APRENDIZADO DE MÁQUINA

PROF. RONALDO CRISTIANO PRATI

RONALDO.PRATI@UFABC.EDU.BR

BLOCO A, SALA 513-2

TÓPICOS DA AULA

- Implementações do algoritmo da descida do gradiente para regressão linear
  - Direta
  - Vetorizada (`numpy`)
  - Algebra Linear
- Regressão linear multivariada

FUNÇÃO DE CUSTO

- Vamos definir a função de custo (em python). As variáveis `x` e `y` são listas de tamanho m, e `theta` é uma lista de tamanho 2.

```
def costFunction(x,y,theta):
    error = 0
    m = len(x)
    for i in range(m):
        error += ((theta[0] + theta[1]*x[i]) - y[i])**2
    return error / (2*m)
```

DESCIDA DO GRADIENTE (1 PASSO)

- Calcula o gradiente em cada ponto e calcula o novo valor de θ.

```
def step_gradient(x,y,theta,alpha):

    new_theta = [0,0]
    m = len(x)

    theta_0_grad = []
    theta_1_grad = []

    for i in range(m):
        theta_0_grad.append((theta[0] + theta[1]*x[i]) - y[i])
        theta_1_grad.append(((theta[0] + theta[1]*x[i]) - y[i])* x[

    new_theta[0] = theta[0] - (alpha * sum(theta_0_grad)/m)
    new_theta[1] = theta[1] - (alpha * sum(theta_1_grad)/m)
    return(new_theta)
```

DESCIDA DO GRADIENTE (ITERANDO)

- Chama o método `step_gradient` iterativamente para atualizar o valor de θ. Armazena os valores intermediários para plotar

```
theta = [0,0]
theta_logs = [theta] # para plotar
cost_logs = [costFunction(x,y,theta)] # para plotar
n_iter=200
alpha = 0.001
for i in range(n_iter):
    theta = step_gradient(x,y,theta,alpha)
    theta_logs.append(theta) # para plotar
    cost_logs.append(costFunction(x,y,theta)) # para plotar
```

ALPHA = 0.03

ALPHA = 0.02

ALPHA = 0.01

ALPHA = 0.001

ALPHA = 0.0001

VETORIZAÇÃO

- Algumas linguagens (como `matlab` e `R`) permite o uso de técnicas de vetorização. Nessas linguagens, algumas operacoes são transparentemente aplicadas a vetores ou matrizes.
- Em `python`, a biblioteca `numpy` provê várias funcionalidades para explorar vetorização

FUNÇÃO DE CUSTO - VETORIZADA

- Uma versão da função de custo vetorizada é mostrada a seguir. Observe que não é utilizado o laço.
- As variáveis `x`, `y` e `theta` devem ser `numpy.array`

```
import numpy as np

def costFunction(x,y,theta):
    error = np.sum(((theta[0] + theta[1]*x) - y)**2)
    return error / (2*m)
```

DESCIDA DO GRADIENTE (1 PASSO)

```
def step_gradient(x,y,theta,alpha):

    theta_gradient = np.zeros(2)
    theta_gradient[0] = np.mean((theta[0] + theta[1]*x) - y)
    theta_gradient[1] = np.mean(((theta[0] + theta[1]*x) - y)*x)

    new_theta = theta - (alpha * theta_gradient)

    return(new_theta)
```

ALGEBRA LINEAR

- Suponha que nosso modelo seja

$$h_\theta(x) = -40 + 0.25x$$

- Queremos aplicá-la em uma base de dados com 4 casas

Tamanho
2104
1416
1534
852

ALGEBRA LINEAR

- Podemos calcular o valor predito pelo modelo acrescentando uma coluna de 1's ao conjunto de dados, e multiplicando pela "matriz" de coeficientes.

$$\begin{bmatrix} 1 & 2104 \\ 1 & 1416 \\ 1 & 1534 \\ 1 & 852 \end{bmatrix} \times \begin{bmatrix} -40 \\ 0.25 \end{bmatrix} = \begin{bmatrix} 486.0 \\ 314.0 \\ 343.5 \\ 173.0 \end{bmatrix}$$

- Isso pode ser muito mais eficiente computacionalmente que usar laços, e também facilita a codificação (se tivermos uma biblioteca de algegra linear)

GRADIENTE DESCENDENTE - ALGLIN

- Podemos reescrever a função de custo como:

$$J_\theta(x) = \frac{1}{2m} \sum (x\theta - y)^2$$

- e o seu gradiente como:

$$\frac{\partial}{\partial_\theta} J_\theta(x) = \frac{1}{m} \left((x\theta - y)^\intercal x\right)^\intercal$$

$$= \frac{1}{m} \left(x^\intercal \times (x\theta - y)\right)$$

- Para derivação do gradiente, veja aqui

GRADIENTE DESCENDENTE - ALGLIN

```
def costFunction(X, y, theta):
    error = np.dot(X, theta) - y
    return np.mean(error) / 2

def step_gradient(x,y,theta,alpha):
    m = len(x)
    theta = theta - (alpha/m) * np.dot(x.T, np.dot(x, theta) - y)
    return(theta)

x = np.column_stack((np.ones(len(x)),x))
theta = [0,0]
n_iter=200
alpha = 0.001
for i in range(n_iter):
    theta = step_gradient(x,y,theta,alpha)
```

REGRESSÃO LINEAR MULTIVARIADA

- Múltiplas variáveis = múltiplos atributos
- No problema anterior, tinhamos
  - x = tamanho da casa (em ft^2)
  - y = preço da casa
- Mas podemos adicionar novas variáveis, tais como
  - $x_1, x_2, x_3, x4$ são quatro atributos:

    x_1 = tamanho (ft^2)

    x_2 = número de quartos

    x_3 = number de andares

    x_4 = idade da casa (anos)

    y = preço das casas

MAIS NOTAÇÃO

- n = número de atributos $(n = 4)$
- m = número de exemplos
- x^i = vetor de entrada para um exemplo. i é o índice do conjunto de treinamento, então x^3 é, por exemplo a 3^a casa. x é um vetor n-dimensional
- x^i_j é o valor do atributo j do i-ésimo exemplo de treino. Por exemplo, x^3_2 é o número de quartos da terceira casa.

FORMA DA HIPÓTESE

- Anteriormente nossa hipótese era da forma:

$$h_\theta(x) = \theta_0 + \theta_1 x$$

- Agora que temos multiplos atributos:

$$h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x + \theta_3 x + \theta_4 x$$

- Se adicionarmos uma feature adicional $x_0 = 1$, em notação vetorial temos:

$$h_\theta(x) = \overrightarrow{\theta}\, x$$

DESCIDA DO GRADIENTE

- Os parâmetros ainda podem ser determinados pela função de custo
- Atualizamos cada parâmetro θ_j fazendo

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

- em que

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{m} \sum_{i}^{m} (h_\theta(x^i) - y^i) x_j^i$$

- A forma vetorial se ajusta ao número de variáveis

DIFERENÇA DE ESCALA

- Atributos em diferentes escalas podem atrapalhar a descida do Gradiente. Por exemplo, se tivermos
  x_1 = Tamanho (0 - 2000 ft^2)
  x_2 = número de quartos (1-5)
  Os contornos gerados pelo plot de θ_1 \emph{vs} θ_2 será alongado em um dos eixos e estreito no outro

DIFERENÇA DE ESCALA

- Colocar os atributos na mesma escala pode fazer o algoritmo covergir mais rapidamente
  - As linhas de contorno se parecerão mais com um círculo
- Possíveis opções
  - Ajustar entre 0 e 1: $\frac{x_j^i - \min(x_j)}{\max(x_j) - \min(x_j)}$
  - Normalização: $\frac{x_j^i - \mu(x_j)}{\sigma(x_j)}$

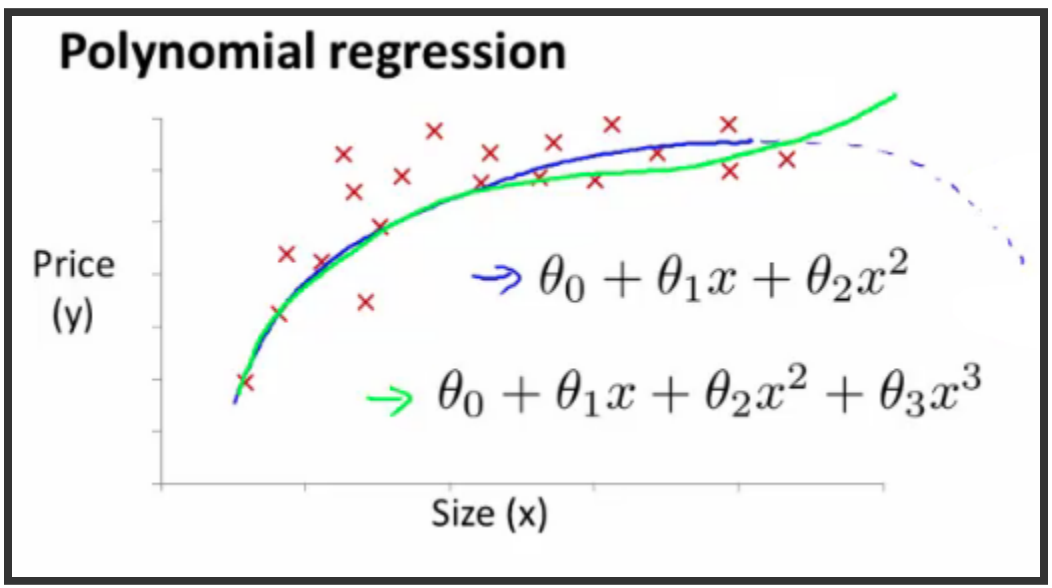
CRIAÇÃO DE ATRIBUTOS

- Suponha que tenhamos os atributos:
  - x_1 = Largura da frente to terreno
  - x_2 = Profundidade do terreno
- Podemos criar um novo atributo x_3 = Área fazendo

$$x_3 = x_1 * x_2$$

- Em muitos casos, a definição de novos atributos pode gerar modelos melhores!

REGRESSÃO POLINOMIAL



REGRESSÃO POLINOMIAL

- Como podemos ajustar um polinômio de grau $n > 1$ aos dados?
- Uma ideia simples é criar novos atribtos:

$$x_1 = x$$
$$x_2 = x^2$$
$$x_3 = x^3$$

- Criarndo atributos como esses e aplicando o algoritmo de regressão linear podemos fazer regressão polinomial!

REGRESSÃO POLINOMIAL

- Reescalar os atributos se torna ainda mais importated nesse caso
- Ao invés de um polinômio convencional, também podemos usar $x^{\frac{1}{z}}$, $z \in \{2, 3, \dots\}$ (i.e., raiz quadrada, raiz cúbica, etc)
- Podemos criar muitos atributos - posteriormente vamos ver algoritmos para selecionar os melhores atributos

EQUAÇÃO NORMAL

- Para alguns problemas de regressão linear, a **equação normal** provê uma melhor solução
  - Até agora estivemos usando a descida do Gradiente
  - Algoritmo iterativo que dá passos até convergir
- A equação normal resolve θ analiticamente
  - Resolve para o valor ótimo de θ em um único passo
- Tem vantagens e desvantagens

EQUAÇÃO NORMAL

- Vamos reescrever a função de custo Como

$$
\begin{aligned}
J_\theta(x) &= \frac{1}{2m}(x\theta - y)^\intercal(x\theta - y) \\
&= \frac{1}{2m}((x\theta)^\intercal - y^\intercal)(x\theta - y) \\
&= \frac{1}{2m}(x\theta)^\intercal(x\theta) - (x\theta)^\intercal y - y^\intercal(x\theta) + y^\intercal y \\
&= \frac{1}{2m}\theta^\intercal x^\intercal x\theta - 2(x\theta)^\intercal + y^\intercal y
\end{aligned}
$$

EQUAÇÃO NORMAL

- Calculando a derivada

$$\frac{\partial J}{\partial_\theta} = 2x^\intercal x\theta - 2x^\intercal y$$

- Igualando a zero (para achar o mínimo)

$$2x^\intercal x\theta - 2x^\intercal y = 0$$

$$2x^\intercal x\theta = 2x^\intercal y$$

$$\theta = (x^\intercal x)^{-1}x^\intercal y$$

QUANDO USAR?

- Gradiente descendente:
  - Precisa escolher a taxa de aprendizado
  - Precisa de muitas iterações - pode ser demorado
  - Funciona bem quando n é grande (adequado para Big Data)

QUANDO USAR?

- Equação Normaliza
  - não precisa escolher a taxa de aprendizado
  - não precida iterar, checar convergencia, etc.
  - Precisa inverter a matriz $x^\intercal x$ (inverso de uma matrix $n \times n$)
  - Devagar se n é grande (pode ser bem lento)