**大数据电商用户画像项目**

**第五部分◎教案**

|  |  |  |
| --- | --- | --- |
| **课程教案版本** | **日期** | **备注** |
| **V1.0** | **20190515** |  |
|  |  |  |

# 第5章：挖掘类标签之SparkMllib特征工程基础实战

# 项目架构再梳理

## 项目描述

企业级360°全方位用户画像是基于电商平台进行设计和开发，是面向注册会员的偏好、行为习惯和 人口属性的画像还原，同时也包括对商品信息的画像还原。提供用户喜好和商品特征帮助营销平台提升营销的精准度，也方便个性化推荐系统快速准确的为每个用户推荐相关的商品。



1.标签体系

标签体系模块基于本体论建立，包括基础标签和组合标签两个子模块。基础标签维护了人口属性、商业属性、行为属性和用户价值4个方面。组合标签是在基础标签的基础上按TGI创建的，用于反映目标群体在特定研究范围的强势和弱势。

2.标签引擎

标签引擎模块是用户画像运转的核心实现，维护标签的具体实现逻辑，包括标签依赖的数据源、规则和模型的信息。同时提供每一个标签引擎运行状态的可视化监控。当Super User登录时会出现新增标签及其引擎的审核。

3.画像模块

画像模块是对实体（用户/物品）信息全貌可视化的精准还原，包括对个体的全方位还原和群体的全方位还原。

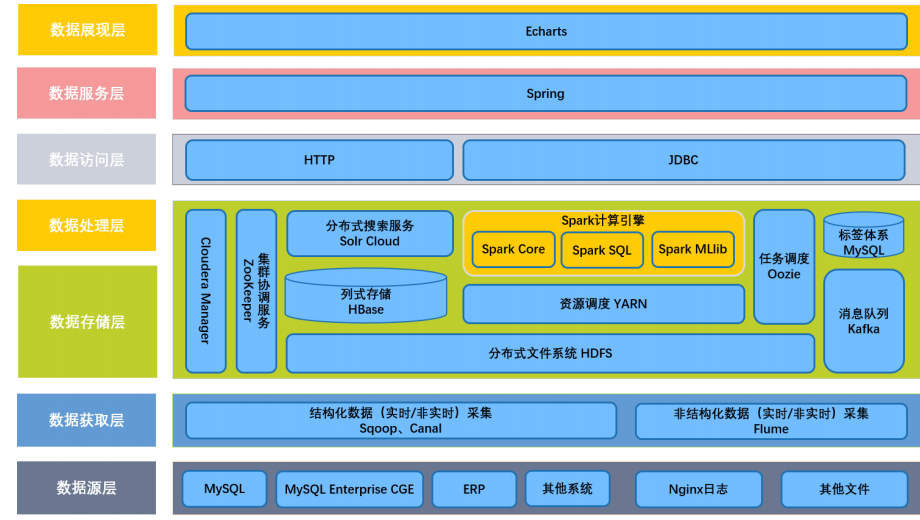
4.标签查询

标签查询模块是查看标签所覆盖的实体，用于查看标签体系中所有标签包含的商品详细信息和经过 脱敏后的用户信息。

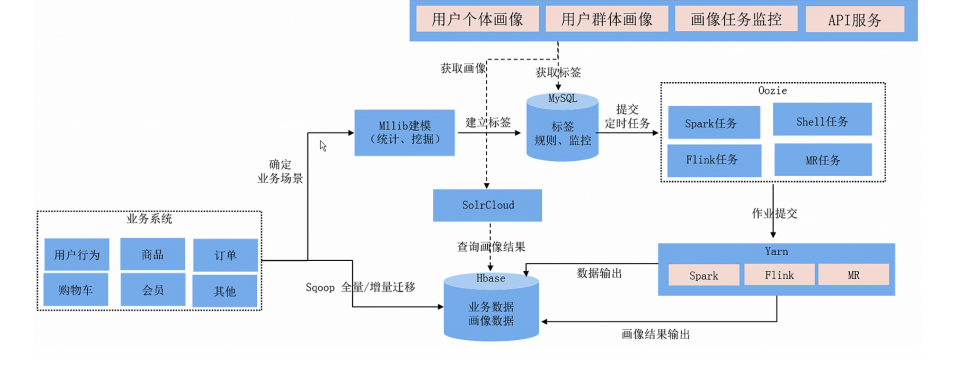
5.系统设置

系统设置模块主要包括用户管理、权限管理。

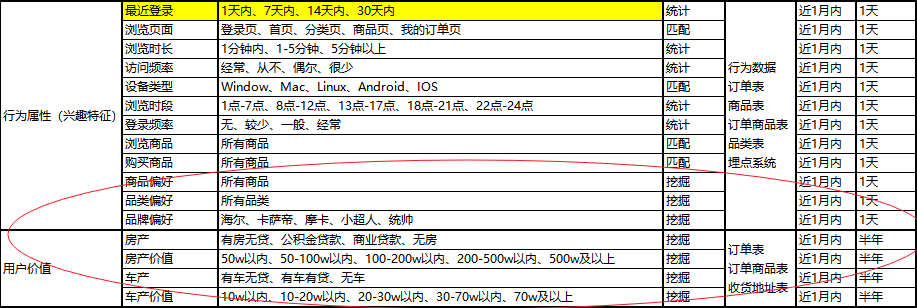
## 技术架构



## 项目功能架构



## 挖掘类标签的举例



# SparkMllib的功能和应用场景

* 学习目标
  + 了解SparkMllib框架基础知识
  + 掌握SparkMllib机器学习流程
  + 掌握SparkMllib特征工程处理方法
  + 掌握SparkMllib几种数据结构构成

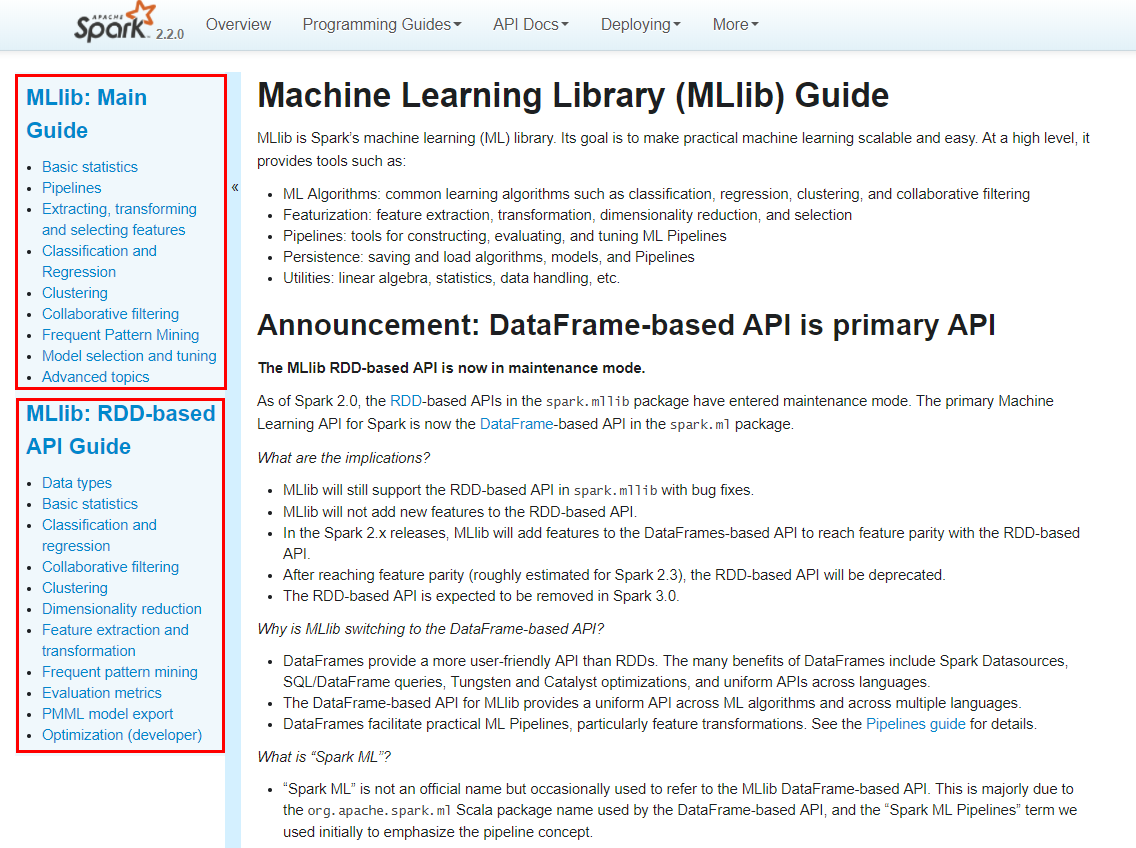
## 2.1SparkMllib简介及功能介绍

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、回归、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

## 2.3SparkML版本变迁

通过官网了解SparkMllib的版本：



* 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。究竟两者有什么区别呢？

## 2.2SparkMllib架构详解

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：

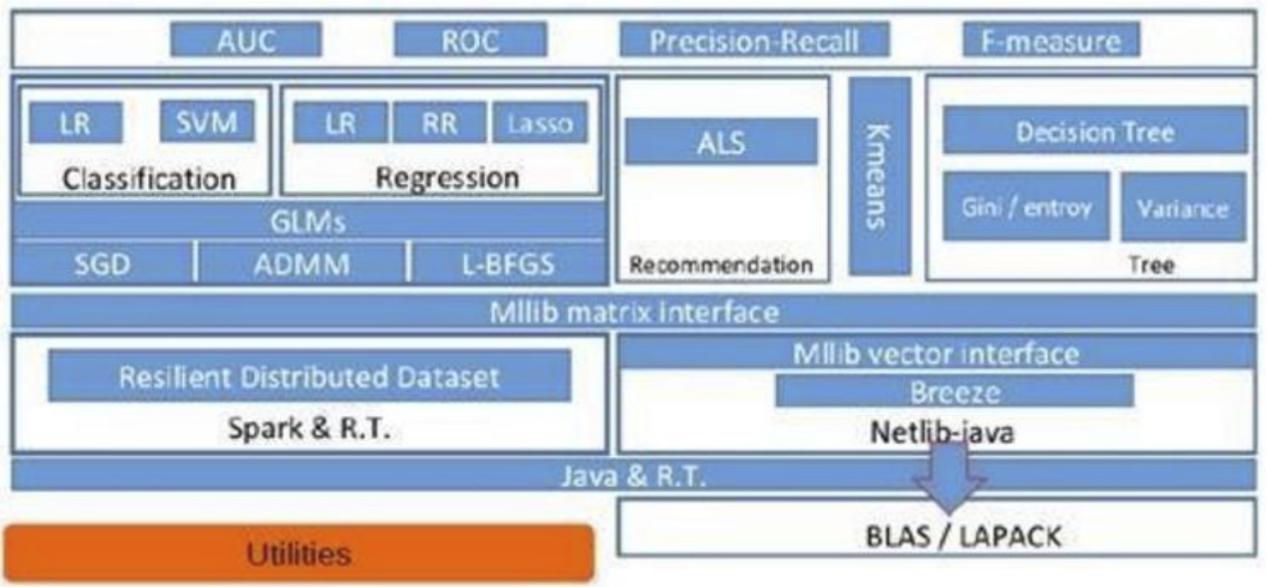
1. 底层基础：主要包括Spark的运行库、矩阵库和向量库。其中向量接口和矩阵接口基于Nelib和BLAS/LAPACK开发的线性代数库Breeze。MLlib支持本地的密集向量和本地向量，并且支持标量向量；同时支持本地矩阵和分布式矩阵，分布式矩阵分为：RowMatrix、IndexedRowMatrix和CoordinateMatrix等。
2. 算法库：包含分类、回归、聚集、协同过滤、梯度下降和特征提取和变换等算法。

**从架构图可以看出MLlib主要包含三个部分：**

底层基础：包括Spark的运行库、矩阵库和向量库；

算法库：包含广义线性模型、推荐系统、聚类、决策树和评估的算法；

实用程序：包括测试数据的生成、外部数据的读入等功能。



下图是MLlib算法库的核心内容：



MLlib由一些通用的学习算法和工具组成，包括分类、回归、聚类、协同过滤、降维等，同时还包括底层的优化原语和高层的管道API。

## 2.4基于RDD的API与基于DataFrame的API区别和应用

* 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DataFrame的API添加功能，以实现与基于RDD的API的功能奇偶校验。
* 基于Dataframe的API在达到功能（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。
* 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。
  + 两套API的介绍

## 2.5SparkMllib的环境搭建与IDEA环境配置

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-2.2.0-hadoop2.7.2.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\_2.11</artifactId>  
 <version>${spark.version}</version>  
 </dependency> //代码基于2.2.0语法代码

## 2.6RDD、DataSet、Dataframe区别及转化过程

RDD(Spark1.0)--->DataFrame(Spark1.3)---->DataSet(Spark1.6)

SparkSql提供了Dataframe和DataSet的数据抽象，DataFrame就是RDD+Schema，可以认为是一张二维表格。它的劣势是在编译器不对表格中的字段进行类型检查。在运行期间检查。DataSet是Spark最新数据抽象，Spark的发展会逐步将DataSet作为主要的数据抽象，弱化RDD和DataFrame。DataSet包含了DataFrame所有的优化机制。除此之外提供了以样例类为Schema模型的强类型。

DataFrame可以通过DataSet[Row]方式转换。DataFrame和DataSet都有可控的内存管理机制，所有的数据都保存在非堆上，都使用了catalyst进行sql优化。

首先，回顾下Spark SQL 客户端查询几种方式

* 通过Spark-shell来操作Spark SQL，spark作为SparkSession的变量名，sc作为SparkContext的变量名
* 通过Spark提供的方法读取JSON文件，将JSON文件转换成DataFrame
* 通过DataFrame提供的API来操作DataFrame里面的数据。
* 可以通过将DataFrame注册成为一个临时表的方式，来通过Spark.sql方法运行标准的SQL语句来查询。

关于 DataFrame 查询方式：DataFrame支持两种查询方式一种是**DSL**风格，另外一种是**SQL**风格

* DSL风格：

需要引入 import spark.implicit.\_ 这个隐式转换，可以将DataFrame隐式转换成RDD。

* SQL风格：

1. 你需要将DataFrame注册成一张表格，如果你通过CreateTempView这种方式来创建，那么该表格Session有效，如果你通过CreateGlobalTempView来创建，那么该表格跨Session有效，但是SQL语句访问该表格的时候需要加上前缀 global\_temp
2. 你需要通过sparkSession.sql 方法来运行你的SQL语句。

DataSet:首先定义一个DataSet，你需要先定义一个Case类。

### 2.6.1 RDD、DataSet、DataFrame之间的转换总结:

#### (1)RDD与DataFrame转换：

1、 RDD -> DataFrame : rdd.map(para=>(para(0).trim(),para(1).trim().toInt)).toDF("name","age")

//通过反射来设置

rdd.map(attributes => Person(attributes(0), attributes(1).trim.toInt)).toDF()

//通过编程方式来设置Schema，适合于编译期不能确定列的情况

schemaString.map(fieldName => StructField(fieldName, StringType, nullable = true))

val schema = StructType(fields)

val rdd[Row] = rdd.map(attributes => Row(attributes(0), attributes(1).trim))

val peopeDF = spark.createDataFrame(rdd[Row],schema)

2、 DataFrame -> RDD : dataFrame.rdd 注意输出：Array([Michael,29], [Andy,30], [Justin,19])

#### (2)RDD与DataSet转换

1、 RDD -> DataSet : rdd.map(para=> Person(para(0).trim(),para(1).trim().toInt)).toDS

2、DataSet -> RDD:dataSet.rdd 输出:Array(Person(Michael,29), Person(Andy,30), Person(Justin,19))

1、 DataFrame -> DataSet: dataFrame.to[Person]

2、 DataSet -> DataFrame： dataSet.toDF

#### (3)DataSet与DataFrame转换

对于DataFrame Row对象的访问方式

1、DataFrame = DataSet[Row]， DataFrame里面每一行都是Row对象

2、如果需要访问Row对象中的每一个元素，你可以通过下标 row(0)；你也可以通过列名 row.getAs[String]("name")

### 2.6.2各类型转化实战

各个数据的转换的练习,数据集来源于官方example,参考SparkSql官方文档：

**import** org.apache.spark.sql.SparkSession  
**case class** Person1(name: String, age: Int)**object** rdd\_df\_ds {  
 **def** main(args: Array[String]): Unit = {  
 *//1-准备环境* **val** spark: SparkSession = SparkSession.*builder*().master(**"local[\*]"**).getOrCreate()  
 **import** spark.implicits.\_  
 **val** sc = spark.sparkContext  
 *//2-加载数据* **val** peopleRDD = sc.textFile(**"D:\\BigData\\Workspace\\SparkMllibLesson\\src\\main\\scala\\sparkmllib\_part1\\sparkmllib\_helloworld\\data\\people.txt"**)  
 *//3-RDD的打印* peopleRDD.foreach(*println*(\_))  
 *// Justin, 19  
 // Michael, 29  
 // Andy, 30* **val** collectArray = peopleRDD.collect()  
 *println*(collectArray.mkString(**""**))  
 *// Michael, 29Andy, 30Justin, 19  
 //4-rdd转化df* **val** peopleDF = peopleRDD.map(\_.split(**","**)).map(x => (x(0).trim(), x(1).trim().toInt)).toDF(**"name"**, **"age"**)  
 peopleDF.show()  
 *// +-------+---+  
 // | name|age|  
 // +-------+---+  
 // |Michael| 29|  
 // | Andy| 30|  
 // | Justin| 19|  
 // +-------+---+  
 //5-df转rdd* **val** peoplerdd = peopleDF.*rdd* peoplerdd.foreach(*println*(\_))  
 *//5-rdd转ds---ds进行类型检查的需要声明样例类* **val** PeopleDS = *Seq*(*Person1*(**"Jack"**, 25)).toDS  
 PeopleDS.show()  
 *// +----+---+  
 // |name|age|  
 // +----+---+  
 // |Jack| 25|  
 //6-ds转rdd* **val** rdd2 = PeopleDS.*rdd* rdd2.foreach(*println*(\_)) *//Person1(Jack,25)  
 //7-ds-df* PeopleDS.toDF().show()  
 *//8-df--ds* peopleDF.as[Person1].show()  
 *// +-------+---+  
 // | name|age|  
 // +-------+---+  
 // |Michael| 29|  
 // | Andy| 30|  
 // | Justin| 19|* }  
}

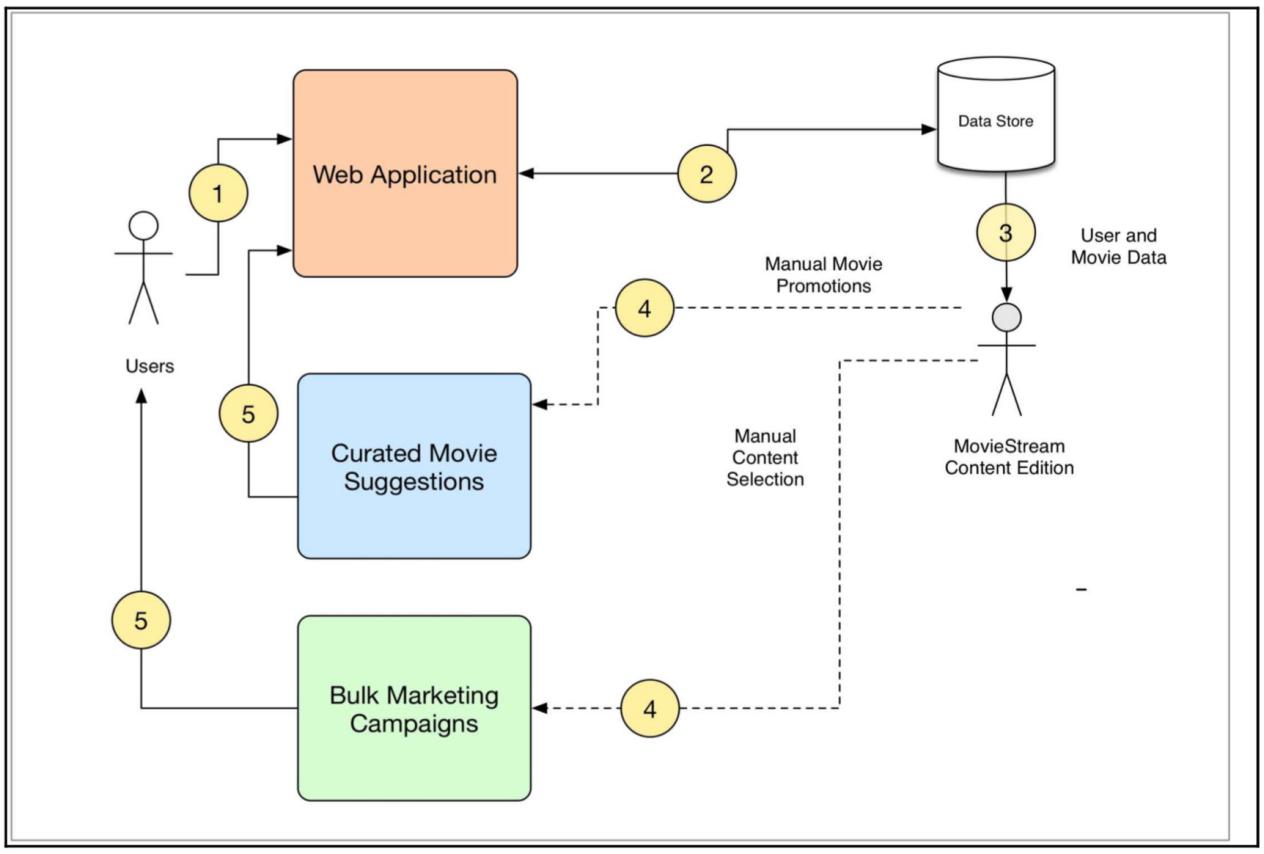
# SparkMllib的算法的分类和应用场景

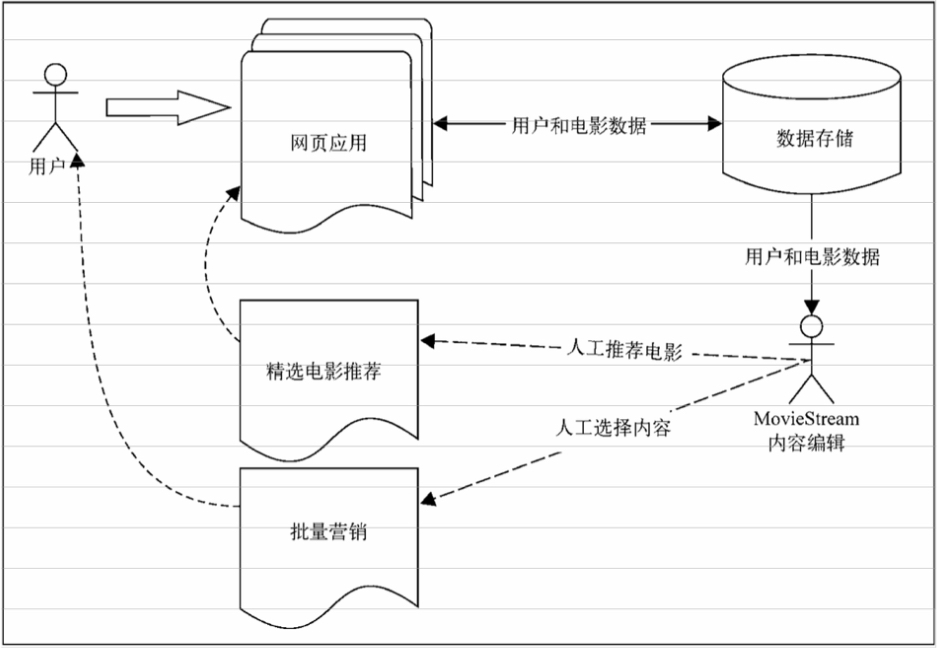
这里通过设计一套商品推荐系统来分析解析Spark中提供算法对大数据从场景下业务的支持。

## 3.1机器学习系统设计及架构

这里以推荐系统为例，假设存在一个贴近现实的情景。假设我们受命领导MovieStream数据科学团队。MovieStream是一家假想的互联网公司，为用户提供在线电影和电视节目的内容服务。

MovieStream现有系统可概括为：

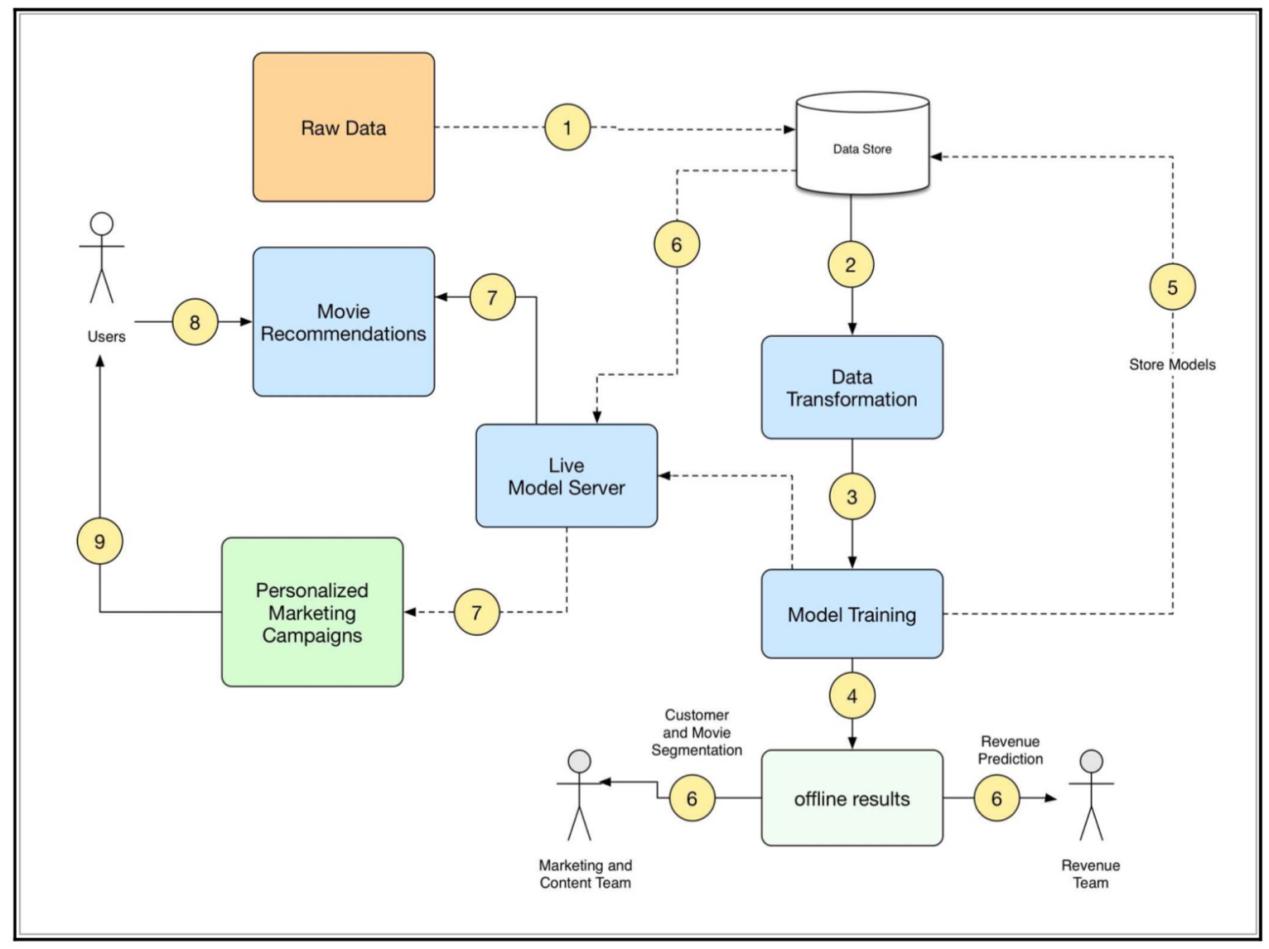


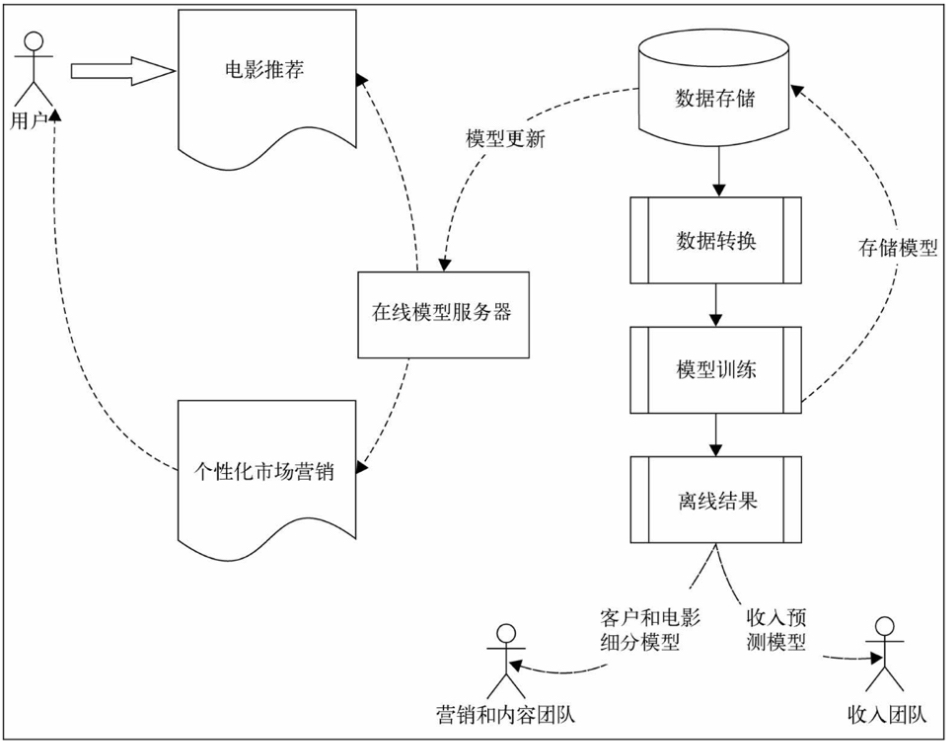


如图所示，向用户推荐哪些电影和节目以及在站点的何处显示，都由MovieStream内容编辑团队负责。该团队还负责MovieStream的群发营销，包括电子邮件和其他直销渠道。现阶段，MovieStream以汇总的方式来收集用户的电影浏览记录，并能访问一些用户注册时所填写的资料。此外，他们还能访问其所收录的电影的一些基本元数据。

随着业务快速发展，新发布的电影和用户的活动不断增加，MovieStream团队愈发难以跟上这样的趋势。MovieStream的CEO之前对大数据，机器学习和人工智能有过较多的了解。他希望我们能为MovieStream创建一个机器学习系统，以处理现在由内容团队人工处理的许多内容。

接下来提出机器学习对MovieStream重构设计：





如图所示，该系统包含了早先机器学习流程示意图的内容，此外还包括：

Ø 收集与用户，用户行为和电影标题有关的数据；

Ø 将这些数据转为特征；

Ø 模型训练，包括训练-测试和模型选择环节；

Ø 将已训练模型部署到在线服务系统，并用于离线处理；

Ø 通过推荐和目标页面将模型结果反馈到MovieStream站点；

Ø 将模型结果返回到MovieStream的个性化营销渠道；

Ø 使用离线模型来为MovieStream的各个团队提供工具，以帮助其理解用户的行为，内容目录的特点和业务收入的驱动因素。

在数据量剧增的情况下，要做到上述智能化的推荐，我们需要来进一步了解SparkMllib所支持的算法部分。

## 3.2 SparkMllib算法对机器学习建模各过程的支持

上述需求为什么要使用机器学习？为何不直接仍以人工方式来支持MovieStream？使用机器学习的理由有很多，其中最为重要的几点有：

a. 涉及的数据规模意味着完全依靠人工处理会很快跟不上MovieStream的发展；

b. 机器学习和统计模型等基于模型的方式能发现人类（因数据集量级和复杂度过高）难以发现的模式；

c. 基于模型的方式能避免个人或是情感上的偏见（只要应用时足够细心且正确）。

然而，没有任何理由说基于模型和基于人工的处理和决策不能并存。比如**，许多机器学习系统依赖已标记的数据来训练模型**。通常来说，标记数据代价高昂，耗时且需人工参与。文本数据分类和文本的情感标识便是很好的例子。许多现实中的系统会采取某种人力机制来为数据生成标识，并用于训练模型。之后，这些模型则部署到在线系统中用于大规模环境下的预测。

在MovieStream的案例中，我们并不需要担心机器学习的引入会使得内容团队多余。事实上，我们的目标是**让机器学习来负担那些耗时且机器擅长的任务，并向内容团队提供工具以帮助她们更好地理解用户和内容**。比如，帮助他们确定向电影库中新增哪些电影（新增电影代价高昂，而且对业务至关重要）。

### 3.2.1个性化推荐

对于MovieStream的业务来说，个性化或许是机器学习最为重要的潜在应用。一般来说，个性化是根据各种因素来改变用户体验和呈现给用户内容。这些因素可能包括用户的行为数据和外部因素。

推荐从根本上说是个性化的一种，常指向用户呈现一个他们可能感兴趣的物品列表。推荐可用于网页（如推荐相关产品），电子邮件，其他直销渠道或移动应用等。

个性化和推荐十分相似，但推荐通常专指向用户显式地呈现某些产品或是内容，而个性化有时也偏向隐式。比如说，对MovieStream的搜索功能个性化，以根据该用户的数据来改变搜索结果。这些数据可能包括基于推荐数据（搜索产品或内容时），或基于地理位置和搜索历史等各种数据。用户可能不会明显感觉到搜索结果的变化，这就是个性化更偏向隐性的原因。

### 3.2.2目标营销和客户细分业务

目标营销用与推荐类似的方法**从用户群中找出要营销的对象**。一般来说，推荐和个性化的应用场景都是一对一，而客户细分则试图将用户分成不同的组。其分组根据用户的特征进行，并可能参考行为数据。这种方法可能比较简单，也可能使用了某种机器学习模型，比如聚类。但无论如何，其结果都是对市场的若干细分。这些细分或许有助于理解各组用户的共性，同组用户之间的相似性，以及不同组之间的差异。

这些将能版主MovieStream理解用户行为背后的动机。相比个性化的一对一营销，它们甚至还能有助于制定针对用户群的更为广泛的营销策略。

当没有已标记数据时，这些方法能帮助制定营销策略，而非采取一刀切的方法。

### 3.2.3预测建模与分析

第三种机器学习的应用领域是**预测分析**。这个词的范围很宽泛，甚至从某种意义上说还覆盖**推荐，个性化和目标营销**。再考虑到推荐和市场细分有所区别，这里用预测建模（predictive modeling）来表示其他做预测的模型。借助活动记录，输入数据以及内容属性，MovieStream可以创建一个回归模型（regression model）来预测新电影的市场表现。

另外，我们也可以用分类模型(classification model)来对只有部分数据的新电影自动分配标签，关键字或分类。

### 3.2.4机器学习模型分类

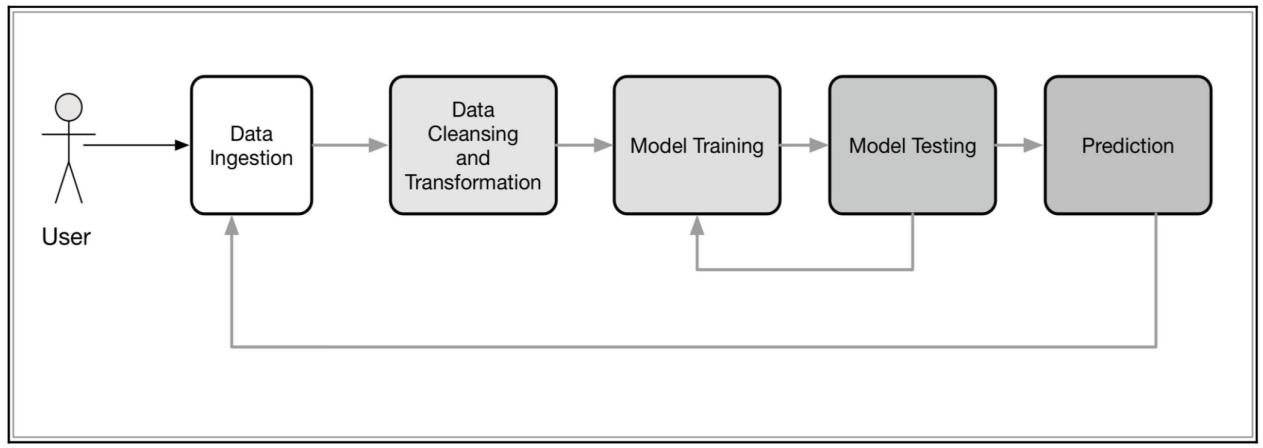
以上应用案例和方法可分为如下几种。

监督学习（supervised learning）：这种方法使用已标记数据来学习。推荐引擎，回归和分类便是例子。它们所使用的标记数据可以是用户对电影的评级（对推荐来说），电影标签（对上述分类例子来说）或是收入数字（对回归预测来说）。

无监督学习（unsupervised learning）：一些模型的学习过程不需要标记数据，我们称其为无监督学习。这类模型试图学习或是提取数据背后的结构或从中抽取最为重要的特征。聚类，降维和文本处理的某些特征提取都是无监督学习。

### 3.2.5. 数据驱动的机器学习系统的组成

从高层设计来看，我们的机器学习系统的组成如下图，其中展示了机器学习的流程。该流程始于从数据存储处获取数据，之后将其转换为可用于机器学习模型的形式。随后的环节有对模型的训练，测试和完善，以及将最终的模型部署到盛昌系统中。有新数据产生时则重复该流程。



#### (1)数据获取与存储

机器学习流程的第一步是获取训练模型所需的数据。与其他公司类似，MovieStream的数据通常来自用户活动，其他系统（通常称作机器生成的数据）和外部数据源（比如某个用户访问站点的时间和当时的天气）。

获取这些数据的途径很多，比如收集浏览器里用户的活动记录，移动应用的事件日志或通过外部网络API来获取地理或天气信息。

获取数据后通常将其存储起来。要存储的数据包括：原始数据，即时处理后的数据，以及可用于生产系统的最终建模结果。

数据存储并不简单，可能涉及多种系统。文件系统，如HDFS，Amazon S3等；SQL数据库，如MySQL或PostgreSQL；分布式NoSQL数据存储，如HBase ,Cassandra和DynamoDB; 搜索引擎Solr和Elasticsearch; 流数据系统，如Kafka,Flume和Amazon kinesis.

#### (2)数据清理与转换

大部分机器学习模型所处理的都是特征（feature）。特征通常是输入变量所对应的可用于模型的数值表示。

虽然我们希望将大部分时间用于机器学习模型探索，但通常经上述途径获取到的数据都是原始形式，需要进一步处理。比如我们记录的一些用户事件的细节，比如用户查看电影页面的时间，观看某部电影的时间或给出某些反馈的时间。我们还可能收集了一些外部信息，比如用户的位置（通过他们的IP查到）。这些时间日志通常由一些文字或数值信息组合而成。

绝大部分情况下，这些原始数据都需要经过预处理才能为模型所使用。预处理的情况包括以下几种。

1. **数据过滤**：比如我们想从原始数据的部分数据中创建一个模型，而所需数据只是最近几月的活动数据或是满足特定条件的事件数据。

2. **处理数据缺失**，不完整或有缺陷：许多现实中的数据集都存在某种程度上的不完整。这可能包括数据缺失（比如用户没有输入），数据存在错误或是缺陷（比如数据收集或存储时的错误，又或是技术问题或漏洞，以及软硬件故障）。可能要过滤掉非规整数据，或通过某种方式来填充缺失的数据点（比如选取数据集的平均值来作为缺失点的值）。

3. **处理可能的异常**，错误和异常值：错误或异常的数据可能不利于模型的训练，所以需要过滤掉，或是通过某些方法来处理。

4. **合并多个数据源**：比如可能要将各个用户的事件数据与不同的内部数据或是外部数据合并。内部数据如用户属性；外部数据如地理位置，天气和经济数据。

5. **数据汇总**：某些模型需要输入的数据进行过某种汇总，比如统计各用户经历过的事件类型的总数目。

对数据进行初步预处理后，需要将其转换为一种适合机器学习模型的表示形式。对许多模型类型来说，这种表示就是包含数据的向量或矩阵。数据转换和特征提取时常见的挑战，包括以下这种情况。

¬ 将类别数据（比如地理位置所在的国家或是电影的类别）编码为对应的数值表示。

¬ 从文本数据提取有用信息。

¬ 处理图像或是音频数据。

¬ 数值数据常被转换为类别数据以减少某个变量的可能值的数目。例如将年龄分为几个段（比如25～35，45～55等）

¬ 对数值特征进行转换。比如对数值变量应用对数转换，这会有助于处理值域很大的变量。

* 对数值特征进行转换。比如对数值变量应用对数转换，这会有助于处理值域很大的变量。
* 对特征进行正则化，标准化，以保证同一模型的不同输入变量的值域相同。
* **特征工程是对现有变量进行组合或转换以生成新特征的过程**。例如从其他数据求平均数，像求某个用户看电影的平均时间。

这些数据清理，探索，聚合和转换步骤，都能通过Spark核心API，SparkSQL引擎和其他外部Scala, Java或Python包做到。借助Spark的hadoop功能还能实现上述多种存储系统上的读写。

### 3.2.6 模型训练与测试回路

当数据已换为可用于模型的形式，便可开始模型的训练和测试。在这个部分，我们主要关注**模型选择（model selection）问题**。这可以归结为对特定任务最优建模方法的选择，或是对特定模型最佳参数的选择问题。在许多情况下，我们会想尝试许多模型并选出表现最好的那个（各模型都采用了最佳的参数时）。因而，这个词在现实中经常同时指代这两个过程。在这个阶段，探索多个模型组合（esemble method）的效果也很常见。

在训练数据集上运行模型并在测试数据集（即为评估模型而预留的数据，在训练阶段模型没有接触过该数据）上测试其效果，这个过程一般相对直接，被称作交叉验证（cross-validation）。

然而我们所处理的通常是大型数据集。这样，先在具有代表性的小样本数据集上进行初步的训练-测试回路，或是尽可能并行地选择模型，都会有所帮助。Spark内置的机器学习库MLib完全能胜任这个阶段的需求。

### 3.2.7 模型部署与整合

通过训练测试循环找出最佳模型后，要让它能得出可付诸实践的预测，还需将其部署到生产系统中。这个过程一般要将已训练的模型导入特定的数据存储中。该位置也是生产系统获取新版本的地方。通过这种方式，实时服务系统能在训练模型时进行周期性的更新。

### 3.2.8 模型监控与反馈

监控机器学习系统在生产环境下的表现十分重要。在部署了最优训练的模型后，我们会想知道其在实际中的表现如何：它在新的未知数据集上的表现是否符合预期？其准确度怎么样？毕竟不管之前的模型选择和优化做得如何，检验其实际表现的唯一方法是观察其在生产环境下的表现。

同样值得注意的是，模型准确度和预测效果只是现实中系统表现的一部分。通常还应该关注其他业务效果（比如收入和利润率）或用户体验（比如站点使用时间和用户总体活跃度）的相关指标。多数情况下很难将它们与模型预测能力直接关联。推荐系统或目标营销系统的准确度可能很重要，但它只与我们真正关心的那些指标（如用户体验度，活跃度以及最终收入）间接相关。

所以，现实中应该同时监控模型准确度相关指标和业务指标。我们可以尽可能在生产系统中部署不同的模型，通过调整它们而优化业务指标。实践中，这通常通过在线分割测试（live split test）进行。然而，做好这类测试并不容易。在线测试和实验可能引发错误，也可能效果不好，或者会使用基准模型，这些都会给用户体验和收入带来负面影响，故其代价高昂。

本阶段另一个重要的方面是**模型反馈（model feedback）**，指通过用户的行为来对模型的预测进行反馈的过程。在现实系统中，模型的应用将影响用户的决策和潜在行为，从而反过来将从根本上改变模型自己将来的训练数据。

举例来说，假设我们部署了一个推荐系统。由于推荐实际上限制了用户的可选项，从而影响了用户的选择。我们希望用户的选择不会受模型的影响，然而这种反馈回路会反过来影响模型的训练数据，并最终对模型准确度和重要的业务指标产生不利影响。

好在我们可以借助一些机制来降低反馈回路的这种负面影响，比如提供了一些无偏见的训练数据。这类数据来自那些没有被推荐的用户，又或者在一开始就考虑到这种平衡需求而划分出来的客户。这些机制有助于对数据的理解，探索以及利用已有的经验来提升系统的表现。

### 3.2.9 批处理或实时方案的选择

模型用所有的数据或一部分数据进行周期性的重新训练。由于上述流程会花费一定的时间，这就使得批处理方法难以在新数据到达时立即完成模型的更新。

本书讨论的是批处理机器学习方法，但的确存在一类名为**在线学习（online learning）**的方法。它们在新数据到达时便能立即更新模型，从而使实时系统成为可能。常见的例子有对线性模型的在线优化算法，如**SGD**。我们可以通过例子来学习该算法。这类方法的优势在于其系统将能对新的信息和底层行为（即输入数据的特征或是分布会随时间变化，现实中的绝大部分情况都会如此）作出快速的反应和调整。

但在实际生产环境中，在线学习模型也会面对特有的挑战。比如，对数据的获取和转换难以做到实时。在一个纯在线环境下选择适当的模型也不简单。在线训练和模型选择以及部署阶段的延时可能难以达到实时性的需求（比如在线广告对延时的需求是以毫秒计）。最后，批处理框架不适合对本质为流的数据进行实时处理。

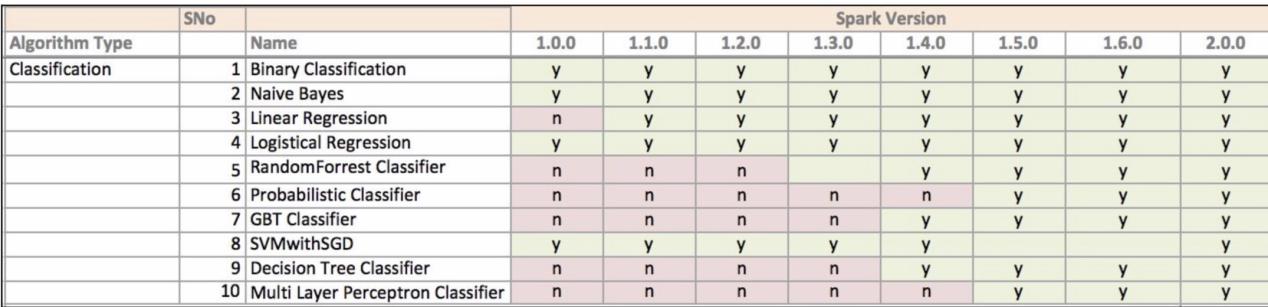
现实中的实时机器学习系统具有天生的复杂性，故实践中大部分的系统都以近实时性为设计目标。这是一种混合方法，它并不要求模型一定在数据到达时立即更新。相反，新的数据会被收集为小批量的训练数据，再输入给在线学习算法。大部分情况下，该方法会周期性地进行某种批处理。处理的内容可能包括在整个数据集上重新计算模型，或是更为复杂的某些数据处理以及模型的选择。这些能保证实时模型的表现不会随时间推移而变差。

另一种类似的方法是，在周期行批处理中进行重新计算时，若有新的数据到来则只对更复杂的模型进行近似更新。这样模型可从新的数据学习，但有短暂延迟。因为近似更新，所以模型的准确度会随着时间推移而下降。但周期性地在所有数据上重新计算模型能弥补这一点。

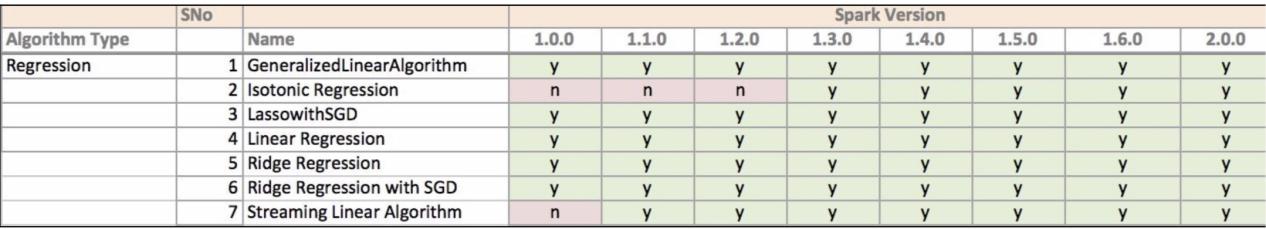
### 3.2.10 Spark支持算法比较

通过上述机器学习解决实际业务问题的重述，SparkMllib为了解决在大数据场景下建立机器学习模型解决实际业务问题，提供了从基础的数据统计、数据类型、管道、特征抽取、加载、转换等，分类、聚类、回归算法、推荐领域提供了协同过滤和关联规则算法，模型选择部分提供了模型优化和超参数优化场景算法。

## 3.3SparkMllib分类算法比较及应用场景详解



## 3.4SparkMllib回归算法比较及应用场景详解



## 3.5SparkMllib聚类算法比较及应用场景详解



## 3.6SparkMllib关联算法比较及应用场景详解

Spark支持关联挖掘场景下的FP-Growth算法，达到关联挖掘业务推荐。

## 3.7SparkMllib推荐算法比较及应用场景详解

SparkMllib支持推荐算法领域的基于模型的协同过滤算法推荐，仅实现了ALS交替最小二乘法的推荐。

# SparkMllib的基础数据类型

## 4.1SparkMllIb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。

## 4.2Spark的LocalVector本地向量详解及实战

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))  
 }  
}

## 4.3Spark的LabelPoint标签向量详解及实战

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")

## 4.4Spark的LocalMatrix本地矩阵详解及实战

\* 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))  
 }  
}

## 4.5Spark的DistributedMatrix分布式矩阵详解及实战

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。

### 4.5.1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  
 }  
}

### 4.5.2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  
 }  
}

### 4.5.3三元组矩阵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  
 }  
}

### 4.5.4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  
 }  
}

# 基于SparkMllib的统计特征实践

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

## 5.1SparkMllib的SummaryStatistic摘要统计

### 5.1.1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]

### 5.1.2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  
 }  
}

## 5.2SparkMllib的Correlation相关系数详解及实战

基于rdd的相关系数实践：

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 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)  
  }

}

基于Spark ml的机器学习库的相关系数实践：**(Dataframe)**

import org.apache.spark.ml.linalg.{Matrix, Vectors}

import org.apache.spark.ml.stat.Correlation

import org.apache.spark.sql.Row

val data = Seq(

Vectors.sparse(4, Seq((0, 1.0), (3, -2.0))),

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

Vectors.dense(6.0, 7.0, 0.0, 8.0),

Vectors.sparse(4, Seq((0, 9.0), (3, 1.0))))

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

val Row(coeff1: Matrix) = Correlation.corr(df, "features").head

println("Pearson correlation matrix:\n" + coeff1.toString)

val Row(coeff2: Matrix) = Correlation.corr(df, "features", "spearman").head

println("Spearman correlation matrix:\n" + coeff2.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  
 }  
}

## 5.3SparkMllib的HypothesisTesting原理及实战

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。

### 5.3.1卡方检验：

基于rdd卡方验证：

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。

**基于SparkMl的机器学习库实践：(Dataframe)**

import org.apache.spark.ml.linalg.{Vector, Vectors}

import org.apache.spark.ml.stat.ChiSquareTest

val data = Seq(

(0.0, Vectors.dense(0.5, 10.0)),

(0.0, Vectors.dense(1.5, 20.0)),

(1.0, Vectors.dense(1.5, 30.0)),

(0.0, Vectors.dense(3.5, 30.0)),

(0.0, Vectors.dense(3.5, 40.0)),

(1.0, Vectors.dense(3.5, 40.0))

)

val df = data.toDF("label", "features")

val chi = ChiSquareTest.test(df, "features", "label").head

println("pValues = " + chi.getAs[Vector](0))

println("degreesOfFreedom = " + chi.getSeq[Int](1).mkString("[", ",", "]"))

println("statistics = " + chi.getAs[Vector](2))

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..

### 5.3.2Kolmogorov-Smirnov（KS）测试

KS-检验（Kolmogorov-Smirnov test） -- 检验数据是否符合某种分布

Kolmogorov-Smirnov是比较一个频率分布f(x)与理论分布g(x)或者两个观测值分布的检验方法。其原假设H0:两个数据分布一致或者数据符合理论分布。D=max| f(x)- g(x)|，当实际观测值D>D(n,α)则拒绝H0，否则则接受H0假设。

KS检验与t-检验之类的其他方法不同是KS检验不需要知道数据的分布情况，可以算是一种非参数检验方法。当然这样方便的代价就是当检验的数据分布符合特定的分布时，KS检验的灵敏度没有相应的检验来的高。在样本量比较小的时候，KS检验最为非参数检验在分析两组数据之间分布是否不同时相当常用。

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.

### 5.3.3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)  
 }  
}

## 5.4SparkMllib的随机数生成时间

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  
 }}

# 基于SparkMllib数据特征工程构建过程

## 6.1SparkMllib特征工程构建过程

* **提取：**从“原始”数据中提取特征
* **转换：**缩放，转换或修改特征
* **选择：**从更大的特征集中选择特征子集

## 6.2SparkMllib特征提取操作实践

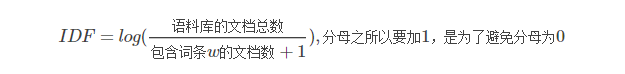
### 6.2.1 TF-IDF

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：



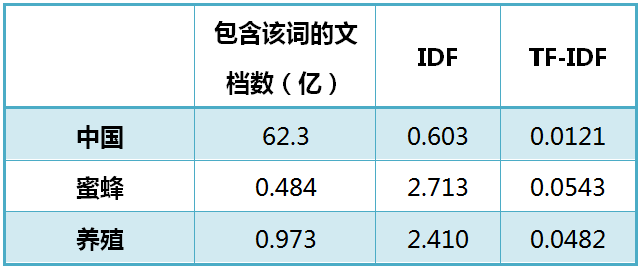
其中，|D|表示是语料库中的文档总数。由于采用了对数，如果一个词出现在所有的文件，其IDF值变为0。



某一特定文件内的高词语频率，以及该词语在整个文件集合中的低文件频率，可以产生出高权重的TF-IDF。因此，TF-IDF倾向于过滤掉常见的词语，保留重要的词语。

可以看到，TF-IDF与一个词在文档中的出现次数成正比，与该词在整个语言中的出现次数成反比。所以，自动提取关键词的算法就很清楚了，就是计算出文档的每个词的TF-IDF值，然后按降序排列，取排在最前面的几个词。

接下来看一个例子，假定该文长度为1000个词，"中国"、"蜜蜂"、"养殖"各出现20次，则这三个词的"词频"（TF）都为0.02。然后，搜索Google发现，包含"的"字的网页共有250亿张，假定这就是中文网页总数。包含"中国"的网页共有62.3亿张，包含"蜜蜂"的网页为0.484亿张，包含"养殖"的网页为0.973亿张。则它们的逆文档频率（IDF）和TF-IDF如下：



从上表可见，"蜜蜂"的TF-IDF值最高，"养殖"其次，"中国"最低。（如果还计算"的"字的TF-IDF，那将是一个极其接近0的值。）所以，如果只选择一个词，"蜜蜂"就是这篇文章的关键词。

除了自动提取关键词，TF-IDF算法还可以用于许多别的地方。比如，信息检索时，对于每个文档，都可以分别计算一组搜索词（"中国"、"蜜蜂"、"养殖"）的TF-IDF，将它们相加，就可以得到整个文档的TF-IDF。这个值最高的文档就是与搜索词最相关的文档。

接下来我们就通过SparkMllib实现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)

}

}

### 6.2.2Word2Vec

word2vec，字面意思，将word转化为vector，word是顺序有意义的实体，比如文档中单词、用户依次点击的商品。

word2vec是NLP领域的重要算法，它的功能是将word用K维的dense vector来表达，训练集是语料库，不含标点，以空格断句。因此可以看作是种特征处理方法。

word2vec两种实现方式：

Skip-gram：用一个词语作为输入，来预测它周围的上下文。同义词p(word1|word2)

CBOW ：用一个词语的上下文作为输入，来预测这个词语本身。完形填空p(word1|word2,word3)

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]]

### 6.2.3CountVectorizer

CountVectorizer并CountVectorizerModel旨在帮助将一组文本文档转换为标签计数的向量。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")

在文本中类型是 Array[String].

调用CountVectorizer的拟合会生成带有词汇表（a，b，c）的CountVectorizerModel。 然后转换后的输出列“vector”包含：

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...|

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

## 6.3SparkMllib特征转换操作详解及实践

### 6.3.1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)

}

}

### 6.3.2StopWordsRemover

[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）。

案例

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]|

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

### 6.3.3二值化

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]

### 6.3.4PCA

[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...|

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

### 6.3.5StringIndexer

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|

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

### 6.3.6IndexToString

对称的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

代码：

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|

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

### 6.3.7OneHotEncoder

[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])|

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

### 6.3.8VectorIndexer

**VectorIndexer可以自动将决定哪些features是类别的，并且可以将原始值转换成类别指标。而在决策树和提升树算法中，能够确定类别特征指标，VectorIndexer可以提升这些算法的性能。**

假设所有特征都已经被组织在一个向量中，又想对其中某些单个分量进行处理时，Spark ML提供了VectorIndexer类来解决向量数据集中的类别性特征转换。

通过为其提供maxCategories超参数，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。它基于不同特征值的数量来识别哪些特征需要被类别化，那些取值可能性最多不超过maxCategories的特征需要会被认为是类别型的。

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

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

import org.apache.spark.sql.SparkSession

object 特征变换\_VectorIndexer {

val spark=SparkSession.builder().master("local[2]").appName("IndexToString").getOrCreate()

import spark.implicits.\_

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

val data=Seq(

Vectors.dense(-1,1,1,8,56),

Vectors.dense(-1,3,-1,-9,88),

Vectors.dense(0,5,1,10,96),

Vectors.dense(0,5,1,11,589),

Vectors.dense(0,5,1,11,688)

)

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

val indexer= new VectorIndexer().setInputCol("features").setOutputCol("indexed").setMaxCategories(4) //那些取值可能性最多不超过maxCategories的特征会被认为是类别型的,进而将原始值转换为类别索引

val indexer\_model=indexer.fit(df)

val categoricalFeatures= indexer\_model.categoryMaps.keys.toSet

println(s"Chose ${categoricalFeatures.size} categorical features: " + categoricalFeatures.mkString(", "))

val indexed=indexer\_model.transform(df)

indexed.show(false)

}

}

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。
5. 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。

下面通过libsvm数据集的案例实践：

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

### 6.3.9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)//L1正则--2.0为L2正则

val l1NormData = normalizer.transform(dataFrame)

l1NormData.show()

// 结果分析：  
    // 如：向量(5,[1,2,4],[145.0,253.0,211.0])  
    // p-Norm=145.0+253.0+211.0=609.0   p=1  
    // L1正则化：(5,[1,2,4],[145.0/609.0,253.0/609.0,211.0/609.0])  
    // 正则化结果：(5,[1,2,4],[0.23809523809523808,0.4154351395730706,0.3464696223316913])]

 // L2正则化  
     val l2InfNormData = normalizer.transform(dataFrame, normalizer.p -> 2)  
    l2InfNormData.foreach(println)

 // 结果：  
    // [0.0,(5,[1,2,4],[51.0,253.0,253.0]),(5,[1,2,4],[0.1411129915384288,0.7000311148867153,0.7000311148867153])]  
    // [1.0,(5,[1,2,4],[145.0,253.0,211.0]),(5,[1,2,4],[0.40284772269065683,0.702899819591284,0.5862128930188178])]  
    // [1.0,(5,[1,2,3],[124.0,253.0,255.0]),(5,[1,2,3],[0.3263044910373298,0.6657664212293906,0.6710293968912829])]  
    // 结果分析：  
    // 如：向量(5,[1,2,4],[145.0,253.0,211.0])  
    // p-Norm=math.sqrt(145\*145+253\*253+211\*211)=359.9374945737107 p=2  
    // L2正则化：(5,[1,2,4],[145.0/p-Norm,253.0/p-Norm,211.0/p-Norm])  
    // 正则化结果：(5,[1,2,4],[0.40284772269065683,0.702899819591284,0.5862128930188178])]

// 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

### 6.3.10StandardScaler

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

### 6.3.11MinMaxScaler

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

### 6.3.12MaxAbsScaler

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

### 6.3.13Bucketizer

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。

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|

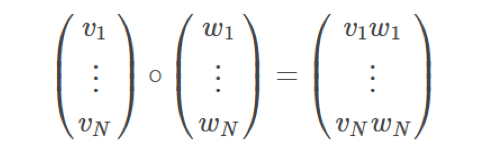
| 0.0| 2.0|

| 0.2| 2.0|

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

### 6.3.14ElementwiseProduct

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]|

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

### 6.3.15SQLTransformer

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|

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

### 6.3.16VectorAssembler

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]

代码：

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]

### 6.3.17QuantileDiscretizer

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

代码：

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|

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

## 6.4SparkMllib特征选择操作及实践

### 6.4.1VectorSlicer

VectorSlicer用于从原来的特征向量中切割一部分，形成新的特征向量，比如，原来的特征向量长度为10，我们希望切割其中的5~10作为新的特征向量，使用VectorSlicer可以快速实现。

SparkMllib中的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"]

代码：

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")

//构造特征数组

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

//为特征数组设置属性名（字段名），分别为f1 f2 f3

val defaultAttr = NumericAttribute.defaultAttr

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

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

//构造DataFrame

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

//构造切割器

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

//根据索引号，截取原始特征向量的第1列和第3列

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

//根据字段名，截取原始特征向量的f2和f3

//索引号和字段名也可以组合使用，截取原始特征向量的第1列和f2

// 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]]

总结：

VectorSlicer是一个转换器，它对于输入的特征向量，输出一个新的原始特征子集的特征向量。对于从列向量中提取特征很有帮助。  
 VectorSlicer对于指定索引的列向量，输出一个新的列向量，所选择的列向量通过这些索引进行选择。有两种类型的索引：

整数索引：代表列向量的下标，setIndices()

字符串索引：代表列的特征名称，setNames()。这要求列向量有AttributeGroup，因为实现中是在Attribute上的name字段匹配。

整数和字符串的索引都可以接受。此外，还可以同时使用整数索引和字符串名称索引。但必须至少选择一个特征。重复的特征选择是不允许的，所以选择的索引和名称之间不能有重叠。请注意，如果选择了特征的名称索引，则遇到空的输入属性时会抛出异常。  
 输出时将按照选择中给出的特征索引的先后顺序进行向量及其名称的输出。

### 6.4.2RFormula

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|

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

### 6.4.3ChiSqSelector

(1)**卡方检验来选择特征的原因**

对于建立模型而言并非特征越多越好，因为建模的**目标**是使用尽量简单的模型去实现尽量好的效果。减少一些**价值小贡献小**的特征有利于在表现效果不变或降低的前提下，新找到最简单的模型。

那么什么样的特征是价值小的呢？想想我们之所以用机器学习的模型去学习特征，**是为了更好地预测被特征影响的因变量（标签）**。那么那些根本不会对应变量产生影响，或者影响很小的特征理应事先去掉。

那么怎么判断特征对应变量的影响程度的大小呢？我们可以使用**卡方检验**对特征与应变量进行**独立性检验**，**如果独立性高，那么表示两者没太大关系，特征可以舍弃**；**如果独立性小，两者相关性高，则说明该特征会对应变量产生比较大的影响，应当选择。**

（2）卡方独立性检验的过程

卡方检验适用于类别变量，如果要对连续型变量做检验，可以将连续型变量分成多个区间，变成类别的形式。

如上，卡方检验可以判断两个变量之间是否有显著的相关性（也可以说成是否有显著的独立性）。

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。

Spark中的ChiSqSelector 就是使用了卡方独立性检验来决定哪些特征优秀应该被选择。它提供了3中方法：

（1）numTopFeatures，设置固定的提取特征的数量，程序会根据卡方值的高低返回前n个卡方值最高的特征。（预测能力最强的前n个特征）

（2）percentile，与上类似，设置的是一个比例。

（3）fpr，预先设定一个显著性水平α，所有p值低于α的特征将会被选择出来。

**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")

import spark.implicits.\_

val data = Seq(

(7, Vectors.dense(0.0, 0.0, 18.0, 1.0), 1.0),

(8, Vectors.dense(0.0, 1.0, 12.0, 0.0), 0.0),

(9, Vectors.dense(1.0, 0.0, 15.0, 0.1), 0.0)

)

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

val selector = new ChiSqSelector()

.setNumTopFeatures(1)

.setFeaturesCol("features")

.setLabelCol("clicked")

.setOutputCol("selectedFeatures")

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

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

result.show()

}

}

下面通过一个案例理解如何选择重要特征：

首先查看数据源格式

数据说明：

gender: 0–女 1–男

hobby: 1-篮球 2-足球 3-乒乓 4-逛街 5-购物

age：年龄

salary：工资

makeup：是否化妆 1–化妆 0–不化妆

说明：数据源一定要是数字类型的，很多算法都要求是double的，所以你在使用spark做处理的时候，要把数据源归类处理。



代码：

package sparkmllib\_part1.sparkmllib\_featurazation  
  
import org.apache.spark.ml.feature.ChiSqSelector  
import org.apache.spark.ml.linalg.Vectors  
import org.apache.spark.sql.SparkSession  
  
object chisquareTest {  
 def main(args: Array[String]): Unit = {  
 val spark: SparkSession = SparkSession.builder()  
 .appName("SparkMlilb")  
 .master("local[2]")  
 .getOrCreate()  
 spark.sparkContext.setLogLevel("WARN")  
 import spark.implicits.\_  
 val data = Seq(  
 //gender-hobby-age-salary- makeup：是否化妆 1–化妆 0–不化妆  
 (1, Vectors.dense(1.0, 2, 22.0, 22222), 1.0),  
 (2, Vectors.dense(1.0, 2, 22.0, 22222), 1.0),  
 (3, Vectors.dense(1.0, 3, 22.0, 2323), 1.0),  
 (4, Vectors.dense(1.0, 3, 22.0, 32323), 1.0),  
 (5, Vectors.dense(1.0, 4,22.0, 4444), 0.0),  
 (6, Vectors.dense(0.0, 9, 22.0, 345345), 0.0),  
 (7, Vectors.dense(0.0, 9, 22.0, 435345), 0.0),  
 (8, Vectors.dense(0.0, 5, 22.0, 234234), 0.0),  
 (9, Vectors.dense(1.0, 5, 22.0, 4564), 1.0)  
 )  
 val df = spark.createDataset(data).toDF("id", "features", "clicked")  
  
 val selector = new ChiSqSelector()  
 .setNumTopFeatures(1)  
 .setFeaturesCol("features")  
 .setLabelCol("clicked")  
 .setOutputCol("selectedFeatures")  
  
 val result = selector.fit(df).transform(df)  
 println(s"ChiSqSelector output with top ${selector.getNumTopFeatures} features selected")  
 result.show()  
 }  
}

根据计算结果说明,选择了features中的第一列(gender), 那么就说明, makeup这列的值, 是与gender这列的值有很大关系的, 那么经过卡方特征选择后得到的特征与我们所理解的与化妆最相关的特征是哪个特征猜测的一致。 如下结果输出。

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

| id| features|clicked|selectedFeatures|

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

| 1|[1.0,2.0,22.0,222...| 1.0| [1.0]|

| 2|[1.0,2.0,22.0,222...| 1.0| [1.0]|

| 3|[1.0,3.0,22.0,232...| 1.0| [1.0]|

| 4|[1.0,3.0,22.0,323...| 1.0| [1.0]|

| 5|[1.0,4.0,22.0,444...| 0.0| [1.0]|

| 6|[0.0,9.0,22.0,345...| 0.0| [0.0]|

| 7|[0.0,9.0,22.0,435...| 0.0| [0.0]|

| 8|[0.0,5.0,22.0,234...| 0.0| [0.0]|

| 9|[1.0,5.0,22.0,456...| 1.0| [1.0]|

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

## 6.5项目案例-业务数据统计分析实践

### 6.5.1需求分析

案例对loc-gowalla中获取的地理位置信息进行数据分析，使用上述学习的特征进行分析，得到该数据的均值、方差、皮尔逊相关系数等特性分析。

### 6.5.2数据准备

0 2010-10-19T23:55:27Z 30.2359091167 -97.7951395833 22847

0 2010-10-18T22:17:43Z 30.2691029532 -97.7493953705 420315

0 2010-10-17T23:42:03Z 30.2557309927 -97.7633857727 316637

0 2010-10-17T19:26:05Z 30.2634181234 -97.7575966669 16516

0 2010-10-16T18:50:42Z 30.2742918584 -97.7405226231 5535878

0 2010-10-12T23:58:03Z 30.261599404 -97.7585805953 15372

0 2010-10-12T22:02:11Z 30.2679095833 -97.7493124167 21714

0 2010-10-12T19:44:40Z 30.2691029532 -97.7493953705 420315

0 2010-10-12T15:57:20Z 30.2811204101 -97.7452111244 153505

0 2010-10-12T15:19:03Z 30.2691029532 -97.7493953705 420315

0 2010-10-12T00:21:28Z 40.6438845363 -73.7828063965 23261

数据格式为：

**用户user-签到时间check-in time-维度latitude-经度longitude-位置标识locationid**

注意时间转换包的maven仓库地址：

<!-- https://mvnrepository.com/artifact/joda-time/joda-time -->  
<dependency>  
 <groupId>joda-time</groupId>  
 <artifactId>joda-time</artifactId>  
 <version>2.9.9</version>  
</dependency>

#### (1)数据清洗

在数据清洗阶段过滤掉不符合条件的数据，并将数据进行格式化转换，保证数据的完整性、唯一性、合法性、一致性，并按照CheckIn类填充数据，具体实现方法如下：

|  |
| --- |
| // 定义数据类 case class CheckIn(user: String, time: String, latitude: Double, longitude: Double, location: String)  //数据清洗  val gowalla = sc.textFile(input).map(\_.split("\t")).mapPartitions {    case iter =>      val format = DateTimeFormat.forPattern("yyyy-MM-dd\'T\'HH:mm:ss\'Z\'")      iter.map {        //DateTime.parse(terms(1),format)        case terms => CheckIn(terms(0), terms(1).substring(0, 10), terms(2).toDouble, terms(3).toDouble, terms(4))      }  }  gowalla.collect().foreach(println) |

#### (2)数据转换

在数据转换阶段，将数据转换向量的形式，供后面数据分析使用。

|  |
| --- |
| //数据转换  // user, checkins, check in days, locations  val data = gowalla.map {    case check: CheckIn => (check.user, (1L, Set(check.time), Set(check.location))) // Set(集合)是没有重复的对象集合，所有的元素都是唯一的。  }.reduceByKey {    // 并集 union    case (left, right) => (left.\_1 + right.\_1, left.\_2.union(right.\_2), left.\_3.union(right.\_3))  }.map {    case (user, (checkins, days: Set[String], locations: Set[String])) =>      Vectors.dense(checkins.toDouble, days.size.toDouble, locations.size.toDouble)  }  //数据打印  data.collect().foreach(println)  //保存  data.coalesce(1).saveAsTextFile(output) |

### 6.5.3数据分析

通过简单的数据分析流程，实现对均值、方差、非0元素的目录统计，以及皮尔逊相关系数，来实现对数据分析。

|  |
| --- |
| //统计方法  val summary: MultivariateStatisticalSummary = Statistics.colStats(data)  //均值=【60，25，37】  println("Mean" + summary.mean)  //方差=【18547，1198，7350】  println("Variance" + summary.variance)  //非零元素的目录【107092，107092，107092】  println("NumNonzeros" + summary.numNonzeros)  //皮尔逊【1.0，0.7，0.9】  val correlMatrix: Matrix = Statistics.corr(data, "pearson")  println("correlMatrix" + correlMatrix.toString)  sc.stop() |

## 6.6项目案例-Iris鸢尾花数据分析案例实践

前面已经对IRIS鸢尾花数据集有过清晰的描述分析，接下来通过SparkSql结合SparkMllib完成IRIS鸢尾花数据的数据分析和特征构建，为建模分析打下基础。

### 6.6.0Iris读取方式

#### （1）样例类方式

package sparkMllib\_classfication\_day03.IrisDemo

import org.apache.spark.rdd.RDD

import org.apache.spark.{SparkConf, SparkContext}

import org.apache.spark.sql.{DataFrame, SparkSession}

/\*\*

\* DESC: 目标：通过SparkCore实现Iris数据的读取转化为DF

\* Complete data processing and modeling process steps:

\* 1-准备环境

\* 2-准备数据---读取iris的数据

\* 3-将数据进行转化----通过Row()对象转换，case样例类的方式，toDF方式

\* 4-查看schema信息转换

\* sepal\_length,sepal\_width,petal\_length,petal\_width,class

\*/

case class IrisData(sepal\_length: Double, sepal\_width: Double,

petal\_length: Double, petal\_width: Double, classlabel: String)

object IrisSparkCoreLoader {

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

// \* 1-准备环境

val conf: SparkConf = new SparkConf().setAppName("IrisSparkCoreLoader").setMaster("local[\*]")

val spark: SparkSession = SparkSession.builder().config(conf).getOrCreate()

val sc: SparkContext = spark.sparkContext

sc.setLogLevel("WARN")

import spark.implicits.\_

// \* 2-准备数据---读取iris的数据

val datapath = "D:\\BigData\\Workspace\\SparkMachineLearningTest\\SparkMllib\_BigDataSH16\\src\\main\\resources\\iris.data"

val data: RDD[String] = sc.textFile(datapath)

// \* 3-将数据进行转化----通过Row()对象转换，case样例类的方式，toDF方式

// map(lambda x=>x.split(,)) 匿名表达式-函数

val irisData: RDD[IrisData] = data.map(\_.split(",")).map(filed => IrisData(filed(0).toDouble, filed(1).toDouble,

filed(2).toDouble, filed(3).toDouble, filed(4)))

val irisDataFrame: DataFrame = irisData.toDF()

// \* 4-查看schema信息转换

println("iris schema value is:")

irisDataFrame.printSchema()

// iris schema value is:

// root

// |-- sepal\_length: double (nullable = false)

// |-- sepal\_width: double (nullable = false)

// |-- petal\_length: double (nullable = false)

// |-- petal\_width: double (nullable = false)

// |-- classlabel: string (nullable = true)

}

}

#### （2）Row方式

package sparkMllib\_classfication\_day03.IrisDemo

import org.apache.spark.rdd.RDD

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

import org.apache.spark.{SparkConf, SparkContext}

import org.apache.spark.sql.{DataFrame, Row, SparkSession}

/\*\*

\* DESC: 目标：通过SparkCore实现Iris数据的读取转化为DF

\* Complete data processing and modeling process steps:

\* 1-准备环境

\* 2-准备数据---读取iris的数据

\* 3-将数据进行转化----通过Row()对象转换，case样例类的方式，toDF方式

\* 4-查看schema信息转换

\* sepal\_length,sepal\_width,petal\_length,petal\_width,class

\*/

object IrisSparkCoreLoader2 {

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

// \* 1-准备环境

val conf: SparkConf = new SparkConf().setAppName("IrisSparkCoreLoader").setMaster("local[\*]")

val spark: SparkSession = SparkSession.builder().config(conf).getOrCreate()

val sc: SparkContext = spark.sparkContext

sc.setLogLevel("WARN")

import spark.implicits.\_

// \* 2-准备数据---读取iris的数据

val datapath = "D:\\BigData\\Workspace\\SparkMachineLearningTest\\SparkMllib\_BigDataSH16\\src\\main\\resources\\iris.data"

val data: RDD[String] = sc.textFile(datapath)

// \* 3-将数据进行转化----通过Row()对象转换，case样例类的方式，toDF方式

// map(lambda x=>x.split(,)) 匿名表达式-函数

val irisData: RDD[Row] = data.map(\_.split(",")).map(filed => Row(filed(0).toDouble, filed(1).toDouble, filed(2).toDouble, filed(3).toDouble, filed(4)))

//sepal\_length,sepal\_width,petal\_length,petal\_width,class

//way1

/\*val schema: StructType = new StructType().add("sepal\_length", DoubleType, true)

.add("sepal\_width", DoubleType, true) //"double"

.add("petal\_length", DoubleType, true)

.add("petal\_width", DoubleType, true)

.add("class\_label", StringType, true)\*/

//way2

val schema = StructType(StructField("sepal\_length", DoubleType, true) ::

StructField("sepal\_width", DoubleType, false) ::

StructField("petal\_length", DoubleType, false) ::

StructField("petal\_width", DoubleType, false) ::

StructField("class\_label", StringType, false) :: Nil)

schema

//way3--官网中

// val field="a b c d e"

// filed.split(" ").map(x=>Row(x(0)))

val irisDataFrame: DataFrame = spark.createDataFrame(irisData, schema)

// \* 4-查看schema信息转换

println("iris schema value is:")

irisDataFrame.printSchema()

// iris schema value is:

// root

// |-- sepal\_length: double (nullable = false)

// |-- sepal\_width: double (nullable = false)

// |-- petal\_length: double (nullable = false)

// |-- petal\_width: double (nullable = false)

// |-- classlabel: string (nullable = true)

}

}

#### （4）SQL方式

package sparkMllib\_classfication\_day03.IrisDemo

import org.apache.spark.ml.feature.\_

import org.apache.spark.sql.{DataFrame, SparkSession}

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

import org.apache.spark.{SparkConf, SparkContext}

/\*\*

\* DESC: 使用SQL方式读取数据

\* Complete data processing and modeling process steps:

\* 1-准备环境

\* 2-准备读取数据---option方法读取

\* 3-解析数据

\* 4-打印schema

\*/

object IrisSparkSQLFeaturesEngineer {

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

// \* 1-准备环境

val conf: SparkConf = new SparkConf().setAppName("IrisSparkCoreLoader").setMaster("local[\*]")

val spark: SparkSession = SparkSession.builder().config(conf).getOrCreate()

val sc: SparkContext = spark.sparkContext

sc.setLogLevel("WARN")

import spark.implicits.\_

// \* 2-准备读取数据---option方法读取

val datapath = "D:\\BigData\\Workspace\\SparkMachineLearningTest\\SparkMllib\_BigDataSH16\\src\\main\\resources\\iris.csv"

val data: DataFrame = spark.read.format("csv").option("header", "true").option("inferschema", true).load(datapath)

// \* 3-解析数据

data.printSchema()

data.show(false)

// \* 4-打印schema

// root

// |-- sepal\_length: double (nullable = true)

// |-- sepal\_width: double (nullable = true)

// |-- petal\_length: double (nullable = true)

// |-- petal\_width: double (nullable = true)

// |-- class: string (nullable = true)

//1-首先将数据的标签列进行labelencoder的编码的操作0-1-2

val strIndex: StringIndexer = new StringIndexer().setInputCol("class").setOutputCol("labelclass")

val strModel: StringIndexerModel = strIndex.fit(data)

val strResult: DataFrame = strModel.transform(data)

strResult.show(false)

//2-可以将4个特征列转化为3个特征列

//2-1特征选择------df.secelt------ChiSquareSeletor

data.select("sepal\_length").show(false)

data.select($"sepal\_length").show(false)

data.select(col("sepal\_length"))

data.select($"sepal\_length", col("sepal\_width")).show(false)

val vec: VectorAssembler = new VectorAssembler()

.setInputCols(Array("sepal\_length", "sepal\_width", "petal\_length", "petal\_width"))

.setOutputCol("features")

val vecResult: DataFrame = vec.transform(data)

val strIndex1: StringIndexer = new StringIndexer().setInputCol("class").setOutputCol("labelclass")

val strModel1: StringIndexerModel = strIndex1.fit(vecResult)

val strResult1: DataFrame = strModel1.transform(vecResult)

println("strResult1")

strResult1.show(false)

strResult1.printSchema()

//卡方验证选特征

val chi: ChiSqSelector = new ChiSqSelector().setFeaturesCol("features").setLabelCol("labelclass").setNumTopFeatures(3)

val chiModel: ChiSqSelectorModel = chi.fit(strResult1)

val chiResult: DataFrame = chiModel.transform(strResult1)

chiResult.show(false)

// +------------+-----------+------------+-----------+-----------+-----------------+----------+----------------------------------+

// |sepal\_length|sepal\_width|petal\_length|petal\_width|class |features |labelclass|chiSqSelector\_761366e608ad\_\_output|

// +------------+-----------+------------+-----------+-----------+-----------------+----------+----------------------------------+

// |5.1 |3.5 |1.4 |0.2 |Iris-setosa|[5.1,3.5,1.4,0.2]|0.0 |[5.1,1.4,0.2] |

// |4.9 |3.0 |1.4 |0.2 |Iris-setosa|[4.9,3.0,1.4,0.2]|0.0 |[4.9,1.4,0.2] |

// |4.7 |3.2 |1.3 |0.2 |Iris-setosa|[4.7,3.2,1.3,0.2]|0.0 |[4.7,1.3,0.2] |

// |4.6 |3.1 |1.5 |0.2 |Iris-setosa|[4.6,3.1,1.5,0.2]|0.0 |[4.6,1.5,0.2] |

// |5.0 |3.6 |1.4 |0.2 |Iris-setosa|[5.0,3.6,1.4,0.2]|0.0 |[5.0,1.4,0.2] |

// |5.4 |3.9 |1.7 |0.4 |Iris-setosa|[5.4,3.9,1.7,0.4]|0.0 |[5.4,1.7,0.4] |

//2-2特征降维------PCA

println("pca transfomation:")

val pca: PCA = new PCA().setInputCol("features").setOutputCol("pca\_features").setK(3)

val pcaModel: PCAModel = pca.fit(vecResult)

val pcaRE: DataFrame = pcaModel.transform(vecResult)

// +------------+-----------+------------+-----------+-----------+-----------------+----------------------------------------+

// |sepal\_length|sepal\_width|petal\_length|petal\_width|class |features |pca\_features |

// +------------+-----------+------------+-----------+-----------+-----------------+----------------------------------------+

// |5.1 |3.5 |1.4 |0.2 |Iris-setosa|[5.1,3.5,1.4,0.2]|[-2.827135972679027,-5.641331045573321] |

// |4.9 |3.0 |1.4 |0.2 |Iris-setosa|[4.9,3.0,1.4,0.2]|[-2.7959524821488437,-5.145166883252896]|

// |4.7 |3.2 |1.3 |0.2 |Iris-setosa|[4.7,3.2,1.3,0.2]|[-2.6215235581650584,-5.177378121203909]|

// |4.6 |3.1 |1.5 |0.2 |Iris-setosa|[4.6,3.1,1.5,0.2]|[-2.7649059004742402,-5.003599415056946]|

// |5.0 |3.6 |1.4 |0.2 |Iris-setosa|[5.0,3.6,1.4,0.2]|[-2.7827501159516603,-5.648648294377395]|

// |5.4 |3.9 |1.7 |0.4 |Iris-setosa|[5.4,3.9,1.7,0.4]|[-3.231445736773378,-6.062506444034077] |

val pca1: PCA = new PCA().setInputCol("pca\_features").setOutputCol("pca11\_features").setK(2)

val pcaModel1: PCAModel = pca1.fit(pcaRE)

pcaModel1.transform(pcaRE).show(false)

}

}

#### （5）相关系数代码

package sparkMllib\_classfication\_day03.IrisDemo

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

import org.apache.spark.sql.{DataFrame, SparkSession}

import org.apache.spark.{SparkConf, SparkContext}

import org.apache.spark.ml.stat.Correlation

/\*\*

\* DESC: 使用SQL方式读取数据

\* Complete data processing and modeling process steps:

\* 1-准备环境

\* 2-准备读取数据---option方法读取

\* 3-解析数据

\* 4-打印schema

\*/

object IrisSparkSQLStaticesDemo {

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

// \* 1-准备环境

val conf: SparkConf = new SparkConf().setAppName("IrisSparkCoreLoader").setMaster("local[\*]")

val spark: SparkSession = SparkSession.builder().config(conf).getOrCreate()

val sc: SparkContext = spark.sparkContext

sc.setLogLevel("WARN")

// \* 2-准备读取数据---option方法读取

val datapath = "D:\\BigData\\Workspace\\SparkMachineLearningTest\\SparkMllib\_BigDataSH16\\src\\main\\resources\\iris.csv"

val data: DataFrame = spark.read.format("csv").option("header", "true").option("inferschema", true).load(datapath)

// \* 3-解析数据

data.printSchema()

data.show(false)

// \* 4-打印schema

// root

// |-- sepal\_length: double (nullable = true)

// |-- sepal\_width: double (nullable = true)

// |-- petal\_length: double (nullable = true)

// |-- petal\_width: double (nullable = true)

// |-- class: string (nullable = true)

val vec: VectorAssembler = new VectorAssembler().setInputCols(Array("sepal\_length", "sepal\_width", "petal\_length", "petal\_width"))

.setOutputCol("features")

val vecResult: DataFrame = vec.transform(data)

//Compute the Pearson correlation matrix for the input Dataset of Vectors.

val corr: DataFrame = Correlation.corr(vecResult, "features", "pearson")

println("corr matrix is:")

corr.show(false)

}

}

#### （6）特征工程实践

package sparkMllib\_classfication\_day03.IrisDemo

import org.apache.spark.ml.feature.\_

import org.apache.spark.sql.{DataFrame, SparkSession}

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

import org.apache.spark.{SparkConf, SparkContext}

/\*\*

\* DESC: 使用SQL方式读取数据

\* Complete data processing and modeling process steps:

\* 1-准备环境

\* 2-准备读取数据---option方法读取

\* 3-解析数据

\* 4-打印schema

\*/

object IrisSparkSQLFeaturesEngineer {

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

// \* 1-准备环境

val conf: SparkConf = new SparkConf().setAppName("IrisSparkCoreLoader").setMaster("local[\*]")

val spark: SparkSession = SparkSession.builder().config(conf).getOrCreate()

val sc: SparkContext = spark.sparkContext

sc.setLogLevel("WARN")

import spark.implicits.\_

// \* 2-准备读取数据---option方法读取

val datapath = "D:\\BigData\\Workspace\\SparkMachineLearningTest\\SparkMllib\_BigDataSH16\\src\\main\\resources\\iris.csv"

val data: DataFrame = spark.read.format("csv").option("header", "true").option("inferschema", true).load(datapath)

// \* 3-解析数据

data.printSchema()

data.show(false)

// \* 4-打印schema

// root

// |-- sepal\_length: double (nullable = true)

// |-- sepal\_width: double (nullable = true)

// |-- petal\_length: double (nullable = true)

// |-- petal\_width: double (nullable = true)

// |-- class: string (nullable = true)

//1-首先将数据的标签列进行labelencoder的编码的操作0-1-2

val strIndex: StringIndexer = new StringIndexer().setInputCol("class").setOutputCol("labelclass")

val strModel: StringIndexerModel = strIndex.fit(data)

val strResult: DataFrame = strModel.transform(data)

strResult.show(false)

//2-可以将4个特征列转化为3个特征列

//2-1特征选择------df.secelt------ChiSquareSeletor

data.select("sepal\_length").show(false)

data.select($"sepal\_length").show(false)

data.select(col("sepal\_length"))

data.select($"sepal\_length", col("sepal\_width")).show(false)

val vec: VectorAssembler = new VectorAssembler()

.setInputCols(Array("sepal\_length", "sepal\_width", "petal\_length", "petal\_width"))

.setOutputCol("features")

val vecResult: DataFrame = vec.transform(data)

val strIndex1: StringIndexer = new StringIndexer().setInputCol("class").setOutputCol("labelclass")

val strModel1: StringIndexerModel = strIndex1.fit(vecResult)

val strResult1: DataFrame = strModel1.transform(vecResult)

println("strResult1")

strResult1.show(false)

strResult1.printSchema()

//卡方验证选特征

val chi: ChiSqSelector = new ChiSqSelector().setFeaturesCol("features").setLabelCol("labelclass").setNumTopFeatures(3)

val chiModel: ChiSqSelectorModel = chi.fit(strResult1)

val chiResult: DataFrame = chiModel.transform(strResult1)

chiResult.show(false)

// +------------+-----------+------------+-----------+-----------+-----------------+----------+----------------------------------+

// |sepal\_length|sepal\_width|petal\_length|petal\_width|class |features |labelclass|chiSqSelector\_761366e608ad\_\_output|

// +------------+-----------+------------+-----------+-----------+-----------------+----------+----------------------------------+

// |5.1 |3.5 |1.4 |0.2 |Iris-setosa|[5.1,3.5,1.4,0.2]|0.0 |[5.1,1.4,0.2] |

// |4.9 |3.0 |1.4 |0.2 |Iris-setosa|[4.9,3.0,1.4,0.2]|0.0 |[4.9,1.4,0.2] |

// |4.7 |3.2 |1.3 |0.2 |Iris-setosa|[4.7,3.2,1.3,0.2]|0.0 |[4.7,1.3,0.2] |

// |4.6 |3.1 |1.5 |0.2 |Iris-setosa|[4.6,3.1,1.5,0.2]|0.0 |[4.6,1.5,0.2] |

// |5.0 |3.6 |1.4 |0.2 |Iris-setosa|[5.0,3.6,1.4,0.2]|0.0 |[5.0,1.4,0.2] |

// |5.4 |3.9 |1.7 |0.4 |Iris-setosa|[5.4,3.9,1.7,0.4]|0.0 |[5.4,1.7,0.4] |

//2-2特征降维------PCA

println("pca transfomation:")

val pca: PCA = new PCA().setInputCol("features").setOutputCol("pca\_features").setK(3)

val pcaModel: PCAModel = pca.fit(vecResult)

val pcaRE: DataFrame = pcaModel.transform(vecResult)

// +------------+-----------+------------+-----------+-----------+-----------------+----------------------------------------+

// |sepal\_length|sepal\_width|petal\_length|petal\_width|class |features |pca\_features |

// +------------+-----------+------------+-----------+-----------+-----------------+----------------------------------------+

// |5.1 |3.5 |1.4 |0.2 |Iris-setosa|[5.1,3.5,1.4,0.2]|[-2.827135972679027,-5.641331045573321] |

// |4.9 |3.0 |1.4 |0.2 |Iris-setosa|[4.9,3.0,1.4,0.2]|[-2.7959524821488437,-5.145166883252896]|

// |4.7 |3.2 |1.3 |0.2 |Iris-setosa|[4.7,3.2,1.3,0.2]|[-2.6215235581650584,-5.177378121203909]|

// |4.6 |3.1 |1.5 |0.2 |Iris-setosa|[4.6,3.1,1.5,0.2]|[-2.7649059004742402,-5.003599415056946]|

// |5.0 |3.6 |1.4 |0.2 |Iris-setosa|[5.0,3.6,1.4,0.2]|[-2.7827501159516603,-5.648648294377395]|

// |5.4 |3.9 |1.7 |0.4 |Iris-setosa|[5.4,3.9,1.7,0.4]|[-3.231445736773378,-6.062506444034077] |

val pca1: PCA = new PCA().setInputCol("pca\_features").setOutputCol("pca11\_features").setK(2)

val pcaModel1: PCAModel = pca1.fit(pcaRE)

pcaModel1.transform(pcaRE).show(false)

}

}

### 6.6.1均值方差分析

代码1：均值方差

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

import org.apache.spark.mllib.stat.Statistics

import org.apache.spark.{SparkConf, SparkContext}

object iris\_mean\_Std {

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("irisAll ") //设定名称

val sc = new SparkContext(conf) //创建环境变量实例

val data = sc.textFile("/Users/zhao-chj/develop/IdeaProjects/SparkTest/src/main/scala/com/test/spark/ASparkMllibBase/base21/moviesRec\_1/SparkDataProcess/data/SparkPro2/C13/data/SpetalLength.csv")

val data1 = data.map(\_.toDouble).map(line => Vectors.dense(line)) //转成Vector格式

val summary = Statistics.colStats(data1) //计算统计量

println("全部Sepal.Length的均值为：" + summary.mean) //打印均值

println("全部Sepal.Length的方差为：" + summary.variance) //打印方差

//计算向量整体之间的距离

println("全部Sepal.Length的l1-value为：" + summary.normL1) //打印l1

println("全部Sepal.Length的l2-value为：" + summary.normL2) //打印l2

}

}

### 6.6.2相关系数分析

代码2：--相关系数

import org.apache.spark.ml.linalg.Matrix

import org.apache.spark.ml.stat.Correlation

import org.apache.spark.mllib.stat.Statistics

import org.apache.spark.sql.{Row, SparkSession}

import org.apache.spark.{SparkConf, SparkContext}

object irisTest1 {

def main(args: Array[String]) {

val conf = new SparkConf().setMaster("local").setAppName("irisTest")

val sc = new SparkContext(conf)

val data = sc.textFile(

"/Users/zhao-chj/develop/IdeaProjects/SparkTest/src/main/scala/com/test/spark/ASparkMllibBase/base21/moviesRec\_1/SparkDataProcess/data/SparkPro2/C13/data/iris.data")

data.take(2).foreach(println(\_))

//1--使用rdd获取对应列的特征进行相似度对比方法1

//val rdd3 = data.map(line => (line.split(",")(0),line .split(",")(3)))

val rdd4 = data

//flatMap(line => line.split(","))

.map(line => (line.split(",")(0))).map(\_.toDouble)

val rdd5 = data

//flatMap(line => line.split(","))

.map(line => (line.split(",")(3))).map(\_.toDouble)

println("=" \* 100)

//2--使用rdd获取对应列的特征进行相似度对比方法1

// val rdd7=data.map {

// x =>

// val column = x.split(",")

// (column(0), column(1), column(2))

// }.foreach(println(\_))

// val rdd5 = data.map {

// x =>

// val column = x.split(",")

// (column(0))

// }.map(\_.toDouble)

// val rdd6 = data.map {

// x =>

// val column = x.split(",")

// (column(3))

// }.map(\_.toDouble)

// rdd6.foreach(println(\_))

val corr = Statistics.corr(rdd4, rdd5)

println("corr relation is", corr) //(corr relation is,0.8179536333691776)

}

}

### 6.6.3Datafrmae统计分析

代码3：SparkSql实现iris的各种函数的统计

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

import org.apache.spark.ml.linalg.{Matrix, Vector}

import org.apache.spark.ml.stat.{ChiSquareTest, Correlation}

import org.apache.spark.mllib.stat.Statistics

import org.apache.spark.sql.{Row, SparkSession}

import org.apache.spark.{SparkConf, SparkContext}

case class IrisData(SepalWidth: Double, SepalLength: Double, PetalLength: Double, PetalWidth: Double, Name: String)

object irisTest {

def main(args: Array[String]) {

val conf = new SparkConf().setMaster("local").setAppName("irisTest")

val sc = new SparkContext(conf)

val spark: SparkSession = SparkSession.builder().getOrCreate()

import spark.implicits.\_

val data = sc.textFile(

"/Users/zhao-chj/develop/IdeaProjects/SparkTest/src/main/scala/com/test/spark/ASparkMllibBase/base21/moviesRec\_1/SparkDataProcess/data/SparkPro2/C13/data/iris.data")

val rowData = data.map(\_.split(",")).

map(fields => IrisData(

fields(0).toDouble,

fields(1).toDouble,

fields(2).toDouble,

fields(3).toDouble,

fields(4)))

.toDF()

rowData.show()

rowData.printSchema()

//rowData.select("SepalWidth").write.format("csv").save("/Users/zhao-chj/develop/IdeaProjects/SparkTest/src/main/scala/com/test/spark/ASparkMllibBase/base21/moviesRec\_1/SparkDataProcess/data/SparkPro2/C13/data/SpetalLength1.csv")

rowData.createOrReplaceTempView("rowdata")

//spark.sql("select SepalWidth from rowdata where Name='Iris-setosa'").show()

//println(spark.sql("select SepalWidth from rowdata where Name='Iris-setosa'").count())

//spark.sql("select SepalWidth from rowdata where Name='Iris-setosa'").write.format("csv").save("/Users/zhao-chj/develop/IdeaProjects/SparkTest/src/main/scala/com/test/spark/ASparkMllibBase/base21/moviesRec\_1/SparkDataProcess/data/SparkPro2/C13/data/SpetalLengthSetosa1.csv")

val df4 = spark.sql("SELECT mean(SepalWidth),variance(SepalWidth),stddev(SepalWidth),corr(SepalWidth,Name),skewness(SepalWidth),kurtosis(SepalWidth) FROM rowdata")

// df4.show()

val assembler = new VectorAssembler()

.setInputCols(Array("SepalWidth", "SepalLength", "PetalLength", "PetalWidth")).setOutputCol("features")

val df5 = assembler.transform(rowData)

val df6 = df5.select("features")

val Row(coeff1: Matrix) = Correlation.corr(df6, "features").head

println(s"Pearson correlation matrix:\n $coeff1")

val Row(coeff2: Matrix) = Correlation.corr(df6, "features", "spearman").head

println(s"Spearman correlation matrix:\n $coeff2")

val df\_final = new StringIndexer().setInputCol("Name").setOutputCol("label").fit(df5)

val df\_final\_model = df\_final.transform(df5)

val df7 = df\_final\_model.select("label", "features")

// val df = data.toDF("label", "features")

// ChiSquareTest对标签的每个功能进行Pearson独立测试。对于每个特征，

// （特征，标签）对被转换为矩阵，对其计算卡方统计量。所有标签和特征值必须是分类的。

// http://spark.apache.org/docs/latest/ml-statistics.html

val chi = ChiSquareTest.test(df7, "features", "label").head

println(s"pValues = ${chi.getAs[Vector](0)}")

println(s"degreesOfFreedom ${chi.getSeq[Int](1).mkString("[", ",", "]")}")

println(s"statistics ${chi.getAs[Vector](2)}")

//2.4.0测试可用

// val df = df\_final\_model.toDF( "weight","features")

// val (meanVal, varianceVal) = df.select(metrics("mean", "variance")

// .summary($"features", $"weight").as("summary"))

// .select("summary.mean", "summary.variance")

// .as[(Vector, Vector)].first()

//

// println(s"with weight: mean = ${meanVal}, variance = ${varianceVal}")

//

// val (meanVal2, varianceVal2) = df.select(mean($"features"), variance($"features"))

// .as[(Vector, Vector)].first()

//

// println(s"without weight: mean = ${meanVal2}, sum = ${varianceVal2}")

}

}

### 6.6.4SparkMllib特征工程构建

代码5建模前的特征工程：

import org.apache.spark.SparkConf

import org.apache.spark.ml.classification.DecisionTreeClassifier

import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator

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

import org.apache.spark.sql.SparkSession

import scala.util.Random

object irisTestModel {

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

val conf = new SparkConf().setMaster("local").setAppName("iris")

val spark = SparkSession.builder().config(conf).getOrCreate()

spark.sparkContext.setLogLevel("WARN")

val file = spark.read.format("csv").load("iris.data")

import spark.implicits.\_

val random = new Random()

val data = file.map(row => {

val label = row.getString(4) match {

case "Iris-setosa" => 0

case "Iris-versicolor" => 1

case "Iris-virginica" => 2

}

//i. Feature Extractors way2  
//val classToIndex: StringIndexer = new //StringIndexer().setInputCol("class").setOutputCol("indexedClass").setHandleInvalid("keep")

(row.getString(0).toDouble,

row.getString(1).toDouble,

row.getString(2).toDouble,

row.getString(3).toDouble,

label,

random.nextDouble())

}).toDF("\_c0", "\_c1", "\_c2", "\_c3", "label", "rand").sort("rand")

val assembler = new VectorAssembler().setInputCols(Array("\_c0", "\_c1", "\_c2", "\_c3")).setOutputCol("features")

val dataset = assembler.transform(data)

val Array(train, test) = dataset.randomSplit(Array(0.8, 0.2))

val dt = new DecisionTreeClassifier().setFeaturesCol("features").setLabelCol("label")

val model = dt.fit(train)

val result = model.transform(test)

result.show()

val evaluator = new MulticlassClassificationEvaluator()

.setLabelCol("label")

.setPredictionCol("prediction")

.setMetricName("accuracy")

val accuracy = evaluator.evaluate(result)

println(s"""accuracy is $accuracy""")

}

}

## 6.7项目案例Homeprise数据分析实践

字段描述分别为：mlsNum: Double, 城市city: String, 平方英尺sqFt: Double, 卧室数据bedrooms: Double, 卫生间数据bathrooms: Double,车库garage: Double, 年龄age: Double,房屋占地面积 acres: Double, 房屋价格price: Double

### 6.7.1数据描述

4424109|Apple Valley|1634.0|2|2|2|33|0.04|119900.0

4404211|Rosemount|13837.0|4|6|4|17|14.46|3500000.0

4339082|Burnsville|9040.0|4|6|8|12|0.74|2690000.0

4362154|Lakeville|6114.0|7|5|12|25|14.83|1649000.0

4388419|Lakeville|6546.0|5|5|11|38|5.28|1575000.0

4188305|Rosemount|1246.0|4|1|2|143|56.28|1295000.0

4350149|Eagan|8699.0|5|6|7|28|2.62|1195000.0

4409729|Rosemount|6190.0|7|7|7|22|4.128|1195000.0

4408821|Lakeville|5032.0|5|5|3|9|1.1|1125000.0

4342395|Lakeville|4412.0|4|5|4|9|0.924|1100000.0

4361031|Lakeville|5451.0|5|5|2|22|23.83|975000.0

4424555|Apple Valley|8539.0|5|6|6|20|2.399|975000.0

4416412|Rosemount|4910.0|5|4|3|29|7.99|799000.0

4420237|Apple Valley|5000.0|4|4|3|14|0.77|796000.0

4392412|Eagan|7000.0|4|5|3|21|1.65|789900.0

4432729|Rosemount|6300.0|5|5|3|22|4.724|789000.0

4349895|Lakeville|5001.0|4|4|6|13|2.62|778500.0

4376726|Burnsville|5138.0|4|5|3|24|1.83|749900.0

### 6.7.2SparkMllib数据分析及特征构建过程

package com.test.spark.ASparkMllibBase.base21.moviesRec\_1.SparkDataProcess.data.HomePrice

import org.apache.hadoop.conf.Configuration

import org.apache.hadoop.fs.{Path, FileSystem}

import org.apache.spark.mllib.feature.{StandardScalerModel, StandardScaler}

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

import org.apache.spark.mllib.regression.{LinearRegressionWithSGD, LabeledPoint}

import org.apache.spark.mllib.stat.Statistics

import org.apache.spark.{SparkContext, SparkConf}

case class Home(mlsNum: Double, city: String, sqFt: Double, bedrooms: Double, bathrooms: Double,

garage: Double, age: Double, acres: Double, price: Double)

object HomePriceRecommender extends Serializable {

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

val sc = new SparkContext(new SparkConf().setMaster("local[2]").setAppName("Home Price Recommender"))

//val base = "hdfs:///user/root/homeprice.data"

val base = "/Users/zhao-chj/develop/IdeaProjects/SparkTest/src/main/scala/com/test/spark/ASparkMllibBase/base21/moviesRec\_1/SparkDataProcess/data/HomePrice/homeprice.data"

val homeData = sc.textFile(base)

val parsed = homeData.map(line => parse(line))

// look at some statistics of the data

val priceStats = Statistics.colStats(parsed.map(home => Vectors.dense(home.price)))

println("Price mean: " + priceStats.mean)

println("Price max: " + priceStats.max)

println("Price min: " + priceStats.min)

// filter out anomalous data

val filtered = parsed.filter(home => (home.price > 100000.0 && home.price < 400000.0 && home.sqFt > 1000.0))

// see how correlated price and square feet are

val corr = Statistics.corr(filtered.map(home => home.price), filtered.map(home => home.sqFt))

println("Price and square feet corr: " + corr)

// convert to labeled data for MLLib

val labelData = filtered.map { home =>

LabeledPoint(home.price, Vectors.dense(home.age, home.bathrooms,

home.bedrooms, home.garage, home.sqFt))

}.cache()

// Scale features to 0 mean and common variance

val scaler = new StandardScaler(withMean = true, withStd = true).fit(labelData.map(x => x.features))

println("Scaler mean: " + scaler.mean.toArray.mkString(","))

// println("Scaler variance: " + scaler.variance.toArray.mkString(","))

val scaledData = labelData.map { data =>

LabeledPoint(data.label, scaler.transform(Vectors.dense(data.features.toArray)))

}

val numIterations = 1000

val stepSize = 0.2

// Setup linear regression model and ensure it finds the intercept

val linearReg = new LinearRegressionWithSGD()

linearReg.setIntercept(true)

linearReg.optimizer

.setNumIterations(numIterations)

.setStepSize(stepSize)

// run linear regresssion

val model = linearReg.run(scaledData)

println("Model: " + model)

// determine how well the model predicts the trained data's home prices

val valuesAndPreds = scaledData.map { point =>

val prediction = model.predict(point.features)

(point.label, prediction)

}

val power = valuesAndPreds.map {

case (v, p) => math.pow((v - p), 2)

}

// Mean Square Error

val MSE = power.reduce((a, b) => a + b) / power.count()

println("Mean Square Error: " + MSE)

// persist model to HDFS

sc.parallelize(Seq(model), 1).saveAsObjectFile("/Users/zhao-chj/develop/IdeaProjects/SparkTest/src/main/scala/com/test/spark/ASparkMllibBase/base21/moviesRec\_1/SparkDataProcess/data/HomePrice/linReg.model")

sc.parallelize(Seq(scaler), 1).saveAsObjectFile("/Users/zhao-chj/develop/IdeaProjects/SparkTest/src/main/scala/com/test/spark/ASparkMllibBase/base21/moviesRec\_1/SparkDataProcess/data/HomePrice/scaler.model")

}

// parse home price data into case class

def parse(line: String) = {

val split = line.split('|')

val mlsNum = split(0).toDouble

val city = split(1).toString

val sqFt = split(2).toDouble

val bedrooms = split(3).toDouble

val bathrooms = split(4).toDouble

val garage = split(5).toDouble

val age = split(6).toDouble

val acres = split(7).toDouble

val price = split(8).toDouble

Home(mlsNum, city, sqFt, bedrooms, bathrooms, garage, age, acres, price)

}

}

### 6.7.3SparkMllib模型效果测试

import org.apache.spark.{SparkConf, SparkContext}

import org.apache.spark.mllib.feature.StandardScalerModel

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

import org.apache.spark.mllib.regression.LinearRegressionModel

import io.\_

import scala.io.Source

object HomePriceCLI {

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

//https://github.com/phdata/exploring-mllib-post

val sc = new SparkContext(new SparkConf().setMaster("local[1]").setAppName("Home Price CLI"))

val path1 = "/Users/zhao-chj/develop/IdeaProjects/SparkTest/src/main/scala/com/test/spark/ASparkMllibBase/base21/moviesRec\_1/SparkDataProcess/data/HomePrice/linReg.model"

val path2 = "/Users/zhao-chj/develop/IdeaProjects/SparkTest/src/main/scala/com/test/spark/ASparkMllibBase/base21/moviesRec\_1/SparkDataProcess/data/HomePrice/scaler.model"

val linRegModel = sc.objectFile[LinearRegressionModel](path1).first()

val scalerModel = sc.objectFile[StandardScalerModel](path2).first()

// home.age, home.bathrooms, home.bedrooms, home.garage, home.sqF

val vectorD = Vectors.dense(11.0, 2.0, 2.0, 1.0, 2200.0)

println(vectorD)

val vector = scalerModel.transform(vectorD)

println(vector)

println(linRegModel.predict(vector))

//直接运行

//println(linRegModel.predict(scalerModel.transform(Vectors.dense(11.0, 2.0, 2.0, 1.0, 2200.0))))

sc.stop()

}

}