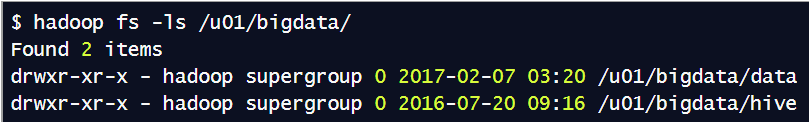
# Spark学习笔记

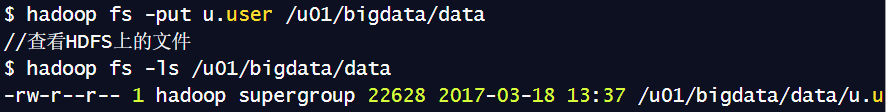
## 2.3加载数据

### 常用命令：

1. 查看当前HDFS的目录信息



1. 把本地文件u.user复制到HDFS上

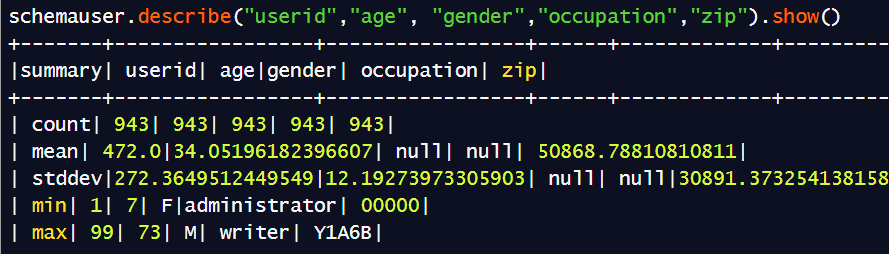


1. 通过Spark的textFile方法读取加载HDFS上的文件

*###以spark独立模式，启动Pyspark客户端*  
pyspark --master spark://master:7077 --driver-memory 1G --total-executor-cores 2  
*###导入需要的包*  
from pyspark.sql import SparkSession  
from pyspark.sql import Row  
*##初始化sparkSession*  
spark = SparkSession \  
.builder \  
.appName("Python Spark SQL basic example") \  
.config("spark.some.config.option", "some-value") \  
.getOrCreate()  
*###加载数据，并处理分割符数据*  
sc = spark.sparkContext  
userrdd = sc.textFile("hdfs://master:9000/u01/bigdata/data/u.user").map(lambda line: line.split("|"))  
*###利用反射机制推断模式(Schema),把dataframe注册为一个table*  
df = userrdd.map(lambda fields: Row(userid=fields[0], age=int(fields[1]),gender=fields[2],occupation=fields[3],zip=fields[4]))  
schemauser = spark.createDataFrame(df)  
schemauser.createOrReplaceTempView("user")

## 2.4探索数据

### 1.统计信息



### 2.数据质量分析

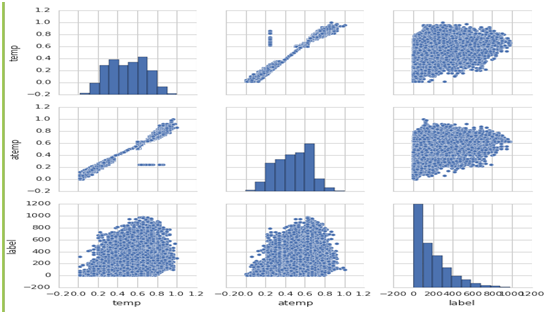
*##以spark独立模式，启动Pyspark客户端*  
pyspark --master spark://master:7077 --driver-memory 1G --total-executor-cores 2  
*###导入需要的库*  
import pandas as pd  
import matplotlib.pyplot as plt  
*###加载数据，使用标题行*  
df=pd.read\_csv("/home/hadoop/data/catering\_sale.csv",header=0)  
*##查看df的统计信息*  
df.count() *##统计非空值记录数*  
sale\_date 200  
sale\_amt 198 *###说明sale\_amt有两个空值*  
df.describe() *###获取df的统计信息*  
sale\_amt  
count 198.000000  
mean 2765.545152  
std 709.557639  
min 22.000000  
25% 2452.725000  
50% 2655.850000  
75% 3023.500000  
max 9106.440000  
  
*#建立图像*  
plt.figure()  
*#画箱线图*  
bp = df.boxplot()  
*# flies为异常值的标签*  
x = bp['fliers'][0].get\_xdata()  
y = bp['fliers'][0].get\_ydata()  
y.sort()  
  
*#用annotate添加注释*  
for i in range(len(x)):  
plt.annotate(y[i], xy = (x[i],y[i]), xytext=(x[i]+0.1-0.8/(y[i]-y[i-1]),y[i]))  
  
plt.show()

### 3.数据特征分析

*# 选取用户职业数据.*  
count\_occp = spark.sql("SELECT occupation,count(occupation) as cnt FROM user Group by occupation order by cnt")  
*#查看前5行数据*  
count\_occp.show(5)

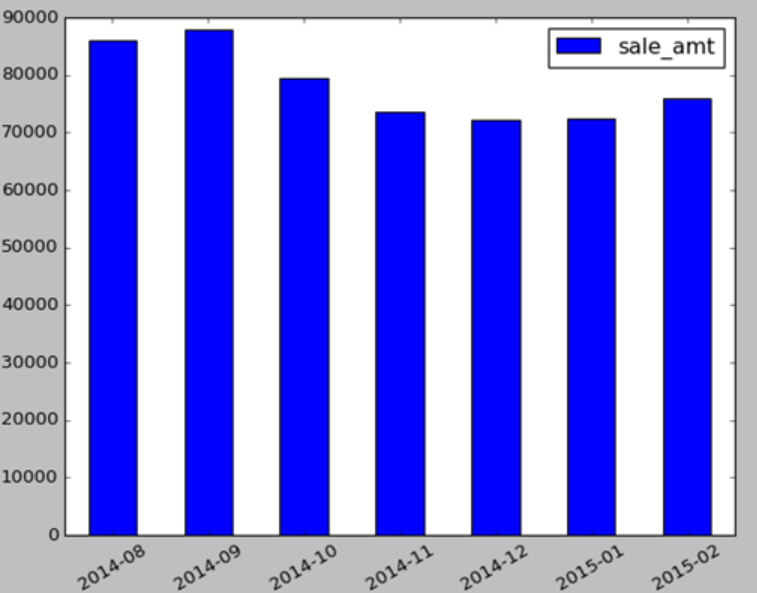
### 4特征分布及相关性分析

*###探索特征间分布、相关性等*  
import pandas as pd  
import seaborn as sns  
import matplotlib.pyplot as plt  
  
data1=pd.read\_csv('/home/hadoop/data/bike/hour.csv',header=0)  
data\_pd=data1.toPandas()  
sns.set(style='whitegrid',context='notebook')  
cols=['temp','atemp','label']  
sns.pairplot(data\_pd[cols],size=2.5)  
plt.show()



5.对比分析

*###导入需要的库*  
import pandas as pd  
*###把日期列作为索引，并转换为日期格式*  
df=pd.read\_csv("/home/hadoop/data/catering\_sale.csv",header=0,index\_col='sale\_date',parse\_dates=True)  
*###把空值置为0*  
df1=df.fillna(0)  
*###根据年月求和*  
df\_ym=df1.resample('M',how='sum')  
*##取年月*  
df2=df\_ym.to\_period('M')  
*##数据可视化*  
df2.plot(kind='bar',rot=30)



## 2.5数据预处理

### 数据清理

#### 处理缺失值

import pandas as pd  
*##读取HDFS上的数据*  
df=pd.read\_csv("/home/hadoop/data/catering\_sale.csv",header=0)  
*##定位数据集中的空值*  
df[df.isnull().values==True]  
*##显示结果如下，说明有2个空值*  
sale\_date sale\_amt  
13 2015/2/14 NaN  
32 2015/1/26 NaN  
*###以0填补空值*  
df.fillna(0)  
*##或该列的平均值填补空值*  
df['sale\_amt'].fillna(df['sale\_amt'].count())  
*##或用该列前一行值填补空值*  
df.fillna(method='pad')

#### 处理奇异值

*#读取CSV文件，保留文件标题，并创建spark 的一张derby数据库的表*  
df=spark.read.csv("/home/hadoop/data/catering\_sale.csv",header=True)  
*##转换数据类型*  
df1=df.select(df['sale\_date'],df['sale\_amt'].cast("Double"))  
*###假设把22.0奇异值替换为200.0*  
df1.replace(22.0,200.0,'sale\_amt')

from pyspark.sql.functions import \*  
*###去空格*  
df.select(trim(df.sale\_date)).show()  
*###去年份*  
df.select(substring(df.sale\_date,1,4).alias('year'),df.sale\_amt).show()

#### 数据变换

**//定义特征向量  
featuresArray =["season","yr","mnth","hr","holiday","weekday","workingday",\  
"weathersit","temp","atemp","hum","windspeed"]  
  
###把各特征组合成特征向量features  
assembler = VectorAssembler(inputCols=featuresArray,outputCol="features")  
###选择贡献度较大的前5个特征  
selectorfeature = ChiSqSelector(numTopFeatures=5, featuresCol="features",outputCol="selectedFeatures", labelCol="label")**

#### 数据归约

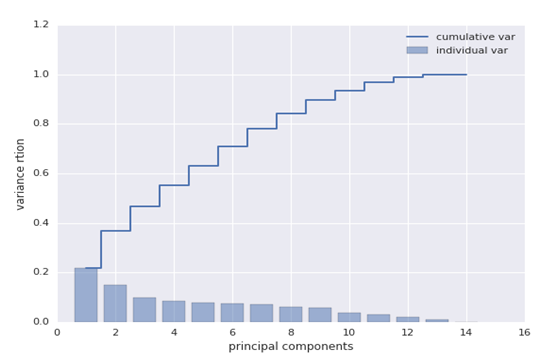
我们可以通过数据归约技术，删除或减少冗余属性（或维）、精简数据集等，使归约后数据比原数据小或小得多，但仍然接近于保持原数据的完整性

Spark ML自带数据选择算法如下：



**代码：**

**import** org.apache.spark.mllib.linalg.Vectors  
**import** org.apache.spark.mllib.linalg.Vector  
**import** org.apache.spark.mllib.linalg.distributed.RowMatrix  
**import** org.apache.spark.{SparkConf, SparkContext}  
**import** org.apache.spark.mllib.linalg.Matrix  
  
**object** Sca2\_5\_6 {  
 **def** main(args: Array[String]): Unit = {  
 **val** conf = **new** SparkConf().setAppName("SparkPi").setMaster("local[2]")  
 **val** sc = **new** SparkContext(conf)  
// val data = Array(  
// Vectors.dense(1,2,3,4,5,6,7,8,9),  
// Vectors.dense(5,6,7,8,9,0,8,6,7),  
// Vectors.dense(9,0,8,7,1,4,3,2,1),  
// Vectors.dense(6,4,2,1,3,4,2,1,5),  
// Vectors.dense(4,5,7,1,4,0,2,1,8))  
// val dataRDD = sc.parallelize(data, 2)  
// val mat: RowMatrix = new RowMatrix(dataRDD)  
// val svd = mat.computeSVD(3,true)  
// val U: RowMatrix = svd.U  
// val s: Vector = svd.s  
// val V: Matrix = svd.V  
// print(V)  
 **val** data = *Array*( Vectors.*dense*(1,2,3,4,5,6,7,8,9),  
 Vectors.*dense*(5,6,7,8,9,0,8,6,7),  
 Vectors.*dense*(9,0,8,7,1,4,3,2,1),  
 Vectors.*dense*(6,4,2,1,3,4,2,1,5),  
 Vectors.*dense*(4,5,7,1,4,0,2,1,8))  
 **val** dataRDD = sc.parallelize(data,2)  
 **val** mat: RowMatrix = **new** RowMatrix(dataRDD)  
 **val** pc: Matrix = mat.computePrincipalComponents(3)  
 *print*(pc)  
}}



## 2.6构建模型与评估

**object** Sca2\_6 {  
 **def** main(args: Array[String]): Unit = {  
 **val** spark = SparkSession  
 .*builder* .appName(**"MySparkApp"**)  
 .master(**"local[2]"**)  
 .getOrCreate()  
 **val** path=**"D:\\soft\\spark-2.4.2-bin-hadoop2.6\\data\\mllib\\sample\_libsvm\_data.txt"  
 val** data=spark.read.format(**"libsvm"**).load(path)  
 **val** *Array*(trainingData, testData) = data.randomSplit(*Array*(0.7, 0.3),1234L)  
 **val** lr = **new** LogisticRegression().setThreshold(0.6).setMaxIter(10).setRegParam(0.3).setElasticNetParam(0.8)  
 **val** lrModel = lr.fit(trainingData)  
 **val** predictions = lrModel.transform(testData)  
 predictions.show()  
 **val** evaluator = **new** BinaryClassificationEvaluator().setLabelCol(**"label"**)  
 **val** accuracy = evaluator.evaluate(predictions)  
 **val** rm2 = **new** RegressionMetrics(predictions.select(**"prediction"**, **"label"**)  
 .rdd.map(x =>(x(0).asInstanceOf[Double], x(1).asInstanceOf[Double])))  
 *println*(**"MSE: "** + rm2.meanSquaredError)  
 *println*(**"MAE: "** + rm2.meanAbsoluteError)  
 *println*(**"RMSE Squared: "** + rm2.rootMeanSquaredError)  
 *//将其作为多分类结果进行评估，可计算F1、准确率、召回率、正确率* **val** multiclassClassificationEvaluator = **new** MulticlassClassificationEvaluator()  
 **def** printlnMetric(metricName: String): Unit = {  
 *println*(metricName + **" = "** + multiclassClassificationEvaluator.setMetricName(metricName).evaluate(predictions))  
 }  
 printlnMetric(**"f1"**)*//f1 = 0.9646258503401359* printlnMetric(**"weightedPrecision"**)*//weightedPrecision = 0.9675324675324675* printlnMetric(**"weightedRecall"**)*//weightedRecall = 0.9642857142857142* printlnMetric(**"accuracy"**)*//accuracy = 0.9642857142857143  
  
 //将其作为二分类结果进行评估，可计算areaUnderROC、areaUnderPR* **val** binaryClassificationEvaluator = **new** BinaryClassificationEvaluator()  
 **def** printlnMetric1(metricName: String): Unit = {  
 *println*(metricName + **" = "** + binaryClassificationEvaluator.setMetricName(metricName).evaluate(predictions))  
 }  
  
 printlnMetric1(**"areaUnderROC"**) *//结果为areaUnderROC = 0.9944444444444444* printlnMetric1(**"areaUnderPR"**)*//结果为areaUnderPR = 0.9969948018193632  
 //分类正确且分类为1的样本数量 TP 是17* predictions.filter(**$"label"** === **$"prediction"**).filter(**$"label"**===1).count  
 *//分类正确且分类为0的样本数量 TN 是10* predictions.filter(**$"label"** === **$"prediction"**).filter(**$"label"**===0).count  
 *//分类错误且分类为0的样本数量 FN是1* predictions.filter(**$"label"** !== **$"prediction"**).filter(**$"prediction"**===0).count  
 *//分类错误且分类为1的样本数量 FP是0* predictions.filter(**$"label"** !== **$"prediction"**).filter(**$"prediction"**===1).count  
 }  
}

## 2.7组装

采用Spark pipeline能很好解决这些问题。我们只要把这些任务（数据处理、转换、构建模型、评估模型），作为pipeline的stage，按照其本身的执行次序把这些stages组装到一个pipeline上。（当然如果任务比较复杂，我们也可以采用多个pipeline，然后把这些作为pipeline的stage，组装到一个新的pipeline。）

组装的步骤一般是：  
1、创建pipeline，并各个stages依次组装在一起，如：

val pipeline = new Pipeline()  
.setStages(Array(tokenizer, hashingTF, lr))

2、在训练集上拟合这个pipeline

val model = pipeline.fit(training)

3、在测试集上，做预测。

model.transform(test).select("label", "prediction")

例子：

**import** org.apache.spark.ml.{Pipeline, PipelineModel}  
**import** org.apache.spark.ml.classification.LogisticRegression  
**import** org.apache.spark.ml.feature.{HashingTF, Tokenizer}  
**import** org.apache.spark.ml.linalg.Vector  
**import** org.apache.spark.sql.Row  
**import** org.apache.spark.sql.SparkSession  
  
**object** Sca2\_7 {  
  
 **def** main(args: Array[String]): Unit = {  
  
 **val** spark = SparkSession.*builder*.appName(**"Simple Application"**).master(**"local[2]"**).getOrCreate()  
 *// Prepare training documents from a list of (id, text, label) tuples.* **val** training = spark.createDataFrame(*Seq*(  
 (0L, **"a b c d e spark"**, 1.0),  
 (1L, **"b d"**, 0.0),  
 (2L, **"spark f g h"**, 1.0),  
 (3L, **"hadoop mapreduce"**, 0.0)  
 )).toDF(**"id"**, **"text"**, **"label"**)  
  
 *// Configure an ML pipeline, which consists of three stages: tokenizer, hashingTF, and lr.* **val** tokenizer = **new** Tokenizer()  
 .setInputCol(**"text"**)  
 .setOutputCol(**"words"**)  
 **val** hashingTF = **new** HashingTF()  
 .setNumFeatures(1000)  
 .setInputCol(tokenizer.getOutputCol)  
 .setOutputCol(**"features"**)  
 **val** lr = **new** LogisticRegression()  
 .setMaxIter(10)  
 .setRegParam(0.001)  
 **val** pipeline = **new** Pipeline()  
 .setStages(*Array*(tokenizer, hashingTF, lr))  
  
 *// Fit the pipeline to training documents.* **val** model = pipeline.fit(training)  
  
 *// Now we can optionally save the fitted pipeline to disk* model.write.overwrite().save(**"/SparkWorkSpace/tmp/spark-logistic-regression-model"**)  
  
 *// We can also save this unfit pipeline to disk* pipeline.write.overwrite().save(**"/SparkWorkSpace/tmp/unfit-lr-model"**)  
  
 *// And load it back in during production* **val** sameModel = PipelineModel.*load*(**"/SparkWorkSpace/tmp/spark-logistic-regression-model"**)  
  
 *// Prepare test documents, which are unlabeled (id, text) tuples.* **val** test = spark.createDataFrame(Seq(  
 (4L, **"spark i j k"**),  
 (5L, **"l m n"**),  
 (6L, **"spark hadoop spark"**),  
 (7L, **"apache hadoop"**)  
 )).toDF(**"id"**, **"text"**)  
  
 *// Make predictions on test documents.* model.transform(test)  
 .select(**"id"**, **"text"**, **"probability"**, **"prediction"**)  
 .collect()  
 .foreach { **case** Row(id: Long, text: String, prob: Vector, prediction: Double) =>  
 *println*(**s"($**id**, $**text**) --> prob=$**prob**, prediction=$**prediction**"**)  
 }  
 spark.stop()  
 }  
}

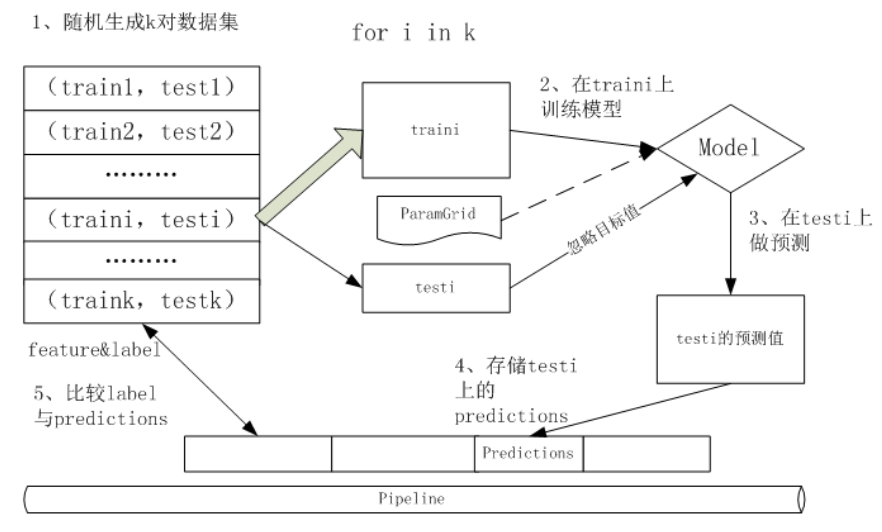
## 2.8模型选择或调优

在ML中一个重要的任务就是模型选择，或者使用给定的数据为给定的任务寻找最适合的模型或参数。这个过程也叫做调优。MLlib支持使用像CrossValidator和TrainValidationSplit这样的工具进行模型选择。这些工具需要以下的组件：  
Estimator：用户调优的算法或Pipline。  
ParamMap集合：提供参数选择，有时也叫作用户查找的参数网格(parameter grid),参数网格可以使用ParamGridBuilder来构建。  
Evaluator：衡量模型在测试数据上的拟合程度。

模型选择工具工作原理如下：  
1.将输入数据划分为训练数据和测试数据。  
2. 对于每个（训练，测试）对，遍历一组ParamMaps。用每一个ParamMap参数来拟合估计器，得到训练后的模型，再使用评估器来评估模型表现。  
3.选择性能表现最优模型对应参数表

## 2.9交叉验证 （CrossValidator）

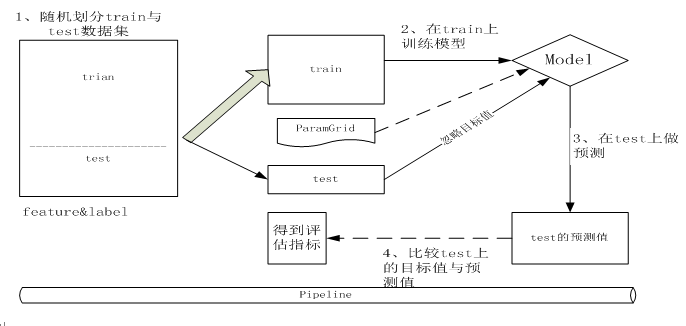
交叉验证（CrossValidator）会从将数据集切分成K折数据集合，分别用于训练和测试，。例如，K=3折时，CrossValidator会生成3个（训练数据，测试数据）对，每一个数据对的训练数据占2/3，测试数据占1/3。为了评估一个ParamMap，CrossValidator 会计算这三个不同的（训练，测试）数据集对在Estimator拟合出的模型上的平均评估指标。  
在找出最好的ParamMap后，CrossValidator 会利用此ParamMap在整个训练集上可以训练（fit）出一个泛化能力强，误差相对小的的最佳模型，整个过程处于流程化管理之中，其工作流程图如下：



虽然利用CrossValidator来训练模型，可以提升泛化能力，但其的代价也比较高，如选择k=3,regParam=(0.1,0.01)，numIters=(10,20)这样就需要对模型训练3\*2\*2=12次。然而，对比启发式的手动调优，这是选择参数的行之有效的方法。

## 2.10训练-验证切分

交叉验证的代价比较高昂，为此Spark也为超参数调优提供了训练-验证切分（TrainValidationSplit）。TrainValidationSplit创建单一的（训练，测试）数据集对。它使用trainRatio参数将数据集切分成两部分。例如，当设置trainRatio=0.8时，TrainValidationSplit将会将数据切分80%作为数据集，20%作为验证集，来生成训练、测试集对，并最终使用最好的ParamMap和完整的数据集来拟合评估器。  
相对于CrossValidator对每一个参数进行k次评估，TrainValidationSplit只对每个参数组合评估1次。因此它的评估代价没有这那么高，但是当训练数据集不够大的时候其结果相对不够可信。



## 2.11保存模型

1)保存拟合后流水线（pipeline）到磁盘

model.write.overwrite().save("/tmp/spark-logistic-regression-model")

2)保存未拟合的流水线（pipeline）到磁盘

pipeline.write.overwrite().save("/tmp/spark-logistic-regression-model")

3)把拟合后流水线部署到其他环境中。

val sameModel = PipelineModel.load("/tmp/spark-logistic-regression-model")

## 3ML Pipelines

### 3.1DataFrame

**object** Sca3\_1 {  
 **def** main(args: Array[String]): Unit = {  
 **val** spark = SparkSession.*builder*()  
 .appName("Spark SQL basic example")  
 .master("local[2]")  
 .getOrCreate()  
  
 //创建DataFrame  
 **val** df1 = spark.read.option("header", **true**).format("csv").load("file:///test/dataset/customer.csv")  
 // 转换字符类型  
 **val** df2 = df1.select(  
 df1("name").cast("String"),  
 df1("age").cast("Double"),  
 df1("gender").cast("String"))  
 //显示df2的Schema  
 df2.printSchema()  
  
 df2.createOrReplaceTempView("customer")  
 //查询  
 **val** cust1 = spark.sql("SELECT \* FROM customer WHERE age BETWEEN 30 AND 35")  
 cust1.limit(5).show  
 **val** cust2 = spark.sql("SELECT \* FROM customer WHERE gender like 'M'")  
 cust2.limit(5).show  
 }  
  
}

### 3.2 Pipeline原理

按照具体的处理逻辑来有序的组织Stages并创建一个Pipeline

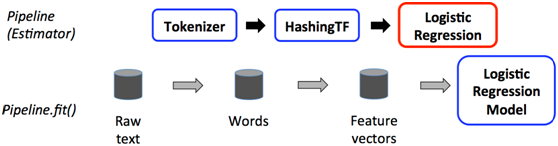


图3-1 pipeline在训练数据上的流程

整个流水线是一个估计器。所以当流水线的fit()方法运行后，会产生一个流水线模型，流水线模型是转换器。流水线模型会在测试时被调用，下面的图示说明用法

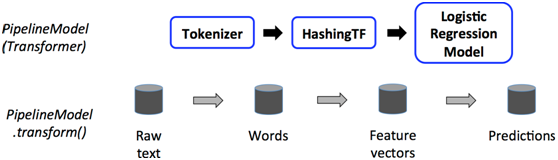


图3-2 pipeline在测试数据上的流程

上面的图示中，流水线模型和原始流水线有同样数目的阶段，然而原始流水线中的估计器此时变为了转换器。当流水线模型的transform()方法被调用于测试数据集时，数据依次经过流水线的各个阶段。每个阶段的transform()方法更新数据集，并将之传到下个阶段。  
流水线和流水线模型有助于确认训练数据和测试数据经过同样的特征处理流程。

以上两图如果合并为一图，可用如下图形表达：

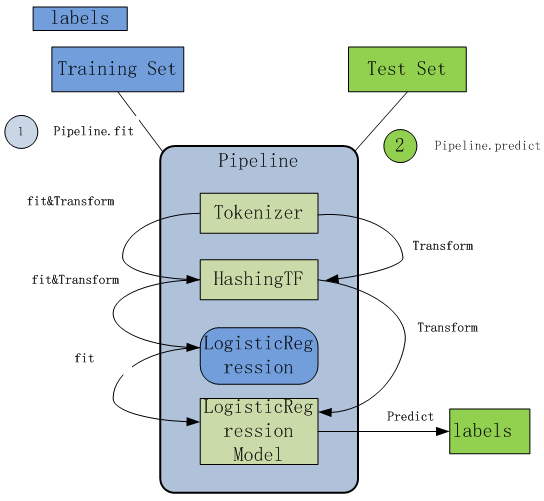


图3-3 Spark pipeline 流程图

其中Pipeline及LogisticRegression都Estimator，Tokenizer，HashingTF,LogisticRegression Model为Transformer。

### 3.3Pipeline实例

#### 3.3.1使用Estimator, Transformer, and Param实例

**import** org.apache.spark.ml.classification.LogisticRegression  
**import** org.apache.spark.ml.linalg.{Vector, Vectors}  
**import** org.apache.spark.ml.param.ParamMap  
**import** org.apache.spark.sql.{Row, SparkSession}  
  
**object** Sca3\_2 {  
 **def** main(args: Array[String]): Unit = {  
 **val** spark = SparkSession.*builder*()  
 .appName("Spark SQL basic example")  
 .master("local[2]")  
 .getOrCreate()  
  
 //从（标识、特征）元组开始训练数据.  
 **val** training = spark.createDataFrame(*Seq*(  
 (1.0, Vectors.*dense*(0.0, 1.1, 0.1)),  
 (0.0, Vectors.*dense*(2.0, 1.0, -1.0)),  
 (0.0, Vectors.*dense*(2.0, 1.3, 1.0)),  
 (1.0, Vectors.*dense*(0.0, 1.2, -0.5))  
 )).toDF("label", "features")  
  
 //创建一个LogisticRegression实例。 这个实例是一个估计器.  
 **val** lr = **new** LogisticRegression()  
 //打印参数，文档和任何默认值.  
 *println*("LogisticRegression parameters:\n" + lr.explainParams() + "\n")  
  
 //我们可以使用setter方法来设置参数.  
 lr.setMaxIter(10)  
 .setRegParam(0.01)  
  
 //训练LogisticRegression模型，这里使用了存储在lr中的参数。.  
 **val** model1 = lr.fit(training)  
 //由于模型1是模型(即由估计器生成的转换器),  
 //我们可以查看它在fit（）中使用的参数。  
 //打印参数（名称：值）对，其中名称是唯一的ID,  
 // LogisticRegression实例。  
 *println*("Model 1 was fit using parameters: " + model1.*parent*.extractParamMap)  
  
 //我们可以用ParamMap来指定参数,  
 //它支持几种指定参数的方法。  
 **val** paramMap = *ParamMap*(lr.*maxIter* -> 20)  
 .put(lr.*maxIter*, 30) //指定1个参数。 这会覆盖原来的maxIter。  
 .put(lr.*regParam* -> 0.1, lr.*threshold* -> 0.55) // 指定多个参数。  
  
 //也可以组合ParamMaps.  
 **val** paramMap2 = *ParamMap*(lr.*probabilityCol* -> "myProbability") // 修改输出列名  
 **val** paramMapCombined = paramMap ++ paramMap2  
  
 //现在使用paramMapCombined参数学习一个新的模型。  
 // paramMapCombined覆盖之前通过lr.set \*方法设置的所有参数。  
 **val** model2 = lr.fit(training, paramMapCombined)  
 *println*("Model 2 was fit using parameters: " + model2.*parent*.extractParamMap)  
  
 // 准备测试数据  
 **val** test = spark.createDataFrame(*Seq*(  
 (1.0, Vectors.*dense*(-1.0, 1.5, 1.3)),  
 (0.0, Vectors.*dense*(3.0, 2.0, -0.1)),  
 (1.0, Vectors.*dense*(0.0, 2.2, -1.5))  
 )).toDF("label", "features")  
  
 //使用Transformer.transform（）方法对测试数据进行预测  
 // LogisticRegression.transform将仅使用“特征”列  
 //请注意，model2.transform（）输出一个“myProbability”列，而不是通常的。  
 // 我们先前通过lr.probabilityCol参数重新命名了'probability'列  
 model2.transform(test)  
 .select("features", "label", "myProbability", "prediction")  
 .collect()  
 .foreach { **case** *Row*(features: Vector, label: Double, prob: Vector, prediction: Double) =>  
 *println*(s"(**$**features, **$**label) -> prob=**$**prob, prediction=**$**prediction")  
 }  
 }  
}

#### 3.3.2ML使用Pipeline实例

**import** org.apache.spark.ml.{Pipeline, PipelineModel}  
**import** org.apache.spark.ml.classification.LogisticRegression  
**import** org.apache.spark.ml.feature.{HashingTF, Tokenizer}  
**import** org.apache.spark.ml.linalg.Vector  
**import** org.apache.spark.sql.{Row, SparkSession}  
  
**object** Sca3\_3 {  
 **def** main(args: Array[String]): Unit = {  
 **val** spark = SparkSession.*builder*()  
 .appName("Spark SQL basic example")  
 .master("local[2]")  
 .getOrCreate()  
  
 //准备训练文档，（id，内容，标签）  
 **val** training = spark.createDataFrame(*Seq*(  
 (0L, "a b c d e spark", 1.0),  
 (1L, "b d", 0.0),  
 (2L, "spark f g h", 1.0),  
 (3L, "hadoop mapreduce", 0.0)  
 )).toDF("id", "text", "label")  
  
 //配置ML Pipeline，由三个stage组成，tokenizer, hashingTF, and lr.  
 **val** tokenizer = **new** Tokenizer()  
 .setInputCol("text")  
 .setOutputCol("words")  
 **val** hashingTF = **new** HashingTF()  
 .setNumFeatures(1000)  
 .setInputCol(tokenizer.getOutputCol)  
 .setOutputCol("features")  
 **val** lr = **new** LogisticRegression()  
 .setMaxIter(10)  
 .setRegParam(0.001)  
 **val** pipeline = **new** Pipeline()  
 .setStages(*Array*(tokenizer, hashingTF, lr))  
  
 //在训练数据集上使用Pipeline  
 **val** model = pipeline.fit(training)  
  
 // Now we can optionally save the fitted pipeline to disk  
 //现在可以保存安装好的流水线到磁盘上  
 model.write.overwrite().save("/tmp/spark-logistic-regression-model")  
  
 //现在可以保存未安装好的Pipeline保存到磁盘上  
 pipeline.write.overwrite().save("/tmp/unfit-lr-model")  
  
 // 装载模型  
 **val** sameModel = PipelineModel.*load*("/tmp/spark-logistic-regression-model")  
  
 //准备测试文档，不包含标签(id, text) .  
 **val** test = spark.createDataFrame(*Seq*(  
 (4L, "spark i j k"),  
 (5L, "l m n"),  
 (6L, "spark hadoop spark"),  
 (7L, "apache hadoop")  
 )).toDF("id", "text")  
 //在测试文档上做出预测.  
 sameModel.transform(test)  
 .select("id", "text", "probability", "prediction")  
 .collect()  
 .foreach { **case** *Row*(id: Long, text: String, prob: Vector, prediction: Double) =>  
 *println*(s"(**$**id, **$**text) --> prob=**$**prob, prediction=**$**prediction")  
 }  
 }  
}

## 4特征提取、转换和选择

### 4.1 特征提取

#### 4.1.1 词频－逆向文件频率（TF-IDF）

词频－逆向文件频率（TF-IDF）是一种特征向量化方法，它可以体现一个文档中词语在语料库中的重要程度。  
在下面的代码段中，我们以一组句子开始。首先使用分解器Tokenizer把句子划分为单个词语。对每一个句子（词袋），我们使用HashingTF将句子转换为特征向量，最后使用IDF重新调整特征向量。这种转换通常可以提高使用文本特征的性能。

代码：

**import** org.apache.spark.ml.feature.{HashingTF, IDF, Tokenizer}  
**import** org.apache.spark.sql.SparkSession  
  
**object** Sca4\_1 {  
 **def** main(args: Array[String]): Unit = {  
 **val** spark = SparkSession.*builder*()  
 .appName("Spark SQL basic example")  
 .master("local[2]")  
 .getOrCreate()  
 **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)  
 // CountVectorizer也可获取词频向量  
  
 **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*)  
 }  
}

#### 4.1.2 Word2Vec

Word2vec是一个Estimator，它采用一系列代表文档的词语来训练word2vecmodel。 在下面的代码段中，我们首先用一组文档，其中每一个文档代表一个词语序列。对于每一个文档，我们将其转换为一个特征向量。此特征向量可以被传递到一个学习算法。

代码:

**import** org.apache.spark.ml.feature.Word2Vec  
**import** org.apache.spark.sql.SparkSession  
  
**object** Sca4\_2 {  
 **def** main(args: Array[String]): Unit = {  
 **val** spark = SparkSession.*builder*()  
 .appName("Spark SQL basic example")  
 .master("local[2]")  
 .getOrCreate()  
 // 输入数据，每行为一个词袋，可来自语句或文档。  
 **val** documentDF = spark.createDataFrame(*Seq*(  
 "Hi I heard about Spark".split(" "),  
 "I wish Java could use case classes".split(" "),  
 "Logistic regression models are neat".split(" ")  
 ).map(Tuple1.*apply*)).toDF("text")  
 documentDF.show(5)  
 //训练从词到向量的映射  
 **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*)  
 }  
}

#### 4.1.3 计数向量器

计数向量器（Countvectorizer）和计数向量器模型（Countvectorizermodel）旨在通过计数来将一个文档转换为向量

计数向量器的使用代码:

**import** org.apache.spark.ml.feature.{CountVectorizer,CountVectorizerModel}  
**import** org.apache.spark.sql.SparkSession  
  
**object** Sca4\_3 {  
 **def** main(args: Array[String]): Unit = {  
 **val** spark = SparkSession.*builder*()  
 .appName("Spark SQL basic example")  
 .master("local[2]")  
 .getOrCreate()  
  
 **val** df = spark.createDataFrame(*Seq*(  
 (0,*Array*("a","b","c")),  
 (1,*Array*("a","b","b","c","a"))  
 )).toDF("id","words")  
 //每行text都是Array [String]类型的文档  
 df.show(5)  
 //从语料库中拟合CountVectorizerModel  
 **val** cvModel:CountVectorizerModel=**new** CountVectorizer()  
 .setInputCol("words")  
 .setOutputCol("features")  
 .setVocabSize(3)  
 .setMinDF(2)  
 .fit(df)  
 //调用fit，CountVectorizer产生CountVectorizerModel含词汇（a，b，c）  
 //或者，用先验词汇表定义CountVectorizerModel,已有词典才可用  
 **val** cvm =**new** CountVectorizerModel(*Array*("a","b","c"))  
 .setInputCol("words")  
 .setOutputCol("features")  
 //每个向量代表文档的词汇表中每个词语出现的次数。  
 cvModel.transform(df).show(**false**)  
 cvm.transform(df).show(**false**)  
 }  
}

### 4.2 特征转换

对原有特征做多种处理，如类型转换、标准化特征、新增衍生特征等等，Spark ML包中提供了很多现成转换器，例如：StringIndexer、IndexToString、OneHotEncoder、VectorIndexer，它们提供了十分方便的特征转换功能，这些转换器类都位于org.apache.spark.ml.feature包下。

#### 4.2.1分词器

分词器（Tokenization）将文本划分为独立个体（通常为单词），代码如下：

**import** org.apache.spark.sql.SparkSession  
**import** org.apache.spark.ml.feature.{RegexTokenizer,Tokenizer}  
**import** org.apache.spark.sql.functions.\_  
**object** Sca4\_4 {  
 **def** main(args: Array[String]): Unit = {  
 **val** spark = SparkSession.*builder*()  
 .appName("Spark SQL basic example")  
 .master("local[2]")  
 .getOrCreate()  
 **val** sentenceDataFrame = spark.createDataFrame(*Seq*(  
 (0,"Hi I heard about Spark"),  
 (1,"I wish Java could use case classes"),  
 (2,"Logistic,regression,models,are,neat")  
 )).toDF("id","sentence")  
 //token默认是按空格分词  
 **val** tokenizer =**new** Tokenizer().setInputCol("sentence").setOutputCol("words")  
 //Regextoken是按正则表达式分词，适用更广泛  
 **val** regexTokenizer =**new** RegexTokenizer()  
 .setInputCol("sentence")  
 .setOutputCol("words")  
 .setPattern("\\W")//或者使用 .setPattern("\\w+").setGaps(false)  
  
 **val** countTokens = *udf* {(words:Seq[String])=> words.length }  
  
 **val** tokenized = tokenizer.transform(sentenceDataFrame)  
 tokenized.select("sentence","words")  
 .withColumn("tokens", countTokens(*col*("words"))).show(**false**)  
 **val** regexTokenized = regexTokenizer.transform(sentenceDataFrame)  
 regexTokenized.select("sentence","words")  
 .withColumn("tokens", countTokens(*col*("words"))).show(**false**)  
 }  
}

#### 4.2.2 移除停用词

**移除“I”, “the”, “had”以及“a”被移除演示：**

**import** org.apache.spark.sql.SparkSession  
**import** org.apache.spark.ml.feature.StopWordsRemover  
**object** Sca4\_5 {  
 **def** main(args: Array[String]): Unit = {  
 **val** spark = SparkSession.*builder*()  
 .appName("Spark SQL basic example")  
 .master("local[2]")  
 .getOrCreate()  
 **val** remover = **new** StopWordsRemover()  
 .setInputCol("raw")  
 .setOutputCol("filtered")  
 //其中，“I”, “the”, “had”以及“a”被移除。  
 **val** dataSet = spark.createDataFrame(*Seq*(  
 (0, *Seq*("I", "saw", "the", "red", "balloon")),  
 (1, *Seq*("Mary", "had", "a", "little", "lamb"))  
 )).toDF("id", "raw")  
 remover.transform(dataSet).show(**false**)  
 }  
}

#### 4.2.3 n-gram

一个n-gram是一个长度为整数n的字序列。NGram可以用来将输入转换为n-gram。代码:

**import** org.apache.spark.sql.SparkSession  
**import** org.apache.spark.ml.feature.NGram  
**object** Sca4\_6 {  
 **def** main(args: Array[String]): Unit = {  
 **val** spark = SparkSession.*builder*()  
 .appName("Spark SQL basic example")  
 .master("local[2]")  
 .getOrCreate()  
 **val** wordDataFrame = spark.createDataFrame(*Seq*(  
 (0, *Array*("Hi", "I", "heard", "about", "Spark")),  
 (1, *Array*("I", "wish", "Java", "could", "use", "case", "classes")),  
 (2, *Array*("Logistic", "regression", "models", "are", "neat"))  
 )).toDF("id", "words")  
 //setN(2):设置为2-grams  
 **val** ngram = **new** NGram().setN(2).setInputCol("words").setOutputCol("ngrams")  
  
 **val** ngramDataFrame = ngram.transform(wordDataFrame)  
 ngramDataFrame.select("ngrams").show(**false**)  
 spark.stop()

//2grams ： [Hi I, I heard, heard about, about Spark]   
 }  
}

#### 4.2.4 二值化

二值化，通过设置阀值，将连续型的特征转化为两个值。大于阀值为1，否则为0。注：以下规范化操作一般是针对一个特征向量（dataFrame中的一个colum）来操作的。代码：

**import** org.apache.spark.sql.SparkSession  
**import** org.apache.spark.ml.feature.Binarizer  
  
**object** Sca4\_7 {  
 **def** main(args: Array[String]): Unit = {  
 **val** spark = SparkSession.*builder*()  
 .appName("Spark SQL basic example")  
 .master("local[2]")  
 .getOrCreate()  
 **val** data = *Array*((0, 0.1), (1, 0.8), (2, 0.2))  
 **val** dataFrame = spark.createDataFrame(data).toDF("id", "feature")  
 **val** binarizer: Binarizer = **new** Binarizer()  
 .setInputCol("feature")  
 .setOutputCol("binarized\_feature")  
 .setThreshold(0.5)  
 //此方法可应用在将概率值转化为二值，如logistics回归，概率转二值  
 **val** binarizedDataFrame = binarizer.transform(dataFrame)  
  
 *println*(s"Binarizer output with Threshold = **$**{binarizer.getThreshold}")  
 binarizedDataFrame.show()  
 spark.stop()  
 }  
}

#### 4.2.5 主成分分析

注意：PCA前一定要对特征向量进行规范化（标准化）！！

代码：  
**import** org.apache.spark.ml.feature.PCA  
**import** org.apache.spark.ml.feature.StandardScaler  
**import** org.apache.spark.sql.SparkSession  
  
**object** Sca4\_8 {  
 **def** main(args: Array[String]): Unit = {  
 **val** spark = SparkSession.*builder*()  
 .appName("Spark SQL basic example")  
 .master("local[2]")  
 .getOrCreate()  
 **val** rawDataFrame=spark.read.format("libsvm").load("file:///test/dataset/sample\_libsvm\_data.txt")  
 **val** scaledDataFrame=**new** StandardScaler()  
 .setInputCol("features")  
 .setOutputCol("scaledFeatures")  
 .setWithMean(**false**)//对于稀疏数据（如本次使用的数据），不要使用平均值  
 .setWithStd(**true**)  
 .fit(rawDataFrame)  
 .transform(rawDataFrame)  
 //PCA Model  
 **val** pcaModel=**new** PCA().setInputCol("scaledFeatures")  
 .setOutputCol("pcaFeatures")  
 .setK(3)//  
 .fit(scaledDataFrame)  
 //进行PCA降维  
 pcaModel.transform(scaledDataFrame).select("label","pcaFeatures").show(10,**false**)  
 //没有标准化特征向量，直接进行PCA主成分：各主成分之间值变化太大，有数量级的差别。  
 //标准化特征向量后PCA主成分，各主成分之间值基本上在同一水平上，结果更合理  
 //如何选择k值？  
 **val** pcaModel1=**new** PCA().setInputCol("scaledFeatures")  
 .setOutputCol("pcaFeatures")  
 .setK(50)//  
 .fit(scaledDataFrame)  
 **var** i=1  
 **for**( x<-pcaModel1.explainedVariance.toArray){  
 *println*(i+"\t"+x+" ")  
 i +=1  
 }  
 //运行结果（前10行），随着k的增加，精度趋于平稳。  
 spark.stop()  
 }  
}