CAPÍTULO 4

Modelos de treinamento

Com os e-books de lançamento antecipado, você recebe os livros em sua forma mais antiga - o conteúdo bruto e não editado do autor enquanto ele escreve - para que possa aproveitar essas tecnologias muito antes do lançamento oficial desses títulos. O capítulo a seguir será o capítulo 4 da versão final do livro.

Até agora, tratamos os modelos de Machine Learning e seus algoritmos de treinamento principalmente como caixas pretas. Se você fez alguns dos exercícios dos capítulos anteriores, talvez tenha se surpreendido com o quanto pode ser feito sem saber nada sobre o que está por trás disso: você otimizou um sistema de regressão, melhorou um classificador de imagens de dígitos e até mesmo criou um classificador de spam do zero - tudo isso sem saber como eles realmente funcionam. De fato, em muitas situações, não é necessário conhecer os detalhes da implementação.

Entretanto, ter uma boa compreensão de como as coisas funcionam pode ajudá-lo a identificar rapidamente o modelo apropriado, o algoritmo de treinamento correto a ser usado e um bom conjunto de hiperparâmetros para a sua tarefa. Entender o que está por trás do sistema também o ajudará a depurar problemas e a realizar análises de erros com mais eficiência. Por fim, a maioria dos tópicos discutidos neste capítulo será essencial para entender, criar e treinar redes neurais (discutidas na Parte II deste livro).

Neste capítulo, começaremos examinando o modelo de Regressão Linear, um dos modelos mais simples que existem. Discutiremos duas maneiras bem diferentes de treiná-lo:

- Usando uma equação direta de "forma fechada" que calcula diretamente os parâmetros do modelo que melhor se ajustam ao conjunto de treinamento (ou seja, os parâmetros do modelo que minimizam a função de custo no conjunto de treinamento).

- Usando uma abordagem de otimização iterativa, denominada Gradient Descent (GD), que ajusta gradualmente os parâmetros do modelo para minimizar a função de custo no conjunto de treinamento, convergindo, por fim, para o mesmo conjunto de parâmetros do primeiro método. Veremos algumas variantes do Gradient Descent que usaremos repetidamente quando estudarmos redes neurais na Parte II: Batch GD, Mini-batch GD e Stochastic GD.

Em seguida, veremos a Regressão Polinomial, um modelo mais complexo que pode se ajustar a conjuntos de dados não lineares. Como esse modelo tem mais parâmetros do que a Regressão Linear, ele é mais propenso a se ajustar excessivamente aos dados de treinamento. Por isso, veremos como detectar se esse é o caso ou não, usando curvas de aprendizado, e depois examinaremos várias técnicas de regularização que podem reduzir o risco de ajuste excessivo do conjunto de treinamento.

Por fim, examinaremos mais dois modelos que são comumente usados em tarefas de classificação: Regressão logística e Regressão Softmax.

Este capítulo apresenta algumas equações matemáticas, usando noções básicas de álgebra linear e cálculo. Para entender essas equações, você precisará saber o que são vetores e matrizes, como transpô-los, multiplicá-los e invertê-los, e o que são derivadas parciais. Se você não estiver familiarizado com esses conceitos, consulte os tutoriais introdutórios de álgebra linear e cálculo disponíveis como notebooks Jupyter no material suplementar on-line. Para aqueles que são realmente alérgicos à matemática, você ainda deve ler este capítulo e simplesmente pular as equações; esperamos que o texto seja suficiente para ajudá-lo a entender a maioria dos conceitos.

Regressão linear

No Capítulo 1, examinamos um modelo de regressão simples de satisfação com a vida: $life\_satisfac\text{-}tion = \theta_0 + \theta_{(1)} \times GDP\_per\_capita$.

Esse modelo é apenas uma função linear do recurso de entrada `GDP_per_capita`. θ_0 e θ_1 são os parâmetros do modelo.

De modo mais geral, um modelo linear faz uma previsão simplesmente calculando uma soma ponderada dos recursos de entrada, mais uma constante chamada de *termo de polarização* (também chamado de *termo de interceptação*), conforme mostrado na Equação 4-1.

Equação 4-1. Previsão do modelo de regressão linear

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

- $\hat{y}$ é o valor previsto.

- n é o número de recursos.
- x_ié o valor do i^{ésimo}recurso.
- θ_jé o parâmetro do modelo jth(incluindo o termo de polarização θ_0e os pesos dos recursos $\theta_1, \theta_2, \cdots, \theta_n$).

Isso pode ser escrito de forma muito mais concisa usando uma forma vetorizada, conforme mostrado na Equação 4-2.

Equação 4-2. Previsão do modelo de regressão linear (forma vetorizada)

$$\hat{y} = h_{\boldsymbol{\theta}}(\mathbf{x}) = \boldsymbol{\theta} \cdot \mathbf{x}$$

- $\boldsymbol{\theta}$ é o *vetor de parâmetros* do modelo, que contém o termo de polarização θ_0e os pesos das características θ_1a θ_n.
- $\mathbf{x}$ é o *vetor de características* da instância, contendo x_0a x_n, com $x_0$sempre igual a 1.
- $\boldsymbol{\theta} \cdot \mathbf{x}$ é o produto escalar dos vetores $\boldsymbol{\theta}$ e $\mathbf{x}$, que, obviamente, é igual a $\theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_{(n)}$.
- $h_{\boldsymbol{\theta}}$é a função de hipótese, usando os parâmetros do modelo $\boldsymbol{\theta}$.

No aprendizado de máquina, os vetores são geralmente representados como *vetores coluna*, que são matrizes 2D com uma única coluna. Se $\boldsymbol{\theta}$ e $\mathbf{x}$ forem vetores coluna-coluna, a previsão será: $y = \boldsymbol{\theta}^{(T)} \mathbf{x}$, em que $\boldsymbol{\theta}^T$é a *transposição* de $\boldsymbol{\theta}$ (um vetor de linha em vez de um vetor de coluna) e $\boldsymbol{\theta}^T\mathbf{x}$ é a multiplicação da matriz de $\boldsymbol{\theta}^T$e $\mathbf{x}$. É claro que se trata da mesma previsão, exceto pelo fato de que agora ela é representada como uma matriz de célula única em vez de um valor escalar. Neste livro, usaremos essa notação para evitar a alternância entre produtos de pontos e multiplicações de matrizes.

Muito bem, esse é o modelo de regressão linear, então como podemos treiná-lo? Bem, lembre-se de que treinar um modelo significa definir seus parâmetros de modo que o modelo se ajuste melhor ao conjunto de treinamento. Para isso, primeiro precisamos de uma medida de quão bem (ou mal) o modelo se ajusta aos dados de treinamento. No Capítulo 2, vimos que a medida de desempenho mais comum de um modelo de regressão é a raiz do erro quadrático médio (RMSE) (Equação 2-1). Portanto, para treinar um modelo de regressão linear, você precisa encontrar o valor de $\boldsymbol{\theta}$ que minimize o RMSE. Na prática, é mais simples minimizar o erro quadrático médio (MSE)

do que o RMSE, e isso leva ao mesmo resultado (porque o valor que minimiza uma função também minimiza sua raiz quadrada).[1]

O MSE de uma hipótese de regressão linear $h_{\boldsymbol{\theta}}$em um conjunto de treinamento $\mathbf{X}$ é calculado usando a Equação 4-3.

Equação 4-3. Função de custo MSE para um modelo de regressão linear

$$\text{MSE}\left(\mathbf{X}, h_{\boldsymbol{\theta}}\right) = \frac{1}{m}\sum_{i=1}^{m}\left(\boldsymbol{\theta}^{(T)}\mathbf{x}^{(i)} - y^{(i)}\right)^2$$

A maioria dessas notações foi apresentada no Capítulo 2 (consulte "Notações" na página 43). A única diferença é que escrevemos $h_{\boldsymbol{\theta}}$em vez de apenas h para deixar claro que o modelo é parametrizado pelo vetor $\boldsymbol{\theta}$. Para simplificar as notações, escreveremos apenas MSE($\boldsymbol{\theta}$) em vez de MSE($\mathbf{X}$, $h_{\boldsymbol{\theta}}$).

A equação normal

Para encontrar o valor de $\boldsymbol{\theta}$ que minimiza a função de custo, há uma *solução de forma fechada* -em outras palavras, uma equação matemática que fornece o resultado diretamente. Isso é chamado de *Equação Normal* (Equação 4-4).[2]

Equação 4-4. Equação normal

$$\widehat{\boldsymbol{\theta}} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{y}$$

- $\widehat{\boldsymbol{\theta}}$ é o valor de $\boldsymbol{\theta}$ que minimiza a função de custo.
- $\mathbf{y}$ é o vetor de valores-alvo que contém $y^{(1)}$a $y^{(m)}$.

Vamos gerar alguns dados de aparência linear para testar essa equação (Figura 4-1):

```
import numpy as np

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

1 É comum que um algoritmo de aprendizado tente otimizar uma função diferente da medida de desempenho usada para avaliar o modelo final. Isso geralmente ocorre porque essa função é mais fácil de calcular, porque tem propriedades de diferenciação úteis que a medida de desempenho não tem ou porque queremos restringir o modelo durante o treinamento, como veremos quando discutirmos a regularização.

2 A demonstração de que isso retorna o valor de $\boldsymbol{\theta}$ que minimiza a função de custo está fora do escopo deste livro.

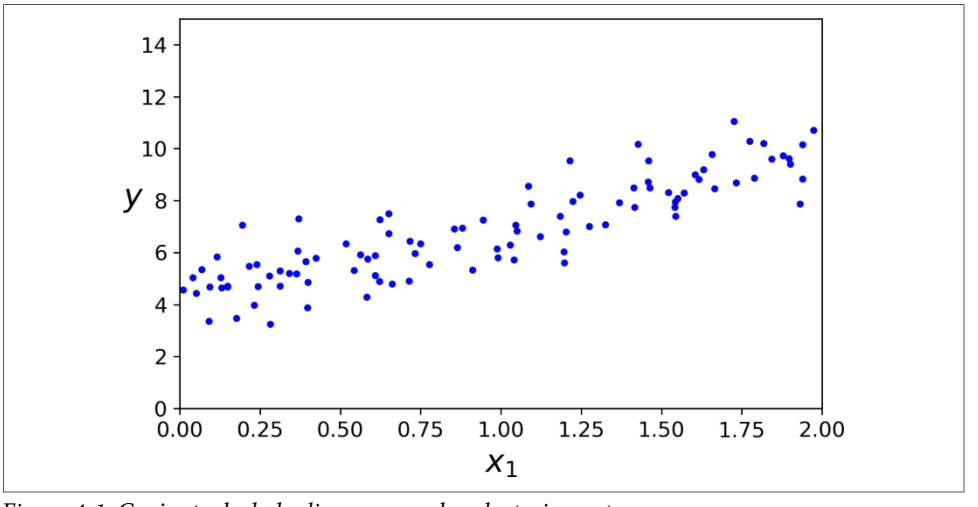

Figura 4-1. Conjunto de dados lineares gerados aleatoriamente

Agora vamos calcular $\hat{\boldsymbol{\theta}}$ usando a equação normal. Usaremos a função `inv()` do módulo Linear Algebra do NumPy (`np.linalg`) para calcular o inverso de uma matriz e o método `dot()` para multiplicação de matriz:

```
X_b= np.c_[np.ones((100, 1)), X] # adicione x0= 1 a cada instância
theta_best=  np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```

A função real que usamos para gerar os dados é $y = 4 + 3x_1 +$ ruído gaussiano. Vamos ver o que a equação encontrou:

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

Esperávamos que $\theta_0 = 4$ e $\theta_1 = 3$ em vez de $\theta_0 = 4{,}215$ e $\theta_1 = 2{,}770$. Bastante próximo, mas o ruído tornou impossível recuperar os parâmetros exatos da função original.

Agora você pode fazer previsões usando $\hat{\boldsymbol{\theta}}$:

```
>>> X_new=  np.array([[0], [2]])
>>> X_new_b= np.c_[np.ones((2, 1)), X_new] # adicione x0= 1 a cada instância
>>> y_predict=  X_new_b.dot(theta_best)
>>> y_predict array([[4.21509616],
       [9.75532293]])
```

Vamos plotar as previsões desse modelo (Figura 4-2):

```
plt.plot(X_new, y_predict, "r-")
plt.plot(X, y, "b.")
```

```
plt.axis([0, 2, 0, 15]) plt.show()
```

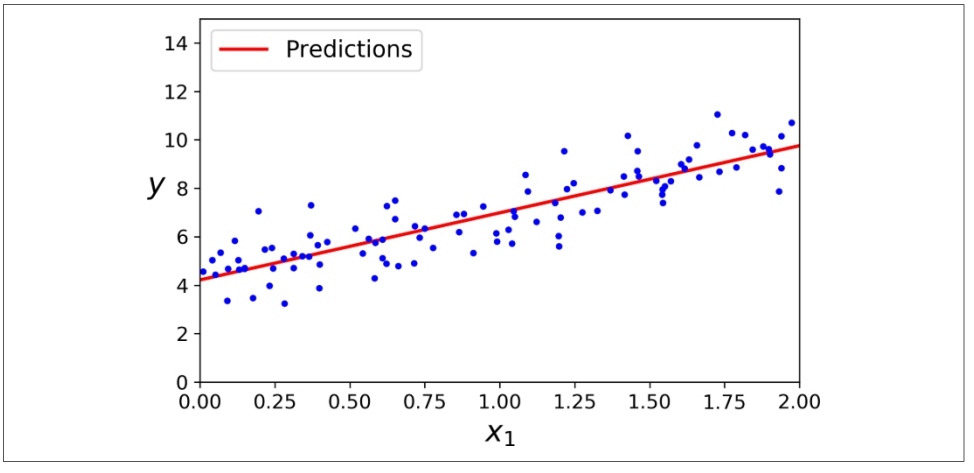

Figura 4-2. Previsões do modelo de regressão linear

A execução da regressão linear usando o Scikit-Learn é bastante simples:[3]

```
>>> from sklearn.linear_model import LinearRegression
>>> lin_reg=  LinearRegression()
>>> lin_reg.fit(X, y)
>>> lin_reg.intercept_, lin_reg.coef_
(matriz([4.21509616]), matriz([[2.77011339]]))
>>> lin_reg.predict(X_new)
array([[4.21509616],
       [9.75532293]])
```

A classe `LinearRegression` é baseada na função `scipy.linalg.lstsq()` (o nome significa "mínimos quadrados"), que você pode chamar diretamente:

```
>>> theta_best_svd, residuals, rank, s=  np.linalg.lstsq(X_b, y, rcond= 1e-6)
>>> theta_best_svd
array([[4.21509616],
       [2.77011339]])
```

Essa função calcula $\mathbf{\theta}\ \mathbf{X}= {}^{+}\hat{}\ \mathrm{y}$, em que $^{+}$ é o *pseudoinverso* de **X** (especificamente o inverso de Moore-Penrose). Você pode usar `np.linalg.pinv()` para calcular o pseudoinverso diretamente:

```
>>> np.linalg.pinv(X_b).dot(y)
array([[4.21509616],
       [2.77011339]])
```

3 Observe que o Scikit-Learn separa o termo de viés (`intercept_`) dos pesos dos recursos (`coef_`).

O próprio pseudoinverso é calculado usando uma técnica de fatoração de matriz padrão chamada *Decomposição de Valor Singular* (SVD), que pode decompor a matriz do conjunto de treinamento $\mathbf{X}$ na multiplicação da matriz de três matrizes $\mathbf{U}$ $\mathbf{\Sigma}$ $\mathbf{V}^{(T)}$ (consulte `numpy.linalg.svd()`). A pseudoinversa é calculada como $\mathbf{X}$ $\mathbf{V\Sigma}^{+}$ = $^{+U(T)}$. Para calcular a matriz $\mathbf{\Sigma}^{+}$, o algoritmo pega $\mathbf{\Sigma}$ e define como zero todos os valores menores do que um pequeno valor de limiar, depois substitui todos os valores diferentes de zero por seu inverso e, por fim, transpõe a matriz resultante. Essa abordagem é mais eficiente do que calcular a Equação Normal, além de lidar bem com os casos extremos: de fato, a Equação Normal pode não funcionar se a matriz $\mathbf{X}^{T}\mathbf{X}$ não for invertível (ou seja, singular), como se $m<$ n ou se alguns recursos forem redundantes, mas o pseudoinverso é sempre definido.

Complexidade computacional

A Equação Normal calcula o inverso de $\mathbf{X}^{(T)\,X}$, que é uma matriz $(n + 1) \times (n + 1)$ (em que n é o número de recursos). A *complexidade computacional* da inversão dessa matriz é normalmente de $O(n^{2,4})$ a $O(n^{3})$ (dependendo da implementação). Em outras palavras, se você dobrar o número de recursos, multiplicará o tempo de computação por aproximadamente $2^{2,4}$= 5,3 a 2^{3}= 8.

A abordagem SVD usada pela classe `LinearRegression` do Scikit-Learn é aproximadamente $O(n^{2)}$. Se você dobrar o número de recursos, o tempo de computação será multiplicado por aproximadamente 4.

Tanto a abordagem da Equação Normal quanto a da SVD ficam muito lentas quando o número de recursos aumenta (por exemplo, 100.000). Pelo lado positivo, ambas são lineares com relação ao número de instâncias no conjunto de treinamento (são $O(m)$), portanto, lidam com conjuntos de treinamento grandes de forma eficiente, desde que caibam na memória.

Além disso, depois de treinar o modelo de regressão linear (usando a equação normal ou qualquer outro algoritmo), as previsões são muito rápidas: a complexidade computacional é linear em relação ao número de instâncias em que você deseja fazer previsões e ao número de recursos. Em outras palavras, fazer previsões com o dobro de instâncias (ou o dobro de recursos) levará aproximadamente o dobro do tempo.

Agora veremos maneiras bem diferentes de treinar um modelo de regressão linear, mais adequadas para casos em que há um grande número de recursos ou instâncias de treinamento demais para caber na memória.

Gradiente Descendente

O Gradient Descent é um algoritmo de otimização muito genérico capaz de encontrar soluções ideais para uma ampla gama de problemas. A ideia geral do Gradient Descent é ajustar os parâmetros iterativamente para minimizar uma função de custo.

Suponha que você esteja perdido nas montanhas em meio a uma densa neblina; você só consegue sentir a inclinação do solo sob seus pés. Uma boa estratégia para chegar rapidamente ao fundo do vale é descer na direção do declive mais acentuado. É exatamente isso que o Gradient Descent faz: ele mede o gradiente local da função de erro com relação ao vetor de parâmetros $\boldsymbol{\theta}$ e segue na direção do gradiente descendente. Quando o gradiente é zero, você atingiu um mínimo!

Em termos concretos, você começa preenchendo $\boldsymbol{\theta}$ com valores aleatórios (isso é chamado de *inicialização aleatória*) e, em seguida, melhora-o gradualmente, dando um passo de bebê de cada vez, cada passo tentando diminuir a função de custo (por exemplo, o MSE), até que o algoritmo *converge* para um mínimo (veja a Figura 4-3).

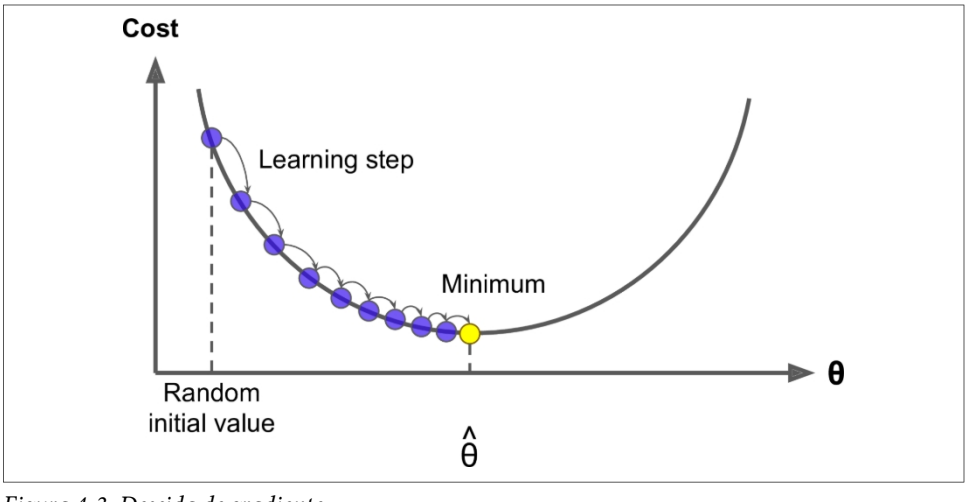

Figura 4-3. Descida de gradiente

Um parâmetro importante no Gradient Descent é o tamanho das etapas, determinado pelo hiperparâmetro *da taxa de aprendizado*. Se a taxa de aprendizagem for muito pequena, o algoritmo terá de passar por muitas iterações para convergir, o que levará muito tempo (veja a Figura 4-4).

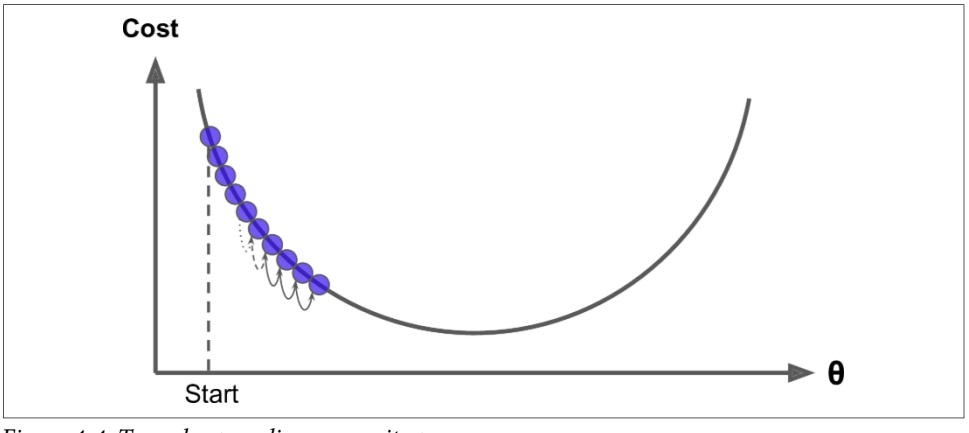

Figura 4-4. Taxa de aprendizagem muito pequena

Por outro lado, se a taxa de aprendizado for muito alta, você poderá atravessar o vale e acabar do outro lado, possivelmente ainda mais alto do que estava antes. Isso pode fazer com que o algoritmo divirja, com valores cada vez maiores, não conseguindo encontrar uma boa solução (veja a Figura 4-5).

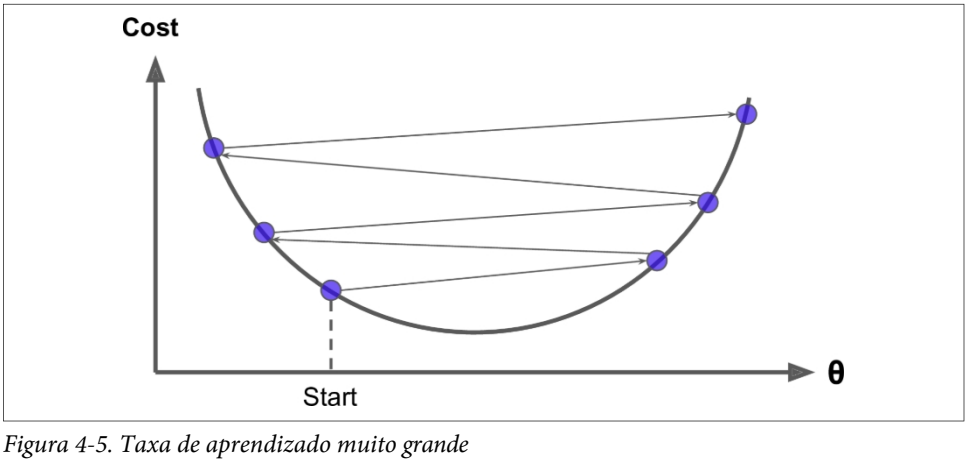

Figura 4-5. Taxa de aprendizado muito grande

Por fim, nem todas as funções de custo se parecem com tigelas regulares e bonitas. Pode haver buracos, cumes, platôs e todos os tipos de terrenos irregulares, o que dificulta muito a convergência para o mínimo. A Figura 4-6 mostra os dois principais desafios do Gradient Descent: se a inicialização do ran-dom iniciar o algoritmo à esquerda, ele convergirá para um *mini-mínimo local*, que não é tão bom quanto o *mínimo global*. Se começar à direita, levará muito tempo para cruzar o platô e, se parar muito cedo, nunca alcançará o mínimo global.

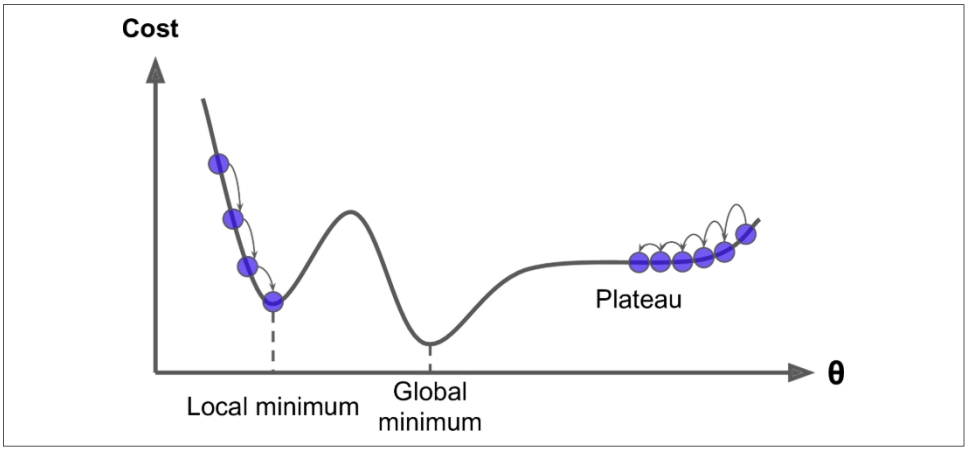

Figura 4-6. Armadilhas do Gradient Descent

Felizmente, a função de custo MSE para um modelo de regressão linear é uma *função convexa*, o que significa que, se você escolher dois pontos quaisquer na curva, o segmento de linha que os une nunca cruzará a curva. Isso implica que não há mínimos locais, apenas um mínimo global. Ela também é uma função contínua com uma inclinação que nunca muda abruptamente.[4]Esses dois fatos têm uma grande consequência: É garantido que o Gradient Descent se aproxime arbitrariamente do mínimo global (se você esperar o tempo suficiente e se a taxa de aprendizado não for muito alta).

De fato, a função de custo tem a forma de uma tigela, mas pode ser uma tigela alongada se os recursos tiverem escalas muito diferentes. A Figura 4-7 mostra o Gradient Descent em um conjunto de treinamento em que os recursos 1 e 2 têm a mesma escala (à esquerda) e em um conjunto de treinamento em que o recurso 1 tem valores muito menores do que o recurso 2 (à direita).[5]

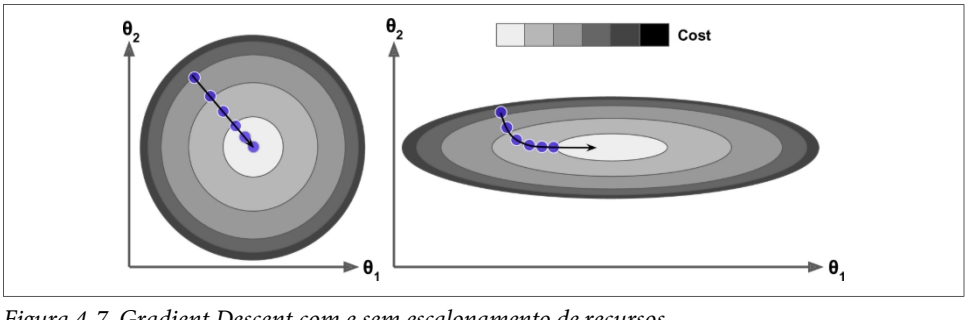

Figura 4-7. Gradient Descent com e sem escalonamento de recursos

4 Tecnicamente falando, sua derivada é *Lipschitz contínua.*

5 Como a característica 1 é menor, é necessária uma mudança maior em $\theta_1$para afetar a função de custo, razão pela qual a tigela é alongada ao longo do eixo θ_1.

Como você pode ver, à esquerda, o algoritmo Gradient Descent vai direto em direção ao mínimo, alcançando-o rapidamente, enquanto à direita ele vai primeiro em uma direção quase ortogonal à direção do mínimo global e termina com uma longa marcha por um vale quase plano. Ele acabará atingindo o mínimo, mas levará muito tempo.

Ao usar o Gradient Descent, você deve garantir que todos os recursos tenham uma escala semelhante (por exemplo, usando a classe `StandardScaler` do Scikit-Learn), caso contrário, a convergência levará muito mais tempo.

Esse diagrama também ilustra o fato de que treinar um modelo significa procurar uma combinação de parâmetros de modelo que minimize uma função de custo (no conjunto de treinamento). É uma pesquisa no *espaço de parâmetros* do modelo: quanto mais parâmetros um modelo tiver, mais dimensões esse espaço terá e mais difícil será a pesquisa: procurar uma agulha em um palheiro de 300 dimensões é muito mais complicado do que em três dimensões. Felizmente, como a função de custo é convexa no caso da Regressão Linear, a agulha está simplesmente no fundo da tigela.

Descida de gradiente em lote

Para implementar o Gradient Descent, você precisa calcular o gradiente da função de custo em relação a cada parâmetro do modelo θ_j. Em outras palavras, é preciso calcular o quanto a função de custo mudará se você alterar $\theta_j$apenas um pouco. Isso é chamado de *derivada parcial*. É como perguntar "qual é a inclinação da montanha sob meus pés se eu estiver voltado para o leste?" e depois fazer a mesma pergunta voltado para o norte (e assim por diante para todas as outras dimensões, se você puder imaginar um universo com mais de três dimensões). A Equação 4-5 calcula a derivada parcial da função de custo com relação ao parâmetro-ter θ_j, observada em $\frac{\partial}{\partial \theta^j}\text{MSE}(\boldsymbol{\theta})$.

Equação 4-5. Derivadas parciais da função de custo

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

Em vez de calcular essas derivadas parciais individualmente, você pode usar a Equação 4-6 para calcular todas de uma só vez. O vetor de gradiente, conhecido como $\nabla_{\theta}\text{MSE}(\boldsymbol{\theta})$, contém todas as derivadas parciais da função de custo (uma para cada parâmetro do modelo).

Equação 4-6. Vetor de gradiente da função de custo

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

Observe que essa fórmula envolve cálculos sobre todo o conjunto de treinamento **X**, em cada etapa do Gradient Descent! É por isso que o algoritmo é chamado de *Batch Gradient Descent*: ele usa todo o lote de dados de treinamento em cada etapa (na verdade, *Full Gradient Descent* provavelmente seria um nome melhor). Como resultado, ele é terrivelmente lento em conjuntos de treinamento muito grandes (mas veremos algoritmos de gradiente descendente muito mais rápidos em breve). No entanto, o Gradient Descent se adapta bem ao número de recursos; treinar um modelo de regressão linear quando há centenas de milhares de recursos é muito mais rápido usando o Gradient Descent do que usando a Equação Normal ou a decomposição SVD.

Quando você tiver o vetor de gradiente, que aponta para cima, basta seguir na direção oposta para descer. Isso significa subtrair $\nabla_{\boldsymbol{\theta}}\mathrm{MSE}(\boldsymbol{\theta})$ de $\boldsymbol{\theta}$. É aqui que a taxa de aprendizado η entra em ação:[6] multiplique o vetor de gradiente por η para determinar o tamanho da etapa de descida (Equação 4-7).

Equação 4-7. Etapa de descida do gradiente

$$\boldsymbol{\theta}^{(\text{próximo etapa})} = \boldsymbol{\theta} - \eta\nabla_{\boldsymbol{\theta}}\mathrm{MSE}(\boldsymbol{\theta})$$

Vamos dar uma olhada em uma implementação rápida desse algoritmo:

```
eta= 0.1 # taxa de aprendizado
n_iterações=  1000
m=  100

theta= np.random.randn(2,1) # inicialização aleatória

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

6 Eta (η) é a sétima letra do alfabeto grego.

Isso não foi muito difícil! Vamos dar uma olhada no `theta` resultante:

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

Ei, isso é exatamente o que a Equação Normal encontrou! O Gradient Descent funcionou perfeitamente. Mas e se você tivesse usado uma taxa de aprendizado `eta` diferente? A Figura 4-8 mostra as 10 primeiras etapas do Gradient Descent usando três taxas de aprendizado diferentes (a linha tracejada representa o ponto inicial).

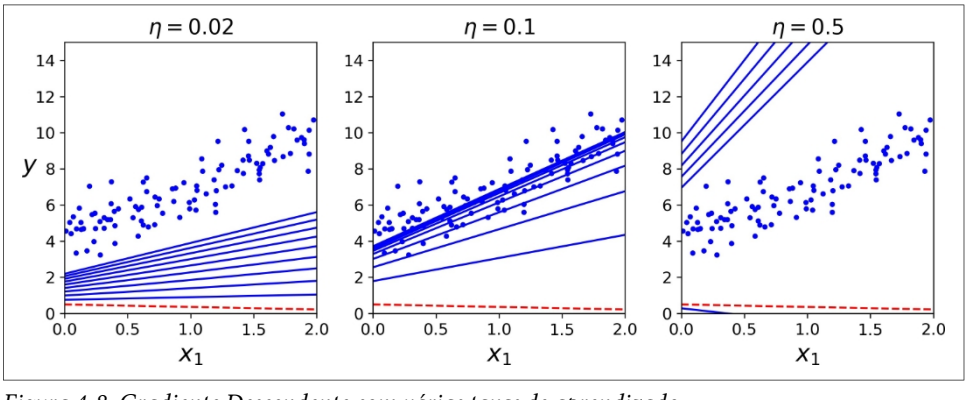

Figura 4-8. Gradiente Descendente com várias taxas de aprendizado

À esquerda, a taxa de aprendizado é muito baixa: o algoritmo acabará chegando à solução, mas levará muito tempo. No meio, a taxa de aprendizado parece muito boa: em apenas algumas iterações, ele já convergiu para a solução. À direita, a taxa de aprendizado é muito alta: o algoritmo diverge, pulando por todos os lados e, na verdade, ficando cada vez mais longe da solução a cada etapa.

Para encontrar uma boa taxa de aprendizado, você pode usar a pesquisa em grade (consulte o Capítulo 2). Entretanto, talvez você queira limitar o número de iterações para que a busca em grade possa eliminar os modelos que demoram muito para convergir.

Você pode se perguntar como definir o número de iterações. Se ele for muito baixo, você ainda estará longe da solução ideal quando o algoritmo parar, mas se for muito alto, você perderá tempo enquanto os parâmetros do modelo não mudarem mais. Uma solução simples é definir um número muito grande de iterações, mas interromper o algoritmo quando o vetor de gradiente se tornar minúsculo, ou seja, quando sua norma se tornar menor do que um número minúsculo ϵ (chamado de *tolerância*), pois isso acontece quando o Gradient Descent (quase) atingiu o mínimo.

Taxa de convergência

Quando a função de custo é convexa e sua inclinação não muda abruptamente (como é o caso da função de custo MSE), o Batch Gradient Descent com uma taxa de aprendizado fixa acabará convergindo para a solução ideal, mas talvez seja necessário esperar um pouco: pode levar $O(1/\epsilon)$ iterações para atingir o ideal em um intervalo de ϵ, dependendo da forma da função de custo. Se você dividir a tolerância por 10 para obter uma solução mais precisa, o algoritmo poderá ter de ser executado cerca de 10 vezes mais.

Descida de gradiente estocástica

O principal problema do Batch Gradient Descent é o fato de que ele usa todo o conjunto de treinamento para calcular os gradientes a cada etapa, o que o torna muito lento quando o conjunto de treinamento é grande. No extremo oposto, o *Stochastic Gradient Descent* escolhe apenas uma instância aleatória no conjunto de treinamento em cada etapa e calcula os gradientes com base apenas nessa única instância. Obviamente, isso torna o algoritmo muito mais rápido, pois ele tem muito poucos dados para manipular a cada iteração. Também possibilita o treinamento em conjuntos de treinamento enormes, já que apenas uma instância precisa estar na memória a cada iteração (o SGD pode ser implementado como um algoritmo fora do núcleo.[7])

Por outro lado, devido à sua natureza estocástica (ou seja, aleatória), esse algoritmo é muito menos regular do que o Batch Gradient Descent: em vez de diminuir suavemente até atingir o mínimo, a função de custo vai saltar para cima e para baixo, diminuindo apenas na média. Com o tempo, ela acabará muito próxima do mínimo, mas, quando chegar lá, continuará a oscilar, nunca se estabilizando (veja a Figura 4-9). Portanto, quando o algoritmo parar, os valores finais dos parâmetros serão bons, mas não ideais.

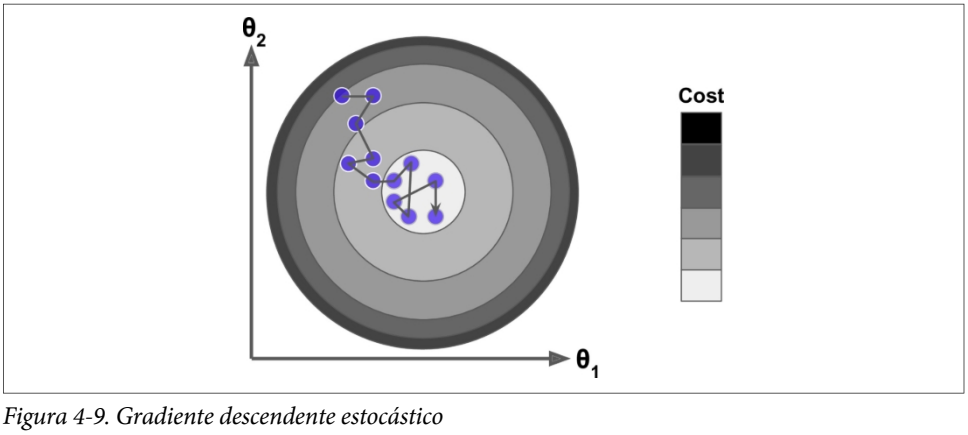

Figura 4-9. Gradiente descendente estocástico

7 Os algoritmos fora do núcleo são discutidos no Capítulo 1.

Quando a função de custo é muito irregular (como na Figura 4-6), isso pode ajudar o algoritmo a sair dos mínimos locais, de modo que o Stochastic Gradient Descent tem mais chance de encontrar o mínimo global do que o Batch Gradient Descent.

Portanto, a aleatoriedade é boa para escapar dos ótimos locais, mas é ruim porque significa que o algoritmo nunca pode se estabelecer no mínimo. Uma solução para esse dilema é reduzir gradualmente a taxa de aprendizado. As etapas começam grandes (o que ajuda a progredir rapidamente e a escapar de mínimos locais), depois ficam cada vez menores, permitindo que o algoritmo se estabeleça no mínimo global. Esse processo é semelhante ao *recozimento simulado*, um algoritmo inspirado no processo de recozimento na metalurgia, em que o metal fundido é resfriado lentamente. A função que determina a taxa de aprendizado em cada iteração é chamada de *programação de aprendizado*. Se a taxa de aprendizado for reduzida muito rapidamente, você poderá ficar preso em um mínimo local ou até mesmo acabar congelado no meio do caminho para o mínimo. Se a taxa de aprendizado for reduzida muito lentamente, você poderá saltar em torno do mínimo por um longo período e acabar com uma solução abaixo do ideal se interromper o treinamento muito cedo.

Esse código implementa o Stochastic Gradient Descent usando um cronograma de aprendizado simples:

```
n_epochs=  50
t0, t1= 5, 50 # hiperparâmetros da programação de aprendizado

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

theta= np.random.randn(2,1) # inicialização aleatória

for epoch in range(n_epochs):
    for i in 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 convenção, iteramos por rodadas de *m* iterações; cada rodada é chamada de *época*. Enquanto o código do Batch Gradient Descent iterou 1.000 vezes em todo o conjunto de treinamento, esse código percorre o conjunto de treinamento apenas 50 vezes e chega a uma solução razoavelmente boa:

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

A Figura 4-10 mostra as primeiras 20 etapas do treinamento (observe como as etapas são irregulares).

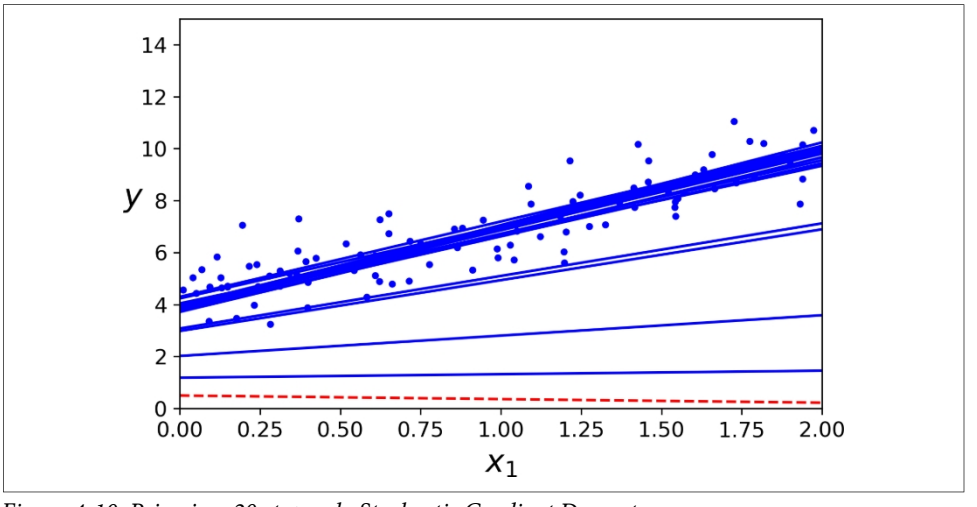

Figura 4-10. Primeiras 20 etapas do Stochastic Gradient Descent

Observe que, como as instâncias são selecionadas aleatoriamente, algumas instâncias podem ser selecionadas várias vezes por época, enquanto outras podem não ser selecionadas. Se você quiser ter certeza de que o algoritmo percorrerá todas as instâncias em cada época, outra abordagem é embaralhar o conjunto de treinamento (certificando-se de embaralhar os recursos de entrada e os rótulos em conjunto), depois percorrê-lo instância por instância, depois embaralhá-lo novamente e assim por diante. No entanto, isso geralmente converge mais lentamente.

Ao usar o Stochastic Gradient Descent, as instâncias de treinamento devem ser independentes e identicamente distribuídas (IID), para garantir que os parâmetros sejam puxados em direção ao ótimo global, em média. Uma maneira simples de garantir isso é embaralhar as instâncias durante o treinamento (por exemplo, escolher cada instância aleatoriamente ou embaralhar o conjunto de treinamento no início de cada época). Se você não fizer isso, por exemplo, se as instâncias forem classificadas por rótulo, o SGD começará a otimizar para um rótulo, depois para o próximo e assim por diante, e não se estabelecerá próximo ao mínimo global.

Para executar a regressão linear usando o SGD com o Scikit-Learn, você pode usar a classe `SGDRe gressor`, cujo padrão é otimizar a função de custo de erro quadrático. O código a seguir é executado por no máximo 1.000 épocas (`max_iter=1000`) ou até que a perda caia para menos de 1e-3 durante uma época (`tol=1e-3`), começando com uma taxa de aprendizado de 0,1 (`eta0=0,1`), usando o cronograma de aprendizado padrão (diferente do anterior) e não usa nenhuma regularização (`penalty=None`; mais detalhes sobre isso em breve):

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

Mais uma vez, você encontra uma solução muito próxima à retornada pela Equação Normal:

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

Descida de gradiente em miniatura

O último algoritmo de gradiente descendente que examinaremos é o chamado Mini-batch *Gradient Descent*. É muito simples de entender quando se conhece o Batch Gradient Descent e o Stochastic Gradient Descent: em cada etapa, em vez de calcular os gradientes com base no conjunto completo de treinamento (como no Batch GD) ou com base em apenas uma instância (como no Stochastic GD), o Mini-batch GD calcula os gradientes em pequenos conjuntos aleatórios de instâncias chamados de *minibatches*. A principal vantagem do Mini-batch GD em relação ao Stochastic GD é que você pode obter um aumento de desempenho com a otimização de hardware das operações de matriz, especialmente ao usar GPUs.

O progresso do algoritmo no espaço de parâmetros é menos errático do que com o SGD, especialmente com minilotes razoavelmente grandes. Como resultado, o Mini-batch GD acabará se aproximando um pouco mais do mínimo do que o SGD. Mas, por outro lado, pode ser mais difícil para ele escapar de mínimos locais (no caso de problemas que sofrem de mínimos locais, ao contrário da regressão linear, como vimos anteriormente). A Figura 4-11 mostra os caminhos percorridos pelos três algoritmos de gradiente descendente no espaço de parâmetros durante o treinamento. Todos eles terminam perto do mínimo, mas o caminho do Batch GD de fato para no mínimo, enquanto o Stochastic GD e o Mini-batch GD continuam a caminhar. No entanto, não se esqueça de que o Batch GD leva muito tempo para dar cada passo, e o Stochastic GD e o Mini-batch GD também atingiriam o mínimo se você usasse um bom cronograma de aprendizado.

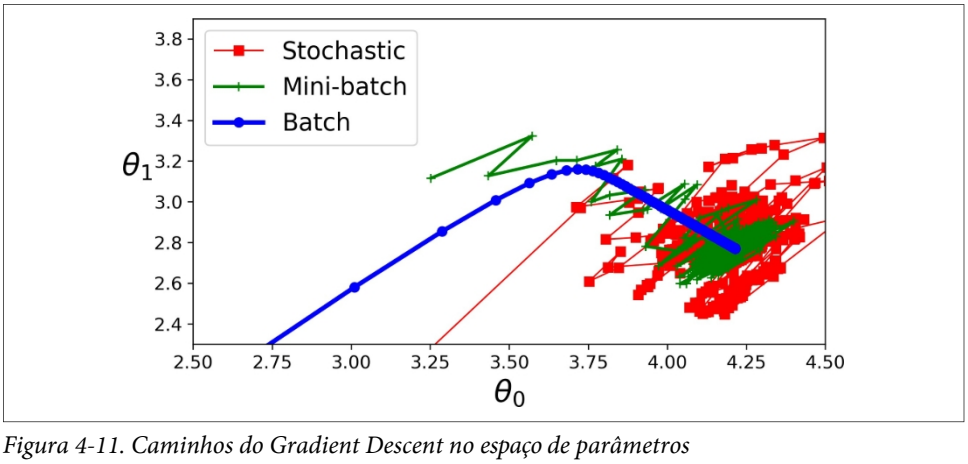

Figura 4-11. Caminhos do Gradient Descent no espaço de parâmetros

Vamos comparar os algoritmos que discutimos até agora para Regressão Linear[8] (lembre-se de que *m* é o número de instâncias de treinamento e *n* é o número de recursos); consulte a Tabela 4-1.

Tabela 4-1. Comparação de algoritmos para regressão linear

Algoritmo	Grande *m*	Suporte fora do núcleo	Grande *n*	Hiperparâmetros	Escala necessária	Scikit-Learn
Equação normal	Rápida	Não	Lenta	0	Não	n/d
SVD	Rápido	Não	Lento	0	Não	`LinearRegression` (regressão linear)
Lote GD	Lento	Não	Rápida	2	Sim	`SGDRegressor`
GD estocástico	Rápido	Sim	Rápido	≥2	Sim	`SGDRegressor`
GD de mini-lote	Rápido	Sim	Rápido	≥2	Sim	`SGDRegressor`

Quase não há diferença após o treinamento: todos esses algoritmos terminam com modelos muito semelhantes e fazem previsões exatamente da mesma maneira.

Regressão polinomial

E se seus dados forem, na verdade, mais complexos do que uma simples linha reta? Surpreendentemente, você pode usar um modelo linear para ajustar dados não lineares. Uma maneira simples de fazer isso é adicionar potências de cada recurso como novos recursos e, em seguida, treinar um modelo linear nesse conjunto estendido de recursos. Essa técnica é chamada de *Regressão Polinomial*.

Vamos dar uma olhada em um exemplo. Primeiro, vamos gerar alguns dados não lineares, com base em uma simples
equação quadrática simples[9] (mais algum ruído; veja a 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)
```

8 Embora a Equação Normal só possa executar a Regressão Linear, os algoritmos de Descida de Gradiente podem ser usados para treinar muitos outros modelos, como veremos.

9 Uma equação quadrática tem a forma $y = ax^2 bx + + c$.

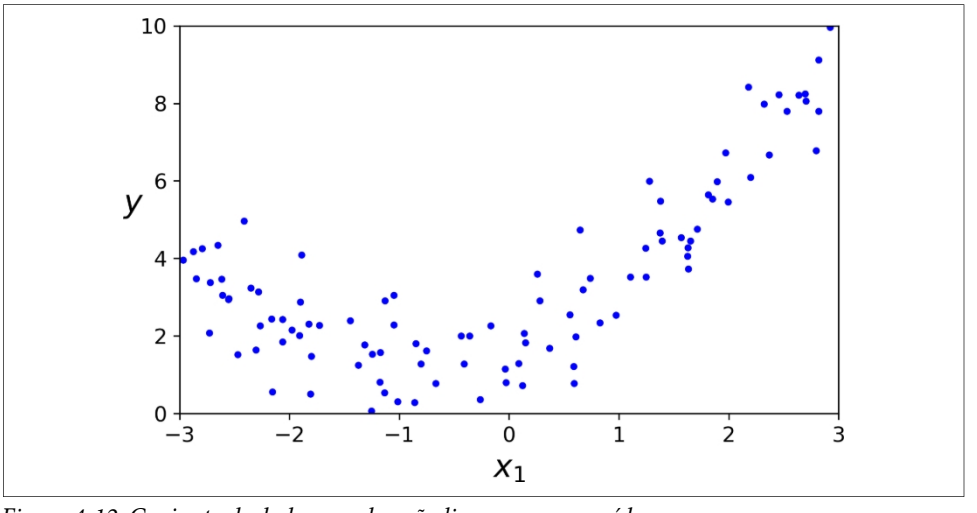

Figura 4-12. Conjunto de dados gerados não lineares e com ruído

Claramente, uma linha reta nunca se ajustará adequadamente a esses dados. Portanto, vamos usar a classe `PolynomialFeatures` do Scikit-Learn para transformar nossos dados de treinamento, adicionando o quadrado (polinômio de 2ndgraus) de cada recurso no conjunto de treinamento como novos recursos (nesse caso, há apenas um recurso):

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> poly_features= PolynomialFeatures(degree= 2, include_bias= False)
>>> X_poly= poly_features.fit_transform(X)
>>> X[0]
array([-0.75275929])
>>> X_poly[0]
array([-0.75275929, 0.56664654])
```

`X_poly` agora contém o recurso original de `X` mais o quadrado desse recurso. Agora você pode ajustar um modelo `LinearRegression` a esses dados de treinamento estendidos (Figura 4-13):

```
>>> lin_reg= LinearRegression()
>>> lin_reg.fit(X_poly, y)
>>> lin_reg.intercept_, lin_reg.coef_ (array([1.78134581]),
array([[0.93366893, 0.56456263]]))
```

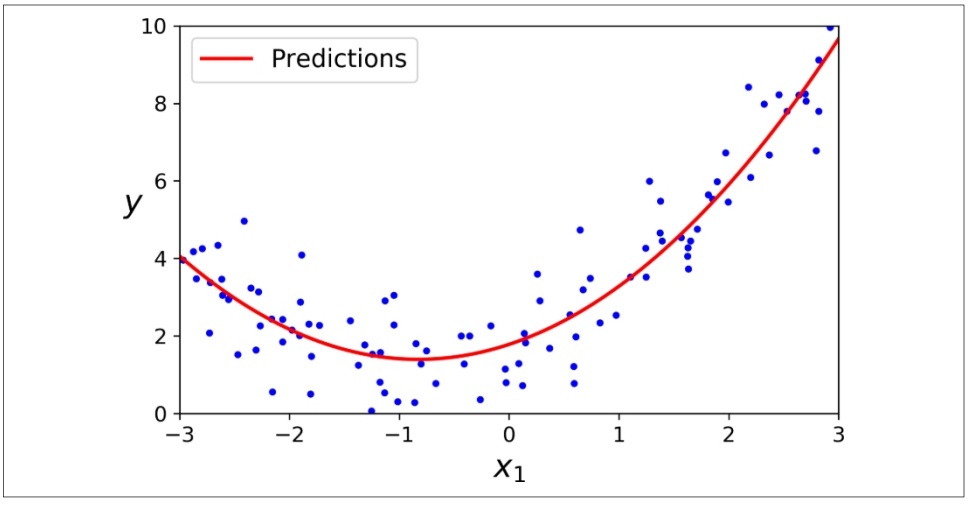

Figura 4-13. Previsões do modelo de regressão polinomial

Nada mal: o modelo estima $y = 0.56x_1^2 + 0.93x_1 + 1.78$ quando, de fato, o valor original A função era $y = 0.5x_1^2 + 1.0x_1 + 2.0 +$ ruído gaussiano.

Observe que, quando há vários recursos, a regressão polinomial é capaz de encontrar relações entre os recursos (algo que um modelo de regressão linear simples não consegue fazer). Isso é possível pelo fato de que o `PolynomialFeatures` também adiciona todas as combinações de recursos até o grau determinado. Por exemplo, se houvesse dois recursos a e b, `PolynomialFeatures` com `grau=3` não apenas adicionaria os recursos a^2, a^3, b^2e b^3, mas também as combinações ab, $a^{(2)\,b}$e ab^2.

`PolynomialFeatures(degree=d)` transforma uma matriz contendo n recursos em uma matriz contendo $\frac{(n+d)!}{d!\,n!}$ recursos, em que $n!$ é o grau *fatorial* de n, igual a $1 \times 2 \times 3 \times \cdots \times n$. Cuidado com a explosão combinatória do número de recursos!

Curvas de aprendizado

Se você executar a Regressão Polinomial de alto grau, provavelmente ajustará os dados de treinamento muito melhor do que com a Regressão Linear simples. Por exemplo, a Figura 4-14 aplica um modelo polinomial de 300 graus aos dados de treinamento anteriores e compara o resultado com um modelo linear puro e um modelo quadrático (polinômio de 2ndgraus). Observe como o modelo polinomial de 300 graus se movimenta para chegar o mais próximo possível das instâncias de treinamento.

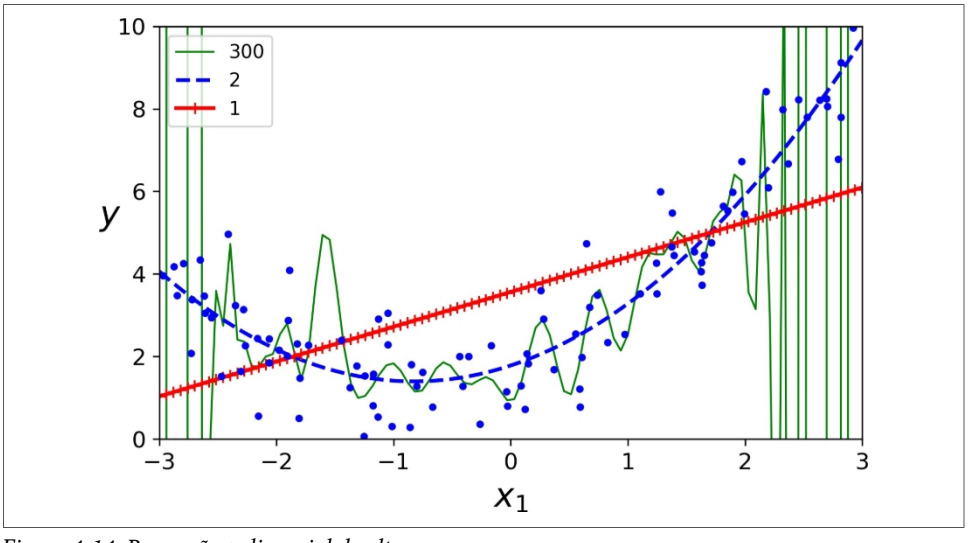

Figura 4-14. Regressão polinomial de alto grau

Obviamente, esse modelo de regressão polinomial de alto grau está se ajustando muito acima dos dados de treinamento, enquanto o modelo linear está se ajustando abaixo. O modelo que generalizará melhor nesse caso é o modelo quadrático. Isso faz sentido, pois os dados foram gerados usando um modelo quadrático, mas, em geral, você não saberá qual função gerou os dados, portanto, como pode decidir a complexidade do modelo? Como saber se o modelo está se ajustando demais ou de menos aos dados?

No Capítulo 2, você usou a validação cruzada para obter uma estimativa do desempenho de generalização de um modelo. Se um modelo tiver um bom desempenho nos dados de treinamento, mas generalizar mal de acordo com as métricas de validação cruzada, então o modelo está se ajustando demais. Se o desempenho for ruim em ambos, ele está subajustado. Essa é uma maneira de saber se um modelo é muito simples ou muito complexo.

Outra maneira é observar as *curvas de aprendizado*: são gráficos do desempenho do modelo no conjunto de treinamento e no conjunto de validação como uma função do tamanho do conjunto de treinamento (ou da iteração de treinamento). Para gerar os gráficos, basta treinar o modelo várias vezes em subconjuntos de tamanhos diferentes do conjunto de treinamento. O código a seguir define uma função que traça as curvas de aprendizado de um modelo com base em alguns dados de treinamento:

```
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

def plot_learning_curves(model, X, y):
    X_train, X_val, y_train, y_val= train_test_split(X, y, test_size= 0.2) train_errors,
    val_errors=  [], []
    for m in range(1, len(X_train)):
        model.fit(X_train[:m], y_train[:m])
        y_train_predict= model.predict(X_train[:m])
```

```
        y_val_predict= model.predict(X_val)
        train_errors.append(mean_squared_error(y_train[:m], y_train_predict))
        val_errors.append(mean_squared_error(y_val, y_val_predict))
    plt.plot(np.sqrt(train_errors), "r-+", linewidth= 2, label= "train")
    plt.plot(np.sqrt(val_errors), "b-", linewidth= 3, label= "val")
```

Vamos dar uma olhada nas curvas de aprendizado do modelo de regressão linear simples (uma linha reta; Figura 4-15):

```
lin_reg= LinearRegression()
plot_learning_curves(lin_reg, X, y)
```

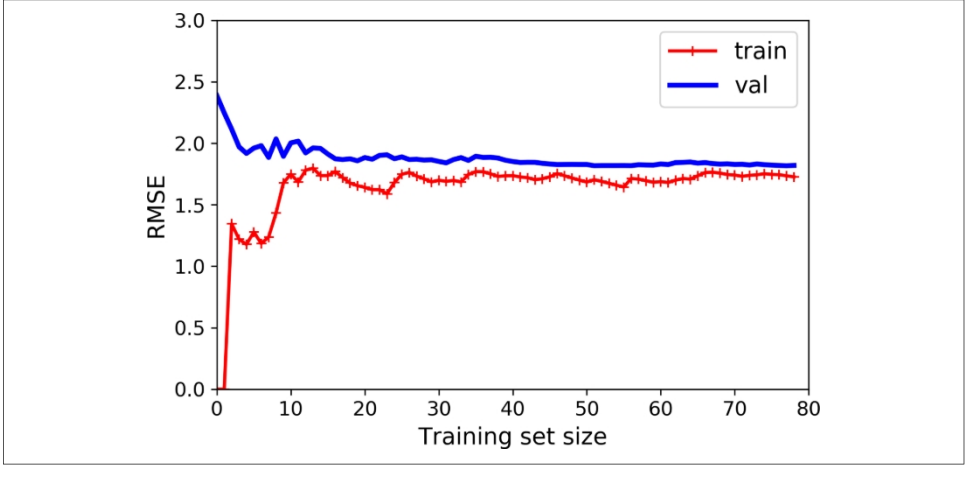

Figura 4-15. Curvas de aprendizado

Isso merece um pouco de explicação. Primeiro, vamos analisar o desempenho nos dados de treinamento: quando há apenas uma ou duas instâncias no conjunto de treinamento, o modelo pode se ajustar perfeitamente a elas, razão pela qual a curva começa em zero. Porém, à medida que novas instâncias são adicionadas ao conjunto de treinamento, torna-se impossível para o modelo ajustar-se perfeitamente aos dados de treinamento, tanto porque os dados são ruidosos quanto porque não são lineares. Portanto, o erro nos dados de treinamento aumenta até atingir um patamar, momento em que a adição de novas instâncias ao conjunto de treinamento não melhora nem piora o erro médio. Agora vamos analisar o desempenho do modelo nos dados de validação. Quando o modelo é treinado com pouquíssimas instâncias de treinamento, ele é incapaz de generalizar adequadamente, e é por isso que o erro de validação é inicialmente muito grande. Em seguida, à medida que o modelo recebe mais exemplos de treinamento, ele aprende e, portanto, o erro de validação diminui lentamente. Entretanto, mais uma vez, uma linha reta não consegue modelar bem os dados, de modo que o erro acaba em um platô, muito próximo da outra curva.

Essas curvas de aprendizado são típicas de um modelo com ajuste insuficiente. Ambas as curvas atingiram um platô; elas estão próximas e são bastante altas.

Se o seu modelo estiver se ajustando mal aos dados de treinamento, adicionar mais exemplos de treinamento não ajudará. Você precisa usar um modelo mais complexo ou criar recursos melhores.

Agora, vejamos as curvas de aprendizagem de um modelo polinomial de 10thgraus nos mesmos dados (Figura 4-16):

```
de sklearn.pipeline import Pipeline

polynomial_regression=  Pipeline([
        ("poly_features", PolynomialFeatures(degree= 10, include_bias= False)),
        ("lin_reg", LinearRegression()),
    ])

plot_learning_curves(polynomial_regression, X, y)
```

Essas curvas de aprendizado se parecem um pouco com as anteriores, mas há duas diferenças muito importantes:

- O erro nos dados de treinamento é muito menor do que no modelo de Regressão Linear.
- Há uma lacuna entre as curvas. Isso significa que o modelo tem um desempenho significativamente melhor nos dados de treinamento do que nos dados de validação, o que é a marca registrada de um modelo com ajuste excessivo. Entretanto, se você usasse um conjunto de treinamento muito maior, as duas curvas continuariam a se aproximar.

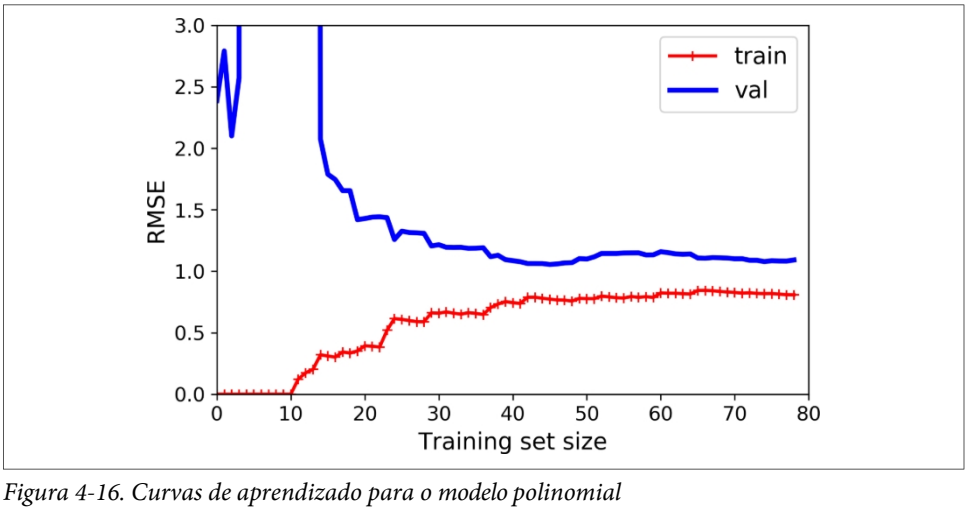

Figura 4-16. Curvas de aprendizado para o modelo polinomial

Uma maneira de melhorar um modelo com ajuste excessivo é alimentá-lo com mais dados de treinamento até que o erro de validação atinja o erro de treinamento.

A compensação de viés/variância

Um resultado teórico importante da estatística e do aprendizado de máquina é o fato de que o erro de generalização de um modelo pode ser expresso como a soma de três erros muito diferentes:

Viés
: Essa parte do erro de generalização se deve a suposições erradas, como supor que os dados são lineares quando, na verdade, são quadráticos. É mais provável que um modelo com alto viés não se ajuste aos dados de treinamento.[10]

Variância
: Essa parte se deve à sensibilidade excessiva do modelo a pequenas variações nos dados de treinamento. Um modelo com muitos graus de liberdade (como um modelo polinomial de alto grau) provavelmente terá uma variância alta e, portanto, se ajustará demais aos dados de treinamento.

Erro irredutível
: Essa parte se deve ao ruído dos próprios dados. A única maneira de reduzir essa parte do erro é limpar os dados (por exemplo, corrigir as fontes de dados, como sensores quebrados, ou detectar e remover outliers).

Aumentar a complexidade de um modelo normalmente aumentará sua variação e reduzirá sua tendência. Por outro lado, a redução da complexidade de um modelo aumenta sua tendência e reduz sua variação. É por isso que isso é chamado de tradeoff.

Modelos lineares regularizados

Como vimos nos Capítulos 1 e 2, uma boa maneira de reduzir o superajuste é regularizar o modelo (ou seja, restringi-lo): quanto menos graus de liberdade ele tiver, mais difícil será superajustar os dados. Por exemplo, uma maneira simples de regularizar um modelo polinomial é reduzir o número de graus polinomiais.

Em um modelo linear, a regularização é normalmente obtida restringindo os pesos do modelo. Analisaremos agora a Ridge Regression, a Lasso Regression e a Elastic Net, que implementam três maneiras diferentes de restringir os pesos.

10 Essa noção de viés não deve ser confundida com o termo de viés dos modelos lineares.

Regressão Ridge

A regressão Ridge (também chamada de *regularização de Tikhonov*) é uma versão regularizada da regressão Lin-ear: um *termo de regularização* igual a $\alpha\sum_{i=1}^{n}\theta_i^2$ é adicionado à função de custo.

Isso força o algoritmo de aprendizado a não apenas ajustar os dados, mas também a manter os pesos do modelo tão pequenos quanto possível. Observe que o termo de regularização só deve ser adicionado à função de custo durante o treinamento. Depois que o modelo é treinado, você deseja avaliar o desempenho do modelo usando a medida de desempenho não regularizada.

É bastante comum que a função de custo usada durante o treinamento seja diferente da medida de desempenho usada no teste. Além da regularização, outro motivo pelo qual elas podem ser diferentes é que uma boa função de custo de treinamento deve ter derivadas favoráveis à otimização, enquanto a medida de desempenho usada para teste deve ser a mais próxima possível do objetivo final. Um bom exemplo disso é um classificador treinado usando uma função de custo, como a perda de log (discutida em um momento), mas avaliado usando precisão/recuperação.

O hiperparâmetro α controla o quanto você deseja regularizar o modelo. Se $\alpha = 0$, então a Ridge Regression é apenas Linear Regression. Se α for muito grande, todos os pesos ficarão muito próximos de zero e o resultado será uma linha plana que passa pela média dos dados. A Equação 4-8 apresenta a função de custo da Regressão Ridge.[11]

Equação 4-8. Função de custo da regressão de cumeeira

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

Observe que o termo de polarização θ_0 não é regularizado (a soma começa em $i = 1$, não em 0). Se definirmos $\mathbf{w}$ como o vetor de pesos de recursos (θ_1 a θ_n), então o termo de regularização é simplesmente igual a $\frac{1}{2}(\|\mathbf{w}\|_2)^2$, em que $\|\mathbf{w}\|_2$ representa a norma ℓ_2 do vetor de pesos.[12] Para o Gradient Descent, basta adicionar $\alpha\mathbf{w}$ ao vetor de gradiente MSE (Equação 4-6).

É importante dimensionar os dados (por exemplo, usando um `StandardScaler`) antes de executar a Ridge Regression, pois ela é sensível à escala dos recursos de entrada. Isso se aplica à maioria dos modelos regularizados.

11 É comum usar a notação $J(\boldsymbol{\theta})$ para funções de custo que não têm um nome curto; usaremos essa notação com frequência no restante deste livro. O contexto deixará claro qual função de custo está sendo discutida.

12 As normas são discutidas no Capítulo 2.

A Figura 4-17 mostra vários modelos Ridge treinados em alguns dados lineares usando diferentes valores de α. À esquerda, são usados modelos Ridge simples, o que leva a previsões lineares. À direita, os dados são primeiro expandidos usando `PolynomialFeatures(degree=10)`, depois são escalonados usando um `StandardScaler` e, por fim, os modelos Ridge são aplicados aos recursos resultantes: é a regressão polinomial com regularização Ridge. Observe como o aumento de α leva a previsões mais planas (ou seja, menos extremas, mais razoáveis); isso reduz a variação do modelo, mas aumenta sua tendência.

Assim como na regressão linear, podemos executar a regressão Ridge calculando uma equação de forma fechada ou executando a descida do gradiente. Os prós e contras são os mesmos. A Equação 4-9 mostra a solução de forma fechada (em que **A** é a *matriz identidade* [13] $(n + 1) \times (n + 1)$, exceto com um 0 na célula superior esquerda, correspondente ao termo de viés).

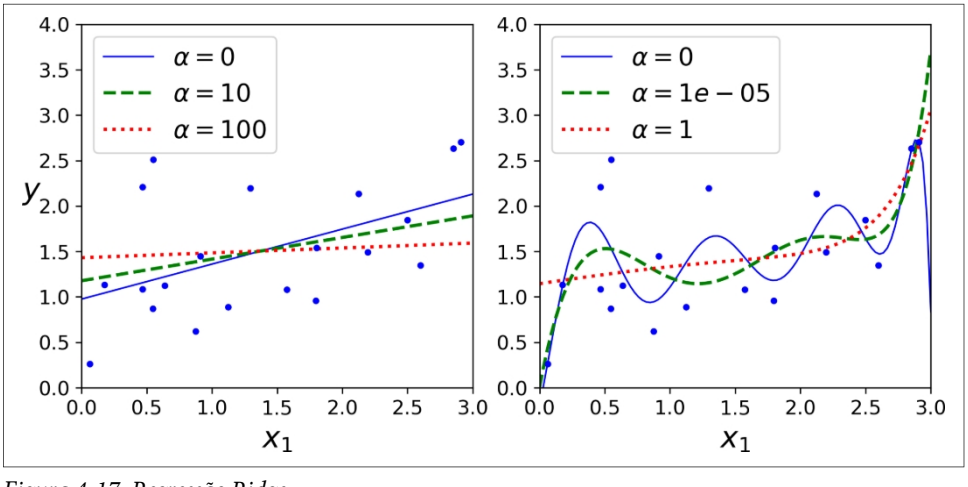

Figura 4-17. Regressão Ridge

Equação 4-9. Solução de forma fechada da regressão Ridge

$$\hat{\theta} = \left(\mathbf{X}^T \mathbf{X} + \alpha \mathbf{A} \right)^{-1} \quad \mathbf{X}^T \quad \mathbf{y}$$

Veja a seguir como executar a Ridge Regression com o Scikit-Learn usando uma solução de forma fechada (uma variante da Equação 4-9 usando uma técnica de fatoração de matriz de André-Louis Cholesky):

```
>>> from sklearn.linear_model import Ridge
>>> ridge_reg=  Ridge(alpha= 1, solver= "cholesky")
>>> ridge_reg.fit(X, y)
```

13 Uma matriz quadrada cheia de 0s, exceto por 1s na diagonal principal (de cima para baixo, da esquerda para a direita).

```
>>> ridge_reg.predict([[1.5]])
array([[1.55071465]])
```

E usando o Stochastic Gradient Descent:[14]

```
>>> sgd_reg=  SGDRegressor(penalty= "l2")
>>> sgd_reg.fit(X, y.ravel())
>>> sgd_reg.predict([[1.5]])
array([1.47012588])
```

O hiperparâmetro `de penalidade` define o tipo de termo de regularização a ser usado. Especificar `"l2"` indica que você deseja que o SGD adicione um termo de regularização à função de custo igual à metade do quadrado da norma $\ell_{(2)}$do vetor de peso: isso é simplesmente Ridge Regression.

Regressão Lasso

A Least Absolute Shrinkage and Selection Operator Regression (simplesmente chamada de *Regressão Lasso*) é outra versão regularizada da Regressão Linear: assim como a Regressão Ridge, ela adiciona um termo de regularização à função de custo, mas usa a norma $\ell_{(1)}$do vetor de peso em vez de metade do quadrado da norma $\ell_{(2)}$(consulte a Equação 4-10).

Equação 4-10. Função de custo da regressão Lasso

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

A Figura 4-18 mostra a mesma coisa que a Figura 4-17, mas substitui os modelos Ridge por modelos Lasso e usa valores α menores.

14 Como alternativa, você pode usar a classe `Ridge` com o solucionador `"sag"`. O Stochastic Average GD é uma variante do SGD. Para obter mais detalhes, consulte a apresentação "Minimizing Finite Sums with the Stochastic Average Gradient Algo-rithmMinimizando somas finitas com o algoritmo de gradiente médio estocástico" () de Mark Schmidt et al. da University of British Columbia.

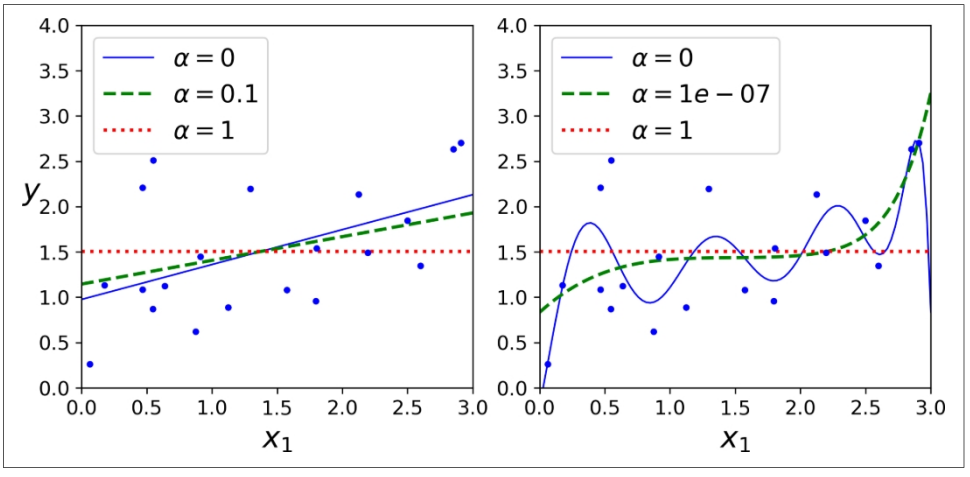

Figura 4-18. Regressão Lasso

Uma característica importante da regressão Lasso é que ela tende a eliminar completamente os pesos dos recursos menos importantes (ou seja, defini-los como zero). Por exemplo, a linha tracejada no gráfico da direita na Figura 4-18 (com $\alpha = 10^{-7}$) parece quadrática, quase linear: todos os pesos dos recursos polinomiais de alto grau são iguais a zero. Em outras palavras, a regressão Lasso realiza automaticamente a seleção de recursos e produz um *modelo esparso* (ou seja, com poucos pesos de recursos diferentes de zero).

Você pode ter uma ideia de por que isso acontece observando a Figura 4-19: no gráfico superior esquerdo, os contornos do plano de fundo (elipses) representam uma função de custo MSE não regularizada ($\alpha = 0$), e os círculos brancos mostram o caminho de gradiente descendente em lote com essa função de custo. Os contornos em primeiro plano (diamantes) representam a penalidade $\ell_{(1)}$, e os triângulos mostram o caminho BGD somente para essa penalidade ($\alpha \to \infty$). Observe como o caminho primeiro atinge $\theta_1 = 0$ e, em seguida, desce por uma calha até atingir $\theta_2 = 0$. No gráfico superior direito, os contornos representam a mesma função de custo mais uma penalidade $\ell_{(1)}$) com $\alpha = 0{,}5$. O mínimo global está no eixo $\theta_2 = 0$. O BGD primeiro atinge $\theta_2 = 0$ e, em seguida, desce pela calha até atingir o mínimo global. Os dois gráficos inferiores mostram a mesma coisa, mas usam uma penalidade de $\ell_{(2)}$. O mínimo regularizado está mais próximo de $\boldsymbol{\theta} = 0$ do que o mínimo não regularizado, mas os pesos não são totalmente eliminados.

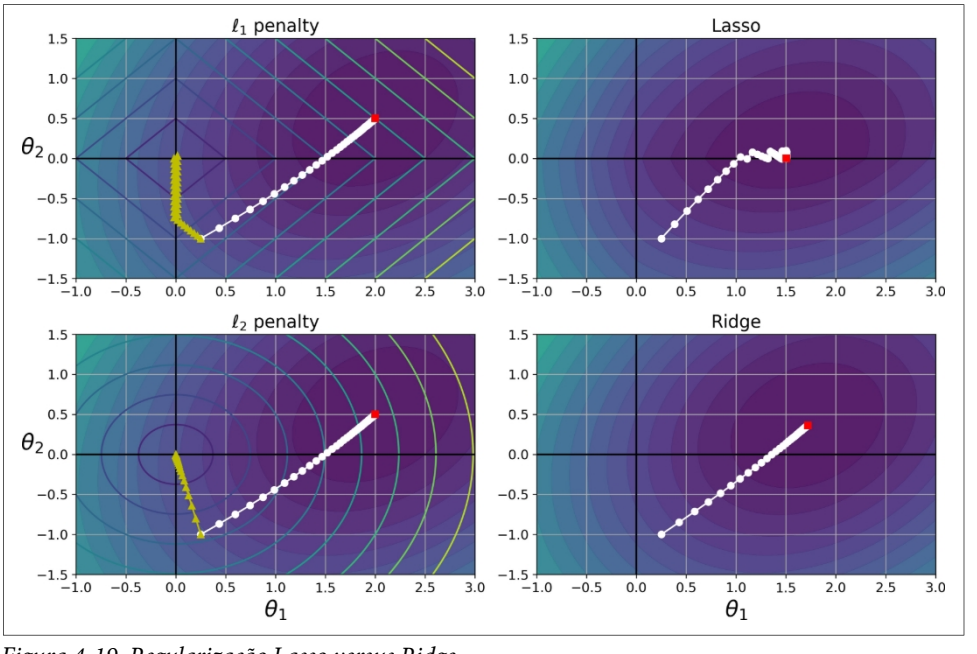

Figura 4-19. Regularização Lasso versus Ridge

Na função de custo Lasso, o caminho do BGD tende a saltar pela sarjeta no final. Isso ocorre porque a inclinação muda abruptamente em $\theta_2 = 0$. É necessário reduzir gradualmente a taxa de aprendizado para convergir de fato para o mínimo global.

A função de custo Lasso não é diferenciável em $\theta_i = 0$ (para $i = 1, 2, \cdots, n$), mas o Gradient Descent ainda funciona bem se, em vez disso, você usar um *vetor de subgradiente* $\mathbf{g}$[15] quando qualquer $\theta_i = 0$. A Equação 4-11 mostra uma equação de vetor de subgradiente que pode ser usada para o Gradient Descent com a função de custo Lasso.

Equação 4-11. Vetor de subgradiente da regressão Lasso

$$g(\boldsymbol{\theta}, J) = \nabla_{\boldsymbol{\theta}}\mathrm{MSE}(\boldsymbol{\theta}) + \alpha \begin{pmatrix} \mathrm{sinal}(\theta_1) \\ \mathrm{sinal}(\theta_2) \\ \vdots \\ \mathrm{sinal}(\theta_n) \end{pmatrix} \quad \text{em que } \mathrm{sign}(\theta_i) = \begin{cases} -1 \text{ se } \theta_i < 0 \\ 0 \text{ se } \theta_i = 0 \\ +1 \text{ se } \theta_i > 0 \end{cases}$$

15 Você pode pensar em um vetor de subgradiente em um ponto não diferenciável como um vetor intermediário entre os vetores de gradiente em torno desse ponto.

Aqui está um pequeno exemplo do Scikit-Learn usando a classe `Lasso`. Observe que, em vez disso, você poderia usar um `SGDRegressor(penalty="l1")`.

```
>>> from sklearn.linear_model import Lasso
>>> lasso_reg=  Lasso(alpha= 0.1)
>>> lasso_reg.fit(X, y)
>>> lasso_reg.predict([[1.5]])
array([1.53788174])
```

Rede elástica

A Elastic Net é um meio termo entre a regressão Ridge e a regressão Lasso. O termo de regularização é uma mistura simples dos termos de regularização de Ridge e Lasso, e você pode controlar a proporção de mistura *r*. Quando $r = 0$, a Elastic Net é equivalente à Ridge Regression, e quando $r = 1$, é equivalente à Lasso Regression (consulte a Equação 4-12).

Equação 4-12. Função de custo da Elastic Net

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

Então, quando você deve usar a Regressão Linear simples (ou seja, sem nenhuma regularização), Ridge, Lasso ou Elastic Net? Quase sempre é preferível ter pelo menos um pouco de regularização, portanto, em geral, você deve evitar a regressão linear simples. O Ridge é um bom padrão, mas se você suspeitar que apenas alguns recursos são realmente úteis, prefira o Lasso ou o Elastic Net, pois eles tendem a reduzir os pesos dos recursos inúteis a zero, conforme discutimos. Em geral, o Elastic Net é preferível ao Lasso, pois o Lasso pode se comportar de forma errática quando o número de recursos é maior que o número de instâncias de treinamento ou quando vários recursos estão fortemente correlacionados.

Aqui está um pequeno exemplo usando o `ElasticNet` do Scikit-Learn (`l1_ratio` corresponde à proporção de mistura *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 antecipada

Uma maneira muito diferente de regularizar os algoritmos de aprendizado iterativo, como o Gradient Descent, é interromper o treinamento assim que o erro de validação atingir o mínimo. Isso é chamado de *parada antecipada*. A Figura 4-20 mostra um modelo complexo (nesse caso, um modelo de regressão polinomial de alto grau) sendo treinado usando o Batch Gradient Descent. Com o passar das épocas, o algoritmo aprende e o erro de previsão (RMSE) no conjunto de treinamento diminui naturalmente, assim como o erro de previsão no conjunto de validação. No entanto,

depois de algum tempo, o erro de validação para de diminuir e, na verdade, começa a subir novamente. Isso indica que o modelo começou a se ajustar demais aos dados de treinamento. Com o early stop-ping, você simplesmente interrompe o treinamento assim que o erro de validação atinge o mínimo. Essa é uma técnica de regularização tão simples e eficiente que Geoffrey Hinton a chamou de "belo almoço grátis".

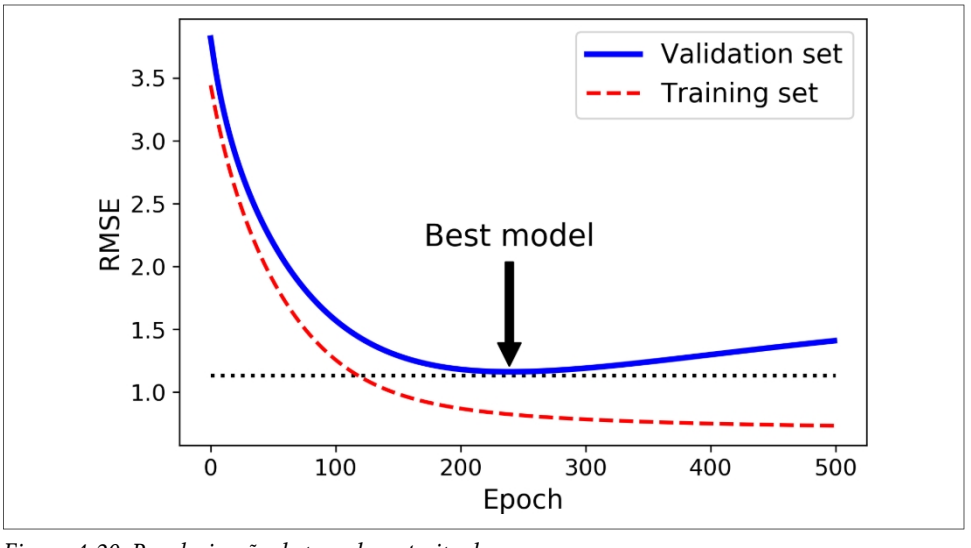

Figura 4-20. Regularização de parada antecipada

Com o Stochastic e o Mini-batch Gradient Descent, as curvas não são tão suaves, e pode ser difícil saber se você atingiu o mínimo ou não. Uma solução é parar somente depois que o erro de validação estiver acima do mínimo por algum tempo (quando você estiver confiante de que o modelo não se sairá melhor) e, em seguida, reverter os parâmetros do modelo para o ponto em que o erro de validação estava no mínimo.

Aqui está uma implementação básica de parada antecipada:

```
from sklearn.base import clone

# Preparar os dados
poly_scaler=  Pipeline([
        ("poly_features", PolynomialFeatures(degree= 90, include_bias= False)),
        ("std_scaler", StandardScaler())
    ])
X_train_poly_scaled= poly_scaler.fit_transform(X_train)
X_val_poly_scaled=  poly_scaler.transform(X_val)

sgd_reg=  SGDRegressor(max_iter= 1, tol=-np.infty, warm_start= True,
                       penalty= None, learning_rate= "constant", eta0= 0.0005)
```

```
minimum_val_error= float("inf")
best_epoch=  None
best_model=  None
para época em range(1000):
    sgd_reg.fit(X_train_poly_scaled, y_train)    # continua de onde parou
    y_val_predict=  sgd_reg.predict(X_val_poly_scaled)
    val_error= mean_squared_error(y_val, y_val_predict) if
    val_error<  minimum_val_error:
        minimum_val_error= val_error
        best_epoch=  epoch
        best_model=  clone(sgd_reg)
```

Observe que, com `warm_start=True`, quando o método `fit()` é chamado, ele continua o treinamento de onde parou, em vez de reiniciar do zero.

Regressão logística

Como discutimos no Capítulo 1, alguns algoritmos de regressão também podem ser usados para classificação (e vice-versa). *A Regressão Logística* (também chamada de *Regressão Logit*) é comumente usada para estimar a probabilidade de uma instância pertencer a uma determinada classe (por exemplo, qual é a probabilidade de esse e-mail ser spam?). Se a probabilidade estimada for maior que 50%, o modelo prevê que a instância pertence a essa classe (chamada de classe positiva, rotulada como "1"), caso contrário, prevê que não pertence (ou seja, pertence à classe negativa, rotulada como "0"). Isso o torna um classificador binário.

Estimativa de probabilidades

Então, como ele funciona? Assim como um modelo de regressão linear, um modelo de regressão logística calcula uma soma ponderada dos recursos de entrada (mais um termo de viés), mas, em vez de emitir o resultado diretamente, como faz o modelo de regressão linear, ele emite a *logística* desse resultado (consulte a Equação 4-13).

Equação 4-13. Probabilidade estimada do modelo de regressão logística (forma vetorizada)

$$\hat{p} = h_{\boldsymbol{\theta}}(\mathbf{x}) = \sigma\left(\mathbf{x}^{(T)}\boldsymbol{\theta}\right)$$

A logística - indicada como $\sigma(\cdot)$ - é uma *função sigmoide* (ou seja, *em forma de S*) que produz um número entre 0 e 1. Ela é definida conforme mostrado na Equação 4-14 e na Figura 4-21.

Equação 4-14. Função logística

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

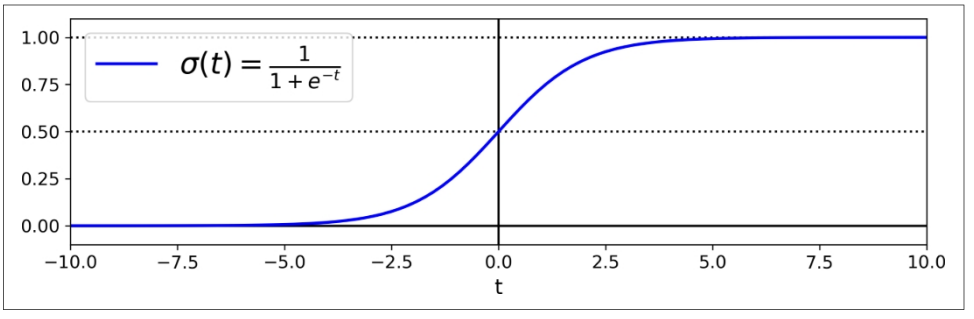

Figura 4-21. Função logística

Quando o modelo de regressão logística tiver estimado a probabilidade $p=\hat{h}_{\theta}(\mathbf{x})$ de que uma instância **x** pertença à classe positiva, ele poderá fazer sua previsão $\hat{y}$ facilmente (consulte a Equação 4-15).

Equação 4-15. Previsão do modelo de regressão logística

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

Observe que $\sigma(t) < 0{,}5$ quando $t < 0$ e $\sigma(t) \geq 0{,}5$ quando $t \geq 0$, portanto, um modelo de regressão logística prevê 1 se $\mathbf{x}^T \boldsymbol{\theta}$ for positivo e 0 se for negativo.

A pontuação *t* é frequentemente chamada de *logit*: esse nome vem do fato de que a função logit, definida como logit(p) = log(p / (1 - p)), é o inverso da função logística. De fato, se você calcular o logit da probabilidade estimada p, verá que o resultado é t. O logit também é chamado de *log-odds*, pois é o log da razão entre a probabilidade estimada para a classe positiva e a probabilidade estimada para a classe negativa.

Função de treinamento e custo

Bom, agora você sabe como um modelo de regressão logística estima as probabilidades e faz previsões. Mas como ele é treinado? O objetivo do treinamento é definir o vetor de parâmetros $\boldsymbol{\theta}$ de modo que o modelo estime altas probabilidades para instâncias positivas ($y = 1$) e baixas probabilidades para instâncias negativas ($y = 0$). Essa ideia é capturada pela função de custo mostrada na Equação 4-16 para uma única instância de treinamento **x**.

Equação 4-16. Função de custo de uma única instância de treinamento

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

Essa função de custo faz sentido porque - log(*t*) cresce muito quando *t* se aproxima de 0, portanto, o custo será grande se o modelo estimar uma probabilidade próxima de 0 para uma instância positiva e também será muito grande se o modelo estimar uma probabilidade próxima de 1 para uma instância negativa. Por outro lado, - log(*t*) é próximo de 0 quando *t* é próximo de 1, portanto, o custo será próximo de 0 se a probabilidade estimada for próxima de 0 para uma instância negativa ou próxima de 1 para uma instância positiva, que é exatamente o que queremos.

A função de custo sobre todo o conjunto de treinamento é simplesmente o custo médio sobre todas as instâncias de treinamento. Ela pode ser escrita em uma única expressão (como você pode verificar facilmente), chamada de *perda de log*, mostrada na Equação 4-17.

Equação 4-17. Função de custo da regressão logística (perda de log)

$$J(\boldsymbol{\theta}) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\left(\hat{p}^{(i)}\right) + \left(1-y^{(i)}\right)\log\left(1-\hat{p}^{(i)}\right)\right]$$

A má notícia é que não há uma equação conhecida de forma fechada para calcular o valor de $\boldsymbol{\theta}$ que minimiza essa função de custo (não há equivalente à Equação Normal). Mas a boa notícia é que essa função de custo é convexa, de modo que o Gradient Descent (ou qualquer outro algoritmo de otimização) tem a garantia de encontrar o mínimo global (se a taxa de aprendizado não for muito grande e você esperar o tempo suficiente). As derivadas parciais da função de custo com relação ao parâmetro do modelo jth $\theta_j$são dadas pela Equação 4-18.

Equação 4-18. Derivadas parciais da função de custo logístico

$$\frac{\partial}{\partial\theta_j} J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m}\left(\left(\boldsymbol{\theta}^T\mathbf{x}^{(i)} - y^{(i)}\right)x_j^{(i)}\right)$$

Essa equação se parece muito com a Equação 4-5: para cada instância, ela calcula o erro de previsão e o multiplica pelo valor do recurso jthe, em seguida, calcula a média de todas as instâncias de treinamento. Quando você tiver o vetor de gradiente contendo todas as derivadas parciais, poderá usá-lo no algoritmo Batch Gradient Descent. É isso: agora você sabe como treinar um modelo de regressão logística. Para a GD estocástica, é claro que você usaria apenas uma instância de cada vez e, para a GD em minilote, usaria um minilote de cada vez.

Limites de decisão

Vamos usar o conjunto de dados da íris para ilustrar a Regressão Logística. Esse é um conjunto de dados famoso que contém o comprimento e a largura da sépala e da pétala de 150 flores de íris de três espécies diferentes: Iris-Setosa, Iris-Versicolor e Iris-Virginica (veja a Figura 4-22).

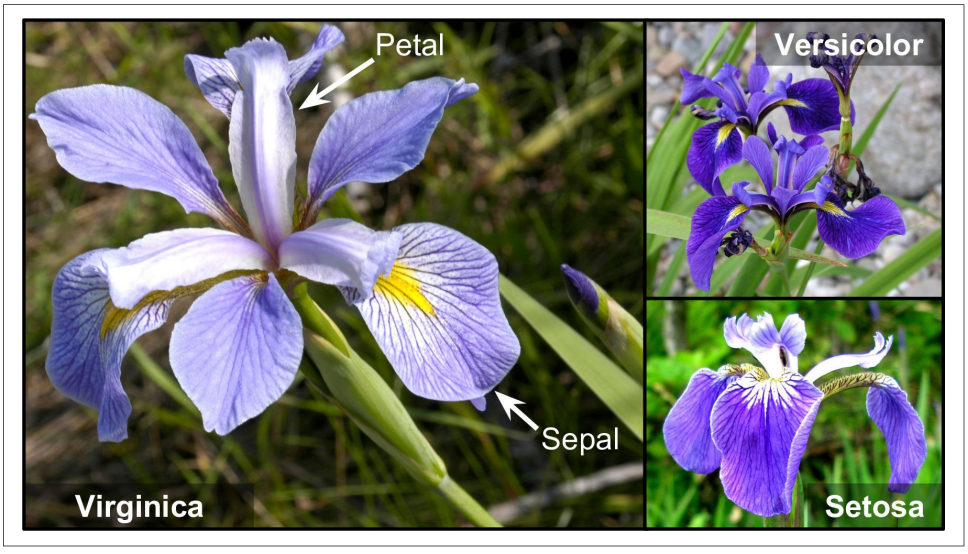

Figura 4-22. Flores de três espécies de íris[16]

Vamos tentar criar um classificador para detectar o tipo Iris-Virginica com base apenas no recurso de largura da pétala. Primeiro, vamos carregar os dados:

```
>>> from sklearn import datasets
>>> iris=  datasets.load_iris()
>>> list(iris.keys())
['data', 'target', 'target_names', 'DESCR', 'feature_names', 'filename']
>>> X= iris["data"][:, 3:] # largura da pétala
>>> y= (iris["target"]== 2).astype(np.int) # 1 if Iris-Virginica, else 0
```

Agora vamos treinar um modelo de regressão logística:

```
from sklearn.linear_model import LogisticRegression

log_reg= LogisticRegression()
log_reg.fit(X, y)
```

Vejamos as probabilidades estimadas do modelo para flores com largura de pétala variando de 0 a 3 cm [17](Figura 4-23):

```
X_new= np.linspace(0, 3, 1000).reshape(-1, 1) y_proba=
log_reg.predict_proba(X_new)
plt.plot(X_new, y_proba[:, 1], "g-", label= "Iris-Virginica")
```

16 Fotos reproduzidas das páginas correspondentes da Wikipédia. Foto de Iris-Virginica por Frank Mayfield (Creative Commons BY-SA 2.0), foto de Iris-Versicolor por D. Gordon E. Robertson (Creative Commons BY-SA 3.0) e foto de Iris-Setosa é de domínio público.

17 A função `reshape()` do NumPy permite que uma dimensão seja -1, o que significa "não especificado": o valor é inferido a partir do comprimento da matriz e das dimensões restantes.

```
plt.plot(X_new, y_proba[:, 0], "b--", label= "Not Iris-Virginica")
#+ mais código Matplotlib para deixar a imagem bonita
```

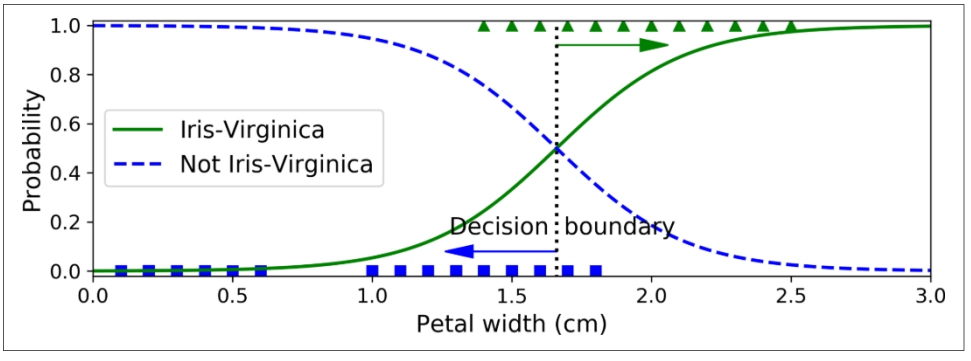

Figura 4-23. Probabilidades estimadas e limite de decisão

A largura das pétalas das flores de Iris-Virginica (representadas por triângulos) varia de 1,4 cm a 2,5 cm, enquanto as outras flores de íris (representadas por quadrados) geralmente têm uma largura de pétala menor, variando de 0,1 cm a 1,8 cm. Observe que há um pouco de sobreposição. Acima de cerca de 2 cm, o classificador está altamente confiante de que a flor é uma Iris-Virginica (ele atribui uma alta probabilidade a essa classe), enquanto abaixo de 1 cm ele está altamente confiante de que não é uma Iris-Virginica (alta probabilidade para a classe "Not Iris-Virginica"). Entre esses extremos, o classificador não tem certeza. Entretanto, se você pedir que ele preveja a classe (usando o método `predict()` em vez do método `predict_proba()`), ele retornará a classe mais provável. Portanto, há um *limite de decisão* em torno de 1,6 cm em que ambas as probabilidades são iguais a 50%: se a largura da pétala for maior que 1,6 cm, o classificador preverá que a flor é uma Iris-Virginica, caso contrário, preverá que não é (mesmo que não esteja muito confiante):

```
>>> log_reg.predict([[1.7], [1.5]])
array([1, 0])
```

A Figura 4-24 mostra o mesmo conjunto de dados, mas dessa vez exibindo dois recursos: largura e comprimento da pétala. Uma vez treinado, o classificador de regressão logística pode estimar a probabilidade de uma nova flor ser uma Iris-Virginica com base nesses dois recursos. A linha tracejada representa os pontos em que o modelo estima uma probabilidade de 50%: esse é o limite de decisão do modelo. [18]Cada linha paralela representa os pontos em que o modelo gera uma probabilidade específica, de 15% (canto inferior esquerdo) a 90% (canto superior direito). Todas as flores além da linha superior direita têm mais de 90% de chance de serem Iris-Virginica de acordo com o modelo.

18 É o conjunto de pontos **x** de modo que $\theta_0 + \theta_1 x_1 + \theta_2 x_2 = 0$, o que define uma linha reta.

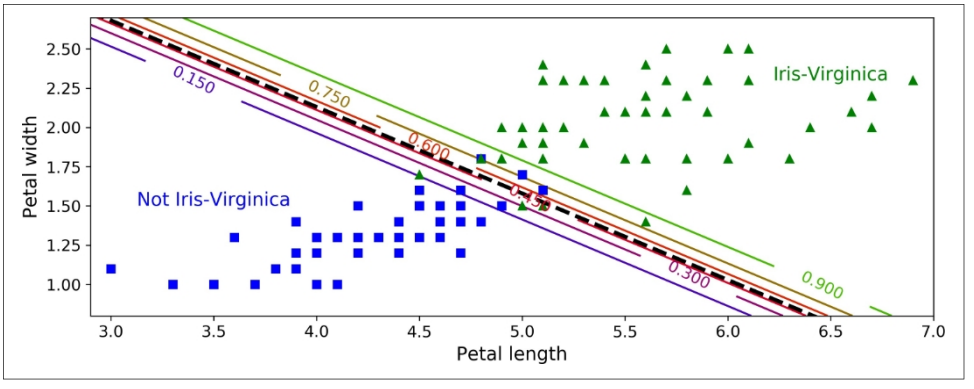

Figura 4-24. Limite de decisão linear

Assim como os outros modelos lineares, os modelos de regressão logística podem ser regularizados usando penalidades $\ell_{(1)}$ou $\ell_{(2)}$. Na verdade, o Scitkit-Learn adiciona uma penalidade $\ell_{(2)}$por padrão.

O hiperparâmetro que controla a força de regularização de um modelo Scikit-Learn `LogisticRegression` não é `alfa` (como em outros modelos lineares), mas seu inverso: `C`. Quanto maior o valor de `C`, *menos* o modelo é regularizado.

Regressão Softmax

O modelo de regressão logística pode ser generalizado para suportar várias classes diretamente, sem a necessidade de treinar e combinar vários classificadores binários (conforme discutido no Capítulo 3). Isso é chamado de *Regressão Softmax* ou *Regressão Logística Multinomial*.

A ideia é bem simples: quando é dada uma instância **x**, o modelo de regressão Softmax primeiro calcula uma pontuação $s_{(k)\,(}$**x**) para cada classe *k* e, em seguida, estima a probabilidade de cada classe aplicando a *função softmax* (também chamada de *exponencial normalizada*) às pontuações. A equação para calcular $s_{(k)\,(}$**x**) deve parecer familiar, pois é igual à equação para previsão de regressão linear (consulte a Equação 4-19).

Equação 4-19. Pontuação Softmax para a classe k

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

Observe que cada classe tem seu próprio vetor de parâmetro dedicado $\boldsymbol{\theta}^{(k)}$. Todos esses vetores são normalmente armazenados como linhas em uma *matriz de parâmetros* Θ.

Depois de calcular a pontuação de cada classe para a instância **x**, você pode estimar a probabilidade $p_k$de a instância pertencer à classe *k* executando as pontuações por meio da função softmax (Equação 4-20): ela calcula o exponencial de cada pontuação,

depois as normaliza (dividindo-as pela soma de todas as exponenciais). As pontuações são geralmente chamadas de logits ou log-odds (embora sejam, na verdade, log-odds não normalizadas).

Equação 4-20. Função Softmax

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

- K é o número de classes.
- $\mathbf{s}(\mathbf{x})$ é um vetor que contém as pontuações de cada classe para a instância $\mathbf{x}$.
- $\sigma(\mathbf{s}(\mathbf{x}))_k$ é a probabilidade estimada de que a instância $\mathbf{x}$ pertença à classe k, dadas as pontuações de cada classe para essa instância.

Assim como o classificador de regressão logística, o classificador de regressão softmax prevê a classe com a maior probabilidade estimada (que é simplesmente a classe com a maior pontuação), conforme mostrado na Equação 4-21.

Equação 4-21. Previsão do classificador Softmax Regression

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

- O operador *argmax* retorna o valor de uma variável que maximiza uma função. Nessa equação, ele retorna o valor de k que maximiza a probabilidade estimada $\sigma(\mathbf{s}(\mathbf{x}))_k$.

O classificador Softmax Regression prevê apenas uma classe por vez (ou seja, é multiclasse, não multiproduto), portanto, deve ser usado apenas com classes mutuamente exclusivas, como diferentes tipos de plantas. Não é possível usá-lo para reconhecer várias pessoas em uma imagem.

Agora que você sabe como o modelo estima as probabilidades e faz previsões, vamos dar uma olhada no treinamento. O objetivo é ter um modelo que estime uma alta probabilidade para a classe-alvo (e, consequentemente, uma baixa probabilidade para as outras classes). Minimizar a função de custo mostrada na Equação 4-22, chamada de *entropia cruzada*, deve levar a esse objetivo porque penaliza o modelo quando ele estima uma baixa probabilidade para uma classe-alvo. A entropia cruzada é usada com frequência para medir a

bem um conjunto de probabilidades de classe estimada corresponde às classes-alvo (usaremos essa expressão várias vezes nos próximos capítulos).

Equação 4-22. Função de custo de entropia cruzada

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

- $y_k^{(i)}$ é a probabilidade-alvo de que a i^ésima^ instância pertença à classe *k*. Em geral, é igual a 1 ou 0, dependendo do fato de a instância pertencer ou não à classe.

Observe que, quando há apenas duas classes ($K = 2$), essa função de custo é equivalente à função de custo da regressão logística (perda de log; consulte a Equação 4-17).

Entropia cruzada

A entropia cruzada teve origem na teoria da informação. Suponha que você queira transmitir com eficiência informações sobre o clima todos os dias. Se houver oito opções (ensolarado, chuvoso etc.), você poderá codificar cada opção usando 3 bits, pois $2^3 = 8$. No entanto, se você acha que fará sol quase todos os dias, seria muito mais eficiente codificar "ensolarado" em apenas um bit (0) e as outras sete opções em 4 bits (começando com 1). A entropia cruzada mede o número médio de bits que você realmente envia por opção. Se sua suposição sobre o clima for perfeita, a entropia cruzada será igual à entropia do próprio clima (ou seja, sua imprevisibilidade intrínseca). Mas se suas suposições estiverem erradas (por exemplo, se chover com frequência), a entropia cruzada será maior em um valor chamado *divergência de Kullback-Leibler*.

A entropia cruzada entre duas distribuições de probabilidade p e q é definida como $H(p, q) = -\sum_x p(x) \log q(x)$ (pelo menos quando as distribuições são discretas). Para obter mais detalhes, assista a este vídeo.

O vetor de gradiente dessa função de custo com relação a $\boldsymbol{\theta}^{(k)}$ é dado pela Equação 4-23:

Equação 4-23. Vetor de gradiente de entropia cruzada para a classe k

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

Agora você pode calcular o vetor de gradiente para cada classe e, em seguida, usar o Gradient Descent (ou qualquer outro algoritmo de otimização) para encontrar a matriz de parâmetros $\mathbf{\Theta}$ que minimiza a função de custo.

Vamos usar a Regressão Softmax para classificar as flores da íris em todas as três classes. O `LogisticRegression` do Scikit-Learn usa one-versus-all por padrão quando você o treina em mais de duas classes, mas você pode definir o hiperparâmetro `multi_class` como `"multinomial"` para mudar para Softmax Regression. Você também deve especificar um solucionador que suporte a Softmax Regression, como o solucionador `"lbfgs"` (consulte a documentação do Scikit-Learn para obter mais detalhes). Ele também aplica a regularização $\ell_{(2)}$ por padrão, que você pode controlar usando o hiperparâmetro `C`.

```
X= iris["data"][:, (2, 3)] # comprimento da pétala, largura da pétala
y=  iris["target"]

softmax_reg= LogisticRegression(multi_class= "multinomial",solver= "lbfgs", C= 10)
softmax_reg.fit(X, y)
```

Assim, da próxima vez que encontrar uma íris com pétalas de 5 cm de comprimento e 2 cm de largura, você poderá pedir ao modelo que lhe diga que tipo de íris ela é, e ele responderá Iris-Virginica (classe 2) com 94,2% de probabilidade (ou Iris-Versicolor com 5,8% de probabilidade):

```
>>> softmax_reg.predict([[5, 2]]) array([2])
>>> softmax_reg.predict_proba([[5, 2]]) array([[6.38014896e-
07, 5.74929995e-02, 9.42506362e-01]])
```

A Figura 4-25 mostra os limites de decisão resultantes, representados pelas cores de fundo. Observe que os limites de decisão entre quaisquer duas classes são lineares. A figura também mostra as probabilidades da classe Iris-Versicolor, representadas pelas linhas curvas (por exemplo, a linha rotulada com 0,450 representa o limite de probabilidade de 45%). Observe que o modelo pode prever uma classe que tenha uma probabilidade estimada abaixo de 50%. Por exemplo, no ponto em que todos os limites de decisão se encontram, todas as classes têm a mesma probabilidade estimada de 33%.

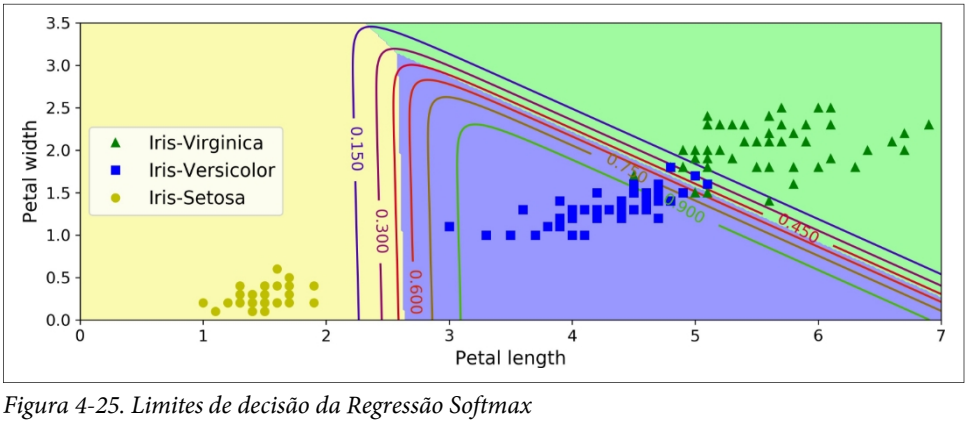

Figura 4-25. Limites de decisão da Regressão Softmax

Exercícios

1. Que algoritmo de treinamento de regressão linear você pode usar se tiver um conjunto de treinamento com milhões de recursos?
2. Suponha que os recursos em seu conjunto de treinamento tenham escalas muito diferentes. Que algoritmos podem sofrer com isso, e como? O que você pode fazer a respeito?
3. O Gradient Descent pode ficar preso em um mínimo local ao treinar um modelo de regressão logística?
4. Todos os algoritmos de Gradient Descent levam ao mesmo modelo, desde que você os deixe rodar por tempo suficiente?
5. Suponha que você use o Batch Gradient Descent e desenhe o erro de validação em cada época. Se você notar que o erro de validação aumenta consistentemente, o que provavelmente está acontecendo? Como você pode corrigir isso?
6. É uma boa ideia interromper o Mini-batch Gradient Descent imediatamente quando o erro de validação aumenta?
7. Qual algoritmo de Gradient Descent (entre os que discutimos) chegará mais rápido às proximidades da solução ideal? Qual deles realmente convergirá? Como você pode fazer os outros convergirem também?
8. Suponha que você esteja usando a regressão polinomial. Você traça as curvas de aprendizado e percebe que há uma grande diferença entre o erro de treinamento e o erro de validação. O que está acontecendo? Quais são as três maneiras de resolver esse problema?
9. Suponha que você esteja usando a Regressão Ridge e perceba que o erro de treinamento e o erro de validação são quase iguais e bastante altos. Você diria que o modelo sofre de alta tendência ou alta variação? Você deve aumentar o hiperparâmetro de regularização α ou reduzi-lo?
10. Por que você gostaria de usar:
    - Regressão Ridge em vez de Regressão Linear simples (ou seja, sem nenhuma regularização)?
    - Lasso em vez de Ridge Regression?
    - Elastic Net em vez de Lasso?
11. Suponha que você queira classificar as imagens como externas/internas e diurnas/noturnas. Você deve implementar dois classificadores de regressão logística ou um classificador de regressão Softmax?
12. Implemente o Batch Gradient Descent com parada antecipada para Softmax Regression (sem usar o Scikit-Learn).

As soluções para esses exercícios estão disponíveis em ???.