Ignorar esta parte

Complejidad computacional

La Ecuación Normal calcula la inversa de X^T **X**, que es un $(n + 1) \times (n + 1)$ (donde n es el número de características). La *complejidad computacional* de invertir una matriz de este tipo suele oscilar entre $O(n2,4)$ y $O(n3)$, dependiendo de la implementación. En otras palabras, si se duplica el número de características, el tiempo de cálculo se multiplica aproximadamente entre $22,4 = 5,3$ y $23 = 8$.

El enfoque SVD utilizado por la clase `LinearRegression` de Scikit-Learn es de aproximadamente $O(n^2)$. Si se duplica el número de características, el tiempo de cálculo se multiplica aproximadamente por 4.

> **ADVERTENCIA**
>
> Tanto la Ecuación Normal como el enfoque SVD se vuelven muy lentos cuando el número de características aumenta (por ejemplo, 100.000). Por el lado positivo, ambos son lineales con respecto al número de instancias en el conjunto de entrenamiento (son $O(m)$), por lo que manejan grandes conjuntos de entrenamiento de manera eficiente, siempre que quepan en la memoria.

Además, una vez entrenado el modelo de regresión lineal (mediante la ecuación normal o cualquier otro algoritmo), las predicciones son muy rápidas: la complejidad computacional es lineal con respecto al número de casos sobre los que se quieren hacer predicciones y al número de características. En otras palabras, hacer predicciones sobre el doble de casos (o el doble de características) llevará aproximadamente el doble de tiempo.

Ahora veremos una forma muy diferente de entrenar un modelo de Regresión Lineal, que se adapta mejor a los casos en los que hay un gran número de características o demasiadas instancias de entrenamiento para que quepan en la memoria.

Descenso gradual

Desde aquí

El Descenso Gradiente es un algoritmo de optimización genérico capaz de encontrar soluciones óptimas a una amplia gama de problemas. La idea general de Gradient

El descenso consiste en ajustar los parámetros de forma iterativa para minimizar una función de coste.

Supongamos que estás perdido en las montañas, en medio de una densa niebla, y sólo puedes sentir la pendiente del suelo bajo tus pies. Una buena estrategia para llegar rápidamente al fondo del valle es ir cuesta abajo en la dirección de la pendiente más pronunciada. Esto es exactamente lo que hace el Descenso Gradiente: mide el gradiente local de la función de error con respecto al vector de parámetros **θ**, y va en la dirección del gradiente descendente. Una vez que el gradiente es cero, ¡se ha alcanzado un mínimo!

En concreto, se empieza llenando **θ** con valores aleatorios (lo que se denomina *inicialización aleatoria*). A continuación, se mejora gradualmente, dando un pequeño paso cada vez, cada paso tratando de disminuir la función de coste (por ejemplo, el MSE), hasta que el algoritmo *converge* a un mínimo (véase la Figura 4-3).

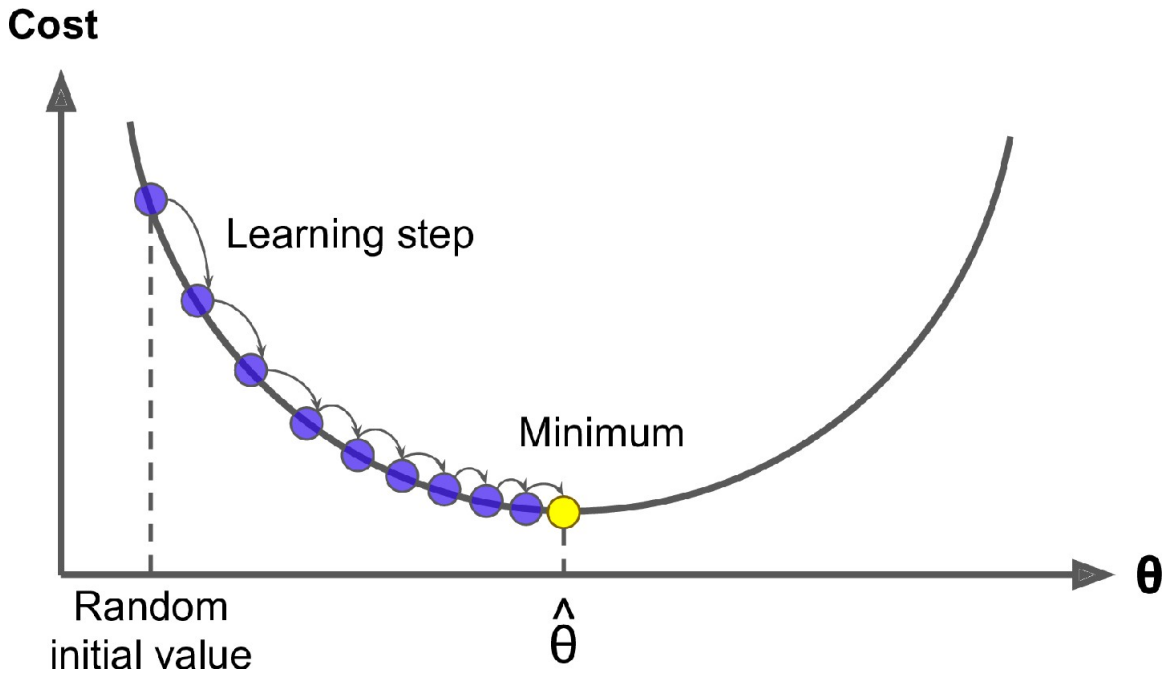

Figura 4-3. En esta representación of Gradient Descent, los parámetros del modelo se inicializan aleatoriamente y se ajustan repetidamente para minimizar la función de coste; el tamaño del paso de aprendizaje es proporcional a la pendiente de la función de coste, por lo que los pasos se hacen gradualmente más pequeños a medida que los parámetros se acercan al mínimo.

Un parámetro importante en el Descenso Gradiente es el tamaño de los pasos, determinado por el hiperparámetro tasa de aprendizaje. Si la tasa de aprendizaje es demasiado

pequeño, entonces el algoritmo tendrá que pasar por muchas iteraciones para converger, lo que llevará mucho tiempo (véase la Figura 4-4).

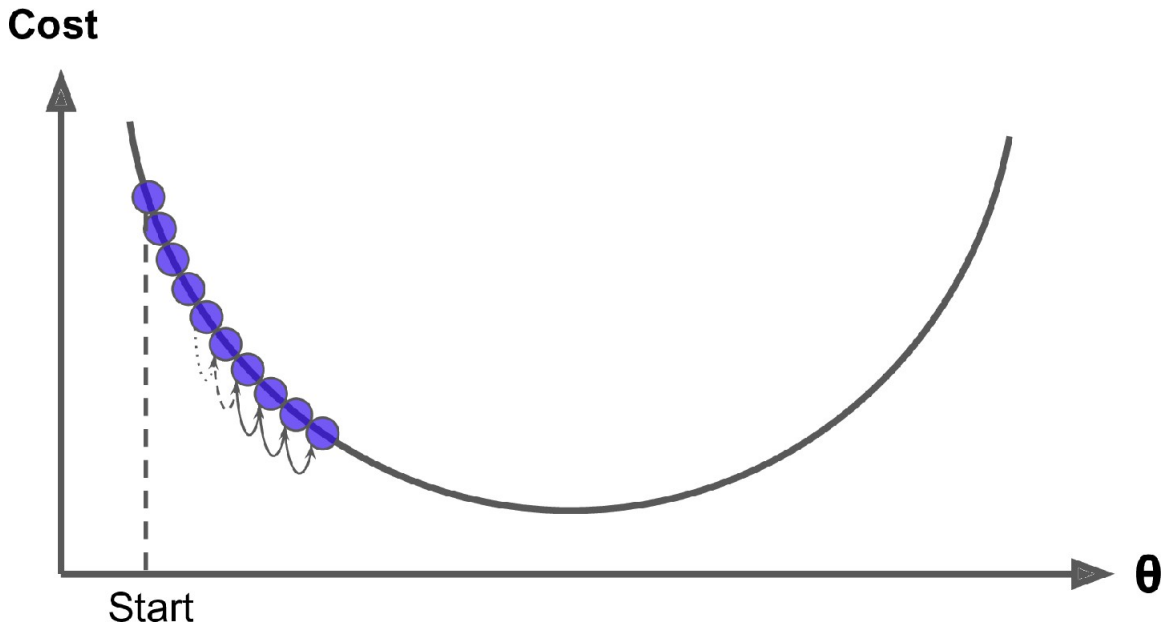

Figura 4-4. La tasa de aprendizaje es demasiado pequeña

Por otro lado, si la tasa de aprendizaje es demasiado alta, podría saltar a través del valle y acabar en el otro lado, posiblemente incluso más arriba de lo que estaba antes. Esto podría hacer que el algoritmo divergiera, con valores cada vez mayores, sin encontrar una buena solución (véase la Figura 4-5).

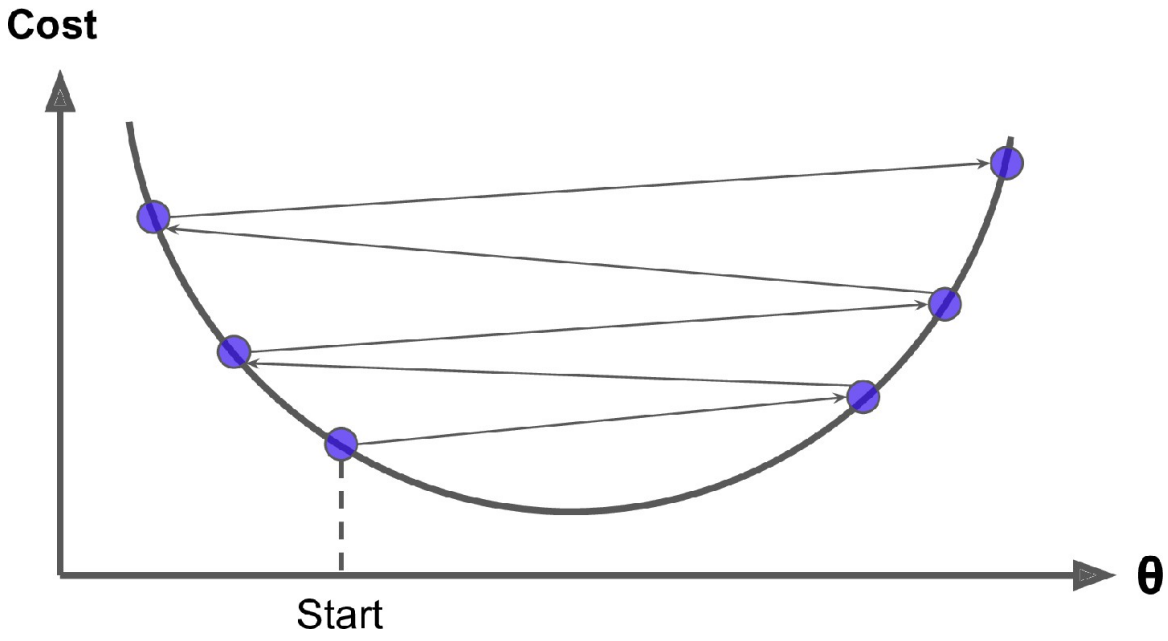

Figura 4-5. La tasa de aprendizaje es demasiado alta

Por último, no todas las funciones de coste parecen cuencos bonitos y regulares. Puede haber agujeros, crestas, mesetas y todo tipo de terrenos irregulares, lo que dificulta la convergencia al mínimo. La Figura 4-6 muestra los dos retos principales de la Descendencia Gradiente. Si la inicialización aleatoria inicia el algoritmo a la izquierda, entonces convergerá a un *mínimo local*, que no es tan bueno como el *mínimo global*. Si empieza por la derecha, tardará mucho tiempo en cruzar la meseta. Y si se detiene demasiado pronto, nunca alcanzará el mínimo global.

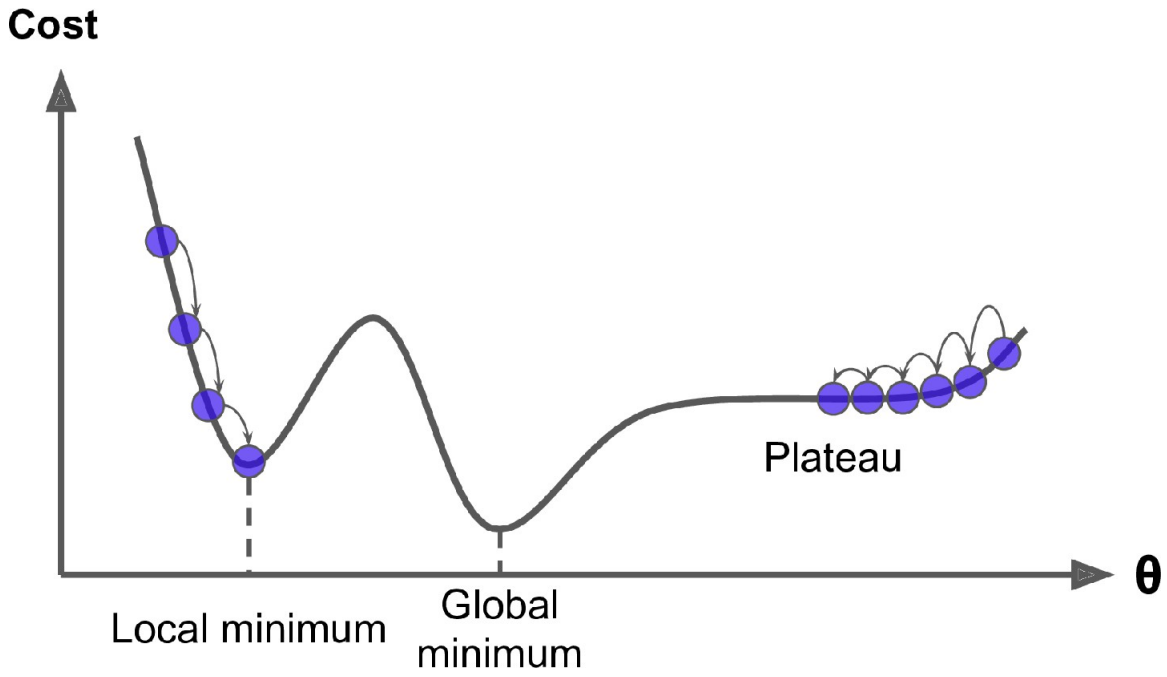

Figura 4-6. Gradiente de descenso pitfalls

Afortunadamente, la función de coste MSE para un modelo de regresión lineal es una *función convexa*, lo que significa que si se eligen dos puntos cualesquiera de la curva, el segmento de línea que los une nunca cruza la curva. Esto implica que no hay mínimos locales, sólo un mínimo global. También es una función continua con una pendiente que nunca cambia bruscamente.[3] Estos dos hechos tienen una gran consecuencia: Se garantiza que el Descenso Gradiente se acerque arbitrariamente al mínimo global (si se espera lo suficiente y si la tasa de aprendizaje no es demasiado alta).

De hecho, la función de coste tiene la forma de un cuenco, pero puede ser un cuenco alargado si las características tienen escalas muy diferentes. La Figura 4-7 muestra el Descenso Gradiente en un conjunto de entrenamiento donde las características 1 y 2 tienen la misma escala (a la izquierda), y en un conjunto de entrenamiento donde la característica 1 tiene valores mucho más pequeños que la característica 2 (a la derecha).[4]

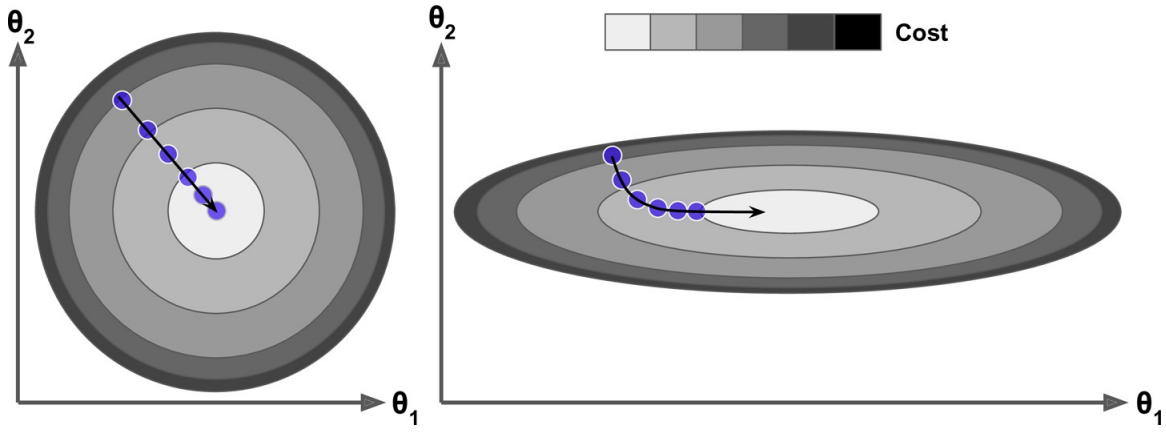

Figura 4-7. Descenso de gradiente con (left) y sin (derecha) escalado de feature.

Como puedes ver, a la izquierda el algoritmo de Descenso Gradiente va directo hacia el mínimo, alcanzándolo así rápidamente, mientras que a la derecha va primero en una dirección casi ortogonal a la dirección del mínimo global, y termina con una larga marcha por un valle casi plano. Al final llegará al mínimo, pero tardará mucho tiempo.

> **ADVERTENCIA**
>
> Cuando utilice el Descenso Gradiente, debe asegurarse de que todas las características tienen una escala similar (por ejemplo, utilizando la clase `StandardScaler` de Scikit-Learn), o de lo contrario tardará mucho más en converger.

Este diagrama también ilustra el hecho de que entrenar un modelo significa buscar una combinación de parámetros del modelo que minimice una función de coste (sobre el conjunto de entrenamiento). Se trata de una búsqueda en el *espacio de parámetros del modelo*: cuantos más parámetros tenga un modelo, más dimensiones tendrá este espacio y más difícil será la búsqueda: buscar una aguja en un pajar de 300 dimensiones es mucho más complicado que en 3 dimensiones. Afortunadamente, como la función de coste es convexa en el caso de la regresión lineal, la aguja está simplemente en el fondo del pajar.

Descenso gradual por lotes

Para aplicar el Descenso Gradiente, hay que calcular el gradiente de la función de coste con respecto a cada parámetro θ_j del modelo. En otras palabras, es necesario calcular cuánto cambiará la función de coste si se cambia θj sólo un poco. Esto se llama *derivada parcial*. Es como preguntar "¿Cuál es la pendiente de la montaña bajo mis pies si miro hacia el este?" y luego hacer la misma pregunta mirando hacia el norte (y así sucesivamente para todas las demás dimensiones, si se puede imaginar un universo con más de tres dimensiones). La ecuación 4-5 calcula la derivada parcial de la función de coste con respecto al parámetro θ_j, anotado $\frac{\partial}{\partial \theta_j} \text{MSE}(\boldsymbol{\theta})$.

Ecuación 4-5. Derivadas parciales de la función de costes

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

En lugar de calcular estas derivadas parciales individualmente, se puede utilizar la Ecuación 4-6 para calcularlas todas de una sola vez. El vector gradiente, anotado $\nabla_{\boldsymbol{\theta}} \text{MSE}(\boldsymbol{\theta})$, contiene todas las derivadas parciales de la función de coste (una por cada parámetro del modelo).

Ecuación 4-6. Vector gradiente de la función coste

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

> **ADVERTENCIA**
>
> Observe que esta fórmula implica cálculos sobre todo el conjunto de entrenamiento **X**, ¡en cada paso del Gradient Descent! Por eso el algoritmo se llama *Batch Gradient Descent*: utiliza todo el lote de datos de entrenamiento en cada paso (en realidad, *Full Gradient Descent sería* probablemente un nombre mejor). Como resultado, es terriblemente lento en conjuntos de entrenamiento muy grandes (pero en breve veremos algoritmos de Gradient Descent mucho más rápidos). Sin embargo, el Descenso Gradiente se escala bien con el número de características; entrenar un modelo de Regresión Lineal cuando hay cientos de miles de características es mucho más rápido usando el Descenso Gradiente que usando la Ecuación Normal o la descomposición SVD.

Una vez que tienes el vector gradiente, que apunta cuesta arriba, basta con ir en la dirección opuesta para ir cuesta abajo. Esto significa restar $\nabla_{\theta}\text{MSE}(\theta)$ de θ.
Aquí es donde entra en juego la tasa de aprendizaje η:[5] multiplica el vector gradiente por η para determinar el tamaño del paso descendente (Ecuación 4-7).

Ecuación 4-7. Paso de descenso gradiente

$$\theta^{(\text{siguiente paso})} = \theta - \eta \nabla_{\theta} \text{MSE}(\theta)$$

Veamos una aplicación rápida de este algoritmo:

```
eta = 0.1 # tasa de aprendizaje
n_iteraciones = 1000
m = 100

theta = np.random.randn(2,1) # inicialización aleatoria

for iteración in range(n_iteraciones):
    gradientes = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradientes
```

No ha sido tan difícil. Veamos la `theta` resultante:

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

¡Eso es exactamente lo que encontró la Ecuación Normal! El Descenso Gradiente funcionó perfectamente. Pero, ¿y si hubieras utilizado una tasa de aprendizaje `eta` diferente?

La Figura 4-8 muestra los 10 primeros pasos del Descenso Gradiente utilizando tres velocidades de aprendizaje diferentes (la línea discontinua representa el punto de partida).

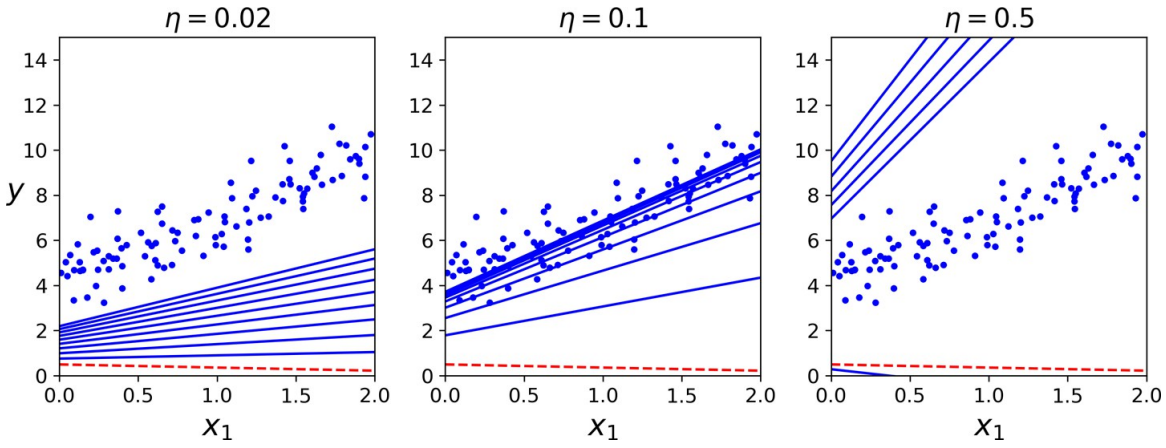

Figura 4-8. Descenso por gradiente con distintos ritmos de aprendizaje

A la izquierda, la tasa de aprendizaje es demasiado baja: el algoritmo acabará llegando a la solución, pero tardará mucho tiempo. En el centro, la tasa de aprendizaje parece bastante buena: en unas pocas iteraciones, ya ha convergido a la solución. A la derecha, la tasa de aprendizaje es demasiado alta: el algoritmo diverge, saltando por todas partes y alejándose cada vez más de la solución a cada paso.

Para encontrar una buena tasa de aprendizaje, puede utilizar la búsqueda en la cuadrícula (consulte el capítulo 2). Sin embargo, es posible que desee limitar el número de iteraciones para que la búsqueda de cuadrícula pueda eliminar los modelos que tardan demasiado en converger.

Quizá se pregunte cómo fijar el número de iteraciones. Si es demasiado bajo, todavía estará lejos de la solución óptima cuando el algoritmo se detenga; pero si es demasiado alto, perderá tiempo mientras los parámetros del modelo ya no cambian. Una solución simple es establecer un número muy grande de iteraciones, pero para interrumpir el algoritmo cuando el vector gradiente se convierte en pequeña-es decir, cuando su norma se hace más pequeño que un pequeño número ϵ (llamado la *tolerancia*)-porque esto sucede cuando Gradient Descent (casi) ha alcanzado el mínimo.

TASA DE CONVERGENCIA

Cuando la función de coste es convexa y su pendiente no cambia bruscamente (como es el caso de la función de coste MSE), el Descenso Gradiente por Lotes con una tasa de aprendizaje fija acabará convergiendo a la solución óptima, pero es posible que tenga que esperar un poco: puede tardar $O(1/\epsilon)$ iteraciones en alcanzar el óptimo dentro de un rango de ϵ, dependiendo de la forma de la función de coste. Si divides la tolerancia por 10 para tener una solución más precisa, es posible que el algoritmo tenga que ejecutarse unas 10 veces más.

Descenso Gradiente Estocástico

El principal problema del descenso gradiente por lotes es que utiliza todo el conjunto de entrenamiento para calcular los gradientes en cada paso, lo que lo hace muy lento cuando el conjunto de entrenamiento es grande. En el extremo opuesto, el Descenso Gradiente *Estocástico* elige una instancia aleatoria del conjunto de entrenamiento en cada paso y calcula los gradientes basándose sólo en esa única instancia. Obviamente, trabajar con un único caso a la vez hace que el algoritmo sea mucho más rápido porque tiene muy pocos datos que manipular en cada iteración. También `permite` entrenar conjuntos de entrenamiento enormes, ya que sólo se necesita una instancia en memoria en cada iteración (la DG estocástica puede implementarse como un algoritmo fuera del núcleo; véase el capítulo 1).

Por otra parte, debido a su naturaleza estocástica (es decir, aleatoria), este algoritmo es mucho menos regular que el Descenso Gradiente por Lotes: en lugar de disminuir suavemente hasta alcanzar el mínimo, la función de coste rebotará hacia arriba y hacia abajo, disminuyendo sólo en promedio. Con el tiempo, acabará muy cerca del mínimo, pero una vez allí seguirá rebotando, sin estabilizarse nunca (véase la Figura 4-9). Por tanto, una vez que el algoritmo se detiene, los valores finales de los parámetros son buenos, pero no óptimos.

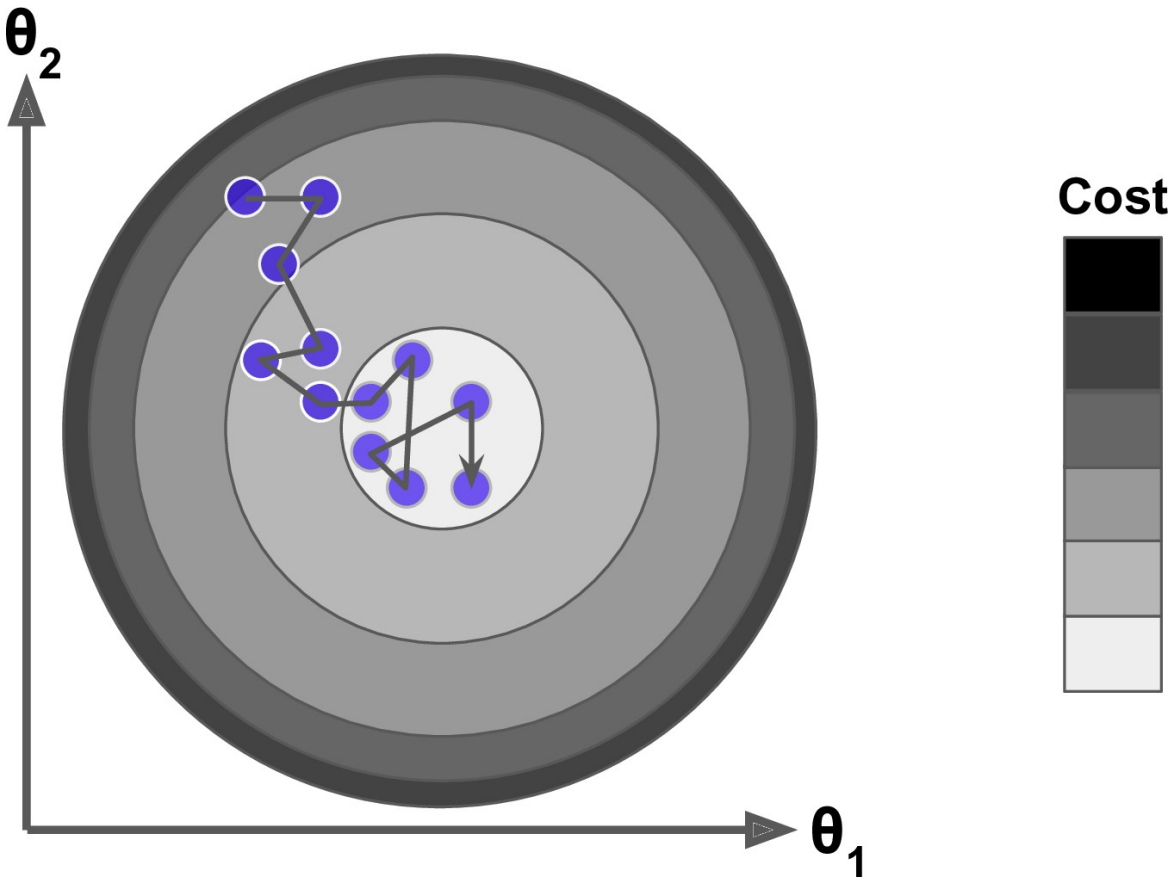

Figura 4-9. Con el Descenso Gradiente Estocástico, cada paso de entrenamiento es mucho más rápido pero también mucho más estocástico que cuando se utiliza el Descenso Gradiente Por Lotes.

Cuando la función de coste es muy irregular (como en la Figura 4-6), esto puede ayudar al algoritmo a salir de los mínimos locales, por lo que el Descenso Gradiente Estocástico tiene más posibilidades de encontrar el mínimo global que el Descenso Gradiente por Lotes.

Por tanto, la aleatoriedad es buena para escapar de los óptimos locales, pero mala porque significa que el algoritmo nunca puede asentarse en el mínimo. Una solución a este dilema es reducir gradualmente el ritmo de aprendizaje. Los pasos comienzan siendo grandes (lo que ayuda a progresar rápidamente y escapar de los mínimos locales) y luego se van haciendo cada vez más pequeños, lo que permite al algoritmo asentarse en el mínimo global. Este proceso es similar al recocido *simulado,* un algoritmo inspirado en el proceso metalúrgico del recocido, en el que el metal fundido se enfría lentamente. La función que determina el ritmo de aprendizaje en cada iteración se denomina *programa de aprendizaje*. Si el ritmo de aprendizaje se reduce demasiado deprisa, puede quedarse atascado en un mínimo local, o incluso acabar congelado a medio camino del

mínimo. Si el ritmo de aprendizaje se reduce demasiado despacio, puede saltar alrededor del mínimo durante mucho tiempo y acabar con una solución subóptima si detiene el entrenamiento demasiado pronto.

Este código implementa el Descenso Gradiente Estocástico utilizando un programa de aprendizaje simple:

```
n_epocs = 50
t0, t1 = 5, 50 # hiperparámetros del programa de aprendizaje

def programa_aprendizaje(t):
    return t0 / (t + t1)

theta = np.random.randn(2,1) # inicialización aleatoria

for epochin range(n_epochs):for
    iin range(m):
        random_index = np.random.randint(m)
        xi = X_b[random_index:random_index+1]
        yi = y[random_index:random_index+1]
        gradientes = 2 * xi.T.dot(xi.dot(theta) - yi)
        eta = learning_schedule(epoch * m + i)
        theta = theta - eta * gradientes
```

Por convención, iteramos por rondas de *m* iteraciones; cada ronda se denomina *época*. Mientras que el código Batch Gradient Descent itera 1.000 veces a través de todo el conjunto de entrenamiento, este código lo hace sólo 50 veces y alcanza una solución bastante buena:

```
>>> theta
array([[4.21076011],
       [2.74856079]])
```

La Figura 4-10 muestra los primeros 20 pasos del entrenamiento (observe lo irregulares que son los pasos).

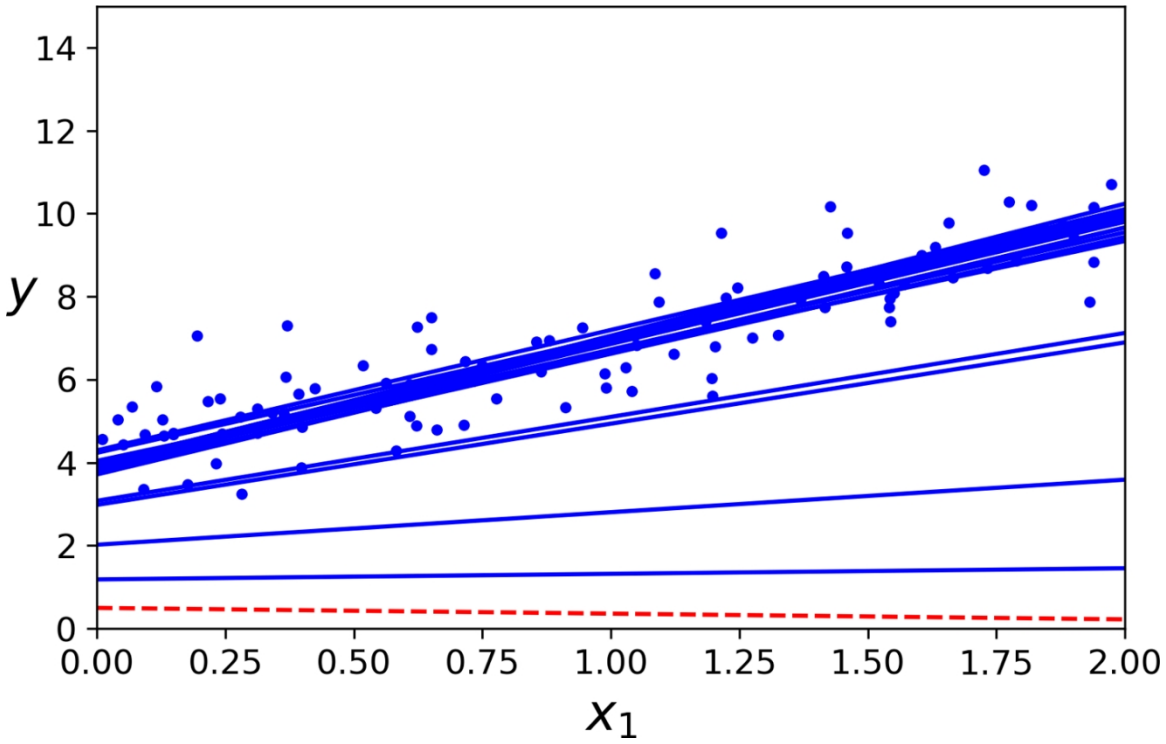

Figura 4-10. Los fprimeros 20 pasos del Descenso Gradiente Estocástico

Tenga en cuenta que como las instancias se eligen aleatoriamente, algunas instancias pueden ser elegidas varias veces por época, mientras que otras pueden no ser elegidas en absoluto. Si quiere estar seguro de que el algoritmo pasa por todas las instancias en cada epoch, otro enfoque es barajar el conjunto de entrenamiento (asegurándose de barajar las características de entrada y las etiquetas conjuntamente), luego pasar por él instancia por instancia, luego barajarlo de nuevo, y así sucesivamente. Sin embargo, este método suele converger más lentamente.

ADVERTENCIA

Cuando se utiliza el Descenso Gradiente Estocástico, las instancias d e entrenamiento deben ser independientes e idénticamente distribuidas (IID) para asegurar que los parámetros se dirigen hacia el óptimo global, en promedio. Una forma sencilla de garantizarlo es barajar las instancias durante el entrenamiento (por ejemplo, elegir cada instancia aleatoriamente o barajar el conjunto de entrenamiento al principio de cada época). Si no baraja las instancias -por ejemplo, si las instancias están ordenadas por etiqueta- entonces SGD empezará optimizando para una etiqueta, luego para la siguiente, y así sucesivamente, y no se asentará cerca del mínimo global.

Para realizar una regresión lineal utilizando GD estocástico con Scikit-Learn, puede utilizar la clase `SGDRegressor`, que por defecto optimiza la función de coste de error al cuadrado. El siguiente código se ejecuta durante un máximo de 1.000 épocas o hasta que la pérdida disminuye en menos de 0,001 durante una época (`max_iter=1000`, `tol=1e-3`). Comienza con una tasa de aprendizaje de 0,1 (`eta0=0,1`), utilizando el programa de aprendizaje por defecto (diferente del anterior). Por último, no utiliza ninguna regularización (`penalty=None`; más detalles al respecto en breve):

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

Una vez más, se encuentra una solución bastante cercana a la que devuelve la Ecuación Normal:

```
>>> sgd_reg.intercept_, sgd_reg.coef_
(array([4.24365286]), array([2.8250878]))
```

Mini-lote de Descenso Gradiente

El último algoritmo de descenso gradiente que vamos a estudiar se llama *Mini-lote de descenso gradiente*. Es fácil de entender una vez que se conocen los algoritmos Batch y Stochastic Gradient Descent: en cada paso, en lugar de calcular los gradientes basándose en el conjunto de entrenamiento completo (como en Batch GD) o basándose en una sola instancia (como en Stochastic GD), Mini-batch GD calcula los gradientes en pequeños conjuntos aleatorios de instancias llamados *mini-batches*. La principal ventaja de Mini-batch GD sobre Stochastic GD es que puede obtener un aumento de rendimiento gracias a la optimización por hardware de las operaciones matriciales, especialmente cuando se utilizan GPU.

El progreso del algoritmo en el espacio de parámetros es menos errático que con la DG Estocástica, especialmente con minilotes bastante grandes. Como resultado, Mini-lote GD terminará caminando un poco más cerca del mínimo que Estocástico GD-pero puede ser más difícil para él escapar de los mínimos locales (en el caso de problemas que sufren de mínimos locales, a diferencia de Regresión Lineal). La Figura 4-11 muestra los caminos tomados por los tres algoritmos de Descenso Gradiente en el espacio de parámetros durante el entrenamiento. Todos acaban cerca del

mínimo, pero el camino de Batch GD se detiene en el mínimo, mientras que

tanto Stochastic GD como Mini-batch GD siguen dando vueltas. Sin embargo, no olvide que Batch GD tarda mucho tiempo en dar cada paso, y Stochastic GD y Mini-batch GD también llegarían al mínimo si utilizara un buen programa de aprendizaje.

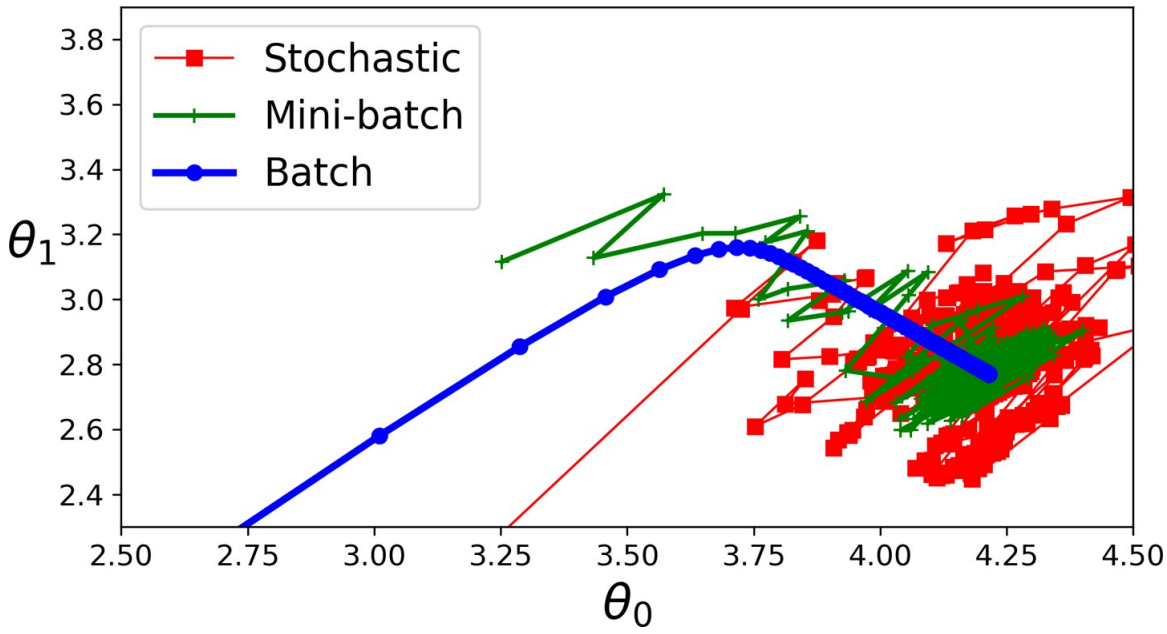

Figura 4-11. Trayectorias de Descenso Gradiente en el espacio de parámetros

Comparemos los algoritmos que hemos discutido hasta ahora para la Regresión Lineal[6] (recuerde que *m* es el número de instancias de entrenamiento y *n* es el número de características); vea la Tabla 4-1.

Tabla 4-1. Comparación de algoritmos de regresión lineal

Algoritmos	Gran *m*	Soporte fuera del núcleo	Gran *n*	Hiperparámetros	Escala necesaria	Scikit-Learn
Ecuación Normal	Rápido	No	Lento	0	No	N/A
SVD	Rápido	No	Lento	0	No	`Regresión lineal`
Lote GD	Lento	No	Rápido	2	Sí	`SGDRegresor`
Estocástico GD	Rápido	Sí	Rápido	≥2	Sí	`SGDRegresor`
Minilotes	Rápido	Sí	Rápido	≥2	Sí	`SGDRegresor`

NOTA

Casi no hay diferencias después del entrenamiento: todos estos algoritmos acaban con modelos muy similares y hacen predicciones exactamente de la misma manera.

Regresión polinómica

Saltarse esto

¿Y si los datos son más complejos que una línea recta? Sorprendentemente, puede utilizar un modelo lineal para ajustar datos no lineales. Una forma sencilla de hacerlo es añadir potencias de cada característica como nuevas características y, a continuación, entrenar un modelo lineal en este conjunto ampliado de características. Esta técnica se denomina *regresión polinómica*.

Veamos un ejemplo. Primero, generemos algunos datos no lineales, basados en una simple *ecuación cuadrática*[7] (más algo de ruido; ver Figura 4-12):

```
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0,5 * X**2 + X + 2 + np.random.randn(m, 1)
```

Elastic Net es un término medio entre Ridge Regression y Lasso Regression. El término de regularización es una simple mezcla de los términos de regularización de Ridge y Lasso, y se puede controlar la relación de mezcla *r*. Cuando $r = 0$, Elastic Net es equivalente a Ridge Regression, y cuando $r = 1$, es equivalente a Lasso Regression (véase la ecuación 4-12).

Ecuación 4-12. Coste neto elástico ffunción

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

Entonces, ¿cuándo se debe utilizar la regresión lineal simple (es decir, sin ninguna regularización), Ridge, Lasso o Elastic Net? Casi siempre es preferible tener al menos un poco de regularización, por lo que en general debe evitar la regresión lineal simple. Ridge es una buena opción por defecto, pero si sospecha que sólo unas pocas características son útiles, debería preferir Lasso o Elastic Net porque tienden a reducir los pesos de las características inútiles a cero, como ya hemos comentado. En general, se prefiere Elastic Net a Lasso porque Lasso puede comportarse de forma errática cuando el número de características es mayor que el número de instancias de entrenamiento o cuando varias características están fuertemente correlacionadas.

He aquí un breve ejemplo que utiliza `ElasticNet` de Scikit-Learn (`l1_ratio` corresponde a la proporción de mezcla *r*):

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

Parada anticipada

Leer esta parte

Una forma muy distinta de regularizar los algoritmos de aprendizaje iterativo, como el Descenso Gradiente, consiste en detener el entrenamiento en cuanto el error de validación alcanza un mínimo. Esto se denomina *parada temprana*. La Figura 4-20 muestra un modelo complejo (en este caso, un modelo de Regresión Polinómica de alto grado) siendo entrenado con el Descenso Gradiente por Lotes. A medida que pasan las épocas el algoritmo aprende, y su error de predicción (RMSE) en el conjunto de entrenamiento

disminuye, junto con su error de predicción en el conjunto de validación. Sin embargo, al cabo de un tiempo, el error de validación deja de disminuir y empieza a aumentar. Esto indica que el modelo

ha empezado a sobreajustar los datos de entrenamiento. Con la parada anticipada se detiene el entrenamiento en cuanto el error de validación alcanza el mínimo. Es una técnica de regularización tan sencilla y eficaz que Geoffrey Hinton la llamó "un hermoso almuerzo gratis".

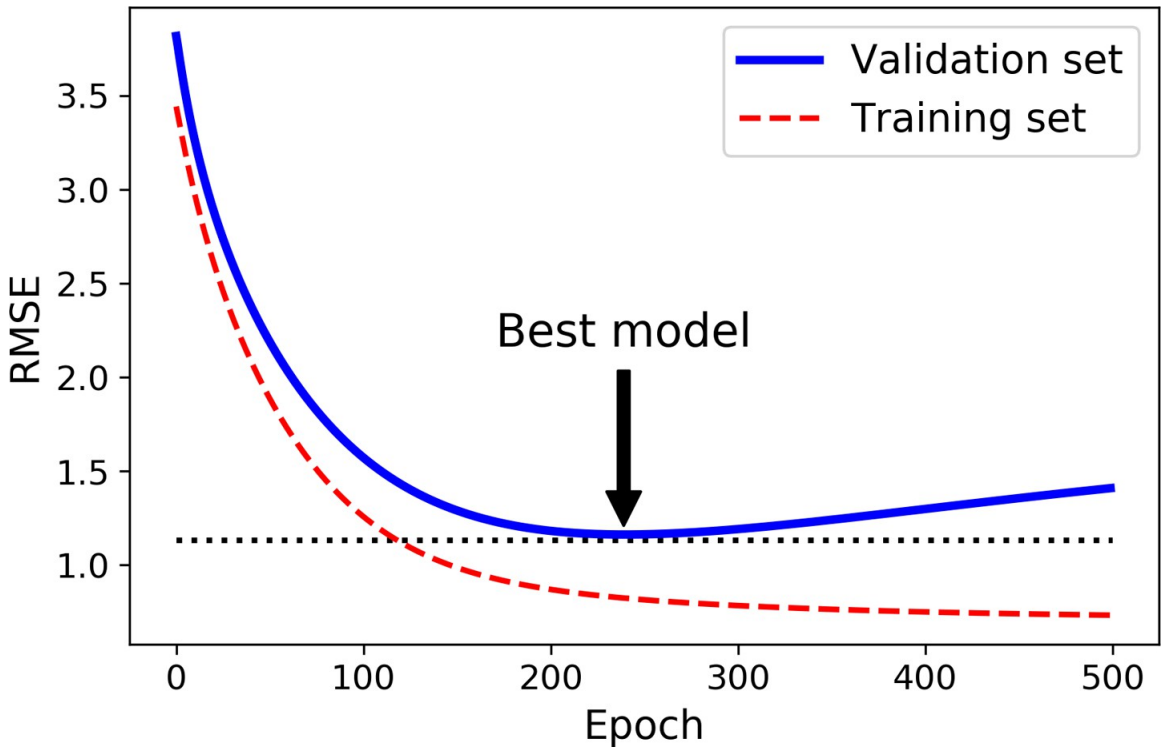

Figura 4-20. Regularización de parada temprana

> **CONSEJO**
>
> Con el Descenso Gradiente Estocástico y Mini-lote, las curvas no son tan suaves, y puede ser difícil saber si se ha alcanzado el mínimo o no. Una solución es detenerse sólo después de que el error de validación haya estado por encima del mínimo durante algún tiempo (cuando esté seguro de que el modelo no lo hará mejor), y luego retroceder los parámetros del modelo hasta el punto en que el error de validación estaba en el mínimo.

He aquí una aplicación básica de la parada anticipada:

```
from sklearn.base import clonar

# preparar los datos
poly_scaler = Tubería([
        ("poly_features", PolynomialFeatures(grado=90,
```