# Spark最优化

# 重新定个框架

1. GeneralizedLinearAlgorithm
2. LogisticGradient
3. SquaredL2Updater
4. LogisticRegressionWithSGD
5. LBFGS
6. LogisticGradient和LogisticGradient
7. LogisticRegressionWithLBFGS

## 引言

## 最优化方法及其原理

BFGS、SGD

## SGD

### SGD怎样用？

**class** LogisticRegressionWithSGD **private**[mllib] (  
 **private var** stepSize: Double,  
 **private var** numIterations: Int,  
 **private var** regParam: Double,  
 **private var** miniBatchFraction: Double)  
 **extends** GeneralizedLinearAlgorithm[LogisticRegressionModel] **with** Serializable {

// 定义梯度函数  
 **private val** *gradient* = **new** LogisticGradient()

// 定义更新函数  
 **private val** *updater* = **new** SquaredL2Updater()

// 最优化对象  
 **override val** *optimizer* = **new** GradientDescent(*gradient*, *updater*)  
 .setStepSize(stepSize)  
 .setNumIterations(numIterations)  
 .setRegParam(regParam)  
 .setMiniBatchFraction(miniBatchFraction)

真正的调用在其继承自GeneralizedLinearAlgorithm的run方法中。用来迭代求解最优系数和截距项。

**abstract class** GeneralizedLinearAlgorithm[M <: GeneralizedLinearModel]  
 **extends** Logging **with** Serializable {

…

**def** run(input: RDD[LabeledPoint], initialWeights: Vector): M = {

…

**val** weightsWithIntercept = optimizer.optimize(data, initialWeightsWithIntercept)

…

## BFGS

### BFGS怎样用？

**class** LogisticRegressionWithLBFGS  
 **extends** GeneralizedLinearAlgorithm[LogisticRegressionModel] **with** Serializable {  
  
 **this**.setFeatureScaling(**true**)  
// 最优化，其中gradient为logistic中的gradient，//updater为平方误差函数的updater  
 **override val** *optimizer* = **new** LBFGS(**new** LogisticGradient, **new** SquaredL2Updater)

其调用过程要多一层，最初的ogisticRegression调用的是LogisticRegressionWithLBFGS中的run方法，而run方法同样源自GeneralizedLinearAlgorithm。

LogisticRegression  
 **extends** ProbabilisticClassifier[Vector, LogisticRegression, LogisticRegressionModel]  
 **with** LogisticRegressionParams {

。。。

**val** lr = **new** LogisticRegressionWithLBFGS  
lr.*optimizer* .setRegParam(paramMap(*regParam*))  
 .setNumIterations(paramMap(*maxIter*))  
**val** oldModel = lr.run(oldDataset)

两个类SGD和LBFGS都继承自optimizer

**trait** Optimizer **extends** Serializable {  
  
 */\*\*  
 \* Solve the provided convex optimization problem.  
 \*/* **def** optimize(data: RDD[(Double, Vector)], initialWeights: Vector): Vector  
}

里面几乎没有什么代码，主要是一个壳子，用于定义参数和一个最优化函数用于解决凸优化问题。

## 第一次小结

我们发现：

1. SGD的最优化的实现依据类GradientDescent，LBFGS的实现依据LBFGS。而两者都继承了Optimizer。两者需要gradient: Gradient, updater: Updater参数，一个用于梯度下降一个用于权值更新。
2. 两者的最终应用实在GeneralizedLinearAlgorithm中的run方法中的最优化步骤实现的。

下面我们就研究一下optimizer特质及其两个子类应用，并研究一下optimizer需要的数据类Gradient和Updater。

## LBFGS

我们发现其主要调用最优化方法的函数如下：

**def** runLBFGS(  
 data: RDD[(Double, Vector)],  
 gradient: Gradient,  
 updater: Updater,  
 numCorrections: Int,  
 convergenceTol: Double,  
 maxNumIterations: Int,  
 regParam: Double,  
 initialWeights: Vector): (Vector, Array[Double]) = {  
  
 **val** lossHistory = **new** ArrayBuffer[Double](maxNumIterations)  
  
 **val** numExamples = data.count()  
  
 **val** costFun =  
 **new** CostFun(data, gradient, updater, regParam, numExamples)  
  
 **val** lbfgs = **new** BreezeLBFGS[BDV[Double]](maxNumIterations, numCorrections, convergenceTol)  
  
 **val** states =  
 lbfgs.iterations(**new** CachedDiffFunction(costFun), initialWeights.toBreeze.toDenseVector)  
  
 */\*\*  
 \* NOTE: lossSum and loss is computed using the weights from the previous iteration  
 \* and regVal is the regularization value computed in the previous iteration as well.  
 \*/* **var** state = states.next()  
 **while**(states.hasNext) {  
 lossHistory.append(state.value)  
 state = states.next()  
 }  
 lossHistory.append(state.value)  
 **val** weights = Vectors.*fromBreeze*(state.x)  
  
 logInfo("LBFGS.runLBFGS finished. Last 10 losses %s".format(  
 lossHistory.takeRight(10).mkString(", ")))  
  
 (weights, lossHistory.toArray)  
}  
  
*/\*\*  
 \* 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)  
 **val** localGradient = gradient

// 计算梯度和损失  
 **val** (gradientSum, lossSum) = data.treeAggregate((Vectors.*zeros*(n), 0.0))(  
 seqOp = (c, v) => (c, v) **match** { **case** ((grad, loss), (label, features)) =>

// 对每个数据计算梯度和损失，注意grad在内部赋值，这个在之后的gradient解析中进行分析。最终算出了损失函数  
 **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)  
 })  
  
 */\*\*  
 \* 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  
  
 **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.toBreeze.asInstanceOf[BDV[Double]])  
 }  
}

*l(w, x) = -log P(y|x, w) = -\alpha(y) log P(y=0|x, w) - (1-\alpha(y)) log P(y|x, w)  
 = log(1 + \sum\_i^{K-1}\exp(x w\_i)) - (1-\alpha(y)) x w\_{y-1}  
 = log(1 + \sum\_i^{K-1}\exp(margins\_i)) - (1-\alpha(y)) margins\_{y-1}*

## SGD

在类class LogisticGradient(numClasses: Int) extends Gradient 中实现

numClasses为分类数  
**class** LogisticGradient(numClasses: Int) **extends** Gradient {  
  
 **def this**() = **this**(2)  
// 外层的compute函数  
 **override def** compute(data: Vector, label: Double, weights: Vector): (Vector, Double) = {

// 首先定义一个初始的累积梯度向量（初始值全为0），进行真正的cpmpute函数，计算累积梯度和loss  
 **val** gradient = Vectors.*zeros*(weights.size)  
 **val** loss = compute(data, label, weights, gradient)  
 (gradient, loss)  
 }

// 1.要介绍一下：

Weight对于K分类来说是*(K-1) \* (N+1)其中K是分类数N为data的维度数。*

*也就是：这也是为什么先判断一下维度是否符合的原因。*

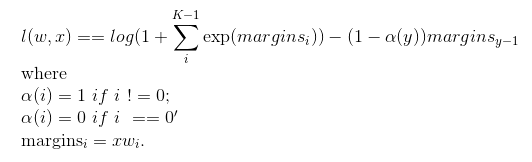
**override def** compute(  
 data: Vector,  
 label: Double,  
 weights: Vector,  
 cumGradient: Vector): Double = {

// 判断一下维度是否符合

**val** dataSize = data.size  
  
 // (weights.size / dataSize + 1) is number of classes  
 *require*(weights.size % dataSize == 0 && numClasses == weights.size / dataSize + 1)  
 numClasses **match** {  
 **case** 2 =>  
 */\*\*  
 \* For Binary Logistic Regression.  
 \*  
 \* Although the loss and gradient calculation for multinomial one is more generalized,  
 \* and multinomial one can also be used in binary case, we still implement a specialized  
 \* binary version for performance reason.  
 \*/*

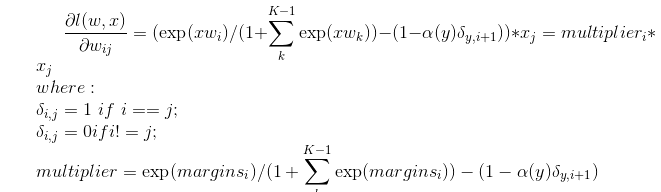
*// 2.累积函数的更新：*

*1）loss函数*

**

*2）梯度*

*一介导数*

**

*// case 2时，式子比较简单，直接计算出multiplier的值*

**val** margin = -1.0 \* *dot*(data, weights)  
 **val** multiplier = (1.0 / (1.0 + math.*exp*(margin))) – label

// 梯度 即fraction

*axpy*(multiplier, data, cumGradient)

// 损失函数

**if** (label > 0) {  
 // The following is equivalent to log(1 + exp(margin)) but more numerically stable.  
 MLUtils.*log1pExp*(margin)  
 } **else** {  
 MLUtils.*log1pExp*(margin) - margin  
 }  
 **case** \_ =>  
 */\*\*  
 \* For Multinomial Logistic Regression.  
 \*/* **val** weightsArray = weights **match** {  
 **case** dv: DenseVector => dv.values  
 **case** \_ =>  
 **throw new** IllegalArgumentException(  
 s"weights only supports dense vector but got type **$**{weights.getClass}.")  
 }  
 **val** cumGradientArray = cumGradient **match** {  
 **case** dv: DenseVector => dv.values  
 **case** \_ =>  
 **throw new** IllegalArgumentException(  
 s"cumGradient only supports dense vector but got type **$**{cumGradient.getClass}.")  
 }  
  
 // marginY is margins(label - 1) in the formula.  
 **var** marginY = 0.0  
 **var** maxMargin = Double.*NegativeInfinity* **var** maxMarginIndex = 0  
  
 **val** margins = Array.*tabulate*(numClasses - 1) { i =>  
 **var** margin = 0.0  
 data.foreachActive { (index, value) =>  
 **if** (value != 0.0) margin += value \* weightsArray((i \* dataSize) + index)  
 }  
 **if** (i == label.toInt - 1) marginY = margin  
 **if** (margin > maxMargin) {  
 maxMargin = margin  
 maxMarginIndex = i  
 }  
 margin  
 }  
  
 */\*\*  
 \* When maxMargin > 0, the original formula will cause overflow as we discuss  
 \* in the previous comment.  
 \* We address this by subtracting maxMargin from all the margins, so it's guaranteed  
 \* that all of the new margins will be smaller than zero to prevent arithmetic overflow.  
 \*/* **val** sum = {  
 **var** temp = 0.0  
 **if** (maxMargin > 0) {  
 **for** (i <- 0 until numClasses - 1) {  
 margins(i) -= maxMargin  
 **if** (i == maxMarginIndex) {  
 temp += math.*exp*(-maxMargin)  
 } **else** {  
 temp += math.*exp*(margins(i))  
 }  
 }  
 } **else** {  
 **for** (i <- 0 until numClasses - 1) {  
 temp += math.*exp*(margins(i))  
 }  
 }  
 temp  
 }  
  
 **for** (i <- 0 until numClasses - 1) {  
 **val** multiplier = math.*exp*(margins(i)) / (sum + 1.0) - {  
 **if** (label != 0.0 && label == i + 1) 1.0 **else** 0.0  
 }  
 data.foreachActive { (index, value) =>  
 **if** (value != 0.0) cumGradientArray(i \* dataSize + index) += multiplier \* value  
 }  
 }  
  
 **val** loss = **if** (label > 0.0) math.*log1p*(sum) - marginY **else** math.*log1p*(sum)  
  
 **if** (maxMargin > 0) {  
 loss + maxMargin  
 } **else** {  
 loss  
 }  
 }  
 }  
}