

Spark MLlib

yiddishkop

2018 年 8 月 17 日

目录

1	Spark MLlib 简介	2
2	机器学习工作流	7
2.1	工作流简介	7
2.2	工作流实例	11
3	特征抽取, 转化, 选择	15
3.1	IF-IDF	15
3.2	word2vec	20
3.3	CountVectorizer	22
4	分类与回归	24
4.1	logistic regression 分类器	24
4.2	决策树分类器	30



Spark MLlib简介

机器学习工作流

特征抽取、转化和选择

分类与回归

聚类算法

推荐算法

机器学习参数调优

1 SP

算法



19.bb

MLlib 簡介/screenshot_2018-08-17_03-04-
38.pngMLlib /screenshot_2018-08-17_03-04-

1 SP

38.bb

MLlib 簡介/screenshot₂₀₁₈₋₀₈₋₁₇_{03-06-44.png}MLlib /screenshot₂₀₁₈₋₀₈₋₁₇_{03-06-44.png}

3

1 SP

44.bb

MLlib 簡介/screenshot2018 - 08 - 17_03 - 07 -
49.pngMLlib /screenshot2018 - 08 - 17_03 - 07 -



1 SP



49.bb

MLlib 簡介/screenshot2018 - 08 - 17_03 - 08 -
24.pngMLlib /screenshot2018 - 08 - 17_03 - 08 -

1 SP



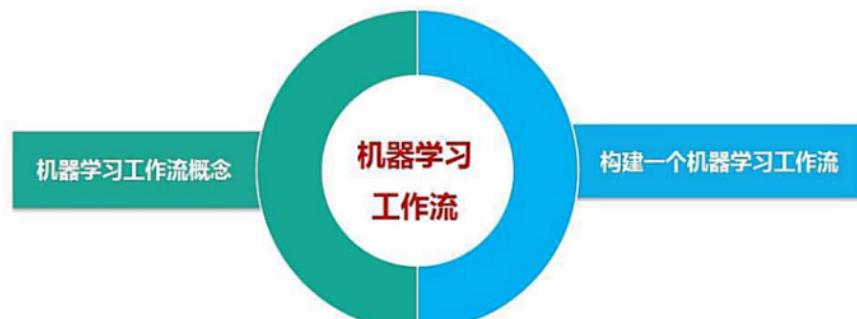
24.bb

MLlib 簡介/screenshot2018 - 08 - 17_03 - 09 -
58.pngMLlib /screenshot2018 - 08 - 17_03 - 09 -

2 机器学习工作流

2 机器学习工作流

2.1 工作流简介



机器学习工作流概念

DataFrame

使用Spark SQL中的DataFrame作为数据集，它可以容纳各种数据类型。较之RDD，DataFrame包含了schema信息，更类似传统数据库中的二维表格。它被ML Pipeline用来存储源数据。例如，DataFrame中的列可以是存储的文本、特征向量、真实标签和预测的标签等。

整个 Spark 机器学习库就是针对 DataFrame 进行一次又一次的转换。

DataFrame-1

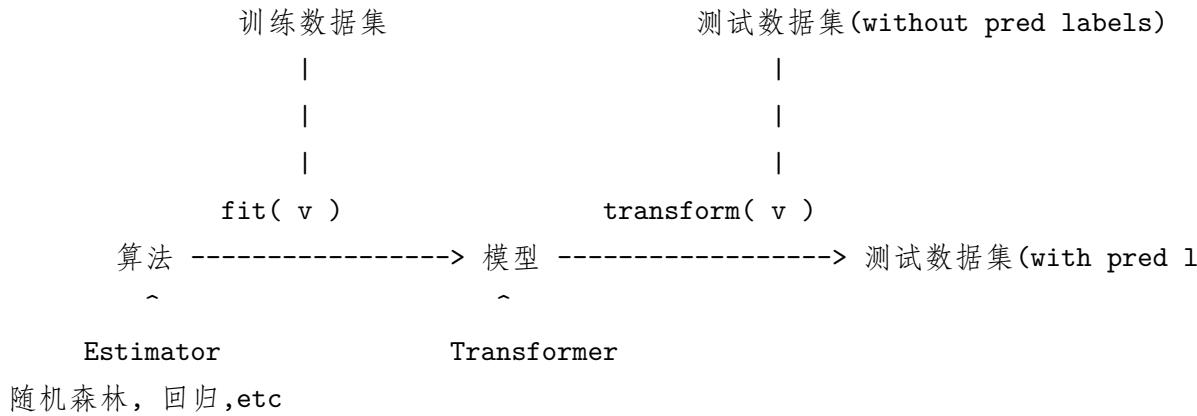
data	true label
x	1
y	0
z	0

DataFrame-2

机器学习工作流概念

Transformer

翻译成转换器，是一种可以将一个DataFrame转换为另一个DataFrame的算法。比如一个模型就是一个 Transformer。它可以把一个不包含预测标签的测试数据集 DataFrame 打上标签，转化成另一个包含预测标签的 DataFrame。技术上，Transformer实现了一个方法transform()，它通过附加一个或多个列将一个DataFrame转换为另一个DataFrame。



机器学习工作流概念

Estimator

翻译成估计器或评估器，它是学习算法或在训练数据上的训练方法的概念抽象。在 Pipeline 里通常是被用来操作 DataFrame 数据并生成一个 Transformer。从技术上讲，Estimator 实现了一个方法 fit()，它接受一个 DataFrame 并产生一个转换器。比如，一个随机森林算法就是一个 Estimator 它可以调用 fit()，通过训练特征数据而得到一个随机森林模型

Estimator, 就可以理解为一个 算法, 调用点 fit()

机器学习工作流概念



机器学习工作流概念

- 要构建一个 Pipeline 工作流，首先需要定义 Pipeline 中的各个工作流阶段 PipelineStage（包括转换器和评估器），比如指标提取和转换模型训练等。有了这些处理特定问题的转换器和评估器，就可以按照具体的处理逻辑有序地组织 PipelineStages 并创建一个 Pipeline

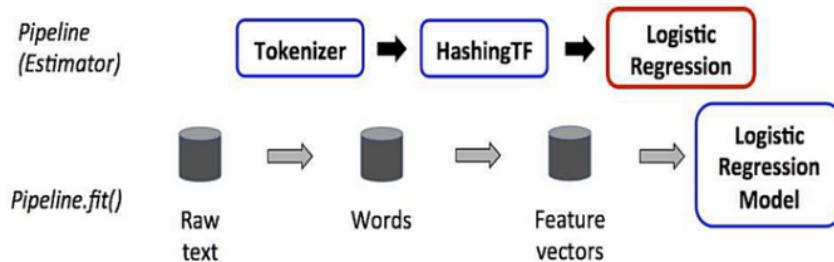
```
val pipeline = new
Pipeline().setStages(Array(stage1,stage2,stage3,...))
```

- 然后就可以把训练数据集作为输入参数，调用 Pipeline 实例的 fit 方法来开始以流的方式来处理源训练数据。这个调用会返回一个 PipelineModel 类实例，进而被用来预测测试数据的标签

Pipeline.fit() —> Pipeline Model

机器学习工作流概念

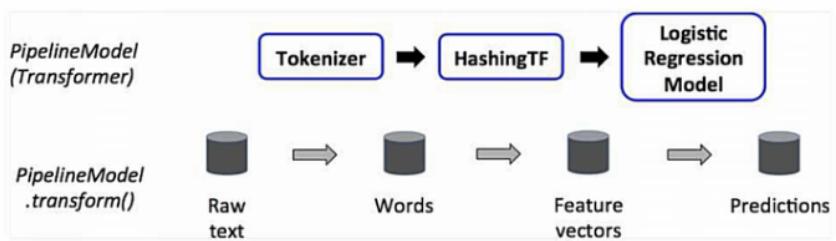
- 工作流的各个阶段按顺序运行，输入的 DataFrame 在它通过每个阶段时被转换



蓝色是 Estimator(各种模型), 红色是 Transformer(各种算法)

机器学习工作流概念

值得注意的是，工作流本身也可以看做是一个估计器。在工作流的fit（）方法运行之后，它产生一个PipelineModel，它是一个Transformer。这个管道模型将在测试数据的时候使用。下图说明了这种用法



2.2 工作流实例

构建一个机器学习工作流

本节以逻辑斯蒂回归为例，构建一个典型的机器学习过程，来具体介绍一下工作流是如何应用的



任务描述

找出所有包含"spark"的句子，即将包含"spark"的句子的标签设为1，没有"spark"的句子的标签设为0

构建一个机器学习工作流

➤ 需要使用SparkSession对象需要使用SparkSession对象

➤ Spark2.0以上版本的spark-shell在启动时会自动创建一个名为spark的SparkSession对象，当需要手工创建时，SparkSession可以由其伴生对象的builder()方法创建出来，如下代码段所示：

```
import org.apache.spark.sql.SparkSession  
val spark = SparkSession.builder()  
.master("local").  
.appName("my App Name").  
.getOrCreate()
```



(1) 引入要包含的包并构建训练数据集

```
import org.apache.spark.ml.feature._  
import org.apache.spark.ml.classification.LogisticRegression  
import org.apache.spark.ml.Pipeline  
import org.apache.spark.ml.linalg.Vector  
import org.apache.spark.sql.Row  
  
scala> 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")  
training: org.apache.spark.sql.DataFrame = [id: bigint, text: string, label: double]
```

构建训练数据集 DataFrame.

```
sparkSession.createDataFrame( 训练集:seq of vectorized sample ) // 创建无字段名 DataFrame  
.toDF( 字段名 ) // 创建带字段名的 DataFrame.
```

(2) 定义 Pipeline 中的各个工作流阶段 PipelineStage，包括转换器和评估器，具体地，包含 tokenizer, hashingTF 和 lr

```
scala> val tokenizer = new Tokenizer().  
| setInputCol("text").  
| setOutputCol("words")  
tokenizer: org.apache.spark.ml.feature.Tokenizer = tok_5151ed4fa43e  
  
scala> val hashingTF = new HashingTF().  
| setNumFeatures(1000).  
| setInputCol(tokenizer.getOutputCol).  
| setOutputCol("features")  
hashingTF: org.apache.spark.ml.feature.HashingTF = hashingTF_332f74b21ecb  
  
scala> val lr = new LogisticRegression().  
| setMaxIter(10).  
| setRegParam(0.01)  
lr: org.apache.spark.ml.classification.LogisticRegression = logreg_28a670ae952f
```

1. Tokenizer():

- setInputCol("要做 tokenize 的列名")
- setOutputCol("做完 tokenize 后输出的列名")

2. hashingTF: 把单词映射为向量

- setNumFeatures("hash 桶数量, 词袋中单词书目")
- setInputCol("要做 tokenize 的列名")
- setOutputCol("做完 tokenize 后输出的列名")

3. LogisticRegression():

- setMaxIter(最大循环次数)
- setRegParam(0.01)



(3) 按照具体的处理逻辑有序地组织PipelineStages，并创建一个Pipeline

```
scala> val pipeline = new Pipeline().  
| setStages(Array(tokenizer, hashingTF, lr))  
pipeline: org.apache.spark.ml.Pipeline = pipeline_4dabd24db001
```

现在构建的Pipeline本质上是一个Estimator，在它的fit()方法运行之后，它将产生一个PipelineModel，它是一个Transformer。

```
scala> val model = pipeline.fit(training)  
model: org.apache.spark.ml.PipelineModel = pipeline_4dabd24db001
```

可以看到，model的类型是一个PipelineModel，这个工作流模型将在测试数据的时候使用

整个 Pipeline 还没有 fit 数据时，是一个 Estimator, fit 数据之后就变成了一个 Transformer. 可以进行数据预测 (transform).

Estimator ----- .fit ---> Transformer --- .transform --->



(4) 构建测试数据

```
scala> val test = spark.createDataFrame(Seq(  
| (4L, "spark i j k"),  
| (5L, "l m n"),  
| (6L, "spark a"),  
| (7L, "apache hadoop")  
| )).toDF("id", "text")  
test: org.apache.spark.sql.DataFrame = [id: bigint, text: string]
```

注意，这里 Seq 里面是一个个的元组 tuple, 如果不是 tuple 需要调用 seq(xxx).map(Tuple1.apply) 进行转换，成为 seq of tuples, one sample one tuple, 之后才能够通过 .toDF 转换成 DataFrame.



(5) 调用之前训练好的PipelineModel的transform()方法，让测试数据按顺序通过拟合的工作流，生成预测结果

```
scala> 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")
| }
| (4, spark i j k) -> prob=[0.5406433544851421,0.45935664551485783], prediction=0.0
| (5, l m n) -> prob=[0.9334382627383259,0.06656173726167405], prediction=0.0
| (6, spark a) -> prob=[0.15041430048068286,0.8495856995193171], prediction=1.0
| (7, apache hadoop) -> prob=[0.9768636139518304,0.023136386048169585], prediction=0.0
```

```
model.transform(test).select("id", "text", "prob", "pred") // DataFrame 可以直接使用
               .collect()
// 收集到本地
               .foreach{
                 case Row(id:Long, text:String, prob:Vector,
                          print(s"($id, $text)-->prob=$prob, prediction=$prediction")
                } // 每个 transformer 产生的都是 Row 集合，我
```

Tokenizer 和 hashingTF 本身就是一个 Transformer, 所以他不需要 fit 数据就可以直接 transfor data points.

3 特征抽取, 转化, 选择

3.1 IF-IDF

TF-IDF 方法, 用于给单词进行向量化的方法.

特征抽取：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

➤ TF-IDF就是在数值化文档信息，衡量词语能提供多少信息以区分文档，其定义如下

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

➤ TF-IDF 度量值表示如下

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

➤ 在Spark ML库中，TF-IDF被分成两部分



过程描述

-  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），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）

(1) 导入TF-IDF所需要的包

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

开启RDD的隐式转换

```
import spark.implicits._
```

(2) 创建一个简单的DataFrame，每一个句子代表一个文档

```
scala> val sentenceData = spark.createDataFrame(Seq(  
|   (0, "I heard about Spark and I love Spark"),  
|   (0, "I wish Java could use case classes"),  
|   (1, "Logistic regression models are neat")  
| ).toDF("label", "sentence")  
sentenceData: org.apache.spark.sql.DataFrame = [label: int, sentence:  
string]
```

(3) 得到文档集合后，即可用tokenizer对句子进行分词

```

scala> val tokenizer = new
Tokenizer().setInputCol("sentence").setOutputCol("words")
tokenizer: org.apache.spark.ml.feature.Tokenizer = tok_494411a37f99

scala> val wordsData = tokenizer.transform(sentenceData)
wordsData: org.apache.spark.sql.DataFrame = [label: int, sentence: string, words: array<string>]

scala> wordsData.show(false)
+---+-----+-----+
|label|sentence |words |
+---+-----+-----+
|0 |I heard about Spark and I love Spark|[i, heard, about, spark, and, i, love, spark]|
|0 |I wish Java could use case classes |[i, wish, java, could, use, case, classes] |
|1 |Logistic regression models are neat |[logistic, regression, models, are, neat] |
+---+-----+-----+

```

(4) 得到分词后的文档序列后，即可使用HashingTF的transform()方法把句子哈希成特征向量，这里设置哈希表的桶数为2000

```

scala> val hashingTF = new HashingTF().
    | setInputCol("words").setOutputCol("rawFeatures").setNumFeatures(2000)
hashingTF: org.apache.spark.ml.feature.HashingTF = hashingTF_2591ec73cea0
scala> val featurizedData = hashingTF.transform(wordsData)
featurizedData: org.apache.spark.sql.DataFrame = [label: int, sentence: string,
words: array<string>, rawFeatures: vector]

scala> featurizedData.select("rawFeatures").show(false)
+-----+
|rawFeatures |
+-----+
|(2000,[240,333,1105,1329,1357,1777],[1.0,1.0,2.0,2.0,1.0,1.0]) |
|(2000,[213,342,489,495,1329,1809,1967],[1.0,1.0,1.0,1.0,1.0,1.0,1.0])|
|(2000,[286,695,1138,1193,1604],[1.0,1.0,1.0,1.0,1.0]) |
+-----+

```

注意，经过某些 transformer 之后，得到的新列，是有自己特殊格式的，比如对于 HashingTF 来说，得到的新列"rawFeatures" 的格式就是一个 3-Tuple:

(词袋数量，向量化的句子，该句子中每个单词的权重)

```

| rawFeatures
|-----

```

```
| (2000, [240, 333, 1105, 1329, 135, 1777], [1.0, 2.0, 2.0, 2.0, 1.0, 1.0])
```

词袋中单词数量,	向量化的句子,	单词权重
5	[240, 333, 1105, 1329, 135, 1777]	[1.0, 2.0, 2.0, 2.0, 1.0, 1.0]

(5) 使用IDF来对单纯的词频特征向量进行修正, 使其更能体现不同词汇对文本的区别能力

```
scala> val idf = new IDF().  
|   setInputCol("rawFeatures").setOutputCol("features")  
idf: org.apache.spark.ml.feature.IDF = idf_7fcc9063de6f  
  
scala> val idfModel = idf.fit(featurizedData)  
idfModel: org.apache.spark.ml.feature.IDFModel = idf_7fcc9063de6f
```

IDF是一个Estimator, 调用fit()方法并将词频向量传入, 即产生一个IDFModel

➤ IDFModel是一个Transformer, 调用它的transform()方法, 即可得到每一个单词对应的TF-IDF度量值

```
scala> val rescaledData = idfModel.transform(featurizedData)  
rescaledData: org.apache.spark.sql.DataFrame = [label: int, sentence: string,  
words: array<string>, rawFeatures: vector, features: vector]  
  
scala> rescaledData.select("features", "label").take(3).foreach(println)  
[(2000,[240,333,1105,1329,135,1777],[0.6931471805599453,0.6931471805599453,  
53,1.3862943611198906,0.5753641449035617,0.6931471805599453,0.6931471805599453]),0]  
[(2000,[213,342,489,495,1329,1809,1967],[0.6931471805599453,0.6931471805599453,  
0.6931471805599453,0.6931471805599453,0.6931471805599453,0.28768207245178085,0.693  
1471805599453,0.6931471805599453]),0]  
[(2000,[286,695,1138,1193,1604],[0.6931471805599453,0.6931471805599453,0.  
6931471805599453,0.6931471805599453,0.6931471805599453]),1]
```

3.2 word2vec

01

Word2Vec是一种著名的词嵌入 (Word Embedding) 方法，它可以计算每个单词在其给定语料库环境下的分布式词向量

02

词向量表示可以在一定程度上刻画每个单词的语义

03

如果词的语义相近，它们的词向量在向量空间中也相互接近

任务描述

一组文档，其中一个词语序列代表一个文档。

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



(1) 首先导入Word2Vec所需要的包，并创建三个词语序列，每个代表一个文档

```
scala> import org.apache.spark.ml.feature.Word2Vec  
scala> 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: org.apache.spark.sql.DataFrame = [text: array<string>]
```

.map(Tuple1.apply)

"Hi I heard about spank".split("") 生成的是一个数组: array("Hi", "I", "heard", xxx)

DataFrame 需要的是一个元组 tuple, 所以调用这个方法可以把 array==>tuple.



(2) 新建一个Word2Vec，显然，它是一个Estimator，设置相应的超参数，这里设置特征向量的维度为3

```
scala> val word2Vec = new Word2Vec().  
|   setInputCol("text").  
|   setOutputCol("result").  
|   setVectorSize(3).  
|   setMinCount(0)  
word2Vec: org.apache.spark.ml.feature.Word2Vec = w2v_e2d5128ba199
```

- setInputCol
- setOutputCol
- setVectorSize(3) // Embedding 空间维度
- setMinCount(0) //



(3) 读入训练数据, 用fit()方法生成一个Word2VecModel

```
scala> val model = word2Vec.fit(documentDF)
model: org.apache.spark.ml.feature.Word2VecModel = w2v_e2d5128ba199
```



(4) 利用Word2VecModel把文档转变成特征向量

```
scala> val result = model.transform(documentDF)
result: org.apache.spark.sql.DataFrame = [text: array<string>, result: vector]

scala> result.select("result").take(3).foreach(println)
[[0.018490654602646827,-0.016248732805252075,0.04528368394821883]]
[[0.05958533100783825,0.023424440695505054,-0.027310076036623544]]
[[-0.011055880039930344,0.020988055132329465,0.042608972638845444]]
```

3.3 CountVectorizer

CountVectorizer

- CountVectorizer旨在通过计数来将一个文档转换为向量
- 当不存在先验字典时, Countvectorizer作为Estimator提取词汇进行训练并生成一个CountVectorizerModel用于存储相应的词汇向量空间

CountVectorizer

- 该模型产生文档关于词语的稀疏表示, 其表示可以传递给其他算法, 例如LDA
- 在CountVectorizerModel的训练过程中, CountVectorizer将根据语料库中的词频排序从高到低进行选择, 词汇表的最大含量由vocabsize超参数来指定超参数minDF, 则指定词汇表中的词语至少要在多少个不同文档中出现

(1) 首先导入CountVectorizer所需要的包

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

(2) 假设有如下的DataFrame , 其包含id和words两列 , 可以看成是一个包含两个文档的迷你语料库

```
scala> val df = spark.createDataFrame(Seq(  
| (0, Array("a", "b", "c")),  
| (1, Array("a", "b", "b", "c", "a"))  
| )).toDF("id", "words")  
df: org.apache.spark.sql.DataFrame = [id: int, words: array<string>]
```

(3) 通过CountVectorizer设定超参数 , 训练一个CountVectorizerModel
这里设定词汇表的最大量为3 , 设定词汇表中的词至少要在2个文档中出现过
以过滤那些偶然出现的词汇

```
scala> val cvModel: CountVectorizerModel = new CountVectorizer().  
|   setInputCol("words").  
|   setOutputCol("features").  
|   setVocabSize(3).  
|   setMinDF(2).  
|   fit(df)  
cvModel: org.apache.spark.ml.feature.CountVectorizerModel =  
cntVec_237a080886a2
```

(4) 在训练结束后 , 可以通过CountVectorizerModel的vocabulary成员
获得得到模型的词汇表

```
scala> cvModel.vocabulary  
res7: Array[String] = Array(b, a, c)
```

从打印结果我们可以看到 , 词汇表中有 “a” , “b” , “c” 三个词 , 且这
三个词都在2个文档中出现过 (前文设定了minDF为2)

(5) 使用这一模型对DataFrame进行变换，可以得到文档的向量化表示

```
scala> cvModel.transform(df).show(false)
+---+-----+-----+
|id |words |features |
+---+-----+-----+
|0 |[a, b, c] |(3,[0,1,2],[1.0,1.0,1.0])|
|1 |[a, b, b, c, a]|(3,[0,1,2],[2.0,2.0,1.0])|
+---+-----+-----+
```

➤ 和其他Transformer不同，CountVectorizerModel可以通过指定一个先验词汇表来直接生成，如以下例子，直接指定词汇表的成员是“a”，“b”，“c”三个词

```
scala> val cvm = new CountVectorizerModel(Array("a", "b", "c")).  
| setInputCol("words").  
| setOutputCol("features")  
cvm: org.apache.spark.ml.feature.CountVectorizerModel =  
cntVecModel_c6a17c2befee  
  
scala> cvm.transform(df).select("features").foreach { println }  
[(3,[0,1,2],[1.0,1.0,1.0])]  
[(3,[0,1,2],[2.0,2.0,1.0])]
```

4 分类与回归

4.1 logistic regression 分类器

逻辑斯蒂回归

逻辑斯蒂回归（logistic regression）是统计学习中的经典分类方法，属于对数线性模型。logistic回归的因变量可以是二分类的，也可以是多分类的

任务描述

我们以iris数据集（iris）为例进行分析（iris下载地址：<http://dblab.xmu.edu.cn/blog/wp-content/uploads/2017/03/iris.txt>）。iris以鸢尾花的特征作为数据来源，数据集包含150个数据集，分为3类，每类50个数据，每个数据包含4个属性，是在数据挖掘、数据分类中非常常用的测试集、训练集。为了便于理解，我们这里主要用后两个属性（花瓣的长度和宽度）来进行分类。目前spark.ml中支持二分类和多分类，我们将分别从“用二项逻辑斯蒂回归来解决二分类问题”、“用多项逻辑斯蒂回归来解决二分类问题”、“用多项逻辑斯蒂回归来解决多分类问题”三个方面进行分析

➤ 首先我们先取其中的后两类数据，用二项逻辑斯蒂回归进行二分类分析

1. 导入需要的包

```
import org.apache.spark.sql.Row
import org.apache.spark.sql.SparkSession
import org.apache.spark.ml.linalg.{Vector, Vectors}
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator
import org.apache.spark.ml.Pipeline, PipelineModel
import org.apache.spark.ml.feature.{IndexToString, StringIndexer,
VectorIndexer, HashingTF, Tokenizer}
import org.apache.spark.ml.classification.LogisticRegression
import org.apache.spark.ml.classification.LogisticRegressionModel
import org.apache.spark.ml.classification.{BinaryLogisticRegressionSummary,
LogisticRegression}
import org.apache.spark.sql.functions;
```

2. 读取数据，简要分析

```
scala> import spark.implicits._
import spark.implicits._

scala> case class Iris(features: org.apache.spark.ml.linalg.Vector, label: String)
defined class Iris

scala> val data = spark.sparkContext.textFile("file:///usr/local/spark/iris.txt").
| map(_.split(","))
| map(p => Iris(Vectors.dense(p(0).toDouble,p(1).toDouble,p(2).toDouble,
p(3).toDouble), p(4).toString())))
| toDF()
data: org.apache.spark.sql.DataFrame = [features: vector, label: string]
```

```
scala> data.show()
+-----+-----+
| features| label|
+-----+-----+
|[5.1,3.5,1.4,0.2]|Iris-setosa|
|[4.9,3.0,1.4,0.2]|Iris-setosa|
|[4.7,3.2,1.3,0.2]|Iris-setosa|
|[4.6,3.1,1.5,0.2]|Iris-setosa|
|[5.0,3.6,1.4,0.2]|Iris-setosa|
|[5.4,3.9,1.7,0.4]|Iris-setosa|
|[4.6,3.4,1.4,0.3]|Iris-setosa|
|[5.0,3.4,1.5,0.2]|Iris-setosa|
|[4.4,2.9,1.4,0.2]|Iris-setosa|
|[4.9,3.1,1.5,0.1]|Iris-setosa|
|[5.4,3.7,1.5,0.2]|Iris-setosa|
|[4.8,3.4,1.6,0.2]|Iris-setosa|
|[4.8,3.0,1.4,0.1]|Iris-setosa|
|[4.3,3.0,1.1,0.1]|Iris-setosa|
|[5.8,4.0,1.2,0.2]|Iris-setosa|
|[5.7,4.4,1.5,0.4]|Iris-setosa|
|[5.4,3.9,1.3,0.4]|Iris-setosa|
|[5.1,3.5,1.4,0.3]|Iris-setosa|
|[5.7,3.8,1.7,0.3]|Iris-setosa|
|[5.1,3.8,1.5,0.3]|Iris-setosa|
+-----+-----+
only showing top 20 rows
```

➤ 因为我们现在处理的是2分类问题，所以我们不需要全部的3类数据，我们要从中选出两类的数据

➤ 首先把刚刚得到的数据注册成一个表iris，注册成这个表之后，我们就可以通过sql语句进行数据查询

```
scala> data.createOrReplaceTempView("iris")  
  
scala> val df = spark.sql("select * from iris where label != 'Iris-setosa'")  
df: org.apache.spark.sql.DataFrame = [features: vector, label: string]  
  
scala> df.map(t => t(1)+":" +t(0)).collect().foreach(println)  
Iris-versicolor:[7.0,3.2,4.7,1.4]  
Iris-versicolor:[6.4,3.2,4.5,1.5]  
Iris-versicolor:[6.9,3.1,4.9,1.5]  
.....  
.....
```

3. 构建ML的pipeline

(1) 分别获取标签列和特征列，进行索引，并进行了重命名

```
scala> val labelIndexer = new  
StringIndexer().setInputCol("label").setOutputCol("indexedLabel").fit(df)  
labelIndexer: org.apache.spark.ml.feature.StringIndexerModel =  
strIdx_e53e67411169  
scala> val featureIndexer = new VectorIndexer().  
| setInputCol("features").setOutputCol("indexedFeatures").fit(df)  
featureIndexer: org.apache.spark.ml.feature.VectorIndexerModel =  
vecIdx_53b988077b38
```

(2) 接下来，我们把数据集随机分成训练集和测试集，其中训练集占70%

```
scala> val Array(trainingData, testData) = df.randomSplit(Array(0.7, 0.3))
trainingData: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] =
[features: vector, label: string]
testData: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] =
[features: vector, label: string]
```

(3) 然后，我们设置logistic的参数，这里我们统一用setter的方法来设置，也可以用ParamMap来设置（具体的可以查看spark mllib的官网）。这里我们设置了循环次数为10次，正则化项为0.3等

```
scala> val lr = new LogisticRegression().setLabelCol("indexedLabel").
|   setFeaturesCol("indexedFeatures").setMaxIter(10).
|   setRegParam(0.3).setElasticNetParam(0.8)
lr: org.apache.spark.ml.classification.LogisticRegression = logreg_692899496c23
```

(4) 这里我们设置一个labelConverter，目的是把预测的类别重新转化成字符型的

```
scala> val labelConverter = new IndexToString().
|   setInputCol("prediction").setOutputCol("predictedLabel").
|   setLabels(labelIndexer.labels)
labelConverter: org.apache.spark.ml.feature.IndexToString =
idxToStr_c204eafabf57
```

(5) 构建pipeline，设置stage，然后调用fit()来训练模型

```
scala> val lrPipeline = new Pipeline().
|   setStages(Array(labelIndexer, featureIndexer, lr, labelConverter))
lrPipeline: org.apache.spark.ml.Pipeline = pipeline_eb1b201af1e0

scala> val lrPipelineModel = lrPipeline.fit(trainingData)
lrPipelineModel: org.apache.spark.ml.PipelineModel =
pipeline_eb1b201af1e0
```

(6) pipeline本质上是一个Estimator，当pipeline调用fit()的时候就产生了一个PipelineModel，本质上是一个Transformer。然后这个PipelineModel就可以调用transform()来进行预测，生成一个新的DataFrame，即利用训练得到的模型对测试集进行验证

```
scala> val lrPredictions = lrPipelineModel.transform(testData)
lrPredictions: org.apache.spark.sql.DataFrame = [features: vector,
label: string ... 6 more fields]
```

(7) 最后我们可以输出预测的结果，其中select选择要输出的列，collect获取所有行的数据，用foreach把每行打印出来。其中打印出来的值依次分别代表该行数据的真实分类和特征值、预测属于不同分类的概率、预测的分类

```
scala> lrPredictions.select("predictedLabel", "label", "features", "probability").  
| collect().  
| foreach { case Row(predictedLabel: String, label: String, features: Vector,  
prob: Vector) => println(s"($label, $features) --> prob=$prob, predicted  
Label=$predictedLabel")}  
  
(Iris-virginica, [4.9,2.5,4.5,1.7]) -->  
prob=[0.4796551461409372,0.5203448538590628], predictedLabel=Iris-virginica  
(Iris-versicolor, [5.1,2.5,3.0,1.1]) -->  
prob=[0.5892626391059901,0.41073736089401], predictedLabel=Iris-versicolor  
(Iris-versicolor, [5.5,2.3,4.0,1.3]) -->  
prob=[0.5577310241453046,0.4422689758546954], predictedLabel=Iris-versicolor
```

4. 模型评估

创建一个MulticlassClassificationEvaluator实例，用setter方法把预测分类的列名和真实分类的列名进行设置；然后计算预测准确率和错误率

```
scala> val evaluator = new MulticlassClassificationEvaluator().  
| setLabelCol("indexedLabel").setPredictionCol("prediction")  
evaluator: org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator =  
mcEval_a80353e4211d  
  
scala> val lrAccuracy = evaluator.evaluate(lrPredictions)  
lrAccuracy: Double = 1.0  
  
scala> println("Test Error = " + (1.0 - lrAccuracy))  
Test Error = 0.0
```

➤ 从上面可以看到预测的准确性达到100%

接下来我们可以通过`model`来获取我们训练得到的逻辑斯蒂模型。前面已经说过`model`是一个`PipelineModel`，因此我们可以通过调用它的`stages`来获取模型，具体如下

```
scala> val lrModel = lrPipelineModel.stages(2).
|   asInstanceOf[LogisticRegressionModel]
lrModel: org.apache.spark.ml.classification.LogisticRegressionModel =
logreg_692899496c23

scala> println("Coefficients: " + lrModel.coefficients+"Intercept:
"+lrModel.intercept+"numClasses: "+lrModel.numClasses+"numFeatures:
"+lrModel.numFeatures)
Coefficients: [-0.0396171957643483,0.0,0.0,0.07240315639651046]Intercept:
-0.23127346342015379numClasses: 2numFeatures: 4
```

4.2 决策树分类器

决策树 (decision tree)

决策树 (decision tree) 是一种基本的分类与回归方法，这里主要介绍用于分类的决策树。决策树模式呈树形结构，其中每个内部节点表示一个属性上的测试，每个分支代表一个测试输出，每个叶节点代表一种类别。学习时利用训练数据，根据损失函数最小化的原则建立决策树模型；预测时，对新的数据，利用决策树模型进行分类。

决策树学习通常包括3个步骤



(一) 特征选择

特征选择在于选取对训练数据具有分类能力的特征，这样可以提高决策树学习的效率。通常特征选择的准则是信息增益（或信息增益比、基尼指数等），每次计算每个特征的信息增益，并比较它们的大小，选择信息增益最大（信息增益比最大、基尼指数最小）的特征。

(二) 决策树的生成

- 从根结点开始，对结点计算所有可能的特征的信息增益，选择信息增益最大的特征作为结点的特征，由该特征的不同取值建立子结点，再对子结点递归地调用以上方法，构建决策树；直到所有特征的信息增益很小或没有特征可以选择为止，最后得到一个决策树
- 决策树需要有停止条件来终止其生长的过程。一般来说最低的条件是：当该节点下面的所有记录都属于同一类，或者当所有的记录属性都具有相同的值时。这两种条件是停止决策树的必要条件，也是最低的条件。在实际运用中一般希望决策树提前停止生长，限定叶节点包含的最低数据量，以防止由于过度生长造成的过拟合问题

(三) 决策树的剪枝

决策树生成算法递归地产生决策树，直到不能继续下去为止。这样产生的树往往对训练数据的分类很准确，但对未知的测试数据的分类却没有那么准确，即出现过拟合现象。解决这个问题的办法是考虑决策树的复杂度，对已生成的决策树进行简化，这个过程称为剪枝

我们以iris数据集（iris）为例进行分析（iris下载地址：

<http://dblab.xmu.edu.cn/blog/wp-content/uploads/2017/03/iris.txt>
iris以鸢尾花的特征作为数据来源，数据集包含150个数据集，分为3类，每类50个数据，每个数据包含4个属性，是在数据挖掘、数据分类中非常常用的测试集、训练集

1 导入需要的包

```
import org.apache.spark.sql.SparkSession  
import org.apache.spark.ml.linalg.{Vector, Vectors}  
import org.apache.spark.ml.Pipeline  
import org.apache.spark.ml.feature.{IndexToString, StringIndexer,  
VectorIndexer}
```

2 读取数据，简要分析

```
scala> import spark.implicits._  
import spark.implicits._  
  
scala> case class Iris(features: org.apache.spark.ml.linalg.Vector, label: String)  
defined class Iris  
  
scala> val data = spark.sparkContext.  
|   textFile("file:///usr/local/spark/iris.txt").map(_.split(",")).  
|   map(p =>  
Iris(Vectors.dense(p(0).toDouble,p(1).toDouble,p(2).toDouble,p(3).toDouble),p(4).  
toString())).  
|   toDF()  
data: org.apache.spark.sql.DataFrame = [features: vector, label: string]
```

剩余代码见下一页

```
scala> data.createOrReplaceTempView("iris")  
  
scala> val df = spark.sql("select * from iris")  
df: org.apache.spark.sql.DataFrame = [features: vector, label: string]  
  
scala> df.map(t => t(1)+":" +t(0)).collect().foreach(println)  
Iris-setosa:[5.1,3.5,1.4,0.2]  
Iris-setosa:[4.9,3.0,1.4,0.2]  
Iris-setosa:[4.7,3.2,1.3,0.2]  
Iris-setosa:[4.6,3.1,1.5,0.2]  
Iris-setosa:[5.0,3.6,1.4,0.2]  
Iris-setosa:[5.4,3.9,1.7,0.4]  
Iris-setosa:[4.6,3.4,1.4,0.3]  
  
....
```

3 进一步处理特征和标签，以及数据分组

```
//分别获取标签列和特征列，进行索引，并进行了重命名。  
scala> val labelIndexer = new StringIndexer().setInputCol("label").  
| setOutputCol("indexedLabel").fit(df)  
labelIndexer: org.apache.spark.ml.feature.StringIndexerModel =  
strIdx_107f7e530fa7  
  
scala> val featureIndexer = new VectorIndexer().setInputCol("features").  
| setOutputCol("indexedFeatures").setMaxCategories(4).fit(df)  
featureIndexer: org.apache.spark.ml.feature.VectorIndexerModel =  
vecIdx_0649803dfa70
```

剩余代码见下一页

```
//这里我们设置一个labelConverter，目的是把预测的类别重新转化成字符串型的。  
scala> val labelConverter = new IndexToString().setInputCol("prediction").  
| setOutputCol("predictedLabel").setLabels(labelIndexer.labels)  
labelConverter: org.apache.spark.ml.feature.IndexToString =  
idxToStr_046182b2e571  
//接下来，我们把数据集随机分成训练集和测试集，其中训练集占70%。  
scala> val Array(trainingData, testData) = data.randomSplit(Array(0.7, 0.3))  
trainingData: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [features:  
vector, label: string]  
testData: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [features:  
vector, label: string]
```

4 构建决策树分类模型

```
//导入所需要的包  
import org.apache.spark.ml.classification.DecisionTreeClassificationModel  
import org.apache.spark.ml.classification.DecisionTreeClassifier  
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator  
//训练决策树模型，这里我们可以通过setter的方法来设置决策树的参数，也可以用ParamMap来  
设置（具体的可以查看spark mllib的官网）。具体的可以设置的参数可以通过explainParams()  
来获取。  
scala> val dtClassifier = new DecisionTreeClassifier().  
| setLabelCol("indexedLabel").  
| setFeaturesCol("indexedFeatures")  
dtClassifier: org.apache.spark.ml.classification.DecisionTreeClassifier = dtc_029ea28aceb1  
//在pipeline中进行设置  
scala> val pipelinedClassifier = new Pipeline().  
| setStages(Array(labelIndexer, featureIndexer, dtClassifier, labelConverter))  
pipelinedClassifier: org.apache.spark.ml.Pipeline = pipeline_a254dfd6dfb9
```

```
//训练决策树模型
scala> val modelClassifier = pipelinedClassifier.fit(trainingData)
modelClassifier: org.apache.spark.ml.PipelineModel = pipeline_a254dfd6dfb9
//进行预测
scala> val predictionsClassifier = modelClassifier.transform(testData)
predictionsClassifier: org.apache.spark.sql.DataFrame = [features: vector, label: string ... 6 more fields]
//查看部分预测的结果
scala> predictionsClassifier.select("predictedLabel", "label", "features").show(20)
+-----+-----+-----+
| predictedLabel| label| features|
+-----+-----+-----+
| Iris-setosa| Iris-setosa|[4.4,2.9,1.4,0.2]|
| Iris-setosa| Iris-setosa|[4.6,3.6,1.0,0.2]|
| Iris-virginica| Iris-versicolor|[4.9,2.4,3.3,1.0]|
| Iris-setosa| Iris-setosa|[4.9,3.1,1.5,0.1]|
| Iris-setosa| Iris-setosa|[4.9,3.1,1.5,0.1]|
```

5 评估决策树分类模型

```
scala> val evaluatorClassifier = new MulticlassClassificationEvaluator().
  | setLabelCol("indexedLabel").
  | setPredictionCol("prediction").setMetricName("accuracy")
evaluatorClassifier: org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator =
mcEval_4abc19f3a54d

scala> val accuracy = evaluatorClassifier.evaluate(predictionsClassifier)
accuracy: Double = 0.8648648648648649

scala> println("Test Error = " + (1.0 - accuracy))
Test Error = 0.1351351351351351

scala> val treeModelClassifier = modelClassifier.stages(2).
  | asInstanceOf[DecisionTreeClassificationModel]
treeModelClassifier: org.apache.spark.ml.classification.DecisionTreeClassificati
onModel = DecisionTreeClassificationModel (uid=dtc_029ea28aceb1) of depth 5 with 13 nodes
```

```
scala> println("Learned classification tree model:\n" + treeModelClassifier.toDebugString)
Learned classification tree model:
DecisionTreeClassificationModel (uid=dtc_029ea28aceb1) of depth 5 with 13 nodes
If (feature 2 <= 1.9)
Predict: 2.0
Else (feature 2 > 1.9)
If (feature 2 <= 4.7)
If (feature 0 <= 4.9)
Predict: 1.0
Else (feature 0 > 4.9)
Predict: 0.0
Else (feature 2 > 4.7)
If (feature 3 <= 1.6)
If (feature 2 <= 4.8)
Predict: 0.0
Else (feature 2 > 4.8)
If (feature 0 <= 6.0)
Predict: 0.0
Else (feature 0 > 6.0)
Predict: 1.0
Else (feature 3 > 1.6)
Predict: 1.0
```