Spark MLlib StreamingKmeans实时KMeans聚类算法源代码解读

最近花了一段时间看了一下聚类算法中的StreamingKMeans算法。在Spark MLlib的聚类算法中实现多个聚类算法，KMeans也包含多个版本，其中一个是通过Spark平台来S实现Kmeans，还有一个是通Spark-streaming平台来实现Kmeans。这个类似于数据通过一个实时平台比如说Flume，或者是Kafk或者是通过Socket来发送到Spark-Streaming，然后Spark-Streaming来进行实时的数据聚类。

关于Kmeans的思想，可以参考我之前的一篇博客。
http://blog.csdn.net/stevekangpei/article/details/73380638

Spark-Streaming实现实时聚类的方法还是比较简单的。在StreamingKMeans的代码注释里面包含了样一段话。

```
/**
 * StreamingKMeansModel extends MLlib's KMeansModel for streaming
 * algorithms, so it can keep track of a continuously updated weight
 * associated with each cluster, and also update the model by
 * doing a single iteration of the standard k-means algorithm.
 *
 *
 * The update algorithm uses the "mini-batch" KMeans rule,
 * generalized to incorporate forgetfullness (i.e. decay).
 * The update rule (for each cluster) is:
 *
 * {{{
 * c_t+1 = [(c_t * n_t * a) + (x_t * m_t)] / [n_t + m_t]
 * n_t+t = n_t * a + m_t
 * }}}
 *
 * Where c_t is the previously estimated centroid for that cluster,
 * n_t is the number of points assigned to it thus far, x_t is the centroid
 * estimated on the current batch, and m_t is the number of points assigned
 * to that centroid in the current batch.
 *
 * The decay factor 'a' scales the contribution of the clusters as estimated thus far,
 * by applying a as a discount weighting on the current point when evaluating
```

```
 * new incoming data. If a=1, all batches are weighted equally. If a=0, new centroids
 * are determined entirely by recent data. Lower values correspond to
 * more forgetting.
 *
 * Decay can optionally be specified by a half life and associated
 * time unit. The time unit can either be a batch of data or a single
 * data point. Considering data arrived at time t, the half life h is defined
 * such that at time t + h the discount applied to the data from t is 0.5.
 * The definition remains the same whether the time unit is given
 * as batches or points.
 */
```

```
StreamingKMeansModel继承自MLlib的KMeansModel来作为实时处理的算法。因此它可以持续的追踪着和每一个cluste

更新算法采用“mini-batch” KMeans 方法，同时也包含了消失因子(decay).
更新的法则如下。

{{{
c_t+1 = [(c_t * n_t * a) + (x_t * m_t)] / [n_t + m_t]
n_t+t = n_t * a + m_t
}}}

其中c_t 是之前估计的cluster的中心点，n_t表示的是这个中心点的点的个数。x_t是现在当前这个batch的中心点，
那么可以理解，c_t+1就是当前处理后的中心点，n_t+t表示的就是当前中心点的周边的点的个数。

dacay因子 a 表示的是之前的数据集的下降因子。如果a = 1 的话，表示的是所有的batch的数据的权重都 是一样的
越多的下降因子。

decay因子可以通过一个 time unit 来指定，time unit 可以是一个 数据的batch，也可以是一个单个的数据点。如
```

实时Kmeans聚类算法最重要的方法是实现了这个注释中的那个数学公式来进行聚类。这个算法在upd方法里面。StreamingKMeans有两个类，一个是StreamingKMeansModel，另一个是StreamingKMeans。

```
class StreamingKMeansModel @Since("1.2.0") (
    @Since("1.2.0") override val clusterCenters: Array[Vector],
    @Since("1.2.0") val clusterWeights: Array[Double])
  extends KMeansModel(clusterCenters) with Logging {
```

```
//第一个参数clusterCenters表示的是数据的聚类中心点，每个元素为Vector。
//第二个参数指的是clusterWeights，通过对上面的算法的分析，我认为这个参数表示的是每个数据中心点这个类别
```

接下来是一个update方法，实时聚类算法也是在这个方法中实现的。

```
def update(data: RDD[Vector], decayFactor: Double, timeUnit: String): StreamingKMeansModel = {

  // find nearest cluster to each point
  val closest = data.map(point => (this.predict(point), (point, 1L)))
  //这个表示的是通过当前的model找到现在这个batch数据集中地这个点所在的聚类中心的index，
  //同时返回一个元祖格式的数据(index: Int, (point: Vector, 1: Long))。
  //这个表示的是当前这个batch数据里面的每个数据所属的类别，和这个点出现的次数，
  //注意我们之前的那个注释，这个算法中需要用到每个中心点的周围的点的个数。

  // get sums and counts for updating each cluster
  //这里定义了一个函数，这个函数对两个(Vector, Long)类型的元祖进行一个合并操作，
  //返回的类型也是一个元祖的类型。
  val mergeContribs: ((Vector, Long), (Vector, Long)) => (Vector, Long) = (p1, p2) => {
    BLAS.axpy(1.0, p2._1, p1._1)
    //这个做一个向量的加法操作，
    //表示p1._1 = p1._1 + 1.0 * p2._1
    (p1._1, p1._2 + p2._2) //然后返回的是(两个向量的和，两个向量出现的次数之和)。
  }
  val dim = clusterCenters(0).size //这个表示的是聚类中心点的维度的数目。

  val pointStats: Array[(Int, (Vector, Long))] = closest
    .aggregateByKey((Vectors.zeros(dim), 0L))(mergeContribs, mergeContribs)
    .collect()

    //这个里面用到了一个aggregateByKey算子，这个算子在我之前的博客中讲过，
    //第一个参数表示的是zeroValue，初始值，这个会作用到第一个函数但是不会作用到第二个函数。
    //第一个函数值得是对每个分区进行merge操作，就是我们之前定义的函数，
    //第二个表示的是对每个分区作用后的结果进行操作。
    //最后返回的结果通过collect返回一个数组,每个元素的类型为(Int(表示这个所有batch的数据出现在第int个
    //(表示的是出现在这个中心点附近的所有的向量的矢量和，出现的向量的次数))

  //这个表示的是用来判断discount 的类型，是batch类型还是points类型，如果是batch类型，则直接使用decayF
  val discount = timeUnit match {
```

```
        case StreamingKMeans.BATCHES => decayFactor
        case StreamingKMeans.POINTS =>
          val numNewPoints = pointStats.view.map { case (_, (_, n)) =>
            n
          }.sum
          math.pow(decayFactor, numNewPoints)
      }

      // apply discount to weights
      BLAS.scal(discount, Vectors.dense(clusterWeights))
      // 这个计算好的discount的值被用来scale之前cluster的权重，这个和之前的第一个公式的第一部分是类似的。
      //c_t+1 = [(c_t * n_t * a) + (x_t * m_t)] / [n_t + m_t]
     //即将decay值作用于n_t(之前点出现的次数)。

      // implement update rule
      //然后是实现这个update算法
      pointStats.foreach { case (label, (sum, count)) =>
        val centroid = clusterCenters(label) //获取对应的中心点，
        val updatedWeight = clusterWeights(label) + count
        //更新现在的权重值，其实指的就是n_t + m_t

        val lambda = count / math.max(updatedWeight, 1e-16)
        //将count/updatedWeight作为现在的lamda值。这个表示的就是

        clusterWeights(label) = updatedWeight //同时更新现在的clusterWeights数组。
        BLAS.scal(1.0 - lambda, centroid)
        BLAS.axpy(lambda / count, sum, centroid)
        //这个指的就是对所有的点的向量和除以总的出现的次数来作为新的centroid。

        // display the updated cluster centers
        //接下来打印出聚类中心点。
        val display = clusterCenters(label).size match {
          case x if x > 100 => centroid.toArray.take(100).mkString("[", ",", "...")
          case _ => centroid.toArray.mkString("[", ",", "]")
        }

        logInfo(s"Cluster $label updated with weight $updatedWeight and centroid: $display")
      }

      // Check whether the smallest cluster is dying. If so, split the largest cluster.
      //接下来判断有没有哪个点正在消失，如果消失的话，那么歼最大的cluster进行分开。
      val weightsWithIndex = clusterWeights.view.zipWithIndex
```

```
      //这个表示将每个数据集和它出现的索引zip起来。然后找到最大的和最小的数据聚类。
    val (maxWeight, largest) = weightsWithIndex.maxBy(_._1)

    val (minWeight, smallest) = weightsWithIndex.minBy(_._1)
    if (minWeight < 1e-8 * maxWeight) {
    //如果最小的太小的话
      logInfo(s"Cluster $smallest is dying. Split the largest cluster $largest into two.")
      val weight = (maxWeight + minWeight) / 2.0
       //将大数据聚类进行拆分，然后重新分给小数据聚类和大数据聚类。
      clusterWeights(largest) = weight
      clusterWeights(smallest) = weight
      val largestClusterCenter = clusterCenters(largest)
      val smallestClusterCenter = clusterCenters(smallest)
      var j = 0
      while (j < dim) {
        val x = largestClusterCenter(j)
        val p = 1e-14 * math.max(math.abs(x), 1.0)
        largestClusterCenter.toBreeze(j) = x + p
        smallestClusterCenter.toBreeze(j) = x - p
        j += 1
      }
    }

    this  //最后返回这个新的模型。
  }
}
```

这个基本的类包含的是StreamingKmeansModel的初始化。
k表示的是中心点的个数，decayFactor表示的是下降因子，timeUnit有两种类型为batch和points。

```
class StreamingKMeans @Since("1.2.0") (
    @Since("1.2.0") var k: Int,
    @Since("1.2.0") var decayFactor: Double,
    @Since("1.2.0") var timeUnit: String) extends Logging with Serializable {
```

然后也包含一些predict方法。

```
  @Since("1.4.0")
  def predictOn(data: JavaDStream[Vector]): JavaDStream[java.lang.Integer] = {
    JavaDStream.fromDStream(predictOn(data.dstream).asInstanceOf[DStream[java.lang.Integer]])
  }

  /**
   * Use the model to make predictions on the values of a DStream and carry over its keys.
   *
   * @param data DStream containing (key, feature vector) pairs
   * @tparam K key type
   * @return DStream containing the input keys and the predictions as values
   */
  @Since("1.2.0")
  def predictOnValues[K: ClassTag](data: DStream[(K, Vector)]): DStream[(K, Int)] = {
    assertInitialized()
    data.mapValues(model.predict)
  }

  /**
   * Java-friendly version of `predictOnValues`.
   */
  @Since("1.4.0")
  def predictOnValues[K](
      data: JavaPairDStream[K, Vector]): JavaPairDStream[K, java.lang.Integer] = {
    implicit val tag = fakeClassTag[K]
    JavaPairDStream.fromPairDStream(
      predictOnValues(data.dstream).asInstanceOf[DStream[(K, java.lang.Integer)]])
  }
```