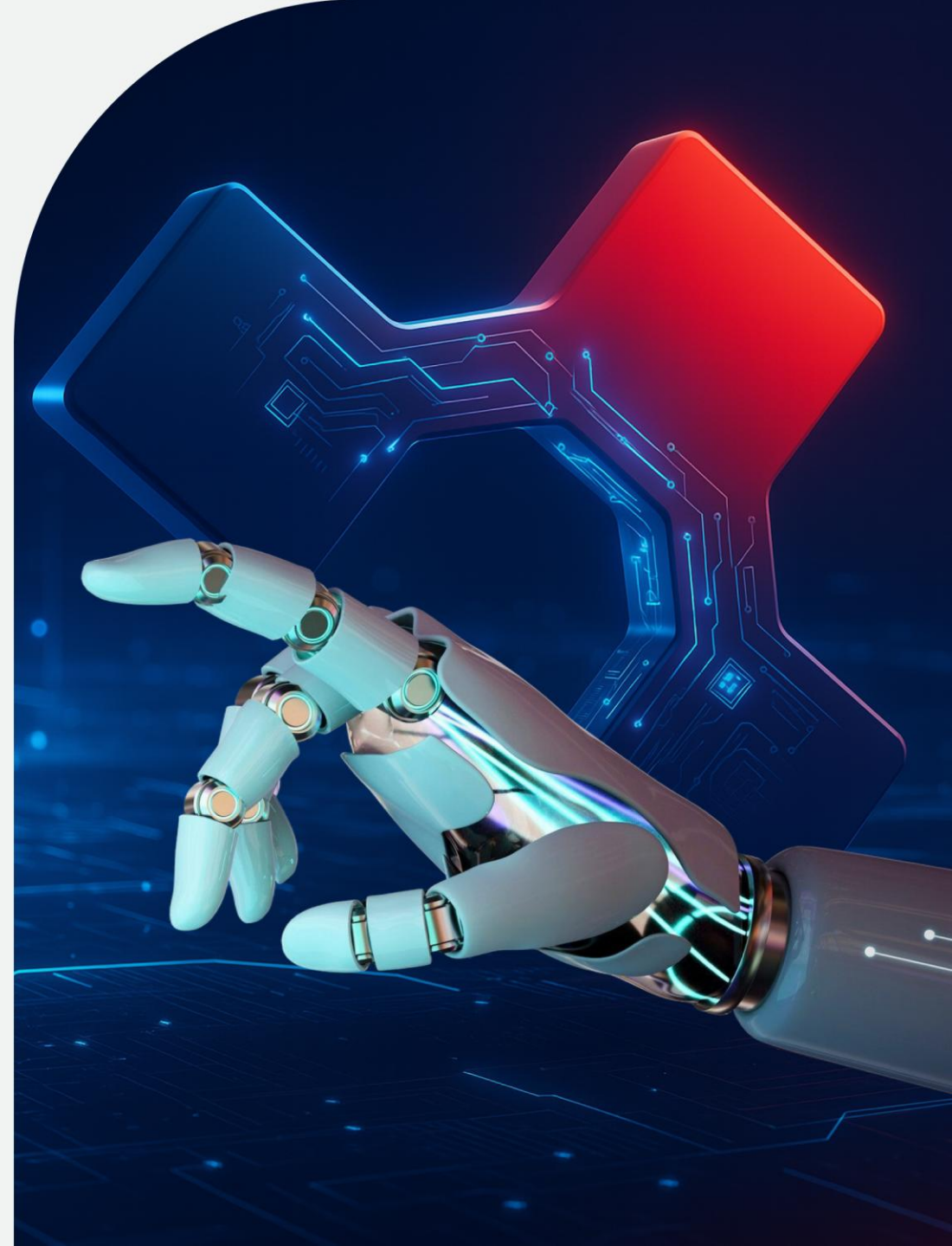


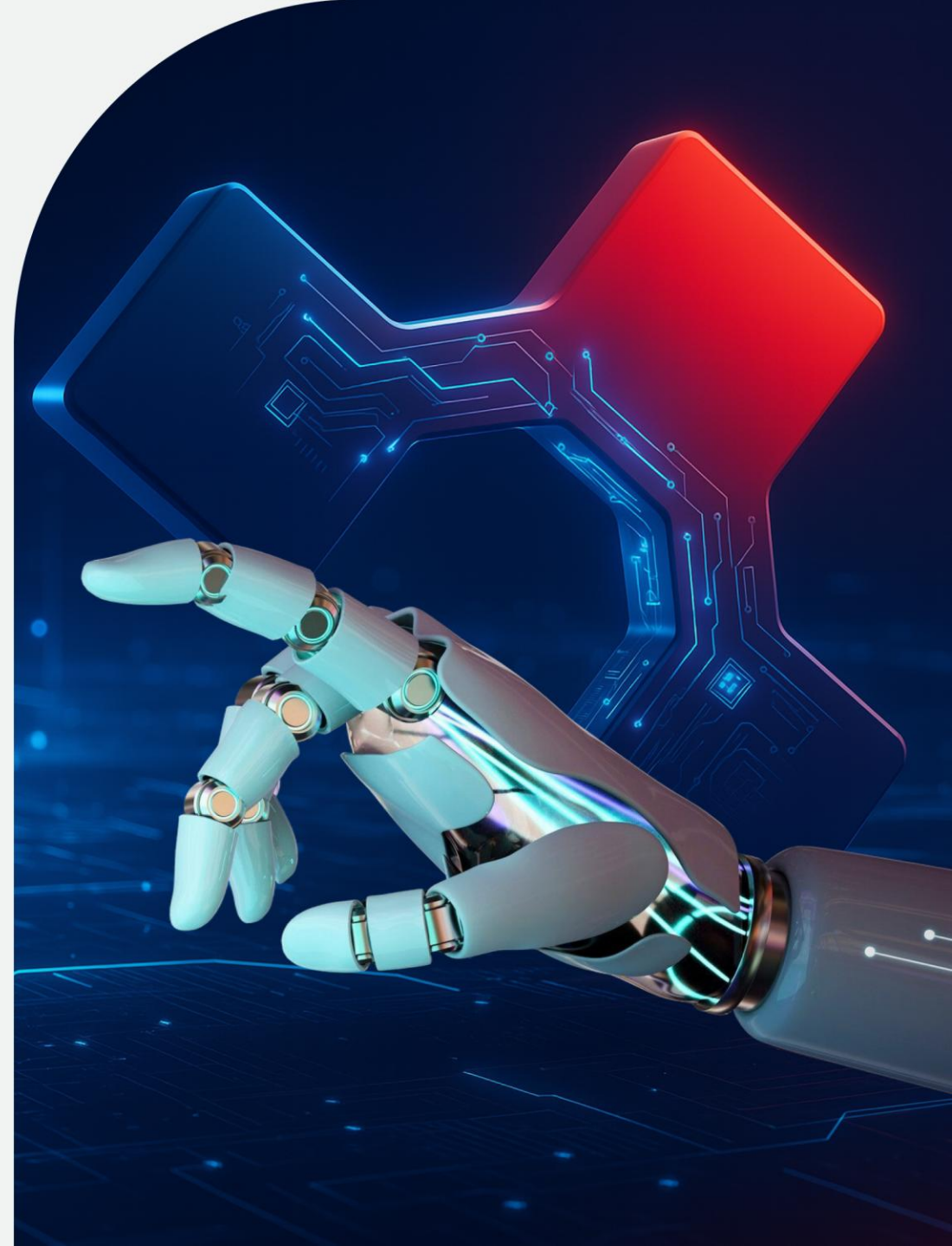


Núcleo de Capacitação em Inteligência Artificial



Unsupervised Learning Techniques

Clustering Algorithms: k-means and DBSCAN; k-means; Limits of k-means; Using Clustering for Image Segmentation; Using Clustering for Semi-Supervised Learning; DBSCAN; Other Clustering Algorithms; Gaussian Mixtures; Using Gaussian Mixtures for Anomaly Detection; Selecting the Number of Clusters; Bayesian Gaussian Mixture Models; Other Algorithms for Anomaly and Novelty Detection;





A maior parte dos dados disponíveis no mundo real **não possui rótulos**, apenas variáveis de entrada. Enquanto o aprendizado supervisionado depende de dados rotulados, o aprendizado não supervisionado consegue **aproveitar informações sem a necessidade de rótulos**. **Yann LeCun** comparou esse cenário dizendo que:

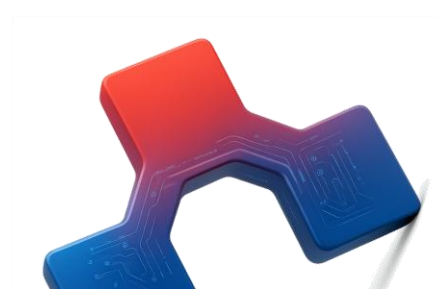
Se a inteligência fosse um bolo, o aprendizado não supervisionado seria o bolo em si, o aprendizado supervisionado seria a cobertura e o aprendizado por reforço a cereja no topo





Imagine uma fábrica que gera milhares de imagens de produtos todos os dias. Para treinar um classificador supervisionado, seria necessário rotular manualmente cada imagem como “defeituosa” ou “normal”, o que é **caro, demorado e cansativo**. Além disso, se o produto mudar, todo o processo de rotulagem teria que **ser refeito do zero**. O aprendizado não supervisionado surge como solução, permitindo que o modelo aproveite os dados **sem depender de rótulos humanos**.

Entre as tarefas mais importantes do aprendizado não supervisionado estão o **agrupamento de instâncias semelhantes em clusters**, a **detecção de anomalias** ou outliers para identificar padrões fora do normal, e a **estimativa de densidade**, que busca compreender a distribuição dos dados. Essas técnicas são usadas em áreas como análise de dados, recomendação de produtos, identificação de fraudes e muito mais.



Iris Flower Dataset

Data Card

Code (1488)

Discussion (2)

Suggestions (0)

The Iris flower data set is a multivariate data set introduced by the British statistician and biologist Ronald Fisher in his 1936 paper The use of multiple measurements in taxonomic problems. It is sometimes called Anderson's Iris data set because Edgar Anderson collected the data to quantify the morphologic variation of Iris flowers of three related species. The data set consists of 50 samples from each of three species of Iris (Iris Setosa, Iris virginica, and Iris versicolor). Four features were measured from each sample: the length and the width of the sepals and petals, in centimeters.

This dataset became a typical test case for many statistical classification techniques in machine learning such as support vector machines

Content

The dataset contains a set of 150 records under 5 attributes - Petal Length, Petal Width, Sepal Length, Sepal width and Class(Species).

Acknowledgements

This dataset is free and is publicly available at the UCI Machine Learning Repository

IRIS.csv (4.62 kB)

Detail

Compact

Column

5 of 5 columns

About this file

Suggest Edits

The dataset is a CSV file which contains a set of 150 records under 5 attributes - Petal Length, Petal Width, Sepal Length, Sepal width and Class(Species)

sepal_length

sepal_width

petal_length

petal_width

species

4.3

5.1

NCIA

7.9

3.5

FOXCONN

5.15 - 5.72

Count: 18

6.9

1.4

PF tech

0.1

2.5

0.2

3

unique values

Iris-setosa

IRIS.csv > data

```
1  sepal_length,sepal_width,petal_length,petal_width,species
2  5.1,3.5,1.4,0.2,Iris-setosa
3  4.9,3,1.4,0.2,Iris-setosa
4  4.7,3.2,1.3,0.2,Iris-setosa
5  4.6,3.1,1.5,0.2,Iris-setosa
6  5,3.6,1.4,0.2,Iris-setosa
7  5.4,3.9,1.7,0.4,Iris-setosa
8  4.6,3.4,1.4,0.3,Iris-setosa
9  5,3.4,1.5,0.2,Iris-setosa
10 4.4,2.9,1.4,0.2,Iris-setosa
11 4.9,3.1,1.5,0.1,Iris-setosa
12 5.4,3.7,1.5,0.2,Iris-setosa
13 4.8,3.4,1.6,0.2,Iris-setosa
14 4.8,3,1.4,0.1,Iris-setosa
15 4.3,3,1.1,0.1,Iris-setosa
16 5.8,4,1.2,0.2,Iris-setosa
17 5.7,4.4,1.5,0.4,Iris-setosa
18 5.4,3.9,1.3,0.4,Iris-setosa
19 5.1,3.5,1.4,0.3,Iris-setosa
20 5.7,3.8,1.7,0.3,Iris-setosa
21 5.1,3.8,1.5,0.3,Iris-setosa
22 5.4,3.4,1.7,0.2,Iris-setosa
```

Iris Virgínica



Íris Setosa

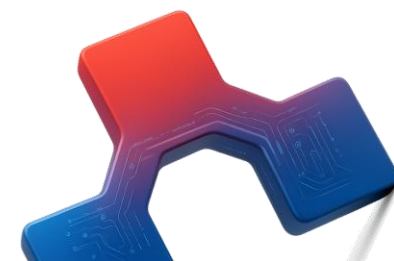
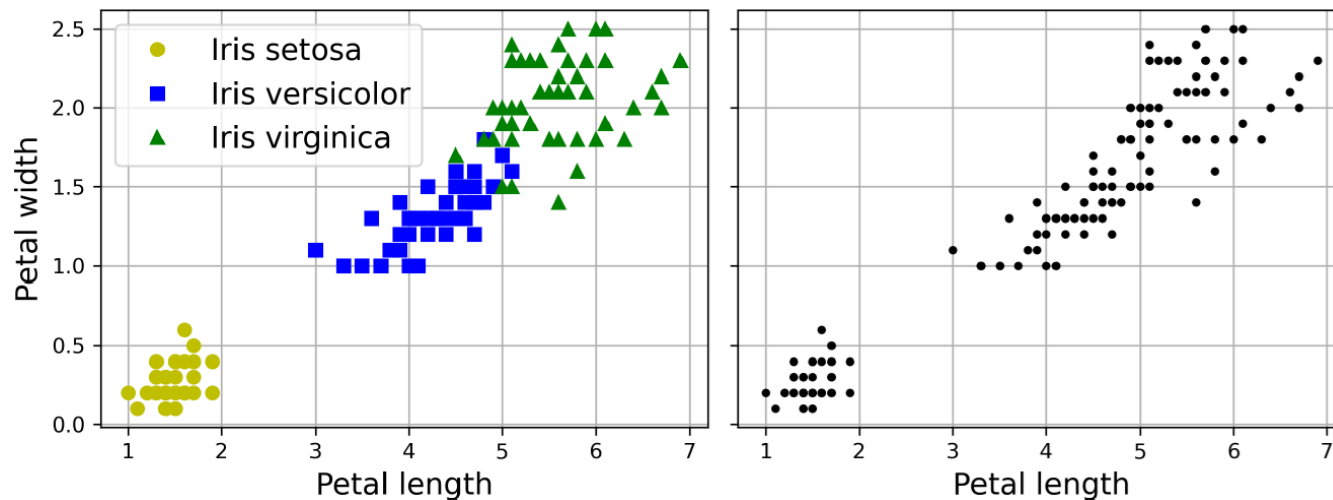


Íris Versicolor



1. Clustering Algorithms: k-means and DBSCAN

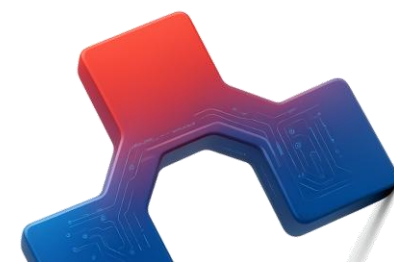
Na abaixo vemos a diferença entre classificação e clustering usando o famoso conjunto de dados das **flores íris**. No gráfico à esquerda temos as espécies rotuladas, como em um problema supervisionado. No gráfico à direita temos os mesmos dados sem rótulos, e algoritmos de clustering conseguem identificar grupos: o cluster inferior esquerdo aparece claramente, enquanto o superior direito contém dois subgrupos que não são tão óbvios. Considerando todas as variáveis do conjunto, algoritmos conseguem identificar os três grupos principais com bastante precisão.





O clustering tem aplicações práticas em diversas áreas. Ele pode ser usado para segmentar clientes e personalizar marketing, analisar novos conjuntos de dados explorando clusters separadamente, reduzir a dimensionalidade criando vetores de afinidade, enriquecer modelos com novas variáveis, detectar anomalias em dados, propagar rótulos em aprendizado semi-supervisionado, organizar imagens semelhantes em mecanismos de busca e até segmentar pixels de uma imagem para facilitar o reconhecimento de objetos.

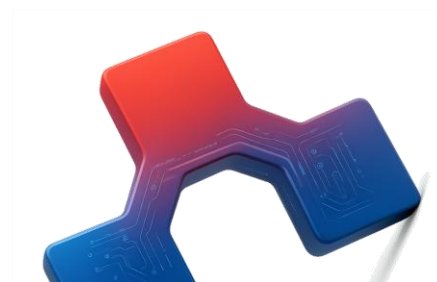
Não existe uma definição universal de cluster, e diferentes algoritmos capturam diferentes padrões. Alguns buscam grupos em torno de pontos centrais, chamados centroides. Outros procuram regiões densas de pontos, permitindo clusters de formas variadas. Existem ainda os métodos hierárquicos, que criam agrupamentos de clusters em diferentes níveis. Essa diversidade de técnicas amplia o leque de aplicações do clustering no aprendizado não supervisionado





1.1 K-Means

O algoritmo K-Means é uma técnica simples e eficiente de **agrupamento**, muito útil quando temos **dados** que formam **grupos bem definidos**. No exemplo abaixo, vemos cinco aglomerados de instâncias. O K-Means busca encontrar o **centro de cada grupo**, chamado de **centróide**, e **atribui cada instância ao centróide mais próximo**. Esse algoritmo foi proposto em 1957 por **Stuart Lloyd**, mas só foi publicado em 1982, e também é conhecido como algoritmo **Lloyd–Forgy**.



```

from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

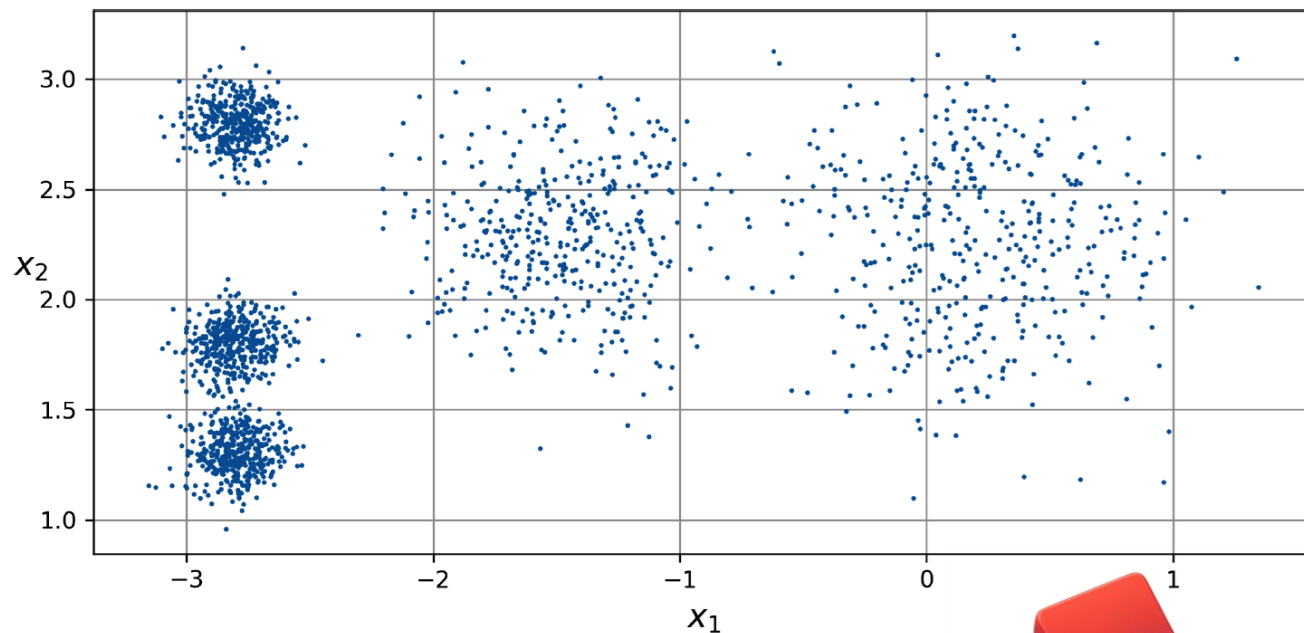
# extra code - the exact arguments of make_blobs() are not important
blob_centers = np.array([[ 0.2,  2.3], [-1.5,  2.3], [-2.8,  1.8],
                           [-2.8,  2.8], [-2.8,  1.3]])
blob_std = np.array([0.4, 0.3, 0.1, 0.1, 0.1])
X, y = make_blobs(n_samples=2000, centers=blob_centers, cluster_std=blob_std,
                  random_state=7)
k = 5
kmeans = KMeans(n_clusters=k, random_state=43)
y_pred = kmeans.fit_predict(X)

def plot_clusters(X, y=None):
    plt.scatter(X[:, 0], X[:, 1], c=y, s=1)
    plt.xlabel("$x_1$")
    plt.ylabel("$x_2$", rotation=0)

plt.figure(figsize=(8, 4))
plot_clusters(X)
plt.gca().set_axisbelow(True)
plt.grid()

plt.show()

```





Para usar o K-Means é necessário informar o **número de clusters**, representado por **k**. O algoritmo então **atribui cada instância ao cluster mais próximo**, representado por um centróide. Diferente da classificação, onde os rótulos são conhecidos, no clustering os **rótulos são índices de clusters gerados pelo algoritmo**. O modelo também armazena os centróides encontrados, permitindo prever a qual cluster novas instâncias pertencerão.

O resultado do K-Means pode ser representado graficamente, dividindo o espaço em regiões chamadas **tesselações de Voronoi**. Cada centróide é marcado com um "X" e cada instância pertence ao centróide mais próximo. Embora a maior parte das instâncias seja agrupada corretamente, algumas próximas das fronteiras podem ser atribuídas incorretamente, já que o algoritmo considera apenas a distância ao centróide.

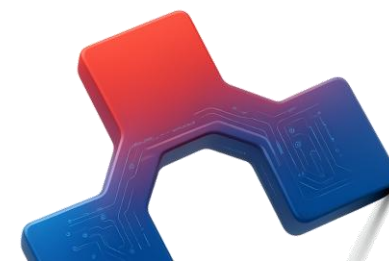
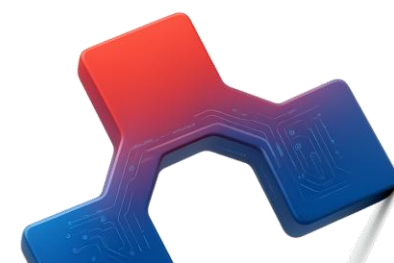
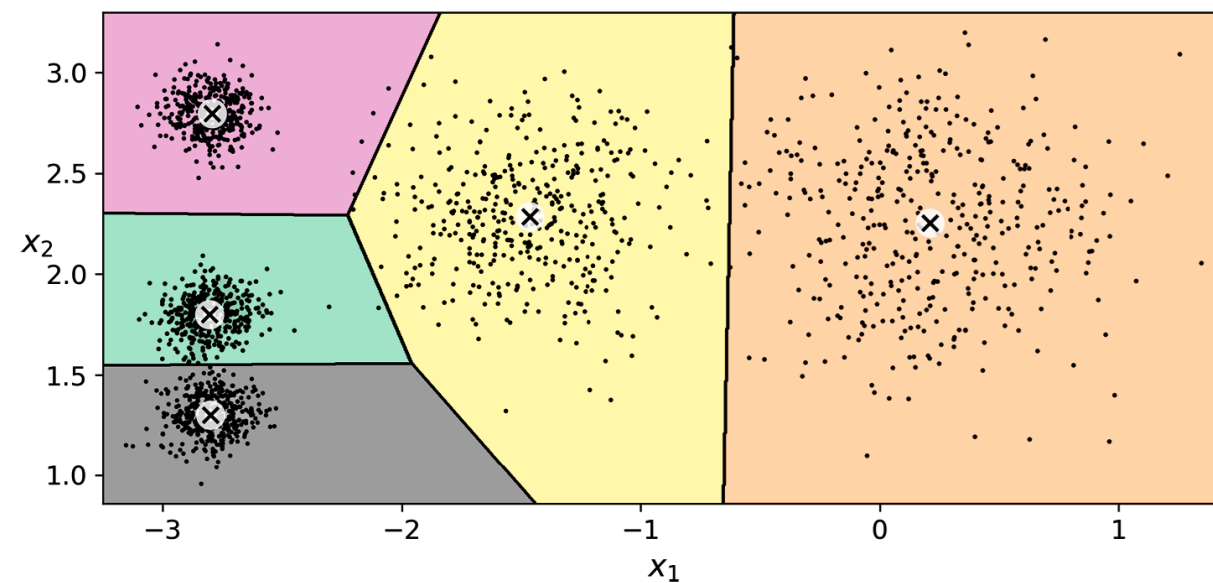
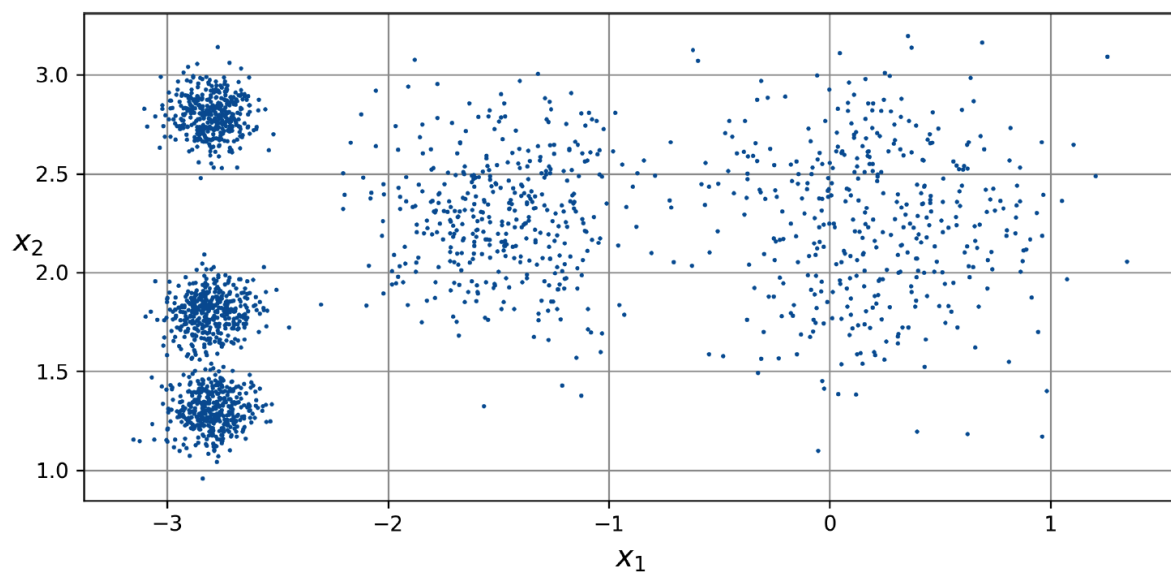





Diagrama de Voronoi





O K-Means tradicional realiza **hard clustering**, em que cada instância pertence a **apenas um cluster**. Em muitos casos, pode ser útil calcular a afinidade ou a distância de uma instância a **todos os centróides**, o que chamamos de **soft clustering**. Essa abordagem permite capturar melhor a **incerteza** e até **gerar novas variáveis** a partir dessas distâncias, observe o método `transform()`:

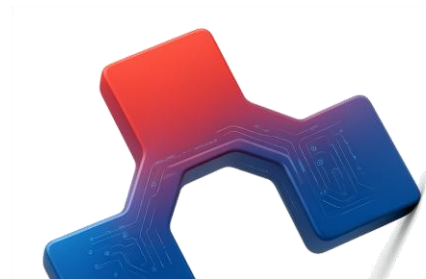
```
print(kmeans.transform(X_new).round(2))
```

```
Output: array([[2.81, 0.33, 2.9 , 1.49, 2.89],
               [5.81, 2.8 , 5.85, 4.48, 5.84],
               [1.21, 3.29, 0.29, 1.69, 1.71],
               [0.73, 3.22, 0.36, 1.55, 1.22]])
```

Você pode verificar que esta é de fato a **distância euclidiana** entre cada instância e cada centróide

```
np.linalg.norm(np.tile(X_new, (1, k)).reshape(-1, k, 2)
               - kmeans.cluster_centers_, axis=2).round(2)
```

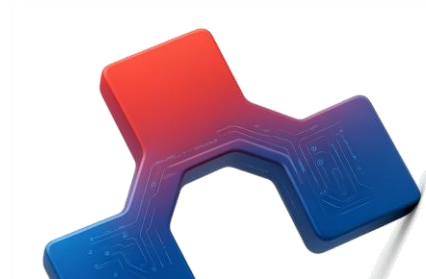
```
array([[2.81, 0.33, 2.9 , 1.49, 2.89],
       [5.81, 2.8 , 5.85, 4.48, 5.84],
       [1.21, 3.29, 0.29, 1.69, 1.71],
       [0.73, 3.22, 0.36, 1.55, 1.22]])
```

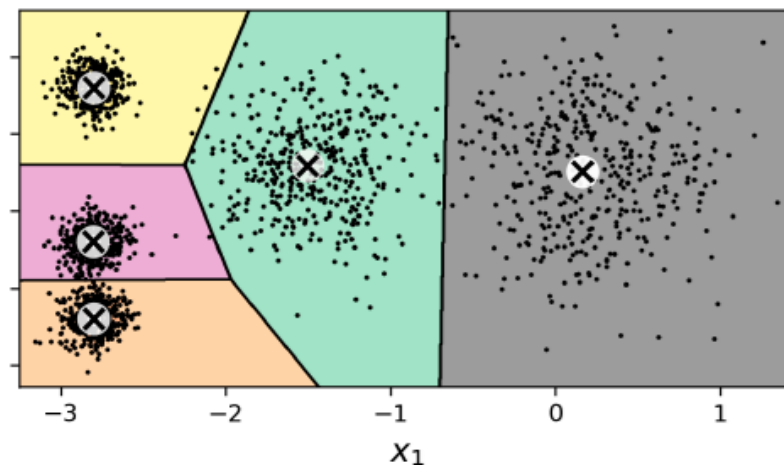
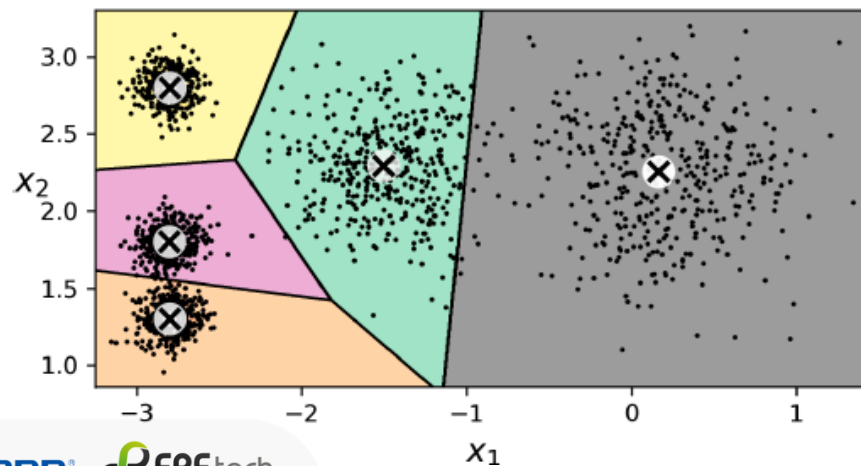
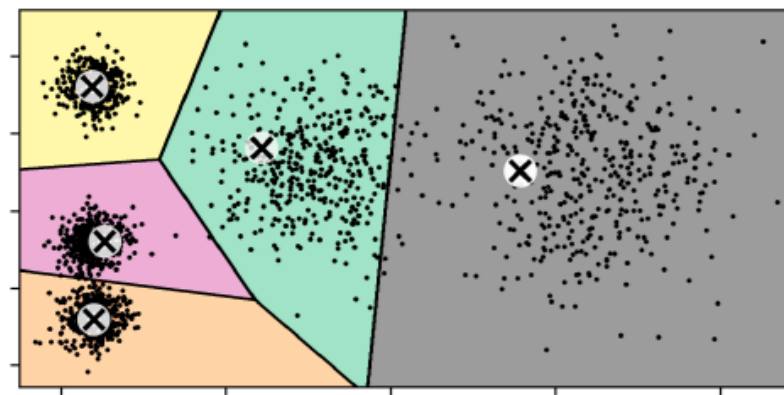
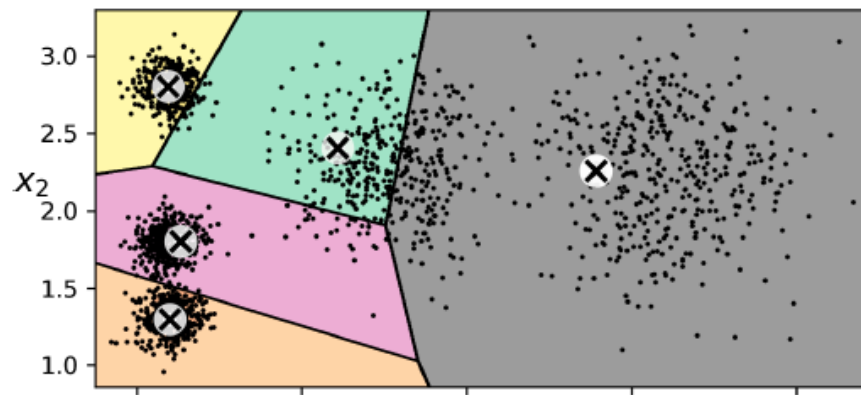
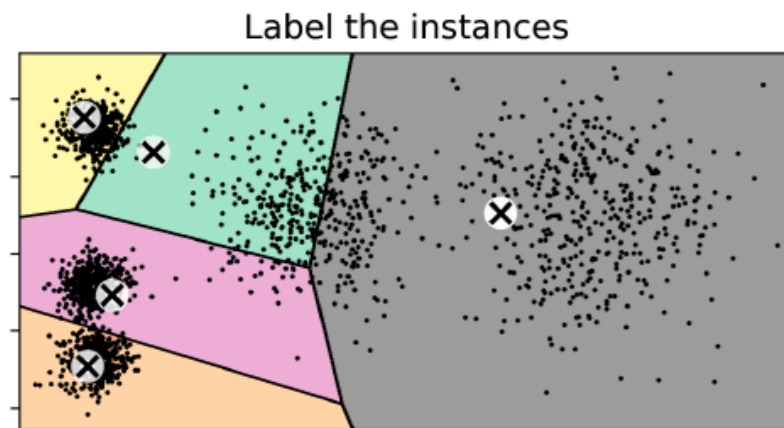
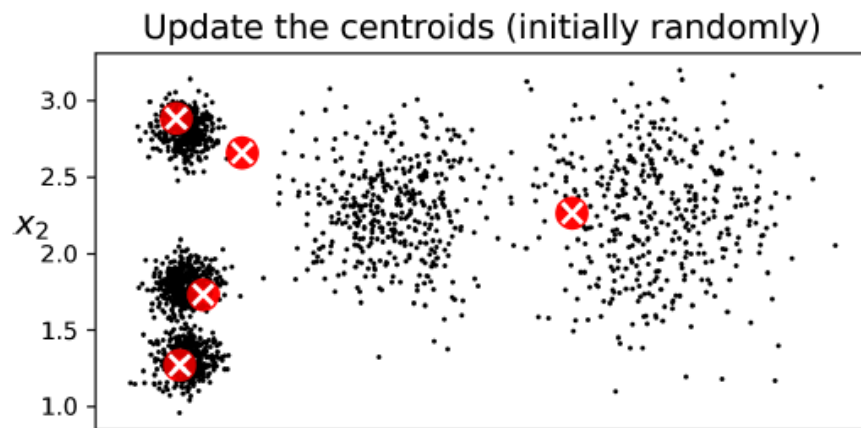




1.1.1 The k-means algorithm

O funcionamento do K-Means é **iterativo**. Inicialmente, os centróides são escolhidos aleatoriamente. Em seguida, cada instância é atribuída ao centróide mais próximo. Os centróides são recalculados como a média das instâncias atribuídas. Esse processo de atualização e rotulação é repetido até que os centróides se estabilizem. O algoritmo **sempre converge em poucas épocas**, pois a soma das distâncias quadráticas só pode diminuir.





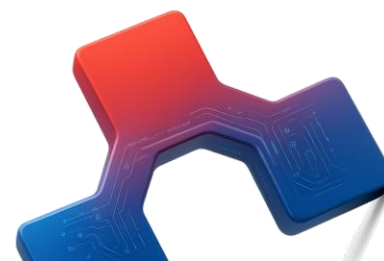
NCIA



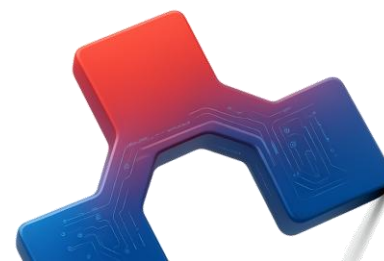
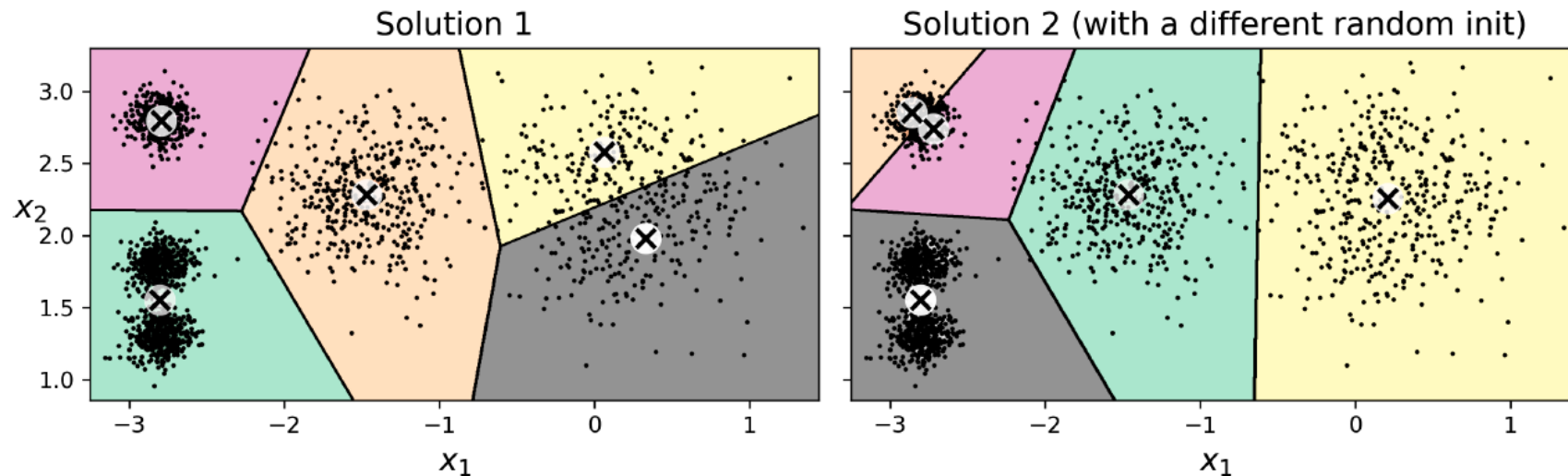
FOXCONN



PPFtech



Embora o algoritmo sempre convirja, ele pode parar em soluções **subótimas** se os centroides forem **inicializados em posições ruins**. Isso pode levar a agrupamentos que **não refletem bem a estrutura real dos dados**. Na abaixo, vemos exemplos de inicializações ruins que resultaram em soluções insatisfatórias.



1.1.2 Centroid initialization methods

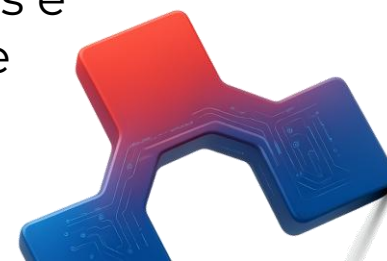
Para melhorar a qualidade do resultado, o K-Means pode ser **inicializado manualmente** com centróides escolhidos pelo usuário, ou **executado várias vezes com diferentes inicializações aleatórias**. O Scikit-Learn faz isso automaticamente, repetindo o processo várias vezes (`n_init`) e escolhendo a solução com **menor inércia**, que é a soma das distâncias quadráticas entre instâncias e centróides.

```
good_init = np.array([[ -3,  3], [ -3,  2], [ -3,  1], [ -1,  2], [  0,  2]])
kmeans = KMeans(n_clusters=5, init=good_init, n_init=1, random_state=42)
kmeans.fit(X)

print(kmeans.inertia_)
print(kmeans.score(X)) # retorna a inércia negativa
```

Output: 211.59853725816836
-211.5985372581684

Uma melhoria chamada **K-Means++** propõe uma inicialização mais inteligente, **escolhendo centróides mais distantes entre si**. Isso reduz o risco de soluções ruins e aumenta a probabilidade de convergir rapidamente para o resultado ótimo. Esse método é utilizado por padrão na implementação do Scikit-Learn

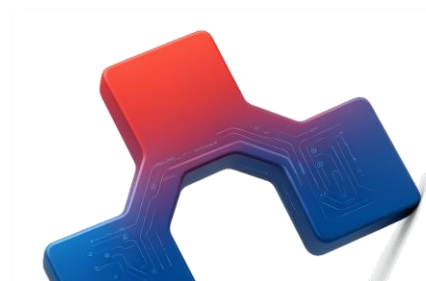
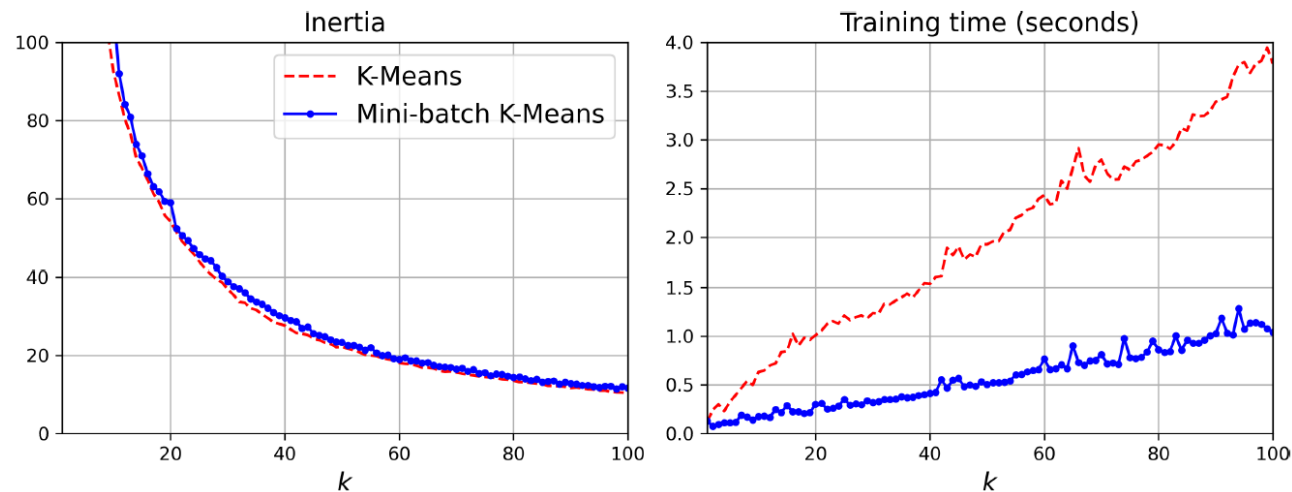


1.1.3 Accelerated k-means and mini-batch k-means

O Mini-Batch K-Means é uma variação projetada para grandes volumes de dados. Ele atualiza os centróides usando pequenos lotes de instâncias, tornando o processo até 3,5 vezes mais rápido. A desvantagem é que sua inércia tende a ser um pouco maior, ou seja, o resultado é ligeiramente menos preciso. Ainda assim, é uma excelente alternativa para datasets muito grandes.

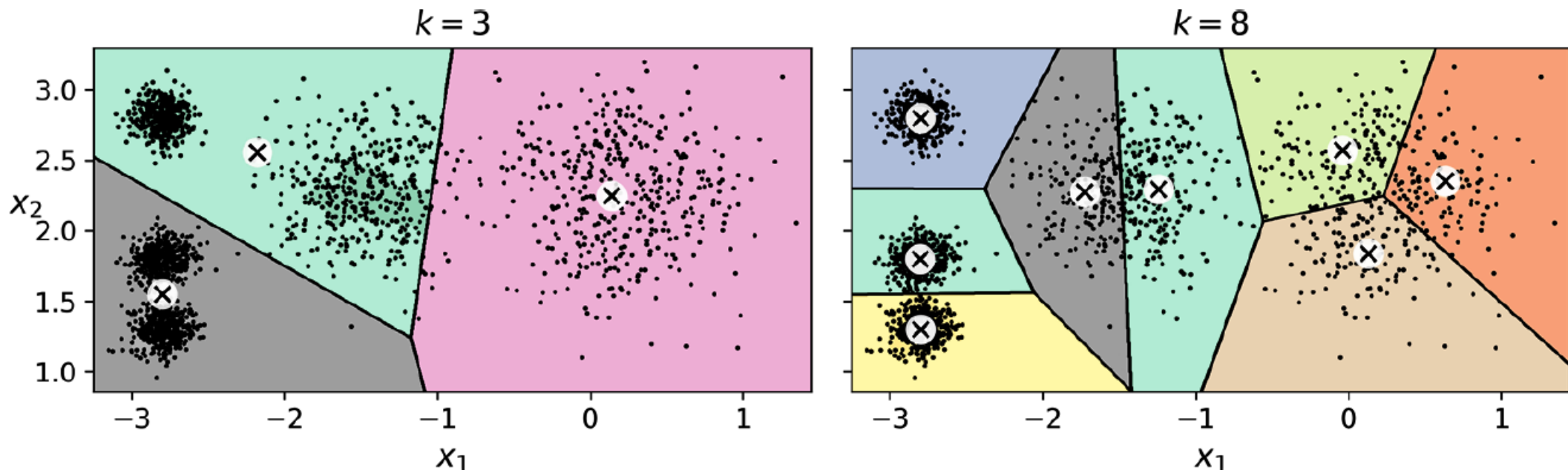
```
from sklearn.cluster import MiniBatchKMeans
```

```
minibatch_kmeans = MiniBatchKMeans(n_clusters=5, random_state=42)  
minibatch_kmeans.fit(X)
```

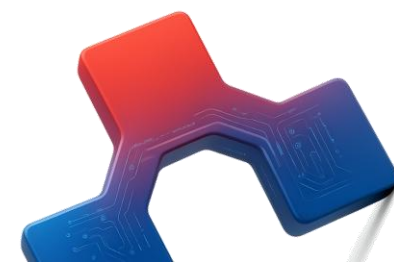
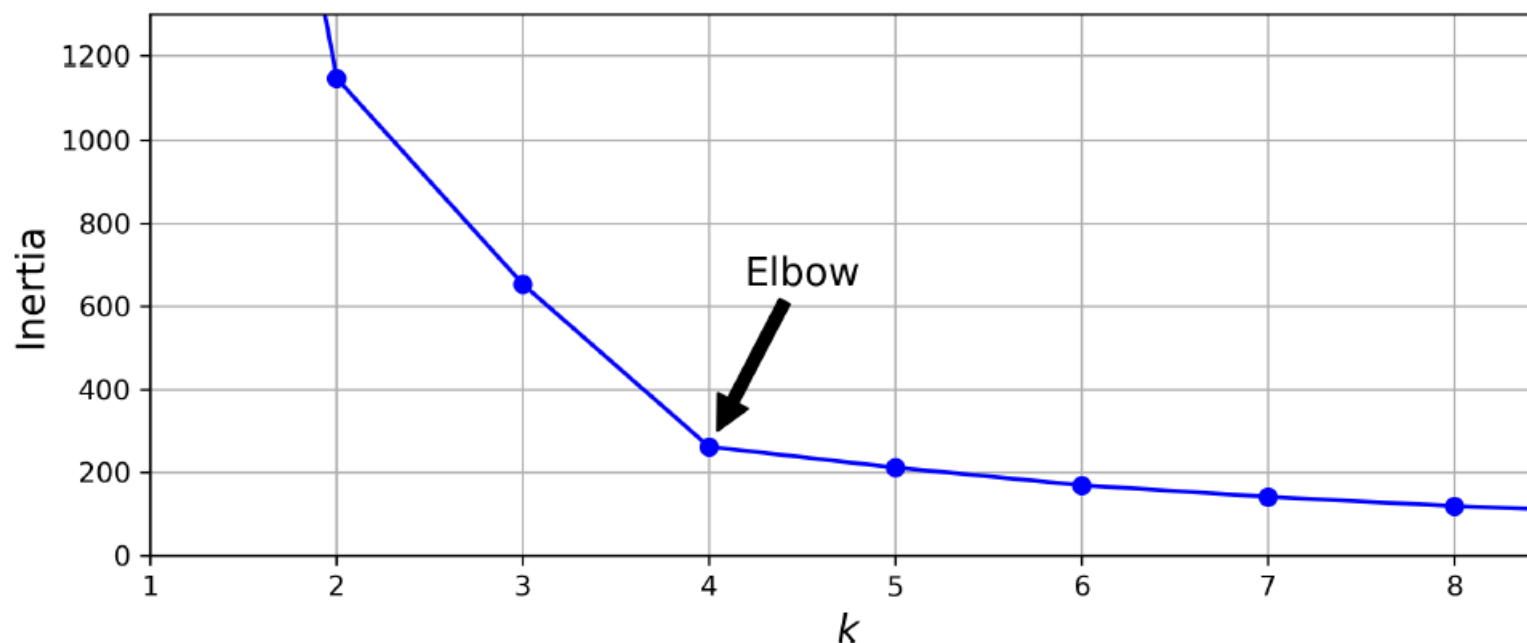


1.1.4 Finding the optimal number of clusters

Até agora escolhemos $k=5$ porque era fácil de visualizar no conjunto de dados. Porém, na prática, escolher o **número correto de clusters** não é trivial, e valores errados podem gerar resultados ruins. Como mostra a Figura abaixo, se k for **muito pequeno**, clusters diferentes acabam se misturando; se k for **muito grande**, clusters verdadeiros podem ser divididos à força em vários pedaços.



Um critério simples é analisar a **inércia**, que é a soma das distâncias quadráticas entre as instâncias e seus centróides. A inércia diminui conforme aumentamos k , mas isso não significa necessariamente uma melhor solução. O método do cotovelo consiste em traçar a curva da inércia e identificar o ponto de inflexão, o “cotovelo”. Na Figura 9-8, a curva cai rápido até $k=4$, e depois a queda é mais lenta. Esse ponto indica que $k=4$ pode ser uma boa escolha.





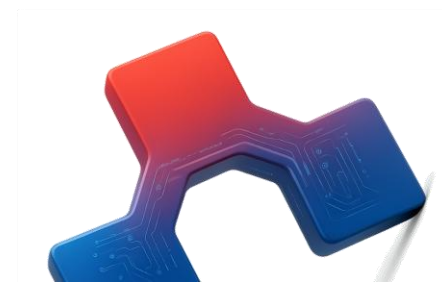
Apesar de útil, o método do cotovelo é apenas uma aproximação grosseira. Nem sempre o ponto de inflexão é claro, e clusters mal formados podem parecer aceitáveis pela curva de inércia. Para ter uma avaliação mais precisa da qualidade dos clusters, é necessário utilizar métricas adicionais, como o [silhouette score](#).

```
from sklearn.metrics import silhouette_score

silhouette_scores = [silhouette_score(X, model.labels_)
                      for model in kmeans_per_k[1:]]

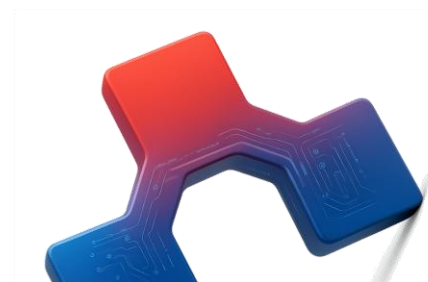
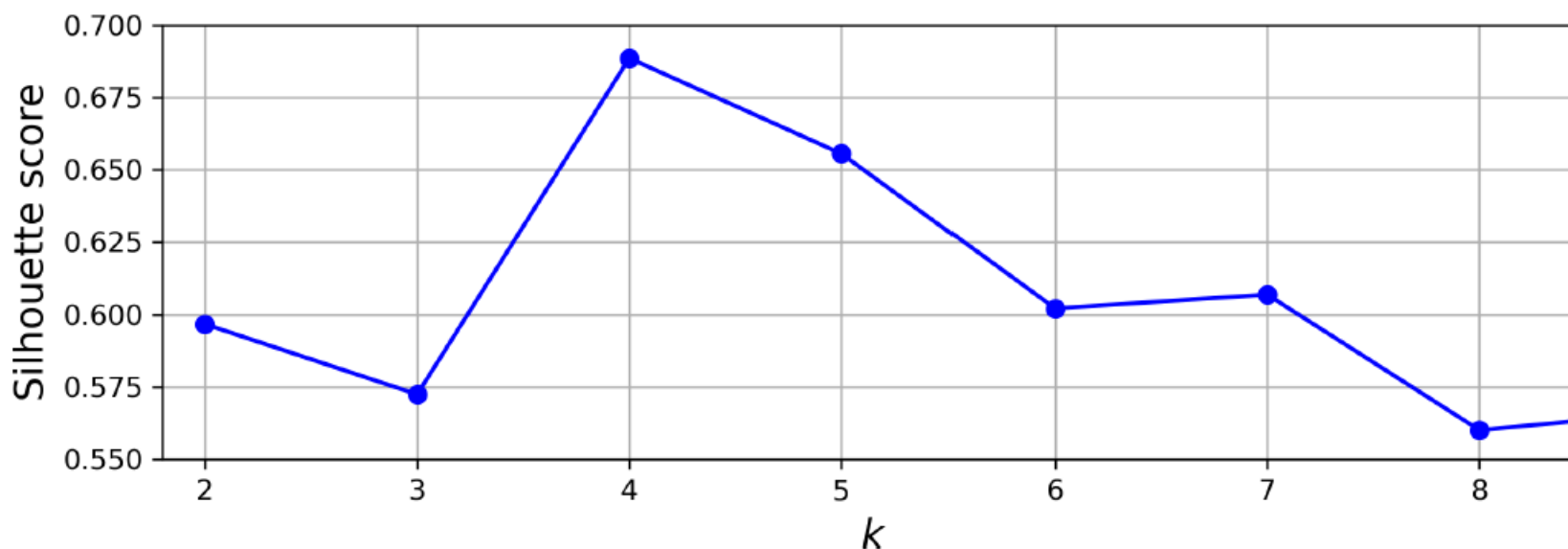
plt.figure(figsize=(8, 3))
plt.plot(range(2, 10), silhouette_scores, "bo-")
plt.xlabel("$k$")
plt.ylabel("Silhouette score")
plt.axis([1.8, 8.5, 0.55, 0.7])
plt.grid()

plt.show()
```



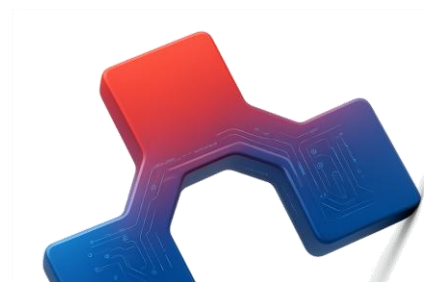


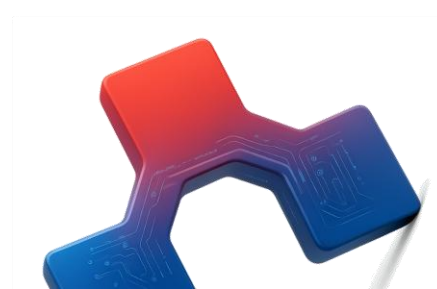
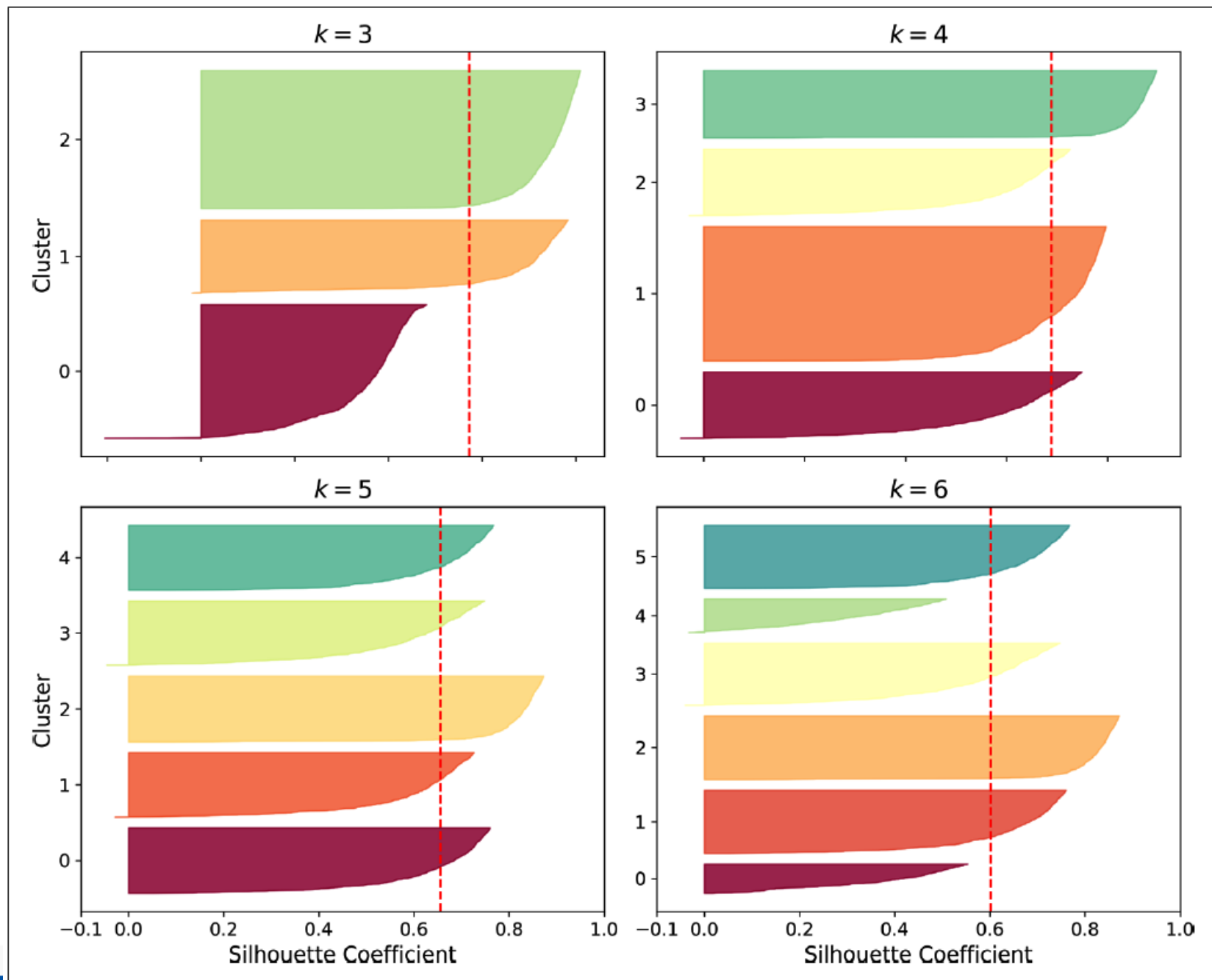
O coeficiente de silhouette mede o quão bem cada instância está posicionada no seu cluster em comparação com outros clusters. Ele varia de -1 a +1: valores próximos de +1 indicam instâncias bem agrupadas, próximos de 0 indicam proximidade a fronteiras, e negativos sugerem que a instância foi alocada no cluster errado. Na abaixo, vemos que $k=4$ e $k=5$ são boas escolhas, enquanto $k=6$ ou 7 produzem resultados ruins.





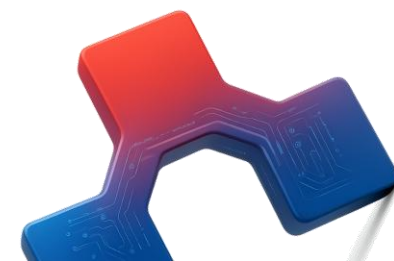
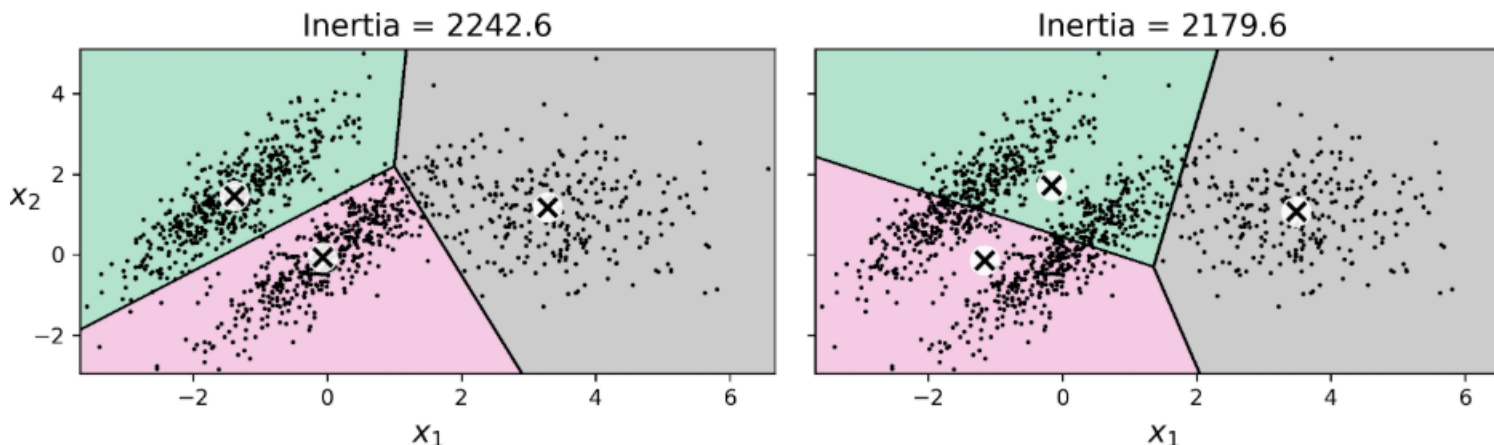
Uma forma ainda mais informativa é o [diagrama de silhouette](#), que mostra a distribuição dos coeficientes de todas as instâncias dentro de cada cluster. Cada cluster aparece como uma forma em “faca”, cujo [tamanho](#) representa o [número de instâncias](#) e cuja [largura](#) indica a [qualidade do agrupamento](#). Linhas tracejadas mostram a média do coeficiente. Quando muitos pontos ficam antes da linha, o cluster é ruim. A figura a seguir mostra que $k=3$ e $k=6$ geram clusters fracos, enquanto $k=4$ e $k=5$ apresentam agrupamentos bons. Entre eles, $k=5$ se destaca por produzir clusters de tamanhos mais equilibrados.





1.2 Limits of k-means

Apesar de ser rápido e escalável, o **K-Means tem limitações importantes**. O algoritmo precisa ser executado **várias vezes para evitar soluções ruins**, além de exigir a definição prévia do **número de clusters**, o que pode ser trabalhoso. Outro problema é que ele funciona mal quando os clusters têm **tamanhos diferentes, densidades distintas** ou **formas não esféricas**. Na a seguir, vemos três agrupamentos elipsoidais: o K-Means não consegue representá-los corretamente. No lado esquerdo, cerca de 25% de um cluster é cortado e atribuído incorretamente. No lado direito, a solução é ainda pior, embora apresente uma inércia menor. Isso mostra que a métrica de inércia nem sempre garante bons agrupamentos. Para dados desse tipo, **modelos de mistura Gaussiana funcionam melhor**.





1.3 Using Clustering for Image Segmentation

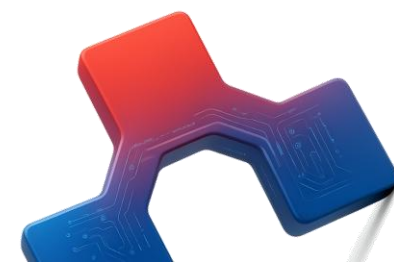
Segmentação de imagens é o processo de dividir uma imagem em múltiplas partes (segmentos). Existem diferentes abordagens:

Segmentação por cor: pixels com cores semelhantes são agrupados no mesmo segmento. Exemplo: medir área de floresta em imagens de satélite.

Segmentação semântica: pixels que pertencem a um mesmo tipo de objeto recebem o mesmo rótulo. Exemplo: todos os pixels de pedestres em uma cena de carro autônomo são agrupados como “pedestre”.

Segmentação por instância: cada objeto individual recebe um segmento separado. Exemplo: cada pedestre é identificado individualmente.

Modelos de ponta em segmentação semântica e por instância usam CNNs (Cap. 12) ou Transformers de visão (Cap. 16). Aqui, porém, focaremos em uma abordagem simples de segmentação por cor com K-Means.





Usaremos a biblioteca [Pillow \(PIL\)](#) para carregar uma imagem. A imagem é representada como um array 3D: altura \times largura \times canais de cor. Cada pixel possui três valores (R, G, B) entre 0 e 255.

```
import PIL

image = np.asarray(PIL.Image.open(filepath))
image.shape

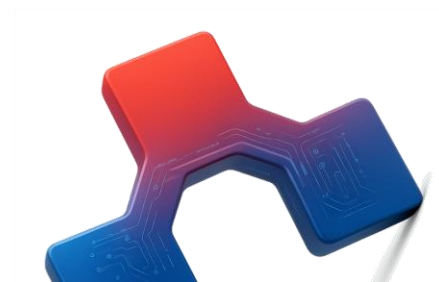
(533, 800, 3)
```

A primeira dimensão é a altura da imagem.

A segunda é a largura.

A terceira são os canais de cor (RGB).

Algumas imagens podem ter apenas 1 canal (tons de cinza) ou mais canais (como infravermelho em imagens de satélite).





O próximo passo é reformatar a **matriz de pixels** para uma **lista longa de cores RGB**, aplicar o K-Means para agrupar as cores em k clusters, e reconstruir a imagem substituindo cada pixel pela cor média do cluster a que pertence.

- O resultado é uma imagem onde as cores originais **são reduzidas para k cores médias**.
- Cada pixel **assume a cor de seu centróide**, criando uma versão simplificada e segmentada da imagem.

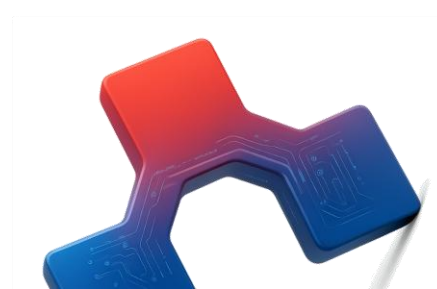
```
X = image.reshape(-1, 3)
kmeans = KMeans(n_clusters=8, random_state=42).fit(X)
segmented_img = kmeans.cluster_centers_[kmeans.labels_]
segmented_img = segmented_img.reshape(image.shape)

segmented_imgs = []
n_colors = (10, 8, 6, 4, 2)
for n_clusters in n_colors:
    kmeans = KMeans(n_clusters=n_clusters, random_state=42).fit(X)
    segmented_img = kmeans.cluster_centers_[kmeans.labels_]
    segmented_imgs.append(segmented_img.reshape(image.shape))

plt.figure(figsize=(10, 5))
plt.subplots_adjust(wspace=0.05, hspace=0.1)

plt.subplot(2, 3, 1)
plt.imshow(image)
plt.title("Original image")
plt.axis('off')

for idx, n_clusters in enumerate(n_colors):
    plt.subplot(2, 3, 2 + idx)
    plt.imshow(segmented_imgs[idx] / 255)
    plt.title(f"{n_clusters} colors")
    plt.axis('off')
plt.show()
```



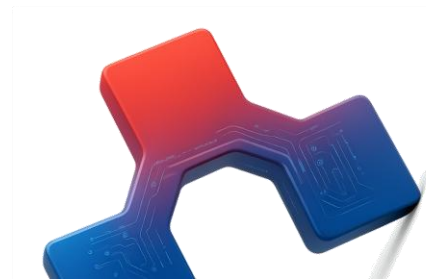


1.4 Using Clustering for Semi-Supervised Learning

Em muitos cenários temos muitos exemplos não rotulados e apenas alguns poucos rotulados. O aprendizado [semi-supervisionado](#) busca aproveitar os [dados não rotulados](#) para melhorar o desempenho. O exemplo usa o [conjunto digits do Scikit-Learn](#), que contém 1.797 imagens em tons de cinza de dígitos (0–9), cada uma com tamanho 8×8 pixels.

```
from sklearn.datasets import load_digits

X_digits, y_digits = load_digits(return_X_y=True)
X_train, y_train = X_digits[:1400], y_digits[:1400]
X_test, y_test = X_digits[1400:], y_digits[1400:]
```

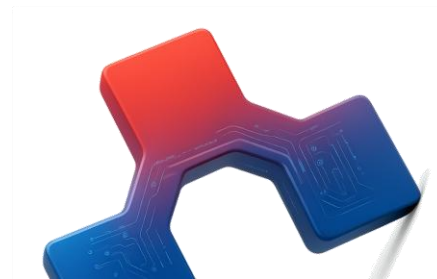




Vamos supor que temos apenas 50 instâncias rotuladas. Treinando uma regressão logística nesses dados, obtemos uma acurácia de apenas 74,8%, bem abaixo dos 90,7% alcançados quando usamos todo o conjunto de treino.

```
from sklearn.linear_model import LogisticRegression

n_labeled = 50
log_reg = LogisticRegression(max_iter=10_000)
log_reg.fit(X_train[:n_labeled], y_train[:n_labeled])
print(log_reg.score(X_test, y_test)) # 0.748
```



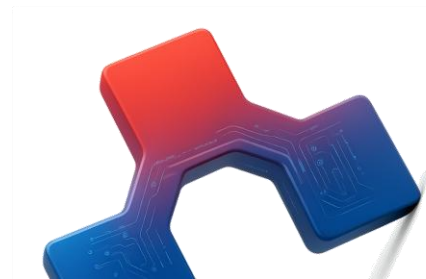
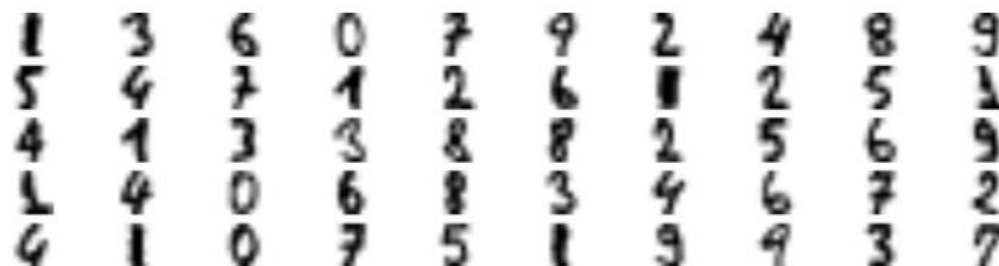


Para melhorar, aplicamos o **K-Means** com **50 clusters**. Seleccionamos as imagens mais próximas de cada centróide como representativas de seus grupos. Assim, em vez de rotular instâncias aleatórias, rotulamos manualmente apenas essas 50 imagens representativas.

```
from sklearn.cluster import KMeans
import numpy as np

k = 50
kmeans = KMeans(n_clusters=k, random_state=42)
X_digits_dist = kmeans.fit_transform(X_train)

representative_digit_idx = np.argmin(X_digits_dist, axis=0)
X_representative_digits = X_train[representative_digit_idx]
```

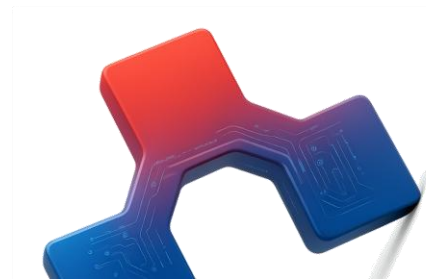




Após rotularmos as 50 imagens representativas, treinamos novamente o modelo. A acurácia subiu para 84,9%, mostrando que é muito melhor rotular exemplos representativos do que exemplos aleatórios.

```
y_representative_digits = np.array([1, 3, 6, 0, 7, 9, 2, ..., 5, 1, 9, 9, 3, 7])

log_reg = LogisticRegression(max_iter=10_000)
log_reg.fit(X_representative_digits, y_representative_digits)
print(log_reg.score(X_test, y_test)) # 0.849
```

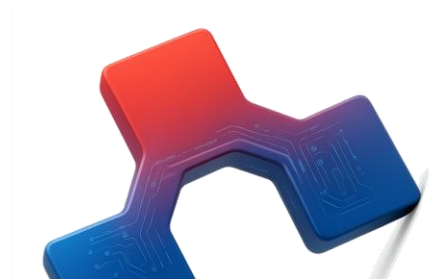




Podemos dar mais um passo: propagar o rótulo de cada instância representativa para todas as instâncias de seu cluster. Esse processo é chamado de **propagação de rótulos**. Com isso, o modelo atinge **89,4% de acurácia**.

```
y_train_propagated = np.empty(len(X_train), dtype=np.int64)
for i in range(k):
    y_train_propagated[kmeans.labels_ == i] = y_representative_digits[i]

log_reg = LogisticRegression()
log_reg.fit(X_train, y_train_propagated)
print(log_reg.score(X_test, y_test)) # 0.894
```





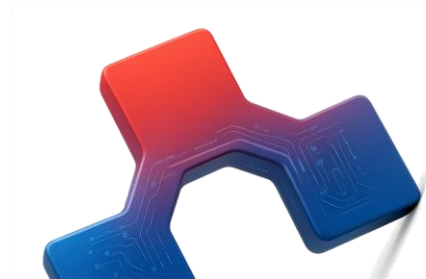
Para melhorar ainda mais, removemos 1% das instâncias mais distantes de seus centróides, que provavelmente são outliers. Com esse ajuste, a acurácia chegou a 90,9%, superando até o treinamento feito com todo o conjunto rotulado (90,7%).

```
percentile_closest = 99
X_cluster_dist = X_digits_dist[np.arange(len(X_train)), kmeans.labels_]

for i in range(k):
    in_cluster = (kmeans.labels_ == i)
    cluster_dist = X_cluster_dist[in_cluster]
    cutoff_distance = np.percentile(cluster_dist, percentile_closest)
    above_cutoff = (X_cluster_dist > cutoff_distance)
    X_cluster_dist[in_cluster & above_cutoff] = -1

partially_propagated = (X_cluster_dist != -1)
X_train_partially_propagated = X_train[partially_propagated]
y_train_partially_propagated = y_train_propagated[partially_propagated]

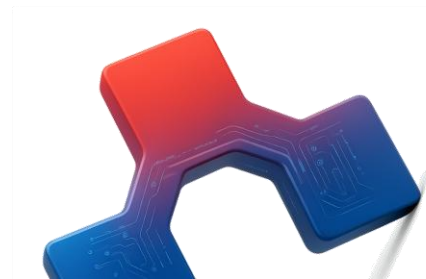
log_reg = LogisticRegression(max_iter=10_000)
log_reg.fit(X_train_partially_propagated, y_train_partially_propagated)
print(log_reg.score(X_test, y_test)) # 0.909
```





O Scikit-Learn oferece classes que fazem a propagação de rótulos automaticamente: [LabelPropagation](#) e [LabelSpreading](#). Outra técnica é o [SelfTrainingClassifier](#), que combina aprendizado supervisionado e auto-rotulagem iterativa. Essas estratégias não são soluções mágicas, mas podem trazer bons ganhos quando há poucos rótulos disponíveis.

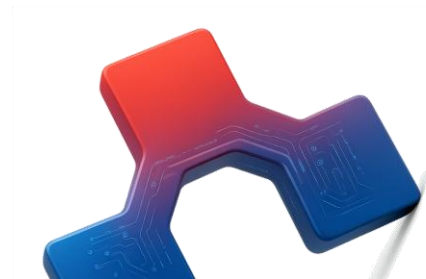
```
from sklearn.semi_supervised import LabelPropagation, SelfTrainingClassifier
```





1.5 DBSCAN

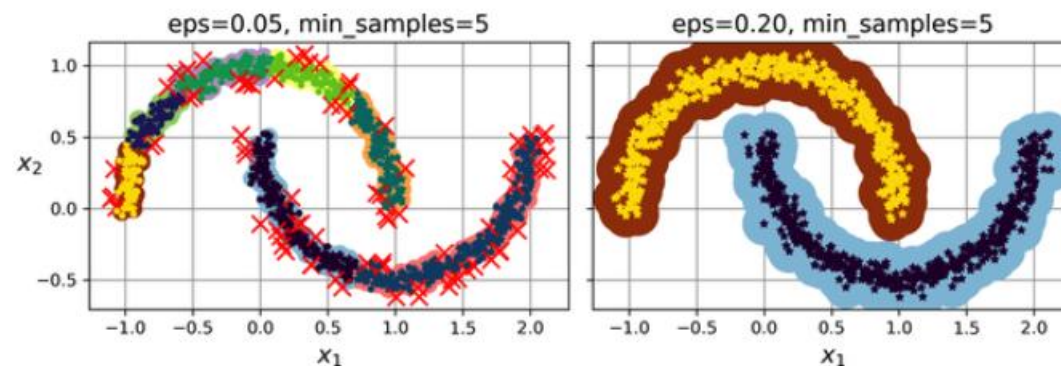
O algoritmo DBSCAN ([Density-Based Spatial Clustering of Applications with Noise](#)) define clusters como regiões contínuas de alta densidade. Para cada instância, ele verifica quantos vizinhos estão dentro de uma distância ϵ ([epsilon](#)). Se houver pelo menos [min_samples](#) vizinhos, a instância é considerada um [ponto central \(core\)](#). Todos os vizinhos de um ponto central fazem parte do mesmo cluster, e uma sequência de pontos centrais conectados forma um único cluster. Instâncias que não atendem a esse critério são consideradas [anomalias](#).



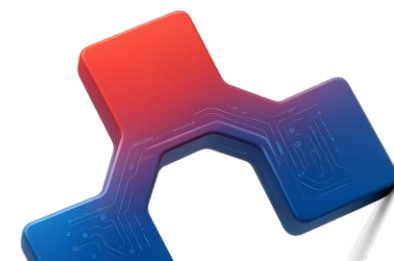
O DBSCAN funciona bem quando os clusters estão separados por regiões de baixa densidade. No exemplo abaixo, usamos o dataset `make_moons`:

```
from sklearn.cluster import DBSCAN
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=1000, noise=0.05)
dbscan = DBSCAN(eps=0.05, min_samples=5)
dbscan.fit(X)
print(dbscan.labels_[:20])
```



- Com $\epsilon = 0.05$, muitos pontos foram considerados anomalias e 7 clusters apareceram.
- Aumentando para $\epsilon = 0.2$, o algoritmo encontra os dois clusters corretos, sem excesso de ruído.



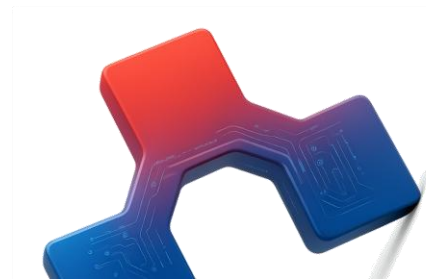


O DBSCAN armazena informações úteis:

- `labels_`: índice do cluster de cada instância (-1 indica anomalia).
- `core_sample_indices_`: índices das instâncias centrais.
- `components_`: as próprias instâncias centrais.

```
print(dbscan.core_sample_indices_[:10])  
print(dbscan.components_[:5])  
  
array([ 0,  2, -1, -1,  1,  0,  0,  0,  2,  5])  
array([0, 4, 5, 6, 7])
```

Esses atributos permitem diferenciar pontos centrais, pontos de borda e anomalias.

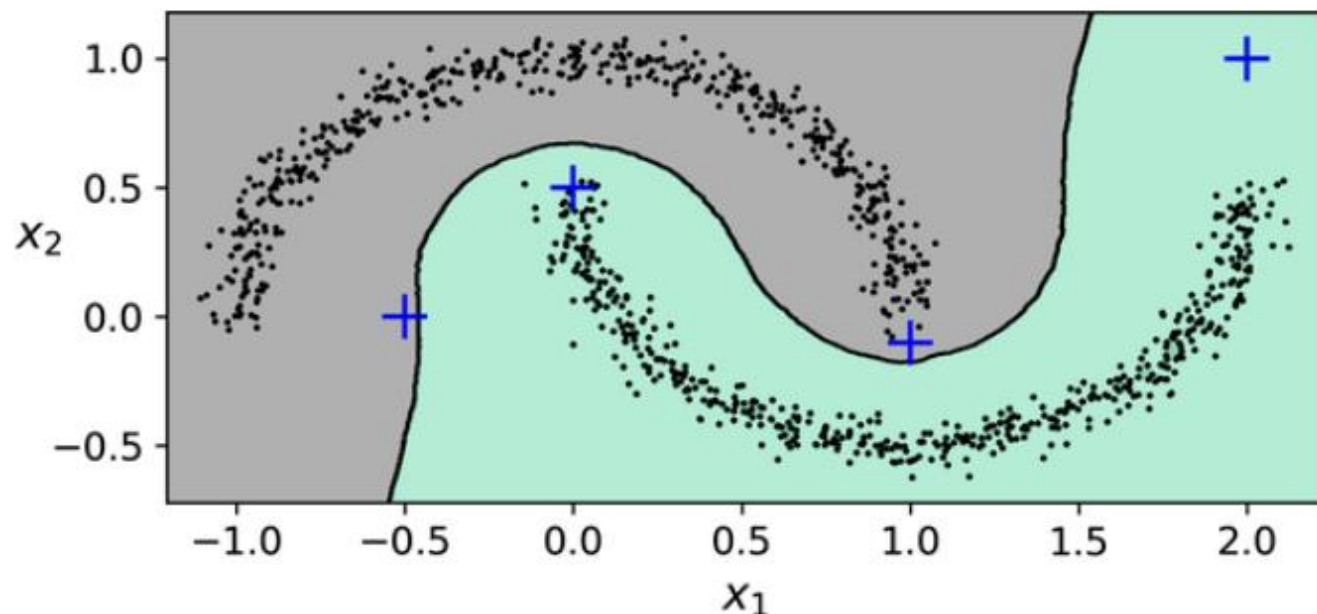


O DBSCAN não possui um método `predict()`. Para prever o cluster de novos pontos, podemos treinar um classificador supervisionado (como o `KNeighborsClassifier`) usando apenas os pontos centrais.

```
from sklearn.neighbors import KNeighborsClassifier
import numpy as np

knn = KNeighborsClassifier(n_neighbors=50)
knn.fit(dbscan.components_, dbscan.labels_[dbscan.core_sample_indices_])

X_new = np.array([[-0.5, 0], [0, 0.5], [1, -0.1], [2, 1]])
print(knn.predict(X_new))
print(knn.predict_proba(X_new))
```



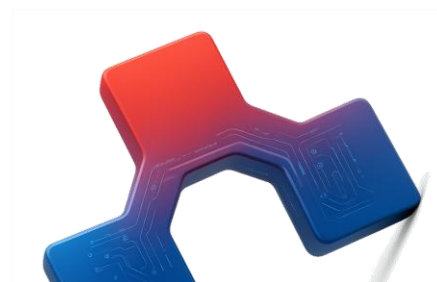


Por padrão, o classificador sempre atribui um cluster. Para identificar outliers distantes, podemos definir um **limite máximo de distância**. Se o ponto estiver além desse limite, ele é classificado como anomalia.

```
y_dist, y_pred_idx = knn.kneighbors(X_new, n_neighbors=1)
y_pred = dbscan.labels_[dbscan.core_sample_indices_][y_pred_idx]
y_pred[y_dist > 0.2] = -1
print(y_pred.ravel()) # [-1, 0, 1, -1]
```

Assim, instâncias muito afastadas são corretamente marcadas como **anomalias**.

O DBSCAN é um algoritmo simples e poderoso, capaz de identificar **qualquer número de clusters de qualquer formato**. Ele é robusto a outliers e precisa de apenas dois hiperparâmetros (**eps** e **min_samples**).

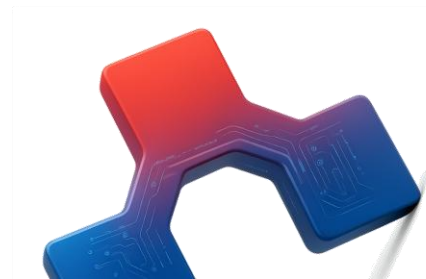




Apesar de suas vantagens, o DBSCAN **apresenta desafios**:

- Se os clusters tiverem **densidades muito diferentes**, o algoritmo pode falhar.
- Se não houver **regiões de baixa densidade claras**, os clusters podem se confundir.
- Sua complexidade é aproximadamente $O(m^2n)$, o que limita a escalabilidade para bases muito grandes.

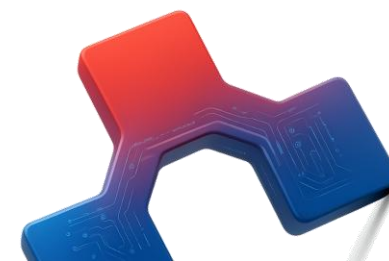
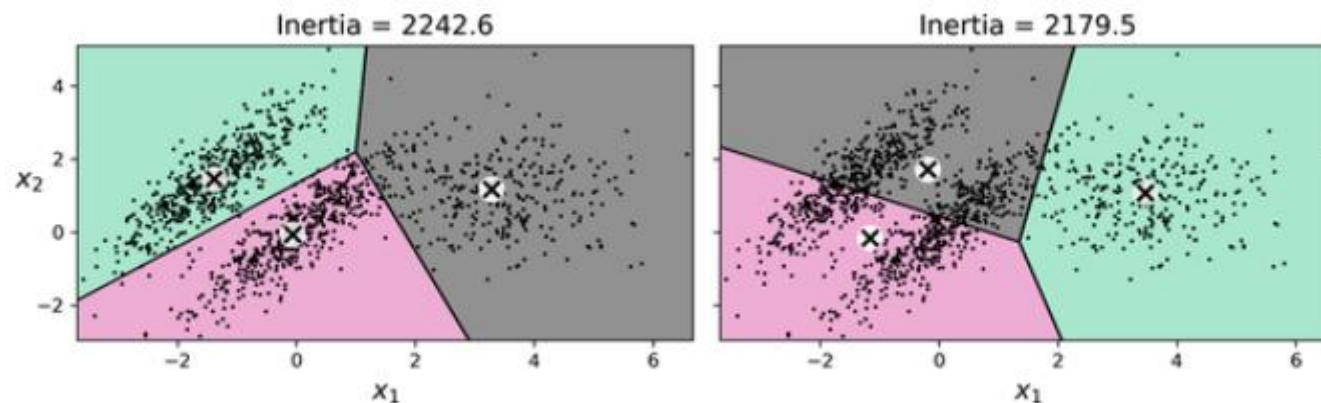
Uma alternativa é o **HDBSCAN**, disponível no projeto *scikit-learn-contrib*, que lida melhor com clusters de densidades variadas.





2. Gaussian Mixtures

Um **GMM** é um modelo probabilístico que assume que os dados foram gerados por uma mistura de várias **distribuições Gaussianas**. Cada cluster corresponde a uma Gaussiana, que pode ter formato **elipsoidal**, **tamanhos**, **densidades** e **orientações diferentes**. Dada uma instância, sabemos que ela vem de uma das distribuições, mas não sabemos de qual, nem seus parâmetros.





O processo de geração pode ser descrito assim:

1. Escolhe-se aleatoriamente um cluster j , com probabilidade proporcional ao seu peso $\phi(j)$.
2. A instância é gerada a partir da distribuição Gaussiana correspondente:

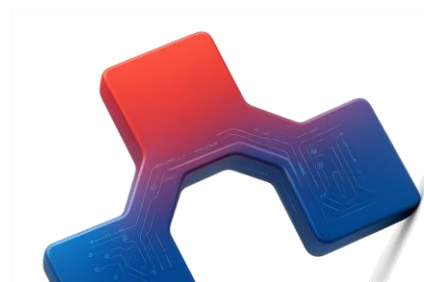
$$\mathbf{x}^{(i)} \sim \mathcal{N}(\boldsymbol{\mu}^{(j)}, \boldsymbol{\Sigma}^{(j)})$$

O objetivo do modelo é estimar os pesos (ϕ), as médias (μ) e as covariâncias (Σ). No Scikit-Learn, isso é feito com a classe `GaussianMixture`:

```
from sklearn.mixture import GaussianMixture

gm = GaussianMixture(n_components=3, n_init=10)
gm.fit(X)

print(gm.weights_)      # pesos
print(gm.means_)        # médias
print(gm.covariances_)  # matrizes de covariância
```



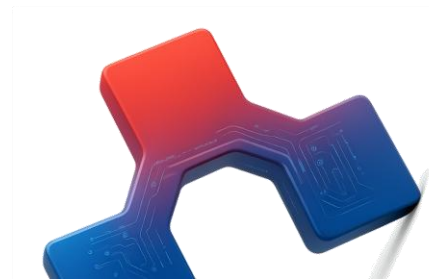


O GMM é treinado com o algoritmo **EM (Expectation-Maximization)**, semelhante ao K-Means:

- **Passo de Expectation (E)**: estima a probabilidade de cada instância pertencer a cada cluster.
- **Passo de Maximization (M)**: atualiza os parâmetros do cluster (μ , Σ , ϕ), ponderando cada instância por essa probabilidade.

O **EM** é uma generalização do K-Means: além de encontrar **os centros**, encontra também **tamanho, forma e orientação dos clusters**, e suas **probabilidades relativas**.
Importante: **EM** usa **soft clustering** (probabilidades), enquanto o K-Means usa hard clustering.

```
print(gm.converged_) # True
print(gm.n_iter_)    # número de iterações até convergência
```





Um GMM pode realizar tanto:

Hard clustering: atribui cada instância ao cluster mais provável (predict).

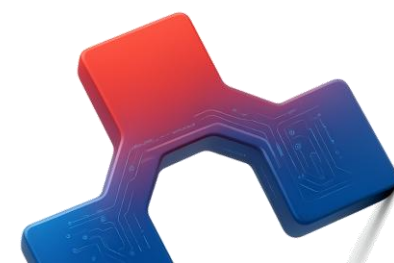
Soft clustering: calcula a probabilidade de pertencer a cada cluster (predict_proba).

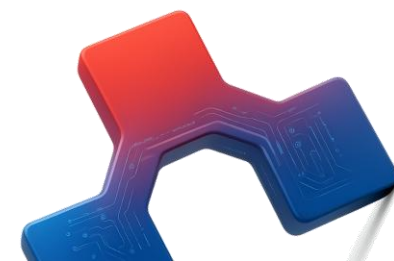
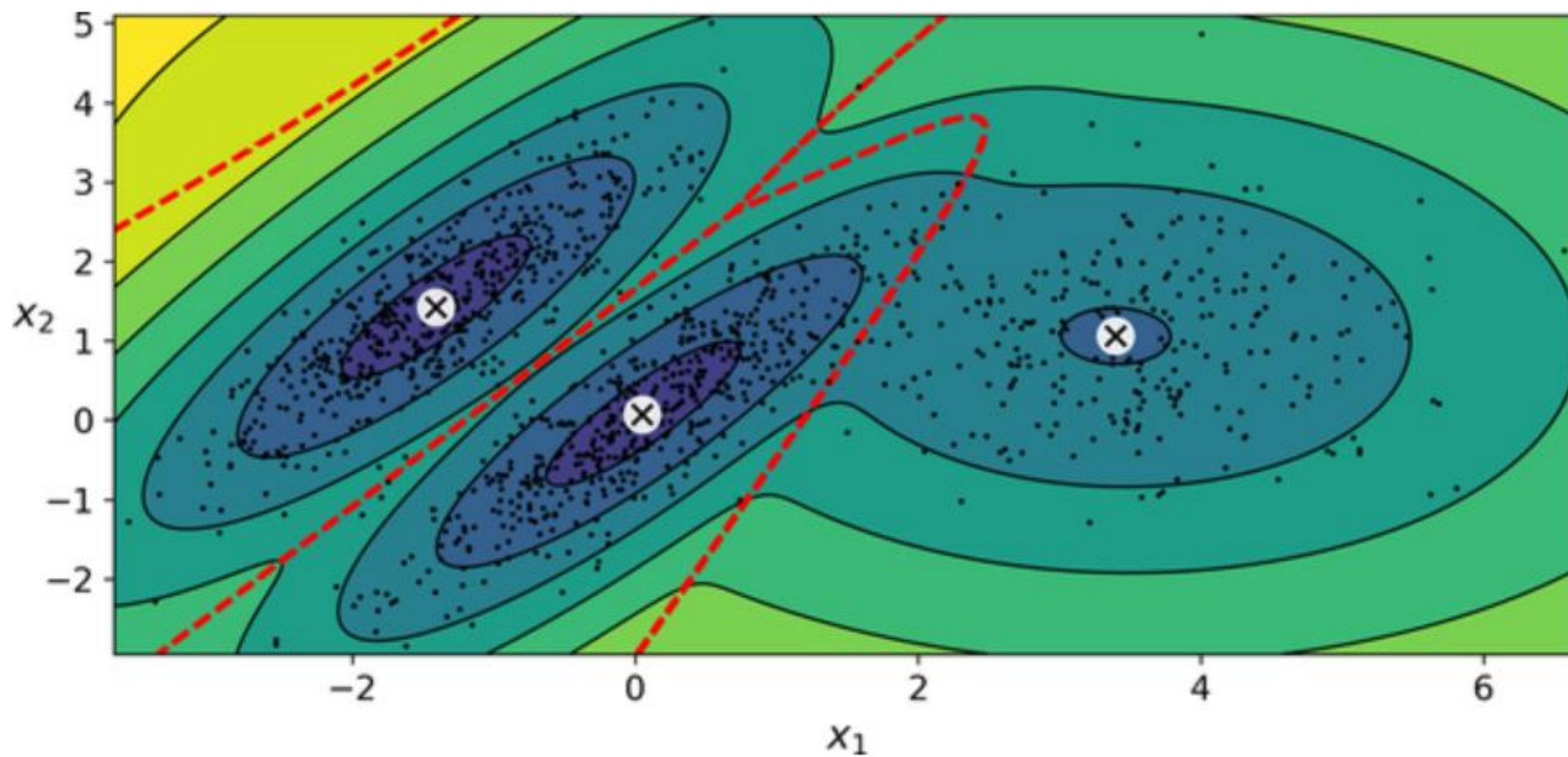
```
print(gm.predict(X)[:10])          # hard clustering
print(gm.predict_proba(X)[:5])     # soft clustering (probabilidades)
```

Além disso, por ser um **modelo generativo**, podemos gerar novas amostras ou estimar a densidade dos dados.

```
X_new, y_new = gm.sample(6)
print(X_new)
print(y_new)

print(gm.score_samples(X)[:10])    # log densidade
```







No exemplo, os pesos verdadeiros eram 0.4, 0.4 e 0.2. O modelo estimou valores muito próximos. As médias e covariâncias também ficaram próximas dos verdadeiros. Isso mostra que o GMM conseguiu recuperar bem a estrutura dos dados.

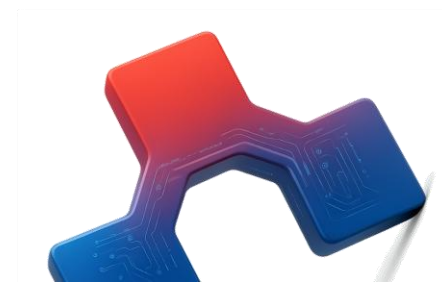
Para tornar o problema mais fácil ou acelerar o treinamento, podemos restringir as matrizes de covariância:

"spherical" → clusters esféricos, com variâncias diferentes.

"diag" → elipses cujos eixos são paralelos aos eixos das coordenadas.

"tied" → todos os clusters compartilham a mesma forma, tamanho e orientação.

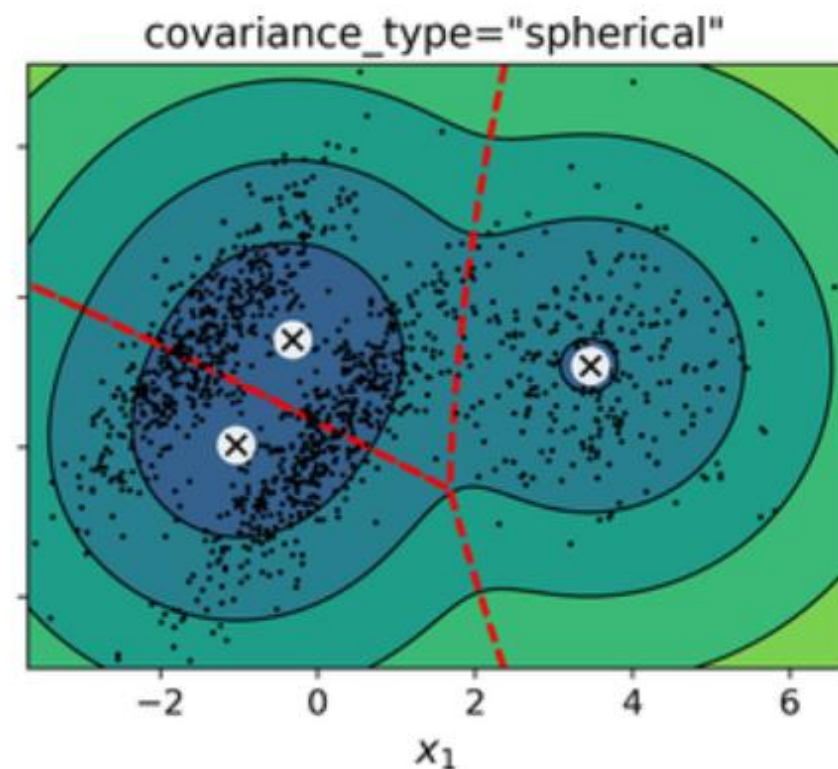
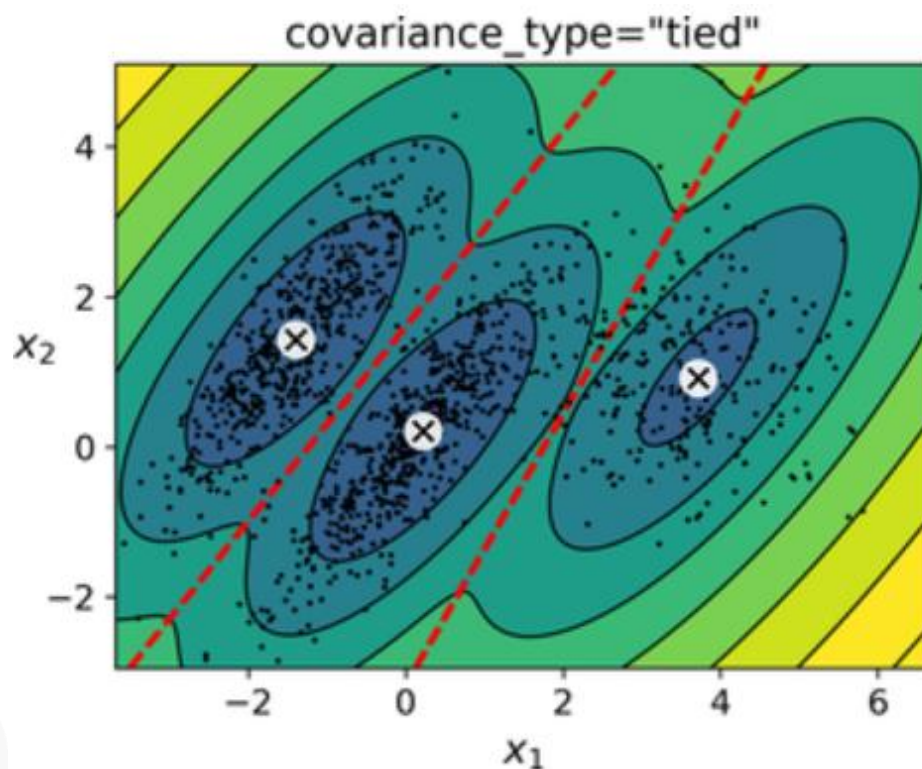
"full" (padrão) → cada cluster pode ter forma, tamanho e orientação independentes.





Resultados com covariance_type = tied (esquerda) e spherical (direita)

```
gm = GaussianMixture(n_components=3, covariance_type="tied", n_init=10)  
gm.fit(X)
```





O custo de treinamento de um GMM depende de m (número de instâncias), n (dimensões), k (clusters) e do tipo de covariância:

- "spherical" ou "diag" $\rightarrow O(kmn)$
- "tied" ou "full" $\rightarrow O(kmn^2 + kn^3)$

Isso significa que modelos "tied" ou "full" não escalam bem para dados com muitas features.

Os **GMMs** são modelos probabilísticos flexíveis:

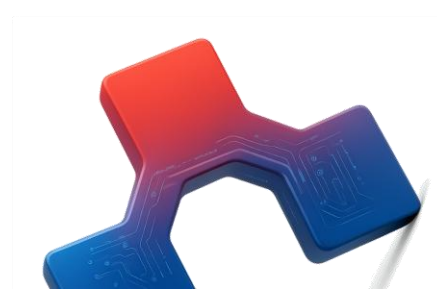
Capturam clusters de diferentes tamanhos, formas e densidades.

Usam **soft clustering**, atribuindo probabilidades aos pontos.

São modelos **generativos**, permitindo gerar novos dados.

Precisam de múltiplas inicializações para evitar soluções ruins.

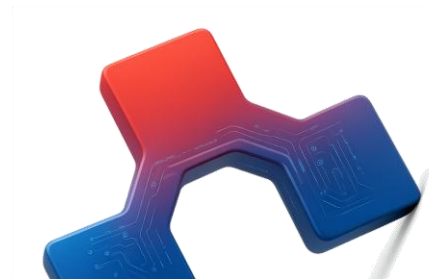
São mais poderosos que K-Means, mas também mais custosos computacionalmente.





2.1 Using Gaussian Mixtures for Anomaly Detection

No K-Means, usamos métricas como inércia ou silhouette score para escolher o **número de clusters**. Porém, no caso dos **Gaussian Mixtures**, essas métricas **não são confiáveis** quando os clusters têm **tamanhos ou formatos diferentes**. Para GMM, usamos critérios teóricos como o **AIC (Akaike Information Criterion)** e o **BIC (Bayesian Information Criterion)**, que equilibram ajuste do modelo e complexidade.





2.2 Selecting the Number of Clusters

Os critérios AIC e BIC funcionam assim:

m = número de instâncias.

p = número de parâmetros do modelo.

$\mathcal{L}(\theta | X)$ = valor máximo da função de verossimilhança.

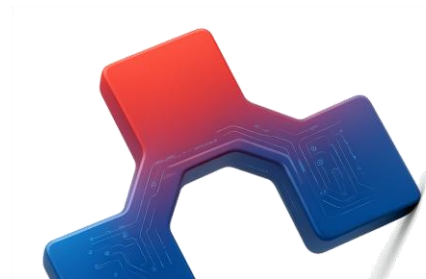
Ambos penalizam modelos muito complexos (com muitos parâmetros) e recompensam modelos que se ajustam bem aos dados. Normalmente, escolhem o mesmo modelo; quando diferem, o BIC tende a escolher soluções mais simples.

$$BIC = \log(m)p - 2 \log(\widehat{\mathcal{L}})$$

$$AIC = 2p - 2 \log(\widehat{\mathcal{L}})$$

```
print(gm.bic(X)) # 8189.74
```

```
print(gm.aic(X)) # 8102.52
```





PDF, função de verossimilhança e log-verossimilhança

É essencial entender a diferença:

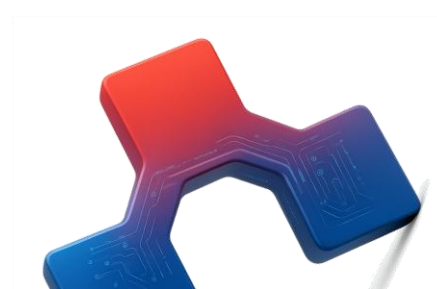
- **Probabilidade (PDF)**: quão plausível é um futuro valor x , dado um parâmetro θ fixo.
- **Verossimilhança (Likelihood)**: quão plausível é um valor de θ , dado um valor observado de x .

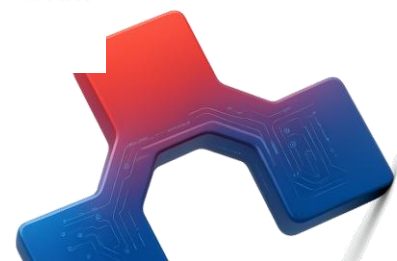
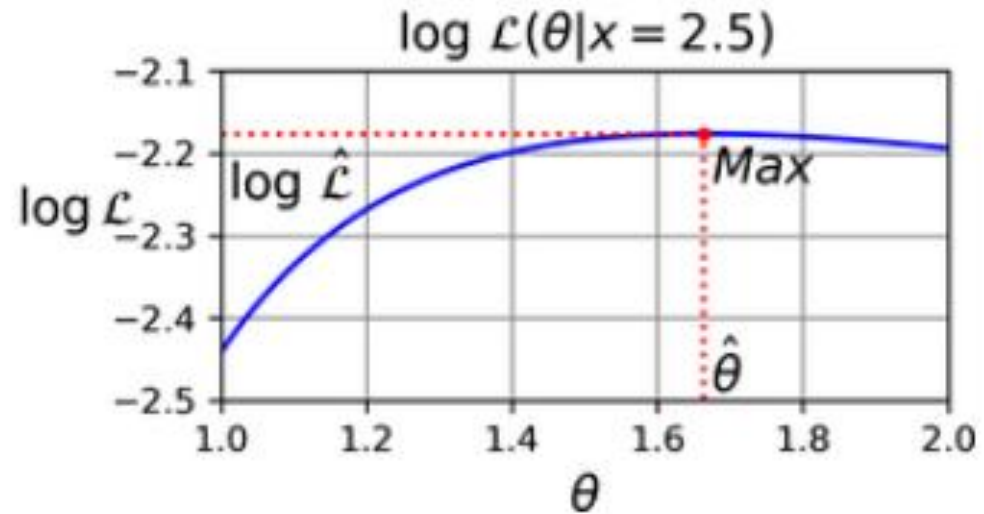
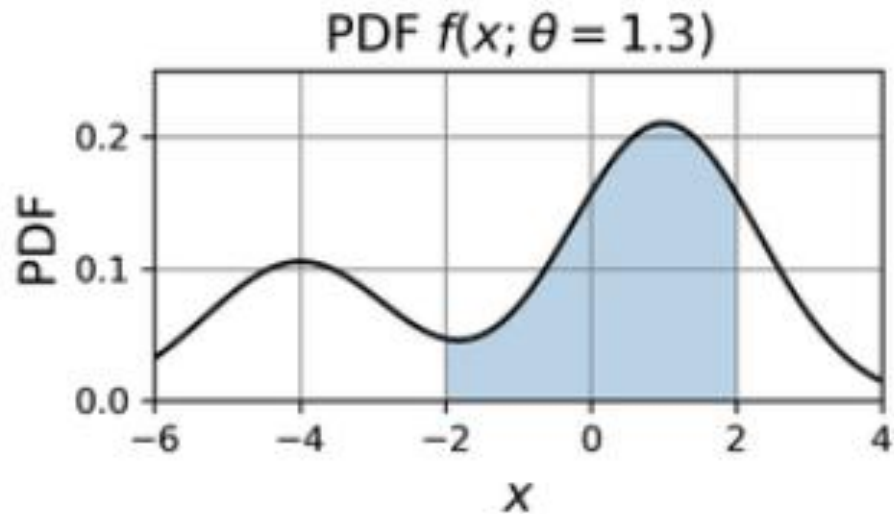
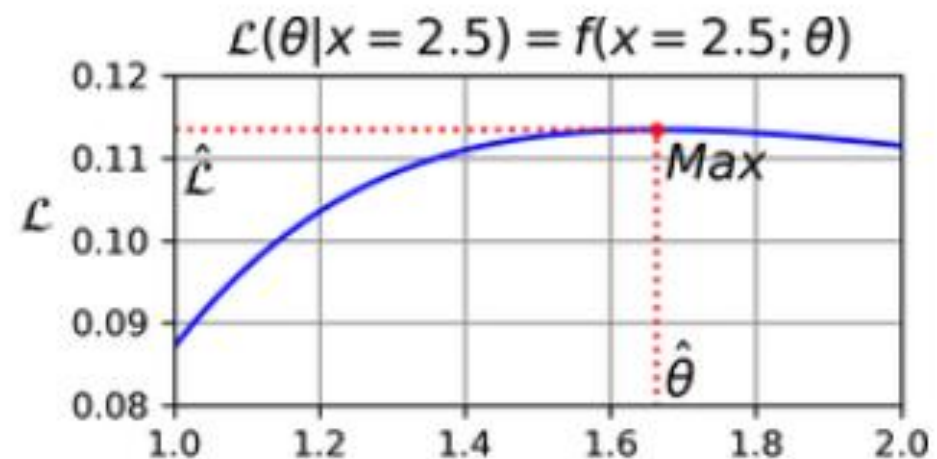
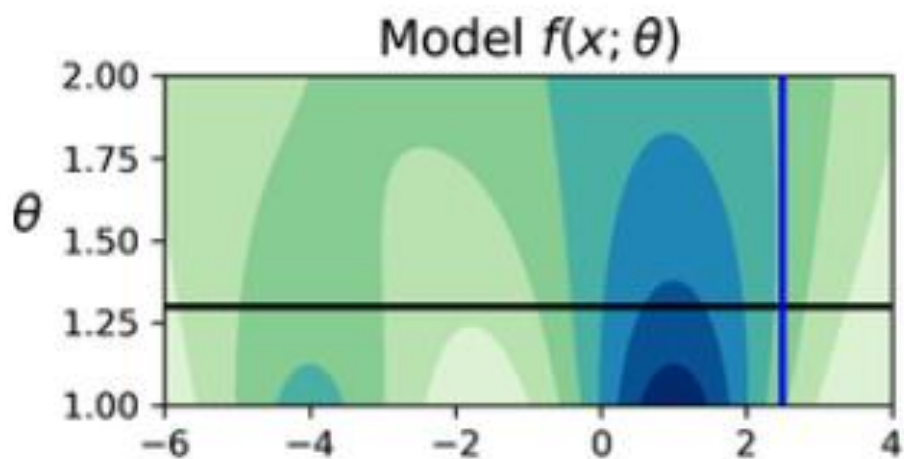
Na figura a seguir, vemos que a **PDF** é uma **função de x** (com θ fixo), enquanto a **verossimilhança** é uma **função de θ** (com x fixo). A **log-verossimilhança** facilita os cálculos, pois transforma produtos em somas.

O valor de θ que maximiza a verossimilhança é chamado de **MLE (Maximum Likelihood Estimate)**.

Se tivermos uma distribuição a priori $g(\theta)$, podemos maximizar $\mathcal{L}(\theta|x)g(\theta)$, obtendo a **MAP (Maximum A Posteriori)**, que funciona como uma versão regularizada do MLE.

Na prática, quase sempre maximizamos a **log-verossimilhança**, pois é matematicamente mais simples e numericamente mais estável.



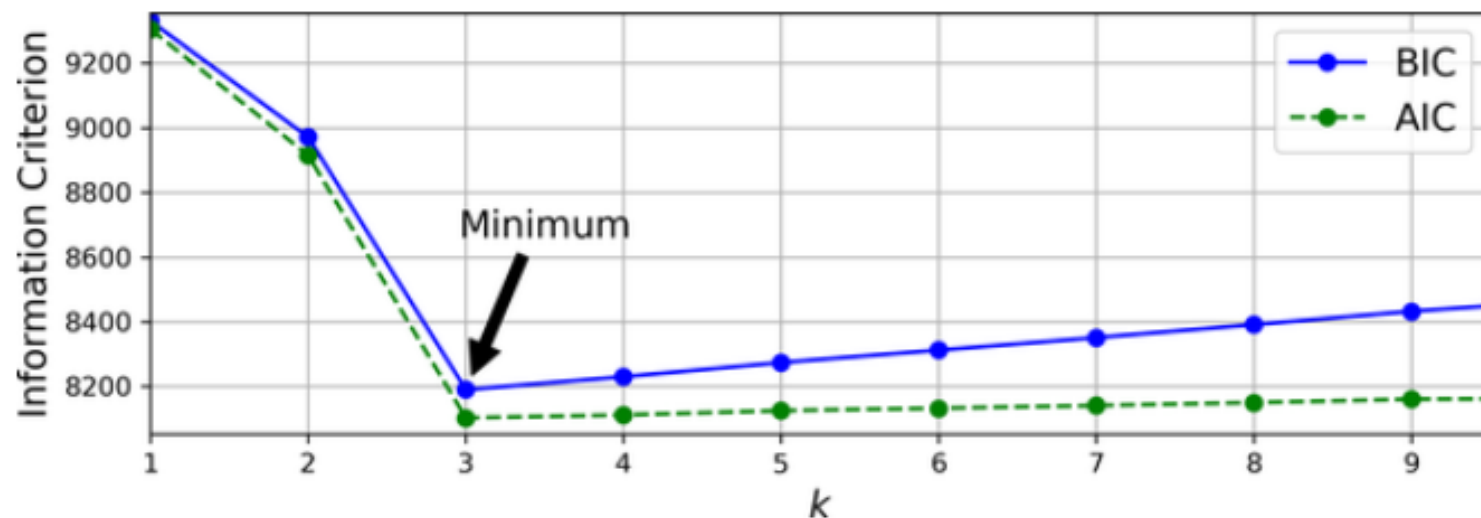




Ao treinar modelos GMM com diferentes valores de k , podemos comparar seus valores de [AIC](#) e [BIC](#). O número de clusters escolhido é aquele que minimiza esses critérios. Na figura abaixo, ambos são mínimos em $k=3$, sugerindo que esse é o número ideal de clusters para o dataset.

```
from sklearn.mixture import GaussianMixture

bics = []
aics = []
for k in range(1, 10):
    gm = GaussianMixture(n_components=k, n_init=10, random_state=42).fit(X)
    bics.append(gm.bic(X))
    aics.append(gm.aic(X))
```



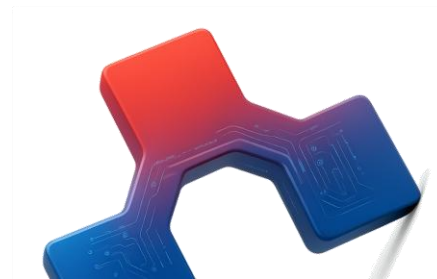


AIC e BIC são critérios de seleção de modelos baseados em equilíbrio entre ajuste e simplicidade.

Ambos penalizam excesso de parâmetros e evitam overfitting.

AIC tende a escolher modelos mais complexos; **BIC** tende a escolher modelos mais simples.

A escolha final deve considerar não apenas o valor mínimo, mas também a interpretabilidade dos clusters e o contexto do problema.





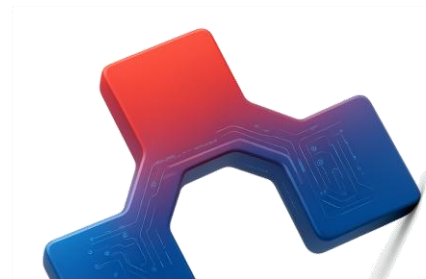
2.3 Bayesian Gaussian Mixture Models

O [BayesianGaussianMixture](#) é uma variação do GMM que elimina automaticamente clusters desnecessários. Definimos um número de clusters maior que o necessário, e o algoritmo atribui pesos próximos de zero aos clusters irrelevantes.

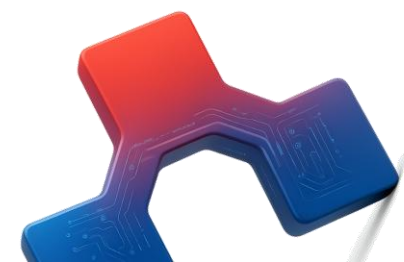
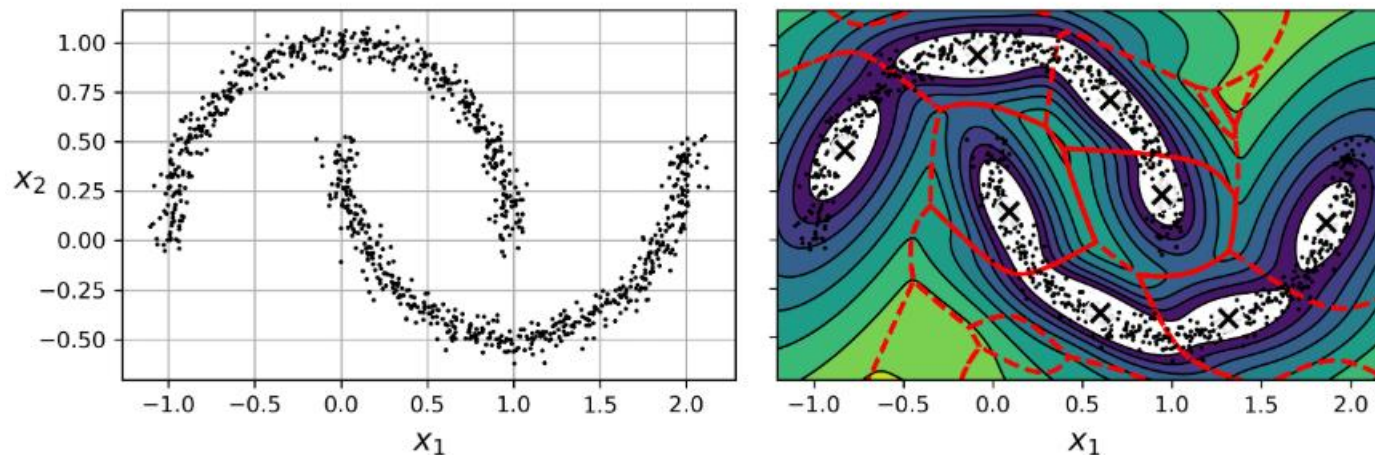
```
from sklearn.mixture import BayesianGaussianMixture

bgm = BayesianGaussianMixture(n_components=10, n_init=10, max_iter=500,
                               random_state=42)
bgm.fit(X)
print(bgm.weights_.round(2))
# Exemplo: [0.4, 0.21, 0.39, 0., 0., 0., 0., 0., 0., 0.]
```

No exemplo, apenas [3 clusters foram realmente utilizados](#), mostrando que o modelo consegue simplificar automaticamente.



Embora o Bayesian GMM seja prático, ele assume que os clusters têm **formato elipsoidal**. Em dados com formatos diferentes, como o dataset *moons*, o algoritmo falha, encontrando vários clusters artificiais (8 em vez de 2). Ainda assim, a estimação de densidade pode ser útil para **detecção de anomalias**, mas o agrupamento fica incorreto.

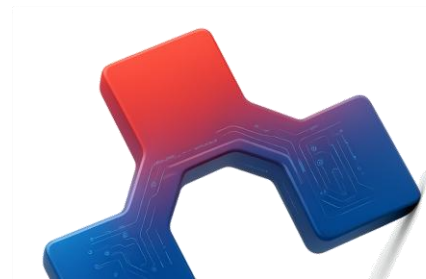




2.4 Other Algorithms for Anomaly and Novelty Detection

O Scikit-Learn implementa diferentes algoritmos voltados para **detecção de anomalias e novidades**:

- **Fast-MCD (EllipticEnvelope)**: assume que inliers vêm de uma Gaussiana única; ignora outliers ao ajustar a distribuição.
- **Isolation Forest**: usa árvores de decisão aleatórias para isolar instâncias; anomalias tendem a ser isoladas em menos passos.
- **Local Outlier Factor (LOF)**: compara a densidade local de uma instância com a de seus vizinhos; pontos mais isolados são anomalias.





O **One-Class SVM** é usado para **novelty detection**. Ele mapeia instâncias para um espaço de alta dimensão e encontra uma região que abrange os dados normais. Novas instâncias fora dessa região são consideradas anomalias.

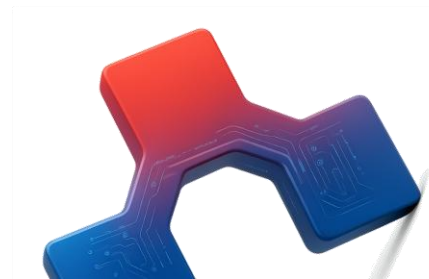
- Funciona muito bem em dados de alta dimensão.
- Tem hiperparâmetros como o kernel e a margem (que controla a taxa de falsos positivos).
- Limitação: não escala bem para grandes bases de dados.

Técnicas de redução de dimensionalidade, como o **PCA**, também podem ser usadas para detectar anomalias. A ideia é comparar o **erro de reconstrução**:

Instâncias normais → erro de reconstrução pequeno.

Anomalias → erro de reconstrução muito maior.

Esse método é simples, eficiente e pode ser combinado com outras técnicas.





Neste capítulo vimos como o [aprendizado não supervisionado](#) pode ser usado em várias aplicações:

- [K-Means](#) para clustering, redução de dimensionalidade e segmentação de imagens.
- [DBSCAN](#) para clusters de formas arbitrárias e robustez a outliers.
- [Gaussian Mixtures e Bayesian GMMs](#) para modelagem probabilística, soft clustering e estimação de densidade.

Algoritmos especializados em [detecção de anomalias](#), como Isolation Forest, LOF, One-Class SVM e PCA. O aprendizado não supervisionado é uma ferramenta poderosa para explorar dados sem rótulos e construir sistemas mais inteligentes.

