MAC0460/5832 - Introdução ao Aprendizado de Máquina (2022)

Notas de aula[1]

Em constante revisão. Último update: 16/05/2022

(Nina S. T. Hirata)

Índice

[1]Estas notas de aula estão sendo preparadas *on-the-fly*. Assim, elas podem não estar 100% precisas e completas. Não posso fazer como o D. Knuth (https://en.wikipedia.org/wiki/Knuth_reward_check) e recompensar aqueles que encontrarem erros, mas agradeço qualquer correção/sugestão.

1 Aprendizado supervisionado (ou preditivo)

Considere um espaço $\mathcal{X}$, que chamaremos de espaço de entrada, e um espaço $\mathcal{Y}$ que chamaremos de espaço de saída. Assim, um elemento de $\mathcal{X} \times \mathcal{Y}$ é um par $(\mathbf{x}, y)$, com $\mathbf{x} \in \mathcal{X}$ e $y \in \mathcal{Y}$.

Por exemplo, suponha que $\mathbf{x}$ corresponde a altura e y ao peso de pessoas. É um tanto intuitivo que quanto maior for a altura, maior tende a ser o peso de uma pessoa. Ainda por exemplo, suponha que $\mathbf{x} = (x_1, x_2, \ldots, x_d)$ indica se uma pessoa apresenta ou não cada um dos sintomas x_j $(j = 1, \ldots, d)$ de uma lista predeterminada e y indica se a pessoa está doente ou não. Em muitas situações, a partir dos sintomas pode-se saber se a pessoa está doente ou não.

Portanto, há várias situações em que é razoável supor que existe uma relação entre entradas $\mathbf{x}$ e saídas y. Muitas dessas relações podem ser descritas por alguma função $f : \mathcal{X} \to \mathcal{Y}$, muitas vezes chamada de **função-alvo**.

O problema de nosso interesse é identificar f a partir de um conjunto de observações $\mathcal{D} = \{(\mathbf{x}_i, y_i) \in \mathcal{X} \times \mathcal{Y} : i = 1, \ldots, N\}$. Desejamos encontrar uma função $h : \mathcal{X} \to \mathcal{Y}$ que seja uma boa aproximação de f. Isto é, tal que $\hat{y} = h(\mathbf{x})$ seja o mais próximo possível de $y = f(\mathbf{x})$ para qualquer par $(\mathbf{x}, y) \in \mathcal{X} \times \mathcal{Y}$. Um ponto importante a ser observado é que a função-alvo f é desconhecida.

No primeiro exemplo, de estimar peso a partir da altura, a variável-alvo y é um número real e os problemas nessses casos são chamados de *regressão*. Já no segundo exemplo, de determinar se uma pessoa está doente ou não, y é uma variável categórica ($\{\text{doente}, \text{saudável}\}$) e nesses casos os problemas são chamados de *classificação*.

Erro empírico: Seja $l(y, \hat{y}) : \mathbb{R} \times \mathbb{R} \to \mathbb{R}_{\geq 0}$ uma função de erro (ou perda). Por exemplo, podemos ter $l(y, \hat{y}) = |y - \hat{y}|$ (erro absoluto) ou $l(y, \hat{y}) = (y - \hat{y})^2$ (erro quadrático). Dada uma função $h : \mathcal{X} \to \mathcal{Y}$ qualquer, o erro empírico de f com respeito a D é dado por

$$\mathcal{L}_D(h) = \frac{1}{N} \sum_{i=1}^{N} l(y_i, h(\mathbf{x}_i)) \tag{1}$$

Leituras adicionais sugeridas: Seções 1.1 e 1.2 de [1].

2 Regressão

Vamos supor que $\mathcal{X} = \mathbb{R}^d$ $(d > 0)$ e $\mathcal{Y} = \mathbb{R}$ e que temos um conjunto de exemplos $\mathcal{D} = \{(\mathbf{x}_i, y_i) \in \mathcal{X} \times \mathcal{Y} : i = 1, \ldots, N\}$ $(N \geq 1)$.

O problema de regressão: Dados o conjunto de exemplos D, uma função de perda l, e uma família de funções $\mathcal{H}$, encontrar $h \in \mathcal{H}$ com o menor erro empírico $\mathcal{L}_D$.

2.1 Regressão linear

Neste caso, a família de funções $\mathcal{H}$ é a família de funções lineares. Denotando $\mathbf{x} = (x_1, x_2, \ldots, x_d)$, então $h(\mathbf{x}) = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_d x_d$, na qual w_j são coeficientes reais.

Podemos simplificar a escrita, usando a notação vetorial. Note que aqui $\mathbf{x} = (x_1, x_2, \ldots, x_d)$ corresponde a um vetor em $\mathbb{R}^d$ e, embora esteja denotado horizontalmente, iremos supor que são vetores coluna; logo a sua transposta $\mathbf{x}^T$ é um vetor linha. Além disso, quando consideramos uma instância específica $\mathbf{x}_i$, escreveremos os componentes como $\mathbf{x}_i = (x_{i1}, x_{i2}, \ldots, x_{id})$ (até que essa notação fique inconveniente ... Quando isso acontecer, faremos adaptações na notação.)

Para nos livrarmos do coeficiente w_0, adicionamos um componente 1 no vetor $\mathbf{x}$, e temos $\tilde{\mathbf{x}} = (1, x_1, x_2, \ldots, x_d)$. Os coeficientes da função são denotados pelo vetor $\mathbf{w} = (w_0, w_1, \cdots, w_d)$. Assim podemos escrever: $h(\mathbf{x}) = \mathbf{w}^T \tilde{\mathbf{x}}$.

A figura 1 mostra um exemplo com $d = 1$. Observe que a relação entre x e y é aproximadamente linear (linha vermelha), a qual pode ser expressa por $\hat{y} = h(x) = w_o + w_1 x$.

2.2 Solução analítica

O problema de regressão linear, com erro quadrático, pode ser resolvido analiticamente.

Uma variável: Consideremos inicialmente exemplos entrada de dimensão 1, isto é, com $d = 1$. Note que neste caso $\mathbf{x} = (x_1)$ (apenas uma variável/atributo). Para não carregar a notação,

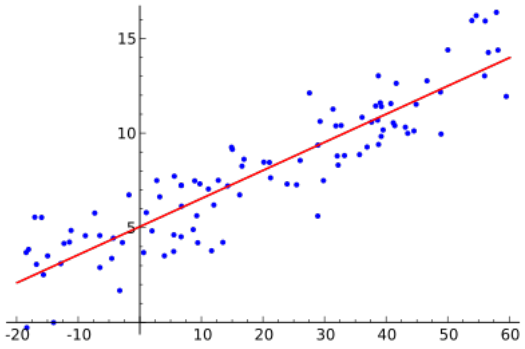


Figura 1: Uma relação aproximadamente linear entre entradas x e saída y.

uma instância $\mathbf{x}_i = (x_{i1})$ será denotado simplesmente por x_i. Queremos encontrar $\mathbf{w} = (w_0, w_1)$ que minimiza o erro empírico

$$\mathcal{L}_D(h) = \frac{1}{N}\sum_{i=1}^{N}\big(h(x_i) - y_i\big)^2 = \frac{1}{N}\sum_{i=1}^{N}\big(w_0 + w_1 x_i - y_i\big)^2 \tag{2}$$

Como esta função é convexa, basta encontrarmos o ponto de mínimo da função $\mathcal{L}$. Para maior clareza, para indicar que os parâmetros são w_0 e w_1, podemos escrever $\mathcal{L}_{\mathcal{D}}(\mathbf{w})$. Para efeito de minimização da função $\mathcal{L}$, y_i e x_i são constantes. Para simplificar a notação, omitiremos D e escrevemos $\mathcal{L}(\mathbf{w})$.

Calculando a derivada parcial com respeito a w_0 e w_1, temos

$$\frac{\partial\mathcal{L}(w_0, w_1)}{\partial w_0} = \frac{2}{N}\sum_{i=1}^{N}(w_0 + w_1 x_i - y_i)$$

$$\frac{\partial\mathcal{L}(w_0, w_1)}{\partial w_1} = \frac{2}{N}\sum_{n=1}^{N}(w_0 + w_1 x_i - y_i)\, x_i$$

Supondo $\overline{x} = \dfrac{1}{N}\displaystyle\sum_{i=1}^{N} x_i$, temos que $\displaystyle\sum_{i=1}^{N} x_i = N\,\overline{x}$. Analogamente, temos $\displaystyle\sum_{i=1}^{N} y_i = N\,\overline{y}$. Portanto,

$$\begin{aligned}\frac{\partial\mathcal{L}(w_0, w_1)}{\partial w_0} &= \frac{2}{N}\sum_{i=1}^{N}(w_0 + w_1 x_i - y_i)\\ &= \frac{2}{N}\big(N\,w_0 + w_1\,N\,\overline{x} - N\,\overline{y}\big)\\ &= 2\,w_0 + 2\,w_1\,\overline{x} - 2\,\overline{y}\end{aligned}$$

Similarmente,

$$\begin{aligned}\frac{\partial \mathcal{L}(w_0, w_1)}{\partial w_1} &= \frac{2}{N}\sum_{n=1}^{N}(w_0 + w_1 x_i - y_i)\, x_i \\ &= \frac{2}{N}\big(w_0\, N\, \overline{x} + w_1\, N\, \overline{x^2} - N\, \overline{x\, y}\big) \\ &= = 2\, w_0\, \overline{x} + 2\, w_1\, \overline{x^2} - 2\, \overline{x\, y}\end{aligned}$$

Igualando a primeira equação a zero, temos $w_0 = \overline{y} - w_1\, \overline{x}$.

Analogamente, igualando a segunda equação a zero e substituindo w_0 temos $w_1 = \dfrac{\overline{x\, y} - \overline{x}\, \overline{y}}{\overline{x^2} - \overline{x}^2}$.

Múltiplas variáveis: Podemos proceder da mesma forma acima para o caso de dimensão d qualquer. Teremos $d+1$ equações lineares com $d+1$ incógnitas. Qualquer método para solução de sistemas de equações lineares pode ser utilizado para o cálculo da solução.

No entanto, quando lidamos com grande quantidade de dados, pode ser mais interessante utilizar a notação matricial.

No caso geral, a função de custo que queremos otimizar é:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N}\Big(h(\mathbf{x}_i) - y_i\Big)^2 \tag{3}$$

em que $h(\mathbf{x}) = \mathbf{w}^T\tilde{\mathbf{x}}$.

Vamos escrever o vetor de erros (resíduos) – com respeito aos N exemplos, na forma vetor coluna:

$$\begin{bmatrix} h(\mathbf{x}_1) - y_1 \\ h(\mathbf{x}_2) - y_2 \\ \vdots \\ h(\mathbf{x}_N) - y_N \end{bmatrix} = \begin{bmatrix} h(\mathbf{x}_1) \\ h(\mathbf{x}_2) \\ \vdots \\ h(\mathbf{x}_N) \end{bmatrix} - \underbrace{\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}}_{\mathbf{y}} = \begin{bmatrix} \mathbf{w}^T\mathbf{x}_1 \\ \mathbf{w}^T\mathbf{x}_2 \\ \vdots \\ \mathbf{w}^T\mathbf{x}_N \end{bmatrix} - \mathbf{y} = \begin{bmatrix} w_0 + w_1 x_{11} + \ldots + w_d x_{1d} \\ w_0 + w_1 x_{21} + \ldots + w_d x_{2d} \\ \vdots \\ w_0 + w_1 x_{N1} + \ldots + w_d x_{Nd} \end{bmatrix} - \mathbf{y} =$$

$$\underbrace{\begin{bmatrix} 1 & x_{11} & \ldots & x_{1d} \\ 1 & x_{21} & \ldots & x_{2d} \\ \vdots & & & \\ 1 & x_{N1} & \ldots & x_{Nd} \end{bmatrix}\begin{bmatrix} w_0 \\ \vdots \\ w_d \end{bmatrix}}_{\mathbf{X}\mathbf{w}} - \mathbf{y} = \mathbf{X}\mathbf{w} - \mathbf{y}$$

A matriz $\mathbf{X}$ é a matriz de dados (de entrada), de dimensão $N \times (d+1)$, na qual cada linha corresponde ao vetor $\tilde{\mathbf{x}}_\mathbf{i}^T$. O vetor $\mathbf{y}$ é um vetor coluna de dimensão N. O que queremos de fato é o quadrado dos resíduos. Logo podemos escrever:

$$\begin{bmatrix} (h(\mathbf{x}_1) - y_1)^2 \\ (h(\mathbf{x}_2) - y_2)^2 \\ \vdots \\ (h(\mathbf{x}_N) - y_N)^2 \end{bmatrix} = (\mathbf{Xw} - \mathbf{y})^T(\mathbf{Xw} - \mathbf{y}) = ||\mathbf{Xw} - \mathbf{y}||^2$$

E, portanto, podemos escrever a função de custo como:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{N}||\mathbf{Xw} - \mathbf{y}||^2$$

Para encontrar o vetor $\mathbf{w}$ que corresponde ao ponto de mínimo dessa função devemos resolver

$$\nabla\mathcal{L}(\mathbf{w}) = \frac{2}{N}X^T(\mathbf{Xw} - \mathbf{y}) = 0$$

Temos, então,

$$\mathbf{X}^T\mathbf{Xw} = \mathbf{X}^T\mathbf{y}$$

e portanto

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

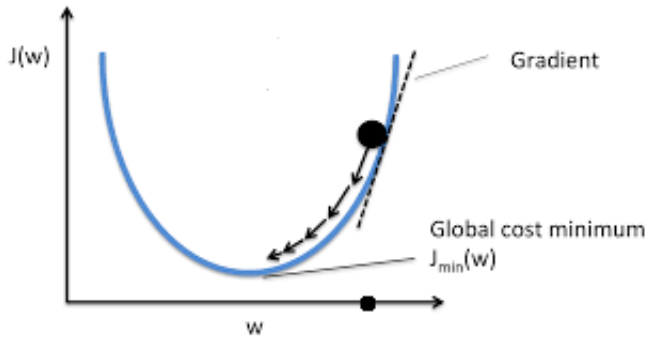
A matriz $(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T$ é denominada pseudo-inversa.

Esta solução requer inicialmente o cálculo do produto $\mathbf{X}^T\mathbf{X}$ (produto de uma matriz $(d+1) \times N$ com uma matriz $N \times (d+1)$) que resulta em uma matriz $(d+1) \times (d+1)$. Essa matriz produto precisa ser invertida, e a complexidade computacional de algoritmos de inversão de matriz é cúbica. Em seguida, mais uma multiplicação de matriz é necessária. Na prática, isso significa que se d e N são grandes, o cálculo dessa solução pode se tornar extremamente demorada, ou até mesmo impossível (por conta de espaço de memória).

2.3 Solução iterativa

Gradiente descendente: O sistema de equações acima pode ser solucionado usando-se outras técnicas. Uma que é bastante conhecida é numérica e iterativa, baseada em gradientes, utilizada para econtrar pontos de mínimo de funções.

No caso de funções quadráticas, uma vez que elas são convexas, garante-se que há um ponto de mínimo global. A ideia da técnica de gradiente descendente é escolher um vetor inicial $\mathbf{w}$, e a cada iteração t atualizá-lo em direção oposta ao do vetor gradiente da função $\mathcal{L}$, calculada no ponto $\mathbf{w}$. A figura a seguir ilustra a ideia. O ponto no eixo x indica o ponto inicial $\mathbf{w}$, e a cada iteração esse ponto é deslocado um pouco para à esquerda, no sentido oposto ao do gradiente de $\mathcal{L}$, e desta forma aproxima-se gradativamente do ponto onde a função $\mathcal{L}$ tem valor mínimo.



Otimização usando gradiente descendente: Note que a função custo (equação 3) é definida como sendo o erro médio sobre os exemplos em D. Por conta disso, há a constante $\frac{1}{N}$ em sua definição. Como trata-se de uma constante positiva, para efeitos de minimização, podemos removê-la ou substitui-la por outra constante positiva.

Para o cálculo do vetor vetor gradiente de $\mathcal{L}$

$$\nabla\mathcal{L}(\mathbf{w}) = \Big(\frac{\partial\mathcal{L}}{\partial w_0}, \frac{\partial\mathcal{L}}{\partial w_1}, \ldots, \frac{\partial\mathcal{L}}{\partial w_d}\Big)$$

é conveniente trocar $\frac{1}{N}$ por $\frac{1}{2}$. Temos, assim, para um componente específico j ($j = 1, 2, \ldots, d$):

$$\begin{aligned}
\frac{\partial\mathcal{L}}{\partial w_j} &= \frac{\partial}{\partial w_j}\frac{1}{2}\sum_i (\hat{y}_i - y_i)^2 \\
&= \frac{1}{2}\sum_i \frac{\partial}{\partial w_j}(\hat{y}_i - y_i)^2 \\
&= \frac{1}{2}\sum_i 2(\hat{y}_i - y_i)\frac{\partial}{\partial w_j}(\hat{y}_i - y_i) \\
&= \sum_i (\hat{y}_i - y_i)\frac{\partial}{\partial w_j}\big((w_0 + w_1 x_{i1} + \ldots + w_j x_{ij} + \ldots + w_d x_{id}) - y_i\big) \\
&= \sum_i (\hat{y}_i - y_i)\, x_{ij}
\end{aligned}$$

O componente j do vetor gradiente é a soma dos componentes j dos exemplos $\mathbf{x}_i$ ponderados por $\hat{y}_i - y_i$.

Algoritmo: O esquema geral está no quadro abaixo. Primeiramente calcula-se o negativo do vetor gradiente ($\Delta w_j = -\frac{\partial\mathcal{L}}{\partial w_j} = \sum_i (y_i - \hat{y}_i)\, x_{ij}$). Em seguida, atualiza-se $\mathbf{w}$, por uma pequena fração do negativo do vetor gradiente, fração especificada pelo hiperparâmetro η (*learning rate*) que usualmente é um valor pequeno (por exemplo 0.001). No esquema abaixo $\mathbf{w}(r)$ indica o vetor $\mathbf{w}$ em uma determinada iteração r.

Initial weight: $\mathbf{w}(0)$

Weight update rule (iteration r):

Compute $\Delta\mathbf{w}(r) = -\nabla\mathcal{L}(\mathbf{w})$

$\mathbf{w}(r+1) = \mathbf{w}(r) + \eta\Delta\mathbf{w}(r)$

Batch gradient descent: Este é o algoritmo que considera o conjunto D completo a cada iteração

Algorithm 1 Batch GradientDescent

Input: D, η, *epochs*
Output: **w**

```
w ← small random value
repeat
    Δw_j ← 0,   j = 0, 1, 2, ..., d
    for all (x, y) in D do
        compute ŷ = w^T x̃
        Δw_j ← Δw_j + (y − ŷ) x_j,   j = 0, 1, 2, ..., d
    end for
    w_j ← w_j + ηΔw_j,   j = 0, 1, 2, ..., d
until number of iterations = epochs
return w
```

Note que para o cômputo de Δ, é preciso processar todos os N exemplos em D para calcular os valores de $\hat{y}_i$, $i = 1, \ldots, N$. Este processo pode ser computacionalmente caro, se N for grande. Assim, há variantes do algoritmo que, em cada iteração de alteração do valor de $\mathbf{w}$, consideram apenas uma parcela dos dados em D.

Stochastic gradient descent: Este é o algoritmo que considera apenas um exemplo a cada iteração de atualização de peso.

Algorithm 2 Stochastic GradientDescent

Input: D, η, *ephocs*

Output: $\mathbf{w}$

$\mathbf{w} \leftarrow$ small random value

repeat

  for all $(\mathbf{x}, y)$ in D **do**

    compute $\hat{y} = \mathbf{w}^T \tilde{\mathbf{x}}$

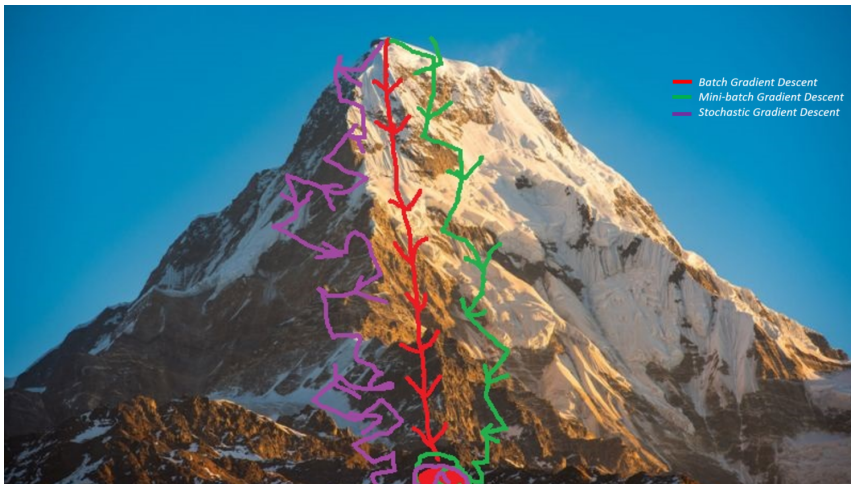
    $w_j \leftarrow w_j + \eta(y - \hat{y})\, x_j, \quad j = 0, 1, 2, \ldots, d$

  end for

until number of iterations = *epochs*

return $\mathbf{w}$

Batches: Além do *Batch gradient descent* $(\Delta w_j(r) = \sum_{i=1}^{N}(y_i - \hat{y}_i)\, x_{ij})$ e do *Stochastic gradient descent* $(\Delta w_j(r) = (y_i - \hat{y}_i)\, x_{ij})$, temos também o *Mini-batch gradient descent* que utiliza um subconjunto de exemplos em D a cada iteração para a atualização de $\mathbf{w}$. Embora tanto a versão estocástica como o mini-batch processem apenas parte de D a cada iteração de atualização de $\mathbf{w}$, ambas tratam de percorrer o conjunto D inteiro. Em geral, o termo **época** refere-se a um percorrimento completo sobre D. A figura 2 ilustra os passos dessas variantes até a convergência. A versão estocástica tipicamente pode demorar por conta do número de iterações (a cada época são N iterações). Por outro lado, a versão *batch* pode demorar pois a cada época os N exemplos precisam ser processados. Na prática, observa-se que *mini-batches* geram um bom balanço entre velocidade de convergência e qualidade das soluções.



Fonte: https://imaddabbura.github.io/post/gradient_descent_algorithms/

Figura 2: Ilustração do processo de convergência ao ponto de mínimo quando comparados as versões *bacth*, *mini-batch* e estocástico.

Finalmente, cabe observar que linguagens de programação como Python ou Matlab oferecem

implementações eficientes de operações com matrizes. Assim, para a implementação desses algoritmos é indicado realizar as computações na forma matricial, sempre que possível.

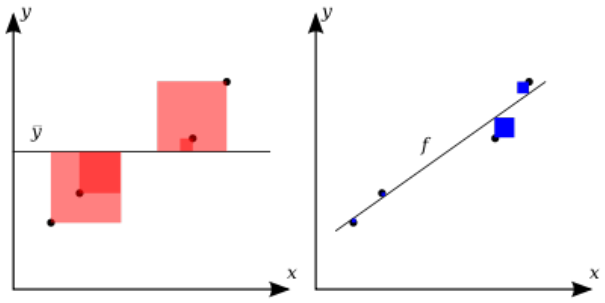
2.4 Avaliação da qualidade de uma regressão

Como podemos avaliar se o resultado da regressão é aceitável ou não? Uma métrica comumente utilizada para isso é o coeficiente de determinação (também chamado de R^2) e definido por

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$

em que $SS_{res} = \sum_{i=1}^{N}(y_i - \hat{y}_i)^2$ é a soma dos erros ao quadrado e $SS_{tot} = \sum_{i=1}^{N}(y_i - \overline{y})^2$ é a variância.

A figura 3 ilustra a ideia do coeficiente de determinação. Note que SS_{res} (que corresponde à área dos quadrados azuis) é muito menor do que SS_{tot} (que corresponde à área dos quadrados vermelhos). Neste caso, R^2 é um valor próximo a 1. Por outro lado, se considerássemos uma reta descendente, SS_{res} teria um valor muito alto, resultado em R^2 negativo.



Fonte: https://en.wikipedia.org/wiki/Coefficient_of_determination

Figura 3: ilustração gráfica de variância e resíduo ao quadrado, usados no cálculo do coeficiente de determinação.

2.5 Comentários finais

Do ponto de vista de ML, o problema de regressão discutido acima pretendeu chamar atenção para dois pontos relevantes:

1. A definição de uma métrica que caracteriza o que é um bom ajuste. Estamos falando sobre a função de perda $\mathcal{L}$. Fazer um bom ajuste significa minimizar essa função. No caso de regressão linear, encontramos comumente o uso do erro quadrátido uma vez que há soluções analíticas para o problema de minimização.

2. A família de funções usadas para o ajuste. Quanto mais expressiva for a família de funções, é de se esperar que ajustes melhores (no sentido de minimização do erro empírico) são possíveis. No entanto, mais adiante iremos discutir o problema do sobreajuste (*overfitting*).

Veremos mais adiante que esses dois pontos irão se manifestar em problemas de classificação. Em outras palavras, de forma geral podemos dizer que o problema de aprendizado supervisionado reduz-se a um problema de escolha de uma família de funções $\mathcal{H}$ e uma função de perda $\mathcal{L}$ adequados. Uma vez realizada essa escolha, a solução do problema em si consiste muitas vezes na solução de um problema de otimização numérica ou matemática.

3 Classificação binária

No caso de problemas de classificação, podemos começar supondo que os dados de entrada são elementos $\mathbf{x} \in \mathbb{R}^d$ e a saída correspondente é um "rótulo" de classe (categoria). Por exemplo, em problemas de classificação binária, no qual cada instância de entrada está associada a uma entre duas classes possíveis, os rótulos comumente utilizados são $\{0, 1\}$ ou $\{-1, +1\}$, denotando as classes negativa e positiva. Em um problema com k possíveis classes, os rótulos costumam ser representados por $\{1, 2, \ldots, k\}$ (em geral não há relação entre o rótulo e o significado da classe).

Em aprendizado de máquina, uma possível abordagem para resolver problemas de classificação binária consiste em determinar uma função discriminante f, tal que $f(\mathbf{x}) < 0$ se $\mathbf{x}$ é uma instância da classe negativa e $f(\mathbf{x}) > 0$ se $\mathbf{x}$ é uma instância da classe positiva. Os pontos $\mathbf{x}$ onde $f(\mathbf{x}) == 0$ são comumente chamados de fronteira de decisão.

3.1 Problemas linearmente separáveis

Para começar, vamos supor um problema de classificação binária em que as classes são linearmente separáveis. Vamos supor também que temos um conjunto $\mathcal{D} = \{(\mathbf{x}_i, y_i) \in \mathbb{R}^d \times \{-1, +1\} : i = 1, \ldots, N\}$ de exemplos. O objetivo neste caso é encontrar uma função linear discriminante $f(\mathbf{x}) = \mathbf{w}^T\tilde{\mathbf{x}}$. Os pontos em que $f(\mathbf{x}) = 0$ formam, neste caso, um hiperplano em $\mathbb{R}^d$.

De fato, o classificador linear pode ser definido a partir da função discriminante da seguinte forma:

$$h(\mathbf{x}) = \text{sign}(\mathbf{w}^T\tilde{\mathbf{x}})$$

Neste caso, temos que $h : \mathbb{R}^d \to \{-1, +1\}$.

Algoritmo perceptron: É um dos primeiros algoritmos criados para classificação binária, para os casos em que as classes são linearmente separáveis.

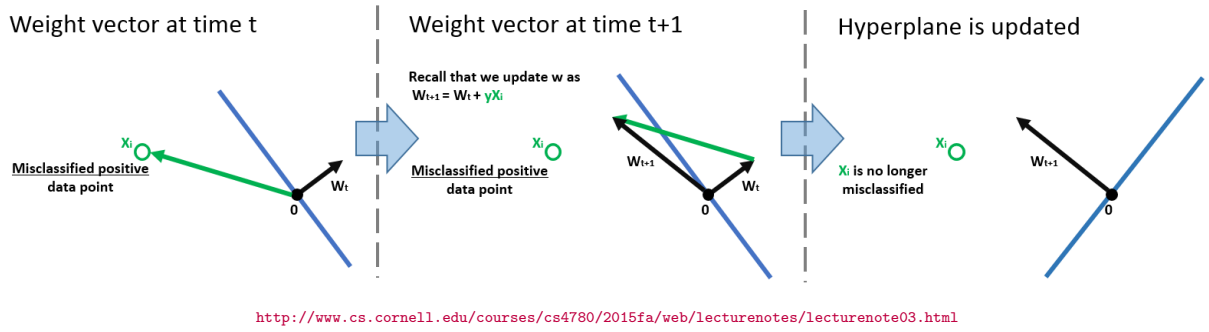
Algorithm 3 Perceptron

Input: $D = \{(\mathbf{x}_i, y_i) \in \mathbb{R}^d \times \{-1, +\} : i = 1, \ldots, N\}$
Output: $\mathbf{w} \in \mathbb{R}^{1+d}$

```
w ← small random value
while there exists (x, y) in D such that sign(w^T x̃) ≠ y do
    w ← w + y x̃
end while
return w
```

Por mais simples que esse algoritmo possa parecer, ele converge. Isto é, após um certo número de iterações, o algoritmo encontra o hiperplano separador (caracterizado pelo $\mathbf{w}$ final).

Para entender a atualização de peso, especificamente a linha "while there exists $(\mathbf{x}, y)$ in D such that sign$(\mathbf{w}^T\tilde{\mathbf{x}}) \neq y$", convém observar a figura 4. Ilustração similar aparece também no slide 12 da Lecture 1 de Yaser Abu-Mostafa.



http://www.cs.cornell.edu/courses/cs4780/2015fa/web/lecturenotes/lecturenote03.html

Figura 4: Efeito da atualização do vetor de pesos $\mathbf{w}$. Na ilustração, o exemplo $\mathbf{x}_i$, positivo, encontra-se no lado negativo da reta (que tem normal $\mathbf{w}_t$). Após a atualização, esse mesmo exemplo fica no lado positivo da nova reta (com normal $\mathbf{w}_{t+1}$).

Convergência do Perceptron: Existe prova para a convergência do algoritmo perceptron. Veja, por exemplo https://www.cse.iitb.ac.in/~shivaram/teaching/old/cs344+386-s2017/resources/classnote-1.pdf ou então o livro "Perceptrons" de Marvin Minsky and Seymour Papert.

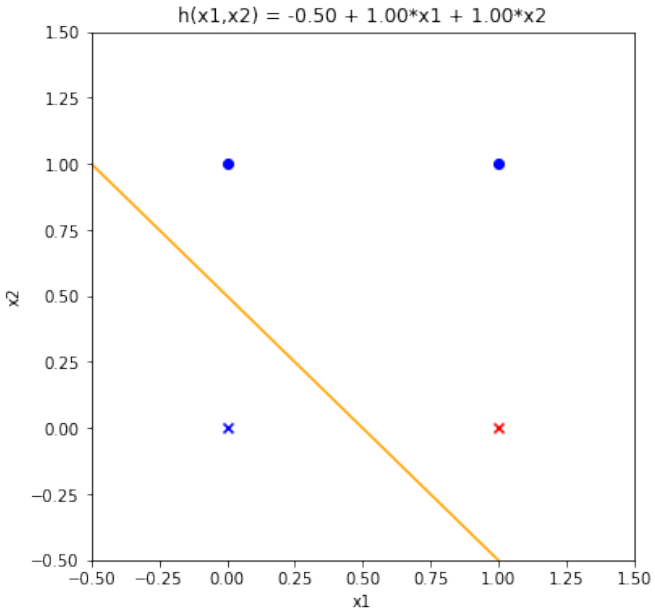
A demonstração parte de alguns pressupostos: assume-se que esiste um hiperplano separador com margem γ, isto é, que existe um vetor de pesos $\mathbf{w}$ de norma 1 (i.e., $||\mathbf{w}|| = 1$) tal que $y_i \mathbf{w}^T \tilde{\mathbf{x}}_i > \gamma$, $\forall i$. Considera-se que R é a maior norma dentre os exemplos $\mathbf{x}_i \in \mathcal{D}$.

Então, denotando o número de iterações do algoritmo por k, a prova consiste em mostrar que k é limitado por $\mathcal{O}(R^2/\gamma^2)$. Para isso, mostra-se que (1) $\| \mathbf{w}^{k+1} \| > k\gamma$ e que (2) $\| \mathbf{w}^{k+1} \|^2 \leq kR^2$, e então de (1) e (2) temos que $k^2\gamma^2 < \| \mathbf{w}^{k+1} \|^2 \leq kR^2 \implies k < \frac{R^2}{\gamma^2}$.

Exercício: Sejam os seguintes quatro exemplos (à esquerda) e o vetor de pesos inicial $\mathbf{w}(0)$ e a representação gráfica dos exemplos no painel à direita. Simule o algoritmo perceptron e encontre uma reta separadora.

$$\mathbf{w}(0) = (-0.5,\ 1,\ 1)$$

$x_1\ x_2$	y
0 0	+1
0 1	+1
1 0	−1
1 1	+1



3.2 Problemas não linearmente separáveis

O perceptron funciona apenas quando os dados são linearmente separáveis. Logo, na prática ele não tem muita utilidade uma vez que em geral não é possível saber a priori se um conjunto de dados é linearmente separável. O seu valor é muito mais didático do que prático.

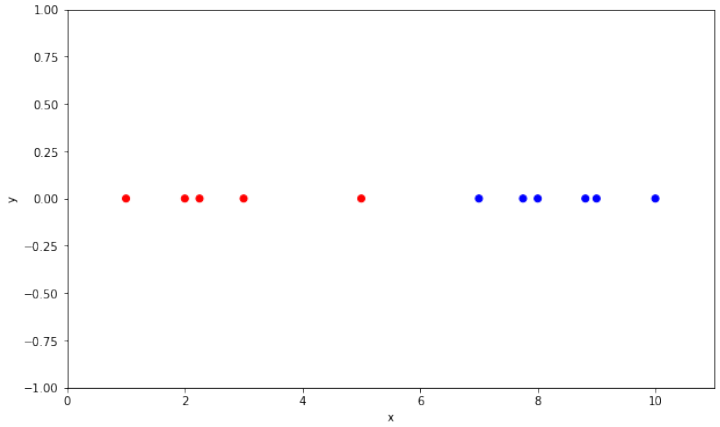
Adaptação do perceptron Uma forma de utilizá-lo seria fazendo uma pequena adaptação: fixa-se o número máximo de iterações do algoritmo e, a cada iteração verifica-se se o hiperplano correspondente ao vetor de pesos $\mathbf{w}$ atual apresenta um erro de classificação menor em relação ao melhor resultado obtido até a iteração atual. Em caso positivo, troca-se o vetor de pesos previamente armazenado em lugar especial pelo peso atual (indicando que até o presente momento o melhor resultado foi obtido com este último vetor de peso). Desta forma, garante-se que ao final da simulação tem-se armazenado o peso do hiperplano que corresponde à melhor separação constatada ao longo das iterações. Esta variante é denominada por Yaser Abu-Mostafa como Pocket Algorithm. A solução pode ser aceitável nos casos em que os dados são aproximadamente linearmente separáveis.

Existem porém algoritmos que conseguem tratar problemas não linearmente separáveis. Estamos falando de algoritmos que encontram um "melhor" hiperplano, isto é, um hiperplano com erro mínimo. Para explicar esse algoritmo, convém antes especularmos se faria sentido transformar um problema de classificação binária em um problema de regressão linear.

3.3 Transformação de um problema de classificação binária em um problema de regressão linear

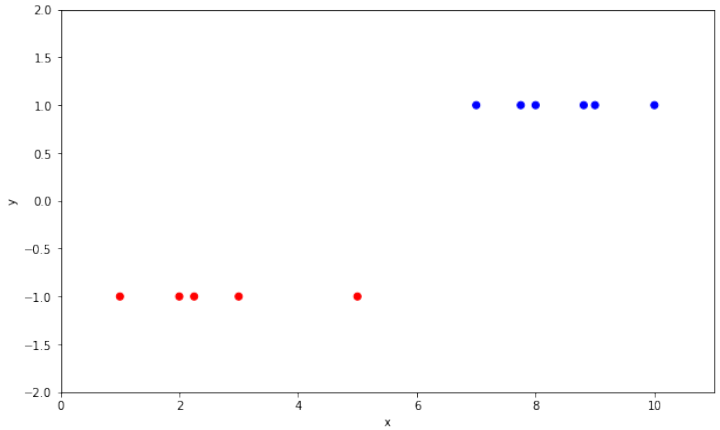
Apesar de o alvo em problemas de classificação ser um valor discreto (no caso da classificação binária é, por exemplo, -1 ou +1), podemos pensá-los como um problema de regressão. Vejamos o exemplo a seguir.

Suponha que temos dados de dimensão $d = 1$, $D_X = \{1, 2, 2.25, 3, 5, 7, 7.75, 8, 8.81, 9, 10\}$. No gráfico a seguir, esses pontos estão dispostos no eixo x. A cor azul indica exemplos da classe positiva e a cor vermelha os da classe negativa.



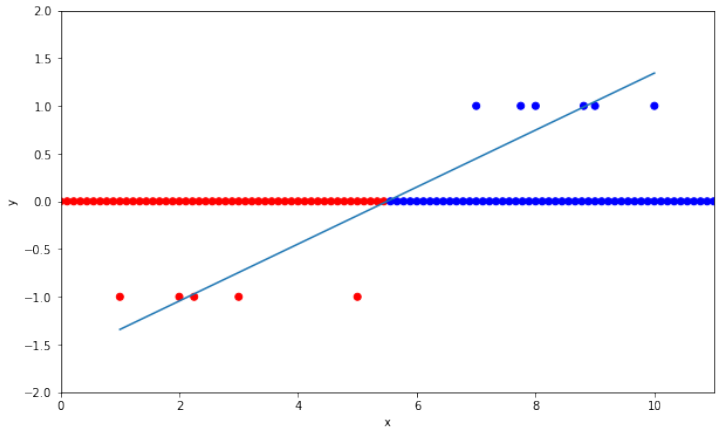
Um corte em torno de $x = 6$ é suficiente para separar os negativos dos positivos.

Aproveitando o fato de que positivos tem rótulo $+1$ e negativos tem rótulo -1, vamos representar os pares $(\mathbf{x}_i, y_i)$ no gráfico, usando y_i como a coordenada no eixo y. Temos então os pontos espalhados na seguinte forma:



Aplicar uma regressão linear não faz muito sentido, pois não vemos uma função linear como um ajuste natural quando pensamos na relação entre $\mathbf{x}$ e y. O que parece mais natural seria o ajuste de uma função constante ($h_-(x) = -1$ para $x < 6$ e $h_+(x) = +1$ para $x > 6$).

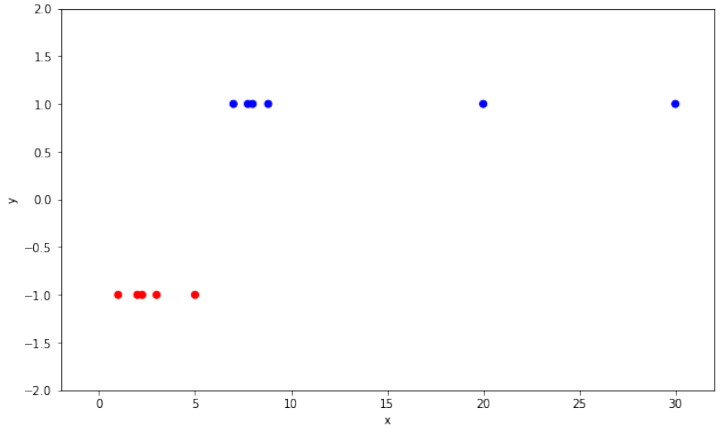
Deixando de lado a questão do natural ou não, o que acontece se simplesmente aplicamos a regressão linear sobre os exemplos do gráfico acima? Obteremos uma reta h representada pela linha azul a seguir:



A partir desse gráfico, poderíamos então calcular sign($h(\mathbf{x})$) para decidir se um certo $\mathbf{x}$ é da classe positiva ou negativa. No gráfico acima, a parte equerda do eixo x está em vermelho para indicar que todos os exemplos naquela região ($h(\mathbf{x}) < 0$) será classificado como da classe negativa. Similarmente, todos os pontos à direita, na região azul ($h(\mathbf{x}) > 0$) serão classificados como positivos.

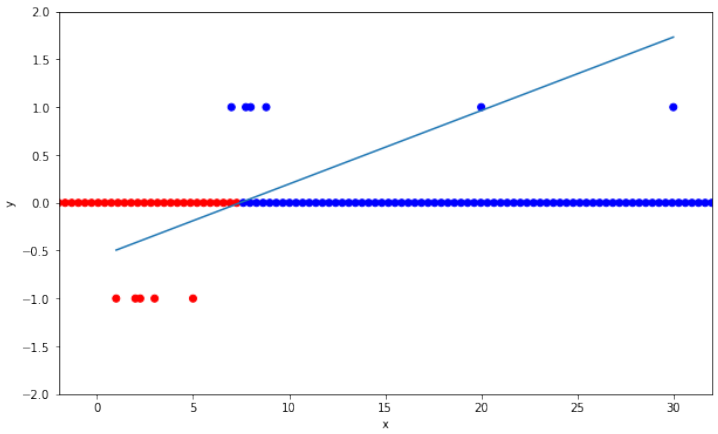
Em particular, a solução para o problema de classificação parece muito bom. O ponto de corte no eixo x está próximo de 6, como havíamos sugerido anteriormente baseado em uma inspeção visual.

Examinemos agora um segundo exemplo. Vamos considerar um conjunto de exemplos parecido ao anterior, porém com exemplos mais à direita no eixo x: $D_2 = \{1, 2, 2.25, 3, 5, 7, 7.75, 8, 8.81, 20, 30\}$. O gráfico correspondente está abaixo:

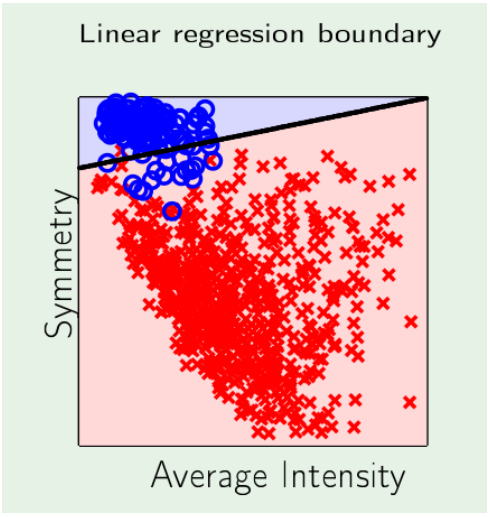


Ao aplicarmos a regressão linear, temos agora uma reta que corta o eixo x no ponto próximo

a 7.5 (ou seja, a fronteira de decisão é deslocada para à direita). Com isso, o classificador resultante irá errar a classificação dos exemplos positivos menores que $x = 7.5$.



Este exemplo mostra o que já discutimos anteriormente sobre a regressão linear: os outliers (ou pontos que aparentemente não fazem parte da distribuição) podem afetar "muito" o ajuste (isto ocorre pois o ajuste tenta minimizar o MSE, e outliers contribuem com um erro muito grande). Além disso, classes desbalanceadas também podem fazer a fronteira de decisão deslocar-se em relação ao local ideal, como no exemplo a seguir (a classe negativa é muito maior; note que os dados de entrada são 2D e que a fronteira de decisão – a linha em preto – é a interseção entre o plano de regressão no espaço 3D e o plano xy):



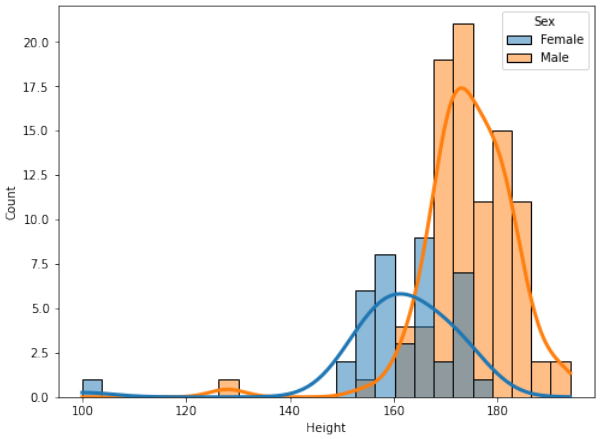
Estes dois últimos exemplos mostram que a aplicação ingênua de regressão linear em problemas de classificação binária, como forma para encontrar uma fronteira de decisão (no espaço $\mathbf{x}$), em geral gera resultados não-ótimos. De fato, existem outras técnicas que geram fronteiras ótimas. Veremos a seguir a técnica conhecida por **regressão logística**.

4 Regressão logística

Suponha que sabemos qual é a altura de uma pessoa. Será que conseguimos advinhar se essa pessoa é do sexo feminino ou masculino? Em geral, não conseguiremos. Porém, se pensarmos em média de acertos, isto é, em repetir esse experimento várias vezes, será que existe uma regra que nos leva a uma maior taxa de acerto ?

Na figura 5, no painel da esquerda vemos os histogramas de alturas de pessoas do sexo feminino (em azul) e as de sexo masculino (em laranja). Supondo que esses histogramas representem o grupo de pessoas, é um tanto intuitivo que para alturas menores que 160cm aumentamos a chance de acerto se chutamos que a pessoa é do sexo feminino e, para alturas maiores que 170cm, que é do sexo masculino. Há uma faixa de alturas no meio que é mais confusa. De qualquer forma, o ponto de corte (fronteira) ótimo está localizado nessa região.

Se soubéssemos a distribuição de probabilidade, poderíamos calcular a fronteira ótima. Uma formulação comum é por meio da Teoria de decisão Bayesiana. No painel à direita da figura 5 está o Teorema de Bayes (adaptado para o contexto de classificação) e abaixo dele está a regra ótima de decisão.



Teorema de Bayes

$$P(y \mid \mathbf{x}) = \frac{P(y)\, p(\mathbf{x} \mid y)}{p(\mathbf{x})}$$

Regra de decisão que minimiza a probabilidade de erro:

$$y^* = \arg\max_y \{P(y|\mathbf{x})\}$$

Figura 5

No exemplo da figura 5, se tomarmos as curvas laranja e azul como sendo a função densidade de probabilidade das classes Male e Female, respectivamente, já "ponderados" por $P(y = \text{Male})$ e $P(y = \text{Female})$, então o ponto de corte ótimo estaria em torno de 165 que é precisamente o ponto onde $P(y = \text{Male} \mid \mathbf{x}) = P(y = \text{Female} \mid \mathbf{x})$.

No exemplo que consideramos temos $y \in \{\text{Female}, \text{Male}\}$ e $x \in \mathbb{R}$. Cabe notar, porém, que o teorema e a regra de decisão valem para casos de múltiplas classes e com $x \in \mathbb{R}^d$ $(d > 1)$. A regra simplesmente diz que para minimizar a probabilidade de erro, basta predizer que uma entrada $\mathbf{x}$ é da classe que tem a maior probabilidade a posteriori $P(y|\mathbf{x})$. Mais detalhes e uma demonstração da regra de Bayes podem ser encontrados por exemplo em [2] (capítulo 2).

Na prática, não temos conhecimento sobre essas distribuições. Tudo que temos são exemplos $\mathcal{D} = \{(\mathbf{x}_i, y_i) \in \mathcal{X} \times \mathcal{Y}, i = 1, \ldots, N\}$, que seguem uma distribuição desconhecida $P(\mathbf{x}, y)$. Assim, qualquer informação sobre $\mathcal{X} \times \mathcal{Y}$ terá que ser extraída a partir de D.

Abordagens gerativa e discriminativa: Uma possibilidade para resolver o problema consiste em estimar a distribuição de probabilidade $P(\mathbf{x}, y)$ (ou equivalentemente $P(y)\, p(\mathbf{x} \,|\, y)$), calcular $P(y \,|\, \mathbf{x})$ e então aplicar a regra de decisão de Bayes. Essa solução faz parte da **abordagem gerativa** pois busca modelar explicitamente o processo de geração dos dados. Uma segunda possibilidade seria estimar diretamente a probabilidade a posteriori $P(y \,|\, \mathbf{x})$. Soluções deste tipo fazem parte da chamada **abordagem discriminativa** (ou também *distribution-free*).

Veremos a seguir uma abordagem discriminativa.

4.1 Formulação com $y \in \{-1, +1\}$

A regressão logística é um abordagem discriminativa, de acordo com o que foi discutido acima, e consiste em aproximar $P(y \,|\, \mathbf{x})$.

Em outras palavras, iremos considerar que nosso alvo é uma função $f(\mathbf{x}) = P(y = +1 \,|\, \mathbf{x})$. Desta forma, podemos escrever

$$P(y \,|\, \mathbf{x}) = \begin{cases} f(\mathbf{x}), & \text{if } y = +1, \\ 1 - f(\mathbf{x}), & \text{if } y = -1 \end{cases}$$

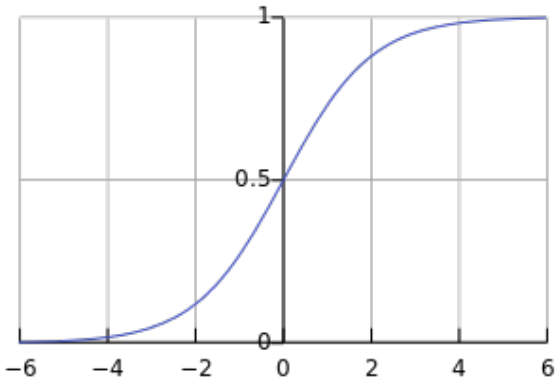
Note que na prática não temos acesso a $f(\mathbf{x})$, mas apenas aos rótulos $+1$ ou -1. No entanto, se conseguirmos "aprender" a função f, isso significa que aprendemos a distribuição de probabilidade $P(y \,|\, \mathbf{x})$.

Família de funções: Pensando em uma formulação como um problema de aprendizado supervisionado, uma importante escolha é a família de funções a serem usadas para aproximar f. Uma vez que f é tal que $0 \leq f(\mathbf{x}) \leq 1$, vamos considerar uma família do mesmo tipo:

$$h(\mathbf{x}) = \theta(\mathbf{w}^T \tilde{\mathbf{x}})$$

em que $\theta(z) = \dfrac{1}{1 + e^{-z}} = \dfrac{e^z}{e^z + 1}$.

A função θ, denominada sigmóide ou logística, é ilustrada abaixo.



Como pode ser visto, temos que $0 \leq \theta(z) \leq 1$. Logo, segue que $0 \leq h(\mathbf{x}) \leq 1$.

Se $h(\mathbf{x}) \approx f(\mathbf{x})$, então

$$\hat{P}(y|\mathbf{x}) = \begin{cases} h(\mathbf{x}), & \text{if } y = +1, \\ 1 - h(\mathbf{x}), & \text{if } y = -1 \end{cases}$$

será um boa aproximação de $P(y \,|\, \mathbf{x})$ e uma decisão quanto à classificação final pode ser baseada na aproximação obtida.

Um truque interessante é usar o fato de que $1 - \theta(z) = \theta(-z)$ e escrever $P(y \,|\, \mathbf{x}) = \theta(y\, \mathbf{w}^T \tilde{\mathbf{x}})$. Essa notação é conveniente pois permite tratarmos os casos em que $y = +1$ e $y = -1$ sem tratar cada caso individualmente.

Função de perda: Além da família de funções para aproximar f, precisamos também definir uma função de perda que servirá de guia para escolher uma função $h \in \mathcal{H}$ que melhor aproxima f.

Enquanto na regressão linear temos acesso ao valor $y_i = f(\mathbf{x}_i)$ para cada exemplo $\mathbf{x}_i$, no problema de classificação binária apenas sabemos que $y_i = +1$ ou $y_i = -1$, mas não temos acesso a $f(\mathbf{x}_i) = P(y = +1 \,|\, \mathbf{x}_i)$.

Especificamente, temos $\mathcal{D} = \{(\mathbf{x}_i, y_i) \in \mathcal{X} \times \{-1, +1\}, i = 1, \ldots, N\}$. Vamos supor que esses exemplos são observações i.i.d. de uma distribuição $P(\mathbf{x}, y)$.

Logo, a probabilidade de se observar os exemplos em D pode ser expresso pela função de verossimilhança:

$$\prod_{i=1}^{N} P(\mathbf{x} = \mathbf{x}_i, y = y_i) \tag{4}$$

[Possivelmente, as notações não estarão 100% precisas. Por exemplo, em vez de $P(\mathbf{x} = \mathbf{x}_i, y = y_i)$ mais adiante irá aparecer bastante $P(\mathbf{x}_i, y_i)$. Também já vi que usei $p(\cdot)$ e $P(\cdot)$ ...]

Se imaginarmos todas as possíveis distribuições $P(\mathbf{x}, y)$, o valor da probabilidade na equação 4 irá variar, dependendo de qual distribuição é considerada. É razoável supor que existe alguma

distribuição $P(\mathbf{x}, y)$ em particular que resultará em máxima probabilidade. Se precisássemos fazer um exercício de engenharia reversa, de escolher qual é a distribuição que está governando as observações em D, parece bastante natural que uma escolha óbvia seja a distribuição que maximiza a probabilidade especificada na equação 4.

Esta é a ideia do princípio da máxima verossimilhança, que leva à estimação por máxima verossimilhança. Assumimos uma distruição paramétrica e então encontramos os valores dos parâmetros que correspondem a uma distribuição específica, aquela que maximiza a probabilidade de se observar os dados em D. Em nosso caso, supondo que $\mathbf{w}$ é o conjunto de parâmetros, que caracterizam distribuições $P(\mathbf{x}, y; \mathbf{w})$, o ajuste que queremos fazer consiste em encontrar $\mathbf{w}$ que maximiza a função de verossimilhança $\prod_{i=1}^{N} P(\mathbf{x}_i, y_i; \mathbf{w})$.

Observe que $P(\mathbf{x}, y) = P(y|\mathbf{x})\, P(\mathbf{x})$. Como na nossa formulação apenas $P(y|\mathbf{x})$ é afetado por $\mathbf{w}$, podemos simplificar a função de verossimilhança e, usando o fato $P(y \,|\, \mathbf{x}) = \theta(y\, \mathbf{w}^T \tilde{\mathbf{x}})$, escrever:

$$\prod_{i=1}^{N} P(y_i|\mathbf{x}_i) = \prod_{i=1}^{N} \theta(y_i\, \mathbf{w}^T \tilde{\mathbf{x}}_i)$$

O problema de otimização: De acordo com o discutido acima, desejamos encontrar $\mathbf{w}$ que maximiza a função de verossimilhança. Isto é, queremos maximizar

$$\prod_{i=1}^{N} \theta(y_i\, \mathbf{w}^T \tilde{\mathbf{x}}_i)$$

Como a função ln é monotônica, e N é constante, maximizar a função acima é equivalente a maximizar (não sei exatamente qual a motivação da divisão por N ...)

$$\frac{1}{N} \ln \Big(\prod_{i=1}^{N} \theta(y_i\, \mathbf{w}^T \tilde{\mathbf{x}}_i) \Big)$$

que por sua vez é equivalente a minimizar

$$-\frac{1}{N} \ln \Big(\prod_{i=1}^{N} \theta(y_i\, \mathbf{w}^T \tilde{\mathbf{x}}_i) \Big)$$

Uma vez que $\ln \prod a_i = \sum \ln a_i$, podemos reescrever a expressão acima como

$$-\frac{1}{N} \sum_{i=1}^{N} \ln \Big(\theta(y_i\, \mathbf{w}^T \tilde{\mathbf{x}}_i) \Big)$$

e como $\ln \frac{1}{a} = -\ln a$, segue que ela é igual a

$$\frac{1}{N}\sum_{i=1}^{N} \ln\Big(\frac{1}{\theta(y_i\, \mathbf{w}^T\tilde{\mathbf{x}}_i)}\Big)$$

Usando o fato de que $\frac{1}{\theta(z)} = \frac{1}{\frac{1}{1+e^{-z}}}$ segue que a última é equivalente a

$$\frac{1}{N}\sum_{i=1}^{N} \ln\Big(1 + e^{-y_i\, \mathbf{w}^T\tilde{\mathbf{x}}_i}\Big)$$

Assim, a função de perda ou custo a ser minimizada na regressão logística é

$$\mathcal{L}(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N} \underbrace{\ln\Big(1 + e^{-y_i\, \mathbf{w}^T\tilde{\mathbf{x}}_i}\Big)}_{err(y_i,\hat{y}_i)}$$

Vamos tentar entender o que essa expressão significa. Quando os sinais de y_i e $\mathbf{w}^T\tilde{\mathbf{x}}_i$ são iguais, o expoente em $e^{-y_i\, \mathbf{w}^T\tilde{\mathbf{x}}_i}$ é negativo, o que faz com que $err(y_i, \hat{y}_i)$ seja pequeno (próximo de zero). Por outro lado, quando y_i and $\mathbf{w}^T\tilde{\mathbf{x}}_i$ possuem sinais opostos, o expoente em $e^{-y_i\, \mathbf{w}^T\tilde{\mathbf{x}}_\mathbf{i}}$ é positivo e portanto $err(y_i, \hat{y}_i)$ tende a ser grande.

Otimização usando gradiente descendente: Vamos calcular o gradiente da função de custo acima.

Primeiro, note que $\frac{\partial}{\partial \mathbf{w}}[\ln[f(x)] = \dfrac{f'(x)}{f(x)}$. Assim, temos que

$$\frac{\partial}{\partial \mathbf{w}}[\ln(1 + e^{\mathbf{w}^T\mathbf{s}})] = \frac{(1 + e^{\mathbf{w}^T\mathbf{s}})'}{1 + e^{\mathbf{w}^T\mathbf{s}}} = \frac{\mathbf{s}\, e^{\mathbf{w}^T\mathbf{s}}}{1 + e^{\mathbf{w}^T\mathbf{s}}} = \mathbf{s}\frac{e^{\mathbf{w}^T\mathbf{s}}}{1 + e^{\mathbf{w}^T\mathbf{s}}} = \mathbf{s}\frac{1}{1 + e^{-\mathbf{w}^T\mathbf{s}}}$$

Logo, se tomarmos $\mathbf{s} = -y\tilde{\mathbf{x}}$, segue que

$$\frac{\partial}{\partial \mathbf{w}}[\ln\Big(1 + e^{-y\, \mathbf{w}^T\tilde{\mathbf{x}}}\Big)] = \frac{\partial}{\partial \mathbf{w}}[\ln\Big(1 + e^{\mathbf{w}^T\mathbf{s}}\Big)] = \mathbf{s}\frac{1}{1 + e^{-\mathbf{w}^T\mathbf{s}}} = -\frac{y\tilde{\mathbf{x}}}{1 + e^{y\mathbf{w}^T\tilde{\mathbf{x}}}}$$

O algoritmo está ilustrado abaixo:

Logistic regression algorithm

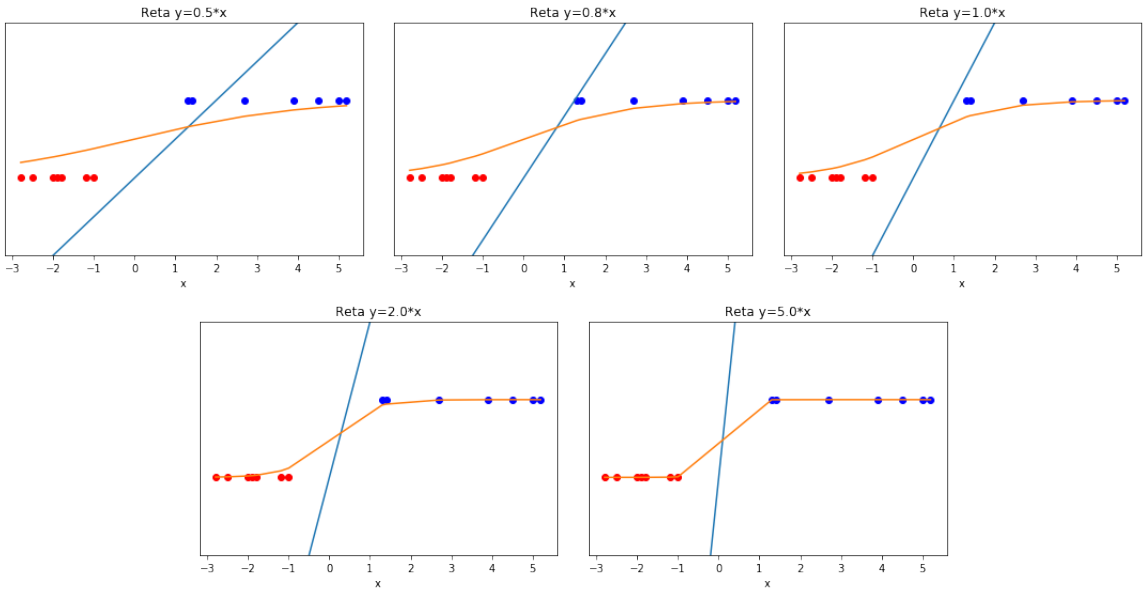
1. Initialize the weights at $t = 0$ to $\mathbf{w}(0)$
2. **for** $t = 0, 1, 2, \ldots$ **do**
3. Compute the gradient

$$\nabla E_{\text{in}} = -\frac{1}{N}\sum_{n=1}^{N}\frac{y_n \mathbf{x}_n}{1 + e^{y_n \mathbf{w}^{\intercal}(t)\mathbf{x}_n}}$$

4. Update the weights: $\mathbf{w}(t+1) = \mathbf{w}(t) - \eta \nabla E_{\text{in}}$
5. Iterate to the next step until it is time to stop
6. Return the final weights $\mathbf{w}$

Retirado dos slides 09 da Caltech

A figura 6 mostra como fica o ajuste quando diferentes valores de $\mathbf{w}$ são considerados.



Reta azul: $\mathbf{w}^T \tilde{\mathbf{x}} = w_0 + w_1 x = 0$ Curva laranja: $h(\mathbf{x}) = \theta(\mathbf{w}^T \tilde{\mathbf{x}})$

Figura 6: Distintas retas geram diferentes ajustes. No exemplo, quanto mais vertical é a reta definida por $h(\mathbf{x}) = 0$, menor o custo.

Exercício: Mostre que a fronteira de decisão gerada pelo algoritmo de regressão logística é um hiperplano.

Comentários: Note que $y_i \in \{-1, +1\}$ enquanto $\hat{y}_i = h(\mathbf{x}) \in [0, 1]$. Esta pode ser vista como a razão de, embora estarmos trabalhando com problemas de classificação, o método de ajuste

visto ser chamado de regressão logística (o termo logística vem do fato de ser empregado a função logística).

Na literatura frequentemente encontramos formulações de classificação binária na qual as classes são representadas pelos rótulos $y \in \{0, 1\}$.

Tipos de erros de classificação: O algoritmo de regressão logística devolve $h(\mathbf{x}_i) = \theta(\mathbf{w}^T \tilde{\mathbf{x}}_i) \approx P(y_i = +1|\mathbf{x}_i)$. A partir disso, em geral atribui-se a classificação $+1$ se $h(\mathbf{x}) > 0.5$ e -1 em caso contrário. Feita a classificação, os seguintes erros podem ocorrer:

		Actual	
		Positive	Negative
Predicted	Positive	**True Positive**	**False Positive**
	Negative	**False Negative**	**True Negative**

Em algumas situações, os dois tipos de erro estão associados a gravidades distintas. Por exemplo, em caso de ambientes com acesso restrito, um sistema que controla o acesso por meio de recomhecimento biométrico poderia ser rigoroso e aceitar somente pessoas que são reconhecidas com confiança $h(\mathbf{x}) > 0.8$. Neste caso, talvez o sistema barre o acesso a pessoas que de fato têm direito ao acesso, causando alguma inconveniência, mas diminui drasticamente o acesso de pessoas estranhas ao ambiente. Tal medida pode ser necessária quando o acesso ao ambiente envolve questões de segurança.

4.2 Formulação com $y \in \{0, 1\}$

A formulação acima descrita, considerando $y \in \{-1, +1\}$ pode ser adaptada para o caso em que consideramos $y \in \{0, 1\}$. Em essência, trata-se do mesmo problema, porém por conta de alterações nos possíveis valores de y, uma técnica ligeiramente distinta é utilizada para escrever $P(y|\mathbf{x})$, para não tratar os casos $P(y = 0|\mathbf{x})$ e $P(y = 1|\mathbf{x})$ de forma individualizada.

Observe que, para $y \in \{0, 1\}$, podemos escrever

$$\begin{aligned} P(y \,|\, \mathbf{x}) &= P(y = 1|\mathbf{x})^y \, P(y = 0|\mathbf{x})^{1-y} \\ &= P(y = 1|\mathbf{x})^y \, [1 - P(y = 1|\mathbf{x})]^{1-y} \end{aligned}$$

Logo, a função de verossimilhança pode ser escrita como

$$\begin{aligned}\prod_{(\mathbf{x}_i,y_i)\in D} P(y_i|\mathbf{x}_i) &= \prod_{(\mathbf{x}_i,y_i)\in D} P(y_i=1|\mathbf{x}_i)^{y_i}\left[1-P(y_i=1|\mathbf{x}_i)\right]^{1-y_i} \\ &\approx \prod_{(\mathbf{x}_i,y_i)\in D} [\theta(\mathbf{w}^T\tilde{\mathbf{x}}_\mathbf{i})]^{y_i}\left[1-\theta(\mathbf{w}^T\tilde{\mathbf{x}}_\mathbf{i})\right]^{1-y_i} \\ &= \prod_{(\mathbf{x}_i,y_i)\in D} \hat{y}_i^{y_i}\,(1-\hat{y}_i)^{1-y_i}\end{aligned}$$

Seguindo os mesmos argumentos vistos anteriormente, temos que a maximização da função acima é equivalente à minimização de

$$\begin{aligned}&-\ln \prod_{(\mathbf{x},y)\in D} \hat{y}^y\,(1-\hat{y})^{1-y} \\ &-\sum_{(\mathbf{x},y)\in D} \ln\left(\hat{y}^y\,(1-\hat{y})^{1-y}\right) \\ &-\sum_{(\mathbf{x},y)\in D} \ln(\hat{y}^y) + \ln((1-\hat{y})^{1-y}) \\ &-\sum_{(\mathbf{x},y)\in D} y\ln\hat{y} + (1-y)\ln(1-\hat{y})\end{aligned}$$

A função de custo resultante,

$$\mathcal{L}(\mathbf{w}) = -\frac{1}{N}\sum_{i=1}^{N} y_i\ln\hat{y}_i + (1-y_i)\ln(1-\hat{y}_i)$$

em que $\hat{y}_i = \theta(\mathbf{w}^T\tilde{\mathbf{x}}_i)$, é denominada **cross-entropy loss**.

Novamente, vamos analisar o que significa essa função de perda. Quando $y_i = 1$ ($\mathbf{x}_i$ é da classe positiva) e $\hat{y}_i$ é próximo a 1, temos que $\ln\hat{y}_i \approx 0$ o que faz o $y_i\ln\hat{y}_i \approx 0$. Ao mesmo tempo, o segundo termo $(1-y_i)\ln(1-\hat{y}_i)$ é igual a zero. Quando $y_i = 0$ ($\mathbf{x}_i$ é da classe negativa) e $\hat{y}_i$ é próximo a 0, temos uma situação similar: o termo $y_i\ln\hat{y}_i$ é igual a 0 e o $1-\hat{y}_i$ é próximo de 1, fazendo com que o segundo termo seja próximo de zero. Por outro lado, se y_i e $\hat{y}_i$ não forem próximos, um dos termos pode ter um valor grande. Assim, quando minimizamos a função, estamos de fato forçando $\hat{y}_i$ a se aproximar de y_i.

Otimização usando gradiente descendente: O gradiente da função *cross-entropy loss* é dada por

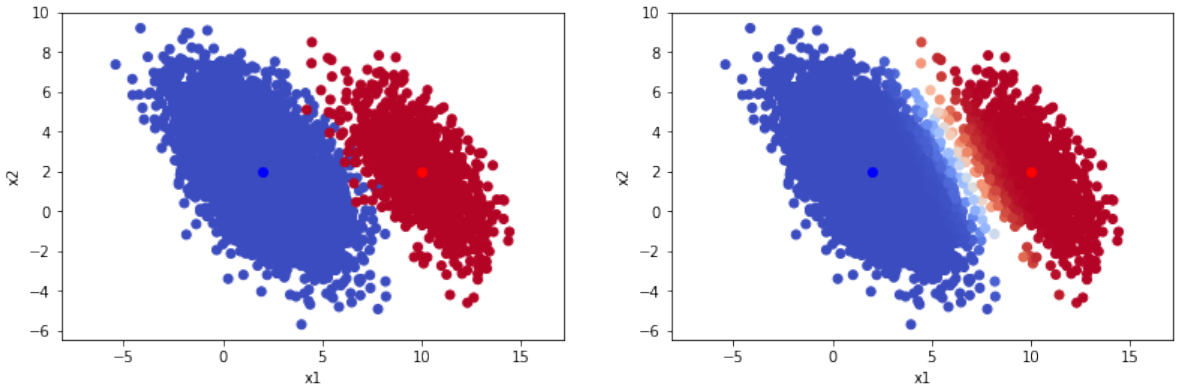
$$\frac{\partial}{\partial w_j}\mathcal{L}(\mathbf{w}) = \sum_{i=1}^{N}(\hat{y}_i - y_i)x_{ij}, \quad j = 1,\ldots,d$$

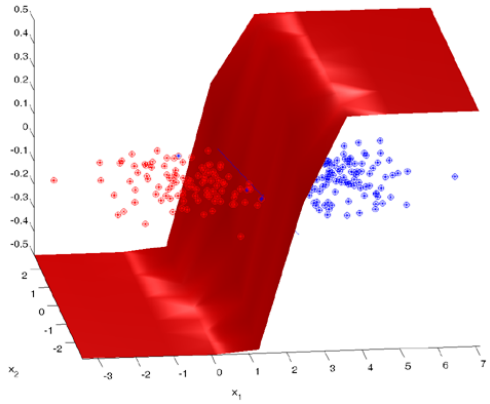
(a derivação do gradiente fica como exercício)

A expressão é a mesma do caso da regressão linear, mas é importante notar que no caso da regressão linear tínhamos $\hat{y}_i = \mathbf{w}^T\tilde{\mathbf{x}}_i$ enquanto no caso da regressão logística temos $\hat{y}_i = \theta(\mathbf{w}^T\tilde{\mathbf{x}}_i)$.

4.3 Algumas figurinhas :-)

Abaixo uma ilustração do problema de separação de dois grupos de pontos, ambos com uma mesma distribuição gaussiana, apenas com ponto médio em posições distintas. A fronteira de decisão e também o efeito da função sigmoide mostrado em 3D.





Source: http://strijov.com/sources/demoDataGen.php

Exercício: Mostre que a fronteira de decisão gerada pelo classificador logístico é sempre linear.

5 Não-linearidade e multiclassificação

Nas seções anteriores vimos soluções lineares para o problema de regressão e de classificação binária. Apenas para recordar:

- Classificação binária linear (caso do perceptron): usamos $h(\mathbf{x}) = \text{sign}(\mathbf{w}^T\tilde{\mathbf{x}})$
- Regressão linear: usamos $h(\mathbf{x}) = \mathbf{w}^T\tilde{\mathbf{x}}$

- Classificação binária (regressão logística): usamos $h(\mathbf{x}) = \theta(\mathbf{w}^T\tilde{\mathbf{x}})$

Nesta seção veremos como podemos utilizar os algoritmos vistos nas seções anteriores para encontrar fronteiras não-lineares ou então solucionar problemas de classificação com mais de duas classes.

5.1 Linear × Non-linear

Quando falamos de funções lineares, estamos nos referindo a funções da forma

$$h(\mathbf{x}) = w_1x_1 + w_2x_2 + \ldots + w_dx_d + w_0$$

Se pensarmos s como uma função discriminante, para classificar um exemplo $\mathbf{x}$, basta verificamos se $h(\mathbf{x}) > 0$ ou $h(\mathbf{x}) < 0$.

Funções não-lineares incluem, por exemplo, termos produto envolvendo ao menos duas ocorrências de variáveis como nos seguintes casos:

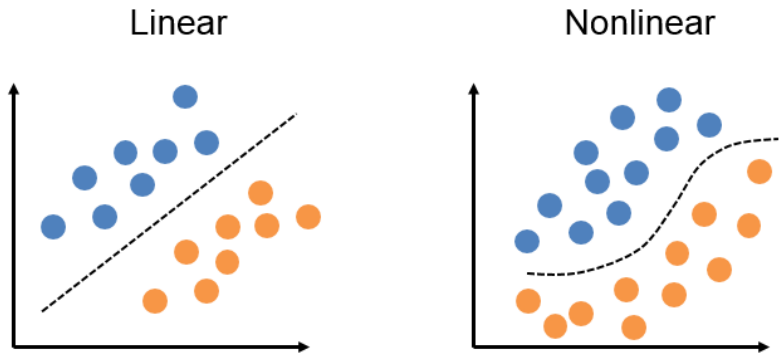
$$h(\mathbf{x}) = w_1x_1 + w_2x_2 + w_3x_1x_2 + w_4x_1^2 + w_5x_2^2 + b$$

ou

$$h(\mathbf{x}) = w_1x_1^2 + w_2x_2^2 + b$$

Não importa h, se linear ou não linear, podemos sempre fazer

- $h(\mathbf{x}) < 0 \implies$ class := negative
- $h(\mathbf{x}) > 0 \implies$ class := positive
- $h(\mathbf{x}) = 0 \implies$ decision boundary



Fonte: https://jtsulliv.github.io/perceptron/

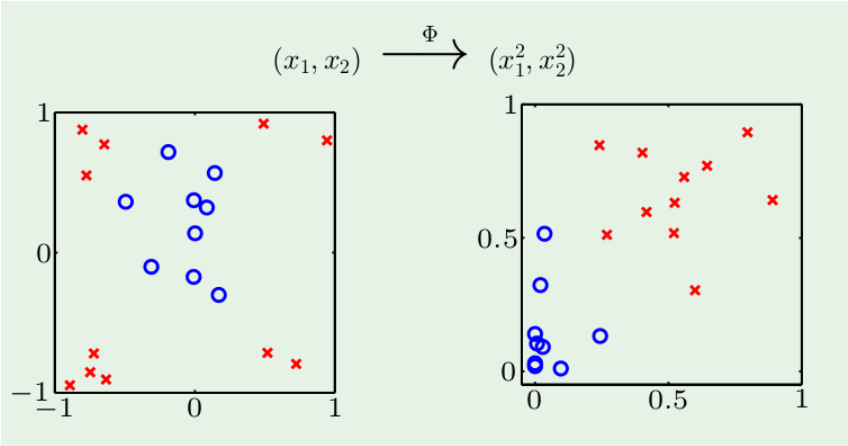
Até agora vimos algoritmos que encontram superfícies de decisão lineares. Não sabemos ainda como ecnontrar superfícies não-lineares. Veremos que com o que sabemos, já podemos gerar superfícies não lineares.

Para isso, tomemos a função a seguir:

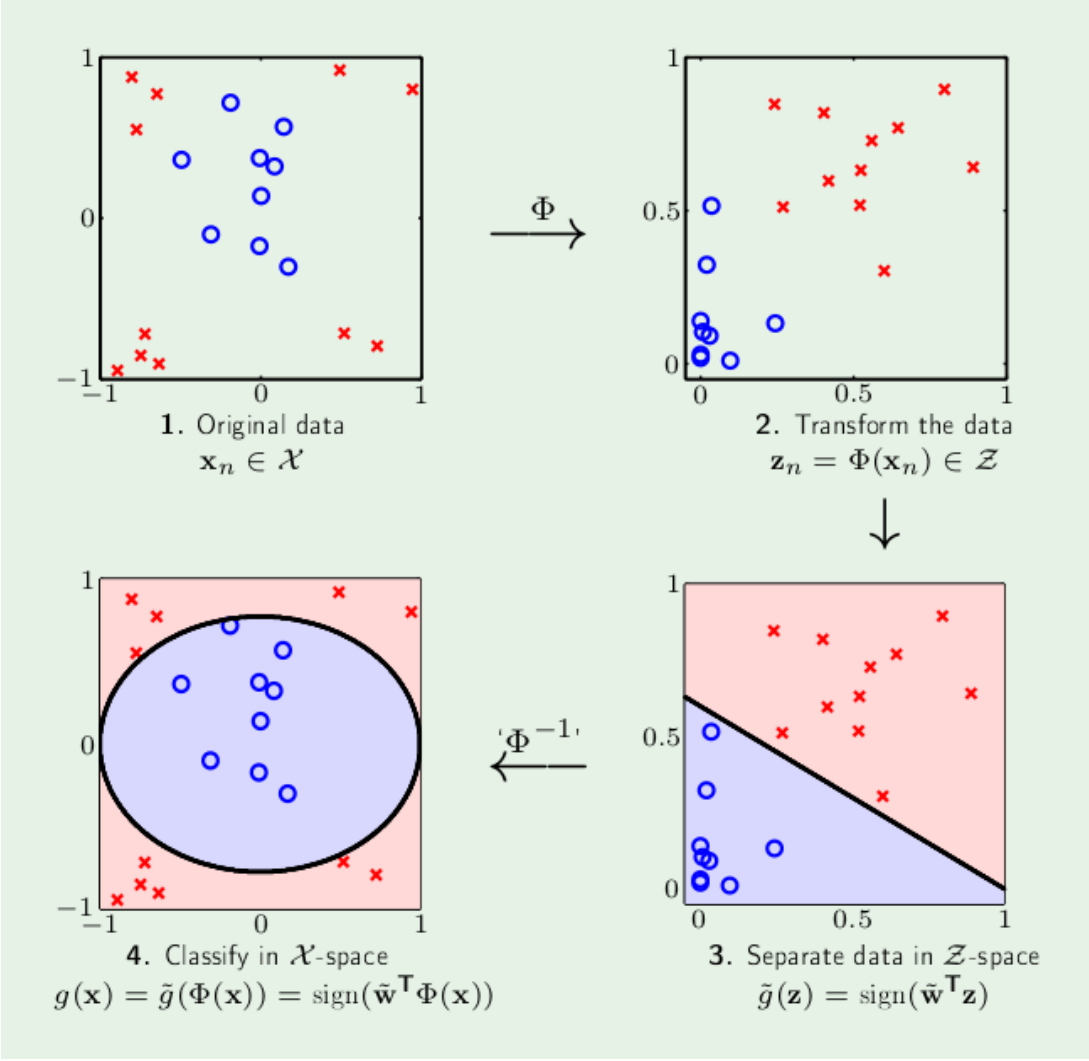
$$g(x_1, x_2) = w_0 + w_1\, x_1^2 + w_2\, x_1\, x_2$$

Claramente g é não-linear com respeito a x_1 e a x_2. Mas ela é linear com respeito a $\mathbf{w}$ (supondo $\mathbf{x}$ constante).

Então, podemos pensar em primeiramente transformar os dados como ilustrado no exemplo a seguir:



Em seguida, podemos aplicar a regressão logística no novo espaço. Quando a superfície de decisão linear neste novo espaço é mapeado de volta para o espaço original, teremos de fato uma superfície não necessariamente linear. O esquema é ilustrado no seguinte exemplo:



Assim ganhamos flexibilidade quanto às superfícies de decisão possíveis, sem alterar o algoritmo. No entanto, o preço que pagamos para termos grande flexibilidade é o aumento da dimensão dos dados. Imagine que você queira considerar a família de polinômios de grau 3. Se supormos que o dado original tem dois componentes (x_1 e x_2), quantos termos precisariam ser calculados (qual seria a dimensão dos dados no espaço transformado) ?

Mais para a frente veremos que o SVM permite explorar essa ideia sem a necessidade de explicitamente transformar os dados.

5.2 Classificação Multiclasses

Estudamos alguns algoritmos para o problema de classificação binária. Podemos utilizá-los para resolver problemas de classificação multi-classes ?

5.2.1 Via combinação de classificadores binários

Intuitivamente, parece natural "quebrarmos" um problema de classificação multi-classes em vários problemas de classificação binária.

De fato, existem alguns métodos bem conhecidos que seguem essa ideia.

Suponha que temos entradas $\mathbf{x}$ que podem pertencer a um dentre K classes possíveis. Assim, o conjunto de rótulos de classe é definido por $Y = \{1, 2, \ldots, K\}$.

Esquema OVA (*One versus All*) Podemos ter um classificador binário, específico para cada classe k, que é projetado para separar instâncias da classe k das demais instâncias (por isso o nome *One versus All*). Neste caso teremos K classificadores h_j, $j = 1, 2, \ldots, K$.

Podemos supor que cada classificador h_j devolve um escore no intervalo $[0, 1]$ (poderia ser a regressão logística).

Precisamos então ter uma regra para, dada uma instância $\mathbf{x}$, calcular a classificação final para ela. Uma regra comumente usada é calcular a classe final como sendo

$$\arg\max_j \{h_j(\mathbf{x})\}$$

(isto é, a classe j do classificador h_j que atribui maioe escore a $\mathbf{x}$, ou em outras palavras, aquela classe tal que o classificador está mais confiante).

Esquema OVO (*One versus One*) Neste caso teremos um classificador para cada par de classes: h_{jk} é um classificador binário treinado usando apenas exemplos das classes j (positivo) e k (negativo).

Teremos, neste caso, $\frac{K(K-1)}{2}$ classificadores binários, h_{jk}, $j < k$, $j, k = 1, 2, \ldots, K$ (note que para $k > j$, temos $h_{kj} = 1 - h_{jk}$).

Vamos supor novamente que cada um desses classificadores devolve um escore em $[0, 1]$.

A regra de decisão para a classificação final, dada uma instância $\mathbf{x}$, pode ser

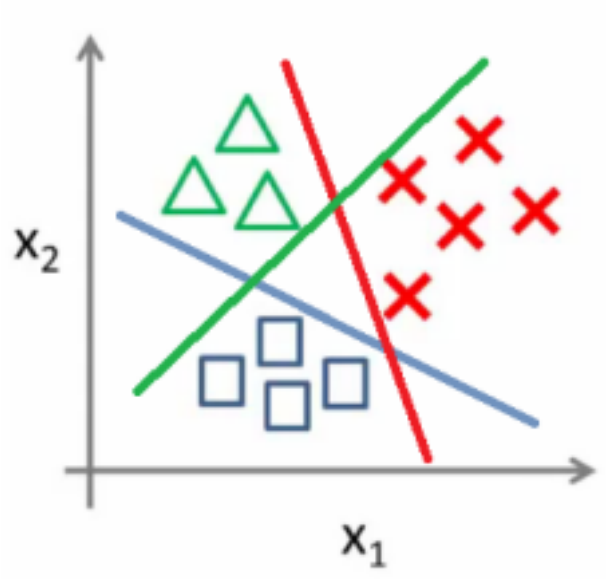
$$\arg\max_j \left\{ \sum_{k=1}^{K} h_{jk}(\mathbf{x}) \right\}$$

Isto é, para cada classe j, calculamos a soma dos escores atribuídos a j pelos $K-1$ classificadores h_{jk}. Em seguida escolhemos a classe j (dentre 1 a k), que recebeu maior escore total.

Em ambos os casos acima, qualquer clasificador binário pode ser utilizado. Se os classificadores binários devolvem um escore (como o classificador logístico), então a decisão pode ser baseada nas regras dadas acima.

No entanto, se os classificadores devolvem a classe da instância $\mathbf{x}$ (e não escores), o que podemos

fazer ? Podemos considerar o número de votos recebidos de cada classe e escolher a classe mais votada. Porém, neste caso podemos ter regiões em $\mathcal{X}$ sem classificação definida. No exemplo a seguir é mostrado uma situação para o caso *One versus All*, com três classe.



Fonte: https://utkuufuk.com/2018/06/03/one-vs-all-classification/

Cada um das retas separa as instâncias de uma determinada cor, das demais. Neste caso, a região triangular no centro receberá apenas votos do tipo "resto" (e portanto não será enm vermelho, sem azul, nem verde). Já os três cones externos (brancos) recebem sempre dois votos, de cores distintas, configurando empate.

Os exemplos acima são apenas os mais comuns citados na literatura. Existem várias outras variantes para se combinar múltiplos classificadores binários de forma a implementar um classificador multi-classes. Algumas referências que podem ser consultadas são *Combining Pattern Classifiers: Methods and Algorithms* de Ludmila I. Kuncheva [3], e *A review on the combination of binary classifiers in multiclass problems*, Ana C. Lorena, André C. P. L. F. de Carvalho, João M. P. Gama [4].

5.2.2 Classificadores inerentemente multiclasses

Qualquer abordagem que estima as K condicionais $P(y = j \,|\, \mathbf{x})$, $j = 1, 2, \ldots, K$ de forma conjunta enquadra-se neste caso.

A generalização da regressão logística para o caso de múltiplas classe é chamada de **regressão**

logística multinomial. Para tanto, utiliza-se a função `softmax`:

$$\hat{p}_j = \hat{P}(y = j|\mathbf{x}) = \frac{e^{\mathbf{w}_j^T \tilde{\mathbf{x}}}}{\sum_{i=1}^{K} e^{\mathbf{w}_i^T \tilde{\mathbf{x}}}}, \quad j = 1, 2, \ldots, K$$

Por exemplo, para $K = 3$ classes, temos:

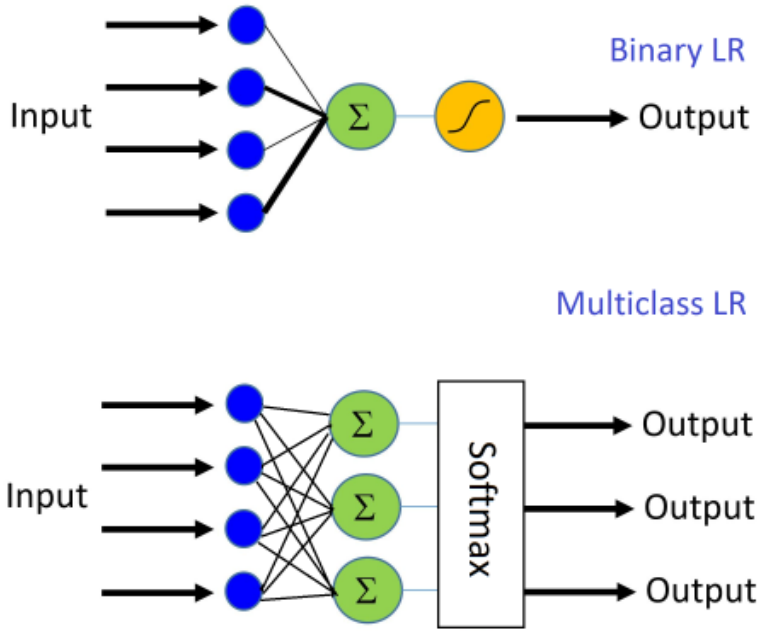
$$\hat{p}_1 = \hat{P}(y = 1|\mathbf{x}) = \frac{e^{\mathbf{w}_1^T \tilde{\mathbf{x}}}}{e^{\mathbf{w}_1^T \tilde{\mathbf{x}}} + e^{\mathbf{w}_2^T \tilde{\mathbf{x}}} + e^{\mathbf{w}_3^T \tilde{\mathbf{x}}}}$$

$$\hat{p}_2 = \hat{P}(y = 2|\mathbf{x}) = \frac{e^{\mathbf{w}_2^T \tilde{\mathbf{x}}}}{e^{\mathbf{w}_1^T \tilde{\mathbf{x}}} + e^{\mathbf{w}_2^T \tilde{\mathbf{x}}} + e^{\mathbf{w}_3^T \tilde{\mathbf{x}}}}$$

$$\hat{p}_3 = \hat{P}(y = 3|\mathbf{x}) = \frac{e^{\mathbf{w}_3^T \tilde{\mathbf{x}}}}{e^{\mathbf{w}_1^T \tilde{\mathbf{x}}} + e^{\mathbf{w}_2^T \tilde{\mathbf{x}}} + e^{\mathbf{w}_3^T \tilde{\mathbf{x}}}}$$

Claramente $\hat{p}_1 + \hat{p}_2 + \hat{p}_3 = 1$. Alem disso, $0 \leq \hat{p}_j \leq 1$.

A figura abaixo mostra o caso visto anteriormente em classificação binária, em que o regressor logístico prediz $\hat{p}_1 = \hat{P}(y = 1|\mathbf{x})$, e um caso de regressão multinomial com $K = 3$ classes. Enquanto no caso binário, para o cenário abaixo de 4 entradas, é necessário ajustar 4 pesos, no caso de 3 classes é preciso ajustar 12 pesos.



Fonte: https://www.cntk.ai/pythondocs/CNTK_103B_MNIST_LogisticRegression.html

O caso binário pode ser resolvido usando `softmax`. Enquanto anteriormente tratamos ele como

um problema de uma única saída fazendo

$$\hat{p}_1 = \hat{P}(y=1|\mathbf{x}) = \theta(\mathbf{w}^T\mathbf{x}) = \frac{1}{1+e^{-\mathbf{w}^T\tilde{\mathbf{x}}}} \qquad e \qquad \hat{p}_0 = \hat{P}(y=0|\mathbf{x}) = 1 - \hat{P}(y=1|\mathbf{x})$$

podemos também formular como um problema de duas saídas $\hat{p}_0$ e $\hat{p}_1$.

Vejamos como podemos partir da equação de $\hat{p}_1$ acima e chegar na formulação `softmax`. Primeiramente, podemos reescrever $\hat{p}_1$ como:

$$\hat{p}_1 = \frac{1}{1+e^{-\mathbf{w}^T\tilde{\mathbf{x}}}} = \frac{e^{\mathbf{w}^T\tilde{\mathbf{x}}}}{e^{\mathbf{w}^T\tilde{\mathbf{x}}}} \frac{1}{(1+e^{-\mathbf{w}^T\tilde{\mathbf{x}}})} = \frac{e^{\mathbf{w}^T\tilde{\mathbf{x}}}}{e^{\mathbf{w}^T\tilde{\mathbf{x}}}+1}$$

Então, já que $\hat{p}_0 = 1 - \hat{p}_1$, temos

$$\hat{p}_0 \quad = \quad 1 - \frac{e^{\mathbf{w}^T\tilde{\mathbf{x}}}}{e^{\mathbf{w}^T\tilde{\mathbf{x}}}+1} = \frac{1}{1+e^{\mathbf{w}^T\tilde{\mathbf{x}}}}$$

É fácil verificar que:

$$\hat{p}_1 + \hat{p}_0 = \hat{P}(y=1|\mathbf{x}) + \hat{P}(y=0|\mathbf{x}) = 1$$

Agora façamos como segue:

$$\hat{P}(y=0|\mathbf{x}) = \frac{1}{(1+e^{\mathbf{w}^T\tilde{\mathbf{x}}})} \frac{e^{\mathbf{w}_0^T\tilde{\mathbf{x}}}}{e^{\mathbf{w}_0^T\tilde{\mathbf{x}}}} = \frac{e^{\mathbf{w}_0^T\tilde{\mathbf{x}}}}{e^{\mathbf{w}_0^T\tilde{\mathbf{x}}} + e^{(\mathbf{w}+\mathbf{w}_0)^T\tilde{\mathbf{x}}}}$$

$$\hat{P}(y=1|\mathbf{x}) = \frac{e^{\mathbf{w}^T\tilde{\mathbf{x}}}}{(1+e^{\mathbf{w}^T\tilde{\mathbf{x}}})} \frac{e^{\mathbf{w}_0^T\tilde{\mathbf{x}}}}{e^{\mathbf{w}_0^T\tilde{\mathbf{x}}}} = \frac{e^{(\mathbf{w}+\mathbf{w}_0)^T\tilde{\mathbf{x}}}}{e^{\mathbf{w}_0^T\tilde{\mathbf{x}}} + e^{(\mathbf{w}+\mathbf{w}_0)^T\tilde{\mathbf{x}}}}$$

Temos a formulação `softmax` na qual $(\mathbf{w} + \mathbf{w}_0)$ corresponde ao $\mathbf{w}_1$.

Otimização: Para múltiplas saídas, é preciso considerar o chamado *One-hot encoding of the output*. Isto é, para cada entrada $\mathbf{x}_i$, a saída (alvo) será agora um vetor de dimensão K, definido como $\mathbf{y}_i = (y_{i1}, y_{i2}, \ldots, y_{iK})$ tal que $y_{ij} = 1 \iff \mathbf{x}_i$ é da classe j, $j = 1, 2, \ldots, K$.

A função de custo a ser otimizada é a **Cross-entropy loss**:

$$\mathcal{L}(\mathbf{w}) = -\sum_{i=1}^{N}\sum_{j=1}^{K} y_{ij} \log \hat{p}_{ij}$$

Relembrando, $\hat{p}_{ij} = \hat{P}(y_i = j \,|\, \mathbf{x}_i)$, $\sum_{j=1}^{K} \hat{p}_{ij} = 1$, e os parâmetros a serem otimizados, $\mathbf{w}_j$, são

aqueles na função `softmax`

$$\hat{p}_{ij} = \frac{e^{\mathbf{w}_j^T \tilde{\mathbf{x}}_i}}{\sum_{k=1}^{K} e^{\mathbf{w}_k^T \tilde{\mathbf{x}}_i}}$$

.

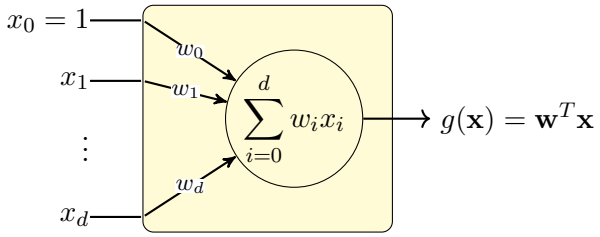
6 Redes neurais

Para falar sobre redes neurais, convém voltar ao perceptron (seção 3.1). Vimos que o algoritmo perceptron converge somente se as classes são linearmente separáveis. Em particular é bem conhecido o fato de que perceptrons não são capazes de representar a função lógica XOR. No entanto, sabe-se que combinando vários perceptrons é possível expressar não apenas a função XOR mas diversas outras.

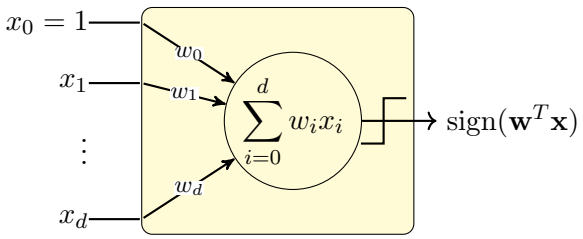
A seguir, iremos inicialmente entender a expressividade de redes perceptron de múltiplas camadas (MLP). Posteriormente iremos ver como redes neurais são treinadas.

6.1 MLP e sua expressividade

Primeiramente vejamos um diagrama de uma função linear:



Passando a função linear pela função sinal ou degrau, temos o perceptron, que pode ser esquematizado como segue:



Na literatura da área, encontramos vários relatos sobre perceptron terem sido inspirados em neurônios biológicos. De fato há alguma semelhança estrutural (ver figura 7): sinais de entrada são agregados e se, após agregados, passarem de um certo limiar, um sinal é transmitido para

a saída e em seguida propagado para outros neurônios. Mas, de forma geral, a similaridade termina aí.

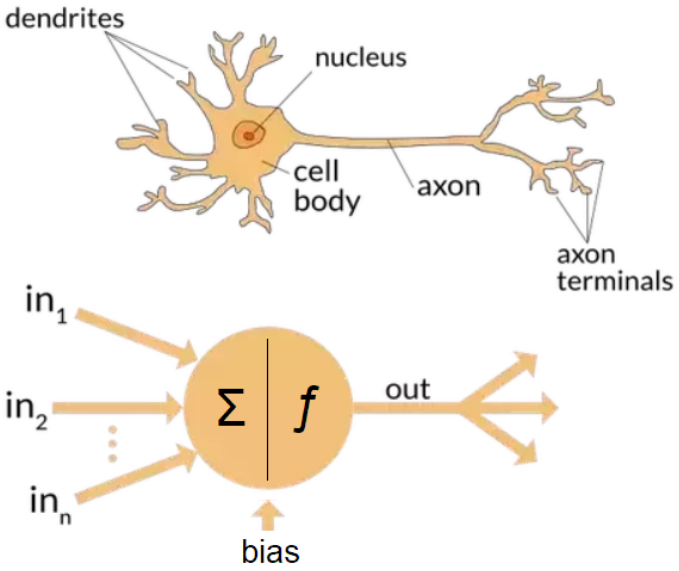


Figura 7: Semelhança estrutural entre o neurônio biológico e artificial.

No contexto de rede neural artificial, um neurônio é uma unidade como a mostrada acima. Tipicamente os sinais de entrada são linearmente combinados e em seguida passam por uma função de ativação:

$$\text{output} = \phi(g(\mathbf{x})) = \phi(\mathbf{w}^T\mathbf{x})$$

Dois exemplos de função de ativação que geram saída binária são:

Função sinal: $\phi(z) = \begin{cases} +1, & \text{se } z > 0, \\ -1 & \text{se } z \leq 0. \end{cases}$

Função degrau: $\phi(z) = \begin{cases} 1, & \text{se } z > 0, \\ 0 & \text{se } z \leq 0. \end{cases}$

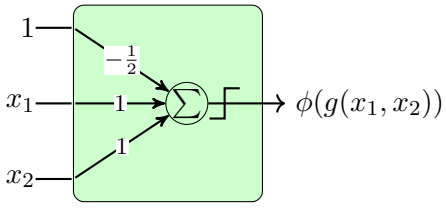
Além disso, temos também a função sigmóide, e mais outras, que geram valores reais (geralmente limitados a intervalos como $[0, 1]$ ou $[-1, 1]$).

6.1.1 Implementação de funções lógicas

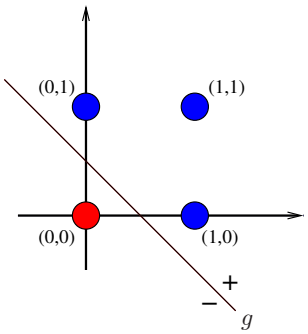
Vamos examinar como as funções lógicas OR, AND e XOR podem ser implementadas usando uma rede de neurônios, com ativação degrau (ou seja, perceptrons).

Função OR: Abaixo a tabela que define a função, um neurônio que a implementa e uma representação diagramática.

x_1	x_2	$\phi(g(x_1, x_2))$
0	0	0
0	1	1
1	0	1
1	1	1

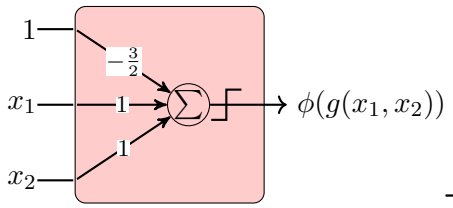


$$g(x_1, x_2) = x_1 + x_2 - \tfrac{1}{2}$$

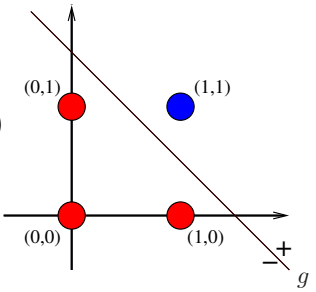


Função AND: Abaixo a tabela que define a função, um neurônio que a implementa e uma representação diagramática.

x_1	x_2	$\phi(g(x_1, x_2))$
0	0	0
0	1	0
1	0	0
1	1	1

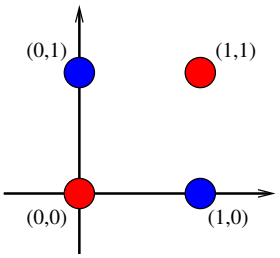


$$g(x_1, x_2) = x_1 + x_2 - \tfrac{3}{2}$$



XOR não é linearmente separável: Abaixo a definição da função.

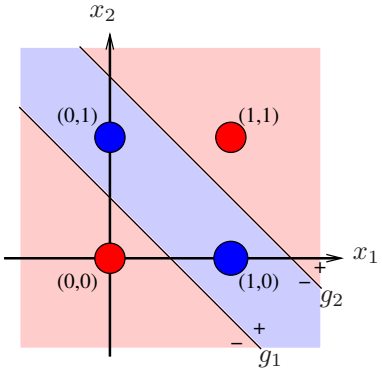
x_1	x_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

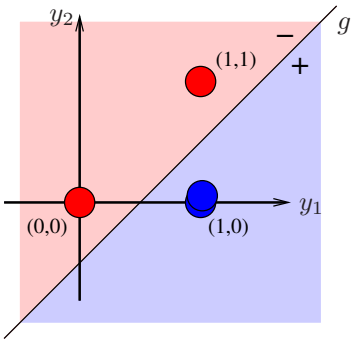


Podemos, porém, implementar a função XOR usando três perceptrons, como mostrado a seguir.

$$g_1(\mathbf{x}) > 0 \quad \text{and} \quad g_2(\mathbf{x}) < 0$$
$$\Downarrow$$
$$y_1 = 1$$

$$g_1(\mathbf{x}) < 0 \quad \text{or} \quad g_2(\mathbf{x}) > 0$$
$$\Downarrow$$
$$y_2 = 0$$



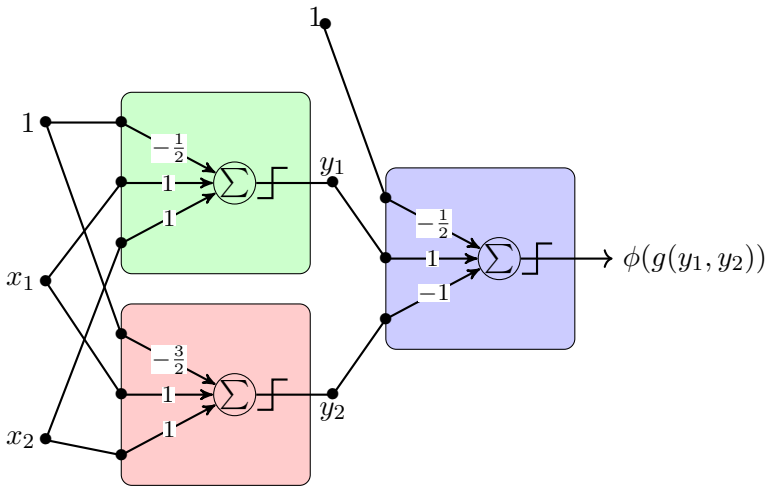


À esquerda está a lógica de combinação, no centro as duas funções a serem combinadas, e à direita o resultado da combinação dessas duas funções. Essas funções e sua combinação estão detalhadas a seguir. Note que um ponto (x_1, x_2) é mapeado para outro ponto (y_1, y_2) em outro

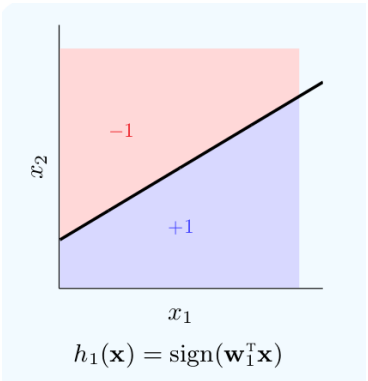
espaço (e carrega consigo a "cor" original).

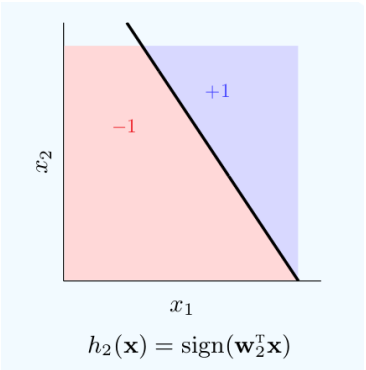
x_1	x_2	g_1	g_2	$y_1 = \phi(g_1)$	$y_2 = \phi(g_2)$	$g(y_1, y_2)$	$\phi(g(y_1, y_2))$
0	0	-	-	0	0	-	0
0	1	+	-	1	0	+	1
1	0	+	-	1	0	+	1
1	1	+	+	1	1	-	0

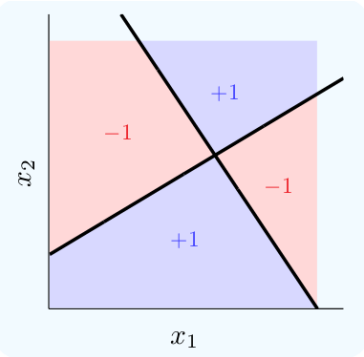
A rede de perceptrons que implementa a função XOR é mostrada a seguir.



Nos slides da Caltech, mostra-se a mesma ideia, porém usando-se a função de ativação sinal. Isto é, na explicação acima consideramos 1=TRUE e 0=FALSE enquanto o prof. Abu-Mostafa considera +1=TRUE e -1=FALSE (como mostrado abaixo). Note ainda que o produto e adição nas expressões abaixo representam as operações lógicas AND e OR, respectivamente.







$$h = h_1\,\overline{h}_2 + \overline{h}_1\,h_2$$

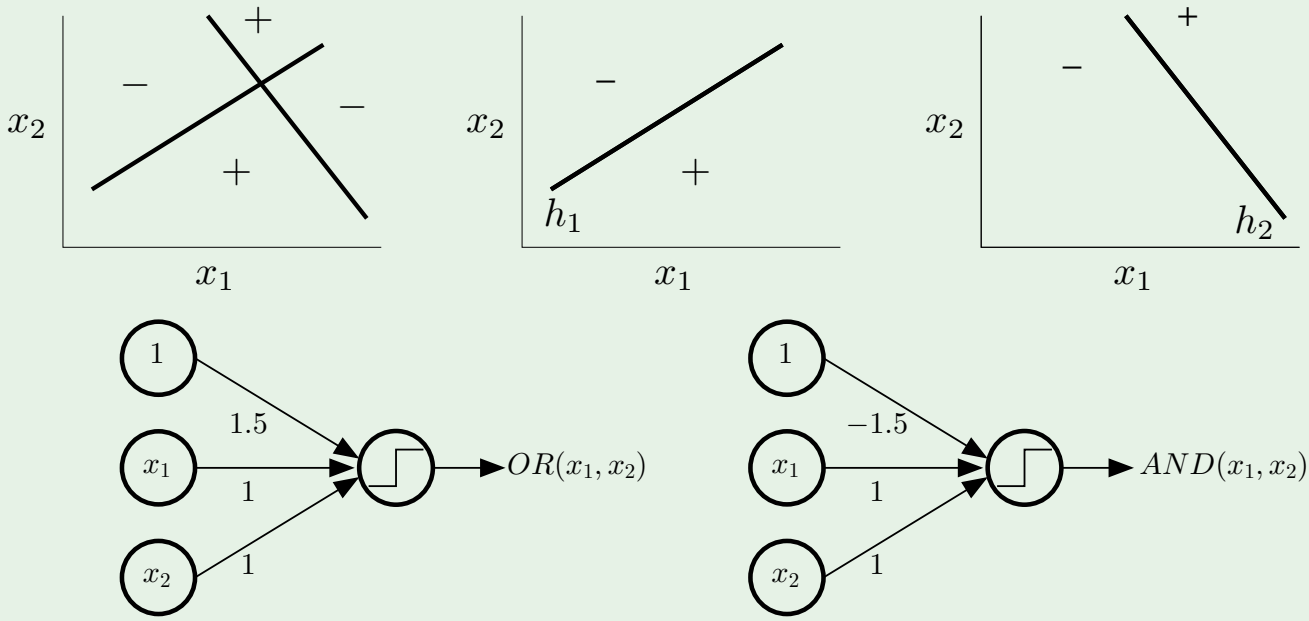
Este é um exemplo em que as duas funções definem 4 regiões (2 positivas e 2 negativas). No exemplo anterior tínhamos 3 regiões (uma positiva e duas negativas). Para implementar esse caso, basta implementarmos a função h, que inclui dois AND e um OR.

Os três seguintes slides da CALTECH mostram:

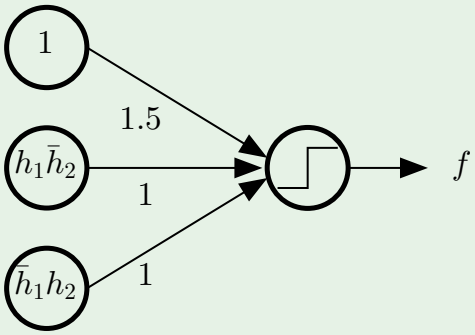
1. neurônios que representam o OR e o AND
2. rede OR com as entradas sendo $h_1\,\overline{h}_2$ e $\overline{h}_1\,h_2$ (esquerda) e a mesma rede com nós AND representando o $h_1\,\overline{h}_2$ e o $\overline{h}_1\,h_2$ (direita)
3. a rede completa, substituindo o h_1 e h_2 da rede anterior por combinações lineares $\mathbf{w}^T\tilde{\mathbf{x}}$.

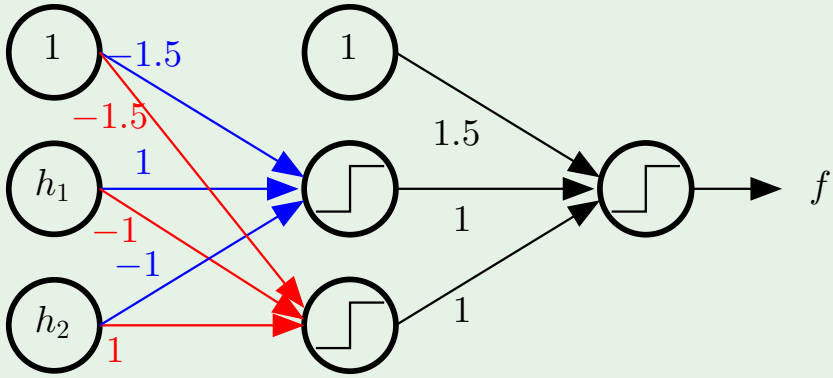
O quarto slide mostra que, ao se ultilizar infinitos perceptrons, podemos reproduzir qualquer segmentação do espaço $\mathbb{R}^d$. O exemplo mostra como ao aumentar o número de reetas, podemos aproximar um círculo.

Combining perceptrons

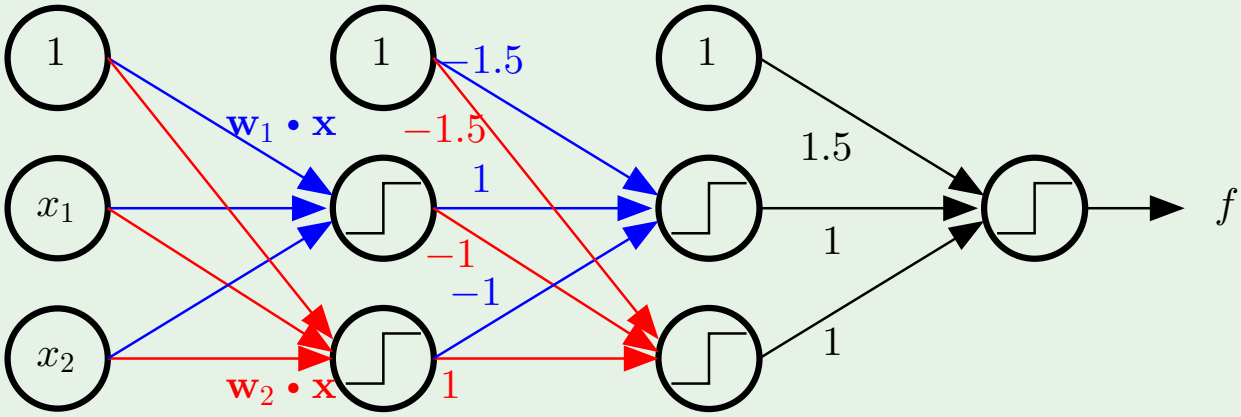


Creating layers



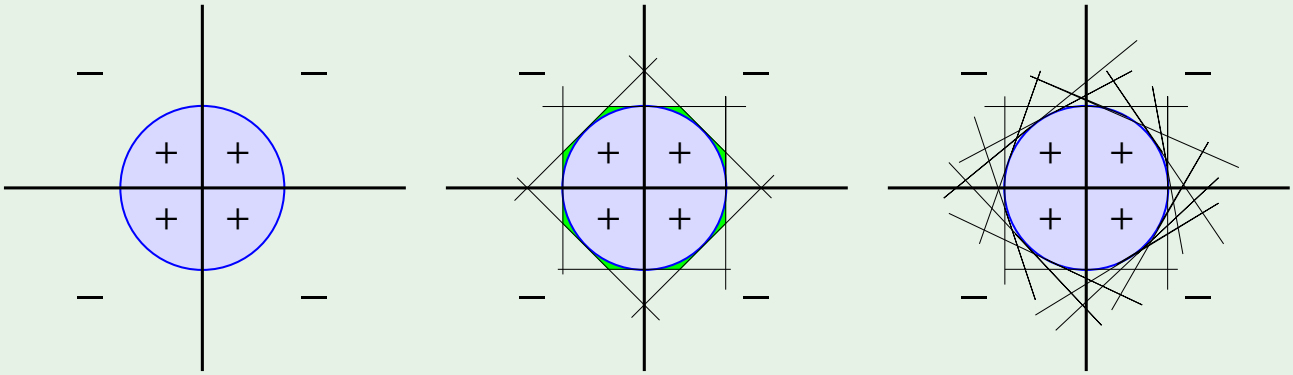


The multilayer perceptron



3 layers "feedforward"

A powerful model



Target | 8 perceptrons | 16 perceptrons

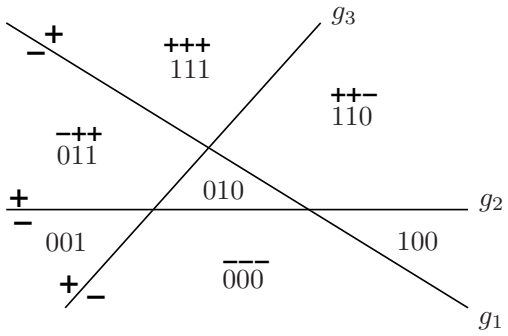
2 red flags for **generalization** and **optimization**

6.1.2 Rede de perceptrons

De fato, qualquer função binária pode ser representada por meio de uma rede MLP com até 3 camadas. Mostramos aqui um explicação por meio de um exemplo.

Suponha que o espaço de instâncias $\mathbf{x}$ é $\mathbb{R}^2$. Suponha que usamos três funções lineares g_1, g_2, g_3. Os pontos nos quais essas funções tomam valor 0, as fronteiras de decisão lineares, correspondem às retas na figura a seguir. O espaço $\mathbb{R}^2$ é segmentado em 7 regiões. Para cada região podemos associar um código binário de tamanho $p = 3$. Por exemplo, 011 significa que os pontos daquela região estão no lado negativo do hiperplano definido por g_1, no lado positivo do hiperplano definido por g_2 (o 1 do centro) e no lado também positivo do hiperplano definido por g_3.

Essa ideia pode ser estendida para $\mathbf{x}$ em $\mathbb{R}^d$ (dados no espaço de dimensçao d) e para p hiperplanos (muito mais que 3).



Em resumo, dados p hiperplanos, cada região criada por esses hiperplanos pode ser mapeada para um vértice do hipercubo unitário de dimensão p (isto é, para um elemento em $\{0,1\}^p$).

- Os p hiperplanos são definidos pelas funções lineares g_i $(i = 1, 2, \ldots, p)$. O perceptron $\phi(g_i(\mathbf{x}))$ determina a qual lado do hiperplano g_i está o ponto $\mathbf{x}$. Esses podem ser considerados como os nós na primeira camada.

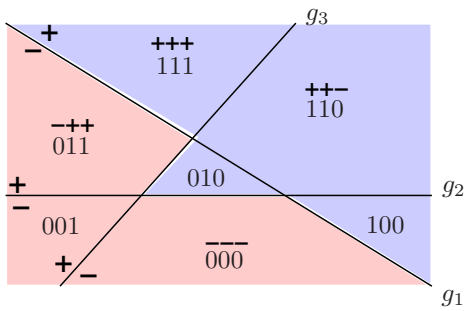
- A saída dos nós da primeira camada formam um vetor $\mathbf{y} \in \{0,1\}^p$

$$\mathbf{y} = \Big(\phi(g_1(\mathbf{x})), \phi(g_2(\mathbf{x})), \ldots, \phi(g_p(\mathbf{x}))\Big) = (y_1, y_2, \ldots, y_p)$$

  Ou seja, $(y_1, y_2, \ldots, y_p)$ é um vértice do hipercubo unitário H_p em $\mathbb{R}^p$

- Assim, podemos pensar que todos os pontos em $\mathbb{R}^d$ em um certa região (dentre as regiões definidas pelos $g_i()$) serão mapeados para um mesmo vértice de H_p

- Supondo que as fronteiras de decisão de interesse são de fato aqueles definidos pelas funções $g_i()$, tudo que precisamos fazer é separar os vértices de H_p que correspondem às regiões positivas daqueles que correspondem às regiões negativas.



- Porém se os vértices positivos e negativos em H_p formarem uma configuração do tipo XOR, um perceptron aplicado sobre H_p não será suficiente para fazer a separação.

- Mas note que podemos facilmente especificar, para cada vértice de H_p, um perceptron que separa o vértice dos demais vértices (é uma função do tipo AND).

- Assim, se queremos apenas selecionar os vértices positivos, e supondo que são m deles, podemos criar os m perceptrons na segunda camada, e então na terceira camada da rede, adicionar um perceptron que realiza o OU das saídas da segunda camada.

A rede descrita acima irá responder 1 caso o ponto $\mathbf{x}$ na entrada da rede seja da classe positiva e irá reponder 0 em caso contrário.

Portanto, com uma rede com 3 camadas de perceptrons, podemos em princípio implementar qualquer superfície de decisão em problemas de classificação binária. Porém, há alguns problemas de ordem prática:

- uma quantidade gigantesca (infinita?) de perceptrons pode ser necessária na primeira camada para aproximar fronteiras de decisão complexas.

- Isso implica um grande número de nós na segunda camada também.

- O pior de tudo é que não há algoritmos para a construção dessas redes. As que vimos acima, foram construídas à mão. Construir à mão só é viável no caso de fronteiras de separação simples.

Podemos também construir redes do tipo acima com um número maior de camadas. As redes desse tipo são conhecidas por *Multilayer Perceptons* (MLP). Note que nas explicações acima usamos funções de ativação do tipo degrau. Com a evolução das redes, passaram a ser utilizadas funções de ativação distintas, tais como a sigmóide ou a tangente hiperbólica, dando origem aos chamados *Feed-forward neural networks*. No entanto, em alguns lugares, essas redes, mesmo com funções de ativações gerais, continuam sendo chamadas de MLP.

6.2 *Feed-forward neural networks*

Uma rede neural padrão do tipo *feed-forward* tem uma estrutura similar às MLPs, com a diferença de seus neurônios utilizarem uma função de ativação contínua e com a saída limitada (geralmente) a um pequeno intervalo. A figura 8 mostra exemplos de funções de ativação (incluir a ReLU no futuro).

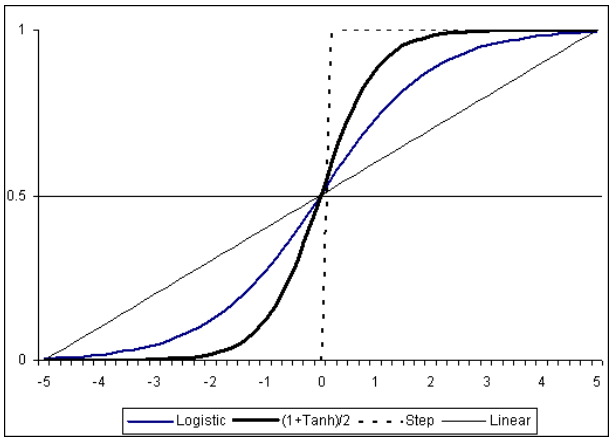


Figura 8: Exemplos de função de ativação comumente usadas em redes neurais.

Vale a pena observar que a computação realizada pelo algoritmo de regressão logística pode ser pensada como um neurônio, conforme mostrado na figura 9. O *forward pass* consiste em alimentar a entrada ($\mathbf{x} = (x_1, \ldots, x_d)$ – na figura m em lugar de d e b no lugar de w_0) e propagar o sinal até a saída ($y = \sigma(\mathbf{w}^T\tilde{\mathbf{x}})$ – na figura, φ em lugar de ϕ).

Uma rede neural do tipo *feed forwrad* pode ser construída, portanto, combinando-se vários nós do tipo logístico (ou com outra função de ativação), como mostrado na figura 10.

Na figura, os nós amarelos correspondem a camada de entrada (os componenets de $\mathbf{x} = (x_1, \ldots, x_d)$). Os nós salmão ao final correspondem à camada de saída; no caso a saída é composta de 3 valores. Em problemas de classificação, tipicamente temos uma ativação `softmax` na camada de saída (veja ativação `softmax` na seção em que discutimos a regressão logística multinomial, usada para problemas de classificação com múltiplas classes). As demais camadas, cada um dos arranjos verticais de nós azuis, correspondem às camadas intermediárias (ou ocultas) da rede.

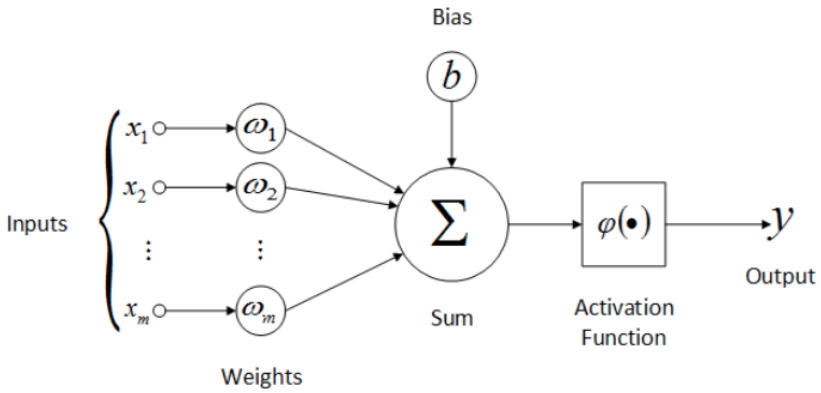


Figura 9: Computação efetuada na regressão logística.

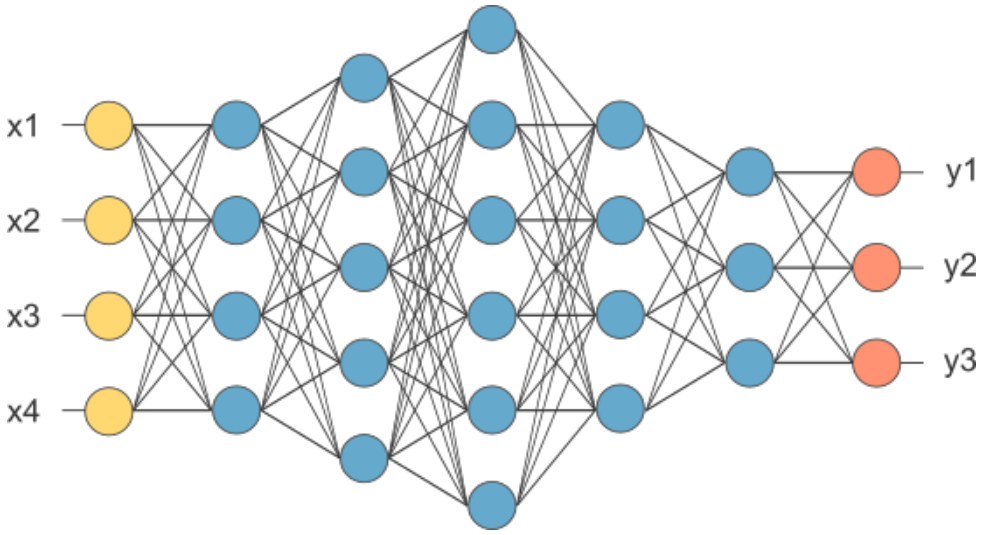


Figura 10: Uma arquitetura de uma rede do tipo *feed-forwrd*.

As redes neurais *feed-forward* padrão são totalmente conectadas (*fully connected*), isto é todos os nós de uma determinada camada l recebem como entrada a saída de todos os nós da camada anterior $l-1$. A cada uma das conexões está associado um peso w. No processamento de um dado de entrada, deve-se realizar o *forward pass*, que pode ser calculado camada a camada, até o sinal atingir a saída. Dependendo do valor dos pesos nas arestas, a resposta pode ser uma ou outra.

6.2.1 Pausa, para amenidades

O desenvolvimento de redes neurais ao longo dos tempos costuma ser contado por alguns em um tom um tanto dramático. Aqui listamos alguns momentos marcantes (não necessariamente precisos):

- Décadas de 1950 e 1960: um período de euforia, com a criação do algoritmo perceptron

- Ano de 1969: Minsky *et al* publicaram "Perceptrons" [5], e uma das coisas que eles fizeram foi mostrar que o perceptron não podia ser usado para problemas não linearmente separáveis. Isso desanimou a comunidade científica e tivemos um período que muitos chamam de **AI winter**.

- Ano de 1988: Rumelhart *et al* publicaram "Learning Representations by Back-Propagating Errors" [6], introduzindo o termo *backpropagation* para descrever a forma como o gradiente é calculado no treinamento de redes neurais.

- Década de 1990: Apesar do algoritmo *backpropagation*, na prática treinar a rede era difícil e não se conseguia produzir resultados interessantes. Assim veio o segundo **AI winter**. Neste período floresceu o algoritmo SVM (*support vector machines*) [7].

- 1998: Apesar da descrença em redes neurais, alguns pesquisadores continuaram trabalhando com redes neurais. Ume exemplo interessante é o trabalho de Yan LeCun, "Gradient-based Learning Applied to Document Recognition" [], publicado em 1998 e que muito influenciou as redes neurais convolucionais modernas.

- **2006**: Geoffrey E. Hinton et al, "A Fast Learning Algorithm for Deep Belief Nets" [] propôs um método que permitiu o treinamento efetivo de redes neurais com um número relativamente grande de camadas. Alguns consideram esse período como o marco inicial da era *Deep Learning*

- **2012**: Foi somente em 2012 que *Deep Learning* popularizou-se. O evento marcante foi o fato de uma solução baseada em rede neural comvolucional (depois chamada de Alexnet []) vencer uma competição para classificação de imagens (no contexto do ImageNet – um grande conjunto de imagens anotadas).

- **2018**: Bengio, Hinton and LeCun foram agraciados com o *Turing Award* (o Nobel da computação). São três pesquisadores que trabalham com redes neurais desde a década de 1980 (ou antes?)

Yoshua Bengio, Geoffrey Hinton and Yann LeCun

LeCun is a mathematical sciences professor at New York University and the vice president and chief AI scientist at Facebook. Hinton is a vice president and engineering fellow at Google. Bengio is a professor at the University of Montreal and the scientific director of both Quebec's Artificial Intelligence Institute and the Institute for Data Valorization.

https://awards.acm.org/about/2018-turing

6.2.2 Treinamento de redes neurais

Seja dado um conjunto de treinamento $D = \{(\mathbf{x}_i, y_i) \in \mathbb{R}^d \times \{0,1\} : i = 1, \ldots, N\}$, para classificação binária. No caso do algoritmo de regressão logística, para o treinamento (ajuste dos valores de $\mathbf{w}$), vimos que devemos minimizar a função de perda *binary cross-entropy*, dada por

$$\mathcal{L}(\mathbf{w}) = -\frac{1}{N}\sum_{n=1}^{N}\Big[y_i \ln \hat{y}_i + (1-y_i)\ln(1-\hat{y}_i)\Big]$$

na qual $\hat{y}_i = \theta(\mathbf{w}^T\tilde{\mathbf{x}})$ (aqui θ denota a função sigmóide; já não me lembro mais se usei essa notação antes – TODO: rever a notação de funções de ativação ...).

Caso o problema seja de classificação multiclasses, então teremos $y_i \in \{1, 2, \ldots, K\}$ em que K é o número de classes, e então devemos minimizar a função *cross-entropy*, dada por

$$\mathcal{L}(\mathbf{w}) = -\sum_{i=1}^{N}\sum_{j=1}^{K} y_{ij}\log \hat{y}_{ij}$$

(lembrar aqui que $\hat{y}_{ij}$ é o valor que a rede produz no nó j da camada de saída quando faz-se o *forward pass* da entrada $\mathbf{x}_i$)

Para a minimização das funções de perda, podemos aplicar o algoritmo de gradiente descendente. Isto significa que precisaríamos calcular o gradiente de $\mathcal{L}$ com respeito a todos os pesos (que na

equação acima estão "escondidos" em $\hat{y}_{ij}$).

No caso da regressão logística, o gradiente é um vetor de dimensão $d+1$ (precisamente o número de pesos a serem ajustados). No caso de redes neurais, a dimensão é exatamente igual ao número de pesos (arestas que conectam nós) na rede e esse número poderá ser muito maior. Mais ainda, a rede pode ser vista como uma composição de várias funções. Portanto, para o cálculo do gradiente precisamos calcular derivadas parciais de funções compostas.

Algoritmo *backpropagation*: O algoritmo *backpropagation* é baseado no fato de que o gradiente pode ser calculado iterativamente, começando pelos pesos mais ao final da rede, e retropropagando esses valores para as camadas anteriores. Isso permite a computação eficiente do gradiente.

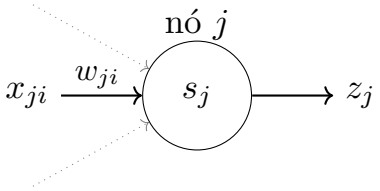
Explicaremos como funciona o algoritmo para a função de perda quadrática, por ela ser mais simples. A mecânica do algoritmo é a mesma para demais funções de perdas. Por simplicidade, também consideraremos o *stochastic gradient descent*, no qual apenas uma instância de entrada é usada para calcular o valor da função de perda $\mathcal{L}$ (e portanto, os pesos da rede são alterados com base na perda referente apenas a essa entrada).

Vamos mudar um pouco a notação, para as expressões matemáticas não ficarem "pesadas". Considere a função de perda erro quadrático (referente a uma única instência de entrada)

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2}\sum_{k=1}^{K}(t_k - z_k)^2$$

em que t_k indica o valor-alvo da componente k e z_k indica o valor produzido pela rede no nó k na camada de saída.

Agora vamos considerar um nó j qualquer na rede. O diagrama a seguir ilustra a estrutura geral:



Associado a ele temos um conjunto de entradas x_{ji} (x_{ji} é a i-ésima entrada do nó j), os respectivos pesos w_{ji} (w_{ji} é o peso associado à i-ésima entrada do nó j), temos a combinação linear s_j calculada pelo nó j ($s_j = \sum_i w_{ji}x_{ji}$) e, finalmente, temos a saída z_j do nó j ($z_j = \phi(s_j)$). Note que seguindo essa notação, temos $x_{ji} = z_i$.

Para calcular a componente do **gradient de** $\mathcal{L}$ referente a w_{ji}, observe que w_{ji} influencia o resto

da rede por meio de s_j. Assim temos:

$$\frac{\partial \mathcal{L}}{\partial w_{ji}} = \frac{\partial \mathcal{L}}{\partial s_j} \frac{\partial s_j}{\partial w_{ji}} \tag{5}$$

Portanto, para calcular $\frac{\partial \mathcal{L}}{\partial w_{ji}}$ precisamos calcular as duas derivadas parciais no lado direito da equação 5. No caso da segunda, como temos que $s_j = \sum_i w_{ji}\, x_{ji}$, então

$$\frac{\partial s_j}{\partial w_{ji}} = \frac{\partial}{\partial w_{ji}} [\sum_i w_{ji}\, x_{ji}] = x_{ji}$$

Já para a primeira, devemos considerar duas possibilidades:

- se j é um nó na camada de saída, da mesma forma que w_{ji} pode influenciar o restante da rede por meio de s_j, s_j pode influenciar o restante da rede apenas por meio de z_j (já que $z_j = \phi(s_j)$). Então podemos escrever

$$\frac{\partial \mathcal{L}}{\partial s_j} = \frac{\partial \mathcal{L}}{\partial z_j} \frac{\partial z_j}{\partial s_j} \tag{6}$$

- se j é um nó em uma camada oculta, então s_j afeta $\mathcal{L}$ por meio de todos os nós na camada seguinte. Então temos que:

$$\frac{\partial \mathcal{L}}{\partial s_j} = \sum_k \frac{\partial \mathcal{L}}{\partial s_k} \frac{\partial s_k}{\partial s_j} \tag{7}$$

Pesos associados à camada de saída: Vamos desenvolver a equação 6. Primeiro, o segundo fator:

$$\frac{\partial z_j}{\partial s_j} = \frac{\partial \phi(s_j)}{\partial s_j} = \phi'(s_j)$$

Aqui ϕ é a função de ativação e, para não especificar uma, simplesmente deixamos a notação $\phi'(s_j)$.

Agora o primeiro fator da equação 6:

$$\frac{\partial \mathcal{L}}{\partial z_j} = \frac{\partial}{\partial z_j} \Big[\frac{1}{2} \sum_{k=1}^{K} (t_k - z_k)^2\Big] = \frac{1}{2} 2(t_j - z_j) \frac{\partial (t_j - z_j)}{\partial z_j} = -(t_j - z_j)$$

Só resta a diferença referente ao nó de saída $k = j$.

Juntando tudo, com respeito a um peso w_{ji} da camada de saída, a Eq. 5 fica:

$$\frac{\partial \mathcal{L}}{\partial w_{ji}} = \frac{\partial \mathcal{L}}{\partial s_j}\frac{\partial s_j}{\partial w_{ji}} = \frac{\partial \mathcal{L}}{\partial s_j}\, x_{ji} = -\underbrace{(t_j - z_j)\phi'(s_j)}_{\delta_j}\, x_{ji} \tag{8}$$

Em destaque, o valor δ_j. Este será um valor a ser retropropagado, como veremos adiante.

Pesos associados à camada oculta: Vamos desenvolver a equação 7

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial s_j} &= \sum_k \frac{\partial \mathcal{L}}{\partial s_k}\frac{\partial s_k}{\partial s_j} \quad \text{(essa é a equação 7)} \\
&= \sum_k -\delta_k \frac{\partial s_k}{\partial s_j} \quad \text{(o } \delta_k \text{ vem da eq. 8)} \\
&= \sum_k -\delta_k \frac{\partial s_k}{\partial z_j}\frac{\partial z_j}{\partial s_j} \quad (s_j \text{ afeta } s_k \text{ por meio de } z_j) \\
&= \sum_k -\delta_k w_{kj} \frac{\partial z_j}{\partial s_j} \quad (\tfrac{\partial s_k}{\partial z_j} = w_{kj} \text{ pois } z_j = x_{kj} \text{ e } s_k = \textstyle\sum w_{kj}x_{kj}) \\
&= \sum_k -\delta_k w_{kj}\phi'(s_j) \quad \text{(como } z_j = \phi(s_j)\text{, temos o } \phi'(s_j))
\end{aligned}$$

Juntando tudo, com respeito a uma peso w_{ji} de uma camada oculta, a Eq. 5 fica:

$$\frac{\partial \mathcal{L}}{\partial w_{ji}} = \frac{\partial \mathcal{L}}{\partial s_j}\, x_{ji} = -\underbrace{\left[\sum w_{kj}\delta_k\right]\phi'(s_j)}_{\delta_j}\, x_{ji} \tag{9}$$

na qual k correspondem aos índices dos nós na camada seguinte ao qual o nó j se encontra.

Compare as equações 8 e 9. Note que eles possuem uma esrutura similar. No caso da equação 8, como j é um nó na camada de saída, um termo do gradiente é $(t_j - z_j)$, justamente a discrepância entre a saída correta e t_j e a saída produzida pela rede. Já na equação 8, existe um somatório referente a todos os nós k da camada seguinte em que podemos pensar que δ_k é o erro retropropagado por via do nó k, e que será usado para calcular o erro a ser retropropagado para as camadas anteriores.

6.2.3 Resumindo ...

O algoritmo gradiente descentende requer o ajuste de pesos:

$$\mathbf{w}(r+1) = \mathbf{w}(r) + \Delta\mathbf{w}(r)$$

$$\Delta\mathbf{w}(r) = -\eta\nabla\mathcal{L}(\mathbf{w})$$

O algoritmo *backpropagation* consiste em calcular os componentes do gradiente, propagando os erros de trás para a frente, conforme vimos acima. Ou seja, se j é um nó na camada de saída, o ajuste a ser feito no peso w_{ji} associado à aresta que liga o nó i da camada anterior e o nó j, é dado por

$$\Delta w_{ji} = \eta \underbrace{\left(t_j - z_j\right) \phi'(s_j)}_{\delta_j} x_{ji}$$

e se j é um nó nas demais camadas, o ajuste é dado por

$$\Delta w_{ji} = \eta \underbrace{\left[\sum w_{kj}\delta_k\right] \phi'(s_j)}_{\delta_j} x_{ji}$$

Se consideramos a função sigmóide como a função de ativação, então temos na camada de saída

$$\delta_j = z_j(1 - z_j)(t_j - z_j)$$

e nas demais camadas

$$\delta_j = z_j(1 - z_j) \sum w_{kj}\delta_k$$

Note que o Prof. Abu-Mostafa utiliza a função de ativação tangente hiperbólica em suas explicações.

6.2.4 Informações e referências adicionais sobre redes neurais

Um resultado teórico frequentemente usado para fundamentar a flexibilidade de redes neurais quanto à capacidade de representarem uma grande família de funções é o *Universal approximation theorem*. Cybenko, G. (1989) mostrou que qualquer função contínua, sob certas restrições não muito fortes, pode ser aproximada por uma superposição de funções sigmóide [8].

Há algumas explicações em http://neuron.eng.wayne.edu/tarek/MITbook/chap2/2_3.html. De acordo com a Wikipedia, *In the mathematical theory of artificial neural networks, the universal approximation theorem states that a feed-forward network with a single hidden layer containing a finite number of neurons can approximate continuous functions on compact subsets of* $\mathbb{R}^n$*, under mild assumptions on the activation function.*

Let $\varphi : \mathbb{R} \to \mathbb{R}$ be a nonconstant, bounded, and continuous function. Let I_m denote the m-dimensional unit hypercube $[0,1]^m$. The space of real-valued continuous functions on I_m is denoted by $C(I_m)$. Then, given any $\varepsilon > 0$ and any function $f \in C(I_m)$, there exist an integer N, real constants $v_i, b_i \in \mathbb{R}$ and real vectors $w_i \in \mathbb{R}^m$ for $i = 1, \ldots, N$, such that we may define:

$$F(x) = \sum_{i=1}^{N} v_i \varphi\left(w_i^T x + b_i\right)$$

as an approximate realization of the function f; that is,

$$|F(x) - f(x)| < \varepsilon$$

for all $x \in I_m$. In other words, functions of the form $F(x)$ are dense in $C(I_m)$.

Há várias bibliografias para o estudo de redes neurais em geral:

- **Neural Networks and Deep Learning**, Michael Nielsen, http://neuralnetworksanddeeplearning.com/
- Capítulo 4 do livro de Tom Mitchell, Machine Learning, 1997
-

Mais recentemente, redes neurais ganharam vida nova com a popularização do chamado *deep learning* [9]. As redes neurais profundas (com várias camadas) têm apresentando desempenho antes não alcançado por outros algoritmos, principalmente no processamento de dados complexos como imagens, vídeos, texto, linguagem natural, fala, entre outros. Em aplicações que envolvem o processamento desse tipo de dados, antes tipicamente era necessário fazer um pré-processamento para a extração de features. Extração de features, na área de machine learning, refere-se ao processo de encontrar uma representação mais compacta e no formato vetorial (um certo número de *features* ou variáveis). Por exemplo, no caso de imagens é bastante comum a extração de informações relativas à geometria dos objetos (isto requer, por exemplo, detectar a borda dos objetos), à cor e textura, entre outras características. Com *deep learning*, esse pré-processamento para a extração de features ficou de lado, pois aparentemente a própria rede realiza essa etapa. Em outras palavras, as ativações na saída de uma determinada camada da rede neural podem ser vistas como *features* extraídas da imagem de entrada.

Por outro lado, treinar redes neurais é uma tarefa não simples. Além de recursos computacionais (memória, capacidade de processamento, disco), requer em geral uma grande quantidade de dados de treinamento. Há vários hiperparâmetros que precisam ser configurados. Alguns deles são:

- network architecture
- activation function
- cost function

- data normalization
- regularization
- Batch training × sthocastic training × mini-batch training
- stopping criteria
- learning rate, momentum
- optimization algorithm
- etc

Do ponto de vista prático, a disponibilização pública de bibliotecas tais com o TensorFlow (https://www.tensorflow.org/), Keras (https://keras.io/), PyTorch (https://pytorch.org/), entre outros ajudou a popularização de *deep learning*.

Uma característica interessante dessas bibliotecas é que elas são capazes de automatizar o cálculo do gradiente, isto é, em geral não é preciso reprogramar a parte referente ao cálculo das derivadas parciais, mesmo quando alteramos a função de perda ou de ativação. O conceito por trás disso são os *Computation graphs*. No caso do Pytorch, um lugar que fala um pouco sobre isso é https://blog.paperspace.com/pytorch-101-understanding-graphs-and-automatic-differentiation/.

No caso do Keras, pode-se experimentar deep-learning-with-python-notebooks disponível em https://github.com/fchollet/deep-learning-with-python-notebooks.

O scikit-learn tem também a implementação de redes *feed-forward* em `sklearn.neural_network.MLPClassifier`, https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html

7 Is learning feasible ?

Esta é uma parte do curso que aborda como podemos formalizar o problema de aprendizado. Do ponto de vista teórico, o que podemos dizer sobre o treinamento de algoritmos ? É razoável esperar que as hipóteses geradas após o treinamento funcionem bem ?

A formalização gira em torno de desenvolver ferramental teórico que nos permita dizer o que o $E_{in}(h)$ (*In-sample error*, ou erro no conjunto de treinamento) pode revelar sobre o $E_{out}(h)$ (erro com respeito ao domínio de todas as instâncias possíveis). O $E_{out}(h)$ é precisamente o desempenho esperado de uma hipótese h se a mesma pudesse ser usada para fazer predições sobre todas as instâncias $\mathbf{x}$ possíveis. Claramente esse valor existe, mas ele é desconhecido (não é possível calculá-lo).

VC analysis é uma formulação teórica que mostra a expressão de E_{out} em termos de E_{in}. Especificamente, mostra-se que com alta probabilidade $(1 - \delta)$, podemos dizer que

$$E_{out} \leq E_{in} + \Omega(N, H, \delta)$$

Neste caso, $\Omega(N, H, \delta)$ pode ser pensada como uma precisão (ϵ). O que acontece com $\Omega(N, H, \delta)$ é que ele aumenta à medida que a dimensão VC de H e/ou $1 - \delta$ aumenta. Para compensar esse aumento, e manter ϵ pequeno, torna-se necessário aumentar N (a quantidade de instâncias de treinamento).

Uma segunda formulação é Bias-Variance analysis, que expressa E_{out} da seguinte forma:

$$E_{out} = Bias + Variance$$

Bias captura a "distância" entre a função target e uma hipótese média, enquanto a Variance capture o quanto a hipótese escolhida pelo algoritmo desvia em relação à hipótese média.

Tanto no caso VC analysis como no caso Bias-Variance analysis, temos uma expressão de E_{out} como soma de dois termos, na qual um termo pode ser associado à capacidade de *fitting* de H (E_{in} ou Bias) – isto é, o quão bem as hipóteses de H conseguem representar o target – e o outro termo à capacidade de aproximação ($\Omega(N, H, \delta)$ ou Variance) – isto é, o quão bem o algoritmo é capaz de escolher uma hipótese de H que seja próximo do target. Ambas as formulações mostram que capacidade de fitting e aproximação comportam-se de forma oposta: ao escolher um H bem expressivo melhoramos *fitting*, mas pioramos a aproximação, e vice-versa.

Essas formulações estão nas Lectures 2, 5, 6, 7 e 8 da Caltech, ou na seção 1.3 + capítulo 2 do livro de Abu-Mostafa *et al.*

8 Overfitting, Regularização e Validação

Em poucas palavras, *overfitting* pode ser explicado como a situação na qual E_{in} é pequeno, porém quando testamos a hipótese em dados novos, o erro é significativamente maior. O teste sobre dados novos dá uma estimativa do E_{out}. Assim, *overfitting* é aquela situação na qual $E_{in} <<< E_{out}$.

Uma vez detectado o *overfitting*, podemos tentar reduzi-lo. Duas abordagens são comumente empregadas: Regularização e Validação

Validation versus regularization

In one form or another, $E_{\text{out}}(h) = E_{\text{in}}(h) +$ overfit penalty

Regularization:

$$E_{\text{out}}(h) = E_{\text{in}}(h) + \underbrace{\text{overfit penalty}}_{\text{regularization estimates this quantity}}$$

Validation:

$$\underbrace{E_{\text{out}}(h)}_{\text{validation estimates this quantity}} = E_{\text{in}}(h) + \text{overfit penalty}$$

Overfitting é discutida na Lecture 11

Regularização na Lecture 12. De forma simples, a forma mais conhecida consiste em adicionar um termo na função de perda de tal forma que hipóteses com certas características são penalizadas (adicionam mais custo à função de perda). Uma vez que o processo de otimização durante o treinamento visa minimizar o valor da função de perda, durante o treinamento os parâmetros da hipótese são ajustados de forma a evitar soluções "caras". Por exemplo, a regularização L_2 adiciona um termo que é a soma do quadrado dos parâmetros da hipótese (no caso de regressão ou redes neurais, os pesos). Desta forma, tendem a ser escolhidas hipóteses cuja magnitude dos pesos são menores – de certa forma, essas podem ser associadas a hipóteses mais suaves e mais estáveis (pequenas perturbações na entrada causarão apenas pequenas alterações na saída).

Validação na Lecture 13. Neste caso, a técnica consiste em se utilizar um conjunto de instâncias (conjunto de validação) distintas daquelas usadas no treinamento (conjunto de treinamento) para obter uma estimativa do E_{out}. Pelo fato do erro de validação E_{val} ser uma estimativa de E_{out}, obviamente está associada a erros de estimação. Na literatura há várias formas para se calcular o erro de validação. Uma família bem conhecida é a validação cruzada que essencialmente consiste em se calcular o erro de validação média sobre diferentes subconjuntos de treinamento/validação. Naturalmente há variantes com respeito a como esses subconjuntos treinamento/validação são criados a partir de um conjunto D. Algumas referências sobre validação: [10, 11, 12].

9 Seleção e avaliação de modelos

O erro de validação serve, obviamente, para se ter uma estimativa do E_{out}. Mas, ele é mais geralmente utilizado para se fazer seleção de modelo. Isto é, suponha que T hipóteses (modelos) foram geradas. Como escolher um modelo ? Uma possibilidade é fazer testes de hipóteses para testar se um modelo é melhor que o outro (estatisticamente falando).

Da mesma forma que existem diferentes formas de se estimar E_{out} (calcular E_{val}), é possível que existam diferentes testes de hipóteses, nem sempre adequadas para todas as situações [10, 11, 13, 12].

- Matriz de confusão

- Curva ROC

- AUC-ROC

- Curva PR

10 SVM

Há um slide bem interessante relacionando hinge loss com SVM em https://www.cs.ubc.ca/ schmidtm/Courses/340-F17/L21.pdf

Referências

[1] Yaser S. Abu-Mostafa, Hsuan-Tien Lin, and Malik Magdon-Ismail. *Learning From Data*. AMLBook, 2012. 3

[2] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*. John Wiley and Sons, 2001. 17

[3] Ludmila I. Kuncheva. *Combining Pattern Classifiers: Methods and Algorithms*. Wiley, 2004. 30

[4] Ana Carolina Lorena, André C. Carvalho, and João M. Gama. A review on the combination of binary classifiers in multiclass problems. *Artif. Intell. Rev.*, 30(1-4):19–37, dec 2008. 30

[5] Marvin Minsky and Seymour A. Papert. *Perceptrons: An Introduction to Computational Geometry*. The MIT Press, 2017. Reissue of the 1988 Expanded Edition with a new foreword by Lon Bottou. 46

[6] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. *Learning Representations by Back-Propagating Errors*, pages 696–699. MIT Press, Cambridge, MA, USA, 1988. 46

[7] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995. 46

[8] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)*, 2(4):303–314, 1989. 51

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. 52

[10] Thomas G. Dietterich. Approximate statistical tests for comparing supervised classification learning algorithms. *Neural Comput.*, 10(7):1895â1923, oct 1998. 56

[11] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference and prediction*. Springer, 2 edition, 2009. 56

[12] T. M. Mitchell. *Machine Learning*. McGraw-Hill Series in Computer Science. McGraw-Hill, March 1997. 56

[13] Steven L. Salzberg. On comparing classifiers: Pitfalls to avoid and a recommended approach. *Data Mining and Knowledge Discovery*, 1(3):317–328, 1997. 56