**[Spark2.1特征处理:提取/转换/选择](https://vimsky.com/article/2049.html" \o "Spark2.1特征处理:提取/转换/选择)**

本文是Spark2.1官方文档的翻译：Extracting, transforming and selecting features => Spark特征抽取、特征转换、特征选择，全文目录如下：

* [Feature Extractors（特征提取）](https://vimsky.com/article/2049.html" \l "Extracting,transformingandselectingfeatures-FeatureExtractors%EF%BC%88%E7%89%B9%E5%BE%81%E6%8F%90%E5%8F%96%EF%BC%89)
  + [TF-IDF](https://vimsky.com/article/2049.html" \l "Extracting,transformingandselectingfeatures-TF-IDF)
  + [Word2Vec](https://vimsky.com/article/2049.html" \l "Extracting,transformingandselectingfeatures-Word2Vec)
  + [CountVectorizer](https://vimsky.com/article/2049.html" \l "Extracting,transformingandselectingfeatures-CountVectorizer)
* [Feature Transformers（特征变换）](https://vimsky.com/article/2049.html" \l "Extracting,transformingandselectingfeatures-FeatureTransformers%EF%BC%88%E7%89%B9%E5%BE%81%E5%8F%98%E6%8D%A2%EF%BC%89)
  + [Tokenizer（分词器）](https://vimsky.com/article/2049.html" \l "Extracting,transformingandselectingfeatures-Tokenizer%EF%BC%88%E5%88%86%E8%AF%8D%E5%99%A8%EF%BC%89)
  + [StopWordsRemover（停用字清除）](https://vimsky.com/article/2049.html" \l "Extracting,transformingandselectingfeatures-StopWordsRemover%EF%BC%88%E5%81%9C%E7%94%A8%E5%AD%97%E6%B8%85%E9%99%A4%EF%BC%89)
  + [n-gram](https://vimsky.com/article/2049.html" \l "Extracting,transformingandselectingfeatures-n-gram)
  + [Binarizer(二元化方法)](https://vimsky.com/article/2049.html" \l "Extracting,transformingandselectingfeatures-Binarizer(%E4%BA%8C%E5%85%83%E5%8C%96%E6%96%B9%E6%B3%95))
  + [PCA（主成成分分析）](https://vimsky.com/article/2049.html" \l "Extracting,transformingandselectingfeatures-PCA%EF%BC%88%E4%B8%BB%E6%88%90%E6%88%90%E5%88%86%E5%88%86%E6%9E%90%EF%BC%89)
  + [PolynomialExpansion（多项式扩展）](https://vimsky.com/article/2049.html" \l "Extracting,transformingandselectingfeatures-PolynomialExpansion%EF%BC%88%E5%A4%9A%E9%A1%B9%E5%BC%8F%E6%89%A9%E5%B1%95%EF%BC%89)
  + [Discrete Cosine Transform (DCT-离散余弦变换)](https://vimsky.com/article/2049.html" \l "Extracting,transformingandselectingfeatures-DiscreteCosineTransform(DCT-%E7%A6%BB%E6%95%A3%E4%BD%99%E5%BC%A6%E5%8F%98%E6%8D%A2))
  + [StringIndexer（字符串-索引变换）](https://vimsky.com/article/2049.html" \l "Extracting,transformingandselectingfeatures-StringIndexer%EF%BC%88%E5%AD%97%E7%AC%A6%E4%B8%B2-%E7%B4%A2%E5%BC%95%E5%8F%98%E6%8D%A2%EF%BC%89)
  + [IndexToString（索引-字符串变换）](https://vimsky.com/article/2049.html" \l "Extracting,transformingandselectingfeatures-IndexToString%EF%BC%88%E7%B4%A2%E5%BC%95-%E5%AD%97%E7%AC%A6%E4%B8%B2%E5%8F%98%E6%8D%A2%EF%BC%89)
  + [OneHotEncoder（独热编码）](https://vimsky.com/article/2049.html" \l "Extracting,transformingandselectingfeatures-OneHotEncoder%EF%BC%88%E7%8B%AC%E7%83%AD%E7%BC%96%E7%A0%81%EF%BC%89)
  + [VectorIndexer(向量类型索引化)](https://vimsky.com/article/2049.html" \l "Extracting,transformingandselectingfeatures-VectorIndexer(%E5%90%91%E9%87%8F%E7%B1%BB%E5%9E%8B%E7%B4%A2%E5%BC%95%E5%8C%96))
  + [Normalizer(范数p-norm规范化)](https://vimsky.com/article/2049.html" \l "Extracting,transformingandselectingfeatures-Normalizer(%E8%8C%83%E6%95%B0p-norm%E8%A7%84%E8%8C%83%E5%8C%96))
  + [StandardScaler](https://vimsky.com/article/2049.html" \l "Extracting,transformingandselectingfeatures-StandardScaler)
  + [MinMaxScaler（最大-最小规范化）](https://vimsky.com/article/2049.html" \l "Extracting,transformingandselectingfeatures-MinMaxScaler%EF%BC%88%E6%9C%80%E5%A4%A7-%E6%9C%80%E5%B0%8F%E8%A7%84%E8%8C%83%E5%8C%96%EF%BC%89)
  + [MaxAbsScaler(绝对值规范化)](https://vimsky.com/article/2049.html" \l "Extracting,transformingandselectingfeatures-MaxAbsScaler(%E7%BB%9D%E5%AF%B9%E5%80%BC%E8%A7%84%E8%8C%83%E5%8C%96))
  + [Bucketizer（分箱器）](https://vimsky.com/article/2049.html" \l "Extracting,transformingandselectingfeatures-Bucketizer%EF%BC%88%E5%88%86%E7%AE%B1%E5%99%A8%EF%BC%89)
  + [ElementwiseProduct (Hadamard乘积)](https://vimsky.com/article/2049.html" \l "Extracting,transformingandselectingfeatures-ElementwiseProduct(Hadamard%E4%B9%98%E7%A7%AF))
  + [SQLTransformer（SQL变换）](https://vimsky.com/article/2049.html" \l "Extracting,transformingandselectingfeatures-SQLTransformer%EF%BC%88SQL%E5%8F%98%E6%8D%A2%EF%BC%89)
  + [VectorAssembler（特征向量合并）](https://vimsky.com/article/2049.html" \l "Extracting,transformingandselectingfeatures-VectorAssembler%EF%BC%88%E7%89%B9%E5%BE%81%E5%90%91%E9%87%8F%E5%90%88%E5%B9%B6%EF%BC%89)
  + [QuantileDiscretizer（分位数离散化）](https://vimsky.com/article/2049.html" \l "Extracting,transformingandselectingfeatures-QuantileDiscretizer%EF%BC%88%E5%88%86%E4%BD%8D%E6%95%B0%E7%A6%BB%E6%95%A3%E5%8C%96%EF%BC%89)
* Feature Selectors（特征选择）
  + [VectorSlicer（向量选择）](https://vimsky.com/article/2049.html" \l "Extracting,transformingandselectingfeatures-VectorSlicer%EF%BC%88%E5%90%91%E9%87%8F%E9%80%89%E6%8B%A9%EF%BC%89)
  + [RFormula（R模型公式）](https://vimsky.com/article/2049.html" \l "Extracting,transformingandselectingfeatures-RFormula%EF%BC%88R%E6%A8%A1%E5%9E%8B%E5%85%AC%E5%BC%8F%EF%BC%89)
  + [ChiSqSelector（卡方特征选择）](https://vimsky.com/article/2049.html" \l "Extracting,transformingandselectingfeatures-ChiSqSelector%EF%BC%88%E5%8D%A1%E6%96%B9%E7%89%B9%E5%BE%81%E9%80%89%E6%8B%A9%EF%BC%89)

CountVectorizer

CountVectorizer和CountVectorizerModel旨在通过计数将文本文档转换为特征向量。当不存在先验字典时，CountVectorizer可以作为Estimator提取词汇，并生成CountVectorizerModel。该模型产生关于该文档词汇的稀疏表示（稀疏特征向量），这个表示（特征向量）可以传递给其他像 LDA 算法。

在拟合fitting过程中， CountVectorizer将根据语料库中的词频排序选出前vocabSize个词。其中一个配置参数minDF通过指定词汇表中的词语在文档中出现的最小次数 (或词频 if < 1.0) ，影响拟合（fitting）的过程。另一个可配置的二进制toggle参数控制输出向量。如果设置为 true 那么所有非零计数设置为 1。这对于二元型离散概率模型非常有用。

**Examples**

假设我们有如下的DataFrame包含id和texts两列：

id | texts

----|----------

0 | Array("a", "b", "c")

1 | Array("a", "b", "b", "c", "a")

文本中每行都是一个文本类型的数组（字符串）。调用CountVectorizer产生词汇表（a, b, c）的CountVectorizerModel模型，转后后的输出向量如下：

id | texts | vector

----|---------------------------------|---------------

0 | Array("a", "b", "c") | (3,[0,1,2],[1.0,1.0,1.0])

1 | Array("a", "b", "b", "c", "a") | (3,[0,1,2],[2.0,2.0,1.0])

每个向量表示文档词汇表中每个词语出现的次数

**Scala**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | import org.apache.spark.ml.feature.{CountVectorizer, CountVectorizerModel}    val df = spark.createDataFrame(Seq(    (0, Array("a", "b", "c")),    (1, Array("a", "b", "b", "c", "a"))  )).toDF("id", "words")    // fit a CountVectorizerModel from the corpus  val cvModel: CountVectorizerModel = new CountVectorizer()    .setInputCol("words")    .setOutputCol("features")    .setVocabSize(3)    .setMinDF(2)    .fit(df)    // alternatively, define CountVectorizerModel with a-priori vocabulary  val cvm = new CountVectorizerModel(Array("a", "b", "c"))    .setInputCol("words")    .setOutputCol("features")    cvModel.transform(df).select("features").show() |

请阅读英文原文[CountVectorizer Scala 文档](http://spark.apache.org/docs/2.0.2/api/scala/index.html" \l "org.apache.spark.ml.feature.CountVectorizer)和[CountVectorizerModel Scala 文档](http://spark.apache.org/docs/2.0.2/api/scala/index.html" \l "org.apache.spark.ml.feature.CountVectorizerModel)了解相关的 API 的详细信息。

在Spark repo中在"examples/src/main/scala/org/apache/spark/examples/ml/CountVectorizerExample.scala"找到完整的示例代码。

Feature Transformers（特征变换）

Tokenizer（分词器）

[Tokenization](http://en.wikipedia.org/wiki/Lexical_analysis" \l "Tokenization)（文本符号化）是将文本 （如一个句子）拆分成单词的过程。（在Spark ML中）[Tokenizer](http://spark.apache.org/docs/2.0.2/api/scala/index.html" \l "org.apache.spark.ml.feature.Tokenizer)（分词器）提供此功能。下面的示例演示如何将句子拆分为词的序列。

[RegexTokenizer](http://spark.apache.org/docs/2.0.2/api/scala/index.html" \l "org.apache.spark.ml.feature.RegexTokenizer)提供了（更高级的）基于正则表达式 (regex) 匹配的（对句子或文本的）单词拆分。默认情况下，参数"pattern"(默认的正则表达式: "\\s+") 作为分隔符用于拆分输入的文本。或者，用户可以将参数“gaps”设置为 false ，指定正则表达式"pattern"表示"tokens"，而不是分隔符，这样作为分词结果找到的所有匹配项。

**Scala**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | import org.apache.spark.ml.feature.{RegexTokenizer, Tokenizer}    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("label", "sentence")    val tokenizer = new Tokenizer().setInputCol("sentence").setOutputCol("words")  val regexTokenizer = new RegexTokenizer()    .setInputCol("sentence")    .setOutputCol("words")    .setPattern("\\W") // alternatively .setPattern("\\w+").setGaps(false)    val tokenized = tokenizer.transform(sentenceDataFrame)  tokenized.select("words", "label").take(3).foreach(println)  val regexTokenized = regexTokenizer.transform(sentenceDataFrame)  regexTokenized.select("words", "label").take(3).foreach(println) |

请阅读英文原文[Tokenizer Scala docs](http://spark.apache.org/docs/2.0.2/api/scala/index.html" \l "org.apache.spark.ml.feature.Tokenizer) 和[RegexTokenizer Scala docs](http://spark.apache.org/docs/2.0.2/api/scala/index.html" \l "org.apache.spark.ml.feature.RegexTokenizer)了解相关的 API 的详细信息。

|  |
| --- |
| [0,Hi I heard about Spark]  [1,I wish Java could use case classes]  [2,Logistic,regression,models,are,neat]    [WrappedArray(hi, i, heard, about, spark),0]  [WrappedArray(i, wish, java, could, use, case, classes),1]  [WrappedArray(logistic,regression,models,are,neat),2] |

在Spark repo中在"examples/src/main/scala/org/apache/spark/examples/ml/TokenizerExample.scala"找到完整的示例代码。

StopWordsRemover（停用字清除）

[Stop words （停用字）](https://en.wikipedia.org/wiki/Stop_words)是（在文档中）频繁出现，但未携带太多意义的词语，它们不应该参与算法运算。

 StopWordsRemover（的作用是）将输入的字符串 （如分词器[Tokenizer](http://spark.apache.org/docs/2.0.2/ml-features.html" \l "tokenizer)的输出）中的停用字删除（后输出）。停用字表由stopWords参数指定。对于某些语言的默认停止词是通过调用StopWordsRemover.loadDefaultStopWords(language)设置的，可用的选项为"丹麦"，"荷兰语"、"英语"、"芬兰语"，"法国"，"德国"、"匈牙利"、"意大利"、"挪威"、"葡萄牙"、"俄罗斯"、"西班牙"、"瑞典"和"土耳其"。布尔型参数caseSensitive指示是否区分大小写 （默认为否）。

**Examples**

假设有如下DataFrame，有id和raw两列：

|  |
| --- |
| id | raw  ----|----------   0  | [I, saw, the, red, baloon]   1  | [Mary, had, a, little, lamb] |

通过对raw列调用StopWordsRemover，我们可以得到筛选出的结果列如下：

|  |
| --- |
| id | raw                         | filtered  ----|-----------------------------|--------------------   0  | [I, saw, the, red, baloon]  |  [saw, red, baloon]   1  | [Mary, had, a, little, lamb]|[Mary, little, lamb] |

其中，“I”, “the”, “had”以及“a”被移除。

**Scala**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | import org.apache.spark.ml.feature.StopWordsRemover    val remover = new StopWordsRemover()    .setInputCol("raw")    .setOutputCol("filtered")    val dataSet = spark.createDataFrame(Seq(    (0, Seq("I", "saw", "the", "red", "baloon")),    (1, Seq("Mary", "had", "a", "little", "lamb"))  )).toDF("id", "raw")    remover.transform(dataSet).show() |

在Spark repo中路径"examples/src/main/scala/org/apache/spark/examples/ml/StopWordsRemoverExample.scala"可以找到完整的示例代码。

n-gram

一个 [n-gram](https://en.wikipedia.org/wiki/N-gram)是一个长度为n（整数）的字的序列。NGram可以用来将输入特征转换成n-grams。

NGram 的输入为一系列的字符串（例如：[Tokenizer](http://spark.apache.org/docs/2.0.2/ml-features.html" \l "tokenizer)分词器的输出）。参数n表示每个n-gram中单词（terms）的数量。NGram的输出结果是多个n-grams构成的序列，其中，每个n-gram表示被空格分割出来的n个连续的单词。如果输入的字符串少于n个单词，NGram输出为空。

**Scala**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | import org.apache.spark.ml.feature.NGram    val wordDataFrame = spark.createDataFrame(Seq(    (0, Array("Hi", "I", "heard", "about", "Spark")),    (1, Array("I", "wish", "Java", "could", "use", "case", "classes")),    (2, Array("Logistic", "regression", "models", "are", "neat"))  )).toDF("label", "words")    val ngram = new NGram().setInputCol("words").setOutputCol("ngrams").setN(2)  val ngramDataFrame = ngram.transform(wordDataFrame)  ngramDataFrame.take(3).map(\_.getAs[Stream[String]]("ngrams").toList).foreach(println) |

请阅读英文原文 [NGram Scala docs](http://spark.apache.org/docs/2.0.2/api/scala/index.html" \l "org.apache.spark.ml.feature.NGram)了解相关的 API 的详细信息。

在Spark repo中路径"examples/src/main/scala/org/apache/spark/examples/ml/NGramExample.scala"里可以找到完整的示例代码。

Binarizer(二元化方法)

二元化（Binarization）是通过（选定的）阈值将数值化的特征转换成二进制（0/1）特征表示的过程。

Binarizer（ML提供的二元化方法）二元化涉及的参数有inputCol（输入）、outputCol（输出）以及threshold（阀值）。（输入的）特征值大于阀值将映射为1.0，特征值小于等于阀值将映射为0.0。（Binarizer）支持向量（Vector）和双精度（Double）类型的输出

**Scala**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | 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("label", "feature")    val binarizer: Binarizer = new Binarizer()    .setInputCol("feature")    .setOutputCol("binarized\_feature")    .setThreshold(0.5)    val binarizedDataFrame = binarizer.transform(dataFrame)  val binarizedFeatures = binarizedDataFrame.select("binarized\_feature")  binarizedFeatures.collect().foreach(println) |

请阅读英文原文  [Binarizer Scala docs](http://spark.apache.org/docs/2.0.2/api/scala/index.html" \l "org.apache.spark.ml.feature.Binarizer)了解相关的 API 的详细信息。

在Spark repo中路径"examples/src/main/scala/org/apache/spark/examples/ml/BinarizerExample.scala"里可以找到完整的示例代码

PCA（主成成分分析）

主成分分析是一种统计学方法，它使用正交转换从一系列可能线性相关的变量中提取线性无关变量集，提取出的变量集中的元素称为主成分（principal components）。（ML中）PCA 类通过PCA  
方法对项目向量进行降维。下面的示例介绍如何将5维特征向量转换为3维主成分向量。

**Scala**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | import org.apache.spark.ml.feature.PCA  import org.apache.spark.ml.linalg.Vectors    val data = Array(    Vectors.sparse(5, Seq((1, 1.0), (3, 7.0))),    Vectors.dense(2.0, 0.0, 3.0, 4.0, 5.0),    Vectors.dense(4.0, 0.0, 0.0, 6.0, 7.0)  )  val df = spark.createDataFrame(data.map(Tuple1.apply)).toDF("features")  val pca = new PCA()    .setInputCol("features")    .setOutputCol("pcaFeatures")    .setK(3)    .fit(df)  val pcaDF = pca.transform(df)  val result = pcaDF.select("pcaFeatures")  result.show() |

请阅读英文原文  [PCA Scala docs](http://spark.apache.org/docs/2.0.2/api/scala/index.html" \l "org.apache.spark.ml.feature.PCA)了解相关的 API 的详细信息。

在Spark repo中路径"examples/src/main/scala/org/apache/spark/examples/ml/PCAExample.scala"里可以找到完整的示例代码

PolynomialExpansion（多项式扩展）

[多项式扩展（Polynomial expansion）](http://en.wikipedia.org/wiki/Polynomial_expansion)是将n维的原始特征组合扩展到多项式空间的过程。（ML中） [PolynomialExpansion](http://spark.apache.org/docs/2.0.2/api/scala/index.html" \l "org.apache.spark.ml.feature.PolynomialExpansion) 提供多项式扩展的功能。下面的示例会介绍如何将你的特征集拓展到3维多项式空间。

**Scala**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import org.apache.spark.ml.feature.PolynomialExpansion  import org.apache.spark.ml.linalg.Vectors    val data = Array(    Vectors.dense(-2.0, 2.3),    Vectors.dense(0.0, 0.0),    Vectors.dense(0.6, -1.1)  )  val df = spark.createDataFrame(data.map(Tuple1.apply)).toDF("features")  val polynomialExpansion = new PolynomialExpansion()    .setInputCol("features")    .setOutputCol("polyFeatures")    .setDegree(3)  val polyDF = polynomialExpansion.transform(df)  polyDF.select("polyFeatures").take(3).foreach(println) |

请阅读英文原文  [PolynomialExpansion Scala docs](http://spark.apache.org/docs/2.0.2/api/scala/index.html" \l "org.apache.spark.ml.feature.PolynomialExpansion) 了解相关的 API 的详细信息。

在Spark repo中路径"examples/src/main/scala/org/apache/spark/examples/ml/PolynomialExpansionExample.scala"里可以找到完整的示例代码

Discrete Cosine Transform (DCT-离散余弦变换)

The [Discrete Cosine Transform](https://en.wikipedia.org/wiki/Discrete_cosine_transform) transforms a length N   N real-valued sequence in the time domain into another length N   N real-valued sequence in the frequency domain. A [DCT](http://spark.apache.org/docs/2.0.2/api/scala/index.html" \l "org.apache.spark.ml.feature.DCT) class provides this functionality, implementing the [DCT-II](https://en.wikipedia.org/wiki/Discrete_cosine_transform" \l "DCT-II) and scaling the result by  12√12 such that the representing matrix for the transform is unitary. No shift is applied to the transformed sequence (e.g. the 0  0th element of the transformed sequence is the 0  0th DCT coefficient and *not* the N /2  N/2th).

[离散余弦变换（Discrete Cosine Transform）](https://en.wikipedia.org/wiki/Discrete_cosine_transform) 是将时域的N维实数序列转换成频域的N维实数序列的过程（有点类似离散傅里叶变换）。（ML中的）DCT类提供了离散余弦变换[DCT-II](https://en.wikipedia.org/wiki/Discrete_cosine_transform" \l "DCT-II)的功能，将离散余弦变换后结果乘以 12√12 得到一个与时域矩阵长度一致的矩阵。输入序列与输出之间是一一对应的。

**Scala**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | import org.apache.spark.ml.feature.DCT  import org.apache.spark.ml.linalg.Vectors    val data = Seq(    Vectors.dense(0.0, 1.0, -2.0, 3.0),    Vectors.dense(-1.0, 2.0, 4.0, -7.0),    Vectors.dense(14.0, -2.0, -5.0, 1.0))    val df = spark.createDataFrame(data.map(Tuple1.apply)).toDF("features")    val dct = new DCT()    .setInputCol("features")    .setOutputCol("featuresDCT")    .setInverse(false)    val dctDf = dct.transform(df)  dctDf.select("featuresDCT").show(3) |

请阅读英文原文 [DCT Scala docs](http://spark.apache.org/docs/2.0.2/api/scala/index.html" \l "org.apache.spark.ml.feature.DCT)  了解相关的 API 的详细信息。

在Spark repo中路径"examples/src/main/scala/org/apache/spark/examples/ml/DCTExample.scala"里可以找到完整的示例代码

StringIndexer（字符串-索引变换）

StringIndexer（字符串-索引变换）将字符串的（以单词为）标签编码成标签索引（表示）。标签索引序列的取值范围是[0，numLabels（字符串中所有出现的单词去掉重复的词后的总和）]，按照标签出现频率排序，出现最多的标签索引为0。如果输入是数值型，我们先将数值映射到字符串，再对字符串进行索引化。如果下游的pipeline（例如：Estimator或者Transformer）需要用到索引化后的标签序列，则需要将这个pipeline的输入列名字指定为索引化序列的名字。大部分情况下，通过setInputCol设置输入的列名。

**Examples**

假设我们有如下的DataFrame，包含有id和category两列

|  |
| --- |
| id | category  ----|----------   0  | a   1  | b   2  | c   3  | a   4  | a   5  | c |

标签类别（category）是有3种取值的标签：“a”，“b”，“c”。使用StringIndexer通过category进行转换成categoryIndex后可以得到如下结果：

|  |
| --- |
| 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 |

“a”因为出现的次数最多，所以得到为0的索引（index）。“c”得到1的索引，“b”得到2的索引

另外，StringIndexer在转换新数据时提供两种容错机制处理训练中没有出现的标签

* StringIndexer抛出异常错误（默认值）
* 跳过未出现的标签实例。

**Examples**

回顾一下上一个例子，这次我们将继续使用上一个例子训练出来的StringIndexer处理下面的数据集

|  |
| --- |
| id | category  ----|----------   0  | a   1  | b   2  | c   3  | d |

如果没有在StringIndexer里面设置未训练过（unseen）的标签的处理或者设置未“error”，运行时会遇到程序抛出异常。当然，也可以通过设置setHandleInvalid("skip")，得到如下的结果

|  |
| --- |
| id | category | categoryIndex  ----|----------|---------------   0  | a        | 0.0   1  | b        | 2.0   2  | c        | 1.0 |

 注意：输出里面没有出现“d”

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | 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")    val indexer = new StringIndexer()    .setInputCol("category")    .setOutputCol("categoryIndex")    val indexed = indexer.fit(df).transform(df)  indexed.show() |

请阅读英文原文  [StringIndexer Scala docs](http://spark.apache.org/docs/2.0.2/api/scala/index.html" \l "org.apache.spark.ml.feature.StringIndexer)了解相关的 API 的详细信息。

在Spark repo中路径"examples/src/main/scala/org/apache/spark/examples/ml/StringIndexerExample.scala"里可以找到完整的示例代码

IndexToString（索引-字符串变换）

 与StringIndexer对应，IndexToString将索引化标签还原成原始字符串。一个常用的场景是先通过StringIndexer产生索引化标签，然后使用索引化标签进行训练，最后再对预测结果使用IndexToString来获取其原始的标签字符串。

**Examples**

假设我们有如下的DataFrame包含id和categoryIndex两列：

|  |
| --- |
| id | categoryIndex  ----|---------------   0  | 0.0   1  | 2.0   2  | 1.0   3  | 0.0   4  | 0.0   5  | 1.0 |

使用IndexToString我们可以获取其原始的标签字符串如下：

|  |
| --- |
| id | categoryIndex | originalCategory  ----|---------------|-----------------   0  | 0.0           | a   1  | 2.0           | b   2  | 1.0           | c   3  | 0.0           | a   4  | 0.0           | a   5  | 1.0           | c |

**Scala**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import org.apache.spark.ml.feature.{IndexToString, StringIndexer}    val df = spark.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 converter = new IndexToString()    .setInputCol("categoryIndex")    .setOutputCol("originalCategory")    val converted = converter.transform(indexed)  converted.select("id", "originalCategory").show() |

请阅读英文原文  [IndexToString Scala docs](http://spark.apache.org/docs/2.0.2/api/scala/index.html" \l "org.apache.spark.ml.feature.IndexToString) 了解相关的 API 的详细信息。

在Spark repo中路径"examples/src/main/scala/org/apache/spark/examples/ml/IndexToStringExample.scala"里可以找到完整的示例代码

OneHotEncoder（独热编码）

[独热编码（One-hot encoding）](http://en.wikipedia.org/wiki/One-hot)将类别特征映射为二进制向量，其中只有一个有效值（为1，其余为0）。这样在诸如Logistic回归这样需要连续数值值作为特征输入的分类器中也可以使用类别（离散）特征

**Scala**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | import org.apache.spark.ml.feature.{OneHotEncoder, StringIndexer}    val df = spark.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() |

 请阅读英文原文  [OneHotEncoder Scala docs](http://spark.apache.org/docs/2.0.2/api/scala/index.html" \l "org.apache.spark.ml.feature.OneHotEncoder)了解相关的 API 的详细信息。

 在Spark repo中路径"examples/src/main/scala/org/apache/spark/examples/ml/OneHotEncoderExample.scala"里可以找到完整的示例代码

VectorIndexer(向量类型索引化)

VectorIndexer是对数据集特征向量中的类别特征（index categorical features categorical features ，eg：枚举类型）进行编号索引。它能够自动判断那些特征是可以重新编号的类别型，并对他们进行重新编号索引，具体做法如下：

1.获得一个向量类型的输入以及maxCategories参数。

2.基于原始向量数值识别哪些特征需要被类别化：特征向量中某一个特征不重复取值个数小于等于maxCategories则认为是可以重新编号索引的。某一个特征不重复取值个数大于maxCategories，则该特征视为连续值，不会重新编号（不会发生任何改变）

3.对于每一个可编号索引的类别特征重新编号为0～K（K<=maxCategories-1）。

4.对类别特征原始值用编号后的索引替换掉。

索引后的类别特征可以帮助决策树等算法处理类别型特征，提高性能。

在下面的例子中，我们读入一个数据集，然后使用VectorIndexer来决定哪些类别特征需要被作为索引类型处理，将类型特征转换为他们的索引。转换后的数据可以传递给DecisionTreeRegressor之类的算法出来类型特征。

简单理解一下：以C为例，假如一个星期的枚举型的类型enum weekday{ sun = 4,mou =5, tue =6, wed = 7, thu =8, fri = 9, sat =10 };如果需要进行这个特征带入运算，可以将这些枚举数值重新编号为 { sun = 0 , mou =1, tue =2, wed = 3, thu =4, fri = 5, sat =6 }，通常是出现次数越多的枚举，编号越小（从0开始）

**Scala**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | import org.apache.spark.ml.feature.VectorIndexer    //val data = spark.read.format("libsvm").load("data/mllib/sample\_libsvm\_data.txt")  val data1 = Seq(        Vectors.dense(2, 5, 7, 3),        Vectors.dense(4, 2, 4, 7),        Vectors.dense(5, 3, 4, 7),        Vectors.dense(6, 2, 4, 7),        Vectors.dense(7, 2, 4, 7),        Vectors.dense(8, 2, 5, 1))    val data = spark.createDataFrame(data1.map(Tuple1.apply)).toDF("features")    val indexer = new VectorIndexer()    .setInputCol("features")    .setOutputCol("indexed")    .setMaxCategories(10)    val indexerModel = indexer.fit(data)    val categoricalFeatures: Set[Int] = indexerModel.categoryMaps.keys.toSet  println(s"Chose ${categoricalFeatures.size} categorical features: " +    categoricalFeatures.mkString(", "))    // Create new column "indexed" with categorical values transformed to indices  val indexedData = indexerModel.transform(data)  indexedData.show() |

请阅读英文原文  [VectorIndexer Scala docs](http://spark.apache.org/docs/2.0.2/api/scala/index.html" \l "org.apache.spark.ml.feature.VectorIndexer) 了解相关的 API 的详细信息。

在Spark repo中路径"examples/src/main/scala/org/apache/spark/examples/ml/VectorIndexerExample.scala"里可以找到完整的示例代码

Normalizer(范数p-norm规范化)

Normalizer是一个转换器，它可以将一组特征向量（通过计算p-范数）规范化。参数为p（默认值：2）来指定规范化中使用的p-norm。规范化操作可以使输入数据标准化，对后期机器学习算法的结果也有更好的表现。

下面的例子展示如何读入一个libsvm格式的数据，然后将每一行转换为 L2 以及 L∞ 形式。

**Scala**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | import org.apache.spark.ml.feature.Normalizer    val dataFrame = spark.read.format("libsvm").load("data/mllib/sample\_multiclass\_classification\_data.txt")    // Normalize each Vector using $L^1$ norm.  val normalizer = new Normalizer()    .setInputCol("features")    .setOutputCol("normFeatures")    .setP(1.0)    val l1NormData = normalizer.transform(dataFrame)  l1NormData.show()    val l2NormData = normalizer.transform(dataFrame,  normalizer.p -> 2)  l2NormData.show(10, false)    // Normalize each Vector using $L^\infty$ norm.  val lInfNormData = normalizer.transform(dataFrame, normalizer.p -> Double.PositiveInfinity)  lInfNormData.show() |

请阅读英文原文  [Normalizer Scala docs](http://spark.apache.org/docs/2.0.2/api/scala/index.html" \l "org.apache.spark.ml.feature.Normalizer) 了解相关的 API 的详细信息。

在Spark repo中路径"examples/src/main/scala/org/apache/spark/examples/ml/NormalizerExample.scala"里可以找到完整的示例代码

StandardScaler

StandardScaler（z-score规范化：零均值标准化）可以将输入的一组Vector特征向量规范化（标准化），使其有统一的的标准差（均方差？）以及均值为0。它需要如下参数：

1. withStd：默认值为真，将数据缩放到统一标准差方式。

2. withMean：默认为假。将均值为0。该方法将产出一个稠密的输出向量，所以不适用于稀疏向量。

StandardScaler是一个Estimator，它可以通过拟合（fit）数据集产生一个StandardScalerModel，用来统计汇总。StandardScalerModel可以用来将向量转换至统一的标准差以及（或者）零均值特征。

注意如果特征的标准差为零，则该特征在向量中返回的默认值为0.0。

下面的示例展示如果读入一个libsvm形式的数据以及返回有统一标准差的标准化特征。

**Scala**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import org.apache.spark.ml.feature.StandardScaler    val dataFrame = spark.read.format("libsvm").load("data/mllib/sample\_libsvm\_data.txt")    val scaler = new StandardScaler()    .setInputCol("features")    .setOutputCol("scaledFeatures")    .setWithStd(true)    .setWithMean(false)    // Compute summary statistics by fitting the StandardScaler.  val scalerModel = scaler.fit(dataFrame)    // Normalize each feature to have unit standard deviation.  val scaledData = scalerModel.transform(dataFrame)  scaledData.show() |

请阅读英文原文[StandardScaler Scala docs](http://spark.apache.org/docs/2.0.2/api/scala/index.html" \l "org.apache.spark.ml.feature.StandardScaler)了解相关的 API 的详细信息。

在Spark repo中路径"examples/src/main/scala/org/apache/spark/examples/ml/StandardScalerExample.scala"里可以找到完整的示例代码

MinMaxScaler（**最大-最小规范化**）

MinMaxScaler将所有特征向量线性变换到指定范围（最小-最大值）之间（归一化到[min, max]，通常为[0,1]）。它的参数有：

1. min：默认为0.0，为转换后所有特征的下边界。

2. max：默认为1.0，为转换后所有特征的上边界。

MinMaxScaler根据数据集的汇总统计产生一个MinMaxScalerModel。在计算时，该模型将特征向量一个一个分开计算并转换到指定的范围内的。

对于特征E来说，调整后的特征值如下：

Rescaled(ei)=ei−EminEmax−Emin∗(max−min)+minRescaled(ei)=ei−EminEmax−Emin∗(max−min)+min

如果 Emax=EminEmax=Emin ，则 Rescaled=0.5∗(max−min)Rescaled=0.5∗(max−min) 。

注意：**（1）最大最小值可能受到离群值的左右。（2）零值可能会转换成一个非零值，因此稀疏矩阵将变成一个稠密矩阵。**

下面的示例展示如何读入一个libsvm形式的数据以及调整其特征值到[0,1]之间。

**Scala**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import org.apache.spark.ml.feature.MinMaxScaler    val dataFrame = spark.read.format("libsvm").load("data/mllib/sample\_libsvm\_data.txt")    val scaler = new MinMaxScaler()    .setInputCol("features")    .setOutputCol("scaledFeatures")    // Compute summary statistics and generate MinMaxScalerModel  val scalerModel = scaler.fit(dataFrame)    // rescale each feature to range [min, max].  val scaledData = scalerModel.transform(dataFrame)  scaledData.show() |

请阅读英文原文[MinMaxScaler Scala docs](http://spark.apache.org/docs/2.0.2/api/scala/index.html" \l "org.apache.spark.ml.feature.MinMaxScaler) 和 [MinMaxScalerModel Scala docs](http://spark.apache.org/docs/2.0.2/api/scala/index.html" \l "org.apache.spark.ml.feature.MinMaxScalerModel) 了解相关的 API 的详细信息。

在Spark repo中路径"examples/src/main/scala/org/apache/spark/examples/ml/MinMaxScalerExample.scala"里可以找到完整的示例代码

MaxAbsScaler(绝对值规范化)

MaxAbsScaler使用每个特征的最大值的绝对值将输入向量的特征值（各特征值除以最大绝对值）转换到[-1,1]之间。因为它不会转移／集中数据，所以不会破坏数据的稀疏性。

下面的示例展示如果读入一个libsvm形式的数据以及调整其特征值到[-1,1]之间。

**Scala**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | import org.apache.spark.ml.feature.MaxAbsScaler    val dataFrame = spark.read.format("libsvm").load("data/mllib/sample\_libsvm\_data.txt")  val scaler = new MaxAbsScaler()    .setInputCol("features")    .setOutputCol("scaledFeatures")    // Compute summary statistics and generate MaxAbsScalerModel  val scalerModel = scaler.fit(dataFrame)    // rescale each feature to range [-1, 1]  val scaledData = scalerModel.transform(dataFrame)  scaledData.show() |

请阅读英文原文 [MaxAbsScaler Scala docs](http://spark.apache.org/docs/2.0.2/api/scala/index.html" \l "org.apache.spark.ml.feature.MaxAbsScaler) 和 [MaxAbsScalerModel Scala docs](http://spark.apache.org/docs/2.0.2/api/scala/index.html" \l "org.apache.spark.ml.feature.MaxAbsScalerModel) 了解相关的 API 的详细信息。

在Spark repo中路径"examples/src/main/scala/org/apache/spark/examples/ml/MaxAbsScalerExample.scala"里可以找到完整的示例代码

Bucketizer（**分箱器**）

Bucketizer将一列连续的特征转换为（离散的）特征区间，区间由用户指定。参数如下：

splits（分箱数）：分箱数为n+1时，将产生n个区间。除了最后一个区间外，每个区间范围［x,y］由分箱的x，y决定。分箱必须是严格递增的。分箱（区间）见在分箱（区间）指定外的值将被归为错误。两个分裂的例子为Array(Double.NegativeInfinity, 0.0, 1.0, Double.PositiveInfinity)以及Array(0.0, 1.0, 2.0)。

注意：

* 当不确定分裂的上下边界时，应当添加Double.NegativeInfinity和Double.PositiveInfinity以免越界。
* 分箱区间必须严格递增，例如： s0 < s1 < s2 < ... < sn

下面将展示Bucketizer的使用方法。

**Scala**

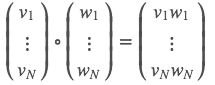
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import org.apache.spark.ml.feature.Bucketizer    val splits = Array(Double.NegativeInfinity, -0.5, 0.0, 0.5, Double.PositiveInfinity)    val data = Array(-0.5, -0.3, 0.0, 0.2)  val dataFrame = spark.createDataFrame(data.map(Tuple1.apply)).toDF("features")    val bucketizer = new Bucketizer()    .setInputCol("features")    .setOutputCol("bucketedFeatures")    .setSplits(splits)    // Transform original data into its bucket index.  val bucketedData = bucketizer.transform(dataFrame)  bucketedData.show() |

请阅读英文原文 [Bucketizer Scala docs](http://spark.apache.org/docs/2.0.2/api/scala/index.html" \l "org.apache.spark.ml.feature.Bucketizer)了解相关的 API 的详细信息。

在Spark repo中路径"examples/src/main/scala/org/apache/spark/examples/ml/BucketizerExample.scala"里可以找到完整的示例代码

ElementwiseProduct (Hadamard乘积)

ElementwiseProduct对输入向量的每个元素乘以一个权重（weight），即对输入向量每个元素逐个进行放缩。对输入向量v 和变换向量scalingVec 使用Hadamard product(阿达玛积)进行变换，最终产生一个新的向量。用向量 w 表示 scalingVec ，则Hadamard product可以表示为



下面例子展示如何通过转换向量的值来调整向量。

**Scala**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import org.apache.spark.ml.feature.ElementwiseProduct  import org.apache.spark.ml.linalg.Vectors    // Create some vector data; also works for sparse vectors  val dataFrame = spark.createDataFrame(Seq(    ("a", Vectors.dense(1.0, 2.0, 3.0)),    ("b", Vectors.dense(4.0, 5.0, 6.0)))).toDF("id", "vector")    val transformingVector = Vectors.dense(0.0, 1.0, 2.0)  val transformer = new ElementwiseProduct()    .setScalingVec(transformingVector)    .setInputCol("vector")    .setOutputCol("transformedVector")    // Batch transform the vectors to create new column:  transformer.transform(dataFrame).show() |

请阅读英文原文 [ElementwiseProduct Scala docs](http://spark.apache.org/docs/2.0.2/api/scala/index.html" \l "org.apache.spark.ml.feature.ElementwiseProduct) 了解相关的 API 的详细信息。

在Spark repo中路径"examples/src/main/scala/org/apache/spark/examples/ml/ElementwiseProductExample.scala"里可以找到完整的示例代码

SQLTransformer（SQL变换）

SQLTransformer用来转换由SQL定义的陈述。目前仅支持SQL语法如"SELECT ... FROM \_\_THIS\_\_ ..."，其中"\_\_THIS\_\_"代表输入数据的基础表。选择语句指定输出中展示的字段、元素和表达式，支持Spark SQL中的所有选择语句。用户可以基于选择结果使用Spark SQL建立方程或者用户自定义函数（UDFs）。SQLTransformer支持语法示例如下：

* SELECT a, a + b AS a\_b FROM \_\_THIS\_\_
* SELECT a, SQRT(b) AS b\_sqrt FROM \_\_THIS\_\_ where a > 5
* SELECT a, b, SUM(c) AS c\_sum FROM \_\_THIS\_\_ GROUP BY a, b

**Examples**

假设我们有如下DataFrame包含id，v1，v2列：

|  |
| --- |
| id |  v1 |  v2  ----|-----|-----   0  | 1.0 | 3.0   2  | 2.0 | 5.0 |

使用SQLTransformer语句"SELECT \*, (v1 + v2) AS v3, (v1 \* v2) AS v4 FROM \_\_THIS\_\_"转换后得到输出如下：

|  |
| --- |
| id |  v1 |  v2 |  v3 |  v4  ----|-----|-----|-----|-----   0  | 1.0 | 3.0 | 4.0 | 3.0   2  | 2.0 | 5.0 | 7.0 |10.0 |

**Scala**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | import org.apache.spark.ml.feature.SQLTransformer    val df = spark.createDataFrame(    Seq((0, 1.0, 3.0), (2, 2.0, 5.0))).toDF("id", "v1", "v2")    val sqlTrans = new SQLTransformer().setStatement(    "SELECT \*, (v1 + v2) AS v3, (v1 \* v2) AS v4 FROM \_\_THIS\_\_")    sqlTrans.transform(df).show() |

请阅读英文原文 [SQLTransformer Scala docs](http://spark.apache.org/docs/2.0.2/api/scala/index.html" \l "org.apache.spark.ml.feature.SQLTransformer)  了解相关的 API 的详细信息。

在Spark repo中路径"examples/src/main/scala/org/apache/spark/examples/ml/SQLTransformerExample.scala"里可以找到完整的示例代码

VectorAssembler（特征向量合并）

VectorAssembler是一个转换器，它将给定的若干列合并为单列向量。它可以将原始特征和一系列通过其他转换器变换得到的特征合并为单一的特征向量，来训练如逻辑回归和决策树等机器学习算法。VectorAssembler可接受的输入列类型：数值型、布尔型、向量型。输入列的值将按指定顺序依次添加到一个新向量中。  
**Examples**

假设我们有如下DataFrame包含id, hour, mobile, userFeatures以及clicked列：

|  |
| --- |
| id | hour | mobile | userFeatures     | clicked  ----|------|--------|------------------|---------   0  | 18   | 1.0    | [0.0, 10.0, 0.5] | 1.0 |

userFeatures列中含有3个用户特征，我们将hour, mobile以及userFeatures合并为一个新的单一特征向量。将VectorAssembler的输入指定为hour, mobile以及userFeatures，输出指定为features，通过转换我们将得到以下结果：

|  |
| --- |
| 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] |

**Scala**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | import org.apache.spark.ml.feature.VectorAssembler  import org.apache.spark.ml.linalg.Vectors    val dataset = spark.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)  println(output.select("features", "clicked").first()) |

请阅读英文原文  [VectorAssembler Scala docs](http://spark.apache.org/docs/2.0.2/api/scala/index.html" \l "org.apache.spark.ml.feature.VectorAssembler)了解相关的 API 的详细信息。

在Spark repo中路径"examples/src/main/scala/org/apache/spark/examples/ml/VectorAssemblerExample.scala"里可以找到完整的示例代码

QuantileDiscretizer（分位数离散化）

QuantileDiscretizer（分位数离散化）将一列连续型的特征向量转换成分类型数据向量。分级的数量由numBuckets参数决定。分级的范围由渐进算法（[approxQuantile](http://spark.apache.org/docs/2.0.2/api/scala/index.html" \l "org.apache.spark.sql.DataFrameStatFunctions) ）决定。

渐进的精度由relativeError参数决定。当relativeError设置为0时，将会计算精确的分位点（计算代价较高）。分级的上下边界为负无穷（-Infinity）到正无穷（+Infinity），覆盖所有的实数值。

**Examples**

假设我们有如下DataFrame包含id, hour：

|  |
| --- |
| id | hour  ----|------   0  | 18.0  ----|------   1  | 19.0  ----|------   2  | 8.0  ----|------   3  | 5.0  ----|------   4  | 2.2 |

hour是一个Double类型的连续特征，我们希望将它转换成分级特征。将参数numBuckets设置为3，可以如下分级特征

|  |
| --- |
| id | hour | result  ----|------|------   0  | 18.0 | 2.0  ----|------|------   1  | 19.0 | 2.0  ----|------|------   2  | 8.0  | 1.0  ----|------|------   3  | 5.0  | 1.0  ----|------|------   4  | 2.2  | 0.0 |

**Scala**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | import org.apache.spark.ml.feature.QuantileDiscretizer    val data = Array((0, 18.0), (1, 19.0), (2, 8.0), (3, 5.0), (4, 2.2))  var df = spark.createDataFrame(data).toDF("id", "hour")    val discretizer = new QuantileDiscretizer()    .setInputCol("hour")    .setOutputCol("result")    .setNumBuckets(3)    val result = discretizer.fit(df).transform(df)  result.show() |

请阅读英文原文  [QuantileDiscretizer Scala docs](http://spark.apache.org/docs/2.0.2/api/scala/index.html" \l "org.apache.spark.ml.feature.QuantileDiscretizer)了解相关的 API 的详细信息。

在Spark repo中路径"examples/src/main/scala/org/apache/spark/examples/ml/QuantileDiscretizerExample.scala"里可以找到完整的示例代码

Feature Selectors（特征选择）

VectorSlicer（向量选择）

 VectorSlicer是一个将输入特征向量转换为输出原始特征向量子集的转换器。VectorSlicer对特征提取非常有帮助。

VectorSlicer接收带有特定索引的向量列，通过对这些索引的值进行筛选得到新的向量集。可接受如下两种索引

1. 整数索引，setIndices()。
2. 字符串索引，setNames()，此类要求向量列有*AttributeGroup*，因为该工具根据*Attribute*来匹配属性字段。

可以指定整数或者字符串类型。另外，也可以同时使用整数索引和字符串名字。不允许使用重复的特征，所以所选的索引或者名字必须是独一的。注意如果使用名字特征，当遇到空值的时候将会抛异常。

输出将会首先按照所选的数字索引排序（按输入顺序），其次按名字排序（按输入顺序）。

**Examples**

假设我们有一个DataFrame含有userFeatures列：

|  |
| --- |
| userFeatures  ------------------   [0.0, 10.0, 0.5] |

userFeatures是一个包含3个用户特征的特征向量。假设userFeatures的第一列全为0，我们希望删除它并且只选择后两项。我们可以通过索引setIndices(1, 2)来选择后两项并产生一个新的features列：

|  |
| --- |
| userFeatures     | features  ------------------|-----------------------------   [0.0, 10.0, 0.5] | [10.0, 0.5] |

假设我们还有如同["f1", "f2", "f3"]的属性，可以通过名字setNames("f2", "f3")的形式来选择：

|  |
| --- |
| userFeatures     | features  ------------------|-----------------------------   [0.0, 10.0, 0.5] | [10.0, 0.5]   ["f1", "f2", "f3"] | ["f2", "f3"] |

**Scala**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.util.Arrays    import org.apache.spark.ml.attribute.{Attribute, AttributeGroup, NumericAttribute}  import org.apache.spark.ml.feature.VectorSlicer  import org.apache.spark.ml.linalg.Vectors  import org.apache.spark.sql.Row  import org.apache.spark.sql.types.StructType    val data = Arrays.asList(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 dataset = spark.createDataFrame(data, 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)  println(output.select("userFeatures", "features").first()) |

 请阅读英文原文  [VectorSlicer Scala docs](http://spark.apache.org/docs/2.0.2/api/scala/index.html" \l "org.apache.spark.ml.feature.VectorSlicer)了解相关的 API 的详细信息。

在Spark repo中路径"examples/src/main/scala/org/apache/spark/examples/ml/VectorSlicerExample.scala"里可以找到完整的示例代码

RFormula（R模型公式）

 RFormula通过R模型公式（[R model formula](https://stat.ethz.ch/R-manual/R-devel/library/stats/html/formula.html)）来将数据中的字段转换成特征值。ML只支持R操作中的部分操作，包括‘~’, ‘.’, ‘:’, ‘+’以及‘-‘，基本操作如下：

* ~分隔目标和对象
* +合并对象，“+ 0”意味着删除空格
* :交互（数值相乘，类别二元化）
* . 除了目标外的全部列

假设有双精度的a和b两列，RFormula的使用用例如下

* y ~ a + b表示模型y ~ w0 + w1 \* a +w2 \* b其中w0为截距，w1和w2为相关系数。
* y ~a + b + a:b – 1表示模型y ~ w1\* a + w2 \* b + w3 \* a \* b，其中w1，w2，w3是相关系数。

RFormula产生一个特征向量和一个double或者字符串标签列（label）。就如R中使用formulas一样，字符型的输入将转换成one-hot编码，数字输入转换成双精度。如果类别列是字符串类型，它将通过StringIndexer转换为double类型索引。如果标签列不存在，则formulas输出中将通过特定的响应变量创造一个标签列。

**Examples**

假设我们有一个DataFrame含有id,country, hour和clicked四列：

|  |
| --- |
| id | country | hour | clicked  ---|---------|------|---------   7 | "US"    | 18   | 1.0   8 | "CA"    | 12   | 0.0   9 | "NZ"    | 15   | 0.0 |

如果使用RFormula公式clicked ~ country + hour，则表明我们希望基于country 和hour预测clicked，通过转换我们可以得到如下DataFrame：

|  |
| --- |
| id | country | hour | clicked | features         | label  ---|---------|------|---------|------------------|-------   7 | "US"    | 18   | 1.0     | [0.0, 0.0, 18.0] | 1.0   8 | "CA"    | 12   | 0.0     | [0.0, 1.0, 12.0] | 0.0   9 | "NZ"    | 15   | 0.0     | [1.0, 0.0, 15.0] | 0.0 |

**Scala**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | import org.apache.spark.ml.feature.RFormula    val dataset = spark.createDataFrame(Seq(    (7, "US", 18, 1.0),    (8, "CA", 12, 0.0),    (9, "NZ", 15, 0.0)  )).toDF("id", "country", "hour", "clicked")  val formula = new RFormula()    .setFormula("clicked ~ country + hour")    .setFeaturesCol("features")    .setLabelCol("label")  val output = formula.fit(dataset).transform(dataset)  output.select("features", "label").show() |

请阅读英文原文  [RFormula Scala docs](http://spark.apache.org/docs/2.0.2/api/scala/index.html" \l "org.apache.spark.ml.feature.RFormula)了解相关的 API 的详细信息。

在Spark repo中路径"examples/src/main/scala/org/apache/spark/examples/ml/RFormulaExample.scala"里可以找到完整的示例代码