Capítulo 3

Aprendizagem não Supervisionada

Até agora, orientou-se a máquina para que esta aprendesse a encontrar a solução para o nosso problema. Na regressão, treina-se a máquina para prever um valor futuro. Na classificação, treina-se a máquina para classificar um objeto desconhecido numa das categorias previamente definidas. Em suma, treinam-se máquinas para que possam prever Y para nossos dados X.

Dado um enorme conjunto de dados e não estimando nem o número nem as categorias, seria difícil treinar a máquina usando a aprendizagem supervisionada.

E se a máquina pudesse pesquisar e analisar a "big data", em execução em vários Gigabytes e Terabytes, e determinar que esses dados contêm um certo conjunto de categorias distintas?

Considere-se por exemplo os dados dos eleitores.
Através de algumas entradas (variáveis) de cada eleitor (chamados recursos na terminologia de IA), deixa-se a máquina prever o número de eleitores que votariam no partido político X, Y, e assim sucessivamente.

Em geral, questiona-se a máquina sobre um enorme conjunto de pontos de dados X:
"O que me pode dizer sobre X?".

Ou pode ser uma pergunta como
"Quais os cinco melhores grupos que podemos fazer com X?".

Ou poderia ser até mesmo
"Quais são os três recursos que ocorrem juntos com mais frequência no partido X?".

É exatamente nisto que consiste a Aprendizagem não Supervisionada

Vamos agora considerar o algoritmo mais utilizado em classificação, na aprendizagem de máquina não supervisionado.

3.1 Agrupamento (Clustering) k-means

As eleições presidenciais de 2000 e 2004 nos Estados Unidos foram apertadas – muito apertadas. A maior percentagem do voto popular de cada candidato foi de $50,7\%$ e a menor foi de $47,9\%$.

Se uma pequena percentagem dos eleitores mudasse de sentido de voto, o resultado das eleições teria sido diferente.

Existem pequenos grupos de eleitores que, quando devidamente aliciados, mudam de lado e isso pode fazer toda a diferença.

Esses grupos podem não ser enormes, mas em disputas tão renhidas, podem ser suficiente grandes para mudar os resultados das eleições.

Como encontrar esses grupos de pessoas?
Como cativá-los com um orçamento limitado?

A resposta é a associação, agrupamento, *clustering*...

Com efeito, pode considerar-se a seguinte sequência:

- Primeiro trata-se de recolher informações sobre essas pessoas (com ou sem seu consentimento): qualquer tipo de informação que possa dar alguma pista sobre o que consideram verdadeiramente relevante e que influenciará o seu sentido de voto.
- Essas informações são introduzidas nalgum tipo de algoritmo de *clustering* e, de seguida, para cada *cluster* (será conveniente começar pelo maior), cria-se uma mensagem que atrairá esses eleitores.
- Por fim, difunde-se essa mensagem na campanha e mede-se até que ponto a estratégia está a funcionar.

Clustering é um tipo de aprendizagem não supervisionada que forma automaticamente agrupamentos de coisas semelhantes. É como uma classificação mas, neste caso, automática. Pode agrupar-se quase tudo, e quanto mais semelhantes forem os itens no *cluster*, melhores serão os agrupamentos.

Neste capítulo, estuda-se um tipo de algoritmo de agrupamento chamado k-means. É chamado k-means porque encontra "k" *clusters* únicos, e o centro (centroide) de cada *cluster*, Figura 3.1, é a média dos valores nesse agrupamento.
K-means é muitas vezes referido como algoritmo de Lloyd. Em termos básicos, o algoritmo tem três etapas. A primeira etapa escolhe os centróides iniciais, sendo o método mais básico a escolha de um ponto do conjunto de dados. Após a inicialização, o K-means faz um ciclo com as duas etapas seguintes.

Na segunda etapa, para cada ponto, determina-se o seu centróide mais próximo e associa-se o ponto a esse cluster. Na terceira etapa cria novos centroides (ou recentra os centroides anteriores) considerando o valor médio de todos os pontos associados previamente a cada cluster.

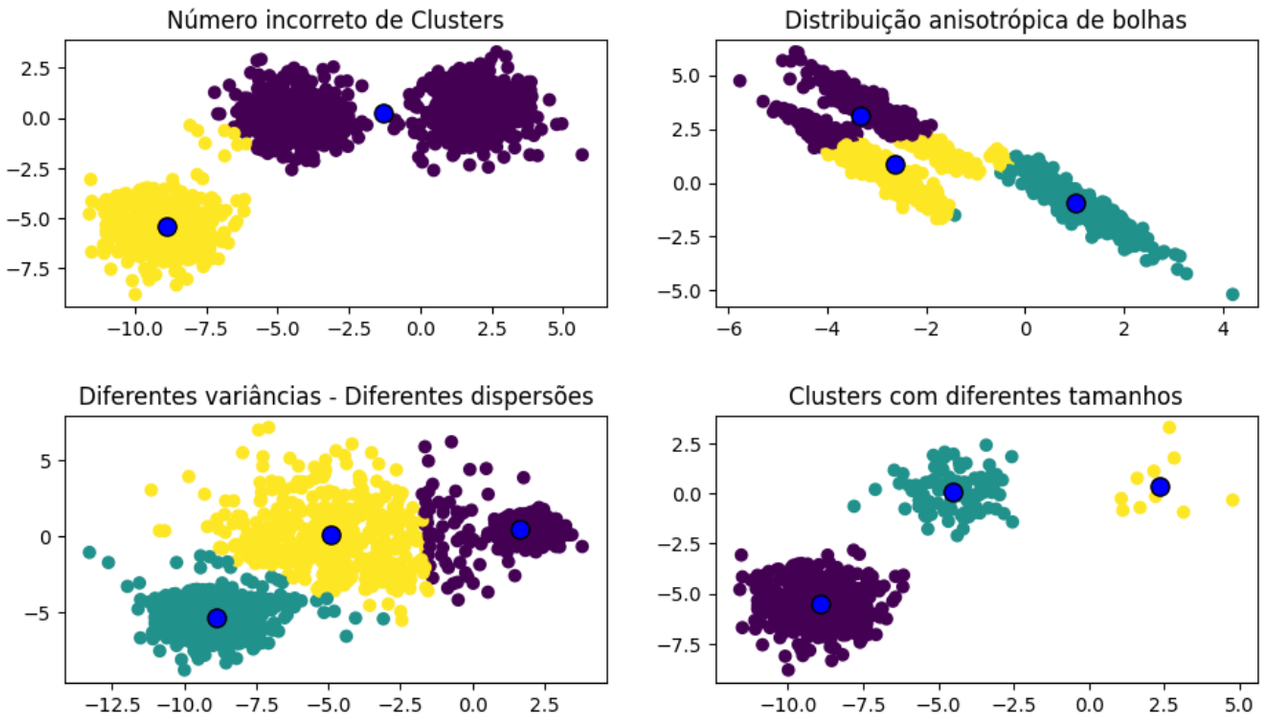

Figura 3.1: Agrupamento k-means clustering com centroides

A diferença entre o centroide antigo e o novo é computada e o algoritmo repete as duas últimas etapas até esse valor ser menor que um determinado limite. Noutras palavras, repete-se o ciclo até que os centróides se acabem por fixar definitivamente (ou, em alternativa, até se atingir um número máximo de iterações).

Apresenta-se a seguir um possível código[1] Python para a solução de um problema de *Clustering*, semelhante ao descrito na Figura 3.1, usando o algoritmo K-means.

```
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs
plt.figure(figsize=(12, 12))
n_samples = 1500
random_state = 170
X, y = make_blobs(n_samples=n_samples, random_state=random_state)
```

make_blobs(n_samples, random_state, cluster_std, centers, n_features):

n_samples: Se int, é o número total de pontos dividido igualmente entre os clusters. Se for do tipo array (por ex. [100, 500, 1000]), cada elemento da sequência indica o número de amostras por cluster.

[1]Ficheiro "..//plot kmeans assumptionscom centroides.py"

random_state: O parâmetro **random_state** determina/condiciona a geração de números aleatórios. Se se considera o valor padrão `random_state=None`, cada vez que se corre um determinado algoritmo (função) este irá produzir resultados diferentes.

cluster_std: O desvio padrão dos clusters. Se for do tipo array (por ex. [1.0, 2.5, 0.5]), cada cluster terá um desvio padrão distinto.

centers: O número de centros a serem gerados ou as localizações de centros fixos. Se *n_samples* = *int* e *centers* = *None*, 3 centros serão gerados. Se *n_samples* for semelhante a uma matriz, os centros devem ser None ou uma matriz de comprimento igual ao comprimento de n_samples.

n_features: Representa o número de coordenadas (atributos) do "vetor" X. Para ser possível a sua visualização gráfica n_features terá que ser igual a 2, ou quando muito, igual a 3.

Gráfico 1 da Figura 3.1, dados com um número incorreto de Clusters

```
# Incorrect number of clusters
y_pred = KMeans(n_clusters=2, random_state=random_state).fit_predict(X)
plt.subplot(221)
plt.scatter(X[:, 0], X[:, 1], c=y_pred)
# Plot the centroids as a black/blue O - Calculate from kmeans++
from sklearn.cluster import kmeans_plusplus
kmeans = KMeans(init="k-means++", n_clusters=2, n_init=10)
kmeans.fit(X)
# Plot the centroids as a black/blue O
centroids = kmeans.cluster_centers_
plt.scatter(centroids[:, 0], centroids[:, 1], marker="o", s=90,
    linewidths=1, color="b", edgecolor="black")
plt.title("Número incorreto de Clusters")
```

KMeans(n_clusters=8, init='k-means++', max_iter=300, random_state=None, copy_x=True, algorithm='auto).fit_predict(X):

n_clusters: O número de clusters a ser considerado (8, por defeito).

init: Método de inicialização. ('k-means++', por defeito: seleciona centros de cluster iniciais para clustering k-mean de forma inteligente para acelerar a convergência. Não é necessário n_clusters.

n_initint: Número de vezes que o algoritmo k-means será executado com diferentes sementes de centroides (10, por defeito). Os resultados finais serão a melhor saída de n_init execuções consecutivas em termos de inércia.

max_iter: Número de vezes que o algoritmo k-means será executado com diferentes sementes de centroides (300, por defeito).

random_state: O parâmetro **random_state** determina/condiciona a geração de números aleatórios. Se se considera o valor padrão `random_state=None`, cada vez que se corre um determinado algoritmo (função) este irá produzir resultados diferentes.

***.fit_predict(X)**: Computa o algoritmo k-means clustering para os dados X.

Gráfico 2 da Figura 3.1, dados com distribuição anisotrópica[1]

```
# Anisotropicly distributed data
transformation = [[0.60834549, -0.63667341], [-0.40887718, 0.85253229]]
X_aniso = np.dot(X, transformation)
y_pred = KMeans(n_clusters=3, random_state=random_state).fit_predict(X_aniso)
plt.subplot(222)
plt.scatter(X_aniso[:, 0], X_aniso[:, 1], c=y_pred)
# Plot the centroids as a black/blue O - Calculate from kmeans++
from sklearn.cluster import kmeans_plusplus
kmeans = KMeans(init="k-means++", n_clusters=3, n_init=10)
kmeans.fit(X_aniso)
# Plot the centroids as a black/blue O
centroids = kmeans.cluster_centers_
plt.scatter(centroids[:, 0], centroids[:, 1], marker="o", s=90,
    linewidths=1, color="b", edgecolor="black")
plt.title("Distribuição anisotrópica de bolhas")
```

np.dot(X, transformation): Multiplicação (generalizada) de matrizes ou de matrizes por vetores.

Gráfico 3 da Figura 3.1, dados com diferentes variâncias - diferentes dispersões

```
# Different variance
X_varied, y_varied = make_blobs(n_samples=n_samples,
        cluster_std=[1.0, 2.5, 0.5], random_state=random_state)
y_pred = KMeans(n_clusters=3,random_state=random_state).fit_predict(X_varied)
plt.subplot(223)
plt.scatter(X_varied[:, 0], X_varied[:, 1], c=y_pred)
```

[1] Distribuições de pontos muito diferenciadas consoante as direções.

```
# Plot the centroids as a black/blue O - Calculate from kmeans++
from sklearn.cluster import kmeans_plusplus
kmeans = KMeans(init="k-means++", n_clusters=3, n_init=10)
kmeans.fit(X_varied)
# Plot the centroids as a black/blue O
centroids = kmeans.cluster_centers_
plt.scatter(centroids[:, 0], centroids[:, 1], marker="o", s=90,
    linewidths=1, color="b", edgecolor="black")
plt.title("Diferentes variâncias - Diferentes dispersões")
```

cluster_std=[1.0, 2.5, 0.5]: Diferentes desvio padrão para cada clusters.

Gráfico 4 da Figura 3.1, Clusters com diferentes tamanhos

```
# Unevenly sized blobs
X_filtered = np.vstack((X[y == 0][:500], X[y == 1][:100], X[y == 2][:10]))
#ou X_filtered, y = make_blobs(n_samples=[500,100,10], random_state=random_state)
y_pred = KMeans(n_clusters=3, random_state=random_state).fit_predict(X_filtered)

plt.subplot(224)
plt.scatter(X_filtered[:, 0], X_filtered[:, 1], c=y_pred)
# Plot the centroids as a black/blue O - Calculate from kmeans++
from sklearn.cluster import kmeans_plusplus
kmeans = KMeans(init="k-means++", n_clusters=3, n_init=10)
kmeans.fit(X_filtered)

# Plot the centroids as a black/blue O
centroids = kmeans.cluster_centers_
plt.scatter(centroids[:, 0], centroids[:, 1], marker="o", s=90,
    linewidths=1, color="b", edgecolor="black")
#plt.title("Unevenly Sized Blobs")
plt.title("Clusters com diferentes tamanhos")

plt.show()
```

np.vstack((X[y == 0][:500], X[y == 1][:100], X[y == 2][:10])):
O mesmo que **make_blobs(n_samples=[500, 100, 10], ...)**, onde cada elemento da sequência indica o número de amostras do respetivo cluster.

Guia geral do algoritmo de agrupamento k-means

Parâmetros:
KMeans(n_clusters=8, *, init='k-means++', n_init=10, max_iter=300, tol=0.0001, verbose=0, random_state=None, copy_x=True, algorithm='auto')

n_clusters int, default=8

O número de clusters a serem formados e respetivo número de centroides.

init: {'k-means++', 'random'}, default='k-means++'

dados (X, n_clusters) ou um vetor da forma (n_clusters, n_features - caraterísticas / funcionalidades)

Método de inicialização:
'k-means++' : seleciona os centros dos clusters iniciais do algoritmo k-means de forma inteligente, para acelerar a convergência.
'random': escolha de n_clusters aleatoriamente dos dados para os centróides iniciais.

Se for dado um vetor, deve ter formato (n_clusters, n_features) e fornecer os centros iniciais.

Se forem introduzidos dados, deverão conter os argumentos, X, n_clusters e um estado aleatório e reinicialização.

n_init: int, default=10

Número de vezes que o algoritmo k-means será executado com diferentes sementes de centroides. Os resultados finais serão a melhor saída de n_init execuções consecutivas em termos de inércia.

max_iter: int, default=300

Número máximo de iterações do algoritmo k-means para uma única execução.

tol: float, default=1e-4

Tolerância relativa em relação à norma de Frobenius da diferença nos centros do cluster de duas iterações consecutivas para declarar convergência.

verbose: int, default=0 Modo verbosidade.

random_state: int, RandomState instance or None, default=None

O parâmetro **random_state** determina/condiciona a geração de números aleatórios. Se se considera o valor padrão `random_state=None`, cada vez que se corre um determinado algoritmo (função) este irá produzir resultados diferentes.

copy_x: bool, default=True

Ao pré-computar distâncias, é numericamente mais preciso centralizar os dados primeiro. Se copy_x for True (padrão), os dados originais não serão modificados. Se for False, os dados originais são modificados e colocados antes que a função retorne, mas pequenas diferenças numéricas podem ser introduzidas subtraindo e adicionando a média dos dados. Se os dados originais não forem contíguos a C, será feita uma cópia, mesmo se copy_x for False. Se os dados originais forem esparsos, mas não estiverem no formato CSR (formato de matriz esparsa), será feita uma cópia mesmo se copy_x for False.

algorithm: {"auto", "full", "elkan"}, default="auto"

O algoritmo K-means a usar. O algoritmo clássico do estilo EM é "completo". A variação "elkan" é mais eficiente em dados com clusters bem definidos, usando a desigualdade triangular. No entanto, consome mais memória devido à alocação de uma matriz extra de forma (n_samples, n_clusters).

Por enquanto "auto"(mantido para compatibilidade com versões anteriores) escolhe "elkan", mas pode mudar no futuro para uma melhor heurística[1].

Atributos:

cluster_centers_ndarray da forma (n_clusters, n_features)

Coordenadas dos centros do cluster. Se o algoritmo parar antes de convergir totalmente (consulte tol e max_iter), eles não serão consistentes com os labels_.

labels_ndarray da forma (n_samples,)

Etiquetas (Labels) de cada ponto

inertia_float

Soma dos quadrados das distâncias das amostras ao centro do cluster mais próximo, ponderado pelos pesos das amostras, se fornecido.

n_iter_int

Número de iterações executadas.

n_features_in_int

Número de recursos (features, coordenadas do "vetor" X) considerados durante o ajuste.

feature_names_in_ndarray of shape (n_features_in_,)

Número de recursos (features, coordenadas do "vetor" X) considerados durante o ajuste. Definido apenas quando todos os nomes dos recursos de X são strings.

De seguida são esquematizadas as diversas funcionalidades (métodos e respetivos parâmetros) associadas ao algoritmo `K-Means`.

[1] Processo pedagógico que encaminha os alunos a descobrir por si mesmos, geralmente através de perguntas; Arte de inventar ou descobrir.

KMeans( ).Métodos	
fit(X[, y, sample_weight])	Computa k-means clustering.
fit_predict(X[, y, sample_weight])	Calcula os centroides dos clusters e prevê o índice de cluster para cada amostra.
fit_transform(X[, y, sample_weight])	Calcula clustering e transforma X num espaço de distância de cluster.
get_params([deep])	Obter parâmetros do modelo.
predict(X[, sample_weight])	Prevê o cluster mais próximo de cada amostra.
score(X[, y, sample_weight])	Oposto do valor de X no objetivo K-means.
set_params(**params)	Define os parâmetros do modelo.
transform(X)	Transforma X num espaço de distância de cluster.

Relativamente ao Método `fit(X[, y, sample_weight])` temos os seguintes inputs (parâmetros) e outputs (saídas).

fit(X, y, sample_weight)	
Parâmetros	X:{array-like, sparse matrix} da forma (n_samples, n_features) Condições de treino para cluster. Deve notar-se que os dados serão convertidos para ordenação C, o que causará uma cópia de memória se os dados fornecidos não forem C-contíguos. Se for dada uma matriz esparsa, será feita uma cópia se não estiver no formato CSR.
	y: Ignored Não usado, presente aqui para consistência da API por convenção.
	sample_weight: array-like da forma (n_samples,), default=None Os pesos (weights) para cada observação em X. Se None, todas as observações recebem o mesmo peso.
Saída	self: object Modelo ajustado.

A funcionalidade `fit(X, y, sample_weight)` com os mesmos inputs (`(X, y, sample_weight)`) tem as diferentes extensões (outputs).

fit_"_____"(X, y, sample_weight)	
fit_predict(X, y, sample_weight)	labels: ndarray da forma (n_samples,) Índice do cluster a que pertence cada amostra.
fit_transform(X, y, sample_weight)	X_new: ndarray da forma (n_samples, n_clusters) X transformado num novo espaço.

Após o ajustamento do modelo (`KMeans( ).fit(X, y, sample_weight)`) temos ainda as funcionalidades:

A função que identifica os parâmetros do modelo `get_params(deep=True)`, com os seguintes inputs (parâmetros).

get_params(deep=True)	
Parâmetros	deep: bool, default=True Se True, devolve os parâmetros deste modelo.
Saída	params: dict Nomes de parâmetros mapeados para os seus valores.

A funcionalidade habitual `predict(X, sample\_weight=None)`, que identifica o cluster da amostra `X`.

predict(X, sample_weight=None)	
Parâmetros	X:{array-like, sparse matrix} da forma (n_samples, n_features) Novos dados para prever.
	sample_weight: array-like da forma (n_samples,), default=None Os pesos (weights) para cada observação em X. Se None, todas as observações recebem o mesmo peso.
Saída	labels: ndarray da forma (n_samples,) Índice do cluster a que pertence cada amostra.

A função que identifica os parâmetros do modelo `score(X, y, sample\_weight)`, com os seguintes inputs (parâmetros).

score(X, y, sample_weight)	
Parâmetros	X:{array-like, sparse matrix} da forma (n_samples, n_features) Novos dados.
	y: Ignored Não usado, presente aqui para consistência da API por convenção.
	sample_weight: array-like da forma (n_samples,), default=None Os pesos (weights) para cada observação em X. Se None, todas as observações recebem o mesmo peso.
Saída	score: float Oposto do valor de X no objetivo K-means.

A função que define novos parâmetros do modelo `set\_params(**params)`, com os seguintes inputs (parâmetros).

set_params(**params)	
Parâmetros	**params: dict Parâmetros do estimador.
Saída	self: estimator instance Instância do estimador.

A função que define o transformado de X num novo espaço, após o ajustamento do modelo.

transform(X)	
Parâmetros	X:{array-like, sparse matrix} da forma (n_samples, n_features)
Saída	X_new: ndarray of shape (n_samples, n_clusters) O transformado de X num novo espaço após o ajustamento do modelo.

3.2 Otimização/Identificação do número de clusters k

A identificação do *cluster* insere a seguinte informação num algoritmo: "Aqui estão alguns dados. Agora agrupa coisas semelhantes e carateriza esses *clusters* resultantes."

Tal como uma classificação mas, neste caso, automática. A principal diferença em relação ao *clustering* é que na classificação sabe-se o que se procura.

Ainda assim, relativamente ao algoritmo `KMeans`, é necessário atribuir um valor ao parâmetro `n_clusters`, do tipo `int`, que representa o número de agrupamento ou clusters k.

Essa atribuição, sendo o valor de k conhecido ou sendo evidente, de acordo com o contexto, não representa qualquer problema.

Caso contrário, este valor pode ter que ser estimado mediante diversas ferramentas disponíveis para o efeito.

Ver Figura 3.2, onde se estima o valor de k usando a ferramenta `silhouette_score` ou ainda otimizando as medidas/índices `KMeans().score` ou `KMeans.inertia_`, relativamente a uma sequência de modelos construídos a partir de diferentes valores do parâmetro `n_clusters`.

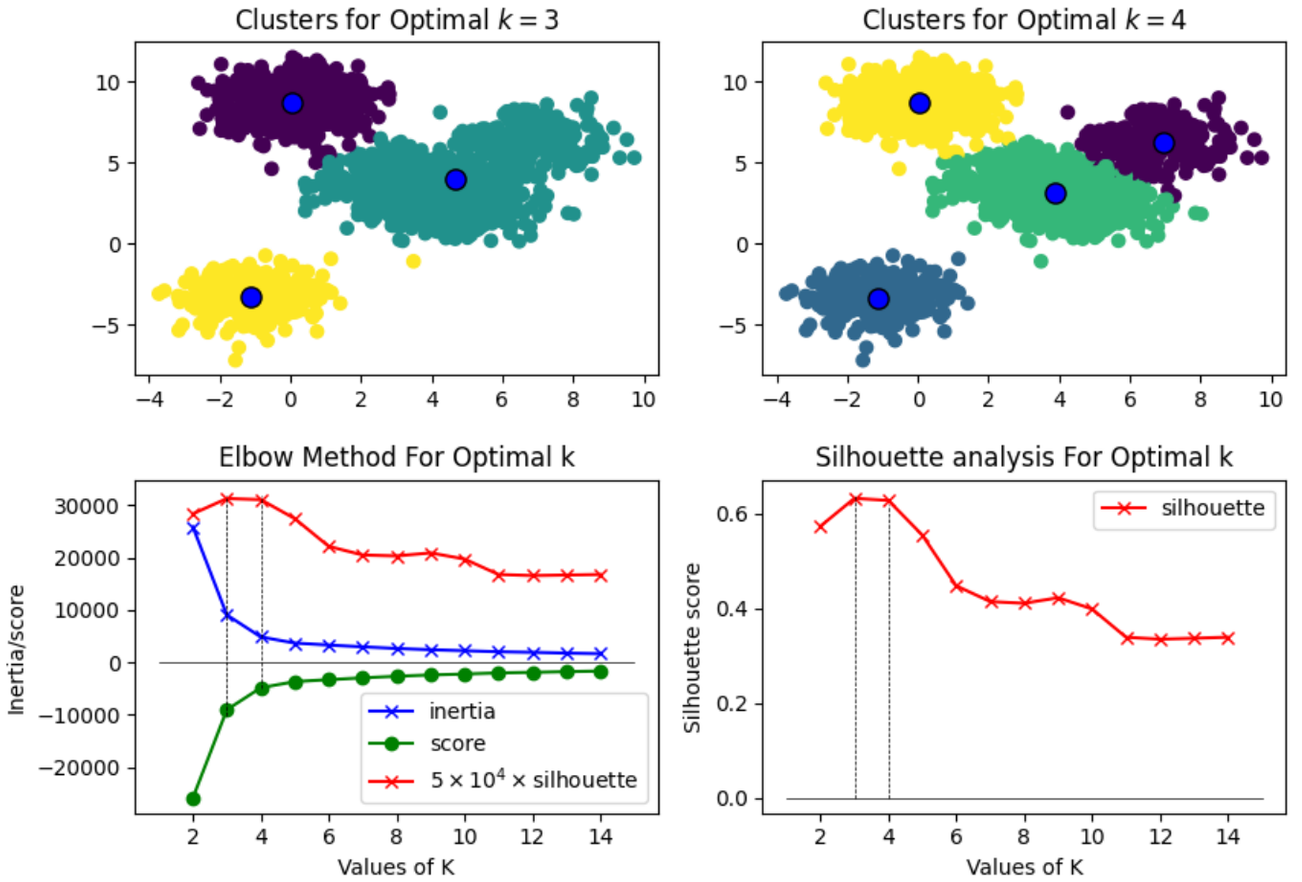

Figura 3.2: Estimativa do valor de k, usando `silhouette_score`, `KMeans().score` ou `KMeans.inertia_`

No caso da curva `silhouette` o valor de k ótimo corresponde sensivelmente ao ponto onde se atinge o valor máximo (aproximadamente $k = 3$ ou $k = 4$).
No caso das medidas `KMeans().score` e `KMeans.inertia_` o valor de k ótimo corresponde ao ponto onde, após uma redução mais acentuada em termos absolutos, se estabiliza o valor desses parâmetros (aproximadamente $k = 3$ ou $k = 4$).
Apresenta-se a seguir um possível código Python de construção da Figura 3.2

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
n_samples = 2000
random_state = 190
X, y = make_blobs(n_samples=n_samples,centers=Kk, random_state=random_state)
```

```
Sum_of_squared_distances = []
Kk = range(2,15)
y_score1 = []
silhouette_avg = []

for num_clusters in Kk :
    kmeans = KMeans(n_clusters=num_clusters)
    kmeans.fit(X)
    y_score = kmeans.score(X)
    y_score1 = np.append(y_score1,y_score)
    Sum_of_squared_distances.append(kmeans.inertia_)
    cluster_labels = kmeans.labels_
    if num_clusters >= 2 :
        silhouette_avg.append(silhouette_score(X, cluster_labels))


plt.figure(figsize=(10, 8))

K = Kk[np.argmax(silhouette_avg)]

kmeans = KMeans(init="k-means++", n_clusters=K, n_init=10)
y_pred = kmeans.fit_predict(X)
centroids = kmeans.cluster_centers_


plt.subplot(221)
plt.scatter(X[:, 0], X[:, 1], c=y_pred)
plt.scatter(centroids[:, 0], centroids[:, 1], marker="o", s=90,
    linewidths=1, color="b", edgecolor="black")
plt.title("Clusters for Optimal $k=%i$" % K)

K = K+1
kmeans = KMeans(init="k-means++", n_clusters=K, n_init=10)
y_pred = kmeans.fit_predict(X)
centroids = kmeans.cluster_centers_


plt.subplot(222)
plt.scatter(X[:, 0], X[:, 1], c=y_pred)
plt.scatter(centroids[:, 0], centroids[:, 1], marker="o", s=90,
    linewidths=1, color="b", edgecolor="black")
plt.title("Clusters for Optimal $k=%i$" % K)

m = (abs(max(y_score1))/abs(min(silhouette_avg)))*10
sill=np.dot(m,silhouette_avg)
```

```
plt.subplot(223)
plt.plot(Kk,Sum_of_squared_distances,'bx-',label="inertia")
plt.plot(Kk,y_score1, 'go-',label="score")#,'bx-')
plt.plot(Kk, sill,'rx-',label='$5\\times10^4\\times$'"silhouette")
xx = [3,3]
xx1 = [4,4]
yy = [-10200,32000]
yy1 = [-5000,32000]
plt.plot(xx, yy, c='black', linestyle="--",linewidth=0.5)
plt.plot(xx1, yy1, c='black', linestyle="--",linewidth=0.5)
plt.plot([1,15], [0,0], c='black', linestyle="-",linewidth=0.5)
plt.xlabel('Values of K')
plt.ylabel('Inertia/score')
plt.title('Elbow Method For Optimal k')
plt.legend()

plt.subplot(224)
plt.plot(Kk, silhouette_avg,'rx-',label="silhouette")
plt.xlabel('Values of K')
plt.ylabel('Silhouette score')
plt.title('Silhouette analysis For Optimal k')
yy = [0,0.64]
yy1 = [0,0.62]
plt.plot(xx, yy, c='black', linestyle="--",linewidth=0.5)
plt.plot(xx1, yy1, c='black', linestyle="--",linewidth=0.5)
plt.plot([1,15], [0,0], c='black', linestyle="-",linewidth=0.5)
plt.legend()
plt.show()
```

Em alternativa, pode optar-se por outro tipo de algoritmo que não necessite do input do número de Clusters.
A esse título, ver a página do *scikit-learn* 2.3. Clustering ou os algoritmos,
– Mean-shift
– Affinity propagation

O agrupamento é por vezes chamado de classificação não supervisionada porque produz o mesmo resultado que a classificação, mas sem ter classes predefinidas.
Para perceber as restante categorias de aprendizagem de máquina, deve-se primeiro entender as Redes Neuronais Artificiais (RNA, ou sigla em inglês, ANN), que se descrevem no próximo capítulo.

Laboratório 3