Regressão Linear: Implementação e Teoria

Glauco Fleury

1 Introdução à Regressão Linear

Trata-se em suma de um modelo de 'curve fitting': dado um domínio que apresenta diversas features, isto é, tem um número D de características (representado matematicamente por vetores $x \in \mathbb{R}^D$), e uma imagem y a qual se deseja estimar com a função, o objetivo é encontrar uma curva a qual melhor se encaixe nos dados já presentes, de modo que seja possível utilizar a função que a descreve para tentar predizer resultados futuros.

A regressão linear assume que a função buscada terá o seguinte formato:

$$\begin{aligned} f(x,\theta) &= \theta_0 + \theta_1 x_1 + \theta_2 x_2 + ... + \theta_D x_D \\ &= \theta \cdot x \end{aligned} \tag{1}$$

Isto é, ela assume que a variável a ser prevista pode ser entendida como a combinação linear das dimensões do vetor de input. Do modo como está escrita, a função buscada será necessariamete, uma reta (2D), plano (3D), ou qualquer tipo de hiperplano para mais dimensões.

1.1 Ruído e Probabilidade

Considerando $y = \hat{y} + \epsilon = f(x,\theta) + \epsilon$, sendo ϵ o ruído intrínseco à coleta de dados e $f(x,\theta) = \hat{y}$ a melhor previsão do modelo acerca do real resultado, torna-se necessária a abordagem estatística sobre a modelagem da regressão. Ela toma o seguinte formato:

$$p(y|x,\theta) \rightarrow y \sim \mathcal{N}(x \cdot \theta, \sigma^2) \tag{2}$$

O que basicamente significa que assumimos que a chance de $\hat{y}$ "acertar" y é dada por uma distribuição normal, em que a maior parte das vezes ele acerta, mas pode errar, porém a estimativa tem chances menores de cometer erros quanto mais absurdos eles são.

Considerando que o ruído ϵ advém de uma função de densidade de probabilidade normal, com média 0 (a maior parte das vezes, $\epsilon = 0$) e desvio padrão $= \sigma$, podemos escrever que $\epsilon \sim \mathcal{N}(0, \sigma^2)$. A variança dessa normal é idêntica à outra devido ao fato do ruído ser o fator comum de variação em ambos os casos.

2 Maximum Likelihood Estimation

Maximum Likelihood Estimation se resume a buscar o vetor de parâmetros θ_{MLE} tal que a curva descrita melhor se adeque aos dados, parametrizando-a de modo a encaixá-la nos pontos do espaço $\mathbb{R}^D$. Para isso, é desejável maximizar a probabilidade de que cada resultado y advenha de nosso modelo probabilístico (cuja média nós definiremos com $f(x, \theta) = x^T\theta$). Em resumo, buscamos (para um número N de vetores x):

$$\theta_{MLE} = \arg\max_{\theta} \prod_{n=1}^{N} p(y_n|x_n, \theta) \tag{3}$$

$$= \arg\max_{\theta} \prod_{n=1}^{N} \mathcal{N}(y|x \cdot \theta, \sigma^2) \tag{4}$$

$$= \arg\min_{\theta} - \Sigma_{n=1}^{N} \log(\mathcal{N}(y|x \cdot \theta, \sigma^2)) \tag{5}$$

Sabe-se que $\mathcal{N}(\mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}$, e, portanto, é possível rescreever θ_{MLE} na forma:

$$\theta_{MLE} = \arg\min_{\theta}[-N\log(\sigma) - \frac{N}{2}\log(2\pi) - \frac{1}{2\sigma^2}\Sigma_{n=1}^{N}(y_n - x_n^T\theta)^2] \tag{6}$$

$$= \arg\min_{\theta}[-\frac{1}{2\sigma^2}\Sigma_{n=1}^{N}(y_n - x_n^T\theta)^2] \tag{7}$$

Para facilitar, vamos escrever o somatório na forma matricial, e também igualar a equação acima a uma função, de tal forma que:

$$\theta_{MLE} = \arg\min_{\theta}[L(\theta)] \tag{8}$$

$$L(\theta) = -\frac{1}{2\sigma^2}||(Y - X\theta)||^2 \tag{9}$$

Onde $X = [x_1, x_2, ..., x_n]^T$ e $Y = [y_1, y_2, ..., y_n]^T$.

A partir daqui, para achar o nosso parâmetro, fica claro que o problema se torna a minimazação de uma função quadrada. Como a Hessiana de $L(\theta)$ é positiva e semi-definida, fica claro que estamos lidando com um problema de otimização convexa, ou seja, é possível encontrar uma solução óptima.

Efetuando $\frac{dL}{d\theta} = 0$ (tal qual ensinam em cálculo I), encontra-se uma fórmula para a otimização:

$$\theta_{MLE} = (X^TX)^{-1}X^TY \tag{10}$$

O problema com essa fórmula está em encontrar a matriz inversa. O algoritmo atual mais rápido para tal apresenta complexidade $O(n^3)$, o que é subóptimo, para dizer o mínimo. Como alternativa, é possível utilizar um método

iterativo que aproxime nosso tão sonhado vetor de parâmetros. Nesse projeto, escolhi particularmente o famoso "gradient descent" para estimar os valores desconhecidos.

2.1 Expansão Polinomial

Como eu havia dito no começo, esse modelo de regressão linear é limitado pela presença de formas lineares (hiperplanos) para descrever nossas previsões. Observe o seguinte caso, por exemplo:

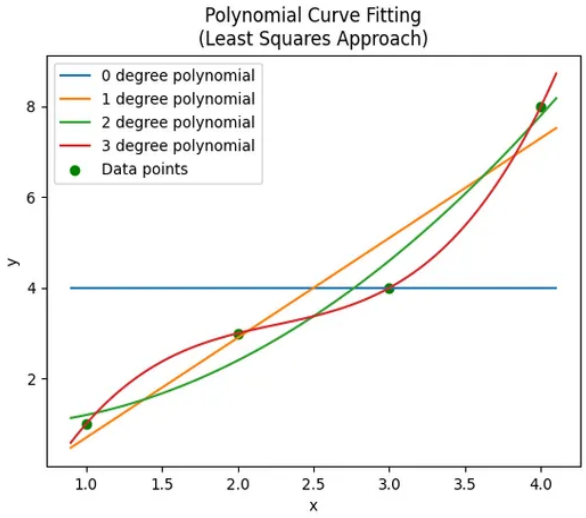

Figure 1: modelagem com diferentes graus de polinômios

Claramente a reta não é adequada para tentar descrever o padrão descrito pelos dados. A solução? Expandir polinomialmente as dimensões analisadas, a fim de capturar padrões mais complexos. A isso damos o nome de "polynomial expansion", uma forma de inclusão de features.

Formalmente, o que buscamos é uma forma de construir um novo vetor, chamado de feature vector ($\phi(x)$), o qual mudará o domínio de nossa função para incluir dimensões não lineares. A função resultante permanece uma combinação linear de dimensões, já que todos os parâmetros tem grau 1. Desse modo, para um feature vector qualquer: $\phi(x) : \mathbb{R}^D \rightarrow \mathbb{R}^K$

O feature vector utilizado nesse projeto é conhecido como "expansão polinomial": ele expande o vetor x para incluir combinações de grau maior de seus componentes. Dessa forma, se $x = [x_1, x_2, ..., x_n]^T$:

$$\phi(x) = [CR(x_1, x_2, ..., x_n)]^T \tag{11}$$

CR é uma sigla para combinação com reposição. Por exemplo, para $x =$

$[x_1, x_2]^T$ com grau 2 de expansão, e $x' = [x_1]$ com grau 5, teremos, respectivamente:

$$\phi(x) = [x_1, x_2, x_1^2, x_1 x_2, x_2^2]^T \tag{12}$$

$$\phi(x') = [x_1, x_1^2, x_1^3, x_1^4, x_1^5]^T \tag{13}$$

Dessa maneira, é possível realizar justamente o que figura anterior exibe: criar curvas com curvatura não nula, que se adequem melhor à descrição do problema, encontrando agora um vetor de pesos $\theta \in \mathbb{R}^K$.

2.2 Regularização

O que acontece quando nosso modelo é perfeito para os dados que utilizamos? Digamos que, para K pontos de dados que eu tenha, eu crie uma função com K dimensões; nesse caso, o modelo terá uma dimensão x_k para cada ponto P_k, e acabará por passar precisamente por todos estes pontos perfeitamente, tornando $L(\theta) = 0$. O problema com isso é a perda de generalidade: o modelo "decorou" o caso utilizado para treiná-lo, e, em qualquer outro conjunto de pontos fornecidos que advenham da mesma distribuição-fonte desse caso, ele terá uma performance horrível.

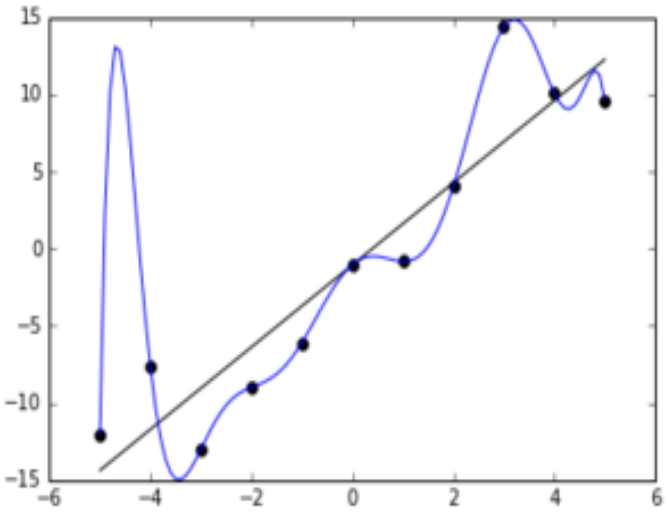

Figure 2: exemplo de overfitting

Mesmo que haja menos dimensões comparado ao número de pontos, o modelo ainda pode generalizar muito mal. A fim de prevenir ainda mais isso, introduz-se o bridge-regularization term, somando à loss function um termo que penaliza a adição de parâmetros muito elevados:

$$L(\theta, \lambda) = -\frac{1}{2\sigma^2}||(Y - X\theta)||^2 + \lambda||\theta||_2^2 \tag{14}$$

Onde λ é o coeficiente de aprendizado ("Learning Rate"), proporcional ao quanto queremos punir nosso modelo por overfitting, e o 2 embaixo do módulo de θ simboliza a norma euclidiana (é possível utilizar outras também, mas foi essa a que eu usei).

3 Implementação

3.1 Design Matrix

Essa matriz, em suma,contém os pontos do domínio do dataset após aplicar a polynomial expansion. Portanto, suas dimensões são de: K por $\frac{(N+R-1)!}{R!(N-1)!}$, sendo K o número de exemplos providenciados, e o número de colunas igual a $C(N, R)$ (número de combinações com repetição para N objetos e R amostragens).

Visualmente, para uma transformação $\phi(x) : \mathbb{R}^D \to \mathbb{R}^K$, a Design Matrix $\Phi(X)$ será dada por:

$$\Phi(X) = \begin{bmatrix} 1 & \phi(x_1^1) & \phi(x_2^1) & ... & \phi(x_N^1) \\ 1 & \phi(x_1^2) & \phi(x_2^2) & ... & \phi(x_N^2) \\ ... & ... & ... & ... & ... \\ 1 & \phi(x_1^K) & \phi(x_2^K) & ... & \phi(x_N^K) \end{bmatrix}$$

Onde os primeiros termos serem 1 satisfazem a necessidade de ter o termo independente θ_0 de $f(x, \theta)$ (isso ficará mais claro em gradient descent), e a notação x_i^j nesse caso indica que x pertence à i-ésima dimensão do j-ésimo vetor de treinamento. Por exemplo, para uma matriz com 3 vetores-exemplo de dimensão 2 em que deseja-se expandir polinomialmente até a terceira potência, teríamos:

$$X = \begin{bmatrix} x_1^1 & x_2^1 \\ x_1^2 & x_2^2 \\ x_1^3 & x_2^3 \end{bmatrix}$$

$$\Phi(X) = \begin{bmatrix} 1 & x_1^1 & x_2^1 & (x_1^1)^2 & x_1^1 x_2^1 & (x_2^1)^2 \\ 1 & x_1^2 & x_2^2 & (x_1^2)^2 & x_1^2 x_2^2 & (x_2^2)^2 \\ 1 & x_1^3 & x_2^3 & (x_1^3)^2 & x_1^3 x_2^3 & (x_2^3)^2 \end{bmatrix}$$

A design matrix, no fim das contas, nada mais é além de toda a informação necessária para treinar o modelo, já que contém todos os vetores de treinamento do dataset, e também apresenta-os expandidos ao grau que bem desejarmos.

3.2 Gradient Descent

A idéia básica do gradient descent é tentar minimizar a Loss Function descrita na equação 14 via o cálculo do vetor gradiente, que aponta para a direção

de máxima mudança positiva na função (logo, basta invertê-lo para adquirir a direção de máxima mudança negativa). Tudo se resume a um algoritmo muito simples:

1. calcular $\nabla L(\theta, \lambda)$
2. atualizar $\theta = \theta - \alpha_1 \nabla L(\theta, \lambda)$
3. repetir o processo para o mesmo $\Phi(X)$

Sendo que o mesmo vale para o gradiente de σ. Algumas ressalvas importantes:

- o gradiente na verdade não precisa ser necessariamente um vetor; ele pode ser um tensor de qualquer grau, sendo que ele só o é nesse caso graças ao fato da regressão linear trabalhar com uma função $f : \mathbb{R}^K \to \mathbb{R}$ (generalizando: para $f : \mathbb{R}^E \to \mathbb{R}^W$, ∇f tem dimensões $W \times E$)
- α_1 é a constante de aprendizado para a atualização dos pesos; como usarei gradient descent para estimar simultaneamente os valores de σ e θ, eles terão constantes igual a α_2, α_1, respectivamente. É necessário ajustar α com cuidado: passos muito largos podem provocar overshooting, e os pequenos, demorarem demais para convergir ao ideal.
- apesar de muito famoso, o gradient descent não é único método iterativo para aproximação de extremos funcionais (quem diria); procure também pelo método de Newton.

Dito isso, como fazemos o computador calcular esse gradiente? Primeiro, é necessário calculá-lo manualmente:

$$L(\theta, \lambda, \sigma) = -\frac{1}{2\sigma^2}||(Y - X\theta)||^2 + \lambda||\theta||_2^2 \tag{15}$$

$$= -\frac{1}{2\sigma^2}[\Sigma_{n=1}^{N}(y_n - x_n^T\theta)^2] + \lambda||\theta||_2^2 \tag{16}$$

$$\nabla L_\theta = \frac{dL}{d\theta} \tag{17}$$

$$= -\frac{1}{2\sigma^2}[\Sigma_{k=1}^{K}(y_n - \theta^T\phi(x_n))\phi(x_n)] + 2\lambda\theta \tag{18}$$

$$= [\frac{\partial L}{\partial \theta_1}, \frac{\partial L}{\partial \theta_2}, ..., \frac{\partial L}{\partial \theta_K}] \tag{19}$$

$$\nabla L_\sigma = \frac{dL}{d\sigma} \tag{20}$$

$$= -\frac{1}{2\sigma^4}[\Sigma_{k=1}^{K}(y_n - \theta^T\phi(x_n))^2] \tag{21}$$

Daí em diante, baste seguir o algoritmo já descrito acima para ∇L_θ e ∇L_σ, iterando até um ponto em que se estime já ser o suficiente.