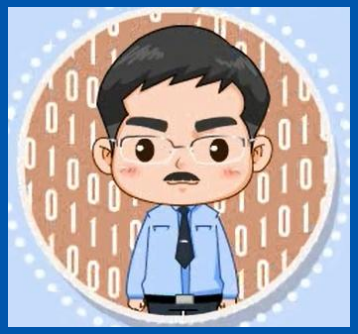


《Spark编程基础》

教材官网： http://dblab.xmu.edu.cn/post/spark/

温馨提示：编辑幻灯片母版，可以修改每页PPT的厦大校徽和底部文字

第8章 Spark MLlib

（PPT版本号：2018年2月）

扫一扫访问教材官网

林子雨

厦门大学计算机科学系

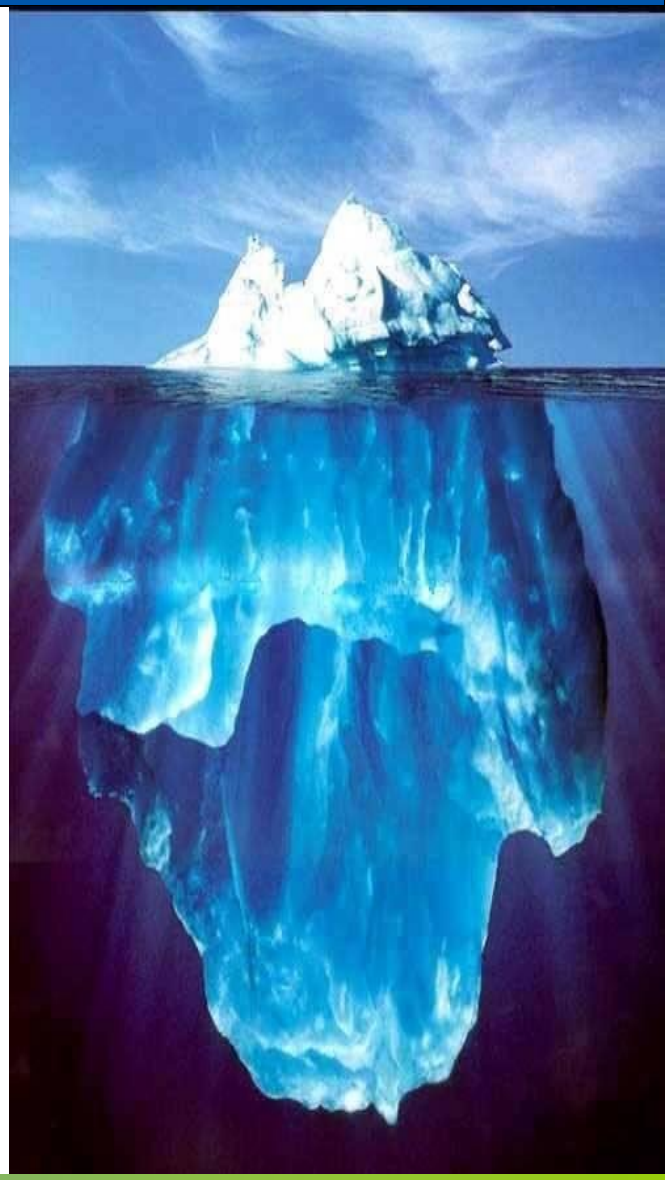
E-mail: ziyulin@xmu.edu.cn

主页：http://www.cs.xmu.edu.cn/linziyu

课程配套授课视频

课程在线视频地址：http://dblab.xmu.edu.cn/post/10482/

提纲

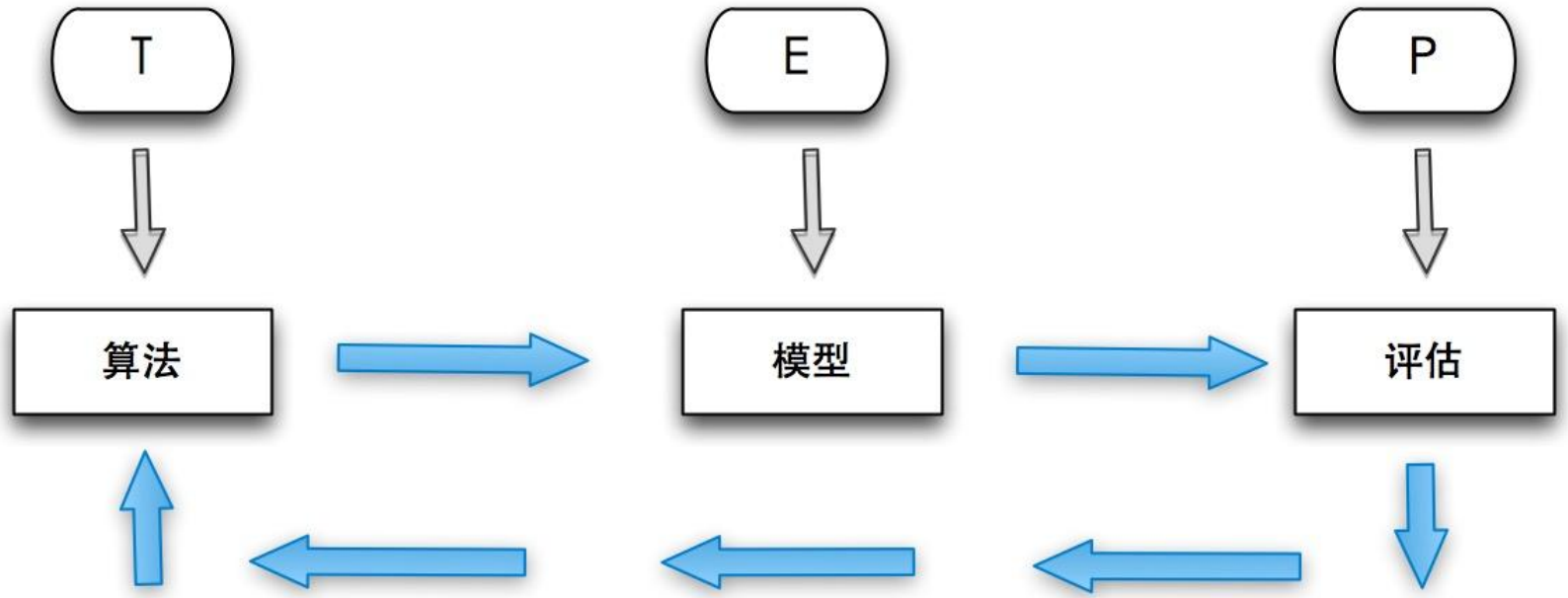


8.1 Spark MLlib简介

8.1.1 什么是机器学习
8.1.2 基于大数据的机器学习
8.1.3 Spark 机器学习库MLLib

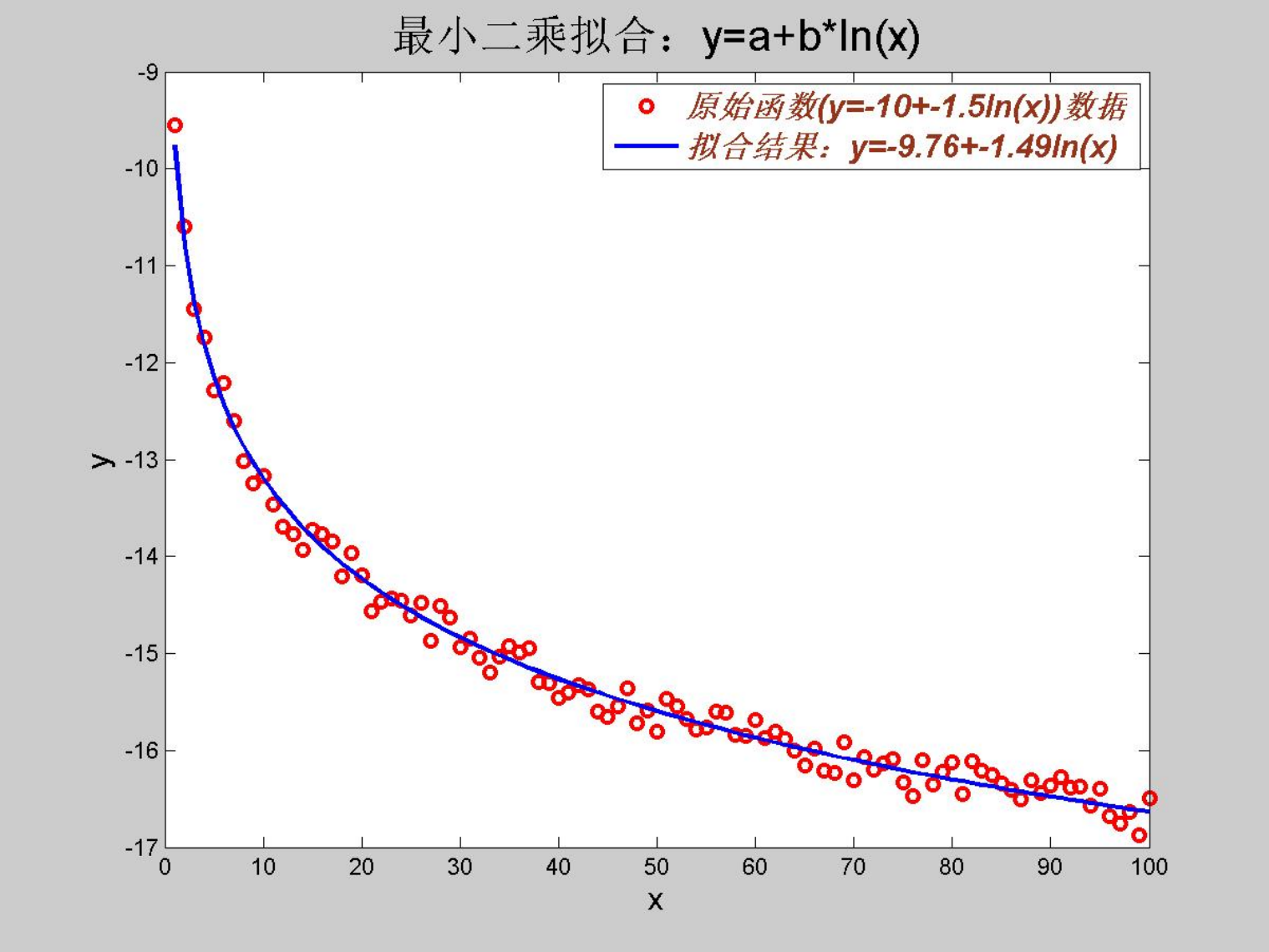
8.1.1 什么是机器学习

机器学习可以看做是一门人工智能的科学，该领域的主要研究对象是人工智能。**机器学习利用数据或以往的经验，以此优化计算机程序的性能标准。**



机器学习强调三个关键词：算法、经验、性能

从最小二乘法说起


最小二乘拟合：y=a+b*ln(x)
原始函数(y=-10+-1.5ln(x))数据
拟合结果：y=-9.76+-1.49ln(x)
y
x
-9
-10
-11
-12
-13
-14
-15
-16
-17
0
10
20
30
40
50
60
70
80
90
100

机器学习的泛化能力

机器学习中的逼近目标函数过程

监督式机器学习通常理解为逼近一个目标函数(f)，此函数映射输入变量(X)到输出变量(Y).

$$Y=f(X)$$

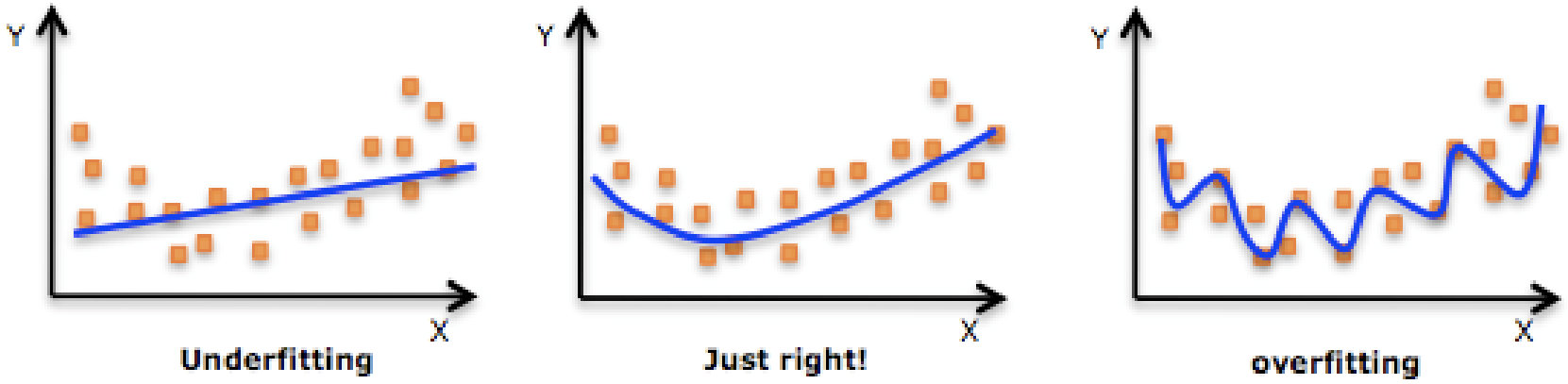
这种特性描述可以用于定义分类和预测问题和机器学习算法的领域。

从训练数据中学习目标函数的过程中，必须考虑的问题是模型在预测新数据时的泛化性能。

机器学习的过拟合问题

Torture the data, and it will confess to anything

在机器学习表现不佳的原因要么是过度拟合或欠拟合数据。



8.1.2 基于大数据的机器学习

- 传统的机器学习算法，由于技术和单机存储的限制，只能在少量数据上使用，依赖于数据抽样
- 大数据技术的出现，可以支持在全量数据上进行机器学习
- 机器学习算法涉及大量迭代计算
- 基于磁盘的MapReduce不适合进行大量迭代计算
- 基于内存的Spark比较适合进行大量迭代计算

8.1.3 Spark 机器学习库MLlib

- Spark提供了一个基于海量数据的**机器学习库**，它提供了常用机器学习算法的分布式实现
- 开发者只需要有 Spark 基础并且了解机器学习算法的原理，以及方法相关参数的含义，就可以轻松的通过调用相应的 API 来实现基于海量数据的机器学习过程
- Spark-Shell的**即席查询**也是一个关键。算法工程师可以边写代码边运行，边看结果

8.1.3 Spark 机器学习库MLlib

•MLlib是Spark的机器学习（Machine Learning）库，旨在简化机器学习的工程实践工作

•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工作流（Pipeline）API，具体如下：

•**算法工具**：常用的学习算法，如分类、回归、聚类和协同过滤；

•**特征化工具**：特征提取、转化、降维和选择工具；

•**工作流**(Pipeline)：用于构建、评估和调整机器学习工作流的工具;

•**持久性**：保存和加载算法、模型和管道;

•**实用工具**：线性代数、统计、数据处理等工具。

8.1.3 Spark 机器学习库MLlib

Spark 机器学习库从1.2 版本以后被分为两个包：

•**spark.mllib** 包含基于RDD的原始算法API。Spark MLlib历史比较长，在1.0 以前的版本即已经包含了，提供的算法实现都是基于原始的 RDD

•**spark.ml** 则提供了基于DataFrames 高层次的API，可以用来构建机器学习工作流（PipeLine）。ML Pipeline 弥补了原始 MLlib 库的不足，向用户提供了一个基于 DataFrame 的机器学习工作流式 API 套件

8.1.3 Spark 机器学习库MLlib

MLlib目前支持4种常见的机器学习问题: 分类、回归、聚类和协同过滤

	离散数据	连续数据
监督学习	Classification、LogisticRegression(with Elastic-Net)、SVM、DecisionTree、RandomForest、GBT、NaiveBayes、MultilayerPerceptron、OneVsRest	Regression、LinearRegression(with Elastic-Net)、DecisionTree、RandomFores、GBT、AFTSurvivalRegression、IsotonicRegression
无监督学习	Clustering、KMeans、GaussianMixture、LDA、PowerIterationClustering、BisectingKMeans	Dimensionality Reduction, matrix factorization、PCA、SVD、ALS、WLS

8.2 机器学习工作流

8.2.1 机器学习工作流概念
8.2.2 构建一个机器学习工作流

8.2.1 机器学习工作流概念

在介绍工作流之前，先来了解几个重要概念：

- **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

8.2.1 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

8.2.1 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。

8.2.1 机器学习工作流概念

- **Parameter：Parameter** 被用来设置 Transformer 或者 Estimator 的参数。现在，所有转换器和估计器可共享用于指定参数的公共API。ParamMap是一组（参数，值）对

- **PipeLine：** 翻译为工作流或者管道。工作流将多个工作流阶段（转换器和估计器）连接在一起，形成机器学习的工作流，并获得结果输出

8.2.1 机器学习工作流概念

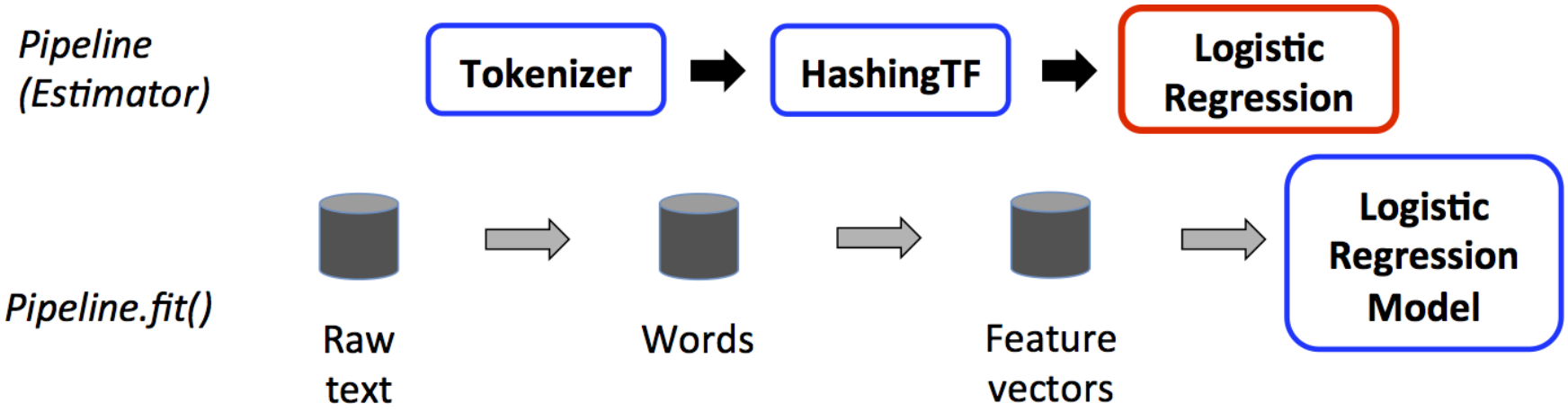
要构建一个 Pipeline工作流，首先需要定义 Pipeline 中的各个工作流阶段PipelineStage（包括转换器和评估器），比如指标提取和转换模型训练等。有了这些处理特定问题的转换器和评估器，就可以按照具体的处理逻辑有序地组织 PipelineStages 并创建一个Pipeline

```
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

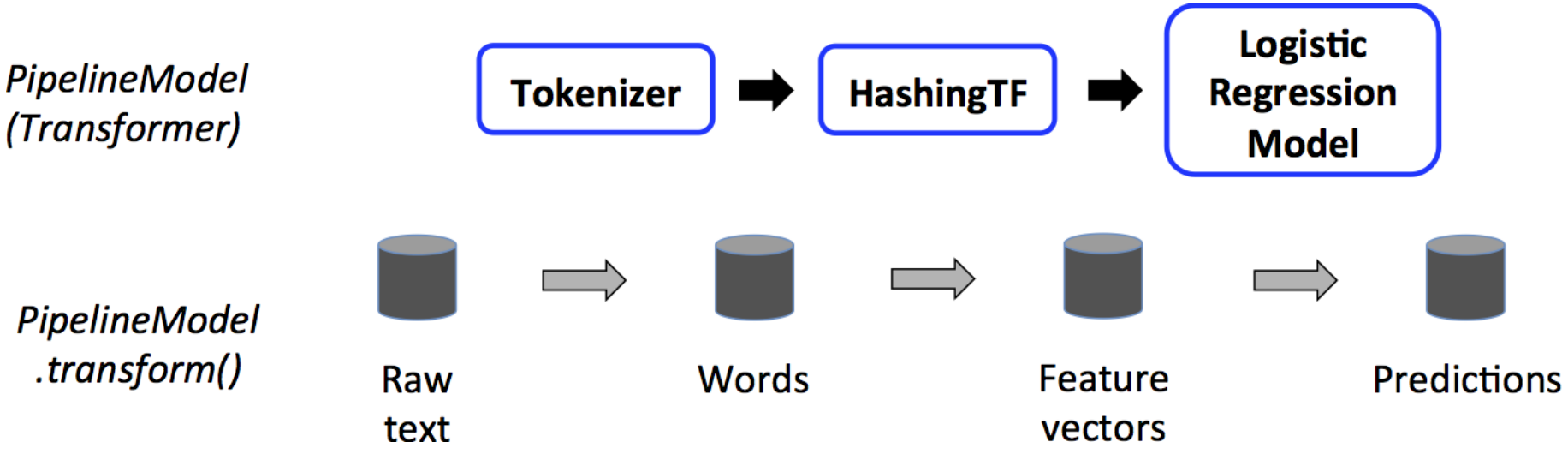
8.2.1 机器学习工作流概念

- 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**



8.2.1 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。



8.2.2 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

查找出所有包含"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。

8.2.2 构建一个机器学习工作流

- 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()
```

8.2.2 构建一个机器学习工作流

- （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,PipelineModel}
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]
```

8.2.2 构建一个机器学习工作流

（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
```

8.2.2 构建一个机器学习工作流

（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

8.2.2 构建一个机器学习工作流

（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]
```

8.2.2 构建一个机器学习工作流

（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=[0.5406433544851421,0.4593566455148578 3],
prediction=0.0
(5, l m n) --> prob=[0.9334382627383259,0.06656173726167405], prediction=0.0
(6, spark a) --> prob=[0.15041430048068286,0.849585695193171], prediction=1.0
(7, apache hadoop) --> prob=[0.9768636139518304,0.02313638604816958 5],
prediction=0.0
```

8.3 特征抽取、转化和选择

8.3.1 特征抽取：TF-IDF
8.3.2 特征抽取：Word2Vec
8.3.3 特征抽取：CountVectorizer
8.3.4 特征变换：标签和索引的转化
8.3.5 特征选取：卡方选择器

8.3.1 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：

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

8.3.1 特征抽取：TF-IDF

在Spark ML库中，TF-IDF被分成两部分：
•TF (+hashing)
• IDF

•**TF**: HashingTF 是一个Transformer，在文本处理中，接收词条的集合然后把这些集合转化成固定长度的特征向量。这个算法在哈希的同时会统计各个词条的词频。

•**IDF**: IDF是一个Estimator，在一个数据集上应用它的fit()方法，产生一个IDFModel。 该IDFModel 接收特征向量（由HashingTF产生），然后计算每一个词在文档中出现的频次。IDF会减少那些在语料库中出现频率较高的词的权重。

8.3.1 特征抽取：TF-IDF

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）

8.3.1 特征抽取：TF-IDF

（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]
```

8.3.1 特征抽取：TF-IDF

（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] |
+-----+-----------------------------------+-----------------------------------------+
```

8.3.1 特征抽取：TF-IDF

（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]) |
+-----------------------------------------------------------------------+
```

8.3.1 特征抽取：TF-IDF

（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

8.3.1 特征抽取：TF-IDF

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,[240,333,1105,1329,1357,1777],[0.6931471805599453,0.6931471805599453,1.3862943611198906,0.5753641449035617,0.6931471805599453,0.6931471805599453]),0]
[(2000,[213,342,489,495,1329,1809,1967],[0.6931471805599453,0.6931471805599453,0.6931471805599453,0.6931471805599453,0.28768207245178085,0.6931471805599453,0.6931471805599453]),0]
[(2000,[286,695,1138,1193,1604],[0.6931471805599453,0.6931471805599453,0.6931471805599453,0.6931471805599453,0.6931471805599453]),1]
```

8.3.2 特征抽取：Word2Vec

- 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
- 词向量表示可以在一定程度上刻画每个单词的语义
- 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
- Word2vec是一个Estimator，它采用一系列代表文档的词语来训练word2vecmodel
- 该模型将每个词语映射到一个固定大小的向量
- word2vecmodel使用文档中每个词语的平均数来将文档转换为向量，然后这个向量可以作为预测的特征，来计算文档相似度计算等等

8.3.2 特征抽取：Word2Vec

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。

8.3.2 特征抽取：Word2Vec

（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>]
```

8.3.2 特征抽取：Word2Vec

（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
```

8.3.2 特征抽取：Word2Vec

（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
```

8.3.2 特征抽取：Word2Vec

- （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,-
0.016248732805252075,0.045283683948218 83]]
[[0.05958533100783825,0.02342444069550 5054,-
0.027310076036623544]]
[[-
0.011055880039930344,0.020988055132329465,0.0426
089726388454 44]]
```

8.3.3 特征抽取：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
- 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

8.3.3 特征抽取：CountVectorizer

（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>]
```

8.3.3 特征抽取：CountVectorizer

（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
```

8.3.3 特征抽取：CountVectorizer

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

```
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）

8.3.3 特征抽取：CountVectorizer

（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])|
+---+--------------+-----------------------+
```

8.3.3 特征抽取：CountVectorizer

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])]
```

8.3.4 特征变换：标签和索引的转化

•在机器学习处理过程中，为了方便相关算法的实现，经常需要把标签数据（一般是字符串）转化成**整数索引**，或是在计算结束后将整数索引还原为相应的标签

•Spark ML包中提供了几个相关的转换器，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

•值得注意的是，用于特征转换的转换器和其他的机器学习算法一样，也属于ML Pipeline模型的一部分，可以用来构成机器学习流水线，以StringIndexer为例，其存储着进行标签数值化过程的相关 超参数，是一个Estimator，对其调用fit(..)方法即可生成相应的模型StringIndexerModel类，很显然，它存储了用于DataFrame进行相关处理的 参数，是一个Transformer（其他转换器也是同一原理）

8.3.4 特征变换：标签和索引的转化

- **StringIndexer**

- StringIndexer转换器可以把一列**类别型的特征**（或标签）进行编码，使其数值化，索引的范围从0开始，该过程可以使得相应的特征索引化，使得某些无法接受类别型特征的算法可以使用，并提高诸如决策树等机器学习算法的效率

- 索引构建的顺序为标签的频率，优先编码频率较大的标签，所以出现频率最高的标签为0号

- 如果输入的是数值型的，我们会把它转化成字符型，然后再对其进行编码

8.3.4 特征变换：标签和索引的转化

（1）首先引入必要的包，并创建一个简单的DataFrame，它只包含一个id列和一个标签列category

```
import org.apache.spark.ml.feature.{StringIndexer, StringIndexerModel}

scala> val df1 = spark.createDataFrame(Seq(
 |              (0, "a"),
 |              (1, "b"),
 |              (2, "c"),
 |              (3, "a"),
 |              (4, "a"),
 |              (5, "c"))).toDF("id", "category")
df1: org.apache.spark.sql.DataFrame = [id: int, category: string]
```

8.3.4 特征变换：标签和索引的转化

（2）随后，我们创建一个StringIndexer对象，设定输入输出列名，其余参数采用默认值，并对这个DataFrame进行训练，产生StringIndexerModel对象：

```
scala> val indexer = new StringIndexer().
| setInputCol("category").
| setOutputCol("categoryIndex")
indexer: org.apache.spark.ml.feature.StringIndexer =
strIdx_95a0a5afdb8b

scala> val model = indexer.fit(df1)
model: org.apache.spark.ml.feature.StringIndexerModel
= strIdx_4fa3ca8a82ea
```

8.3.4 特征变换：标签和索引的转化

（3）随后即可利用该对象对DataFrame进行转换操作，可以看到，StringIndexerModel依次按照出现频率的高低，把字符标签进行了排序，即出现最多的“a”被编号成0，“c”为1，出现最少的“b”为0

```
scala> val indexed1 = model.transform(df1)
indexed1: org.apache.spark.sql.DataFrame = [id: int, category: string,
categoryIndex: double]
scala> indexed1.show()
+---+--------+-------------+
| 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|
+---+--------+-------------+
```

8.3.4 特征变换：标签和索引的转化

- **IndexToString**

- 与StringIndexer相对应，IndexToString的作用是把标签索引的一列重新映射回原有的字符型标签

- 其主要使用场景一般都是**和StringIndexer配合**，先用StringIndexer将标签转化成标签索引，进行模型训练，然后在预测标签的时候再把标签索引转化成原有的字符标签。当然，你也可以另外定义其他的标签

8.3.4 特征变换：标签和索引的转化

（1）首先，和StringIndexer的实验相同，我们用StringIndexer读取数据集中的“category”列，把字符型标签转化成标签索引，然后输出到“categoryIndex”列上，构建出新的DataFrame

8.3.4 特征变换：标签和索引的转化

```
scala> val df = spark.createDataFrame(Seq(
 |       (0, "a"),
 |       (1, "b"),
 |       (2, "c"),
 |       (3, "a"),
 |       (4, "a"),
 |       (5, "c")
 |     )).toDF("id", "category")
df: org.apache.spark.sql.DataFrame = [id: int, category: string]

scala> val model = new StringIndexer().
 |       setInputCol("category").
 |       setOutputCol("categoryIndex").
 |       fit(df)
indexer: org.apache.spark.ml.feature.StringIndexerModel = strIdx_00fde0fe64d0

scala> val indexed = indexer.transform(df)
indexed: org.apache.spark.sql.DataFrame = [id: int, category: string, categoryIndex:
double]
```

8.3.4 特征变换：标签和索引的转化

（2）随后，创建IndexToString对象，读取“categoryIndex”上的标签索引，获得原有数据集的字符型标签，然后再输出到“originalCategory”列上。最后，通过输出“originalCategory”列，可以看到数据集中原有的字符标签

8.3.4 特征变换：标签和索引的转化

```
scala> val converter = new IndexToString().
| setInputCol("categoryIndex").
| setOutputCol("originalCategory")
converter: org.apache.spark.ml.feature.IndexToString = idxToStr_b95208a0e7ac
scala> val converted = converter.transform(indexed)
converted: org.apache.spark.sql.DataFrame = [id: int, category: string,
categoryIndex: double, originalCategory: string]
scala> converted.select("id", "originalCategory").show()
+---+---------------+
| id|originalCategory|
+---+---------------+
| 0| a|
| 1| b|
| 2| c|
| 3| a|
| 4| a|
| 5| c|
+---+---------------+
```

8.3.4 特征变换：标签和索引的转化

- **OneHotEncoder**

- **独热编码（One-Hot Encoding）** 是指把一列类别性特征（或称名词性特征，nominal/categorical features）映射成**一系列的二元连续特征**的过程，原有的类别性特征有几种可能取值，这一特征就会被映射成几个二元连续特征，每一个特征代表一种取值，若该样本表现出该特征，则取1，否则取0

- One-Hot编码适合一些期望类别特征为连续特征的算法，比如说逻辑斯蒂回归等。

8.3.4 特征变换：标签和索引的转化

（1）首先创建一个DataFrame，其包含一列类别性特征，需要注意的是，在使用OneHotEncoder进行转换前，DataFrame需要先使用StringIndexer将原始标签数值化：

8.3.4 特征变换：标签和索引的转化

```
import org.apache.spark.ml.feature.{OneHotEncoder, StringIndexer}
scala> val df = spark.createDataFrame(Seq(
| (0, "a"),
| (1, "b"),
| (2, "c"),
| (3, "a"),
| (4, "a"),
| (5, "c"),
| (6, "d"),
| (7, "d"),
| (8, "d"),
| (9, "d"),
| (10, "e"),
| (11, "e"),
| (12, "e"),
| (13, "e"),
| (14, "e")
| )).toDF("id", "category")
df: org.apache.spark.sql.DataFrame = [id: int, category: string]
```

剩余代码见下一页

8.3.4 特征变换：标签和索引的转化

```
scala> val indexer = new StringIndexer().
| setInputCol("category").
| setOutputCol("categoryIndex").
| fit(df)
indexer: org.apache.spark.ml.feature.StringIndexerModel =
strIdx_b315cf21d22d

scala> val indexed = indexer.transform(df)
indexed: org.apache.spark.sql.DataFrame = [id: int, category: string,
categoryIndex: double]
```

8.3.4 特征变换：标签和索引的转化

（2）随后，我们创建OneHotEncoder对象对处理后的DataFrame进行编码，可以看见，编码后的二进制特征呈稀疏向量形式，与StringIndexer编码的顺序相同，需注意的是最后一个Category（"b"）被编码为全0向量，若希望"b"也占有一个二进制特征，则可在创建OneHotEncoder时指定setDropLast(false)

8.3.4 特征变换：标签和索引的转化

```
scala> val encoder = new OneHotEncoder().
| setInputCol("categoryIndex").
| setOutputCol("categoryVec")
encoder: org.apache.spark.ml.feature.OneHotEncoder =
oneHot_bbf16821b33a

scala> val encoded = encoder.transform(indexed)
encoded: org.apache.spark.sql.DataFrame = [id: int,
category: string, categoryIndex: double, categoryVec:
vector]
```

剩余代码见下一页

8.3.4 特征变换：标签和索引的转化

```
scala> val encoder = new OneHotEncoder().
     |       setInputCol("categoryIndex").
     |       setOutputCol("categoryVec")
encoder: org.apache.spark.ml.feature.OneHotEncoder = oneHot_bbf16821b33a

scala> val encoded = encoder.transform(indexed)
encoded: org.apache.spark.sql.DataFrame = [id: int, category: string, categoryIndex: double, categoryVec: vector]

scala> encoded.show()
+---+--------+-------------+-------------+
| id|category|categoryIndex|  categoryVec|
+---+--------+-------------+-------------+
|  0|       a|          2.0|(4,[2],[1.0])|
|  1|       b|          4.0|    (4,[],[])|
|  2|       c|          3.0|(4,[3],[1.0])|
|  3|       a|          2.0|(4,[2],[1.0])|
|  4|       a|          2.0|(4,[2],[1.0])|
|  5|       c|          3.0|(4,[3],[1.0])|
|  6|       d|          1.0|(4,[1],[1.0])|
|  7|       d|          1.0|(4,[1],[1.0])|
|  8|       d|          1.0|(4,[1],[1.0])|
|  9|       d|          1.0|(4,[1],[1.0])|
| 10|       e|          0.0|(4,[0],[1.0])|
| 11|       e|          0.0|(4,[0],[1.0])|
| 12|       e|          0.0|(4,[0],[1.0])|
| 13|       e|          0.0|(4,[0],[1.0])|
| 14|       e|          0.0|(4,[0],[1.0])|
+---+--------+-------------+-------------+
```

8.3.4 特征变换：标签和索引的转化

- **VectorIndexer**

- 之前介绍的StringIndexer是针对单个类别型特征进行转换，倘若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

8.3.4 特征变换：标签和索引的转化

在下面的例子中，我们读入一个数据集，然后使用VectorIndexer训练出模型，来决定哪些特征需要被作为类别特征，将类别特征转换为索引，这里设置maxCategories为2，即只有种类小于2的特征才被认为是类别型特征，否则被认为是连续型特征：

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

scala> val data = Seq(
 |         Vectors.dense(-1.0, 1.0, 1.0),
 |         Vectors.dense(-1.0, 3.0, 1.0),
 |         Vectors.dense(0.0, 5.0, 1.0))
data: Seq[org.apache.spark.ml.linalg.Vector] = List([-1.0,1.0,1.0], [-1.0,3.0,1.0],
[0.0,5.0,1.0])
```

剩余代码见下一页

8.3.4 特征变换：标签和索引的转化

```
scala> val df =
spark.createDataFrame(data.map(Tuple1.apply)).toDF("features")
df: org.apache.spark.sql.DataFrame = [features: vector]

scala> val indexer = new VectorIndexer().
| setInputCol("features").
| setOutputCol("indexed").
| setMaxCategories(2)
indexer: org.apache.spark.ml.feature.VectorIndexer = vecIdx_abee81bafba8

scala> val indexerModel = indexer.fit(df)
indexerModel: org.apache.spark.ml.feature.VectorIndexerModel =
vecIdx_abee81bafba8
```

8.3.4 特征变换：标签和索引的转化

可以通过VectorIndexerModel的categoryMaps成员来获得被转换的特征及其映射，这里可以看到共有两个特征被转换，分别是0号和2号

```
scala> val categoricalFeatures: Set[Int] =
indexerModel.categoryMaps.keys.toSet
categoricalFeatures: Set[Int] = Set(0, 2)

scala> println(s"Chose ${categoricalFeatures.size}
categorical features: " + categoricalFeatures.mkString(", "))
Chose 2 categorical features: 0, 2
```

8.3.4 特征变换：标签和索引的转化

可以看到，0号特征只有-1，0两种取值，分别被映射成0，1，而2号特征只有1种取值，被映射成0

```
scala> val indexed = indexerModel.transform(df)
indexed: org.apache.spark.sql.DataFrame = [features: vector, indexed:
vector]

scala> indexed.show()
+-------------+-------------+
| features| indexed|
+-------------+-------------+
|[-1.0,1.0,1.0]|[1.0,1.0,0.0]|
|[-1.0,3.0,1.0]|[1.0,3.0,0.0]|
| [0.0,5.0,1.0]|[0.0,5.0,0.0]|
+-------------+-------------+
```

8.3.5 特征选取：卡方选择器

•特征选择（Feature Selection）指的是在特征向量中选择出那些“优秀”的特征，组成新的、更“精简”的特征向量的过程。它在高维数据分析中十分常用，可以剔除掉“冗余”和“无关”的特征，提升学习器的性能

•特征选择方法和分类方法一样，也主要分为有监督（Supervised）和无监督（Unsupervised）两种

•卡方选择则是统计学上常用的一种有监督特征选择方法，它通过对特征和真实标签之间进行卡方检验，来判断该特征和真实标签的关联程度，进而确定是否对其进行选择

8.3.5 特征选取：卡方选择器

•和ML库中的大多数学习方法一样，ML中的卡方选择也是以estimator+transformer的形式出现的，其主要由ChiSqSelector和ChiSqSelectorModel两个类来实现

•（1）在进行实验前，首先进行环境的设置。引入卡方选择器所需要使用的类：

```
import org.apache.spark.ml.feature.{ChiSqSelector,
ChiSqSelectorModel}
import org.apache.spark.ml.linalg.Vectors
```

8.3.5 特征选取：卡方选择器

（2）随后，创造实验数据，这是一个具有三个样本，四个特征维度的数据集，标签有1，0两种，我们将在此数据集上进行卡方选择：

```
scala> val df = spark.createDataFrame(Seq(
| (1, Vectors.dense(0.0, 0.0, 18.0, 1.0), 1),
| (2, Vectors.dense(0.0, 1.0, 12.0, 0.0), 0),
| (3, Vectors.dense(1.0, 0.0, 15.0, 0.1), 0)
| )).toDF("id", "features", "label")
df: org.apache.spark.sql.DataFrame = [id: int, features: vector ... 1 more field]

scala> df.show()
+---+-----------------+-----+
| id| features|label|
+---+-----------------+-----+
| 1|[0.0,0.0,18.0,1.0]| 1|
| 2|[0.0,1.0,12.0,0.0]| 0|
| 3|[1.0,0.0,15.0,0.1]| 0|
+---+-----------------+-----+
```

8.3.5 特征选取：卡方选择器

（3）现在，用卡方选择进行特征选择器的训练，为了观察地更明显，我们设置只选择和标签关联性最强的一个特征（可以通过setNumTopFeatures(..)方法进行设置）：

```
scala> val selector = new ChiSqSelector().
| setNumTopFeatures(1).
| setFeaturesCol("features").
| setLabelCol("label").
| setOutputCol("selected-feature")
selector: org.apache.spark.ml.feature.ChiSqSelector =
chiSqSelector_688a180ccb71

scala> val selector_model = selector.fit(df)
selector_model: org.apache.spark.ml.feature.ChiSqSelectorModel =
chiSqSelector_688a180ccb71
```

8.3.5 特征选取：卡方选择器

（4）用训练出的模型对原数据集进行处理，可以看见，第三列特征被选出作为最有用的特征列：

```
scala> val selector_model = selector.fit(df)
selector_model: org.apache.spark.ml.feature.ChiSqSelectorModel =
chiSqSelector_688a180ccb71

scala> val result = selector_model.transform(df)
result: org.apache.spark.sql.DataFrame = [id: int, features: vector ... 2
more fields]

scala> result.show(false)
+---+-----------------+-----+---------------+
|id |features |label|selected-feature|
+---+-----------------+-----+---------------+
|1 |[0.0,0.0,18.0,1.0]|1.0 |[18.0] |
|2 |[0.0,1.0,12.0,0.0]|0.0 |[12.0] |
|3 |[1.0,0.0,15.0,0.1]|0.0 |[15.0] |
+---+-----------------+-----+---------------+
```

8.4 分类与回归

8.4.1 逻辑斯蒂回归分类器
8.4.2 决策树分类器

8.4.1 逻辑斯蒂回归分类器

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。

二项logistic回归模型如下：

$$P(Y=1|x)=\frac{exp(w\cdot x+b)}{1+exp(w\cdot x+b)}$$

$$P(Y=0|x)=\frac{1}{1+exp(w\cdot x+b)}$$

其中，$x\in R^n$ 是输入，$Y\in 0,1$ 是输出，w称为权值向量，b称为偏置，$w\cdot x$ 为w和x的内积。

8.4.1 逻辑斯蒂回归模型

假设：

$$P(Y=1|x)=\pi(x), \quad P(Y=0|x)=1-\pi(x)$$

则采用“极大似然法”来估计w和b。似然函数为：

$$\prod_{i=1}^{N}[\pi(x_i)]^{y_i}[1-\pi(x_i)]^{1-y_i}$$

为方便求解，对其“对数似然”进行估计：

$$L(w)=\sum_{i=1}^{N}[y_i\log\pi(x_i)+(1-y_i)\log(1-\pi(x_i))]$$

从而对 $L(w)$ 求极大值，得到 w 的估计值。求极值的方法可以是梯度下降法，梯度上升法等。

8.4.1 逻辑斯蒂回归分类器

任务描述：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，这里主要用后两个属性（花瓣的长度和宽度）来进行分类。

8.4.1 逻辑斯蒂回归分类器

8.4.1.1 用二项逻辑斯蒂回归来解决二分类问题
8.4.1.2 用多项逻辑斯蒂回归来解决二分类问题（略）
8.4.1.3 用多项逻辑斯蒂回归来解决多分类问题（略）

8.4.1.1 用二项逻辑斯蒂回归来解决二分类问题

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;
```

8.4.1.1 用二项逻辑斯蒂回归来解决二分类问题

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 => I
ris(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

8.4.1.1 用二项逻辑斯蒂回归来解决二分类问题

```
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
```

8.4.1.1 用二项逻辑斯蒂回归来解决二分类问题

- 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]
……
……
```

8.4.1.1 用二项逻辑斯蒂回归来解决二分类问题

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
```

8.4.1.1 用二项逻辑斯蒂回归来解决二分类问题

（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]
```

8.4.1.1 用二项逻辑斯蒂回归来解决二分类问题

（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("index
edFeatures").setMaxIter(10).setRegParam(0.3).setElasticNetParam(0.8)
lr: org.apache.spark.ml.classification.LogisticRegression =
logreg_692899496c23
```

8.4.1.1 用二项逻辑斯蒂回归来解决二分类问题

（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").setOut
putCol("predictedLabel").setLabels(labelIndexer.labels)
labelConverter:
org.apache.spark.ml.feature.IndexToString =
idxToStr_c204eafabf57
```

8.4.1.1 用二项逻辑斯蒂回归来解决二分类问题

（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
```

8.4.1.1 用二项逻辑斯蒂回归来解决二分类问题

（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]
```

8.4.1.1 用二项逻辑斯蒂回归来解决二分类问题

（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
```

8.4.1.1 用二项逻辑斯蒂回归来解决二分类问题

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").setPredictio
nCol("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%

8.4.1.1 用二项逻辑斯蒂回归来解决二分类问题

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: [-
0.039617195764348 3,0.0,0.0,0.07240315639651046]Intercept: -
0.2312734634201537 9numClasses: 2numFeatures: 4
```

8.4.2 决策树分类器

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：特征选择、决策树的生成和决策树的剪枝

8.4.2 决策树分类器

（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

信息增益：特征A对训练数据集D的信息增益 $g(D,A)$，定义为集合D的经验熵 $H(D)$ 与特征A给定条件下D的经验条件熵 $H(D|A)$ 之差，即

$$g(D,A) = H(D) - H(D|A)$$

信息增益比：特征A对训练数据集D的信息增益比 $g_R(D,A)$ 定义为其信息增益 $g(D,A)$ 与训练数据集D关于特征A的值的熵 $H_A(D)$ 之比，即

$$g_R(D,A) = \frac{g(D,A)}{H_A(D)}$$

其中，$H_A(D) = -\sum_{i=1}^{n} \frac{|D_i|}{|D|} log_2 \frac{|D_i|}{|D|}$，n是特征A取值的个数。

8.4.2 决策树分类器

（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均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。

8.4.2 决策树分类器

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。

8.4.2 决策树分类器

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}
```

8.4.2 决策树分类器

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(",")).m
ap(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

8.4.2 决策树分类器

```
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]

... ...
```

8.4.2 决策树分类器

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").setOutpu
tCol("indexedFeatures").setMaxCategories(4).fit(df)
featureIndexer: org.apache.spark.ml.feature.VectorIndexerModel =
vecIdx_0649803dfa70
```

剩余代码见下一页

8.4.2 决策树分类器

```
//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").setOut
putCol("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]
```

8.4.2 决策树分类器

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
```

剩余代码见下一页

8.4.2 决策树分类器

```
//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]|
```

8.4.2 决策树分类器

5. 评估决策树分类模型

```
scala> val evaluatorClassifier = new MulticlassClassificationEvaluator().s
etLabelCol("indexedLabel").setPredictionCol("prediction").setMetricName("accu
racy")
evaluatorClassifier:
org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator =
mcEval_4abc19f3a54d

scala> val accuracy = evaluatorClassifier.evaluate(predictionsClassifier)
accuracy: Double = 0.864864864864649

scala> println("Test Error = " + (1.0 - accuracy))
Test Error = 0.135135135135135

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

剩余代码见下一页

8.4.2 决策树分类器

```
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
```

8.4.2 决策树分类器

6. 构建决策树回归模型

```
//导入所需要的包
import org.apache.spark.ml.evaluation.RegressionEvaluator
import org.apache.spark.ml.regression.DecisionTreeRegressionModel
import org.apache.spark.ml.regression.DecisionTreeRegressor
//训练决策树模型
scala> val dtRegressor = new
DecisionTreeRegressor().setLabelCol("indexedLabel")
.setFeaturesCol("indexedFeatures")
dtRegressor: org.apache.spark.ml.regression.DecisionTreeRegressor =
dtr_358e08c37f0c
//在pipeline中进行设置
scala> val pipelineRegressor = new Pipeline().setStages(Array(labelIndexer,
featureIndexer, dtRegressor, labelConverter))
pipelineRegressor: org.apache.spark.ml.Pipeline = pipeline_ae699675d015
```

剩余代码见下一页

8.4.2 决策树分类器

```
//训练决策树模型
scala> val modelRegressor = pipelineRegressor.fit(trainingData)
modelRegressor: org.apache.spark.ml.PipelineModel =
pipeline_ae699675d015
//进行预测
scala> val predictionsRegressor = modelRegressor.transform(testData)
predictionsRegressor: org.apache.spark.sql.DataFrame = [features: vector,
label: string ... 4 more fields]
//查看部分预测结果
scala> predictionsRegressor.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]|
```

8.4.2 决策树分类器

7. 评估决策树回归模型

```
scala> val evaluatorRegressor = new RegressionEvaluator().setLabelCol("ind
exedLabel").setPredictionCol("prediction").setMetricName("rmse")
evaluatorRegressor: org.apache.spark.ml.evaluation.RegressionEvaluator =
regEval_425d2aeea2dd

scala> val rmse = evaluatorRegressor.evaluate(predictionsRegressor)
rmse: Double = 0.3676073110469039

scala> println("Root Mean Squared Error (RMSE) on test data = " + rmse)
Root Mean Squared Error (RMSE) on test data = 0.3676073110469039

scala> val treeModelRegressor = modelRegressor.stages(2).asInstanceOf[Deci
sionTreeRegressionModel]
treeModelRegressor:
org.apache.spark.ml.regression.DecisionTreeRegressionModel =
DecisionTreeRegressionModel (uid=dtr_358e08c37f0c) of depth 5 with 13 nodes
```

剩余代码见下一页

8.4.2 决策树分类器

```
scala> println("Learned regression tree model:\n" + treeModelRegressor.toDebugString)
Learned regression tree model:
DecisionTreeRegressionModel (uid=dtr_358e08c37f0c) 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: 0.5
Else (feature 0 > 6.0)
Predict: 1.0
Else (feature 3 > 1.6)
Predict: 1.0
```

从上述结果可以看到模型的标准误差为 0.3676073110469039以及训练的决策树模型结构

8.5 聚类算法

KMeans 是一个迭代求解的聚类算法

其属于**划分（Partitioning）**型的聚类方法，即首先创建K个划分，然后迭代地将样本从一个划分转移到另一个划分来改善最终聚类的质量

8.5 聚类算法

1、 随机选取k个聚类质心点（cluster centroids）为$\mu_1, \mu_2, \ldots, \mu_k \in \mathbb{R}^n$。

2、 重复下面过程直到收敛 {

对于每一个样例i，计算其应该属于的类

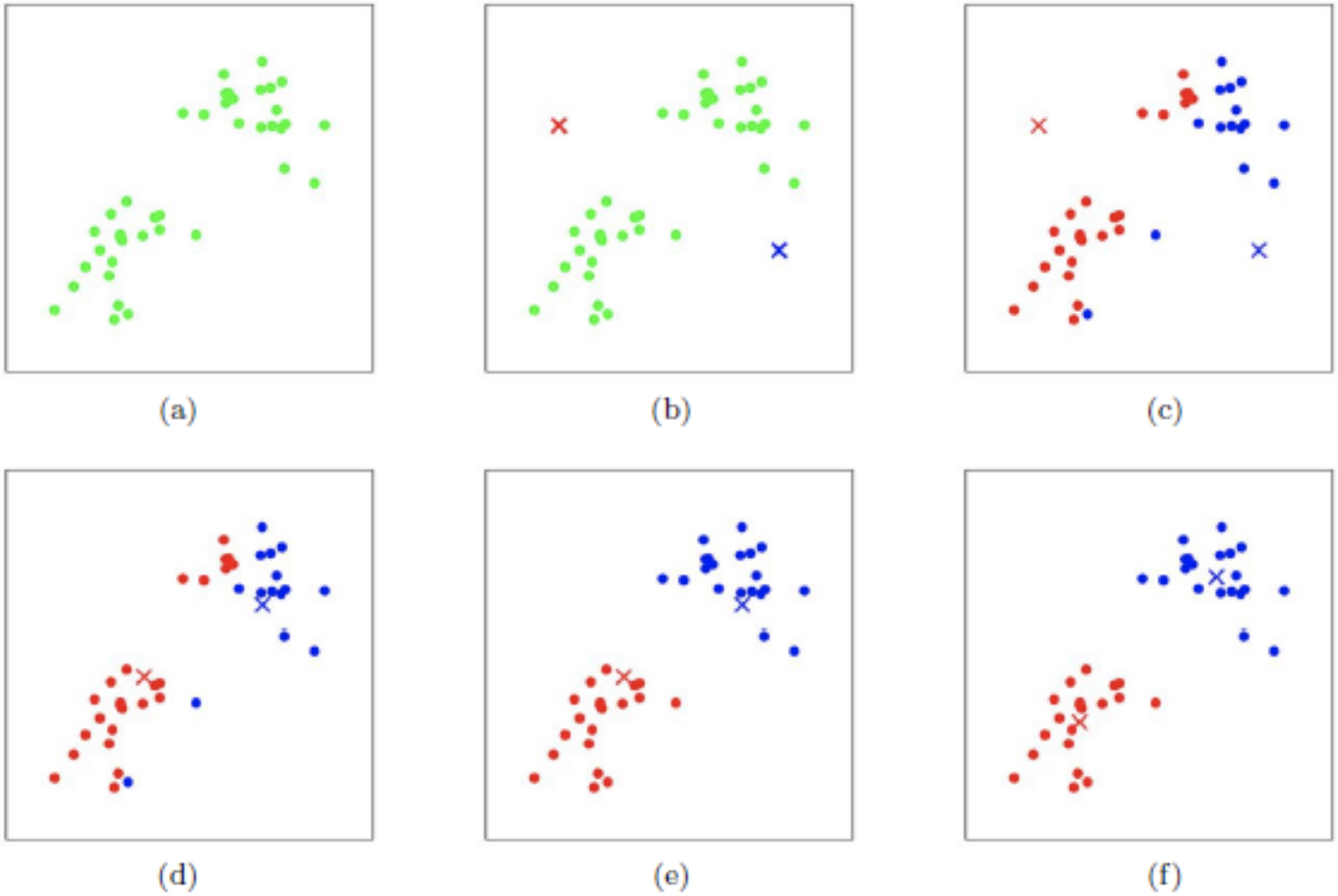
$$c^{(i)} := \arg\min_j ||x^{(i)} - \mu_j||^2.$$

对于每一个类j，重新计算该类的质心

$$\mu_j := \frac{\sum_{i=1}^{m} 1\{c^{(i)} = j\} x^{(i)}}{\sum_{i=1}^{m} 1\{c^{(i)} = j\}}.$$

}

K=2示意图



8.5 聚类算法

ML包下的KMeans方法位于org.apache.spark.ml.clustering包下，其过程大致如下：

1.根据给定的k值，选取k个样本点作为初始划分中心
2.计算所有样本点到每一个划分中心的距离，并将所有样本点划分到距离最近的划分中心
3.计算每个划分中样本点的平均值，将其作为新的中心；循环进行2~3步直至达到最大迭代次数，或划分中心的变化小于某一预定义阈值

8.5 聚类算法

数据集：使用UCI数据集中的鸢尾花数据Iris进行实验，它可以在iris获取，Iris数据的样本容量为150，有四个实数值的特征，分别代表花朵四个部位的尺寸，以及该样本对应鸢尾花的亚种类型（共有3种亚种类型）

```
5.1,3.5,1.4,0.2,setosa
...
5.4,3.0,4.5,1.5,versicolor
...
7.1,3.0,5.9,2.1,virginica
...
```

8.5 聚类算法

在使用前，引入需要的包：

```
import org.apache.spark.ml.clustering.{KMeans,KMeansModel}
import org.apache.spark.ml.linalg.Vectors
```

开启RDD的隐式转换：

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

为了便于生成相应的DataFrame，这里定义一个名为model_instance的case class作为DataFrame每一行（一个数据样本）的数据类型

```
scala> case class model_instance (features: Vector)
defined class model_instance
```

8.5 聚类算法

在定义数据类型完成后，即可将数据读入RDD[model_instance]的结构中，并通过RDD的隐式转换.toDF()方法完成RDD到DataFrame的转换：

```
scala> val rawData = sc.textFile("file:///usr/local/spark/iris.txt")
rawData: org.apache.spark.rdd.RDD[String] = iris.csv MapPartitionsRDD[48]
at textFile at <console>:33

scala> val df = rawData.map(line =>
| { model_instance( Vectors.dense(line.split(",").filter(p =>
p.matches("\\d*(\\.?)\\d*"))
| .map(_.toDouble)) )}).toDF()
df: org.apache.spark.sql.DataFrame = [features: vector]
```

8.5 聚类算法

在得到数据后，我们即可通过ML包的固有流程：创建Estimator并调用其fit()方法来生成相应的Transformer对象，很显然，在这里KMeans类是Estimator，而用于保存训练后模型的KMeansModel类则属于Transformer

```
scala> val kmeansmodel = new KMeans().
| setK(3).
| setFeaturesCol("features").
| setPredictionCol("prediction").
| fit(df)
kmeansmodel: org.apache.spark.ml.clustering.KMeansModel =
kmeans_d8c043c3c339
```

8.5 聚类算法

与MLlib中的实现不同，KMeansModel作为一个Transformer，不再提供predict()样式的方法，而是提供了一致性的transform()方法，用于将存储在DataFrame中的给定数据集进行整体处理，生成带有预测簇标签的数据集

```
scala> val results = kmeansmodel.transform(df)
results: org.apache.spark.sql.DataFrame = [features: vector, prediction: int]
```

8.5 聚类算法

为了方便观察，我们可以使用collect()方法，该方法将DataFrame中所有的数据组织成一个Array对象进行返回：

```
scala> results.collect().foreach(
| row => {
| println( row(0) + " is predicted as cluster " + row(1))
| })
[5.1,3.5,1.4,0.2] is predicted as cluster 2
...
[6.3,3.3,6.0,2.5] is predicted as cluster 1
...
[5.8,2.7,5.1,1.9] is predicted as cluster 0
...
```

8.5 聚类算法

也可以通过KMeansModel类自带的clusterCenters属性获取到模型的所有聚类中心情况：

```
scala> kmeansmodel.clusterCenters.foreach(
| center => {
| println("Clustering Center:"+center)
| })
Clustering
Center:[5.883606557377049,2.740983606557377,4.388524590163936
,1.4344262295081964]
Clustering
Center:[6.853846153846153,3.076923076923076,5.715384615384614
,2.053846153846153]
Clustering
Center:[5.005999999999999,3.4180000000000006,1.464000000000002
,0.2439999999999999]
```

8.5 聚类算法

与MLlib下的实现相同，KMeansModel类也提供了计算集合内误差平方和（Within Set Sum of Squared Error, WSSSE) 的方法来度量聚类的有效性，在真实K值未知的情况下，该值的变化可以作为选取合适K值的一个重要参考：

```
scala> kmeansmodel.computeCost(df)
res15: Double = 78.94084142614622
```

8.6 推荐算法

推荐算法是计算机专业中的一种算法，通过一些数学算法，推测出用户可能喜欢的东西。

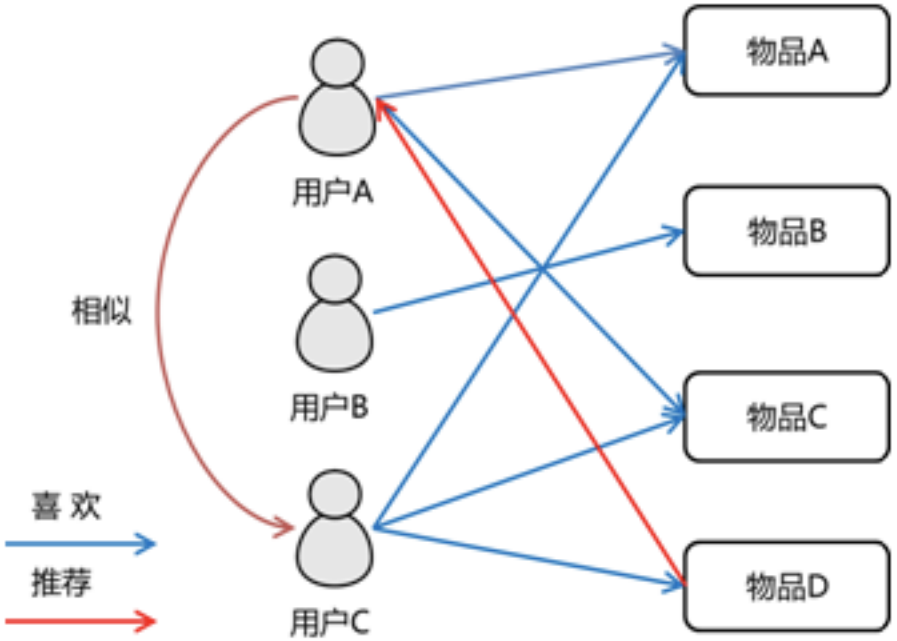
1、基于内容的信息推荐方法的理论依据主要来自于信息检索和信息过滤。根据用户过去的浏览记录来向用户推荐用户没有接触过的推荐项。

2、基于协同过滤的推荐算法理论上可以推荐世界上的任何一种东西。图片、音乐、样样可以。 协同过滤算法主要是通过对未评分项进行评分 预测来实现的。

3、基于关联规则的推荐（Association Rule-based Recommendation）是以关联规则为基础，把已购商品作为规则头，规则体为推荐对象。

8.6.1 协同过滤

关于协同过滤的一个经典的例子就是看电影。如果你不知道哪一部电影是自己喜欢的或者评分比较高的，那么通常的做法就是问问周围的朋友，看看最近有什么好的电影推荐。



8.6.1 协同过滤

协同过滤算法主要分为基于用户的协同过滤算法和基于项目的协同过滤算法。MLlib当前支持基于模型的协同过滤，其中用户和商品通过一小组隐语义因子进行表达，并且这些因子也用于预测缺失的元素。

Spark MLlib实现了 交替最小二乘法 (ALS) 来学习这些隐性语义因子。

8.6.1 显现和隐性反馈

显性反馈行为包括用户明确表示对物品喜好的行为，
隐性反馈行为指的是那些不能明确反应用户喜好的行为。

在许多的现实生活中的很多场景中，我们常常只能接触到隐性的反馈，例如页面游览，点击，购买，喜欢，分享等等。

基于矩阵分解的协同过滤的标准方法，一般将用户商品矩阵中的元素作为用户对商品的显性偏好。

8.6.1 显现和隐性反馈

在 MLlib 中所用到的处理这种数据的方法来源于文献： Collaborative Filtering for Implicit Feedback Datasets 。

评价就不是与用户对商品的显性评分，而是与所观察到的用户偏好强度关联起来。然后，这个模型将尝试找到隐语义因子来预估一个用户对一个商品的偏好。

8.6.2 ALS算法

获取MovieLens数据集，其中每行包含一个用户、一个电影、一个该用户对该电影的评分以及时间戳。

我们使用默认的ALS.train() 方法，即显性反馈（默认implicitPrefs 为false）来构建推荐模型并根据模型对评分预测的均方根误差来对模型进行评估。

8.6.2 ALS算法

1. 导入需要的包：

scala

```scala
import org.apache.spark.ml.evaluation.RegressionEvaluator
import org.apache.spark.ml.recommendation.ALS
```

2. 根据数据结构创建读取规范：

创建一个Rating类型，即[Int, Int, Float, Long];然后建造一个把数据中每一行转化成Rating类的函数。

scala

```
1. scala> case class Rating(userId: Int, movieId: Int,
   rating: Float, timestamp: Long)
2. defined class Rating
3.
4. scala> def parseRating(str: String): Rating = {
5.      |         val fields = str.split("::")
6.      |         assert(fields.size == 4)
7.      |         Rating(fields(0).toInt,
   fields(1).toInt, fields(2).toFloat,
   fields(3).toLong)
8.      |     }
9. parseRating: (str: String)Rating
```

3. 读取数据：

导入implicits，读取MovieLens数据集，把数据转化成Rating类型；

scala

```
1. scala> import spark.implicits._
2. import spark.implicits._
3.
4. scala> val ratings = spark.sparkContext.textFile("fi
   le:///usr/local/spark/data/mllib/als/sample_movielen
   s_ratings.txt").map(parseRating).toDF()
5. ratings: org.apache.spark.sql.DataFrame = [userId: i
   nt, movieId: int ... 2 more fields]
```

3. 构建模型

把MovieLens数据集划分训练集和测试集

scala

```
1. scala> val Array(training, test) = ratings.randomSplit(Array(0.8, 0.2))
2. training: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [userId: int, movieId: int... 2 more fields]
3. test: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [userId: int, movieId: int ... 2 more fields]
```

使用ALS来建立推荐模型，这里我们构建了两个模型，一个是显性反馈，一个是隐性反馈

使用ALS来建立推荐模型，这里我们构建了两个模型，一个是显性反馈，一个是隐性反馈

scala

```
scala> val alsExplicit = new ALS().setMaxIter(5).setRegParam(0.01).setUserCol("userId"). setItemCol("movieId").setRatingCol("rating")
alsExplicit: org.apache.spark.ml.recommendation.ALS = als_05fe5d65ffc3

scala> val alsImplicit = new ALS().setMaxIter(5).setRegParam(0.01).setImplicitPrefs(true). setUserCol("userId").setItemCol("movieId").setRatingCol("rating")
alsImplicit: org.apache.spark.ml.recommendation.ALS = als_7e9b959fbdae
```

在 ML 中的实现有如下的参数:

- `numBlocks` 是用于并行化计算的用户和商品的分块个数 (默认为10)。
- `rank` 是模型中隐语义因子的个数（默认为10）。
- `maxIter` 是迭代的次数（默认为10）。
- `regParam` 是ALS的正则化参数（默认为1.0）。
- `implicitPrefs` 决定了是用显性反馈ALS的版本还是用适用隐性反馈数据集的版本（默认是false，即用显性反馈）。
- `alpha` 是一个针对于隐性反馈 ALS 版本的参数，这个参数决定了偏好行为强度的基准（默认为1.0）。
- `nonnegative` 决定是否对最小二乘法使用非负的限制（默认为false）。

可以调整这些参数，不断优化结果，使均方差变小。比如：imaxIter越大，regParam越 小，均方差会越小，推荐结果较优。

接下来，把推荐模型放在训练数据上训练：

scala

```
scala> val modelExplicit = alsExplicit.fit(training)
modelExplicit: org.apache.spark.ml.recommendation.ALSModel = als_05fe5d65ffc3

scala> val modelImplicit = alsImplicit.fit(training)
modelImplicit: org.apache.spark.ml.recommendation.ALSModel = als_7e9b959fbdae
```

4. 模型预测

使用训练好的推荐模型对测试集中的用户商品进行预测评分，得到预测评分的数据集

scala

```
scala> val predictionsExplicit = modelExplicit.transform(test)
predictionsExplicit: org.apache.spark.sql.DataFrame = [userId: int, movieId: int ... 3 more fields]

scala> val predictionsImplicit = modelImplicit.transform(test)
predictionsImplicit: org.apache.spark.sql.DataFrame = [userId: int, movieId: int ... 3 more fields]
```

我们把结果输出，对比一下真实结果与预测结果：

scala

```
scala> predictionsExplicit.show()

+------+-------+------+----------+-----------+
|userId|movieId|rating| timestamp|  prediction|
+------+-------+------+----------+-----------+
|    13|     31|   1.0|1424380312|  0.86262053|
|     5|     31|   1.0|1424380312|-0.033763513|
|    24|     31|   1.0|1424380312|   2.3084288|
|    29|     31|   1.0|1424380312|   1.9081671|
|     0|     31|   1.0|1424380312|   1.6470298|
|    28|     85|   1.0|1424380312|   5.7112412|
|    13|     85|   1.0|1424380312|   2.4970412|
|    20|     85|   2.0|1424380312|   1.9727222|
|     4|     85|   1.0|1424380312|   1.8414592|
|     8|     85|   5.0|1424380312|   3.2290685|
|     7|     85|   4.0|1424380312|   2.8074787|
|    29|     85|   1.0|1424380312|   0.7150749|
|    19|     65|   1.0|1424380312|   1.7827456|
|     4|     65|   1.0|1424380312|   2.3001173|
|     2|     65|   1.0|1424380312|   4.8762875|
|    12|     53|   1.0|1424380312|   1.5465991|
|    20|     53|   3.0|1424380312|    1.903692|
|    19|     53|   2.0|1424380312|   2.6036916|
|     8|     53|   5.0|1424380312|   3.1105173|
|    23|     53|   1.0|1424380312|   1.0042696|
+------+-------+------+----------+-----------+
```

5. 模型评估

通过计算模型的均方根误差来对模型进行评估，均方根误差越小，模型越准确：

scala

```
1. scala> val evaluator = new RegressionEvaluator().set
   MetricName("rmse").setLabelCol("rating"). setPredict
   ionCol("prediction")
2. evaluator: org.apache.spark.ml.evaluation.Regression
   Evaluator = regEval_bc9d91ae7b1a
3.
4. scala> val rmseExplicit = evaluator.evaluate(predict
   ionsExplicit)
5. rmseExplicit: Double = 1.699518911876551 7
6.
7. scala> val rmseImplicit = evaluator.evaluate(predict
   ionsImplicit)
8. rmseImplicit: Double = 1.8011620822359165
```

8.7 超参数调优

在机器学习中非常重要的任务就是模型选择，或者使用数据来找到具体问题的最佳的模型和参数，这个过程也叫做调试（Tuning）。

调试可以在独立的估计器中完成（如逻辑斯蒂回归），也可以在包含多样算法、特征工程和其他步骤的工作流中完成。用户应该一次性调优整个工作流，而不是独立的调整PipeLine中的每个组成部分。

8.7.1 超参数调优的原理

MLlib支持**交叉验证**（CrossValidator）和**训练验证分割**（TrainValidationSplit）两个模型选择工具。

1.估计器：待调试的算法或管线。
2.一系列参数表（ParamMaps）：可选参数，也叫做“参数网格”搜索空间。
3.评估器：评估模型拟合程度的准则或方法。

8.7.1 超参数调优的原理

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。

8.7.1 超参数调优的原理

交叉验证CrossValidator将数据集切分成k折叠数据集合，并被分别用于训练和测试。

例如，k=3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3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 会使用这个ParamMap和整个的数据集来重新拟合Estimator。

也就是说通过交叉验证找到最佳的ParamMap，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。

8.7.1 超参数调优的原理

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=0.75时，TrainValidationSplit将会将数据切分75%作为数据集，25%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。

8.7.2 使用交叉验证进行模型选择

首先，导入必要的包：

scala

```scala
import org.apache.spark.ml.linalg.{Vector,Vectors}
import spark.implicits._
import org.apache.spark.ml.feature.{HashingTF, Tokenizer}
import org.apache.spark.ml.tuning.{CrossValidator, ParamGridBuilder}
import org.apache.spark.sql.Row
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator
import org.apache.spark.ml.feature.{IndexToString, StringIndexer, VectorIndexer}
import org.apache.spark.ml.classification.{LogisticRegression,LogisticRegressionModel}
import org.apache.spark.ml.{Pipeline,PipelineModel}
```

8.7.2 使用交叉验证进行模型选择

读取Irisi数据集，分别获取标签列和特征列，进行索引、重命名，并设置机器学习工作流。

scala

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

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()

scala>val Array(trainingData, testData) = data.randomSplit(Array(0.7, 0.3))

scala>val labelIndexer = new StringIndexer().setInputCol("label").setOutputCol("indexedLabel").fit(data)

scala> val featureIndexer = new VectorIndexer().setInputCol("features").setOutputCol("indexedFeatures").fit(data)

scala> val lr = new LogisticRegression().setLabelCol("indexedLabel").setFeaturesCol("indexedFeatures").setMaxIter(50)

scala>val labelConverter = new IndexToString().setInputCol("prediction").setOutputCol("predictedLabel").setLabels(labelIndexer.labels)


scala> val lrPipeline = new Pipeline().setStages(Array(labelIndexer, featureIndexer, lr, labelConverter))
```

8.7.2 使用交叉验证进行模型选择

使用ParamGridBuilder方便构造参数网格。

其中regParam参数定义规范化项的权重；elasticNetParam是Elastic net 参数，取值介于0和1之间。elasticNetParam设置2个值，regParam设置3个值。最终将有(3 * 2) = 6个不同的模型将被训练。

scala

```
scala> val paramGrid = new ParamGridBuilder().
     addGrid(lr.elasticNetParam, Array(0.2,0.8)).
     addGrid(lr.regParam, Array(0.01, 0.1, 0.5)).
     build()


paramGrid: Array[org.apache.spark.ml.param.ParamMap] =
Array({
logreg_cd4ae130834c-elasticNetParam: 0.2,
logreg_cd4ae130834c-regParam: 0.01
}, {
logreg_cd4ae130834c-elasticNetParam: 0.2,
logreg_cd4ae130834c-regParam: 0.1
}, {
logreg_cd4ae130834c-elasticNetParam: 0.2,
logreg_cd4ae130834c-regParam: 0.5
}, {
logreg_cd4ae130834c-elasticNetParam: 0.8,
logreg_cd4ae130834c-regParam: 0.01
}, {
logreg_cd4ae130834c-elasticNetParam: 0.8,
logreg_cd4ae130834c-regParam: 0.1
}, {
logreg_cd4ae130834c-elasticNetParam: 0.8,
logreg_cd4ae130834c-regParam: 0.5
})
```

8.7.2 使用交叉验证进行模型选择

再接下来，构建针对整个机器学习工作流的交叉验证类，定义验证模型、参数网格，以及数据集的折叠数，并调用fit方法进行模型训练。

其中，对于回归问题评估器可选择RegressionEvaluator，二值数据可选择BinaryClassificationEvaluator，多分类问题可选择MulticlassClassificationEvaluator。评估器里默认的评估准则可通过setMetricName方法重写。

8.7.2 使用交叉验证进行模型选择

scala

```
scala> val cv = new CrossValidator().
 setEstimator(lrPipeline).
  setEvaluator(new MulticlassClassificationEvaluator().setLabelCol("indexedLabel").setPredictionCol("prediction")).
  setEstimatorParamMaps(paramGrid).
  setNumFolds(3) // Use 3+ in practice
cv: org.apache.spark.ml.tuning.CrossValidator = cv_6a92251c1281

scala>val cvModel = cv.fit(trainingData)
cvModel: org.apache.spark.ml.tuning.CrossValidatorModel = cv_6a92251c1281
```

接下来，调动transform方法对测试数据进行预测，并打印结果及精度。

scala

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

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-setosa, [4.4,2.9,1.4,0.2]) --> prob=[0.0363183436604630 34,1.5386309810869498E-4,0.9635277932414283], predicted Label=Iris-setosa
(Iris-setosa, [4.4,3.0,1.3,0.2]) --> prob=[0.023099824195518 13,8.133697579147449E-5,0.9768188388286903], predicted Label=Iris-setosa
(Iris-setosa, [4.5,2.3,1.3,0.3]) --> prob=[0.295171926064505 ,0.002812152256808639,0.70201592167868 64], predicted Label=Iris-setosa
(Iris-setosa, [4.7,3.2,1.3,0.2]) --> prob=[0.016215630150446 59,3.621933015640408E-5,0.983748150519397], predicted Label=Iris-setosa
```

8.7.2 使用交叉验证进行模型选择

```
55.
56. scala>  val evaluator = new MulticlassClassificationEvaluator().
57.   setLabelCol("indexedLabel").
58.   setPredictionCol("prediction")
59. evaluator: org.apache.spark.ml.evaluation.MulticlassClassificationEval
    uator = mcEval_d56fa096e993
60.
61. scala>  val lrAccuracy = evaluator.evaluate(lrPredictions)
62. lrAccuracy: Double = 0.9773399014778326
```

还可以获取最优的逻辑斯蒂回归模型，并查看其具体的参数.
对于参数网格，其最优参数取值是regParam=0.01，elasticNetParam=0.2。

scala

```
scala>  val bestModel= cvModel.bestModel.asInstanceOf[PipelineModel]
bestModel: org.apache.spark.ml.PipelineModel = pipeline_656a51f08dc4
scala>  val lrModel = bestModel.stages(2).
      asInstanceOf[LogisticRegressionModel]
lrModel: org.apache.spark.ml.classification.LogisticRegressionModel =
logreg_b3490372f4dd
scala>  println("Coefficients: " + lrModel.coefficientMatrix + "Interc
ept: "+lrModel.interceptVector+ "numClasses: "+lrModel.numClasses+"num
Features: "+lrModel.numFeatures)
Coefficients: 0.8354793833288098    -0.8387183954210289  0.10818320355
08958   -0.0
0.054256275295321135  -1.9426755118201342  0.8538718077791957   3.0072
40263438064
-0.3972137619684629   2.8149119052091205   -0.9001646661509906  -1.972
6757956849224  Intercept: [0.18591575015360287,-0.2846540294397142,0.0
9873827928611131]numClasses: 3numFeatures: 4

scala>  lrModel.explainParam(lrModel.regParam)
res8: String = regParam: regularization parameter (>= 0) (default:
0.0, current: 0.01)
scala>  lrModel.explainParam(lrModel.elasticNetParam)
res9: String = elasticNetParam: the ElasticNet mixing parameter, in ra
nge [0, 1]. For alpha = 0, the penalty is an L2 penalty. For alpha =
1, it is an L1 penalty (default: 0.0, current: 0.2)
```

本章小结

1、机器学习、Spark MLlib的基本概念

2、机器学习工作流

构建一个机器学习工作流、特征抽取、转化和选择

[TF-IDF、Word2Vec、CountVectorizer、标签和索引的转化、卡方选择器]

3、分类与回归

（逻辑斯蒂回归分类器、决策树分类器）

4、 聚类算法（KMeans聚类算法、）

5、 推荐算法（协同过滤算法）

6、参数调优算法

附录A：主讲教师林子雨简介

主讲教师：林子雨

单位：厦门大学计算机科学系
E-mail: ziyulin@xmu.edu.cn
个人网页：http://www.cs.xmu.edu.cn/linziyu
数据库实验室网站：http://dblab.xmu.edu.cn

扫一扫访问个人主页

林子雨，男，1978年出生，博士（毕业于北京大学），现为厦门大学计算机科学系助理教授（讲师），曾任厦门大学信息科学与技术学院院长助理、晋江市发展和改革局副局长。中国计算机学会数据库专业委员会委员，中国计算机学会信息系统专业委员会委员。国内高校首个“数字教师”提出者和建设者，厦门大学数据库实验室负责人，厦门大学云计算与大数据研究中心主要建设者和骨干成员，2013年度和2017年度厦门大学教学类奖教金获得者，荣获2017年福建省精品在线开放课程、2017年福建省本科优秀特色教材和2017年厦门大学高等教育成果二等奖。主要研究方向为数据库、数据仓库、数据挖掘、大数据、云计算和物联网，并以第一作者身份在《软件学报》《计算机学报》和《计算机研究与发展》等国家重点期刊以及国际学术会议上发表多篇学术论文。作为项目负责人主持的科研项目包括1项国家自然科学青年基金项目(No.61303004)、1项福建省自然科学青年基金项目(No.2013J05099)和1项中央高校基本科研业务费项目(No.2011121049)，主持的教改课题包括1项2016年福建省教改课题和1项2016年教育部产学协作育人项目，同时，作为课题负责人完成了国家发改委城市信息化重大课题、国家物联网重大应用示范工程区域试点泉州市工作方案、2015泉州市互联网经济调研等课题。中国高校首个“数字教师”提出者和建设者，2009年至今，“数字教师”大平台累计向网络免费发布超过500万字高价值的研究和教学资料，累计网络访问量超过500万次。打造了中国高校大数据教学知名品牌，编著出版了中国高校第一本系统介绍大数据知识的专业教材《大数据技术原理与应用》，并成为京东、当当网等网店畅销书籍；建设了国内高校首个大数据课程公共服务平台，为教师教学和学生学习大数据课程提供全方位、一站式服务，年访问量超过100万次。

附录B：大数据学习路线图

大数据学习路线图访问地址：http://dblab.xmu.edu.cn/post/10164/

附录C：《大数据技术原理与应用》教材

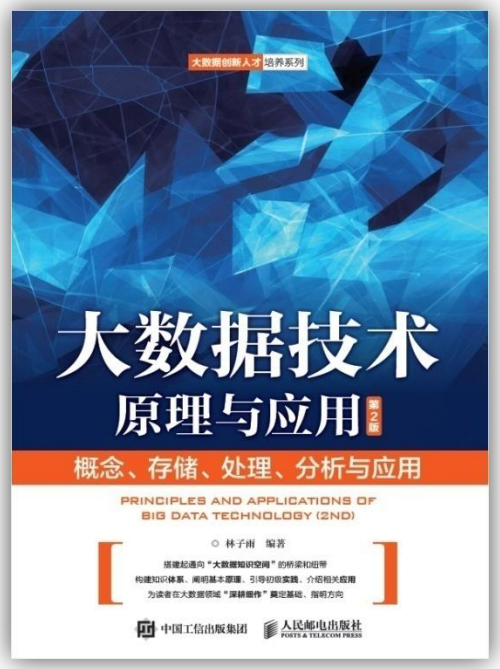
《大数据技术原理与应用——概念、存储、处理、分析与应用（第2版）》，由厦门大学计算机科学系林子雨博士编著，是国内高校第一本系统介绍大数据知识的专业教材。人民邮电出版社 ISBN:978-7-115-44330-4 定价：49.80元

全书共有15章，系统地论述了大数据的基本概念、大数据处理架构Hadoop、分布式文件系统HDFS、分布式数据 库HBase、NoSQL数据库、云数据库、分布式并行编程模型MapReduce、Spark、流计算、图计算、数据可视化以及大数据在互联网、生物医学和物流等各个领域的应用。在Hadoop、HDFS、HBase和MapReduce等重要章节，安排了入门级的实践操作，让读者更好地学习和掌握大数据关键技术。

本书可以作为高等院校计算机专业、信息管理等相关专业的大数据课程教材，也可供相关技术人员参考、学习、培训之用。

欢迎访问《大数据技术原理与应用——概念、存储、处理、分析与应用》教材官方网站：
http://dblab.xmu.edu.cn/post/bigdata

扫一扫访问教材官网



附录D：《大数据基础编程、实验和案例教程》

本书是与《大数据技术原理与应用（第2版）》教材配套的唯一指定实验指导书

大数据教材

1+1黄金组合

厦门大学林子雨编著

配套实验指导书

清华大学出版社 ISBN:978-7-302-47209-4 定价：59元

- 步步引导，循序渐进，详尽的安装指南为顺利搭建大数据实验环境铺平道路
- 深入浅出，去粗取精，丰富的代码实例帮助快速掌握大数据基础编程方法
- 精心设计，巧妙融合，五套大数据实验题目促进理论与编程知识的消化和吸收
- 结合理论，联系实际，大数据课程综合实验案例精彩呈现大数据分析全流程

附录E：《Spark编程基础》

《Spark编程基础》

厦门大学 林子雨，赖永炫，陶继平 编著

披荆斩棘，在大数据丛林中开辟学习捷径
填沟削坎，为快速学习Spark技术铺平道路
深入浅出，有效降低Spark技术学习门槛
资源全面，构建全方位一站式在线服务体系

人民邮电出版社出版发行，ISBN:978-7-115-47598-5
教材官网：http://dblab.xmu.edu.cn/post/spark/

本书以Scala作为开发Spark应用程序的编程语言，系统介绍了Spark编程的基础知识。全书共8章，内容包括大数据技术概述、Scala语言基础、Spark的设计与运行原理、Spark环境搭建和使用方法、RDD编程、Spark SQL、Spark Streaming、Spark MLlib等。本书每个章节都安排了入门级的编程实践操作，以便读者更好地学习和掌握Spark编程方法。本书官网免费提供了全套的在线教学资源，包括讲义PPT、习题、源代码、软件、数据集、授课视频、上机实验指南等。

附录F：高校大数据课程公共服务平台

高校大数据课程

公 共 服 务 平 台

http://dblab.xmu.edu.cn/post/bigdata-teaching-platform/

扫一扫访问平台主页

扫一扫观看3分钟FLASH动画宣传片

Department of Computer Science, Xiamen University, 2018