# spark中logistc及最优化源码学习

## 引言

Logistic回归

最优化方法：BFGS和SGD

## Logistic回归

## BFGS

## GD和SGD

## Spark中SGD

class GradientDescent private[mllib] (private var gradient: Gradient, private var updater: Updater) extends Optimizer with Logging

// 梯度函数，用于随机梯度下降时的更新

Set the gradient function (of the loss function of one single data example)

\* to be used for SGD.

// 正则化系数

/\*\*

\* Set the regularization parameter. Default 0.0.

\*/

def setRegParam(regParam: Double): this.type = {

this.regParam = regParam

this

}

// 更新方法

*/\*\*  
 \* Set the updater function to actually perform a gradient step in a given direction.  
 \* The updater is responsible to perform the update from the regularization term as well,  
 \* and therefore determines what kind or regularization is used, if any.  
 \*/***def** setUpdater(updater: Updater): **this**.**type** = {  
 **this**.updater = updater  
 **this**}

// 编外

toBreeze是什么方法

// 最优化方法

**def** optimize(data: RDD[(Double, Vector)], initialWeights: Vector): Vector = {  
 **val** (weights, \_) = GradientDescent.*runMiniBatchSGD*(  
 data,  
 gradient,  
 updater,  
 *stepSize*,  
 *numIterations*,  
 *regParam*,  
 *miniBatchFraction*,  
 initialWeights)  
 weights  
}

// 核心的算法

**def** runMiniBatchSGD(  
 data: RDD[(Double, Vector)],  
 gradient: Gradient,  
 updater: Updater,  
 stepSize: Double,  
 numIterations: Int,  
 regParam: Double,  
 miniBatchFraction: Double,  
 initialWeights: Vector): (Vector, Array[Double]) = {  
  
 **val** stochasticLossHistory = **new** ArrayBuffer[Double](numIterations)  
  
 **val** numExamples = data.count()  
  
 // if no data, return initial weights to avoid NaNs  
 **if** (numExamples == 0) {  
 logWarning("GradientDescent.runMiniBatchSGD returning initial weights, no data found")  
 **return** (initialWeights, stochasticLossHistory.toArray)  
 }  
  
 **if** (numExamples \* miniBatchFraction < 1) {  
 logWarning("The miniBatchFraction is too small")  
 }  
  
 // Initialize weights as a column vector  
 **var** weights = Vectors.*dense*(initialWeights.toArray)  
 **val** n = weights.size  
  
 */\*\*  
 \* For the first iteration, the regVal will be initialized as sum of weight squares  
 \* if it's L2 updater; for L1 updater, the same logic is followed.  
 \*/* **var** regVal = updater.compute(  
 weights, Vectors.*dense*(**new** Array[Double](weights.size)), 0, 1, regParam).\_2  
  
 **for** (i <- 1 to numIterations) {  
 **val** bcWeights = data.context.broadcast(weights)  
 // Sample a subset (fraction miniBatchFraction) of the total data  
 // compute and sum up the subgradients on this subset (this is one map-reduce)  
 **val** (gradientSum, lossSum, miniBatchSize) = data.sample(**false**, miniBatchFraction, 42 + i)  
 .treeAggregate((BDV.*zeros*[Double](n), 0.0, 0L))(  
 seqOp = (c, v) => {  
 // c: (grad, loss, count), v: (label, features)  
 **val** l = gradient.compute(v.\_2, v.\_1, bcWeights.value, Vectors.*fromBreeze*(c.\_1))  
 (c.\_1, c.\_2 + l, c.\_3 + 1)  
 },  
 combOp = (c1, c2) => {  
 // c: (grad, loss, count)  
 (c1.\_1 += c2.\_1, c1.\_2 + c2.\_2, c1.\_3 + c2.\_3)  
 })  
  
 **if** (miniBatchSize > 0) {  
 */\*\*  
 \* NOTE(Xinghao): lossSum is computed using the weights from the previous iteration  
 \* and regVal is the regularization value computed in the previous iteration as well.  
 \*/* stochasticLossHistory.append(lossSum / miniBatchSize + regVal)  
 **val** update = updater.compute(  
 weights, Vectors.*fromBreeze*(gradientSum / miniBatchSize.toDouble), stepSize, i, regParam)

// 更新weight和regVal  
 weights = update.\_1  
 regVal = update.\_2  
 } **else** {  
 logWarning(s"Iteration (**$**i/**$**numIterations). The size of sampled batch is zero")  
 }  
 }  
  
 logInfo("GradientDescent.runMiniBatchSGD finished. Last 10 stochastic losses %s".format(  
 stochasticLossHistory.takeRight(10).mkString(", ")))  
  
 (weights, stochasticLossHistory.toArray)  
  
}