一、 Birch 算法简介

BIRCH 的全称是利用层次方法的平衡迭代规约和聚类（Balanced Iterative Reducing and Clustering Using Hierarchies），它是一种非常有效的、传统的层次聚类算法，该算法能够用一遍扫描有效地进行聚类，并能够有效地处理离群点。Birch 算法是基于距离的层次聚类，综 合了层次凝聚和迭代的重定位方法，首先用自底向上的层次算法，然后用迭代的重定位来改进结果。层次凝聚是采用自底向上策略，首先将每个对象作为一个原子簇，然后合并这些 原子簇形成更大的簇，减少簇的数目，直到所有的对象都在一个簇中，或某个终结条件被满足。

二、 Birch 算法原理

BIRCH 算法利用了一个树结构来帮助我们快速的聚类，这个数结构类似于平衡 B+树，一般将它称之为聚类特征树(Clustering Feature Tree，简称 CF Tree)。这颗树的每一个节点是由若干个聚类特征(Clustering Feature，简称 CF)组成。从下图我们可以看看聚类特征树是什么样子的：每个节点包括叶子节点都有若干个 CF，而内部节点的 CF 有指向孩子节点的指针，所有的叶子节点用一个双向链表链接起来。

三、 聚类特征 CF 与聚类特征树 CF Tree

- **聚类特征 CF：**

CF 是 BIRCH 增量聚类算法的核心，CF 树中得节点都是由 CF 组成，一个 CF 是一个三元组，这个三元组就代表了簇的所有信息。

给定 N 个 d 维的数据点$\{x_1, x_2, \dots, x_n\}$，CF 定义为：CF=（N，LS，SS）其中，N 是子类中节点的数目，LS 是 N 个节点的线性和，SS 是 N 个节点的平方和。

CF 的特性是可以求和，例如：

CF_1=（n_1，LS_1，SS_1），CF_2=（n_2，LS_2，SS_2），

则$CF_1 + CF_2 = (n_1 + n_2, LS_1 + LS_2, SS_1 + SS_2)$

假如一个簇中包含 n 个数据点：$\{X_i\}$，i=1,2,3...n.，则定义

簇的质心 C：C=$\frac{X_1+X_2\ +...+X_n}{n}$，（$X_1+X_2\ +...+X_n$是向量加）

簇的半径 R：

$$\frac{|x_1 - c|^2 + |x_2 - c|^2 + \cdots + |x_n - c|^2}{n}$$

簇半径表示簇中所有点到簇质心的平均距离。CF 中存储的是簇中所有数据点的特性的统计和，所以当我们把一个数据点加入某个簇的时候，那么这个数据点的详细特征就会失，由于这个特征，BIRCH 聚类可以在很大程度上对数据集进行压缩。

- **CF Tree**

CF 树是一棵具有两个参数的高度平衡树，用来存储层次聚类的聚类特征。它涉及到两个参数分支因子和阈值。其中，分支因子 B 指定子节点的最大数目，即每个非叶节点可以拥有的孩子的最大数目。阈值 T 指定存储在叶节点的子簇的最大直径，它影响着 CF 树的大小。改变阈值可以改变树的大小。CF 树是随着数据点的插入而动态创建的，因此该方法是增量的。

要想构造一个 CF 树，需要按照如下流程：

1. 从根节点 root 开始递归往下，计算当前条目与要插入数据点之间的距离，寻找距 离最小的那个路径，直到找到与该数据点最接近的叶节点中的条目。
2. 比较计算出的距离是否小于阈值 T，如果小于则当前条目吸收该数据点；反之，则 继续第三步。
3. 判断当前条目所在叶节点的条目个数是否小于 L，如果是，则直接将数据点插入作 为该数据点的新条目，否则需要分裂该叶节点。分裂的原则是寻找该叶节点中距离最远的两 个条目并以这两个条目作为分裂后新的两个叶节点的起始条目，其他剩下的条目根据距离最 小原则分配到这两个新的叶节点中，删除原叶节点并更新整个 CF 树。
4. 当数据点无法插入时，这个时候需要提升阈值 T 并重建树来吸收更多的叶节点条目， 直到把所有数据点全部插入完毕。

四、 Birch 算法流程

1. 将所有的样本依次读入，在内存中建立一颗 CF Tree；
2. 将建立的 CF Tree 进行筛选，去除一些异常 CF 节点，对于一些超球体距离非常近的元组进行合并；
3. 利用其它的一些聚类算法比如 K-Means 对所有的 CF 元组进行聚类，得到一颗比较好的 CF Tree。主要目的是消除由于样本读入顺序导致的不合理的树结构，以及一些由于节点 CF 个数限制导致的树结构分裂。

4. 利用第三步生成的 CF Tree 的所有 CF 节点的质心，作为初始质心点，对所有的样本点按距离远近进行聚类。可以进一步减少由于 CF Tree 的一些限制导致的聚类不合理的情况。

步骤 2 是可选的，主要是因为步骤 3 对输入的即将进行全局聚类的聚簇的个数有要求（这里考虑到步骤 3 的全局聚类算法在小数据集上效果更好），所以步骤 2 主要扫描初始构建的 CF Tree 的叶子节点来重新构建一个更小的 CF Tree，同时会移除异常点并把相聚较近中的子类合并成更大的类。

步骤 3 针对数据中存在极大的分布不均衡情况，上述步骤 1 中的插入顺序不同也会导致分类效果不好，所以这里的步骤 3 使用全局或半全局聚类算法（层次聚类）来重新聚类所有的叶子节点（这里直接利用每个子簇的 CF 向量在聚簇级别进行重新聚类）。

步骤 4 是可选的，针对的是个别类簇中数据点的误放导致的结果不精确问题，使用步骤 3 中产生的聚类的中心点作为种子，重新分布数据点到最近的种子中，在这一步会最终生成每个数据点的聚类标签，并能剔除一些异常点。

五、 Birch 算法优缺点总结

优点

- 节省内存。叶子节点放在磁盘分区上，非叶子节点仅仅是存储了一个 CF 值，外加指向父节点和孩子节点的指针。
- 速度快。合并两个两簇只需要两个类簇的 CF 元组直接相加即可，计算

两个簇的距离只需要用到(N,LS,SS)这三个值。

- 一遍扫描数据库即可建立 CF Tree。
- 可识别噪声点。建立好 CF Tree 后把那些包含数据点少的 MinCluster 当作 outlier。
- 由于 CF Tree 是高度平衡的，所以在树上进行插入或查找操作很快。

缺点

- 结果依赖于数据点的插入顺序。本属于同一个簇的点可能由于插入顺序相差很远 而分到不同的簇中，即使同一个点在不同的时刻被插入，也会被分到不同的簇中。
- 对非球状的簇聚类效果不好。这取决于簇直径和簇间距离的计算方法。
- 对高维数据聚类效果不好。
- 由于每个节点只能包含一定数目的子节点，最后得出来的簇可能和自然簇相差很大。BIRCH 算法在整个过程中算法一旦中断，一切必须从头再来。
- 局部性导致了 BIRCH 的聚类效果欠佳。当一个新数据点要插入时，它只跟很少一部分簇进行了相似性（通过计算簇间距离）比较，效果不一定好。