第四课 Spark ML特征的提取、转换和选择

目录

特征的提取及ML实现详解

org.apache.spark.ml.feature hide focus

- Binarizer
- BucketedRandomProjectionLSH
- BucketedRandomProjectionLSHModel
- Bucketizer
- ChiSqSelector
- ChiSqSelectorModel
- CountVectorizer
- CountVectorizerModel
- DCT
- ElementwiseProduct
- HashingTF
- IDF
- IDFModel
- Imputer
- ImputerModel
- IndexToString
- Interaction
- LabeledPoint
- MaxAbsScaler
- MaxAbsScalerModel
- MinHashLSH
- MinHashLSHModel
- MinMaxScaler
- MinMaxScalerModel
- NGram
- Normalizer
- OneHotEncoder
- PCA
- PCAModel
- PolynomialExpansion
- QuantileDiscretizer
- RegexTokenizer
- RFormula
- RFormulaModel
- SQLTransformer
- StandardScaler
- StandardScalerModel
- StopWordsRemover
- StringIndexer
- StringIndexerModel
- Tokenizer
- VectorAssembler
- VectorIndexer
- VectorIndexerModel
- VectorSlicer
- Word2Vec
- Word2VecModel

目录

特征的提取

特征的提取及ML实现详解

- **1、TF-IDF（词频-逆向文档频率）**
- 词频（Term Frequency）- 逆向文档频率（Inverse Document Frequency）是一种在文本挖掘中广泛使用的特征向量化方法，以反映一个单词在语料库中的重要性。定义：*t* 表示由一个单词，*d* 表示一个文档，*D* 表示语料库（corpus），词频 *TF(t,d)* 表示某一个给定的单词 *t* 出现在文档 *d* 中的次数， 而文档频率 *DF(t,D)* 表示包含单词 *t* 的文档次数。如果我们只使用词频 *TF* 来衡量重要性，则很容易过分强调出现频率过高并且文档包含少许信息的单词，例如，'a'，'the'，和 'of'。如果一个单词在整个语料库中出现的非常频繁，这意味着它并没有携带特定文档的某些特殊信息（换句话说，该单词对整个文档的重要程度低）。

特征的提取及ML实现详解

- **1、TF-IDF（词频-逆向文档频率）**
- 逆向文档频率是一个数字量度，表示一个单词提供了多少信息：

$$IDF(t, D) = \log \frac{|D| + 1}{DF(t, D) + 1},$$

- 其中，$|D|$ 是在语料库中文档总数。由于使用对数，所以如果一个单词出现在所有的文件，其IDF值变为0。注意，应用平滑项以避免在语料库之外的项除以零（为了防止分母为0，分母需要加1）。因此，TF-IDF测量只是TF和IDF的产物：（对TF-IDF定义为TF和IDF的乘积）

$$TFIDF(t, d, D) = TF(t, d) \cdot IDF(t, D).$$

- 关于词频TF和文档频率DF的定义有多种形式。在MLlib，我们分离TF和IDF，使其灵活。

特征的提取及ML实现详解

- **1、TF-IDF（词频-逆向文档频率）**
- **TF（词频Term Frequency）**：HashingTF与CountVectorizer用于生成词频TF向量。
- HashingTF是一个特征词集的转换器（Transformer），它可以将这些集合转换成固定长度的特征向量。HashingTF利用hashing trick，原始特征通过应用哈希函数映射到索引中。然后根据映射的索引计算词频。这种方法避免了计算全局特征词对索引映射的需要，这对于大型语料库来说可能是昂贵的，但是它具有潜在的哈希冲突，其中不同的原始特征可以在散列之后变成相同的特征词。为了减少碰撞的机会，我们可以增加目标特征维度，即哈希表的桶数。由于使用简单的模数将散列函数转换为列索引，建议使用两个幂作为特征维，否则不会将特征均匀地映射到列。默认功能维度为2^18=262144。可选的二进制切换参数控制词频计数。当设置为true时，所有非零频率计数设置为1。这对于模拟二进制而不是整数的离散概率模型尤其有用。

特征的提取及ML实现详解

- **1、TF-IDF（词频-逆向文档频率）**
- **IDF（逆向文档频率）**：IDF是一个适合数据集并生成IDFModel的评估器（Estimator），IDFModel获取特征向量（通常由HashingTF或CountVectorizer创建）并缩放每列。直观地说，它下调了在语料库中频繁出现的列。

- spark.ml不提供文本分割的工具。我们推荐用户参考Stanford NLP Group 和 scalanlp/chalk.

特征的提取及ML实现详解

```
    // 1 样本准备
    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")

    sentenceData.show
```

特征的提取及ML实现详解

```
    // 2 tokenizer、hashingTF、idf
    val tokenizer = new
Tokenizer().setInputCol("sentence").setOutputCol("words")
    val wordsData = tokenizer.transform(sentenceData)
    val hashingTF = new HashingTF()
      .setInputCol("words").setOutputCol("rawFeatures").setNumFeatures(20)
    val featurizedData = hashingTF.transform(wordsData)
    // 通过CountVectorizer也可以获得词频向量
    val idf = new IDF().setInputCol("rawFeatures").setOutputCol("features")
    val idfModel = idf.fit(featurizedData)
    // 3 测试及结果展示
    val rescaledData = idfModel.transform(featurizedData)
    rescaledData.select("label", "features").show()
```

特征的提取及ML实现详解

```
// 1 样本准备
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")

sentenceData.show
```

```
sentenceData: org.apache.spark.sql.DataFrame = [label: double, sentence: string]
+-----+--------------------+
|label|            sentence|
+-----+--------------------+
|  0.0|Hi I heard about ...|
|  0.0|I wish Java could...|
|  1.0|Logistic regressi...|
+-----+--------------------+
```

特征的提取及ML实现详解

```
// 2 tokenizer、hashingTF、idf
val tokenizer = new Tokenizer().setInputCol("sentence").setOutputCol("words")
val wordsData = tokenizer.transform(sentenceData)

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

val featurizedData = hashingTF.transform(wordsData)
// 通过CountVectorizer也可以获得词频向量

val idf = new IDF().setInputCol("rawFeatures").setOutputCol("features")
val idfModel = idf.fit(featurizedData)
```

```
tokenizer: org.apache.spark.ml.feature.Tokenizer = tok_91228706f512
wordsData: org.apache.spark.sql.DataFrame = [label: double, sentence: string ... 1 more field]
hashingTF: org.apache.spark.ml.feature.HashingTF = hashingTF_0b9f907f797a
featurizedData: org.apache.spark.sql.DataFrame = [label: double, sentence: string ... 2 more fields]
idf: org.apache.spark.ml.feature.IDF = idf_3fb1b5507b24
idfModel: org.apache.spark.ml.feature.IDFModel = idf_3fb1b5507b24
```

特征的提取及ML实现详解

```
// 3 测试及结果展示
val rescaledData = idfModel.transform(featurizedData)
rescaledData.select("label", "features").show(false)
```

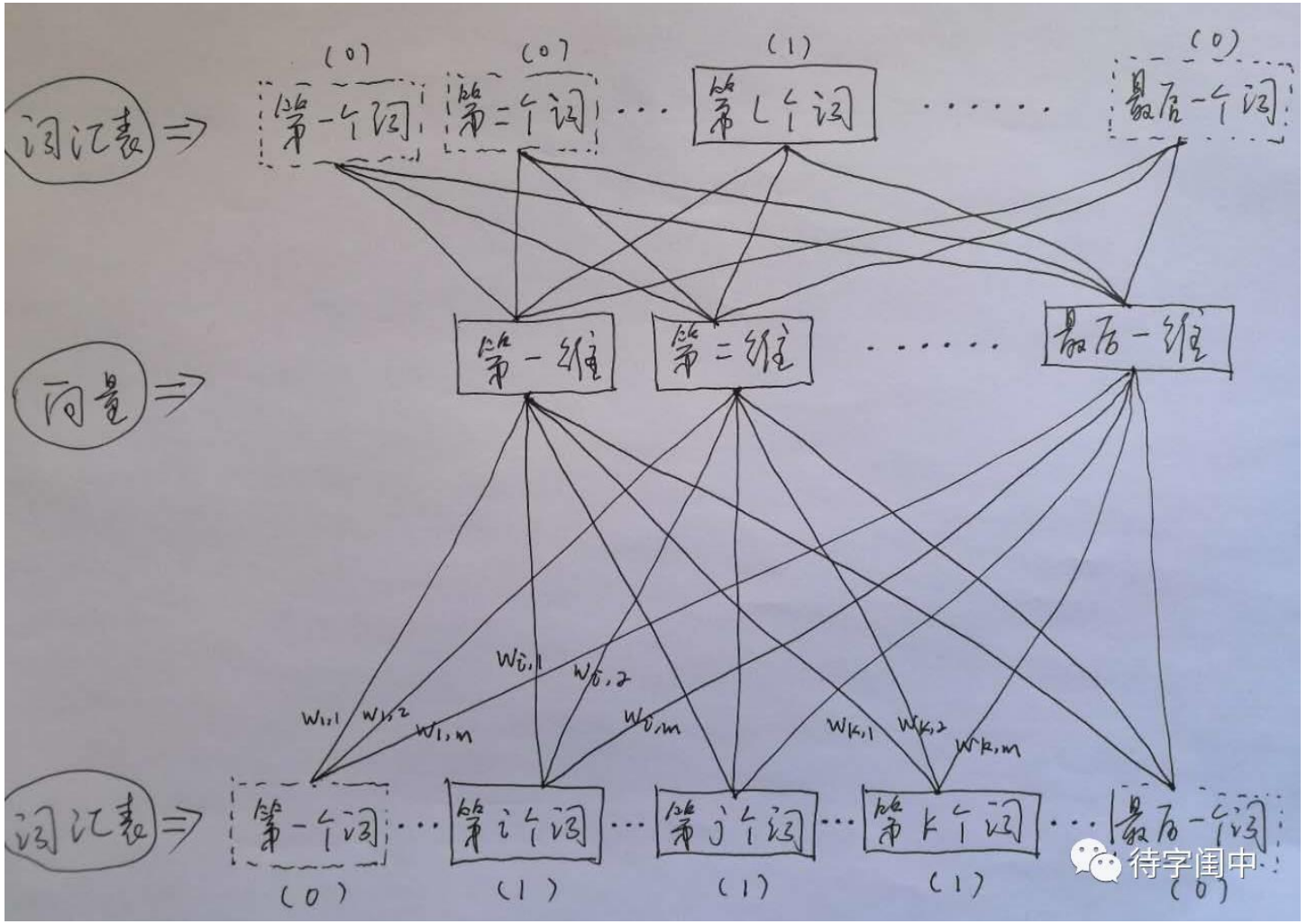
```
rescaledData: org.apache.spark.sql.DataFrame = [label: double, sentence: string ... 3 more fields]
+-----+-----------------------------------------------------------------------------------------------------------------------------------------+
|label|features                                                                                                                                 |
+-----+-----------------------------------------------------------------------------------------------------------------------------------------+
|0.0  |(20,[0,5,9,17],[0.6931471805599453,0.6931471805599453,0.28768207245178085,1.3862943611198906])                                           |
|0.0  |(20,[2,7,9,13,15],[0.6931471805599453,0.6931471805599453,0.8630462173553426,0.28768207245178085,0.28768207245178085])                    |
|1.0  |(20,[4,6,13,15,18],[0.6931471805599453,0.6931471805599453,0.28768207245178085,0.28768207245178085,0.6931471805599453])                   |
+-----+-----------------------------------------------------------------------------------------------------------------------------------------+
```

特征的提取及ML实现详解

- **2、Word2Vec**
- Word2Vec是一个Estimator(评估器)，它采用表示文档的单词序列，并训练一个Word2VecModel。 该模型将每个单词映射到一个唯一的固定大小向量。 Word2VecModel使用文档中所有单词的平均值将每个文档转换为向量; 该向量然后可用作预测，文档相似性计算等功能。

特征的提取及ML实现详解

- 2、Word2Vec



特征的提取及ML实现详解

```
    // 1 样本准备
    val documentDF = spark.createDataFrame(Seq(
      "Hi I heard about Spark".split(" "),
      "I wish Java could use case classes".split(" "),
      "Logistic regression models are neat".split("
")).map(Tuple1.apply)).toDF("text")

    documentDF.show
```

特征的提取及ML实现详解

```
// 2 word2Vec
val word2Vec = new Word2Vec()
  .setInputCol("text")
  .setOutputCol("result")
  .setVectorSize(3)
  .setMinCount(0)
val model = word2Vec.fit(documentDF)

// 3 测试及结果展示
val result = model.transform(documentDF)
result.collect().foreach {
  case Row(text: Seq[_], features: Vector) =>
    println(s"Text: [${text.mkString(", ")}] => \nVector: $features\n")
}
```

特征的提取及ML实现详解

```
// 1 样本准备
val documentDF = spark.createDataFrame(Seq(
  "Hi I heard about Spark".split(" "),
  "I wish Java could use case classes".split(" "),
  "Logistic regression models are neat".split(" ")).map(Tuple1.apply)).toDF("text")

documentDF.show
```

```
documentDF: org.apache.spark.sql.DataFrame = [text: array<string>]
+--------------------+
|                text|
+--------------------+
|[Hi, I, heard, ab...|
|[I, wish, Java, c...|
|[Logistic, regres...|
+--------------------+
```

特征的提取及ML实现详解

```
// 2 word2Vec
val word2Vec = new Word2Vec()
  .setInputCol("text")
  .setOutputCol("result")
  .setVectorSize(3)
  .setMinCount(0)
val model = word2Vec.fit(documentDF)

// 3 测试及结果展示
val result = model.transform(documentDF)
result.collect().foreach {
  case Row(text: Seq[_], features: Vector) =>
    println(s"Text: [${text.mkString(", ")}] => \nVector: $features\n")
}
```

```
word2Vec: org.apache.spark.ml.feature.Word2Vec = w2v_486f728716c8
model: org.apache.spark.ml.feature.Word2VecModel = w2v_486f728716c8
result: org.apache.spark.sql.DataFrame = [text: array<string>, result: vector]
Text: [Hi, I, heard, about, Spark] =>
Vector: [0.03173386193811894,0.009443491697311401,0.024377789348363876]
Text: [I, wish, Java, could, use, case, classes] =>
Vector: [0.025682436302304268,0.0314303718706859,-0.01815584538105343]
Text: [Logistic, regression, models, are, neat] =>
Vector: [0.022586782276630402,-0.01601201295852661,0.05122732147574425]
```

特征的提取及ML实现详解

- **3、CountVectorizer**
- CountVectorizer和CountVectorizerModel是将文本文档集合转换为向量。 当先验词典不可用时，CountVectorizer可以用作估计器来提取词汇表，并生成CountVectorizerModel。 该模型通过词汇生成文档的稀疏表示，然后可以将其传递给其他算法，如LDA。
- 在拟合过程中，CountVectorizer将选择通过语料库按术语频率排序的top前几vocabSize词。 可选参数minDF还通过指定术语必须出现以包含在词汇表中的文档的最小数量（或小于1.0）来影响拟合过程。 另一个可选的二进制切换参数控制输出向量。 如果设置为true，则所有非零计数都设置为1.对于模拟二进制而不是整数的离散概率模型，这是非常有用的。

特征的提取及ML实现详解

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")
```

文本中的每一行都是Array[String]类型的文档。调用CountVectorizer的拟合产生一个具有词汇表（a, b, c）的CountVectorizerModel。：

```
 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])
```

每个向量表示文档在词汇表上的标记数。

特征的提取及ML实现详解

```
// 1 样本准备
val df = spark.createDataFrame(Seq(
  (0, Array("a", "b", "c")),
  (1, Array("a", "b", "b", "c", "a")))).toDF("id", "words")


df.show
```

特征的提取及ML实现详解

```
// 2 CountVectorizer
val cvModel: CountVectorizerModel = new CountVectorizer()
  .setInputCol("words")
  .setOutputCol("features")
  .setVocabSize(3)
  .setMinDF(2)
  .fit(df)

// 3 测试及结果展示
val cvm = new CountVectorizerModel(Array("a", "b", "c"))
  .setInputCol("words")
  .setOutputCol("features")

cvModel.transform(df).show(false)
```

特征的提取及ML实现详解

```
// 1 样本准备
val df = spark.createDataFrame(Seq(
  (0, Array("a", "b", "c")),
  (1, Array("a", "b", "b", "c", "a")))).toDF("id", "words")

df.show
```

```
df: org.apache.spark.sql.DataFrame = [id: int, words: array<string>]
+---+---------------+
| id|          words|
+---+---------------+
|  0|      [a, b, c]|
|  1|[a, b, b, c, a]|
+---+---------------+
```

特征的提取及ML实现详解

```
// 2 CountVectorizer
val cvModel: CountVectorizerModel = new CountVectorizer()
  .setInputCol("words")
  .setOutputCol("features")
  .setVocabSize(3)
  .setMinDF(2)
  .fit(df)

// 3 测试及结果展示
val cvm = new CountVectorizerModel(Array("a", "b", "c"))
  .setInputCol("words")
  .setOutputCol("features")

cvModel.transform(df).show(false)
```

```
cvModel: org.apache.spark.ml.feature.CountVectorizerModel = cntVec_68020b2917c8
cvm: org.apache.spark.ml.feature.CountVectorizerModel = cntVecModel_55b35be738c6
+---+---------------+-------------------------+
|id |words          |features                 |
+---+---------------+-------------------------+
|0  |[a, b, c]      |(3,[0,1,2],[1.0,1.0,1.0])|
|1  |[a, b, b, c, a]|(3,[0,1,2],[2.0,2.0,1.0])|
+---+---------------+-------------------------+
```

目录

特征的变换

特征的变换及ML实现详解

- **1、Tokenizer（分词器）**
- Tokenization是将文本（如一个句子）拆分成单词的过程。（在Spark ML中）Tokenizer（分词器）提供此功能。
- 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，这样作为划分结果找到的所有匹配项。

特征的变换及ML实现详解

```
//1 样本准备
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")

  sentenceDataFrame.show(false)
```

特征的变换及ML实现详解

```
    //2  Tokenizer分词器
    val tokenizer = new
Tokenizer().setInputCol("sentence").setOutputCol("words")
    //3  RegexTokenizer分词器
    val regexTokenizer = new RegexTokenizer()
      .setInputCol("sentence")
      .setOutputCol("words")
      .setPattern("\\W")
    // 或者通过gaps”设置为 false ，指定正则表达式"pattern"表示"tokens"，而不是分隔符，例如：.setPattern("\\w+").setGaps(false)
    val regexTokenizer2 = new RegexTokenizer()
      .setInputCol("sentence")
      .setOutputCol("words")
      .setPattern("\\w+")
      .setGaps(false)
```

特征的变换及ML实现详解

```
// udf 计算长度
val countTokens = udf { (words: Seq[String]) => words.length }
// 测试1
val tokenized = tokenizer.transform(sentenceDataFrame)
tokenized.select("sentence", "words")
  .withColumn("tokens", countTokens(col("words"))).show(false)
// 测试2
val regexTokenized = regexTokenizer.transform(sentenceDataFrame)
regexTokenized.select("sentence", "words")
  .withColumn("tokens", countTokens(col("words"))).show(false)
// 测试3
val regexTokenized2 = regexTokenizer2.transform(sentenceDataFrame)
regexTokenized2.select("sentence", "words")
  .withColumn("tokens", countTokens(col("words"))).show(false)
```

特征的变换及ML实现详解

```
//1 样本准备
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")

  sentenceDataFrame.show(false)
```

特征的变换及ML实现详解

```
import org.apache.spark.sql.SparkSession

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

import org.apache.spark.ml.feature._
import org.apache.spark.ml.classification.LogisticRegression
import org.apache.spark.ml.{ Pipeline, PipelineModel }
import org.apache.spark.ml.param.ParamMap
import org.apache.spark.ml.linalg.{ Vector, Vectors }
import org.apache.spark.sql.Row
```

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.ml.feature.{RegexTokenizer, Tokenizer}
import org.apache.spark.sql.functions._
import org.apache.spark.ml.feature._
import org.apache.spark.ml.classification.LogisticRegression
import org.apache.spark.ml.{Pipeline, PipelineModel}
import org.apache.spark.ml.param.ParamMap
import org.apache.spark.ml.linalg.{Vector, Vectors}
import org.apache.spark.sql.Row
```

特征的变换及ML实现详解

```
    //1 样本准备
    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")

sentenceDataFrame.show(false)
```

```
sentenceDataFrame: org.apache.spark.sql.DataFrame = [id: int, sentence: string]
+---+-----------------------------------+
|id |sentence                           |
+---+-----------------------------------+
|0  |Hi I heard about Spark             |
|1  |I wish Java could use case classes |
|2  |Logistic,regression,models,are,neat|
+---+-----------------------------------+
```

特征的变换及ML实现详解

```
//2  Tokenizer分词器
val tokenizer = new Tokenizer().setInputCol("sentence").setOutputCol("words")

//3  RegexTokenizer分词器
val regexTokenizer = new RegexTokenizer()
  .setInputCol("sentence")
  .setOutputCol("words")
  .setPattern("\\W")
// 或者通过gaps"设置为 false ， 指定正则表达式"pattern"表示"tokens"， 而不是分隔符， 例如： .setPattern("\\w+").setGaps(false)
val regexTokenizer2 = new RegexTokenizer()
  .setInputCol("sentence")
  .setOutputCol("words")
  .setPattern("\\w+")
  .setGaps(false)
```

```
tokenizer: org.apache.spark.ml.feature.Tokenizer = tok_673d61760718
regexTokenizer: org.apache.spark.ml.feature.RegexTokenizer = regexTok_fc1925997963
regexTokenizer2: org.apache.spark.ml.feature.RegexTokenizer = regexTok_492aaea9d657
```

特征的变换及ML实现详解

```
// udf 计算长度
val countTokens = udf { (words: Seq[String]) => words.length }

// 测试1
val tokenized = tokenizer.transform(sentenceDataFrame)
tokenized.select("sentence", "words")
  .withColumn("tokens", countTokens(col("words"))).show(false)
```

```
countTokens: org.apache.spark.sql.expressions.UserDefinedFunction = UserDefinedFunction(<function1>,IntegerType,Sc
tokenized: org.apache.spark.sql.DataFrame = [id: int, sentence: string ... 1 more field]
+----------------------------------+-------------------------------------------+------+
|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     |
+----------------------------------+-------------------------------------------+------+
```

特征的变换及ML实现详解

```
// 测试2
val regexTokenized = regexTokenizer.transform(sentenceDataFrame)
regexTokenized.select("sentence", "words")
  .withColumn("tokens", countTokens(col("words"))).show(false)

// 测试3
val regexTokenized2 = regexTokenizer2.transform(sentenceDataFrame)
regexTokenized2.select("sentence", "words")
  .withColumn("tokens", countTokens(col("words"))).show(false)
```

```
regexTokenized: org.apache.spark.sql.DataFrame = [id: int, sentence: string ... 1 more field]
+-----------------------------------+-------------------------------------------+------+
|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     |
+-----------------------------------+-------------------------------------------+------+
regexTokenized2: org.apache.spark.sql.DataFrame = [id: int, sentence: string ... 1 more field]
+-----------------------------------+-------------------------------------------+------+
|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     |
+-----------------------------------+-------------------------------------------+------+
```

特征的变换及ML实现详解

- **2、StopWordsRemover（去停用词）**
- Stop words（停用字）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是将输入的字符串（如分词器 Tokenizer 的输出）中的停用字删除。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）。

特征的变换及ML实现详解

```
// 1 样本准备
val dataSet = spark.createDataFrame(Seq(
  (0, Seq("I", "saw", "the", "red", "baloon")),
  (1, Seq("Mary", "had", "a", "little", "lamb")))).toDF("id", "raw")

// 2 StopWordsRemover
val remover = new StopWordsRemover()
  .setInputCol("raw")
  .setOutputCol("filtered")

// 3 结果显示
remover.transform(dataSet).show(false)
```

特征的变换及ML实现详解

```scala
// 1 样本准备
val dataSet = spark.createDataFrame(Seq(
  (0, Seq("I", "saw", "the", "red", "baloon")),
  (1, Seq("Mary", "had", "a", "little", "lamb")))).toDF("id", "raw")

// 2 StopWordsRemover
val remover = new StopWordsRemover()
  .setInputCol("raw")
  .setOutputCol("filtered")

// 3 结果显示
remover.transform(dataSet).show(false)
```

```
dataSet: org.apache.spark.sql.DataFrame = [id: int, raw: array<string>]
remover: org.apache.spark.ml.feature.StopWordsRemover = stopWords_e80d56a5ee85
+---+----------------------------+--------------------+
|id |raw                         |filtered            |
+---+----------------------------+--------------------+
|0  |[I, saw, the, red, baloon]  |[saw, red, baloon]  |
|1  |[Mary, had, a, little, lamb]|[Mary, little, lamb]|
+---+----------------------------+--------------------+
```

特征的变换及ML实现详解

- **3、n-gram（N元模型）**
- 一个 n-gram是一个长度为n（整数）的字的序列。NGram可用于将输入特征转换成n-grams。
- N-Gram 的输入为一系列的字符串（例如：Tokenizer分词器的输出）。参数 n 表示每个 n-gram 中单词（terms）的数量。输出将由 n-gram 序列组成，其中每个 n-gram 由空格分隔的 n 个连续词的字符串表示。如果输入的字符串序列少于n个单词，NGram 输出为空。

特征的变换及ML实现详解

```
    // 1 样本准备
    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("id", "words")

    // 2 ngram
    val ngram = new
NGram().setN(2).setInputCol("words").setOutputCol("ngrams")
    val ngramDataFrame = ngram.transform(wordDataFrame)

    // 3 结果显示
    ngramDataFrame.select("ngrams").show(false)
```

特征的变换及ML实现详解

```
// 1 样本准备
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("id", "words")

// 2 ngram
val ngram = new NGram().setN(2).setInputCol("words").setOutputCol("ngrams")
val ngramDataFrame = ngram.transform(wordDataFrame)

// 3 结果显示
ngramDataFrame.select("ngrams").show(false)
```

```
wordDataFrame: org.apache.spark.sql.DataFrame = [id: int, words: array<string>]
ngram: org.apache.spark.ml.feature.NGram = ngram_a99339fb612e
ngramDataFrame: org.apache.spark.sql.DataFrame = [id: int, words: array<string> ... 1 more field]
+------------------------------------------------------------------+
|ngrams                                                            |
+------------------------------------------------------------------+
|[Hi I, I heard, heard about, about Spark]                         |
|[I wish, wish Java, Java could, could use, use case, case classes]|
|[Logistic regression, regression models, models are, are neat]    |
+------------------------------------------------------------------+
```

特征的变换及ML实现详解

- **4、Binarizer（二值化）**
- Binarization （二值化）是将数值特征阈值化为二进制（0/1）特征的过程。
- 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）特征值大于阀值将二值化为1.0，特征值小于等于阀值将二值化为0.0。inputCol 支持向量（Vector）和双精度（Double）类型。

特征的变换及ML实现详解

```
 // 1 样本准备
val data = Array((0, 0.1), (1, 0.8), (2, 0.2))
val dataFrame = spark.createDataFrame(data).toDF("id", "feature")

// 2 Binarizer
val binarizer: Binarizer = new Binarizer()
  .setInputCol("feature")
  .setOutputCol("binarized_feature")
  .setThreshold(0.5)
val binarizedDataFrame = binarizer.transform(dataFrame)

// 3 结果显示
println(s"Binarizer output with Threshold = ${binarizer.getThreshold}")
binarizedDataFrame.show()
```

特征

```scala
// 1 样本准备
val data = Array((0, 0.1), (1, 0.8), (2, 0.2))
val dataFrame = spark.createDataFrame(data).toDF("id", "feature")

// 2 Binarizer
val binarizer: Binarizer = new Binarizer()
  .setInputCol("feature")
  .setOutputCol("binarized_feature")
  .setThreshold(0.5)
val binarizedDataFrame = binarizer.transform(dataFrame)

// 3 结果显示
println(s"Binarizer output with Threshold = ${binarizer.getThreshold}")
binarizedDataFrame.show()
```

```
data: Array[(Int, Double)] = Array((0,0.1), (1,0.8), (2,0.2))
dataFrame: org.apache.spark.sql.DataFrame = [id: int, feature: double]
binarizer: org.apache.spark.ml.feature.Binarizer = binarizer_2e98a379e4ed
binarizedDataFrame: org.apache.spark.sql.DataFrame = [id: int, feature: double ... 1 more field]
Binarizer output with Threshold = 0.5
+---+-------+-----------------+
| id|feature|binarized_feature|
+---+-------+-----------------+
|  0|    0.1|              0.0|
|  1|    0.8|              1.0|
|  2|    0.2|              0.0|
+---+-------+-----------------+
```

特征的变换及ML实现详解

- **5、PCA（主元分析）**
- PCA 是使用正交变换将可能相关变量的一组观察值转换为称为主成分的线性不相关变量的值的一组统计过程。 PCA 类训练使用 PCA 将向量投影到低维空间的模型。

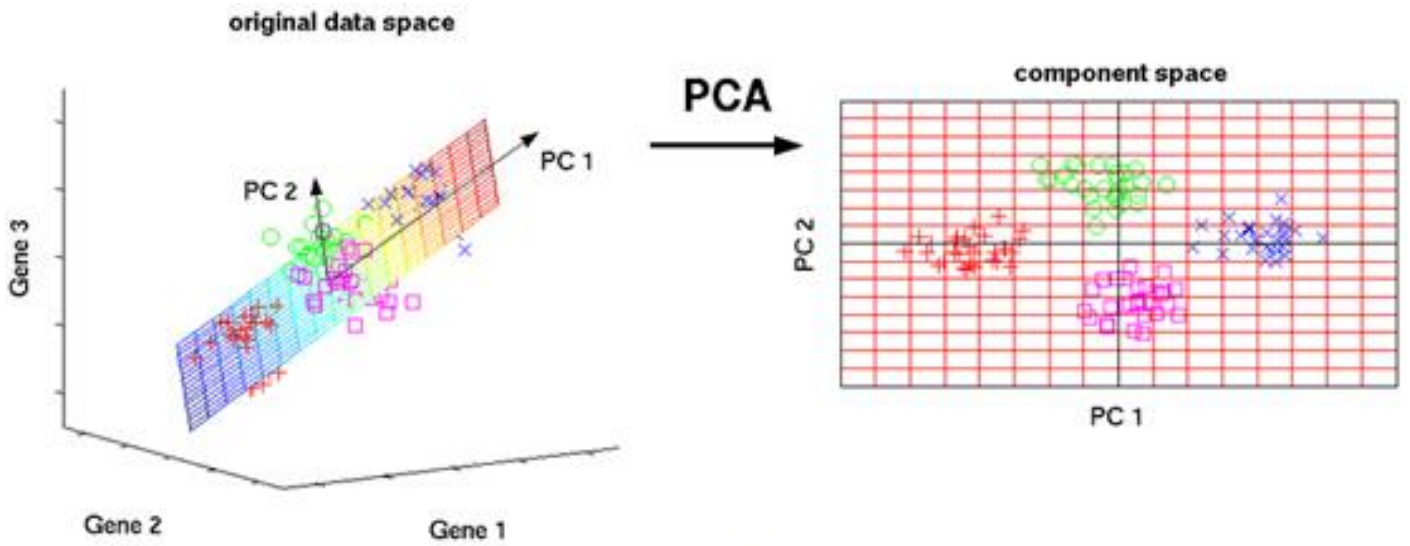


图 1

特征的变换及ML实现详解

```
// 1 样本准备
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")
// 2 CountVectorizer
val pca = new PCA()
  .setInputCol("features")
  .setOutputCol("pcaFeatures")
  .setK(3)
  .fit(df)
// 3 测试及结果展示
val result = pca.transform(df).select("pcaFeatures")
result.show(false)
```

```
    // 2 CountVectorizer
    val pca = new PCA()
      .setInputCol("features")
      .setOutputCol("pcaFeatures")
      .setK(3)
      .fit(df)

    // 3 测试及结果展示
    val result = pca.transform(df).select("pcaFeatures")
    result.show(false)
```

```
data: Array[org.apache.spark.ml.linalg.Vector] = Array((5,[1,3],[1.0,7.0]), [2.0,0.0,3.0,4.0,5.0], [4.0,0.0,0.0,6.0,7.0])
df: org.apache.spark.sql.DataFrame = [features: vector]
+--------------------+
|            features|
+--------------------+
| (5,[1,3],[1.0,7.0])|
|[2.0,0.0,3.0,4.0,...|
|[4.0,0.0,0.0,6.0,...|
+--------------------+
pca: org.apache.spark.ml.feature.PCAModel = pca_da4aae2c5f9d
result: org.apache.spark.sql.DataFrame = [pcaFeatures: vector]
+-----------------------------------------------------------+
|pcaFeatures                                                |
+-----------------------------------------------------------+
|[1.6485728230883807,-4.013282700516296,-5.524543751369388] |
|[-4.645104331781534,-1.1167972663619026,-5.524543751369387]|
|[-6.428880535676489,-5.337951427775355,-5.524543751369389] |
+-----------------------------------------------------------+
```

特征的变换及ML实现详解

- **6、PolynomialExpansion（多项式扩展）**
- Polynomial expansion（多项式展开）是将特征扩展为多项式空间的过程，多项式空间由原始维度的n度组合组成。 PolynomialExpansion类提供此功能。

特征的变换及ML实现详解

```
// 1 样本准备
val data = Array(
  Vectors.dense(2.0, 1.0),
  Vectors.dense(0.0, 0.0),
  Vectors.dense(3.0, -1.0))
val df = spark.createDataFrame(data.map(Tuple1.apply)).toDF("features")
// 2 polyExpansion
val polyExpansion = new PolynomialExpansion()
  .setInputCol("features")
  .setOutputCol("polyFeatures")
  .setDegree(3)
// 3 测试及结果展示
val polyDF = polyExpansion.transform(df)
polyDF.show(false)
```

```scala
// 2 polyExpansion
val polyExpansion = new PolynomialExpansion()
  .setInputCol("features")
  .setOutputCol("polyFeatures")
  .setDegree(3)

// 3 测试及结果展示
val polyDF = polyExpansion.transform(df)
polyDF.show(false)
```

```
data: Array[org.apache.spark.ml.linalg.Vector] = Array([2.0,1.0], [0.0,0.0], [3.0,-1.0])
df: org.apache.spark.sql.DataFrame = [features: vector]
+----------+
|  features|
+----------+
| [2.0,1.0]|
| [0.0,0.0]|
|[3.0,-1.0]|
+----------+
polyExpansion: org.apache.spark.ml.feature.PolynomialExpansion = poly_865d6f0babda
polyDF: org.apache.spark.sql.DataFrame = [features: vector, polyFeatures: vector]
+----------+----------------------------------------+
|features  |polyFeatures                            |
+----------+----------------------------------------+
|[2.0,1.0] |[2.0,4.0,8.0,1.0,2.0,4.0,1.0,2.0,1.0]   |
|[0.0,0.0] |[0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0]   |
|[3.0,-1.0]|[3.0,9.0,27.0,-1.0,-3.0,-9.0,1.0,3.0,-1.0]|
+----------+----------------------------------------+
```

特征的变换及ML实现详解

- **7、Discrete Cosine Transform（DCT 离散余弦变换）**
- Discrete Cosine Transform（离散余弦变换） 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的功能，将离散余弦变换后结果乘以 得到一个与时域矩阵长度一致的矩阵。没有偏移被应用于变换的序列（例如，变换的序列的第0个元素是第0个DCT系数，而不是第N / 2个），即输入序列与输出之间是一一对应的。

特征的变换及ML实现详解

```
// 1 样本准备
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")
// 2 DCT
val dct = new DCT()
  .setInputCol("features")
  .setOutputCol("featuresDCT")
  .setInverse(false)
// 3 测试及结果展示
val dctDf = dct.transform(df)
dctDf.select("featuresDCT").show(false)
```

```scala
// 2 DCT
val dct = new DCT()
  .setInputCol("features")
  .setOutputCol("featuresDCT")
  .setInverse(false)

// 3 测试及结果展示
val dctDf = dct.transform(df)
dctDf.select("featuresDCT").show(false)
```

```
data: Seq[org.apache.spark.ml.linalg.Vector] = List([0.0,1.0,-2.0,3.0], [-1.0,2.0,4.0,-7.0], [14.0,-2.0,-5.0,1.0])
df: org.apache.spark.sql.DataFrame = [features: vector]
+--------------------+
|            features|
+--------------------+
|  [0.0,1.0,-2.0,3.0]|
| [-1.0,2.0,4.0,-7.0]|
|[14.0,-2.0,-5.0,1.0]|
+--------------------+
dct: org.apache.spark.ml.feature.DCT = dct_06ab8d30e486
dctDf: org.apache.spark.sql.DataFrame = [features: vector, featuresDCT: vector]
+----------------------------------------------------------------+
|featuresDCT                                                     |
+----------------------------------------------------------------+
|[1.0,-1.1480502970952693,2.0000000000000004,-2.7716385975338604]|
|[-1.0,3.378492794482933,-7.000000000000001,2.9301512653149677]  |
|[4.0,9.304453421915744,11.000000000000002,1.5579302036357163]   |
+----------------------------------------------------------------+
```

特征的变换及ML实现详解

- **8、StringIndexer（字符串-索引变换）**
- StringIndexer（字符串-索引变换）将标签的字符串列编号变成标签索引列。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。

特征的变换及ML实现详解

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）。第二多的“c”得到1的索引，“b”得到2的索引

特征的变换及ML实现详解

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").

```
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”

特征的变换及ML实现详解

- **9、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。

特征的变换及ML实现详解

9、IndexToString（索引-字符串变换）

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
```

特征的变换及ML实现详解

```
// 1 样本准备
val df = spark.createDataFrame(Seq(
  (0, "a"),
  (1, "b"),
  (2, "c"),
  (3, "a"),
  (4, "a"),
  (5, "c"))).toDF("id", "category")
// 2 StringIndexer
val indexer = new StringIndexer()
  .setInputCol("category")
  .setOutputCol("categoryIndex")
  .fit(df)
val indexed = indexer.transform(df)
indexed.show()
```

特征的变换及ML实现详解

```
// 3 IndexToString
val converter = new IndexToString()
  .setInputCol("categoryIndex")
  .setOutputCol("originalCategory")
val converted = converter.transform(indexed)
converted.select("id", "categoryIndex", "originalCategory").show()
```

特征的

```scala
// 1 样本准备
val df = spark.createDataFrame(Seq(
  (0, "a"),
  (1, "b"),
  (2, "c"),
  (3, "a"),
  (4, "a"),
  (5, "c"))).toDF("id", "category")

// 2 StringIndexer
val indexer = new StringIndexer()
  .setInputCol("category")
  .setOutputCol("categoryIndex")
  .fit(df)
val indexed = indexer.transform(df)
indexed.show()
```

```
df: org.apache.spark.sql.DataFrame = [id: int, category: string]
indexer: org.apache.spark.ml.feature.StringIndexerModel = strIdx_d1286a68cb04
indexed: org.apache.spark.sql.DataFrame = [id: int, category: string ... 1 more field]
+---+--------+-------------+
| 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|
+---+--------+-------------+
```

特征的变换及ML实现详解

```
// 3 IndexToString
val converter = new IndexToString()
  .setInputCol("categoryIndex")
  .setOutputCol("originalCategory")
val converted = converter.transform(indexed)
converted.select("id", "categoryIndex", "originalCategory").show()
```

```
converter: org.apache.spark.ml.feature.IndexToString = idxToStr_79c8b3fc9520
converted: org.apache.spark.sql.DataFrame = [id: int, category: string ... 2 more fields]
+---+-------------+---------------+
| 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|
+---+-------------+---------------+
```

特征的变换及ML实现详解

- **10、OneHotEncoder（独热编码）**
- 独热编码（One-hot encoding）将一列标签索引映射到一列二进制向量，最多只有一个单值。 该编码允许期望连续特征（例如逻辑回归）的算法使用分类特征。

4个样本：类1， 类2， 类3， 类4 => 1，2，3，4

One-hot：

类1 => [1,0,0,0]

类2 => [0,1,0,0]

类3 => [0,0,1,0]

类4 => [0,0,0,1]

10、OneHotEncoder（独热编码）

1、Why do we binarize categorical features?

We binarize the categorical input so that they can be thought of as a vector from the Euclidean space (we call this as embedding the vector in the Euclidean space).使用one-hot编码，将离散特征的取值扩展到了欧式空间，离散特征的某个取值就对应欧式空间的某个点。

2、Why do we embed the feature vectors in the Euclidean space?

Because many algorithms for classification/regression/clustering etc. requires computing distances between features or similarities between features. And many definitions of distances and similarities are defined over features in Euclidean space. So, we would like our features to lie in the Euclidean space as well.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。

特征的变换及ML实现详解

```
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.show()
```

特征的

```
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.show()
```

```
df: org.apache.spark.sql.DataFrame = [id: int, category: string]
indexer: org.apache.spark.ml.feature.StringIndexerModel = strIdx_76d5e017d28f
indexed: org.apache.spark.sql.DataFrame = [id: int, category: string ... 1 more field]
encoder: org.apache.spark.ml.feature.OneHotEncoder = oneHot_fd446e338966
encoded: org.apache.spark.sql.DataFrame = [id: int, category: string ... 2 more fields]
+---+--------+-------------+-------------+
| 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])|
+---+--------+-------------+-------------+
```

特征的变换及ML实现详解

- **11、VectorIndexer(向量类型索引化)**
- VectorIndexer是指定向量数据集中的分类（离散）特征。它可以自动确定哪些特征是离散的，并将原始值转换为离散索引。具体来说，它执行以下操作：
- 取一个Vector类型的输入列和一个参数maxCategories。
- 根据不同值的数量确定哪些特征是离散，其中最多maxCategories的功能被声明为分类。
- 为每个分类功能计算基于0的类别索引。
- 索引分类特征并将原始特征值转换为索引。
- 索引分类功能允许诸如决策树和树组合之类的算法适当地处理分类特征，提高性能。

特征的变换及ML实现详解

```
 val data =
spark.read.format("libsvm").load("data/mllib/sample_libsvm_data.txt")

    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(", "))
    val indexedData = indexerModel.transform(data)
    indexedData.show()
```

```scala
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(", "))
```

```
data: org.apache.spark.sql.DataFrame = [label: double, features: vector]
indexer: org.apache.spark.ml.feature.VectorIndexer = vecIdx_c36135cb8579
indexerModel: org.apache.spark.ml.feature.VectorIndexerModel = vecIdx_c36135cb8579
categoricalFeatures: Set[Int] = Set(645, 69, 365, 138, 101, 479, 333, 249, 0, 555, 666, 88, 170, 115, 276, 308, 5, 449, 120,
 42, 417, 24, 37, 25, 257, 389, 52, 14, 504, 110, 587, 619, 196, 559, 638, 20, 421, 46, 93, 284, 228, 448, 57, 78, 29, 475, 1
 396, 89, 133, 116, 1, 507, 312, 74, 307, 452, 6, 248, 60, 117, 678, 529, 85, 201, 220, 366, 534, 102, 334, 28, 38, 561, 392,
 65, 97, 665, 583, 285, 224, 650, 615, 9, 53, 169, 593, 141, 610, 420, 109, 256, 225, 339, 77, 193, 669, 476, 642, 637, 590,
8, 311, 558, 674, 530, 586, 618, 166, 32, 34, 148, 45, 161, 279, 64, 689...Chose 351 categorical features: 645, 69, 365, 138,
 5, 449, 120, 247, 614, 677, 202, 10, 56, 533, 142, 500, 340, 670, 174, 42, 417, 24, 37, 25, 257, 389, 52, 14, 504, 110, 587,
 78, 29, 475, 164, 591, 646, 253, 106, 121, 84, 480, 147, 280, 61, 221, 396, 89, 133, 116, 1, 507, 312, 74, 307, 452, 6, 248,
 38, 561, 392, 70, 424, 192, 21, 137, 165, 33, 92, 229, 252, 197, 361, 65, 97, 665, 583, 285, 224, 650, 615, 9, 53, 169, 593,
2, 637, 590, 679, 96, 393, 647, 173, 13, 41, 503, 134, 73, 105, 2, 508, 311, 558, 674, 530, 586, 618, 166, 32, 34, 148, 45, 1
4, 59, 118, 281, 27, 641, 71, 391, 12, 445, 54, 313, 611, 144, 49, 335, 86, 672, 172, 113, 681, 219, 419, 81, 230, 362, 451,
1, 66, 251, 668, 198, 108, 278, 223, 394, 306, 135, 563, 226, 3, 505, 80, 167, 35, 473, 675, 589, 162, 531, 680, 255, 648, 11
0, 50, 67, 199, 673, 16, 585, 502, 338, 643, 31, 336, 613, 11, 72, 175, 446, 612, 143, 43, 250, 231, 450, 99, 363, 556, 87, 2
14, 171, 139, 418, 23, 8, 75, 119, 58, 667, 478, 536, 82, 620, 447, 36, 168, 146, 30, 51, 190, 19, 422, 564, 305, 107, 4, 136
4, 47, 15, 163, 200, 68, 62, 277, 691, 501, 90, 111, 254, 227, 337, 122, 83, 309, 560, 639, 676, 222, 592, 364, 100
```

特征的变换及ML实现详解

```
// Create new column "indexed" with categorical values transformed to indices
val indexedData = indexerModel.transform(data)
indexedData.show()
```

```
|  0.0|(692,[127,128,129...|(692,[127,128,129...|
|  1.0|(692,[158,159,160...|(692,[158,159,160...|
|  1.0|(692,[124,125,126...|(692,[124,125,126...|
|  1.0|(692,[152,153,154...|(692,[152,153,154...|
|  1.0|(692,[151,152,153...|(692,[151,152,153...|
|  0.0|(692,[129,130,131...|(692,[129,130,131...|
|  1.0|(692,[158,159,160...|(692,[158,159,160...|
|  1.0|(692,[99,100,101,...|(692,[99,100,101,...|
|  0.0|(692,[154,155,156...|(692,[154,155,156...|
|  0.0|(692,[127,128,129...|(692,[127,128,129...|
|  1.0|(692,[154,155,156...|(692,[154,155,156...|
|  0.0|(692,[153,154,155...|(692,[153,154,155...|
|  0.0|(692,[151,152,153...|(692,[151,152,153...|
|  1.0|(692,[129,130,131...|(692,[129,130,131...|
|  0.0|(692,[154,155,156...|(692,[154,155,156...|
|  1.0|(692,[150,151,152...|(692,[150,151,152...|
|  0.0|(692,[124,125,126...|(692,[124,125,126...|
|  0.0|(692,[152,153,154...|(692,[152,153,154...|
```

特征的变换及ML实现详解

- **12、Interaction（相互作用）**
- 交互是一个变换器，它采用向量或双值列，并生成一个单个向量列，其中包含来自每个输入列的一个值的所有组合的乘积。
- 例如，如果您有2个向量类型的列，每个列具有3个维度作为输入列，那么您将获得一个9维向量作为输出列。

- For example, given the input feature values Double(2) and Vector(3, 4), the output would be Vector(6, 8) if all input features were numeric. If the first feature was instead nominal with four categories, the output would then be Vector(0, 0, 0, 0, 3, 4, 0, 0).

特征的变换及ML实现详解

12、Interaction（相互作用）

Examples

假设我们有如下DataFrame，列为“id1”, “vec1” 和 “vec2”:

```
id1|vec1          |vec2
 ---|--------------|--------------
 1  |[1.0,2.0,3.0] |[8.0,4.0,5.0]
 2  |[4.0,3.0,8.0] |[7.0,9.0,8.0]
 3  |[6.0,1.0,9.0] |[2.0,3.0,6.0]
 4  |[10.0,8.0,6.0]|[9.0,4.0,5.0]
 5  |[9.0,2.0,7.0] |[10.0,7.0,3.0]
 6  |[1.0,1.0,4.0] |[2.0,8.0,4.0]
```

应用与这些输入列的交互，然后将交互作为输出列包含：

```
id1|vec1          |vec2          |interactedCol
 ---|--------------|--------------|------------------------------------------------------
 1  |[1.0,2.0,3.0] |[8.0,4.0,5.0] |[8.0,4.0,5.0,16.0,8.0,10.0,24.0,12.0,15.0]
 2  |[4.0,3.0,8.0] |[7.0,9.0,8.0] |[56.0,72.0,64.0,42.0,54.0,48.0,112.0,144.0,128.0]
 3  |[6.0,1.0,9.0] |[2.0,3.0,6.0] |[36.0,54.0,108.0,6.0,9.0,18.0,54.0,81.0,162.0]
 4  |[10.0,8.0,6.0]|[9.0,4.0,5.0] |[360.0,160.0,200.0,288.0,128.0,160.0,216.0,96.0,120.0]
 5  |[9.0,2.0,7.0] |[10.0,7.0,3.0]|[450.0,315.0,135.0,100.0,70.0,30.0,350.0,245.0,105.0]
 6  |[1.0,1.0,4.0] |[2.0,8.0,4.0] |[12.0,48.0,24.0,12.0,48.0,24.0,48.0,192.0,96.0]
```

特征的变换及ML实现详解

```
val df = spark.createDataFrame(Seq(
  (1, 1, 2, 3, 8, 4, 5),
  (2, 4, 3, 8, 7, 9, 8),
  (3, 6, 1, 9, 2, 3, 6),
  (4, 10, 8, 6, 9, 4, 5),
  (5, 9, 2, 7, 10, 7, 3),
  (6, 1, 1, 4, 2, 8, 4))).toDF("id1", "id2", "id3", "id4", "id5", "id6", "id7")

df.show
```

```
df: org.apache.spark.sql.DataFrame = [id1: int, id2: int ... 5 more fields]
+---+---+---+---+---+---+---+
|id1|id2|id3|id4|id5|id6|id7|
+---+---+---+---+---+---+---+
|  1|  1|  2|  3|  8|  4|  5|
|  2|  4|  3|  8|  7|  9|  8|
|  3|  6|  1|  9|  2|  3|  6|
|  4| 10|  8|  6|  9|  4|  5|
|  5|  9|  2|  7| 10|  7|  3|
|  6|  1|  1|  4|  2|  8|  4|
+---+---+---+---+---+---+---+
```

特征的变换及ML实现详解

```
val assembler1 = new VectorAssembler().
  setInputCols(Array("id2", "id3", "id4")).
  setOutputCol("vec1")

val assembled1 = assembler1.transform(df)

val assembler2 = new VectorAssembler().
  setInputCols(Array("id5", "id6", "id7")).
  setOutputCol("vec2")

val assembled2 = assembler2.transform(assembled1).select("id1", "vec1", "vec2")

val interaction = new Interaction()
  .setInputCols(Array("id1", "vec1", "vec2"))
  .setOutputCol("interactedCol")
```

```
assembler1: org.apache.spark.ml.feature.VectorAssembler = vecAssembler_7cbabc71c244
assembled1: org.apache.spark.sql.DataFrame = [id1: int, id2: int ... 6 more fields]
assembler2: org.apache.spark.ml.feature.VectorAssembler = vecAssembler_114ff1bb8241
assembled2: org.apache.spark.sql.DataFrame = [id1: int, vec1: vector ... 1 more field]
interaction: org.apache.spark.ml.feature.Interaction = interaction_7d8dd301d558
```

特征的变换及ML实现详解

```
    val interacted = interaction.transform(assembled2)

    interacted.show(truncate = false)
```

```
interacted: org.apache.spark.sql.DataFrame = [id1: int, vec1: vector ... 2 more fields]
+---+--------------+--------------+-----------------------------------------------------+
|id1|vec1          |vec2          |interactedCol                                        |
+---+--------------+--------------+-----------------------------------------------------+
|1  |[1.0,2.0,3.0] |[8.0,4.0,5.0] |[8.0,4.0,5.0,16.0,8.0,10.0,24.0,12.0,15.0]           |
|2  |[4.0,3.0,8.0] |[7.0,9.0,8.0] |[56.0,72.0,64.0,42.0,54.0,48.0,112.0,144.0,128.0]    |
|3  |[6.0,1.0,9.0] |[2.0,3.0,6.0] |[36.0,54.0,108.0,6.0,9.0,18.0,54.0,81.0,162.0]       |
|4  |[10.0,8.0,6.0]|[9.0,4.0,5.0] |[360.0,160.0,200.0,288.0,128.0,160.0,216.0,96.0,120.0]|
|5  |[9.0,2.0,7.0] |[10.0,7.0,3.0]|[450.0,315.0,135.0,100.0,70.0,30.0,350.0,245.0,105.0]|
|6  |[1.0,1.0,4.0] |[2.0,8.0,4.0] |[12.0,48.0,24.0,12.0,48.0,24.0,48.0,192.0,96.0]      |
+---+--------------+--------------+-----------------------------------------------------+
```

- **13、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。

特征的变换及ML实现详解

```
    val dataFrame = spark.createDataFrame(Seq(
      (0, Vectors.dense(1.0, 0.5, -1.0)),
      (1, Vectors.dense(2.0, 1.0, 1.0)),
      (2, Vectors.dense(4.0, 10.0, 2.0)))).toDF("id", "features")
    val normalizer = new Normalizer()
      .setInputCol("features")
      .setOutputCol("normFeatures")
      .setP(1.0)
    val l1NormData = normalizer.transform(dataFrame)
    println("Normalized using L^1 norm")
    l1NormData.show()
    val lInfNormData = normalizer.transform(dataFrame, normalizer.p ->
Double.PositiveInfinity)
    println("Normalized using L^inf norm")
    lInfNormData.show()
```

特征的

```
val dataFrame = spark.createDataFrame(Seq(
  (0, Vectors.dense(1.0, 0.5, -1.0)),
  (1, Vectors.dense(2.0, 1.0, 1.0)),
  (2, Vectors.dense(4.0, 10.0, 2.0)))).toDF("id", "features")

// Normalize each Vector using $L^1$ norm.
val normalizer = new Normalizer()
  .setInputCol("features")
  .setOutputCol("normFeatures")
  .setP(1.0)

val l1NormData = normalizer.transform(dataFrame)
println("Normalized using L^1 norm")
l1NormData.show()
```

```
dataFrame: org.apache.spark.sql.DataFrame = [id: int, features: vector]
normalizer: org.apache.spark.ml.feature.Normalizer = normalizer_07cb796c66ad
l1NormData: org.apache.spark.sql.DataFrame = [id: int, features: vector ... 1 more field]
Normalized using L^1 norm
+---+--------------+------------------+
| id|      features|      normFeatures|
+---+--------------+------------------+
|  0|[1.0,0.5,-1.0]|    [0.4,0.2,-0.4]|
|  1| [2.0,1.0,1.0]|   [0.5,0.25,0.25]|
|  2|[4.0,10.0,2.0]|[0.25,0.625,0.125]|
+---+--------------+------------------+
```

特征的变换及ML实现详解

```
    // Normalize each Vector using $L^\infty$ norm.
    val lInfNormData = normalizer.transform(dataFrame, normalizer.p -> Double.PositiveInfinity)
    println("Normalized using L^inf norm")
    lInfNormData.show()
```

```
lInfNormData: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [id: int, features: vector ... 1 more field]
Normalized using L^inf norm
+---+--------------+--------------+
| id|      features|  normFeatures|
+---+--------------+--------------+
|  0|[1.0,0.5,-1.0]|[1.0,0.5,-1.0]|
|  1| [2.0,1.0,1.0]| [1.0,0.5,0.5]|
|  2|[4.0,10.0,2.0]| [0.4,1.0,0.2]|
+---+--------------+--------------+
```

特征的变换及ML实现详解

- **14、StandardScaler（标准化）**
- StandardScaler转换Vector行的数据集，使每个要素标准化以具有单位标准偏差和 或 零均值。它需要参数：
- withStd：默认为True。将数据缩放到单位标准偏差。
- withMean：默认为false。在缩放之前将数据中心为平均值。它将构建一个密集的输出，所以在应用于稀疏输入时要小心。
- StandardScaler是一个Estimator，可以适合数据集生成StandardScalerModel; 这相当于计算汇总统计数据。 然后，模型可以将数据集中的向量列转换为具有单位标准偏差和/或零平均特征。
- 请注意，如果特征的标准偏差为零，它将在该特征的向量中返回默认的0.0值。

特征的变换及ML实现详解

```
 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()
```

```scala
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(1,false)
```

```
|0.0  |(692,[127,128,129,130,131,154,155,156,157,158,159,181,182,183,184,185,186,187,188,189,207,208,209,210,211,212,213,214,
262,263,264,265,266,267,268,269,270,271,272,273,289,290,291,292,293,294,295,296,297,300,301,302,316,317,318,319,320,321,328,3
73,374,384,385,386,399,400,401,412,413,414,426,427,428,429,440,441,442,454,455,456,457,466,467,468,469,470,482,483,484,493,49
7,548,549,550,566,567,568,569,570,571,572,573,574,575,576,577,578,594,595,596,597,598,599,600,601,602,603,604,622,623,624,625
159.0,253.0,159.0,50.0,48.0,238.0,252.0,252.0,252.0,237.0,54.0,227.0,253.0,252.0,239.0,233.0,252.0,57.0,6.0,10.0,60.0,224.0,2
0,252.0,252.0,253.0,252.0,252.0,96.0,189.0,253.0,167.0,51.0,238.0,253.0,253.0,190.0,114.0,253.0,228.0,47.0,79.0,255.0,168.0,4
43.0,50.0,38.0,165.0,253.0,233.0,208.0,84.0,253.0,252.0,165.0,7.0,178.0,252.0,240.0,71.0,19.0,28.0,253.0,252.0,195.0,57.0,252
253.0,196.0,76.0,246.0,252.0,112.0,253.0,252.0,148.0,85.0,252.0,230.0,25.0,7.0,135.0,253.0,186.0,12.0,85.0,252.0,223.0,7.0,13
73.0,86.0,253.0,225.0,114.0,238.0,253.0,162.0,85.0,252.0,249.0,146.0,48.0,29.0,85.0,178.0,225.0,253.0,223.0,167.0,56.0,85.0,2
0.0,28.0,199.0,252.0,252.0,253.0,252.0,252.0,233.0,145.0,25.0,128.0,252.0,253.0,252.0,141.0,37.0])|(692,[127,128,129,130,131,
189,207,208,209,210,211,212,213,214,215,216,217,235,236,237,238,239,240,241,242,243,244,245,262,263,264,265,266,267,268,269,2
01,302,316,317,318,319,320,321,328,329,330,343,344,345,346,347,348,349,356,357,358,371,372,373,374,384,385,386,399,400,401,41
6,467,468,469,470,482,483,484,493,494,495,496,497,510,511,512,520,521,522,523,538,539,540,547,548,549,550,566,567,568,569,570
```

特征的变换及ML实现详解

- **15、MinMaxScaler（最大-最小规范化）**
- MinMaxScaler转换Vector行的数据集，将每个要素的重新映射到特定范围（通常为[0，1]）。它需要参数：

- min：默认为0.0，转换后的下限，由所有功能共享。
- max：默认为1.0，转换后的上限，由所有功能共享。

- MinMaxScaler计算数据集的统计信息，并生成MinMaxScalerModel。然后，模型可以单独转换每个要素，使其在给定的范围内。
- 特征E的重新缩放值被计算为：

$$Rescaled(e_i) = \frac{e_i - E_{\min}}{E_{\max} - E_{\min}} * (\max - \min) + \min$$

特征的变换及ML实现详解

```
    val dataFrame = spark.createDataFrame(Seq(
      (0, Vectors.dense(1.0, 0.1, -1.0)),
      (1, Vectors.dense(2.0, 1.1, 1.0)),
      (2, Vectors.dense(3.0, 10.1, 3.0)))).toDF("id", "features")
    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)
    println(s"Features scaled to range: [${scaler.getMin},
${scaler.getMax}]")
    scaledData.select("features", "scaledFeatures").show()
```

特征的

```scala
val dataFrame = spark.createDataFrame(Seq(
  (0, Vectors.dense(1.0, 0.1, -1.0)),
  (1, Vectors.dense(2.0, 1.1, 1.0)),
  (2, Vectors.dense(3.0, 10.1, 3.0)))).toDF("id", "features")

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)
println(s"Features scaled to range: [${scaler.getMin}, ${scaler.getMax}]")
scaledData.select("features", "scaledFeatures").show()
```

```
dataFrame: org.apache.spark.sql.DataFrame = [id: int, features: vector]
scaler: org.apache.spark.ml.feature.MinMaxScaler = minMaxScal_d2a7e90b6c85
scalerModel: org.apache.spark.ml.feature.MinMaxScalerModel = minMaxScal_d2a7e90b6c85
scaledData: org.apache.spark.sql.DataFrame = [id: int, features: vector ... 1 more field]
Features scaled to range: [0.0, 1.0]
+--------------+--------------+
|      features|scaledFeatures|
+--------------+--------------+
|[1.0,0.1,-1.0]| [0.0,0.0,0.0]|
| [2.0,1.1,1.0]| [0.5,0.1,0.5]|
|[3.0,10.1,3.0]| [1.0,1.0,1.0]|
+--------------+--------------+
```

特征的变换及ML实现详解

- **16、MaxAbsScaler（绝对值规范化）**
- MaxAbsScaler转换Vector行的数据集，通过划分每个要素中的最大绝对值，将每个要素的重新映射到范围[-1,1]。 它不会使数据移动/居中，因此不会破坏任何稀疏性。
- MaxAbsScaler计算数据集的统计信息，并生成MaxAbsScalerModel。 然后，模型可以将每个要素单独转换为范围[-1,1]。

特征的变换及ML实现详解

```
val dataFrame = spark.createDataFrame(Seq(
  (0, Vectors.dense(1.0, 0.1, -8.0)),
  (1, Vectors.dense(2.0, 1.0, -4.0)),
  (2, Vectors.dense(4.0, 10.0, 8.0)))).toDF("id", "features")

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.select("features", "scaledFeatures").show()
```

特征

```
val dataFrame = spark.createDataFrame(Seq(
  (0, Vectors.dense(1.0, 0.1, -8.0)),
  (1, Vectors.dense(2.0, 1.0, -4.0)),
  (2, Vectors.dense(4.0, 10.0, 8.0)))).toDF("id", "features")

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.select("features", "scaledFeatures").show()
```

```
dataFrame: org.apache.spark.sql.DataFrame = [id: int, features: vector]
scaler: org.apache.spark.ml.feature.MaxAbsScaler = maxAbsScal_0a28c84aaa0c
scalerModel: org.apache.spark.ml.feature.MaxAbsScalerModel = maxAbsScal_0a28c84aaa0c
scaledData: org.apache.spark.sql.DataFrame = [id: int, features: vector ... 1 more field]
+--------------+----------------+
|      features|  scaledFeatures|
+--------------+----------------+
|[1.0,0.1,-8.0]|[0.25,0.01,-1.0]|
|[2.0,1.0,-4.0]|  [0.5,0.1,-0.5]|
|[4.0,10.0,8.0]|   [1.0,1.0,1.0]|
+--------------+----------------+
```

特征的变换及ML实现详解

- **17、VectorAssembler（特征向量合并）**
- **VectorAssembler** 是将给定的一系列的列合并到单个向量列中的 **transformer**。它可以将原始特征和不同特征transformers（转换器）生成的特征合并为单个特征向量，来训练 **ML** 模型,如逻辑回归和决策树等机器学习算法。**VectorAssembler** 可接受以下的输入列类型：所有数值型、布尔类型、向量类型。输入列的值将按指定顺序依次添加到一个向量中
  - **Examples**
    假设我们有一个 **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 是一个包含3个用户特征的特征列，我们希望将 **hour**, **mobile** 以及 **userFeatures** 组合为一个单一特征向量叫做 **features**，**userFeatures**，输出列为 **features**，转换后我们应该得到以下结果：

```
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]
```

特征的变换及ML实现详解

```
    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())
```

特征的

```
val dataFrame = spark.createDataFrame(Seq(
  (0, Vectors.dense(1.0, 0.1, -8.0)),
  (1, Vectors.dense(2.0, 1.0, -4.0)),
  (2, Vectors.dense(4.0, 10.0, 8.0)))).toDF("id", "features")

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.select("features", "scaledFeatures").show()
```

```
dataFrame: org.apache.spark.sql.DataFrame = [id: int, features: vector]
scaler: org.apache.spark.ml.feature.MaxAbsScaler = maxAbsScal_0a28c84aaa0c
scalerModel: org.apache.spark.ml.feature.MaxAbsScalerModel = maxAbsScal_0a28c84aaa0c
scaledData: org.apache.spark.sql.DataFrame = [id: int, features: vector ... 1 more field]
+--------------+----------------+
|      features|  scaledFeatures|
+--------------+----------------+
|[1.0,0.1,-8.0]|[0.25,0.01,-1.0]|
|[2.0,1.0,-4.0]|  [0.5,0.1,-0.5]|
|[4.0,10.0,8.0]|   [1.0,1.0,1.0]|
+--------------+----------------+
```

特征的变换及ML实现详解

- **18、QuantileDiscretizer（分位数离散化）**
- QuantileDiscretizer（分位数离散化）采用具有连续特征的列，并输出具有分类特征的列.**bin**（分级）的数量由**numBuckets** 参数设置。**buckets（区间数）**有可能小于这个值，例如,如果输入的不同值太少,就无法创建足够的不同的**quantiles**（分位数）.
- **NaN values**：在 **QuantileDiscretizer fitting** 时，**NaN**值会从列中移除.这将产生一个 **Bucketizer** 模型进行预测.在转换过程中,**Bucketizer** 会发出错误信息当在数据集中找到 **NaN** 值，但用户也可以通过设置 **handleInvalid** 来选择保留或删除数据集中的 **NaN** 值.如果用户选择保留 **NaN** 值,那么它们将被特别处理并放入自己的 **bucket**（区间）中.例如,如果使用4个 **buckets**（区间）,那么非 **NaN** 数据将放入 **buckets[0-3]**，NaN将计数在特殊的 **bucket[4]** 中.

特征的变换及ML实现详解

- **18、QuantileDiscretizer（分位数离散化）**
- **Algorithm**：使用近似算法来选择 **bin** 的范围（有关详细说明可以参考 approxQuantile 的文档）。可以使用relativeError参数来控制近似的精度。当设置为零时,计算精确的 **quantiles**（分位数）（注意：计算 **quantiles**（分位数）是一项昂贵的操作.下边界和上边界将被 **-Infinity**（负无穷） 和 **+Infinity**（正无穷） 覆盖所有实际值）.

特征的

Examples

假设我们有一个 **DataFrame** 包含 **id**, **hour** 列：

```
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，我们想要将连续的特征变成一个特征。将参数 **numBuckets** 设置为 **3**，我们应该得到以下 **DataFrame** ：

```
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
```

特征的变换及ML实现详解

```
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()
```

特征的变换及ML实现详解

```
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()
```

```
data: Array[(Int, Double)] = Array((0,18.0), (1,19.0), (2,8.0), (3,5.0), (4,2.2))
df: org.apache.spark.sql.DataFrame = [id: int, hour: double]
discretizer: org.apache.spark.ml.feature.QuantileDiscretizer = quantileDiscretizer_a
result: org.apache.spark.sql.DataFrame = [id: int, hour: double ... 1 more field]
+---+----+------+
| 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|
+---+----+------+
```

目录

特征的选择

特征的选择及ML实现详解

- **1、VectorSlicer（向量切片机）**
- VectorSlicer是一个转换器，它采用特征向量，并输出一个新的特征向量与原始特征的子阵列。从向量列中提取特征很有用。
- VectorSlicer接受具有指定索引的向量列，然后输出一个新的向量列，其值通过这些索引进行选择。有两种类型的指数：
- 代表向量中的索引的整数索引，setIndices()。
- 表示向量中特征名称的字符串索引，setNames()，此类要求向量列有*AttributeGroup*，因为实现在*Attribute*的name字段上匹配。
- 整数和字符串的规格都可以接受。此外，您可以同时使用整数索引和字符串名称。必须至少选择一个特征。重复的功能是不允许的，所以选择的索引和名称之间不能有重叠。请注意，如果选择了功能的名称，则会遇到空的输入属性时会抛出异常。

特征的选择及ML实现详解

- 1、Vect

Examples

假设我们有一个含有userFeatures列的DataFrame：

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

userFeatures是一个包含三个用户功能的向量列。
假设userFeature的第一列全部为0，因此我们要删除它并仅选择最后两列。
VectorSlicer使用setIndices（1,2）选择最后两个元素，然后生成一个名为features的新向量列：

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

假设我们对userFeatures具有潜在的输入属性，即["f1", "f2", "f3"]，那么我们可以使用setNames("f2", "f3")来选择它们。

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

特征的选择及ML实现详解

```
    val data = Arrays.asList(
      Row(Vectors.sparse(3, Seq((0, -2.0), (1, 2.3)))),
      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"))
    val output = slicer.transform(dataset)
    output.show(false)
```

```
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.sparse(3, Seq((0, -2.0), (1, 2.3)))),
      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())))
    dataset.show
```

```
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
data: java.util.List[org.apache.spark.sql.Row] = [[(3,[0,1],[-2.0,2.3])], [[-2.0,2.3,0.0]]]
defaultAttr: org.apache.spark.ml.attribute.NumericAttribute = {"type":"numeric"}
attrs: Array[org.apache.spark.ml.attribute.NumericAttribute] = Array({"type":"numeric","name":"f1"},
attrGroup: org.apache.spark.ml.attribute.AttributeGroup = {"ml_attr":{"attrs":{"numeric":[{"idx":0,"r
dataset: org.apache.spark.sql.DataFrame = [userFeatures: vector]
+--------------------+
|        userFeatures|
+--------------------+
|(3,[0,1],[-2.0,2.3])|
|      [-2.0,2.3,0.0]|
+--------------------+
```

特征的选择及ML实现详解

```
    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)
    output.show(false)
```

```
slicer: org.apache.spark.ml.feature.VectorSlicer = vectorSlicer_56e4c1c3ac7d
res150: slicer.type = vectorSlicer_56e4c1c3ac7d
output: org.apache.spark.sql.DataFrame = [userFeatures: vector, features: vector]
+--------------------+-------------+
|userFeatures        |features     |
+--------------------+-------------+
|(3,[0,1],[-2.0,2.3])|(2,[0],[2.3])|
|[-2.0,2.3,0.0]      |[2.3,0.0]    |
+--------------------+-------------+
```

特征的选择及ML实现详解

- **2、RFormula（R模型公式）**
- RFormula选择由R模型公式（R model formula）指定的列。目前，我们支持R运算符的有限子集，包括‘~’，‘.’，‘:’，‘+’以及‘- ‘，基本操作如下：
- ~分隔目标和对象
- +合并对象，“+ 0”表示删除截距
- - 删除对象，“ - 1”表示删除截距
- :交互（数字乘法或二值化分类值）
- . 除了目标外的全部列

特征的选择及ML实现详解

- **2、RFormula（R模型公式）**
- 假设a和b是double列，我们使用以下简单的例子来说明RFormula的效果：
- 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产生一个特征向量列和一个标签的double列或label列。 像R在线性回归中使用公式时，字符型的输入将转换成one-hot编码，数字列将被转换为双精度。 如果label列是类型字符串，则它将首先使用StringIndexer转换为double。 如果DataFrame中不存在label列，则会从公式中指定的响应变量创建输出标签列。

特征的选择及ML实现详解

2、RFormula（R模型公式）

Examples

假设我们有一个具有列id，country，hour和clicked的DataFrame：

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

如果我们使用具有clicked ~ country + hour的公式字符串的RFormula，这表示我们想要基于country 和hour预测clicked，转换后我们应该得到以下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
```

特征的选择及ML实现详解

```
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()
```

```
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.sparse(3, Seq((0, -2.0), (1, 2.3)))),
      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())))
    dataset.show
```

```
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
data: java.util.List[org.apache.spark.sql.Row] = [[(3,[0,1],[-2.0,2.3])], [[-2.0,2.3,0.0]]]
defaultAttr: org.apache.spark.ml.attribute.NumericAttribute = {"type":"numeric"}
attrs: Array[org.apache.spark.ml.attribute.NumericAttribute] = Array({"type":"numeric","name":"f1"},
attrGroup: org.apache.spark.ml.attribute.AttributeGroup = {"ml_attr":{"attrs":{"numeric":[{"idx":0,"r
dataset: org.apache.spark.sql.DataFrame = [userFeatures: vector]
+--------------------+
|        userFeatures|
+--------------------+
|(3,[0,1],[-2.0,2.3])|
|      [-2.0,2.3,0.0]|
+--------------------+
```

特征的选择及ML实现详解

```
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()
```

```
dataset: org.apache.spark.sql.DataFrame = [id: int, country: string ... 2 more fields]
formula: org.apache.spark.ml.feature.RFormula = RFormula(clicked ~ country + hour) (uid=rFormula_21602d03f353)
output: org.apache.spark.sql.DataFrame = [id: int, country: string ... 4 more fields]
+--------------+-----+
|      features|label|
+--------------+-----+
|[0.0,0.0,18.0]|  1.0|
|[1.0,0.0,12.0]|  0.0|
|[0.0,1.0,15.0]|  0.0|
+--------------+-----+
```

特征的选择及ML实现详解

- **3、ChiSqSelector（卡方特征选择器）**
- ChiSqSelector代表卡方特征选择。它适用于带有类别特征的标签数据。ChiSqSelector使用卡方独立测试来决定选择哪些特征。它支持三种选择方法：numTopFeatures, percentile, fpr：
- numTopFeatures根据卡方检验选择固定数量的顶级功能。这类似于产生具有最大预测能力的功能。
- percentile类似于numTopFeatures，但选择所有功能的一部分，而不是固定数量。
- fpr选择p值低于阈值的所有特征，从而控制选择的假阳性率。
- 默认情况下，选择方法是numTopFeatures，默认的顶级功能数量设置为50.用户可以使用setSelectorType选择一种选择方法。

特征的选择及ML实现详解

3、ChiSqSelector（卡方特征选择器）

Examples

假设我们有一个具有列id, features和clicked的DataFrame，这被用作我们预测的目标：

```
id | features              | clicked
---|-----------------------|---------
 7 | [0.0, 0.0, 18.0, 1.0] | 1.0
 8 | [0.0, 1.0, 12.0, 0.0] | 0.0
 9 | [1.0, 0.0, 15.0, 0.1] | 0.0
```

如果我们使用ChiSqSelector并设置numTopFeatures=1，根据我们所有的特征，其中最后一列标签clicked是认为最有用的特征：

```
id | features              | clicked | selectedFeatures
---|-----------------------|---------|------------------
 7 | [0.0, 0.0, 18.0, 1.0] | 1.0     | [1.0]
 8 | [0.0, 1.0, 12.0, 0.0] | 0.0     | [0.0]
 9 | [1.0, 0.0, 15.0, 0.1] | 0.0     | [0.1]
```

特征的选择及ML实现详解

```
    val data = Seq(
      (7, Vectors.dense(0.0, 0.0, 18.0, 1.0), 1.0),
      (8, Vectors.dense(0.0, 1.0, 12.0, 0.0), 0.0),
      (9, Vectors.dense(1.0, 0.0, 15.0, 0.1), 0.0))
    val df = spark.createDataset(data).toDF("id", "features", "clicked")
    val selector = new ChiSqSelector()
      .setNumTopFeatures(1)
      .setFeaturesCol("features")
      .setLabelCol("clicked")
      .setOutputCol("selectedFeatures")
    val result = selector.fit(df).transform(df)
    println(s"ChiSqSelector output with top ${selector.getNumTopFeatures}
features selected")
    result.show()
```

```scala
val data = Seq(
  (7, Vectors.dense(0.0, 0.0, 18.0, 1.0), 1.0),
  (8, Vectors.dense(0.0, 1.0, 12.0, 0.0), 0.0),
  (9, Vectors.dense(1.0, 0.0, 15.0, 0.1), 0.0))

val df = spark.createDataset(data).toDF("id", "features", "clicked")

val selector = new ChiSqSelector()
  .setNumTopFeatures(1)
  .setFeaturesCol("features")
  .setLabelCol("clicked")
  .setOutputCol("selectedFeatures")

val result = selector.fit(df).transform(df)

println(s"ChiSqSelector output with top ${selector.getNumTopFeatures} features selected")
result.show()
```

```
data: Seq[(Int, org.apache.spark.ml.linalg.Vector, Double)] = List((7,[0.0,0.0,18.0,1.0],1.0), (8,[0.0,1.0,1
df: org.apache.spark.sql.DataFrame = [id: int, features: vector ... 1 more field]
selector: org.apache.spark.ml.feature.ChiSqSelector = chiSqSelector_128c3fc7d9d9
result: org.apache.spark.sql.DataFrame = [id: int, features: vector ... 2 more fields]
ChiSqSelector output with top 1 features selected
+---+-----------------+-------+----------------+
| id|         features|clicked|selectedFeatures|
+---+-----------------+-------+----------------+
|  7|[0.0,0.0,18.0,1.0]|    1.0|         [18.0]|
|  8|[0.0,1.0,12.0,0.0]|    0.0|         [12.0]|
|  9|[1.0,0.0,15.0,0.1]|    0.0|         [15.0]|
+---+-----------------+-------+----------------+
```

Thanks

FAQ时间