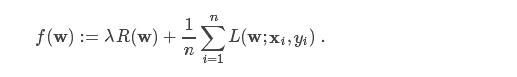
# 1.Mathematical formulation

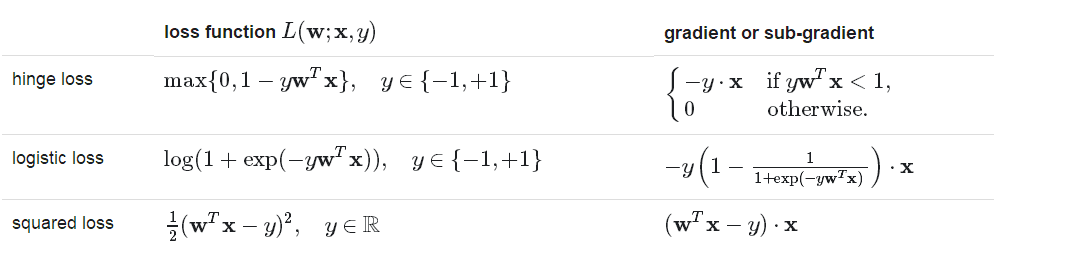
许多标准机器学习方法可以被公式化为凸优化问题，即找到取决于具有d个条目的变量向量w（在代码中称为权重）的凸函数f的最小化的任务。形式上，我们可以将其写为优化问题，其中目标函数形式如下：



这里向量xi∈Rd是训练数据的样本，对于1≤i≤n，yi∈R是它们对应的我们想要预测的标签。如果L（w; **x**，y）可以表示为wTx和y的函数，则称该方法是线性的。spark.mllib的几个分类和回归算法属于这一类，并在此处讨论。目标函数f有两部分：控制模型复杂性的正则化部分，以及测量模型对训练数据的误差的损失部分。损失函数L（w;.）通常是权重w的凸函数。固定正则化参数λ≥0（代码中的regParam）定义了最小化损失（即，训练误差）和最小化模型复杂度（即，避免过度拟合）这两个目标之间的权衡。

## 1.1损失函数

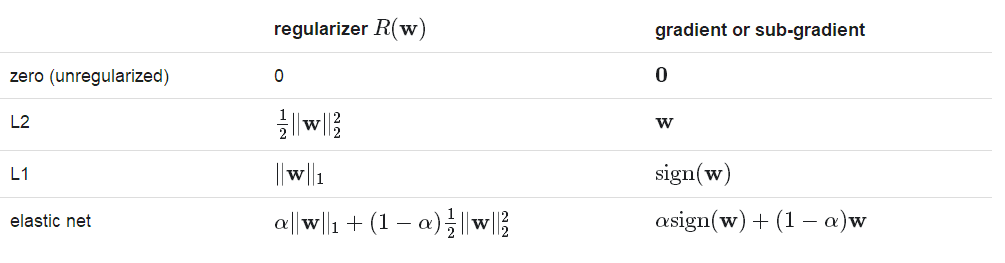
下表总结了spark.mllib支持的方法的损失函数及其渐变或子梯度：



注意，在上面的数学公式中，二元标签y表示为+ 1（正）或-1（负），这便于计算。但是，负标签在spark.mllib中由0表示，而不是-1，以与多分类标签一致。

## 1.2正则化

正则化器的目的是鼓励简单模型并避免过度拟合。 spark.mllib中我们支持以下正规化器：



这里sign（w）是由w的所有条目的sign（±1）组成的向量。由于平滑性，L2正则化问题通常比L1正则化更容易解决。然而，L1正则化可以帮助促进权重的稀疏性，从而导致更小和更可解释的模型，后者可以用于特征选择。 [Elastic net](http://en.wikipedia.org/wiki/Elastic_net_regularization)是L1和L2正则化的组合。不建议在没有任何正则化的情况下训练模型，特别是当训练样本的数量很少时。

## 1.3 优化

线性模型使用凸优化方法来优化目标函数。spark.mllib使用两种方法，SGD和L-BFGS。目前，大多数算法API支持随机梯度下降（SGD），少数支持L-BFGS。

# 2.分类

分类旨在将输入项目分为不同类别。最常见的分类类型是二分类，其中有两类，通常称为正类和负类。如果有两个以上的类别，则称为多类分类。spark.mllib支持两种线性分类方法：线性支持向量机（SVM）和逻辑回归。线性SVM仅支持二分类，而逻辑回归支持二分类和多类分类问题。对于这两种方法，spark.mllib支持L1和L2正则化。训练数据集由MLlib中LabeledPoint的RDD表示，其中标签是从零开始的类索引：0,1,2，...

## 2.1线性SVM(Linear Support Vector Machines)

线性SVM是大规模分类任务的标准方法。它是如上面等式（1）中所述的线性方法，其中由hinge损失给出的公式中的损失函数：



默认情况下，线性SVM使用L2正则化进行训练。我们还支持使用L1正则化。在这种情况下，问题变成线性。线性SVM算法输出SVM模型。给定一个新的数据点，用x表示，该模型根据wTx的值进行预测。默认情况下，如果wTx≥0则结果为正，否则为负。

MLlib的所有方法都使用Java友好类型，因此您可以像在Scala中一样导入和调用它们。唯一需要注意的是，这些方法使用Scala RDD对象，而Spark Java API使用单独的JavaRDD类。您可以通过在JavaRDD对象上调用.rdd（）将Java RDD转换为Scala。下面给出了一个与Scala中提供的示例等效的应用程序示例：

|  |
| --- |
| SparkConf conf = **new** SparkConf().setAppName("JavaSVMWithSGDExample").setMaster("local");  SparkContext sc = **new** SparkContext(conf);  String path = "F:\\Learning\\java\\project\\LearningSpark\\src\\main\\resources\\sample\_libsvm\_data.txt";  JavaRDD<LabeledPoint> data = MLUtils.*loadLibSVMFile*(sc, path).toJavaRDD();  data.take(2);  System.***out***.println(data.take(2).get(0).features().size()); // 692个特征  System.***out***.println(data.take(2).get(1).features().size());  // 将RDD划分为两部分，60%作为训练数据，40%为测试数据  JavaRDD<LabeledPoint> training = data.sample(**false**, 0.6, 11L); // 训练数据  training.cache();  JavaRDD<LabeledPoint> test = data.subtract(training);// 测试数据  // 模型训练  **int** numIterations = 100;  SVMModel model = SVMWithSGD.*train*(training.rdd(), numIterations);  model.clearThreshold();      // //在测试数据集上计算得分  // JavaRDD<Tuple2<Object, Object>>scoreAndLabels=test.map(p->new Tuple2(model.predict(p.features()), p.label()));  // System.out.println(scoreAndLabels.take(1).get(0));  // //获取评估指标  // BinaryClassificationMetrics metrics=new BinaryClassificationMetrics(JavaRDD.toRDD(scoreAndLabels));  // double auRoc=metrics.areaUnderROC();  // System.out.println("area under Roc="+auRoc);  //  String model\_path="F:\\Learning\\java\\project\\LearningSpark\\src\\main\\resources\\model\\javaSVMWithSGDModel";  // model.save(sc, model\_path);    SVMModel model2=SVMModel.*load*(sc, model\_path);  JavaRDD<Tuple2<Object, Object>>testScoreAndLabels=test.map(p->**new** Tuple2(model2.predict(p.features()), p.label()));  System.***out***.println(testScoreAndLabels.take(1).get(0));  sc.stop(); |

默认情况下，SVMWithSGD.train（）方法执行L2正则化，并将正则化参数设置为1.0。如果我们想要配置这个算法，我们可以通过直接创建一个新对象并调用setter方法来进一步自定义SVMWithSGD。所有其他spark.mllib算法也以这种方式支持自定义。例如，以下代码生成SVM的L1正则化，其正则化参数设置为0.1，并运行200次迭代的训练算法。

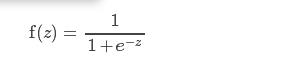
|  |
| --- |
| SVMWithSGD svmAlg = **new** SVMWithSGD();  svmAlg.optimizer().setNumIterations(200).setRegParam(0.1).setUpdater(**new** L1Updater());  SVMModel modelL1 = svmAlg.run(training.rdd()); |

## 2.2 逻辑回归(Logistic Regression)

Logistic回归广泛用于预测二元分类。如上面等式（1）中所述的线性方法，其中由逻辑损失给出的公式中的损失函数：



对于二分类问题，算法输出二元逻辑回归模型。给定一个新的数据点，用x表示，该模型通过应用逻辑函数进行预测



其中z = wTx。 默认情况下，如果f（wTx）> 0.5，则结果为正，否则为负，尽管与线性SVM不同，逻辑回归模型的原始输出f（z）具有 概率解释（即x为正的概率）。二元逻辑回归可以推广到多项Logistic回归中，以训练和预测多类分类问题。例如，对于K个可能的结果，可以选择其中一个结果作为“支点”，并且可以针对支点结果单独回归其他K-1个结果。在spark.mllib中，第一个类0被选为“pivot”类。

对于多类分类问题，该算法将输出多项Logistic回归模型，该模型包含针对第一类回归的K-1个二元逻辑回归模型。给定新的数据点，将运行K-1个模型，并且将选择具有最大概率的类作为预测类。我们实现了两种算法来解决逻辑回归：小批量梯度下降和L-BFGS。 我们建议使用L-BFGS，以实现更快的收敛。

以下代码说明了如何加载样本多类数据集，将其拆分为训练和测试，以及如何使用LogisticRegressionWithLBFGS来拟合逻辑回归模型。然后根据测试数据集评估模型并将其保存到磁盘。

|  |
| --- |
| SparkConf conf = **new** SparkConf().setAppName("JavaLogisticRegressionWithLBFGSExample").setMaster("local");  SparkContext sc = **new** SparkContext(conf);  String path = "F:\\Learning\\java\\project\\LearningSpark\\src\\main\\resources\\sample\_libsvm\_data.txt";  JavaRDD<LabeledPoint> data = MLUtils.*loadLibSVMFile*(sc, path).toJavaRDD();  // 划分训练机和测试集  JavaRDD<LabeledPoint>[] splits = data.randomSplit(**new** **double**[] { 0.6, 0.4 }, 11L);  JavaRDD<LabeledPoint> tringing = splits[0].cache();  JavaRDD<LabeledPoint> test = splits[1];  // 模型构建和训练  LogisticRegressionModel model = **new** LogisticRegressionWithLBFGS().setNumClasses(10).run(tringing.rdd());  // 在测试集上计算得分  JavaPairRDD<Object, Object> predictionAndLabels = test  .mapToPair(p -> **new** Tuple2(model.predict(p.features()), p.label()));  // 获取评价指标  MulticlassMetrics metrics = **new** MulticlassMetrics(predictionAndLabels.rdd());  **double** accuracy = metrics.accuracy();  System.***out***.println("Accuracy:" + accuracy);  model.save(sc, "F:\\Learning\\java\\project\\LearningSpark\\src\\main\\resources\\model\\logistic");  sc.stop(); |