# Máquinas de soporte vectorial

Minería de Datos: Preprocesamiento y clasificación

Salvador García

salvagl@decsai.ugr.es

# SVMs: Un poco de historia

- Las máquinas de vectores soporte (Support Vector Machines –SVMs-) se introducen en 1992 por Boser, Guyon, Vapnik en COLT-92
- Se populariza por la comunidad de RNAs, pero hoy día es un campo muy activo en todo el campo de Machine Learning
- ¡Ojo! Las SVMs son un caso particular de "Kernel Machines" (KM)← amplia familia de algoritmos de aprendizaje

# Clasificación SVM lineal

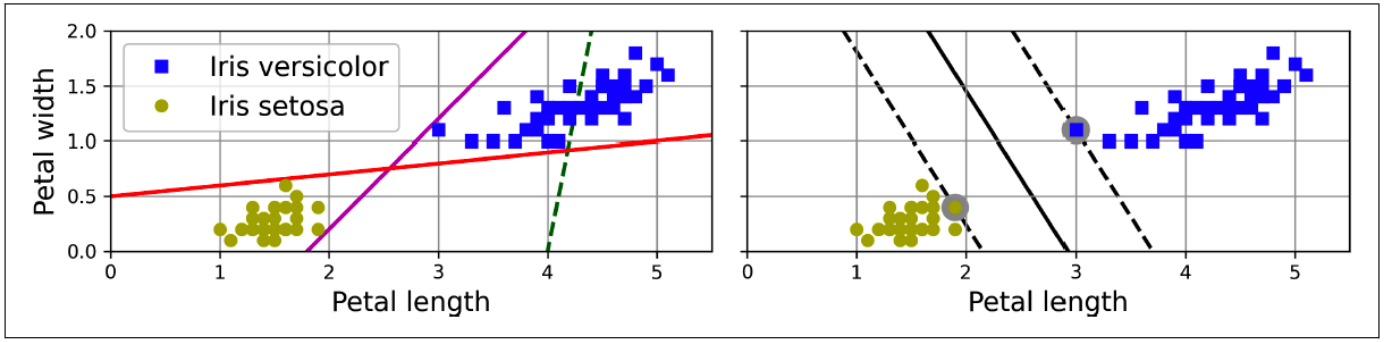
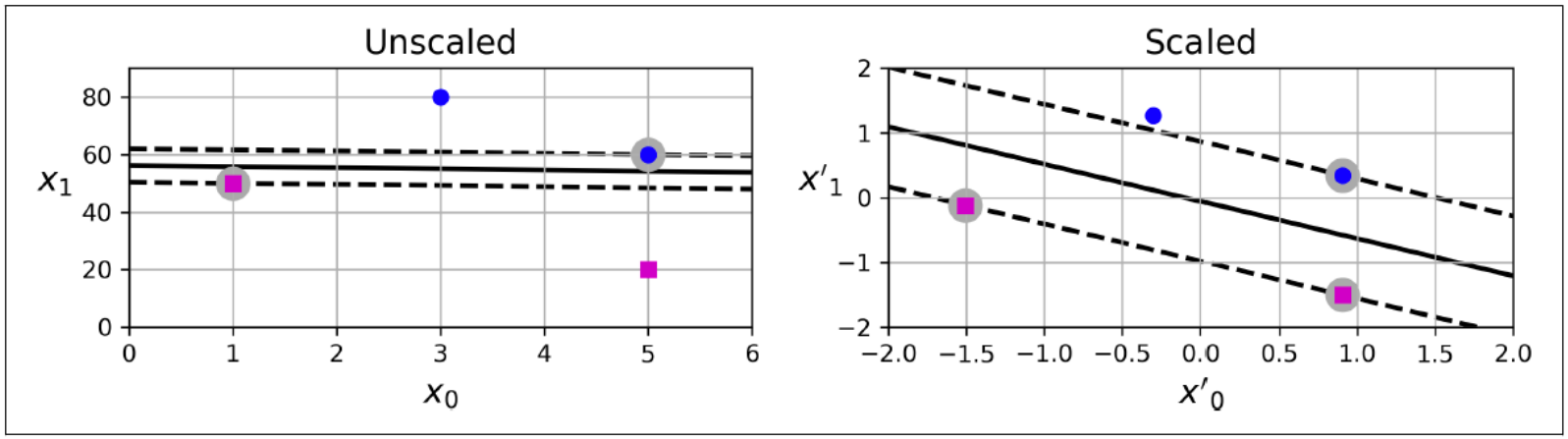



*Figure 5-1. Large margin classification*

- Las dos clases son linealmente separables.
- El gráfico de la izquierda muestra los límites de decisión de tres posibles clasificadores lineales. El modelo cuyo límite de decisión está representado por la línea discontinua es tan malo que ni siquiera separa correctamente las clases. Los otros dos modelos funcionan perfectamente con este conjunto de entrenamiento, pero sus límites de decisión se acercan tanto a las instancias que probablemente no funcionen tan bien con instancias nuevas.
- Por el contrario, la línea continua del gráfico de la derecha representa el límite de decisión de un clasificador SVM; esta línea no sólo separa las dos clases, sino que también se mantiene lo más alejada posible de las instancias de entrenamiento más cercanas. Puede pensar que un clasificador SVM se ajusta a la calle más ancha posible (representada por las líneas discontinuas paralelas) entre las clases.
- Esto se denomina *clasificación de gran margen o margen máximo*.

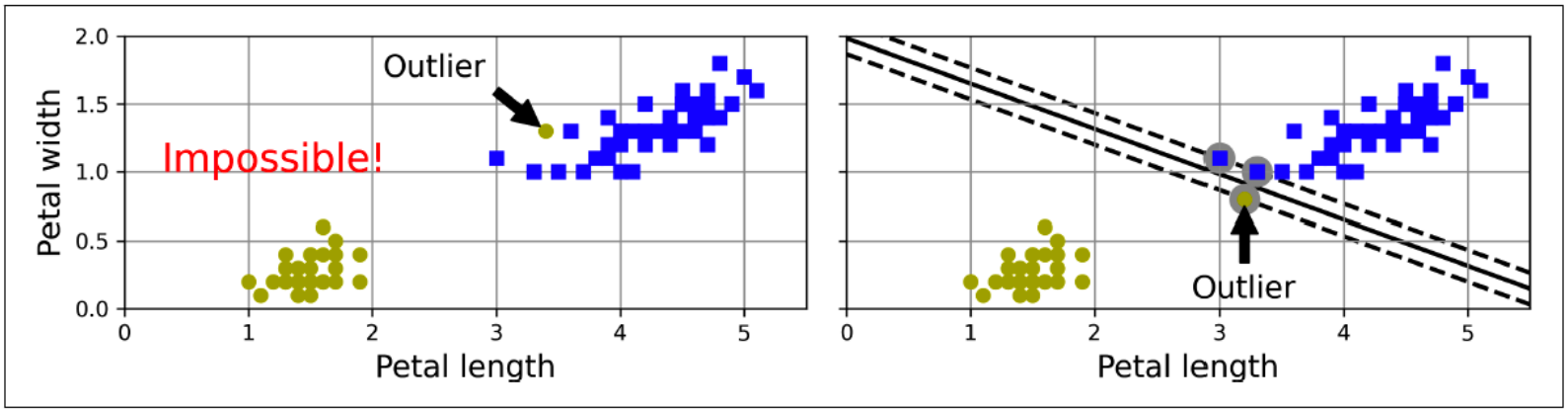
# Clasificación SVM lineal

- Añadir más instancias de entrenamiento "fuera de la calle" no afectará en absoluto al límite de decisión: está totalmente determinado (o "apoyado") por las instancias situadas en el borde de la calle.
- Estas instancias de borde se denominan *vectores de soporte*.
- Las SVM son sensibles a las escalas de las características, como puede verse en esta figura.



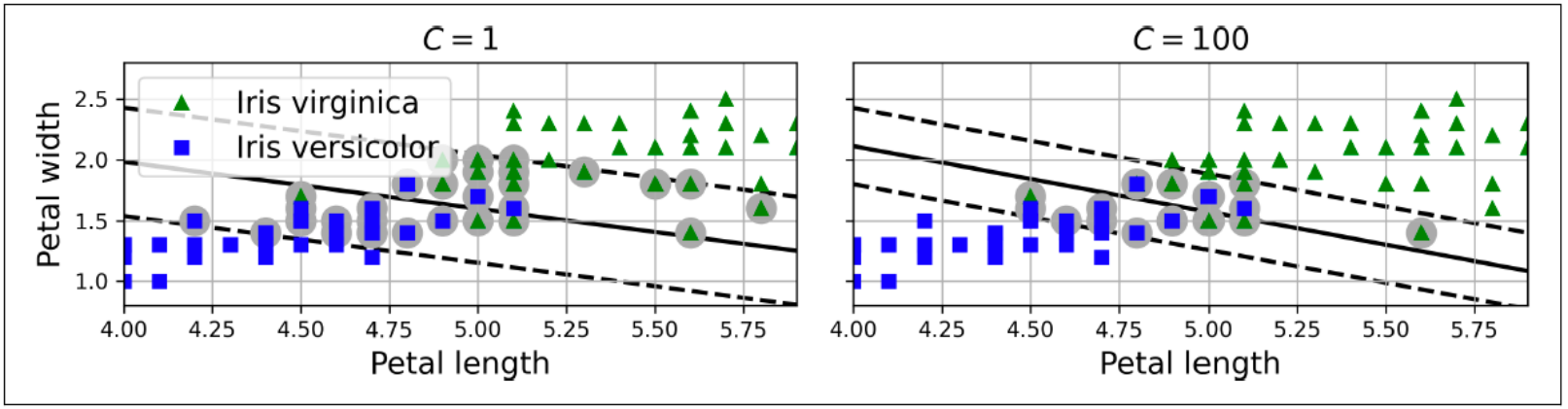
# Clasificación del margen blando

- Si imponemos estrictamente que todas las instancias deben estar fuera de la calle y en el lado correcto, esto se denomina *clasificación de margen duro*.
- Dos problemas con la clasificación de márgenes duros.
  - Sólo funciona si los datos son linealmente separables.
  - Es sensible a los valores atípicos.



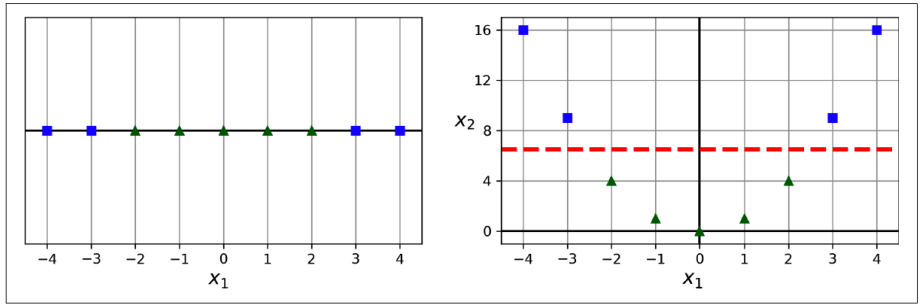
# Clasificación del margen blando

- Para evitar estos problemas, tenemos que utilizar un modelo más flexible. El objetivo es encontrar un buen equilibrio entre mantener la calle lo más grande posible y limitar las violaciones del margen. Esto se denomina *clasificación de margen suave*.
- Al crear un modelo SVM con Scikit-Learn, puede especificar varios hiperparámetros, incluido el hiperparámetro de regularización C.



# Clasificación SVM no lineal

- Aunque los clasificadores SVM lineales son eficientes y a menudo funcionan sorprendentemente bien, muchos conjuntos de datos ni siquiera están cerca de ser linealmente separables.
- Un método para tratar los conjuntos de datos no lineales consiste en añadir más características, como las polinómicas (como hicimos con la regresión polinómica); en algunos casos, esto puede dar lugar a un conjunto de datos linealmente separable.
- Consideremos el gráfico de la izquierda de la siguiente figura: representa un conjunto de datos simple con una sola característica, $x_1$ . Este conjunto de datos no es linealmente separable. Pero si se añade una segunda característica $x_2 = (x_1)^2$ , el conjunto de datos 2D resultante es perfectamente separable linealmente.

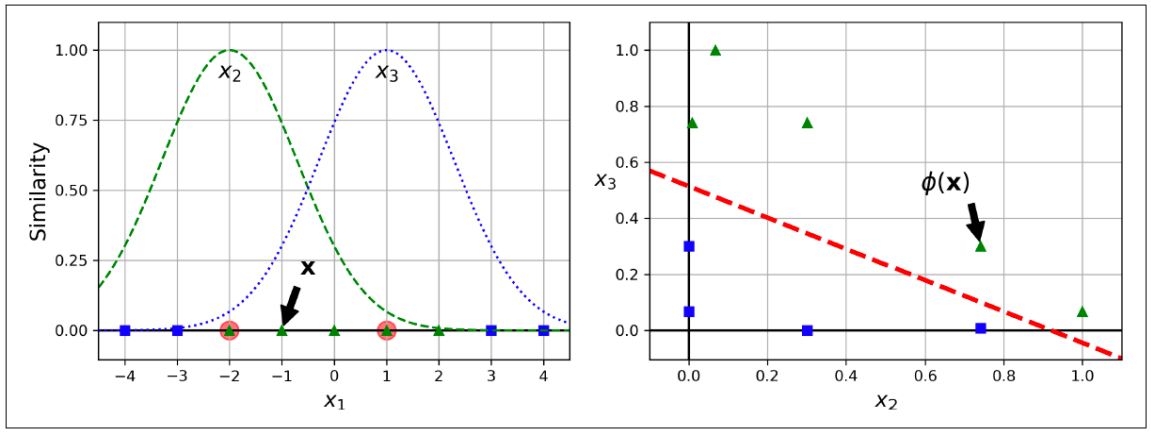


# Kernel polinómico

- Añadir características polinómicas es sencillo de implementar y puede funcionar muy bien con todo tipo de algoritmos de aprendizaje automático.
- Con un grado polinómico bajo, este método no puede tratar conjuntos de datos muy complejos, y con un grado polinómico alto crea un número enorme de características, lo que hace que el modelo sea demasiado lento.
- Afortunadamente, cuando se utiliza SVM se puede aplicar una técnica matemática casi milagrosa llamada el *truco del kernel*.
- El truco del kernel permite obtener el mismo resultado que si hubiera añadido muchas características polinómicas, incluso con un grado muy alto, sin tener que añadirlas realmente.
- Esto significa que no hay explosión combinatoria del número de características.

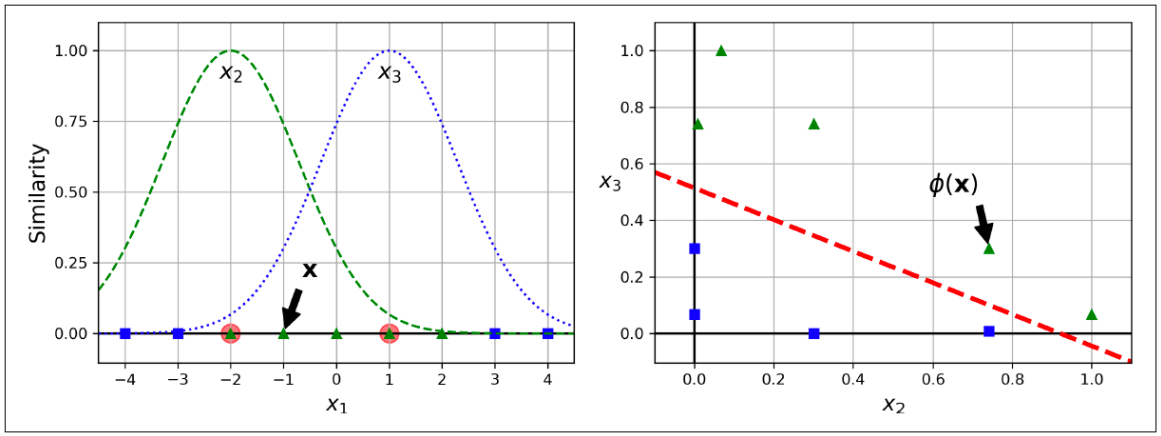
# Características de similitud

- Otra técnica para abordar problemas no lineales consiste en añadir características calculadas mediante una función de similitud, que mide cuánto se parece cada instancia a un punto de referencia concreto.
- Por ejemplo, tomemos el conjunto de datos 1D de antes y añadámosle dos puntos de referencia en $x_1 = -2$ y $x_1 = 1$ (véase el gráfico de la izquierda de la siguiente figura).
- A continuación, definiremos la función de similitud como la RBF gaussiana con $\gamma = 0{,}3$. Se trata de una función en forma de campana que varía de 0 (muy lejos del punto de referencia) a 1 (en el punto de referencia).



# Características de similitud

- Ahora estamos listos para calcular las nuevas características.
- Por ejemplo, veamos la instancia $x_1 = -1$: se encuentra a una distancia de 1 del primer punto de referencia y de 2 del segundo. Por tanto, sus nuevas características son $x_2 = exp(-0{,}3 \times 1^2) \approx 0{,}74$ y $x_3 = exp(-0{,}3 \times 2^2) \approx 0{,}30$. El gráfico de la derecha muestra el conjunto de datos transformado (sin las características originales).
- Como puede ver, ahora es linealmente separable.



# Kernel RBF gaussiano

- ¿Cómo seleccionar los puntos de referencia?
- El enfoque más sencillo consiste en crear un punto de referencia en la ubicación de todos y cada uno de los casos del conjunto de datos. Esto crea muchas dimensiones y aumenta las posibilidades de que el conjunto de entrenamiento transformado sea linealmente separable.
- El método de las características de similitud puede ser útil con cualquier algoritmo de aprendizaje automático, pero calcular todas las características adicionales puede resultar costoso desde el punto de vista informático.
- Una vez más el truco del kernel hace su magia SVM, haciendo posible obtener un resultado similar al que se obtendría si se hubieran añadido muchas características de similitud, pero sin hacerlo realmente.

# Kernels para SVM

- Con tantos kernels para elegir, ¿cómo decidir cuál usar?
- Como regla general, siempre debe probar primero el kernel lineal. La clase LinearSVC es mucho más rápida que SVC(kernel="linear"), especialmente si el conjunto de entrenamiento es muy grande.
- Si no es demasiado grande, también debería probar las SVM kernelizadas, empezando por el kernel RBF gaussiano; suele funcionar realmente bien.
- Si tiene tiempo libre y potencia de cálculo, puede experimentar con otros núcleos utilizando la búsqueda de hiperparámetros. Si hay núcleos especializados para la estructura de datos de su conjunto de entrenamiento, asegúrese de probarlos también.

# Clases de SVM y complejidad computacional

- La clase *LinearSVC* se basa en la biblioteca *liblinear*, que implementa un algoritmo optimizado para SVM lineales.
- No admite el truco del kernel, pero se escala casi linealmente con el número de instancias de entrenamiento y el número de características.
- Su complejidad en tiempo de entrenamiento es aproximadamente $O(m \times n)$.
- El algoritmo tarda más si se requiere una precisión muy alta.
- Esto se controla mediante el hiperparámetro de tolerancia $\epsilon$ (llamado tol en Scikit-Learn).
- En la mayoría de las tareas de clasificación, la tolerancia por defecto está bien.

Chih-Jen Lin et al., "A Dual Coordinate Descent Method for Large-Scale Linear SVM", Proceedings of the 25th International Conference on Machine Learning (2008): 408-415.

# Clases de SVM y complejidad computacional

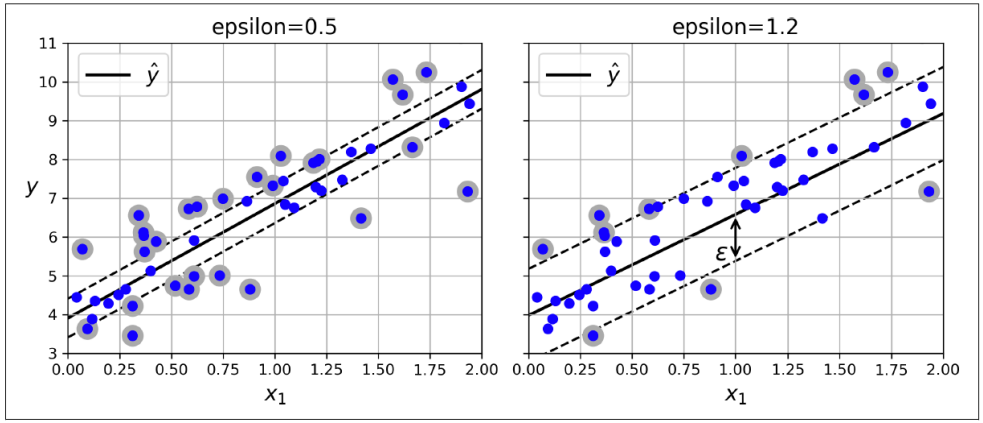
- La clase *SVC* se basa en la biblioteca *libsvm*, que implementa un algoritmo compatible con el truco del kernel.
- La complejidad del tiempo de entrenamiento suele estar entre $O(m^2 \times n)$ y $O(m^3 \times n)$.
- Por desgracia, esto significa que se vuelve terriblemente lento cuando el número de instancias de entrenamiento es grande (por ejemplo, cientos de miles de instancias), por lo que este algoritmo es mejor para conjuntos de entrenamiento no lineales de tamaño pequeño o mediano.
- Se adapta bien al número de características, sobre todo si son dispersas.
- En este caso, el algoritmo se escala aproximadamente con el número medio de características distintas de cero por instancia.

| Class | Time complexity | Out-of-core support | Scaling required | Kernel trick |
|---|---|---|---|---|
| `LinearSVC` | $O(m \times n)$ | No | Yes | No |
| `SVC` | $O(m^2 \times n)$ to $O(m^3 \times n)$ | No | Yes | Yes |
| `SGDClassifier` | $O(m \times n)$ | Yes | Yes | No |

John Platt, " Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines "

# Regresión SVM

- Para utilizar las SVM para la regresión en lugar de la clasificación, el truco consiste en modificar el objetivo: en lugar de intentar ajustar la mayor calle posible entre dos clases limitando al mismo tiempo las violaciones del margen, la regresión SVM intenta ajustar el mayor número posible de instancias en la calle limitando al mismo tiempo las violaciones del margen (es decir, instancias fuera de la calle).
- La anchura de la calle se controla mediante un hiperparámetro, $\epsilon$.
- Esta figura muestra dos modelos de regresión lineal SVM entrenados sobre unos datos lineales, uno con un margen pequeño ($\epsilon = 0.5$) y otro con un margen mayor ($\epsilon = 1.2$).

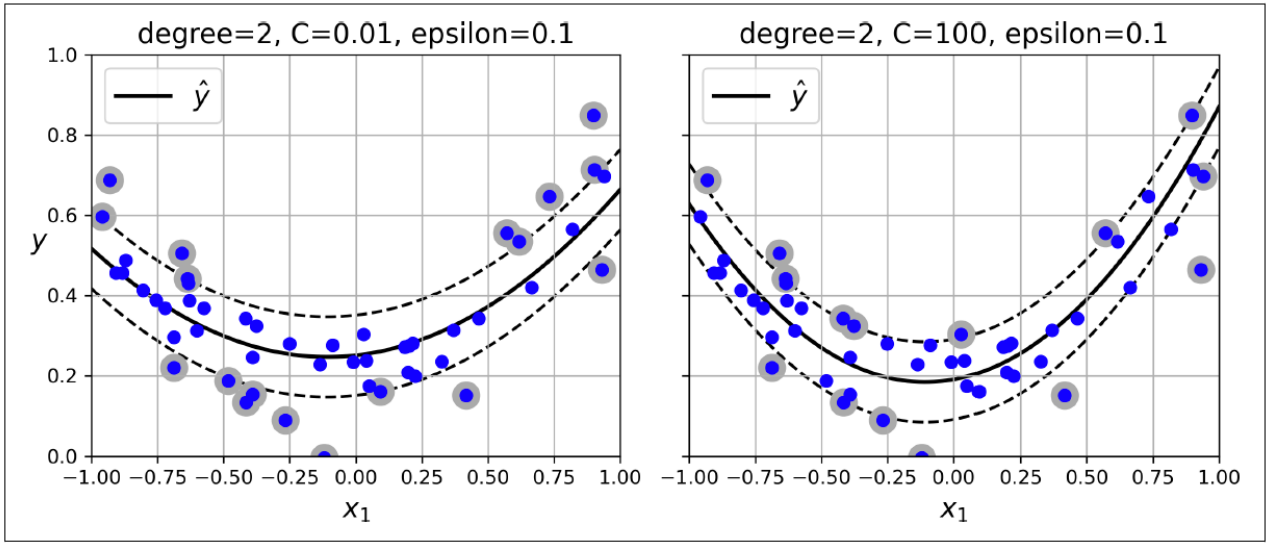


# Regresión SVM

- Reducir $\epsilon$ aumenta el número de vectores de soporte, lo que regulariza el modelo.
- Además, si añades más instancias de entrenamiento dentro del margen, no afectará a las predicciones del modelo; por tanto, se dice que el modelo es $\epsilon$-*insensible*.
- Para abordar tareas de regresión no lineal, puede utilizar un modelo SVM kernelizado.

# Regresión SVM

- La siguiente figura muestra la regresión SVM en un conjunto de entrenamiento cuadrático aleatorio, utilizando un kernel polinómico de segundo grado. Hay algo de regularización en el gráfico de la izquierda (es decir, un valor de C pequeño), y mucho menos en el gráfico de la derecha (es decir, un valor de C grande).



# Funcionamiento de los clasificadores SVM lineales

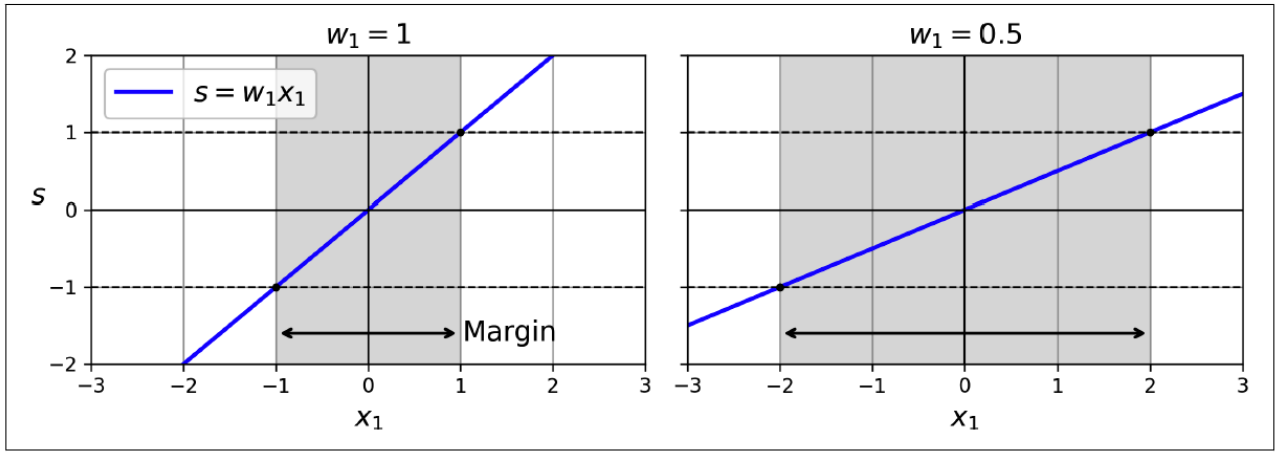
- Un clasificador SVM lineal predice la clase de una nueva instancia **x** calculando primero la función de decisión $\mathbf{\theta}^T x = \theta_0 x_0 + \cdots + \theta_n x_n$ , donde $x_0$ es la característica de sesgo (siempre igual a 1). Si el resultado es positivo, la clase predicha $\hat{y}$ *es* la clase positiva (1); en caso contrario, es la clase negativa (0). Es exactamente igual que *la regresión logística*.
- Hasta ahora, hemos utilizado la convención de poner todos los parámetros del modelo en un vector **θ**, incluyendo el término de sesgo $\theta_0$ y los pesos de las características de entrada $\theta_1$ a $\theta_n$ . Esto requería añadir una entrada de sesgo $x_0 = 1$ a todas las instancias.
- Otra convención muy común es separar el término de sesgo *b* (igual a $\theta_0$ ) y el vector de pesos de características ***w*** (que contiene $\theta_1$ a $\theta_n$ ). En este caso, no es necesario añadir ninguna característica de sesgo a los vectores de características de entrada, y la función de decisión de la SVM lineal es igual a $\mathbf{w}^T x + b = w_1 x_1 + \cdots + w_n x_n + b$.

# Funcionamiento de los clasificadores SVM lineales

- Hacer predicciones con un clasificador SVM lineal es bastante sencillo. ¿Y el entrenamiento?
- Para ello es necesario encontrar el vector de ponderaciones **w** y el término de sesgo *b* que hagan que la calle, o margen, sea lo más amplia posible limitando al mismo tiempo el número de violaciones del margen.
- Empecemos por la anchura de la calle: para hacerla más grande, tenemos que hacer **w** más pequeña.
- Esto puede ser más fácil de visualizar en 2D, como se muestra a continuación.

# Funcionamiento de los clasificadores SVM lineales

- Definamos los márgenes de la calle como los puntos en los que la función de decisión es igual a -1 o +1. En el gráfico de la izquierda, el peso $w_1$ es 1, por lo que los puntos en los que $w_1 x_1 = -1$ o +1 son $x_1 = -1$ y +1: por tanto, el tamaño del margen es 2. En el gráfico de la derecha, el peso es 0,5, por lo que los puntos en los que $w_1 x_1 = -1$ o +1 son $x_1 = -2$ y +2: el tamaño del margen es 4. Por tanto, debemos mantener **w** lo más pequeño posible. Tenga en cuenta que el término de sesgo *b* no influye en el tamaño del margen: modificarlo sólo desplaza el margen, sin afectar a su tamaño.



# Funcionamiento de los clasificadores SVM lineales

- También queremos evitar violaciones del margen, por lo que necesitamos que la función de decisión sea mayor que 1 para todas las instancias de entrenamiento positivas y menor que -1 para las instancias de entrenamiento negativas.
- Si definimos $t^{(i)} = -1$ para instancias negativas (cuando $y^{(i)} = 0$) y $t^{(i)} = 1$ para instancias positivas (cuando $y^{(i)} = 1$), entonces podemos escribir esta restricción como $t^{(i)} (\mathbf{w}^T \mathbf{x}^{(i)} + b) \geq 1$ para todas las instancias.
- Por lo tanto, podemos expresar el objetivo del clasificador lineal SVM de margen duro como el problema de optimización restringido en la siguiente ecuación:

$$\underset{\mathbf{w}, b}{\text{minimize}} \quad \frac{1}{2}\mathbf{w}^T\mathbf{w}$$

$$\text{subject to} \quad t^{(i)}\left(\mathbf{w}^T\mathbf{x}^{(i)} + b\right) \geq 1 \quad \text{for } i = 1, 2, \cdots, m$$

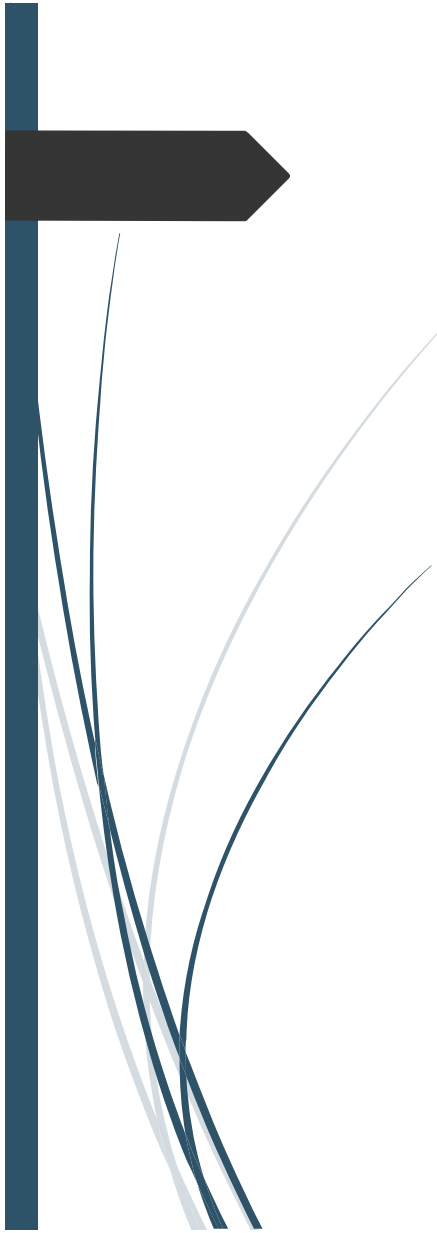
Estamos minimizando ½ $\mathbf{w}^T \mathbf{w}$, que es igual a ½$\| \mathbf{w} \|^2$ , en lugar de minimizar $\| \mathbf{w} \|$ (la norma de $\mathbf{w}$). De hecho, ½$\| \mathbf{w} \|^2$ tiene una derivada bonita y sencilla (es sólo $\mathbf{w}$), mientras que $\| \mathbf{w} \|$ no es diferenciable en $\mathbf{w} = 0$. Los algoritmos de optimización suelen funcionar mucho mejor con funciones diferenciables.

# Funcionamiento de los clasificadores SVM lineales

- Para obtener el objetivo de margen suave, necesitamos introducir una variable de holgura $\zeta^{(i)} \geq 0$ para cada instancia: $\zeta^{(i)}$ mide cuánto se permite que la instancia $i^{th}$ viole el margen.
- Ahora tenemos dos objetivos en conflicto: hacer que las variables de holgura sean lo más pequeñas posible para reducir las violaciones del margen, y hacer que ½ $\mathbf{w}^T$ **w sea** lo más pequeño posible para aumentar el margen. Aquí es donde entra en juego el hiperparámetro C: nos permite definir el equilibrio entre estos dos objetivos. Esto nos da el problema de optimización restringida de la siguiente ecuación:

$$\underset{\mathbf{w},\, b,\, \zeta}{\text{minimize}} \quad \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{m}\zeta^{(i)}$$

$$\text{subject to} \quad t^{(i)}\left(\mathbf{w}^T\mathbf{x}^{(i)} + b\right) \geq 1 - \zeta^{(i)} \quad \text{and} \quad \zeta^{(i)} \geq 0 \quad \text{for } i = 1, 2, \cdots, m$$

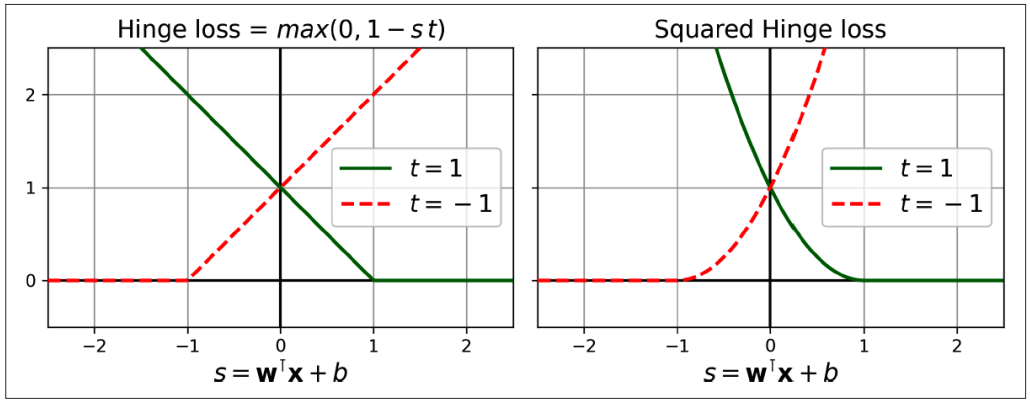


# Funcionamiento de los clasificadores SVM lineales

- Tanto el problema del margen duro como el del margen blando son problemas de optimización cuadrática convexa con restricciones lineales.
- Estos problemas se conocen como problemas de *programación cuadrática (*QP).
- Hay muchos solucionadores disponibles para resolver problemas QP utilizando diversas técnicas (no dedicamos tiempo a esto en esta asignatura).
- Utilizar un solucionador QP es una forma de entrenar una SVM. Otra es utilizar el descenso de gradiente para minimizar el *hinge loss* o el *squared hinge los,* en la siguiente diapositiva.

# Funcionamiento de los clasificadores SVM lineales

- Dada una instancia **x** de la clase positiva (es decir, con t = 1), la pérdida es 0 si la salida *s* de la función de decisión ($s = \mathbf{w}^T \mathbf{x} + b$) es mayor o igual que 1.
- Dada una instancia de la clase negativa (es decir, con t = -1), la pérdida es 0 si $s \leq -1$.
- Cuanto más alejada esté una instancia del lado correcto del margen, mayor será la pérdida: crece linealmente para la hinge loss y cuadráticamente para la squared hinge loss. Esto hace que la hinge loss al cuadrado sea más sensible a los valores atípicos. Sin embargo, si el conjunto de datos está limpio, tiende a converger más rápidamente.



# El doble problema

- Dado un problema de optimización con restricciones, conocido como *problema primario*, es posible expresar un problema diferente pero estrechamente relacionado, llamado su *problema dual*.
- La solución del problema dual suele dar una cota inferior a la solución del problema primario, pero en algunas condiciones puede tener la misma solución que el problema primario.
- Afortunadamente, el problema SVM pasa a cumplir estas condiciones (la función objetivo es convexa, y las restricciones de desigualdad son continuamente diferenciables y optimización convexa), por lo que puede optar por resolver el problema primal o el problema dual; ambos tendrán la misma solución.
- Forma dual del objetivo SVM lineal:

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

$$\text{subject to } \alpha^{(i)} \geq 0 \text{ for all } i = 1, 2, \ldots, m \text{ and } \sum_{i=1}^{m}\alpha^{(i)}t^{(i)} = 0$$

# El doble problema

- Una vez que encuentre el vector $\widehat{\boldsymbol{\alpha}}$ que minimiza esta ecuación (utilizando un solucionador QP), utilice la siguiente ecuación para calcular el $\widehat{\mathbf{w}}$ y $\widehat{b}$ que minimizan el problema primal. En esta ecuación, $n_s$ representa el número de vectores de soporte.

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

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

- El problema dual es más rápido de resolver que el primal cuando el número de instancias de entrenamiento es menor que el número de características. Y lo que es más importante, el problema dual hace posible el truco del kernel, mientras que el problema primario no.

# SVM kernelizadas

- Supongamos que desea aplicar una transformación polinómica de segundo grado a un conjunto de entrenamiento bidimensional y, a continuación, entrenar un clasificador SVM lineal en el conjunto de entrenamiento transformado.
- Esta ecuación muestra la función cartográfica polinómica de segundo grado φ que se desea aplicar.

$$\varphi(\mathbf{x}) = \varphi\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right) = \begin{pmatrix} {x_1}^2 \\ \sqrt{2}\, x_1 x_2 \\ {x_2}^2 \end{pmatrix}$$

- Observe que el vector transformado es 3D en lugar de 2D.

# SVM kernelizadas

- Veamos ahora qué ocurre con un par de vectores 2D, **a** y **b**, si aplicamos este polinomio de segundo grado y calculamos el producto escalar de los vectores transformados.

$$\varphi(\mathbf{a})^{\mathsf{T}}\varphi(\mathbf{b}) = \begin{pmatrix} a_1^2 \\ \sqrt{2}\,a_1 a_2 \\ a_2^2 \end{pmatrix}^{\mathsf{T}} \begin{pmatrix} b_1^2 \\ \sqrt{2}\,b_1 b_2 \\ b_2^2 \end{pmatrix} = a_1^2 b_1^2 + 2a_1 b_1 a_2 b_2 + a_2^2 b_2^2$$

$$= (a_1 b_1 + a_2 b_2)^2 = \left(\begin{pmatrix} a_1 \\ a_2 \end{pmatrix}^{\mathsf{T}} \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}\right)^2 = (\mathbf{a}^{\mathsf{T}}\mathbf{b})^2$$

- El producto punto de los vectores transformados es igual al cuadrado del producto punto de los vectores originales: $\varphi(\mathbf{a})^{\mathsf{T}}\,\varphi(\mathbf{b}) = (\mathbf{a}^{\mathsf{T}}\,\mathbf{b})^2$.
- Aquí está la clave: si aplica la transformación $\varphi$ a todas las instancias de entrenamiento, entonces el problema dual contendrá el producto punto $\varphi(\mathbf{x}^{(i)})^{\mathsf{T}}\,\varphi(\mathbf{x}^{(j)})$. Pero si $\varphi$ es la transformación polinómica de segundo grado, entonces se puede sustituir este producto punto de vectores transformados simplemente por $(\mathbf{x}^{(i)\mathsf{T}}\,\mathbf{x}^{(j)})^2$.

# SVM kernelizadas

- No es necesario transformar las instancias de entrenamiento; basta con sustituir el producto punto por su cuadrado. El resultado será estrictamente el mismo que si hubiera pasado por la molestia de transformar el conjunto de entrenamiento y luego ajustar un algoritmo SVM lineal, pero este truco hace que todo el proceso sea mucho más eficiente computacionalmente.
- La función $K(\boldsymbol{a}, \boldsymbol{b}) = (\boldsymbol{a}^T \boldsymbol{b})^2$ es un núcleo polinómico de segundo grado. En aprendizaje automático, un *núcleo* es una función capaz de calcular el producto punto $\varphi(\mathbf{a})^T \varphi(\mathbf{b})$, basándose únicamente en los vectores originales **a** y **b**, sin tener que calcular (o incluso conocer) la transformación $\varphi$.
- Kernels más utilizados.

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

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

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

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

# SVM kernelizadas

- Mostramos cómo pasar de la solución dual a la solución primaria en el caso de un clasificador SVM lineal. Pero si aplicamos el truco del kernel, se termina con ecuaciones que incluyen $\varphi(x^{(i)})$. De hecho, $\hat{\mathbf{w}}$ debe tener el mismo número de dimensiones que $\varphi(x^{(i)})$, que puede ser enorme o incluso infinito, por lo que no se puede calcular.
- Pero, ¿cómo se pueden hacer predicciones sin conocer $\hat{\mathbf{w}}$? Bueno, la buena noticia es que puedes introducir la fórmula para $\hat{\mathbf{w}}$ de la ecuación anterior en la función de decisión para una nueva instancia $\mathbf{x}^{(n)}$ , y obtendrás una ecuación con sólo productos escalares entre vectores de entrada. Esto permite utilizar el truco del kernel.

$$h_{\hat{\mathbf{w}},\hat{b}}\left(\varphi\left(\mathbf{x}^{(n)}\right)\right) = \hat{\mathbf{w}}^{\mathsf{T}}\varphi\left(\mathbf{x}^{(n)}\right) + \hat{b} = \left(\sum_{i=1}^{m} \hat{\alpha}^{(i)} t^{(i)} \varphi\left(\mathbf{x}^{(i)}\right)\right)^{\mathsf{T}} \varphi\left(\mathbf{x}^{(n)}\right) + \hat{b}$$

$$= \sum_{i=1}^{m} \hat{\alpha}^{(i)} t^{(i)} \left(\varphi\left(\mathbf{x}^{(i)}\right)^{\mathsf{T}} \varphi\left(\mathbf{x}^{(n)}\right)\right) + \hat{b}$$

$$= \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^{m} \hat{\alpha}^{(i)} t^{(i)} K\left(\mathbf{x}^{(i)}, \mathbf{x}^{(n)}\right) + \hat{b}$$

# SVM kernelizadas

- Tenga en cuenta que como $\boldsymbol{\alpha}^{(i)} \neq 0$ sólo para los vectores de soporte, hacer predicciones implica calcular el producto punto del nuevo vector de entrada $\mathbf{x}^{(n)}$ sólo con los vectores de soporte, no con todas las instancias de entrenamiento. Por supuesto, es necesario utilizar el mismo truco para calcular el término de sesgo *b*:

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

$$= \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^{m} \left( t^{(i)} - \sum_{\substack{j=1 \\ \hat{\alpha}^{(j)} > 0}}^{m} \hat{\alpha}^{(j)} t^{(j)} K\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right) \right)$$

# Teorema de Mercer

- Según el *teorema de Mercer*, si una función $K(\mathbf{a}, \mathbf{b})$ respeta unas condiciones matemáticas llamadas *condiciones de Mercer* (por ejemplo, $K$ debe ser continua y simétrica en sus argumentos de modo que $K(\mathbf{a}, \mathbf{b}) = K(\mathbf{b}, \mathbf{a})$, etc.), entonces existe una función $\varphi$ que mapea $\mathbf{a}$ y $\mathbf{b}$ en otro espacio (posiblemente con dimensiones mucho mayores) tal que $K(\mathbf{a}, \mathbf{b}) = \varphi(\mathbf{a})^T \varphi(\mathbf{b})$.
- Puedes usar $K$ como kernel porque sabes que $\varphi$ existe, aunque no sepas qué es $\varphi$. En el caso del núcleo RBF gaussiano, se puede demostrar que $\varphi$ asigna cada instancia de entrenamiento a un espacio de dimensión infinita, por lo que es bueno no tener que realizar la asignación.
- Obsérvese que algunos kernels utilizados con frecuencia (como el kernel sigmoide) no respetan todas las condiciones de Mercer, aunque suelen funcionar bien en la práctica.

# Strengths and Weaknesses of SVM

- Strengths
  - Training is relatively easy
    - No local optimal, unlike in neural networks
  - It scales relatively well to high dimensional data
  - Tradeoff between classifier complexity and error can be controlled explicitly
  - Non-traditional data like strings and trees can be used as input to SVM, instead of feature vectors
- Weaknesses
  - Need to choose a “good” kernel function.

# Other Types of Kernel Methods

- A lesson learnt in SVM: a linear algorithm in the feature space is equivalent to a non-linear algorithm in the input space
- Standard linear algorithms can be generalized to its non-linear version by going to the feature space
  - Kernel principal component analysis, kernel independent component analysis, kernel canonical correlation analysis, kernel k-means, 1-class SVM are some examples

# Other Types of SVMs

- **Twin support vector machine**

TWSVM: This type of SVM is again a binary classifier that aim's to have two hyperplanes to distinguish data. There is a little difference here that using two hyperplanes would change some facts about SVM. The difference here is we aren't going to find the biggest margin. The goal here is to find two hyperplanes that try to fit with each data, meaning each hyperplane is going to have the closest relationship with each class (Think of a regression line for each class). At last, there is a voting process that tries to classify the test data into the closest hyperplane.

- **K-SVCR**

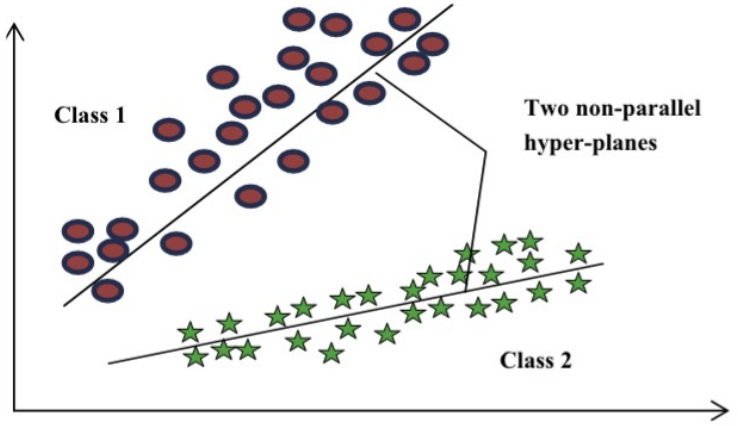
This type of support vector machine is suited for more than two classes classification. It is a mixed type of classification and regression model that uses the method called one-vs-one-vs-rest. To get a better understanding of how this model works we can say, this model at a time evaluates every two classes and tries to maximize the margin between two classes. This type of SVM is closer to the original SVM than TWSVM and the computational complexity is more than the original SVM because we have added the evaluation of other classes in the time of classification.

- **LSTKSVC**

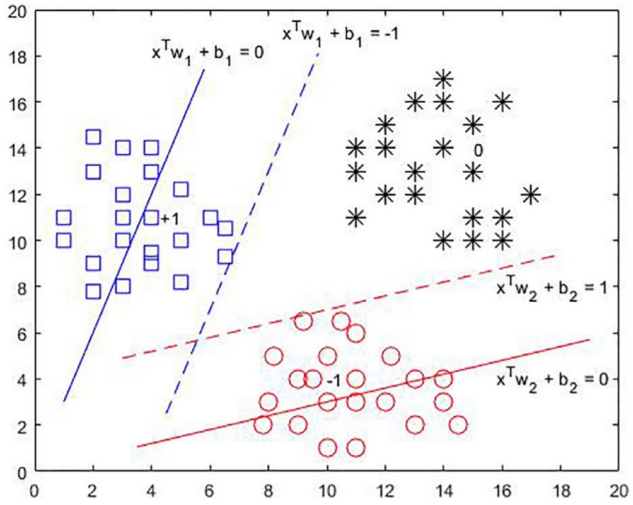
The last type of SVM we're going to introduce is named Least squared twin multi-class support vector machine. This type of SVM is a mixture of the two SVM we introduced above. To find the optimal solution it creates two hyperplanes and uses the method one-vs-one-vs-rest.

# Other Types of SVMs

- **Twin support vector machine**

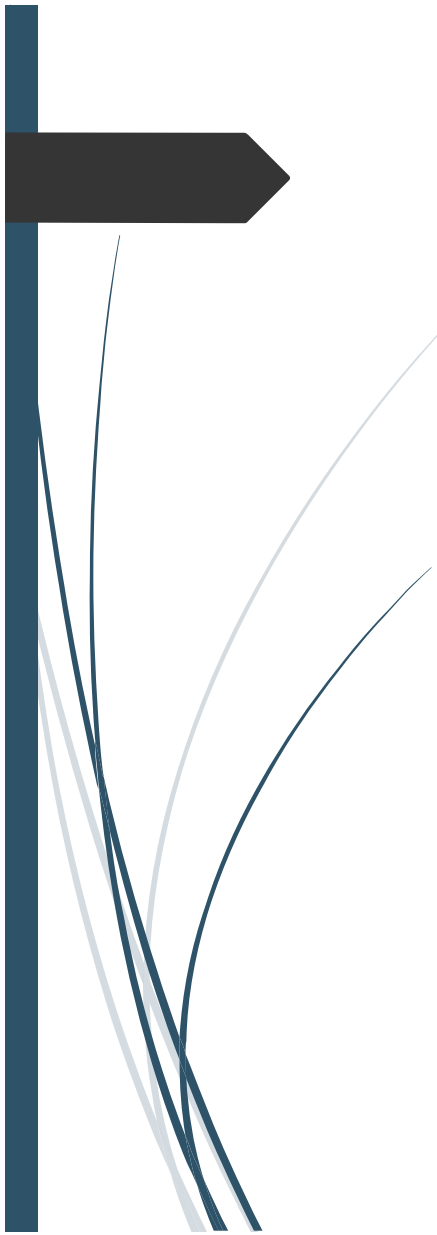



- **LSTKSVC**



Minería de Datos: Preprocesamiento y clasificación

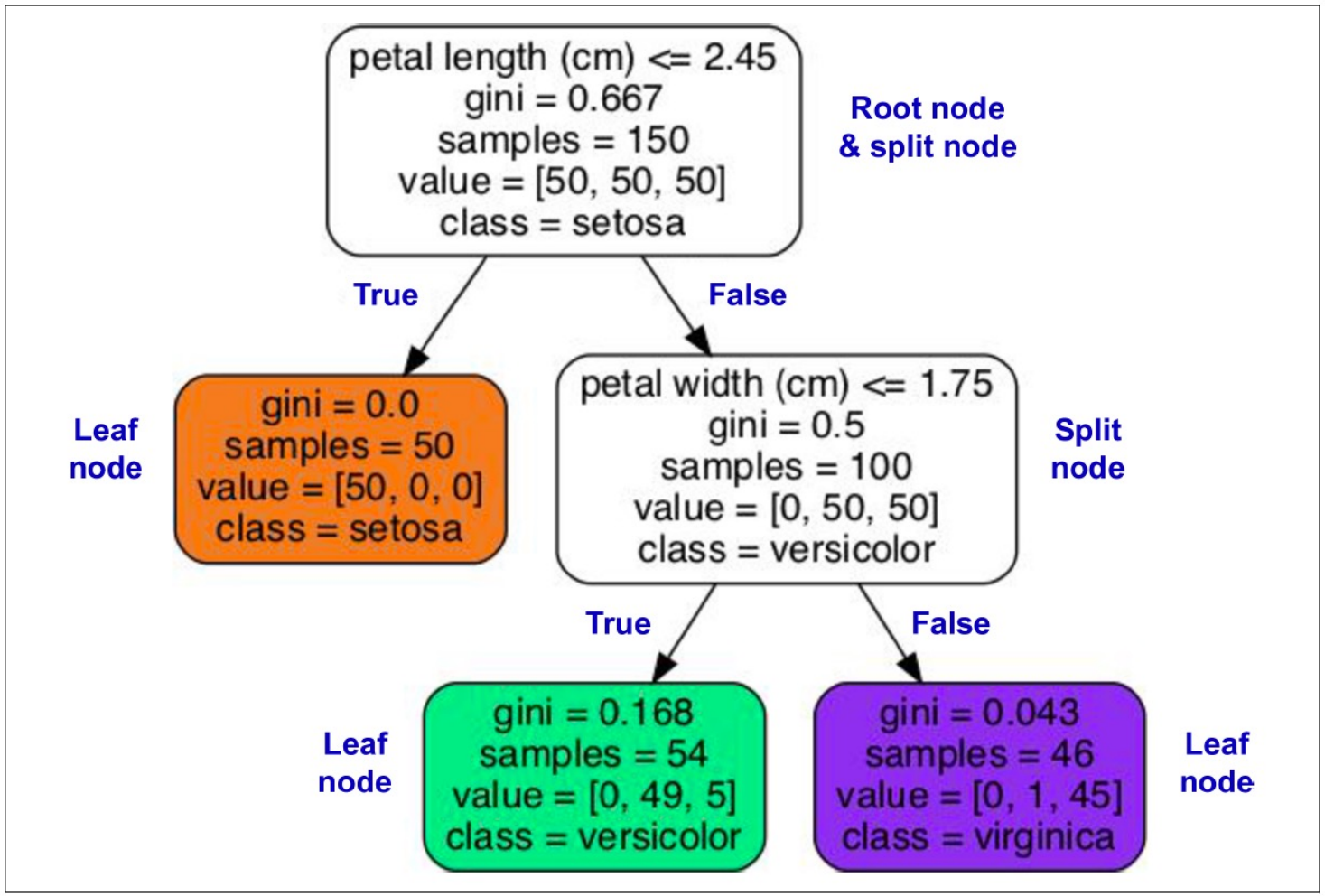
# Tree-BASED METHODS



# Árboles de decisión

- *Los árboles de decisión* son algoritmos versátiles de aprendizaje automático que pueden realizar tareas tanto de clasificación como de regresión.
- Son algoritmos potentes, capaces de ajustar conjuntos de datos complejos.
- Los árboles de decisión son también los componentes fundamentales de los *random forests*, que se encuentran entre los algoritmos de aprendizaje automático más potentes de la actualidad.
- Aquí, empezaremos discutiendo cómo entrenar, visualizar y hacer predicciones con árboles de decisión. A continuación, repasaremos el algoritmo de entrenamiento CART utilizado por Scikit-Learn, y exploraremos cómo regularizar árboles y utilizarlos para tareas de regresión. Por último, discutiremos algunas de las limitaciones de los árboles de decisión.

# Árboles de decisión



# Predicciones

- Veamos cómo hace predicciones el árbol representado en la figura anterior. Supongamos que encuentra una flor de iris y quiere clasificarla en función de sus pétalos. Se empieza por el *nodo raíz* (profundidad 0, en la parte superior): este nodo pregunta si la longitud de los pétalos de la flor es inferior a 2,45 cm. En caso afirmativo, se desciende hasta el nodo hijo izquierdo de la raíz (profundidad 1, izquierda). En este caso, es un *nodo hoja* (es decir, no tiene ningún nodo hijo), por lo que no hace ninguna pregunta: simplemente mire la clase predicha para ese nodo, y el árbol de decisión predice que su flor es una *Iris setosa*.
- Suponga ahora que encuentra otra flor, y que esta vez la longitud del pétalo es superior a 2,45 cm. Comienza de nuevo en la raíz, pero ahora desciende hasta su nodo hijo derecho (profundidad 1, derecha). Este no es un nodo hoja, es un *nodo partido*, por lo que hace otra pregunta: ¿la anchura del pétalo es menor de 1,75 cm? Si es así, lo más probable es que su flor sea un Iris versicolor (profundidad 2, izquierda). En caso contrario, es probable que se trate de un *Iris virginica* (profundidad 2, derecha). Así de sencillo.

# Predicciones

- El atributo *samples de* un nodo cuenta a cuántas instancias de entrenamiento se aplica. Por ejemplo, 100 instancias de entrenamiento tienen una longitud de pétalo superior a 2,45 cm (profundidad 1, derecha), y de esas 100, 54 tienen una anchura de pétalo inferior a 1,75 cm (profundidad 2, izquierda).
- El atributo *value* de un nodo te indica a cuántas instancias de entrenamiento de cada clase se aplica este nodo: por ejemplo, el nodo inferior derecho se aplica a 0 *Iris setosa*, 1 *Iris versicolor* y 45 *Iris virginica*.
- El atributo *gini* de un nodo mide su *impureza de Gini*: un nodo es "puro" (*gini=0*) si todas las instancias de entrenamiento a las que se aplica pertenecen a la misma clase. Por ejemplo, dado que el nodo izquierdo de profundidad 1 sólo se aplica a las instancias de entrenamiento de Iris setosa, es puro y su impureza de Gini es 0.

# Predicciones

- La siguiente ecuación muestra cómo el algoritmo de entrenamiento calcula la impureza de Gini $G_i$ del nodo $i^{th}$ . El nodo izquierdo de profundidad 2 tiene una impureza de Gini igual a $1 - (0/54)^2 - (49/54)^2 - (5/54)^2 \approx 0{,}168$.

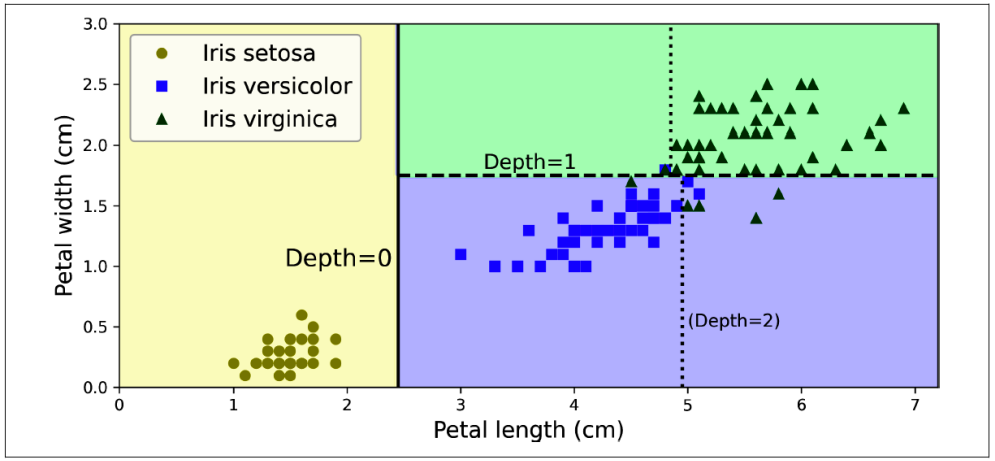
$$G_i = 1 - \sum_{k=1}^{n} p_{i,k}^2$$

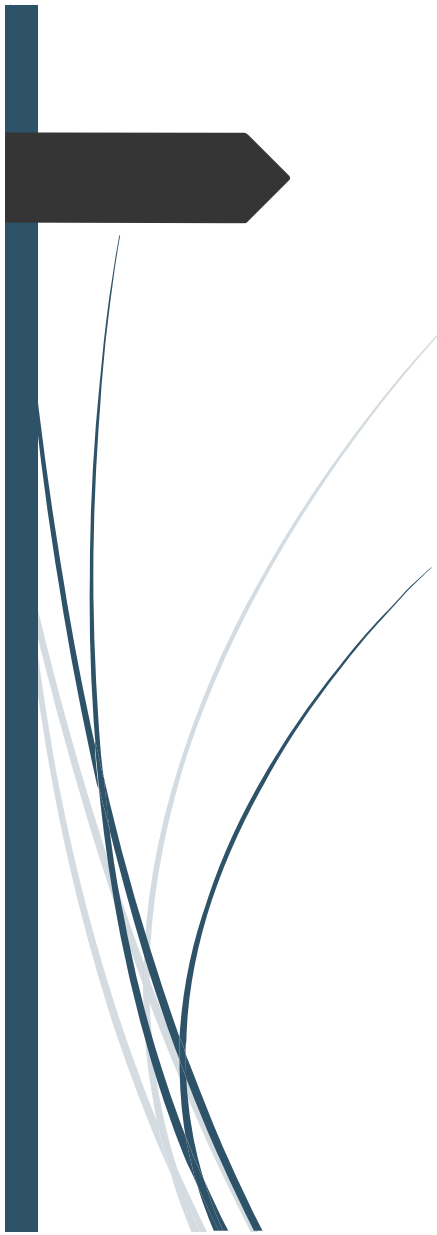
- $G_i$ es la impureza Gini del nodo $i^{th}$ .
- $p_{i,k}$ es la proporción de instancias de clase $k$ entre las instancias de entrenamiento en el nodo $i^{th}$ .

- Scikit-Learn utiliza el algoritmo CART, que sólo produce árboles binarios, es decir, árboles en los que los nodos divididos siempre tienen exactamente dos hijos. Sin embargo, otros algoritmos, como ID3, C4.5, See5; pueden producir árboles de decisión con nodos que tienen más de dos hijos.

# Predicciones

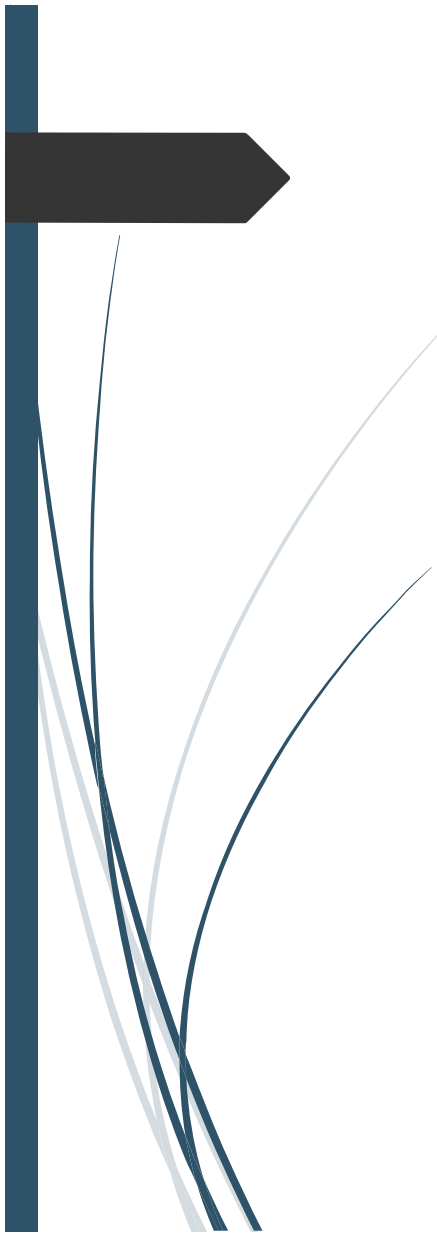
- Esta figura muestra los límites de decisión de este árbol de decisión. La línea vertical gruesa representa el límite de decisión del nodo raíz (profundidad 0): longitud del pétalo = 2,45 cm. Como la zona izquierda es pura (sólo *Iris setosa*), no puede dividirse más. Sin embargo, la zona derecha es impura, por lo que el nodo derecho de profundidad 1 la divide a una anchura de pétalo = 1,75 cm (representada por la línea discontinua). Como *max_depth* se ha establecido en 2, el árbol de decisión se detiene justo ahí. Si fijara *max_depth en* 3, entonces los dos nodos de profundidad-2 añadirían cada uno otro límite de decisión (representado por las dos líneas punteadas verticales).





# Interpretación de modelos: Caja blanca frente a caja negra

- Los árboles de decisión son intuitivos y sus decisiones son fáciles de interpretar.
- Estos modelos suelen denominarse modelos de caja blanca. En cambio, como verás, los random forests y las redes neuronales suelen considerarse modelos de caja negra. Hacen grandes predicciones, y puedes comprobar fácilmente los cálculos que realizaron para hacer estas predicciones; sin embargo, suele ser difícil explicar en términos sencillos por qué se hicieron las predicciones.
- Por ejemplo, si una red neuronal dice que una persona concreta aparece en una foto, es difícil saber qué contribuyó a esta predicción: ¿Reconoció el modelo los ojos de esa persona? ¿la boca? ¿la nariz? ¿Sus zapatos? ¿O incluso el sofá en el que estaba sentada?
- Por el contrario, los árboles de decisión proporcionan reglas de clasificación sencillas y agradables que incluso pueden aplicarse manualmente si es necesario (por ejemplo, para la clasificación de flores). El campo del ML interpretable tiene como objetivo crear sistemas de ML que puedan explicar sus decisiones de forma comprensible para los humanos. Esto es importante en muchos ámbitos, por ejemplo, para garantizar que el sistema no tome decisiones injustas.



# Estimación de las probabilidades de clase

- Un árbol de decisión también puede estimar la probabilidad de que una instancia pertenezca a una determinada clase *k*.
- En primer lugar, recorre el árbol para encontrar el nodo hoja de esta instancia y, a continuación, devuelve la proporción de instancias de entrenamiento de la clase *k* en este nodo.

# Algoritmo de entrenamiento CART

- Scikit-Learn utiliza el algoritmo *Classification and Regression Tree* (CART) para entrenar árboles de decisión. El algoritmo funciona dividiendo primero el conjunto de entrenamiento en dos subconjuntos utilizando una única característica $k$ y un umbral $t_k$ (por ejemplo, " petal length ≤ 2,45 cm").
- ¿Cómo elige $k$ y $t_k$ ? Busca el par $(k, t_k)$ que produce los subconjuntos más puros, ponderados por su tamaño. La siguiente ecuación da la función de coste que el algoritmo intenta minimizar:

$$J(k, t_k) = \frac{m_{\text{left}}}{m} G_{\text{left}} + \frac{m_{\text{right}}}{m} G_{\text{right}}$$

$$\text{where} \begin{cases} G_{\text{left/right}} \text{ measures the impurity of the left/right subset} \\ m_{\text{left/right}} \text{ is the number of instances in the left/right subset} \end{cases}$$

# Algoritmo de entrenamiento CART

- Una vez que el algoritmo CART ha dividido con éxito el conjunto de entrenamiento en dos, divide los subconjuntos utilizando la misma lógica, luego los subsubconjuntos, y así sucesivamente, de forma recursiva.
- Deja de buscar una vez que alcanza la profundidad máxima (definida por el hiperparámetro *max_depth*), o si no puede encontrar una división que reduzca la impureza.
- Algunos otros hiperparámetros controlan condiciones de parada adicionales: *min_samples_split, min_samples_leaf, min_weight_fraction_leaf y max_leaf_nodes*.
- El algoritmo CART es un *algoritmo greedy*: no comprueba si la división conducirá o no a la impureza más baja posible varios niveles más abajo. Un algoritmo greedy suele producir una solución razonablemente buena, pero no garantiza que sea la óptima.
- Por desgracia, se sabe que encontrar el árbol óptimo es un problema *NP-completo*. Requiere un tiempo $O(\exp(m))$, lo que hace que el problema sea intratable incluso para conjuntos de entrenamiento pequeños.

# Complejidad computacional

- Para hacer predicciones hay que recorrer el árbol de decisión desde la raíz hasta una hoja. Los árboles de decisión suelen estar aproximadamente equilibrados, por lo que recorrerlos requiere pasar por aproximadamente $O(\log_2(m))$ nodos.
- Dado que cada nodo sólo requiere comprobar el valor de una característica, la complejidad global de la predicción es $O(\log_2(m))$, independientemente del número de características.
- Las predicciones son muy rápidas, incluso cuando se trata de grandes conjuntos de entrenamiento.
- El algoritmo de entrenamiento compara todas las características (o menos si se establece *max_features*) en todas las muestras de cada nodo. La comparación de todas las características en todas las muestras de cada nodo da como resultado una complejidad de entrenamiento de $O(n \times m \log_2(m))$.

# ¿Impureza o entropía de Gini?

- Por defecto, la clase *DecisionTreeClassifier* utiliza la medida de impureza de Gini, pero puede seleccionar la medida de impureza *de entropía* en su lugar estableciendo el hiperparámetro de *criterion* en "entropy".
- El concepto de entropía se originó en la termodinámica como medida del desorden molecular: la entropía se aproxima a cero cuando las moléculas están quietas y bien ordenadas. Posteriormente, la entropía se extendió a una gran variedad de ámbitos, incluida la teoría de la información de Shannon, donde mide el contenido medio de información de un mensaje. La entropía es cero cuando todos los mensajes son idénticos.
- En el aprendizaje automático, la entropía se utiliza con frecuencia como medida de impureza: la entropía de un conjunto es cero cuando contiene instancias de una sola clase.

# ¿Impureza o entropía de Gini?

- La siguiente ecuación muestra la definición de la entropía del nodo $i^{th}$ . Por ejemplo, el nodo izquierdo de profundidad 2 de la figura anterior tiene una entropía igual a -(49/54) $\log_2$ (49/54) - (5/54) $\log_2$ (5/54) ≈ 0,445.

$$H_i = - \sum_{\substack{k=1 \\ p_{i,k} \neq 0}}^{n} p_{i,k} \log_2 \left(p_{i,k}\right)$$

- ¿Hay que utilizar la impureza de Gini o la entropía?
  - La verdad es que la mayoría de las veces no hay gran diferencia: conducen a árboles similares.
  - La impureza de Gini es ligeramente más rápida de calcular, por lo que es una buena opción por defecto.
  - Sin embargo, cuando difieren, la impureza de Gini tiende a aislar la clase más frecuente en su propia rama del árbol, mientras que la entropía tiende a producir árboles ligeramente más equilibrados.

# Hiperparámetros de regularización

- Los árboles de decisión hacen muy pocas suposiciones sobre los datos de entrenamiento.
- Si no se impone ninguna restricción, la estructura del árbol se adaptará a los datos de entrenamiento y se ajustará mucho a ellos; de hecho, lo más probable es que se sobreajuste.
- Un modelo de este tipo suele denominarse *modelo no paramétrico*, porque el número de parámetros no se determina antes del entrenamiento, por lo que la estructura del modelo es libre de ceñirse a los datos. En cambio, un *modelo paramétrico*, como un modelo lineal, tiene un número predeterminado de parámetros, por lo que su grado de libertad es limitado, lo que reduce el riesgo de sobreajuste (pero aumenta el riesgo de infraajuste).
- Para evitar el sobreajuste de los datos de entrenamiento, es necesario restringir la libertad del árbol de decisión durante el entrenamiento.
- Los hiperparámetros de regularización dependen del algoritmo utilizado, pero en general se puede al menos restringir la profundidad máxima del árbol de decisión.
- En Scikit-Learn, esto se controla mediante el hiperparámetro *max_depth*. El valor por defecto es None, que significa ilimitado. Reducir la max_depth regularizará el modelo y reducirá el riesgo de sobreajuste.

# Hiperparámetros de regularización

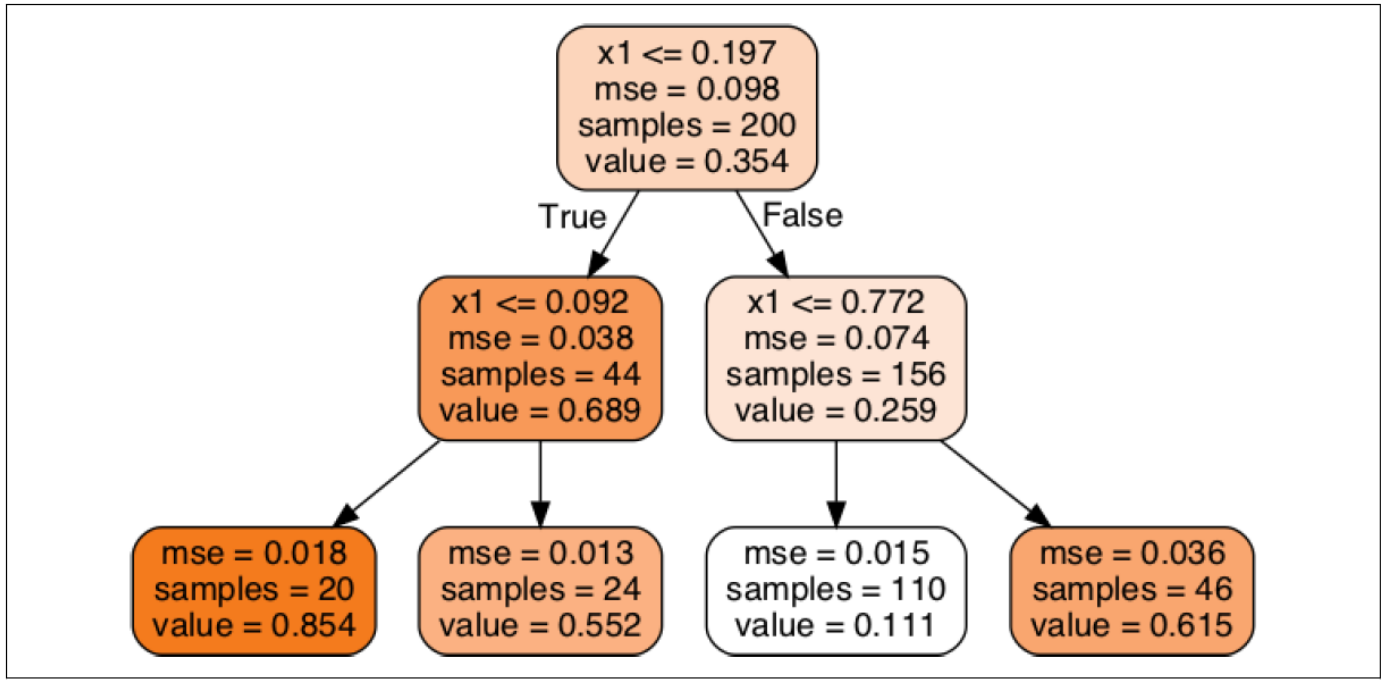
La clase *DecisionTreeClassifier* tiene algunos otros parámetros que restringen de forma similar la forma del árbol de decisión:

- *max_features*
  - Número máximo de características que se evalúan para la división en cada nodo
- *max_leaf_nodes*
  - Número máximo de nodos hoja
- *min_samples_split*
  - Número mínimo de muestras que debe tener un nodo para poder dividirse
- *min_samples_leaf*
  - Número mínimo de muestras que debe tener un nodo hoja para ser creado
- *min_weight_fraction_leaf*
  - Igual que min_samples_leaf pero expresado como fracción del número total de instancias ponderadas

Aumentar los hiperparámetros *min_** o reducir los hiperparámetros *max_** regularizará el modelo.

# Regresión

- Los árboles de decisión también son capaces de realizar tareas de regresión.

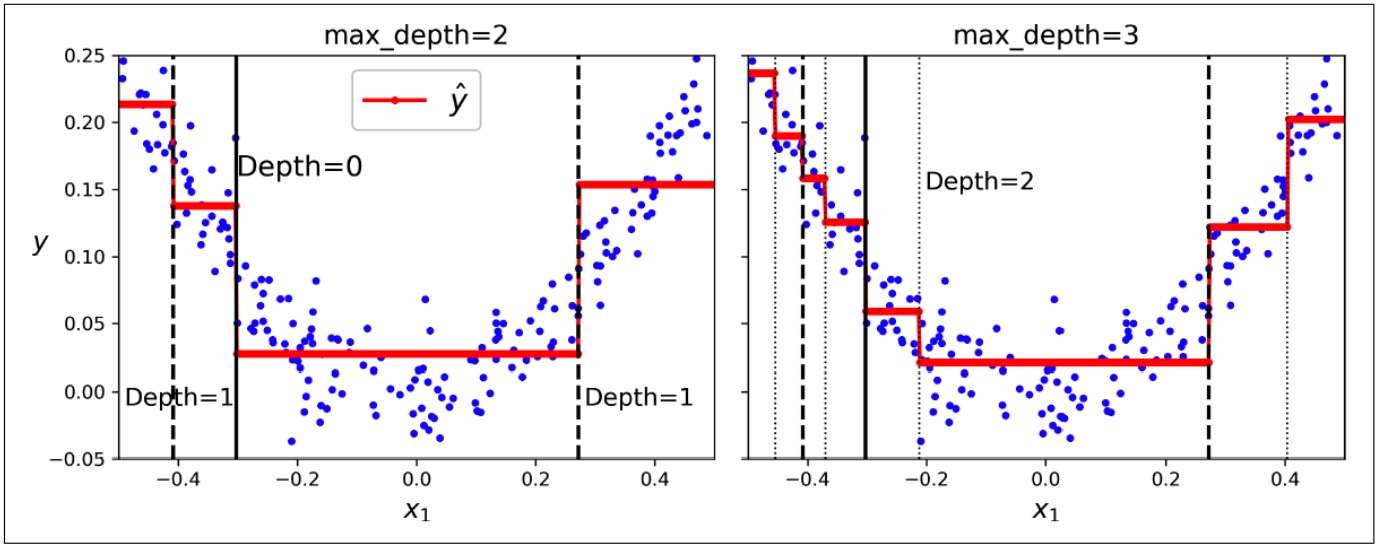


# Regresión

- Este árbol se parece mucho al árbol de clasificación que construyó anteriormente.
- La principal diferencia es que en lugar de predecir una clase en cada nodo, predice un valor.
- Por ejemplo, supongamos que desea hacer una predicción para una nueva instancia con $x_1 = 0,2$. El nodo raíz pregunta si $x_1 \leq 0,197$. Si no lo es, el algoritmo pasa al nodo hijo de la *derecha,* que pregunta si $x_1 \leq 0,772$. Si lo es, el algoritmo pasa al nodo hijo de la *derecha, que pregunta si* $x \leq 0,772$. Si lo es, el algoritmo pasa al nodo hijo izquierdo. Éste es un nodo hoja y predice *el valor = 0,111*.
- Esta predicción es el valor objetivo medio de las 110 instancias de entrenamiento asociadas a este nodo hoja, y da como resultado un error cuadrático medio igual a 0,015 sobre estas 110 instancias.

# Regresión

- Las predicciones de este modelo se representan en la siguiente figura.
- Observe cómo el valor predicho para cada región es siempre el valor objetivo medio de las instancias de esa región. El algoritmo divide cada región de forma que la mayoría de las instancias de entrenamiento se acerquen lo máximo posible a ese valor predicho.



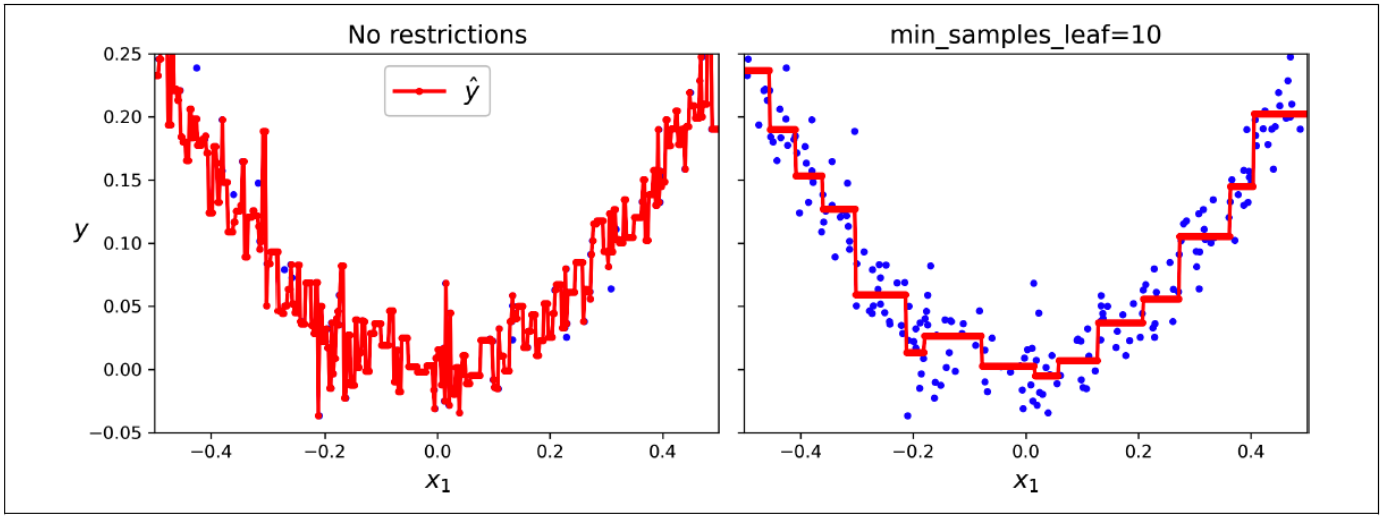
# Regresión

- El algoritmo CART trabaja dividiendo el conjunto de entrenamiento de forma que se minimice el MSE. La siguiente ecuación muestra la función de coste que el algoritmo intenta minimizar.

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

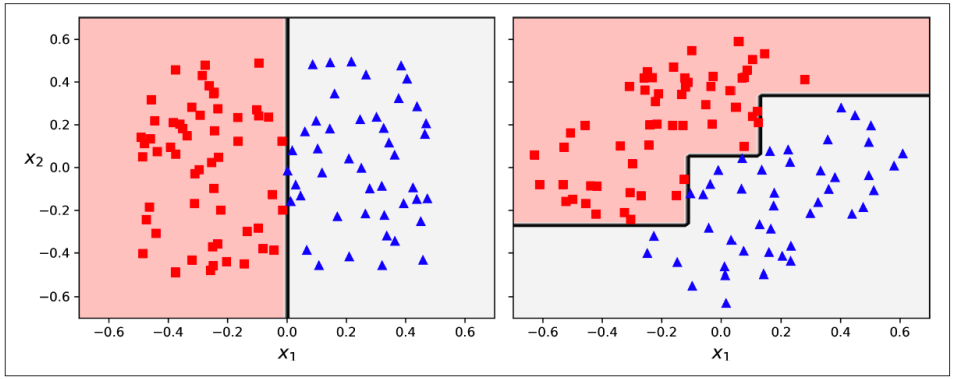
# Regresión

- Al igual que en las tareas de clasificación, los árboles de decisión son propensos al sobreajuste cuando se trata de tareas de regresión.
- Sin ninguna regularización (es decir, utilizando los hiperparámetros por defecto), se obtienen las predicciones de la izquierda.
- Obviamente, estas predicciones se ajustan en exceso al conjunto de entrenamiento. Basta con establecer *min_samples_leaf=10 para* obtener un modelo mucho más razonable.



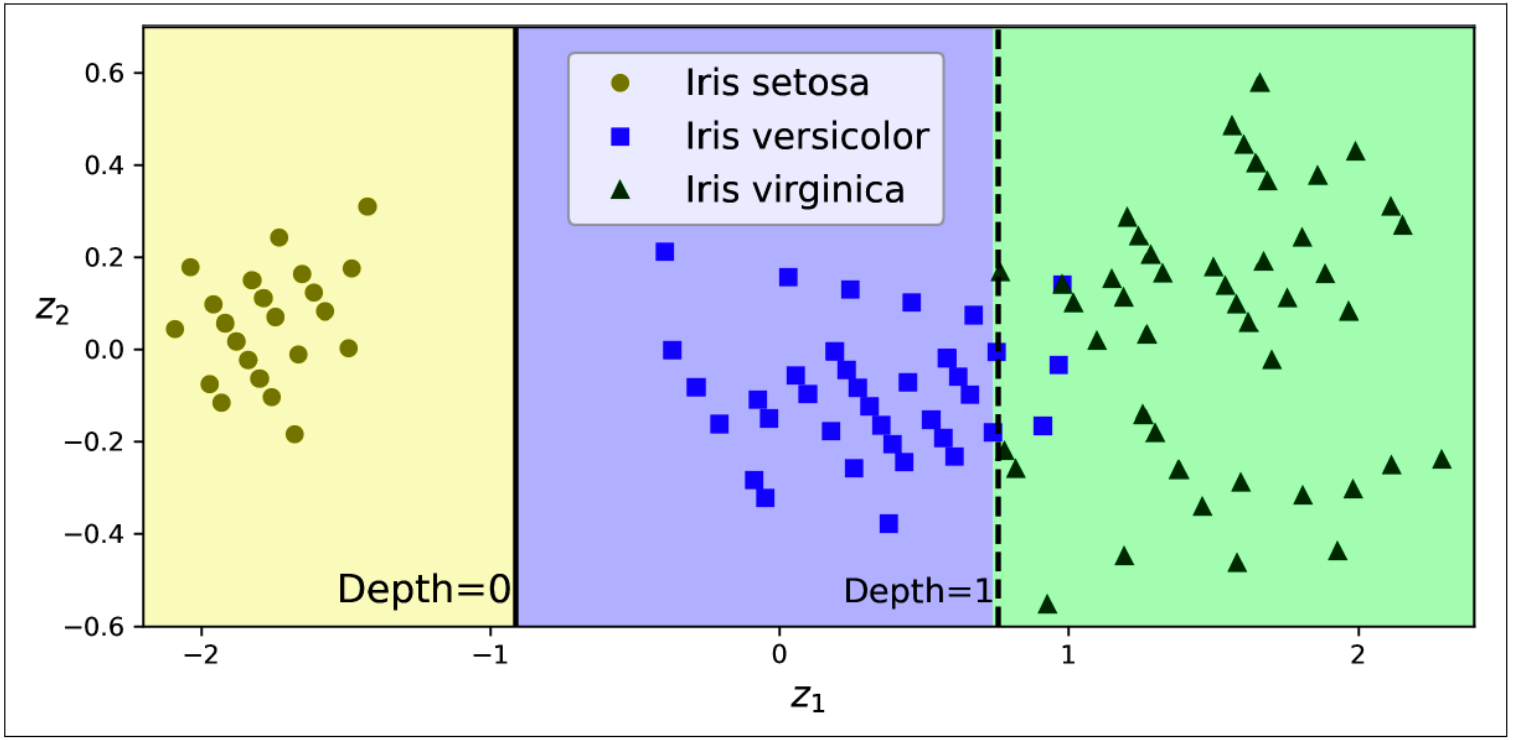
# Sensibilidad a la orientación del eje

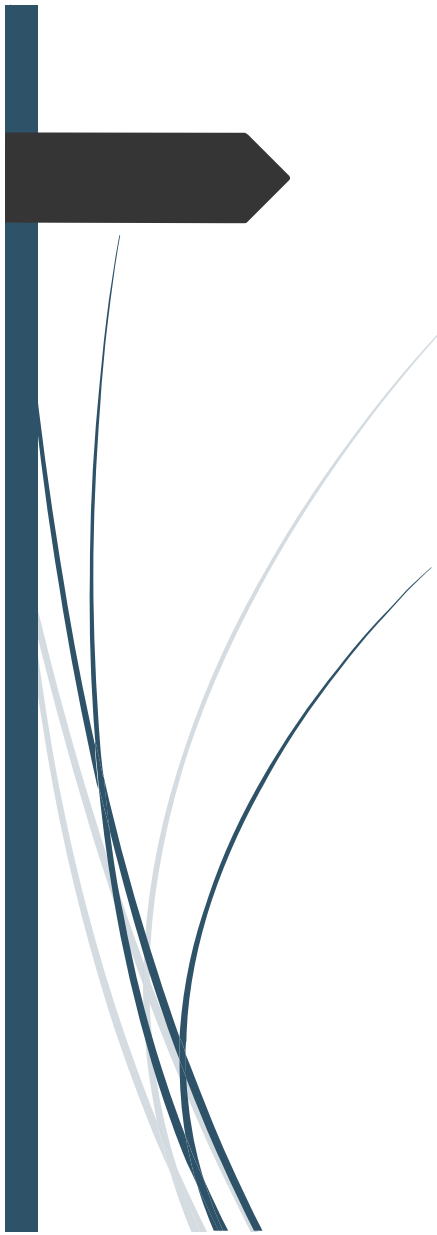
- Los Árboles de decisión son relativamente fáciles de entender e interpretar, sencillos de utilizar, versátiles y potentes.
- Sin embargo, tienen algunas limitaciones.
  - Como habrá observado, a los árboles de decisión les encantan los límites de decisión ortogonales (lo que los hace sensibles a la orientación de los datos).
  - La siguiente figura muestra un conjunto de datos sencillo linealmente separable: a la izquierda, un árbol de decisión puede dividirlo fácilmente, mientras que a la derecha, después de girar el conjunto de datos 45°, el límite de decisión parece innecesariamente enrevesado. Aunque ambos árboles de decisión se ajustan perfectamente al conjunto de entrenamiento, es muy probable que el modelo de la derecha no generalice bien.



# Sensibilidad a la orientación del eje

- Una forma de limitar este problema es escalar los datos y, a continuación, aplicar una transformación de *análisis de componentes principales (Principal Component Analysis, PCA)*.





# Los árboles de decisión tienen una alta varianza

- El principal problema de los árboles de decisión es que tienen una varianza bastante alta: pequeños cambios en los hiperparámetros o en los datos pueden producir modelos muy diferentes.
- Dado que el algoritmo de entrenamiento utilizado por Scikit-Learn es estocástico -selecciona aleatoriamente el conjunto de características a evaluar en cada nodo-, incluso volver a entrenar el mismo árbol de decisión con exactamente los mismos datos puede producir un modelo muy diferente.
- Por suerte, al promediar las predicciones de muchos árboles, es posible reducir la varianza de forma significativa. Este conjunto de árboles se denomina *random forest* y es uno de los tipos de modelos más potentes que existen en la actualidad.