**Spark mllib-Logistic Regression With LBFGS**

1. **算法理论**
   1. **线性回归**

机器学习利用一些训练数据来分析未知数据，分为有监督学习(supervised learning)和无监督学习(unsupervised learning)。其中，有监督学习是指输入的训练数据有特征(feature)，有标签(label)，通过学习找到特征和标签间的关系(mapping)。

有监督学习又可以分为分类(classification)和回归(regression)，分类的标签是离散的，个数有限，回归的标签是连续的。

线性回归假设特征和结果都满足线性。收集的数据中，每一个分量，就可以看做一个特征数据。每个特征至少对应一个未知的参数。这样就形成了一个线性模型函数，向量表示形式：



求里面的参数，很可能无法直接求解，因为很有可能得到一个超定方程组（方程个数大于未知量个数）或欠定方程组（方程个数小于未知量个数）。

求解超定方程组通常用到最小二乘法或梯度下降法（最小化损失函数），将参数求解问题，转化为求最小误差问题，得到一个最接近的解，也就是最优化问题。

* 1. **广义线性模型**

将线性回归推广为广义线性模型（generalizedlinear model）。根据输出，也就是因变量的不同，广义的线性回归可以分为以下几种：

如果因变量是连续的，就是多重线性回归；

如果因变量是二项分布，就是Logistic回归；

如果因变量是Poisson分布，就是Poisson回归（泊松分布：描述某段时间内，事件具体的发生概率）；

如果因变量是负二项分布，就是负二项回归。

* 1. **逻辑回归**

假设事件符合伯努利分布（进行一次伯努利试验，成功( )概率为()，失败()概率为）。

则：





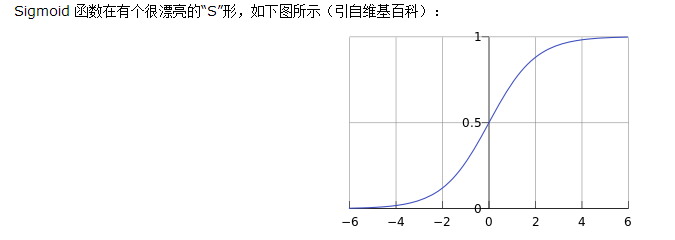
令，得到

因此逻辑回归采用Sigmoid函数：





如图：



大于0.5被分入1类，小于0.5被分入0类。

分类边界：时，y=1;时，y=0。

对于特征数集（到），对应的结果集（到），分类结果为类别1和类别0的概率分别为：





用极大似然估计来进行参数估计。

\*\*

* Logistic 回归模型的参数估计为什么不能采用最小二乘法？https://www.zhihu.com/question/23817253
* 最大似然估计和最小二乘法怎么理解？

http://www.zhihu.com/question/20447622

\*\*

其极大似然估计如下：



两边取对数，得到



最大似然估计就是求使取最大值时的。

多类的逻辑回归分类可以看成是二分类逻辑回归的推广。

* 1. **梯度下降**

对极大似然函数求导：



得到批量梯度下降法则：



* 1. **最优化方法**

有多种最优化方法求：梯度下降法、共轭梯度法、BFGS、L-BFGS（限制内存BFGS）等。

MLlib中实现了[梯度下降法](https://github.com/endymecy/spark-ml-source-analysis/blob/master/%E6%9C%80%E4%BC%98%E5%8C%96%E7%AE%97%E6%B3%95/%E6%A2%AF%E5%BA%A6%E4%B8%8B%E9%99%8D/gradient-descent.md)、[L-BFGS](https://github.com/endymecy/spark-ml-source-analysis/blob/master/%E6%9C%80%E4%BC%98%E5%8C%96%E7%AE%97%E6%B3%95/L-BFGS/lbfgs.md)，

L-BFGS较之梯度下降法有更快的收敛速度，其他特点包括：

1. 只保存近m次迭代信息；

2. 通过line search来确定学习率（而非手动选择）；

3. 求可行方向r，可以使用two-loop recursion算法来求。

1. **常见应用场景**

逻辑回归通常用于分类，也可以用于回归。

用作分类时可以是二分类也可以是多分类，通常为二分类。

1. **算法逻辑**
2. 输入：

算法的输入是labeled point（有监督的学习）

逻辑回归的输入数据是带label的稀疏向量(sparse vector, 一个稀疏向量通过两个并行的数组（indices和values）保存数据，如(3, [0, 2], [1.0, 3.0])，其中3表示数组的大小。)。 可以为libSVM格式:



通过MLUtils.loadLibSVMFile载入。

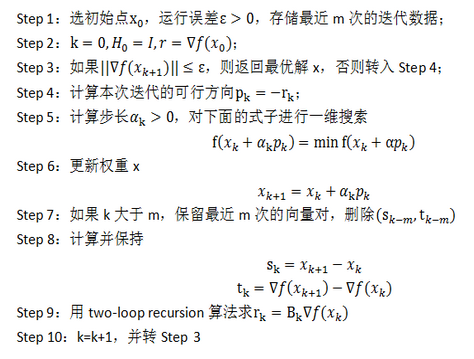
1. 输出：

通过MulticlassMetrics输出算法的准确率-召回率、F-Measure、F1-measure、precision等；

用作二分类的时候通过BinaryClassificationMetrics输出AUC、准确率-召回率、F-Measure。

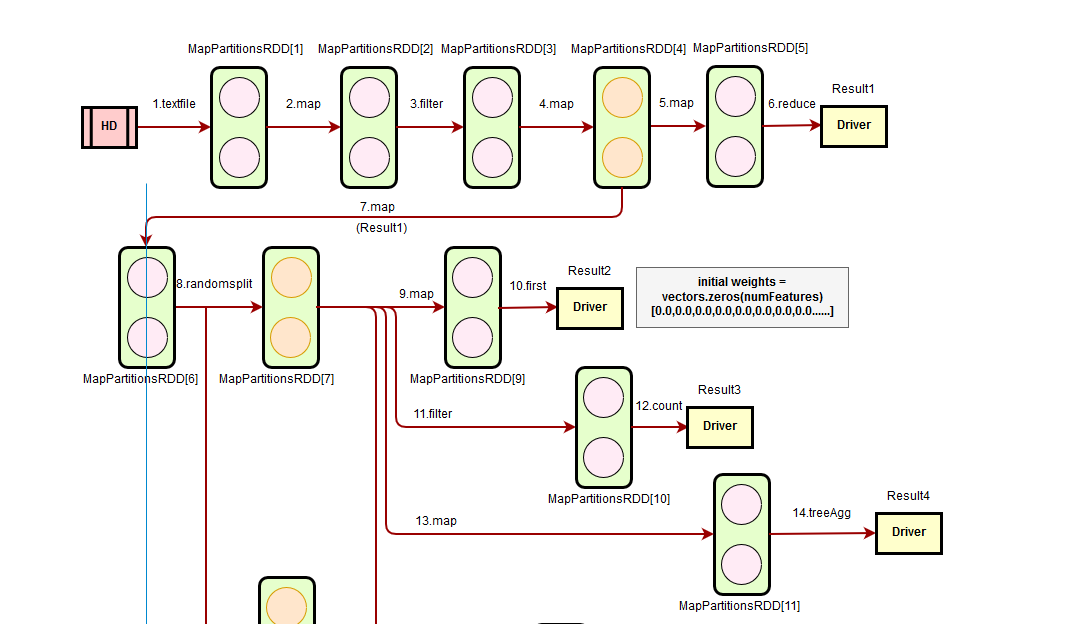
1. L-BFGS算法逻辑：
2. 初始化权重向量都为0；
3. 判断输入数据的label是否符合k分类；
4. 对每一个维度进行标准化/归一化；
5. 选择梯度下降方向
6. 计算步长
7. 更新权重
8. 计算梯度和损失值
9. 计算s和t，并更新history

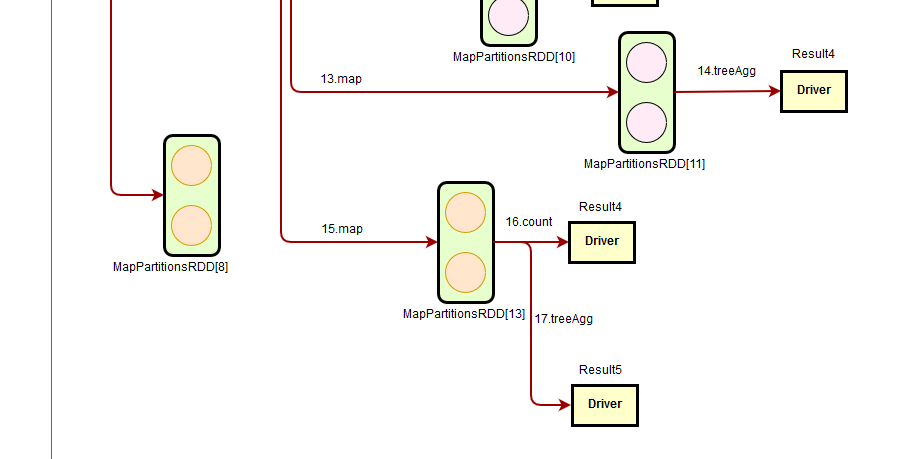
如图：



1. **算法分布式实现**

数据流图：





1. 读入数据
2. 去除首尾空格
3. 去除空行和注释掉的行
4. 转换格式
5. 获得每条数据的最大特征数
6. 计算features数
7. 转换成labeled point
8. 划分成训练数据集和测试数据集
9. 转换格式
10. 获取特征数量
11. 找出标签不在分类1-k中的LabeledPoint
12. 计算标签不在分类1-k中的LabeledPoint数
13. 得到features
14. 计算每一维特征的均值和标准差

**def** fit(data: RDD[Vector]): StandardScalerModel = {  
 // TODO: skip computation if both withMean and withStd are false  
 **val** summary = data.treeAggregate(**new** MultivariateOnlineSummarizer)(  
 (aggregator, data) => aggregator.add(data), //executor本地计算均值/方差等  
 (aggregator1, aggregator2) => aggregator1.merge(aggregator2)) //在driver上合并executors上的计算结果  
 **new** StandardScalerModel(  
 Vectors.*dense*(summary.variance.toArray.map(v => math.*sqrt*(v))), //得到每一维特征的标准差数组  
 summary.mean, //得到每一维特征的均值  
 withStd, //默认true：将数据缩放到单位标准偏差  
 withMean) //默认false：在缩放之前用平均值居中数据  
}

举例：计算平均值

**Add：**localCurrMean(index) = prevMean + weight \* diff / (localNnz(index) + weight)

**Merge：***currMean*(i) += deltaMean \* otherNnz / totalNnz

1. 标准化：将数据集的列的标准差归一化为1
2. 得到RDD中元素个数（即训练样本数）
3. 计算梯度和损失值

**val** costFun =  
 **new** CostFun(data, gradient, updater, regParam, numExamples)  
**val** lbfgs = **new** BreezeLBFGS[BDV[Double]](maxNumIterations, numCorrections, convergenceTol)

*/\*\*  
 \* CostFun implements Breeze's DiffFunction[T], which returns the loss and gradient  
 \* at a particular point (weights). It's used in Breeze's convex optimization routines.  
 \*/***private class** CostFun(  
 data: RDD[(Double, Vector)],  
 gradient: Gradient,  
 updater: Updater,  
 regParam: Double,  
 numExamples: Long) **extends** DiffFunction[BDV[Double]] {  
  
 **override def** calculate(weights: BDV[Double]): (Double, BDV[Double]) = {  
 // Have a local copy to avoid the serialization of CostFun object which is not serializable.  
 **val** w = Vectors.*fromBreeze*(weights)  
 **val** n = w.size  
 **val** bcW = data.context.broadcast(w) //广播权重  
 **val** localGradient = gradient //计算本地梯度  
  
 **val** (gradientSum, lossSum) = data.treeAggregate((Vectors.*zeros*(n), 0.0))(  
 seqOp = (c, v) => (c, v) **match** { **case** ((grad, loss), (label, features)) =>  
 **val** l = localGradient.compute(  
 features, label, bcW.value, grad)  
 (grad, loss + l)  
 },  
 combOp = (c1, c2) => (c1, c2) **match** { **case** ((grad1, loss1), (grad2, loss2)) =>  
 *axpy*(1.0, grad2, grad1)  
 (grad1, loss1 + loss2)  
 }) //计算梯度和损失值  
  
 */\*\*  
 \* regVal is sum of weight squares if it's L2 updater;  
 \* for other updater, the same logic is followed.  
 \*/* **val** regVal = updater.compute(w, Vectors.*zeros*(n), 0, 1, regParam).\_2  
 //L2更新权重  
 **val** loss = lossSum / numExamples + regVal */\*\*  
 \* It will return the gradient part of regularization using updater.  
 \*  
 \* Given the input parameters, the updater basically does the following,  
 \*  
 \* w' = w - thisIterStepSize \* (gradient + regGradient(w))  
 \* Note that regGradient is function of w  
 \*  
 \* If we set gradient = 0, thisIterStepSize = 1, then  
 \*  
 \* regGradient(w) = w - w'  
 \*  
 \* TODO: We need to clean it up by separating the logic of regularization out  
 \* from updater to regularizer.  
 \*/* // The following gradientTotal is actually the regularization part of gradient.  
 // Will add the gradientSum computed from the data with weights in the next step.  
 **val** gradientTotal = w.copy  
 *axpy*(-1.0, updater.compute(w, Vectors.*zeros*(n), 1, 1, regParam).\_1, gradientTotal)  
  
 // gradientTotal = gradientSum / numExamples + gradientTotal  
 *axpy*(1.0 / numExamples, gradientSum, gradientTotal)  
  
 (loss, gradientTotal.asBreeze.asInstanceOf[BDV[Double]])  
 }  
}

其中：localGradient.compute(LogisticGradient)对本地每个vector进行梯度运算:



**val** margin = -1.0 \* *dot*(data, weights)  
**val** multiplier = (1.0 / (1.0 + math.*exp*(margin))) - label  
*axpy*(multiplier, data, cumGradient) //计算每个样本的梯度  
**if** (label > 0) {  
 // The following is equivalent to log(1 + exp(margin)) but more numerically stable.  
 MLUtils.*log1pExp*(margin)  
} **else** {  
 MLUtils.*log1pExp*(margin) - margin  
}

SquaredL2Updater：对每个样本进行L2正则化，更新权重值。

为了防止过拟合，需要在Loss Function后面加入一个正则化项一起求最小值。正则化项相当于对Weights向量的惩罚，期望求出一个更简单的模型。 MLlib目前支持两种正则化方法L1和L2。 L2正则化假设模型参数服从高斯分布，L2正则化函数比L1更光滑，所以更容易计算；L1假设模型参数服从拉普拉斯分布，L1正则化具备产生稀疏解的功能，从而具备Feature Selection的能力。

lbfgs.iterations迭代计算梯度，更新权重。

注解treeAggregate：

先在每个分区并行执行seqOp，局部累加，同时根据tree的depth得到scale，具体由各个executor上的task来完成计算；

再串行执行combOp：结合不同partition上的结果，在JobWaiter的taskSucceeded函数中被调用。其中，partition用的HashPartitioner。

示例:

MapPartitionRDD[3]：[(0 1:51 4:159 5:253 8:159 9:50)......]

MapPartitionRDD[4]:[(0.0, [0, 3, 4, 7, 8], [51.0, 159.0, 253.0,159.0, 50.0])......]

MapPartitionRDD[5]:[9 ......]

Result1:10

MapPartitionRDD[6]:Labeled point[(0.0,(10[0, 3, 4, 7, 8], [51.0, 159.0, 253.0,159.0, 50.0]))......]

MapPartitionRDD[7]:Labeled point: training[(0.0,(10[0, 3, 4, 7, 8], [51.0, 159.0, 253.0,159.0, 50.0]))......]

MapPartitionRDD[8]:Labeled point: test

MapPartitionRDD[9]:[10.....]

Result2:10

Result3:0

MapPartitionRDD[11]:[(10[0, 3, 4, 7, 8], [51.0, 159.0, 253.0,159.0, 50.0])...]

MapPartitionRDD[13]:[(10[0, 3, 4, 7, 8], [0.88, 1.34, 1.93,1.25, 0.43])...]

Result4:10

Result5:10

**5. 测试**

输入数据格式：

*label index0:value0 index1:value1 ...*

在二分类中，其中*label*为0或1。

在k分类中，*label*为[0, k-1]。

数据生成：

不同维度，样本数，稀疏度，正负样例比，分布形式。

分布形式：通过RandomRDD生成

|  |  |
| --- | --- |
| 标准正态分布 | RandomRDDs.normalRDD |
| Gamma分布 | RandomRDDs.gammaRDD |
| 泊松分布 | RandomRDDs.poissonRDD |
| 指数分布 | RandomRDDs.exponentialRDD |
| 均匀分布 | RandomRDDs.uniformRDD |
| 混合分布 | 混合以上所有分布 |

配置：

Input split size：通过loadLibSVMFile的参数设置input split。

Partition number

Partition function：默认为Hash。

Buffer size

稳定性测试：

衡量标准：

运行结果 —— 模型参数(weights)：

相同的训练数据在相同的系统和应用参数配置下运行10次得到的模型参数weights一致。

扩展性测试：

Loss:

0.7605785320139856, 0.7605779829360662, 0.7605777939932867, 0.7605776225079214, 0.7605774814780468, 0.7605774257741045, 0.7605774210285483, 0.7605774205414916, 0.7605774193512876, 0.7605774186752784

运行时间

问题：

1 迭代无法收敛的问题。

2 欠拟合问题。

解释1：

如果没有做归一化处理，最大可能是迭代速度会比较慢，会出现在最大迭代次数内迭代无法收敛的问题；

如果归一化了还无法收敛，也有很多可能。

比如：

1. 用了已有的数据测试出现了line search failed的问题，找不到合适的学习率/步长；

2. 结果最终在一个区间内来回振荡而不收敛

解释2：

特征集过小

An Experimental Comparison of Iterative MapReduce Frameworks

1.数据集

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Real-World Data Sets | 数据集 | 大小 | 扩展 | 扩展方式 | 说明 |
| Cosmo-Gas | 1亿，3维 | 1到5倍 | 加入高斯噪声 | 用在K-means中 |
| Cosmo-All | 3亿，10维 | 取20%，40%，60%，80% | 随机取样 | 用在LR中 |
| 合成数据集 | 旨在生成倾斜数据，不用在LR中 | | | | |

1. 评估指标

|  |  |
| --- | --- |
| 指标 | 说明 |
| Total elapsed time（总运行时间） | 任务结束时间-开始时间（从log中得到），不包含任务开始前数据的复制或格式化（只进行一次） |
| Normalized time | 特定配置下的总运行时间:基础配置的运行时间，评价运行时间对某个参数的敏感度 |
| Total HDFS read | 从HDFS读的数据量 |
| Reduce shuffle time | 整个算法执行过程中从mapper到reducer的总shuffle数据量 |

1. 评估结果

|  |  |  |
| --- | --- | --- |
| 指标 | 结果 | 原因 |
| Normalized time | 数据集大小：非线性 | 数据集大小超过内存时，开始使用磁盘 |
| 迭代次数：线性 | 将数据从HDFS或本地磁盘读进来后，只要内存够用，就不再需要从磁盘中读 |
| Total HDFS read | 8200 |  |
| Reduce shuffle time | 0 | 没有shuffle过程 |

L0，L1，L2范数

L0范数：向量中非0元素的个数。期望向量矩阵是稀疏的。

L1范数：向量中各个元素的绝对值之和。Lasso

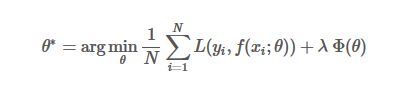
系数化向量矩阵：特征选择，可解释性

L2范数：向量中各个元素平方和开根号。岭回归

L1正则化和L2正则化

Lasso回归和岭回归（Ridge regression）

损失函数：



经验风险项+正则项

log对数损失函数（逻辑回归）；

平方损失函数（最小二乘法, Ordinary Least Squares ）；

指数损失函数（Adaboost）；

Hinge损失函数（SVM）；

测试：

1. Numclass = 2 的运行过程，二分类和多分类的不同处：
2. 都用costFun
3. 都用LogisticCostFun，不能，因为该算法不能实现多分类。
4. 对比costFun迭代次数和结果，lossfunction
5. 生成不同规模数据进行测试，设置executor-memory和executor-cores，对比迭代次数，运行时间，运行结果

### 测试

* **数据：**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Records | Dimensions | Input size | Input split size | Input split | Distribution |
| 99748 | 1000 | 910.4 MB | 113.74 MB(<128MB) | 8 | hybrid |

* **应用配置：**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| numIterations | regParam | convergenceTol | numCorrections | numClasses |
| 100 | 10 | 1.0 | 10E-6 | 2 |

* **系统配置：**

|  |  |  |
| --- | --- | --- |
| total-executor-cores | memory per node | number of cores per executor |
| 8 | 3GB | 2 |

* **运行结果：**

26次迭代，收敛。

Weights:[-0.0011064131369508405,0.0027046292431097763,2.1123630564586983E-5,0.0017374842255659037,... ...,-3.990738493105064E-4,-8.750113548532511E-4,0.0010607528026060598,0.002228669085367928,8.221243749031369E-4,-4.7917658529173646E-4,6.59350772785657E-4]

多次运行，结果相同。

*结论1：*LogisticRegression算法、LBFGS算法稳定性好。

最后7次迭代结果（Job26的结果没有打印出来）：

... ...

*Job21* Val and Grad Norm: 0.692356 (rel: 1.09e-08) 0.000491846

*Job22* Val and Grad Norm: 0.692356 (rel: 1.01e-08) 0.000354928

*Job23* Val and Grad Norm: 0.692356 (rel: 2.00e-08) 0.000277916

*Job24* Val and Grad Norm: 0.692356 (rel: 6.62e-09) 0.000167108

*Job25* Val and Grad Norm: 0.692356 (rel: 3.06e-09) 0.000154125

*Job27* Val and Grad Norm: 0.692356 (rel: 1.63e-09) 6.65409e-05

*Job28* Val and Grad Norm: 0.692356 (rel: 4.40e-10) 3.87919e-05

这里的val是指losses。

### 2. 数据流分析

***Job1*** (12 s)

reduce at MLUtils.scala:92

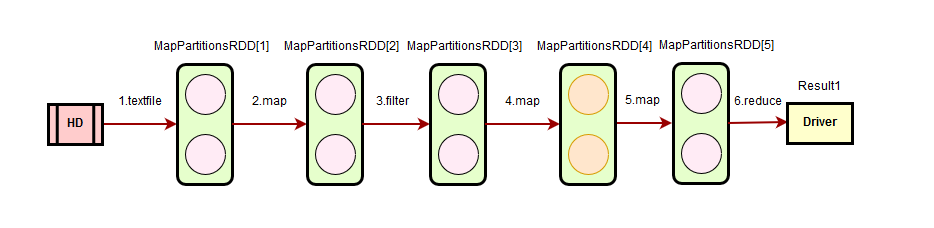
* 主要逻辑：

computeNumFeatures计算features最大数

* 数据流图：

8 tasks, 4 executors, 1 stage.

输入原始的文件，输出最大features数。其中，MapPartitionsRDD[4]被缓存。MapPartitionsRDD[4]：[(label: double, index:Array[int], value:Array[double])......]



* **问题1：**

Example中先通过loadLibSVMFile导入包括训练数据和测试数据在内的所有数据，计算最大features个数为训练数据和测试数据中的最大features，然后再划分为训练数据集和测试数据集。

如果训练数据集和测试数据集分别通过loadLibSVMFile，则训练时用的features数为训练数据集的最大features数目，测试数据集中的最大features数据大于训练数据集时就会出错。

**通过原有程序的API如何避免：**

通过loadLibSVMFile(sc, path, numFeatures)自己设定，需要自己确定数据中的最大features数。

**改进：**

理论上，训练数据集中没有的维度即模型没有该维度，那么测试数据集中就可以将这些维度默认为0，符合该模型。

***Job2*** (0.2s)

first at GeneralizedLinearAlgorithm.scala:204

* 主要逻辑：

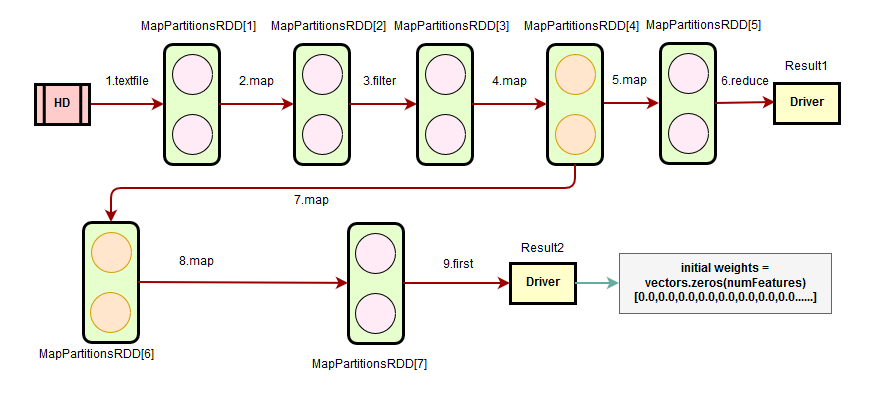
转换成labeled point（Input split size: 从113.74 MB变为73.4 MB，每个partition的records数量不变）并初始化权重向量（[0.0, 0.0, ..., 0.0]）。

* 步骤说明：

1. 转换成labeled point；
2. 每一个元素转换成features数, 输入为1个partition，73.4MB；
3. 获得features数

* 数据流图：

1 task, 1 executor, 1 stage.



* **问题2：**

data.map(\_.features.size).first()对其中一个partition进行map再取第一个元素，map操作对所有元素进行转换是否多余？

**测试对比运行时间：**

|  |  |  |
| --- | --- | --- |
|  | data.map(\_.features.size).first() | data.first().features.size |
| 100d，107records，10cores | 0.1 s | 97 ms |
| 10d，107records，4cores | 0.2 s | 0.2s |

结论：

运行时间与cores数目无关，因为每个partition大小固定，只能改变一个partition中的records数，records数越多，运行时间越长。

由于first前面的transformation操作不止map，还包括了之前的，手动加上shuffle或者action算子分开。

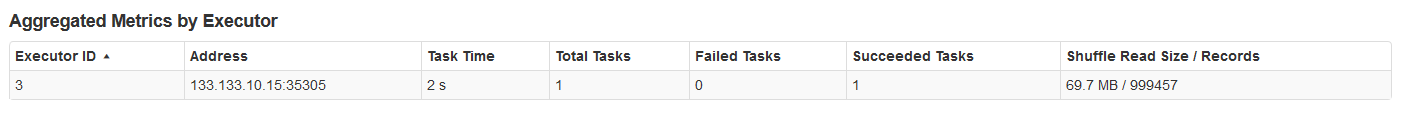
尝试：

--------------------------------------------------------------------------------------------------------------------------

val data = MLUtils.loadLibSVMFile(sc, path).coalesce(10, true).cache()  
 val features2 = data.first().features.size

--------------------------------------------------------------------------------------------------------------------------

结果：first这个stage的时间反而更长了，为什么？



疑问：即使map只要操作一个partition中的records，first操作只需要第一条record，是否需要改进？从测试来看，不需要改进，但是理论上呢？

***Job3*** (1s)

treeAggregate at LogisticRegression.scala:287

* 主要逻辑：

RDD转换成DataFrame，计算每一列的均值、方差等。

* 步骤说明：

1. 转换成asML格式 ；
2. createDataFrame 将RDD转换成DataSet [label:double, features: vector] ；

因为MapPartitionsRDD[6]缓存了，不再缓存DataSet.

12.13.14.15.16 DataSet转换成RDD [(label: Double, weight: Double, features: Vector)] 这里的weight指每一个样本的权重，都为1.0；

* **问题3：**

这里判断MapPartitionsRDD[6]已经缓存，不再缓存mappartitionsRDD[14]，但是MapPartitionsRDD[14]多次迭代使用，是否需要缓存？

1. treeAggregate 计算本地每一列的均值、方差等，再merge到一起，得到每一列的均值、方差等。

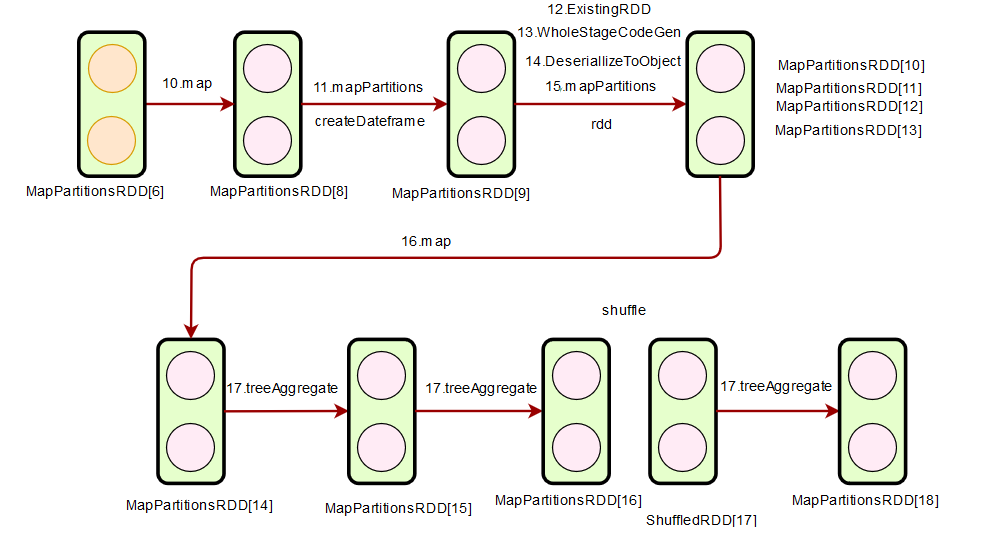
* 数据流图：

2 stages.

Stage1: 8 tasks, 4 executors, shuffle write 381.4 KB, each task 47.7 KB / 1,

Stage2: 2 tasks, 2 executors, shuffle read 381.4 KB, each task 190.7 KB / 4.

输入为labeled point，73.4 MB Input size / 12464 records。



***Job4~29*** (0.2s~0.5s)

treeAggregate at LogisticRegression.scala:1103

* 主要逻辑：

计算costFun，并用breezeLBFGS.iterations迭代

--------------------------------------------------------------------------------------------------------------------------

val costFun = new LogisticCostFun(instances, numClasses, $(fitIntercept),

$(standardization), featuresStd, featuresMean, regParamL2)

--------------------------------------------------------------------------------------------------------------------------

* 步骤说明：

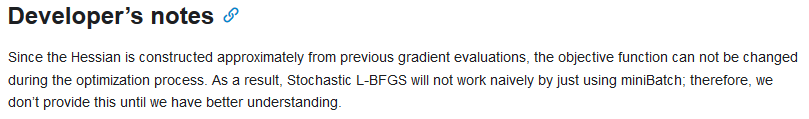
1. 用LogisticAggregator计算本地gradient和loss，只用于二分类，再将结果合并。

*--------------------------------------------------------------------------------------------------------------------------*

*TODO:*

对比LogisticAggregator中对gradient和loss的计算与LogisticGradient中的异同，以及根据LogisticGradient中对多分类计算梯度的实现来实现LogisticAggregator中的梯度计算，从而对比LogisticGradient和LogisticAggregator中的对二分类的Gradient、loss迭代结果的影响。

两者都是用梯度下降法，是否可以借鉴批随机梯度下降法runMiniBatchSGD？



--------------------------------------------------------------------------------------------------------------------------

* 数据流图：

2 stages.

Stage1: 8 tasks, 4 executors, shuffle write 65.5 KB, each task 8.2 KB / 1,

Stage2: 2 tasks, 2 executors, shuffle read 65.5 KB, each task 32.7 KB / 4.

输入仍然为labeled point, 73.4 MB Input size / 12464 records

