可细分部分

可补充次线性方法部分

算法部分被查重？

大数据背景下的机器学习算法

在大数据背景下，大规模数据集要求我们开发出的机器学习算法更加高效，并行度更大。在机器学习研究领域，学者们已经针对大规模数据集做了不少卓有成效的工作。

早期的工作，有比方说PSVM（Parallel Support Vector Machines）[4]的开发。PSVM利用了近似矩阵分解的方法。这种分解方法是基于行抽取的，它可以在算法运行时大大降低内存使用。该算法同时从而可以通过增加参与并行计算节点的数量来减少算法运行时间。该算法针对机群的可扩展性在上百台这样一个数量级上。

此后，研究者又提出了PLDA (Parallel Latent Dirichlet Allocation) [26]算法。该算法可以大幅提高LDA算法的运行效率。它采用随机抽取的方法。PLDA算法所发布的Hadoop版本具有很强的鲁棒性，它具有承受节点失败的能力，这其中既利用了算法的随机性，也利用了Hadoop的容错优势。

最近，Dean等学者的工作[18]表明并行化的优势可以再深度学习（deep learning）领域起到重大作用。他们的工作可以把机器学习算法的并行度推进到上亿个计算节点。这是迄今为止，机器学习研究领域所达到的最大并行度，同时也促使深度学习算法在图像识别、语音识别和很多其他应用上取得了目前最好的学习效果。

除了Hadoop并行系统所代表的框架，机器学习学术界也有很多对其他并行框架的尝试。GraphLab[17]是卡耐基梅隆大学开发的，专门针对机器学习中大规模图算法的一个计算工具。它可以大幅提升此类型算法的运算效率和在机群上的可扩展性。

并行算法框架

本文工作建立在三个并行系统上，这三个系统在运行机器学习算法时的算法框架都有其各自特点。

Apache Hadoop[27]作为一个软件代码库，实现了对大规模数据集的分布式处理。它被广泛应用与各类机器集群上，并且其使用了简洁的编程模型。它是一个开源并行框架，支持对高密度分布式数据的处理。而Hadoop所支持的机器集群可以是大规模的并且支持商品级硬件。Hadoop是从Google的MapReduce和GFS（Google File System）中演化而来的。Hadoop使用Java语言。它的开源特性保证了全球大量开发者的贡献都体现在了这个Apache的顶层项目中。Hadoop平台在一般意义上，包含Hadoop kernel，MapReduce和Hadoop Distributed File System (HDFS)，以及一些其他相关的项目，包括Apache Hive和Apache HBase。Hadoop的设计目标就是要大幅提高机器集群的可扩展性，从几台服务器上升到成千上万个运算节点。在Hadoop中每个节点都能提供本地的计算和存储。Hadoop框架同时对运行在其上的应用同时透明的提供数据可靠性保证以及数据通信保证。

Hadoop所执行的运算模型为MapReduce[7]。每个应用被分割为大量小片段，每个片段可以在机器集群中的任意一个节点上执行和重复执行。在MapReduce运算模型能够应用的问题中，输入会被解析为键-值对的集合。一个map函数会把这些键值对经过用户所定义的计算转为中间结果的键-值对。一个reduce函数会接着把这些中间结果按照键值索引聚集（其聚集方法是用户所定义的）。事实上，MapReduce所能应用的问题很广泛，不少问题都可以转化为一个或者一系列MapReduce任务。这种运算模型也使得它很易于底层的并行实现。所有的处理都是独立的，所以可以分配给不同的节点分别执行。除此之外，Hadoop还提供了分布式的文件系统HDFS。它可以给整个机器集群提供很高的通信带宽。这用map/reduce以及HDFS的设计保证了当节点失败时可以自动被系统处理而不影响整个任务执行。而这种容错性保证，并不依赖于硬件，而是Hadoop代码库本身的设计在应用层考虑了节点失败的检测和处理。所以尽管每个节点可能是易于出现问题的，但整个机器集群作为一个运算资源来说是可靠地。

Apache Mahout[20]是一个专门针对大规模机器学习算法所开发的代码库。其核心设计目标就是要提高在大规模数据上，提高机器学习算法的可扩展性。它的核心算法包括聚类算法、分类算法和协同过滤。这些算法的很大一部分是建立在Apache Hadoop的基础之上运用MapReduce框架的。同时，Mahout的核心代码库也含有一部分非并行计算程序（针对罗吉斯回归模型的算法就属于这一部分）。它们在代码优化上做了大量的工作，从而达到了很多好的运算效果。Mahout甚至广泛应用与商业领域，它良好的可扩展性，以及对很多流行机器学习算法的完整支持都促进了它的推广。

目前，Mahout主要支持以下四个应用场景：

1. 推荐系统：把用户行为作为输入，并根据用户行为进行个性化推荐。
2. 聚类：例如把文档按照主题进行分类。
3. 分类：例如从已经分好类的文档中学习模型，并将一个新的未被分类的文档分入正确的分类中。
4. 频繁模式：将一系列事项集合作为输入（例如查询词、销售图表等等），从中识别出那些事项模式出现的较为频繁。

Mahout当前支持很多机器学习算法，包括协同过滤算法、基于用户和基于产品的推荐算法、K-Means算法、模糊K-means聚类算法、平均偏移聚类算法、狄利克雷过程聚类算法、LDA算法、奇异值分解算法、并行频繁模式算法、互补朴素贝叶斯分类器和随机决策树分类器。

Spark[29]系统是由加州大学伯克利分校的AMPLab开发的。它是一个开源的分布式计算系统。Spark的设计目标是使得数据分析更加快速：一方面是运行时间的加速，另一方面是读写的加速。为了使程序运行得更快，Spark提供了基于内存和缓存的集群计算基本操作：一个任务可以将数据最充分地载入内存中，从而使得反复不断地访问操作要比硬盘读写来的更快。而硬盘读写多这一点，正是Hadoop系统的最大劣势。为了使得在Spark系统上的编程开发更为快速，Spark支持Scala和Python语言，并提供了简洁的API。用户甚至可以以交互式的方式，通过Scala和Python语言来快速处理大数据集。Spark系统在设计之初，主要是面向两类应用问题：迭代程序（这是在机器学习算法中广泛使用的）和交互式数据挖掘。如果能够把需要重复利用的数据始终保存在内存中，这两类应用问题的算法执行效率都必然能够获得极大提升。而随后的测试也表明，相比于Hadoop系统，Spark确实在这两个应用中取得了100倍以上的加速效果。

Spark系统较为年轻，但也已经在工业界获得了不少应用。Spark系统是Shark系统的后端引擎，而Shark系统是一个与Apache Hive系统相兼容的数据仓储系统。同样的，Shark系统也比Hive系统有100倍以上的加速。尽管Spark系统是新开发的一个并行框架，它支持访问HDFS上的数据，从这个意义上讲，它也与Hadoop兼容。这个特性使得很多基于Hadoop的程序不用再重新开发，从而给Spark也带来了更广的应用前景。

Spark是建立在Mesos上的系统。Mesos是一种运行于集群上的操作系统。Mesos可以使得多个并行程序良好地共享一个集群。它也同时提供了API来支持其上的应用在集群上部署并行任务。Spark通过利用Mesos系统可以在集群上与其他系统良好共存，比如说同样建立在Mesos系统上的Hadoop和MPI。另外，利用Mesos系统，使得Spark的开发过程节约了原本所需的大量编程工作。

Spark系统得以成功的关键在于运用了弹性分布式数据集（Resilient Distributed Dataset，简称RDD）这个概念。每个弹性分布式数据集代表一块分布在集群中一部分节点上的只读对象。它具有容错特性，既当一个弹性分布式数据集的分布方式信息丢失时，系统可以自动重构这个弹性分布式数据集，从而保证数据的可靠性。用户可以清晰地在应用程序代码中创建RDD，并根据需要把它载入整个集群的缓存中，并在像如前所述的MapReduce框架下的map和reduce函数中不断重复访问这些数据。弹性分布式数据集通过仅支持线性操作（lineage）来保证容错性。如果一个弹性分布式数据集丢失，系统可以通过它所对应的线性操作方式（也可以是一串线性操作）来对重新生成这个弹性分布式数据集。

次线性方法

近些年来，次线性方法逐渐为人所熟知。Clarkson等学者充分运用随机算法提出了这一新的方法[5]。他们在机器学习领域首先提出了这种次线性时间的近似优化算法。他们把这个方法应用在线性分类器和最小闭包球这两个基础问题上。次线性方法结合了一种新型抽样方法和一种新的可乘式更新算法。他们同事证明了算法下界。该算法下界表明算法理论运行时间对于RAM模型来说，已接近最优。

Hazan等学者把这种次线性方法用在了带二阶惩罚项的支持向量机模型上[16]。此后，Cotter更进一步，把次线性方法运用在了了带核函数的支持向量机模型中[6]。随后，Hazan继续把此方法做进一步推广，分别发展到了带一阶惩罚项和带二阶惩罚项的线性回归模型中[15]。Garber和Hazan合作，又把此方法用在了半正定规划问题（Semidenfinite Programming）当中[10]。在2012年，彭昊若等研究人员，在次线性方法上做了进一步研究，把它用在了解决带一阶或二阶惩罚项的罗吉斯回归模型中，并提出了对应的串行算法[22]。

带惩罚项的罗吉斯回归模型和次线性方法

罗吉斯回归模型在机器学习领域中的分类问题上获得了广泛应用。在本文中，特别的，我们将讨论二分类问题。具体的模型定义如下：

1 假设有一个训练数据集为



其中  代表输入的训练样本，而  是相对应的分类标签。

这里  是样本个数，即样本空间大小；而  是样本维度，即特征空间大小。

为了简化表达起见，我们可以将训练数据重组为两部分。一部分为训练数据矩阵，其中每行代表一个训练样本，每列代表一维特征，即



另一部分为分类标签向量，其中的每个元素与训练数据矩阵中的每一行对应，即



2 不带惩罚项的简单罗吉斯回归模型可以表述为下面的形式



即定义了每个数据样本所对应的分类标签的条件概率。

其中 是回归向量，而  是偏移量。

这两个变量正是学习算法需要近似优化求解的目标。

3 为了优化求解上述变量，对于这种显示表达的模型，一种常用的经典方法是考虑最大似然。更进一步，为了后面的计算方便，这里我们考虑取过对数后的最大似然。由于对数函数本身是单调增函数，所以在优化问题中，优化取对数后的最大似然与优化原始最大似然完全等价。因而，在整个训练数据集上的对数最大似然可以表达为



带惩罚项的罗吉斯回归模型

在带惩罚项的模型框架中，我们往往需要假设回归向量服从某一特定分布 。这一过程也可以称作先验假设。它使得我们可以将最大似然估计转换为考虑最大后验估计（maximum a posteriori，简称MAP）。所以，在整个训练数据集上的最大后验估计可以表达为



我们下面将会分别推导出一阶和二阶带惩罚项的罗吉斯回归模型优化求解目标的表达式。对于二阶模型，我们会引入拉布拉斯先验假设；对于一阶模型，我们会引入高斯先验。

如果我们假设回归向量服从一个高斯分布。假设该高斯分布的均值为 ，协方差矩阵为  。其中 表示一个  的单位矩阵。数学上表达为，



那么，由于



在这种情况下，在式中带入上述表达式，得到



从而，我们就把罗吉斯回归模型的参数优化问题发展为一个带二阶惩罚项的优化目标。

带二阶惩罚项进行优化的优势在于可以得到更为稳定的优化近似解，在算法迭代过程中促进更为快速的求解收敛。

同样的过程，如果我们假设回归向量服从一个参数为 的拉普拉斯分布。我们可以得到，



那么，在这种情况下，在式中带入上述表达式，得到



从而，我们就把罗吉斯回归模型的参数优化问题发展为一个带一阶惩罚项的优化目标。

带一阶惩罚项进行优化的相比于带二阶惩罚项进行优化往往更有优势。其独特性在于，一阶惩罚项易于进行稀疏模型构建[16]。因而，从带一阶惩罚项的罗吉斯回归问题求解得到的结果往往既有良好的分类效果，也可以同时用来进行对数据的特征选择。

次线性方法

我们所使用的次线性方法的框架是一个分别同时处理的硬边际量和软边际量的混合方法。这种方法可以在同时在硬边际量和软边际量的优化求解过程中进行快速收敛。

在次线性方法中，每个迭代过程需要经过两步。第一步是随机原始更新（Stochastic Primal Update）。它又包含以下两个步骤：

1 从样本数据中随机抽取出一个。其中，每个样本数据的被抽取的概率组成长度为的概率分布向量  。（初始为每个样本等概率，随后概率分布向量 会被不断更新。）

2 通过计算上一步骤被抽取的样本数据的梯度方向，来更新回归向量以及偏移量。注意，这是一考虑带惩罚项的在线(online)更新过程。

第二步是随机对偶更新（Stochastic Dual Update）。它也包含两个步骤：

1 对训练数据集中的每个数据样本都使用的时间计算一个在该数据样本上的硬边际量加软边际量的近似估计值。

2 使用针对在单纯形（Simplex）上做在线优化的可乘式更新（Multiplicative Updates，简称MW）算法，来更新第一个步骤中的概率分布向量 。

使用次线性方法的罗吉斯回归模型串行优化算法

我们借助以下符号来定义使用次线性方法的罗吉斯回归模型串行优化算法。我们在引文[22]的基础上做了必要的简化。辅助定义的符号也会在随后涉及的其他算法中使用。

1 我们定义一个投射函数，如下



2 我们用符号代表示性函数，定义为



3 我们用符号来表示罗吉斯函数，即



在算法1中，我们给出了使用次线性方法的罗吉斯回归模型串行优化算法的具体细节。

|  |
| --- |
| Algorithm 1 SLLR |
| 1: Input parameters: |
| 2: Initialize parameters: |
| 3: Iterations: |
| 4: |
| 5: Choose  with probability |
| 6: |
| 7: |
| 8: Update soft margin if input  for -penalty |
| 9: Update  by soft-threshold operations if input  for -penalty |
| 10: |
| 11: |
| 12: Choose  with probability |
| 13: Iterations: |
| 14: |
| 15: |
| 16: |
| 17: Output: |

在上述所示的算法1中，从第4行到第11行是原始更新部分。其中是对梯度方向的估计。从第12行到第16行是对偶更新部分。在这一部分中，是对硬边际量和软边际量加和的估计值。而与此同时，也是的梯度值。尽管第15行和第16行的计算使得成为一个对的带偏移近似值，但这对保障整个算法的稳定性至关重要。而由此所产生的近似误差对最终优化结果的影响很小，可以忽略不计。在这个算法描述中，我们统一了带一阶惩罚项和二阶惩罚项的次线性方法罗吉斯回归模型串行优化算法。两者不同的处理体现在第8行和第9行。如果需要更进一步研究，请参考引文[22]中分开定义的两种算法。

罗吉斯回归模型优化算法并行框架

在此章，我们首先将给出我们在Hadoop MapReduce架构下设计的运用次线性方法的并行优化算法。紧接着，我们将会给出一个在Spark系统上运行的运用次线性方法的并行优化算法。该版本算法与Hadoop MapReduce架构下设计的算法略有不同。然后我们将介绍在我们测试中作为基准所使用的传统算法。它们包括在Spark系统下运行的并行梯度下降方法，以及在Mahout中集成的在线随机梯度下降法。

Hadoop MapReduce架构下使用次线性方法的并行优化算法

我们通过充分利用MapReduce的变成模式，在Hadoop系统上设计了一个针对罗吉斯回归模型的并行优化算法。具体的算法请看下面的算法2、原始Map过程、原始Reduce过程、原始更新过程、对偶Map过程和对偶更新过程。

算法2 并行-次线性-罗吉斯回归模型-MapReduce架构算法

|  |
| --- |
| Algorithm 2 PSUBPLR-MR |
| 1: Input parameters: |
| 2: Initialize parameters: |
| 3: Iterations: |
| 4: storeInHdfsFile("hdfs://paraw").addToDistributedCache() |
| 5: storeInHdfsFile("hdfs://parap").addToDistributedCache() |
| 6: conf\_primalnew Configuration() |
| 7: job\_primalnew MapReduce-Job(conf\_primal) |
| 8: conf\_primal.passParameters() |
| 9: job\_primal.setInputPath("...") |
| 10: job\_primal.setOutputPath("tmp/primal") |
| 11: job\_primal.run() |
| 12: ()PrimalUpdate() |
| 13: Choose  with probability |
| 14: storeInHdfsFile("hdfs://paraw").addToDistributedCache() |
| 15: conf\_dual  new Configuration() |
| 16: job\_dual  new MapReduce-Job(conf\_dual) |
| 17: conf\_dual.passParameters() |
| 18: job\_primal.setInputPath("...") |
| 19: job\_dual.setOutputPath("tmp/dual") |
| 20: job\_dual.run() |
| 21: DualUpdate() |
| 22: Output: |

过程 原始Map

|  |
| --- |
| Procedure Primal-Map(inputfile) |
| 1: Configuration.getParameters() |
| 2: readCachedHdfsFile("paraw") |
| 3: readCachedHdfsFile("parap") |
| 4: parseRowIndx(inputfile) |
| 5: parseRowVector(inputfile) |
| 6: parseRowLabel(inputfile) |
| 7: *random(seed)* |
| 8: *if* |
| 9: |
| 10: *else* |
| 11: |
| 12: Iterations: |
| 13: Set |
| 14: Set |
| 15: *Output* (key, value) |

过程 原始Reduce

|  |
| --- |
| Procedure Primal-Reduce(key\_in, value\_in) |
| 1: key\_outkey\_in |
| 2: |
| 3: *Output* (key\_out, value\_out) |

过程 原始更新

|  |
| --- |
| Procedure PrimalUpdate() |
| 1: readFromHdfsFile("tmp/primal") |
| 2: |
| 3: |
| 4: |

过程 对偶Map

|  |
| --- |
| Procedure Dual-Map(inputfile) |
| 1: Configuration.getParameters() |
| 2: readCachedHdfsFile("paraw") |
| 3: parseRowIndx(inputfile) |
| 4: parseRowVector(inputfile) |
| 5: parseRowLabel(inputfile) |
| 6: |
| 7: |
| 8: |
| 9: |
| 10: |
| 11: *Output* (key, value) |

过程 对偶更新

|  |
| --- |
| Procedure DualUpdate () |
| 1: readFromHdfsFile("tmp/dual$t$") |
| 2: Iterations: |
| 3: |

算法2的并行设计基本遵循了使用次线性方法的罗吉斯回归模型串行优化算法的算法框架。我们在并行算法2中依然在每次迭代计算中保留了两个主要部分：从第4行到第12行是原始更新过程，而从第13开始，一直到第21行是对偶更新过程。在原始更新过程中，有并行执行的部分，它们是从第4行到第11行部分；但同时，也又无法避免的串行部分，即由第12行的调用PrimalUpdate函数开体现。而在对偶更新的过程中，情况是类似的，从第14行到第21行是并行执行的的部分，但第13行和21行是串行执行的。

更进一步考察，我们事实上可以在实际程序执行中进一步增大并行度，其方法是在同一次迭代计算过程中，同时启动原始MapReduce任务和对偶MapReduce任务。由于这两个MapReduce任务中所访问和修改的参数通过延后更新的准则考虑可以互相隔离，所以这样同时执行的方式和两个任务串行执行的方式是等价的。从而，我们可以在Hadoop MapReduce的框架下更充分地增大并行度，是算法执行更为高效。这样的并行设计在图。。。中得到了更加清晰的说明。

图。。。算法2并行框架设计(需要改)

在算法2的并行执行部分，我们充分考虑了MapReduce本身的设计特点。在算法2的job\_primal中，我们将训练数据矩阵全部载入并进行解析和处理，这符合MapReduce以相同方式同时处理大量数据的设计思想。在原本的次线性串行算法中，我们在原始更新的过程中只随机抽取一个样本数据用来计算梯度；而在新设计的并行框架中，我们计算了“一部分”样本数据上的梯度，并根据概率向量进行了加权叠加，并以Reduce过程的的最终结果作为更新的梯度。实现的细节具体参考原始Map过程和原始Reduce过程。在此，我们使用随机方法来确定计算哪些样本数据上的梯度，并进行加权。从原始Map过程的第7行和第8行可以看出，如果取，那么所有的样本节点都将参与梯度计算。又由于概率向量中的期望值是，所以在算法执行中，我们一般会取。

在算法2的job\_dual中，我们设计的是原始串行算法的简单并行处理。并行算法对每个样本数据根据可乘式更新算法计算出一个值。这个过程可以直接分散到各个计算节点上单独完成，甚至可以省略Reduce的过程。

关于算法2的设计，还有以下三点值得注意：

1 有关参数传递的问题

在Hadoop框架下，合适的选择一种高效地传递参数的方式对性能是尤为关键的。在算法2中，参数传递的过程既有不同迭代步之间的传递，也有不同MapReduce任务之间的传递。同时，另一个挑战是，需要针对Hadoop框架下的HDFS有针对性的进行处理。

很显然，最为快速的通信方式是通过函数进行参数传递。但是，这样的传参方法在Hadoop环境下总是会遇到缓冲区大小限制的问题。一旦当需要传递的参数很大时（高维数据正符合这一点），如果仅使用此方法，将严重危害算法的可扩展性。

另一种方法是把需要传递的参数在信道传递时进行信息压缩。这需要在发送时进行压缩，在接收时进行解压。其弊端是会造成额外的计算开销。并且这种方法也不能从本质上解决参数传递时缓冲区大小限制的问题。

当我们重新回顾Hadoop的设计思想时，可以发现，最自然而然的选择是通过文件传递参数。因为Hadoop本身在自动连接Map任务和Reduce任务之间就包含了通过文件传递键-值对的方法。当然，由于频繁的读写HDFS系统，这种方法并不高效，但这是为了支持大数据环境而采取的合理选择。

2 有关一阶和二阶惩罚项的问题

算法2在数学上严格来说是针对不带惩罚项的罗吉斯回归模型的。如果要是模型带有一阶或二阶惩罚项，我们必须对算法2进行一些小的改动。这些改动只会在原始更新的过程中出现。而改动方法和算法1中第8和第9行所示的方法是完全一致的。这些改动仅发正在串行执行部分，对本身的算法框架和执行没有本质影响。

3 训练数据集的数据稀疏问题

可以被罗吉斯模型所描述的训练数据集，如果具有很高的维度，往往具有数据稀疏的特点。这就要求我们在真正实现算法时，充分考虑这一点，在代码中进行有针对性的优化。但这一点在算法2自身的描述中难以体现，在此特别说明。

我们放弃了针对稠密数据有效地简单存储和运算模式。针对数据稀疏性，我们会分别存储每个样本数据对应的稀疏向量中每个数据点的维度坐标和具体数值。采用这样的方式，自然也会使得所有涉及稀疏向量的运算都要对代码进行相应更改优化，比如说最简单的稀疏向量求点积过程。考虑数据稀疏性的代码优化是极为有效的，也对算法性能带来了显著提升。我们在后文所展示的实验结果均为使用针对数据稀疏性优化后的代码进行测试所得。

Spark系统上运行的运用次线性方法的并行优化算法

我们设计了在Spark系统上运行的运用次线性方法的并行优化算法。具体的算法请看下面的算法3。而其中所使用的原始更新过程和对偶更新过程是与前面介绍的算法2完全相同的。

该版本算法与Hadoop MapReduce架构下设计的算法略有不同。然后我们将介绍在我们测试中作为基准所使用的传统算法。它们包括在Spark系统下运行的并行梯度下降方法，以及在Mahout中集成的在线随机梯度下降法。

算法3 并行-次线性-罗吉斯回归模型-Spark架构算法

|  |
| --- |
| Algorithm 3 PSUBPLR-SPARK |
| 1: Input parameters: |
| 2: Initialize parameters: |
| 3: pointsspark.textFile(inputfile).map(parsePoint()).cache() |
| 4: Iterations: |
| 5: gradpoints.map().reduce( \_+\_ ) |
| 6: ()PrimalUpdate() |
| 7: Choose  with probability |
| 8: pAdjustpoints.map(MW-Update()).reduce(copy()) |
| 9: DualUpdate() |
| 10: Output: () |

算法3与算法2在基本并行框架上的设计非常相似。而它们最主要的区别在于算法3的第3行执行了操作。为了使得算法能够适合于Spark环境，我们遵循Spark的规则生成RDD来更合理高效的利用内存和缓存。

同样，考虑到数据稀疏性问题，我们设计的RDD也采用了存储稀疏向量每个数据点的维度坐标和具体数值的方式。同样的，数据点的维度坐标将和数据点具体数值一样参与针对稀疏向量专门设计的运算。

而针对带惩罚项的罗吉斯回归问题，算法所需要的改动由于体现在原始更新的过程中。由于这一部分算法3与算法2一致，其改动方法也是完全相同的。

现在，我们可以着手分析罗吉斯回归模型优化问题并行算法的时间复杂度。在理想的并行状态下，原始更新的串行部分需要更新回归向量，这需要的时间来完成；并行部分可以看做在时间内完成。而在对偶更新过程中，串行部分含有一个随机抽样过程来抽取，需要的时间；并行部分更行概率向量，可以看做在时间内完成。所以，总的来说，每个迭代需要的时间，这正体现除了对训练数据集的次线性。

将此结果和算法1所代表的次线性串行算法相比较，可以发现两者相同，即上述的并行算法设计并没有降低算法复杂度。但是考虑常量的话，并行算法可以把2降为1，即在完全不考虑额外计算和通信开销的情况下，理论上速度能快一倍。更进一步，如果同前所述，将每次迭代过程中的两个MapReduce任务同时启动，那么理论上的算法复杂度可以降为。

Spark系统上运行的并行梯度下降法

我们设计了在Spark系统上运行的并行梯度下降法。具体的算法请看下面的算法4。此算法为经典算法，但与前文统一了描述风格。

算法4 并行-梯度下降-罗吉斯回归模型-Spark架构算法

|  |
| --- |
| Algorithm 4 PGDPLR-SPARK |
| 1: Input parameters: |
| 2: Initialize parameters: |
| 3: pointsspark.textFile(inputfile).map(parsePoint()).cache() |
| 4: Iterations: |
| 5: gradientpoints.map().reduce( \_+\_ ) |
| 6: |
| 7: |
| 8: Output: () |

算法4与算法2在基本并行框架上的设计非常相似。而它们最主要的区别在于算法3的第3行执行了操作。为了使得算法能够适合于Spark环境，我们遵循Spark的规则生成RDD来更合理高效的利用内存和缓存。