . La hipótesis que se acepta cuando se rechaza la hipótesis nula se llama hipótesis alternativa y se escribe como  $H_a$ .

La hipótesis alternativa es con frecuencia más difícil de probar y es la interesante. Por ejemplo podemos establecer como hipótesis nula  $H_0$  que el tamaño de una casa no tiene influencia en su precio. En este caso el coeficiente asociado al tamaño será 0 en la ecuación de regresión lineal ( $y = m * \text{tamaño} + c$ ). La hipótesis alternativa a esta es que el tamaño sí que influye en el precio, es decir, su parámetro asociado no será cero en la ecuación de regresión.

Para poder decir cuando las estimaciones se acercan lo suficiente a la hipótesis para darla como cierta, tomamos el rango de la estimación comparándola con la varianza y miramos si este rango contiene el valor de la hipótesis. Para hacer esto transformamos la estimación dentro de una distribución normal estándar en la que sabemos que el 95% de todos los valores de la variable que tienen de media 0 y varianza 1 estarán dentro de 0 y 2 desviaciones estándar. Como la regresión no sa su estimación y su error estándar podemos asegurarnos de que el verdadero valor (desconocido en realidad) de  $m$  estará dentro de esta región. Mira la figura 54.

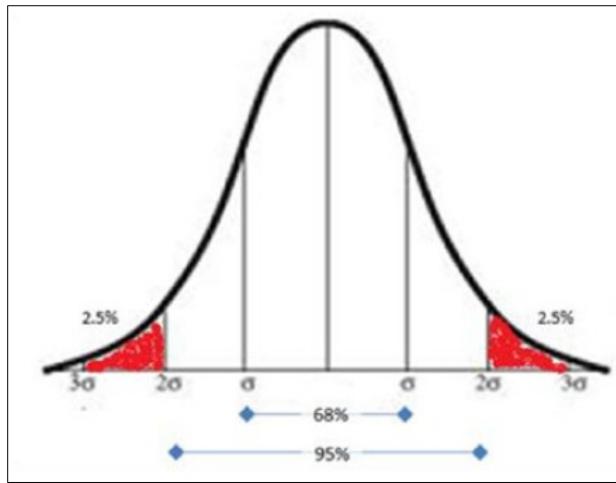


Figura 54. Distribución Normal (en rojo las zonas de rechazo)

El *t-value* se usa para determinar un *p value* (*probabilidad*), y un  $p\text{-value} \leq 0.05$  significa una fuerte evidencia contra la hipótesis nula, así que se rechaza esta hipótesis. Un  $p\text{-value} > 0.05$  significa una evidencia débil contra la hipótesis nula, así que no puedes rechazarla. Así que en nuestro caso las variables que tengan un  $p\text{-value} \leq 0.05$  son importantes para el modelo y le aportan información.

El proceso de comprobación de una hipótesis indica que hay posibilidades de cometer un error. Hay siempre dos posibles errores en cualquier dataset y estos errores están inversamente relacionados, cuando uno es pequeño el otro es mayor y al contrario.

- El primer tipo de error es que rechacemos la hipótesis nula  $H_0$  cuando en realidad sea cierta.
- El segundo tipo de error es aceptar la hipótesis nula  $H_0$  cuando en realidad es falsa.

Observa por ejemplo que la variable *stores\_three* y *recroom* tienen un alto valor de  $p$  lo que significa que son insignificantes. Si ejecutamos el modelo de regresión sin estas variables y vemos el resultado, nos daremos cuenta que borrarlas no ha tenido ningún impacto en el  $R^2$  ajustado.

- **Confidence interval:** es el coeficiente que calcula el 95% del intervalo de confianza para la pendiente de la variable *target*.

En la tercera zona aparecen parámetros de todo el conjunto, por ejemplo:

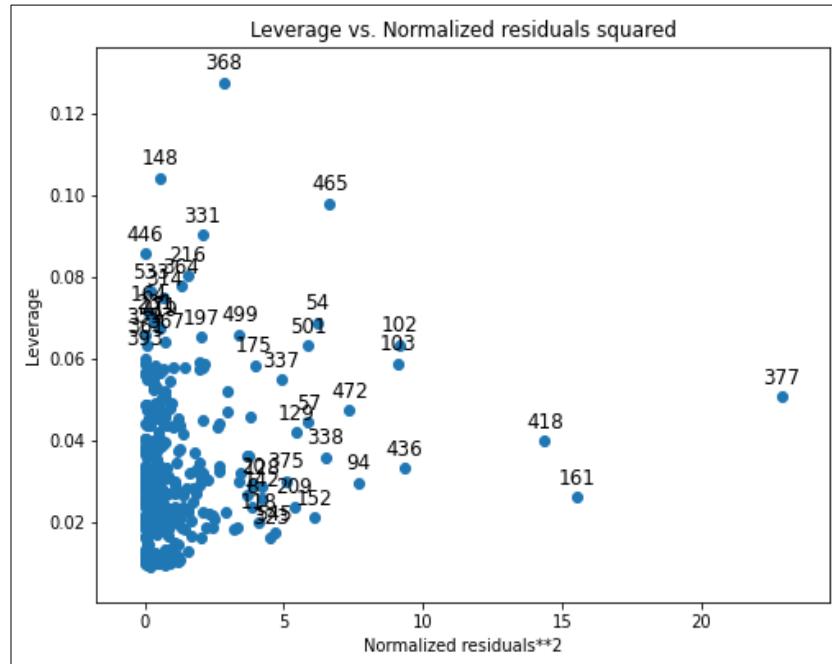
- **Durbin-Watson:** es uno de los estadísticos usados para detectar si hay multicolinealidad, es decir, dos o más predictoras están muy correlacionadas. Su valor siempre está entre 0 y 4. Un valor cercano a 2 sería el ideal (entre 1.5 y 2.5 es normal y nos indicaría que no hay correlaciones entre las predictoras).

## OUTLIERS

Los datos que son anómalos quedan muy alejados de la línea de regresión y provocan que el modelo pierda precisión. Dibujar los residuos normalizados contra el provecho (*Leverage*) nos da una buena comprensión de los puntos que son outliers. Los residuos son las diferencias entre los valores predichos y los que predice el modelo, y el provecho es una medida de como de lejos se encuentran los valores del target de una observación con respecto a las otras observaciones.

**EJEMPLO 21: Dibujar los residuos contra el aprovechamiento.**

```
# Dibujamos los residuos normalizados contra el leverage
import matplotlib.pyplot as plt
from statsmodels.graphics.regressionplots import plot_leverage_resid2
fig, ax = plt.subplots(figsize=(8,6))
fig = plot_leverage_resid2(modelo, ax = ax)
```



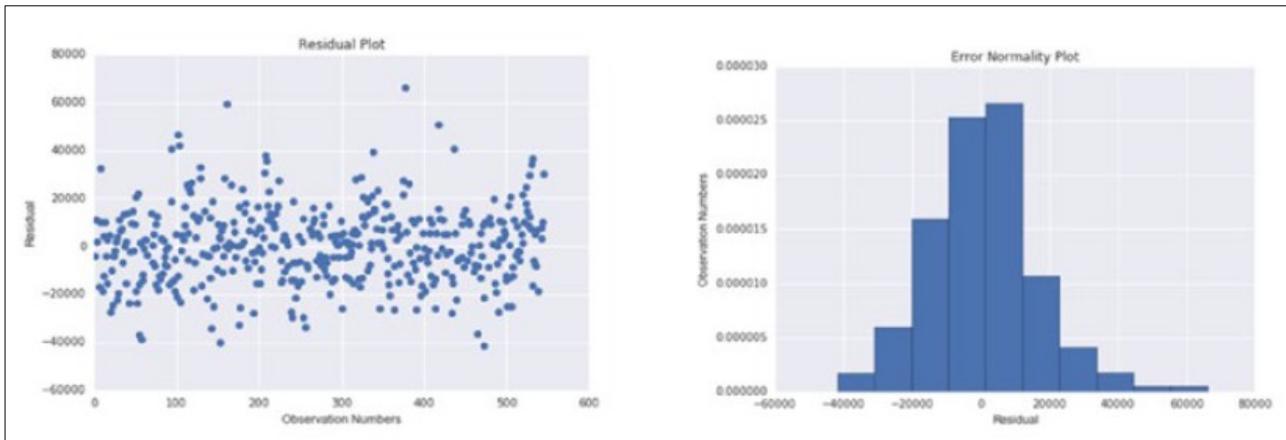


Figura 56. gráficos de residuos y su distribución.

## LINEALIDAD

Las relaciones entre las predictoras y el target debería ser lineal. Si la relación no es lineal sería bueno transformar los datos de la predictoras o del target (como aplicar logaritmos, raíz cuadrada, polinomios de orden superior, etc.).

### EJEMPLO 24: Generar gráficos de linealidad del target con respecto de las predictoras.

```
import statsmodels.api as sm
f= plt.figure(figsize=(10,15))
f= sm.graphics.plot_partregress_grid(modelo,fig=f)
```

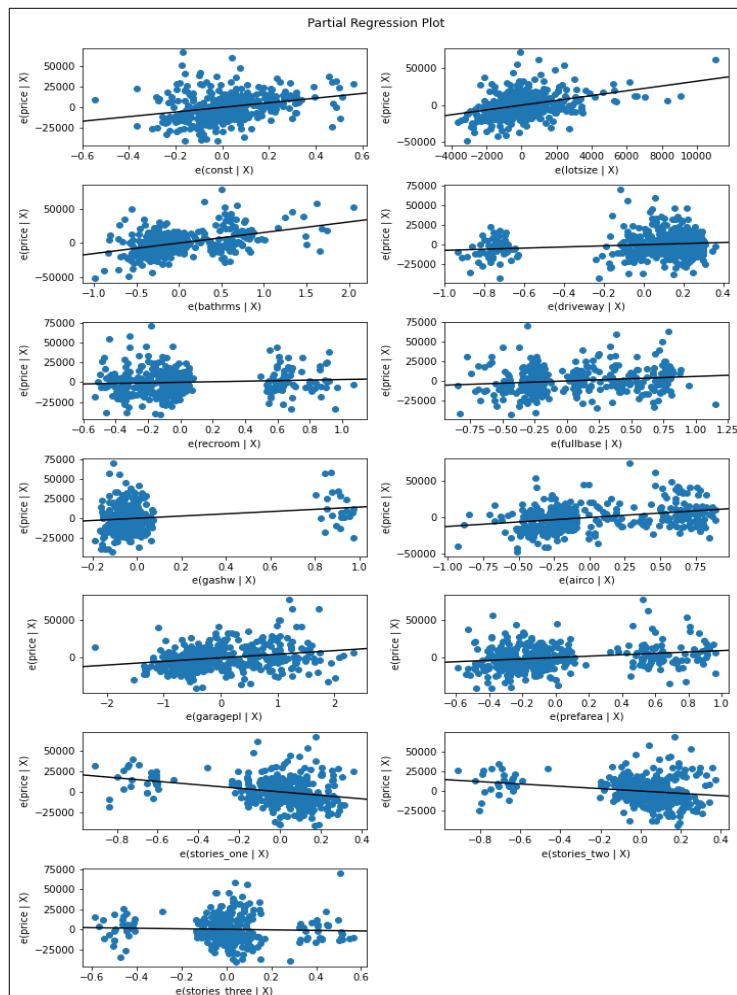


Figura 57. gráficos de rejilla de regresiones parciales con cada predictor.

## 7. ALGORITMOS LINEALES GENERALIZADOS.

Los modelos lineales generalizados (**GLM**) extienden a los modelos lineales de dos formas:

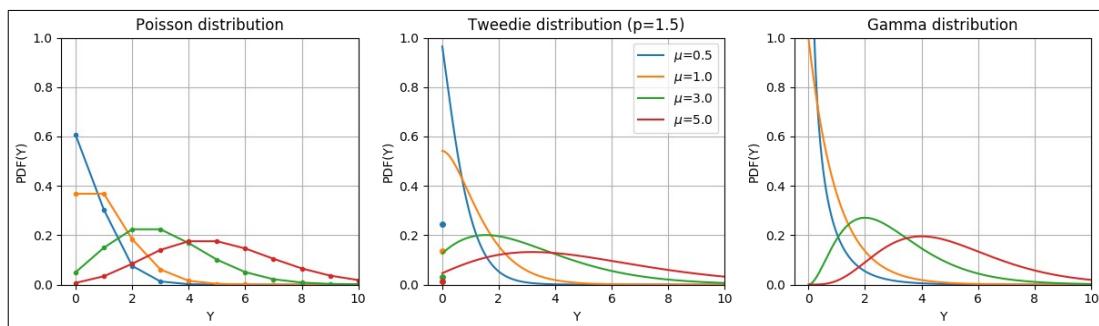
- **Primera:** los valores  $\hat{y}$  que se predicen están asociados (**link**) con una combinación lineal de la entrada  $X$  a través de la inversa de una función  $h$ :  $\hat{y} = h(X)w$
- **Segunda:** el cuadrado de la función de coste se sustituye por la desviación unidad  $d$  de una distribución aleatoria de la familia de distribuciones exponenciales. Por tanto, el problema de la minimización se convierte en encontrar los  $w$  que hagan mínima esta expresión:

$$\min_w \frac{1}{2n_{\text{samples}}} \sum_i d(y_i, \hat{y}_i) + \frac{\alpha}{2} \|w\|_2^2,$$

donde  $\alpha$  es la regularización L2. Cuando se tienen los pesos de la muestra, la media se puede convertir en una media ponderada. La siguiente tabla muestra varias distribuciones:

Distribution	Target Domain	Unit Deviance $d(y, \hat{y})$
Normal	$y \in (-\infty, \infty)$	$(y - \hat{y})^2$
Bernoulli	$y \in \{0, 1\}$	$2(y \log \frac{y}{\hat{y}} + (1 - y) \log \frac{1-y}{1-\hat{y}})$
Categorical	$y \in \{0, 1, \dots, k\}$	$2 \sum_{i \in \{0, 1, \dots, k\}} I(y = i) y_i \log \frac{I(y=i)}{I(\hat{y}=i)}$
Poisson	$y \in [0, \infty)$	$2(y \log \frac{y}{\hat{y}} - y + \hat{y})$
Gamma	$y \in (0, \infty)$	$2(\log \frac{\hat{y}}{y} + \frac{y}{\hat{y}} - 1)$
Inverse Gaussian	$y \in (0, \infty)$	$\frac{(y - \hat{y})^2}{y \hat{y}^2}$

Las funciones de densidad de estas distribuciones aparecen en esta imagen:



La elección de estas distribuciones dependen de tipo de problema que tengas entre manos y de qué valores son los que tiene la característica target:

- Si los valores son contables (enteros no negativos como 0, 1, 2,...) o frecuencias relativas no negativas, usas la distribución de Poisson con log-link.
- Si los valores son positivos y están sesgados puedes usar una Gamma con log-link.
- Si los valores parecen estar más distribuidos en los extremos que una distribución Gamma puedes usar la Inversa Gausiana (o una Tweedie con exponente alto).
- Si los valores representan probabilidades, puedes usar la distribución de Bernoulli y un log-link para realizar clasificaciones binarias y la distribución categórica con un softmax-link puede realizar clasificaciones multiclas.

Para usarlo en *scikit-learn* tienes el método ***TweedieRegressor()*** que te permite seleccionar la distribución a través del parámetro ***power*** (0 para distribución normal, 1 para Poisson, 2 para gamma y 3 para *gausiana inversa*).

**EJEMPLO 25:** Ajustar unos datos con un regresor lineal generalizado usando *scikit-learn*.

```
from sklearn.linear_model import TweedieRegressor
datos = [[0, 0], [0, 1], [2, 2]]
reg = TweedieRegressor(power=1, alpha=0.5, link='log')
reg.fit(datos, [0, 1, 2])
print("Coeficientes:", reg.coef_, "Punto de corte:", reg.intercept_)
```

## 7.1. REGRESIÓN POLINÓMICA.

Aunque tengamos datos que no se ajustan bien a una línea recta todavía podemos usar regresión lineal para ajustar datos no lineales. Una forma de hacerlo es añadir a nuestros datos columnas o características que sean potencias de otras características o combinaciones de potencias de características. De esta forma aumentamos la complejidad del modelo. Cuando hayamos extendido nuestros datos, les aplicamos regresión lineal. Esta técnica se llama Regresión polinómica. Aunque también podríamos usar la misma técnica con otras transformaciones como logaritmos, exponenciales, senos, cosenos, etc.

**EJEMPLO 26:** Generar datos aleatorios que no sigan una recta sino una parábola (*polinomio de grado 2:  $ax^2 + bx + c + ruido\_aleatorio$* ).

```
# -*- coding: utf-8 -*-
import numpy as np
import matplotlib.pyplot as plt
n = 100 # nº de datos
np.random.seed(0) # Hacerlo repetible
X = 6 * np.random.rand(n, 1) - 3
y = 0.5 * X**2 + X + 2 + np.random.randn(n, 1)
plt.scatter(X,y)
plt.xlabel("X")
plt.ylabel("Y")
plt.show()
```

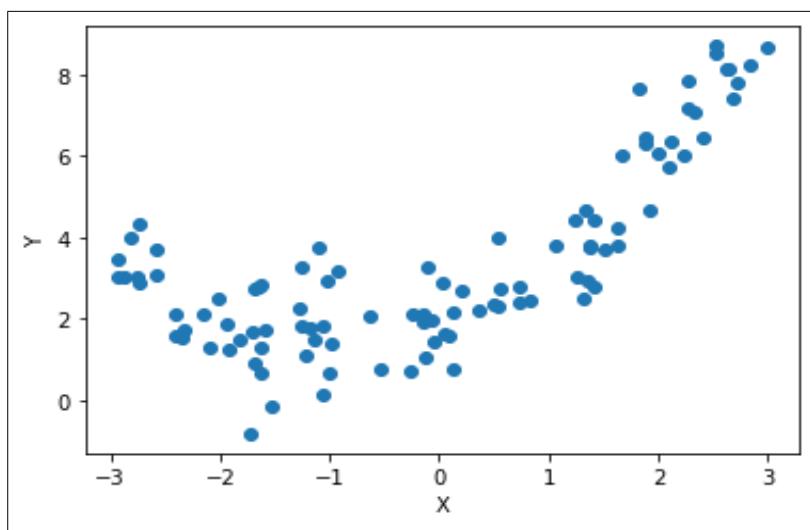


Figura 58: Datos generados que siguen una parábola  $0.5x^2+x+2+ruido$ .

Está claro que una línea recta no se va a ajustar bien a estos datos. Así que vamos a usar la clase ***PolynomialFeatures*** de *Scikit-Learn*. Nos devuelve un nuevo conjunto de datos al que añade para cada característica que tenga todas las combinaciones posibles de potencias con el grado indicado en el parámetro *degree*. Por ejemplo, si tenemos datos con las características  $[a, b]$  y hacemos la llamada con grado 2, devuelve datos con columnas  $[a^2, b^2, ab, a, b, 1]$ . Como en el ejemplo anterior solo

tenemos una característica  $[x]$ , si la llamamos con grado 2 nos devuelve  $[x^2, x, 1]$ . Luego llamando a su método `fit_transform(datos)` nos devolverá los datos aplicando esas transformaciones.

### EJEMPLO 27: Generar datos para realizar regresión polinómica de grado 2.

```
# -*- coding: utf-8 -*-
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
n = 100 # nº de datos
np.random.seed(0) # Hacerlo repetible
X = 6 * np.random.rand(n, 1) - 3
X = np.sort(X, axis=0) # Los ordeno para dibujarlos bien
y = 0.5 * X**2 + X + 2 + np.random.randn(n, 1)
# Generar datos de potencias de los originales
from sklearn.preprocessing import PolynomialFeatures
x_polinomial = PolynomialFeatures(degree=2, include_bias=False)
print(x_polinomial)
X_polinomial = x_polinomial.fit_transform(X)
print(X_polinomial)
print("bias:", X[0], "Polinomial:", X_polinomial[0])
# Aplicar el modelo
rl = LinearRegression()
rl.fit(X_polinomial, y)
print("Coeficientes B0+B1X+B2X2 B0:", rl.intercept_, "[B1,B2]:", rl.coef_)
titulo = f'Regresión Polinomial con R2: {rl.score(X_polinomial, y)}'
# Dibujar el modelo
plt.scatter(X,y)
plt.xlabel("X")
plt.ylabel("Y")
plt.plot(X, rl.predict(X_polinomial), color='red', linewidth=2)
plt.title(titulo)
plt.show()
```

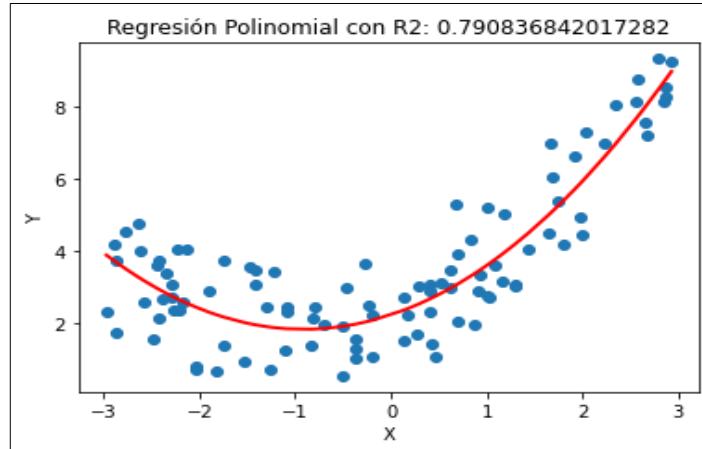


Figura 59: Regresión Polinomial de grado 2.

Ten en cuenta que cuando tengas  $n$  características la clase `PolynomialFeatures(degree=d)` genera un array que tiene  $(n + d)! / (d! n!)$  características, lo que genera una explosión de características. En estos casos quizás es más conveniente que las generes a mano. Es decir si tienes  $[a, b]$  y usas  $d = 3$  acabas con  $[a^3, b^3, a^2, b^2, ab^2, ba^2, ab, a, b, 1]$ . Si por ejemplo solo quieres  $[a^3, b^2, ab, a, b, 1]$  puedes hacer algo similar a lo mostrado en el siguiente ejemplo:

### EJEMPLO 28: Generar datos para realizar regresión polinómica de grado 3 pero usando solo $X^3 + X$ .

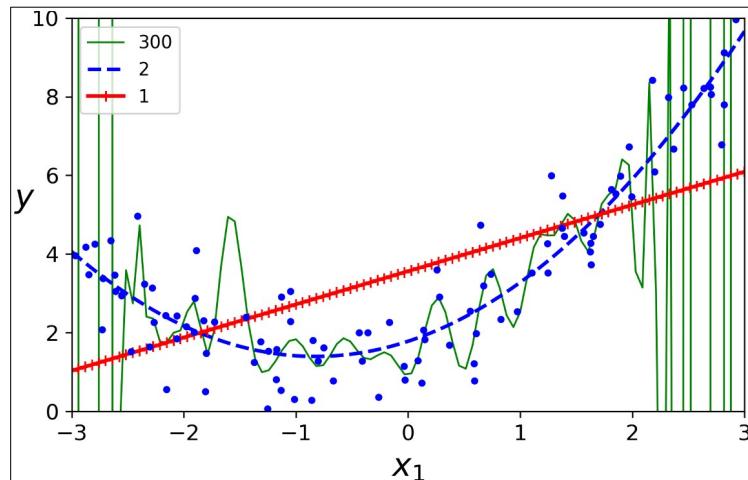
```
# -*- coding: utf-8 -*-
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
n = 100 # nº de datos
np.random.seed(0) # Hacerlo repetible
x = np.sort(10 * np.random.rand(n, 1)-5, axis=0)
```

```

print(x)
y = 0.2 * x**3 - 0.5 * x + 2 + np.random.randn(n, 1)
x_poly = np.hstack([x, x**3])
rl = LinearRegression().fit(x, y) # Añadir componentes a mano
rp = LinearRegression().fit(x_poly, y) # Un modelo lineal
# Un modelo polinómico
# crear datos de prueba para dibujar
rejilla_lineal = np.linspace(-5, 5, 100).reshape(-1, 1)
rejilla_poly = np.hstack([rejilla_lineal, rejilla_lineal**3])
plt.scatter(x, y, color="black", label="datos")
plt.plot(rejilla_lineal, rl.predict(rejilla_lineal), color="blue", label="Lineal")
plt.plot(rejilla_lineal, rp.predict(rejilla_poly), color="red", label="Polinómica")
plt.legend()
plt.show()

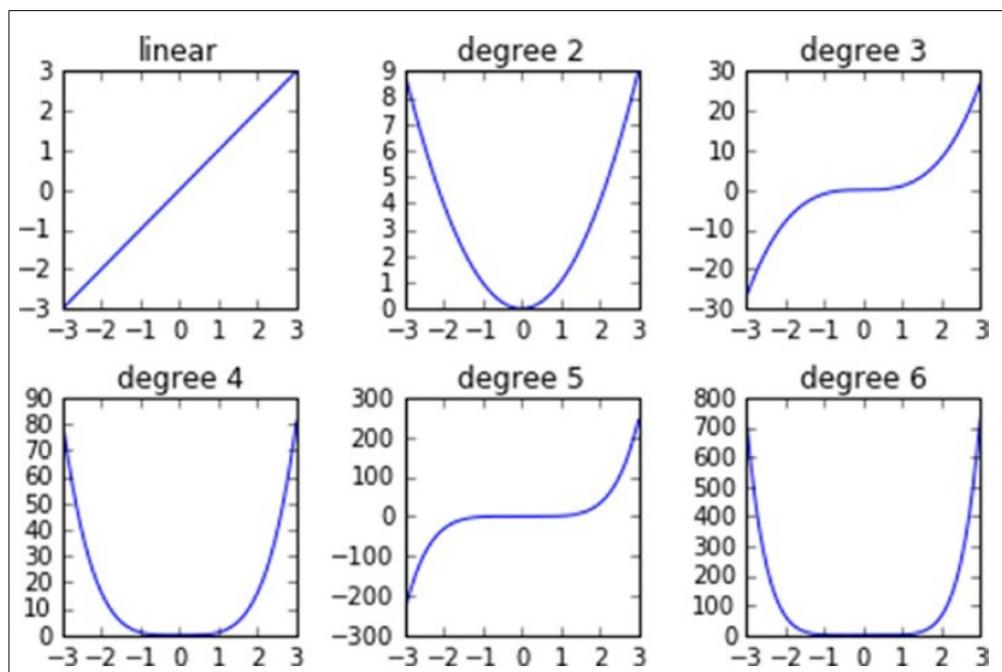
```

El peligro de los modelos polinómicos como ya hemos comentado es que sobreajusten los datos de entrenamiento. En esta figura aparece como ajustan a los datos de entrenamiento varios modelos. La línea (en rojo) es como un polinomio de grado 1, en principio es la que menos ajusta y probablemente tenga underfitting. La de azul es una parábola, un polinomio de grado 2 que probablemente sea el mejor de los tres modelos. Por último la línea verde es de un modelo de 300 grados y ajusta casi perfectamente los datos, pero no funcionará bien porque tiene overfitting.



*Figura 60: Polinomios de alto grado son propensos a sobreajustar.*

Para los datos de la figura 54 el modelo que mejor generaliza es el cuadrático.



*Figura 61: Polinomios de varios grados y las formas que adoptan.*

## 7.2. REGRESIÓN LOGÍSTICA

Algunos algoritmos de regresión también se pueden utilizar para clasificación y viceversa. La *regresión logística* (conocida como *regresión logit*) se utiliza para estimar la probabilidad de que un ejemplo pertenezca a una de dos posibles clases conocidas como positiva y negativa. Las etiquetas son valores numéricos 1 (la clase positiva) y 0 la clase negativa. Cuando se entrena el modelo con mail que es spam, su etiqueta será 1. Cuando el modelo haga una predicción, recibe un mail y responde con una estimación de la probabilidad de que el mail sea spam. Si la probabilidad estimada es mayor o igual al 50% el modelo predice que es spam (lo clasifica como de la clase 1) y en caso contrario, si la probabilidad estimada es menor de 0.5 lo clasifica como no spam.

### ¿CÓMO SE ESTIMAN LAS PROBABILIDADES?

Igual que hace un modelo de regresión lineal, el modelo de regresión logística calcula la suma de pesos de los ejemplos de entrada multiplicados por los coeficientes y le suman un bias. Pero la regresión logística no persigue un número, debe transformar ese número en una probabilidad, porque su salida es logística no numérica. Así que debe aplicar una función que haga esa transformación. A esa función la llamamos  $\sigma$  y es la función sigmoidea (tiene forma de letra S, no es la única):

$$\hat{y} = h(\theta^T x) \quad (\text{regresión lineal})$$

$$\hat{y} = h(\theta^T x) = \sigma(x^T \theta) \quad (\text{regresión logística})$$

La función logística usada es la **función sigmoide  $\sigma(t)$**  que devuelve números entre 0 y 1 ante cualquier entrada  $t$ .

$$\sigma(t) = \frac{1}{1 + \exp(-t)}$$

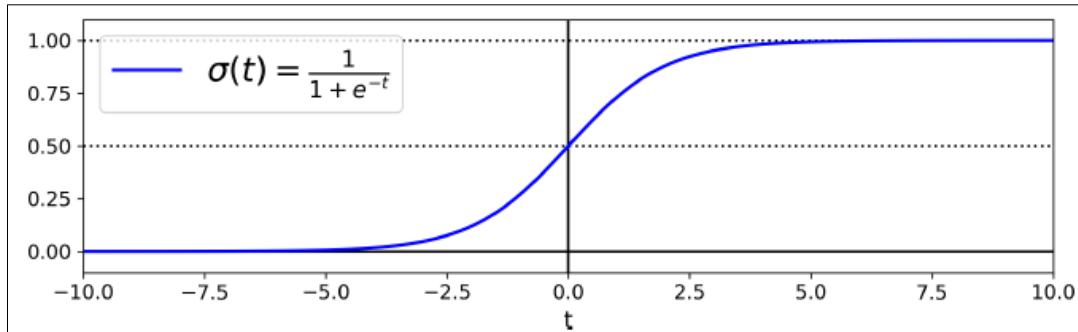


Figura 62: Función sigmoide.

En la figura 62 puedes observar que  $\sigma(t) < 0.5$  cuando  $t < 0$ , y que  $\sigma(t) \geq 0.5$  cuando  $t \geq 0$ , así que un modelo de regresión logística predice 1 cuando  $x^T \theta$  es positivo y 0 si es negativo. El valor  $t$  se llama a menudo el **logit**: este nombre se debe a que la función *logit* es la inversa de la función logística.

$$\text{logit}(p) = \ln\left(\frac{p}{1-p}\right)$$

Es decir, si calculas la función logit de una probabilidad estimada  $p$ , es el logaritmo neperiano del ratio de la probabilidad de pertenecer a la clase positiva entre la probabilidad de no pertenecer a esa clase, ese es el valor  $t$  asociado a la probabilidad.

Si un ejemplo tiene una probabilidad muy alta de ser de la clase positiva, por ejemplo 0.8, entonces  $t = 1.38$  que como es mayor de 0, positivo, lo asignará a la clase positiva. De la misma manera, si tiene una probabilidad muy baja de pertenecer a la clase positiva, digamos 0.2, su logit será de -1.38.

Así que podemos usar el signo, o la magnitud de  $p$  (que es la sigmoide de  $t$ ) si solo nos interesa la clase, o bien el valor de  $p$  si nos interesa la probabilidad.



$$\hat{y} = \begin{cases} 0 & \text{si } p < 0.5 \\ 1 & \text{si } p \geq 0.5 \end{cases}$$

## FUNCIÓN DE COSTE Y ENTRENAMIENTO

El objetivo del entrenamiento del modelo es encontrar un vector de parámetros  $\theta$  que haga que el modelo estime altas probabilidades para ejemplos etiquetados como 1 (de la clase positiva  $y=1$ ) y bajas probabilidades para ejemplos etiquetados como de la clase negativa ( $y=0$ ). Para una única instancia esta idea la recoge esta función de coste:

$$c(\theta) = \begin{cases} -\log(\hat{p}) & \text{if } y = 1 \\ -\log(1 - \hat{p}) & \text{if } y = 0 \end{cases}$$

Como  $-\log(t)$  crece cuando  $t$  se

acerca a 0 ( $\log(t)$  es el logaritmo neperiano de  $t$ ), crece mucho cuando el modelo estima una probabilidad cercana a 0 para un ejemplo positivo  $y$  también muy alto si el modelo estima una probabilidad cercana a 1 para un ejemplo de la clase negativa.

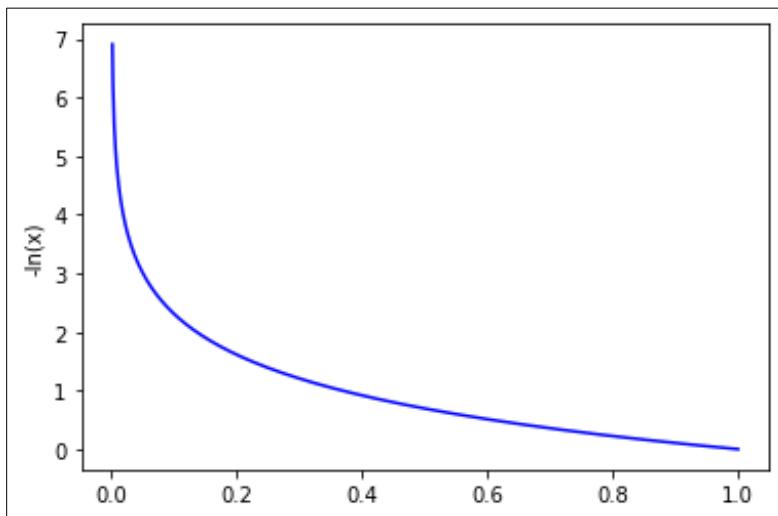


Figura 63: Función  $-\ln(t)$ .

Por otro lado,  $-\log(t)$  está cerca de 0 cuando  $t$  se acerca a 1, así que el coste se acerca a 0 si el modelo estima una probabilidad cercana a 0 para un ejemplo con etiqueta 0, y también un coste cercano a 0 si estima una probabilidad cercana a 1 para un ejemplo con etiqueta 1, que es precisamente lo que debe hacer la función de coste. La expresión del coste medio de todos los  $n$  ejemplos  $x^{(i)}$  es:

$$J(\theta) = \frac{-1}{n} \sum_{i=1}^n [y^{(i)} \ln(\hat{p}^{(i)}) + (1 - y^{(i)}) \ln(1 - \hat{p}^{(i)})]$$

El entrenamiento lo que hace es buscar los parámetros que minimizan esta función. Y hay una mala noticia y otra buena. La mal es que no hay ninguna solución analítica de la misma. La buena es que es que es convexa y por tanto se puede utilizar el algoritmo de descenso por gradiente con garantías de encontrar un mínimo global o una aproximación óptima si eliges un learning rate bajo y das los pasos suficientes. La derivada parcial del  $j$ -ésimo parámetro  $\theta_j$  es:

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{n} \sum_{i=1}^n (\sigma(\theta^T x^{(i)}) - y^{(i)}) x_j^{(i)}$$

Una vez que calculas todas las derivadas parciales de todos los parámetros, puedes usar ese vector como un gradiente den el descenso por gradiente en cualquiera de sus variantes.

**EJEMPLO 29:** Vamos a usar el dataset iris que contiene 150 ejemplos de flores con la longitud y anchura de pétalos que pertenecen a 3 especies distintas: Iris-Setosa, Iris-Versicolor e Iris-Virginica para construir un modelo de regresión logística usando scikit-learn.



Figura 64: Flores iris de 3 especies.

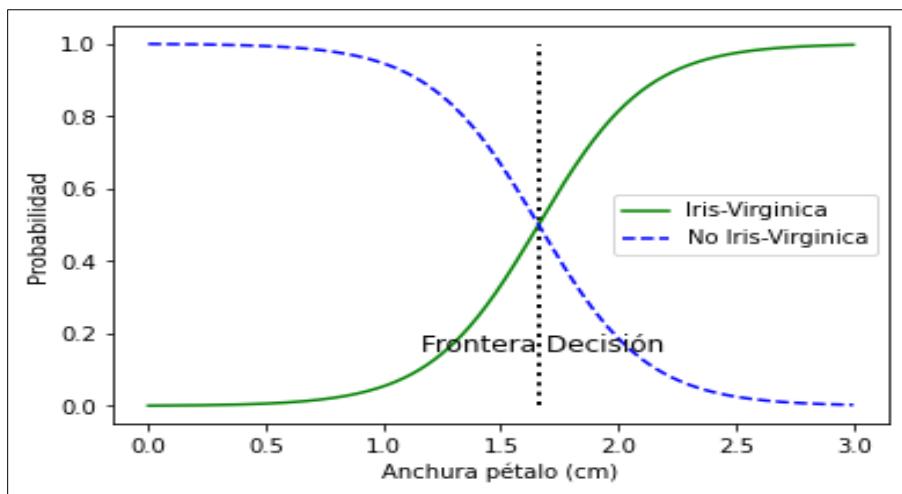


Figura 65: Probabilidades estimadas y frontera de decisión.

Vamos a entrenar un clasificador capaz de detectar flores de tipo *Iris-Virginica* (clase positiva) o no (clase negativa) basándonos solamente en la anchura de los pétalos.

```
from sklearn import datasets
import numpy as np
import matplotlib.pyplot as plt
# preparar datos
iris = datasets.load_iris()
list(iris.keys()) # ['data', 'target', 'target_names', 'DESCR', 'feature_names', 'filename']
X = iris["data"][:, 3:] # anchura de pétalos
y = (iris["target"] == 2).astype(int) # 1 si Iris-Virginica, sino 0
# Entrenar el modelo
from sklearn.linear_model import LogisticRegression
rlog = LogisticRegression()
rlog.fit(X, y)
# Ver probabilidades estimadas según ancho de pétalo
X_new = np.linspace(0, 3, 1000).reshape(-1, 1)
y_prob = rlog.predict_proba(X_new)
frontera = X_new[y_prob[:, 1] >= 0.5][0]
plt.plot([frontera, frontera], [0, 1], "k:", linewidth=2)
plt.text(frontera + 0.02, 0.15, "Frontera Decisión", fontsize=12, color="k", ha="center")
plt.plot(X_new, y_prob[:, 1], "g-", label="Iris-Virginica")
plt.plot(X_new, y_prob[:, 0], "b--", label="No Iris-Virginica")
plt.legend()
plt.xlabel("Anchura pétalo (cm)")
plt.ylabel("Probabilidad")
plt.plot()
print(f"Frontera de decisión: {frontera}")
ejemplos = [[1.7], [1.5]]
```

```
print(f"Predecir {ejemplos} = {rlog.predict(ejemplos)}")
print(f"Predecir con probabilidades {ejemplos} = {rlog.predict_proba(ejemplos)[1]}")
```

## 7.2.1. SOLVERS DE SCIKIT-LEARN.

Uno de los parámetros que tiene la clase `LinearRegression()` es `solver` que puede tener uno de estos posibles valores: `['lbfsgs', 'liblinear', 'newton-cg', 'newton-cholesky', 'sag', 'saga']` y por defecto en la versión 2.4.0 de scikit-learn es `'lbfsgs'`. Vamos a explicar que es un solver y algunos de ellos (si tienes más interés consulta la documentación).

El objetivo es encontrar la tangente de la función en un punto (su pendiente), podemos hacerlo mediante aproximaciones numéricas. Aquí hay dos métodos genéricos:

- **Aproximación lineal:** Si tenemos la función  $f(x)$  encontramos la ecuación de la línea tangente  $L(x) = f(a) + f'(a)(x-a)$  Por tanto, cerca de  $a$  podemos usar esta línea para aproximar la función:  $f(a) = L(x) - f'(a)(x-a)$
- **Aproximación cuadrática:** Usamos una parábola para aproximar la función. Tanto la función como la parábola deberían tener el mismo valor en su primera y segunda derivada así que:  $Q_a(x) = f(a) + f'(a)(x-a) + f''(a)(x-a)^2/2$

Hemos comentado que para encontrar los parámetros debe minimizar la función de coste y lo hace calculando derivadas parciales. Bueno, pues el solver es el la parte del modelo que calcula estas derivadas parciales, porque hay varios métodos, cada uno con sus ventajas e inconvenientes que explicamos a continuación:

- **'newton-cg' Método del gradiente cuadrático de Newton:** Utiliza una versión mejorada de la función cuadrática porque utiliza la primera y la segunda derivada, es decir utiliza la matriz Hessiana (una matriz  $p \times p$  con las segundas derivadas parciales). La ventaja es que en un solo paso encuentra la solución. La desventaja es que calcular la Hessiana es computacionalmente costoso y que si la función de coste tiene zonas suaves puedes acabar en un mínimo/máximo local.
- **'lbfsgs' es el algoritmo Limited-memory Bryden-Fletcher-Goldfarb-Shanno:** Es una variante del método anterior donde la *Hessiana* no es calculada sino aproximada usando los valores calculados por el gradiente, realizando una estimación de la inversa de la matriz *Hessiana*. Las palabras *Limited-memory* indican que solo guarda unos cuantos vectores del gradiente para realizar la aproximación. Este algoritmo utiliza otro algoritmo llamado **Coordinate Descent (CD)** que soluciona el problema de la optimización realizando sucesivas aproximaciones minimizando a lo largo de las coordenadas de las direcciones de los hiperplanos. Es menos costoso que el método de Newton, pero además de tener los mismos problemas es menos efectivo.
- **'Liblinear'**: aplica regularización de tipo  **$\ell_1$**  y es aconsejable usarlo cuando tenemos datos con altas dimensiones (muchas características). Como desventajas tiene que podría quedar atrapado en puntos no óptimos si la función de coste tiene curvas de nivel no suaves (funciones muy poco convexas). Otra desventaja es que no puede aprender realmente un problema multivariado de clasificación, y por tanto debe usar una aproximación **'one-vs-rest'** (para cada diferente clase del target, crear un modelo de clasificación binario).
- **'sag' Stochastic Average Gradient:** optimiza la suma de un número finito de funciones convexas suaves. El coste computacional de cada iteración es independiente de la cantidad de términos de la suma. Es más rápido que otros solver para grandes datasets en el que la cantidad de ejemplos y de predictoras son elevados. Sus inconvenientes son que solo soporta regularización  **$\ell_2$**  y que su consumo de memoria es  **$O(n)$** .

- ‘**saga**’: Es una variante de SAG que soporta regularización  $\ell_1$ . Es la opción ideal si vas a realizar regresión multivariada con datasets con datos dispersos (**sparse**).

Tabla resumen:

Penalizaciones	liblineal	lbfgs	newton-cg	sag	saga
Multinomial +L2	no	si	si	si	si
OVR + L2	si	si	si	si	si
Multinomial + L1	no	no	no	no	si
OVR+L1	no	no	no	no	si
Elastic-Net	no	no	no	no	si
Sin penalización	no	si	si	si	si
<b>Comportamiento</b>					
Penaliza el punto de corte (malo)	si	no	no	no	no
Rápido muchos datos	no	no	no	si	si
Robusto con datos no escalados	si	si	si	no	no

**EJEMPLO 30:** Ahora creamos un nuevo modelo de regresión logística usando de nuevo el dataset iris pero usando de predictoras tanto el ancho de los pétalos como su longitud. Como tenemos dos características predictoras, la frontera de decisión separa superficies. Además haremos líneas discontinuas para la probabilidad 90% y 15% de pertenecer a Iris-Virginica que será la clase positiva.

```
from sklearn.linear_model import LogisticRegression
from sklearn import datasets
import numpy as np
import matplotlib.pyplot as plt
# preparar los datos
iris = datasets.load_iris()
X = iris["data"][:,(2, 3)]          # ancho y largo de pétalos
y = (iris["target"] == 2).astype(int) # clase Iris-Virginica
# Crear y entrenar el modelo
rlog = LogisticRegression(solver="lbfgs", C=10**10, random_state=42)
rlog.fit(X, y)
# Crear una malla bidimensional de datos de anchos y longitudes
x0, x1 = np.meshgrid(
    np.linspace(2.9, 7, 500).reshape(-1, 1),
    np.linspace(0.8, 2.7, 200).reshape(-1, 1),
)
X_new = np.c_[x0.ravel(), x1.ravel()]
y_proba = rlog.predict_proba(X_new)      # Obtener probabilidades de los datos
# Crear el gráfico
plt.figure(figsize=(10, 4))
plt.plot(X[y==0, 0], X[y==0, 1], "bs")
plt.plot(X[y==1, 0], X[y==1, 1], "g^")
zz = y_proba[:, 1].reshape(x0.shape)
contorno = plt.contour(x0, x1, zz, cmap=plt.cm.brg)
izq_der = np.array([2.9, 7])
frontera = -(rlog.coef_[0][0] * izq_der + rlog.intercept_[0]) / rlog.coef_[0][1]
plt.clabel(contorno, inline=1, fontsize=12)
plt.plot(izq_der, frontera, "k--", linewidth=3)
plt.text(3.5, 1.5, "No Iris-virginica", fontsize=12, color="b", ha="center")
plt.text(6.5, 2.3, "Iris virginica", fontsize=12, color="g", ha="center")
plt.xlabel("Longitud Pétalos", fontsize=12)
plt.ylabel("Ancho Pétalos", fontsize=12)
plt.axis([2.9, 7, 0.8, 2.7])
plt.show()
```

Igual que el resto de modelos lineales, los de regresión logística pueden regularizarse usando penalizaciones  $\ell_1$  o  $\ell_2$ . De hecho, por defecto se aplica  $\ell_2$ . El hiperparámetro que controla la fuerza de

la regularización no es  $\alpha$  sino su inversa  $C$ . Un valor alto de  $C$  indica poca regularización. El uso de una u otra puede estar limitada por el solver que se utilice.

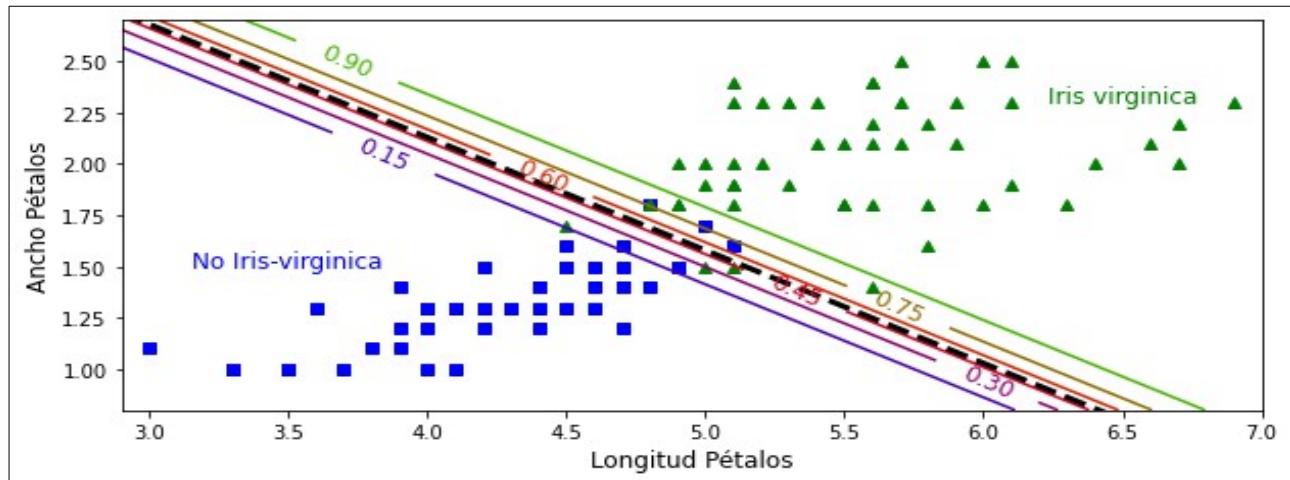


Figura 66: Fronteras de decisión del ejemplo 25.

### 7.3. REGRESIÓN SOFTMAX.

El modelo de regresión logística puede generalizarse para soportar la clasificación directa de un ejemplo en varias posibles clases (más de dos) sin la necesidad de crear un clasificador por cada clase de la variable target. Crear y entrenar varios clasificadores y combinarlos se llama '**one-vs-rest**'. Un modelo que clasifica varias clases usando *softmax* se denomina **Regresión Softmax** o **Regresión Logística Multinomial**.

La función **softmax** es una generalización de la función logística que permite calcular todas las probabilidades de pertenencia de un ejemplo en una regresión logística multiclas. La entrada de *softmax* es un vector numérico que representa los valores predichos para cada clase. Por ejemplo si tenemos un modelo de 3 clases podemos recibir el vector [1, 1, 1]. La función *softmax* convierte estos valores en las probabilidades de cada clase multiplicando la entrada por los pesos del modelo. Imagina que los pesos son [1, 2, 3]. Luego  $z = [1 * 1, 1 * 2, 1 * 3] = [1, 2, 3]$ . Ahora convierte este vector en una distribución de probabilidad, el  $i$ -ésimo elemento del vector describe la probabilidad de que el ejemplo de coste  $z$  pertenezca a la  $i$ -ésima clase. Este valor puede calcularse dividiendo su exponencial entre un término de penalización en el denominador que es la suma de todas las  $M$  funciones lineales:

$$P(y = i | z) = \phi_{\text{softmax}}(z) = \frac{e^z}{\sum_{m=1}^M e^z}$$

Así que la idea es sencilla, si tenemos un modelo que sabe clasificar en 3 clases diferentes ( $k=3$ ) ejemplos con dos predictoras  $[x_1, x_2]$ . Al modelo le llega  $x^{(i)}=[1,1]$  un ejemplo con esos valores de características. Suponiendo que los pesos sean  $\theta=[[1,2], [0.5,0.5], [1,0.5]]$ . Observa que cada clase tiene su propio grupo de parámetros, por tanto como  $k = 3$  hay 3 grupos de parámetros. Debe calcular los  $k$  scores, uno para cada clase con la fórmula:

$$s_k(x) = x^T \theta^{(k)}$$

En el ejemplo:  $s=[s1, s2, s3] = [[1,1] [1,2], [1,1] [0.5,0.5], [1,1] [1, 0.5]]= [3, 1, 1.5]$  y entonces estima la probabilidad de cada clase usando la función *softmax* (también llamada exponencial normalizada).

$$\hat{p}_k = \sigma(s(x))_k = \frac{\exp(s_k(x))}{\sum_{j=1}^K \exp(s_j(x))}$$

$$\text{El denominador sería: } e^3 + e^1 + e^{1.5} = \frac{20,085536923187667740928529654582}{2,7182818284590452353602874713527} + \frac{4,4816890703380648226020554601193}{27,285507821984777798890872586054} =$$

$$\text{probabilidades} = [e^3, e^1, e^{1.5}] / 27,285507821984777798890872586054 = [0.736, 0.026, 0.164]$$

La función *softmax* ha transformado los valores en una distribución de probabilidad (la suma de los valores es 1, salvo por los errores de redondeo).

Si predice el número de clase con la mayor probabilidad solo debe devolver el índice donde está la probabilidad mayor. Si quieras saber las probabilidades (podría ocurrir que todas fuesen bajas) puede devolver el vector de probabilidades.

$$\hat{y} = \operatorname{argmax}_k \sigma(s(\mathbf{x}))_k = \operatorname{argmax}_k s_k(\mathbf{x}) = \operatorname{argmax}_k \left( (\boldsymbol{\theta}^{(k)})^T \mathbf{x} \right)$$

La regresión *softmax* debe utilizarse para clases mutuamente excluyentes (clasificación multiclase pero no multisalida), es decir no podrías usarla para reconocer a varias personas en una fotografía.

**EJEMPLO 31:** Vamos a programar en Python el ejemplo que hemos calculado a mano:

```
# -*- coding: utf-8 -*-
import numpy as np

x = np.array([1, 2])
pesos = np.array([[1,2], [0.5,0.5], [1,0.5]])

def softmax(z):
    return np.exp(z) / np.sum(np.exp(z))

def scores(X, w):
    z = w.dot(X.T)
    return softmax(z)

y_probas = softmax( scores(x, pesos) )
print('Probabilidades:', y_probas)
print('La suma es 1:', y_probas.sum())
print("La clase es:", np.argmax(y_probas))
```

Una vez que conocemos como el modelo estima las probabilidades y hace predicciones, vamos a mirar como configura sus parámetros con el entrenamiento. El objetivo es que el modelo estime probabilidades altas a una de las clases target, la clase correcta, y bajas a las otras clases.

Minimizando la función de coste llamada *entropía cruzada* alcanza la configuración de sus parámetros porque penaliza al modelo cuando estima una probabilidad baja a la clase correcta. La función de *entropía cruzada* se emplea frecuentemente para medir como de bien se ajusta un conjunto de probabilidades estimadas a las clases objetivo.

$$J(\boldsymbol{\Theta}) = -\frac{1}{m} \sum_{i=1}^m \sum_{k=1}^K y_k^{(i)} \log(\hat{p}_k^{(i)})$$

$y_k^{(i)}$  es la probabilidad target de que el  $i$ -ésimo ejemplo pertenezca a la clase  $k$ . En general, será igual a 1 o a 0, según la instancia pertenezca a la clase o no. Cuando solo haya dos clases ( $K = 2$ ), esta función de coste es equivalente a la usada por la función de coste de la *regresión logística*.

## ENTROPÍA CRUZADA

La fórmula de la entropía cruzada proviene de la teoría de la información. Imagina que quieres transmitir información de manera eficiente sobre el tiempo de cada día. Si hay ocho opciones

(soleado, lluvioso, ventoso, etc.), puedes codificar cada opción usando 3 bits ya que  $2^3 = 8$ . Sin embargo, si lo piensas, la mayor parte de los días son soleados, por tanto sería más eficiente codificar 'soleado' con un solo bit '0' y el resto de opciones con 4 que comiencen por 1.

La entropía cruzada mide la media de bits que envías por opción. Si asumes que el clima es perfecto, la entropía cruzada será igual a la entropía del propio clima (es decir, intrínsecamente impredecible). Pero si tus asunciones son falsas (llueve con frecuencia), la entropía cruzada será mayor en una cantidad denominada **divergencia de Kullback-leiber**.

La entropía cruzada entre dos distribuciones de probabilidad  $p$  y  $q$  se define como  $H(p, q) = -\sum_x (p(x) \log(q(x)))$  cuando las distribuciones son discretas. Para más información visita este [video](#).

El vector gradiente de esta función de coste para calcular los parámetros de la clase  $k$  son  $\Theta(k)$ :

$$\nabla_{\Theta^{(k)}} J(\Theta) = \frac{1}{m} \sum_{i=1}^m (\hat{p}_k^{(i)} - y_k^{(i)}) \mathbf{x}^{(i)}$$

**EJEMPLO 32:** Entrenar un modelo para clasificar mediante **softmax** las flores del dataset Iris en las 3 clases posibles y mostrar sus fronteras de decisión.

```
# -*- coding: utf-8 -*-
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.linear_model import LogisticRegression
# preparar los datos
iris = datasets.load_iris()
X = iris["data"][:, (2, 3)]      # Predictoras: longitud y anchura de pétalos
y = iris["target"]
# Entrenar modelo: multinomial usa entropía cruzada y softmax
# El parámetro multi_class='ovr' marcado como deprecated. Usar multiclass.OneVsRestClassifier
softmax_rl=LogisticRegression(multi_class="multinomial",solver="lbfgs",C=10, random_state=42)
softmax_rl.fit(X, y)
# Crear la malla de valores de prueba
x0, x1 = np.meshgrid(
    np.linspace(0, 8, 500).reshape(-1, 1),
    np.linspace(0, 3.5, 200).reshape(-1, 1),
)
X_new = np.c_[x0.ravel(), x1.ravel()]
y_proba = softmax_rl.predict_proba(X_new)
y_predict = softmax_rl.predict(X_new)
zz1 = y_proba[:, 1].reshape(x0.shape)
zz = y_predict.reshape(x0.shape)
# Realizar gráfico de fronteras de separación
plt.figure(figsize=(10, 4))
plt.plot(X[y==2, 0], X[y==2, 1], "g^", label="Iris-virginica")
plt.plot(X[y==1, 0], X[y==1, 1], "bs", label="Iris-versicolor")
plt.plot(X[y==0, 0], X[y==0, 1], "yo", label="Iris-setosa")
from matplotlib.colors import ListedColormap
custom_cmap = ListedColormap(['#fcfbc1', '#999bff', '#a5fcf4'])
plt.contourf(x0, x1, zz1, cmap=custom_cmap)
contour = plt.contour(x0, x1, zz1, cmap=plt.cm.brg)
plt.clabel(contour, inline=1, fontsize=12)
plt.xlabel("Longitud Pétalos", fontsize=12)
plt.ylabel("Anchura de Pétalos", fontsize=12)
plt.legend(loc="center left", fontsize=12)
plt.axis([0, 7, 0, 3.5])
plt.show()
```

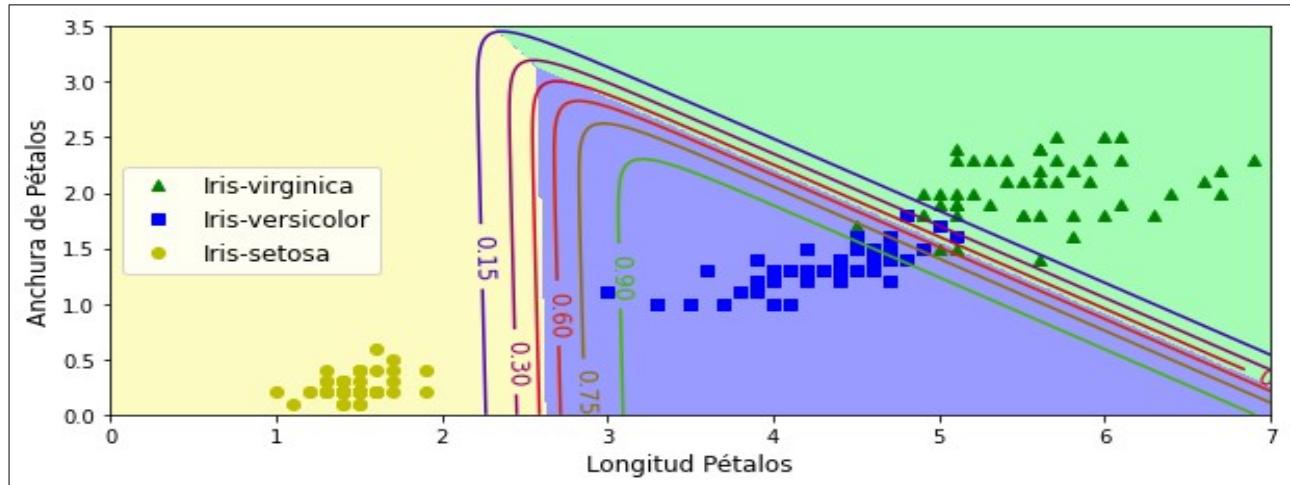


Figura 67. Softmax y fronteras de decisión.

## 8. ALGORITMO K-NEAREST NEIGHBORS (KNN)

Este algoritmo de clasificación es interesante porque es muy diferente de los que hemos visto hasta ahora. **KNN** es un ejemplo típico de un aprendiz perezoso (*lazy learner*). No se llama así por su aparente simplicidad sino porque al entrenarse (estudiar) no crea una función que le permita realizar predicciones (no crea unos apuntes) sino que memoriza los datos de entrenamiento.

### MODELOS PARAMÉTRICOS Y NO PARAMÉTRICOS

Los modelos generados por los algoritmos de ML se pueden clasificar en *modelos paramétricos y no paramétricos*. Cuando usamos modelos paramétricos, estimamos parámetros a partir del dataset de entrenamiento. Aprender estos parámetros de funciones que les permiten realizar una tarea como clasificar de manera que ya no necesitan más los datos de entrenamiento. Ejemplos de modelos paramétricos son:

- El perceptrón.
- La regresión Logística.
- Y las *SVM* lineales (máquinas de soporte vectorial).

Por contra, los *modelos no paramétricos* no pueden definirse mediante cierta cantidad fija de parámetros, porque esta cantidad de parámetros aumenta a medida que aumenta la cantidad de ejemplos a partir de los que se entrena. Ejemplos de este tipo de modelos serían los *árboles de decisión para clasificación*, *random forest* y las *SVM con kernels*.

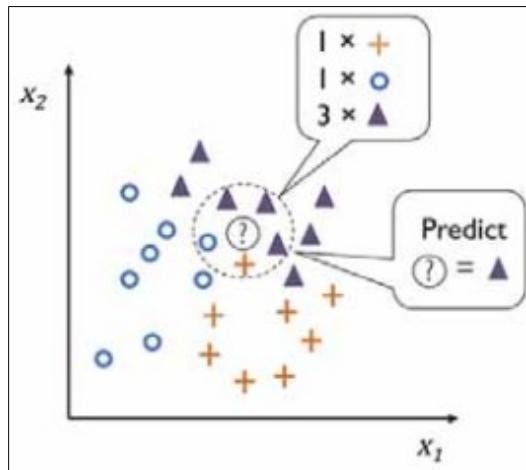
El algoritmo **KNN** pertenece a una subcategoría de modelos no paramétricos denominada aprendizaje basado en instancias. Estos modelos memorizan el dataset de entrenamiento para no tener que procesarlo una y otra vez de manera completa. Los algoritmos de aprendizaje perezoso son otra nueva subcategoría de los basados en instancias, que se asocian a no tener costes durante el proceso de aprendizaje.

El algoritmo **KNN** puede resumirse en estos 3 pasos:

1. Escoger el número **K** y una métrica de distancia.
2. Encontrar los **K-vecinos** más cercanos a la muestra que queremos clasificar.
3. Asignar al ejemplo la etiqueta de clase mayoritaria entre los vecinos.

La siguiente figura ilustra como la información que aporta un nuevo punto de datos (identificado por el símbolo (?)) puede incorporarse a la información previa sin necesidad de volver a procesar todos los datos de entrenamiento. El punto se clasifica como perteneciente a la clase triángulo porque de los 5 vecinos más próximos a él, la mayoría son triángulos.

Según la métrica de distancia que se escoja, **KNN** busca los **K** vecinos más cercanos al nuevo dato que queremos clasificar.



*Figura 68. Clasificación del ejemplo (?) por votación de sus vecinos.*

La ventaja de este modelo es que en realidad no necesita entrenamiento, el entrenamiento consiste realmente en memorizar el dataset y añadir nuevos datos al dataset supone adaptarse inmediatamente. La desventaja es el consumo de memoria RAM y el tiempo en responder que irá aumentando a medida que aumente la cantidad de ejemplos que tenga. Hay implementaciones de árboles **KD-Trees** que buscan matchear en órdenes de tiempo logarítmicos.

**EJEMPLO 33:** Entrenar un modelo **K-NN** para clasificar una flor del dataset iris en una de sus clases y dibujar las fronteras de decisión.

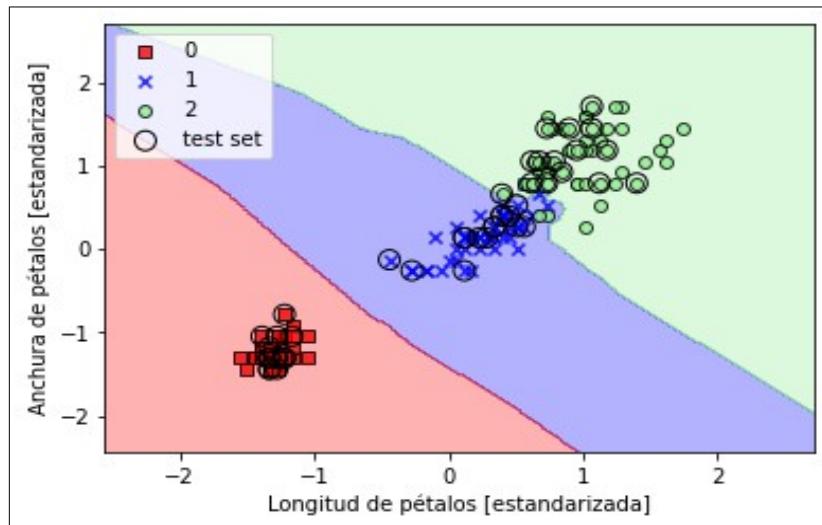
```
# -*- coding: utf-8 -*-
import numpy as np
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

def plot_regiones(X, y, clasificador, test_idx=None, resolucion=0.02):
    # Configurar marcas y colores
    marcadores = ('s', 'x', 'o', '^', 'v')
    colores = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colores[:len(np.unique(y))])
    # Dibujar superficies de decisión
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolucion),
                           np.arange(x2_min, x2_max, resolucion))
    Z = clasificador.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1], alpha=0.8, color=colores[idx],
                    marker=marcadores[idx], label=cl, edgecolor='black')
    # Resaltar los ejemplos de test
    if test_idx:
        X_test, y_test = X[test_idx, :], y[test_idx]
        plt.scatter(X_test[:, 0], X_test[:, 1], c='none', edgecolor='black',
                    alpha=1.0, linewidth=1, marker='o', s=100, label='test set')
    # Preparar los datos
    iris = datasets.load_iris()
```

```

X = iris["data"][:, (2, 3)]      # Predictoras: longitud y anchura de pétalos
y = iris["target"]                # o tb. iris.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=1,
stratify=y)
sc = StandardScaler()
sc.fit(X_train)
X_train_std = sc.transform(X_train)
X_test_std = sc.transform(X_test)
# Entrenar modelo KNN con 5 vecinos
knn = KNeighborsClassifier(n_neighbors=5, p=2, metric='minkowski')
knn.fit(X_train_std, y_train)
X_combinado_std = np.vstack((X_train_std, X_test_std))
y_combinado = np.hstack((y_train, y_test))
plot_regiones(X_combinado_std, y_combinado, clasificador=knn, test_idx=range(105,150))
plt.xlabel('Longitud de pétalos [estandarizada]')
plt.ylabel('Anchura de pétalos [estandarizada]')
plt.legend(loc='upper left')
plt.show()

```



*Figura 69. Clasificación del ejemplo (?) por votación de sus vecinos.*

En caso de empate, la implementación de **K-NN** de **scikit-learn** se decanta por etiquetar el ejemplo como el vecino más cercano. Si todos los vecinos tienen distancias muy similares, el algoritmo escoge de las posibles, la clase que predomine en el dataset.

Escoger un valor acertado de K es fundamental para obtener un buen balance entre overfitting y underfitting. También es importante elegir bien la métrica utilizada para medir distancias, que sea apropiada para las características de los datos. Con frecuencia, una simple distancia Euclídea es apropiada para datos numéricos. Sin embargo, si usamos características con distintos rangos de valores **es necesario estandarizar los valores** para que cada característica aporte la misma cantidad a la distancia. La distancia de Minkowski usada en el código del ejemplo 28 es solamente una generalización de las distancias Euclídea y Manhattan, que puede escribirse como:

$$d(x^{(i)}, x^{(j)}) = \sqrt[p]{\sum_k |x_k^{(i)} - x_k^{(j)}|^p}$$

que es la distancia Euclídea si  $p=2$  o la distancia Manhattan cuando  $p=1$ . Hay otras muchas distancias que puedes consultar en la documentación de **scikit-learn**.

En modelos donde la regularización no es aplicable (como **K-NN** y **CARTS**) podemos usar **selección de características y reducción de dimensionalidad** para evitar el overfitting.

## PIPELINES DE OPERACIONES

En vez de hacer cada cosa por separado (adaptar datos, entrenar, ..., testear) podemos encadenar todo en un solo objeto pipeline añadiéndole por ejemplo un objeto **StandardScaler**, otro objeto **PCA** y otro objeto **LogisticRegression** de forma que se comporte como un nuevo objeto.

**EJEMPLO 34:** Crear un pipeline para preparar y entrenar un modelo de regresión logística.

```
# -*- coding: utf-8 -*-
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline
X_train = np.array([[1, 2, 3], [2, 4, 6], [2, 0, 2], [1, 3, 3], [1, 5, 7]] )
y_train = np.array([0, 1, 1, 0, 0]) # Son todos pares
X_test = np.array([[4, 2, 6], [3, 4, 2]])
y_test = np.array([1, 0])
pipe_rl = make_pipeline(StandardScaler(),
                        PCA(n_components=2),
                        LogisticRegression(random_state=1))
pipe_rl.fit(X_train, y_train)
y_pred = pipe_rl.predict(X_test)
print('Test Accuracy: %.3f' % pipe_rl.score(X_test, y_test))
```

La función **make\_pipeline()** acepta un número arbitrario de objetos transformadores de *scikit-learn* (los transformadores soportan los métodos **fit()** y **transform()**), seguidos por un estimador que implementa los métodos **fit()** y **predict()**. En el ejemplo 29 pasamos los transformadores **StandardScaler** y **PCA**, y el estimador **LogisticRegression**. Un pipeline de *scikit-learn* es por tanto como un meta-estimador o envoltura de transformadores y estimadores. Al llamar a su método **fit()**, los datos irán sufriendo una cadena de transformaciones aplicadas por los **fit()** de los objetos transformadores, hasta que lleguen a un objeto estimador que será el que les aplique la última llamada.

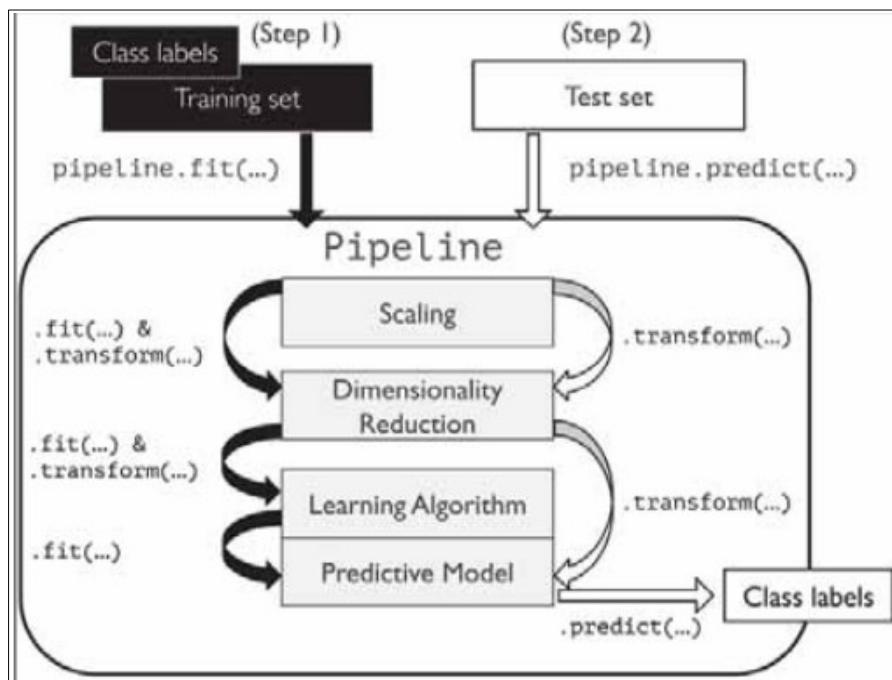
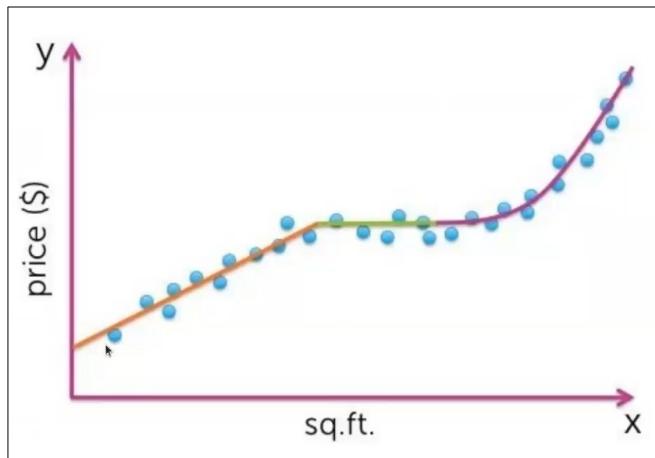


Figura 70. Representación de un pipeline de senencias en Scikit-learn.

### USAR K-NN COMO MODELO DE REGRESIÓN

No es un algoritmo paramétrico, de ahí que sea tan efectivo en modelos de regresión. En el gráfico del ejemplo no nos vale un modelo lineal, porque tenemos una parte que es no lineal. Tampoco resulta útil un modelo cuadrático, porque tenemos una parte lineal. Incluso nos costará encontrar un buen

modelo polinomial. Podríamos ir ajustándolo y haciéndolo más y más complejo para luego regularizarlo. O, podemos usar un modelo que se base en los datos que tiene. Un **K-NN** para regresión funciona mirando las **k** instancias más cercanas y calculando una media de sus valores.



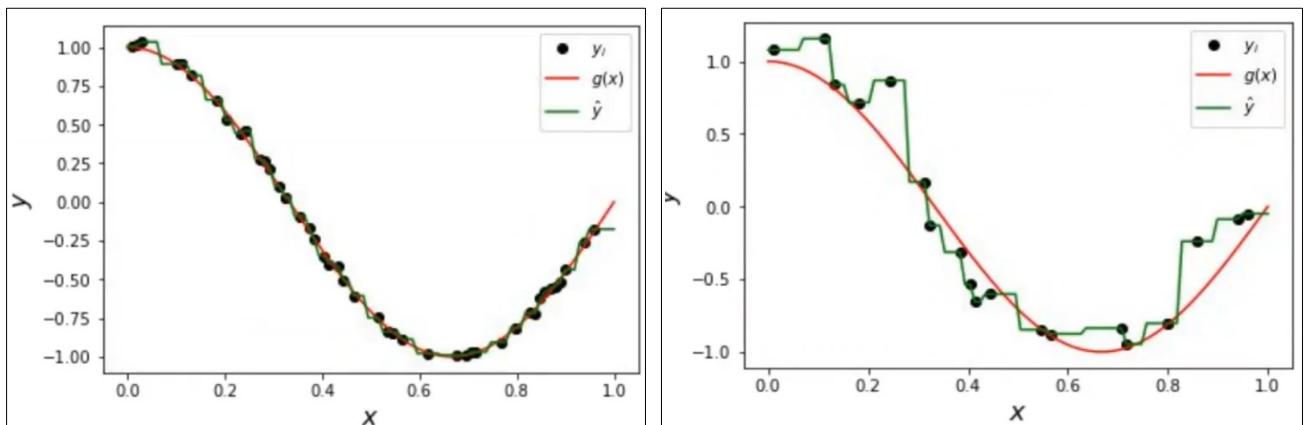
La media cambia en función de cuántas instancias cercanas escojamos (según el valor de k). Por ejemplo si queremos estimar el precio (y) a partir de los metros cuadrados (x) en un nuevo punto  $x_0$ , un K-NN busca las K ejemplos vecinos (más cercanos) de  $x_0$  calculando sus distancias:

$$d_i = ||x_0 - x_i||^2$$

El valor estimado es la media de los vecinos:

$$\hat{y}_0 = \frac{1}{K} \sum_{i=1}^K y_i$$

Entonces, si se define un  $K=1$ , miramos solo el vecino más cercano y vamos a tener un buen ajuste si hay mucha densidad de puntos y bajo ruido. Este va a ser un modelo muy complejo y van a estar muy ajustados los datos, pero puede que en ocasiones nos interese. Es como el gráfico de la izquierda. Si no tenemos suficientes datos, seguramente un valor tan bajo de K genera *overfitting* y va a generalizar mal. Además tendremos regiones sin ejemplos, mal ajustadas y muy sensibles al ruido. Por lo tanto, si no hemos eliminado *outliers* correctamente y tenemos de repente un punto mal puesto, ese punto va a impactar mucho.



¿Cuál es la solución a esto? Debemos aumentar el K y podemos ponderar la estimación de distancia entre los vecinos. Esto implica:

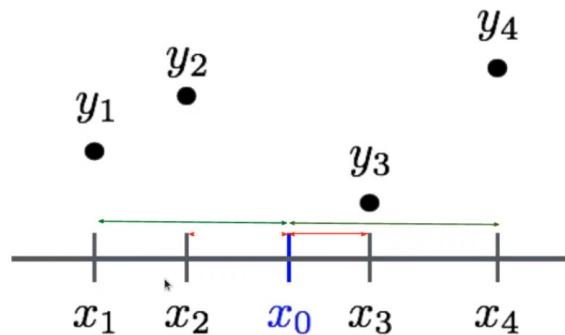
- Dar más peso a los más cercanos.
- Reducir el peso de los más alejados.

Veamos cómo ponderar la estimación  $K = 4$  ¿Cómo elegimos  $\theta$ ? En función de la distancia:

$$\theta_i = 1 / d_i$$

donde:

$$d_i = ||x_0 - x_i||^2$$



$$y_0 = \frac{\theta_1 y_1 + \theta_2 y_2 + \theta_3 y_3 + \theta_4 y_4}{\sum_{i=1}^K \theta_i}$$

Podemos utilizar otras opciones, otras medidas de similitud:

$$e^{-\frac{||x_0 - x_i||_2^2}{2\sigma^2}}$$

En *Scikit-learn* dispone de [KNeighborsRegressor](#)

## 9. EVALUAR MODELOS PARA CLASIFICACIÓN.

Ahora que ya sabemos definir y entrenar algunos clasificadores, podríamos haber implementado uno para predecir el comportamiento de futuros clientes usando datos diferentes a los usados para entrenar el modelo. Te pueden surgir algunas preguntas: ¿Cómo de bien clasifica mi modelo? Y si implementas varios, ¿Cuál de ellos será el mejor? ¿Cómo medir lo bien o mal que funcionan?

La exactitud (**accuracy**) de un clasificador simplemente es el porcentaje de aciertos que tiene cuando realiza su trabajo de clasificación. Si tienes 100 monedas y 10 de ellas son falsas, si un clasificador clasifica bien 89 de las legales y clasifica como falsas a 2 de las falsas, el clasificador tendrá una exactitud de:

$$\text{exactitud} = \text{accuracy} = \frac{\text{aciertos}}{\text{aciertos} + \text{fallos}} = \frac{89 + 2}{100} = \frac{91}{100} = 0,91 = 91\%$$

Con esta cifra parece que es un buen clasificador. Sin embargo, como muchas veces ocurre en la vida real, la cantidad de monedas falsas es muy inferior a la cantidad de monedas legales. El dataset no está balanceado, la proporción de cada clase es muy diferente. Ocurre lo mismo con los e-mail que son spam y los que no, los pacientes con cáncer y los que no, etc. En realidad el clasificador no es bueno, porque aunque lo hace bien con la clase mayoritaria, con la otra clase lo hace bastante mal:

$$\text{accuracy con monedas legales} = 89 / 90 = 98,8\%$$

$$\text{accuracy con monedas falsas} = 2 / 10 = 20\%$$

Si utilizamos solamente el *accuracy* podemos hacer pasar bueno un clasificador mediocre, al meter en la misma bolsa todos los casos, sin entrar en detalle de como es cada clase. En el ejemplo detectaría bien las monedas legales, pero cometería muchos fallos con las falsas. Necesitamos una herramienta que el *accuracy* para medir el desempeño de un clasificador. Esta mejora es **la matriz de confusión**.

## 9.1 LA MATRIZ DE CONFUSIÓN Y MÉTRICAS.

Un modelo de clasificación predice valores que representan la clase del *target* a la que pertenece el ejemplo que se le proporciona. El target puede tener dos posibles valores en la clasificación binaria (que se denominan positivo y negativo) o más de dos en una clasificación multiclase. Si estamos en una clasificación binaria el clasificador puede predecir que un ejemplo es de clase 1 y ser cierto o fallar. Pero ese valor 1 no es en realidad un valor numérico que se pueda utilizar para medir cuánto error ha cometido el modelo. Para medir el comportamiento de los modelos de clasificación no sirven las métricas usadas en los modelos de regresión numérica.

Contar la proporción de aciertos al realizar predicciones (la *exactitud* o *accuracy*) es la métrica más usada, pero es insuficiente. La matriz de confusión es una tabla para medir el rendimiento de un modelo de clasificación. En el caso de la clasificación binaria contiene 4 valores numéricos donde se cuentan las veces que el modelo ha predicho que:

- El ejemplo es de clase positiva y acierta (True Positivo=TP).
- El ejemplo es de clase positiva y falla (Falso positivo=FP).
- El ejemplo es de clase negativa y acierta (True Negativo= TN).
- El ejemplo es de clase negativa y falla (Falso Negativo = FN).

		Predicted Class		Sensitivity $\frac{TP}{(TP + FN)}$	Specificity $\frac{TN}{(TN + FP)}$	Accuracy $\frac{TP + TN}{(TP + FP + FN + TN)}$			
		Positive							
Actual Class	Positive	True Positive (TP)	False Negative (FN) Type II Error						
	Negative	False Positive (FP) Type I Error	True Negative (TN)						
		Precision $\frac{TP}{(TP + FP)}$	Negative Predictive Value $\frac{TN}{(TN + FN)}$						

Figura 71. Matriz de confusión.

Los aciertos son **TP** y **TN** y los fallos son **FP** y **FN**. Idealmente un modelo debería tener un alto valor de **TN** y **TP** y valores pequeños de **FN** y **FP**.

La matriz de confusión **MC** cuando hay **k** clases (**k ≥ 2**) es una matriz cuadrada de tamaño **k × k**. La entrada **MC<sub>i,j</sub>** en la *i*-ésima fila y la *j*-ésima columna de la tabla indica el número de ejemplos de la clase **i** que han sido clasificados por el modelo como de la clase **j**.

Para que el clasificador tenga una buena exactitud, la mayoría de ejemplos se tienen que contar en la diagonal principal, porque esta diagonal cuenta los ejemplos que han sido correctamente clasificados y el resto a cero (en ese caso **FP** y **FN** serán 0).

Por ejemplo en un clasificador de clientes podríamos tener la clase **compra\_ordenador** como clase positiva y **no\_compra\_ordenador** como clase negativa. El clasificador predice **predice\_p** ejemplos como positivos y **predice\_N** ejemplos como negativos. Para cada ejemplo, como tenemos la clase a la que pertenece anotada en su columna *label* o *target*, podemos comprobar si ha acertado.

Imagina que la matriz de confusión del modelo es esta:

Clases	compra_ordenador	no_compra_ordenador	total	% acierto
compra_ordenador	6954	46	7000	99,34
no_compra_ordenador	412	2588	3000	86,27
Total	7366	2634	10000	95,42

A partir de estos contadores se definen estas métricas:

### EXACTITUD (ACCURACY)

La primera métrica que podemos obtener se llama **exactitud** o **accuracy** y es la ratio de ejemplos o instancias del dataset que el modelo clasifica de manera correcta entre el total de instancias clasificadas. Su fórmula:

$$\text{accuracy} = \frac{TP + TN}{P + N}$$

La exactitud es una buena métrica si los datos están balanceados (misma cantidad de unas clases que de otras).

También podemos calcular el ratio de error o el ratio de fallos que comete el modelo con alguna de estas dos fórmulas:  $M = 1 - \text{accuracy}$  o con la fórmula:

$$\text{ratio error} = 1 - \text{accuracy} \quad \text{ratio error} = \frac{FP + FN}{P + N}$$

Si estamos usando los datos de entrenamiento (en vez de los de test) este valor se conoce como el error de resustitución. Este error es una estimación optimista del error con los datos de test (pues esos datos son ajenos al modelo).

La **eficiencia balanceada** (*balanced accuracy* o abreviada *BACC*) se usa para medir el desempeño de datos no balanceados:

$$BACC = \frac{1}{2}(\text{sensitividad} + \text{recall})$$

### SENSITIVIDAD (RECALL) Y ESPECIFICIDAD (RECALL NEGATIVO)

Mide la habilidad del modelo de detectar un positivo cuando lo ve. La **sensitividad** es el ratio de verdaderos positivos (los positivos que acierta) entre la cantidad de positivos que hay. Si es alta, significa que se le escapan pocos positivos, es decir, los *FN* son pocos.

$$\text{sensitividad} = \text{recall} = \frac{TP}{P} = \frac{TP}{TP + FN}$$

La **especificidad** es el equivalente para la clase negativa, es decir, la habilidad del clasificador de reconocer un negativo. Es el ratio de verdaderos negativos correctamente identificados entre todas las instancias que son negativas.

$$\text{especificidad} = \frac{TN}{N} = \frac{TN}{TN + FP}$$

También podemos relacionar estas tres métricas entre sí:

$$\text{accuracy} = \text{sensitividad} * \left( \frac{P}{P + N} \right) + \text{especificidad} * \left( \frac{N}{P + N} \right)$$

**EJEMPLO 35:** Calcular las métricas con los datos de la matriz de confusión de la clasificación de clientes.

$$\begin{aligned}\text{Sensitividad} &= 90 / 300 = 30\% \\ \text{Especificidad} &= 9560 / 9700 = 98.56\% \\ \text{Eficiencia} &= 96.50\%.\end{aligned}$$

Las conclusiones serían que es eficiente clasificando instancias negativas, pero poco eficiente clasificando las instancias positivas. Aunque su eficiencia global es del 96.5% no es un buen clasificador. La forma de mejorar el modelo entrenado con datos no balanceados las explicaremos más adelante.

### PRECISIÓN POSITIVA Y PRECISIÓN NEGATIVA

Mide la fiabilidad a la hora de detectar positivos. La **precisión** es el ratio de aciertos con los verdaderos positivos. Es decir, cuando dice que una instancia es positiva, ¿Cuánto nos lo debemos creer? So los verdaderos positivos entre todas las que deben serlo. Si es alta, significa que los falsos positivos son bajos:

$$\text{precisión} = \frac{TP}{TP+FP}$$

Una precisión de 1 para la clase C significa que cada instancia que el clasificador asocia a C es de la clase C. Sin embargo no te dice nada sobre la cantidad de instancias que ha clasificado como C y no lo eran.

De igual forma, tendríamos una precisión para la clase negativa como:

$$\text{precisión} = \frac{TN}{TN+FN}$$

### LA SENSITIVIDAD (RECALL) Y LA PRECISIÓN SON INCOMPATIBLES

Si prestamos atención a las fórmulas de la sensitividad (recall) y la precisión de un clasificador, la diferencia es que:

precisión sube si bajan los FP

$$\text{precisión} = \frac{TP}{TP+FP}$$

recall sube si bajan los FN

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

Cuando entrenamos un modelo, debemos fijarnos en la métrica que preferimos usar. Si estamos interesados en tener pocos falsos positivos o pocos falsos negativos elegiremos una u otra métrica.

Imagina que estamos realizando un clasificador binario que distinga dígitos 5 del resto de dígitos. Para cada instancia se calcula una puntuación basada en una función de decisión y si esa puntuación es mayor que un umbral asigna la instancia a la clase positiva o bien la asigna a la clase negativa desde la puntuación más baja a la izquierda hasta la puntuación más alta a la derecha. Supongamos que el umbral de decisión esté situado en la flecha central de la figura (entre los dos 5): encontrarás 4 TP (en realidad 5) a la derecha de ese umbral y un FP (en realidad 6).

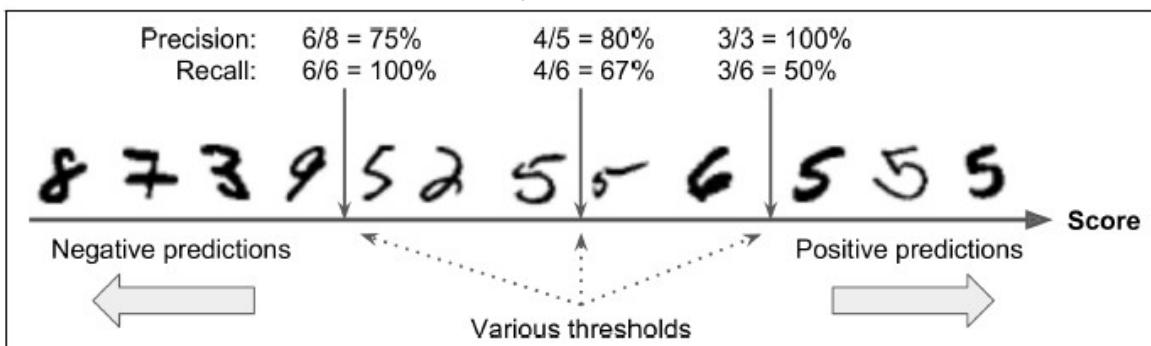


Figura 72: Al cambiar el umbral de decisión cambias precisión (recall) y FN ( ).

Por tanto, con ese umbral, la precisión es del 80% (4 sobre 5). Pero el clasificador, de 6 dígitos 5 actuales solo detecta 4, por lo que el *recall* es del 67% (4 de 6). Si subes el umbral (mueves la flecha de la derecha), el *FP* (el 6) se convierte en un *TN*, aumentando así la precisión (hasta el 100% en este caso), pero al hacerlo también conviertes un *TP* en un *FN*, lo que reduce el *recall* hasta un 50%. En cambio, si reduces el umbral (lo mueves a la izquierda) aumentas el *recall* y bajas la precisión. Es decir, **si aumentas el *recall* (bajas los *FP*) pero bajas la precisión (suben los *FN*) y viceversa. Por tanto ambos comportamientos suelen ser contrarios.**

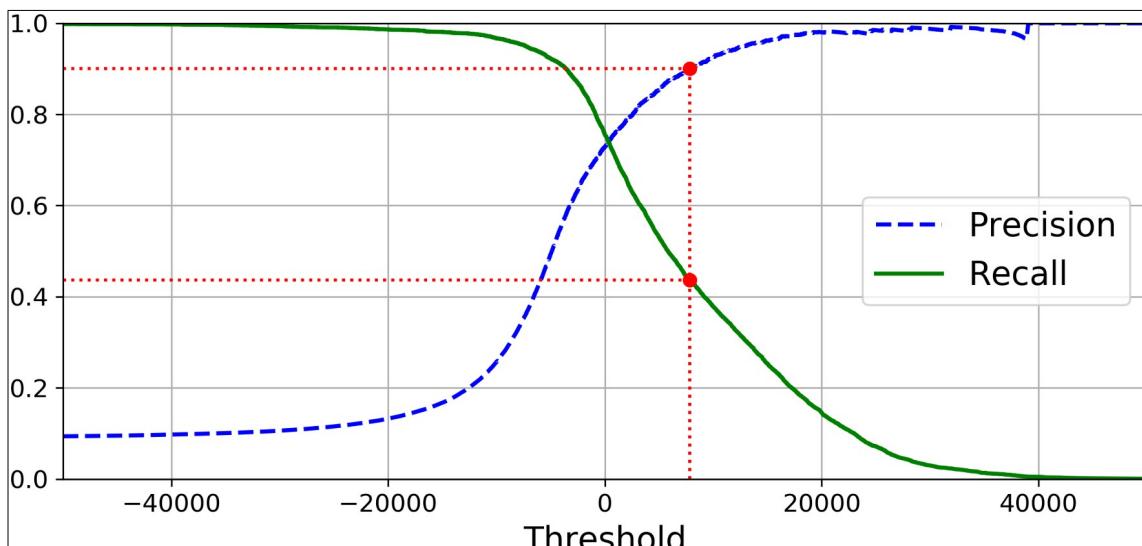


Figura 73: Curva precisión vs recall.

Por ejemplo un clasificador médico puede conseguir una alta precisión clasificando ejemplos de cáncer pero entonces tendrán un bajo *recall*. Los valores de *precisión* y *recall* se suelen utilizar juntos, siendo necesario priorizar uno u otro según el contexto en que se utilice el clasificador o ambos (encontrando una medida de equilibrio usando el F1-score). Por ejemplo se puede comparar varias precisiones con un recall de 0.75. Cuando interesa usar como métrica las dos cosas, usamos las métricas **F-SCORE**.

### F-SCORE

Para usar la precisión y el recall de manera combinada en una única medida se utiliza lo que se conoce como el **F-score** (también conocido como **F $\beta$ -score**) que se define así:

$$F_1 = \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

$$F_\beta = \frac{(1+\beta^2) * \text{precision} * \text{recall}}{(\beta^2 * \text{precision}) + \text{recall}}$$

donde  $\beta$  es un número real no negativo. La métrica F1 es la media armónica de la precisión y el *recall*. Da la misma importancia o peso a ambos. La métrica  $F_\beta$  es la media ponderada de la precisión y el *recall*.

$$F_1 = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} = \frac{\text{TP}}{\text{TP} + \frac{\text{FN} + \text{FP}}{2}}$$

**Nota:** Estamos asumiendo que dado un ejemplo, este siempre pertenece a una única clase. Pero a veces esta suposición no es acertada. ¿Cómo medir la eficiencia en estos casos? En vez de devolver un valor para la clase, se devuelve una distribución de probabilidades para todas las clases. Y se utiliza una heurística como juzgar que la clase predicha es la primera o segunda más probable.

**EJEMPLO 36:** Calcular la matriz de confusión del modelo del ejemplo 34.

```
from sklearn import metrics
```

```
# ...Usa por ejemplo las sentencias del modelo del ejemplo 29...
# genera métricas de evaluación
print("Accuracy : ", metrics.accuracy_score(y, model.predict(x)) )
print("AUC:", metrics.roc_auc_score(y, model.predict_proba(x)[:,1]) )
print("Matriz de confusión:",metrics.confusion_matrix(y, model.predict(x)))
print("Informe de la clasificación:", metrics.classification_report(y, model.predict(x)))
# Otro juego de datos preparado manualmente con etiquetas y predicciones
y_pred = [0, 1, 0, 0]
y_true = [0, 1, 0, 1]
print("Accuracy:", metrics.accuracy_score(y_true, y_pred))
# La precisión y el recall
print("Precisión:", metrics.precision_score(y_true, y_pred))
print("Recall:", metrics.recall_score(y_true, y_pred))
# Sobre clases individuales: sensibilidad & Especificidad
recalls = metrics.recall_score(y_true, y_pred, average=None)
print("Recall/especificidad clase 0:", recalls[0])
print("Recall/especificidad clase 1:", recalls[1])
# Accuracy balanceada
print("Accuracy balanceada:", recalls.mean())
# La precisión y el recall de cada clase individual
p, r, f, s = metrics.precision_recall_fscore_support(y_true, y_pred)
print("Precisión:", p)
print("Recall:", r)
print("F:", f)
print("Soporte:", s);
```

## OTRAS MÉTRICAS

Además de métricas basadas en eficacia, los clasificadores también pueden compararse por:

- **Velocidad:** el coste computacional que supone crear, entrenar y utilizar el modelo.
- **Robusted:** la habilidad del modelo de realizar predicciones correctas ante datos ruidosos o datos con valores ausentes. Se mide probando el clasificador con diferentes ejemplos cada más ruidosos y con menos datos.
- **Escalabilidad:** la habilidad de generar el clasificador de manera eficiente usando una gran cantidad de datos. Se mide generando el modelo con datasets cada vez mayores.
- **Interpretabilidad:** se refiere al nivel de comprensión y conocimiento de las decisiones que toma el clasificador. Esto es un poco subjetivo y complicado de medir. Los CART y las reglas de decisión son muy sencillos de interpretar aunque su interpretabilidad puede disminuir si aumenta su complejidad.

Realiza el [ejercicio 3](#) y el [ejercicio 4](#) de la relación de problemas.

## 9.2. CURVAS ROC.

Las métricas  $TP$ ,  $TN$ ,  $FP$  y  $FN$  de la matriz de confusión también pueden utilizarse en la validación de los modelos de clasificación calculando las curvas de costes y beneficios o **curvas ROC (Receiver operating characteristic curves)**. El coste asociado con un falso negativo (como de incorrecto es predecir que un paciente de cáncer no lo tiene) es más perjudicial que un simple falso positivo (clasificar incorrectamente un paciente como que tiene cáncer cuando en realidad no lo tiene). En estos casos, deberíamos dar más importancia a ciertos tipos de error al considerarlos más peligrosos que los otros. Estos costes que podemos considerar (más allá de los resultados matemáticos) como el peligro para un paciente, costes financieros o terapias innecesarias, etc. Hasta ahora, cuando hemos calculado las métricas a partir de la matriz de confusión, hemos considerado que todos los tipos de error tienen el mismo coste asociado.

Otra posibilidad es incorporar costes y beneficios cuando se calculen los costes medios (o los beneficios) a la hora de tomar una decisión. Otras aplicaciones implican análisis de costes y beneficios en la concesión de préstamos, hipotecas, campañas publicitarias y muchos otros tipos de toma de decisiones.

Las *curvas ROC* se usan como una herramienta visual para los modelos de clasificación, principalmente binaria. Provienen de la teoría de detección de señales desarrollada durante la segunda guerra mundial para el análisis de imágenes de radar.

Muy similar a *la curva precisión vs recall*, pero dibuja el ratio de verdaderos positivos (otro nombre para el *recall*) contra el ratio de falsos positivos. El *FPR* es el ratio de instancias negativas incorrectamente clasificadas como positivas. Es 1 menos el ratio de verdaderos negativos llamado *TNR* que se conoce como especificidad. Una *curva ROC* de un modelo muestra la diferencia entre el ratio de verdaderos positivos (*TPR*) y el ratio de falsos positivos (*FPR*). Dados un conjunto de datos de test y un modelo, *TPR* es la proporción de ejemplos positivos que han sido correctamente clasificados por el modelo y *FPR* es la proporción de ejemplos negativos que han sido clasificados de manera incorrecta como positivos, es decir:

$$TPR = \frac{TP}{P} \quad FPR = \frac{FP}{N}$$

Para una clase con dos clases o etiquetas, una curva ROC nos permite visualizar la diferencia entre el ratio en el que un modelo puede reconocer casos positivos contra el ratio en el que podría identificar como positivos casos que no lo son en diferentes partes de los datos de test. Cualquier incremento en *TPR* ocurre a costa de un incremento en *FPR*. El área de la *curva ROC* es una medida de la *accuracy* del modelo.

Para dibujar una *curva ROC* de un modelo de clasificación *M*, el modelo debería devolver la probabilidad de la clase a la que pertenece el ejemplo. Con esta información ordenamos los ejemplos para que la clase con más probabilidad de ser positiva quede en la parte derecha del gráfico y la que menos probabilidad quede a la parte baja de la lista. Por ejemplo los modelos Naïve Bayes y Regresión Logística devuelven probabilidades, aunque otros como los CART pueden modificarse para lo hagan.

La diagonal de una curva ROC muestra donde los verdaderos positivos son igual de probables que los falsos positivos. Cuanto más pegada está la curva a la diagonal, peor eficiencia (*accurate*) tiene el modelo. Para calcular la *accuracy* del modelo, podemos mediar el área bajo la curva (**AUC**, Area Under Curve). Una AUC cercana a 0.5, indica un modelo con poca *accuracy* y cuanto más cercana a 1.0 esté mejor.

#### EJEMPLO 44: Calcular la *curva ROC* de un clasificador.

```
# -*- coding: utf-8 -*-
from sklearn import metrics
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression
import matplotlib.pyplot as plt
iris = load_iris()
x = iris["data"][:,(2, 3)] # ancho y largo de pétalos
y = (iris["target"] == 2).astype(int) # clase Iris-Virginica
# Crear y entrenar el modelo
rlog = LogisticRegression(solver="lbfgs", C=1000, random_state=42)
modelo = LogisticRegression() # Crea el objeto regresión logística
modelo.fit(x, y)
# Calcular los ratios FPR y TPR
fpr, tpr, _ = metrics.roc_curve(y, modelo.predict_proba(x)[:,1])
roc_auc = metrics.auc(fpr, tpr) # Calcula AUC
print("ROC AUC: %0.3f" % roc_auc)
# Dibujar la curva ROC
plt.figure()
plt.plot(fpr, tpr, label='curva ROC (area = %0.3f)' % roc_auc)
plt.plot([0, 1], [0, 1], 'k--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('Ratio de Falsos Positivos (FPR)')
plt.ylabel('Ratio de Verdaderos Positivos (TPR)')
```

```
plt.title('Curva ROC')
plt.legend(loc="lower right")
plt.show()
```

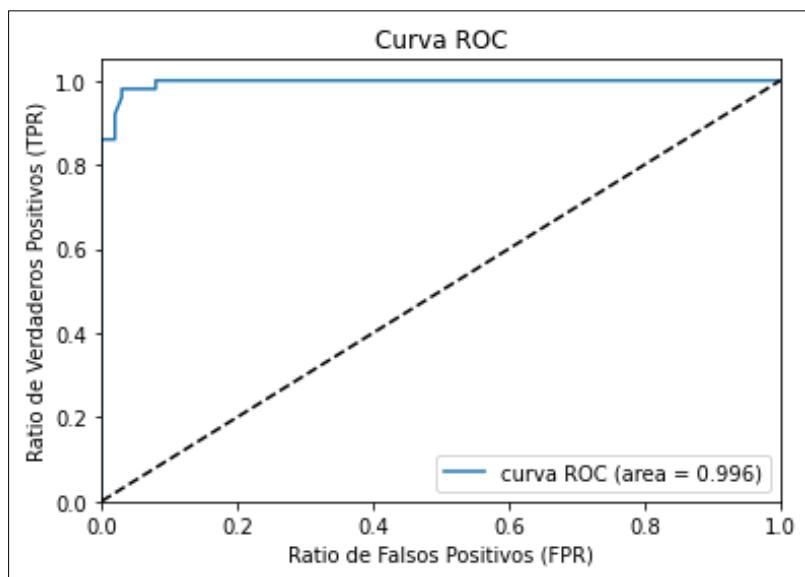


Figura 74: Curva ROC de un modelo.

Haz el [ejercicio 5](#) de la relación de problemas.

## 10. SELECCIÓN DE MODELOS.

Elegir uno de los posibles modelos que estemos entrenando para quedarnos con el mejor. Y para quedarnos con el mejor debemos alimentarlos con los mejores datos y configurar los algoritmos de aprendizaje con los mejores hiperparámetros. De estas cuestiones trata este apartado de la unidad.

### 10.1. MÉTODOS HOLDOUT Y RANDOM SUBSAMPLING.

El método **holdout** consiste en particionar aleatoriamente los datos en dos conjuntos independientes, un conjunto de entrenamiento (*train*) y otro de comprobación (*test*). Normalmente un 70% o un 80% de los datos se envían al conjunto *train* y el resto al *test*. Los datos de entrenamiento se emplean para entrenar el modelo y los datos de *test* se emplean para medir su eficiencia. La estimación es pesimista porque solamente una parte de los datos iniciales se ha usado para generar el modelo.

**EJEMPLO 37:** Validar un modelo de la manera incorrecta. Esta aproximación es muy simplona y poco fiable.

```
from sklearn.datasets import load_iris
iris = load_iris()
X = iris.data
y = iris.target
from sklearn.neighbors import KNeighborsClassifier
modelo = KNeighborsClassifier(n_neighbors=1)
modelo.fit(X, y)
y_modelo = modelo.predict(X)
from sklearn.metrics import accuracy_score
print("Accuracy:", accuracy_score(y, y_modelo)) # Salida: 1.0
```

El resultado es 1 lo que significa que acierta el 100% de las veces en clasificar los ejemplos proporcionados para predecir. Pero esto es poco fiable porque se han usado los mismos datos para entrenar que para validar. Además, **KNN** es un modelo basado en instancias que almacena los datos de entrenamiento, luego siempre nos dará el 100% de eficiencia hasta que le demos datos reales y en ese caso fallará. Por esos dos motivos esta validación no es fiable.

**EJEMPLO 38:** Validar usando *holdout*.

```
from sklearn.cross_validation import train_test_split
```

```
# ...Sentencias que cargan datos y crean el modelo como el ejemplo 32
X1, X2, y1, y2 = train_test_split(X, y, random_state=0, train_size=0.7)
modelo.fit(X1, y1) # Entrenar el modelo en un conjunto de datos
y2_modelo= modelo.predict(X2) # Evaluar el modelo con el segundo conjunto de datos
print("Accuracy:",accuracy_score(y2, y2_modelo)) # Salida: 0.9066666666666662
```

**Random subsampling** es una variación del método *holdout* en el que el método *holdout* se repite k veces. La métrica accuracy se estima como la media de todas las accuracy obtenidas en cada iteración.

**Nota:** si además queremos seleccionar el mejor de entre varios modelos, deberíamos buscar los mejores hiperparámetros de cada uno antes de escoger el mejor. En este caso, deberíamos particionar los datos en train, validación y test y validar los modelos contra los datos de validación reservando los datos de test para cuando ya se hayan encontrado los mejores parámetros y así evitar que los hiperparámetros sobreajusten a los datos de test dándonos resultados poco fiables en cuanto a su desempeño real.

## 10.2. VALIDACIÓN CRUZADA (CROSS-VALIDATION).

En la **k-fold cross-validation**, los datos iniciales son aleatoriamente particionados en k subconjuntos mutuamente excluyentes también llamados **folds** y que representamos como  $D_1, D_2, \dots, D_k$  y aproximadamente del mismo tamaño. El entrenamiento y el test se realizan k veces. En la iteración  $i$ , la partición  $D_i$  se reserva como conjunto de test y el resto de particiones son colectivamente usadas para entrenar el modelo. Así que en la primera iteración los folds  $D_2, \dots, D_k$  sirven como datos de entrenamiento para generar al primer modelo que se testeó contra  $D_1$ . En la segunda iteración, para entrenar el segundo modelo se utilizan  $D_1, D_3, \dots, D_k$  y se testeó con  $D_2$ . Al contrario que el método *holdout* y *random subsampling*, todos los ejemplos se utilizan el mismo número de veces para entrenar y una para validar. Para un problema de clasificación la *accuracy* se estima como el número de clasificaciones correctas en las k-iteraciones de todos los k modelos, dividido por el número de tuplas en los datos iniciales.

**Leave-one-out-cross-validation** es una variante de la *cross-validation* donde k se fija al número de tuplas inicial. Así que en cada iteración, solamente se aparta un ejemplo para realizar el test. Se usa cuando la cantidad de datos es pequeña.

En **stratified cross-validation**, los folds se estratifican para que la distribución de clases de las tuplas en cada fold sea aproximadamente la misma que en los datos iniciales.

En la práctica se recomienda utilizar una validación cruzada de 10-folds para estimar la accuracy de un clasificador (incluso aunque tengas potencia de computación para utilizar un mayor número) debido a que ya tienes un bias relativamente bajo y varianza.

En la figura se ve una división en 5 grupos, cada uno con aproximadamente 5/4 de los datos dedicados a entrenar y 1/5 dedicado a comprobar. Hacerlo en *Scikit-learn* es sencillo:

```
from sklearn.cross_validation import cross_val_score
cross_val_score(modelo, X, y, cv=5)
```

Además *Scikit-Learn* implementa diferentes variaciones para casos especiales. Por ejemplo el caso extremo de hacer una *leave-one-out*:

```
from sklearn.cross_validation import LeaveOneOut
scores = cross_val_score(modelo, X, y, cv=LeaveOneOut(len(X)))
print(scores)
```

Si tenemos 150 ejemplos, crea 150 intentos y nos devuelve para cada uno un 1.0 si ha tenido éxito y 0.0 en caso de que falle. Tomando la media nos da una estimación del ratio de error:

```
scores.mean()
```

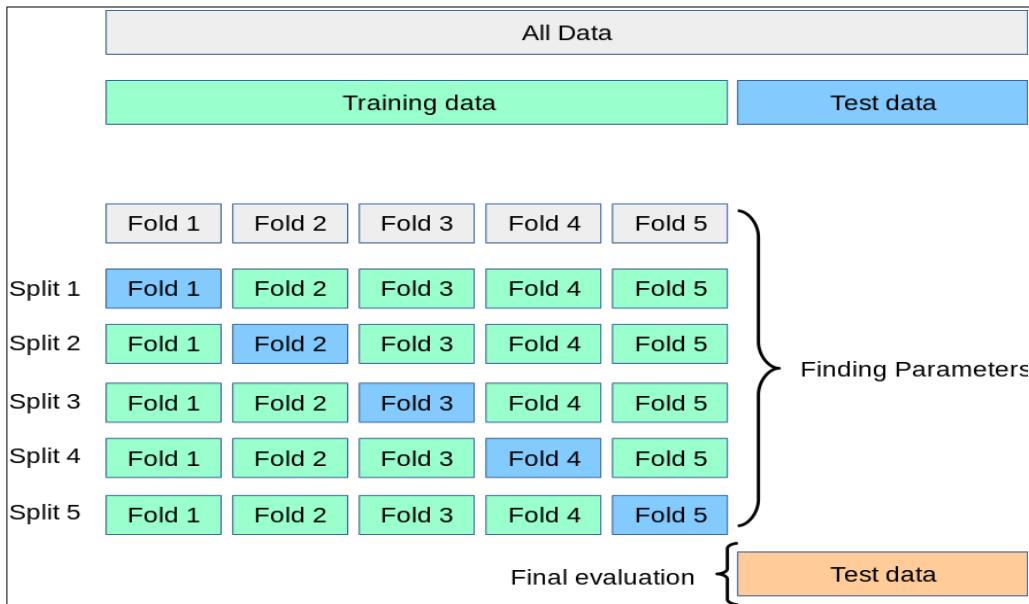


Figura 75: Representación de 5-validación cruzada.

Otros tipos de validaciones cruzadas se utilizan de manera similar. Puedes consultar la documentación de *Scikit-Learn* para ampliar.

Las medidas de rendimiento (la función  $\text{score } \mathcal{S}$ ), ya sea una medida de una clasificación correcta o un error medio en las predicciones, que expresamos como  $\mathcal{L}$ . Esta función se evalúa sobre el grupo de datos del test. Hay dos estrategias:

- **Micro medidas (medidas individuales):** calcula un score  $\mathcal{S}$  para cada ejemplo y saca la media de todos los ejemplos. Es similar a la media.

$$\mathcal{S}(f) = \frac{1}{N} \sum_i^N \mathcal{L}(y_i, f(\mathbf{x}_{-k(i)}, \mathbf{y}_{-k(i)}))$$

- **Macro medidas (CV scores):** calcula el score  $\mathcal{S}$  en cada fold y hace la media de todos ellos.

$$\mathcal{S}(f) = \frac{1}{K} \sum_k^K \mathcal{S}_k(f).$$

$$\mathcal{S}(f) = \frac{1}{K} \sum_k^K \frac{1}{N_k} \sum_{i \in k} \mathcal{L}(y_i, f(\mathbf{x}_{-k(i)}, \mathbf{y}_{-k(i)}))$$

Estas dos medidas (una media de medias vs una media global) son generalmente muy parecidas.

**EJEMPLO 39:** Validación cruzada para Regresión realizada de manera manual y de forma automática.

```
import numpy as np
from sklearn import datasets
import sklearn.linear_model as lm
import sklearn.metrics as metrics
from sklearn.model_selection import KFold
import matplotlib.pyplot as plt
X, y = datasets.make_regression(n_samples=100, n_features=100, n_informative=10,
random_state=42)
modelo = lm.Ridge(alpha=10)
cv = KFold(n_splits=5)
r2_train, r2_test = list(), list()
```

```

for train, test in cv.split(X):
    modelo.fit(X[train, :], y[train])
    r2_train.append(metrics.r2_score(y[train], modelo.predict(X[train, :])))
    r2_test.append(metrics.r2_score(y[test], modelo.predict(X[test, :])))
print("Train r2:%.2f" % np.mean(r2_train))
print("Test r2:%.2f" % np.mean(r2_test))
# De manera automática
from sklearn.model_selection import cross_val_score
scores = cross_val_score(estimator=modelo, X=X, y=y, cv=5)
print("Test r2:%.2f" % scores.mean())
# Aportando un cv
cv = KFold(n_splits=5)
scores = cross_val_score(estimator=modelo, X=X, y=y, cv=cv)
print("Test r2:%.2f" % scores.mean())

```

**EJEMPLO 40:** Validación cruzada para Clasificación. Cuando hagamos divisiones sería bueno que en cada fold hubiese aproximadamente la misma proporción de ejemplos de cada clase, lo que se conoce como estratificación. Por eso usaremos una variación de *KFold* que es *StratifiedKFold*. Normalmente la función de coste L() es la sensitividad y la especificidad.

```

import numpy as np
from sklearn import datasets
import sklearn.linear_model as lm
import sklearn.metrics as metrics
from sklearn.model_selection import StratifiedKFold
X, y = datasets.make_classification(n_samples=100, n_features=100,
n_informative=10, random_state=42)
modelo = lm.LogisticRegression(C=1, solver='lbfgs')
cv = StratifiedKFold(n_splits=5)
recalls_train, recalls_test, acc_test = list(), list(), list()
y_test_pred = np.zeros(len(y))
for train, test in cv.split(X, y):
    modelo.fit(X[train, :], y[train])
    recalls_train.append(metrics.recall_score(y[train], modelo.predict(X[train, :]),
                                                average=None))
    recalls_test.append(metrics.recall_score(y[test], modelo.predict(X[test, :]),
                                              average=None))
    acc_test.append(metrics.accuracy_score(y[test], modelo.predict(X[test, :])))
    # Almacenar las predicciones de test (para micro medidas)
    y_test_pred[test] = modelo.predict(X[test, :])
print("== Macro medidas ==")
# Usar una lista de scores
recalls_train = np.array(recalls_train)
recalls_test = np.array(recalls_test)
print("Train SPC:.2f; SEN:.2f" % tuple(recalls_train.mean(axis=0)))
print("Test SPC:.2f; SEN:.2f" % tuple(recalls_test.mean(axis=0)), )
print("Test ACC:.2f, BACC:.2f" %
(np.mean(acc_test), recalls_test.mean(axis=1).mean()), "Folds:", acc_test)
# O usar un vector para testear predicciones
acc_test = [metrics.accuracy_score(y[test], y_test_pred[test]) for train, test in cv.split(X, y)]
print("Test ACC:.2f" % np.mean(acc_test), "Folds:", acc_test)
print("== Micro medidas ==")
print("Test SPC:.2f; SEN:.2f" % tuple(metrics.recall_score(y, y_test_pred, average=None)))
print("Test ACC:.2f" % metrics.accuracy_score(y, y_test_pred))
# De manera automática
from sklearn.model_selection import cross_val_score
scores = cross_val_score(estimator=modelo, X=X, y=y, cv=5)
print("Scores:", scores.mean())
# provide CV and score
def balanced_acc(estimator, X, y, **kwargs):
    return metrics.recall_score(y, estimator.predict(X), average=None).mean()
scores = cross_val_score(estimator=modelo, X=X, y=y, cv=5, scoring=balanced_acc)
print("Test ACC:.2f" % scores.mean())

```

Las métricas que podemos escoger se pueden indicar en el parámetro **scoring** (el nombre) y son:

<b>Regresión</b>	
"explained_variance"	metrics.explained_variance_score
"max_error"	metrics.max_error
"neg_mean_absolute_error"	metrics.mean_absolute_error
"neg_mean_squared_error"	metrics.mean_squared_error
"neg_root_mean_squared_error"	metrics.mean_squared_error
"neg_mean_squared_log_error"	metrics.mean_squared_log_error
"neg_median_absolute_error"	metrics.median_absolute_error
"r2"	metrics.r2_score
"neg_mean_poisson_deviance"	metrics.mean_poisson_deviance
"neg_mean_gamma_deviance"	metrics.mean_gamma_deviance
"neg_mean_absolute_percentage_error"	metrics.mean_absolute_percentage_error

Para clasificación:

<b>Clasificación</b>	
"accuracy"	metrics.accuracy_score
"balanced_accuracy"	metrics.balanced_accuracy_score
"top_k_accuracy"	metrics.top_k_accuracy_score
"average_precision"	metrics.average_precision_score
"neg_brier_score"	metrics.brier_score_loss
"f1"	metrics.f1_score
"f1_micro"	metrics.f1_score
"f1_macro"	metrics.f1_score
"f1_weighted"	metrics.f1_score
"f1_samples"	metrics.f1_score
"neg_log_loss"	metrics.log_loss
"precisión", etc.	metrics.precision_score
"recuperación" etc.	metrics.recall_score
"jaccard" etc.	metrics.jaccard_score
"roc_auc"	metrics.roc_auc_score
"roc_auc_ovr"	metrics.roc_auc_score
"roc_auc_ovo"	metrics.roc_auc_score
"roc_auc_ovr_weighted"	metrics.roc_auc_score
"roc_auc_ovo_weighted"	metrics.roc_auc_score

Para *clustering*:

<b>Análisis de conglomerados</b>	
"adjusted_mutual_info_score"	metrics.adjusted_mutual_info_score
"adjusted_rand_score"	metrics.adjusted_rand_score
"completeness_score"	metrics.completeness_score
"fowlkes_mallows_score"	metrics.fowlkes_mallows_score
"homogeneity_score"	metrics.homogeneity_score
"mutual_info_score"	metrics.mutual_info_score
"normalized_mutual_info_score"	metrics.normalized_mutual_info_score
"rand_score"	metrics.rand_score
"v measure score"	metrics.v_measure_score

Consultar [documentación](#). Realizar ejercicio 7 de X.

Realiza [el ejercicio 6](#) de la relación de problemas.

## 10.3. USANDO TEST ESTADÍSTICOS.

Imagina que generamos dos modelos de clasificación  $M_1$  y  $M_2$  a partir de los mismos datos. Hemos realizado una  $CV$  de  $10\text{-fold}$  para obtener la media de error de cada uno. ¿Qué modelo es mejor? Es intuitivo seleccionar el modelo con el menor ratio de error, sin embargo la media de error solo es una estimación del error que cometerán con los futuros datos que se les proporcione. Puede que haya una

considerable varianza entre los ratios de error de cualquier experimento encontrados con *CV de 10-fold*. Aunque los ratios de error medios obtenidos de  $M_1$  y  $M_2$  puedan parecer muy diferentes, estas diferencias podrían no ser estadísticamente significativas porque pueden atribuirse al azar al escoger diferentes datos. En este apartado intentamos ver métodos que nos ayuden a saber si cualquier diferencia en las medias de ratios de error es real o no.

Necesitamos emplear un **test de significancia estadística**. Además sería bueno obtener algunos límites de confianza para los ratios de error para poder hacer afirmaciones como “*Cualquier media observada no variará en  $\pm 2$  veces el error estándar un 95% de las veces con datos futuros*” o “*Un modelo es mejor que el otro con un margen de error del  $\pm 4\%$* ”.

Recordamos que el **ratio de error de un modelo  $M = 1.0 - accuracy(M)$** . Si hemos entrenado los dos modelos usando CV de 10-fold, usando datos particionados de manera independiente, podemos calcular los ratios de error de  $M_1$  y de  $M_2$  y las medias. Para un modelo, los ratios de error individuales de la CV pueden considerarse diferentes, e independientes muestras de una distribución de probabilidades.

En general, siguen una *t-distribución con  $k - 1$  grados de libertad* donde  $k = 10$  (esta distribución parece muy similar a una normal, o distribución gausiana hasta cuando las funciones que las definen sean dos bastantes diferentes. Ambas son unimodales, simétricas y acampanadas). Esto nos permitirá hacer contraste de hipótesis donde el test de significancia usado es el ***t-test* o *test de Student***.

Nuestra hipótesis es que los dos modelos son el mismo, es decir, que la diferencia en el ratio de error medio entre los dos sea cero. Si podemos rechazar esta hipótesis (*la hipótesis nula*), entonces podemos concluir que la diferencia entre los dos modelos es estadísticamente significante, en cuyo caso podemos seleccionar el modelo con menor ratio de error.

El mismo conjunto de test pueden utilizarse tanto con  $M_1$  como con  $M_2$ . En la  $i$ -ésima ronda de la CV de 10-fold obtenemos el ratio de error de  $M_1$  y  $M_2$  digamos  $err(M_1)_i$  y  $err(M_2)_i$  y las medias definen el error de cada uno como  $err(M_1)$  y  $err(M_2)$ . La varianza de la diferencia entre los dos modelos es  $var(M_1 - M_2)$ . El *t* -test calcula el *t-estadístico con  $k - 1$  grados de libertad* para  $k$  muestras. En nuestro ejemplo tenemos  $k = 10$ , el número de muestras que hemos usado. El *t*-estadístico para calcular una pareja de modelos se calcula como:

$$t = \frac{\overline{err}(M_1) - \overline{err}(M_2)}{\sqrt{var(M_1 - M_2)/k}},$$

donde,

$$var(M_1 - M_2) = \frac{1}{k} \sum_{i=1}^k [err(M_1)_i - \overline{err}(M_1) - (\overline{err}(M_1) - \overline{err}(M_2))]^2$$

Para averiguar cuando  $M_1$  y  $M_2$  son significativamente diferentes, calculamos  $t$  y seleccionamos un **nivel de significancia**, llamado *sig*. En la práctica un valor de 5% o 1% es lo normalmente usado. Consultamos una tabla para la *t-distribución* (esta tabla muestra filas ordenadas por grados de libertad y las columnas tienen diferentes niveles de columnas). Imagina que queremos afirmar que la diferencia entre  $M_1$  y  $M_2$  es significativamente diferente para un 95% de las posibilidades, es decir,  $sig = 5\%$  o 0.05. Necesitamos encontrar el valor de la *t-distribución* correspondiente a  $k-1$  grados de libertad (o 9 grados de libertad en nuestro ejemplo). Pero como la *t-distribución* es simétrica, solamente se suele mostrar la parte superior de la distribución en las tablas. Así que buscamos el valor de para  $z = sig/2$ , que en este caso es 0.025, donde  $z$  **sería el límite de confidencia**. Si  $t > z$  o  $t < -z$ , entonces nuestro valor de  $t$  queda en la región de rechazo, dentro de la cola de la distribución. Esto significa que podemos rechazar la hipótesis nula que era que las medias de  $M_1$  y  $M_2$  son la misma y concluimos que hay diferencias significativas entre los dos modelos. En otro caso, si no podemos rechazar la hipótesis nula, concluimos que cualquier diferencia entre  $M_1$  y  $M_2$  pueden atribuirse a la suerte.

Si hay dos conjuntos de test disponibles, entonces se puede usar la versión no pareada del test, donde la varianza entre las medias se estima como:

$$\text{var}(M_1 - M_2) = \frac{\text{var}(M_1)}{k_1} + \frac{\text{var}(M_2)}{k_2},$$

y  $k_1$  y  $k_2$  son el número de muestras de las CV y se conocen como las **dos muestras del t-test**. Cuando consultamos la tabla de la  $t$ -distribución, el número de grados de libertad es el mínimo de los dos modelos.

**EJEMPLO 45:** Extraer el el estadístico para comprobar si podemos considerar que un modelo es mejor que otro usando una utilidad del paquete `mlxtend` (extensión para machine learning). Si no lo tienes puedes instalarlo con: `pip install...`

```
# -*- coding: utf-8 -*-
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from mlxtend.data import iris_data
# Cargamos datos
x, y = iris_data()
# Creamos dos clasificadores diferentes y los entrenamos con los mismos datos
lr = LogisticRegression(max_iter=150)
dt = DecisionTreeClassifier(max_depth = 1)
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3)
lr.fit(x_train, y_train)
dt.fit(x_train, y_train)
s1 = lr.score(x_test, y_test)
s2 = dt.score(x_test, y_test)
print('Modelo A accuracy: %.2f%%' % (s1*100))
print('Modelo B accuracy: %.2f%%' % (s2*100))
# Probar si las accuracy son significativas para decantarnos por un modelo
from mlxtend.evaluate import paired_ttest_5x2cv
t, p = paired_ttest_5x2cv(estimator1=lr, estimator2=dt, X=x, y=y)
alfa = 0.05 # Significancia del 5%
print('t-estadístico: %.3f' % t)
print('Significancia alfa ', alfa)
print('p-value: %.3f' % p)
if p > alfa:
    print("Se acepta la hipótesis nula")
else:
    print("Se rechaza la hipótesis nula")
```

## 10.4. PROCEDIMIENTO GRID-SEARCH.

Es importante mencionar que CV puede utilizarse para dos objetivos diferentes:

1. **Validación de modelos**: habiendo escogido el modelo final, estimar su error cuando predice (error de generalización) sobre nuevos datos.

2. **Selección de Modelos**: estimar la eficiencia de diferentes modelos para escoger el mejor. Un caso especial de selección de modelos es la selección de hiperparámetros de los modelos. En vez de recordar que la mayoría de algoritmos de aprendizaje tienen hiperparámetros que deben ser fijados. Generalmente debemos solucionar los dos problemas de manera simultánea. La aproximación usada para ambos problemas puede consistir en dividir el dataset aleatoriamente en 3 partes: datos de entrenamiento, datos de validación y datos de test.

- **Datos de entrenamiento** (*train*) usados para entrenar los modelos.
- **Datos de validación** (*val*) usados para estimar el error de las predicciones o la selección de hiperparámetros sobre una rejilla de posibles valores.
- **Datos de test (test)**: usados para medir el error de generalización del modelo finalmente escogido.

Otra alternativa es la validación cruzada. Pero para encontrar los mejores hiperparámetros tenemos dos alternativas: buscar de entre unos cuantos posibles de manera exhaustiva los mejores, o buscar de

entre esos algunos aleatorios. El **grid-search** consiste en seleccionar el modelo probando los mejores hiperparámetros de entre una rejilla de posibles valores de manera sistemática.

**Para cada posible valor del hiperparámetro  $\alpha_k$ :**

1. Fijar el aprendiz en el conjunto de entrenamiento:  $f(X_{train}, y_{train}, \alpha_k)$
2. Evaluar el modelo con los datos de validación y recordar los parámetros que minimicen la medida del error  $\alpha_* = \arg \min L(f(X_{train}), y_{val}, \alpha_k)$
3. Reentrenar al aprendiz con todos los datos de entrenamiento + datos de validación usando los mejores hiperparámetros:  $f^* \equiv f(X_{train} \cup val, y_{train} \cup val, \alpha_*)$
4. Comprobación de  $f^*$  (del modelo elegido) con los datos de test:  $L(f^*(X_{test}), y_{test})$

## CV ANIDADA PARA TESTEAR MODELOS Y SELECCIONARLOS

La mayoría de las veces no podemos practicar una división de 3 formas. Pero usamos validación cruzada anidada y realizamos dos validaciones cruzadas.

**Un bucle externo de validaciones cruzadas para comprobar modelos.** Esta CV realiza  $K$  divisiones de los datos en train + val ( $X_K, y_K$ ) y un test  $X_K, y_K$

**Un bucle interno CV, para selección de modelo.** Para cada ejecución del bucle externo, el bucle interno realiza  $L$  divisiones del dataset ( $X_K, y_K$ ) en train ( $X_{K,L}, y_{K,L}$ ) y val ( $X_{K,L}, y_{K,L}$ ).

En este ejemplo, el bucle CV interno se combina con el aprendiz formando un nuevo aprendiz con un procedimiento de selección de modelo (parámetros) automático. Este nuevo aprendiz puede construirse fácilmente con *Scikit learn*. El aprendiz es envuelto dentro de la clase **GridSearchCV**. Luego el nuevo aprendiz puede ser pegado dentro del bucle CV externo.

**EJEMPLO 41:** Implementación de *grid-search*.

```
import numpy as np
from sklearn import datasets
import sklearn.linear_model as lm
from sklearn.model_selection import GridSearchCV
import sklearn.metrics as metrics
from sklearn.model_selection import KFold
# Dataset
ruido_sd = 10
X, y, coef = datasets.make_regression(n_samples=50, n_features=100, noise=ruido_sd,
n_informative=2, random_state=42, coef=True)
# Usar esto para ajustar el parámetro noise para que snr < 5
print("SNR:", np.std(np.dot(X, coef)) / ruido_sd)
# Parámetro grid sobre alfa & l1_ratio
param_grid = {'alpha': 10.0 ** np.arange(-3, 3), 'l1_ratio':[.1, .5, .9]}
# Warp
modelo = GridSearchCV(lm.ElasticNet(max_iter=10000), param_grid, cv=5)
```

## MODELOS DE REGRESIÓN CON VALIDACIÓN CRUZADA PREDEFINIDA

*Sklearn* automáticamente selecciona un grid de parámetros, la mayoría de las veces usando valores por defecto, por ejemplo **n\_jobs** es el número de CPU's que se usará durante la CV. Si es -1 usa todas las CPU's. Aquí aparecen algunas alternativas que puedes usar con *grid search*:

**1) Todos los datos para entrenar y omitir el bucle externo de CV:**

```
modelo.fit(X, y)
print(f"Train r2:{%.2f} {metrics.r2_score(y, modelo.predict(X))}")
print(modelo.best_params_)
```

**2) El usuario hace el bucle externo de CV y lo usa para extraer información concreta:**

```
cv = KFold(n_splits=5, random_state=42)
r2_train, r2_test = list(), list()
alfas = list()
for train, test in cv.split(X, y):
    X_train, X_test, y_train, y_test = X[train, :], X[test, :], y[train], y[test]
    modelo.fit(X_train, y_train)
    r2_train.append(metrics.r2_score(y_train, modelo.predict(X_train)))
    r2_test.append(metrics.r2_score(y_test, modelo.predict(X_test)))
    alfas.append(modelo.best_params_['alpha'])
```

```

r2_test.append(metrics.r2_score(y_test, modelo.predict(X_test)))
r2_train.append(metrics.r2_score(y_train, modelo.predict(X_train)))
alfas.append(modelo.best_params_)
print("Train r2:%.2f" % np.mean(r2_train))
print("Test r2:%.2f" % np.mean(r2_test))
print("Alfas seleccionados:", alfas)

```

**3) Usar sklearn para definir el bucle externo de CV de manera amigable.**

```

from sklearn.model_selection import cross_val_score
scores = cross_val_score(estimator=modelo, X=X, y=y, cv=cv)
print("Test r2:%.2f" % scores.mean())
# Dataset
X, y, coef = datasets.make_regression(n_samples=50, n_features=100, noise=10,
                                         n_informative=2, random_state=42, coef=True)
print("== Ridge (L2 penalización) ==")
modelo = lm.RidgeCV(cv=3)
# Con sklearn selecciona una lista de alfas con valores por defecto de L00-CV
scores = cross_val_score(estimator=modelo, X=X, y=y, cv=5)
print("Test r2:%.2f" % scores.mean())
print("== Lasso (L1 penalización) ==")
modelo = lm.LassoCV(n_jobs=-1, cv=3)
# Con sklearn selecciona una lista de alfas por defecto con 3CV
scores = cross_val_score(estimator=modelo, X=X, y=y, cv=5)
print("Test r2:%.2f" % scores.mean())
print("== ElasticNet (L1 penalización ==")
modelo = lm.ElasticNetCV(l1_ratio=[.1, .5, .9], n_jobs=-1, cv=3)
# Con sklearn selecciona una lista de alfas con 3CV por defecto
scores = cross_val_score(estimator=modelo, X=X, y=y, cv=5)
print("Test r2:%.2f" % scores.mean())

```

**4) Validación de modelos con CV predefinida:**

```

from sklearn import datasets
import sklearn.linear_model as lm
import sklearn.metrics as metrics
from sklearn.model_selection import cross_val_score
X, y = datasets.make_classification(n_samples=100, n_features=100, n_informative=10,
                                         random_state=42)
# Aportar valores de CV y score
def balanced_acc(estimator, X, y, **kwargs):
    return metrics.recall_score(y, estimator.predict(X), average=None).mean()
print("== Ridge Logístico (L2 penalización) ==")
modelo= lm.LogisticRegressionCV(class_weight='balanced', scoring=balanced_acc, n_jobs=-1,
                                         cv=3)
# Con sklearn selecciona una lista de alfas con L00-CV
scores = cross_val_score(estimator=modelo, X=X, y=y, cv=5)
print("Test ACC:%.2f" % scores.mean())

```

Realiza el ejercicio 7 de la relaciónnd e problemas.

## 10.5. TEST DE PERMUTACIONES ALEATORIAS.

Un *test de permutaciones* es un tipo de test no paramétrico en el que la distribución nula (la hipótesis nula) de un test estadístico es estimada realizando permutaciones aleatorias de las observaciones. El test de permutaciones es muy atractivo porque no hace suposiciones de que las observaciones sean independientes ni estén distribuidas de manera idéntica bajo la hipótesis nula.

1. Calcular el estadístico observado (*tobs*) en los datos.
2. Usa aleatorización para calcular la distribución de *t* bajo la hipótesis nula: Realiza *N* permutaciones aleatorias de los datos. Para cada ejemplo *i* de los datos permutados calcula su estadístico *t<sub>i</sub>*. Este procedimiento nos aporta la distribución de *t* bajo la hipótesis nula *H<sub>0</sub>*: *P(t|H<sub>0</sub>)*
3. Calcular el *p-value* = *P(t > tobs|H<sub>0</sub>)* |{*t<sub>i</sub> > tobs*}|, donde *t<sub>i</sub>* incluye *tobs*.

**EJEMPLO 42:** Utiliza el test de permutaciones aleatorias para medir el estadístico de una correlación.

```

import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt
np.random.seed(42)
x = np.random.normal(loc=10, scale=1, size=100)
y = x + np.random.normal(loc=-3, scale=3, size=100) # snr = 1/2
# Permutación: simula la hipótesis nula
nperm = 10000
perms = np.zeros(nperm + 1)
perms[0] = np.corrcoef(x, y)[0, 1]
for i in range(1, nperm):
    perms[i] = np.corrcoef(np.random.permutation(x), y)[0, 1]
# Dibujar recalcando pesos para obtener la distribución
pesos = np.ones(perms.shape[0]) / perms.shape[0]
plt.hist([perms[perms >= perms[0]], perms], histtype='stepfilled', bins=100,
         label=[["t > t_obs (p-value)", "t<t_obs"], pesos[perms >= perms[0]], pesos])
plt.xlabel("Distribución estadística bajo la hipótesis nula")
plt.axvline(x=perms[0], color='blue', linewidth=1, label="Estadístico observado")
_ = plt.legend(loc="upper left")
# p-value observado en test de 1-cola
pval_perm = np.sum(perms >= perms[0]) / perms.shape[0]
# Compararlo con el test de correlación de Pearson
_, pval_test = stats.pearsonr(x, y)
print("Permutación doble cola p-value=%.5f. Test de Pearson p-value=%.5f" % (2*pval_perm,
pval_test))

```

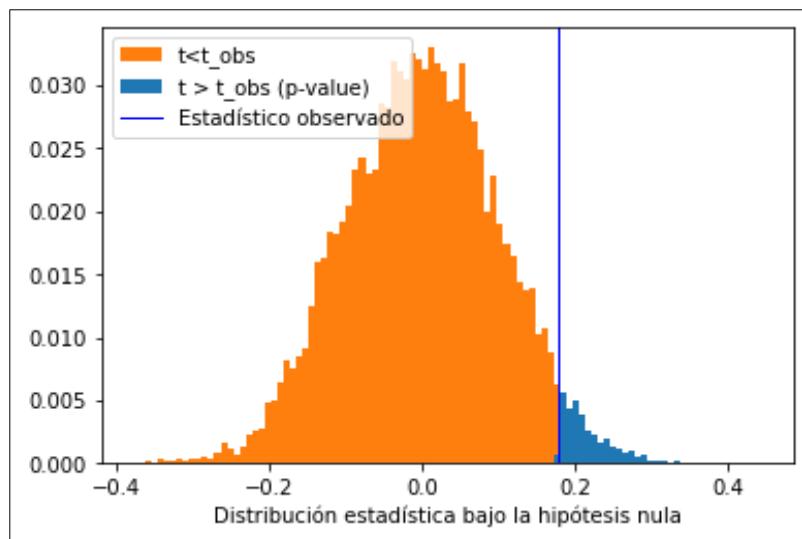


Figura 76: Distribución de la correlación.

## 10.6. MÉTODOS BOOTSTRAP.

Al contrario que los métodos mencionados con anterioridad, el método **bootstrap** muestrea los ejemplos de entrenamiento uniformemente con remplazamiento. Así que cada vez que un ejemplo es seleccionado, puede volver a ser seleccionado y vuelto a añadir al conjunto de entrenamiento.

Hay varios métodos *bootstrap*. Uno muy usado es el **.632 bootstrap**, que dado un conjunto de  $d$  ejemplos, los datos son muestreados  $d$  veces, con remplazamiento, dando como resultado una muestra *bootstrap sample* o datos de entrenamiento de  $d$  ejemplos. Algunos de los ejemplos usados pueden aparecer más de una vez en la muestra. Los ejemplos que no se utilicen en los datos de entrenamiento se usarán como datos de test. Si intentamos realizarlo varias veces, de media, el 63.2% de los ejemplos acaban usándose en la muestra *bootstrap* y el 36.8% restante formarán parte de los datos de test. Cada ejemplo tiene una probabilidad de  $1/d$  de acabar siendo seleccionado, y una probabilidad de  $(1 - 1/d)$  de no ser seleccionado. Si seleccionamos  $d$  veces, la probabilidad de que un ejemplo no sea escogido nunca es de  $(1 - 1/d)^d$ . Si  $d$  es grande, la probabilidad se acerca a  $e^{-1} = 0.368$ .

Si repetimos el muestreo  $k$  veces, en cada iteración usamos los actuales datos de test para obtener una estimación de la accuracy del modelo obtenido con los datos de bootstrap. La *accuracy* del modelo  $M$  se estima como:

$$Acc(M) = \frac{1}{k} \sum_{i=1}^k (0.632 \times Acc(M_i)_{test\_set} + 0.368 \times Acc(M_i)_{train\_set}),$$

donde  $Acc(M_i)_{test\_set}$  es la accuracy del modelo obtenido con los datos de la  $i$ -ésima muestra de *bootstrap* cuando se aplica con sus  $i$ -ésimos datos de test y  $Acc(M_i)_{train\_set}$  es la accuracy del modelo obtenido con los  $i$ -ésimos datos de entrenamiento de bootstrap usando los datos de entrenamiento del conjunto original. *Bootstrapping* tiende a ser optimista. Funciona bien con pequeñas cantidades de datos.

Una gran ventaja de *bootstrap* es su simplicidad. Es una forma muy directa de encontrar una estimación de los errores estándar y de intervalos de confianza para estimadores complejos de complejos parámetros de la distribución, como puntos percentiles, proporciones, ratios de ventajas (*odds ratio*) y coeficientes de correlación. El algoritmo:

1. Realizar  $B$  muestras del dataset con remplazamiento.
2. Para cada muestra  $i$  entrenar el modelo y calcular sus scores.
3. Calcular los errores estándar y los intervalos de confianza de los scores usando los scores obtenidos en los  $B$  datos muestreados.

**EJEMPLO 43:** Utiliza el test de permutaciones aleatorias para medir el estadístico de una correlación.

```
import numpy as np
from sklearn import datasets
import sklearn.linear_model as lm
import sklearn.metrics as metrics
import pandas as pd
# Dataset de Regresión
n_predictoras = 5
n_predictoras_info = 2
n_ejemplos = 100
X = np.random.randn(n_ejemplos, n_predictoras)
beta = np.zeros(n_predictoras)
beta[:n_predictoras_info] = 1
Xbeta = np.dot(X, beta)
eps = np.random.randn(n_ejemplos)
y = Xbeta + eps
# Entrenar el modelo con todos los datos (riesgo de sobreajuste)
modelo = lm.RidgeCV()
modelo.fit(X, y)
print("Coeficientes con todos los datos:\n", modelo.coef_)
# Bucle Bootstrap
nboot = 100 # En realidad como mínimo 1000
scores_nombres = ["r2"]
scores_boot = np.zeros((nboot, len(scores_nombres)))
coefs_boot = np.zeros((nboot, X.shape[1]))
orig_all = np.arange(X.shape[0])
for boot_i in range(nboot):
    boot_tr = np.random.choice(orig_all, size=len(orig_all), replace=True)
    boot_te = np.setdiff1d(orig_all, boot_tr, assume_unique=False)
    Xtr, ytr = X[boot_tr, :], y[boot_tr]
    Xte, yte = X[boot_te, :], y[boot_te]
    modelo.fit(Xtr, ytr)
    y_pred = modelo.predict(Xte).ravel()
    scores_boot[boot_i, :] = metrics.r2_score(yte, y_pred)
    coefs_boot[boot_i, :] = modelo.coef_
# Calcular la media, Error Estándar, Intervalos de Confianza
scores_boot = pd.DataFrame(scores_boot, columns=scores_nombres)
scores_stat = scores_boot.describe(percentiles=[.975, .5, .025])
print("r2: Media=%.2f, Error Standar=%.2f, Intervalos de Confianza=(%.2f %.2f)" %\n
```

```

tuple(scores_stat.loc[["mean", "std", "2.5%", "97.5%"], "r2"]))
coefs_boot = pd.DataFrame(coefs_boot)
coefs_stat = coefs_boot.describe(percentiles=[.975, .5, .025])
print("Distribución de coeficientes:", coefs_stat)

```

## 11. MEJORAR LA EFICIENCIA DE LOS MODELOS.

Los métodos *Bagging*, *boosting* y *random forests* son ejemplos de **métodos ensemble**. Un ensemble combina una serie de  $k$  *modelos aprendices* (o clasificadores base) denominados  $M_1, M_2, \dots, M_k$  con la intención de crear un modelo  $M$  de clasificación compuesto y mejorado.

Dados un conjunto de datos  $D$ , se usa para crear  $k$  conjuntos de datos de *train* denominados  $D_1, D_2, \dots, D_k$  donde  $D_i$  ( $1 \leq i \leq k$ ) se utiliza para generar el modelo clasificador  $M_i$ . Dado un nuevo ejemplo a clasificar, cada uno de los clasificadores calcula su predicción y la devuelve como un voto. El modelo *ensemble* devuelve una predicción basándose en los votos de los clasificadores que lo componen.

**El modelo ensemble tiende a ser más eficiente que sus clasificadores base.** Por ejemplo considera un ensamble que realiza una votación por mayoría. Dado un ejemplo  $x$  a clasificar, recopila las clasificaciones de cada uno de sus clasificadores. Algunos clasificadores pueden cometer un error, pero el modelo solamente clasificará mal si la mitad de ellos lo hace mal.

Los modelos base se denominan *weak learners* y se usan como bloques de construcción para diseñar modelos más complejos combinando varios de ellos. Muchas veces los modelos básicos no realizan demasiado bien su trabajo porque tienen alto bias (modelos con pocos grados de libertad que son demasiado simples) o demasiada varianza para ser robustos (demasiado complejos). La idea es que al combinar varios modelos simples, crean un modelo fuerte. Generalmente, los modelos ensemble consiguen:

- Decrementar la varianza con el método *bagging* (*Bootstrap Aggregating*)
- Reducir el bias con la técnica de *boosting*.
- Mejorar la eficiencia predictiva con la técnica de *stacking*.

### 11.1. BAGGING.

**Bootstrap aggregation** (*bagging*) fue propuesto por Leo Breiman en 1994, y consiste en una agregación de modelos para reducir la varianza. Los datos de entrenamiento se dividen en una serie de muestras con remplazamiento llamadas **muestras bootstrap**. El tamaño de las *muestras bootstrap* será la misma que el tamaño de la muestra original con  $\frac{3}{4}$  de valores originales y el resto de ejemplos completados con valores repetidos (ver figura 75).

Original Sample	1	2	3	4	5	6	7	8	9	10
Bootstrap Sample 1	3	1	6	5	10	7	7	9	2	3
Bootstrap Sample 2	3	10	9	3	9	8	7	10	2	10
Bootstrap Sample 3	3	5	4	10	7	7	6	2	6	9
Bootstrap Sample 4	10	10	10	3	6	2	5	4	7	9

Figura 77: Bootstrap rellenas con valores repetidos (resaltados).

Se crea y entrena un modelo base en cada *muestra bootstrap*, y la media de las predicciones en el caso de regresiones o la mayoría de votos en el caso de la clasificación la usa el modelo final. Si  $N$  es el número de *muestras bootstrap* creadas a partir de los datos originales. Para  $i = 1$  hasta  $N$ , entrenamos el modelo base  $C_i$ .

$$C_{final} = \text{aggregate max of } y \sum_i I(C_i = y)$$

**EJEMPLO 46:** Definir un modelo ensamble y comparar su funcionamiento con un modelo aislado.

```
# -*- coding: utf-8 -*-
# Árboles de Decisión para clasificación
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import StratifiedKFold, train_test_split, cross_val_score
from sklearn.metrics import accuracy_score
# Leer datos de entrada
df = pd.read_csv("Diabetes.csv")
X = df.iloc[:,[0, 1, 2, 3, 4, 5, 6, 7]] # variables independientes
y = df['class'].values # variable dependiente
X = StandardScaler().fit_transform(X) # Normalizar las variables
# evaluar el modelo dividiendo en train y test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=2017)
kfolds = StratifiedKFold(n_splits=5, random_state=2017, shuffle=True)
n_arboles = 100
# Árbol de decisión con 5-fold CV
dtc = DecisionTreeClassifier(random_state=2017).fit(X_train,y_train)
resultado = cross_val_score(dtc, X_train, y_train, cv=kfolds)
print("Decision Tree (aislado) - Train:", resultado.mean())
print("Decision Tree (aislado) - Test: ", accuracy_score(dtc.predict(X_test), y_test))
# Usar Bagging con 100 árboles por media/mayoría
dct_bag= BaggingClassifier(estimator=dtc, n_estimators=n_arboles, random_state=2017).
    fit(X_train, y_train)
resultados = cross_val_score(dct_bag, X_train, y_train, cv=kfolds)
print("Decision Tree (Bagging) - Train:", resultados.mean())
print("Decision Tree (Bagging) - Test: ", accuracy_score(dct_bag.predict(X_test), y_test))
```

## 11.2. BOOSTING.

Introducido por Freud y Schapire en 1995 en el algoritmo **AdaBoost** (*adaptive boosting*). El concepto clave del boosting es que en vez de la hipótesis de una variable independiente individual, se combinan hipótesis en un orden secuencial incrementando la *accuracy*. Los algoritmos boosting convierten los modelos débiles en fuertes solucionando problemas de bias. A alto nivel el proceso *AdaBoosting* consiste en 3 pasos:

- Asignar pesos uniformes a todos los datos:  $w_\theta(x) = 1/N$ , donde N es el número de datos de entrenamiento.
- En cada iteración se entrena un clasificador  $y_m(x_n)$  con los datos de entrenamiento y se actualizan los pesos para minimizar la función de error. Los pesos se calculan como:

$$W_n^{(m+1)} = W_n^{(m)} \exp \left\{ \alpha_m y_m(x_n) \neq t_n \right\}$$

La hipótesis de pesos o función de coste viene dada por:

$$\alpha_m = \frac{1}{2} \log \left\{ \frac{1 - \epsilon_m}{\epsilon_m} \right\}$$

Y el término que aparece en el ratio:

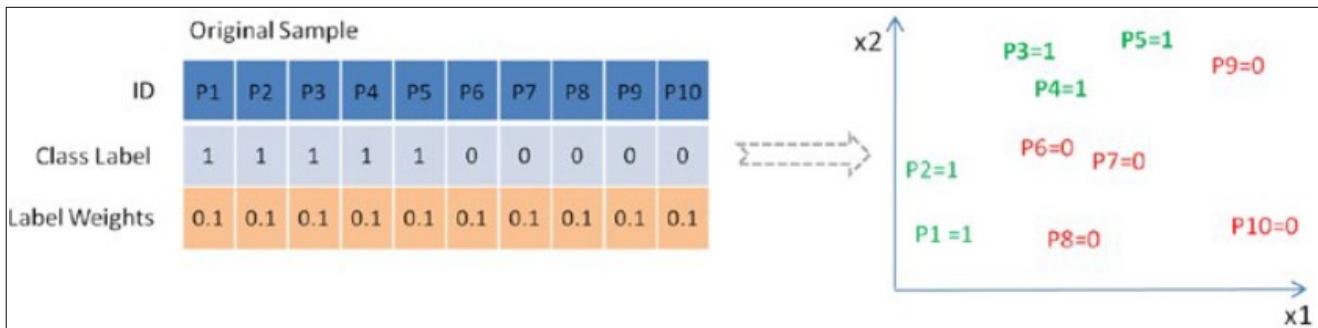
$$\epsilon_m = \frac{\sum_{n=1}^N W_n^{(m)} I(y_m(x_n) \neq t_n)}{\sum_{n=1}^N W_n^{(m)}}$$

Donde  $Y_m(x_n) \neq t_n$  tiene valores 0 y 1. Vale 0 cuando clasifica correctamente y 1 cuando comete un error.

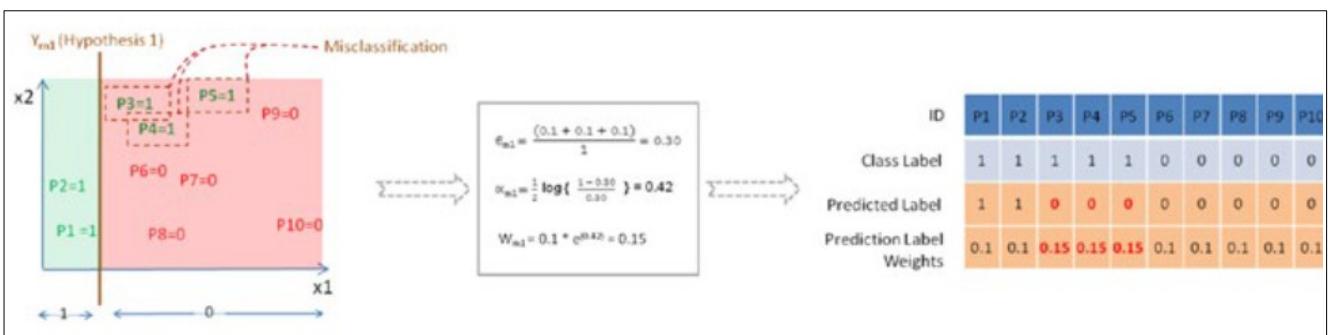
- El valor final devuelto por el modelo es:

$$Y_m = \text{sign} \left( \sum_{m=1}^M \alpha_m y_m(x) \right)$$

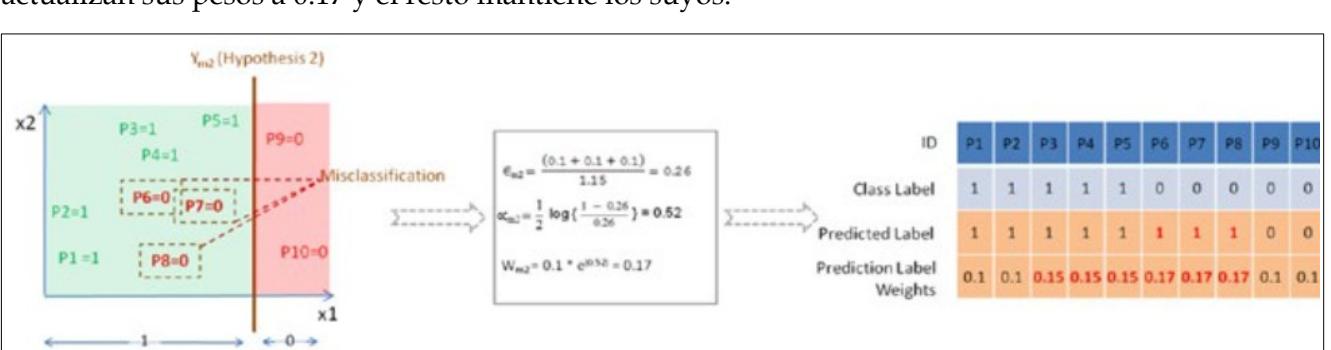
Si realizamos un ejemplo ilustrado imagina que partimos de un dataset de 10 datos con dos variables  $x_1$  y  $x_2$  y los pesos son  $1/10 = 0.1$



**Iteración 1:** 3 datos de la clase positiva se clasifican mal, así que se les asigna pesos más altos. El término de error y la función de coste se calculan como 0.3 y 0.42 respectivamente. Los datos P3, P4 y P5 toman un peso más alto de 0.15 por los errores y el resto retiene sus pesos de 0.1.



**Iteración 2:** se entrena otro modelo de clasificación como se ve en la figura 78 donde se aprecia que 3 datos de la clase negativa se clasifican mal. Los datos P6, P7 y P8 se clasifican mal. Así que se actualizan sus pesos a 0.17 y el resto mantiene los suyos.



**Iteración 3:** clasifica mal 3 datos, los datos P1 y P2 de la clase positiva y el dato P9 de la clase negativa. Así que cambian sus pesos a 0.19 y el resto mantiene sus pesos.

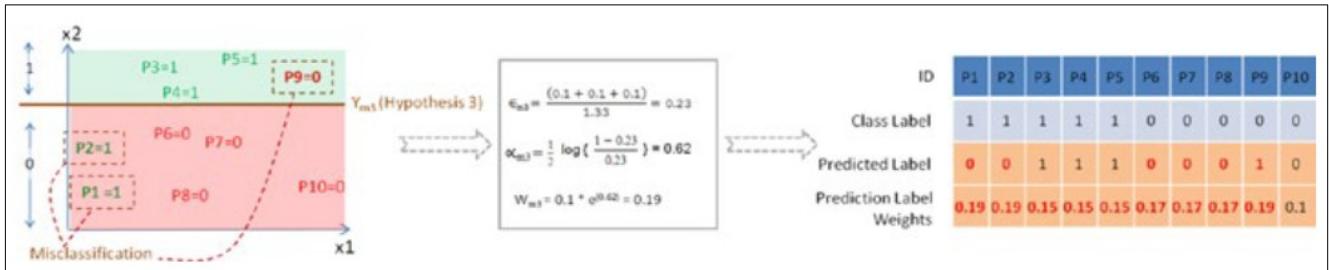


Figura 81: AdaBoost, tercer paso.

**Modelo final:** EL modelo combina los clasificadores débiles.

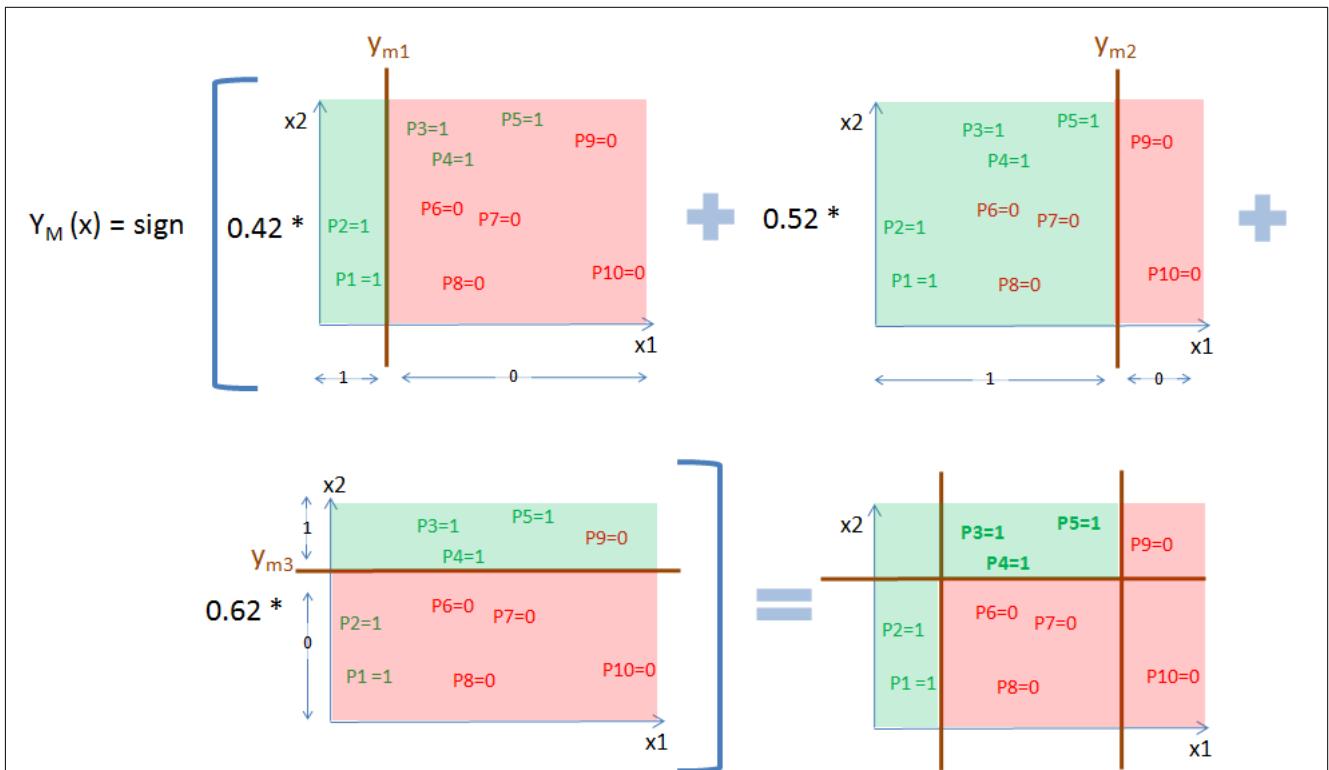


Figura 82: AdaBoost, modelo final.

**EJEMPLO 47:** Usar un clasificador AdaBoost y compararlo con uno aislado.

```
# -*- coding: utf-8 -*-
# Bárboles de Decisión para Clasificación
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split, StratifiedKFold, cross_val_score
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn import metrics
# read the data in
df = pd.read_csv("Diabetes.csv")
X = df[['age', 'serum_insulin']] # variables independientes
y = df['class'].values # variables dependientes
X = StandardScaler().fit_transform(X) # Normalizar los datos
# evaluar el modelo
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=2017)
kf = StratifiedKFold(n_splits=5, random_state=2017, shuffle=True)
n_arboles = 100
# Árbol con 5-fold CV y restricción de max_depth a 1 para tener impurezas
dt = DecisionTreeClassifier(max_depth=1, random_state=2017).fit(X_train,y_train)
r = cross_val_score(dt, X_train, y_train, cv=kf)
print("Decision Tree (aislado)-Train:", r.mean())
print("Decision Tree (aislado)-Test:", metrics.accuracy_score(dt.predict(X_test), y_test))
# Usar Adaptive Boosting de 100 iteraciones
```

```

dt_b = AdaBoostClassifier(estimator=dt, n_estimators=n_arboles, learning_rate=0.1,
random_state=2017).fit(X_train,y_train)
r = cross_val_score(dt_b, X_train, y_train, cv=kf)
print("Decision Tree (AdaBoost)-Train:", r.mean())
print("Decision Tree (AdaBoost)-Test:", metrics.accuracy_score(dt_b.predict(X_test), y_test))

```

## GRADIENT BOOSTING

Debido a la aditividad de cada etapa, la función de coste se puede expresar de una manera más adecuada para aplicar la optimización. Esto genera un nuevo tipo de algoritmos conocidos como **algoritmos boosting generalizados (GBM)**. El algoritmo **Gradient boosting** es un ejemplo de la implementación de **GBM** y puede funcionar con diferentes funciones de coste para realizar regresiones, clasificaciones, modelización de riesgos, etc. Como sugiere su nombre es un algoritmo de *boosting* que identifica el camino más corto para aprender por gradientes, *Adaboost* usa puntos de datos de alto nivel, de ahí el nombre de *Gradient Boosting*.

- Iterativamente entrena un clasificador  $y_m(x_n)$  con los datos de entrenamiento. EL modelo inicial tendrá valores constantes:

$$y_0(x) = \operatorname{argmin} \delta \sum_{i=1}^n L(y_i, \delta)$$

- Calcula la función de coste (es decir, los valores predichos contra los reales) para cada modelo entrenado en la iteración  $g_m(x)$  o calcula el gradiente negativo y lo usa para entrenar un nuevo modelo base con la función  $h_m(x)$  y encuentra el mejor gradiente para dar un paso de cierto tamaño y descender:

$$\delta_m = \operatorname{argmin} \delta \sum_{i=1}^n L(y_i, y_{m-1}(x) + \delta h_m(x))$$

- Actualiza la función estimada y devuelve  $y_m(x)$

$$y_m(x) = y_{m-1}(x) + \delta h_m(x)$$

### EJEMPLO 48: Clasificador Gradient Boosting

```

# -*- coding: utf-8 -*-
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split, StratifiedKFold, cross_val_score
from sklearn import metrics
# Leer los datos y prepararlos
df = pd.read_csv("Diabetes.csv")
X = df[['age','serum insulin']] # variables independientes
y = df['class'].values # variables dependientes
X = StandardScaler().fit_transform(X) # Normalizar los datos
# evaluar el modelo
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=2017)
kf = StratifiedKFold(n_splits=5, random_state=2017, shuffle=True)
n_arboles = 100
from sklearn.ensemble import GradientBoostingClassifier
# Usar Gradient Boosting de 100 iteraciones
gbc = GradientBoostingClassifier(n_estimators=n_arboles, learning_rate=0.1,
random_state=2017).fit(X_train, y_train)
r = cross_val_score(gbc, X_train, y_train, cv=kf)
print("Gradient Boosting - CV Train : %.2f" % r.mean())
print("Gradient Boosting - Train : %.2f" % metrics.accuracy_score(gbc.predict(X_train),
y_train))
print("Gradient Boosting - Test : %.2f" % metrics.accuracy_score(gbc.predict(X_test),
y_test))

```

## XGBOOST (eXtreme Gradient Boosting)

En marzo de 2014, Tianqi Chen diseñó **xgboost** en C++ como parte de la Distributed (Deep) Machine Learning Community y tiene una interface para Python. Es una extensión más regularizada del algoritmo *gradient boosting*. Es uno de los algoritmos más escalables de machine learning que ha surgido como solución ganadora de un concurso de Kaggle (foro de competiciones para modelado y análisis de ciencia de datos).

La función objetivo de **XGBoost** denominada  $\text{obj}(\theta)$  se calcula como:

$$\sum_i^n l(y_i - \hat{y}_i) + \sum_{k=1}^K \Omega(f_k)$$

El término de regularización es:

$$\Omega(f) = \gamma T + \frac{1}{2} \lambda \sum_{j=1}^T W_j^2$$

↓   ↓  
 Controls the overall      +      Scores of the overall  
 number of leaves created      number of leaves created

Es uno de los más usados actualmente en Machine Learning, da mejores resultados incluso que *Random-Forest* y es más rápido en encontrar la solución. Es importante elegir la cantidad de árboles que debe usar. Si usas muy pocos tendrás *underfitting* y si usas muchos tendrás *overfitting*. Una forma de forzar que se autoajuste y no cree más de los necesarios es usar ***early\_stopping\_rounds*** que habrá que darle un 10 o un 20% aproximadamente de la cantidad de iteraciones. Con este parámetro el modelo para de generar árboles si detecta que la función de coste en un conjunto de datos de validación en vez de bajar aumenta.

Algunas de ventajas clave de este algoritmo son:

- Se puede paralelizar.
- Tiene un mecanismo preconstruido para solucionar el problema de los valores ausentes, que permite que el usuario indique un valor diferente a los demás para indicar que el valor no existe (como -1 o -999).
- Dividirá los árboles hasta una profundidad máxima pero al contrario que el *Gradient Boosting* para la división de nodos si encuentra una división negativa al dividir.

**XGboost** tiene una gran cantidad de parámetros y a alto nivel podemos agruparlos en 3 categorías.

### 1. Parámetros Generales:

- a. ***nthread*** Número de threads usados. Si no se indica se usarán todos los cores.
- b. ***Booster*** El tipo de modelo que se ejecuta con ***gbtree*** (basado en árboles) por defecto para clasificaciones y ***gblinear*** para modelos lineales.

### 2. Parámetros de Boosting:

- a. ***eta*** Es el *learning rate* para prevenir overfitting. Por defecto es 0.3 y puede estar entre 0 y 1.
- b. ***max\_depth*** Máxima profundidad de los árboles con valor por defecto 6.
- c. ***min\_child\_weight*** Mínima suma de pesos de todas las observaciones de los hijos. Comienza en 1/raíz\_cuadrada(ratio\_evento).
- d. ***colsample\_bytree*** Fracción de columnas que se muestran aleatoriamente para cada árbol con valor por defecto de 1.
- e. ***Subsample*** Fracción de observaciones que son muestradas para cada árbol con valor por defecto de 1. Bajando el valor el algoritmo se vuelve más conservador para evitar el *overfitting*.

- f. **lambda** regularización de tipo L2 sobre los pesos con valor por defecto de 1.
- g. **alpha** regularización de pesos de tipo L1.

### 3. Parámetros de Tareas:

- a. **objective** - define la función de coste que se minimiza. Su valor por defecto es '**reg:linear**'. Para clasificación binaria debería ser '**binary:logistic**' y para clasificación multiclas '**multi:softprob**' para obtener el valor de la probabilidad y '**multi:softmax**' para obtener la clase. Para clasificación multiclas hay que indicar **num\_class** (número de clases).
- b. **eval\_metric** Métrica que se usará para validar la eficiencia.

**EJEMPLO 49:** Usar Xgboost para realizar una clasificación binaria.

```
# -*- coding: utf-8 -*-
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn import metrics
from xgboost.sklearn import XGBClassifier
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.model_selection import StratifiedKFold, cross_val_score
# Leer los datos de entrada
df = pd.read_csv("Diabetes.csv")
predictoras = ['age', 'serum_insulin']
target = 'class'
# Preprocesamiento como codificar clases del target y valores ausentes
for f in df.columns:
    if df[f].dtype=='object':
        lbl = LabelEncoder()
        lbl.fit(list(df[f].values))
        df[f] = lbl.transform(list(df[f].values))

df.fillna((-999), inplace=True)
# Algunas características débiles para construir los árboles
X = df[['age', 'serum_insulin']] # variables independientes
y = df['class'].values # variables dependientes
# Normalizar los datos
X = StandardScaler().fit_transform(X)
# evaluar el modelo
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=2017)
n_iter = 100 # Número de iteraciones
kf = StratifiedKFold(n_splits=5, random_state=2017, shuffle=True)
xgb = XGBClassifier(n_estimators=n_iter, objective='binary:logistic', seed=2017)
# No uso early_stopping_rounds porque me ha dado problemas ¿Versión?
xgb.fit(X_train, y_train)
r = cross_val_score(xgb, X_train,y_train, cv=kf)
print("xgBoost - CV Train: %.3f" % r.mean())
print("xgBoost - Train: %.3f" % metrics.accuracy_score(xgb.predict(X_train),
y_train))
print("xgBoost - Test: %.3f" % metrics.accuracy_score(xgb.predict(X_test),
y_test))
```

## 11.3. RANDOM FOREST.

Un subconjunto de observaciones y un subconjunto de variables que son aleatoriamente escogidas para construir varios modelos base independientes basados en árboles de decisión. Los árboles independientes están menos correlacionados porque cada uno utiliza un conjunto distinto de características predictoras durante la división, de manera que tenemos muchos pequeños y simples árboles que por separado no funcionan demasiado bien porque han sido entrenados con pocos datos, en vez de un complejo árbol que haya visto todos los datos. Los árboles son propensos a tener *overfitting* y son muy sensibles a los datos.

**EJEMPLO 50:** Usar RamdomForest de 100 árboles simples comparados con un solo cart complejo.

```
# -*- coding: utf-8 -*-
import pandas as pd
from sklearn.preprocessing import StandardScaler
```

```

from sklearn import metrics
from sklearn.model_selection import train_test_split, cross_val_score, StratifiedKFold
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
# Leer los datos, prepararlos y dividirlos en train + test
df = pd.read_csv("Diabetes.csv")
X = df.iloc[:, [0, 1, 2, 3, 4, 5, 6, 7]] # variables independientes
y = df['class'].values # variable dependiente
X = StandardScaler().fit_transform(X) # Normalizar
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=2017)
kf = StratifiedKFold(n_splits=5, random_state=2017, shuffle=True)
# Entrenar un árbol complejo con 5-fold CV
cart = DecisionTreeClassifier(random_state=2017).fit(X_train,y_train)
r = cross_val_score(cart, X_train, y_train, cv=kf)
print("Decision Tree (solitario - Train:", r.mean())
print("Decision Tree (solitario) - Test:", metrics.accuracy_score(cart.predict(X_test), y_test))
# Entrenar 100 árboles para formar un randomforest
n_arboles=100
rf = RandomForestClassifier(n_estimators=n_arboles).fit(X_train, y_train)
r = cross_val_score(rf, X_train, y_train, cv=kf)
print("Random Forest - Train:", r.mean())
print("Random Forest - Test: ", metrics.accuracy_score(rf.predict(X_test), y_test))

```

## ÁRBOLES EXTREMADAMENTE RANDOMIZADOS (eXtraTree)

Este algoritmo es un esfuerzo por introducir más aleatorización al proceso de *bagging*. Las divisiones de los árboles se escogen completamente al azar desde un rango de valores en la muestra de cada división, lo que permite reducir la varianza del modelo final, aunque a costa de aumentar su *bias*.

**EJEMPLO 51:** Añadir al código del ejemplo 46 este, que crea un modelo *ExtraTreesClassifier*.

```

# Modelo ExtraTreesClassifier
from sklearn.ensemble import ExtraTreesClassifier
et = ExtraTreesClassifier(n_estimators=n_arboles).fit(X_train, y_train)
r = cross_val_score(et, X_train, y_train, cv=kf)
print("ExtraTree - Train:", r.mean())
print("ExtraTree - Test: ", metrics.accuracy_score(et.predict(X_test), y_test))

```

## 11.4. MODELOS ENSEMBLE POR VOTACIÓN.

Un modelo compuesto por votación permite combinar modelos de diferente tipo y combinarlos para usar sus respuestas a modo de votación para generar una respuesta. Los modelos base pueden ser de diferentes tipos al contrario de lo que ocurre con el *bagging* o el *boosting*. El mecanismo es similar, se hace la media de los valores en el caso de regresiones o se elige lo que la mayoría si es una clasificación. Además podemos dar pesos a los submodelos para que no todos tengan la misma importancia. A esta variante se la conoce como **stacking** (agregación apilada).

**EJEMPLO 52:** Implementar un modelo ensemble por votación.

```

# -*- coding: utf-8 -*-
import pandas as pd
import numpy as np
np.random.seed(2017) # Hacerlo reproducible
from sklearn import metrics
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.ensemble import BaggingClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
from mlxtend.classifier import EnsembleVoteClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score

```

```

# Leer datos
df = pd.read_csv("Diabetes.csv")
X = df.iloc[:,[0, 1, 2, 3, 4, 5, 6, 7]] # variables independientes
y = df['class'] # variable dependiente
# Dividir datos en train + test y escalar características
X = StandardScaler().fit_transform(X)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=2017)
# Entrenar diferentes modelos para clasificar
RL = LogisticRegression(random_state=2017)
RF = RandomForestClassifier(n_estimators = 100, random_state=2017)
SVM = SVC(random_state=0, probability=True)
KNC = KNeighborsClassifier()
CART = DecisionTreeClassifier()
ABC = AdaBoostClassifier(n_estimators = 100)
BC = BaggingClassifier(n_estimators = 100)
GBC = GradientBoostingClassifier(n_estimators = 100)
clfs = []
print('==== 5-fold validación cruzada: ====\n')
for clf, label in zip([RL, RF, SVM, KNC, CART, ABC, BC, GBC],
                      ['Regresión Logística', 'Random Forest', 'Máquinas de Soporte
Vectorial',
                       'KNeighbors', 'Arbol de Decision', 'Ada Boost', 'Bagging',
                       'Gradient Boosting']):
    scores = cross_val_score(clf, X_train, y_train, cv=5, scoring='accuracy')
    print("Train CV Accuracy: %0.3f (+/- %0.3f) [%s]" % (scores.mean(), scores.std(), label))
    md = clf.fit(X, y)
    clfs.append(md)
    print("Test Accuracy: %0.3f " % (metrics.accuracy_score(clf.predict(X_test), y_test)))
# Ensemble Voting
clfs = []
print('\n==== 5-fold CV: ====\n')
ECH = EnsembleVoteClassifier(clfs=[RL, RF, GBC, BC], voting='hard')
ECS = EnsembleVoteClassifier(clfs=[RL, RF, GBC, BC], voting='soft', weights= [1, 1, 1, 1])
for clf, label in zip([ECH, ECS], ['Ensemble Hard Voting', 'Ensemble Soft Voting']):
    scores = cross_val_score(clf, X_train, y_train, cv=5, scoring='accuracy')
    print("Train CV Accuracy: %0.3f (+/- %0.3f) [%s]" % (scores.mean(), scores.std(), label))
    md = clf.fit(X, y)
    clfs.append(md)
    print("Test Accuracy: %0.2f " % (metrics.accuracy_score(clf.predict(X_test), y_test)))

```

### VOTACIONES ESTRICAS (HARD) O FLEXIBLES (SOFT)

Cuando la votación la gana la mayoría se habla de una votación estricta (*hard voting*). Si el sistema devuelve la operación *argmax* aplicada a la suma de las probabilidades predichas se conoce como votación flexible (*soft voting*).

Classifier	Class 1	Class 2	Class 3	.	.	Class n
Classifier 1	w1 * 0.3	w1 * 0.1	w1 * 0.6	.	.	w1 * 0.1
Classifier 2	w2 * 0.4	w2 * 0.3	w2 * 0.3	.	.	w2 * 0.3
Classifier 3	w3 * 0.5	w3 * 0.4	w3 * 0.2	.	.	w3 * 0.3
Weighted average	0.4	0.12	0.37	.	.	0.23

Figura 83: Ejemplo de ejecución de un clasificador ensemble por votación.

El parámetro '**weights**' puede utilizarse para asignar diferentes pesos o importancia a cada clasificador base. La probabilidad de cada clasificador base se multiplica por su peso y a estos valores se les saca la media, es decir, es una media ponderada. Asumiendo que todos los pesos de los clasificadores es 1, la clase predicha en la figura 81 sería 1.

## 11.5. STACKING.

Wolpert David H presentó en 1992 el concepto de generalización apilada más conocida como '**stacking'** en su publicación journal *Neural Networks*. En esta técnica, inicialmente entrena diferentes modelos base sobre unos datos de entrenamiento y test. La idea es mezclar los diferentes modelos (kNN, bagging, boosting, etc.) para que puedan aprender bien alguna parte del problema. En el nivel 1, usa los valores predecidos por los modelos base como características de entrenamiento de un modelo conocido como **meta-modelo**, combinando lo que han aprendido los modelos individuales mejora la accuracy. Esto es un modelo de *stacking* de nivel 1, y de manera similar, puedes apilar más niveles de modelos cada vez más complejos.

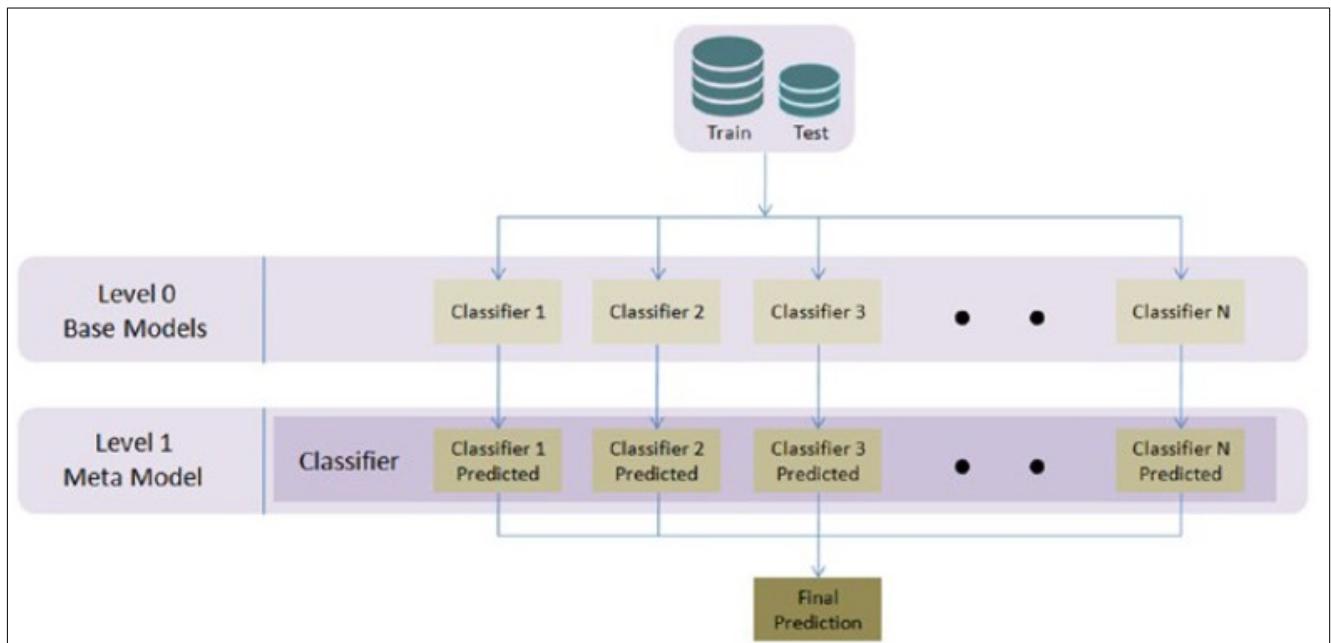


Figura 82: Un clasificador que usa stacking.

### EJEMPLO 53: Clasificador que utiliza stacking

```

# -*- coding: utf-8 -*-
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split, StratifiedKFold, cross_val_score
from sklearn.metrics import accuracy_score
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
np.random.seed(2017) # semilla para hacer repetible un resultado
# preparar datos
df = pd.read_csv("Diabetes.csv")
X = df.iloc[:,[0,1,2,3,4,5,6,7]] # variables independientes
y = df['class'].values # variable dependiente
X = StandardScaler().fit_transform(X) # Normalizar
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=2017)
kf = StratifiedKFold(n_splits=5, random_state=2017, shuffle=True)
n_arboles = 10
clfs = [KNeighborsClassifier(),
        RandomForestClassifier(n_estimators=n_arboles, random_state=2017),
        GradientBoostingClassifier(n_estimators=n_arboles, random_state=2017)]
ds_blend_train = np.zeros((X_train.shape[0], len(clfs)))
ds_blend_test = np.zeros((X_test.shape[0], len(clfs)))
print('5-fold CV:\n')
for i, clf in enumerate(clfs):
    scores = cross_val_score(clf, X_train, y_train, cv=kf, scoring='accuracy')
    print("#### Modelo base %0.0f %% ##### % (i, clfs[i]))
```

```

print(" Train CV Accuracy: %0.3f (+/- %0.3f)" % (scores.mean(), scores.std()))
clf.fit(X_train, y_train)
print(" Train Accuracy: %0.3f " % (accuracy_score(clf.predict(X_train), y_train)))
ds_blend_train[:,i] = clf.predict_proba(X_train)[:, 1]
ds_blend_test[:,i] = clf.predict_proba(X_test)[:, 1]
print(" Test Accuracy: %0.3f " % (accuracy_score(clf.predict(X_test), y_test)))
print("##### Meta Modelo #####")
clf = LogisticRegression()
scores = cross_val_score(clf, ds_blend_train, y_train, cv=kf, scoring='accuracy')
clf.fit(ds_blend_train, y_train)
print("Train CV Accuracy: %0.3f (+/- %0.3f)" % (scores.mean(), scores.std()))
print("Train Accuracy: %0.3f " % (accuracy_score(clf.predict(ds_blend_train), y_train)))
print("Test Accuracy: %0.3f " % (accuracy_score(clf.predict(ds_blend_test), y_test)))

```

## 12. EJERCICIOS.

### EJERCICIO 1: Responde a estas preguntas.

1. ¿Qué algoritmo de regresión lineal usarías si tuvieras datos de entrenamiento de miles de millones de predictoras?
2. Imagina que las características de tus datos de entrenamiento tienen escalas de valores muy diferentes. ¿Qué algoritmos puedes usar para remediarlo y en qué consisten?
3. ¿Puede acabar el algoritmo de descenso por gradiente en un mínimo local alejado de la solución óptima cuando estás entrenando un modelo de regresión logística?
4. ¿Todos los algoritmos de descenso por gradiente conducen al mismo resultado si se los deja funcionar el suficiente tiempo?
5. Si estás usando descenso por gradiente de tipo *Batch* y dibujas el error de validación en cada época y observas que a partir de cierto momento el error comienza a subir ¿Qué puede estar pasando? ¿Cómo puedes solucionarlo?
6. ¿Es buena idea parar de ejecutar el algoritmo de descenso por gradiente de tipo *Mini-Batch* en cuanto detectes que el error de validación experimenta una subida?
7. ¿Qué algoritmo de descenso por gradiente (de los comentados en la unidad) alcanza más rápidamente la vecindad de la solución óptima? ¿Cómo puedes hacer que los otros converjan de forma similar?
8. Imagina que estás usando regresión polinomial. Dibujas las curvas de aprendizaje y observas que hay un importante hueco entre el error de entrenamiento y el de validación. ¿Qué está ocurriendo? De qué tres formas lo solucionarías?
9. Imagina que estás usando regresión regularizada con *Ridge* y observas que el error de entrenamiento y el de validación permanecen igual de alejados y ambos cada vez más alejados. ¿Se puede decir que tiene alto bias o alta varianza? ¿Deberías incrementar el parámetro de regularización o reducirlo?
10. ¿Porqué podría interesarte utilizar?
  - Regresión *Ridge* en vez de Regresión Lineal (es decir, sin regularización).
  - Regresión *Lasso* en vez de *Ridge*.
  - *Elastic Net* en vez de *Lasso*.
11. Imagina que quieres clasificar imágenes como interiores/exteriores, es decir, realizadas dentro de edificios o fuera de edificios y también si se han realizado de día o por la noche. ¿Deberías implementar dos clasificadores de *Regresión Logística* o un solo clasificador de tipo *Softmax*?

**EJERCICIO 2:** Mira el video [SAA U02 Elegir el mejor modelo.mp4](#) y responde a estas preguntas:

- a) El problema que plantea el vídeo ¿Es de aprendizaje supervisado, no supervisado, reforzado o semisupervisado?
  - b) ¿Qué métrica utiliza para medir el desempeño de los modelos?
  - c) Al usar una red neuronal además de otros modelos ¿Es necesario medir el desempeño de todos? ¿O podemos ahorrarnos este proceso y apostar directamente por la red neuronal?
  - d) ¿Qué es mejor? Que la métrica sea pequeña o que sea grande.
  - e) Al final, ¿Cuál es el mejor modelo de los usados en el vídeo?
  - f) Si hubiese utilizado un problema de clasificación, ¿Podría usar la misma métrica?

**EJERCICIO 3:** Mira el vídeo [SAA U02 La matriz de confusión.mp4](#) y responde a los siguientes problemas:

- a) ¿Porqué tener una buena accuracy no es una medida definitiva para un clasificador?
  - b) ¿Qué es o como se calcula la accuracy?
  - c) Si tenemos 6 ejemplos que queremos clasificar en Animal y No animal y un clasificador realiza esta clasificación, define la matriz de confusión y calcula el accuracy:

		Predicted	
		Animal	Not animal
lechuza		manzana	
chiguagua		muffin	
shetter		mopa	
Actual			
Animal		  	
Not animal			 

d) ¿Con estos datos es una buena métrica?

**EJERCICIO 4.** Mira el vídeo [SAA U02 precisión recall y f score.mp4](#) y completa los siguientes problemas:

- a) Calcula todas las métricas de la matriz de confusión que hiciste en el apartado c) del ejercicio 3.
  - b) ¿Qué dos métricas son incompatibles y si sube una baja la otra?
  - c) Si clasificas setas en comestibles y venenosas y hay muchos más ejemplos de comestibles que de venenosas, ¿Qué usarías como métrica, el recall o la precisión?
  - d) Si prefieres sobre todo tener pocos FP ¿Qué prefieres aumentar: el recall, la sensitividad, la accuracy, la precisión, el f1-score, la precisión negativa?

**EJERCICIO 5.** Mira el vídeo [SAA\\_U02\\_ROC\\_y\\_AUC.mp4](#) y completa los siguientes apartados:

- a) La curva ROC en el eje X usa FPR y en el eje Y?
  - b) ¿Cómo es la ROC de un clasificador perfecto?
  - c) ¿Cómo es la ROC de un clasificador aleatorio?
  - d) ¿La curva ROC sirve para comparar varios clasificadores?
  - e) ¿Es bueno que la AUC de una curva ROC sea cercano a 1?
  - f) Un mal clasificador ¿Entre qué valores AUC estará?

**EJERCICIO 6.** Mira el vídeo [SAA U02 k fold cross validation.mp4](#) y completa los siguientes apartados:

- a) ¿Qué valor de k se puede utilizar?

- b) ¿En caso de tener unos 200 datos el valor adecuado de k podría ser?
- c) ¿La ventaja de k-fold CV sobre particionar en train+validación+test?
- d) ¿La desventaja de k-fold CV sobre particionar en train+validación+test?

**EJERCICIO 7.** Mira el vídeo [SAA\\_U02\\_configurar\\_hiperparámetros.mp4](#) y completa los siguientes apartados:

- a) ¿Qué dos métodos se pueden utilizar?
- b) ¿Ventaja del enfoque grid-search?
- c) ¿Desventaja del método grid-search?

**EJERCICIO 7:** En el ejemplo 2

**EJERCICIO 8:** En el [ejemplo 2](#) de la página 17 se ha construido un modelo de regresión lineal que intenta explicar la nota obtenida en un control (target) a partir de las horas dedicadas a estudiarlo (predictora). Se ha calculado el  $R^2$  con datos de test dando un valor de 0.97. Responde a estas preguntas:

- a) Parece tener sentido usar este modelo a la vista del valor del coeficiente de determinación.
- b) Calcula el  $SSR$  usando su fórmula (el  $SSR$  que aparece calculado en el ejemplo nosotros lo llamamos  $SSE$ ).
- c) Comprueba que se cumple que  $SST = SSE + SSR$  (los valores ya los tienes).
- d) Calcula  $SSR$  usando la fórmula  $SST = SSE + SSR$ . ¿Coincide con el valor calculado?

**EJERCICIO 9:** En el ejemplo 17 ([enlace](#)) se calcula el índice de inflación de la varianza usando dataset 'BIM.csv'. Calcula ahora su matriz de correlaciones para el dataset y comprueba si visualmente también se detectan las mismas columnas que pueden causar colinealidad en un modelo de regresión. Muestra el gráfico e indica las columnas correlacionadas.

**EJERCICIO 10:** En el ejemplo 18 ([enlace](#)) se calculan los  $VIF$  del ejemplo 17 y se elimina una de las columnas que puede causar colinealidad. Modifica el código para que siga descartando columnas mientras alguna pase el límite aconsejable de  $VIF$ . Muestra una captura de su ejecución.

**EJERCICIO 11.** Ejecuta el [ejemplo 21](#) y el [ejemplo 22](#) aplicado al modelo del ejercicio anterior. Indica qué datos son outliers. Ahora aplica alguno de los algoritmos básicos (o el basado en z-scores o el basado en IQR) y comprueba si detectan los mismos outliers.

**EJERCICIO 12:** Modifica el [ejemplo 28](#) para calcular también un modelo lineal a los mismos datos y muestra el  $R^2$  del modelo lineal y del modelo polinómico con los datos de entrenamiento.

**EJERCICIO 13:** Repite el ejercicio anterior, pero ahora divide los datos en train y test. Saca los modelos y calcula el  $R^2$  en los datos de test. ¿Cuál es mejor?

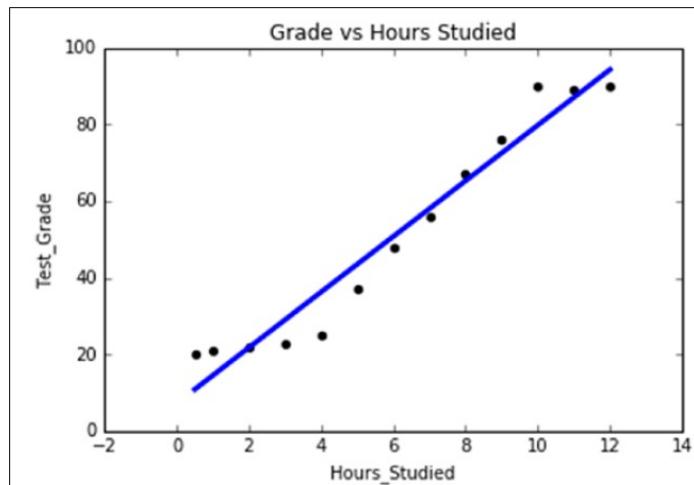
**EJERCICIO 14:** Tenemos los datos de 'Grade\_Set\_2.csv', considera otro conjunto de datos de notas de estudiantes.

```
# Carga datos
df = pd.read_csv('Grade_Set_2.csv')
print df
# Gráfico
df.plot(kind='scatter', x='Hours_Studied', y='Test_Grade', title='Grade vs
Hours Studied')
# check the correlation between variables
print("Correlation Matrix: ")
df.corr()
# Create linear regression object
lr = lm.LinearRegression()
x= df.Hours_Studied[:, np.newaxis] # independent variable
y= df.Test_Grade # dependent variable
# Train the model using the training sets
```

```

lr.fit(x, y)
# plotting fitted line
plt.scatter(x, y, color='black')
plt.plot(x, lr.predict(x), color='blue', linewidth=3)
plt.title('Grade vs Hours Studied')
plt.ylabel('Test_Grade')
plt.xlabel('Hours_Studied')
print "R Squared: ", r2_score(y, lr.predict(x))

```

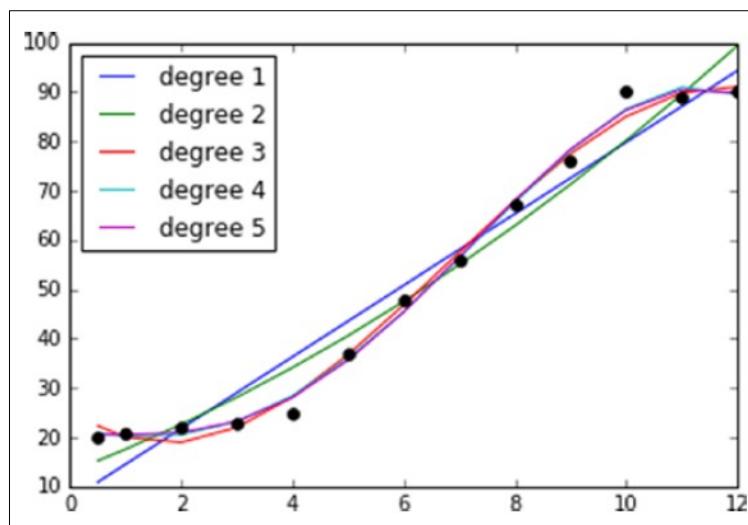


El análisis de correlación muestra el 97% de relación positiva entre las horas estudiadas y la nota obtenida y el 95% ( $R^2$ ) de variación puede ser explicada por las horas estudiadas. Observa que menos de 4 horas de media estudiadas da una puntuación menor de 30 y cada 9 horas añade a la nota un nivel. Esto puede modelarse también con un regresor polinómico.

```

lr = lm.LinearRegression()
x= df.Hours_Studied    # independent variable
y= df.Test_Grade        # dependent variable
for deg in [1, 2, 3, 4, 5]:
    lr.fit(np.vander(x, deg + 1), y)      # función vander() devuelve potencias del vector
    y_lr = lr.predict(np.vander(x, deg + 1))
    plt.plot(x, y_lr, label='degree ' + str(deg));
    plt.legend(loc=2);
    print r2_score(y, y_lr)
    plt.plot(x, y, 'ok')

```



**EJERCICIO 15:** Calcula la precisión y el recall del ejemplo de la matriz de confusión de la página 71.

**EJERCICIO 16:** Ejecuta el ejemplo 40. ¿Cuál de los dos modelos es mejor? (justifica tu respuesta) ¿Es significativa tu respuesta? (¿Está respaldada por alguna prueba más que por tu intuición?)

**EJERCICIO 17:** Ejecuta el ejemplo 40. ¿Cuál de los dos modelos es mejor? (justifica tu respuesta) ¿Es significativa tu respuesta? (¿Está respaldada por alguna prueba más que por tu intuición? )

**EJERCICIO 12:** Ejecuta el ejemplo 40. ¿Cuál de los dos modelos es mejor? (justifica tu respuesta) ¿Es significativa tu respuesta? (¿Está respaldada por alguna prueba más que por tu intuición? )

**EJERCICIO 18.** Observa el código del [ejemplo 47](#) y responde y realiza las siguientes actividades:

- a) Ejecuta el código y toma captura del resultado de la ejecución.
- b) ¿Qué 5 modelos base tienen la mejor accuracy? Los ordenas de mejor a peor por su nombre.
- c) Modifica el meta-modelo hard-voting pasándole esos 5 modelos base.
- d) Modifica el meta-modelo soft-voting pasándole esos 5 modelos base, y además le pasas los pesos modificados de manera que el peso del modelo i-ésimo sea su accuracy dividido por la suma de todas las accuracies.

**EJERCICIO 19.** Observa el código del [ejemplo 52](#) y responde y realiza las siguientes actividades:

- a) Entrena un modelo lineal con regularización que te permita conocer al menos dos predictoras que se puedan anular porque no aparten información.
- b) ¿Qué otros métodos conoces para saber qué predictoras son importantes y cuales no? No incluyas en la respuesta PCA que aún no lo hemos visto aunque lo hayamos usado.
- c) Modifica el código del ejemplo para definir los modelos base con una variable que si aporte información y otra que no.
- d) Añade el código necesario para visualizar las regiones de decisión del meta-modelo basado en *stacking* si usamos esas dos variables.