## ALS算法在Spark上的实现

零、基础知识

从广义上讲，推荐系统基于两种不同的策略：基于内容的方法和基于协同过滤的方法。Spark中使用协同过滤的方式。协同过滤分析用户以及用户相关的产品的相关性，用以识别新的用户-产品相关性。协同过滤系统需要的唯一信息是用户过去的行为信息，比如对产品的评价信息。协同过滤是领域无关的，所以它可以方便解决基于内容方法难以解决的许多问题。

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。

 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：

（1） 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。

（2） 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。

（3） 显示反馈的数值值表示偏好（preference），隐式回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。

（4） 评价隐式反馈推荐系统需要合适的手段。

针对不同的反馈模型，提出了两种不同的计算模型，

（1）显示反馈模型

（2）隐式反馈模型

### 一、ALS算法实现的评测指标： Root-mean-square error

|  |
| --- |
| 均方根值（RMS）+ 均方根误差（RMSE）+标准差（Standard Deviation）  1. **均方根值（RMS）也称作为效值**，计算方法是先平方、再平均、然后开方。      1. 均方根误差，**它是观测值与真值偏差的平方和观测次数n比值的平方根，**在实际测量中，观测次数n总是有限的，真值只能用最可信赖（最佳）值来代替.方根误差对一组测量中的特大或特小误差反映非常敏感，所以，**均方根误差能够很好地反映出测量的精密度。**均方根误差，当对某一量进行甚多次的测量时，取这一测量列真误差的均方根差(真误差平方的算术平均值再开方)，称为标准偏差，以σ表示。σ反映了测量数据偏离真实值的程度，σ越小，表示测量精度越高，因此可用σ作为评定这一测量过程精度的标准。      1. **标准差（Standard Deviation）**，标准差是方差的算术平方根，也称均方差（mean square error），是各数据偏离平均数的距离的平均数，它是离均差平方和平均后的方根，用σ表示，标准差**能反映一个数据集的离散程度**。 |

### ALS算法在spark上的实现的基础概念

2.1、求解最小化损失函数

考虑到损失函数包含m\*n个元素，m是用户的数量，n是商品的数量。一般情况下，m\*n可以到达几百亿。这么多的元素应该避免使用随机梯度下降法来求解，因此，spark选择使用交替最优化方式求解。

具体的：

固定公式中的用户-特征向量或者商品-特征向量，公式就会变成二次方程，可以求出全局的极小值。交替最小二乘的计算过程是：交替的重新计算用户-特征向量和商品-特征向量，每一步都保证降低损失函数的值，直到找到极小值。

2.1、在spark实现ALS算法中，具有如下的参数：

1、numBlocks 是并行化的块数

2、rank 模型中的隐藏因子（矩阵的秩）

3、iterations 算法迭代次数，通常情况下ALS在20次或者更少的迭代就可以收敛

4、lambda 是正则化参数，为了防止过拟合，引入正则化参数，通过实验确定

5、implicitPrefs 是一个boolean值，确定的是是否使用隐式方式进行分解

6、alpha 是应用于ALS的隐式反馈变，用于控制观察值的基准置信度

2.2 Explicit 和 implicit反馈

基于协同过滤的标准矩阵分解的方法把用户-产品矩阵的数据当成显示偏好矩阵，这是由用户显示给出。

在实际情况下，通常只有对那些隐式反馈的访问。Spark的MLLib通过观察并处理用户行为的一些特征的权重，而不是直接得到评价矩阵。这些权重作为推到出用户偏好的置信度参考值，而不是隐式的对商品的评价值。这个模型试图发现

2.3 正则化参数的比例

引入正则化比例是为了防止过拟合的问题，这个思路是来自于“[Large-Scale Parallel Collaborative Filtering for the Netflix Prize](http://dx.doi.org/10.1007/978-3-540-68880-8_32)”，他的特点是lambda的值不是太依赖于数据集的大小，所以在选取的小数据集上的得到的模型，在大的数据集上也具有相似度性能和结果。（在文章中确定lambda的方法是不断的进行实验，得到他的最佳值）。具体的做法是固定Nf，那么得到lambda是RMSE的函数，通过不同的实验，确定他的最佳值。

### ALS算法在Spark上的实现代码分析

代码从官方给出的例子出发，进行确定：

3.1输入数据的格式是

|  |
| --- |
| 1::93::1::1424380312  数据格式的解释说明：(1) 第一个是用户ID  (2) 第二个是电影ID  (3) 第三个是用户对电影的评价值  (4) 第四个是时间戳（评价时间） |

3.2例子程序是

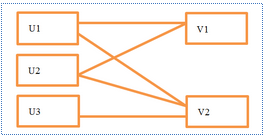
|  |
| --- |
| **import** org.apache.spark.ml.evaluation.RegressionEvaluator **import** org.apache.spark.ml.recommendation.ALS // $example off$ **import** org.apache.spark.sql.SparkSession */\*\*  \* An example demonstrating ALS.  \* Run with  \** {{{  *\* bin/run-example ml.ALSExample  \** }}}  *\*/* **object** ALSExample {  // $example on$  **case class** Rating(userId: Int, movieId: Int, rating: Float, timestamp: Long)  **def** parseRating(str: String): Rating = {  **val** fields = str.split("::")  *assert*(fields.size == 4)  *Rating*(fields(0).toInt, fields(1).toInt, fields(2).toFloat, fields(3).toLong)  }  // $example off$  **def** main(args: Array[String]) {  **val** spark = SparkSession  .*builder* .appName("ALSExample").master("local")  .getOrCreate()  **import** spark.implicits.\_  // $example on$  val ratings = spark.read.textFile("data/mllib/als/sample\_movielens\_ratings.txt")  .map(*parseRating*)  .toDF()   **val** *Array*(training, test) = ratings.randomSplit(*Array*(0.8, 0.2))  // Build the recommendation model using ALS on the training data  **val als = new ALS()  .setMaxIter(5)  .setRegParam(0.01)  .setUserCol("userId")  .setItemCol("movieId")  .setRatingCol("rating")**   **val model = als.fit(training)**  // Evaluate the model by computing the RMSE on the test data  **val** predictions = model.transform(test)  **val** evaluator = **new** RegressionEvaluator()  .setMetricName("rmse")  .setLabelCol("rating")  .setPredictionCol("prediction")  **val** rmse = evaluator.evaluate(predictions)  *println*(s"Root-mean-square error = **$**rmse")  // $example off$  spark.stop()  } } |
| 在这个方法中，1、读入评价矩阵，  2、把数据放入到DataFrame中，  3、把数据随机的分成训练集和测试集（在官方给出的例子中，给出的训练集占到80%，测试集占到20%）  4、生成一个新的ALS的实例，在实例中设置以下的参数：  **（1）最大迭代次数**  **（2）设置初始的正则系数 lambda**  （3）设置矩阵描述的元数据，在这个例子中是 用户ID 电影ID和评分  **5、调用als.fit()函数，传入训练数据集，返回训练的模型**  6、在模型上调用 model.transform(test),得到predictions  **7、在测试集上进行评价，使用的RegressionEvaluator这个评价实例**  **在这个实例中设置，评价参数（这里使用的RMSE）。返回evaluator.**  8、调用evalutor.evaluate(predictions)，得到rmse的结果。 |

3.3 ALS类分析

|  |
| --- |
| 类名： org.apache.spark.ml.recommendation.ALS.scala |
| Trait ALSModelParams：  1、userCol 描述的是 用户ID的列名  2、itemCol 描述的是 商品ID的列名 |
| Trait ALSParams：  1、rank 矩阵分解的秩，分解后的矩阵的秩 默认值是10  2、numblocks 用户数据块的数量 默认值是10  3、numItemBlocks 商品数据块的数量 默认值是10  4、implicitPrefs 是否使用隐式偏好 默认值是false  5、alpha 在隐式偏好方程中的系数，默认值是1.0  6、ratingCol 评价矩阵的评价指标的列名 默认值是“rating” |
| Class ALSModel：  **override def** transform(dataset: Dataset[\_]): DataFrame = {  transformSchema(dataset.schema)  // Register a UDF for DataFrame, and then  // create a new column named map(predictionCol) by running the predict UDF.  **val** predict = *udf* { (userFeatures: Seq[Float], itemFeatures: Seq[Float]) =>  **if** (userFeatures != **null** && itemFeatures != **null**) {  *blas*.sdot(rank, userFeatures.toArray, 1, itemFeatures.toArray, 1)  } **else** {  Float.*NaN* }  }  dataset  **.join(**userFactors,  *checkedCast*(dataset($(*userCol*)).cast(DoubleType)) === userFactors("id"), "left")   **.join**(itemFactors,  *checkedCast*(dataset($(*itemCol*)).cast(DoubleType)) === itemFactors("id"), "left")   **.select(**dataset("\*"),  predict(userFactors("features"), itemFactors("features")).as($(*predictionCol*))) } |
| Object ALSModel： |
| Class ALS：  1、在这个类中有一些列的参数的设置  2、其核心是fit函数： |
| @Since("2.0.0") **override def fit(dataset: Dataset[\_]): ALSModel = {**  transformSchema(dataset.schema)  **import** dataset.sparkSession.implicits.\_  **val** r = **if** ($(*ratingCol*) != "") *col*($(*ratingCol*)).cast(FloatType) **else** *lit*(1.0f)  val ratings = dataset  .select(*checkedCast*(*col*($(*userCol*)).cast(DoubleType)),  *checkedCast*(*col*($(*itemCol*)).cast(DoubleType)), r)  .*rdd* .map { row =>  *Rating*(row.getInt(0), row.getInt(1), row.getFloat(2))  }  **/\*返回的是用户子矩阵和产品子矩阵\*/**  **val (userFactors, itemFactors) = ALS.*train*(ratings, rank = $(*rank*),  numUserBlocks = $(*numUserBlocks*), numItemBlocks = $(*numItemBlocks*),  maxIter = $(*maxIter*), regParam = $(*regParam*), implicitPrefs = $(*implicitPrefs*),  alpha = $(*alpha*), nonnegative = $(*nonnegative*),  intermediateRDDStorageLevel = StorageLevel.*fromString*($(*intermediateStorageLevel*)),  finalRDDStorageLevel = StorageLevel.*fromString*($(*finalStorageLevel*)),  checkpointInterval = $(*checkpointInterval*), seed = $(*seed*))**  **val** userDF = userFactors.toDF("id", "features")  **val** itemDF = itemFactors.toDF("id", "features")  **val** model = **new** ALSModel(uid, $(*rank*), userDF, itemDF).setParent(**this**)  instrLog.logSuccess(model)  copyValues(model) }  函数分析：1、在函数中进行数据的检查  2、调用伴生对象ALS中的train方法，返回userFactors, itemFactors  3、通过使用userFactors, itemFactors的方法得到用户矩阵和商品矩阵  4、得到两个矩阵，生成ALSmodel。 |
| Object ALS：在这个类中具体实现ALS算法。  */\*\*  \* :: DeveloperApi ::  \* Implementation of the ALS algorithm.  \*/* @DeveloperApi def train[ID: ClassTag]( // scalastyle:ignore  ratings: RDD[Rating[ID]],  rank: Int = 10,  numUserBlocks: Int = 10,  numItemBlocks: Int = 10,  maxIter: Int = 10,  regParam: Double = 1.0,  implicitPrefs: Boolean = **false**,  alpha: Double = 1.0,  nonnegative: Boolean = **false**,  intermediateRDDStorageLevel: StorageLevel = StorageLevel.*MEMORY\_AND\_DISK*,  finalRDDStorageLevel: StorageLevel = StorageLevel.*MEMORY\_AND\_DISK*,  checkpointInterval: Int = 10,  seed: Long = 0L)(  **implicit** ord: Ordering[ID]): (RDD[(ID, Array[Float])], RDD[(ID, Array[Float])]) = {  *require*(intermediateRDDStorageLevel != StorageLevel.*NONE*,  "ALS is not designed to run without persisting intermediate RDDs.")  **val** sc = ratings.sparkContext  **/\*按照0—9的方式进行分片\*/**  **val** userPart = **new** ALSPartitioner(numUserBlocks)  **val** itemPart = **new** ALSPartitioner(numItemBlocks)  /\*  numberOfLeadingZeros这个方法返回二进制从左开始连续不为0的位数，这里传入9，返回28，就是前28位都为0，28和31取最小numLocalIndexBits 是28  \*/  **val** userLocalIndexEncoder = **new** LocalIndexEncoder(userPart.numPartitions)  **val** itemLocalIndexEncoder = **new** LocalIndexEncoder(itemPart.numPartitions)  **val solver = if (nonnegative) new NNLSSolver else new CholeskySolver**  **val** blockRatings = *partitionRatings*(ratings, userPart, itemPart)  .persist(intermediateRDDStorageLevel)   **val (userInBlocks, userOutBlocks) =  *makeBlocks*("user", blockRatings, userPart, itemPart, intermediateRDDStorageLevel)  // materialize blockRatings and user blocks  userOutBlocks.count()**  **val** swappedBlockRatings = blockRatings.map {  **case** ((userBlockId, itemBlockId), *RatingBlock*(userIds, itemIds, localRatings)) =>  ((itemBlockId, userBlockId), *RatingBlock*(itemIds, userIds, localRatings))  }  **val** (itemInBlocks, itemOutBlocks) =  *makeBlocks*("item", swappedBlockRatings, itemPart, userPart, intermediateRDDStorageLevel)  // materialize item blocks  itemOutBlocks.count()  **val** seedGen = **new** XORShiftRandom(seed)  **var** userFactors = *initialize*(userInBlocks, rank, seedGen.nextLong())  **var** itemFactors = *initialize*(itemInBlocks, rank, seedGen.nextLong())  **var** previousCheckpointFile: Option[String] = None  **val** shouldCheckpoint: Int => Boolean = (iter) =>  sc.*checkpointDir*.isDefined && checkpointInterval != -1 && (iter % checkpointInterval == 0)  **val** deletePreviousCheckpointFile: () => Unit = () =>  previousCheckpointFile.foreach { file =>  **try** {  **val** checkpointFile = **new** Path(file)  checkpointFile.getFileSystem(sc.hadoopConfiguration).delete(checkpointFile, **true**)  } **catch** {  **case** e: IOException =>  logWarning(s"Cannot delete checkpoint file **$**file:", e)  }  }  **if** (implicitPrefs) {  **for** (iter <- 1 to maxIter) {  userFactors.setName(s"userFactors-**$**iter").persist(intermediateRDDStorageLevel)  **val** previousItemFactors = itemFactors  itemFactors = *computeFactors*(userFactors, userOutBlocks, itemInBlocks, rank, regParam,  userLocalIndexEncoder, implicitPrefs, alpha, solver)  previousItemFactors.unpersist()  itemFactors.setName(s"itemFactors-**$**iter").persist(intermediateRDDStorageLevel)  // TODO: Generalize PeriodicGraphCheckpointer and use it here.  **val** deps = itemFactors.dependencies  **if** (shouldCheckpoint(iter)) {  itemFactors.checkpoint() // itemFactors gets materialized in computeFactors  }  **val** previousUserFactors = userFactors  userFactors = *computeFactors*(itemFactors, itemOutBlocks, userInBlocks, rank, regParam,  itemLocalIndexEncoder, implicitPrefs, alpha, solver)  **if** (shouldCheckpoint(iter)) {  ALS.*cleanShuffleDependencies*(sc, deps)  deletePreviousCheckpointFile()  previousCheckpointFile = itemFactors.getCheckpointFile  }  previousUserFactors.unpersist()  }  } **else** {  **for** (iter <- 0 until maxIter) {  itemFactors = *computeFactors*(userFactors, userOutBlocks, itemInBlocks, rank, regParam,  userLocalIndexEncoder, solver = solver)  **if** (shouldCheckpoint(iter)) {  **val** deps = itemFactors.dependencies  itemFactors.checkpoint()  itemFactors.count() // checkpoint item factors and cut lineage  ALS.*cleanShuffleDependencies*(sc, deps)  deletePreviousCheckpointFile()  previousCheckpointFile = itemFactors.getCheckpointFile  }  userFactors = *computeFactors*(itemFactors, itemOutBlocks, userInBlocks, rank, regParam,  itemLocalIndexEncoder, solver = solver)  }  }  **val** userIdAndFactors = userInBlocks  .mapValues(\_.srcIds)  .join(userFactors)  .mapPartitions({ items =>  items.flatMap { **case** (\_, (ids, factors)) =>  ids.view.zip(factors)  }  // Preserve the partitioning because IDs are consistent with the partitioners in userInBlocks  // and userFactors.  }, preservesPartitioning = **true**)  .setName("userFactors")  .persist(finalRDDStorageLevel)  **val** itemIdAndFactors = itemInBlocks  .mapValues(\_.srcIds)  .join(itemFactors)  .mapPartitions({ items =>  items.flatMap { **case** (\_, (ids, factors)) =>  ids.view.zip(factors)  }  }, preservesPartitioning = **true**)  .setName("itemFactors")  .persist(finalRDDStorageLevel)  **if** (finalRDDStorageLevel != StorageLevel.*NONE*) {  userIdAndFactors.count()  itemFactors.unpersist()  itemIdAndFactors.count()  userInBlocks.unpersist()  userOutBlocks.unpersist()  itemInBlocks.unpersist()  itemOutBlocks.unpersist()  blockRatings.unpersist()  }  (userIdAndFactors, itemIdAndFactors) } |
| 1、方法参数：  ratings: RDD[Rating[ID]], rank: Int = 10, numUserBlocks: Int = 10, numItemBlocks: Int = 10, maxIter: Int = 10, regParam: Double = 1.0, implicitPrefs: Boolean = false, alpha: Double = 1.0, nonnegative: Boolean = false, intermediateRDDStorageLevel:StorageLevel=StorageLevel.MEMORY\_AND\_DISK, finalRDDStorageLevel: StorageLevel = StorageLevel.MEMORY\_AND\_DISK, checkpointInterval: Int = 10, seed: Long = 0L |
| 2、代码解释说明 |
| ALSPartitioner: 使用的是HashPartitionser，就是按照Hash方法把输入的数据进行分片。 |
| 矩阵分解的方法:  使用两种矩阵分级方法， |
| */\*\* Trait for least squares solvers applied to the normal equation. \*/* **private**[recommendation] **trait** LeastSquaresNESolver **extends** Serializable {  */\*\* Solves a least squares problem with regularization (possibly with other constraints). \*/* **def** solve(ne: NormalEquation, lambda: Double): Array[Float] } |
| Class NNLSSolver |
| Class CholeskySolver说明：  */\*\* Cholesky solver for least square problems. \*/* **private**[recommendation] **class** CholeskySolver **extends** LeastSquaresNESolver {  */\*\*  \* Solves a least squares problem with L2 regularization:  \*  \* min norm(A x - b)*^*2*^ *+ lambda \* norm(x)*^*2*^  *\*  \** ***@param ne*** *a* [[*NormalEquation*]] *instance that contains AtA, Atb, and n (number of instances)  \** ***@param lambda*** *regularization constant  \** ***@return*** *the solution x  \*/* **override def** solve(ne: NormalEquation, lambda: Double): Array[Float] = {  **val** k = ne.k  // Add scaled lambda to the diagonals of AtA.  **var** i = 0  **var** j = 2  **while** (i < ne.*triK*) {  ne.*ata*(i) += lambda  i += j  j += 1  }  CholeskyDecomposition.*solve*(ne.*ata*, ne.*atb*)  **val** x = **new** Array[Float](k)  i = 0  **while** (i < k) {  x(i) = ne.*atb*(i).toFloat  i += 1  }  ne.reset()  x  } } |
| Case class Rating: 包括用户ID ， 商品ID，还有评分 rating。 |
| 方法 makeBlocks：  */\*\*  \* Creates in-blocks and out-blocks from rating blocks.  \*  \** ***@param prefix*** *prefix for in/out-block names  \** ***@param ratingBlocks*** *rating blocks  \** ***@param srcPart*** *partitioner for src IDs  \** ***@param dstPart*** *partitioner for dst IDs  \** ***@return*** *(in-blocks, out-blocks)  \*/* **private def** makeBlocks[ID: ClassTag](  prefix: String,  ratingBlocks: RDD[((Int, Int), RatingBlock[ID])],  srcPart: Partitioner,  dstPart: Partitioner,  storageLevel: StorageLevel)(  **implicit** srcOrd: Ordering[ID]): (RDD[(Int, InBlock[ID])], RDD[(Int, OutBlock)]) = {  **val** inBlocks = ratingBlocks.map {  **case** ((srcBlockId, dstBlockId), *RatingBlock*(srcIds, dstIds, ratings)) =>  // The implementation is a faster version of  // val dstIdToLocalIndex = dstIds.toSet.toSeq.sorted.zipWithIndex.toMap  **val** start = System.*nanoTime*()  **val** dstIdSet = **new** OpenHashSet[ID](1 << 20)  dstIds.foreach(dstIdSet.add)  **val** sortedDstIds = **new** Array[ID](dstIdSet.size)  **var** i = 0  **var** pos = dstIdSet.nextPos(0)  **while** (pos != -1) {  sortedDstIds(i) = dstIdSet.getValue(pos)  pos = dstIdSet.nextPos(pos + 1)  i += 1  }  *assert*(i == dstIdSet.size)  Sorting.*quickSort*(sortedDstIds)  **val** dstIdToLocalIndex = **new** OpenHashMap[ID, Int](sortedDstIds.length)  i = 0  **while** (i < sortedDstIds.length) {  dstIdToLocalIndex.update(sortedDstIds(i), i)  i += 1  }  logDebug(  "Converting to local indices took " + (System.*nanoTime*() - start) / 1e9 + " seconds.")  **val** dstLocalIndices = dstIds.map(dstIdToLocalIndex.apply)  (srcBlockId, (dstBlockId, srcIds, dstLocalIndices, ratings))  }.groupByKey(**new** ALSPartitioner(srcPart.numPartitions))  .mapValues { iter =>  **val** builder =  **new** UncompressedInBlockBuilder[ID](**new** LocalIndexEncoder(dstPart.numPartitions))  iter.foreach { **case** (dstBlockId, srcIds, dstLocalIndices, ratings) =>  builder.add(dstBlockId, srcIds, dstLocalIndices, ratings)  }  builder.build().compress()  }.setName(prefix + "InBlocks")  .persist(storageLevel)  **val** outBlocks = inBlocks.mapValues { **case** *InBlock*(srcIds, dstPtrs, dstEncodedIndices, \_) =>  **val** encoder = **new** LocalIndexEncoder(dstPart.numPartitions)  **val** activeIds = Array.*fill*(dstPart.numPartitions)(mutable.ArrayBuilder.*make*[Int])  **var** i = 0  **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)) {  activeIds(dstBlockId) += i // add the local index in this out-block  seen(dstBlockId) = **true** }  j += 1  }  i += 1  }  activeIds.map { x =>  x.result()  }  }.setName(prefix + "OutBlocks")  .persist(storageLevel)  (inBlocks, outBlocks) } |

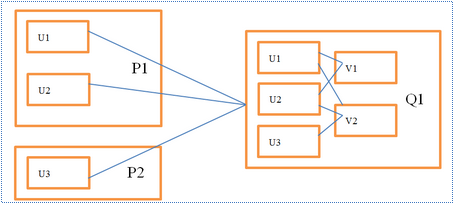
inBlocks和outBlocks的解释和理解：

  获取inblocks和outblocks数据是数据处理的重点。我们知道，通信复杂度是分布式实现一个算法时要重点考虑的问题，不同的实现可能会对性能产生很大的影响。我们假设最坏的情况：即求解商品需要的所有用户特征都需要从其它节点获得。 如下图3.1所示，求解v1需要获得u1,u2，求解v2需要获得u1,u2,u3等，在这种假设下，每步迭代所需的交换数据量是O(m\*rank)，其中m表示所有观察到的打分集大小，rank表示特征数量。



从上图中，我们知道，如果计算v1和v2是在同一个分区上进行的，那么我们只需要把u1和u2一次发给这个分区就好了，而不需要将u2分别发给v1,v2，这样就省掉了不必要的数据传输。

  下图描述了如何在分区的情况下通过U来求解V，注意节点之间的数据交换量减少了。使用这种分区结构，我们需要在原始打分数据的基础上额外保存一些信息。



在Q1中，我们需要知道和v1相关联的用户向量及其对应的打分，从而构建最小二乘问题并求解。这部分数据不仅包含原始打分数据，还包含从每个用户分区收到的向量排序信息，在代码里称作InBlock。在P1中，我们要知道把u1,u2 发给Q1。我们可以查看和u1相关联的所有产品来确定需要把u1发给谁，但每次迭代都扫一遍数据很不划算，所以在spark的实现中只计算一次这个信息，然后把结果通过RDD缓存起来重复使用。这部分数据我们在代码里称作OutBlock。 所以从U求解V，我们需要通过用户的OutBlock信息把用户向量发给商品分区，然后通过商品的InBlock信息构建最小二乘问题并求解。从V求解U，我们需要商品的OutBlock信息和用户的InBlock信息。所有的InBlock和OutBlock信息在迭代过程中都通过RDD缓存。打分数据在用户的InBlock和商品的InBlock各存了一份，但分区方式不同。这么做可以避免在迭代过程中原始数据的交换。

|  |
| --- |
| makeBlocks:  */\*\*  \* Creates in-blocks and out-blocks from rating blocks.  \*  \** ***@param prefix*** *prefix for in/out-block names  \** ***@param ratingBlocks*** *rating blocks  \** ***@param srcPart*** *partitioner for src IDs  \** ***@param dstPart*** *partitioner for dst IDs  \** ***@return*** *(in-blocks, out-blocks)  \*/* **private def** makeBlocks[ID: ClassTag](  prefix: String,  ratingBlocks: RDD[((Int, Int), RatingBlock[ID])],  srcPart: Partitioner,  dstPart: Partitioner,  storageLevel: StorageLevel)(  **implicit** srcOrd: Ordering[ID]): (RDD[(Int, InBlock[ID])], RDD[(Int, OutBlock)]) = {  **val** inBlocks = ratingBlocks.map {  **case** ((srcBlockId, dstBlockId), *RatingBlock*(srcIds, dstIds, ratings)) =>  // The implementation is a faster version of  // val dstIdToLocalIndex = dstIds.toSet.toSeq.sorted.zipWithIndex.toMap  **val** start = System.*nanoTime*()  **val** dstIdSet = **new** OpenHashSet[ID](1 << 20)  dstIds.foreach(dstIdSet.add)  **val** sortedDstIds = **new** Array[ID](dstIdSet.size)  **var** i = 0  **var** pos = dstIdSet.nextPos(0)  **while** (pos != -1) {  sortedDstIds(i) = dstIdSet.getValue(pos)  pos = dstIdSet.nextPos(pos + 1)  i += 1  }  *assert*(i == dstIdSet.size)  Sorting.*quickSort*(sortedDstIds)  **val** dstIdToLocalIndex = **new** OpenHashMap[ID, Int](sortedDstIds.length)  i = 0  **while** (i < sortedDstIds.length) {  dstIdToLocalIndex.update(sortedDstIds(i), i)  i += 1  }  **val** dstLocalIndices = dstIds.map(dstIdToLocalIndex.apply)  (srcBlockId, (dstBlockId, srcIds, dstLocalIndices, ratings))  }.groupByKey(**new** ALSPartitioner(srcPart.numPartitions))  .mapValues { iter =>  **val** builder =  **new** UncompressedInBlockBuilder[ID](**new** LocalIndexEncoder(dstPart.numPartitions))  iter.foreach { **case** (dstBlockId, srcIds, dstLocalIndices, ratings) =>  builder.add(dstBlockId, srcIds, dstLocalIndices, ratings)  }  builder.build().compress()  }.setName(prefix + "InBlocks")  .persist(storageLevel)  } |

这段代码首先对ratingBlocks数据集作map操作，将ratingBlocks转换成（商品分区id，（用户分区id，商品集合，用户id在分区中相对应的位置，打分）这样的集合形式。然后对这个数据集作groupByKey操作，以商品分区id为key值，处理key对应的值，将数据集转换成（商品分区id，InBlocks）的形式。 这里值得我们去分析的是输入块（InBlock）的结构。

|  |
| --- |
| val outBlocks = inBlocks.mapValues { case InBlock(srcIds, dstPtrs, dstEncodedIndices, \_) =>  val encoder = new LocalIndexEncoder(dstPart.numPartitions)  val activeIds = Array.fill(dstPart.numPartitions)(mutable.ArrayBuilder.make[Int])  var i = 0  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)) {  activeIds(dstBlockId) += i // add the local index in this out-block  seen(dstBlockId) = true  }  j += 1  }  i += 1  }  activeIds.map { x =>  x.result()  }  }.setName(prefix + "OutBlocks")  .persist(storageLevel) |

这段代码中，inBlocks表示用户的输入分区块，格式为（用户分区id，（不重复的用户id集，用户位置偏移集，商品id集对应的编码集，打分集））。 activeIds表示商品分区中涉及的用户id集，也即上文所说的需要发送给确定的商品分区的用户信息。activeIds是一个二维数组，第一维表示分区，第二维表示用户id集。用户OutBlocks的最终格式是（用户分区id，OutBlocks）。

  通过用户的OutBlock把用户信息发给商品分区，然后结合商品的InBlock信息构建最小二乘问题，我们就可以借此解得商品的极小解。反之，通过商品OutBlock把商品信息发送给用户分区，然后结合用户的InBlock信息构建最小二乘问题，我们就可以解得用户解。 第（6）步会详细介绍如何构建最小二乘。

接着初始化用户特征矩阵和商品特征矩阵来进行交替计算。交换最小二乘算法是分别固定用户特征矩阵和商品特征矩阵来交替计算下一次迭代的商品特征矩阵和用户特征矩阵。通过下面的代码初始化第一次迭代的特征矩阵。

var userFactors = initialize(userInBlocks, rank, seedGen.nextLong())

var itemFactors = initialize(itemInBlocks, rank, seedGen.nextLong())

  初始化后的userFactors的格式是（用户分区id，用户特征矩阵factors），其中factors是一个二维数组，第一维的长度是用户数，第二维的长度是rank数。初始化的值是异或随机数的F范式。itemFactors的初始化与此类似。

|  |
| --- |
| **for** (iter <- 0 until maxIter) {  itemFactors = *computeFactors*(userFactors, userOutBlocks, itemInBlocks, rank, regParam,  userLocalIndexEncoder, solver = solver)  **if** (shouldCheckpoint(iter)) {  **val** deps = itemFactors.dependencies  itemFactors.checkpoint()  itemFactors.count() // checkpoint item factors and cut lineage  ALS.*cleanShuffleDependencies*(sc, deps)  deletePreviousCheckpointFile()  previousCheckpointFile = itemFactors.getCheckpointFile  }  userFactors = *computeFactors*(itemFactors, itemOutBlocks, userInBlocks, rank, regParam,  itemLocalIndexEncoder, solver = solver) } |

  构建最小二乘的方法是在computeFactors方法中实现的。我们以商品inblock信息结合用户outblock信息构建最小二乘为例来说明这个过程。代码首先用用户outblock与userFactor进行join操作，然后以商品分区id为key进行分组。 每一个商品分区包含一组所需的用户分区及其对应的用户factor信息，格式即（用户分区id集，用户分区对应的factor集）。紧接着，用商品inblock信息与merged进行join操作，得到商品分区所需要的所有信息，即（商品inblock，（用户分区id集，用户分区对应的factor集））。 有了这些信息，构建最小二乘的数据就齐全了。详细代码如下：

|  |
| --- |
| */\*\*  \* Compute dst factors by constructing and solving least square problems.  \*  \** ***@param srcFactorBlocks*** *src factors  \** ***@param srcOutBlocks*** *src out-blocks  \** ***@param dstInBlocks*** *dst in-blocks  \** ***@param rank*** *rank  \** ***@param regParam*** *regularization constant  \** ***@param srcEncoder*** *encoder for src local indices  \** ***@param implicitPrefs*** *whether to use implicit preference  \** ***@param alpha*** *the alpha constant in the implicit preference formulation  \** ***@param solver*** *solver for least squares problems  \** ***@return*** *dst factors*  **val** srcOut = srcOutBlocks.join(srcFactorBlocks).flatMap {  **case** (srcBlockId, (srcOutBlock, srcFactors)) =>  srcOutBlock.view.zipWithIndex.map { **case** (activeIndices, dstBlockId) =>  (dstBlockId, (srcBlockId, activeIndices.map(idx => srcFactors(idx))))  } } **val** merged = srcOut.groupByKey(**new** ALSPartitioner(dstInBlocks.partitions.length)) dstInBlocks.join(merged) |

我们知道求解商品值时，我们需要通过所有和商品关联的用户向量信息来构建最小二乘问题。这里有两个选择，第一是扫一遍InBlock信息，同时对所有的产品构建对应的最小二乘问题； 第二是对于每一个产品，扫描InBlock信息，构建并求解其对应的最小二乘问题。第一种方式复杂度较高，具体的复杂度计算在此不作推导。spark选取第二种方法求解最小二乘问题，同时也做了一些优化。 做优化的原因是二种方法针对每个商品，都会扫描一遍InBlock信息，这会浪费较多时间，为此，将InBlock按照商品id进行排序（前文已经提到过），我们通过一次扫描就可以创建所有的最小二乘问题并求解。 构建代码如下所示：

|  |
| --- |
| while (j < dstIds.length) {  ls.reset()  var i = srcPtrs(j)  var numExplicits = 0  while (i < srcPtrs(j + 1)) {  val encoded = srcEncodedIndices(i)  val blockId = srcEncoder.blockId(encoded)  val localIndex = srcEncoder.localIndex(encoded)  val srcFactor = sortedSrcFactors(blockId)(localIndex)  val rating = ratings(i)  ls.add(srcFactor, rating)  numExplicits += 1  i += 1  }  dstFactors(j) = solver.solve(ls, numExplicits \* regParam)  j += 1  } |

如何进行矩阵分解呢？

因为相应的评价都是非负数，所以在进行矩阵分解的时候，使用的非负最小二乘法进行分解。

## 1 最小二乘法

### 1.1 最小二乘问题

在某些最优化问题中，目标函数由若干个函数的平方和构成，它的一般形式如下所示：



其中x=（x1,x2,…,xn），一般假设m>=n。把极小化这类函数的问题称为最小二乘问题。



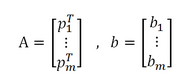
当$f\_{i}(x)$为x的线性函数时，称（1.2）为线性最小二乘问题，当$f\_{i}(x)$为x的非线性函数时，称（1.2）为非线性最小二乘问题。

### 1.2 线性最小二乘问题

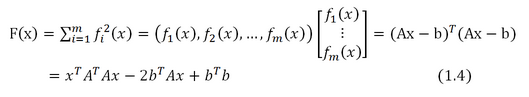
在公式（1.1）中，假设



其中，p是n维列向量，bi是实数，这样我们可以用矩阵的形式表示（1.1）式。令:



 A是m \* n矩阵，b是m维列向量。则:



因为F(x)是凸的，所以对（1.4）求导可以得到全局极小值，令其导数为0，我们可以得到这个极小值。

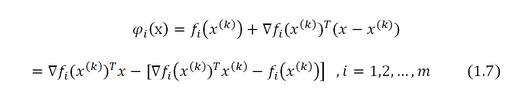


假设A为满秩，$A^{T}A$为n阶对称正定矩阵，我们可以求得x的值为以下的形式：

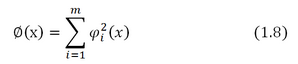


### 1.3 非线性最小二乘问题

假设在（1.1）中，$f\_{i}(x)$为非线性函数，且F(x)有连续偏导数。由于$f\_{i}(x)$为非线性函数，所以（1.2）中的非线性最小二乘无法套用（1.6）中的公式求得。 解这类问题的基本思想是，通过解一系列线性最小二乘问题求非线性最小二乘问题的解。设$x^{(k)}$是解的第k次近似。在$x^{(k)}$时，将函数$f\_{i}(x)$线性化，从而将非线性最小二乘转换为线性最小二乘问题， 用（1.6）中的公式求解极小点$x^{(k+1)}$ ，把它作为非线性最小二乘问题解的第k+1次近似。然后再从$x^{(k+1)}$出发，继续迭代。下面将来推导迭代公式。令



上式右端是$f\_{i}(x)$在$x^{(k)}$处展开的一阶泰勒级数多项式。令



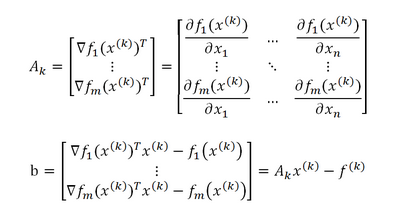
用∅(x)近似F(x)，从而用∅(x)的极小点作为目标函数F(x)的极小点的估计。现在求解线性最小二乘问题:



把（1.9）写成



在公式（1.10）中，



将$A\_{k}$和b带入公式（1.5）中，可以得到，



如果$A\_{k}$为列满秩，且$A\_{k}^{T}A\_{k}$是对称正定矩阵，那么由（1.11）可以得到x的极小值。



  可以推导出$2A\_{k}^{T}f\_{k}$是目标函数F(x)在$x^{(k)}$处的梯度，$2A\_{k}^{T}A\_{k}$是函数∅(x)的海森矩阵。所以（1.12）又可以写为如下形式。



  公式（1.13）称为Gauss-Newton公式。向量



为点$x^{(k)}$处的Gauss-Newton方向。为保证每次迭代能使目标函数值下降（至少不能上升），在求出$d^{(k)}$后，不直接使用$x^{(k)}+d^{(k)}$作为k+1次近似，而是从$x^{(k)}$出发，沿$d^{(k)}$方向进行一维搜索。



求出步长$\lambda ^{(k)}$后，令



**最小二乘的计算步骤如下:**

（1） 给定初始点$x^{(1)}$，允许误差ε>0，k=1

（2） 计算函数值$f\_{i}(x)$，得到向量$f^{(k)}$，再计算一阶偏导，得到m\*n矩阵$A\_{(k)}$

（3） 解方程组（1.14）求得Gauss-Newton方向$d^{(k)}$

（4）从$x^{(k)}$出发，沿着$d^{(k)}$作一维搜索，求出步长$\lambda ^{(k)}$ ，并令$x^{(k+1)}=x^{(k)}-\lambda d^{(k)}$

（5）若$||x^{(k+1)}-x^{(k)}||<=\epsilon$停止迭代，求出x，否则，k=k+1，返回步骤（2）

  在某些情况下，矩阵$A^{T}A$是奇异的，这种情况下，我们无法求出它的逆矩阵，因此我们需要对其进行修改。用到的基本技巧是将一个正定对角矩阵添加到$A^{T}A$上，改变原来矩阵的特征值结构，使其变成条件较好的对称正定矩阵。 典型的算法是Marquardt。



其中，I是n阶单位矩阵，alpha是一个正实数。当alpha为0时，$d^{(k)}$就是Gauss-Newton方向，当alpha充分大时，这时$d^{(k)}$接近F(x)在$x^{(k)}$处的最速下降方向。

## 2 共轭梯度法

### 2.1 共轭方向

**定义2.1** 设A是n\*n对称正定矩阵，若两个方向$d^{(1)}$ 和$d^{(2)}$满足



则称这两个方向关于A共轭。若$d^{(1)},d^{(2)},...,d^{(k)}$是k个方向，它们两两关于A共轭，则称这组方向是关于A共轭的。即



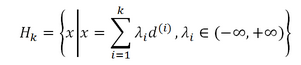
在上述定义中，如果A是单位矩阵，那么两个方向关于A共轭等价于两个方向正交。如果A是一般的对称正定矩阵，$d^{(i)}$与$d^{(j)}$共轭，就是$d^{(i)}$与$Ad^{(j)}$正交。共轭方向有一些重要的性质。

**定理2.1** 设A是n阶对称正定矩阵，$d^{(1)},d^{(2)},...,d^{(k)}$是k个A的共轭的非零向量，则这个向量组线性无关。

**定理2.2 (扩张子空间定理）** 设有函数



其中，A是n阶对称正定矩阵，$d^{(1)},d^{(2)},...,d^{(k)}$是k个A的共轭的非零向量，以任意的$x^{(1)}$为初始点， 沿$d^{(1)},d^{(2)},...,d^{(k)}$进行一维搜索，得到$x^{(2)},x^{(3)},...,x^{(k+1)}$，则$d^{(k+1)}$是线性流型 $x^{(1)}+H\_{k}$上的唯一极小点，特别的，当k=n时，$x^{(n+1)}$是函数f(x)的唯一极小点。其中，



是$d^{(1)},d^{(2)},...,d^{(k)}$生成的子空间。

考虑问题



其中A是对称正定矩阵，c是常数。

具体求解方式如下：

  首先给定任何一个初始点$x^{(1)}$ ，计算目标函数f(x)在这点的梯度$g\_{(1)}$ ，若$||g\_{(1)}||=0$ ，则停止计算；否则令:



沿方向$d^{(1)}$搜索，得到点$x^{(2)}$ 。计算$x^{(2)}$处的梯度，若$||g\_{(2)}||!=0$，则利用$g\_{(2)}$和$d^{(1)}$构造第二个搜索方向$d^{(2)}$，再沿$d^{(2)}$搜索。

 一般的，若已知$x^{(k)}$和搜索方向$d^{(k)}$，则从$x^{(k))}$出发，沿方向$d^{(k)}$搜索，得到



其中步长lambda满足



此时可以求得lambda的显式表达。令



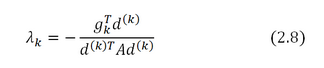
通过求导可以求上面公式的极小值，即



根据二次函数梯度表达式，（2.6）式可以推出如下方程



由（2.7）式可以得到



 计算f(x)在$x^{(k+1)}$处的梯度，若$||g\_{k+1}||=0$ ，则停止计算， 否则用$g\_{(k+1)}$和$d^{(k)}$构造下一个搜索方向$d^{(k+1)}$ ，并使$d^{(k)}$与$d^{(k+1)}$共轭。按照这种设想，令



在公式（2.9）两端同时乘以$d^{(k)T}A$，并令

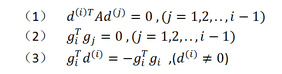


 可以求得



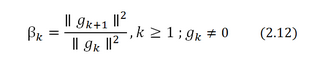
再从$x^{(k+1)}$出发，沿$d^{(k+1)}$方向搜索。综上分析 ，在第1个搜索方向取负梯度的前提下，重复使用公式（2.5）、（2.8）、（2.9）、（2.11），我们就能够构造一组搜索方向。当然，前提是这组方向是关于A共轭的。 定理2.3说明了这组方向是共轭的。

**定理2.3** 对于正定二次函数（2.3），具有精确一维搜索的的共轭梯度法在m<=n次一维搜索后终止，并且对于所有i(1<=i<=m)，下列关系成立：

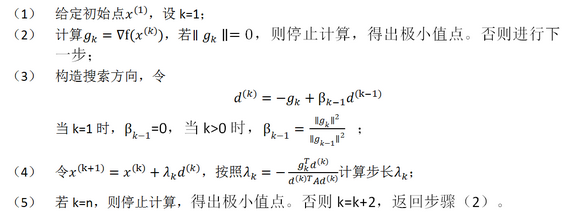


  还可以证明，对于正定二次函数，运用共轭梯度法时，不做矩阵运算也可以计算变量beta\_k。

**定理2.4** 对于正定二次函数，共轭梯度法中因子beta\_k具有下列表达式



对于二次凸函数，共轭梯度法的计算步骤如下：



### 2.2 共轭梯度法

## 3 最小二乘法在spark中的具体实现

Spark ml中解决最小二乘可以选择两种方式，一种是非负正则化最小二乘，一种是乔里斯基分解（Cholesky）。

乔里斯基分解分解是把一个对称正定的矩阵表示成一个上三角矩阵U的转置和其本身的乘积的分解。在ml代码中，直接调用[netlib-java](https://github.com/fommil/netlib-java)封装的dppsv方法实现。

|  |
| --- |
| lapack.dppsv(“u”, k, 1, ne.ata, ne.atb, k, info) |

可以深入dppsv代码（Fortran代码）了解更深的细节。我们分析的重点是非负正则化最小二乘的实现，因为在某些情况下，方程组的解为负数是没有意义的。虽然方程组可以得到精确解，但却不能取负值解。在这种情况下，其非负最小二乘解比方程的精确解更有意义。非负最小二乘问题要求解的问题如下公式



其中ata是半正定矩阵。

  在ml代码中，org.apache.spark.mllib.optimization.NNLS对象实现了非负最小二乘算法。该算法结合了投影梯度算法和共轭梯度算法来求解。

|  |
| --- |
| **class** Workspace(**val** n: Int) {  **val** *scratch* = **new** Array[Double](n)  **val** *grad* = **new** Array[Double](n) //梯度投影  **val** *x* = **new** Array[Double](n)  **val** *dir* = **new** Array[Double](n) //搜索方向  **val** *lastDir* = **new** Array[Double](n)  **val** *res* = **new** Array[Double](n) //梯度   **def** wipe(): Unit = {  ju.Arrays.*fill*(*scratch*, 0.0)  ju.Arrays.*fill*(*grad*, 0.0)  ju.Arrays.*fill*(*x*, 0.0)  ju.Arrays.*fill*(*dir*, 0.0)  ju.Arrays.*fill*(*lastDir*, 0.0)  ju.Arrays.*fill*(*res*, 0.0)  } |

  在Workspace中，res表示梯度，grad表示梯度的投影，dir表示迭代过程中的搜索方向（共轭梯度中的搜索方向$d^{(k)}$），scratch代表公式（2.8）中的 $d^{(k)T}A$。

NNLS对象中，sort方法用来解最小二乘，它通过迭代求解极小值。我们将分步骤剖析该方法。

具体的方法在：solve中

|  |
| --- |
| */\*\*  \* Solve a least squares problem, possibly with nonnegativity constraints, by a modified  \* projected gradient method. That is, find x minimising ||Ax - b||\_2 given A*^*T A and A*^*T b.  \*  \* We solve the problem  \* min\_x 1/2 x*^*T ata x*^*T - x*^*T atb  \* subject to x >= 0  \*  \* The method used is similar to one described by Polyak (B. T. Polyak, The conjugate gradient  \* method in extremal problems, Zh. Vychisl. Mat. Mat. Fiz. 9(4)(1969), pp. 94-112) for bound-  \* constrained nonlinear programming. Polyak unconditionally uses a conjugate gradient  \* direction, however, while this method only uses a conjugate gradient direction if the last  \* iteration did not cause a previously-inactive constraint to become active.  \*/* **def** solve(ata: Array[Double], atb: Array[Double], ws: Workspace): Array[Double] = {  ws.wipe()  **val** n = atb.length  **val** scratch = ws.*scratch* // find the optimal unconstrained step  **def** steplen(dir: Array[Double], res: Array[Double]): Double = {  **val** top = *blas*.ddot(n, dir, 1, res, 1)  *blas*.dgemv("N", n, n, 1.0, ata, n, dir, 1, 0.0, scratch, 1)  // Push the denominator upward very slightly to avoid infinities and silliness  top / (*blas*.ddot(n, scratch, 1, dir, 1) + 1e-20)  }   // stopping condition  **def** stop(step: Double, ndir: Double, nx: Double): Boolean = {  ((step.isNaN) // NaN  || (step < 1e-7) // too small or negative  || (step > 1e40) // too small; almost certainly numerical problems  || (ndir < 1e-12 \* nx) // gradient relatively too small  || (ndir < 1e-32) // gradient absolutely too small; numerical issues may lurk  )  }   **val** grad = ws.*grad* **val** x = ws.*x* **val** dir = ws.*dir* **val** lastDir = ws.*lastDir* **val** res = ws.*res* **val** iterMax = math.*max*(400, 20 \* n)  **var** lastNorm = 0.0  **var** iterno = 0  **var** lastWall = 0 // Last iteration when we hit a bound constraint.  **var** i = 0  **while** (iterno < iterMax) {  // find the residual  *blas*.dgemv("N", n, n, 1.0, ata, n, x, 1, 0.0, res, 1)  *blas*.daxpy(n, -1.0, atb, 1, res, 1)  *blas*.dcopy(n, res, 1, grad, 1)  // project the gradient  i = 0  **while** (i < n) {  **if** (grad(i) > 0.0 && x(i) == 0.0) {  grad(i) = 0.0  }  i = i + 1  }  **val** ngrad = *blas*.ddot(n, grad, 1, grad, 1)  *blas*.dcopy(n, grad, 1, dir, 1)  // use a CG direction under certain conditions  **var** step = steplen(grad, res)  **var** ndir = 0.0  **val** nx = *blas*.ddot(n, x, 1, x, 1)  **if** (iterno > lastWall + 1) {  **val** alpha = ngrad / lastNorm  *blas*.daxpy(n, alpha, lastDir, 1, dir, 1)  **val** dstep = steplen(dir, res)  ndir = *blas*.ddot(n, dir, 1, dir, 1)  **if** (stop(dstep, ndir, nx)) {  // reject the CG step if it could lead to premature termination  *blas*.dcopy(n, grad, 1, dir, 1)  ndir = *blas*.ddot(n, dir, 1, dir, 1)  } **else** {  step = dstep  }  } **else** {  ndir = *blas*.ddot(n, dir, 1, dir, 1)  }   // terminate?  **if** (stop(step, ndir, nx)) {  **return** x.clone  }   // don't run through the walls  i = 0  **while** (i < n) {  **if** (step \* dir(i) > x(i)) {  step = x(i) / dir(i)  }  i = i + 1  }   // take the step  i = 0  **while** (i < n) {  **if** (step \* dir(i) > x(i) \* (1 - 1e-14)) {  x(i) = 0  lastWall = iterno  } **else** {  x(i) -= step \* dir(i)  }  i = i + 1  }   iterno = iterno + 1  *blas*.dcopy(n, dir, 1, lastDir, 1)  lastNorm = ngrad  }  x.clone } |