**Spark机器学习回归算法基础**

**教案**

|  |  |  |
| --- | --- | --- |
| **课程教案版本** | **日期** | **备注** |
| **V1.0** | **20190515** |  |
|  |  |  |

# SparkMliib实战回归算法

学习目标：

1.SparkMllib的Pipeline原理及实践

2.SparkMllib模型选择与优化

3.线性回归原理及实战

4.逻辑回归原理及实战

## 1.SparkMliib高级模块

### 1.1 SparkMllib的Pipeline原理及实战

#### 1.1.1Pipeline入门

从Spark1.2版本之后引入了ML Pipeline，经过多个版本的发展，SparkMl克服了Mllib在处理复杂机器学习问题的一些不足，如工作比较复杂，流程不够清晰等，向用户提供基于DataFrame之上的更高层次的API库，以方便的构建复杂的机器学习工作流式应用，使得整个机器学习构建过程更加简单、高效和规范。

一个Pipeline在结构上会包含一个或多个Stage，每一个Stage都会完成一个任务，如数据处理、数据转换、模型训练、参数设置或数据预测等，其中两个Stage为Transformer和Estimator。

Transformer主要是用于操作一个DataFrame数据并生成另一个DataFrame数据，比如决策树模型，一个特征提取器都可以抽象为一个Transformer。

Estimator则主要是用来做模型拟合，用来生成一个Transformer。这些Stage有序组成一个Pipeline。

#### 1.1.2DataFrame组件

import org.apache.spark.sql.SparkSession

object dataframeTest1 {

def main(args: Array[String]): Unit = {

//sparksql入口点

val spark = SparkSession

.builder()

.appName("Spark SQL basic example")

.config("spark.some.config.option", "some-value")

.master("local[2]")

.getOrCreate()

// For implicit conversions like converting RDDs to DataFrames

import spark.implicits.\_

val df1 = spark.read.option("header", true).format("csv").load("/Users/zhao-chj/develop/IdeaProjects/SparkTest/src/main/scala/com/test/spark/sparkmllib\_helloworld/Pipeline\_ml\_1/customer.txt")

// Displays the content of the DataFrame to stdout

df1.show()

//转换字符类型

val df2 = df1.select(

df1("name").cast("String"),

df1("age").cast("Double"),

df1("gender").cast("String"))

df2.printSchema()

df2.createOrReplaceTempView("customer") // createOrReplaceGlobalTempView

df2.select("name").filter($"age" > 25).show()

val cust1 = spark.sql("select \* from customer where age between 30 and 50")

cust1.limit(5).show()

val cust2 = spark.sql("select \* from customer where gender like 'M'")

cust2.limit(5).show()

}

}

#### 1.1.3Pipeline组件

* **DataFrame**:ML API使用这个来自Spark SQL的概念作为ML dataset，可以保存多种数据类型。比如：使用不同的列存储文本、特征向量、真实标签和预测结果。**Spark ML采用[SchemaRDD](http://spark.apache.org/docs/1.2.0/api/scala/index.html" \l "org.apache.spark.sql.SchemaRDD)Spark SQL，以便在统一的Dataset概念下支持各种数据类型。**
* **Transformer**:转化器，**包括特征变化和学习模型两部分。**这是个是指一个算法将一个DataFrame transform成另一个DataFrame。也就是训练好的模型。比如：一个ML模型就是一个Transformer能够将一个特征数据的DataFrame转成预测结果的DataFrame。
* **Estimator**:估计器，**是指一个操作DataFrame产生Transformer的算法。**比如：一个学习算法就是一个Estimator，可以在一个DataFrame上训练得到一个模型。
  + 估计器通过调用fit方法，接收一个DataFrame产生一个模型，这个模型就是一个Transformer。比如逻辑回归就是一个调用fit方法训练后产生的一个逻辑回归模型。
* **Pipeline**:一个Pipeline链是将多个Transformer和Estimator组合在一起组成一个ML workflow。
* [Param](http://spark.apache.org/docs/1.2.0/ml-guide.html" \l "parameters)：所有Transformers和Estimators现在共享一个用于指定参数的通用API。

Parameter:所有的Transformer和Estimator共享一个公共的说明参数的API。

保存或加载管道 通常情况下，将模型或管道保存到磁盘供以后使用是值得的。模型的导入导出功能在spark1.6的时候加入了pipeline API。大多数基础transformers和基本ML models都支持。

可以将训练好的pipeline输出到磁盘保存起来

model.write.overwrite().save("/opt/spark-logistic-regression-model")

#### 1.1.4Pipeline原理

参考：[http://spark.apache.org/docs/2.2.0/ml-guide.html](http://spark.apache.org/docs/1.2.0/ml-guide.html)

在机器学习中，通常运行一系列算法来处理和训练数据。例如，简单的文本文档处理工作流程可能包括几个阶段：

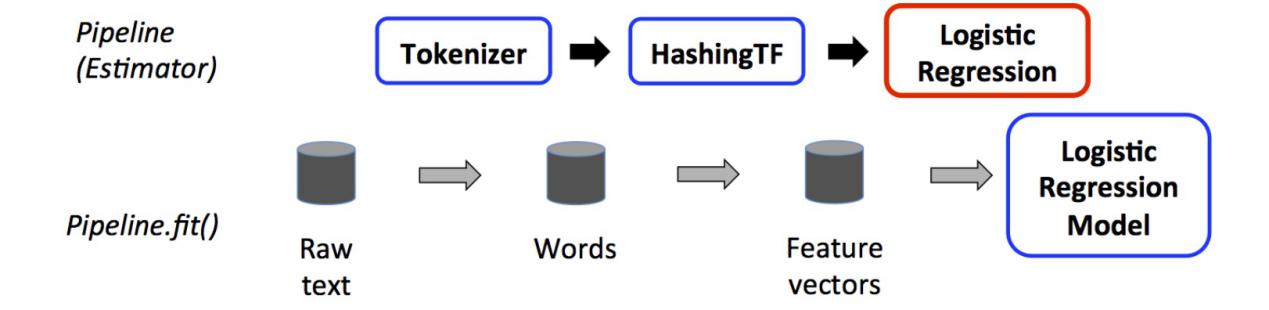
* 将每个文档的文本拆分为单词。
* 将每个文档的单词转换为数字特征向量。
* 使用特征向量和标签学习预测模型。

Spark ML表示这样的工作流程[Pipeline](http://spark.apache.org/docs/1.2.0/api/scala/index.html" \l "org.apache.spark.ml.Pipeline)，它由一系列以特定顺序运行的[PipelineStages](http://spark.apache.org/docs/1.2.0/api/scala/index.html" \l "org.apache.spark.ml.PipelineStage)（Transformers和Estimators）组成。

这个究竟是怎么运作？

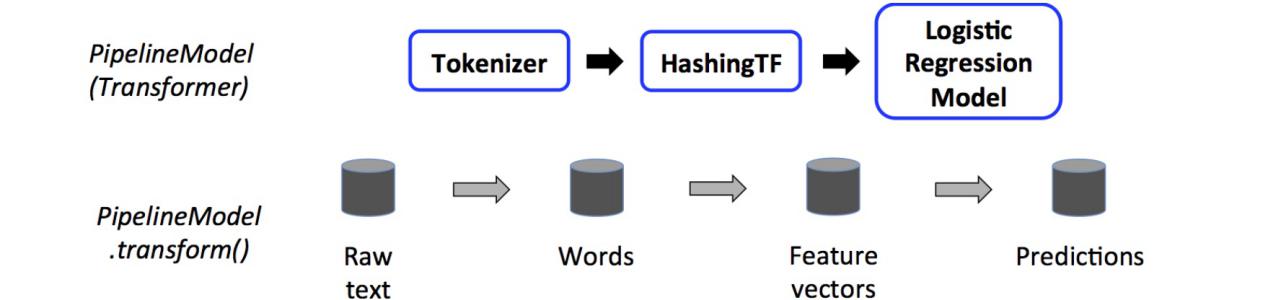
一个Pipeline被指定为阶段序列，每个阶段是一个Transformer或一个 Estimator。这些阶段按顺序运行，输入数据集在通过每个阶段时进行修改。对于Transformer阶段，transform()在数据集上调用该方法。对于Estimator阶段，fit()方法被调用，以产生Transformer（它成为的部分PipelineModel或Pipeline），以及Transformer的transform()方法被称为数据集上。

我们为简单的文本文档工作流说明了这一点。下图是针对a的训练时间使用情况Pipeline。



上面，顶行表示Pipeline三个阶段。前两个（Tokenizer和HashingTF）是Transformers（蓝色），第三个（LogisticRegression）是Estimator（红色）。底行表示流经管道的数据，其中柱面表示SchemaRDDs。Pipeline.fit()在**原始文本文档和标签**的原始数据集上调用该方法。该Tokenizer.transform()方法将原始文本文档拆分为单词，将带有单词的新列添加到数据集中。该HashingTF.transform()方法将单词列转换为要素向量，将具有这些向量的新列添加到数据集。现在，因为LogisticRegression是一个Estimator，Pipeline第一次调用LogisticRegression.fit()生产一个LogisticRegressionModel。如果Pipeline有更多的阶段，它会称之为LogisticRegressionModelstransform() 在将数据集传递到下一阶段之前对数据集的方法。

一个Pipeline是一个Estimator。因此，之后Pipeline的fit()方法运行时，它产生一个PipelineModel其实是Transformer。这PipelineModel是在测试时使用的 ; 如下图：



在上图中，它PipelineModel具有与原始相同的阶段数Pipeline，但原始中的所有Estimators Pipeline都变为Transformers。当PipelineModel的transform()方法用在测试数据集，该数据通过顺序给传递Pipeline。**每个阶段的transform()方法都会更新数据集并将其传递到下一个阶段。**

Pipelines和PipelineModels有助于确保训练和测试数据经过相同的功能处理步骤。

**细节**

* + DAG Pipeline：一个Pipeline的级被指定为一个有序阵列。这里给出的例子都是线性Pipelines，即Pipelines，其中每个阶段使用前一阶段产生的数据。Pipeline只要数据流图形成有向无环图（DAG），就可以创建非线性PIpelines。目前，此图基于每个阶段的输入和输出列名称（通常指定为参数）隐式指定。如果Pipeline形成DAG，则必须按拓扑顺序指定阶段。
  + 运行时检查：由于Pipelines可以对具有不同类型的数据集进行操作，因此它们不能使用编译时类型检查。 Pipelines和PipelineModels代替在实际运行之前进行运行时检查Pipeline。此类型检查是使用数据集模式完成的，该模式是对列中数据类型的描述SchemaRDD。

**参数：**

Spark ML Estimator和Transformers使用统一的API来指定参数。

一个[Param](http://spark.apache.org/docs/1.2.0/api/scala/index.html" \l "org.apache.spark.ml.param.Param)是带有自包含文档的命名参数。一个[ParamMap](http://spark.apache.org/docs/1.2.0/api/scala/index.html" \l "org.apache.spark.ml.param.ParamMap)是一组（参数，值）对。

**将参数传递给算法有两种主要方法：**

设置实例的参数。例如，如果lr是实例LogisticRegression，则可以调用lr.setMaxIter(10)以lr.fit()最多使用10次迭代。此API类似于MLlib中使用的API。

传递ParamMap到fit()或transform()。在任何参数ParamMap将覆盖以前通过setter方法指定的参数。

参数属于Estimators和Transformers的特定实例。例如，如果我们有两个LogisticRegression实例lr1和lr2，然后我们可以建立一个ParamMap与两个maxIter指定的参数：ParamMap(lr1.maxIter -> 10, lr2.maxIter -> 20)。

#### 1.1.5PipeLine实践

##### （1）Estimator，Transformer和Param

import org.apache.spark.{SparkConf, SparkContext}

import org.apache.spark.ml.classification.LogisticRegression

import org.apache.spark.ml.param.ParamMap

import org.apache.spark.ml.classification.LogisticRegression

import org.apache.spark.ml.linalg.{Vector, Vectors}

import org.apache.spark.ml.param.ParamMap

import org.apache.spark.sql.Row

// Prepare training data from a list of (label, features) tuples.

val training = spark.createDataFrame(Seq(

(1.0, Vectors.dense(0.0, 1.1, 0.1)),

(0.0, Vectors.dense(2.0, 1.0, -1.0)),

(0.0, Vectors.dense(2.0, 1.3, 1.0)),

(1.0, Vectors.dense(0.0, 1.2, -0.5))

)).toDF("label", "features")

// Create a LogisticRegression instance. This instance is an Estimator.

val lr = new LogisticRegression()

// Print out the parameters, documentation, and any default values.

println("LogisticRegression parameters:\n" + lr.explainParams() + "\n")

// We may set parameters using setter methods.

lr.setMaxIter(10)

.setRegParam(0.01)

// Learn a LogisticRegression model. This uses the parameters stored in lr.

val model1 = lr.fit(training)

// Since model1 is a Model (i.e., a Transformer produced by an Estimator),

// we can view the parameters it used during fit().

// This prints the parameter (name: value) pairs, where names are unique IDs for this

// LogisticRegression instance.

println("Model 1 was fit using parameters: " + model1.parent.extractParamMap)

// We may alternatively specify parameters using a ParamMap,

// which supports several methods for specifying parameters.

val paramMap = ParamMap(lr.maxIter -> 20)

.put(lr.maxIter, 30) // Specify 1 Param. This overwrites the original maxIter.

.put(lr.regParam -> 0.1, lr.threshold -> 0.55) // Specify multiple Params.

// One can also combine ParamMaps.

val paramMap2 = ParamMap(lr.probabilityCol -> "myProbability") // Change output column name.

val paramMapCombined = paramMap ++ paramMap2

// Now learn a new model using the paramMapCombined parameters.

// paramMapCombined overrides all parameters set earlier via lr.set\* methods.

val model2 = lr.fit(training, paramMapCombined)

println("Model 2 was fit using parameters: " + model2.parent.extractParamMap)

// Prepare test data.

val test = spark.createDataFrame(Seq(

(1.0, Vectors.dense(-1.0, 1.5, 1.3)),

(0.0, Vectors.dense(3.0, 2.0, -0.1)),

(1.0, Vectors.dense(0.0, 2.2, -1.5))

)).toDF("label", "features")

// Make predictions on test data using the Transformer.transform() method.

// LogisticRegression.transform will only use the 'features' column.

// Note that model2.transform() outputs a 'myProbability' column instead of the usual

// 'probability' column since we renamed the lr.probabilityCol parameter previously.

model2.transform(test)

.select("features", "label", "myProbability", "prediction")

.collect()

.foreach { case Row(features: Vector, label: Double, prob: Vector, prediction: Double) =>

println(s"($features, $label) -> prob=$prob, prediction=$prediction") }

##### （2）PIpeline文本处理

import org.apache.spark.ml.{Pipeline, PipelineModel}

import org.apache.spark.ml.classification.LogisticRegression

import org.apache.spark.ml.feature.{HashingTF, Tokenizer}

import org.apache.spark.ml.linalg.Vector

import org.apache.spark.sql.Row

// Prepare training documents from a list of (id, text, label) tuples.

val training = spark.createDataFrame(Seq(

(0L, "a b c d e spark", 1.0),

(1L, "b d", 0.0),

(2L, "spark f g h", 1.0),

(3L, "hadoop mapreduce", 0.0)

)).toDF("id", "text", "label")

// Configure an ML pipeline, which consists of three stages: tokenizer, hashingTF, and lr.

val tokenizer = new Tokenizer()

.setInputCol("text")

.setOutputCol("words")

val hashingTF = new HashingTF()

.setNumFeatures(1000)

.setInputCol(tokenizer.getOutputCol)

.setOutputCol("features")

val lr = new LogisticRegression()

.setMaxIter(10)

.setRegParam(0.001)

val pipeline = new Pipeline()

.setStages(Array(tokenizer, hashingTF, lr))

// Fit the pipeline to training documents.

val model = pipeline.fit(training)

// Now we can optionally save the fitted pipeline to disk

model.write.overwrite().save("/tmp/spark-logistic-regression-model")

// We can also save this unfit pipeline to disk

pipeline.write.overwrite().save("/tmp/unfit-lr-model")

// And load it back in during production

val sameModel = PipelineModel.load("/tmp/spark-logistic-regression-model")

// Prepare test documents, which are unlabeled (id, text) tuples.

val test = spark.createDataFrame(Seq(

(4L, "spark i j k"),

(5L, "l m n"),

(6L, "spark hadoop spark"),

(7L, "apache hadoop")

)).toDF("id", "text")

// Make predictions on test documents.

model.transform(test)

.select("id", "text", "probability", "prediction")

.collect()

.foreach { case Row(id: Long, text: String, prob: Vector, prediction: Double) =>

println(s"($id, $text) --> prob=$prob, prediction=$prediction")

}

##### （3）交叉验证

ML中的一项重要任务是模型选择，或使用数据来查找给定任务的最佳模型或参数。这也称为调参。 Pipeline有助于模型选择，方便Pipeline一次调整整个模型，而不是Pipeline单独调整每个部分。

目前，spark.ml支持使用CrossValidator类的模型选择，该类采用Estimator一组ParamMaps和一组Evaluator。 CrossValidator首先将数据集拆分为一组折叠，用作单独的训练和测试数据集; 例如，将一个数据集切分为3个（训练，测试）数据集子集，每个数据集对使用2/3的数据进行训练，1/3进行测试。 遍历一组数据。对于每一个，它训练给定的参数并使用给定的该参数评估模型，ParamMap中产生的参数比较之后产生最好的评价指标（平均值的倍）被选择作为最佳模型。

下面讲解如何使用CrossValidator从GridSearch中进行选择。为了帮助构造参数网格，我们使用该ParamGridBuilder实用程序。

请注意，通过参数网格进行交叉验证非常昂贵。例如，在下面的示例中，参数网格具有3个值hashingTF.numFeatures和2个值lr.regParam，而且CrossValidator使用2个折叠。这会成倍增加到正在训练的不同模型。在实际设置中，通常可以尝试更多参数并使用更多折叠（并且是常见的）。换句话说，使用可能非常昂贵。然而，它也是一种成熟的方法，用于选择比启发式手动调整更具统计学特征的参数。得到(3×2)×2=12k个不同模型，一般训练模型的时候用的3则交叉验证和10则交叉验证较多。

import org.apache.spark.ml.Pipeline

import org.apache.spark.ml.classification.LogisticRegression

import org.apache.spark.ml.evaluation.BinaryClassificationEvaluator

import org.apache.spark.ml.feature.{HashingTF, Tokenizer}

import org.apache.spark.ml.linalg.Vector

import org.apache.spark.ml.tuning.{CrossValidator, ParamGridBuilder}

import org.apache.spark.sql.Row

// Prepare training data from a list of (id, text, label) tuples.

val training = spark.createDataFrame(Seq(

(0L, "a b c d e spark", 1.0),

(1L, "b d", 0.0),

(2L, "spark f g h", 1.0),

(3L, "hadoop mapreduce", 0.0),

(4L, "b spark who", 1.0),

(5L, "g d a y", 0.0),

(6L, "spark fly", 1.0),

(7L, "was mapreduce", 0.0),

(8L, "e spark program", 1.0),

(9L, "a e c l", 0.0),

(10L, "spark compile", 1.0),

(11L, "hadoop software", 0.0)

)).toDF("id", "text", "label")

// Configure an ML pipeline, which consists of three stages: tokenizer, hashingTF, and lr.

val tokenizer = new Tokenizer()

.setInputCol("text")

.setOutputCol("words")

val hashingTF = new HashingTF()

.setInputCol(tokenizer.getOutputCol)

.setOutputCol("features")

val lr = new LogisticRegression()

.setMaxIter(10)

val pipeline = new Pipeline()

.setStages(Array(tokenizer, hashingTF, lr))

// We use a ParamGridBuilder to construct a grid of parameters to search over.

// With 3 values for hashingTF.numFeatures and 2 values for lr.regParam,

// this grid will have 3 x 2 = 6 parameter settings for CrossValidator to choose from.

val paramGrid = new ParamGridBuilder()

.addGrid(hashingTF.numFeatures, Array(10, 100, 1000))

.addGrid(lr.regParam, Array(0.1, 0.01))

.build()

// We now treat the Pipeline as an Estimator, wrapping it in a CrossValidator instance.

// This will allow us to jointly choose parameters for all Pipeline stages.

// A CrossValidator requires an Estimator, a set of Estimator ParamMaps, and an Evaluator.

// Note that the evaluator here is a BinaryClassificationEvaluator and its default metric

// is areaUnderROC.

val cv = new CrossValidator()

.setEstimator(pipeline)

.setEvaluator(new BinaryClassificationEvaluator)

.setEstimatorParamMaps(paramGrid)

.setNumFolds(2) // Use 3+ in practice

// Run cross-validation, and choose the best set of parameters.

val cvModel = cv.fit(training)

// Prepare test documents, which are unlabeled (id, text) tuples.

val test = spark.createDataFrame(Seq(

(4L, "spark i j k"),

(5L, "l m n"),

(6L, "mapreduce spark"),

(7L, "apache hadoop")

)).toDF("id", "text")

// Make predictions on test documents. cvModel uses the best model found (lrModel).

cvModel.transform(test)

.select("id", "text", "probability", "prediction")

.collect()

.foreach { case Row(id: Long, text: String, prob: Vector, prediction: Double) =>

println(s"($id, $text) --> prob=$prob, prediction=$prediction")

}

### 1.2 SparkMllib的模型选择和优化

##### 1.2.1模型选择

**ML中的一个重要任务是模型选择**，或使用数据找到给定任务的最佳模型或参数。这也叫**调优**。可以针对个体估算器（如Logistic回归）或包括多个算法，特征化和其他步骤的整个管道完成调整。用户可以一次调整整个流水线，而不是单独调整管道中的每个元素。

MLlib支持使用 CrossValidator和TrainValidationSplit等工具进行模型选择。这些工具需要以下项目：

* Estimator（估算器）：算法或管道调整
* ParamMaps集：可供选择的参数，有时称为“**参数网格**”进行搜索
* Evaluator（评估者）：衡量拟合模型对延伸测试数据有多好的度量

这些模型选择工具的工作如下：

* + 他们将输入数据分成单独的训练和测试数据集。
  + 对于每个（训练，测试）对，他们一组ParamMaps集合进行迭代；
  + 对于每个ParamMap，它们使用这些参数适合Estimator，获得拟合的Model，并使用Evaluator评估Model的性能。

他们选择由最佳性能参数组合生成的模型。

评估者可以是回归问题的RegressionEvaluator，二进制数据的BinaryClassificationEvaluator或多类问题的MulticlassClassificationEvaluator。用于选择最佳ParamMap的默认度量可以被这些评估器中的每一个的setMetricName方法覆盖。

##### 1.2.2交叉验证

CrossValidator首先将数据集分成一组折叠，这些折叠用作单独的训练和测试数据集。 例如，k = 3倍，CrossValidator将生成3个（训练，测试）数据集对，每个数据集使用2/3的数据进行训练，1/3进行测试。为了评估一个特定的ParamMap，CrossValidator通过在3个不同的（训练，测试）数据集对上拟合Estimator来计算3个模型的平均评估度量。在确定最佳ParamMap之后，CrossValidator最终使用最好的ParamMap和整个数据集重新拟合Estimator。

案例：通过交叉-验证进行模型选择，如何使用CrossValidator从参数网格中进行选择。请注意，通过参数网格的交叉验证是昂贵的。 例如，在下面的示例中，参数网格具有3个值，用于hashingTF.numFeatures，2个值用于lr.regParam，CrossValidator使用2折交叉验证。 得到模型一共 (3×2)×2 = 12。一般k=3或10较为常用。

import org.apache.spark.ml.Pipeline

import org.apache.spark.ml.classification.LogisticRegression

import org.apache.spark.ml.evaluation.BinaryClassificationEvaluator

import org.apache.spark.ml.feature.{HashingTF, Tokenizer}

import org.apache.spark.ml.linalg.Vector

import org.apache.spark.ml.tuning.{CrossValidator, ParamGridBuilder}

import org.apache.spark.sql.Row

// Prepare training data from a list of (id, text, label) tuples.

val training = spark.createDataFrame(Seq(

(0L, "a b c d e spark", 1.0),

(1L, "b d", 0.0),

(2L, "spark f g h", 1.0),

(3L, "hadoop mapreduce", 0.0),

(4L, "b spark who", 1.0),

(5L, "g d a y", 0.0),

(6L, "spark fly", 1.0),

(7L, "was mapreduce", 0.0),

(8L, "e spark program", 1.0),

(9L, "a e c l", 0.0),

(10L, "spark compile", 1.0),

(11L, "hadoop software", 0.0)

)).toDF("id", "text", "label")

// Configure an ML pipeline, which consists of three stages: tokenizer, hashingTF, and lr.

val tokenizer = new Tokenizer()

.setInputCol("text")

.setOutputCol("words")

val hashingTF = new HashingTF()

.setInputCol(tokenizer.getOutputCol)

.setOutputCol("features")

val lr = new LogisticRegression()

.setMaxIter(10)

val pipeline = new Pipeline()

.setStages(Array(tokenizer, hashingTF, lr))

// We use a ParamGridBuilder to construct a grid of parameters to search over.

// With 3 values for hashingTF.numFeatures and 2 values for lr.regParam,

// this grid will have 3 x 2 = 6 parameter settings for CrossValidator to choose from.

val paramGrid = new ParamGridBuilder()

.addGrid(hashingTF.numFeatures, Array(10, 100, 1000))

.addGrid(lr.regParam, Array(0.1, 0.01))

.build()

// We now treat the Pipeline as an Estimator, wrapping it in a CrossValidator instance.

// This will allow us to jointly choose parameters for all Pipeline stages.

// A CrossValidator requires an Estimator, a set of Estimator ParamMaps, and an Evaluator.

// Note that the evaluator here is a BinaryClassificationEvaluator and its default metric

// is areaUnderROC.

val cv = new CrossValidator()

.setEstimator(pipeline)

.setEvaluator(new BinaryClassificationEvaluator)

.setEstimatorParamMaps(paramGrid)

.setNumFolds(2) // Use 3+ in practice

// Run cross-validation, and choose the best set of parameters.

val cvModel = cv.fit(training)

// Prepare test documents, which are unlabeled (id, text) tuples.

val test = spark.createDataFrame(Seq(

(4L, "spark i j k"),

(5L, "l m n"),

(6L, "mapreduce spark"),

(7L, "apache hadoop")

)).toDF("id", "text")

// Make predictions on test documents. cvModel uses the best model found (lrModel).

cvModel.transform(test)

.select("id", "text", "probability", "prediction")

.collect()

.foreach { case Row(id: Long, text: String, prob: Vector, prediction: Double) =>

println(s"($id, $text) --> prob=$prob, prediction=$prediction")

}

##### 1.2.3训练验证方法切分

除了CrossValidator Spark，还提供了用于超参数调整的TrainValidationSplit。 TrainValidationSplit仅对参数每个组合进行一次评估，而在CrossValidator的情况下，则不是k次。 因此，它较便宜，但在训练数据集不够大时不会产生可靠的结果。与CrossValidator不同，TrainValidationSplit创建一个（训练，测试）数据集对。 它使用trainRatio参数将数据集分成这两个部分。 例如，trainRatio = 0.75，TrainValidationSplit将生成训练和测试数据集对，其中75％的数据用于训练，25％用于验证。

像CrossValidator一样，TrainValidationSplit最终适合使用最好的ParamMap和整个数据集的Estimator。

代码：

import org.apache.spark.ml.evaluation.RegressionEvaluator

import org.apache.spark.ml.evaluation.RegressionEvaluator

import org.apache.spark.ml.regression.LinearRegression

import org.apache.spark.ml.tuning.{ParamGridBuilder, TrainValidationSplit}

// Prepare training and test data.

val data = spark.read.format("libsvm").load("data/mllib/sample\_linear\_regression\_data.txt")

val Array(training, test) = data.randomSplit(Array(0.9, 0.1), seed = 12345)

val lr = new LinearRegression()

.setMaxIter(10)

// We use a ParamGridBuilder to construct a grid of parameters to search over.

// TrainValidationSplit will try all combinations of values and determine best model using

// the evaluator.

val paramGrid = new ParamGridBuilder()

.addGrid(lr.regParam, Array(0.1, 0.01))

.addGrid(lr.fitIntercept)

.addGrid(lr.elasticNetParam, Array(0.0, 0.5, 1.0))

.build()

// In this case the estimator is simply the linear regression.

// A TrainValidationSplit requires an Estimator, a set of Estimator ParamMaps, and an Evaluator.

val trainValidationSplit = new TrainValidationSplit()

.setEstimator(lr)

.setEvaluator(new RegressionEvaluator)

.setEstimatorParamMaps(paramGrid)

// 80% of the data will be used for training and the remaining 20% for validation.

.setTrainRatio(0.8)

// Run train validation split, and choose the best set of parameters.

val model = trainValidationSplit.fit(training)

// Make predictions on test data. model is the model with combination of parameters

// that performed best.

model.transform(test).select("features", "label", "prediction") .show()

##### 1.2.4自定义模型选择

案例分析：

package cn.itcast.sparkmllib\_day03\_classification  
  
import org.apache.spark.SparkConf  
import org.apache.spark.ml.Pipeline  
import org.apache.spark.ml.classification.DecisionTreeClassifier  
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator  
import org.apache.spark.ml.feature.{StringIndexer, VectorAssembler}  
import org.apache.spark.ml.param.ParamMap  
import org.apache.spark.ml.tuning.{CrossValidator, CrossValidatorModel, ParamGridBuilder, TrainValidationSplit}  
import org.apache.spark.sql.{DataFrame, Dataset, Row, SparkSession}  
  
/\*\*  
 \* DESC: 构建机器学习模型  
 \* Complete data processing and modeling process steps:  
 \* 构建机器学习算法的流程  
 \* \*  
 \*- 1-准备数据  
 \*- 2-数据处理  
 \*- 3-特征工程  
 \*- 特征提取  
 \*- 特征转换  
 \*- 特征选择  
 \*- 4-数据集的拆分  
 \*- 5-准备算法------分类算法  
 \*- 6-训练模型  
 \*- 7-模型预测分析  
 \*- 8-模型的校验  
 \*- 9-模型的保存  
 \*/  
object sparkmllib\_irisModel\_pipeline\_trainvalidationsplit {  
 def main(args: Array[String]): Unit = {  
 // \*- 0-准备环境  
 val conf: SparkConf = new SparkConf().setAppName("sparkmllib\_irisModel").setMaster("local[\*]")  
 val spark: SparkSession = SparkSession.builder().config(conf).getOrCreate()  
 spark.sparkContext.setLogLevel("WARN")  
 // \*- 1-准备数据  
 val path = "D:\\BigData\\Workspace\\SparkMachineLearningTest\\SparkMllib\_BigData30\\src\\main\\resources\\iris.csv"  
 val data: DataFrame = spark.read.format("csv").option("header", "true").option("inferSchema", "true").load(path)  
 data.printSchema()  
 // \*- 2-数据处理  
 // \*- 3-特征工程  
 // \*- 特征提取  
 // \*- 特征转换  
 //3-1-StringIndexer转换模块：  
 //.setInputCol("class") 设置输入列，该列一定是数据集中本身存在的需要处理的特征列，如class特征需要转化为index信息  
 //.setOutputCol("indexedclass") 设置输出列，自己可以设定名字  
 val strIndex: StringIndexer = new StringIndexer().setInputCol("class").setOutputCol("indexedclass")  
 //3-2VectorAssemble  
 //为什么需要做特征组合的转换？原因是因为上述的4个特征信息是独立的，而建立模型需要整合为一个样本含有四个特征列的形式  
 //.setInputCols()需要设置Array数组代表的不同的特征字段  
 //解析：def setInputCols(value: Array[String]): this.type = set(inputCols, value)  
 // .setOutputCol() 自己设定的，显示的是将几个特征整合为一个特征向量的过程  
 val vec: VectorAssembler = new VectorAssembler()  
 .setInputCols(Array("sepal\_length", "sepal\_width", "petal\_length", "petal\_width")).setOutputCol("features")  
 // \*- 特征选择  
 // \*- 4-数据集的拆分  
 // seed随机数种子----保证每次随机切分结果的可重复性  
 val Array(trainingSet, testSet): Array[Dataset[Row]] = data.randomSplit(Array(0.8, 0.2), seed = 123L)  
 // val data = vecResult.randomSplit(Array(0.8, 0.2), seed = 123L)  
 // trainingSet=data(0)  
 // testSet=data(1)  
 // \*- 5-准备算法------分类算法  
 // .setFeaturesCol() 特征列，需要给定string类型的特征列  
 // .setLabelCol() 标签列，需要给定过的是数据中的label列  
 // .setImpurity() 不纯度的度量，entropy和gini系数的参数  
 // .setMaxDepth() 树的深度，默认5  
 // .setMaxBins() 分箱的个数，默认32  
 // .setMinInfoGain() 最小的不纯度，默认0  
 // .setMinInstancesPerNode() 每个节点含有的样本的个数最小的阈值，设定默认值是1  
 val dtc = new DecisionTreeClassifier()  
 .setFeaturesCol("features")  
 .setLabelCol("indexedclass")  
 .setImpurity("entropy")  
 .setMaxDepth(5)  
 .setMaxBins(32)  
 .setMinInfoGain(0.0)  
 .setMinInstancesPerNode(1)  
 .setPredictionCol("prces")  
 // \*- 6-训练模型  
 val pipeline: Pipeline = new Pipeline().setStages(Array(strIndex, vec, dtc))  
 //网格搜索指定超参数  
 val param: Array[ParamMap] = new ParamGridBuilder().addGrid(dtc.impurity, Array("gini", "entropy")).addGrid(dtc.maxDepth, Array(4, 5, 6, 7)).build()  
 // \*- 7-模型的校验  
 //.setLabelCol() 标签列  
 // .setPredictionCol() 预测列  
 // .setMetricName("accuracy") 参数列表  
 val evaluator: MulticlassClassificationEvaluator = new MulticlassClassificationEvaluator()  
 .setLabelCol("indexedclass")  
 .setPredictionCol("prces")  
 .setMetricName("accuracy")  
 //8-交叉验证  
 // .setEstimator() 设置学习器或具体的算法  
 // .setEvaluator() 设置校验方式---分类问题如何检验  
 // .setNumFolds() 几折交叉验证  
 // .setEstimatorParamMaps() 指定的不同的超参数  
 val crossvaliator = new TrainValidationSplit().setEstimator(pipeline).setEvaluator(evaluator).setEstimatorParamMaps(param).setTrainRatio(0.8)  
 val crossModel = crossvaliator.fit(trainingSet)  
 crossModel.transform(testSet)  
 // \*- 7-通过crossvalia方法输出参数  
 val extractParamMap: ParamMap = crossModel.extractParamMap()  
 println(extractParamMap)  
 // {  
 // tvs\_cab848d53081-estimator: pipeline\_8a77b264e2d1,  
 // tvs\_cab848d53081-estimatorParamMaps: [Lorg.apache.spark.ml.param.ParamMap;@78494a59,  
 // tvs\_cab848d53081-evaluator: mcEval\_709a58fd527e,  
 // tvs\_cab848d53081-seed: -1772833110,  
 // tvs\_cab848d53081-trainRatio: 0.8  
 // }  
 import org.apache.spark.ml.PipelineModel  
 crossModel  
 .  
 val pipeModel = crossModel.bestModel.asInstanceOf[PipelineModel]  
 println(pipeModel.stages(2).extractParamMap)  
 /\* {  
 dtc\_e77eebb80fef-cacheNodeIds: false,  
 dtc\_e77eebb80fef-checkpointInterval: 10,  
 dtc\_e77eebb80fef-featuresCol: features,  
 dtc\_e77eebb80fef-impurity: gini,  
 dtc\_e77eebb80fef-labelCol: indexedclass,  
 dtc\_e77eebb80fef-maxBins: 32,  
 dtc\_e77eebb80fef-maxDepth: 4,  
 dtc\_e77eebb80fef-maxMemoryInMB: 256,  
 dtc\_e77eebb80fef-minInfoGain: 0.0,  
 dtc\_e77eebb80fef-minInstancesPerNode: 1,  
 dtc\_e77eebb80fef-predictionCol: prces,  
 dtc\_e77eebb80fef-probabilityCol: probability,  
 dtc\_e77eebb80fef-rawPredictionCol: rawPrediction,  
 dtc\_e77eebb80fef-seed: 159147643  
 }\*/  
 }  
}