Распределенные алгоритмы. Spark MLlib. Стохастический градиентный спуск.

Содержание:

- Стохастический градиентный спуск
- Алгоритм распределенного стохастического градиентного спуска в Spark MLlib
- Реализация алгоритмов в Spark MLlib
- Список литературы

Стохастический градиентный спуск

Функция потерь (целевая функция) на множестве данных D (тренировочная выборка размера n):

$$L(y, x, \theta) = \sum_{(x,y)\in D} l(y, x, \theta),$$

где x – вектор признаков; y – целевое значение; θ – неизвестные значения параметров; для линейной регрессии:

$$l(y, x, \theta) = (y - x^T\theta)^2.$$

В стохастическом градиентном спуске подстройка параметров осуществляется на основе одного экземпляра данных:

$$\theta \leftarrow \theta - \eta \nabla l(y, x, \theta),$$

где η – скорость обучения;

$$\nabla l(y, x, \theta) = (y - x^T\theta)x.$$

Весь процесс оптимизации целевой функции можно представить в виде итеративного алгоритма 1.

Алгоритм 1. SGD (без проверки на сходимость)

1	$\text{initialize}(\theta, \eta)$	Инициализация
2	$\text{for } i \text{ in } 1..T$:	Количество эпох
3	$\quad\quad\text{shuffle}(D)$	Перетасовка
4	$\quad\quad\text{for } (x, y) \text{ in } D$:	
5	$\quad\quad\quad\quad\theta \leftarrow \theta - \eta \nabla l(y, x, \theta)$	
6	$\text{return } \theta$	

Мини-пакеты (mini-batch):

$$\theta \leftarrow \theta - \eta \cdot \sum_{(x,y)\in \mathcal{B}} \nabla l(y, x, \theta),$$

где $\mathcal{B}$ - случайная выборка из множества тренировочных данных некоторого фиксированного размера m, как правило, $m \ll n$

Если стохастический градиентный спуск используется для задачи регрессии, то значение предсказания ($\hat{y}$) для некоторого входного вектора признаков (x) вычисляется как

$$\hat{y} = x^T \hat{\theta},$$

где $\hat{\theta}$ – оценка параметров модели посредством стохастического градиентного спуска.

Алгоритм распределенного стохастического градиентного спуска в Spark MLlib

В общем виде при распределенном стохастическом градиентном спуске данные D разбиваются на P частей, каждая из которых обрабатывается отдельно. В этом случае подстройка параметров на основное мини-пакетов может быть получена следующим образом:

$$\theta \leftarrow \theta - \eta \cdot \sum_{p=1}^{P} \sum_{(x,y)\in\mathcal{B}^p} \nabla l(y, x, \theta),$$

где $\mathcal{B}^p$ – случайная выборка в части p.

Данный подход реализован в Spark MLlib, краткий алгоритм которого приведен ниже.

Алгоритм 2. SGD: Spark mini-batch

1	initialize(θ, η)	
2	for i in $1..T$:	Количество эпох
3	broadcast(θ)	Параметры рассылаются на все узлы
4	for p in $1..P$ in parallel:	
5	$\mathcal{B}^p \leftarrow$ sample$(D^p, batch_{size})$	Случайная выборка из D^p размером $batch_{size}$
6	$\text{grad}_p \leftarrow 0$	
7	for (x, y) in $\mathcal{B}^p$:	
8	$\text{grad}^p \leftarrow \text{grad}^p + \nabla l(y, x, \theta)$	
9	collect$(\text{grad}^1, .., \text{grad}^P)$	Градиенты от P частей собираются на мастере
10	$\theta \leftarrow \theta - \eta \cdot \sum_{p=1}^{P} \text{grad}^p$	
11	return θ	

Более детальный вариант представлен далее в алгоритмах 3-6.

Алгоритм 3. Spark RDD API. SGD. Gradient Descent

1	load$\left(D, step, iter_{max}, batch_{fraction}, convTol\right)$	
2	$\theta \leftarrow \mathbf{0}; i \leftarrow 1; converged \leftarrow$ false	
3	while (not $converged$) and $(i \leq iter_{max})$	
4	broadcast(θ)	
5	$\mathcal{B} \leftarrow$ sampleEachPartition$\left(D, batch_{fraction}\right)$	RDD
6	for p in $1..P$ in parallel:	Для каждой части $\mathcal{B}$
7	$\text{grad}^p \leftarrow \mathbf{0}; loss^p \leftarrow 0; count^p \leftarrow 0$	
8	for (x, y) in $\mathcal{B}^p$:	
9	$(\text{grad}, loss) \leftarrow$ **gradient**$(x, y, \theta, \text{grad}^p)$	
10	$\text{grad}^p \leftarrow \text{grad}^p + \text{grad}$	
11	$loss^p \leftarrow loss^p + loss$	

12	$count^p \leftarrow count^p + 1$	
13	collect$\left((\text{grad}^1, loss^1, count^1), .., (\text{grad}^P, loss^P, count^P)\right)$	
14	for p in $1..P$:	
15	grad $\leftarrow$ grad + grad^p	
16	$loss \leftarrow loss + loss^p$	
17	$batch_{size} \leftarrow batch_{size} + count^p$	
18	$(\theta, 0) \leftarrow$ **updater**$(\theta, \text{grad}/batch_{size}, step, i)$	
19	$\theta^{prev} \leftarrow \theta^{current}$	
20	$\theta^{current} \leftarrow \theta$	
21	$converged \leftarrow$ **isConverged**$(\theta^{prev}, \theta^{current}, convTol)$	
22	$i \leftarrow i + 1$	
23	return(θ)	

Алгоритм 4. Spark RDD API. SGD. Gradient

1	load(x, y, w, grad)
2	$diff \leftarrow x^T w - y$
3	grad $\leftarrow$ grad $+ diff \cdot x$
4	$loss \leftarrow diff \cdot diff/2$
5	return$(\text{grad}, loss)$

Алгоритм 5. Spark RDD API. SGD. Updater

1	load$(w, \text{grad}, step, i)$
2	$\eta \leftarrow step/\sqrt{i}$
3	$w \leftarrow w - \eta \cdot \text{grad}$
4	return$(w, 0)$

Алгоритм 6. Spark RDD API. SGD. Проверка на сходимость

1	load$(\theta^{prev}, \theta^{current}, convTol)$
2	$vecDiff \leftarrow \|\theta^{prev} - \theta^{current}\|$
3	if $vecDiff < convTol \cdot \max(\|\theta^{current}\|, 1)$:
4	return(true)
5	return(false)

Реализация алгоритмов в Spark MLlib

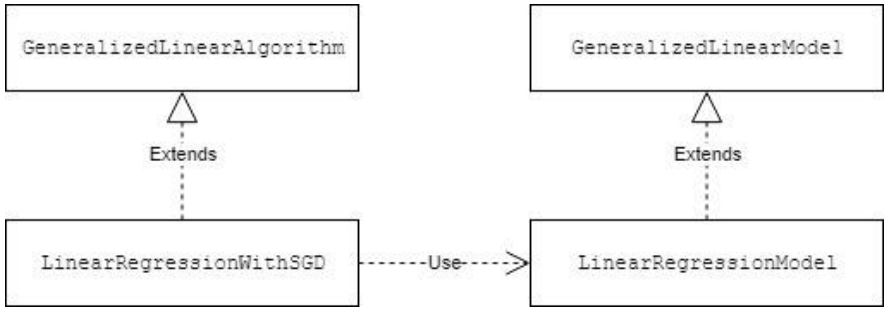

Рисунок 1 – Диаграмма классов реализации линейной регрессии в Spark MLlib RDD API

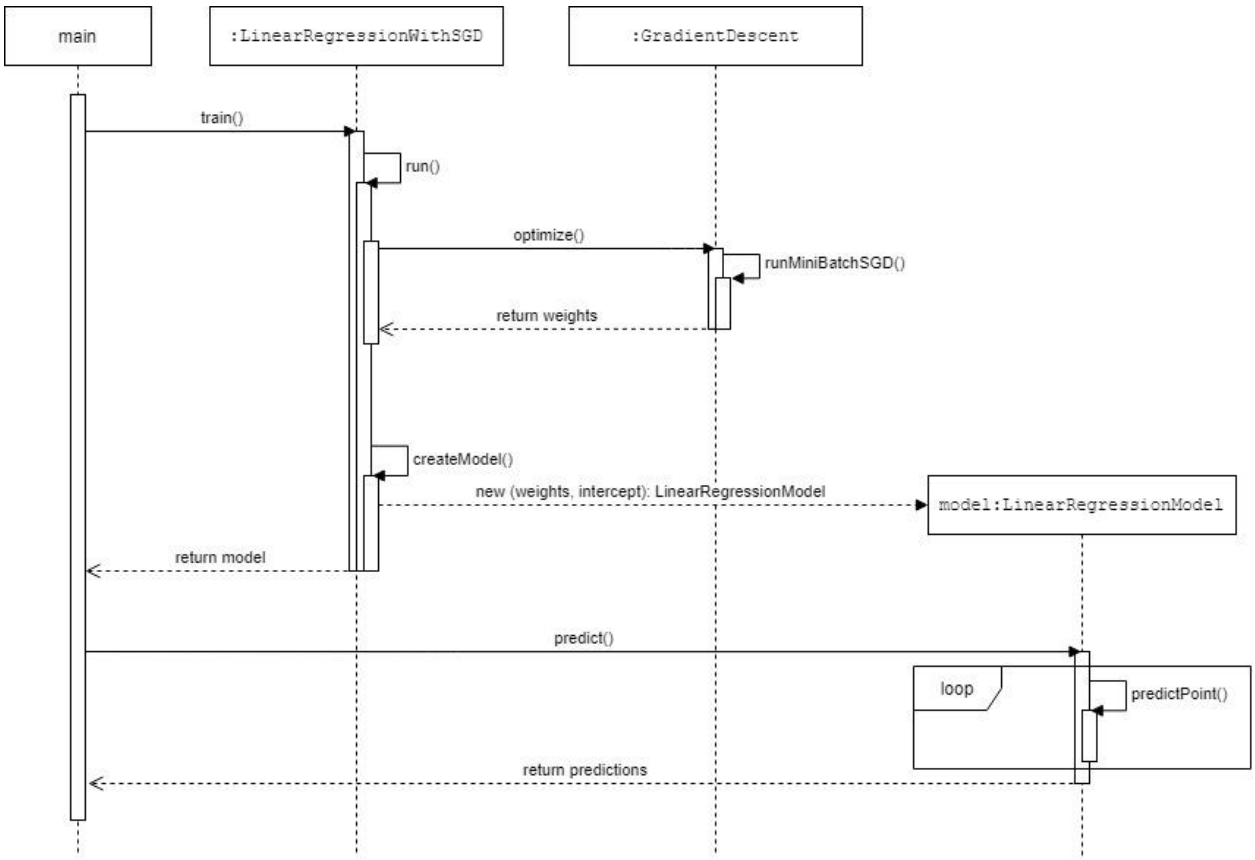

Рисунок 2 – Диаграмма последовательности для обучения и предсказания посредством линейной регрессии в Spark MLlib RDD API

Листинг 1. Spark MLlib RDD API. Linear Regression with SGD

```
@Since("0.8.0")
class LinearRegressionWithSGD private[mllib] (
    private var stepSize: Double,
    private var numIterations: Int,
    private var regParam: Double,
    private var miniBatchFraction: Double)
  extends GeneralizedLinearAlgorithm[LinearRegressionModel] with Serializable {

  private val gradient = new LeastSquaresGradient()
  private val updater = new SimpleUpdater()
  @Since("0.8.0")
  override val optimizer = new GradientDescent(gradient, updater)
    .setStepSize(stepSize)
    .setNumIterations(numIterations)
    .setRegParam(regParam)
    .setMiniBatchFraction(miniBatchFraction)

  /**
   * Construct a LinearRegression object with default parameters: {stepSize: 1.0,
   * numIterations: 100, miniBatchFraction: 1.0}.
   */
  @Since("0.8.0")
  @deprecated("Use ml.regression.LinearRegression or LBFGS", "2.0.0")
  def this() = this(1.0, 100, 0.0, 1.0)

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

@Since("0.8.0")
@DeveloperApi
abstract class GeneralizedLinearAlgorithm[M <: GeneralizedLinearModel]
  extends Logging with Serializable {
```

```
  @Since("0.8.0")
  def optimizer: Optimizer

  protected def createModel(weights: Vector, intercept: Double): M

  @Since("0.8.0")
  def run(input: RDD[LabeledPoint]): M = {
    run(input, generateInitialWeights(input))
  }

  @Since("1.0.0")
  def run(input: RDD[LabeledPoint], initialWeights: Vector): M = {

    val weightsWithIntercept = optimizer.optimize(data, initialWeightsWithIntercept)

    val intercept = if (addIntercept && numOfLinearPredictor == 1) {
      weightsWithIntercept(weightsWithIntercept.size - 1)
    } else {
      0.0
    }

    var weights = if (addIntercept && numOfLinearPredictor == 1) {
      Vectors.dense(weightsWithIntercept.toArray.slice(0, weightsWithIntercept.size - 1))
    } else {
      weightsWithIntercept
    }

    createModel(weights, intercept)
  }
}
```

Листинг 2. Spark MLlib RDD API. Linear Regression with SGD. Gradient

```
@DeveloperApi
class LeastSquaresGradient extends Gradient {
  override def compute(data: Vector, label: Double, weights: Vector): (Vector, Double) = {
    val diff = dot(data, weights) - label
    val loss = diff * diff / 2.0
    val gradient = data.copy
    scal(diff, gradient)
    (gradient, loss)
  }

  override def compute(
      data: Vector,
      label: Double,
      weights: Vector,
      cumGradient: Vector): Double = {
    val diff = dot(data, weights) - label
    axpy(diff, data, cumGradient)
    diff * diff / 2.0
  }
}
```

Листинг 3. Spark MLlib RDD API. Linear Regression with SGD. Updater

```
@DeveloperApi
class SimpleUpdater extends Updater {
  override def compute(
      weightsOld: Vector,
      gradient: Vector,
      stepSize: Double,
      iter: Int,
      regParam: Double): (Vector, Double) = {
    val thisIterStepSize = stepSize / math.sqrt(iter)
    val brzWeights: BV[Double] = weightsOld.asBreeze.toDenseVector
    brzAxpy(-thisIterStepSize, gradient.asBreeze, brzWeights)

    (Vectors.fromBreeze(brzWeights), 0)
  }
}
```

Листинг 4. Spark MLlib RDD API. Linear Regression with SGD. Gradient Descent

```
@DeveloperApi
object GradientDescent extends Logging {

  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 previousWeights: Option[Vector] = None
    var currentWeights: Option[Vector] = None

    val numExamples = data.count()

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

    var converged = false // indicates whether converged based on convergenceTol
    var i = 1
    while (!converged && i <= 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)
          })
      bcWeights.destroy(blocking = false)

      if (miniBatchSize > 0) {
        /**
         * lossSum is computed using the weights from the previous iteration
         * and regVal is the regularization value computed in the previous iteration as well.
         */
        stochasticLossHistory += lossSum / miniBatchSize + regVal
        val update = updater.compute(
          weights, Vectors.fromBreeze(gradientSum / miniBatchSize.toDouble),
          stepSize, i, regParam)
        weights = update._1
        regVal = update._2

        previousWeights = currentWeights
        currentWeights = Some(weights)
        if (previousWeights != None && currentWeights != None) {
          converged = isConverged(previousWeights.get,
            currentWeights.get, convergenceTol)
        }
      } else {
        logWarning(s"Iteration ($i/$numIterations). The size of sampled batch is zero")
      }
      i += 1
    }
```

```
    logInfo("GradientDescent.runMiniBatchSGD finished. Last 10 stochastic losses %s".format(
      stochasticLossHistory.takeRight(10).mkString(", ")))

    (weights, stochasticLossHistory.toArray)

  }

  private def isConverged(
      previousWeights: Vector,
      currentWeights: Vector,
      convergenceTol: Double): Boolean = {
    // To compare with convergence tolerance.
    val previousBDV = previousWeights.asBreeze.toDenseVector
    val currentBDV = currentWeights.asBreeze.toDenseVector

    // This represents the difference of updated weights in the iteration.
    val solutionVecDiff: Double = norm(previousBDV - currentBDV)

    solutionVecDiff < convergenceTol * Math.max(norm(currentBDV), 1.0)
  }
}
```

Список литературы

- Source Code of MLlib for RDD API. URL: https://github.com/apache/spark/tree/master/mllib/src/main/scala/org/apache/spark/mllib