K-means

K均值（K-means）是一种用于数据聚类的无监督机器学习算法。它的目标是将一组数据点分成K个不同的簇，以使每个数据点都属于距离最近的簇中心。这个算法是一种迭代方法，通过不断优化簇中心的位置来实现数据点的分组。

聚类：

- 无监督问题：没有标签
- 聚类：相似的东西分到一组
- 难点：如果评估，如何调参

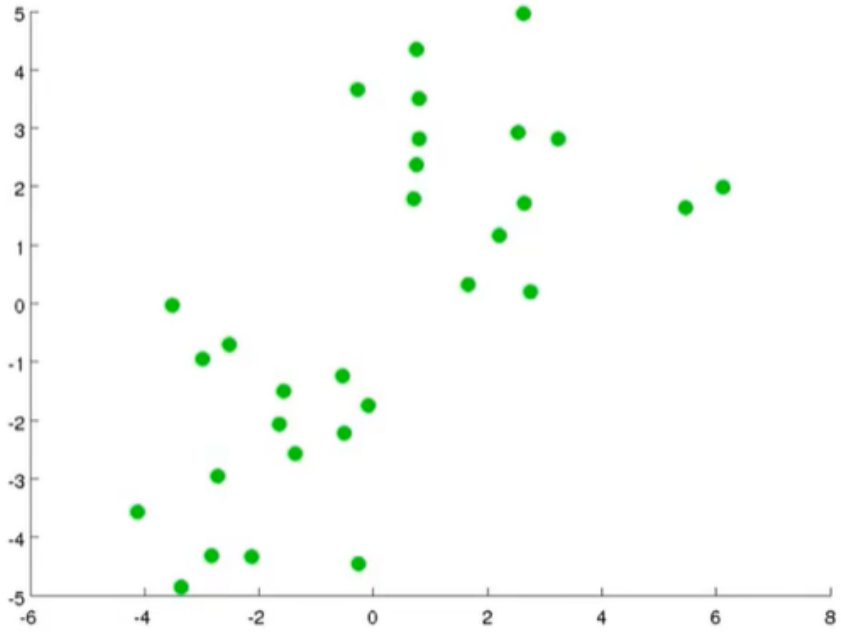
K-means算法的基本步骤如下：

1. 选择K个初始簇中心点，这些初始点可以随机选择，也可以通过其他方法来确定。
2. 对于每个数据点，计算它与K个簇中心之间的距离，通常使用欧氏距离或其他距离度量。
3. 将每个数据点分配到与之距离最近的簇中心所属的簇。
4. 对于每个簇，计算新的簇中心，通常是该簇内所有数据点的均值。
5. 重复步骤3和步骤4，直到满足停止条件，例如簇中心不再发生显著变化，或达到预定的迭代次数。

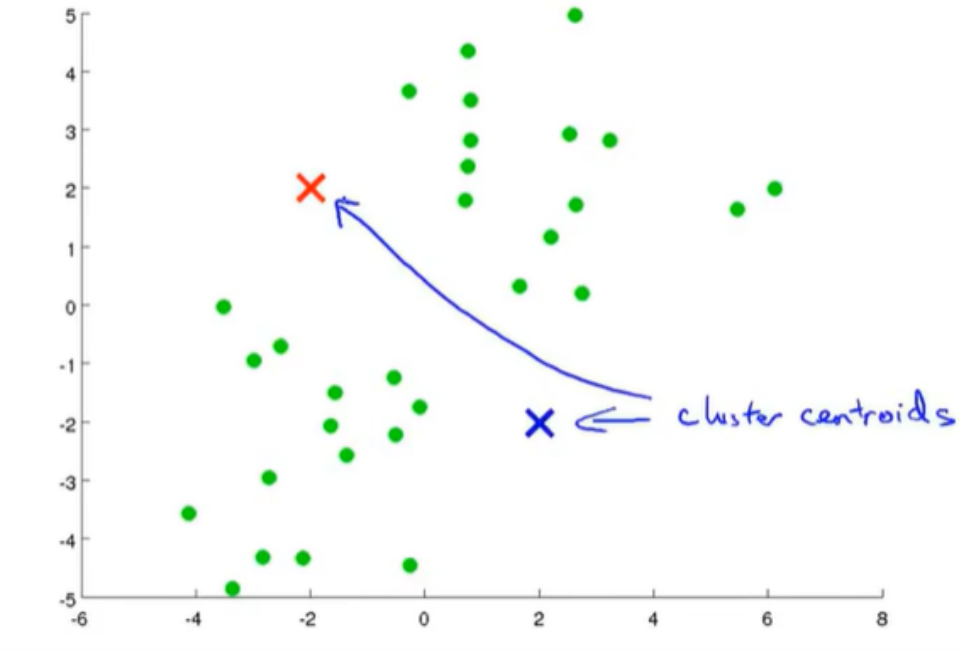
K-means算法的输出是K个不同的簇，每个簇包含一组数据点，这些数据点在某种程度上相似，而不同簇之间的数据点应该具有较大的差异。K-means算法是一种有效的数据聚类方法，通常用于图像分割、文本分析、市场分析等领域。然而，它对于K值的选择和初始簇中心的选择都有一定的敏感性，需要谨慎处理以获得好的聚类结果。

举个例子：

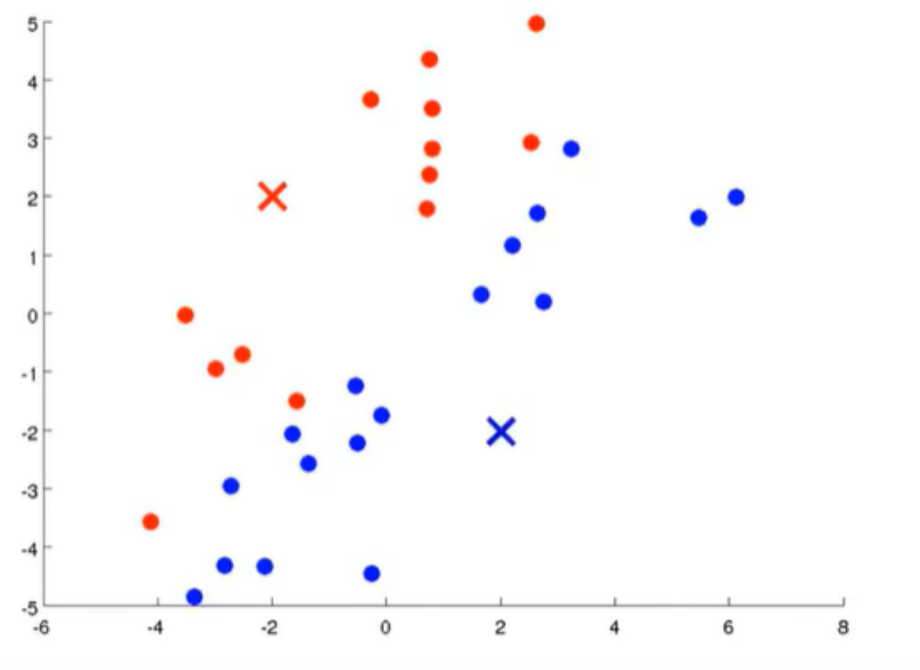
有一个无标签数据集，目标是将其分成两簇。



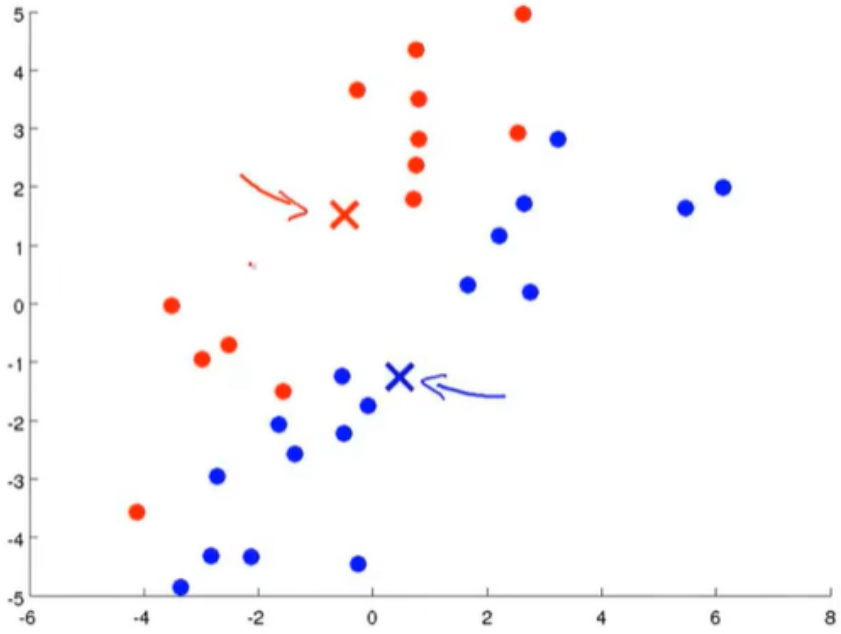
1.随机生成两点，这两点为聚类中心。（为什么是两点，因为要分成两类）



2.进行簇分配：数据离那个中心近，就分成那一类。



3.移动聚类中心:将两个聚类中心移动到同色的均值位置。



4.重复2，3部知道聚类中心不再移动。

Kmeans损失函数是如何定义？

K-means的损失函数通常被称为"簇内平方和"（Within-Cluster Sum of Squares，WCSS），它用于衡量聚类的质量。WCSS表示了每个数据点与其所属簇中心之间的距离的平方和，其定义如下：

对于每个数据点xi，它所属的簇中心为$c(xi)$，WCSS的计算如下：

$WCSS = \Sigma(xi - c(xi))^2$

其中，Σ表示对所有数据点i的求和，xi是数据点，$c(xi)$是数据点xi所属的簇中心。

WCSS的值越小，表示数据点越紧密地聚集在它们所属的簇中心周围，聚类的质量越高。K-means的主要目标是最小化WCSS，通过不断调整簇中心的位置来实现这一目标。选择K值（簇的数量）的过程通常涉及到在不同K值下计算WCSS，然后选择使WCSS值下降幅度明显减小的K值作为最佳的聚类数。

在K-means中，最优化WCSS的过程是通过迭代的方式进行的，不断更新簇中心，并将数据点重新分配到最近的簇中心，直到满足停止条件为止。这个迭代过程旨在找到最佳的簇中心，以最小化WCSS，从而得到高质量的聚类结果。

你是如何选择初始类族的中心点？

选择初始簇中心点对于K-means算法的性能和聚类结果的影响很大，不同的初始中心点选择方法可以导致不同的聚类结果。以下是一些常见的方法用于选择初始簇中心点：

1. **随机选择：** 最简单的方法是随机选择K个数据点作为初始簇中心。虽然这种方法很容易实现，但它对初始选择的敏感性较高，可能会导致陷入局部最小值的问题。
2. **K-means++：** K-means++是一种改进的初始中心点选择方法，旨在减少陷入局部最小值的风险。它的步骤如下：
   - 选择第一个簇中心点随机从数据点中选择一个。
   - 对于每个数据点，计算它与已经选择的簇中心的最短距离（即离它最近的簇中心）。
   - 根据距离的概率分布选择下一个簇中心，距离越远的点被选中的概率较大。
   - 重复上述步骤，直到选择K个初始簇中心。
3. **K-means||：** K-means||是另一种改进的初始中心点选择方法，它尝试更好地适应大规模数据集。它的工作原理与K-means++类似，但包含多次重复随机选择和距离计算步骤，以减少计算的复杂性。

选择哪种初始中心点选择方法通常取决于问题的规模、数据分布和计算资源。K-means++和K-means||通常在实践中表现更好，因为它们降低了陷入局部最小值的风险，但在小规模数据集上，随机选择也可以给出合理的结果。如果对初始中心点的选择不满意，通常可以多次运行K-means算法，并选择具有最小WCSS的结果作为最终的聚类结果。

如何提升kmeans效率？

提高K-means算法的效率对于处理大规模数据集非常重要。以下是一些方法和技巧，可以用来提升K-means的效率：

1. **减小数据集维度：** 如果数据集的维度非常高，可以使用特征选择或降维技术（如主成分分析PCA）来减小维度。这可以减少计算的复杂性，提高K-means的速度。

2. **随机子样本：** 如果数据集非常大，可以考虑在K-means之前使用随机子样本来代替完整数据集。这可以减小计算开销，同时保留了数据集的主要结构。
3. **并行化：** 使用并行计算来加速K-means。多核处理器或分布式计算环境可以用来并行计算不同数据点的距离和簇中心的更新。
4. **增量学习：** 在某些情况下，可以考虑使用增量学习方法，将新数据逐步添加到现有的聚类中，而不必每次重新运行K-means。
5. **使用更快的距离计算：** 使用更快速的距离计算方法，如欧氏距离的平方，可以减少距离计算的开销。此外，距离计算可以通过专门的线性代数库进行优化。
6. **早期停止：** 在K-means的迭代过程中，可以使用早期停止条件，当簇中心不再显著变化时提前终止迭代，而不是运行到固定的迭代次数。
7. **初始化优化：** 使用更好的初始中心点选择方法，如K-means++或K-means||，可以减少收敛所需的迭代次数，从而提高效率。
8. **使用优化库：** 使用高性能计算库和工具，如NumPy、Scikit-learn（Python）或其他支持多线程和并行计算的工具，可以提高K-means的运行效率。
9. **分布式K-means：** 对于大规模数据集，可以考虑使用分布式K-means实现，将计算任务分散到多台计算机或集群上，以加速处理。
10. **硬件加速：** 利用GPU或专用硬件加速卡进行K-means计算，这可以显著提高计算速度。

综合利用这些方法和技巧可以显著提高K-means算法的效率，特别是在处理大规模数据集时。选择适当的策略取决于具体的问题和计算资源。

Kmeans对异常值是否敏感？为什么？

敏感，因为需要计算距离，使用传统的欧式距离度量方式。所以需要预处理离群点、数据归一化

如何评估聚类效果？

要评估聚类效果，有多种方法和指标可供选择，具体的选择取决于问题的性质和数据的特点。以下是一些常见的聚类效果评估方法和指标：

1. **WCSS（Within-Cluster Sum of Squares）：** WCSS是K-means中常用的聚类评估指标，它测量每个数据点到其所属簇中心的距离的平方和。WCSS越小，表示聚类效果越好。可以使用不同的K值进行K-means聚类，然后选择WCSS最小的K值作为最佳的簇数。
2. **Silhouette分数：** Silhouette分数是一种度量聚类效果的指标，它考虑了簇内的紧密度和簇间的分离度。Silhouette分数的取值范围在-1到1之间，越接近1表示聚类效果越好。负值表示数据点更可能被分配到错误的簇。
3. **轮廓图（Silhouette Plot）：** 轮廓图可用于可视化Silhouette分数的结果，帮助你理解聚类的分离度和紧密度。轮廓图将每个数据点表示为横向的条形，条形的长度与Silhouette分数相关，可以帮助你快速识别聚类的均匀性。

4. **ARI（Adjusted Rand Index）：** ARI是一种用于比较聚类结果与真实标签的指标，通常在有地面真实标签的情况下使用。ARI的取值范围在-1到1之间，0表示随机分配，1表示完美匹配。
5. **NMI（Normalized Mutual Information）：** NMI是另一种用于比较聚类结果与真实标签的指标，它测量两者之间的相互信息。NMI的取值范围在0到1之间，0表示没有共享信息，1表示完美匹配。
6. **外部指标（External Cluster Evaluation）：** 如果你有真实标签可用，你可以使用外部指标来评估聚类效果，如精确度、召回率、F1分数等，以比较聚类结果和真实标签之间的一致性。
7. **内部指标（Internal Cluster Evaluation）：** 内部指标是不依赖于真实标签的聚类评估方法，如DBI（Davies-Bouldin Index）、Dunn指数、Gap统计等。它们提供了有关簇内部紧密度和簇间距离的信息。
8. **可视化：** 使用可视化工具和技巧，如散点图、聚类图、t-SNE、PCA等，来直观地展示聚类结果，以帮助理解数据的分布和聚类结构。

选择合适的聚类评估方法应根据具体问题的性质和数据的特点来决定。通常，结合多个指标和可视化方法，可以更全面地评估聚类效果。

优缺点

优点：

简单，快速，适合常规数据集

缺点：

1.K值难确定

2.复杂度与样本呈线性关系

3.很难发现任意形状的簇

4.对于不是凸的数据集比较难收敛
5.如果各隐含类别的数据不平衡，比如各隐含类别的数据量严重失衡，或者各隐含类别的方差不同，则聚类效果不佳。
6.采用迭代方法，得到的结果只是局部最优。（可以尝试采用二分K-Means算法）
7.对噪音和异常点比较敏感。（改进1：离群点检测的LOF算法，通过去除离群点后再聚类，可以减少离群点和孤立点对于聚类效果的影响；改进2：改成求点的中位数，这种聚类方式即K-Mediods聚类）
8.只适用于数值型数据，只能发现球型类簇。

代码：

```
import numpy as np
from matplotlib import pyplot as plt


def euclidean_distance(vecA, vecB):
    '''计算vecA与vecB之间的欧式距离'''
```

```
    # return np.sqrt(np.sum(np.square(vecA - vecB)))
    return np.linalg.norm(vecA - vecB)


def random_centroids(data, k):
    ''' 随机创建k个中心点'''
    dim = np.shape(data)[1]  # 获取向量的维度
    centroids = np.mat(np.zeros((k, dim)))
    for j in range(dim):  # 随机生成每一维中最大值和最小值之间的随机数
        min_j = np.min(data[:, j])
        range_j = np.max(data[:, j]) - min_j
        centroids[:, j] = min_j * np.mat(np.ones((k, 1))) + np.random.rand(k, 1) * range_j
    return centroids


def KMeans(data, k, distance_func=euclidean_distance):
    '''根据k-means算法求解聚类的中心'''
    m = np.shape(data)[0]  # 获得行数m
    cluster_assment = np.mat(np.zeros((m, 2)))  # 初试化一个矩阵，用来记录簇索引和存储距离平方
    centroids = random_centroids(data, k)  # 生成初始化点
    cluster_changed = True  # 判断是否需要重新计算聚类中心
    while cluster_changed:
        cluster_changed = False
        for i in range(m):
            distance_min = np.inf  # 设置样本与聚类中心之间的最小的距离，初始值为正无穷
            index_min = -1  # 所属的类别
            for j in range(k):
                distance_ji = distance_func(centroids[j, :], data[i, :])
                if distance_ji &distance_min:
                    distance_min = distance_ji
                    index_min = j
            if cluster_assment[i, 0] != index_min:
                cluster_changed = True
                cluster_assment[i, :] = index_min, distance_min ** 2  # 存储距离平方
        for cent in range(k):  # 更新质心，将每个族中的点的均值作为质心
            pts_in_cluster = data[np.nonzero(cluster_assment[:, 0].A == cent)[0]]
            centroids[cent, :] = np.mean(pts_in_cluster, axis=0)
    return centroids, cluster_assment

def show_cluster(data, k, centroids, cluster_assment):
    num, dim = data.shape
    mark = ['or', 'ob', 'og', 'oy', 'oc', 'om']
    for i in range(num):
        mark_index = int(cluster_assment[i, 0])
        plt.plot(data[i, 0], data[i, 1], mark[mark_index])
    for i in range(k):
        plt.plot(centroids[i, 0], centroids[i, 1], 'o', markeredgecolor='k', markersize=16)
    plt.show()


if __name__ == "__main__":
    data = []
    f = open("sz.txt", 'r')
    for line in f:
        data.append([float(line.split(',')[0]), float(line.split(',')[1])])
    data = np.array(data)
    k = 4
    centroids = random_centroids(data, k)
    centroids, cluster_assment = KMeans(data, k)
    show_cluster(data, k, centroids, cluster_assment)
```

K-Means++算法

k个初始化的质心的位置选择对最后的聚类结果和运行时间都有很大的影响，因此需要选择合适的k个质心。如果仅仅是完全随机的选择，有可能导致算法收敛很慢。K-Means++算法就是对K-Means随机初始化质心的方法的优化。K-Means++算法与K-Means算法最本质的区别是在k个聚类中心的初始化过程。

K-Means++算法的基本思路

K-Means++算法在聚类中心的初始化过程中的基本原则是使得**初始的聚类中心之间的相互距离尽可能远**，这样可以避免出现上述的问题。K-Means++算法的初始化过程如下所示：

1.在数据集中随机选择一个样本点作为第一个初始化的聚类中心
2.选择出其余的聚类中心：
计算样本中的每一个样本点与已经初始化的聚类中心之间的距离，并选择其中最短的距离，记为d_i
选择一个新的数据点作为新的聚类中心，选择的原则是：距离较大的点，被选取作为聚类中心的概率较大
重复上述过程，直到k个聚类中心都被确定
3.对k个初始化的聚类中心，利用K-Means算法计算最终的聚类中心。

代码：

```
def nearest(data, cluster_centers, distance_func=euclidean_distance):
    min_dist = np.inf
    m = np.shape(cluster_centers)[0]  # 当前已经初始化的聚类中心的个数
    for i in range(m):
        d = distance_func(data, cluster_centers[i, ])  # 计算point与每个聚类中心之间的距离
        if min_dist &gt; d:  # 选择最短距离
            min_dist = d
    return min_dist

def get_centroids(data, k, distance_func=euclidean_distance):
    m, n = np.shape(data)
    cluster_centers = np.mat(np.zeros((k, n)))
    index = np.random.randint(0, m)  # 1、随机选择一个样本点为第一个聚类中心
    cluster_centers[0, ] = np.copy(data[index, ])
    d = [0.0 for _ in range(m)]  # 2、初始化一个距离的序列
    for i in range(1, k):
        sum_all = 0
        for j in range(m):
            d[j] = nearest(data[j, ], cluster_centers[0:i, ], distance_func)  # 3、对每一个样本找到最近的聚类中心点
            sum_all += d[j]  # 4、将所有的最短距离相加
        sum_all *= random()  # 5、取得sum_all之间的随机值
        for j, di in enumerate(d):  # 6、获得距离最远的样本点作为聚类中心点
            sum_all -= di
            if sum_all &gt; 0:
                continue
            cluster_centers[i] = np.copy(data[j, ])
            break
    return cluster_centers
```