# Spark KMeans源码分析

## 引言

本文重点介绍Spark MLlib中的KMeans源码及其实现原理。

KMeans是聚类算法中最为著名的方法之一。KMeans也是比较容易实现分布式处理的聚类算法之一。因为在每次迭代过程中判断每个点归属于哪一类是独立的。即，我们判断每个点归属于哪一个类的步骤中，和其他点是没有关系的，只需要找到距离该点最近的中心点。

## KMeans算法

KMeans聚类算法如下

### Lloyd-Forgy迭代

Step1.从D中随机取k个元素，作为k个簇的各自的中心。

step2.分别计算剩下的元素到k个簇中心的相异度，将这些元素分别划归到相异度最低的簇。

step3.根据聚类结果，重新计算k个簇各自的中心，计算方法是取簇中所有元素各自维度的算术平均数。

step4.将D中全部元素按照新的中心重新聚类。

step5.重复第4步，直到聚类结果不再变化。

优点：容易实现并行化；收敛快；

缺点：容易到局部最优解，容易出现个别类群过小或过大的情况

### Mac-queen迭代

具体步骤如下：

step1. 选取头K个点作为K个类的中心点

step2. 选取下一个点计算与K个中心点的距离，选取距离最小的分配到该类中

step3. 更新该类的中心点

step4. 重复2,3直到所有的点分配完毕

优点：是一种online算法

缺点：比Lloyd慢

### Hartigan-Wong 算法

### 

## Spark中KMeans源码概览

## 首先看一下源码中的说明。

/\*\*

\* K-means clustering with support for multiple parallel runs and a k-means++ like initialization

\* mode (the k-means|| algorithm by Bahmani et al). When multiple concurrent runs are requested,

\* they are executed together with joint passes over the data for efficiency.

\*

\* This is an iterative algorithm that will make multiple passes over the data, so any RDDs given

\* to it should be cached by the user.

\*/

该算法实现了KMeans的并行化版本以及一个基于并行化的kmeans++初始化算法。

由于迭代，data需要缓存。

## 再看一下主要结构

1. 一个定义主要运行方法的类

class KMeans private (  
 private var k: Int,  
 private var maxIterations: Int,  
 private var runs: Int,  
 private var initializationMode: String,  
 private var initializationSteps: Int,  
 private var epsilon: Double,  
 private var seed: Long) extends Serializable with Logging

1. 一个最终实现的伴随对象

**object** KMeans

1. 用于存储数据的基本类型

**class** VectorWithNorm(**val** vector: Vector, **val** norm: Double) **extends** Serializable {  
  
 **def this**(vector: Vector) = **this**(vector, Vectors.*norm*(vector, 2.0))  
  
 **def this**(array: Array[Double]) = **this**(Vectors.*dense*(array))  
  
 */\*\* Converts the vector to a dense vector. \*/* **def** toDense = **new** VectorWithNorm(Vectors.*dense*(vector.toArray), norm)  
}

1. 此外还用到

MLUtils.*fastSquaredDistance*

和

**private**[mllib] **object** LocalKMeans **extends** Logging

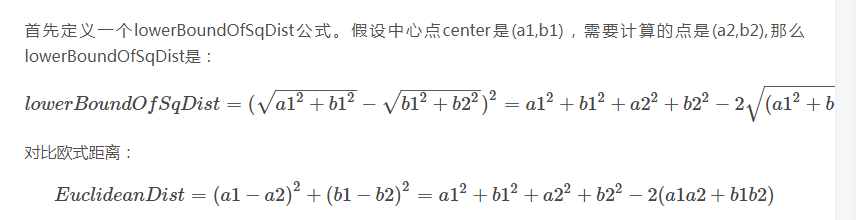
里面定义了Kmeans++的初始化方法。

## Spark KMeans中用到的基本原理补充

### KMeans基本原理

采用了基于Lloyd迭代的算法，但在求距离时略有不同，利用了距离上的基本不等式（区别于三角不等式）。

### 基本不等式



利用基本不等式可以产生如下的优化规则：

当l(x, c) > l(x, c(x))时可以不用计算欧式距离

同时由于优化的需要，以空间换时间，data除了存储数据点之外，还需要存储二阶范式。同样聚类中心除了需要存储数据点之外，也需要存储二阶范式。对应的数据结构为VectorWithNorm类。

### Runs

此处runs的作用是每次进行多组，最终从进行了多组K均值中选最小均方差的一组作为最终结果。

Runs设计的目的主要是避免kmeans算法对初始值的依赖。如果不考虑这些，仅考虑kmeans的分布式实现，初次阅读源码runs会产生一定的干扰。这里大家记住每个runs间是独立的，runs只是将中心增加了一个维度（每次迭代中，由原有的k个中心变为了k\*runs个中心）。

## Spark KMeans源码

### KMeans的Lloyd迭代过程

1. Run函数

def run(data: RDD[Vector]): KMeansModel = {  
// 判断数据是否cache  
 if (data.getStorageLevel == StorageLevel.NONE) {  
 logWarning("The input data is not directly cached, which may hurt performance if its"  
 + " parent RDDs are also uncached.")  
 }  
  
 // 计算二阶范式，将数据装进VectorWithNorm中。  
 val norms = data.map(Vectors.norm(\_, 2.0))  
 norms.persist()  
 val zippedData = data.zip(norms).map { case (v, norm) =>  
 new VectorWithNorm(v, norm)  
 }

//主运行函数runAlgorithm  
 val model = **runAlgorithm**(zippedData)  
 norms.unpersist()  
  
 // Warn at the end of the run as well, for increased visibility.  
 if (data.getStorageLevel == StorageLevel.NONE) {  
 logWarning("The input data was not directly cached, which may hurt performance if its"  
 + " parent RDDs are also uncached.")  
 }  
 model  
}

1. runAlgorithm函数

private def runAlgorithm(data: RDD[VectorWithNorm]): KMeansModel = {  
 val sc = data.sparkContext  
 val initStartTime = System.nanoTime()

**//step1.初始化**  
**// step1-1. centers初始化，这里有两种初始化方式：完全随机初始化和Kmean||初始化。该部分后面会详解一遍。此处仅需知道centers类型为：Array[Array[VectorWithNorm]]即runs\*k个中心点。**

val centers = if (initializationMode == KMeans.RANDOM) {  
 initRandom(data)  
 } else {  
 initKMeansParallel(data)  
 }  
  
 val initTimeInSeconds = (System.nanoTime() - initStartTime) / 1e9  
 logInfo(s"Initialization with $initializationMode took " + "%.3f".format(initTimeInSeconds) +  
 " seconds.")  
**// step1-2. Active初始化用来记录每个runs的收敛状态，如果收敛了则该runs不再继续迭代。初始化的**activeRuns有runs个，之后会以可变变量的形式在个数上发生变化。  
 val active = Array.fill(runs)(true)  
 val costs = Array.fill(runs)(0.0)  
 var activeRuns = new ArrayBuffer[Int] ++ (0 until runs)  
 var iteration = 0  
  
 val iterationStartTime = System.nanoTime()

**//step2 Llyod迭代**  
 // Execute iterations of Lloyd's algorithm until all runs have converged  
 while (iteration < maxIterations && !activeRuns.isEmpty) {  
 type WeightedPoint = (Vector, Long)  
 def mergeContribs(x: WeightedPoint, y: WeightedPoint): WeightedPoint = {  
 axpy(1.0, x.\_1, y.\_1)  
 (y.\_1, x.\_2 + y.\_2)  
 }  
**//step2-1**选出仍在跑的run对应的center  
 val activeCenters = activeRuns.map(r => centers(r)).toArray

**//step2-1**每次迭代过程中每个runs的sse初始化为0.0以后foreach累加。  
 val costAccums = activeRuns.map(\_ => sc.accumulator(0.0))  
  
 val bcActiveCenters = sc.broadcast(activeCenters)

**//step2-2 遍历每个分区中的数据找到最近的中心点，并记录一些迭代信息。**  
 // Find the sum and count of points mapping to each center  
 val totalContribs = data.mapPartitions { points =>  
 val thisActiveCenters = bcActiveCenters.value  
 val runs = thisActiveCenters.length  
 val k = thisActiveCenters(0).length  
 val dims = thisActiveCenters(0)(0).vector.size  
  
 val sums = Array.fill(runs, k)(Vectors.zeros(dims))

// 用来记录个数  
 val counts = Array.fill(runs, k)(0L)  
  
 points.foreach { point =>

// 在每个runs中对每个数据点找最近的中心点。  
 (0 until runs).foreach { i =>  
 val (bestCenter, cost) = KMeans.findClosest(thisActiveCenters(i), point)

**// 记录迭代中每个runs的ss**   
 costAccums(i) += cost

**// 记录迭代中每个runs中每个中心id的所有点的和以及个数。**  
 val sum = sums(i)(bestCenter)  
 axpy(1.0, point.vector, sum)  
 counts(i)(bestCenter) += 1  
 }  
 }

**// 抹平方便输出为Map形式**  
  
 val contribs = for (i <- 0 until runs; j <- 0 until k) yield {  
 ((i, j), (sums(i)(j), counts(i)(j)))  
 }  
 contribs.iterator

// merge，此处merge其实是，(sum+sum, count+count)  
 }.reduceByKey(mergeContribs).collectAsMap()

**//step2-3 更新中心点和sse**  
 // Update the cluster centers and costs for each active run  
 for ((run, i) <- activeRuns.zipWithIndex) {  
 var changed = false  
 var j = 0  
 while (j < k) {  
 val (sum, count) = totalContribs((i, j))  
 if (count != 0) {

// 将sum除以count求均值，求新的中心  
 scal(1.0 / count, sum)  
 val newCenter = new VectorWithNorm(sum)

// 如果某个run的所有的新旧中心点的移动都小于epsilon \* epsilon那么run的状态依旧是false.否则changed为true，更新迭代点，并在下次while循环时继续迭代。  
 if (KMeans.fastSquaredDistance(newCenter, centers(run)(j)) > epsilon \* epsilon) {  
 changed = true  
 }  
 centers(run)(j) = newCenter  
 }  
 j += 1  
 }  
 if (!changed) {  
 active(run) = false  
 logInfo("Run " + run + " finished in " + (iteration + 1) + " iterations")  
 }

//记录每个run最后一次迭代的sse用于最后的模型选择。  
 costs(run) = costAccums(i).value  
 }

// 更新activeRuns  
 activeRuns = activeRuns.filter(active(\_))  
 iteration += 1  
 }  
  
 val iterationTimeInSeconds = (System.nanoTime() - iterationStartTime) / 1e9  
 logInfo(s"Iterations took " + "%.3f".format(iterationTimeInSeconds) + " seconds.")  
  
 if (iteration == maxIterations) {  
 logInfo(s"KMeans reached the max number of iterations: $maxIterations.")  
 } else {  
 logInfo(s"KMeans converged in $iteration iterations.")  
 }

**// Step3模型选择**  
 val (minCost, bestRun) = costs.zipWithIndex.min  
  
 logInfo(s"The cost for the best run is $minCost.")  
  
 new KMeansModel(centers(bestRun).map(\_.vector))  
}

### KMeans的初始化过程

### 完全随机

private def initRandom(data: RDD[VectorWithNorm])  
: Array[Array[VectorWithNorm]] = {  
 // Sample all the cluster centers in one pass to avoid repeated scans  
 val sample = data.takeSample(true, runs \* k, new XORShiftRandom(this.seed).nextInt()).toSeq  
 Array.tabulate(runs)(r => sample.slice(r \* k, (r + 1) \* k).map { v =>  
 new VectorWithNorm(Vectors.dense(v.vector.toArray), v.norm)  
 }.toArray)  
}

### KMeans||

1. 请做好准备，这里较为复杂。

KMeans++和KMeans||很相似

1. KMeans++和KMeans||

KMeans||相比于KMeans++的优势在于数据集很大，并且当k也很大时，KMeans++每次迭代中都需要计算每个点到迭代中心的概率，计算k次花费比较大。KMeans||可以每次迭代中以对应概率取多个点，从而在log(k)次迭代完成。

1. KMeans||算法

private def initKMeansParallel(data: RDD[VectorWithNorm])  
 : Array[Array[VectorWithNorm]] = {  
 // Initialize empty centers and point costs.  
 val centers = Array.tabulate(runs)(r => ArrayBuffer.empty[VectorWithNorm])  
 var costs = data.map(\_ => Vectors.dense(Array.fill(runs)(Double.PositiveInfinity))).cache()  
  
 // Initialize each run's first center to a random point.  
 val seed = new XORShiftRandom(this.seed).nextInt()  
 val sample = data.takeSample(true, runs, seed).toSeq  
 val newCenters = Array.tabulate(runs)(r => ArrayBuffer(sample(r).toDense))  
  
 /\*\* Merges new centers to centers. \*/  
 def mergeNewCenters(): Unit = {  
 var r = 0  
 while (r < runs) {  
 centers(r) ++= newCenters(r)  
 newCenters(r).clear()  
 r += 1  
 }  
 }  
  
 // On each step, sample 2 \* k points on average for each run with probability proportional  
 // to their squared distance from that run's centers. Note that only distances between points  
 // and new centers are computed in each iteration.  
 var step = 0  
 while (step < initializationSteps) {  
 val bcNewCenters = data.context.broadcast(newCenters)  
 val preCosts = costs  
 costs = data.zip(preCosts).map { case (point, cost) =>  
 Vectors.dense(  
 Array.tabulate(runs) { r =>  
 math.min(KMeans.pointCost(bcNewCenters.value(r), point), cost(r))  
 })  
 }.cache()  
 val sumCosts = costs  
 .aggregate(Vectors.zeros(runs))(  
 seqOp = (s, v) => {  
 // s += v  
 axpy(1.0, v, s)  
 s  
 },  
 combOp = (s0, s1) => {  
 // s0 += s1  
 axpy(1.0, s1, s0)  
 s0  
 }  
 )  
 preCosts.unpersist(blocking = false)  
 val chosen = data.zip(costs).mapPartitionsWithIndex { (index, pointsWithCosts) =>  
 val rand = new XORShiftRandom(seed ^ (step << 16) ^ index)  
 pointsWithCosts.flatMap { case (p, c) =>  
 val rs = (0 until runs).filter { r =>  
 rand.nextDouble() < 2.0 \* c(r) \* k / sumCosts(r)  
 }  
 if (rs.length > 0) Some(p, rs) else None  
 }  
 }.collect()  
 mergeNewCenters()  
 chosen.foreach { case (p, rs) =>  
 rs.foreach(newCenters(\_) += p.toDense)  
 }  
 step += 1  
 }  
  
 mergeNewCenters()  
 costs.unpersist(blocking = false)  
  
 // Finally, we might have a set of more than k candidate centers for each run; weigh each  
 // candidate by the number of points in the dataset mapping to it and run a local k-means++  
 // on the weighted centers to pick just k of them  
 val bcCenters = data.context.broadcast(centers)  
 val weightMap = data.flatMap { p =>  
 Iterator.tabulate(runs) { r =>  
 ((r, KMeans.findClosest(bcCenters.value(r), p).\_1), 1.0)  
 }  
 }.reduceByKey(\_ + \_).collectAsMap()  
 val finalCenters = (0 until runs).par.map { r =>  
 val myCenters = centers(r).toArray  
 val myWeights = (0 until myCenters.length).map(i => weightMap.getOrElse((r, i), 0.0)).toArray  
 LocalKMeans.kMeansPlusPlus(r, myCenters, myWeights, k, 30)  
 }  
  
 finalCenters.toArray  
 }  
}

## 收获

### 在同时监督多个状态是可以加入一个Array[Boolean]的active数组

### 确定所有的都满足条件A可以变为：

Flag = true

While{

If(A) flag = false

}

另外如果用到每隔一次进行一次操作f()的场景（如geohash经纬度的二进制数需要交替）可以

var flag = ture

while(flag){

f()

var flag = !flag

}

### Array等Object中有很多自定义方法

1. 创建Array

Array.fill()()

Array.tabulate()()

Array.empty

Array.ofDim

1. range和concat

val arr1 = Array.range(0, 5)

val arr2 = Array.range(0, 10, 2)

val u = Array.concat(arr1, arr2)

1. 将一个iterater顺序输出至一个arr

// 当然这是没事找抽，直接toArray就好

val iter = {for(i <- 0 until 5) yield 3\*i}.toIterator

val newArr = Array.empty[Int] ++ iter

newArr.foreach(println)

1. tabulate

// 默认从0开始对每个索引id按一定map方式构造元素数组

Array.tabulate(10)(i => Array.fill(i + 1)(0.0)).foreach(x => println(x.length))

// 打印方形乘法表

val multiTable = Array.tabulate(9, 9)((i, j) => f"${j + 1}%2d \* ${i + 1}%2d = ${(i + 1)\*(j + 1)}%2d")

multiTable.foreach(arr => println(arr.mkString(" ")))

// 打印三角形乘法表，这个不能只根据这几个函数实现

val triMultiTable = Array.tabulate(9)(i => Array.range(1, i + 2).map(j => f"$j%2d \* ${i + 1}%2d = ${(i + 1) \* j}%2d"))

triMultiTable.foreach(arr => println(arr.mkString(" ")))

1. 上述大部分函数其他的Seq对象也有，比如如ArrayBuffer等等

不在赘述。