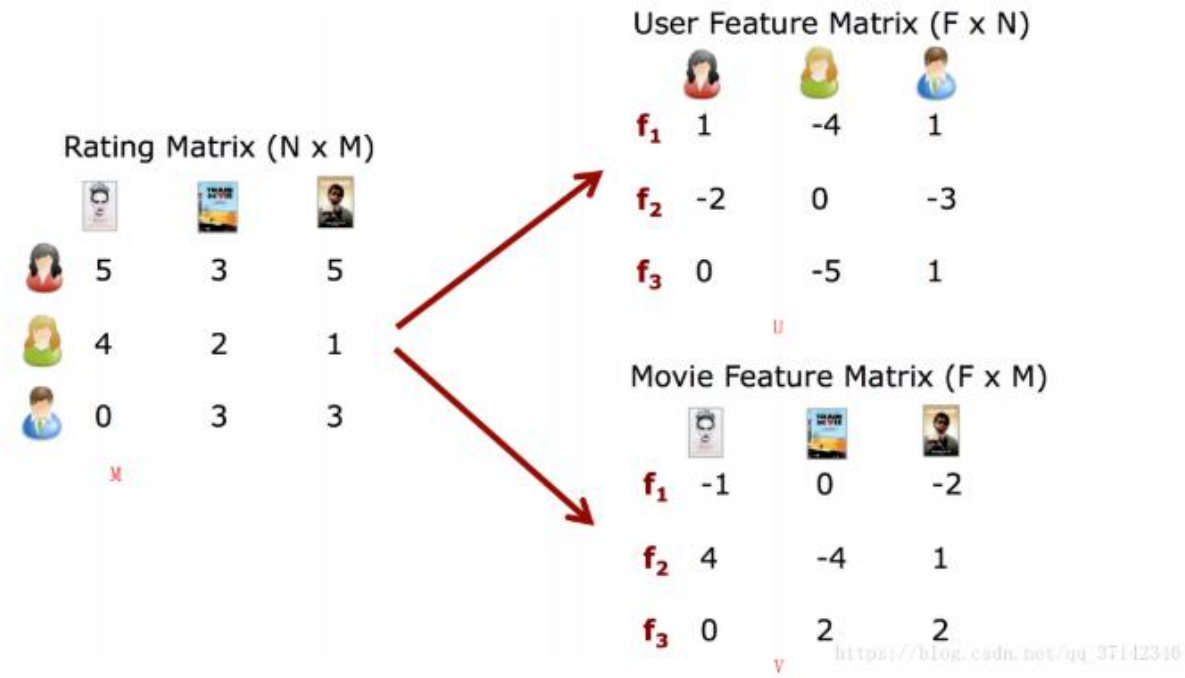
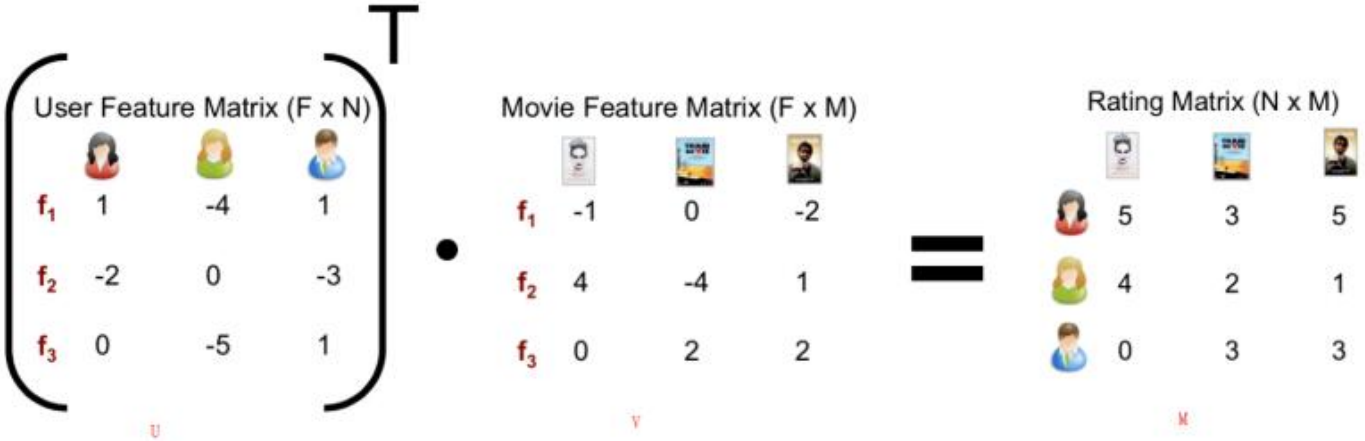
1. 前言

Spark 平台推出至今已经地带到 2.X 的版本了，很多地方都有了重要的更新，加入了很多新的东西。但是在协同过滤这一块却一直以来都只有 **ALS** 一种算法。同样是大规模计算平台，Hadoop 中的机器学习算法库 Mahout 就集成了多种推荐算法，不但有 user-cf 和 item-cf 这种经典算法，还有 KNN、SVD，Slope one 这些，可谓随意挑选，简繁由君。我们知道得是，推荐系统这个应用本身并没有过时，那么 spark 如此坚定地只维护一个算法，肯定是有他的理由的，让我们来捋一捋。

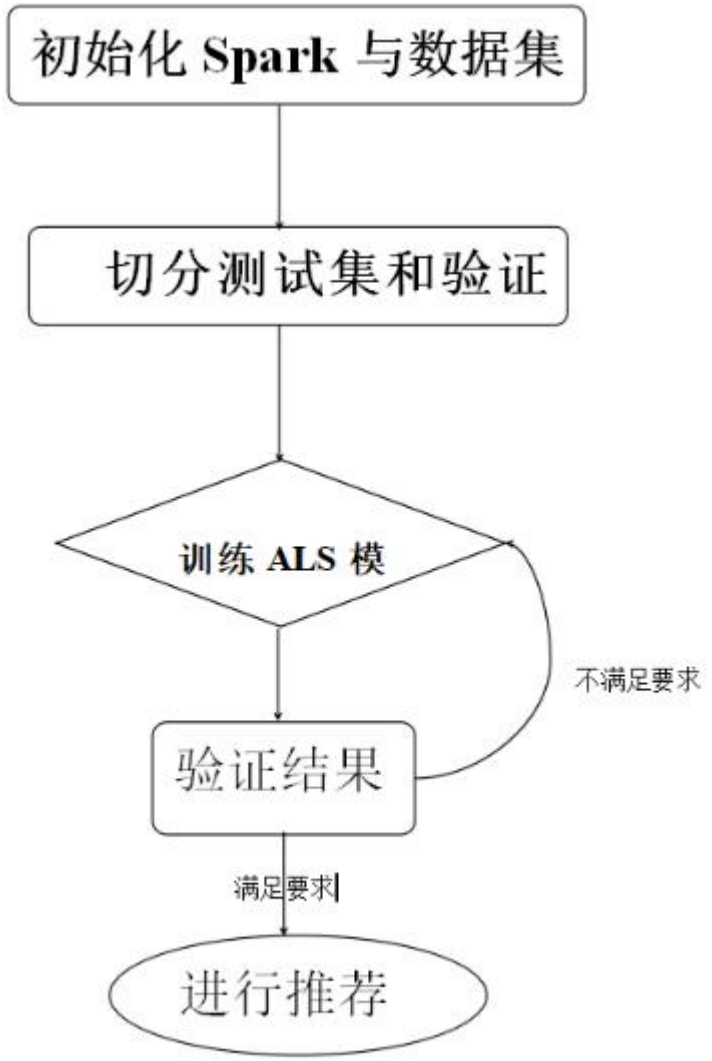
以用户，商品为例说明，一个基于用户名，物品表的用户评分矩阵可以被分解成 2 个较为小型化的矩阵，即 M=U 的转置矩阵*V 。





在这里 U 和 V 分别表示 用户和物品的矩阵，在 MLlib 的 ALS 算法中，首先会对 U 或者 V 矩阵**随机生成**，之后固定某一个特定的对象，去求取另一个未随机化的矩阵对象。之后利用求取的矩阵对象去求随机化矩阵对象。最后两个对象相互迭代计算，**求取与实际矩阵差异达到程序设定的最小阀值位**

置。通俗的说就是**先固定 U 矩阵，求取 V，然后再固定 V 矩阵再求取 U 矩阵**，一直这样交替迭代计算直到误差达到一定的阀值条件或者达到迭代次数的上限。具体流程步骤如下所示：



ALS 算法流程图

2. 实战推荐代码

2.1 余弦相似度

```
import org.apache.spark.metrics.source
import org.apache.spark.{SparkConf, SparkContext}

/**
  * 余弦相似度
  */
object ConsineSimilar {
```

```
val conf=new SparkConf()
   .setAppName("ConsineSimilar")
   .setMaster("local")

val sc=new SparkContext(conf)

//实例化环境
val users=sc.parallelize(Array("aaa","bbb","ccc","ddd","eee"))
//设置电影名
val films=sc.parallelize(Array("smzdm","yixb","znh","nhsc","fcwr"))
//使用一个 source 嵌套 map 作为姓名电影名和分值的存储
var source=Map[String,Map[String,Int]]()
//设置一个用以存放电影分的 map
val filmSource=Map[String,Int]()

/**
   *  设置电影评分
   * @return
   */
def getSource():Map[String,Map[String,Int]] = {

   val user1FilmSource= Map("smzdm" -> 2,"yixb" -> 3,"znh" -> 1,"nhsc" -> 0,"fcwr" -> 1)

   val user2FilmSource= Map("smzdm" -> 1,"yixb" -> 2,"znh" -> 2,"nhsc" -> 1,"fcwr" -> 4)

   val user3FilmSource= Map("smzdm" -> 2,"yixb" -> 1,"znh" -> 0,"nhsc" -> 1,"fcwr" -> 4)

   val user4FilmSource= Map("smzdm" -> 3,"yixb" -> 2,"znh" -> 0,"nhsc" -> 5,"fcwr" -> 3)

   val user5FilmSource= Map("smzdm" -> 5,"yixb" -> 3,"znh" -> 1,"nhsc" -> 1,"fcwr" -> 2)

   source += ("aaa" -> user1FilmSource)       //对人名进行存储
   source += ("bbb" -> user2FilmSource)       //对人名进行存储
```

```
    source += ("ccc" -> user3FilmSource)       //对人名进行存储
    source += ("ddd" -> user4FilmSource)       //对人名进行存储
    source += ("eee" -> user5FilmSource)       //对人名进行存储

    source       //返回 map
  }

  /**
    *  计算余弦相似性
    * @param user1
    * @param user2
    * @return
    */
  def getCollaborateSource(user1:String,user2:String): Double ={
    //获得第一个用户的评分
    val user1FilmSource = source.get(user1).get.values.toVector
    //获得第二个用户的评分
    val user2FileSource = source.get(user2).get.values.toVector
    //对欧几里得公式分子部分进行计算
    val member =
user1FilmSource.zip(user2FileSource).map(num=>num._1*num._2).reduce(_+_).toDouble
    //求出分母第一个变量的值
    val temp1 = math.sqrt(user1FilmSource.map(num => {
      math.pow(num, 2)
    }).reduce(_+_)).toDouble
    //求出分母第二个变量的值
    val temp2 = math.sqrt(user2FileSource.map(num => {
      math.pow(num,2)
    }).reduce(_+_)).toDouble
    //求出分母
    val denominator = temp1*temp2
    //返回结果
    member/denominator
```

```
  }

  def main(args: Array[String]): Unit = {
    //初始化分数
    getSource()
    //设定目标对象
    val name="bbb"
    //迭代进行计算
    users.foreach(user=>{
      println(name + "  相对于  "+user +"的相似性分数为: "+getCollaborateSource(name,user))
    })
    val frist=users.sortBy((user=>getCollaborateSource(name,user)),false,1).first()
    println("----------------------------------------------------------")
    println("相似度最高的用户为：  "+frist)

    /**
      * 计算结果如下：
      * bbb  相对于  aaa 的相似性分数为: 0.7089175569585667
        bbb  相对于  bbb 的相似性分数为: 1.0000000000000002
        bbb  相对于  ccc 的相似性分数为: 0.8780541105074453
        bbb  相对于  ddd 的相似性分数为: 0.6865554812287477
        bbb  相对于  eee 的相似性分数为: 0.6821910402406466
      */

  }
}
```

2.2 商品推荐

以商品推荐为例进行，使用的数据按照 MLlib 中 ALS 算法固有的数据格式。其中源码如下：

```
case class Rating @Since("0.8.0") (
    @Since("0.8.0") user: Int,
```

```
    @Since("0.8.0") product: Int,
    @Since("0.8.0") rating: Double)
```

这里用户和商品分别用代号表示，然后 rating 就是用户对于该商品的评分。因此数据集格式如下：

```
1 11 2
1 12 3
1 13 1
1 14 0
1 15 1
2 11 1
.....
```

ALS 算法的第二步就是数据建模，其实在 MLlib 算法库中有可以直接使用的训练算法，ALS.tran 方法源码如下：

```
def train(
      ratings: RDD[Rating],          //需要训练的数据集
      rank: Int,                     //模型中隐藏因子数
      iterations: Int,               //算法中迭代次数
      lambda: Double,                //ALS 中的正则化参数
      blocks: Int,                   //并行计算的 block 数(-1 为自动配置)
      alpha: Double,                   //ALS 隐式反馈变化率用于控制每次拟合修正的幅度(置信参数)
      seed: Long                       //加载矩阵的随机数
    ): MatrixFactorizationModel = {
    new ALS(blocks, blocks, rank, iterations, lambda, true, alpha, seed).run(ratings)
  }
```

在了解了如何建立模型之后就可以编写代码进行实战了，如下所示：

```
import org.apache.spark.mllib.recommendation.{ALS, Rating}
import org.apache.spark.{SparkConf, SparkContext}

object CollaborativeFilter {
  def main(args: Array[String]): Unit = {
    val conf=new SparkConf()
                .setAppName("CollaborativeFilter")
                .setMaster("local")
    val sc=new SparkContext(conf)
```

```
    //设置数据集
    val data=sc.textFile("..\\d.txt")
    //处理数据
    val ratings=data.map(_.split(" ") match {
      case Array(user,item,rate) =>                                    //转化数据集
        Rating(user.toInt,item.toInt,rate.toDouble)                    //将数据集转化为专用的 Rating
    })
    val rank=2                        //设置隐藏因子

    val numIterations=2          //设置迭代次数

    val model=ALS.train(ratings,rank,numIterations,0.01)          //进行模型训练

    val rs=model.recommendProducts(2,1)                                  //为用户 2 推荐一个商品

    rs.foreach(println)                                                  //打印推荐结果

    val result:Double=rs(0).rating                                       //预测的评分结果

    val realilty=data.map(_.split(" ") match {
      case Array(user,item,rate) =>
        Rating(user.toInt,item.toInt,rate.toDouble)
    }).map(num=>{
      if(num.user==2&&num.product==15)
        num.rating                                                       //返回实际评分结果
      else
        0
    }).foreach(num=>{
      if(num!=0)
       println("对 15 号商品预测的准确率为："+(1-(math.abs(result-num)/1)))
    })
  }
}
```

上面代码在建立好模型之后，然后使用 recommendProducts 方法为第二个用户推荐一个物品，程序结果如下所示：

```
Rating(2,15,3.97662718480575)
对 15 号商品预测的准确率为：0.9941567962014375
```

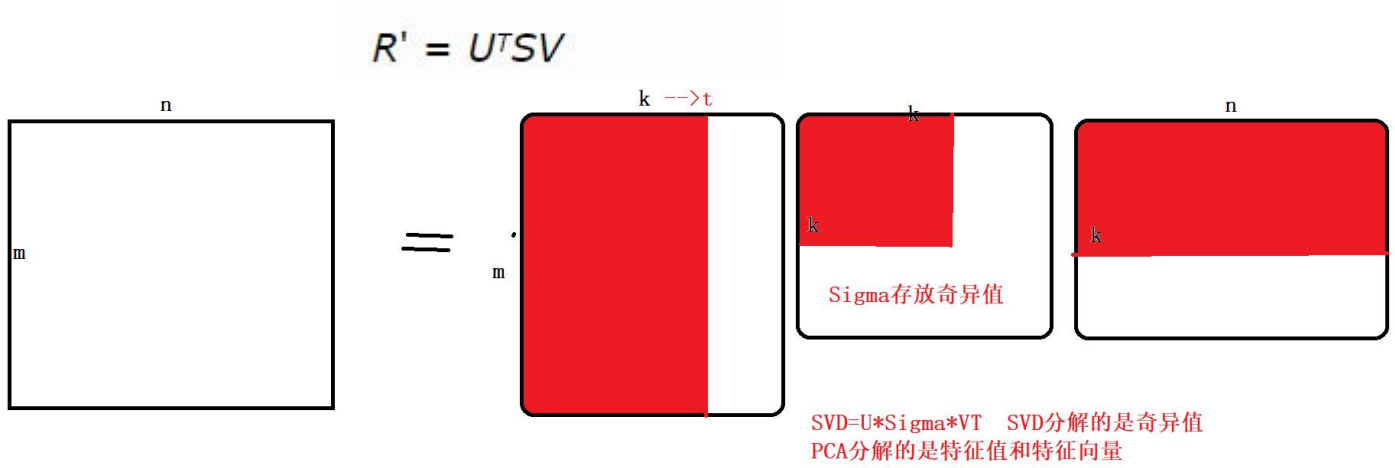
根据结果显示，为第二个用户优先推荐了编号为 15 的物品，同时预测的评分为 3.97662718480575，而实际值为 4，准确率为 0.9941567962014375，说明推荐的效果还是不错的。

代码：

https://github.com/ljcan/Spark-Scala/blob/MLlib/CollaborativeFilter.scala

3. ALS 算法原理简介

ALS 的意思是交替最小二乘法（Alternating Least Squares），它只是是一种优化算法的名字，被用在求解 spark 中所提供的推荐系统模型的最优解。Spark 中协同过滤的文档中一开始就说了，这是一个基于模型的协同过滤（model-based CF），其实它是一种近几年推荐系统界大火的**隐语义模型**中的一种。**隐语义模型**又叫潜在因素模型，它试图通过数量相对少的未被观察到的底层原因，来解释大量用户和产品之间可观察到的交互。操作起来就是通过降维的方法来补全用户-物品矩阵，对矩阵中没有出现的值进行估计。基于这种思想的早期推荐系统常用的一种方法是 **SVD（奇异值分解）**。该方法在矩阵分解之前需要先把评分矩阵 R 缺失值补全，补全之后稀疏矩阵 R 表示成稠密矩阵 R'，然后将 R’分解成如下形式：



然后再选取 U 中的 K 列和 V 中的 S 行作为隐特征的个数，**达到降维的目的**。K 的选取通常用启发式策略。

这种方法有两个缺点，**第一是补全成稠密矩阵**之后需要耗费巨大的存储空间，在实际中，用户对物品的行为信息何止千万，对这样的稠密矩阵的存储是不现实的；**第二，SVD 的计算复杂度很高**，更不用说这样的大规模稠密矩阵了。所以关于 **SVD 的研究很多都是在小数据集上进行的。**

隐语义模型也是基于矩阵分解的，但是和 SVD 不同，它是把原始矩阵分解成两个矩阵相乘而不是三个。

$$A = XY^T$$

现在的问题就变成了确定 X 和 Y ，我们把 X 叫做**用户因子矩阵**，Y 叫做**物品因子矩阵**。通常上式不能达到精确相等的程度，我们要做的就是要**最小化他们之间的差距**，从而又变成了一个**最优化问题**。求解最优化问题我们很容易就想到了**随机梯度下降**，其中有一种方法就是这样，通过优化如下损失函数来找到 X 和 Y 中合适的参数：

$$C = \sum_{(u,i)\in K} (r_{ui} - \hat{r}_{ui})^2 = \sum_{(u,i)\in K} \left(r_{ui} - \sum_{k=1}^{K} p_{u,k} q_{i,k} \right)^2 + \lambda \left\| p_u \right\|^2 + \lambda \left\| q_i \right\|^2$$

其中 puk 就是 X 矩阵中 u 行 k 列的参数，度量了用户 u 和第 k 个隐类的关系；qik 是 Y 矩阵中 i 行 k 列的参数，度量了物品 i 和第 k 个隐类的关系。这种方式也是一种很流行的方法，有很多对它的相关扩展，比如加上偏置项的 LFM。

http://courses.ischool.berkeley.edu/i290-dm/s11/SECURE/a1-koren.pdf

得到上述损失函数在利用梯度下降法得到最优参数，这是第一种解法。

然而 **ALS 用的是另一种求解方法**，它先用随机初始化的方式固定一个矩阵，例如 Y：

$$A_i Y (Y^T Y)^{-1} = X_i$$

然后通过最小化等式两边差的平方来更新另一个矩阵 X，这就是“最小二乘”的由来。**得到 X 之后，又可以固定 X 用相同的方法求 Y，如此交替进行，直到最后收敛或者达到用户指定的迭代次数为止，是为“交替”是也。** 从上式可以看出，X 的第 i 行是 A 的第 i 行和 Y 的函数，因此可以很容易地分开计算 X 的每一行，这就为并行就算提供了很大的便捷，也正是如此，Spark 这种面向大规模计算的平台选择了这个算法。在 3 这篇文章中，作者用了 embarrassingly parallel 来形容这个算法，意思是高度易并行化的——它的每个子任务之间没有什么依赖关系。

在现实中，不可能每个用户都和所有的物品都有行为关系，事实上，有交互关系的用户-物品对只占很小的一部分，换句话说，用户-物品关系列表是非常稀疏的。**和 SVD 这种矩阵分解不同，ALS 所用的矩阵分解技术在分解之前不用把系数矩阵填充成稠密矩阵之后再分解，这不但大大减少了存储空间，而且 spark 可以利用这种稀疏性用简单的线性代数计算求解**。这几点使得本算法在大规模数据上计算非常快，解释了为什么 spark mllib 目前只有 ALS 一种推荐算法。

4. ALS 算法步骤

1.总结

ALS 是 alternating least squares 的缩写 ，意为交替最小二乘法；而 ALS-WR 是 *alternating-least-squares with weighted-λ -regularization* 的缩写，意为加权正则化交替最小二乘法。**而交替最小二乘法是对最小二乘法处理多个变量时的扩展(面试中问到请熟知)**。其基本原理是如果有两个变量需要确定，那 ALS 先固定第一个变量，然后求解第二个变量。之后固定第二个变量，求解第一个变量。如此交替迭代直至收敛或者达到最大迭代次数。

2. 算法推导

1.$L(U,V)=\sum_{i,j}[(r_{ij}-u_i^T v_j)^2+\lambda(|u_i|^2+|v_j|^2)]$

其中 U,V 分别是用户和商品的隐变量矩阵，r_{ij} 是用户i对商品j的评分值，λ 是正则项系数

2.算法过程：

2.1先随机初始化商品矩阵V

2.2固定矩阵V, 求偏导 $\frac{\delta L}{\delta u_i}$,得到更新公式 $u_i=(V^TV+\lambda I)^{-1}V^Tr_i$ 更新 U

2.3固定矩阵U, 求偏导 $\frac{\delta L}{\delta v_j}$,得到更新公式 $v_j=(U^TU+\lambda I)^{-1}U^Tr_j$,更新 V

2.4循环直至收敛或者达到最大迭代次数

这里将 ALS 用于解决稀疏矩阵分解的问题

3.推导过程

我们使用用户喜好特征矩阵 U(m* k) 中的第 i 个用户的特征向量 ui ，和产品特征矩阵 V(n* k) 第 j 个产品的特征向量 vj 来预测打分矩阵 A(m* n) 中的 aij。我们可以得出一下的矩阵分解模型的损失函数为：

$$C=\sum_{(i,j)\in R}[(a_{ij}-u_i v_j^T)^2+\lambda(u_i^2+v_j^2)]$$

有了损失函数之后，下面就开始介绍优化方法。通常的优化方法分为两种：交叉最小二乘法（alternative least squares）和随机梯度下降法（stochastic gradient descent）。Spark 使用的是**交叉最小二乘法（ALS）**来最优化损失函数。算法的思想就是：我们先随机生成然后固定它求解，再固定求解，这样交替进行下去，直到取得最优解 min(C)。因为每步迭代都会降低误差，并且误差是有下界的，**所以 ALS 一定会收敛**。但由于问题是非凸的，ALS 并不保证会收敛到全局最优解。但在实际应用中，ALS 对初始点不是很敏感，是否全局最优解造成的影响并不大。

算法的执行步骤：

- 先随机生成一个。一般可以取0值或者全局均值。
- 固定，即认为是已知的常量，来求解：

  $C = \sum_{(i,j)\in R}[(a_{ij} - u_i^{(0)}v_j^T)^2 + \lambda((u_i^2)^{(0)} + v_j^2)]$

  由于上式中只有v_j一个未知变量，因此C的最优化问题转化为最小二乘问题，用最小二乘法求解v_j的最优解：

  固定$j, j \in (1, 2, \ldots, n)$ ，则：等式两边关于为v_j求导得：

  $\frac{d(c)}{d(v_j)}$

  $= \frac{d}{d(v_j)}(\sum_{i=1}^{m}[(a_{ij} - u_i^{(0)}v_j^T)^2 + \lambda((u_i^2)^{(0)} + v_j^2)])$

  $= \sum_{i=1}^{m}[2(a_{ij} - u_i^{(0)}v_j^T)(-(u_i^T)^{(0)}) + 2\lambda v_j]$

  $= 2\sum_{i=1}^{m}[(u_i^{(0)}(u_i^T)^{(0)} + \lambda)v_j - a_{ij}(u_i^T)^{(0)}]$

  令$\frac{d(c)}{d(v_j)} = 0$ ，可得：

  $\sum_{i=1}^{m}[(u_i^{(0)}(u_i^T)^{(0)} + \lambda)v_j] = \sum_{i=1}^{m}a_{ij}(u_i^T)^{(0)}$

  $=> (U^{(0)}(U^T)^{(0)} + \lambda E)v_j = a_j^T U^{(0)}$

  令 $M_1 = U^{(0)}(U^T)^{(0)} + \lambda E, M_2 = a_j^T U^{(0)}$ ，则$v_j = M_1^{-1}M_2$

  按照上式依次计算v_1，v_2，…，v_n ，从而得到$V^{(0)}$
- 同理，用步骤2中类似的方法：

  $C = \sum_{(i,j)\in R}[(a_{ij} - u_i(v_j^T)^{(0)})^2 + \lambda(u_i^2 + (v_j^2)^{(0)})]$

  固定$i, i \in (1, 2, \ldots, m)$ ，则：等式两边关于为u_i求导得：

  $\frac{d(c)}{d(u_i)}$

  $= \frac{d}{d(u_i)}(\sum_{j=1}^{n}[(a_{ij} - u_i(v_j^T)^{(0)})^2 + \lambda((u_i^2) + (v_j^2)^{(0)})])$

  $= \sum_{j=1}^{n}[2(a_{ij} - u_i(v_j^T)^{(0)})(-(v_j^T)^{(0)}) + 2\lambda u_i]$

  $= 2\sum_{j=1}^{n}[(v_j^{(0)}(v_j^T)^{(0)} + \lambda)u_i - a_{ij}(v_j^T)^{(0)}]$

  令$\frac{d(c)}{d(u_i)} = 0$ ，可得：

  $\sum_{j=1}^{n}[(v_j^{(0)}(v_j^T)^{(0)} + \lambda)u_i] = \sum_{j=1}^{n}a_{ij}(v_j^T)^{(0)}$

  $=> ((V^{(0)}(V^T)^{(0)} + \lambda E)u_i = a_i^T V^{(0)}$

  令 $M_1 = V^{(0)}(V^T)^{(0)} + \lambda E, M_2 = a_i^T V^{(0)}$ ，则$u_i = M_1^{-1}M_2$

  按照上式依次计算u_1，u_2，…，u_n ，从而得到$U^{(1)}$
- 循环执行步骤2、3，直到损失函数C的值收敛（或者设置一个迭代次数N，迭代执行步骤2、3，N次后停止）。这样，就得到了C最优解对应的矩阵U、V。

落地：在理解下面的步骤中两个矩阵的求法：

1.$L(U,V)=\sum_{i,j}[(r_{ij}-u_i^T v_j)^2+\lambda(|u_i|^2+|v_j|^2)]$

其中U,V分别是用户和商品的隐变量矩阵，r_{ij}是用户i对商品j的评分值，λ是正则项系数

2.算法过程：

2.1先随机初始化商品矩阵V

2.2固定矩阵V，求偏导$\frac{\delta L}{\delta u_i}$，得到更新公式$u_i=(V^TV+\lambda I)^{-1}V^Tr_i$，更新$U$

2.3固定矩阵U，求偏导$\frac{\delta L}{\delta v_j}$，得到更新公式$v_j=(U^TU+\lambda I)^{-1}U^Tr_j$，更新$V$

2.4循环直至收敛或者达到最大迭代次数

参考：https://blog.csdn.net/oucpowerman/article/details/49847979

https://www.cnblogs.com/mooba/p/6539142.html

4 显示反馈与隐式反馈

推荐系统依赖不同类型的输入数据，最方便的是高质量的显式反馈数据，它们包含用户对感兴趣商品明确的评价。例如，Netflix 收集的用户对电影评价的星星等级数据。但是显式反馈数据不一定总是找得到，因此推荐系统可以从更丰富的隐式反馈信息中推测用户的偏好。

隐式反馈类型包括**购买历史、浏览历史、搜索模式甚至鼠标动作**。例如，购买同一个作者许多书的用户可能喜欢这个作者。

许多研究都集中在处理显式反馈，然而在很多应用场景下，应用程序重点关注隐式反馈数据。因为可能用户不愿意评价商品或者由于系统限制我们不能收集显式反馈数据。在隐式模型中，一旦用户允许收集可用的数据，在客户端并不需要额外的显式数据。

了解隐式反馈的特点非常重要，因为这些特质使我们避免了直接调用基于显式反馈的算法。**最主要的特点有如下几种：**

- 没有负反馈。通过观察用户行为，我们可以推测那个商品他可能喜欢，然后购买，但是我们很难推测哪个商品用户不喜欢。这在显式反馈算法中并不存在，因为用户明确告诉了我们哪些他喜欢哪些他不喜欢。
- 隐式反馈是内在的噪音。虽然我们拼命的追踪用户行为，但是我们仅仅只是猜测他们的偏好和真实动机。例如，我们可能知道一个人的购买行为，但是这并不能完全说明偏好和动机，因为这个商品可能作为礼物被购买而用户并不喜欢它。

- 显示反馈的数值值表示偏好（preference），隐式反馈的数值值表示信任（confidence）。基于显示反馈的系统用星星等级让用户表达他们的喜好程度，例如一颗星表示很不喜欢，五颗星表示非常喜欢。基于隐式反馈的数值值描述的是动作的频率，例如用户购买特定商品的次数。**一个较大的值并不能表明更多的偏爱。但是这个值是有用的，它描述了在一个特定观察中的信任度**。 一个发生一次的事件可能对用户偏爱没有用，但是一个周期性事件更可能反映一个用户的选择。
- 评价隐式反馈推荐系统需要合适的手段。

4.1 显式反馈模型

潜在因素模型由一个针对协同过滤的交替方法组成，它以一个更加全面的方式发现潜在特征来解释观察的 ratings 数据。我们关注的模型由奇异值分解（SVD）推演而来。一个典型的模型将每个用户 u （包含一个用户-因素向量 ui ）和每个商品 v （包含一个用户-因素向量 vj ）联系起来。

预测通过内积 $r_{ij} = (u^T)^i v^{j_i}$来实现。另一个需要关注的地方是参数估计。许多当前的工作都应用到了显式反馈数据集中，这些模型仅仅基于观察到的 rating 数据直接建模，同时通过一个适当的正则化来避免过拟合。公式就如上一节所提到的：

$$min_{u,v} \sum_{(i,j)\in R} [(a_{ij} - u_i v_j^T)^2 + \lambda(u_i^2 + v_j^2)]$$

4.2 隐式反馈模型

在显式反馈的基础上，我们需要做一些改动得到我们的隐式反馈模型。**首先**，我们需要形式化由 rij 变量衡量的**信任度**的概念。我们引入了一组二元变量 Pij，它表示用户 u 对商品 v 的偏好。Pij 的公式如下：

$$p_{ij} = \begin{cases} 1, r_{ij} > 0 \\ 0, r_{ij} = 0 \end{cases}$$

换句话说，如果用户购买了商品，我们认为用户喜欢该商品，否则我们认为用户不喜欢该商品。然而我们的信念（beliefs）与变化的信任（confidence）等级息息相关。首先，很自然的，pij 的值为 0 和低信任有关。用户对一个商品没有得到一个正的偏好可能源于多方面的原因，并不一定是不喜欢该商品。例如，用户可能并不知道该商品的存在。

另外，用户购买一个商品也并不一定是用户喜欢它。因此我们**需要一个新的信任等级来显示用户偏爱某个商品**。一般情况下，**rij 越大，越能暗示用户喜欢某个商品**。因此，我们引入了一组**变量 cij** ，

它衡量了我们观察到 pij 的信任度。cij 一个合理的选择如下所示：

$$c_{ij} = 1 + \alpha r_{ij}$$

Rij 表示的是交互的次数，次数越多，越信任商品，推荐后点击的概率增大

按照这种方式，我们存在最小限度的信任度，并且随着我们观察到的正偏向的证据越来越多，信任度也会越来越大。

我们的目的是找到用户向量 ui 以及商品向量 vj 来表明用户偏好。这些向量分别是**用户因素**（特征）向量和**商品因素**（特征）向量。本质上，这些向量将用户和商品映射到一个公用的隐式因素空间，从而使它们可以直接比较。这和用于显式数据集的矩阵分解技术类似，但是包含两点不一样的地方：

（1）**我们需要考虑不同的信任度**

（2）最优化需要考虑所有可能的 u，v 对，而不仅仅是和观察数据相关的 u，v 对。因此，通过最小化下面的损失函数来计算相关因素（factors）：

$$min_{u,v} \sum_{(i,j)\in R} [c_{ij}(a_{ij} - u_i v_j^T)^2 + \lambda(u_i^2 + v_j^2)]$$

参考：https://blog.csdn.net/u011239443/article/details/51752904

5.算法特性及优缺点

优点：相比 SVD 可以解决稀疏矩阵分解的问题

缺点：对于隐变量个数选择比较敏感，k 越大越精确，但是计算时间越长

6.注意事项

评分矩阵缺失值处理，填充 0

5. ALS 源码

https://github.com/apache/spark/blob/branch-2.1/mllib/src/main/scala/org/apache/spark/ml/recommendation/ALS.scala

Spark ALS 是 ALS 的分布式实现，非常高效，代码进行了大量的优化，有许多可以借鉴和思考的地方，它实现了 Explicit ALS 和 Implicit ALS 分布式算法。

Implicit ALS 原理

先说一下隐式数据的特点:

- 没有负反馈
- 充满噪声
- 显式数据的数值代表偏好，隐式数据的数值代表了置信等级（confidence level）。例如，我们观测到该用户看了某部电影，推测他可能喜欢该电影，如果发现他看了这部电影好几遍，我们就很有信心认为他喜欢这部电影了
- 基于隐式数据的特点，Implicit ALS 做如下假设：

- 引入二分值p_{ui}:

$$p_{u,i} = \begin{cases} 1 & \text{if } r_{u,i} > 0 \\ 0 & \text{otherwise} \end{cases}$$

- 引入置信等级c_{ui}

$$c_{ui} = 1 + \alpha r_{ui}$$

- 损失函数

$$\min_{x,y} \sum_{(u,i) \in K} c_{u,i}(p_{u,i} - x_u^\top y_i)^2 + \lambda(\sum_u ||x_u||^2 + \sum_i ||y_i||^2)$$

- 迭代公式

$$x_u = (Y^\top C^u Y + \lambda I)^{-1} Y^\top C^u p_u$$
$$y_i = (X^\top C^i X + \lambda I)^{-1} X^\top C^i p_i$$

其中，

$$C^u = \begin{pmatrix} c_{u1} & & \\ & \ddots & \\ & & c_{un} \end{pmatrix}$$

- 变换

以x_u为例，直接计算的话，计算量太大，注意到$Y^\top C^u Y = Y^\top Y + Y^\top (C^u - I) Y$，而$C^u - I$使得只需要计算用户$u$有过行为的物品集合，$Y^\top Y$可以在一轮迭代里只需要计算一次。这里已经有点分布式的味道了。

5.1Spark ALS 并行化分析

以 x_u 为例（之后也该角度分析），计算它需要两个要素：

- 用户 u 的评分详情（对哪个物品评了多少分），用于计算$C^u - I,\ p_u$
- 用户 u 关联的所有物品集的隐式因子，用于计算$Y^\top(C^u - I)Y$和$Y^\top C^u p_u$

Spark ALS 主要有三个步骤

- partitionRatings:将原始评分数据分片为块
- makeBlocks:产生 InBlock 和 OutBlock
- computeFactors：Normal Equation 求解

分布式计算关注的重点是**控制计算复杂度和通信复杂度**。Spark ALS 首先是以 Block 为单位进行正态方程求解的。例如在迭代 Block 1 中的所有用户 U1 的 factors 的时候，需要将 U1 的所有评分详情和所需的物品因子集全部汇总到同一个 Partition 中。

这里 Spark ALS 设计了两个结构 InBlock 和 OutBlock。**InBlock 存储评分详情，OutBlock 存储“因子关联索引”**，用于索引相关的 Item Facors，这部分的源码是个难点。

5.2Spark ALS 数据格式衍变

通过阅读源码，总结其从最初的评分数据格式衍变格式如下

编号	函数	形式 K→V
1	partitionRatings 将原始ratings分片为blocks	(srcBlockId, dstBlockId) → RatingBlock(srcIds, dstIds, ratings)
2.1	makeBlocks 建立InBlock和OutBlock	srcBlockId →(dstBlockId, srcIds, dstLocalIndices, ratings) dstLocalIndices表示Block本地索引
2.2		srcBlockId →(srcIds, dstEncodedIndices, ratings) dstEncodedIndices是dstBlockId和dstLocalIndices的组合
3	InBlock形态	srcBlockId→InBlock(uniqueSrcIds, dstPtrs, dstEncodedIndices, ratings) 这里进行了排序和矩阵压缩
4	OutBlock形态	srcBlockId→[dstBlockId][uniqSrcIdLocalIndex]
5	Factor形态	srcBlockId→[srcIdLocalIndex][Array[float]]

Block 用户集获取所需 Item 集 Factors 的过程：

```
ItemOutBlock.join(ItemFactors).flatMap {
    case(ItemBlockId, (ItemOutBlock, ItemFactors)) =>
     ItemOutBlock.view.zipWithIndex.map { case ([uniqItemIdLocalIndex], UserBlockId) =>
        (UserBlockId, (ItemBlockId, AssocItemFactors)))
      }
} =>(UserBlockId, Array[(ItemBlockId, ItemFactors)])
```

通过以上方式，所有需要的元素都传输到了一起。

5.3 重点源码分析

核心源码在 org.apache.spark.ml.recommendation.ALS 中，相比于 Spark SVD++的 200 多行的代码，ALS 的代码量真是巨无霸，只有 1~2 千行，不过核心模块的代码也是几百行左右，这里主要分析 makeBlocks 源码片段。

partitionRatings 的作用

原始评分数据是(user:ID,item:ID,rating:Float)这样的大量 tuple

把这些打分直接按照 **Tuple** 存的话会有几个问题。首先是空间的额外开销，每个 Tuple 实例都需要一个指针，而每个 Tuple 所存的数据不过是两个 ID 和一个打分，非常不划算。而且存储大量的 Tuple 实例会降低 Java 垃圾回收效率。所以我们使用**三个原始数组来存 InBlock 信息**：([v1, v2, v1, v2, v2], [u1, u1, u2, u2, u3], [a11, a12, a21, a22, a32])。这样不仅大幅减少了实例数量，还有效地利用了连续内存。

5.4 编号 3 矩阵压缩

注释上说 CSC 压缩，但是这块可能是 CSR 压缩，并且和普通的 CSR 压缩不同，因为是 hash 到了 block 单位，每个 block 上的 SrcIds 更加稀疏，此时用普通矩阵压缩会产生大量的 0 值。于是就有了这种形式，注意在压缩之前已经排好序了，这样做的便利很多，1 是方便压缩 2 是方便最后的正态方程的计算。

$$InBlock(uniqueSrcIds, dstPtrs, dstEncodedIndices, ratings)$$

其中，dstPtrs 表示 uniqueSrcId 步长

扩展：https://www.cnblogs.com/xbinworld/p/4273506.html **压缩格式**

5.5 编号 4 OutBlock 生成

OutBlock 的意义就是建立连接，当前的 SrcBlockId 中的哪些 srcIdLocalIndex 是需要发送到哪些 DstBlockId 的。

```
val outBlocks = inBlocks.mapValues { case InBlock(srcIds, dstPtrs, dstEncodedIndices, _) =>
        val encoder = new LocalIndexEncoder(dstPart.numPartitions)
```

```
    //activeIds 就是需要建立的“联系”，发送给哪些 dst，发送哪些 srcIdLocalIndex
    val activeIds = Array.fill(dstPart.numPartitions)(mutable.ArrayBuilder.make[Int])
    var i = 0
    //标识该 uniqSrcId 已经发送
    val seen = new Array[Boolean](dstPart.numPartitions)
    while (i < srcIds.length) {
      var j = dstPtrs(i)
      ju.Arrays.fill(seen, false)
      while (j < dstPtrs(i + 1)) {
        val dstBlockId = encoder.blockId(dstEncodedIndices(j))

        if (!seen(dstBlockId)) {
          //添加 local index 到该 out-block
          activeIds(dstBlockId) += i
          seen(dstBlockId) = true
        }
        j += 1
      }
      //该 uniqSrcId 发送完毕
      i += 1
    }
    activeIds.map { x =>
      x.result()
    }
  }.setName(prefix + "OutBlocks")
    .persist(storageLevel)
```

5.6topK 推荐

userFactors 和 itemFactorsuserFactors 和 itemFactors 计算完毕后，如何对每个用户对推荐 topK 物品呢，很显然计算复杂度为 **O(M* N)** 。Spark ALS 没有对该计算复杂度进行优化，主要在计算方式上进行优化，依旧是分块计算，并使用 level-3 BLAS 来进行矩阵内积运算。

看了 Spark ALS 的 JIRA 下的讨论，mahout 的作者 Sean Owen 提供了一种方法 **LSH 来缩减搜索**

范围，不过该方法会损失虽小但可见的精度，最后讨论的结果还是维持原状。

在实际操作的时候，我们可以根据领域知识缩减这个内积操作的范围，比如推荐餐馆，我们通过标识该用户的活动区域，只把该用户的隐式因子与这个区域的餐馆的隐式因子乘积，这样可以极大地缩减计算量。

5.7 参考：

- https://issues.apache.org/jira/browse/SPARK-3066
- http://www.csdn.net/article/2015-05-07/2824641
- https://blog.csdn.net/buptfanrq/article/details/73299116

6.ALS 推荐算法在 Spark 上的优化

参考：https://blog.csdn.net/butterluo/article/details/48271361

业务场景简述：

推荐系统需要基于 Spark 用 ALS 算法对近一天的数据进行实时训练，然后进行推荐. 输入的数据有 114G，但训练时间加上预测的时间需要 50 多分钟，而业务的要求是在 15 分钟左右，远远达不到实时推荐的要求，因此，我们与业务侧一起对 Spark 应用进行了优化.

更多参考：< Spark + Kafka 流计算优化 > https://blog.csdn.net/butterluo/article/details/47083773

6.1 优化分析

从数据分析，虽然数据有 114G，但 ALS 的模型训练时间并不长，反而是**数据加载和 ALS 预测**所占用的时间较长. 因此我们把重点放在这两点的优化中.

从 Spark 的 Web UI 可以看到，**数据加载的 job 中 task 的数量较多**，较小模型预测的 job 也是有这样的问题，因此，可以猜测**并行度过大造成了集群的协调负荷过重。**

6.2 仅降低并行度和优化 JVM 参数

我们用常见的"rdd.repartitionBy"把并行度降低后，作业计算耗时减少并不明显，且在模型预测的 job 中会有 executor 死掉的现象. 进而查看日志，发现是内存占用过多, yarn 把 Spark 应用给 kill 了.

为解决该现象，加入这些 JVM 参数: "

-XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:MaxTenuringThreshold=72 -XX:NewRatio=2 -XX:SurvivorRatio=6 -XX:+CMSParallelRemarkEnabled -XX:+ParallelRefProcEnabled -XX:+CMSPermGenSweepingEnabled -XX:+CMSClassUnloadingEnabled -XX:MaxTenuringThreshold=31 -XX:SurvivorRatio=8 -XX:+ExplicitGCInvokesConcurrent -XX:+ExplicitGCInvokesConcurrentAndUnloadsClasses -XX:+AlwaysPreTouch -XX:-OmitStackTraceInFastThrow -XX:+UseCompressedStrings -XX:+UseStringCache

-XX:+OptimizeStringConcat -XX:+UseCompressedOops -XX:+CMSScavengeBeforeRemark -XX:+UseBiasedLocking -XX:+AggressiveOpts " 具体说明和原因请参考本人的另一篇 blog < Spark + Kafka 流计算优化 >.

6.3 笛卡尔积操作中的预先分块

作了上述例行优化后， ALS 预测步骤的耗时减少依然不明显，为发掘原因，我们看了下源码，惊奇的发现在 **ALS 预测中竟然是用了笛卡尔积操作**, 114G 的数据少说也有几千万行记录，几千万行记录进行笛卡尔积，不慢才怪吧.

还好，我们有扩展版的 ALS 预测方法，可以**将数据预先分块，而不必一行行的进行笛卡尔积，加快了笛卡尔积的速度**.

```
val recommendations = ExtMatrixFactorizationModelHelper.recommendProductsForUsers( model.get , numRecommendations，420000，StorageLevel.MEMORY_AND_DISK_SER )
```

该方法会对 model 中的 userFeatures 和 itemFeatures 矩阵进行**预先分块**，减少网络包的传输量和笛卡尔积的计算量.

这里的第三个参数是每一个块所包含的行数，此处的 420000 表示当我们对 userFeatures 或 itemFeatures 进行分块时，每一个块包含了矩阵的 420000 行.

如何计算这里的一个块要包含多少行呢，举例如下:

由于 ALS 算法中设置的 rank 是 10，因此生成的 userFeatures 和 itemFeatures 的个数是 10，它们的每一行是(Int,Array[Double])，其中 Array.size 是 10.

因此可根据如下计算每行所占的空间大小:

空 Array[10] 的大小=16+8*10=96Byte. 数组中的元素是 Double，十个 Double 对象的大小是 16*10=160Byte. 作为 key 的 Integer 的大小是 16Byte. 因此每行占空间 96+160+16=**212Byte**.

另外,要计算带宽: 由于是千兆网卡,因此带宽为 1Gbit,转换为 Byte 也就是 128MByte 的带宽.

考虑到“ spark.akka.frameSize=100”以及网络包包头需要占的空间，和 Java 的各种封装要占的空间，我们计划让 1 个 block 就几乎占满带宽，也就是一个 block 会在 100MByte 左右.

因此，一个 block 要占 60*1024*1024/212=296766 行，因此 blocksize=494611，考虑到各个 object 也占内存，因此行数定为 420000 左右.

在分块后 ExtMatrixFactorizationModelHelper.recommendProductsForUsers 中会对块进行重新分区，以达到基于块的均匀分布.

6.4.提高文件加载速度

以前都是加载小文件，每个文件才几 M 大小，远远低于 Hadoop 的块大小，使得 IO 频繁，文件也频繁打开关闭，加载速度自然就慢. 为解决该问题，我们使用 sc.wholeTextFiles(dirstr,inputSplitNum) 来加载 HDFS 的文件到 Spark 中. **该方法使用 Hadoop 的 CombineFileInputFormat 把多个小文件合并成一个 Split 再加载到 Spark 中**.

6.5 减少笛卡尔积计算量

回到对笛卡尔积计算的优化，因为 50 分钟的计算量基本上都是耗在笛卡尔积的计算上的.

我们先看一下 task 的分布图，由图看到，数据并不是十分散列:

猜测原因肯是通过 Array.mkString.hashCode 作为 key 并不能保证数据的均匀散列.因此，我们 disable 掉了在笛卡尔积计算前的预分块时的再分区，而是**把再分区提到分块之前**.

这样一来, task 分布稍微均衡了一些，但依然不甚理想. 为了合理的降低 task 数量和均匀 task 的分

布，我们进一步使用了 Spark 扩展版本的**自动分区功能**.

```
val userFactors = ExtSparkHelper.repartionPairRDDBySize(oldUsrFact， ThreeM)
```

这个方法有两个参数，**第一个是需要分区的 RDD，第二个是我们期望的每一个 task 的 input data 的大小**. 一般来说, input data 与计算产生的 data 的大小相差不大，但笛卡尔积却不同，有可能产生上百倍的中间数据量. 因此，我这里设置的每个 task 的 input data 是 3M，计算产生的中间数据刚好在 1G 左右，一个 executor 可以同时跑 3 个 task，也算是比较理想的.

优化后的 task 分布图也比较理想，十分均匀且没有任何浪费，如下:

因此，这一步优化后，原笛卡尔积的运行速度从几十分钟变为十几秒.

整个 ALS 应用原来跑 114G 数据需要 20 多分钟,如下图现在只需要不到 4min:

Completed Jobs (57)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
56	saveAsTextFile at <console>:44	2015/09/02 13:31:07	1 s	2/2 (113 skipped)	176/176 (10526 skipped)
55	saveAsTextFile at <console>:44	2015/09/02 13:30:41	25 s	7/7 (106 skipped)	418/418 (10108 skipped)
54	count at <console>:59	2015/09/02 13:30:39	70 ms	1/1 (55 skipped)	1/1 (5360 skipped)
53	count at <console>:56	2015/09/02 13:30:39	0.2 s	1/1 (56 skipped)	22/22 (5448 skipped)
52	first at MatrixFactorizationModel.scala:64	2015/09/02 13:30:38	0.4 s	2/2 (54 skipped)	89/89 (5272 skipped)
51	first at MatrixFactorizationModel.scala:64	2015/09/02 13:30:38	0.7 s	2/2 (55 skipped)	89/89 (5360 skipped)
50	RangePartitioner at <console>:70	2015/09/02 13:30:37	0.7 s	2/2 (54 skipped)	176/176 (5272 skipped)
49	aggregate at ALS.scala:1175	2015/09/02 13:30:36	0.5 s	2/2 (53 skipped)	176/176 (5184 skipped)
48	aggregate at ALS.scala:1175	2015/09/02 13:30:35	0.6 s	2/2 (52 skipped)	176/176 (5096 skipped)
47	aggregate at ALS.scala:1175	2015/09/02 13:30:35	0.7 s	2/2 (51 skipped)	176/176 (5008 skipped)
46	aggregate at ALS.scala:1175	2015/09/02 13:30:34	0.3 s	2/2 (50 skipped)	176/176 (4920 skipped)
45	aggregate at ALS.scala:1175	2015/09/02 13:30:33	0.9 s	2/2 (49 skipped)	176/176 (4832 skipped)
44	aggregate at ALS.scala:1175	2015/09/02 13:30:33	0.5 s	2/2 (48 skipped)	176/176 (4744 skipped)
43	aggregate at ALS.scala:1175	2015/09/02 13:30:32	0.6 s	2/2 (47 skipped)	176/176 (4656 skipped)
42	aggregate at ALS.scala:1175	2015/09/02 13:30:30	1.0 s	2/2 (46 skipped)	176/176 (4568 skipped)
41	aggregate at ALS.scala:1175	2015/09/02 13:30:29	1 s	2/2 (45 skipped)	176/176 (4480 skipped)
40	aggregate at ALS.scala:1175	2015/09/02 13:30:29	0.5 s	2/2 (44 skipped)	176/176 (4392 skipped)
39	aggregate at ALS.scala:1175	2015/09/02 13:30:28	0.5 s	2/2 (43 skipped)	176/176 (4304 skipped)
38	aggregate at ALS.scala:1175	2015/09/02 13:30:27	0.8 s	2/2 (42 skipped)	176/176 (4216 skipped)
37	aggregate at ALS.scala:1175	2015/09/02 13:30:26	0.8 s	2/2 (41 skipped)	176/176 (4128 skipped)
36	aggregate at ALS.scala:1175	2015/09/02 13:30:26	0.6 s	2/2 (40 skipped)	176/176 (4040 skipped)
35	aggregate at ALS.scala:1175	2015/09/02 13:30:25	0.8 s	2/2 (39 skipped)	176/176 (3952 skipped)
34	aggregate at ALS.scala:1175	2015/09/02 13:30:24	0.4 s	2/2 (38 skipped)	176/176 (3864 skipped)
33	aggregate at ALS.scala:1175	2015/09/02 13:30:23	1 s	2/2 (37 skipped)	176/176 (3776 skipped)
32	aggregate at ALS.scala:1175	2015/09/02 13:30:22	0.8 s	2/2 (36 skipped)	176/176 (3688 skipped)
31	aggregate at ALS.scala:1175	2015/09/02 13:30:21	0.5 s	2/2 (35 skipped)	176/176 (3600 skipped)
30	aggregate at ALS.scala:1175	2015/09/02 13:30:20	0.8 s	2/2 (34 skipped)	176/176 (3512 skipped)
29	aggregate at ALS.scala:1175	2015/09/02 13:30:19	0.7 s	2/2 (33 skipped)	176/176 (3424 skipped)
28	aggregate at ALS.scala:1175	2015/09/02 13:30:19	0.6 s	2/2 (32 skipped)	176/176 (3336 skipped)
27	aggregate at ALS.scala:1175	2015/09/02 13:30:18	0.6 s	2/2 (31 skipped)	176/176 (3248 skipped)
26	aggregate at ALS.scala:1175	2015/09/02 13:30:17	0.4 s	2/2 (30 skipped)	176/176 (3160 skipped)
25	aggregate at ALS.scala:1175	2015/09/02 13:30:16	0.8 s	2/2 (29 skipped)	176/176 (3072 skipped)
24	aggregate at ALS.scala:1175	2015/09/02 13:30:15	1 s	2/2 (28 skipped)	176/176 (2984 skipped)
23	aggregate at ALS.scala:1175	2015/09/02 13:30:14	1 s	2/2 (27 skipped)	176/176 (2896 skipped)
22	aggregate at ALS.scala:1175	2015/09/02 13:30:13	0.4 s	2/2 (26 skipped)	176/176 (2808 skipped)
21	aggregate at ALS.scala:1175	2015/09/02 13:30:13	0.7 s	2/2 (25 skipped)	176/176 (2720 skipped)
20	aggregate at ALS.scala:1175	2015/09/02 13:30:12	0.7 s	2/2 (24 skipped)	176/176 (2632 skipped)
19	aggregate at ALS.scala:1175	2015/09/02 13:30:11	0.7 s	2/2 (23 skipped)	176/176 (2544 skipped)
18	aggregate at ALS.scala:1175	2015/09/02 13:30:10	0.4 s	2/2 (22 skipped)	176/176 (2456 skipped)
17	aggregate at ALS.scala:1175	2015/09/02 13:30:10	0.7 s	2/2 (21 skipped)	176/176 (2368 skipped)
16	aggregate at ALS.scala:1175	2015/09/02 13:30:09	0.7 s	2/2 (20 skipped)	176/176 (2280 skipped)
15	aggregate at ALS.scala:1175	2015/09/02 13:30:08	0.6 s	2/2 (19 skipped)	176/176 (2192 skipped)
14	aggregate at ALS.scala:1175	2015/09/02 13:30:08	0.7 s	2/2 (18 skipped)	176/176 (2104 skipped)
13	aggregate at ALS.scala:1175	2015/09/02 13:30:07	0.7 s	2/2 (17 skipped)	176/176 (2016 skipped)
12	aggregate at ALS.scala:1175	2015/09/02 13:30:06	0.5 s	2/2 (16 skipped)	176/176 (1928 skipped)
11	aggregate at ALS.scala:1175	2015/09/02 13:30:05	0.9 s	3/3 (8 skipped)	264/264 (964 skipped)
10	aggregate at ALS.scala:1175	2015/09/02 13:30:05	0.2 s	1/1 (7 skipped)	88/88 (876 skipped)
9	count at ALS.scala:551	2015/09/02 13:30:04	0.7 s	2/2 (6 skipped)	176/176 (788 skipped)
8	count at ALS.scala:543	2015/09/02 13:30:02	2 s	3/3 (5 skipped)	264/264 (700 skipped)
7	count at <console>:82	2015/09/02 13:30:01	0.3 s	2/2 (4 skipped)	176/176 (612 skipped)
6	count at <console>:82	2015/09/02 13:30:01	0.5 s	2/2 (4 skipped)	176/176 (612 skipped)
5	count at <console>:82	2015/09/02 13:30:00	0.9 s	2/2 (4 skipped)	176/176 (612 skipped)
4	RangePartitioner at <console>:71	2015/09/02 13:30:00	0.2 s	1/1 (4 skipped)	88/88 (612 skipped)
3	RangePartitioner at <console>:71	2015/09/02 13:29:59	0.1 s	1/1 (4 skipped)	88/88 (612 skipped)
2	RangePartitioner at <console>:71	2015/09/02 13:29:56	3 s	2/2 (3 skipped)	241/241 (459 skipped)
1	RangePartitioner at <console>:67	2015/09/02 13:29:35	20 s	4/4	612/612
0	RangePartitioner at <console>:56	2015/09/02 13:28:03	1.5 min	1/1	153/153