1、可视化了解Kmeans、DBSCAN
K-means算法流程
DBSCAN算法流程
两种算法的优缺点
K-means算法
DBSCAN算法
总结
2、Kmeans 在sklearn中的基本实践
2.1 决策边界
2.2 算法流程
2.3 不稳定的结果
2.4 评估方法
2.5 找到最佳簇数
2.6 轮廓系数
3、动手实践（要求报告）——聚类评估+图像分割小例子

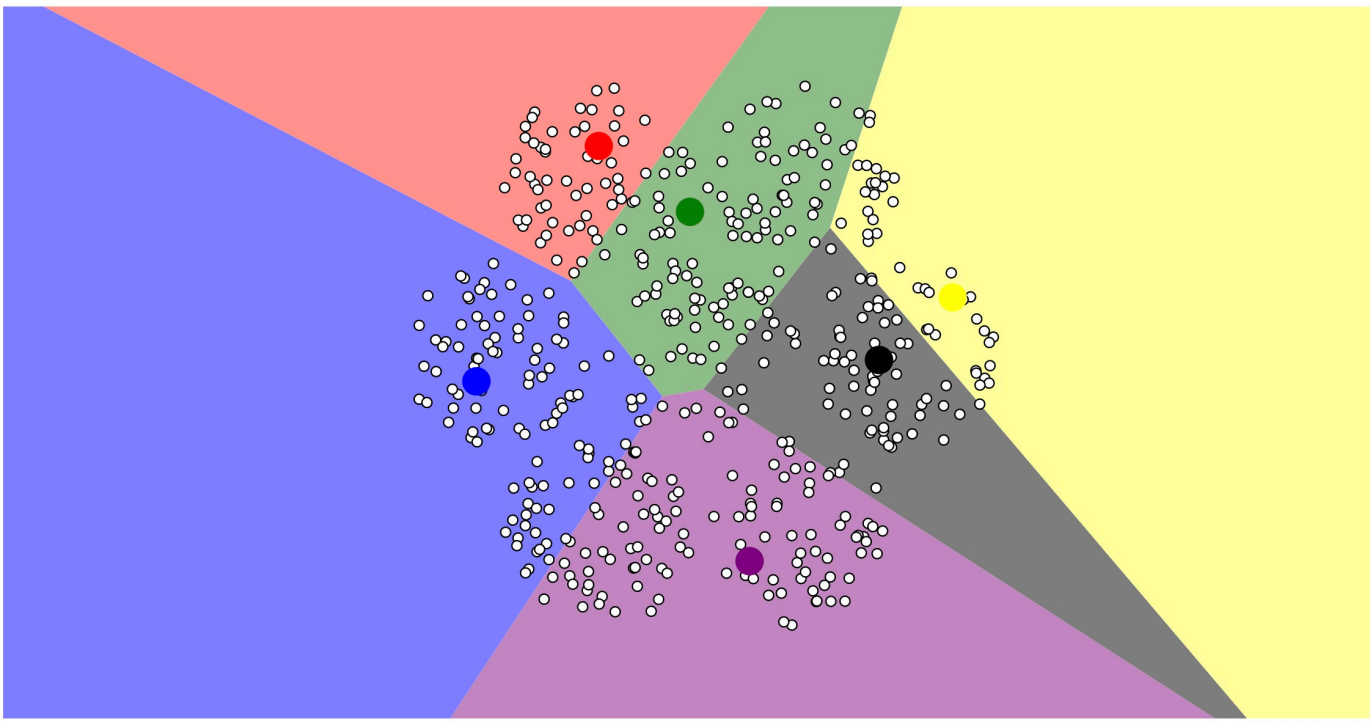
1、可视化了解Kmeans、DBSCAN

- Kmeans
- DBSCAN

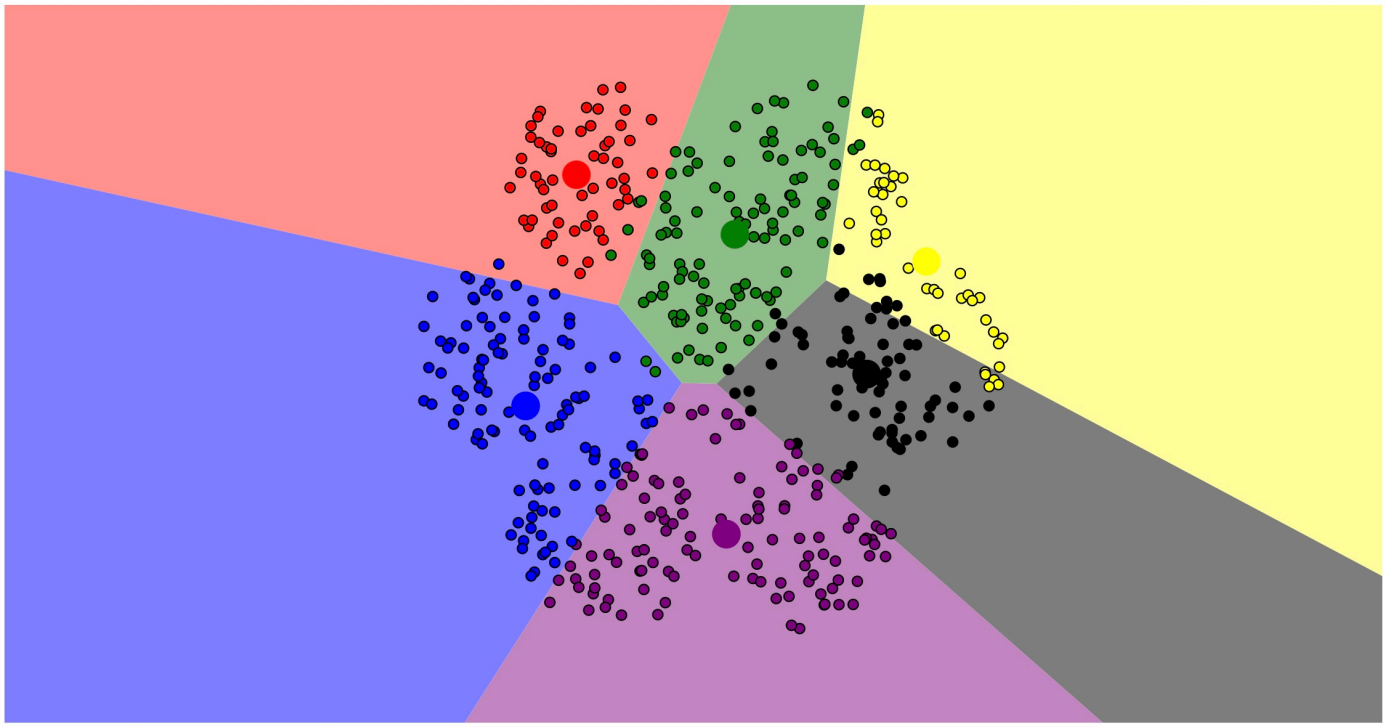
通过以上两个链接可以看到Kmeans与DBSCAN的可视化聚类过程，简单来说：

Kmeans：

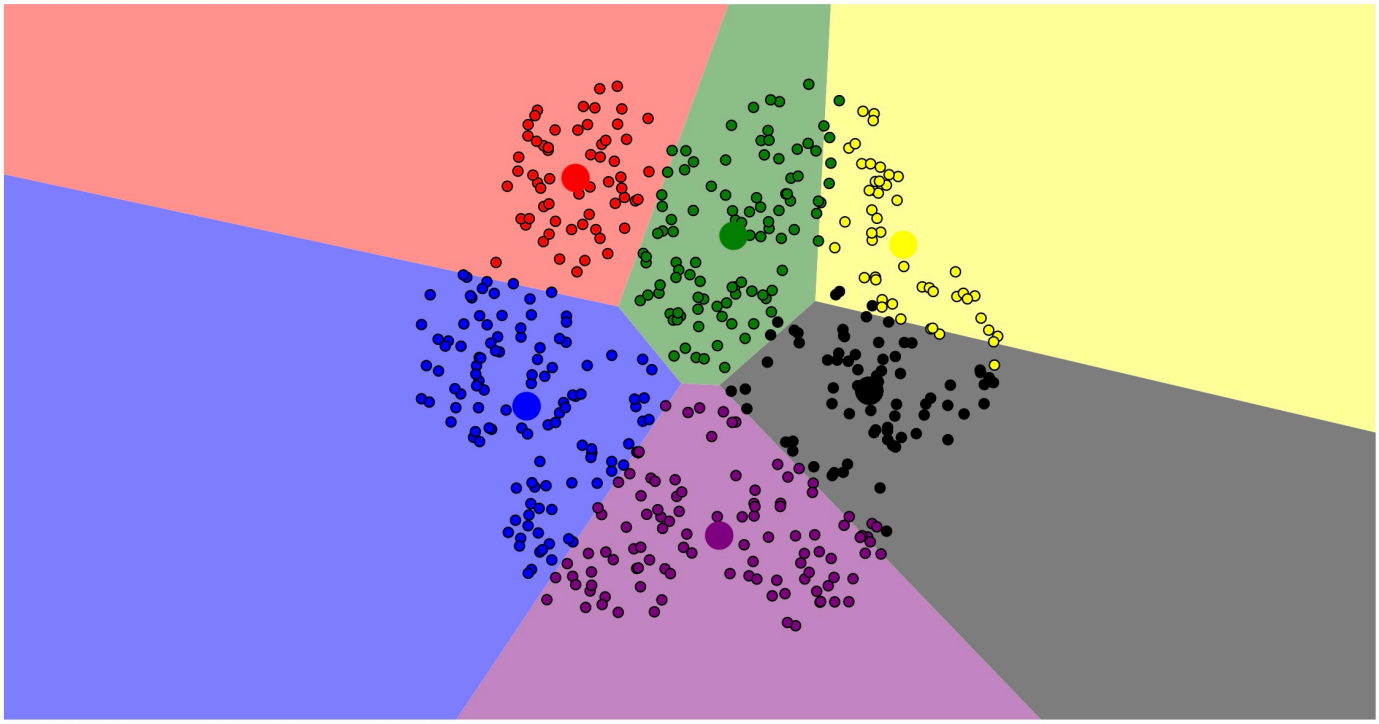
初始化：



第一次迭代：

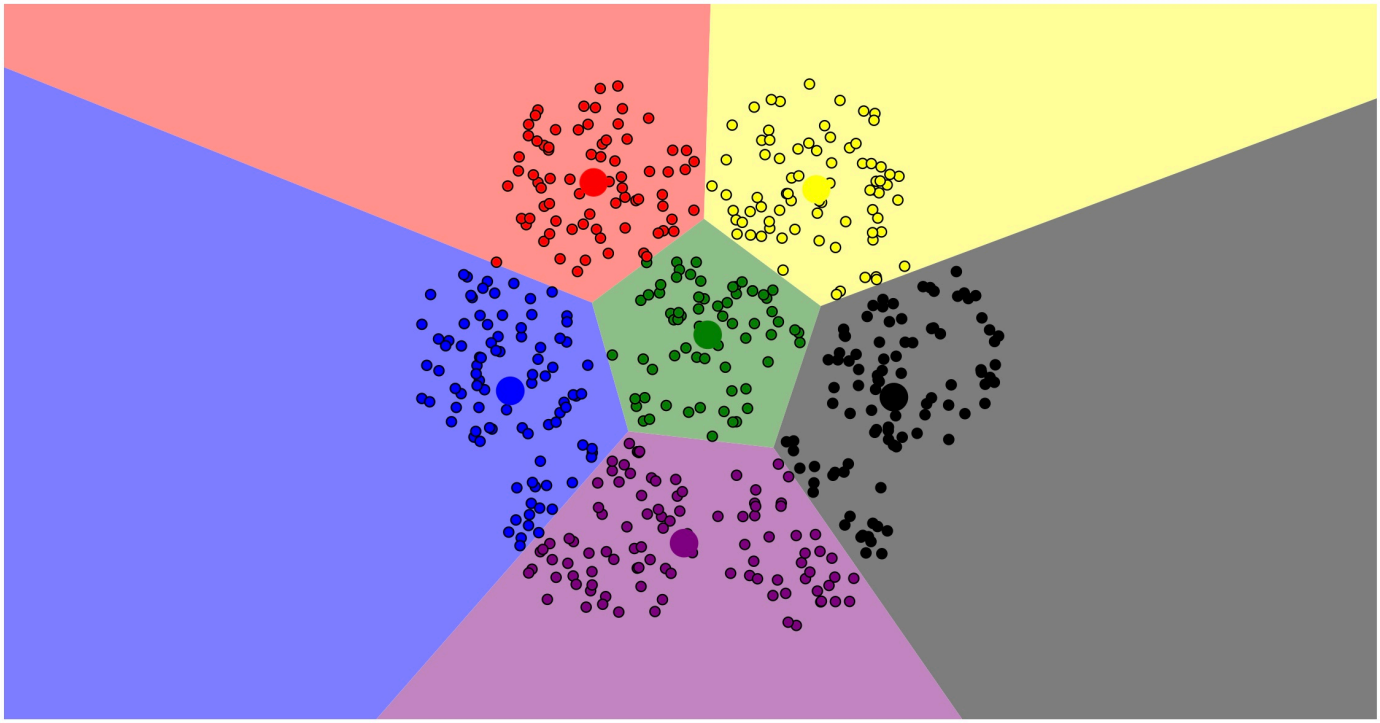


第二次迭代：



...

第n次迭代：



对于DBSCAN算法，则可以得到一个动画的效果，同学们可以动手自己尝试一下。

了解了Kmeans算法与DBSCAN算法，我们需要理清这两种的流程：

K-means算法流程

K-means是一种基于划分的聚类算法，用于将n个对象根据特征分为k个聚类，目标是使得同一聚类中的对象间的相似度高，而不同聚类间的对象相似度低。

流程步骤如下：

1. **初始化**
    - 随机选择k个数据点作为初始聚类中心。
2. **分配步骤**
    - 对于每个数据点，计算它与所有聚类中心的距离，并将其分配给最近的聚类中心所代表的聚类。
3. **更新步骤**
    - 对于每个聚类，重新计算该聚类所有数据点的均值，将该均值作为新的聚类中心。
4. **重复**
    - 重复步骤2和步骤3，直到聚类中心不再发生变化，或者达到预定的迭代次数，算法结束。

DBSCAN算法流程

DBSCAN（Density-Based Spatial Clustering of Applications with Noise）是一种基于密度的聚类算法，能够识别出任意形状的聚类，并处理噪声数据。

流程步骤如下：

1. **定义参数**
    - ε（eps）：搜索半径。

- 最小点数（MinPts）：形成密集区域所需的最小点数。

2. **对于每个点x**
   - 使用ε邻域查询找到点x的所有邻居。
   - 如果x的邻居数量小于MinPts，标记x为噪声。
   - 如果x的邻居数量大于等于MinPts，创建一个新的聚类，并将x及其所有可达的密度直接连通的点加入该聚类。
   - 继续递归扩展聚类，直到没有新的点可以添加。
3. **处理所有点**
   - 继续处理所有的点，直到每个点都被访问，被标记为某个聚类的一部分或者被标记为噪声。

两种算法的优缺点

K-means算法和DBSCAN算法是两种常用的聚类方法，它们在数据分析和机器学习中有着广泛的应用。每种方法都有其独特的优势和局限性，适用于不同的场景。下面我将比较这两种方法的优缺点，并说明它们各自适用的场景。

K-means算法

优点:

- **简单直观**: K-means算法容易理解和实现，对于大规模数据集也能保持较高的效率。
- **计算效率高**: 在处理大数据集时，尤其是数据维度相对较低时，K-means可以非常高效。
- **广泛应用**: 适用于各种类型的数据分析任务。

缺点:

- **需要预先指定聚类数K**: 在算法运行前需要确定聚类的数量，这在很多实际应用中是一个难点。
- **对初始值敏感**: 初始聚类中心的选择可能会影响到最终的聚类结果。
- **可能陷入局部最优**: 根据初始聚类中心的不同，可能得到不同的聚类结果。
- **对异常值敏感**: 异常值或噪声可以对聚类结果产生较大影响。
- **假设聚类为球形**: 在实际应用中，如果聚类的形状远离球形，K-means的效果可能不佳。

适用场景:

- 数据集较大，维度较低。
- 聚类呈现球形分布。
- 对聚类形状和大小的偏好不强。

DBSCAN算法

优点:

- **不需要预设聚类数**: 与K-means不同，DBSCAN不需要在算法运行前确定聚类的数量。
- **可以发现任意形状的聚类**: DBSCAN可以识别出任意形状的聚类，更加灵活。

- **对噪声有较好的鲁棒性**: 能有效地识别并处理噪声点。
- **参数少，易于实现**: 虽然需要设置ε和MinPts，但通常情况下比选择K值更直观。

缺点:

- **对参数敏感**: ε和MinPts的选择直接影响聚类的结果，不同的参数设置可能导致截然不同的聚类结构。
- **处理高维数据的效率较低**: 当数据维度增加时，计算每个点的邻域变得复杂和耗时，效率较低。
- **变量密度的聚类效果不佳**: 如果数据集中的聚类具有非常不同的密度，DBSCAN可能难以识别所有的聚类。

适用场景:

- 数据形状复杂，聚类结构不规则。
- 数据集中存在噪声或异常值。
- 不希望预先设定聚类数量或假设聚类形状。

总结

K-means适合处理大规模的数据集，特别是当聚类呈现球形分布时效果较好，但需要预先确定聚类数目。DBSCAN更适合于识别任意形状的聚类，对噪声和异常值有较好的处理能力，但对参数选择敏感，且在处理高维数据时效率较低。选择哪种算法取决于具体的数据特征和分析目标。

2、Kmeans 在sklearn中的基本实践

官方文档

下面是对Kmeans算法的一些使用，以及一些常用的聚类评价指标的使用，建议大家使用jupyter notebook一步一步进行学习。

引入必要的库和模块、配置Matplotlib参数、忽略警告以及设置随机种子：

```
import numpy as np
import os
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
plt.rcParams['axes.labelsize'] = 14
plt.rcParams['xtick.labelsize'] = 12
plt.rcParams['ytick.labelsize'] = 12
import warnings
warnings.filterwarnings('ignore')
np.random.seed(42)
```

使用`make_blobs`函数来生成模拟数据，这些数据可以用于聚类算法（如K-means或DBSCAN）的演示或测试。`make_blobs`是一个方便的工具，用于生成多类随机分布的点集，通常用于聚类和分类算法的模拟数据生成。构建5个中心，进行发散，生成5个簇。

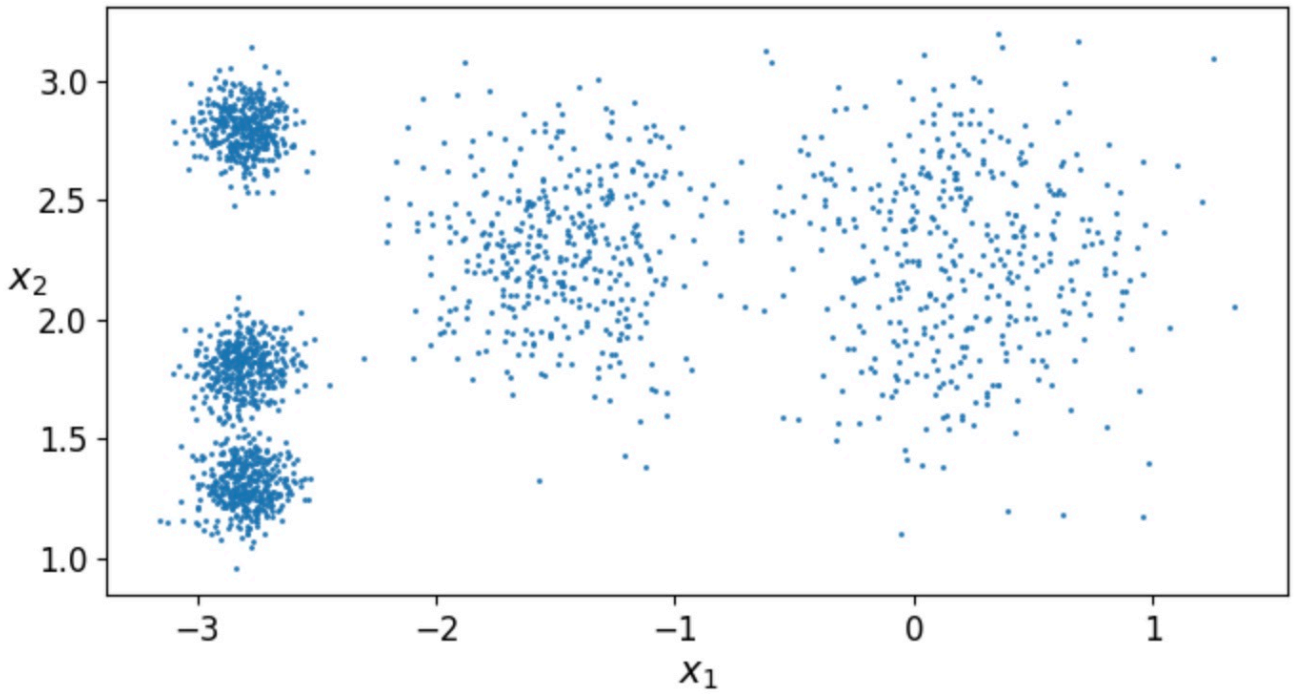
```
from sklearn.datasets import make_blobs

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])
```

生成了一个包含2000个数据点的数据集，这些点分布在由`blob_centers`指定位置的群集周围，每个群集的数据点散布范围由相应的`blob_std`值决定。生成的数据集`X`和每个点的群集标签`y`可以用于训练和测试聚类算法，比如K-means或DBSCAN，来检验算法的有效性和性能。

```
X,y = make_blobs(n_samples=2000,centers=blob_centers,
                      cluster_std = blob_std,random_state=7)
```

函数`plot_clusters`用于绘制聚类结果，并随后使用这个函数来绘制之前通过`make_blobs`函数生成的数据集`X`的散点图。

```
def plot_clusters(X, y=None):
    plt.scatter(X[:, 0], X[:, 1], c=y, s=1)
    plt.xlabel("$x_1$", fontsize=14)
    plt.ylabel("$x_2$", fontsize=14, rotation=0)
plt.figure(figsize=(8, 4))
plot_clusters(X)
plt.show()
```



使用scikit-learn（sklearn）库中的KMeans聚类算法来对数据集`X`进行聚类。

```
from sklearn.cluster import KMeans
k = 5
kmeans = KMeans(n_clusters = k,random_state=42)
y_pred =  kmeans.fit_predict(X)
```

fit_predict(X)与kmeans.labels_ 得到预测结果是一致的

```
y_pred
# array([0, 0, 4, ..., 3, 1, 0], dtype=int32)
kmeans.labels_
# array([0, 0, 4, ..., 3, 1, 0], dtype=int32)
kmeans.cluster_centers_
# array([[-2.80214068,  1.55162671],
#        [ 0.08703534,  2.58438091],
#        [-1.46869323,  2.28214236],
#        [-2.79290307,  2.79641063],
#        [ 0.31332823,  1.96822352]])
```

使用先前训练好的KMeans模型来预测新数据点的聚类标签

```
X_new = np.array([[0,2],[3,2],[-3,3],[-3,2.5]])
kmeans.predict(X_new)
# array([4, 4, 3, 3], dtype=int32)
```

2.1 决策边界

定义了三个函数，用于可视化聚类数据、聚类中心，以及聚类的决策边界。

plot_data(X)

- 该函数绘制数据集`X`中所有点的散点图。每个点用黑色的点（'k.'）表示，点的大小设置为2。
- `plt.plot(X[:, 0], X[:, 1], 'k.', markersize=2)`：这行代码实现了上述绘图，`X[:, 0]`和`X[:, 1]`分别代表数据点的x坐标和y坐标。

plot_centroids(centroids, weights=None, circle_color='w', cross_color='k')

- 该函数用于绘制聚类中心。聚类中心以圆圈（默认为白色）和叉号（默认为黑色）标记，可通过`weights`参数来过滤掉一些中心。
- `if weights is not None`：如果提供了`weights`，则仅绘制权重大于权重最大值十分之一的聚类中心。
- `plt.scatter(..., marker='o', ...)`：以圆圈的形式绘制过滤后的聚类中心。
- `plt.scatter(..., marker='x', ...)`：在相同的聚类中心位置上绘制叉号，以增强可视化效果。

plot_decision_boundaries(clusterer, X, resolution=1000, show_centroids=True, show_xlabels=True, show_ylabels=True)

- 该函数绘制聚类的决策边界，并根据需要显示聚类中心、x轴标签和y轴标签。

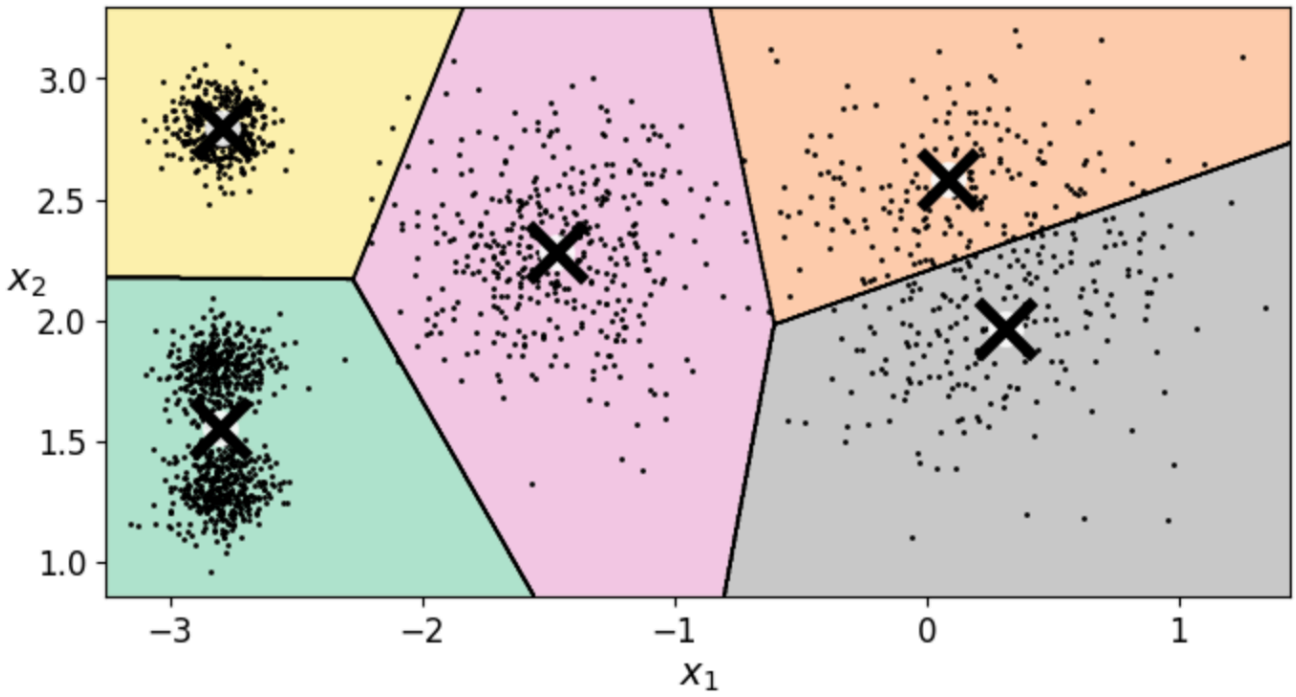
- 首先，创建一个网格覆盖整个数据空间，`np.meshgrid`根据`X`数据的最小值和最大值生成。
- `Z = clusterer.predict(np.c_[xx.ravel(), yy.ravel()])`：对于网格中的每个点，使用聚类器（如KMeans实例）的`predict`方法预测它所属的聚类。
- `Z = Z.reshape(xx.shape)`：将预测结果`Z`重塑为与`xx`网格相同的形状。
- 使用`plt.contourf`和`plt.contour`绘制聚类的决策区域和边界线。
- 调用`plot_data(X)`绘制原始数据点。
- 如果`show_centroids`为True，调用`plot_centroids`函数显示聚类中心。
- 根据`show_xlabels`和`show_ylabels`参数决定是否显示坐标轴标签。

```
def plot_data(X):
    plt.plot(X[:, 0], X[:, 1], 'k.', markersize=2)

def plot_centroids(centroids, weights=None, circle_color='w', cross_color='k'):
    if weights is not None:
        centroids = centroids[weights > weights.max() / 10]
    plt.scatter(centroids[:, 0], centroids[:, 1],
                marker='o', s=30, linewidths=8,
                color=circle_color, zorder=10, alpha=0.9)
    plt.scatter(centroids[:, 0], centroids[:, 1],
                marker='x', s=10, linewidths=30,
                color=cross_color, zorder=11, alpha=1)

def plot_decision_boundaries(clusterer, X, resolution=1000, show_centroids=True,
                             show_xlabels=True, show_ylabels=True):
    mins = X.min(axis=0) - 0.1
    maxs = X.max(axis=0) + 0.1
    xx, yy = np.meshgrid(np.linspace(mins[0], maxs[0], resolution),
                         np.linspace(mins[1], maxs[1], resolution))
    Z = clusterer.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    plt.contourf(Z, extent=(mins[0], maxs[0], mins[1], maxs[1]),
                cmap="Pastel2")
    plt.contour(Z, extent=(mins[0], maxs[0], mins[1], maxs[1]),
                linewidths=1, colors='k')
    plot_data(X)
    if show_centroids:
        plot_centroids(clusterer.cluster_centers_)

    if show_xlabels:
        plt.xlabel("$x_1$", fontsize=14)
    else:
        plt.tick_params(labelbottom='off')
    if show_ylabels:
        plt.ylabel("$x_2$", fontsize=14, rotation=0)
    else:
```

```
        plt.tick_params(labelleft='off')
```

绘图

```
plt.figure(figsize=(8, 4))
plot_decision_boundaries(kmeans, X)
plt.show()
```



2.2 算法流程

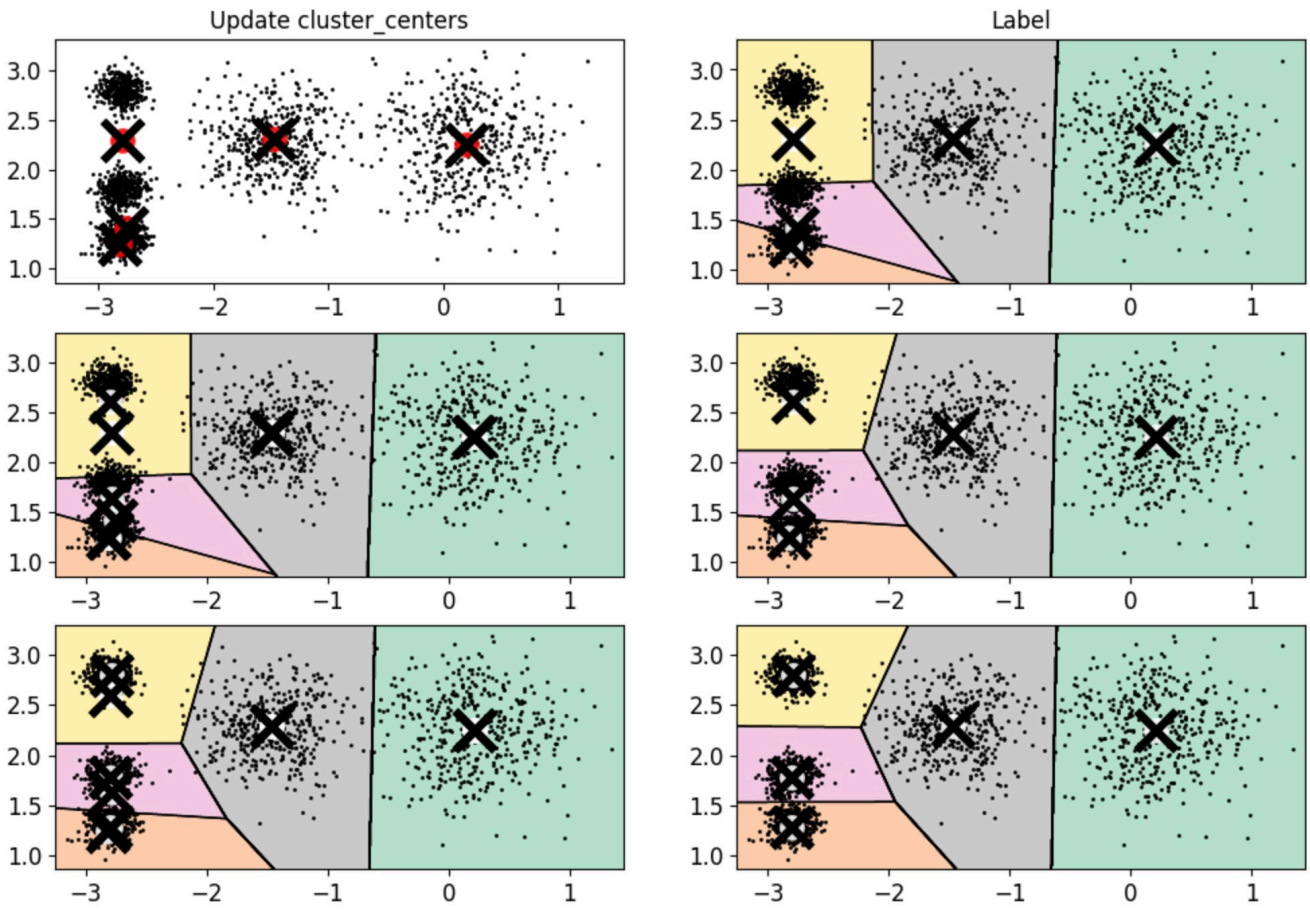
使用scikit-learn的`KMeans`类来执行K-means聚类算法，并展示了算法迭代过程中的不同阶段。在这个例子中，创建了三个`KMeans`实例，每个实例都用于对相同的数据集`X`进行聚类，但是它们迭代的次数（通过`max_iter`参数控制）各不相同。

```
kmeans_iter1 = KMeans(n_clusters = 5,init = 'random',n_init =
1,max_iter=1,random_state=1)
kmeans_iter2 = KMeans(n_clusters = 5,init = 'random',n_init =
1,max_iter=2,random_state=1)
kmeans_iter3 = KMeans(n_clusters = 5,init = 'random',n_init =
1,max_iter=3,random_state=1)

kmeans_iter1.fit(X)
kmeans_iter2.fit(X)
kmeans_iter3.fit(X)
```

```
▾ KMeans
KMeans(init='random', max_iter=3, n_clusters=5, n_init=1, random_state=1)
```

利用`matplotlib`库创建了一个图形，展示了KMeans聚类算法在不同迭代阶段的聚类结果及其决策边界。

```
plt.figure(figsize=(12,8))
plt.subplot(321)
plot_data(X)
plot_centroids(kmeans_iter1.cluster_centers_, circle_color='r', cross_color='k')
plt.title('Update cluster_centers')

plt.subplot(322)
plot_decision_boundaries(kmeans_iter1, X,show_xlabels=False, show_ylabels=False)
plt.title('Label')

plt.subplot(323)
plot_decision_boundaries(kmeans_iter1, X,show_xlabels=False, show_ylabels=False)
plot_centroids(kmeans_iter2.cluster_centers_,)

plt.subplot(324)
plot_decision_boundaries(kmeans_iter2, X,show_xlabels=False, show_ylabels=False)

plt.subplot(325)
plot_decision_boundaries(kmeans_iter2, X,show_xlabels=False, show_ylabels=False)
plot_centroids(kmeans_iter3.cluster_centers_,)

plt.subplot(326)
plot_decision_boundaries(kmeans_iter3, X,show_xlabels=False, show_ylabels=False)

plt.show()
```



- **左侧列**展示了在不同迭代次数下，聚类中心是如何被更新的。
- **右侧列**则显示了在每次迭代后，数据点是如何根据最新的聚类中心被标记（或分配）到不同聚类的，以及聚类决策边界的变化。

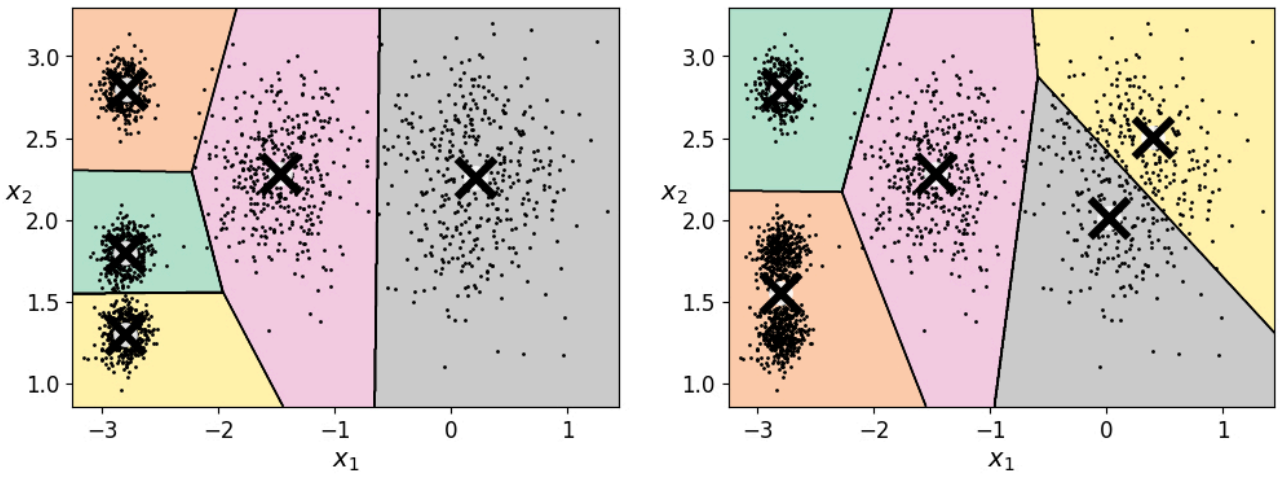
2.3 不稳定的结果

函数`plot_clusterer_comparison`，用于比较和展示两个聚类器（`c1`和`c2`）在同一个数据集`X`上的聚类结果及其决策边界。

```
def plot_clusterer_comparison(c1,c2,X):
    c1.fit(X)
    c2.fit(X)

    plt.figure(figsize=(12,4))
    plt.subplot(121)
    plot_decision_boundaries(c1,X)
    plt.subplot(122)
    plot_decision_boundaries(c2,X)
```

创建两个KMeans聚类器实例`c1`和`c2`，并使用之前定义的`plot_clusterer_comparison`函数比较它们在同一数据集`X`上的聚类效果。

```
c1 = KMeans(n_clusters = 5,init='random',n_init = 1,random_state=11)
c2 = KMeans(n_clusters = 5,init='random',n_init = 1,random_state=22)
plot_clusterer_comparison(c1,c2,X)
```



随着随机种子或者初始位置的不同，得到的结果具有不稳定性。

2.4 评估方法

- Inertia指标：每个样本与其质心的距离

```
kmeans.inertia_
# 219.428000736476
```

- transform得到的是当前样本到每个簇中心距离

```
kmeans.transform(X)
# array([[0.23085922, 3.04838567, 1.54568385, 1.45402521, 3.07528232],
#        [0.26810747, 3.06126045, 1.48314418, 0.99002955, 3.19186267],
#        [3.78216716, 1.66209651, 2.67172567, 4.09069201, 1.02742236],
#        ...,
#        [1.17785478, 2.89371096, 1.4073312 , 0.06769209, 3.20799557],
#        [3.15905017, 0.23914671, 1.71339651, 3.05913478, 0.43887998],
#        [0.43658314, 2.79657627, 1.21395695, 0.85434589, 2.95143035]])
```

- labels_得到每个样本点属于哪个簇

```
kmeans.labels_
# array([0, 0, 4, ..., 3, 1, 0], dtype=int32)
```

- 找到每个样本到最近的中心距离

```
X_dist[np.arange(len(X_dist)),kmeans.labels_]
# array([0.23085922, 0.26810747, 1.02742236, ..., 0.06769209, 0.23914671,
#        0.43658314])
```

- Inertia指标计算

```
np.sum(X_dist[np.arange(len(X_dist)),kmeans.labels_]**2)
# 219.42800073647652
```

- 得分（负值）

```
kmeans.score(X)
# -219.428000736476
```

```
c1.inertia_
# 211.59853725816834
```

```
c2.inertia_
# 223.2910857281904
```

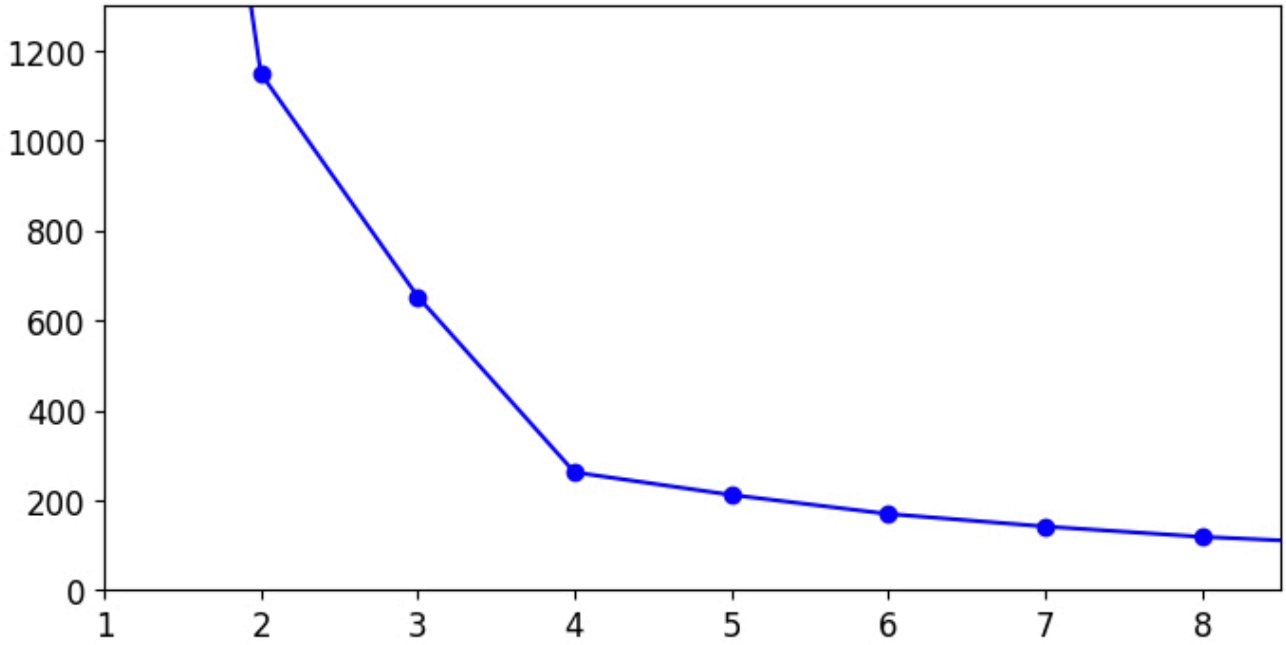
2.5 找到最佳簇数

如果K值越大，得到的结果(Inertia)肯定越小。

使用不同数量的聚类中心（从1到9）执行KMeans算法。计算并收集每个KMeans模型的惯性（inertia）。

```
kmeans_per_k = [KMeans(n_clusters = k).fit(X) for k in range(1,10)]
inertias = [model.inertia_ for model in kmeans_per_k]
```

```
plt.figure(figsize=(8,4))
plt.plot(range(1,10),inertias,'bo-')
plt.axis([1,8.5,0,1300])
plt.show()
```



助教个人觉得可以通过斜率变化比较大的点（如4）大概确定K值，但也不尽相同，还是结合实际情况。

2.6 轮廓系数

- ai：计算样本i到同簇其他样本的平均距离ai。ai越小，说明样本i越应该被聚类到该簇。将ai称为样本i的簇内不相似度。
- bi：计算样本i到其他某簇Cj的所有样本的平均距离bij，称为样本i与簇Cj的不相似度。定义为样本i的簇间不相似度：bi =min{bi1, bi2, ..., bik}

$$s(i)=\frac{b(i)-a(i)}{\max\{a(i),b(i)\}}$$

$$s(i)=\begin{cases}1-\dfrac{a(i)}{b(i)}, & a(i)<b(i)\\ 0, & a(i)=b(i)\\ \dfrac{b(i)}{a(i)}-1, & a(i)>b(i)\end{cases}$$

结论：

- si接近1，则说明样本i聚类合理；
- si接近-1，则说明样本i更应该分类到另外的簇；
- 若si 近似为0，则说明样本i在两个簇的边界上。

计算KMeans聚类结果的轮廓系数（Silhouette Score）

```
from sklearn.metrics import silhouette_score
silhouette_score(X,kmeans.labels_)
# 0.6353422668284152
```

当K值从2到9时，对应的轮廓系数。

```
silhouette_scores = [silhouette_score(X,model.labels_) for model in kmeans_per_k[1:]]
# [0.5966442557582528,
# 0.5723900247411775,
# 0.688531617595759,
# 0.655517642572828,
# 0.6020248775444942,
# 0.6070979466596362,
# 0.5614686225605264,
# 0.567647042788722]
```

绘图。

```
plt.figure(figsize=(8,4))
plt.plot(range(2,10),silhouette_scores,'bo-')
plt.show()
```