

数据预处理

陶时



東北大學
Northeastern University



HORIZON
昊宸科技



The image features a clear blue sky as the background. In the foreground, several wind turbines are visible, their white towers and blades with orange and white striped tips reaching upwards. Overlaid on the center of the image is a graphic consisting of four overlapping circles in shades of blue and teal. The text "ML Pipeline" is written in a white, serif font across the middle of these circles.

ML Pipeline

- 受 scikit-learn 项目的启发，并且总结了MLlib在处理复杂机器学习问题的弊端(主要为工作繁杂，流程不清晰)
- 旨在向用户提供基于DataFrame 之上的更加高层次的 API 库，以更加方便的构建复杂的机器学习工作流式应用。
- 一个pipeline 在结构上会包含一个或多个Stage，每一个 Stage 都会完成一个任务，如数据集处理转化，模型训练，参数设置或数据预测等，这样的Stage 在 ML 里按照处理问题类型的不同都有相应的定义和实现。两个主要的stage为Transformer和Estimator。
- Transformer主要是用来操作一个DataFrame 数据并生成另外一个DataFrame 数据，比如svm模型、一个特征提取工具，都可以抽象为一个Transformer。 Estimator 则主要是用来做模型拟合用的，用来生成一个Transformer。

➤ 以下四个步骤可以抽象为一个包括多个步骤的流水线式工作，从数据收集开始至输出我们需要的最终结果。因此，对以下多个步骤、进行抽象建模，简化为流水线式工作流程则存在着可行性，对使用spark机器学习算法的用户来说，流水线式机器学习比单个步骤独立建模更加高效、易用。

- 1、源数据ETL
- 2、数据预处理
- 3、特征选取
- 4、模型训练与验证

Pipeline 主要概念(Basic Concepts)



➤ DataFrame: DataFrame 用作数据源支持丰富的数据类型，DataFrame 可以很方便的把特征，真正的标签或者预测值作放在DataFrame的不同列。DataFrame可以和RDD互操作。（RDD2DF）

➤ Transformer: Transformer可以是一种算法，将一个DF转换为另外一个DF,ML 的Model 是一个Transformer 将特征转换为预测值。

Transformer实现了transform方法，将一个DF增加一些列后转换为其他DF。

Transformer读取DF中的特征向量并预测每个向量的标签，并将带标签的DF输出。

➤ Estimator : 是学习算法或者其他算法的抽象，用来训练数据。预测器继承fit方法，可以接收一个DataFrame输入，然后产出一个模型。例如，像逻辑回归算法是一种Estimator，调用fit方法来训练一个逻辑回归模型。

➤ Pipeline : 在机器学习中，很常见的一种现象就是运行一系列的Estimator 和 Transformer来学习数据。

Pipeline 主要概念 (Basic Concepts)



➤ Parameters: MLlib的预测和转换使用统一的API来指定参数。一个参数是一个包含参数属性值的属性名称，一个ParamMap由一组（参数名称，值）这样的对组成。有两种传参的方法：

(1)为实例设置参数。

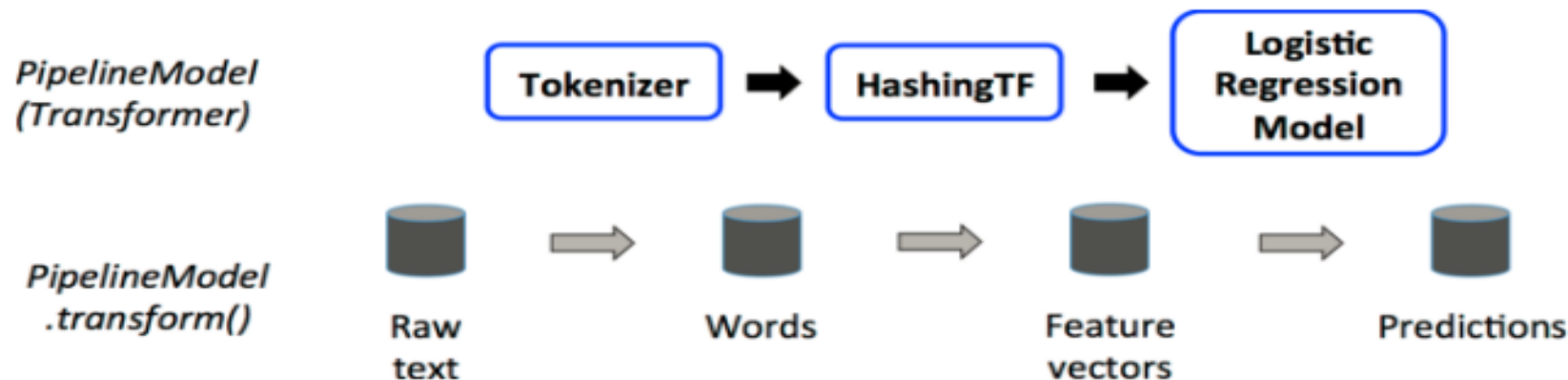
(2) 将参数封装到ParamMap，然后解析ParamMap为fit和transform传递参数。ParamMap的参数将覆盖原始的参数。

Pipeline 由一系列有顺序的阶段指定，每个状态时转换器或估计器。每个状态的运行是有顺序的，输入的数据框通过每个阶段进行改变。在转换器阶段，transform()方法被调用于数据框上。对于估计器阶段，fit()方法被调用来产生一个转换器，然后该转换器的transform()方法被调用在数据框上。



第一行代表 **Pipeline** 处理的三个阶段。第一二个蓝色的阶段是 **Transformer**，第三个红色框中的逻辑回归是 **Estimator**。底下一行代表管道中的数据流，圆筒指 **DataFrame**。**Pipeline** 的 `fit()` 方法被调用于原始的 **DataFrame** 中，里面包含原始的文档和标签。**Tokenizer** 的 `transform()` 方法将原始文档分为词语，添加新的词语列到数据框中。**HashingTF** 的 `transform()` 方法将词语列转换为特征向量，添加新的向量列到 **DataFrame** 中。然后，因为逻辑回归是 **Estimator**，**Pipeline** 先调用逻辑回归的 `fit()` 方法来产生逻辑回归模型。如果 **Pipeline** 还有其它更多阶段，在将 **DataFrame** 传入下一个阶段之前，**Pipeline** 会先调用逻辑回归模型的 `transform()` 方法。

PipelineModel



上面的图示中，PipelineModel 和 Pipeline有同样数目的阶段，然而Pipeline中的Estimator此时变为了Transformer。当PipelineModel的transform()方法被调用于测试数据集时，数据依次经过管道的各个阶段。每个阶段的transform()方法更新数据集，并将之传到下个阶段。

Extracting, transforming and selecting features



This section covers algorithms for working with features, roughly divided into these groups:

Extraction: Extracting features from “raw” data

Transformation: Scaling, converting, or modifying features

Selection: Selecting a subset from a larger set of features

- 特征提取
- 特征转换
- 特征选择

- Feature Extractors
 - TF-IDF (HashingTF and IDF)
 - Word2Vec
 - CountVectorizer
- Feature Transformers
 - Tokenizer
 - StopWordsRemover
 - n -gram
 - Binarizer
 - PCA
 - PolynomialExpansion
 - Discrete Cosine Transform (DCT)
 - StringIndexer
 - IndexToString
 - OneHotEncoder
 - VectorIndexer
 - Normalizer
 - StandardScaler
 - MinMaxScaler
 - Bucketizer
 - ElementwiseProduct
 - SQLTransformer
 - VectorAssembler
 - QuantileDiscretizer
- Feature Selectors
 - VectorSlicer
 - RFormula
 - ChiSqSelector

特征提取



Feature Extractors下的api主要针对文本处理的api

TF-IDF (HashingTF and IDF)

Word2Vec

CountVectorizer

$$TF_w = \frac{\text{在某一类中词条}w\text{出现的次数}}{\text{该类中所有的词条数目}}$$

$$IDF = \log\left(\frac{\text{语料库的文档总数}}{\text{包含词条}w\text{的文档数} + 1}\right), \text{分母之所以要加1, 是为了避免分母为0}$$

$$TF - IDF = TF * IDF$$

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

➤ 词语由 t 表示，文档由 d 表示，语料库由 D 表示。词频 $TF(t,d)$ 是词语 t 在文档 d 中出现的次数。文件频率 $DF(t,D)$ 是包含词语的文档的个数。如果我们只使用词频来衡量重要性，很容易过度强调在文档中经常出现而并没有包含太多与文档有关的信息的词语，比如“a”，“the”以及“of”。如果一个词语经常出现在语料库中，它意味着它并没有携带特定的文档的特殊信息。逆向文档频率数值化衡量词语提供多少信息：

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

其中， $|D|$ 是语料库中的文档总数。由于采用了对数，如果一个词出现在所有的文件，其IDF值变为0。

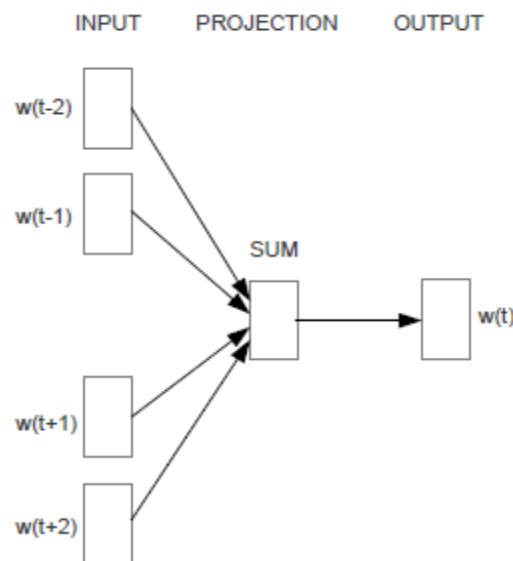
$$TFIDF(t, d, D) = TF(t, d) \bullet IDF(t, D)$$

- Word2vec是由谷歌发布开源的自然语言处理算法，其目的是把words转换成vectors，从而可以用数学的方法来分析words之间的关系。Spark其该算法进行了封装，并在 mllib 中实现。
- Word2vec是一个Estimator，它采用一系列代表文档的词语来训练word2vecmodel。该模型将每个词语映射到一个固定大小的向量。word2vecmodel使用文档中每个词语的平均数来将文档转换为向量，然后这个向量可以作为预测的特征。

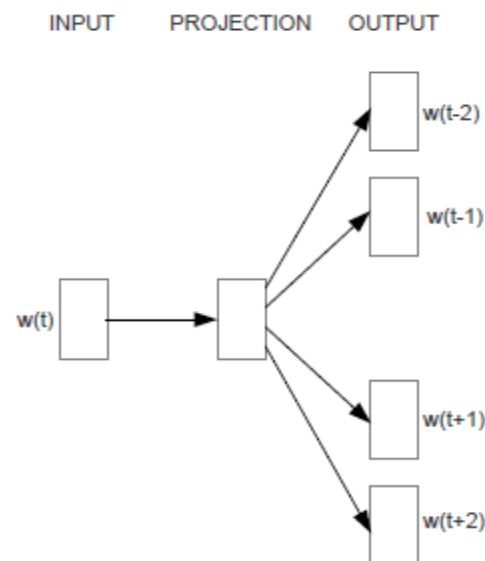
特征提取- Word2Vec



- Word2vec 提供了两种模型 Continuous Bag-of-Words Model (CBOW) 和 Continuous Skip-gram Model (Skip-gram)。
- CBOW根据语境预测目标单词，使用围绕目标单词的其他单词（语境）作为输入，在映射层做加权处理后输出目标单词。Skip-gram其思想是通过每个中心词来预测其上下文窗口词，并根据预测结果来修正中心词的词向量。
- 。 skip-gram慢，对罕见字有利；CBOW快。



CBOW



Skip-Gram

- Countvectorizer 和 Countvectorizermodel 旨在通过计数来将一个文档转换为向量。当不存在先验字典时，Countvectorizer可作为Estimator来提取词汇，并生成一个Countvectorizermodel。该模型产生文档关于词语的稀疏表示，其表示可以传递给其他算法如LDA。
- 在fitting过程中，countvectorizer 将根据语料库中的词频排序选出前vocabsize个词。一个可选的参数minDF也影响fitting过程中，它指定词汇表中的词语在文档中最少出现的次数。另一个可选的二值参数控制输出向量，如果设置为真那么所有非零的计数为1。这对于二值型离散概率模型非常有用。

特征转换



特征转换—Tokenizer/分词器

- 中文中单个字往往表达不了语义，一句话往往能通过几个词提出主干，充分表达其含义。

因此NLP相关的业务之中分词往往是非常重要的预处理过程。

今天/天气/不错

what a nice weather today

分词方式① 启发式 ②统计方法 HMM CRF

- Spark 中提供Tokenizer作为英文文本的切分工具。中文NLP可以通过分词器实现

<https://github.com/fxsjy/jieba> --python api

https://github.com/NLPchina/ansj_seg --java api

特征转换—Tokenizer



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

val sentenceDataFrame = sqlContext.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)
```

- Tokenization将文本划分为独立个体（通常为单词）。
- RegexTokenizer 基于正则表达式提供更多的划分选项。默认情况下，参数“pattern”为划分文本的分隔符。或者可以指定参数“gaps”来指明正则“pattern”表示“tokens”而不是分隔符，这样来为分词结果找到所有可能匹配的情况。

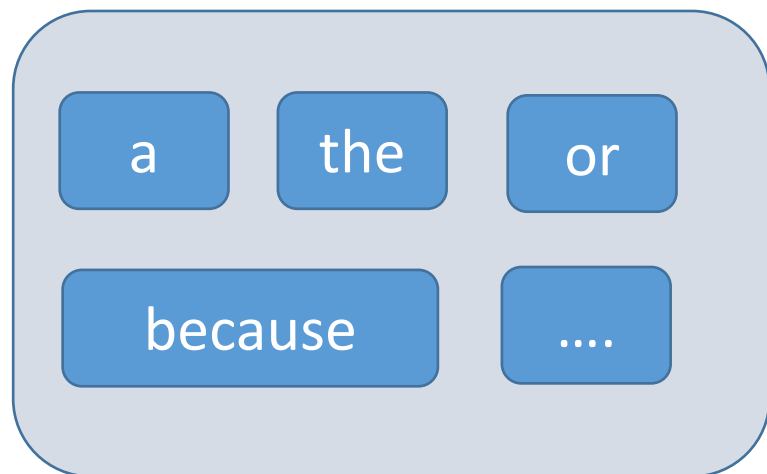
特征转换—StopWordsRemover



➤ 停用词：是由英文单词:stopword翻译过来的。

原来在英语里面会遇到很多a, the, or等使用频率很多的字或词，常为冠词、介词、副词或连词等。如果搜索引擎要将这些词都索引的话，那么几乎每个网站都会被索引，也就是说工作量巨大

➤ 停用词为在文档中频繁出现，但未承载太多意义的词语，他们不应该被包含在算法输入中。



特征转换—StopWordsRemover

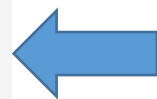
- 停用词为在文档中频繁出现，但未承载太多意义的词语，他们不应该被包含在算法输入中。
- StopWordsRemover的输入为一系列字符串（如分词器输出），输出中删除了所有停用词。停用词表由stopWords参数提供。一些语言的默认停用词表可以通过以下语句：

```
StopWordsRemover.loadDefaultStopWords(language)
```

布尔参数 caseSensitive 指明是否区分大小写（默认为否）

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, balloon]	[saw, red, balloon]
1	[Mary, had, a, little, lamb]	[Mary, little, lamb]



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

特征转换—StopWordsRemover



```
import org.apache.spark.ml.feature.StopWordsRemover

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

val dataset = sqlContext.createDataFrame(Seq(
  (0, Seq("I", "saw", "the", "red", "balloon")),
  (1, Seq("Mary", "had", "a", "little", "lamb"))
)).toDF("id", "raw")

remover.transform(dataset).show()
```

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



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

1.Tokenizer: 转小写, 以空格分隔成数组

label	sentence	words
0	Hi I heard about Spark	[hi, i, heard, about, spark]
0	I wish Java could use case classes	[i, wish, java, could, use, case, classes]
1	Logistic regression models are neat	[logistic, regression, models, are, neat]

2.StopWordsRemover:

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

特征转换—n-gram



```
import org.apache.spark.ml.feature.NGram

val wordDataFrame = sqlContext.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")
val ngramDataFrame = ngram.transform(wordDataFrame)
ngramDataFrame.take(3).map(_.getAs[Stream[String]]("ngrams").toList).foreach(println)
```

```
List(Hi I, I heard, heard about, about Spark)
```

```
List(I wish, wish Java, Java could, could use, use case, case classes)
```

```
List(Logistic regression, regression models, models are, are neat)
```

- 一个n-gram是一个长度为整数n的字序列。Ngram 可以用来将输入转换为n-gram。
- NGram的输入为一系列字符串（如分词器输出）。参数n决定每个n-gram包含的对象个数。结果包含一系列n-gram，其中每个n-gram代表一个空格分割的n个连续字符。如果输入少于n个字符串，将没有输出结果。

将相邻的N个元素组成一个维度，下图N=3

+-----+-----+-----+-----+	
label words	ngrams
+-----+-----+-----+-----+	
0	[Hi, I, heard, about, Spark] [Hi I heard, I heard about, heard about Spark]
1	[I, wish, Java, could, use, case, classes] [I wish Java, wish Java could, Java could use, could use case, use case classes]
2	[Logistic, regression, models, are, neat] [Logistic regression models, regression models are, models are neat]
+-----+-----+-----+-----+	

特征转换—Binarizer—特征的类型

- 数值特征(numerical feature): 实数或者整数，比如说年龄或者高度。
- 类别特征(categorical feature): 它们的取值只能是可能状态的一种，比如性别
- 文本特征(text feature): 它们是派生自数据的文本内容，比如电影名，描述，或者评论
- 其他特征：大部分其他特征都表示为数字，比如图像，视频。

特征转换—Binarizer



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

val data = Array((0, 0.1), (1, 0.8), (2, 0.2))
val dataframe: DataFrame = sqlContext.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)
```

- 二元优化方法是根据阈值将连续数值特征转换为0-1特征的过程。
- Binarizer参数有输入、输出以及阈值。特征值大于阈值将映射为1.0，特征值小于等于阈值将映射为0.0。

label	feature	binarized_feature
0	0.1	0.0
1	0.8	1.0
2	0.2	0.0



label	feature
0	0.1
1	0.8
2	0.2

特征转换—PCA



```
import org.apache.spark.ml.feature.PCA
import org.apache.spark.mllib.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 = sqlContext.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()
```

```
[[1.6485728230883807, -4.013282700516296, -5.524543751369388]]
[[-4.645104331781534, -1.1167972663619026, -5.524543751369387]]
[[-6.428880535676489, -5.337951427775355, -5.524543751369389]]
```

- 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。
- 使用PCA方法可以对变量集合进行降维。示例介绍如何将5维特征向量转换为3维主成分向量。



```
+-----+
|          features          |
+-----+
| (5, [1, 3], [1.0, 7.0]) |
| [2.0, 0.0, 3.0, 4.0, ...] |
| [4.0, 0.0, 0.0, 6.0, ...] |
+-----+
```

特征转换—PolynomialExpansion




```
import org.apache.spark.ml.feature.PCA
import org.apache.spark.mllib.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 = sqlContext.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()
```

```
[[[-2.0,4.0,-8.0,2.3,-4.6,9.2,5.289999999999999,-10.579999999999998,12.166999999999996]]
[[0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0]]
[[0.6,0.36,0.216,-1.1,-0.66,-0.396,1.2100000000000002,0.7260000000000001,-1.3310000000000004]]
```

- 多项式扩展通过产生n维组合将原始特征将特征扩展到多项式空间。下面的示例会介绍如何将特征集拓展到3维多项式空间。



```
+-----+
| features|
+-----+
| [-2.0,2.3]|
| [0.0,0.0]|
| [0.6,-1.1]|
+-----+
```

1.PCA: 将数据降到K维, 下图K=3

features	pcaFeatures
[2.0, 1.0, 3.0, 9.0, 5.0]	[7.11854802755693, -5.687985566873541, -5.96167313955506]
[2.0, 0.0, 3.0, 4.0, 5.0]	[2.7269440414332804, -3.096882630079468, -5.961673139555059]
[4.0, 0.0, 0.0, 6.0, 7.0]	[2.3113780016027357, -7.660576842725574, -5.961673139555062]

2.PolynomialExpansion: 将数据拓展到多项式空间, 下图N=2 (x, x * x, y, x * y, y * y)

features	polyFeatures
[-2.0, 2.3]	[-2.0, 4.0, 2.3, -4.6, 5.289999999999999]
[0.0, 0.0]	[0.0, 0.0, 0.0, 0.0, 0.0]
[0.6, -1.1]	[0.6, 0.36, -1.1, -0.66, 1.2100000000000002]

特征转换—Discrete Cosine Transform



```
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 = sqlContext.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)
```


- 离散余弦变换是与傅里叶变换相关的一种变换，它类似于离散傅立叶变换但是只使用实数。离散余弦变换相当于一个长度大概是它两倍的离散傅里叶变换，这个离散傅里叶变换是对一个实偶函数进行的（因为一个实偶函数的傅里叶变换仍然是一个实偶函数）。离散余弦变换，经常被信号处理和图像处理使用，用于对信号和图像（包括静止图像和运动图像）进行有损数据压缩。

特征转换—StringIndexer and IndexToString



- StringIndexer将一系列labels转译成 $[0, \text{labels基数})$ 的index，labels基数即为labels的去重后总量，index的顺序为labels频次升序，因此出现最多次labels的index为0。如果输入的列是数字类型，我们会把它转化成string，并且使用string转译成index。当pipeline的下游组件例如Estimator或者Transformer使用生成的index时，需要将该组件的输入列名称设置为index的列名。在多数情况下，你可以使用setInputCol设置列名。另外StringIndexer fit了一个dataset后，transformer一个dataset遇到没见过的labels时，有两种处理策略：抛出异常（默认）跳过整行数据，setHandleInvalid(“skip”)
- 与StringIndexer对称的，IndexToString将index映射回原先的labels。通常我们使用StringIndexer产生index，然后使用模型训练数据，最后使用IndexToString找回原先的labels。

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



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

特征转换—OneHotEncoder



➤ 有以下语句。

Join likes to watch movie.Mary likes too.

Join also likes to watch football games.

词典{ "Join" : 1, "likes " :2 , "to" :3 , "watch " :4 , "movie" :5 , "also " :6 , "football " :7
 , "games" :8 , "Mary " :9 , "too" :10}

John: [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]

likes: [0, 1, 0, 0, 0, 0, 0, 0, 0, 0]

....

特征转换—OneHotEncoder



- 独热编码将标签指标映射为二值向量，其中最多一个单值。这种编码被用于将种类特征使用到需要连续特征的算法，如逻辑回归等。

```
val df = sqlContext.createDataFrame(Seq(
  (0, "a"),
  (1, "b"),
  (2, "c"),
  (3, "a"),
  (4, "a"),
  (5, "c")
)).toDF("id", "category")

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

val encoder = new OneHotEncoder()
  .setInputCol("categoryIndex")
  .setOutputCol("categoryVec")
val encoded = encoder.transform(indexed)
encoded.select("id", "categoryVec").show()
```

id	category	categoryIndex	categoryVec
0	a	0.0	(2, [0], [1.0])
1	b	2.0	(2, [], [])
2	c	1.0	(2, [1], [1.0])
3	a	0.0	(2, [0], [1.0])
4	a	0.0	(2, [0], [1.0])
5	c	1.0	(2, [1], [1.0])

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

1. 获得一个向量类型的输入以及maxCategories参数。
2. 基于不同特征值的数量来识别哪些特征需要被类别化，其中最多maxCategories个特征需要被类别化。
3. 对于每一个类别特征计算0-based（从0开始）类别索引。
4. 对类别特征进行索引然后将原始特征值转换为索引。

索引后的类别特征可以帮助决策树等算法恰当的处理类别型特征，并得到较好结果。

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

特征转换—VectorIndexer



```
val data = Seq(Vectors.dense(-1.0, 1.0, 1.0), Vectors.dense(-1.0, 3.0, 1.0), Vectors.dense(0.0, 5.0, 1.0))
val df = sqlContext.createDataFrame(data.map(Tuple1.apply)).toDF("features")
val indexer = new VectorIndexer().|
    setInputCol("features").
    setOutputCol("indexed").
    // 设置条件: 只有种类小于2的特征才被认为是类别型特征
    setMaxCategories(2)
val indexerModel = indexer.fit(df)
val categoricalFeatures: Set[Int] = indexerModel.categoryMaps.keys.toSet
println(s"Chose ${categoricalFeatures.size} categorical features: " + categoricalFeatures.mkString(", "))
// [features: vector, indexed: vector] indexData 包含两部分 features vector 和 indexed vector
val indexedData = indexerModel.transform(df)
indexedData.foreach { println }
```

- 从上例可以看到，我们设置maxCategories为2，即只有种类小于2的特征才被认为是类别型特征，否则被认为是连续型特征。于是，我们可以看到第0类和第2类的特征由于种类数不超过2，被划分成类别型特征，并进行了索引，且为0的特征值也被编号成了0号。

特征转换—Normalizer



- Normalizer是一个转换器，它可以将多行向量输入转化为统一的形式。参数为p（默认值：2）来指定正则化中使用的p-norm。正则化操作可以使输入数据标准化并提高后期学习算法的效果。

```
val dataframe = sqlContext.read.format("libsvm").load("data/mllib/sample_libsvm_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()

// Normalize each vector using  $L^\infty$  norm.
val lInfNormData = normalizer.transform(dataframe, normalizer.p -> Double.PositiveInfinity)
lInfNormData.show()
```


特征转换—StandardScaler



- StandardScaler处理Vector数据，标准化每个特征使得其有统一的标准差以及（或者）均值为零。它需要如下参数：
- 1. withStd：默认值为真，使用统一标准差方式。
- 2. withMean：默认为假。此种方法将产出一个稠密输出，所以不适用于稀疏输入。
- StandardScaler是一个Estimator，它可以fit数据集产生一个StandardScalerModel，用来计算汇总统计。然后产生的模可以用来转换向量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。

特征标准化，计算公式： $(x - \text{mean}) / \text{std}$

withMean:是否减去均值

withStd:是否进行标准化

```
final val inputCol: Param[String]  
Param for input column name.
```

```
final val outputCol: Param[String]  
Param for output column name.
```

```
val withMean: BooleanParam  
Whether to center the data with mean before scaling.
```

```
val withStd: BooleanParam  
Whether to scale the data to unit standard deviation.
```

特征转换—StandardScaler



```
import org.apache.spark.ml.feature.StandardScaler

val dataframe = sqlContext.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()
```

```
final val inputCol: Param[String]
      Param for input column name.
```

```
final val outputCol: Param[String]
      Param for output column name.
```

```
val withMean: BooleanParam
      Whether to center the data with mean before scaling.
```

```
val withStd: BooleanParam
      Whether to scale the data to unit standard deviation.
```

特征标准化，计算公式： $(x - \text{mean}) / \text{std}$

withMean:是否减去均值

withStd:是否进行标准化

特征转换—MinMaxScaler



➤ MinMaxScaler通过重新调节大小将Vector形式的列转换到指定的范围内，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计算数据集的汇总统计量，并产生一个MinMaxScalerModel。该模型可以将独立的特征的值转换到指定的范围内。

对于特征 E 来说，调整后的特征值如下：

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

如果 $E_{\max} = E_{\min}$ ，则 $\text{Rescaled} = 0.5 * (\max - \min)$ 。

注意因为零值转换后可能变为非零值，所以即便为稀疏输入，输出也可能为稠密向量。

特征转换—Bucketizer



➤ Bucketizer将一系列连续的特征转换为特征区间，区间由用户指定。参数如下：

1. splits：分裂数为 $n+1$ 时，将产生 n 个区间。除了最后一个区间外，每个区间范围 $[x, y]$ 由分裂的 x, y 决定。分裂必须是严格递增的。在分裂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。

```
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 = sqlContext.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()
```

特征转换—QuantileDiscretizer

- QuantileDiscretizer将连续型特征转换为类别特征。分级的数量由numBuckets参数决定。分级的范围有渐进算法决定。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到正无穷，覆盖所有的实数值。

```
val data = Array((0, 18.0), (1, 19.0), (2, 8.0), (3, 5.0), (4, 2.2))
val df = sc.parallelize(data).toDF("id", "hour")

val discretizer = new QuantileDiscretizer()
    .setInputCol("hour")
    .setOutputCol("result")
    .setNumBuckets(3)

val result = discretizer.fit(df).transform(df)
result.show()
```

Binarizer:

设置阈值将特征二元化

```
inputCol: Param[String]
```

Param for input column name.

```
outputCol: Param[String]
```

Param for output column name.

```
threshold: DoubleParam
```

Param for threshold used to bina

threshold: 设置分隔的阈值

Bucketizer:

设置分隔区间进行离散化

```
val inputCol: Param[String]
```

Param for input column name.

```
val outputCol: Param[String]
```

Param for output column name.

```
val splits: DoubleArrayParam
```

Parameter for mapping continuou

splits: 设置分隔的阈值数组

quantileDiscretizer:

基于抽样数据进行等频离散

```
val inputCol: Param[String]
```

Param for input column name.

```
val numBuckets: IntParam
```

Maximum number of buckets (qual

```
val outputCol: Param[String]
```

Param for output column name.

numBuckets: 将数据离散成几个类别

特征转换—ElementwiseProduct



- ElementwiseProduct按提供的“weight”向量，返回与输入向量元素级别的乘积。也就是，按提供的权重分别对输入数据进行缩放，得到输入向量v以及权重向量w的Hadamard积。

$$\begin{pmatrix} v_1 \\ \vdots \\ v_N \end{pmatrix} \circ \begin{pmatrix} w_1 \\ \vdots \\ w_N \end{pmatrix} = \begin{pmatrix} v_1 w_1 \\ \vdots \\ v_N w_N \end{pmatrix}$$

```
| id |      vector |  
+---+-----+  
|  a | [1.0,2.0,3.0]|  
|  b | [4.0,5.0,6.0]|  
+---+-----+
```



[0.0,1.0,2.0]



```
·+-----+  
·|transformedVector|  
·+-----+  
||    [0.0,2.0,6.0] ||  
||    [0.0,5.0,12.0] ||  
·+-----+
```

向量特征变化-1



ElementwiseProduct: 向量相乘, 下图 $v_1=0$, $v_2=1$, $v_3=2$

id	vector	transformedVector
a	[1.0, 2.0, 3.0]	[0.0, 2.0, 6.0]
b	[4.0, 5.0, 6.0]	[0.0, 5.0, 12.0]

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

features	featuresDCT	recovery
[0.0, 1.0, -2.0, 3.0]	[0.46507563265748386, -0.6892463972414662, 2.6892463972414666, -2.4650756326574843]	[0.45984444731456486, 0.3117941502192956, -2.535964914803279, 2.694477582584386]
[-1.0, 2.0, 4.0, -7.0]	[0.9123766143646872, 2.6141664772135154, -7.614166477213516, 2.0876233856353137]	[-1.078201562410237, 3.6068541969490733, 4.9196888946291315, -5.623588300438593]
[14.0, -2.0, -5.0, 1.0]	[3.46403508519672, 8.305522417415615, 10.694477582584385, 5.535964914803277]	[14.003121642382286, -4.984306443971243, -2.246136053416424, 0.15539102539882155]

向量特征变化-2



Normalizer: $p=1$, $p=\text{负无穷}$

features	normFeatures
[5.0, 0.0, 6.0, 4.0, 9.0]	[0.20833333333333334, 0.0, 0.25, 0.16666666666666666, 0.375]
[2.0, 0.0, 3.0, 4.0, 5.0]	[0.14285714285714285, 0.0, 0.21428571428571427, 0.2857142857142857, 0.35714285714285715]
[4.0, 0.0, 0.0, 6.0, 7.0]	[0.23529411764705882, 0.0, 0.0, 0.35294117647058826, 0.4117647058823529]

features	normFeatures
[5.0, 0.0, 6.0, 4.0, 9.0]	[0.5555555555555556, 0.0, 0.6666666666666666, 0.4444444444444444, 1.0]
[2.0, 0.0, 3.0, 4.0, 5.0]	[0.4, 0.0, 0.6, 0.8, 1.0]
[4.0, 0.0, 0.0, 6.0, 7.0]	[0.5714285714285714, 0.0, 0.0, 0.8571428571428571, 1.0]

特征转换—SQLTransformer



- SQLTransformer 通过SQL 语法实现数据转换

id	v1	v2
0	1.0	3.0
2	2.0	5.0

```
SELECT *, (v1 + v2) AS v3, (v1 * v2) AS v4  
FROM __THIS__
```



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

```
val df = sqlContext.createDataFrame(  
  Seq((0, 1.0, 3.0), (2, 2.0, 5.0))).toDF("id", "v1", "v2")  
//改表名  
df.registerTempTable("a")
```

```
val sqlTrans = new SQLTransformer().setStatement(  
  "SELECT *, (v1 + v2) AS v3, (v1 * v2) AS v4 FROM a")  
  
sqlTrans.transform(df).show()
```

特征转换—VectorAssembler



- VectorAssembler 是一个转换器，它将给定的若干列合并为一列向量。它可以将原始特征和一系列通过其他转换器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。

```
final val inputCols: StringArrayParam  
          Param for input column names.
```

```
final val outputCol: Param[String]  
                  Param for output column name.
```

inputCols: 构成vector的列的列名
outputCol: 输出Vector列

```
val dataset = sqlContext.createDataFrame(  
  Seq((0, 18, 1.0, Vectors.dense(0.0, 10.0, 0.5), 1.0))  
)  
.toDF("id", "hour", "mobile", "userFeatures", "clicked")
```

```
val assembler = new VectorAssembler()  
  .setInputCols(Array("hour", "mobile", "userFeatures"))  
  .setOutputCol("features")
```

```
val output = assembler.transform(dataset)  
println(output.select("features", "clicked").first())
```

特征选择



Feature Selectors—VectorSlicer



➤ **VectorSlicer**是一个转换器。输入特征向量，输出原始特征向量子集。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

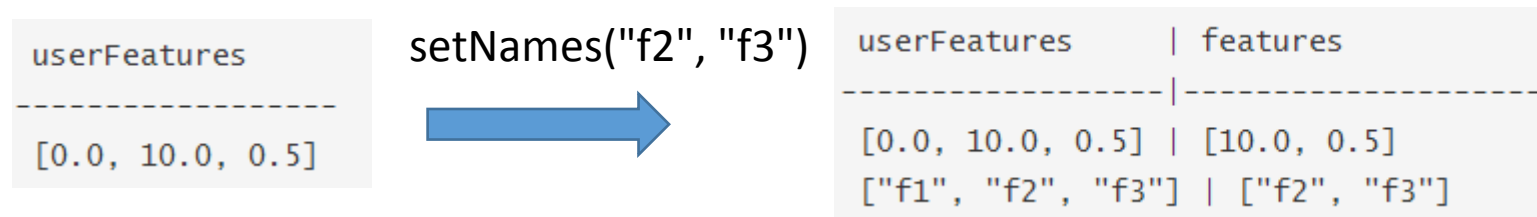
1. 整数索引，`setIndices()`。

2. 字符串索引代表向量中特征的名字，此类要求向量列有`AttributeGroup`，因为该工具根据`Attribute`来匹配名字字段。

例如：假设我们有一个DataFrame含有userFeatures列：



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



Feature Selectors—RFormula



➤ RFormula通过R模型公式来选择列。支持R操作中的部分操作，包括 '~' ,
'.' , ':' , '+' 以及 '-' , 基本操作如下：

1. ~ 分隔目标和对象
2. + 合并对象，“+ 0”意味着删除空格
3. : 交互（数值相乘，类别二值化）
4. . 除了目标外的全部列

假设a和b为两列：

1. $y \sim a + b$ 表示模型 $y \sim w_0 + w_1 * a + w_2 * b$ 其中 w_0 为截距， w_1 和 w_2 为相关系数。
2. $y \sim a + b + a:b - 1$ 表示模型 $y \sim w_1 * a + w_2 * b + w_3 * a * b$ ，其中 w_1 ， w_2 ， w_3 是相关系数。

RFormula产生一个向量特征列以及一个double或者字符串标签列。如果类别列是字符串类型，它将通过StringIndexer转换为double类型。如果标签列不存在，则输出中将通过规定的响应变量创建一个标签列。

关于R语言Model Formulae的介绍可参考：<https://stat.ethz.ch/R-manual/R-devel/library/stats/html/formula.html>

Feature Selectors—RFormula



- 假设有一个DataFrame 含有id, country, hour和clicked四列

如果我们使用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
7	"US"	18	1.0
8	"CA"	12	0.0
9	"NZ"	15	0.0

`clicked ~ country + hour`



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

$y \sim a + b$ 表示模型 $y \sim w_0 + w_1 * a + w_2 * b$
其中 w_0 为截距， w_1 和 w_2 为相关系数

Feature Selectors—ChiSqSelector



- 特征选择方法和分类方法一样，也主要分为有监督（Supervised）和无监督（Unsupervised）两种，卡方选择则是统计学上常用的一种有监督特征选择方法，它通过对特征和真实标签之间进行卡方检验，来判断该特征和真实标签的关联程度，进而确定是否对其进行选择。

id	features	label	selected-feature
1	[0.0,0.0,18.0,1.0]	1.0	[18.0,1.0]
2	[0.0,1.0,12.0,0.0]	0.0	[12.0,0.0]
3	[1.0,0.0,15.0,0.1]	0.0	[15.0,0.1]

```
val df = sqlContext.createDataFrame(Seq(
  (1, Vectors.dense(0.0, 0.0, 18.0, 1.0), 1.0),
  (2, Vectors.dense(0.0, 1.0, 12.0, 0.0), 0.0),
  (3, Vectors.dense(1.0, 0.0, 15.0, 0.1), 0.0)
)).toDF("id", "features", "label")
df.show()
val selector = new ChiSqSelector().
  // 我们设置只选择和标签关联性最强的两个特征
  setNumTopFeatures(2).
  setFeaturesCol("features").
  setLabelCol("label").
  setOutputCol("selected-feature")
val selector_model = selector.fit(df)
val result = selector_model.transform(df)
result.show(false)
```


习题



1.通过招聘网站爬取的招聘信息对招聘信息进行分类

结合文本相关spark 算法

①.通过spark相关API将爬取到的数据进行处理，得到结构化的数据表

②.分析某几个条件下的分类数量排名（1-3年工作经验的大数据工程师的平均薪资情况）

③ 将 dscr 描述字段提纯后通过算法对用人需求记录打标签。

把算法执行结果，简要的处理思路列在报告上！

```
title = Field() # 急招5KJava软件工程师/双休/五险一金
id = Field() # url
slry = Field() # 职位月薪: 5000-8000元/月
expr = Field() # 工作经验: 不限
need = Field() # 招聘人数: 5人
addr = Field() # 工作地点: 广州-黄埔区
fupa = Field() # 工作性质: 全职
qual = Field() # 最低学历: 不限
cate = Field() # 职位类别: 系统架构设计师

dscr = Field() # 职位描述: ...

comp = Field() # 公司名称
scle = Field() # 公司规模: 100-499人
prop = Field() # 公司性质: 民营
indu = Field() # 公司行业: IT服务(系统/数据/维护)
hmpg = Field() # 公司主页: www.phxg.cn
loca = Field() # 公司地址:
北京市丰台区南四环西路186号四区汉威国际广场6号楼9层
```

THANKS

