Capítulo 4 – Treinando modelos lineares

Até agora, tratamos os modelos de Machine Learning e seus algoritmos de treinamento principalmente como caixas pretas.

No entanto, ter uma boa compreensão de como as coisas funcionam pode ajudá-lo a localizar rapidamente o modelo apropriado, o algoritmo de treinamento correto a ser usado e um bom conjunto de hiperparâmetros para sua tarefa. Entender o que está por trás também ajudará você a depurar problemas e realizar análises de erros com mais eficiência

Começaremos analisando o modelo de Regressão Linear, um dos modelos mais simples que existem. Vamos discutir duas maneiras muito diferentes de treiná-lo:

- Usando uma equação direta de "**forma fechada**" que calcula diretamente os parâmetros do modelo que melhor se ajustam ao modelo ao conjunto de treinamento (ou seja, os parâmetros do modelo que minimizam a função de custo sobre o conjunto de treinamento).
- Usando uma abordagem de otimização iterativa chamada **Gradient Descent (GD)** que ajusta gradualmente os parâmetros do modelo para minimizar a função de custo sobre o conjunto de treinamento, eventualmente convergindo para o mesmo conjunto de parâmetros do primeiro método.

Configuração

Primeiro, vamos importar alguns módulos comuns e garantir que o MatplotLib plote as figuras inline.

```
# Importações comuns
import numpy as np
import os

# para tornar a saída deste notebook estável em todas as execuções
np.random.seed(42)

# Para traçar figuras bonitas
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)

PROJECT_ROOT_DIR = "."
```

1) Regressão Linear

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**)

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

Nessa equação:

- y é o valor previsto
- n é o número de caracteristicas
- xi é o valor da i-ésima caracteristica

- θj é o j-ésimo parametro do modelo

Em um exemplo, y pode ser o valor do indice de qualidade de vida e xi pode ser o PIB per capta do pais e a inflação do país. Assim xi são apenas duas características.

Isso pode ser escrito de forma muito mais concisa usando uma forma vetorizada

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

Nessa equação:

- θ é o vetor de parâmetros do modelo
- x é o vetor de valores de características
- θ . x é o produto escalar (dot product)
- hθ é a função hipotética que representa os dados

OK, esse é o modelo de Regressão Linear - mas como o treinamos?

Treinar um modelo significa **definir seus parâmetros** para 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.

A **medida de desempenho** mais comum de um modelo de regressão é a **raiz do erro quadrático médio (RMSE)**

Portanto, para treinar um modelo de Regressão Linear, precisamos encontrar **o valor de θ que minimiza o RMSE**.

Na prática, é mais simples minimizar o **erro quadrático médio (MSE)**

A função de custo MSE de uma hipótese de Regressão Linear hθ em um conjunto de treinamento X é calculado usando:

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

Atenção: Em Machine Learning, os vetores são frequentemente representados como **vetores de coluna**, que são arrays 2D com uma única coluna. Se θ e x são vetores coluna, então a previsão é y = θ⊺x, onde **θ⊺ é a transposição de θ** (um vetor linha em vez de um vetor coluna) e **θ⊺x é a multiplicação de matrizes de θ⊺ e x**.

A Equação Normal (**Método dos Mínimos Quadrados**)

Para encontrar o vetor θ, que minimiza a função custo, existe uma solução de **forma fechada**

— em outras palavras, uma equação matemática que fornece o resultado diretamente. Isso é chamado de equação normal

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

Lembrando que:

- X é o vetor de valores de características
- y é o vetor de valores do alvo associados a essas caracteristicas

Vamos gerar alguns dados de aparência linear para testar essa equação

```
import numpy as np
np.random.rand(2, 1)
```

Vamos fazer um X que varia de 0 a 2. Podemos

```
X = 2 * np.random.rand(100, 1)
```

```
X[:10]
```

Vamos fazer um y que será uma evolução linear ruidosa. 4 e 3 são os valores dos parametros dessa equação linear, portanto a linha começa com y=4 e evolui com multiplicação por 3

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

```
plt.plot(X, y, "b.")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([0, 2, 0, 15])

plt.show()
```

Atividade

Crie um conjunto de dados de uma evolução linear ruidosa com tendência decrescente.

Vamos testar a Equação Normal.

Precisamos das operações de **transposição**, de **produto escalar**, e de **matriz inversa** (opcional)

O que é a transposição? (opcional)

Matriz Original

```
matriz_original = np.array([[1,2],[3,4]])
matriz_original
```

Matriz Transposta

```
matriz_transposta = matriz_original.T # calcula a transposta de uma matriz
matriz_transposta
```

O que é o produto escalar? (opcional)

O termo multiplicação escalar refere-se ao produto de um número real com uma matriz. Em multiplicações escalares, cada elemento da matriz é multiplicado pelo escalar determinado.

```
a=np.array([2,2,2,2])
b=np.array([2,2,2,2])
```

```
c=a.dot(b.T)
print(c)
```

O que é a **Matriz Inversa**? (opcional)

A **Matriz Inversa** é a matriz que multiplicada pela matriz original gera a **Matriz Identidade.**

```
matriz_inversa = np.linalg.inv(matriz_original) # calcula o inverso de uma matriz
matriz_inversa
```

Matriz Identidade

```
matriz_identidade = np.matmul(matriz_inversa, matriz_original)
matriz_identidade
```

Testando a equação normal para um modelo linear

Para usar a equação normal para um modelo linear é necessário acrescentar um valor 1 para todos os valores de X, que significa que cada X será uma reta com constante 1.

```
X_b = np.c_[np.ones((100, 1)), X]  # adiciona x0 = 1 em cada instância
```

```
X_b.shape
```

```
X_b[:10]
```

```
melhor_theta = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
melhor_theta
```

Esperaríamos que θ0 = 4 e θ1 = 3 em vez dos que apareceram. Perto o suficiente, mas o ruído tornou impossível recuperar os parâmetros exatos da função original.

Vamos fazer previsões para o menor e maior valor de x

```
novos_X = np.array([[0], [2]]) # valores do mínimo e do máximo de x
novos_X_b = np.c_[np.ones((2, 1)), novos_X]  # adicionando x0 = 1 para cada instância
y_previsto = novos_X_b.dot(melhor_theta)
y_previsto # valores do mínimo e máximo de y
```

podemos entao plotar uma reta que inicia no minimo e termina no maximo

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

Executar a regressão linear usando o Scikit-Learn é mais simples

```
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(X, y)
lin_reg.intercept_, lin_reg.coef_
```

```
lin_reg.predict(novos_X)
```

A abordagem da Equação Normal e outra abordagem chamada SVD ficam muito lentas quando o número de características aumenta (por exemplo, 100.000).

Atividade

Teste a equação normal no seu conjunto com tendência decrescente

Abordagem Singular Value Decomposition - SVD (Opcional)

np.linalg.lstsq

A classe `LinearRegression` é baseada na função `scipy.linalg.lstsq()` (o nome significa "least squares" que é mínimos quadrados), e podemos chamá-la diretamente usando a matriz com a coluna extra X_b

```
theta_best_svd, residuals, rank, s = np.linalg.lstsq(X_b, y, rcond=1e-6)
theta_best_svd
```

np.linalg.pinv

Essa função calcula $\mathbf{X}^{+}\mathbf{y}$, onde $\mathbf{X}^{+}$ é a *pseudoinversa* de $\mathbf{X}$ (especificamente inversa Moore-Penrose). Essa equação é mais precisa e versátil que a equação normal. Voce pode usar `np.linalg.pinv()` para calcular a pseudoinversa diretamente:

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

A **Equação Normal** calcula o inverso de X^T X, que é uma matriz (n + 1) × (n + 1) (onde n é o número de características). A complexidade computacional de inverter tal matriz é tipicamente cerca de **O(n^3)**, dependendo da implementação.

A abordagem SVD usada pela classe LinearRegression do Scikit-Learn é sobre **O(n2)**.

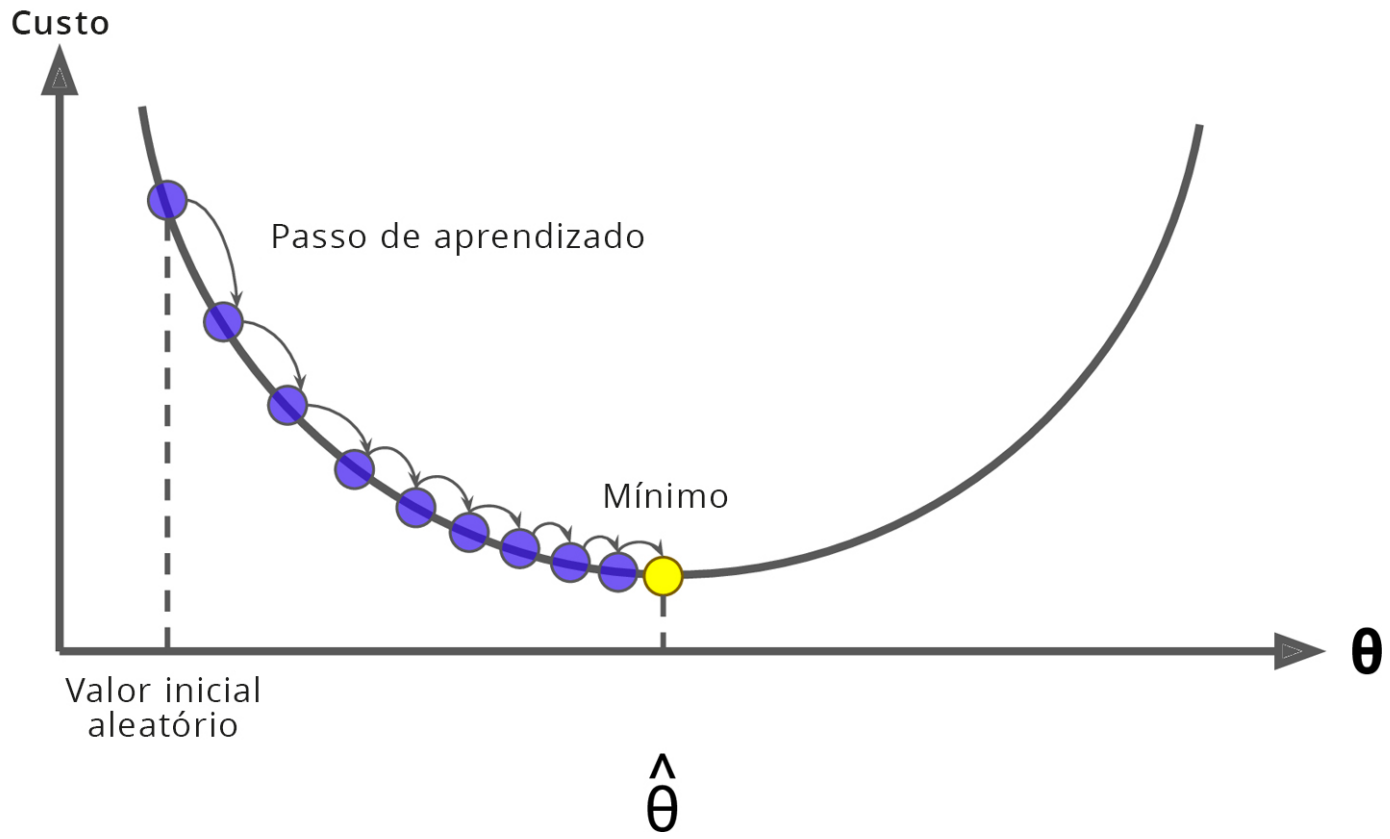
2) Gradiente Descendente (Gradiente Descent)

Gradient Descent é um algoritmo de otimização genérico capaz de encontrar soluções ótimas 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 um nevoeiro denso, e você só pode sentir a inclinação do solo abaixo de seus pés. Uma boa estratégia para chegar rapidamente ao fundo do vale é descer em direção à encosta mais íngreme.

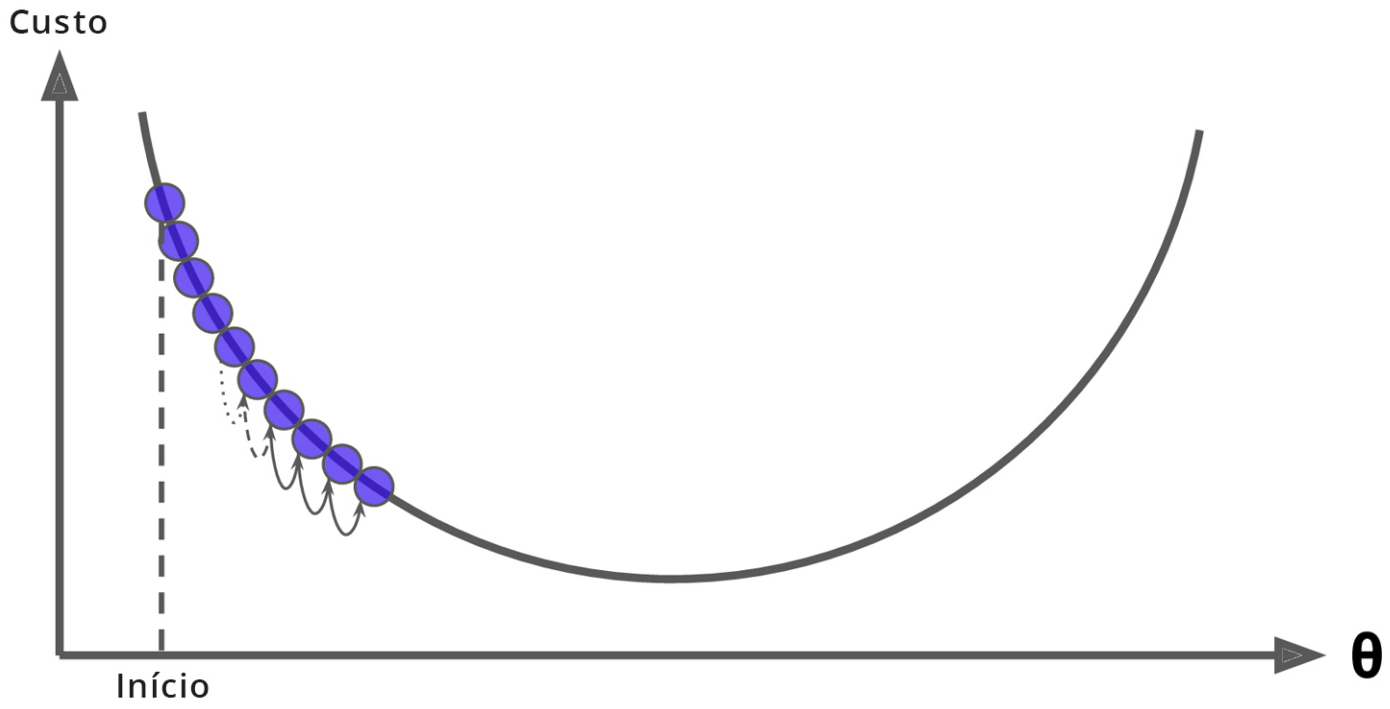
Concretamente, você começa preenchendo θ com valores aleatórios (isso é chamado de **inicialização aleatória**).

Então você o melhora gradualmente, dando um pequeno passo de cada vez, cada passo tentando diminuir a função de custo (por exemplo, o **MSE**), até que o algoritmo convirja para um mínimo

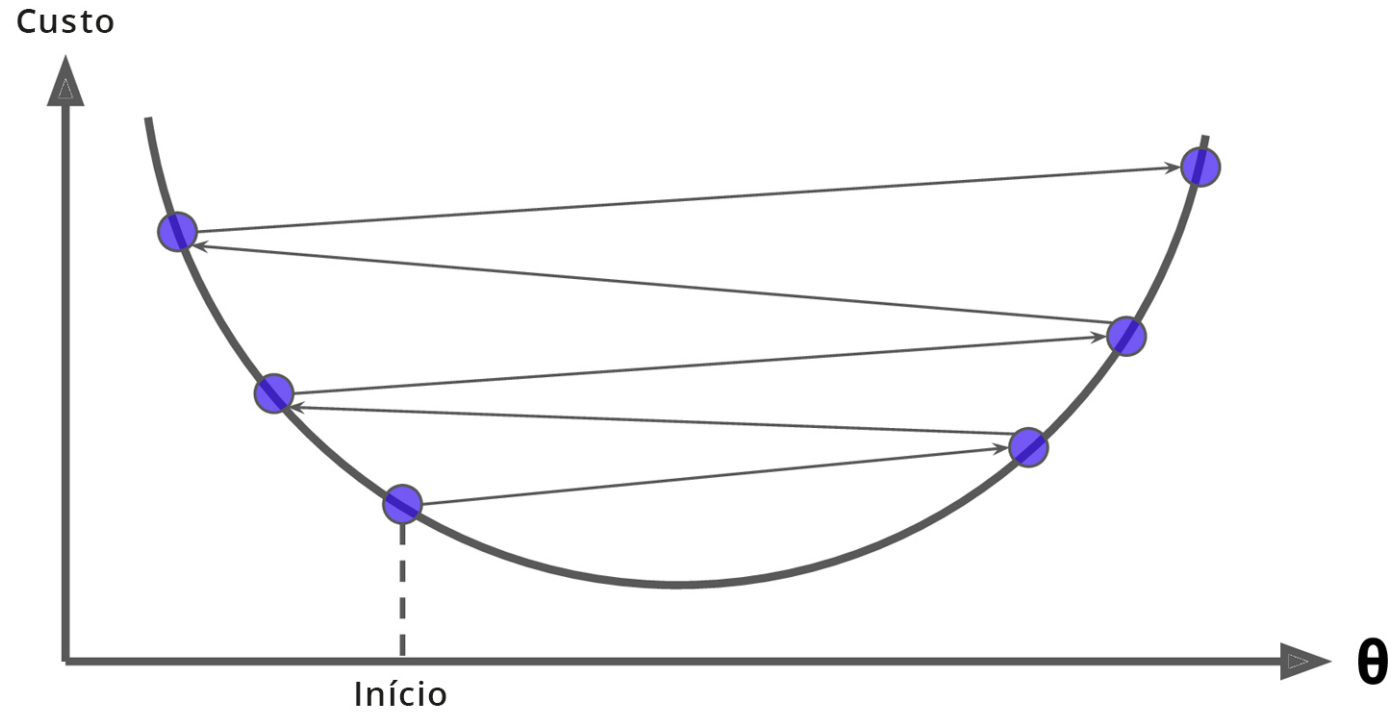


A importância da taxa de aprendizado

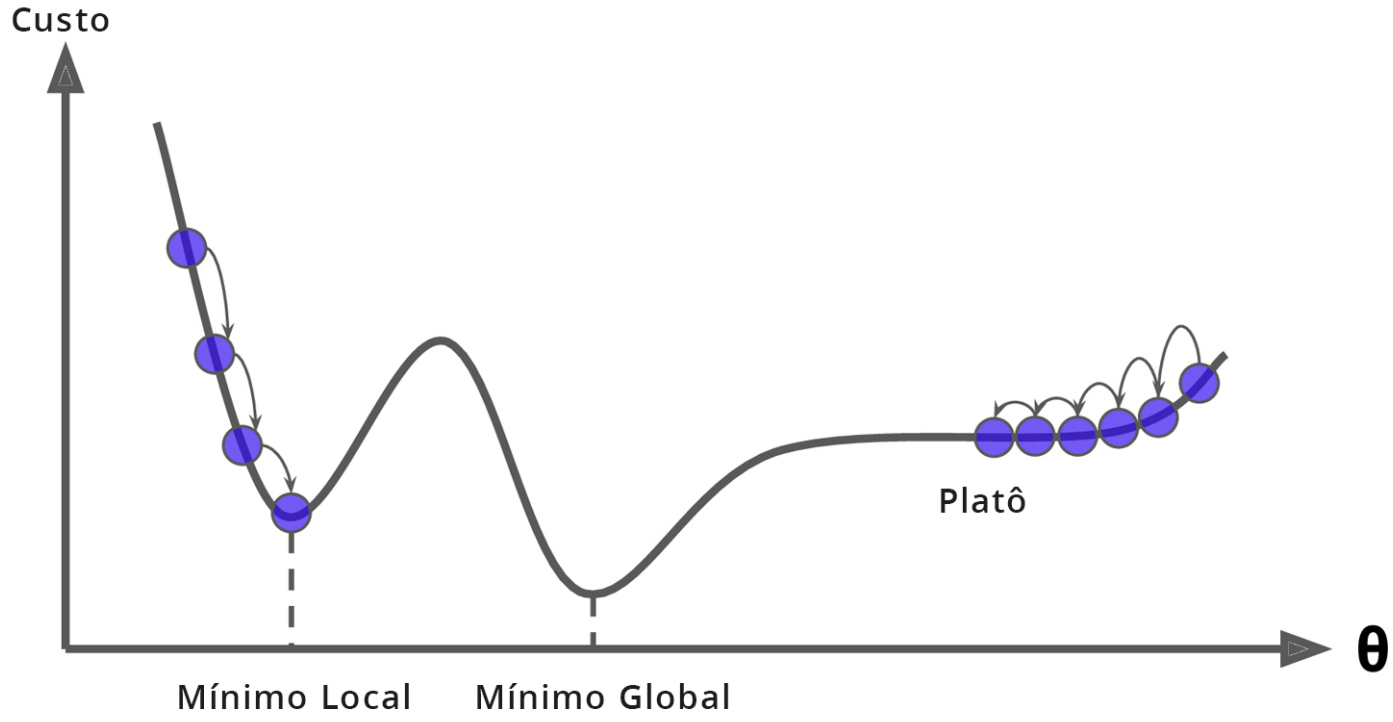
Caso a taxa de aprendizado seja muito pequena



Caso a taxa de aprendizado seja muito grande



Finalmente, nem todas as funções de custo parecem tigelas bonitas e regulares.



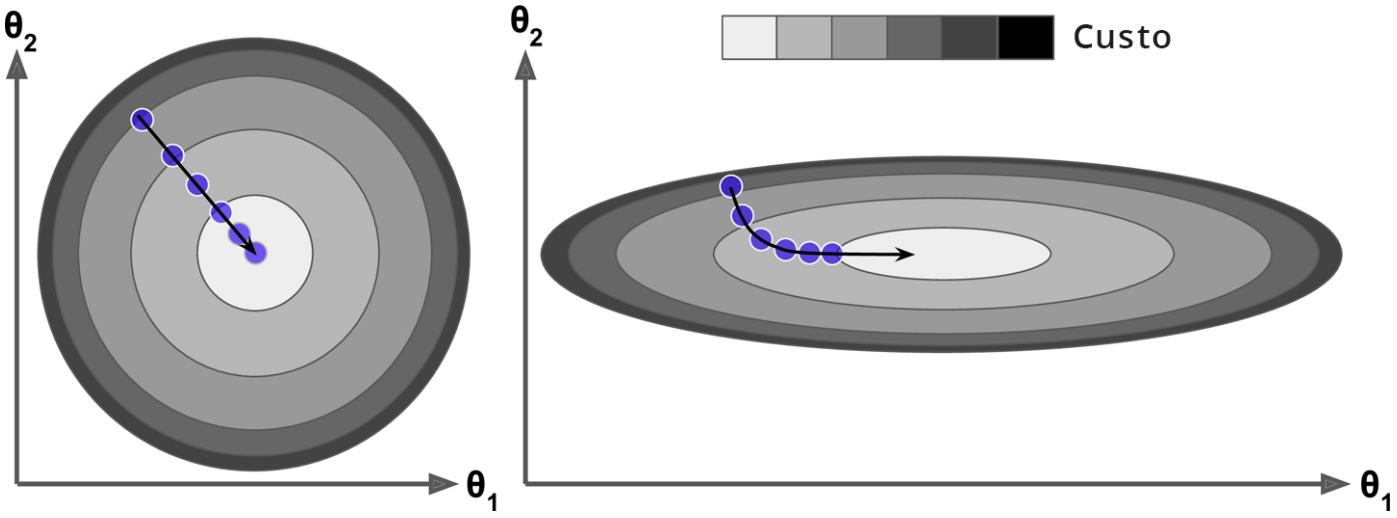
Felizmente, a **função de custo MSE** para um modelo de Regressão Linear é uma **função convexa**, o que significa que, se você escolher dois pontos quaisquer na curva, o segmento de linha que os une nunca cruzará a curva.

Isso implica que não há **mínimos locais**, apenas um **mínimo global**. É também uma função contínua com uma inclinação que nunca muda abruptamente.

Esses dois fatos têm uma grande consequência:

- a descida do gradiente é garantida para se aproximar arbitrariamente do mínimo global (se você esperar o suficiente e se

Na verdade, a função de custo tem a forma de uma tigela (esquerda), mas pode ser uma tigela alongada se os recursos tiverem **escalas muito diferentes**(direita).



Conclusão: Ao usar Gradient Descent, você deve garantir que todos os atributos tenham uma escala semelhante ou então levará muito mais tempo para convergir para o ótimo global.

Gradiente Descendente em Lote (Batch Gradient Descent)

Para implementar o **Gradiente Descendente em Lote (batch)**, você precisa calcular o gradiente da função de custo em relação a cada parâmetro do modelo θj.

Em outras palavras, você precisa calcular quanto a função de custo mudará se você alterar θj um pouco. Isso é chamado de **derivada parcial**.

É como perguntar “Qual é a inclinação da montanha sob meus pés se eu estiver de frente para o leste?” e, em seguida, fazendo a mesma pergunta voltada para o norte

Derivadas parciais da função de custo:

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

Depois de ter o vetor gradiente, que aponta para cima, basta ir na direção oposta para descer.

Passo do gradiente descendente:

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

```
eta = 0.1  # taxa de aprendizado
n_iterations = 1000
```

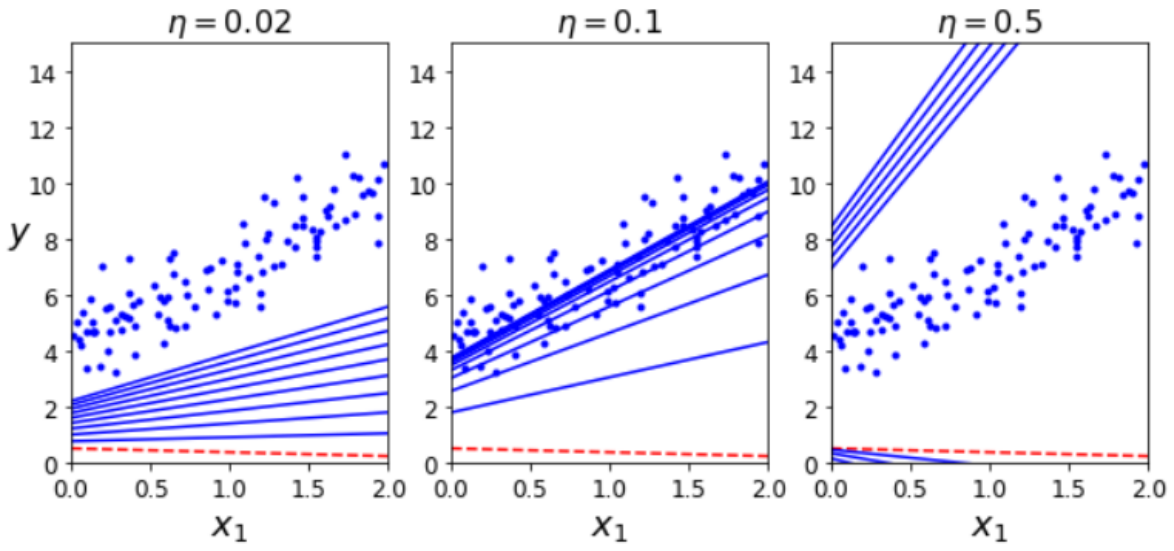
```
m = 100 #quantidade de amostras

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

for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y) # cálculo do erro e da direcao dele
    theta = theta - eta * gradients # passo do grandiente descendente
    print(theta[0],' ',theta[1])
```

```
theta
```

Ei, isso é exatamente o que a Equação Normal encontrou! Gradient Descent funcionou perfeitamente.

Mas e se você tivesse usado um eta de taxa de aprendizado diferente?

A Figura abaixo mostra os primeiros 10 passos do Gradient Descent usando três taxas de aprendizado diferentes (a linha tracejada representa o ponto de partida).



À esquerda, **a taxa de aprendizado é muito baixa**: o algoritmo eventualmente alcançará a solução, mas levará muito tempo.

No meio, **a taxa de aprendizado parece muito boa**: em apenas algumas iterações, já convergiu para a solução.

À direita, **a taxa de aprendizado é muito alta**: o algoritmo diverge, pulando para todo lado e chegando cada vez mais longe da solução a cada passo.

Para encontrar uma boa taxa de aprendizado, você pode usar a **pesquisa de grade**

Você pode se perguntar como definir o número de iterações. Se for muito baixo, você ainda estará longe da solução ótima quando o algoritmo parar; mas se for muito alto, você perderá tempo enquanto os parâmetros do modelo não mudam mais. Uma solução simples é definir um número muito grande de iterações, mas interromper o algoritmo quando o vetor gradiente se torna pequeno, isto é, quando sua norma se torna menor que um pequeno número ϵ (chamado de **tolerância**) - porque isso acontece quando Gradient Descente (quase) atingiu o mínimo.

Atividade

Experimente alguns valores de taxa de aprendizado e iterações para tentar encontrar um conjunto que alcance o valor de theta da equação normal, com um erro menor que 0,001 e com o menor número de iterações.

Código para plotar as figuras (opcional)

```
theta_path_bgd = []   #caminho do theta do batch gradient descent

def plot_gradient_descent(theta, eta, theta_path=None):
    m = len(X_b)
    plt.plot(X, y, "b.")
    n_iterations = 1000
    for iteration in range(n_iterations):
        if iteration < 10:
            y_predict = novos_X_b.dot(theta)
            style = "b-" if iteration > 0 else "r--"
            plt.plot(novos_X, y_predict, style)
        gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
        theta = theta - eta * gradients
        if theta_path is not None:
            theta_path.append(theta)
    plt.xlabel("$x_1$", fontsize=18)
    plt.axis([0, 2, 0, 15])
    plt.title(r"$\eta = {}$".format(eta), fontsize=16)
```

```
np.random.seed(42)
theta = np.random.randn(2,1)  # random initialization

plt.figure(figsize=(10,4))
plt.subplot(131); plot_gradient_descent(theta, eta=0.02)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.subplot(132); plot_gradient_descent(theta, eta=0.1, theta_path=theta_path_bgd)
plt.subplot(133); plot_gradient_descent(theta, eta=0.5)


plt.show()
```

Gradiente Descendente Estocástico (Stochastic Gradient Descent)

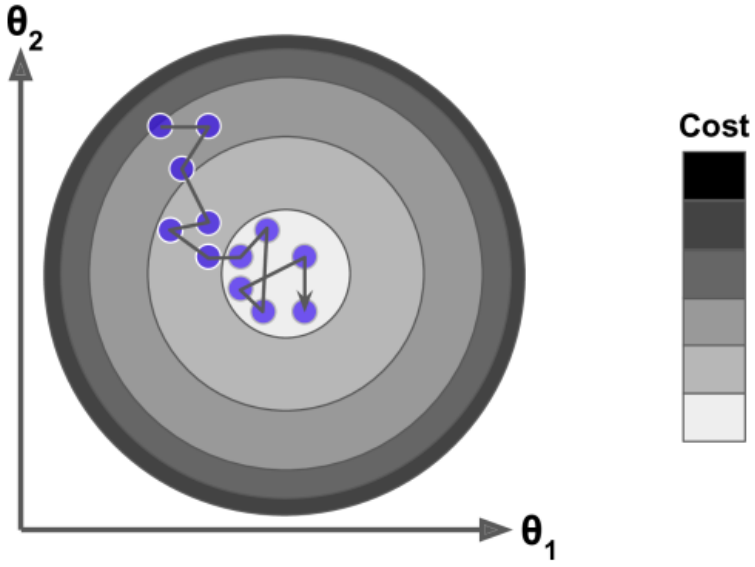
O principal problema com o **Gradiente Descendente em Lote (batch)** é o fato de que ele usa todo o conjunto de treinamento para calcular os gradientes em cada etapa, o que o torna muito lento quando o conjunto de treinamento é grande.

No extremo oposto, **Gradiente Descendente Estocástico** escolhe uma instância aleatória no conjunto de treinamento em cada etapa e calcula os gradientes com base apenas nessa única instância.

Obviamente, trabalhar em uma única instância de cada vez torna o algoritmo muito mais rápido porque tem muito poucos dados para manipular em cada iteração. Também possibilita treinar em grandes conjuntos de treinamento, pois apenas uma instância precisa estar na memória a cada iteração

Por outro lado, devido à sua natureza estocástica (ou seja, aleatória), este algoritmo é muito menos regular do que Batch Gradient Descent

Com o tempo, ele terminará muito próximo do mínimo, mas, quando chegar lá, continuará oscilando, nunca se acalmando (veja a Figura 4-9). Assim, uma vez que o algoritmo para, os valores finais dos parâmetros são bons, mas não ótimos.



Quando a função de custo é muito irregular (como na Figura abaixo), isso pode realmente ajudar o algoritmo a sair dos mínimos locais, de modo que a Descida do Gradiente Estocástico tem uma chance melhor de encontrar o mínimo global do que a Descida do Gradiente em Lote.

Portanto, a aleatoriedade é boa para escapar de ótimos locais, mas ruim porque significa que o algoritmo nunca pode se estabelecer no mínimo.

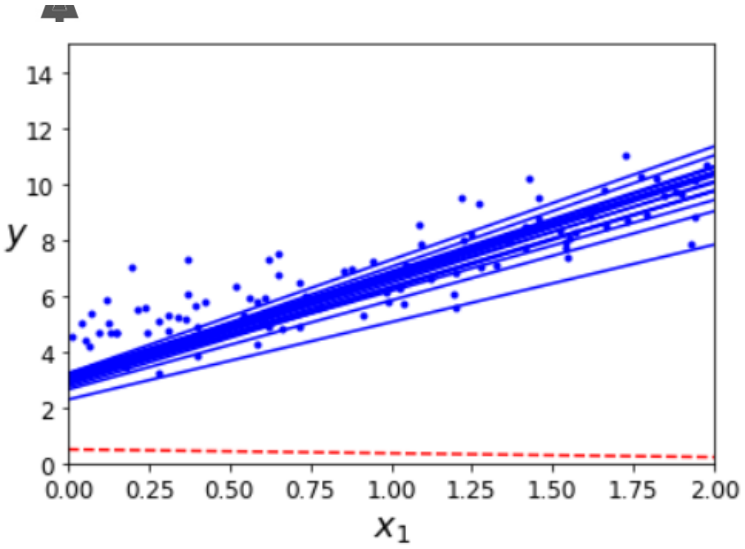
Uma solução para esse dilema é **reduzir gradualmente a taxa de aprendizado**.

As etapas começam grandes (o que ajuda a progredir rapidamente e escapar de mínimos locais), depois ficam cada vez menores, permitindo que o algoritmo se estabeleça no mínimo global.

Este processo é semelhante ao recozimento simulado, um algoritmo inspirado no processo em metalurgia de recozimento, onde o metal fundido é resfriado lentamente.

A função que determina a taxa de aprendizado em cada iteração é chamada de **cronograma de aprendizado**.

Resultado de uma implementação do SGD. O algoritmo gera menos linhas distantes da linha ideal.



Observe que, como as instâncias são escolhidas aleatoriamente, algumas instâncias podem ser escolhidas várias vezes por época, enquanto outras podem não ser escolhidas.

Se você quiser ter certeza de que o algoritmo passa por todas as instâncias em cada época, outra abordagem é embaralhar o conjunto de treinamento (certificando-se de embaralhar os atributos de entrada e os rótulos em conjunto), passar instância por instância e embaralhá-lo novamente, e assim por diante. No entanto, essa abordagem geralmente converge mais lentamente.

Código para demonstrar o SGD (Opcional)

```
theta_path_sgd = []
m = len(X_b)
np.random.seed(42)
```

```
n_epochs = 50
t0, t1 = 5, 50  # hiperparametros do cronograma de aprendizado

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

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

for epoch in range(n_epochs):
    for i in range(m):
        if epoch == 0 and i < 20:                    # not shown in the book
            y_predict = novos_X_b.dot(theta)         # not shown
            style = "b-" if i > 0 else "r--"         # not shown
            plt.plot(novos_X, y_predict, style)      # not shown
        random_index = np.random.randint(m)
        xi = X_b[random_index:random_index+1]
        yi = y[random_index:random_index+1]
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
        eta = learning_schedule(epoch * m + i)
        theta = theta - eta * gradients
        theta_path_sgd.append(theta)                 # not shown

plt.plot(X, y, "b.")                                 # not shown
plt.xlabel("$x_1$", fontsize=18)                     # not shown
plt.ylabel("$y$", rotation=0, fontsize=18)           # not shown
plt.axis([0, 2, 0, 15])                              # not shown

plt.show()                                           # not shown
```

```
1 theta
```

SGD no no sklearn

```
1 from sklearn.linear_model import SGDRegressor
2 sgd_reg = SGDRegressor(max_iter=1000, tol=1e-3, penalty=None, eta0=0.1, random_state=42) # penalty é um regularizad
3 sgd_reg.fit(X, y.ravel()) # SGD Instância
```

```
1 sgd_reg.intercept_, sgd_reg.coef_
```

```
1 sgd_reg.n_iter_
```

Gradiente Descendente em Minilote (Mini-batch gradient descent)

O último algoritmo Gradient Descent que veremos é chamado de **Mini-batch Gradient Descent**.

É simples de entender uma vez que você conhece Batch e Stochastic Gradient Descent: em cada etapa,

- em vez de calcular os gradientes com base no conjunto de treinamento completo (como em Batch GD) ou com base em apenas uma instância (como em Stochastic GD),
- Mini- batch GD calcula os gradientes em pequenos conjuntos aleatórios de instâncias chamados mini-lotes.

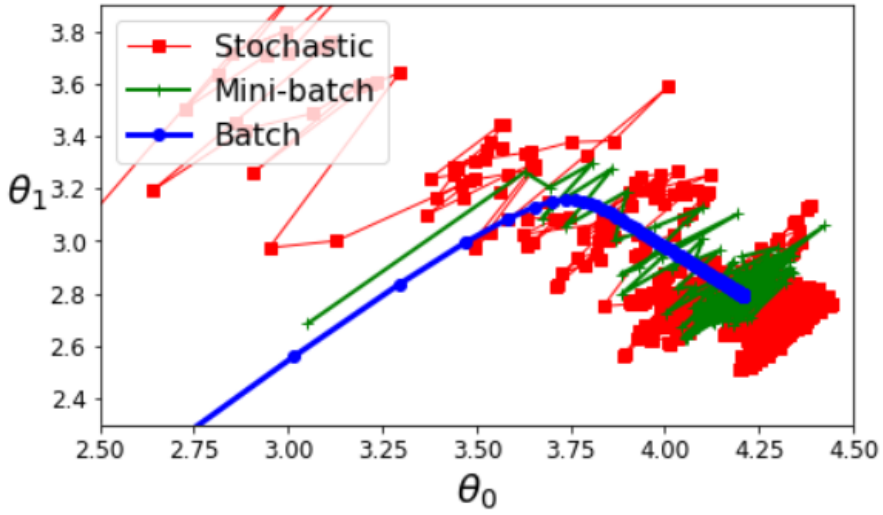
A principal vantagem do Mini-batch GD sobre o Stochastic GD é que você pode obter um aumento de desempenho da otimização de hardware das operações de matriz, especialmente ao usar GPUs.

O progresso do algoritmo no espaço de parâmetros **é menos errático** do que com Stochastic GD, especialmente com mini-lotes razoavelmente grandes.

A Figura a seguir mostra os caminhos percorridos pelos três algoritmos Gradient Descent no espaço de parâmetros durante o treinamento.

Todos eles terminam perto do mínimo, mas o caminho do Batch GD na verdade para no mínimo, enquanto o Stochastic GD e o Mini-batch GD continuam andando.

No entanto, não se esqueça que GD em lote leva muito tempo para dar cada etapa, e GD estocástico e GD de mini-lote também atingiriam o mínimo se você usasse um bom cronograma de aprendizado.



Implementação do Mini batch GD (Opcional)

Apenas executar o código abaixo para gerar figuras a seguir. Não precisa discutir o código.

```
theta_path_mgd = []

n_iterations = 50
minibatch_size = 20

np.random.seed(42)
theta = np.random.randn(2,1)  # random initialization

t0, t1 = 200, 1000
def learning_schedule(t):
    return t0 / (t + t1)

t = 0
for epoch in range(n_iterations):
    shuffled_indices = np.random.permutation(m)
    X_b_shuffled = X_b[shuffled_indices]
    y_shuffled = y[shuffled_indices]
    for i in range(0, m, minibatch_size):
        t += 1
        xi = X_b_shuffled[i:i+minibatch_size]
        yi = y_shuffled[i:i+minibatch_size]
        gradients = 2/minibatch_size * xi.T.dot(xi.dot(theta) - yi)
        eta = learning_schedule(t)
        theta = theta - eta * gradients
        theta_path_mgd.append(theta)
```

```
theta
```

(Opcional) Código para gerar o caminho de treino dos 3 algoritmos

```
theta_path_bgd = np.array(theta_path_bgd)
theta_path_sgd = np.array(theta_path_sgd)
theta_path_mgd = np.array(theta_path_mgd)
```

```
plt.figure(figsize=(7,4))
plt.plot(theta_path_sgd[:, 0], theta_path_sgd[:, 1], "r-s", linewidth=1, label="Stochastic")
plt.plot(theta_path_mgd[:, 0], theta_path_mgd[:, 1], "g-+", linewidth=2, label="Mini-batch")
plt.plot(theta_path_bgd[:, 0], theta_path_bgd[:, 1], "b-o", linewidth=3, label="Batch")
plt.legend(loc="upper left", fontsize=16)
plt.xlabel(r"$\theta_0$", fontsize=20)
plt.ylabel(r"$\theta_1$   ", fontsize=20, rotation=0)
plt.axis([2.5, 4.5, 2.3, 3.9])

plt.show()
```

Comparação de algoritmos para regressão linear

Table 4-1. Comparison of algorithms for Linear Regression

Algorithm	Large *m*	Out-of-core support	Large *n*	Hyperparams	Scaling required	Scikit-Learn
Normal Equation	Fast	No	Slow	0	No	N/A
SVD	Fast	No	Slow	0	No	`LinearRegression`
Batch GD	Slow	No	Fast	2	Yes	`SGDRegressor`
Stochastic GD	Fast	Yes	Fast	≥2	Yes	`SGDRegressor`
Mini-batch GD	Fast	Yes	Fast	≥2	Yes	`SGDRegressor`

m é a quantidade de amostras e **n** é a quantidade de caracteristicas

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

Obs: **out-of-core** é a capacidade de treinar sistemas em grandes conjuntos de dados que não cabem na memória principal de uma máquina

Atividade

Qual algoritmo de treinamento de regressão linear você pode usar se tiver um conjunto de treinamento com milhões de características? Justifique sua resposta.

Resposta

Se você tem um conjunto de treinamento com milhões de recursos, você pode usar **Stochastic Gradient Descent** ou **Mini-batch Gradient Descent**, e talvez **Batch Gradient Descent** se o conjunto de treinamento couber na memória. Se você tiver um treinamento **online** que precisa ser atualizado com muita frequência, melhor usar **SGD** ao custo de sacrificar a acurácia. Mas você não pode usar a **Equação Normal** (Método dos Mínimos Quadrados) ou a abordagem **SVD** porque a complexidade computacional cresce rapidamente (mais do que quadrática) com o número de recursos.

3) Regressão Polinomial

E se seus dados forem mais complexos do que uma linha reta? Surpreendentemente, você pode usar um modelo linear para ajustar dados não lineares.

```
1 import numpy as np
2 import numpy.random as rnd
3 np.random.seed(42)
```

gerar dados não lineares baseados numa equação quadrática

$y = 0,5x_1^2 + 1,0x_1 + 2,0$

```
1 m = 100
2 X = 6 * np.random.rand(m, 1) - 3
3 y = 0.5 * X**2 + 1 * X + 2 + np.random.randn(m, 1)
```

```
plt.plot(X, y, "b.")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([-3, 3, 0, 10])

plt.show()
```

Claramente, uma linha reta nunca se ajustará a esses dados corretamente.

Então, vamos usar a classe **PolynomialFeatures** do Scikit-Learn para transformar nossos dados de treinamento, adicionando o quadrado (polinômio de segundo grau) de cada característica no conjunto de treinamento como uma nova característica. É parecido com o que fizemos para o cálculo da regressão linear.

```
from sklearn.preprocessing import PolynomialFeatures
poly_features = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly_features.fit_transform(X)
```

```
X[0]
```

```
X[0]**2
```

```
X_poly[0]
```

X_poly agora contém a característica original de X mais o quadrado dessa característica.

```
X_poly.shape
```

```
lin_reg = LinearRegression()
lin_reg.fit(X_poly, y)
lin_reg.intercept_,·lin_reg.coef_
```

O modelo estima:

$$y = 0,56x_1^2 + 0,93x_1 + 1,78$$

Quando na verdade a função original era:

$$y = 0,5x_1^2 + 1,0x_1 + 2,0$$

Intercept é a constante, que é o valor inicial de y, os coeficientes são equivalentes ao angulo das duas retas estimadas (valor de x e de x^2). Duas retas de angulos diferentes fazem uma curva.

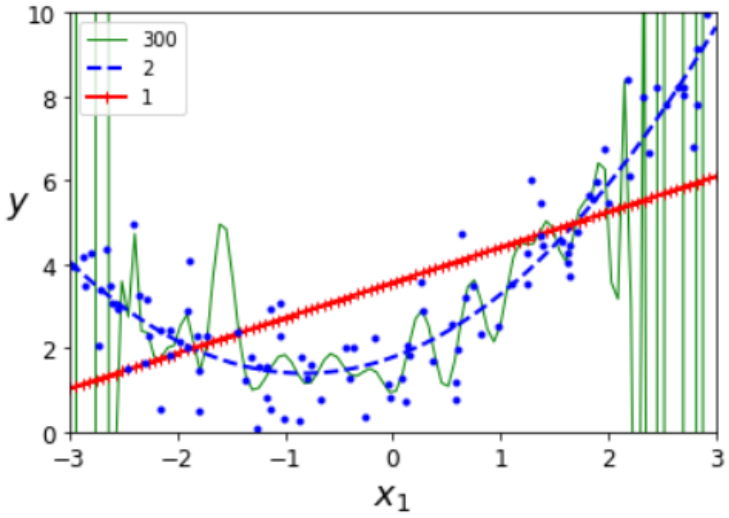
A constante estimada é muito proxima da constante 2 original. O mesmo vale para os outros dois.

```
X_new = np.linspace(-3, 3, 100).reshape(100, 1) # geração de um espaço linear de 100 valores entre -3 e 3
X_new_poly = poly_features.transform(X_new)
y_new = lin_reg.predict(X_new_poly) # predicao para os valores do espaço linear

plt.plot(X, y, "b.")
plt.plot(X_new, y_new, "r-", linewidth=2, label="Predictions")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.legend(loc="upper left", fontsize=14)
plt.axis([-3, 3, 0, 10])

plt.show()
```

4) Curvas de Aprendizado

Se você executar uma regressão polinomial de alto grau, provavelmente ajustará os dados de treinamento muito melhor do que com a regressão linear simples.

Por exemplo, a Figura mostra o resultado de 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 segundo grau).



(Opcional) Código para treinar e prever regressões com diferentes graus polinomiais

```
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

for style, width, degree in (("g-", 1, 300), ("b--", 2, 2), ("r-+", 2, 1)):
    polybig_features = PolynomialFeatures(degree=degree, include_bias=False)
    std_scaler = StandardScaler()
    lin_reg = LinearRegression()
    polynomial_regression = Pipeline([
            ("poly_features", polybig_features),
            ("std_scaler", std_scaler),
            ("lin_reg", lin_reg),
        ])
    polynomial_regression.fit(X, y)
    y_newbig = polynomial_regression.predict(X_new)
    plt.plot(X_new, y_newbig, style, label=str(degree), linewidth=width)

plt.plot(X, y, "b.", linewidth=3)
plt.legend(loc="upper left")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([-3, 3, 0, 10])

plt.show()
```

Explicação

Este **modelo de regressão polinomial de alto grau** está **superajustando** severamente os dados de treinamento, enquanto o **modelo linear** está **subajustando-os**. O modelo que irá generalizar melhor neste caso é o **modelo quadrático**, o que faz sentido porque os dados foram gerados usando um modelo quadrático.

Mas em geral você não saberá qual função gerou os dados, então como você pode decidir o quão complexo seu modelo deve ser?

Como você pode dizer que seu modelo está superajustando ou subajustando os dados?

Anteriormente usamos **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, **seu modelo está superajustado.**
- Se um modelo tiver um desempenho ruim em ambos, então,**seu modelo está subajustado**.
- Essa é uma maneira de saber quando um modelo é muito simples ou muito complexo.

Outra maneira de saber é observar as **curvas de aprendizado**:

- são gráficos do desempenho do modelo no conjunto de treinamento e o conjunto de validação em função do tamanho do conjunto de treinamento (ou iteração de treinamento).
- Para gerar os gráficos, treine o modelo várias vezes em subconjuntos de tamanhos diferentes do conjunto de treinamento.

O código a seguir define uma função que, dado alguns dados de treinamento, traça as curvas de aprendizado de um modelo treinado a cada ciclo com mais dados do conjunto de treino.

O código abaixo é necessário para exemplos posteriores. (Discutir o código é opcional.)

```
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, random_state=10)
    train_errors, val_errors = [], []

    for m in range(1, len(X_train) + 1): # a cada repetição teremos um tamanho maior para o conjunto de treino
        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="trainamento")
    plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="validação")
    plt.legend(loc="upper right", fontsize=14)   # not shown in the book
    plt.xlabel("Tamanho do Conjunto de Treinamento", fontsize=14) # not shown
    plt.ylabel("RMSE", fontsize=14)              # not shown
```

```
lin_reg = LinearRegression()
plot_learning_curves(lin_reg, X, y)
plt.axis([0, 80, 0, 3])                         # not shown in the book
plt.show()                                      # not shown
```

Este modelo que está **underfitting** merece um pouco de explicação.

Primeiro, vamos olhar para o **desempenho do modelo nos dados de treinamento**:

- quando há apenas uma ou duas instâncias no conjunto de treinamento, o modelo pode ajustá-las perfeitamente, e é por isso que a curva começa em zero.
- Mas à medida que novas instâncias são adicionadas ao conjunto de treinamento, torna-se impossível para o modelo ajustar os dados de treinamento perfeitamente, tanto porque os dados são ruidosos quanto porque não são lineares.
- **Portanto, o erro nos dados de treinamento aumenta até atingir um platô, ponto em que adicionar novas instâncias ao conjunto de treinamento não torna o erro médio muito melhor ou pior.**

Segundo, vamos ver o desempenho do **modelo nos dados de validação**.

Quando o modelo é treinado em poucas instâncias de treinamento, ele é incapaz de generalizar adequadamente, razão pela qual o erro de validação é inicialmente muito grande.

Essas curvas de aprendizado são típicas de um modelo que está subajustado: Ambas as curvas atingiram um platô; eles são próximos e bastante altos.

Correção do Problema:

Se o seu modelo estiver subajustando aos dados de treinamento, adicionar mais exemplos de treinamento não ajudará. **Você precisá usar um modelo mais complexo ou obter melhores caractrísticas.**

Agora vamos ver as curvas de aprendizado de um modelo polinomial de 10º grau nos mesmos dados (Figura abaixo):

```
from 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)
plt.axis([0, 80, 0, 3])           # not shown
plt.show()                        # not shown
```

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 com o modelo de Regressão Linear puro.**
- **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, que é a marca registrada de um **modelo de overfitting**. Se você usasse um conjunto de treinamento muito maior, no entanto, as duas curvas continuariam a se aproximar.

Correção do Problema:

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

Atividade

Teste diferentes valores de graus polinomiais para encontrar um que apresente um erro de validação menor do que com 10 graus. Use o código do pipeline acima.

Resposta

```
from sklearn.pipeline import Pipeline

polynomial_regression = Pipeline([
        ("poly_features", PolynomialFeatures(degree=4, include_bias=False)),
        ("lin_reg", LinearRegression()),
    ])

plot_learning_curves(polynomial_regression, X, y)
plt.axis([0, 80, 0, 3])           # not shown

plt.show()
```

5) Regularização de Modelos Lineares

Uma boa maneira de **reduzir o overfitting é regularizar o modelo** (ou seja, restringi-lo):

- quanto menos graus de liberdade ele tiver, mais difícil será para ele superajustar os dados.
- **Para um modelo linear**, a regularização é alcançada restringindo os pesos do modelo.
- **Para um modelo polinomial**, a regularização é alcançada reduzindo o número de graus polinomiais.

Ridge Regression (Regressão do cume)

Ridge Regression (também chamada de regularização de Tikhonov) **é uma versão regularizada da Regressão Linear**:

- um termo de regularização é adicionado à função de custo para aumentar o custo (erro) quando os pesos aumentam.
- Isso diminui a variação dos pesos durante o treinamento.

Observação:

O termo de regularização só deve ser adicionado à função de custo durante o treinamento. Depois que o modelo for treinado, você deseja usar a medida de desempenho não regularizada para avaliar o desempenho do modelo.

Equation 4-8. Ridge Regression cost function

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

Observe que o termo de vies θ0 não é regularizado (a soma começa em i = 1, não 0)

O hiperparâmetro α (alpha) controla o quanto você deseja regularizar o modelo.

- **Se α = 0**, então Ridge Regression é apenas Regressão Linear.
- **Se α for muito grande**, então todos os pesos terminam muito próximos de zero e o resultado é uma linha reta passando pela média dos dados.

obs: com w sendo o vetor de pesos, estamos aplicando uma regularização de 1/2 (||w||2)^2 , onde ||w||2 é a norma l2 do vetor de pesos

Observação:

É importante **dimensionar os dados** (por exemplo, usando um **StandardScaler**) antes de executar a regressão de cume, pois é sensível à escala dos recursos de entrada. Isso é verdade para a maioria dos modelos regularizados.

A Figura abaixo mostra vários modelos Ridge treinados em alguns dados lineares usando diferentes valores de α.

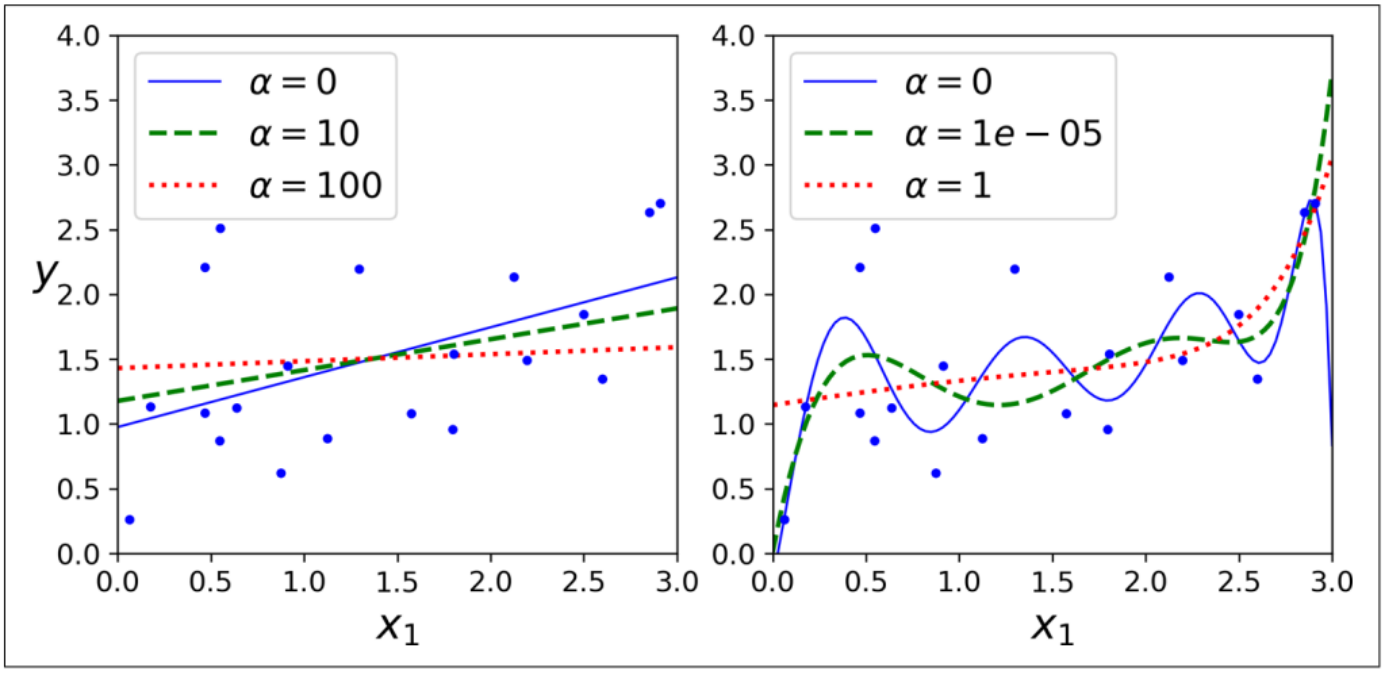


Figure 4-17. A linear model (left) and a polynomial model (right), both with various levels of Ridge regularization

(Opcional) Código para gerar a figura com exemplos de diferentes niveis de regularização de ridge (l2)

```
np.random.seed(42)
m = 20
X = 3 * np.random.rand(m, 1)
y = 1 + 0.5 * X + np.random.randn(m, 1) / 1.5
X_new = np.linspace(0, 3, 100).reshape(100, 1)
```

```
plt.plot(X, y, "b.")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
#plt.axis([-3, 3, 0, 10])

plt.show()
```

```
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import Ridge

def plot_model(model_class, polynomial, alphas, **model_kargs):
    for alpha, style in zip(alphas, ("b-", "g--", "r:")):
        model = model_class(alpha, **model_kargs) if alpha > 0 else LinearRegression()
        if polynomial:
            model = Pipeline([
                    ("poly_features", PolynomialFeatures(degree=10, include_bias=False)),
                    ("std_scaler", StandardScaler()),
                    ("regul_reg", model),
                ])
        model.fit(X, y)
        y_new_regul = model.predict(X_new)
        lw = 2 if alpha > 0 else 1
        plt.plot(X_new, y_new_regul, style, linewidth=lw, label=r"$\alpha = {}$".format(alpha))
    plt.plot(X, y, "b.", linewidth=3)
    plt.legend(loc="upper left", fontsize=15)
    plt.xlabel("$x_1$", fontsize=18)
    plt.axis([0, 3, 0, 4])

plt.figure(figsize=(8,4))
```

```
plt.subplot(121)
plot_model(Ridge, polynomial=False, alphas=(0, 10, 100), random_state=42)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.subplot(122)
plot_model(Ridge, polynomial=True, alphas=(0, 10**-5, 1), random_state=42)

plt.show()
```

Assim como na Regressão Linear, podemos realizar a Regressão de Ridge calculando uma equação de forma fechada ou executando Gradient Descent. As vantagens e desvantagens são as mesmas.

Regressão Ridge com o Scikit-Learn usando uma solução de formato fechado

```
from sklearn.linear_model import Ridge
ridge_reg = Ridge(alpha=1, solver="cholesky", random_state=42)
ridge_reg.fit(X, y)
ridge_reg.predict([[1.5]])
```

```
ridge_reg.score(X, y)
```

```
y.shape
```

```
X.shape
```

Regressão Ridge usando SGD

Observação: para ser à prova do futuro, definimos `max_iter=1000` e `tol=1e-3` porque esses serão os valores padrão no Scikit-Learn 0.21.

```
sgd_reg = SGDRegressor(penalty="l2", max_iter=1000, tol=1e-3, random_state=42)
sgd_reg.fit(X, y.ravel()) # SGD Instância
sgd_reg.predict([[1.5]])
```

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 a metade do quadrado da norma ℓ2 do vetor de peso: isso é simplesmente Regressão de Ridge.

```
sgd_reg.score(X, y)
```

Atividade

Experimente valores diferentes para os hiperparametros max_iter e tol para encontrar um score melhor que o anterior.

Lasso Regression

A **Regressão do Operador de Retração e Seleção Mínima Absoluta** (geralmente chamada simplesmente de **Lasso Regression**) **é outra versão regularizada da Regressão Linear:**

- assim como a Regressão de Ridge, ela adiciona um termo de regularização à função de custo,
- mas usa a norma ℓ1 do vetor de peso em vez de metade o quadrado da norma ℓ2

Equation 4-10. Lasso Regression cost function

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

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

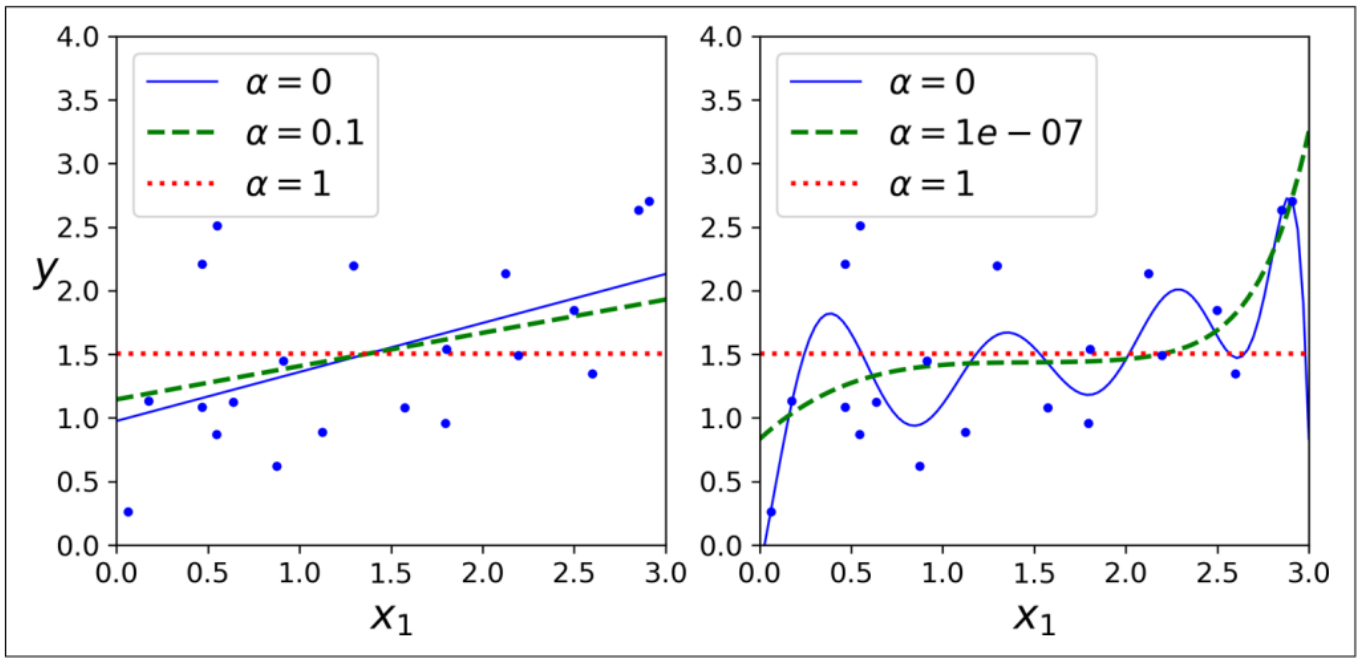


Figure 4-18. A linear model (left) and a polynomial model (right), both using various levels of Lasso regularization

Uma característica importante do **Lasso Regression** é que ele tende a eliminar os pesos dos recursos menos importantes (ou seja, defini-los como zero).

- Por exemplo, a linha tracejada verde no gráfico à direita na Figura 4-18 (com α = 10-7) parece quadrática, quase linear: todos os pesos para os recursos polinomiais de alto grau são iguais a zero.
- Em outras palavras, o Lasso Regression executa automaticamente a seleção de recursos e gera um modelo esparso (ou seja, com poucos pesos de recursos diferentes de zero).

(opcional) código para geração da figura comparativa

```
from sklearn.linear_model import Lasso

plt.figure(figsize=(8,4))
plt.subplot(121)
plot_model(Lasso, polynomial=False, alphas=(0, 0.1, 1), random_state=42)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.subplot(122)
plot_model(Lasso, polynomial=True, alphas=(0, 10**-7, 1), random_state=42)


plt.show()
```

Exemplos de Regressão Lasso com o Scikit-Learn usando uma solução de formato fechado

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

```
lasso_reg.score(X,y)
```

```
X_poly.shape
```

Exemplos de Regressão Lasso usando SGD

```
sgd_reg = SGDRegressor(penalty="l1", max_iter=1000, tol=1e-3, random_state=42)
sgd_reg.fit(X,y.ravel()) # SGD Instância
sgd_reg.predict([[1.5]])
```

```
sgd_reg.score(X,y)
```

A quantidade de características não é desafiante para vermos a supressão de alguma.

```
from sklearn.preprocessing import PolynomialFeatures
poly_features = PolynomialFeatures(degree=4, include_bias=False)
X_poly = poly_features.fit_transform(X)
```

```
from sklearn.linear_model import Lasso
lasso_reg = Lasso(alpha=0.1)
lasso_reg.fit(X_poly, y)
lasso_reg.score(X_poly, y)
```

```
lasso_reg.coef_
```

Algumas coef são praticamente 0.

Agora o SGD

```
sgd_reg = SGDRegressor(penalty="l1", max_iter=10000, tol=1e-3, random_state=42)
sgd_reg.fit(X_poly, y.ravel())
sgd_reg.score(X_poly, y.ravel())
```

O SGD padrão não conseguiu convergir devido a quantidade de características. Vamos ver o que ocorre durante o treinamento com verbose=1

```
sgd_reg = SGDRegressor(penalty="l1", max_iter=10000, tol=1e-3, random_state=42, verbose=1)
sgd_reg.fit(X_poly, y.ravel())
sgd_reg.score(X_poly, y.ravel())
```

Aparentemente o SGD está parando aumaticamente por ficar 5 iterações sem melhorar.

```
sgd_reg = SGDRegressor(penalty="l1", max_iter=10000, tol=1e-3, random_state=42, n_iter_no_change=100)
sgd_reg.fit(X_poly, y.ravel())
sgd_reg.score(X_poly, y.ravel())
```

```
sgd_reg.coef_
```

Atividade

Compare os resultados do SGD com regularização l2 e l1 usando os dados com 4 graus polinomiais.

```
sgd_reg = SGDRegressor(penalty="l2", max_iter=10000, tol=1e-3, random_state=42, n_iter_no_change=100)
sgd_reg.fit(X_poly, y.ravel())
sgd_reg.score(X_poly, y.ravel())
```

```
sgd_reg.coef_
```

```
sgd_reg = SGDRegressor(penalty="l1", max_iter=10000, tol=1e-3, random_state=42, n_iter_no_change=100)
sgd_reg.fit(X_poly, y.ravel())
sgd_reg.score(X_poly, y.ravel())
```

```
sgd_reg.coef_
```

Elastic Net

O **Elastic Net** é um meio termo entre Ridge Regression e Lasso Regression. O termo de regularização é uma mistura simples dos termos de regularização de Ridge e Lasso, e você pode controlar a proporção de mistura.

Quando r = 0, a Elastic Net é equivalente a Regressão de Ridge, e, quando r = 1, ela é equivalente a Regressão Lasso.

```
from sklearn.linear_model import ElasticNet
elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5, random_state=42)
elastic_net.fit(X, y)
elastic_net.predict([[1.5]])
```

```
elastic_net.score(X,y)
```

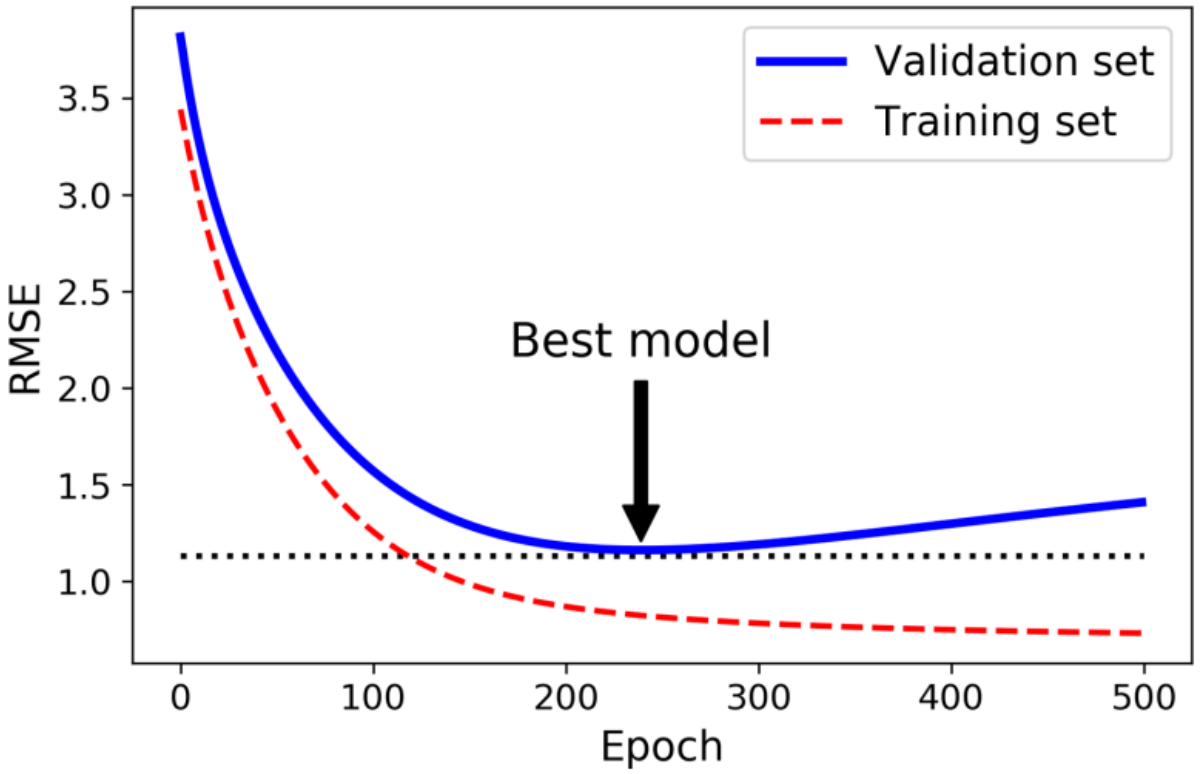
Então, quando você deve usar Regressão Linear simples (ou seja, sem qualquer regularização), Ridge, Lasso ou Elastic Net?

- É quase sempre preferível ter pelo menos um pouco de regularização, então geralmente você deve evitar a regressão linear simples.
- Ridge é um bom padrão, mas se você suspeitar que apenas algumas características são úteis, você deve preferir Lasso ou Elastic Net porque eles tendem a reduzir o peso dos recursos inúteis a zero (seleção de características automático).
- Em geral, o Elastic Net é preferível ao Lasso porque o Lasso pode se comportar de forma irregular quando o número de características é maior que o número de instâncias de treinamento ou quando várias características estão fortemente correlacionados.

https://scikit-learn.org/stable/modules/linear_model.html#elastic-net

Early Stopping (parada antecipada)

Uma maneira muito diferente de regularizar algoritmos de aprendizado iterativo, como Gradient Descent, é interromper o treinamento assim que o erro de validação atingir um mínimo.



▾ (Opcional) Código para criar o gráfico:

```
sgd_reg = SGDRegressor(max_iter=1, tol=-np.infty, warm_start=True,
                       penalty=None, learning_rate="constant", eta0=0.0005, random_state=42)

n_epochs = 500
train_errors, val_errors = [], []

for epoch in range(n_epochs):
    sgd_reg.fit(X_train_poly_scaled, y_train)

    y_train_predict = sgd_reg.predict(X_train_poly_scaled)
    y_val_predict = sgd_reg.predict(X_val_poly_scaled)

    train_errors.append(mean_squared_error(y_train, y_train_predict))
    val_errors.append(mean_squared_error(y_val, y_val_predict))

best_epoch = np.argmin(val_errors)
best_val_rmse = np.sqrt(val_errors[best_epoch])

plt.annotate('Melhor Modelo',
             xy=(best_epoch, best_val_rmse),
             xytext=(best_epoch, best_val_rmse + 1),
             ha="center",
             arrowprops=dict(facecolor='black', shrink=0.05),
             fontsize=16,
            )

best_val_rmse -= 0.03  # apenas para melhorar o gráfico
plt.plot([0, n_epochs], [best_val_rmse, best_val_rmse], "k:", linewidth=2)
plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="Conjunto de Validação")
plt.plot(np.sqrt(train_errors), "r--", linewidth=2, label="Conjunto de Treino")
plt.legend(loc="upper right", fontsize=14)
plt.xlabel("Epocas", fontsize=14)
plt.ylabel("RMSE", fontsize=14)


plt.show()
```

Analisando os Parâmetros do SGD

No SGD temos o parâmetro **early_stopping** com padrão False

Se **early_stopping=True** o SGD vai separar uma parte dos dados para fazer validação a cada iteração. Ele vai parar quando o erro de validação entre uma iteração e outra não reduzir ao menos o valor do **tol**.

```
sgd_reg = SGDRegressor(penalty="l1", max_iter=10000, tol=1e-3, random_state=42, n_iter_no_change=100, early_stoppin
sgd_reg.fit(X_poly, y.ravel())
sgd_reg.score(X_poly, y.ravel())
```

n_iter_no_change=100 não foi suficiente para a validação

```
sgd_reg = SGDRegressor(penalty="l1", max_iter=10000, tol=1e-3, random_state=42, n_iter_no_change=1000, early_stoppi
sgd_reg.fit(X_poly, y.ravel())
sgd_reg.score(X_poly, y.ravel())
```

```
sgd_reg.n_iter_
```

Podemos baixar o **tol** para um nível de erro menor.

```
sgd_reg = SGDRegressor(penalty="l1", max_iter=10000, tol=1e-6, random_state=42, n_iter_no_change=1000, early_stoppi
sgd_reg.fit(X_poly, y.ravel())
sgd_reg.score(X_poly, y.ravel())
```

```
sgd_reg.n_iter_
```

Podemos usar o parâmetro **verbose=1** para ver a evolução

```
sgd_reg = SGDRegressor(penalty="l1", max_iter=10000, tol=1e-6, random_state=42, n_iter_no_change=1000, early_stoppi
sgd_reg.fit(X_poly, y.ravel())
sgd_reg.score(X_poly, y.ravel())
```

```
sgd_reg.n_iter_
```

Até o momento não há uma forma prática de extrair o histórico dos valores de perda por validação durante as iterações no SGD.

(Opcional) Código para demonstrar a parada antecipada manual com o SGD

```
np.random.seed(42)
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 2 + X + 0.5 * X**2 + np.random.randn(m, 1)
```

```
plt.scatter(X, y)
```

```
X_train, X_val, y_train, y_val = train_test_split(X[:50], y[:50].ravel(), test_size=0.5, random_state=10)
```

```
from copy import deepcopy

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, random_state=42)

minimum_val_error = float("inf")
best_epoch = None
best_model = None

for epoch in 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 = deepcopy(sgd_reg)
```

```
best_epoch, best_model
```

```
best_model.score(X_val_poly_scaled, y_val)
```

Atividade

Experimente valores diferentes de iterações e tol e tente criar um modelo com score mais alto.

6) Logistic Regression (Regressão Logística)

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 que esse e-mail seja 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") e, caso contrário, prevê que não (ou seja, pertence à classe negativa, rotulado como “0”). Isso o torna um classificador binário.

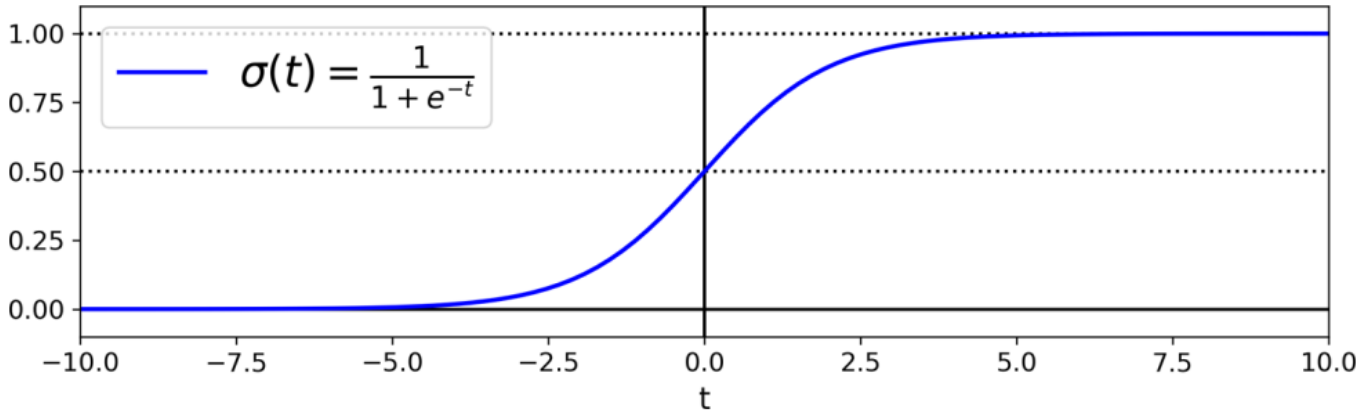
Estimando probabilidades

Assim como um modelo de regressão linear, um modelo de regressão logística calcula uma soma ponderada dos recursos de entrada (mais um termo de polarização), mas em vez de gerar o resultado diretamente como o modelo de regressão linear faz, ele gera a logística desse resultado

Equation 4-13. Logistic Regression model estimated probability (vectorized form)

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

A logística - notada σ(·) - é uma função sigmóide (ou seja, em forma de S) que gera um número entre 0 e 1. É definido como mostrado na Equação



Uma vez que o modelo de Regressão Logística tenha estimado a probabilidade p= hθ (x) de que uma instância x pertença à classe positiva, ele pode fazer sua previsão ŷ facilmente

Equation 4-15. Logistic Regression model prediction

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

Gráfico da Função Sigmóide

```
# função sigmóide
t = np.linspace(-10, 10, 100) # 100 parte no intervalo -10 e 10
sig = 1 / (1 + np.exp(-t))

plt.figure(figsize=(9, 3))
plt.plot([-10, 10], [0, 0], "k-")
plt.plot([-10, 10], [0.5, 0.5], "k:")
plt.plot([-10, 10], [1, 1], "k:")
plt.plot([0, 0], [-1.1, 1.1], "k-")
plt.plot(t, sig, "b-", linewidth=2, label=r"$\sigma(t) = \frac{1}{1 + e^{-t}}$")
plt.xlabel("t")
plt.legend(loc="upper left", fontsize=20)
plt.axis([-10, 10, -0.1, 1.1])
plt.show()
```

Função de Treinamento e Custo

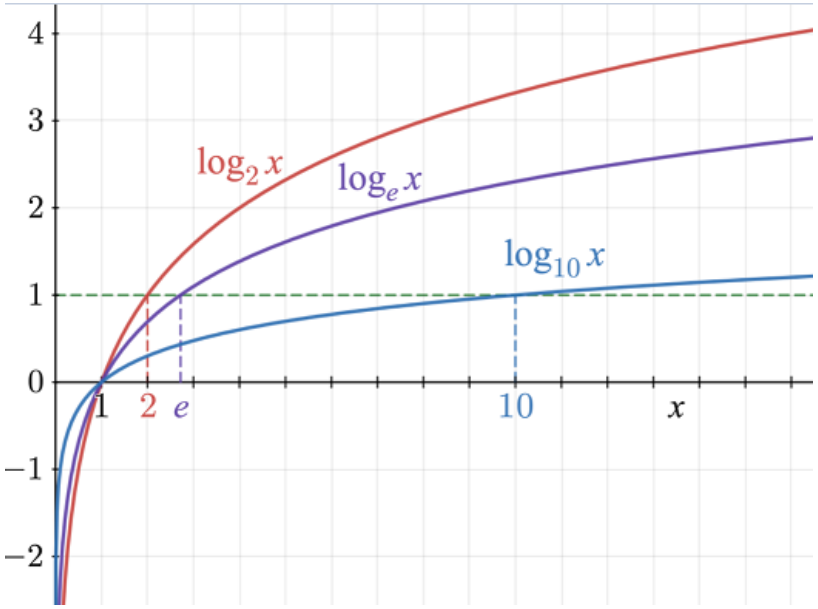
O objetivo do treinamento é definir o vetor de parâmetros θ para que o modelo estime altas probabilidades para instâncias positivas (y = 1) e baixas probabilidades para instâncias negativas (y = 0).

Essa ideia é capturada pela função de custo mostrada na Equação 4-16 para uma única instância de treinamento x.

Equation 4-16. Cost function of a single training instance

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

Essa função de custo faz sentido porque –log(t) cresce muito quando t se aproxima de 0, então 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.

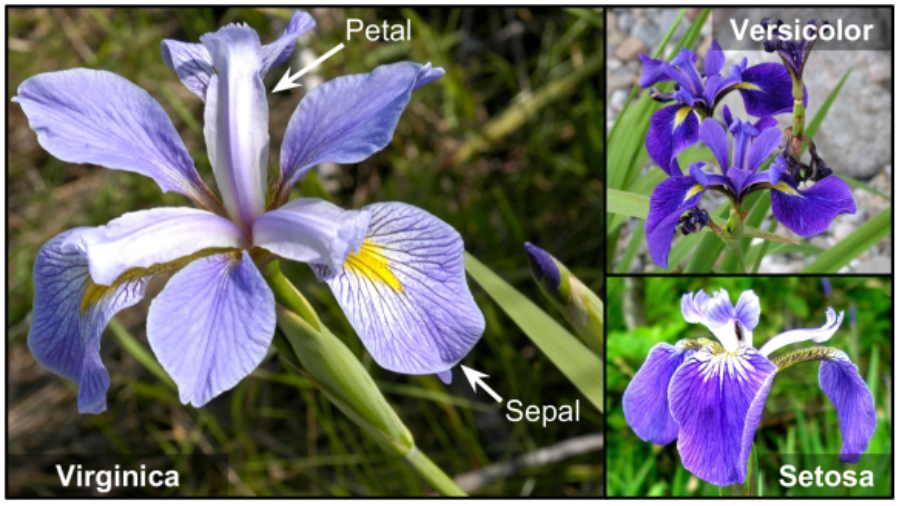


A má notícia é que não existe nenhuma equação de forma fechada conhecida para calcular o valor de θ que minimize essa função de custo (não há equivalente da Equação Normal).

A boa notícia é que essa função de custo é convexa, então Gradient Descent (ou qualquer outro algoritmo de otimização) é garantido para encontrar o mínimo global (se a taxa de aprendizado não for muito grande e você esperar o suficiente).

Limites de decisão

Vamos usar o conjunto de dados da íris para ilustrar a regressão logística.



Este é um famoso conjunto de dados 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

```
from sklearn import datasets
iris = datasets.load_iris()
list(iris.keys())
```

```
print(iris.DESCR)
```

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

```
max(X[:,1])
```

```
plt.plot(X[:, 0][y==1], X[:, 1][y==1], "bs", label="Iris versicolor")
plt.plot(X[:, 0][y==0], X[:, 1][y==0], "yo", label="Iris setosa")
plt.plot(X[:, 0][y==2], X[:, 1][y==2], "ro", label="Iris virginica")
plt.xlabel("petal length)", fontsize=14)
plt.ylabel("petal width)", fontsize=14)
plt.legend(loc="upper left", fontsize=14)
plt.show()
```

```
X = iris["data"][:, 3:]  # largura da petala (width)
y = (iris["target"] == 2).astype(int)  # 1 se Iris virginica, senão 0
```

```
X.shape
```

Observação: Para ser à prova de futuro, definimos **solver="lbfgs"** já que este será o valor padrão no Scikit-Learn 0.22.

```
from sklearn.linear_model import LogisticRegression
log_reg = LogisticRegression(solver="lbfgs", random_state=42)
log_reg.fit(X, y)
```

```
# Visualização do limite de decisão
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-", linewidth=2, label="Iris virginica")
plt.plot(X_new, y_proba[:, 0], "b--", linewidth=2, label="Not Iris virginica")
plt.legend(loc="upper left", fontsize=14)
plt.xlabel("Petal width", fontsize=14)
plt.ylabel("probability", fontsize=14)
plt.show()
```

Figura com limite de decisão detalhado (código oculto opcional)

```
#@title Figura com limite de decisão detalhado (código oculto opcional)
X_new = np.linspace(0, 3, 1000).reshape(-1, 1)
y_proba = log_reg.predict_proba(X_new)
decision_boundary = X_new[y_proba[:, 1] >= 0.5][0]

plt.figure(figsize=(8, 3))
plt.plot(X[y==0], y[y==0], "bs")
plt.plot(X[y==1], y[y==1], "g^")
plt.plot([decision_boundary, decision_boundary], [-1, 2], "k:", linewidth=2)
plt.plot(X_new, y_proba[:, 1], "g-", linewidth=2, label="Iris virginica")
plt.plot(X_new, y_proba[:, 0], "b--", linewidth=2, label="Not Iris virginica")
plt.text(decision_boundary+0.02, 0.15, "Decision  boundary", fontsize=14, color="k", ha="center")
plt.arrow(decision_boundary, 0.08, -0.3, 0, head_width=0.05, head_length=0.1, fc='b', ec='b')
plt.arrow(decision_boundary, 0.92, 0.3, 0, head_width=0.05, head_length=0.1, fc='g', ec='g')
plt.xlabel("Petal width (cm)", fontsize=14)
plt.ylabel("Probability", fontsize=14)
plt.legend(loc="center left", fontsize=14)
plt.axis([0, 3, -0.02, 1.02])

plt.show()
print('decision_boundary: ', decision_boundary)
```

```
log_reg.predict([[1.5]])
```

```
log_reg.predict([[1.7]])
```

A Figura abaixo mostra o mesmo conjunto de dados, mas desta vez exibindo duas características: largura e comprimento da pétala. Uma vez treinado, o classificador de Regressão Logística pode, com base nessas duas características, estimar a probabilidade de uma nova flor ser uma Iris virginica.

A linha tracejada representa os pontos onde o modelo estima uma probabilidade de 50%: este é o limite de decisão do modelo. Observe que é um limite linear.16 Cada linha paralela representa os pontos onde 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 ser Iris virginica, de acordo com o modelo.

Figura com o limite de decisão de duas características (código oculto opcional para gerar a figura)

```
#@title Figura com o limite de decisão de duas características (código oculto opcional para gerar a figura)

from sklearn.linear_model import LogisticRegression

X = iris["data"][:, (2, 3)]  # petal length, petal width
y = (iris["target"] == 2).astype(int)

log_reg = LogisticRegression(solver="lbfgs", C=10**10, random_state=42)
log_reg.fit(X, y)

x0, x1 = np.meshgrid(
        np.linspace(2.9, 7, 500).reshape(-1, 1),
        np.linspace(0.8, 2.7, 200).reshape(-1, 1),
    )
X_new = np.c_[x0.ravel(), x1.ravel()]

y_proba = log_reg.predict_proba(X_new)

plt.figure(figsize=(10, 4))
plt.plot(X[y==0, 0], X[y==0, 1], "bs")
plt.plot(X[y==1, 0], X[y==1, 1], "g^")

zz = y_proba[:, 1].reshape(x0.shape)
contour = plt.contour(x0, x1, zz, cmap=plt.cm.brg)


left_right = np.array([2.9, 7])
boundary = -(log_reg.coef_[0][0] * left_right + log_reg.intercept_[0]) / log_reg.coef_[0][1]

plt.clabel(contour, inline=1, fontsize=12)
plt.plot(left_right, boundary, "k--", linewidth=3)
plt.text(3.5, 1.5, "Not Iris virginica", fontsize=14, color="b", ha="center")
plt.text(6.5, 2.3, "Iris virginica", fontsize=14, color="g", ha="center")
plt.xlabel("Petal length", fontsize=14)
plt.ylabel("Petal width", fontsize=14)
plt.axis([2.9, 7, 0.8, 2.7])


plt.show()
```

Assim como os demais modelos lineares, os modelos de Regressão Logística podem ser regularizados usando penalidades de ℓ1 ou ℓ2.

O Scikit-Learn na verdade adiciona uma penalidade de ℓ2 por padrão.

Regressão Logistica Multiclasse

```
X = iris["data"][:, (2, 3)]  # petal length, petal width
y = iris["target"]
```

```
np.unique(y)
```

```
softmax_reg = LogisticRegression(solver="lbfgs", random_state=42)
softmax_reg.fit(X, y)
softmax_reg.score(X, y)
```

7) Exercicio

a) - O link a seguir faz a regressão linear da progressão da diabetes usando apenas uma característica de um dataset.

https://scikit-learn.org/stable/auto_examples/linear_model/plot_ols.html

Faça a regressão linear com pelo menos duas características e compare com a regressão do link de exemplo.

Você pode se informar mais sobre o dataset no link a seguir https://scikit-learn.org/stable/datasets/toy_dataset.html#diabetes-dataset

b) - No dataset de média de preços de casa, com o Scikitlean, use Regressão Linear, Polinomial, Rigde, Lasso e Elastic net. Compare os resultados.