亚太研究院

Spark 官方文档翻译

Spark 机器学习库

（V1.1.0）

翻译者　吴洪泽

Spark 官方文档翻译团成员

前 言

世界上第一个Spark 1.1.0 中文文档问世了！

伴随着大数据相关技术和产业的逐步成熟，继Hadoop之后，Spark技术以集大成的无可比拟的优势，发展迅速，将成为替代Hadoop的下一代云计算、大数据核心技术。

Spark是当今大数据领域最活跃最热门的高效大数据通用计算平台，基于RDD，Spark成功的构建起了一体化、多元化的大数据处理体系，在“One Stack to rule them all”思想的引领下，Spark成功的使用Spark SQL、Spark Streaming、MLLib、GraphX近乎完美的解决了大数据中Batch Processing、Streaming Processing、Ad-hoc Query等三大核心问题，更为美妙的是在Spark中Spark SQL、Spark Streaming、MLLib、GraphX四大子框架和库之间可以无缝的共享数据和操作，这是当今任何大数据平台都无可匹敌的优势。

在实际的生产环境中，世界上已经出现很多一千个以上节点的Spark集群，以eBay为例，eBay的Spark集群节点已经超过2000个，Yahoo 等公司也在大规模的使用Spark，国内的淘宝、腾讯、百度、网易、京东、华为、大众点评、优酷土豆等也在生产环境下深度使用Spark。2014 Spark Summit上的信息，Spark已经获得世界 20 家顶级公司的支持，这些公司中包括Intel、IBM等，同时更重要的是包括了最大的四个Hadoop发行商，都提供了对Spark非常强有力的支持。

与Spark火爆程度形成鲜明对比的是Spark人才的严重稀缺，这一情况在中国尤其严重，这种人才的稀缺，一方面是由于Spark技术在 2013、2014 年才在国内的一些大型企业里面被逐步应用，另一方面是由于匮乏Spark相关的中文资料和系统化的培训。为此，Spark亚太研究院和 51CTO联合推出了“Spark亚太研究院决胜大数据时代 100 期公益大讲堂”，来推动Spark技术在国内的普及及落地。

具体视频信息请参考 http://edu.51cto.com/course/course_id-1659.html

与此同时，为了向Spark学习者提供更为丰富的学习资料，Spark亚太研究院发起并号召，结合网络社区的力量构建了Spark中文文档专家翻译团队，历经 1 个月左右的艰苦努力和反复修改，Spark中文文档V1.1 终于完成。尤其值得一提的是，在此次中文文档的翻译期间，Spark官方团队发布了Spark 1.1.0 版本，为了让学习者了解到最新的内容，Spark中文文档专家翻译团队主动提出基于最新的Spark 1.1.0 版本，更新了所有已完成的翻译内容，在此，我谨代表Spark亚太研究院及广大Spark学习爱好者向专家翻译团队所有成员热情而专业的工作致以深刻的敬意！

当然，作为世界上第一份相对系统的Spark中文文档，不足之处在所难免，大家有任何建议或者意见都可以发邮件到marketing@sparkinchina.com ;同时如果您想加入

Spark中文文档翻译团队，也请发邮件到marketing@sparkinchina.com进行申请；Spark中文文档的翻译是一个持续更新的、不断版本迭代的过程，我们会尽全力给大家提供更高质量的Spark中文文档翻译。

最后，也是最重要的，请允许我荣幸的介绍一下我们的Spark中文文档第一个版本翻译的专家团队成员，他们分别是（排名不分先后）：

- 傅智勇，《快速开始(v1.1.0)》（和唐海东翻译的是同一主题，大家可以对比参考）
- 吴洪泽,《Spark机器学习库（v1.1.0)》（其中聚类和降维部分是蔡立宇翻译）
- 武扬，《在Yarn上运行Spark (v1.1.0)》《Spark 调优(v1.1.0)》
- 徐骄，《Spark配置(v1.1.0)》《Spark SQL编程指南(v1.1.0)》（Spark SQL和韩保礼翻译的是同一主题，大家可以对比参考）
- 蔡立宇，《Bagel 编程指南(v1.1.0)》
- harli，《Spark 编程指南（v1.1.0)》
- 吴卓华，《图计算编程指南(1.1.0)》
- 樊登贵，《EC2(v1.1.0)》《Mesos(v1.1.0)》
- 韩保礼，《Spark SQL编程指南(v1.1.0)》（和徐骄翻译的是同一主题，大家可以对比参考）
- 颜军，《文档首页(v1.1.0)》
- Jack Niu，《Spark实时流处理编程指南(v1.1.0)》
- 俞杭军，《sbt-assembly》《使用Maven编译Spark(v1.1.0)》
- 唐海东，《快速开始(v1.1.0)》（和傅智勇翻译的是同一主题，大家可以对比参考）
- 刘亚卿，《硬件配置(v1.1.0)》《Hadoop 第三方发行版(v1.1.0)》《给Spark提交代码(v1.1.0)》
- 耿元振《集群模式概览(v1.1.0)》《监控与相关工具(v1.1.0)》《提交应用程序(v1.1.0)》
- 王庆刚，《Spark作业调度(v1.1.0)》《Spark安全(v1.1.0)》
- 徐敬丽，《Spark Standalone 模式 (v1.1.0)》

另外关于Spark API的翻译正在进行中，敬请大家关注。

Life is short, You need Spark!

Spark亚太研究院院长 王家林
2014 年 10 月

Spark 亚太研究院决胜大数据时代 100 期公益大讲堂

简　介

作为下一代云计算的核心技术，Spark性能超Hadoop百倍，算法实现仅有其 1/10 或 1/100,是可以革命Hadoop的目前唯一替代者，能够做Hadoop做的一切事情，同时速度比Hadoop快了 100 倍以上。目前Spark已经构建了自己的整个大数据处理生态系统，国外一些大型互联网公司已经部署了Spark。甚至连Hadoop的早期主要贡献者Yahoo现在也在多个项目中部署使用Spark；国内的淘宝、优酷土豆、网易、Baidu、腾讯、皮皮网等已经使用Spark技术用于自己的商业生产系统中，国内外的应用开始越来越广泛。Spark正在逐渐走向成熟，并在这个领域扮演更加重要的角色，刚刚结束的 2014 Spark Summit上的信息，Spark已经获得世界 20 家顶级公司的支持，这些公司中包括Intel、IBM等，同时更重要的是包括了最大的四个Hadoop发行商都提供了对非常强有力的支持Spark的支持.

鉴于Spark的巨大价值和潜力，同时由于国内极度缺乏Spark人才，Spark亚太研究院在完成了对Spark源码的彻底研究的同时，不断在实际环境中使用Spark的各种特性的基础之上，推出了Spark亚太研究院决胜大数据时代 100 期公益大讲堂，希望能够帮助大家了解Spark的技术。同时，对Spark人才培养有近一步需求的企业和个人，我们将以公开课和企业内训的方式，来帮助大家进行Spark技能的提升。同样，我们也为企业提供一体化的顾问式服务及Spark一站式项目解决方案和实施方案。

Spark亚太研究院决胜大数据时代 100 期公益大讲堂是国内第一个Spark课程免费线上讲座，每周一期，从 7 月份起，每周四晚 20:00-21:30，与大家不见不散！老师将就Spark内核剖析、源码解读、性能优化及商业实战案例等精彩内容与大家分享，干货不容错过！

时间：从 7 月份起，每周一期，每周四晚 20:00-21:30
形式：腾讯课堂在线直播
学习条件：对云计算大数据感兴趣的技术人员
课程学习地址：http://edu.51cto.com/course/course_id-1659.html

Spark 机器学习库（MLlib)1.1.0

（翻译者：吴洪泽）

Machine Learning Library (MLlib)，原文档链接：http://spark.apache.org/docs/latest/mllib-guide.html

目录

第 1 章 综述

MLlib 是 Spark 的规模自适应机器学习库，由通用的机器学习算法和工具组成，包括分类，回归，聚类，协同过滤，降维，以及底层优化组件等。列示如下：

- 数据类型
- 基本统计分析
  - 概要统计
  - 相关性
  - 分层抽样
  - 假设检验
  - 随机数据生成
- 分类和回归
  - 线性模型(支持向量机，逻辑回归，线性回归)
  - 决策树
  - 朴素贝叶斯
- 协同过滤
  - 交替最小二乘法（ALS）
- 聚类
  - K-means 聚类
- 降维
  - 奇异值分解（SVD）
  - 主成份分析（PCA）
- 特征提取和变换
- 优化（开发者）
  - 随机梯度下降算法
  - 有限内存的 BFGS 算法 (L-BFGS)

MLlib处于开发发展中 ,含有 Experimental/DeveloperApi标记的API会在将来版本中被调整，下面的移植指导将解释发布版本之间的所有差异。

1.1 依赖

MLlib 使用了线性代数包 Breeze ,Breeze 依赖于 netlib-java 和 jblas。而 netlib-java 和 jblas 又依赖于原生的 Fortan 执行单元。所以 ,节点中需要安装 gfortran 运行库。MLlib 若找不到上面这些库，就会抛出一个链接错误。

源于 License 问题，默认配置时，我们未添加 MLlib 所依赖的 netlib-java 原生库。运

行时若找不到原生库，你会收到一个警告信息。为了使用 netlib-java 的原生库，请加上参数-Pnetlib-lgpl 编译 spark，或者在您的项目中添加 com.github.fommil.netlib:all:1.1.2 依赖。

如果您想使用优化过的BLAS/LAPACK库，比如OpenBLAS，请链接到其相应的动态库 /usr/lib/libblas.so.3和 /usr/lib/liblapack.so.3。在工作节点上，BLAS/LAPACK库应该以非多线程方式编译。

想要在 Python 中使用 MLlib，您需要 Numpy 1.4 及其以上版本。

1.2 移植指导

1.2.1 从 1.0 到 1.1

在 MLlib 1.1 版本中，唯一变化的 API 是 DecisionTree，但在 1.1 版本中，它仍然是一个实验性 API。

1. （关键变化）为了与 scikit-learn 和 ipart 中对树的实现相匹配，树的深度（tree depth）的含义被改变。在 MLlib 1.0 版本中，一个深度为 1 的树含有一个叶子节点，一个深度为 2 的树含有一个根节点和两个叶子节点。但在 MLlib 1.1 版本中，一个深度为 0 的树含有一个叶子节点，而一个深度为 1 的树含有一个根节点和两个叶子节点。这里的深度可由 maxDepth 参数指定，它或通过 Strategy 构造函数传入，或通过 DecisionTree 的 trainClassifier 和 trainRegressor 方法传入。
2. （非关键变化）我们建议使用新添加的 trainClassifier 和 trainRegressor 方法来构建一个 DecisionTree，而不是旧的参数化类 Strategy。新的训练方法严格区分了聚类和回归，并且使用特定参数类型替换了 String 类型。

新添加且建议使用的trainClassifier和trainRegressor示例请参见决策树指南章节（Decision Trees Guide）。

1.2.2 从 0.9 到 1.0

在 MLlib 1.0 版本中，为了以一致的方式处理密集输入或稀疏输入，我们添加了几个关键变化。如果您的数据是稀疏的，为了获得存储和计算时的稀疏化处理优势，请以稀疏格式而不是密集格式存储输入数据。详细描述如下：

Scala

我们过去使用Array[Double]来代表一个特征向量，但在 1.0 版本被 Vector替代。算法的输入参数类型从过去的RDD[Array[Double]]变为现在的RDD[Vector]。LabeledPoint现在是(Double, Vector)的封装类而不再是 (Double, Array[Double]) 的封装类。从

Array[Double]改变到Vector是直接明了的：

```
import org.apache.spark.mllib.linalg.{Vector, Vectors}


val array: Array[Double] = ... // a double array

val vector: Vector = Vectors.dense(array) // a dense vector
```

Vectors 提供了工厂方法来创建稀疏向量。

注意：Scala语言默认引入的是 scala.collection.immutable.Vector，为了使用MLlib的Vector，你必须显示引入org.apache.spark.mllib.linalg.Vector。

Java

我们过去使用double[]来代表一个特征向量，但在 1.0 版本中使用 Vector。算法的输入参数类型从过去的RDD<double[]>变为现在的RDD<Vector>。LabeledPoint现在是(double, Vector)的封装类而不再是 (double, double[])的封装类。从Array[Double]改变到Vector是直接明了的：

```
import org.apache.spark.mllib.linalg.Vector;

import org.apache.spark.mllib.linalg.Vectors;


double[] array = ... // a double array

Vector vector = Vectors.dense(array); // a dense vector
```

Vectors 提供了工厂方法来创建稀疏向量。

Python

我们过去使用NumPy array来代表一个含有类标签的特征向量，其中第一个实体是类标签，其他是特征。现在我们使用 LabeledPoint，其特征既可以是密集向量也可以是稀疏向量。

```
from pyspark.mllib.linalg import SparseVector

from pyspark.mllib.regression import LabeledPoint


# Create a labeled point with a positive label and a dense feature vector.

pos = LabeledPoint(1.0, [1.0, 0.0, 3.0])
```

```
# Create a labeled point with a negative label and a sparse feature vector.
neg = LabeledPoint(0.0, SparseVector(3, [0, 2], [1.0, 3.0]))
```

第 2 章　数据类型

- 本地向量（Local vector）
- 含类标签的点（Labeled point）
- 本地矩阵（Local matrix）
- 分布式矩阵（Distributed matrix）
  - 面向行矩阵（RowMatrix）
  - 行索引矩阵（IndexedRowMatrix）
  - 三元组矩阵（CoordinateMatrix）

MLlib 支持存储在单个机器中本地向量（local vectors）和本地矩阵（local matirces）以及由一个或多个 RDD 支撑实现的分布式矩阵。本地向量和本地聚类是对外公开接口中的的简单数据模型。底层线性代数操作通过 Breeze 和 jablas 来实现。在 MLlib 中，监督学习的一个训练实例被称之为“含有类标签的点（labeled point）”。

2.1.　本地向量

一个本地向量由基于 0 的整型索引数据和对应的 Double 型值数据组成，存储在某一个机器中。MLlib 支持两种类型的本地向量：密集的和稀疏的。一个密集向量的值通过一个 double array 来表示，而一个稀疏向量通过两个并列的数组 indices 和 values 来表示。比如，向量(1.0, 0.0, 3.0)可以以密集格式[1.0, 0.0, 3.0]或稀疏格式(3, [0, 2], [1.0, 3.0])表示，后者第一个 3 表示向量长度。

2.1.1 Scala

本地向量的基类是 Vector，我们提供了两个实现 DenseVector 和 SparseVector。我们建议通过 Vectors中实现的工厂方法来创建本地向量：

```
import org.apache.spark.mllib.linalg.{Vector, Vectors}
```

```scala
// 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)))
```

注意：Scala语言默认引入的是 scala.collection.immutable.Vector，为了使用MLlib的Vector，你必须显示引入org.apache.spark.mllib.linalg.Vector。

2.1.2 Java

本地向量的基类是 Vector，我们提供了两个实现 DenseVector 和 SparseVector。我们建议通过 Vectors中实现的工厂方法来创建本地向量：

```java
import org.apache.spark.mllib.linalg.Vector;
import org.apache.spark.mllib.linalg.Vectors;

// Create a dense vector (1.0, 0.0, 3.0).
Vector dv = 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.
Vector sv = Vectors.sparse(3, new int[] {0, 2}, new double[] {1.0, 3.0});
```

2.1.3 Python

MLlib 将下面类型识别为密集向量：

- Numpy 的 array
- Python 的 list，如[1, 2, 3]

将下面类型识别为稀疏向量：

- MLlib 的 SparseVector

- Pythony 的单列 csc_matrix（csc_matrix with a single column）

出于性能考虑，我们建议使用Numpy的array而不是Python的list，以及通过 Vectors中实现的工厂方法来创建稀疏向量。

```
import numpy as np

import scipy.sparse as sps

from pyspark.mllib.linalg import Vectors


# Use a NumPy array as a dense vector.

dv1 = np.array([1.0, 0.0, 3.0])

# Use a Python list as a dense vector.

dv2 = [1.0, 0.0, 3.0]

# Create a SparseVector.

sv1 = Vectors.sparse(3, [0, 2], [1.0, 3.0])

# Use a single-column SciPy csc_matrix as a sparse vector.

sv2 = sps.csc_matrix((np.array([1.0, 3.0]), np.array([0, 2]), np.array([0,
2])), shape = (3, 1))
```

2.2. 含类标签的点（Labeled point）

一个含类标签的点，由一个本地向量（或密集或稀疏）和一个类标签（label/response）组成。在 MLlib 中，监督学习算法会使用含类标签的点。我们使用一个 double 来存储一个类标签，所以我们可以在回归和分类中使用到 LabeledPoint。对二元分类来说，一个标签或为 0（负向）或为 1（正向）；对多元分类来源，标签应该是从 0 开始的类索引，如 0，1，2，……

2.2.1 Scala

含有类标签的点通过case class LabeledPoint来表示。

```
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)))
```

2.2.2 Java

含有类标签的点通过 LabeledPoint来表示。

```
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.
LabeledPoint pos = new LabeledPoint(1.0, Vectors.dense(1.0, 0.0, 3.0));

// Create a labeled point with a negative label and a sparse feature vector.
LabeledPoint neg = new LabeledPoint(1.0, Vectors.sparse(3, new int[] {0, 2}, n
ew double[] {1.0, 3.0}));
```

2.2.3 Python

含有类标签的点通过 LabeledPoint 来表示。

```
from pyspark.mllib.linalg import SparseVector
from pyspark.mllib.regression import LabeledPoint

# Create a labeled point with a positive label and a dense feature vector.
pos = LabeledPoint(1.0, [1.0, 0.0, 3.0])

# Create a labeled point with a negative label and a sparse feature vector.
neg = LabeledPoint(0.0, SparseVector(3, [0, 2], [1.0, 3.0]))
```

2.3. 稀疏数据（Sparse data）

实际运用中，稀疏数据是很常见的。MLlib可以读取以LIBSVM格式存储的训练实例，LIBSVM格式是 LIBSVM 和 LIBLINEAR的默认格式，这是一种文本格式，每行代表一个含类标签的稀疏特征向量。格式如下：

```
label index1:value1 index2:value2 ...
```

索引是从 1 开始并且递增。加载完成后，索引被转换为从 0 开始。

2.3.1 Scala

通过 MLUtils.loadLibSVMFile读取训练实例并以LIBSVM 格式存储。

```
import org.apache.spark.mllib.regression.LabeledPoint

import org.apache.spark.mllib.util.MLUtils

import org.apache.spark.rdd.RDD


val examples: RDD[LabeledPoint] = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_data.txt")
```

2.3.2 Java

通过 MLUtils.loadLibSVMFile读取训练实例并以LIBSVM 格式存储。

```
import org.apache.spark.mllib.regression.LabeledPoint;

import org.apache.spark.mllib.util.MLUtils;

import org.apache.spark.api.java.JavaRDD;


JavaRDD<LabeledPoint> examples =

  MLUtils.loadLibSVMFile(jsc.sc(), "data/mllib/sample_libsvm_data.txt").toJavaRDD();
```

2.3.3 Python

通过 MLUtils.loadLibSVMFile读取训练实例并以LIBSVM 格式存储。

```
from pyspark.mllib.util import MLUtils
```

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

2.4.本地矩阵

一个本地矩阵由整型的行列索引数据和对应的 double 型值数据组成，存储在某一个机器中。MLlib 支持密集矩阵（暂无稀疏矩阵！），实体值以列优先的方式存储在一个 double 数组中。比如下面的矩阵

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

其存储方式是一个一维数组 [1.0, 3.0, 5.0, 2.0, 4.0, 6.0] 和矩阵大小(3, 2)。

2.4.1 Scala

本地矩阵的基类是 Matrix，我们提供了一个实现 DenseMatrix。我们建议通过 Matrices 中实现的工厂方法来创建本地矩阵：

```
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))
```

2.4.2Java

本地矩阵的基类是 Matrix，我们提供了一个实现 DenseMatrix。我们建议通过 Matrices 中实现的工厂方法来创建本地矩阵：

```
import org.apache.spark.mllib.linalg.Matrix;

import org.apache.spark.mllib.linalg.Matrices;


// Create a dense matrix ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))
Matrix dm = Matrices.dense(3, 2, new double[] {1.0, 3.0, 5.0, 2.0, 4.0, 6.0});
```

2.5.分布式矩阵

一个分布式矩阵由 long 型行列索引数据和对应的 double 型值数据组成，分布式存储在一个或多个 RDD 中。对于巨大的分布式的矩阵来说，选择正确的存储格式非常重要。将一个分布式矩阵转换为另一个不同格式需要全局洗牌（shuffle），所以代价很高。目前，实现了三类分布式矩阵存储格式。

最基本的类型是 RowMatrix。一个 RowMatrix 是一个面向行的分布式矩阵，其行索引是没有具体含义的。比如一系列特征向量的一个集合。通过一个 RDD 来代表所有的行，每一行就是一个本地向量。对于 RowMatrix，我们假定其列数量并不巨大，所以一个本地向量可以恰当的与驱动节点（driver）交换信息，并且能够在某一节点中存储和操作。IndexedRowMatrix 与 RowMatrix 相似，但有行索引，可以用来识别行和进行 join 操作。而 CoordinateMatrix 是一个以三元组列表格式（coordinate list ，COO）存储的分布式矩阵，其实体集合是一个 RDD。

注意：因为我们需要缓存矩阵大小，分布式矩阵的底层 RDD 必须是确定的（deterministic）。通常来说，使用非确定的 RDD（non-deterministic RDDs）会导致错误。

2.5.1 面向行的分布式矩阵（RowMatrix）

一个 RowMatrix 是一个面向行的分布式矩阵，其行索引是没有具体含义的。比如一系列特征向量的一个集合。通过一个 RDD 来代表所有的行，每一行就是一个本地向量。既然每一行由一个本地向量表示，所以其列数就被整型数据大小所限制，其实实践中列数是一个很小的数值。

Scala

一个 RowMatrix可从一个RDD[Vector]实例创建。然后我们可以计算出其概要统计信息。

```
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()
```

Java

一个 RowMatrix可从一个JavaRDD<Vector>实例创建。然后我们可以计算出其概要统计信息。

```
import org.apache.spark.api.java.JavaRDD;

import org.apache.spark.mllib.linalg.Vector;

import org.apache.spark.mllib.linalg.distributed.RowMatrix;


JavaRDD<Vector> rows = ... // a JavaRDD of local vectors

// Create a RowMatrix from an JavaRDD<Vector>.

RowMatrix mat = new RowMatrix(rows.rdd());


// Get its size.

long m = mat.numRows();

long n = mat.numCols();
```

2.5.2 行索引矩阵（IndexedRowMatrix）

IndexedRowMatrix 与 RowMatrix 相似，但其行索引具有特定含义，本质上是一个含有索引信息的行数据集合（an RDD of indexed rows）。每一行由 long 型索引和一个本地向量组成。

Scala

一个 IndexedRowMatrix可从一个RDD[IndexedRow]实例创建，这里的 IndexedRow是(Long, Vector)的封装类。剔除IndexedRowMatrix中的行索引信息就变成一个RowMatrix。

```
import org.apache.spark.mllib.linalg.distributed.{IndexedRow, IndexedRowMatr
```

```
ix, 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()
```

Java

一个 IndexedRowMatrix 可从一个 JavaRDD<IndexedRow> 实例创建，这里的 IndexedRow是(long, Vector)的封装类。剔除IndexedRowMatrix中的行索引信息就变成一个RowMatrix。

```
import org.apache.spark.api.java.JavaRDD;

import org.apache.spark.mllib.linalg.distributed.IndexedRow;

import org.apache.spark.mllib.linalg.distributed.IndexedRowMatrix;

import org.apache.spark.mllib.linalg.distributed.RowMatrix;


JavaRDD<IndexedRow> rows = ... // a JavaRDD of indexed rows

// Create an IndexedRowMatrix from a JavaRDD<IndexedRow>.

IndexedRowMatrix mat = new IndexedRowMatrix(rows.rdd());


// Get its size.

long m = mat.numRows();

long n = mat.numCols();
```

```
// Drop its row indices.
RowMatrix rowMat = mat.toRowMatrix();
```

2.5.3 三元组矩阵（CoordinateMatrix）

一个 CoordinateMatrix 是一个分布式矩阵，其实体集合是一个 RDD。每一个实体是一个(i: Long, j: Long, value: Double)三元组，其中 i 代表行索引，j 代表列索引，value 代表实体的值。只有当矩阵的行和列都很巨大，并且矩阵很稀疏时才使用 CoordinateMatrix。

Scala

一个 CoordinateMatrix可从一个RDD[MatrixEntry]实例创建，这里的 MatrixEntry 是 (Long, Long, Double) 的封装类。通过调用 toIndexedRowMatrix 可以将一个 CoordinateMatrix转变为一个IndexedRowMatrix（但其行是稀疏的）。目前暂不支持其他计算操作。

```
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()
```

Java

一个 CoordinateMatrix 可从一个 JavaRDD<MatrixEntry> 实例创建，这里的 MatrixEntry是(long, long, double)的封装类。通过调用toIndexedRowMatrix可以将一个CoordinateMatrix转变为一个IndexedRowMatrix（但其行是稀疏的）。目前暂不支持其他计算操作。

```
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()
```

第 3 章 基本统计分析

- 汇总统计（Summary statistics）
- 关联（Correlations）
- 分层抽样（Stratified sampling）
- 假设检验（Hypothesis testing）
- 随机数据生成（Random data generation）

3.1 汇总统计

对RDD[Vector]格式数据的列汇总统计，我们提供Statistics中的colStats方法来实现。

3.1.1 Scala

colStats()方法返回一个 MultivariateStatisticalSummary实例，里面包括面向列的最大值、最小值、均值、方差、非零值个数以及总数量。

```
import org.apache.spark.mllib.linalg.Vector

import org.apache.spark.mllib.stat.{MultivariateStatisticalSummary, Statistics}


val observations: RDD[Vector] = ... // an RDD of Vectors


// Compute column summary statistics.

val summary: MultivariateStatisticalSummary = Statistics.colStats(observations)

println(summary.mean) // a dense vector containing the mean value for each column

println(summary.variance) // column-wise variance

println(summary.numNonzeros) // number of nonzeros in each column
```

3.1.2 Java

colStats()方法返回一个 MultivariateStatisticalSummary实例，里面包括面向列的最大值、最小值、均值、方差、非零值个数以及总数量。

```
import org.apache.spark.api.java.JavaRDD;

import org.apache.spark.api.java.JavaSparkContext;

import org.apache.spark.mllib.linalg.Vector;

import org.apache.spark.mllib.stat.MultivariateStatisticalSummary;

import org.apache.spark.mllib.stat.Statistics;


JavaSparkContext jsc = ...


JavaRDD<Vector> mat = ... // an RDD of Vectors
```

```
// Compute column summary statistics.
MultivariateStatisticalSummary summary = Statistics.colStats(mat.rdd());
System.out.println(summary.mean()); // a dense vector containing the mean val
ue for each column
System.out.println(summary.variance()); // column-wise variance
System.out.println(summary.numNonzeros()); // number of nonzeros in each colu
mn
```

3.1.3 Python

colStats()方法返回一个 MultivariateStatisticalSummary实例，里面包括面向列的最大值、最小值、均值、方差、非零值个数以及总数量。

```
from pyspark.mllib.stat import Statistics

sc = ... # SparkContext

mat = ... # an RDD of Vectors

# Compute column summary statistics.
summary = Statistics.colStats(mat)
print summary.mean()
print summary.variance()
print summary.numNonzeros()
```

3.2 相关性

在统计分析中，计算两个系列数据之间的相关性很常见。在 MLlib 中，我们提供了用于计算多系列数据之间两两关系的灵活性。目前支持的相关性算法是 Perarson 相关和 Spearsman 相关。

3.2.1Scala

Statistics提供了计算系列间相关性的方法。根据输入类型，两个RDD[Double]或 RDD[Vector]，输出将相应是一个Double或相关矩阵。

```
import org.apache.spark.SparkContext

import org.apache.spark.mllib.linalg._

import org.apache.spark.mllib.stat.Statistics


val sc: SparkContext = ...


val seriesX: RDD[Double] = ... // a series

val seriesY: RDD[Double] = ... // must have the same number of partitions and
cardinality as seriesX


// compute the correlation using Pearson's method. Enter "spearman" for Spearm
an's method. If a

// method is not specified, Pearson's method will be used by default.

val correlation: Double = Statistics.corr(seriesX, seriesY, "pearson")


val data: RDD[Vector] = ... // note that each Vector is a row and not a column


// calculate the correlation matrix using Pearson's method. Use "spearman" for
 Spearman's method.

// If a method is not specified, Pearson's method will be used by default.

val correlMatrix: Matrix = Statistics.corr(data, "pearson")
```

3.2.2 Java

Statistics提供了计算系列间相关性的方法。根据输入类型，两个 JavaDoubleRDDs或 JavaRDD<Vector>，输出将相应是一个Double或相关矩阵。

```
import org.apache.spark.api.java.JavaDoubleRDD;
```

```
import org.apache.spark.api.java.JavaSparkContext;

import org.apache.spark.mllib.linalg.*;

import org.apache.spark.mllib.stat.Statistics;


JavaSparkContext jsc = ...


JavaDoubleRDD seriesX = ... // a series

JavaDoubleRDD seriesY = ... // must have the same number of partitions and car
dinality as seriesX


// compute the correlation using Pearson's method. Enter "spearman" for Spearm
an's method. If a

// method is not specified, Pearson's method will be used by default.

Double correlation = Statistics.corr(seriesX.srdd(), seriesY.srdd(), "pearson
");


JavaRDD<Vector> data = ... // note that each Vector is a row and not a column


// calculate the correlation matrix using Pearson's method. Use "spearman" for
 Spearman's method.

// If a method is not specified, Pearson's method will be used by default.

Matrix correlMatrix = Statistics.corr(data.rdd(), "pearson");
```

3.2.2 Python

Statistics提供了计算系列间相关性的方法。根据输入类型，两个RDD[Double]或 RDD[Vector]，输出将相应是一个Double或相关矩阵。

```
from pyspark.mllib.stat import Statistics


sc = ... # SparkContext
```

```
seriesX = ... # a series

seriesY = ... # must have the same number of partitions and cardinality as seriesX


# Compute the correlation using Pearson's method. Enter "spearman" for Spearman's method. If a

# method is not specified, Pearson's method will be used by default.

print Statistics.corr(seriesX, seriesY, method="pearson")


data = ... # an RDD of Vectors

# calculate the correlation matrix using Pearson's method. Use "spearman" for Spearman's method.

# If a method is not specified, Pearson's method will be used by default.

print Statistics.corr(data, method="pearson")
```

3.3 分层抽样

在 MLlib 中，不同于其他统计方法，分层抽样方法如 sampleByKey 和 sampleByKeyExact，运行在键值对格式的 RDD 上。对分层抽样来说，keys 是一个标签，值是特定的属性。比如，key 可以是男人或女人、文档 ID，其相应的值可以是人口数据中的年龄列表或者文档中的词列表。sampleByKey 方法对每一个观测掷币决定是否抽中它，所以需要对数据进行一次遍历，也需要输入期望抽样的大小。而 sampleByKeyExact 方法并不是简单地在每一层中使用 sampleByKey 方法随机抽样，它需要更多资源，但将提供置信度高达 99.99%的精确抽样大小。在 Python 中，目前不支持 sampleByKeyExact。

3.3.1 Scala

`sampleByKeyExact()` 允许使用者准确抽取 $\lceil f_k \cdot n_k \rceil \forall k \in K$ 个元素，这里的 f_k是从键k中期望抽取的比例，n_k 是从键k中抽取的键值对数量，而K是键集合。为了确保凑样大小，无放回抽样对数据会多一次遍历；然而，有放回的抽样会多两次遍历。

```
import org.apache.spark.SparkContext

import org.apache.spark.SparkContext._

import org.apache.spark.rdd.PairRDDFunctions
```

```scala
val sc: SparkContext = ...

val data = ... // an RDD[(K, V)] of any key value pairs
val fractions: Map[K, Double] = ... // specify the exact fraction desired from
 each key

// Get an exact sample from each stratum
val approxSample = data.sampleByKey(withReplacement = false, fractions)
val exactSample = data.sampleByKeyExact(withReplacement = false, fractions)
```

3.3.2 Java

sampleByKeyExact() 允许使用者准确抽取 $\lceil f_k \cdot n_k \rceil \forall k \in K$ 个元素，这里的 f_k 是从键k中期望抽取的比例，n_k 是从键k中抽取的键值对数量，而K是键集合。为了确保凑样大小，无放回抽样对数据会多一次遍历；然而，有放回的抽样会多两次遍历。

```java
import java.util.Map;

import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaSparkContext;

JavaSparkContext jsc = ...

JavaPairRDD<K, V> data = ... // an RDD of any key value pairs
Map<K, Object> fractions = ... // specify the exact fraction desired from each
 key

// Get an exact sample from each stratum
JavaPairRDD<K, V> approxSample = data.sampleByKey(false, fractions);
JavaPairRDD<K, V> exactSample = data.sampleByKeyExact(false, fractions);
```

3.3.3 Python

sampleByKey() 允许使用者抽取 ⌈fk·nk⌉∀k∈K 个元素，这里的 fk是从键k中期望抽取的比例，nk 是从键k中抽取的键值对数量，而K是键集合。

注意：在 Python 中，目前不支持 sampleByKeyExact。

```
sc = ... # SparkContext


data = ... # an RDD of any key value pairs

fractions = ... # specify the exact fraction desired from each key as a dictio
nary


approxSample = data.sampleByKey(False, fractions);
```

3.4 假设检验

在统计分析中，假设检验是一个强大的工具，用来判断（1）结果是不是统计充分的；以及（2）结果是不是随机的。MLlib目前支持Pearson卡方校验（χ2）来测试适配度和独立性。输入数据类型决定了是否产生适配度或独立性。适配度校验需要Vector输入类型，而独立性测试需要一个Matrix矩阵输入。

MLlib 也支持 RDD[LabeledPoint]输入类型，然后使用卡方独立性测试来进行特征选择。

3.4.1 Scala

Statistics提供了进行Pearson卡方校验的方法。下面示例演示了怎样运行和解释假设校验。

```
import org.apache.spark.SparkContext

import org.apache.spark.mllib.linalg._

import org.apache.spark.mllib.regression.LabeledPoint

import org.apache.spark.mllib.stat.Statistics._


val sc: SparkContext = ...
```

```scala
val vec: Vector = ... // a vector composed of the frequencies of events


// compute the goodness of fit. If a second vector to test against is not supplied as a parameter,
// the test runs against a uniform distribution.
val goodnessOfFitTestResult = Statistics.chiSqTest(vec)
println(goodnessOfFitTestResult) // summary of the test including the p-value, degrees of freedom,
                                 // test statistic, the method used, and the null hypothesis.


val mat: Matrix = ... // a contingency matrix


// conduct Pearson's independence test on the input contingency matrix
val independenceTestResult = Statistics.chiSqTest(mat)
println(independenceTestResult) // summary of the test including the p-value, degrees of freedom...


val obs: RDD[LabeledPoint] = ... // (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.
val featureTestResults: Array[ChiSqTestResult] = Statistics.chiSqTest(obs)
var i = 1
featureTestResults.foreach { result =>
   println(s"Column $i:\n$result")
   i += 1
} // summary of the test
```

3.4.2 Java

Statistics提供了进行Pearson卡方校验的方法。下面示例演示了怎样运行和解释假设校验。

```
import org.apache.spark.api.java.JavaRDD;

import org.apache.spark.api.java.JavaSparkContext;

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;


JavaSparkContext jsc = ...


Vector vec = ... // a vector composed of the frequencies of events


// compute the goodness of fit. If a second vector to test against is not suppl
ied as a parameter,

// the test runs against a uniform distribution.

ChiSqTestResult goodnessOfFitTestResult = Statistics.chiSqTest(vec);

// summary of the test including the p-value, degrees of freedom, test statist
ic, the method used,

// and the null hypothesis.

System.out.println(goodnessOfFitTestResult);


Matrix mat = ... // a contingency matrix


// conduct Pearson's independence test on the input contingency matrix

ChiSqTestResult independenceTestResult = Statistics.chiSqTest(mat);

// summary of the test including the p-value, degrees of freedom...

System.out.println(independenceTestResult);
```

```
JavaRDD<LabeledPoint> obs = ... // an RDD of labeled points


// The contingency table is constructed from the raw (feature, label) pairs an
d used to conduct

// the independence test. Returns an array containing the ChiSquaredTestResult
 for every feature

// against the label.

ChiSqTestResult[] featureTestResults = Statistics.chiSqTest(obs.rdd());

int i = 1;

for (ChiSqTestResult result : featureTestResults) {

   System.out.println("Column " + i + ":");

   System.out.println(result); // summary of the test

   i++;

}
```

3.5 随机数据生成

随机数据生成对对随机的算法、原型和性能测试来说是有用的。MLlib 支持指定分布类型来生成随机 RDD，如均匀分布、标准正态分布和泊松分布。

3.5.1 Scala

RandomRDDs提供工厂方法来生成随机double RDDs或者vector RDDs。下面示例生成一个随机的double RDD，其值服从标准正态分布N(0,1)，然后将其映射为N(1,4)。

```
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 fr
om the

// standard normal distribution `N(0, 1)`, evenly distributed in 10 partition
```

```
s.

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)
```

3.5.2 Java

RandomRDDs提供工厂方法来生成随机double RDDs或者vector RDDs。下面示例生成一个随机的double RDD，其值服从标准正态分布N（0，1），然后将其映射为N（1，4）。

```
import org.apache.spark.SparkContext;

import org.apache.spark.api.JavaDoubleRDD;

import static org.apache.spark.mllib.random.RandomRDDs.*;


JavaSparkContext jsc = ...


// Generate a random double RDD that contains 1 million i.i.d. values drawn fr
om the

// standard normal distribution `N(0, 1)`, evenly distributed in 10 partition
s.

JavaDoubleRDD u = normalJavaRDD(jsc, 1000000L, 10);

// Apply a transform to get a random double RDD following `N(1, 4)`.

JavaDoubleRDD v = u.map(

  new Function<Double, Double>() {

    public Double call(Double x) {

      return 1.0 + 2.0 * x;

    }

  });
```

3.5.3 Python

RandomRDDs提供工厂方法来生成随机double RDDs或者vector RDDs。下面示例生成一个随机的double RDD，其值服从标准正态分布N(0,1)，然后将其映射为N(1,4)。

```
from pyspark.mllib.random import RandomRDDs


sc = ... # SparkContext


# Generate a random double RDD that contains 1 million i.i.d. values drawn fro
m the
# standard normal distribution `N(0, 1)`, evenly distributed in 10 partitions.
u = RandomRDDs.uniformRDD(sc, 1000000L, 10)
# Apply a transform to get a random double RDD following `N(1, 4)`.
v = u.map(lambda x: 1.0 + 2.0 * x)
```

第 4 章 分类和回归

MLlib 支持二元分类、多类分类、回归分析等多种算法。下表列出了问题类别及其相关算法：

问题类别	支持的方法
二元分类	线性支持向量机，逻辑回归，决策树，朴素贝叶斯
多类分类	决策树，朴素贝叶斯
回归	线性最小二乘法，Lasso，岭回归（ridge regression），决策树

这些方法的更多细节请参见下面内容：

- 线性模型
  - 二元分类（支持向量机，逻辑回归）
  - 线性回归（最小二乘法，Lasso，ridge）
- 决策树
- 朴素贝叶斯

4.1. 线性模型

- 数学公式
  - 损失函数
  - 正则化因子（Regularizers）
- 二元分类
  - 线性支持向量机
  - 逻辑回归
  - 评估度量
  - 示例
- 线性最小二乘法，Lasso，岭回归（ridge regression）
  - 示例
- 流的线性回归（Streaming linear regression）
  - 示例
- 实现（开发者）

4.1.1. 数学公式

许多标准机器学习方法可以被转换为凸优化问题，即，一个为凸函数 f 找到最小值的任务，这个函数 f 依赖于一个有 d 个值的向量变量 w（代码中的 weights）。更正式点，这是一个$\min_{\mathbf{w}\in\mathbb{R}^d} \; f(\mathbf{w})$优化问题，其目标函数 f 具有下面形式：

$$f(\mathbf{w}) := \lambda\, R(\mathbf{w}) + \frac{1}{n} \sum_{i=1}^{n} L(\mathbf{w};\mathbf{x}_i,y_i)$$

向量 xi∈R^d是训练数据样本，其中 1≤i≤n。 yi∈R 是相应的类标签，也是我们想要预测的目标。如果 L(w;x,y)能被表述为 w^Tx和 y的一个函数，我们称该方法是线性的。有几个MLlib分类和回归算法属于该范畴，我们在此一一讨论。

目标函数f包括两部分：控制模型复杂度的正则化因子和度量模型误差的损失函数。损失函数 L(w;.)是典型关于w的凸函数。事先锁定的正则化参数 λ≥0（代码中的regParam）承载了我们在最小化损失量（训练误差）和最小化模型复杂度（避免过渡拟合）两个目标之间的权衡取舍。

损失函数

下表概述了 MLlib 支持的损失函数及其梯度和子梯度：

	损失函数 $L(w;x,y)$	梯度或子梯度
hinge loss	$\max\{0,1-yw^Tx\},y\in\{-1,+1\}$	$\begin{cases} -y\cdot\mathbf{x} & \text{if } y\mathbf{w}^T\mathbf{x}<1, \\ 0 & \text{otherwise.} \end{cases}$
logistic loss	$\log(1+\exp(-yw^Tx)),y\in\{-1,+1\}$	$-y\left(1-\frac{1}{1+\exp(-y\mathbf{w}^T\mathbf{x})}\right)\cdot\mathbf{x}$
squared loss	$\frac{1}{2}\left(\mathbf{w}^T\mathbf{x}-y\right)^2, \quad y\in\mathbb{R}$	$(w^Tx-y)\cdot x$

正则化因子

正则化因子的目标是获得简单模型和避免过度拟合。在 MLlib 中，支持下面正则化因子：

	正则化因子 **R(w)**	梯度或子梯度
零（未正则化）	0	0
L2 范数	$\frac{1}{2}\left\|\mathbf{w}\right\|_2^2$	w
L1 范数	$\left\|\mathbf{w}\right\|_1$	sign(w)

在这里，sign(w)是一个代表w中所有实体的类标签（signs (±1)）的向量。

与 L1 正则化问题比较，由于 L2 的平滑性，L2 正则化问题一般较容易解决。但是，由于可以强化权重的稀疏性，L1 正则化更能产生较小的和更容易解释的模型，而后者在特征选择是非常有用的。不正则化而去训练模型是不恰当的，尤其是在训练样本数量较小的时候。

4.1.2. 二元分类

二元分类将数据项划分为两类：正例和反例。MLlib 支持两种二元分类的线性方法：线性支持向量机和逻辑回归。对两种方法来说，MLlib 都支持 L1、L2 正则化。在 MLlib 中，训练数据集用一个 LabeledPoint 格式的 RDD 来表示。需要注意，本指南中的数学公司里，

约定训练标签 y 为+1（正例）或-1（反例）。但是在 MLlib 中，为了与多类标签保持一致，反例标签是 0，而不是-1。

线性支持向量机（SVMs）

对于大规模的分类任务来说，线性支持向量机是标准方法。它是上面“数学公式”一节中所描述的线性方法，其损失函数是 hinge loss：

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

默认配置下，线性 SVM 使用 L2 正则化训练。我们也支持 L1 正则化。通过这种方式，问题变为线性规划问题。

线性支持向量机算法的产出是一个SVM模型。给定新数据点X，该模型基于 w^Tx的值来预测。默认情形下，$w^Tx\geq 0$时为正例，否则为反例。

逻辑回归

逻辑回归广泛运用于二元因变量预测。它是上面“数学公式”一节中所描述的线性方法，其损失函数是logistic loss：

$$L(w;x,y):=\log(1+\exp(-yw^Tx)).$$

逻辑回归算法的产出是一个逻辑回归模型。给定新数据点 X，该模型运用下面逻辑函数来预测：

$$f(z)=\frac{1}{1+e^{-z}}$$

在这里，$z=w^Tx$。默认情况下，若 $f(w^Tx)>0.5$，输出是正例，否则是反例。与线性支持向量机不同之处在于，线性回归模型 f(z)的输出含有一个概率解释（即，x是正例的概率）。

评估度量

MLlib 支持常用的二元分类评估度量方法(在 PySpark 中不可用)。包括精度、召回率、F 度量、接收者特征操作曲线、精度-召回率曲线和 AUC。AUC 常用来比较不同模型的性能，精度/召回率/F 度量用来决定阈值时为预测指定恰当阈值。

示例

Scala

下面代码片段演示了如何加载数据集、运用算法对象的静态方法执行训练算法、以及运用模型预测来计算训练误差。

```scala
import org.apache.spark.SparkContext
import org.apache.spark.mllib.classification.SVMWithSGD
import org.apache.spark.mllib.evaluation.BinaryClassificationMetrics
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.linalg.Vectors
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)
```

默认配置下，SVMWithSGD.train()将正则化参数设置为 1.0 来进行 L2 正则化。如果我们想配置算法参数，我们可以直接生成一个 SVMWithSGD 对象然后调用 setter 方法。所有其他 MLlib 算法都支持这种自定义化方法。举例来说，下面代码生了一个用于 SVM 的 L1 正则化变量，其正则化参数为 0.1，且迭代次数为 200。

```
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)
```

LogisticRegressionWithSGD的使用方法与SVMWithSGD相似。

Java

所有 MLlib 方法都使用 Java 友好的类型，所以您可以像 scala 中那样导入和调用。唯一要说明的是那些使用 Scala RDD 对象的方法，因为在 spark 的 java API 中使用的是 JavaRDD 类。对 JavaRDD 对象，您能够通过调用.rdd()方法将其转换为对应的 Scala 对象。与 Scala 示例等效的应用示例如下：

```
import java.util.Random;



import scala.Tuple2;



import org.apache.spark.api.java.*;

import org.apache.spark.api.java.function.Function;

import org.apache.spark.mllib.classification.*;

import org.apache.spark.mllib.evaluation.BinaryClassificationMetrics;

import org.apache.spark.mllib.linalg.Vector;
```

```
import org.apache.spark.mllib.regression.LabeledPoint;
import org.apache.spark.mllib.util.MLUtils;
import org.apache.spark.SparkConf;
import org.apache.spark.SparkContext;

public class SVMClassifier {
  public static void main(String[] args) {
    SparkConf conf = new SparkConf().setAppName("SVM Classifier Example");
    SparkContext sc = new SparkContext(conf);
    String path = "data/mllib/sample_libsvm_data.txt";
    JavaRDD<LabeledPoint> data = MLUtils.loadLibSVMFile(sc, path).toJavaRDD
();

    // Split initial RDD into two... [60% training data, 40% testing data].
    JavaRDD<LabeledPoint> training = data.sample(false, 0.6, 11L);
    training.cache();
    JavaRDD<LabeledPoint> test = data.subtract(training);

    // Run training algorithm to build the model.
    int numIterations = 100;
    final SVMModel model = SVMWithSGD.train(training.rdd(), numIterations);

    // Clear the default threshold.
    model.clearThreshold();

    // Compute raw scores on the test set.
    JavaRDD<Tuple2<Object, Object>> scoreAndLabels = test.map(
      new Function<LabeledPoint, Tuple2<Object, Object>>() {
        public Tuple2<Object, Object> call(LabeledPoint p) {
          Double score = model.predict(p.features());
```

```
        return new Tuple2<Object, Object>(score, p.label());
      }
    }
  );

  // Get evaluation metrics.
  BinaryClassificationMetrics metrics =
    new BinaryClassificationMetrics(JavaRDD.toRDD(scoreAndLabels));
  double auROC = metrics.areaUnderROC();

  System.out.println("Area under ROC = " + auROC);
 }
}
```

默认配置下，SVMWithSGD.train()将正则化参数设置为 1.0 来进行 L2 正则化。如果我们想配置算法参数，我们可以直接生成一个 SVMWithSGD 对象然后调用 setter 方法。所有其他 MLlib 算法都支持这种客户化方法。举例来说，下面代码生了一个用于 SVM 的 L1 正则化变量，其正则化参数为 0.1，且迭代次数为 200。

```
import org.apache.spark.mllib.optimization.L1Updater;

SVMWithSGD svmAlg = new SVMWithSGD();
svmAlg.optimizer()
  .setNumIterations(200)
  .setRegParam(0.1)
  .setUpdater(new L1Updater());
final SVMModel modelL1 = svmAlg.run(training.rdd());
```

为了运行上面的独立程序，请参考 Spark quick-start 指引中的 Standalone Applications 章节。务必将 spark-mllib 作为编译依赖库。

Python

下面代码片段演示了如何加载数据集、运用算法对象的静态方法执行训练算法、以及运用模型预测来计算训练误差。

```
from pyspark.mllib.classification import LogisticRegressionWithSGD
from pyspark.mllib.regression import LabeledPoint
from numpy import array


# Load and parse the data
def parsePoint(line):
    values = [float(x) for x in line.split(' ')]
    return LabeledPoint(values[0], values[1:])


data = sc.textFile("data/mllib/sample_svm_data.txt")
parsedData = data.map(parsePoint)


# Build the model
model = LogisticRegressionWithSGD.train(parsedData)


# Evaluating the model on training data
labelsAndPreds = parsedData.map(lambda p: (p.label, model.predict(p.feature
s)))
trainErr = labelsAndPreds.filter(lambda (v, p): v != p).count() / float(parsed
Data.count())
print("Training Error = " + str(trainErr))
```

4.1.3. 线性最小二乘法，Lasso，岭回归（ridge regression）

线性最小二乘法是回归问题中最常用的的公式。它是上面“数学公式”一节中所描述的线性方法，其损失函数是平方损失（squared loss）：

$$L(\mathbf{w}; \mathbf{x}, y) := \frac{1}{2} (\mathbf{w}^T \mathbf{x} - y)^2$$

根据正则化参数类型的不同，将相关算法分为不同回归算法：

- 普通最小二乘法或线性最小二乘法：未正则化

- 岭回归算法：使用 L2 正则化
- Lasso 算法：使用 L1 正则化

所有相关模型，其平均损失或训练误差计算公式为 $\frac{1}{n}\sum_{i=1}^{n}(\mathbf{w}^T x_i - y_i)^2$，即著名的均方误差（mean squared error）。

示例

Scala

下面示例演示了如何加载训练数据、将其解析为一个LabeledPoint 格式的RDD。然后使用LinearRegressionWithSGD 建立一个用于预测类标签的模型。最后我们计算均方误差来评估拟合度。

```
import org.apache.spark.mllib.regression.LinearRegressionWithSGD

import org.apache.spark.mllib.regression.LabeledPoint

import org.apache.spark.mllib.linalg.Vectors


// Load and parse the data

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(_.toD
ouble)))

}


// Building the model

val numIterations = 100

val model = LinearRegressionWithSGD.train(parsedData, numIterations)


// Evaluate model on training examples and compute training error

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)
```

RidgeRegressionWithSGD 和 LassoWithSGD 的使用方法与 LinearRegressionWithSGD 相似。

Java

所有 MLlib 方法都使用 Java 友好的类型，所以您可以像 scala 中那样导入和调用。唯一要说明的是那些使用 Scala RDD 对象的方法，因为在 spark 的 java API 中使用的是 JavaRDD 类。对 JavaRDD 对象 您能够通过调用.rdd()方法将其转换为对应的 Scala 对象。与 Scala 示例等效的 Java 示例如下：

```
import scala.Tuple2;

import org.apache.spark.api.java.*;
import org.apache.spark.api.java.function.Function;
import org.apache.spark.mllib.linalg.Vector;
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;
import org.apache.spark.SparkConf;

public class LinearRegression {
  public static void main(String[] args) {
    SparkConf conf = new SparkConf().setAppName("Linear Regression Example");
    JavaSparkContext sc = new JavaSparkContext(conf);

    // Load and parse the data
    String path = "data/mllib/ridge-data/lpsa.data";
```

```
    JavaRDD<String> data = sc.textFile(path);
    JavaRDD<LabeledPoint> parsedData = data.map(
      new Function<String, LabeledPoint>() {
        public LabeledPoint call(String line) {
          String[] parts = line.split(",");
          String[] features = parts[1].split(" ");
          double[] v = new double[features.length];
          for (int i = 0; i < features.length - 1; i++)
            v[i] = Double.parseDouble(features[i]);
          return new LabeledPoint(Double.parseDouble(parts[0]), Vectors.dense
(v));
        }
      }
    );

    // Building the model
    int numIterations = 100;
    final LinearRegressionModel model =
      LinearRegressionWithSGD.train(JavaRDD.toRDD(parsedData), numIteration
s);

    // Evaluate model on training examples and compute training error
    JavaRDD<Tuple2<Double, Double>> valuesAndPreds = parsedData.map(
      new Function<LabeledPoint, Tuple2<Double, Double>>() {
        public Tuple2<Double, Double> call(LabeledPoint point) {
          double prediction = model.predict(point.features());
          return new Tuple2<Double, Double>(prediction, point.label());
        }
      }
    );
```

```
    JavaRDD<Object> MSE = new JavaDoubleRDD(valuesAndPreds.map(
      new Function<Tuple2<Double, Double>, Object>() {
        public Object call(Tuple2<Double, Double> pair) {
          return Math.pow(pair._1() - pair._2(), 2.0);
        }
      }
    ).rdd()).mean();
    System.out.println("training Mean Squared Error = " + MSE);
  }
}
```

为了运行上面的独立程序，请参考 Spark quick-start 指引中的 Standalone Applications 章节。务必将 spark-mllib 作为编译依赖库。

Python

下面示例演示了如何加载训练数据、将其解析为一个 LabeledPoint 格式的 RDD。然后使用 LinearRegressionWithSGD 建立一个用于预测类标签的模型。最后我们计算均方误差来评估拟合度。

```
from pyspark.mllib.regression import LabeledPoint, LinearRegressionWithSGD
from numpy import array

# Load and parse the data
def parsePoint(line):
    values = [float(x) for x in line.replace(',', ' ').split(' ')]
    return LabeledPoint(values[0], values[1:])

data = sc.textFile("data/mllib/ridge-data/lpsa.data")
parsedData = data.map(parsePoint)

# Build the model
model = LinearRegressionWithSGD.train(parsedData)
```

```
# Evaluate the model on training data
valuesAndPreds = parsedData.map(lambda p: (p.label, model.predict(p.features)))
MSE = valuesAndPreds.map(lambda (v, p): (v - p)**2).reduce(lambda x, y: x + y) / valuesAndPreds.count()
print("Mean Squared Error = " + str(MSE))
```

4.1.4. 流的线性回归

当数据以流的形式传入，当收到新数据时更新模型参数，在线拟合回归模型是有用的。MLlib 目前使用普通最小二乘法实现流的线性回归。这种拟合的处理机制与离线方法相似，但其拟合发生于每一数据块到达时之外，目的是为了持续更新以反应流中数据。

示例

下面示例演示了如何从两个文本流中加载训练数据和验证数据、将其解析为 LabeledPoint 流、基于第一个流在线拟合线性回归模型、然后在第二个流上进行预测。

Scala

首先，我们导入用来解析输入数据和创建模型的类。

```
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.regression.StreamingLinearRegressionWithSGD
```

然后创建训练集和测试集的输入流。我们假定一个StreamingContext ssc已经被创建，请参见 Spark Streaming Programming Guide 获取更多信息。对这个例子来说，我们在流中使用含类标签的点，但在实际引用中，您更可能使用不含有类标签的数据作为测试集。

```
val trainingData = ssc.textFileStream('/training/data/dir').map(LabeledPoint.parse)
val testData = ssc.textFileStream('/testing/data/dir').map(LabeledPoint.parse)
```

我们将权重初始化为 0 来创建模型

```
val numFeatures = 3
```

```
val model = new StreamingLinearRegressionWithSGD()

    .setInitialWeights(Vectors.zeros(numFeatures))
```

接下来我们注册用于训练和测试的流并开始任务。并打印其正确的类标签结果来观察结果

```
model.trainOn(trainingData)

model.predictOnValues(testData.map(lp => (lp.label, lp.features))).print()



ssc.start()

ssc.awaitTermination()
```

我们现在可以在训练目录和测试目录中存入文本数据来模拟流事件。每一行数据应该是一个(y,[x1,x2,x3])格式的数据点，其中 y 是类标签，而 x1,x2,x3 是特征。每当一个文本文件放入/training/data/dir 目录时模型会更新。每当一个文本文件放入到/testing/data/dir 目录时您将观察到预测结果。在训练目录中放入越多数据，预测越好。

4.1.5. 实现（开发者）

在具体场景之外，MLlib 还实现了随机梯度下降(stochastic gradient descent (SGD)) 的一个简单分布式版本，该实现基于底层的梯度下降功能单元（请参见“优化”章节）。所有提供的算法接收一个正则化参数(regParam)和不同的随机梯度下降相关参数(stepSize, numIterations, miniBatchFraction)作为输入。对每一个算法来说，我们都支持三种可能正则化方法（不正则化，L1 和 L2）。

Scala 中实现了下面算法：

1) SVMWithSGD
2) LogisticRegressionWithSGD
3) LinearRegressionWithSGD
4) RidgeRegressionWithSGD
5) LassoWithSGD

Python 通过 PythonMLLibAPI 来调用 Scala 实现。

4. 2. 决策树

- 基本算法
  - 节点不纯度和信息增益（Node impurity and information gain）
  - 切分备选方案（Split candidates）

- 停止规则
- 实现细节
  - 最大内存需求
  - 特征值装箱（Binning feature values）
  - 规模自适应（Scaling）
- 示例
  - 分类
  - 回归

决策树及其家族是解决分类和回归机器学习任务的热点算法。决策树方法易于解释和处理分类特征、能够扩展处理多类分类、不需要特征变换以及能够捕获非线性特征关系，使得决策树得到广泛应用。该体系中的算法如随机森林、更新等都是分类和回归任务中的高性能算法。

MLlib 中的决策树支持二元和多类分类，支持连续特征和分类特征的回归。其实现将数据按行分区，可以分布式训练数百万样本。

4.2.1. 基本算法

决策树是一种将特征空间迭代地进行二元切分的贪心算法。树为每个最底层分区（叶子分区）分配一个类标签。每一次切分都贪心的从可能切分集合中选择一个最佳切分，目的是从树节点中获得最大信息增益。换言之，在每一节点的切分都是基于

$$\underset{s}{\operatorname{argmax}}\, IG(D, s)$$

原则选择，在这里，IG(D,s)是每次切分产生的信息增益。

节点不纯度和信息增益

节点不存度用于度量节点中类标签的同质性。当前实现为分类提供了两种不纯度度量（Gini 不纯度和熵），为回归提供了一种不纯度度量（方差不纯度）。

不纯度	任务	公式	描述
Gini 不纯度	分类	$\sum_{i=1}^{M} f_i(1-f_i)$	f_i是类i在某节点中出现频率，M是不重复类标签的数量。
熵	分类	$\sum_{i=1}^{M} -f_i log(f_i)$	f_i是类i在某节点中出现频率，M是不重复类标签的数量。

方差	回归	$\frac{1}{n}\sum_{i=1}^{N}(x_i-\mu)^2$	x_i 是某一样本的类标签，N是样本个数，μ 是通过 $\frac{1}{N}\sum_{i=1}^{n}x_i$ 计算得到的均值。

信息增益是父节点不纯度与两个子节点不纯度加权和的差值。假定某次切分s将一个大小为N的数据集D切分为两个子集 *Dleft*和 *Dright*。左集大小为 *Nleft*，右集大小为 *Nright*，相应的信息增益可用下面公式表达：

$$IG(D,s)=Impurity(D)-\frac{N_{left}}{N}Impurity(D_{left})-\frac{N_{right}}{N}Impurity(D_{right})$$

切分备选方案

连续特征

对小数据集的单机实现来说，每一连续特征的切分备选方案通常是一系列唯一值（the split candidates for each continuous feature are typically the unique values for the feature）。一些实现先对特征值排序，然后使用排序后的唯一值作为切分备选方案以更快的执行树计算（Some implementations sort the feature values and then use the ordered unique values as split candidates for faster tree calculations）。

对于巨大的分布式数据集来说，特征值排序代价很高。实际实现是在数据抽样上执行分位数计算，来获得一个近似的切分备选方案集。排序的切分创建箱（“bins”），装箱的最大数量通过 maxBins 参数指定。

请注意，装箱数量不能大于样本数量 N（由于 maxBins 的默认值是 100，所以实际上很少会出现）。如果条件不满足，决策树自动的减少装箱数量。

分类特征

对一个具有M个可能值（分类标签）的分类特征来说，切分方法是从 $2^{M-1}-1$种备选切分方案中选择一种。对二元分类和回归，我们可以根据类标签调和均值(the average label)将特征值排序，如此可以将备选分类方案的数量降低到到M-1 个（请参见《Elements of Statistical Machine Learning》一书的 9.2.4 章节）。比如，对某个具有分类值A/B/C的分类特征，A/B/C在标签 1 上的比例依次是 0.2/0.6/0.4，其分类值排序是A、C、B。对二元分类来说，可能切分备选方案是“A | C, B”和“A , C | B”，其中竖线是分隔符。

对于多类分类，只要可能，所有从 $2^{M-1}-1$种备选切分方案都使用。当大从 $2^{M-1}-1$大

于maxBins参数时，我们使用一种类似于二元分类和回归的启发式方法。这M个特征值根据不纯度排序，然后从M-1 个切分备选方案选择。

停止规则

当下面两种条件之一满足时，停止递归树构造：

1. 节点深度已经等于训练参数 maxDepth
2. 在节点上已经没有切分备选方案能够获得信息增益。

4.2.2. 实现细节

最大内存需求

为了快速处理，决策树算法对树同一等级上的所有节点同时执行直方图计算（performs simultaneous histogram computations）。于是，随着树越来越深度，对内存的需求也越大，可能导致内存溢出。为了缓解该问题，训练参数 maxMemoryInMB（在 master 上翻倍）限定了工作节点上的用于直方图计算的最大内存。保守地将默认值设置为 128MB，在大多数场景下，决策算法可以正常工作。一旦某一等级上所需内存超过 maxMemoryInMB 阈值，在该等级之下的所有训练任务都被拆分为较小的多个任务。

请注意，如果您有更大的内存，增加 maxMemoryInMB 能减少数据转移从而加快训练进度。

特征值装箱

增大 maxBins 使得算法考虑更多的切分备选方案，作更细粒度的决策。但是，它也增加了计算量和通讯量。

请注意，对任意分类特征来说，maxBins 参数的最小也必须是最大标签数 M。

规模自适应(Scaling)

计算量随着训练样本数量、特征数量、maxBins 参数值近似地线性增长。通讯量随着特征数量和 maxBins 参数值近似地线性增长。

算法实现可以接收稀疏数据和密集数据。但是，并未针对稀疏数据进行优化。

4.2.3. 示例

分类

下面示例演示了如何加载 LIBSVM 格式的数据 将其解析为 LabeledPoint 格式的 RDD，然后基于 Gini 不纯度使用决策树进行分类，在这里，最大树深度为 5。最后，计算训练误差用来评估算法精度。

Scala

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

// Load and parse the data file.
// Cache the data since we will use it again to compute training error.
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_data.txt").ca
che()

// Train a DecisionTree model.
//  Empty categoricalFeaturesInfo indicates all features are continuous.
val numClasses = 2
val categoricalFeaturesInfo = Map[Int, Int]()
val impurity = "gini"
val maxDepth = 5
val maxBins = 100

val model = DecisionTree.trainClassifier(data, numClasses, categoricalFeature
sInfo, impurity,
  maxDepth, maxBins)

// Evaluate model on training instances and compute training error
val labelAndPreds = data.map { point =>
```

```
  val prediction = model.predict(point.features)
  (point.label, prediction)
}
val trainErr = labelAndPreds.filter(r => r._1 != r._2).count.toDouble / data.c
ount
println("Training Error = " + trainErr)
println("Learned classification tree model:\n" + model)
```

Java

```
import java.util.HashMap;
import scala.Tuple2;
import org.apache.spark.api.java.function.Function2;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.Function;
import org.apache.spark.api.java.function.PairFunction;
import org.apache.spark.mllib.regression.LabeledPoint;
import org.apache.spark.mllib.tree.DecisionTree;
import org.apache.spark.mllib.tree.model.DecisionTreeModel;
import org.apache.spark.mllib.util.MLUtils;
import org.apache.spark.SparkConf;

SparkConf sparkConf = new SparkConf().setAppName("JavaDecisionTree");
JavaSparkContext sc = new JavaSparkContext(sparkConf);

// Load and parse the data file.
// Cache the data since we will use it again to compute training error.
String datapath = "data/mllib/sample_libsvm_data.txt";
JavaRDD<LabeledPoint> data = MLUtils.loadLibSVMFile(sc.sc(), datapath).toJava
```

```
RDD().cache();


// Set parameters.
//  Empty categoricalFeaturesInfo indicates all features are continuous.
Integer numClasses = 2;
HashMap<Integer, Integer> categoricalFeaturesInfo = new HashMap<Integer, Integer>();
String impurity = "gini";
Integer maxDepth = 5;
Integer maxBins = 100;


// Train a DecisionTree model for classification.
final DecisionTreeModel model = DecisionTree.trainClassifier(data, numClasses,
  categoricalFeaturesInfo, impurity, maxDepth, maxBins);


// Evaluate model on training instances and compute training error
JavaPairRDD<Double, Double> predictionAndLabel =
  data.mapToPair(new PairFunction<LabeledPoint, Double, Double>() {
    @Override public Tuple2<Double, Double> call(LabeledPoint p) {
      return new Tuple2<Double, Double>(model.predict(p.features()), p.label());
    }
  });
Double trainErr =
  1.0 * predictionAndLabel.filter(new Function<Tuple2<Double, Double>, Boolean>() {
    @Override public Boolean call(Tuple2<Double, Double> pl) {
      return !pl._1().equals(pl._2());
    }
  }).count() / data.count();
```

```
System.out.println("Training error: " + trainErr);

System.out.println("Learned classification tree model:\n" + model);
```

Python

```
from pyspark.mllib.regression import LabeledPoint

from pyspark.mllib.tree import DecisionTree

from pyspark.mllib.util import MLUtils


# Load and parse the data file into an RDD of LabeledPoint.

# Cache the data since we will use it again to compute training error.

data = MLUtils.loadLibSVMFile(sc, 'data/mllib/sample_libsvm_data.txt').cache
()


# Train a DecisionTree model.

#  Empty categoricalFeaturesInfo indicates all features are continuous.

model = DecisionTree.trainClassifier(data, numClasses=2, categoricalFeaturesI
nfo={},

                                     impurity='gini', maxDepth=5, maxBins=100)


# Evaluate model on training instances and compute training error

predictions = model.predict(data.map(lambda x: x.features))

labelsAndPredictions = data.map(lambda lp: lp.label).zip(predictions)

trainErr = labelsAndPredictions.filter(lambda (v, p): v != p).count() / float
(data.count())

print('Training Error = ' + str(trainErr))

print('Learned classification tree model:')

print(model)
```

注意：当对一个数据集作预测时，批量预测比在每个数据点上分别调用方法`predict`分别预测的效率要高很多。原因在于 Python 代码在底层调用的是 Scala 的决策树模型。

回归

下面示例演示了如何加载 LIBSVM 格式的数据 将其解析为 LabeledPoint 格式的 RDD，然后基于方差不纯度使用决策树进行回归，在这里，最大树深度为 5。最后，计算均方误差用来评估拟合度。

Scala

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

// Load and parse the data file.
// Cache the data since we will use it again to compute training error.
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_data.txt").cache()

// Train a DecisionTree model.
//  Empty categoricalFeaturesInfo indicates all features are continuous.
val categoricalFeaturesInfo = Map[Int, Int]()
val impurity = "variance"
val maxDepth = 5
val maxBins = 100

val model = DecisionTree.trainRegressor(data, categoricalFeaturesInfo, impurity,
  maxDepth, maxBins)

// Evaluate model on training instances and compute training error
val labelsAndPredictions = data.map { point =>
  val prediction = model.predict(point.features)
  (point.label, prediction)
}
```

```
val trainMSE = labelsAndPredictions.map{ case(v, p) => math.pow((v - p), 2)}.mean()

println("Training Mean Squared Error = " + trainMSE)

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

Java

```
import java.util.HashMap;

import scala.Tuple2;

import org.apache.spark.api.java.function.Function2;

import org.apache.spark.api.java.JavaPairRDD;

import org.apache.spark.api.java.JavaRDD;

import org.apache.spark.api.java.JavaSparkContext;

import org.apache.spark.api.java.function.Function;

import org.apache.spark.api.java.function.PairFunction;

import org.apache.spark.mllib.regression.LabeledPoint;

import org.apache.spark.mllib.tree.DecisionTree;

import org.apache.spark.mllib.tree.model.DecisionTreeModel;

import org.apache.spark.mllib.util.MLUtils;

import org.apache.spark.SparkConf;


// Load and parse the data file.

// Cache the data since we will use it again to compute training error.

String datapath = "data/mllib/sample_libsvm_data.txt";

JavaRDD<LabeledPoint> data = MLUtils.loadLibSVMFile(sc.sc(), datapath).toJavaRDD().cache();


SparkConf sparkConf = new SparkConf().setAppName("JavaDecisionTree");

JavaSparkContext sc = new JavaSparkContext(sparkConf);


// Set parameters.
```

```
// Empty categoricalFeaturesInfo indicates all features are continuous.
HashMap<Integer, Integer> categoricalFeaturesInfo = new HashMap<Integer, Integer>();
String impurity = "variance";
Integer maxDepth = 5;
Integer maxBins = 100;

// Train a DecisionTree model.
final DecisionTreeModel model = DecisionTree.trainRegressor(data,
  categoricalFeaturesInfo, impurity, maxDepth, maxBins);

// Evaluate model on training instances and compute training error
JavaPairRDD<Double, Double> predictionAndLabel =
  data.mapToPair(new PairFunction<LabeledPoint, Double, Double>() {
    @Override public Tuple2<Double, Double> call(LabeledPoint p) {
      return new Tuple2<Double, Double>(model.predict(p.features()), p.label());
    }
  });
Double trainMSE =
  predictionAndLabel.map(new Function<Tuple2<Double, Double>, Double>() {
    @Override public Double call(Tuple2<Double, Double> pl) {
      Double diff = pl._1() - pl._2();
      return diff * diff;
    }
  }).reduce(new Function2<Double, Double, Double>() {
    @Override public Double call(Double a, Double b) {
      return a + b;
    }
  }) / data.count();
```

```
System.out.println("Training Mean Squared Error: " + trainMSE);

System.out.println("Learned regression tree model:\n" + model);
```

Python

```
from pyspark.mllib.regression import LabeledPoint

from pyspark.mllib.tree import DecisionTree

from pyspark.mllib.util import MLUtils


# Load and parse the data file into an RDD of LabeledPoint.

# Cache the data since we will use it again to compute training error.

data = MLUtils.loadLibSVMFile(sc, 'data/mllib/sample_libsvm_data.txt').cache
()


# Train a DecisionTree model.

#  Empty categoricalFeaturesInfo indicates all features are continuous.

model = DecisionTree.trainRegressor(data, categoricalFeaturesInfo={},

                                    impurity='variance', maxDepth=5, maxBins=100)


# Evaluate model on training instances and compute training error

predictions = model.predict(data.map(lambda x: x.features))

labelsAndPredictions = data.map(lambda lp: lp.label).zip(predictions)

trainMSE = labelsAndPredictions.map(lambda (v, p): (v - p) * (v - p)).sum() /
float(data.count())

print('Training Mean Squared Error = ' + str(trainMSE))

print('Learned regression tree model:')

print(model)
```

注意：当对一个数据集作预测时，批量预测比在每个数据点上分别调用方法`predict`分别预测的效率要高很多。原因在于 Python 代码在底层调用的是 Scala 的决策树模型。

4.3. 朴素贝叶斯

朴素贝叶斯是一种简单的多类分类方法，它假定分类特征之间两两不相关。朴素贝叶斯在训练上非常有效率。对训练集中的每一记录，它计算每一特征在类标签上的条件概率分布，然后运用贝叶斯理论计算某一观察在类标签的条件概率分布，并用之来预测。

MLlib 支持多模朴素贝叶斯（multinomial naive Bayes），一种主要用于文档分类的算法。用于此场景时，每个观察者是一个文档，每个特征代表一个单词，特征的值是单词的频率。特征值必须是非零的单词出现频率。附加的平滑处理可通过设置参数 λ（默认值为 1.0）来完成。对于文档分类，输入特征向量通常是稀疏的，并且能够获得稀疏输入的特有优势。由于熟练数据只是用一次，因此没有必要缓冲它。

4.3.1. 示例

Scala

NaiveBayes实现了多模朴素贝叶斯算法。它接收一个 LabeledPoint格式的RDD和一个可选的平滑参数lambda作为输入，输出一个用于评估和预测的 NaiveBayesModel模型。

```
import org.apache.spark.mllib.classification.NaiveBayes

import org.apache.spark.mllib.linalg.Vectors

import org.apache.spark.mllib.regression.LabeledPoint


val data = sc.textFile("data/mllib/sample_naive_bayes_data.txt")

val parsedData = data.map { line =>

  val parts = line.split(',')

  LabeledPoint(parts(0).toDouble, Vectors.dense(parts(1).split(' ').map(_.toD
ouble)))

}

// Split data into training (60%) and test (40%).

val splits = parsedData.randomSplit(Array(0.6, 0.4), seed = 11L)

val training = splits(0)

val test = splits(1)


val model = NaiveBayes.train(training, lambda = 1.0)
```

```
val predictionAndLabel = test.map(p => (model.predict(p.features), p.label))
val accuracy = 1.0 * predictionAndLabel.filter(x => x._1 == x._2).count() / te
st.count()
```

Java

NaiveBayes实现了多模朴素贝叶斯算法。它接收一个 LabeledPoint格式的Scala RDD和一个可选的平滑参数lambda作为输入，输出一个用于评估和预测的 NaiveBayesModel模型。

```
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.function.Function;
import org.apache.spark.api.java.function.PairFunction;
import org.apache.spark.mllib.classification.NaiveBayes;
import org.apache.spark.mllib.classification.NaiveBayesModel;
import org.apache.spark.mllib.regression.LabeledPoint;
import scala.Tuple2;

JavaRDD<LabeledPoint> training = ... // training set
JavaRDD<LabeledPoint> test = ... // test set

final NaiveBayesModel model = NaiveBayes.train(training.rdd(), 1.0);

JavaPairRDD<Double, Double> predictionAndLabel =
  test.mapToPair(new PairFunction<LabeledPoint, Double, Double>() {
    @Override public Tuple2<Double, Double> call(LabeledPoint p) {
      return new Tuple2<Double, Double>(model.predict(p.features()), p.label
());
    }
  });
double accuracy = 1.0 * predictionAndLabel.filter(new Function<Tuple2<Double,
```

```
Double>, Boolean>() {
   @Override public Boolean call(Tuple2<Double, Double> pl) {
     return pl._1() == pl._2();
   }
 }).count() / test.count();
```

Python

NaiveBayes实现了多模朴素贝叶斯算法。它接收一个 LabeledPoint格式的RDD和一个可选的平滑参数lambda作为输入 ,输出一个用于评估和预测的 NaiveBayesModel模型。

```
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.classification import NaiveBayes

# an RDD of LabeledPoint
data = sc.parallelize([
  LabeledPoint(0.0, [0.0, 0.0])
  ... # more labeled points
])

# Train a naive Bayes model.
model = NaiveBayes.train(data, 1.0)

# Make prediction.
prediction = model.predict([0.0, 0.0])
```

第 5 章 协同过滤

- 协同过滤
  - 显式反馈与隐式反馈
  - 对正则化参数的调整（Scaling of the regularization parameter）
- 示例

- 教程

5.1. 协同过滤

协同过滤常用来实现推荐系统。该技术力图填充用户与物品关联矩阵（user-item association matrix）中缺失的实体。MLlib 支持基于模型的协同过滤，在这里，用户和产品称之为是潜伏因子的一个子集，其用途是预测缺失实体。MLlib 使用交替最小二乘法（ALS）来学习这些潜伏因子。MLlib 中实现含有下面参数：

- numBlocks 是用于并行计算的块数量（设置为-1 表示自动配置）.
- rank 是模型中潜伏因子的数量。
- iterations 是迭代次数。
- lambda 指定交替最小二乘法中的正则化参数。
- implicitPrefs 指定究竟使用显性反馈 ALS 变体还是从数据中抽取隐形反馈（implicitPrefs specifies whether to use the explicit feedback ALS variant or one adapted for implicit feedback data）。
- alpha 是隐式反馈 ALS 变体的参数，它管控了观测偏好的置信度阈值（is a parameter applicable to the implicit feedback variant of ALS that governs the baseline confidence in preference observations）。

5.1.1. 显式反馈与隐式反馈

标准处理方式下，协同过滤把用户与物品关联矩阵中的实体当作用户对物品的显性喜好，基于此来发现矩阵因子。

但在现实世界里，更常见的情形是隐式反馈（如浏览，点击，购买，点赞，分享等）并且只有隐式反馈。在MLlib中处理这类数据的方法请参见“Collaborative Filtering for Implicit Feedback Datasets”。该方法并不直接拟合评级矩阵，而是将数据视为“是否喜好”和“置信度水平”的组合。对用户的喜好来说，评级与置信度水平相关，而不是用户对物品的显式评级值。然后模型尽可能的寻找潜伏因子，以用来预测用户对物品喜好。

5.1.2. 对正则化参数的调整（Scaling of the regularization parameter）

从版本 1.1 开始，在解决每一最小二乘法问题时，我们使用（1）更新用户因子时用户生成的评级数量或（2）更新产品因子时产品得到的产品评级数量，来调整正则化参数 lambda。这种方法称之为“ALS-WR”，在文章 “Large-Scale Parallel Collaborative Filtering for the Netflix Prize” 中有详细论述。它降低lambda对数据集大小的相关性。也因于此，我们能够把从抽样数据子集学习获得最佳参数应用到数据全集。

5.2. 示例

5.2.1 Scala

在下面例子中，我们加载评级数据。每行由一个用户，一个产品和一个评级组成。我们假定评级信息是显性的，所以使用默认的 ALS.train()方法来训练。最后，我们计算评级预测的均方误差来评估推荐模型。

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

// Load and parse the data
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)
  })

// Build the recommendation model using ALS
val rank = 10
val numIterations = 20
val model = ALS.train(ratings, rank, numIterations, 0.01)

// Evaluate the model on rating data
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)
```

如果评级矩阵从另一信息演化而来（比如，从其他信号推断获得），您可以使用 trainImplicit 方法以获得更佳结果。

```
val alpha = 0.01
val model = ALS.trainImplicit(ratings, rank, numIterations, alpha)
```

5.2.2 Java

所有 MLlib 方法都使用 Java 友好的类型，所以您可以像 scala 中那样导入和调用。唯一要说明的是那些使用 Scala RDD 对象的方法，因为在 spark 的 java API 中使用的是 JavaRDD 类。对 JavaRDD 对象，您能够通过调用.rdd()方法将其转换为对应的 Scala 对象。与 Scala 示例等效的应用示例如下：

```
import scala.Tuple2;

import org.apache.spark.api.java.*;
import org.apache.spark.api.java.function.Function;
import org.apache.spark.mllib.recommendation.ALS;
import org.apache.spark.mllib.recommendation.MatrixFactorizationModel;
import org.apache.spark.mllib.recommendation.Rating;
import org.apache.spark.SparkConf;

public class CollaborativeFiltering {
  public static void main(String[] args) {
    SparkConf conf = new SparkConf().setAppName("Collaborative Filtering Examp
le");
```

```
    JavaSparkContext sc = new JavaSparkContext(conf);

    // Load and parse the data
    String path = "data/mllib/als/test.data";
    JavaRDD<String> data = sc.textFile(path);
    JavaRDD<Rating> ratings = data.map(
      new Function<String, Rating>() {
        public Rating call(String s) {
          String[] sarray = s.split(",");
          return new Rating(Integer.parseInt(sarray[0]), Integer.parseInt(sarr
ay[1]),
                            Double.parseDouble(sarray[2]));
        }
      }
    );

    // Build the recommendation model using ALS
    int rank = 10;
    int numIterations = 20;
    MatrixFactorizationModel model = ALS.train(JavaRDD.toRDD(ratings), rank, n
umIterations, 0.01);

    // Evaluate the model on rating data
    JavaRDD<Tuple2<Object, Object>> userProducts = ratings.map(
      new Function<Rating, Tuple2<Object, Object>>() {
        public Tuple2<Object, Object> call(Rating r) {
          return new Tuple2<Object, Object>(r.user(), r.product());
        }
      }
    );
    JavaPairRDD<Tuple2<Integer, Integer>, Double> predictions = JavaPairRDD.fr
```

```
omJavaRDD(
    model.predict(JavaRDD.toRDD(userProducts)).toJavaRDD().map(
      new Function<Rating, Tuple2<Tuple2<Integer, Integer>, Double>>() {
        public Tuple2<Tuple2<Integer, Integer>, Double> call(Rating r){
          return new Tuple2<Tuple2<Integer, Integer>, Double>(
            new Tuple2<Integer, Integer>(r.user(), r.product()), r.rating());
        }
      }
  ));
  JavaRDD<Tuple2<Double, Double>> ratesAndPreds =
    JavaPairRDD.fromJavaRDD(ratings.map(
      new Function<Rating, Tuple2<Tuple2<Integer, Integer>, Double>>() {
        public Tuple2<Tuple2<Integer, Integer>, Double> call(Rating r){
          return new Tuple2<Tuple2<Integer, Integer>, Double>(
            new Tuple2<Integer, Integer>(r.user(), r.product()), r.rating());
        }
      }
  )).join(predictions).values();
  double MSE = JavaDoubleRDD.fromRDD(ratesAndPreds.map(
    new Function<Tuple2<Double, Double>, Object>() {
      public Object call(Tuple2<Double, Double> pair) {
        Double err = pair._1() - pair._2();
        return err * err;
      }
    }
  ).rdd()).mean();
  System.out.println("Mean Squared Error = " + MSE);
 }
}
```

为了运行上面的独立程序，请参考 Spark quick-start 指引中的 Standalone

Applications 章节。务必将 spark-mllib 作为编译依赖库。

5.2.3 Python

在下面例子中，我们加载评级数据。每行由一个用户，一个产品和一个评级组成。我们假定评级信息是显性的，所以使用默认的 ALS.train()方法来训练。最后，我们计算评级预测的均方误差来评估推荐模型。

```
from pyspark.mllib.recommendation import ALS
from numpy import array


# Load and parse the data
data = sc.textFile("data/mllib/als/test.data")
ratings = data.map(lambda line: array([float(x) for x in line.split(',')]))


# Build the recommendation model using Alternating Least Squares
rank = 10
numIterations = 20
model = ALS.train(ratings, rank, numIterations)


# Evaluate the model on training data
testdata = ratings.map(lambda p: (int(p[0]), int(p[1])))
predictions = model.predictAll(testdata).map(lambda r: ((r[0], r[1]), r[2]))
ratesAndPreds = ratings.map(lambda r: ((r[0], r[1]), r[2])).join(predictions)
MSE = ratesAndPreds.map(lambda r: (r[1][0] - r[1][1])**2).reduce(lambda x, y:
x + y)/ratesAndPreds.count()
print("Mean Squared Error = " + str(MSE))
```

如果评价矩阵从另一信息员演化而来（比如，从其他信号推断获得），您可以使用 trainImplicit 方法以获得更佳结果。

```
# Build the recommendation model using Alternating Least Squares based on impl
icit ratings
model = ALS.trainImplicit(ratings, rank, numIterations, alpha = 0.01)
```

5.3 教程教程

Spark 2014 年峰会中包括一个使用 MLlib 建立个性化影视推荐系统的实践教程。

第 6 章 聚类

- 聚类
- 示例

6.1. 聚类

聚类是一个无监督学习问题，我们的意图在于基于某种相似性概念将数据实体分成不同的子集。聚类常用语探索式分析和（或）作为多层监督学习管道中的一个组件（其中，为每一个簇训练出独立的分类或回归模型）。

MLLIb支持 k-means聚类算法，这是将数据点划分为预期簇个数的最常用聚类算法之一。MLLib实现了 k-means++的并行化的演变版本 kmeans||. 该算法在MLLib中的实现的参数有：

- k ，期望的簇个数.
- maxIterations ，算法最大迭代次数.
- initializationMode ，初始化方法，即使用随机方法还是通过 k-means||方法进行初始化.
- runs ，运行 k-means 的次数(k-means 并不保证找到一个全局最优解，在给定数据集上多次运行得到多个结果时，程序将返回最优的那个结果).
- initializationSteps ，k-means||算法的步数
- epsilon ，即用于认定 K-means 收敛的最小距离.

6.2. 示例

6.2.1 Scala

如下代码可在 spark-shell 上执行。

在下面的示例中，我们将首先导入和解析输入数据，之后使用 KMeans对象将各数据点分到两个类簇中。期望得到的类簇个数将作为参数传给算法，然后计算集内均方差总和(Within Set Sum of Squared Error, WSSSE)。你可通过增加k值来降低该误差. 事实上，当k的取值理想时，WSSSE图中将会有一个“低谷点”。

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

```scala
import org.apache.spark.mllib.linalg.Vectors


// Load and parse the data
val data = sc.textFile("data/mllib/kmeans_data.txt")
val parsedData = data.map(s => Vectors.dense(s.split(' ').map(_.toDouble)))


// Cluster the data into two classes using KMeans
val numClusters = 2
val numIterations = 20
val clusters = KMeans.train(parsedData, numClusters, numIterations)


// Evaluate clustering by computing Within Set Sum of Squared Errors
val WSSSE = clusters.computeCost(parsedData)
println("Within Set Sum of Squared Errors = " + WSSSE)
```

6.2.2 Java

MLlib 中的函数都使用 Java 友好的定义方式，所以你如在 Scala 中那样导入并调用他们。唯一需要留心的是这些函数都需要引入 Scala RDD 对象，但在 Spark 的 Java API 中使用独立的 JavaRDD 类。你可以通过调用 JavaRDD 的.rdd()方法来将一个 Java RDD 转为一个 Scala RDD。和 Scala 实现等价的一种 Java 实现方法如下:

```java
import org.apache.spark.api.java.*;
import org.apache.spark.api.java.function.Function;
import org.apache.spark.mllib.clustering.KMeans;
import org.apache.spark.mllib.clustering.KMeansModel;
import org.apache.spark.mllib.linalg.Vector;
import org.apache.spark.mllib.linalg.Vectors;
import org.apache.spark.SparkConf;


public class KMeansExample {
  public static void main(String[] args) {
```

```
    SparkConf conf = new SparkConf().setAppName("K-means Example");
    JavaSparkContext sc = new JavaSparkContext(conf);

    // Load and parse data
    String path = "data/mllib/kmeans_data.txt";
    JavaRDD<String> data = sc.textFile(path);
    JavaRDD<Vector> parsedData = data.map(
      new Function<String, Vector>() {
        public Vector call(String s) {
          String[] sarray = s.split(" ");
          double[] values = new double[sarray.length];
          for (int i = 0; i < sarray.length; i++)
            values[i] = Double.parseDouble(sarray[i]);
          return Vectors.dense(values);
        }
      }
    );

    // Cluster the data into two classes using KMeans
    int numClusters = 2;
    int numIterations = 20;
    KMeansModel clusters = KMeans.train(parsedData.rdd(), numClusters, numIter
ations);

    // Evaluate clustering by computing Within Set Sum of Squared Errors
    double WSSSE = clusters.computeCost(parsedData.rdd());
    System.out.println("Within Set Sum of Squared Errors = " + WSSSE);
  }
}
```

为了运行上面的独立程序，请参考 Spark quick-start 指引中的 Standalone

Applications 章节。务必将 spark-mllib 作为编译依赖库。

6.2.3 Python

如下例子可在 PySpark shell 中测试。

下面示例中，我们将首先导入和解析输入数据，之后使用 KMeans 对象将各数据点分到两个簇中。期望得到的簇个数将作为参数传给算法，然后计算集内均方差总和(Within Set Sum of Squared Error, WSSSE)。你可通过增加 k 值来降低该误差。事实上，当 k 的取值理想时 WSSE 图中将会有一个”低谷点”。

```
from pyspark.mllib.clustering import KMeans
from numpy import array
from math import sqrt


# Load and parse the data
data = sc.textFile("data/mllib/kmeans_data.txt")
parsedData = data.map(lambda line: array([float(x) for x in line.split(' ')]))


# Build the model (cluster the data)
clusters = KMeans.train(parsedData, 2, maxIterations=10,
       runs=10, initializationMode="random")


# Evaluate clustering by computing Within Set Sum of Squared Errors
def error(point):
   center = clusters.centers[clusters.predict(point)]
   return sqrt(sum([x**2 for x in (point - center)]))


WSSSE = parsedData.map(lambda point: error(point)).reduce(lambda x, y: x + y)
print("Within Set Sum of Squared Error = " + str(WSSSE))
```

第 7 章 降维

降维是减少所需要考虑变量个数的过程。它能从原始的有噪音的特征或压缩数据中提取隐藏特征，同时保留原有的结构。MLlib 提供 RowMatrix 用于降维。

7.1. 奇异值分解

奇异值分解将一个矩阵分解成为 U, Σ, 和 V三个 子矩阵， 它们满足：

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

其中：

- U 是一个正交矩阵，其每一个列被称为一个左奇异向量。
- Σ 是一个对角矩阵，其对角线上元素非负且按降序排列。对角线上的元素被称为奇异值。
- V 是一个正交矩阵，其每一个列被称为一个右奇异向量。

对于大的矩阵，我们通常并不需要进行完全的因式分解，而只需要求值靠前的部分奇异值和对应的奇异向量即可。这样能降低存储需求，去噪声而且降低了子矩阵的阶数。

假设我们保留前 k 个奇异值，那么相应的子矩阵的维度为:

- U: m×k,
- Σ: k×k,
- V: n×k.

7.1.1 性能分析

我们假设 n比m要小。奇异值和右奇异矩阵可从格拉姆矩阵(Gramian matrix) A^TA.的共轭值和共轭向量的求解而得出。如果用户通过输入computeU 这一参数指定了需求解左奇异矩阵U，则该矩阵可通过$U=A(VS^{-1})$这一矩阵乘积得出. 实际计算时，程序会自动根据计算的复杂度来选择不同的方法来得出结果：

- 如果n较小或者k相对n来说比较大，那么我们会先计算格拉姆矩阵，然后在driver本地计算它的前K个奇异值和特征值。其代价包括一次在driver和各个executor本地上的一次复杂度为O(n^2)的存储，外加一次driver本地进行的复杂度为O(n^2k)的存储。
- 否则，我们将分布式计算 ，然后将其作为输入通过 ARPACK 来计算(ATA)的前 K 各特征值和特征矩阵。ARPACK 的计算在 driver 上进行。这个过程的需要 O(k)轮， 在每个 executor 上 O(n)存储，以及 driver 上 O(nk)存储。

7.1.2 SVD 应用示范

MLLib可对行矩阵进行SVD运算。RowMatrix类包含了该类矩阵。

Scala

```
import org.apache.spark.mllib.linalg.Matrix

import org.apache.spark.mllib.linalg.distributed.RowMatrix

import org.apache.spark.mllib.linalg.SingularValueDecomposition


val mat: RowMatrix = ...


// Compute the top 20 singular values and corresponding singular vectors.

val svd: SingularValueDecomposition[RowMatrix, Matrix] = mat.computeSVD(20, c
omputeU = true)

val U: RowMatrix = svd.U // The U factor is a RowMatrix.

val s: Vector = svd.s // The singular values are stored in a local dense vecto
r.

val V: Matrix = svd.V // The V factor is a local dense matrix.
```

如果 U 被定义为一个 ，那么我们仍然可用上述代码来处理 U。

Java

```
import java.util.LinkedList;


import org.apache.spark.api.java.*;

import org.apache.spark.mllib.linalg.distributed.RowMatrix;

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.rdd.RDD;
import org.apache.spark.SparkConf;
import org.apache.spark.SparkContext;

public class SVD {
  public static void main(String[] args) {
    SparkConf conf = new SparkConf().setAppName("SVD Example");
    SparkContext sc = new SparkContext(conf);

    double[][] array = ...
    LinkedList<Vector> rowsList = new LinkedList<Vector>();
    for (int i = 0; i < array.length; i++) {
      Vector currentRow = Vectors.dense(array[i]);
      rowsList.add(currentRow);
    }
    JavaRDD<Vector> rows = JavaSparkContext.fromSparkContext(sc).parallelize
(rowsList);

    // Create a RowMatrix from JavaRDD<Vector>.
    RowMatrix mat = new RowMatrix(rows.rdd());

    // Compute the top 4 singular values and corresponding singular vectors.
    SingularValueDecomposition<RowMatrix, Matrix> svd = mat.computeSVD(4, tru
e, 1.0E-9d);
    RowMatrix U = svd.U();
    Vector s = svd.s();
    Matrix V = svd.V();
```

```
  }
}
```

如果 U 被定义为一个 ，那么我们仍然可用上述代码来处理 U。

运行上述代码前请参见Spark 快速指南中的Standalone Applications 章节中的相关说明。同时也需要将 spark-mllib 作为依赖加入到你程序的构建配置中。

7.2. 主成份分析

主成份分析是使用统计方法求矩阵一个旋转的方法。该旋转需要能最大化矩阵第一坐标的方差，这同时也将使得后续坐标的方差也最大化。旋转后矩阵的每一个列被称作一个主成份. 主成份分析被广泛用于降维中。

MLLib 的 SVD 计算适用于行数远多于列数的矩阵。这些矩阵以每行代表一条记录的面向行格式来存储。

7.2.1 Scala

如下代码演示如何对一个 RowMatrix 的进行主成份分析，然后利用分析结果将原始的向量投影到一个低维空间中。

```
import org.apache.spark.mllib.linalg.Matrix

import org.apache.spark.mllib.linalg.distributed.RowMatrix


val mat: RowMatrix = ...


// Compute the top 10 principal components.

val pc: Matrix = mat.computePrincipalComponents(10) // Principal components a
re stored in a local dense matrix.


// Project the rows to the linear space spanned by the top 10 principal compon
ents.

val projected: RowMatrix = mat.multiply(pc)
```

7.2.2 Java

如下代码演示如何对一个 RowMatrix 的进行主成份分析，然后利用分析结果将原始的

向量投影到一个低维空间中。数据的列数应比较小，如，低于 100 列。

```
import java.util.LinkedList;


import org.apache.spark.api.java.*;
import org.apache.spark.mllib.linalg.distributed.RowMatrix;
import org.apache.spark.mllib.linalg.Matrix;
import org.apache.spark.mllib.linalg.Vector;
import org.apache.spark.mllib.linalg.Vectors;
import org.apache.spark.rdd.RDD;
import org.apache.spark.SparkConf;
import org.apache.spark.SparkContext;


public class PCA {
  public static void main(String[] args) {
    SparkConf conf = new SparkConf().setAppName("PCA Example");
    SparkContext sc = new SparkContext(conf);


    double[][] array = ...
    LinkedList<Vector> rowsList = new LinkedList<Vector>();
    for (int i = 0; i < array.length; i++) {
      Vector currentRow = Vectors.dense(array[i]);
      rowsList.add(currentRow);
    }
    JavaRDD<Vector> rows = JavaSparkContext.fromSparkContext(sc).parallelize
(rowsList);


    // Create a RowMatrix from JavaRDD<Vector>.
    RowMatrix mat = new RowMatrix(rows.rdd());


    // Compute the top 3 principal components.
```

```
    Matrix pc = mat.computePrincipalComponents(3);

    RowMatrix projected = mat.multiply(pc);

  }

}
```

为了运行上面的独立程序，请参考 Spark quick-start 指引中的 Standalone Applications 章节。务必将 spark-mllib 作为编译依赖库。

第 8 章 特征提取和变换

- 词频-逆文档频率（TF-IDF）
  - 二元分类（支持向量机，逻辑回归）
  - 线性回归（最小二乘法，Lasso，ridge）
- 词向量化工具（Word2Vec）
  - 模型
  - 示例
- 标准化（StandardScaler）
  - 模型适配
  - 示例
- 范数化（Normalizer）
  - 示例

8.1 词频-逆文档频率（TF-IDF）

词频-逆文档频率（TF-IDF）是一个在文本挖掘中广泛应用的特征向量化方法，它能反映出语料库中某篇文档中某个词的重要性。假定t表示一个词，d表示一篇文档，则词频 TF(t,d)是某个词t在文档d中的出现次数，而文档频率 DF(t,D)是包含词t的文档d的数目。如果我们仅使用词频来衡量重要性，则很容易过分强调那些出现非常频繁但携带很少与文档相关信息量的词，比如英语中的“a”、“the”和“of”。如果一个词在在语料库中出现非常频繁，意味着它更不能携带特定文档的特定信息。逆文档频率就是一个用于度量一个词能提供多少信息量的数值：

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

在这里，|D| 是语料库中文档总数。由于使用对数，如果一个词在所有文档中都出现，

则其IDF的值变为 0。请注意，一个平滑值添加到公式中，目的是避免出现一个词不在语料库而导致零除问题。而词频-逆文档频率（TF-IDF）是TF和IDF简单相乘：

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

当然，存在其他用于衡量词频和文档频率的变体。在 MLlib 中，为了扩展性，我们将 TF 和 IDF 分开。

我们使用散列技巧（hashing trick）来实现词频。我们运用一个哈希函数将原始特征映射到一个特征索引值。然后基于映射索引值来计算词频。这种方法避免计算全局“词-索引（term-to-index）”映射，而在超大语料中计算全局“词-索引”的代价非常高。这种方法代价是会出现潜在的哈希值冲突——即不同原始特征被映射到同一个哈希值，从而变成同一个词。为了降低这种冲突概率，我们增加了的目标特征的维数，也即，在哈希表中散列桶的数量。默认的特征维数是 2^{20}=1,048,576。

注意：MLlib并不提供文本分割工具，我们建议用户参考“斯坦福自然语言处理组（Stanford NLP Group）”和“scalanlp/chalk开源项目”

8.1.1 Scala

词频（TF）和逆文档频率（IDF）在HashingTF和IDF中实现。HashingTF接收一个 RDD[Iterable[_]]实例作为输入。每个记录都是一个可遍历的String或其他类型。

```
import org.apache.spark.rdd.RDD

import org.apache.spark.SparkContext

import org.apache.spark.mllib.feature.HashingTF

import org.apache.spark.mllib.linalg.Vector


val sc: SparkContext = ...


// Load documents (one per line).

val documents: RDD[Seq[String]] = sc.textFile("...").map(_.split(" ").toSeq)


val hashingTF = new HashingTF()

val tf: RDD[Vector] = hasingTF.transform(documents)
```

然而，应用 HashingTF只需要遍历一次数据，而应用 IDF则需要遍历两次：第一次用于计算IDF向量，第二次使用IDF来调整词频。

```
import org.apache.spark.mllib.feature.IDF
```

```
// ... continue from the previous example
tf.cache()
val idf = new IDF().fit(tf)
val tfidf: RDD[Vector] = idf.transform(tf)
```

8.2. 词向量化工具（Word2Vec）

Word2Vec 用于将词转换为分布式词向量格式(distributed vector representation of words)。分布式格式的主要优点在于在向量空间中相似词相近，使得更易生成小说模式以及模型评估更加健壮(which makes generalization to novel patterns easier and model estimation more robust)。分布式向量格式在很多自然语言分析应用中很有用，如命名实体识别、消歧、解析、打标签和机器翻译。

8.2.1 模型

我们使用skip-gram模型来实现Word2Vec。在skip-gram模型中，训练目标是学习到同一句子中最能预测其环境的词向量表示。从数学上来说，给定一系列词 w1,w2,...,wT，skip-gram模型最大化对数似然均值：

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

在这里，K 是训练窗口大小。

在skip-gram模型中，每个词W都与两个向量 uw和 vw相关，uw和 vw表示词W自身及其上下文。在softmax模型中，给定词 wj，正确预测词 wi的概率由下式决定：

在这里，V 是词量大小。

在skip-gram模型中使用softmax的代价很高，因为 logp(wi|wj)的计算量随着V线性增长，而V很容易就达到百万级。为了提高训练Word2Vec的速度，我们使用分层softmax技术，该方法可以将计算 logp(wi|wj)的复杂度降低到 O(log(V))。

8.2.2 示例

下面示例演示了怎样加载文本数据，将其解析为一个格式为 Seq[String]的RDD，构建一个实例 Word2Vec，然后将输入数据匹配为一个 Word2VecModel模型。最后，我们展

示了与给定词最同义的 40 个词。为了运行该示例，首先需要下载 text8数据并解压到目标目录。在这里，我们假定解压 text8到你运行spark shell目录。

Scala

```
import org.apache.spark._

import org.apache.spark.rdd._

import org.apache.spark.SparkContext._

import org.apache.spark.mllib.feature.Word2Vec


val input = sc.textFile("text8").map(line => line.split(" ").toSeq)


val word2vec = new Word2Vec()


val model = word2vec.fit(input)


val synonyms = model.findSynonyms("china", 40)


for((synonym, cosineSimilarity) <- synonyms) {

  println(s"$synonym $cosineSimilarity")

}
```

8.3. 标准化（StandardScaler

特征标准化是在训练集上对每列使用统计分析技术，将数据调整为标准差的倍数以及（或）去均值。这是一种非常通用的预处理步骤。

比如，支持向量机中的 RBF 内核或 L1/L2 正则化线性模型就在特征具有单位变化以及（或）零均值时效果更好。

StandardScaler 能够在优化过程中加快收敛速度，也能够让特征免于被训练时的一个超大值发生剧烈影响。

8.3.1 模型适配

StandardScaler 的构建函数具有下列参数：

- withMean，默认值 False。在调整前将数据中心化处理。它的输出是密集的，不能工作于稀疏输入（抛出异常）。
- withStd，默认值 True。将数据调整为标准差。

在 StandardScaler 中，我们提供一个 fit 方法，它接受 RDD[Vector]格式的输入，进行统计分析，然后输出一个标准差的倍数以及（或）去均值化的模型，模型结果依赖于我们如何配置 StandardScaler。

模型输出一个 StandardScaler 模型，它能够将一个 Vector 标准化，也能够将一个 RDD[Vector]标准化。

8.3.2 示例

下面示例演示了怎样加载一个 libsvm 格式的数据集，将特征标准化，转换后的新特征是标准差的倍数以及（或）具有 0 均值。

Scala

```
import org.apache.spark.SparkContext._
import org.apache.spark.mllib.feature.StandardScaler
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))

// data1 will be unit variance.
val data1 = data.map(x => (x.label, scaler1.transform(x.features)))

// Without converting the features into dense vectors, transformation with zer
```

```
o mean will raise

// exception on sparse vector.

// data2 will be unit variance and zero mean.

val data2 = data.map(x => (x.label, scaler2.transform(Vectors.dense(x.feature
s.toArray))))
```

8.4. 范数化（Normalizer）

范数化（Normalizer）将独立的样本调整为具有 Lp 范数。这是在文本分类或聚类中广泛应用的一个操作。比如，两个 L2 化的 TF-IDF 向量，其点积就是是向量间的余弦相似度。

Normalizer 的构建函数具有下列参数：

- p，默认值 2。在 Lp 空间范数化。

在 Normalizer 中，提供 transform 方法，它能够将一个 Vector 范数化化，也能够将一个 RDD[Vector]范数化。

注意：若输入的模数是 0，它将返回原值。

8.4.1 示例

下面示例演示了怎样加载一个 libsvm 格式的数据集，将其分别以 L2 范数和 L∞范数范数化。

Scala

```
import org.apache.spark.SparkContext._

import org.apache.spark.mllib.feature.Normalizer

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 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)))
```

第 9 章. 优化器（开发者）

- 数学描述
  - 梯度下降（Gradient descent）
  - 随机梯度下降（Stochastic gradient descent，SGD）
  - Update schemes for distributed SGD
  - Limited-memory BFGS (L-BFGS)
- MLlib 的实现
  - 梯度下降（Gradient descent）与随机梯度下降（Stochastic gradient descent，SGD）
  - L-BFGS

9.1. 数学描述

9.1.1 梯度下降（Gradient descent）

解决具有$\min_{\mathbf{w}\in\mathbb{R}^d} f(\mathbf{w})$形式优化问题的最简单方法是梯度下降。这些第一类优化器（包括梯度下降和她的随机变体）非常适用于大规模的分布式计算。

梯度下降方法解决的是一个函数局部最小化问题，它沿着坡度更强的方向迭代前进，该方向是该函数在当前点的导函数的负值（称之为梯度），也即，当前的参数化值。如果目标函数并不是处处可导，但依然是凸的，于是存在一个子梯度（是梯度概念的延伸），替代逼近方向。在任何场景中，计算一个数据集的梯度或子梯度代价很高——为了计算所有缺失值（loss terms）的贡献度，它需要遍历整个数据集。

9.1.2 随机梯度下降（Stochastic gradient descent，SGD）

目标函数可以用一个和值来表达的优化问题，特别适合使用随机梯度下降（SGD）方法来解决。在我们的场景里，即为在监督机器学习中广泛应用的优化公式：

$$f(\mathbf{w}) := \lambda R(\mathbf{w}) + \frac{1}{n}\sum_{i=1}^{n} L(\mathbf{w};\mathbf{x}_i,y_i)$$

这很自然，因为缺失值一般记为每个数据点中个别缺失值的平均值(an average of the individual losses coming from each datapoint)。

随机子梯度是在一个向量中一次随机化选择，但我们期望得到原始梯度函数一个真正的子梯度。机会均等的选择一个数据点 i∈[1..n]，我们得到上面公司的一个随机子梯度，其关于w的公式是：

$$f'_{\mathbf{w},i} := L'_{\mathbf{w},i} + \lambda R'_{\mathbf{w}}$$

其中 $L'_{\mathbf{w},i} \in \mathbb{R}^d$ 是由第i个数据点决定的损失函数第一部分的一个子梯度，其公式为 $L'_{\mathbf{w},i} \in \frac{\partial}{\partial \mathbf{w}} L(\mathbf{w};\mathbf{x}_i,y_i)$。此外，$R'_{\mathbf{w}}$ 是正则化器 R(w)的的一个子梯度，即 $R'_{\mathbf{w}} \in \frac{\partial}{\partial \mathbf{w}} R(\mathbf{w})$。$R'_{\mathbf{w}}$并不依赖于哪个随机数据点被选中。当然，期望在对 i∈[1..n]的随机选择时，我们得到的$f'_{\mathbf{w},i}$是原始目标函数f的子梯度，即 $\mathbb{E}\left[f'_{\mathbf{w},i}\right] \in \frac{\partial}{\partial \mathbf{w}} f(\mathbf{w})$。

于是，运行 SGD 就是简单地沿着梯度下降$J_{\mathbf{w},i}$ 相反的方向前进，也即：

$$\mathbf{w}^{(t+1)} := \mathbf{w}^{(t)} - \gamma\, f'_{\mathbf{w},i}$$

- Step-size. 参数 γ是步长，在默认实现里，随着迭代次数逐渐降低，也即，在第t次迭代时，$\gamma := \frac{s}{\sqrt{t}}$，这里的s是输入参数 s= stepSize。请注意，在实践中，为SGD选择最佳步长通常很微妙，也是一个活跃的研究主题。
- 梯度。在分类与回归章节（classification and regression）中，有一张关于MLlib机器学习算法相关梯度的表。
- 近似更新。在逐步逼近过程中，除了正则化因子的子梯度$R'(\mathbf{w})$外，在某些场景下，一种替代方案是使用近似操作替代。对L1 正则化来说，其近似操作由软阈值提供，请参考 L1Updater实现。

9.1.3 分布式 SGD 的更新模式（Update schemes for distributed SGD）

在 GradientDescent中实现的SGD，对数据样本进行使用简单（分布）抽样。回顾一下第一类优化问题的损失函数是 $\frac{1}{n}\sum_{i=1}^{n} L(\mathbf{w};\mathbf{x}_i,y_i)$，以及 $\frac{1}{n}\sum_{i=1}^{n} L'_{\mathbf{w},i}$ 将是一个真正的子梯度。因为这要求访问数据全集，替代方式是运用参数miniBatchFraction用来指定全集数据的一个子集。在子集中的梯度平均值，即 $\frac{1}{|S|}\sum_{i\in S} L'_{\mathbf{w},i}$ 是一个随机梯度。在这里，S是一个大小为 $|S| = \texttt{miniBatchFraction} \cdot n$ 的抽样子集。在每一次迭代中，在分布式数据集（RDD）上的抽样，就像从每一工作节点计算部分和值一样，有标准的Spark子例程实现。

如果抽样比例 miniBatchFraction 设置为 1（这是默认值），那么在每一次迭代会得到严格意义上的梯度（子梯度）。在这种场景下，在逐步逼近时没有随机问题和变体问题。在另一个极端，如果 miniBatchFraction 设置得非常小，以致于仅仅抽样到一个点，即 $|S| = \texttt{miniBatchFraction} \cdot n = 1$，那么算法就等价于标准的 SGD。在那时，逐步逼近就直接依赖于对数据点进行抽样的均衡性。

9.1.4 内存受限(L-BFGS)

L-BFGS 是 quasi-Newton 方法体系中的一个优化算法，用于解决具有 $\min_{\mathbf{w}\in\mathbb{R}^d} f(\mathbf{w})$ 格式的优化问题。L-BFGS 使用一个二次方程来近似目标函数，而不是通过对目标函数求二阶偏导数来构建黑塞矩阵（Hessian Matrix）。黑塞矩阵（Hessian Matrix）使用前一次的梯度评估来近似模拟，所以在 Newton 方法里计算 Hessian 时没有没有纵向扩展问题（即训练特征的数量）。相应地，与其他第一类优化比较， L-BFGS 通常更能快速逼近。

9.2. MLlib 实现

9.2.1. 梯度下降（Gradient descent）与随机梯度下降（Stochastic gradient descent，SGD）

在 MLlib 中，梯度下降方法包括底层组件随机梯度下降（SGD），基于它开发了不同的机器学习算法，请参见线性方法（linear methods）章节获取示例。

SGD 类 GradientDescent 设置下面参数：

- Gradient 是计算被优化函数随机梯度的类，也即，对单个训练样本来说，其当前参数值。MLlib 包括常用损失函数的梯度类，如 hinge，logistic，least-squares。梯度类的输入包括一个训练样本，它的类标签，以及当前参数值。
- Updater 是执行实际梯度逼近的类，也即，在每一次迭代中根据损失值更新权重。Updater 也具有为正则化因子更新权重的功能。MLlib 既包括用于无正则化场景的 Updater，也包括用于 L1 和 L2 正则化的 Updater。
- stepSize 是为梯度下降指定初始步长的标度值。在 MLlib 中，所有 Updater 在第 t 步使用一个步长 stepSize/ $\sqrt{t}$。
- numIterations 是迭代次数。
- regParam 是使用 L1 或 L2 正则化时的正则化参数。
- miniBatchFraction 是在每一次迭代中，用于计算梯度方向的抽样比例，默认值是 1.

梯度下降中的可用算法请参见 API：

- GradientDescent

9.2.2 L-BFGS

L-BFGS 目前是 MLlib 中一个底层优化组件。如果你想在机器学习算法（如线性回归，逻辑回归）中使用 L-BFGS，你必须传递一个梯度目标函数，以及一个自优化的更新器

（updater）,这个优化器替代训练 API 如 LogisticRegressionWithSGD。请参见下面示例。

（后面还有一句：It will be addressed in the next release.

）

自从在 L1Updater 中添加了梯度下降的软阈值逻辑后，不能在使用 L1Updater 来进行 L1 正则化。请参见“开发人员注意事项”。

实现 L-BFGS 的 LBFGS.runLBFGS 方法含有下面参数：

- Gradient 是计算被优化函数随机梯度的类，也即，对单个训练样本来说，其当前参数值。MLlib 包括常用损失函数的梯度类，如 hinge，logistic，least-squares。梯度类的输入包括一个训练样本，它的类标签，以及当前参数值。
- Updater 是 L-BFGS 中为正则化要素计算梯度和目标函数损失的实现类。MLlib 既包括用于无正则化场景的 Updater，也包括用于 L1 和 L2 正则化的 Updater。
- numCorrections 是在 L-BFGS 中更新时修正数。建议为 10.
- maxNumIterations 是 L-BFGS 的最大迭代次数。
- regParam 是使用正则化时的正则化参数。

该方法的返回值是一个二元组。第一个元素是一个列矩阵，值为每一个特征的权重；第二个元素是数组，值为每一次迭代时的损失量。

9.3. 示例

9.3.1 Scala

下面示例使用 L-BFGS 优化器训练一个 L2 正则化的二元逻辑回归模型。

```
import org.apache.spark.SparkContext
import org.apache.spark.mllib.evaluation.BinaryClassificationMetrics
```

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

import org.apache.spark.mllib.util.MLUtils

import org.apache.spark.mllib.classification.LogisticRegressionModel


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

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


// Split data into training (60%) and test (40%).

val splits = data.randomSplit(Array(0.6, 0.4), seed = 11L)


// Append 1 into the training data as intercept.

val training = splits(0).map(x => (x.label, MLUtils.appendBias(x.features))).
cache()


val test = splits(1)


// Run training algorithm to build the model

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))

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

9.3.2 Java

下面示例使用 L-BFGS 优化器训练一个 L2 正则化的二元逻辑回归模型。

```
import java.util.Arrays;
```

```
import java.util.Random;


import scala.Tuple2;


import org.apache.spark.api.java.*;
import org.apache.spark.api.java.function.Function;
import org.apache.spark.mllib.classification.LogisticRegressionModel;
import org.apache.spark.mllib.evaluation.BinaryClassificationMetrics;
import org.apache.spark.mllib.linalg.Vector;
import org.apache.spark.mllib.linalg.Vectors;
import org.apache.spark.mllib.optimization.*;
import org.apache.spark.mllib.regression.LabeledPoint;
import org.apache.spark.mllib.util.MLUtils;
import org.apache.spark.SparkConf;
import org.apache.spark.SparkContext;


public class LBFGSExample {
  public static void main(String[] args) {
    SparkConf conf = new SparkConf().setAppName("L-BFGS Example");
    SparkContext sc = new SparkContext(conf);
    String path = "data/mllib/sample_libsvm_data.txt";
    JavaRDD<LabeledPoint> data = MLUtils.loadLibSVMFile(sc, path).toJavaRDD
();
    int numFeatures = data.take(1).get(0).features().size();


    // Split initial RDD into two... [60% training data, 40% testing data].
    JavaRDD<LabeledPoint> trainingInit = data.sample(false, 0.6, 11L);
    JavaRDD<LabeledPoint> test = data.subtract(trainingInit);


    // Append 1 into the training data as intercept.
```

```
    JavaRDD<Tuple2<Object, Vector>> training = data.map(
      new Function<LabeledPoint, Tuple2<Object, Vector>>() {
        public Tuple2<Object, Vector> call(LabeledPoint p) {
          return new Tuple2<Object, Vector>(p.label(), MLUtils.appendBias(p.features()));
        }
      });
    training.cache();

    // Run training algorithm to build the model.
    int numCorrections = 10;
    double convergenceTol = 1e-4;
    int maxNumIterations = 20;
    double regParam = 0.1;
    Vector initialWeightsWithIntercept = Vectors.dense(new double[numFeatures + 1]);

    Tuple2<Vector, double[]> result = LBFGS.runLBFGS(
      training.rdd(),
      new LogisticGradient(),
      new SquaredL2Updater(),
      numCorrections,
      convergenceTol,
      maxNumIterations,
      regParam,
      initialWeightsWithIntercept);
    Vector weightsWithIntercept = result._1();
    double[] loss = result._2();

    final LogisticRegressionModel model = new LogisticRegressionModel(
      Vectors.dense(Arrays.copyOf(weightsWithIntercept.toArray(), weightsWith
```

```
Intercept.size() - 1)),
      (weightsWithIntercept.toArray())[weightsWithIntercept.size() - 1]);


    // Clear the default threshold.
    model.clearThreshold();


    // Compute raw scores on the test set.
    JavaRDD<Tuple2<Object, Object>> scoreAndLabels = test.map(
      new Function<LabeledPoint, Tuple2<Object, Object>>() {
      public Tuple2<Object, Object> call(LabeledPoint p) {
        Double score = model.predict(p.features());
        return new Tuple2<Object, Object>(score, p.label());
      }
    });


    // Get evaluation metrics.
    BinaryClassificationMetrics metrics =
      new BinaryClassificationMetrics(scoreAndLabels.rdd());
    double auROC = metrics.areaUnderROC();


    System.out.println("Loss of each step in training process");
    for (double l : loss)
      System.out.println(l);
    System.out.println("Area under ROC = " + auROC);
  }
}
```

9.3.3 开发者注意事项

由于Hession近似地通过前一子梯度评估构建，目标函数在整个优化过程中不能改变。

其结果就是，随机梯度下降并不能自自然然的使用 miniBatch。也源于此，再有更好的想法前，我们不提供该功能。

Updater 是为梯度下降设计的类，用于计算实际梯度下降步长。但是，对 L-BFGS 来说，通过忽略进用于梯度下降的逻辑部分（比如 adaptive step size stuff），我们能够获得正则化目标函数的梯度和损失值。在后续工作中，我们将使用正则化器（regularizer）重构这部分内容，将正则化和逼近更新的逻辑分开。

Spark 亚太研究院

Spark 亚太研究院是中国最专业的一站式大数据 Spark 解决方案供应商和高品质大数据企业级完整培训与服务供应商，以帮助企业规划、架构、部署、开发、培训和使用 Spark 为核心，同时提供 Spark 源码研究和应用技术训练。针对具体 Spark 项目，提供完整而彻底的解决方案。包括 Spark 一站式项目解决方案、Spark 一站式项目实施方案及 Spark 一体化顾问服务。

官网： www.sparkinchina.com

近期活动

- 2014 年亚太地区规格最高的 Spark 技术盛会！
- 面向大数据、云计算开发者、技术爱好者的饕餮盛宴！
- 云集国内外 Spark 技术领军人物及灵魂人物！
- 技术交流、应用分享、源码研究、商业案例探询！

时间：2014 年 12 月 6-7 日

地点：北京珠三角万豪酒店

Spark亚太峰会网址：http://www.sparkinchina.com/meeting/2014yt/default.asp

- 如果你是对 Spark 有浓厚兴趣的初学者，在这里你会有绝佳的入门和实践机会！
- 如果你是 Spark 的应用高手，在这里以“武”会友，和技术大牛们尽情切磋！
- 如果你是对 Spark 有深入独特见解的专家，在这里可以尽情展现你的才华！

比赛时间：
2014 年 9 月 30 日—12 月 3 日
Spark开发者大赛网址：http://www.sparkinchina.com/meeting/2014yt/dhhd.asp

视频课程：

《大数据 Spark 实战高手之路》 国内第一个 Spark 视频系列课程

从零起步，分阶段无任何障碍逐步掌握大数据统一计算平台 Spark，从 Spark 框架编写和开发语言 Scala 开始，到 Spark 企业级开发，再到 Spark 框架源码解析、Spark 与 Hadoop 的融合、商业案例和企业面试，一次性彻底掌握 Spark，成为云计算大数据时代的幸运儿和弄潮儿，笑傲大数据职场和人生！

- 第一阶段：熟练的掌握 Scala 语言
  课程学习地址：http://edu.51cto.com/pack/view/id-124.html
- 第二阶段：精通 Spark 平台本身提供给开发者 API
  课程学习地址：http://edu.51cto.com/pack/view/id-146.html
- 第三阶段：精通 Spark 内核
  课程学习地址：http://edu.51cto.com/pack/view/id-148.html
- 第四阶段：掌握基于 Spark 上的核心框架的使用
  课程学习地址：http://edu.51cto.com/pack/view/id-149.html
- 第五阶段：商业级别大数据中心黄金组合：Hadoop+ Spark
  课程学习地址：http://edu.51cto.com/pack/view/id-150.html
- 第六阶段：Spark 源码完整解析和系统定制
  课程学习地址：http://edu.51cto.com/pack/view/id-151.html

近期公开课：

《决胜大数据时代：Hadoop、Yarn、Spark 企业级最佳实践》

集大数据领域最核心三大技术：Hadoop 方向 50%：掌握生产环境下、源码级别下的 Hadoop 经验，解决性能、集群难点问题；Yarn 方向 20%：掌握最佳的分布式集群资源管理框架，能够轻松使用 Yarn 管理 Hadoop、Spark 等；Spark 方向 30%：未来统一的