# SparkMLlib的简介

MLLIB是Spark的机器学习库。提供了利用Spark构建大规模和易用性的机器学习平台，组件：

1. ML 算法：包括了分类、聚类、降维、协同过滤
2. Featurization特征化：特征抽取、特征转换、特征降维、特征选择
3. Pipelines管道：tools for constructing, evaluating, and tuning ML Pipelines
4. Persistence持久化：模型的保存、读取、管道操作
5. Utilities：提供了线性代数、统计学以及数据处理工具

# **基于DataFrame的API是主要API**

**\*Spark ml基于DataFrame的API**

**\*Spark mllib基于RDD的API**

****基于MLlib RDD的API现在处于维护模式。****

从Spark 2.0开始，软件包中基于[RDD](http://spark.apache.org/docs/latest/rdd-programming-guide.html" \l "resilient-distributed-datasets-rdds)的API spark.mllib已进入维护模式。Spark的主要机器学习API现在是[包中](http://spark.apache.org/docs/latest/sql-programming-guide.html)基于[DataFrame](http://spark.apache.org/docs/latest/sql-programming-guide.html)的API spark.ml。

*有什么影响？*

* MLlib仍将支持基于RDD的API spark.mllib以及错误修复。
* MLlib不会为基于RDD的API添加新功能。
* 在Spark 2.x版本中，MLlib将为基于DataFrames的API添加功能，以实现与基于RDD的API的功能奇偶校验。
* 在达到功能奇偶校验（粗略估计Spark 2.3）之后，将弃用基于RDD的API。
* 预计将在Spark 3.0中删除基于RDD的API。

*为什么MLlib会切换到基于DataFrame的API？*

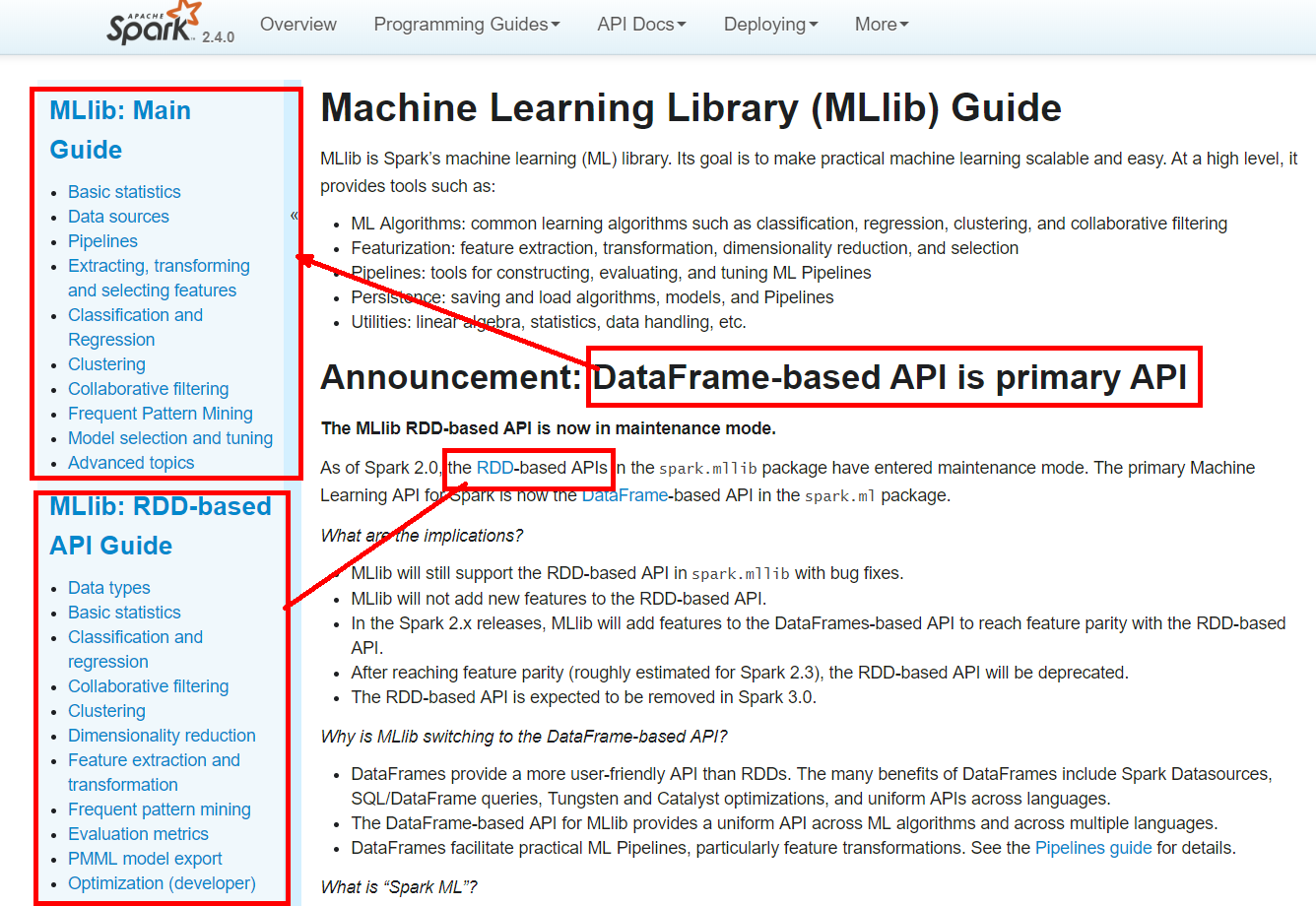
* + DataFrames提供比RDD更加用户友好的API。DataFrame的许多好处包括Spark数据源，SQL / DataFrame查询，Tungsten和Catalyst优化以及跨语言的统一API。
  + 基于DataFrame的MLlib API跨ML算法和多种语言提供统一的API。
  + DataFrames有助于实用的ML管道，特别是功能转换。有关详细信息，请参阅[管道指南](http://spark.apache.org/docs/latest/ml-pipeline.html)。

*什么是“Spark ML”？*

* “Spark ML”不是官方名称，但偶尔用于指代基于MLlib DataFrame的API。这主要是由于org.apache.spark.ml基于DataFrame的API使用的Scala包名称，以及我们最初用来强调管道概念的“Spark ML Pipelines”术语。

*MLlib已被弃用吗？*

* + MLlib包括基于RDD的API和基于DataFrame的API。基于RDD的API现在处于维护模式。但是这两种API都没有被弃用，整个也没有MLlib。
  + 两套API的介绍



# SparkMLlib的环境支持

1. Spark单机版本

首先在官网下载Spark预编译版本，将lib目录下的spark-assembly-1.3.0-hadoop2.4.0.jar文件复制到IDEA安装目录的lib文件夹下。

单击IDEA菜单上File选项，选择Project Strcture，在弹出的对话框单击左侧的Libraries,之后单击中部上方绿色+号，添加刚才下载的jar包文件即可。

1. Maven构建依赖环境

在pom文件中加入mllib包的依赖，保存后IDEA会帮我们自动下载。

<**dependency**>  
 <**groupId**>org.apache.spark</**groupId**>  
 <**artifactId**>spark-mllib\_2.10</**artifactId**>  
 <**version**>${spark.version}</**version**>  
</**dependency**>

\*\*代码基于2.0.2语法代码

# SparkMLLIB

MLlib是Spark机器学习库，它是MLBase的一部分，MLBase一共分为一下4部分：

* ML Optimizer：会选择它认为最适合的已经在内部实现好了的机器学习算法和相关参数来处理用户输入的数据，并返回模型或者其他的帮助分析结果。
* MLLIB是一个进行特征提取的和高级ML编程抽象的算法实现的API平台。
* MLLIB是Spark实现一些常见的机器学习算法和实用程序。
* MLRuntime是基于Spark计算框架，将Spark的分布式计算应用到机器学习领域。

MLlib提供了常见机器学习算法的实现，包括分类、聚类、协同过滤和降维等。使用MLlib来做机器学习工作通常只需要在对原始数据处理之后，然后直接调用相应的API就可以实现了。但是想要选择合适的算法，必须了解算法的原理以及MLlib API。接下来比较下Spark.mllib和spark.ml

Spark.mllib已经很长时间了，1.0之前的版本已经包含了，提供算法实现都是基于原始的RDD，我们只需要掌握mllib的API就可以完成机器学习工作。但是想要构建完整并且复杂的机器学习流水线是比较困难的，因此有了Spark.ml。

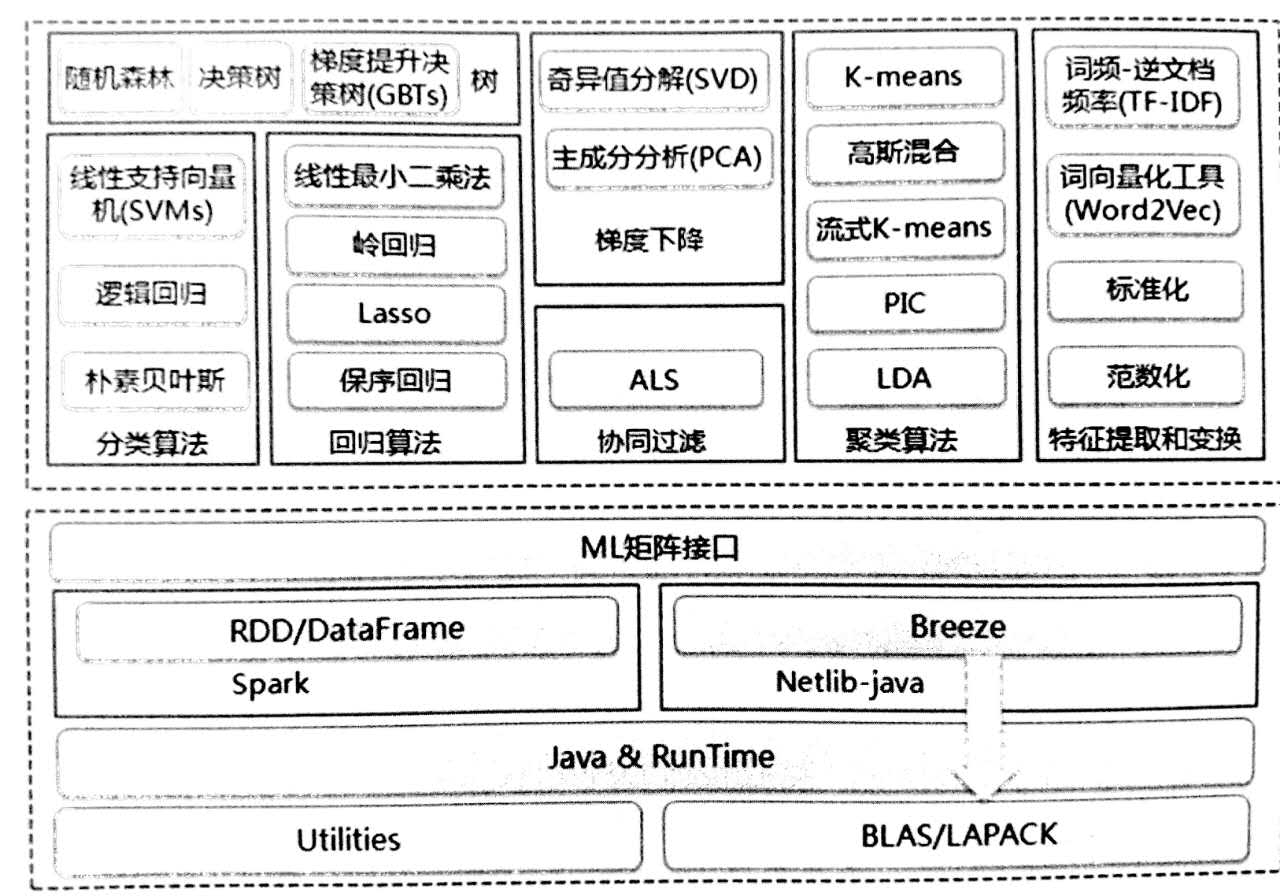
Spark ML Pipeline从Spark1.2版本开始，目前已经从Alpha阶段毕业，成为可用的并且较为稳定的新的机器学习库。ML Pipeline弥补了MLlib库的不足，向用户提供了一个基于DataFrame的机器学习的工作流式API套件，使用ML Pipeline API我们可以很方便地把数据处理、特征转化、正则化以及多个机器学习算法联合起来，构建一个单一完整的机器学习流水线。

从官方文档看，Spark ML Pipeline虽然是被推荐的机器学习方式，但是不会在短期内替代原始的MLlib库，因为MLlib已经包含了丰富稳定的算法实现，并且部分ML Pipeline实现基于MLLib。实际工作中，并不是所有的机器学习过程都需要构建成一个流水线，有时候原始的数据格式整齐且完整，而且使用单一的算法就能实现目标，也没有把事情复杂化，采用最简单且容易理解的方式才是正确的选择。

算法架构如下：

MLLIB主要包含两个部分：

1. 底层基础：主要包括Spark的运行库、矩阵库和向量库。其中向量接口和矩阵接口基于Nelib和BLAS/LAPACK开发的线性代数库Breeze。MLlib支持本地的密集向量和本地向量，并且支持标量向量；同时支持本地矩阵和分布式矩阵，分布式矩阵分为：RowMatrix、IndexedRowMatrix和CoordinateMatrix等。
2. 算法库：包含分类、回归、聚集、协同过滤、梯度下降和特征提取和变换等算法。



以上是Spark.mllib算法架构。

## 1.1MLLIB基本数据类型

### 1.1.1数据类型简介

MLLIB支持很多种机器学习算法中类型，主要有向量和矩阵两种类型。

有下面四种分类：

1. Local vector本地向量集，主要向Spark提供一组可进行操作的数据集合。
2. Labeled Point向量标签，让用户能够分类不同的数据集合。
3. Local matrix本地矩阵，将数据集合以矩阵形式存储在本地计算机中。
4. Distribute matrix分布式矩阵。将数据集以矩阵的形式存储在分布式的计算机中。

### 1.1.2 Local Vector 本地向量

本地向量主要由两种类型构成：（1）稀疏型数据集spares（2）密集型数据集（dense）。

假设一个向量(9,5,2,7)，按密集型数据格式可以设置为（9,5,2,7）进行存储，数据集被作为一个集合的形式整体存储。按稀疏性方式存储，可以按向量的大小存储为(4,Array(0,1,2,3),Array(9,5,2,7)).

代码如下：

import org.apache.spark.mllib.linalg.{Vector, Vectors}  
object testVector {  
 def main(args: Array[String]) {  
 val vd: Vector = Vectors.dense(2, 0, 6) //建立密集向量  
 println(vd(2)) //打印稀疏向量第3个值  
 val vs: Vector = Vectors.sparse(4, Array(0, 1, 2, 3), Array(9, 5, 2, 7)) //建立稀疏向量  
 //第一个参数4代表输入数据的大小，一般要求大于等于输入的数据值，第二个参数是数据下标，第三个参数是数据值  
 println(vs(2)) //打印稀疏向量第3个值  
 }  
}

Output：

6.0

2.0

在MLlib的数据支持格式中，目前**仅支持整数与浮点型数**。其他数据类型不在支持范围之内，主要也是因为MLLIB主要用于做数值计算。

注意：scala.collection.immutable.Vector默认情况下， Scala会导入，因此您必须org.apache.spark.mllib.linalg.Vector明确导入 才能使用MLlibVector。

另外的官方API参考例子：

package MLLIB.C04  
  
import org.apache.spark.mllib.linalg.{Vector, Vectors}  
  
object LocalVector {  
 def main(args: Array[String]) {  
 val vd: Vector = Vectors.dense(2, 0, 6) //建立密集向量  
 println(vd(2)) //打印稀疏向量第3个值  
 val vs: Vector = Vectors.sparse(4, Array(0, 1, 2, 3), Array(9, 5, 2, 7)) //建立稀疏向量  
 //第一个参数4代表输入数据的大小，一般要求大于等于输入的数据值，第二个参数是数据下标，第三个参数是数据值  
 println(vs(2)) //打印稀疏向量第3个值  
 //通过指定其非零条目来创建稀疏向量（1.0,0.0,3.0）  
 val sv2: Vector = Vectors.sparse(3, Seq((0, 1.0), (2, 3.0)))  
 println(sv2(0))  
 }  
}

### 1.1.3 Labled Point向量标签

向量标签用于MLLIB中机器学习算法做标记。在分类问题中，可以将不同的数据集分成若干份，以整数型0、1、2进行标记。

LaebledPoint是建立向量标签的静态类，主要有两个方法（1）Features用于显示打印标记点所代表的数据内容，而label用于显示标记数。

import org.apache.spark.mllib.linalg.{Vector, Vectors}  
import org.apache.spark.mllib.regression.LabeledPoint  
  
object testLabeledPoint {  
 def main(args: Array[String]) {  
 val vd: Vector = Vectors.dense(2, 0, 6) //建立密集向量  
 val pos = LabeledPoint(1, vd) //对密集向量建立标记点  
 println(pos.features) //打印标记点内容数据  
 println(pos.label) //打印既定标记  
val vs: Vector = Vectors.sparse(4, Array(0,1,2,3), Array(9,5,2,7)) //建立稀疏向量  
 val neg = LabeledPoint(2, vs) //对密集向量建立标记点  
 println(neg.features) //打印标记点内容数据  
 println(neg.label) //打印既定标记  
 }  
}

Output：

[2.0,0.0,6.0]

1.0

(4,[0,1,2,3],[9.0,5.0,2.0,7.0])

2.0

案例2：

<https://www.csie.ntu.edu.tw/~cjlin/libsvm/>

SparkMLlib读取LibSvm数据，libsvm数据格式为：

Label index1：value1 index2：value2 ......如：

2 1:5 2:8 3:9**(索引从1开始，从0开始会报错)**

1 1:7 2:6 3:7

1 1:3 2:2 3:1

package MLLIB.C04  
  
  
import org.apache.spark.\_  
import org.apache.spark.mllib.util.MLUtils  
import org.apache.spark.mllib.regression.LabeledPoint  
import org.apache.spark.rdd.RDD  
  
object LabeledPoint2Test {  
 def main(args: Array[String]) {  
 val conf = new SparkConf().setMaster("local").setAppName("testLabeledPoint2") //建立本地环境变量  
 val sc = new SparkContext(conf) //建立Spark处理  
 **val mu: RDD[LabeledPoint] = MLUtils.loadLibSVMFile(sc, "D://a.txt") //读取文件**  
 mu.foreach(println) //打印内容(1.0,(3,[0,1,2],[2.0,3.0,5.0]))  
 }  
}

官方文档

Import org.apache.spark.mllib.regression.LabeledPoint

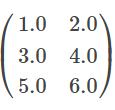
import org.apache.spark.mllib.util.MLUtils

import org.apache.spark.rdd.RDD

Valexamples:RDD[LabeledPoint]=MLUtils.loadLibSVMFile(sc, "data/mllib/sample\_libsvm\_data.txt")

### 1.1.4 Local matrix本地矩阵

\* 局部矩阵具有整数类型的行和列索引和双类型值，存储在单个机器上。  
\* MLlib支持密集矩阵，其入口值以列主序列存储在单个双阵列中，稀疏矩阵的  
\* 非零入口值以**列**主要顺序存储在压缩稀疏列（CSC）格式中。 例如，以下密集矩阵



按照一个列向量存储[1.0, 3.0, 5.0, 2.0, 4.0, 6.0] 矩阵尺寸为 (3, 2)

package MLLIB.C04  
  
/\*\*  
 \* Created by zhao-chj on 2017/8/3.  
 \* 局部矩阵具有整数类型的行和列索引和双类型值，存储在单个机器上。  
 \* MLlib支持密集矩阵，其入口值以列主序列存储在单个双阵列中，稀疏矩阵的  
 \* 非零入口值以列主要顺序存储在压缩稀疏列（CSC）格式中。 例如，以下密集矩阵  
 \*  
 \* 局部矩阵的基类是Matrix，  
 \* 我们提供了两个实现：DenseMatrix和SparseMatrix。 我们建议使用Matrices中实现的工厂方法来创建本地矩阵。  
 \* 记住，MLlib中的局部矩阵以列主要顺序存储.  
 \*/  
object LocalMatrix {  
 def main(args: Array[String]) {  
 import org.apache.spark.mllib.linalg.{Matrix, Matrices}  
  
 // Create a dense matrix ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))  
 val dm: Matrix = Matrices.dense(3, 2, Array(1.0, 3.0, 5.0, 2.0, 4.0, 6.0))  
 println(dm(2,0))  
 // Create a sparse matrix ((9.0, 0.0), (0.0, 8.0), (0.0, 6.0))  
 val sm: Matrix = Matrices.sparse(3, 2, Array(0, 1, 3), Array(0, 2, 1), Array(9, 6, 8))  
 println(dm(2,1))  
 }  
}

### 1.1.5 Distributed matrix分布式矩阵

分布式矩阵由**长整型行列索引和双精度浮点型值**数据组成，分布式存储在一个或多个RDD中，对于巨大的分布式矩阵来说，选择正确的存储格式非常重要，将一个分布式矩阵转化为另外一个不同格式需要混洗(shuffle)，其代价很高。在MLlib实现了三类分布式矩阵存储格式，分别是**行矩阵（RowMatrix）**、**行索引矩阵（IndexedRowMatrix）**、**三元组矩阵（CoordinateMatrix**）和**分块矩阵（BlockMatrix）**等四种。

#### **行矩阵**RowMatrix：

package MLLIB.C04  
  
import org.apache.spark.\_  
import org.apache.spark.mllib.linalg.{Vector, Vectors}  
import org.apache.spark.mllib.linalg.distributed.RowMatrix  
//行矩阵是一个面向行的分布式矩阵，行索引是没有具体含义的。比如一系列特征向量的一个集合，通过一个RDD来代表所有的行，  
//每一个行就是一个本地向量。  
object RowMatrixTest {  
 def main(args: Array[String]) {  
 val conf = new SparkConf() //创建环境变量  
 .setMaster("local") //设置本地化处理  
 .setAppName("testRowMatrix") //设定名称  
 val sc = new SparkContext(conf) //创建环境变量实例  
 val rdd = sc.textFile("D://BigData//Workspace//IDEAScala//SparkDemo//src//MLLIB//C04//RowMatrix.txt") //创建RDD文件路径  
 .map(\_.split(' ') //按“ ”分割  
 .map(\_.toDouble)) //转成Double类型  
 .map(line => Vectors.dense(line)) //转成Vector格式  
 val rm = new RowMatrix(rdd) //读入行矩阵  
 //RowMatrix是一个transform操作，得到的并不是最终的运行结果。  
 println(rm.numRows()) //打印列数  
 println(rm.numCols()) //打印行数  
 }  
}

#### **行索引矩阵**IndexedRowMatrix：

An IndexedRowMatrix类似于a RowMatrix但具有有意义的行索引。它由索引行的RDD支持，因此每行由其索引（long-typed）和本地向量表示。

一个 [IndexedRowMatrix](http://spark.apache.org/docs/latest/api/scala/index.html" \l "org.apache.spark.mllib.linalg.distributed.IndexedRowMatrix) 可以从创建RDD[IndexedRow]例如，在 [IndexedRow](http://spark.apache.org/docs/latest/api/scala/index.html" \l "org.apache.spark.mllib.linalg.distributed.IndexedRow)已经结束的包装(Long, Vector)。一个IndexedRowMatrix可以被转换为RowMatrix通过降低其行索引。

package MLLIB.C04  
  
import org.apache.spark.\_  
import org.apache.spark.mllib.linalg.distributed.{IndexedRow, RowMatrix, IndexedRowMatrix}  
import org.apache.spark.mllib.linalg.{Vector, Vectors}  
//行索引矩阵：本质上是含有索引信息的行数据集合，每一行由长整型索引和一个本地向量组成  
//行索引矩阵可从一个RDD[indexRow]实例创建，这里的IndexedRow是（Long,Vector）的封装类  
//剔除了行索引信息就是一个行矩阵  
object IndexedRowMatrixTest {  
 def main(args: Array[String]) {  
val conf = new SparkConf() //创建环境变量  
.setMaster("local") //设置本地化处理  
 .setAppName("testIndexedRowMatrix") //设定名称  
 val sc = new SparkContext(conf) //创建环境变量实例  
 val rdd = sc.textFile( "D://BigData//Workspace//IDEAScala//SparkDemo//src//MLLIB//C04//RowMatrix.txt") //创建RDD文件路径  
 .map(\_.split(' ') //按“ ”分割  
 .map(\_.toDouble)) //转成Double类型  
 .map(line => Vectors.dense(line)) //转化成向量存储  
 .map((vd) => new IndexedRow(vd.size,vd)) //转化格式  
 val irm = new IndexedRowMatrix(rdd) //建立索引行矩阵实例  
 println(irm.getClass) //打印类型  
 println(irm.rows.foreach(println)) //打印内容数据  
 }  
}

#### **三元组矩阵**CoordinateMatrix：

A CoordinateMatrix是由其条目的RDD支持的分布式矩阵。每个条目都是一个元组(i: Long, j: Long, value: Double)，其中i是行索引，j是列索引， value是条目值。只有当矩阵的两个维度都很大并且矩阵非常稀疏时，才应该使用A CoordinateMatrix

[CoordinateMatrix](http://spark.apache.org/docs/latest/api/scala/index.html" \l "org.apache.spark.mllib.linalg.distributed.CoordinateMatrix) 可以从RDD[MatrixEntry]实例创建A ，其中 [MatrixEntry](http://spark.apache.org/docs/latest/api/scala/index.html" \l "org.apache.spark.mllib.linalg.distributed.MatrixEntry)是包装器(Long, Long, Double)。CoordinateMatrix可以IndexedRowMatrix 通过调用将A 转换为具有稀疏行的a toIndexedRowMatrix。CoordinateMatrix目前不支持其他计算 。.

package MLLIB.C04  
//三元组矩阵（coordinateMatrix）：是一个分布式矩阵，其实体集合是一个RDD，每一个实体是一个（i:Long,j:Ling,value:Double）三元组  
//其中i代表行索引，j代表列索引，value代表实体值  
//三元组矩阵常用于表示稀疏性比较高的计算中，是由RDD[MatrixEntry]来构建的。  
//MatrixEntry是一个Tuple类型的元素，包含行、列和元素的值  
import org.apache.spark.\_  
import org.apache.spark.mllib.linalg.{Vector, Vectors}  
import org.apache.spark.mllib.linalg.distributed.{CoordinateMatrix, MatrixEntry}  
  
object CoordinateRowMatrixTest {  
 def main(args: Array[String]) {  
val conf = new SparkConf() //创建环境变量  
.setMaster("local") //设置本地化处理  
.setAppName("testCoordinateRowMatrix") //设定名称  
 val sc = new SparkContext(conf) //创建环境变量实例  
 val rdd = sc.textFile( "D://BigData//Workspace//IDEAScala//SparkDemo//src//MLLIB//C04//RowMatrix.txt") //创建RDD文件路径  
 .map(\_.split(' ') //按“ ”分割  
 .map(\_.toDouble)) //转成Double类型  
 .map(vue => (vue(0).toLong,vue(1).toLong,vue(2))) //转化成坐标格式  
 .map(vue2 => new MatrixEntry(vue2 \_1,vue2 \_2,vue2 \_3)) //转化成坐标矩阵格式  
 val crm = new CoordinateMatrix(rdd) //实例化坐标矩阵  
  
 println(crm.entries.foreach(println)) //打印数据  
 println(crm.numRows())//打印行数据  
 println(crm.numCols())//打印列数据  
 }  
}

#### **分块矩阵**BlockMatrix：

//分块矩阵：BlockMatrix是支持矩阵分块RDD的分布式矩阵，其中矩阵分块由((int,int),matrix)元祖所构成  
//（int,int）是该部分矩阵所处的矩阵的索引位置，Matrix表示该索引位置上的子矩阵  
//分块矩阵支持矩阵加法和乘法，并设有辅助函数验证用于检查矩阵是否设置正确。  
import org.apache.spark.\_  
import org.apache.spark.mllib.linalg.distributed.{CoordinateMatrix, MatrixEntry,BlockMatrix}  
import org.apache.spark.storage.StorageLevel  
  
object BlockMatrixTest {  
 def main(args: Array[String]) {  
val conf = new SparkConf() //创建环境变量  
.setMaster("local") //设置本地化处理  
.setAppName("testCoordinateRowMatrix") //设定名称  
 val sc = new SparkContext(conf) //创建环境变量实例  
 val rdd = sc.textFile( "D://BigData//Workspace//IDEAScala//SparkDemo//src//MLLIB//C04//RowMatrix.txt") //创建RDD文件路径  
 .map(\_.split(' ') //按“ ”分割  
 .map(\_.toDouble)) //转成Double类型  
 .map(vue => (vue(0).toLong,vue(1).toLong,vue(2))) //转化成坐标格式  
 .map(vue2 => new MatrixEntry(vue2 \_1,vue2 \_2,vue2 \_3))  
 //转化成坐标矩阵格式  
 val crm = new CoordinateMatrix(rdd) //实例化坐标矩阵  
 //使用toBlockMatrix把三元组矩阵转化为分块矩阵  
 val matA: BlockMatrix =crm.toBlockMatrix().cache()  
 // 对该分块矩阵进行检验，确认该分块是否正确，如果不正确则抛出异常  
 //验证BlockMatrix是否正确设置。 当无效时抛出异常。  
 // Nothing happens if it is valid.  
 matA.validate()  
 // Calculate A^T A.  
 val ata = matA.transpose.multiply(matA)  
 // print(ata.persist(StorageLevel.DISK\_ONLY))//因为transpose是trnsform操作不会直接得到结果，调用action算子会得到结果  
  
 }  
}

## 2.2统计量的MLLIB实现

RDD[Vector]通过colStats 可用的功能提供列摘要统计信息Statistics。

### 2.2.1均值和方差

[colStats()](http://spark.apache.org/docs/latest/api/scala/index.html" \l "org.apache.spark.mllib.stat.Statistics$)返回一个实例 [MultivariateStatisticalSummary](http://spark.apache.org/docs/latest/api/scala/index.html" \l "org.apache.spark.mllib.stat.MultivariateStatisticalSummary)，其中包含按列的max，min，mean，variance和非零数，以及总计数。

package MLLIB.C04  
  
import org.apache.spark.\_  
import org.apache.spark.mllib.linalg.Vectors  
import org.apache.spark.mllib.stat.Statistics  
import org.apache.spark.{SparkConf, SparkContext}  
  
object SummaryTest {  
 def main(args: Array[String]) {  
 val conf = new SparkConf() //创建环境变量  
 .setMaster("local") //设置本地化处理  
 .setAppName("testSummary") //设定名称  
 val sc = new SparkContext(conf) //创建环境变量实例  
 val rdd = sc.textFile( "D://BigData//Workspace//IDEAScala//SparkDemo//src//MLLIB//C04//testSummary.txt") //创建RDD文件路径  
 .map(\_.split(' ') //按“ ”分割  
 .map(\_.toDouble)) //转成Double类型  
 .map(line => Vectors.dense(line)) //转成Vector格式  
 val summary = Statistics.colStats(rdd) //获取Statistics实例  
 println(summary.mean) //计算均值  
 println(summary.variance) //计算标准差  
 }  
}

API测试例子：

package MLLIB.C04  
  
import org.apache.spark.{SparkContext, SparkConf}  
import org.apache.spark.mllib.linalg.Vectors  
import org.apache.spark.mllib.stat.{MultivariateStatisticalSummary, Statistics}  
/\*\*  
 \* Created by zhao-chj on 2017/8/3.  
 \* colStats（）返回一个MultivariateStatisticalSummary的实例，它包含列的最大值，最小值，平均值，方差和非数字数，以及总计数。  
 \*/  
object SummaryStatisticsTestAPI {  
 def main(args: Array[String]) {  
 val conf = new SparkConf().setMaster("local").setAppName("testLabeledPoint2") //建立本地环境变量  
 val sc = new SparkContext(conf) //建立Spark处理  
 val observations = sc.parallelize(  
 Seq(  
 Vectors.dense(1.0, 10.0, 100.0),  
 Vectors.dense(2.0, 20.0, 200.0),  
 Vectors.dense(3.0, 30.0, 300.0)  
 )  
 )  
 // Compute column summary statistics.  
 val summary: MultivariateStatisticalSummary = Statistics.colStats(observations)  
 println(summary.mean) // a dense vector containing the mean value for each column  
 println(summary.variance) // column-wise variance  
 println(summary.numNonzeros) // number of nonzeros in each column  
 }  
}  
//[2.0,20.0,200.0]  
//[1.0,100.0,10000.0]  
//[3.0,3.0,3.0]

### 2.2.2距离计算

package MLLIB.C04  
  
import org.apache.spark.mllib.linalg.Vectors  
import org.apache.spark.mllib.stat.Statistics  
import org.apache.spark.{SparkConf, SparkContext}  
  
object Summary2Test{  
 def main(args: Array[String]) {  
val conf = new SparkConf() //创建环境变量  
.setMaster("local") //设置本地化处理  
.setAppName("testSummary2") //设定名称  
 val sc = new SparkContext(conf) //创建环境变量实例  
 val rdd = sc.textFile("D://BigData//Workspace//IDEAScala//SparkDemo//src//MLLIB//C04//testSummary.txt") //创建RDD文件路径  
 .map(\_.split(' ') //按“ ”分割  
 .map(\_.toDouble)) //转成Double类型  
 .map(line => Vectors.dense(line)) //转成Vector格式  
 **val summary = Statistics.colStats(rdd) //获取Statistics实例**  
 println(summary.**normL1**) //计算曼哈段距离  
 println(summary.**normL2**) //计算欧几里得距离  
 }  
}

### 2.2.3相关系数计算

API实践

import org.apache.spark.mllib.linalg.\_  
import org.apache.spark.mllib.stat.Statistics  
import org.apache.spark.rdd.RDD  
import org.apache.spark.{SparkConf, SparkContext}  
object Correlation\_3 {  
 def main(args: Array[String]): Unit = {  
 val conf = new SparkConf() //创建环境变量  
 .setMaster("local") //设置本地化处理  
 .setAppName("testSummary") //设定名称  
 val sc = new SparkContext(conf) //创建环境变量实例  
  
  
 val seriesX: RDD[Double] = sc.parallelize(Array(1, 2, 3, 3, 5)) // a series  
 // must have the same number of partitions and cardinality as seriesX  
 val seriesY: RDD[Double] = sc.parallelize(Array(11, 22, 33, 33, 555))  
  
 // compute the correlation using Pearson's method. Enter "spearman" for Spearman's method. If a  
 // method is not specified, Pearson's method will be used by default.  
 val correlation: Double = Statistics.corr(seriesX, seriesY, "pearson")  
 println(s"Correlation is: $correlation")  
  
 val data: RDD[Vector] = sc.parallelize(  
 Seq(  
 Vectors.dense(1.0, 10.0, 100.0),  
 Vectors.dense(2.0, 20.0, 200.0),  
 Vectors.dense(5.0, 33.0, 366.0))  
 ) // note that each Vector is a row and not a column  
  
 // calculate the correlation matrix using Pearson's method. Use "spearman" for Spearman's method  
 // If a method is not specified, Pearson's method will be used by default.  
 val correlMatrix: Matrix = Statistics.corr(data, "pearson")  
 println(correlMatrix.toString)  
 }  
}

皮尔逊相关系数

package MLLIB.C04  
  
import org.apache.spark.mllib.linalg.Vectors  
import org.apache.spark.mllib.stat.Statistics  
import org.apache.spark.{SparkConf, SparkContext}  
  
object CorrectTest {  
 def main(args: Array[String]) {  
val conf = new SparkConf() //创建环境变量  
.setMaster("local") //设置本地化处理  
.setAppName("testCorrect ") //设定名称  
 val sc = new SparkContext(conf) //创建环境变量实例  
 val rddX = sc.textFile("D://BigData//Workspace//IDEAScala//SparkDemo//src//MLLIB//C04//testCorrectX.txt") //读取数据  
 .flatMap(\_.split(' ') //进行分割  
 .map(\_.toDouble)) //转化为Double类型  
 val rddY = sc.textFile("D://BigData//Workspace//IDEAScala//SparkDemo//src//MLLIB//C04//testCorrectY.txt") //读取数据  
 .flatMap(\_.split(' ') //进行分割  
 .map(\_.toDouble)) //转化为Double类型  
 val correlation: Double = Statistics.corr(rddX, rddY) //计算不同数据之间的相关系数  
 println(correlation) //打印结果  
 }  
}

斯皮尔曼相关系数：

package MLLIB.C04  
  
import org.apache.spark.mllib.linalg.Vectors  
import org.apache.spark.mllib.stat.Statistics  
import org.apache.spark.{SparkConf, SparkContext}  
  
object Correct2Test {  
 def main(args: Array[String]) {  
 val conf = new SparkConf() //创建环境变量  
 .setMaster("local") //设置本地化处理  
 .setAppName("testCorrect2 ") //设定名称  
 val sc = new SparkContext(conf) //创建环境变量实例  
 val rddX = sc.textFile("D://BigData//Workspace//IDEAScala//SparkDemo//src//MLLIB//C04//testCorrectX.txt") //读取数据  
 .flatMap(\_.split(' ') //进行分割  
 .map(\_.toDouble)) //转化为Double类型  
 val rddY = sc.textFile("D://BigData//Workspace//IDEAScala//SparkDemo//src//MLLIB//C04//testCorrectY.txt") //读取数据  
 .flatMap(\_.split(' ') //进行分割  
 .map(\_.toDouble)) //转化为Double类型  
 val correlation: Double = Statistics.corr(rddX, rddY, "spearman") //使用斯皮尔曼计算不同数据之间的相关系数  
 println(correlation) //打印结果  
 }  
}

### 2.2.4随机数

随机数是机器学习中生成数据集的常用方法，一般用来检验随机算法和执行效率等，RandomRDDs类是随机数生成类，使用normalRDD随机生成100个随机数。

package MLLIB.C04  
  
import org.apache.spark.{SparkContext, SparkConf}  
import org.apache.spark.mllib.random.RandomRDDs.\_  
  
object RandomRDDTest {  
 def main(args: Array[String]) {  
 val conf = new SparkConf()  
 .setMaster("local")  
 .setAppName("RandomRDDTest")  
 val sc = new SparkContext(conf)  
 val randomNum = normalRDD(sc, 100)  
 randomNum.foreach(println)  
 //API测试  
 // 生成一个包含100万id.的随机数RDD。 从标准正态分布“N（0,1）”中绘出的值，均匀分布在10个分区中。  
 val u = normalRDD(sc, 1000000L, 10)  
 // 应用变换获得随机的双RDD跟随“N（1，4）”。  
 val v = u.map(x => 1.0 + 2.0 \* x)  
 v.foreach(println) //输出100万数据  
 }  
}

### 2.2.5假设检验

假设检验是统计学中一种强有力的工具，用于确定结果是否具有统计显着性，无论该结果是否偶然发生。spark.mllib目前支持Pearson的卡方（χ2）测试适合度和独立性。输入数据类型确定是否进行拟合优度或独立性测试。拟合优度测试需要输入类型Vector，而独立性测试需要Matrix输入。

spark.mllib还支持输入类型RDD[LabeledPoint]，通过卡方独立测试启用特征选择。

究竟对于数据的好坏，我们需要一个能够反映和检验结果正确与否的方法。卡方检验是一种常用的检验方法，能够较好地对数据集之间的拟合度、相关性和独立性进行检验。MLLIB规定常用的卡方检验使用的数据集一般为向量和矩阵。

最早的卡方检验开始与抽查检测工厂合格品概率，在网站上一般用作转化率等指标的计算和衡量。

卡方检验：

package MLLIB.C04  
  
import org.apache.spark.mllib.linalg.{Matrices, Vectors}  
import org.apache.spark.mllib.stat.Statistics  
  
object testChiSq{  
 def main(args: Array[String]) {  
 val vd = Vectors.dense(1,2,3,4,5) //  
 val vdResult = Statistics.chiSqTest(vd)  
 println(vdResult)  
 println("-------------------------------")  
 val mtx = Matrices.dense(3, 2, Array(1, 3, 5, 2, 4, 6))  
 val mtxResult = Statistics.chiSqTest(mtx)  
 println(mtxResult)  
 }  
}  
//Chi squared test summary:  
//method: pearson  
//degrees of freedom = 4  
//statistic = 3.333333333333333  
//pValue = 0.5036682742334986  
//No presumption against null hypothesis: observed follows the same distribution as expected..  
//-------------------------------  
//Chi squared test summary:  
//method: pearson  
//degrees of freedom = 2  
//statistic = 0.14141414141414144  
//pValue = 0.931734784568187  
//No presumption against null hypothesis: the occurrence of the outcomes is statistically independent..

从结果可以看出来，假设检验的输出结果包含三个数据，分别为自由度、P值及统计量。

其中：P值是显著性差异指标，统计量是指不同方法下的统计量，自由度是总体参数估计量中变量值独立自由变化的数目。

API假设检验案例2：

package MLLIB.C04  
  
import org.apache.spark.{SparkContext, SparkConf}  
import org.apache.spark.mllib.linalg.\_  
import org.apache.spark.mllib.regression.LabeledPoint  
import org.apache.spark.mllib.stat.Statistics  
import org.apache.spark.mllib.stat.test.ChiSqTestResult  
import org.apache.spark.rdd.RDD  
/\*\*  
 \* Created by zhao-chj on 2017/8/3.  
 \* 假设检验是统计学中强大的工具，用于确定结果是否具有统计学意义，无论该结果是否偶然发生。  
 \* spark.mllib目前支持Pearson的卡方（χ2χ2）测试，以获得适合度和独立性。 输入数据类型确定是否进行拟合优度或独立性测试。  
 \* 适合度测试需要输入类型的Vector，而独立性测试需要一个Matrix作为输入。  
 \* spark.mllib还支持输入类型RDD [LabeledPoint]，通过卡方独立测试来启用特征选择。  
 \*/  
object HypothesisTestingTestAPI {  
 def main(args: Array[String]) {  
 val conf = new SparkConf().setMaster("local").setAppName("testLabeledPoint2") //建立本地环境变量  
 val sc = new SparkContext(conf) //建立Spark处理  
 // a vector composed of the frequencies of events  
 val vec: Vector = Vectors.dense(0.1, 0.15, 0.2, 0.3, 0.25)  
  
 // compute the goodness of fit. If a second vector to test against is not supplied  
 // as a parameter, the test runs against a uniform distribution.  
 val goodnessOfFitTestResult = Statistics.chiSqTest(vec)  
 // summary of the test including the p-value, degrees of freedom, test statistic, the method  
 // used, and the null hypothesis.  
 println(s"$goodnessOfFitTestResult\n")  
  
 // a contingency matrix. Create a dense matrix ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))  
 val mat: Matrix = Matrices.dense(3, 2, Array(1.0, 3.0, 5.0, 2.0, 4.0, 6.0))  
  
 // conduct Pearson's independence test on the input contingency matrix  
 val independenceTestResult = Statistics.**chiSqTest**(mat)  
 // summary of the test including the p-value, degrees of freedom  
 println(s"$independenceTestResult\n")  
  
 val obs: RDD[LabeledPoint] =  
 sc.parallelize(  
 Seq(  
 LabeledPoint(1.0, Vectors.dense(1.0, 0.0, 3.0)),  
 LabeledPoint(1.0, Vectors.dense(1.0, 2.0, 0.0)),  
 LabeledPoint(-1.0, Vectors.dense(-1.0, 0.0, -0.5)  
 )  
 )  
 ) // (feature, label) pairs.  
  
 // The contingency table is constructed from the raw (feature, label) pairs and used to conduct  
 // the independence test. Returns an array containing the ChiSquaredTestResult for every feature  
 // against the label.  
 val featureTestResults: Array[ChiSqTestResult] = Statistics.chiSqTest(obs)  
 featureTestResults.zipWithIndex.foreach { case (k, v) =>  
 println("Column " + (v + 1).toString + ":")  
 println(k)  
 } // summary of the test  
 }  
}  
//Column 1:  
//Chi squared test summary:  
//method: pearson  
//degrees of freedom = 1  
//statistic = 3.0000000000000004  
//pValue = 0.08326451666354884  
//Low presumption against null hypothesis: the occurrence of the outcomes is statistically independent..  
//Column 2:  
//Chi squared test summary:  
//method: pearson  
//degrees of freedom = 1  
//statistic = 0.75  
//pValue = 0.3864762307712326  
//No presumption against null hypothesis: the occurrence of the outcomes is statistically independent..  
//Column 3:  
//Chi squared test summary:  
//method: pearson  
//degrees of freedom = 2  
//statistic = 3.0  
//pValue = 0.22313016014843035  
//No presumption against null hypothesis: the occurrence of the outcomes is statistically independent..

Kolmogorov-Smirnov（KS）测试

package MLLIB.C04  
  
import org.apache.spark.mllib.stat.Statistics  
import org.apache.spark.rdd.RDD  
import org.apache.spark.{SparkConf, SparkContext}  
  
/\*\*  
 \* Created by zhao-chj on 2017/8/3.  
 \* 此外，spark.mllib提供了对于概率分布相等的Kolmogorov-Smirnov（KS）测试的1样本双边实现。  
 \* 通过提供理论分布（目前仅为正态分布支持）及其参数的名称，或根据给定理论分布计算累积分布的函数，  
 \* 用户可以测试其假设，即样本是从该 分配。 在用户根据正常分布（distName =“norm”）  
 \* 进行测试但不提供分发参数的情况下，测试将初始化为标准正态分布并记录适当的消息。  
 \*/  
object HypothesisTestingTestAPI2 {  
 def main(args: Array[String]) {  
  
 val conf = new SparkConf().setMaster("local").setAppName("testLabeledPoint2") //建立本地环境变量  
 val sc = new SparkContext(conf) //建立Spark处理  
 val data: RDD[Double] = sc.parallelize(Seq(0.1, 0.15, 0.2, 0.3, 0.25)) // an RDD of sample data  
  
 // run a KS test for the sample versus a standard normal distribution  
 val testResult = Statistics.kolmogorovSmirnovTest(data, "norm", 0, 1)  
 // summary of the test including the p-value, test statistic, and null hypothesis if our p-value  
 // indicates significance, we can reject the null hypothesis.  
 println(testResult)  
 println()  
  
 // perform a KS test using a cumulative distribution function of our making  
 val myCDF = Map(0.1 -> 0.2, 0.15 -> 0.6, 0.2 -> 0.05, 0.3 -> 0.05, 0.25 -> 0.1)  
 val testResult2 = Statistics.kolmogorovSmirnovTest(data, myCDF)  
 println(testResult2)  
 }  
}  
//Kolmogorov-Smirnov test summary:  
//degrees of freedom = 0  
//statistic = 0.95  
//pValue = 6.249999999763389E-7  
//Very strong presumption against null hypothesis: Sample follows theoretical distribution.

### 2.2.6分层采样

分层抽样是一种数据提取方法，先将总体的单位按某种特征分为若干级总体(层)，然后在从每一层内进行单纯的随机抽样，组成一个新的样本的统计学计算方法。这种方法常常用于数据量比较大，计算处理非常不方便的情况下。

package MLLIB.C04  
  
import org.apache.spark.{SparkContext, SparkConf}  
  
/\*\*  
 \* Created by zhao-chj on 2017/8/3.  
 \* 与驻留在spark.mllib中的其他统计功能不同，可以对键值对的RDD执行分层抽样方法sampleByKey和sampleByKeyExact。  
 \* 对于分层采样，键可以被认为是一个标签，该值作为一个特定属性。 例如，密钥可以是男人或女人或文档ID，并且相应  
 \* 的值可以是人口中的人的年龄列表或文档中的单词列表。 sampleByKey方法将翻转硬币来决定观察是否被采样，因此需要  
 \* 一次通过数据，并提供预期的样本大小。 sampleByKeyExact需要比sampleByKey中使用的每层简单随机抽样更多的资源，但  
 \* 将提供99.99％置信度的确切抽样大小。 python当前不支持sampleByKeyExact。  
 \*/  
object StratifiedSamplingTestAPI {  
 def main(args: Array[String]) {  
 val conf = new SparkConf().setMaster("local").setAppName("testLabeledPoint2") //建立本地环境变量  
 val sc = new SparkContext(conf) //建立Spark处理  
 // an RDD[(K, V)] of any key value pairs  
 val data = sc.parallelize(  
 Seq((1, 'a'), (1, 'b'), (2, 'c'), (2, 'd'), (2, 'e'), (3, 'f')))  
  
 // specify the exact fraction desired from each key  
 val fractions = Map(1 -> 0.1, 2 -> 0.6, 3 -> 0.3)  
  
 // Get an approximate sample from each stratum  
 val approxSample = data.**sampleByKey**(withReplacement = false, fractions = fractions)  
 approxSample.foreach(println)//打印结果  
// (1,b)  
// (2,d)  
 // Get an exact sample from each stratum  
 val exactSample = data.**sampleByKeyExact**(withReplacement = false, fractions = fractions)  
 exactSample.foreach(println)//打印结果  
// (2,e)  
// (3,f)  
 }  
}

# **Mllib抽取，转换和选择功能**

* 提取：从“原始”数据中提取要素
* 转换：缩放，转换或修改功能
* 选择：从更大的功能集中选择子集

## 1.1特征提取器

### TF-IDF

import org.apache.spark.sql.SparkSession

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

object TfIdf\_1 {

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

val spark: SparkSession = SparkSession.builder()

.appName("SparkMlilb")

.master("local[2]")

.getOrCreate()

spark.sparkContext.setLogLevel("WARN")

val sentenceData = spark.createDataFrame(Seq(

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

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

(1, "Logistic regression models are neat")

)).toDF("label", "sentence")

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)

// alternatively, CountVectorizer can also be used to get term frequency vectors

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

val idfModel = idf.fit(featurizedData)

val rescaledData = idfModel.transform(featurizedData)

rescaledData.select("features", "label").take(3).foreach(println)

}

}

### Word2Vec

import org.apache.spark.sql.SparkSession

import org.apache.spark.ml.feature.Word2Vec

object Word2Vec\_3 {

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

val spark: SparkSession = SparkSession.builder()

.appName("SparkMlilb")

.master("local[2]")

.getOrCreate()

spark.sparkContext.setLogLevel("WARN")

// Input data: Each row is a bag of words from a sentence or document.

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")

// Learn a mapping from words to Vectors.

val word2Vec = new Word2Vec()

.setInputCol("text")

.setOutputCol("result")

.setVectorSize(3)

.setMinCount(0)

val model = word2Vec.fit(documentDF)

val result = model.transform(documentDF)

result.select("result").take(3).foreach(println)

}

}

结果：

[[0.03173386193811894,0.009443491697311401,0.024377789348363876]]

[[0.025682436302304268,0.0314303718706859,-0.01815584538105343]]

[[0.022586782276630402,-0.01601201295852661,0.05122732147574425]]

### CountVectorizer

CountVectorizer并CountVectorizerModel旨在帮助将一组文本文档转换为令牌计数的向量。当apriori字典不可用时，CountVectorizer可以用作Estimator提取词汇表，并生成一个CountVectorizerModel。该模型为词汇表上的文档生成稀疏表示，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vocabSize按语料库中的术语**频率排序**的顶部单词。可选参数minDF还通过指定术语必须出现在文档中的最小数量（或<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.这对于模拟二进制而非整数计数的离散概率模型特别有用。

****Examples****

假设我们有以下的DataFrame，带有列id和文本：

id | texts

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

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

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

each row in texts is a document of type Array[String].

Invoking fit of CountVectorizer produces a CountVectorizerModel with vocabulary (a, b, c). Then the output column “vector” after transformation contains:

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])

import org.apache.spark.ml.feature.{CountVectorizer, CountVectorizerModel}

import org.apache.spark.sql.SparkSession

object CountVector\_3 {

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

val spark: SparkSession = SparkSession.builder()

.appName("SparkMlilb")

.master("local[2]")

.getOrCreate()

spark.sparkContext.setLogLevel("WARN")

val df = spark.createDataFrame(Seq(

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

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

)).toDF("id", "words")

// fit a CountVectorizerModel from the corpus

val cvModel: CountVectorizerModel = new CountVectorizer()

.setInputCol("words")

.setOutputCol("features")

.setVocabSize(3)

.setMinDF(2)

.fit(df)

// alternatively, define CountVectorizerModel with a-priori vocabulary

val cvm = new CountVectorizerModel(Array("a", "b", "c"))

.setInputCol("words")

.setOutputCol("features")

cvModel.transform(df).select("features").show()

}

}

结果:

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

| features|

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

|(3,[0,1,2],[1.0,1...|

|(3,[0,1,2],[2.0,2...|

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

## 1.2特征转换

### 分词器

分词是将文本（例如句子）分解为单个术语（通常是单词）的过程。一个简单的[Tokenizer](http://spark.apache.org/docs/2.0.2/api/scala/index.html" \l "org.apache.spark.ml.feature.Tokenizer)类提供此功能。下面的示例显示了如何将句子拆分为单词序列。

[RegexTokenizer](http://spark.apache.org/docs/2.0.2/api/scala/index.html" \l "org.apache.spark.ml.feature.RegexTokenizer)允许基于正则表达式（正则表达式）匹配的更高级标记化。默认情况下，参数“pattern”（正则表达式，默认值:) "\\s+"用作分隔符以分割输入文本。或者，用户可以将参数“gap”设置为false，指示正则表达式“pattern”表示“令牌”而不是分割间隙，并找到所有匹配的出现作为标记化结果。

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

import org.apache.spark.sql.SparkSession

object Tokenizener {

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

val spark: SparkSession = SparkSession.builder()

.appName("SparkMlilb")

.master("local[2]")

.getOrCreate()

spark.sparkContext.setLogLevel("WARN")

val sentenceDataFrame = spark.createDataFrame(Seq(

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

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

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

)).toDF("label", "sentence")

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

val regexTokenizer = new RegexTokenizer()

.setInputCol("sentence")

.setOutputCol("words")

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

val tokenized = tokenizer.transform(sentenceDataFrame)

tokenized.select("words", "label").take(3).foreach(println)

val regexTokenized = regexTokenizer.transform(sentenceDataFrame)

regexTokenized.select("words", "label").take(3).foreach(println)

}

}

### StopWordsRemover

[停止词](https://en.wikipedia.org/wiki/Stop_words)是应该从输入中排除的词，通常是因为词经常出现而且没有那么多含义。

StopWordsRemover将字符串序列（例如，[Tokenizer](http://spark.apache.org/docs/2.0.2/ml-features.html" \l "tokenizer)的输出）作为输入，并从输入序列中删除所有停用词。停用词列表由stopWords参数指定。某些语言的默认停用词可通过调用访问StopWordsRemover.loadDefaultStopWords(language)，其中可用选项为“丹麦语”，“荷兰语”，“英语”，“芬兰语”，“法语”，“德语”，“匈牙利语”，“意大利语”，“挪威语” “，”葡萄牙语“，”俄语“，”西班牙语“，”瑞典语“和”土耳其语“。布尔参数caseSensitive指示匹配项是否区分大小写（默认为false）。

****例子****

假设我们有列如下数据帧id，并raw：

id | raw

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

0 | [I, saw, the, red, baloon]

1 | [Mary, had, a, little, lamb]

应用StopWordsRemoverwith raw作为输入列和filtered输出列，我们应该得到以下结果：

id | raw | filtered

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

0 | [I, saw, the, red, baloon] | [saw, red, baloon]

1 | [Mary, had, a, little, lamb]|[Mary, little, lamb]

在filtered，过滤掉了停用词“I”，“the”，“have”和“a”。

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

import org.apache.spark.sql.SparkSession

object StopWords\_3 {

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

val spark: SparkSession = SparkSession.builder()

.appName("SparkMlilb")

.master("local[2]")

.getOrCreate()

spark.sparkContext.setLogLevel("WARN")

val remover = new StopWordsRemover()

.setInputCol("raw")

.setOutputCol("filtered")

val dataSet = spark.createDataFrame(Seq(

(0, Seq("I", "saw", "the", "red", "baloon")),

(1, Seq("Mary", "had", "a", "little", "lamb"))

)).toDF("id", "raw")

remover.transform(dataSet).show()

}

}

结果：

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

| id| raw| filtered|

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

| 0|[I, saw, the, red...| [saw, red, baloon]|

| 1|[Mary, had, a, li...|[Mary, little, lamb]|

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

### n-gram

一个[n-gram中](https://en.wikipedia.org/wiki/N-gram)是序列n某些整数n的标记（通常是单词）ñ，它的NGram类可用于输入特征转变成ñ-grams。

NGram将字符串序列（例如，[Tokenizer](http://spark.apache.org/docs/2.0.2/ml-features.html" \l "tokenizer)的输出）作为输入。该参数n用于确定每个n中的术语ñ-gram。输出将由n序列组成nms每个ñ-gram由以空格分隔的n字符串表示ñ连续的话。如果输入序列包含少于n字符串，则不会生成输出。

1. gram----P(word|hello)
2. gram---P(word|hello,hi)

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

import org.apache.spark.sql.SparkSession

object nGram\_3 {

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

val spark: SparkSession = SparkSession.builder()

.appName("SparkMlilb")

.master("local[2]")

.getOrCreate()

spark.sparkContext.setLogLevel("WARN")

val wordDataFrame = spark.createDataFrame(Seq(

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

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

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

)).toDF("label", "words")

val ngram = new NGram().setInputCol("words").setOutputCol("ngrams")

val ngramDataFrame = ngram.transform(wordDataFrame)

ngramDataFrame.take(3).map(\_.getAs[Stream[String]]("ngrams").toList).foreach(println)

}

}

输出结果：

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)

### 二值化

二值化是将数值特征阈值化为二进制（0/1）特征的过程。

Binarizer取共同参数inputCol和outputCol，以及所述threshold 二值化。大于阈值的特征值被二进制化为1.0; 等于或小于阈值的值被二值化为0.0。支持Vector和Double类型inputCol。

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

import org.apache.spark.sql.SparkSession

object Binaziner\_3 {

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

val spark: SparkSession = SparkSession.builder()

.appName("SparkMlilb")

.master("local[2]")

.getOrCreate()

spark.sparkContext.setLogLevel("WARN")

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

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

val binarizer: Binarizer = new Binarizer()

.setInputCol("feature")

.setOutputCol("binarized\_feature")

.setThreshold(0.5)

val binarizedDataFrame = binarizer.transform(dataFrame)

val binarizedFeatures = binarizedDataFrame.select("binarized\_feature")

binarizedFeatures.collect().foreach(println)

}

}

结果：

[0.0]

[1.0]

[0.0]

### PCA

[PCA](http://en.wikipedia.org/wiki/Principal_component_analysis)是一种统计过程，它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一组值。[PCA](http://spark.apache.org/docs/2.0.2/api/scala/index.html" \l "org.apache.spark.ml.feature.PCA)类使用[PCA](http://spark.apache.org/docs/2.0.2/api/scala/index.html" \l "org.apache.spark.ml.feature.PCA)训练模型以将向量投影到低维空间。下面的示例显示了如何将5维特征向量投影为3维主成分。

import org.apache.spark.ml.feature.PCA

import org.apache.spark.ml.linalg.Vectors

import org.apache.spark.sql.SparkSession

object PCA\_3 {

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

val spark: SparkSession = SparkSession.builder()

.appName("SparkMlilb")

.master("local[2]")

.getOrCreate()

spark.sparkContext.setLogLevel("WARN")

val data = Array(

Vectors.sparse(5, Seq((1, 1.0), (3, 7.0))),

Vectors.dense(2.0, 0.0, 3.0, 4.0, 5.0),

Vectors.dense(4.0, 0.0, 0.0, 6.0, 7.0)

)

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

val pca = new PCA()

.setInputCol("features")

.setOutputCol("pcaFeatures")

.setK(3)

.fit(df)

val pcaDF = pca.transform(df)

val result = pcaDF.select("pcaFeatures")

result.show()

}

}

Result：

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

| pcaFeatures|

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

|[1.64857282308838...|

|[-4.6451043317815...|

|[-6.4288805356764...|

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

### PolynomialExpansion

[多项式展开](http://en.wikipedia.org/wiki/Polynomial_expansion)是将[要素扩展](http://en.wikipedia.org/wiki/Polynomial_expansion)为多项式空间的过程，该多项式空间由原始维度的n度组合制定。一个[PolynomialExpansion](http://spark.apache.org/docs/2.0.2/api/scala/index.html" \l "org.apache.spark.ml.feature.PolynomialExpansion)类提供此功能。以下示例显示如何将要素扩展为3次多项式空间。

import org.apache.spark.ml.feature.PolynomialExpansion

import org.apache.spark.ml.linalg.Vectors

import org.apache.spark.sql.SparkSession

object Poly\_3 {

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

val spark: SparkSession = SparkSession.builder()

.appName("SparkMlilb")

.master("local[2]")

.getOrCreate()

spark.sparkContext.setLogLevel("WARN")

val data = Array(

Vectors.dense(-2.0, 2.3),

Vectors.dense(0.0, 0.0),

Vectors.dense(0.6, -1.1)

)

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

val polynomialExpansion = new PolynomialExpansion()

.setInputCol("features")

.setOutputCol("polyFeatures")

.setDegree(3)

val polyDF = polynomialExpansion.transform(df)

polyDF.select("polyFeatures").take(3).foreach(println)

}

}

Result:

[[-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]]

### 离散余弦变换（DCT）

离散余弦变换将时域内的一个长度为N的实值序列变换为频域内的另一个长度为N的实值序列。

import org.apache.spark.ml.feature.DCT

import org.apache.spark.ml.linalg.Vectors

import org.apache.spark.sql.SparkSession

object DCT\_3 {

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

val spark: SparkSession = SparkSession.builder()

.appName("SparkMlilb")

.master("local[2]")

.getOrCreate()

spark.sparkContext.setLogLevel("WARN")

val data = Seq(

Vectors.dense(0.0, 1.0, -2.0, 3.0),

Vectors.dense(-1.0, 2.0, 4.0, -7.0),

Vectors.dense(14.0, -2.0, -5.0, 1.0))

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

val dct = new DCT()

.setInputCol("features")

.setOutputCol("featuresDCT")

.setInverse(false)

val dctDf = dct.transform(df)

dctDf.select("featuresDCT").show(3)

}

}

结果：

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

| featuresDCT|

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

|[1.0,-1.148050297...|

|[-1.0,3.378492794...|

|[4.0,9.3044534219...|

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

### StringIndexer

StringIndexer将标签的字符串列编码为标签索引列。索引[0, numLabels)按标签频率排序，因此最常用的标签获得索引0。如果输入列是数字，我们将其转换为字符串并索引字符串值。当下游管道组件（例如Estimator或 Transformer使用此字符串索引标签）时，必须将组件的输入列设置为此字符串索引列名称。在许多情况下，您可以使用设置输入列setInputCol。

****例子****

假设我们有列如下数据帧id，并category：

id | category

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

0 | a

1 | b

2 | c

3 | a

4 | a

5 | c

category是一个带有三个标签的字符串列：“a”，“b”和“c”。应用StringIndexerwith category作为输入列和categoryIndex输出列，我们应该得到以下结果：

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”得到索引，0因为它是最常见的，其次是带有索引的“c”和带索引的1“b” 2。

此外，StringIndexer当您StringIndexer在一个数据集上拟合然后使用它来转换另一个数据集时，有两种策略可以处理看不见的标签：

* 抛出异常（这是默认值）
* 完全跳过包含看不见的标签的行

****例子****

让我们回到之前的示例，但这次重用我们之前StringIndexer在以下数据集上定义 的内容：

id | category

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

0 | a

1 | b

2 | c

3 | d

如果您没有设置如何StringIndexer处理看不见的标签或将其设置为“错误”，则会抛出异常。但是，如果您已调用setHandleInvalid("skip")，则将生成以下数据集：

id | category | categoryIndex

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

0 | a | 0.0

1 | b | 2.0

2 | c | 1.0

请注意，不显示包含“d”的行。

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

import org.apache.spark.sql.SparkSession

object StringIndex {

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

val spark: SparkSession = SparkSession.builder()

.appName("SparkMlilb")

.master("local[2]")

.getOrCreate()

spark.sparkContext.setLogLevel("WARN")

val df = spark.createDataFrame(

Seq((0, "a"), (1, "b"), (2, "c"), (3, "a"), (4, "a"), (5, "c"))

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

val indexer = new StringIndexer()

.setInputCol("category")

.setOutputCol("categoryIndex")

val indexed = indexer.fit(df).transform(df)

indexed.show()

}

}

结果：

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

| 0| a| 0.0|

| 1| b| 2.0|

| 2| c| 1.0|

| 3| a| 0.0|

| 4| a| 0.0|

| 5| c| 1.0|

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

### IndexToString

对称地StringIndexer，IndexToString将一列标签索引映射回包含原始标签作为字符串的列。一个常见的用例是从标签生成索引StringIndexer，使用这些索引训练模型，并从预测索引列中检索原始标签IndexToString。但是，您可以自由提供自己的标签。

****例子****

对StringIndexer例子，假设我们有一个栏目下面的数据帧id，并categoryIndex：

id | categoryIndex

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

0 | 0.0

1 | 2.0

2 | 1.0

3 | 0.0

4 | 0.0

5 | 1.0

应用IndexToString与categoryIndex作为输入列， originalCategory作为输出列，我们能找回我们原来的标签（它们将从列的元数据来推断）：

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

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

import org.apache.spark.sql.SparkSession

object indexTosTRING {

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

val spark: SparkSession = SparkSession.builder()

.appName("SparkMlilb")

.master("local[2]")

.getOrCreate()

spark.sparkContext.setLogLevel("WARN")

val df = spark.createDataFrame(Seq(

(0, "a"),

(1, "b"),

(2, "c"),

(3, "a"),

(4, "a"),

(5, "c")

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

val indexer = new StringIndexer()

.setInputCol("category")

.setOutputCol("categoryIndex")

.fit(df)

val indexed = indexer.transform(df)

val converter = new IndexToString()

.setInputCol("categoryIndex")

.setOutputCol("originalCategory")

val converted = converter.transform(indexed)

converted.select("id", "originalCategory").show()

}

}

结果：

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

| id|originalCategory|

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

| 0| a|

| 1| b|

| 2| c|

| 3| a|

| 4| a|

| 5| c|

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

### OneHotEncoder

[单热编码](http://en.wikipedia.org/wiki/One-hot)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。此编码允许期望连续特征（例如Logistic回归）的算法使用分类特征。

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

import org.apache.spark.sql.SparkSession

object OneHotEncoder {

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

val spark: SparkSession = SparkSession.builder()

.appName("SparkMlilb")

.master("local[2]")

.getOrCreate()

spark.sparkContext.setLogLevel("WARN")

val df = spark.createDataFrame(Seq(

(0, "a"),

(1, "b"),

(2, "c"),

(3, "a"),

(4, "a"),

(5, "c")

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

val indexer = new StringIndexer()

.setInputCol("category")

.setOutputCol("categoryIndex")

.fit(df)

val indexed = indexer.transform(df)

val encoder = new OneHotEncoder()

.setInputCol("categoryIndex")

.setOutputCol("categoryVec")

val encoded = encoder.transform(indexed)

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

}

}

结果：

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

| id| categoryVec|

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

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

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

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

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

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

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

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

### VectorIndexer

VectorIndexer帮助索引Vectors的数据集中的分类特征。它既可以自动确定哪些特征是分类的，也可以将原始值转换为类别索引。具体来说，它执行以下操作：

1. 获取[Vector](http://spark.apache.org/docs/2.0.2/api/scala/index.html" \l "org.apache.spark.mllib.linalg.Vector)类型的输入列和参数maxCategories。
2. 根据不同值的数量确定哪些maxCategories要素应该是分类的，其中最多的要素被声明为分类。
3. 为每个分类要素计算基于0的类别索引。
4. 索引分类要素并将原始要素值转换为索引。

索引分类特征允许决策树和树集合等算法适当地处理分类特征，从而提高性能。

在下面的示例中，我们读入标记点的数据集，然后用于VectorIndexer确定哪些要素应被视为分类。我们将分类特征值转换为它们的索引。然后可以将这种转换后的数据传递给DecisionTreeRegressor处理分类特征的算法。

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

import org.apache.spark.sql.SparkSession

object Vector\_3 {

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

val spark: SparkSession = SparkSession.builder()

.appName("SparkMlilb")

.master("local[2]")

.getOrCreate()

spark.sparkContext.setLogLevel("WARN")

val data = spark.read.format("libsvm").load("D:\\BigData\\Workspace\\Spark\_Test\\SaprkMLFoundation\\src\\main\\scala\\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(", "))

// Create new column "indexed" with categorical values transformed to indices

val indexedData = indexerModel.transform(data)

indexedData.show()

}

}

结果：

Chose 351 categorical features: 645, 69, 365, 138, 101, 479, 333, 249, 0, 555, 666, 88, 170, 115, 276, 308, 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, 619, 196, 559, 638, 20, 421, 46, 93, 284, 228, 448, 57, 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, 60, 117, 678, 529, 85, 201, 220, 366, 534, 102, 334, 28, 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, 141, 610, 420, 109, 256, 225, 339, 77, 193, 669, 476, 642, 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, 161, 279, 64, 689, 17, 149, 584, 562, 176, 423, 191, 22, 44, 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, 76, 7, 39, 649, 98, 616, 477, 367, 535, 103, 140, 621, 91, 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, 112, 617, 194, 145, 48, 557, 690, 63, 640, 18, 282, 95, 310, 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, 203, 671, 688, 104, 368, 588, 40, 304, 26, 258, 390, 55, 114, 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, 506, 79, 195, 474, 664, 532, 94, 283, 395, 332, 528, 644, 47, 15, 163, 200, 68, 62, 277, 691, 501, 90, 111, 254, 227, 337, 122, 83, 309, 560, 639, 676, 222, 592, 364, 100

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

|label| features| indexed|

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

| 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...|

| 1.0|(692,[97,98,99,12...|(692,[97,98,99,12...|

| 1.0|(692,[124,125,126...|(692,[124,125,126...|

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

only showing top 20 rows

### 正规化

import org.apache.spark.sql.SparkSession

/\*\*

\* Created by zhao-chj on 2018/8/13.

\*/

object Normalization\_3 {

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

import org.apache.spark.ml.feature.Normalizer

val spark: SparkSession = SparkSession.builder()

.appName("SparkMlilb")

.master("local[2]")

.getOrCreate()

spark.sparkContext.setLogLevel("WARN")

val dataFrame = spark.read.format("libsvm").load("D:\\BigData\\Workspace\\Spark\_Test\\SaprkMLFoundation\\src\\main\\scala\\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()

}

}

结果：

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

|label| features| normFeatures|

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

| 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...|

| 1.0|(692,[97,98,99,12...|(692,[97,98,99,12...|

| 1.0|(692,[124,125,126...|(692,[124,125,126...|

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

only showing top 20 rows

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

|label| features| normFeatures|

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

| 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...|

| 1.0|(692,[97,98,99,12...|(692,[97,98,99,12...|

| 1.0|(692,[124,125,126...|(692,[124,125,126...|

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

only showing top 20 rows

### StandardScaler

StandardScaler转换Vector行的数据集，将每个要素标准化以具有单位标准差和/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。在缩放之前使用均值将数据居中。它将构建一个密集的输出，因此这不适用于稀疏输入并将引发异常。

StandardScaler是一个Estimator可以fit在数据集上产生一个StandardScalerModel; 这等于计算摘要统计。然后，模型可以变换Vector数据集中的列以具有单位标准偏差和/或零均值特征。

请注意，如果要素的标准差为零，则会返回该要素的默认0.0值Vector。

以下示例演示如何以libsvm格式加载数据集，然后将每个要素标准化以具有单位标准偏差。

import org.apache.spark.sql.SparkSession

/\*\*

\* Created by zhao-chj on 2018/8/13.

\*/

object StandScater {

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

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

val spark: SparkSession = SparkSession.builder()

.appName("SparkMlilb")

.master("local[2]")

.getOrCreate()

spark.sparkContext.setLogLevel("WARN")

val dataFrame = spark.read.format("libsvm").load("D:\\BigData\\Workspace\\Spark\_Test\\SaprkMLFoundation\\src\\main\\scala\\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()

}

}

结果：

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

|label| features| scaledFeatures|

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

| 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...|

| 1.0|(692,[97,98,99,12...|(692,[97,98,99,12...|

| 1.0|(692,[124,125,126...|(692,[124,125,126...|

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

only showing top 20 rows

### MinMaxScaler

import org.apache.spark.ml.feature.MinMaxScaler

import org.apache.spark.sql.SparkSession

object VectorSlicer\_3 {

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

val spark: SparkSession = SparkSession.builder()

.appName("SparkMlilb")

.master("local[2]")

.getOrCreate()

spark.sparkContext.setLogLevel("WARN")

val dataFrame = spark.read.format("libsvm").load("D:\\BigData\\Workspace\\Spark\_Test\\SaprkMLFoundation\\src\\main\\scala\\sample\_libsvm\_data.txt")

val scaler = new MinMaxScaler()

.setInputCol("features")

.setOutputCol("scaledFeatures")

// Compute summary statistics and generate MinMaxScalerModel

val scalerModel = scaler.fit(dataFrame)

// rescale each feature to range [min, max].

val scaledData = scalerModel.transform(dataFrame)

scaledData.show()

}

}

结果：

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

|label| features| scaledFeatures|

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

| 0.0|(692,[127,128,129...|[0.5,0.5,0.5,0.5,...|

| 1.0|(692,[158,159,160...|[0.5,0.5,0.5,0.5,...|

| 1.0|(692,[124,125,126...|[0.5,0.5,0.5,0.5,...|

| 1.0|(692,[152,153,154...|[0.5,0.5,0.5,0.5,...|

| 1.0|(692,[151,152,153...|[0.5,0.5,0.5,0.5,...|

| 0.0|(692,[129,130,131...|[0.5,0.5,0.5,0.5,...|

| 1.0|(692,[158,159,160...|[0.5,0.5,0.5,0.5,...|

| 1.0|(692,[99,100,101,...|[0.5,0.5,0.5,0.5,...|

| 0.0|(692,[154,155,156...|[0.5,0.5,0.5,0.5,...|

| 0.0|(692,[127,128,129...|[0.5,0.5,0.5,0.5,...|

| 1.0|(692,[154,155,156...|[0.5,0.5,0.5,0.5,...|

| 0.0|(692,[153,154,155...|[0.5,0.5,0.5,0.5,...|

| 0.0|(692,[151,152,153...|[0.5,0.5,0.5,0.5,...|

| 1.0|(692,[129,130,131...|[0.5,0.5,0.5,0.5,...|

| 0.0|(692,[154,155,156...|[0.5,0.5,0.5,0.5,...|

| 1.0|(692,[150,151,152...|[0.5,0.5,0.5,0.5,...|

| 0.0|(692,[124,125,126...|[0.5,0.5,0.5,0.5,...|

| 0.0|(692,[152,153,154...|[0.5,0.5,0.5,0.5,...|

| 1.0|(692,[97,98,99,12...|[0.5,0.5,0.5,0.5,...|

| 1.0|(692,[124,125,126...|[0.5,0.5,0.5,0.5,...|

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

only showing top 20 rows

### MaxAbsScaler

import org.apache.spark.sql.SparkSession

/\*\*

\* Created by zhao-chj on 2018/8/13.

\*/

object MaxAbsScaler {

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

import org.apache.spark.ml.feature.MaxAbsScaler

val spark: SparkSession = SparkSession.builder()

.appName("SparkMlilb")

.master("local[2]")

.getOrCreate()

spark.sparkContext.setLogLevel("WARN")

val dataFrame = spark.read.format("libsvm").load("D:\\BigData\\Workspace\\Spark\_Test\\SaprkMLFoundation\\src\\main\\scala\\sample\_libsvm\_data.txt")

val scaler = new MaxAbsScaler()

.setInputCol("features")

.setOutputCol("scaledFeatures")

// Compute summary statistics and generate MaxAbsScalerModel

val scalerModel = scaler.fit(dataFrame)

// rescale each feature to range [-1, 1]

val scaledData = scalerModel.transform(dataFrame)

scaledData.show()

}

}

结果：

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

|label| features| scaledFeatures|

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

| 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...|

| 1.0|(692,[97,98,99,12...|(692,[97,98,99,12...|

| 1.0|(692,[124,125,126...|(692,[124,125,126...|

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

only showing top 20 rows

### Bucketizer

Bucketizer将一列连续特征转换为一列要素存储区，其中存储区由用户指定。它需要一个参数：

splits：用于**将连续要素映射到存储桶的参数**。对于n + 1个分裂，有n个桶。由splits x，y定义的存储区包含除最后一个存储区之外的[x，y]范围内的值，该存储区还包括y。拆分应该严格增加。必须明确提供-inf，inf处的值以涵盖所有Double值; 否则，指定的拆分之外的值将被视为错误。两个例子splits是Array(Double.NegativeInfinity, 0.0, 1.0, Double.PositiveInfinity)和Array(0.0, 1.0, 2.0)。

请注意，如果您不知道目标列的上限和下限，则应添加Double.NegativeInfinity和Double.PositiveInfinity作为拆分的边界，以防止可能超出Bucketizer边界异常。

另请注意，您提供的分割必须严格按顺序增加，即s0 < s1 < s2 < ... < sn。

有关详细信息，请[参阅Bucketizer](http://spark.apache.org/docs/2.0.2/api/scala/index.html" \l "org.apache.spark.ml.feature.Bucketizer)的API文档。

下面是如何将一列Doubles 拼接成另一个索引列。

import org.apache.spark.sql.SparkSession

/\*\*

\* Created by zhao-chj on 2018/8/13.

\*/

object Bucketizer {

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

import org.apache.spark.ml.feature.Bucketizer

val spark: SparkSession = SparkSession.builder()

.appName("SparkMlilb")

.master("local[2]")

.getOrCreate()

spark.sparkContext.setLogLevel("WARN")

val splits = Array(Double.NegativeInfinity, -0.5, 0.0, 0.5, Double.PositiveInfinity)

val data = Array(-0.5, -0.3, 0.0, 0.2)

val dataFrame = spark.createDataFrame(data.map(Tuple1.apply)).toDF("features")

val bucketizer = new Bucketizer()

.setInputCol("features")

.setOutputCol("bucketedFeatures")

.setSplits(splits)

// Transform original data into its bucket index.

val bucketedData = bucketizer.transform(dataFrame)

bucketedData.show()

}

}

结果：

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

|features|bucketedFeatures|

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

| -0.5| 1.0|

| -0.3| 1.0|

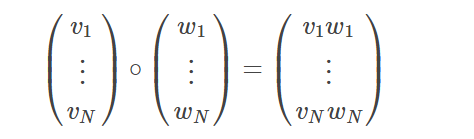
| 0.0| 2.0|

| 0.2| 2.0|

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

### ElementwiseProduct

ElementwiseProduct使用逐元素乘法将每个输入向量乘以提供的“权重”向量。换句话说，它通过标量乘数缩放数据集的每一列。这表示输入矢量和变换矢量之间的[Hadamard乘积](https://en.wikipedia.org/wiki/Hadamard_product_(matrices))，以产生结果矢量。



下面的示例演示了如何使用变换矢量值变换矢量。

import org.apache.spark.sql.SparkSession

/\*\*

\* Created by zhao-chj on 2018/8/13.

\*/

object ElementwiseProdcuts {

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

import org.apache.spark.ml.feature.ElementwiseProduct

import org.apache.spark.ml.linalg.Vectors

val spark: SparkSession = SparkSession.builder()

.appName("SparkMlilb")

.master("local[2]")

.getOrCreate()

spark.sparkContext.setLogLevel("WARN")

// Create some vector data; also works for sparse vectors

val dataFrame = spark.createDataFrame(Seq(

("a", Vectors.dense(1.0, 2.0, 3.0)),

("b", Vectors.dense(4.0, 5.0, 6.0)))).toDF("id", "vector")

val transformingVector = Vectors.dense(0.0, 1.0, 2.0)

val transformer = new ElementwiseProduct()

.setScalingVec(transformingVector)

.setInputCol("vector")

.setOutputCol("transformedVector")

// Batch transform the vectors to create new column:

transformer.transform(dataFrame).show()

}

}

结果：

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

| 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]|

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

### SQLTransformer

SQLTransformer实现由SQL语句定义的转换。目前我们只支持SQL语法，例如"SELECT ... FROM \_\_THIS\_\_ ..." where "\_\_THIS\_\_"表示输入数据集的基础表。select子句指定要在输出中显示的字段，常量和表达式，并且可以是Spark SQL支持的任何select子句。用户还可以使用Spark SQL内置函数和UDF对这些选定列进行操作。例如，SQLTransformer支持如下语句：

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

****例子****

假设我们有以下带有列的DataFrame id，v1并且v2：

id | v1 | v2

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

0 | 1.0 | 3.0

2 | 2.0 | 5.0

这是SQLTransformerwith语句的输出"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

代码：

import org.apache.spark.ml.feature.SQLTransformer

import org.apache.spark.sql.SparkSession

object SqlTransformer {

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

val spark: SparkSession = SparkSession.builder()

.appName("SparkMlilb")

.master("local[2]")

.getOrCreate()

spark.sparkContext.setLogLevel("WARN")

val df = spark.createDataFrame(

Seq((0, 1.0, 3.0), (2, 2.0, 5.0))).toDF("id", "v1", "v2")

val sqlTrans = new SQLTransformer().setStatement(

"SELECT \*, (v1 + v2) AS v3, (v1 \* v2) AS v4 FROM \_\_THIS\_\_")

sqlTrans.transform(df).show()

}

}

结果：

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

| id| v1| v2| v3| v4|

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

| 0|1.0|3.0|4.0| 3.0|

| 2|2.0|5.0|7.0|10.0|

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

### VectorAssembler

VectorAssembler是一个变换器，它将给定的列表组合到一个向量列中。将原始特征和由不同特征变换器生成的特征组合成单个特征向量非常有用，以便训练ML模型，如逻辑回归和决策树。 VectorAssembler接受以下输入列类型：所有数字类型，布尔类型和矢量类型。在每一行中，输入列的值将按指定的顺序连接到一个向量中。

****例子****

假设我们有与列的数据帧id，hour，mobile，userFeatures，和clicked：

id | hour | mobile | userFeatures | clicked

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

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

userFeatures是一个包含三个用户特征的矢量列。我们要结合起来hour，mobile和userFeatures成称为单一的特征向量**features**，并用它来预测clicked或没有。如果我们设置VectorAssembler的输入列hour，mobile以及userFeatures输出列features，改造后我们应该得到以下数据框：

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]

import org.apache.spark.sql.SparkSession

/\*\*

\* Created by zhao-chj on 2018/8/13.

\*/

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

import org.apache.spark.ml.linalg.Vectors

object VectorSlicer\_2 {

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

val spark: SparkSession = SparkSession.builder()

.appName("SparkMlilb")

.master("local[2]")

.getOrCreate()

spark.sparkContext.setLogLevel("WARN")

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())

}

结果：

[[18.0,1.0,0.0,10.0,0.5],1.0]

### QuantileDiscretizer

连续属性离散化

QuantileDiscretizer采用具有连续特征的列，并输出具有**分箱分类特征**的列。箱数由numBuckets参数设定。使用近似算法选择bin范围（有关详细说明，请参阅[aboutQuantile](http://spark.apache.org/docs/2.0.2/api/scala/index.html" \l "org.apache.spark.sql.DataFrameStatFunctions)的文档 ）。可以使用relativeError参数控制近似的精度 。设置为零时，将计算精确分位数（****注意：****计算精确分位数是一项昂贵的操作）。下部和上部箱边界会-Infinity和+Infinity涵盖所有实际值。

****例子****

假设我们有与列的数据帧id，hour：

id | hour

----|------

0 | 18.0

----|------

1 | 19.0

----|------

2 | 8.0

----|------

3 | 5.0

----|------

4 | 2.2

hour是一个连续的Double类型的功能。我们希望将连续特征变为分类特征。鉴于**numBuckets = 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

import org.apache.spark.sql.SparkSession

/\*\*

\* Created by zhao-chj on 2018/8/13.

\*/

object QuantileDiscreator {

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

import org.apache.spark.ml.feature.QuantileDiscretizer

val spark: SparkSession = SparkSession.builder()

.appName("SparkMlilb")

.master("local[2]")

.getOrCreate()

spark.sparkContext.setLogLevel("WARN")

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()

}

}

结果：

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

| 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|

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

## 1.3特征选择器

### VectorSlicer

VectorSlicer是一个变换器，它采用一个特征向量，并输出一个带有原始特征子阵列的新特征向量。它对于从向量列中提取要素非常有用。

VectorSlicer接受具有指定索引的向量列，然后输出一个新的向量列，其值通过这些索引选择。有两种类型的指数，

1.表示向量索引的整数索引setIndices()。

2.字符串索引，表示向量中要素的名称，setNames()。 *这要求向量列具有一个，3.AttributeGroup因为实现在一个名称字段上匹配Attribute。*

整数和字符串的规范都是可以接受的。而且，您可以同时使用整数索引和字符串名称。必须至少选择一个功能。不允许使用重复的功能，因此所选索引和名称之间不能重叠。请注意，如果选择了要素名称，则在遇到空输入属性时将引发异常。

输出向量将首先按所选索引（按给定顺序）排序要素，然后是所选名称（按给定顺序）。

****例子****

假设我们有一个带有列的DataFrame userFeatures：

userFeatures

------------------

[0.0, 10.0, 0.5]

userFeatures是一个包含三个用户功能的矢量列。假设第一列userFeatures都是零，所以我们要删除它并仅选择最后两列。该VectorSlicer选择与最后两个元件setIndices(1, 2)然后产生一个名为新向量列features：

userFeatures | features

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

[0.0, 10.0, 0.5] | [10.0, 0.5]

假设我们还有潜在的输入属性userFeatures，即 ["f1", "f2", "f3"]我们可以setNames("f2", "f3")用来选择它们。

userFeatures | features

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

[0.0, 10.0, 0.5] | [10.0, 0.5]

["f1", "f2", "f3"] | ["f2", "f3"]

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, SparkSession}

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

object VectorSlicer\_4 {

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

import java.util.Arrays

val spark: SparkSession = SparkSession.builder()

.appName("SparkMlilb")

.master("local[2]")

.getOrCreate()

spark.sparkContext.setLogLevel("WARN")

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

val defaultAttr = NumericAttribute.defaultAttr

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

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

val dataset = spark.createDataFrame(data, StructType(Array(attrGroup.toStructField())))

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

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

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

val output = slicer.transform(dataset)

println(output.select("userFeatures", "features").first())

}

}

结果：

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

### RFormula

RFormula选择由[R模型公式](https://stat.ethz.ch/R-manual/R-devel/library/stats/html/formula.html)指定的列。目前，我们支持R运算符的有限子集，包括'〜'，'。'，'：'，'+'和' - '。基本的运营商是：

* ~ 单独的目标和条款
* + concat术语，“+ 0”表示删除拦截
* - 删除一个术语，“ - 1”表示删除拦截
* : 交互（数值乘法或二进制分类值）
* . 除目标之外的所有列

假设a并且b是双列，我们使用以下简单示例来说明以下效果RFormula：

* y ~ a + b意味着模型y ~ w0 + w1 \* a + w2 \* b在哪里w0是截距并且w1, w2是系数。
* y ~ a + b + a:b - 1表示模型y ~ w1 \* a + w2 \* b + w3 \* a \* b在哪里w1, w2, w3是系数。

RFormula生成一个特征向量列和一个标签的双列或字符串列。就像在R中使用公式进行线性回归一样，字符串输入列将是单热编码的，而数字列将被转换为双精度。如果label列的类型为string，则首先将其转换为double StringIndexer。如果DataFrame中不存在标签列，则将从公式中的指定响应变量创建输出标签列。

****例子****

假设我们有与列的数据帧id，country，hour，和clicked：

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RFormula公式字符串clicked ~ country + hour，表示我们想要clicked根据转换进行预测，country并且hour在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

代码：

import org.apache.spark.ml.feature.RFormula

import org.apache.spark.sql.SparkSession

object RSelectElement {

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

val spark: SparkSession = SparkSession.builder()

.appName("SparkMlilb")

.master("local[2]")

.getOrCreate()

spark.sparkContext.setLogLevel("WARN")

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()

}

}

结果：

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

| 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|

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

### ChiSqSelector

ChiSqSelector代表Chi-Squared特征选择。它使用具有分类特征的标记数据进行操作。ChiSqSelector根据 类[的独立性Chi-Squared测试来](https://en.wikipedia.org/wiki/Chi-squared_test)命令特征 ，然后过滤（选择）类标签最依赖的顶级特征。这类似于产生具有最强预测能力的特征。

****例子****

假设我们有与列的数据帧id，features和clicked，这是用来作为预测我们的目标：

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

如果我们用ChiSqSelector用numTopFeatures = 1，然后根据我们的标签clicked在我们的最后一列features被选为最有用的功能：

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]

代码：

/\*\*

\* Created by zhao-chj on 2018/8/13.

\* 运行代码出现问题

\*/

import org.apache.spark.ml.feature.ChiSqSelector

import org.apache.spark.ml.linalg.Vectors

import org.apache.spark.sql.SparkSession

object chiSquare {

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

val spark: SparkSession = SparkSession.builder()

.appName("SparkMlilb")

.master("local[2]")

.getOrCreate()

spark.sparkContext.setLogLevel("WARN")

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)

result.show()

}

}

# R与SparkR

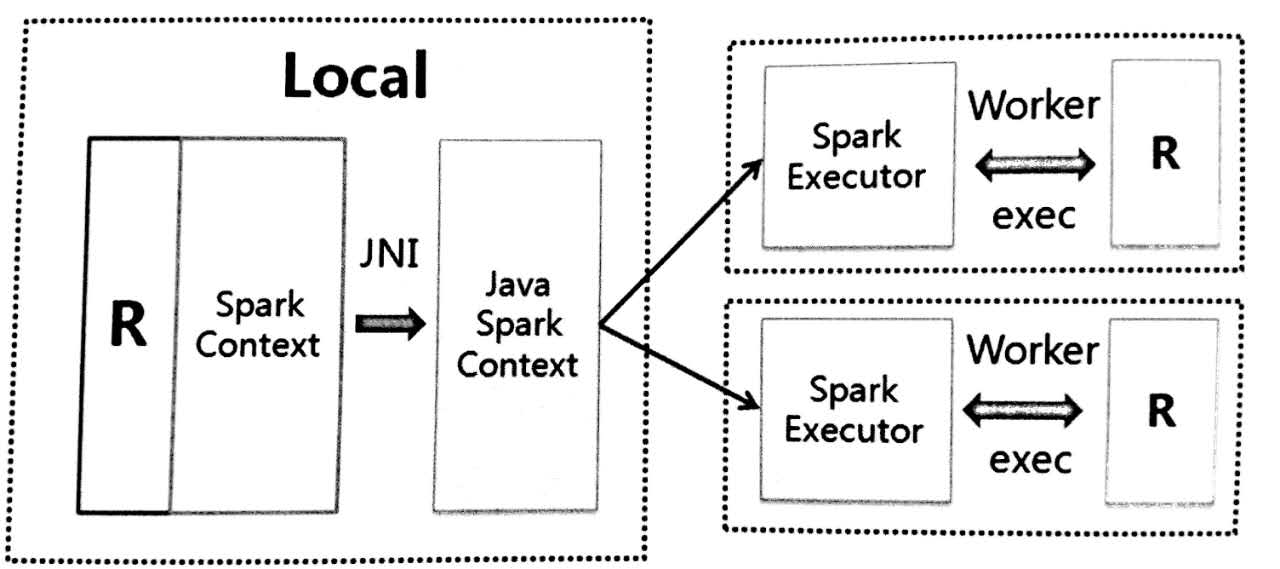
R是遵循GNU协议的一款开源、免费的软件，广泛的应用于统计计算和统计制图，但是它只能单机运行。为了能够使用R语言分析大规模分布式数据，伯克利分校的AMP实验室开发了SparkR，并在1.4版本中加入了该组件。通过SparkR可以分析大规模的数据集，并通过R shell交互式地在SparkR上运行作业。

SparkR特性如下：

1. 提供了Spark中弹性式分布式数据集（RDDs）的API，用户可以在集群上通过R shell交互性地运行Spark任务。
2. 支持序列化闭包功能，可以将用户定义函数中引用到的变量自动序列化发送到集群中其他机器上。
3. SparkR还可以很容易地调用R开发包，只需要在集群上执行操作前用install.package安装R包即可。

注意：Apache Spark产品版本更新周期一般为三个月，Spark1.6.0版本更新于2016年1月4日。Apache Saprk1.3版本包含有DataFrames API和ML Pipelines API。自Apache Spark1.4版本之后，官方的程序中已经自带了R的语言和界面支持，也就是SparkR。

SparkR处理流程示意图：



R与Spark的具体内容请参考课件《R与SparkR》。

同时分析了R与Python的应用场景、SparkR的安装。