

UNIDAD 3

SISTEMAS DE APRENDIZAJE AUTOMÁTICO

IES SERRA PERENXISA (Torrent) Valencia

APRENDIZAJE SUPERVISADO 2

Profesor: José Rosa Rodríguez
Curso 2024-2025

OBJETIVOS:

- Identificar las variantes y ventajas de algoritmos naive Bayes.
- Identificar y manejar SVM y sus variedades.
- Identificar los árboles de decisión y conocer sus algoritmos ID3, ID5, CART.
- Ventajas de métodos ensamble
- Medir el desempeño y conocer métricas de estos modelos.
- Aplicar los algoritmos a resolver algún problema planteado.
- Comparar diferentes algoritmos para resolver un problema de manera metodológica y con un enfoque científico.

BIBLIOGRAFÍA:

- Documentación oficial de Scikit-learn.
- Machine Learning and Deep learning with python, scikit-learn and TensorFlow (Sebastian Raschka & Vahid Mirjalili), 3Ed. 2023.
- Throuthful Machine Learning. 2017.
- Mark Stamp. “A revealing introduction to hidden Markov models”. Tech. rep. Department of Computer Science, San Jose State University, 2018.
url: <http://www.cs.sjsu.edu/~stamp/RUA/HMM.pdf>.
- Tutorial de HMM-learn: <https://hmmlearn.readthedocs.io/en/latest/tutorial.html#available-models>
- Departamento de computación e inteligencia artificial de la Escuela Técnica Superior de Informática de la Universidad Politécnica de Madrid:
https://dcain.etsin.upm.es/~carlos/bookAA/04.04_ModelosMarkovOcultos.html

TABLA DE CONTENIDOS:

Sumario

1. ALGORITMOS NAIVE BAYES.....	4
1.1. REPASO DE PROBABILIDAD.....	4
1.2. ALGORITMOS DE APRENDIZAJE NAIVE BAYES.....	11
1.3. NAIVE BAYES GAUSIANO.....	15
1.4. NAIVE BAYES MULTINOMIAL.....	19
2. MÁQUINAS DE VECTORES SOPORTE (SVM).....	22
2.1. SVM PARA CLASIFICACIÓN BINARIA LINEAL.....	22
2.2. SVM PARA CLASIFICACIÓN NO LINEAL.....	24
2.3. SVM PARA REGRESIÓN (SVR).....	35
2.4. SVM PARA DETECCIÓN DE OUTLIERS Y DE NOVELTIES.....	38
2.5. ALGORITMO DE ENTRENAMIENTO DE SVM.....	41
2.6. SVM Online.....	46
2.7. PREPROCESAMIENTO Y TUNING DE PARÁMETROS DE SVM.....	46
3. ÁRBOLES DE DECISIÓN (DECISIÓN TREES).....	49
3.1. ALGORITMOS DE ENTRENAMIENTO.....	52
3.2. MÉTRICAS PARA MEDIR LO BUENA QUE ES UNA DIVISIÓN.....	55
3.3. ALGORITMO CART.....	58
3.4. ENTRENAR, PREDICIR Y VISUALIZAR CON SCIKIT-LEARN.....	61
3.4. ÁRBOLES DE DECISIÓN PARA REGRESIÓN.....	67
3.5. INESTABILIDAD.....	69
4. MÉTODOS DE ENSAMBLAJE (ENSEMBLES).....	70
4.1. VOTING.....	70
4.2. BAGGING/BOOTSTRAP Y PASTING.....	72
4.3. RANDOM FOREST.....	74
4.4. BOOSTING.....	75
5. MODELOS OCULTOS DE MARKOV (HMM).....	79
5.1. ALGORITMOS USADOS EN HMM.....	84
6. EJERCICIOS.....	90

1. ALGORITMOS NAIVE BAYES.

Los modelos **Naive Bayes** son un grupo de algoritmos extremadamente rápidos y simples que se pueden utilizar para clasificación y regresión. Son adecuados para datos de muy alta dimensionalidad. Como son rápidos y tienen pocos hiperparámetros han sido utilizados con éxito en varios problemas de clasificación.

1.1. REPASO DE PROBABILIDAD.

La probabilidad es una métrica usada en matemáticas para medir lo esperable que es obtener un determinado resultado al realizar un experimento aleatorio. Es una medida basada en proporciones que puede tomar valores entre 0 (resultado imposible) y 1 (resultado seguro). Vamos a repasar algunos conceptos:

La probabilidad de que ocurra un **evento A** se escribe como **P(A)** y se calcula como la proporción de **cantidad de resultados favorables dividido entre el cantidad de resultados posibles** que puede arrojar el experimento. Por ejemplo si lanzo un dado, la probabilidad de obtener un 3 es:

$$A = \text{Obtener un } 3 \rightarrow P(A) = \frac{\text{casos favorables}}{\text{casos posibles}} = \frac{[3]}{[1, 2, 3, 4, 5, 6]} = \frac{1 \text{ caso}}{6 \text{ casos}} = 1/6 = 0.166$$

$$A = \text{Obtener par} \rightarrow P(A) = \frac{\text{casos favorables}}{\text{casos posibles}} = \frac{[2, 4, 6]}{[1, 2, 3, 4, 5, 6]} = \frac{3 \text{ casos}}{6 \text{ casos}} = 3/6 = 0.5$$

$$A = \text{Obtener } > 4 \rightarrow P(A) = \frac{\text{casos favorables}}{\text{casos posibles}} = \frac{[5, 6]}{[1, 2, 3, 4, 5, 6]} = \frac{2 \text{ casos}}{6 \text{ casos}} = 2/6 = 0.333$$

El resultado opuesto al resultado A se escribe como **\bar{A}** (NO A) tendrá probabilidad **$P(\bar{A}) = 1 - P(A)$** . Ejemplo:

Probabilidad de obtener un número menor o igual a 4 al lanzar un dado es el resultado opuesto de sacar un número mayor de 4. Si $P(>4) = 0.333$, entonces $P(<=4) = 1 - 0.333 = 0.666$. Hagamos la comprobación:

$$A = \text{Obtener } <= 4 \rightarrow P(A) = \frac{\text{casos favorables}}{\text{casos posibles}} = \frac{[1, 2, 3, 4]}{[1, 2, 3, 4, 5, 6]} = \frac{4 \text{ casos}}{6 \text{ casos}} = 4/6 = 0.666 = 1 - 0.333$$

Dados dos sucesos A1 y A2, la probabilidad de **A1 OR A2** es **$P(A1 \text{ OR } A2) = P(A1 \text{ UNION } A2) = P(A1 \cup A2)$** . Si los sucesos son independientes (no tienen resultados comunes) **$P(A1 \cup A2) = P(A1) + P(A2)$** . Ejemplo:

$$A1 = \text{un } 2 \rightarrow P(A1) = 1/6$$

$$A2 = \text{un } 6 \rightarrow P(A2) = 1/6$$

$$P(A1 \text{ OR } A2) = 1/6 + 1/6 = \frac{[2]}{[1, 2, 3, 4, 5, 6]} + \frac{[6]}{[1, 2, 3, 4, 5, 6]} = \frac{[2, 6]}{[1, 2, 3, 4, 5, 6]} = 2/6$$

La probabilidad de que ocurran dos sucesos A1 y A2 al mismo tiempo se expresa como **A1 AND A2** y la probabilidad es **$P(A1 \text{ INTERSECCIÓN } A2) = P(A1 \cap A2)$** (porque la probabilidad es una métrica que se basa en la cantidad de elementos). Ejemplos:

$$A1 = \text{un } 2 \rightarrow P(A1) = 1/6$$

$$A2 = \text{un } 6 \rightarrow P(A2) = 1/6$$

$$P(A1 \text{ AND } A2) = \frac{[2]}{[1, 2, 3, 4, 5, 6]} \text{ Y } \frac{[6]}{[1, 2, 3, 4, 5, 6]} = \frac{[\text{nada en común} = 0]}{[1, 2, 3, 4, 5, 6]} = 0/6 = 0$$

$$A1 = \text{un } 2 \rightarrow P(A1) = 1/6 [2]$$

$$A2 = \text{par} \rightarrow P(A2) = 3/6 [2, 4, 6]$$

$$A1 \text{ INTERSECCIÓN } A2 = [2] \text{ INTERSECCIÓN } [2, 4, 6] = [2]$$

$$P(A1 \text{ AND } A2) = \frac{[2] \cap [2, 4, 6]}{[1, 2, 3, 4, 5, 6]} = \frac{[2]}{[1, 2, 3, 4, 5, 6]} = \frac{1}{[1, 2, 3, 4, 5, 6]} = 1/6$$

Si dos sucesos A1 y A2 no tienen su intersección vacía, se dice que no son independientes, en ese caso **$P(A1 \text{ OR } A2) = P(A1) + P(A2) - P(A1 \cap A2)$** . Ejemplo:

$$\begin{aligned} A1 &= \text{un } 2 \rightarrow P(A1) = 1/6 [2] \\ A2 &= \text{par} \rightarrow P(A2) = 3/6 [2, 4, 6] \\ A1 \cap A2 &= [2] \cap [2, 4, 6] = [2] \\ A1 \cup A2 &= [2] \cup [2, 4, 6] = [2, 4, 6] \end{aligned}$$

$$P(A1 \text{ OR } A2) = P(A1 \cup A2) = [2, 4, 6] / [1, 2, 3, 4, 5, 6] = 3/6 = 0.5$$

$$\text{No son independientes: } P(A1 \cup A2) = P(A1) + P(A2) - P(A1 \cap A2) = 1/6 + 3/6 - 1/6 = 0.5$$

La probabilidad del suceso A condicionada a que sepamos que ha ocurrido el suceso B se escribe como $P(A/B)$ y se lee "probabilidad de A condicionada a B" es decir, cuando sabemos algo más, tenemos más información. La fórmula sería:

$$P(A/B) = \frac{P(A \cap B)}{P(B)} \quad \text{Nota: abajo lo que ya ha ocurrido, por tanto } P(B) \neq 0$$

Del mismo modo:

$$P(B/A) = \frac{P(A \cap B)}{P(A)} \quad \text{Nota: abajo lo que ya ha ocurrido, por tanto } P(A) \neq 0$$

De ambas podemos deducir que:

$$\begin{aligned} P(A/B) &= P(A \cap B) / P(B) \rightarrow P(A \cap B) = P(A/B) P(B) \\ P(B/A) &= P(A \cap B) / P(A) \rightarrow P(A \cap B) = P(B/A) P(A) \end{aligned}$$

De ambas deducimos un importante teorema denominado el teorema de Bayes:

$$\left. \begin{aligned} P(A/B) &= \frac{P(A \cap B)}{P(B)} \\ P(A \cap B) &= P(B/A) P(A) \end{aligned} \right\} P(A/B) = \frac{P(B/A) P(A)}{P(B)}$$

Dados los eventos **A** y **B** (un evento puede ser que llueva mañana, que salga un rey en una mano de un juego de cartas o que una persona tenga cáncer). El teorema de Bayes dice que la probabilidad de que ocurra el evento **A** siendo cierto el hecho de que ha ocurrido **B** (probabilidad de **A** condicionada a **B**) se calcula como:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Donde $P(B|A)$ es la probabilidad de observar el suceso **B** habiendo ocurrido **A**, y $P(A)$ y $P(B)$ es la probabilidad e que ocurra **A** y **B** respectivamente. Igual es demasiado abstracto, así que vamos a un ejemplo concreto o si lo prefieres te dejo enlace a un vídeo: [enlace](#).

EJEMPLO 1: Tenemos dos monedas: una está trucada y al lanzarla el 90% de las veces sale cara y el 10% sale cruz. La otra moneda no está trucada y aproximadamente la probabilidad tanto de cara como de cruz es del 50%. Si cogemos al azar una de las dos monedas, ¿Cuál es la probabilidad de que sea la trucada si sacamos una cara en el lanzamiento?

Llamamos **T** al evento de usar la moneda trucada y **C** al evento de obtener una cara. La probabilidad de que sea la moneda trucada cuando sale una cara se puede calcular como:

$$P(T|C) = \frac{P(C|T) P(T)}{P(C)}$$

$P(C|T)$ es 90%, $P(T)$ es 0.5 al elegir aleatoriamente una de las dos monedas juntas. Sin embargo no podemos deducir directamente la probabilidad de obtener cara ya que podemos elegir la trucada **T** o la no trucada **NT**. Sin embargo tenemos:

$$P(T|C) = \frac{P(C|T) P(T)}{P(C)} = \frac{P(C|T) P(T)}{P(C|T)P(T) + P(C|NT)P(NT)} = \frac{0.9 * 0.5}{0.9 * 0.5 + 0.5 * 0.5} = 0.64$$

EJEMPLO 2: Usar el teorema de Bayes para encontrar pedidos fraudulentos. Imagina que diriges una tienda que sirve pedidos online y últimamente recibes pedidos fraudulentos. Estimas que alrededor del 10% de todos los pedidos son fraudulentos. Quieres minimizar las pérdidas pero cada mes recibes unos 1000 pedidos y si tienes que revisarlos de uno en uno vas a perder más dinero del que pierdes por el propio fraude, ya que revisarlos supone un coste de 15,00€ por cliente, unas 200 horas y 3000,00€ al año.

Observamos también que con frecuencia los pedidos fraudulentos usan tarjetas regalo y varios códigos promocionales.

Las probabilidades condicionales nos permiten asociar la probabilidad de un evento con la de otro. Podemos hablar de la probabilidad de un pedido fraudulento condicionado a que use una tarjeta regalo. Dados dos eventos **A** y **B** la probabilidad condicional de **A** condicionado al **B** se expresa como $P(A|B)$ y se calcula como (el símbolo \cap indica la intersección, la probabilidad de que se den los dos):

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

Si representamos cada evento con un conjunto de todos sus posibles valores, y definimos dos eventos, **A** \cap **B** se llama la intersección de los dos eventos y se corresponde con la función lógica **AND** que en Python es el símbolo **and**:

```
a = [1, 3, 5]          # Evento de que al tirar un dado salga un valor impar
b = [4, 5, 6]          # Evento de que al tirar un dado salga un número mayor de 3
set(a) & set(b)        # Devuelve el set [5], los elementos comunes
```

A \cup **B** se llama la función **OR** de **A** y **B** u y sería la unión de dos eventos. En Python se usa la barra | para expresarla:

```
set(a) | set(b) # Devuelve la unión de los eventos a y b: [1, 3, 4, 5, 6]
```

Finalmente, recordamos que las probabilidades miden cuantos casos tiene un evento entre todos los posibles casos que existen, es decir, expresa cantidades. Para calcularla en Python debemos contar los valores de cada evento sabiendo que los casos posibles al lanzar un dado son 6 (un número del 1 al 6):

```
a = set([1, 3, 5])
b = set([4, 5, 6])
p_a = len(a) / 6           #=> 0.5
p_b = len(b) / 6           #=> 0.5
p_a_Y_b = len(a & b) / 6   #=> 0.166
p_a_O_b = len(a | b) / 6   #=> 0.833
p_a_condi_b = p_a_Y_b / p_b #=> 0.33
print("P(a)=", p_a, "P(b)=", p_b, "P(a Y b)", p_a_Y_b, "P(a O b)=", p_a_O_b)
print("P(a | b)=", p_a_condi_b) #=> 0.33
```

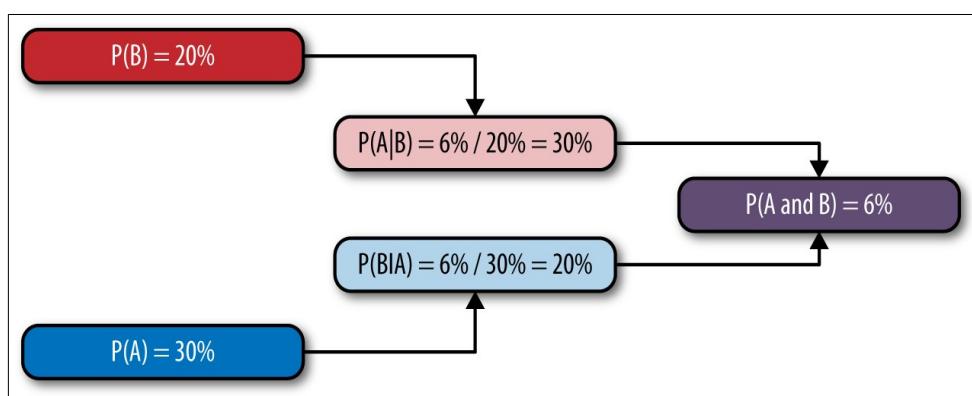


Figura 1. Cómo se calcula la probabilidad condicional.

Esta definición de probabilidad condicional básicamente indica que la probabilidad de que ocurra el evento **A** sabiendo que ha ocurrido el evento **B** es la probabilidad de que ocurran tanto **A** como **B** dividida entre la probabilidad de que ocurra **B**.

La figura muestra como se relacionan $P(A|B)$ con $P(A \text{ AND } B)$ y $P(B)$. En nuestro ejemplo donde queremos medir la probabilidad de pedido fraudulento condicionado a que use una tarjeta de descuento:

$$P(\text{fraude}|\text{tarjeta}) = \frac{P(\text{fraude} \text{ and } \text{tarjeta})}{P(\text{tarjeta})}$$

Esto nos ayuda a resolver el problema si conocemos la probabilidad de fraude y tarjeta. Pero la probabilidad de las dos cosas juntas es difícil de calcular y aquí nos bloqueamos. En estos casos es donde podemos ayudarnos del *teorema de Bayes*.

PROBABILIDAD INVERSA CONDICIONAL (es decir, TEOREMA DE BAYES)

En 1700, el reverendo *Thomas Bayes* descubrió su teorema y *Pierre-Simon Laplace* lo extendió para poder utilizarlo tal y como lo usamos hoy día:

$$P(B|A) = \frac{P(A|B) P(B)}{P(A)}$$

Podemos usar esto en nuestro detector de fraudes porque podemos calcular la probabilidad de la intersección a partir de otra información:

$$P(\text{fraude}|\text{tarjeta}) = \frac{P(\text{tarjeta} \mid \text{fraude}) P(\text{fraude})}{P(\text{tarjeta})}$$

Recordemos que la probabilidad de fraude era del 10%. Digamos que la probabilidad de usar tarjeta regalo es del 10% y basándonos en nuestras observaciones la probabilidad de fraude al usar la tarjeta es del 60%. Así que ¿Cuál es la probabilidad de que un pedido sea fraudulento si usa una tarjeta de regalo?

$$P(\text{fraude}|\text{tarjeta}) = \frac{60\% \cdot 10\%}{10\%} = 60\%$$

La ventaja de esto que que medir los pedidos fraudulentos se reduce mucho porque solamente debes comprobar los pedidos que usen tarjeta. Como el número de pedidos es 1000 y 100 son fraudulentos, solo tenemos que mirar 60 de ellos. De los 900 pedidos restantes, 90 usarán tarjeta regalo, así que en total hay que mirar $60 + 90 = 150$ pedidos.

Hemos reducido el trabajo de investigar 1000 pedidos a investigar solo 150 (un 15% del total, ¿Cómo hacerlo? Porque hemos contemplado que investigar en los que tengan tarjetas regalo, pero y otros que tengan códigos promocionales, o bien otras características?)

Necesitamos solucionar el problema de encontrar $P(A \mid B, C) = ?$ es decir, la probabilidad de un evento, condicionado a que ocurran uno o más de otros eventos. Necesitamos para hacerlo algo más de información, lo que se conoce como **regla de la cadena**.

REGLA DE LA CADENA

Si volvemos a fijarnos en el tipo de probabilidades, puedes observar que la probabilidad de que **A** y **B** ocurran es la probabilidad de que ocurra **B** multiplicada por la probabilidad de que ocurra **A**. Matemáticamente se puede decir que $P(A \text{ and } B) = P(B|A)P(A)$. Esto es cierto siempre que los eventos no sean mutuamente excluyentes. Usando esta probabilidad conjunta surge la regla de la cadena. Las probabilidades conjuntas de varios eventos es la probabilidad de que todos ellos ocurran. Lo expresamos con el operador intersección \cap . El caso genérico de la regla de la cadena es:

$$P(A_1, A_2, \dots, A_n) = P(A_1) P(A_2|A_1) P(A_3|A_1, A_2) \dots P(A_n|A_1, A_2, \dots, A_{n-1})$$

Esta versión expandida puede utilizarse para resolver nuestro problema de detectar pedidos fraudulentos si tenemos suficientes ejemplos de los que aprender usando Bayes para estimar las

probabilidades. Pero todavía hay un problema: su cálculo es complejo a menos que hagamos una suposición que lo simplifique.

LA PALABRA NAIVE DEL RAZONAMIENTO BAYESIANO

La regla de la cadena se utiliza para resolver problemas potencialmente inclusivos, pero no tenemos la habilidad para calcular las probabilidades que la definen. Por ejemplo si tenemos promociones además de tarjetas regalo en nuestros datos de cada pedido, tendríamos que calcular:

$$P(\text{fraude} | \text{tarjeta, promo}) = \frac{P(\text{tarjeta, promo} | \text{fraude}) P(\text{fraude})}{P(\text{tarjeta, promo})}$$

Ignoramos el denominador (porque no depende de si hay o no fraude) y nos centramos en calcular el numerador. Aplicando la regla de la cadena es equivalente a:

$$\begin{aligned} P(\text{fraude/tarjeta, promo}) &= P(\text{fraude}) P(\text{tarjeta, promo} | \text{fraude}) = \\ &P(\text{fraude}) P(\text{tarjeta/fraude}) P(\text{promo/fraude, tarjeta}) \end{aligned}$$

El problema que tenemos es que no podemos calcular fácilmente la probabilidad de una promoción dado el fraude y una tarjeta de regalo. Aunque está bien expresado matemáticamente, en la realidad es difícil de medir, especialmente a medida que tengamos muchas características. **Y** aquí es donde entra en juego el término **naive**, porque hacemos una suposición, una simplificación en relación a que cada característica no tiene ninguna interacción ni relación con otra, es decir, son independientes las **promo** y las **tarjeta**, solo tienen relación con el fraude, no entre ellas. Tras esta simplificación:

$$P(\text{fraude/tarjeta, promo}) = P(\text{fraude}) P(\text{tarjeta/fraude}) P(\text{promo/fraude})$$

Esto es proporcional al numerador de la fórmula, y para simplificar más aún las cosas, podemos asegurar que se normaliza con algún valor Z que será la suma de todas las probabilidades de las clases. Así que ahora el modelo queda así:

$$P(\text{fraude/tarjeta, promo}) = 1/Z P(\text{fraude}) P(\text{tarjeta/fraude}) P(\text{promo/fraude})$$

Para usar esto en un problema de clasificación, tenemos que calcular simplemente qué entrada (fraude o no fraude) es la que tiene mayor probabilidad.

	Fraude	No Fraude
Tarjeta regalo presente	60%	30%
Varias promociones usadas	50%	30%
Probabilidad de la clase	10%	90%

En este momento puedes usar esta información para determinar cuando un pedido es fraudulento basándote en si tiene una tarjeta de regalo y cuando ha usado varias promociones. La probabilidad de que el pedido sea fraudulento dado que usa tarjeta y múltiples promociones es del 62.5%. Aunque no podemos saber el número exacto de pedidos que debes revisar, si que nos permite realizar un mejor juicio.

Aún hay un escollo que solucionar y consiste en ¿Qué hacer cuando las probabilidades de usar varias promociones dado un pedido fraudulento sea cero? Este resultado puede ocurrir por diferentes razones, como que los datos no tengan suficientes ejemplos (tamaño de la muestra). La manera de solucionarlo es usando **pseudocontadores**.

PSEUDOCONTADORES

Cuando se introduce nueva información o los datos tienen poco tamaño, por ejemplo, tenemos un lote de datos sobre *emails* que queremos clasificar como spam o no spam. Hemos implementado las probabilidades usando estos datos, pero aparece una nueva palabra usada en algunos spam, la

palabra '**fuzzbolt**', así que cuando calculamos la probabilidad de spam condicionado a que contenga esta palabra, obtenemos probabilidad cero.

Este efecto provoca que falle la predicción ya que los clasificadores naive Bayes multiplican todas las probabilidades para calcular la predicción, si cualquiera de ellas es cero, la probabilidad será cero. Por ejemplo si llega un mail con el campo *subject* a '*Fuzzbolt: Prince of Nigeria*' suponiendo que se omite la palabra 'of'.

Si queremos calcular el ratio de spam o ham, en ambos casos obtenemos cero porque la palabra no estará presente o estará con probabilidad cero.

Word	Spam	Ham
Fuzzbolt	0	0
Prince	75%	15%
Nigeria	85%	10%

La solución es usar pseudocontadores. Cuando vamos a calcular las probabilidades añadimos 1 a la cuenta de todas las palabras. Es decir, al contar cuantas veces aparece cada palabra, usamos **contador + 1**.

EJEMPLO 3: Usando el teorema de Bayes clasifica un día en: se juega o no al tenis según el clima que haga ese día. Tenemos anotados unos datos sobre el clima que hace cada día y si hemos jugado a tenis (yes) o no. Queremos poder predecir si cierto día jugaremos sabiendo el tiempo que hará. Tienes que calcular la probabilidad de jugar en función del clima (característica *whether*).

The diagram illustrates the process of calculating probabilities for a weather-based tennis prediction model. It starts with a raw data table on the left, which is then processed through two frequency tables (Frequency Table and Likelihood Table 1) before arriving at Likelihood Table 2.

Whether	Play
Sunny	No
Sunny	No
Overcast	Yes
Rainy	Yes
Rainy	Yes
Rainy	No
Overcast	Yes
Sunny	No
Sunny	Yes
Rainy	Yes
Sunny	Yes
Overcast	Yes
Overcast	Yes
Rainy	No

Frequency Table

Whether	No	Yes
Overcast		4
Sunny	2	3
Rainy	3	2
Total	5	9

Likelihood Table 1

Whether	No	Yes
Overcast	4	=4/14 0.29
Sunny	2	=5/14 0.36
Rainy	3	=5/14 0.36
Total	5	9
	=5/14 =9/14	
	0.36	0.64

Likelihood Table 2

Whether	No	Yes	Posterior Probability for No	Posterior Probability for Yes
Overcast	4	3	0/5=0	4/9=0.44
Sunny	2	3	2/5=0.4	3/9=0.33
Rainy	3	2	3/5=0.6	2/9=0.22
Total	5	9		

El clasificador *Naive Bayes* calcula la probabilidad de un suceso en los siguientes pasos:

- **Paso 1:** calcula la probabilidad a priori de las etiquetas (clases) (Table).
- **Paso 2:** calcula la probabilidad de cada valor en cada clase (Table 1).
- **Paso 3:** usa los valores para calcular la probabilidad a posteriori (condicionadas, Table2).
- **Paso 4:** comprueba qué clase tiene mayor probabilidad (la entrada pertenece a la clase de mayor probabilidad).

Para simplificar el cálculo de probabilidades a priori y a posteriori, puedes utilizar las dos tablas de frecuencias y probabilidades como se ve en la figura de arriba. Ambas tablas te ayudarán a calcular la probabilidad a priori y a posteriori. La tabla de frecuencias (*Frequency table*) contiene la frecuencia de aparición de etiquetas para todas las características. Hay dos tablas de probabilidad:

- "Likelihood Table 1" muestra las probabilidades a priori de las etiquetas.
- "Likelihood Table 2" muestra la probabilidad a posteriori.

Imagina que quieres calcular la probabilidad de jugar cuando el tiempo está nublado.

Probabilidad de jugar:

$$P(\text{Yes}|\text{Overcast}) = P(\text{Overcast}|\text{Yes}) P(\text{Yes}) / P(\text{Overcast}) \quad [\text{ecuación 1}]$$

1. Calcula las probabilidades previas (a priori):

$$P(\text{Overcast}) = 4/14 = 0.29$$

$$P(\text{Yes}) = 9/14 = 0.64$$

2. Calcula las Probabilidades Posterioras (a posteriori o condicionadas):

$$P(\text{Overcast}|\text{Yes}) = 4/9 = 0.44$$

3. Pon las probabilidades en la ecuación 1.

$$P(\text{Yes}|\text{Overcast}) = 0.44 * 0.64 / 0.29 = 0.98$$

Del mismo modo, puedes calcular la probabilidad de no jugar:

$$P(\text{No}|\text{Overcast}) = P(\text{Overcast}|\text{No}) P(\text{No}) / P(\text{Overcast}) \quad [\text{Ecuación 2}]$$

1. Calcula las probabilidades previas:

$$P(\text{Overcast}) = 4/14 = 0.29$$

$$P(\text{No}) = 5/14 = 0.36$$

2. Calcula las Probabilidades a posteriori (condicionales):

$$P(\text{Overcast}|\text{No}) = 0/9 = 0$$

3. Pon las probabilidades a priori y condicionales (a posteriori) en la ecuación 2.

$$P(\text{No}|\text{Overcast}) = 0 * 0.36 / 0.29 = 0$$

La probabilidad de una clase "Yes" es mayor. Así que puedes determinar que si el tiempo está nublado se juega a tenis.

EJEMPLO 4: El mismo caso que el ejemplo 1, pero ahora vamos a usar más de una característica: además del clima usamos la temperatura (*hot*=calor, *Mild*=templado, *Cool*=frío). Tenemos estos datos:

Para este caso hay que realizar estos pasos:

1. Calcular probabilidades a priori de las etiquetas de las clases.
2. Calcular la probabilidad condicional de cada atributo para cada clase.
3. Multiplicar las probabilidades condicionales de la misma clase.
4. Multiplicar la probabilidad a priori con el producto del paso 3.
5. Se predice la clase con probabilidad más alta.

Supón ahora que quieres calcular la probabilidad de jugar cuando el tiempo está nublado y la temperatura es suave.

Probabilidad de jugar:

$$P(\text{Yes}|\text{Overcast}, \text{Mild}) = P(\text{Overcast}, \text{Mild}|\text{Yes}) P(\text{Yes}) \quad [\text{Ec.1}]$$

$$P(\text{Overcast}, \text{Mild}|\text{Yes}) = P(\text{Overcast}|\text{Yes}) P(\text{Mild}|\text{Yes}) \quad [\text{Ec.2}]$$

1. Calcula las probabilidades previas:

$$P(\text{Yes}) = 9/14 = 0.64$$

Whether	Temperature	Play
Sunny	Hot	No
Sunny	Hot	No
Overcast	Hot	Yes
Rainy	Mild	Yes
Rainy	Cool	Yes
Rainy	Cool	No
Overcast	Cool	Yes
Sunny	Mild	No
Sunny	Cool	Yes
Rainy	Mild	Yes
Sunny	Mild	Yes
Overcast	Mild	Yes
Overcast	Hot	Yes
Rainy	Mild	No

2. Calcula las Probabilidades Posterioras:

$$P(\text{Overcast}|\text{Yes}) = 4/9 = 0.44$$

$$P(\text{Mild}|\text{Yes}) = 4/9 = 0.44$$

3. Pon las probabilidades Posterioras en la ecuación 2.

$$P(\text{Overcast}, \text{Mild}|\text{Yes}) = 0.44 * 0.44 = 0.1936$$

4. Pon las probabilidades a priori y a posteriori en la ecuación 1.

$$P(\text{Yes}|\text{Overcast}, \text{Mild}) = 0.1936 * 0.64 = 0.124$$

Del mismo modo, puedes calcular la probabilidad de no jugar:

$$P(\text{No}|\text{Overcast}, \text{Mild}) = P(\text{Overcast}, \text{Mild}|\text{No}) P(\text{No}) \quad [\text{Ecuación 3}]$$

$$P(\text{Overcast}, \text{Mild}|\text{No}) = P(\text{Overcast}|\text{No}) P(\text{Mild}|\text{No}) \quad [\text{Ecuación 4}]$$

1. Calcula las probabilidades previas:

$$P(\text{No}) = 5/14 = 0.36$$

2. Calcula las Probabilidades Posteriores:

$$P(\text{Overcast}|\text{No}) = 0/9 = 0$$

$$P(\text{Mild}|\text{No}) = 2/5 = 0.4$$

3. Pon las probabilidades posteriores en la ecuación 4.

$$P(\text{Overcast}, \text{Mild}|\text{No}) = 0 * 0.4 = 0$$

4. Pon las probabilidades a priori y a posteriori en la ecuación 3.

$$P(\text{No}|\text{Overcast}, \text{Mild}) = 0 * 0.36 = 0$$

La probabilidad de una clase "Yes" es mayor. Así que se puede decir que si el tiempo está nublado y la temperatura es templada, se juega al tenis.

1.2. ALGORITMOS DE APRENDIZAJE NAIVE BAYES.

Pertenecen a una **familia de algoritmos probabilísticos** que calculan las probabilidades de cada característica predictoría de los datos que pertenecen a cada clase para calcular una estimación de la distribución de probabilidad de todas las clases más probables a las que está asociado un ejemplo. Bayes es el método usado para mapear las probabilidades de los datos que observa (los datos de entrada) sobre las clases que debe predecir.

La otra palabra del nombre naive¹ viene del hecho de que estamos suponiendo (de forma simplona, inocente) que todas las características predictoras son completamente independientes entre sí. En realidad podría ocurrir perfectamente que no lo sean, o al menos que no lo sean totalmente y esto genera errores en la estimación de las distribuciones de probabilidad.

Estos algoritmos se basan en aplicar el teorema de Bayes con la suposición naive de que cada dos predictoras son condicionalmente independientes dado el valor que tiene la clase a la que pertenecen. El *teorema de Bayes* establece una relación entre la variable independiente y el vector de predictoras $\mathbf{x}_1, \dots, \mathbf{x}_n$:

$$P(y | x_1, \dots, x_n) = \frac{P(y)P(x_1, \dots, x_n | y)}{P(x_1, \dots, x_n)}$$

Usando la suposición naive de independencia condicional:

$$P(\mathbf{x}_i | \mathbf{y}, \mathbf{x}_1, \dots, \mathbf{x}_{i-1}, \mathbf{x}_{i+1}, \dots, \mathbf{x}_n) = P(\mathbf{x}_i | \mathbf{y})$$

¹ **Naive:** se dice de cualquier cosa que no está respaldada de forma objetiva por los hechos, sino cogida con pinzas. Una suposición simplista, una persona sin malicia, con comportamiento infantiloide o simple, inocente, etc.

para todos los i , estas relaciones se simplifican a

$$P(y \mid x_1, \dots, x_n) = \frac{P(y) \prod_{i=1}^n P(x_i \mid y)}{P(x_1, \dots, x_n)}$$

Puesto que $P(x_1, \dots, x_n)$ es un valor constante dada una entrada, podemos establecer la siguiente regla de clasificación cuando nos encontremos ese ejemplo:

$$\hat{y} = \arg \max_y P(y) \prod_{i=1}^n P(x_i \mid y),$$

Y usamos el máximo a posteriori (**MAP**) para estimar $P(y)$ y $P(x_i|y)$ que dan forma a la frecuencia relativa de la clase y en los datos de entrenamiento, lo que nos sirve para realizar una predicción.

Los diferentes clasificadores *naive Bayes* se diferencian principalmente en las suposiciones sobre la distribuciones de probabilidades de $P(x_i|y)$.

A pesar de las suposiciones que aparentemente sobre simplifican la realidad, los clasificadores *naive Bayes* funcionan bastante bien en situaciones reales, obteniendo éxitos importantes en la clasificación de documentos y el filtrado de correos spam. Requieren poca cantidad de datos de entrenamiento para estimar los parámetros que necesitan. Los clasificadores son extremadamente rápidos comparados con otros modelos equivalentes más sofisticados. Como desacopla las predictoras (lo supone) puedes estimarlas de manera individual con una distribución de probabilidades unidimensional. Aunque se considera que los clasificadores *naive Bayes* tienen un comportamiento decente, en realidad son estimadores poco fiables, así que las probabilidades que calculan no hay que tomarlas muy en serio.

EJEMPLO 5: Vamos a construir un clasificador de spam usando el teorema de Bayes. Imagina que tenemos 100 correos y de ellos 25 son spam. Usamos las palabras de cada mail para clasificarlos.



Por ejemplo la palabra "barato" aparece en 20 de los 25 correos *spam* y aparece en 5 de los 75 correos normales.



2 El símbolo \propto : significa “proporcional a”, lo de la izquierda es lo que hay a la derecha multiplicado por un factor.

Nos olvidamos de los correos que no tengan esta palabra y calculamos que probabilidad hay de que un correo sea spam o normal si contiene la palabra "barato". ¿Cuál es la probabilidad de...?:

$$P(\text{spam}|\text{barato}) = \frac{20}{20 + 5} = 0.8 \quad P(\text{normal}|\text{barato}) = \frac{5}{20 + 5} = 0.2$$

Podemos repetir el mismo cálculo con todas las palabras que aparezcan en nuestros correos, por ejemplo ahora usamos la palabra "comprar":



$$P(\text{spam}|\text{comprar}) = \frac{15}{15 + 10} = 0.6 \quad P(\text{normal}|\text{comprar}) = \frac{10}{15 + 10} = 0.4$$

Si durante la fase de entrenamiento recolectamos estas probabilidades de cada palabra que aparezca en uno de los correos que usemos para entrenar y luego queremos predecir si un nuevo correo es o no *spam* ¿Cómo combinamos las probabilidades de sus palabras? Es decir, imagina que el correo tiene "barato" y tiene "comprar":



Si repetimos lo que hemos hecho hasta ahora, podemos llegar a la siguiente situación: aunque en nuestros datos de entrenamiento tenemos 12 correos *spam* con ambas palabras pero no tenemos ni un solo correo normal que contiene las dos palabras (aunque hay 5 que contienen "barato" y hay 10 que contienen "comprar").



$$P(\text{spam}|\text{barato y comprar}) = \frac{12}{12 + 0} = 1 \quad P(\text{normal}|\text{barato y comprar}) = \frac{0}{12 + 0} = 0$$

Decir que hay una seguridad del 100% en que es *spam*, es la consecuencia de que no tengamos un correo normal con las dos palabras ¿Pero realmente significa que no pueden existir esos correos? No, evidentemente lo que significa es que nosotros ahora mismo no los tenemos, no que no existan. Para solucionarlo podemos recolectar más correos hasta que encontremos los que contengan esas palabras. Pero igual nos pasa con otras palabras. Así que recolectar más correos no parece la solución idónea. La otra alternativa es usar el *teorema de Bayes* para estimar la

probabilidad de que aparezca ese correo con las dos palabras (o otras palabras cualesquiera) sin la necesidad de tener ese correo en tus datos de entrenamiento.

Este segundo enfoque usando el *teorema de Bayes* es más apropiado:

- En total hay 100 correos.
- 5 correos de esos 100 tienen la palabra "barato" → el 5% de ellos, $P(B)=0.05$
- 10 correos de esos 100 tienen la palabra "comprar" → el 10%, $P(C)=0.1$
- ¿Probabilidad de un correo que tenga las dos palabras? El 5% del 10% = $P(B \cap C)= 0.005$
- ¿Cuántos correos tendrán las dos (se estima)? $P(B \cap C) \times \text{nº_correos} = 0.005 \times 100 = 0.5$

Podemos hacer lo mismo fijándonos solamente en los correos que son *spam*:

- Total de correos *spam* = 25
- Tienen la palabra "barato" = 20 → $20 / 25 = 0.8$
- Tienen la palabra "comprar" = 15 → $15 / 25 = 0.6$
- ¿Probabilidad e que un correo *spam* tenga las dos? $0.8 \times 0.6 = 0.48$
- ¿Cuántos *spam* hay con las dos palabras? $25 \times 0.48 = 12$ (no es la realidad, es una aproximación, aunque en este caso coincide con la realidad, solo es una casualidad)

Hacemos lo mismo fijándonos solamente en los correos que son normales:

- Total de correos normales = 75
- Tienen la palabra "barato" = 5 → $5 / 75 = 0.066$
- Tienen la palabra "comprar" = 10 → $10 / 75 = 0.133$
- ¿Probabilidad de correo *normal* que tenga las dos? $0.066 \times 0.133 = 0.00888$
- ¿Cuántos normales hay con las dos palabras? $75 * 0.00888 = 0.6666$

Así que en nuestros datos podemos estimar que con las palabras "barato" y "comprar":

- Cantidad de *spam*: 12
- Cantidad de correos normales: 0.6666

Ahora sí podemos calcular las probabilidades de que un correo que tenga las dos palabras sea *spam* o normal:

$$P(\text{spam} | \text{barato y comprar}) = P(\text{barato y comprar} | \text{spam}) p(\text{spam}) \\ = 12 / (12 + 0.6666) = 0.9473$$

$$p(\text{normal} | \text{barato y comprar}) = 0.6666 / (12 + 0.6666) = 0.0526$$

Hacerlo usando el teorema de Bayes: $s=\text{spam}$, $b=\text{barato}$, $c=\text{comprar}$, $n=\text{normal}$

$$P(s|b) = \frac{P(b|s)P(s)}{P(b|s)P(s)+P(b|n)P(n)} = \frac{20/25 \quad 25/100}{20/25 \quad 25/100 + 5/75 \quad 75/100} = 0.8$$

$$P(s|c) = \frac{P(c|s)P(s)}{P(c|s)P(s)+P(c|n)P(n)} = \frac{15/25 \quad 25/100}{15/25 \quad 25/100 + 10/75 \quad 75/100} = 0.6$$

Ahora asumiendo que la aparición de "barato" es independiente de la aparición de "comprar" (algo ingenuo, naive): $P(B \text{ y } C) = P(B) P(C)$. Sustituimos en la fórmula de Bayes:

$$P(s|b \text{ y } c) = \frac{P(b \text{ y } c|s)P(s)}{P(b \text{ y } c|s)P(s)+P(b \text{ y } c|n)P(n)} = \frac{P(b|s)P(c|s)P(s)}{P(b|s)P(c|s)P(s) + P(b|n)P(c|n)P(n)} = \\ = \frac{20/25 \quad 15/25 \quad 25/100}{20/25 \quad 15/25 \quad 25/100 + 5/75 \quad 10/75 \quad 75/100} = 94.737\%$$

CUANDO USAR NAIVE BAYES

Como los clasificadores Bayesianos hacen suposiciones forzadas sobre los datos, generalmente no tienen tan buen desempeño como un modelo más complejo. Pero tienen varias ventajas sobre estos:

- Extremadamente rápidos tanto para entrenar como para predecir.
- Aportan predicciones probabilísticas directas.
- Son fácilmente interpretables.
- Tienen muy pocos parámetros ajustables.

Estas propiedades los hacen **ideales para usarlos como un primer clasificador base**, con el objetivo de mejorar su funcionamiento con otros modelos más complejos. Si su funcionamiento es adecuado, enhorabuena porque tienes un clasificador rapidísimo que además funciona bien. En general, se suelen comportar bien en las siguientes situaciones:

- **Si las suposiciones naive casan con el patrón** que siguen los datos (raro en la práctica).
- **Categorías muy bien separadas** en cuyo caso la complejidad del modelo no importa.
- **Datos con alta dimensionalidad**: no hay muchos datos para entrenar un modelo complejo.

Los dos últimos puntos son distintos aunque están relacionados: cuando las dimensiones de un dataset crece, es menos probable que dos puntos de datos se encuentren cerca. Esto significa que los grupos de datos tienden a estar más separados entre sí, de media, que los grupos de bajas dimensiones, asumiendo que nuevas dimensiones añaden información. Por esta razón, **los clasificadores simples como los bayesianos tienden a funcionar bien o mejor que los complejos cuando la dimensionalidad crece**: con suficientes datos, un modelo simple puede ser muy potente.

Hay diferentes tipos de clasificadores *naive Bayes* para diferentes suposiciones sobre los datos, en los siguientes apartados veremos las dos más usadas, para una lista más exhaustiva mira la documentación de **scikit-learn**.

1.3. NAIVE BAYES GAUSIANO.

En este clasificador se asume que los datos de cada etiqueta siguen una distribución de probabilidad gausiana. La clase **GaussianNB** del paquete **sklearn.naive_bayes** implementa el algoritmo para clasificación. La probabilidad condicional de las características se calculan como:

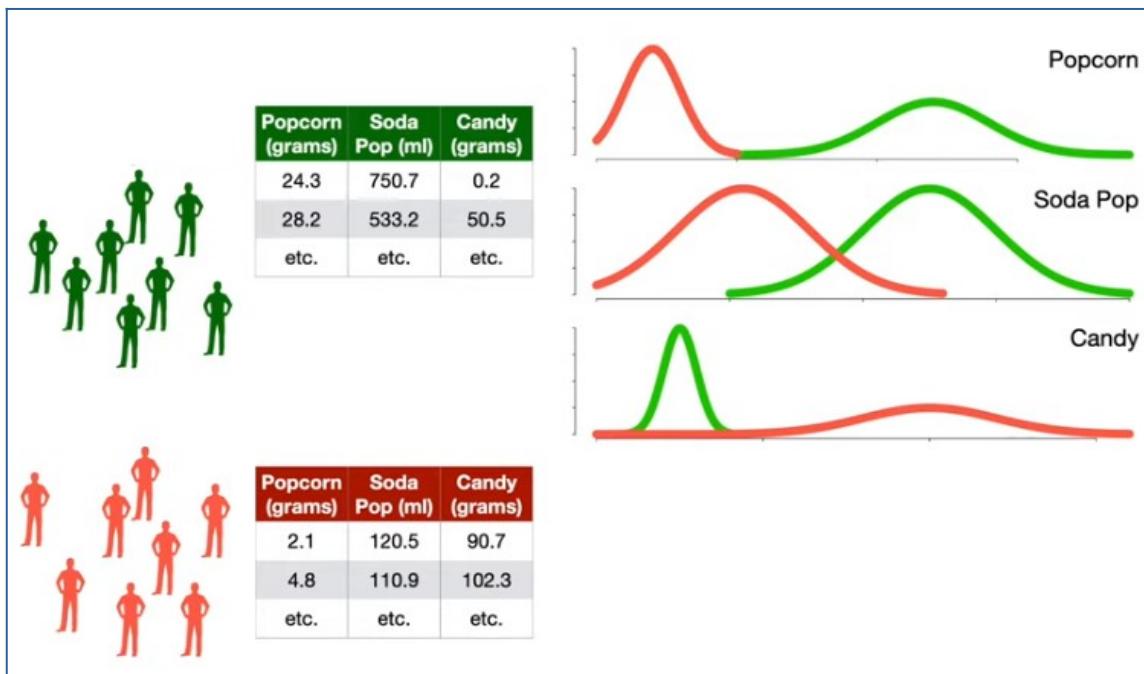
$$P(x_i | y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right)$$

EJEMPLO 6: Queremos saber si a los espectadores de una película les ha gustado la película a través de su comportamiento. Se coleccionan datos de cuantos gramos de palomitas comen (popcorn), mililitros de refresco que beben (soda) y gramos de chucherías (candy) que comen. Se entrena un modelo **Naive Bayes Gausiano** con los espectadores de la primera sesión y se les pide que puntúen la película de 1 a 4, de manera que podamos saber su opinión y asociarla a su comportamiento. Hay 8 espectadores a los que ha gustado (en verde) y 8 a los que no (en rojo).

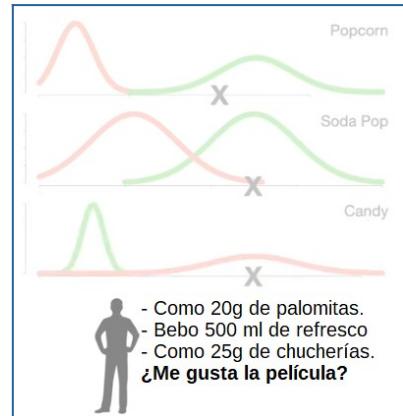
A partir de los datos se calculan las *distribuciones de probabilidad gausianas* de cada serie de datos que aparece en la figura (en rojo la de espectadores que no les ha gustado la película y en verde la de espectadores a quien sí les ha gustado). Por ejemplo, el consumo de palomitas de quien le ha gustado sigue una normal de media 24 y desviación 4 y la distribución de quienes no les ha gustado sigue una normal de media 4 y desviación 2.

Una vez entrenado el modelo, cuando un nuevo espectador aparezca, según la cantidad de palomitas, refresco y chucherías que coma lo vamos a clasificar en las categorías "le gusta" y "no le gusta". La probabilidad a priori de que a un espectador le guste o no es:

$$P(\text{le gusta}) = \frac{8}{8 + 8} = 0.5 \quad P(\text{no le gusta}) = \frac{8}{8 + 8} = 0.5$$



Ahora usamos el *Likelihood* de cada valor. El *Likelihood* no siempre equivale a una probabilidad, porque en una distribución de probabilidad continua la probabilidad es el área que encierra un trozo de la curva, mientras que el *Likelihood* sería el valor de la curva asociado a un punto (el valor de la curva en un punto de la variable independiente).

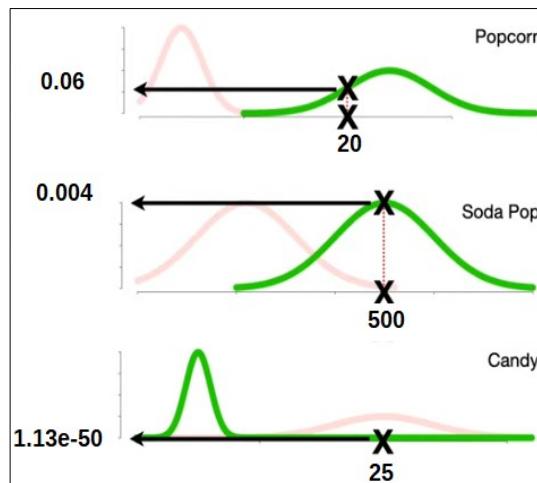


La probabilidad de que el espectador que ha comido 200g de palomitas, bebido 500 ml de refresco y comido 25 gramos de chucherías le guste la película se puede calcular como:

$$P(\text{le gusta} | \text{palomitas}=20 \wedge \text{refresco}=500 \wedge \text{chucherías}=25) =$$

$$P(\text{le gusta}) P(\text{palomitas}=20 | \text{le gusta}) P(\text{refresco}=500 | \text{le gusta}) P(\text{chucherías}=25 | \text{le gusta})$$

Como no sabemos las probabilidades las estimamos usando el *Likelihood*, cambiamos $P()$ por $L()$:



$$P(\text{le gusta} | \text{palomitas}=20 \text{ Y refresco}=500 \text{ Y chucherías}=25) =$$

P(<i>le gusta</i>) x	P(<i>le gusta</i>) x	0.5 x
P(palomitas=20 <i>le gusta</i>) x	L(palomitas=20 <i>le gusta</i>) x	0.06 x
P(refresco=500 <i>le gusta</i>) x	L(refresco=500 <i>le gusta</i>) x	0.004 x
P(chucherias=25 <i>le gusta</i>)	L(chucherias=25 <i>le gusta</i>) =	¡¡peligro underflow!!

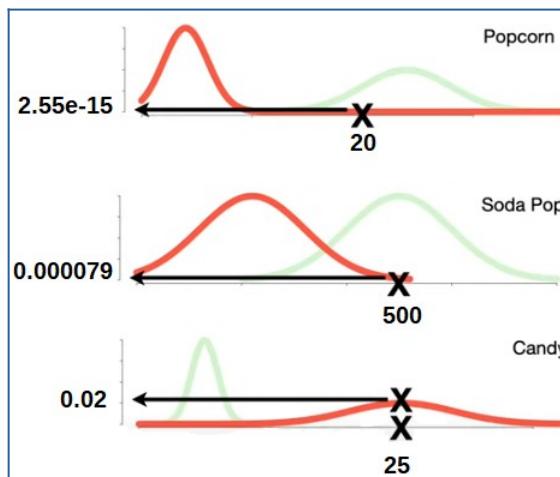
El **underflow** ocurre cuando un valor es demasiado pequeño y se corre el peligro de que el formato numérico no sea capaz de representarlo y acabe aproximándolo a cero. Eso podría provocar una falla en cascada de los siguientes cálculos, en este caso puede hacer que el *score* sea cero.

Una forma de evitar este problema es utilizar el logaritmo de las probabilidades en vez de los valores. Podemos usar logaritmos de cualquier base, pero en machine learning cuando usamos *log()* nos estamos refiriendo al logaritmo neperiano: $\ln(x) = \log_e(x)$. Recuerda que es la inversa de la exponencial: $\log_e(e^x) = x$. Recuerda también que $\log(x \cdot y) = \log(x) + \log(y)$. Por tanto:

$$\begin{aligned} \log(P(\text{le gusta} | \text{palomitas}=20 \text{ Y refresco}=500 \text{ Y chucherías}=25)) &= \\ \log(P(\text{le gusta}) L(\text{palomitas}=20|\text{le gusta}) L(\text{refresco}=500|\text{le gusta}) L(\text{chucherias}=25|\text{le gusta})) &= \\ \log(L(\text{le gusta})) + &\quad \log(0.5) + & -0.69 + \\ \log(L(\text{palomitas}=20|\text{le gusta})) + &\quad \log(0.06) + & -2.81 + \\ \log(L(\text{refresco}=500|\text{le gusta})) + &\quad \log(0.004) + & -5.521 + \\ \log(L(\text{chucherias}=25|\text{le gusta})) = &\quad \log(1.13e-50) = & -115.01 = \mathbf{-124} \end{aligned}$$

Si hacemos lo mismo para

$$\begin{aligned} \log(P(\text{no le gusta} | \text{palomitas}=20 \text{ Y refresco}=500 \text{ Y chucherías}=25)) &= \\ \log(L(\text{no le gusta})) + &\quad \log(0.5) + & -0.69 + \\ \log(L(\text{palomitas}=20|\text{no le gusta})) + &\quad \log(2.55e-15) + & -33.6 + \\ \log(L(\text{refresco}=500|\text{no le gusta})) + &\quad \log(0.02) + & -9.45 + \\ \log(L(\text{chucherias}=25|\text{no le gusta})) = &\quad \log(0.000079) = & -3.91 = \mathbf{-47.66} \end{aligned}$$



Como $-47.66 > -124$, es más probable que al espectador no le guste la película.

Imagina que tienes los siguientes datos que aparecen en la figura 2 y que genera este código:

```
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
X, y = make_blobs(100, 2, centers=2, random_state=449,
                  cluster_std=1.5)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='RdBu');
```

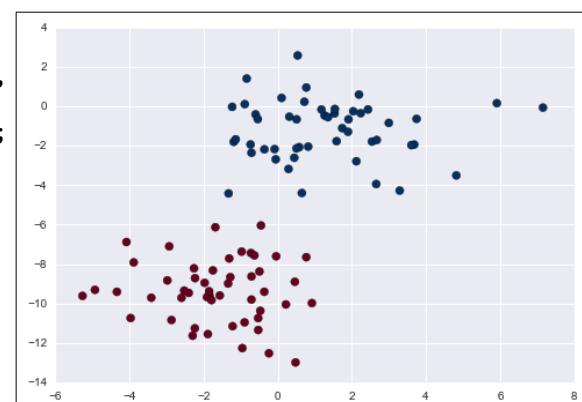


Figura 2. Datos creados para un clasificador naive Bayes Gausiano.

Una forma rápida de crear un sencillo modelo es suponer que los datos pueden ser descritos por una distribución gausiana sin covarianza entre sus dimensiones.

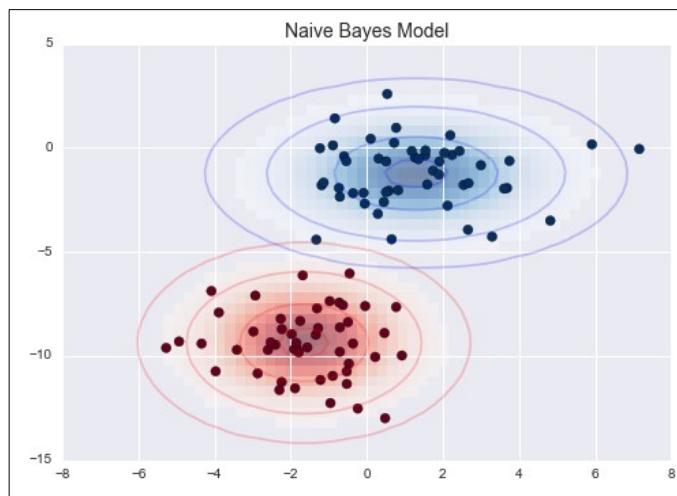


Figura 3. Visualización del modelo naive Bayes Gausiano.

Podemos definir este modelo encontrando la media y la desviación estándar de los valores que tenga cada columna, que es todo lo que se necesita para definir la distribución. El resultado se aprecia en la figura 3.

Las elipses de las figuras representan la superficie que genera el modelo para cada etiqueta, con mayor probabilidad en el centro de las elipses. Con este modelo generativo en lugar de tener un modelo para cada clase, tenemos una simple fórmula que nos permite calcular la estimación de la probabilidad P que caracteriza al valor del target $L1$ para cualquier punto de datos, y así a posteriori podemos calcular el ratio y conocer el valor más probable para ese ejemplo de entrada. Este procedimiento está implementado en el estimador `sklearn.naive_bayes.GaussianNB`.

```
from sklearn.naive_bayes import GaussianNB
modelo = GaussianNB()
modelo.fit(X, y);
# Ahora generaremos nuevos datos y predecimos su etiqueta
import numpy as np
rng = np.random.RandomState(0)
nuevo_X = [-6, -14] + [14, 18] * rng.rand(2000, 2)
nuevo_y = modelo.predict(nuevo_X)
```

Podemos dibujar estos datos para hacernos una idea de donde están las fronteras de decisión:

```
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='RdBu')
limite = plt.axis()
plt.scatter(nuevo_X[:, 0], nuevo_X[:, 1], c=nuevo_y, s=20, cmap='RdBu', alpha=0.1)
plt.axis(limite)
```

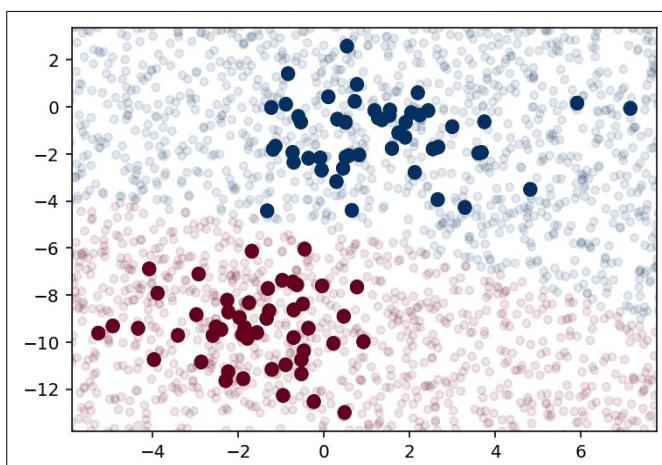


Figura 4. Visualización de la clasificación de naive Bayes Gausiano.

Como se ve en la imagen, las fronteras de decisión están ligeramente curvadas porque en naive Bayes Gausiano las fronteras son cuadráticas.

Con este modelo también tenemos la posibilidad de pedirle las probabilidades de una clasificación con su método `predict_proba`:

```
y_proba = modelo.predict_proba(nuevo_X)
print(y_proba[-8:].round(2)) # salida: [[ 0.89, 0.11], [1. , 0.], ... [0.15, 0.85]]
```

Las columnas dan las probabilidades a posteriori de la primera y segunda etiqueta respectivamente. Si quieras estimar la incertidumbre de tu clasificación, puede ser útil. La clasificación que realice será tan buena como buena sea la suposición sobre la distribución *gausiana* de los datos y ese es el motivo de que a veces no produzca buenos resultados, sobre todo si el número de características es grande.

EJEMPLO 7: Usar un modelo *Naive Bayes Gausiano* con el dataset iris.

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
X, y = load_iris(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5, random_state=0)
gnb = GaussianNB()
y_pred = gnb.fit(X_train, y_train).predict(X_test)
print("Hay %d ejemplos mal clasificados de %d" % ((y_test != y_pred).sum(),
X_test.shape[0]) )
```

1.4. NAIVE BAYES MULTINOMIAL.

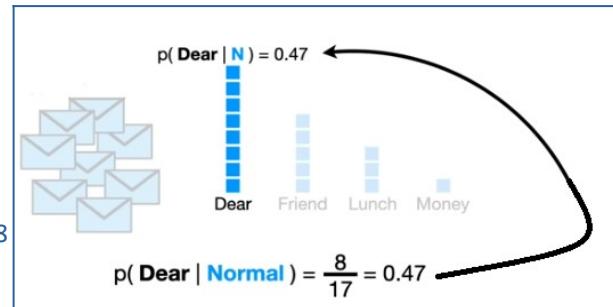
Se asume que las características pueden describirse o generarse mediante una *distribución de probabilidad multinomial*. La distribución multinomial describe la probabilidad de observar contadores de cierta cantidad de categorías, de forma que es apropiada cuando se trabaja con contadores o ratios de contadores. La idea del algoritmo es encontrar la mejor distribución multinomial que describa los datos.

Una aplicación donde se utiliza con mucha frecuencia es el de la clasificación de textos, donde cada característica es el contador de una palabra o su frecuencia de aparición dentro de un texto. que quiere clasificarse. Haremos algunas actividades donde también aprenderemos a extraer las características a partir de textos.

EJEMPLO 8: Vamos a clasificar correos en *spam* o no fijándonos en las palabras que contienen. Tenemos un total de 12 correos, 8 de ellos son normales y 4 son *spam*. Entrenamos un modelo *Naive Bayes Multinomial* y necesitamos contar las apariciones de cada palabra significativa que aparezca en cualquiera de los correos.

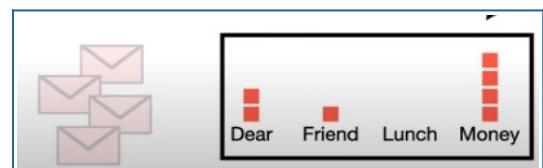
En cuanto a los correos normales tenemos:

- 8 correos normales.
- Palabra "Dear" aparece 8: $8/17 = 0.47$
- Palabra "Friend" aparece 5: $5/17 = 0.29$
- Palabra "Lunch" aparece 3: $3/17 = 0.17$
- La palabra "Money" aparece 1: $1/17 = 0.058$



En cuanto a los *spam*:

- 4 correos *spam*.
- Palabra "Dear" aparece 2: $2/7 = 0.29$
- Palabra "Friend" aparece 1: $1/7 = 0.14$
- Palabra "Lunch" aparece 0: $0/7 = 0$
- La palabra "Money" aparece 4: $4/7 = 0.57$



Como tenemos calculado las probabilidades no sobre un continuo sino sobre puntos discretos, en realidad son *likelihoods*:

$$\begin{array}{ll}
 L(dear|normal) = 0.47 & L(dear|spam) = 0.29 \\
 L(friend|normal) = 0.29 & L(friend|spam) = 0.14 \\
 L(lunch|normal) = 0.17 & L(lunch|spam) = 0 \\
 L(money|normal) = 0.058 & L(money|spam) = 0.57
 \end{array}$$

Una vez entrenado el modelo, si aparece un correo con las palabras "Dear friend" podemos saber que la probabilidad de ser *spam* o *normal* será proporcional a:

$$P(normal|Dear \text{ Y } friend) \propto^3 P(normal) L(dear|normal) L(friend|normal) = 8/(8+4) 0.47 0.29 = 0.091$$

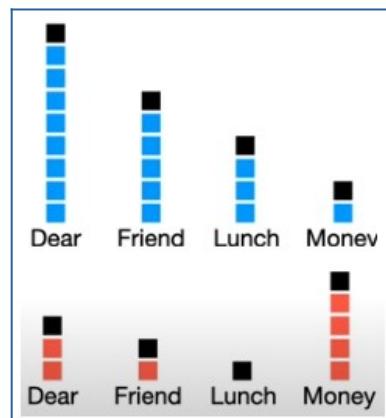
$$P(spam|Dear \text{ Y } friend) \propto^4 P(spam) L(dear|spam) L(friend|spam) = 4/(8+4) 0.29 0.14 = 0.0116$$

Por tanto el correo sería clasificado como normal porque es el valor mayor.

Si ahora entrenamos otro modelo con otros 8 correos normales y 4 *spam* pero distintos y obtenemos los resultados llega otro correo con las palabras "*lunch money money money*" tenemos un problema y es que la palabra "*lunch*" no aparece en los correos *spam* y eso hace que la probabilidad calculada sea 0.

Sin embargo es un resultado incorrecto, demasiado tajante, como consecuencia de nuestra incapacidad de recolectar los infinitos correos para entrenar nuestro modelo. ¿Cómo solucionarlo? A todas las palabras (aunque no aparezcan en nuestros correos de entrenamiento) les regalamos una cantidad fija (α), digamos 1. Son los cuadrados negros del gráfico. Lo que hará que el método se comporte mejor en esta situación aunque debemos recalcular las probabilidades:

$$\begin{array}{ll}
 L(dear|normal) = 9/21 = 0.43 & L(dear|spam) = 3/11 = 0.27 \\
 L(friend|normal) = 6/21 = 0.28 & L(friend|spam) = 2/11 = 0.18 \\
 L(lunch|normal) = 4/21 = 0.19 & L(lunch|spam) = 1/11 = 0.09 \\
 L(money|normal) = 2/21 = 0.09 & L(money|spam) = 5/11 = 0.45
 \end{array}$$



La distribución está parametrizada por los vectores $\theta_y = (\theta_{y_1}, \dots, \theta_{y_n})$ para cada clase y , donde n es el número de características (en una tarea de clasificación de textos sería el tamaño del vocabulario) y θ_{y_i} es la probabilidad $P(x_i|y)$ de que la característica i aparezca en un ejemplo que se asocia con la clase y .

Los parámetros θ_y son estimados por una versión suavizada de máxima verosimilitud, es decir, sus contadores de frecuencia relativa:

$$\hat{\theta}_{yi} = \frac{N_{yi} + \alpha}{N_y + \alpha n}$$

Donde:

- $N_{yi} = \sum_{x_i \in T} x_i$ es el número de veces que la característica i aparece en un ejemplo de la clase y en el conjunto de datos train T ,
- $N_y = \sum_{i=1}^n N_{yi}$ es el contador total de todos los ejemplos con etiqueta y .
- El factor $\alpha \geq 0$ se utiliza para prevenir que los valores devueltos por la fórmula sean cero para cuando haya palabras que no aparezcan. Si fijas $\alpha=1$ se llama **suavizado de Laplace** mientras que si es inferior se llama **suavizado de Lidstone**.

³ α significa proporcional a la parte derecha.

⁴ α significa proporcional a la parte derecha.

EJEMPLO 9: Usamos contadores de palabras de 20 noticias de un foro para ver como clasifica.

```
from sklearn.datasets import fetch_20newsgroups
datos = fetch_20newsgroups()
print(datos.target_names) # Las clases que tiene el target
```

Por simplicidad, usamos solamente unas cuantas categorías y descargamos sus datos train y test:

```
categorias = ['talk.religion.misc', 'soc.religion.christian','sci.space','comp.graphics']
train = fetch_20newsgroups(subset='train', categories=categorias)
test = fetch_20newsgroups(subset='test', categories=categorias)
```

Podemos visualizar algunos de los textos de trabajo:

```
print(train.data[5])
```

Para usar estos datos en machine learning, es necesario convertir el contenido de cada frase en un vector de números. Para hacerlo usaremos el vectorizador **TF-IDF** y crearemos un pipeline que ataque un clasificador naive Bayes multinomial:

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import make_pipeline
model = make_pipeline(TfidfVectorizer(), MultinomialNB())
```

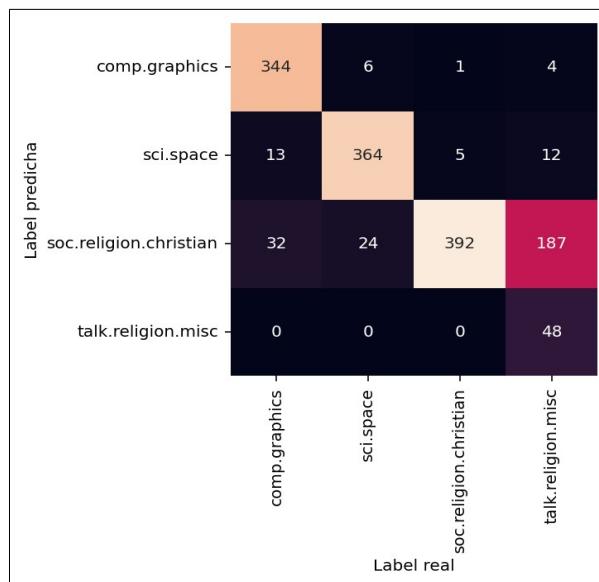


Figura 5. Matriz de confusión del clasificador de texto naive Bayes multinomial

Con este pipeline creamos un modelo que podemos usar para entrenar y clasificar:

```
modelo.fit(train.data, train.target)
labels = modelo.predict(test.data)
```

Ahora que tenemos las etiquetas predichas para los datos de test, podemos evaluarlo para saber su eficiencia a la hora de clasificar. Por ejemplo, generamos su matriz de confusión en los datos de test entre las etiquetas reales y las que predice, que es la que aparece en la figura 5:

```
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt
mc = confusion_matrix(test.target, labels)
sns.heatmap(mc.T, square=True, annot=True, fmt='d', cbar=False,
            xticklabels=train.target_names, yticklabels=train.target_names)
plt.xlabel('Label real')
plt.ylabel('Label predicha')
```

Hasta este extremadamente sencillo clasificador puede diferenciar satisfactoriamente textos que traten de por ejemplo religión de aquellos que hablen por ejemplo de informática gráfica, pero aún se confunde si se habla de cristianismo o de religión en general.

Lo interesante del ejemplo es que ahora tenemos una herramienta para clasificar cualquier texto en una de estas áreas o clases usando el método `predict()` del modelo. Podemos hacernos una función que nos ayude a realizarlo:

```
def predice_categoria(texto, train=train, model=modelo):
    pred = model.predict([texto])
    return train.target_names[pred[0]]
```

Si lo probamos:

```
print(predice_categoria('sending a payload to the ISS'))      #'sci.space'
print(predice_categoria('discussing islam vs atheism'))     #'soc.religion.christian'
print(predice_categoria('determining the screen resolution')) #'comp.graphics'
```

Recuerda que aunque esto no es nada más sofisticado que un simple modelo probabilístico para los pesos de cada palabra (frecuencias) bien entrenado con suficientes datos tiene resultados muy efectivos.

2. MÁQUINAS DE VECTORES SOPORTE (SVM).

Una **Support Vector Machine (SVM)** es un modelo de machine learning potente y versátil, capaz de realizar clasificaciones lineales y no lineales, regresiones y detección de *outliers*. Las SVM son particularmente adecuadas para clasificación de datos complejos pero de tamaño pequeño o mediano. Desarrolladas por *Vladimir Vapnik* y colaboradores en 1964 en los laboratorios *AT&T Bell* y mejorado en 1994 con la introducción de los kernel y los soft-margin de Corinna Cortes y Vapnik en 1995.

2.1. SVM PARA CLASIFICACIÓN BINARIA LINEAL.

Se aplica cuando las dos clases a distinguir son claramente separables mediante una línea recta (son *linealmente separables*). En la figura 6, el gráfico de la izquierda muestra las fronteras de decisión de 3 modelos lineales. La línea punteada de color verde no clasifica bien a los datos porque no separa los dos grupos de datos de manera adecuada. Los otros dos modelos lo hacen de manera perfecta en estos datos de entrenamiento, pero las líneas se quedan demasiado cerca de los ejemplos y probablemente no funcionen tan bien con nuevos datos, porque habrán cometido *overfitting*.

El gráfico de la derecha, tiene 3 líneas. La continua sería la frontera de decisión. Esta línea no solo separa los dos grupos de datos, sino que guarda la mayor distancia posible con ellos para evitar el *overfitting* y funcionar lo mejor posible con los nuevos datos que aparezcan. Esto es un clasificador SVM. Las líneas discontinuas que son paralelas a la central, son los vectores soporte. Definen una especie de carretera de dos carriles. La distancia entre ellas es el margen.

Un clasificador **SVM** (llamado **SVC**) consiste en encontrar la línea que separa a dos grupos de datos maximizando la distancia de los márgenes. También se le denomina **clasificador de margen amplio**.

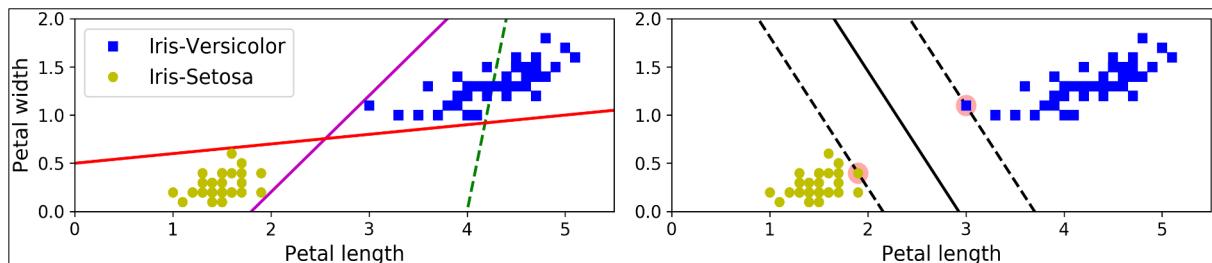


Figura 6. Clasificaciones con mayor margen.

Una manera de encontrar esta línea es localizar los puntos más cercanos entre ambas clases, trazar la línea que los une y las líneas perpendiculares que pasan por esos puntos forman los vectores

soporte. La frontera de decisión sería la línea paralela a los vectores soporte y que se encuentre en la mitad entre ellos.

Añadir más instancias a los datos de entrenamiento no afecta a la frontera de decisión si estas instancias están fuera de la calzada o carretera que define la SVM.

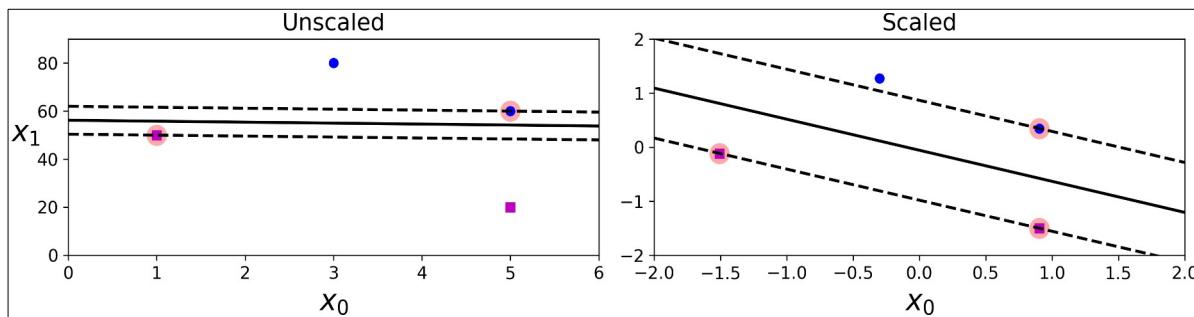


Figura 7. Las SVM son sensibles a las escalas

Las SVM son sensibles a las escalas de las características como puedes apreciar en la figura 7, en el gráfico de la izquierda, la escala vertical es mucho mayor que la horizontal (la característica vertical toma valores mucho mayores que la característica horizontal que tiene valores más pequeños y parece que están más juntos). Esto hace que los márgenes queden muy pequeños horizontalmente. Para evitar esto, antes de entrenar es conveniente escalar o normalizar los datos de entrenamiento. Al hacerlo obtenemos el caso del gráfico de la derecha. Podemos usar los objetos **StandardScaler** de Scikit-learn o definirnos nuestro propio código. Al hacerlo las SVM funcionan mejor como se puede apreciar en el gráfico.

HARD-MARGEN, SOFT-MARGEN E HIPERPARÁMETRO C

Si imponemos de manera estricta que todas las instancias estén alejadas de la línea central del modelo estamos aplicando una clasificación de tipo **hard margin**. Esta opción solamente funciona si los datos son linealmente separables. Además es un tipo de clasificador muy sensible a los **outliers**: en la figura 8 se ve el gráfico de la izquierda donde los datos no son linealmente separables, por tanto no es posible crear una SVM de tipo *hard margin*.

En el gráfico de la derecha los datos circulares tienen un *outlier* que provoca que la SVM tenga unos carriles muy pequeños. Esto hace que el modelo tenga *overfitting* y no generalice bien, tendrá *bias* alto porque su tendencia será clasificar como círculos, ejemplos que en realidad sean cuadrados.

Para evitar estos inconvenientes es preferible usar un modelo más complejo. El objetivo es encontrar un equilibrio entre mantener una carretera con carriles tan anchos como sea posible y limitar la cantidad de violaciones del margen (instancias que acaban quedando dentro de la carretera incluso en el lado incorrecto de la misma). A esta opción se la conoce como **SVM de soft-margen**.

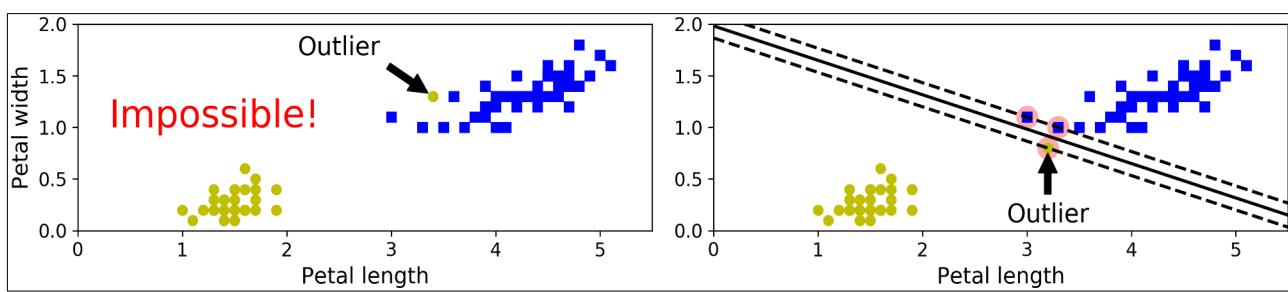


Figura 8. Las SVM hard-margin son sensibles a outliers.

En Scikit-Learn las clases **SVM** pueden controlar este balance usando **el hiperparámetro de nombre C**. Un valor pequeño permite carriles anchos y muchas violaciones. La figura 9 muestra una SVM con valor de C=1 a la izquierda. Estos valores se utilizan cuando los datasets no son linealmente separables. Como se aprecia, los márgenes son anchos y muchos ejemplos invaden los carriles.

Del otro lado, a la derecha aparece un modelo con un valor alto de C, lo que provoca acercarte a *hard-margin*. Quedan pocos datos dentro de los carriles, pero su anchura es menor y eso genera que el modelo probablemente tenga más errores a la hora de predecir.

Sin embargo en este ejemplo, incluso esta configuración ($C=100$) podría ser aceptable puesto que la mayoría de los datos que invaden los carriles quedan en el lado correcto.

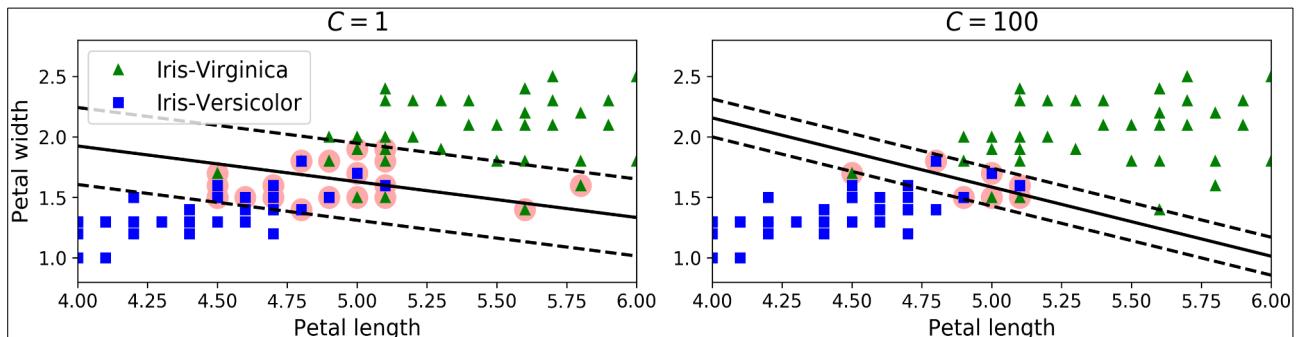


Figura 9: soft-margin a la izquierda (C bajo) vs hard-margin a la derecha (C alto).

Si tu modelo tiene *overfitting*, puedes regularizarlo reduciendo el valor de C . El siguiente ejemplo carga el dataset Iris, escala las características y entrena un modelo lineal SVM (usando la clase *LinearSVC* con $C=1$ y la función *hinge Loss*, que describimos más adelante) para clasificar flores Iris-Virginica. El modelo resultante aparece en la figura 9.

EJEMPLO 10: Crear y entrenar un modelo *LinearSVC*.

```
import numpy as np
from sklearn import datasets
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC
iris = datasets.load_iris()
X = iris["data"][:, (2, 3)]          # Longitud y anchura de pétalos
y = (iris["target"] == 2).astype(np.float64)    # Iris-Virginica
svm_clf = Pipeline([("scaler", StandardScaler()),
                    ("linear_svc", LinearSVC(C=1, loss="hinge")),
])
svm_clf.fit(X, y)
# Usar el modelo para hacer predicciones
print(svm_clf.predict([[5.5, 1.7]]))
```

Al contrario de los clasificadores de regresión logística, los clasificadores **SVM** no devuelven probabilidades para cada clase. Alternativamente podemos usar la clase **SVC**, usando **SVC(kernel="Linear", C=1)**, pero es mucho más lenta, especialmente con conjuntos de datos grandes, así que no es muy recomendable normalmente.

Otra posibilidad es usar la clase **SGDClassifier(loss="hinge", alpha=1/(m*C))**. Esto aplica descenso por gradiente estocástico para entrenar un clasificador lineal SVM. No converge tan rápido como la clase *LinearSVC*, pero puede utilizarse con grandes conjuntos de datos que incluso no caben en memoria (*out-of-core training*) o manejar tareas de clasificación online.

La clase **LinearSVC** regulariza el término *bias*, así que deberías centrar los datos de entrenamiento antes restando su media. Esto es automático cuando escalas los datos con **StandardScaler**. Además, debes asegurarte de que estableces el hiperparámetro *loss* a "*hinge*", porque no es el valor por defecto. Por último, para mejorar el rendimiento deberías poner el parámetro *dual* a *False*, salvo que haya más características que ejemplos.

2.2. SVM PARA CLASIFICACIÓN NO LINEAL.

Aunque los clasificadores *SVM* son eficientes y trabajan sorprendentemente bien en muchos casos, algunos datos no son linealmente separables.

En espacios de datos con pocas dimensiones los modelos lineales encuentran muchas limitaciones. Una manera de hacerlos más flexibles es aumentar la cantidad de dimensiones que tienen los datos, es decir añadir características que sean interacciones o polinomios de las existentes.

EJEMPLO 11: Creamos el fichero u03_util.py donde implementamos varias funciones de dibujo que usaremos en los siguientes ejemplos.

```
# FICHERO u03_util.py
# contiene varios métodos que se usarán en varios ejemplos
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap, colorConverter

cm2 = ListedColormap(['#0000aa', '#ff2020'])

def discrete_scatter(x1, x2, y=None, markers=None, s=10, ax=None,
                      labels=None, padding=.2, alpha=1, c=None, markeredgewidth=None):
    if ax is None:
        ax = plt.gca()
    if y is None:
        y = np.zeros(len(x1))
    unique_y = np.unique(y)
    if markers is None:
        markers = ['o', '^', 'v', 'D', 's', '*', 'p', 'h', 'H', '8', '<', '>'] * 10
    if len(markers) == 1:
        markers = markers * len(unique_y)
    if labels is None:
        labels = unique_y
    lineas = []
    current_cycler = mpl.rcParams['axes.prop_cycle']
    for i, (yy, cycle) in enumerate(zip(unique_y, current_cycler())):
        mask = y == yy
        if c is None: # si c es none, usa colores repetidos
            color = cycle['color']
        elif len(c) > 1:
            color = c[i]
        else:
            color = c
        if np.mean(colorConverter.to_rgb(color)) < .4: # frontera clara si marker oscuro
            markeredgecolor = "grey"
        else:
            markeredgecolor = "black"
        lineas.append(ax.plot(x1[mask], x2[mask], markers[i], markersize=s,
                              label=labels[i], alpha=alpha, c=color,
                              markeredgewidth=markeredgewidth,
                              markeredgecolor=markeredgecolor)[0])
    if padding != 0:
        pad1 = x1.std() * padding
        pad2 = x2.std() * padding
        xlim = ax.get_xlim()
        ylim = ax.get_ylimits()
        ax.set_xlim(min(x1.min()) - pad1, xlim[0]), max(x1.max() + pad1, xlim[1]))
        ax.set_ylimits(min(x2.min()) - pad2, ylim[0]), max(x2.max() + pad2, ylim[1]))
    return lineas

def plot_2d_separator(classifier, X, fill=False, ax=None, eps=None, alpha=1,
                      cm=cm2, linewidth=None, threshold=None, linestyle="solid"):
    if eps is None:
        eps = X.std() / 2.
    if ax is None:
        ax = plt.gca()
    x_min, x_max = X[:, 0].min() - eps, X[:, 0].max() + eps
    y_min, y_max = X[:, 1].min() - eps, X[:, 1].max() + eps
    xx = np.linspace(x_min, x_max, 1000)
    yy = np.linspace(y_min, y_max, 1000)
    X1, X2 = np.meshgrid(xx, yy)
    X_grid = np.c_[X1.ravel(), X2.ravel()]
```

```

try:
    decision_values = classifier.decision_function(X_grid)
    levels = [0] if threshold is None else [threshold]
    fill_levels = [decision_values.min()] + levels + [decision_values.max()]
except AttributeError:
    # no decision_function
    decision_values = classifier.predict_proba(X_grid)[:, 1]
    levels = [.5] if threshold is None else [threshold]
    fill_levels = [0] + levels + [1]
if fill:
    ax.contourf(X1, X2, decision_values.reshape(X1.shape),
                 levels=fill_levels, alpha=alpha, cmap=cmap)
else:
    ax.contour(X1, X2, decision_values.reshape(X1.shape), levels=levels,
               colors="black", alpha=alpha, linewidths=linewidth,
               linestyles=linestyle, zorder=5)
ax.set_xlim(x_min, x_max)
ax.set_ylim(y_min, y_max)
ax.set_xticks(())
ax.set_yticks(())

def plot_dataset(X, y, axes):
    plt.plot(X[:, 0][y==0], X[:, 1][y==0], "bs")
    plt.plot(X[:, 0][y==1], X[:, 1][y==1], "g^")
    plt.axis(axes)
    plt.grid(True, which='both')
    plt.xlabel(r"$x_1$", fontsize=14)
    plt.ylabel(r"$x_2$", fontsize=14, rotation=0)

def plot_predicciones(clf, axes):
    x0s = np.linspace(axes[0], axes[1], 100)
    x1s = np.linspace(axes[2], axes[3], 100)
    x0, x1 = np.meshgrid(x0s, x1s)
    X = np.c_[x0.ravel(), x1.ravel()]
    y_pred = clf.predict(X).reshape(x0.shape)
    y_decision = clf.decision_function(X).reshape(x0.shape)
    plt.contourf(x0, x1, y_pred, cmap=plt.cm.brg, alpha=0.2)
    plt.contourf(x0, x1, y_decision, cmap=plt.cm.brg, alpha=0.1)

```

EJEMPLO 12: Generar y dibujar datos linealmente no separables *en 2D*.

```

import numpy as np
import matplotlib.pyplot as plt
from u03_tools import discrete_scatter, plot_2d_separator, cm2
from sklearn.datasets import make_blobs
X, y = make_blobs(centers=4, random_state=8)
y = y % 2
discrete_scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Característica 0")
plt.ylabel("Característica 1")  # plt.plot()

```

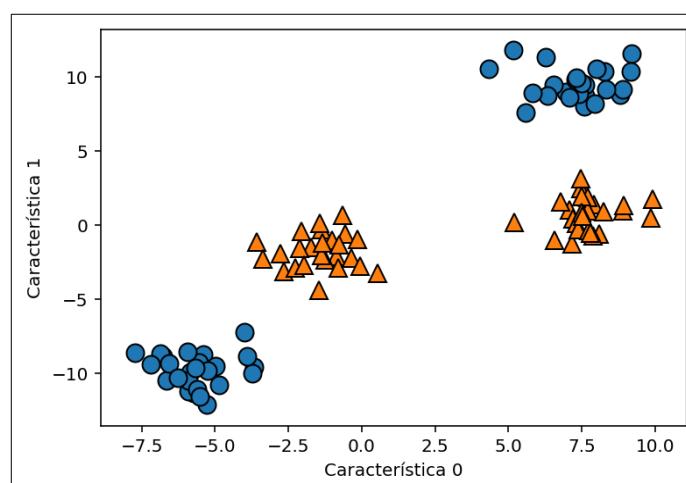


Figura 10. Datos con dos clases que no son linealmente separables.

Un modelo lineal como SVM solamente separa ejemplos usando una línea (hiperplano en realidad) y no tendrá mucha habilidad con datasets como el que acabamos de generar.

EJEMPLO 13: Crear y entrenar un modelo clasificador *SVM* para los datos del EJEMPLO 12. Añade:

```
from sklearn.svm import LinearSVC
lineal_svm = LinearSVC().fit(X, y)
plot_2d_separator(lineal_svm, X)
discrete_scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Característica 0")
plt.ylabel("Característica 1")
```

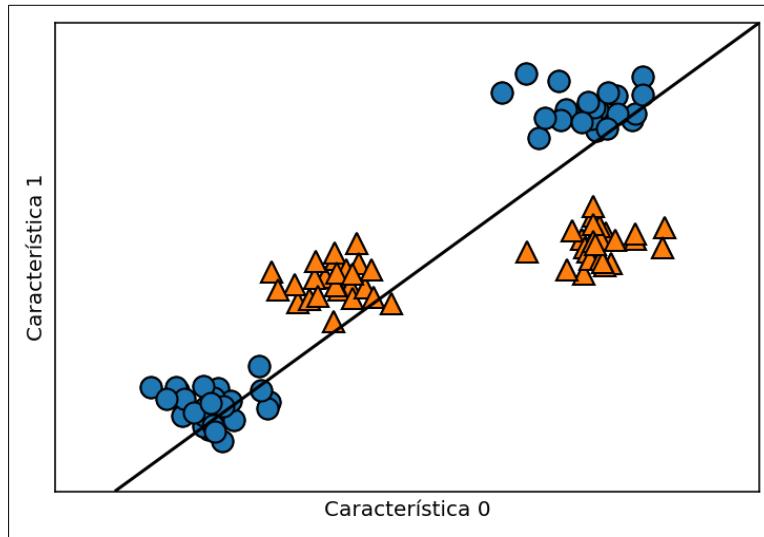


Figura 11: Frontera de decisión calculada por un clasificador SVM lineal.

Ahora expandimos las características de los datos, por ejemplo con otra nueva característica que sea la característica1 elevada al cuadrado (`característica1 ** 2`). Ahora los datos tienen tres dimensiones, han pasado de dos dimensiones (`característica0, característica1`) a 3 dimensiones (`característica0, característica1, característica1**2`).

EJEMPLO 14: Crear y entrenar un modelo clasificador *SVM* para los datos del EJEMPLO 12. Añade este código

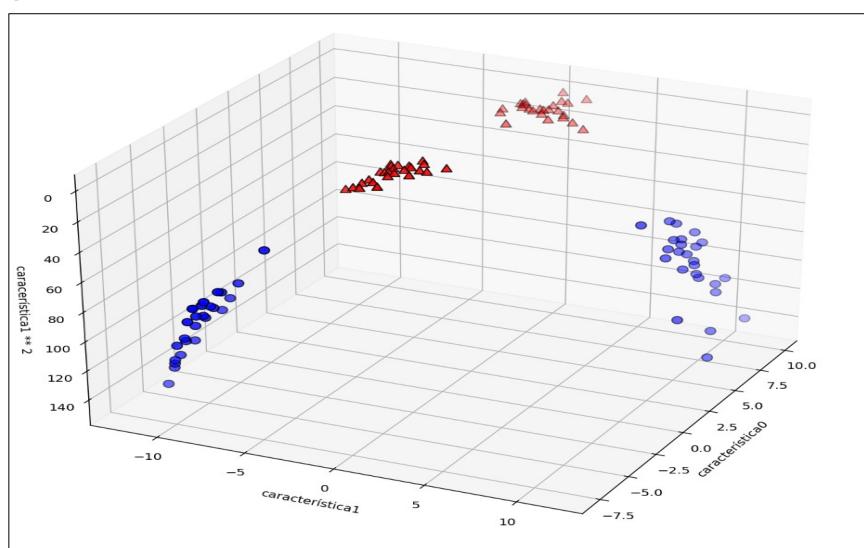


Figura 11: Expansión de un dataset 2D a 3D añadiendo característica derivada.

```
# Añadimos la nueva característica
X_new = np.hstack( [X, X[:, 1:] ** 2] )
ax = plt.figure(figsize=(12,11)).add_subplot(projection='3d')
```

```
# Dibujar primero todos los puntos de la clase y == 0, y luego y == 1
mascara = y == 0
ax.scatter(X_new[mascara, 0], X_new[mascara, 1], X_new[mascara, 2], c='b', cmap= cm2,
           s=60, edgecolor='k')
ax.scatter(X_new[~mascara, 0], X_new[~mascara, 1], X_new[~mascara, 2], c='r', marker='^',
           cmap=cm2, s=60, edgecolor='k')
ax.set_xlabel("característica0")
ax.set_ylabel("característica1")
ax.set_zlabel("caracerística1 ** 2")
ax.view_init(elev=-152., azim=-26, roll=0) # plt.show()
```

Con esta nueva separación de los datos, ahora es posible separar las dos clases mediante un plano 3D, que podemos dibujar también para visualizarlo.

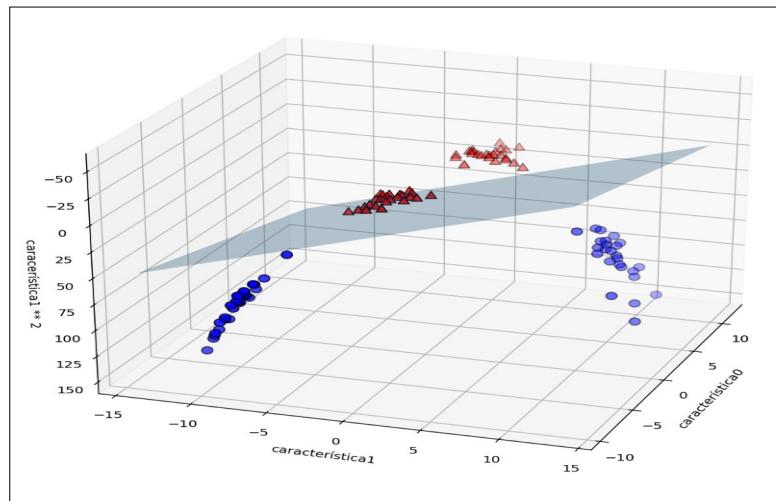


Figura 12. Plano de decisión encontrado por la SVM lineal en el dataset expandido a 3D.

EJEMPLO 15: Dibujar el plano 3D que separa los datos de las dos clases.

```
lineal_svm_3d = LinearSVC().fit(X_new, y)
coef, intercept = lineal_svm_3d.coef_.ravel(), lineal_svm_3d.intercept_
# Mostrar la frontera de decisión
ax = plt.figure(figsize=(10,11)).add_subplot(projection='3d')
xx = np.linspace(X_new[:, 0].min() - 2, X_new[:, 0].max() + 2, 50)
yy = np.linspace(X_new[:, 1].min() - 2, X_new[:, 1].max() + 2, 50)
XX, YY = np.meshgrid(xx, yy)
ZZ = (coef[0] * XX + coef[1] * YY + intercept) / -coef[2]
ax.plot_surface(XX, YY, ZZ, rstride=8, cstride=8, alpha=0.3)
ax.scatter(X_new[mascara, 0], X_new[mascara, 1], X_new[mascara, 2], c='b',
           cmap=cm2, s=60, edgecolor='k')
ax.scatter(X_new[~mascara, 0], X_new[~mascara, 1], X_new[~mascara, 2], c='r',
           marker='^', cmap=cm2, s=60, edgecolor='k')
ax.set_xlabel("característica0")
ax.set_ylabel("característica1")
ax.set_zlabel("caracerística1 ** 2")
ax.view_init(elev=-156., azim=-20, roll=0) # plt.show()
```

Como usamos una función de las características originales, el modelo SVM lineal ya no es lineal. No es una línea, sino más bien una elipse, como se aprecia en la figura 13 donde dibujamos las superficies de decisión en 2D.

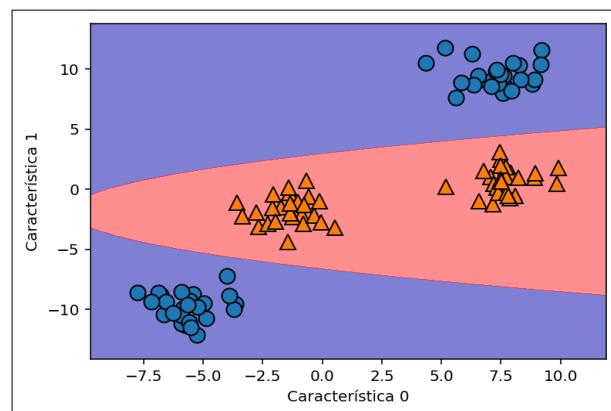


Figura 13. Superficies de decisión de la SVM lineal en 2D.

EJEMPLO 16: Dibujar las superficies de decisión en 2D.

```
ZZ = YY ** 2
dec = lineal_svm_3d.decision_function(np.c_[XX.ravel(), YY.ravel(), ZZ.ravel()])
plt.contourf(XX, YY, dec.reshape(XX.shape), levels=[dec.min(), 0, dec.max()], cmap= cm2,
alpha=0.5)
discrete_scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Característica 0")
plt.ylabel("Característica 1") # plt.show()
```

EL TRUCO DEL KERNEL

La moraleja que hemos aprendido: añadir características no lineales a los datos, hacen que los modelos lineales sean más potentes. Sin embargo, con frecuencia ocurre que no sabemos qué características añadir y añadir muchas (como poner todas las combinaciones de un espacio de dastos 100-dimensional) puede hacer que el entrenamiento del modelo sea muy costoso.

Por suerte, tenemos un truco matemático clave que nos va a permitir que el clasificador aprenda en un espacio de datos de altas dimensiones sin tener que calcular una nueva (posiblemente grande) representación de los datos. **Se conoce como el truco del kernel** y funciona calculando directamente la distancia (productos escalares) de los ejemplos en su representación extendida, sin haber tenido que calcular realmente la expansión.

Hay dos formas de mapear los datos originales en espacios de más dimensiones usados frecuentemente con SVM:

- El **kernel polinomial**: que calcula todos los posibles polinomios de las características originales hasta alcanzar cierto grado (como $\text{característica}^1 * \text{característica}^2$ ⁵).
- El **kernel de función de base radial (RBF)**, también conocido como **kernel Gausiano**. Este es más difícil de explicar porque se corresponde con un espacio de características de dimensiones infinitas. Una forma de imaginarlo es como si fuese un método polinomial que considera todos los posibles polinomios de todos los grados pero la importancia de las características decrece para los grados altos⁵.

No conocer los detalles matemáticos que hay detrás de los kernel usados con SVM en la práctica no impide usarlos. Básicamente, durante el entrenamiento, el algoritmo SVM aprende de cada ejemplo de entrenamiento cómo es de importante para definir la frontera de decisión. En general, solamente un subconjunto de los puntos son los más decisivos, los que están en el borde de la frontera de decisión y por tanto hacen de frontera entre las clases. Estos se llaman **vectores soporte** y son los que dan nombre al modelo.

Al hacer una predicción para un nuevo punto, se mide la distancia con cada uno de los vectores soporte. La decisión de como se clasifica el dato se realiza basándose en estas distancias con los vectores soporte y la importancia de estos vectores soporte que se aprendió durante el entrenamiento (estos datos están almacenados en el atributo **dual_coef_** del objeto SVC). Las distancias entre los puntos de datos la mide el kernel Gausiano de esta manera:

$$k_{rbf}(x_1, x_2) = \exp(-\gamma \|x_1 - x_2\|^2)$$

Donde x_1 y x_2 son dos datos, $\|x_1 - x_2\|$ es la distancia euclídea entre ellos y γ (gamma) es un parámetro que controla la anchura del kernel Gausiano. El centro de la función exponencial está marcada por el vector ele (l) que se denomina marca (*landmark*) de la función (destacado en círculo rojo en el gráfico de la izquierda). El radio de la base de la montaña se denomina constante sigma. Estos dos valores se ponen en la ecuación.

La figura 15 muestra el resultado de entrenar una SVC en un dataset 2D de dos clases. La frontera de decisión se muestra en negro y los vectores soporte son los puntos con mayor borde. En este caso, es la frontera es una curva muy suave (no es una línea recta).

⁵ Esto proviene de la expansión de Taylor de la exponencial.

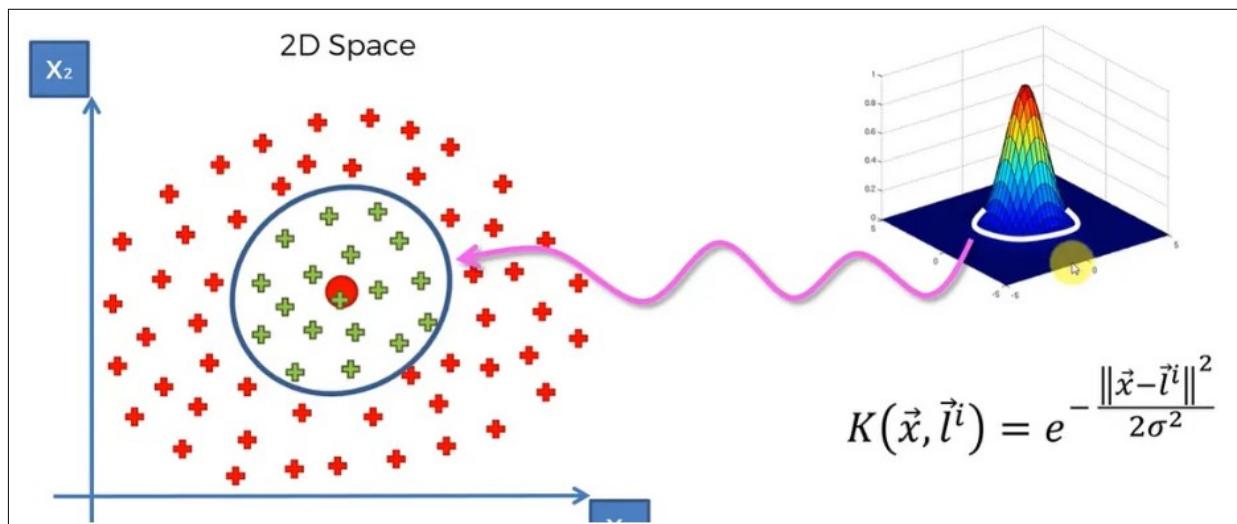


Figura 14. Kernel RBF de SVC.

EJEMPLO 17: Crear una SVC, entrenarla y mostrar su frontera de decisión.

En primer lugar añade esta función al fichero U03_tools.py.

```
def crea_dataset_revuelto():
    X, y = make_blobs(centers=2, random_state=4, n_samples=30)
    y[np.array([7, 27])] = 0
    mascara = np.ones(len(X), dtype=bool)
    mascara[np.array([0, 1, 5, 26])] = 0
    X, y = X[mascara], y[mascara]
    return X, y
```

Ahora crea el fichero U03_Ejemplo14.py:

```
# -*- coding: utf-8 -*-
from sklearn.svm import SVC
import matplotlib.pyplot as plt
from u03_tools import crea_daset_revuelto, discrete_scatter, plot_2d_separator

X, y = crea_dataset_revuelto()
svm = SVC(kernel='rbf', C=10, gamma=0.1).fit(X, y)
plot_2d_separator(svm, X, eps=.5)
discrete_scatter(X[:, 0], X[:, 1], y)
# Dibujar los vectores soporte
sv = svm.support_vectors_
# La clase de los vectores soporte lo indica el signo de los coeficientes duales
sv_labels = svm.dual_coef_.ravel() > 0
discrete_scatter(sv[:, 0], sv[:, 1], sv_labels, s=15, markeredgewidth=3)
plt.xlabel("Característica 0")
plt.ylabel("Característica 1")
```

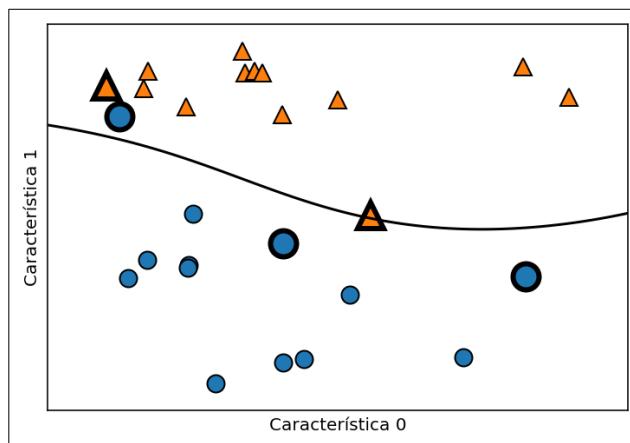


Figura 15. Frontera de decisión y vectores soporte encontrados por una SVM con kernel RBF.

AÑADIR COLUMNAS PARA SOLUCIONAR LA NO SEPARABILIDAD LINEAL

Para solucionar el problema de los datos no separables linealmente se añaden más características (características polinomiales) que permitan separar las clases con un hiperplano en una mayor dimensionalidad.

Considera el dibujo izquierdo de la figura 16 que representa un dataset con una sola característica x_1 . Hay dos clases (cuadrados azules y triángulos verdes) y no son linealmente separables. Si añadimos una segunda característica $x_2 = (x_1)^2$, el resultado es un dataset en 2D donde las clases son perfectamente separables por una línea recta tal y como se ve en el gráfico de la derecha.

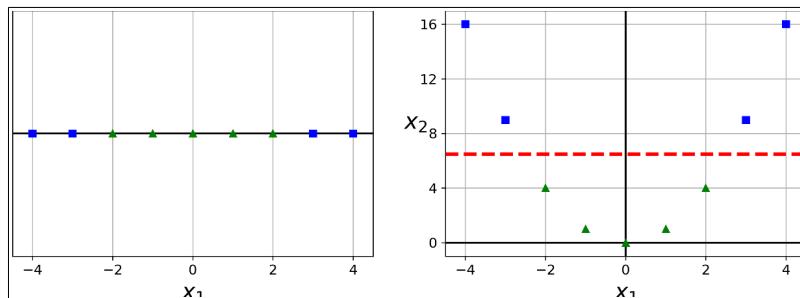


Figura 16. Añadir características para hacer linealmente separable un dataset.

Para implementar la idea en *Scikit-Learn*, puedes crear un Pipeline que contenga un transformador **PolynomialFeatures** seguido de un **StandardScaler** y un **LinearSVC**. Vamos a probar esto en el dataset *moons* de *Scikit-Learn* que es un dataset de prueba para clasificación binaria con los datos distribuidos en dos semicírculos que se encajan. Puedes generar este dataset usando la función *make_moons()*.

EJEMPLO 18: Crear una SVC y añadir características polinomiales antes de entrenarla.

```
# -*- coding: utf-8 -*-
from sklearn.datasets import make_moons
import matplotlib.pyplot as plt
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.svm import LinearSVC
from u03_tools import plot_dataset, plot_predicciones

X, y = make_moons(n_samples=100, noise=0.15, random_state=42)
svm_poli = Pipeline([
    ("poly_features", PolynomialFeatures(degree=3)),
    ("scaler", StandardScaler()),
    ("svm_clf", LinearSVC(C=10, loss="hinge"))
])
svm_poli.fit(X, y)
plot_predicciones(svm_poli, [-1.5, 2.5, -1, 1.5])
plot_dataset(X, y, [-1.5, 2.5, -1, 1.5])
plt.show()
```

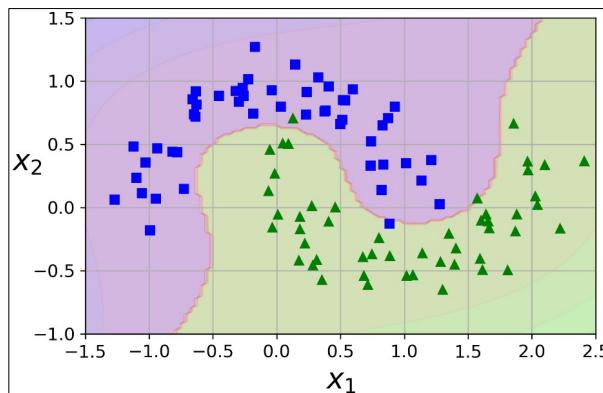


Figura 17. Clasificador lineal SVM usando característica no polinomial.

USAR KERNEL POLINOMIAL PARA SOLUCIONAR LA NO SEPARABILIDAD LINEAL

Añadir columnas polinomiales es sencillo y puede funcionar bien con la mayoría de los algoritmos de clasificación (no solo con *SVM*), pero cuando grado del polinomio es bajo con puede solucionar el problema en conjuntos de datos complejos, y cuando es alto la cantidad de nuevas características hace que el modelo sea lento de entrenar y utilizar.

Usando un *kernel polinomial* consigues los mismos resultados pero sin la necesidad de añadir nuevas características. Por tanto eliminas la multiplicación explosiva de características. Esta técnica está implementada en la propia clase *SVC*.

EJEMPLO 19: Volvemos a usar el dataset *moons* pero ahora usando un kernel polinomial.

```
from sklearn.datasets import make_moons
import matplotlib.pyplot as plt
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC

X, y = make_moons(n_samples=100, noise=0.15, random_state=42) # Crea dataset
# Definir y entrenar un SVC con kernel polinomial de grado 3, coef0 1 y C=5
kernel_poli = Pipeline([
    ("scaler", StandardScaler()),
    ("svm_clf", SVC(kernel="poly", degree=3, coef0=1, C=5))
])
kernel_poli.fit(X, y)
# Definir y entrenar un SVC con kernel polinomial de grado 10, coef0 100 y C=5
kernel_poli100 = Pipeline([
    ("scaler", StandardScaler()),
    ("svm_clf", SVC(kernel="poly", degree=10, coef0=100, C=5))
])
kernel_poli100.fit(X, y)
# Dibujar las superficies de clasificación de ambos
fig, axes = plt.subplots(ncols=2, figsize=(10.5, 4), sharey=True)
plt.sca(axes[0])
plot_predicciones(kernel_poli, [-1.5, 2.45, -1, 1.5])
plot_dataset(X, y, [-1.5, 2.4, -1, 1.5])
plt.title(r"$grado=3, coef0=1, C=5$", fontsize=14)
plt.sca(axes[1])
plot_predicciones(kernel_poli100, [-1.5, 2.45, -1, 1.5])
plot_dataset(X, y, [-1.5, 2.4, -1, 1.5])
plt.title(r"$grado=10, coef0=100, C=5$", fontsize=14)
plt.ylabel("") # plt.show()
```

El código del ejemplo 19 entrena un clasificador *SVM* usando un *kernel polinomial* de grado 3. Es el gráfico izquierdo de la figura 18. El de la derecha es otro clasificador *SVM* con *kernel polinomial* de grado 10. Si tu modelo tiene *overfitting*, debes bajar el grado del polinomio, pero si tiene *underfitting* debes incrementarlo. El hiperparámetro **coef0** controla como es influenciado el modelo por los términos de alto grado en contra de los términos de bajo grado.

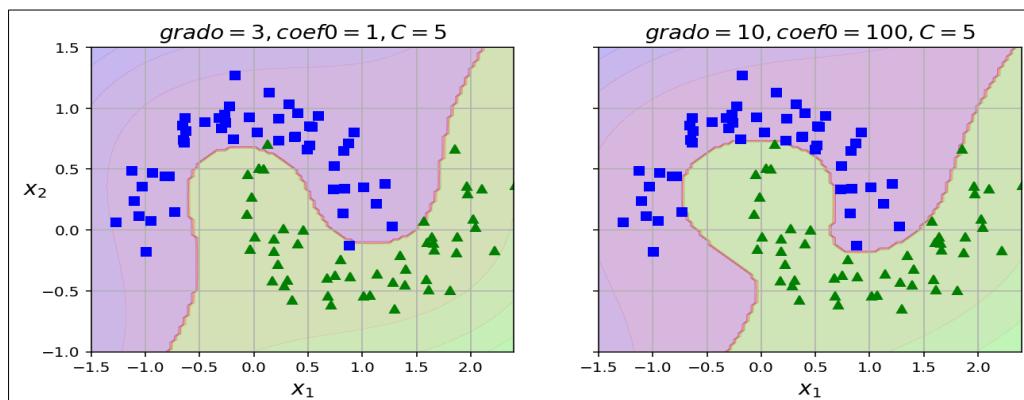


Figura 18. Clasificadores SVM con kernel polinomial.

Una aproximación para encontrar los valores correctos de los hiperparámetros es usar *grid search*. Normalmente es más rápido hacer una primera búsqueda con brocha gorda y luego afinar con una búsqueda cerca de los mejores valores encontrados en la primera búsqueda. Entender bien lo que hace cada hiperparámetro ayuda a realizar una búsqueda en los lugares correctos.

AÑADIR CARACTERÍSTICAS POR SIMILARIDAD

Otra técnica para resolver la no separabilidad lineal de los datos es añadir características calculados usando una función de similaridad que mida como cada ejemplo de datos influye en separar dos datos usados como marcas. Vamos a usar el dataset unidimensional usado anteriormente y vamos a definir dos marcas en $x_1=-2$ y $x_1=1$ (como se ve en la figura 19). A continuación usamos como función de similitud la *Gausiana Radial Basis Function (RBF)* con $\gamma = 0.3$.

$$\phi_\gamma(x, \ell) = \exp(-\gamma \|x - \ell\|^2)$$

La función tiene una forma acampanada que cambia de 0 (cuando está muy alejada de la marca) hasta 1 (cuando está sobre la marca). Para calcular la nueva característica nos fijamos en que $x_1 = -1$ está a una distancia 1 de la primera marca y a 2 de la segunda. Así que su nueva característica será $x_2 = \exp(-0.3 \times 1^2) \approx 0.74$ y $x_3 = \exp(-0.3 \times 2^2) \approx 0.30$. El gráfico de la derecha de la figura 19 muestra el dataset transformado sin características originales y ahora son separables linealmente.

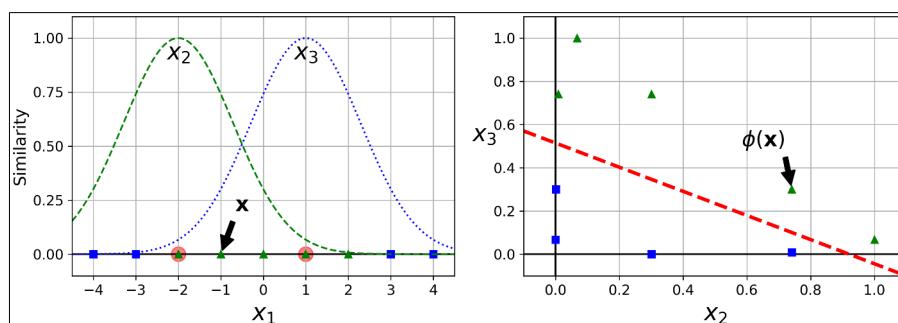


Figura 19. Similitud de características usando la función Gausiana RBF.

La pregunta que queda pendiente es ¿Cómo elegir las marcas? La aproximación más sencilla es crear una marca en cada una de las localizaciones del dataset. Pero esto genera muchas dimensiones y casi tendrás garantizada la separabilidad lineal. Pero el inconveniente de esto es que transformas un dataset con m ejemplos y n características en otro con m ejemplos y m características (asumiendo que quitas las características originales). Si tus datos de entrenamiento son muy grandes, acabas con un número de características excesivo.

USAR KERNEL GAUSIANO RBF

Los métodos polinomiales o las funciones de similaridad se pueden usar con cualquier algoritmo clasificador de machine learning, pero pueden ser computacionalmente exigentes. Sin embargo, el truco de los *kernel* de SVM nos permite obtener los mismos resultados sin ese coste computacional.

En el ejemplo 20 representado en el gráfico de abajo a la izquierda de la figura 19 definimos un modelo que podemos comparar con otros que tienen otros valores de los hiperparámetros γ (γ) y C. Al incrementar γ la forma acampanada se reduce y como consecuencia su rango de influencia se hace menor de manera que la frontera de decisión acaba siendo más irregular rodeando incluso cada ejemplo de datos individual.

EJEMPLO 20: Con el dataset *moons* usamos kernel RBF.

```
from sklearn.datasets import make_moons
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
import matplotlib.pyplot as plt
from u03_tools import plot_dataset, plot_predicciones

X, y = make_moons(n_samples=100, noise=0.15, random_state=42) # Crea dataset
```

```

gamma1, gamma2 = 0.1, 5
C1, C2 = 0.001, 1000
hiperparametros = (gamma1, C1), (gamma1, C2), (gamma2, C1), (gamma2, C2)

svm_clfs = []
for gamma, C in hiperparametros:
    svm_kernel_rbf = Pipeline([
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="rbf", gamma=gamma, C=C))
    ])
    svm_kernel_rbf.fit(X, y)
    svm_clfs.append(svm_kernel_rbf)

fig, ejes = plt.subplots(nrows=2, ncols=2, figsize=(10.5, 7), sharex=True, sharey=True)
for i, svm_clf in enumerate(svm_clfs):
    plt.sca(ejes[i // 2, i % 2])
    plot_predicciones(svm_clf, [-1.5, 2.45, -1, 1.5])
    plot_dataset(X, y, [-1.5, 2.45, -1, 1.5])
    gamma, C = hiperparametros[i]
    plt.title(r"$\gamma = {}$, $C = {}$".format(gamma, C), fontsize=16)
    if i in (0, 1):
        plt.xlabel("")
    if i in (1, 3):
        plt.ylabel("")

```

Un valor pequeño de γ hace que la forma de campana se extienda y los ejemplos tienen mayor influencia lo que genera que la frontera de decisión se suavice. Así que γ actúa como un hiperparámetro de regularización: si el modelo tiene *overfitting* se reduce γ y si tiene *underfitting* se incrementa γ (de manera similar el hiperparámetro C).

Existen otros *kernel* pero no se usan tanto. Por ejemplo hay *kernel* especializados en estructuras de datos concretas como los kernels de Strings que se usan para clasificar documentos de secuencias de ADN (*kernel de subsecuencias de string* o *kernel basado en distancia de Levenshtein*).

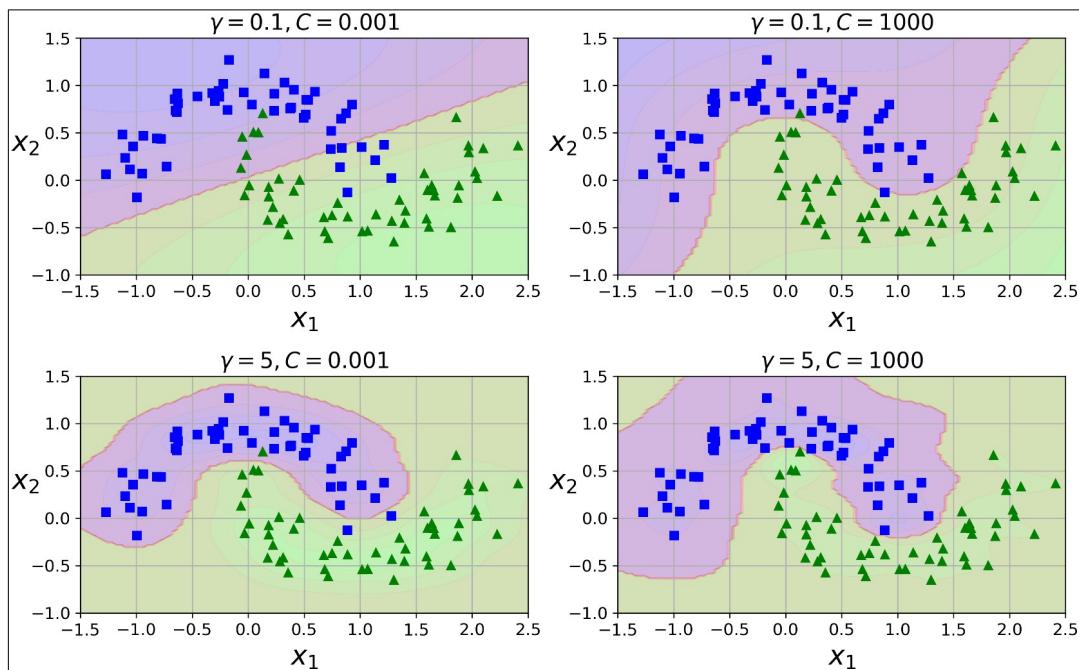


Figura 20. Clasificadores SVM usando kernel RBF.

¿Qué *kernel* usar de los que hay disponibles? La regla es usar primero el lineal (recuerda que *LinearSVC* es mucho más rápido que *SVC(kernel="linear")*), especialmente si los datos de entrenamiento son muy grandes o tienen muchas características. Si los datos de entrenamiento no son demasiado grandes deberías usar el *kernel gausiano RBF* que funciona bien en la mayoría de los casos.

Luego si tienes tiempo y equipos con potencia de cálculo, también puedes experimentar con otros utilizando validación cruzada y *grid-search*, sobre todo si tienes *kernel* especializados en las estructuras de datos que son tus datos de entrenamiento.

Una nota sobre la complejidad computacional: la clase **LinearSVC** se apoya en la librería **liblinear**, que implementa un algoritmo optimizado para SVM lineal⁶. No tiene soporte para el truco del *kernel*, pero escala linealmente cuando aumenta la cantidad de ejemplos, así que su complejidad es $O(m \times n)$. Pero si necesitas mucha precisión en el resultado tardará más tiempo en encontrar la solución. Esto lo controla el **hiperparámetro de tolerancia ϵ** (llamado **tol** en Scikit-Learn). En la mayoría de los problemas de clasificación la tolerancia por defecto es correcta.

La clase **SVC** se apoya en la librería **LIBSVM**, que implementa un algoritmo que soporta *kernel*⁷. La complejidad del tiempo de entrenamiento está entre $O(m^2 \times n)$ y $O(m^3 \times n)$. Esto significa que es mucho más lento cuando la cantidad de ejemplos crece (cientos o miles de instancias). Este algoritmo es perfecto para datos de entrenamiento complejos y de tamaño medio o pequeño. Sin embargo escala bien cuando aumenta la cantidad de características, especialmente con *sparse features* (es decir, cuando cada instancia tiene unas cuantas características que no son cero), en ese caso el algoritmo escala a razón del número de características no cero por ejemplo. La siguiente tabla recoge un resumen de lo que comentamos:

Algoritmo	Entrenamiento	Out-of-core	Datos Escalados	Truco del Kernel
LinearSVC	$O(m \times n)$	No	Yes	No
SGDClassifier	$O(m \times n)$	Yes	Yes	No
SVC	$O(m^2 \times n)$ to $O(m^3 \times n)$	No	Yes	Yes

2.3. SVM PARA REGRESIÓN (SVR).

El algoritmo SVM además de soportar clasificaciones lineales y no lineales, también permite realizar regresiones además de clasificaciones. Las regresiones pueden ser lineales o no lineales. Este algoritmo fue propuesto por Vapnik en 1998.

Para conseguirlo lo que hace es darle la vuelta a su función objetivo: en vez de intentar encontrar los carriles lo más alejados que sea posible de las dos clases mientras limita la cantidad de violaciones de estos márgenes, SVR (la versión de SVM para regresión) intenta dejar la mayor cantidad posible de datos dentro de los carriles mientras limita la cantidad de ejemplos que violan la frontera del carril central o se quedan fuera de los carriles.

El ancho de los carriles lo controla el parámetro de **insensitividad ϵ** . La figura 20 muestra dos modelos de regresión lineal SVM entrenados con algunos datos lineales aleatorios, el primero con un margen grande ($\epsilon = 1.5$) y el otro configurado con un margen más pequeño ($\epsilon = 0.5$).

EJEMPLO 21: Regresión lineal con SVM.

```
import numpy as np
from sklearn.svm import LinearSVR
import matplotlib.pyplot as plt
# definir los datos de entrenamiento
np.random.seed(42)
m = 50
X = 2 * np.random.rand(m, 1)
y = (4 + 3 * X + np.random.randn(m, 1)).ravel()
# Entrenar los SVM para regresión
svm_reg1 = LinearSVR(epsilon=1.5, random_state=42)
svm_reg2 = LinearSVR(epsilon=0.5, random_state=42)
svm_reg1.fit(X, y)
svm_reg2.fit(X, y)
```

6 El método conocido como "Dual Coordinate Descent Method for Large-scale Linear SVM", Lin et al. (2008).

7 "Sequential Minimal Optimization (SMO)," J. Platt (1998).

```

def encontrar_vectores_soporte(svm_reg, X, y):
    y_pred = svm_reg.predict(X)
    fuera_de_margen = (np.abs(y - y_pred) >= svm_reg.epsilon)
    return np.argwhere(fuera_de_margen)

svm_reg1.support_ = encontrar_vectores_soporte(svm_reg1, X, y)
svm_reg2.support_ = encontrar_vectores_soporte(svm_reg2, X, y)
eps_x1 = 1
eps_y_pred = svm_reg1.predict([[eps_x1]])

def plot_svm_regresion(svm_reg, X, y, ejes): # Puedes llevarte esta a U03_tools.py
    x1s = np.linspace(ejes[0], ejes[1], 100).reshape(100, 1)
    y_pred = svm_reg.predict(x1s)
    plt.plot(x1s, y_pred, "k-", linewidth=2, label=r"\hat{y}")
    plt.plot(x1s, y_pred + svm_reg.epsilon, "k--")
    plt.plot(x1s, y_pred - svm_reg.epsilon, "k--")
    plt.scatter(X[svm_reg.support_], y[svm_reg.support_], s=180, facecolors='#FFAAAA')
    plt.plot(X, y, "bo")
    plt.xlabel(r"$x_1$", fontsize=18)
    plt.legend(loc="upper left", fontsize=18)
    plt.axis(ejes)

fig, ejes = plt.subplots(ncols=2, figsize=(9, 4), sharey=True)
plt.sca(ejes[0])
plot_svm_regresion(svm_reg1, X, y, [0, 2, 3, 11])
plt.title(r"\epsilon = {}".format(svm_reg1.epsilon), fontsize=16)
plt.ylabel(r"$y$", fontsize=16, rotation=0)
plt.annotate(' ', xy=(eps_x1, eps_y_pred), xycoords='data',
            xytext=(eps_x1, eps_y_pred - svm_reg1.epsilon),
            textcoords='data', arrowprops={'arrowstyle': '<->', 'linewidth': 1.5})
plt.text(0.91, 5.6, r"\epsilon", fontsize=18)
plt.sca(ejes[1])
plot_svm_regresion(svm_reg2, X, y, [0, 2, 3, 11])
plt.title(r"\epsilon = {}".format(svm_reg2.epsilon), fontsize=16) # plt.show()

```

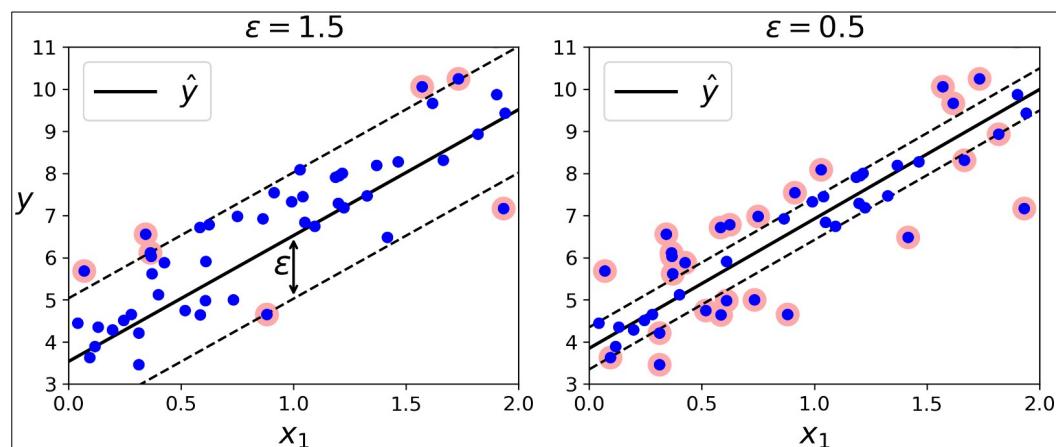


Figura 21. SVM de Regresión Lineal con dos valores del hiperparámetro

Si añadimos más ejemplos de entrenamiento dentro del margen, el modelo no cambia, por eso recibe el adjetivo de **ϵ -insensitivo**.

También puedes usar la clase **LinearSVR** de *Scikit-Learn* para realizar regresiones lineales. El siguiente ejemplo la usa y genera el mismo gráfico izquierdo de la figura 21 pero antes de entrenar los datos deben ser centrados y escalados.

```

from sklearn.svm import LinearSVR
svm_reg = LinearSVR(epsilon=1.5)
svm_reg.fit(X, y)

```

También podemos realizar tareas de regresión no lineal usando un modelo SVM kernelizado. Por ejemplo, en la figura 22 aparecen unos datos que siguen una distribución aleatoria cuadrática. El

gráfico de la izquierda tiene poca regularización (C tiene un valor alto) y el de la derecha tiene mucha más regularización (valor de C más bajo).

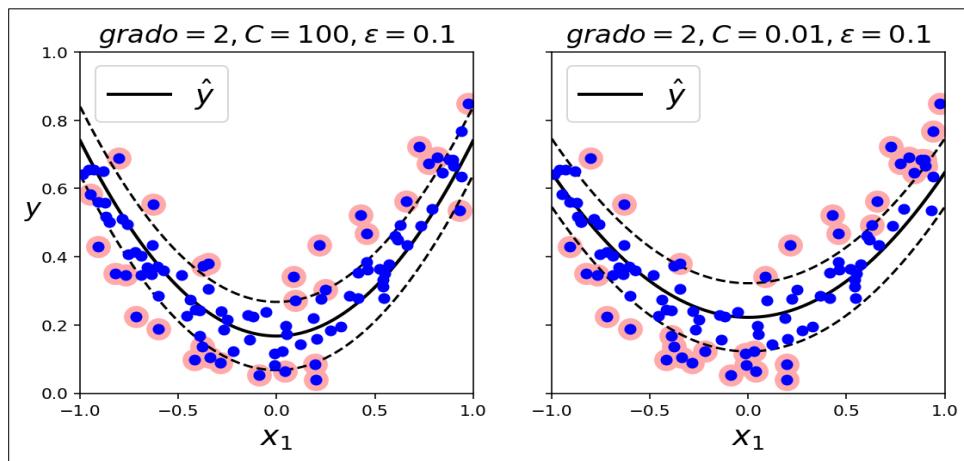


Figura 22. SVM para regresión lineal usando un kernel polinomial de grado 2.

La clase `LinearSVR` es para la regresión lo que `LinearSVC` es para la clasificación y también soporta kernels. Escala linealmente con el tamaño de los datos de entrenamiento (como la clase `LinearSVC`), mientras que es muy lenta cuando la cantidad de datos aumenta mucho (como la `LinearSVC`).

EJEMPLO 22: Añade estas sentencias detrás de las últimas del EJEMPLO 21.

```
from sklearn.svm import SVR

np.random.seed(42) # Generar datos
m = 100
X = 2 * np.random.rand(m, 1) - 1
y = (0.2 + 0.1 * X + 0.5 * X**2 + np.random.randn(m, 1)/10).ravel()
# Entrenar modelos SVR
svr_poli1 = SVR(kernel="poly", degree=2, C=100, epsilon=0.1, gamma="scale")
svr_poli2 = SVR(kernel="poly", degree=2, C=0.01, epsilon=0.1, gamma="scale")
svr_poli1.fit(X, y)
svr_poli2.fit(X, y)
# Dibujar
fig, ejes = plt.subplots(ncols=2, figsize=(9, 4), sharey=True)
plt.sca(ejes[0])
plot_svm_regresion(svr_poli1, X, y, [-1, 1, 0, 1])
plt.title(r"$grado={}$, $C={}$, $\epsilon={}$".format(svr_poli1.degree, svr_poli1.C,
svr_poli1.epsilon), fontsize=16)
plt.ylabel(r"$y$", fontsize=16, rotation=0)
plt.sca(ejes[1])
plot_svm_regresion(svr_poli2, X, y, [-1, 1, 0, 1])
plt.title(r"$grado={}$, $C={}$, $\epsilon={}$".format(svr_poli2.degree, svr_poli2.C,
svr_poli2.epsilon), fontsize=16) # plt.show()
```

La librería `scikit Learn` contiene tres clases de máquinas de soporte vectorial para regresión que son **SVR**, **NuSVR** y **LinearSVR**. **LinearSVR** ofrece una implementación más rápida que **SVR** pero solamente puede utilizar un kernel lineal mientras que **NuSVR** implementa una fórmula ligeramente distinta a las anteriores. **LinearSVR** utiliza **Liblinear** y también regulariza el punto de corte (*intercept*) si se indica. Los *scores* conseguidos por cada variante pueden ser diferentes.

Mientras que la clasificación intenta dejar fuera de los carriles la mayor parte de los datos, la regresión pretende hacer lo contrario. Es decir, un epsilon-SVR dados los n ejemplos como n vectores x_i de R^p (p predictoras) y el vector y de R^n , soluciona el problema ([mirar scikit](#)):

$$\min_{w,b,\zeta,\zeta^*} \frac{1}{2} w^T w + C \sum_{i=1}^n (\zeta_i + \zeta_i^*)$$

subject to $y_i - w^T \phi(x_i) - b \leq \varepsilon + \zeta_i$,
 $w^T \phi(x_i) + b - y_i \leq \varepsilon + \zeta_i^*$,
 $\zeta_i, \zeta_i^* \geq 0, i = 1, \dots, n$

Donde se penaliza aquellos ejemplos cuya predicción está más alejada que *epsilon* de su valor real. Por ejemplo **LinearSVR**, intenta minimizar esta expresión:

$$\min_{w,b} \frac{1}{2} w^T w + C \sum_{i=1}^n \max(0, |y_i - (w^T \phi(x_i) + b)| - \varepsilon),$$

2.4. SVM PARA DETECCIÓN DE OUTLIERS Y DE NOVELTIES.

Las SVM se utilizan normalmente para clasificación binaria aunque también se pueden utilizar para regresión y para detección de *outliers* en ciertos escenarios. Aquí aparecen algunas pautas para saber si es apropiado usarlas para esta tarea:

1. **Cuando los datos están bien definidos:** SVM funciona bien detectando datos anómalos cuando los puntos de datos están bien definidos y forman agrupamientos en el espacio de características. En este caso SVM es capaz de aprender un hiperplano que separa los datos normales de potenciales *outliers*.
2. **Espacio de Anomalías dispersas:** si los datos anómalos están dispersos con respecto a los datos normales, es decir, son relativamente raros y diferentes de los datos normales.
3. **Relaciones no lineales:** las SVM con funciones kernel como RBF pueden capturar relaciones no lineales complejas para detectar anomalías en datasets con alta dimensionalidad o mucha complejidad.
4. **Etiquetas de anomalías limitada:** SVM basada en detección de anomalías puede beneficiarse cuando has limitado o no etiquetado las anomalías durante el entrenamiento. **One-Class SVM**, es una variante de SVM, que aprende la distribución de los datos normales sin la necesidad de que las anomalías estén etiquetadas, esto hace que el algoritmo sea apropiado en la detección de anomalías con datasets no etiquetados, en aprendizaje no supervisado o semisupervisado.
5. **Robustez ante los valores atípicos:** SVM es relativamente robusto ante valores atípicos de los datos de entrenamiento si se usa el parámetro de regularización C para tener un equilibrio entre maximizar el margen y minimizar los errores de clasificación.
6. **Detección de anomalías binaria:** SVM está diseñado para tareas de clasificación binaria, lo que lo hace adecuado para detectar anomalías en escenarios donde el objetivo es clasificar los datos en anómalos y normales.
7. **Preprocesamiento e ingeniería de características:** las SVM para detectar anomalías pueden beneficiarse de un procesamiento cuidadoso de los datos y de técnicas de ingeniería de características que consigan aumentar la separación entre los datos normales y los anómalos. Por tanto le viene bien la normalización de los datos, la reducción de dimensionalidad y la eliminación de datos con *outliers*.

Sin embargo, tienen algunas limitaciones:

- **Escalabilidad:** SVM puede no tener un buen rendimiento con datasets grandes cuando los datos tienen alta dimensionalidad.
- **Tuning de hiperparámetros:** para que tenga un buen rendimiento SVM necesita hiperparámetros como el tipo de *kernel* o el parámetro de regularización *C* y los hiperparámetros del *kernel* usado como *gamma* en el caso de usar el *kernel RBF*.
- **Datos no balanceados:** Si SVM se enfrenta a datos muy imbalanceados donde las anomalías son extremadamente raras, necesita seleccionar cuidadosamente el hiperparámetro de regularización *C* o considerar usar pesos en las clases para funcionar correctamente.

LA CLASE ONE-CLASS SVM

Para detectar anomalías, **One-Class SVM** define un límite de puntuación para las anomalías. Los datos que tengan una puntuación por encima de este valor límite se consideran datos anómalos y los que tengan una puntuación inferior se consideran datos normales.

El límite puede ajustarse según el equilibrio que se necesite entre los falsos positivos (datos normales que se clasifiquen como anómalos) y los falsos negativos (anomalías que se clasifiquen como normales).

La clase **One-Class SVM** fue introducida por Schölkopf *et al*, y está implementada en el módulo *svm* de **Scikit-Learn** (para usarla debes ejecutar: `from sklearn import svm.OneClassSVM`) y necesitas indicar un *kernel* y el escalar que define la frontera de *outliers*. El kernel suele ser el *RBF* definiendo su *gamma* mediante algún cálculo sobre los datos porque no hay ninguna manera exacta de definirlo a priori. Otro parámetro es *nu*, conocido como *el margen* representa la probabilidad de encontrar una nueva observación normal fuera de la frontera. Una vez entrenado el modelo, debes usar la **`decision_function(x)`** para obtener como considera a los datos *x* (devuelve un valor negativo si los considera *outliers* o un valor positivo si los considera *inliers*).

Algunas aplicaciones necesitan decidir cuando una nueva observación que pertenece a la misma distribución de observaciones existentes (es un *inlier*), o debe considerarse como diferentes (es un *outlier*). Hay que hacer dos distinciones importantes:

- Detección de *outlier*: los datos de entrenamiento contienen *outliers* que son datos muy alejados de los otros. Los detectores deben definir regiones donde los datos de entrenamiento estén más concentrados e ignorar las observaciones alejadas.
- Detección de novedades: nos interesa detectar si un nuevo dato es un *outlier* pero los datos de entrenamiento no los tienen.

EJEMPLO 23: Detección de outliers usando One-class SVM.

```
import numpy as np
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt

# Generar datos de prueba y dibujarlos
X, y = make_blobs(n_features=2, centers=3, n_samples=500, random_state=123)
print("Formato de los datos de entrenamiento:", X.shape)
print("Primeros 3 datos:\n", X[:3,:])
plt.figure()
plt.scatter(X[:, 0], X[:, 1]) # plt.show()

# Detectar outliers
from sklearn.svm import OneClassSVM
nu = 0.05 # Suponemos que hay un 5% de datos anómalos
```

```

svm_oc = OneClassSVM(kernel='rbf', gamma=0.05, nu=nu)
svm_oc.fit(X)
X_outliers = X[svm_oc.predict(X) == -1]
print("Outliers detectados:\n", X_outliers)

# Definir rejilla para marcar la frontera de decisión
n_ejemplos, n_características = X.shape
X_range = np.zeros((n_características, 2))
X_range[:, 0] = np.min(X, axis=0) - 1.
X_range[:, 1] = np.max(X, axis=0) + 1.
h = 0.1 # Tamaño de paso de la rejilla
x_min, x_max = X_range[0]
y_min, y_max = X_range[1]
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
grid = np.c_[xx.ravel(), yy.ravel()]
# Dibujar la frontera y marcar de rojo los outliers
alfa_set = 0.95
Z_ocsvm = svm_oc.decision_function(grid)
Z_ocsvm = Z_ocsvm.reshape(xx.shape)
plt.figure()
c_0 = plt.contour(xx, yy, Z_ocsvm, levels=[0], colors='red', linewidths=3)
plt.clabel(c_0, inline=1, fontsize=12, fmt={0: str(alfa_set)})
plt.scatter(X[:, 0], X[:, 1])
plt.scatter(X_outliers[:, 0], X_outliers[:, 1], color='red')
plt.legend() # plt.show()
# En el one-class SVM, no todos los vectores soporte son outliers
X_SV = X[svm_oc.support_]
n_SV = len(X_SV)
n_outliers = len(X_outliers)
print('{0:.2f} <= {1:.2f} <= {2:.2f}'.format(
    1./n_ejemplos * n_outliers, nu, 1./n_ejemplos * n_SV))
# Marcamos los vectores soporte
plt.figure()
plt.contourf(xx, yy, Z_ocsvm, 10, cmap=plt.cm.Blues_r)
plt.scatter(X[:, 0], X[:, 1], s=1.)
plt.scatter(X_SV[:, 0], X_SV[:, 1], color='orange') # plt.show()

```

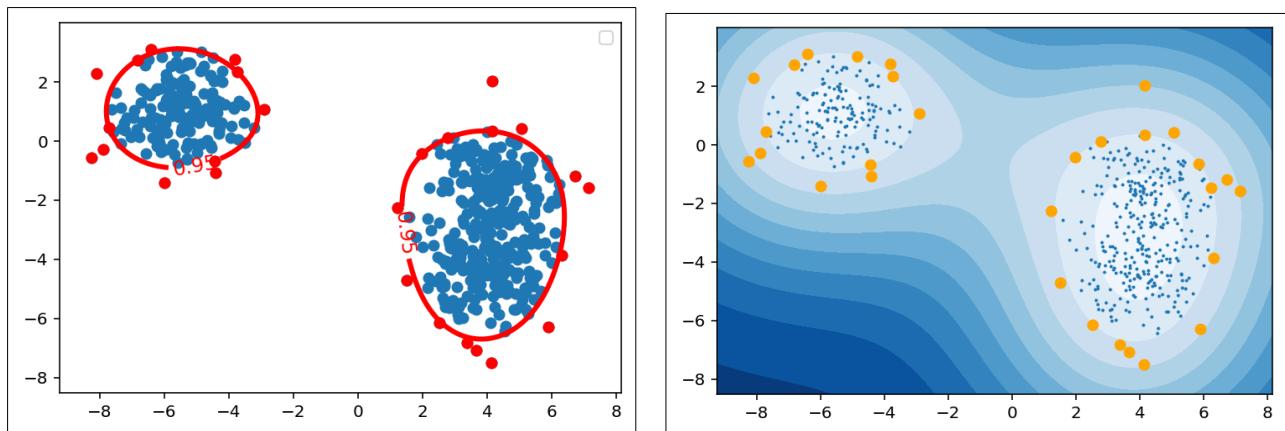


Figura 23. one-class SVM detectando outliers.

LA CLASE SGDOneClassSVM

La clase `sklearn.linear_model.SGDOneClassSVM` implementa una versión online de la clase *One-Class SVM* usando descenso por gradiente estocástico combinado con técnicas *kernel* que consiguen un comportamiento similar a `svm.OneClassSVM` pero más escalable porque tiene un orden lineal que depende del número de ejemplos mientras que la original es menos escalable (orden cuadrático) lo que la hace más adecuada cuando el número de ejemplos de entrenamiento es superior a 10.000.

Puedes ver este [enlace](#) en *scikit-learn* para comparar el uso de ambas clases y en este otro [enlace](#) puedes ver una comparativa de diferentes algoritmos usados para detectar *outliers*.

2.5. ALGORITMO DE ENTRENAMIENTO DE SVM.

Cuando hemos explicado la ecuación normal para calcular los parámetros de un modelo de regresión lineal dejábamos los parámetros del modelo en un vector θ , incluyendo el término bias como θ_0 y a las características de entrada que comenzaban desde θ_1 hasta θ_n se añadía un bias $x_0=1$ a todos los ejemplos. Ahora tendremos separados el término *bias* llamado b y el vector de los pesos de las características llamado w . No se añade ninguna columna *bias* a los vectores de entrada.

El modelo clasificador lineal SVM predice la clase de una nueva instancia x calculando simplemente la función de decisión $w^T x + b = w_1 x_1 + \dots + w_n x_n + b$: si el resultado es positivo entonces la clase que se predice \hat{y} es la clase positiva (1), o si es negativo se predice la clase (0).

$$\hat{y} = \begin{cases} 0 & \text{if } w^T x + b < 0, \\ 1 & \text{if } w^T x + b \geq 0 \end{cases}$$

La figura 24 muestra un plano 2D (dataset con las características “petal width” y “petal length”). La frontera de decisión es el conjunto de puntos donde la función de decisión es cero: es la intersección de dos planos, que es una línea recta (la línea sólida en la figura)⁸.

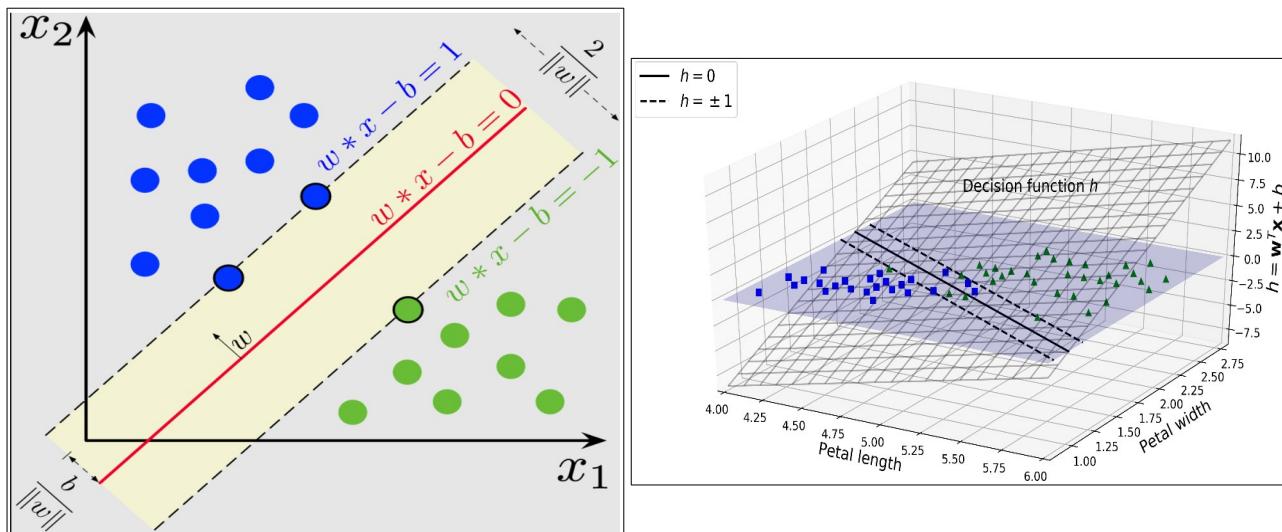


Figura 24. Elementos y Función de decisión para el dataset iris.

El w es el vector normal al hiperplano que separa a los datos (puede estar normalizado o no). Las líneas punteadas representan los puntos donde la función de decisión es igual a +1 o es igual a -1: son rectas paralelas e iguales en distancia a la frontera de decisión, formando un margen a su alrededor. Entrenar un clasificador lineal SVM significa encontrar los valores de los pesos w y del bias b que hacen este margen todo lo ancho que sea posible al mismo tiempo que evitan violaciones del margen (hard margin) o limitan la cantidad de ellas (soft margin).

LAS ECUACIONES OBJETIVO EN EL ENTRENAMIENTO DE CLASIFICADORES

Si una instancia está sobre la línea de decisión, evaluarlo con la función dará un resultado de 0. Si da un valor positivo estará a la derecha y si da negativo estará a la izquierda. Considera la pendiente de la función de decisión, es igual a la norma del vector de pesos $\|w\|$. Si dividimos esta pendiente por 2, los puntos donde esta función de decisión son iguales a -1 son dos veces se alejan el doble de la frontera de decisión. En otras palabras, dividir por 2 la pendiente multiplica el margen por 2. Esto seguramente es más fácil de visualizar en 2D como se representa en la figura 25.

⁸ Cuando hay más características, digamos n , la función de decisión es un hiperplano n -dimensional y la frontera de decisión es un hiperplano $(n - 1)$ -dimensional.

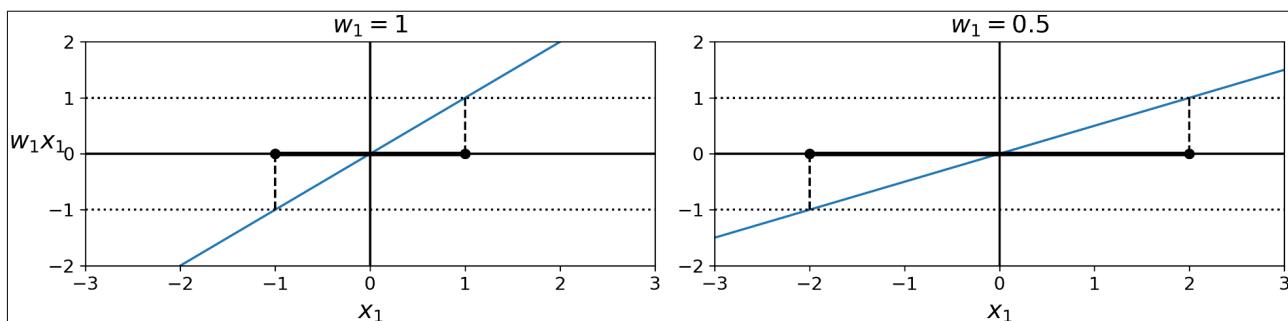


Figura 25. Un vector de pesos pequeño genera un margen grande.

Así que hay que minimizar $\|\mathbf{w}\|$ para obtener un margen amplio. Sin embargo, si también queremos evitar violaciones del margen (*hard margin*), entonces necesitamos que la función de decisión sea mayor de 1 para todas las instancias de entrenamiento positivas y menores de -1 para todas las instancias de entrenamiento negativas. Si definimos $t^{(i)}=-1$ para instancias negativas (si $y^{(i)}=0$) y $t^{(i)}=1$ para instancias positivas (si $y^{(i)}=1$), entonces podemos expresar esta restricción como $t^{(i)}(\mathbf{w}^T \mathbf{x}^{(i)} + b) \geq 1$ para todas las instancias.

Podemos expresar la función objetivo de un clasificador lineal SVM con *hard-margin* como un problema de optimización, maximizar la ecuación del ancho de sus carriles que sería $2/\|\mathbf{w}\|$. Maximizar esa fórmula equivale a un problema de minimizar su inversa (la pendiente):

$$\begin{aligned} & \underset{\mathbf{w}, b}{\text{minimize}} \quad \frac{1}{2} \mathbf{w}^T \mathbf{w} \\ & \text{subject to} \quad t^{(i)} (\mathbf{w}^T \mathbf{x}^{(i)} + b) \geq 1 \quad \text{for } i = 1, 2, \dots, m \end{aligned}$$

Nota: minimizamos $\frac{1}{2} \mathbf{w}^T \mathbf{w}$, que es igual a $\frac{1}{2} \|\mathbf{w}\|^2$, en vez de minimizar $\|\mathbf{w}\|$. El motivo es que $\frac{1}{2} \|\mathbf{w}\|^2$ tiene una bonita y sencilla derivada (solo \mathbf{w}) mientras que $\|\mathbf{w}\|$ no es diferenciable en $\mathbf{w}=\mathbf{0}$. Los algoritmos de optimización funcionan mejor con funciones que sean diferenciables.

Para obtener el objetivo del *soft margin*, necesitamos introducir una variable *slack variable* $\zeta^{(i)} \geq 0$ para cada instancia (la letra ζ): $\zeta^{(i)}$ mide cuantas veces tiene permitido la i -ésima instancia violar el margen. En este caso tenemos dos objetivos que entran en conflicto: hacer que las variables *slack* sean tan pequeñas como sea posible para reducir las violaciones del margen y por otro lado hacer que $\frac{1}{2} \mathbf{w}^T \mathbf{w}$ sea tan pequeño como sea posible para incrementar el margen. Aquí es donde entra el hiperparámetro C : permite definir el equilibrio de la ecuación de un clasificador lineal SVM con *soft margin*:

$$\begin{aligned} & \underset{\mathbf{w}, b, \zeta}{\text{minimize}} \quad \frac{1}{2} \mathbf{w}^T \mathbf{w} + C \sum_{i=1}^m \zeta^{(i)} \\ & \text{subject to} \quad t^{(i)} (\mathbf{w}^T \mathbf{x}^{(i)} + b) \geq 1 - \zeta^{(i)} \quad \text{and} \quad \zeta^{(i)} \geq 0 \quad \text{for } i = 1, 2, \dots, m \end{aligned}$$

Los problemas de minimizar las ecuaciones *hard margin* y *soft margin* son problemas de optimización cuadrática con restricciones lineales. Estos problemas se conocen en matemáticas como problemas *Quadratic Programming* (QP). Hay disponibles muchos métodos solucionadores utilizando técnicas que quedan fuera de nuestras posibilidades¹⁰. Una formulación general al problema sería:

$$\begin{aligned} & \underset{\mathbf{p}}{\text{Minimize}} \quad \frac{1}{2} \mathbf{p}^T \mathbf{H} \mathbf{p} + \mathbf{f}^T \mathbf{p} \\ & \text{subject to} \quad \mathbf{A} \mathbf{p} \leq \mathbf{b} \end{aligned}$$

⁹ ζ es la sexta letra del alfabeto griego que se llama zeta.

¹⁰ Si quieres aprender más sobre programación cuadrática puedes comenzar leyendo el libro de Stephen Boyd y Lieven Vandenberghe, *Convex Optimization* (Cambridge, UK: Cambridge University Press, 2004) o ver la serie de video lecturas de Richard Brown: [vídeos](#).

donde:

- \mathbf{p} es un vector de n_p dimensiones donde n_p es el número de parámetros.
- \mathbf{H} es una matriz de dimensiones $n_p \times n_p$
- \mathbf{f} es un vector de n_p dimensiones
- \mathbf{A} es una matriz de $n_c \times n_p$ dimensiones (n_c es el número de restricciones)
- \mathbf{b} es un vector de n_c dimensiones

Observa que la expresión $\mathbf{A} \mathbf{p} \leq \mathbf{b}$ define n_c restricciones: $\mathbf{p}^T \mathbf{a}^{(i)} \leq b^{(i)}$ para $i=1, 2, \dots, n_c$ donde $\mathbf{a}^{(i)}$ es el vector que contiene los elementos de la i-ésima fila de \mathbf{A} y $b^{(i)}$ es el i-ésimo elemento del vector \mathbf{b} .

Puedes comprobar que si estableces los parámetros del problema QP de la siguiente manera, obtienes la función objetivo de un clasificador lineal SVM hard-margin:

- $n_p = n + 1$, donde n es el número de características (el +1 es por el bias).
- $n_c = m$, donde m es el número de ejemplos usados para el entrenamiento.
- \mathbf{H} es la matriz identidad de dimensiones $n_p \times n_p$, excepto ceros en el elemento izquierdo de arriba (para ignorar el término del bias).
- $\mathbf{f} = \mathbf{0}$, un vector de n_p dimensiones todo a ceros.
- $\mathbf{b} = -\mathbf{1}$, un vector de n_c dimensiones todo lleno de valores -1.
- $\mathbf{a}^{(i)} = -\mathbf{t}^{(i)} \bar{\mathbf{x}}^{(i)}$, donde $\bar{\mathbf{x}}^{(i)}$ es igual a $\mathbf{x}^{(i)}$ con una característica extra de bias $\bar{\mathbf{x}}_0 = 1$.

Así que entrenar un clasificador lineal SVM de *hard-margin* consiste en usar un solver de QP pasándole estos parámetros. El resultado es un vector \mathbf{p} que contendrá el término bias en $b = p_0$ y los pesos de las características $w_i = p_i$ para $i=1, 2, \dots, n$. De manera parecida podemos usar un solver QP para solucionar el problema de un *soft-margin*.

Sin embargo, para usar un *kernel* debemos usar una aproximación diferente utilizando el concepto de el problema dual que básicamente consiste en que si tenemos un problema principal $\mathbf{f}(\mathbf{w})$ a minimizar y unas restricciones $\mathbf{g}(\mathbf{w}, \mathbf{b})$ podemos transformar ese problema con restricciones en otro sin restricciones (integrando las restricciones en la misma ecuación) llamado el problema secundario. Solucionar el problema secundario da un límite inferior al problema primario y bajo ciertas condiciones las mismas soluciones que el primario. Tenemos mucha suerte porque como la función objetivo es convexa y la desigualdad de las restricciones son diferenciables se cumple que solucionar el problema secundario equivale a solucionar el primario. En nuestro caso tenemos que:

$$\mathbf{f}(\mathbf{x}) = \frac{1}{2} \mathbf{w}^T \mathbf{w} \quad \mathbf{g}(\mathbf{w}, \mathbf{b}) = y_i(\mathbf{w}^T \mathbf{x}_i + b) - 1 = y_i(\mathbf{w}^T \mathbf{x}_i + b) - 1 = 0$$

Aplicando el operador de *Lagrange* transformamos el problema de minimizar el problema primario en minimizar el problema secundario. El operador de *Lagrange* es una función que añade un nuevo vector de parámetros α pero que es sencillo de solucionar:

$$L(\mathbf{w}, b, \alpha_i) = \frac{1}{2} \mathbf{w}^T \mathbf{w} - \sum_{i=1}^L \alpha_i [y_i(\mathbf{w}^T \mathbf{x}_i + b) - 1]$$

donde L es el número de vectores soporte. Derivando e igualando a cero:

$$\begin{cases} \frac{\partial L}{\partial \mathbf{w}} = 0 \\ \frac{\partial L}{\partial b} = 0 \end{cases} = \dots$$

$$\text{minimize}_{\alpha} \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha^{(i)} \alpha^{(j)} t^{(i)} t^{(j)} \mathbf{x}^{(i)} T \mathbf{x}^{(j)} - \sum_{i=1}^m \alpha^{(i)} \quad (\text{ecuación dual})$$

subject to $\alpha^{(i)} \geq 0$ for $i = 1, 2, \dots, m$

Una vez que encuentras el valor del vector α que minimiza la ecuación (usando un *QP solver*), puedes calcular w y b que minimizan el problema primario usando las ecuaciones:

$$\hat{w} = \sum_{i=1}^m \hat{\alpha}^{(i)} t^{(i)} \mathbf{x}^{(i)}$$

$$\hat{b} = \frac{1}{n_s} \sum_{i=1}^m \left(t^{(i)} - \hat{w}^T \mathbf{x}^{(i)} \right)_{\hat{\alpha}^{(i)} > 0}$$

Solucionar el problema dual es más rápido que solucionar el primario cuando el número de ejemplos es más pequeño que el número de características. Pero lo más importante, es que resolver el problema dual hace posible utilizar *kernel*, cosa que no es posible si resolvemos directamente el primario. Bueno, ¿Y si hay *kernel*?

Supongamos que queremos aplicar una transformación polinomial de grado 2 a unos datos de entrenamiento 2D (como en el dataset *moon* por ejemplo), porque queremos entrenar un clasificador lineal SVM sobre los datos transformados. Esta ecuación es la transformación que queremos aplicar a los datos.

$$\phi(\mathbf{x}) = \phi\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{pmatrix}$$

El dato original es un vector 2D y una vez transformado es un vector 3D. Vamos a ver lo que les ocurre a dos vectores 2D llamados a y b , si les aplicamos la transformación y entonces calculamos el producto¹¹ en el nuevo espacio.

$$\begin{aligned} \phi(a)^T \phi(b) &= \begin{pmatrix} a_1^2 \\ \sqrt{2}a_1a_2 \\ a_2^2 \end{pmatrix}^T \begin{pmatrix} b_1^2 \\ \sqrt{2}b_1b_2 \\ b_2^2 \end{pmatrix} = a_1^2 b_1^2 + 2a_1 b_1 a_2 b_2 + a_2^2 b_2^2 \\ &= (a_1 b_1 + a_2 b_2)^2 = \left(\begin{pmatrix} a_1 \\ a_2 \end{pmatrix}^T \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \right)^2 = (\mathbf{a}^T \mathbf{b})^2 \end{aligned}$$

El *dot product* de los vectores transformados es el cuadrado del *dot product* de los vectores originales: $\phi(a)^T \phi(b) = (\mathbf{a}^T \mathbf{b})^2$.

Aquí está la clave: si aplicas la transformación ϕ a todos los ejemplos de entrenamiento, el problema dual debe contener el *dot product* $\phi(\mathbf{x}^{(i)})^T \phi(\mathbf{x}^{(j)})$. Pero si ϕ es la transformación polinomial definida antes, puedes cambiar el *dot product* de los vectores transformados simplemente por $(\mathbf{x}^{(i)T} \mathbf{x}^{(j)})^2$. De esta forma no necesitas transformar los datos de entrenamiento en absoluto: solo debes remplazar el *dot product* por su cuadrado en la ecuación dual.

11 El producto externo (*dot product*) de dos vectores a y b normalmente se expresa como $a \cdot b$ y genera un valor escalar (un número). Pero como en Machine Learning los vectores se representan a menudo como vectores columna (matrices de una sola columna), su producto puede calcularse también realizando la operación $\mathbf{a}^T \mathbf{b}$. Aunque hablando de forma rigurosa este resultado devuelve una matriz 1×1 la consideramos un escalar.

La función $K(\mathbf{a}, \mathbf{b}) = (\mathbf{a}^T \mathbf{b})^2$ se llama *kernel* polinomial de grado 2. En Machine Learning, un *kernel* es una función que es capaz de calcularse basándose en el *dot product* $\phi(\mathbf{a})^T \phi(\mathbf{b})$ de los vectores originales \mathbf{a} y \mathbf{b} , sin tener que calcular realmente (o incluso sin tener que saber como hacerlo) la transformación ϕ . Aquí aparecen algunos *kernel*:

$$\text{Linear: } K(\mathbf{a}, \mathbf{b}) = \mathbf{a}^T \mathbf{b}$$

$$\text{Polynomial: } K(\mathbf{a}, \mathbf{b}) = (\gamma \mathbf{a}^T \mathbf{b} + r)^d$$

$$\text{Gaussian RBF: } K(\mathbf{a}, \mathbf{b}) = \exp(-\gamma \| \mathbf{a} - \mathbf{b} \|^2)$$

$$\text{Sigmoid: } K(\mathbf{a}, \mathbf{b}) = \tanh(\gamma \mathbf{a}^T \mathbf{b} + r)$$

TEOREMA DE MERCER

Si una función $K(\mathbf{a}, \mathbf{b})$ respeta unas cuantas condiciones matemáticas llamadas las *condiciones de Mercer* (K debe ser continua, simétrica en sus argumentos de manera que $K(\mathbf{a}, \mathbf{b}) = K(\mathbf{b}, \mathbf{a})$, etc.), entonces existe una función ϕ que mapea \mathbf{a} y \mathbf{b} a otro espacio (posiblemente con más alta dimensionalidad) de forma que $K(\mathbf{a}, \mathbf{b}) = \phi(\mathbf{a})^T \phi(\mathbf{b})$.

Así que K puede utilizarse como un *kernel* si sabes que existe ϕ , o incluso cuando no conoces cuál es su ϕ . Este es el caso del *kernel Gausiano RBF*, donde su ϕ mapea cada ejemplo de entrenamiento a un espacio dimensional infinito, así que ¡no tener realizar el mapeo es un alivio!

Algunos *kernel* que se usan con frecuencia (como el *kernel sigmoide*) no respetan todas las condiciones de *Mercer* aunque en la práctica suelen funcionar bien.

Otro asunto pendiente: hemos visto como ir de la solución dual en el caso del clasificador lineal SVM, pero si aplicas un *kernel* acabas con una ecuación donde aparece $\phi(\mathbf{x}^{(i)})$. De hecho, \mathbf{w} debe tener el mismo número de dimensiones que $\phi(\mathbf{x}^{(i)})$, lo que puede dar lugar a muchos o hasta infinitos cálculos. ¿Cómo se pueden hacer predicciones sin conocer \mathbf{w} ? Como puedes mover \mathbf{w} desde la ecuación donde aparece hasta la función de decisión de una nueva instancia $\mathbf{x}(n)$, que se queda con operaciones dot product entre los vectores de entrada.

$$\begin{aligned} h_{\widehat{\mathbf{w}}, \hat{b}}(\phi(\mathbf{x}^{(n)})) &= \widehat{\mathbf{w}}^T \phi(\mathbf{x}^{(n)}) + \hat{b} = \left(\sum_{i=1}^m \hat{\alpha}^{(i)} t^{(i)} \phi(\mathbf{x}^{(i)}) \right)^T \phi(\mathbf{x}^{(n)}) + \hat{b} \\ &= \sum_{i=1}^m \hat{\alpha}^{(i)} t^{(i)} \left(\phi(\mathbf{x}^{(i)})^T \phi(\mathbf{x}^{(n)}) \right) + \hat{b} \\ &= \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m \hat{\alpha}^{(i)} t^{(i)} K(\mathbf{x}^{(i)}, \mathbf{x}^{(n)}) + \hat{b} \end{aligned}$$

Como $\mathbf{a}(i) \neq \mathbf{0}$ solo para los vectores soporte, predecir implica hacer el *dot product* del nuevo vector de entrada $\mathbf{x}(n)$ solo con los vectores soporte, no con todos los ejemplos. El bias b es:

$$\begin{aligned} \hat{b} &= \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m \left(t^{(i)} - \widehat{\mathbf{w}}^T \phi(\mathbf{x}^{(i)}) \right) = \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m \left(t^{(i)} - \left(\sum_{j=1}^m \hat{\alpha}^{(j)} t^{(j)} \phi(\mathbf{x}^{(j)}) \right)^T \phi(\mathbf{x}^{(i)}) \right) \\ &= \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m \left(t^{(i)} - \sum_{\substack{j=1 \\ \hat{\alpha}^{(j)} > 0}}^m \hat{\alpha}^{(j)} t^{(j)} K(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) \right) \end{aligned}$$

2.6. SVM Online.

Recordemos que online significa que el aprendizaje se realiza incrementalmente, normalmente cada vez que llega una nueva instancia o ejemplo. Para los clasificadores *SVM*, un método utilizado es usar el algoritmo descenso por gradiente (usando **SGDClassifier**) para minimizar una función de coste como la de abajo que se obtiene del problema primario. El inconveniente que tiene es que la convergencia es mucho más lenta que el método basado en *QP*.

$$J(\mathbf{w}, b) = \frac{1}{2} \mathbf{w}^T \mathbf{w} + C \sum_{i=1}^m \max(0, 1 - t^{(i)} (\mathbf{w}^T \mathbf{x}^{(i)} + b))$$

La primera suma es la función de coste que intenta que el modelo tenga un vector \mathbf{w} de pesos pequeño para así tener un margen grande. La segunda suma calcula el total de violaciones del margen. Una instancia que genere una violación del margen a 0 estará fuera de los carriles y en el lado correcto, o de lo contrario a una distancia proporcional a la distancia del lado correcto. Minimizando estos términos asegura que el modelo tenga una cantidad de violaciones del margen tan pequeña como sea posible.

HINGE LOSS

La función $\max(0, 1 - t)$ se llama la función **hinge loss** y está representada en la figura 26. Es igual a 0 cuando $t \geq 1$. Su derivada (pendiente) es -1 si $t < 1$ y 0 si $t > 1$. No es diferenciable en $t = 1$, pero como en la regresión Lasso todavía se puede usar el descenso por gradiente usando alguna subderivada en $t = 1$ (es decir, cualquier valor entre -1 y 0).

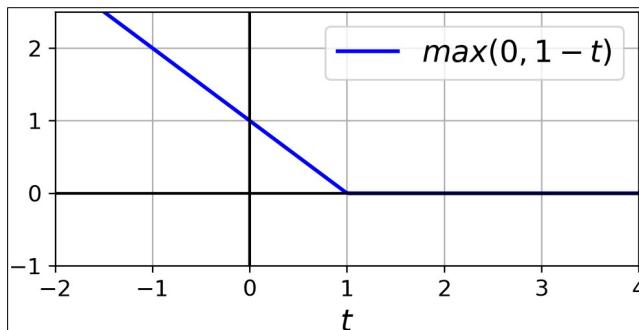


Figura 26. Función hinge Loss.

También hay otras implementaciones por ejemplo usando “[Incremental and Decremental SVM Learning](#)” o “[Fast Kernel Classifiers with Online and Active Learning](#).” que están implementados en Matlab y C++. Sin embargo, para problemas no lineales a gran escala es mejor usar redes neuronales.

2.7. PREPROCESAMIENTO Y TUNING DE PARÁMETROS DE SVM.

El parámetro gamma que usa el *kernel gausiano RBF* controla su anchura lo que determina la escala de lo que significa que dos ejemplos cercanos estén juntos. El parámetro C es un parámetro de regularización similar al que se usa en los modelos lineales y limita la importancia de cada ejemplo (en concreto su atributo *dual_coef_*). Veamos lo que ocurre cuando cambiamos estos parámetros.

EJEMPLO 24: Generar varios clasificadores SVM con diferentes hiperparámetros.

En primer lugar copia este código en el fichero *U03_tools.py*

```
from sklearn.svm import SVC

def plot_svm(log_C, log_gamma, ax=None):
    X, y = crea_dataset_revuelto()
    C = 10. ** log_C
    gamma = 10. ** log_gamma
```

```

svm = SVC(kernel='rbf', C=C, gamma=gamma).fit(X, y)
if ax is None:
    ax = plt.gca()
plot_2d_separator(svm, X, ax=ax, eps=.5)
discrete_scatter(X[:, 0], X[:, 1], y, ax=ax)      # dibujar datos
sv = svm.support_vectors_                         # dibuja vectores soporte
sv_labels = svm.dual_coef_.ravel() > 0           # Etiquetas de clases de vectores soporte
discrete_scatter(sv[:, 0], sv[:, 1], sv_labels, s=15, markeredgewidth=3, ax=ax)
ax.set_title("C = %.4f gamma = %.4f" % (C, gamma))

```

Y ahora crea el programa que genere varios gráficos de una SVM en U03_Ejemplo24.py

```

# -*- coding: utf-8 -*-
import matplotlib.pyplot as plt
from u03_tools import plot_svm

fig, ejes = plt.subplots(3, 3, figsize=(15, 10))
for ax, C in zip(ejes, [-1, 0, 3]):
    for a, gamma in zip(ax, range(-1, 2)):
        plot_svm(log_C=C, log_gamma=gamma, ax=a)
ejes[0, 0].legend(["clase 0", "clase 1", "sv clase 0", "sv clase 1"],
                  ncol=4, loc=(.9, 1.2))

```

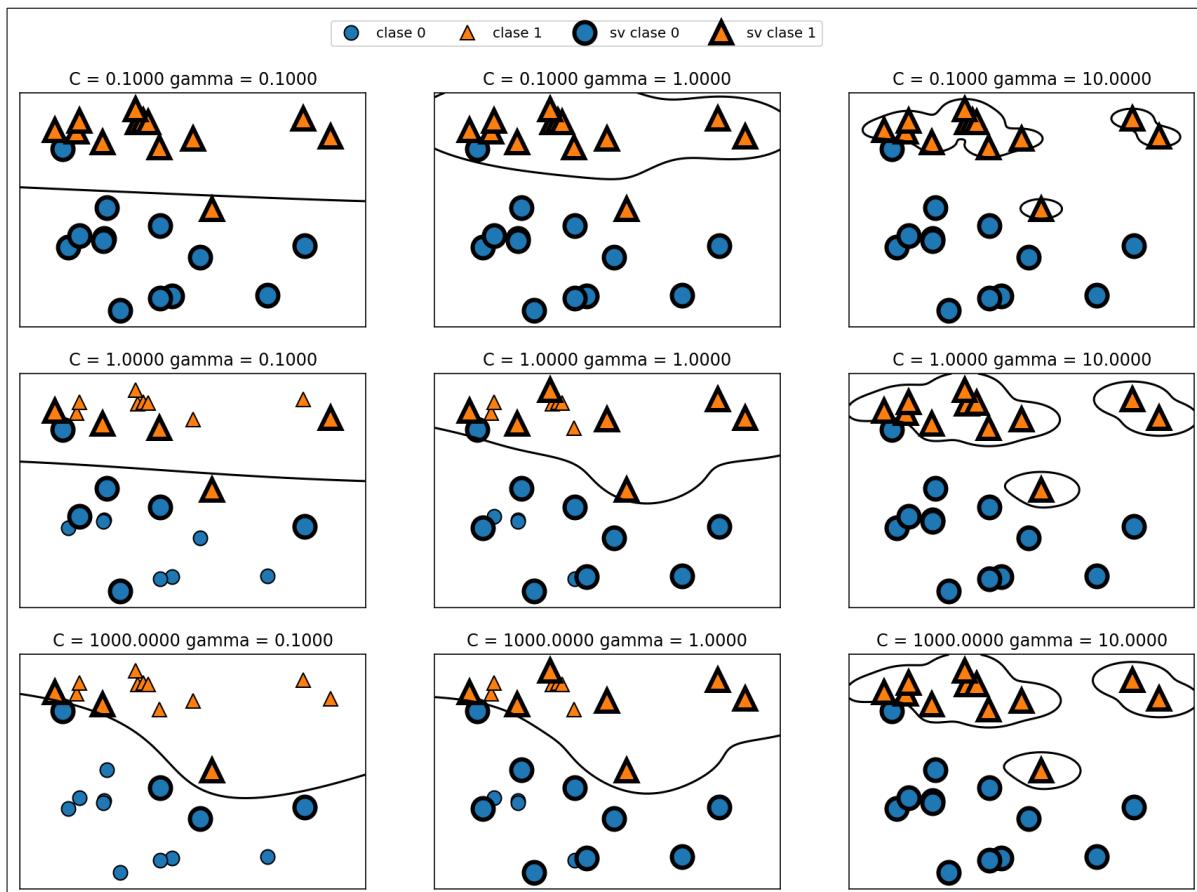


Figura 27. Límites de decisión y vectores soporte para valores diferentes de C y γ .

Vamos a ir de izquierda a derecha, incrementando el valor del parámetro γ de 0.1 a 10 multiplicando por 10 en cada cambio. Tener un γ pequeño significa un alto radio del kernel gausiano y eso se traduce en que muchos puntos se consideran cercanos. Esto provoca que la frontera de decisión es muy suave hacia la izquierda mientras que cuantos más a la derecha más se ajusta a cada punto individual. Así que un valor bajo de γ tiene una frontera que cambia muy suavemente y genera un modelo de complejidad baja mientras que si es alto aumenta la complejidad del modelo.

De arriba hacia abajo se incrementa el valor del parámetro C desde 0.1 hasta 1000. Como en los modelos lineales, un valor pequeño de C crea un modelo muy restrictivo donde cada dato tiene

una influencia muy limitada. Por ese motivo la frontera del modelo de arriba a la izquierda parece un clasificador lineal, con puntos mal clasificados. Al incrementar C , como se ve en los gráficos de abajo, estos puntos tienen un fuerte influencia en el modelo y hacen que la frontera de decisión se ajuste correctamente para clasificarlos. Si aplicamos el *kernel RBF* en un SVM al dataset *Breast Cancer* que viene como un dataset de ejemplo en *scikit-learn* con valores por defecto de $C=1$ y $\gamma=1/n_caracteristicas$:

EJEMPLO 25: Cargar datos del dataset, entrenar un modelo SVC y medir su eficiencia.

```
# -*- coding: utf-8 -*-
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
import sklearn.datasets as ds

cancer = ds.load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target,
                                                    random_state=4)
svc = SVC()
svc.fit(X_train, y_train)
print("Accuracy en train: {:.2f}".format(svc.score(X_train, y_train)))
print("Accuracy en test: {:.2f}".format(svc.score(X_test, y_test)))
```

El modelo tiene algo de sobreajuste aunque no exagerado, con un *accuracy* en datos *train* de 0.93 y en *test* 0.88. Normalmente SVM funciona muy bien, pero es muy sensible a los hiperparámetros escogidos y la escala de los datos. Necesita que todas las características cambien de manera similar en la misma escala. Vamos a comprobar el valor máximo y mínimo de cada característica del dataset en escala logarítmica. También hacemos un gráfico de *boxplot* para representar los datos.

EJEMPLO 26: Añadimos al código del ejemplo 25 el siguiente para ver como son los máximos y mínimos en escala logarítmica (máximos con triángulos y mínimos con círculos).

```
import matplotlib.pyplot as plt
plt.plot(X_train.min(axis=0), 'o', label="min")
plt.plot(X_train.max(axis=0), '^', label="max")
plt.legend(loc=4)
plt.xlabel("Índice de características")
plt.ylabel("Magnitud de Características (log)")
plt.yscale("log")
plt.show()
plt.boxplot(X_train, manage_ticks=False)
plt.yscale("symlog")
plt.xlabel("Índice de características")
plt.ylabel("Magnitud de Características (log)")
```

Mirando la figura 28 que genera el código vemos que los datos del dataset tienen órdenes de magnitud completamente diferentes, y esto puede ser problemático para otros modelos (haciendo que el entrenamiento sea más lento o que no encuentren soluciones óptimas) pero esto para SVM es devastador sobre todo si usas *kernel*.

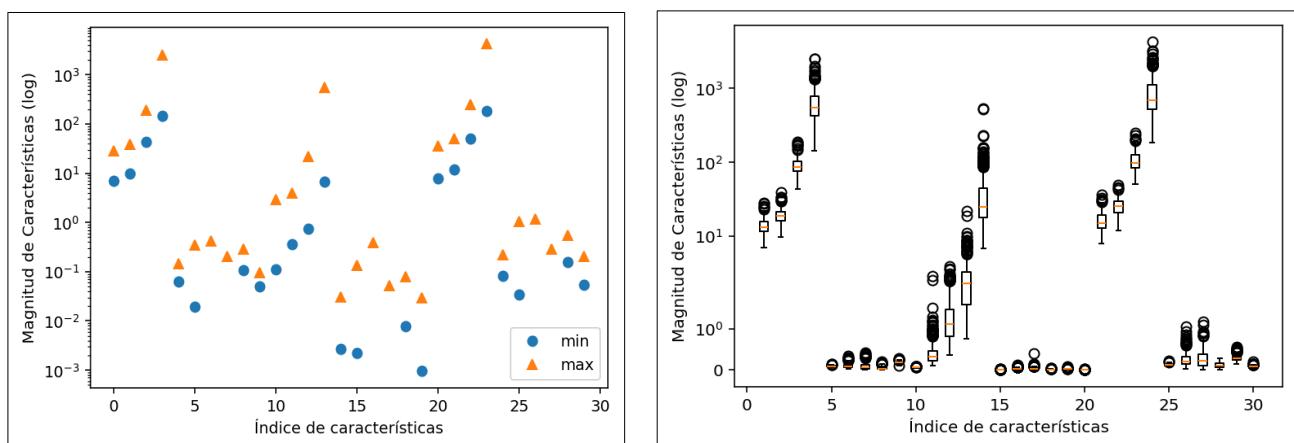


Figura 28. Rangos de valores de las características del dataset (eje y en escala logarítmica) y boxplot.

Vamos a mejorar los resultados simplemente actuando sobre los datos, por ejemplo escalando cada característica a una escala similar. Un método muy usado para SVM es dejar todos los valores de cada característica entre 0 y 1. En vez de hacerlo de manera manual usaremos la clase **MinMaxScale**:

EJEMPLO 27: Escalar todos los datos de train. Añade el código al ejemplo 25.

```
# Calcular el mínimo valor de cada columna
min_train = X_train.min(axis=0)
rango_train = (X_train - min_train).max(axis=0)
X_train_escalado = (X_train - min_train) / rango_train
print("Mínimo de cada característica\n", X_train_escalado.min(axis=0))
print("Máximo de cada característica\n", X_train_escalado.max(axis=0))
X_test_escalado = (X_test - min_train) / rango_train # Escala test
# Volver e antrenar modelo
svc = SVC()
svc.fit(X_train_escalado, y_train)
print(f"Accuracy de train: {svc.score(X_train_escalado, y_train):.3f}")
print(f"Accuracy de test: {svc.score(X_test_escalado, y_test):.3f}")
```

Mejoramos un 5% en train (0.981) y un 10% en test (0.972). Son buenos números, quedan valores muy similares en ambos conjuntos de datos. Ahora podríamos incluso intentar incrementar **C** o **gamma** para aumentar la complejidad del modelo a ver qué ocurre.

EJEMPLO 28: Modificar hiperparámetros **C** o **gamma**. Añade el código al ejemplo 22 y lo pruebas.

```
svc = SVC(C=1000)
svc.fit(X_train_escalado, y_train)
print(f"Accuracy de train: {svc.score(X_train_escalado, y_train):.3f}")
print(f"Accuracy de test: {svc.score(X_test_escalado, y_test):.3f}")
```

Los SVM con *kernel* son modelos potentes que funcionan bien en una gran variedad de datasets. Permiten definir fronteras de decisión complejas hasta cuando los datos tienen poca dimensionalidad (características). Funcionan bien con datos de baja y alta dimensionalidad. Pero no escalan bien cuando aumenta la cantidad de filas (el tamaño del dataset). Entrenar SVM con más de cien mil filas exige mucha potencia de cálculo y memoria.

Otro inconveniente es que necesitan datos preprocesados y tuning de parámetros. Por eso en la actualidad la mayoría prefiere utilizar modelos basados en árboles de decisión como *random forest* o *gradient boosting* (necesitan menos preprocesamiento). Además es difícil explicar el motivo de que realicen cierta decisión.

Los hiperparámetros importantes en los *Kernel* de las SVM son el parámetro de regularización **C**, elegir el *kernel* que se quiera y los parámetros específicos de cada *kernel*. El *RBF* tiene **gamma** como único parámetro (la inversa de la anchura del *kernel gausiano*). Cuando aumentas **C** o **gamma** aumentas la complejidad del modelo. Como ambas están muy correlacionadas deberías ajustarlas juntas.

3. ÁRBOLES DE DECISIÓN (DECISION TREES).

El objetivo de estos algoritmos de aprendizaje automático supervisado es crear un modelo. A partir de los datos de entrenamiento construyen una estructura de árbol que contiene lo que ha aprendido de los datos. Un árbol de decisión es similar a un diagrama de flujo, donde cada nodo interno (no hoja) representa una pregunta sobre una característica, cada rama representa el resultado de la prueba y cada hoja (o nodo terminal) tiene una etiqueta de clase. El nodo superior en un árbol es el nodo raíz.

Un árbol se genera fraccionamiento o particionando el conjunto inicial de datos en varios subconjuntos. La partición se basa en que los datos cumplen o no una pregunta sobre el valor de alguna de las características. Por ejemplo una prueba podría ser ¿La edad del cliente es mayor de

26? los datos que lo cumplan van a un grupo y los que no, van a otro grupo. Esa pregunta produce una partición de los datos en dos subconjuntos usando la característica edad y el valor 26.

Este proceso de división se repite en cada subconjunto de una manera recursiva y se llama **particionamiento recursivo**. Este proceso termina cuando el subconjunto de datos de un nodo tiene todos los datos con el mismo valor de la variable objetivo o cuando la partición ya no ayuda a clasificar los datos, o si se incumplen ciertas condiciones de parada. Este proceso de **inducción top-down** de los árboles de decisión es un ejemplo de algoritmos voraces (greedy).

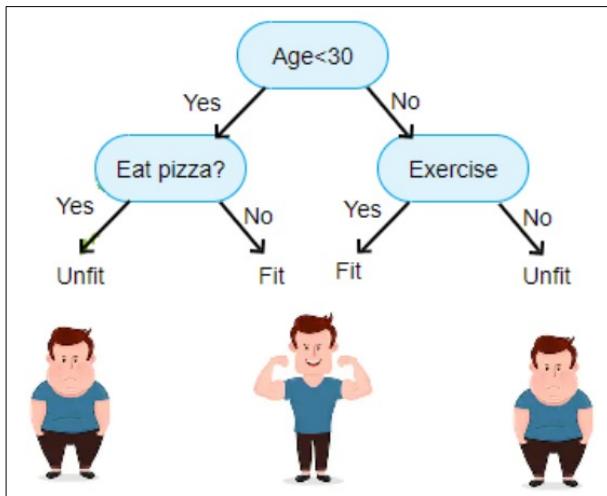


Figura 29. Árbol de decisión.

Los árboles de decisión se pueden emplear en dos tareas principales:

- **Clasificación** cuando el resultado predicho es la clase a la que pertenecen los datos.
- **Regresión** cuando el resultado predicho se puede considerar un número real (por ejemplo, el precio de una casa o el número de días de estancia de un paciente en un hospital).

Hay muchos algoritmos para generar árboles de decisión, los más destacados:

- [ID3](#)(Iterative Dichotomiser 3)
- [C4.5](#)(Sucesor de ID3)
- [CART](#)(Classification And Regression Trees) El término **Árboles de Clasificación y Regresión** (CART) es un término introducido por primera vez por [Breiman](#) et al. Los árboles utilizados para regresión y para la clasificación tienen algunas similitudes y también algunas diferencias, como el procedimiento utilizado para decidir donde dividir.
- [CHAID](#) (Detector automático de Chi-cuadrado de interacción). Realiza divisiones de múltiples niveles al calcular los árboles de clasificación.
- [MARS](#): Extiende los árboles de decisión para manejar mejor datos numéricos.
- [Árboles de Inferencia Condicional](#). Enfoque que utiliza pruebas no paramétricas como criterios de división, correcciones para múltiples pruebas para evitar el sobreajuste. Este enfoque se traduce en la selección de un predictor imparcial y no requiere poda.

Cada nodo intermedio del árbol actúa como un caso de prueba para algún atributo y cada rama que desciende de ese nodo corresponde a una de las posibles respuestas al caso de prueba. Este proceso es recursivo y se repite para cada subárbol. También es sencillo traducirlos a una serie de reglas anidadas de tipo SI-SINO o realizar un programa que nos vaya haciendo preguntas y acabe indicándonos la categoría como refleja este pseudocódigo que comenzando en el nodo raíz va mostrando la pregunta de cada nodo intermedio y según la respuesta que se le da (SI/NO) irá moviéndose al nodo hijo que corresponde a la respuesta hasta alcanzar un nodo hoja:

```

Nodo_actual <- raíz;
MIENTRAS nodo_actual no sea hoja HACER
    ESCRIBIR nodo_actual.pregunta;
    LEER respuesta;
    SI respuesta = "SI" ENTONCES
        nodo_actual <- nodo_actual.hijo_izquierda;
    SINO
        nodo_actual <- nodo_actual.hijo_derecha
    FINSI
FINMIENTRAS
ESCRIBIR "Categoría asignada: ", nodo_actual.categoría;

```

Como ves, la traducción desde el árbol a código es casi directa. En realidad son una forma de expresar un algoritmo formado por muchas sentencias **if-else** anidadas. Antes de profundizar, revisamos algunos términos:

- **Nodo raíz (nodo de decisión superior):** Representa a todos los datos.
- **División:** Es un proceso de división de los datos de un nodo en dos o más subnodos.
- **Nodo de decisión:** Cuando un subnodo se divide en subnodos adicionales.
- **Nodo de hoja / terminal:** nodos sin hijos (sin división adicional).
- **Poda:** reducir el tamaño de los árboles eliminando nodos (opuesto a la división).
- **Rama / Subárbol:** Una parte del árbol de decisión se denomina rama o subárbol.
- **Nodo padre e hijo:** Un nodo, que se divide en subnodos se denomina nodo principal o padre de sus subnodos, mientras que los subnodos son su hijos.

Ventajas de los Árboles de Decisión

- **Fáciles de entender.** Las personas comprenden sus decisiones y funcionamiento.
- **Requieren poca preparación de los datos.** El preprocessamiento es mínimo.
- **Trabajan con datos numéricos y categóricos.** Otros algoritmos y modelos no (redes neuronales solo valores numéricos → obliga a transformar datos, etc.).
- **Utilizan un modelo de caja blanca.** Es fácil explicar e interpretar los resultados que genera.
- **Se validan con pruebas estadísticas.** Se mide su funcionamiento.
- **Robustos.** Los valores anómalos apenas les afectan.
- **Escalables.** Pueden analizar grandes cantidades de datos y no necesitan excesivos recursos.

Limitaciones

- **El aprendizaje de un árbol óptimo es un problema NP-completo incluso para conceptos simples.** Los algoritmos de aprendizaje se basan en heurísticas (algoritmos voraces) y las decisiones localmente óptimas se hacen en cada nodo. Los algoritmos no pueden garantizar generar el árbol globalmente óptimo.
- **Los algoritmos de aprendizaje pueden crear árboles excesivamente complejos con overfitting:** aparece a veces la necesidad de aplicar mecanismos correctores como la poda (con excepción de algunos algoritmos como la Inferencia Condicional que no los requiere).
- **Hay conceptos difíciles de representar:** no expresan fácilmente relaciones como XOR, paridad o problemas de multiplexión. Para conseguirlos se vuelven prohibitivamente grandes. Resolver estos problemas suponen el cambio de la representación del dominio del problema o el uso de algoritmos de aprendizaje con representaciones más expresivas (aprendizaje relacional estadística o programación lógica inductiva).
- **Datos no balanceados:** si hay características con más datos de un tipo que de otro, el árbol se decanta por la opción más numerosa (**alto bias**). Pero esto tiene fácil solución al crear los datos de entrenamiento.

- **Sensibles a pequeños cambios:** un pequeño cambio en los datos genera árboles totalmente diferentes. El *bagging* y el *boosting* lo corrigen.
- **Sensibles a datos ruidosos:** sobreajusta el ruido.

3.1. ALGORITMOS DE ENTRENAMIENTO.

¿Cómo consigue un algoritmo de aprendizaje construir el mejor árbol que represente un algoritmo de este tipo, con la mínima cantidad de preguntas y con la máxima cantidad de aciertos al predecir? Se parte de todos los datos de entrenamiento en el nodo raíz. Hay que escoger una de las características y hacer una pregunta usando alguno de sus valores para comprobar que datos responden si y cuáles responden no, ejemplos de preguntas pueden ser:

- Si la característica es categórica:
 - La característica es igual a este valor?
- Si la característica es numérica:
 - La característica es mayor que este valor?
 - La característica está entre estos dos valores?

Se procesan las filas o ejemplos de datos y se separan en dos grupos: el grupo de filas que responden si a la pregunta y el grupo de filas que responden no a la pregunta. Es decir, la pregunta genera **una división de las filas en dos grupos**. A cada nuevo grupo se le vuelve a aplicar el mismo proceso de división y el proceso se repite una y otra vez de manera recursiva. Pero para aplicar este proceso surgen varias preguntas importantes:

- ¿Qué característica escojo para dividir al grupo de datos originales?
- ¿Qué pregunta con esa característica hago? Es decir, de entre los posibles valores ¿cuál uso?
- ¿Hasta cuando continúa el proceso de división?
- ¿Cómo saber si una división es útil o no? ¿Cómo medir lo útil que es?

Vamos a ir dando respuestas a cada una de las preguntas anteriores:

- **¿Qué característica se escoge para dividir? ¿Qué valor se usa para hacer la división?** No hay ningún método a priori que permita saber en cada momento la mejor característica y el mejor valor a usar para realizar una división. Por tanto hay que averiguarlo por prueba y error. Durante la división recursiva de los datos hay que ir probando a dividir por todas las características. Para cada característica se prueban diferentes puntos de división (se prueban distintos valores para formular preguntas relacionadas con cada columna). Al hacer cada prueba, se dividen los datos en dos grupos: filas que responden SI a la pregunta y otro grupo con las filas que responden NO. Ahora se puntuán los nuevos datos generados con una función de coste. Se va recordando la división con mejor coste (la división que ha dado el valor más bajo de puntuación). **Esto significa recordar en cada prueba, la columna y el valor usado que ha generado la mejor división.**
- **¿Qué preguntas puedo hacer en cada columna?** Si los datos de la columna son numéricos la condición puede ser: "característica == valor1", "característica < valor1", "característica > valor1", "característica está entre valor1 y valor2", etc. Y si los datos son simbólicos (String, booleanos, códigos numéricos, etc.) suele utilizarse la igualdad o la desigualdad: "característica == "valor", "característica != "valor"..."
- **¿Hasta cuando continúa el proceso de división?** Cuando un problema tiene muchas características, la cantidad de divisiones que pueden aplicarse será enorme y el árbol que se genera puede ser muy complejo y propenso a tener **overfitting**. Así que la decisión de cuándo parar de hacer divisiones es importante. Una forma de hacerlo es **fijar un número mínimo de filas** en cada grupo hoja del árbol. Por ejemplo, si al hacer una división nos sale

un grupo de menos de 10 filas, se descarta esa división. Otra estrategia es **fijar una altura máxima del árbol**, es decir cuando un grupo supera cierta separación con respecto a la raíz, se descarta la división.

- **¿Cómo se calcula el coste de una división?** Una función de coste mide lo buena o la mala que es una división. Hay varias métricas, y cada algoritmo utiliza una o varias de ellas.
 - **Si usamos el árbol para clasificación:** en general las métricas intentan medir como de homogéneos son los grupos de datos que se van creando (se intenta maximizar la cantidad de filas que tienen valores similares en la columna target). Esto tiene sentido si podemos asegurar que los datos de entrada siguen cierto patrón.

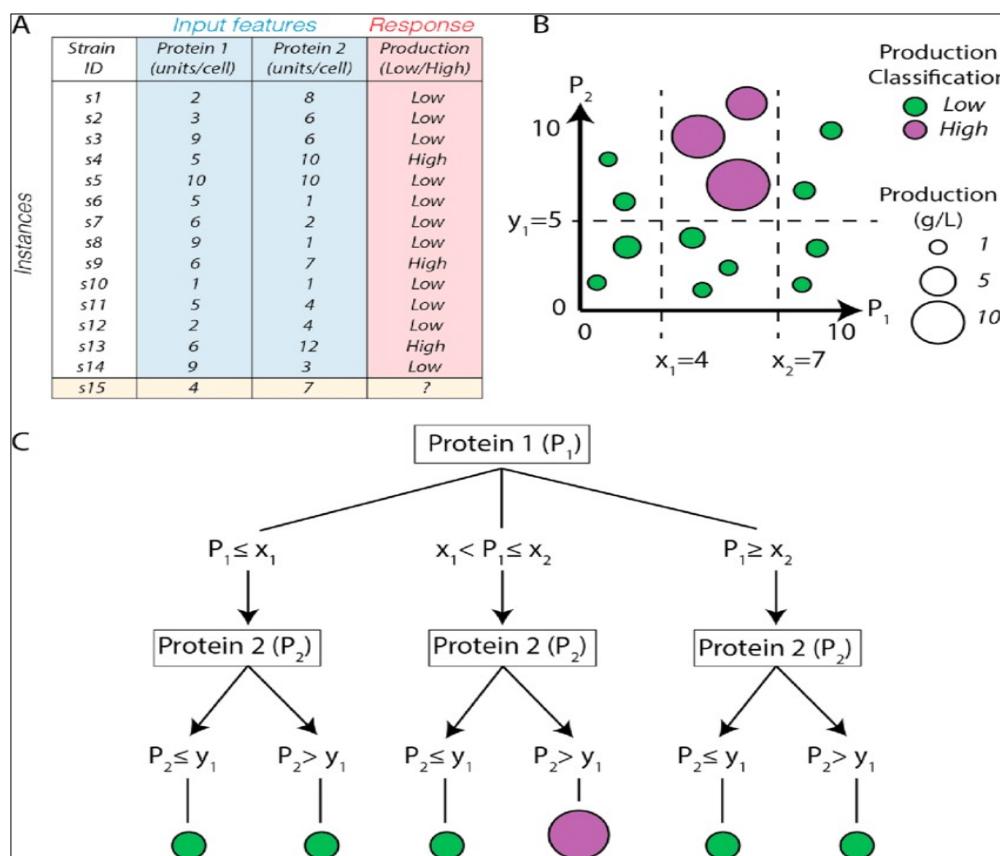


Figura 30. Datos train (A), divisiones que genera en los datos (B) y Árbol de decisión para clasificación (C).

- **Si usamos el árbol para regresión** (buscamos un valor numérico) utilizamos el error cuadrático medio entre el valor real de los datos del grupo (su columna y) y el valor predicho (predicción):

$$\text{Coste en regresión } \text{SME} = 1/n * \text{suma}(y - \text{predicción})^2$$

La función anterior se aplica a todas las divisiones candidatas y se escoge la que menor coste genera (la que minimiza el error). Se pueden usar otras funciones de coste.

ID3 (ITERATIVE DICOTHOMISER)

ID3 y CART se inventaron de forma independiente en la misma época (entre 1970 y 1980), siguen un enfoque similar para el aprendizaje basado en árboles de decisión a partir de tuplas de entrenamiento. ID3 solo se utiliza para clasificar y solamente puede utilizar datos categóricos (no numéricos continuos).

Algoritmo ID3(ejemplos, target, atributos)

Si todas las muestras son positivas devolver un nodo hoja positivo.

Si todos las muestras son negativas devolver un nodo hoja negativo.

```

Si atributos está vacío devolver un nodo hoja con el valor target mayoritario.
Sino
    A ← el MEJOR atributo de atributos (según entropía)
    Para cada valor v de atributo A hacer
        ejemplos(v) ← el subconjunto de atributos cuyo valor de atributo A es v
        Si ejemplos(v) está vacío devolver nodo hoja con valor target más repetido
        Sino devolver ID3(ejemplos(v), target, atributos - {A})

```

Observa que la construcción del árbol se hace forma recursiva, siendo las 3 primeras líneas y la penúltima los casos base que construyen los nodos hojas. La elección del mejor atributo se decide mediante la entropía. Eligiendo aquel que proporciona una mejor ganancia de información. La función elegida puede variar, pero en su forma más sencilla es como esta:

$$-\left(\frac{|p|}{|d|}\right)\log_2\left(\frac{|p|}{|d|}\right) - \left(\frac{|n|}{|d|}\right)\log_2\left(\frac{|n|}{|d|}\right)$$

Donde **p** es el conjunto de los ejemplos positivos, **n** el de los negativos y **d** el total de ellos.

C.4.5

Es un algoritmo usado para generar un árbol de decisión desarrollado por *Ross Quinlan*. Es una mejora de ID3 desarrollado anteriormente también por *Quinlan*. Los árboles de decisión generados por C4.5 pueden ser usados para clasificación.

C4.5 construye árboles de decisión de la misma forma que ID3 usando el concepto de entropía. En cada nodo del árbol, C4.5 elige el mejor atributo usando la ganancia de información (como diferencia de entropía) que se obtiene al dividir los datos. El atributo con la mayor ganancia de información normalizada se elige como parámetro de decisión. Este algoritmo tiene unos pocos casos base.

- Si todos los ejemplos del grupo tienen el mismo valor de target (clase), crea un nodo de hoja para esa clase.
- Si ninguna de las características proporciona ganancia de información, crea un nodo de decisión más arriba utilizando el valor esperado de la clase.
- Aparece un valor de la clase no vista previamente, crea un nodo de decisión más arriba con el valor esperado.

Algoritmo C4.5(**ejemplo, target, atributos**)

```

Comprobar los casos base
mejor_gi ← 0, mejor_a ← ninguno, mejor_si ← ninguno, mejor_no ← ninguno
Para cada atributo a de atributos hacer
    gi ← ganancia de información normalizada de dividir ejemplos por a en grupo_si, grupo_no
    si gi > mejor_gi entonces
        mejor_gi ← gi
        mejor_a ← a
        mejor_si ← grupo_si, mejor_no ← grupo_no
    finsi
    Crear un nodo de decisión (mejor_a, mejor_si, mejor_no)
    C4.5(mejor_si, target, atributos)
    C4.5(mejor_no, target, atributos)

```

En C4.5 se hicieron algunas mejoras respecto a ID3:

- Trabaja con atributos continuos y discretos. Para atributos continuos, C4.5 crea un umbral y se divide la lista en aquellos cuyo valor es superior al umbral y los que son menores o iguales a él.
- Trabaja con valores ausentes: los valores ausentes no se usa en los cálculos de la ganancia de la entropía.

- Permite definir pesos a diferentes atributos.
- Poda después de la creación: C4.5 recorre hacia arriba el árbol tras crearlo e intenta eliminar ramas que no ayudan, reemplazándolas con nodos hoja.

C5.0/See5

Quinlan continuó con la creación del C5.0 y del See5 (C5.0 para Unix/Linux, See5 para Windows) con fines comerciales. C5.0 ofrece una serie de mejoras con respecto al C4.5. Algunas de estas son:

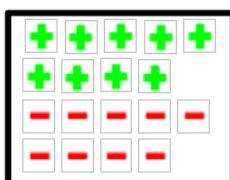
- **Más escalable:** C5.0 es significativamente más rápido que C4.5 (varias órdenes de magnitud) y consume menos memoria.
- **Árboles más pequeños:** obtiene resultados similares a C4.5 con árboles más pequeños.
- **Soporta boosting:** el boosting mejora los árboles y les da una mayor precisión.
- **Soporta ponderación:** permite ponderar los distintos casos y errores de clasificación.
- **Aplica Winnowing:** una opción automática de C5.0 consiste en aplicar un algoritmo de clasificación (algoritmo Winnow) a los atributos para eliminar los que influyen poco.

CART

A este algoritmo le dedicaremos su propio apartado más adelante porque permite utilizar tanto datos categóricos como datos numéricos y se puede utilizar en clasificación como en regresión.

3.2. MÉTRICAS PARA MEDIR LO BUENA QUE ES UNA DIVISIÓN.

Los algoritmos de aprendizaje que construyen los árboles de decisión suelen trabajar de manera *top-down*: escogen en cada paso la variable y el dato que mejor dividen el conjunto de elementos. Diferentes algoritmos utilizan diferentes métricas para decidir cuál es la "mejor" división. Las métricas miden generalmente la homogeneidad de la variable target en los subconjuntos de datos. Estas métricas se calculan en cada nuevo subconjunto y sus valores se combinan (por ejemplo un promedio) para proporcionar una medida de lo buena que es la división. Por ejemplo, como se ve en la figura, tenemos unos datos formados por 18 ejemplos, de los cuales 9 son de la clase + y los otros 9 son de la clase - (es un conjunto de datos perfectamente balanceado).



Se nos ocurren dos preguntas para separarlos en dos grupos: pregunta izquierda y pregunta derecha. ¿Cuál es la pregunta que mejor divide a los datos? Bueno, pues la mejor pregunta (que equivale a la mejor división) es la que genere grupos de datos más homogéneos o más parecidos o más puros, es decir, grupos menos diferentes o menos impuros o menos mezclados.

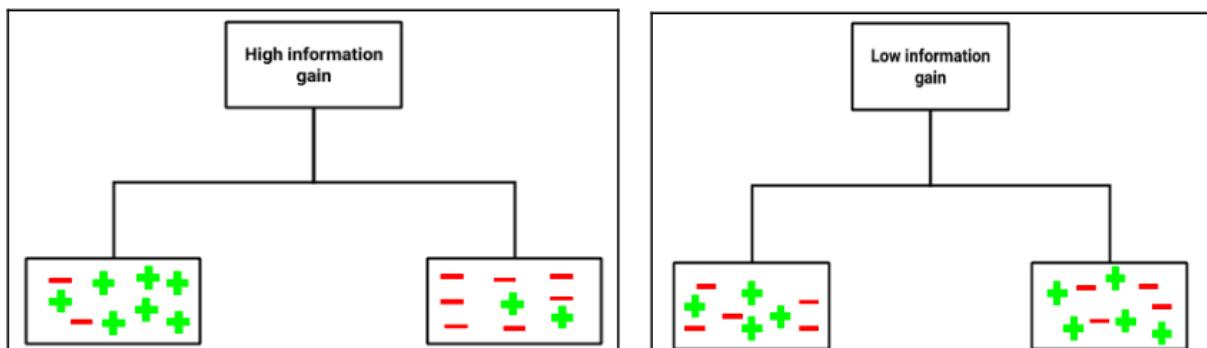


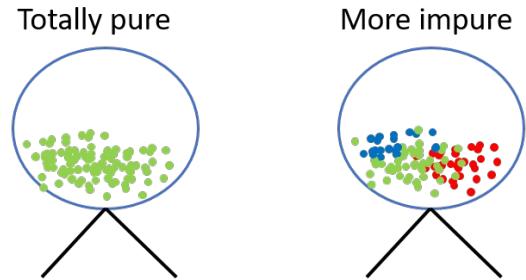
Figura 31: 2 divisiones, la mejor es la que más ganancia de información tenga.

Si las preguntas producen estas divisiones, la mejor es la izquierda porque los grupos de datos que genera son más similares en la izquierda ($[7+, 2-]$ y $[2+, 7-]$) mientras que la pregunta derecha genera grupos más mezclados ($[4+, 5-]$ y $[5+, 4-]$).

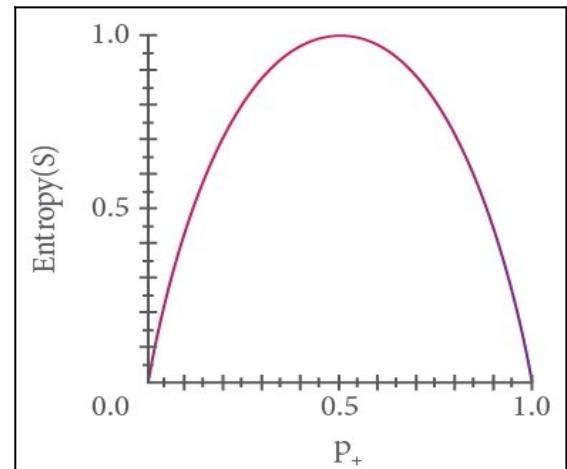
ENTROPÍA DE UN CONJUNTO S: $H(S)$

Shannon definió el concepto de entropía de un conjunto S y se escribe como $H(S)$, como una medida de su impureza. Es un concepto que surge en la teoría de la información y ha saltado también a la física y a las matemáticas. Cuando algo se genera aleatoriamente se dice que es impuro y como la entropía mide la impureza, cuanto más aleatorio, más entropía tiene. La entropía es como la ausencia de orden, puro azar. **A mayor entropía, más azar, más desorden, menos orden y menos información.** Por tanto, ganar información es disminuir la entropía.

La idea detrás de la entropía simplificando: una rueda de bingo con 100 bolas verdes es un conjunto totalmente homogéneo o puro (todos los elementos iguales) por tanto su entropía es 0 (impureza cero). Si cambiamos 30 de estas bolas por bolas rojas y 20 por bolas azules, al sacar una bola al azar del nuevo bombo la probabilidad de recibir una verde ha bajado de 1 a 0.5. Se debe a que la entropía ha subido.



La entropía es 0 si todos los elementos de un conjunto S pertenecen a la misma clase. Por ejemplo, si tenemos dos clases (un problema de clasificación binaria: clase P+ y clase P-) si todos los miembros de un grupo son P+ entonces $H(S)= 0$. De igual modo, si todos los miembros son P-, la entropía también será 0. La entropía es un valor que se mueve entre 0 y 1. En el caso de que haya dos clases diferentes (dos posibilidades) será 1 cuando el 50% de los elementos sea de una clase y el otro 50% de la otra clase. Aquí estaría la gráfica donde en el eje horizontal se representa la proporción de elementos de P+ y en el vertical el valor de la entropía.



Se puede calcular la entropía de un conjunto S que pueda tener elementos de dos clases {P+ y P-} como la suma de sus probabilidades por los logaritmos de sus probabilidades con signo inverso. Si llamamos p a la probabilidad o la proporción de elementos P+ y q a la probabilidad de los elementos P- o la proporción de esos elementos ($q = 1-p$), la fórmula para calcular la entropía es:

$$\text{entropía}(S) = H(S) = -p \log_2(p) - q \log_2(q)$$

La entropía puede utilizarse con variables target que sean categóricas. Los algoritmos que la utilizan como métrica para decidir la mejor división cuando construyen un árbol de decisión lo hacen escogiendo la división que tiene la entropía más baja en comparación con el nodo principal y otras divisiones. Cuanto menor sea la entropía, mejor.

La ganancia de información que genera aplicar una división en un nodo se calcula como la diferencia entre la entropía antes de la división y la media ponderada de las entropías de los

conjunto de datos hijos (los pesos son proporcionales a la cantidad de filas total que tienen, a más filas más peso):

$$\text{total} = \text{datos}_{izq} + \text{datos}_{der}$$

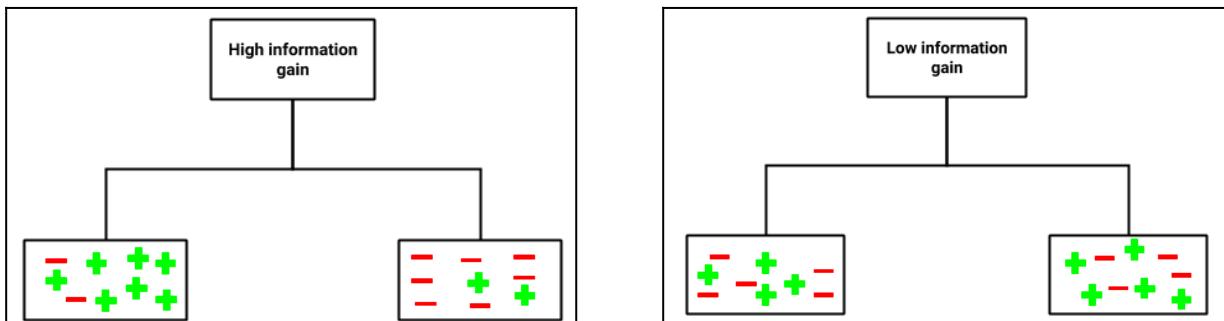
$$\Delta \text{Información} = H(\text{padre}) - \left(\frac{\text{datos}_{izq}}{\text{total}} H(\text{datos}_{izq}) + \frac{\text{datos}_{der}}{\text{total}} H(\text{datos}_{der}) \right)$$

$$\text{Ganancia de información} = H(\text{nodo_padre}) - [\text{media } H(\text{nodos_hijos})]$$

Pasos para calcular la entropía para una división:

1. Calcular la entropía del nodo principal
2. Calcular la entropía de cada nodo individual de división y calcular el promedio ponderado de todos los subnodos disponibles en una división.

EJEMPLO 29: Calcular la ganancia de información de estas dos divisiones e indicar cuál es la mejor y por tanto la que aplicaría un algoritmo de aprendizaje de árboles de decisión.



$H(\text{padre}) = 1$ (porque tiene el 50% de los elementos de la clase + y el 50% restante de la clase -)

División izquierda:

$$\begin{aligned} H(\text{hijo_izquierdo}) &= -(7/9) \log_2(7/9) - (2/9) \log_2(2/9) = \\ &= -(0,7778) (-0,36257) - (0,22222) (0,301029) = \\ &= 0,2819996946 - 0,06689466438 = 0,21510503022 \end{aligned}$$

$$\begin{aligned} H(\text{hijo_derecho}) &= -(2/9) \log_2(2/9) - (7/9) \log_2(7/9) = \\ &= -0,06689466438 + 0,2819996946 = 0,21510503022 \end{aligned}$$

$$\begin{aligned} \text{Ganancia de información} &= 1 - (0,5 * 0,21510503022 + 0,5 * 0,21510503022) = \\ &= 1 - 0,21510503022 = 0,78489496978 \end{aligned}$$

División derecha:

$$\begin{aligned} H(\text{hijo_izquierdo}) &= -(4/9) \log_2(4/9) - (5/9) \log_2(5/9) = \\ &= -(0,44444) (0,30103) - (0,55555) (-0,84799) = \\ &= 0,51996 + 0,47111 = 0,99107 \end{aligned}$$

$$H(\text{hijo_derecho}) = -0,4711 + 0,51996 = 0,99107$$

$$\begin{aligned} \text{Ganancia de información} &= 1 - (0,5 * 0,99107 + 0,5 * 0,99107) = \\ &= 1 - 0,99107 = 0,00893 \end{aligned}$$

Lógicamente, la mejor división es la izquierda porque genera grupos con más información, elementos más homogéneos en cada grupo y por tanto la ganancia es mayor.

ÍNDICE GINI

Mide el grado de impureza de un nodo. Si es igual a cero significa que sus datos son puros (todos los elementos tienen la misma categoría). Si su valor está entre cero y uno significa que el nodo tiene impurezas (es decir, tiene datos de varias categorías o clases). Es la métrica que utiliza el algoritmo CART. Aunque en realidad es muy parecida a la entropía y generan árboles similares, por comentar algunas diferencias:

- *Gini* es un poco más rápido de calcular.
- La entropía genera árboles más equilibrados.
- *Gini* tiende a aislar la clase más frecuente en su propia rama.

Para calcular la impureza *Gini* de un conjunto de elementos, supongamos que cada elemento del conjunto toma valores en las clases $\{c_1, c_2, \dots, c_m\}$ y sea f_i la proporción o frecuencia relativa de elementos etiquetados con valor c_i en el conjunto:

$$I_G(f) = \sum_{i=1}^m f_i(1 - f_i) = \sum_{i=1}^m (f_i - f_i^2) = \sum_{i=1}^m f_i - \sum_{i=1}^m f_i^2 = 1 - \sum_{i=1}^m f_i^2$$

Por ejemplo, si tenemos conjuntos con dos clases (bolas rojas y bolas verdes) su índice *Gini* se calcularía para cada uno de esta manera:

	$1 - (10/10)^2 - (0/10)^2 = 1 - 1 - 0 = 0$
	$1 - (0/10)^2 - (10/10)^2 = 1 - 0 - 1 = 0$
	$1 - (7/10)^2 - (3/10)^2 = 1 - 0,49 - 0,09 = 0,42$
	$1 - (5/10)^2 - (5/10)^2 = 1 - 0,25 - 0,25 = 0,5$

Para calcular el *Gini* de la división a realizar en un nodo padre, se calculan los *Gini* de cada subconjunto de datos que se generan al aplicar la división del conjunto original (*gini_izquierdo* y *gini_derecho*) y se promedian ponderando la cantidad de datos que tiene cada uno para tener más influencia el que más datos tiene:

$$\text{gini_división} = \frac{n_{\text{izquierda}}}{n_{\text{izquierda}} + n_{\text{derecha}}} * \text{gini}_{\text{izquierda}} + \frac{n_{\text{derecha}}}{n_{\text{izquierda}} + n_{\text{derecha}}} * \text{gini}_{\text{derecha}}$$

EJEMPLO 30: Calcula el índice *Gini* de realizar la división por la pregunta “¿dolor de pecho?” con respuesta si/no.



Primero se calcula el *Gini* de cada hijo:

$$1 - (\text{probabilidad de Sí})^2 - (\text{probabilidad de No})^2$$

$$1 - (105/(105+39))^2 - (39/(105+39))^2 = 0,395$$

$$1 - (34/(34+125))^2 - (125/(34+125))^2 = 0,336$$

El coeficiente final será la suma ponderada de ambos (porque cada hijo tiene diferente cantidad de pacientes: 144 y 159):

$$0.395 \cdot (144/(144+159)) + 0.336 \cdot (159/(144+159)) = 0.364$$

3.3. ALGORITMO CART.

El algoritmo *CART* para clasificación usará la métrica *Gini* para ir dividiendo recursivamente los datos del dataset de entrenamiento escogiendo la mejor división. Si no encuentra una división con menor *Gini* que la del nodo, no divide el nodo. Si deja a los nodos hoja puros (todos los ejemplos de la misma clase) será propenso a tener *overfitting* y no va a generalizar bien, por ese motivo, se suelen aplicar reglas para limitar la cantidad de divisiones que puede hacer:

- Una cantidad mínima de ejemplos en cada nodo. Si se alcanza, en vez de dividirlo de nuevo se asigna al nodo la categoría que más aparezca y se convierte en nodo hoja.
- Una altura máxima (distancia máxima con la raíz del árbol). Mismo criterio que en el caso anterior.
- Procesos de poda (eliminación de hojas o ramas) posteriores a la construcción del árbol.

El algoritmo sería:

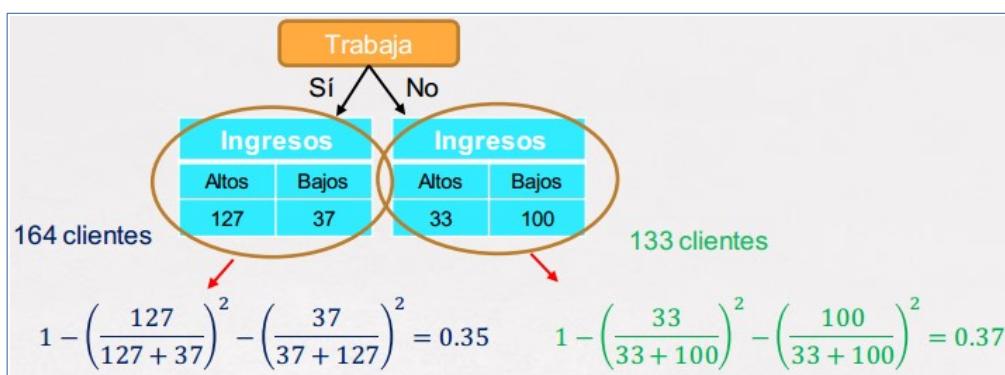
```

algoritmo CART(ejemplos, nodo)
    gini_actual <- gini(ejemplos)
    mejor_hijo_iz <- ninguno
    mejor_hijo_der <- ninguno
    mejor_gini <- gini_actual
    para cada característica c de los ejemplos hacer
        para cada valor v de prueba de la característica c hacer
            hi, hd <- divide_datos(ejemplos, c, v)
            gini <- gini_division(hi, hd)
            si gini < mejor_gini entonces
                mejor_gini <- gini
                mejor_hijo_iz <- hi
                mejor_hijo_der <- hd
            fisi
        finpara
    finpara
    si mejor_gini < gini_actual entonces
        Añade a ejemplos la pregunta (c,v) y los hijos (mejor_hi, mejor_hd)
        si es divisible(mejor_hi) entonces CART(mejor_hi) sino asigna_categoria(mejor_hi) finsi
        si es divisible(mejor_hd) entonces CART(mejor_hd) sino asigna_categoria(mejor_hd) finsi
    sino
        asigna_categoria(ejemplos)
    finsi
finalgoritmo

```

En estas figuras vemos dos posibles divisiones de un nodo y hay que decidir cuál es la mejor. Ninguna consigue obtener grupos sin impurezas, así que calculamos el *Gini* de la división:

		Trabaja				Hipoteca	
		Sí	No			Sí	No
		Ingresos		Ingresos		Ingresos	
		Altos	Bajos	Altos	Bajos	Altos	Bajos
		37	127	100	33	105	39
							34
							125



Una vez calculado el *Gini* de cada posible nodo hoja, se calcula el *Gini* de la división por esa característica como la suma ponderada de todos sus *Gini*. El *Gini* de dividir por la característica Trabaja con el operador '=' y el valor 'SI' sería:

$$0.35 \cdot \left(\frac{164}{164+133} \right) + 0.37 \cdot \left(\frac{133}{164+133} \right) = 0.36$$

Si realizamos estos cálculos a cada característica por la que podemos dividir tenemos que:

Gini de dividir por Trabaja es 0.36

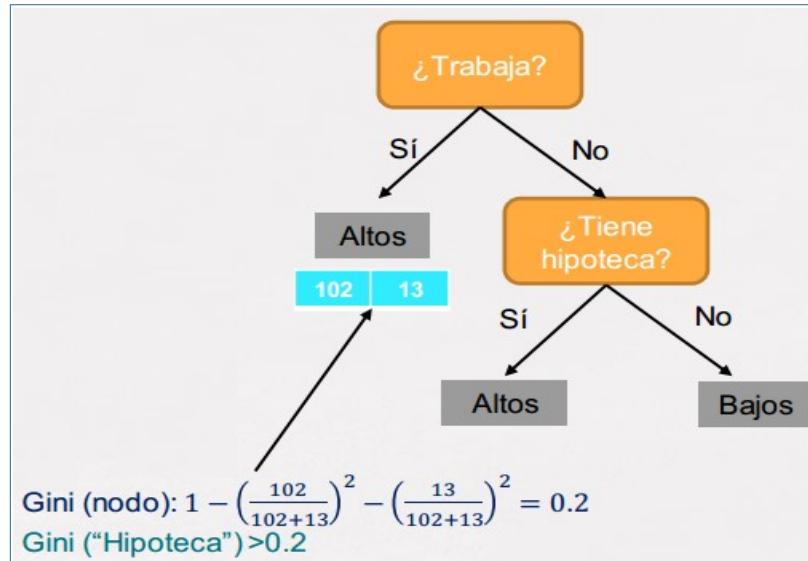
Gini de dividir por Hipoteca es 0.364

Como dividir por Trabaja tiene menor impureza, se escoge como la mejor opción de división.

Ahora continuamos repitiendo el proceso de manera recursiva a los nuevos nodos generados. Observa que cuando en un grupo todas las filas son de la misma categoría, su gini es 0.

Puede ocurrir que **si al dividir un nodo, ninguna característica consiga una división con un Gini inferior al del propio nodo padre**

(los datos sin separar) se detiene el proceso de división y el nodo no se separa más. Por ejemplo en la figura, la división consigue igualar el *Gini* del nodo, pero no bajarlo. En ese caso, no se aplica la división.



Cuando se prueban diferentes valores de una característica para buscar la mejor división hay que distinguir dos casos:

- Cuando los valores son simbólicos.
- Cuando los valores numéricos.

VALORES SIMBÓLICOS

Hay que probar a dividir por cada uno de sus diferentes valores y dividir el grupo de filas entre las que tengan todas las posibles combinaciones de esos valores y los que no. Se suelen hacer preguntas del tipo "**característica = valor?**", que separa las filas en dos grupos: **las que efectivamente tienen ese valor y las que no**. Y para cada grupo generado se calcula el *Gini*. Otras alternativas podrían hacer preguntas más complejas:

Estado civil	Ingresos
Soltero	Altos
Casado	Altos
Viudo	Bajos
Casado	Bajos
Soltero	Bajos

Soltero
 Sí → Soltero o Casado
 No → Soltero o Viudo

Casado
 Sí → Soltero o Viudo
 No → Casado o Viudo

Viudo
 Sí → Casado o Viudo
 No → Casado o Viudo

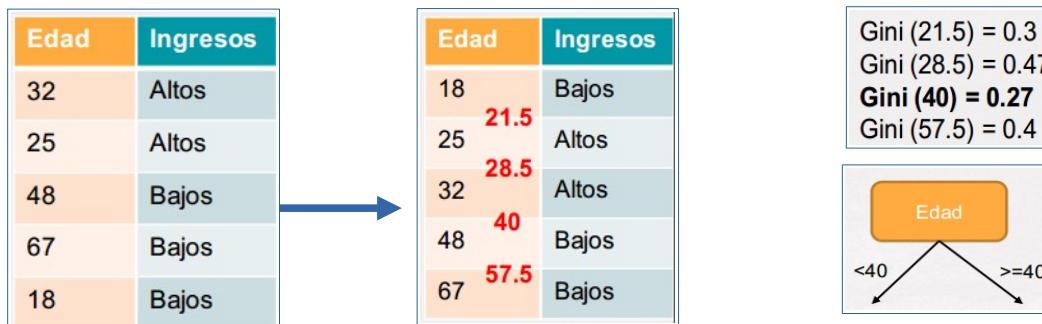
VALORES NUMÉRICOS CONTÍNUOS

En este caso hace lo siguiente:

- Ordena los valores de menor a mayor.
- Hace las sumas de los valores contiguos.
- **Usa estos valores como valores de prueba preguntando "característica < prueba?". Si la respuesta es NO es porque será mayor o igual. El SI de la pregunta contendrá las filas cuyo valor de columna sea inferior al valor de corte y a la derecha (NO) estarán las filas con mayor o igual valor en la columna que el valor de corte.**

- Calcula el *Gini* de cada grupo y valor de prueba y recuerda el que produce la mejor división.

En este ejemplo usa la columna **edad** que tiene valores numéricos. A la izquierda aparecen los valores originales. Ordenamos una copia de los valores numéricos de menor a mayor y en la figura del centro aparecen en rojo las medias de cada dos valores contiguos (21.5, 28.5 ...). Habrá que probar como de bueno es dividir las filas por cada uno de estos valores intermedios de la columna. El *Gini* más bajo es para el valor 40 y por este valor aplica la división de los datos, dejando a la izquierda las filas que tengan una edad < de 40 y a la derecha las filas que tengan una edad mayor o igual de 40 (el resto). La pregunta del nodo sería "**edad < 40?**".



3.4. ENTRENAR, PREDECIR Y VISUALIZAR CON SCIKIT-LEARN.

Scikit-learn utiliza el algoritmo *CART* que genera solamente árboles de decisión binarios, que puede manejar tanto datos categóricos como numéricos continuos y puede utilizar clasificación binaria o múltiple usando la métrica *Gini*. Las clases de árboles de decisión están en el módulo **sklearn.tree**. El siguiente código entrena un **DecisionTreeClassifier** con el dataset Iris.

EJEMPLO 31: Entrenar un árbol de decisión, dibujarlo, hacer predicciones y dibujar las fronteras.

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
from sklearn import tree
# Cargar datos
iris = load_iris()
X = iris.data[:, 2:] # longitud y anchura de pétalos
y = iris.target
# Entrenar el modelo
arbol = DecisionTreeClassifier(max_depth=2, random_state=42)
arbol.classes_ = ["Setosa", "Virginica", "Versicolor"]
arbol.fit(X, y)
tree.plot_tree(arbol, filled=True)
# hacer predicciones
while True:
    petalo_largo = float(input("Longitud del pétalo (-1 para salir):"))
    if petalo_largo < 0:
        break
    petalo_ancho = float(input("Anchura del pétalo:"))
    ejemplos = [[petalo_largo, petalo_ancho]]
    print("Predicción: ", arbol.predict(ejemplos))
    print(f"Probabilidades de: {arbol.classes_}", arbol.predict_proba(ejemplos))
# ----- AÑADIR ESTE TROZO AL FICHERO U03_tools.py -----
# Dibujar fronteras de decisión
from matplotlib.colors import ListedColormap

def plot_fronteras_decision(arbol, X, y, ejes=[0, 7.5, 0, 3], iris=True, legend=True,
                           plot_training=True):
    x1s = np.linspace(ejes[0], ejes[1], 100)
    x2s = np.linspace(ejes[2], ejes[3], 100)
    x1, x2 = np.meshgrid(x1s, x2s)
    X_nuevo = np.c_[x1.ravel(), x2.ravel()]
    if plot_training:
        plt.scatter(X[:, 0], X[:, 1], c=y, cmap=ListedColormap(['red', 'blue', 'green']))
    plt.contourf(x1, x2, arbol.predict(X_nuevo), colors=['red', 'blue', 'green'],
                 alpha=0.3)
    if legend:
        plt.legend(loc='best')
    plt.xlabel('Petal length')
    plt.ylabel('Petal width')
```

```

y_pred = arbol.predict(X_nuevo).reshape(x1.shape)
personal_cmap = ListedColormap(['#fafab0', '#9898ff', '#a0faa0'])
plt.contourf(x1, x2, y_pred, alpha=0.3, cmap=personal_cmap)
if not iris:
    personal_cmap2 = ListedColormap(['#7d7d58', '#4c4c7f', '#507d50'])
    plt.contour(x1, x2, y_pred, cmap=personal_cmap2, alpha=0.8)
if plot_training:
    plt.plot(X[:, 0][y==0], X[:, 1][y==0], "yo", label="Iris setosa")
    plt.plot(X[:, 0][y==1], X[:, 1][y==1], "bs", label="Iris versicolor")
    plt.plot(X[:, 0][y==2], X[:, 1][y==2], "g^", label="Iris virginica")
    plt.axis(ejes)
if iris:
    plt.xlabel("Largo Pétalo", fontsize=14)
    plt.ylabel("Ancho Pétalo", fontsize=14)
else:
    plt.xlabel(r"$x_1$", fontsize=16)
    plt.ylabel(r"$x_2$", fontsize=16, rotation=0)
if legend:
    plt.legend(loc="lower right", fontsize=12)
# ----- FIN DEL TROZO A AÑADIR A U03_tools.py -----
from U03_tools import plot_fronteras_decision
import matplotlib.pyplot as plt
plt.figure(figsize=(8, 4))
plot_fronteras_decision(arbol, X, y)
plt.plot([2.45, 2.45], [0, 3], "k-", linewidth=2)
plt.plot([2.45, 7.5], [1.75, 1.75], "k--", linewidth=2)
plt.plot([4.95, 4.95], [0, 1.75], "k:", linewidth=2)
plt.plot([4.85, 4.85], [1.75, 3], "k:", linewidth=2)
plt.text(1.40, 1.0, "Profundidad=0", fontsize=13)
plt.text(3.2, 1.80, "Profundidad=1", fontsize=11)
plt.text(4.05, 0.5, "(Profundidad=2)", fontsize=9)
plt.show()

```

Puedes visualizar el árbol usando la función `tree.plot_tree()` (mira documentación *scikit-learn*).

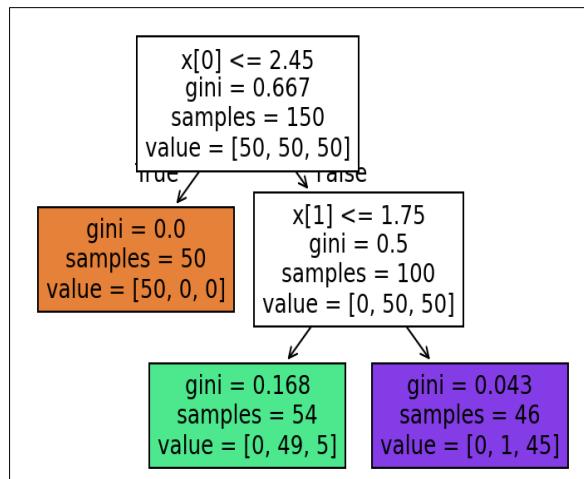


Figura 32: Visualización del árbol de decisión.

Para hacer predicciones usamos el método `predict()` que devuelve la clase que predice y si queremos saber las probabilidades de que pertenezca a cada clase con el método `predict_proba()`, que devuelve un array con un elemento por clase de la columna target. Las probabilidades las estima dividiendo la cantidad de ejemplos de cada clase entre el total de ejemplos.

En la siguiente figura vemos el dibujo de las fronteras de decisión que crea el árbol que acabamos de entrenar. Las líneas verticales continuas son la división del nodo raíz. El resto de fronteras se representan con líneas discontinuas.

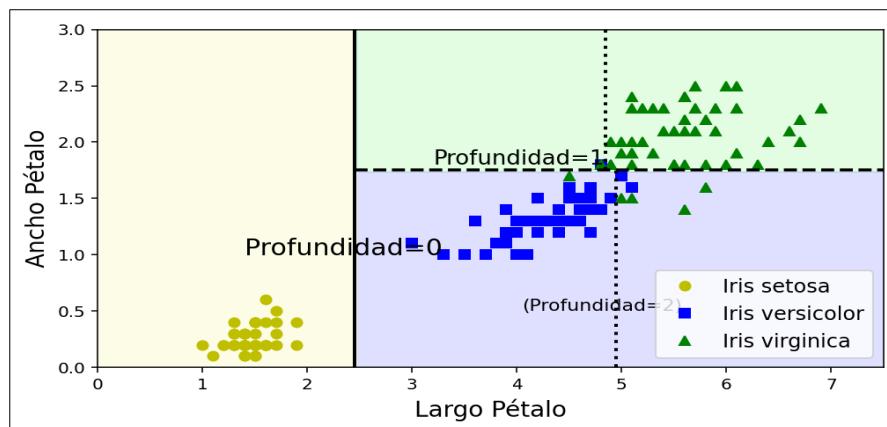


Figura 33: Visualización de las fronteras de decisión del árbol de decisión.

HIPERPARÁMETROS

Hay unos pocos hiperparámetros que controlan las condiciones de parada de la división recursiva: `min_samples_split`, `min_samples_leaf`, `min_weight_fraction_leaf` y `max_leaf_nodes`.

El algoritmo CART es de tipo *greedy*: busca una división óptima de manera exhaustiva en un nivel, y repite el proceso en cada subnivel. No comprueba cuando una división ha llegado o no al nivel más bajo posible de impureza. Con frecuencia produce una solución razonablemente buena, pero no hay garantías de que sea la solución óptima.

Desafortunadamente, encontrar una solución óptima es un problema de tipo **NP-Completo**¹²: problemas que tienen un orden $O(\exp(m))$, lo que hace que sean intratables hasta para pequeños dataset. Por eso CART encuentra “soluciones razonablemente buenas”.

Hacer predicciones exige atravesar el árbol de decisión desde la raíz a una hoja. Los árboles generalmente están bien balanceados, así que recorrer el árbol necesita examinar $O(\log_2(m))$ nodos¹³. Como en cada nodo solo hay que comprobar un valor, la sobrecarga de realizar toda una predicción es también $O(\log_2(m))$ con independencia del número de características. Es rápido.

Sin embargo, el algoritmo de entrenamiento debe examinar todas las características predictoras (menos si se indica `max_features`) de todos los ejemplos en cada nodo. Esto genera una complejidad de $O(n \times m \log(m))$. Para pequeños datasets de entrenamiento (menos de unos miles de instancias), *Scikit-Learn* puede acelerar el proceso entrenando unos datos preordenados (indicando `presort=True`), pero esto mismo ralentiza el proceso para datasets grandes.

Los árboles de decisión hacen pocas suposiciones sobre los datos de entrenamiento (al contrario que los modelos lineales, que obviamente asumen que los datos están distribuidos linealmente, por ejemplo). Esto es así porque los árboles no tienen ninguna restricción en cómo ajustar las superficies de decisión a los datos, esto que en principio es muy bueno, acaba haciendo que sean muy propensos a padecer *overfitting*. Este tipo de modelos a menudo reciben el nombre de **no**

12 *P* es el conjunto de problemas que pueden resolverse en un tiempo polinomial. *NP* es el conjunto de problemas cuyas soluciones pueden verificarse en un tiempo polinomial. Un problema *NP-Hard* es un problema que puede ser reducido a problemas en tiempo polinomial. Un problema *NP-Completo* es tanto *NP* como *NP-Hard*. Una cuestión matemática pendiente de resolver es si *P* = *NP*. Si *P* ≠ *NP* (lo que parece), no hay algoritmo polinomial que exista para resolver cualquier problema *NP-Completo* (salvo quizás en una computadora cuántica).

13 \log_2 es el logaritmo en base 2. $\log_2(m) = \log_{10}(m) / \log_{10}(2) = \ln(m) / \ln(2)$.

paramétricos, no porque no tengan parámetros (que tienen muchos) sino porque su número está indeterminado hasta que no acaban de entrenar. En contraste, los modelos paramétricos como un modelo lineal, antes de entrenar ya sabe cuántos parámetros debe calcular y eso hace que sus grados libertad estén limitados reduciendo la posibilidad de tener *overfitting*.

EVITAR EL OVERFITTING

Identificamos que el origen del *overfitting* en los árboles de decisión es la facilidad con la que se ramifican (adquieren complejidad) y terminan ajustándose perfectamente a las observaciones de entrenamiento. Las estrategias para prevenir este problema son: limitar el tamaño (con parada temprana) y la poda (*pruning*).

Controlar el tamaño del árbol (parada temprana)

El tamaño final que adquiere un árbol puede controlarse mediante reglas de parada que detengan la división de los nodos dependiendo de si se cumplen o no determinadas condiciones. El nombre de estas condiciones puede variar dependiendo del software o librería empleada, pero suelen estar presentes todos ellos. Ejemplos:

- **Instancias mínimas en división:** número mínimo de ejemplos que debe tener un nodo para poder dividirse. A mayor valor, menos flexible es el modelo.
- **Instancias mínimas de nodo hoja:** número mínimo de observaciones que deben tener los nodos hoja. Su efecto es muy similar al anterior.
- **Profundidad máxima del árbol:** define la profundidad o el número de divisiones de la rama más larga (en sentido descendente) del árbol.
- **Número máximo de nodos terminales:** define el número máximo de nodos terminales que puede tener el árbol. Una vez alcanzado el límite, se detienen las divisiones. Su efecto es similar al de controlar la profundidad máxima del árbol.
- **Reducción mínima de error:** define la reducción mínima de error que tiene que conseguir una división para que se lleve a cabo.

Poda del árbol

La estrategia de controlar el tamaño del árbol mediante reglas de parada tiene un inconveniente, el árbol se crece seleccionando la mejor división en cada momento. Al evaluar las divisiones sin tener en cuenta las que vendrán después, nunca se elige la opción que genera el mejor árbol final, a no ser que también sea la que genera en ese momento la mejor división. A este tipo de estrategias se les conoce como *greedy*. Un ejemplo que ilustra el problema de este tipo de estrategia es el siguiente: imagina que un coche circula por el carril izquierdo de una carretera de dos carriles en la misma dirección. En el carril que se encuentra hay muchos coches circulando a 100 km/h, mientras que el otro carril se encuentra vacío. A cierta distancia se observa que hay un vehículo circulando por el carril derecho a 20 km/h. Si el objetivo del conductor es llegar a su destino lo antes posible tiene dos opciones: cambiarse de carril o mantenerse en el que está. Una aproximación de tipo *greedy* evalúa la situación en ese instante y determinaría que la mejor opción es cambiarse de carril y acelerar a más de 100 km/h, sin embargo, a largo plazo, esta no es la mejor solución, ya que una vez alcance al vehículo lento, tendrá que reducir mucho su velocidad.

Una alternativa no *greedy* consiste en generar árboles grandes y complejos, sin condiciones de parada más allá de las necesarias por las limitaciones computacionales, para después podarlos, manteniendo únicamente la estructura robusta que consigue un *test error* bajo. La selección del *subárbol* óptimo puede hacerse mediante *cross-validation*, sin embargo, dado que los árboles se crecen lo máximo posible (tienen muchos nodos terminales) no suele ser viable estimar el *test error*

de todas las posibles subestructuras que se pueden generar. En su lugar, se recurre al *cost complexity pruning* o *weakest link pruning*.

- *Cost complexity pruning* es un método de penalización de tipo *Loss + Penalización*, similar al empleado en *ridge* o *lasso*. En este caso, se busca el *subárbol* TT que minimiza la ecuación donde $|T|$ es el número de hojas del árbol.

$$\sum_{j=1}^{|T|} \sum_{i \in R_j} (y_i - \hat{y}_{R_j})^2 + \alpha|T|$$

En la implementación de *Scikit-learn* para evitar el overfitting a los datos de entrenamiento debes bajar los grados de libertad al algoritmo de entrenamiento, es decir, aplicar lo que hemos llamado *regularización*. El parámetro de regularización depende del algoritmo de división que se use, pero puedes restringir la profundidad máxima que puede alcanzar el árbol con el hiperparámetro ***max_depth*** (cuyo valor por defecto es *None*, que significa ilimitado).

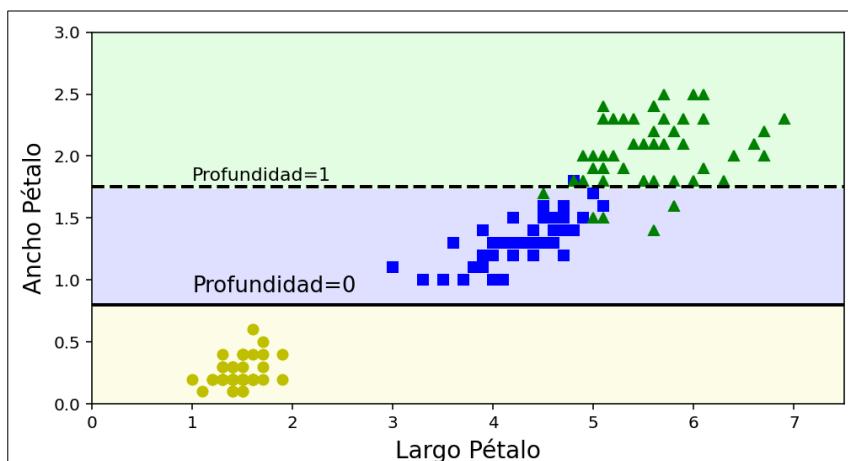


Figura 34: Árbol restringido con hiperparámetro *max_depth* = 2.

EJEMPLO 32: uso de hiperparámetros de los árboles de decisión para clasificación.

```
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeClassifier
from u03_tools import plot_fronteras_decision

iris = load_iris()
X = iris.data[:, 2:] # longitud y anchura de pétalos
y = iris.target
arbol_regularizado = DecisionTreeClassifier(max_depth=2, random_state=40)
arbol_regularizado.fit(X, y)
plt.figure(figsize=(8, 4))
plot_fronteras_decision(arbol_regularizado, X, y, legend=False)
plt.plot([0, 7.5], [0.8, 0.8], "k-", linewidth=2)
plt.plot([0, 7.5], [1.75, 1.75], "k--", linewidth=2)
plt.text(1.0, 0.9, "Profundidad=0", fontsize=13)
plt.text(1.0, 1.80, "Profundidad=1", fontsize=11)
```

Hay otros hiperparámetros que restringen la forma que puede adoptar el árbol de decisión y que aumentarlos o disminuirlos implica regularizar el modelo:

- ***min_samples_split*** indica el mínimo número de ejemplos que un nodo del árbol puede tener antes de que pueda ser candidato a dividirse.

- `min_samples_leaf` indica el número mínimo de ejemplos que un nodo hoja debe tener.
- `min_weight_fraction_leaf` es igual que el anterior pero expresado como una fracción del número total de características que son examinadas en cada nodo para decidir una división.

EJEMPLO 33: Restringir el árbol de decisión con el hiperparámetro `min_samples_split`.

```
from sklearn.datasets import make_moons
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeClassifier
from u03_tools import plot_fronteras_decision

Xm, ym = make_moons(n_samples=100, noise=0.25, random_state=53)

arbol_1 = DecisionTreeClassifier(random_state=42)
arbol_2 = DecisionTreeClassifier(min_samples_leaf=4, random_state=42)
arbol_1.fit(Xm, ym)
arbol_2.fit(Xm, ym)
fig, ejes = plt.subplots(ncols=2, figsize=(10, 4), sharey=True)
plt.sca(ejes[0])
plot_fronteras_decision(arbol_1, Xm, ym, ejes=[-1.5, 2.4, -1, 1.5], iris=False)
plt.title("Sin restricciones", fontsize=12)
plt.sca(ejes[1])
plot_fronteras_decision(arbol_2, Xm, ym, ejes=[-1.5, 2.4, -1, 1.5], iris=False)
plt.title("min_samples_leaf = {}".format(arbol_2.min_samples_leaf), fontsize=12)
plt.ylabel("")
```

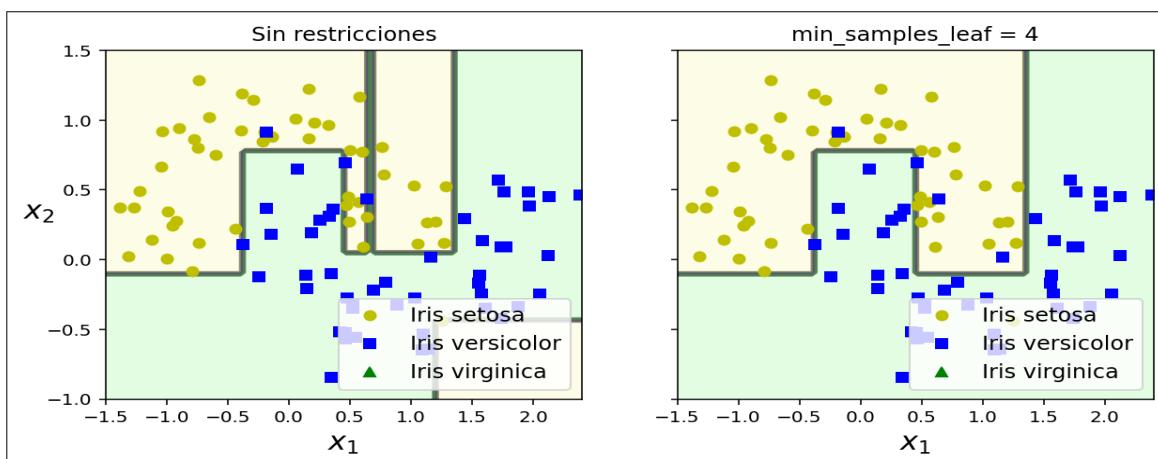


Figura 35: Árbol para moons, izquierda con hiperparámetros por defecto, en derecha con restricciones.

En el gráfico de la derecha, el árbol ha sido entrenado con el hiperparámetro `min_samples_leaf=4`. Es bastante obvio a simple vista que el modelo de la izquierda tiene sobreajuste y que el de la derecha probablemente generalice mejor.

Otros algoritmos distintos al *CART*, primero entranan el árbol sin aplicar ninguna regularización, y luego hacen una poda **pruning** (borrado de nodos innecesarios). Un nodo cuyos hijos tienen todos los nodos hoja es innecesario si la pureza no aporta una **significancia estadística**. Hay test estadísticos como el test Chi cuadrado (χ^2), que se usan para estimar la probabilidad de que una mejora sea simplemente el resultado del azar (hipótesis nula). Si esta probabilidad llamada **pvalue**, es mayor que un límite (normalmente 5%, controlada por un hiperparámetro) se elimina el nodo y se repite el proceso hasta que todos los nodos innecesarios sean podados.

3.4. ÁRBOLES DE DECISIÓN PARA REGRESIÓN.

Los árboles de decisión CART y C4.5 también pueden utilizarse para tareas de regresión. Podemos usar la clase **DecisionTreeRegressor**.

EJEMPLO 34: Entrenar un árbol de decisión CART para regresión con máxima profundidad de 2 sobre un dataset que generamos nosotros mismos para ver que tal lo hace.

```
import numpy as np
np.random.seed(42)
m = 200
X = np.random.rand(m, 1)
y = 3 * (X - 0.5) ** 2
y = y + np.random.randn(m, 1) / 10
from sklearn import tree
from sklearn.tree import DecisionTreeRegressor
import matplotlib.pyplot as plt
arbol_reg = DecisionTreeRegressor(max_depth=2, random_state=42)
arbol_reg.fit(X, y)
tree.plot_tree(arbol_reg, filled=True)
plt.show()
plt.scatter(X,y)
```

Datos cuadráticos + ruido aleatorio
Entrenar modelo
Dibujar datos y modelo

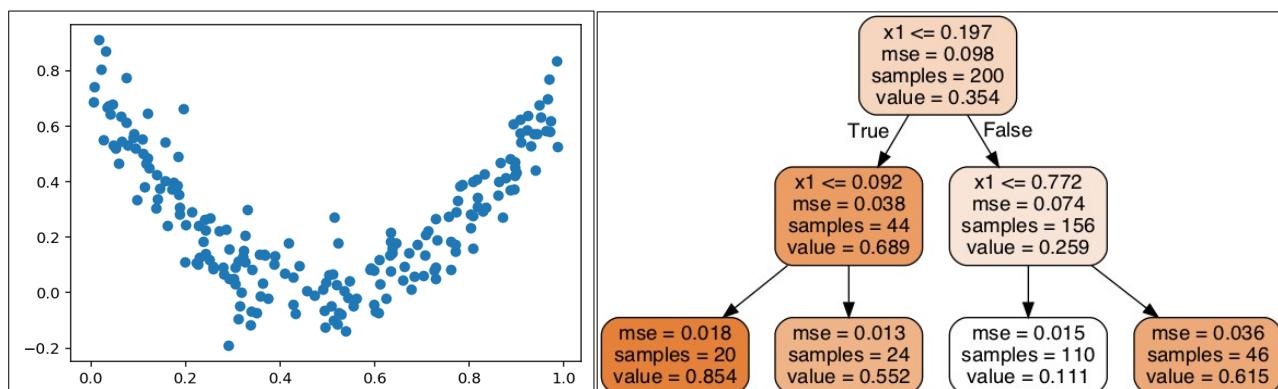


Figura 36. Árbol de decisión para regresión sobre un dataset autogenerado.

La apariencia es muy similar a la de los árboles de clasificación. La principal diferencia es que si quieras hacer una predicción para $x_1 = 0.6$ atraviesas el árbol y alcanzas una hoja que predice un valor de $y = 0.111$. Este valor es simplemente la media de los valores del target de los 110 ejemplos asociados con este nodo hoja. Esta predicción tiene MSE de 0.015 en estos 110 ejemplos.

El algoritmo CART trabaja como antes, pero en vez de dividir los datos de train para minimizar la impureza, ahora intenta minimizar el **MSE** (Medium Square Error):

$$J(k, t_k) = \frac{m_{\text{left}}}{m} \text{MSE}_{\text{left}} + \frac{m_{\text{right}}}{m} \text{MSE}_{\text{right}} \quad \text{where} \quad \begin{cases} \text{MSE}_{\text{node}} = \sum_{i \in \text{node}} (\hat{y}_{\text{node}} - y^{(i)})^2 \\ \hat{y}_{\text{node}} = \frac{1}{m_{\text{node}}} \sum_{i \in \text{node}} y^{(i)} \end{cases}$$

Si fijas el hiperparámetro **max_depth=3**, obtienes las predicciones representadas en el gráfico de la derecha de la figura 37. Los valores predichos para cada región siempre son la media de los valores de los ejemplos en esa región. El algoritmo divide cada región de forma que deja la mayoría de ejemplos tan cerca como es posible del valor predicho.

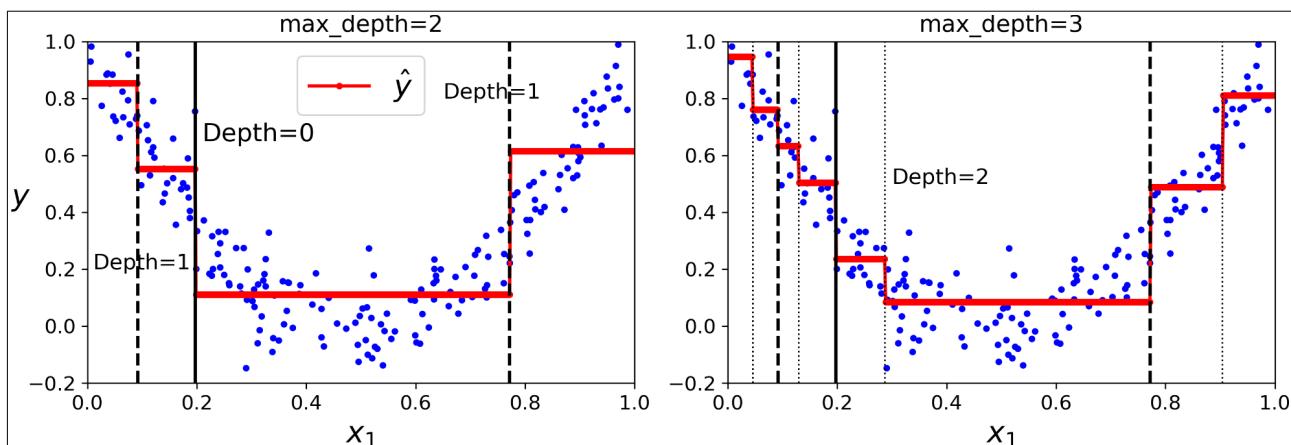


Figura 37. Predicciones de dos árboles de decisión para regresión.

Si entrenas un CART regresor sin restricciones es muy probable que tenga overfitting como se ve en la figura 38 en el gráfico de la izquierda. Si indicas `min_samples_leaf=10` ya se convierte en un regresor más razonable que generalizará mejor, como se ve en esa misma figura en el gráfico de la derecha.

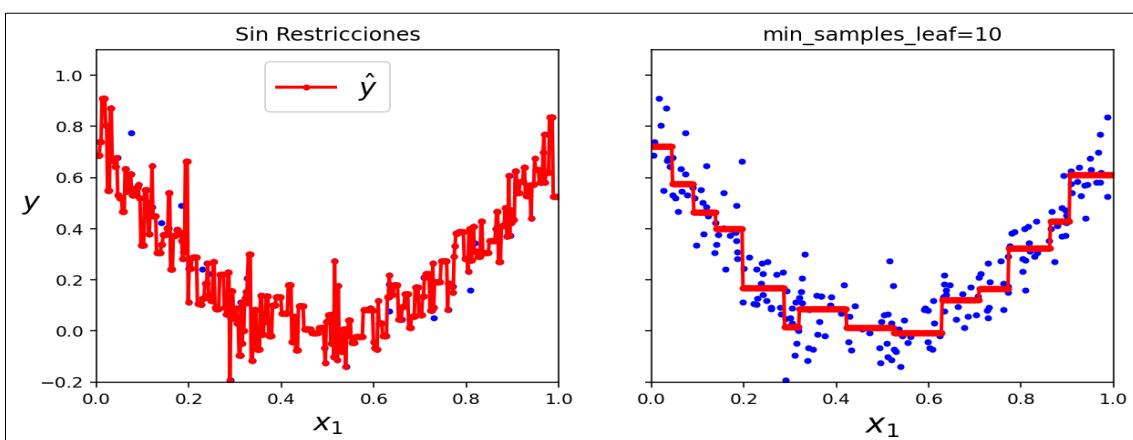


Figura 38. Regularizando un árbol de decisión regresor.

EJEMPLO 35: Código que genera la figura 38, entrenando un árbol de decisión sin restricciones y regularizado limitando la cantidad de mínima de ejemplos por nodo.

```
# Datos cuadráticos + ruido aleatorio
import numpy as np
np.random.seed(42)
m = 200
X = np.random.rand(m, 1)
y = 3 * (X - 0.5) ** 2
y = y + np.random.randn(m, 1) / 10
from sklearn.tree import DecisionTreeRegressor
import matplotlib.pyplot as plt
arbol_r1 = DecisionTreeRegressor(random_state=42)
arbol_r2 = DecisionTreeRegressor(random_state=42, min_samples_leaf=10)
arbol_r1.fit(X, y)
arbol_r2.fit(X, y)
x1 = np.linspace(0, 1, 500).reshape(-1, 1)
y_pred1 = arbol_r1.predict(x1)
y_pred2 = arbol_r2.predict(x1)
fig, ejes = plt.subplots(ncols=2, figsize=(10, 4), sharey=True)
plt.sca(ejes[0])
plt.plot(X, y, "b.")
```

```

plt.plot(x1, y_pred1, "r.-", linewidth=2, label=r"\hat{y}")
plt.axis([0, 1, -0.2, 1.1])
plt.xlabel("$x_1$", fontsize=16)
plt.ylabel("$y$", fontsize=16, rotation=0)
plt.legend(loc="upper center", fontsize=16)
plt.title("Sin Restricciones", fontsize=12)
plt.sca(ejes[1])
plt.plot(X, y, "b.")
plt.plot(x1, y_pred2, "r.-", linewidth=2, label=r"\hat{y}")
plt.axis([0, 1, -0.2, 1.1])
plt.xlabel("$x_1$", fontsize=18)
plt.title("min_samples_leaf={}".format(arbol_r2.min_samples_leaf), fontsize=12)

```

3.5. INESTABILIDAD.

Habráis observado que a los árboles les gusta usar líneas ortogonales (paralelas a los ejes) para tirar sus fronteras de decisión y esto los hace sensibles a las rotaciones. Por ejemplo en la figura 39 un dataset separable linealmente con una línea vertical (izquierda) y a la derecha otro dataset al que le cuesta más trabajo separar. Pero en realidad son el mismo dataset, lo que pasa es que hemos girado el de la derecha 45°. Aunque consigue clasificarlo, es posible que no generalice bien. Una manera de limitar el problema es utilizar **PCA** (lo veremos más adelante) para obtener una orientación más adecuada de los datos.

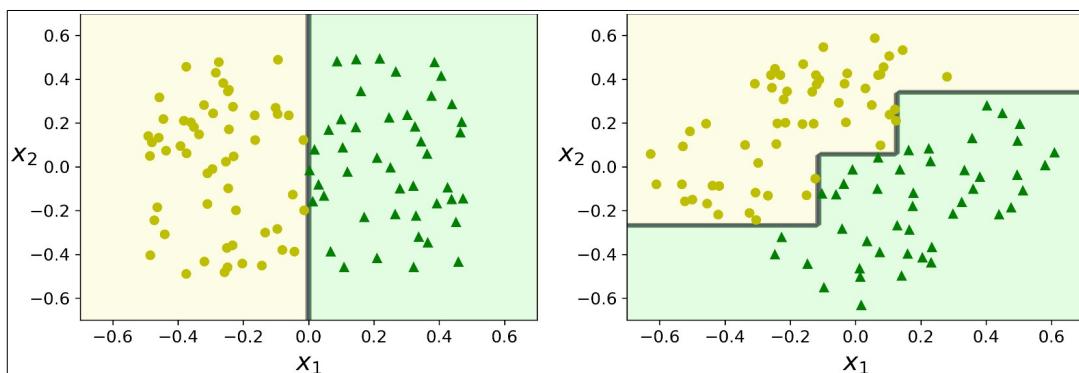


Figure 39. Sensibilidad a las rotaciones de los datos.

Además de a las rotaciones, también les afectan pequeños cambios. Por ejemplo si eliminás la flor *Versicolor* más alejada de los datos de entrenamiento (la que tiene ancho de 4.8 cm y largo de 1.8 cm) y entrenas un nuevo árbol tendrás el modelo de la figura 40. Como ves, un solo dato cambia y el árbol que obtienes es totalmente diferente.

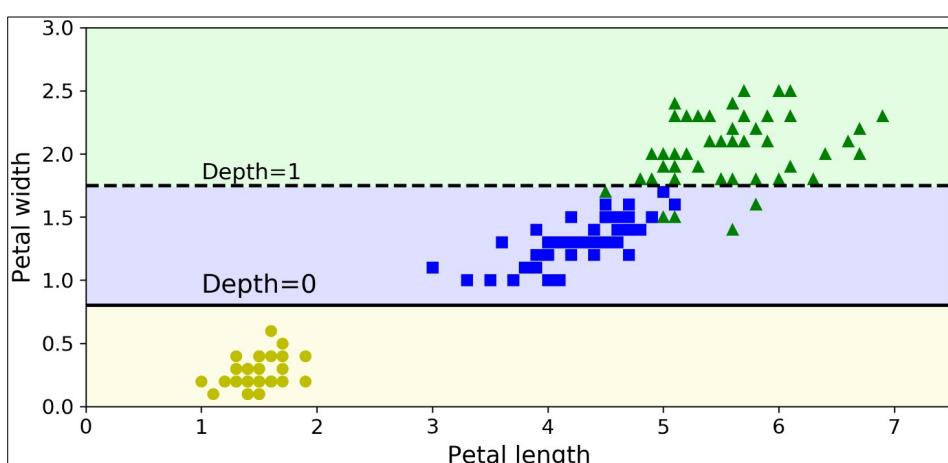


Figura 40. Sensibilidad a cambios en los datos de entrenamiento.

Actualmente el algoritmo usado en *Scikit-Learn* es estocástico¹⁴ y puedes tener hasta más variación (salvo que indiques el hiperparámetro *random_state*).

RandomForest limita esta inestabilidad promediando las predicciones en muchos árboles.

4. MÉTODOS DE ENSAMBLAJE (ENSEMBLES).

Existe un término denominado **la sabiduría de la multitud**. Es un fenómeno estadístico que afirma que si haces un pregunta a mil personas ajenas al tema que preguntas y agregamos sus respuestas obtendremos mejores resultados que solo con la respuesta de un experto en el tema.

Podemos aplicar el mismo fenómeno en Machine Learning. Si unimos (ensamblamos) distintos predictores mediocres, **obtendremos un mejor resultado que con un gran predictor por si solo**. Es lo que se denomina métodos de ensamblaje y es una técnica bastante sencilla pero muy potente con la que podemos obtener grandes resultados en todo tipo de problemas. Ya los hemos visto en la unidad anterior, pero volvemos a ellos ahora que conocemos más algoritmos de aprendizaje.

4.1. VOTING.

Supón que tienes distintos clasificadores implementados y entrenados con los mismos datos: un SVC, un KNN, un modelo de *Regresión Logística* y un *Árbol de Decisión* que de manera individual alcanzan una *accuracy* de alrededor de un 80%.

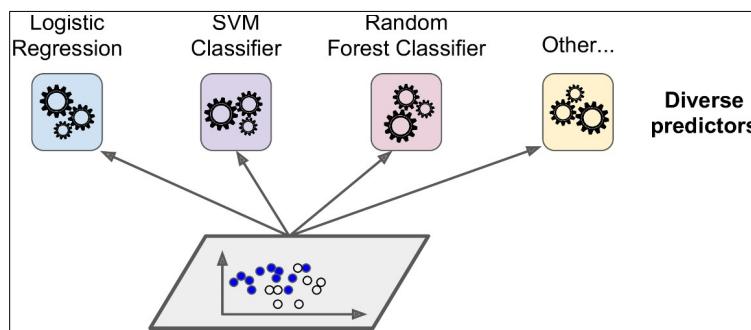


Figura 41. Modelos individuales.

Una manera sencilla de mejorar los resultados es unir sus predicciones: se predice lo que la mayoría de ellos indique. De esta forma conseguimos con unos cuantos clasificadores “normalitos”, competir con el mejor de los clasificadores. Aunque nada impide que los clasificadores individuales también tengan buen desempeño. Este tipo de funcionamiento es *hard voting*.

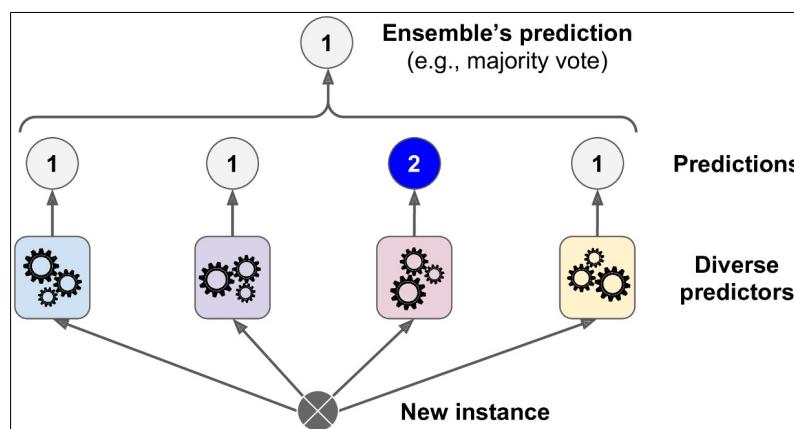


Figura 42. Modelo de ensamblaje de tipo hard-voting.

14 Escoge hiperparámetros al azar en cada ejecución salvo que se le indiquen valores predeterminados.

¿A qué se debe esto? Si lo pensamos, no tiene mucho sentido que por juntar la opinión de unas cuantas personas que no tienen ni idea de lo que están hablando obtengamos una mejor respuesta que si solo preguntamos a un experto en esa materia. Y a pesar de esto, tanto en ese caso como en los métodos de ensamblaje, sucede y se debe a la [ley de los grandes números](#).

Imagina que tiras una moneda que está un poco trucada y el 51% de las veces sale cara y el otro 49% cruz. Si la tiras 1000 veces, normalmente obtendrás 510 caras y 490 cruces. Si haces cálculos matemáticos esto significa que hay una probabilidad de 0.75 de obtener más caras que cruces. Si sigues aumentando el número de tiradas, esta probabilidad aumenta (con 10000 tiradas 0.95). Conforme más lanzas la moneda, el ratio de las tiradas se acercar cada vez más a la probabilidad de que el 51% sea cara.

Ahora ponte en el caso de tengamos 1000 clasificadores con un pobre desempeño, una precisión individual del 51% (apenas mejor que una estimación aleatoria). Si realizamos *hard voting*, tendremos una precisión del 75%. Entonces... ya tenemos todo solucionado ¿no? Aumentamos el número de predictores hasta que nos acercarnos al 100%. Pues lamentablemente no todo es tan sencillo. Esto **solo ocurrirá si los predictores fueran completamente independientes unos de otros**, lo cual obviamente no es el caso ya que se entrenan con los mismos datos. Por este motivo los algoritmos de ensamblaje funcionan mejor cuanto más independientes son sus predictores entre sí.

EJEMPLO 36 Implementar un sistema *Voting* con un Árbol de Decisión, Regresión Logística, KNN y Bosque Aleatorio:

```
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=500, noise=0.25, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

#----- modelo HARD-VOTING -----
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
arbol_cla = DecisionTreeClassifier()
bosque_cla = RandomForestClassifier()
relog_cla = LogisticRegression()
knn_cla = KNeighborsClassifier()
voting_cla = VotingClassifier(
    estimators=[('rnd', bosque_cla), ('dt', arbol_cla), ('knn', knn_cla),
                ('log', relog_cla)],
    voting='hard', n_jobs=-1)
#----- Medir eficiencia individual y agregada -----
from sklearn.metrics import accuracy_score
for cla in (arbol_cla, bosque_cla, knn_cla, relog_cla, voting_cla):
    cla.fit(X_train, y_train)
    y_pred = cla.predict(X_test)
    print(cla.__class__.__name__, accuracy_score(y_test, y_pred))
```

Hay que inicializar los modelos y pasar a ***VotingClassifier()*** una lista con tuplas que contienen con el nombre del clasificador y el clasificador en sí.

`[(nombre_clasificador1, objeto_1), (nombre_clasificador2, objeto_2), ...]`

Elegimos el modo de **voting**, aunque por defecto ya está en **hard**. Por último **n_jobs** es el número de procesos que hará en paralelo, con -1 indicas todos, lo que hará que vaya más rápido (como estas entrenando varios modelos a la vez, si por ejemplo tienes **n_jobs=4**, estarías usando 4 núcleos de tu ordenador si los tienes y entrenaría los cuatro modelos a la vez). Aunque no por mucho, el predictor ensamblado es mejor que el resto.

En el caso de que todos los predictores pudiesen estimar probabilidades, es decir, tuviesen método **predict_proba()**, podríamos indicar a *Scikit-Learn* que elija la clase con la probabilidad media mayor. Este método de *Voting* se llama **soft voting**, y suele dar mejores resultados gracias a que tienen más peso los votos de un clasificador si ese clasificador está seguro de lo que vota.

EJEMPLO 37: Implementar un sistema *Soft-Voting*:

```
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=500, noise=0.25, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
#----- modelo SOFT-VOTING -----
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.ensemble import VotingClassifier
relog_cla = LogisticRegression(solver="lbfgs", random_state=42)
bosque_cla = RandomForestClassifier(n_estimators=100, random_state=42)
svm_cla = SVC(gamma="scale", probability=True, random_state=42)
voting_cla = VotingClassifier(
    estimators=[('lr', relog_cla), ('rf', bosque_cla), ('svc', svm_cla)],
    voting='soft')
#----- Medir eficiencia individual y agregada
from sklearn.model_selection import cross_val_score
for cla in (relog_cla, bosque_cla, svm_cla, voting_cla):
    resultado = cross_val_score(cla, X, y, scoring='accuracy', cv=10)
    print(cla.__class__.__name__, resultado.mean())
```

4.2. BAGGING/BOOTSTRAP Y PASTING.

Otro enfoque para crear métodos de ensamblaje puede ser usar el mismo tipo de predictor pero entrenar varios utilizando muestras aleatorias de datos de entrenamiento distintas. Si en las muestras hay reemplazo (es decir, se puede usar el mismo dato para más de un predictor) estaremos usando **Bagging o Bootstrap**. Si no hay reemplazo, será **Pasting**.

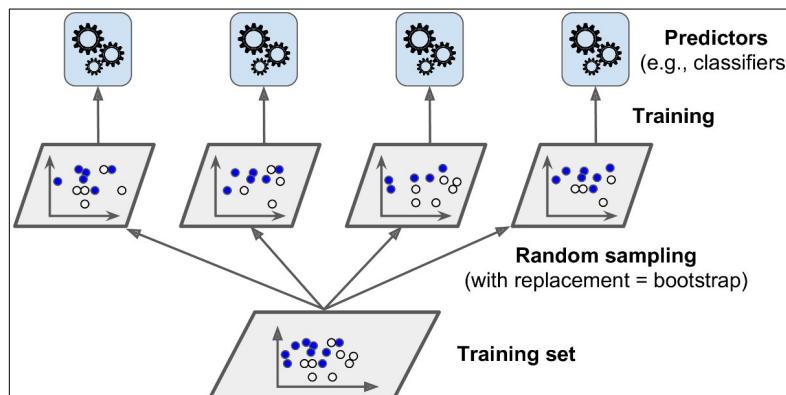


Figura 43. Modelo de ensamblaje Bagging y Pasting.

Cuando todos los predictores están entrenados, el ensamblado puede realizar una predicción sencillamente agrupando todas las predicciones de nuestros clasificadores (o por votación o por media de probabilidades) o con la media de los regresores.

Cada predictor individual tiene un sesgo (*bias*) mayor, ya que están siendo entrenados en una parte menor de los datos. Sin embargo, cuando los agregamos reducimos el sesgo y la varianza. Además, los distintos predictores que forman el ensamblado pueden ser entrenados paralelamente en distintos cores de la CPU (igual que con *Voting*, indicando *n_jobs*), lo que reduce considerablemente el tiempo de entrenamiento y provoca que escalen muy bien.

EJEMPLO 38: Agregar 600 Árboles de Decisión, cada uno de ellos entrenados en 150 muestras de los datos usando un *BaggingClassifier*. En el caso de que queramos realizar *Pasting*, solo tendremos que poner el hiperparámetro *bootstrap* (que es un término estadístico que significa reemplazo) como *False*:

```
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=500, noise=0.25, random_state=42)
# Un árbol aislado
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import cross_val_score
resultado = cross_val_score(DecisionTreeClassifier(), X, y, scoring='accuracy', cv=10)
print("Árbol de decisión aislado: ", resultado.mean())
# Bootstrap
from sklearn.ensemble import BaggingClassifier
bag_cla = BaggingClassifier(DecisionTreeClassifier(), n_estimators=600,
                           max_samples=150, bootstrap=True, n_jobs=-1)
resultado = cross_val_score(bag_cla, X, y, scoring='accuracy', cv=10)
print("Bootstrap:", resultado.mean())
```

En general *Bagging* obtiene mejores resultados, sin embargo, si tienes tiempo y potencia de computo, no está de más que uses también *Pasting* y compares ambos.

EVALUACIÓN OUT-OF-BAG

Cuando entrenamos cada uno de los clasificadores usando *Bagging*, a cada uno de ellos solo los entrenamos con una parte de los datos, un 63%. El otro 37% de los datos que Ne usan se les llama *instancias out-of-bag (oob)*. Estas 37% de muestras *oob* no son las mismas para cada predictor.

Gracias a que los predictores no entran con las muestras *oob*, las podemos utilizar como datos de *test* y así aprovecharíamos más nuestros datos. Posteriormente solo tendremos que hacer una media de todas las puntuaciones de *oob* y tendríamos nuestra precisión final, sin necesidad de separar un *set* de validación.

Para implementarlo en *Scikit-Learn* solamente tenemos que indicar *oob_score=True* cuando creamos el modelo *Bagging*. Por último, para obtener los resultados solo hay que acceder al valor medio a través de *oob_score_*. Veamos un ejemplo:

EJEMPLO 39: Añade este código al del ejemplo 38:

```
bag_cla2 = BaggingClassifier(DecisionTreeClassifier(), n_estimators=600,
                           max_samples=150, bootstrap=True, n_jobs=-1,
                           oob_score=True)
bag_cla2.fit(X, y)
print("Bootstrap out-of-bag: ", bag_cla2.oob_score_)
```

Si lo comparo con el resultado anterior realizando *cross validation*, me ha salido tan solo una diferencia de 0.002 entre ambos resultados, muy similar.

La función de decisión de cada clasificador está disponible usando la variable **`oob_decision_function_`**. En este caso como el estimador tiene el método **`predict_proba()`** devuelve las probabilidades de cada clase. Por ejemplo en la salida estima que el primer clasificador da un 68.25% de probabilidad de pertenecer a la clase positiva:

```
print( bag_cla.oob_decision_function_ )
```

4.3. RANDOM FOREST.

Los Bosques Aleatorios son un método de ensamblaje en el que se agregan predictores de Árboles de Decisión usando *bagging* (o a veces *pasting*). En vez de utilizar el objeto **`BaggingClassifier`** y pasar Árboles de Decisión, podemos usar **`RandomForestClassifier`** (o **`RandomForestRegressor`** en el caso de regresión) que es más sencillo y más optimizado.

Con algunas excepciones, los Bosques Aleatorios cuentan con todos los parámetros de los Árboles de Decisión y de los algoritmos de *bagging*. Otra característica de este modelo es que contiene una mayor aleatoriedad, ya que **al hacer la división de los datos cuando está creando árboles, en vez de buscar la mejor variable por la que dividir el nodo**, lo hace en un subespacio aleatorio de las variables, es decir, no tiene en cuenta todas las variables. Esto provoca que se generan árboles con mayor diversidad, lo que a la larga resulta en un mejor modelo.

EJEMPLO 40: Crear un modelo para ver cómo se comporta con los datos de Iris.

```
from sklearn.datasets import load_iris
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score
iris = load_iris()
X = iris.data
y = iris.target
bosque_cla = RandomForestClassifier(n_estimators=600, max_leaf_nodes=16, n_jobs=-1)
resultado = cross_val_score(bosque_cla, X, y, scoring='accuracy', cv=10)
print("Random Forest:", resultado.mean())
```

IMPORTANCIA DE LAS VARIABLES

A pesar de que tanto los Árboles de Decisión como los Bosques Aleatorios son modelos no lineales, son muy intuitivos y pueden darnos información muy valiosa sobre las decisiones que toman.

Una ventaja de los Bosques Aleatorios es que miden la importancia de las variables de los datos que utilizan. Para ello, observan como mejora la pureza de los nodos que utilizan las variables y calculan la media ponderada en función del número de muestras que tenían. Es decir, **ven si la pureza de un árbol mejora o no al usar la variable X para hacer la división**.

Nuevamente, *Scikit-Learn* registra estos datos automáticamente y después de entrenar **los escala para que la suma de todos ellos sea 1**. Podemos acceder a ellos mediante el atributo **`feature_importance_`**.

EJEMPLO 41: Añade al ejemplo 41 este código para acceder a la importancia de cada característica:

```
bosque_cla.fit(iris["data"], iris["target"])
for nombre, score in zip(iris['feature_names'], bosque_cla.feature_importances_):
    print(nombre, score)
```

Observamos que la importancia del ancho y longitud del pétalo son las variables más importantes. Esto significa que posiblemente si usamos solo estas dos variables el modelo será más eficiente e incluso generalizará mejor. Añadimos también estas sentencias para probarlo:

```
# Probar a usar solamente dos características importantes
X = iris.data[:, 2:] # ancho y longitud de Petalis
y = iris.target
bosque_cla = RandomForestClassifier(n_estimators=600, max_leaf_nodes=16, n_jobs=-1)
resultado = cross_val_score(bosque_cla, X, y, scoring='accuracy', cv=10)
print("Eliminando características: ", resultado.mean())
```

Vemos que nuestra predicción ha mejorado en 0.006666. Puede no parecer mucho, pero hemos reducido en 16.66% el error. Este es uno de los usos que pueden tener los Bosques Aleatorios. Otro puede ser el ayudar a tener **Inteligencia Artificial explicable**. Es decir, entender por qué un modelo hace buenas predicciones y en qué se basa.

4.4. BOOSTING.

Es cualquier tipo de método de ensamblaje que convierte varios predictores débiles (hacen predicciones poco mejores que aleatorias ~51%) en predictores fuertes (hacen buenas predicciones). La forma de ensamblar estos modelos normalmente consiste en **entrenarlos secuencialmente, uno detrás de otro, de manera que cada uno intente corregir a su predecesor**. Hay bastantes técnicas para hacer esto, pero las que más se usan son *AdaBoost* y *Gradient Boosting*.

ADABOOST

Una forma de que un predictor pueda corregir a su predecesor es fijarse más en las muestras en las que el modelo anterior tiene *underfitting*. De esta manera se van centrando en los casos más difíciles. Es igual que en el colegio, cuando hacías muchos tipos de problemas, el profesor te hacía practicar en los que tenías problemas y solías cometer errores.

AdaBoost hace esto de manera muy sencilla. Una vez ha entrenado un modelo, le da un peso mayor a las instancias mal clasificadas. Así, el siguiente predictor entrenará más tiempo con ellas.

Una desventaja de este modelo es que no se puede paralelizar ya que no puedes entrenar el siguiente modelo hasta que el anterior haya terminado, lo que provoca que no escale tan bien y tenga un alto coste computacional.

Scikit-Learn utiliza una variante de *AdaBoost* llamada **SAMME**. Cuando solo hay que clasificar dos clases, ambos algoritmos son equivalentes. Además, en caso de que los predictores que se usen puedan predecir probabilidades, se puede usar la variante **SAMME.R**, que se basa en la probabilidad de las clases, lo que normalmente da mejores resultados.

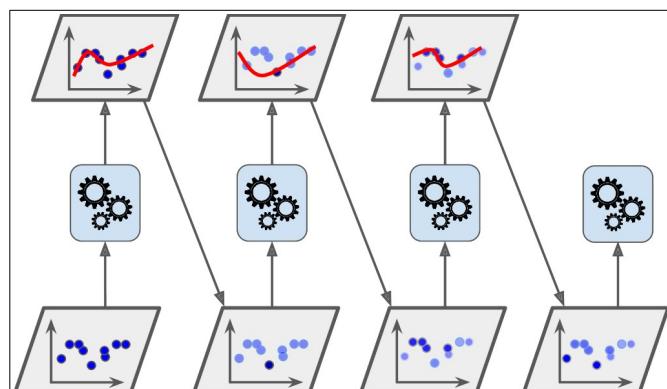


Figura 44. Entrenamiento secuencial de AdaBoost actualizando pesos de ejemplos.

Para implementar este modelo solo hay que usar la clase ***AdaBoostClassifier*** para clasificar o ***AdaBoostRegressor*** para realizar regresiones e indicar el algoritmo que usaremos en el parámetro ***algorithm***:

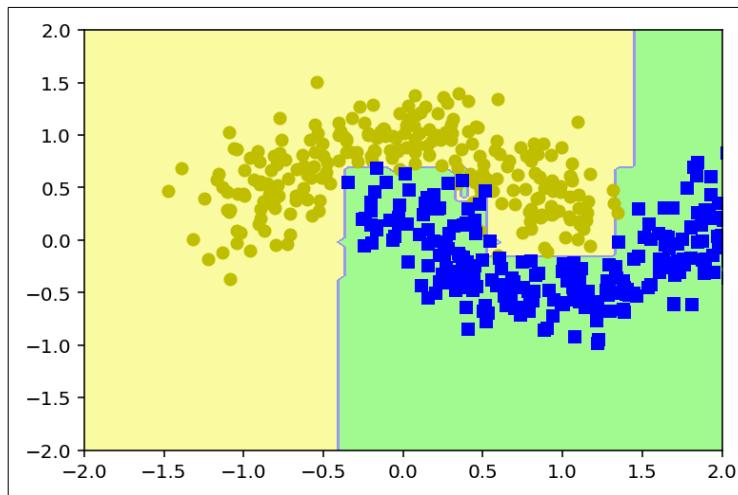


Figura 45. Fronteras de decisión de AdaBoost de 200 árboles de decisión con learning-rate de 0.5.

EJEMPLO 42: Añade al ejemplo anterior.

```
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=500, noise = 0.2, random_state=42)

from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
ada_cla = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=2), n_estimators=200,
    algorithm='SAMME.R', learning_rate=0.5)
ada_cla.fit(X, y)

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

def plot_fronteras_decision(arbol, X, y, ejes=[0, 7.5, 0, 3]):
    x1s = np.linspace(ejes[0], ejes[1], 100)
    x2s = np.linspace(ejes[2], ejes[3], 100)
    x1, x2 = np.meshgrid(x1s, x2s)
    X_nuevo = np.c_[x1.ravel(), x2.ravel()]
    y_pred = arbol.predict(X_nuevo).reshape(x1.shape)
    plt.plot(X[:, 0][y==0], X[:, 1][y==0], "yo", label="clase1")
    plt.plot(X[:, 0][y==1], X[:, 1][y==1], "bs", label="clase2")
    plt.axis(ejes)
    plt.contourf(x1, x2, y_pred, cmap=ListedColormap(['#fafafa0', '#9898ef', '#a0fa90']))


plot_fronteras_decision(ada_cla, X, y, ejes=[-2,2,-2,2])
```

Vamos a describir el algoritmo *AdaBoost* de manera más detallada. Se asigna un peso $w^{(i)}$ a cada una de las m instancias de los datos de entrenamiento. Inicialmente se les da un valor de $1/m$.

Un primer predictor es entrenado y se calcula su ratio de error r_1 sobre los datos de entrenamiento utilizando la ecuación 4.4.1:

donde $\hat{y}_j^{(i)}$ es el j-ésimo predictor de la predicción para la i-ésima instancia o ejemplo de datos.

$$r_j = \frac{\sum_{i=1}^m w^{(i)}}{\sum_{\substack{i=1 \\ \hat{y}_j^{(i)} \neq y^{(i)}}}^m w^{(i)}} \quad [\text{Ecuación 4.4.1}]$$

El peso del predictor α_j se calcula usando la ecuación 4.4.2, donde η es el hiperparámetro ratio de aprendizaje (por defecto a 1). El predictor más exacto es el que tendrá un peso más alto. Si sus predicciones se parecen a las de un predictor aleatorio entonces su peso estará cerca de cero. Incluso, si su desempeño es peor que el de un predictor aleatorio (falla más que el aleatorio) entonces su peso será negativo.

$$\alpha_j = \eta \log \frac{1 - r_j}{r_j} \quad [\text{Ecuación 4.4.2}]$$

A continuación se actualizan los pesos de las instancias usando la ecuación 4.4.3 donde las instancias mal clasificadas se potencian para que se centre en ellas el siguiente clasificador.

$$\begin{aligned} & \text{for } i = 1, 2, \dots, m \\ w^{(i)} & \leftarrow \begin{cases} w^{(i)} & \text{if } \hat{y}_j^{(i)} = y^{(i)} \\ w^{(i)} \exp(\alpha_j) & \text{if } \hat{y}_j^{(i)} \neq y^{(i)} \end{cases} \quad [\text{Ecuación 4.4.3}] \end{aligned}$$

Luego todos los pesos de las instancias se normalizan (se dividen por la suma de todos: $\sum_{i=1}^m w^{(i)}$)

Por último, se entrena un nuevo predictor usando los pesos que se acaban de actualizar. El mismo proceso se repite con este predictor que actualiza los pesos para el siguiente y así sucesivamente hasta que se alcanza la cantidad de predictores indicada o se encuentra un predictor perfecto.

Para hacer las predicciones, *AdaBoost* simplemente calcula las predicciones de todos los predictores y los pondera usando los pesos α_j de cada predictor. Se predice la clase que recibe la mayoría de votos tal y como describe la siguiente ecuación donde N es el número de predictores:

$$\hat{y}(\mathbf{x}) = \operatorname{argmax}_k \sum_{j=1}^N \alpha_j \quad \hat{y}_j(\mathbf{x}) = k$$

GRADIENT BOOSTING

Es similar a *AdaBoost* porque entrena modelos secuencialmente centrándose en el error del modelo anterior. Sin embargo, en vez de darle más peso a las instancias en las que falla el predictor anterior, entrena el modelo sucesivo con los residuos (distancias entre el punto predicho y el real).

Supongamos un Árbol de Decisión de regresión. Para cada instancia del entrenamiento predice un valor. Entonces se guarda el residuo, la diferencia entre el valor predicho y el real y al siguiente predictor se le entrenará con este dato. Funciona de manera similar para problemas de clasificación. En *Scikit-Learn* usamos las clases ***GradientBoostingRegressor*** y ***GradientBoostingClassifier***.

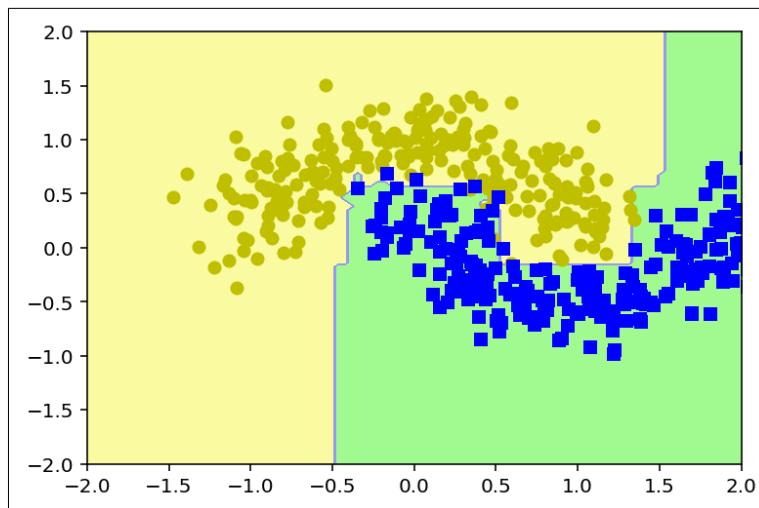


Figura 46. Fronteras de decisión de GradientBoostingClassifier de 100 árboles de decisión y learning-rate 0.5

EJEMPLO 43: Vamos a usar *GradientBoostingClassifier*.

```
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=500, noise = 0.2, random_state=42)
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

from sklearn.ensemble import GradientBoostingClassifier
gb_cla = GradientBoostingClassifier(max_depth=3, n_estimators=100, learning_rate=0.5)
gb_cla.fit(X_train, y_train)

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

def plot_fronteras_decision(arbol, X, y, ejes=[0, 7.5, 0, 3]):
    x1s = np.linspace(ejes[0], ejes[1], 100)
    x2s = np.linspace(ejes[2], ejes[3], 100)
    x1, x2 = np.meshgrid(x1s, x2s)
    X_nuevo = np.c_[x1.ravel(), x2.ravel()]
    y_pred = arbol.predict(X_nuevo).reshape(x1.shape)
    plt.plot(X[:, 0][y==0], X[:, 1][y==0], "yo", label="clase1")
    plt.plot(X[:, 0][y==1], X[:, 1][y==1], "bs", label="clase2")
    plt.axis(ejes)
    plt.contourf(x1, x2, y_pred, cmap=ListedColormap(['#fafaa0', '#9898ef', '#a0fa90']))
```

plot_fronteras_decision(gb_cla, X, y, ejes=[-2,2,-2,2])

El *learning_rate* escala la contribución de cada árbol. Si lo pones a un valor bajo, como 0.1 necesitarás más predictores para entrenar con los datos.

Alternativamente podemos hacer uso de la librería **XGBoost** (*Extreme Gradient Boosting*) de la que hablamos en la unidad anterior. Es una implementación optimizada de este método de ensamblaje.

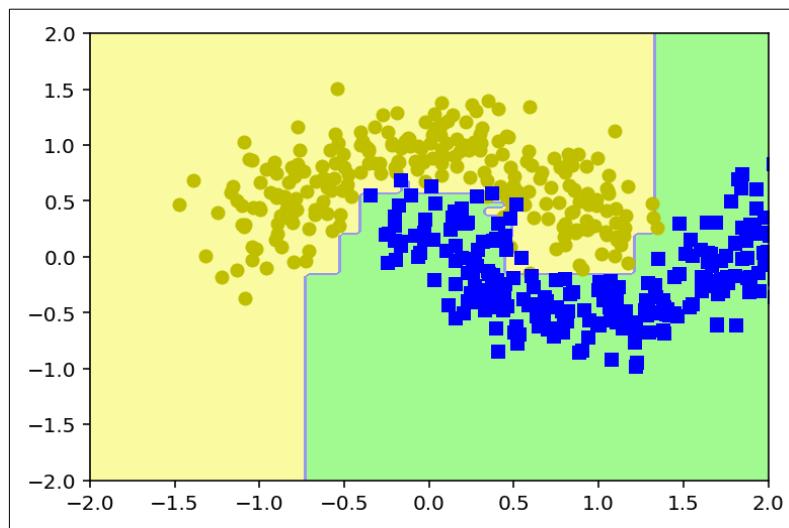


Figura 47. Fronteras de decisión de XGBoost con valores por defecto.

EJEMPLO 44: Vamos a usar *GradientBoostingClassifier*.

```
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=500, noise = 0.2, random_state=42)
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

from xgboost.sklearn import XGBClassifier # Quizás debas instalar: pip install xgboost
xgb_cla = XGBClassifier()
xgb_cla.fit(X_train, y_train)

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

def plot_fronteras_decision(arbol, X, y, ejes=[0, 7.5, 0, 3]):
    x1s = np.linspace(ejes[0], ejes[1], 100)
    x2s = np.linspace(ejes[2], ejes[3], 100)
    x1, x2 = np.meshgrid(x1s, x2s)
    X_nuevo = np.c_[x1.ravel(), x2.ravel()]
    y_pred = arbol.predict(X_nuevo).reshape(x1.shape)
    plt.plot(X[:, 0][y==0], X[:, 1][y==0], "yo", label="clase1")
    plt.plot(X[:, 0][y==1], X[:, 1][y==1], "bs", label="clase2")
    plt.axis(ejes)
    plt.contourf(x1, x2, y_pred, cmap=ListedColormap(['#fafaa0', '#9898ef', '#a0fa90']))
```

plot_fronteras_decision(gb_cla, X, y, ejes=[-2,2,-2,2])

5. MODELOS OCULTOS DE MARKOV (HMM).

La intuición muchas veces nos dice lo que tenemos que hacer: por ejemplo si nos encontramos ciertas palabras sabemos en qué partes de una conversación deben aparecer o si un usuario visita la página web de firmar sabemos que hay una alta probabilidad de que se convierta en un cliente. Pero ¿Cómo modelar esa intuición?

Porque el mundo está lleno de fenómenos para los que vemos el resultado final pero no tenemos la posibilidad de observar los mecanismos internos que han generado estos resultados. Un ejemplo es predecir la climatología, saber si mañana llueve o hará sol basándonos en las observaciones del pasado y en las probabilidades de las diferentes fenómenos. Aunque estos fenómenos estén gobernados por factores que no podemos observar, los **modelos ocultos de Markov (HMM)** pueden modelar estos fenómenos como sistemas probabilísticos.

Un **HMM** es un modelo estadístico que funciona como una secuencia de problemas etiquetados. Este tipo de problemas describen como funcionan o evolucionan los eventos que somos capaces de observar de un sistema, aunque ellos mismos dependen de factores internos que no podemos observar directamente y los denominamos ocultos.

Un *HMM* tiene dos procesos estocásticos¹⁵ distintos, lo que significa que son procesos que podemos definir como secuencias de variables aleatorias que dependen de eventos aleatorios: unos procesos invisibles y unos procesos observables.

- **Los procesos invisibles:** son una *cadena de Markov*, como están encadenados juntos al atravesarlos generan una salida. Es un proceso probabilístico porque todos los parámetros de la *cadena de Markov* incluidos los valores de cada secuencia son probabilidades.
- **Los procesos observables:** Los *HMM* describen la evolución de los eventos observables, que ellos mismos dependen de factores internos que no podemos observar directamente porque están ocultos.

Como cualquier otra *cadena de Markov*, para saber el estado al que vas a ir a continuación, lo único importante es el estado de la cadena donde estás actualmente. Ninguno de los estados previos (la historia de estados por los que has atravesado) aporta información de cuál será el estado siguiente. Este tipo de memoria a corto plazo es una de las características de los *HMM* se denomina la **asunción de Markov**, indicando que la probabilidad de alcanzar el siguiente estado solamente depende del estado actual:

$$P(q_i = a \mid q_1 \dots q_{i-1}) = P(\overbrace{q_i = a}^{\text{next state}} \mid \overbrace{q_1 \dots q_{i-1}}^{\text{history of states traversed}})$$

current state

La otra característica clave de un *HMM*, es que también asume que cada observación solamente depende del estado en que se ha producido y es completamente independiente de cualquier otro estado de la cadena.

Con esta aproximación las *HMM* pueden modelizar muchos tipos de problemas (evolución de las existencias en un almacén, procesamiento de texto, procesamiento de audio como por ejemplo convertir un audio en texto, etc) y pueden calcular muchos tipos de preguntas sobre ese fenómeno, por ejemplo:

- **Likelihood o Scoring**, como calcular la probabilidad de observar una secuencia.
- **Decodificar** la mejor secuencia de estados generada por una observación concreta.
- **Aprender** los parámetros del *HMM* usando una observación que atraviesa un conjunto de estados.

Para aplicar el modelo a un problema práctico sencillo imagina que tienes un perro al que quieres educar y lo llevas a un entrenador. Durante las sesiones de entrenamiento quiere enseñarle unos cuantos trucos y puede observar y medir su comportamiento. Después de algunas sesiones quiere evaluar si el perro necesita más entrenamiento o ya ha aprendido lo suficiente. Pero el entrenador solamente observa el resultado, aunque hay muchos factores que afectan a lo que observa: si el

¹⁵ Estocástico es un sinónimo de aleatorio.

perro está cansado (*tired*) o feliz (*happy*), si no le gusta el entrenador o el resto de perros que están, etc.

Durante los entrenamientos hemos recolectado información y vamos a usar un *HMM* para predecir como se comportará tu perro en un examen. Para definir el *HMM* necesitamos:

- Estados ocultos
- Matriz de transiciones
- Secuencia de observaciones
- Matriz de probabilidades de observaciones.
- Distribución inicial de probabilidades.

Los estados ocultos son aquellos factores no observables que influyen en la secuencia de observaciones. Solo vamos a considerar si el perro está cansado o feliz.

$$Q = Q_1, Q_2, \dots, Q_n \longrightarrow Q = \text{Tired, Happy}$$

Matriz de transiciones (A) o de probabilidades (P): necesitamos las probabilidades de pasar de un estado a otro. Esta información es la que contiene esta matriz. La suma de los valores de cada fila de la matriz debe dar 1 y representa la probabilidad de pasar de un estado a otro o de permanecer en cada estado.

$$A = a_{11}, a_{ij}, \dots, a_{NM} \quad \sum_{j=1}^N a_{ij} = 1 \forall i$$

Sum of state transitions = 1

	Tired	Happy
Tired	0.4	0.6
Happy	0.2	0.8

Secuencia de observaciones (símbolos): cada observación representa el resultado de atravesar la *cadena de Markov*. Cada observación se puede extraer desde un vocabulario específico del problema que estamos modelando. Por ejemplo en el caso del entrenamiento observamos en cada ejercicio 3 palabras: "fallo", "Ok", "perfecto".

$$V = V_1, V_2, \dots, V_n \longrightarrow \text{Vocabulario } V = \{\text{fallo, Ok, Perfecto}\}$$

Matriz de probabilidades de emisión (E): En el caso del examen al perro observarás la puntuación que obtiene tras intentar realizar cada ejercicio que podrá ser uno de estos resultados *{Fallo, OK, Perfecto}*. Estos son todos los posibles términos del vocabulario (posibles resultados) del problema. La matriz de probabilidades de observaciones nos dice la probabilidad de que una observación (resultado) se haya generado desde un estado concreto. Las filas son estados y las columnas símbolos del vocabulario (son probabilidades condicionales):

	Fail	OK	Perfect
Tired	0.3	0.5	0.2
Happy	0.1	0.5	0.4

Distribución inicial de probabilidades (π): es la probabilidad de que cada *cadena de Markov* se inicie en cada estado oculto. Puede haber estados ocultos en los que nunca se inicie una observación, y en estos casos su probabilidad inicial será cero. Y al igual que ocurría en la matriz de transiciones, su suma debe dar 1.

$$\pi = \pi_1, \pi_2, \dots, \pi_n \longrightarrow \pi = [\text{Tired, Happy}] = [0.1, 0.9]$$

La distribución inicial de probabilidades (π), la matriz de transiciones y la matriz de probabilidades de observaciones son los hiperparámetros del modelo HMM. Estas son las probabilidades que:

- Tu intuición y tus observaciones te dan.
- Si tienes una secuencia de observaciones y estados ocultos, intentas configurar el modelo HMM concreto que podría haberlas generado.
- Incluso podrías intentar averiguar qué estados ocultos podrían generarlas.

Es decir, las tareas en las que podemos utilizar un HMM y los algoritmos de aprendizaje que se utilizan para hacerlo son:

- Aprendizaje:** dada una lista de secuencias de símbolos del vocabulario que se han observado queremos **aprender** los parámetros (π , P, E) de un **HMM**. En ese caso podemos utilizar el **algoritmo de Baum-Welch**.
- Evaluación:** una vez que el modelo está configurado ¿Cuál es la verosimilitud de observar una determinada secuencia? Para ello usamos el algoritmo **algoritmo forward**.
- Decodificación:** una vez que el modelo está ajustado, ¿Cuál es la secuencia de estados más **probable** para generar de una determinada secuencia de símbolos observada? Usaremos el **algoritmo de Viterbi** o el **algoritmo map (smoothing o forward-backward)**.

Colocando juntas todas estas piezas, un modelo HMM puede representar el comportamiento del perro durante el examen y tendrá un aspecto similar a este:

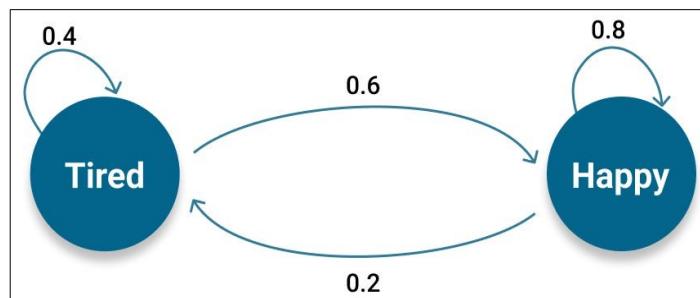


Figura 48. Máquina de estados asociada a un modelo HMM.

MODELOS HMM EN HMMLEARN

Para usar modelos de Markov podemos instalar: `pip install hmmLearn` y luego importar: `from hmmLearn import hmm`.

Algunos de los modelos disponibles están en esta tabla:

MODELOS	Descripción
<code>CategoricalHMM</code>	Emisiones son símbolos discretos o categóricos
<code>GaussianHMM</code>	Emisiones son valores continuos que siguen distribución Gausiana
<code>GMMHMM</code>	Emisiones siguen distribución Gausiana Mixture
<code>MultinomialHMM</code>	Emisiones siguen distribución multinomial

Puedes instanciar un modelo **HMM** pasando sus parámetros en el constructor y luego usarlo para generar muestras aleatorias con el método `sample()`. Una vez instanciado el modelo, antes de entrenarlo también puedes dar valor a alguno de sus componentes como la matriz inicial de estados (`startprob` → "s") la matriz de probabilidades de transición entre estados (`transmat` →

"**t**") y la matriz de probabilidades de emisiones (**emissionprob** → "e"). El parámetro puede modificarse durante el entrenamiento si está incluido en **param="ste"** y los que pueden modificarse antes en **init_param="ste"**. De esta manera si quieras dejar fijo algún parámetro y no quieras que el entrenamiento lo modifique, puedes quitar su símbolo y permanecerá intacto.

n_states es el número de estados y **n_features** es la cantidad de símbolos que puede emitir y se infiere en el entrenamiento si no se indica. **Algorithm** es el algoritmo de decodificación que se utiliza.

EJEMPLO 45: configurar un modelo HMM conociendo las probabilidad iniciales, 3 estados,

```
import numpy as np
from hmmlearn import hmm
np.random.seed(449)

modelo = hmm.GaussianHMM(n_components=3, covariance_type="full")
modelo.startprob_ = np.array([0.6, 0.3, 0.1])
modelo.transmat_ = np.array([[0.7, 0.2, 0.1],
                           [0.3, 0.5, 0.2],
                           [0.3, 0.3, 0.4]])
modelo.means_ = np.array([[0.0, 0.0], [3.0, -3.0], [5.0, 10.0]])
modelo.covars_ = np.tile(np.identity(2), (3, 1, 1))
_, Z = modelo.sample(n_samples=5)
X, Z = modelo.sample(n_samples=5, currstate=Z[-1])
print(X)
print(Z)
```

ENTRENAR HMM PARA INFERIR LOS ESTADOS OCULTOS

Además de configurar manualmente un HMM, puedes entrenarlo mostrándole secuencias de símbolos (observaciones) que le pasas al método **fit()**. Debes pasarle una matriz que contenga las secuencias de observaciones concatenadas y la longitudes de estas secuencias.

La entrada puede ser una única secuencia de valores observados:

```
modelo = hmm.GaussianHMM(n_components=3, covariance_type="full", n_iter=100)
modelo.fit(X)
Z2 = modelo.predict(X)
```

Todas las muestras usan una única secuencia observada. En otro caso, cuando hay varias secuencias hay que indicar su tamaño. Ejemplo:

```
# Secuencias 1D:
X1 = [[0.5], [1.0], [-1.0], [0.42], [0.24]]
X2 = [[2.4], [4.2], [0.5], [-0.24]]
```

Para enviar ambas secuencias a los métodos **fit()** y **predict()**, primero se concatenan en un único array y luego se calcula un array con las longitudes de cada una:

```
X = np.concatenate([X1, X2])
longitudes = [len(X1), len(X2)]
hmm.GaussianHMM(n_components=3).fit(X, longitudes)
```

Como el algoritmo EM está basado en un método optimizado de gradiente puedes tener la mala suerte de quedar atrapado en un mínimo local. Por tanto deberías ejecutar varias veces el entrenamiento con varias inicializaciones diferentes y seleccionar el modelo que mejor *score* obtenga. Los *scores* de un modelo los calcula el método **score()**.

Los estados ocultos óptimos calculados pueden obtenerse con el método `predict()` al que puedes indicar un algoritmo de decodificación como "viterbi" o "map".

MONITORIZAR LA CONVERGENCIA

El número de iteraciones está limitado por el hiperparámetro `n_iter`. El entrenamiento puede detenerse al alcanzarlo o bien si el score es más bajo que el valor indicado en el hiperparámetro `tol`. Ten en cuenta que según los datos, el algoritmo puede no alcanzar una solución en la cantidad de pasos indicada. Puedes usar el atributo `monitor_` para diagnosticar la convergencia a una solución.

```
print(modelo.monitor_)
# Posible salida:
# ConvergenceMonitor(
#     history=[...],
#     iter=...,
#     n_iter=100,
#     tol=0.01,
#     verbose=False,
# )
print(modelo.monitor_.converged)    # True o False
```

Realizar ejercicios 18 y 19 de la relación de problemas.

5.1. ALGORITMOS USADOS EN HMM.

El examen del perro consiste en pedirle que intente 3 ejercicios y aprueba solamente si no obtiene *Fail* en dos de ellos. Al final del día, si necesita más entrenamiento se repite el proceso. Podemos estar interesados en cómo se ha sentido el perro durante el examen? Imagina que al examinarse aprueba con una secuencia de resultados de: *OK – Fail – Perfect* exactamente en este orden, ¿Qué estado emocional tenía el perro? ¿Estaba cansado, o enfadado o una mezcla de todo? Este tipo de problemas cae dentro de la categoría que se denomina **problemas de decodificación**. HMM puede aplicarse a este tipo de problemas. En este caso estamos interesados en saber qué secuencia de estados es la más probable que pueda generar una secuencia de resultados concreta, en el ejemplo *OK – Fail – Perfect*.

Estos problemas de decodificación puede resolverlos el algoritmo de Viterbi. Pero para comprender este algoritmo, antes debemos conocer otro algoritmo que usa el *algoritmo de Viterbi* y que se denomina *el algoritmo Forward* (algoritmo de avance).

ALGORITMO FORWARD

Si modelas un problema con una *cadena regular de Markov* y quieres calcular el *likelihood* (las posibilidades) que influyen en observar una secuencia de resultados (*OK, Fail, Perfect* en el ejemplo) debes atravesar la cadena aterrizando en cada estado específico que genera el resultado deseado. En cada paso, anotas la probabilidad condicional de observar el resultado actual dado que has observado el resultado anterior y multiplicas esa probabilidad por la probabilidad de transición de pasar de un estado a otro.

La diferencia con un HMM es que en una *cadena regular de Markov* todos los estados se conocen y son observables. Pero esto no siempre ocurre en el HMM porque desconoces la secuencia de posibles estados se han atravesado y han generado esa cadena de resultados y si desconoces los estados también desconoces sus probabilidades de transición.

En este momento podrías pensar: *Bueno pues simplemente atraveso todas las posibles rutas y quizás calcule una regla que me de la posibilidad de elegir entre rutas equivalentes*. La definición matemática de esta aproximación se parece a esta fórmula:

$$\underbrace{P(O)}_{\substack{\text{observed sequence} \\ \text{of scores}}} = \sum_Q \overline{P(O, Q)} = \sum_Q P(O|Q) \times P(Q)$$

joint probability of observing
a particular hidden state
sequence

$$P(O|Q) = \prod_{i=1}^M P(o_i|q_i)$$

For M hidden states

hidden state sequence

Debes calcular la probabilidad de observar la secuencia *OK, Fail, Perfect* para cada combinación única de estados ocultos que puedan generar esa secuencia. Cuando tienes una pequeña cantidad de estados ocultos y de secuencias de salida generada, es posible calcularlas en un tiempo razonable.

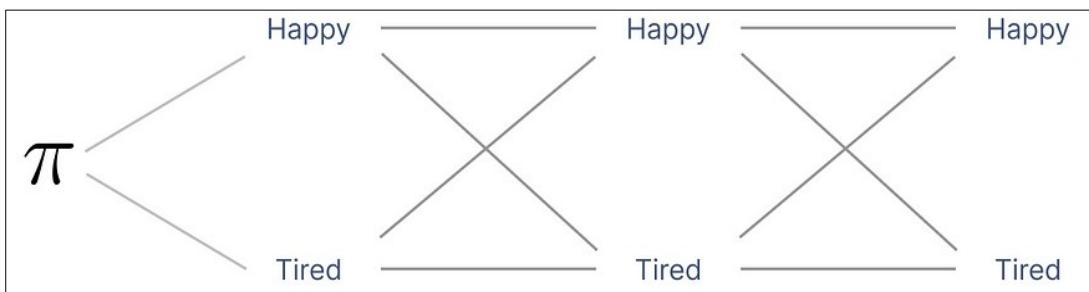


Figura 49. Rutas posibles en el HMM de ejemplo.

Afortunadamente, el HMM que acabamos de definir es bastante simple con 3 salidas observadas y 2 estados ocultos. Para una secuencia observada de longitud L , en un HMM con M estados ocultos tienes M^L posibles estados que estudiar, en el ejemplo significa 2^3 , es decir 8 posibles rutas para la secuencia *OK – Fail – Perfect*, lo que significa que la complejidad computacional del cálculo es de orden $O(LM^L)$. Cuando la complejidad del modelo aumenta, el número de rutas que es necesario revisar crece exponencialmente.

$$\alpha(n, i) = P(x, \dots, x_n, y_n = i | \Theta)$$

nth outcome in the
sequence of observed
outcomes x

Current Hidden State

Hidden State i in the
sequence of hidden
states y

HMM parameters

$$\alpha(n, i) = \sum_k [\overline{\alpha(n-1, k)} \overline{t(k, i)} \overline{P(x_n = x | y_n = i)}]$$

previous step
in recursion

Transition probability
from k to i

Probability the nth observation will be x
(given you're in state i)

!!Y el *algoritmo forward* aparece para hacer su magia!! Este algoritmo calcula la probabilidad de un nuevo símbolo de la secuencia observada sin necesidad de calcular las probabilidades de todas las

posibles rutas que forman la secuencia. En vez de eso, define una **variable forward** y calcula su valor recursivamente.

La recursión es la clave de que el algoritmo sea más rápido que calcular las probabilidades de las posibles rutas. En efecto, puede calcular la probabilidad de una x observada en la secuencia en solo LM^2 cálculos en vez de en LM^t . En el caso del ejemplo, con 2 estados y una secuencia de salida de longitud 3 la diferencia es hacer 12 pasos en lugar de 24.

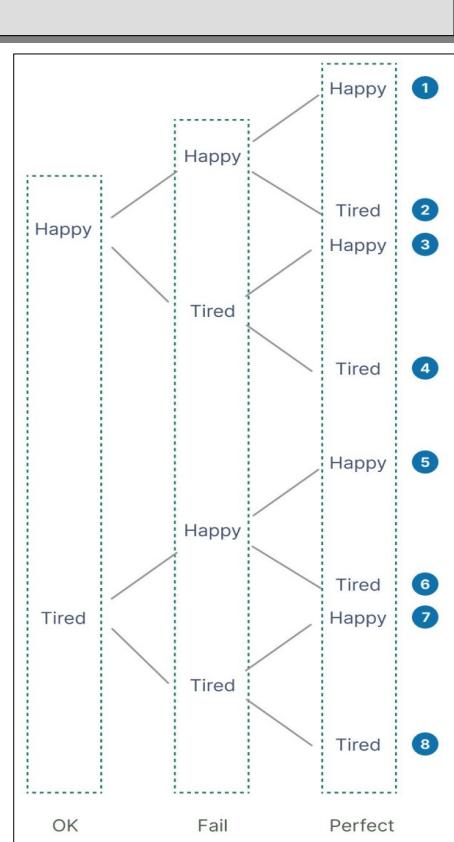
La reducción en la cantidad de cálculos se debe a una técnica de programación denominada **programación dinámica** que se ayuda de estructuras de datos para almacenar información intermedia y no tener que volver a calcularla varias veces. Cada vez que el algoritmo debe calcular una nueva probabilidad comprueba si ya la calculó con anterioridad y en ese caso accede a su valor almacenado en la estructura, lo que es más rápido que calcularla y si no lo había hecho lo hace y guarda su valor en la estructura. Enlace a una [explicación en vídeo](#).

ALGORITMO VITERBI

Pensando en *pseudocódigo*, si intentas decodificar por fuerza bruta la secuencia de estados ocultos que genera una secuencia concreta observada lo que necesitarías hacer es:

- Generar todas las posibles permutaciones de rutas que lleguen a la secuencia deseada.
- Usar el *algoritmo Forward* para calcular las probabilidades de cada secuencia de observaciones para cada posible secuencia de estados ocultos.
- Elegir la secuencia de estados ocultos con más alta probabilidad.

Para el HMM del ejemplo hay 8 posibles rutas que generan la salida *OK – Fail – Perfect*. Si añadimos una observación adicional !Tendremos el doble de cantidad de posibles secuencias de estados ocultos! (al tener 2 estados ocultos). Igual que hemos descrito la situación para el *algoritmo Forward*, al aumentar la longitud de las observaciones la complejidad del algoritmo por fuerza bruta crece exponencialmente. Y el *algoritmo Viterbi* es lo que nos permite escapar de esa gran cantidad de operaciones.



Cuando la secuencia de estados ocultos en el HMM es atravesada, en cada paso, la probabilidad $vt(j)$ es la probabilidad de que el HMM esté en el estado oculto j después de haber visto la observación y justo antes de atravesar el estado más probable que le permite alcanzar j .

$$v_t(j) = \max_{q_1, q_2, \dots, q_t} P(q_1 q_2 \dots q_{t-1}, \overline{o_1 o_2 \dots o_{t-1}}, q_t = j \mid \lambda)$$

indicates the algorithm
 is looking for
 the most probable path

observation sequence
 until the last time step

HMM parameters

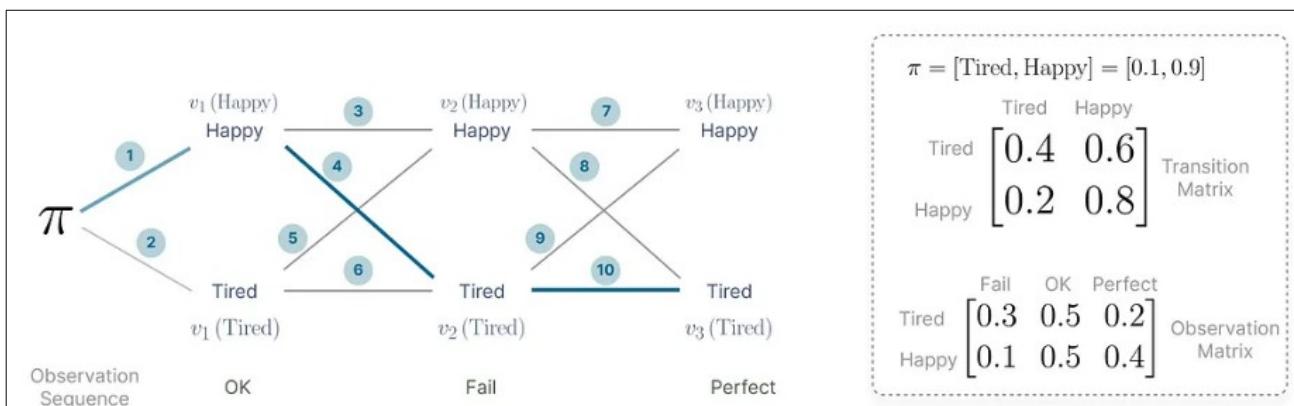
hidden path sequence
 until the last time step

current hidden state

Viterbi path
 to
 hidden state j

La clave para decodificar la secuencia de estados que generan la observación de una secuencia es el concepto de **ruta más probable**. También se le llama **ruta Viterbi** y es la ruta con más altas posibilidades de todas las rutas que pueden alcanzar a cualquier estado oculto.

Puedes pensar en un paralelismo entre el *algoritmo Forward* y el *algoritmo Viterbi* porque el primero suma todas las probabilidades para obtener las posibilidades de alcanzar cierto estado teniendo en cuenta todas las rutas que llegan a él y el *algoritmo Viterbi* no quiere explorar todas las posibilidades y se centra en las rutas más probables que llegan a cualquier estado. Aquí tienes un [enlace a las cadenas de Markov](#) y otro [enlace que explica el algoritmo](#) con más profundidad. Aquí tenemos una ejecución del algoritmo:



$$1 \rightarrow P(\text{Happy} | \text{Start}) \times P(\text{OK} | \text{Happy}) = 0.9 \times 0.5 = 0.45$$

$$v_1(\text{Happy}) = \textcircled{1} = P(\text{Happy} | \text{Start}) \times P(\text{OK} | \text{Happy}) = 0.9 \times 0.5 = 0.45$$

$$2 \rightarrow P(\text{Tired} | \text{Start}) \times P(\text{OK} | \text{Tired}) = 0.1 \times 0.5 = 0.05$$

$$v_1(\text{Tired}) = \textcircled{2} = P(\text{Tired} | \text{Start}) \times P(\text{OK} | \text{Tired}) = 0.1 \times 0.5 = 0.05$$

$$3 \rightarrow P(\text{Happy} | \text{Happy}) \times P(\text{Fail} | \text{Happy}) = 0.8 \times 0.1 = 0.08$$

$$4 \rightarrow P(\text{Tired} | \text{Happy}) \times P(\text{Fail} | \text{Tired}) = 0.2 \times 0.3 = 0.06$$

$$5 \rightarrow P(\text{Happy} | \text{Tired}) \times P(\text{Fail} | \text{Happy}) = 0.6 \times 0.1 = 0.06$$

$$6 \rightarrow P(\text{Tired} | \text{Tired}) \times P(\text{Fail} | \text{Tired}) = 0.4 \times 0.4 = 0.12$$

$$= \max((0.45 \times 0.08), (0.05 \times 0.06)) =$$

$$7 \rightarrow P(\text{Happy} | \text{Happy}) \times P(\text{Perfect} | \text{Happy}) = 0.8 \times 0.4 = 0.32$$

$$8 \rightarrow P(\text{Tired} | \text{Happy}) \times P(\text{Perfect} | \text{Tired}) = 0.2 \times 0.2 = 0.04$$

$$9 \rightarrow P(\text{Happy} | \text{Tired}) \times P(\text{Perfect} | \text{Happy}) = 0.6 \times 0.4 = 0.24$$

$$10 \rightarrow P(\text{Tired} | \text{Tired}) \times P(\text{Perfect} | \text{Tired}) = 0.4 \times 0.2 = 0.08$$

$$\begin{aligned}
 v_3(\text{Happy}) &= \max((v_2(\text{Happy}) \times \textcircled{7}), (v_2(\text{Tired}) \times \textcircled{9})) = \\
 &= \max((0.36 \times 0.32), (0.027 \times 0.24)) = \\
 &= \max(0.012, 0.006) = 0.012
 \end{aligned}$$

$$\begin{aligned}
 v_3(\text{Tired}) &= \max((v_2(\text{Happy}) \times \textcircled{8}), (v_2(\text{Tired}) \times \textcircled{10})) = \\
 &= \max((0.36 \times 0.04), (0.027 \times 0.08)) = \\
 &= \max(0.001, 0.002) = 0.002
 \end{aligned}$$

Decode Sequence using Backpointers

$$\begin{aligned}
 \text{Decoded Sequence} &= \max(v_3(\text{Happy}), v_3(\text{Tired})) \\
 &= v_3(\text{Tired}) \rightarrow v_2(\text{Tired}) \rightarrow v_1(\text{Happy})
 \end{aligned}$$



Otra característica del *algoritmo de Viterbi* es que debe registrar las rutas que le permiten llegar a un estado para comparar sus probabilidades. Para hacer esto utiliza punteros a cada estado oculto usando estructuras de datos típicas de la programación dinámica.

En el ejemplo del examen del perro se calcula las rutas $v3(\text{Happy})$ y $v3(\text{Tired})$, se elige el de más alta probabilidad y comienza retrocediendo usando *backtracking*, atravesando todas las rutas que permiten llegar al sitio donde estás.

EJEMPLO 46: Crear el ejemplo del HMM del examen del perro en Python. Programar los algoritmos *Forward* y *Viterbi* es complejo, así que usaremos un módulo que ya los tiene hechos y

que se llama hmmlearn.

```

from hmmlearn import hmm
import numpy as np
## Parte 1. Generar un modelo HMM definiendo parámetros
print("===== Configurando modelo HMM")
print(" ini_param inicializan el modelo")
print(" s         semilla de probabilidad")
print(" t         probabilidades de transición")
print(" e         probabilidades de emisión")
modelo = hmm.CategoricalHMM(n_components=2, random_state=425, init_params='ste')
# probabilidades iniciales
# probabilidad de iniciar con el estado Tired = 0
# probabilidad de iniciar con el estado Happy = 1
distribucion_inicial = np.array([0.1, 0.9])
modelo.startprob_ = distribucion_inicial
print("-- Paso 2. Definir Matriz de Transición")
# Las probabilidades de transmisión son
#     tired happy
# tired  0.4  0.6
# happy   0.2  0.8
distribucion_transmision = np.array([[0.4, 0.6], [0.2, 0.8]])
modelo.transmat_ = distribucion_transmision
print("-- Paso 3. Definir Matriz de Probabilidades de Observación")
# probabilidades_observacion
#     Fail    OK    Perfect
# tired  0.3  0.5      0.2
# happy   0.1  0.5      0.4
matriz_observacion = np.array([[0.3, 0.5, 0.2], [0.1, 0.5, 0.4]])
modelo.emissionprob_ = matriz_observacion
print("-- Paso 4. Muestras simuladas")
intentos, estados_simulados = modelo.sample(100000)
# Una muestra de los intentos simulados
# 0 -> Fail
# 1 -> OK
# 2 -> Perfect
print("\nMuestra de ejemplo de los intentos simulados")
print(intentos[:10])
print("===== PARTE 2 - Decodificar la secuencia de estados ocultos")
## A una secuencia OK - Fail - Perfect
X_train = intentos[:intentos.shape[0] // 2]
X_test = intentos[intentos.shape[0] // 2:]
modelo.fit(X_train)
exam_observations = [[1, 0, 2]]
estados_predichos = modelo.predict(X=[[1, 0, 2]])
print("Predicciones de estados que generan OK, Fail, Perfect: \n 0 -> Tired , "
"1 -> Happy")
print(estados_predichos)

```

6. EJERCICIOS.

EJERCICIO 17 tenemos un tarro con 20 bolas de colores rojo y negro que tienen inscrito un número. Vas a usarlo para calcular probabilidades y probabilidades condicionadas cuando sacamos 3 bolas sin mirar, al azar. Calcula:



a) P(roja):

b) P(negra) sin necesidad de contar las negras que hay.

- b) $P(\text{par})$
- c) $P(\text{roja y par})$
- d) $P(< 20)$
- e) $P(\text{roja / par})$
- f) $P(\text{par/roja})$ calculada de dos formas diferentes

Para facilitar los cálculos puedes usar una tabla de contingencia que da acceso a todos los contadores de forma resumida:

Tabla de doble entrada o de contingencia

		$A: \text{rojas}$	$\bar{A}: \text{negras}$	
		6	7	13
$B: \text{pares}$	2	5	7	
	8	12	20	

EJERCICIO 2. El 55.26% de los automóviles de un estacionamiento son de 4 puertas. Los automóviles de color blanco son el 21.27% del total y los automóviles de 4 puertas escogidos de entre los blancos son el 59.77%. Determina el porcentaje de coches blancos escogidos de entre los de 4 puertas.

$$A = 4 \text{ puertas}$$

$$B = \text{blanco}$$

$$\text{¿P}(B / A)?$$

Solución: Sea

$$A = \text{Porcentaje de 4 puertas} = 55.26\% = 0.5526$$

$$B = \text{Porcentaje de blancos} = 21.27\% = 0.2127$$

$$A/B = \text{Porcentaje de 4 puertas que son blancos} = 59.77\% = 0.5977$$

El porcentaje deseado es: $P(\text{blancos que son de 4 puertas})$, lo cual puede obtenerse aplicando la fórmula de Bayes para probabilidades condicionales. $P(B/A) = \frac{P(B \cap A)}{P(A)} = \frac{P(A/B) P(B)}{P(A)}$

EJERCICIO 3. Una empresa de tecnología tiene 3 departamentos: Desarrollo (D), Marketing (M) y Ventas (V). La empresa tiene en la actualidad 100 empleados que se reparten de la siguiente manera: 50 trabajan en desarrollo, 30 en marketing y 20 en ventas.

Se sabe que la probabilidad de que un empleado reciba un aumento de sueldo no es igual en todos los departamentos:

- $P(\text{aumento} / D) = 0.6$
- $P(\text{aumento} / M) = 0.4$
- $P(\text{aumento} / V) = 0.2$

¿Cuál es la probabilidad total de que un empleado seleccionado al azar reciba un aumento?

Nota: Utiliza la regla de la cadena para calcular la probabilidad total:

$$P(A) = P(A/B_1) P(B_1) + P(A / B_2) P(B_2) + \dots + P(A/B_n) P(B_n)$$

EJERCICIO 4: El EJEMPLO 3 quizás tenga aun error de cálculo. Te propongo que hagas un programa en Python que realice los cálculos.

EJERCICIO 5. Calcula la probabilidad de que en el EJEMPLO 5, el correo con las palabras “*Lunch money money*” sea *spam* o *normal*:

- a) Justo antes de añadir el contador extra ([enlace](#): cuando hay 8 correos normales con 17 palabras en *normal* y 4 correos *spam* con 7 palabras en *spam*)
- b) Justo después de añadir un contador fijo para evitar el cero a cada palabra.

EJERCICIO 6: Usando los datos del ejercicio anterior, entrena un modelo *Bayesiano* creado con scikit-learn. Y predice lo mismo que en el ejemplo3.

EJERCICIO 7. Entrena un modelo Bayesiano para los datos del EJEMPLO 4 y predice el mismo caso que en el ejemplo.

EJERCICIO 8. Vamos a clasificar frases en dos categorías: Deportivas y tecnológicas. Copia el siguiente código:

```

1 import numpy as np
2 from sklearn.feature_extraction.text import TfidfVectorizer
3 from sklearn.naive_bayes import MultinomialNB
4 from sklearn.model_selection import train_test_split
5 from sklearn.metrics import accuracy_score
6 # Definir los datos
7 textos = [
8     "El equipo ganó el partido", "Gran jugada en la final de fútbol",
9     "El nuevo teléfono tiene una gran cámara", "Apple lanza una nueva actualización",
10    "El tenista venció a su rival", "Los procesadores Intel ahora son más rápidos",
11    "ram y cpu son elementos básicos del computador", "Un sistema operativo muy exigente de recursos",
12    "la selección nacional de fútbol ha jugado un buen partido", "han batido el record del mundo",
13    "los partidos de la nba y la acb me gustan", "las gui necesitan una cpu potente"
14 ]
15 clases= ["deporte", "tecnología"]
16 etiquetas = [0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1] # 0 = deporte, 1 = tecnología
17 # Preproceso
18 vectorizador = TfidfVectorizer()
19 X = vectorizador.fit_transform(textos) # Convertir textos en matriz de características
20 X_train, X_test, y_train, y_test = train_test_split(X, etiquetas, test_size=0.2, random_state=449)
21 # Entrenar y validar modelo
22 modelo = MultinomialNB()
23 modelo.fit(X_train, y_train)
24 y_pred = modelo.predict(X_test)
25 accuracy = accuracy_score(y_test, y_pred)
26 print("Exactitud del modelo:", accuracy)

```

a) ¿Por qué usamos *MultinomialNB* y no *GaussianNB*?

b) Modifica el código para usar tu semilla aleatoria y añade código al final de manera que pregunte repetidamente por una frase hasta que la frase esté vacía y haga la predicción de a qué clase pertenece (deporte o tecnología) mostrando las probabilidades calculadas.

```

Exactitud del modelo: 0.6666666666666666
'el sistema operativo no está actualizado' clasificada como: tecnología
Probabilidades: [[0.39893332 0.60106668]]
'el partido acabó mal' clasificada como: deporte
Probabilidades: [[0.58652296 0.41347704]]

```

EJERCICIO 9: Copia el siguiente código que define 100 muestras de peso y altura de 5 hombres y 50 mujeres, luego implementa y entrena un clasificador de tipo *GaussianNB* (naïve Bayes gausiano), lo valida y dibuja los datos usados y lo testeá mostrando su desempeño.

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from sklearn.model_selection import train_test_split
4 from sklearn.naive_bayes import GaussianNB
5 from sklearn.metrics import accuracy_score
6
7 np.random.seed(449) # Personaliza la semilla del generador aleatorio
8 # Generamos datos: altura(cm) peso(kg) de 50 hombres(clase 0) y mujeres(clase 1)
9 altura_hombres = np.random.normal(loc=175, scale=6, size=50)
10 peso_hombres = np.random.normal(loc=78, scale=8, size=50)
11 altura_mujeres = np.random.normal(loc=162, scale=5, size=50)
12 peso_mujeres = np.random.normal(loc=60, scale=6, size=50)
13 X = np.vstack((np.column_stack((altura_hombres, peso_hombres)),
14 | | | | np.column_stack((altura_mujeres, peso_mujeres))))
15 y = np.array([0] * 50 + [1] * 50) # 0 = Hombre, 1 = Mujer
16 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=449)
17
18 # Entrenar y validar el modelo
19 modelo = GaussianNB()
20 modelo.fit(X_train, y_train)
21 y_pred = modelo.predict(X_test)
22 accuracy = accuracy_score(y_test, y_pred)
23 print("Precisión del modelo:", accuracy)
24
25 plt.scatter(altura_hombres, peso_hombres, label="Hombres", color="blue", alpha=0.6)
26 plt.scatter(altura_mujeres, peso_mujeres, label="Mujeres", color="red", alpha=0.6)
27 plt.xlabel("Altura (cm)")
28 plt.ylabel("Peso (kg)")
29 plt.legend()
30 plt.title("Distribución de altura y peso por género") |

```

a) Modifica el código y personaliza los procesos aleatorios con tu semilla aleatoria (líneas 7 y 16). Luego añade código para que pregunte repetidamente (hasta que se indique un peso 0) un peso y una altura e indique si corresponden a un hombre o a una mujer.

b) ¿Porqué hemos escogido el *GaussianNB* en vez de por ejemplo el el *MultinomialNB*?

EJERCICIO 10. Responde a estas cuestiones sobre las máquinas de vector soporte.

- c1. ¿La idea fundamental en que se basa las *Support Vector Machines*?
- c2. ¿Qué es un vector soporte?
- c3. ¿Porqué es importante escalar las entradas al usar SVM?
- c4. ¿Un clasificador SVM puede devolver un *score* de confidencia cuando clasifica una instancia? ¿Y una probabilidad?
- c5. ¿Deberías usar la versión única o la dual de una SVM para entrenar un modelo con millones de instancias y miles de características? ¿Cuál sería el modelo a elegir en scikit?
- c6. Imagina que entrenas un clasificador SVM con *kernel RBF*. Compruebas que tiene *underfitting* con el set *train* ¿Debes incrementar o decrementar γ (gamma)? ¿Y C?

EJERCICIO 11: ejecuta el código del [ejemplo 20](#) e indica la consecuencia de:

- a) Subir de valor el parámetro gamma del detector de *outliers*.
- b) Bajar el valor del parámetro gamma del detector de *outliers*.

EJERCICIO 12: Usa el enlace del [ejemplo](#) que compara las clases `Linear.SGDOneClassSVM` y `svm.OneClassSVM` y usando uno de los métodos más sencillos de detección de *outliers* que vimos en la unidad 1, analiza los resultados y saca conclusiones de cada método.

EJERCICIO 13: Copia el siguiente código que define unos datos para realizar clasificaciones en 2 clases y crea, entrena y valida un clasificador de tipo SVC.

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from sklearn.svm import SVC
4 from sklearn.datasets import make_classification
5 from sklearn.model_selection import train_test_split
6 from sklearn.metrics import accuracy_score
7
8 # Generar datos (2 clases)
9 X, y = make_classification(n_samples=300, n_features=2, n_classes=2,
10 | | | | | | | | | | | | n_clusters_per_class=1, n_redundant=0, random_state=449)
11 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=449)
12
13 # Entrenar un SVM con underfitting (C bajo, gamma bajo)
14 svm_under = SVC(kernel='rbf', C=0.1, gamma=0.01)
15 svm_under.fit(X_train, y_train)
16
17 # Predecir y evaluar
18 train_acc_under = accuracy_score(y_train, svm_under.predict(X_train))
19 test_acc_under = accuracy_score(y_test, svm_under.predict(X_test))
20 print(f"SVM Underfitting - Accuracy Train: {train_acc_under:.3f}, Test: {test_acc_under:.3f}")

```

a) Modifica el código y personaliza la semilla de procesos aleatorios (líneas 10 y 11). Lo ejecutas y muestra el accuracy que se consigue en train y test e indica el % de underfitting que tienes: $(\text{score_train} - \text{score_test}) / \text{score_train} * 100$

b) Añade esta función que colorea y dibuja las fronteras de decisión y representa los datos de entrenamiento y destaca los vectores soporte.

```

23 def plot_frontera_decision(modelo, X, y):
24     xx, yy = np.meshgrid(np.linspace(X[:, 0].min()-1, X[:, 0].max()+1, 100),
25 | | | | | | | | | | | | np.linspace(X[:, 1].min()-1, X[:, 1].max()+1, 100))
26     Z = modelo.decision_function(np.c_[xx.ravel(), yy.ravel()])
27     Z = Z.reshape(xx.shape)
28     # Graficar frontera de decisión, datos y vectores soporte
29     plt.contourf(xx, yy, Z, levels=[-1, 0, 1], alpha=0.3, colors=['blue', 'black', 'red'])
30     plt.contour(xx, yy, Z, levels=[-1], colors='red', linestyles='dashed')
31     plt.contour(xx, yy, Z, levels=[1], colors='blue', linestyles='dashed')
32     X_0 = X[y == 0]
33     X_1 = X[y != 0]
34     plt.scatter(X_0[:, 0], X_0[:, 1], color='green', edgecolors='k', label='Clase 0')
35     plt.scatter(X_1[:, 0], X_1[:, 1], color='red', edgecolors='k', label='Clase 1')
36     plt.scatter(modelo.support_vectors_[:, 0], modelo.support_vectors_[:, 1],
37 | | | | | s=100, facecolors='none', edgecolors='k', linewidths=1.1, label="Vectores Soporte")
38     plt.legend()
39     titulo=f"SVC (C={modelo.C}, gamma={modelo.gamma})"
40     plt.title(titulo)
41     plt.show()

```

c) Ahora en una nueva celda del *notebook* crea y entrena con los mismos datos un nuevo *SVC* cambiando el valor de los hiperparámetros *C* y *gamma* que consigan bajar el porcentaje de underfitting por debajo del 2%. Muestra el gráfico y los valores de accuracy como en el caso anterior que se realizaría:

```

43 # Gráfica del modelo con underfitting
44 plot_frontera_decision(svm_under, X_train, y_train)

```

EJERCICIO 14. Responde a estas preguntas sobre los árboles de decisión.

1. ¿Cuál podría ser aproximadamente la profundidad de un árbol de decisión entrenado sin restricciones usando un conjunto de entrenamiento de un millón de muestras?
2. Un nodo cuando se utiliza Gini como criterio de división su nivel de impureza es normalmente más bajo o más alto que sus nodos antecesores? ¿normalmente o siempre?
3. Si un árbol de decisión tiene *overfitting* ¿es buena idea decrementar ***max_depth***?
4. Un árbol de decisión tiene *underfitting* ¿Es buena solución escalar características?
5. Si tardamos una hora en entrenar un árbol de decisión en un conjunto de datos de entrenamiento con 1 millón de muestras, ¿Cuánto tardemos aproximadamente en entrenar otro con un dataset de 100 millones de instancias?
6. Si entrenas un árbol con 100 mil instancias ¿Aumentas la velocidad si indicas ***presort=True*** durante el entrenamiento?

EJERCICIO 15. Tienes este conjunto de datos y un algoritmo de aprendizaje ID3 (utiliza la entropía como métrica) va a construir un árbol de decisión.

a) Consulta el algoritmo en el [enlace](#) y haz los cálculos necesarios a mano para averiguar cuál es la primera pregunta o la primera prueba que se aplicará en el nodo raíz del árbol (la mejor división). Usa este algoritmo y haz los cálculos a mano.

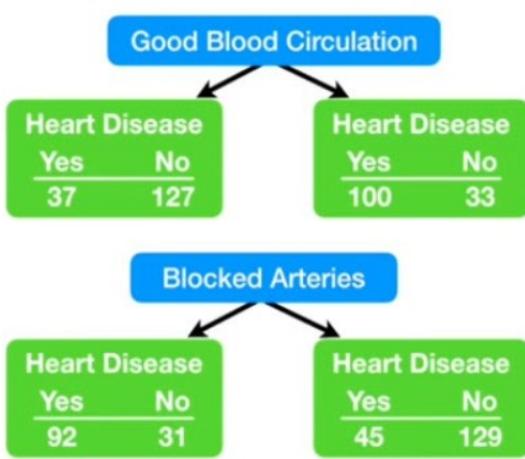
b) Crea un programa en python que implemente estos pasos.

c) Crea otro programa que use scikit a ver si obtiene resultados similares.

d) ¿ID3 sirve para hacer regresiones?

dientes	pelo	pulmones	patas	especie
True	True	True	True	mamífero
True	True	True	True	mamífero
True	False	True	False	reptil
False	True	True	True	mamífero
True	True	True	True	mamífero
True	True	False	True	mamífero
True	False	False	False	reptil
True	False	True	False	reptil
True	True	True	True	mamífero
False	False	True	True	reptil

EJERCICIO 16. En un árbol CART (utiliza Gini como métrica de impureza) se estudia dividir por dos criterios que generan las divisiones en clases que se ven en la figura:



- a) Calcula la impureza que obtendrías si divides por la pregunta "Buena circulación?"
- b) Calcula la impureza si preguntas por "Arterias bloqueadas?".
- c) Indica qué pregunta escogería, es decir, cuál sería la mejor división.

EJERCICIO 17: En el ejercicio 24, ¿porque ponemos el parámetro ***probability=true*** en el constructor del clasificador ***SVC*** que se utiliza en el método ensamblaje ***voting_cla***?

EJERCICIO 18: EJEMPLO DE MODELO HMM MULTINOMIAL. Un modelo con distribución categórica generan salidas (como el lanzamiento de un dado) con ***n_features*** posibles valores, es decir, son una generalización de la distribución de probabilidad de *Bernoulli* porque hay ***n_features*** categorías en vez de dos posibles resultados: éxito/fallo. Un HMM categórico tiene

EJERCICIO 18: EJEMPLO DE MODELO HMM MULTINOMIAL. Un modelo con distribución categórica generan salidas (como el lanzamiento de un dado) con ***n_features*** posibles valores, es decir, son una generalización de la distribución de probabilidad de *Bernoulli* porque hay ***n_features*** categorías en vez de dos posibles resultados: éxito/fallo. Un HMM categórico tiene

probabilidades de emisión para cada componente parametrizado por distribuciones Categóricas.

Un modelo Multinomial genera *n_trials* tiradas independientes de un dado, cada una con *n_features* posibles valores, es decir:

- Cuando *n_trials* = 1 y *n_features* = 2, es como una distribución de *Bernoulli*.
- Cuando *n_trials* > 1 y *n_features* = 2, es una distribución *Binomial*.
- Cuando *n_trials* = 1 y *n_features* > 2, es una distribución Categórica.

Las probabilidades de emisión de cada componente de un **HMM Multinomial** está parametrizado por distribuciones *Multinomiales*.

Sospechamos que un casino está intercambiando un dado (uno o dos dados) por un dado trucado. Queremos averiguar:

- 1) Cuándo se utiliza el dado trucado (es decir, la ruta más probable)
- 2) Con qué frecuencia se utiliza el dado trucado (es decir, probabilidades de transición) y
- 3) Probabilidades de cada resultado en una tirada (es decir, probabilidades de emisión).

Copia el siguiente código:

```

1 # Cargamos librerías
2 import numpy as np
3 import matplotlib.pyplot as plt
4 from hmmlearn import hmm
5
6 # Hacemos modelo generativo con 2 componentes; dado legal + dado trucado
7 gen_modelo = hmm.CategoricalHMM(n_components=2, random_state=449) # Personaliza tu semilla
8 # El primer estado es el dado legal así que comenzamos siempre con él
9 gen_modelo.startprob_ = np.array([1.0, 0.0])
10 # Ahora usamos el dado trucado:
11 # - Tenemos un 95 % de posibilidades de seguir usando el dado ok y un 5 %
12 #   de posibilidades de cambiar al dado trucado
13 # - Cuando entramos en el estado del dado trucado, tenemos un 90 % de
14 #   posibilidades de permanecer en ese estado y un 10 % de posibilidades de salir
15 gen_modelo.transmat_ = np.array([[0.95, 0.05],
16 |                                | [0.1, 0.9]])
17 # El primer estado(fila 0) es del dado legal: probabilidades típicas de un dado.
18 # El segundo (dado trucado) y está sesgado hacia el 6 (lo peor en el juego)
19 gen_modelo.emissionprob_ = np.array([[1/6, 1/6, 1/6, 1/6, 1/6, 1/6],
20 |                                | [1/10, 1/10, 1/10, 1/10, 1/10, 1/2]])
21 # simulamos 30 mil tiradas
22 tiradas, gen_estados = gen_modelo.sample(30000)

```

Y creamos dos gráficos donde vemos como en las primeras 500 tiradas se va cambiando de dado:

```

23 # Dibujamos los estados a lo largo del tiempo
24 fig, ax = plt.subplots()
25 ax.plot(gen_estados[:500])
26 ax.set_title('Estados sobre tiempo')
27 ax.set_xlabel('Tiempo (# de tiradas)')
28 ax.set_ylabel('Estado')
29 plt.show()

```

```

30 # Dibujamos tiradas del dado ok y del trucado
31 fig, ax = plt.subplots()
32 ax.hist(tiradas[gen_estados == 0], label='legal', alpha=0.5,
33 |      bins=np.arange(7) - 0.5, density=True)
34 ax.hist(tiradas[gen_estados == 1], label='trucado', alpha=0.5,
35 |      bins=np.arange(7) - 0.5, density=True)
36 ax.set_title('Probabilidades de tirada por estado')
37 ax.set_xlabel('Contador')
38 ax.set_ylabel('Tirada')
39 ax.legend()
40 plt.show()

```

Ahora, vamos a volver a recalcular los estados ocultos, la matriz de transición y la matriz de emisión entrenando un nuevo modelo con las observaciones generadas con nuestro sintético que intentaba reproducir el escenario del casino. Copia el código y personaliza las semillas aleatorias (línea 9).

```

1 # dividimos datos en train y test (50/50)
2 X_train = tiradas[:tiradas.shape[0] // 2]
3 X_test = tiradas[tiradas.shape[0] // 2:]
4
5 # Buscar un modelo con score óptimo
6 gen_score = gen_modelo.score(X_test)
7 mejor_score = mejor_modelo = None
8 n_entrenamientos = 50
9 np.random.seed(449)      # Personaliza la semilla aleatoria
10 for idx in range(n_entrenamientos):
11     modelo1 = hmm.CategoricalHMM(n_components=2, random_state=idx, init_params='se')
12     # No se inicializa la matriz de transiciones, deben dejarse aleatorias
13     # porque el valor por defecto es una transición likelihood
14     # sabemos que son raras (de otra forma el casino queda descubierto!)
15     # así que usamos una distribución aleatoria Dirichlet a-priori con un alfa de
16     # (0.1, 0.9) para forzar nuestra suposición de que las transiciones ocurren un 10%
17     modelo1.transmat_ = np.array([np.random.dirichlet([0.9, 0.1]),
18                                   np.random.dirichlet([0.1, 0.9])])
19     modelo1.fit(X_train)
20     score = modelo1.score(X_test)
21     print(f'Modelo #{idx}\tScore: {score}')
22     if mejor_score is None or score > mejor_score:
23         mejor_modelo = modelo1
24         mejor_score = score
25 print(f'Score generado: {gen_score}\tmejor score: {mejor_score}')
26

```

Una vez tenemos el modelo entrenado, generamos los mejores estados ocultos posibles con el algoritmo de codificación, en este caso **Viterbi**.

```

27 # Usar Viterbi para predecir secuencia de estados más probable
28 estados = mejor_modelo.predict(tiradas)
29 # Dibujar los estados recuperados comparados con los generados (primer objetivo)
30 fig, ax = plt.subplots()
31 ax.plot(gen_estados[:500], label='generados')
32 ax.plot(estados[:500] + 1.5, label='recuperados')
33 ax.set_yticks([])
34 ax.set_title('Estados vs Generados')
35 ax.set_xlabel('Tiempo (# tiradas)')
36 ax.set_xlabel('Estado')
37 ax.legend()
38 plt.show()

```

Y tras dibujar el gráfico que nos muestra cuando se está utilizando el dado trucado (probabilísticamente) intentamos responder las otras cuestiones:

- 2) Con qué frecuencia se utiliza el dado trucado (es decir, probabilidades de transición)

```

1 # Comprobamos transiciones aprendidas y miramos si concuerdan
2 print(f'Matriz de Transición Generada:\n{gen_modelo.transmat_.round(3)}\n\n'
3 |     f'Matriz de Transición Recuperada:\n{mejor_modelo.transmat_.round(3)}\n\n')

```

3) Probabilidades de cada resultado en una tirada (es decir, probabilidades de emisión).

```

4 # Finalmente vemos de qué forma está trucado el dado
5 print(f'Matriz de Emisiones Generada:\n{gen_modelo.emissionprob_.round(3)}\n\n'
6 |     f'Matriz de Emisiones Recuperada:\n{mejor_modelo.emissionprob_.round(3)}\n\n')

```

EJERCICIO 19: Predicción del Clima con un *Modelo Oculto de Markov (HMM)*. Vamos a modelar una situación donde el **clima** (soleado, nublado, lluvioso) es un **estado oculto** y lo que observamos son actividades diarias (caminar, ir de compras, limpiar de personas). Es decir, nuestro problema es modelizar **Estados ocultos (Clima)**: ☀ Soleado (0), ☁ Nublado (1) y ☨ Lluvioso (2) cuando observamos **Observaciones (Actividades)**: ⚑ Caminar (0), 🛍 Comprar (1) y 🏠 Limpiar (2). Las probabilidades iniciales, de transición y emisión quedan recogidas en esta figura. Usa python para modelizar este sistema y genera 20 secuencias que debes imprimir:

