数据挖掘算法中有很大一部分都是数据预处理工作，毕竟现有模型都是比较成熟的，只需要学会调用就好，如何把原始数据转化为算法模型适用的数据结构也是很重要的一步。spark ML中提供了对特征的提取（Extracting），转换（transforming）和选择（selecting）工具。

* 特征提取：从原始数据中提取特征
* 特征转换：特征的扩展，特征的转化，特征的修改
* 特征选择：从大规模特征集中选取一个子集

### 特征提取（需查阅图片特征提取，视频特征提取相关算法）

#### TF-IDF

“词频－逆向文件频率”（TF-IDF）是一种在文本挖掘中广泛使用的特征向量化方法，它可以体现一个文档中词语在语料库中的重要程度。

文本特征提取:TF-IDF文档矩阵

基于神经网络的word2vec词嵌入矩阵

基于词频数的文档向量CountVectorizer

词频TF(t,d)是词语t在文档d中出现的次数。文件频率DF(t,D)是包含词语的文档的个数。如果我们只使用词频来衡量重要性，很容易过度强调在文档中经常出现，却没有太多实际信息的词语，比如“a”，“the”以及“of”。如果一个词语经常出现在语料库中，意味着它并不能很好的对文档进行区分。TF-IDF就是在数值化文档信息，衡量词语能提供多少信息以区分文档。定义如下：

IDF(t,D)=log((|D|+1)/(DF(t,D)+1))

​ 此处 |D| 是语料库中总的文档数。公式中使用log函数，当词出现在所有文档中时，它的IDF值变为0。加1是为了避免分母为0的情况。TF-IDF 度量值表示如下：

TFIDF(t,d,D)=TF(td)⋅IDF(t,D)

在Spark ML库中，TF-IDF被分成两部分：TF(HashingTF) 和 IDF

HashingTF 是一个Transformer，在文本处理中，接收词条的集合然后把这些集合转化成固定长度的特征向量。这个算法在哈希的同时会统计各个词条的词频。

IDF是一个Estimator，在一个数据集上应用它的fit（）方法，产生一个IDFModel。 该IDFModel 接收特征向量（由HashingTF产生），然后计算每一个词在文档中出现的频次。IDF会减少那些在语料库中出现频率较高的词的权重。

Spark.mllib 中实现词频率统计使用特征hash的方式，原始特征通过hash函数，映射到一个索引值。后面只需要统计这些索引值的频率，就可以知道对应词的频率。这种方式避免设计一个全局1对1的词到索引的映射，这个映射在映射大量语料库时需要花费更长的时间。但需要注意，通过hash的方式可能会映射到同一个值的情况，即不同的原始特征通过Hash映射后是同一个值。为了降低这种情况出现的概率，我们只能对特征向量升维。i.e., 提高hash表的桶数，默认特征维度是 2^20 = 1,048,576.

在下面的代码段中，我们以一组句子开始。首先使用分解器Tokenizer把句子划分为单个词语。对每一个句子（词袋），我们使用HashingTF将句子转换为特征向量，最后使用IDF重新调整特征向量。这种转换通常可以提高使用文本特征的性能。

在下面的代码中，我们以一组句子为例，使用Tokenizer将每一条句子分解为单词，对每一条句子（词袋），我们使用HashingTF 将其转化为特征向量，最后使用IDF 重新调整特征向量。

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

// 创建一个简单的DataFrame，每一个句子代表一个文档

val sentenceData = spark.createDataFrame(Seq( (0.0, "Hi I heard about Spark"), (0.0, "I wish Java could use case classes"), (1.0, "Logistic regression models are neat") )).toDF("label", "sentence")

// 在得到文档集合后，即可用tokenizer对句子进行分词。

val tokenizer=new Tokenizer().setInputCol("sentence").setOutputCol("words")

Val wordsData=tokenizer.transform(sentenceData) wordsData.show(false)

// 如果不设置为false的话，将会省略掉20个以后的字符 /\* +-----+-----------------------------------+------------------------------------------+ |label|sentence |words | +-----+-----------------------------------+----------------------------------

|0.0 |Hi I heard about Spark |[hi, i, heard, about, spark] |

|0.0 |I wish Java could use case classes |[i, wish, java, could, use, case, classes]|

|1.0 |Logistic regression models are neat|[logistic, regression, models, are, neat] | +-----+-----------------------------------+------------------------------------------+

// 使用HashingTF的transform()方法把句子哈希成特征向量，这里设置哈希表的桶数为2000。

Val hashingTF = new HashingTF() .setInputCol("words").setOutputCol("rawFeatures").setNumFeatures(20)

val featurizedData = hashingTF.transform(wordsData) featurizedData.show(false)

/\* 可以看到，分词序列被变换成一个稀疏特征向量，其中每个单词都被散列成了一个不同的索引值，特征向量在某一维度上的值即该词汇在文档中出现的次数

+-----------------------------------------+

|rawFeatures |

+-----------------------------------------+

|(20,[5,6,9],[2.0,1.0,2.0]) |

|(20,[3,5,12,14,18],[2.0,2.0,1.0,1.0,1.0])|

|(20,[5,12,14,18],[1.0,2.0,1.0,1.0]) |

+-----------------------------------------+

// 最后，使用IDF来对单纯的词频特征向量进行修正，使其更能体现不同词汇对文本的区别能力，IDF是一个Estimator，调用fit()方法并将词频向量传入，即产生一个IDFModel。

val idf = new IDF().setInputCol("rawFeatures").setOutputCol("features")

val idfModel = idf.fit(featurizedData)

// 很显然，IDFModel是一个Transformer，调用它的transform()方法，即可得到每一个单词对应的TF-IDF度量值。

val rescaledData = idfModel.transform(featurizedData) rescaledData.select("label", "features").take(3).foreach(println)

### 特征转换

值得注意的是，用于特征转换的转换器和其他的机器学习算法一样，也属于ML Pipeline模型的一部分，可以用来构成机器学习流水线，以StringIndexer为例，其存储着进行标签数值化过程的相关 超参数，是一个Estimator，对其调用fit(..)方法即可生成相应的模型StringIndexerModel类，很显然，它存储了用于DataFrame进行相关处理的 参数，是一个Transformer（其他转换器也是同一原理）

### Tokenizer

根据源码中的解释 ：

A tokenizer that converts the input string to lowercase and then splits it by white spaces.

tokenizer 主要有两个功能，1. 将词语转为小写字母 2. 按空格划分字符串

注： 如果想使用其他规则划分字符串，可以使用 RegexTokenizer

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

import org.apache.spark.sql.functions.\_

val sentenceDataFrame = spark.createDataFrame(Seq(

(0, "Hi I heard about Spark"),

(1, "I wish Java could use case classes"),

(2, "Logistic,regression,models,are,neat")

)).toDF("id", "sentence")

val tokenizer = new Tokenizer().setInputCol("sentence").setOutputCol("words")

val regexTokenizer = new RegexTokenizer()

.setInputCol("sentence")

.setOutputCol("words")

.setPattern("\\W") // alternatively .setPattern("\\w+").setGaps(false)

// 自定义一个函数

val countTokens = udf { (words: Seq[String]) => words.length }

val tokenized = tokenizer.transform(sentenceDataFrame)

val regexTokenized = regexTokenizer.transform(sentenceDataFrame)

// 增加一个自定义列

tokenized.select("sentence", "words")

.withColumn("tokens", countTokens(col("words"))).show(false)

/\*

+-----------------------------------+------------------------------------------+------+

|sentence |words |tokens|

+-----------------------------------+------------------------------------------+------+

|Hi I heard about Spark |[hi, i, heard, about, spark] |5 |

|I wish Java could use case classes |[i, wish, java, could, use, case, classes] |7 |

|Logistic,regression,models,are,neat |[logistic,regression,models,are,neat] |1 |

+-----------------------------------+------------------------------------------+------+

\*/

regexTokenized.select("sentence", "words")

.withColumn("tokens", countTokens(col("words"))).show(false)

/\*+-----------------------------------+------------------------------------------+------+

|sentence |words |tokens|

+-----------------------------------+------------------------------------------+------+

|Hi I heard about Spark |[hi, i, heard, about, spark] |5 |

|I wish Java could use case classes |[i, wish, java, could, use, case, classes] |7 |

|Logistic,regression,models,are,neat |[logistic, regression, models, are, neat] |5 |

+-----------------------------------+------------------------------------------+------+\*/

很明显的可以看到二者的区别，tokenized是将非空格分割的字符串当做 一个整体。

### Binarizer

连续特征根据阈值二值化，大于阈值的为1.0，小于等于阈值的为0.0。二值化是机器学习中很常见的思路，可以将连续型数据转化为离散型。

import org.apache.spark.ml.feature.Binarizer

val data = Array((0, 0.1), (1, 0.8), (2, 0.2))

val dataFrame = spark.createDataFrame(data).toDF("id", "feature")

val binarizer: Binarizer = new Binarizer() .setInputCol("feature") .setOutputCol("binarized\_feature") .setThreshold(0.5)

val binarizedDataFrame = binarizer.transform(dataFrame)

println(s"Binarizer output with Threshold = ${binarizer.getThreshold}")

binarizedDataFrame.show()

/\*

+---+-------+-----------------+| id|feature|binarized\_feature|

+---+-------+-----------------+

| 0| 0.1| 0.0|

| 1| 0.8| 1.0|| 2| 0.2| 0.0|

+---+-------+-----------------+\*/

### StringIndexer

StringIndexer转换器可以把一列类别型的特征（或标签）进行编码，使其数值化，索引的范围从0开始，该过程可以使得相应的特征索引化，使得某些无法接受类别型特征的算法可以使用，并提高诸如决策树等机器学习算法的效率。

索引构建的顺序为标签的频率，优先编码频率较大的标签，所以出现频率最高的标签为0号。   
如果输入的是数值型的，我们会把它转化成字符型，然后再对其进行编码。直接上代码。

首先创建一个简单的DataFrame，它只包含一个id列和一个标签列category：

import org.apache.spark.ml.feature.StringIndexer

val df = spark.createDataFrame( Seq((0, "a"), (1, "b"), (2, "c"), (3, "a"), (4, "a"), (5, "c"))

).toDF("id", "category")

df.show(false)

/\*

+---+--------+

|id |category|

+---+--------+

|0 |a |

|1 |b |

|2 |c |

|3 |a |

|4 |a ||5 |c |

+---+--------+\*/

随后，我们创建一个StringIndexer对象，设定输入输出列名，其余参数采用默认值，并对这个DataFrame进行训练，产生StringIndexerModel对象, 最后利用该对象对DataFrame进行转换操作。可以看到，StringIndexerModel依次按照出现频率的高低，把字符标签进行了排序，即出现最多的“a”被编号成0，“c”为1，出现最少的“b”为2。

val indexer = new StringIndexer() .setInputCol("category") .setOutputCol("categoryIndex")

val indexed = indexer.fit(df)

val model = indexed.transform(df)

model.show()

/\*+---+--------+-------------+

|id |category|categoryIndex|

+---+--------+-------------+

|0 |a |0.0 |

|1 |b |2.0 |

|2 |c |1.0 |

|3 |a |0.0 |

|4 |a |0.0 |

|5 |c |1.0 |

+---+--------+-------------+\*/

考虑这样一种情况，我们使用已有的数据构建了一个StringIndexerModel，然后再构建一个新的DataFrame，这个DataFrame中有着模型内未曾出现的标签“d”，用已有的模型去转换这一DataFrame会有什么效果？

实际上，如果直接转换的话，Spark会抛出异常，报出“Unseen label: d”的错误。

为了处理这种情况，在模型训练后，可以通过设置setHandleInvalid(“skip”)来忽略掉那些未出现的标签，这样，带有未出现标签的行将直接被过滤掉，

测试一下，我们创建一个df2，包含df未出现的字符d，用df训练的模型来转换df2。

val df2 = spark.createDataFrame( Seq((0, "a"), (1, "b"), (2, "c"), (3, "a"), (4, "a"), (5, "c"), (6, "d"))).toDF("id", "category")

indexed.transform(df2).show(false)

报错：org.apache.spark.SparkException: Unseen label: d.

indexed.setHandleInvalid("skip").transform(df2).show(false)

/\*

+---+--------+-------------+

|id |category|categoryIndex|

+---+--------+-------------+

|0 |a |0.0 |

|1 |b |2.0 |

|2 |c |1.0 |

|3 |a |0.0 |

|4 |a |0.0 |

|5 |c |1.0 |

+---+--------+-------------+

\*/

### IndexToString

与StringIndexer相对应，IndexToString的作用是把标签索引的一列重新映射回原有的字符型标签，一般都是和StringIndexer配合使用，先用StringIndexer将原有标签转化成标签索引，进行模型训练，然后在预测标签的时候再把标签索引转化成原有的字符标签。   
我们还是先来创建个DataFrame

val df = sqlContext.createDataFrame(Seq(

(0, "a"),

(1, "b"),

(2, "c"),

(3, "a"),

(4, "a"),

(5, "c")

)).toDF("id", "category")

然后使用StringIndexer标签索引化。

val indexer = new StringIndexer() .setInputCol("category") .setOutputCol("categoryIndex") .fit(df)

val indexed = indexer.transform(df)/\*

+---+--------+-------------+

|id |category|categoryIndex|

+---+--------+-------------+

|0 |a |0.0 |

|1 |b |2.0 |

|2 |c |1.0 |

|3 |a |0.0 |

|4 |a |0.0 |

|5 |c |1.0 |

+---+--------+-------------+\*/

紧接着，使用IndexToString还原标签索引

val converter = new IndexToString()

.setInputCol("categoryIndex")

.setOutputCol("originalCategory")

val converted = converter.transform(indexed)

/\*

+---+--------+-------------+----------------+

|id |category|categoryIndex|originalCategory|

+---+--------+-------------+----------------+

|0 |a |0.0 |a |

|1 |b |2.0 |b |

|2 |c |1.0 |c |

|3 |a |0.0 |a |

|4 |a |0.0 |a |

|5 |c |1.0 |c |

+---+--------+-------------+----------------+

\*/

### OneHotEncoder

对于每一个特征，如果它有m个可能值，那么经过独热编码后，就变成了m个二元特征。并且，这些特征互斥，每次只有一个激活。因此，数据会变成稀疏的。

将离散特征通过one-hot编码映射到欧式空间，是因为，在回归，分类，聚类等机器学习算法中，**特征之间距离的计算或相似度的计算是非常重要**的，而我们常用的距离或相似度的计算都是在欧式空间的相似度计算，计算余弦相似性，基于的就是欧式空间。   
 将离散型特征使用one-hot编码，可以会让特征之间的距离计算更加合理。比如，有一个离散型特征，代表工作类型，该离散型特征，共有三个取值，不使用one-hot编码，其表示分别是x\_1 = (1), x\_2 = (2), x\_3 = (3)。两个工作之间的距离是，(x\_1, x\_2) = 1, d(x\_2, x\_3) = 1, d(x\_1, x\_3) = 2。那么x\_1和x\_3工作之间就越不相似吗？显然这样的表示，计算出来的特征的距离是不合理。那如果使用one-hot编码，则得到x\_1 = (1, 0, 0), x\_2 = (0, 1, 0), x\_3 = (0, 0, 1)，那么两个工作之间的距离就都是sqrt(2).即每两个工作之间的距离是一样的，显得更合理。

独热编码是指把一列标签索引映射成一列二进制数组，且最多的时候只有一位有效。这种编码适合一些期望类别特征为连续特征的算法（回归），个人感觉独热编码比较适用，其原理这里就不说了，给大家一个连接：   
[http://blog.csdn.net/bitcarmanlee/article/details/51472816](http://blog.csdn.net/bitcarmanlee/article/details/51472816" \t "http://blog.csdn.net/chenguangchun1993/article/details/_blank)   
与sklearn 不同之处在于sparkml中使用SparseVector来表示。

import org.apache.spark.ml.feature.{OneHotEncoder, StringIndexer}

val df = sqlContext.createDataFrame(Seq(

(0, "a"),

(1, "b"),

(2, "c"),

(3, "a"),

(4, "a"),

(5, "c")

)).toDF("id", "category")

val indexer = new StringIndexer()

.setInputCol("category")

.setOutputCol("categoryIndex")

.fit(df)

val indexed = indexer.transform(df)

val encoder = new OneHotEncoder()

.setInputCol("categoryIndex")

.setOutputCol("categoryVec")

val encoded = encoder.transform(indexed)

encoded.select("id", "categoryVec").show()

/\*+---+--------+-------------+-------------+

| id|category|categoryIndex| categoryVec|

+---+--------+-------------+-------------+

| 0| a| 0.0|(2,[0],[1.0])|

| 1| b| 2.0| (2,[],[])|

| 2| c| 1.0|(2,[1],[1.0])|

| 3| a| 0.0|(2,[0],[1.0])|

| 4| a| 0.0|(2,[0],[1.0])|

| 5| c| 1.0|(2,[1],[1.0])|

+---+--------+-------------+-------------+\*/

将categoryVec转换成编码模式：a -> 10 , c ->01 ,至于b为空，是应为spark默认不包含最后一个类别，OneHotEncoder.setDropLast(false)可以设置是否包含最后一个类别。

### VectorIndexer

VectorIndexer解决向量数据集中的类别特征索引。它可以自动识别哪些特征是类别型(离散)的，并且将原始值转换为类别索引。它的处理流程如下：

​ 1. 获得一个向量类型的输入以及maxCategories参数。

​ 2. 基于不同特征值的数量来识别哪些特征需要被类别化，不同特征值的数量小于maxCategories，则这个特征需要被类别化（离散化），否则视为连续值，不做改变。

​ 3.对于每一个类别特征计算0-based（从0开始）类别索引。

索引后的类别特征可以帮助决策树等算法恰当的处理类别型特征，并得到较好结果。

我们读入一个数据集，然后使用VectorIndexer来决定哪些特征需要被作为类别特征，将类别特征转换为他们的索引。

import org.apache.spark.ml.feature.VectorIndexer

val data = Seq(Vectors.dense(-1.0, 1.0, 1.0),Vectors.dense(-1.0, 3.0, 1.0), Vectors.dense(0.0, 5.0, 1.0))

val df = sqlContext.createDataFrame(data.map(Tuple1.apply)).toDF("features")

val indexer = new VectorIndexer()

.setInputCol("features")

.setOutputCol("indexed")

.setMaxCategories(2)

val indexerModel = indexer.fit(df)

val indexedData = indexerModel.transform(df)

indexedData.show()

/\*

+--------------+-------------+

| features| indexed|

+--------------+-------------+

|[-1.0,1.0,1.0] |[1.0,1.0,0.0]|

|[-1.0,3.0,1.0] |[1.0,3.0,0.0]|

|[ 0.0,5.0,1.0] |[0.0,5.0,0.0]|

+--------------+-------------+

\*/

从上例可以看到，我们设置maxCategories为2，即只有种类小于2的特征才被认为是类别型特征，否则被认为是连续型特征。其中类别型特征将被进行编号索引，为了索引的稳定性，规定如果这个特征值为0，则一定会被编号成0，这样可以保证向量的稀疏度。于是，我们可以看到第0类和第2类的特征由于种类数不超过2，被划分成类别型特征，并进行了索引，且为0的特征值也被编号成了0号。

### Normalizer

Normalizer是将数据集的每一行数据归一化的转换器，一行数据看做一个向量，它带一个参数P（默认值是2），归一化就是向量中的每个元素除以向量的范数（P规定使用哪一个范数）。

val data = Seq(Vectors.dense(-1.0, 1.0, 1.0),Vectors.dense(-1.0, 3.0, 1.0), Vectors.dense(0.0, 5.0, 1.0))

val df = sqlContext.createDataFrame(data.map(Tuple1.apply)).toDF("features")

val normalizer = new Normalizer() .setInputCol("features") .setOutputCol("normFeatures") .setP(2.0)

val l1NormData = normalizer.transform(dataFrame)

l1NormData.show()/\*

+--------------+-------------------------------------------------------------+

|features |normFeatures |

+--------------+-------------------------------------------------------------+

|[-1.0,1.0,1.0] |[-0.5773502691896258,0.5773502691896258,0.5773502691896258] |

|[-1.0,3.0,1.0] |[-0.30151134457776363,0.9045340337332909,0.30151134457776363]|

|[0.0,5.0,1.0] |[0.0,0.9805806756909202,0.19611613513818404]

|+--------------+-------------------------------------------------------------+\*/

我们设置的P为2，则使用L2范数来归一化向量，L2范数是指向量各元素的平方和然后求平方根，范数的相关资料可以自行百度，针对[-1,1,1],他的L2范数为3√, 所以归一化后的向量为[−13√,13√,13√]

val lInfNormData = normalizer.transform(dataFrame, normalizer.p -> Double.PositiveInfinity)

lInfNormData.show(false)

/\*

+--------------+--------------------------------------------+

|features |normFeatures |

+--------------+--------------------------------------------+

|[-1.0,1.0,1.0]| [-1.0,1.0,1.0] |

|[-1.0,3.0,1.0] |[-0.3333333333333333,1.0,0.3333333333333333]|

|[0.0,5.0,1.0] |[0.0,1.0,0.2]

|+--------------+--------------------------------------------+\*/

无穷范数就是取向量中的最大值，所以使用无穷范数归一化后的向量就是原向量中元素除以向量中的最大值。

### VectorAssembler

VectorAssembler是非常实用的一个转换器，它的作用很简单，就是把DataFrame中的若干列合并为一个向量列，可类似于Excel中的合并单元格。有了VectorAssembler之后，除了使用反射机制创建DataFrame之外，我们又有了一种新思路，即通过普通方式创建DataFrame，再使用VectorAssembler创造自己所需要的数据格式。

import org.apache.spark.ml.feature.VectorAssembler

import org.apache.spark.mllib.linalg.Vectors

val dataset = sqlcontext.createDataFrame(

Seq((0, 18, 1.0, Vectors.dense(0.0, 10.0, 0.5), 1.0))

).toDF("id", "hour", "mobile", "userFeatures", "clicked")

val assembler = new VectorAssembler()

.setInputCols(Array("hour", "mobile", "userFeatures"))

.setOutputCol("features")

val output = assembler.transform(dataset)

output.show(false)

/\*

+---+----+------+--------------+-------+-----------------------+

|id |hour|mobile|userFeatures |clicked|features |

+---+----+------+--------------+-------+-----------------------+

|0 |18 |1.0 |[0.0,10.0,0.5]| 1.0 |[18.0,1.0,0.0,10.0,0.5]|

+---+----+------+--------------+-------+-----------------------+

\*/

### 特征选择

通常，预处理之后获得的特征有成千上万维，出于去除冗余特征、消除维数灾难、提高模型质量的考虑，需要进行选择。在此，使用卡方检验方法

利用特征与类标签之间的相关性，进行特征选取：

1. /\*特征较多时，使用卡方检验进行特征选择，主要是考察特征与类标签的相关性\*/
2. ChiSqSelector chiSqSelector = **new** ChiSqSelector().setFeaturesCol("vectorFeature").setLabelCol("label").setNumTopFeatures(10).setOutputCol("selectedFeature");

### VectorSlicer

VectorSlicer是一个切片工具，和VectorAssembler相对应，它的作用就是将一个向量切片，选择向量的子集。它提供了两种切片方式。

setIndices(), 按照向量的下标索引切片（从0开始）

setNames()，按照列名切片

import org.apache.spark.ml.attribute.{Attribute, AttributeGroup, NumericAttribute}

import org.apache.spark.ml.feature.VectorSlicer

import org.apache.spark.mllib.linalg.Vectors

import org.apache.spark.sql.Row

import org.apache.spark.sql.types.StructType

val data = Array(Row(Vectors.dense(-2.0, 2.3, 0.0)))

val defaultAttr = NumericAttribute.defaultAttr

val attrs = Array("f1", "f2", "f3").map(defaultAttr.withName)

val attrGroup = new AttributeGroup("userFeatures", attrs.asInstanceOf[Array[Attribute]])

val dataRDD = sc.parallelize(data)

val dataset = sqlContext.createDataFrame(dataRDD, StructType(Array(attrGroup.toStructField())))

val slicer = new VectorSlicer().setInputCol("userFeatures").setOutputCol("features")

slicer.setIndices(Array(1)).setNames(Array("f3"))

// or slicer.setIndices(Array(1, 2)), or slicer.setNames(Array("f2", "f3"))

val output = slicer.transform(dataset)

ouput.show(false)/\*

+--------------+---------+

|userFeatures |features |

+--------------+---------+

|[-2.0,2.3,0.0]|[2.3,0.0]|

+--------------+---------+

\*/

# **SparkML模型选择（超参数调整）与调优**

Mllib支持模型选择，可以使用工具**CrossValidator**和**TrainValidationSplit**，这些工具支持下面的条目：

Estimator：需要调优的算法或者pipeline。

ParamMaps的集合：可供选择的参数，有时称为用来搜索“参数网格”

Evaluator：度量标准来衡量一个拟合Model在测试数据上的表现

在高层面上，这些模型选择工具的作用如下：

他们将输入数据分成单独的训练和测试数据集

对每个(训练，测试)对，他们迭代遍历ParamMaps集合：对于每一个ParamMap，他们使用这些参数调用Estimator的fit，得到拟合Model，并使用Evaluator评估Model的性能。

他们选择由产生的最佳性能参数生成的模型。

Evaluator可以是RegressionEvaluator 用于回归问题中，BinaryClassificationEvaluator 对于二分类，或MulticlassClassificationEvaluator 为多类问题。用于选择最佳值ParamMap的默认度量指标可以被evaluators的setMetricName方法覆盖。

CrossValidator开始的时候会将数据分割成很多测试集和训练集对儿。例如，k=3folds，crossValidator将会产生三组(training,test)数据集对儿，没对都是2/3用来训练，1/3用来测试。为了评估出一个组特殊的paramMap，crossValidator 会计算通过Estimator在三组不同数据集上调用fit产生的3个模型的平均评估指标。

确定最佳ParamMap后，CrossValidator最后使用最佳ParamMap和整个数据集重新拟合Estimator。

**例子**

以下示例演示如何使用CrossValidator从参数网格中进行选择。

请注意，参数网格上的交叉验证非常耗性能的。例如，在下面的例子中，参数网格中hashingTF.numFeatures有三个值，并且lr.regParam两个值，CrossValidator使用了2folds。将会倍增到(3×2)×2=12模型需要训练。在现实的设置中，尝试更多的参数并且使用更多的folds(k=3，k=10是非常常见的)。换句话说使用交叉验证代价是非常大的。然而，它也是一个比较合理的方法，用于选择比启发式手调整更具统计稳健性的参数。

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  
//准备训练数据，格式(id,text,label)  
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")  
  
//配置一个ML pipeline，总共有三个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)//输入label，features，prediction均可采用默认值名称。  
val pipeline = new Pipeline()  
 .setStages(Array(tokenizer, hashingTF, lr))  
  
//用ParamGridBuilder构建一个查询用的参数网格  
//hashingTF.numFeatures有三个值，lr.regParam有两个值，  
//该网格将会有3\*2=6组参数被CrossValidator使用  
val paramGrid = new ParamGridBuilder()  
 .addGrid(hashingTF.numFeatures, Array(10, 100, 1000))  
 .addGrid(lr.regParam, Array(0.1, 0.01))  
 .build()

//这里对将整个PipeLine视为一个Estimator  
//这种方式允许我们联合选择这个Pipeline stages参数  
//一个CrossValidator需要一个Estimator，一组Estimator ParamMaps，一个Evaluator。  
//这个Evaluator是一个BinaryClassificationEvaluator，它默认度量是areaUnderROC  
val cv = new CrossValidator()  
 .setEstimator(pipeline)  
 .setEvaluator(new BinaryClassificationEvaluator)  
 .setEstimatorParamMaps(paramGrid)  
 .setNumFolds(2)  // 生产中使用3+  
  
// 运行交叉验证，选择最佳参数  
val cvModel = cv.fit(training)  
  
//准备测试文档，这些文档是未打标签的  
val test = spark.createDataFrame(Seq(  
 (4L, "spark i j k"),  
 (5L, "l m n"),  
 (6L, "mapreduce spark"),  
 (7L, "apache hadoop")  
)).toDF("id", "text")  
  
//使用训练好的最佳模型，去对测试集进行预测。  
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")  
 }

除了CrossValidator，spark还提供了TrainValidationSplit用于超参数的调整。TrainValidationSplit只对一次参数的每个组合进行一次评估，与CrossValidator的k词调整相对。真就意味着代价相对少了一些，当训练集不是很大的时候，将不会产生一个可靠的结果。

不像CrossValidator，TrainValidationSplit产生一个(training，test)数据集对。通过使用trainRatio参数将数据集分割成两个部分。例如，trainRatio=0.75, TrainValidationSplit将会产生一个训练集和一个测试集，其中75%数据用来训练，25%数据用来验证。

和CrossValidator一样, TrainValidationSplit在最后会使用最佳的参数和整个数据集对Estimator进行拟合。

import org.apache.spark.ml.evaluation.RegressionEvaluator  
import org.apache.spark.ml.regression.LinearRegression  
import org.apache.spark.ml.tuning.{ParamGridBuilder, TrainValidationSplit}  
  
// 准测试数据  
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.  
//使用ParamGridBuilder构建一个parameters网格，用来存储查询参数  
//TrainValidationSplit会尝试所有值的组合使用evaluator来产生一个最佳模型  
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()

// A TrainValidationSplit requires an Estimator, a set of Estimator ParamMaps, and an Evaluator.  
//在这个例子中，Estimator选用简单的线性回归模型  
val trainValidationSplit = new TrainValidationSplit()  
 .setEstimator(lr)  
 .setEvaluator(new RegressionEvaluator)  
 .setEstimatorParamMaps(paramGrid)  
 //80%数据用来训练，20%用来验证  
 .setTrainRatio(0.8)  
  
//运行TrainValidationSplit，选出最佳参数  
val model = trainValidationSplit.fit(training)  
  
//对测试数据进行预测。参数就是刚刚训练的最佳参数。  
model.transform(test)  
 .select("features", "label", "prediction")  
 .show()