Broadview®
www.broadview.com.cn

Spark MLlib 机器学习

算法、源码及实战详解

黄美灵 著

本书系统、全面、深入地解析Spark MLlib机器学习的相关知识
以源码为基础，兼顾算法、理论与实战，帮助读者在实际工作中进行MLlib的应用开发和定制开发

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

内 容 简 介

本书以 Spark 1.4.1 版本源码为切入点，全面并且深入地解析 Spark MLlib 模块，着力于探索分布式机器学习的底层实现。

书中本着循序渐进的原则，首先解析 MLlib 的底层实现基础：数据操作及矩阵向量计算操作，该部分是 MLlib 实现的基础；接着对各个机器学习算法的理论知识进行讲解，并且解析机器学习算法如何在 MLlib 中实现分布式计算；然后对 MLlib 源码进行详细的讲解；最后进行 MLlib 实例的讲解。相信通过本书的学习，读者可全面掌握 Spark MLlib 机器学习，能够进行 MLlib 实战、MLlib 定制开发等。

本书适合大数据、Spark、数据挖掘领域的从业人员阅读，同时也为 Spark 开发者和大数据爱好者展现了分布式机器学习的原理和实现细节。

图书在版编目（CIP）数据

Spark MLlib 机器学习：算法、源码及实战详解 / 黄美灵著.—北京：电子工业出版社，2016.4
ISBN 978-7-121-28214-0

Ⅰ. ①S… Ⅱ. ①黄… Ⅲ. ①数据处理软件－机器学习 Ⅳ. ①TP274②TP181

中国版本图书馆 CIP 数据核字（2016）第 039755 号

策划编辑：付　睿
责任编辑：李云静
印　　刷：三河市双峰印刷装订有限公司
装　　订：三河市双峰印刷装订有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　1/16　　印张：25.25　　字数：563 千字
版　　次：2016 年 4 月第 1 版
印　　次：2016 年 4 月第 1 次印刷
印　　数：3000 册　　定价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

前言

机器学习是一门多领域交叉学科，涉及概率论、统计学、逼近论、凸分析、计算复杂性理论等多门学科，其中大部分理论来源于 18、19 世纪，例如贝叶斯定理，是 18 世纪英国数学家托马斯 • 贝叶斯（Thomas Bayes）提出的重要概率论理论；而 21 世纪则侧重于如何将机器学习理论运用在工业化中，帮助改进性能及提升其效率。

机器学习理论主要是设计和分析一些让计算机可以自动“学习”的算法。机器学习算法是一类从数据中自动分析获得规律，并利用规律对未知数据进行预测的算法。在算法设计方面，机器学习理论关注可以实现的、行之有效的学习算法；机器学习研究的不是求解精确的结果，而是研究开发容易处理的近似求解算法。尤其是在 21 世纪，知识和数据量爆发的时代，机器学习面临大数据的求解难题。

随着数据量的增加，从传统的单机计算发展到大规模的集群计算，以至发展到今天的一种大规模、快速计算的集群平台——Apache Spark。Spark 是一个开源集群运算框架，最初由加州大学伯克利分校 AMP 实验室开发。相对于 Hadoop 的 MapReduce 会在执行完工作后将中介资料存放到磁盘中，Spark 使用了内存内运算技术，能在资料尚未写入硬盘时即在内存内分析运算。Spark 在内存上的运算速度比 Hadoop MapReduce 的运算速度快 100 倍，即便是在磁盘上运行也能快 10 倍。Spark 允许将数据加载至集群内存，并多次对其进行查询，非常适合用于机器学习算法。

本书侧重讲解 Spark MLlib 模块。Spark MLlib 是一种高效、快速、可扩展的分布式计算框架，实现了常用的机器学习，如聚类、分类、回归等算法。本文循序渐进，从 Spark 的基础知识、矩阵向量的基础知识开始，然后再讲解各种算法的理论知识，以及 Spark 源码实现和实例

实战，帮助读者从基础到实践全面掌握 Spark MLlib 分布式机器学习。

学习本书需要的基础知识包括：Spark 基础入门、Scala 入门、线性代数基础知识。

本书面向的读者：Spark 开发者、大数据工程师、数据挖掘工程师、机器学习工程师、研究生和高年级本科生等。

本书学习指南：

<table>
<tr><td>第一部分 Spark MLlib基础</td><td rowspan="4">Spark MLlib机器学习的基础包括：Spark数据操作、矩阵向量，它们都是各个机器学习算法的底层实现基础</td><td rowspan="4">通过这部分掌握：RDD的基础操作、矩阵和向量的运算、数据格式等</td></tr>
<tr><td>第1章 Spark机器学习简介</td></tr>
<tr><td>第2章 Spark数据操作</td></tr>
<tr><td>第3章 Spark MLlib矩阵向量</td></tr>
<tr><td>第二部分 Spark MLlib回归算法</td><td rowspan="16">Spark MLlib机器学习算法的全面解析。包含常见机器学习：回归、分类、聚类、关联、推荐和神经网络</td><td rowspan="16">通过这部分掌握：机器学习算法理论知识、机器学习算法的分布实现方法、MLlib源码解析、实例解析

其中第14、15章是基于Spark MLlib上实现或者定制开发机器学习算法，读者可掌握分布式机器学习的开发

分布式机器学习的学习路径：
理论→分布式实现逻辑→开发→实例</td></tr>
<tr><td>第4章 Spark MLlib线性回归算法</td></tr>
<tr><td>第5章 Spark MLlib逻辑回归算法</td></tr>
<tr><td>第6章 Spark MLlib保序回归算法</td></tr>
<tr><td>第三部分 Spark MLlib分类算法</td></tr>
<tr><td>第7章 Spark MLlib贝叶斯分类算法</td></tr>
<tr><td>第8章 Spark MLlib SVM支持向量机算法</td></tr>
<tr><td>第9章 Spark MLlib决策树算法</td></tr>
<tr><td>第四部分 Spark MLlib聚类算法</td></tr>
<tr><td>第10章 Spark MLlib KMeans聚类算法</td></tr>
<tr><td>第11章 Spark MLlib LDA主题模型算法</td></tr>
<tr><td>第五部分 Spark MLlib关联规则挖掘算法</td></tr>
<tr><td>第12章 Spark MLlib FPGrowth关联规则算法</td></tr>
<tr><td>第六部分 Spark MLlib推荐算法</td></tr>
<tr><td>第13章 Spark MLlib ALS交替最小二乘算法</td></tr>
<tr><td>第14章 Spark MLlib 协同过滤推荐算法</td></tr>
<tr><td>第七部分 Spark MLlib神经网络算法</td></tr>
<tr><td>第15章 Spark MLlib神经网络算法综述</td></tr>
</table>

在本书的编写过程中，何娟、何丹、王蒙、叶月媚参与了全书的编写、整理及校对工作，刘程辉、李俊、廖宏参与了 Spark 集群运维和第 2 章数据操作的实例部分工作，刘晓宏、方佳武、于善龙参与了全书的实例部分工作。

由于笔者水平有限，编写时间仓促，书中难免会出现一些错误或者不准确的地方，恳请读

者批评指正。您也可以通过博客 http://blog.csdn.net/sunbow0、邮箱 humeli317@163.com 和 QQ 群 487540403 联系到我，期待能够得到读者朋友们的真挚反馈，在技术之路上互勉共进。

本书在写作的过程中，得到了很多朋友及同事的帮助和支持，在此表示衷心感谢！

感谢久邦数据大数据团队的同事们。在两年的工作中，笔者得到了很多同事的指导、支持和帮助，尤其感谢杨树清、周小平、梁宁、刘程辉、刘晓宏、方佳武、于善龙、王蒙、叶月媚、廖宏、谭钊承、吴梦玲、邹桂芳、曹越等。

感谢电子工业出版社的付睿编辑，她不仅积极策划和推动本书的出版，而且在写作过程中还给出了极为详细的改进意见。感谢电子工业出版社的李云静编辑为本书做了非常辛苦和专业的编辑工作。

感谢我的父母和妻子，有了你们的帮助和支持，我才有时间和精力去完成写作。

谨以此书献给热爱大数据技术的朋友们！

目录

第一部分　Spark MLlib 基础

第三部分 Spark MLlib 分类算法

第四部分 Spark MLlib 聚类算法

第七部分　Spark MLlib 神经网络算法

第 3 章
Spark MLlib 矩阵向量

Spark MLlib 底层的向量、矩阵运算使用了 Breeze 库，Breeze 库提供了 Vector/Matrix 的实现及相应计算的接口（Linalg）。但是在 MLlib 里面同时也提供了 Vector 和 Linalg 等的实现。在 MLlib 函数里的参数传递均使用 MLlib 自己的 Vector，而且在函数内的矩阵计算又通过 ToBreeze.ToDenseVector 变成 Breeze 的形式进行运算。这样做的目的一是保持自己函数接口的稳定性，不会因为 Breeze 的变化而变化；另外一个就是可以把 Distributed Matrix 作为一种 Matrix 的实现而被使用。

3.1 Breeze 介绍

ScalaNLP 是一套机器学习和数值计算的库。它主要是关于科学计算（SC）、机器学习（ML）和自然语言处理（NLP）的。它包括了三个库：Breeze 、Epic 和 Puck，如图 3-1 所示。Breeze 是机器学习和数值计算库，Epic 是一种高性能统计分析器和结构化预测库，Puck 是一个快速 GPU 加速解析器。

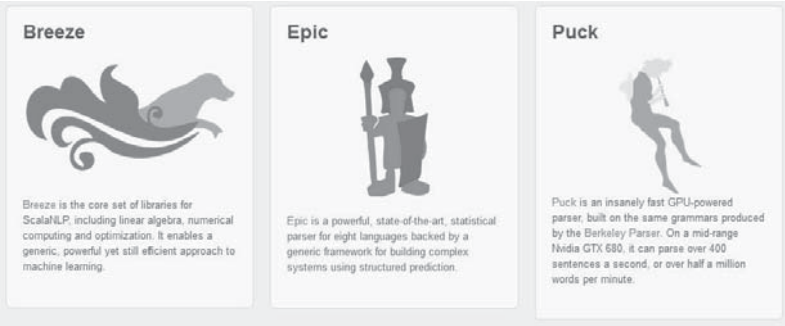

图 3-1　ScalaNLP 组件

Breeze 是 ScalaNLP 的核心，包括线性代数、数值计算及优化，它是一种通用、功能强大、有效的机器学习方法。

在 Spark MLlib 中机器学习算法的底层实现，采用的是 Breeze 库的矩阵、向量、数值计算等函数。

Breeze 的基本函数如下。

3.1.1　Breeze 创建函数

在使用 Breeze 库时，需要导入相关包：

```
import breeze.linalg._
import breeze.numerics._
```

1）Breeze 创建函数如表 3-1 所示。

表 3-1　Breeze创建函数

操作名称	Breeze函数	对应Matlab函数	对应Numpy函数
全0矩阵	DenseMatrix.zeros[Double](n,m)	zeros(n,m)	zeros((n,m))
全0向量	DenseVector.zeros[Double](n)	zeros(n)	zeros(n)
全1向量	DenseVector.ones[Double](n)	ones(n)	ones(n)
按数值填充向量	DenseVector.fill(n){5.0}	ones(n) * 5	ones(n) * 5
生成随机向量	DenseVector.range(start,stop,step)或Vector.rangeD(start,stop,step)		
线性等分向量（用于产生start、stop之间的*N*点行向量）	DenseVector.linspace(start,stop,numvals)	linspace(0,20,15)	
单位矩阵	DenseMatrix.eye[Double](n)	eye(n)	eye(n)
对角矩阵	diag(DenseVector(1.0,2.0,3.0))	diag([1 2 3])	diag((1,2,3))

续表

操作名称	Breeze函数	对应Matlab函数	对应Numpy函数
按照行创建矩阵	DenseMatrix((1.0,2.0), (3.0,4.0))	[1 2; 3 4]	array([[1,2], [3,4]])
按照行创建向量	DenseVector(1,2,3,4)	[1 2 3 4]	array([1,2,3,4])
向量转置	DenseVector(1,2,3,4).t	[1 2 3 4]'	array([1,2,3]). reshape(-1,1)
从函数创建向量	DenseVector.tabulate(3){i => 2*i}		
从函数创建矩阵	DenseMatrix.tabulate(3, 2){case (i, j) => i+j}		
从数组创建向量	new DenseVector(Array(1, 2, 3, 4))		
从数组创建矩阵	new DenseMatrix(2, 3, Array(11, 12, 13, 21, 22, 23))		
0到1的随机向量	DenseVector.rand(4)		
0到1的随机矩阵	DenseMatrix.rand(2, 3)		

2）Breeze 创建函数的实例如下：

```
scala> val m1 = DenseMatrix.zeros[Double](2,3)
m1: breeze.linalg.DenseMatrix[Double] =
0.0  0.0  0.0
0.0  0.0  0.0

scala> val v1 = DenseVector.zeros[Double](3)
v1: breeze.linalg.DenseVector[Double] = DenseVector(0.0, 0.0, 0.0)

scala> val v2 = DenseVector.ones[Double](3)
v2: breeze.linalg.DenseVector[Double] = DenseVector(1.0, 1.0, 1.0)

scala> val v3 = DenseVector.fill(3){5.0}
v3: breeze.linalg.DenseVector[Double] = DenseVector(5.0, 5.0, 5.0)

scala> val v4 = DenseVector.range(1,10,2)
v4: breeze.linalg.DenseVector[Int] = DenseVector(1, 3, 5, 7, 9)

scala> val m2 = DenseMatrix.eye[Double](3)
m2: breeze.linalg.DenseMatrix[Double] =
1.0  0.0  0.0
0.0  1.0  0.0
0.0  0.0  1.0

scala> val v6 = diag(DenseVector(1.0,2.0,3.0))
v6: breeze.linalg.DenseMatrix[Double] =
1.0  0.0  0.0
```

```
0.0  2.0  0.0
0.0  0.0  3.0

scala> val m3 = DenseMatrix((1.0,2.0), (3.0,4.0))
m3: breeze.linalg.DenseMatrix[Double] =
1.0  2.0
3.0  4.0

scala> val v8 = DenseVector(1,2,3,4)
v8: breeze.linalg.DenseVector[Int] = DenseVector(1, 2, 3, 4)

scala> val v9 = DenseVector(1,2,3,4).t
v9: breeze.linalg.Transpose[breeze.linalg.DenseVector[Int]] = Transpose(DenseVector(1, 2, 3, 4))

scala> val v10 = DenseVector.tabulate(3){i => 2*i}
v10: breeze.linalg.DenseVector[Int] = DenseVector(0, 2, 4)

scala> val m4 = DenseMatrix.tabulate(3, 2){case (i, j) => i+j}
m4: breeze.linalg.DenseMatrix[Int] =
0  1
1  2
2  3

scala> val v11 = new DenseVector(Array(1, 2, 3, 4))
v11: breeze.linalg.DenseVector[Int] = DenseVector(1, 2, 3, 4)

scala> val m5 = new DenseMatrix(2, 3, Array(11, 12, 13, 21, 22, 23))
m5: breeze.linalg.DenseMatrix[Int] =
11  13  22
12  21  23

scala> val v12 = DenseVector.rand(4)
v12: breeze.linalg.DenseVector[Double] = DenseVector(0.7517657487447951, 0.8171495400874123,
  0.8923542318540489, 0.174311259949119)

scala> val m6 = DenseMatrix.rand(2, 3)
m6: breeze.linalg.DenseMatrix[Double] =
0.5349430131148125   0.8822136832272578  0.7946323804433382
0.41097756311601086  0.3181490074596882  0.34195102205697414
```

3.1.2　Breeze 元素访问及操作函数

1）Breeze 元素访问函数如表 3-2 所示。

表 3-2　Breeze元素访问函数

操作名称	Breeze函数	对应Matlab函数	对应Numpy函数
指定位置	a(0,1)	a(1,2)	a[0,1]
向量子集	a(1 to 4)或a(1 until 5)或a.slice(1,5)	a(2:5)	a[1:5]
按照指定步长取子集	a(5 to 0 by -1)	a(6:-1:1)	a[5:0:-1]
指定开始位置至结尾	a(1 to -1)	a(2:end)	a[1:]
最后一个元素	a(-1)	a(end)	a[-1]
矩阵指定列	a(::, 2)	a(:,3)	a[:,2]

2）Breeze 元素访问函数的实例如下：

```
scala> val a = DenseVector(1,2,3,4,5,6,7,8,9,10)
a: breeze.linalg.DenseVector[Int] = DenseVector(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> a(0)
res2: Int = 1

scala> a(1 to 4)
res4: breeze.linalg.DenseVector[Int] = DenseVector(2, 3, 4, 5)

scala> a(5 to 0 by -1)
res5: breeze.linalg.DenseVector[Int] = DenseVector(6, 5, 4, 3, 2, 1)

scala> a(1 to -1)
res6: breeze.linalg.DenseVector[Int] = DenseVector(2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> a( -1 )
res7: Int = 10

scala> val m = DenseMatrix((1.0,2.0,3.0), (3.0,4.0,5.0))
m: breeze.linalg.DenseMatrix[Double] =
1.0  2.0  3.0
3.0  4.0  5.0

scala> m(0,1)
res8: Double = 2.0

scala> m(::,1)
res9: breeze.linalg.DenseVector[Double] = DenseVector(2.0, 4.0)
```

3）Breeze 元素操作函数如表 3-3 所示。

表 3-3　Breeze元素操作函数

操作名称	Breeze函数	对应Matlab函数	对应Numpy函数
调整矩阵形状	a.reshape(3, 2)	reshape(a, 3, 2)	a.reshape(3,2)
矩阵转成向量	a.toDenseVector	a(:)	a.flatten()
复制下三角	lowerTriangular(a)	tril(a)	tril(a)
复制上三角	upperTriangular(a)	triu(a)	triu(a)
矩阵复制	a.copy		np.copy(a)
取对象线元素	diag(a)	NA	diagonal(a)(Numpy >= 1.9)
子集赋数值	a(1 to 4) := 5.0	a(2:5) = 5	a[1:4] = 5
子集赋向量	a(1 to 4) := DenseVector(1.0,2.0,3.0)	a(2:5) = [1 2 3]	a[1:4] = array([1,2,3])
矩阵赋值	a(1 to 3,1 to 3) := 5.0	a(2:4,2:4) = 5	a[1:3,1:3] = 5
矩阵列赋值	a(::, 2) := 5.0	a(:,3) = 5	a[:,2] = 5
垂直连接矩阵	DenseMatrix.vertcat(a,b)	[a ; b]	vstack((a,b))
横向连接矩阵	DenseMatrix.horzcat(d,e)	[a , b]	hstack((a,b))
向量连接	DenseVector.vertcat(a,b)	[a b]	concatenate((a,b))

4）Breeze 元素操作函数实例如下：

```
scala> val m = DenseMatrix((1.0,2.0,3.0), (3.0,4.0,5.0))
m: breeze.linalg.DenseMatrix[Double] =
1.0  2.0  3.0
3.0  4.0  5.0

scala> m.reshape(3, 2)
res11: breeze.linalg.DenseMatrix[Double] =
1.0  4.0
3.0  3.0
2.0  5.0

scala>m.toDenseVector
res12: breeze.linalg.DenseVector[Double] = DenseVector(1.0, 3.0, 2.0, 4.0, 3.0, 5.0)

scala> val m = DenseMatrix((1.0,2.0,3.0), (4.0,5.0,6.0) , (7.0,8.0,9.0))
m: breeze.linalg.DenseMatrix[Double] =
1.0  2.0  3.0
4.0  5.0  6.0
7.0  8.0  9.0

scala> val m = DenseMatrix((1.0,2.0,3.0), (4.0,5.0,6.0) , (7.0,8.0,9.0))
m: breeze.linalg.DenseMatrix[Double] =
1.0  2.0  3.0
```

```
4.0  5.0  6.0
7.0  8.0  9.0

scala> lowerTriangular(m)
res19: breeze.linalg.DenseMatrix[Double] =
1.0  0.0  0.0
4.0  5.0  0.0
7.0  8.0  9.0

scala> upperTriangular(m)
res20: breeze.linalg.DenseMatrix[Double] =
1.0  2.0  3.0
0.0  5.0  6.0
0.0  0.0  9.0

scala>  m.copy
res21: breeze.linalg.DenseMatrix[Double] =
1.0  2.0  3.0
4.0  5.0  6.0
7.0  8.0  9.0

scala> diag(m)
res22: breeze.linalg.DenseVector[Double] = DenseVector(1.0, 5.0, 9.0)

scala> m(::, 2) := 5.0
res23: breeze.linalg.DenseVector[Double] = DenseVector(5.0, 5.0, 5.0)

scala> m
res24: breeze.linalg.DenseMatrix[Double] =
1.0  2.0  5.0
4.0  5.0  5.0
7.0  8.0  5.0

scala> m(1 to 2,1 to 2) := 5.0
res32: breeze.linalg.DenseMatrix[Double] =
5.0  5.0
5.0  5.0

scala> m
res33: breeze.linalg.DenseMatrix[Double] =
1.0  2.0  5.0
4.0  5.0  5.0
7.0  5.0  5.0
```

```
scala> val a = DenseVector(1,2,3,4,5,6,7,8,9,10)
a: breeze.linalg.DenseVector[Int] = DenseVector(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> a(1 to 4) := 5
res27: breeze.linalg.DenseVector[Int] = DenseVector(5, 5, 5, 5)

scala> a(1 to 4) := DenseVector(1,2,3,4)
res29: breeze.linalg.DenseVector[Int] = DenseVector(1, 2, 3, 4)

scala> a
res30: breeze.linalg.DenseVector[Int] = DenseVector(1, 1, 2, 3, 4, 6, 7, 8, 9, 10)

scala> val a1 = DenseMatrix((1.0,2.0,3.0), (4.0,5.0,6.0))
a1: breeze.linalg.DenseMatrix[Double] =
1.0  2.0  3.0
4.0  5.0  6.0

scala> val a2 = DenseMatrix((1.0,1.0,1.0), (2.0,2.0,2.0))
a2: breeze.linalg.DenseMatrix[Double] =
1.0  1.0  1.0
2.0  2.0  2.0

scala> DenseMatrix.vertcat(a1,a2)
res34: breeze.linalg.DenseMatrix[Double] =
1.0  2.0  3.0
4.0  5.0  6.0
1.0  1.0  1.0
2.0  2.0  2.0

scala> DenseMatrix.horzcat(a1,a2)
res35: breeze.linalg.DenseMatrix[Double] =
1.0  2.0  3.0  1.0  1.0  1.0
4.0  5.0  6.0  2.0  2.0  2.0

scala> val b1 = DenseVector(1,2,3,4)
b1: breeze.linalg.DenseVector[Int] = DenseVector(1, 2, 3, 4)

scala> val b2 = DenseVector(1,1,1,1)
b2: breeze.linalg.DenseVector[Int] = DenseVector(1, 1, 1, 1)

scala> DenseVector.vertcat(b1,b2)
res36: breeze.linalg.DenseVector[Int] = DenseVector(1, 2, 3, 4, 1, 1, 1, 1)
```

3.1.3 Breeze 数值计算函数

1）Breeze 数值计算函数如表 3-4 所示。

表 3-4 Breeze数值计算函数

操作名称	Breeze函数	对应Matlab函数	对应Numpy函数
元素加法	a + b	a + b	a + b
元素乘法	a :* b	a .* b	a * b
元素除法	a :/ b	a ./ b	a / b
元素比较	a :< b	a < b	a < b
元素相等	a :== b	a == b	a == b
元素追加	a :+= 1.0	a += 1	a += 1
元素追乘	a :*= 2.0	a *= 2	a *= 2
向量点积	a dot b,a.t * b†	dot(a,b)	dot(a,b)
元素最大值	max(a)	max(a)	a.max()
元素最大值及位置	argmax(a)	[v i] = max(a); i	a.argmax()

2）Breeze 数值计算函数实例如下：

```
scala> val a = DenseMatrix((1.0,2.0,3.0), (4.0,5.0,6.0))
a: breeze.linalg.DenseMatrix[Double] =
1.0  2.0  3.0
4.0  5.0  6.0

scala>  val b = DenseMatrix((1.0,1.0,1.0), (2.0,2.0,2.0))
b: breeze.linalg.DenseMatrix[Double] =
1.0  1.0  1.0
2.0  2.0  2.0

scala> a + b
res37: breeze.linalg.DenseMatrix[Double] =
2.0  3.0  4.0
6.0  7.0  8.0

scala> a :* b
res38: breeze.linalg.DenseMatrix[Double] =
1.0  2.0   3.0
8.0  10.0  12.0

scala> a :/ b
res39: breeze.linalg.DenseMatrix[Double] =
1.0  2.0  3.0
2.0  2.5  3.0
```

```
scala> a :< b
res40: breeze.linalg.DenseMatrix[Boolean] =
false  false  false
false  false  false

scala> a :== b
res41: breeze.linalg.DenseMatrix[Boolean] =
true   false  false
false  false  false

scala> a :+= 1.0
res42: breeze.linalg.DenseMatrix[Double] =
2.0  3.0  4.0
5.0  6.0  7.0

scala> a :*= 2.0
res43: breeze.linalg.DenseMatrix[Double] =
4.0   6.0   8.0
10.0  12.0  14.0

scala> max(a)
res47: Double = 14.0

scala> argmax(a)
res48: (Int, Int) = (1,2)

scala> DenseVector(1, 2, 3, 4) dot DenseVector(1, 1, 1, 1)
res50: Int = 10
```

3.1.4　Breeze 求和函数

1）Breeze 求和函数如表 3-5 所示。

表 3-5　Breeze求和函数

操作名称	Breeze函数	对应Matlab函数	对应Numpy函数
元素求和	sum(a)	sum(sum(a))	a.sum()
每一列求和	sum(a, Axis._0)或sum(a(::, *))	sum(a)	sum(a,0)
每一行求和	sum(a, Axis._1)或sum(a(*, ::))	sum(a')	sum(a,1)
对角线元素和	trace(a)	trace(a)	a.trace()
累积和	accumulate(a)	cumsum(a)	a.cumsum()

2）Breeze 求和函数实例如下：

```
scala> val a = DenseMatrix((1.0,2.0,3.0), (4.0,5.0,6.0) , (7.0,8.0,9.0))
a: breeze.linalg.DenseMatrix[Double] =
1.0  2.0  3.0
4.0  5.0  6.0
7.0  8.0  9.0

scala> sum(a)
res51: Double = 45.0

scala> sum(a, Axis._0)
res52: breeze.linalg.DenseMatrix[Double] = 12.0  15.0  18.0

scala> sum(a, Axis._1)
res53: breeze.linalg.DenseVector[Double] = DenseVector(6.0, 15.0, 24.0)

scala> trace(a)
res54: Double = 15.0

scala> accumulate(DenseVector(1, 2, 3, 4))
res56: breeze.linalg.DenseVector[Int] = DenseVector(1, 3, 6, 10)
```

3.1.5 Breeze 布尔函数

1）Breeze 布尔函数如表 3-6 所示。

表 3-6 Breeze布尔函数

操作名称	Breeze函数	对应Matlab函数	对应Numpy函数
元素与操作	a :& b	a && b	a & b
元素或操作	a :| b	a || b	a | b
元素非操作	!a	~a	~a
任意元素非零	any(a)	any(a)	any(a)
所有元素非零	all(a)	all(a)	all(a)

2）Breeze 布尔函数实例如下：

```
scala> val a = DenseVector(true, false, true)
a: breeze.linalg.DenseVector[Boolean] = DenseVector(true, false, true)

scala> val b = DenseVector(false, true, true)
b: breeze.linalg.DenseVector[Boolean] = DenseVector(false, true, true)
```

```
scala> a :& b
res57: breeze.linalg.DenseVector[Boolean] = DenseVector(false, false, true)

scala> a :| b
res58: breeze.linalg.DenseVector[Boolean] = DenseVector(true, true, true)

scala> !a
res59: breeze.linalg.DenseVector[Boolean] = DenseVector(false, true, false)

scala> val a = DenseVector(1.0, 0.0, -2.0)
a: breeze.linalg.DenseVector[Double] = DenseVector(1.0, 0.0, -2.0)

scala> any(a)
res60: Boolean = true

scala> all(a)
res61: Boolean = false
```

3.1.6　Breeze 线性代数函数

1）Breeze 线性代数函数如表 3-7 所示。

表 3-7　Breeze线性代数函数

操作名称	Breeze函数	对应Matlab函数	对应Numpy函数
线性求解	a \ b	a \ b	linalg.solve(a,b)
转置	a.t	a'	a.conj.transpose()
求特征值	det(a)	det(a)	linalg.det(a)
求逆	inv(a)	inv(a)	linalg.inv(a)
求伪逆	pinv(a)	pinv(a)	linalg.pinv(a)
求范数	norm(a)	norm(a)	norm(a)
特征值和特征向量	eigSym(a)	[v,l] = eig(a)	linalg.eig(a)[0]
特征值	val (er, ei, _) = eig(a)（实部与虚部分开）	eig(a)	linalg.eig(a)[0]
特征向量	eig(a)._3	[v,l] = eig(a)	linalg.eig(a)[1]
奇异值分解	val svd.SVD(u,s,v) = svd(a)	svd(a)	linalg.svd(a)
求矩阵的秩	rank(a)	rank(a)	rank(a)
矩阵长度	a.length	size(a)	a.size
矩阵行数	a.rows	size(a,1)	a.shape[0]
矩阵列数	a.cols	size(a,2)	a.shape[1]

2）Breeze 线性代数函数实例如下：

```
scala> val a = DenseMatrix((1.0,2.0,3.0), (4.0,5.0,6.0) , (7.0,8.0,9.0))
a: breeze.linalg.DenseMatrix[Double] =
1.0  2.0  3.0
4.0  5.0  6.0
7.0  8.0  9.0

scala> val b = DenseMatrix((1.0,1.0,1.0), (1.0,1.0,1.0) , (1.0,1.0,1.0))
b: breeze.linalg.DenseMatrix[Double] =
1.0  1.0  1.0
1.0  1.0  1.0
1.0  1.0  1.0

scala> a \ b
res74: breeze.linalg.DenseMatrix[Double] =
-2.5  -2.5  -2.5
4.0   4.0   4.0
-1.5  -1.5  -1.5

scala> a.t
res63: breeze.linalg.DenseMatrix[Double] =
1.0  4.0  7.0
2.0  5.0  8.0
3.0  6.0  9.0

scala> det(a)
res64: Double = 6.661338147750939E-16

scala> inv(a)
res65: breeze.linalg.DenseMatrix[Double] =
-4.503599627370499E15  9.007199254740992E15    -4.503599627370495E15
9.007199254740998E15   -1.8014398509481984E16  9.007199254740991E15
-4.503599627370498E15  9.007199254740992E15    -4.5035996273704955E15

scala> val svd.SVD(u,s,v) = svd(a)
u: breeze.linalg.DenseMatrix[Double] =
-0.21483723836839663  0.8872306883463706   0.4082482904638625
-0.5205873894647373   0.2496439529882974   -0.8164965809277261
-0.8263375405610782   -0.3879427823697744  0.4082482904638633
s: breeze.linalg.DenseVector[Double] = DenseVector(16.84810335261421, 1.06836951455471,
  4.4184247511933675E-16)
v: breeze.linalg.DenseMatrix[Double] =
```

```
-0.4796711778777715  -0.5723677939720624   -0.6650644100663531
-0.7766909903215595  -0.07568647010455853  0.6253180501124426
-0.4082482904638631  0.8164965809277263    -0.4082482904638631

scala> a.rows
res72: Int = 3

scala> a.cols
res73: Int = 3
```

3.1.7　Breeze 取整函数

1）Breeze 取整函数如表 3-8 所示。

表 3-8　Breeze 取整函数

操作名称	Breeze函数	对应Matlab函数	对应Numpy函数
四舍五入	round(a)	round(a)	around(a)
最小整数	ceil(a)	ceil(a)	ceil(a)
最大整数	floor(a)	floor(a)	floor(a)
符号函数	signum(a)	sign(a)	sign(a)
取正数	abs(a)	abs(a)	abs(a)

2）Breeze 取整函数实例如下：

```
scala> val a = DenseVector(1.2, 0.6, -2.3)
a: breeze.linalg.DenseVector[Double] = DenseVector(1.2, 0.6, -2.3)

scala> round(a)
res75: breeze.linalg.DenseVector[Long] = DenseVector(1, 1, -2)

scala> ceil(a)
res76: breeze.linalg.DenseVector[Double] = DenseVector(2.0, 1.0, -2.0)

scala> floor(a)
res77: breeze.linalg.DenseVector[Double] = DenseVector(1.0, 0.0, -3.0)

scala> signum(a)
res78: breeze.linalg.DenseVector[Double] = DenseVector(1.0, 1.0, -1.0)

scala> abs(a)
res79: breeze.linalg.DenseVector[Double] = DenseVector(1.2, 0.6, 2.3)
```

3.1.8 Breeze 常量函数

Breeze 常量函数如表 3-9 所示。

表 3-9 Breeze常量函数

操作名称	Breeze函数	对应Matlab函数	对应Numpy函数
非数	NaN或者nan	NaN	nan
无穷大量	Inf或者inf	Inf	inf
圆周率π	Constants.Pi	pi	math.pi
以e为底的指数函数	Constants.E	exp(1)	math.e

3.1.9 Breeze 复数函数

Breeze 复数函数如表 3-10 所示。

表 3-10 Breeze复数函数

操作名称	Breeze函数	对应Matlab函数	对应Numpy函数
虚数单位	i	i	z = 1j
复数	3 + 4 * i或Complex(3,4)	3 + 4i	z = 3 + 4j
绝对值	abs(z)或z.abs	abs(z)	abs(z)
实数	z.real	real(z)	z.real
虚数	z.conjugate	conj(z)	z.conj()或z.conjugate()

3.1.10 Breeze 三角函数

Breeze 三角函数包括：

- sin，sinh，asin，asinh
- cos，cosh，acos，acosh
- tan，tanh，atan，atanh
- atan2
- sinc(x)，即 sin(x)/x
- sincpi(x)，即 sinc(x * Pi)

3.1.11 Breeze 对数和指数函数

Breeze 对数和指数函数包括：

- log，exp log10

- log1p，expm1
- sqrt，sbrt
- pow

3.2　BLAS 介绍

BLAS（Basic Linear Algebra Subprograms，基础线性代数程序集）是一个应用程序接口（API）标准，用以规范发布基础线性代数操作的数值库（如向量或矩阵乘法）。该程序集最初发布于 1979 年，并用于创建更大的数值程序包（如 LAPACK）。在高性能计算领域，BLAS 被广泛使用。例如，LINPACK 的运算成绩很大程度上取决于 BLAS 中子程序 DGEMM 的表现。为提高性能，各软硬件厂商针对其产品对 BLAS 接口实现进行高度最优化。

BLAS 按照功能被分为三个级别。

- Level 1：向量-向量运算，比如点积（ddot）、加法和数乘（daxpy），绝对值的和（dasum）等。

$$\boldsymbol{y} \leftarrow \alpha \boldsymbol{x} + \boldsymbol{y}$$

- Level 2：矩阵-向量运算，最重要的函数是一般的矩阵向量乘法（dgemv）。

$$\boldsymbol{y} \leftarrow \alpha \boldsymbol{A}\boldsymbol{x} + \beta \boldsymbol{y}$$

- Level 3：矩阵-矩阵运算，最重要的函数是一般的矩阵乘法（dgemm）。

$$\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}\boldsymbol{B} + \beta \boldsymbol{C}$$

每一种函数操作都区分不同数据类型（单精度、双精度、复数）：比如矩阵乘法分为 sgemm（单精度一般矩阵乘法）、dsymm（双精度对称矩阵乘法）、zhemm（双精度复数埃米特矩阵乘法）。之所以要分成这么多种，主要是针对每种不同类型的矩阵都要分别设计专门的算法，使得计算性能最优。

在 Spark MLlib 中，采用了 BLAS 线性代数运算库，BLAS 的基本运算函数如表 3-11 所示。

表 3-11　BLAS基本运算函数

函数	名称
点积	dot
常数乘以向量加另一个向量	axpy
准备Givens旋转	rotg
实施旋转	rot

第 10 章
Spark MLlib KMeans 聚类算法

10.1 KMeans 聚类算法

10.1.1 KMeans 算法

KMeans 算法的基本思想是初始随机给定 K 个簇中心，按照最邻近原则把待分类样本点分到各个簇。然后按平均法重新计算各个簇的质心，从而确定新的簇心。一直迭代，直到簇心的移动距离小于某个给定的值。

KMeans 聚类算法主要分为 3 个步骤。

1）第 1 步是为待聚类的点寻找聚类中心；

2）第 2 步是计算每个点到聚类中心的距离，将每个点聚类到离该点最近的聚类中去；

3）第 3 步是计算每个聚类中所有点的坐标平均值，并将这个平均值作为新的聚类中心。

反复执行 2）、3），直到聚类中心不再进行大范围移动或者聚类次数达到要求为止。

10.1.2　演示 KMeans 算法

图 10-1 展示了对 n 个样本点进行 KMeans 聚类的效果，这里 k 取 2。

（a）未聚类的初始点集；

（b）随机选取两个点作为聚类中心；

（c）计算每个点到聚类中心的距离，并聚类到离该点最近的聚类中去；

（d）计算每个聚类中所有点的坐标平均值，并将这个平均值作为新的聚类中心；

（e）重复（c），计算每个点到聚类中心的距离，并聚类到离该点最近的聚类中去；

（f）重复（d），计算每个聚类中所有点的坐标平均值，并将这个平均值作为新的聚类中心。

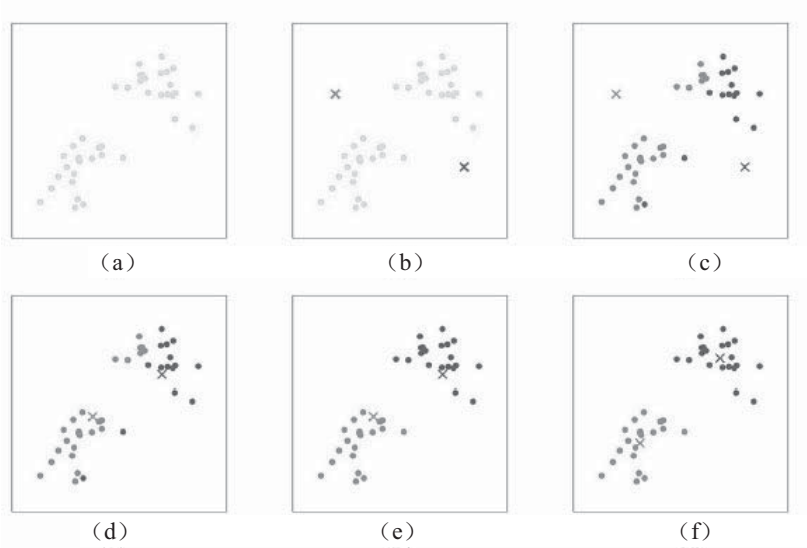

图 10-1　KMeans 算法演示

10.1.3　初始化聚类中心点

对于初始化聚类中心点，我们可以在输入的数据集中随机地选择 k 个点作为中心点，但是随机选择初始中心点可能会造成聚类的结果和数据的实际分布相差很大。k-means++就是选择初始中心点的一种算法。

k-means++算法选择初始中心点的基本思想就是：初始的聚类中心之间的相互距离要尽可能

远。初始化过程如下。

1）从输入的数据点集合中随机选择一个点作为第一个聚类中心；

2）对于数据集中的每一个点 x，计算它与最近聚类中心（指已选择的聚类中心）的距离 $D(x)$；

3）选择一个新的数据点作为新的聚类中心，选择的原则是：$D(x)$较大的点，被选取作为聚类中心的概率较大；

4）重复 2）和 3），直到 k 个聚类中心被选出来；

5）利用这 k 个初始的聚类中心来运行标准的 KMeans 算法。

从上面的算法描述可以看到，算法的关键是第 3 步，如何将 $D(x)$反映到点被选择的概率上。一种算法如下。

1）随机从点集 D 中选择一个点作为初始的中心点。

2）计算每一个点到最近中心点的距离 S_i，对所有 S_i 求和得到 sum。

3）然后再取一个随机值，用权重的方式计算下一个“种子点”。取随机值 random（0<random<sum），对点集 D 循环，做 random $-=S_i$ 运算，直到 random < 0，那么点 i 就是下一个中心点。

4）重复 2）和 3），直到 k 个聚类中心被选出来。

5）利用这 k 个初始的聚类中心来运行标准的 KMeans 算法。

10.2 源码分析

MLlib 的 KMeans 聚类模型采用 KMeans 算法来计算聚类中心点。MLlib 实现 KMeans 聚类算法：首先随机生成聚类中心点，支持随机选择样本点当作初始中心点，还支持 k-means++方法选择最优的聚类中心点。然后迭代计算样本的中心点，迭代计算中心点的分布式实现是：首先计算每个样本属于哪个中心点，之后采用聚合函数统计属于每个中心点的样本值之和以及样本数量，最后求得最新中心点，并且判断中心点是否发生改变。

MLlib 的 KMeans 聚类模型的 runs 参数可以设置并行计算聚类中心的数量，runs 代表同时计算多组聚类中心点，最后取计算结果最好的那一组中心点作为聚类中心点。

MLlib 的 KMeans 聚类模型对于计算样本属于哪个中心点，采用了一种快速查找、计算距离的方法，其方法如下。

首先定义 lowerBoundOfSqDist 距离公式，假设中心点 center 是 (a_1,b_1)，需要计算的点 point 是 (a_2,b_2)，那么 lowerBoundOfSqDist 是：

$$\begin{aligned}\text{lowerBoundOfSqDist} &= (\sqrt{a_1^2+b_1^2}-\sqrt{a_2^2+b_2^2})^2 \\ &= a_1^2+b_1^2+a_2^2+b_2^2-2\sqrt{(a_1^2+b_1^2)(a_2^2+b_2^2)}\end{aligned}$$

对比欧氏距离：

$$\begin{aligned}\text{EuclideanDist} &= (a_1-a_2)^2+(b_1-b_2)^2 \\ &= a_1^2+b_1^2+a_2^2+b_2^2-2(a_1a_2+b_1b_2)\end{aligned}$$

可轻易证明 lowerBoundOfSqDist 将会小于或等于 EuclideanDist，因此在进行距离比较的时候，先计算很容易计算的 lowerBoundOfSqDist（只需要计算 center、point 的 L2 范数）。如果 lowerBoundOfSqDist 都不小于之前计算得到的最小距离 bestDistance，那真正的欧氏距离也不可能小于 bestDistance 了。因此在这种情况下就不需要去计算欧氏距离了，省去了很多计算工作。

如果 lowerBoundOfSqDist 小于 bestDistance，则进行距离的计算，调用 fastSquaredDistance，该方法是一种快速计算距离的方法。fastSquaredDistance 方法会先计算一个精度，有关精度的计算：precisionBound1 = 2.0 * EPSILON * sumSquaredNorm / (normDiff * normDiff + EPSILON)。如果精度满足条件，则欧氏距离为 EuclideanDist = sumSquaredNorm - 2.0 * v1.dot(*v2*)，其中 sumSquaredNorm 为 $a_1^2+b_1^2+a_2^2+b_2^2$，2.0 * v1.dot(v2)为 $2(a_1a_2+b_1b_2)$。这里可以直接利用之前计算的 L2 范数。如果精度不满足要求，则进行原始的距离计算，公式为 $(a_1-a_2)^2+(b_1-b_2)^2$。

MLlib 的 KMeans 聚类模型的源码分解参见表 10-1，具体源码解析参见下面各节。

表 10-1　KMeans聚类模型的源码分解

KMeans源码分解说明		
1. KMeans聚类伴生对象	KMeans	KMeans对象是基于随机梯度下降的SVM分类的伴生对象
1.1　train静态方法	train	train是KMeans对象的静态方法。该方法根据设置KMeans聚类参数，新建KMeans聚类，并执行run方法进行训练
2. KMeans聚类类	KMeans	KMeans类
2.1　run方法	run	run是KMeans类方法，该方法主要用runAlgorithm方法进行聚类中心点的计算

续表

KMeans源码分解说明		
3. 聚类中心点计算	runAlgorithm	runAlgorithm方法
3.1 初始化中心	initRandom	初始化中心点方法支持随机选择中心点和k-means++方法生成中心点
3.2 迭代计算更新中心	iteration	迭代计算样本属于哪个计算中心点，并更新最新中心点
4. KMeans聚类模型	KMeansModel	KMeansModel类
4.1 预测计算	predict	预测样本属于哪个类

10.2.1 建立 KMeans 聚类

1. KMeans 聚类伴生对象

KMeans 聚类伴生对象是 object KMeans，该对象是建立 KMeans 聚类模型的入口，其主要定义训练 KMeans 聚类模型的 train 方法。train 方法通过设置训练参数进行模型训练，其参数主要包括：

- data——数据样本，格式为 RDD[Vector]。
- k——聚类数量。
- maxIterations——最大迭代次数。
- runs——并行计算数，默认为 1。可根据计算资源，设置为大于 1 的整数，结果最终会返回 1 个最佳模型。
- initializationMode——初始化中心，支持 random 或者 k-means++（默认）。
- seed——初始化时的随机种子。

KMeans 伴生对象的源码解析如下。

```
object KMeans {

  val RANDOM = "random"
  val K_MEANS_PARALLEL = "k-means||"

  /**
   * 根据给定的参数集训练一个 KMeans 模型
   *
   * @param data 数据样本，格式为 RDD[Vector]
   * @param k 聚类数量
   * @param maxIterations 最大迭代次数
   * @param runs 并行计算数，默认为 1。返回最佳模型
   * @param initializationMode 初始化中心，支持 random 或者 k-means++（默认）
```

```
     * @param seed 初始化时的随机种子
     */
    def train(
        data: RDD[Vector],
        k: Int,
        maxIterations: Int,
        runs: Int,
        initializationMode: String,
        seed: Long): KMeansModel = {
      new KMeans().setK(k)
        .setMaxIterations(maxIterations)
        .setRuns(runs)
        .setInitializationMode(initializationMode)
        .setSeed(seed)
        .run(data)
    }

    /**
     * 根据给定的参数集训练一个 KMeans 模型
     *
     * @param data 数据样本，格式为 RDD[Vector]
     * @param k 聚类数量
     * @param maxIterations 最大迭代次数
     * @param runs 并行计算数，默认为 1。返回最佳模型
     * @param initializationMode 初始化中心，支持 random 或者 k-means++（默认）
     */
    def train(
        data: RDD[Vector],
        k: Int,
        maxIterations: Int,
        runs: Int,
        initializationMode: String): KMeansModel = {
      new KMeans().setK(k)
        .setMaxIterations(maxIterations)
        .setRuns(runs)
        .setInitializationMode(initializationMode)
        .run(data)
    }

    /**
     *根据给定的参数及默认参数训练一个 KMeans 模型
     */
    def train(
```

```
      data: RDD[Vector],
      k: Int,
      maxIterations: Int): KMeansModel = {
    train(data, k, maxIterations, 1, K_MEANS_PARALLEL)
  }

  /**
   *根据给定的参数及默认参数训练一个 KMeans 模型
   */
  def train(
      data: RDD[Vector],
      k: Int,
      maxIterations: Int,
      runs: Int): KMeansModel = {
    train(data, k, maxIterations, runs, K_MEANS_PARALLEL)
  }
}
```

2. KMeans 聚类类

KMeans 聚类类是 class KMeans，该类主要根据初始化参数运行 run 方法开始训练模型。

KMeans 类的源码解析如下。

```
/**
 * KMeans 聚类算法，支持并行计算及 k-means++的初始化算法
 * 这是一个迭代算法，样本数据 RDD 应该被缓存
 * @param k 聚类个数
 * @param maxIterations 迭代次数
 * @param runs 并行度
 * @param initializationMode 初始中心算法
 * @param initializationSteps 初始步长
 * @param epsilon 中心距离阈值
 * @param seed 随机种子
 *
 */
class KMeans private (
    private var k: Int,
    private var maxIterations: Int,
    private var runs: Int,
    private var initializationMode: String,
    private var initializationSteps: Int,
    private var epsilon: Double,
    private var seed: Long) extends Serializable with Logging {
```

```
/**
 * 构建 KMeans 实例的默认参数：{k: 2, maxIterations: 20, runs: 1,
 * initializationMode: "k-means||", initializationSteps: 5, epsilon: 1e-4, seed: random}
 */
def this() = this(2, 20, 1, KMeans.K_MEANS_PARALLEL, 5, 1e-4, Utils.random.nextLong())

/**
 * 聚类个数
 */
def getK: Int = k

/** 设置聚类个数。默认：2 */
def setK(k: Int): this.type = {
  this.k = k
  this
}

/**
 * 最大迭代次数
 */
def getMaxIterations: Int = maxIterations

/** 设置最大迭代次数。默认：20 */
def setMaxIterations(maxIterations: Int): this.type = {
  this.maxIterations = maxIterations
  this
}

/**
 * 初始中心算法。支持 random 或者 k-means++
 */
def getInitializationMode: String = initializationMode

/**
 * 设置初始中心算法。可以选择 random 模式，选择随机点来初始化中心，
 * 也可以选择 k-means++模式，参照：
 * (Bahmani et al., Scalable K-Means++, VLDB 2012)。默认：k-means++
 */
def setInitializationMode(initializationMode: String): this.type = {
  if (initializationMode != KMeans.RANDOM && initializationMode != KMeans.K_MEANS_PARALLEL) {
    throw new IllegalArgumentException("Invalid initialization mode: " + initializationMode)
  }
  this.initializationMode = initializationMode
```

```
    this
  }

  /**
   * :: Experimental ::
   * 并行执行的并行算法的数量
   */
  @Experimental
  def getRuns: Int = runs

  /**
   * :: Experimental ::
   * 设置并行计算的数量。默认：1
   */
  @Experimental
  def setRuns(runs: Int): this.type = {
    if (runs <= 0) {
      throw new IllegalArgumentException("Number of runs must be positive")
    }
    this.runs = runs
    this
  }

  /**
   * 初始化时的初始步长
   */
  def getInitializationSteps: Int = initializationSteps

  /**
   * 设置初始步长，用于 k-means++初始化模型. 默认：5
   */
  def setInitializationSteps(initializationSteps: Int): this.type = {
    if (initializationSteps <= 0) {
      throw new IllegalArgumentException("Number of initialization steps must be positive")
    }
    this.initializationSteps = initializationSteps
    this
  }

  /**
   * 中心距离阈值
   */
  def getEpsilon: Double = epsilon
```

```
  /**
   * 设置中心距离阈值
   * 如果所有的中心移动距离小于这个距离阈值，我们将停止迭代运行
   */
  def setEpsilon(epsilon: Double): this.type = {
    this.epsilon = epsilon
    this
  }

  /**
   * 聚类初始化的随机种子
   */
  def getSeed: Long = seed

  /** 设置聚类初始化的随机种子 */
  def setSeed(seed: Long): this.type = {
    this.seed = seed
    this
  }
训练代码，将在 10.2.2 节中解析
}
```

10.2.2　模型训练 run 方法

KMeans 类的 run 方法开始训练模型，该方法主要用 runAlgorithm 方法进行中心点的计算。

KMeans 类的 run 方法源码解析如下。

```
/**
   * 根据给定的训练数据开始训练模型，其中数据应该缓存（因为要进行迭代计算）
   */
  def run(data: RDD[Vector]): KMeansModel = {

    if (data.getStorageLevel == StorageLevel.NONE) {
      logWarning("The input data is not directly cached, which may hurt performance if its"
        + " parent RDDs are also uncached.")
    }

    // 计算 L2 范数，并且缓存。zippedData 格式为（向量，向量的 L2 范数）
    val norms = data.map(Vectors.norm(_, 2.0))
    norms.persist()
    val zippedData = data.zip(norms).map { case (v, norm) =>
```

```
        new VectorWithNorm(v, norm)
      }
      // 运行 KMeans 算法
      val model = runAlgorithm(zippedData)
      norms.unpersist()

      // 数据检测警告
      if (data.getStorageLevel == StorageLevel.NONE) {
        logWarning("The input data was not directly cached, which may hurt performance if its"
          + " parent RDDs are also uncached.")
      }
      model
    }
```

10.2.3 聚类中心点计算

KMeans 类中的 runAlgorithm 方法是 KMeans 聚类模型实现的核心。

首先初始化中心，然后 KMeans 迭代执行，计算每个样本属于哪个中心点、中心点累加的样本值及计数，之后根据中心点的所有样本数据进行中心点的更新，并比较更新前的数值，判断是否完成。

聚类中心点计算采用并行计算，也就是分成多组（runs）中心点同时进行计算，最终选择最好的一组作为聚类中心点。其中 runs 代表并行度。

KMeans 类的 runAlgorithm 方法源码解析如下。

```
  /**
   * KMeans 算法的实现
   */
  private def runAlgorithm(data: RDD[VectorWithNorm]): KMeansModel = {

    val sc = data.sparkContext

    val initStartTime = System.nanoTime()

    // 初始化中心，支持 random 或者 k-means++
    val centers = if (initializationMode == KMeans.RANDOM) {
      initRandom(data)
    } else {
      initKMeansParallel(data)
    }
```

```
val initTimeInSeconds = (System.nanoTime() - initStartTime) / 1e9
logInfo(s"Initialization with $initializationMode took " + "%.3f".format
  (initTimeInSeconds) + "seconds.")

val active = Array.fill(runs)(true)
val costs = Array.fill(runs)(0.0)

var activeRuns = new ArrayBuffer[Int] ++ (0 until runs)
var iteration = 0

val iterationStartTime = System.nanoTime()

// KMeans 迭代执行，计算每个样本属于哪个中心点、中心点累加的样本值及计数，
// 然后根据中心点的所有样本数据进行中心点的更新，并比较更新前的数值，判断是否完成
// 其中，runs 代表并行度
while (iteration < maxIterations && !activeRuns.isEmpty) {
// 迭代次数小于最大迭代次数，并行计算的中心点还没有收敛

  type WeightedPoint = (Vector, Long)

  def mergeContribs(x: WeightedPoint, y: WeightedPoint): WeightedPoint = {
    axpy(1.0, x._1, y._1) // y._1 + x._1
    (y._1, x._2 + y._2) // y._1 + x._1 , x._2 + y._2
  }

  // 各个并行计算下的聚类中心点
  val activeCenters = activeRuns.map(r => centers(r)).toArray
  // 各个并行计算下的 cost 值之和
  val costAccums = activeRuns.map(_ => sc.accumulator(0.0))

  // 广播聚类中心点
  val bcActiveCenters = sc.broadcast(activeCenters)

  // 计算属于每个中心点的样本，对每个中心点的样本进行累加和计算
  // runs 代表并行度，k 代表中心点个数，sums 代表中心点样本累加值，counts 代表中心点样本计数
  // contribs 代表（(并行度 I，中心 J)，(中心 J 样本之和，中心 J 样本个数)）
  // findClosest 方法：找到点与所有聚类中心最近的一个中心
  val totalContribs = data.mapPartitions { points =>
    val thisActiveCenters = bcActiveCenters.value // 当前聚类中心
    val runs = thisActiveCenters.length //并行计算数量
    val k = thisActiveCenters(0).length // 聚类数量
    val dims = thisActiveCenters(0)(0).vector.size // 中心点的维度
```

```
        val sums = Array.fill(runs, k)(Vectors.zeros(dims))
        // 计算每一个中心点下的所有样本的向量之和，其中 runs 表示并行度
        val counts = Array.fill(runs, k)(0L)
        // 计算每一个中心点下所有样本的数量，其中 runs 表示并行度

        points.foreach { point => // 针对每一个样本
          (0 until runs).foreach { i => // 并行计算
            val (bestCenter, cost) = KMeans.findClosest(thisActiveCenters(i), point)
            // 计算样本 point 属于哪个中心点，以及在该中心点下的 cost 值
            costAccums(i) += cost // 并行度 i 下的 cost 之和
            val sum = sums(i)(bestCenter)
            axpy(1.0, point.vector, sum) // sum = sum + point
            counts(i)(bestCenter) += 1 // counts(i)(bestCenter)的计数
          }
        }

        // 每一个聚类下样本的向量和、样本点的数量和，i 为并行度，j 为聚类中心
        val contribs = for (i <- 0 until runs; j <- 0 until k) yield {
          ((i, j), (sums(i)(j), counts(i)(j)))
        }
        contribs.iterator
      }.reduceByKey(mergeContribs).collectAsMap() // 聚合操作，对于相同（i，j），其中 i 为并行度，
      // j 为聚类中心，对属于同一中心点下的样本向量之和、样本数量进行累加操作

      // 更新中心点，更新中心点= sum/count
      // 判断 newCenter 与 centers 之间的距离是否 > epsilon * epsilon
      for ((run, i) <- activeRuns.zipWithIndex) {
        var changed = false
        var j = 0
        while (j < k) {
          val (sum, count) = totalContribs((i, j))
          if (count != 0) {
            scal(1.0 / count, sum) // sum = sum/count
            val newCenter = new VectorWithNorm(sum)
            if (KMeans.fastSquaredDistance(newCenter, centers(run)(j)) > epsilon * epsilon)
              { // 计算中心点之间的移动距离
              changed = true // 移动距离大于 epsilon * epsilon，则认为中心点位置发生改变了
            }
            centers(run)(j) = newCenter // 更新中心点
          }
          j += 1
        }
        if (!changed) {
```

```
          active(run) = false // 所有中心点的位置不再改变了
          logInfo("Run " + run + " finished in " + (iteration + 1) + " iterations")
        }
        costs(run) = costAccums(i).value
      }

      activeRuns = activeRuns.filter(active(_)) // 过滤中心点已完成收敛的并行计算
      iteration += 1
    }

    val iterationTimeInSeconds = (System.nanoTime() - iterationStartTime) / 1e9
    logInfo(s"Iterations took " + "%.3f".format(iterationTimeInSeconds) + " seconds.")

    if (iteration == maxIterations) {
      logInfo(s"KMeans reached the max number of iterations: $maxIterations.")
    } else {
      logInfo(s"KMeans converged in $iteration iterations.")
    }

    val (minCost, bestRun) = costs.zipWithIndex.min

    logInfo(s"The cost for the best run is $minCost.")

    new KMeansModel(centers(bestRun).map(_.vector))
  }
```

10.2.4　中心点初始化

在 runAlgorithm 方法中，初始化样本中心点，目前支持采用 k-means++及随机初始化两种。

- 随机初始化，就是随机抽取样本数据作为中心点。
- k-means++算法选择初始中心点的基本思想就是：初始的聚类中心之间的相互距离要尽可能远。

initRandom 和 initKMeansParallel 方法的源码解析如下。

```
  /**
   * 初始化，随机选择中心点
   */
  private def initRandom(data: RDD[VectorWithNorm])
  : Array[Array[VectorWithNorm]] = {
    // 从数据样本中，随机抽取数据作为中心点，runs 是并行度，k 是聚类中心数
    val sample = data.takeSample(true, runs * k, new XORShiftRandom(this.seed).nextInt()).toSeq
```

```
      Array.tabulate(runs)(r => sample.slice(r * k, (r + 1) * k).map { v =>
        new VectorWithNorm(Vectors.dense(v.vector.toArray), v.norm)
        // 返回（中心向量，中心向量 L2 范数）
      }.toArray)
    }

    /**
     * 初始化样本中心点，采用 k-means++算法
     * 参照：(Bahmani et al., Scalable K-Means++, VLDB 2012)
     *
     * 原始论文在 http://theory.stanford.edu/~sergei/papers/vldb12-kmpar.pdf
     */
    private def initKMeansParallel(data: RDD[VectorWithNorm])
    : Array[Array[VectorWithNorm]] = {
      // 初始化中心及 costs
      val centers = Array.tabulate(runs)(r => ArrayBuffer.empty[VectorWithNorm])
      var costs = data.map(_ => Vectors.dense(Array.fill(runs)(Double.PositiveInfinity))).
        cache()

      // 初始化第一个中心点
      val seed = new XORShiftRandom(this.seed).nextInt()
      val sample = data.takeSample(true, runs, seed).toSeq
      val newCenters = Array.tabulate(runs)(r => ArrayBuffer(sample(r).toDense))

      /** 合并新的中心到中心 */
      def mergeNewCenters(): Unit = {
        var r = 0
        while (r < runs) {
          centers(r) ++= newCenters(r)
          newCenters(r).clear()
          r += 1
        }
      }

      // 在每次迭代中，抽样 2 * k 个样本进行计算
      // 注意：每次迭代中计算样本点与中心点之间的距离
      var step = 0
      while (step < initializationSteps) {
        val bcNewCenters = data.context.broadcast(newCenters) // 新中心点
        val preCosts = costs // 前 costs
        // costs 计算
        costs = data.zip(preCosts).map { case (point, cost) =>
          Vectors.dense(
```

```
            Array.tabulate(runs) { r =>
              math.min(KMeans.pointCost(bcNewCenters.value(r), point), cost(r))
            })
        }.cache()
        val sumCosts = costs
          .aggregate(Vectors.zeros(runs))(
            seqOp = (s, v) => {
              // s += v
              axpy(1.0, v, s)
              s
            },
            combOp = (s0, s1) => {
              // s0 += s1
              axpy(1.0, s1, s0)
              s0
            }
          )
        preCosts.unpersist(blocking = false)
        // 选择中心点
        val chosen = data.zip(costs).mapPartitionsWithIndex { (index, pointsWithCosts) =>
          val rand = new XORShiftRandom(seed ^ (step << 16) ^ index)
          pointsWithCosts.flatMap { case (p, c) =>
            val rs = (0 until runs).filter { r =>
              rand.nextDouble() < 2.0 * c(r) * k / sumCosts(r)
            }
            if (rs.length > 0) Some(p, rs) else None
          }
        }.collect()
        mergeNewCenters()
        chosen.foreach { case (p, rs) =>
          rs.foreach(newCenters(_) += p.toDense)
        }
        step += 1
      }

      mergeNewCenters()
      costs.unpersist(blocking = false)

      // 最后，可能会有超过 k 个的候选中心点，通过候选中心点对应的样本并且运行本地 k-means++来选择 k
      // 个中心点
      val bcCenters = data.context.broadcast(centers)
      val weightMap = data.flatMap { p =>
        Iterator.tabulate(runs) { r =>
```

```
          ((r, KMeans.findClosest(bcCenters.value(r), p)._1), 1.0)
        }
      }.reduceByKey(_ + _).collectAsMap()
      val finalCenters = (0 until runs).par.map { r =>
        val myCenters = centers(r).toArray
        val myWeights = (0 until myCenters.length).map(i => weightMap.getOrElse((r, i),
          0.0)).toArray
        LocalKMeans.kMeansPlusPlus(r, myCenters, myWeights, k, 30)
      }

      finalCenters.toArray
    }
  }
```

10.2.5 快速距离计算

KMeans 伴生对象中定义了 findClosest 方法，该方法是一种快速找到点与所有聚类中心最近的一个中心的方法。

KMeans 伴生对象中定义了 fastSquaredDistance 方法，该方法是一种快速计算两点之间距离的方法。其中调用了 2.3.1 节中的 MLUtils.fastSquaredDistance 方法，参见 2.3.1 节的解析。

KMeans 伴生对象的 findClosest 和 fastSquaredDistance 方法的源码解析如下。

```
  /**
    * findClosest 方法：找到点与所有聚类中心最近的一个中心
    */
   private[mllib] def findClosest(
      centers: TraversableOnce[VectorWithNorm],
      point: VectorWithNorm): (Int, Double) = {
    var bestDistance = Double.PositiveInfinity
    var bestIndex = 0
    var i = 0
    centers.foreach { center =>
      var lowerBoundOfSqDist = center.norm - point.norm
      lowerBoundOfSqDist = lowerBoundOfSqDist * lowerBoundOfSqDist
      if (lowerBoundOfSqDist < bestDistance) {
        val distance: Double = fastSquaredDistance(center, point)
        if (distance < bestDistance) {
          bestDistance = distance
          bestIndex = i
        }
      }
```

```
        i += 1
      }
      (bestIndex, bestDistance)
    }

    /**
     * 计算样本点与中心点之间的 cost
     */
    private[mllib] def pointCost(
        centers: TraversableOnce[VectorWithNorm],
        point: VectorWithNorm): Double =
      findClosest(centers, point)._2

    /**
     * 快速计算向量之间的距离
     * [[org.apache.spark.mllib.util.MLUtils#fastSquaredDistance]].
     */
    private[clustering] def fastSquaredDistance(
        v1: VectorWithNorm,
        v2: VectorWithNorm): Double = {
      MLUtils.fastSquaredDistance(v1.vector, v1.norm, v2.vector, v2.norm)
    }
  }

  /**
   * 对于快速计算，自定义向量格式：（向量，向量 L2 范数） *
   * @see [[org.apache.spark.mllib.clustering.KMeans#fastSquaredDistance]]
   */
  private[clustering]
  class VectorWithNorm(val vector: Vector, val norm: Double) extends Serializable {

    def this(vector: Vector) = this(vector, Vectors.norm(vector, 2.0))

    def this(array: Array[Double]) = this(Vectors.dense(array))

    /**转换成密集向量*/
    def toDense: VectorWithNorm = new VectorWithNorm(Vectors.dense(vector.toArray), norm)
  }
```

10.2.6　KMeans 聚类模型

训练完成后，生成 KMeans 聚类模型，KMeans 聚类模型包含参数：中心点向量，KMeans

聚类模型包含方法：预测、保存模型、加载模型等。

KMeansModel 类中定义了 predict 方法，predict 支持 RDD 格式及向量格式的数据计算。

KMeansModel.predict 方法中调用了 KMeans 对象的 findClosest 方法。

KMeansModel 中 predict 方法的源码解析如下。

```
/**
 * KMeans 聚类模型。按照最邻近原则把待分类样本点分到各个簇
 */
class KMeansModel (
    val clusterCenters: Array[Vector]) extends Saveable with Serializable with PMMLExportable {

  /** java 接口的构造函数 */
  def this(centers: java.lang.Iterable[Vector]) = this(centers.asScala.toArray)

  /** 聚类个数 */
  def k: Int = clusterCenters.length

  /** 预测样本点属于哪个类 */
  def predict(point: Vector): Int = {
    KMeans.findClosest(clusterCentersWithNorm, new VectorWithNorm(point))._1
  }

  /**预测样本 RDD 的分类 */
  def predict(points: RDD[Vector]): RDD[Int] = {
    val centersWithNorm = clusterCentersWithNorm
    val bcCentersWithNorm = points.context.broadcast(centersWithNorm)
    points.map(p => KMeans.findClosest(bcCentersWithNorm.value, new VectorWithNorm(p))._1)
  }

  /** 预测样本的分类，JavaRDD 数据类型 */
  def predict(points: JavaRDD[Vector]): JavaRDD[java.lang.Integer] =
    predict(points.rdd).toJavaRDD().asInstanceOf[JavaRDD[java.lang.Integer]]

  /**
   * 返回 KMeans 的 cost，计算样本点到最近中心点的平方距离之和
   *
   */
  def computeCost(data: RDD[Vector]): Double = {
    val centersWithNorm = clusterCentersWithNorm
    val bcCentersWithNorm = data.context.broadcast(centersWithNorm)
    data.map(p => KMeans.pointCost(bcCentersWithNorm.value, new VectorWithNorm(p))).sum()
```

```
  }

  private def clusterCentersWithNorm: Iterable[VectorWithNorm] =
    clusterCenters.map(new VectorWithNorm(_))

  override def save(sc: SparkContext, path: String): Unit = {
    KMeansModel.SaveLoadV1_0.save(sc, this, path)
  }

  override protected def formatVersion: String = "1.0"
}

object KMeansModel extends Loader[KMeansModel] {
  override def load(sc: SparkContext, path: String): KMeansModel = {
    KMeansModel.SaveLoadV1_0.load(sc, path)
  }

  private case class Cluster(id: Int, point: Vector)

  private object Cluster {
    def apply(r: Row): Cluster = {
      Cluster(r.getInt(0), r.getAs[Vector](1))
    }
  }

  private[clustering]
  object SaveLoadV1_0 {

    private val thisFormatVersion = "1.0"

    private[clustering]
    val thisClassName = "org.apache.spark.mllib.clustering.KMeansModel"

    def save(sc: SparkContext, model: KMeansModel, path: String): Unit = {
      val sqlContext = new SQLContext(sc)
      import sqlContext.implicits._
      val metadata = compact(render(
        ("class" -> thisClassName) ~ ("version" -> thisFormatVersion) ~ ("k" -> model.k)))
      sc.parallelize(Seq(metadata), 1).saveAsTextFile(Loader.metadataPath(path))
      val dataRDD = sc.parallelize(model.clusterCenters.zipWithIndex).map { case (point, id) =>
        Cluster(id, point)
      }.toDF()
      dataRDD.write.parquet(Loader.dataPath(path))
```

```
    }

    def load(sc: SparkContext, path: String): KMeansModel = {
      implicit val formats = DefaultFormats
      val sqlContext = new SQLContext(sc)
      val (className, formatVersion, metadata) = Loader.loadMetadata(sc, path)
      assert(className == thisClassName)
      assert(formatVersion == thisFormatVersion)
      val k = (metadata \ "k").extract[Int]
      val centriods = sqlContext.read.parquet(Loader.dataPath(path))
      Loader.checkSchema[Cluster](centriods.schema)
      val localCentriods = centriods.map(Cluster.apply).collect()
      assert(k == localCentriods.size)
      new KMeansModel(localCentriods.sortBy(_.id).map(_.point))
    }
  }
}
```

10.3 实例

10.3.1 训练数据

数据格式为：特征 1 特征 2

```
1.629502   1.666991
1.871226   1.898365
1.46171    1.91306
1.58579    1.537943
2.018275   1.836801
1.98899    2.006619
1.599317   1.991072
1.991236   1.235661
1.057009   1.601767
1.889463   1.86318
1.368395   1.213885
1.251551   1.821578
1.904642   1.523114
1.383058   1.641584
1.182018   1.286603
1.030947   1.093305
2.050907   1.327946
1.74832    2.008842
……
```

10.3.2　实例代码

1. 具体实例代码

```
import org.apache.log4j.{ Level, Logger }
import org.apache.spark.{ SparkConf, SparkContext }
import org.apache.spark.mllib.clustering._
import org.apache.spark.mllib.linalg.Vectors

object KMeans {

  def main(args: Array[String]) {
    //1 构建 Spark 对象
    val conf = new SparkConf().setAppName("KMeans")
    val sc = new SparkContext(conf)
    Logger.getRootLogger.setLevel(Level.WARN)

    //2 读取样本数据，为 LIBSVM 格式
    val data = sc.textFile("hdfs://192.168.180.79:9000/user/huangmeiling/kmeans_data.txt")
    val parsedData = data.map(s => Vectors.dense(s.split(' ').map(_.toDouble))).cache()

    //3 新建 KMeans 聚类模型，并训练
    val initMode = "k-means||"
    val numClusters = 2
    val numIterations = 20
    val model = new KMeans()
      .setInitializationMode(initMode)
      .setK(numClusters)
      .setMaxIterations(numIterations)
      .run(parsedData)

    //4 误差计算
    val WSSSE = model.computeCost(parsedData)
    println("Within Set Sum of Squared Errors = " + WSSSE)

    //5 保存模型
    val ModelPath = "/user/huangmeiling/KMeans_Model"
    model.save(sc, ModelPath)
    val sameModel = KMeansModel.load(sc, ModelPath)

  }

}
```

2. 运行结果

聚类中心点：

1.531328	1.559521
3.528368	3.513859
2.523455	2.512361
4.527434	2.57113

聚类效果如图 10-2 所示。

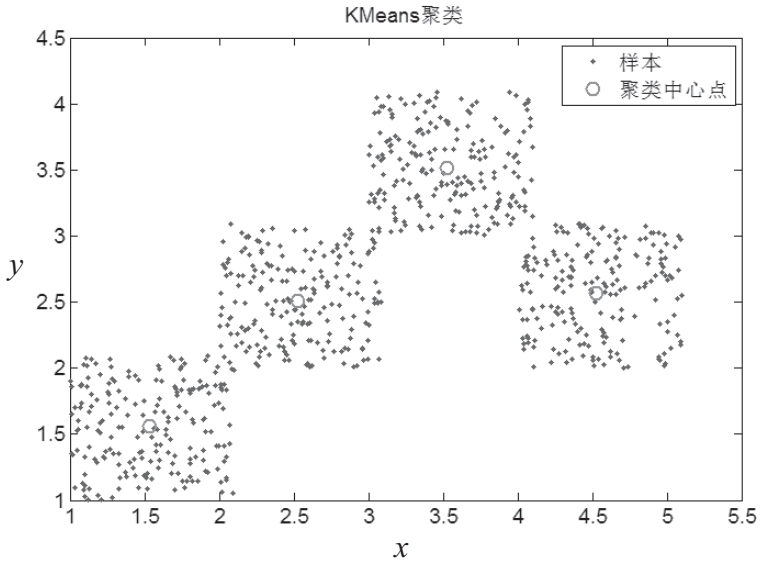

图 10-2　KMeans 聚类效果

第 14 章
Spark MLlib 协同过滤推荐算法

14.1 协同过滤推荐算法

14.1.1 协同过滤推荐概述

协同过滤推荐（Collaborative Filtering Recommendation）算法是最经典、最常用的推荐算法。该算法通过分析用户兴趣，在用户群中找到指定用户的相似用户，综合这些相似用户对某一信息的评价，形成系统关于该指定用户对此信息的喜好程度预测。比如你现在想看一部电影，但你不知道具体看哪部，你会怎么做？大部分人会问问周围的朋友，看看他们最近有什么好看的电影推荐，大家一般更倾向于从口味比较类似的朋友那里得到推荐。这就是协同过滤的核心思想。

要实现协同过滤，需要以下几个步骤：

1）收集用户偏好；

2）找到相似的用户或物品；

3）计算推荐。

14.1.2 用户评分

从用户的行为和偏好中发现规律，并基于此进行推荐，所以收集用户的偏好信息成为系统推荐效果最基础的决定因素。用户有很多种方式向系统提供自己的偏好信息，比如：评分、投票、转发、保存书签、购买、点击流、页面停留时间等。

以上用户行为都是通用的，在实际推荐引擎设计中可以自己多添加一些特定的用户行为，并用它们表示用户对物品的喜好程度。通常情况下，在一个推荐系统中，用户行为都会多于一种，那么如何组合这些不同的用户行为呢？基本上有如下两种方式。

1. 将不同的行为分组

一般可以分为查看和购买，然后基于不同的用户行为，计算不同用户或者物品的相似度。类似于当当网或者亚马逊给出的“购买了该书的人还购买了”“查看了该书的人还查看了”等。

2. 对不同行为进行加权

对不同行为产生的用户喜好进行加权，然后求出用户对物品的总体喜好。当我们收集好用户的行为数据后，还要对数据进行预处理，最核心的工作就是减噪和归一化。

- 减噪：因为用户数据在使用过程中可能存在大量噪声和误操作，所以需要过滤掉这些噪声。
- 归一化：不同行为数据的差别比较大，例如用户的查看数据肯定比购买数据大得多。通过归一化，才能使数据更加准确。

通过上述步骤的处理，就得到了一张二维表，其中一维是用户列表，另一维是商品列表，值是用户对商品的喜好，如表 14-1 所示。

表 14-1 用户评分表

用户/物品	物品A	物品B	物品C
用户A	0.1	0.8	1
用户B	0.1	0	0.02
用户C	0.5	0.3	0.1

14.1.3 相似度计算

对用户的行为分析得到用户的偏好后，可以根据用户的偏好计算相似用户和物品，然后可以基于相似用户或物品进行推荐。这就是协同过滤中的两个分支了，即基于用户的协同过滤和基于物品的协同过滤。

关于相似度的计算，现有的几种基本方法都是基于向量（Vector）的，其实也就是计算两

个向量的距离，距离越近相似度越大。在推荐的场景中，在用户-物品偏好的二维矩阵中，我们可以将一个用户对所有物品的偏好作为一个向量来计算用户之间的相似度，或者将所有用户对某个物品的偏好作为一个向量来计算物品之间的相似度。下面我们详细介绍几种常用的相似度计算方法。

1. 同现相似度

物品 i 和物品 j 的同现相似度公式定义：

$$w_{i,j}=\frac{|N(i)\cap N(j)|}{|N(i)|}$$

其中，分母$|N(i)|$是喜欢物品 i 的用户数，而分子$|N(i)\cap N(j)|$是同时喜欢物品 i 和物品 j 的用户数据。因此，上述公式可以理解为喜欢物品 i 的用户中有多少比例的用户也喜欢物品 j。

上述公式存在一个问题，如果物品 j 是热门物品，很多人都喜欢，那么 $w_{i,j}$ 就会很大，接近1。因此，该公式会造成任何物品都会和热门物品有很大的相似度。为了避免推荐出热门的物品，可以用如下公式：

$$w_{i,j}=\frac{|N(i)\cap N(j)|}{\sqrt{|N(i)||N(j)|}}$$

这个公式惩罚了物品 j 的权重，因此减轻了热门物品与很多物品相似的可能性。

2. 欧氏距离（Euclidean Distance）

最初用于计算欧几里得空间中两个点的距离，假设 x、y 是 n 维空间的两个点，它们之间的欧几里得距离是：

$$d(x,y)=\sqrt{\sum(x_i-y_i)^2}$$

可以看出，当 n=2 时，欧几里得距离就是平面上两个点的距离。

当用欧几里得距离表示相似度时，一般采用以下公式进行转换：距离越小，相似度越大。

$$\mathrm{sim}(x,y)=\frac{1}{1+d(x,y)}$$

3. 皮尔逊相关系数（Pearson Correlation Coefficient）

皮尔逊相关系数一般用于计算两个定距变量间联系的紧密程度，它的取值在[-1，+1]之间。

$$p(x,y)=\frac{\sum x_i y_i - n\overline{xy}}{(n-1)s_x s_y}=\frac{n\sum x_i y_i-\sum x_i \sum y_i}{\sqrt{n\sum x_i^2-(\sum x_i)^2}\sqrt{n\sum y_i^2-(\sum y_i)^2}}$$

其中，s_x、s_y是 x 和 y 的样品标准偏差。

4. Cosine 相似度（Cosine Similarity）

Cosine 相似度被广泛应用于计算文档数据的相似度：

$$T(x,y)=\frac{x\cdot y}{\|x\|^2\times\|y\|^2}=\frac{\sum x_i y_i}{\sqrt{\sum x_i^2}\sqrt{\sum y_i^2}}$$

5. Tanimoto 系数（Tanimoto Coefficient）

Tanimoto 系数也被称为 Jaccard 系数，是 Cosine 相似度的扩展，也多用于计算文档数据的相似度：

$$T(x,y)=\frac{x\cdot y}{\|x\|^2+\|y\|^2-x\cdot y}=\frac{\sum x_i y_i}{\sqrt{\sum x_i^2}+\sqrt{\sum y_i^2}-\sum x_i y_i}$$

在计算用户之间的相似度时，是将一个用户对所有物品的偏好作为一个向量；而在计算物品之间的相似度时，是将所有用户对某个物品的偏好作为一个向量。求出相似度后，接下来可以求相邻用户和相邻物品。

14.1.4 推荐计算

根据相似度计算用户和物品的相似度后，可以分别基于用户或者基于物品进行协同过滤推荐。

1. 基于用户的 CF（User CF）

基于用户的 CF 的基本思想相当简单：基于用户对物品的偏好找到相邻的邻居用户，然后将邻居用户喜欢的推荐给当前用户。在计算上，就是将一个用户对所有物品的偏好作为一个向量来计算用户之间的相似度，找到 K 邻居后，根据邻居的相似度权重及其对物品的偏好，预测当前用户没有偏好的未涉及物品，计算得到一个排序的物品列表作为推荐。图 14-1 给出了一个例子，对于用户 A，根据用户的历史偏好，这里只计算得到一个邻居–用户 C，然后将用户 C 喜欢的物品 D 推荐给用户 A。

2. 基于物品的 CF（Item CF）

基于物品的 CF 的原理和基于用户的 CF 类似，只是在计算邻居时采用物品本身，而不是从用户的角度。即基于用户对物品的偏好找到相似的物品，然后根据用户的历史偏好，推荐相似

的物品给他。从计算的角度看，就是将所有用户对某个物品的偏好作为一个向量来计算物品之间的相似度，得到物品的相似物品后，根据用户历史的偏好预测当前用户还没有表示偏好的物品，计算得到一个排序的物品列表作为推荐。图 14-2 给出了一个例子，对于物品 A，根据所有用户的历史偏好，喜欢物品 A 的用户都喜欢物品 C，得出物品 A 和物品 C 比较相似，而用户 C 喜欢物品 A，那么可以推断出用户 C 可能也喜欢物品 C。

用户/物品	物品A	物品B	物品C	物品D
用户A	√		√	推荐
用户B		√		
用户C	√		√	√

图 14-1　基于用户的 CF 的基本原理

用户/物品	物品A	物品B	物品C
用户A	√		√
用户B	√	√	√
用户C	√		推荐

图 14-2　基于物品的 CF 的基本原理

14.2　协同推荐算法实现

目前 Spark MLlib 中未实现协同过滤推荐算法，本节根据上面的理论知识，实现了基于物品的协同过滤推荐算法。

基于 MLlib 实现了相似度计算模型，可以计算物品与物品之间的相似度，支持同现相似度、欧氏距离相似度、余弦相似度 3 种相似度计算。图 14-3 所示是根据用户评分矩阵采用同现相似度计算物品相似度矩阵。

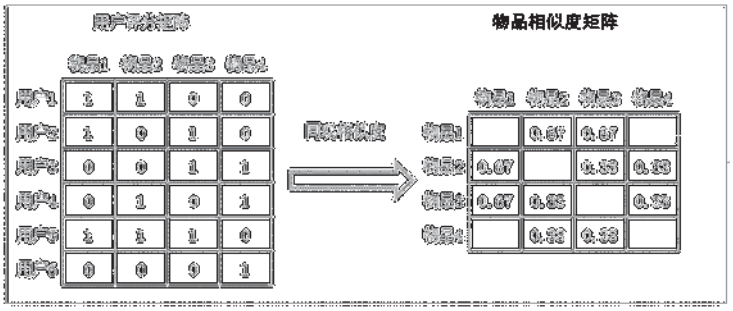

图 14-3　物品相似度矩阵

其相似度计算实现了分布式计算，实现过程如下：

对于同现相似度矩阵计算，首先对每一个样本（一条用户数据），取所有物品的笛卡儿积，

得到物品与物品的组合，然后通过 reduceByKey 统计所有物品与物品的出现频次，最后按照同现相似公式（$w(i,j) = N(i) \cap N(j)/\text{sqrt}(N(i)*N(j))$，其中分子是 i 与 j 的同现频次，分母的 $N(i)$是 i 频次、$N(j)$是 j 频次）计算物品与物品的相似度，过程如图 14-4 所示。

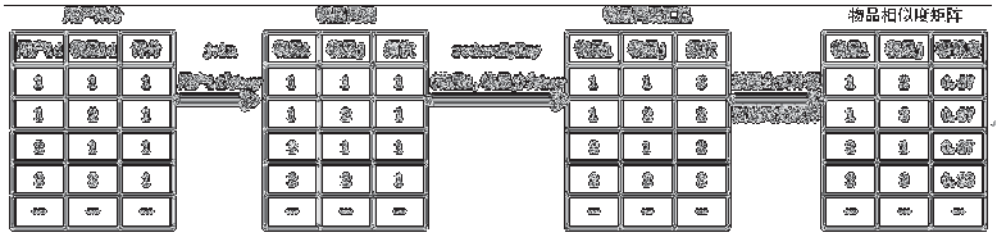

图 14-4　分布式同现相似度矩阵计算过程

对于欧氏相似度的计算，采用离散计算公式 $d(x, y) = \text{sqrt}(\sum((x(i)-y(i)) * (x(i)- y(i))))$。其中，$i$ 只取 x、y 同现的点，未同现的点不参与相似度计算；$\text{sim}(x, y) = m / (1 + d(x, y))$，$m$ 为 x、y 重叠数（同现次数）。首先对每一个样本（一条用户数据），取所有（物品，评分）的笛卡儿积，得到（物品 i，物品 j，评分 i，评分 j）的组合；然后以（物品 i，物品 j）为 key 对（评分 i 减去评分 j）进行累加统计，并且同时以（物品 i，物品 j）为 key 进行重叠数的统计；最后根据公式 $\text{sim}(x, y)$计算相似度，过程如图 14-5 所示。

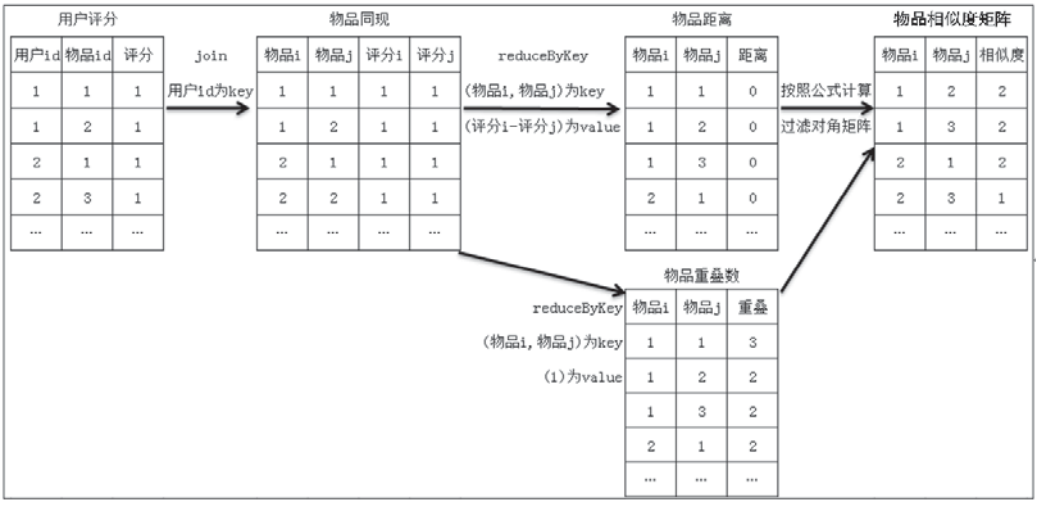

图 14-5　分布式欧氏距离相似度矩阵计算过程

基于 MLlib 实现了基于物品的协同过滤推荐模型，根据物品相似度模型、用户评分和指定最大推荐数量进行用户推荐，如图 14-6 所示是根据物品相似度矩阵和用户评分计算用户推荐列表，

计算公式是 ***R=W*A***，取推荐计算中用户未评分过的物品，并且按照计算结果倒序推荐给用户。

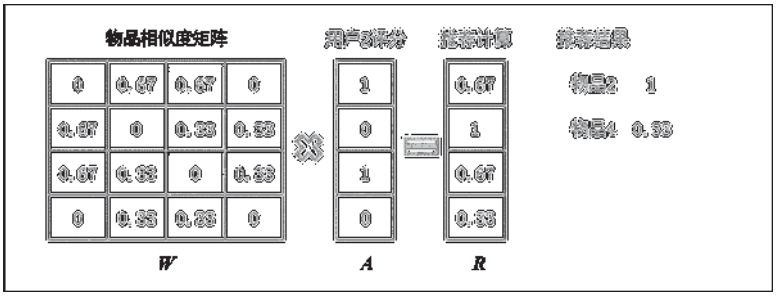

图 14-6　协同推荐计算

其推荐计算实现了分布式计算。首先对相似表和用户评分表以物品 id 为 key 进行关联，得到用户和物品之间的关系（评分乘以相似度）；然后通过 reduceByKey 进行汇总；最后通过过滤用户已评分物品，并对物品进行倒序排列推荐给用户，过程如图 14-7 所示。

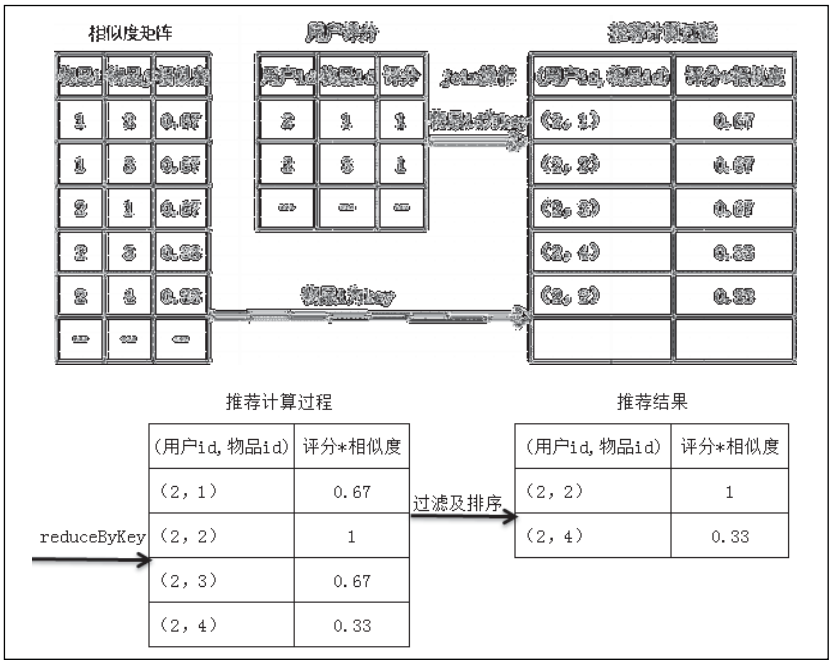

图 14-7　分布式协同推荐计算过程

基于 MLlib 实现的协同过滤推荐算法的源码分解参见表 14-2，具体源码解析参见下面各节。

表 14-2　ItemCF协同过滤源码分解

源码分解说明		
1. 物品相似度类	ItemSimilarity	ItemSimilarity类
1.1　相似度计算	Similarity	ItemSimilarity类的Similarity方法。该方法计算物品与物品之间的相似度
2. 基于物品协同推荐类	RecommendedItem	RecommendedItem类
2.1　推荐计算	Recommend	RecommendedItem类的Recommend方法。该方法根据物品与物品的相似度模型和用户评分，计算用户的推荐物品列表

14.2.1　相似度计算

通过 new ItemSimilarity，建立物品相似度计算类。通过设置模型参数后，执行 Similarity 方法，进行相似度计算，返回物品与物品的相似度 RDD。

相似度计算支持：同现相似度、余弦相似度、欧氏距离相似度；ItemSimilarity 类及对象的源码解析如下。

```
import scala.math._
import org.apache.spark.rdd.RDD
import org.apache.spark.SparkContext._

/**
 * 用户评分
 * @param userid 用户
 * @param itemid 评分物品
 * @param pref 评分
 */
case class ItemPref(
  val userid: String,
  val itemid: String,
  val pref: Double) extends Serializable
/**
 * 用户推荐
 * @param userid 用户
 * @param itemid 推荐物品
 * @param pref 评分
 */
case class UserRecomm(
  val userid: String,
  val itemid: String,
  val pref: Double) extends Serializable
```

```
/**
 * 相似度
 * @param itemid1 物品
 * @param itemid2 物品
 * @param similar 相似度
 */
case class ItemSimi(
  val itemid1: String,
  val itemid2: String,
  val similar: Double) extends Serializable

/**
 * 相似度计算
 * 支持：同现相似度、欧氏距离相似度、余弦相似度
 *
 */
class ItemSimilarity extends Serializable {

  /**
   * 相似度计算
   * @param user_rdd 用户评分
   * @param stype 计算相似度公式
   * @param RDD[ItemSimi] 返回物品相似度
   *
   */
  def Similarity(user_rdd: RDD[ItemPref], stype: String): (RDD[ItemSimi]) = {
    val simil_rdd = stype match {
      case "cooccurrence" =>
        ItemSimilarity.CooccurrenceSimilarity(user_rdd)
      case "cosine" =>
        ItemSimilarity.CosineSimilarity(user_rdd)
      case "euclidean" =>
        ItemSimilarity.EuclideanDistanceSimilarity(user_rdd)
      case _ =>
        ItemSimilarity.CooccurrenceSimilarity(user_rdd)
    }
    simil_rdd
  }

}

object ItemSimilarity {
```

```
/**
 * 同现相似度矩阵计算
 * w(i,j) = N(i)∩N(j)/sqrt(N(i)*N(j))
 * @param user_rdd 用户评分
 * @param RDD[ItemSimi] 返回物品相似度
 *
 */
def CooccurrenceSimilarity(user_rdd: RDD[ItemPref]): (RDD[ItemSimi]) = {
  // 1 数据做准备
  val user_rdd1 = user_rdd.map(f => (f.userid, f.itemid, f.pref))
  val user_rdd2 = user_rdd1.map(f => (f._1, f._2))
  // 2（用户，物品）笛卡儿积操作 => 物品与物品的组合
  val user_rdd3 = user_rdd2.join(user_rdd2)
  val user_rdd4 = user_rdd3.map(f => (f._2, 1))
  // 3（物品，物品，频次）
  val user_rdd5 = user_rdd4.reduceByKey((x, y) => x + y)
  // 4 对角矩阵
  val user_rdd6 = user_rdd5.filter(f => f._1._1 == f._1._2)
  // 5 非对角矩阵
  val user_rdd7 = user_rdd5.filter(f => f._1._1 != f._1._2)
  // 6 计算同现相似度（物品 1，物品 2，同现频次）
  val user_rdd8 = user_rdd7.map(f => (f._1._1, (f._1._1, f._1._2, f._2))).
    join(user_rdd6.map(f => (f._1._1, f._2)))
  val user_rdd9 = user_rdd8.map(f => (f._2._1._2, (f._2._1._1,
    f._2._1._2, f._2._1._3, f._2._2)))
  val user_rdd10 = user_rdd9.join(user_rdd6.map(f => (f._1._1, f._2)))
  val user_rdd11 = user_rdd10.map(f => (f._2._1._1, f._2._1._2, f._2._1._3, f._2._1._4,
    f._2._2))
  val user_rdd12 = user_rdd11.map(f => (f._1, f._2, (f._3 / sqrt(f._4 * f._5))))
  // 7 结果返回
  user_rdd12.map(f => ItemSimi(f._1, f._2, f._3))
}

/**
 * 余弦相似度矩阵计算
 * T(x,y) = ∑x(i)y(i) / sqrt(∑(x(i)*x(i)) * ∑(y(i)*y(i)))
 * @param user_rdd 用户评分
 * @param RDD[ItemSimi] 返回物品相似度
 *
 */
def CosineSimilarity(user_rdd: RDD[ItemPref]): (RDD[ItemSimi]) = {
  // 1 数据做准备
  val user_rdd1 = user_rdd.map(f => (f.userid, f.itemid, f.pref))
```

```
    val user_rdd2 = user_rdd1.map(f => (f._1, (f._2, f._3)))
    // 2（用户，物品，评分）笛卡儿积操作 =>（物品 1，物品 2，评分 1，评分 2）组合
    val user_rdd3 = user_rdd2.join(user_rdd2)
    val user_rdd4 = user_rdd3.map(f => ((f._2._1._1, f._2._2._1), (f._2._1._2, f._2._2._2)))
    // 3（物品 1，物品 2，评分 1，评分 2）组合 => （物品 1，物品 2，评分 1*评分 2）组合并累加
    val user_rdd5 = user_rdd4.map(f => (f._1, f._2._1 * f._2._2)).reduceByKey(_ + _)
    // 4 对角矩阵
    val user_rdd6 = user_rdd5.filter(f => f._1._1 == f._1._2)
    // 5 非对角矩阵
    val user_rdd7 = user_rdd5.filter(f => f._1._1 != f._1._2)
    // 6 计算相似度
    val user_rdd8 = user_rdd7.map(f => (f._1._1, (f._1._1, f._1._2, f._2))).
      join(user_rdd6.map(f => (f._1._1, f._2)))
    val user_rdd9 = user_rdd8.map(f => (f._2._1._2, (f._2._1._1,
      f._2._1._2, f._2._1._3, f._2._2)))
    val user_rdd10 = user_rdd9.join(user_rdd6.map(f => (f._1._1, f._2)))
    val user_rdd11 = user_rdd10.map(f => (f._2._1._1, f._2._1._2, f._2._1._3, f._2._1._4, f._2._2))
    val user_rdd12 = user_rdd11.map(f => (f._1, f._2, (f._3 / sqrt(f._4 * f._5))))
    // 7 结果返回
    user_rdd12.map(f => ItemSimi(f._1, f._2, f._3))
  }

  /**
   * 欧氏距离相似度矩阵计算
   * d(x, y) = sqrt(∑((x(i)-y(i)) * (x(i)-y(i))))
   * sim(x, y) = n / (1 + d(x, y))
   * @param user_rdd 用户评分
   * @param RDD[ItemSimi] 返回物品相似度
   *
   */
  def EuclideanDistanceSimilarity(user_rdd: RDD[ItemPref]): (RDD[ItemSimi]) = {
    // 1 数据做准备
    val user_rdd1 = user_rdd.map(f => (f.userid, f.itemid, f.pref))
    val user_rdd2 = user_rdd1.map(f => (f._1, (f._2, f._3)))
    // 2（用户，物品，评分）笛卡儿积操作 =>（物品 1，物品 2，评分 1，评分 2）组合
    val user_rdd3 = user_rdd2 join user_rdd2
    val user_rdd4 = user_rdd3.map(f => ((f._2._1._1, f._2._2._1), (f._2._1._2, f._2._2._2)))
    // 3（物品 1，物品 2，评分 1，评分 2）组合 =>（物品 1，物品 2，评分 1 减去评分 2）组合并累加
    val user_rdd5 = user_rdd4.map(f => (f._1, (f._2._1 - f._2._2) * (f._2._1 - f._2._2))).
      reduceByKey(_ + _)
    // 4 （物品 1，物品 2，评分 1，评分 2）组合 =>（物品 1，物品 2，1）组合计算物品 1 与物品 2 的重叠数
    val user_rdd6 = user_rdd4.map(f => (f._1, 1)).reduceByKey(_ + _)
    // 5 非对角矩阵
```

```
    val user_rdd7 = user_rdd5.filter(f => f._1._1 != f._1._2)
    // 6 计算相似度
    val user_rdd8 = user_rdd7.join(user_rdd6)
    val user_rdd9 = user_rdd8.map(f => (f._1._1, f._1._2, f._2._2 / (1 + sqrt(f._2._1))))
    // 7 结果返回
    user_rdd9.map(f => ItemSimi(f._1, f._2, f._3))
  }

}
```

14.2.2 协同推荐计算

通过 new RecommendedItem，建立物品推荐计算类，通过设置模型参数后，执行 Recommend 方法，进行推荐计算，返回用户的推荐物品 RDD。

推荐计算根据物品相似度和用户评分进行推荐物品计算，并过滤用户已有物品及过滤最大推荐数量。RecommendedItem 类及对象的源码解析如下。

```
import org.apache.spark.rdd.RDD
import org.apache.spark.SparkContext._
  /**
   * 用户推荐计算
   * 根据物品相似度、用户评分、指定最大推荐数量进行用户推荐
   */
class RecommendedItem {
  /**
   * 用户推荐计算
   * @param items_similar 物品相似度
   * @param user_prefer 用户评分
   * @param r_number 推荐数量
   * @param RDD[UserRecomm] 返回用户推荐物品
   *
   */
  def Recommend(items_similar: RDD[ItemSimi],
    user_prefer: RDD[ItemPref],
    r_number: Int): (RDD[UserRecomm]) = {
    // 1 数据准备
    val rdd_app1_R1 = items_similar.map(f => (f.itemid1, f.itemid2, f.similar))
    val user_prefer1 = user_prefer.map(f => (f.userid, f.itemid, f.pref))
    // 2 矩阵计算——i 行与 j 列 join
    val rdd_app1_R2 = rdd_app1_R1.map(f => (f._1, (f._2, f._3))).
      join(user_prefer1.map(f => (f._2, (f._1, f._3))))
    // 3 矩阵计算——i 行与 j 列元素相乘
```

```
    val rdd_app1_R3 = rdd_app1_R2.map(f => ((f._2._2._1, f._2._1._1), f._2._2._2 * f._2._1._2))
    // 4 矩阵计算——用户：元素累加求和
    val rdd_app1_R4 = rdd_app1_R3.reduceByKey((x, y) => x + y)
    // 5 矩阵计算——用户：对结果过滤已有的物品
    val rdd_app1_R5 = rdd_app1_R4.leftOuterJoin(user_prefer1.map(f => ((f._1, f._2), 1))).
      filter(f => f._2._2.isEmpty).map(f => (f._1._1, (f._1._2, f._2._1)))
    // 6 矩阵计算——用户：用户对结果排序，过滤
    val rdd_app1_R6 = rdd_app1_R5.groupByKey()
    val rdd_app1_R7 = rdd_app1_R6.map(f => {
      val i2 = f._2.toBuffer
      val i2_2 = i2.sortBy(_._2)
      if (i2_2.length > r_number) i2_2.remove(0, (i2_2.length - r_number))
      (f._1, i2_2.toIterable)
    })
    val rdd_app1_R8 = rdd_app1_R7.flatMap(f => {
      val id2 = f._2
      for (w <- id2) yield (f._1, w._1, w._2)
    })
    rdd_app1_R8.map(f => UserRecomm(f._1, f._2, f._3))
  }

  /**
   * 用户推荐计算
   * @param items_similar 物品相似度
   * @param user_prefer 用户评分
   * @param RDD[UserRecomm] 返回用户推荐物品
   *
   */
  def Recommend(items_similar: RDD[ItemSimi],
    user_prefer: RDD[ItemPref]): (RDD[UserRecomm]) = {
    // 1 数据准备
    val rdd_app1_R1 = items_similar.map(f => (f.itemid1, f.itemid2, f.similar))
    val user_prefer1 = user_prefer.map(f => (f.userid, f.itemid, f.pref))
    // 2 矩阵计算——i 行与 j 列 join
    val rdd_app1_R2 = rdd_app1_R1.map(f => (f._1, (f._2, f._3))).
      join(user_prefer1.map(f => (f._2, (f._1, f._3))))
    // 3 矩阵计算——i 行与 j 列元素相乘
    val rdd_app1_R3 = rdd_app1_R2.map(f => ((f._2._2._1, f._2._1._1), f._2._2._2 * f._2._1._2))
    // 4 矩阵计算——用户：元素累加求和
    val rdd_app1_R4 = rdd_app1_R3.reduceByKey((x, y) => x + y)
    // 5 矩阵计算——用户：对结果过滤已有物品
    val rdd_app1_R5 = rdd_app1_R4.leftOuterJoin(user_prefer1.map(f => ((f._1, f._2), 1))).
      filter(f => f._2._2.isEmpty).map(f => (f._1._1, (f._1._2, f._2._1)))
```

```
    // 6 矩阵计算——用户：用户对结果排序，过滤
    val rdd_app1_R6 = rdd_app1_R5.map(f => (f._1, f._2._1, f._2._2)).
      sortBy(f => (f._1, f._3))
    rdd_app1_R6.map(f => UserRecomm(f._1, f._2, f._3))
  }

}
```

14.3 实例

14.3.1 训练数据

数据格式为：用户 id，物品 id，评分

```
1,1,1
1,2,1
2,1,1
2,3,1
3,3,1
3,4,1
4,2,1
4,4,1
5,1,1
5,2,1
5,3,1
6,4,1
```

14.3.2 实例代码

1. 具体实例代码

```
import org.apache.log4j.{ Level, Logger }
import org.apache.spark.{ SparkConf, SparkContext }
import org.apache.spark.rdd.RDD

object ItemCF {
  def main(args: Array[String]) {

    //1 构建 Spark 对象
    val conf = new SparkConf().setAppName("ItemCF")
    val sc = new SparkContext(conf)
    Logger.getRootLogger.setLevel(Level.WARN)
```

```
    //2 读取样本数据
    val data_path = "/home/jb-huangmeiling/sample_itemcf2.txt"
    val data = sc.textFile(data_path)
    val userdata = data.map(_.split(",")).map(f => (ItemPref(f(0), f(1), f(2).toDouble))).
      cache()

    //3 建立模型
    val mysimil = new ItemSimilarity()
    val simil_rdd1 = mysimil.Similarity(userdata, "cooccurrence")
    val recommd = new RecommendedItem
    val recommd_rdd1 = recommd.Recommend(simil_rdd1, userdata, 30)

    //4 打印结果
    println(s"物品相似度矩阵: ${simil_rdd1.count()}")
    simil_rdd1.collect().foreach { ItemSimi =>
      println(ItemSimi.itemid1 + ", " + ItemSimi.itemid2 + ", " + ItemSimi.similar)
    }
    println(s"用户推荐列表: ${recommd_rdd1.count()}")
    recommd_rdd1.collect().foreach { UserRecomm =>
      println(UserRecomm.userid + ", " + UserRecomm.itemid + ", " + UserRecomm.pref)
    }

  }
}
```

2. 运行结果

相似度矩阵（物品 i，物品 j，相似度）：

```
2, 4, 0.3333333333333333
3, 4, 0.3333333333333333
4, 2, 0.3333333333333333
3, 2, 0.3333333333333333
1, 2, 0.6666666666666666
4, 3, 0.3333333333333333
2, 3, 0.3333333333333333
1, 3, 0.6666666666666666
2, 1, 0.6666666666666666
3, 1, 0.6666666666666666
```

用户推荐列表（用户，物品，推荐值）：

```
4, 3, 0.6666666666666666
4, 1, 0.6666666666666666
```

```
6, 2, 0.3333333333333333
6, 3, 0.3333333333333333
2, 4, 0.3333333333333333
2, 2, 1.0
5, 4, 0.6666666666666666
3, 2, 0.6666666666666666
3, 1, 0.6666666666666666
1, 4, 0.3333333333333333
1, 3, 1.0
```