在目前 spark 的 MLLib 中实现的 SVM 算法是 SVMWithSGD 支持向量机。Hinge-loss 说明：

1、目标函数： y = wx （注：w 是超平面的法向量）

2、损失函数(当前误差)：HingeGradient

公式： max(0, 1 - (2 y - 1) f_w(x)))

解释：Spark 的 SVM 算法要求他的类别符号是{0，1}，其中 y 就是类别符号，(2y -1) 运算的时候，当 y=0 时，2y-1=-1， 当 y=1 时，2y-1=+1，f_w(x)是新输入的样本点的类别值。

3、机梯度下降

梯度： -(2y - 1) * x

正则项： L2 = (1/2) * w^2

权值更新方法： weight = weight - **lambda** (gradient + regParam * weight)

在这个中是：weight = weight - **stepSize/sqrt(iter)** * (gradient + regParam * weight)

4、SVM 算法的第一阶段输入是训练数据样本，输出是训练模型，就是超平面，

第二阶段的输入是数据样本，输出的是样本的类别,输出数据所属的类别。

1.8.2、实现 API 及其说明

功能类	方法	方法说明
MLUtils	loadLibSVMFile 返回 RDD (符合 LIBSVM 的 RDD)	从文件中加载具有类别的数据样本，其中数据的格式是： {label index1:value1 index2:value2 ...}
	saveAsLibSVMFile	以 LIBSVM 的格式保存数据，已经标记好的

		数据。
SVMWithSGD	train 返回的是 SVMModel 实际上是生成了一个新的 SVMWithSGD 的实例。	在给定的 RDD 上进行训练，方法的参数是： 1、rdd 训练的 RDD 2、梯度下降的迭代次数 3、每个迭代的步骤 4、正则化参数 5、每一个迭代使用的数据部分

1.8.3、实现解析

SVMWithSGDExample 类（自己编写）

val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_data.txt")
// Split data into training (60%) and test (40%).
val splits = data.randomSplit(Array(0.6, 0.4), seed = 11L)
val training = splits(0).cache()
val test = splits(1)
// Run training algorithm to build the model
val numIterations = 100
val model = SVMWithSGD.train(training, numIterations)
/***/
loadLibSVMFile 方法的参数：
1、sparkcontext
2、文件路径
3、特征数量
4、最小的数据分片

org.apache.spark.mllib.classification.SVMWithSGD 类

```
def train(
      input: RDD[LabeledPoint],
      numIterations: Int,
      stepSize: Double,
      regParam: Double,
      miniBatchFraction: Double,
      initialWeights: Vector): SVMModel = {
   new SVMWithSGD(stepSize, numIterations, regParam, miniBatchFraction)
      .run(input, initialWeights)
}
```

/**

构造函数参数说明：stepSize　每次梯度下降的步幅, 迭代步长，默认为 1.0

numIterations　迭代次数,默认值是 100

regParam　正则式，默认值是 0.01

miniBatchFraction　每次迭代使用的数据的部分，默认是 1.0

initialWeights：每一维的权值，初试是 0.

**/

run 方法说明：方法接收输入的数据和初始的权重，

在 run 方法中个使用的的是 optimizer 的 optimize 方法，接受输入数据和权重，

计算产生相应的梯度下降，在 SVMWithSGD 中覆写了 optimizer 方法，

```
override val optimizer = new GradientDescent(gradient, updater)
                    .setStepSize(stepSize)
                    .setNumIterations(numIterations)
                    .setRegParam(regParam)
                    .setMiniBatchFraction(miniBatchFraction)
```

在这个类中设置相应的参数，比如迭代的次数，使用的正则运算的表达式，以及每一步的步长等。

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

```
  val (weights, _) = GradientDescent.runMiniBatchSGD(
    data,
    gradient,
    updater,
    stepSize,
    numIterations,
    regParam,
    miniBatchFraction,
    initialWeights,
    convergenceTol)
  weights
}
```

在 SVM 算法中是用的是：

```
private val gradient = new HingeGradient()
private val updater = new SquaredL2Updater()
```

```
class HingeGradient extends Gradient {
  override def compute(data: Vector, label: Double, weights: Vector): (Vector, Double) = {
    val dotProduct = dot(data, weights)
    // Our loss function with {0, 1} labels is max(0, 1 - (2y - 1) (f_w(x)))
    // Therefore the gradient is -(2y - 1)*x
    val labelScaled = 2 * label - 1.0
    if (1.0 > labelScaled * dotProduct) {
      val gradient = data.copy
      scal(-labelScaled, gradient)
      (gradient, 1.0 - labelScaled * dotProduct)
    } else {
      (Vectors.sparse(weights.size, Array.empty, Array.empty), 0.0)
    }
  }
  override def compute(
      data: Vector,
      label: Double,
      weights: Vector,
      cumGradient: Vector): Double = {
    val dotProduct = dot(data, weights)

    val labelScaled = 2 * label - 1.0
    if (1.0 > labelScaled * dotProduct) {
      axpy(-labelScaled, data, cumGradient)
      1.0 - labelScaled * dotProduct
    } else {
```

```
      0.0
    }
  }
}
```

```
class SquaredL2Updater 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
    brzWeights :*= (1.0 - thisIterStepSize * regParam)
    brzAxpy(-thisIterStepSize, gradient.asBreeze, brzWeights)
    val norm = brzNorm(brzWeights, 2.0)
    (Vectors.fromBreeze(brzWeights), 0.5 * regParam * norm * norm)
  }
}
```

Optimize 方法中调用了 GradientDescent.runMiniBatchSGD 方法。

```
runMiniBatchSGD ( data: RDD[(Double, Vector)],
    gradient: Gradient,
    updater: Updater,
    stepSize: Double,
    numIterations: Int,
    regParam: Double,
    miniBatchFraction: Double,
    initialWeights: Vector,
    convergenceTol: Double): (Vector, Array[Double]) = {

    val stochasticLossHistory = new ArrayBuffer[Double](numIterations)
  // Record previous weight and current one to calculate solution vector difference
  var previousWeights: Option[Vector] = None
  var currentWeights: Option[Vector] = None
  val numExamples = data.count()

  if (numExamples == 0) {
    return (initialWeights, stochasticLossHistory.toArray)
  }
```

```
  // 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

  var i = 1
  while (!converged && i <= numIterations) {
    val bcWeights = data.context.broadcast(weights)

    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) {
      /**
       * 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)
      weights = update._1
      regVal = update._2
      previousWeights = currentWeights
      currentWeights = Some(weights)
      if (previousWeights != None && currentWeights != None) {
```

```
            converged = isConverged(previousWeights.get,
              currentWeights.get, convergenceTol)
          }
        } else {      }
        i += 1
      }
        (weights, stochasticLossHistory.toArray)
    }


    private def isConverged(
        previousWeights: Vector,
        currentWeights: Vector,
        convergenceTol: Double): Boolean = {

      val previousBDV = previousWeights.asBreeze.toDenseVector
      val currentBDV = currentWeights.asBreeze.toDenseVector
      val solutionVecDiff: Double = norm(previousBDV - currentBDV)
      solutionVecDiff < convergenceTol * Math.max(norm(currentBDV),
    1.0)
    }
```

方法说明：data：输入的数据。

gradient：梯度下降函数

Updater：梯度更新函数

stepSize：第一次迭代的次数

numIteration：迭代次数

regParam：正则表达式

miniBatchFraction：数据量（样本量）

返回值：返回值是相应的权重，和对应的损失值。

算法的执行流程：

MLUtils.loadLibSVMFile —> SVMWithSGD.train —> new SVMWithSGD.run —> Optimzer.optimze —>GradientDescent.runMiniBatchSGD —> updater.compute

和 hingeGradient.compute

算法本身的数据量：

1、输入的数据样本的特点

2、迭代的次数

3、损失的差异值

4、每一步的执行次数