P2 - Statistical Learning

Mateus Marques

12 de dezembro de 2022

1 Neural Networks

1.1 Perceptron

Perceptron são funções simples assim:

$$Y = f\left(w_0 + \sum_{i=1}^{n} w_i X_i\right),$$

onde $f(X) = 1$ se $X \geq 0$, e -1 caso contrário.

Só que os perceptrons só conseguem resolver problemas linearmente separáveis (tipo SVM).

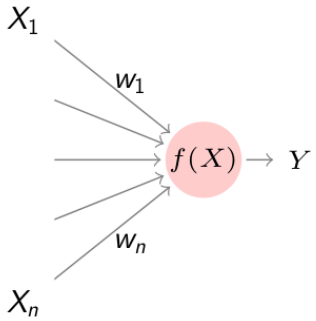


Figura 1: Perceptron.

1.2 Multi-Layer Perceptron (MLP)

Com novas camadas temos mais liberdade:

$$Y = \beta_0 + \sum_{k=1}^{K} \beta_k \, g\left(w_{k0} + \sum_{j=1}^{p} w_{kj} X_k\right),$$

onde p é o número de variáveis na camada de input e K o número de variáveis na camada oculta.

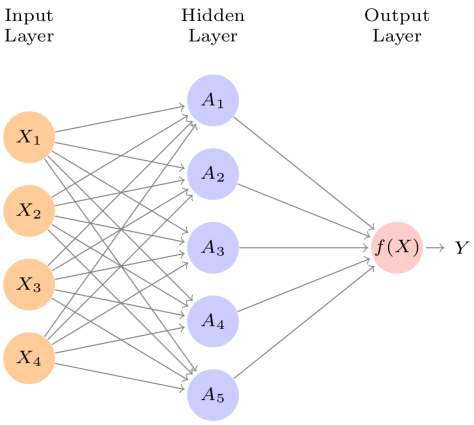


Figura 2: Multi-Layer Perceptron (MLP).

Com camadas oculta, a função de ativação $g(\cdot)$ deve ser não-linear, do contrário teremos uma estrutura linear. Geralmente escolhemos $g(z)$ assim (`ReLU` é mais comum nos dias de hoje):

- Classic: sigmoid.
- Modern: rectified linear (ReLU).

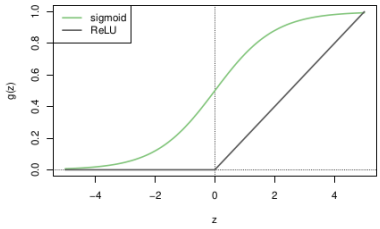
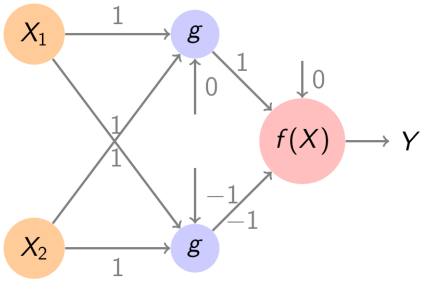


Figura 3: Função de ativação.

Existe o chamado **Universal Approximation Result (1989)**: um Multi-Layer Perceptron (MLP) com uma única camada oculta consegue aproximar qualquer função contínua arbitrariamente bem.

- Take $g(z) = 1$ if $z \geq 1/2$, 0 otherwise.



- What is Y for binary X_1 and X_2?

Figura 4: Exemplo de Multi-Layer Perceptron (MLP).

No exemplo acima, os pesos são:

$$\begin{cases} \beta_0 = 0, \beta_1 = 1, \beta_2 = -1; \\ w_{10} = 0, w_{11} = 1, w_{12} = 1; \\ w_{20} = -1, w_{21} = 1, w_{22} = 1. \end{cases}$$

Isso nos dá que

$$Y = g(X_1 + X_2) - g(X_1 + X_2 - 1),$$

onde $g(z) = 1$ se $z \geq 1/2$ e $g(z) = 0$ caso contrário.

Com muitos outputs, a esquematização é desse jeito:

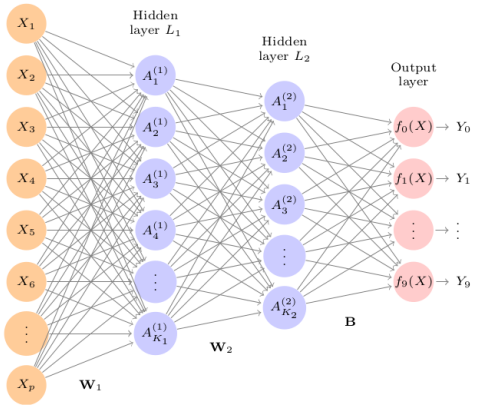


Figura 5: Esquematização de muitos outputs.

No caso de muitos outputs, por exemplo classificar uma imagem para saber qual dígito de 0-9 ela possui, é comum utilizarmos o *one-hot coding*, usando assim 10 variáveis Z_m com valores binários $m = 0$ ou 1, ao invés de somente uma variável com 10 valores de 0-9.

Se todas os 10 outputs fossem separados e independentes, colocaríamos $f_m(X) = Z_m$ e seria isso mesmo, mas note que o *one-hot coding* não é independente. Assim, interpretamos então $f_m(X) = \mathbb{P}(Y = m \mid X)$ e utilizamos a função especial *softmax activation function*

$$f_m(X) = \mathbb{P}(Y = m \mid X) = \frac{e^{Z_m}}{\sum_{\ell=0}^{9} e^{Z_\ell}}.$$

Já que os outputs são qualitativos, para treinar esta rede tentamos estimar os coeficientes que minimizam a log-likelihood multinomial negativa

$$-\sum_{i=1}^{n}\sum_{m=0}^{9} y_{im} \log(f_m(x_i)),$$

também conhecida como *cross-entropy*. Isso é uma generalização do critério de regressão logística para duas classes. Se os outputs fossem quantitativos, nós minimizaríamos o erro quadrático, como veremos na próxima seção.

1.3 Treinando uma MLP

Primeiro, escolher o número de camadas, número de variáveis em cada camada e as funções de ativação. Então, minimizamos o erro quadrático:

$$R_\theta = \frac{1}{2}\sum_{j=1}^{N}(y_j - f_\theta(x_j))^2.$$

O algoritmo que geralmente é usado em redes neurais para minimizar o erro quadrático é o *gradiente descendente*. A cada iteração t, calculamos

$$\boldsymbol{\nabla} R_\theta^t = \left.\frac{\partial R_\theta}{\partial \theta}\right|_{\theta^t}$$

e tomamos $w^{t+1} = w^t - \rho \boldsymbol{\nabla} R_\theta^t$, onde ρ é a *taxa de aprendizado* (o número pequeno do gradiente descendente, nada demais).

Claramente, para calcularmos as derivadas de R_θ^t, aplicamos a *regra da cadeia* sem muito mistério.

Os algoritmos do estado da arte utilizam o *gradiente descendente estocástico* para calcular as derivadas a partir de amostras dos dados selecionadas aleatoriamente.

Também é comum regularizar a minimização com ridge ou lasso:

$$R(\theta;\lambda) = -\sum_{i=1}^{n}\sum_{m=0}^{9} y_{im}\log(f_m(x_i)) + \lambda\sum_j \theta_j^2. \quad \text{(ridge)}$$

1.4 Dropout

A ideia é do método de regularização *dropout* é simplesmente remover aleatoriamente alguma das unidades (variáveis da camada de input) com probabilidade p e reescalar todos as outras unidades por $\frac{1}{1-p}$ para compensar. É mais ou menos uma forma de fazer *feature selection*.

1.5 CNN e RNN

Convolutional Neural Networks (CNN) são redes neurais (utilizadas para tratamento de imagens), só seus pesos w_{kj} são mais estruturados. Interpretamos eles como sendo filtros convolucionais.

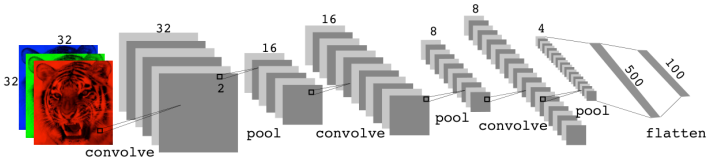


Figura 6: Arquitetura de uma CNN.

Recurrent Neural Networks (RNN) são redes neurais utilizadas para tratamento de dados sequenciais (por exemplo séries temporais).

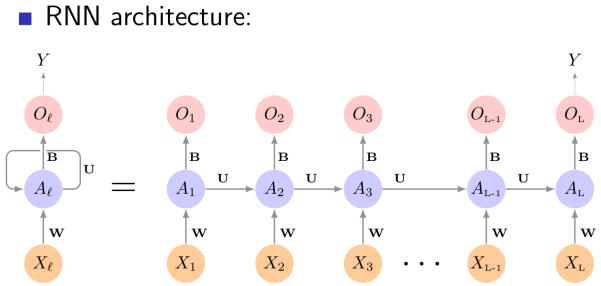


Figura 7: Arquitetura de uma RNN.

1.6 Lembrete

- Classificação: utilizamos *softmax* e a *cross-entropy*.
- Regressão: utilizamos o erro quadrático.

2 Árvores

A ideia de métodos baseados em árvores é literalmente fazer um monte de `if` e `else` para ir classificando as coisas.

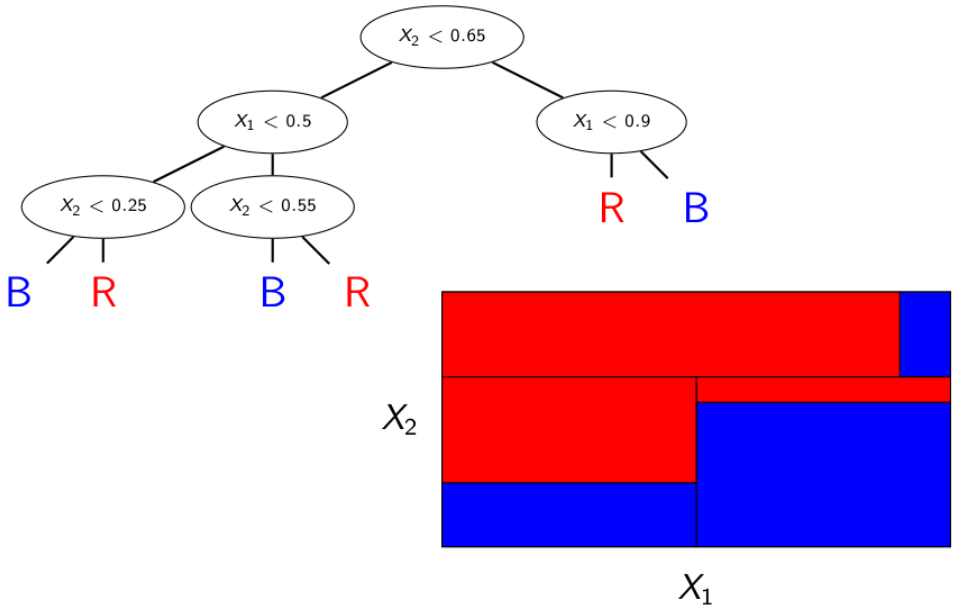


Figura 8: Método baseado em árvores.

É fácil de entender e desenhar, é flexível em capturar fronteiras e consegue lidar com features contínuas ou categóricas.

Para acharmos bons *splits*, para *árvores de regressão* existe a ideia básica (escolha o melhor split para cada passo de forma *greedy*) e para *árvores de classificação*, existe o algoritmo `C4.5`, que é baseado em entropia.

2.1 Árvores de classificação

Utilizamos o algoritmo `C4.5` para classificação.

A entropia de uma conjunto de observações D é

$$I(D) = -\sum_{k=1}^{K} p_k \log(p_k),$$

onde $p_k = |D_k|/|D|$ e D_k é o subconjunto de observações tal que $Y = k$.

O ganho de informação de uma partição de D em datasets $D^1, \ldots, D^m$ é

$$G(D^1, \ldots, D^m) = I(D) - \sum_{j=1}^{m} \frac{|D^j|}{|D|} I(D^j).$$

Assim, o *gain ratio* é $\frac{G(D^1,\ldots,D^m)}{I(D)}$.

O algoritmo é

1. Input: dataset de treino.
2. Se o dataset está vazio, STOP. Se todos os labels no dataset forem idênticos, retornar uma *leaf* com esse label.
3. Para cada split possível, computar a *gain ratio* da partição resultante do dataset de treino.
4. *Greedy step*: selecionar o split com maior *gain ratio*.
5. O split corresponde a um nodo. Cada element da partição do dataset é uma aresta ao nodo possível. Passo recursivo: chamar o algoritmo para cada tal elemento.

Suponha que tenhamos uma região contendo algumas observações coletadas no dataset D. O índice Gini dessa região é

$$G = \sum_{k=1}^{K} \frac{|D_k|}{|D|}\left(1 - \frac{|D_k|}{|D|}\right) = 1 - \sum_{k=1}^{K} \left(\frac{|D_k|}{|D|}\right)^2.$$

Ele não é nada mais do que expandir o log em primeira ordem em Taylor (entropia linear) e é uma medida de pureza (assim como na Mecânica Quântica, para a matriz densidade).

Podemos avaliar um split com a entropia ou com Gini index (que é uma aproximação da entropia).

2.2 Árvores de regresão

No caso de Y ser uma variável contínua, a ideia é a mesma: fazer splits de forma *greedy*, só que dessa vez minimizamos o RSS

$$\sum_{j=1}^{J} \sum_{i \in R_j} (y_i - \hat{y}_{R_j})^2.$$

Em cada passo, para cada j e s nós definimos os pares

$$R_1(j,s) = \{X \mid X_j < s\} \quad \text{e} \quad R_2(j,s) = \{X \mid X_j \geq s\},$$

e procuramos os valores de j e s que minimizam a expressão

$$\sum_{i:\, x_i \in R_1(j,s)} (y_i - \hat{y}_{R_1})^2 + \sum_{i:\, x_i \in R_2(j,s)} (y_i - \hat{y}_{R_2})^2.$$

No próximo passo, ao invés de splitar o espaço inteiro, nós splitamos uma das duas regiões identificadas previamente. O processo continua até um critério de parada ser satisfeito.

O processo descrito acima pode ter bons resultados no dataset de treino, mas é bem provável que aconteça um overfitting. Isso é porque a árvore resultante pode acabar sendo muito complexa.

Uma boa estratégia é montar uma árvore grande T_0 e depois cortá-la para obter uma sub-árvore menor.

Podemos selecionar um pequeno subconjunto de sub-árvores através do *cost complexity pruning*. Nós consideramos uma sequência de árvores indexadas por um parâmetro $\alpha \geq 0$. Para cada valor de α corresponde uma sub-árvore $T \subset T_0$ tal que

$$\sum_{m=1}^{|T|} \sum_{i:\, x_i \in R_m} (y_i - \hat{y}_{R_m})^2 + \alpha |T|$$

é o mínimo possível. Aqui, $|T|$ indica o número de *leaves* da árvore T. O parâmetro α controla o trade-off entre a complexidade da árvore e o seu fit ao dataset de treino. A ideia então é selecionar vários valores de α e aplicar validação cruzada para tuná-lo para o melhor possível que minimizar o erro médio.

2.3 Vantagens e desvantagens

Vantagens:

- Árvores são fáceis de entender.
- Podem ser traduzidas em "regras".
- São mais flexíveis que regressão logística (basicamente linear).
- Conseguem lidar com variáveis contínuas ou categóricas.

Desvantagens:

- Não têm muita acurácia.
- Não são robustas; muito sensíveis ao dataset de treino (variância é grande).

2.4 Bagging trees

Uma única árvore é boazinha. Mas por que não aprender um conjunto de árvores e fazer a média dos resultados? Essa ideia leva a *bagging trees* e *random forests*.

Bagging é o seguinte. Pegamos o dataset de treino e produzimos B amostras dele próprio através de *bootstrap*. Então tomamos a média para regressão e a maioria por votos para classificação. Isso é *bagging*, e funciona para qualquer tipo de classificador. Geralmente é aplicado para *weak classifiers*, que sozinhos não são muito bons mas com *bagging* o resultado é melhorado. Esse é o caso para árvores de classificação ou regressão.

Para estimar o erro utilizando *bagging*, não é necessário fazer validação cruzada. Basta que, para cada observação, nós produzimos a média de todas as árvores que foram aprendidas sem essa observação no conjunto de treino. Assim computamos o erro médio para essa observação. Isso é uma boa estimativa do erro, e quando $B \to \infty$ este erro tende ao erro do leave-one-out cross validation.

2.5 Random forests

O método de *random forest* é dentro do contexto de *bagging*. Só que, todavez ao fazer um split, nos restringimos a um subconjunto de m features (das p features/predictors $X_1, \ldots, X_p$) escolhido aleatoriamente para realizar o splitting. Normalmente toma-se $m \approx \sqrt{p}$.

Pela influência aleatória dos splis, as árvores tornam-se mais descorrelacionadas com as outras. Pela diminuição de correlação, a variância é reduzida.

Note que se tomássemos $m = p$ isso se reduziria ao caso normal de *bagging*, já que qualquer split possível seria considerado.

Geralmente *random forests* apresentam uma pequena melhoria com relação ao *bagging* normal.

2.6 Boosting

Boosting é um método geral para melhorar os resultados de um conjunto de *weak classifiers* $g_k(X)$ (eles são fracos no sentido de que podem ser um pouco melhor do que chutes aleatórios, mas podem em tese ter boa acurácia se disponível).

A ideia de *boosting* é combinar eles de maneira a produzir um classificador mais foda. Enquanto g_1 é aprendido no training dataset original, cada g_k é aprendido numa versão ponderada do dataset original. Em outras palavras, sendo a variável de classe $Y \in \{-1, 1\}$, o *boosted classifier* é

$$g(x) = \text{sign}\left(\sum_{k=1}^{M} \alpha_k g_k(x)\right),$$

para alguns α_k.

Ideia simples:

- Aprender g_1.
- Aprender g_2, enfatizando que deve fazer a coisa certa quando g_1 cometer um erro.
- Aprender g_3, enfatizando que deve fazer a coisa certa quando g_1 ou g_2 cometerem um erro. And so on...

O algoritmo mais famoso é o `AdaBoost`:

1. Inicializar $w_j = 1/N$, para $j \in \{1, \ldots, N\}$.
2. Para k de 1 a M:
   (a) Aprender $g_k(X)$ no training dataset usando os pesos w_j.
   (b) Calcular
   $$\beta_k = \frac{\sum_{j=1}^{N} w_j \, I_{y_j \neq g_k(x^j)}}{\sum_{j=1}^{N} w_j}.$$
   (c) Calcular $\alpha_k = \ln\left(\frac{1-\beta_k}{\beta_k}\right)$.
   (d) Multiplicar cada peso w_j por $\exp\left(\alpha_k I_{y_j \neq g_k(x^j)}\right)$.
3. Output $g(x) = \text{sign}\left\{\sum_{k=1}^{M} \alpha_k g_k(x)\right\}$.

Um esquema popular é utilizar *boosting* com *stumps* (troncos: árvores com um único split). Boosted classifiers com um grande número de stumps funciona muito bem.

O outro algoritmo de boosting é parecido com um gradiente descendente.

Algorithm 8.2 *Boosting for Regression Trees*

1. Set $\hat{f}(x) = 0$ and $r_i = y_i$ for all i in the training set.
2. For $b = 1, 2, \ldots, B$, repeat:
   (a) Fit a tree $\hat{f}^b$ with d splits ($d+1$ terminal nodes) to the training data (X, r).
   (b) Update $\hat{f}$ by adding in a shrunken version of the new tree:
   $$\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}^b(x). \tag{8.10}$$
   (c) Update the residuals,
   $$r_i \leftarrow r_i - \lambda \hat{f}^b(x_i). \tag{8.11}$$
3. Output the boosted model,
   $$\hat{f}(x) = \sum_{b=1}^{B} \lambda \hat{f}^b(x). \tag{8.12}$$

Figura 9: Algoritmo tipo gradiente de boosting.

3 Aprendizado não-supervisionado

Como não temos acesso aos rótulos Y para checar nossos resultados, a ideia consiste em fazer uma análise exploratória para entender os dados melhor. Nisso entram técnicas de data visualization, agrupamento e estatística. Representation learning é justamente o processo de tentar representar os dados da melhor forma possível para aprender seus padrões.

3.1 PCA

O *Principal Component Analysis* (PCA) é basicamente uma mudança de coordenadas para conseguir explorar as combinações de features que mais importam no dataset.

Assumindo que $X_1, \ldots, X_n$ sejam normalizados e com média zero, definimos cada componente principal Z_i como a combinação linear das features original que tenham as maiores variâncias:

$$Z_j = \phi_{1j}X_1 + \phi_{2j}X_2 + \cdots + \phi_{nj}X_n,$$

onde $\sum_j \phi_{ij}^2 = 1$ e $\sigma_1^2 \geq \sigma_2^2 \geq \cdots \geq \sigma_p^2$, sendo cada σ_j^2 a variância de Z_j. Visualmente a ideia é tipo essa aqui

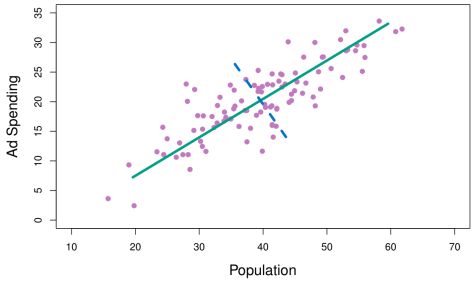


Figura 10: Ideia de achar as componentes principais no PCA.

Os coeficientes ϕ_{ij} são chamados de *loadings*. A variância de Z_1 é

$$\sigma_1^2 = \sum_{j=1}^{N} \left(\sum_{i=1}^{n} \phi_{n1} x_{ij} \right)^2.$$

Para acharmos os loadings, temos que maximizar

$$\max_{\phi_{11},\ldots,\phi_{n1}} \sum_{j=1}^{N} \left(\sum_{i=1}^{n} \phi_{n1} x_{ij} \right)^2,$$

com o vínculo $\sum_{i=1}^{n} \phi_{i1}^2 = 1$. É um problema quadrático que pode ser resolvido com good software. Após achar Z_1, achar Z_2, Z_3 até Z_n para algum n que seja bom.

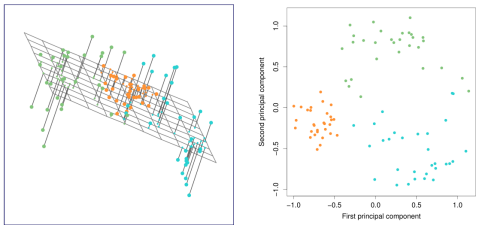


Figura 11: Exemplo em três dimensões ($N = 3$) e duas componentes principais ($n = 2$).

Se utilizarmos somente as componentes principais, conseguimos explicar a maior parte da variância dos dados. Com esperança, com as componentes principais temos features melhores.

A variância total é

$$\frac{1}{N}\sum_{i=1}^{n}\sum_{j=1}^{N}x_{ij}^2.$$

A variância é explicada pela componente principal Z_k é

$$\frac{1}{N}\sum_{j=1}^{N}z_{jk}^2.$$

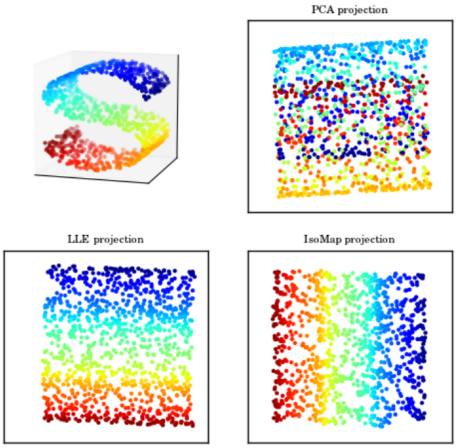


Figura 12: Outras técnicas de manifolds.

Quantas componentes principais usar? Pergunta difícil de responder. Mas uma ideia prática é checar a proporção das variâncias e parar quando elas forem baixas o suficiente.

O PCA é uma ferramenta para explorar os dados, mas também pode ser usada para produzir um conjunto mais compacto de features.

O algoritmo para resolver o PCA na prática é traduzido em termos de autovalores e autovetores, e diagonalização de matrizes simétricas.

Consideramos a matriz

$$\mathbf{X} = \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \cdots \\ \mathbf{x}_N \end{pmatrix}$$

e então calculamos

$$\frac{1}{N-1}\mathbf{X}^T\mathbf{X}.$$

Cada elemento dessa matriz é a covariância

$$\mathbf{A} = \frac{1}{N-1}\sum_{i=1}^{N}x_{ij}x_{kj}.$$

As componentes principais então são dadas por

$$\begin{pmatrix} Z_1 \\ \vdots \\ Z_n \end{pmatrix} = \mathbf{U}^T \begin{pmatrix} X_1 \\ \vdots \\ X_n \end{pmatrix},$$

onde $\mathbf{U}$ é a matriz de autovetores $\mathbf{A}$. O melhor algoritmo para resolver isso *singular value decomposition* (SVD).

Resumindo:

1. Compute $\mathbf{m} = \sum_{j=1}^{N} \mathbf{x}_j$ and
$$\mathbf{S} = (\mathbf{X} - \mathbf{m})^T(\mathbf{X} - \mathbf{m}).$$
2. Compute the eigenvalues and eigenvectors of $\mathbf{S}$.
3. Order the eigenvectors by the value of their eigenvalues (from largest to smallest).
4. Select the K eigenvectors with largest eigenvalues, and build the matrix
$$\mathbf{U} = [\mathbf{u}_1 \dots \mathbf{u}_K].$$
5. Then define
$$\mathbf{Z} = \mathbf{U}^T(\mathbf{X} - \mathbf{m}).$$

Figura 13: Algoritmo para achar as componentes principais Z_k do PCA.

3.2 Clustering

A ideia é achar subgrupos do dataset, cada subgrupo é chamado de *cluster*. É um pouco subjetivo e tenta realizar uma medida de similaridade. A ideia é minimizar similaridades intra-cluster e maximizar similaridades inter-clusters. Existem dois métodos importantes: K-means e hierarchical clustering.

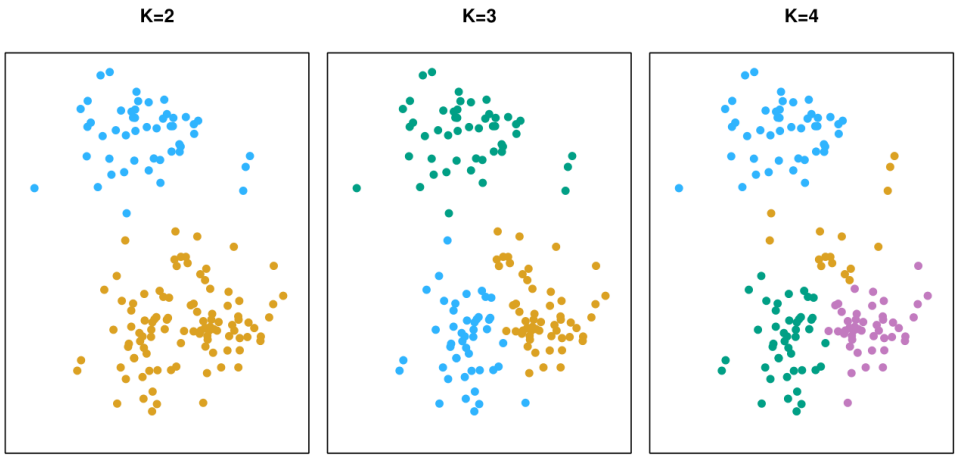


Figura 14: Exemplo de clustering com $K-$means.

Como dito anteriormente, devemos achar clusters $C_1, \dots, C_K$ que maximizem

$$\max_{C_1,\dots,C_K} \sum_{k=1}^{K} s(C_k)$$

para alguma medida s de intra-cluster similarity. Uma medida razoável é

$$s(C_k) = -\frac{1}{|C_k|} \sum_{x',x'' \in C_k} \sum_{i=1}^{n} (x_i' - x_i'')^2.$$

3.2.1 K-means

O $K-$means não é um algoritmo estrito que maximiza a quantidade acima, mas é um método aproximado que dá uma solução razoável para esse problema.

O algoritmo do $K-$means é

1. Input: dataset, medida de distância, número K.
2. Selecionar a atribuição de K clusters iniciais.
    - Uma possibilidade: selecionar eles aleatoriamente.
    - Outra possibilidade: selecionar aleatoriamente K pontos como centroides e atribuir pontos mais próximos aos centroides.
3. Iterar até um critério de parada for atingido.
    - Calcular um centroide para cada cluster.
    - Agrupar cada ponto ao centroide mais próximo (utilizar a métrica de distância).

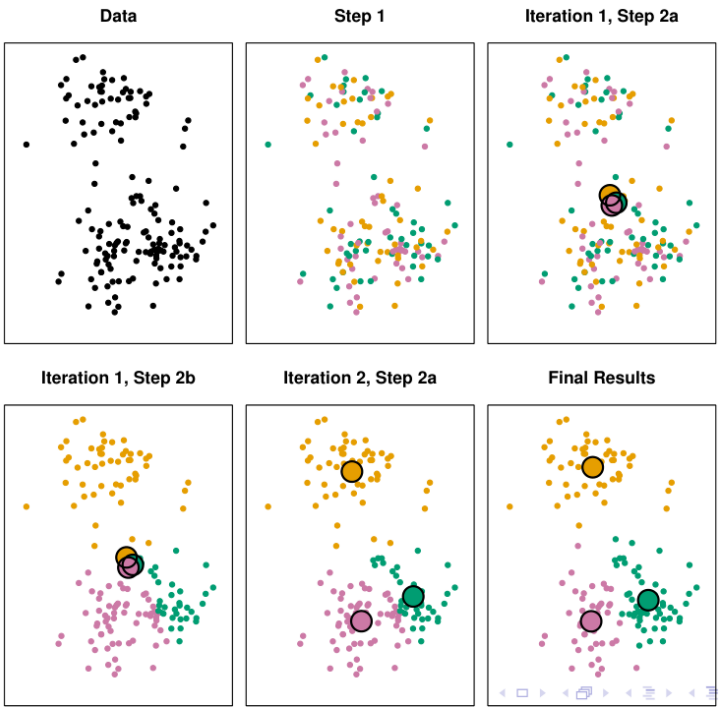


Figura 15: Iterações de $K-$means.

Quando parar?

- Depois de um número máximo de iterações.
- Quando os centroides pararem de mudar (ou tiverem pouca mudança).
- Quando a medida $\sum_{k=1}^{K} s(C_k)$ tiver pouca mudança.

- Easy to understand, easy to implement.
- Cost is proportional to number of points, number of clusters, and number of iterations.

Figura 16: Vantagens do $K-$means.

- Requires K.
- Results depend on initial guess.
- Sensitive to outliers.
- No real theoretical guarantees; stopping is somewhat ad hoc.
- Centroid computation may be meaningless for categorical data.

Figura 17: Fraquezas do $K-$means.

3.2.2 Hierarchical clustering

Algorithm 12.3 *Hierarchical Clustering*

1. Begin with n observations and a measure (such as Euclidean distance) of all the $\binom{n}{2} = n(n-1)/2$ pairwise dissimilarities. Treat each observation as its own cluster.
2. For $i = n, n-1, \ldots, 2$:
   (a) Examine all pairwise inter-cluster dissimilarities among the i clusters and identify the pair of clusters that are least dissimilar (that is, most similar). Fuse these two clusters. The dissimilarity between these two clusters indicates the height in the dendrogram at which the fusion should be placed.
   (b) Compute the new pairwise inter-cluster dissimilarities among the $i-1$ remaining clusters.

Figura 18: Algoritmo de hierarchical clustering.

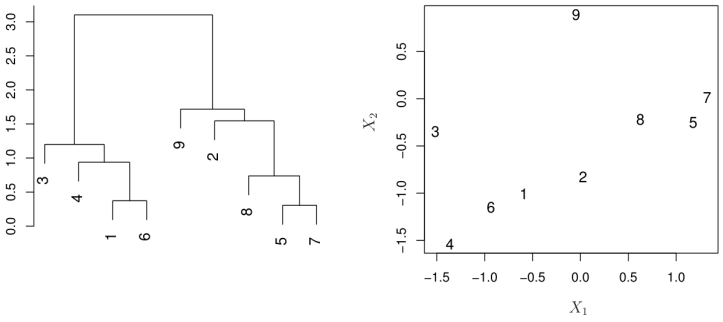


Figura 19: Esquematização de hierarchical clustering.

Complete linkage: distance between two points (one in each cluster) that are most distant.

Single linkage: distance between two points (one in each cluster) that are closest.

Average linkage: average of the distance between all pairs of points (one in each cluster).

Centroid linkage: distance between centroids of both clusters.

Figura 20: Maneiras de calcular distâncias entre clusters.

4 Decisões sequenciais e Aprendizado por reforço

4.1 Markov decision processes (MDPs)

- MDPs are quite popular in economics, management and operations research.
- An MDP consists of
  1. A state space S.
  2. An action space A.
  3. Transition probabilities $p_a(r|s) = P_a(s_{t+1} = r|s_t = s)$.
  4. Costs $c_a(s)$.
- Often represented as graphs where nodes are states.
- Another representation: a transition matrix P_a for each action a.

Figura 21: Descrição de MDPs.

4.2 Reinforcement Learning

- Basic idea: learning to behave based only on reward/punishment after actions.
- Often thought of as the estimation of an MDP based only on information about the cost of each visited state.
  - Sometimes learning is passive, sometimes active (that is, with active exploration).
- Often it is not necessary to learn the whole MDP; just to search for the optimal policy.
- Often just the value function is learned; sometimes just approximations of the value function.

Figura 22: Descrição de Reinforcement Learning.