# spark中logistc及最优化源码学习

## 引言

GMM的框架

Logistic回归

最优化方法：LBFGS和SGD

## GMM框架

两个重要的元素

1. Optimizer

trait Optimizer extends Serializable {

def optimize(data: RDD[(Double, Vector)], initialWeights: Vector): Vector

}

1. GeneralizedLinearAlgorithm

定义了一个最优化函数和run函数

GeneralizedLinearAlgorithm[M <: GeneralizedLinearModel]

extends Logging with Serializable {

def optimizer: Optimizer最优化函数，一个供子类覆盖的方法

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

… … …

val weightsWithIntercept = optimizer.optimize(data, initialWeightsWithIntercept)

… … …

createModel(weights, intercept)

}

}

1. 基于GMM模型写一个Optimizer

如LogisticWithSGD

A.写一个Gradient重新定义compute函数以计算梯度

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

就是求f’（data\*weights）

}

B.写一个Updater重新定义一个compute函数

def compute(

weightsOld: Vector,

gradient: Vector,

stepSize: Double,

iter: Int,

regParam: Double): (Vector, Double)

}{

W新 = weightsOld + …\* gradient

}

定义了一个通过Vector迭代求出梯度和损失函数的方法供覆盖

def compute(data: Vector, label: Double, weights: Vector, cumGradient: Vector): Double

C.最终通过梯度和更新写了一个Optimizer并定义了其中的optimize函数

GradientDescent

new GradientDescent(gradient, updater)

定义了一个随机梯度下降算法runMiniBatchSGD和一个最优化算法

def runMiniBatchSGD(

data: RDD[(Double, Vector)],

gradient: Gradient,

updater: Updater,

stepSize: Double,

numIterations: Int,

regParam: Double,

miniBatchFraction: Double,

initialWeights: Vector,

convergenceTol: Double): (Vector, Array[Double]) = {

… … …

var regVal = updater.compute(

weights, Vectors.zeros(weights.size), 0, 1, regParam).\_2 //

var converged = false // indicates whether converged based on convergenceTol

var i = 1

while (!converged && i <= numIterations) {

… … …

val (gradientSum, lossSum, miniBatchSize) = data.sample(false, miniBatchFraction, 42 + i) // 抽样

.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)) // 计算梯度

(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)

})

val update = updater.compute(

weights, Vectors.fromBreeze(gradientSum / miniBatchSize.toDouble),

stepSize, i, regParam) // 更新权重

… … …

(weights, stochasticLossHistory.toArray)

}

def optimize(data: RDD[(Double, Vector)], initialWeights: Vector): Vector = {

val (weights, \_) = GradientDescent.runMiniBatchSGD

## LogisticWithSGD回归

### 代码是如何组织和实现的

LogisticWithSGD是如何从零实现的，按照刚才的步骤：

一、 写一个Optimizer

A.写一个Gradient重新定义compute函数以计算梯度

这里写了LogisticGradient

B.写一个Updater重新定义一个compute函数

这里写了SquaredL2Updater

C.最终通过梯度和更新写了一个Optimizer——GradientDescent

定义了一个随机梯度下降算法runMiniBatchSGD和并基于它实现了optimize方法

def runMiniBatchSGD(

data: RDD[(Double, Vector)],

gradient: Gradient,

updater: Updater,

stepSize: Double,

numIterations: Int,

regParam: Double,

miniBatchFraction: Double,

initialWeights: Vector,

convergenceTol: Double): (Vector, Array[Double]) = {

… … …

var regVal = updater.compute(

weights, Vectors.zeros(weights.size), 0, 1, regParam).\_2 //

var converged = false // indicates whether converged based on convergenceTol

var i = 1

while (!converged && i <= numIterations) {

… … …

val (gradientSum, lossSum, miniBatchSize) = data.sample(false, miniBatchFraction, 42 + i) // 抽样

.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)) // 计算梯度

(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)

})

val update = updater.compute(

weights, Vectors.fromBreeze(gradientSum / miniBatchSize.toDouble),

stepSize, i, regParam) // 更新权重

… … …

(weights, stochasticLossHistory.toArray)

}

def optimize(data: RDD[(Double, Vector)], initialWeights: Vector): Vector = {

val (weights, \_) = GradientDescent.runMiniBatchSGD

}

1. 写一个createModel

A．创建了LogisticRegressionModel类型继承自GeneralizedLinearModel

B．写了createModel方法

override protected[mllib] def createModel(weights: Vector, intercept: Double) = {

new LogisticRegressionModel(weights, intercept)

}

### 注意的事项

## LogisticRegressionWithLBFGS回归

### 代码是如何组织和实现的

1. 写一个optimizer

写了一个LBFGS()

override val optimizer = new LBFGS(new LogisticGradient, new SquaredL2Updater)

1. 写createModel同样输出的是LogisticRegressionModel类型

override protected def createModel(weights: Vector, intercept: Double) = {

if (numOfLinearPredictor == 1) {

new LogisticRegressionModel(weights, intercept)

} else {

new LogisticRegressionModel(weights, intercept, numFeatures, numOfLinearPredictor + 1)

}

}

### 注意事项

## GD和SGD

是基于梯度下降法求解logit回归的系数

需要梯度gradient和迭代更新式updater

new GradientDescent(gradient, updater)

## LBFGS

是基于拟牛顿法LBFGS求解logit回归的系数。全称是Limited-memory BFGS。Spark调用的是scala.breeze.optimize中的LBFGS。

需要梯度gradient和迭代更新式updater

new LBFGS(new LogisticGradient, new SquaredL2Updater)

显然LBFGS的更新不能直接依靠SquaredL2Updater，真正的更新方式在LBFGS的CostFun中封装着。

override def optimize(data: RDD[(Double, Vector)], initialWeights: Vector): Vector = {

val (weights, \_) = LBFGS.runLBFGS(

data,

gradient,

updater,

numCorrections,

convergenceTol,

maxNumIterations,

regParam,

initialWeights)

weights

}

最终实现依靠runLBFGS