9.7 分类算法



分类是一种重要的机器学习和数据挖掘技术。分类的目的是根据数据集的特点构造一个分类函数或分类模型（也称作分类器），分类模型能把未知类别的样本映射到给定类别中。

9.7 分类算法



分类的具体规则可描述如下：给定一组训练数据的集合 T，T 的每一条记录都是包含若干条属性的特征向量，用向量 $\boldsymbol{X}=(x_1,x_2,\cdots,x_n)$ 表示。$x_i(i=1,2,\cdots,n)$ 可以有不同的值域，当一属性的值域为连续域时，该属性为连续属性（Numerical Attribute），否则为离散属性（Discrete Attribute）。用 $\boldsymbol{C}=(c_1,c_2,\cdots,c_k)$ 表示类别属性，即数据集有 k 个不同的类别。那么，T 就隐含了一个从向量 X 到类别属性 C 的映射函数：$f(\boldsymbol{X})\rightarrow\boldsymbol{C}$。分类的目的就是分析输入数据，通过在训练集中的数据表现出来的特性，为每一个类找到一种准确的分类函数或者分类模型，采用该分类函数或者分类模型将隐含的映射函数表示出来。

9.7 分类算法



构造分类模型的过程一般分为训练和测试两个阶段。在构造模型之前，将数据集随机地分为训练数据集和测试数据集。先使用训练数据集来构造分类模型，然后使用测试数据集来评估模型的分类准确率。如果认为模型的准确率可以接受，就可以用该模型对其他数据元组进行分类。一般来说，测试阶段的代价远低于训练阶段的。

9.7 分类算法



分类算法具有多种不同的类型，例如，支持向量机（Support Vector Machines，SVM）、决策树算法、贝叶斯算法等。spark.mllib包支持各种分类算法，涉及的问题类型主要包含二分类、多分类和回归分析等。表9-4列出了spark.mllib包为不同类型问题提供的算法。

问题类型	支持的算法
二分类	线性支持向量机、逻辑斯谛回归、决策树、随机森林、梯度上升树、朴素贝叶斯
多类分类	逻辑斯谛回归、决策树、随机森林、朴素贝叶斯
回归	线性最小二乘法、Lasso回归、岭回归、决策树、随机森林、梯度上升树、保序回归

9.7 分类算法(Classification Algorithms in Mllib)



9.7.1 逻辑斯蒂回归分类算法(Logistic Regression)

9.7.2 决策树分类算法(Decision Tree Classifier)

9.7.1 逻辑斯蒂回归分类算法



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。

9.7.1 逻辑斯蒂回归分类算法



逻辑斯谛回归是统计学习中的经典分类方法，其模型属于对数线性模型。逻辑斯谛回归的因变量可以是二分类的，也可以是多分类的。二项逻辑斯谛回归模型如下。

$$P(Y=1|\boldsymbol{x})=\frac{\exp(\boldsymbol{w}\cdot\boldsymbol{x}+b)}{1+\exp(\boldsymbol{w}\cdot\boldsymbol{x}+b)} \tag{9-7}$$

$$P(Y=0|\boldsymbol{x})=\frac{1}{1+\exp(\boldsymbol{w}\cdot\boldsymbol{x}+b)} \tag{9-8}$$

其中，$\boldsymbol{x}\in\mathbf{R}^n$ 是输入，$Y\in\{0,1\}$ 是输出，$\boldsymbol{w}$ 称为权值向量，b 称为偏置，$\boldsymbol{w}\cdot\boldsymbol{x}$ 为 $\boldsymbol{w}$ 和 $\boldsymbol{x}$ 的内积。参数估计的方法是在给定训练样本点和已知的公式后，对于一个或多个未知参数枚举参数的所有可能取值，找到最符合样本点分布的参数（或参数组合）。假设

$$P(Y=1|\boldsymbol{x})=\pi(\boldsymbol{x}),\quad P(Y=0|\boldsymbol{x})=1-\pi(x) \tag{9-9}$$

则采用极大似然法来估计 $\boldsymbol{w}$ 和 b，似然函数为

$$\prod_{i=1}^{N}[\pi(x_i)]^{y_i}[1-\pi(x_i)]^{1-y_i} \tag{9-10}$$

其中，N 是训练样本的个数，(x_i,y_i)表示样本变量 x_i 对应的值为 y_i。为方便求解，对其对数似然进行估计。

9.7.1 逻辑斯蒂回归分类算法



$$L(\boldsymbol{w})=\sum_{i=1}^{N}\{y_i \ln[\pi(x_i)]+(1-y_i)\log[1-\pi(x_i)]\} \tag{9-11}$$

从而对 $L(\boldsymbol{w})$求极大值，得到 $\boldsymbol{w}$ 的估计值。为了避免过拟合的问题，一般会对成本 $L(\boldsymbol{w})$增加规范化项

$$J(\boldsymbol{w})=L(\boldsymbol{w})+\gamma\left(\alpha|\boldsymbol{w}|+(1-\alpha)\frac{1}{2}|\boldsymbol{w}|^2\right) \tag{9-12}$$

其中，参数γ称为规范化系数，用于定义规范化项的权重；α称为 Elastic Net 参数，取值介于 0 和 1 之间。α=0 时采用 L2 规范化，α=1 时采用 L1 规范化。求极值的方法可以是梯度下降法、梯度上升法等。

9.7.1 逻辑斯蒂回归分类算法



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。

9.7.1 逻辑斯蒂回归分类算法



第1步：导入本地向量Vector和Vectors，导入所需要的类。

```
>>> from pyspark.ml.linalg import Vector, Vectors
>>> from pyspark.ml.feature import IndexToString, StringIndexer, VectorIndexer
>>> from pyspark.ml.classification import LogisticRegression
>>> from pyspark.ml import Pipeline, PipelineModel
>>> from pyspark.sql import Row
>>> from pyspark.ml.classification import LogisticRegressionModel
>>> from pyspark.ml.evaluation import MulticlassClassificationEvaluator
>>> from pyspark.ml.feature import VectorAssembler
>>> from pyspark.sql.types import DoubleType
```

9.7.1 逻辑斯蒂回归分类算法



第2步：使用spark.read.csv方法读取CSV文件，再将4个特征转化为Double类型，最后利用VectorAssembler将4个特征组合成向量（转化为Double，是为了适应文件末尾空行等情况）。最后利用VectorAssembler( )将4个特征组合成向量。

```
# 指定CSV文件路径
>>> path = "file:///usr/local/spark/iris.data"

# 创建一个 SparkSession（如果尚未创建）
>>> spark = SparkSession.builder.appName("example").getOrCreate()

# 使用Spark读取CSV文件并推断列的数据类型，然后重命名列
>>> df_raw = spark.read.option("inferSchema", "true").csv(path).toDF("c0
", "c1", "c2", "c3", "label")
```

9.7.1 逻辑斯蒂回归分类算法



```
# 将列的数据类型转换为Double
>>> df_double = df_raw.select(
                        df_raw["c0"].cast(DoubleType()),
                        df_raw["c1"].cast(DoubleType()),
                        df_raw["c2"].cast(DoubleType()),
                        df_raw["c3"].cast(DoubleType()),
                        df_raw["label"]
)

# 创建VectorAssembler并设置输入列和输出列
>>> assembler = VectorAssembler(inputCols=["c0", "c1", "c2", "c3"], outputC
ol="features")

# 使用VectorAssembler将特征列组装成特征向量
>>> data = assembler.transform(df_double).select("features", "label")
```

9.7.1 逻辑斯蒂回归分类算法



```
# 显示DataFrame
>>> 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
```

9.7.1 逻辑斯蒂回归分类算法



•第3步：分别获取标签列和特征列，进行索引并进行重命名。

```
# 使用StringIndexer对标签列进行索引
>>> label_indexer = StringIndexer(inputCol="label", outputCol="indexedLabel").fit(data)

# 使用VectorIndexer对特征列进行索引
>>> feature_indexer = VectorIndexer(inputCol="features", outputCol="indexedFeatures").fit(data)
```

9.7.1 逻辑斯蒂回归分类算法



第4步：设置LogisticRegression算法的参数。这里设置了循环次数为100次，规范化项为0.3等，具体可以设置的参数，可以通过explainParams()来获取，还能看到程序已经设置的参数的结果。

```
# 创建LogisticRegression模型并设置参数
>>> lr = LogisticRegression(
    labelCol="indexedLabel",
    featuresCol="indexedFeatures",
    maxIter=100,
    regParam=0.3,
    elasticNetParam=0.8
)

# 打印LogisticRegression模型的参数说明
>>> print("LogisticRegression parameters:\n" + lr.explainParams() + "\n")
```

9.7.1 逻辑斯蒂回归分类算法



LogisticRegression( )的参数及其含义如表9-5所示。

参数	含义
elasticNetParam	ElasticNet参数α，α介于0和1之间，默认为0。当α=0时，**采用L2规范化**；当α=1时，**采用L1规范化**
family	**用来设置**描述模型的标签分类，可选项为auto、binomial和multinomial，默认为auto。auto：自动选择分类的数量，如果分类数等于1或2，设置为binomial（二分类），否则设置为multinomial。binomial：二元逻辑斯谛回归。multinomial：多元逻辑斯谛回归（softmax）
featuresCol	用来设置特征列名，默认为"features"
fitIntercept	**用来设置**是否匹配一个截距项，默认为true
labelCol	用来设置标签列名，默认为"label"
maxIter	**用来设置**最大的迭代次数，默认为100
predictionCol	用来设置预测列名，默认为"prediction"

9.7.1 逻辑斯蒂回归分类算法



probabilityCol	用来设置预测属于某一类的条件概率的列名，默认值为"probability"。因为不是所有的模型输出都是精确校准后的概率估计，所以这些概率应当视为置信度，而不是精确的概率估计
rawPredictionCol	用来设置原始预测值（也称为置信度）列名
regParam	**用来设置**正则化参数，默认为0
standardization	**用来设置**是否在模型拟合前对训练特征进行标准化处理，默认为true
threshold	**用来设置**二元分类预测的阈值，默认为0.5
thresholds	多元分类中用来调整每一个分类预测概率的阈值参数，未定义默认值。阈值参数（数组的形式）的长度要等于分类数，每一个值都要大于0（最多只能有一个值可能等于0），p/t值最大的类成为预测的类，其中p是属于某一个分类的原始概率，t是每个分类的阈值参数
tol	**用来设置**迭代算法的收敛阈值（**大于等于**0），默认为1.0E-6
weightCol	用来设置权重列名，未定义默认值，如果没有设置或设置为空，则把所有实例的权重设为1

9.7.1 逻辑斯蒂回归分类算法



第5步：设置一个IndexToString的转换器，把预测的类别重新转化成字符型的。构建一个机器学习流水线，设置各个阶段。上一个阶段的输出将是本阶段的输入。

```
# 创建IndexToString转换器并设置输入列、输出列以及标签
>>> label_converter = IndexToString(inputCol="prediction", outputCol="predictedLabel", labels=label_indexer.labels)

# 创建Pipeline，并将每个阶段添加到Pipeline中
>>> lr_pipeline = Pipeline(stages=[label_indexer, feature_indexer, lr, label_converter])
```

9.7.1 逻辑斯蒂回归分类算法



第6步：把数据集随机分成训练集和测试集，其中训练集占70%。Pipeline本质上是一个评估器，当Pipeline调用fit()的时候就产生了一个PipelineModel，它是一个转换器。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。

```
# 将数据拆分为训练集和测试集
>>> training_data, test_data = data.randomSplit([0.7, 0.3])

# 拟合Pipeline模型
>>> lr_pipeline_model = lr_pipeline.fit(training_data)

# 进行预测
>>> lr_predictions = lr_pipeline_model.transform(test_data)
```

9.7.1 逻辑斯蒂回归分类算法



第7步：输出预测的结果。用select( )选择要输出的列，用collect( )获取所有行的数据，用for循环把每行都输出。

```
# 选择要显示的列，并遍历预测结果
>>> lr_predictions.select("predictedLabel", "label", "features", "probability").collect()
>>> for row in lr_predictions.collect():
                        predicted_label = row.predictedLabel
                        label = row.label
                        features = row.features
                        prob = row.probability
                        print(f"({label}, {features}) --> prob={prob}, predicted  label={pred
icted_label}")
(Iris-setosa, [4.4,2.9,1.4,0.2]) --> prob=[0.6116230067690759,0.21558269790029488,
0.17279429533062915], predicted Label=Iris-setosa
(Iris-setosa, [4.4,3.0,1.3,0.2]) --> prob=[0.6184252581013114,0.21180686225720038,
0.1697678796414881 8], predicted Label=Iris-setosa
……
```

9.7.1 逻辑斯蒂回归分类算法



第8步：对训练的模型进行评估。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。

```
# 创建MulticlassClassificationEvaluator并设置标签列和预测列
>>> evaluator = MulticlassClassificationEvaluator()
.setLabelCol("indexedLabel").setPredictionCol("prediction")

# 计算模型的准确性
>>> lr_accuracy = evaluator.evaluate(lr_predictions)
>>> print("Logistic Regression Accuracy:", lr_accuracy)
Logistic Regression Accuracy: 0.8390640167577786
```

9.7.1 逻辑斯蒂回归分类算法



第9步：通过Pipeline Model来获取训练得到的逻辑斯谛模型。lrPipelineModel是一个PipelineModel，因此，可以通过调用它的stages( )方法来获取模型

```
# 获取LogisticRegression模型
>>> lr_model = lr_pipeline_model.stages[2]

# 打印模型的系数矩阵、截距、类别数和特征数
>>> print("Coefficients: \n", lr_model.coefficientMatrix)
Coefficients:
 3 X 4 CSRMatrix
(0,2) -0.2873
(0,3) -0.3604
(1,3) 0.1404
>>> print("Intercept: ", lr_model.interceptVector)
Intercept:  [1.0944864826554006,-0.4225827695767035,-0.6719037130786973]
>>> print("numClasses: ", lr_model.numClasses)
numClasses:  3
>>> print("numFeatures: ", lr_model.numFeatures)
numFeatures:  4
```

9.7.2 决策树分类算法



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

9.7.2 决策树分类算法



（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

9.7.2 决策树分类算法



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

$$g(D,A)=H(D)-H(D\mid A) \tag{9-13}$$

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 的经验熵 $H_A(D)$之比，即

$$g_R(D,A)=\frac{g(D,A)}{H_A(D)} \tag{9-14}$$

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。

基尼指数：分类问题中，假设有 K 个类，样本点属于第 K 类的概率为 p_k，则概率分布的基尼指数定义为

$$Gini(p)=\sum_{k=1}^{K}p_k(1-p_k)=1-\sum_{k=1}^{K}p_k^2 \tag{9-15}$$

9.7.2 决策树分类算法



（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

9.7.2 决策树分类算法



（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。

9.7.2 决策树分类算法



（三）决策树的剪枝

决策树的剪枝往往通过极小化决策树整体的损失函数来实现。一般来说，损失函数可以定义如下。

$$C_a(T)=C(T)+a|T| \tag{9-16}$$

其中，T为任意子树，$C(T)$为对训练数据的预测误差（如基尼指数），$|T|$为子树的叶节点个数，$a\geqslant 0$且为参数，$C_a(T)$为参数是a时的子树T的整体损失，参数a权衡训练数据的拟合程度与模型的复杂度。对于固定的a，一定存在使损失函数$C_a(T)$的值最小的最优子树，将其表示为T_a。当a大的时候，最优子树T_a偏小；当a小的时候，最优子树T_a偏大。

9.7.2 决策树分类算法



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。

9.7.2 决策树分类算法



第1步：导入需要的包。

```
>>> from pyspark.ml.classification import DecisionTreeClassifier
>>> from pyspark.ml import Pipeline
>>> from pyspark.ml.evaluation import MulticlassClassificationEvaluator
>>> from pyspark.ml.feature import VectorAssembler
>>> from pyspark.sql.types import DoubleType
>>> from pyspark.sql.session import SparkSession
```

9.7.2 决策树分类算法



第2步：使用spark.read.csv( )方法读取CSV文件，再将4个特征转化为Double类型数据（转化为Double类型数据是为了适应文件末尾存在空行等情况），最后利用VectorAssembler( )将4个特征组合成向量。

```
# 创建SparkSession
>>> spark = SparkSession.builder.appName("example").getOrCreate()

# 指定CSV文件路径
>>> path = "file:///usr/local/spark/iris.data"

# 使用Spark读取CSV文件并推断列的数据类型，然后重命名列
>>> df_raw = spark.read.option("inferSchema", "true").csv(path).toDF("c0",
"c1", "c2", "c3", "label")
```

剩余代码见下一页

9.7.2 决策树分类算法



```
# 将列的数据类型转换为Double
>>> df_double = df_raw.select(
                        df_raw["c0"].cast(DoubleType()),
                        df_raw["c1"].cast(DoubleType()),
                        df_raw["c2"].cast(DoubleType()),
                        df_raw["c3"].cast(DoubleType()),
                        df_raw["label"]
)

# 创建VectorAssembler并设置输入列和输出列
>>> assembler = VectorAssembler(inputCols=["c0", "c1", "c2", "c3"], output
Col="features")

# 使用VectorAssembler将特征列组装成特征向量
>>> df = assembler.transform(df_double).select("features", "label")
```

9.7.2 决策树分类算法



第3步：进一步处理特征和标签，把数据集随机分成训练集和测试集，其中训练集占70%。

```
# 创建StringIndexer用于标签列
>>> labelIndexer = StringIndexer(inputCol="label", outputCol="indexedLabel").fit(df)

# 创建VectorIndexer用于特征列
>>> featureIndexer = VectorIndexer(inputCol="features", outputCol="indexedFeatures", maxCategories=4).fit(df)

# 创建IndexToString用于将预测的标签转换回原始标签
>>> labelConverter = IndexToString(inputCol="prediction", outputCol="predictedLabel", labels=labelIndexer.labels)

# 随机拆分数据集为训练集和测试集
>>> trainingData, testData = df.randomSplit([0.7, 0.3])
```

9.7.2 决策树分类算法



第4步：创建决策树模型（DecisionTreeClassifier( )），通过setter( )方法来设置决策树的参数，也可以用ParamMap( )来设置。这里仅需要设置特征列FeaturesCol和待预测列LabelCol。具体可以设置的参数可以通过explainParams( )来获取。

```
# 创建DecisionTreeClassifier
>>> dtClassifier=DecisionTreeClassifier()
.setLabelCol("indexedLabel").setFeaturesCol("indexedFeatures")
```

9.7.2 决策树分类算法



DecisionTreeClassifier( )的参数及其含义如表9-6所示。

参数	含义
checkpointInterval	用来设置检查点的区间（大于等于1）或者使检查点不生效（-1），默认为10。例如，10就意味着缓存中每隔10次循环进行一次检查
featuresCol	用来设置特征列名，默认为"features"
impurity	用来设置信息增益的准则（大小写敏感），支持"entropy"和"gini"，默认为"gini"
labelCol	用来设置标签列名，默认为"label"
maxBins	用来设置用于离散化连续型特征以及选择在每个节点上如何对特征进行分裂的最大箱数，一定要大于等于2，并且大于等于任意类属特征的类别数量，默认为32
maxDepth	用来设置树的最大深度（大于等于0），默认为5。例如，depth设为0是指只有一个根节点；depth设为1是指有一个根节点和两个叶子节点
minInfoGain	用来设置可以将数分裂成一个树节点的最小信息增益，要求大于等于0，默认为0

9.7.2 决策树分类算法



minInstancesPerNode	用来设置分裂后每一个子节点上的最少实例数量，如果一次分裂会导致左孩子节点或右孩子节点的实例数量少于minInstancesPerNode，则认为该次分裂是无效的，将舍弃该次分裂，要求大于等于1，默认为1
predictionCol	用来设置预测列名，默认为"prediction"
probabilityCol	用来设置预测属于某一类的条件概率的列名，默认值为"probability"。因为不是所有的模型输出都是精确校准后的概率估计，所以这些概率应当视为置信度，而不是精确的概率估计
rawPredictionCol	用来设置原始预测值（也称为置信度）的列名
seed	用来设置随机数种子，默认为159147643

9.7.2 决策树分类算法



thresholds	多元分类中用来调整每一个分类预测概率的阈值参数，未定义默认值。阈值参数（数组的形式）的长度要等于分类数，每一个值都要大于0（最多只能有一个值可能等于0），p/t值最大的类成为预测的类，其中p是属于某一个分类的原始概率，t是每个分类的阈值参数
weightCol	用来设置权重列名，未定义默认值，如果没有设置或设置为空，则把所有实例的权重设为1
minWeightFractionPerNode	每个孩子节点在拆分后必须拥有的加权样本数的最小分数。如果拆分导致左孩子节点或右孩子节点总权重的分数小于minWeightFractionPerNode，则这次拆分将视为无效的而被丢弃。应在区间[0.0,0.5)内，默认为0.0
leafCol	用来设置叶子节点索引列名。通过先序方式预测每棵树中每个实例的叶子节点索引（默认为""）

9.7.2 决策树分类算法



第5步：构建机器学习流水线，在训练数据集上调用fit( )进行模型训练，并在测试数据集上调用transform( )方法进行预测。

```
# 创建Pipeline，设置阶段
>>> dtPipeline = Pipeline(stages=[labelIndexer, featureIndexer, dtClassifier, labelConverter])
# 训练Pipeline模型
>>> dtPipelineModel = dtPipeline.fit(trainingData)

# 进行预测
>>> dtPredictions = dtPipelineModel.transform(testData)

# 选择需要展示的列
>>> selected = dtPredictions.select("predictedLabel", "label", "features")
```

9.7.2 决策树分类算法



```
# 显示前100行
>>> selected.show(100)
+--------------+--------------+-----------------+
| predictedLabel|         label|         features|
+--------------+--------------+-----------------+
|    Iris-setosa|    Iris-setosa|[4.3,3.0,1.1,0.1]|
|    Iris-setosa|    Iris-setosa|[4.4,3.2,1.3,0.2]|
|    Iris-setosa|    Iris-setosa|[4.5,2.3,1.3,0.3]|
|    Iris-setosa|    Iris-setosa|[4.6,3.2,1.4,0.2]|
|Iris-versicolor|Iris-versicolor|[4.9,2.4,3.3,1.0]|
|Iris-versicolor|Iris-versicolor|[5.0,2.3,3.3,1.0]|
|    Iris-setosa|    Iris-setosa|[5.0,3.5,1.3,0.3]|
|Iris-versicolor|Iris-versicolor|[5.1,2.5,3.0,1.1]|
|    Iris-setosa|    Iris-setosa|[5.1,3.5,1.4,0.3]|
|    Iris-setosa|    Iris-setosa|[5.1,3.7,1.5,0.4]|
|    Iris-setosa|    Iris-setosa|[5.1,3.8,1.9,0.4]|
|Iris-versicolor|Iris-versicolor|[5.2,2.7,3.9,1.4]|
|    Iris-setosa|    Iris-setosa|[5.2,3.5,1.5,0.2]|
|    Iris-setosa|    Iris-setosa|[5.4,3.4,1.5,0.4]|
|    Iris-setosa|    Iris-setosa|[5.4,3.9,1.3,0.4]|
|    Iris-setosa|    Iris-setosa|[5.4,3.9,1.7,0.4]|
|Iris-versicolor|Iris-versicolor|[5.5,2.3,4.0,1.3]|
|Iris-versicolor|Iris-versicolor|[5.5,2.4,3.7,1.0]|
|Iris-versicolor|Iris-versicolor|[5.5,2.4,3.8,1.1]|
|    Iris-setosa|    Iris-setosa|[5.5,4.2,1.4,0.2]|
|Iris-versicolor|Iris-versicolor|[5.6,2.7,4.2,1.3]|
|Iris-versicolor|Iris-versicolor|[5.6,3.0,4.5,1.5]|
...
| Iris-virginica| Iris-virginica|[7.2,3.6,6.1,2.5]|
| Iris-virginica| Iris-virginica|[7.7,3.0,6.1,2.3]|
+--------------+--------------+-----------------+
```

9.7.2 决策树分类算法



```
# 创建多类别分类评估器并计算准确率
>>> evaluator = MulticlassClassificationEvaluator(labelCol="indexedLabel", predictionCol="prediction")
>>> dtAccuracy = evaluator.evaluate(dtPredictions)
>>> print("Model Accuracy:", dtAccuracy)
Model Accuracy: 0.9575250312891115  //模型的预测准确率
```

9.7.2 决策树分类算法



第6步：通过调用DecisionTreeClassificationModel的toDebugString( )方法，查看训练的决策树模型结构。

```
# 获取DecisionTreeClassificationModel
>>> treeModelClassifier = dtPipelineModel.stages[2]

# 打印学习到的分类树模型
>>> print("Learned classification tree model:\n" + treeModelClassifier.toDebugString)
Learned classification tree model:
DecisionTreeClassificationModel: uid=DecisionTreeClassifier_d30c1c6bc6be, depth=4, numNodes=9,
numClasses=3, numFeatures=4
  If (feature 2 <= 2.7)
   Predict: 0.0
  Else (feature 2 > 2.7)
   If (feature 2 <= 4.95)
    If (feature 3 <= 1.6)
     Predict: 1.0
    Else (feature 3 > 1.6)
     If (feature 1 <= 3.05)
      Predict: 2.0
     Else (feature 1 > 3.05)
      Predict: 1.0
   Else (feature 2 > 4.95)
    Predict: 2.0
```

9.8 聚类算法



聚类又称群分析，是一种重要的机器学习和数据挖掘技术。聚类的目的是将数据集中的数据对象划分到若干个簇中，并且保证每个簇的样本尽量接近，不同簇的样本尽量远离。通过聚类生成的簇是一组数据对象的集合，簇满足以下两个条件：

（1）每个簇至少包含一个数据对象；

（2）每个数据对象仅属于一个簇。

9.8 聚类算法



聚类算法可形式化描述如下：给定一组数据的集合 D，D 的每一条记录都是包含若干属性的特征向量，用向量 $\boldsymbol{x}=(x_1,x_2,\cdots,x_n)$ 表示。$x_i(i=1,2,\cdots,n)$ 可以有不同的值域，当一属性的值域为连续域时，该属性为连续属性，否则为离散属性。聚类算法将数据集 D 划分为 k 个不相交的簇 $\{C=c_1,c_2,\cdots,c_k\}$，其中 $c_i\cap c_j=\varnothing(i\neq j)$，且 $D=\bigcup_{i=1}^{k}c_i$。

9.8 聚类算法



聚类一般属于无监督分类的范畴，按照一定的要求和规律，在没有关于分类的先验知识的情况下，对数据进行区分和分类。聚类既可以作为一个单独过程，用于找寻数据内部的分布结构，也可以作为分类等其他学习任务的前驱过程。聚类算法可分为划分法（Partitioning Method）、层次法（Hierarchical Method）、基于密度的方法（Density-Based Method）、基于网格的方法（Grid-Based Method）、基于模型的方法（Model-Based Method）等。这些方法没有统一的评价指标，因为不同聚类算法的目标函数相差很大。有些聚类是基于距离的（如*K*-Means），有些是假设先验分布的（如GMM、LDA），有些是带有图聚类和谱分析性质的（如谱聚类），还有些是基于密度的（如DBSCAN）。聚类算法应该嵌入问题中进行评价。

9.8 聚类算法



9.8.1 K-Means 聚类算法
9.8.2 GMM 聚类算法

9.8.1 K-Means 聚类算法



K-Means是一个迭代求解的聚类算法，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。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；

（4）循环进行2～3步直至最大迭代次数，或划分中心的变化小于某一预定义阈值。

9.8.1 K-Means 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
...
```

9.8.1 K-Means 聚类算法



第1步：引入必要的类。

```
>>> from pyspark.ml.linalg import Vectors
>>> from pyspark.ml.clustering import KMeans, KMeansModel
>>> from pyspark.ml.evaluation import ClusteringEvaluator
>>> from pyspark.ml.feature import VectorAssembler
>>> from pyspark.sql.types import DoubleType
>>> from pyspark.sql.session import SparkSession
```

9.8.1 K-Means 聚类算法



第2步：创建数据集。使用spark.read.csv( )方法读取CSV文件，再将4个属性转化为Double类型，最后利用VectorAssembler( )将4个属性组合成向量。

```
# 创建 SparkSession
>>> spark = SparkSession.builder.appName("example").getOrCreate()

# 加载数据
>>> path = "file:///usr/local/spark/iris.data"
>>> df_raw = spark.read.option("inferSchema", "true").csv(path).toDF("c0", "c1", "c2", "c3", "label")
```



```
# 将列的数据类型转换为Double
>>> df_double = df_raw.select(
                        df_raw["c0"].cast(DoubleType()),
                        df_raw["c1"].cast(DoubleType()),
                        df_raw["c2"].cast(DoubleType()),
                        df_raw["c3"].cast(DoubleType()),
                        df_raw["label"]
)

# 创建 VectorAssembler
>>> assembler = VectorAssembler(inputCols=["c0", "c1", "c2", "c3"], outpu
tCol="features")

# 使用 VectorAssembler 转换数据
>>> df = assembler.transform(df_double).select("features")
```

9.8.1 K-Means 聚类算法



第3步：数据构建好后，即可创建Kmeans实例，并进行参数设置。

```
# 创建 KMeans 模型
>>> kmeans = KMeans().setK(3)  # 设置簇的数量
>>> kmeans.setFeaturesCol("features")  # 设置特征列
>>> kmeans.setPredictionCol("prediction")  # 设置预测列

# 训练 KMeans 模型
>>> kmeansmodel = kmeans.fit(df)
```



Kmeans( )的参数及其含义如表9-7所示。

参数	含义
featuresCol	用于指明DataFrame中用于存储训练KMeans模型的特征列的名称，默认为"features"
predictionCol	用于指明DataFrame中用于存储KMeans模型的预测结果列的名称，默认为"prediction"
k	用于指明KMeans模型形成的簇的个数，默认为2
maxIter	用于指明KMeans模型训练时最大的迭代次数，超过该迭代次数，即使残差尚未收敛，训练过程也不再继续，默认为20
Seed	用于指明KMeans（初始化）过程中产生随机数的种子，默认值是使用类名的Long类型哈希值



参数	含义
tol	用于指明KMeans模型训练时的残差收敛阈值，默认为10^{-4}
initMode	用于指明KMeans模型训练时寻找初始划分中心的方法，默认为K-Means||( )（即KMeans++( )的并行化版本）
InitSteps	用于指明使用K-Means||( )方法进行初始化时的步数，默认为2
weightCol	用于设置权重列名，未定义默认值，如果没有设置或将其设置为空，则把所有实例的权重设为1
distanceMeasure	用于指明计算样本距离的方式，可选"euclidean"和"cosine"

9.8.1 K-Means 聚类算法



第4步：通过transform( )方法对存储在df中的数据集进行整体处理，生成带有预测簇标签的数据集。

```
# 进行聚类预测
>>> results = kmeansmodel.transform(df)

# 打印结果
>>> for result in results.collect():
                        print(str(result.features) + " => cluster " + str(result.predicti
on))

[4.6,3.2,1.4,0.2] =>  cluster 1
[5.3,3.7,1.5,0.2] =>  cluster 1
[5.0,3.3,1.4,0.2] =>  cluster 1
[7.0,3.2,4.7,1.4] =>  cluster 0
[6.4,3.2,4.5,1.5] =>  cluster 0
[6.9,3.1,4.9,1.5] =>  cluster 2
[5.5,2.3,4.0,1.3] =>  cluster 0
[6.5,2.8,4.6,1.5] =>  cluster 0
```

9.8.1 K-Means 聚类算法



第5步：通过KMeansModel类自带的clusterCenters属性获取模型的所有划分中心情况。

```
# 打印 KMeans 模型的聚类中心
>>> for center in kmeansmodel.clusterCenters():
                        print("Clustering Center: {}".format(center))
Clustering Center:[5.883606557377049,2.740983606557377,4.388524590163936,1.434426229508196 4]
Clustering Center:[6.853846153846153 5,3.076923076923076,5.715384615384614,2.053846153846153]
Clustering Center:[5.005999999999999 9,3.4180000000000006,1.4640000000000002,0.2439999999999999]
```

9.8.1 K-Means 聚类算法



第6步：使用spark.ml.evaluation.ClusteringEvaluator( )计算Silhouette分数来度量聚类的有效性，该值属于区间[-1,1]，且越接近1表示簇内样本距离越小，不属于同一簇的样本距离越大。在K值未知的情况下，可利用该值选取合适的K值。

```
# 创建聚类评估器
>>> evaluator = ClusteringEvaluator()

# 计算轮廓系数
>>> silhouette = evaluator.evaluate(results)
>>> print("Silhouette Score:", silhouette)
Silhouette Score:0.7354567373091194
```

9.8.2 GMM 聚类算法



GMM 是一种概率式的聚类算法，属于生成式模型，它假设所有的数据样本都是由某一个给定参数的多元高斯分布所生成的。具体地，给定类个数 K，对于给定样本空间中的样本 $\boldsymbol{x}$，一个 GMM 的概率密度函数可以由 K 个多元高斯分布组合成的混合分布表示，即有

$$p(\boldsymbol{x}) = \sum_{i=1}^{K} w_i \cdot p(\boldsymbol{x} \mid \mu_i, \Sigma_i) \tag{9-17}$$

其中，$p(\boldsymbol{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma})$ 是以$\boldsymbol{\mu}$为均值向量、$\boldsymbol{\Sigma}$为协方差矩阵的多元高斯分布的概率密度函数。可以看出，GMM 由 K 个不同的多元高斯分布共同组成，每一个分布被称为 GMM 中的一个成分（Component），而 w_i 为第 i 个多元高斯分布在混合模型中的权重，且有 $\sum_{i=1}^{K} w_i = 1$ 。



假设存在一个 CMM，那么样本空间中样本的生成过程是，以 $w_1, w_2, \cdots, w_K$ 作为概率选择出一个混合成分，根据该混合成分的概率密度函数采样出相应的样本。实际上，权重可以被直观理解成相应成分产生的样本占总样本的比例。利用 GMM 进行聚类的过程便是利用 GMM 生成数据样本的逆过程：给定聚类簇数 K，通过给定的数据集，以某一种参数估计方法，推导出每一个混合成分的参数（即均值向量$\boldsymbol{\mu}$、协方差矩阵$\boldsymbol{\Sigma}$和权重 $\boldsymbol{w}$），每一个多元高斯分布成分即对应聚类后的一个簇。

GMM 在训练时使用了极大似然估计法，最大化以下对数似然函数：

$$L = \ln \prod_{i=1}^{m} p(\boldsymbol{x}) \tag{9-18}$$

$$L = \sum_{i=1}^{m} \ln \left[\sum_{i=1}^{K} w_i \cdot p(\boldsymbol{x} \mid \mu_i, \Sigma_i) \right] \tag{9-19}$$

9.8.2 GMM 聚类算法



L无法直接通过解析方式求得解，故可采用“期望-最大化”（Expectation-Maximization，EM）方法求解。具体过程如下。

（1）根据给定的K值，初始化K个多元高斯分布及其权重。

（2）根据贝叶斯定理，估计每个样本由每个成分生成的后验概率（EM方法中的E步）。

（3）根据均值、协方差的定义及第（2）步中求出的后验概率，更新均值向量、协方差矩阵和权重（EM方法中的M步）。

（4）重复第（2）～（3）步，直到对数似然函数增加值已小于收敛阈值或达到最大迭代次数。

9.8.2 GMM 聚类算法



参数估计过程完成后，对于每一个样本点，根据贝叶斯定理计算出其属于每一个簇的后验概率，并将样本划分到后验概率最大的簇上去。相对于*K*-Means等直接给出样本点的簇划分的聚类算法，GMM这种给出样本点属于每个簇的概率的聚类算法，被称为软聚类（Soft Clustering / Soft Assignment）。

9.8.2 GMM 聚类算法



第1步：引入需要的包。GMM在spark.ml.clustering包下，具体实现分为两个类：用于抽象GMM的超参数并进行训练的GaussianMixture类和训练后的模型GaussianMixtureModel类。

```
>>> from pyspark.ml.clustering import GaussianMixture, GaussianMixtureModel
>>> from pyspark.ml.linalg import Vectors
>>> from pyspark.sql.session import SparkSession
```

9.8.2 GMM 聚类算法



第2步：创建数据集。使用spark.read.csv( )方法读取CSV文件，再将4个属性转化为Double类型数据，最后利用VectorAssembler( )将4个属性组合成向量。

```
# 创建 SparkSession
>>> spark = SparkSession.builder.appName("example").getOrCreate()

# 加载数据
>>> path = "file:///usr/local/spark/iris.data"
>>> df_raw = spark.read.option("inferSchema", "true").csv(path).toDF("c0
", "c1", "c2", "c3", "label")
```



```
# 将列的数据类型转换为Double
>>> df_double = df_raw.select(
                        df_raw["c0"].cast(DoubleType()),
                        df_raw["c1"].cast(DoubleType()),
                        df_raw["c2"].cast(DoubleType()),
                        df_raw["c3"].cast(DoubleType()),
                        df_raw["label"]
)

# 创建 VectorAssembler
>>> assembler = VectorAssembler(inputCols=["c0", "c1", "c2", "c3"], outp
utCol="features")

# 使用 VectorAssembler 转换数据
>>> df = assembler.transform(df_double).select("features")
```

9.8.2 GMM 聚类算法



第3步：数据构建好后，即可创建一个GaussianMixture对象，设置相应的超参数，并调用fit( )方法来训练一个GMM模型GaussianMixture。

```
# 创建 Gaussian Mixture 模型
>>> gm = GaussianMixture().setK(3).setPredictionCol("Prediction")
.setProbabilityCol("Probability")

# 训练模型
>>> gmm = gm.fit(df)
```



这里建立了一个简单的GaussianMixture对象并设定模型参数，设定其聚类数目为3，其他参数取默认值。GaussianMixture( )的参数及其含义如表9-8所示。

参数	含义
featuresCol	用于指明DataFrame中用于存储训练GMM的特征列的名称，默认为"features"
predictionCol	用于指明DataFrame中用于存储GMM的预测结果列的名称，默认为"prediction"
probabilityCol	用于指明DataFrame中用于存储GMM中每个样本的类条件概率（属于每一个簇的概率）向量的列名称，默认为"probability"
k	用于指明GMM中独立高斯分布的个数，即其他聚类方法中的簇的个数，默认为2



maxIter	用于指明GMM训练时最大的迭代次数，超过该迭代次数，即使残差尚未收敛，训练过程也不再继续，默认为100
seed	用于指明GMM训练过程中产生随机数的种子，默认值是使用类名的Long类型哈希值
tol	用于指明GMM训练时的残差收敛阈值，默认为10^{-2}
weightCol	用于设置权重列名，未定义默认值，如果没有设置或将其设置为空，则把所有实例的权重设为1

9.8.2 GMM 聚类算法



第4步：调用transform( )方法处理数据集并进行输出。除了可以得到样本的聚簇归属预测，GMM还可以得到样本属于各个聚簇的概率（Probability列）。

```
# 使用 Gaussian Mixture 模型进行聚类预测
>>> result = gmm.transform(df)

# 显示结果
>>> result.show(150, truncate=False)
+-----------+----------+-----------------------------------------------------+
|features   |Prediction|Probability                                          |
+-----------+----------+-----------------------------------------------------+
|[5.1,3.5,1.4,0.2]|0    |[0.9999999999999951,4.682229962936943E-17,4.868372929920407E-15]
|
|...........|..        |.....................................................  |
|[5.6,2.8,4.9,2.0]|1    |[8.920203149708086E-16,0.5988576194515217,0.4011423805484774]    |
|...........|..        |.....................................................  |
|[6.3,2.7,4.9,1.8]|2    |[5.703158630226758E-16,0.022033640207248576,0.9779663597927509]
|
+-----------+----------+-----------------------------------------------------+
```

9.8.2 GMM 聚类算法



第5步：得到模型后即可查看模型的相关参数。与K-Means不同，GMM不直接给出划分中心，而是给出各个混合成分（多元高斯分布）的参数。GaussianMixtureModel类的weights成员获取各个混合成分的权重，gaussians成员获取各个混合成分。其中，GMM的每一个混合成分都使用一个MultivariateGaussian类（位于pyspark.ml.stat包中）来存储，可以通过gaussians成员来获取各个混合成分的参数（均值向量和协方差矩阵）。

9.8.2 GMM 聚类算法



```
# 获取聚类簇数
>>> k = gmm.getK()

# 打印每个组件的权重、均值向量和协方差矩阵
>>> for i in range(k):
                        print("Component {}: ".format(i))
                        print("Weight: {}".format(gmm.weights[i]))
                        print("Mu Vector: \n{}".format(gmm.gaussians[i].mean))
                        print("Sigma Matrix: \n{}".format(gmm.gaussians[i].cov))
```

9.8.2 GMM 聚类算法



```
Component 0:
Weight: 0.12546708740088508
Mu Vector:
[6.721852396316658,2.7996898412653586,5.229488485659469,1.570225067661127]
Sigma Matrix:
DenseMatrix([[ 0.40089763,  0.06842869,  0.50528596,  0.17973192],
             [ 0.06842869,  0.06124048, -0.00228475,  0.00110913],
             [ 0.50528596, -0.00228475,  0.9129803 ,  0.2905964 ],
             [ 0.17973192,  0.00110913,  0.2905964 ,  0.10570587]])
Component 1:
Weight: 0.5412029763511748
Mu Vector:
[6.155383209846753,2.888765174842858,4.830983979377062,1.700513183480127]
Sigma Matrix:
DenseMatrix([[0.38247243, 0.14260242, 0.39328439, 0.17608763],
             [0.14260242, 0.11933761, 0.18134559, 0.0951517 ],
             [0.39328439, 0.18134559, 0.58972204, 0.29454179],
             [0.17608763, 0.0951517 , 0.29454179, 0.19234477]])
```

9.8.2 GMM 聚类算法



```
Component 2:
Weight: 0.3333299362479401
Mu Vector:
[5.006002149883481,3.4180031119033694,1.4640001282927495,0.2439994005789338]
Sigma Matrix:
DenseMatrix([[0.1217635 , 0.09829175, 0.01581558, 0.0103358 ],
             [0.09829175, 0.14227602, 0.01144781, 0.01120811],
             [0.01581558, 0.01144781, 0.02950388, 0.00558381],
             [0.0103358 , 0.01120811, 0.00558381, 0.01126387]])
```

9.9 频繁模式挖掘算法



频繁模式挖掘（Frequent Pattern Mining），又称关联规则挖掘，是一个重要的数据挖掘分析过程，它从各种数据库中找到频繁模式（Frequent Pattern）、关联或因果结构。给定一组交易，或一组项目集合，频繁模式挖掘的目的是找到一些关联规则，使我们能够根据交易中其他项目来预测一个特定项目是否在该交易中。给定项目集合 $I=\{a_1,a_2,\cdots,a_m\}$，关联规则可表示为一种归纳形式的规则 $X \Rightarrow Y$，其中 $X,Y \subseteq I$，$X \cap Y = \varnothing$，$|X| \neq 0$，$|Y| \neq 0$，$|X|$ 和 $|Y|$ 表示集合包含的元素个数。这个规则表明如果交易中包含模式 X，则该交易很有可能包含模式 Y。

9.9 频繁模式挖掘算法



频繁模式挖掘可形式化描述为：给定包含交易的数据库 $DB=<T_1,T_2,\cdots,T_n>$，其中 $T_i \subseteq I, i \in [1,2,\cdots,n]$。定义模式 $A \subseteq I$ 的支持度（support）为数据库 DB 中包含模式 A 的交易的数量，记为 $\sup(A)$。给定最小支持阈值 ξ，若 $\sup(A) \geqslant \xi$，则称 A 为频繁模式。频繁模式挖掘的目标就是找出数据库 DB 中的所有频繁模式。

得到频繁模式后，给定置信度计算函数 conf 与置信度阈值 ξ_c，将包含项目数大于 1 的频繁模式划分为不相交的两个子集 X 和 Y。对所有可能的划分计算规则置信度 $\text{conf}(X \Rightarrow Y)$，若 $\text{conf}(X \Rightarrow Y) \geqslant \xi_c$，则将该规则保留。

9.9 频繁模式挖掘算法



9.9.1 FP-Growth 算法

9.9.2 PrefixSpan 算法

9.9.1 FP-Growth 算法



FP-Growth算法包含两项重要内容，分别是频繁模式树（Frequent Pattern Tree，FP-Tree）和基于频繁模式树的模式片段增长挖掘（Pattern Fragment Growth Mining）算法。前者是一种扩展的前缀树结构，是对数据库的压缩表示。树状结构不仅保留了数据库中的项目集，还记录了项目集之间的关联。这种压缩形式能够避免在寻找频繁模式时多次扫描数据库。后者是一种运用分治思想扫描FP-Tree获取频繁模式的算法。

9.9.2 FP-Growth 算法



FP-Tree 的构建是通过将每个项目集逐一映射到树中的一个路径来实现的，而频繁项目则是路径上的节点。树上频繁出现的节点比不频繁出现的节点有更大的机会被共享。给定交易数据库 DB 与最小支持阈值 ξ，FP-Tree 的构建过程如下。

（1）扫描一次交易数据库 DB，收集频繁项目集 F 和项目对应的支持度。将 F 中的频繁项目按支持度降序排序得到 L，即降序的频繁项目列表。

（2）创建一个 FP-Tree 的根 T，并将其标记为"null"。对 DB 中的每个交易 t 执行如下步骤。

① 根据 L 的顺序，选择 t 中的频繁项目再排序。

② 记 t 中已排序的频繁项目列表为[p, P]，其中 p 是第一个元素，P 是列表剩余的元素。然后调用函数 insert_tree([p, P], T)。该函数的功能为：如果 T 有一个子节点 N 与 p 同属一个项目，则该子节点 N 计数加 1，否则创建一个（与 p 同属一个项目的）子节点 N 并将其计数置 1；如果 P 非空，则递归调用 insert_tree(P, N)。

9.9.2 FP-Growth 算法



FP-Tree 构建完成后，FP-Growth 通过模式片段增长挖掘算法找到频繁模式。该算法从一个频繁项目开始，将其作为初始后缀模式（Suffix Pattern），得到其条件模式库（Conditional Pattern Base）。这是一个由与后缀模式共同出现的频繁项目集组成的“子数据库”，再据此构建其条件 FP-Tree（Conditional FP-Tree），并使用这样的树递归地执行挖掘。这里的条件 FP-Tree 可以由从树根开始到后缀模式（不含）的所有路径组合得到。而模式的增长是通过连接后缀模式与由条件 FP-Tree 产生的新模式来实现的。定义 FP-Tree Tree，节点 α 与 FP-Growth 算法函数 FP-Growth(Tree, α)，该函数执行步骤如下。

9.9.1 FP-Growth 算法



（1）如果 Tree 只包含一条路径 P，那么对路径 P 的所有节点组合（记为 β ）执行如下操作：生成模式 $\beta \cap \alpha$，且置 $\sup(\beta \cap \alpha)$ 为 β 中节点的最小支持度。

（2）否则，对 Tree 中每一个项目第一次加入 Tree 的节点 a_i 执行如下操作：

① 生成模式 $\beta = a_i \cap \alpha$，且置 $\sup(\beta) = \sup(a_i)$；

② 构建 β 的条件模式库与条件 FP-Tree Tree_β，如果 $\text{Tree}_\beta \neq \varnothing$，则调用 FP-Growth($\text{Tree}_\beta$, β)。

通过调用 FP-Growth(Tree, null)即可搜索数据库中的所有频繁模式。

9.9.1 FP-Growth 算法



spark.ml包下的FP-Growth( )函数位于pyspark.ml.fpm包下。下面的例子采用Spark自带的sample_fpgrowth数据集，在Spark的安装目录下可以找到该文件。

```
/usr/local/spark/data/mllib/sample_fpgrowth.txt
```

其中，每行代表一次交易，交易中的项目以空格分开。此外，设置最小支持度0.5、置信度阈值0.6。

9.9.2 FP-Growth 算法



第1步：引入需要的包。

```
>>> from pyspark.ml.fpm import FPGrowth
>>> from pyspark.sql import SparkSession
```

9.9.2 FP-Growth 算法



第2步：读取sample_fpgrowth数据集中的每一行，并用空格分割其中的项目，将分割出的项目转化成列表。

```
# 创建 SparkSession
>>> spark = SparkSession.builder \
                        .appName("example") \
                        .getOrCreate()

# 加载文本数据并创建 DataFrame
>>> data = spark.sparkContext.textFile
("file:///usr/local/spark/data/mllib/sample_fpgrowth.txt")
    .map(lambda t: (t.split(" "),))
    .toDF(["items"])
```

9.9.1 FP-Growth 算法



```
# 显示 DataFrame
>>> data.show(truncate=False)
+----------------------+
|items                 |
+----------------------+
|[r, z, h, k, p]       |
|[z, y, x, w, v, u, t, s]|
|[s, x, o, n, r]       |
|[x, z, y, m, t, s, q, e]|
|[z]                   |
|[x, z, y, r, q, t, p] |
+----------------------+
```

9.9.2 FP-Growth 算法



第3步：数据构建好后，即可创建FP-Growth模型，并进行参数设置。

```
# 创建 FPGrowth 模型
>>> fpgrowth = FPGrowth()
.setItemsCol("items").setMinSupport(0.5).setMinConfidence(0.6)
```

9.9.1 FP-Growth 算法



FPGrowth( )的参数及其含义如表9-9所示。

参数	含义
itemsCol	用于指明DataFrame中用于存储训练FP-Growth模型的特征列的名称，默认为"items"
minConfidence	用于指明生成关联规则的最小置信度，默认为0.8
minSupport	用于指明频繁模式的最小支持度，取值属于[0.0,1.0]。任何出现次数超过minSupport * size-of-the-dataset的模式将被输出到频繁项集。默认为0.3
predictionCol	用于指明DataFrame中用于存储FP-Growth模型的预测结果列的名称，默认为"prediction"

9.9.2 FP-Growt

第4步：输出频繁模式集。

```
>>> model.freqItemsets.show()
+------------+----+
|       items|freq|
+------------+----+
|          [s]|   3|
|       [s, x]|   3|
|          [r]|   3|
|          [y]|   3|
|       [y, x]|   3|
|    [y, x, z]|   3|
|       [y, t]|   3|
|    [y, t, x]|   3|
| [y, t, x, z]|   3|
|    [y, t, z]|   3|
|       [y, z]|   3|
|          [x]|   4|
|       [x, z]|   3|
|          [t]|   3|
|       [t, x]|   3|
|    [t, x, z]|   3|
|       [t, z]|   3|
|          [z]|   5|
+------------+----+
```



9.9.2 FP-Gro

第5步：输出生成的关联规则。

```
>>> model.associationRules.show()
+----------+----------+----------+------------------+-------+
|antecedent|consequent|confidence|              lift|support|
+----------+----------+----------+------------------+-------+
|       [t]|       [y]|       1.0|               2.0|    0.5|
|       [t]|       [x]|       1.0|               1.5|    0.5|
|       [t]|       [z]|       1.0|               1.2|    0.5|
| [y, t, x]|       [z]|       1.0|               1.2|    0.5|
|       [x]|       [s]|      0.75|               1.5|    0.5|
|       [x]|       [y]|      0.75|               1.5|    0.5|
|       [x]|       [z]|      0.75|0.8999999999999999|    0.5|
|       [x]|       [t]|      0.75|               1.5|    0.5|
|    [y, z]|       [x]|       1.0|               1.5|    0.5|
|    [y, z]|       [t]|       1.0|               2.0|    0.5|
|    [y, t]|       [x]|       1.0|               1.5|    0.5|
|    [y, t]|       [z]|       1.0|               1.2|    0.5|
|    [y, x]|       [z]|       1.0|               1.2|    0.5|
|    [y, x]|       [t]|       1.0|               2.0|    0.5|
| [y, x, z]|       [t]|       1.0|               2.0|    0.5|
| [y, t, z]|       [x]|       1.0|               1.5|    0.5|
|       [s]|       [x]|       1.0|               1.5|    0.5|
|       [y]|       [x]|       1.0|               1.5|    0.5|
|       [y]|       [t]|       1.0|               2.0|    0.5|
|       [y]|       [z]|       1.0|               1.2|    0.5|
+----------+----------+----------+------------------+-------+
only showing top 20 rows
```



9.9.2 FP-Growth 算法



第6步：对输入交易应用生成的关联规则，并将结果作为预测值输出。

```
>>> model.transform(data).show()
+------------------+----------+
|             items|prediction|
+------------------+----------+
|      [r, z, h, k, p]| [y, x, t]|
|[z, y, x, w, v, u…|        []|
|      [s, x, o, n, r]| [y, z, t]|
|[x, z, y, m, t, s…|        []|
|               [z]| [y, x, t]|
|[x, z, y, r, q, t…|       [s]|
+------------------+----------+
```

9.9.3 PrefixSpan 算法



PrefixSpan 算法用于挖掘序列数据中的频繁序列模式。序列数据是由若干个数据项集组成的序列，项集之间有时间上的先后关系，如<a(bc)(bd)>，它由 a,bc,bd 共 3 个项集数据组成。频繁序列就是满足最小支持度要求的子序列，子序列的数学定义是：对于序列 $A=\{a_1, a_2, \cdots, a_m\}$ 和序列 $B=\{b_1, b_2, \cdots, b_n\}$，如果存在下标序列 $1 \leqslant i_1 \leqslant i_2 \leqslant \cdots \leqslant i_m \leqslant n$，满足 $a_1 \subseteq b_{i_1}, a_2 \subseteq b_{i_2}, \cdots, a_n \subseteq b_{i_m}$，则称 A 是 B 的子序列。通常来说，A 序列的每个元素必须按顺序地是 B 序列的对应元素的子集。

9.9.3 PrefixSpan 算法



下面解释 PrefixSpan 算法中的重要概念，包括前缀、投影、后缀。

前缀的定义如下：对于序列 $A=\{a_1,a_2,\cdots,a_m\}$ 和序列 $B=\{b_1,b_2,\cdots,b_n\}$，如果满足① $a_1=b_1,a_2=b_2,\cdots,a_{m-1}=b_{m-1}$，② $a_m\subseteq b_m$，$m\leqslant n$，③ (b_m-a_m) 中的元素按顺序都排列在 a_m 后面，则称 A 是 B 的前缀。通俗地讲，前缀要求 A 的前 $m-1$ 个元素都必须等于 B 的前 $m-1$ 个元素，A 的第 m 个元素是 B 的第 m 个元素的前面的一部分。比如<a>、<ab>、<abc>都是序列<a(bc)(bd)>的前缀，但<ac>不是<a(bc)(bd)>的前缀。

9.9.3 PrefixSpan 算法



投影的定义如下：给定序列 A、B，且 A 是 B 的子序列，序列 C 是序列 B 的投影，当且仅当①C 是 B 的子序列，②C 包含前缀 A，③不存在比 C 更长的序列 D，使 D 是 B 的子序列，并且 D 包含前缀 A。特别注意：前缀这个概念是针对投影而言的。序列 A 是投影 C 的前缀，只需序列 A 是序列 B 的子序列。比如，<ac>不是序列 B=<a(bc)(bd)>的前缀，却是序列 B 关于<ac>的投影 C=<ac(bd)>的前缀。

每个前缀都有一个对应的后缀，即 B 的投影 C 减去前缀 A 所剩的序列，记作 B/A。如果前缀的最后一项是项集的一部分，则用一个“_”来占位表示。表 9-10 是指定前缀后的序列<a(bc)(bd)>的投影和后缀。

表 9-10　　指定前缀后的序列<a(bc)(bd)>的投影和后缀

给定前缀	投影	后缀
<a>	<a(bc)(bd)>	<(bc)(bd)>
<ac>	<ac(bd)>	<(bd)>
<acb>	<ac(bd)>	<_d>

9.9.3 PrefixSpan 算法



PrefixSpan 算法的目标是挖掘出满足给定的最小支持度要求的频繁序列。它采用分而治之的思想，先从长度为 1 的前缀开始，搜索对应的投影数据库得到长度为 1 的前缀对应的频繁序列。再对长度为 2 的前缀进行搜索，不断递归地进行搜索。以表 9-11 中的序列数据为例，支持度设为 50%。

表 9-11　　序列数据

序列 id	序列
1	<bac)>
2	<c(ab)>
3	<a(bc)(bd)>
4	<cf>

算法的第一步是找出长度为 1 的频繁序列。因此，我们需要对表 9-11 中的序列数据进行统计，结果为<a>:3、<b>:3、<c>:4、<d>:1、<f>:1。<a>:3 的意思是在 4 条序列中，有 3 条包含子序列<a>。因为支持度是 50%，所以统计值小于 2 的子序列不属于频繁序列。在结果中，<d>和<f>的统计值只有 1，所以得到的长度为 1 的频繁序列为<a>、<b>、<c>。

9.9.3 PrefixSpan 算法



PrefixSpan 的思想是分而治之，由第一步的结果可以将频繁序列的全体集合划分为 3 部分：①包含前缀<a>的频繁序列；②包含前缀<b>的频繁序列；③包含前缀<c>的频繁序列。下面以<a>为例解释递归挖掘的过程，其他节点同理。递归过程需要构造前缀<a>的投影数据库，如表 9-12 所示。

表 9-12　　<a>的投影数据库

<c>
<_b>
<(bc)bd>
<>

与第一步的做法相同，对<a>的投影数据库中的元素进行统计，结果为<b>:1、<_b>:1、<c>:2、<d>:1。这里要特别注意的是，<b>和<_b>是不同的，<b>与前缀< a >是不同项集，而<_b>与前缀<a>属于同一项集。除了<c>外，其他元素都不满足最小支持度的要求，所以第 2 轮得到的前缀为<a>的长度为 2 的频繁序列为<ac>。

根据递归搜索，第 3 轮需要构造出前缀<ac>的投影数据库。<ac>的投影只有一项，就是<bd>，元素<b>和<d>的计数都只有 1，都不满足支持度的要求，递归挖掘结束。前缀为<a>的频繁序列的全集为<a>、<ac>。仿照上述过程，可以依次求出前缀为<b>和前缀为<c>的频繁序列。

9.9.3 PrefixSpan 算法



PrefixSpan的算法流程总结如下。

（1）对长度为1的前缀进行计数，得到所有的频繁1项序列，长度L=1。

（2）对于每个长度为L的、满足最小支持度要求的前缀进行递归挖掘。

① 如果前缀对应投影数据库为空，则递归返回。

② 统计对应投影数据库各项的支持度。如果各项支持度都小于阈值，则递归返回。

③ 将满足支持度阈值要求的各个单项和目前的前缀合并，得到若干个新的前缀。

④ L=L+1，前缀为第③步得到的新前缀，分别递归执行第②步。

9.9.3 PrefixSpan 算法



PrefixSpan算法位于pyspark.ml.fpm包下。下面给出PrefixSpan的具体例子，使用上文例子作为输入数据，1对应a，2对应b，以此类推。序列总共有4条，最小支持度设为0.5，这意味着只有出现频率至少为22的子序列才会被选中。

9.9.3 PrefixSpan 算法



第1步：引入需要的包。

```
>>> from pyspark.ml.fpm import PrefixSpan
>>> from pyspark.sql import SparkSession
```

9.9.3 PrefixSpan 算法



第2步：构造输入数据。

```
# 创建 SparkSession
>>> spark = SparkSession.builder \
    .appName("example") \
    .getOrCreate()

# 定义小型测试数据
>>> small_test_data = [
                        [[2], [1], [3]],
                        [[3], [1, 2]],
                        [[1], [2, 3], [2, 4]],
                        [[3], [5]]
]

# 转换为 DataFrame
>>> data = [(seq,) for seq in small_test_data]
>>> df = spark.createDataFrame(data, ["sequence"])
```

9.9.3 PrefixSpan 算法



第3步：创建PrefixSpan模型，并进行参数设置。

```
# 创建 PrefixSpan 模型
model = PrefixSpan() \
    .setMinSupport(0.5) \
    .setMaxPatternLength(5) \
    .setMaxLocalProjDBSize(32000000)
```

9.9.3 PrefixSpan 算法



PrefixSpan( )的参数及其含义如表9-13所示。

参数	含义
MinSupport	用于指定某子序列被认定为频繁序列模式的最小支持度，取值属于[0.0, 1.0]，默认为0.1
MaxPatternLength	用于指定频繁序列模式的最大长度，默认为10
MaxLocalProjDBSize	用于指定在开始对投影数据库进行迭代处理之前，数据库中允许的最大项目数，默认为32000000
SequenceCol	用于指定数据集中sequence列的名称（默认为 "sequence"），该列中的空行将被忽略

9.9.3 PrefixSpan 算法



第4步：查找频繁序列模式。

```
>>> result = model.findFrequentSequentialPatterns(df)
```

9.9.3 PrefixSpan 算法



第5步：输出结果。

```
>>> result.show()
+----------+----+
|  sequence|freq|
+----------+----+
|     [[3]]|   4|
|     [[1]]|   3|
|     [[2]]|   3|
|[[1], [3]]|   2|
|[[3], [2]]|   2|
+----------+----+
```

9.10 协同过滤算法



9.10.1 推荐算法的原理

9.10.2 ALS算法

9.10.1 推荐算法的原理



- 协同过滤算法主要分为基于用户的协同过滤（User-Based CF）算法和基于物品的协同过滤（Item-Based CF）算法。基于用户的协同过滤，通过不同用户对物品的评分来评测用户之间的相似性，并基于用户之间的相似性做出推荐。基于物品的协同过滤，通过用户对不同物品的评分来评测物品之间的相似性，并基于物品之间的相似性做出推荐。MLlib当前支持基于模型的协同过滤，其中，用户和物品通过一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。

9.10.1 推荐算法的原理



- 在推荐过程中，用户的反馈有显性和隐性之分。显性反馈行为指的是那些用户明确表示对物品的喜好程度的行为，隐性反馈行为指的是那些不能明确反应用户喜好的行为。在现实生活的很多场景中，常常只能接触到隐性反馈行为。例如，页面浏览、单击、购买、喜欢、分享等。基于矩阵分解的协同过滤的标准方法，一般将用户物品矩阵中的元素作为用户对物品的显性偏好。

9.10.2 ALS算法



ALS是Alternating Least Squares的缩写，即交替最小二乘法。该方法常用于基于矩阵分解的推荐系统中。例如，将用户（User）对物品（Item）的评分矩阵分解为两个矩阵：一个是用户对物品隐含特征的偏好矩阵，另一个是物品所包含的隐含特征的矩阵。在这个矩阵分解的过程中，评分缺失项得到了填充，即可以基于填充的评分来给用户推荐物品。

9.10.2 ALS算法



具体而言，将用户-物品的评分矩阵 $\boldsymbol{R}$ 分解成两个隐含因子矩阵 $\boldsymbol{P}$ 和 $\boldsymbol{Q}$，从而将用户和物品都投影到一个隐含因子的空间中去。即对于 $\boldsymbol{R}$（$m\times n$）的矩阵，ALS 旨在找到两个低维矩阵 $\boldsymbol{P}$（$m\times k$）和矩阵 $\boldsymbol{Q}$（$n\times k$），来近似逼近 $\boldsymbol{R}$（$m\times n$）：

$$\boldsymbol{R}_{m\times n}\approx \boldsymbol{P}_{m\times k}\boldsymbol{Q}_{n\times k}^{\mathrm{T}} \tag{9-20}$$

其中，$k<<\min(m,n)$。这里相当于降维，矩阵 $\boldsymbol{P}$ 和 $\boldsymbol{Q}$ 也称为低秩矩阵。

为了使低秩矩阵 $\boldsymbol{P}$ 和 $\boldsymbol{Q}$ 尽可能地逼近 R，可以最小化下面的损失函数 L 来完成：

$$L(\boldsymbol{P},\boldsymbol{Q})=\Sigma_{u,i}\left(r_{ui}-\boldsymbol{p}_u^{\mathrm{T}}\boldsymbol{q}_i\right)^2+\lambda\left(|\boldsymbol{p}_u|^2+|\boldsymbol{q}_i|^2\right) \tag{9-21}$$

其中，$\boldsymbol{p}_u$ 表示用户 u 的偏好的隐含特征向量，$\boldsymbol{q}_i$ 表示物品 i 包含的隐含特征向量，r_{ui} 表示用户 u 对物品 i 的评分，向量 $\boldsymbol{p}_u$ 和 $\boldsymbol{q}_i$ 的内积 $\boldsymbol{p}_u^{\mathrm{T}}\boldsymbol{q}_i$ 是用户 u 对物品 i 评分的近似。最小化该损失函数使两个隐含因子矩阵的乘积尽可能逼近原始的评分。同时，损失函数中增加了 L2 规范化项（Regularization Term），对较大的参数值进行惩罚，以减小过拟合造成的影响。

9.10.2 ALS算法



ALS 是求解 $L(\boldsymbol{P},\boldsymbol{Q})$的著名算法，其基本思想是：固定其中一类参数，使其变为单类变量优化问题，利用解析方法进行优化；再反过来，固定先前优化过的参数，再优化另一组参数。此过程迭代进行，直到收敛。具体求解过程如下。

（1）固定 $\boldsymbol{Q}$，对 $\boldsymbol{p}_u$ 求偏导数并令偏导数为 0，即 $\frac{\partial L(\boldsymbol{P},\boldsymbol{Q})}{\partial \boldsymbol{p}_u}=0$，得到求解 $\boldsymbol{p}_u$ 的公式：

$$\boldsymbol{p}_u=\left(\boldsymbol{Q}^{\mathrm{T}}\boldsymbol{Q}+\lambda I\right)^{-1}\boldsymbol{Q}^{\mathrm{T}}r_u \tag{9-22}$$

其中，I 为单位矩阵。

（2）固定 $\boldsymbol{P}$，对 $\boldsymbol{q}_i$ 求偏导数并令偏导数为 0，即 $\frac{\partial L(\boldsymbol{P},\boldsymbol{Q})}{\partial \boldsymbol{q}_i}=0$，得到求解 $\boldsymbol{q}_i$ 的公式：

$$\boldsymbol{q}_i=\left(\boldsymbol{P}^{\mathrm{T}}\boldsymbol{P}+\lambda I\right)^{-1}\boldsymbol{P}^{\mathrm{T}}r_i \tag{9-23}$$

9.10.2 ALS算法



spark.ml包提供了ALS来学习隐语义因子并进行推荐。下面的例子采用Spark自带的MovieLens数据集，在Spark的安装目录下可以找到该文件。

/usr/local/spark/data/mllib/als/sample_movielens_ratings.txt

其中，每行包含一个用户、一部电影、一个该用户对该电影的评分及时间戳。这里使用默认的ALS.train( )方法来构建推荐模型，并进行模型评估。

9.10.2 ALS算法



第1步：引入需要的包。

```
>>> from pyspark.ml.evaluation import RegressionEvaluator
>>> from pyspark.ml.recommendation import ALS
>>> from pyspark.sql import SparkSession
>>> from pyspark.sql import Row
```

9.10.2 ALS算法



第2步：创建一个Rating类和parseRating( )函数。parseRating( )用于把读取的MovieLens数据集中的每一行转化成Rating类的对象。

```
# 创建Spark会话
>>> spark = SparkSession.builder.appName("example").getOrCreate()

# 定义Rating类
>>> class Rating:
                        def __init__(self, userId, movieId, rating, timestamp):
                        self.userId = userId
                        self.movieId = movieId
                        self.rating = rating
                        self.timestamp = timestamp
```

9.10.2 ALS算法



```
# 定义解析函数
>>> def parseRating(line):
                        fields = line.split("::")
                        assert len(fields) == 4
                        return Rating(int(fields[0]), int(fields[1]), float(fields[2]), int(field
s[3]))

# 读取文件并转换为DataFrame
>>> ratings = spark.sparkContext
.textFile("file:///usr/local/spark/data/mllib/als/sample_movielens_ratings.txt") \
    .map(parseRating) \
    .map(lambda x: Row(userId=x.userId, movieId=x.movieId, rating=x.rating, timest
amp=x.timestamp)).toDF()
```

9.10.2 ALS算法



```
# 显示DataFrame的前几行数据
>>> ratings.show()
+------+-------+------+----------+
|userId|movieId|rating| timestamp|
+------+-------+------+----------+
|     0|      2|   3.0|1424380312|
|     0|      3|   1.0|1424380312|
|     0|      5|   2.0|1424380312|
|     0|      9|   4.0|1424380312|
|     0|     11|   1.0|1424380312|
|     0|     12|   2.0|1424380312|
|     0|     15|   1.0|1424380312|
|     0|     17|   1.0|1424380312|
|     0|     19|   1.0|1424380312|
|     0|     21|   1.0|1424380312|
|     0|     23|   1.0|1424380312|
|     0|     26|   3.0|1424380312|
|     0|     27|   1.0|1424380312|
|     0|     28|   1.0|1424380312|
|     0|     29|   1.0|1424380312|
|     0|     30|   1.0|1424380312|
|     0|     31|   1.0|1424380312|
|     0|     34|   1.0|1424380312|
|     0|     37|   1.0|1424380312|
|     0|     41|   2.0|1424380312|
+------+-------+------+----------+
only showing top 20 rows
```

9.10.2 ALS算法



第3步：把MovieLens数据集划分为训练集和测试集，其中训练集占80%，测试集占20%。

```
# 将数据集分割为训练集和测试集
>>> training, test = ratings.randomSplit([0.8, 0.2])
```

9.10.2 ALS算法



第4步：使用ALS来建立推荐模型。这里构建两个模型，一个是显性反馈模型，另一个是隐性反馈模型。

```
# 创建显式评分的ALS模型
>>> alsExplicit = ALS(maxIter=5, regParam=0.01, userCol="userId", itemCol="movieId", ratingCol="rating")

# 创建隐式评分的ALS模型
>>> alsImplicit = ALS(maxIter=5, regParam=0.01, implicitPrefs=True, userCol="userId", itemCol="movieId", ratingCol="rating")
```

9.10.2 ALS算法



ALS的参数及其含义如表9-14所示。

参数	含义
alpha	是一个针对隐性反馈 ALS 的参数，这个参数决定偏好行为强度的基准，默认为 1.0
checkpointInterval	用来设置检查点的区间（大于等于 1）或者使检查点不生效（-1）的参数，默认为 10。比如 10 就意味着缓存中每隔 10 次循环进行一次检查
implicitPrefs	决定是用显性反馈 ALS 还是用隐性反馈 ALS，默认是 false，即用显性反馈 ALS
itemCol	用来设置物品 id 列名，id 列的数据一定要是 Int 类型的，其他数值类型也是支持的，但只要它们落在 Int 域内，就会被强制转化成 Int，默认为"item"
maxIter	最大迭代次数，默认为 10
nonnegative	决定是否对最小二乘法使用非负的限制，默认为 false
numItemBlocks	物品的分块数，默认为 10
numUserBlocks	用户的分块数，默认为 10
predictionCol	用来设置预测列名，默认为"prediction"
rank	矩阵分解的秩，即模型中隐语义因子的个数，默认为 10
ratingCol	用来设置评分列名，默认为"rating"
regParam	正则化参数（大于等于 0），默认为 0.1
seed	随机数种子，默认为 1994790107
userCol	用来设置用户 id 列名，id 列的数据一定要是 Int 类型的，其他数值类型也是支持的。但只要它们落在 Int 域内，就会被强制转化成 Int，默认为"user"

9.10.2 ALS算法



第5步：把推荐模型放在训练数据上训练。

```
# 使用训练数据拟合显式评分的ALS模型
>>> modelExplicit = alsExplicit.fit(training)
# 使用训练数据拟合隐式评分的ALS模型
>>> modelImplicit = alsImplicit.fit(training)
```

9.10.2 ALS算法



第6步：对测试集中的用户-电影进行预测，得到预测评分的数据集。

```
# 使用显式评分的ALS模型进行预测并移除NaN值
>>> predictionsExplicit = modelExplicit.transform(test).na.drop()

# 使用隐式评分的ALS模型进行预测并移除NaN值
>>> predictionsImplicit = modelImplicit.transform(test).na.drop()
```

测试集中如果出现训练集中没有出现的用户，则此次算法将无法进行推荐和评分预测。因此，na.drop( )将删除modelExplicit.transform(test)返回结果的DataFrame中任何出现空值或NaN的行。

9.10.2 ALS算法



第7步：把结果输出，对比一下真实结果与预测结果。

```
>>> predictionsExplicit.show()
+------+-------+------+----------+----------+
|userId|movieId|rating| timestamp|prediction|
+------+-------+------+----------+----------+
|    12|     22|   2.0|1424380312| 3.2479236|
|    12|     35|   5.0|1424380312| -0.530218|
|    12|     45|   1.0|1424380312| 0.8344407|
|    12|     52|   1.0|1424380312|-2.1726284|
|    12|     75|   1.0|1424380312| 2.5783024|
|    12|     77|   1.0|1424380312|-3.2186246|
|    12|     92|   1.0|1424380312| 1.1895456|
|     1|      6|   1.0|1424380312|0.97735095|
|     1|     21|   3.0|1424380312| 0.9701156|
|     1|     28|   3.0|1424380312| 0.8518792|
|     1|     40|   1.0|1424380312| 1.9213806|
|     1|     44|   1.0|1424380312| 1.0629847|
|     1|     57|   1.0|1424380312| 1.9225161|
|     1|     60|   1.0|1424380312|  1.427048|
|     1|     76|   1.0|1424380312|0.53945637|
|     1|     86|   2.0|1424380312|0.74402493|
|     1|     88|   2.0|1424380312|  2.546386|
|    13|      5|   1.0|1424380312|0.72854286|
|    13|     26|   1.0|1424380312|0.61646223|
|    13|     52|   2.0|1424380312| 3.1166124|
+------+-------+------+----------+----------+
only showing top 20 rows
```

9.10.2 ALS算法



```
>>> predictionsImplicit.show()
+------+-------+------+----------+-----------+
|userId|movieId|rating| timestamp|  prediction|
+------+-------+------+----------+-----------+
|    12|     22|   2.0|1424380312|  0.22080728|
|    12|     35|   5.0|1424380312|  0.16498214|
|    12|     45|   1.0|1424380312|   0.4509022|
|    12|     52|   1.0|1424380312| 0.088703245|
|    12|     75|   1.0|1424380312|  0.43563348|
|    12|     77|   1.0|1424380312|-0.054829687|
|    12|     92|   1.0|1424380312|   0.3991751|
|     1|      6|   1.0|1424380312|   0.5279048|
|     1|     21|   3.0|1424380312|   0.9825162|
|     1|     28|   3.0|1424380312|   -0.483464|
|     1|     40|   1.0|1424380312|  0.48943865|
|     1|     44|   1.0|1424380312|  0.41318583|
|     1|     57|   1.0|1424380312| -0.17878239|
|     1|     60|   1.0|1424380312|  0.20307416|
|     1|     76|   1.0|1424380312|  0.16549835|
|     1|     86|   2.0|1424380312|   0.4944438|
|     1|     88|   2.0|1424380312|   0.2217942|
|    13|      5|   1.0|1424380312|-0.035340693|
|    13|     26|   1.0|1424380312|  0.13647152|
|    13|     52|   2.0|1424380312|-2.259016E-5|
+------+-------+------+----------+-----------+
```

9.10.2 ALS算法



第8步：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。

```
# 创建回归评估器
>>> evaluator = RegressionEvaluator(metricName="rmse", labelCol="rating", predi
ctionCol="prediction")

# 使用评估器计算显式评分模型的RMSE
>>> rmseExplicit = evaluator.evaluate(predictionsExplicit)

# 使用评估器计算隐式评分模型的RMSE
>>> rmseImplicit = evaluator.evaluate(predictionsImplicit)

>>> print("RMSE for Explicit Model: ", rmseExplicit)
RMSE for Explicit Model:  1.7802848438813463
>>> print("RMSE for Implicit Model: ", rmseImplicit)
RMSE for Implicit Model:  1.87522745367399 75
```

可以看到评分的均方根误差值为1.78和1.87左右。由于本例的数据较少，预测的结果和实际的结果相比有一定的差距。

9.11 模型选择



9.11.1 模型选择工具

9.11.2 用交叉验证选择模型

9.11 模型选择



- 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，这个过程也叫作调优（Tuning）。
- 调优可以在独立的评估器中（如逻辑斯谛回归）完成，也可以在包含多种算法、特征工程和其他步骤的流水线中完成。用户应该一次性调优整个流水线，而不是独立地调优流水线中的每个组成部分。

9.11.1 模型选择工具



- MLlib支持两个模型选择工具，即交叉验证（CrossValidator）和训练-验证切分（TrainValidationSplit）。使用这些工具要求模式包含如下对象：
- （1）待调优的算法或流水线；
- （2）一系列参数表（ParamMap），是可选参数，也叫作参数网格搜索空间；
- （3）评估模型拟合程度的准则或方法。

9.11.1 模型选择工具



- 模型选择工具的工作原理如下。
- （1）将输入数据划分为训练数据和测试数据。
- （2）对于每个(训练数据,测试数据)，遍历一组ParamMap。用每一个ParamMap参数来拟合评估器，得到训练后的模型，再使用评估器来评估模型表现。
- （3）选择性能表现最优的模型所对应的ParamMap。

9.11.1 模型选择工具



- 更具体地，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)，每一个(训练数据,测试数据)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(训练数据,测试数据)在评估器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，来重新拟合评估器。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出一个泛化能力强、误差小的的最佳模型。

9.11.1 模型选择工具



- 交叉验证的代价比较高昂，为此，Spark也为超参数调优提供了TrainValidationSplit。TrainValidationSplit创建单一的（训练数据,测试数据）。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会将数据切分出75%作为训练集，25%作为测试集，来生成(训练数据,测试数据)，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一次，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。

9.11.2 用交叉验证选择模型



- 使用CrossValidator的代价可能会异常高。然而，对比启发式的手动调优，这是一种选择参数的行之有效的方法。下面通过一个实例来演示如何使用CrossValidator从整个网格的参数中选择合适的参数。

9.11.2 用交叉验证选择模型



第1步：导入必要的包。

```
>>> from pyspark.ml.feature import HashingTF, Tokenizer
>>> from pyspark.ml.tuning import CrossValidator, ParamGridBuil
der
>>> from pyspark.sql import Row, SparkSession
>>> from pyspark.ml.evaluation import MulticlassClassificationEva
luator
>>> from pyspark.ml.feature import IndexToString, StringIndexer,
VectorIndexer
>>> from pyspark.ml.classification import LogisticRegression, Logi
sticRegressionModel
>>> from pyspark.ml import Pipeline, PipelineModel
>>> from pyspark.ml.feature import VectorAssembler
>>> from pyspark.sql.types import DoubleType
>>> from pyspark.sql import SparkSession
```

9.11.2 用交叉验证选择模型



第2步：读取Iris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。通过交叉验证把原始数据集分割为训练集与测试集。值得注意的是，只有训练集才可以用在模型的训练过程中，测试集则作为模型完成之后用来评估模型优劣的依据。此外，训练集中的样本数量必须足够大，一般至少要大于总样本数的50%，且两个子集必须从完整集合中均匀取样。

```
# 创建Spark会话
>>> spark = SparkSession.builder.appName("IrisClassification").g
etOrCreate()

# 读取数据
>>> path = "file:///usr/local/spark/iris.data"
>>> df_raw = spark.read.option("inferSchema", "true").csv(path).to
DF("c0", "c1", "c2", "c3", "label")
```

9.11.2 用交叉验证选择模型



```
# 转换列数据类型为Double
>>> df_double = df_raw.select(
                    df_raw["c0"].cast(DoubleType()),
                    df_raw["c1"].cast(DoubleType()),
                    df_raw["c2"].cast(DoubleType()),
                    df_raw["c3"].cast(DoubleType()),
                    df_raw["label"]
)

# 创建特征向量
>>> assembler = VectorAssembler(inputCols=["c0", "c1", "c2", "c3"],
outputCol="features")
>>> data = assembler.transform(df_double).select("features", "label
")
```

9.11.2 用交叉验证选择模型



```
# 划分训练集和测试集
>>> trainingData, testData = data.randomSplit([0.7, 0.3])

# 对标签进行索引
>>> labelIndexer = StringIndexer(inputCol="label", outputCol="inde
xedLabel").fit(data)

# 对特征向量进行索引
>>> featureIndexer = VectorIndexer(inputCol="features", outputCol=
"indexedFeatures").fit(data)
```

9.11.2 用交叉验证选择模型



```
# 创建Logistic Regression模型
>>> lr = LogisticRegression(labelCol="indexedLabel", featuresCol="
indexedFeatures", maxIter=50)

# 将预测结果转换为原始标签
>>> labelConverter = IndexToString(inputCol="prediction", outputCo
l="predictedLabel", labels=labelIndexer.labels)

# 创建Pipeline
>>> lrPipeline = Pipeline(stages=[labelIndexer, featureIndexer, lr, la
belConverter])
```

9.11.2 用交叉验证选择模型



第3步：使用ParamGridBuilder( )方法构造参数网格。其中，regParam参数是公式（9-12）中的γ，用于定义规范化项的权重；elasticNetParam参数是α，称为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，最终有3×2 = 6个不同的模型将被训练。

```
# 创建参数网格
>>> paramGrid = ParamGridBuilder() \
                        .addGrid(lr.elasticNetParam, [0.2, 0.8]) \
                        .addGrid(lr.regParam, [0.01, 0.1, 0.5]) \
                        .build()
```

9.11.2 用交叉验证选择模型



第4步：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，对于二值数据可选择BinaryClassificationEvaluator，对于多分类问题可选择MulticlassClassificationEvaluator。评估器里默认的评估准则可通过setMetricName( )方法重写。

```
# 创建交叉验证器
>>> cv = CrossValidator(estimator=lrPipeline,estimatorParamMaps=paramGrid,
                        evaluator=MulticlassClassificationEvaluator(labelCol="indexedLabel", predictionCol="prediction"), numFolds=3)

# 在训练数据上进行交叉验证
>>> cvModel = cv.fit(trainingData)
```

9.11.2 用交叉验证选择模型



第5步：调动transform( )方法对测试数据进行预测，并输出结果及精度。

```
# 使用交叉验证模型进行预测
>>> lrPredictions = cvModel.transform(testData)

# 显示前20个预测结果
>>> lrPredictions.select("predictedLabel", "label", "features", "probability").show(20)

# 遍历并输出每个预测结果
```

9.11.2 用交叉验证选择模型



```
>>> for row in lrPredictions.select("predictedLabel", "label", "features", "probability").co
llect():
                                predictedLabel = row["predictedLabel"]
                                label = row["label"]
                               features = row["features"]
                               prob = row["probability"]
                               print(f"({label}, {features}) --> prob={prob}, predicted Label={predi
ctedLabel}")
(Iris-virginica, [7.2,3.6,6.1,2.5]) --> prob=[9.062689448301l8e-05,0.011707200060683
584,0.9882021730448335], predicted Label=Iris-virginica
……
# 创建多类分类评估器
>>> evaluator = MulticlassClassificationEvaluator(labelCol="indexedLabel", prediction
Col="prediction")

# 计算模型的准确度
>>> lrAccuracy = evaluator.evaluate(lrPredictions)
```

9.11.2 用交叉验证选择模型



第6步：获取最优的逻辑斯谛回归模型，并查看其具体的参数。

```
# 获取最佳模型
>>> bestModel = cvModel.bestModel

# 获取Logistic Regression模型
>>> lrModel = bestModel.stages[2]

# 输出模型的系数、截距、类别数和特征数
>>> print("Coefficients: " + str(lrModel.coefficientMatrix))
Coefficients: DenseMatrix([[-1.06361279,  2.97294569, -0.894346
22, -1.89541913],
             [ 0.35660959, -0.9063756 , -0.11732231, -0.74760836],
             [ 0.40563085, -1.48954087,  1.14993765,  2.96398964]])
>>> print("Intercept: " + str(lrModel.interceptVector))
Intercept: [2.947857806108786 3,3.939648927031084 4,-6.887506
7331398 71]
```

9.11.2 用交叉验证选择模型



```
>>> print("numClasses: " + str(lrModel.numClasses))
numClasses: 3
>>> print("numFeatures: " + str(lrModel.numFeatures))
numFeatures: 4
# 解释模型的regParam参数
>>> print(lrModel.explainParam("regParam"))
regParam: regularization parameter (>= 0). (default: 0.0, current: 0.
01)
# 解释模型的elasticNetParam参数
>>> print(lrModel.explainParam("elasticNetParam"))
elasticNetParam: the ElasticNet mixing parameter, in range [0, 1].
For alpha = 0, the penalty is an L2 penalty. For alpha = 1, it is an L
1 penalty. (default: 0.0, current: 0.2)
```

9.12 本章小结



Spark在机器学习方面的发展非常快，目前已经支持了主流的统计和机器学习算法。MLlib以计算效率高而著称，是一个非常优秀的基于分布式架构的开源机器学习库，得到了业界的认可并被广泛使用。MLlib能有效简化机器学习的工程实践工作，并可被方便地扩展到大规模的数据集上进行模型训练和预测。MLlib包括分类、回归、聚类、协同过滤、降维等通用的机器学习算法和工具，同时，还包括底层的优化原语和高层的管道API。本章首先介绍了MLlib的基本数据类型、机器学习流水线的概念和工作过程等。其次，本章对典型的机器学习算法和操作进行了详细的介绍，包括特征提取、转换和选择操作，以及分类算法、聚类算法、协同过滤算法等。本章演示了逻辑斯谛回归、决策树、K-Means、GMM、ALS等经典机器学习算法在MLlib中的使用方法。最后，本章介绍了模型选择的具体方法。