SparkML 入门

机器学习库指南

1.0 Pipelines 管道

1.1 管道中的主要概念

- DataFrame:ML API使用这个来自Spark SQL的概念作为ML dataset，可以保存多种数据类型。比如：使用不同的列存储文本、特征向量、真实标签和预测结果。
- Transformer:这是个是指一个算法将一个DataFrame transform成另一个DataFrame。也就是训练好的模型。比如：一个ML模型就是一个Transformer能够将一个特征数据的DataFrame转成预测结果的DataFrame。
- Estimator:是指一个操作DataFrame产生Transformer的算法。比如：一个学习算法就是一个Estimator，可以在一个DataFrame上训练得到一个模型。
- Pipeline:一个Pipeline链是将多个Transformer和Estimator组合在一起组成一个ML workflow。
- Parameter:所有的Transformer和Estimator共享一个公共的说明参数的API。
- 保存或加载管道 通常情况下，将模型或管道保存到磁盘供以后使用是值得的。模型的导入导出功能在spark1.6的时候加入了pipeline API。大多数基础transformers和基本ML models都支持。

  可以将训练好的pipeline输出到磁盘保存起来

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

1.2 代码示例

Estimator,Transformer,and Param

```
import org.apache.spark.ml.classification.LogisticRegression
import org.apache.spark.ml.linalg.{Vector,Vectors}
import org.apache.spark.ml.param.ParamMap
import org.apache.spark.sql.Row
//准备数据的格式:(label,features)
//注意使用时候特征命名最好命名为features
//因为在一些算法中是取找特征列根据features列名
val training = spark.createDataFrame(
Seq((1.0, Vectors.dense(0.0, 1.1, 0.1)),
(0.0, Vectors.dense(2.0, 1.0, -1.0)),
(0.0, Vectors.dense(2.0, 1.3, 1.0)),
(1.0, Vectors.dense(0.0, 1.2, -0.5)))
).toDF("label","features")
//创建一个LogisticRegression实例，该实例是一个Estimator
val lr = new LogisticRegression()
//使用setter函数设置参数
```

```
lr.setMaxIter(10).setRegParam(0.01)
//学习一个回归模型，使用存储在lr中的参数
val model1 = lr.fit(training)
//由于model1是一个模型（即Estimator生成的Transformer），我们可以查看它//在fit（）中使用的参数。
//打印参数（名称：值）对，其中名称是此
println("Model 1 was fit usingparameters: " + model1.parent.extractParamMap)
//我们也可以使用ParamMap指定参数，
//它支持几种指定参数的方法。
val paramMap = ParamMap(lr.maxIter ->20).put(lr.maxIter, 30).put(lr.regParam -> 0.1,
lr.threshold -> 0.55)
// 修改输出列名称
val paramMap2 = ParamMap(lr.probabilityCol-> "myProbability")
//还可以结合ParamMaps。
val paramMapCombined = paramMap ++paramMap2
//现在使用paramMapCombined参数学习一个新的模型。
// paramMapCombined覆盖之前通过lr.set*方法设置的所有参数。
val model2 = lr.fit(training,paramMapCombined)
println("Model 2 was fit usingparameters: " + model2.parent.extractParamMap)
//准备测试数据
val test = spark.createDataFrame(Seq((1.0,Vectors.dense(-1.0, 1.5, 1.3)),(0.0,
Vectors.dense(3.0, 2.0, -0.1)),(1.0,Vectors.dense(0.0, 2.2, -1.5)))).toDF("label", "features")
//使用Transformer.transform（）方法对测试数据进行预测。
// LogisticRegression.transform将仅使用“特征”列。
//注意model2.transform（）输出一个'myProbability'列，而不是通常的
//'probability'列，因为之前我们重命名了lr.probabilityCol参数。
model2.transform(test).select("features","label",
"myProbability","prediction").collect().foreach { case Row(features: Vector, label:Double, prob:
Vector, prediction: Double) =>println(s"($features,$label) -> prob=$prob,
prediction=$prediction")}
```

Pipeline

```
import org.apache.spark.ml.{Pipeline,PipelineModel}
import org.apache.spark.ml.classification.LogisticRegression
import org.apache.spark.ml.feature.{HashingTF,Tokenizer}
import org.apache.spark.ml.linalg.Vector
import org.apache.spark.sql.Row
// 准备数据数据类型：(id, text,label).
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")
//设计一个包含三个stage的ML pipeline:tokenizer, hashingTF, and lr.
val tokenizer = newTokenizer().setInputCol("text").setOutputCol("words")
val hashingTF =
newHashingTF().setNumFeatures(1000).setInputCol(tokenizer.getOutputCol).setOutputCol("features")
val lr = newLogisticRegression().setMaxIter(10).setRegParam(0.001)
val pipeline = newPipeline().setStages(Array(tokenizer, hashingTF, lr))
// 调用fit，训练数据
val model = pipeline.fit(training)
```

```
// 可以将训练好的pipeline输出到磁盘
model.write.overwrite().save("/opt/spark-logistic-regression-model")
// 也可以直接将为进行训练的pipeline写到文件
pipeline.write.overwrite().save("/opt/unfit-lr-model")
// 加载到出来
val sameModel = PipelineModel.load("/opt/spark-logistic-regression-model")
// (id, text) 这个格式未打标签的数据进行测试
val test = spark.createDataFrame(Seq((4L,"spark i j k"),(5L, "l m n"),(6L, "spark hadoopspark"),
(7L, "apache hadoop"))).toDF("id","text")
// 在测试集上进行预测
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")}
```

2.0 提取、转换、特征选择

2.1 特征抽取

2.1.1 TF-IDF

- 词频-逆文本频率（TF-IDF）是在文本挖掘中广泛使用的特征向量化方法，反映了语料库中一个词对文档的重要性（语料库中的其中一份文档的重要程度）。 用t表示一个单词，用d表示文档，用D表示语料库。词频率TF（t，d）是单词t出现在文档d中的次数，而文档频率DF（t，D）是单词t在语料库D中的频率（出现单词t的文档的次数）。 如果我们只使用词频率来测量重要性，那么很容易过分强调出现得非常频繁但是携带关于文档的信息很少的单词，例如， “a”，“the”和“of”。 如果一个单词在语料库中经常出现， 则意味着它不携带关于特定文档的特殊信息。 逆文本频率是一个单词提供多少信息的数值度量：

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

  $$TF_w, D_i = \frac{Count(w)}{|D_i|}$$

  其中| D | 是语料库中文档的总数。 由于使用对数，如果一个单词出现在所有文档中，则其IDF值变为0。如果一个词越常见，那么分母就越大，逆文档频率就越小越接近0。 分母之所以要加1，是为了避免分母为0（即所有文档都不包含该单词的情况下出现0）。TF-IDF度量仅仅是TF和IDF的乘积：

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

- TF-IDF 举例

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

	包含该词的文档数（亿）	IDF	TF-IDF
中国	62.3	0.603	0.0121
蜜蜂	0.484	2.713	0.0543
养殖	0.973	2.410	0.0482

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

2.1.2 Word2Vec

- Word2Vec是一个估计器，它接受代表文档的单词序列，并训练一个Word2VecModel。 该模型将每个词映射到唯一的固定大小的向量。 Word2VecModel使用文档中所有单词的平均值将每个文档转换为向量; 此向量可以用作预测，文档相似性计算等的特征。

2.1.3 CountVectorizer

- CountVectorizer和CountVectorizerModel旨在帮助将文本文档的集合转换为令牌计数的向量。 当一个先验字典不可用时，CountVectorizer可以用作估计器来提取词汇表，并生成CountVectorizerModel。 该模型为词汇表上的文档生成稀疏表示，然后可以将其传递给其他算法，如LDA。

  在拟合过程中，CountVectorizer将选择通过语料库的词频率排序的顶部vocabSize词。 可选参数minDF还通过指定词汇必须出现在词汇表中的文档的最小数量（或小于1.0）来影响拟合过程。 另一个可选的二进制切换参数控制输出向量。 如果设置为true，则所有非零计数都设置为1.这对于模拟二进制计数而不是整数计数的离散概率模型特别有用。

2.1.4 FeatureHasher

2.2 特征转换

2.2.1 Tokenizer

- **Tokenizer /分词器**
  标记化是获取文本（例如句子）并将其分解成单个术语（通常是单词）的过程。 一个简单的Tokenizer类提供了这个功能 .
  RegexTokenizer允许基于正则表达式（regex）匹配的更高级的标记化。 默认情况下，参数“pattern”（regex，默认：“\ s +”）用作分隔符以分隔输入文本。 或者，用户可以将参数“gap”设置为false，指示正则表达式“模式”表示“令牌”，而不是分割间隙，并且找到所有匹配的出现作为令牌化结果。

2.2.2 StopWordsRemover

- 停止词是应该从输入中排除的词，通常是因为词频繁出现并且不具有任何意义。 StopWordsRemover接受字符串序列（例如Tokenizer的输出）作为输入，并从输入序列中删除所有停止词。 停用词列表由stopWords参

数指定。 某些语言的默认停用词可通过调用StopWordsRemover.loadDefaultStopWords（语言）访问，其中可用的选项是“danish”，“dutch”，“english”，“finnish”，“french”，“german”，“hungarian” “意大利语”，“挪威语”，“葡萄牙语”，“俄语”，“西班牙语”，“瑞典语”和“土耳其语”。 布尔参数caseSensitive指示匹配是否区分大小写（默认为false）。

2.2.3 n-gram

- n-gram是对于某个整数n的n个令牌（通常为字）的序列。 NGram类可以用于将输入特征转换为n-gram。 NGram将字符串序列（例如，Tokenizer的输出）作为输入。 参数n用于确定每个n-gram中的项的数量。 输出将包括一个n-gram序列，其中每个n-gram由n个连续字的空格分隔的字符串表示。 如果输入序列包含少于n个字符串，则不会生成输出。

2.2.4 Binarizer

- 二值化是将数字特征阈值为二进制（0/1）特征的过程。 Binarizer接受通用参数inputCol和outputCol以及二进制阈值。 大于阈值的特征值被二进制化为1.0; 等于或小于阈值的值被二值化为0.0。 inputCol支持Vector和Double类型。

2.2.5 PCA

- PCA是使用正交变换将可能相关的变量的观察值的集合转换成称为主分量的线性不相关变量的值的集合的统计过程。 PCA类训练模型使用PCA将向量投影到低维空间。

2.2.6 PolynomialExpansion

- 多项式展开是将特征扩展到多项式空间的过程，该多项式空间由原始尺寸的n度组合表示。 PolynomialExpansion类提供此功能。

2.2.7 Discrete Cosine Transform (DCT)

- 离散余弦变换将时域中的长度N实值序列变换为频域中的另一长度N实值序列。 DCT类提供此功能，实现DCT-II并且通过1/√2/缩放结果，使得用于变换的表示矩阵是统一的。 没有偏移被应用于变换序列（例如，变换序列的第0个元素是第0个DCT系数而不是第N / 2个）。

2.2.8 StringIndexer

- StringIndexer将标签的字符串列编码为标签索引列。 索引位于[0，numLabels），按标签频率排序，因此最常见的标签获取索引0.如果输入列是数字，我们将其转换为字符串并索引字符串值。 当下游管道组件（如Estimator或Transformer）使用此字符串索引标签时，必须将组件的输入列设置为此字符串索引的列名称。 在许多情况下，可以使用setInputCol设置输入列。

2.2.9 IndexToString

- 对称到StringIndexer，IndexToString将一列标签索引映射回包含原始标签的列作为字符串。 一个常见的用例是使用StringIndexer从标签生成索引，使用这些索引训练模型，并从IndexToString的预测索引列中检索原始标签。 但是，您可以自由提供您自己的标签。

2.2.10 OneHotEncoder

- 独热编码将一列标签索引映射到二进制向量列，最多具有单个一值。 此编码允许期望连续特征（例如逻辑回归）的算法使用分类特征。

2.2.11 VectorIndexer

- 主要作用：提高决策树或随机森林等ML方法的分类效果。 VectorIndexer是对数据集特征向量中的类别（离散值）特征（index categorical features categorical features ）进行编号。 它能够自动判断那些特征是离散值型的特征，并对他们进行编号，具体做法，它执行以下操作： 1、获取Vector类型的输入列和参数maxCategories。 2、基于不同值的数量确定哪些特征应是分类的，其中最多maxCategories的特征被声明为分类。 3、为每个分类特征计算基于0的类别索引。 4、索引分类特征并将原始特征值转换为索引。 5、索引分类特征允许诸如决策树和树集合等算法适当地处理分类特征，从而提高性能。

```
//定义输入输出列和最大类别数为5，某一个特征
//（即某一列）中多于5个取值视为连续值
VectorIndexerModel featureIndexerModel=new VectorIndexer()
                .setInputCol("features")
                .setMaxCategories(5)
                .setOutputCol("indexedFeatures")
                .fit(rawData);
//加入到Pipeline
Pipeline pipeline=new Pipeline()
                .setStages(new PipelineStage[]
                        {labelIndexerModel,
                        featureIndexerModel,
                        dtClassifier,
                        converter});
pipeline.fit(rawData).transform(rawData).select("features","indexedFeatures").show(20,false
);
//显示如下的结果：
+-------------------------+-------------------------+
|features                 |indexedFeatures          |
+-------------------------+-------------------------+
|(3,[0,1,2],[2.0,5.0,7.0])|(3,[0,1,2],[2.0,1.0,1.0])|
|(3,[0,1,2],[3.0,5.0,9.0])|(3,[0,1,2],[3.0,1.0,2.0])|
|(3,[0,1,2],[4.0,7.0,9.0])|(3,[0,1,2],[4.0,3.0,2.0])|
|(3,[0,1,2],[2.0,4.0,9.0])|(3,[0,1,2],[2.0,0.0,2.0])|
|(3,[0,1,2],[9.0,5.0,7.0])|(3,[0,1,2],[9.0,1.0,1.0])|
|(3,[0,1,2],[2.0,5.0,9.0])|(3,[0,1,2],[2.0,1.0,2.0])|
|(3,[0,1,2],[3.0,4.0,9.0])|(3,[0,1,2],[3.0,0.0,2.0])|
|(3,[0,1,2],[8.0,4.0,9.0])|(3,[0,1,2],[8.0,0.0,2.0])|
|(3,[0,1,2],[3.0,6.0,2.0])|(3,[0,1,2],[3.0,2.0,0.0])|
|(3,[0,1,2],[5.0,9.0,2.0])|(3,[0,1,2],[5.0,4.0,0.0])|
+-------------------------+-------------------------+
结果分析：特征向量包含3个特征，即特征0，特征1，特征2。如Row=1,对应的特征分别是2.0,5.0,7.0.被转换为2.0,1.0,1.0。
我们发现只有特征1，特征2被转换了，特征0没有被转换。这是因为特征0有6中取值（2，3，4，5，8，9），多于前面的设置setMaxCategories(5)
，因此被视为连续值了，不会被转换。
特征1中，（4，5，6，7，9）-->(0,1,2,3,4,5)
特征2中，  (2,7,9)-->(0,1,2)

输出DataFrame格式说明（Row=1）：
3个特征 特征0，1，2       转换前的值
|(3,     [0,1,2],       [2.0,5.0,7.0])
3个特征 特征1，1，2        转换后的值
|(3,     [0,1,2],       [2.0,1.0,1.0])|
```

2.2.12 Normalizer

- 交互是一个变换器，它使用向量或双值列，并生成单个向量列，其中包含每个输入列的一个值的所有组合的乘积。例如，如果您有两个向量类型列，每个列有3个维度作为输入列，那么您将获得一个9维向量作为输出列。

```
import org.apache.spark.ml.feature.Interaction
import org.apache.spark.ml.feature.VectorAssembler
val df = spark.createDataFrame(Seq(
  (1, 1, 2, 3, 8, 4, 5),
  (2, 4, 3, 8, 7, 9, 8),
  (3, 6, 1, 9, 2, 3, 6),
  (4, 10, 8, 6, 9, 4, 5),
  (5, 9, 2, 7, 10, 7, 3),
  (6, 1, 1, 4, 2, 8, 4)
)).toDF("id1", "id2", "id3", "id4", "id5", "id6", "id7")
val assembler1 = new VectorAssembler().
  setInputCols(Array("id2", "id3", "id4")).
  setOutputCol("vec1")
val assembled1 = assembler1.transform(df)
val assembler2 = new VectorAssembler().
  setInputCols(Array("id5", "id6", "id7")).
  setOutputCol("vec2")
val assembled2 = assembler2.transform(assembled1).select("id1", "vec1", "vec2")
val interaction = new Interaction()
  .setInputCols(Array("id1", "vec1", "vec2"))
  .setOutputCol("interactedCol")
val interacted = interaction.transform(assembled2)
interacted.show(truncate = false)
```

2.2.13 StandardScaler

- StandardScaler又叫零均值规范化。StandardScaler变换向量行的数据集，规范每个特征以具有单位标准偏差和/或零均值。 它需要参数： withStd：默认为True。 将数据缩放到单位标准偏差。 withMean：默认为False。 在缩放之前用平均值居中数据。 它将构建密集输出，因此在应用于稀疏输入时要小心。 StandardScaler是一个估计器，可以适合数据集以产生一个StandardScalerModel; 这相当于计算摘要统计。 然后，模型可以将数据集中的**向量列**变换为具有单位标准偏差和/或零均值特征。 请注意，如果要素的标准偏差为零，则该要素的向量中将返回默认值0.0。

2.2.14 MinMaxScaler

- 即最大-最小规范化，MinMaxScaler变换矢量行的数据集，将每个要素重新缩放到特定范围（通常为[0，1]）。 它需要参数： min：默认值为0.0。 转换后的下边界，由所有特征共享。 max：1.0。 转换后的上界，由所有特征共享。 MinMaxScaler计算数据集的汇总统计信息，并生成MinMaxScalerModel。 然后，模型可以单独变换每个特征，使得它在给定范围内。
  注意：(1)最大最小值可能受到离群值的左右。(2)零值可能会转换成一个非零值，因此稀疏矩阵将变成一个稠密矩阵。

2.2.15 MaxAbsScaler

- MaxAbsScaler转换Vector行的数据集，通过划分每个要素中的最大绝对值，将每个要素重新缩放到range [-1，1]。 它不会移动/居中数据，因此不会破坏任何稀疏性。MaxAbsScaler计算数据集上的汇总统计信息，并生成MaxAbsScalerModel。 然后，模型可以将每个要素单独转换为range [-1，1]。

2.2.16 Bucketizer

- Bucketizer（连续数据离散化到指定的范围区间）
- Bucketizer将一列连续要素转换为一列要素桶，其中的桶由用户指定。它需要一个参数： split：用于将连续要素映射到bucket的参数。对于n + 1分割，有n个桶。由分割x，y定义的桶保存除了最后一个桶之外的范围[x，y）中的值，其还包括y。分割应严格增加。必须明确提供-inf，inf的值以涵盖所有Double值;否则，指定拆分之外的值将被视为错误。分割的两个示例是Array（Double.NegativeInfinity，0.0,1.0，Double.PositiveInfinity）和Array（0.0,1.0,2.0）。 注意，如果你不知道目标列的上限和下限，你应该添加Double.NegativeInfinity和Double.PositiveInfinity作为分割的边界，以防止潜在的Bucketizer边界异常。还要注意，您提供的分割必须严格按升序，即s0 <s1 <s2 <... <sn。

2.2.17 ElementwiseProduct

- ElementwiseProduct使用元素级乘法将每个输入向量乘以提供的“权重”向量。 换句话说，它通过标量乘法器来缩放数据集的每一列。 这表示输入向量v和变换向量w之间的Hadamard乘积，以产生结果向量。

2.2.18 SQLTransformer

- SQLTransformer实现由SQL语句定义的变换。 目前我们只支持SQL语法，如“SELECT ... FROM **THIS** ...”，其中“**THIS**”表示输入数据集的基础表。 select子句指定要在输出中显示的字段，常量和表达式，并且可以是Spark SQL支持的任何select子句。 用户还可以使用Spark SQL内置函数和UDF操作这些选定的列。 例如，SQLTransformer支持以下语句：
  - `SELECT a, a + b AS a_b FROM __THIS__`
  - `SELECT a, SQRT(b) AS b_sqrt FROM __THIS__ where a > 5`
  - `SELECT a, b, SUM(c) AS c_sum FROM __THIS__ GROUP BY a, b`

2.2.19 VectorAssembler

- VectorAssembler是一个变换器，将给定的列列表组合成一个单一的向量列。 它有用于将由不同特征变换器生成的原始特征和特征组合成单个特征向量，以便训练ML模型，如逻辑回归和决策树。 VectorAssembler接受以下输入列类型：所有数字类型，布尔类型和向量类型。 在每一行中，输入列的值将按指定的顺序连接到向量中。

2.2.20 QuantileDiscretizer

- 分位树为数离散化，和Bucketizer（分箱处理）一样也是：将连续数值特征转换为离散类别特征。实际上Class QuantileDiscretizer extends （继承自） Class（Bucketizer）。
  - 参数1：不同的是这里不再自己定义splits（分类标准），而是定义分几箱(段）就可以了。QuantileDiscretizer自己调用函数计算分位数，并完成离散化。
    -参数2： 另外一个参数是精度，如果设置为0，则计算最精确的分位数，这是一个高时间代价的操作。
  - 另外上下边界将设置为正负无穷，覆盖所有实数范围。

2.3 特征选择

2.3.1 VectorSlicer

- VectorSlicer是一个转换器，**输入**特征向量，**输出**原始特征向量子集。VectorSlicer接收带有特定索引的向量列，通过对这些索引的值进行筛选得到新的向量集。

可接受如下两种索引：

1、整数索引，setIndices()。

2、字符串索引代表向量中特征的名字，此类要求向量列有*AttributeGroup*，因为该工具根据*Attribute*来匹配名字字段。

指定整数或者字符串类型都是可以的。

另外，同时使用整数索引和字符串名字也是可以的。不允许使用重复的特征，所以所选的索引或者名字必须是没有独一的。

注意如果使用名字特征，当遇到空值的时候将会报错。

输出将会首先按照所选的数字索引排序（按输入顺序），其次按名字排序（按输入顺序）。

```
package zhouls.bigdata.DataFeatureSelection
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.ml.attribute.{Attribute, AttributeGroup, NumericAttribute}
import org.apache.spark.ml.feature.VectorSlicer//引入ml里的特征选择的VectorSlicer
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.sql.Row
import org.apache.spark.sql.types.StructType
object VectorSlicer extends App {
    val conf = new SparkConf().setMaster("local").setAppName("VectorSlicer")
    val sc = new SparkContext(conf)
    val sqlContext = new org.apache.spark.sql.SQLContext(sc)
    import sqlContext.implicits._
    //构造特征数组
    val data = Array(Row(Vectors.dense(-2.0, 2.3, 0.0)))
    //为特征数组设置属性名（字段名），分别为f1 f2 f3
    val defaultAttr = NumericAttribute.defaultAttr
    val attrs = Array("f1", "f2", "f3").map(defaultAttr.withName)
    val attrGroup = new AttributeGroup("userFeatures",
attrs.asInstanceOf[Array[Attribute]])
    //构造DataFrame
    val dataRDD = sc.parallelize(data)
    val dataset = sqlContext.createDataFrame(dataRDD,
StructType(Array(attrGroup.toStructField())))
    print("原始特征：")
    dataset.take(1).foreach(println)
    //构造切割器
    var slicer = new VectorSlicer().setInputCol("userFeatures").setOutputCol("features")
    //根据索引号，截取原始特征向量的第1列和第3列
    slicer.setIndices(Array(0,2))
    print("output1: ")
    slicer.transform(dataset).select("userFeatures", "features").first()
    //根据字段名，截取原始特征向量的f2和f3
    slicer = new VectorSlicer().setInputCol("userFeatures").setOutputCol("features")
    slicer.setNames(Array("f2","f3"))
    print("output2: ")
    slicer.transform(dataset).select("userFeatures", "features").first()
    //索引号和字段名也可以组合使用，截取原始特征向量的第1列和f2
    slicer = new VectorSlicer().setInputCol("userFeatures").setOutputCol("features")
    slicer.setIndices(Array(0)).setNames(Array("f2"))
```

```
    print("output3: ")
    slicer.transform(dataset).select("userFeatures", "features").first()
}
```

2.3.2 RFormula

- RFormula通过R模型公式来选择列。支持R操作中的部分操作，包括'~', '.', ':', '+'以及'-'，基本操作如下：
  - 1.~分隔目标和对象
  - 2.+合并对象，"+0"意味着删除空格
  - 3.-删除一个对象，"-1"表示删除空格
  - 4.交互（数值相乘，类别二值化）
  - 5.除了目标列的全部列

  假设a和b为两列：

  1.y ~ a + b表示模型y ~ w0 + w1 * a +w2 * b其中w0为截距，w1和w2为相关系数

  2. y ~a + b + a:b – 1表示模型y ~ w1* a + w2 * b + w3 * a * b，其中w1，w2，w3是相关系数

  RFormula产生一个向量特征列以及一个double或者字符串标签列。如果用R进行线性回归，则对String类型的输入列进行one-hot编码、对数值型的输入列进行double类型转化。如果类别列是字符串类型，它将通过StringIndexer转换为double类型。如果标签列不存在，则输出中将通过规定的响应变量创造一个标签列。

```
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.ml.feature.RFormula;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.RowFactory;
import org.apache.spark.sql.SQLContext;
import org.apache.spark.sql.types.StructField;
import org.apache.spark.sql.types.StructType;
import java.util.Arrays;
import java.util.List;
import static org.apache.spark.sql.types.DataTypes.*;
public class RFormulaDemo {
    public static void main(String[] args){
        SparkConf conf = new SparkConf().setAppName("RFoumula").setMaster("local");
        JavaSparkContext sc = new JavaSparkContext(conf);
        SQLContext sqlContext = new SQLContext(sc);
        List<Row> data = Arrays.asList(
                RowFactory.create(7,"US",18,1.0),
                RowFactory.create(8,"CA",12,0.0),
                RowFactory.create(9,"NZ",15,0.0)
        );
        StructType schema = createStructType(new StructField[]{
                createStructField("id", IntegerType, false),
                createStructField("country", StringType, false),
                createStructField("hour", IntegerType, false),
                createStructField("clicked", DoubleType, false)
        });
        Dataset<Row> dataset = sqlContext.createDataFrame(data,schema);
```

```
        RFormula formula = new RFormula()
                .setFormula("clicked ~ country + hour")
                .setFeaturesCol("features")
                .setLabelCol("label");
        Dataset<Row> output = formula.fit(dataset).transform(dataset);
        output.select("features", "label").show(false);
//        +--------------+-----+
//        |features      |label|
//        +--------------+-----+
//        |[0.0,0.0,18.0]|1.0  |
//        |[1.0,0.0,12.0]|0.0  |
//        |[0.0,1.0,15.0]|0.0  |
//        +--------------+-----+
        sc.stop();
    }
}
```

2.3.3 ChiSqSelector

- ChiSqSelector代表卡方特征选择。它适用于带有类别特征的标签数据。ChiSqSelector根据独立卡方检验，然后选取类别标签主要依赖的特征。它类似于选取最有预测能力的特征。它支持三种特征选取方法：
  - 1.numTopFeatures：通过卡方检验选取最具有预测能力的Top(num)个特征；
  - 2.percentile：类似于上一种方法，但是选取一小部分特征而不是固定(num)个特征；
  - 3.fpr:选择P值低于门限值的特征，这样就可以控制false positive rate来进行特征选择；

    默认情况下特征选择方法是numTopFeatures(50)，可以根据setSelectorType()选择特征选取方法。

  示例：假设我们有一个DataFrame含有id,features和clicked三列，其中clicked为需要预测的目标：

```
id | features              | clicked
---|-----------------------|---------
 7 | [0.0, 0.0, 18.0, 1.0] | 1.0
 8 | [0.0, 1.0, 12.0, 0.0] | 0.0
 9 | [1.0, 0.0, 15.0, 0.1] | 0.0
```

  如果我们使用ChiSqSelector并设置numTopFeatures为1，根据标签clicked，features中最后一列将会是最有用特征：

```
id | features              | clicked | selectedFeatures
---|-----------------------|---------|------------------
 7 | [0.0, 0.0, 18.0, 1.0] | 1.0     | [1.0]
 8 | [0.0, 1.0, 12.0, 0.0] | 0.0     | [0.0]
 9 | [1.0, 0.0, 15.0, 0.1] | 0.0     | [0.1]
```

3.0 分类与回归

3.1 分类

3.1.1 Logistic regression

- 逻辑回归是预测分类问题的流行算法。它是 广义线性模型的一个特例来预测结果的可能性。 在spark.ml逻辑回归中可以使用二项式Logistic回归来预测二分类问题，也可以通过使用多项Logistic回归来预测多分类问题。 使用*family*参数在这两种算法之间进行选择，或者不设置它，让Spark自己推断出正确的值。
    - 通过将*family*参数设置为“多项式”，也可以将多项Logistic回归用于二分类问题。它将产生两个系数的集合和两个intercept。
    - 当在没有intercept的常量非零列的数据集上对LogisticRegressionModel进行拟合时，Spark MLlib为常数非零列输出零系数。此行为与R glmnet相同，但与LIBSVM不同。

3.1.1.1 Binomial logistic regression

3.1.1.2 Multinomial logistic regression

3.1.4 Decision tree classifier

3.1.4.1 决策树分类原理

- 决策树是通过一系列规则对数据进行分类的过程。它提供一种在什么条件下会得到什么值的类似规则的方法。决策树分为分类树和回归树两种，分类树对离散变量做决策树，回归树对连续变量做决策树。

  近来的调查表明决策树也是最经常使用的数据挖掘算法，它的概念非常简单。决策树算法之所以如此流行，一个很重要的原因就是使用者基本上不用了解机器学习算法，也不用深究它是如何工作的。直观看上去，决策树分类器就像判断模块和终止块组成的流程图，终止块表示分类结果（也就是树的叶子）。判断模块表示对一个特征取值的判断（该特征有几个值，判断模块就有几个分支）。

如果不考虑效率等，那么样本所有特征的判断级联起来终会将某一个样本分到一个类终止块上。实际上，样本所有特征中有一些特征在分类时起到决定性作用，决策树的构造过程就是找到这些具有决定性作用的特征，根据其决定性程度来构造一个倒立的树--决定性作用最大的那个特征作为根节点，然后递归找到各分支下子数据集中次大的决定性特征，直至子数据集中所有数据都属于同一类。所以，构造决策树的过程本质上就是根据数据特征将数据集分类的递归过程，我们需要解决的第一个问题就是，当前数据集上哪个特征在划分数据分类时起决定性作用。

3.1.4.2 决策树的学习过程

一棵决策树的生成过程主要分为以下3个部分:

- **特征选择**：特征选择是指从训练数据中众多的特征中选择一个特征作为当前节点的分裂标准，如何选择特征有着很多不同量化评估标准标准，从而衍生出不同的决策树算法。
- **决策树生成**： 根据选择的特征评估标准，从上至下递归地生成子节点，直到数据集不可分则停止决策树停止生长。 树结构来说，递归结构是最容易理解的方式。
- **剪枝**：决策树容易过拟合，一般来需要剪枝，缩小树结构规模、缓解过拟合。剪枝技术有预剪枝和后剪枝两种。

3.1.4.3 基于信息论的三种决策树算法

- 划分数据集的最大原则是：使无序的数据变的有序。如果一个训练数据中有20个特征，那么选取哪个做划分依据？这就必须采用量化的方法来判断，量化划分方法有多重，其中一项就是“信息论度量信息分类”。基于信息论的决策树算法有ID3、CART和C4.5等算法，其中C4.5和CART两种算法从ID3算法中衍生而来。

  CART和C4.5支持数据特征为连续分布时的处理，主要通过使用二元切分来处理连续型变量，即求一个特定的值-分裂值：特征值大于分裂值就走左子树，或者就走右子树。这个分裂值的选取的原则是使得划分后的子树中的“混乱程度”降低，具体到C4.5和CART算法则有不同的定义方式。

ID3算法由Ross Quinlan发明，建立在“奥卡姆剃刀”的基础上：越是小型的决策树越优于大的决策树（be simple简单理论）。**ID3算法中根据信息论的信息增益评估和选择特征**，每次选择信息增益最大的特征做判断模块。ID3算法可用于划分标称型数据集，没有剪枝的过程，为了去除过度数据匹配的问题，可通过裁剪合并相邻的无法产生大量信息增益的叶子节点（例如设置信息增益阀值）。使用信息增益的话其实是有一个缺点，那就是它偏向于具有大量值的属性--就是说在训练集中，某个属性所取的不同值的个数越多，那么越有可能拿它来作为分裂属性，而这样做有时候是没有意义的，另外ID3不能处理连续分布的数据特征，于是就有了C4.5算法。CART算法也支持连续分布的数据特征。

C4.5是ID3的一个改进算法，继承了ID3算法的优点。**C4.5算法用信息增益率来选择属性**，克服了用信息增益选择属性时偏向选择取值多的属性的不足在树构造过程中进行剪枝；能够完成对连续属性的离散化处理；能够对不完整数据进行处理。C4.5算法产生的分类规则易于理解、准确率较高；但效率低，因树构造过程中，需要对数据集进行多次的顺序扫描和排序。也是因为必须多次数据集扫描，C4.5只适合于能够驻留于内存的数据集。

CART算法的全称是Classification And Regression Tree，**采用的是Gini指数（选Gini指数最小的特征s）**作为分裂标准,同时它也是包含后剪枝操作。ID3算法和C4.5算法虽然在对训练样本集的学习中可以尽可能多地挖掘信息，但其生成的决策树分支较大，规模较大。为了简化决策树的规模，提高生成决策树的效率，就出现了根据GINI系数来选择测试属性的决策树算法CART。

3.1.4.4 决策树优缺点

- 决策树适用于数值型和标称型（离散型数据，变量的结果只在有限目标集中取值），能够读取数据集合，提取一些列数据中蕴含的规则。在分类问题中使用决策树模型有很多的优点，决策树计算复杂度不高、便于使用、而且高效，决策树可处理具有不相关特征的数据、可很容易地构造出易于理解的规则，而规则通常易于解释和理解。决策树模型也有一些缺点，比如处理缺失数据时的困难、过度拟合以及忽略数据集中属性之间的相关性等。

3.1.4.5 决策树代码实现

```
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.classification.DecisionTreeClassificationModel
import org.apache.spark.ml.classification.DecisionTreeClassifier
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator
import org.apache.spark.ml.feature.{IndexToString, StringIndexer, VectorIndexer}
// 将以LIBSVM格式存储的数据作为DataFrame加载
val data = spark.read.format("libsvm").load("data/mllib/sample_libsvm_data.txt")

// 索引标签，将元数据添加到标签列
// 适合整个数据集以包括索引中的所有标签
val labelIndexer = new StringIndexer()
  .setInputCol("label")
  .setOutputCol("indexedLabel")
  .fit(data)
// 自动识别分类功能并对其进行索引
val featureIndexer = new VectorIndexer()
  .setInputCol("features")
  .setOutputCol("indexedFeatures")
  .setMaxCategories(4) // features with > 4 distinct values are treated as continuous.
  .fit(data)

// 将数据拆分为训练集和测试集（30%用于测试）
val Array(trainingData, testData) = data.randomSplit(Array(0.7, 0.3))
```

```scala
// 训练决策树模型
val dt = new DecisionTreeClassifier()
  .setLabelCol("indexedLabel")
  .setFeaturesCol("indexedFeatures")

// 将索引标签转换回原始标签
val labelConverter = new IndexToString()
  .setInputCol("prediction")
  .setOutputCol("predictedLabel")
  .setLabels(labelIndexer.labels)

// 管道中的链索引器和树模型
val pipeline = new Pipeline()
  .setStages(Array(labelIndexer, featureIndexer, dt, labelConverter))

// Train model. This also runs the indexers.
// 训练模型
val model = pipeline.fit(trainingData)

// 预测
val predictions = model.transform(testData)

// 选择要显示的示例行
predictions.select("predictedLabel", "label", "features").show(5)

// 选择（预测，真实标签）并计算测试误差
val evaluator = new MulticlassClassificationEvaluator()
  .setLabelCol("indexedLabel")
  .setPredictionCol("prediction")
  .setMetricName("accuracy")
val accuracy = evaluator.evaluate(predictions)
println("Test Error = " + (1.0 - accuracy))

val treeModel = model.stages(2).asInstanceOf[DecisionTreeClassificationModel]
println("Learned classification tree model:\n" + treeModel.toDebugString)
```

3.1.5 Random forest classifier

- 随机森林 是 决策树的集合。随机森林结合了许多决策树，以降低过度拟合的风险。该 `spark.ml` 实现支持使用连续和分类特征的随机森林进行二进制和多类分类以及回归。

3.1.5.1 随机森林的优点:

- 具有极高的准确率
- 随机性的引入，使得随机森林不容易过拟合
- 随机性的引入，使得随机森林有很好的抗噪声能力
- 能处理很高维度的数据，并且不用做特征选择
- 既能处理离散型数据，也能处理连续型数据，数据集无需规范化
- 训练速度快，可以得到变量重要性排序
- 容易实现并行化

3.1.5.2 随机森林的缺点：

- 当随机森林中的决策树个数很多时，训练时需要的空间和时间会较大
- 随机森林模型还有许多不好解释的地方，有点算个黑盒模型

3.1.5.3 随机森林的构建过程

- 从原始训练集中使用Bootstraping方法随机有放回采样选出m个样本，共进行n_tree次采样，生成n_tree个训练集
- 对于n_tree个训练集，我们分别训练n_tree个决策树模型
- 对于单个决策树模型，假设训练样本特征的个数为n，那么每次分裂时根据信息增益/信息增益比/基尼指数选择最好的特征进行分裂
- 每棵树都一直这样分裂下去，直到该节点的所有训练样例都属于同一类。在决策树的分裂过程中不需要剪枝
- 将生成的多棵决策树组成随机森林。对于分类问题，按多棵树分类器投票决定最终分类结果；对于回归问题，由多棵树预测值的均值决定最终预测结果

3.1.5.4 随机森林代码实现

```
package org.apache.spark.examples.ml
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.classification.{RandomForestClassificationModel,
RandomForestClassifier}
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator
import org.apache.spark.ml.feature.{IndexToString, StringIndexer, VectorIndexer}
import org.apache.spark.sql.SparkSession
//TODO:随机森林实现分类
object RandomForestClassifierExample {
  def main(args: Array[String]): Unit = {
    val spark = SparkSession
      .builder
      .appName("RandomForestClassifierExample")
      .getOrCreate()
    // 加载解析数据文件,转换为DataFrame
    val data = spark.read.format("libsvm").load("data/mllib/sample_libsvm_data.txt")
    // 索引标签,添加元数据到列上
    // 在数据集上进行训练,包括索引标签
    val labelIndexer = new StringIndexer()
      .setInputCol("label")
      .setOutputCol("indexedLabel")
      .fit(data)
// 自动识别分类功能并对其进行索引。
// Set maxCategories so features with > 4 distinct values are treated as continuous
// 设置最大分类数 > 4 个连续且不同的特征值
    val featureIndexer = new VectorIndexer()
      .setInputCol("features")
      .setOutputCol("indexedFeatures")
      .setMaxCategories(4)
      .fit(data)
  //切分数据为训练集和测试机,训练集为70% ,测试集为30%
    val Array(trainingData, testData) = data.randomSplit(Array(0.7, 0.3))
    // 训练随机森林模型
    val rf = new RandomForestClassifier()

      .setLabelCol("indexedLabel")
```

```
      .setFeaturesCol("indexedFeatures")
      .setNumTrees(10)
    // 索引标签转换为原来的标签
    val labelConverter = new IndexToString()
      .setInputCol("prediction")
      .setOutputCol("predictedLabel")
      .setLabels(labelIndexer.labels)
    // 通过管道将数据与随机森林模型进行\特征索引\标签转换等串联在一起
    val pipeline = new Pipeline()
      .setStages(Array(labelIndexer, featureIndexer, rf, labelConverter))
    // 训练模型
    val model = pipeline.fit(trainingData)
    // Make predictions.
    val predictions = model.transform(testData)
    // 选择5行进行展示
    predictions.select("predictedLabel", "label", "features").show(5)
    // 选择（预测值，真实值）进行计算测试数据的错误率.
    val evaluator = new MulticlassClassificationEvaluator()
      .setLabelCol("indexedLabel")
      .setPredictionCol("prediction")
      .setMetricName("accuracy")
    val accuracy = evaluator.evaluate(predictions)
    println("Test Error = " + (1.0 - accuracy))
    val rfModel = model.stages(2).asInstanceOf[RandomForestClassificationModel]
    println("Learned classification forest model:\n" + rfModel.toDebugString)
    // $example off$
    spark.stop()
  }
}
// scalastyle:on println
```

3.1.6 Gradient-boosted tree classifier

3.1.6.1 梯度提升树简介

- 梯度提升树（GBT）是决策树的集合。 GBT迭代地训练决策树以便使损失函数最小化。 spark.ml实现支持GBT用于二进制分类和回归，可以使用连续和分类特征。

3.1.6.2 梯度提升树代码实现

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.Dataset
import org.apache.spark.sql.Row
import org.apache.spark.sql.DataFrame
import org.apache.spark.sql.Column
import org.apache.spark.sql.DataFrameReader
import org.apache.spark.rdd.RDD
import org.apache.spark.sql.catalyst.encoders.ExpressionEncoder
import org.apache.spark.sql.Encoder
import org.apache.spark.sql.DataFrameStatFunctions

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

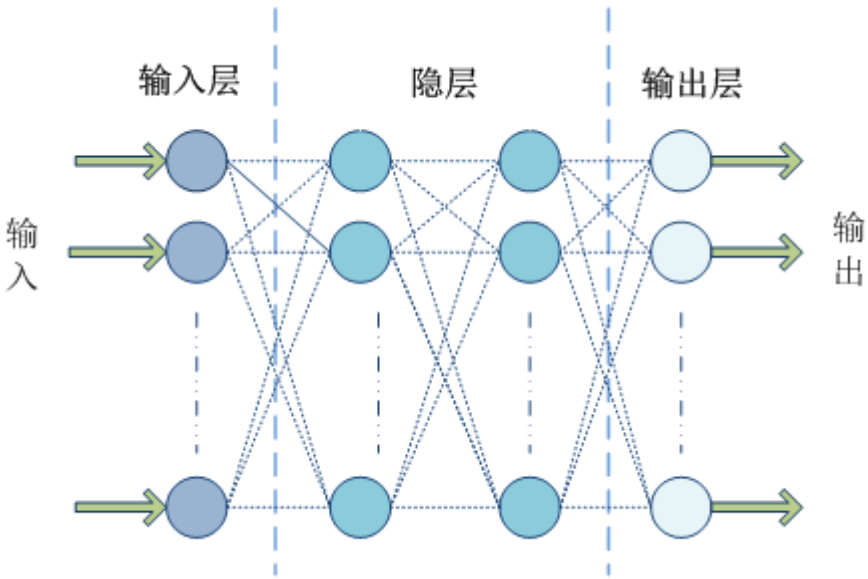
```
import org.apache.spark.ml.linalg.Vectors
import org.apache.spark.ml.feature.{ IndexToString, StringIndexer, VectorIndexer }
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.regression.{ GBTRegressionModel, GBTRegressor }
import org.apache.spark.ml.evaluation.RegressionEvaluator
import org.apache.spark.ml.tuning.{ ParamGridBuilder, CrossValidator }
//导入数据源
val spark = SparkSession.builder().appName("Spark Gradient-boosted tree
regression").config("spark.some.config.option", "some-value").getOrCreate()
// 用于将RDDS转换为数据框的隐式转换
import spark.implicits._

val dataList: List[(Double, String, Double, Double, String, Double, Double, Double, Double)] =
List(
      (0, "male", 37, 10, "no", 3, 18, 7, 4),
      (0, "female", 27, 4, "no", 4, 14, 6, 4),
      (0, "female", 32, 15, "yes", 1, 12, 1, 4),
      (0, "male", 57, 15, "yes", 5, 18, 6, 5),
      (0, "male", 22, 0.75, "no", 2, 17, 6, 3),
      (0, "female", 32, 1.5, "no", 2, 17, 5, 5),
      (7, "female", 32, 10, "yes", 2, 18, 5, 4),
      (2, "male", 32, 10, "yes", 2, 17, 6, 5),
      (2, "male", 22, 7, "yes", 3, 18, 6, 2),
      (1, "female", 32, 15, "yes", 3, 14, 1, 5))
val data = dataList.toDF("affairs", "gender", "age", "yearsmarried", "children",
"religiousness", "education", "occupation", "rating")
data.createOrReplaceTempView("data")
// 字符类型转换成数值
val labelWhere = "affairs as label"
val genderWhere = "case when gender='female' then 0 else cast(1 as double) end as gender"
val childrenWhere = "case when children='no' then 0 else cast(1 as double) end as children"
val dataLabelDF = spark.sql(s"select $labelWhere,
$genderWhere,age,yearsmarried,$childrenWhere,religiousness,education,occupation,rating from
data")
val featuresArray = Array("gender", "age", "yearsmarried", "children", "religiousness",
"education", "occupation", "rating")
// 字段转换成特征向量
val assembler = new VectorAssembler().setInputCols(featuresArray).setOutputCol("features")
val vecDF: DataFrame = assembler.transform(dataLabelDF)
vecDF.show(10, truncate = false)
// 将数据分为训练和测试集（30%进行测试）
val Array(trainingDF, testDF) = vecDF.randomSplit(Array(0.7, 0.3))
// 具有大于5个不同的连续的特征值
val featureIndexer = new
VectorIndexer().setInputCol("features").setOutputCol("indexedFeatures").setMaxCategories(5)
// 训练GBT模型
val gbt = new
GBTRegressor().setLabelCol("label").setFeaturesCol("indexedFeatures").setImpurity("variance").se
tLossType("squared").setMaxIter(100).setMinInstancesPerNode(100)
// Chain indexer and GBT in a Pipeline.

val pipeline = new Pipeline().setStages(Array(featureIndexer, gbt))
```

```
// Train model. This also runs the indexer.
val model = pipeline.fit(trainingDF)
// 做出预测
val predictions = model.transform(testDF)
// 预测样本展示
predictions.select("prediction", "label", "features").show(20,false)
// 选择（预测标签，实际标签），并计算测试误差。
val evaluator = new
RegressionEvaluator().setLabelCol("label").setPredictionCol("prediction").setMetricName("rmse")
val rmse = evaluator.evaluate(predictions)
println("Root Mean Squared Error (RMSE) on test data = " + rmse)
val gbtModel = model.stages(1).asInstanceOf[GBTRegressionModel]
println("Learned regression GBT model:\n" + gbtModel.toDebugString)
```

3.1.7 Multilayer perceptron classifier

3.1.7.1 多层感知机模型简介

- 多层感知器 (MLP, Multilayer Perceptron) 是一种多层的前馈神经网络模型，所谓前馈型神经网络，指其从输入层开始只接收前一层的输入，并把计算结果输出到后一层，并不会给前一层有所反馈，整个过程可以使用有向无环图来表示。该类型的神经网络由三层组成，分别是输入层 (Input Layer)，一个或多个隐层 (Hidden Layer)，输出层 (Output Layer)，如图所示：



- Spark ML 在 1.5 版本后提供一个使用 BP(反向传播，Back Propagation) 算法训练的多层感知器实现，BP 算法的学习目的是对网络的连接权值进行调整，使得调整后的网络对任一输入都能得到所期望的输出。BP 算法名称里的反向传播指的是该算法在训练网络的过程中逐层反向传递误差，逐一修改神经元间的连接权值，以使网络对输入信息经过计算后所得到的输出能达到期望的误差。Spark 的多层感知器隐层神经元使用 sigmoid 函数作为激活函数，输出层使用的是 softmax 函数。

  Spark 的多层感知器分类器 (MultilayerPerceptronClassifer) 支持以下可调参数:

  - featuresCol:输入数据 DataFrame 中指标特征列的名称。
  - labelCol：输入数据 DataFrame 中标签列的名称。

- layers:这个参数是一个整型数组类型，第一个元素需要和特征向量的维度相等，最后一个元素需要训练数据的标签取值个数相等，如 2 分类问题就写 2。中间的元素有多少个就代表神经网络有多少个隐层，元素的取值代表了该层的神经元的个数。例如**val** layers = ArrayInt。
- maxIter：优化算法求解的最大迭代次数。默认值是 100。
- predictionCol:预测结果的列名称。
- tol:优化算法迭代求解过程的收敛阀值。默认值是 1e-4。不能为负数。
- blockSize:该参数被前馈网络训练器用来将训练样本数据的每个分区都按照 blockSize 大小分成不同组，并且每个组内的每个样本都会被叠加成一个向量，以便于在各种优化算法间传递。该参数的推荐值是 10-1000，默认值是 128。

算法的返回是一个 MultilayerPerceptronClassificationModel 类实例。

3.1.8 One-vs-Rest classifier (a.k.a. One-vs-All)

3.1.9 Naive Bayes

-

```
//TODO: 模型的建立与保存
// 网址 :https://blog.csdn.net/bianenze/article/details/76449178
import org.apache.spark.ml.classification.NaiveBayes
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator
import org.apache.spark.ml.feature.{HashingTF, IDF, LabeledPoint, Tokenizer}
import org.apache.spark.ml.linalg.{Vector, Vectors}
import org.apache.spark.sql.Row
import org.apache.spark.{SparkConf, SparkContext}
object shunfeng {
  case class RawDataRecord(label: String, text: String)
    def main(args : Array[String]) {
      val config = new SparkConf().setAppName("createModel").setMaster("local[4]")
      val sc =new  SparkContext(config)
      val sqlContext = new org.apache.spark.sql.SQLContext(sc)
      //开启RDD隐式转换，利用.toDF方法自动将RDD转换成DataFrame；
      import sqlContext.implicits._
      val TrainDf = sc.textFile("E:\\train.txt").map {
        x =>
           val data = x.split("\t")
           RawDataRecord(data(0),data(1))
       }.toDF()
      val TestDf= sc.textFile("E:\\test.txt").map {
        x =>
          val data = x.split("\t")
          RawDataRecord(data(0),data(1))
      }.toDF()
      //tokenizer分解器，把句子划分为词语
      val TrainTokenizer = new Tokenizer().setInputCol("text").setOutputCol("words")
      val TrainWords = TrainTokenizer.transform(TrainDf)
      val TestTokenizer = new Tokenizer().setInputCol("text").setOutputCol("words")
      val TestWords = TestTokenizer.transform(TestDf)
      //特征抽取，利用TF-IDF
      val TrainHashingTF = new
HashingTF().setInputCol("words").setOutputCol("rawFeatures").setNumFeatures(5000)

      val TrainData = TrainHashingTF.transform(TrainWords)
```

```
      val TestHashingTF = new
HashingTF().setInputCol("words").setOutputCol("rawFeatures").setNumFeatures(5000)
      val TestData = TestHashingTF.transform(TestWords)
      val TrainIdf = new IDF().setInputCol("rawFeatures").setOutputCol("features")
      val TrainIdfmodel = TrainIdf.fit(TrainData)
      val TrainForm = TrainIdfmodel.transform(TrainData)
      val TestIdf = new IDF().setInputCol("rawFeatures").setOutputCol("features")
      val TestIdfModel = TestIdf.fit(TestData)
      val TestForm = TestIdfModel.transform(TestData)
      //把数据转换成朴素贝叶斯格式
      val TrainDF = TrainForm.select($"label",$"features").map {
        case Row(label: String, features: Vector) =>
          LabeledPoint(label.toDouble, Vectors.dense(features.toArray))
      }
      val TestDF = TestForm.select($"label",$"features").map {
          case Row(label: String, features: Vector) =>
            LabeledPoint(label.toDouble, Vectors.dense(features.toArray))
        }
      //建立模型
      val model =new NaiveBayes().fit(TrainDF)
      val predictions = model.transform(TestDF)
      predictions.show()
      //评估模型
      val evaluator = new MulticlassClassificationEvaluator()
        .setLabelCol("label")
        .setPredictionCol("prediction")
        .setMetricName("accuracy")
      val accuracy = evaluator.evaluate(predictions)
      println("准确率:"+accuracy)
      //保存模型
      model.write.overwrite().save("model")
    }
}
```

3.2 回归

- 回归模型属于监督式学习，每个个体都有一个与之相关联的实数标签，并且我们希望在给出用于表示这些实体的数值特征后，所预测出的标签值可以尽可能接近实际值。

  回归算法是试图采用对误差的衡量来探索变量之间的关系的一类算法。回归算法是统计机器学习的利器。在机器学习领域，人们说起回归，有时候是指一类问题，有时候是指一类算法，这一点常常会使初学者有所困惑。常见的回归算法包括：普通最小二乘法（OLS）（Ordinary Least Square），它使用损失函数是平方损失函数（1/2 (w^T x-y)2），简单的预测就是y=wT x，标准的最小二乘回归不使用正则化，这就意味着数据中异常数据点非常敏感，因此，在实际应用中经常使用一定程度的正则化（目的避免过拟合、提供泛化能力）。

3.2.1 Linear regression

- 线性回归

```
import org.apache.spark.ml.regression.LinearRegression
```

```
// 加载训练数据
val training = spark.read.format("libsvm")
  .load("data/mllib/sample_linear_regression_data.txt")
val lr = new LinearRegression()
  .setMaxIter(10)
  .setRegParam(0.3)
  .setElasticNetParam(0.8)
// 训练模型
val lrModel = lr.fit(training)
// 打印回归系数和截距
println(s"Coefficients: ${lrModel.coefficients} Intercept: ${lrModel.intercept}")
// 在训练集上进行模型的总结
val trainingSummary = lrModel.summary
println(s"numIterations: ${trainingSummary.totalIterations}")
println(s"objectiveHistory: [${trainingSummary.objectiveHistory.mkString(",")}]")
trainingSummary.residuals.show()
println(s"RMSE: ${trainingSummary.rootMeanSquaredError}")
println(s"r2: ${trainingSummary.r2}")
```

3.2.2 Generalized linear regression

- 广义线性回归

  与线性回归相比，输出被假设为跟随高斯分布，[广义线性模型]（GLM）是线性模型的规范，其中响应变量Yi遵循[指数族分布]的一些分布。 Spark的GeneralizedLinearRegression界面允许灵活的GLM规范，可用于各种类型的预测问题，包括线性回归，泊松回归，逻辑回归等。 目前在spark.ml中，仅支持指数族分布的一部分

- 以下示例演示了使用高斯响应和身份链接功能训练GLM并提取模型汇总统计信息。

```
import org.apache.spark.ml.regression.GeneralizedLinearRegression
// 加载训练数据集
val dataset = spark.read.format("libsvm")
  .load("data/mllib/sample_linear_regression_data.txt")
val glr = new GeneralizedLinearRegression()
  .setFamily("gaussian")
  .setLink("identity")
  .setMaxIter(10)
  .setRegParam(0.3)
// 训练模型
val model = glr.fit(dataset)
// // 打印回归系数和截距
println(s"Coefficients: ${model.coefficients}")
println(s"Intercept: ${model.intercept}")
//在训练集上进行模型的总结
val summary = model.summary
println(s"Coefficient Standard Errors: ${summary.coefficientStandardErrors.mkString(",")}")
println(s"T Values: ${summary.tValues.mkString(",")}")
println(s"P Values: ${summary.pValues.mkString(",")}")
println(s"Dispersion: ${summary.dispersion}")
println(s"Null Deviance: ${summary.nullDeviance}")
println(s"Residual Degree Of Freedom Null: ${summary.residualDegreeOfFreedomNull}")
```

```
println(s"Deviance: ${summary.deviance}")
println(s"Residual Degree Of Freedom: ${summary.residualDegreeOfFreedom}")
println(s"AIC: ${summary.aic}")
println("Deviance Residuals: ")
summary.residuals().show()
```

3.2.3 Decision tree regression

- 决策树是一种流行的分类和回归方法。有关spark.ml实现的更多信息，请参见[决策树部分。
- 以下示例以LibSVM格式加载数据集，将其拆分为训练和测试集，在第一个数据集上训练，然后对所保留的测试集进行评估。我们使用特征变换器来对分类特征进行索引，将元数据添加到决策树算法可以识别的DataFrame中。

```
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.evaluation.RegressionEvaluator
import org.apache.spark.ml.feature.VectorIndexer
import org.apache.spark.ml.regression.DecisionTreeRegressionModel
import org.apache.spark.ml.regression.DecisionTreeRegressor
// 加载数据
val data = spark.read.format("libsvm").load("data/mllib/sample_libsvm_data.txt")
// 自动识别分类功能并对其进行索引。
// Set maxCategories so features with > 4 distinct values are treated as continuous
// 设置最大分类数 > 4 个连续且不同的特征值
val featureIndexer = new VectorIndexer()
  .setInputCol("features")
  .setOutputCol("indexedFeatures")
  .setMaxCategories(4)
  .fit(data)
// 切分数据为训练集和测试集
val Array(trainingData, testData) = data.randomSplit(Array(0.7, 0.3))
// 训练决策树模型
val dt = new DecisionTreeRegressor()
  .setLabelCol("label")
  .setFeaturesCol("indexedFeatures")
// 通过管道进行链接.
val pipeline = new Pipeline()
  .setStages(Array(featureIndexer, dt))
// 训练模型。它也是一个调度器
val model = pipeline.fit(trainingData)
// 预测
val predictions = model.transform(testData)
// 选择要显示的示例行
predictions.select("prediction", "label", "features").show(5)
//计算测试误差
val evaluator = new RegressionEvaluator()
  .setLabelCol("label")
  .setPredictionCol("prediction")
  .setMetricName("rmse")
val rmse = evaluator.evaluate(predictions)
println("Root Mean Squared Error (RMSE) on test data = " + rmse)

val treeModel = model.stages(1).asInstanceOf[DecisionTreeRegressionModel]
```

```
println("Learned regression tree model:\n" + treeModel.toDebugString)
```

3.2.4 Random forest regression

- 随机森林是一类受欢迎的分类和回归方法。有关spark.ml实现的更多信息，请参见随机林部分。
- 案例：以下示例以LibSVM格式加载数据集，将其拆分为训练和测试集，在第一个数据集上训练，然后对所保留的测试集进行评估。我们使用特征变换器对分类特征进行索引，将元数据添加到基于树的算法可以识别的DataFrame中。我们使用特征变换器对分类特征进行索引，将元数据添加到基于树的算法可以识别的DataFrame中。

```
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.evaluation.RegressionEvaluator
import org.apache.spark.ml.feature.VectorIndexer
import org.apache.spark.ml.regression.{RandomForestRegressionModel, RandomForestRegressor}
// 加载数据。转化为DataFrame
val data = spark.read.format("libsvm").load("data/mllib/sample_libsvm_data.txt")
// 自动识别分类功能并对其进行索引。
// Set maxCategories so features with > 4 distinct values are treated as continuous
// 设置最大分类数 > 4 个连续且不同的特征值
val featureIndexer = new VectorIndexer()
  .setInputCol("features")
  .setOutputCol("indexedFeatures")
  .setMaxCategories(4)
  .fit(data)
//切分数据为训练集和测试集
val Array(trainingData, testData) = data.randomSplit(Array(0.7, 0.3))
// 训练随机森林模型
val rf = new RandomForestRegressor()
  .setLabelCol("label")
  .setFeaturesCol("indexedFeatures")
// 通过pipline将属性索引与随机森林模型进行链接
val pipeline = new Pipeline()
  .setStages(Array(featureIndexer, rf))
// 训练模型
val model = pipeline.fit(trainingData)
// 预测
val predictions = model.transform(testData)
// 选择要显示的示例行
predictions.select("prediction", "label", "features").show(5)
// 计算测试误差.
val evaluator = new RegressionEvaluator()
  .setLabelCol("label")
  .setPredictionCol("prediction")
  .setMetricName("rmse")
val rmse = evaluator.evaluate(predictions)
println("Root Mean Squared Error (RMSE) on test data = " + rmse)
val rfModel = model.stages(1).asInstanceOf[RandomForestRegressionModel]
println("Learned regression forest model:\n" + rfModel.toDebugString)
```

3.2.5 Gradient-boosted tree regression

- 梯度增强树回归梯度增强树（GBT）是使用决策树组合的流行回归方法。有关spark.ml实现的更多信息，请参见GBT部分。
- 案例：GBTR实现 注意：对于此示例数据集，GBTRegressor实际上只需要1次迭代，但这一般不会是真的。

```
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.evaluation.RegressionEvaluator
import org.apache.spark.ml.feature.VectorIndexer
import org.apache.spark.ml.regression.{GBTRegressionModel, GBTRegressor}
// 加载数据，转换为dataFrame
val data = spark.read.format("libsvm").load("data/mllib/sample_libsvm_data.txt")
// Automatically identify categorical features, and index them.
// Set maxCategories so features with > 4 distinct values are treated as continuous.
// 设置最大分类数 > 4 个连续且不同的特征值
val featureIndexer = new VectorIndexer()
  .setInputCol("features")
  .setOutputCol("indexedFeatures")
  .setMaxCategories(4)
  .fit(data)
// 切分数据为测试集和训练集
val Array(trainingData, testData) = data.randomSplit(Array(0.7, 0.3))
// 训练GBT模型
val gbt = new GBTRegressor()
  .setLabelCol("label")
  .setFeaturesCol("indexedFeatures")
  .setMaxIter(10)

// Chain indexer and GBT in a Pipeline.
// pipline中的链式分度器和GBT
val pipeline = new Pipeline()
  .setStages(Array(featureIndexer, gbt))

// 训练模型
val model = pipeline.fit(trainingData)

// 预测
val predictions = model.transform(testData)

// 选择要显示的示例行
predictions.select("prediction", "label", "features").show(5)

// Select (prediction, true label) and compute test error.
val evaluator = new RegressionEvaluator()
  .setLabelCol("label")
  .setPredictionCol("prediction")
  .setMetricName("rmse")
val rmse = evaluator.evaluate(predictions)
println("Root Mean Squared Error (RMSE) on test data = " + rmse)

val gbtModel = model.stages(1).asInstanceOf[GBTRegressionModel]
```

```
println("Learned regression GBT model:\n" + gbtModel.toDebugString)
```

3.2.6 Survival regression

- 生存回归

  在spark.ml中，我们实现了[加速失效时间（AFT）模型]，该模型是用于截尾数据的参数生存回归模型。 它描述了生存时间对数的模型，因此通常将其称为生存分析的对数线性模型。 与为同一目的设计的比例危害模型不同，AFT模型更容易并行化，因为每个实例独立地有助于目标函数。

- 案例

```
import org.apache.spark.ml.linalg.Vectors
import org.apache.spark.ml.regression.AFTSurvivalRegression
val training = spark.createDataFrame(Seq(
  (1.218, 1.0, Vectors.dense(1.560, -0.605)),
  (2.949, 0.0, Vectors.dense(0.346, 2.158)),
  (3.627, 0.0, Vectors.dense(1.380, 0.231)),
  (0.273, 1.0, Vectors.dense(0.520, 1.151)),
  (4.199, 0.0, Vectors.dense(0.795, -0.226))
)).toDF("label", "censor", "features")
val quantileProbabilities = Array(0.3, 0.6)
val aft = new AFTSurvivalRegression()
  .setQuantileProbabilities(quantileProbabilities)
  .setQuantilesCol("quantiles")
val model = aft.fit(training)
// 打印 AFT生存回归的系数，截距和比例参数
println(s"Coefficients: ${model.coefficients}")
println(s"Intercept: ${model.intercept}")
println(s"Scale: ${model.scale}")
model.transform(training).show(false)
```

3.2.7 Isotonic regression

- [保序回归]属于回归算法族。 正则等式回归是一个问题，其中给出了一个有限的实数集Y = y1，y2，...，表示观察到的响应，X = x1，x2，...，xn 要拟合的未知响应值，找到最小化的函数

$$f(x) = \sum_{i=1}^{n} w_i(y_i - x_i)^2$$

  对于满足x1≤x2≤...≤xn的完整订单，其中wiwi是正权重。所得到的函数称为等渗回归，它是唯一的。它可以被视为订单限制下的最小二乘问题。基本等渗回归是最接近原始数据点的单调函数。

  我们实施一个相邻违反算法的池，它使用一种方法来平行等渗回归。训练输入是一个DataFrame，它包含三列标签，特征和重量。另外，IsotonicRegression算法有一个可选参数，称为等渗偏差默认为true。这个参数指定等渗回归是等渗的（单调递增）还是反对（单调递减）。

  训练返回一个可以用于预测已知和未知特征的标签的等渗回归模型。等渗回归的结果被视为分段线性函数。因此，预测规则是：

- 如果预测输入与训练特征完全匹配，则返回相关预测。 如果有多个具有相同特征的预测，则返回其中一个。 哪一个是未定义的（与java.util.Arrays.binarySearch相同）。
- 如果预测输入低于或高于所有训练特征，则返回具有最低或最高特征的预测。 如果存在具有相同特征的多个预测，则分别返回最低或最高。
- 如果预测输入落在两个训练特征之间，则预测被视为分段线性函数，并且根据两个最接近的特征的预测来计算内插值。 如果有多个具有相同特征的值，则使用与前一点相同的规则。

- 案例：

```
import org.apache.spark.ml.regression.IsotonicRegression
// 加载数据
val dataset = spark.read.format("libsvm")
  .load("data/mllib/sample_isotonic_regression_libsvm_data.txt")

// 训练一个保序回归模型
val ir = new IsotonicRegression()
val model = ir.fit(dataset)

println(s"Boundaries in increasing order: ${model.boundaries}\n")
println(s"Predictions associated with the boundaries: ${model.predictions}\n")

// 预测
model.transform(dataset).show()
```

3.3 Linear methods

- 线性方法
- 我们通过L1、L2正则化实现了逻辑回归和线性最小二乘法等常用的线性方法。关于实现和调优的详细信息请参考线性方法指南（RDD-based API ）。

  我们也提供了Elastic net的DataFrame API。它发表在Zou et al, Regularization and variable selection via the elastic net，是L1、L2正则化的混合。在数学上定义为L1和L2正则项的凸组合：

  我们实现流行的线性方法，如逻辑回归和L1或L2正则化的线性最小二乘法。 有关实现和调优的详细信息，请参考基于RDD的API的线性方法指南; 这个信息仍然是有关系的。

  我们还包括一个用于elastic net的DataFrame API，Zou等人提出的L1和L2正则化的混合，通过elastic net进行正则化和可变选择。 在数学上，它被定义为L1和L2正则化项的凸组合：

$$\alpha(\lambda\|\mathbf{w}\|_1) + (1-\alpha)\left(\frac{\lambda}{2}\|\mathbf{w}\|_2^2\right), \alpha \in [0,1], \lambda \geq 0$$

  通过正确设置α，elastic net包含L1和L2正则化作为特殊情况。 例如，如果使用elastic net参数α设置为1来训练线性回归模型，则相当于Lasso模型。 另一方面，如果α设定为0，训练模型减少到 岭回归模型。 我们通过elastic net正规化来实现线性回归和逻辑回归的管道API。

3.4 Decision trees

- 决策树

- 决策树及其合奏是分类和回归机器学习任务的流行方法。决策树被广泛使用，因为它们易于解释，处理分类特征，扩展到多类分类设置，不需要特征缩放，并且能够捕获非线性和特征交互。诸如随机森林和提升等树的集成算法是分类和回归任务的最佳表现者之一。

  spark.ml实现支持二分类、多分类和回归的决策树，使用连续和分类特征。该实现按行分隔数据，允许分布式培训，数百万甚至数十亿个实例。

  用户可以在MLlib决策树指南中找到有关决策树算法的更多信息。该API和原始MLlib决策树API之间的主要区别是：

  - 支持ML管道
  - 决策树的分离与回归分离
  - 使用DataFrame元数据来区分连续和分类特征

  决策树的管道API提供比原始API更多的功能。特别地，对于分类，用户可以得到每个类的预测概率（a.k.a.类条件概率）;为了回归，用户可以获得预测的偏差样本方差。

  树的集成方法【Random Forests （随机森林）和 Gradient-Boosted Trees（渐变树）】将在下面的 树的集成方法部分 中进行描述。

3.4.1 Random forest

- 随机森林是 决策树的集合. 随机森林结合多多个决策树来避免 ***过拟合(overfitting)**** 带来的风险。 `spark机器学习(spark.ml)` 的实现支持持续的(continus)还是分类的(categorical)的特征，支持二元分类和多元分类的随机森林及回归。

Inputs and Outputs（输入与输出）

我们在这里列举了输入和输出(预测)列类型。所有的输出列都是可选的；要排除输出列，就把对应参数设为空字符串。

Input Columns（输入列）

参数名称	类型	默认值	描述
labelCol	Double	"label"	要预测的标签(Label to predict)
featuresCol	Vector	"features"	特征向量(Feature vector)

Output Columns（Predictions）输出列（预测）

参数名称	类型	默认值	描述	注意
predictionCol	Double	"prediction"	预测后的标签(Predicted label)	
rawPredictionCol	Vector	"rawPrediction"	长度向量，类及用于预测的树节点上的训练实例标签数量。	只用于分类
probabilityCol	Vector	"probability"	长度向量，与rawPrediction 相等的类被正则化为多元分布。	只用于分类

3.4.2 Gradient-Boosted Trees (GBTs)

- Gradient-Boosted Trees (GBTs) 是 决策树的集合. GBTs 迭代决策树以最小化 **loss function(损失函数)** 。 `spark机器学习库` 的实现使用 ***持续的**(continus)** 和 ***分类的(categorical**)** 的特征来支持 ***GBTs** ，*以用于* ***二元分类**(binary classification)* 和 ***回归**(regression)* 。

Inputs and Outputs（输入和输出）

我们在这里列举了输入和输出(预测)列类型。所有的输出列都是可选的；要排除输出列，就把对应参数设为空字符串。

Input Columns（输入列）

参数名称	类型(s)	默认值	描述
labelCol	Double	"label"	要预测的标签(Label to predict)
featuresCol	Vector	"features"	特征向量(Feature vector)

注意：GBTClassifier目前只支持 ***二元标签(binary labels.)***

Output Columns（Predictions）输出列（预测）

参数名称	类型(s)	默认值	描述	注意点
predictionCol	Double	"prediction"	Predicted label	

在以后，GBTClassifier将会支持 ***rawPrediction*** 和 ***probability*** `的` 输出列，跟 ***RandomForestClassifier*** 一样。

4.0 聚类

4.1 k-means

- k - means 是其中一个最常用的聚类算法,点到一个集群数据 预定义的集群。 MLlib实现包括并行 变体的 k - means + + 方法 被称为 kmeans | | 。

  `KMeans` 被实现为一个 `估计量` 并生成一个 `KMeansModel` 基本模型。

输入列

参数名称	类型(s)	默认的	描述
featuresCol	Vector	"features"	Feature vector

输出列

参数名称	类型(s)	默认的	描述
predictionCol	Int	"prediction"	Predicted cluster center

- 案例：

```
import org.apache.spark.ml.clustering.KMeans
// 加载数据
val dataset = spark.read.format("libsvm").load("data/mllib/sample_kmeans_data.txt")
// 训练k-means模型
val kmeans = new KMeans().setK(2).setSeed(1L)
val model = kmeans.fit(dataset)
// 评估聚类通过计算在平方误差的总和。
val WSSSE = model.computeCost(dataset)
println(s"Within Set Sum of Squared Errors = $WSSSE")
// 显示结果
println("Cluster Centers: ")
model.clusterCenters.foreach(println)
```

4.2 Latent Dirichlet allocation (LDA)

- LDA实现为一个支持EMLDAOptimizer和OnlineLDAOptimizer的估计器，并生成一个LDAModel作为基本模型。 如果需要，专家用户可以将EMLDAOptimizer生成的LDAModel转换为DistributedLDAModel。

```
import org.apache.spark.ml.clustering.LDA
// 加载数据
val dataset = spark.read.format("libsvm")
  .load("data/mllib/sample_lda_libsvm_data.txt")
// 训练LDA模型
val lda = new LDA().setK(10).setMaxIter(10)
val model = lda.fit(dataset)
val ll = model.logLikelihood(dataset)
val lp = model.logPerplexity(dataset)
println(s"The lower bound on the log likelihood of the entire corpus: $ll")
println(s"The upper bound bound on perplexity: $lp")
// 描述topics
val topics = model.describeTopics(3)
println("The topics described by their top-weighted terms:")
topics.show(false)
// 显示结果
val transformed = model.transform(dataset)
transformed.show(false)
```

4.3 角平分线k - means

- 角平分线k - means是一种分层聚类使用一个 分裂的(或“自上而下”)的方法:所有观测开始在一个集群,将递归地执行 沿着层次结构。平分k - means通常可以比常规的k - means要快得多,但是它通常会产生不同的集群。BisectingKMeans `被实现为一个`估计量`并生成一个`BisectingKMeansModel 基本模型。

- 案例：

```
import org.apache.spark.ml.clustering.BisectingKMeans
// 加载数据
val dataset = spark.read.format("libsvm").load("data/mllib/sample_kmeans_data.txt")
// 训练bisecting k-means模型
val bkm = new BisectingKMeans().setK(2).setSeed(1)
val model = bkm.fit(dataset)
// 评估聚类。
val cost = model.computeCost(dataset)
println(s"Within Set Sum of Squared Errors = $cost")
// 显示结果。
println("Cluster Centers: ")
val centers = model.clusterCenters
centers.foreach(println)
```

4.4 高斯混合模型(GMM)

- 高斯混合模型表示复合分布，其中点从k个高斯子分布中的一个绘出，每个具有其自身的概率。 *spark.ml*实现使用期望最大化算法来给出给定一组样本的最大似然模型。

  *GaussianMixture*作为估计器实现，并生成*GaussianMixtureModel* 作为基本模型。

输入列

参数名称	类型(s)	默认的	描述
featuresCol	Vector	"features"	Feature vector

输出列

参数名称	类型(s)	默认的	描述
predictionCol	Int	"prediction"	预测集群中心
probabilityCol	Vector	"probability"	每个集群的概率

- 案例：

```
import org.apache.spark.ml.clustering.GaussianMixture
// 加载数据
val dataset = spark.read.format("libsvm").load("data/mllib/sample_kmeans_data.txt")
// 训练Gaussian Mixture模型
val gmm = new GaussianMixture()
  .setK(2)
val model = gmm.fit(dataset)
 // 输出混合模型参数模型
for (i <- 0 until model.getK) {
  println(s"Gaussian $i:\nweight=${model.weights(i)}\n" +
      s"mu=${model.gaussians(i).mean}\nsigma=\n${model.gaussians(i).cov}\n")
}
```

5.0 协同过滤(Collaborative filtering)

- 简介

  协作过滤通常用于推荐系统。这些技术旨在填补用户-项目关联矩阵中丢失的条目。**spark.ml** 目前支持基于模型的协同过滤，其中用户和项目通过一小组潜在因素来描述，可用于预测缺失的条目。**spark.ml** 使用 **ALS**（交替最小二乘法）算法来学习这些潜在因素。**spark.ml** 中的实现具有以下参数：

  - ***numBlocks*** 是用户和项目将被分区以便并行化计算的块数（默认值为10）。
  - ***rank*** 是模型中潜在因素的数量（默认为10）。
  - ***maxIter*** 是要运行的最大迭代次数（默认为10）。
  - **implicitPrefs** 指定是使用显式反馈ALS的版本还是用适用于隐式反馈数据集的版本（默认值为 false，这意味着使用显式反馈）。
  - ***alpha*** 是适用于ALS的隐式反馈版本的参数，用于控制偏好观察值的基线置信度（默认为1.0）。
  - **nonnegative** 指定是否对最小二乘使用非负约束（默认为 false ）。

  注意：用于 **ALS** 的基于 **DataFrame**（数据框）的 API 目前仅支持整数的用户和项目id。 用户和项目id的列支持其他数字类型，但是列中id的取值必须在整数值范围内。

- ## Explicit vs. implicit feedback（显式与隐式反馈）

  基于矩阵分解的协同过滤的标准方法是将用户-项目关联矩阵中的元素视为用户对项目的显式偏好，例如用户对电影的评分。在现实世界中的许多用例中，通常只能接触到隐式的反馈（例如浏览，点击，购买，喜欢，分享等）。在 **spark.ml** 中使用基于隐式反馈数据集的协同过滤的方法来处理这些数据。实际上，这种方法不是直接对数据矩阵进行建模，而是将数据视为代表用户行为意愿强度的数字（例如点击的次数或某人累积观看电影的时间）。然后，这些数字与观察到的用户偏好的置信水平相关，而不是给予项目的明确评级。 然后，该模型尝试找到可用于预测用户对项目的预期偏好的潜在因素。

- ## Scaling of the regularization parameter（正则化参数的换算）

  我们通过用户在更新用户因素中产生的评分，或产品在更新产品因素中收到的评分来求解每个最小二乘问题的规则化参数 ***regParam***。 这种方法被命名为“ **ALS-WR** ”（加权正则化交替最小二乘法），并在论文“ **Large-Scale Parallel Collaborative Filtering for the Netflix Prize** ”中进行了讨论。 它使 ***regParam*** 对数据集的规模依赖较少，因此我们可以将从采样子集学到的最佳参数应用于完整数据集，并期望能有相似的表现。

- 案例：在以下示例中，我们从 MovieLens 数据集加载评分数据，每行由用户，电影，评分和时间戳组成。 然后，我们训练一个 **ALS** 模型，默认情况下假定评分是显式的（ ***implicitPrefs*** 是 false ）。 我们通过测量评级预测的 **root-mean-square error**（均方根误差）来评估推荐模型。有关API的更多详细信息，请参阅 ALS Scala 文档。

```
import org.apache.spark.ml.evaluation.RegressionEvaluator
import org.apache.spark.ml.recommendation.ALS
case class Rating(userId: Int, movieId: Int, rating: Float, timestamp: Long)
def parseRating(str: String): Rating = {
  val fields = str.split("::")
  assert(fields.size == 4)
  Rating(fields(0).toInt, fields(1).toInt, fields(2).toFloat, fields(3).toLong)
}
val ratings = spark.read.textFile("data/mllib/als/sample_movielens_ratings.txt")
  .map(parseRating)
  .toDF()
val Array(training, test) = ratings.randomSplit(Array(0.8, 0.2))
// 使用ALS在训练数据上构建推荐模型
val als = new ALS()
  .setMaxIter(5)
  .setRegParam(0.01)
  .setUserCol("userId")
  .setItemCol("movieId")
  .setRatingCol("rating")
val model = als.fit(training)
// 通过计算测试数据上的RMSE来评估模型
val predictions = model.transform(test)
val evaluator = new RegressionEvaluator()
  .setMetricName("rmse")
  .setLabelCol("rating")
  .setPredictionCol("prediction")
val rmse = evaluator.evaluate(predictions)
println(s"Root-mean-square error = $rmse")
```

可以在 Spark repo 中的“examples/src/main/scala/org/apache/spark/examples/ml/ALSExample.scala”中查找完整的示例代码。 如果评分矩阵是来自于另一个信息来源（即从其他信号推断出来），您可以将 ***implicitPrefs*** 设置为 true 以获得更好的结果：

```
val als = new ALS()
  .setMaxIter(5)
  .setRegParam(0.01)
  .setImplicitPrefs(true)
  .setUserCol("userId")
  .setItemCol("movieId")
  .setRatingCol("rating")
```

6.0 模型选择和优化

- ML Tuning: model selection and hyperparameter tuning（ML调优：模型选择和超参数调整）
本节介绍如何使用MLlib的工具来调整ML算法和管道。

内置的交叉-验证和其他工具允许用户优化算法和流水线中的超参数。

6.1 模型选择(a.k.a超参数调整)

- ML中的一个重要任务是模型选择，或使用数据找到给定任务的最佳模型或参数。这也叫调优。可以针对个体估算器（如Logistic回归）或包括多个算法，特征化和其他步骤的整个管道完成调整。用户可以一次调整整个流水线，而不是单独调整管道中的每个元素。
  MLlib支持使用 CrossValidator和TrainValidationSplit等工具进行模型选择。这些工具需要以下项目：
  Estimator（估算器）：算法或管道调整
  ParamMaps集：可供选择的参数，有时称为“参数网格”进行搜索
  Evaluator（评估者）：衡量拟合模型对延伸测试数据有多好的度量
  在高层次上，这些模型选择工具的工作如下：
  他们将输入数据分成单独的培训和测试数据集。
  对于每个（训练，测试）对，他们遍历一组ParamMaps：
  对于每个ParamMap，它们使用这些参数适合Estimator，获得拟合的Model，并使用Evaluator评估Model的性能。
  他们选择由最佳性能参数组合生成的模型。
  评估者可以是回归问题的RegressionEvaluator，二进制数据的BinaryClassificationEvaluator或多类问题的MulticlassClassificationEvaluator。用于选择最佳ParamMap的默认度量可以被这些评估器中的每一个的setMetricName方法覆盖。
  为了帮助构建参数网格，用户可以使用ParamGridBuilder实用程序。

6.2 交叉-验证

- CrossValidator首先将数据集分成一组折叠，这些折叠用作单独的训练和测试数据集。 例如，k = 3倍，CrossValidator将生成3个（训练，测试）数据集对，每个数据集使用2/3的数据进行训练，1/3进行测试。 为了评估一个特定的ParamMap，CrossValidator通过在3个不同的（训练，测试）数据集对上拟合Estimator来计算3个模型的平均评估度量。在确定最佳ParamMap之后，CrossValidator最终使用最好的ParamMap和整个数据集重新拟合Estimator。
- 案例：通过交叉-验证进行模型选择 以下示例演示如何使用CrossValidator从参数网格中进行选择。

  请注意，通过参数网格的交叉验证是昂贵的。 例如，在下面的示例中，参数网格具有3个值，用于hashingTF.numFeatures，2个值用于lr.regParam，CrossValidator使用2个折叠。 这被乘以 (3×2)×2 = 12个不同的模型被训练。 在现实的设置中，尝试更多的参数并使用更多的折叠（k = 3和k = 10是常见的）是常见的。 换句话说，使用CrossValidator可能是非常昂贵的。 然而，它也是一种成熟的方法，用于选择比启发式手动调谐更具统计学意义的参数。

  有关API的详细信息，请参阅`CrossValidator` Scala docs。

```
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.classification.LogisticRegression
import org.apache.spark.ml.evaluation.BinaryClassificationEvaluator
import org.apache.spark.ml.feature.{HashingTF, Tokenizer}
import org.apache.spark.ml.linalg.Vector
import org.apache.spark.ml.tuning.{CrossValidator, ParamGridBuilder}
import org.apache.spark.sql.Row
// Prepare training data from a list of (id, text, label) tuples.
//从（id，text，label）元组列表中准备训练数据。
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),
  (4L, "b spark who", 1.0),
  (5L, "g d a y", 0.0),
  (6L, "spark fly", 1.0),
  (7L, "was mapreduce", 0.0),
  (8L, "e spark program", 1.0),
  (9L, "a e c l", 0.0),
  (10L, "spark compile", 1.0),
  (11L, "hadoop software", 0.0)
)).toDF("id", "text", "label")
// Configure an ML pipeline, which consists of three stages: tokenizer, hashingTF, and lr.
//配置ML管道，它由三个阶段组成：tokenizer，hashingTF和lr。
val tokenizer = new Tokenizer()
  .setInputCol("text")
  .setOutputCol("words")
val hashingTF = new HashingTF()
  .setInputCol(tokenizer.getOutputCol)
  .setOutputCol("features")
val lr = new LogisticRegression()
  .setMaxIter(10)
val pipeline = new Pipeline()
  .setStages(Array(tokenizer, hashingTF, lr))
// 我们使用ParamGridBuilder来构建要搜索的参数网格。
// hasingTF.numFeatures有3个值，lr.regParam有2个值，
// 此网格将有3 x 2 = 6个参数设置供CrossValidator选择。
val paramGrid = new ParamGridBuilder()
  .addGrid(hashingTF.numFeatures, Array(10, 100, 1000))
  .addGrid(lr.regParam, Array(0.1, 0.01))
  .build()
 // 我们现在将Pipeline视为Estimator，将其包装在CrossValidator实例中。
// 这将允许我们共同选择所有Pipeline阶段的参数。
// CrossValidator需要Estimator，一组Estimator ParamMaps和一个Evaluator。
// 注意，此处的求值程序是BinaryClassificationEvaluator及其默认度量标准是 areaUnderROC.
val cv = new CrossValidator()
  .setEstimator(pipeline)
  .setEvaluator(new BinaryClassificationEvaluator)
  .setEstimatorParamMaps(paramGrid)
  .setNumFolds(2)  // Use 3+ in practice
 // 运行交叉验证，并选择最佳参数集。
val cvModel = cv.fit(training)
// 准备测试文档，这些文档是未标记的（id，text）元组。
val test = spark.createDataFrame(Seq(
  (4L, "spark i j k"),
  (5L, "l m n"),
  (6L, "mapreduce spark"),
  (7L, "apache hadoop")
)).toDF("id", "text")
// 对测试文档进行预测。 cvModel使用找到的最佳模型（lrModel）。
cvModel.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")
  }
```

6.3 火车-验证 拆分

- 除了CrossValidator Spark，还提供了用于超参数调整的TrainValidationSplit。 TrainValidationSplit仅对参数的每个组合进行一次评估，而在CrossValidator的情况下，则不是k次。 因此，它较便宜，但在训练数据集不够大时不会产生可靠的结果。与CrossValidator不同，TrainValidationSplit创建一个（训练，测试）数据集对。 它使用trainRatio参数将数据集分成这两个部分。 例如，*trainRatio* = 0.75，TrainValidationSplit将生成训练和测试数据集对，其中75%的数据用于培训，25%用于验证。像CrossValidator一样，TrainValidationSplit最终适合使用最好的ParamMap和整个数据集的Estimator。
- 案例：通过列车验证拆分模型选择

```
import org.apache.spark.ml.evaluation.RegressionEvaluator
import org.apache.spark.ml.regression.LinearRegression
import org.apache.spark.ml.tuning.{ParamGridBuilder, TrainValidationSplit}
// 准备培训和测试数据。
val data = spark.read.format("libsvm").load("data/mllib/sample_linear_regression_data.txt")
val Array(training, test) = data.randomSplit(Array(0.9, 0.1), seed = 12345)
val lr = new LinearRegression()
    .setMaxIter(10)
// 我们使用ParamGridBuilder来构建要搜索的参数网格。
// TrainValidationSplit将尝试所有值的组合，并使用评估程序确定最佳模型。
val paramGrid = new ParamGridBuilder()
  .addGrid(lr.regParam, Array(0.1, 0.01))
  .addGrid(lr.fitIntercept)
  .addGrid(lr.elasticNetParam, Array(0.0, 0.5, 1.0))
  .build()
// 在这种情况下，估计量只是线性回归。
// TrainValidationSplit需要Estimator，一组Estimator ParamMaps和一个Evaluator。
val trainValidationSplit = new TrainValidationSplit()
  .setEstimator(lr)
  .setEvaluator(new RegressionEvaluator)
  .setEstimatorParamMaps(paramGrid)
  //80%的数据将用于训练，剩余的20%用于验证。
  .setTrainRatio(0.8)
// 训练验证拆分，并选择最佳参数集。
val model = trainValidationSplit.fit(training)
// 对测试数据进行预测。 模型是具有最佳表现的参数组合的模型。
model.transform(test)
  .select("features", "label", "prediction")
  .show()
```

基于RDD的API指南

1.0 数据类型

- MLlib支持存储在单个机器上的本地向量和矩阵，以及由一个或多个RDD支持的分布式矩阵。 局部向量和局部矩阵是用作公共接口的简单数据模型。 底层线性代数运算由Breeze提供。 在监督学习中使用的训练示例在MLlib中被称为“标记点”。

1.1.1 Local vector（局部向量）

局部向量具有整数类型和基于0的索引和双类型值，存储在单个机器上。 MLlib支持两种类型的局部向量：密集和稀疏。 密集向量由表示其条目值的双数组支持，而稀疏向量由两个并行数组支持：索引和值。 例如，向量（1.00.03.0）可以密集格式表示为1.00.03.0，或以稀疏格式表示为（3，02，1.03.0），其中3是 矢量的大小。

本地向量的基类是Vector，我们提供了两个实现：`DenseVector` 和 `SparseVector`。 我们建议使用 `Vectors` 中实现的工厂方法来创建本地向量。 有关API的详细信息，请参阅 `Vector` Scala docs 和 `Vectors` Scala docs。 注意：Scala默认导入scala.collection.immutable.Vector，因此您必须明确导入org.apache.spark.mllib.linalg.Vector才能使用MLlib的Vector。

```
import org.apache.spark.mllib.linalg.{Vector, Vectors}
// Create a dense vector (1.0, 0.0, 3.0).
val dv: Vector = Vectors.dense(1.0, 0.0, 3.0)
// Create a sparse vector (1.0, 0.0, 3.0) by specifying its indices and values corresponding to
nonzero entries.
val sv1: Vector = Vectors.sparse(3, Array(0, 2), Array(1.0, 3.0))
// Create a sparse vector (1.0, 0.0, 3.0) by specifying its nonzero entries.
val sv2: Vector = Vectors.sparse(3, Seq((0, 1.0), (2, 3.0)))
```

1.1.2 Labeled point（标示点）

标记点是与标签/响应相关联的局部矢量，密集或稀疏。 在MLlib中，标注点用于监督学习算法。 我们使用双重存储标签，所以我们可以在回归和分类中使用标记点。 对于二进制分类，标签应为0（负）或1（正）。 对于多类分类，标签应该是从零开始的类索引：0，1，2，.... 标记点由事例类 `LabeledPoint` 表示。有关 api 的详细信息, 请参阅 `LabeledPoint` Scala docs。

```
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.regression.LabeledPoint
// Create a labeled point with a positive label and a dense feature vector.
val pos = LabeledPoint(1.0, Vectors.dense(1.0, 0.0, 3.0))
// Create a labeled point with a negative label and a sparse feature vector.
val neg = LabeledPoint(0.0, Vectors.sparse(3, Array(0, 2), Array(1.0, 3.0)))
```

1.1.3 Local matrix（本地矩阵）

本地矩阵具有整数类型的行和列索引和双类型值，存储在单个机器上。 MLlib支持密集矩阵，其入口值以列主序列存储在单个双阵列中，稀疏矩阵的非零入口值以列主要顺序存储在压缩稀疏列（CSC）格式中。 例如，以下密集矩阵

$$\begin{pmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \\ 5.0 & 6.0 \end{pmatrix}$$

存储在具有矩阵大小 `(3, 2)` 的一维数组 `[1.0, 3.0, 5.0, 2.0, 4.0, 6.0]` 中。

```
import org.apache.spark.mllib.linalg.{Matrix, Matrices}
// Create a dense matrix ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))
val dm: Matrix = Matrices.dense(3, 2, Array(1.0, 3.0, 5.0, 2.0, 4.0, 6.0))
// Create a sparse matrix ((9.0, 0.0), (0.0, 8.0), (0.0, 6.0))
val sm: Matrix = Matrices.sparse(3, 2, Array(0, 1, 3), Array(0, 2, 1), Array(9, 6, 8))
```

局部矩阵的基类是Matrix，我们提供了两个实现： `DenseMatrix` 和 `SparseMatrix` 。 我们建议使用 `Matrices` 中实现的工厂方法来创建本地矩阵。 记住，MLlib中的局部矩阵以列主要顺序存储。有关API的详细信息，请参阅 `Matrix` Scala docs 和 `Matrices` Scala docs 。

1.1.4 Distributed matrix（分布矩阵）

分布式矩阵具有长类型的行和列索引和双类型值，分布式存储在一个或多个RDD中。选择正确的格式来存储大型和分布式矩阵是非常重要的。将分布式矩阵转换为不同的格式可能需要全局shuffle，这是相当昂贵的。到目前为止已经实现了四种类型的分布式矩阵。

基本类型称为RowMatrix。 RowMatrix是没有有意义的行索引的行向分布式矩阵，例如特征向量的集合。它由其行的RDD支持，其中每行是局部向量。我们假设RowMatrix的列数不是很大，因此单个本地向量可以合理地传递给驱动程序，也可以使用单个节点进行存储/操作。 IndexedRowMatrix与RowMatrix类似，但具有行索引，可用于标识行和执行连接。 CoordinateMatrix是以坐标 list(COO) 格式存储的分布式矩阵，由其条目的RDD支持。 BlockMatrix是由MatrixBlock的RDD支持的分布式矩阵，它是（Int，Int，Matrix）的元组。 Note: 分布式矩阵的底层RDD必须是确定性的，因为我们缓存矩阵大小。一般来说，使用非确定性RDD可能会导致错误

- RowMatrix RowMatrix是一个面向行的分布式矩阵，没有有意义的行索引，由其行的RDD支持，其中每一行都是一个局部向量。 由于每行由局部向量表示，所以列数受到整数范围的限制，但实际应该要小得多。

  ```
  import org.apache.spark.mllib.linalg.Vector
  import org.apache.spark.mllib.linalg.distributed.RowMatrix
  val rows: RDD[Vector] = ... // an RDD of local vectors
  // Create a RowMatrix from an RDD[Vector].
  val mat: RowMatrix = new RowMatrix(rows)
  // Get its size.
  val m = mat.numRows()
  val n = mat.numCols()
  // QR decomposition
  val qrResult = mat.tallSkinnyQR(true)
  ```

- IndexedRowMatrix IndexedRowMatrix 与RowMatrix类似，但具有有意义的行索引。它由索引行的RDD支持，因此每行都由其索引（长类型）和局部向量表示。

  ```
  import org.apache.spark.mllib.linalg.distributed.{IndexedRow, IndexedRowMatrix, RowMatrix}
  val rows: RDD[IndexedRow] = ... // an RDD of indexed rows
  // Create an IndexedRowMatrix from an RDD[IndexedRow].
  val mat: IndexedRowMatrix = new IndexedRowMatrix(rows)
  // Get its size.
  val m = mat.numRows()
  val n = mat.numCols()
  // Drop its row indices.
  val rowMat: RowMatrix = mat.toRowMatrix()
  ```

IndexedRowMatrix可以从RDD [IndexedRow]实例创建，其中 `IndexedRow` 是一个包装器（Long，Vector）。 IndexedRowMatrix可以通过删除其行索引来转换为RowMatrix。

- CoordinateMatrix `CoordinateMatrix` 是由其条目的RDD支持的分布式矩阵。 每个条目是 (i: Long, j: Long, value: Double) 的元组，其中i是行索引，j是列索引，value是条目值。 只有当矩阵的两个维度都很大并且矩阵非常稀疏时才应使用Coordinate矩阵。

```
import org.apache.spark.mllib.linalg.distributed.{CoordinateMatrix, MatrixEntry}
val entries: RDD[MatrixEntry] = ... // an RDD of matrix entries
// Create a CoordinateMatrix from an RDD[MatrixEntry].
val mat: CoordinateMatrix = new CoordinateMatrix(entries)
// Get its size.
val m = mat.numRows()
val n = mat.numCols()
// Convert it to an IndexRowMatrix whose rows are sparse vectors.
val indexedRowMatrix = mat.toIndexedRowMatrix()
```

- BlockMatrix

  BlockMatrix是由MatrixBlocks的RDD支持的分布式矩阵，其中MatrixBlock是 `((Int, Int), Matrix)` 的元组，其中 `(Int, Int)` 是块的索引，Matrix是子矩阵， 在给定索引处的矩阵大小为rowsPerBlock x colsPerBlock。 BlockMatrix支持诸如添加和乘以另一个BlockMatrix的方法。 BlockMatrix还有一个帮助函数validate，可用于检查BlockMatrix是否正确设置。

```
import org.apache.spark.mllib.linalg.distributed.{BlockMatrix, CoordinateMatrix,
MatrixEntry}
val entries: RDD[MatrixEntry] = ... // an RDD of (i, j, v) matrix entries
// Create a CoordinateMatrix from an RDD[MatrixEntry].
val coordMat: CoordinateMatrix = new CoordinateMatrix(entries)
// Transform the CoordinateMatrix to a BlockMatrix
val matA: BlockMatrix = coordMat.toBlockMatrix().cache()
// Validate whether the BlockMatrix is set up properly. Throws an Exception when it is not
valid.
// Nothing happens if it is valid.
matA.validate()
// Calculate A^T A.
val ata = matA.transpose.multiply(matA)
```

2.0 基本统计

2.1 Summary statistics/概要统计

- 我们提供对 RDD[Vector] 中的向量的统计,采用 *Statistics* 里的函数 *colStats*

  `colStats()` 返回一个 `MultivariateStatisticalSummary 的实例`, 包含列的最大值、最小值、均值、方差和非零的数量以及总数量。阅读 `MultivariateStatisticalSummary` Scala 文档查看 API 细节

```
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.stat.{MultivariateStatisticalSummary, Statistics}
val observations = sc.parallelize(

  Seq(
```

```
    Vectors.dense(1.0, 10.0, 100.0),
    Vectors.dense(2.0, 20.0, 200.0),
    Vectors.dense(3.0, 30.0, 300.0)
  )
)
// 计算概要统计.
val summary: MultivariateStatisticalSummary = Statistics.colStats(observations)
println(summary.mean)  //  每一列的均值
println(summary.variance)  // 所有向量的方差
println(summary.numNonzeros)  // 每列中的非零数
```

2.2 Correlations/相关性

- 计算两个系列（series）数据之间的相关性的数据是在统计学一种常见的操作。在 spark.mllib 我们提供灵活的计算两两之间的相关性的方法。支持计算相关性的方法目前有 Pearson’s and Spearman’s (皮尔森和斯皮尔曼) 的相关性.

  `Statistics类` 提供了计算系列之间相关性的方法。 根据输入类型，两个RDD [Double]或RDD [Vector]，输出分别为Double或相关矩阵。阅读 `Statistics` Scala 文档查看 API 细节

```
import org.apache.spark.mllib.linalg._
import org.apache.spark.mllib.stat.Statistics
import org.apache.spark.rdd.RDD
val seriesX: RDD[Double] = sc.parallelize(Array(1, 2, 3, 3, 5))  // a series
// 必须具有相同数量的分区和基数作为一个series
val seriesY: RDD[Double] = sc.parallelize(Array(11, 22, 33, 33, 555))
// 计算 Pearson's 相关性.输入"spearman" 调用 Spearman's 的计算方法.
// 默认使用 Pearson's 方法.
val correlation: Double = Statistics.corr(seriesX, seriesY, "pearson")
println(s"Correlation is: $correlation")
val data: RDD[Vector] = sc.parallelize(
  Seq(
    Vectors.dense(1.0, 10.0, 100.0),
    Vectors.dense(2.0, 20.0, 200.0),
    Vectors.dense(5.0, 33.0, 366.0))
)  // 注意每个向量为行而不是列
// 计算 Pearson's 相关性的矩阵.输入"spearman" 调用 Spearman's 的计算方法.
// 默认使用 Pearson's 方法.
val correlMatrix: Matrix = Statistics.corr(data, "pearson")
println(correlMatrix.toString)
```

2.3 Stratified sampling/分层采样

- 与spark.mllib中的其他统计功能不同，sampleByKey和sampleByKeyExact可以对键值对的RDD执行分层采样方法。 对于分层采样，键可以被认为是一个标签，该值作为一个特定属性。 例如，key 可以是男人或女人或文档ID，并且相应的 value 可以是人的年龄列表或文档中的单词列表。 sampleByKey方法将类似掷硬币方式来决定观察是否被采样，因此需要一次遍历数据，并提供期望的样本大小。 sampleByKeyExact需要比sampleByKey中使用的每层简单随机抽样花费更多的资源，但将提供99.99%置信度的确切抽样大小。 python当前不支持sampleByKeyExact。

`sampleByKeyExact()` 允许用户精确地采样|fk·nk|∀k∈K|fk·nk|∀k∈K项，其中fk fk是关键kk的期望分数，nk nk是关键kk的键值对的数量， 而KK是一组键。 无需更换的采样需要额外通过RDD以保证采样大小，而替换的采样需要两次额外的通过。

```
//  RDD[(K, V)]  任何键值对
val data = sc.parallelize(
  Seq((1, 'a'), (1, 'b'), (2, 'c'), (2, 'd'), (2, 'e'), (3, 'f')))
// 指明每个Key的权重
val fractions = Map(1 -> 0.1, 2 -> 0.6, 3 -> 0.3)
// 从每层获取一个近似的样本
val approxSample = data.sampleByKey(withReplacement = false, fractions = fractions)
// 从每层获取一个确切的样本
val exactSample = data.sampleByKeyExact(withReplacement = false, fractions = fractions)
```

2.4 Hypothesis testing/假设检验

- 假设检验是统计学中强大的工具，用于确定结果是否具有统计学意义，无论该结果是否偶然发生。spark.mllib目前支持Pearson的卡方（χ2χ2）测试，以获得拟合优度和独立性。 输入数据类型确定是否进行拟合优度或独立性测试。 拟合优度测试需要输入类型的Vector，而独立性测试需要一个 `Matrix` 作为输入。

  `spark.mllib还支持输入类型` `RDD[LabeledPoint]` 通过卡方独立测试启用特征选择。Statistics 提供运行Pearson卡方检验的方法。 以下示例演示如何运行和解释假设检验。

```
import org.apache.spark.mllib.linalg._
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.stat.Statistics
import org.apache.spark.mllib.stat.test.ChiSqTestResult
import org.apache.spark.rdd.RDD
// 由事件频率组成的向量
val vec: Vector = Vectors.dense(0.1, 0.15, 0.2, 0.3, 0.25)
// 计算适合度。 如果没有提供第二个测试向量
// 作为参数，针对均匀分布运行测试。
val goodnessOfFitTestResult = Statistics.chiSqTest(vec)
// 测试包括p值，自由度，检验统计量，使用方法,和零假设
println(s"$goodnessOfFitTestResult\n")
// a contingency matrix. Create a dense matrix ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))
val mat: Matrix = Matrices.dense(3, 2, Array(1.0, 3.0, 5.0, 2.0, 4.0, 6.0))
// conduct Pearson's independence test on the input contingency matrix
// 对输入的矩阵进行Pearson独立性测试
val independenceTestResult = Statistics.chiSqTest(mat)
// summary of the test including the p-value, degrees of freedom
// 测试:包括p值，自由度
println(s"$independenceTestResult\n")
val obs: RDD[LabeledPoint] =
  sc.parallelize(
    Seq(
      LabeledPoint(1.0, Vectors.dense(1.0, 0.0, 3.0)),
      LabeledPoint(1.0, Vectors.dense(1.0, 2.0, 0.0)),
      LabeledPoint(-1.0, Vectors.dense(-1.0, 0.0, -0.5)
      )
    )

  ) // (feature, label) pairs.
```

```
// The contingency table is constructed from the raw (feature, label) pairs and used to
conduct
// the independence test. Returns an array containing the ChiSquaredTestResult for every
feature
// against the label.
///应急表由原始的（特征，标签）对构成，用于进行独立性测试。 返回一个包含针对标签的每个功能的
ChiSquaredTestResult的数组。
val featureTestResults: Array[ChiSqTestResult] = Statistics.chiSqTest(obs)
featureTestResults.zipWithIndex.foreach { case (k, v) =>
  println("Column " + (v + 1).toString + ":")
  println(k)
}  // summary of the test
```

2.5 Streaming Significance Testing/流式测试

- `spark.mllib` 提供了一些测试的在线实现，以支持A / B测试用例等。 这些测试可以在Spark Streaming DStream [（Boolean，Double）]上执行，其中每个元组的第一个元素指示控制组（false）或处理组（true），第二个元素是观察值。流式测试显示支持以下参数：
  - `peacePeriod` - 从流中忽略的初始数据点数，用于减轻新奇效应.
  - `windowSize` - - 执行假设测试的过去批次数。 设置为0将使用所有之前的批次执行累积处理。
    `StreamingTest` 提供流式假设检验.

```
val data = ssc.textFileStream(dataDir).map(line => line.split(",") match {
  case Array(label, value) => BinarySample(label.toBoolean, value.toDouble)
})
val streamingTest = new StreamingTest()
  .setPeacePeriod(0)
  .setWindowSize(0)
  .setTestMethod("welch")
val out = streamingTest.registerStream(data)
out.print()
```

2.6Random data generation/随机数据生成

- 随机数据生成对于随机算法，原型设计和性能测试很有用。 spark.mllib支持使用i.i.d.生成随机RDD。 从给定分布绘制的值：均匀，标准正态或泊松分布。

  `RandomRDDs` 提供工厂方法来生成随机double型RDD或向量RDD。 以下示例生成随机double型RDD，其值遵循标准正态分布N（0,1），然后映射到N（1,4）。阅读 `RandomRDDs` Scala docs 文档查看 API 细节

```
import org.apache.spark.SparkContext
import org.apache.spark.mllib.random.RandomRDDs._
val sc: SparkContext = ...
// Generate a random double RDD that contains 1 million i.i.d. values drawn from the
// standard normal distribution `N(0, 1)`, evenly distributed in 10 partitions.
val u = normalRDD(sc, 1000000L, 10)
// Apply a transform to get a random double RDD following `N(1, 4)`.
val v = u.map(x => 1.0 + 2.0 * x)
```

2.7 Kernel density estimation/核密度估计

- 核密度估计 是一种用于可视化经验概率分布的技术，而不需要对所观察到的样本的特定分布进行假设。 它计算在给定集合点评估的随机变量的概率密度函数的估计。 它通过在特定点表达PDF的经验分布来实现这一估计，这是以每个样本为中心的正态分布的PDF平均值.

  `KernelDensity` 提供了从RDD样本计算核密度估计的方法。 以下示例演示如何执行此操作。 阅读 `KernelDensity` 文档查看 API 细节

```
import org.apache.spark.mllib.stat.KernelDensity
import org.apache.spark.rdd.RDD
// an RDD of sample data
val data: RDD[Double] = sc.parallelize(Seq(1, 1, 1, 2, 3, 4, 5, 5, 6, 7, 8, 9, 9))
// Construct the density estimator with the sample data and a standard deviation
// for the Gaussian kernels
val kd = new KernelDensity()
  .setSample(data)
  .setBandwidth(3.0)
// Find density estimates for the given values
val densities = kd.estimate(Array(-1.0, 2.0, 5.0))
```

3.0 分类与回归

spark.mllib包支持 binary classification（二分类），multiclass classification（多分类）和regression analysis（回归分析）各种方法。 下表列出了每种类型问题支持的算法。

问题类型	支持方法
Binary Classification（二分类）	linear SVMs, logistic regression, decision trees, random forests, gradient-boosted trees, naive Bayes
Multiclass Classification（多分类）	logistic regression, decision trees, random forests, naive Bayes
Regression（回归）	linear least squares, Lasso, ridge regression, decision trees, random forests, gradient-boosted trees, isotonic regression

3.1 Linear Methods - RDD-based API（线性方法）

3.1.1 数学方程

- Loss functions（损失函数）
  下面的表格总结了 **spark.mllib** 支持的损失函数和它们的梯度或次梯度。

	loss function L(w;x,y) L(w;x,y)	gradient or sub-gradient
hinge loss	$\max\{0, 1-y\mathbf{w}^T\mathbf{x}\}, \quad y \in \{-1, +1\}$	$\begin{cases} -y \cdot \mathbf{x} & \text{if } y\mathbf{w}^T\mathbf{x} < 1, \\ 0 & \text{otherwise.} \end{cases}$
logistic loss	$\log(1+\exp(-y\mathbf{w}^T\mathbf{x})), \quad y \in \{-1, +1\}$	$-y\left(1-\frac{1}{1+\exp(-y\mathbf{w}^T\mathbf{x})}\right)\cdot\mathbf{x}$
squared loss	$\frac{1}{2}(\mathbf{w}^T\mathbf{x}-y)^2, \quad y \in \mathbb{R}$	$(\mathbf{w}^T\mathbf{x}-y)\cdot\mathbf{x}$

- Regularizers（正则化项）
  正则化的意义在于使模型简化和避免过拟合。在 spark.mllib中支持下面的正则化项：

	regularizer R(w) R(w)	gradient or sub-gradient
zero (unregularized)	0	0
L2	$\frac{1}{2}\|\mathbf{w}\|_2^2$	w
L1	$\|\mathbf{w}\|_1$	$\text{sign}(\mathbf{w})$
elastic net	$\alpha\|\mathbf{w}\|_1 + (1-\alpha)\frac{1}{2}\|\mathbf{w}\|_2^2$	$\alpha\text{sign}(\mathbf{w}) + (1-\alpha)\mathbf{w}$

- Optimization（优化）
  实际上，线性方法用凸优化的方法来优化目标函数。 `**spark.mllib** 用了**SGD and L-BFGS** 两种方法（参考 optimization section）。目前大多算法API支持给**Stochastic Gradient Descent (SGD)**，少部分支持 **Stochastic Gradient Descent (SGD)**。在选择优化方法时，参考 this optimization section 。

3.1.2 Classification（分类）

分类（Classification）的目标是把不同的数据项划分到不同的类别中。其中二元分类（binary classification），有正类和负类两种类别，是最常见的分类类型。如果多于两种类别，就是多元分类（multiclass classification.）。spark.mllib对分类有两种线性方法： linear Support Vector Machines (SVMs) and logistic regression。Linear SVMs 只支持二元分类。而logistic regression对二元和多元分类问题都支持。 对这种方法，spark.mllib都提供有 L1和 L2正则化下的两种变体。在MLlib中，测试数据集用RDD类型的LabeledPoint表示，其中labels从0开始索引：0,1,2.....

- Linear Support Vector Machines (SVMs)支持向量机 linear SVM就是上面方程（1）描述的方法。它是大规模分类任务的标准方法，它的损失函数用hinge loss 给出时的方程：

$$L(\mathbf{w};\mathbf{x},y) := \max\{0, 1-y\mathbf{w}^T\mathbf{x}\}.$$

  默认下，linear SVMs 用L2正则化训练。但我们也提供L1 正则化的选项，当选择L1正则化时，问题变成了线性规划。liner SVMs算法输出一个SVM模型。给出一个用**x**表示的新数据点，模型基于 $\mathbf{w}^T\mathbf{x}$ 做出预测。默认情况，当$\mathbf{w}^T\mathbf{x} \geq 0$，结果是正类，否则，结果是负类。

示例: 下面的代码片段展示了怎么在测试数据上用算法对象的一个静态方法来加载样本数据集，执行训练算法，以及使用训练出来的模型来做预测并计算训练错误率。

```
import org.apache.spark.mllib.classification.{SVMModel, SVMWithSGD}
import org.apache.spark.mllib.evaluation.BinaryClassificationMetrics
import org.apache.spark.mllib.util.MLUtils
// Load training data in LIBSVM format.
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_data.txt")
// Split data into training (60%) and test (40%).
val splits = data.randomSplit(Array(0.6, 0.4), seed = 11L)
val training = splits(0).cache()
val test = splits(1)
// Run training algorithm to build the model
val numIterations = 100
val model = SVMWithSGD.train(training, numIterations)
// Clear the default threshold.
model.clearThreshold()
// Compute raw scores on the test set.
val scoreAndLabels = test.map { point =>
  val score = model.predict(point.features)
  (score, point.label)
}
// Get evaluation metrics.
val metrics = new BinaryClassificationMetrics(scoreAndLabels)
val auROC = metrics.areaUnderROC()
println("Area under ROC = " + auROC)
// Save and load model
model.save(sc, "target/tmp/scalaSVMWithSGDModel")
val sameModel = SVMModel.load(sc, "target/tmp/scalaSVMWithSGDModel")
```

SVMWithSGD.train()方法默认使用正则参数为1.0的L2正则化方法。如果想要配置这个算法，我们可以通过直接创建一个新对象并调用setter方法来定制SVMWithSGD。所有其他的spark.mllib算法同样可以这样定制。比如，下面的代码产生一个SVMs在L1正则化下的变体，它的正则参数设为0.1，并且迭代运行训练算法200次。

```
import org.apache.spark.mllib.optimization.L1Updater
val svmAlg = new SVMWithSGD()
svmAlg.optimizer
  .setNumIterations(200)
  .setRegParam(0.1)
  .setUpdater(new L1Updater)
val modelL1 = svmAlg.run(training)
```

- Logistic regression（逻辑回归）
  - 什么是逻辑回归?

    Logistic回归与多重线性回归实际上有很多相同之处，最大的区别就在于它们的因变量不同，其他的基本都差不多。正是因为如此，这两种回归可以归于同一个家族，即广义线性模型（generalizedlinear model）。

这一家族中的模型形式基本上都差不多，不同的就是因变量不同。

- 如果是连续的，就是多重线性回归；
- 如果是二项分布，就是Logistic回归；
- 如果是Poisson分布，就是Poisson回归；
- 如果是负二项分布，就是负二项回归。

Logistic回归的因变量可以是二分类的，也可以是多分类的，但是二分类的更为常用，也更加容易解释。所以实际中最常用的就是二分类的Logistic回归。

Logistic回归的主要用途：

- 寻找危险因素：寻找某一疾病的危险因素等；
- 预测：根据模型，预测在不同的自变量情况下，发生某病或某种情况的概率有多大；
- 判别：实际上跟预测有些类似，也是根据模型，判断某人属于某病或属于某种情况的概率有多大，也就是看一下这个人有多大的可能性是属于某病。

Logistic回归主要在流行病学中应用较多，比较常用的情形是探索某疾病的危险因素，根据危险因素预测某疾病发生的概率，等等。例如，想探讨胃癌发生的危险因素，可以选择两组人群，一组是胃癌组，一组是非胃癌组，两组人群肯定有不同的体征和生活方式等。这里的因变量就是是否胃癌，即“是”或“否”，自变量就可以包括很多了，例如年龄、性别、饮食习惯、幽门螺杆菌感染等。自变量既可以是连续的，也可以是分类的。

3.1.3 Regression（回归）

- Linear least squares, Lasso, and ridge regression 线性最小二乘法是对回归问题最常见的方程。它可以表示为上面的方程（1），它的损失函数用 **squared loss** 给出时的方程为： $L(\mathbf{w};\mathbf{x},y) := \frac{1}{2}(\mathbf{w}^T\mathbf{x} - y)^2.$

  案例：

```
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.regression.LinearRegressionModel
import org.apache.spark.mllib.regression.LinearRegressionWithSGD
// 加载并解析数据
val data = sc.textFile("data/mllib/ridge-data/lpsa.data")
val parsedData = data.map { line =>
  val parts = line.split(',')
  LabeledPoint(parts(0).toDouble, Vectors.dense(parts(1).split(' ').map(_.toDouble)))
}.cache()
// 建立模型
val numIterations = 100
val stepSize = 0.00000001
val model = LinearRegressionWithSGD.train(parsedData, numIterations, stepSize)
// 评估训练样例的模型并计算训练误差
val valuesAndPreds = parsedData.map { point =>
  val prediction = model.predict(point.features)
  (point.label, prediction)
}
val MSE = valuesAndPreds.map{ case(v, p) => math.pow((v - p), 2) }.mean()
println("training Mean Squared Error = " + MSE)
// 保存并加载模型
model.save(sc, "target/tmp/scalaLinearRegressionWithSGDModel")
val sameModel = LinearRegressionModel.load(sc,
```

```
"target/tmp/scalaLinearRegressionWithSGDModel")
```

- Streaming linear regression （流式线性回归）
  - 当数据流式到达额时候，即时的拟合回归模型，在新数据到达是更新参数是非常有用的。**spark.mllib** 目前用**ordinary least squares**提供流式线性回归。这种方法除了每次只对小批量数据拟合外，其他跟离线情况下相似。这样模型就可以根据数据流持续更新。

    示例

    下面的例子展示了如何加载和测试来自两个文本数据流上的数据，把数据流解析成labeled points，在第一条数据流上拟合出回归模型，在第二条数据流上做预测。

    首先，我们为解析输入数据和创建模型导入必要的类。

    然后为训练和测试数据创建输入流。我们假设 **StreamingContext ssc** 已经被创建，更多信息请看 Spark Streaming Programming Guide 。在这个例子中，我们用 **labeled points** 训练和测试数据流 ，在实践中，测试数据很可能用 **unlabeled vectors**。

    为了得到模型，我们初试化权重为0，注册训练和测试用的数据流，然后启动任务。为了让结果易读，同时打印预测和真标签。

    最后，保存数据文件到训练或者测试目录中。每一行应该是一个数据点，格式为：`(y,[x1,x2,x3])` ，其中 y 是标签，`x1,x2,x3` 是特征值。任何时候，一个文本文件通过 ***args(0)*** 传入，模型将会更新。任何时候，一个文本文件通过args(1) 传入，你将会看到预测结果。随着更多的数据放在训练目录中，预测结果会越来越好。

    这是一个复杂的示例

  - 案例

```
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.regression.StreamingLinearRegressionWithSGD
 val trainingData = ssc.textFileStream(args(0))
                       .map(LabeledPoint.parse)
                       .cache()
val testData = ssc.textFileStream(args(1))
                  .map(LabeledPoint.parse)
val numFeatures = 3
val model = new StreamingLinearRegressionWithSGD()
                .setInitialWeights(Vectors.zeros(numFeatures))
model.trainOn(trainingData)
model.predictOnValues(testData.map(lp => (lp.label, lp.features)))
     .print()
ssc.start()
ssc.awaitTermination()
```

- 实现（开发者）Implementation (developer)
  - 背后，spark.mllib 在底层的原始的梯度下降算法（参考optimization ）的基础上，实现了一个简单的分布式版本的随机梯度下降（stochastic gradient descent ）。提供的所有算法把正则化参数（regParam）作为一个输入。同时还要输入还有与随机梯度下降有关的多种参数 (stepSize , numIterations , miniBatchFraction)。对于每个算法，我都提供三种可能的正则化 (none, L1 or L2)。

对于逻辑回归， L-BFGS 版本在 LogisticRegressionWithLBFGS下实现。L-BFGS 版本同时支持二元和多元回归，而 SGD 版本仅支持二元逻辑回归。但是 L-BFGS 版本不支持 L1 正则化，而 SGD 版本支持L1版本。当L1正则化不是必须的时候，强烈建议使用 L-BFGS 版本，因为它使用牛顿方法逼近逆 Hessian matrix ，比 SGD 版本收敛更快，准确度更高。 所有的算法用scala实现:

- SVMWithSGD
- LogisticRegressionWithLBFGS
- LogisticRegressionWithSGD
- LinearRegressionWithSGD
- RidgeRegressionWithSGD
- LassoWithSGD

3.2 Naive Bayes - RDD-based API（朴素贝叶斯）

- 朴素贝叶斯是一种假设每对features之间具有独立性为前提的简单的多类分类算法。 朴素贝叶斯可以非常有效地进行训练。 在训练数据的单次传递中，计算每个features给定label的条件概率分布，然后应用贝叶斯定理去计算给定label的条件概率分布，并将其用于预测。

  spark.mllib支持多项朴素贝叶斯和伯努利朴素贝叶斯。 这些模型通常用于文档分类。 在这种情况下，每个样本是一个文件，每个feature表示一个词的频率（在多项朴素贝叶斯中），或是0或1去表示该词是否在文档中（在伯努利朴素贝叶斯）。 feature必须是非负数。 使用可选参数“multinomial”或“bernoulli”以“multinomial”作为默认值来选择模型类型。 可以通过设置参数λ（默认为1.01.0）来使用加法平滑。 对于文档分类，输入feature向量通常是稀疏的，稀疏向量应该作为输入提供，以利用稀疏性。 由于训练数据仅使用一次，因此无需缓存。

- 案例：NaiveBayes实现了多项朴素贝叶斯。 它采用LabeledPoint的RDD和可选的平滑参数lambda作为输入，模型类型参数可选（默认为“multinomial”），并输出可用于评估和预测的NaiveBayesModel。有关API的详细信息，请参阅NaiveBayes Scala文档和NaiveBayesModel Scala文档。

```
import org.apache.spark.mllib.classification.{NaiveBayes, NaiveBayesModel}
import org.apache.spark.mllib.util.MLUtils
// 加载并解析数据文件。
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_data.txt")
// 数据分为训练（60%）和测试（40%）。
val Array(training, test) = data.randomSplit(Array(0.6, 0.4))
val model = NaiveBayes.train(training, lambda = 1.0, modelType = "multinomial")
val predictionAndLabel = test.map(p => (model.predict(p.features), p.label))
val accuracy = 1.0 * predictionAndLabel.filter(x => x._1 == x._2).count() / test.count()
// 保存并加载模型
model.save(sc, "target/tmp/myNaiveBayesModel")
val sameModel = NaiveBayesModel.load(sc, "target/tmp/myNaiveBayesModel")
```

3.3 Decision Trees - RDD-based API（决策树）

- 决策树 及其合奏是分类和回归机器学习任务的流行方法。决策树被广泛使用，因为它们易于解释，处理分类特征，扩展到多类分类设置，不需要特征缩放，并且能够捕获非线性和特征交互。诸如随机森林和提升等树组合算法是分类和回归任务的最佳表现者之一。

  spark.mllib支持二进制和多类分类和回归的决策树，使用连续和分类特征。该实现按行分隔数据，允许具有数百万个实例的分布式培训。"Ensemble指南"中描述了树木（随机森林和渐变树）。

- 基本算法 决策树是一种贪心算法，它执行特征空间的递归二进制分区。 树预测每个最低（叶）分区的相同标签。 通过从一组可能的分割中选择最佳分割来贪婪选择每个分区，以便最大化树节点处的信息增益。

- 节点杂质和信息增益

  节点杂质是节点处标签的均匀度的量度。 目前的实施提供了两种分类杂质（基尼杂质和熵）和一种杂质测量回归（方差）。

Impurity	Task	Formula	描述
基尼不纯度	分类	$\sum_{i=1}^{C} f_i(1-f_i)$	f_i 是节点处的标签 i 的频率，C 是唯一标签的数量。
熵	分类	$\sum_{i=1}^{C} -f_i log(f_i)$	f_i 是节点处的标签 i 的频率，C 是唯一标签的数量。
方差	回归	$\frac{1}{N}\sum_{i=1}^{N}(y_i-\mu)^2$	y_i 是一个实例的标签，N 是实例的数量，μ 是 $\frac{1}{N}\sum_{i=1}^{N} y_i$ 给出的平均值

  信息增益是父节点杂质与两个子节点杂质的加权和之间的差异。

- 分裂候选人
  - 连续功能

    对于单机实现中的小数据集，每个连续特征的分离候选通常是该特征的唯一值。 一些实现对特征值进行排序，然后使用有序的唯一值作为快速树计算的分割候选。

    排序特征值对于大型分布式数据集来说是昂贵的。 该实现通过对数据的采样分数执行分位数计算来计算分组候选的近似集合。 有序拆分创建“垃圾箱”，并且可以使用maxBins参数指定此类垃圾桶的最大数量。

    请注意，bin的数量不能大于实例N的数量（由于默认**maxBins**值为32，这是一个罕见的情况）。 如果条件不满足，树算法会自动减少分箱数。

  - 分类功能
- 停止规则

  当满足以下条件之一时，递归树结构停止在节点处：

  1. 节点深度等于**maxDepth**训练参数。
  2. 没有分裂候选人导致信息增益大于***minInfoGain***。
  3. 没有分裂候选产生每个至少具有**minInstancesPerNode**训练实例的子节点。

  使用技巧 我们通过讨论各种参数，包括使用决策树的几个指导原则。这些参数大致按照降序的重要性排列。新用户应主要考虑“问题规范参数”部分和 `maxDepth` 参数。

- 问题规格参数 这些参数描述您要解决的问题和您的数据集。它们应该被指定并且不需要调整。
  - **algo**：决策树的类型，**Classification**或者**Regression**。
  - **numClasses**：课程数量（**Classification**仅限）。
  - **categoricalFeaturesInfo**：指定哪些功能是分类的，以及每个功能可以采用多少种分类值。这是从功能索引到功能特征（类别数量）的映射。本地图中的任何功能都被视为连续的。
    - 例如，`Map(0 -> 2, 4 -> 10)` 指定该特征 `0` 是二进制（取值 `0` 或 `1`），该特征 `4` 具有10个类别（值 `{0, 1, ..., 9}`）。注意，特征索引是基于0的：特征 `0`，并且 `4` 是实例的特征向量的第1和第5个元素。
    - 请注意，您不必指定**categoricalFeaturesInfo**。该算法仍然可以运行，并可能得到合理的结果。但是，如果分类功能被正确指定，性能应该更好。
- 停止标准

  这些参数决定树何时停止构建(添加新节点)。调整这些参数时,请注意在保留的测试数据上进行验证以避免过度拟合。

- **maxDepth**：树的最大深度。更深的树木更具表现力（潜在地允许更高的准确度），但培训成本也更高，更有可能过度使用。
- **minInstancesPerNode**：对于要进一步拆分的节点，每个子节点必须至少接收这一数量的训练实例。这通常与**RandomForest**一起使用，因为它们经常受到比单个树更深的训练。
- **minInfoGain**：对于要进一步拆分的节点，拆分必须至少提高这一点（在信息增益方面）。

- 可调参数

  这些参数可能被调整。为了避免过度调整，请小心在调整时对延迟测试数据进行验证。

  - maxBins：离散化连续特征时使用的分组数。
    - 增加maxBins允许算法考虑更多的分裂候选，并进行细粒度分割决策。然而，它也增加了计算和通信。
    - 请注意，该maxBins参数必须至少为任何分类功能的最大类别数。
  - maxMemoryInMB：用于收集足够统计信息的内存量。
    - 默认值被保守地选择为256 MB，以允许决策算法在大多数情况下工作。maxMemoryInMB通过允许更少的数据传递，增加可以导致更快的训练（如果内存可用）。然而，maxMemoryInMB由于每次迭代的通信量可以成比例，所以可能会有递减的回报maxMemoryInMB。
    - 实现细节：为了更快的处理，决策树算法收集关于要分割的节点组的统计信息（而不是一次一个节点）。可以在一组中处理的节点数由存储器要求确定（每个特征有所不同）。该maxMemoryInMB参数以每个工作人员可以用于这些统计信息的兆字节指定内存限制。
  - subsamplingRate：用于学习决策树的训练数据的分数。该参数对于训练树的组合（使用RandomForest和GradientBoostedTrees）是最相关的，其中可能对原始数据进行子采样是有用的。对于训练单个决策树，此参数不太有用，因为训练实例的数量通常不是主要约束。
  - impurity：杂质测量（上文讨论）用于选择候选分裂。此措施必须与参数匹配algo。

- 缓存和检查点

  MLlib 1.2增加了几个扩展到更大（更深）的树和树组合的功能。何时maxDepth设置为大，启用节点ID缓存和检查点可能很有用。当设置为大时，这些参数对于RandomForest也是有用的numTrees。

  - useNodeIdCache：如果设置为true，则算法将避免在每次迭代时将当前模型（树或树）传递给执行程序。
    - 这对于深树（加速工人计算）和大型随机森林（减少每次迭代中的通信）可能是有用的。
    - 实现细节：默认情况下，算法将当前模型传达给执行器，以便执行程序可以将训练实例与树节点相匹配。当此设置打开时，算法将替代缓存此信息。

  节点ID缓存生成一个RDD序列（每次迭代1个）。这种长期的谱系可能会导致性能问题，但检查中间RDD可以缓解这些问题。请注意，检查点只适用于useNodeIdCache设置为true时。

  - checkpointDir：用于检查点节点ID缓存RDD的目录。
  - checkpointInterval：检查点节点ID缓存RDD的频率。设置太低将导致写入HDFS的额外开销; 如果执行程序出现故障并且需要重新计算RDD，则设置得太高可能会导致问题。

- 缩放 计算在训练实例的数量，特征数量和参数中大致线性地`maxBins`缩放。通信在功能数量上大致线性地缩放**maxBins**。实现的算法读取稀疏密度数据。但是，它并没有针对稀疏输入进行优化。

- 案例： 下面的示例演示如何加载LIBSVM 数据文件，将其解析为RDD **LabeledPoint**，然后使用具有Gini杂质的决策树作为杂质测量和最大树深度为5进行分类。计算测试误差以测量算法准确性。

```
import org.apache.spark.mllib.tree.DecisionTree
import org.apache.spark.mllib.tree.model.DecisionTreeModel
import org.apache.spark.mllib.util.MLUtils
```

```scala
// 加载和解析数据文件
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_data.txt")
// 将数据拆分成训练和测试集（30%用于测试）
val splits = data.randomSplit(Array(0.7, 0.3))
val (trainingData, testData) = (splits(0), splits(1))
// 训练决策树模型
//  空分类功能信息表示所有特征都是连续的。
val numClasses = 2
val categoricalFeaturesInfo = Map[Int, Int]()
val impurity = "gini"
val maxDepth = 5
val maxBins = 32
val model = DecisionTree.trainClassifier(trainingData, numClasses, categoricalFeaturesInfo,
  impurity, maxDepth, maxBins)
// 评估测试实例上的模型并计算测试错误
val labelAndPreds = testData.map { point =>
  val prediction = model.predict(point.features)
  (point.label, prediction)
}
val testErr = labelAndPreds.filter(r => r._1 != r._2).count().toDouble / testData.count()
println("Test Error = " + testErr)
println("学习分类树模型:\n" + model.toDebugString)
// 保存和加载模型
model.save(sc, "target/tmp/myDecisionTreeClassificationModel")
val sameModel = DecisionTreeModel.load(sc, "target/tmp/myDecisionTreeClassificationModel")
```

3.4 Ensembles - RDD-based API（集成方法）

- ensemble method（集成方法）是一种学习算法，是由一组其他基本模型组成的模型。 **spark.mllib** 支持两种主要的集成算法： **GradientBoostedTrees** 和 **RandomForest** 。两种都使用 **decision trees**（决策树）作为基本模型。
- Gradient-Boosted Trees vs. Random Forests（梯度增强树与随机森林）

  Gradient-Boosted Trees (GBTs)（梯度增强树）和 Random Forests（随机森林）都是学习 trees（树）的集成算法的算法，但是训练的过程是不同的。下面有几个实际的 trade-offs ：

  - GBTs 一次训练一棵树，所以他们可以比 random forests（随机森林）训练更长的时间。Random Forests（随机森林）可以并行训练多棵树。
    - 另一方面，使用 GBTs 比较小（较浅）的树与随机森林通常是合理的，训练较小的树需要更少的时间。
  - Random Forests 可能不太容易 overfitting（过拟合）。在 Random Forest（随机森林）中训练更多的树可以减少过拟合的可能性。（在统计语言中，随机森林通过使用更多的树来减少 variance（方差），而 GBTs 通过使用更少的树来减少 bias（偏差））
  - Random Forests 可以更容易调整，因为性能随 trees 数量单调增加（如果 tree 数量过大，则 GBTs 的性能可能会下降）。

  简而言之，这两种算法都是有效的，选择应该基于特定的数据集。
- Random Forests（随机森林）Random forests（随机森林）是 decision trees（决策树）的组合。随机森林是分类和回归最成功的的机器学习模型之一。他们结合了许多决策树，以减少过拟合的风险。像决策树一样，随机森林处理分类特征，扩展到 multiclass classification setting（多类分类设置），不需要 feature scaling (特征缩放)，并且能够捕获 non-linearities and feature interactions（非线性和特征交互）。spark.mllib 支

持二进制和多类分类回归的随机森林，使用分类连续和分类特征。 spark.mllib 使用现有的决策树实现来实现随机森林。有关 tree 的更多信息，请参阅决策树指南。

- Basic algorithm (基本算法) 随机森林分别对一组决策树进行训练，因此可以并行进行训练。该算法将随机性注入到训练过程中，使得每个决策树有一些不同。结合每个树的预测可以减少预测的差异，提高测试数据的性能。
- Training (训练) 注入训练过程的随机性包括：
  - 在每次迭代时对原始数据集进行二次采样 (Subsampling)，以获得不同的训练集 (a.k.a. bootstrapping)
  - 考虑在每个树节点分割的不同的特征的随机子集。
- Prediction (预测) 为了对新实例进行预测，随机森林必须从其决策树集合中预测。对于分类和回归，此聚合方式不同。 Classification (分类)：Majority vote (多数投票)。每棵树的预测都算作一个类的投票。该标签的预测是获得最多的选票的类。 Regression (回归)：Averaging (平均)。每棵树都预测了一个真正的 value 。该标签预测是树预测的平均值。
- Usage tips (使用提示) 我们通过讨论各种参数，包括使用随机森林的几个准则。我们省略了一些决策树参数，因为它们在 决策树指南 中被 covered (覆盖)。 我们提到的前两个参数是最重要的，调整他们通常可以提高性能：
  - numTrees ：随机森林中的树的数目。
    - 增加树的数量将减少预测方差，提高模型的测试时间准确性。
    - 训练时间大致呈线性增加。
  - maxDepth ：随机森林中每个树的最大深度。
    - 增加深度使模型更具表现力和强大。然而，较深的树需要更长的时间训练，也更容易过拟合。
    - 一般来说，在使用随机森林时比使用单个决策树时可以训练更深的树。一棵树比随机森林更容易过拟合 (因为森林中平均多棵树的差异减小)

  接下来的两个参数通常不需要调整。但是他们可以加快训练。

  - subsamplingRate ：此参数指定用于训练随机森林中每棵树的数据集的大小，作为原始数据集大小的一部分。建议使用默认值 (1.0)，但是减小此值可以加快训练。
  - featureSubsetStrategy ：要用作每个树节点分割的候选项的数量。该数字被指定为 fraction or function of the total number of features (功能总数的分数或函数)。减少此数值将加快训练，但是如果设置太低有时会影响性能。
- 案例：Classification (分类)

  下面的示例演示如何加载 **LIBSVM** 数据文件，将其解析为 **LabeledPoint** 的 **RDD** ，然后使用随机森林执行分类。计算出测试误差来测量算法的准确性。

```
import org.apache.spark.mllib.tree.RandomForest
import org.apache.spark.mllib.tree.model.RandomForestModel
import org.apache.spark.mllib.util.MLUtils
// 加载并解析数据文件
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_data.txt")
// 将数据拆分为培训和测试集 ( 30%用于测试 )
val splits = data.randomSplit(Array(0.7, 0.3))
val (trainingData, testData) = (splits(0), splits(1))
// 训练RandomForest模型。
// 空的categoricalFeaturesInfo表示所有功能都是连续的。
val numClasses = 2
val categoricalFeaturesInfo = Map[Int, Int]()
```

```
val numTrees = 3 // Use more in practice.
val featureSubsetStrategy = "auto" // Let the algorithm choose.
val impurity = "gini"
val maxDepth = 4
val maxBins = 32
val model = RandomForest.trainClassifier(trainingData, numClasses, categoricalFeaturesInfo,
  numTrees, featureSubsetStrategy, impurity, maxDepth, maxBins)
// 评估测试实例上的模型并计算测试错误
val labelAndPreds = testData.map { point =>
  val prediction = model.predict(point.features)
  (point.label, prediction)
}
val testErr = labelAndPreds.filter(r => r._1 != r._2).count.toDouble / testData.count()
println("Test Error = " + testErr)
println("Learned classification forest model:\n" + model.toDebugString)
// 保存并加载模型
model.save(sc, "target/tmp/myRandomForestClassificationModel")
val sameModel = RandomForestModel.load(sc, "target/tmp/myRandomForestClassificationModel")
```

3.5 Regression - RDD-based API（回归）

- Isotonic regression（保序回归）spark.mllib支持一个相邻违反算法的池，它使用一种方法并行化保序回归。训练输入是三个双重值的元组的RDD，表示该顺序的标签，特征和权重。另外IsotonicRegression算法还有一个参数名为isotonic，其默认值为真，它指定保序回归为保序（单调递增）或者反序（单调递减）。

  训练返回一个保序回归模型，可以被用于来预测已知或者未知特征值的标签。保序回归的结果是分段线性函数，预测规则如下：

  - 如果预测输入与训练中的特征值完全匹配，则返回相应标签。如果一个特征值对应多个预测标签值，则返回其中一个，具体是哪一个未指定。
  - 如果预测输入比训练中的特征值都高（或者都低），则相应返回最高特征值或者最低特征值对应标签。如果一个特征值对应多个预测标签值，则相应返回最高值或者最低值。
  - 如果预测输入落入两个特征值之间，则预测将会是一个分段线性函数，其值由两个最近的特征值的预测值计算得到。如果一个特征值对应多个预测标签值，则使用上述两种情况中的处理方式解决。

- 案例

```
import org.apache.spark.mllib.regression.{IsotonicRegression, IsotonicRegressionModel}
import org.apache.spark.mllib.util.MLUtils

val data = MLUtils.loadLibSVMFile(sc,
  "data/mllib/sample_isotonic_regression_libsvm_data.txt").cache()

// 从权重设置为默认值1.0的输入数据创建标签，要素和权重元组。
val parsedData = data.map { labeledPoint =>
  (labeledPoint.label, labeledPoint.features(0), 1.0)
}
// 将数据拆分为训练集（60%）和测试（40%）集。
val splits = parsedData.randomSplit(Array(0.6, 0.4), seed = 11L)
val training = splits(0)
val test = splits(1)
```

```
//根据训练数据创建保序回归模型。
// Isotonic参数默认为true，因此仅显示演示
val model = new IsotonicRegression().setIsotonic(true).run(training)

// 创建预测和真实标签的元组。
val predictionAndLabel = test.map { point =>
  val predictedLabel = model.predict(point._2)
  (predictedLabel, point._1)
}

// 计算预测标签和真实标签之间的均方误差。
val meanSquaredError = predictionAndLabel.map { case (p, l) => math.pow((p - l), 2)
}.mean()
println("Mean Squared Error = " + meanSquaredError)

// 保存并加载模型
model.save(sc, "target/tmp/myIsotonicRegressionModel")
val sameModel = IsotonicRegressionModel.load(sc, "target/tmp/myIsotonicRegressionModel")
```

4.0 Collaborative filtering（协同过滤）

- 协作过滤通常用于推荐系统。这些技术旨在填补用户-项目关联矩阵中丢失的条目。**spark.mllib** 目前支持基于模型的协同过滤，其中用户和项目通过一小组潜在因素来描述，可用于预测缺失的条目。**spark.mllib** 使用 **ALS（交替最小二乘法）**算法来学习这些潜在因素。**spark.mllib** 中的实现具有以下参数：
  - ***numBlocks*** 是用于并行化计算的块数（设置为-1到自动配置）。
  - ***rank*** 是模型中潜在因素的数量。
  - ***iterations*** 是运行 **ALS** 的迭代次数。 **ALS** 通常在20次或更少次迭代中收敛到合理的解决方案。
  - ***lambda*** 指定 **ALS** 中的正则化参数。
  - ***implicitPrefs*** 指定是使用显式反馈ALS的版本还是用适用于隐式反馈数据集的版本。
  - ***alpha*** 是适用于控制偏好观察的基线置信度的ALS的隐式反馈版本的参数。

- ## Explicit vs. implicit feedback（显式与隐式反馈）

  基于矩阵分解的协同过滤的标准方法是将用户-项目关联矩阵中的元素视为用户对项目的显式偏好，例如用户对电影的评分。在现实世界中的许多用例中，通常只能接触到隐式的反馈（例如浏览，点击，购买，喜欢，分享等）。在 **spark.mllib** 中使用基于隐式反馈数据集的协同过滤的方法来处理这些数据。实际上，这种方法不是直接对数据矩阵进行建模，而是将数据视为代表用户行为意愿强度的数字（例如点击的次数或某人累积观看电影的时间）。然后，这些数字与观察到的用户偏好的置信水平相关，而不是给予项目的明确评级。然后，该模型尝试找到可用于预测用户对项目的预期偏好的潜在因素。

- ## Scaling of the regularization parameter（正则化参数的换算）

  自 V1.1 起，我们通过用户在更新用户因素中产生的评分，或产品在更新产品因素中收到的评分来求解每个最小二乘问题的规则化参数 ***lambda*** 。 这种方法被命名为“ **ALS-WR** ”（加权正则化交替最小二乘法），并在论文“ **Large-Scale Parallel Collaborative Filtering for the Netflix Prize** ”中进行了讨论。 它使 ***lambda*** 对数据集的规模依赖较少，因此我们可以将从采样子集学到的最佳参数应用于完整数据集，并期望能有相似的表现。

- 案例

```
import org.apache.spark.mllib.recommendation.ALS
```

```
import org.apache.spark.mllib.recommendation.MatrixFactorizationModel
import org.apache.spark.mllib.recommendation.Rating

// 加载和解析数据
val data = sc.textFile("data/mllib/als/test.data")
val ratings = data.map(_.split(',') match { case Array(user, item, rate) =>
  Rating(user.toInt, item.toInt, rate.toDouble)
})

// 使用ALS构建推荐模型
val rank = 10
val numIterations = 10
val model = ALS.train(ratings, rank, numIterations, 0.01)

// 评价模型
val usersProducts = ratings.map { case Rating(user, product, rate) =>
  (user, product)
}
val predictions =
  model.predict(usersProducts).map { case Rating(user, product, rate) =>
    ((user, product), rate)
  }
val ratesAndPreds = ratings.map { case Rating(user, product, rate) =>
  ((user, product), rate)
}.join(predictions)
val MSE = ratesAndPreds.map { case ((user, product), (r1, r2)) =>
  val err = (r1 - r2)
  err * err
}.mean()
println("Mean Squared Error = " + MSE)

//保存并加载模型
model.save(sc, "target/tmp/myCollaborativeFilter")
val sameModel = MatrixFactorizationModel.load(sc, "target/tmp/myCollaborativeFilter")
```

5.0 Clustering - RDD-based API（聚类 - 基于RDD的API）

聚类是一个无监督的学习问题，我们的目标是基于一些相似性的概念将实体的子集相互组合。聚类经常用于探索性分析和/或作为分层 监督学习流水线的组成部分（其中针对每个群集训练不同的分类器或回归模型）。 该 spark.mllib 软件包支持以下模型：

- K-means
- 高斯混合
- 幂迭代聚类（PIC）
- 潜在Dirichlet分配（LDA）
- 平分k-means
- Streaming k-means

5.1 K-means

- K-means是将数据点聚类到预定数量的聚类中最常用的聚类算法之一。该**spark.mllib**实现包括一个称为kmeans ||的k-means ++方法的并行变体 。在`**spark.mllib**中，`这个实现具有以下参数：
  - *k*是所需簇的数量。请注意，可以返回少于k个集群，例如，如果有少于k个不同的集群点。
  - *maxIterations*是要运行的最大迭代次数。
  - *initializationMode*通过k-means ||指定随机初始化或初始化。
  - *run*这个参数从Spark 2.0.0起没有效果。
  - *initializationSteps*确定k-means ||中的步数 算法。
  - *epsilon*决定了我们认为k-means收敛的距离阈值。
  - *initialModel*是用于*初始化*的一组可选集群。如果提供此参数，则只执行一次运行。
- 案例 可以使用*spark-shell*执行以下代码片段。 在加载和解析数据后的以下示例中，我们使用该`KMeans`对象将数据集群到两个集群中。所需的簇的数量被传递给算法。然后我们计算平方误差的集合和（WSSSE）。你可以通过增加*k*来减少此误差度量。实际上，最优的*k*通常是WSSSE图中有一个“elbow”。

```
import org.apache.spark.mllib.clustering.{KMeans, KMeansModel}
import org.apache.spark.mllib.linalg.Vectors
//加载和解析数据
val data = sc.textFile("data/mllib/kmeans_data.txt")
val parsedData = data.map(s => Vectors.dense(s.split(' ').map(_.toDouble))).cache()
// 使用KMeans将数据集成到2个类中
val numClusters = 2
val numIterations = 20
val clusters = KMeans.train(parsedData, numClusters, numIterations)
// 通过计算在平方误差的总和中评估聚类
val WSSSE = clusters.computeCost(parsedData)
println("Within Set Sum of Squared Errors = " + WSSSE)
// 保存和加载模型
clusters.save(sc, "target/org/apache/spark/KMeansExample/KMeansModel")
val sameModel = KMeansModel.load(sc, "target/org/apache/spark/KMeansExample/KMeansModel")
```

5.2 高斯混合

- 高斯混合模型代表一个复合分布，由此点是从一个绘制*k*高斯子分布，每个具有其自己的概率。该**spark.mllib**实现使用 期望最大化算法来给出给定一组样本的最大似然模型。这个实现具有以下参数：
  - *k*是所需簇的数量。
  - *convergenceTol*是我们考虑收敛的对数*似然*的最大变化。
  - *maxIterations*是不达到收敛的最大迭代次数。
  - *initialModel*是启动EM算法的可选起始点。如果省略此参数，将从数据中构建一个随机起始点。
- 案例 在加载和解析数据之后的以下示例中，我们使用 GaussianMixture对象将数据集群到两个集群中。所需的簇的数量被传递给算法。然后我们输出混合模型的参数。

```
import org.apache.spark.mllib.clustering.{GaussianMixture, GaussianMixtureModel}
import org.apache.spark.mllib.linalg.Vectors
// 加载和解析数据
val data = sc.textFile("data/mllib/gmm_data.txt")
val parsedData = data.map(s => Vectors.dense(s.trim.split(' ').map(_.toDouble))).cache()

// 使用GaussianMixture将数据分成两类
```

```
val gmm = new GaussianMixture().setK(2).run(parsedData)
// 保存和加载模型
gmm.save(sc, "target/org/apache/spark/GaussianMixtureExample/GaussianMixtureModel")
val sameModel = GaussianMixtureModel.load(sc,
  "target/org/apache/spark/GaussianMixtureExample/GaussianMixtureModel")
// 输出最大似然模型的参数
for (i <- 0 until gmm.k) {
  println("weight=%f\nmu=%s\nsigma=\n%s\n" format
    (gmm.weights(i), gmm.gaussians(i).mu, gmm.gaussians(i).sigma))
}
```

5.3 幂迭代聚类（PIC）

- 幂迭代聚类（PIC）是一种可扩展和高效的算法，用于将图中的顶点聚类为给定的边缘属性的成对相似性，Lin和Cohen，Power Iteration Clustering中描述。它通过功率迭代计算图的归一化亲和度矩阵的伪特征向量，并将 其用于聚类顶点。*spark.mllib*包括使用GraphX作为其后端的PIC的实现。它需要一个`RDD`的`(srcId, dstId, similarity)`元组，并输出与该聚类分配的模型。相似之处必须是非负的。PIC假定相似性度量是对称的。`(srcId, dstId)`输入数据中最多只能出现一对，而不管排序如何。如果输入中缺少一对，则将其相似性视为零。`spark.mllib`
  - `k`：簇数
  - `maxIterations`：最大功率迭代次数
  - `initializationMode`：初始化模型 这可以是使用随机向量作为顶点属性的“随机”，它是默认的，或者“度”使用归一化的相似度。
- 案例 下面我们来看一下代码片段演示如何在**spark.mllib**使用PIC 。`PowerIterationClustering`实现PIC算法。它需要代表亲和度矩阵`RDD`的`(srcId: Long, dstId: Long, similarity: Double)`元组。调用`PowerIterationClustering.run`返回a `PowerIterationClusteringModel`，其中包含计算的聚类分配。

```
import org.apache.spark.mllib.clustering.PowerIterationClustering
val circlesRdd = generateCirclesRdd(sc, params.k, params.numPoints)
val model = new PowerIterationClustering()
  .setK(params.k)
  .setMaxIterations(params.maxIterations)
  .setInitializationMode("degree")
  .run(circlesRdd)
val clusters = model.assignments.collect().groupBy(_.cluster).mapValues(_.map(_.id))
val assignments = clusters.toList.sortBy { case (k, v) => v.length }
val assignmentsStr = assignments
  .map { case (k, v) =>
    s"$k -> ${v.sorted.mkString("[", ",", "]")}"
  }.mkString(", ")
val sizesStr = assignments.map {
  _._2.length
}.sorted.mkString("(", ",", ")")
println(s"Cluster assignments: $assignmentsStr\ncluster sizes: $sizesStr")
```

5.4 潜在Dirichlet分配（LDA）

- 潜在的Dirichlet分配（LDA） 是一个主题模型，从一组文本文档推断主题。LDA可以被认为是聚类算法如下：

- 主题对应于集群中心，文档对应于数据集中的示例（行）。
- 主题和文档都存在于特征空间中，其中特征向量是字计数（单词包）的向量。
- LDA不是使用传统距离来估计聚类，而是使用基于如何生成文本文档的统计模型的函数。

LDA通过`setOptimizer`功能支持不同的推理算法。`EMLDAOptimizer`使用 期望最大化 对似然函数进行聚类，并产生综合结果，

同时`OnlineLDAOptimizer`使用迭代小批量抽样进行在线变分推理 ，通常内存友好。

LDA将文档集合作为字计数的向量和以下参数（使用构建器模式设置）：

- `k`：主题数量（即集群中心）
- `optimizer`：用于学习LDA模型的优化器，或者`EMLDAOptimizer`或`OnlineLDAOptimizer`
- `docConcentration`：Dirichlet参数，用于事先通过主题分发的文档。较大的值会鼓励更平稳的推断分布。
- `topicConcentration`：Dirichlet参数，用于事先通过术语（词）分配主题。较大的值会鼓励更平稳的推断分布。
- `maxIterations`：限制迭代次数。
- `checkpointInterval`：如果使用检查点（在Spark配置中设置），则此参数指定将创建检查点的频率。如果**maxIterations**大，使用检查点可以帮助减少磁盘上的随机文件大小，并帮助恢复故障。

所有的**spark.mllib的**LDA型号都支持：

- `describeTopics`：将主题作为最重要术语和术语权重的数组返回
- `topicsMatrix`：返回一个`vocabSize`由`k`矩阵，其中各列是一个主题

注意：LDA仍然是积极开发的实验功能。因此，某些功能仅在优化程序生成的两个优化器/模型之一中可用。目前，分布式模型可以转换为本地模型，但反之亦然。

以下讨论将分别描述每个优化器/模型对。

- **期望最大化**

  在`EMLDAOptimizer`和`DistributedLDAModel中的实现`。

  为**LDA**提供的参数：

  - `docConcentration`：只支持对称先验，所以提供的`k`维度向量中的所有值必须相同。所有值也必须为1.0。
    提供默认行为的结果（均值维矢量，值为$（50/k）+1$>1.0 >1.0`Vector(-1)``k`(50/k)+1 (50/k)+1
  - `topicConcentration`：只支持对称先验。值必须为1.0。提供结果，默认值为$0.1+1$。>1.0 >1.0`-1`0.1+1 0.1+1
  - `maxIterations`：EM迭代的最大数量。

  注意：重复执行足够的迭代。在早期迭代中，**EM**通常有无用的主题，但是这些主题在更多的迭代之后会显着改善。使用至少20次和可能的50-100次迭代通常是合理的，具体取决于您的数据集。

  `EMLDAOptimizer`产生一个`DistributedLDAModel`，它不仅存储推断的主题，而且存储训练语料库中每个文档的完整的训练语料库和主题分布。一个`DistributedLDAModel的`支持：

  - `topTopicsPerDocument`：训练语料库中每个文档的主题及其权重
  - `topDocumentsPerTopic`：每个主题的顶部文档以及文档中主题的相应权重。
  - `logPrior`：考虑到超参数的估计主题和文档的主题分布数概率`docConcentration`和`topicConcentration`
  - `logLikelihood`：训练语料库的对数可能性，给出推断的主题和文档主题分布

  在线变异贝叶斯

  在`OnlineLDAOptimizer`和`LocalLDAModel中的实现`。

为**LDA**提供的参数：

- `docConcentration``k` >=0 >=0 `Vector(-1)``k` (1.0/k) (1.0/k)
- `topicConcentration` >=0 >=0 `-1` (1.0/k) (1.0/k)
- `maxIterations` ：要提交的最大批量数量。

另外，`OnlineLDAOptimizer` 接受以下参数：

- `miniBatchFraction` ：每次迭代采样和使用语料库的分数
- `optimizeDocConcentration` ：如果设置为true，则在每个minibatch之后执行超参数 `docConcentration` (aka `alpha`) 的最大似然估计 `docConcentration` ，并将返回的优化 `LocalLDAModel`
- `tau0``kappa` (τ 0 +iter) −κ (τ0+iter)−κiter iter

`OnlineLDAOptimizer` 产生一个 `LocalLDAModel` ，它只存储推断的主题。一个 `LocalLDAModel` 支持：

- `logLikelihood(documents)` ：计算所提供 `documents` 给定推断主题的下限 。
- `logPerplexity(documents)` ：计算提供的 `documents` 给定推断主题的困惑度的上限。

案例
在以下示例中，我们加载表示文档语料库的字数向量。然后，我们使用**LDA** 从文档中推断出三个主题。所需的簇的数量被传递给算法。然后我们输出主题，表示为词的概率分布。

```
import org.apache.spark.mllib.clustering.{DistributedLDAModel, LDA}
import org.apache.spark.mllib.linalg.Vectors

// 加载和解析数据
val data = sc.textFile("data/mllib/sample_lda_data.txt")
val parsedData = data.map(s => Vectors.dense(s.trim.split(' ').map(_.toDouble)))
// 具有唯一ID的索引文档
val corpus = parsedData.zipWithIndex.map(_.swap).cache()

// 使用LDA将文档集成到三个主题中
val ldaModel = new LDA().setK(3).run(corpus)

// 输出主题。每个都是一个分配的单词（匹配的单词向量）
println("Learned topics (as distributions over vocab of " + ldaModel.vocabSize + "
words):")
val topics = ldaModel.topicsMatrix
for (topic <- Range(0, 3)) {
  print("Topic " + topic + ":")
  for (word <- Range(0, ldaModel.vocabSize)) { print(" " + topics(word, topic)); }
  println()
}

// 保存和加载模型
ldaModel.save(sc, "target/org/apache/spark/LatentDirichletAllocationExample/LDAModel")
val sameModel = DistributedLDAModel.load(sc,
  "target/org/apache/spark/LatentDirichletAllocationExample/LDAModel")
```

5.5 平分k-means

- 平均K均值通常比常规K均值快得多，但通常会产生不同的聚类。

平分k均值是一种层次聚类。分层聚类是集群分析中最常用的方法之一，它旨在构建集群的层次结构。层次聚类的策略通常分为两种：

- 集合：这是一种“自下而上”的方法：每个观察在自己的集群中开始，并且一组聚类在层次结构上移动时被合并。
- 分裂：这是一种“自上而下”的方法：所有观察都从一个群集中开始，并且分层在层级结构向下移动时递归执行。

二分法k均值算法是一种分裂算法。**MLlib**中的实现具有以下参数：

- *k*：所需的叶簇数（默认值：4）。如果没有可分割的叶簇，实际数量可能更小。
- *maxIterations*：拆分集群的k-means迭代的最大数量（默认值：20）
- *minDivisibleClusterSize*：可*分组*的最小点数（如果> = 1.0）或点的最小比例（如果<1.0）（默认值：1）
- *种子*：一个随机种子（默认值：类名的哈希值）

- 案例

```
import org.apache.spark.mllib.clustering.BisectingKMeans
import org.apache.spark.mllib.linalg.{Vector, Vectors}
// 加载和解析数据
def parse(line: String): Vector = Vectors.dense(line.split(" ").map(_.toDouble))
val data = sc.textFile("data/mllib/kmeans_data.txt").map(parse).cache()
// 通过BisectingKMeans将数据聚类成6个簇。
val bkm = new BisectingKMeans().setK(6)
val model = bkm.run(data)
// 显示计算成本和集群中心
println(s"Compute Cost: ${model.computeCost(data)}")
model.clusterCenters.zipWithIndex.foreach { case (center, idx) =>
  println(s"Cluster Center ${idx}: ${center}")
}
```

5.5 Streaming k-means

- 当数据到达流时，我们可能需要动态地估计集群，并在新数据到达时进行更新。`spark.mllib`提供对流式k均值聚类的支持，其中包含用于控制估计的衰减（或“健忘”）的参数。该算法使用小批量k-means更新规则的泛化。
- 案例

```
import org.apache.spark.mllib.clustering.StreamingKMeans
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.streaming.{Seconds, StreamingContext}
val conf = new SparkConf().setAppName("StreamingKMeansExample")
val ssc = new StreamingContext(conf, Seconds(args(2).toLong))
val trainingData = ssc.textFileStream(args(0)).map(Vectors.parse)
val testData = ssc.textFileStream(args(1)).map(LabeledPoint.parse)
val model = new StreamingKMeans()
  .setK(args(3).toInt)
  .setDecayFactor(1.0)
  .setRandomCenters(args(4).toInt, 0.0)
```

```
model.trainOn(trainingData)
model.predictOnValues(testData.map(lp => (lp.label, lp.features))).print()
ssc.start()
ssc.awaitTermination()
```

6.0Dimensionality Reduction - RDD-based API（降维）

降维是减少所考虑的变量数量的过程.它可以用于从原始和噪声特征中提取潜在特征,或者在保持结构的同时压缩数据.**spark.mlib** 提供对 **RowMatrix class** 降维的支持.

6.1 Singular value decomposition (SVD)（奇异值分解）

- Singular value decomposition (SVD)（奇异值分解）将矩阵分解为三个矩阵: U, Σ, 和 V,例如

$$A = U\Sigma V^T,$$

- 解释 :
  - **U**是一个正交矩阵（方阵）,其列被称为左奇异向量.
  - **Σ**是一个对角线元素的值为非负数的对角矩阵,其对角线元素被称为奇异值.
  - **V**是一个正交矩阵,其列称为右奇异向量.

  对于大型的矩阵,通常我们不需要完整的因式分解,而只需要顶点奇异值及相关的奇异向量.这可以节省存储,去噪和恢复矩阵的低阶结构.

  如果我们保留k个奇异值（将n维空间映射到k维空间）,则所得到的低阶矩阵的维数将是 :
  - **U: m×k**
  - **Σ: k×k**
  - **V: n×k**
- SVD Example（SVD例子)

```
import org.apache.spark.mllib.linalg.Matrix
import org.apache.spark.mllib.linalg.SingularValueDecomposition
import org.apache.spark.mllib.linalg.Vector
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.linalg.distributed.RowMatrix
val data = Array(
  Vectors.sparse(5, Seq((1, 1.0), (3, 7.0))),
  Vectors.dense(2.0, 0.0, 3.0, 4.0, 5.0),
  Vectors.dense(4.0, 0.0, 0.0, 6.0, 7.0))
val dataRDD = sc.parallelize(data, 2)
val mat: RowMatrix = new RowMatrix(dataRDD)
// 计算最明显的5个奇异值和相应的奇异变量
val svd: SingularValueDecomposition[RowMatrix, Matrix] = mat.computeSVD(5, computeU = true)
val U: RowMatrix = svd.U  // U是一个RowMatrix类.
val s: Vector = svd.s  // 奇异值存储在局部密集矢量中.
val V: Matrix = svd.V  // V是一个局部密集矩阵.
```

6.2 Principal component analysis (PCA)

- [Principal component analysis](#) (PCA) 是一种统计方法来找到一个旋转，使得第一个坐标具有最大的方差可能，每个后续坐标依次具有最大的方差。旋转矩阵的列称为主要组件。 PCA广泛用于降维。

  spark.mllib 支持 **PCA**，用于以行为导向的格式和任何向量存储的高和低密度矩阵。

- 案例：以下代码演示如何计算 **RowMatrix** 上的主要组件，并使用它们将向量投影到低维空间中

```
import org.apache.spark.mllib.linalg.Matrix
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.linalg.distributed.RowMatrix
val data = Array(
  Vectors.sparse(5, Seq((1, 1.0), (3, 7.0))),
  Vectors.dense(2.0, 0.0, 3.0, 4.0, 5.0),
  Vectors.dense(4.0, 0.0, 0.0, 6.0, 7.0))
val dataRDD = sc.parallelize(data, 2)
val mat: RowMatrix = new RowMatrix(dataRDD)
// 计算前4个主要组件
// 主成分存储在局部密集矩阵中
val pc: Matrix = mat.computePrincipalComponents(4)
// 将行投影到由前4个主要成分的线性空间
val projected: RowMatrix = mat.multiply(pc)
```

- 案例：以下代码演示如何计算源向量中的主成分，并使用它们将向量投影到低维空间中，同时保留相关标签

```
import org.apache.spark.mllib.feature.PCA
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.rdd.RDD
val data: RDD[LabeledPoint] = sc.parallelize(Seq(
  new LabeledPoint(0, Vectors.dense(1, 0, 0, 0, 1)),
  new LabeledPoint(1, Vectors.dense(1, 1, 0, 1, 0)),
  new LabeledPoint(1, Vectors.dense(1, 1, 0, 0, 0)),
  new LabeledPoint(0, Vectors.dense(1, 0, 0, 0, 0)),
  new LabeledPoint(1, Vectors.dense(1, 1, 0, 0, 0))))
// 计算前5个主要成分
val pca = new PCA(5).fit(data.map(_.features))

// Project vectors to the linear space spanned by the top 5 principal
// components, keeping the label
val projected = data.map(p => p.copy(features = pca.transform(p.features)))
```

7.0 Feature Extraction and Transformation - RDD-based API（特征的提取和转换）

7.1 TF-IDF

- TF-IDF

  注意 我们建议使用基于 DataFrame 的 API，这在 ML 的关于 TF-IDF 的用户指南中有详细介绍。

词频-逆文档频率（TF-IDF）是一种在文本挖掘中广泛使用的特征向量化方法，以反映字词在语料库中的重要性。用 t 表示词频，d 表示文档，和 D 表示语料库。词频 TF(t,d) 是指字词 t 在文档 d 中出现的次数，而文档频率 `DF(t,D)` 是指语料库 D 含有包含字词 `t 的文档数`。如果我们只使用词频来衡量其重要性，就会很容易过分强调重出现次数多但携带信息少的单词，例如 "a"，“the” 和 "of"。 如果某个字词在语料库中高频出现，这意味着它不包含关于特定文档的特殊信息。 逆文档频率是单词携带信息量的数值度量：

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

其中 |D|是语料中的文档总数。由于使用了log计算，如果单词在所有文档中出现，那么IDF就等于0。注意这里做了平滑处理（+1操作），防止字词没有在语料中出现时IDF计算中除0。 TF-IDF度量是TF和IDF的简单相乘：

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

事实上词频和文档频率的定义有多重变体。在 spark.mllib 中，为了灵活性我们将TF和IDF分开处理。

MLlib中词频统计的实现使用了hashing trick(散列技巧)，也就是使用哈希函数将原始特征映射到一个索引（字词）。然后基于这个索引来计算词频。这个方法避免了全局的单词到索引的映射，全局映射对于大量语料有非常昂贵的计算/存储开销；但是该方法也带来了潜在哈希冲突的问题，不同原始特征可能会被映射到相同的索引。为了减少冲突率，我们可以提升目标特征的维度，换句话说，哈希表中桶的数量。默认特征维度是220 = 1048576。

注意：spark.mllib 没有提供文本分段（例如分词）的工具。用户可以参考 Stanford NLP Group 和 scalanlp/chalk。

- 案例 TF和IDF分别在类HashingTF and IDF中实现。HashingTF 以 RDD[Iterable[_]] 为输入。每条记录是可遍历的字符串或者其他类型。

```
import org.apache.spark.mllib.feature.{HashingTF, IDF}
import org.apache.spark.mllib.linalg.Vector
import org.apache.spark.rdd.RDD
// 加载文件（每行一个）。
val documents: RDD[Seq[String]] = sc.textFile("data/mllib/kmeans_data.txt")
  .map(_.split(" ").toSeq)
val hashingTF = new HashingTF()
val tf: RDD[Vector] = hashingTF.transform(documents)
// 在应用 HashingTF 时，只需要单次传递数据，应用 IDF 需要两次：
// 首先计算 IDF 向量，其次用 IDF 来缩放字词频率。
tf.cache()
val idf = new IDF().fit(tf)
val tfidf: RDD[Vector] = idf.transform(tf)
// spark.mllib IDF 实现提供了忽略少于最少文档数量的字词的选项。
//在这种情况下，这些条款的 IDF 设置为0。
// 通过将minDocFreq值传递给IDF构造函数，可以使用此功能。
val idfIgnore = new IDF(minDocFreq = 2).fit(tf)
val tfidfIgnore: RDD[Vector] = idfIgnore.transform(tf)
```

7.2 Word2Vec

- Word2Vec 计算单词的向量表示。这种表示的主要优点是相似的词在向量空间中离得近，这使得向新模式的泛化更容易并且模型估计更健壮。分布式的向量表示在诸如命名实体识别、歧义消除、句子解析、打标签以及机器翻译等自然语言处理程序中比较有用。

- 模型 在我们实现 Word2Vec 时，使用的是 skip-gram 模型。skip-gram 的目标函数是学习擅长预测同一个句子中词的上下文的词向量表示。用数学语言表达就是，给定一个训练单词序列：w1, w2, ..., wT, skip-gram 模型的目标是最大化平均 log 似然函数(log-likelihood):

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{j=-k}^{j=k}\log p(w_{t+j}|w_t)$$

其中 k 是训练窗口的大小，也就是给定一个词，需要分别查看前后 k 个词。

在 skip-gram 模型中，每个词 w 跟两个向量 uw 和 vw 关联：uw 是 w 的词向量表示，是 vw 上下文。给定单词 wj，正确预测单词 wi 的概率取决于 softmax 模型：

$$p(w_i|w_j)=\frac{\exp(u_{w_i}^{\top}v_{w_j})}{\sum_{l=1}^{V}\exp(u_l^{\top}v_{w_j})}$$

其中 V 是词汇量大小。

使用 softmax 的 skip-gram 模型开销很大，因为 log p(wi|wj) 的计算量跟 V 成比例，而 V 很可能在百万量级。为了加速 Word2Vec 的训练，我们引入了层次 softmax，该方法将计算 log p(wi|wj) 时间复杂度降低到了 O(log(V))。

- 案例：

在下面的例子中，首先导入文本文件，然后将数据解析为 RDD[Seq[String]]，接着构造 Word2Vec 实例并使用输入数据拟合出 `Word2VecModel` 模型。最后，显示了指定单词的 40 个同义词。要运行这段程序，需要先下载 text8 数据并解压到本地目录。这里假设我们抽取的文件是 text8，并在相同目录下运行 spark shell。

```
import org.apache.spark.mllib.feature.{Word2Vec, Word2VecModel}
val input = sc.textFile("data/mllib/sample_lda_data.txt").map(line => line.split("
").toSeq)
val word2vec = new Word2Vec()
val model = word2vec.fit(input)
val synonyms = model.findSynonyms("1", 5)
for((synonym, cosineSimilarity) <- synonyms) {
  println(s"$synonym $cosineSimilarity")
}
// 保存和加载模型
model.save(sc, "myModelPath")
val sameModel = Word2VecModel.load(sc, "myModelPath")
```

7.3 StandardScaler（标准化）

- 标准化是指：对于训练集中的样本，基于列统计信息将数据除以方差或（且）者将数据减去其均值（结果是方差等于1，数据在 0 附近）。这是很常用的预处理步骤。

例如，当所有的特征具有值为 1 的方差且/或值为 0 的均值时，SVM 的径向基函数（RBF）核或者 L1 和 L2 正则化线性模型通常有更好的效果。

标准化可以提升模型优化阶段的收敛速度，还可以避免方差很大的特征对模型训练产生过大的影响。

- **Model Fitting（模型拟合）**

`类 StandardScaler`的构造函数具有下列参数：

- `withMean` 默认值False. 在尺度变换（除方差）之前使用均值做居中处理（减去均值）。这会导致密集型输出，所以在稀疏数据上无效。
- `withStd` 默认值True. 将数据缩放（尺度变换）到单位标准差。

StandardScaler.fit()方法以RDD[Vector]为输入，计算汇总统计信息，然后返回一个模型，该模型可以根据`StandardScaler`配置将输入数据转换为标准差为1，均值为0的特征。

模型中还实现了`VectorTransformer`，这个类可以对 Vector 和 RDD[Vector] 做转化。

注意：如果某特征的方差是 0，那么标准化之后返回默认的 0.0 作为特征值。

- 案例：在下面的例子中，首先导入 libsvm 格式的数据，然后做特征标准化，标准化之后新的特征值有单位长度的标准差和/或均值。

```
import org.apache.spark.mllib.feature.{StandardScaler, StandardScalerModel}
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.util.MLUtils
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_data.txt")
val scaler1 = new StandardScaler().fit(data.map(x => x.features))
val scaler2 = new StandardScaler(withMean = true, withStd = true).fit(data.map(x =>
x.features))
// scaler3是与scaler2相同的模型，并将产生相同的变换
val scaler3 = new StandardScalerModel(scaler2.std, scaler2.mean)
// data1是单位方差。
val data1 = data.map(x => (x.label, scaler1.transform(x.features)))
 // data2是单位方差和零均值。
val data2 = data.map(x => (x.label, scaler2.transform(Vectors.dense(x.features.toArray))))
```

7.4 Normalizer（归一化）

- 归一化是指将每个独立样本做尺度变换从而是该样本具有单位 Lp 范数。这是文本分类和聚类中的常用操作。例如，两个做了L2归一化的TF-IDF向量的点积是这两个向量的cosine（余弦）相似度。

  Normalizer 的构造函数有以下参数：

  - 在Lp空间的p范数, 默认 p=2。

  Normlizer 实现了 VectorTransformer ，这个类可以对 Vector 和 RDD[Vector] 做归一化。 注意：如果输入的范数是 0，会返回原来的输入向量。

- 案例： 在下面的例子中，首先导入libsvm格式的数据，然后使用 L2 范数和 L∞ 范数归一化。

  Scala版本：API 详情参见`Normalizer` Scala 文档

  完整样例代码在 Spark repo 的 "examples/src/main/scala/org/apache/spark/examples/mllib/NormalizerExample.scala" 中。

```
import org.apache.spark.mllib.feature.Normalizer
import org.apache.spark.mllib.util.MLUtils
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_data.txt")
val normalizer1 = new Normalizer()
val normalizer2 = new Normalizer(p = Double.PositiveInfinity)
// Each sample in data1 will be normalized using $L^2$ norm.
val data1 = data.map(x => (x.label, normalizer1.transform(x.features)))
// Each sample in data2 will be normalized using $L^\infty$ norm.
val data2 = data.map(x => (x.label, normalizer2.transform(x.features))
```

7.5 ChiSqSelector（卡方选择）

- `ChiSqSelector`是指使用卡方（Chi-Squared）做特征选择。该方法操作的是有标签的类别型数据。ChiSqSelector基于卡方检验来排序数据，然后选出卡方值较大(也就是跟标签最相关)的特征（topk)。

- **模型拟合**

  `ChiSqSelector` 的构造函数有如下特征：

  - `numTopFeatures` 保留的卡方较大的特征的数量。

  ChiSqSelector.fit() 方法以具有类别特征的RDD[LabeledPoint]为输入，计算汇总统计信息，然后返回ChiSqSelectorModel，这个类将输入数据转化到降维的特征空间。

  模型实现了 `VectorTransformer` ，这个类可以在Vector和RDD[Vector]上做卡方特征选择。

  注意：也可以手工构造一个`ChiSqSelectorModel`，需要提供升序排列的特征索引。

- 案例：下面的例子说明了ChiSqSelector的基本用法。

```
import org.apache.spark.mllib.feature.ChiSqSelector
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.util.MLUtils
// 以libsvm格式加载一些数据
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_data.txt")
// 由于ChiSqSelector需要分类特征，因此将数据分离为16个相等的区间
// 即使特征是双精度，ChiSqSelector也会将每个唯一值视为一个类别
val discretizedData = data.map { lp =>
  LabeledPoint(lp.label, Vectors.dense(lp.features.toArray.map { x => (x / 16).floor }))
}
// 创建ChiSqSelector，从692个特征中选择前50个
val selector = new ChiSqSelector(50)
// Create ChiSqSelector model (selecting features)
val transformer = selector.fit(discretizedData)
// Filter the top 50 features from each feature vector
val filteredData = discretizedData.map { lp =>
  LabeledPoint(lp.label, transformer.transform(lp.features))
}
```

7.6 PCA

- 使用PCA将向量投影到低维空间的特征变换器。 您可以阅读的细节在降低维度。

- 案例：下例展示如何计算特征向量空间的主要组件，并使用主要组件将向量投影到低维空间中，同时保留向量的类标签用于计算的关联标签线性回归。

```
import org.apache.spark.mllib.feature.PCA
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.regression.{LabeledPoint, LinearRegressionWithSGD}

val data = sc.textFile("data/mllib/ridge-data/lpsa.data").map { line =>
val parts = line.split(',')
  LabeledPoint(parts(0).toDouble, Vectors.dense(parts(1).split(' ').map(_.toDouble)))
}.cache()
val splits = data.randomSplit(Array(0.6, 0.4), seed = 11L)
val training = splits(0).cache()
val test = splits(1)

val pca = new PCA(training.first().features.size / 2).fit(data.map(_.features))
val training_pca = training.map(p => p.copy(features = pca.transform(p.features)))
val test_pca = test.map(p => p.copy(features = pca.transform(p.features)))
val numIterations = 100
val model = LinearRegressionWithSGD.train(training, numIterations)
val model_pca = LinearRegressionWithSGD.train(training_pca, numIterations)
val valuesAndPreds = test.map { point =>
val score = model.predict(point.features)
  (score, point.label)
}
val valuesAndPreds_pca = test_pca.map { point =>
val score = model_pca.predict(point.features)
  (score, point.label)
}
val MSE = valuesAndPreds.map { case (v, p) => math.pow((v - p), 2) }.mean()
val MSE_pca = valuesAndPreds_pca.map { case (v, p) => math.pow((v - p), 2) }.mean()

println("Mean Squared Error = " + MSE)
println("PCA Mean Squared Error = " + MSE_pca)
```

8.0 Frequent Pattern Mining - RDD-based API（频繁模式挖掘）

- 近些年来，分析大规模数据集的第一步，就是挖掘频繁项，频繁项集，子序列或者是子结构，并且在数据挖掘领域，这已经是一项热点研究了。关于更多的相关的信息，用户可以去维基百科上了解association rule learning(关联规则的学习)这部分。

8.1 FP-growth

- **FP-growth** 算法是由韩家炜在 Han et al., Mining frequent patterns without candidate generation 提出的，“FP”代表频繁模式的意思。该算法的基本思路如下，给予一个事务数据库，**FP-Growth** 算法的第一步是每一项出现的次数，通过最小支持度进行筛选确定频繁项。不同于另一种关联算法**Apriori** 算法，**FP-growth** 算法第二步是通过产生后缀树(**FP-tree**)来存储所有的事务数据库中的项，而不是像 Apriori 算法那样花费大量的内存去产生候选项集。然后，通过遍历 **FP-Tree** 可以挖掘出频繁项集。在 **spark.mllib** 中，我们实现了并行的 **FP-growth** 算法叫做 **PFP**，正如论文Li et al., PFP: Parallel FP-growth for query recommendation 中描述的，**PFP** 将基于相同后缀的事务分发到相同的机器上，因此相比的单台机器的实现，这样有更好的扩展性。我们推荐用户读 Li et al., PFP: Parallel FP-growth for query recommendation这

篇论文去理解更多的信息。 **spark mllib** 中 **FP-growth** 算法的实现要使用到以下两个超参数:
• **minSupport** (最小支持度):一个项集被认为是频繁项集的最小支持度。例如，如果一个项总共有5个事务的数据集中，出现了3次，那么它的支持度为3/5=0.6 • **numPartitions** (分区数):分区的个数，同于将事务分配到几个分区。

- 案例

```
import org.apache.spark.mllib.fpm.FPGrowth
import org.apache.spark.rdd.RDD
//读取文件
val data = sc.textFile("data/mllib/sample_fpgrowth.txt")
//预处理并转换为RDD[Array[String]]存储
val transactions: RDD[Array[String]] = data.map(s => s.trim.split(' '))
//调用FPGrowth()方法，设定最小支持度与分区数
val fpg = new FPGrowth()
  .setMinSupport(0.2)
  .setNumPartitions(10)
 //得到FPGrowthModel
val model = fpg.run(transactions)
//遍历频繁项集
model.freqItemsets.collect().foreach { itemset =>
  println(itemset.items.mkString("[", ",", "]") + ", " + itemset.freq)
}
//设定最小置信度
val minConfidence = 0.8
 //产生关联规则
model.generateAssociationRules(minConfidence).collect().foreach { rule =>
  println(
    rule.antecedent.mkString("[", ",", "]")
      + " => " + rule.consequent .mkString("[", ",", "]")
      + ", " + rule.confidence)
}
```

8.2 关联规则

- AssociationRules(关联规则)类实现了一个并行的规则生成算法去构建产生一个单项作为结果的规则。
- 案例

```
import org.apache.spark.mllib.fpm.AssociationRules
import org.apache.spark.mllib.fpm.FPGrowth.FreqItemset
val freqItemsets = sc.parallelize(Seq(
  new FreqItemset(Array("a"), 15L),
  new FreqItemset(Array("b"), 35L),
  new FreqItemset(Array("a", "b"), 12L)
))
val ar = new AssociationRules()
  .setMinConfidence(0.8)
val results = ar.run(freqItemsets)
results.collect().foreach { rule =>
  println("[" + rule.antecedent.mkString(",")

    + "=>"
```

```
    + rule.consequent.mkString(",") + "]," + rule.confidence)
}
```

8.3 PrefixSpan

- **PrefixSpan** 是一种序列模式挖掘算法， 具体描述见论文 Pei et al., Mining Sequential Patterns by Pattern-Growth: The PrefixSpan Approach。我们推荐读者去读相关的论文去更加深入的理解的序列模式挖掘问题。

  spark.mllib 中的 **PrefixSpan** 要使用到以下的参数： • **minSupport** (最小支持度):最小支持度要求能够反映频繁序列模式。 • **maxPatternLength** (最大频繁序列的长度):频繁序列模式的最大长度。结果中将不包含任何频繁序列长度如果超过这个长度的频繁序列。

  •**maxLocalProjDBSize** (最大单机投影数据库的项数):这个参数应该与你的 **spark** 集群中 **executors** 参数设置一致。

- 案例： 下面的例子介绍了 **PrefixSpan** 在序列上运行情况(与Pei et al的论文中使用相同的标记)

  ```
  <(12)3>
  <1(32)(12)>
  <(12)5>
  <6>
  ```

  PrefixSpan 类实现了 **PrefixSpan** 算法。调用 **PrefixSpan.run** 返回存储带有频率的频繁序列的模型 PrefixSpanModel。

  ```
  import org.apache.spark.mllib.fpm.PrefixSpan
  val sequences = sc.parallelize(Seq(
    Array(Array(1, 2), Array(3)),
    Array(Array(1), Array(3, 2), Array(1, 2)),
    Array(Array(1, 2), Array(5)),
    Array(Array(6))
  ), 2).cache()
  val prefixSpan = new PrefixSpan()
    .setMinSupport(0.5)
    .setMaxPatternLength(5)
  val model = prefixSpan.run(sequences)
  model.freqSequences.collect().foreach { freqSequence =>
  println(
      freqSequence.sequence.map(_.mkString("[", ", ", "]")).mkString("[", ", ", "]") +
        ", " + freqSequence.freq)
  }
  ```

9.0 Evaluation metrics - RDD-based API（评估指标）

- ***spark.mllib*** 提供了一系列的机器学习算法可以用来从数据中挖掘信息和进行预测。当这些算法被用来建立机器学习模型时，我们同样需要根据实际应用和需求去基于某些标准评估模型的效果。***spark.mllib***同样提供了一整套指标用来满足评估机器学习模型效果的要求。

  一些机器学习算法可被归类于更广泛的机器学习应用类型，比如分类问题、回归问题、聚类问题等。这里面每一种类型都有成熟的效果评估指标并且这些指标可以从 ***spark.mllib*** 中使用，在本章中将会仔细讲述。

9.1 Classification model evaluation（分类模型评估）

- 虽然分类算法的种类有很多，但是分类模型的评估方法的原理都比较相似。在一个有监督分类问题中，对于每一个数据点都会存在一个真实的分类值（标签）和一个模型生成的预测分类值。基于这两个值，每个数据点的结果都可以被归纳为以下四个类别中的一种：
    - **True Positive**（**TP**）- 真实标签为正并且预测正确
    - **True Negative**（**TN**）- 真实标签为负并且预测正确
    - **False Positive**（**FP**）- 真实标签为负但是预测为正
    - **False Negative**（**FN**）- 真实标签为正但是预测为负

  上面四种值是大多数分类器评估指标的基础。如果仅仅依靠最基本的准确率（预测是对的还是错的）去评估一个分类器的效果的话，这往往不是一个好的指标，原因在于一个数据集可能各个类别的分布非常不平衡。举例来说，如果一个预测欺诈的模型是根据这样的数据集设计出来的：95%的数据点是非欺诈数据，5%的数据点是欺诈数据，那么一个简单的分类器去预测非欺诈时，不考虑输入的影响，它的准确率将会是95%。因此，我们经常会使用 **precision** 和 **recall** 这样的指标，因为他们会将"错误"的类型考虑进去。在大多数应用中，**precision** 和 **recall** 会被合并为一个指标来在他们之间进行一些平衡，我们称这个指标为 **F-measure**。

9.2 Binary classification（二项分类）

- 二项分类器可以用来区分数据集中的元素到两个可能的组中（如欺诈和非欺诈），而且它是多项分类器的一种特殊情况。多数的二项分类评估指标可以被推广使用到多项分类评估中。

Threshold tuning（阈值调优）

很多分类模型对于每一个类实际上会输出一个"分数"（经常是一个概率值），其中较高的分数表示更高的可能性，理解这一点非常重要。在二项分类场景中，模型会为每个类别输出一个概率：$P(Y = 1|X)$ 和 $P(Y = 0|X)$。并不是所有场景都会取更高概率的类别，有些场景下模型可能需要进行调整来使得它只有在某个概率非常高的情况下才去预测这个类别（如只有在模型预测的欺诈概率高于90%的情况下才屏蔽这笔信用卡交易）。所以，根据模型输出的概率来确定预测的类别是由预测的阈值来决定的。

调整预测阈值将会改变模型的 **precision** 和 **recall**，而且它是模型调优的重要环节。为了能够将 **precision**、**recall** 和其他评估指标根据阈值的改变是如何变化的这一情况可视化出来，一种常见的做法是通过阈值的参数化来绘制相互作对比的指标。一种图叫做 **P-R** 曲线，它会将 **(precision, recall)** 组成的点根据不同的阈值绘制出来，而另一种叫做 **receiver operating characteristic** 或者 **ROC** 曲线，它会绘制出 **(recall, false positive rate)** 组成的点。

- **可用的评估指标**

指标	定义
Precision（精确率）	$PPV = \frac{TP}{TP+FP}$
Recall（召回率）	$TPR = \frac{TP}{P} = \frac{TP}{TP+FN}$
F-measure	$F(\beta) = (1+\beta^2) \cdot \left(\frac{PPV \cdot TPR}{\beta^2 \cdot PPV + TPR}\right)$
Receiver Operating Characteristic（ROC）	$FPR(T) = \int_T^\infty P_0(T)\,dT$ $TPR(T) = \int_T^\infty P_1(T)\,dT$
Area Under ROC Curve（ROC 曲线下面积	$AUROC = \int_0^1 \frac{TP}{P} d\left(\frac{FP}{N}\right)$
Area Under Precision-Recall Curve（P-R 曲线下面积）	$AUPRC = \int_0^1 \frac{TP}{TP+FP} d\left(\frac{TP}{P}\right)$

- 案例：下面代码片段阐释了如何加载一个数据集样本，并对数据训练一个二项分类模型，最后通过几个二项评估指标来评价该模型的效果。

```
import org.apache.spark.mllib.classification.LogisticRegressionWithLBFGS
import org.apache.spark.mllib.evaluation.BinaryClassificationMetrics
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.util.MLUtils

// 以 LIBSVM 格式加载训练数据
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_binary_classification_data.txt")

// 分割数据为训练集（60%）和测试集（40%）
val Array(training, test) = data.randomSplit(Array(0.6, 0.4), seed = 11L)
training.cache()

// 运行训练算法来建模
val model = new LogisticRegressionWithLBFGS()
  .setNumClasses(2)
  .run(training)

// 清空预测阈值来使模型返回概率值
model.clearThreshold

// 使用测试集计算原始分数
val predictionAndLabels = test.map { case LabeledPoint(label, features) =>
  val prediction = model.predict(features)
  (prediction, label)
}

// 实例化指标对象
val metrics = new BinaryClassificationMetrics(predictionAndLabels)
```

```
// 按阈值分别求 precision
val precision = metrics.precisionByThreshold
precision.foreach { case (t, p) =>
  println(s"Threshold: $t, Precision: $p")
}

// 按阈值分别求 recall
val recall = metrics.recallByThreshold
recall.foreach { case (t, r) =>
  println(s"Threshold: $t, Recall: $r")
}

// Precision-Recall 曲线
val PRC = metrics.pr

// F-measure
val f1Score = metrics.fMeasureByThreshold
f1Score.foreach { case (t, f) =>
  println(s"Threshold: $t, F-score: $f, Beta = 1")
}

val beta = 0.5
val fScore = metrics.fMeasureByThreshold(beta)
f1Score.foreach { case (t, f) =>
  println(s"Threshold: $t, F-score: $f, Beta = 0.5")
}

// AUPRC
val auPRC = metrics.areaUnderPR
println("Area under precision-recall curve = " + auPRC)

// 计算 ROC 和 PR 曲线中使用的阈值
val thresholds = precision.map(_._1)

// ROC 曲线
val roc = metrics.roc

// AUROC
val auROC = metrics.areaUnderROC
println("Area under ROC = " + auROC)
```

9.3 Multiclass classification（多项分类）

- 多项分类描述了这样一种分类问题，对于每个数据点都有$M > 2$种可能的分类标签（当$M = 2$时为二项分类问题）。举例来说，区分手写样本为0到9的数字就是一种有10个可能分类的多项分类问题。

  对于多类别评估指标，积极和消极的概念跟二项分类略有不同。预测值和实际分类仍然可以是积极的或消极的，但是它们必须被归类到某种特定的类别中。每一个类别标签和预测值都会分配给多个类别中的一个，所以它对于分配的类别来说是积极的，但对于其他类别则是消极的。因此，**true positive** 发生在预测值和实际分类相吻合的时候，而 **true negative** 发生在预测值和实际分类都不属于一个类别的时候。由于这个认识，

对于一个数据样本而言存在多个 **true negative**。根据之前对于 **positive** 和 **negative** 的定义扩展到 **false negative** 和 **false positive** 时就相对直接了。

- ## Label based metrics（基于标签的指标）

  相对于只有两种可能分类的二项分类问题，多项分类问题有多种可能的类别，所以基于类别的评估指标概念则被引入。基于所有类别的 **precision** 的准确率计算方式为，任何类别被预测正确的次数（**true positive**）除以数据点的数量。基于一个类别的 **precision** 的计算方式为，某个特定的类别被预测正确的次数除以这个类别在结果中出现的次数。

- **可用的评估指标**

指标	定义
Confusion Matrix（混淆矩阵）	$C_{ij} = \sum_{k=0}^{N-1} \hat{\delta}(\mathbf{y}_k - \ell_i) \cdot \hat{\delta}(\hat{\mathbf{y}}_k - \ell_j)$ $\begin{pmatrix} \sum_{k=0}^{N-1} \hat{\delta}(\mathbf{y}_k - \ell_1) \cdot \hat{\delta}(\hat{\mathbf{y}}_k - \ell_1) & \dots & \sum_{k=0}^{N-1} \hat{\delta}(\mathbf{y}_k - \ell_1) \cdot \hat{\delta}(\hat{\mathbf{y}}_k - \ell_N) \\ \vdots & \ddots & \vdots \\ \sum_{k=0}^{N-1} \hat{\delta}(\mathbf{y}_k - \ell_N) \cdot \hat{\delta}(\hat{\mathbf{y}}_k - \ell_1) & \dots & \sum_{k=0}^{N-1} \hat{\delta}(\mathbf{y}_k - \ell_N) \cdot \hat{\delta}(\hat{\mathbf{y}}_k - \ell_N) \end{pmatrix}$
Accuracy（准确率）	$ACC = \frac{TP}{TP+FP} = \frac{1}{N}\sum_{i=0}^{N-1} \hat{\delta}\left(\hat{\mathbf{y}}_i - \mathbf{y}_i\right)$
Precision by label（标签 precision）	$PPV(\ell) = \frac{TP}{TP+FP} = \frac{\sum_{i=0}^{N-1} \hat{\delta}(\hat{\mathbf{y}}_i - \ell) \cdot \hat{\delta}(\mathbf{y}_i - \ell)}{\sum_{i=0}^{N-1} \hat{\delta}(\hat{\mathbf{y}}_i - \ell)}$
Recall by label（标签 recall）	$TPR(\ell) = \frac{TP}{P} = \frac{\sum_{i=0}^{N-1} \hat{\delta}(\hat{\mathbf{y}}_i - \ell) \cdot \hat{\delta}(\mathbf{y}_i - \ell)}{\sum_{i=0}^{N-1} \hat{\delta}(\mathbf{y}_i - \ell)}$
F-measure by label（标签 F-measure）	$F(\beta, \ell) = \left(1 + \beta^2\right) \cdot \left(\frac{PPV(\ell) \cdot TPR(\ell)}{\beta^2 \cdot PPV(\ell) + TPR(\ell)}\right)$
Weighted precision（加权 precision）	$PPV_w = \frac{1}{N}\sum_{\ell \in L} PPV(\ell) \cdot \sum_{i=0}^{N-1} \hat{\delta}(\mathbf{y}_i - \ell)$
Weighted recall（加权 recall）	$TPR_w = \frac{1}{N}\sum_{\ell \in L} TPR(\ell) \cdot \sum_{i=0}^{N-1} \hat{\delta}(\mathbf{y}_i - \ell)$
Weighted F-measure（加权 F-measure）	$F_w(\beta) = \frac{1}{N}\sum_{\ell \in L} F(\beta, \ell) \cdot \sum_{i=0}^{N-1} \hat{\delta}(\mathbf{y}_i - \ell)$

- 案例：下面代码片段阐释了如何加载一个数据集样本，并对数据训练一个多项分类模型，最后通过多项分类评估指标评估模型效果。

```
import org.apache.spark.mllib.classification.LogisticRegressionWithLBFGS
import org.apache.spark.mllib.evaluation.MulticlassMetrics
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.util.MLUtils
// 以 LIBSVM 格式加载训练数据
val data = MLUtils.loadLibSVMFile(sc,
"data/mllib/sample_multiclass_classification_data.txt")
// 分割数据为训练集（60%）和测试集（40%）
val Array(training, test) = data.randomSplit(Array(0.6, 0.4), seed = 11L)
training.cache()
// 运行训练算法来建模
val model = new LogisticRegressionWithLBFGS()
  .setNumClasses(3)
  .run(training)
// 使用测试集计算原始分数
val predictionAndLabels = test.map { case LabeledPoint(label, features) =>
```

```
  val prediction = model.predict(features)
  (prediction, label)
}
// 实例化指标对象
val metrics = new MulticlassMetrics(predictionAndLabels)
// 混淆矩阵
println("Confusion matrix:")
println(metrics.confusionMatrix)
// 总体统计结果
val accuracy = metrics.accuracy
println("Summary Statistics")
println(s"Accuracy = $accuracy")
// 基于类别计算 Precision
val labels = metrics.labels
labels.foreach { l =>
  println(s"Precision($l) = " + metrics.precision(l))
}
// 基于类别计算 Recall
labels.foreach { l =>
  println(s"Recall($l) = " + metrics.recall(l))
}
// 基于类别计算 False positive rate
labels.foreach { l =>
  println(s"FPR($l) = " + metrics.falsePositiveRate(l))
}
// 基于类别计算 F-measure
labels.foreach { l =>
  println(s"F1-Score($l) = " + metrics.fMeasure(l))
}
// 加权统计结果
println(s"Weighted precision: ${metrics.weightedPrecision}")
println(s"Weighted recall: ${metrics.weightedRecall}")
println(s"Weighted F1 score: ${metrics.weightedFMeasure}")
println(s"Weighted false positive rate: ${metrics.weightedFalsePositiveRate}")
```

9.4 Multilabel classification（多标签分类）

- 多标签分类问题指的是将一个数据集中的每一个样本映射到一组分类标签中去。在这一类分类问题中，各个标签所包含的样本并不是互斥的。举例来说，将一组新闻稿分类到不同的主题中去，一篇稿子可能既属于科技类，又属于政治类。

  由于标签不是彼此互斥，预测值和真实的分类标签就成为了标签集合的向量，而非标签的向量。于是多标签评估指标就可以从基本的 **precision**、**recall** 等概念扩展到对集合的操作上。例如，对于某个类的 true positive 就发生在当这个类别存在于某个数据点的预测值集合中，并且也存在于它的真实类别集合中时。

- 可评价指标

指标	定义
Precision	$\frac{1}{N}\sum_{i=0}^{N-1}\frac{\lvert P_i\cap L_i\rvert}{\lvert P_i\rvert}$
Recall	$\frac{1}{N}\sum_{i=0}^{N-1}\frac{\lvert L_i\cap P_i\rvert}{\lvert L_i\rvert}$
Accuracy	$\frac{1}{N}\sum_{i=0}^{N-1}\frac{\lvert L_i\cap P_i\rvert}{\lvert L_i\rvert+\lvert P_i\rvert-\lvert L_i\cap P_i\rvert}$
Precision by label	$PPV(\ell)=\frac{TP}{TP+FP}=\frac{\sum_{i=0}^{N-1}I_{P_i}(\ell)\cdot I_{L_i}(\ell)}{\sum_{i=0}^{N-1}I_{P_i}(\ell)}$
Recall by label	$TPR(\ell)=\frac{TP}{P}=\frac{\sum_{i=0}^{N-1}I_{P_i}(\ell)\cdot I_{L_i}(\ell)}{\sum_{i=0}^{N-1}I_{L_i}(\ell)}$
F1-measure by label	$F1(\ell)=2\cdot\left(\frac{PPV(\ell)\cdot TPR(\ell)}{PPV(\ell)+TPR(\ell)}\right)$
Hamming Loss	$\frac{1}{N\cdot\lvert L\rvert}\sum_{i=0}^{N-1}\lvert L_i\rvert+\lvert P_i\rvert-2\lvert L_i\cap P_i\rvert$
Subset Accuracy	$\frac{1}{N}\sum_{i=0}^{N-1}I_{\{L_i\}}(P_i)$
F1 Measure	$\frac{1}{N}\sum_{i=0}^{N-1}2\frac{\lvert P_i\cap L_i\rvert}{\lvert P_i\rvert\cdot\lvert L_i\rvert}$
Micro precision	$\frac{TP}{TP+FP}=\frac{\sum_{i=0}^{N-1}\lvert P_i\cap L_i\rvert}{\sum_{i=0}^{N-1}\lvert P_i\cap L_i\rvert+\sum_{i=0}^{N-1}\lvert P_i-L_i\rvert}$
Micro recall	$\frac{TP}{TP+FN}=\frac{\sum_{i=0}^{N-1}\lvert P_i\cap L_i\rvert}{\sum_{i=0}^{N-1}\lvert P_i\cap L_i\rvert+\sum_{i=0}^{N-1}\lvert L_i-P_i\rvert}$
Micro F1 Measure	$2\cdot\frac{TP}{2\cdot TP+FP+FN}=2\cdot\frac{\sum_{i=0}^{N-1}\lvert P_i\cap L_i\rvert}{2\cdot\sum_{i=0}^{N-1}\lvert P_i\cap L_i\rvert+\sum_{i=0}^{N-1}\lvert L_i-P_i\rvert+\sum_{i=0}^{N-1}\lvert P_i-L_i\rvert}$

- 案例：以下代码片段阐释了如何去评估一个多标签分类器的效果。下面的多标签分类例子中使用了伪造的预测数据和标签数据。

  对于每一篇文章的预测结果为：

  - 文章 0 - 预测值为 0, 1 - 类别为 0, 2
  - 文章 1 - 预测值为 0, 2 - 类别为 0, 1
  - 文章 2 - 预测值为空 - 类别为 0
  - 文章 3 - 预测值为 2 - 类别为 2
  - 文章 4 - 预测值为 2, 0 - 类别为 2, 0
  - 文章 5 - 预测值为 0, 1, 2 - 类别为 0, 1
  - 文章 6 - 预测值为 1 - 类别为 1, 2

对于每一类别的预测结果为：

- 类别 0 - 文章 0, 1, 4, 5 (一共 4 篇)
- 类别 1 - 文章 0, 5, 6 (一共 3 篇)
- 类别 2 - 文章 1, 3, 4, 5 (一共 4 篇)

每一类别实际包含的文章为：

- 类别 0 - 文章 0, 1, 2, 4, 5 (一共 5 篇)
- 类别 1 - 文章 1, 5, 6 (一共 3 篇)
- 类别 2 - 文章 0, 3, 4, 6 (一共 4 篇)

```
import org.apache.spark.mllib.evaluation.MultilabelMetrics
import org.apache.spark.rdd.RDD

val scoreAndLabels: RDD[(Array[Double], Array[Double])] = sc.parallelize(
  Seq((Array(0.0, 1.0), Array(0.0, 2.0)),
    (Array(0.0, 2.0), Array(0.0, 1.0)),
    (Array.empty[Double], Array(0.0)),
    (Array(2.0), Array(2.0)),
    (Array(2.0, 0.0), Array(2.0, 0.0)),
    (Array(0.0, 1.0, 2.0), Array(0.0, 1.0)),
    (Array(1.0), Array(1.0, 2.0))), 2)

// 实例化指标对象
val metrics = new MultilabelMetrics(scoreAndLabels)

// 总体统计结果
println(s"Recall = ${metrics.recall}")
println(s"Precision = ${metrics.precision}")
println(s"F1 measure = ${metrics.f1Measure}")
println(s"Accuracy = ${metrics.accuracy}")

// 基于每一标签的统计结果
metrics.labels.foreach(label =>
  println(s"Class $label precision = ${metrics.precision(label)}"))
metrics.labels.foreach(label => println(s"Class $label recall = ${metrics.recall(label)}"))
metrics.labels.foreach(label => println(s"Class $label F1-score =
${metrics.f1Measure(label)}"))

// Micro stats
println(s"Micro recall = ${metrics.microRecall}")
println(s"Micro precision = ${metrics.microPrecision}")
println(s"Micro F1 measure = ${metrics.microF1Measure}")

// Hamming loss
println(s"Hamming loss = ${metrics.hammingLoss}")

// Subset accuracy
println(s"Subset accuracy = ${metrics.subsetAccuracy}")
```

9.5 Ranking system（排名系统）

- 排名算法（通常被作为推荐系统算法）的作用是基于一些训练数据给用户返回一个相关物品或文章的集合。相关性的定义会根据不同的应用场景而有所差异。排名系统评估指标则用于在不同的场景中量化排名或者推荐的效果。某些指标会把系统推荐的文章集合跟实际相关的文章集合做对比，而另外一些指标会显式地结合数值打分。
- 可用指标

一个排名系统通常会处理一个由 M 个用户组成的集合

$$U = \{u_0, u_1, \ldots, u_{M-1}\}$$

每一个用户 (u_i) 有一个由 N 个真正相关的文章组成的集合

$$D_i = \{d_0, d_1, \ldots, d_{N-1}\}$$

并且还有一个由 Q 个推荐文章组成的有序集合，按照相关度降序排序

$$R_i = [r_0, r_1, \ldots, r_{Q-1}]$$

排序系统的目标就是给每个用户提供与其相关度最高的若干个文章集合。集合的相关度以及算法的有效性可以用下列评估指标进行度量。

$$rel_D(r) = \begin{cases} 1 & \text{if } r \in D, \\ 0 & \text{otherwise.} \end{cases}$$

度量	定义	说明
Precision at k	$p(k) = \frac{1}{M}\sum_{i=0}^{M-1}\frac{1}{k}\sum_{j=0}^{\min(\lvert D\rvert,k)-1} rel_{D_i}(R_i(j))$	Precision at k 是用来度量对所有用户来说，平均前k个推荐的文章里有几个在真正相关的文章集合中出现。这个指标不考虑推荐的顺序问题。
Mean Average Precision	$MAP = \frac{1}{M}\sum_{i=0}^{M-1}\frac{1}{\lvert D_i\rvert}\sum_{j=0}^{Q-1}\frac{rel_{D_i}(R_i(j))}{j+1}$	MAP 是用来度量推荐的文章集合里有几个出现在真正相关的文章集合中，并且考虑推荐的顺序性（比如文章的相关性越高，它出错的惩罚就越大）。
Normalized Discounted Cumulative Gain	$NDCG(k) = \frac{1}{M}\sum_{i=0}^{M-1}\frac{1}{IDCG(D_i,k)}\sum_{j=0}^{n-1}\frac{rel_{D_i}(R_i(j))}{\ln(j+1)}$ Where $n = \min(\max(\lvert R_i\rvert, \lvert D_i\rvert), k)$ $IDCG(D,k) = \sum_{j=0}^{\min(\lvert D\rvert,k)-1}\frac{1}{\ln(j+1)}$	NDCG at k 是用来度量对所有用户来说，平均前k个推荐的文章里有几个在真正相关的文章集合中出现。跟 Precision at k 不同，这个指标考虑推荐的顺序性（假设文章按照相关性降序排序）。

- 案例 下面代码片段阐释了如何加载一个数据集样本，并使用数据训练一个交替最小二乘法推荐算法模型，最后通过几个排名评估指标来评价推荐系统效果。下面简单总结了一下使用的方法。

  对于影片的打分在 1 - 5 分区间：

  - 5: 必看
  - 4: 会喜欢
  - 3: 还可以
  - 2: 不怎么样
  - 1: 烂片

  那么如果预测的打分小于3分，我们将不会推荐这部影片。我们使用下面映射关系来表示置信分数：

  - 5 -> 2.5
  - 4 -> 1.5
  - 3 -> 0.5
  - 2 -> -0.5
  - 1 -> -1.5

  这个映射关系表示没有被用户察觉到的影片通常在“还可以”和“不怎么样”之间。权重0在这里的语义表示“跟这个影片从来没有过任何互动”。

```
import org.apache.spark.mllib.evaluation.{RankingMetrics, RegressionMetrics}
import org.apache.spark.mllib.recommendation.{ALS, Rating}

// 读取打分数据
val ratings = spark.read.textFile("data/mllib/sample_movielens_data.txt").rdd.map { line =>
  val fields = line.split("::")
  Rating(fields(0).toInt, fields(1).toInt, fields(2).toDouble - 2.5)
}.cache()
```

```
// 将打分映射到0或者1，1表示影片应该被推荐
val binarizedRatings = ratings.map(r => Rating(r.user, r.product,
  if (r.rating > 0) 1.0 else 0.0)).cache()

// 评分的统计结果
val numRatings = ratings.count()
val numUsers = ratings.map(_.user).distinct().count()
val numMovies = ratings.map(_.product).distinct().count()
println(s"Got $numRatings ratings from $numUsers users on $numMovies movies.")

// 建模
val numIterations = 10
val rank = 10
val lambda = 0.01
val model = ALS.train(ratings, rank, numIterations, lambda)

// 定义一个函数将打分映射到0和1区间
def scaledRating(r: Rating): Rating = {
  val scaledRating = math.max(math.min(r.rating, 1.0), 0.0)
  Rating(r.user, r.product, scaledRating)
}

// 获取排序后对于每个用户前十个预测值，并映射到0和1区间
val userRecommended = model.recommendProductsForUsers(10).map { case (user, recs) =>
  (user, recs.map(scaledRating))
}

// 假设任何被用户打分为3分或更高（映射到1）的影片为一个相关的影片
// 并与前十个最相关的影片做对比
val userMovies = binarizedRatings.groupBy(_.user)
val relevantDocuments = userMovies.join(userRecommended).map { case (user, (actual,
predictions)) =>
  (predictions.map(_.product), actual.filter(_.rating > 0.0).map(_.product).toArray)
}

// 实例化一个评估指标模型
val metrics = new RankingMetrics(relevantDocuments)

// Precision at K
Array(1, 3, 5).foreach { k =>
  println(s"Precision at $k = ${metrics.precisionAt(k)}")
}

// Mean average precision
println(s"Mean average precision = ${metrics.meanAveragePrecision}")

// Normalized discounted cumulative gain
Array(1, 3, 5).foreach { k =>
  println(s"NDCG at $k = ${metrics.ndcgAt(k)}")
}


// 获得每个数据点的 Prediction
```

```
val allPredictions = model.predict(ratings.map(r => (r.user, r.product))).map(r =>
((r.user,
  r.product), r.rating))
val allRatings = ratings.map(r => ((r.user, r.product), r.rating))
val predictionsAndLabels = allPredictions.join(allRatings).map { case ((user, product),
(predicted, actual)) =>
  (predicted, actual)
}

// 使用回归评估指标获得 RMSE
val regressionMetrics = new RegressionMetrics(predictionsAndLabels)
println(s"RMSE = ${regressionMetrics.rootMeanSquaredError}")

// R-squared
println(s"R-squared = ${regressionMetrics.r2}")
```

9.6 Regression model evaluation（回归模型评估）

- **Regression analysis**（回归分析）被用于从一系列自变量中预测连续的因变量的场景中。
- 可用的指标

指标	定义
Mean Squared Error (MSE)	$MSE = \frac{\sum_{i=0}^{N-1}(y_i - \hat{y}_i)^2}{N}$
Root Mean Squared Error (RMSE)	$RMSE = \sqrt{\frac{\sum_{i=0}^{N-1}(y_i - \hat{y}_i)^2}{N}}$
Mean Absolute Error (MAE)	$MAE = \sum_{i=0}^{N-1} \lvert y_i - \hat{y}_i \rvert$
Coefficient of Determination (R^2)	$R^2 = 1 - \frac{MSE}{\mathrm{VAR}(y) \cdot (N-1)} = 1 - \frac{\sum_{i=0}^{N-1}(y_i - \hat{y}_i)^2}{\sum_{i=0}^{N-1}(y_i - \bar{y})^2}$
Explained Variance	$1 - \frac{\mathrm{VAR}(y - \hat{y})}{\mathrm{VAR}(y)}$

- 案例 下面代码片段阐释了如何加载一个数据集样本，并使用数据训练一个线性回归算法模型，最后通过几个回归评估指标来对模型效果进行评价的过程。

```
import org.apache.spark.mllib.evaluation.RegressionMetrics
import org.apache.spark.mllib.linalg.Vector
import org.apache.spark.mllib.regression.{LabeledPoint, LinearRegressionWithSGD}

// 加载数据
val data = spark

  .read.format("libsvm").load("data/mllib/sample_linear_regression_data.txt")
```

```
  .rdd.map(row => LabeledPoint(row.getDouble(0), row.get(1).asInstanceOf[Vector]))
  .cache()

// 建模
val numIterations = 100
val model = LinearRegressionWithSGD.train(data, numIterations)

// 获取预测值
val valuesAndPreds = data.map{ point =>
  val prediction = model.predict(point.features)
  (prediction, point.label)
}

// 实例化一个回归指标对象
val metrics = new RegressionMetrics(valuesAndPreds)

// Squared error
println(s"MSE = ${metrics.meanSquaredError}")
println(s"RMSE = ${metrics.rootMeanSquaredError}")

// R-squared
println(s"R-squared = ${metrics.r2}")

// Mean absolute error
println(s"MAE = ${metrics.meanAbsoluteError}")

// Explained variance
println(s"Explained variance = ${metrics.explainedVariance}")
```

10.0 PMML model export - RDD-based API（PMML模型导出）

- ## spark.mllib支持的模型

Spark.mllib支持模型导出到Predictive Model Markup Language（预测模型标记语言）。

下表列出了可以导出到PMML的spark.mllib模型及其等效的PMML模型。

Spark.mllib模型	PMML模型
KMeansModel	ClusteringModel
LinearRegressionModel	RegressionModel (functionName="regression")
RidgeRegressionModel	RegressionModel (functionName="regression")
LassoModel	RegressionModel (functionName="regression")
SVMModel	RegressionModel (functionName="classification" normalizationMethod="none")
Binary LogisticRegressionModel	RegressionModel (functionName="classification" normalizationMethod="logit")

- 案例 将支持的模型（见上表）导出到PMML，只需调用model.toPMML。

  除了将PMML模型导出为String（model.toPMML，如上例所示），您也可以将PMML模型导出为其他格式。

  这里是一个建立KMeansModel并以PMML格式打印出来的完整示例：

```
import org.apache.spark.mllib.clustering.KMeans
import org.apache.spark.mllib.linalg.Vectors

// 加载并解析数据
val data = sc.textFile("data/mllib/kmeans_data.txt")
val parsedData = data.map(s =>Vectors.dense(s.split(' ').map(_.toDouble))).cache()

// 使用KMeans将数据分成两类
val numClusters = 2
val numIterations = 20
val clusters =KMeans.train(parsedData, numClusters, numIterations)

// 以PMML格式导出一个字符串
println("PMML Model:\n" + clusters.toPMML)
// 将模型导出为PMML格式的本地文件
clusters.toPMML("/tmp/kmeans.xml")
// 以PMML格式将模型导出到分布式文件系统的目录上
clusters.toPMML(sc, "/tmp/kmeans")
// 将模型导出为PMML格式的输出流
clusters.toPMML(System.out)
```

11.0 Optimization - RDD-based API（最优化）

Mathematical description（数学描述）

- Gradient descent（梯度下降）
- Stochastic gradient descent （SGD-随机梯度下降）

- 在Mllib中的实现Gradient descent and stochastic gradient descent（梯度下降和随机梯度下降）包括作为MLlib中的低级原函数的随机子梯度下降（SGD）的梯度下降方法，其中开发各种ML算法，参见线性方法部分。 SGD类GradientDescent可以设置以下参数：
    - Gradient 渐变是计算正在优化的功能的随机梯度的类，即相对于单个训练示例，以当前参数值计算。 MLlib包括常见损失函数的梯度类，例如hinge, logistic, least-squares（最小二乘法）。梯度类将训练样本，其标签和当前参数值作为输入。
    - 更新器是对于给定的损耗部分的梯度，执行实际的梯度下降步骤，即更新每个迭代中的权重。更新者还负责从正规化部分执行更新。 MLlib包括没有正则化的情况的更新程序，以及L1和L2正则化程序。
    - stepSize是一个标量值，表示梯度下降的初始步长。 MLlib中的所有更新者都使用等于 的第t步的步长。
    - numIterations是要运行的迭代次数。
    - regParam是使用L1或L2正则化时的正则化参数。
    - miniBatchFraction是在每次迭代中采样的总数据的一部分，以计算梯度方向。
        - 采样仍然需要遍历整个RDD，因此减少miniBatchFraction可能无法加快优化。当梯度计算成本高时，用户将会看到最大的加速，因为只有选定的样本用于计算梯度。

- ## L-BFGS

  L-BFGS目前只是MLlib中的一个低级优化原语。 如果要在各种ML算法（如线性回归和逻辑回归）中使用L-BFGS，则必须将目标函数的渐变和更新器传递给优化器，而不是使用诸如LogisticRegressionWithSGD之类的培训API。 参见下面的例子。 将在下一个版本中解决。

  使用L1Updater的L1正则化将不起作用，因为L1Updater中的软阈值逻辑被设计用于梯度下降。 请参阅开发人员的说明。

  L-BFGS方法LBFGS.runLBFGS具有以下参数：

  - Gradient是一个类，它以当前参数值计算正在优化的目标函数的梯度，即相对于单个训练示例。 MLlib包括常见损失函数的梯度类，例如铰链，后勤，最小二乘法。梯度类将训练样本，其标签和当前参数值作为输入。
  - 更新器是一种计算L-BFGS正则化部分的目标函数的梯度和损耗的类。 MLlib包括没有正则化的情况的更新程序，以及L2正则化程序。
  - numCorrections是L-BFGS更新中使用的更正次数。建议10。
  - maxNumIterations是L-BFGS可以运行的最大迭代次数。
  - regParam是使用正则化时的正则化参数。
  - converTol控制当L-BFGS被认为收敛时仍然允许多少相对变化。这必须是非负的。较低的值较不容忍，因此通常会导致更多的迭代运行。该值考察了Breeze LBFGS中的平均改善度和梯度范数。

  返回是包含两个元素的元组。 第一个元素是包含每个要素权重的列矩阵，第二个元素是包含为每次迭代计算的损失的数组。

  这是使用L-BFGS优化器来训练二元逻辑回归与L2正则化的一个例子。

```
import org.apache.spark.mllib.classification.LogisticRegressionModel
import org.apache.spark.mllib.evaluation.BinaryClassificationMetrics
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.optimization.{LBFGS, LogisticGradient, SquaredL2Updater}
import org.apache.spark.mllib.util.MLUtils

val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_data.txt")

val numFeatures = data.take(1)(0).features.size
```

```
//将数据拆分为训练集（60%）和测试集（40%）。
val splits = data.randomSplit(Array(0.6, 0.4), seed = 11L)

// 将1作为截距附加到训练数据中
val training = splits(0).map(x => (x.label, MLUtils.appendBias(x.features))).cache()

val test = splits(1)

// 运行训练数据加算法来构建模型
val numCorrections = 10
val convergenceTol = 1e-4
val maxNumIterations = 20
val regParam = 0.1
val initialWeightsWithIntercept = Vectors.dense(new Array[Double](numFeatures + 1))

val (weightsWithIntercept, loss) = LBFGS.runLBFGS(
  training,
  new LogisticGradient(),
  new SquaredL2Updater(),
  numCorrections,
  convergenceTol,
  maxNumIterations,
  regParam,
  initialWeightsWithIntercept)

val model = new LogisticRegressionModel(
  Vectors.dense(weightsWithIntercept.toArray.slice(0, weightsWithIntercept.size - 1)),
  weightsWithIntercept(weightsWithIntercept.size - 1))

// 清除默认阈值
model.clearThreshold()

// 计算测试集上的原始分数
val scoreAndLabels = test.map { point =>
  val score = model.predict(point.features)
  (score, point.label)
}

// 获取评估指标
val metrics = new BinaryClassificationMetrics(scoreAndLabels)
val auROC = metrics.areaUnderROC()

println("Loss of each step in training process")
loss.foreach(println)
println("Area under ROC = " + auROC)
```