

Aprendizado de Máquinas

Autor: João Florindo

Sumário

1	Introdução ao Aprendizado de Máquinas	9
1.1	Introdução	9
1.1.1	Definição	9
1.1.2	O que é Aprendizado de Máquinas	10
1.1.3	Aprendizado Supervisionado	10
1.2	Aprendizado Não Supervisionado	11
1.3	Representação do Modelo	12
2	Regressão Linear	15
2.1	Função de custo	15
2.2	Intuição da função de custo	15
2.3	Gradiente Descendente	17
2.4	Intuição do Gradiente Descendente	20
2.5	Gradiente Descendente na Regressão Linear	21
2.6	Vetorização	24
2.7	Regressão Linear Multivariada	25
2.8	Gradiente Descendente na Regressão Linear Multivariada	26
2.9	Condições Assumidas	27
2.10	Aspectos Práticos - Normalização dos Atributos	27
2.11	Aspectos Práticos - Taxa de Aprendizado	28
2.12	Atributos e Regressão Polinomial	30
2.13	Cálculo Analítico dos Parâmetros - Equação Normal	31
2.13.1	Derivando a equação normal	32
2.14	E se $X^T X$ não for Inversível?	33
3	Regressão Logística	35
3.1	Classificação	35
3.2	Regressão Logística - Função de Hipótese	35
3.3	Fronteira de Decisão	36
3.4	Função de Custo	38
3.5	Versão Compacta da Função de Custo e Gradiente Descendente	40
3.6	Otimização Avançada	41
3.7	Classificação multi-classes: <i>One-vs-all</i>	42
3.8	Dedução da Função de Custo	43

3.9	Dedução do Gradiente	44
4	Regularização	45
4.1	<i>Overfitting</i>	45
4.2	Função de Custo Regularizada	46
4.3	Régressão Linear Regularizada	47
4.3.1	Equação Normal	48
4.4	Régressão Logística Regularizada	49
5	Redes Neurais	51
5.1	Redes Neurais: Hipóteses não lineares	51
5.2	Neurônios e o Cérebro	52
5.2.1	Hipótese do algoritmo de aprendizagem único	53
5.3	Redes Neurais: Representação do Modelo I	53
5.4	Redes Neurais: Representação do Modelo II	56
5.4.1	Atributos aprendidos pela rede	58
5.4.2	Arquitetura da rede	58
5.5	Exemplos e Intuições I	58
5.5.1	Exemplo 1: função AND	60
5.5.2	Exemplo 2: função OR	61
5.6	Exemplos e Intuições II	61
5.6.1	Função NOT	61
5.6.2	Função (NOT x_1) AND (NOT x_2)	62
5.6.3	Juntando as peças: x_1 XNOR x_2	62
5.7	Redes Neurais: Classificação Multi-classes	63
5.8	Redes Neurais: Função de Custo	65
5.8.1	Função de custo	65
5.9	Algoritmo <i>Backpropagation</i>	66
5.9.1	Cálculo do gradiente: <i>backpropagation</i>	67
5.9.2	<i>Backpropagation</i> geral	68
5.10	Intuição do <i>Backpropagation</i>	68
5.11	Nota de implementação: vetorizando parâmetros	70
5.12	Checagem do Gradiente	71
5.13	Inicialização Aleatória	73
5.14	Juntando as Peças	74
5.14.1	Treinamento de uma rede neural	74
5.15	Dedução do <i>backpropagation</i>	76
6	Avaliação e Seleção de Modelos	79
6.1	O próximo passo	79
6.2	Avaliando uma hipótese	80
6.3	Seleção de Modelo: Conjuntos de Treino, Validação e Teste	81
6.4	Diagnosticando Viés vs Variância	83
6.5	Viés/Variância e Regularização	85
6.5.1	Escolha de λ	86
6.5.2	Influência da regularização no erro de treino e validação .	87

6.6	Curvas de Aprendizado	88
6.7	O Próximo Passo Revisitado	90
6.7.1	Redes neurais e <i>overfitting</i>	91
6.8	Prioridade no Desenvolvimento de um Sistema de Aprendizado de Máquinas	91
6.9	Análise de Erro	93
6.9.1	Exemplo de análise de erro	93
6.9.2	Importância da avaliação numérica	93
6.10	Métricas de Erro para Classes Desbalanceadas	94
6.10.1	<i>Precision/Recall</i>	94
6.11	<i>Trade off</i> de <i>Precision</i> e <i>Recall</i>	95
6.11.1	F_1 score (F score)	96
6.12	Usando Grandes Conjuntos de Dados	97
7	Máquina de Vetores de Suporte	99
7.1	Máquina de Vetores de Suporte (SVM)	99
7.1.1	SVM	100
7.2	SVM - Intuição da Margem Larga	101
7.3	Matemática da Classificação de Margem Larga	103
7.3.1	Produto interno	103
7.3.2	Fronteira de decisão do SVM	103
7.4	SVM - Kernels I	104
7.5	SVM - Kernels II	107
7.5.1	SVM com kernel	108
7.5.2	Parâmetros do SVM	108
7.6	Usando um SVM	109
7.6.1	Kernels alternativos	109
7.6.2	Classificação multiclasse	109
7.6.3	Comparação com outros classificadores	110
8	Árvores de Decisão	111
8.1	Árvore de Decisão	111
8.2	Função de Perda	112
8.2.1	Problema	114
8.2.2	Regressão	115
8.3	Outras Considerações	116
8.3.1	Atributos Categóricos	116
8.3.2	Regularização	116
8.3.3	Tempo de Execução	117
8.3.4	Perda de Estrutura Aditiva	117
8.3.5	Melhorias	118

9 Ensemble	119
9.1 Ensemble	119
9.2 <i>Bagging</i>	119
9.2.1 <i>Bootstrap</i>	119
9.2.2 Agregamento	120
9.2.3 <i>Bagging</i> + Árvores de Decisão	120
9.3 <i>Boosting</i>	120
9.3.1 Adaboost	121
9.4 <i>Forward Stagewise Additive Modeling</i>	122
9.5 Gradient Boosting	122
10 Agrupamento	125
10.1 Aprendizado Não Supervisionado - Agrupamento (<i>Clustering</i>)	125
10.2 Agrupamento - K-médias (K-means)	126
10.3 Função Objetivo	128
10.4 Inicialização Aleatória	128
10.5 Número de Clusters	129
11 Redução de Dimensionalidade	131
11.1 Redução de Dimensionalidade - Motivação I: Compressão	131
11.2 Motivação II: Visualização	132
11.3 Análise de Componentes Principais - Formulação	134
11.4 PCA - Algoritmo	135
11.5 Reconstrução	137
11.6 Escolhendo o Número de Componentes Principais	137
11.7 Aplicação do PCA	139
11.7.1 Maus usos de PCA	140
12 Detecção de Anomalia	141
12.1 Detecção de Anomalia - Motivação	141
12.2 Distribuição Gaussiana	142
12.3 Algoritmo	144
12.4 Desenvolvimento e Avaliação de um Sistema de Detecção de Anomalia	145
12.5 Detecção de Anomalia <i>vs</i> Aprendizado Supervisionado	147
12.6 Escolhendo os Atributos	147
12.6.1 Análise de erro	148
12.7 Distribuição Gaussiana Multivariada	149
12.8 Detecção de Anomalia Usando a Distribuição Gaussiana Multivariada	150
12.8.1 Relação com o modelo original	153
13 Sistemas de Recomendação	155
13.1 Sistemas de Recomendação - Fomulação	155
13.2 Recomendações Baseadas em Conteúdo	156
13.3 Filtragem Colaborativa	157

13.4 Filtragem Colaborativa - Algoritmo	158
13.5 Fatorização de Matrix de Baixo Posto (<i>Low Rank Matrix Factorization</i>)	159
13.5.1 Encontrando filmes relacionados	160
13.6 Normalização pela Média	160
14 Aprendizado em Larga Escala	163
14.1 Aprendizado de Máquinas em Larga Escala	163
14.2 Gradiente Descendente Estocástico	164
14.3 Gradiente Estocástico de Mini-Lote (<i>Mini-Batch</i>)	166
14.4 Convergência do Gradiente Descendente Estocástico	167
14.5 Aprendizado <i>Online</i>	168
14.5.1 Outro exemplo - Busca de produtos (“aprender a buscar”) .	169
14.6 <i>MapReduce</i> e Paralelismo de Dados	169
15 Exemplo Completo de Aplicação	173
15.1 Exemplo de Aplicação: OCR em Imagens	173
15.2 Janelas Deslizantes	173
15.3 Síntese de Dados Artificiais	177
15.4 <i>Ceiling Analysis</i> : Em Qual Parte do Pipeline se Concentrar? .	180
15.4.1 Outro exemplo	180

Capítulo 1

Introdução ao Aprendizado de Máquinas

1.1 Introdução

1.1.1 Definição

Sub-área da Inteligência Artificial (IA).

- **Data Science:** Processamento de (grandes quantidades de) dados visando identificar padrões e tendências
- **Inteligência Artificial (IA):** Máquina fazendo tarefas (“inteligentes”) de humanos (possivelmente com maior competência)
- **Aprendizado de Máquinas:** Conjunto de algoritmos que permite a um computador aprender certa tarefa sem ser explicitamente programado para aquilo.

Mas existe *Data Science* sem IA? SIM: armazenamento de dados, sistemas distribuídos, etc.

E existe IA sem *Machine Learning*? SIM: IA clássica (lógica simbólica), sistemas baseados em regras, *fuzzy*, computação evolutiva, *chatbots* mais simples, etc.

O Aprendizado de Máquinas está em tudo, em uma busca no Google (rankamento), na detecção de amigos na foto do Facebook, no filtro anti-spam, etc.

Sistemas de recomendação Netflix, Amazon, Spotify, etc.

Aplicabilidade muito ampla também na ciência e na indústria: medicina, biologia (genômica), gerenciamento de grandes quantidades de dados.

Carro autônomo, reconhecimento de escrita (endereçamento de CEP nos EUA), NLP (Google Translate), visão computacional.

Compreensão do cérebro humano.

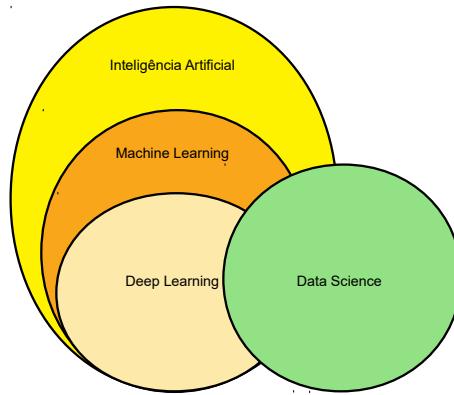


Figura 1.1: *Data Science vs Inteligência Artificial vs Aprendizado de Máquinas.*

1.1.2 O que é Aprendizado de Máquinas

Arthur Samuel 1950s - Conjunto de algoritmos que permite a um computador aprender certa tarefa sem ser explicitamente programado para aquilo.

Programou um jogo de damas que jogava contra ele mesmo e observava as posições vencedoras e perdedoras. O jogo superou o próprio Arthur!

Mais formal: Tom Mitchell - Programa de computador aprende a partir de uma experiência E em relação a uma tarefa T e medida de performance P , se essa performance medida por P em E aumenta com a experiência E .

EX. 1: No jogo de damas, E seria a experiência do programa jogando contra ele mesmo dezenas de milhares de vezes, T a tarefa de jogar damas, P a probabilidade de vencer o próximo jogo.

EX. 2: Em um filtro anti-spam, T é a tarefa de classificar emails como spam ou não, E é a experiência de observar o usuário rotulando como spam ou não, P seria a fração de emails rotulados corretamente pelo filtro.

Dois tipos principais de algoritmos de aprendizado: supervisionado e não supervisionado. Há ainda outros menos comuns como por reforço.

Aprendizado de máquinas é um conjunto de ferramentas, mas até mais importante do que conhecê-las é saber como e quando usar. Muitas vezes um pequeno ajuste no algoritmo resolveria seu problema e você não perderia tanto tempo e dinheiro!

1.1.3 Aprendizado Supervisionado

O aprendizado supervisionado é o tipo mais comum de aprendizado automático.

Imagine que precise resolver um problema para o qual você tenha acesso a um conjunto de problemas parecidos já resolvidos.

Matematicamente, dada uma entrada x e uma saída desejada y (rótulo), obter uma função $f(x)$ tal que $f(x) = y$. Assim, para um novo dado \hat{x} , poderá calcular a nova saída $\hat{y} = f(\hat{x})$.

EXEMPLOS:

- Imagine o gráfico de preço de uma casa em função da área. Qual seria o preço de uma casa de 100m²?
- Previsão do valor de uma ação com base em medidas de desempenho da companhia e dados econômicos.
- Estimar a quantidade de glicose no sangue de um diabético a partir do espectro de infravermelho.

Esses são exemplos de aprendizado supervisionado. Temos a “resposta correta” para um conjunto de amostras de exemplo. Relação entrada/saída: função!

Mais especificamente, esses são exemplos de **regressão**, já que o valor predito é contínuo (função contínua).

Poderia ajustar uma reta aos dados plotados em um gráfico! No exemplo da casa, o preço poderia ser 150 mil. Mas poderia também ajustar uma função quadrática. E agora o preço seria 200 mil. Vamos aprender como fazer essa escolha.

MAIS EXEMPLOS:

- Prever um tumor como maligno ou benigno a partir de seu tamanho. Mais atributos poderiam ser usados, p.ex., idade.
- Prever se um paciente hospitalizado por um ataque cardíaco terá um segundo ataque a partir de sua dieta e medidas clínicas.
- Identificar um dígito em um CEP manuscrito.

Esses são exemplos de **classificação** porque a saída predita é discreta.

Uma visualização muito comum em classificação é usar símbolos diferentes para cada classe.

Um algoritmo possível seria uma reta (ou hiperplano) separando os grupos.
OBSERVAÇÕES:

- Muito mais atributos poderiam ser usados.
- Note que o problema do preço de casas poderia ser convertido em classificação: preço de venda maior ou menor que determinado valor.

1.2 Aprendizado Não Supervisionado

Sem rótulo para os exemplos. Identificar estrutura nos dados.

EXEMPLOS:

- Algoritmo de agrupamento (*clustering*): Google Notícias - agrupa notícias relacionadas; E muitos outros: Facebook identificando grupos de potenciais conhecidos, segmentação de mercado (propaganda e venda direcionadas), astronomia (formação de galáxias).
- Eliminação/redução de dados redundantes (redução de dimensionalidade)
- Detecção de anomalias: fraude de cartão de crédito, etc.
- *Cocktail Party Problem* - Dois microfones captando dois sons simultaneamente, mas cada microfone captando mais um dos sons. Objetivo: separar os áudios.

1.3 Representação do Modelo

No aprendizado supervisionado, o conjunto de exemplos é chamado de **conjunto de treinamento**.

Tabela 1.1: Notação

m	Número de exemplos no treinamento
x	variáveis de entrada / <i>features</i>
y	Variável de saída / alvo
$(x^{(i)}, y^{(i)})$	i -ésimo exemplo de treinamento
X e Y	Espaço das variáveis de entrada e saída. EX.: $X = Y = \mathbb{R}$.

A Figura 1.2 ilustra o fluxo geral do aprendizado supervisionado. Temos aqui uma função $h(x) : X \rightarrow Y$, chamada de **hipótese** (por razões históricas).

Um exemplo é o ajuste por reta

$$h_{\theta}(x) = \theta_0 + \theta_1 x,$$

cujo modelo é chamado de **regressão linear** com uma variável (univariada). θ_i 's são chamados **parâmetros** do modelo.

NOTAÇÃO: O valor $h_{\theta}(x^{(i)})$ predito pela hipótese é denotado $\hat{y}^{(i)}$.

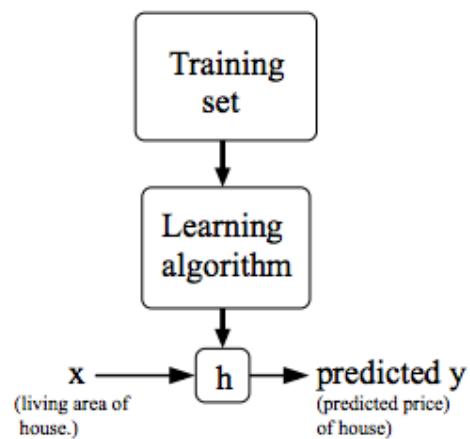


Figura 1.2: Fluxo geral do aprendizado supervisionado.

Capítulo 2

Regressão Linear

2.1 Função de custo

Como escolher os θ_i 's?

Ideia: melhor ajuste - $\theta(x)$ próximo de y no conjunto de treinamento. Uma possibilidade é a média da diferença ao quadrado entre o valor real e o predito para a saída:

$$\underset{\theta_0, \theta_1}{\text{minimize}} \frac{1}{2m} \sum_{i=1}^m (\hat{y}^{(i)} - y^{(i)})^2 = \underset{\theta_0, \theta_1}{\text{minimize}} \frac{1}{2m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2.$$

O fator $\frac{1}{2}$ tem apenas o objetivo de simplificar os cálculos quando esta função for derivada.

Assim, nossa **função de custo** será:

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2,$$

chamada de **função de erro quadrático** ou **erro quadrático médio**. Objetivo:

$$\underset{\theta_0, \theta_1}{\text{minimize}} J(\theta_0, \theta_1)$$

O erro quadrático é a função de custo mais popular em regressão, mas existem outras.

2.2 Intuição da função de custo

Imaginemos um caso simplificado em que $J(\theta) = \theta_1 x$ (reta passando pela origem) e queremos o melhor ajuste possível aos dados de treinamento, que são representados no plano $x - y$ (Figura 2.2). Note que eu consigo uma reta passando exatamente pelos dados ($J(\theta) = 0$) para $\theta_1 = 1$.

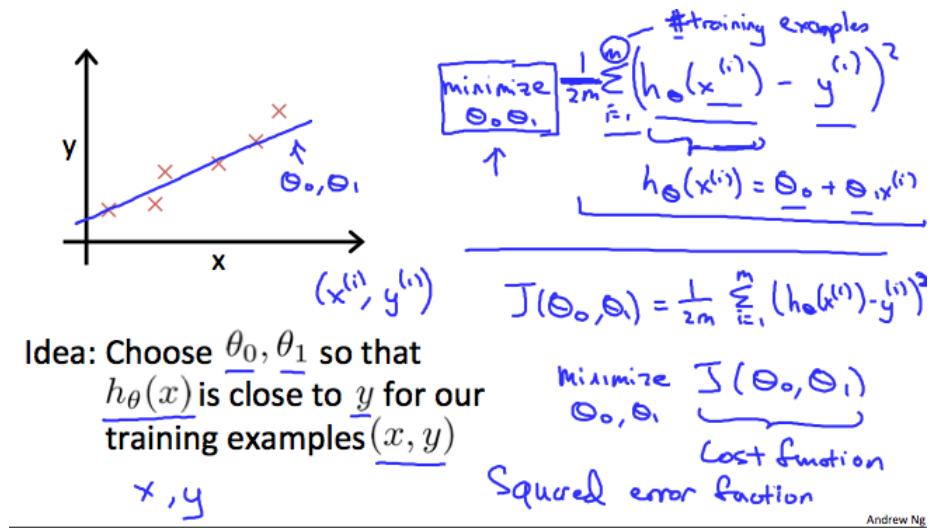
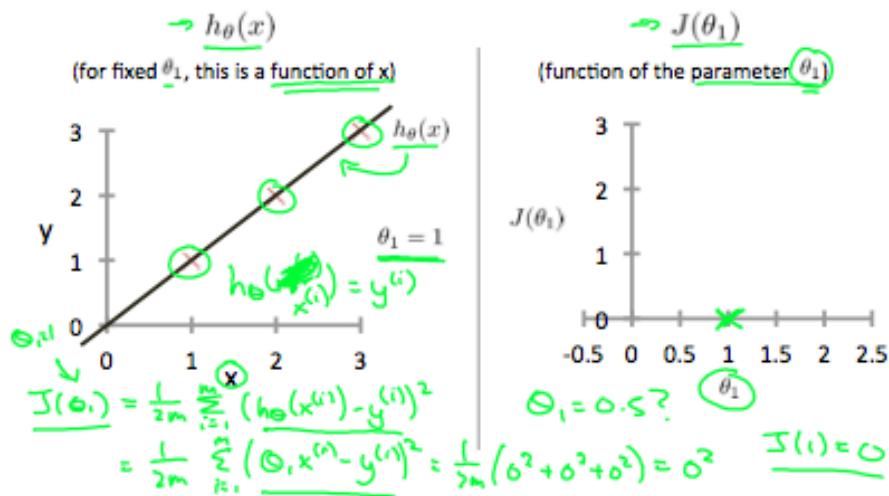
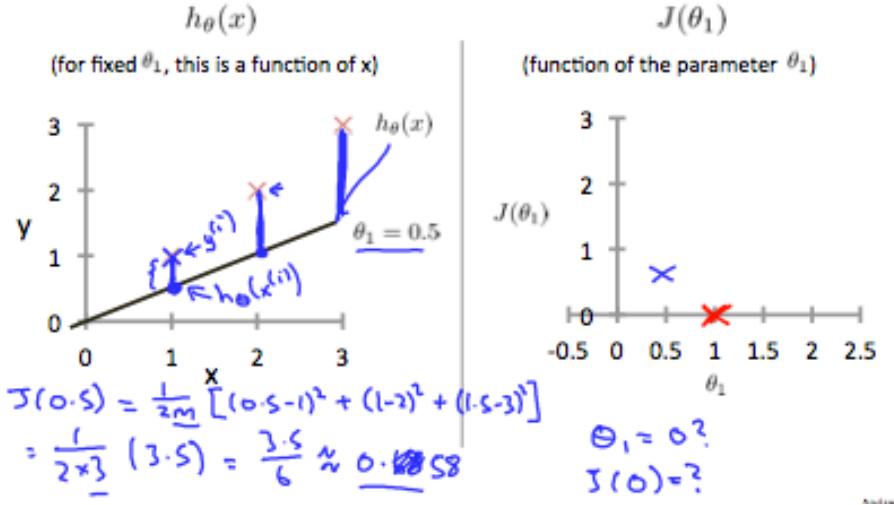


Figura 2.1: Função de custo.

Figura 2.2: Função de custo $J(\theta) = \theta_1 x$.

Figura 2.3: Função de custo $J(\theta) = 0.5x$.

Mas podemos variar θ_1 , p.ex., $\theta_1 = 0.5$ (Figura 2.3).

E ainda outros valores para θ_1 , o que nos leva ao gráfico da Figura 2.4. $\theta_1 = 1$ é o mínimo global de $J(\theta)$ no caso.

Quando os 2 parâmetros θ_0 e θ_1 são considerados, a função $J(\theta_0, \theta_1)$ continua exibindo um formato parabólico, porém agora em 3D. Para facilitar a visualização, usaremos gráficos de contorno, em que cada elipse concêntrica representa valores de (θ_0, θ_1) para os quais $J(\theta_0, \theta_1)$ possui o mesmo valor.

As figuras 2.5 - 2.7 representam valores para (θ_0, θ_1) que vão ficando cada vez mais próximos do mínimo de $J(\theta_0, \theta_1)$, que está no centro.

2.3 Gradiente Descendente

O algoritmo que usaremos para minimizar a função de custo J na regressão linear é o chamado **Gradiente Descendente**.

Imaginemos a função de custo representada como uma “paisagem montanhosa” dependendo dos parâmetros θ_0, θ_1 na Figura 2.8.

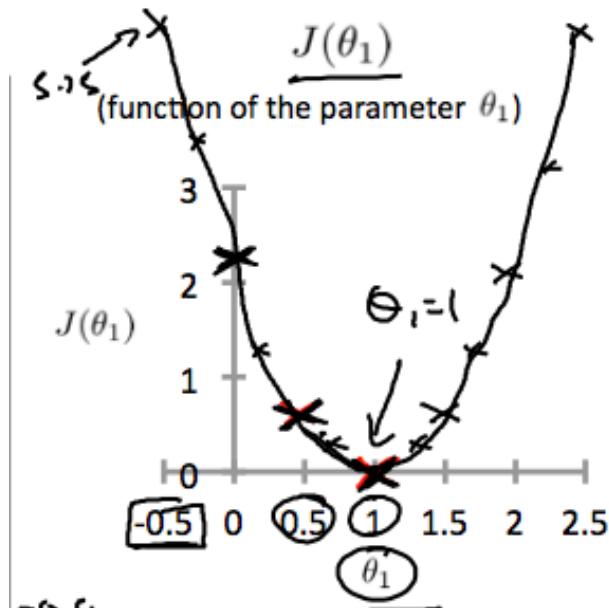
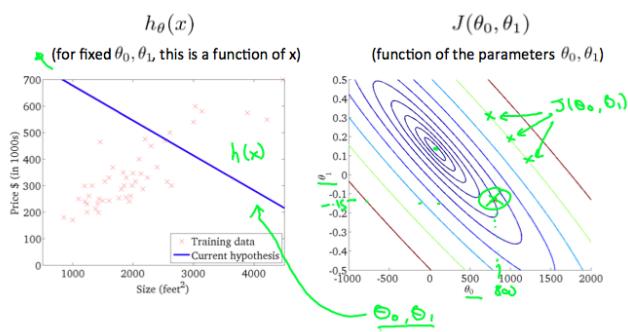
Partindo de um ponto qualquer, estaremos satisfeitos quando chegamos a um “buraco” (mínimo local).

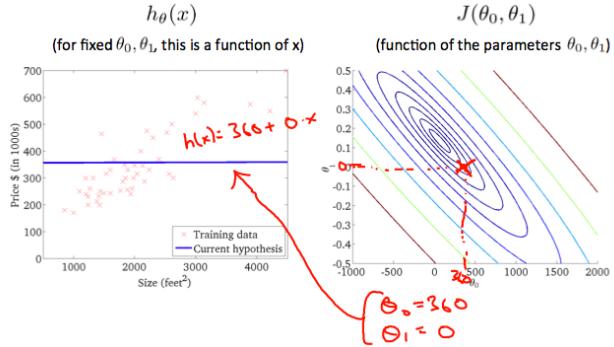
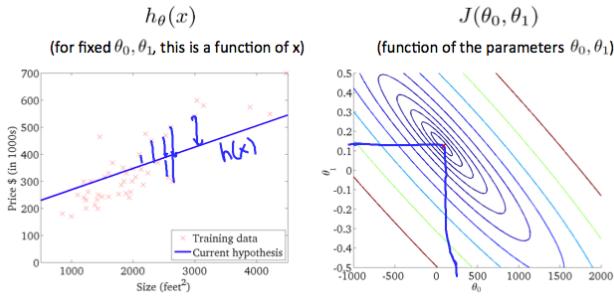
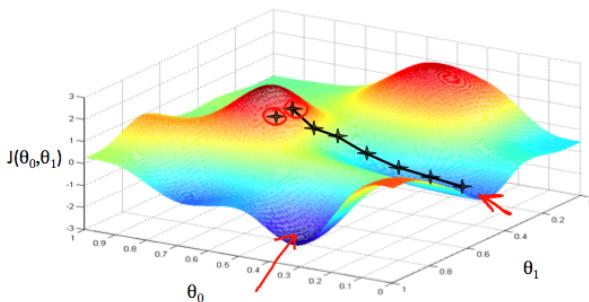
Para isso, olhamos 360 graus em nossa volta e tomamos a direção de descida mais íngreme. O algoritmo para tal é o seguinte.

Repete até convergir:

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1),$$

em que $:=$ simboliza atribuição (em algoritmo), α é o tamanho do passo que

Figura 2.4: Custo $J(\theta) = \theta_1 x$ em função de θ_1 .Figura 2.5: Custo $J(\theta) = \theta_0 + \theta_1 x$ em função de (θ_0, θ_1) .

Figura 2.6: Custo $J(\theta) = \theta_0 + \theta_1x$ em função de (θ_0, θ_1) .Figura 2.7: Custo $J(\theta) = \theta_0 + \theta_1x$ em função de (θ_0, θ_1) .Figura 2.8: Custo $J(\theta_0, \theta_1)$.

Correct: Simultaneous update $\rightarrow \text{temp0} := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$ $\rightarrow \text{temp1} := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$ $\rightarrow \theta_0 := \text{temp0}$ $\rightarrow \theta_1 := \text{temp1}$	Incorrect: ↗ $\rightarrow \text{temp0} := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$ $\rightarrow \theta_0 := \text{temp0}$ $\rightarrow \text{temp1} := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$ ↗ $\rightarrow \theta_1 := \text{temp1}$
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

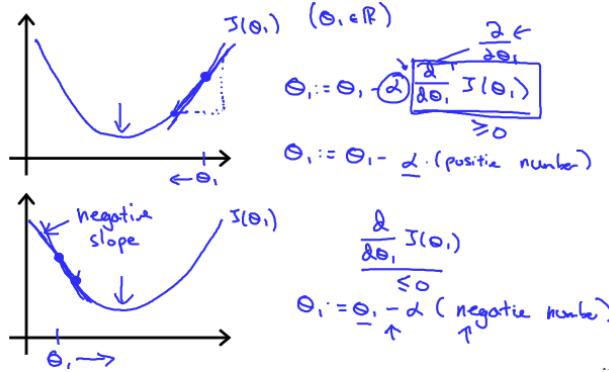
Figura 2.9: Atualização simultânea de θ_0 e θ_1 .

Figura 2.10: Minimização seguindo direção contrária ao coeficiente angular da reta tangente.

daremos e $\frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$ é uma derivada parcial (mais sobre isso em seguida).

Note-se que partindo de 2 pontos próximos (estrelas na Figura 2.8) pode-se chegar a 2 mínimos locais bem distantes entre si.

Outro aspecto importante é que uma implementação correta do gradiente descendente exige que os parâmetros sejam calculados **simultaneamente** (Figura 2.9).

2.4 Intuição do Gradiente Descendente

Trabalhemos com um parâmetro apenas (θ_1) para ganhar intuição:

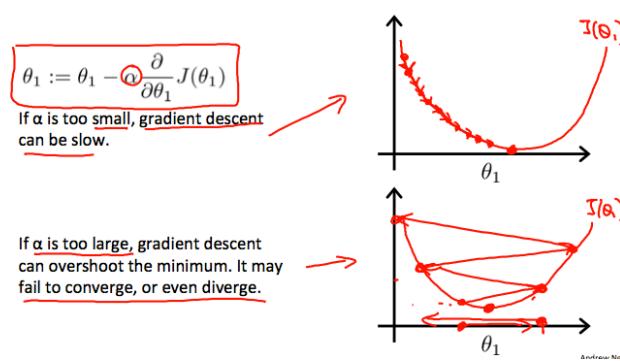
Reita até convergir:

$$\theta_1 := \theta_1 - \alpha \frac{d}{d\theta_1} J(\theta_1).$$

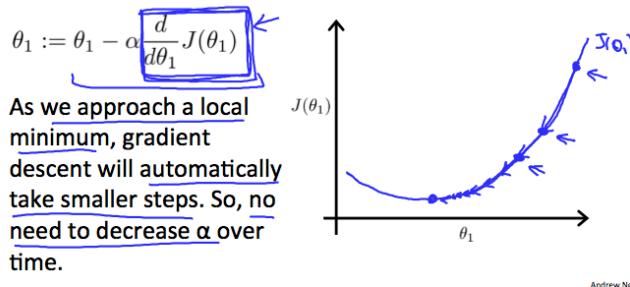
O algoritmo se aproxima do mínimo indo na direção contrária ao sinal da derivada (coeficiente angular da reta tangente) (Figura 2.10). Lembre-se que $\alpha > 0$.

A taxa de aprendizado α controla a convergência. Valor muito pequeno vai demorar muito. Valor muito grande pode perder o mínimo (Figura 2.11).

A convergência de fato ocorre porque o passo vai se tornando cada vez menor à medida que se aproxima do mínimo local (já que a derivada se aproxima de

Figura 2.11: Efeito de α .

Gradient descent can converge to a local minimum, even with the learning rate α fixed.

Figura 2.12: Convergência para α fixado.

zero), até que efetivamente para no mínimo (derivada zero) (Figura 2.12):

$$\theta_1 := \theta_1 - \alpha \cdot 0.$$

2.5 Gradiente Descendente na Regressão Linear

Vamos agora juntar os algoritmos do gradiente descendente com a regressão linear:

Gradiente descendente	Regressão linear
Repita até convergir { $\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$ (para $j = 0$ e $j = 1$) }	$h_\theta(x) = \theta_0 + \theta_1 x$ $J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2$

A parte central agora é calcular as derivadas $\partial/\partial\theta_j$.

Vamos ilustrar o caso em que temos um único par $(x^{(i)}, y^{(i)})$, ou seja, $m = 1$.

Vamos assumir também que $x^{(i)}$ não é mais apenas um único valor, mas sim um vetor \mathbf{x} com dois componentes: $x_0^{(i)}$ fixo e igual a 1 e $x_1^{(i)} = x^{(i)}$.

A função de hipótese da regressão linear para este par $(x^{(i)}, y^{(i)})$ pode ser escrita de forma mais compacta:

$$h_{\theta}(x^{(i)}) = \sum_{j=0}^1 \theta_j x_j^{(i)},$$

Lembre-se que θ_0 “multiplica por 1”!

Podemos agora deduzir uma expressão geral desta derivada que servirá tanto para θ_0 quanto θ_1 .

$$\begin{aligned} \frac{\partial}{\partial\theta_j} J(\theta) &= \frac{\partial}{\partial\theta_j} \frac{1}{2} (h_{\theta}(\mathbf{x}) - y)^2 \\ &= 2 \cdot \frac{1}{2} (h_{\theta}(\mathbf{x}) - y) \cdot \frac{\partial}{\partial\theta_j} (h_{\theta}(\mathbf{x}) - y) && \text{REGRA DA CADEIA} \\ &= (h_{\theta}(\mathbf{x}) - y) \cdot \frac{\partial}{\partial\theta_j} \left(\sum_{j=0}^1 \theta_j x_j - y \right) \\ &= (h_{\theta}(\mathbf{x}) - y) x_j. \end{aligned}$$

A última passagem é consequência de que o único termo do somatório que não é constante em relação à derivada é $\theta_j x_j$.

Assim ficamos com

$$\frac{\partial}{\partial\theta_0} J(\theta) = (h_{\theta}(x^{(i)}) - y^{(i)}),$$

já que $x_0 = 1$, e

$$\frac{\partial}{\partial\theta_1} J(\theta) = (h_{\theta}(x^{(i)}) - y^{(i)}) x^{(i)},$$

já que definimos $x_1 = x^{(i)}$.

Temos finalmente então o algoritmo seguinte:

Repita até convergir: {

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})$$

$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x^{(i)}$$

}

Lembrando sempre que a atualização acima é simultânea.

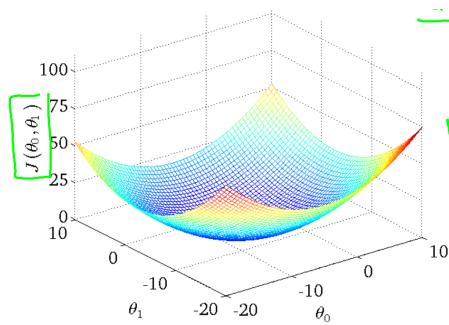


Figura 2.13: Função de custo convexa da regressão linear.

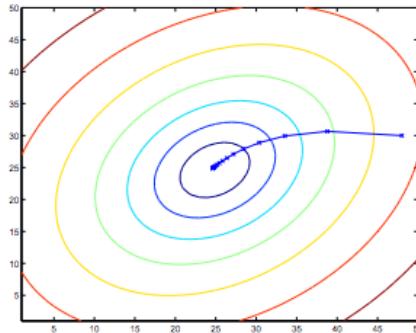


Figura 2.14: Convergência na regressão linear.

Vimos que o gradiente descendente pode ficar preso em mínimos locais, e inclusive em mínimos diferentes dependendo de onde começa.

Porém, especificamente a função de custo $J(\theta)$ da regressão linear tem a forma de uma “bacia lisa”, tendo portanto apenas um único mínimo global (sem outros mínimos locais). Este tipo de função é chamada de **convexa** (Figura 2.13).

Qualquer que seja o chute inicial para θ , o algoritmo vai se aproximar do mínimo em cada passo e, a menos que α seja muito grande, vai sempre convergir para o mínimo global (Figura 2.14).

O algoritmo aqui usou todos os dados de treinamento disponíveis (note a soma de 1 a m). Isso é chamado de **gradiente descendente em batch** (lote). Veremos outras abordagens mais à frente.

Note-se também que no caso da regressão linear, a minimização poderia ser analítica em um passo só (equações normais). Porém o gradiente descendente será útil para nós em muitos outros algoritmos deste tipo.

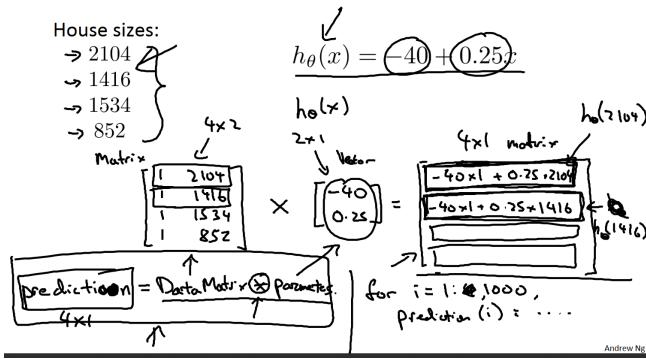


Figura 2.15: Função de hipótese da regressão linear como uma operação vetorial.

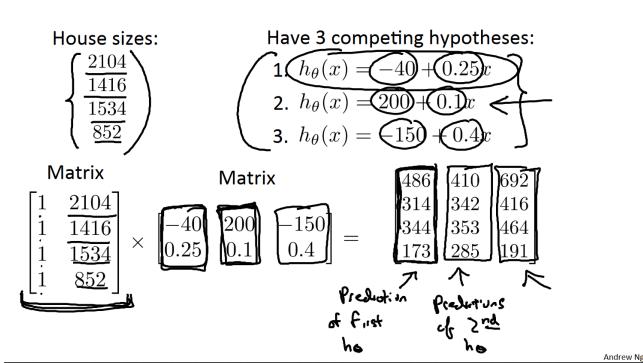


Figura 2.16: Três funções de hipótese da regressão linear como uma operação matricial.

2.6 Vetorização

A função de hipótese da regressão linear pode ser representada convenientemente usando operações entre matriz e vetor, o que além de tornar a notação compacta, torna o algoritmo mais eficiente computacionalmente, já que aproveita melhor cada núcleo de processamento (Figura 2.15).

$$\begin{bmatrix} 1 & x^{(1)} \\ 1 & x^{(2)} \\ 1 & x^{(3)} \\ 1 & x^{(4)} \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix} = \begin{bmatrix} \theta_0 + x^{(1)}\theta_1 \\ \theta_0 + x^{(2)}\theta_1 \\ \theta_0 + x^{(3)}\theta_1 \\ \theta_0 + x^{(4)}\theta_1 \end{bmatrix} = X\theta.$$

E mais ainda se tivermos, por exemplo, 3 funções de hipótese competindo entre si (Figura 2.16).

$$\begin{bmatrix} 1 & x^{(1)} \\ 1 & x^{(2)} \\ 1 & x^{(3)} \\ 1 & x^{(4)} \end{bmatrix} \begin{bmatrix} \theta_0^{(1)} & \theta_0^{(2)} & \theta_0^{(3)} \\ \theta_1^{(1)} & \theta_1^{(2)} & \theta_1^{(3)} \end{bmatrix} = \begin{bmatrix} \theta_0 + x^{(1)}\theta_1 & \theta_0^{(2)} + x^{(1)}\theta_1^{(2)} & \theta_0^{(3)} + x^{(1)}\theta_1^{(3)} \\ \theta_0 + x^{(2)}\theta_1 & \theta_0^{(2)} + x^{(2)}\theta_1^{(2)} & \theta_0^{(3)} + x^{(2)}\theta_1^{(3)} \\ \theta_0 + x^{(3)}\theta_1 & \theta_0^{(2)} + x^{(3)}\theta_1^{(2)} & \theta_0^{(3)} + x^{(3)}\theta_1^{(3)} \\ \theta_0 + x^{(4)}\theta_1 & \theta_0^{(2)} + x^{(4)}\theta_1^{(2)} & \theta_0^{(3)} + x^{(4)}\theta_1^{(3)} \end{bmatrix} = X\Theta.$$

2.7 Regressão Linear Multivariada

Imaginemos agora que tenhamos mais variáveis para descrever o preço de uma casa (Figura 2.17).

x_1	Size (feet ²)	Number of bedrooms	Number of floors	Age of home (years)	y
2104	5	1	45		460
1416	3	2	40		232
1534	3	2	30		315
852	2	1	36		178
...
\tilde{x}_1	\tilde{x}_2	\tilde{x}_3	\tilde{x}_4		
				$n=4$	
					$m=4$
					$\underline{x}^{(2)} = \begin{bmatrix} 1416 \\ 3 \\ 2 \\ 40 \end{bmatrix}$
					$\underline{x}_3^{(2)} = 2$

Figura 2.17: Regressão linear multivariada.

Vamos introduzir mais notação (Tabela 2.1).

Tabela 2.1: Notação na regressão linear multivariada.

$x_j^{(i)}$	=	valor do j -ésimo atributo no i -ésimo exemplo de treinamento
$x^{(i)}$	=	i -ésimo exemplo de treinamento (todos os atributos)
m	=	número de exemplos de treinamento
n	=	número de atributos

Nossa função de hipótese será então:

$$h_{\theta}(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \cdots + \theta_n x_n.$$

Para uma intuição, no caso da Figura 1.2, podemos pensar em θ_0 como o preço-base, θ_1 como o preço por m^2 , θ_2 o preço por cômodo, etc.

Por conveniência, definimos

$$x_0 = 1$$

e assim podemos definir $h_{\theta}(x)$ por uma multiplicação matricial:

$$h_{\theta}(x) = [\theta_0 \quad \theta_1 \cdots \theta_n] \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} = \theta^T x.$$

INTUIÇÃO MATEMÁTICA:

O parâmetro θ_j quantifica a variação em $y = h_\theta(x)$ após uma variação unitária de x_j , mantendo-se os demais atributos fixos.

Justificativa:

$$\frac{\partial}{\partial x_j} h_\theta(x) = \theta_j$$

2.8 Gradiente Descendente na Regressão Linear Multivariada

Podemos pensar na função de custo que depende de h_θ e, consequentemente, de $\theta_0, \theta_1, \dots, \theta_n$ como se dependesse de uma única variável vetorial θ .

Já vimos, por outro lado, na Semana 1, que se fixarmos $x_0 = 1$, a derivada $\partial J / \partial \theta_j$ tem uma expressão geral para todo j :

$$\frac{\partial}{\partial \theta_j} J(\theta) = (h_\theta(x) - y)x_j$$

Assim, a generalização para múltiplas variáveis é direta:

Reita até convergir: {

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}, \quad j = 0, \dots, n$$

}

Lembre-se que isso sempre assumindo $x_0 = 1$.

A Figura 2.18 mostra a conexão entre os algoritmos de regressão linear para uma e múltiplas variáveis.

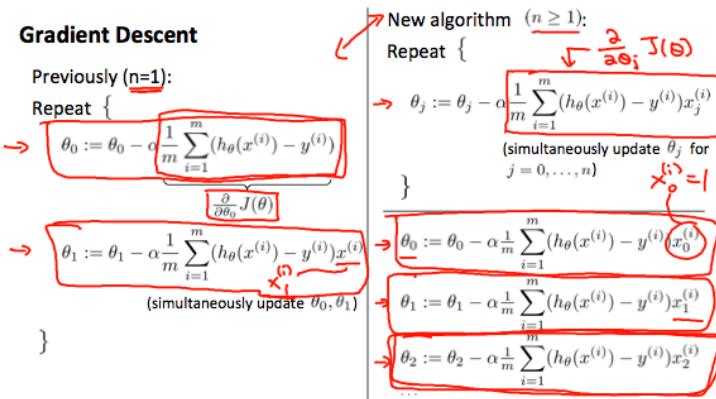


Figura 2.18: Algoritmos da regressão linear de uma variável e multivariada.

2.9 Condições Assumidas

A regressão linear presume algumas características do dado para seu funcionamento ideal:

1. Relação linear entre os atributos e a saída.
2. Pouca ou nenhum multicolinearidade (inter-correlação) entre os atributos.
Plotar matriz de correlação e remover atributos muito correlacionados.
3. Homoscedasticidade: Erro (resíduo) na regressão é o mesmo entre todos os atributos. Fazer *scatter plot* do resíduo *vs* valor predito e checar que não forma nenhum padrão específico (nuvem aleatória de pontos).
4. Distribuição normal dos resíduos: não é necessário quando temos um grande número de amostras (dados moderados a grandes) devido ao Teorema do Limite Central. Checado usando Gráfico QQ (se valores padronizados formarem algo próximo de uma reta é normal).
5. Pouca ou nenhuma auto-correlação entre os resíduos: auto-correlação reduz drasticamente a acurácia do modelo. Comum em séries temporais, em que o dado atual depende dos anteriores. Teste estatístico de Durbin-Watson.

2.10 Aspectos Práticos - Normalização dos Atributos

Atributos em escalas muito diferentes tendem a tornar as linhas de contorno do gradiente em elipses muito alongadas, fazendo o caminho até o mínimo global ser mais longo e tortuoso (Figura 2.19).

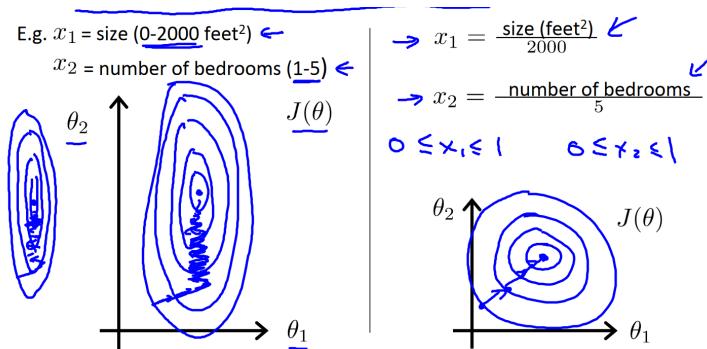


Figura 2.19: Gradiente descendente converge mais lentamente com atributos em diferentes escalas.

Uma propriedade geométrica interessante é que o gradiente sempre aponta na direção perpendicular à linha de contorno. Note como isso gera um comportamento de “zig-zag” no caso da linha elíptica.

Isso se resolve com normalização, por exemplo,

$$-1 \leq x_j \leq 1 \quad \text{ou} \quad -0.5 \leq x_j \leq 0.5.$$

Os valores exatos nos extremos nem fazem diferença no caso, o mais importante é que os atributos não estejam em faixas de valores tão diferentes.

Outra possibilidade de normalização é pela média (padronização). Subtrai-se a média do atributo no conjunto de treinamento, visando deixá-lo com média próxima de zero, e divide-se pela faixa de variação (mínimo – máximo), ou ainda pelo desvio padrão:

$$x_j := \frac{x_j - \mu_j}{s_j},$$

em que μ_j é a média e s_j é $(\max - \min)$ ou desvio padrão.

Por exemplo, se o preço da casa varia entre 100 e 2000, com média 1000, teremos

$$x_j := \frac{\text{preço} - 1000}{1900}.$$

NORMALIZAÇÃO vs PADRONIZAÇÃO

- Normalização: recomendada quando se sabe que o atributo não segue uma distribuição Gaussiana
- Padronização: recomendada quando o dado segue uma distribuição Gaussiana (embora isso não seja obrigatório). Também preserva melhor *outliers* já que o limite de valores não fica fixo.

2.11 Aspectos Práticos - Taxa de Aprendizado

Para checar se o gradiente descendente está funcionando, devemos plotar a curva de $J(\theta)$ por número de iterações. $J(\theta)$ deve diminuir após toda iteração (Figura 2.20).

Pode-se definir um teste automático: o algoritmo converge se $J(\theta)$ diminui menos do que um valor ϵ pequeno em uma iteração, p.ex., $\epsilon = 10^{-3}$.

Dois indícios de que o gradiente não está funcionando são $J(\theta)$ sempre aumentando ou aumentando e diminuindo alternadamente (Figura 2.21). Ambos indicam que se deve usar um valor menor para a taxa de aprendizado α .

Pode-se demonstrar matematicamente que, na regressão linear, $J(\theta)$ diminui após **TODA** iteração, se α for pequeno o suficiente.

Por outro lado, deve-se lembrar que para α muito pequeno a convergência é muito lenta.

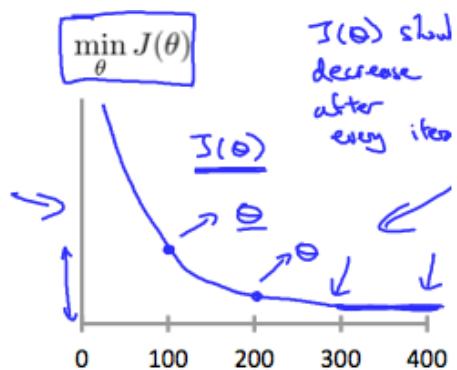


Figura 2.20: Checagem de convergência do gradiente descendente.

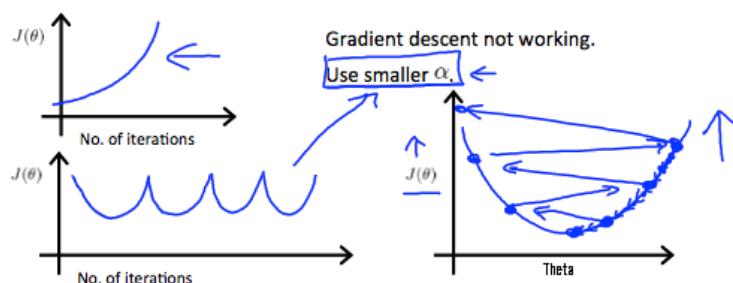


Figura 2.21: Problemas com o gradiente descendente.

Uma boa estratégia é tentar vários α , p.ex.:

$$\alpha = \dots, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, \dots,$$

no caso, algo próximo de valores multiplicados por 3.

2.12 Atributos e Regressão Polinomial

Existem diferentes formas de se melhorar nossa função de hipótese.

Pode-se, por exemplo, combinar atributos. EX.: Na previsão de preço de uma casa, em vez de usar x_1 (largura da frente) e x_2 (profundidade), criar um atributo área $x_3 = x_1 \cdot x_2$.

Outra possibilidade é uma função de hipótese não linear (quadrática, cúbica, raiz quadrada, ou qualquer outra). EX.:

$$h_{\theta}(x) = \theta_0 + \theta_1 x_1^2$$

$$h_{\theta}(x) = \theta_0 + \theta_1 x_1^2 + \theta_2 x_1^3$$

$$h_{\theta}(x) = \theta_0 + \theta_1 x_1 + \theta_2 \sqrt{x_1}.$$

Observar a distribuição dos dados (Figura 2.22). Note que aqui os pontos não parecem colineares. Poderia tentar uma função quadrática. Mas uma parábola decairia em algum momento, o que não se espera do preço de imóveis. Poderia então usar uma cúbica. Ou raiz quadrada, cuja curva se ajustaria bem aos pontos.

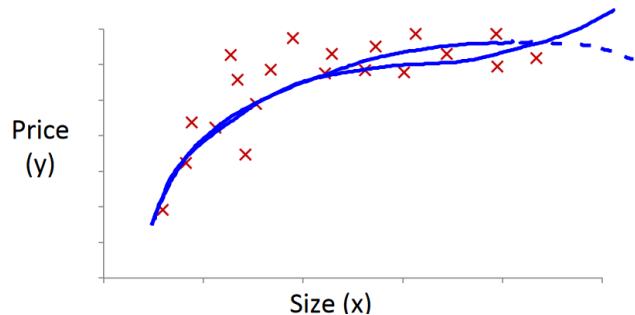


Figura 2.22: Ajuste não-linear.

O algoritmo na prática é o mesmo da regressão linear. Suponha que tenhamos uma hipótese

$$h_{\theta}(x) = \theta_0 + \theta_1 x_1^2 + \theta_2 x_1^3.$$

Basta definir novas variáveis $x_2 = x_1^2$ e $x_3 = x_1^3$ e fazer como antes.

IMPORTANTE: A normalização de atributos é ainda mais crucial neste caso. EX.: Se x_1 varia entre 1 e 1000, x_1^2 vai variar entre 1 e 1000000 e x_1^3 entre 1 e 1000000000!!!

2.13. CÁLCULO ANALÍTICO DOS PARÂMETROS - EQUAÇÃO NORMAL31

2.13 Cálculo Analítico dos Parâmetros - Equação Normal

Uma alternativa ao gradiente descendente na regressão linear é usando a **Equação Normal**.

Este é um método não iterativo, baseado em fazer $\partial J / \partial \theta = 0$. Assim temos a equação normal

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

A matriz X , chamada de matriz de *design*, é montada com uma coluna de 1's (múltiplo de θ_0) e cada atributo nas demais colunas (Figura 2.23).

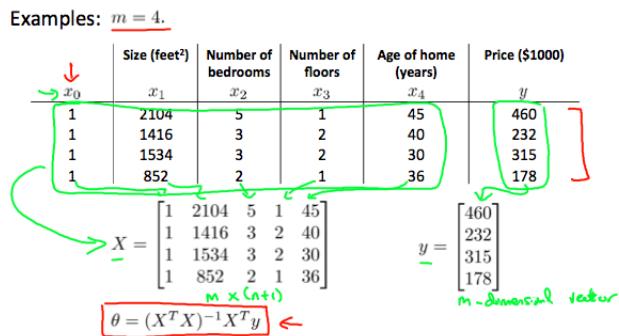


Figura 2.23: Equação normal.

Com a equação normal NÃO é necessário fazer normalização de atributos. Vantagens e desvantagens de cada abordagem na Tabela 2.2. É comum que

Tabela 2.2: Gradiente descendente vs. equação normal.

Gradiente Descende	Equação Normal
Precisa escolher α	Não precisa escolher α
Mais iterações	Sem iterações
$\mathcal{O}(kn^2)$	$\mathcal{O}(n^3)$ devido à inversa de $X^T X$
Funciona bem com n grande	Lento se n for grande

se use gradiente descendente para $n > 10000$ e equação normal para n menor.

O Código 2.1 mostra um exemplo de código para a equação normal em Python.

Listing 2.1: Código em Python para a equação normal.

```
import numpy as np

X = np.array([[1, 2104, 5, 1, 45], [1, 1416, 3, 2, 40], [1, 1534, 3, 2, 30], [1, 852, 2, 1, 36]])
```

```
y = np.array([460, 232, 315, 178])
y = y.transpose()
np.linalg.inv(X.transpose().dot(X)).dot(X.transpose()).dot(y)
```

2.13.1 Derivando a equação normal

Função de hipótese:

$$h_{\theta}(x) = \theta_0 x_0 + \theta_1 x_1 + \cdots + \theta_n x_n.$$

Função de custo:

$$J(\theta_0, \dots, \theta_n) = \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2. \quad (2.1)$$

Na prática, temos um sistema de equações (uma para cada amostra de treinamento) a minimizar, é conveniente então a notação matricial. Definimos então os vetores θ , \mathbf{x} e \mathbf{y} :

$$\theta = \begin{pmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{pmatrix} \quad \mathbf{x} = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{pmatrix} \quad \mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix}$$

em que $\theta_0 = 1$, por conveniência. Assim, para cada $\mathbf{x}^{(i)}$ (i -ésimo exemplo de treinamento) podemos escrever

$$h_{\theta}(\mathbf{x}^{(i)}) = (\mathbf{x}^{(i)})^T \theta.$$

Mas podemos ir mais além e fazer o mesmo para o treinamento todo, usando a matriz de *design* X .

$$X = \begin{bmatrix} (x^{(1)})^T \\ (x^{(2)})^T \\ \vdots \\ (x^{(m)})^T \end{bmatrix}$$

O vetor com $h_{\theta}(\mathbf{x}^{(i)})$ para cada exemplo i é obtido simplesmente fazendo $X\theta$. A expressão dentro do quadrado em (2.1) é vetorizada então como $X\theta - \mathbf{y}$.

Lembramos agora que a soma dos quadrados dos elementos de um vetor-coluna \mathbf{x} é obtida de $\mathbf{x}^T \mathbf{x}$. Isso nos permite reescrever (2.1) de forma compacta e elegante:

$$J(\theta) = \frac{1}{2m} (X\theta - \mathbf{y})^T (X\theta - \mathbf{y}).$$

Podemos descartar o termo $\frac{1}{2m}$ pois constante não faz diferença na minimização de $J(\theta)$. Vamos usar também o fato de que $(A + B)^T = A^T + B^T$ e a distributividade das matrizes:

$$J(\theta) = ((X\theta)^T - \mathbf{y}^T)(X\theta - \mathbf{y})$$

$$J(\theta) = (X\theta)^T X\theta - (X\theta)^T \mathbf{y} - \mathbf{y}^T (X\theta) + \mathbf{y}^T \mathbf{y}$$

Agora, usaremos duas propriedades conhecidas de matrizes e vetores. Na 1^a parcela acima, usamos a propriedade $(AB)^T = B^T A^T$. Já para juntar a 2^a e 3^a parcelas, usamos o fato de que $X\theta$ e \mathbf{y} são vetores-coluna e a propriedade $\mathbf{a}^T \mathbf{b} = \mathbf{b}^T \mathbf{a}$ para dois vetores-coluna \mathbf{a} e \mathbf{b} quaisquer:

$$\begin{aligned} J(\theta) &= \theta^T X^T X\theta - 2\mathbf{y}^T (X\theta) + \mathbf{y}^T \mathbf{y} \\ &= \theta^T X^T X\theta - 2(X^T \mathbf{y})^T \theta + \mathbf{y}^T \mathbf{y} \end{aligned}$$

Agora precisamos derivar $J(\theta)$ em relação a θ e igualar a zero. Usamos então duas propriedades do cálculo diferencial matricial. A primeira é que $\frac{d}{d\mathbf{x}} \mathbf{x}^T A \mathbf{x} = 2A\mathbf{x}$ para qualquer matriz A simétrica. A segunda é que $\frac{d}{d\mathbf{x}} \mathbf{b}^T \mathbf{x} = \mathbf{b}$, para dois vetores-coluna \mathbf{b} e \mathbf{x} quaisquer. Sugere-se consultar a seção 4.3 do material “Linear Algebra Review and Reference” anexo. Assim temos:

$$\frac{\partial J(\theta)}{\partial \theta} = 2X^T X\theta - 2X^T \mathbf{y} = 0.$$

ou ainda

$$X^T X\theta = X^T \mathbf{y}.$$

Finalmente, assumindo que $X^T X$ é invertível:

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

2.14 E se $X^T X$ não for Inversível?

$X^T X$ pode não ser inversível (singular/degenerada).

A função *pinv* (pseudoinversa) sempre vai obter θ ótimo (ao contrário de *inv*).

Principais causas para $X^T X$ não inversível:

- Atributos redundantes (linearmente dependentes), p.ex., área em m^2 e pé 2
- atributos em excesso, p.ex., $m \leq n$ - remover atributos ou regularização (falaremos mais tarde)

Capítulo 3

Regressão Logística

3.1 Classificação

Saída pertence a um pequeno conjunto de valores discretos.

EX.:

- Email é um spam ou não
- Transação fraudulenta ou não
- Tumor benigno ou maligno

Esses são problemas de 2 classes (classificação binária): 0 ou 1, classe “positiva” ou “negativa”.

Podemos usar a regressão linear $h_\theta(x) = \theta^T x$, definindo, por exemplo, que se $h_\theta(x) \geq 0.5$ a classe é positiva, senão é negativa.

Porém temos 2 problemas:

1. Uma amostra de treinamento a mais pode mudar a reta de $h_\theta(x)$ e comprometer o classificador (Figura 3.1). Classificação não é um processo linear.
2. $h_\theta(x)$ na regressão linear pode ter valores < 0 ou > 1 .

3.2 Regressão Logística - Função de Hipótese

Esses problemas da regressão linear são resolvidos pela regressão logística, cuja função de hipótese satisfaz $0 \leq h_\theta(x) \leq 1$.

Temos agora

$$h_\theta(x) = g(\theta^T x),$$

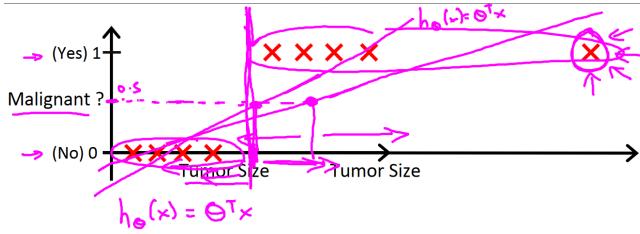


Figura 3.1: Regressão linear para classificação.

em que $g(z)$ é a chamada função sigmoide ou logística:

$$g(z) = \frac{1}{1 + e^{-z}},$$

ou seja:

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}.$$

A Figura 3.2 mostra a função sigmoide (note como os valores estão entre 0 e 1).

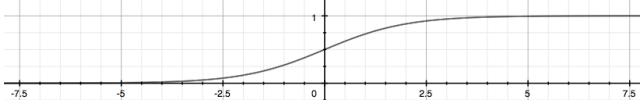


Figura 3.2: Síntese.

$h_\theta(x)$ no caso é interpretado como a probabilidade de a saída correspondente a x ser 1. EX.:

$$x = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} 1 \\ \text{tamanho do tumor} \end{bmatrix}$$

e

$$h_\theta(x) = 0.7$$

significa que a probabilidade de o tumor ser maligno ($y = 1$) é 70%.

Note que como a classe só pode ser 0 ou 1, a probabilidade de $y = 0$ é complementar:

$$P(y = 0|x; \theta) = 1 - P(y = 1|x; \theta),$$

em que a expressão acima se lê como “probabilidade de $y = 1$ (ou $y = 0$), dado x , parametrizado por θ ”.

3.3 Fronteira de Decisão

Como as classes só podem ser 0 ou 1, o classificador precisa definir um limiar, p.ex., $y = 1$ se $h_\theta(x) \geq 0.5$ e $y = 0$ se $h_\theta(x) < 0.5$.

Mas na Figura 2.1 vemos que $g(z) \geq 0.5$ se $z \geq 0$.

Temos portanto $y = 1$ se $\theta^T x \geq 0$ e $y = 0$ se $\theta^T x < 0$.

Agora vamos supor que

$$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$$

e ainda que o vetor θ é

$$\theta = \begin{bmatrix} -3 \\ 1 \\ 1 \end{bmatrix},$$

de modo que $y = 1$ se $-3 + x_1 + x_2 \geq 0$.

A região de $y = 1$ é separada de $y = 0$ pela reta $x_1 + x_2 = 3$ (Figura 3.3).

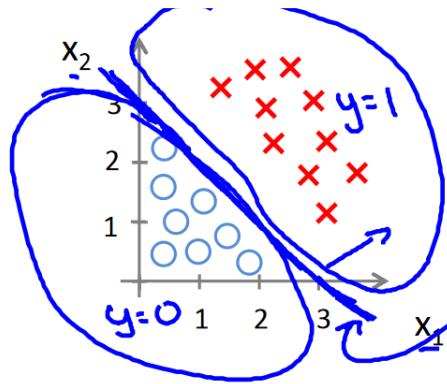


Figura 3.3: Fronteira de decisão.

Ou ainda:

$$\theta = \begin{bmatrix} 5 \\ -1 \\ 0 \end{bmatrix},$$

de modo que $y = 1$ se $5 - x_1 \geq 0$ e a fronteira seria a reta vertical $x_1 = 5$.

Agora note que poderíamos ter fronteiras de separação bem mais complexas (não lineares) no treinamento. Veja o exemplo da Figura 3.4, em que a fronteira é um círculo. Neste caso, a função de hipótese também precisa ser mais complexa, p.ex.

$$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2),$$

com

$$\theta = \begin{bmatrix} -1 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix},$$

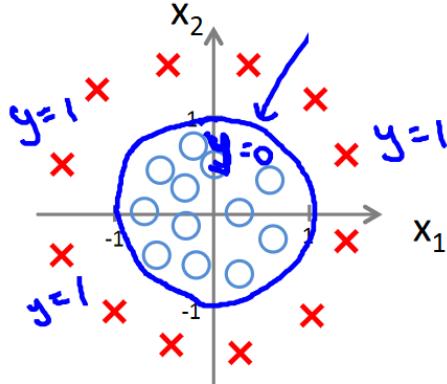


Figura 3.4: Fronteira de decisão não linear.

de modo que $y = 1$ se $-1 + x_1^2 + x_2^2 \geq 1$.

A fronteira então seria $x_1^2 + x_2^2 = 1$, que é a circunferência de raio 1 e separaria nossos pontos.

Funções ainda mais complicadas poderiam ser usadas para fronteiras mais complexas, p.ex.:

$$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_1^2 x_2 + \theta_5 x_1^2 x_2^2 + \theta_6 x_1^3 x_2 + \dots)$$

3.4 Função de Custo

Temos m exemplos de treinamento:

$$\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})\},$$

sendo cada vetor x definido por

$$x = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix},$$

como de praxe $x_0 = 1$. Além disso:

$$y \in \{0, 1\}$$

e

$$h_\theta(x) = \frac{1}{1 + e^{\theta^T x}}.$$

Como encontrar θ ótimo?

Lembrando que na regressão linear nós tínhamos¹

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m \frac{1}{2} (h_\theta(x^{(i)}) - y^{(i)})^2$$

Vamos agora chamar essa expressão dentro do somatório de função **Custo** (podemos também remover o $^{(i)}$ para simplificar):

$$\text{Custo}(h_\theta(x), y) = \frac{1}{2} (h_\theta(x) - y)^2.$$

Na regressão logística teríamos então:

$$\text{Custo}(h_\theta(x), y) = \frac{1}{2} \left(\frac{1}{1 + e^{-\theta^T x}} - y \right)^2.$$

Porém, a função $J(\theta)$ neste caso é **não-convexa**, ou seja, tem ondulações que fazem com que tenha vários mínimos locais, causando problemas para o gradiente descendente (Figura 3.5).

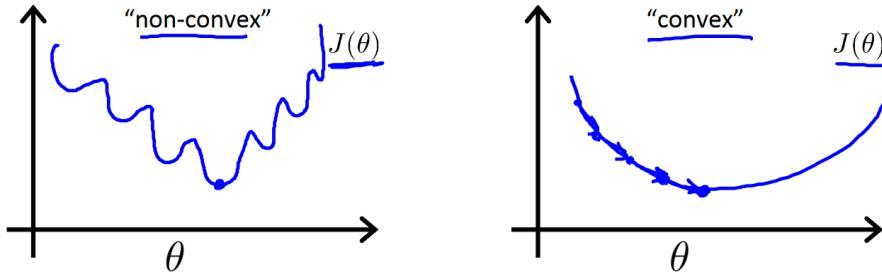


Figura 3.5: Funções de custo não-convexas e convexas.

A solução para isso é definirmos uma nova função *Custo*:

$$\text{Custo}(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & \text{se } y = 1 \\ -\log(1 - h_\theta(x)) & \text{se } y = 0. \end{cases} \quad (3.1)$$

Veja o gráfico na Figura 3.6. Note que:

- Se $h_\theta(x) = y$, $\text{Custo}(h_\theta(x), y) = 0$, para $y = 1$ ou $y = 0$.
- Se $y = 0$ e $h_\theta(x) \rightarrow 1$ (falso positivo), $\text{Custo}(h_\theta(x), y) \rightarrow \infty$
- Se $y = 1$ e $h_\theta(x) \rightarrow 0$ (falso negativo), $\text{Custo}(h_\theta(x), y) \rightarrow \infty$

Isto significa que essa função de custo captura a intuição de que a situação em que $h_\theta(x) = 0$ ($P(y = 1|x; \theta) = 0$), mas $y = 1$, deve ser penalizada com um custo muito alto.

Note que agora temos a garantia de que $J(\theta)$ é convexa.

¹Apenas com um pequeno ajuste, levando $\frac{1}{2}$ para dentro do somatório.

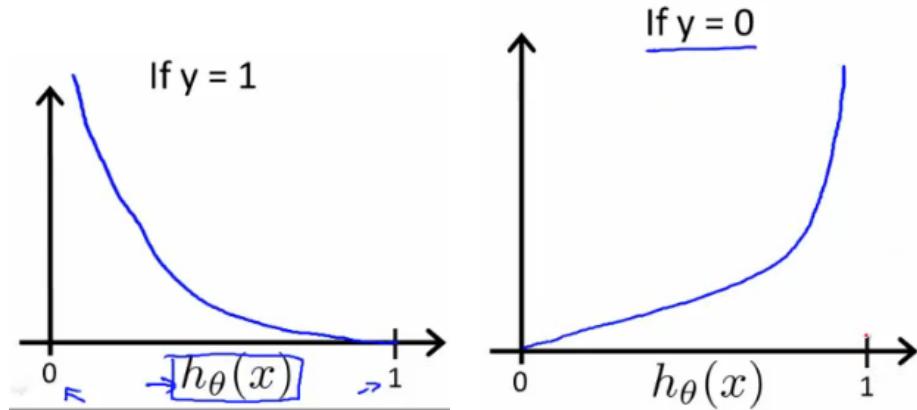


Figura 3.6: Função de custo da regressão logística.

3.5 Versão Compacta da Função de Custo e Gradiante Descendente

Para o uso do gradiente descendente, é conveniente que nossa função de custo seja definida em uma linha. E isso é possível, dado que SEMPRE temos $y = 0$ ou $y = 1$, definindo-se

$$Custo(h_\theta(x), y) = -y \log(h_\theta(x)) - (1 - y) \log(1 - h_\theta(x)).$$

Note que essa é equivalente a (3.1) tanto para $y = 0$ quanto $y = 1$.

Deste modo temos:

$$J(\theta) = \frac{1}{m} \left[y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})) \right].$$

Essa expressão pode ser obtida por uma técnica chamada de “máxima verossimilhança”.

Para o gradiente descendente, lembramos de sua forma geral:

Reita até convergir: {

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

} (atualização simultânea para todo θ_j)

Calculando-se a derivada de $J(\theta)$ usando Cálculo temos

Reita até convergir: {

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

} (atualização simultânea para todo θ_j)

Note que o algoritmo parece igual ao da regressão linear, mas há aqui uma diferença crucial, que é a definição de $h_\theta(x)$:

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}.$$

Por fim, este algoritmo para vários valores de j pode ser vetorizado:

$$\theta := \theta - \frac{\alpha}{m} X^T (g(X\theta) - \mathbf{y}).$$

IMPORTANTE: A normalização dos atributos em escala também faz com que o gradiente descendente encontre o mínimo mais rapidamente na regressão logística.

3.6 Otimização Avançada

Se programarmos um código para calcular $J(\theta)$ e $\frac{\partial}{\partial \theta_j} J(\theta)$, para $j = 0, 1, \dots, n$, então podemos usar outros algoritmos para minimizar $J(\theta)$ em vez do gradiente descendente.

EX.:

- Gradiente conjugado
- BFGS
- L-BFGS

Vantagens	Desvantagens
Não precisa escolher α manualmente	Mais complexo
Normalmente mais rápido do que o gradiente descendente	

Vamos supor que temos

$$\begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix}$$

e a função de custo

$$J(\theta) = (\theta_1 - 5)^2 + (\theta_2 - 5)^2,$$

cujo mínimo sabemos ser $\theta_1 = 5, \theta_2 = 5$.

O Código 3.1 mostra a minimização dessa função em Python usando BFGS.

Listing 3.1: Código em Python para a minimização de $J(\theta) = (\theta_1 - 5)^2 + (\theta_2 - 5)^2$.

```
from scipy.optimize import fmin_bfgs
import numpy as np

def costFunction(theta):
    return (theta[0]-5)**2.0 + (theta[1]-5)**2.0

def costFunctionGradient(theta):
    return np.asarray([2*(theta[0]-5), 2*(theta[1]-5)])
```

```
x0 = np.asarray([0,0])
xopt = fmin_bfgs(costFunction, x0, fprime=costFunctionGradient)
print(xopt)
```

Note que um algoritmo deste tipo pode ser facilmente adaptado na regressão logística (ou linear), apenas lembrando que

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

3.7 Classificação multi-classes: *One-vs-all*

Problemas multi-classes são frequentes. EX.:

- Separação de emails em pastas: Trabalho, Amigos, Família, Lazer
- Exame médico: Saudável, Resfriado, Gripe
- Clima: Ensolarado, Nublado, Chuva, Neve

As classes são comumente numeradas: $y = 1, 2, 3, \dots$ ou $y = 0, 1, 2, \dots$. Os números em si não importam!

Podem ser transformados em um problema de classificação binária, por exemplo, usando a estratégia “one-vs-all” (ou “one-vs-rest”), em que para k classes vamos ter k classificadores (p.ex. regressão logística).

No i -ésimo classificador $h_\theta^{(i)}(x)$, a i -ésima classe é “positiva” e todas as demais são “negativas” (Figura 3.7).

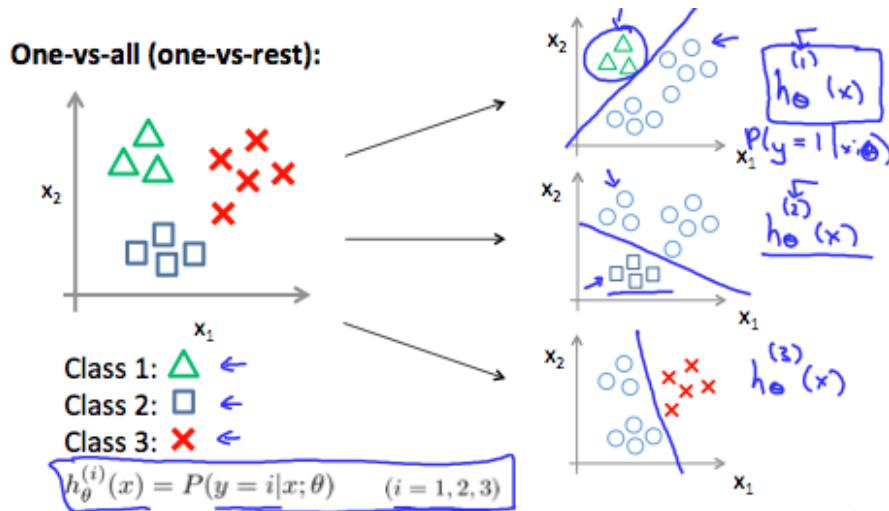


Figura 3.7: Classificação multi-classes.

Portanto, para cada classe i , o classificador $h_{\theta}^{(i)}(x)$ prediz a probabilidade de $y = i$.

Finalmente, dada uma nova entrada x , a classe predita será aquela que maximiza

$$\max_i h_{\theta}^{(i)}(x).$$

3.8 Dedução da Função de Custo

Como vimos da definição da função de hipótese:

$$\begin{aligned} P(y=1|x; \theta) &= h_{\theta}(x) \\ P(y=0|x; \theta) &= 1 - h_{\theta}(x) \end{aligned}$$

Podemos escrever de forma mais compacta:

$$p(y|x; \theta) = (h_{\theta}(x))^y (1 - h_{\theta}(x))^{1-y}.$$

Vamos passar a analisar essa probabilidade agora explicitamente em função de θ . A função nesse caso se chama **verossimilhança** (*likelihood*):

$$L(\theta) = L(\theta; X, y) = p(y|X; \theta).$$

O conceito de verossimilhança é muito importante em aprendizado de máquinas e aparece com muita frequência em deduções mais teóricas.

Se tivermos agora m exemplos de treinamento, todos gerados de modo independente, as probabilidades são multiplicadas:

$$L(\theta) = \prod_{i=1}^m p(y^{(i)}|x^{(i)}; \theta) = \prod_{i=1}^m (h_{\theta}(x^{(i)}))^{y^{(i)}} (1 - h_{\theta}(x^{(i)}))^{1-y^{(i)}}$$

Um dos fundamentos mais importantes do aprendizado de máquinas é o **princípio da máxima verossimilhança**, que nos diz que devemos escolher parâmetros θ que maximizem $L(\theta)$.

Porém essa maximização fica muito mais fácil se tomarmos $\log L(\theta)$. Essa é a log verossimilhança (*log likelihood*). E é exatamente essa função que dá origem ao nosso custo:

$$J(\theta) = \frac{1}{m} \log L(\theta) = \frac{1}{m} \sum_{i=1}^m [y^{(i)} \log h_{\theta}(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_{\theta}(x^{(i)}))].$$

3.9 Dedução do Gradiente

Antes de qualquer coisa, vamos mostrar que a função logística satisfaz uma propriedade em sua derivada que nos será muito útil:

$$\begin{aligned}
 g'(z) &= \frac{d}{dz} \frac{1}{1+e^{-z}} \\
 &= \left(-\frac{1}{(1+e^{-z})^2} \right) (-e^{-z}) \\
 &= \frac{e^{-z}}{(1+e^{-z})^2} \\
 &= \frac{1}{1+e^{-z}} \cdot \frac{e^{-z}}{1+e^{-z}} \\
 &= \frac{1}{1+e^{-z}} \cdot \frac{(1+e^{-z})-1}{1+e^{-z}} \\
 &= \frac{1}{1+e^{-z}} \cdot \left(1 - \frac{1}{1+e^{-z}} \right) \\
 &= g(z)(1-g(z)).
 \end{aligned}$$

Devemos lembrar também que

$$h_\theta(x) = g(\theta^T x).$$

Vamos omitir os índices sobreescritos só para simplificar a notação.

Agora podemos calcular o gradiente:

$$\begin{aligned}
 \frac{\partial}{\partial \theta_j} J(\theta) &= \left(y \frac{1}{g(\theta^T x)} - (1-y) \frac{1}{1-g(\theta^T x)} \right) \frac{\partial}{\partial \theta_j} g(\theta^T x) \\
 &= \left(y \frac{1}{g(\theta^T x)} - (1-y) \frac{1}{1-g(\theta^T x)} \right) g(\theta^T x)(1-g(\theta^T x)) \frac{\partial}{\partial \theta_j} \theta^T x \\
 &= (y(1-g(\theta^T x)) - (1-y)g(\theta^T x))x_j \\
 &= (y(1-h_\theta(x)) - (1-y)h_\theta(x))x_j \\
 &= (y - h_\theta(x))x_j.
 \end{aligned}$$

ATENÇÃO: No nosso algoritmo há uma inversão de sinais para preservar a homogeneidade com o algoritmo da regressão linear. A derivada acima tem o sinal trocado, ficando $(h_\theta(x) - y)x_j$, mas isso é compensado por uma inversão também no sinal do α no gradiente descendente. Como o que desejamos na verdade é MAXIMIZAR $J(\theta)$, aquele sinal deveria ser POSITIVO, mas lá aparece negativo.

Capítulo 4

Regularização

4.1 *Overfitting*

Retomemos nosso modelo de previsão do preço de uma casa em função do tamanho (Figura 4.1).

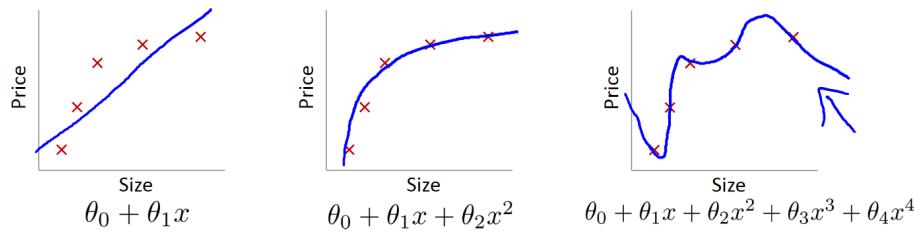


Figura 4.1: *Underfitting* e *overfitting* na regressão linear.

Se usarmos um modelo linear (esquerda), não vai ajustar bem. A função é demasiadamente simples para capturar a estrutura dos dados. Temos um ***underfitting*** ou **viés alto**. Ou seja, o modelo tem uma pré-disposição (viés) a assumir linearidade, que não é correta no caso.

O modelo quadrático (meio) parece o mais adequado.

Mas será que melhora se adicionarmos mais atributos (e parâmetros consequentemente). À direita temos um polinônio de grau 4. Ele se ajusta perfeitamente aos dados de treinamento. Mas varia muito (**alta variância**) e assim não deve generalizar bem para dados novos. Isso é um ***overfitting***. Repare ainda que não temos exemplos suficientes no treinamento para se ajustar bem a um modelo tão complexo.

Assim temos:

Definição: *Overfitting* ocorre quando temos muitos atributos, a função de hipótese se ajusta muito bem ao treinamento, i.e., $J(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2 \approx 0$, mas não generaliza bem, i.e., no exemplo não prevê bem o preço de novos imóveis.

Na classificação com regressão logística ocorre algo parecido (Figura 4.2).

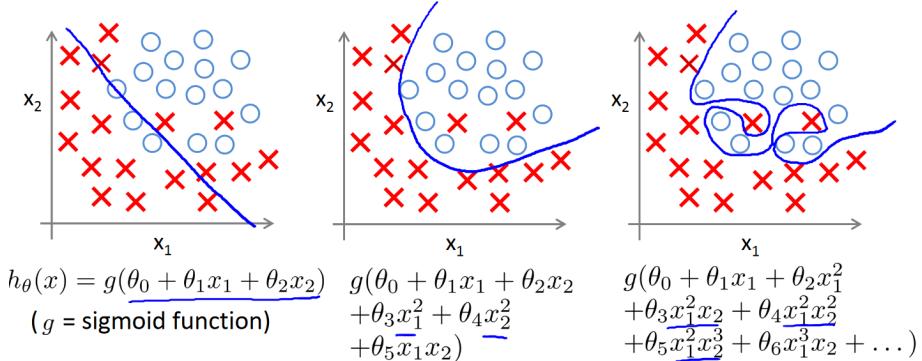


Figura 4.2: *Underfitting* e *overfitting* na regressão logística.

Novamente temos *underfitting* à esquerda e *overfitting* à direita.

O *overfitting* surge na presença de muitos atributos e poucos exemplos de treinamento.

As opções para tratar o *overfitting* são basicamente as seguintes:

1. Reduzir o número de atributos
 - Manualmente
 - Algoritmo de seleção de atributos (veremos mais à frente)
2. Regularização
 - Mantém todos os atributos, mas reduz a magnitude (peso) dos parâmetros θ_j
 - Funciona bem quando temos muitos atributos e todos eles contribuem com algo (mesmo que pouco)

4.2 Função de Custo Regularizada

Vimos que o polinômio de ordem 4

$$\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$$

causa *overfitting*. Mas e se quiséssemos trazer esse polinômio para algo próximo de uma função quadrática, que funcionou melhor, sem exatamente eliminar θ_3 e θ_4 ?

Podemos penalizar valores altos de θ_3 e θ_4 com uma nova função de custo:

$$\underset{\theta}{\text{minimize}} \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2 + 1000\theta_3^2 + 1000\theta_4^2.$$

Note que o multiplicador 1000 aí é só um exemplo de número grande.

A minimização agora fará θ_3 e θ_4 ficarem próximos de zero.

Podemos estender o raciocínio para todos os parâmetros, já que muitas vezes não temos como saber qual(is) parâmetros influenciam menos *a priori*.

Em geral, valores menores para os parâmetros $\theta_0, \theta_1, \dots, \theta_n$ geram:

- Função de hipótese “mais simples”, suavizando a função original
- Menor tendência a *overfitting*

No caso geral, temos a função de custo regularizada:

$$J(\theta) = \frac{1}{2m} \left[\sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^n \theta_j^2 \right]$$

$$\underset{\theta}{\text{minimize}} J(\theta)$$

A segunda parcela de $J(\theta)$ é o termo regularizador e λ é o parâmetro de regularização. Por convenção, θ_0 não aparece no termo regularizador, mas isso não faz diferença significativa.

A razão para isso é que θ_0 controla o ponto em torno do qual os valores preditos variam e forçar uma redução artificial nesse valor não faz sentido. A adição de uma constante c a todos os $y^{(i)}$ faz com que θ_0 também aumente de c e todas as previsões também aumentem c . Isso é um comportamento intuitivo e seria perdido. Além disso, a regularização passaria a depender da origem dos $y^{(i)}$ no treino.

A escolha de λ é uma decisão importante e será vista mais tarde.

Por exemplo, se λ for muito grande, p.ex., $\lambda = 10^{10}$, todos os parâmetros no termo regularizador, i.e., $\theta_1, \theta_2, \dots, \theta_n$ vão tender a zero, sobrando apenas θ_0 . A função de hipótese no caso seria a constante zero e obviamente não se ajustaria bem aos dados de treino, causando *underfitting*.

Já se λ for 0 ou muito pequeno, não será suficiente para resolver o *overfitting*.

4.3 Regressão Linear Regularizada

Como acabamos de ver, o problema de minimização que precisamos resolver na regressão linear regularizada será

$$J(\theta) = \frac{1}{2m} \left[\sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^n \theta_j^2 \right]$$

$$\underset{\theta}{\text{minimize}} J(\theta)$$

Vimos também que o gradiente descendente na regressão linear original (sem regularização) era

Repete até convergir: {

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}, \quad j = 0, \dots, n$$

}

Pode-se mostrar, usando Cálculo para obter a derivada de $J(\theta)$ regularizada, que o gradiente descendente para a regressão linear regularizada fica

Repete até convergir: {

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} + \frac{\lambda}{m} \theta_{j_2}, \quad j = 0, \dots, n \quad j_2 = 1, \dots, n$$

}

Note que o índice de θ no termo de regularização começa em 1 porque isso também ocorre na $J(\theta)$.

Para facilitar essa questão dos índices, vamos separar o algoritmo em duas partes: para $j = 0$ e $j = 1, \dots, n$.

Repete até convergir: {

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_0^{(i)}$$

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} + \frac{\lambda}{m} \theta_j, \quad j = 1, \dots, n$$

}

Para θ_0 é portanto igual à versão sem regularizar. Para os demais valores de j , podemos ainda reescrever como

$$\theta_j := \theta_j \left(1 - \alpha \frac{\lambda}{m}\right) - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

A 2^a parte representa a minimização de $J(\theta)$ da regressão original. Já a 1^a é um número < 1 (já que α, λ e m são todos > 0). Assim, a cada iteração do algoritmo, θ_j diminui, como de fato se espera da regularização.

4.3.1 Equação Normal

Vimos também que a regressão linear original podia ser resolvida analiticamente em passo único pela equação normal.

No caso, definimos a matriz X e o vetor y :

$$X = \begin{bmatrix} (x^{(1)})^T \\ \vdots \\ (x^{(m)})^T \end{bmatrix} \quad y = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

No caso regularizado, θ é obtido por

$$\theta = (X^T X + \lambda L)^{-1} X^T y,$$

em que L é uma matriz $(n+1) \times (n+1)$ semelhante à identidade, exceto que o primeiro elemento da diagonal é zero (devido ao fato de desconsiderarmos θ_0 na regularização):

$$L = \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}$$

IMPORTANTE: A expressão $X^T X$ não é invertível quando $m < n$. Até calculávamos um resultado aproximado desta inversão usando a função *pinv*, mas não podíamos usar a inversa exata. Já para $\lambda > 0$, a expressão $X^T X + \lambda L$ é invertível, permitindo o cálculo exato.

4.4 Regressão Logística Regularizada

A regressão logística pode ser regularizada de forma similar à regressão linear. Vimos que uma $h_\theta(x)$ muito complexa ou simplesmente muitos atributos tendem a causar *overfitting* na regressão logística. Na Figura 4.3 vemos que a curva rosa regularizada é menos propensa a tal fenômeno do que a curva azul original.

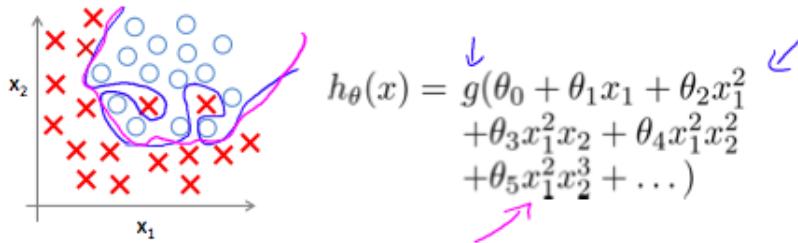


Figura 4.3: Regressão logística regularizada (rosa).

Lembrando que na regressão logística original tínhamos

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))]$$

A versão regularizada se obtém de forma idêntica à da regressão linear:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))] + \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2.$$

Note que novamente θ_0 é explicitamente excluído da regularização e todos os demais θ_j são penalizados.

O gradiente descendente também é adaptado de forma similar:

Repita até convergir: {

$$\begin{aligned} \theta_0 &:= \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_0^{(i)} \\ \theta_j &:= \theta_j - \alpha \left[\frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} + \frac{\lambda}{m} \theta_j \right], j = 1, \dots, n \end{aligned}$$

}

Claro que a diferença aqui para a regressão linear é a definição de $h_\theta(x)$, que agora é:

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}.$$

Note também que a expressão entre colchetes no algoritmo é nada mais do que $\frac{\partial}{\partial \theta} J(\theta)$ para a nova $J(\theta)$ regularizada.

Aqui temos espaço novamente para algoritmos mais avançados a otimização. Basta que tenhamos uma rotina que recebe uma função de várias variáveis e retorna o valor daquela função em um ponto assim como das derivadas em relação a cada argumento. Funções como *optimize* no Python ou *fminunc* no Octave podem ser usadas.

Capítulo 5

Redes Neurais

5.1 Redes Neurais: Hipóteses não lineares

Redes neurais são algoritmos bastante antigos, mas que só recentemente se tornaram o estado-da-arte na maioria das aplicações de aprendizado de máquinas.

Imagine um problema de classificação com uma fronteira de decisão altamente não linear como na Figura 5.1.

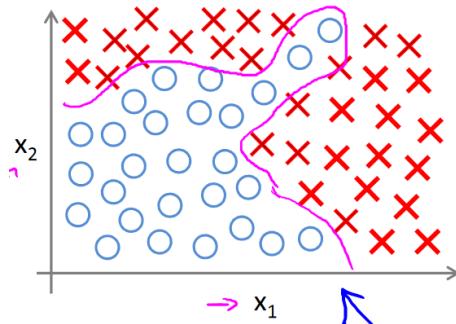


Figura 5.1: Problema de classificação altamente não linear.

Nesse caso ainda conseguiríamos separar pela regressão logística com uma função de hipótese muito complexa, e portanto já propensa a *overfitting*, algo como

$$g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1 x_2 + \theta_4 x_1^2 x_2 + \theta_5 x_1^3 x_2 + \theta_6 x_1 x_2^2 + \dots).$$

Agora imagine que tenhamos, por exemplo, 100 atributos (como no problema do preço de casas), um número que no mundo real pode ser considerado inclusive pequeno.

Só os termos quadráticos (produtos dois a dois $x_i x_j$) agora dariam ≈ 5000 ($\approx \frac{n^2}{2}$), o crescimento é de ordem $\mathcal{O}(n^2)$.

Para os termos cúbicos, teríamos ≈ 170000 ($\mathcal{O}(n^3)$).

Cenário ainda muito pior surge, por exemplo, em visão computacional.

Imagine o problema de detectar se uma imagem é um carro ou não. Para cada região da imagem o computador enxerga uma matriz gigantesca (Figura 5.2).

You see this:

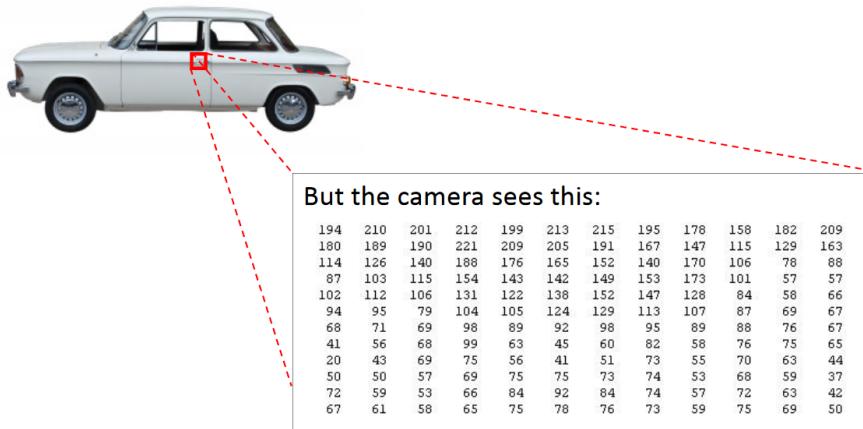


Figura 5.2: Representação de imagens digitais.

Imagine que nosso algoritmo usa cada pixel como um atributo. Para uma imagem cinza 50×50 (pequena), temos 2500 atributos (para RGB seria 7500).

Um polinômio não linear para a regressão logística teria ≈ 3 milhões de atributos só incluindo termos quadráticos.

Note como um modelo de regressão logística desse tipo seria inviável devido ao altíssimo *over-fitting* induzido.

5.2 Neurônios e o Cérebro

Redes neurais são algoritmos bastante antigos que buscam imitar o funcionamento do cérebro, nossa melhor máquina de aprendizado. Redes neurais também estão ligadas ao sonho da IA de um dia termos máquinas inteligentes de fato.

Aqui, porém, nosso interesse não é por essa imitação do cérebro, mas sim porque é uma ferramenta muito poderosa e flexível.

Foram muito usadas nos anos 1980 e início dos 1990, porém perdeu popularidade no final dos anos 1990.

Ressurgiu recentemente devido à disponibilidade de recursos computacionais para rodar e os resultados do estado-da-arte para muitas aplicações.

5.2.1 Hipótese do algoritmo de aprendizagem único

Será que uma máquina que faz tantas tarefas inteligentes como o cérebro não precisaria de muitos (infinitos?) algoritmos para imitá-la?

Estudos, em animais, desconectando o ouvido do córtex auditivo e ligando os olhos àquele córtex mostraram que o cérebro aprende a “ver” com aquela região (no sentido de tomar decisões com base no estímulo visual). Se o sinal dos olhos forem direcionados para o córtex somatossensorial (do tato), aquela região também aprende a ver. E em geral, qualquer região pode aprender qualquer coisa. Isso é chamado de *rewiring*.

Isso mostra que o “algoritmo” de aprendizagem do cérebro é único, independentemente da tarefa que precise realizar.

Existem aplicações práticas muito interessantes desse princípio:

- Sistema BrainPort/FDA para pessoas cegas: uma câmera na cabeça capta imagens em tons de cinza na sua frente e um cabo se conecta a uma matriz de eletrodos sobre a língua, de modo que cada pixel é mapeado para uma posição na língua e, por exemplo, pixels escuros correspondem a alta voltagem e pixels claros a baixa. Em dezenas de minutos, a pessoa aprende a “ver” com a língua!
- Sonar humano: estalando o dedo ou a língua e identificando o padrão do som reverberado, uma pessoa cega aprende a se deslocar pelo ambiente. O Youtube tem um exemplo de uma criança que perdeu os globos oculares devido a um câncer e pode se deslocar sem bater em nada, andar de skate, fazer uma cesta de basquete, etc.!
- Cinto haptico: emite um som quando a pessoa se alinha com o norte, dá um senso de direção similar, por exemplo, ao dos pássaros.
- Exemplos bizarros: um terceiro olho plugado em um sapo, que aprende a usá-lo como um olho natural.

Vemos então que aprender como esse algoritmo único do cérebro funciona e implementá-lo (mesmo que uma aproximação) deve causar um grande impacto no conceito de IA.

5.3 Redes Neurais: Representação do Modelo I

O neurônio é a célula no cérebro responsável pelas inúmeras tarefas inteligentes que fazemos.

Trata-se de uma unidade de processamento com uma conexão de entrada (**dendrito**) e uma de saída (**axônio**) (Figura 5.3).

E os neurônios se comunicam entre si (rede neural), enviando impulsos elétricos que saem pelo axônio de um e entram pelo dendrito de outro. Esses mesmos impulsos (nervosos) são, por exemplo, enviados pelo cérebro para um músculo, para que este contraia, ou recebido dos olhos quando há um estímulo visual.

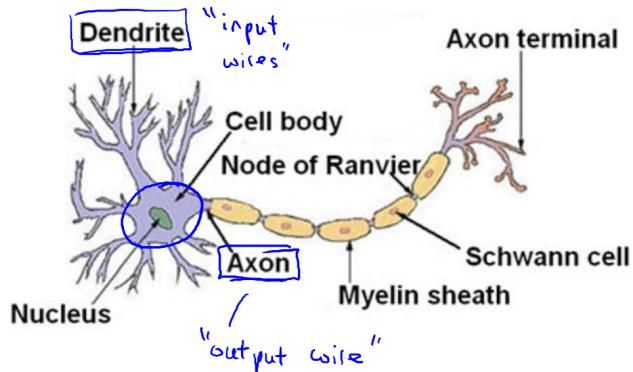


Figura 5.3: Neurônio.

No modelo computacional (Figura 5.4), cada “neurônio” é uma unidade de processamento logística (em laranja na figura), que recebe um vetor x de entrada e devolve uma saída $h_\theta(x)$ (função de hipótese).

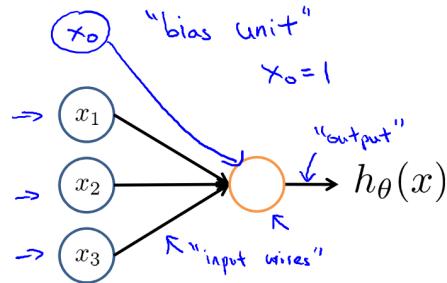


Figura 5.4: Neurônio computacional.

Para definirmos essa função de hipótese $h_\theta(x)$, partimos novamente do vetor de entradas x e vetor de parâmetros θ :

$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}$$

E assim como na regressão logística, temos

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}.$$

Novamente, temos $x_0 = 1$, que na representação gráfica da rede pode ou não aparecer e é chamada de “unidade de bias”.

Os parâmetros θ_j em redes neurais costumam ser chamados de **pesos**.

A função logística

$$g(z) = \frac{1}{1 + e^{-z}}$$

é chamada agora de **função de ativação** sigmoide (logística).

Temos finalmente uma **rede neural** quando juntamos vários desses neurônios e os agrupamos por camadas conectadas entre si (Figura 5.5).

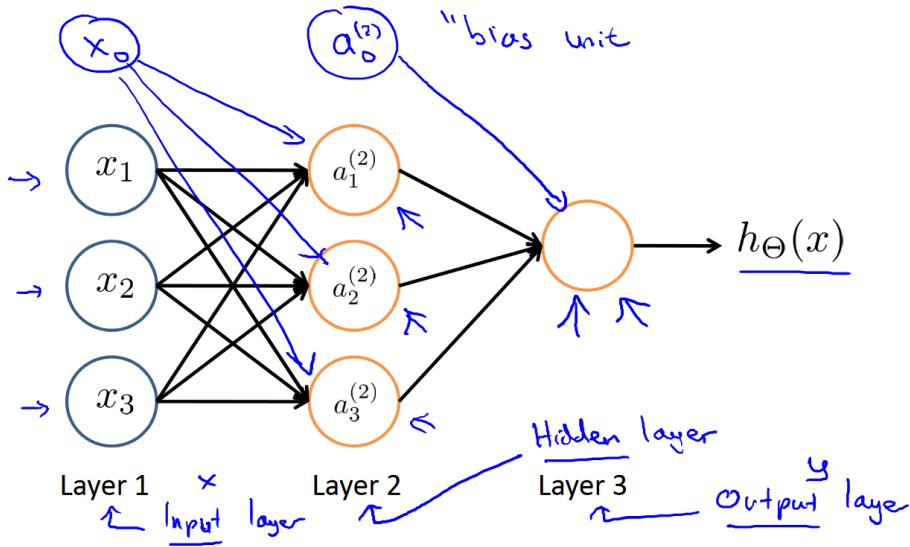


Figura 5.5: Rede neural.

A informação “caminha” pela rede. O dado de entrada é armazenado na camada 1 (**camada de entrada**), é passado para a camada 2 e finalmente para a camada 3 (**camada de saída**), em que a função de hipótese (saída do modelo) é calculada.

A camada 2 é chamada de **camada escondida**. Esse nome se deve a que no treinamento supervisionado conhecemos os valores da entrada e da saída (são visíveis), os demais não. Uma rede pode ter mais de uma camada escondida.

As unidades de processamento nas camadas escondidas e de saída são chamadas de **unidades de ativação**.

Outra novidade é que nossos parâmetros (pesos) serão armazenados agora em uma matriz.

Notação:

$a_i^{(j)}$ = ativação da unidade i na camada j (temos também uma unidade de *bias*).

$\Theta^{(j)}$ = matriz de pesos que controlam a função que mapeia a saída da camada j para a entrada da camada $j + 1$.

$\Theta_{ab}^{(j)}$ = parâmetro que multiplica a saída da unidade b da camada j para compor a entrada da unidade a na camada $j + 1$.

Os valores dessas ativações serão calculados por

$$a_1^{(2)} = g(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3)$$

$$a_2^{(2)} = g(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3)$$

$$a_3^{(2)} = g(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3)$$

E finalmente temos a função de hipótese (saída do modelo);

$$h_\theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)}).$$

Repare, por exemplo, que aqui temos 3 unidades de entrada, 3 unidades de saída e $\Theta^{(1)} \in \mathbb{R}^{3 \times 4}$. Em geral, se temos s_j unidades na camada j e s_{j+1} na camada $j + 1$, a matriz $\Theta^{(j)}$ terá dimensões $s_{j+1} \times (s_j + 1)$.

5.4 Redes Neurais: Representação do Modelo II

Veremos agora como vetorizar os cálculos da rede neural que acabamos de ver, assim como um esboço da intuição de seu funcionamento.

Essa etapa da rede neural em que ela calcula a função de hipótese é chamada de ***forward propagation***.

Lembrando dos cálculos:

$$a_1^{(2)} = g(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3)$$

$$a_2^{(2)} = g(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3)$$

$$a_3^{(2)} = g(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3)$$

$$h_\theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)}).$$

Vamos definir agora uma nova variável $z_k^{(j)}$, que corresponde ao parâmetro recebido pela função $g(x)$ no cálculo de $a_k^{(j)}$.

Ou seja, por exemplo, na Camada 2:

$$a_1^{(2)} = g(z_1^{(2)})$$

$$a_2^{(2)} = g(z_2^{(2)})$$

$$a_3^{(2)} = g(z_3^{(2)})$$

e temos então um vetor $z_k^{(2)}$ dado por

$$z_k^{(2)} = \Theta_{k0}^{(1)}x_0 + \Theta_{k1}^{(1)}x_1 + \Theta_{k2}^{(1)}x_2 + \Theta_{k3}^{(1)}x_3, \quad k = 1, 2, 3. \quad (5.1)$$

Note agora como podemos usar o vetor x e definir um novo vetor $z^{(2)}$:

$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad z^{(2)} = \begin{bmatrix} z_1^{(2)} \\ z_2^{(2)} \\ z_3^{(2)} \end{bmatrix}$$

Note como, pela definição (5.1), podemos escrever

$$z^{(2)} = \Theta^{(1)}x.$$

Mas x pode também ser chamado de $a^{(1)}$, ou seja:

$$z^{(2)} = \Theta^{(1)}a^{(1)}.$$

Podemos então calcular o vetor $a^{(2)}$ por

$$a^{(2)} = g(z^{(2)}),$$

em que a função $g(x)$ é aplicada ponto-a-ponto.

Finalmente, para calcular nossa função de hipótese na Camada 3, precisamos de um *bias* na Camada 2, ou seja, adicionamos $a_0^{(2)} = 1$.

Novamente, chamaremos o argumento da função $g(x)$ na Camada 3 de $z^{(3)}$, de modo que

$$z^{(3)} = \Theta^{(2)}a^{(2)}.$$

Temos uma linha apenas em $z^{(3)}$ agora e esse produto interno que o gera é um escalar.

Finalmente, obtemos a função de hipótese $h_\Theta(x)$:

$$h_\Theta(x) = a^{(3)} = g(z^{(3)}).$$

O quadro abaixo resume o processo:

$z^{(2)} = \Theta^{(1)}a^{(1)}$
$a^{(2)} = g(z^{(2)})$
Adicionar $a_0^{(2)} = 1$
$z^{(3)} = \Theta^{(2)}a^{(2)}$
$h_\Theta(x) = a^{(3)} = g(z^{(3)})$

Na verdade, em geral, mesmo para mais camadas, sempre teremos:

$$z^{(j)} = \Theta^{(j-1)}a^{(j-1)}$$

$$a^{(j)} = g(z^{(j)})$$

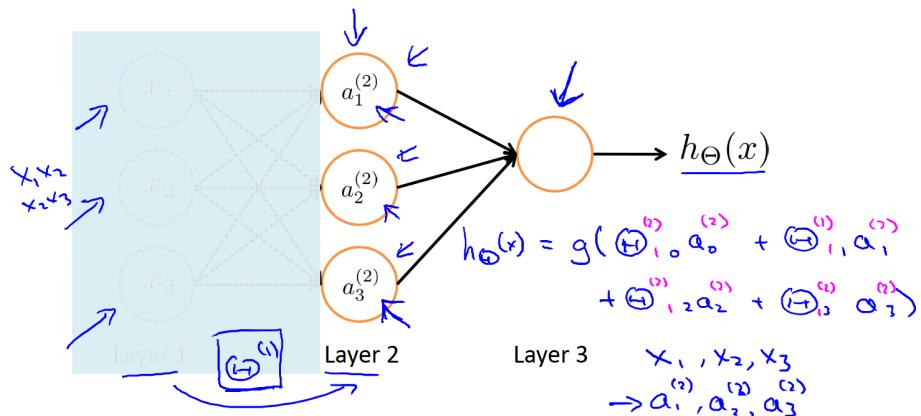


Figura 5.6: Parte de uma rede neural como uma regressão logística.

5.4.1 Atributos aprendidos pela rede

Se tomarmos só a parte da rede na Figura 5.5 a partir da Camada 2, temos nada mais do que a função de hipótese logística que já vimos (Figura 2.5).

A diferença é que ela não é calculada em cima do dado de entrada x , mas sim em cima de atributos da Camada 2 ($a^{(2)}$). Estes por sua vez são atributos de alto nível aprendidos agora sim a partir do dado x . E, novamente, o cálculo desses atributos é feito pela função logística (com outros parâmetros).

Portanto a função de hipótese não fica restrita a depender apenas do dado de entrada, nem de combinações polinomiais, mas a rede é capaz de aprender **seus próprios atributos**. Veremos melhor isso em seguida, com um exemplo mais específico.

5.4.2 Arquitetura da rede

Por fim, podemos ter redes com outros arranjos de nós e conexões, isso é o que se chama de **arquitetura da rede**. A Figura 2.6 mostra um exemplo de outra arquitetura. Quanto maior a camada, mais complexos são os atributos aprendidos em seus neurônios.

Como já vimos, a Camada 1 é a de entrada, a última é a de saída (Camada 4 na Figura 2.6) e todas as demais (Camadas 2 e 3 na Figura 5.7) são as camadas escondidas.

5.5 Exemplos e Intuições I

Vamos ver como a rede consegue aprender uma função de hipótese com fronteiras altamente não lineares. Queremos que a rede aprenda fronteiras complexas como a da Figura 5.8.

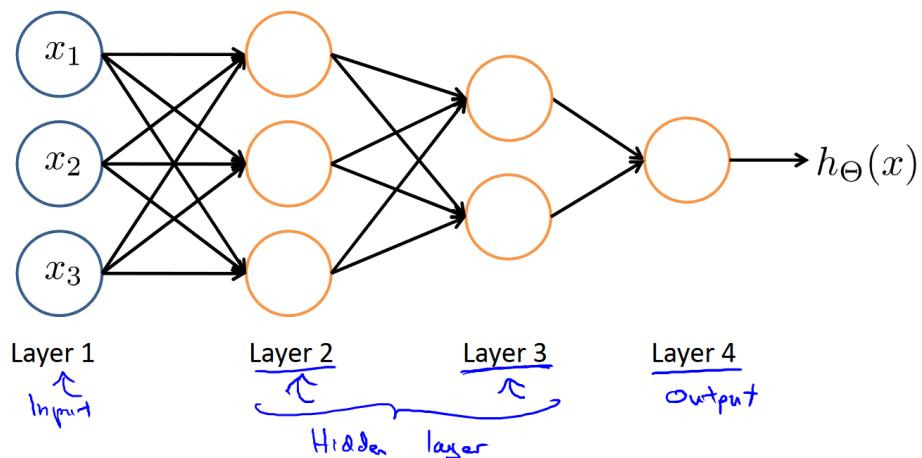


Figura 5.7: Arquitetura de rede neural com mais camadas.

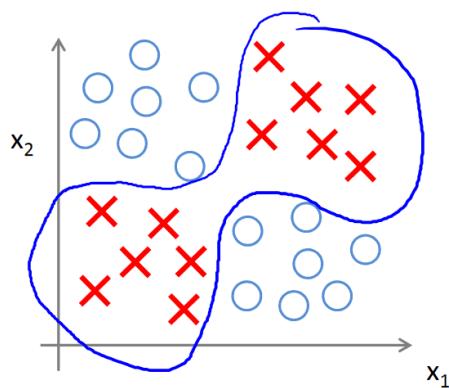


Figura 5.8: Problema com fronteira altamente não linear.

Vamos partir de exemplos em que a entrada (x_1 e x_2) é binária (0 ou 1) e vamos focar nas operações lógicas que o computador faz. Veremos que uma rede neural pode simular qualquer porta lógica.

5.5.1 Exemplo 1: função AND

A função x_1 AND x_2 pode ser executada pela rede na Figura 5.9. Como de praxe, temos uma unidade de *bias* com valor fixo em 1. Temos portanto:

$$\Theta_{10}^{(1)} = -30 \quad \Theta_{11}^{(1)} = 20 \quad \Theta_{12}^{(1)} = 20$$

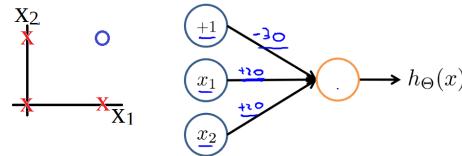


Figura 5.9: Rede neural para a operação AND.

Adotaremos agora a convenção de por o valor dos pesos nas conexões, de modo que

$$h_\Theta(x) = -30 + 20x_1 + 20x_2.$$

Agora, olhamos para o gráfico da função logística na Figura 5.10. Repare como para $x < -4$ temos $g(x) \approx 0$ e para $x > 4$, $g(x) \approx 1$.

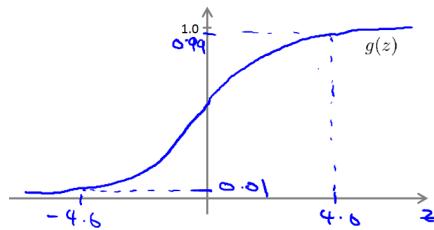


Figura 5.10: Função sigmoide logística.

Vamos agora construir a tabela de valores para $h_\Theta(x)$:

x_1	x_2	$h_\Theta(x)$
0	0	$g(-30) \approx 0$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(10) \approx 1$

Repare como deste modo temos

$$h_\Theta(x) \approx x_1 \text{ AND } x_2.$$

5.5.2 Exemplo 2: função OR

A rede da Figura 5.11 corresponde à função OR. Note que agora

$$h_{\Theta}(x) = -10 + 20x_1 + 20x_2.$$

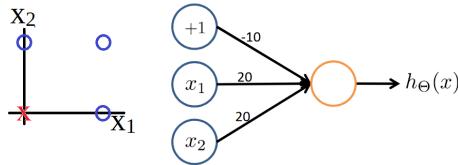


Figura 5.11: Rede neural para a operação OR.

e temos a tabela

x_1	x_2	$h_{\Theta}(x)$
0	0	$g(-10) \approx 0$
0	1	$g(10) \approx 1$
1	0	$g(10) \approx 1$
1	1	$g(10) \approx 1$

e portanto

$$h_{\Theta}(x) \approx x_1 \text{ OR } x_2.$$

5.6 Exemplos e Intuições II

5.6.1 Função NOT

Neste caso, a rede da Figura 5.12 resolve.

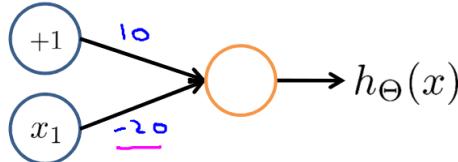


Figura 5.12: Rede neural para a operação NOT.

Repare que teremos

$$h_{\Theta}(x) = g(10 - 20x_1)$$

e a seguinte tabela:

x_1	$h_{\Theta}(x)$
0	$g(10) \approx 1$
1	$g(-10) \approx 0$

5.6.2 Função ($\text{NOT } x_1$) AND ($\text{NOT } x_2$)

Esta é uma função lógica cuja saída é 1 se e somente se $x_1 = x_2 = 0$.

A rede da Figura 5.13 resolve.

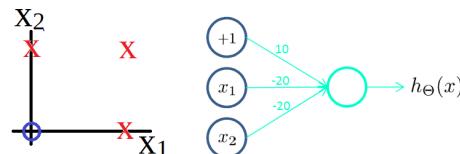


Figura 5.13: Rede neural para a operação ($\text{NOT } x_1$) AND ($\text{NOT } x_2$).

Temos agora

$$h_\theta(x) = 10 - 20x_1 - 20x_2$$

e a tabela

x_1	x_2	$h_\theta(x)$
0	0	$g(10) \approx 1$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(-30) \approx 0$

5.6.3 Juntando as peças: $x_1 \text{ XNOR } x_2$

A Figura 5.14 ilustra todo o processo. Repare como os pontos positivos e negativos não podem ser separados por nenhuma reta: problema não linear.

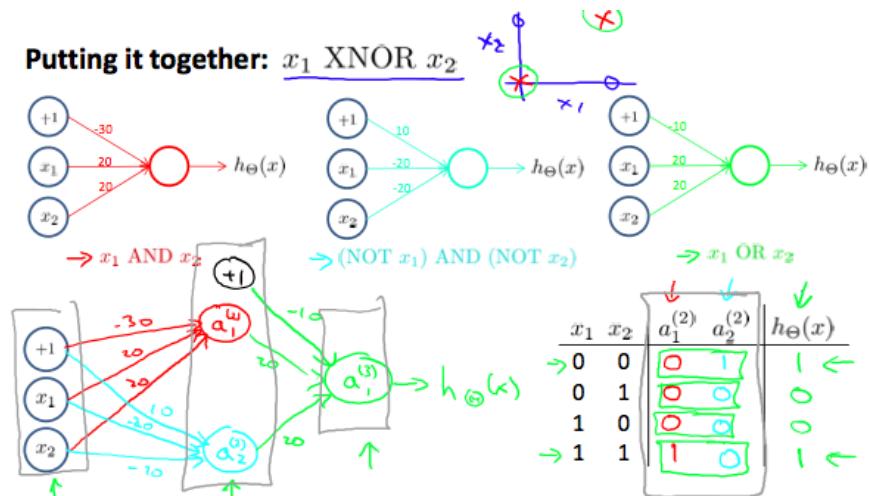


Figura 5.14: Rede neural para a operação XNOR.

Mas podemos combinar as redes $x_1 \text{ AND } x_2$, $(\text{NOT } x_1) \text{ AND } (\text{NOT } x_2)$ e $x_1 \text{ OR } x_2$, já vistas, em 3 camadas e assim resolvemos o problema.

Lembrando da nossa notação, temos aqui então entre a Camada 1 e Camada 2:

$$\Theta^{(1)} = \begin{bmatrix} -30 & 20 & 20 \\ 10 & -20 & -20 \end{bmatrix}$$

e entre a Camada 2 e Camada 3:

$$\Theta^{(2)} = \begin{bmatrix} -10 & 20 & 20 \end{bmatrix}$$

e as ativações então são calculadas por

$$a^{(2)} = g(\Theta^{(1)}x)$$

$$a^{(3)} = g(\Theta^{(2)}a^{(2)})$$

$$h_{\Theta}(x) = a^{(3)}.$$

Qualquer função lógica pode ser executada por uma rede neural deste tipo.

Repare como as funções lógicas em cada camada são aplicadas de forma composta, gerando funções não lineares mais e mais complexas.

Considere, por exemplo, a classificação de dígitos manuscritos (endereçamento automático de cartas nos EUA por reconhecimento de CEP, LeCun, primórdios do deep learning). A partir das primeiras camadas, a rede inicia com traços bem genéricos e vai no decorrer das camadas construindo o dígito correto.

5.7 Redes Neurais: Classificação Multi-classes

Um problema como o de classificar dígitos tem múltiplas classes.

Usamos a abordagem “one-vs-all” que já vimos na regressão logística.

Imagine classificarmos imagens entre “pedestre”, “carro”, “moto” e “caminhão”.

Vamos usar um neurônio para cada classe na saída, de modo que aqui no caso

$$h_{\Theta}(x) \in \mathbb{R}^4.$$

A rede da Figura 5.15 mostra a solução.

E assim nós teremos pedestre se

$$h_{\Theta}(x) \approx \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix},$$

carro se

$$h_{\Theta}(x) \approx \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix},$$

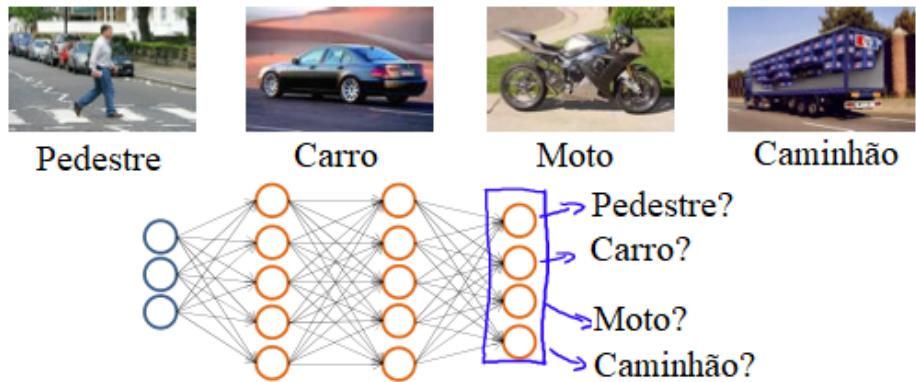


Figura 5.15: Rede neural para múltiplas classes.

moto se

$$h_{\Theta}(x) \approx \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix},$$

etc.

Repare como cada neurônio na saída corresponde a um classificador logístico que separa aquela classe das demais.

Uma consequência disso é que a soma dos elementos do vetor $h_{\Theta}(x)$ será sempre 1.

Por fim, continuamos representando nosso conjunto de treinamento como

$$(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots (x^{(m)}, y^{(m)}),$$

porém agora $y^{(i)}$ é um vetor de 4 elementos, isto é:

$$y^{(i)} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

para pedestre,

$$y^{(i)} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

para carro,

$$y^{(i)} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

para moto e

$$y^{(i)} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

para caminhão.

Finalmente, temos

$$h_{\Theta}(x^{(i)}) = y^{(i)},$$

em que ambos os lados da equação estão em \mathbb{R}^4 .

5.8 Redes Neurais: Função de Custo

Notação:

Treinamento	$\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})\}$
L	Número total de camadas da rede
s_l	Número de unidades (neurônios) (sem contar o <i>bias</i>) na camada l

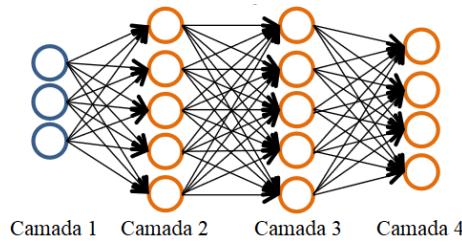


Figura 5.16: Rede neural.

No exemplo da Figura 5.16 temos $L = 4$, $s_1 = 3$, $s_2 = 5$, $s_4 = s_L = 4$.

Na classificação binária ($y = 0$ ou 1) temos uma unidade de saída: $s_L = 1$.

Para K classes, com $K \geq 3$, temos K unidades de saída ($y \in \mathbb{R}^K$), lembrando do exemplo visto:

$$\begin{array}{llll} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} & \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} & \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} & \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \\ \text{pedestre} & \text{carro} & \text{moto} & \text{caminhão} \end{array}$$

5.8.1 Função de custo

Recordando da função de custo da regressão logística:

$$J(\theta) = -\frac{1}{m} \left[\sum_{i=1}^m y^{(i)} \log h_{\theta}(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_{\theta}(x^{(i)})) \right] + \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2,$$

lembrando que não regularizamos θ_0 por convenção (não faz grande diferença, na verdade).

O que temos agora no caso das redes neurais é simplesmente um somatório dessa função de custo para cada unidade de saída:

$$J(\Theta) = -\frac{1}{m} \left[\sum_{i=1}^m \sum_{k=1}^K y_k^{(i)} \log(h_\Theta(x^{(i)}))_k + (1 - y_k^{(i)}) \log(1 - (h_\Theta(x^{(i)}))_k) \right] + \frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (\Theta_{ji}^{(l)})^2,$$

em que $(h_\Theta(x))_i$ é a i -ésima saída da função de hipótese (não confundir com o i -ésimo exemplo de treinamento) e y_k é o k -ésimo componente do vetor de saída y .

Lembrando que, na matriz Θ corrente, o número de colunas é o número de nós na camada corrente (incluindo o *bias*) e o número de linhas é o número de nós na próxima camada (excluindo o *bias*).

Note que, na segunda parcela, não incluímos $i = 0$, que corresponde exatamente às unidades de *bias*.

5.9 Algoritmo *Backpropagation*

Veremos agora como minimizar a função $J(\Theta)$:

$$J(\Theta) = -\frac{1}{m} \left[\sum_{i=1}^m \sum_{k=1}^K y_k^{(i)} \log(h_\Theta(x^{(i)}))_k + (1 - y_k^{(i)}) \log(1 - (h_\Theta(x^{(i)}))_k) \right] + \frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (\Theta_{ji}^{(l)})^2,$$

ou seja, queremos obter

$$\underset{\Theta}{\text{minimize}} J(\Theta).$$

Como já vimos em outros algoritmos, seja para o gradiente descendente ou outro otimizador mais avançado, precisamos sempre calcular:

- $J(\Theta)$
- $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$

A função de custo é conhecida, a questão agora é seu gradiente.

Vamos partir de um exemplo apenas de treinamento (x, y) .

Temos então o *forward propagation*, que no exemplo da Figura 5.17 seria:

$$\begin{aligned} a^{(1)} &= x \\ z^{(2)} &= \Theta^{(1)} a^{(1)} \\ a^{(2)} &= g(z^{(2)}) \text{ (e soma } a_0^{(2)}) \\ z^{(3)} &= \Theta^{(2)} a^{(2)} \\ a^{(3)} &= g(z^{(3)}) \text{ (e soma } a_0^{(3)}) \\ z^{(4)} &= \Theta^{(3)} a^{(3)} \\ a^{(4)} &= h_\Theta(x) = g(z^{(4)}) \end{aligned}$$

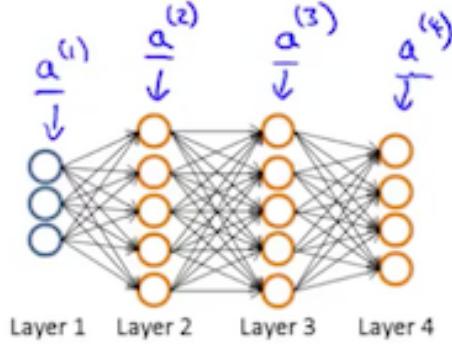


Figura 5.17: Rede neural.

5.9.1 Cálculo do gradiente: *backpropagation*

Definimos uma nova variável:

$$\delta_j^{(l)} = \text{“erro” do nó } j \text{ na camada } l.$$

Este erro está associado à diferença entre o valor calculado para a ativação $a_j^{(l)}$ e o valor esperado daquela ativação para que a função de custo fosse minimizada.

Na unidade de saída ($L = 4$):

$$\delta_j^{(4)} = a_j^{(4)} - y_j,$$

ou seja, é a diferença entre o valor calculado $a_j^{(4)} = (h_\Theta(x))_j$ e a saída “real” y_j no treinamento.

Note ainda que essa operação pode ser vetorizada:

$$\delta^{(4)} = a^{(4)} - y.$$

Para as demais camadas, o erro é calculado de trás para frente (daí o nome “*backpropagation*”):

$$\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} \cdot g'(z^{(3)})$$

$$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} \cdot g'(z^{(2)}),$$

em que $\cdot\cdot\cdot$ é o produto ponto-a-ponto (notação Octave) e $g'(z)$ é a derivada da sigmoide $g(z)$ ponto-a-ponto.

OBSERVAÇÕES:

- Não temos o erro $\delta^{(1)}$ pois na camada 1 temos o dado de entrada x e este não será ajustado.
- Pode-se mostrar usando Cálculo que $g'(z^{(3)}) = a^{(3)} \cdot (1 - a^{(3)})$ e $g'(z^{(2)}) = a^{(2)} \cdot (1 - a^{(2)})$.

Finalmente, para o caso $\lambda = 0$, temos as derivadas parciais dadas por

$$\boxed{\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^{(l)} \delta_i^{(l+1)}}.$$

5.9.2 Backpropagation geral

Agora com vários exemplos de treinamento:

$$\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$$

Para o cálculo de $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$, vamos definir uma variável acumuladora para o gradiente:

$$\Delta_{ij}^{(l)} = 0 \quad \text{para todo } l, i, j$$

O algoritmo então executa os seguintes passos:

Para $i = 1$ até m
 $a^{(1)} := x^{(i)}$
 Calcular $a^{(l)}$ para $l = 2, 3, \dots, L$ (*forward*)
 $\delta^{(L)} := a^{(L)} - y^{(i)}$
 Calcular $\delta^{(L-1)}, \delta^{(L-2)}, \dots, \delta^{(2)}$
 $\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

OBSERVAÇÕES:

- A última linha pode ser vetorizada por $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)}(a^{(l)})^T$.
- Repare como, para cada exemplo do treinamento, roda-se uma vez o *forward* e uma vez o *backward*.

Por fim, o gradiente (derivada parcial) é calculado, separando-se o *bias* dos demais pesos, por

$$\boxed{\begin{aligned} D_{ij}^{(l)} &:= \frac{1}{m} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)} && \text{se } j \neq 0 \\ D_{ij}^{(l)} &:= \frac{1}{m} \Delta_{ij}^{(l)} && \text{se } j = 0 \\ \frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) &= D_{ij}^{(l)} \end{aligned}}$$

A derivada parcial pode ser usada então tanto no gradiente descendente quanto em qualquer algoritmo de otimização mais avançado como vimos anteriormente.

5.10 Intuição do Backpropagation

O algoritmo do *backpropagation* não tem uma intuição simples como a regressão linear ou logística.

Mas podemos partir do *forward propagation* (Figura 5.18). Aqui, cada ativação $a_i^{(l)}$ é calculada pela aplicação da sigmoide sobre a combinação linear $z_i^{(l)}$.

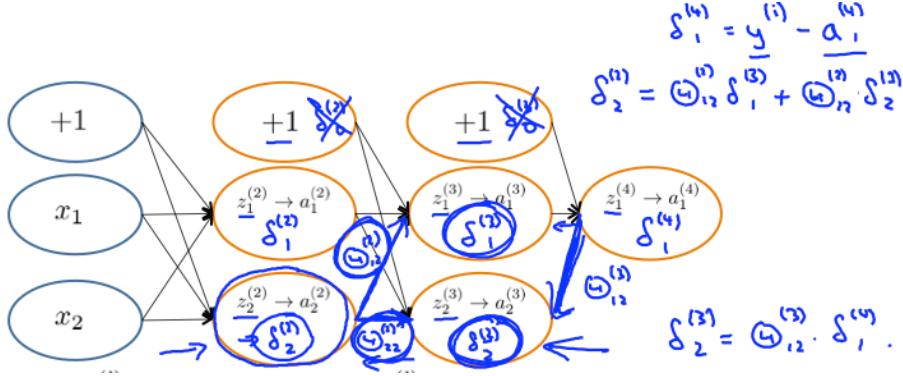


Figura 5.18: Intuição do *backpropagation*.

Agora vejamos, por exemplo, o cálculo de $z_1^{(3)}$:

$$z_1^{(3)} = \Theta_{10}^{(2)} \cdot 1 + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)}.$$

Vamos ver que no *backpropagation* o cálculo é muito parecido, apenas executado na direção oposta.

Vamos lembrar da nossa função de custo, pensando no caso mais simples de uma unidade de saída apenas:

$$J(\Theta) = -\frac{1}{m} \left[\sum_{i=1}^m y^{(i)} \log(h_\Theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - (h_\Theta(x^{(i)}))) \right] + \frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (\Theta_{ji}^{(l)})^2.$$

Vamos ainda considerar um único exemplo $\{x^{(i)}, y^{(i)}\}$ e ignorar a regularização ($\lambda = 0$):

$$\text{custo}(i) = y^{(i)} \log(h_\Theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - (h_\Theta(x^{(i)}))).$$

Pode-se considerar ainda a simplificação seguinte:

$$\text{custo}(i) \approx (h_\Theta(x^{(i)}) - y^{(i)})^2.$$

Temos então que a variável $\delta_j^{(l)}$ é definida por

$\delta_j^{(l)}$ = “erro” do custo associado a $a_j^{(l)}$ (unidade j na camada l). Formalmente, temos

$$\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{custo}(i), \quad j \geq 0,$$

em que, como vimos:

$$\text{custo}(i) = y^{(i)} \log(h_\Theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - (h_\Theta(x^{(i)}))).$$

Ou seja, o que $\delta_j^{(l)}$ efetivamente mede é a alteração causada na função de custo por uma alteração em $z_j^{(l)}$.

Na unidade de saída, este “erro” é calculado então de forma exata:

$$\delta_1^{(4)} = a_1^{(4)} - y^{(i)}.$$

Agora, se quisermos calcular $\delta_j^{(l)}$ para os nós internos, fazemos o *forward* ao contrário, ou seja, multiplicamos os pesos que saem do nó pelos δ nos nós em que estes pesos chegam. Por exemplo:

$$\delta_2^{(2)} = \Theta_{12}^{(2)} \delta_1^{(3)} + \Theta_{22}^{(2)} \delta_2^{(3)}$$

$$\delta_2^{(3)} = \Theta_{12}^{(3)} \delta_1^{(4)}$$

NOTA: Em nossa implementação, não vamos definir um $\delta_0^{(l)}$ para as unidades de *bias*, já que elas são sempre um.

5.11 Nota de implementação: vetorizando parâmetros

Enquanto antes nossos parâmetros eram armazenados em vetores, agora temos matrizes de pesos:

$$\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}, \dots,$$

bem como de gradientes:

$$D^{(1)}, D^{(2)}, D^{(3)}, \dots.$$

Porém, algoritmos de otimização avançados, como *fminunc()* no Octave, pressupõem que os parâmetros e gradientes serão passados como vetores.

Para resolver isso, linearizamos as matrizes gerando um único vetor longo para os parâmetros e outro para os gradientes. No Octave fica:

```
thetaVector = [Theta1(:);Theta2(:);Theta3(:)]
deltaVector = [D1(:);D2(:);D3(:)]
```

Supondo, por exemplo, que $s_1 = 10$, $s_2 = 10$ e $s_3 = 1$. Assim, Theta1 é 10×11 , Theta2 é 10×11 e Theta3 é 1×11 . Nossa vetor thetaVector seria então 231×1 . A matriz de gradientes em cada camada terá o mesmo tamanho e seria linearizada do mesmo modo.

Se quiséssemos então recuperar as matrizes originais, faríamos um *reshape*. No Octave:

```
Theta1 = reshape(thetaVector(1:110),10,11)
Theta2 = reshape(thetaVector(111:220),10,11)
Theta3 = reshape(thetaVector(221:231),1,11)
```

Em resumo, temos o seguinte algoritmo de aprendizado:

```
Entrar parâmetros iniciais  $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$ .
Linearizar para obter  $initialTheta$  e passar para
 $fminunc(@costFunction, initialTheta, options)$ 
```

```
function[jVal, gradientVec] = costFunction(thetaVec).
    Usando thetaVec, obter  $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$ .
    Forward/Back propagation para calcular  $D^{(1)}, D^{(2)}, D^{(3)}$  e  $J(\Theta)$ .
    Linearizar  $D^{(1)}, D^{(2)}, D^{(3)}$  para obter gradientVec.
```

5.12 Checagem do Gradiente

O algoritmo de *backpropagation* é cheio de detalhes e há muitos tipos de *bugs* sutis que podem aparecer.

De modo que mesmo que $J(\theta)$ esteja diminuindo em toda iteração, ele pode estar com algum *bug* imperceptível que faz com que sua rede tenha um erro maior do que se não tivesse *bug*.

Uma forma de checar isso é calculando o gradiente numericamente.

Sabemos que a derivada é a tangente da curva $J(\theta)$, mas podemos aproximá-la pela secante (Figura 5.19), de modo que:

$$\frac{d}{d\theta} J(\theta) \approx \frac{J(\theta + \epsilon) - J(\theta - \epsilon)}{2\epsilon},$$

em que um valor típico para ϵ é 10^{-4} . Valores muito grandes não aproximam bem e muito pequenos podem ocasionar problemas numéricos.

Aqui vale observar que essa aproximação também poderia ser feita pela expressão

$$\frac{J(\theta + \epsilon) - J(\theta)}{\epsilon},$$

chamada de diferença unilateral. Porém a primeira (diferença bilateral) é um pouco mais precisa do que esta última.

Poderíamos implementar essa derivada no Octave como

```
gradApprox = (J(theta + EPSILON)-J(theta - EPSILON)) / (2*EPSILON)
```

Agora, na rede neural, temos um vetor de parâmetros $\theta \in \mathbb{R}^n$:

$$\theta = [\theta_1, \theta_2, \theta_3, \dots, \theta_n]$$

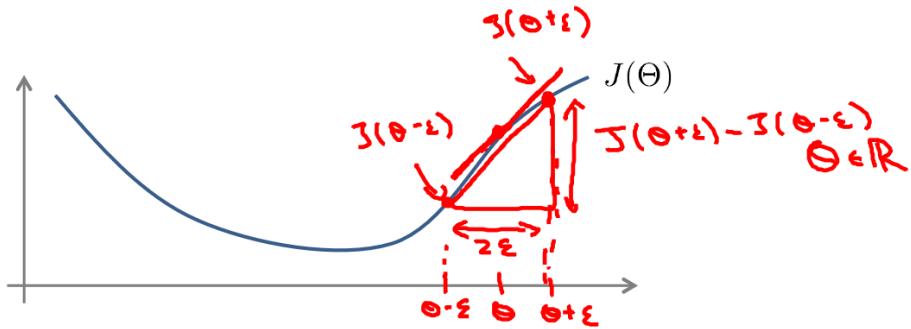


Figura 5.19: Aproximação da derivada.

Este poderia ser, por exemplo, a linearização de $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$.

O gradiente pode ser aproximado numericamente nesse caso pelo conjunto de equações

$$\begin{aligned}\frac{\partial}{\partial \theta_1} J(\theta) &\approx \frac{J(\theta_1 + \epsilon, \theta_2, \theta_3, \dots, \theta_n) - J(\theta_1 - \epsilon, \theta_2, \theta_3, \dots, \theta_n)}{2\epsilon} \\ \frac{\partial}{\partial \theta_2} J(\theta) &\approx \frac{J(\theta_1, \theta_2 + \epsilon, \theta_3, \dots, \theta_n) - J(\theta_1, \theta_2 - \epsilon, \theta_3, \dots, \theta_n)}{2\epsilon} \\ &\vdots \\ \frac{\partial}{\partial \theta_n} J(\theta) &\approx \frac{J(\theta_1, \theta_2, \theta_3, \dots, \theta_n + \epsilon) - J(\theta_1, \theta_2, \theta_3, \dots, \theta_n - \epsilon)}{2\epsilon}.\end{aligned}$$

No Octave, temos a seguinte implementação vetorizada:

```
for i=1:n,
    thetaPlus = theta;
    thetaPlus(i) = thetaPlus(i) + EPSILON;
    thetaMinus = theta;
    thetaMinus(i) = thetaMinus(i) - EPSILON;
    gradApprox(i) = (J(thetaPlus)-J(thetaMinus))/(2*EPSILON);
end;
```

Os seguintes passos são então executados:

- Rodar o *backpropagation* para calcular o vetor de gradientes *DVec* (vetorização de $D^{(1)}, D^{(2)}, D^{(3)}$)
- Rodar o gradiente numérico para calcular *gradApprox*
- Certificar-se de que

$$DVec \approx gradApprox$$

- Tendo verificado **uma vez** que o *backpropagation* está correto, remover o algoritmo de checagem e prosseguir com o *backpropagation* normal. IMPORTANTE: Essa é uma etapa importante pois o cálculo numérico do gradiente é muito lento!

5.13 Inicialização Aleatória

Vimos que tanto o gradiente descendente quanto métodos avançados de otimização precisam de valores iniciais para Θ :

```
optTheta = fminunc(@costFunction,initialTheta,options)
```

No caso do gradiente descendente, poderíamos fazer todos os valores iniciais iguais a zero?

```
initialTheta = zeros(n,1)
```

Vimos que isso funcionava na regressão logística, mas isso não ocorre nas redes neurais.

Observe a Figura 2.4. Se fizermos

$$\Theta_{ij}^{(l)} = 0, \text{ para todo } i, j, l,$$

vamos ter em todas as iterações:

$$\delta_1^{(2)} = \delta_2^{(2)},$$

que por sua vez vai implicar em

$$\frac{\partial}{\partial \Theta_{01}^{(1)}} J(\Theta) = \frac{\partial}{\partial \Theta_{02}^{(1)}} J(\Theta)$$

e portanto a atualização dos parâmetros pelo gradiente resulta em

$$\Theta_{01}^{(1)} = \Theta_{02}^{(1)}$$

e na verdade todos os pares de parâmetros com a mesma cor na Figura 5.20 vão ter sempre os mesmos valores (ainda que esse valor mude a cada iteração).

Finalmente isso faz com que todas as ativações na camada escondida tenham o mesmo valor sempre:

$$a_1^{(2)} = a_2^{(2)}$$

Essa é uma rede altamente redundante pois é equivalente a ter apenas um nó na camada escondida e, portanto, não é capaz de aprender nada interessante.

Esse é o chamado problema da simetria e ocorre na verdade sempre que Θ é inicializado com todos os valores iguais (mesmo que não sejam zero!).

A forma de se fazer uma **quebra de simetria** é usando uma inicialização aleatória independente para cada parâmetro.

Para isso então, inicializamos cada $\Theta_{ij}^{(l)}$ como um valor aleatório pequeno próximo de zero, em $[-\epsilon, \epsilon]$ (i.e., $-\epsilon \leq \Theta_{ij}^{(l)} \leq \epsilon$)

Um exemplo no Octave seria

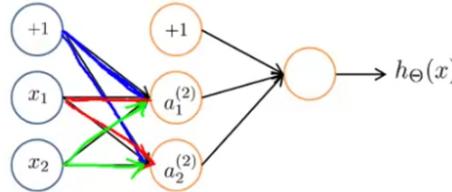


Figura 5.20: Inicialização com zeros.

```
Theta1 = rand(10,11)*(2*INIT_EPSILON) - INIT_EPSILON;
Theta2 = rand(1,11)*(2*INIT_EPSILON) - INIT_EPSILON;
```

em que a função *rand* gera matrizes de números aleatórios entre 0 e 1 e *INIT_EPSILON* aqui não tem nada a ver com *EPSILON* da checagem de gradiente.

5.14 Juntando as Peças

O primeiro passo é definir a arquitetura, i.e., o padrão de conectividade da rede (número de camadas e neurônios).

Na 1a camada, o número de unidades é a dimensão da entrada $x^{(i)}$.

Na saída, é o número de classes. Lembre-se que nosso vetor de saída não contém apenas o número da classe, mas sim é da forma

$$\begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \quad \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix} \quad \dots$$

Na camada escondida, usualmente quanto mais unidades melhor, o limite costuma ser o custo computacional. É comum que se use um valor comparável ao número de unidades na entrada (ou até 3 a 4 vezes a dimensão da entrada).

Para o número de camadas escondidas, usualmente temos apenas **uma**. Se tiver mais de uma, recomenda-se que tenham o mesmo número de unidades.

5.14.1 Treinamento de uma rede neural

1. Inicializar os pesos aleatoriamente (valores próximos de zero)
2. *Forward propagation* para obter $h_{\Theta}(x^{(i)})$ para todo $x^{(i)}$
3. Calcular função de custo $J(\Theta)$
4. *Backpropagation* para calcular as derivadas parciais $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$
for $i = 1:m$
Forward propagation e backpropagation usando os exemplos $(x^{(i)}, y^{(i)})$

- (Obter as ativações $a^{(l)}$ e os deltas $\delta^{(l)}$ para $l = 2, \dots, L$).
 $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)}(a^{(l)})^T$
 ...
 endfor
 ...
 Calcular $\frac{\partial}{\partial \Theta^{(l)}} J(\Theta)$
5. Usar a checagem de gradiente para comparar $\frac{\partial}{\partial \Theta_{jk}^{(l)}}$ calculado pelo *backpropagation* com a estimativa numérica do gradiente de $J(\Theta)$. Em seguida, desabilitar essa checagem de gradiente.
 6. Usar gradiente descendente ou qualquer outro método avançado de otimização **em conjunto com** o *backpropagation* para minimizar $J(\Theta)$. Lembre-se de que todos esses algoritmos vão precisar de $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$ e é exatamente isso que o *backpropagation* calcula.

NOTA: Existem implementações mais avançadas em que os exemplos de entrada são processados de uma vez (ou em grupos) para fomentar ainda mais vetorização. Mas sugere-se sempre usar a versão mais simples, com um exemplo por vez (for $i=1:m$), em uma primeira implementação.

Por fim, cabe destacar que, ao contrário do que ocorria na regressão linear e logística, a função de custo $J(\Theta)$ das redes neurais em geral é **não convexa**. A Figura 5.21 ilustra isso em um caso bem simplificado (olhando para 2 parâmetros apenas).

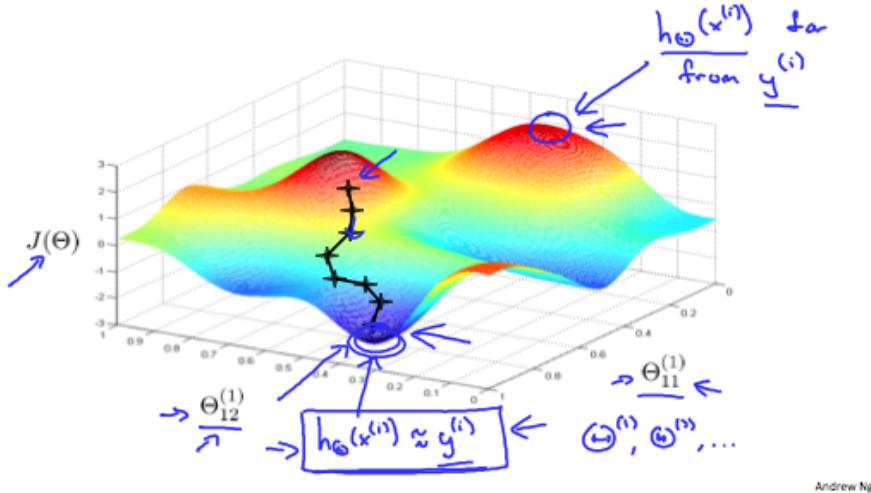


Figura 5.21: Função de custo não convexa da rede neural.

O algoritmo pode portanto ficar preso em mínimos locais, mas o que se mostra na prática é que seguindo o procedimento aqui descrito, isso não é de

fato um grande problema e, mesmo que não se chegue ao mínimo global, o local já costuma ser suficiente para um bom desempenho da rede.

Repare ainda que temos dois cenários aí: 1) Na parte inferior da figura, temos um custo pequeno e portanto $h_\Theta(x^{(i)}) \approx y^{(i)}$. 2) Na parte superior, o custo é grande e então $h_\Theta(x^{(i)})$ está bem longe de $y^{(i)}$.

Em ambos os casos, o algoritmo busca sempre a direção de gradiente negativo, que é “montanha-abixo” em direção ao mínimo.

5.15 Dedução do *backpropagation*

No final, o que queremos é calcular

$$\frac{\partial}{\partial \Theta_{ij}} J(\Theta).$$

Para um peso em uma camada qualquer l temos aplicando a regra da cadeia duas vezes:

$$\frac{\partial J}{\partial \Theta_{ij}^{(l)}} = \frac{\partial J}{\partial a_i^{(l+1)}} \frac{\partial a_i^{(l+1)}}{\partial \Theta_{ij}^{(l)}} = \frac{\partial J}{\partial a_i^{(l+1)}} \frac{\partial a_i^{(l+1)}}{\partial z_i^{(l+1)}} \frac{\partial z_i^{(l+1)}}{\partial \Theta_{ij}^{(l)}}. \quad (5.2)$$

Mas devemos lembrar que

$$z_i^{(l+1)} = \sum_{j=1}^{n_l} \Theta_{ij}^{(l)} a_j^{(l)},$$

de modo que

$$\frac{\partial z_i^{(l+1)}}{\partial \Theta_{ij}^{(l)}} = a_j^{(l)}. \quad (5.3)$$

Já para a derivada de $a_i^{(l+1)}$ em relação a $z_i^{(l+1)}$ temos:

$$\frac{\partial a_i^{(l+1)}}{\partial z_i^{(l+1)}} = g'(z_i^{(l+1)}). \quad (5.4)$$

Por outro lado, a derivada de J em relação a $a_i^{(l+1)}$ vai depender da camada. Na saída, supondo que tenhamos uma única unidade de ativação, temos:

$$\frac{\partial J}{\partial a_1^{(L)}}.$$

Lembrando da definição de J para um exemplo de treinamento:

$$J(\Theta) = -y \log a_1^{(L)} - (1-y) \log(1 - a_1^{(L)})$$

e portanto

$$\begin{aligned}\frac{\partial J}{\partial a_1^{(L)}} &= -y \frac{1}{a_1^{(L)}} + (1-y) \frac{1}{1-a_1^{(L)}} = \frac{-y(1-a_1^{(L)}) + (1-y)a_1^{(L)}}{a_1^{(L)}(1-a_1^{(L)})} \\ &= \frac{-y + ya_1^{(L)} + a_1^{(L)} - ya_1^{(L)}}{a_1^{(L)}(1-a_1^{(L)})}\end{aligned}$$

e finalmente

$$\frac{\partial J}{\partial a_1^{(L)}} = \frac{a_1^{(L)} - y}{a_1^{(L)}(1-a_1^{(L)})}. \quad (5.5)$$

Para as camadas anteriores:

$$\frac{\partial J}{\partial a_j^{(l)}} = \sum_{i=1}^{n_{l+1}} \left(\frac{\partial J}{\partial a_i^{(l+1)}} \frac{\partial a_i^{(l+1)}}{\partial z_i^{(l+1)}} \frac{\partial z_i^{(l+1)}}{\partial a_j^{(l)}} \right).$$

Mas lembramos que

$$z_i^{(l+1)} = \sum_{j=1}^{n_l} \Theta_{ij}^{(l)} a_j^{(l)},$$

de modo que

$$\frac{\partial J}{\partial a_j^{(l)}} = \sum_{i=1}^{n_{l+1}} \left(\frac{\partial J}{\partial a_i^{(l+1)}} \frac{\partial a_i^{(l+1)}}{\partial z_i^{(l+1)}} \Theta_{ij}^{(l)} \right) = \sum_{i=1}^{n_{l+1}} \left(\frac{\partial J}{\partial a_i^{(l+1)}} g'(z_i^{(l+1)}) \Theta_{ij}^{(l)} \right). \quad (5.6)$$

Substituindo (5.3), (5.4), (5.5) e (5.6) em (5.2):

$$\frac{\partial J}{\partial \Theta_{ij}^{(l)}} = \frac{\partial J}{\partial a_i^{(l+1)}} g'(z_i^{(l+1)}) a_j^{(l)}. \quad (5.7)$$

O termo em azul é simplesmente o que definimos como $\delta_i^{(l+1)}$:

$$\delta_i^{(l)} = \frac{\partial J}{\partial a_i^{(l)}} g'(z_i^{(l)}). \quad (5.8)$$

Na última camada, devemos lembrar que se $g(z)$ for a sigmoide, então $g'(z_1^{(L)}) = a_1^{(L)}(1-a_1^{(L)})$. Substituindo em (5.5):

$\delta_1^{(L)} = a_1^{(L)} - y.$

Já para as camadas anteriores, substituímos (5.6) em (5.8) e temos então:

$$\delta_j^{(l)} = \sum_{i=1}^{n_{l+1}} \left(\delta_i^{(l+1)} \Theta_{ij}^{(l)} \right) g'(z_j^{(l)}),$$

ou ainda, no caso em que $g(z)$ é a sigmoide:

$$\delta_j^{(l)} = \sum_{i=1}^{n_{l+1}} \left(\delta_i^{(l+1)} \Theta_{ij}^{(l)} \right) a_j^{(l)} (1 - a_j^{(l)}).$$

Por fim, voltando a (5.7):

$$\frac{\partial J}{\partial \Theta_{ij}^{(l)}} = \delta_j^{(l+1)} a_j^{(l)}.$$

Capítulo 6

Avaliação e Seleção de Modelos

6.1 O próximo passo

Imagine que estamos fazendo regressão linear para prever o preço de imóveis:

$$J(\theta) = \frac{1}{2m} \left[\sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^n \theta_j^2 \right].$$

Porém quando a função de hipótese é testada em novos imóveis que não estavam no treinamento, o erro é muito grande! O que fazer então?

Algumas possibilidades:

- Obter mais exemplos de treinamento (normalmente isso é o mais complicado!)
- Usar menos atributos ou obter mais atributos
- Adicionar atributos polinomiais (x_1^2, x_2^2, x_1x_2 , etc.)
- Aumentar ou diminuir λ

Porém testar essas soluções aleatoriamente costuma não ser efetivo, levando a grandes perdas de tempo.

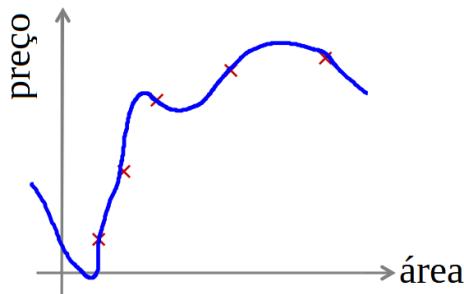
Pode-se então fazer um **diagnóstico** do algoritmo, visando entender o que está ou não funcionando e como melhorar o desempenho.

Este é um processo que pode até tomar algum tempo, mas que no final vale a pena pela economia de recursos (tempo, dinheiro, mão de obra, etc.) no total do projeto.

6.2 Avaliando uma hipótese

Já vimos que uma função de hipótese complexa pode se ajustar perfeitamente aos dados de treinamento, mas não generalizar bem para dados fora dele. Isso é o *overfitting* (Figura 6.1).

Na Figura 6.1 isso é fácil de ver porque só temos um atributo, mas e se tivermos centenas ou milhares, como é usual no mundo real? Como então detectar problemas desse tipo?



$$h_{\theta}(x) = \theta_0 + \theta_1 x + \theta_2 x^2 \\ + \theta_3 x^3 + \theta_4 x^4$$

Figura 6.1: *Overfitting*.

A forma mais popular para se avaliar uma hipótese é dividindo o conjunto de exemplos em dois subconjuntos: de treino e de teste (Figura 6.2).

Uma escolha comum é 70% para treino e 30% para teste. É importante que as amostras sejam embaralhadas aleatoriamente para se evitar qualquer tipo de vício na escolha desses conjuntos.

O procedimento na regressão linear é o seguinte:

- Aprender θ minimizando $J(\theta)$ usando os dados do conjunto de treino
- Calcular o erro de teste:

$$J_{teste}(\theta) = \frac{1}{2m_{teste}} \sum_{i=1}^{m_{teste}} (h_{\theta}(x_{teste}^{(i)}) - y_{teste}^{(i)})^2$$

O caso da regressão logística é parecido, mudando apenas o cálculo do erro:

- Aprender o parâmetro θ a partir do treinamento
- Calcular o erro de teste:

$$J_{teste}(\theta) = -\frac{1}{m_{teste}} \sum_{i=1}^{m_{teste}} y_{teste}^{(i)} \log h_{\theta}(x_{teste}^{(i)}) + (1 - y_{teste}^{(i)}) \log (1 - h_{\theta}(x_{teste}^{(i)}))$$

6.3. SELEÇÃO DE MODELO: CONJUNTOS DE TREINO, VALIDAÇÃO E TESTE 81

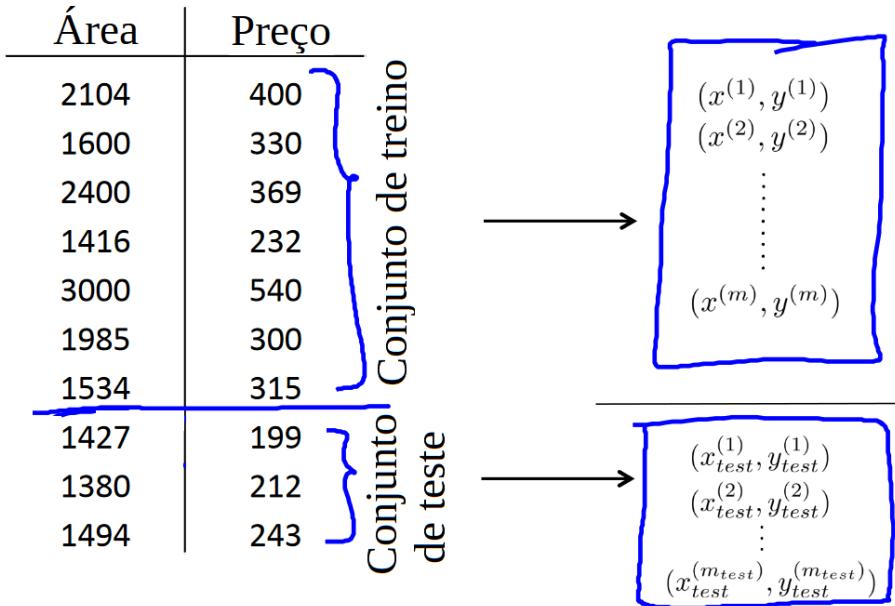


Figura 6.2: Treino/teste.

- Aqui também uma alternativa popular é o erro de classificação (erro 0/1):

$$J_{teste}(\theta) = \frac{1}{m_{teste}} \sum_{i=1}^{m_{teste}} err(h_\theta(x_{teste}^{(i)}), y_{teste}^{(i)}),$$

em que

$$err(h_{teste}(x), y) = \begin{cases} 1 & \text{se } h_\theta(x) \geq 0.5 \text{ e } y = 0 \\ & \text{ou } h_\theta(x) < 0.5 \text{ e } y = 1 \\ 0 & \text{caso contrário,} \end{cases}$$

ou seja, ele soma 1 sempre que há um erro na classificação. O limiar aqui poderia ser diferente de 0.5. Temos assim a proporção do dado de treino com classificação errada.

6.3 Seleção de Modelo: Conjuntos de Treino, Validação e Teste

Imagine que precisamos decidir o grau de um polinômio a ser ajustado a um dado. Ou ainda qual λ usar na regularização. Isso é chamado de **seleção de modelo**.

Já vimos que um erro baixo no treino não garante boa generalização devido ao *overfitting*. Vimos também que uma solução para avaliar essa generalização é com um conjunto de teste.

Mas suponha agora que tenhamos várias hipóteses e queremos saber qual generaliza melhor. Por exemplo, polinômios até ordem 10:

- 1. $h_\theta(x) = \theta_0 + \theta_1 x$
- 2. $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2$
- 3. $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3$
- ⋮
- 10. $h_\theta(x) = \theta_0 + \theta_1 x + \cdots + \theta_{10} x^{10}$

Precisamos definir então um novo parâmetro d = grau do polinômio.

Vamos chamar os parâmetros do polinômio de grau 1,2, etc. de $\Theta^{(1)}$, $\Theta^{(2)}$, etc. Podíamos pensar então em usar d que gere o menor valor entre os erros no teste, i.e., $J_{teste}(\Theta^{(1)})$, $J_{teste}(\Theta^{(2)})$, etc.

Suponha então que assim descobrimos que o melhor modelo é com $d = 5$:

$$\theta_0 + \cdots + \theta_5 x^5.$$

E medimos então a generalização pelo erro no teste $J_{teste}(\Theta^{(5)})$.

PROBLEMA: esse erro é muito provavelmente otimista demais, já que nosso parâmetro extra d foi ajustado exatamente no conjunto de teste.

A solução é criarmos um terceiro conjunto para a seleção de modelo, chamado de **conjunto de validação** ou de **validação cruzada** (Figura 6.3). É usual que tenhamos 60% dos exemplos no treino, 20% na validação e 20% no teste. Novamente é conveniente que as amostras sejam embaralhadas antes da divisão.

Agora temos então 3 erros:

- Erro de treino:

$$J_{treino}(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2$$

- Erro de validação:

$$J_{cv}(\theta) = \frac{1}{2m_{cv}} \sum_{i=1}^{m_{cv}} (h_\theta(x_{cv}^{(i)}) - y_{cv}^{(i)})^2$$

- Erro de teste:

$$J_{teste}(\theta) = \frac{1}{2m_{teste}} \sum_{i=1}^{m_{teste}} (h_\theta(x_{teste}^{(i)}) - y_{teste}^{(i)})^2$$

O processo geral de escolha de modelo segue então os seguintes passos:

1. Otimizar parâmetros Θ no conjunto de treino para cada grau.

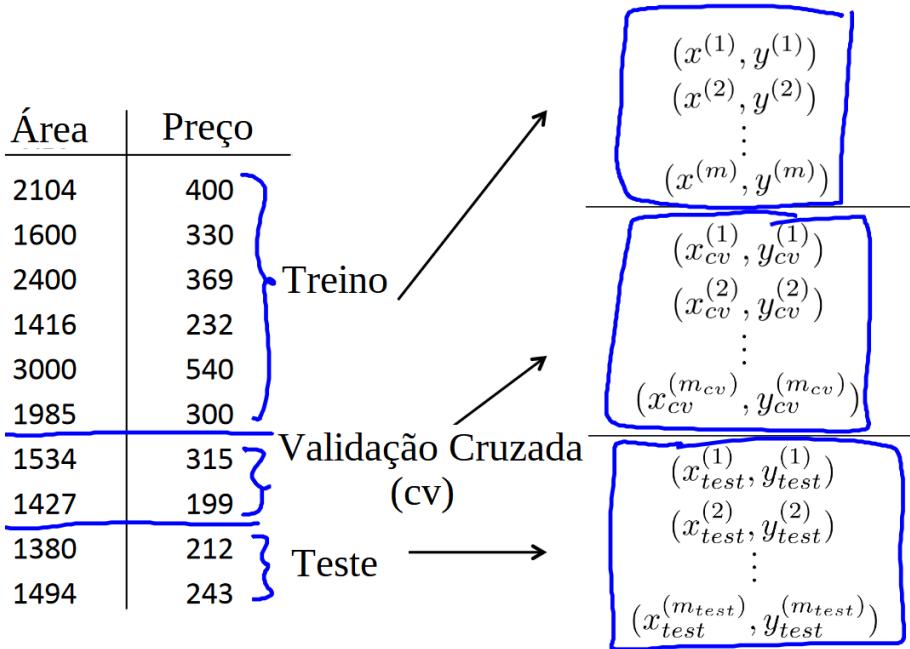


Figura 6.3: Treino/validação/teste.

2. Escolhemos então o modelo cujo grau d propicia o menor erro de validação.
3. Estimar o erro de generalização do modelo escolhido usando $J_{teste}(\Theta^{(d)})$, em que d foi obtido no passo anterior (menor erro de validação).

É bem comum, infelizmente, que as pessoas usem um mesmo conjunto para teste e validação. Isso até pode não ser tão grave se seu conjunto de teste for muito grande, mas o ideal é **sempre** ter os 3 conjuntos separados: treino, validação e teste.

6.4 Diagnosticando Viés vs Variância

Como vimos, existem basicamente 3 cenários que ocorrem no ajuste dos dados por um modelo de aprendizado de máquinas:

- O modelo pode ser muito simples (p.ex. uma reta) e não se ajustar bem aos dados, causando *underfitting*. Este é o problema de alto **viés**.
- O modelo pode ser excessivamente complexo (p.ex. um polinômio de grau muito alto), ajustar-se perfeitamente ao dado de treino, mas não generalizar bem no teste/validação. Este é o problema da alta **variância** (*overfitting*).

- Por fim, temos o cenário ideal em que o ajuste se dá na medida certa e generaliza bem.

Os 3 casos estão ilustrados na Figura 6.4.

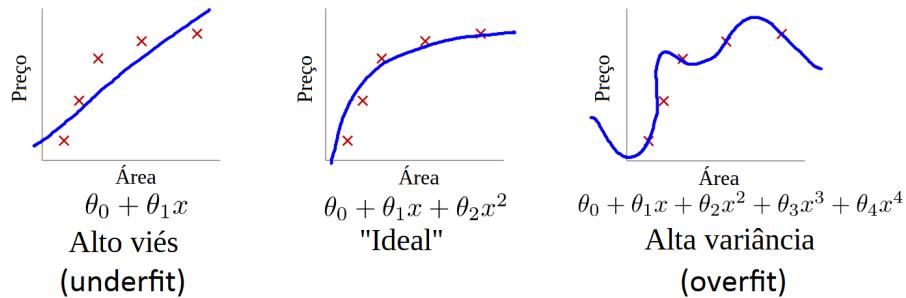


Figura 6.4: *Underfit*, ajuste correto e *overfit*.

Para identificarmos qual problema está prevalecendo quando o erro de nosso algoritmo está alto, podemos comparar o erro de treino com o de validação (ou de teste).

Lembrando:

- Erro de treino: $J_{treino}(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2$
- Erro de validação: $J_{cv}(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_\theta(x_{cv}^{(i)}) - y_{cv}^{(i)})^2$

A Figura 6.5 mostra a curva dos dois erros quando o grau do polinômio varia.

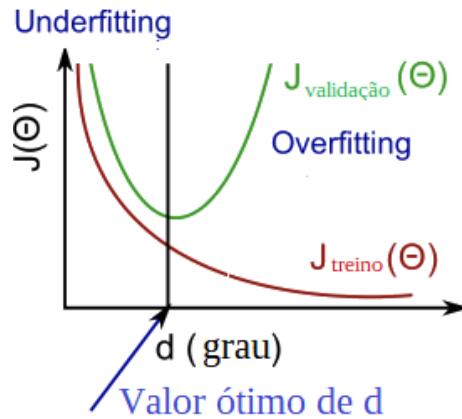


Figura 6.5: Alto viés vs alta variância.

O erro do treino vai ficando cada vez menor à medida que o grau aumenta, já que o ajuste vai ficando mais e mais preciso, chegando a zero após um certo valor.

Já o erro de validação começa também alto para grau pequeno (maior que o erro no treino), porém após um certo valor ótimo do grau d ele começa a aumentar novamente, já que o ajuste excessivo diminui a generalização (*overfitting*).

A partir dessa figura, podemos traçar então os cenários que correspondem a viés alto e variância alta.

No viés alto (*underfit*) temos:

- $J_{treino}(\theta)$ alto
- $J_{cv}(\theta) \approx J_{treino}(\theta)$

enquanto que na variância alta (*overfit*) temos

- $J_{treino}(\theta)$ baixo
- $J_{cv}(\theta) \gg J_{treino}(\theta)$.

6.5 Viés/Variância e Regularização

A Figura 6.6 ilustra o efeito da regularização em um modelo de regressão com polinômio de 4º grau:

$$h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$$

com uma função de custo regularizada

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2.$$

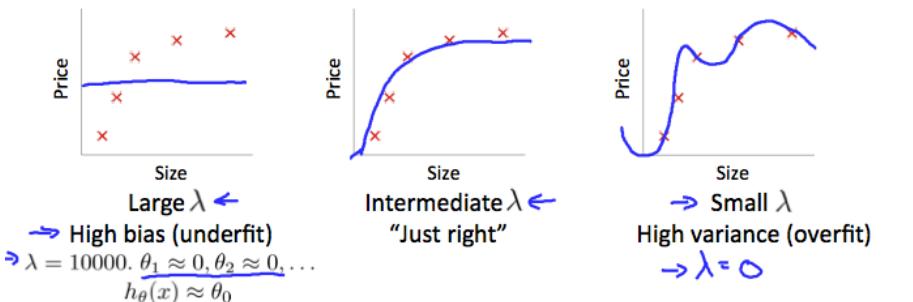


Figura 6.6: Viés/variância e regularização.

Um λ grande penaliza fortemente os parâmetros θ_j fazendo com que estes tendam a 0, sobrando apenas θ_0 . O modelo se torna então uma reta e temos *underfit* (esquerda na Figura 6.6).

Já $\lambda = 0$ é a ausência de regularização, de modo que o polinômio se ajusta perfeitamente ao treino e temos *overfit* (direita na Figura 6.6).

Finalmente, temos um valor intermediário para λ que é ótimo para a generalização (centro na Figura 6.6).

6.5.1 Escolha de λ

A escolha do parâmetro λ usa a estratégia de seleção de modelos que vimos.

Dadas a função de hipótese e de custo:

$$h_{\theta}(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$$

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2.$$

Primeiramente, vamos redefinir os erros $J_{treino}(\theta)$, $J_{cv}(\theta)$ e $J_{teste}(\theta)$ de um modo ligeiramente diferente, excluindo a regularização, i.e.:

$$J_{treino}(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2$$

$$J_{cv}(\theta) = \frac{1}{2m_{cv}} \sum_{i=1}^m (h_{\theta}(x_{cv}^{(i)}) - y_{cv}^{(i)})^2$$

$$J_{teste}(\theta) = \frac{1}{2m_{teste}} \sum_{i=1}^m (h_{\theta}(x_{teste}^{(i)}) - y_{teste}^{(i)})^2$$

Repare como antes as expressões desses erros eram idênticas a $J(\theta)$, mas agora isso não ocorre mais por descartarmos o termo regularizador.

Em seguida, iteramos sobre um intervalo de valores para λ . Uma estratégia usual é usar o seguinte, multiplicando por 2 em cada iteração:

- 1. $\lambda = 0$
- 2. $\lambda = 0.01$
- 3. $\lambda = 0.02$
- 4. $\lambda = 0.04$
- 5. $\lambda = 0.08$
- \vdots
- 12. $\lambda = 10$ (ou $\lambda = 10.24$ para ser exato!)

Para cada um desses valores, obtemos os parâmetros Θ como usual:

$$\underset{\Theta}{\text{minimize}} J(\Theta)$$

e calculamos o erro de validação $J_{cv}(\Theta)$ correspondente.

IMPORTANTE: Note então como o erro de validação que nos servirá de guia é **SEM** regularização.

Escolhe-se então o vetor de parâmetros Θ cujo λ resultou no menor erro de validação.

Por fim, para estimarmos o erro de teste, aplicamos esses mesmos parâmetros em $J_{teste}(\Theta)$.

6.5.2 Influência da regularização no erro de treino e validação

A Figura 6.7 mostra a influência da regularização no erro de treino e validação. Apenas lembrando que esses erros não incluem a regularização:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2$$

$$J_{treino}(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2$$

$$J_{cv}(\theta) = \frac{1}{2m_{cv}} \sum_{i=1}^m (h_\theta(x_{cv}^{(i)}) - y_{cv}^{(i)})^2$$

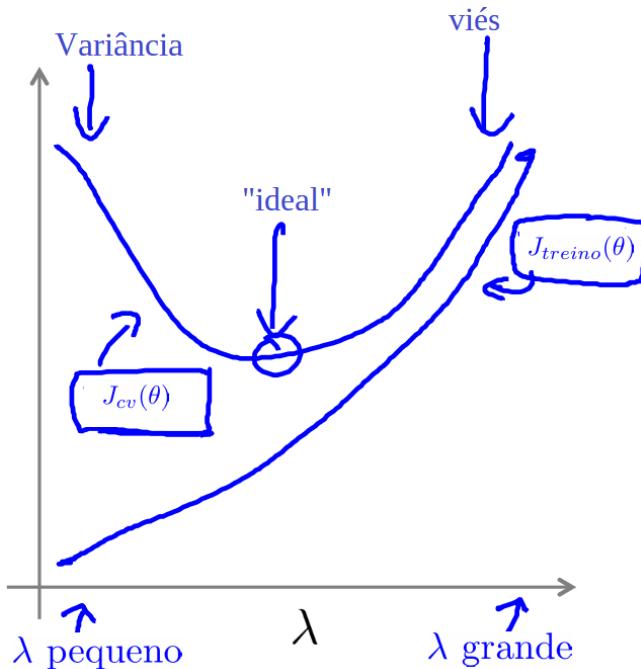


Figura 6.7: Erro de treino e validação quando se varia a regularização.

O erro de treino é pequeno quando $\lambda \rightarrow 0$ pois o polinômio se ajusta perfeitamente (região de alta variância). E vai aumentando à medida que λ aumenta e causa *underfit* (alto viés).

Já o erro de validação é alto para λ pequeno, devido ao *overfit* (alta variância), decai até um valor ótimo de λ e a partir dali começa a subir novamente, agora por *underfit* (alto viés).

6.6 Curvas de Aprendizado

Uma forma de se detectar problemas de alto viés ou alta variância é por meio das curvas de aprendizado.

Este é um gráfico do erro de treino e do erro de validação (ou teste) em função do número m de amostras de treino. Lembrando:

$$J_{treino}(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2$$

$$J_{cv}(\theta) = \frac{1}{2m_{cv}} \sum_{i=1}^{m_{cv}} (h_\theta(x_{cv}^{(i)}) - y_{cv}^{(i)})^2$$

Rpare na Figura 6.8 que uma função de hipótese quadrática pode se ajustar perfeitamente para uma amostra, duas ou três. O erro de treino no caso é zero. Mesmo com regularização, a aproximação ainda é boa. Porém, se m vai aumentando, a aproximação já passa a não ser tão boa e o erro no treino começa a aumentar.

Já o erro na validação (ou teste) começa alto pois uma amostra não é suficiente para a obtenção de uma $h_\theta(x)$ precisa o suficiente para a generalização e vai diminuindo com mais amostras.

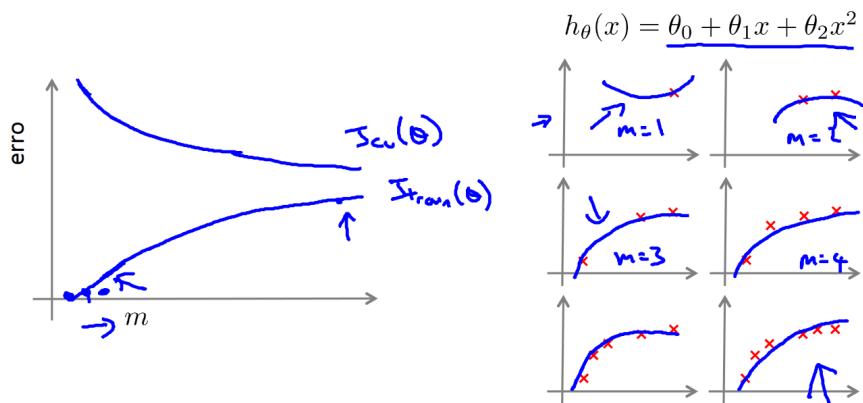


Figura 6.8: Curvas de aprendizado.

Vejamos agora na Figura 2.8 um cenário de alto viés, com uma função de hipótese linear (*underfit*).

Com poucos dados, a tendência é que a reta se ajuste bem e o erro de treino tende a zero, porém, pelo fato de o modelo ser muito simples, esse erro cresce rapidamente até atingir um *plateau* com erro alto, em que o erro não aumenta mais porque as amostras passam a ter um comportamento geral parecido (ver parte superior e inferior à direita na Figura 6.9).

Já o erro na validação (ou teste) começa alto devido ao *underfit* e diminui com mais dados, porém pela simplicidade do modelo atinge também um *plateau* de erro acima do desejado, a partir do qual não melhora.

O erro de validação se aproxima do erro de treino rapidamente, ainda com m pequeno. Neste cenário, obter mais dados (por si só) não vai ajudar muito.

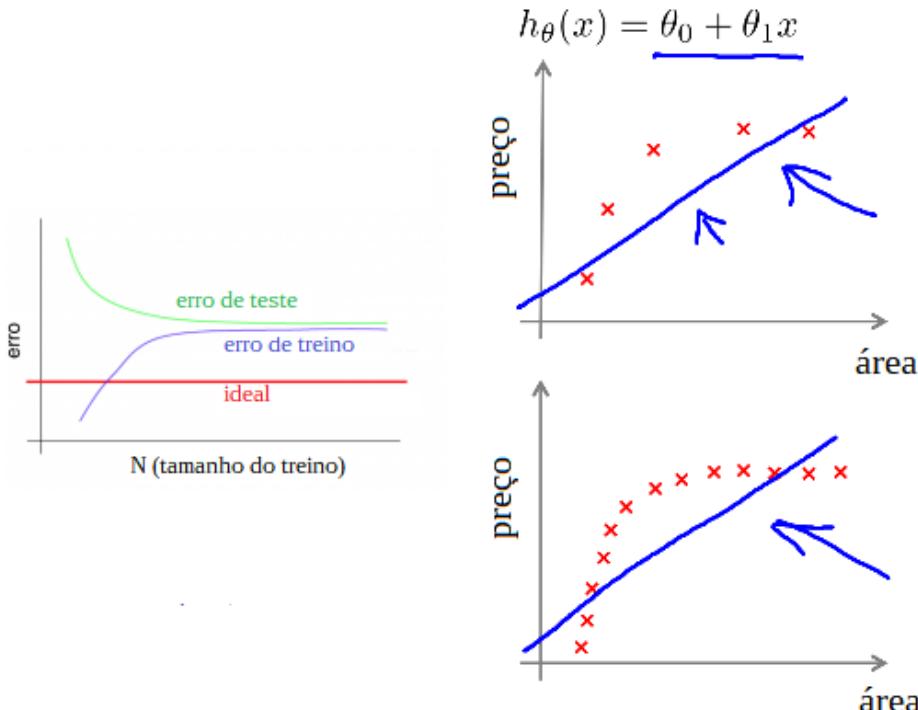


Figura 6.9: Curvas de aprendizado com alto viés.

Por fim, temos na Figura 6.10 o cenário de alta variância.

Aqui, o erro de treino é sempre baixo já que uma $h_\theta(x)$ se ajusta bem mesmo com m grande. Ainda assim, o aumento de m causa um pequeno aumento no erro (já que o ajuste perfeito vai começar a ficar comprometido a partir de algum ponto).

Já o erro da validação (ou teste) começa alto pela existência de poucos dados para generalizar e vai diminuindo com o aumento de m , porém diminui pouco devido ao *overfit*. O que caracteriza esse cenário é um grande **gap** entre a curva de erro do treino e da validação.

Porém, aqui nota-se que ao extrapolarmos as curvas de erro, tanto o erro de treino segue aumentando com m crescendo quanto o erro de validação segue diminuindo, de modo que eles tendem a se encontrar com m suficientemente grande. Isso faz com que, neste caso, a obtenção de mais dados de treinamento costume ajudar a melhorar o desempenho do algoritmo.

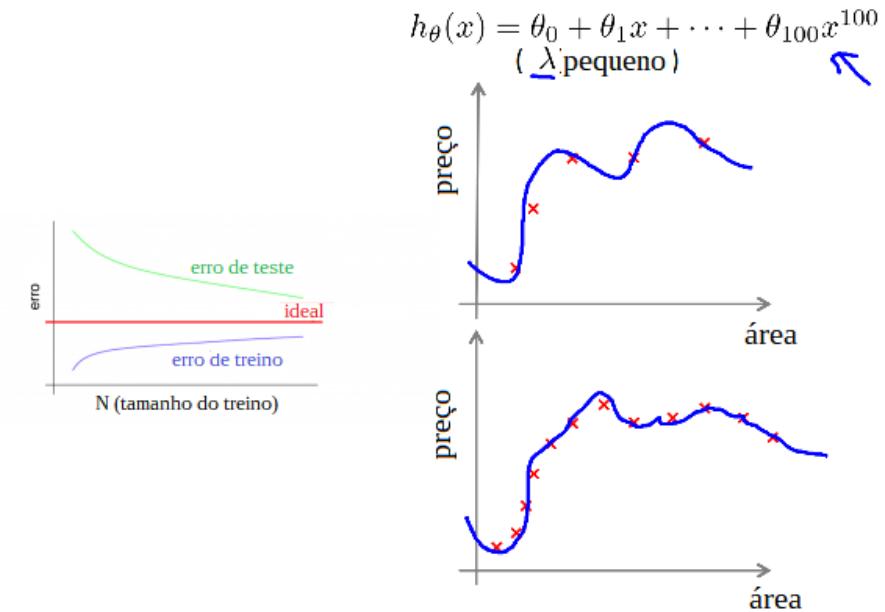


Figura 6.10: Curvas de aprendizado com alta variância.

6.7 O Próximo Passo Revisitado

Já vimos que quando o erro da hipótese no teste está alto, sugere-se tentarmos os passos seguintes. Veremos agora como isso se relaciona com viés e variância:

- Obter mais exemplos de treinamento → corrige variância alta
- Usar menos atributos → corrige variância alta
- Usar mais atributos → corrige viés alto
- Adicionar atributos polinomiais (x_1^2, x_2^2, x_1x_2 , etc.) → corrige viés alto
- Aumentar λ → corrige viés alto
- Diminuir λ → corrige variância alta

Em geral, modelos de complexidade muito baixa (como um polinômio de grau muito baixo), tendem a possuir alto viés e baixa variância, de modo que o modelo tem desempenho ruim consistentemente.

Já modelos de complexidade muito alta (como polinômios de grau muito alto) se ajustam muito bem ao treino, mas pessimamente ao teste (viés baixo, mas variância alta).

O ideal então é um ponto de equilíbrio em que o modelo se ajusta bem ao treino e também generaliza bem.

6.7.1 Redes neurais e *overfitting*

No caso das redes neurais, temos que escolher entre uma rede pequena (uma camada escondida, poucos neurônios), que é barata computacionalmente, mas propensa a *underfitting*, ou uma rede grande (muitos neurônios e/ou muitas camadas), que é mais cara computacionalmente e propensa a *overfitting* (Figura 6.11).

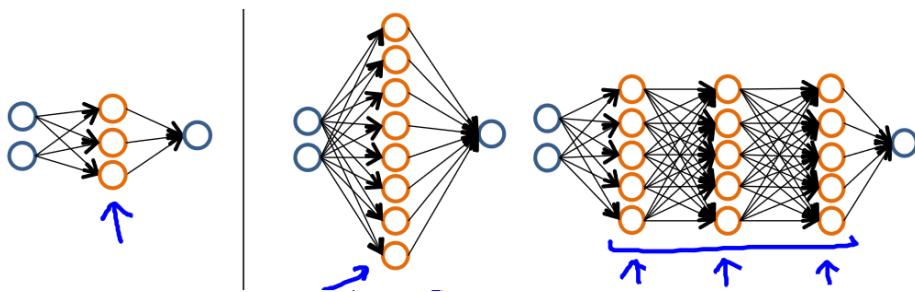


Figura 6.11: Redes neurais. À esquerda uma rede pequena, no centro uma rede grande com muitos neurônios, à direita uma rede grande com muitas camadas.

A questão do *overfitting* pode ser resolvida com regularização.

Em geral, uma rede grande com regularização para prevenir *overfitting* costuma funcionar melhor do que uma rede pequena.

O número de camadas pode ser definido pela validação cruzada. Parte-se de uma camada escondida, vai aumentando e avaliando $J_{cv}(\theta)$. Adota-se então o número de camadas que minimiza o erro na validação.

6.8 Prioridade no Desenvolvimento de um Sistema de Aprendizado de Máquinas

Considere os emails da Figura 6.12, que devemos classificar como *spam* (Classe 1) ou não-*spam* (Classe 0).

Precisamos definir os atributos relevantes do email (x): 100 palavras indicativas de *spam*/não-*spam* e

$$x_j = \begin{cases} 1 & \text{se a palavra } j \text{ aparece no email} \\ 0 & \text{caso contrário.} \end{cases}$$

Tome como exemplo o *spam* da Figura 6.13. Um possível vetor de atributos seria

<pre> From: cheapsales@buystufffromme.com To: ang@cs.stanford.edu Subject: Buy now! Deal of the week! Buy now! Rolex <u>w4tches</u> - \$100 Med<u>1cine</u> (any kind) - \$50 Also low cost <u>M0rtgages</u> available. </pre> <p style="text-align: center;"><i>Spam (1)</i></p>	<pre> From: Alfred Ng To: ang@cs.stanford.edu Subject: Christmas dates? Hey Andrew, Was talking to Mom about plans for Xmas. When do you get off work. Meet Dec 22? Alf </pre> <p style="text-align: center;"><i>Non-spam (0)</i></p>
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Figura 6.12: *Spam vs. Não-Spam.*

```

From: cheapsales@buystufffromme.com
To: ang@cs.stanford.edu
Subject: Buy now!

Deal of the week! Buy now!

```

Figura 6.13: Exemplo de *spam*.

$$x \in \mathbb{R}^{100} = \begin{bmatrix} \text{andrew} \\ \text{buy} \\ \text{deal} \\ \text{discount} \\ \vdots \\ \text{now} \\ \vdots \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ \vdots \\ 1 \\ \vdots \end{bmatrix}$$

Na prática, são escolhidas as palavras mais frequentes no treino (aparecendo de 10 mil a 50 mil vezes).

Como usar seu tempo para reduzir o erro?

- Coletar mais dados, p.ex., projeto “honeypot” (gera endereços falsos para atrair *spammers*). Como já vimos, isso nem sempre vai ajudar.
- Desenvolver atributos sofisticados com base no roteamento do email (cabeçalho).
- Desenvolver atributos sofisticados com base no corpo do email, p.ex., tratar “discount”/“discounts” ou “deal”/“Dealer” como a mesma palavra, analisar pontuação, etc.
- Desenvolver algoritmos sofisticados de pré-processamento, p.ex., detectando erros de ortografia (m0rtgage, med1cine, w4tches, etc.). Os *spammers* costumam “errar” isso de propósito exatamente para o algoritmo não pegar.

É difícil saber *a priori* quais alternativas são mais promissoras.

6.9 Análise de Erro

Recomenda-se

- Começar por um método simples de implementação rápida e testá-lo no conjunto de validação.
- Fazer curvas de aprendizado para ver se mais dados, atributos, etc. ajudam
- Análise de erro: Examinar manualmente os exemplos no conjunto de validação para os quais o algoritmo erra. Tentar identificar alguma tendência nesses erros para, por exemplo, adicionar um atributo que trate aqueles casos.

6.9.1 Exemplo de análise de erro

Considere $m_{cv} = 500$ exemplos de validação.

O algoritmo classifica erradamente 100 emails.

Examinamos manualmente os 100 casos de erro e categorizamos com base em

1. Tipo de email, ex.: medicamento, réplica, roubo de senha, outros.
2. Dicas (atributos) que você acredita que ajudariam o algoritmo a classificá-los corretamente

Exemplo de respostas:

Tipo	Atributo
Medicamento: 12	Erros deliberados de ortografia: 5
Réplica/ <i>fake</i> : 4	Rota incomum: 16
Roubo de senha: 53	Pontuação incomum (típica de <i>spam</i> , p.ex., muitas exclamações): 32
Outros: 31	:

Nota-se no caso que *spams* de roubo de senha e com pontuação incomum são os mais frequentes entre os classificados erradamente. Então concentra-se em novos atributos, algoritmos, etc. para tratar especificamente desses casos.

6.9.2 Importância da avaliação numérica

Uma medida numérica única (número real) do desempenho do algoritmo é fundamental.

EX.: Deveriam palavras como *discount/discounts/discounted/discounting* ser tratadas como uma só?

Podemos usar um programa que identifique radicais da palavra, p.ex., “Porter stemmer”. Problema: pares como *universe/university* teriam o mesmo radical.

A análise de erro pode não ser útil para decidir se isso melhoraria o desempenho. Única solução é tentar e ver se funciona.

Para isso, precisamos de uma medida numérica de avaliação, p.ex., erro de validação com e sem o uso de radicais.

EX.: sem radicais: 5% de erro, com radicais: 3% de erro. Melhor portanto é usar os radicais!

Por outro lado, suponha que distinguir maiúscula de minúscula leva a um erro de 3.2%. Como este erro é maior que os 3% sem a distinção, então é melhor sem.

6.10 Métricas de Erro para Classes Desbalanceadas

Imagine que treinemos uma regressão logística $h_\theta(x)$ para diagnosticar câncer ($y = 1$) ou sem câncer ($y = 0$).

E que obtivemos 1% de erro no teste (99% de acerto). Excelente resultado, não???

Porém, só 0.5% dos pacientes têm câncer! Temos **classes desbalanceadas** (*skewed*).

Nosso resultado não é mais interessante! A função trivial abaixo teria 0.5% de erro!

```
function y = predictCancer(x)
    y = 0; % ignore x!
    return
```

6.10.1 Precision / Recall

Neste caso então introduzimos os conceitos de *precision* e *recall*.

Vamos convencionar que $y = 1$ na presença da classe rara que queremos detectar, no caso, a presença de câncer.

Os cenários da Figura 6.14 são possíveis. O paciente pode ter câncer e o algoritmo confirmar (verdadeiro positivo), não ter câncer e o algoritmo falar que tem (falso positivo), ter câncer e o algoritmo falar que não tem (falso negativo) e não ter câncer e o algoritmo falar que não tem (verdadeiro negativo).

Definimos então:

- **Precision:** De todos os pacientes para os quais o algoritmo retornou $y = 1$, qual proporção **realmente** tem câncer?

$$\frac{\text{Número de verdadeiros positivos}}{\text{Número de previsões positivas}} = \frac{\text{Número de verdadeiros positivos}}{\text{Número de verdadeiros positivos} + \text{número de falsos positivos}}$$

		Actual class	
		1	0
Predicted class	1	True positive	False positive
	0	False negative	True negative

Figura 6.14: Possibilidades em um problema de classificação binário.

- **Recall:** De todos os pacientes que realmente têm câncer, qual proporção o algoritmo **detectou** como tendo câncer?

$$\frac{\text{Número de verdadeiros positivos}}{\text{Número real de positivos}} = \frac{\text{Número de verdadeiros positivos}}{\text{Número de verdadeiros positivos} + \text{número de falsos negativos}}$$

Ou seja, na *precision* o denominador é a soma da primeira **linha** da tabela (casos previstos como da classe 1), no *recall*, é a soma da primeira **coluna** (casos que realmente são da classe 1).

Note que agora, se o classificador prevê $y = 0$ o tempo todo (ninguém tem câncer), tanto o *recall* quanto o *precision* serão 0, pois não haverá verdadeiro positivo.

6.11 Trade off de Precision e Recall

Considere a regressão logística: $0 \leq h_\theta(x) \leq 1$. Prevemos $y = 1$ se $h_\theta(x) \geq 0.5$ e $y = 0$ se $h_\theta(x) < 0.5$. Note que temos um *threshold* (limiar, ponto de corte) de 0.5.

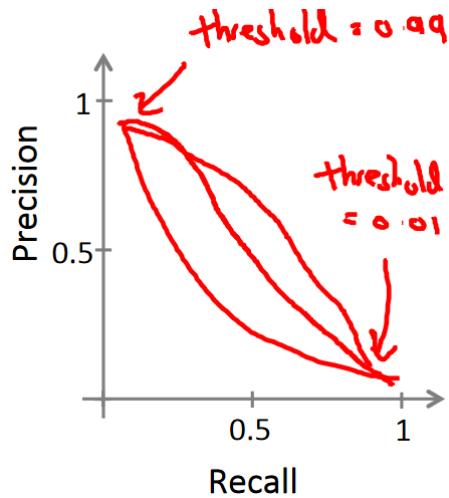
Agora suponha que queiramos prever $y = 1$ (câncer) apenas se tivermos muita certeza. Para isso, subimos nosso *threshold* para 0.7 ou 0.9 e só prevemos câncer com probabilidade bem alta.

Com isso temos **precision mais alto** pois nossas previsões de câncer tendem a ser mais corretas, porém **recall mais baixo** pois vamos detectar menos pacientes que na realidade tenham câncer (pois nosso algoritmo falará que não têm).

Já se nosso objetivo for evitar de passar batido um caso de câncer (falso negativo), devemos baixar o *threshold* para 0.3, por exemplo.

Agora temos **recall mais alto** pois quase todos os casos de câncer serão detectados, porém **precision mais baixo** pois muitos desses casos apontados pelo algoritmo serão na verdade falsos.

Temos aí portanto um *trade off*. Para uma visão mais completa, podemos fazer uma curva variando o *threshold* e prevendo $y = 1$ se $h_\theta(x) > \text{threshold}$ (Figura 6.15).

Figura 6.15: Curva de *precision/recall*.

6.11.1 F_1 score (F score)

Como comparar valores de *precision/recall* usando uma única medida, como vimos na análise de erro?

Imagine o seguinte exemplo:

	<i>Precision</i> (P)	<i>Recall</i> (R)	Média	F_1 score
Algoritmo 1	0.5	0.4	0.45	0.44
Algoritmo 2	0.7	0.1	0.4	0.18
Algoritmo 3	0.02	1.0	0.51	0.04

Uma possibilidade seria a média:

$$\frac{P + R}{2}$$

Note que um algoritmo que prevê $y = 1$ o tempo todo teria os valores de *precision* e *recall* do Algoritmo 3 e seria péssimo! Porém, a média seria maior que a do Algoritmo 1, que seria muito mais interessante na vida real. Essa portanto **não** é uma medida válida.

Uma solução é o F_1 score:

$$2 \frac{PR}{P + R}$$

Note que agora, se $P \approx 0$ OU $R \approx 0$, o score é pequeno e este só fica maior se AMBOS forem relativamente altos.

Essa medida também serve para definir um *threshold* ótimo, como sendo aquele que fornece maior F_1 score no conjunto de validação.

6.12 Usando Grandes Conjuntos de Dados

A ideia de que o uso de grandes quantidades de dados no treinamento é mais importante do que o algoritmo em si vem do trabalho [Banko & Brill, 2001].

Eles compararam algoritmos populares na época no problema de identificar entre palavras similares qual deveria completar uma sentença (Figura 6.16).

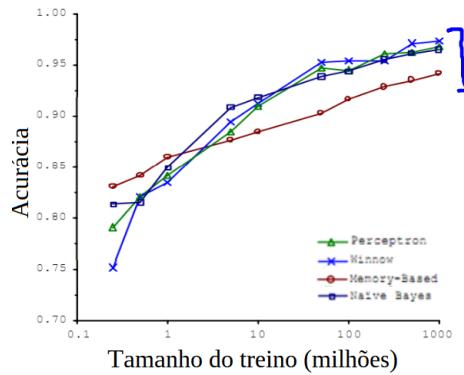


Figura 6.16: Acurácia em função do tamanho do conjunto de treinamento.

Nota-se que os algoritmos têm desempenho parecido e a acurácia sempre aumenta com mais dados no treinamento: “O vencedor não é o melhor algoritmo, mas sim quem tem mais dados!”.

Porém, para que isso seja verdade, precisamos de duas condições:

1. Os atributos devem ser suficientes para prever y corretamente. Exemplo: as palavras em volta na sentença são suficientes para prever qual palavra deveria preencher. Contra-exemplo: só a área de uma casa não é suficiente para prever seu preço. Teste útil: um especialista humano poderia prever y com base apenas em x ?
2. O algoritmo de aprendizado deve ter muitos parâmetros, p.ex., regressão linear/logística com muitos atributos, rede neural com muitas unidades escondidas, etc. Assim teremos $J_{treino}(\theta)$ pequeno (viés baixo) e o uso de um conjunto de treinamento muito grande provavelmente não causará *overfit* (variância baixa), de modo que $J_{treino}(\theta) \approx J_{teste}(\theta)$ e, portanto, $J_{teste}(\theta)$ será pequeno, como desejado!

Capítulo 7

Máquina de Vetores de Suporte

7.1 Máquina de Vetores de Suporte (SVM)

Lembrando da regressão logística:

$$h_{\theta}(x) = \frac{1}{1 + e^{-\theta^T x}}.$$

Vamos considerar em função de $z = \theta^T x$ e observar o gráfico na Figura 7.1.

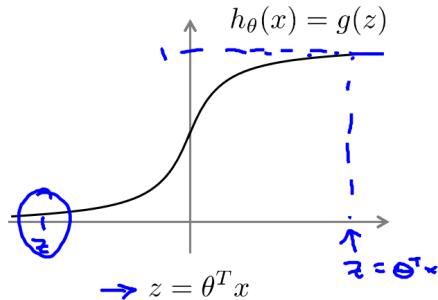


Figura 7.1: Regressão logística.

Note que:

- Se $y = 1$, queremos $h_{\theta}(x) \approx 1$, $\theta^T x \gg 0$
- Se $y = 0$, queremos $h_{\theta}(x) \approx 0$, $\theta^T x \ll 0$

O custo de um exemplo na regressão logística é

$$-(y \log h_{\theta}(x) + (1 - y) \log(1 - h_{\theta}(x)))$$

$$= -y \log \frac{1}{1 + e^{-\theta^T x}} - (1 - y) \log \left(1 - \frac{1}{1 + e^{-\theta^T x}} \right)$$

Substituindo $y = 1$ ($z = \theta^T x \gg 0$) na expressão acima, temos

$$-\log \frac{1}{1 + e^{-z}}$$

como na Figura 2.1 esquerda.

A ideia de SVM é definir um custo que aproxima essa curva por uma função linear por partes. Para $y = 1$ chamamos de $custo_1(z)$ e $custo_1(z) = 0$ para $z \geq 1$ e é uma reta decrescente para $z \leq 1$ (não importa o coeficiente angular).

Já para $y = 0$ ($\theta^T x \ll 0$), temos

$$-\log \left(1 - \frac{1}{1 + e^{-z}} \right)$$

como na Figura 7.2 direita.

Neste caso, então, temos para o SVM um custo $custo_0(z) = 0$ para $z \leq -1$ e uma reta crescente para $z \geq -1$.

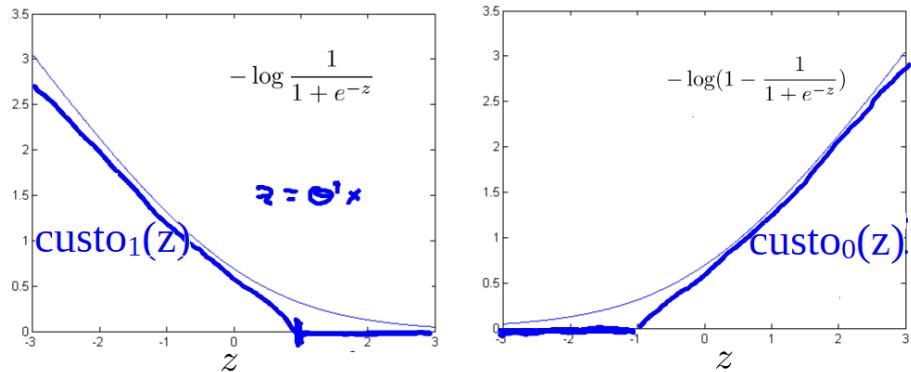


Figura 7.2: Função de custo na regressão logística e SVM para $y = 1$ (esquerda) e $y = 0$ (direita).

7.1.1 SVM

Veja abaixo a função de custo da regressão logística apenas com um pequeno ajuste que leva o sinal negativo para dentro do somatório:

$$\underset{\theta}{\text{minimize}} \frac{1}{m} \left[\sum_{i=1}^m y^{(i)} \left(-\log h_{\theta}(x^{(i)}) \right) + (1 - y^{(i)}) \left(-\log(1 - h_{\theta}(x^{(i)})) \right) \right] + \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2$$

A ideia do SVM então é substituir os termos em vermelho e azul pela aproximação linear por partes na Figura 2.1:

$$\underset{\theta}{\text{minimize}} \frac{1}{m} \sum_{i=1}^m y^{(i)} custo_1(\theta^T x^{(i)}) + (1 - y^{(i)}) custo_0(\theta^T x^{(i)}) + \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2$$

Podemos ainda cortar o denominador m pois não interfere na minimização:

$$\underset{\theta}{\text{minimize}} \sum_{i=1}^m y^{(i)} \text{custo}_1(\theta^T x) + (1 - y^{(i)}) \text{custo}_0(\theta^T x^{(i)}) + \frac{\lambda}{2} \sum_{j=1}^n \theta_j^2$$

Outra convenção que é adotada no SVM é que o termo regularizador fica em outra posição. Enquanto na regressão logística temos uma expressão do tipo $A + \lambda B$, no SVM temos $CA + B$, em que $C = \frac{1}{\lambda}$.

Temos finalmente então:

$$\underset{\theta}{\text{minimize}} C \sum_{i=1}^m \left[y^{(i)} \text{custo}_1(\theta^T x^{(i)}) + (1 - y^{(i)}) \text{custo}_0(\theta^T x^{(i)}) \right] + \frac{1}{2} \sum_{j=1}^n \theta_j^2$$

O comportamento da função de hipótese no SVM também é diferente:

$$h_{\theta}(x) = \begin{cases} 1 & \text{se } \theta^T x \geq 0 \\ 0 & \text{caso contrário.} \end{cases}$$

7.2 SVM - Intuição da Margem Larga

Diz-se que o SVM usa margem larga porque para $y = 1$ não queremos apenas $\theta^T x \geq 0$ como na regressão logística, mas sim $\theta^T x \geq 1$, assim como para $y = 0$ não queremos apenas $\theta^T x < 0$, mas sim $\theta^T x \leq -1$.

Agora vamos supor que C seja muito grande, p.ex. $C = 100000$. Lembrando do custo:

$$\underset{\theta}{\text{minimize}} C \sum_{i=1}^m \left[y^{(i)} \text{custo}_1(\theta^T x^{(i)}) + (1 - y^{(i)}) \text{custo}_0(\theta^T x^{(i)}) \right] + \frac{1}{2} \sum_{j=1}^n \theta_j^2,$$

o fato de C ser muito grande faz com que todo o primeiro somatório vá para zero na minimização.

Ao mesmo tempo, já vimos que:

- Quando $y^{(i)} = 1$, temos $\theta^T x^{(i)} \geq 1$
- Quando $y^{(i)} = 0$, temos $\theta^T x^{(i)} \leq -1$

e assim chegamos finalmente ao problema de otimização

$$\begin{aligned} & \underset{\theta}{\text{minimize}} \frac{1}{2} \sum_{j=1}^n \theta_j^2 \\ & \text{s.t. } \theta^T x^{(i)} \geq 1 \quad \text{se } y^{(i)} = 1 \\ & \quad \theta^T x^{(i)} \leq -1 \quad \text{se } y^{(i)} = 0 \end{aligned}$$

Por motivos que veremos melhor na próxima seção, a fronteira de decisão da função de hipótese formada por este θ maximiza a distância para as amostras de cada classe. Veja o exemplo linearmente separável na Figura 7.3. De todas as retas que separariam, escolhe-se a preta por ter a maior **margem**. Por isso se diz que o SVM é um classificador de margem larga.

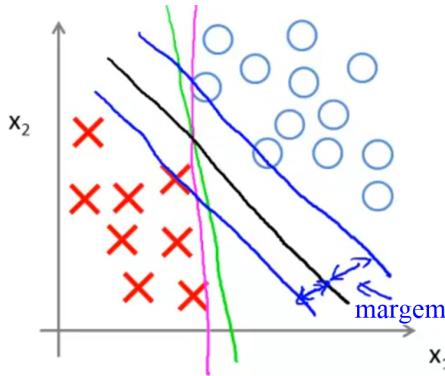


Figura 7.3: SVM - classificador de margem larga.

O grande problema de C muito grande é que ele se torna muito sensível a *outliers* (Figura 7.4). C muito grande mudaria totalmente a fronteira de decisão apenas pela presença de um novo ponto vermelho próximo dos pretos. Já um C menor faz com que a reta quase não se altere, gerando um classificador mais robusto.

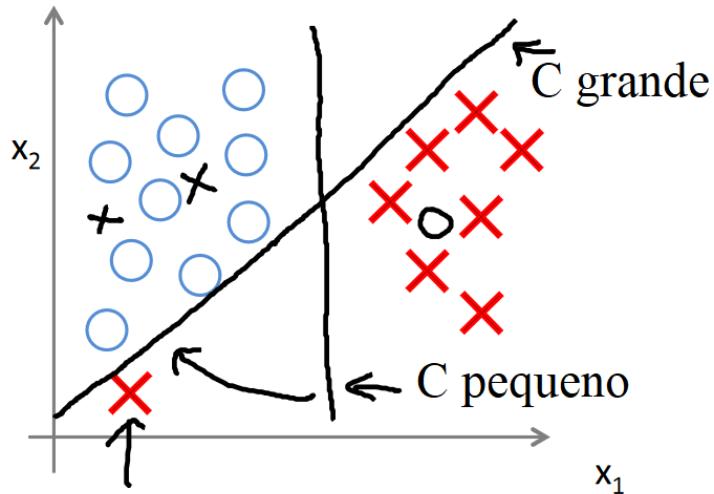


Figura 7.4: SVM - sensibilidade a outliers.

7.3 Matemática da Classificação de Margem Larga

7.3.1 Produto interno

Qual o significado do conceito de produto interno?

Dados os vetores

$$u = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \quad v = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix},$$

definimos então o produto interno

$$u^T v = [u_1 \ u_2] \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = u_1 v_1 + u_2 v_2 = v^T u \in \mathbb{R}.$$

Definimos ainda o comprimento (norma) do vetor u :

$$\|u\| = \sqrt{u_1^2 + u_2^2} \in \mathbb{R}$$

Com base na Figura 7.5, definimos $p \in \mathbb{R}$ como o comprimento da projeção de v sobre u . Repare que este número pode ser negativo se o ângulo entre os vetores for $> 90^\circ$.

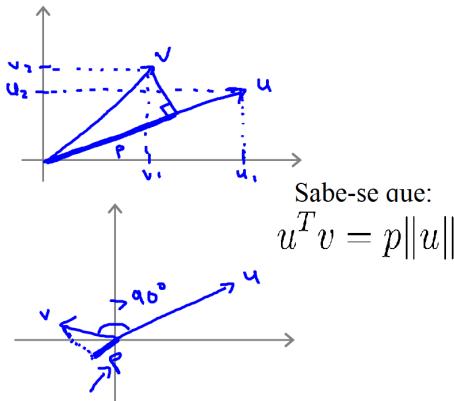


Figura 7.5: Projeção vetorial.

Sabemos então da álgebra linear que

$$u^T v = p \|u\|.$$

7.3.2 Fronteira de decisão do SVM

Voltando ao nosso problema de otimização com C grande:

$$\begin{aligned} &\text{minimize}_{\theta} \frac{1}{2} \sum_{j=1}^n \theta_j^2 \\ &\text{s.t. } \theta^T x^{(i)} \geq 1 \quad \text{se } y^{(i)} = 1 \\ &\quad \theta^T x^{(i)} \leq -1 \quad \text{se } y^{(i)} = 0 \end{aligned}$$

Vamos assumir para simplificar que $n = 2$ e $\theta_0 = 0$. Isso faz com que nossa minimização se torne:

$$\underset{\theta}{\text{minimize}} \frac{1}{2}(\theta_1^2 + \theta_2^2) = \frac{1}{2} \left(\sqrt{\theta_1^2 + \theta_2^2} \right)^2 = \frac{1}{2} \|\theta\|^2.$$

Se chamarmos a projeção de $x^{(i)}$ sobre θ de $p^{(i)}$, então:

$$\theta^T x^{(i)} = p^{(i)} \|\theta\| = \theta_1 x_1^{(i)} + \theta_2 x_2^{(i)}.$$

Nossa otimização com C grande pode ser reescrita:

$$\begin{aligned} & \underset{\theta}{\text{minimize}} \frac{1}{2} \sum_{j=1}^n \theta_j^2 = \frac{1}{2} \|\theta\|^2 \\ \text{s.t. } & p^{(i)} \|\theta\| \geq 1 \quad \text{se } y^{(i)} = 1 \\ & p^{(i)} \|\theta\| \leq -1 \quad \text{se } y^{(i)} = 0 \end{aligned}$$

Sabe-se da álgebra linear que o vetor θ é perpendicular à fronteira de decisão. Como $\|\theta\|$ é minimizado, para termos $p^{(i)} \|\theta\| \geq 1$ para $y = 1$ e $p^{(i)} \|\theta\| \leq -1$ para $y = 0$, precisamos que $p^{(i)}$ tenha magnitude grande e seja positivo para $y = 1$ e negativo para $y = 0$.

Repare visualmente na Figura 7.6 que isso ocorre exatamente quando a fronteira tem a maior margem (direita na figura).

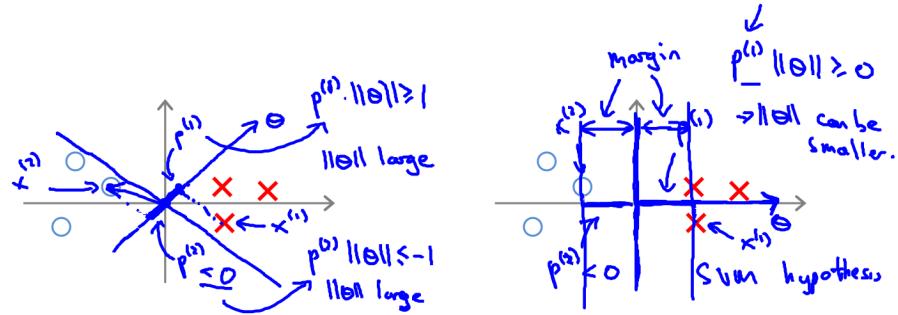


Figura 7.6: Margem larga.

A simplificação θ_0 implica que a fronteira de decisão passa pela origem, mas se não fosse o caso o raciocínio seria o mesmo.

7.4 SVM - Kernels I

Vimos que quando o problema não é linearmente separável (Figura 7.7), a solução na regressão logística é introduzir atributos polinomiais, p.ex., prever $y = 1$ se

$$\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1 x_2 + \theta_4 x_1^2 + \theta_5 x_2^2 + \dots \geq 0.$$

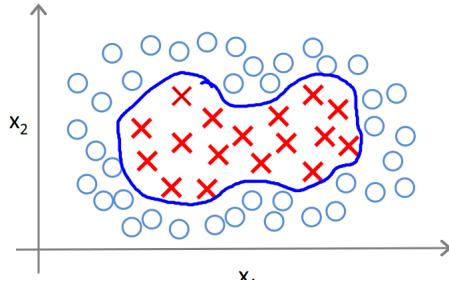


Figura 7.7: Fronteira de decisão não linear.

Já vimos que um problema dessa abordagem é que o número de atributos cresce exponencialmente. Uma opção é trabalhar com outros tipos de funções dos atributos originais f_1, f_2, f_3 , etc.:

$$\theta_0 + \theta_1 f_1 + \theta_2 f_2 + \theta_3 f_3 + \dots$$

Mas como escolher esses novos atributos f_1, f_2, f_3, \dots ?

Dado um exemplo x qualquer (do treino, validação ou teste), o conceito de **kernel** para obter novos atributos é usar uma medida de proximidade entre x e *landmarks* $l^{(1)}, l^{(2)}, l^{(3)}$, etc. (Figura 7.8).

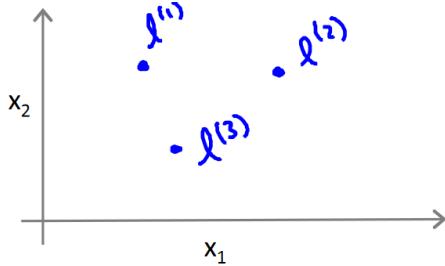


Figura 7.8: Landmarks.

A medida de proximidade mais popular é a seguinte:

$$f_1 = \text{similaridade}(x, l^{(1)}) = \exp\left(-\frac{\|x - l^{(1)}\|^2}{2\sigma^2}\right)$$

$$f_2 = \text{similaridade}(x, l^{(2)}) = \exp\left(-\frac{\|x - l^{(2)}\|^2}{2\sigma^2}\right)$$

$$f_3 = \text{similaridade}(x, l^{(3)}) = \exp\left(-\frac{\|x - l^{(3)}\|^2}{2\sigma^2}\right)$$

Esses são os **Kernels Gaussianos** e os kernels em geral são denotados $k(x, l^{(i)})$.

Note que

- Se $x \approx l^{(1)}$:

$$f_1 \approx \exp\left(-\frac{0^2}{2\sigma^2}\right) \approx 1$$

- Se x está longe de $l^{(1)}$:

$$f_1 = \exp\left(-\frac{\text{número grande}^2}{2\sigma^2}\right) \approx 0$$

Do mesmo modo:

- $f_2 \approx 1$ se $x \approx l^{(2)}$ e $f_2 \approx 0$ se x está longe de $l^{(2)}$
- $f_3 \approx 1$ se $x \approx l^{(3)}$ e $f_3 \approx 0$ se x está longe de $l^{(3)}$
- ...

Vejamos um exemplo:

$$l^{(1)} = \begin{bmatrix} 3 \\ 5 \end{bmatrix}$$

$$f_1 = \exp\left(-\frac{\|x - l^{(1)}\|^2}{2\sigma^2}\right)$$

ilustrado na Figura 7.9 para os casos $\sigma = 1$, $\sigma = 0.5$ e $\sigma = 3$.

Note que a Gaussiana está sempre centrada em $[3, 5]$ e quanto maior o σ mais larga é a superfície, mais lentamente a altura decai.

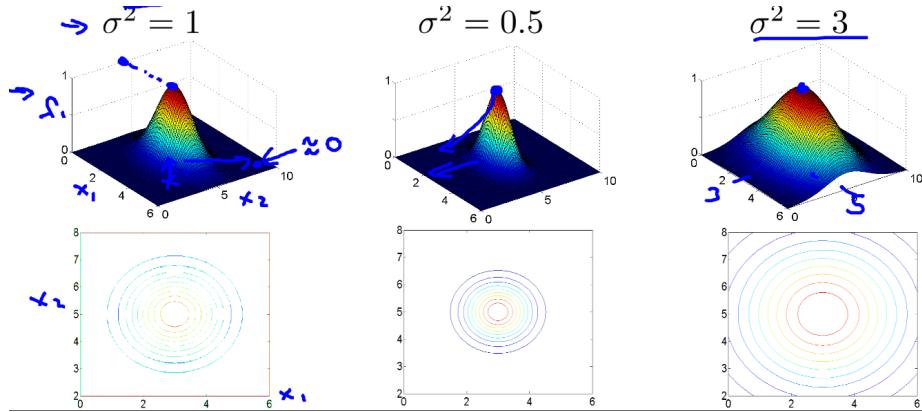


Figura 7.9: Influência de σ sobre f_1 .

A Figura 7.10 ilustra como funciona a fronteira de decisão com kernels. Prevemos $y = 1$ se

$$\theta_0 + \theta_1 f_1 + \theta_2 f_2 + \theta_3 f_3 \geq 0$$

Vamos supor

$$\theta_0 = -0.5 \quad \theta_1 = 1 \quad \theta_2 = 1 \quad \theta_3 = 0.$$

Para o exemplo x próximo de $l^{(1)}$ temos

$$f_1 \approx 1 \quad f_2 \approx 0 \quad f_3 \approx 0$$

e

$$\theta_0 + \theta_1 f_1 + \theta_2 f_2 + \theta_3 f_3 = -0.5 + 1 \cdot 1 = 0.5 \geq 0 \quad (y = 1)$$

Algo parecido ocorre próximo de $l^{(2)}$.

Já o para 3º exemplo, que está distante de todos os *landmarks*, temos:

$$f_1 \approx 0 \quad f_2 \approx 0 \quad f_3 \approx 0$$

e

$$\theta_0 + \theta_1 f_1 + \theta_2 f_2 + \theta_3 f_3 = -0.5 < 0 \quad (y = 0)$$

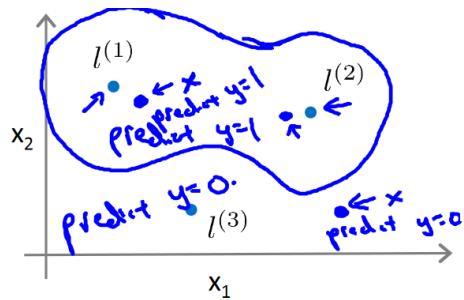


Figura 7.10: Fronteira de decisão com kernels.

7.5 SVM - Kernels II

Onde posicionar os *landmarks* $l^{(1)}, l^{(2)}, l^{(3)}, \dots$?

R.: Vão ser simplesmente os pontos do conjunto de treino!

Dados $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})$, escolhemos então

$$l^{(1)} = x^{(1)}, l^{(2)} = x^{(2)}, \dots, l^{(m)} = x^{(m)}.$$

Para cada exemplo $x^{(i)}$ de treino temos então:

$$\begin{aligned} f_1^{(i)} &= \text{similaridade}(x^{(i)}, l^{(1)}) \\ f_2^{(i)} &= \text{similaridade}(x^{(i)}, l^{(2)}) \\ &\vdots \\ f_i^{(i)} &= \text{similaridade}(x^{(i)}, l^{(i)}) = \exp\left(-\frac{0}{2\sigma^2}\right) = 1 \\ &\vdots \\ f_m^{(i)} &= \text{similaridade}(x^{(i)}, l^{(m)}), \end{aligned}$$

de modo que nosso vetor $x \in \mathbb{R}^{(n+1)}$ (ou \mathbb{R}^n) é substituído pelo novo vetor de atributos

$$f^{(i)} = \begin{bmatrix} f_0^{(i)} \\ f_1^{(i)} \\ f_2^{(i)} \\ \vdots \\ f_m^{(i)}, \end{bmatrix}$$

em que, por convenção, $f_0^{(i)} = 1$.

7.5.1 SVM com kernel

Função de hipótese do SVM com kernel: Dado x , calcular atributos $f \in \mathbb{R}^{m+1}$. O modelo aprende os parâmetros $\theta \in \mathbb{R}^{m+1}$ e prevê $y = 1$ se

$$\theta^T f \geq 0,$$

isto é:

$$\theta_0 f_0 + \theta_1 f_1 + \cdots + \theta_m f_m \geq 0.$$

Treinamento:

$$\underset{\theta}{\text{minimize}} C \sum_{i=1}^m y^{(i)} \text{custo}_1(\theta^T f^{(i)}) + (1 - y^{(i)}) \text{custo}_0(\theta^T f^{(i)}) + \frac{1}{2} \sum_{j=1}^m \theta_j^2.$$

Repare, portanto, que $\theta^T f^{(i)}$ entra no lugar de $\theta_T x^{(i)}$ e o limite superior do somatório regularizador é $j = m$ em vez de $j = n$ como antes simplesmente porque agora $m = n$. Como antes, continuamos excluindo $j = 0$ da regularização.

Um detalhe de implementação aqui é que $\sum_j \theta_j^2 = \theta^T \theta = \|\theta\|^2$ e, para fins de desempenho computacional, esse termo é substituído por $\theta^T M \theta$ para uma matriz M específica. Isso ajuda bastante com o custo computacional quando m é muito grande.

7.5.2 Parâmetros do SVM

Lembramos que $C = \frac{1}{\lambda}$ e assim:

- C grande: viés baixo, variância alta (λ pequeno)
- C pequeno: viés alto, variância baixa (λ grande)

Já para σ^2 :

- σ^2 grande: f_i varia mais suavemente - viés alto, variância baixa
- σ^2 pequeno: f_i varia menos suavemente - viés baixo, variância alta

7.6 Usando um SVM

Existem vários pacotes que resolvem os parâmetros θ para o SVM (liblinear, libsvm,...).

Precisamos especificar

- Parâmetro C
- Kernel (função de similaridade)

O caso sem kernel é chamado de “kernel linear”, i.e., prevê $y = 1$ se

$$\theta^T x \geq 0,$$

ou ainda:

$$\theta_0 + \theta_1 x_1 + \cdots + \theta_n x_n \geq 0.$$

Esta escolha é recomendada quando n é grande e m é pequeno.

Já para o kernel Gaussiano, precisamos ainda escolher σ^2 . Este é recomendado quando n é pequeno e/ou m é grande.

IMPORTANTE: Sempre FAÇA normalização de atributos antes de usar o kernel Gaussiano, já que atributos com magnitudes em um intervalo de valores altos vão influenciar muito no cálculo de $\|x - l\|^2$.

7.6.1 Kernels alternativos

NOTA: Nem toda função similaridade(x, l) gera kernels válidos. Precisa satisfazer uma condição chamada “Teorema de Mercer” para a otimização dos pacotes de SVM rodarem corretamente e não divergirem.

Mas existem outras opções além do Gaussiano:

- Kernel polinomial: $k(x, l) = (x^T l + \text{constante})^{\text{grau}}$. EX.: $(x^T l)^2$, $(x^T l + 5)^4$, etc.
- Kernels mais exóticos: string (usado para texto), chi-quadrado, intersecção de histograma, etc.

7.6.2 Classificação multiclasses

Muitos pacotes de SVM já tratam o problema de K classes (Figura 7.11) naturalmente.

Uma alternativa é usar o método “um-contra-todos”: treinar K SVMs, cada um responsável por distinguir $y = i$ do resto, $i = 1, 2, \dots, K$, obter $\theta^{(1)}, \theta^{(2)}, \dots, \theta^{(K)}$ e tomar a classe i com $(\theta^{(i)})^T x$ maior.

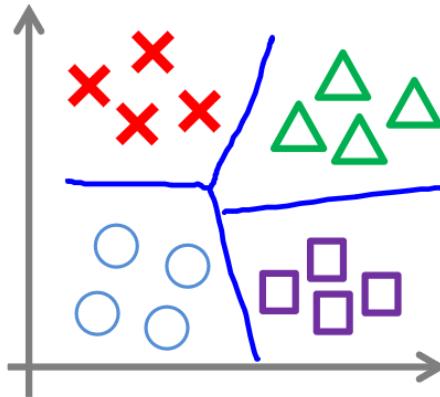


Figura 7.11: Problema multiclasse.

7.6.3 Comparação com outros classificadores

Lembrando: n = número de atributos ($x \in \mathbb{R}^{n+1}$) e m = número de exemplos de treinamento.

Se n é grande (em relação a m), p.ex., $n = 10000$ e $m = 1, \dots, 1000$, usar regressão logística ou SVM sem kernel (“kernel linear”).

Se n é pequeno e m intermediário, p.ex., $n = 1 \sim 1000$ e $m = 10 \sim 10000$, usar SVM com kernel Gaussiano.

Se n é pequeno e m é grande, p.ex., $n = 1 \sim 1000$ e $m = 50000+$, criar/adicionar mais atributos e usar regressão logística ou SVM sem kernel.

Por fim, as redes neurais provavelmente funcionarão bem em todos esses casos, porém, poderão ser mais lentas para treinar.

Capítulo 8

Árvores de Decisão

8.1 Árvore de Decisão

Este é um classificador baseado em regras naturalmente **não linear**.

Imagine um classificador que prevê a viabilidade de esquiar dado o local (latitude) e o momento (mês do ano).

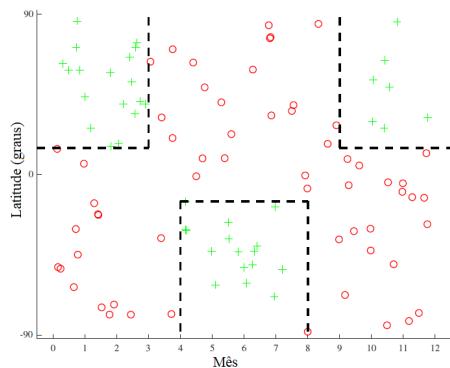


Figura 8.1: Condição para esquiar. Exemplos positivos (bom para esquiar) em verde e negativos em vermelho.

Fronteira de decisão não linear, mas reconhecemos regiões com exemplos positivos e negativos.

Solução: Particionar o espaço de atributos \mathcal{X} em n regiões R_i :

$$\begin{cases} \mathcal{X} = \bigcup_{i=0}^n R_i, & n \in \mathbb{Z}^+ \\ \text{s.t. } R_i \cup R_j = \emptyset \text{ para } i \neq j. \end{cases}$$

Encontrar a partição ótima em geral é **intratável**.

Árvores de decisão usam uma abordagem **gulosa**, **top-down** e **recursiva**.

Algoritmo geral:

1. Divide \mathcal{X} em duas regiões “filhas” por *threshold* de um único atributo
2. Escolhe um novo atributo e divide novamente as regiões filhas em duas por *thresholding*.

Formalmente, temos uma região “pai” R_p , um atributo indexado por j e um *threshold* $t \in \mathbb{R}$. As regiões filhas R_1 e R_2 são obtidas por:

$$\begin{aligned} R_1 &= \{X | X_j < t, X \in R_p\} \\ R_2 &= \{X | X_j \geq t, X \in R_p\} \end{aligned}$$

Mais precisamente:

1. Dividimos \mathcal{X} pelo atributo “latitude” com *threshold* 15 (Figura 8.2).
2. Escolhemos uma das regiões (no caso R_2), o atributo “Mês” e *threshold* 3 (Figura 8.3).
3. Escolhemos um dos nós folha (R_1, R_{21}, R_{22}). No caso, optamos por R_1 , com atributo “longitude” e *threshold* -15 (Figura 8.4).
4. Continuamos até um critério de parada e prevemos a classe majoritária em cada nó folha.

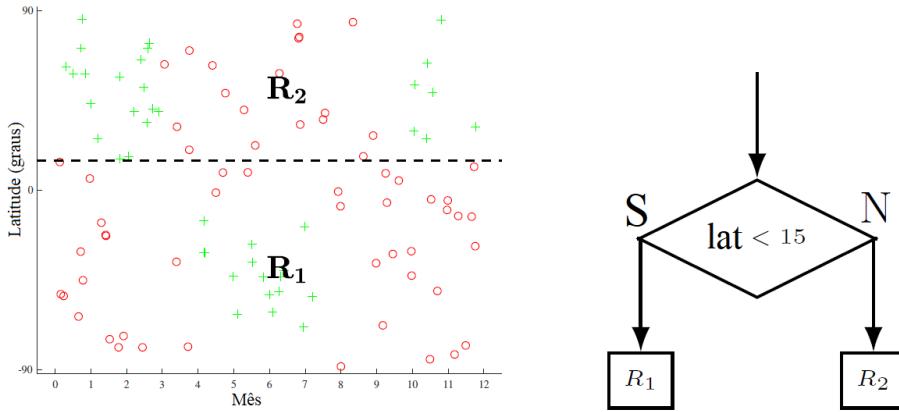


Figura 8.2: Passo 1.

8.2 Função de Perda

Mas afinal, como escolher os pontos de divisão?

Definimos uma perda para cada região pai R_p que se divide em duas filhas R_1 e R_2 :

$$\frac{|R_1|L(R_1) + |R_2|L(R_2)}{|R_1| + |R_2|},$$

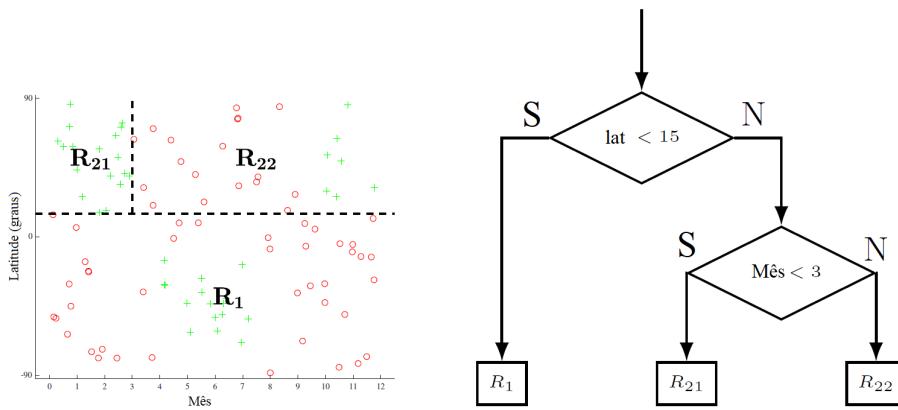


Figura 8.3: Passo 2.

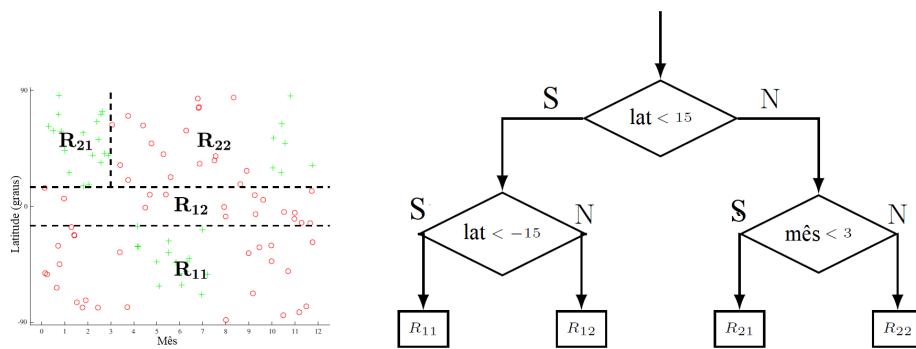


Figura 8.4: Passo 3.

que é a perda ponderada pela cardinalidade de cada região.

O objetivo é maximizar a diminuição da perda em cada divisão:

$$L(R_p) = \frac{|R_1|L(R_1) + |R_2|L(R_2)}{|R_1| + |R_2|}.$$

Na classificação, definimos a perda $L_{misclass}(R)$:

$$L_{misclass}(R) = 1 - \max_c(\hat{p}_c),$$

em que \hat{p}_c é a proporção de exemplos em R que são da classe c . Este é o número de exemplos que seriam classificados erroneamente se previrmos a classe majoritária para a região R (que é exatamente o que o algoritmo faz!).

8.2.1 Problema

$L_{misclass}$ não é sensível à distribuição de probabilidades de cada classe. Veja o exemplo na Figura 8.5.

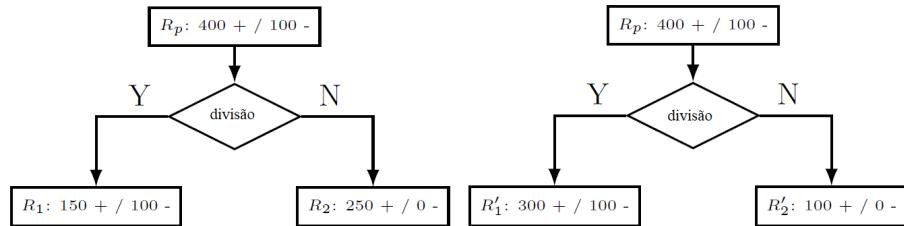


Figura 8.5: Insensibilidade à distribuição das classes (número de exemplos positivos e negativos).

Na 1ª divisão temos:

$$L(R_p) = \frac{|R_1|L(R_1) + |R_2|L(R_2)}{|R_1| + |R_2|} = \frac{250 \cdot 100 + 250 \cdot 0}{250 + 250} = .$$

Já na 2ª:

$$L(R_p) = \frac{|R'_1|L(R'_1) + |R'_2|L(R'_2)}{|R'_1| + |R'_2|} = \frac{400 \cdot 100 + 100 \cdot 0}{400 + 100} = .$$

Note que apesar de a 1ª divisão ter isolado a maioria dos exemplos positivos, temos 2 problemas aí:

1. Ambas as divisões têm a mesma perda.
2. A perda do nó pai $L(R_p)$ não diminuiu.

Uma função de perda mais sensível a isso é a **cross-entropy**:

$$L_{cross} = - \sum_c \hat{p}_c \log_2 \hat{p}_c,$$

em que $\hat{p} \log_2 \hat{p} \equiv 0$ se $\hat{p} = 0$.

Na teoria da informação, a *cross-entropy* mede número de bits necessários para especificar a saída (classe) dado que sua distribuição é conhecida.

A redução na perda do nó pai para os filhos é chamada de **ganho de informação**.

Para entender essa sensibilidade, considere o caso binário, em que \hat{p}_i é a proporção de exemplos positivos em R_i e as perdas são simplificadas:

$$L_{misclass}(R) = L_{misclass}(\hat{p}) = 1 - \max(\hat{p}, 1 - \hat{p})$$

$$L_{cross}(R) = L_{cross}(\hat{p}) = -\hat{p} \log \hat{p} - (1 - \hat{p}) \log(1 - \hat{p}).$$

A Figura 8.6 mostra cada perda em função de \hat{p} na 1ª divisão da Figura 8.5.

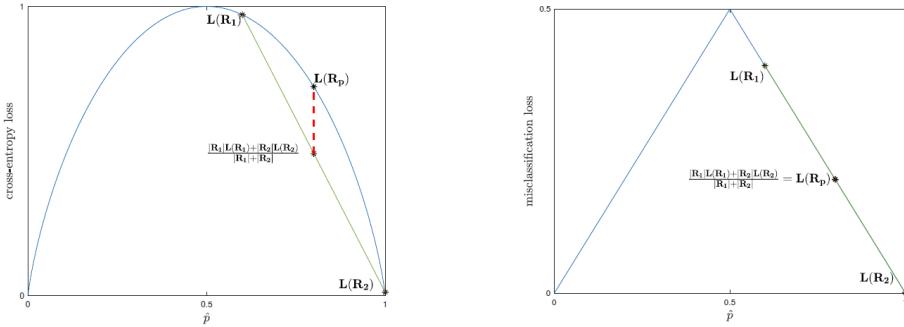


Figura 8.6: $L_{misclass} \times L_{cross}$.

Note que a *cross-entropy* é estritamente côncava e pode-se observar então do gráfico (e facilmente provado) que, sempre que $\hat{p}_1 \neq \hat{p}_2$ e ambas as regiões filhas são não vazias, a soma ponderada da perda dos filhos é sempre menor que a perda do pai. O mesmo não é verdade para a $L_{misclass}$.

Por essa razão, L_{cross} (ou a perda de Gini, que é relacionada) são escolhas naturais para o crescimento de árvores de decisão.

8.2.2 Regressão

A regressão é bem parecida com a classificação, exceto que a predição final é dada pela média:

$$\hat{y} = \frac{\sum_{i \in R} y_i}{|R|}.$$

A função de perda neste caso é a **perda quadrática**:

$$L_{squared}(R) = \frac{\sum_{i \in R} (y_i - \hat{y})^2}{|R|}.$$

8.3 Outras Considerações

Árvores de decisão são populares em grande parte por serem fáceis de explicar e entender e interpretáveis. Porém há outros aspectos relevantes.

8.3.1 Atributos Categóricos

Outra vantagem é a capacidade de tratar atributos categóricos. Por exemplo, a localização no exemplo do esqui poderia ser hemisfério norte, sul ou equador ($\text{loc} \in \{N, S, E\}$) e teríamos a árvore da Figura 8.7.

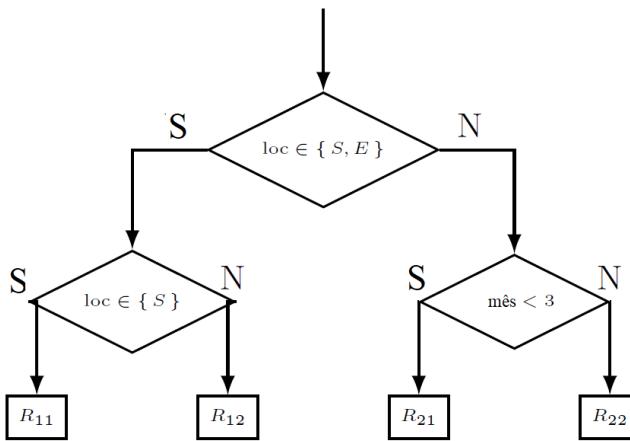


Figura 8.7: Atributo categórico.

ATENÇÃO: Se temos n categorias, o número de decisões possíveis para aquele atributo será 2^n . Assim, um número muito grande de categorias torna o problema intratável. Embora no caso binário, existam otimizações próprias para isso, deve-se considerar seriamente a possibilidade de converter um atributo deste tipo em quantitativo, mesmo porque um número alto de *thresholds* possíveis gera *overfitting*.

8.3.2 Regularização

O critério de parada mais simples é crescer a árvore até que cada nó folha contenha exatamente **um** exemplo de treinamento.

Porém essa abordagem gera um modelo com alta variância (baixo viés).

Outros critérios de parada são usados então:

- **Tamanho mínimo de folha:** Não dividir R se sua cardinalidade ficar abaixo de um *threshold* fixo.
- **Profundidade máxima:** Não dividir R se mais do que um número fixo de divisões foi necessário para chegar até R .

- **Número máximo de nós:** Parar se a árvore tem mais do que um número fixo de nós folha.
- **Poda:** Crescer a árvore totalmente e cortar nós que decresçam minimamente o erro de classificação (ou quadrático) em um conjunto de validação.

ATENÇÃO: Não se recomenda parar apenas porque a redução na perda após uma divisão é pequena. A árvore pode perder interações de alto nível entre os atributos e assim terminar prematuramente.

8.3.3 Tempo de Execução

Árvores de decisão são algoritmos rápidos.

Considere um problema de classificação binária com n exemplos, f atributos e árvore com profundidade d .

No teste, para a busca pela predição deve ir até um nó folha e o tempo então é $\mathcal{O}(d)$. Se a árvore estiver balanceada, temos $d = \mathcal{O}(\log n)$. O tempo de teste costuma ser bem rápido, portanto.

Já no treino, cada ponto pode aparecer no máximo em $\mathcal{O}(d)$ nós. Com técnicas de ordenação e *cache* de valores intermediários, pode-se chegar a um tempo $\mathcal{O}(1)$ em cada nó para um único exemplo e único atributo. O tempo total então é $\mathcal{O}(nfd)$, o que é bem rápido se considerarmos que a matriz de *design* por si só tem tamanho nf .

8.3.4 Perda de Estrutura Aditiva

Uma árvore de decisão não captura facilmente uma estrutura aditiva. Considere na Figura 8.8 uma simples fronteira de decisão $x_1 + x_2$

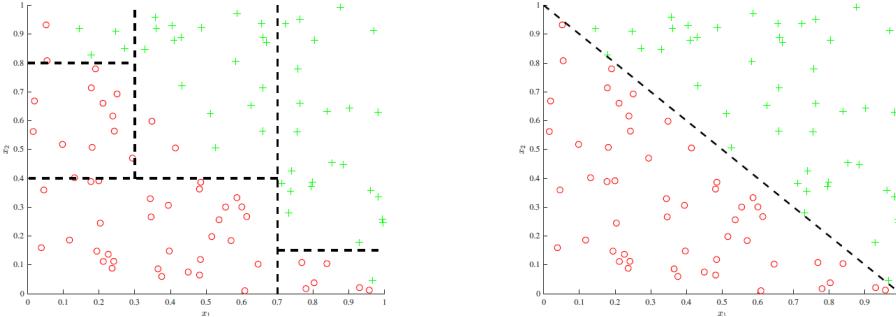


Figura 8.8: Perda de estrutura aditiva.

Uma árvore, considerando um atributo por vez, precisaria de muitas divisões para aproximar esta reta.

Existem soluções de árvores que usam vários atributos cada vez, mas costumam ter maior variância e perder interpretabilidade.

8.3.5 Melhorias

Em geral, a alta variância e perda de estrutura aditiva costuma fazer com que a acurácia de árvores individuais seja baixa. A solução para são os *ensembles*!

Capítulo 9

Ensemble

9.1 Ensemble

IDEIA BÁSICA: Combinar vários classificadores de acurácia baixa para formar um único classificador de acurácia alta.

Se os classificadores-base tiverem baixa correlação, o modelo combinado vai ter baixa variância.

Árvores de decisão são os modelos mais usados como classificadores fracos.

Veremos duas estratégias: *bagging* e *boosting*.

9.2 *Bagging*

Bagging vem de *bootstrap aggregation*. Precisamos definir primeiro então o que é *bootstrap*.

9.2.1 *Bootstrap*

Bootstrap é uma alça ou aba na parte superior de uma bota de cano alto que permite elevar o cano e facilita de calçá-la.

Ganhou vários significados em várias áreas, sempre relacionados à ideia de “melhorar a si mesmo com esforço próprio, sem ajuda externa”.

Na estatística, *bootstrap* é uma técnica para medir a incerteza de algum estimador (por exemplo, a média). A ideia básica é fazer reamostragens aleatórias com repetição.

Suponha que tenhamos uma população P e um conjunto de treinamento S amostrado de P ($S \sim P$). Se quisermos estimar, por exemplo, a média usando S não temos como saber o erro dessa estimativa. Para isso, precisaríamos de vários conjuntos S_1, S_2, \dots independentes, todos amostrados de P .

No *bootstrap*, assume-se que $S = P$ e então são gerados os conjuntos de *bootstrap* a partir de S com repetição ($Z \sim S, |Z| = |S|$). Muitas amostras

Z_1, Z_2, \dots, Z_M podem ser geradas com este processo e assim temos uma medida da variância do nosso estimador nesses conjuntos.

9.2.2 Agregamento

No *ensemble*, treinamos um modelo de aprendizado G_m sobre cada conjunto Z_m . O **preditor agregado** é dado pela média:

$$G(x) = \sum_m \frac{G_m(x)}{M}.$$

Este processo é chamado de **bagging**.

Mostra-se que preditores gerados desta forma tendem a ser menos correlacionados.

Embora o viés de cada preditor individual aumente por não usar todo o conjunto de treino disponível, a queda na variância compensa na prática.

Outra vantagem é poder usar o **out-of-bag estimation**. Mostra-se que cada amostra de *bootstrap* contém em média $\frac{2}{3}$ de S . Assim, $\frac{1}{3}$ pode ser usado para estimar o erro, chamado de *out-of-bag error*. No limite $M \rightarrow \infty$, este erro equivale ao do *leave-one-out*.

9.2.3 Bagging + Árvores de Decisão

Como vimos, árvores são modelos com baixo viés e alta variância, propícios portanto para *bagging*.

Uma primeira vantagem de aplicar *bagging* em árvores é o tratamento de atributos com valores faltantes. No *ensemble*, podemos simplesmente excluir árvores que façam divisão com base naquele atributo.

Já uma desvantagem é a perda de interpretabilidade. Um atenuante é a técnica **medida de importância de atributo**: média da diminuição da perda em todas as divisões feitas sobre aquele atributo no *ensemble*. Outra desvantagem esperada é um aumento no custo computacional.

Por fim, se temos um atributo que é um preditor muito forte, a tendência é que todas as árvores do *ensemble* usem este atributo e teríamos classificadores correlacionados. Um método que resolve isso são as **florestas aleatórias**. Só um sub-conjunto aleatório dos atributos podem ser usados em cada divisão. Isso implica em um aumento no viés, já que restringe o espaço de atributos, mas isso não costuma ser um problema.

9.3 Boosting

Enquanto *bagging* reduzia variância, *boosting* é usado para **redução de viés**.

Classificadores-base com alto viés e baixa variância, chamados de **classificadores fracos**.

Árvores de decisão se tornam classificadores fracos se permitirmos que tomem apenas uma decisão antes da predição. Isso é chamado de árvore de decisão “firme” (*decision stump*).

Veja o exemplo na Figura 9.1. Partimos do conjunto à esquerda e treinamos um único *decision stump* no centro. Em seguida, observamos os exemplos classificados incorretamente e aumentamos o peso relativo desses exemplos em relação aos que foram acertados. Treina-se então um novo *decision stump* que é incentivado a acertar estes “exemplos mais difíceis”. Continua o processo e vai incrementalmente reajustando estes pesos, chegando-se ao final a um *ensemble* de classificadores fracos.

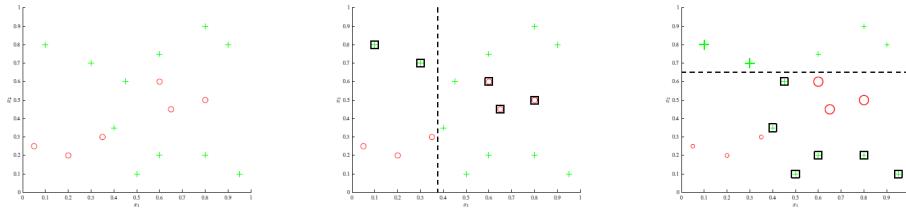


Figura 9.1: *Decision stump*.

9.3.1 Adaboost

Adaboost é o algoritmo de *boosting* mais popular.

Entrada: Conjunto de dados rotulados $\{(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)\}$

Saída : Ensemble classificador $f(x)$

$w_i \leftarrow \frac{1}{N}$ para $i = 1, 2, \dots, N$

for $m = 0$ **to** M **do**

Ajustar o classificador fraco G_m ao conjunto de treino ponderado por w_i

Calcular o erro ponderado $err_m = \frac{\sum_i w_i \mathbb{1}_{(y_i \neq G_m(x_i))}}{\sum w_i}$

Calcular o peso $\alpha_m = \log\left(\frac{1-err_m}{err_m}\right)$

$w_i \leftarrow w_i * \exp(\alpha_m \mathbb{1}_{(y_i \neq G_m(x_i))})$

end

$f(x) = \text{sign}(\sum_m \alpha_m G_m(x))$

Algoritmo 1: Adaboost.

Note que os pesos começam todos iguais e exemplos com classificação incorreta têm seu peso aumentado a cada iteração. O classificador final é uma soma ponderada pelos log-odds negativos dos erros ponderados.

Esta soma final permite que o *ensemble* inclua termos aditivos aumentando a capacidade (e variância) do modelo final. Note também que M muito grande

pode causar *overfitting*, já que cada classificador fraco não é mais independente do modelo anterior na sequência.

Embora pareça arbitrário, cada peso escolhido no Adaboost tem sua justificativa teórica.

9.4 Forward Stagewise Additive Modeling

O Adaboost é parte de uma família maior de métodos chamados de *Forward Stagewise Additive Modeling*.

Entrada: Conjunto de dados rotulados $\{(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)\}$

Saída : Ensemble classificador $f(x)$

Iniciarizar $f_0(x) = 0$

for $m = 0$ **to** M **do**

Calcular $(\beta_m, \gamma_m) = \operatorname{argmin}_{\beta, \gamma} \sum_{i=1}^N L(y_i, f_{m-1}(x_i) + \beta G(x_i; \gamma))$
Fazer $f_m(x) = f_{m-1}(x) + \beta_m G(x; y_i)$

end

$f(x) = f_m(x)$

Algoritmo 2: Forward Stagewise Additive Modeling.

Este método preserva o caráter aditivo do *ensemble*.

Note que agora os classificadores fracos são parametrizados por γ .

O objetivo em cada passo é obter parâmetros do classificador fraco e poderá-lo de modo que melhor representem o erro do *ensemble*.

Uma perda quadrática neste modelo, por exemplo, equivale a ajustar classificadores individuais ao resíduo $y_i - f_{m-1}(x_i)$.

Adaboost é um caso particular com 2 classes com perda exponencial:

$$L(y, \hat{y}) = \exp(-y\hat{y}).$$

9.5 Gradient Boosting

O problema de minimização no *Forward Stagewise Additive Modeling* não costuma ter solução analítica. Métodos de alto desempenho como **xgboost** usam otimização numérica para isso.

A maior dificuldade para se aplicar o gradiente descendente usual sobre a perda é que os passos devem ser feitos dentro da classe de modelos, não são arbitrários.

No **gradient boosting**, o gradiente em cada ponto é calculado em relação ao preditor atual (tipicamente um *decision stump*):

$$g_i = \frac{\partial L(y, f(x_i))}{\partial f(x_i)}.$$

Treina-se então um novo preditor de regressão que se ajusta a este gradiente e é usado como passo do gradiente. No *Forward Stagewise Additive Modeling*, isso se torna

$$\gamma_i = \operatorname{argmin}_{\gamma} \sum_{i=1}^N (g_i - G(x_i; \gamma))^2.$$

Capítulo 10

Agrupamento

10.1 Aprendizado Não Supervisionado - Agrupamento (*Clustering*)

No aprendizado supervisionado, os exemplos tinham rótulos (Figura 10.1 esquerda):

$$\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), (x^{(3)}, y^{(3)}), \dots, (x^{(m)}, y^{(m)})\},$$

enquanto que no não supervisionado os rótulos não existem (Figura 10.1 direita):

$$\{x^{(1)}, x^{(2)}, x^{(3)}, \dots, x^{(m)}\}$$

e o objetivo é encontrar estruturas e grupos coerentes nos dados.

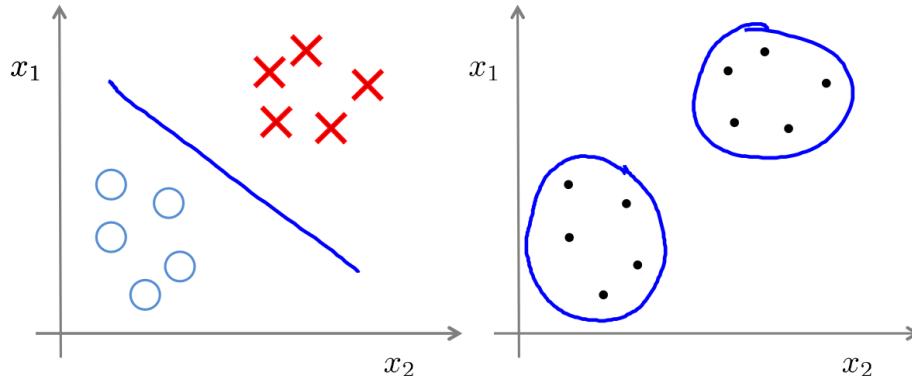


Figura 10.1: Aprendizado supervisionado (esquerda) vs não supervisionado (direita).

Exemplos de aplicação:

- Segmentação de mercado: perfil de consumidores

- Análise de redes sociais: grupos de usuários que interagem mais entre si
- Otimização de clusters de computadores: estrutura de rede que agrupe nós que se comunicam mais
- Análise de dados astronômicos: identificação de galáxias e aglomerados

10.2 Agrupamento - K-médias (K-means)

A Figura 10.2 mostra, de cima para baixo, esquerda para a direita, a execução do algoritmo de K-means.

No exemplo, com 2 grupos, o algoritmo trabalha com 2 pontos de referência, que são os chamados **centroídes** do cluster.

Esses centroídes são inicializados aleatoriamente e cada exemplo é atribuído ao centroide mais próximo (e consequentemente ao cluster correspondente).

Em seguida, esses centroídes são recalculados, como sendo a média dos pontos atribuídos a cada um deles.

Os pontos de exemplo são então redistribuídos aos novos centroídes, novamente usando o critério de atribuir cada ponto ao centroide mais próximo.

Os centroídes são atualizados novamente e o processo vai se repetindo até que nenhuma atualização nas atribuições e nos centroídes seja mais executada.

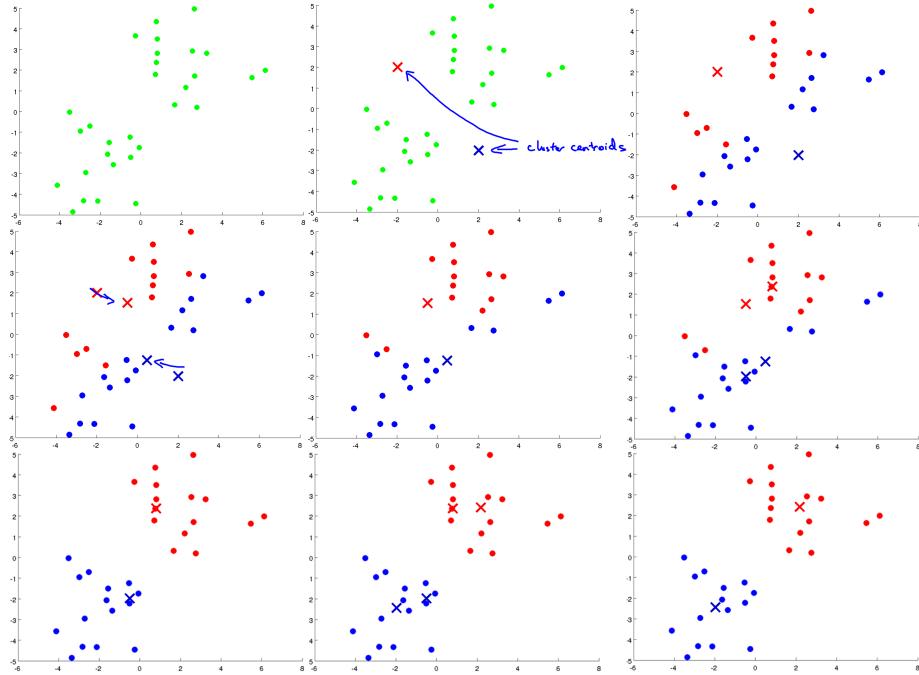


Figura 10.2: K-means.

Para o algoritmo, teremos as seguintes entradas:

- K (número de clusters)
- Conjunto de treinamento $\{x^{(1)}, x^{(2)}, \dots, x^{(m)}\}$

Teremos $x^{(i)} \in \mathbb{R}^n$ e por convenção $x_0 = 1$.

Algoritmo:

```

Iniciar aleatoriamente  $K$  centroides  $\mu_1, \mu_2, \dots, \mu_K \in \mathbb{R}^n$ 
Repita {
    para  $i = 1$  até  $m$  % passo de atribuição do cluster
         $c^{(i)} := \text{minimize}_k \|x^{(i)} - \mu_k\|^2$  % índice (de 1 a  $K$ ) do centroide mais próximo de  $x^{(i)}$ 
    para  $k = 1$  até  $K$  % movimento do centroide
         $\mu_k :=$  média dos pontos atribuídos ao cluster  $k$ 
}

```

No segundo laço, a título de ilustração, se os exemplos $x^{(1)}, x^{(5)}, x^{(6)}$ e $x^{(10)}$ são atribuídos ao cluster 2, i.e., $c^{(1)} = 2, c^{(5)} = 2, c^{(6)} = 2$ e $c^{(10)} = 2$, então μ_2 seria atualizado como

$$\mu_2 = \frac{1}{4} (x^{(1)} + x^{(5)} + x^{(6)} + x^{(10)}) \in \mathbb{R}^n.$$

Por fim, cabe destacar que se por um lado o K-means faz todo sentido quando os clusters são evidentes (Figura 10.3 esquerda), por outro ele também pode ser aplicado quando essa separação não é clara, como na Figura 10.3 direita, que é um exemplo de segmentação de mercado, para o número de opções de tamanho de uma camiseta com base em peso e altura da população.

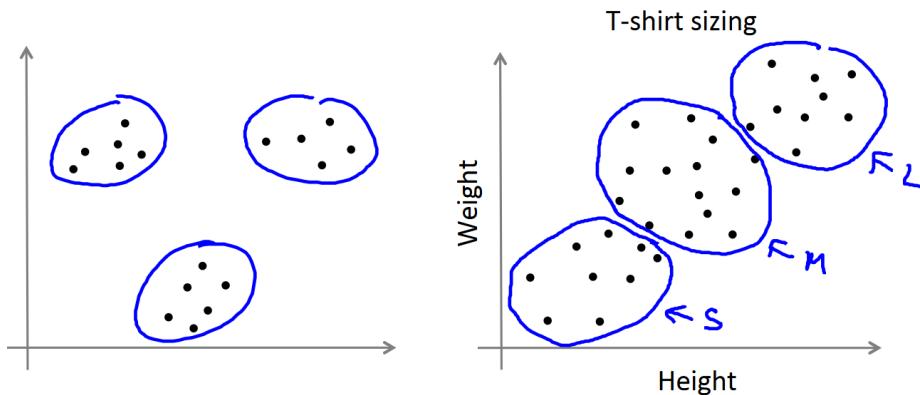


Figura 10.3: K-means para grupos não separados.

10.3 Função Objetivo

Notação:

- $c^{(i)}$: índice do cluster ($1, 2, \dots, K$) ao qual o exemplo $x^{(i)}$ está atribuído
- μ_k : centroide k ($\mu_k \in \mathbb{R}^n$)
- $\mu_{c^{(i)}}$: centroide do cluster ao qual o exemplo $x^{(i)}$ foi atribuído

A função objetivo no K-means é

$$J(c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_K) = \frac{1}{m} \sum_{i=1}^m \|x^{(i)} - \mu_{c^{(i)}}\|^2$$

e o problema de otimização é

$$\underset{\substack{c^{(1)}, \dots, c^{(m)} \\ \mu_1, \dots, \mu_K}}{\text{minimize}} J(c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_K).$$

A função J é chamada de **distorção**.

Pode-se demonstrar que o primeiro laço do algoritmo (atribuição do cluster) minimiza J w.r.t. $c^{(1)}, \dots, c^{(m)}$, mantendo-se μ_1, \dots, μ_K fixos.

Já o segundo laço (movimento do centroide) minimiza J w.r.t. μ_1, \dots, μ_K .

10.4 Inicialização Aleatória

A abordagem mais efetiva para inicialização dos centroides é selecionar K exemplos de treinamento aleatoriamente e fazer μ_1, \dots, μ_K iguais a esses exemplos.

Repare que $K < m$, de outro modo, o agrupamento não faria sentido!

Outra característica do K-means é que diferentes inicializações aleatórias geram diferentes agrupamentos (Figura 10.4). Alguns interessantes, outros não. Fato é que mesmo visualmente existem situações em que o padrão de agrupamento não é óbvio (Figura 2.3 esquerda). Isso é consequência de mínimos locais obtidos para a função objetivo J .

Como chegar então na melhor configuração? Uma opção é rodar o algoritmo várias vezes e tomar o caso com menor custo:

```

Para  $i = 1$  até 100{
    Inicializar o K-means aleatoriamente.
    Rodar o K-means e obter  $c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_K$ .
    Calcular a função de custo (distorção)
         $J(c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_K)$ 
}
Tome o agrupamento com menor custo  $J(c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_K)$ 
```

Essa é uma solução particularmente interessante quando K é pequeno (2 a 5, por exemplo), quando é muito grande geralmente não é necessário.

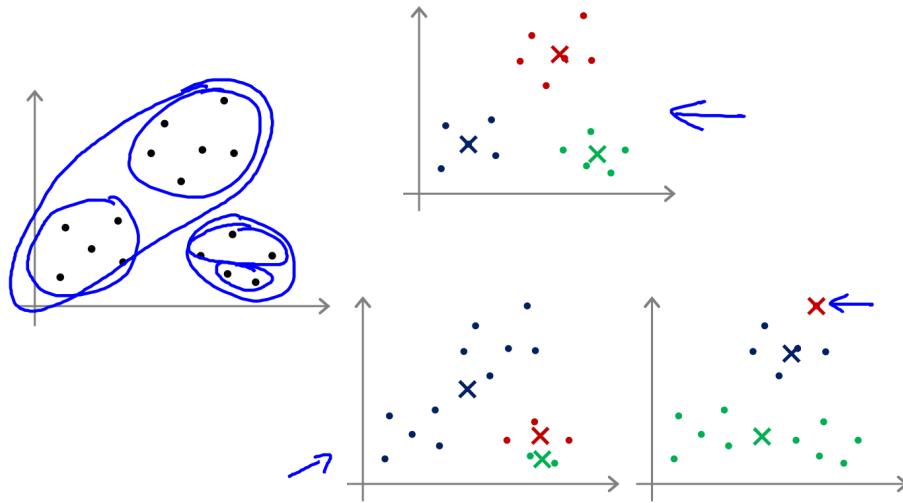


Figura 10.4: Ótimos locais.

10.5 Número de Clusters

Qual o valor correto de K ?

Uma possibilidade é o método do “cotovelo” (“elbow”).

A ideia é plotar o custo J em função do número de clusters e identificar o número de clusters a partir do qual o custo decresce muito mais lentamente, formando um “cotovelo” (Figura 10.5 esquerda). O problema é que muitas vezes o decaimento é homogêneo (Figura 10.5 direita) e não se detecta um cotovelo claro.

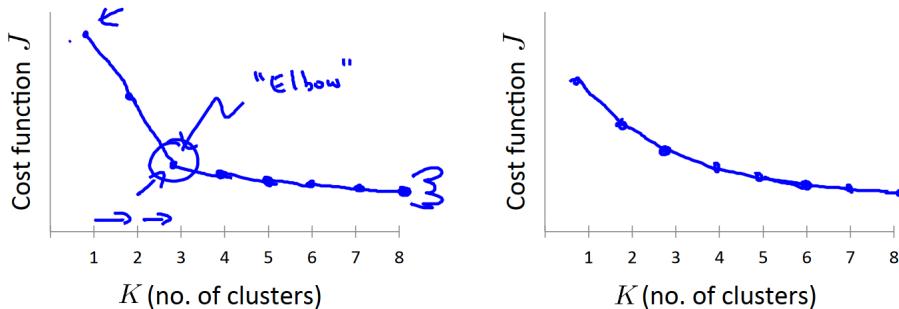


Figura 10.5: Método do cotovelo.

Mas a solução mais interessante na maioria das vezes na prática é levar em conta o contexto do problema sendo resolvido e usar como referência alguma métrica do quanto bem aquele agrupamento funciona para aquele propósito.

Por fim, existem casos em que nenhum exemplo é atribuído a determinado

centroide. Nesses casos, o mais comum é eliminar aquele cluster (o problema passa a ter $K - 1$ grupos). Outra solução, mas menos usual, é rodar novamente com outra inicialização aleatória.

Capítulo 11

Redução de Dimensionalidade

11.1 Redução de Dimensionalidade - Motivação I: Compressão

Considere na Figura 11.1 os atributos x_1 , que é um comprimento em cm, e x_2 , em polegada. Estas são obviamente informações totalmente correlacionadas e, como se vê no gráfico, estão muito bem alinhadas (só não temos uma reta perfeita porque ambos são arredondados para valores inteiros). Não precisamos dos dois atributos e podemos então reduzir o tamanho deste dado (comprimir).

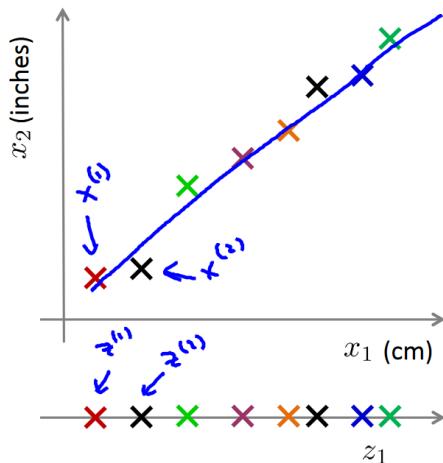


Figura 11.1: Redução de dimensionalidade: 2D para 1D.

Esse pode ter sido um exemplo pouco realista, mas situações de correlação no mundo real são muito comuns. Imagine por exemplo que x_1 poderia ser a habilidade de um piloto de helicóptero de controle remoto e x_2 o seu nível de diversão. Essas são variáveis correlacionadas e poderiam ser substituídas por um único atributo “aptidão”.

Em ambos os casos, podemos reduzir cada um de m exemplos de 2D para 1D, expressando em um único **componente** z_1 , i.e.:

$$\begin{aligned} x^{(1)} \in \mathbb{R}^2 &\rightarrow z^{(1)} \in \mathbb{R} \\ x^{(2)} \in \mathbb{R}^2 &\rightarrow z^{(2)} \in \mathbb{R} \\ &\vdots \\ x^{(m)} \in \mathbb{R}^2 &\rightarrow z^{(m)} \in \mathbb{R} \end{aligned}$$

Processo similar se usa para reduzir de 3D para 2D (Figura 11.2). Nesse caso, temos 2 vetores z_1 e z_2 para formar um plano e, partindo-se de um dado $x^{(i)} \in \mathbb{R}^3$, converte-se para $z^{(i)} \in \mathbb{R}^2$:

$$z = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} \quad z^{(i)} = \begin{bmatrix} z_1^{(i)} \\ z_2^{(i)} \end{bmatrix}$$

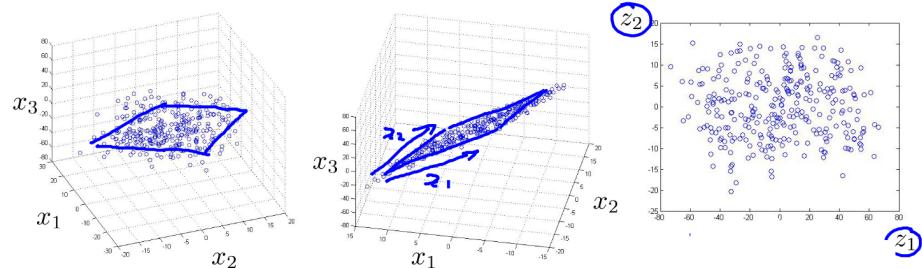


Figura 11.2: Redução de dimensionalidade: 3D para 2D.

Por fim, o mesmo processo é feito para qualquer dimensão, por exemplo, reduzindo de 1000D para 100D.

11.2 Motivação II: Visualização

Considere a tabela na Figura 11.3 que contém 50 atributos para cada país, i.e., $x^{(i)} \in \mathbb{R}^{50}$.

Podemos reduzir esse dado de 50D para 2D, i.e., $z^{(i)} \in \mathbb{R}^2$:

Country	x_1 GDP (trillions of US\$)	x_2 Per capita GDP (thousands of intl. \$)	x_3 Human Develop- ment Index	x_4 Life expectancy	x_5 Poverty Index (Gini as percentage)	x_6 Mean household income (thousands of US\$)	...
Canada	1.577	39.17	0.908	80.7	32.6	67.293	...
China	5.878	7.54	0.687	73	46.9	10.22	...
India	1.632	3.41	0.547	64.7	36.8	0.735	...
Russia	1.48	19.84	0.755	65.5	39.9	0.72	...
Singapore	0.223	56.69	0.866	80	42.5	67.1	...
USA	14.527	46.86	0.91	78.3	40.8	84.3	...
...

[resources from en.wikipedia.org]

Figura 11.3: Atributos de países.

País	z_1	z_2
Canadá	1.6	1.2
China	1.7	0.3
Índia	1.6	0.2
Rússia	1.4	0.5
Singapura	0.5	1.7
EUA	2	1.5
...

e isso então permite a **visualização** do dado na Figura 11.4. Pode-se identificar, por exemplo, que z_1 corresponde ao tamanho do GDP do país (atividade econômica), enquanto z_2 seria o GDP por pessoa.

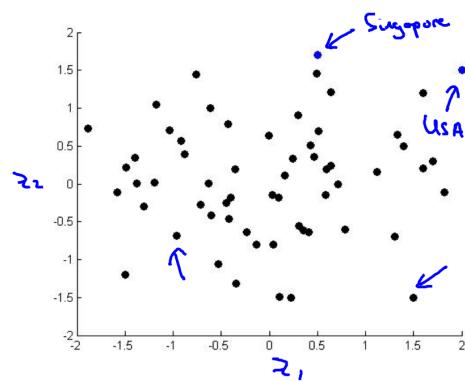


Figura 11.4: Visualização 2D.

Aqui podemos identificar, por exemplo, países com grande atividade econômica

geral e também por pessoa, como os EUA, ou ainda países com grande GDP por pessoa mas menor GDP geral (devido ao menor tamanho) como Singapura, etc.

11.3 Análise de Componentes Principais - Formulação

Considere a Figura 11.5 com $x \in \mathbb{R}^2$. Neste caso, o método de Análise de Componentes Principais (PCA da sigla em inglês) reduz a dimensionalidade de 2D para 1D projetando os dados em um vetor $u^{(1)} \in \mathbb{R}^n$ de modo a minimizar a soma das distâncias de cada ponto ao vetor, elevadas ao quadrado.

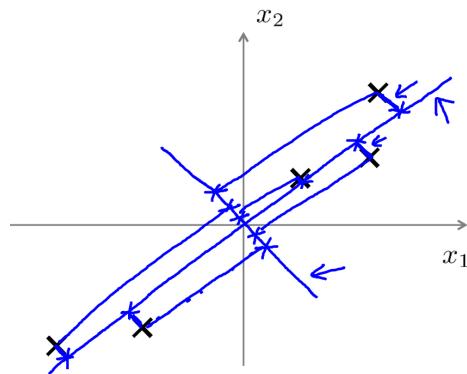


Figura 11.5: Formulação PCA.

Essa soma das distâncias ao quadrado é chamada de **erro de projeção** e pode ser generalizada para maiores dimensões. Na Figura 11.6 temos o caso 3D \rightarrow 2D.

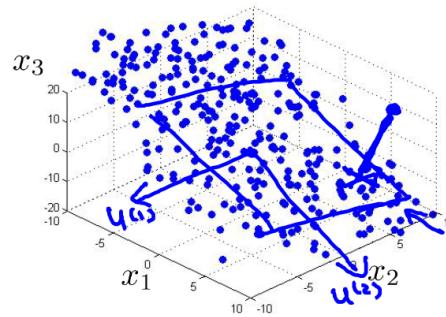


Figura 11.6: Projeção PCA 3D para 2D.

Em geral, reduzimos de n para k dimensões encontrando k vetores

$u^{(1)}, u^{(2)}, \dots, u^{(k)}$ de modo que a projeção do dado sobre esses vetores minimize o erro de projeção.

NOTA: o sentido do vetor não importa, apenas a direção, de modo que na projeção para 1D podemos usar $u^{(1)}$ ou $-u^{(1)}$.

ATENÇÃO: PCA NÃO é regressão linear. Apesar de algumas aparentes semelhanças, são algoritmos completamente distintos:

1. PCA minimiza a distância PERPENDICULAR ao vetor/reta, enquanto a regressão linear minimiza a distância paralela ao eixo y (Figura 11.7)
2. Na regressão linear temos uma variável de resposta diferenciada y , enquanto no PCA todas as variáveis estão no mesmo nível (Figura 11.8)

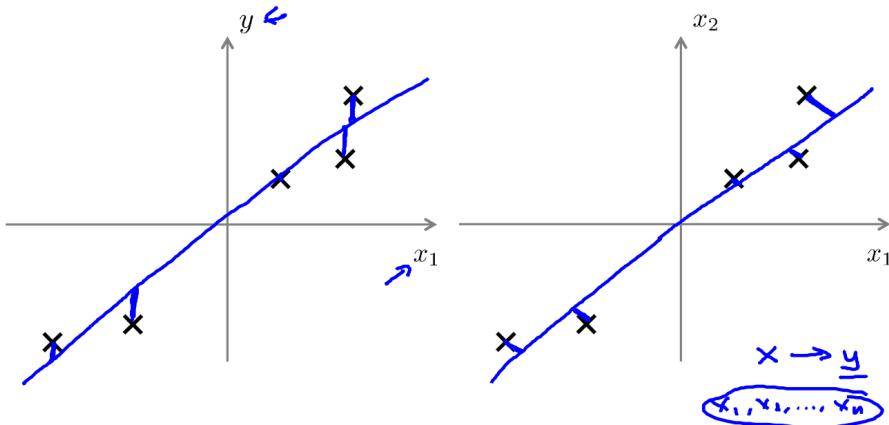


Figura 11.7: PCA vs regressão linear: cálculo diferente das distâncias.

11.4 PCA - Algoritmo

Partimos do conjunto de treinamento: $x^{(1)}, x^{(2)}, \dots, x^{(m)}$. Note que não temos rótulo (não supervisionado).

Iniciamos com um pré-processamento de normalização dos atributos pela média/escala.

Calculamos a média de cada atributo j sobre todo o treinamento:

$$\mu_j = \frac{1}{m} \sum_{i=1}^m x_j^{(i)}$$

e substituímos cada $x_j^{(i)}$ por $x_j - \mu_j$.

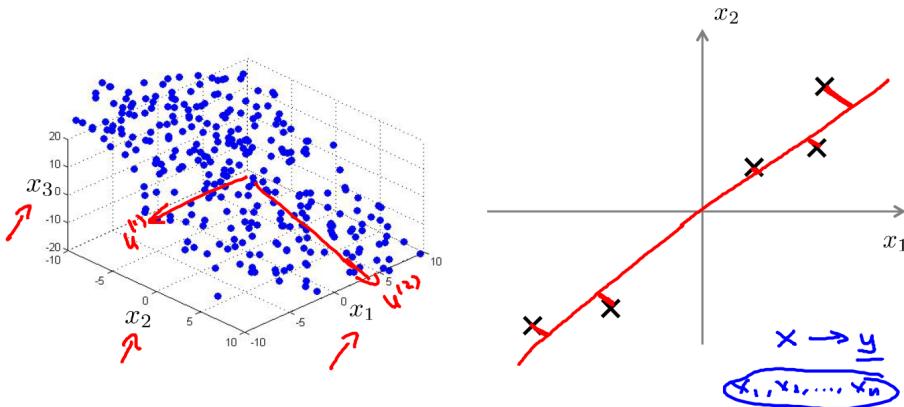


Figura 11.8: PCA vs regressão linear: variáveis com papéis diferentes.

Se os atributos estão em escalas diferentes, p.ex., x_1 = tamanho da casa e x_2 = número de cômodos, estão normalizar escala:

$$x_j^{(i)} \leftarrow \frac{x_j^{(i)} - \mu_j}{s_j}$$

em que s_j pode ser $\max(x_j) - \min(x_j)$, mas mais comumente é

$$s_j = \text{std}(x_j).$$

Vamos então reduzir de n para k dimensões.

Calculamos a “**matriz de covariância**”:

$$\Sigma = \frac{1}{m} \sum_{i=1}^n (x^{(i)}) (x^{(i)})^T$$

Note que cada parcela do somatório dessa matriz, chamada também de matriz *Sigma*, é o produto de um vetor $n \times 1$ ($x^{(i)}$) por um $1 \times n$ ($(x^{(i)})^T$), que resulta em uma matriz $n \times n$. A matriz Sigma tem portanto dimensões $n \times n$.

Em seguida, são calculados os “auto-vetores” da matriz Σ . No Octave:

$[U, S, V] = \text{svd}(\Sigma);$

SVD (*Singular Value Decomposition*) é um dos algoritmos para calcular auto-vetores. Poderia também usar a função *eig*. Mas o SVD é mais estável em casos gerais. Neste caso específico do PCA, porém, são equivalentes pois Σ é uma matriz *semi-positiva definida*.

Os auto-vetores são armazenados na matriz U :

$$U = \begin{bmatrix} | & | & & | \\ u^{(1)} & u^{(2)} & \dots & u^{(n)} \\ | & | & & | \end{bmatrix} \in \mathbb{R}^{n \times n}.$$

Definimos então uma matriz $U_{reduzida}$ com as k primeiras colunas de U e o vetor $z^{(i)}$ com dimensão reduzida é dado por

$$z^{(i)} = U_{reduzida}^T x^{(i)} = \begin{bmatrix} u^{(1)} & u^{(2)} & \dots & u^{(k)} \end{bmatrix}^T x^{(i)}.$$

Repare que U tem dimensões $n \times k$ e portanto $U_{reduzida}$ é $k \times n$. Como $x^{(i)}$ é $n \times 1$, temos que esse produto é portanto $k \times 1$ e, de fato, $z \in \mathbb{R}^k$, como desejado.

Note que os vetores $x^{(i)}$ podem ser representados em uma matriz X :

$$X = \begin{bmatrix} (x^{(1)})^T \\ \vdots \\ (x^{(m)})^T \end{bmatrix}$$

de modo que o cálculo de Σ pode ser vetorizado: $\boxed{\text{Sigma} = (1/m)*X'*X;}$

Em suma, após a normalização temos então

```
Sigma = (1/m)*X'*X;
[U,S,V] = svd(Sigma);
Ureduzida = U(:,1:k);
z = Ureduzida'*x;
```

IMPORTANTE: No caso do PCA, NÃO temos mais a convenção $x_0 = 1$, todos os atributos são tratados por igual.

11.5 Reconstrução

O dado reconstruído é o resultado da projeção de x sobre o vetor u (Figura 11.9).

Este valor $x_{aproximado}^{(i)} \approx x^{(i)}$ é dado por

$$x_{aproximado} = U_{reduzida} z^{(i)}.$$

Note que $U_{reduzida} \in \mathbb{R}^{n \times k}$ e $z^{(i)} \in \mathbb{R}^{k \times 1}$, de modo que este produto tem dimensões $n \times 1$ e portanto, como esperado, $x_{aproximado} \in \mathbb{R}^n$.

11.6 Escolhendo o Número de Componentes Principais

O valor k é chamado de **número de componentes principais**.

Definamos duas grandezas:

- Erro de projeção quadrático médio:

$$\frac{1}{m} \sum_{i=1}^m \|x^{(i)} - x_{aproximado}^{(i)}\|^2$$

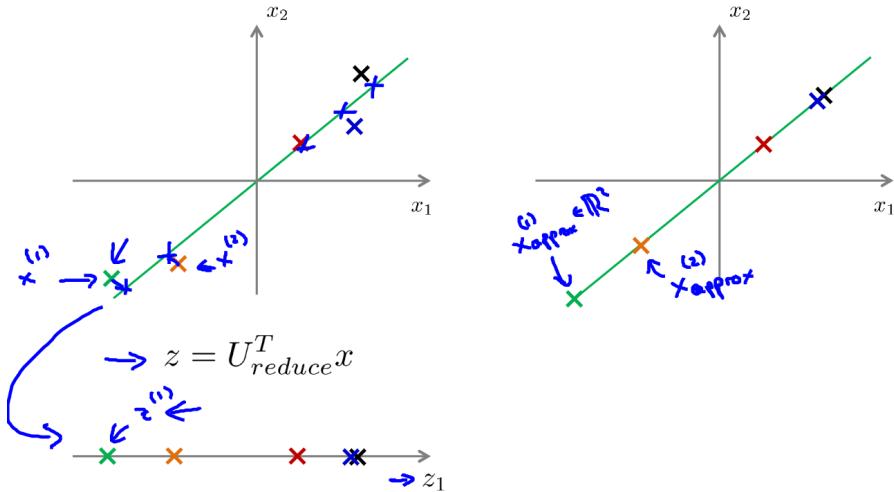


Figura 11.9: Vetor reconstruído a partir da representação comprimida.

- Variação total no dado:

$$\frac{1}{m} \sum_{i=1}^m \|x^{(i)}\|^2$$

Escolhemos k então como o menor valor tal que

$$\frac{\frac{1}{m} \sum_{i=1}^m \|x^{(i)} - x_{aproximado}^{(i)}\|^2}{\frac{1}{m} \sum_{i=1}^m \|x^{(i)}\|^2} \leq 0.01 \quad (1\%).$$

É comum que se diga que “99% da variância é retida”.

Em vez de ≤ 0.01 , usa-se também ≤ 0.05 (5%) e ≤ 0.10 (10%), que correspondem, respectivamente, a 95% e 90% da variância retida.

O procedimento geral poderia ser então:

```

Testar PCA com  $k = 1$ 
Calcular  $U_{reduzida}, z^{(1)}, z^{(2)}, \dots, z^{(m)}, x_{aproximado}^{(1)}, \dots, x_{aproximado}^{(m)}$ 
Checar se  $\frac{\frac{1}{m} \sum_{i=1}^m \|x^{(i)} - x_{aproximado}^{(i)}\|^2}{\frac{1}{m} \sum_{i=1}^m \|x^{(i)}\|^2} \leq 0.01$ 
Repetir esse processo para  $k = 2, 3, 4, \dots$ 
```

Porém, existe um modo muito mais eficiente. Quando rodamos

```
[U,S,V] = svd(Sigma);
```

a matriz S é diagonal e a checagem da variância equivale a

$$1 - \frac{\sum_{i=1}^k S_{ii}}{\sum_{i=1}^n S_{ii}} \leq 0.01$$

ou equivalentemente

$$\frac{\sum_{i=1}^k S_{ii}}{\sum_{i=1}^n S_{ii}} \geq 0.99$$

e o algoritmo então é

$[U, S, V] = svd(\Sigma)$
 Tome o menor valor de k para o qual
 $\frac{\sum_{i=1}^k S_{ii}}{\sum_{i=1}^n S_{ii}} \geq 0.99$
 (99% da variância retida)

11.7 Aplicação do PCA

Um outro uso muito importante de PCA é para acelerar algoritmos de aprendizado supervisionado.

Partimos de um conjunto de exemplos

$$(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})$$

e olhamos só para os atributos. Vamos supor que tenhamos o conjunto sem rótulos

$$\begin{aligned} x^{(1)}, x^{(2)}, \dots, x^{(m)} &\in \mathbb{R}^{(10000)} \\ \downarrow PCA \\ z^{(1)}, z^{(2)}, \dots, z^{(m)} &\in \mathbb{R}^{1000} \end{aligned}$$

e então nosso novo conjunto de dados será

$$(z^{(1)}, y^{(1)}), (z^{(2)}, y^{(2)}), \dots, (z^{(m)}, y^{(m)}).$$

$x^{(i)}$ poderia ser, por exemplo, os pixels de uma imagem 100×100 e poderíamos aplicar uma função de hipótese como da regressão logística sobre os atributos z :

$$h_\theta(z) = \frac{1}{1 + e^{-\theta^T z}}.$$

IMPORTANTE: O mapeamento PCA $x^{(i)} \rightarrow z^{(i)}$ deve ser definido APENAS sobre o conjunto de treinamento, ou seja, a matriz $U_{reduzida}$ e mesmo os parâmetros de normalização (média, desvio) devem ser calculados todos sobre o conjunto de treino apenas.

Somente após tomado esse cuidado é que, aí sim, esse mapeamento pode ser aplicado igualmente aos exemplos de validação e teste.

O algoritmo geral é

Receber o conjunto de treino $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})\}$
 Rodar PCA para reduzir a dimensão de $x^{(i)}$ gerando $z^{(i)}$
 Treinar a regressão logística (ou qualquer outro algoritmo) sobre
 $\{(z^{(1)}, y^{(1)}), \dots, (z^{(m)}, y^{(m)})\}$
 Mapear $x_{teste}^{(i)}$ para $z_{teste}^{(i)}$
 Rodar $h_\theta(z)$ em $\{(z_{teste}^{(1)}, y_{teste}^{(1)}), \dots, (z_{teste}^{(m)}, y_{teste}^{(m)})\}$

Temos em geral os seguintes usos então de PCA:

- Compressão:
 - Reduzir o consumo de memória/armazenamento
 - Acelerar o algoritmo de aprendizado

Em ambos os casos, escolhe-se k pela porcentagem da variância retida

- Visualização: $k = 2$ ou $k = 3$

11.7.1 Maus usos de PCA

Um uso inadequado de PCA é para prevenir *overfit*. A ideia é que o uso de menos atributos fica menos propenso a *overfit*. Isso pode até funcionar, mas não é uma boa prática. O problema é que o PCA não leva em conta o rótulo y e um atributo importante pode desaparecer.

A melhor estratégia para reduzir *overfit* é a **regularização**:

$$\underset{\theta}{\text{minimize}} \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2.$$

Fato é que o PCA é usado às vezes quando não deveria.

A ideia é sempre, antes de qualquer coisa, rodar todo o algoritmo de aprendizado sobre o dado original/bruto $x^{(i)}$.

E então só se estiver com problema de desempenho de CPU/memória considerar usar o dado $z^{(i)}$ do PCA.

Capítulo 12

Detecção de Anomalia

12.1 Detecção de Anomalia - Motivação

EX.: Motores de avião - grande maioria funciona corretamente, uma minoria não funciona.

Atributos:

- x_1 : calor gerado
- x_2 : intensidade de vibração
- ...

Temos um conjunto de treino $\{x^{(1)}, x^{(2)}, \dots, x^{(m)}\}$ (pontos vermelhos na Figura 12.1).

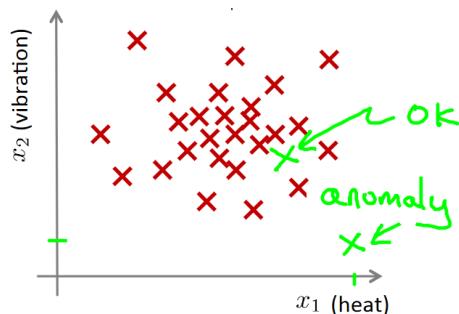


Figura 12.1: Detecção de anomalia.

Chega então um novo exemplo de teste x_{teste} (ponto verde). Este vai ser considerado anomalia se estiver distante dos pontos vermelhos.

A técnica para identificar se x_{teste} é uma anomalia baseia-se em uma **densidade de probabilidade**.

Definimos um modelo de probabilidade $p(x)$ de modo que

- $p(x) < \epsilon \Rightarrow$ anomalia

- $p(x) \geq \epsilon \Rightarrow$ OK

EXEMPLOS:

- Detecção de fraude:

- $x^{(i)}$ = atributos de atividade do usuário i , p.ex., número de logins, número de páginas visitadas, ritmo de digitação, etc.
- Obter modelo $p(x)$ a partir dos dados
- Identificar usuários não usuais checando se $p(x) < \epsilon$

- Linha de produção de uma fábrica: exemplo do motor de avião

- Monitoramento de computadores em um *data center*

- $x^{(i)}$ = atributos da máquina i , p.ex., x_1 = uso de memória, x_2 = número de acessos ao disco/segundo, x_3 = carga da CPU, x_4 = carga da CPU / tráfego de rede.
- Checar se $p(x) < \epsilon$

12.2 Distribuição Gaussiana

Seja $x \in \mathbb{R}$. Se x segue uma distribuição Gaussiana (Normal) com média μ e variância σ^2 , denotamos

$$x \sim \mathcal{N}(\mu, \sigma^2),$$

em que \sim se lê como “distribuído como” e σ (sem o quadrado) é chamado de “desvio padrão”.

Temos então que a probabilidade é dada por

$$p(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x - \mu)^2}{2\sigma^2}\right),$$

em que a notação $p(x; \mu, \sigma^2)$ indica que $p(x)$ é **parametrizado** por μ e σ^2 .

A Figura 12.2 mostra a curva de $p(x)$, com forma de sino. Note que μ determina o centro/posição do pico da curva e σ define a largura.

Veja mais exemplos na Figura 12.3. Note que quanto maior for σ , mais larga é a curva. Note também que a área sob a curva é sempre 1, como uma propriedade de toda distribuição de probabilidade, de modo que uma curva mais estreita será também mais alta, e vice-versa.

Os parâmetros μ e σ são obtidos a partir do conjunto de treinamento $\{x^{(1)}, x^{(2)}, \dots, x^{(m)}\}$, $x^{(i)} \in \mathbb{R}$. Assim:

$$\mu = \frac{1}{m} \sum_{i=1}^m x^{(i)}$$

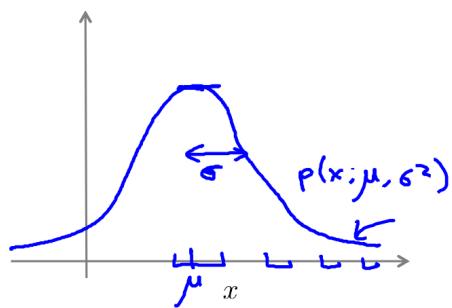
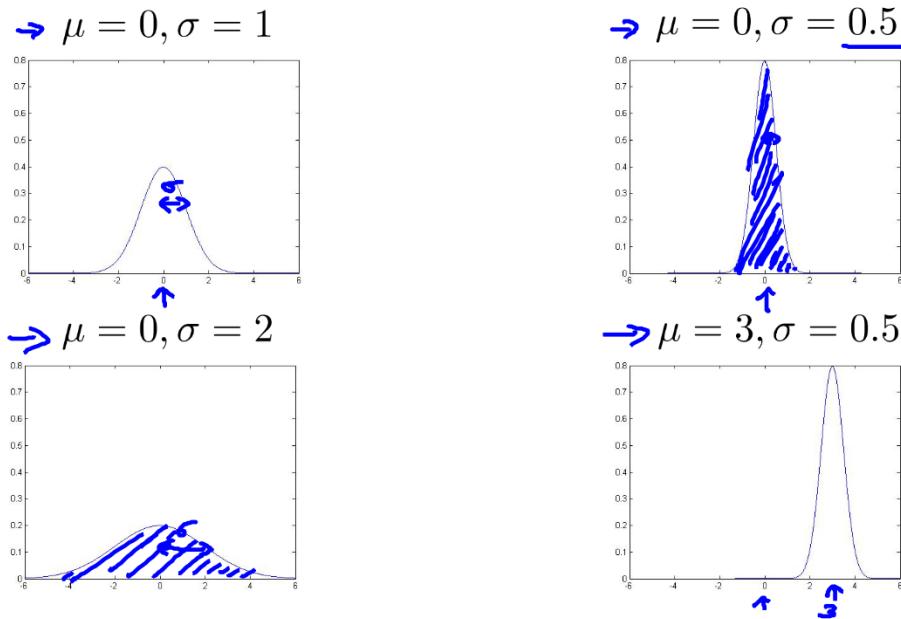


Figura 12.2: Distribuição Gaussiana.

Figura 12.3: Distribuição Gaussiana para diferentes valores de μ e σ .

$$\sigma^2 = \frac{1}{m} \sum_{i=1}^m (x^{(i)} - \mu)^2$$

NOTA: Existe na estatística uma outra definição de σ^2 que usa $\frac{1}{m-1}$. Essas têm propriedades teóricas diferentes, mas, para fins de machine learning, em que usualmente m é grande, não há diferença relevante e o comum em machine learning é a definição com $\frac{1}{m}$.

12.3 Algoritmo

A estimativa dos parâmetros da densidade de probabilidade é feita sempre sobre o conjunto de treinamento.

Seja então o treino $\{x^{(1)}, \dots, x^{(m)}\}$, com $x \in \mathbb{R}^n$.

A primeira abordagem que veremos assume que as distribuições de cada atributo são independentes (sem correlação). Mesmo quando isso não é totalmente verdade, esse modelo ainda pode funcionar bem.

Temos então cada atributo seguindo uma distribuição Gaussiana própria:

$$\begin{aligned} x_1 &\sim \mathcal{N}(\mu_1, \sigma_1^2) \\ x_2 &\sim \mathcal{N}(\mu_2, \sigma_2^2) \\ x_3 &\sim \mathcal{N}(\mu_3, \sigma_3^2) \\ &\dots \end{aligned}$$

e a distribuição geral (conjunta) é o produto:

$$p(x) = p(x_1; \mu_1, \sigma_1^2)p(x_2; \mu_2, \sigma_2^2)p(x_3; \mu_3, \sigma_3^2) \cdots p(x_n; \mu_n, \sigma_n^2) = \prod_{j=1}^n p(x_j; \mu_j, \sigma_j^2).$$

O algoritmo geral é o seguinte:

1. Escolher atributos x_i que poderiam indicar anomalias
2. Ajustar parâmetros $\mu_1, \dots, \mu_n, \sigma_1^2, \dots, \sigma_n^2$:

$$\begin{aligned} \mu_j &= \frac{1}{m} \sum_{i=1}^m x_j^{(i)} \\ \sigma_j^2 &= \frac{1}{m} \sum_{i=1}^m (x_j^{(i)} - \mu_j)^2 \end{aligned}$$

3. Dado um novo exemplo x , calcular $p(x)$:

$$p(x) = \prod_{j=1}^n p(x_j; \mu_j, \sigma_j^2) = \prod_{j=1}^n \frac{1}{\sqrt{2\pi}\sigma_j} \exp\left(-\frac{(x_j - \mu_j)^2}{2\sigma_j^2}\right)$$

Temos uma anomalia se $p(x) < \epsilon$.

OBS.: O Passo 2 pode ser vetorizado:

$$\mu = \begin{bmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_n \end{bmatrix} = \frac{1}{m} \sum_{i=1}^m x^{(i)}.$$

Considere o exemplo na Figura 12.4.

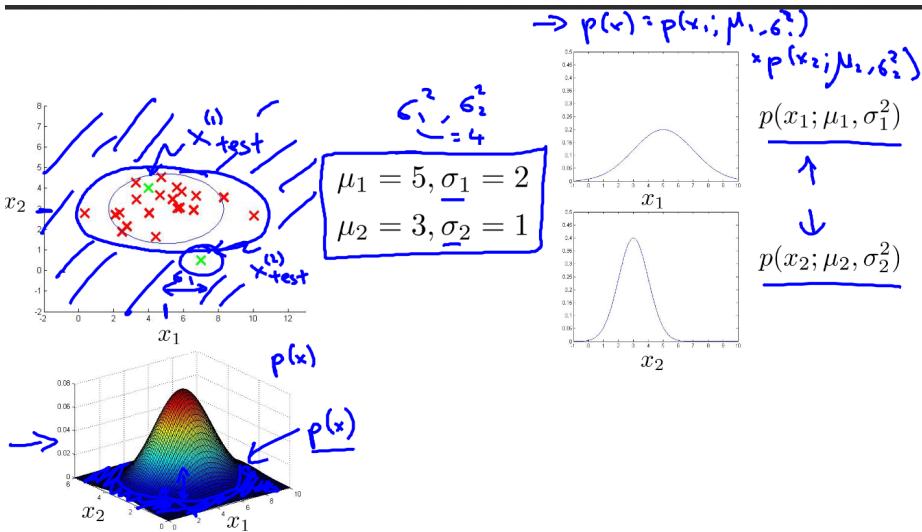


Figura 12.4: Exemplo de detecção de anomalia.

Vamos considerar $\epsilon = 0.02$ e quando calculamos temos

$$p(x_{\text{teste}}^{(1)}) = 0.0426 \geq \epsilon$$

$$p(x_{\text{teste}}^{(2)}) = 0.0021 < \epsilon,$$

de modo que $x^{(2)}$ é uma anomalia. Note que visualmente ele está afastado dos pontos vermelhos dos exemplos “normais”.

12.4 Desenvolvimento e Avaliação de um Sistema de Detecção de Anomalia

Como já vimos, a tomada de decisão em um algoritmo de aprendizado (quais atributos usar, etc.) é muito mais fácil quando temos uma forma de avaliar o algoritmo.

Vamos assumir que tenhamos exemplos rotulados como $y = 0$ se for normal e $y = 1$ se for anômalo.

Em nosso conjunto de treinamento $\{x^{(1)}, x^{(2)}, \dots, x^{(m)}\}$ colocamos apenas exemplos normais / não anômalos.

Separamos ainda um conjunto de validação $\{(x_{cv}^{(1)}, y_{cv}^{(1)}), \dots, (x_{cv}^{(m_{cv})}, y_{cv}^{(m_{cv})})\}$ e um de teste $\{(x_{teste}^{(1)}, y_{teste}^{(1)}), \dots, (x_{teste}^{(m_{teste})}, y_{teste}^{(m_{teste})})\}$.

EXEMPLO: Problema dos motores de avião, com

- 10000 motores bons (normais)
- 20 motores falhos (anômalos) (esse é sempre um número comparativamente pequeno, algo como na faixa $2 \sim 50$)

Um protocolo usual para divisão treino/validação/teste é o seguinte:

- Treino: 6000 motores bons
- Validação: 2000 motores bons ($y = 0$) e 10 anômalos ($y = 1$)
- Teste: 2000 motores bons ($y = 1$) e 10 anômalos ($y = 1$)

Uma alternativa menos recomendada que algumas pessoas fazem seria repetir os motores bons da validação no teste:

- Treino: 6000 motores bons
- Validação: 4000 motores bons ($y = 0$) e 10 anômalos ($y = 1$)
- Teste: 4000 motores bons ($y = 1$) e 10 anômalos ($y = 1$)

A avaliação do algoritmo consiste então no seguinte:

- Ajuste o modelo $p(x)$ sobre o treinamento $\{x^{(1)}, \dots, x^{(m)}\}$
- Para um exemplo x de validação/teste, prever

$$y = \begin{cases} 1 & \text{se } p(x) < \epsilon \text{ (anomalia)} \\ 0 & \text{se } p(x) \geq \epsilon \text{ (normal)} \end{cases}$$

- Repare que temos classes altamente desbalanceadas e as métricas de avaliação nesse caso já são nossas conhecidas:
 - Verdadeiro positivo, falso positivo, falso negativo, verdadeiro negativo
 - *Precision/Recall*
 - F_1 -score
- Pode-se ainda usar o conjunto de validação para escolher ϵ e os atributos ideais

12.5 Detecção de Anomalia vs Aprendizado Supervisionado

Temos basicamente as seguintes diferenças:

Detecção de anomalia	Aprendizado supervisionado
Número muito pequeno de exemplos positivos ($y = 1$) (normalmente $0 \sim 20$). Insuficiente para treinar um algoritmo supervisionado.	Número grande de exemplos positivos e negativos.
Grande número de exemplos negativos ($y = 0$). Usados para formar $p(x)$.	
Muitos “tipos” diferentes de anomalias. Difícil para um algoritmo aprender como é uma anomalia a partir dos exemplos positivos. Anomalias futuras podem não se parecer em nada com os exemplos de anomalia vistos até então.	Exemplos positivos suficientes para que se tenha uma ideia de como um exemplo positivo deve ser no futuro. EX.: Existem muitos tipos de <i>spam</i> , mas esses tipos podem ser incluídos no treinamento.

EXEMPLOS:

Detecção de anomalia	Aprendizado supervisionado
Detecção de fraude	Classificação de <i>spam</i>
Linha de produção (ex. motores de avião)	Previsão do tempo (ensolarado/chuvoso/etc.)
Monitoramento de máquinas em um <i>data center</i>	Classificação de câncer
:	:

ATENÇÃO: Repare que, por exemplo, o problema de detecção de fraude em um grande banco, com muitos exemplos positivos, pode ser abordado como de aprendizado supervisionado.

12.6 Escolhendo os Atributos

A técnica vista até aqui presume que cada atributo x_j é Gaussiano. Para verificar isso, plotamos o histograma do atributo e checamos se o formato é de uma Gaussiana. Na Figura 2.4 esquerda temos um exemplo de atributo Gaussiano.

No centro, não é Gaussiano. Neste caso, faz-se uma transformação do atributo. Uma transformação muito usada é $\log(x)$, ou ainda $\log(x + C)$ para alguma constante C ou ainda $x^{\frac{1}{2}}$, $x^{\frac{1}{3}}$, etc. Esse expoente e a constante C são testados até que se encontre um que deixa o atributo com formato de Gaussiana (Figura 12.5 direita).

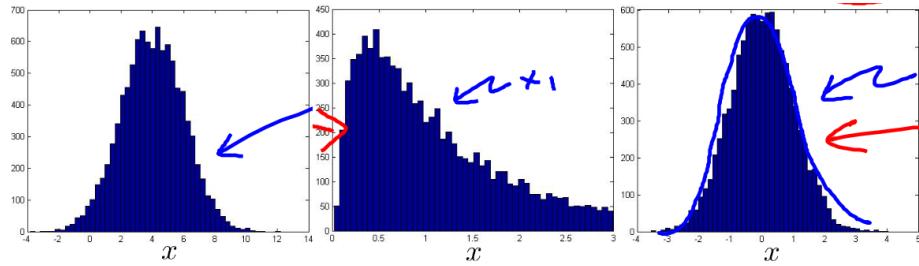


Figura 12.5: Atributo Gaussiano (esquerda), não Gaussiano (centro) e transformado ($\log(x)$) (direita).

12.6.1 Análise de erro

Já vimos que analisar os exemplos classificados erroneamente podem ajudar a melhorar o algoritmo.

Ideia parecida ocorre na detecção de anomalia.

Queremos

- $p(x)$ grande para exemplos x normais
- $p(x)$ pequeno para exemplos x anômalos

O problema mais comum é termos $p(x)$ comparável para exemplos normais e anômalos (p.ex. ambos grandes).

Criar atributos pode ser uma solução.

Escolher atributos que apresentem valores excepcionalmente grandes ou pequenos no caso de anomalia.

EXEMPLO: Computadores em um *data center*.

- x_1 = uso de memória do computador
- x_2 = número de acessos ao disco/segundo
- x_3 = carga da CPU
- x_4 = tráfego de rede

Agora imagine que se identificou que um indicativo consistente de anomalia é quando há grande carga na CPU ao mesmo tempo em que um pequeno tráfego de rede, o que indica, por exemplo, que a máquina entrou em *loop* infinito.

Cria-se então um atributo

$$x_5 = \frac{\text{carga de CPU}}{\text{tráfego de rede}}$$

ou ainda

$$x_6 = \frac{(\text{carga de CPU})^2}{\text{tráfego de rede}},$$

os quais apresentarão valores muito altos em caso de anomalia.

12.7 Distribuição Gaussiana Multivariada

Voltemos ao exemplo do *data center*. Temos um ponto anômalo com $x_1 = 0.5$ e $x_2 = 1.5$ (Figura 12.6 esquerda). Quando olhamos para $p(x_1)$ e $p(x_2)$ para esses valores temos uma altura não tão pequena (Figura 12.6 direita), de modo que $p(x)$ não será tão pequeno e este ponto não será detectado como anomalia.

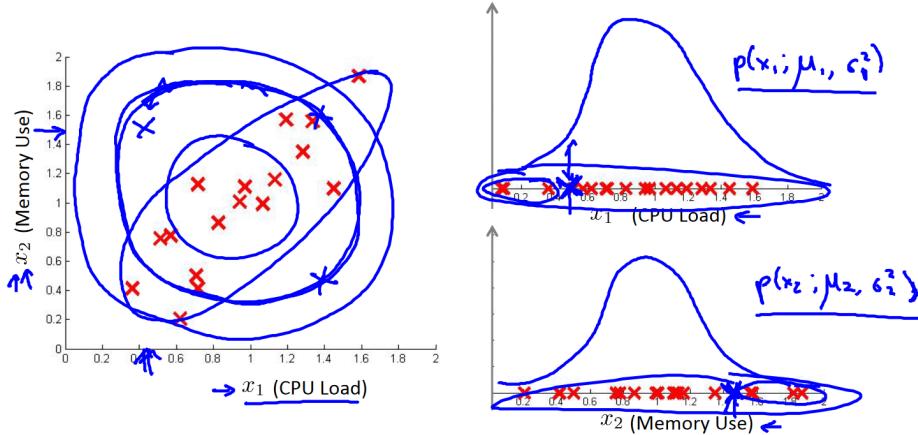


Figura 12.6: Atributos correlacionados na detecção de anomalia.

Isso ocorre porque x_1 e x_2 têm uma relação linear (são correlacionados).

Neste caso, NÃO se deve modelar $p(x_1), p(x_2), \dots$ separadamente, mas sim de uma vez só.

Para $x \in \mathbb{R}^n$, definimos $\mu \in \mathbb{R}^n$ e $\Sigma \in \mathbb{R}^{n \times n}$ (matriz de covariância) e então:

$$p(x) = (x; \mu, \Sigma) = \frac{1}{(2\pi)^{n/2} |\Sigma|^{1/2}} \exp \left(-\frac{1}{2} (x - \mu)^T \Sigma^{-1} (x - \mu) \right),$$

em que $|\Sigma|$ é o determinante de Σ . No Octave:

`det(Sigma)`

NOTA: Essa matriz de covariância é exatamente a mesma que vimos na análise de componentes principais.

As Figuras 12.7-12.12 ilustram os gráficos da distribuição Gaussiana multivariada para diferentes valores de μ e Σ .

Os casos em que Σ é uma matriz diagonal com valores iguais são os mais simétricos, sendo que quanto maior esse valor, mais larga a superfície (Figura 12.7).

Σ ainda diagonal, mas com valores diferentes, implica em uma superfície mais estreita/larga na direção do atributo com menor/maior valor na diagonal de Σ (Figuras 12.8 e 12.9).

Já a presença de valores não nulos fora da diagonal principal implicam em uma superfície rotacionada. Quanto maiores as magnitudes desses valores fora da diagonal, mais estreita é a superfície (Figura 12.10). Valores negativos fora da diagonal implicam em uma rotação no sentido de correlação negativa (Figura 12.11).

Por fim, o vetor de média μ determina a posição do centro/pico da superfície (Figura 12.12).

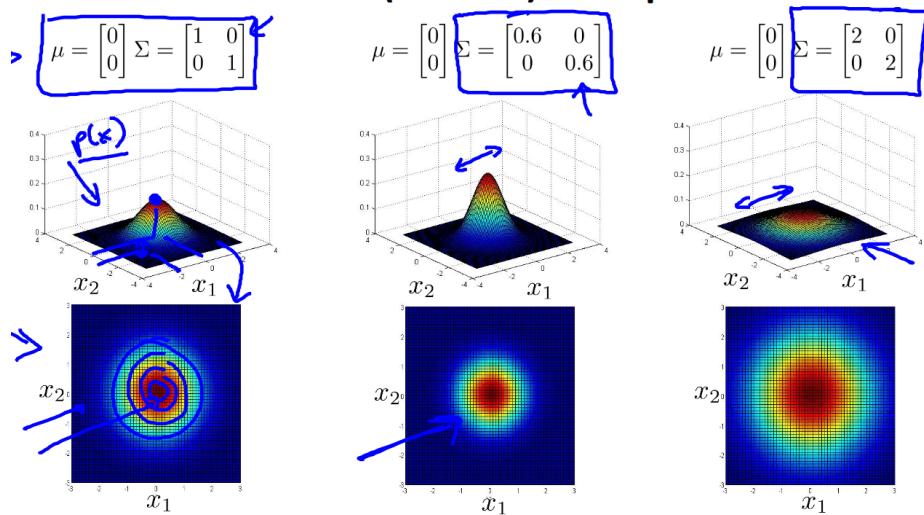


Figura 12.7: Distribuição Gaussiana multivariada (exemplos).

12.8 Detecção de Anomalia Usando a Distribuição Gaussiana Multivariada

Apenas lembrando que, dados os parâmetros $\mu \in \mathbb{R}^n$ e $\Sigma \in \mathbb{R}^{n \times n}$, temos

$$p(x; \mu, \Sigma) = \frac{1}{(2\pi)^{\frac{n}{2}} |\Sigma|^{\frac{1}{2}}} \exp \left(-\frac{1}{2}(x - \mu)^T \Sigma^{-1} (x - \mu) \right).$$

Mas, afinal, como são obtidos os parâmetros μ e Σ ?

Ambos são ajustados a partir do conjunto de treino $\{x^{(1)}, x^{(2)}, \dots, x^{(m)}\}$:

$$\mu = \frac{1}{m} \sum_{i=1}^m x^{(i)}$$

$$\Sigma = \frac{1}{m} \sum_{i=1}^m (x^{(i)} - \mu)(x^{(i)} - \mu)^T.$$

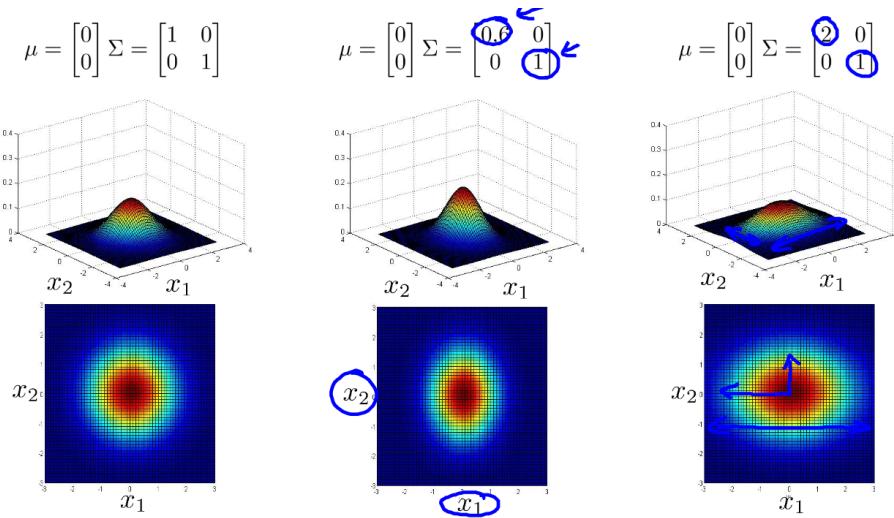


Figura 12.8: Distribuição Gaussiana multivariada (exemplos).

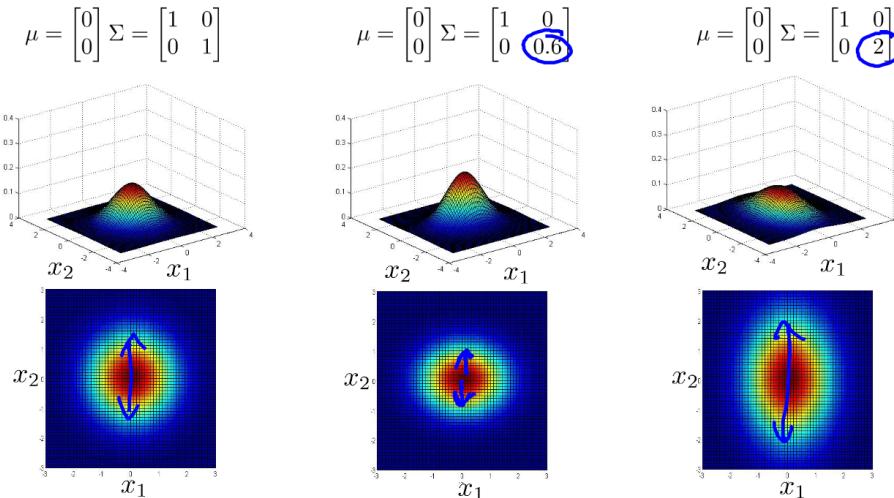


Figura 12.9: Distribuição Gaussiana multivariada (exemplos).

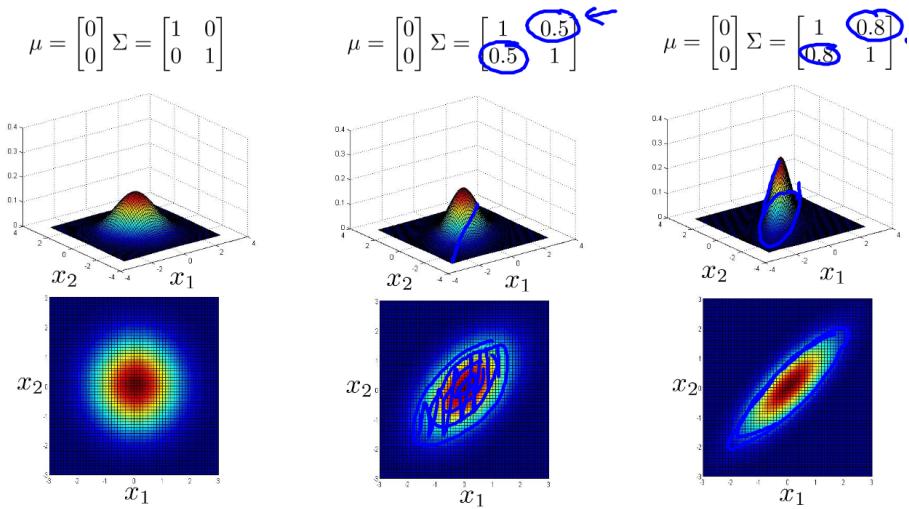


Figura 12.10: Distribuição Gaussiana multivariada (exemplos).

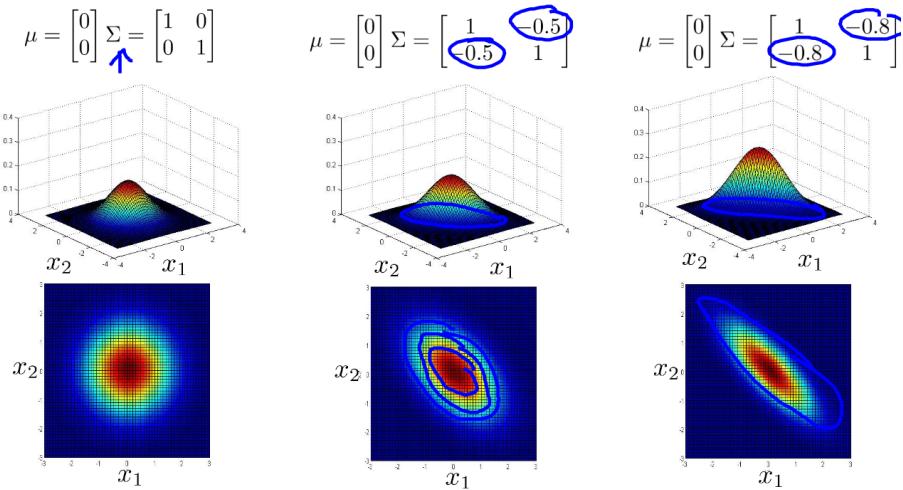


Figura 12.11: Distribuição Gaussiana multivariada (exemplos).

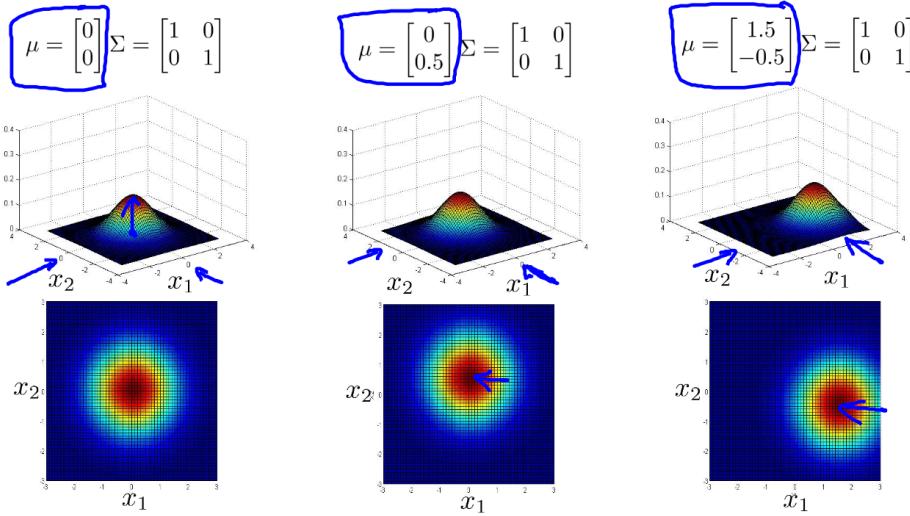


Figura 12.12: Distribuição Gaussiana multivariada (exemplos).

Note que essa expressão de Σ é exatamente equivalente à que usamos no PCA.

Seguimos então os seguintes passos para detecção de anomalia com a distribuição Gaussiana multivariada:

1. Ajustar o modelo $p(x)$ por

$$\begin{aligned}\mu &= \frac{1}{m} \sum_{i=1}^m x^{(i)} \\ \Sigma &= \frac{1}{m} \sum_{i=1}^m (x^{(i)} - \mu)(x^{(i)} - \mu)^T\end{aligned}$$

2. Dado um novo exemplo x , calculamos

$$p(x) = \frac{1}{(2\pi)^{\frac{n}{2}} |\Sigma|^{\frac{1}{2}}} \exp \left(-\frac{1}{2} (x - \mu)^T \Sigma^{-1} (x - \mu) \right).$$

Considerar x anômalo se $p(x) < \epsilon$.

12.8.1 Relação com o modelo original

O modelo original

$$p(x) = p(x_1; \mu_1, \sigma_1^2) \times p(x_2; \mu_2, \sigma_2^2) \times p(x_3; \mu_3, \sigma_3^2) \times \cdots \times p(x_n; \mu_n, \sigma_n^2)$$

corresponde ao multivariado

$$p(x; \mu, \Sigma) = \frac{1}{(2\pi)^{\frac{n}{2}} |\Sigma|^{\frac{1}{2}}} \exp \left(-\frac{1}{2} (x - \mu)^T \Sigma^{-1} (x - \mu) \right)$$

se Σ for a matriz diagonal:

$$\Sigma = \begin{bmatrix} \sigma_1^2 & 0 & \cdots & 0 \\ 0 & \sigma_2^2 & \cdots & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & \cdots & \sigma_n^2 \end{bmatrix}$$

Temos em geral as seguintes diferenças:

Modelo original	Gaussiana multivariada
$p(x_1; \mu_1, \sigma_1^2) \times \cdots \times p(x_n; \mu_n, \sigma_n^2)$	$p(x; \mu, \Sigma) = \frac{1}{(2\pi)^{\frac{n}{2}} \Sigma ^{\frac{1}{2}}} \exp\left(-\frac{1}{2}(x - \mu)^T \Sigma^{-1} (x - \mu)\right)$
Criar atributos manualmente que capturem anomalias em que x_1 e x_2 assumem combinações raras de valores, p.ex., $x_3 = \frac{x_1}{x_2} = \frac{\text{carga da CPU}}{\text{memória}}$	Captura correlações automaticamente
Mais barato computacionalmente (escala melhor para n grande, p.ex. $n = 10000$ ou $n = 100000$)	Mais caro computacionalmente (calcular Σ^{-1} é muito caro para n grande)
OK mesmo se m (tamanho do treinamento) for pequeno	Exige $m > n$, senão Σ não é invertível (singular) Ideal é $m \geq 10n$, já que Σ tem $\frac{n^2}{2}$ valores a serem determinados (já que é simétrica) Para Σ ser invertível é necessário também que não haja atributos redundantes (linearmente dependentes).

Capítulo 13

Sistemas de Recomendação

13.1 Sistemas de Recomendação - Fomulação

Sistemas de recomendação são bastante populares, especialmente na indústria.
EX.: recomendação de produtos na Amazon, filmes na Netflix, etc.

Outra característica interessante é a habilidade desses sistemas de identificar automaticamente os atributos. Como temos visto, atributos têm papel crucial em sistemas de aprendizado de máquinas.

Considere o problema de estimar a nota de um filme. A Netflix tem um sistema de estrelas, que vai de 1 a 5 estrelas, mas aqui vamos considerar de 0 a 5 estrelas.

Imagine que tenhamos os seguintes usuários, filmes e notas:

Filme	Alice	Bob	Carol	Dave
Love at least	5	5	0	0
Romance forever	5	?	?	0
Cute puppies of love	?	4	0	?
Nonstop car chases	0	0	5	4
Swords vs. karate	0	0	5	?

O problema consiste então em estimar a nota de usuários que não deram essa nota e aparecem com ‘?’ na tabela.

Definimos as seguintes variáveis:

$$\begin{aligned} n_u &= \text{número de usuários} \\ n_m &= \text{número de filmes} \\ r(i, j) &= 1 \text{ se o usuário } j \text{ deu nota para o filme } i, 0 \text{ caso contrário} \\ y^{(i,j)} &= \text{nota dada pelo usuário } j \text{ para o filme } i \text{ (definido apenas se } r(i, j) = 1) \end{aligned}$$

Na tabela de exemplo então $n_u = 4$, $n_m = 5$ e todos os campos com ‘?’ terão $r(i, j) = 0$ e os demais $r = 1$. Os valores de $y^{(i,j)}$ estão na faixa $0, \dots, 5$.

Note que os 3 primeiros filmes são românticos e os 2 últimos são de ação. Nota-se então uma preferência de Alice e Bob por romance e de Carol e Dave

por ação.

13.2 Recomendações Baseadas em Conteúdo

Na recomendação baseada em conteúdo, a ideia é criar um modelo de regressão linear para cada usuário. Vamos precisar então de um vetor de parâmetros para cada usuário e um vetor de atributos para cada filme.

Definimos então:

$$\begin{aligned}\theta^{(j)} &= \text{vetor de parâmetros para o usuário } j \\ x^{(i)} &= \text{vetor de atributos para o filme } i \\ m^{(j)} &= \text{número de filmes avaliados pelo usuário } j\end{aligned}$$

Então, para o usuário j e filme i , a nota predita é dada por

$$(\theta^{(j)})^T(x^{(i)}).$$

Assim como na regressão linear, convencionamos $x_0 = 1$, de modo que $x^{(i)} \in \mathbb{R}^{n+1}$ e $\theta^{(j)} \in \mathbb{R}^{n+1}$.

Vamos supor no exemplo anterior que tenhamos 2 atributos ($n = 2$) para compor o vetor $x^{(i)}$: x_1 (nível de romance) e x_2 (nível de ação).

Filme	Alice (1)	Bob (2)	Carol (3)	Dave (4)	x_1 (romance)	x_2 (ação)
Love at least (1)	5	5	0	0	0.9	0
Romance forever (2)	5	?	?	0	1.0	0.01
Cute puppies of love (3)	?	4	0	?	0.99	0
Nonstop car chases (4)	0	0	5	4	0.1	1.0
Swords vs. karate (5)	0	0	5	?	0	0.9

Vamos estimar, por exemplo, a nota dada pelo usuário 1 (Alice) para o filme 3 (Cute puppies of love). Lembrando que $x_0 = 1$ por convenção, temos

$$x^{(3)} = \begin{bmatrix} 1 \\ 0.99 \\ 0 \end{bmatrix}$$

Vamos supor que

$$\theta^{(1)} = \begin{bmatrix} 0 \\ 5 \\ 0 \end{bmatrix},$$

então $y^{(3)}$ será predito por

$$(\theta^{(1)})^T x^{(3)} = 5 \cdot 0.99 = 4.95.$$

Os parâmetros $\theta^{(j)}$ são aprendidos como no modelo de regressão linear já visto, que em sua versão regularizada é:

$$\underset{\theta^{(j)}}{\text{minimize}} \frac{1}{2m^{(j)}} \sum_{i:r(i,j)=1} ((\theta^{(j)})^T(x^{(i)}) - y^{(i,j)})^2 + \frac{\lambda}{2m^{(j)}} \sum_{k=1}^n (\theta_k^{(j)})^2.$$

Repare que a soma inclui apenas casos em que o usuário deu nota ($r(i, j) = 1$). Outro ponto é que em sistemas de recomendação não se inclui $m^{(j)}$ no denominador (para efeitos da minimização não faz diferença), de modo que ficamos então com

$$\underset{\theta^{(j)}}{\text{minimize}} \frac{1}{2} \sum_{i:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{k=1}^n (\theta_k^{(j)})^2.$$

Porém, temos n_u usuários e a minimização deve ser feita sobre todos eles, de modo que aprendemos $\theta^{(1)}, \theta^{(2)}, \dots, \theta^{(n_u)}$ por

$$\underset{\theta^{(1)}, \dots, \theta^{(n_u)}}{\text{minimize}} \frac{1}{2} \sum_{j=1}^{n_u} \sum_{i:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^n (\theta_k^{(j)})^2.$$

Esta função minimizada é a nossa função de custo (objetivo) $J(\theta^{(1)}, \dots, \theta^{(n_u)})$ e podemos minimizá-la por gradiente descendente, exatamente como fizemos na regressão linear:

$$\begin{aligned} \theta_k^{(j)} &:= \theta_k^{(j)} - \alpha \sum_{i:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)}) x_k^{(i)} && (\text{para } k = 0) \\ \theta_k^{(j)} &:= \theta_k^{(j)} - \alpha \left(\sum_{i:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)}) x_k^{(i)} + \lambda \theta_k^{(j)} \right) && (\text{para } k \neq 0). \end{aligned}$$

A diferença é apenas a ausência do fator $\frac{1}{m^{(j)}}$. Como antes, o somatório que multiplica α é a derivada $\frac{\partial}{\partial \theta_k^{(j)}} J(\theta^{(1)}, \dots, \theta^{(n_u)})$.

13.3 Filtragem Colaborativa

Mas afinal como calculamos os valores dos atributos x_1, x_2, \dots ?

Podíamos pedir para cada usuário rotular todos os filmes como de romance, ação, comédia, etc. Mas isso não é viável na prática.

Podemos então aprender esses atributos automaticamente. Se conhecermos os vetores de parâmetros $\theta^{(j)}$, podemos obter $x^{(i)}$ minimizando o erro da previsão. Se na tabela anterior, por exemplo:

$$\theta^{(1)} = \begin{bmatrix} 0 \\ 5 \\ 0 \end{bmatrix}, \quad \theta^{(2)} = \begin{bmatrix} 0 \\ 5 \\ 0 \end{bmatrix}, \quad \theta^{(3)} = \begin{bmatrix} 0 \\ 0 \\ 5 \end{bmatrix}, \quad \theta^{(4)} = \begin{bmatrix} 0 \\ 0 \\ 5 \end{bmatrix},$$

então, por exemplo, $x^{(1)}$ deve satisfazer

$$\begin{aligned} (\theta^{(1)})^T x^{(1)} &\approx 5 \\ (\theta^{(2)})^T x^{(1)} &\approx 5 \\ (\theta^{(3)})^T x^{(1)} &\approx 0 \\ (\theta^{(4)})^T x^{(1)} &\approx 0 \end{aligned}.$$

Dado $\theta^{(1)}, \dots, \theta^{(n_u)}$, nosso problema de otimização para obter $x^{(i)}$ é formalizado então como

$$\underset{x^{(i)}}{\text{minimize}} \frac{1}{2} \sum_{j:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{k=1}^n (x_k^{(i)})^2.$$

Porém temos n_m filmes e os vetores de atributo são aprendidos simultaneamente:

$$\underset{x^{(1)}, \dots, x^{(n_m)}}{\text{minimize}} \frac{1}{2} \sum_{i=1}^{n_m} \sum_{j:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^n (x_k^{(i)})^2.$$

A ideia geral dessa abordagem, chamada **filtragem colaborativa**, é então

- Dado $x^{(1)}, \dots, x^{(n_m)}$ (e as notas dos filmes), estimar $\theta^{(1)}, \dots, \theta^{(n_u)}$
- Dado $\theta^{(1)}, \dots, \theta^{(n_u)}$, estimar $x^{(1)}, \dots, x^{(n_m)}$

Porém, uma coisa depende da outra, como resolver isso?

R.: Chutar θ inicial pequeno aleatório e ir iterativamente calculando $\theta \rightarrow x \rightarrow \theta \rightarrow x \rightarrow \theta \rightarrow x \rightarrow \dots$.

O método se chama filtragem colaborativa porque todos os usuários colaboraram para a indicação de determinado filme.

13.4 Filtragem Colaborativa - Algoritmo

A otimização da função objetivo na filtragem colaborativa consiste em

- Dado $x^{(1)}, \dots, x^{(n_m)}$, estimar $\theta^{(1)}, \dots, \theta^{(n_u)}$:

$$\underset{\theta^{(1)}, \dots, \theta^{(n_u)}}{\text{minimize}} \frac{1}{2} \sum_{j=1}^{n_u} \sum_{i:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^n (\theta_k^{(j)})^2$$

- Dado $\theta^{(1)}, \dots, \theta^{(n_u)}$, estimar $x^{(1)}, \dots, x^{(n_m)}$:

$$\underset{x^{(1)}, \dots, x^{(n_m)}}{\text{minimize}} \frac{1}{2} \sum_{i=1}^{n_m} \sum_{j:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^n (x_k^{(i)})^2$$

Esses dois problemas podem ser juntados em um só, minimizando simultaneamente para $x^{(1)}, \dots, x^{(n_m)}$ e $\theta^{(1)}, \dots, \theta^{(n_u)}$, o que pode ser formalizado como:

$$\begin{aligned} J(x^{(1)}, \dots, x^{(n_m)}, \theta^{(1)}, \dots, \theta^{(n_u)}) &= \frac{1}{2} \sum_{(i,j):r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^n (x_k^{(i)})^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^n (\theta_k^{(j)})^2 \\ &\underset{\substack{x^{(1)}, \dots, x^{(n_m)} \\ \theta^{(1)}, \dots, \theta^{(n_u)}}}{\text{minimize}} J(x^{(1)}, \dots, x^{(n_m)}, \theta^{(1)}, \dots, \theta^{(n_u)}) \end{aligned}$$

Note que a primeira parcela engloba o 1º e 2º casos apenas introduzindo (i, j) no limite inferior do somatório.

Isso equivale à sequência $\theta \rightarrow x \rightarrow \theta \rightarrow x \rightarrow \theta \rightarrow x \rightarrow \dots$ que fizemos antes.

Temos então os seguintes passos para o algoritmo:

1. Inicializar $x^{(1)}, \dots, x^{(n_m)}, \theta^{(1)}, \dots, \theta^{(n_u)}$ com valores aleatórios pequenos (quebra de simetria como nas redes neurais)
2. Minimizar $J(x^{(1)}, \dots, x^{(n_m)}, \theta^{(1)}, \dots, \theta^{(n_u)})$ usando gradiente descendente (ou um algoritmo mais avançado de otimização). Assim, para todo $j = 1, \dots, n_u, i = 1, \dots, n_m$:

$$x_k^{(i)} = x_k^{(i)} - \alpha \left(\sum_{j:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)}) \theta_k^{(j)} + \lambda x_k^{(i)} \right)$$

$$\theta_k^{(j)} = \theta_k^{(j)} - \alpha \left(\sum_{i:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)}) x_k^{(i)} + \lambda \theta_k^{(j)} \right)$$

3. Para um usuário com parâmetro $\theta^{(j)}$ e um filme com atributos (aprendidos) $x^{(i)}$, prever a nota $(\theta^{(j)})^T(x^{(i)})$.

IMPORTANTE: Aqui não temos mais a convenção $x_0 = 1$. Nossa parâmetro x_0 será aprendido por nosso algoritmo como todos os demais. Isso faz com que agora tenhamos $x \in \mathbb{R}^n$ e $\theta \in \mathbb{R}^n$ (o expoente não é mais $n+1$ como antes).

O fato de que x_0 agora é aprendido normalmente também faz com que o gradiente descendente não tenha mais o tratamento especial para x_0 .

13.5 Fatorização de Matrix de Baixo Posto (*Low Rank Matrix Factorization*)

A filtragem colaborativa pode ser descrita vetorialmente. Nossa matriz de notas

Filme	Alice	Bob	Carol	Dave
Love at least	5	5	0	0
Romance forever	5	?	?	0
Cute puppies of love	?	4	0	?
Nonstop car chases	0	0	5	4
Swords vs. karate	0	0	5	?

pode ser representada pela matriz Y :

$$Y = \begin{bmatrix} 5 & 5 & 0 & 0 \\ 5 & ? & ? & 0 \\ ? & 4 & 0 & ? \\ 0 & 0 & 5 & 4 \\ 0 & 0 & 5 & 0 \end{bmatrix}$$

As notas preditas pelo modelo também podem ser representadas matricialmente

$$Y \approx \begin{bmatrix} (\theta^{(1)})^T(x^{(1)}) & (\theta^{(2)})^T(x^{(1)}) & \dots & (\theta^{(n_u)})^T(x^{(1)}) \\ (\theta^{(1)})^T(x^{(2)}) & (\theta^{(2)})^T(x^{(2)}) & \dots & (\theta^{(n_u)})^T(x^{(2)}) \\ \vdots & \vdots & \vdots & \vdots \\ (\theta^{(1)})^T(x^{(n_m)}) & (\theta^{(2)})^T(x^{(n_m)}) & \dots & (\theta^{(n_u)})^T(x^{(n_m)}) \end{bmatrix}$$

Definimos então uma matriz X , idêntica à que tínhamos na regressão linear, colocando $(x^{(i)})^T$ em cada linha e definimos a matriz Θ colocando $(\theta^{(j)})^T$ em cada linha:

$$X = \begin{bmatrix} (x^{(1)})^T \\ (x^{(2)})^T \\ \vdots \\ (x^{(n_m)})^T \end{bmatrix} \quad \Theta = \begin{bmatrix} (\theta^{(1)})^T \\ (\theta^{(2)})^T \\ \vdots \\ (\theta^{(n_u)})^T \end{bmatrix}$$

Temos então que nossa matriz de predição pode ser representada vetorialmente simplesmente por

$$X\Theta^T.$$

O algoritmo que vimos é também chamado de **fatorização de matriz de posto baixo** e o nome se deve ao fato de que essa matriz $X\Theta^T$ tem a propriedade da álgebra linear de ter posto baixo.

13.5.1 Encontrando filmes relacionados

Para cada filme/produto i , aprendemos um vetor de atributos $x^{(i)} \in \mathbb{R}^n$. Pode ser que identifiquemos algum significado para cada atributo, p.ex., $x_1 = \text{romance}$, $x_2 = \text{ação}$, $x_3 = \text{comédia}$, etc. Mas isso muitas vezes não é possível.

Independentemente disso, sempre podemos identificar filmes relacionados. Dizemos que os filmes i e j são relacionados (“similares”) se

$$\|x^{(i)} - x^{(j)}\|$$

for pequeno.

Se queremos encontrar, por exemplo, os 5 filmes mais relacionados ao filme i , buscamos pelos 5 filmes j com os menores $\|x^{(i)} - x^{(j)}\|$.

13.6 Normalização pela Média

Suponha que tenhamos um usuário 5 (Eve) que não deu nota para nenhum filme:

Filme	Alice	Bob	Carol	Dave	Eve
Love at least	5	5	0	0	?
Romance forever	5	?	?	0	?
Cute puppies of love	?	4	0	?	?
Nonstop car chases	0	0	5	4	?
Swords vs. karate	0	0	5	?	?

Se usarmos o algoritmo visto até agora:

$$\underset{\substack{x^{(1)}, \dots, x^{(n_m)} \\ \theta^{(1)}, \dots, \theta^{(n_u)}}}{\text{minimize}} \frac{1}{2} \sum_{(i,j):r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^n (x_k^{(i)})^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^n (\theta_k^{(j)})^2,$$

repare que então apenas a última parcela teria influência do novo usuário, pois na 1^a parcela teríamos sempre $r(i,j) = 0$, que não contaria, e a 2^a depende apenas dos filmes.

Teríamos então, supondo $n = 2$, o termo extra

$$\frac{\lambda}{2} [(\theta_1^{(5)})^2 + (\theta_2^{(5)})^2].$$

No processo de minimização teríamos então

$$\theta^{(5)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

e portanto todas as previsões de notas para o usuário 5 serão

$$(\theta^{(5)})^T x^{(i)} = 0,$$

ou seja, o usuário atribuiria 0 estrela para todos os filmes, o que não é nada realista.

A solução para isso é definir um vetor médio de nota por filme:

$$\mu = \begin{bmatrix} 2.5 \\ 2.5 \\ 2 \\ 2.25 \\ 1.25 \end{bmatrix}$$

e então subtraímos o vetor μ de cada coluna de Y , que se tornaria

$$Y = \begin{bmatrix} 2.5 & 2.5 & -2.5 & -2.5 & ? \\ 2.5 & ? & ? & -2.5 & ? \\ ? & 2 & -2 & ? & ? \\ -2.25 & -2.25 & 2.75 & 1.75 & ? \\ -1.25 & -1.25 & 3.75 & -1.25 & ? \end{bmatrix}$$

E então para o usuário j e filme i predizer

$$(\theta^{(j)})^T (x^{(i)}) + \mu_i.$$

No caso do usuário Eve, em que $\theta^{(5)}$ é o vetor nulo, temos então o vetor médio μ como previsão das notas, algo que faz bem mais sentido.

Capítulo 14

Aprendizado em Larga Escala

14.1 Aprendizado de Máquinas em Larga Escala

Como vimos, existe uma hipótese de que ter acesso a uma grande quantidade de dados para treinamento é mais importante do que o algoritmo em si.

É o caso do problema na Figura 14.1 que já vimos, de classificar entre palavras similares, p.ex., to/two/too, then/than. Todos os algoritmos têm desempenho parecido e todos melhoram com maior tamanho do treinamento.

“O vencedor não é quem tem o melhor algoritmo, mas sim que tem mais dados!”

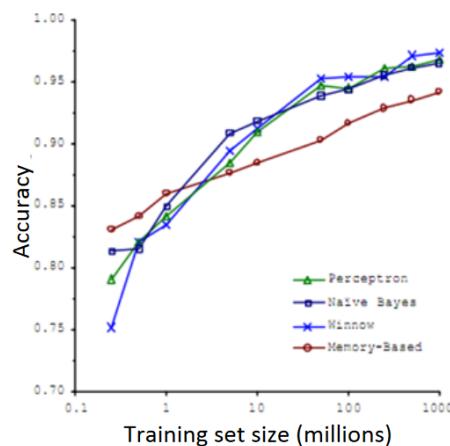


Figura 14.1: Importância de conjuntos grandes para treinamento.

Existe um problema “apenas” com essa ideia: o custo computacional de se tratar com dados muito grandes.

Imagine o gradiente descendente

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

com $m = 100000000$. Pensa nos dados da população de um país como Brasil ou EUA, por exemplo! O processamento seria muito caro!

Mas será que, p.ex., $m = 1000$ não seria suficiente já? Isso vai depender da curva de aprendizado.

Se, com o aumento de m , $J_{treino}(\theta)$ segue aumentando devagar e $J_{cv}(\theta)$ segue diminuindo devagar (Figura 14.2 esquerda), isso indica variância alta e aumentar m é conveniente. Já quando $J_{treino}(\theta)$ aumenta rapidamente e estabiliza e $J_{cv}(\theta)$ diminui rapidamente e estabiliza (Figura 14.2 direita), isso indica alto viés e nesse caso não adianta aumentar m .

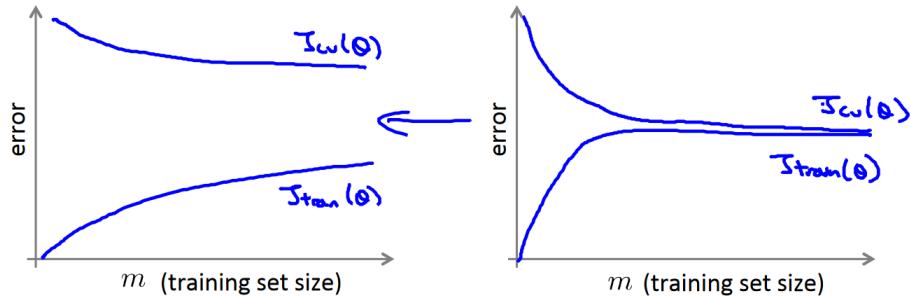


Figura 14.2: Curva de aprendizado.

Porém, ainda neste segundo caso, cabe observar que a recomendação aí é aumentar os atributos ou o número de unidades de uma rede neural, o que no final poderia novamente requerer mais dados de treinamento.

14.2 Gradiente Descendente Estocástico

Lembrando do gradiente descendente na regressão linear:

$$h_\theta(x) = \sum_{j=0}^n \theta_j x_j$$

$$J_{treino}(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2$$

Repita {

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

(para todo $j = 0, \dots, n$)

}

Cada iteração é executada sobre todos os exemplos de treinamento DE UMA VEZ, mesmo que m seja enorme, como p.ex. $m=300000000$, que é algo bem realista, só pensar na população de um país como os EUA, por exemplo.

Note que o somatório representa, diretamente, a derivada

$$\frac{\partial}{\partial \theta_j} J_{treino}(\theta).$$

A função de custo caminha diretamente para o mínimo (Figura 14.3).

Este algoritmo é chamado de gradiente descendente em **lote** (*batch*).

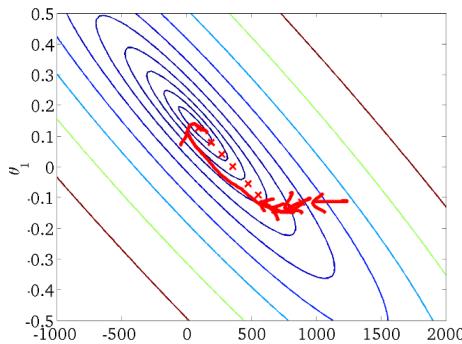


Figura 14.3: Caminhada do gradiente descendente em lote.

Para m grande, este algoritmo tem um alto custo computacional.

Para este caso, desenvolveu-se então o **gradiente descendente estocástico**.

Vamos reescrever nossa função de custo, a partir da definição de uma função auxiliar $custo(\theta, (x^{(i)}, y^{(i)}))$:

$$custo(\theta, (x^{(i)}, y^{(i)})) = \frac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2,$$

que mede a contribuição de cada exemplo INDIVIDUALMENTE para o custo total.

Assim, reescrevemos o custo total do treino:

$$J_{treino}(\theta) = \frac{1}{m} \sum_{i=1}^m custo(\theta, (x^{(i)}, y^{(i)})).$$

O algoritmo se torna então:

1. Embaralhar aleatoriamente (reordenar) os exemplos de treinamento
2. Repita { % Usualmente repete 1 ~ 10×
 - para $i := 1$ até m {
 - $\theta_j := \theta_j - \alpha(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$
 - (para todo $j = 0, \dots, n$)
 - }
}

Ou seja, os parâmetros são atualizados em cada iteração a partir de um ÚNICO

exemplo de treino: 1^a iteração usa $x^{(1)}, y^{(1)}$, 2^a usa $x^{(2)}, y^{(2)}$, etc. Muito mais rápido!

Note que a expressão multiplicando α é simplesmente a derivada $\frac{\partial}{\partial \theta_j} custo(\theta, (x^{(i)}, y^{(i)}))$.

Um ponto apenas é que agora a caminhada do algoritmo rumo ao mínimo é muito mais errática, já que temos um único exemplo para estimar a direção de descida, agregando ruído ao processo (Figura 14.4).

Ele pode de fato ficar flutuando em volta do mínimo sem efetivamente chegar nele. Mas isso não é um grande problema, o algoritmo já costuma funcionar bem no teste apenas com uma aproximação do mínimo verdadeiro.

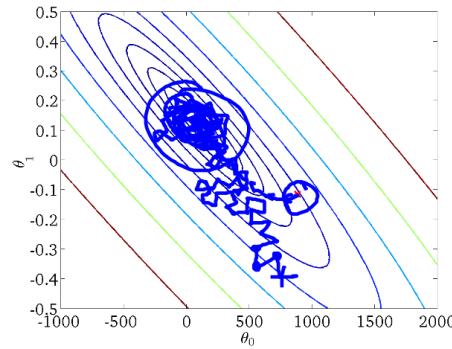


Figura 14.4: Caminhada do gradiente descendente estocástico.

14.3 Gradiente Estocástico de Mini-Lote (*Mini-Batch*)

Um meio-termo entre o gradiente em lote (m exemplos em cada iteração) e o gradiente estocástico ($m = 1$) é o gradiente em mini-lote (*mini-batch*), que usa b exemplos por iteração.

b é o chamado **tamanho do mini-lote** e usualmente assume valores de $2 \sim 100$.

Se, por exemplo, $b = 10$ e $m = 1000$, o algoritmo é:

```

Repete {
    para  $i = 1, 11, 21, 31, \dots, 991$  {
         $\theta_j := \theta_j - \alpha \frac{1}{10} \sum_{k=i}^{i+9} (h_\theta(x^{(k)}) - y^{(k)}) x_j^{(k)}$ 
        (para todo  $j = 0, \dots, n$ )
    }
}

```

Uma vantagem importante da ideia de mini-lote em relação ao gradiente estocástico é que a maioria das bibliotecas de álgebra linear permite a vetorização desse somatório.

14.4 Convergência do Gradiente Descendente Estocástico

No caso do gradiente descendente clássico (em lote) temos

$$J_{treino}(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2$$

e podemos debugar plotando $J_{treino}(\theta)$ em função do número de iterações.

Já no gradiente estocástico temos

$$custo(\theta, (x^{(i)}, y^{(i)})) = \frac{1}{2} (h(x^{(i)}) - y^{(i)})^2,$$

que é usado para atualizar θ a partir de cada $(x^{(i)}, y^{(i)})$ individualmente.

Como vimos, isso faz com que $J_{treino}(\theta)$ flutue muito a cada iteração e não temos mais garantia de que ela sempre diminuirá.

O que se faz nesse caso então é plotar a cada, por exemplo, 1000 iterações, a média de $custo(\theta, (x^{(i)}, y^{(i)}))$ sobre os últimos 1000 exemplos processados pelo algoritmo.

Deve-se notar que, mesmo com alguma flutuação, o valor médio de $custo(\theta, (x^{(i)}, y^{(i)}))$ tende a cair com as iterações.

Normalmente, o uso de uma taxa de aprendizado α menor faz com que essa queda seja menor no começo, mas a partir de um certo ponto o erro pode se tornar menor (Figura 14.5 superior esquerda).

O uso de mais exemplos na média (5000, por exemplo), torna a curva mais suave (Figura 14.5 superior direita), mas gera também um atraso maior entre uma mudança de comportamento do algoritmo e o efeito correspondente na curva média.

Uma curva flutuando em torno de um valor constante (Figura 14.5 inferior esquerda) indica que o algoritmo não está aprendendo. Pode-se tentar diminuir a taxa de aprendizado, aumentar o número de atributos ou mexer em qualquer outro aspecto do algoritmo de aprendizado.

Por fim, uma curva média ascendente (Figura 14.5 inferior direita) é sugestiva de que se deve usar uma taxa de aprendizado menor.

Falando em taxa de aprendizado, por enquanto ela foi mantida constante, mas uma estratégia que pode ser usada é fazer com que ela diminua a cada iteração:

$$\alpha = \frac{const1}{\text{número da iteração} + const2}.$$

Isso diminui as chances de o algoritmo ficar flutuando em torno do mínimo sem efetivamente chegar nele.

O problema é que isso exige que se definam mais dois hiper-parâmetros $const1$ e $const2$. Além de que na prática uma aproximação do mínimo já costuma ser suficiente.

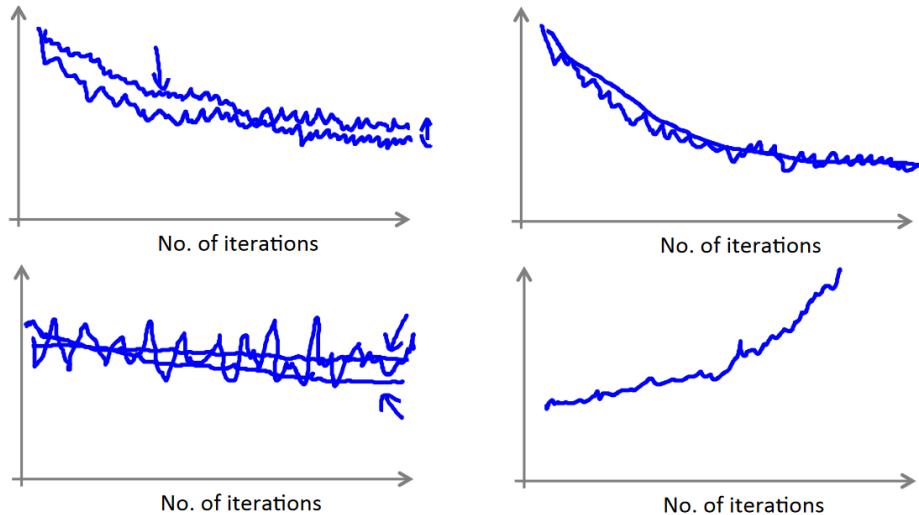


Figura 14.5: $custo(\theta, (x^{(i)}, y^{(i)}))$ para os últimos 1000 exemplos de treino.

14.5 Aprendizado *Online*

Um algoritmo de aprendizado *online* aprende com cada novo exemplo de treino que chega e em seguida o descarta.

Imagine um serviço de entregas (tipo Correios).

O usuário entra no site, coloca origem e destino, e o sistema calcula um preço para a entrega. O usuário pode então contratar o serviço ($y = 1$) ou não ($y = 0$).

Os atributos x podem capturar propriedades do usuário, origem/destino e o preço oferecido. Queremos aprender $p(y = 1|x; \theta)$ para assim otimizar o preço (embutido em x).

Podemos usar, por exemplo, regressão logística e o algoritmo então será

```

Repetir "para sempre" {
    Obter  $(x, y)$  do usuário correspondente
    Atualizar  $\theta$  usando  $(x, y)$ :
         $\theta_j := \theta_j - \alpha(h_\theta(x) - y)x_j \quad (j = 0, \dots, n)$ 
}
  
```

Repare que não temos mais $(x^{(i)}, y^{(i)})$ pois só o usuário atual interessa.

Um aspecto muito interessante desse algoritmo é que ele **SE ADAPTA** a mudanças de preferência dos clientes, p.ex., uma crise que faça com que os clientes precisem pagar menos.

14.5.1 Outro exemplo - Busca de produtos (“aprender a buscar”)

Suponha que um usuário busca por “celular Android com câmera 4k”.

Suponha que existam 100 celulares na loja e essa busca retorna 10.

x = atributos do celular, quantas palavras na busca do usuário casam com o nome do celular, quantas palavras na busca casam com a descrição do celular, etc.

$y = 1$ se o usuário clica no link e $y = 0$ caso contrário.

Aprender $p(y = 1|x; \theta)$. Isso é chamado de previsão de CTR (*Click True Rate*).

Assim o sistema poderá mostrar os 10 celulares mais susceptíveis de serem clicados pelo usuário.

Outros exemplos: escolha de ofertas especiais para mostrar ao usuário, seleção customizada de artigos em um jornal/revista, recomendação de produtos (um sistema de filtragem colaborativa pode gerar atributos), etc.

14.6 MapReduce e Paralelismo de Dados

MapReduce é uma estratégia, proposta por Jefrey Dean e Sanjay Ghemawat, que divide uma tarefa muito grande em tarefas menores independentes e distribui entre máquinas.

Vamos ilustrar com um gradiente em lote e $m = 400$, só para simplificar, pois no mundo real isso seria muito mais interessante, por exemplo, para $m = 400$ milhões.

Temos então

$$\theta_j := \theta_j - \alpha \frac{1}{400} \sum_{i=1}^{400} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

Suponha que tenhamos 4 máquinas (Figura 14.6). Dividimos então o treino em 4 partes.

Temos então:

- Máquina 1: Usa $(x^{(1)}, y^{(1)}), \dots, (x^{(100)}, y^{(100)})$ e calcula

$$temp_j^{(1)} = \sum_{i=1}^{100} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

- Máquina 2: Usa $(x^{(101)}, y^{(101)}), \dots, (x^{(200)}, y^{(200)})$ e calcula

$$temp_j^{(2)} = \sum_{i=101}^{200} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

- Máquina 3: Usa $(x^{(201)}, y^{(201)}), \dots, (x^{(300)}, y^{(300)})$ e calcula

$$temp_j^{(3)} = \sum_{i=201}^{300} (h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$$

- Máquina 4: Usa $(x^{(301)}, y^{(301)}), \dots, (x^{(400)}, y^{(400)})$ e calcula

$$temp_j^{(4)} = \sum_{i=301}^{400} (h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$$

Finalmente, temos uma máquina mestre que combina os resultados parciais:

$$\theta_j := \theta_j - \alpha \frac{1}{400} (temp_j^{(1)} + temp_j^{(2)} + temp_j^{(3)} + temp_j^{(4)}), \quad (j = 0, \dots, n).$$

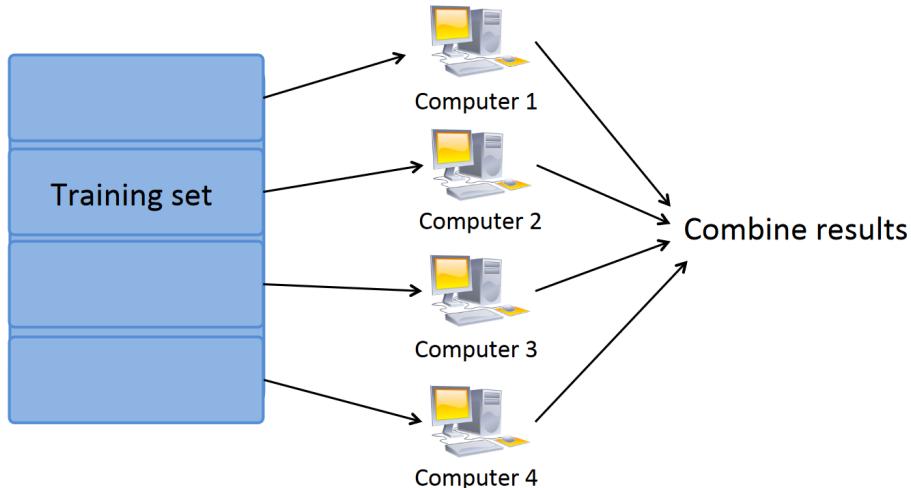


Figura 14.6: *MapReduce*.

O *MapReduce* pode ser aplicado, em geral, sobre qualquer algoritmo que possa ser reescrito como uma soma de partes e isso é verdade para vários algoritmos de aprendizado.

Isso inclui algoritmos mais avançados de otimização. No exemplo da regressão logística temos

$$J_{treino}(\theta) = -\frac{1}{m} \sum_{i=1}^m y^{(i)} \log h_\theta(x^{(i)}) - (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))$$

e para esses algoritmos avançados, como vimos, o que precisamos é da derivada parcial

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

E este somatório pode então ser quebrado em partes e o *MapReduce* ser aplicado.

Por fim, mesmo quando temos uma única máquina, mas com vários *cores*, o *MapReduce* também se aplica de forma idêntica (Figura 14.7).

Uma vantagem é que isso praticamente elimina problemas de latência e *overhead* na comunicação entre os nós, que é um problema no *MapReduce* para múltiplas máquinas.

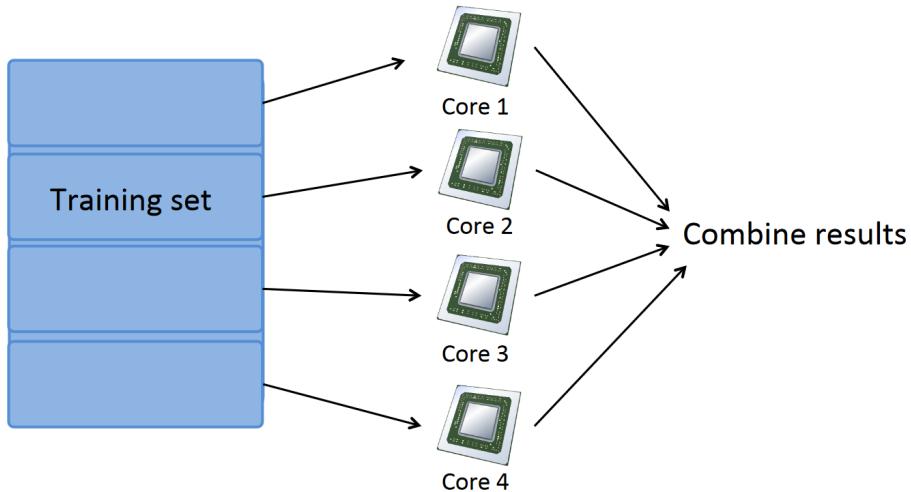


Figura 14.7: *MapReduce* em uma única máquina de vários núcleos.

Vale destacar ainda que uma grande parte das bibliotecas modernas de álgebra linear (operações vetoriais) já fazem também essa divisão de tarefas entre os núcleos automaticamente.

Capítulo 15

Exemplo Completo de Aplicação

15.1 Exemplo de Aplicação: OCR em Imagens

Veremos a seguir um exemplo completo de aplicação de aprendizado de máquinas.

OCR em texto escaneado já é um tópico praticamente resolvido, mas em imagens de cenários reais (“*in the wild*”), ainda é um grande desafio de aprendizado de máquina.

A Figura 15.1 resume o problema de OCR em imagem. Temos uma foto complexa e o objetivo é localizar e extrair textos ali presentes.

O *pipeline* geral de OCR consiste de 3 etapas: localização do texto, segmentação (separação) de caracteres e classificação de cada caractere individual (Figura 15.2).

Podemos ter ainda etapas mais avançadas como de análise de linguagem, que levaria por exemplo um texto identificado como “clean” a ser corrigido para “clean”. Mas não vamos fazer isso aqui.

Essa divisão em etapas permite que a equipe de trabalho também possa ser dividida, ficando, por exemplo, de 1 a 5 engenheiros para detecção de texto, 1 a 5 para segmentação e 1 a 5 para o reconhecimento de caractere.

15.2 Janelas Deslizantes

Uma abordagem clássica supervisionada para detecção de objetos que pode ser aplicada na detecção de texto são as janelas deslizantes.

Considere na Figura 15.3 o exemplo de detecção de pedestres.

Vamos considerar janelas (*patches*) de treinamento com tamanho 82×36 e tomamos x como sendo os pixels nesses patches. Se o *patch* contém um pedestre, temos um exemplo positivo ($y = 1$), senão, um exemplo negativo ($y = 0$).



Figura 15.1: OCR em imagem.

1. Text detection



2. Character segmentation



3. Character classification



Figura 15.2: Pipeline de OCR.

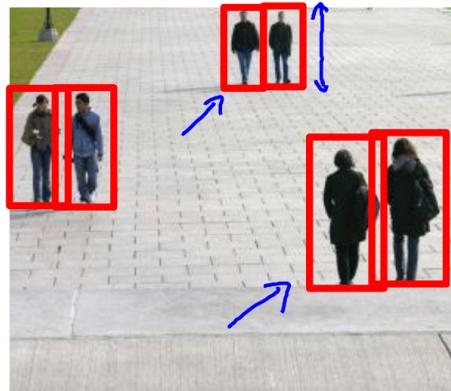


Figura 15.3: Detecção de pedestres.



Figura 15.4: *Patches* de treinamento para detecção de pedestres. Exemplos positivos ($y = 1$) à esquerda e negativos ($y = 0$) à direita.

Em seguida, vamos deslizando uma janela do tamanho do *patch* por toda a imagem, da esquerda para a direita, de cima para baixo, e colocamos aquela região da imagem no conjunto de testes de uma regressão logística, por exemplo. Podemos assim calcular a probabilidade de aquela região corresponder a um pedestre.



Figura 15.5: Janelas deslizantes.

NOTA: A janela pode deslizar de 1 em 1 pixel ou usar um determinado tamanho de passo (*stride*) por eficiência computacional.

Note que esse caso tem a vantagem de que usualmente temos uma ideia do tamanho aproximado do *patch* que conterá um pedestre.

Ainda assim, podemos também repetir o algoritmo com um *patch* maior para detectar pedestres em outra escala.

Uma abordagem similar se aplica também à detecção de texto. Podemos, no caso, treinar com os exemplos da Figura 15.6.

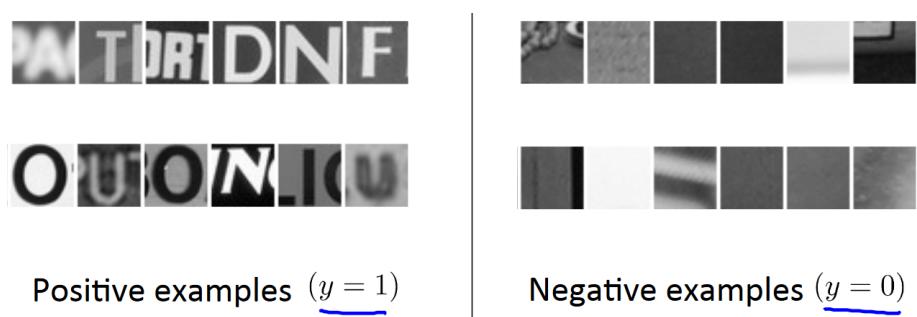


Figura 15.6: Exemplos de treinamento para detecção de texto.

A Figura 15.7 ilustra o resultado do processo.

A Figura 15.7 inferior esquerda mostra um mapa das probabilidades de $y = 1$, em que $p(y = 1) = 1$ é branco e $p(y = 1) = 0$ é preto.

Note que para chegarmos ao resultado final desta etapa (Figura 15.7 inferior direita), temos dois pós-processamentos aí.

Primeiramente os pixels são limiarizados (*threshold*) para ficarem binários e passam por uma “expansão”, em que pixels próximos a um pixel branco ($y = 1$) são também setados para 1.

Outra operação é remover regiões brancas que não tenham aspecto de um retângulo horizontal, que é o formato esperado para um texto. Isso faz com que restem apenas as regiões com retângulo vermelho.

Apenas um texto (circulado em verde) foi perdido, mas estava em um vidro transparente e era mesmo difícil de identificar.



Figura 15.7: Resultado da detecção de texto com janela deslizante.

A mesma ideia de janela deslizante se aplica na segmentação também. Nesse caso, os exemplos positivos ($y = 1$) são aqueles em que dois caracteres aparecem cortados na mesma imagem, enquanto que nos exemplos negativos ($y = 0$), cada imagem contém apenas um único caractere (Figura 15.8). O classificador vai reconhecer assim as regiões de separação entre caracteres.

Por fim, cada caractere é classificado a partir de um conjunto de treinamento que contém imagens deste caractere.

15.3 Síntese de Dados Artificiais

No reconhecimento de caracteres, podemos povoar nosso treinamento coletando imagens do mundo real (Figura 15.9 esquerda), mas também podemos gerar dados artificiais (Figura 15.9 direita).

Uma forma de fazer isso é usar diferentes fontes (existe uma infinidade no processador de texto e na internet) e adicionar um fundo aleatório. Pode-se ainda acrescentar transformações geométricas de espelhamento, rotação, escala, etc.

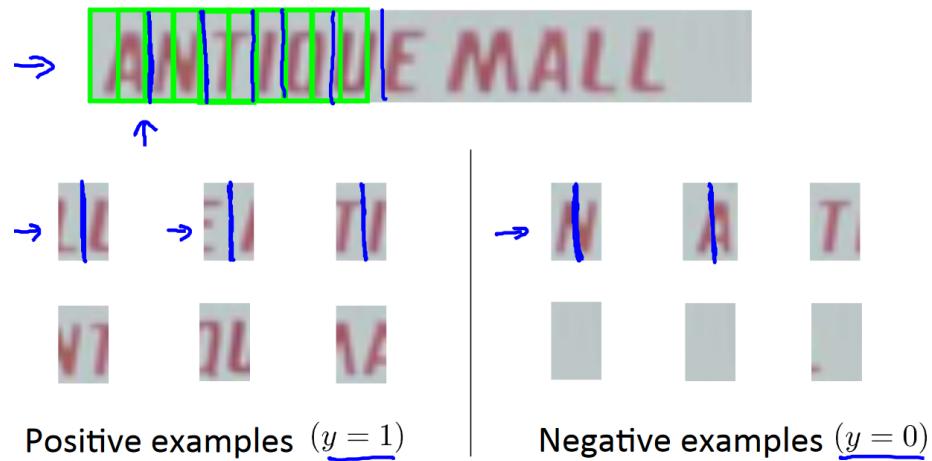


Figura 15.8: Janela deslizante na segmentação de caracteres.



Figura 15.9: Dados reais (esquerda) e sintéticos (direita).

Outra solução é fazer distorções sobre a imagem original (Figura 15.10).

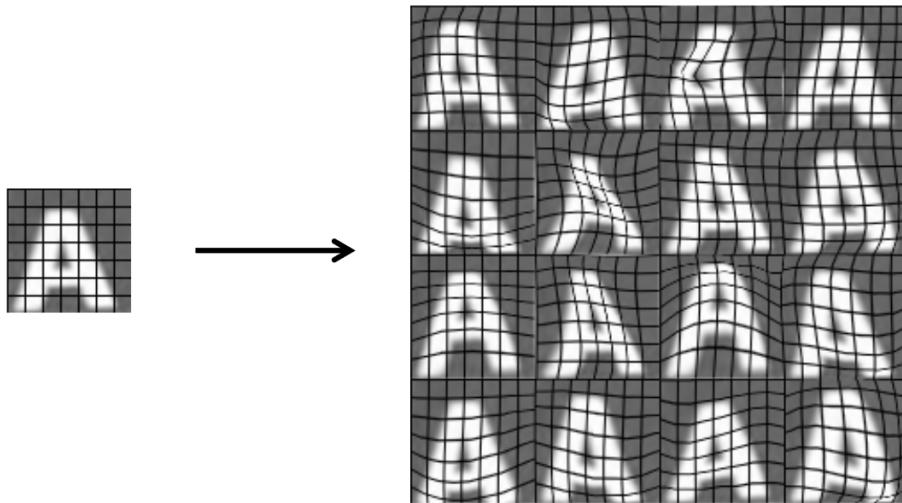


Figura 15.10: Distorções sobre a imagem original.

Vale notar que estratégias desse tipo se aplicam também a outros tipos de dados. Por exemplo, com áudio, podemos simular uma conexão telefônica ruim ou um som de fundo qualquer.

Uma ressalva importante apenas é que as distorções adicionadas devem ser representativas de distorções/ruídos no conjunto de teste.

Em geral, adicionar ruído puramente aleatório ao dado não ajuda.

Temos dois pontos importantes para discutir no processo de aquisição de mais dados.

1. Certifique-se de que o classificador tem um viés baixo antes de gastar tempo adquirindo mais dados. Plote curvas de aprendizado para isso! Se o viés ainda for alto, por exemplo, acrescente mais atributos ou unidades no caso de rede neural.
2. Quanto trabalho terá para obter 10 vezes mais dados do que tem atualmente?
 - Síntese de dados artificiais
 - Precisa coletar e também rotular os dados. Você mesmo vai fazer isso? Quanto tempo demora/custa cada exemplo? Quanto custaria no total? Vale a pena?
 - “*Crowd source*”: ferramentas em que usuários fazem trabalhos manuais pequenos por um pequeno valor. Pode ter problemas com a qualidade de algo que envolve tantas pessoas. EX.: Amazon Mechanical Turk

15.4 *Ceiling Analysis*: Em Qual Parte do Pipeline se Concentrar?

Retomando nosso *pipeline* de OCR (Figura 15.11).

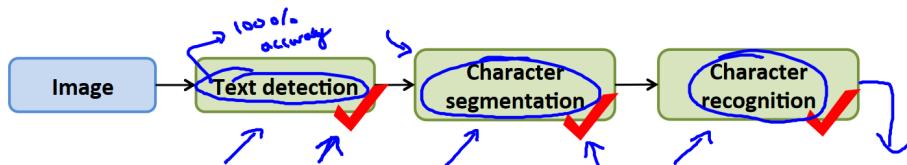


Figura 15.11: *Pipeline* de OCR.

A ideia de *ceiling analysis* (limitante superior de desempenho) é manualmente fazer com que uma etapa do *pipeline* tenha 100% de acurácia e verificar nessas circunstâncias então qual a acurácia na etapa seguinte.

Primeiramente, consideramos a acurácia do sistema original. Suponha que seja 72%.

Em seguida, forçamos 100% de acurácia na detecção de texto, detectando manualmente. Imagine que a acurácia do sistema como um todo nesse caso sobe para 89%.

Logo após, fazemos a segmentação manualmente para que esta tenha 100% de acurácia e recalculamos a acurácia geral, que agora sobe para 90%.

Finalmente, fazemos também o reconhecimento de caractere manualmente e, como é de se esperar, a acurácia geral chega então em 100%.

A Tabela abaixo resume a situação:

Sistema geral	72%
Detecção de texto	89%
Segmentação de caractere	90%
Reconhecimento de caractere	100%

Note que, com a detecção de texto, nossa acurácia aumentou 17%. Com a segmentação 1% e com o reconhecimento 10%.

Isso nos indica que devemos, em primeiro lugar, concentrar nosso tempo e esforços no sistema de detecção de texto, em seguida no reconhecimento de caractere.

15.4.1 Outro exemplo

Veja o *pipeline* na Figura 15.12 para reconhecimento facial.

Fazemos processo parecido. Removemos o fundo manualmente com 100% de acurácia, detectamos a face, segmentamos manualmente os olhos (usualmente a parte mais importante para reconhecimento deste tipo), o nariz, a boca, e finalmente classificamos manualmente tendo certeza de 100% de precisão. Após cada etapa, reavaliarmos a acurácia.

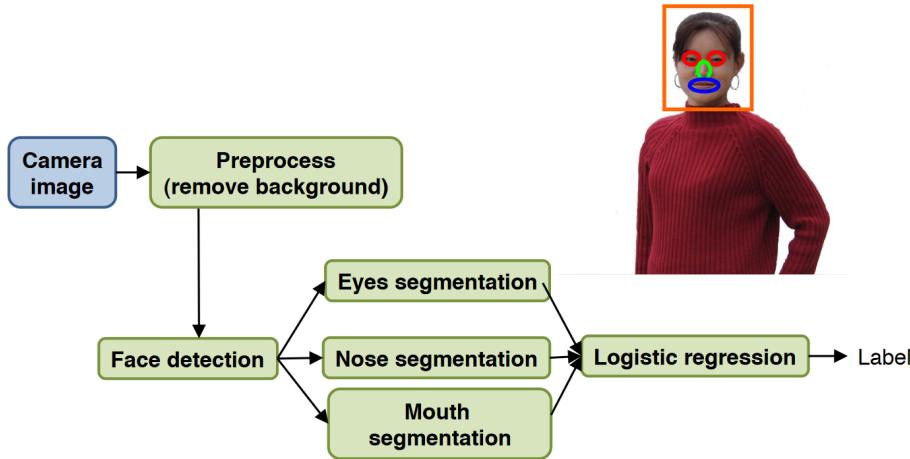


Figura 15.12: Pipeline de reconhecimento facial.

Sistema geral	85%
Pré-processamento (remoção de fundo)	85.1%
Detecção de face	91%
Detecção dos olhos	95%
Detecção do nariz	96%
Detecção da boca	97%
Classificação	100%

Note como a remoção de fundo trouxe um ganho de apenas 0.1%, a detecção de face nos deu 5.9%, a detecção dos olhos 4%, a detecção do nariz 1%, a detecção da boca 1% e a classificação nos trouxe 3%.

Devemos, portanto, nos concentrar em melhorar, pela ordem, os algoritmos de detecção de face, detecção de olhos e o classificador.

Em geral, não se deve confiar na intuição (*gut feeling*) para a tomada de decisões desse tipo em *machine learning*.

Existem casos de equipes que chegaram a gastar um ano e meio em algoritmos de remoção de fundo para reconhecimento facial e ao final descobriram que o ganho com isso era marginal