

SparkMLlib 实战

目 录

1 MLLIB实例.....	3
1.1 聚类实例	3
1.1.1 算法说明.....	3
1.1.2 实例介绍.....	3
1.1.3 测试数据说明.....	4
1.1.4 程序代码.....	4
1.1.5 IDEA执行情况.....	6
1.2 回归算法实例	8
1.2.1 算法说明.....	8
1.2.2 实例介绍.....	8
1.2.3 程序代码.....	9
1.2.4 执行情况.....	10
1.3 协同过滤实例	11
1.3.1 算法说明.....	11
1.3.2 实例介绍.....	13
1.3.3 测试数据说明.....	13
1.3.4 程序代码.....	15
1.3.5 IDEA执行情况.....	20
2 参考资料.....	22

SparkMLlib 实战

1 MLlib 实例

1.1 聚类实例

1.1.1 算法说明

聚类 (Cluster analysis) 有时也被翻译为簇类，其核心任务是：将一组目标 object 划分为若干个簇，每个簇之间的 object 尽可能相似，簇与簇之间的 object 尽可能相异。聚类算法是机器学习 (或者说是数据挖掘更合适) 中重要的一部分，除了最为简单的 K-Means 聚类算法外，比较常见的还有层次法 (CURE、CHAMELEON 等)、网格算法 (STING、WaveCluster 等)，等等。

较权威的聚类问题定义：所谓聚类问题，就是给定一个元素集合 D ，其中每个元素具有 n 个可观察属性，使用某种算法将 D 划分成 k 个子集，要求每个子集内部的元素之间相异度尽可能低，而不同子集的元素相异度尽可能高。其中每个子集叫做一个簇。

K-means 聚类属于无监督学习，以往的回归、朴素贝叶斯、SVM 等都是有类别标签 y 的，也就是说样例中已经给出了样例的分类。而聚类的样本中却没有给定 y ，只有特征 x ，比如假设宇宙中的星星可以表示成三维空间中的点集 (x,y,z) 。聚类的目的是找到每个样本 x 潜在类别 y ，并将同类别 y 的样本 x 放在一起。比如上面的星星，聚类后结果是一个个星团，星团里面的点相互距离比较近，星团间的星星距离就比较远了。

与分类不同，分类是示例式学习，要求分类前明确各个类别，并断言每个元素映射到一个类别。而聚类是观察式学习，在聚类前可以不知道类别甚至不给定类别数量，是无监督学习的一种。目前聚类广泛应用于统计学、生物学、数据库技术和市场营销等领域，相应的算法也非常多。

1.1.2 实例介绍

在该实例中将介绍 K-Means 算法，K-Means 属于基于平方误差的迭代重分配聚类算法，其核心思想十分简单：

- 随机选择 K 个中心点；
- 计算所有点到这 K 个中心点的距离，选择距离最近的中心点为其所在的簇；
- 简单地采用算术平均数 (mean) 来重新计算 K 个簇的中心；

- 重复步骤 2 和 3，直至簇类不再发生变化或者达到最大迭代值；
- 输出结果。

K-Means 算法的结果好坏依赖于对初始聚类中心的选择，容易陷入局部最优解，对 K 值的选择没有准则可依循，对异常数据较为敏感，只能处理数值属性的数据，聚类结构可能不平衡。

本实例中进行如下步骤：

1. 装载数据，数据以文本文件方式进行存放；
2. 将数据集聚类，设置 2 个类和 20 次迭代，进行模型训练形成数据模型；
3. 打印数据模型的中心点；
4. 使用误差平方之和来评估数据模型；
5. 使用模型测试单点数据；
6. 交叉评估 1，返回结果；交叉评估 2，返回数据集和结果。

1.1.3 测试数据说明

该实例使用的数据为 kmeans_data.txt，可以在本系列附带资源/data/class8/目录中找到。在该文件中提供了 6 个点的空间位置坐标，使用 K-means 聚类对这些点进行分类。

使用的 kmeans_data.txt 的数据如下所示：

```
0.0 0.0 0.0  
0.1 0.1 0.1  
0.2 0.2 0.2  
9.0 9.0 9.0  
9.1 9.1 9.1  
9.2 9.2 9.2
```

1.1.4 程序代码

```
import org.apache.log4j.{Level, Logger}  
import org.apache.spark.{SparkConf, SparkContext}  
import org.apache.spark.mllib.clustering.KMeans  
import org.apache.spark.mllib.linalg.Vectors  
  
object Kmeans {  
  def main(args: Array[String]) {
```

// 屏蔽不必要的日志显示在终端上

Logger.getLogger("org.apache.spark").setLevel(Level.WARN)

Logger.getLogger("org.eclipse.jetty.server").setLevel(Level.OFF)

// 设置运行环境

val conf = new SparkConf().setAppName("Kmeans").setMaster("local[4]")

val sc = new SparkContext(conf)

// 装载数据集

val data = sc.textFile("/home/hadoop/upload/class8/kmeans_data.txt", 1)

val parsedData = data.map(s => Vectors.dense(s.split(' ').map(_.toDouble)))

// 将数据集聚类，2 个类，20 次迭代，进行模型训练形成数据模型

val numClusters = 2

val numIterations = 20

val model = KMeans.train(parsedData, numClusters, numIterations)

// 打印数据模型的中心点

println("Cluster centers:")

for (c <- model.clusterCenters) {

println(" " + c.toString)

}

// 使用误差平方之和来评估数据模型

val cost = model.computeCost(parsedData)

println("Within Set Sum of Squared Errors = " + cost)

// 使用模型测试单点数据

println("Vectors 0.2 0.2 0.2 is belongs to clusters:" +

model.predict(Vectors.dense("0.2 0.2 0.2".split(' ').map(_.toDouble))))

println("Vectors 0.25 0.25 0.25 is belongs to clusters:" +

model.predict(Vectors.dense("0.25 0.25 0.25".split(' ').map(_.toDouble))))

println("Vectors 8 8 8 is belongs to clusters:" + model.predict(Vectors.dense("8 8 8".split(' ').map(_.toDouble))))

// 交叉评估 1 , 只返回结果

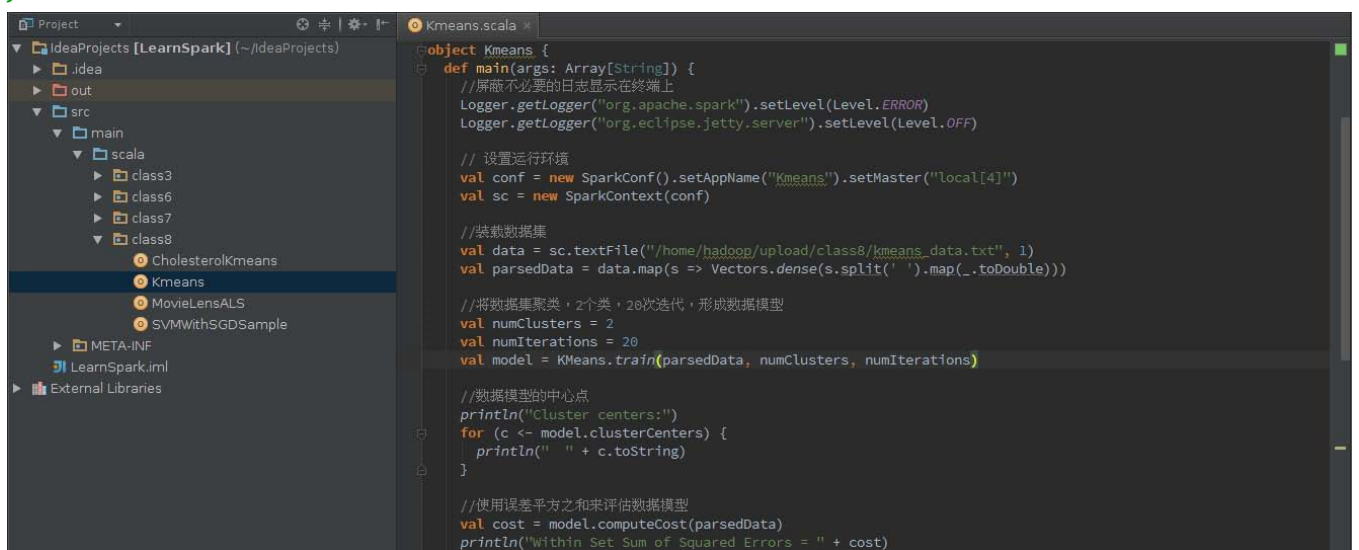
```
val testdata = data.map(s => Vectors.dense(s.split(' ').map(_.toDouble)))  
val result1 = model.predict(testdata)  
result1.saveAsTextFile("/home/hadoop/upload/class8/result_kmeans1")
```

// 交叉评估 2 , 返回数据集和结果

```
val result2 = data.map {  
  line =>  
    val linevectore = Vectors.dense(line.split(' ').map(_.toDouble))  
    val prediction = model.predict(linevectore)  
    line + " " + prediction  
}.saveAsTextFile("/home/hadoop/upload/class8/result_kmeans2")
```

sc.stop()

```
}  
  
}
```



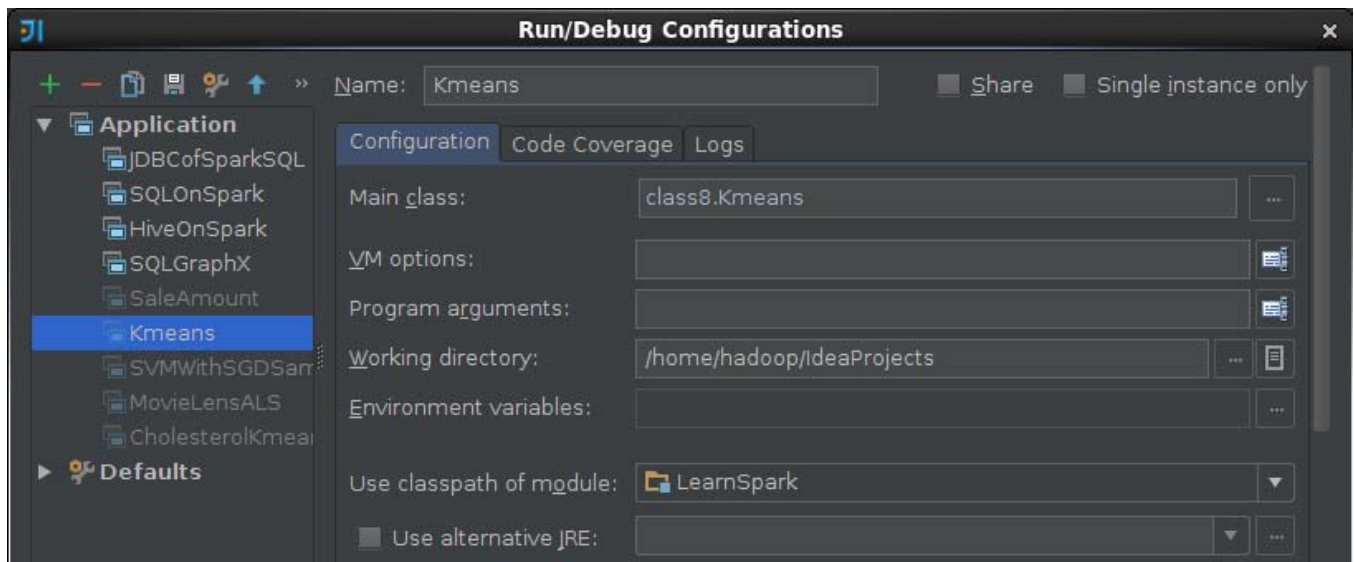
1.1.5 IDEA 执行情况

第一步 使用如下命令启动 Spark 集群

```
$cd /app/hadoop/spark-1.1.0  
$sbin/start-all.sh
```

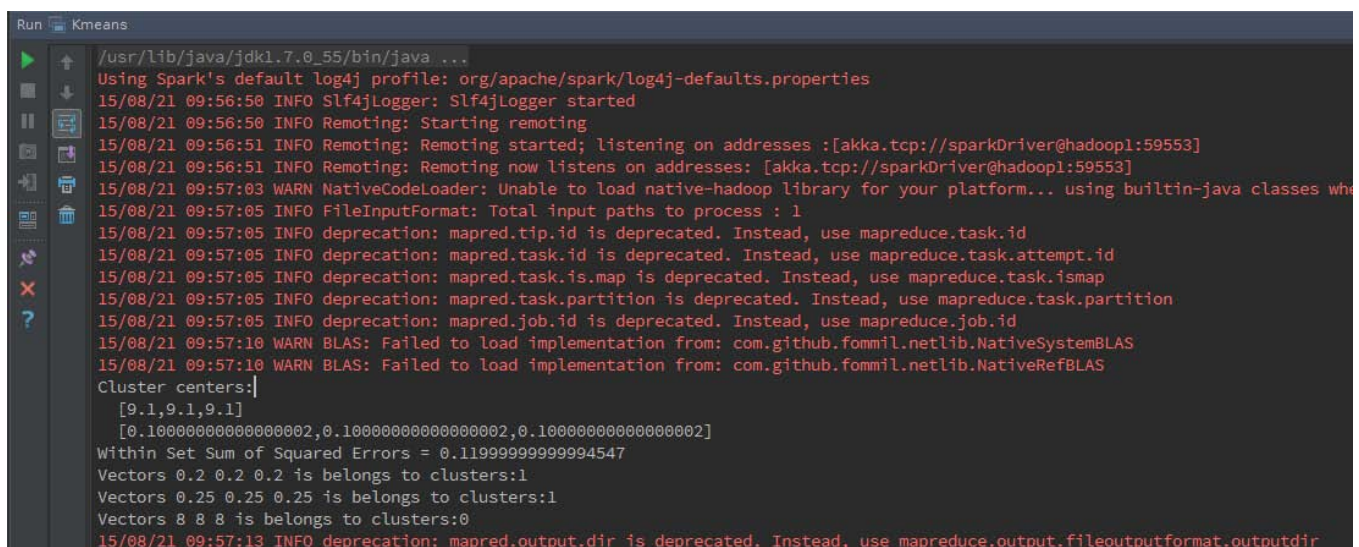
第二步 在 IDEA 中设置运行环境

在 IDEA 运行配置中设置 Kmeans 运行配置，由于读入的数据已经在程序中指定，故在该设置界面中不需要设置输入参数



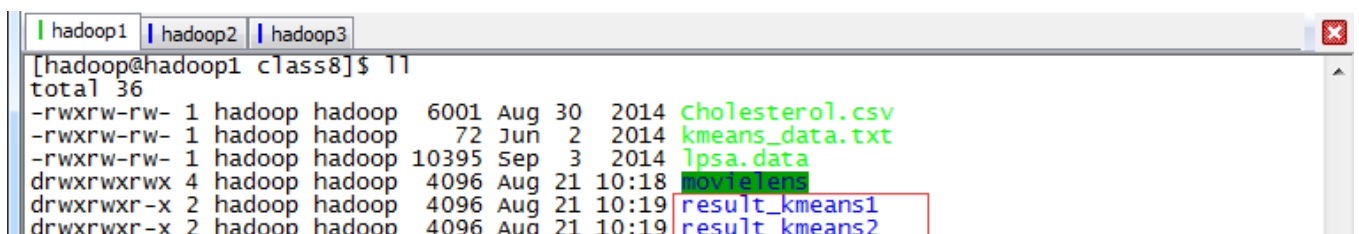
第三步 执行并观察输出

在运行日志窗口中可以看到，通过计算计算出模型并找出两个簇中心点：(9.1, 9.1, 9.1)和(0.1, 0.1, 0.1)，使用模型对测试点进行分类求出分属于族簇。



第四步 查看输出结果文件

在/home/hadoop/upload/class8 目录中有两个输出目录：



查看结果 1，在该目录中只输出了结果，分别列出了 6 个点所属不同的族簇


```
hadoop1 | hadoop2 | hadoop3
[hadoop@hadoop1 class8]$ cd result_kmeans1
[hadoop@hadoop1 result_kmeans1]$ ll
total 4
-rw-r--r-- 1 hadoop hadoop 12 Aug 21 10:19 part-00000
-rw-r--r-- 1 hadoop hadoop 0 Aug 21 10:19 _SUCCESS
[hadoop@hadoop1 result_kmeans1]$ cat part-00000
0
0
0
1
1
1
```

查看结果 2，在该目录中输出了数据集和结果

```
hadoop1 | hadoop2 | hadoop3
[hadoop@hadoop1 class8]$ cd result_kmeans2
[hadoop@hadoop1 result_kmeans2]$ ls
part-00000 _SUCCESS
[hadoop@hadoop1 result_kmeans2]$ cat part-00000
0.0 0.0 0.0 0
0.1 0.1 0.1 0
0.2 0.2 0.2 0
9.0 9.0 9.0 1
9.1 9.1 9.1 1
9.2 9.2 9.2 1
[hadoop@hadoop1 result_kmeans2]$
```

1.2 回归算法实例

1.2.1 算法说明

线性回归是利用称为线性回归方程的函数对一个或多个自变量和因变量之间关系进行建模的一种回归分析方法，只有一个自变量的情况称为简单回归，大于一个自变量情况的叫做多元回归，在实际情况中大多数都是多元回归。

线性回归 (Linear Regression) 问题属于监督学习 (Supervised Learning) 范畴，又称分类 (Classification) 或归纳学习 (Inductive Learning)。这类分析中训练数据集中给出的数据类型是确定的。机器学习的目标是，对于给定的一个训练数据集，通过不断的分析和学习产生一个联系属性集合和类标集合的分类函数 (Classification Function) 或预测函数 (Prediction Function)，这个函数称为分类模型 (Classification Model——或预测模型 (Prediction Model)。通过学习得到的模型可以是一个决策树、规格集、贝叶斯模型或一个超平面。通过这个模型可以对输入对象的特征向量预测或对对象的类标进行分类。

回归问题中通常使用最小二乘 (Least Squares) 法来迭代最优的特征中每个属性的比重，通过损失函数 (Loss Function) 或错误函数 (Error Function)定义来设置收敛状态，即作为梯度下降算法的逼近参数因子。

1.2.2 实例介绍

该例子给出了如何导入训练集数据，将其解析为带标签点的 RDD，然后使用了 LinearRegressionWithSGD 算法来建立一个简单的线性模型来预测标签的值，最后计算了均方差来评估预测值与实际值的吻合度。

线性回归分析的整个过程可以简单描述为如下三个步骤：

(1) 寻找合适的预测函数，即上文中的 $h(x)$ ，用来预测输入数据的判断结果。这个过程是非常关键的，需要对数据有一定的了解或分析，知道或者猜测预测函数的“大概”形式，比如是线性函数还是非线性函数，若是非线性的则无法用线性回归来得出高质量的结果。

(2) 构造一个 Loss 函数（损失函数），该函数表示预测的输出（ h ）与训练数据标签之间的偏差，可以是二者之间的差（ $h-y$ ）或者是其他的形式（如平方差开方）。综合考虑所有训练数据的“损失”，将 Loss 求和或者求平均，记为 $J(\theta)$ 函数，表示所有训练数据预测值与实际类别的偏差。

(3) 显然， $J(\theta)$ 函数的值越小表示预测函数越准确（即 h 函数越准确），所以这一步需要做的是找到 $J(\theta)$ 函数的最小值。找函数的最小值有不同的方法，Spark 中采用的是梯度下降法（stochastic gradient descent，SGD）。

1.2.3 程序代码

```
import org.apache.log4j.{Level, Logger}
import org.apache.spark.{SparkContext, SparkConf}
import org.apache.spark.mllib.regression.LinearRegressionWithSGD
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.linalg.Vectors

object LinearRegression {
  def main(args:Array[String]): Unit = {
    // 屏蔽不必要的日志显示终端上
    Logger.getLogger("org.apache.spark").setLevel(Level.ERROR)
    Logger.getLogger("org.eclipse.jetty.server").setLevel(Level.OFF)

    // 设置运行环境
    val conf = new SparkConf().setAppName("Kmeans").setMaster("local[4]")
    val sc = new SparkContext(conf)

    // Load and parse the data
    val data = sc.textFile("/home/hadoop/upload/class8/lpsa.data")
    val parsedData = data.map { line =>
      val parts = line.split(',')
      LabeledPoint(parts(0).toDouble, Vectors.dense(parts(1).split(' ').map(_.toDouble)))
    }
  }
}
```

```
}
```

```
// 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)}.reduce(_ + _) /
```

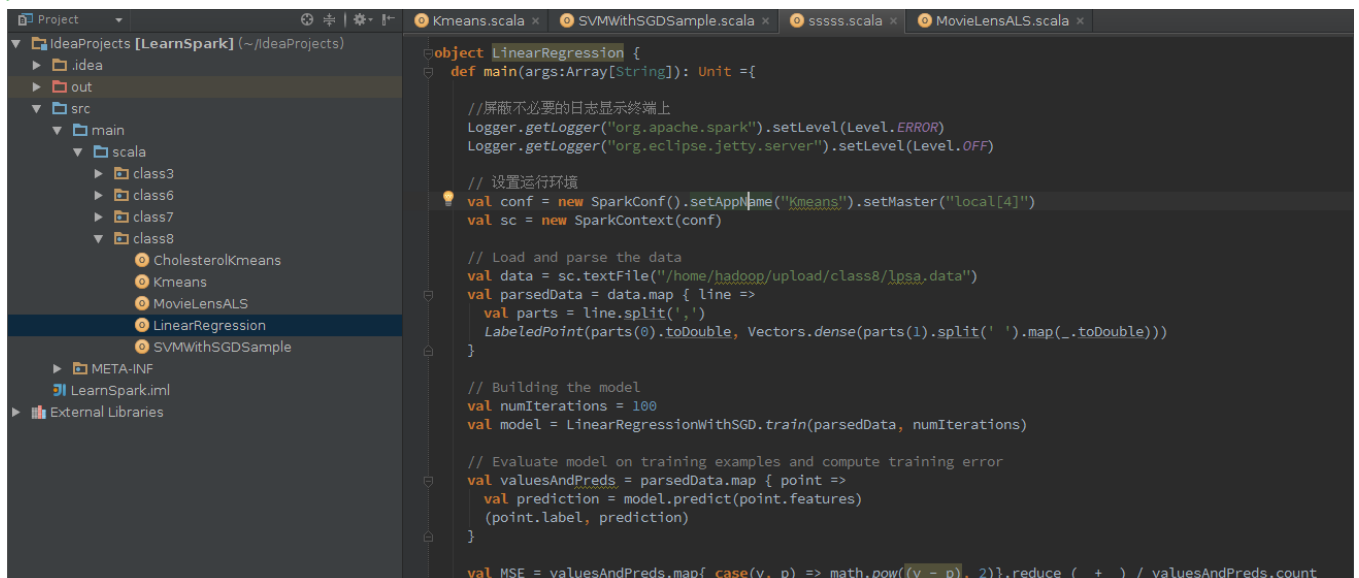
```
valuesAndPreds.count
```

```
println("training Mean Squared Error = " + MSE)
```

```
sc.stop()
```

```
}
```

```
}
```



```
object LinearRegression {  
  def main(args: Array[String]): Unit = {  
    //屏蔽不必要的日志显示终端上  
    Logger.getLogger("org.apache.spark").setLevel(Level.ERROR)  
    Logger.getLogger("org.eclipse.jetty.server").setLevel(Level.OFF)  
  
    // 设置运行环境  
    val conf = new SparkConf().setAppName("Kmeans").setMaster("local[4]")  
    val sc = new SparkContext(conf)  
  
    // Load and parse the data  
    val data = sc.textFile("/home/hadoop/upload/class8/lpsa.data")  
    val parsedData = data.map { line =>  
      val parts = line.split(',')  
      LabeledPoint(parts(0).toDouble, Vectors.dense(parts(1).split(' ').map(_.toDouble)))  
    }  
  
    // 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)}.reduce(_ + _) / valuesAndPreds.count  
  }  
}
```

1.2.4 执行情况

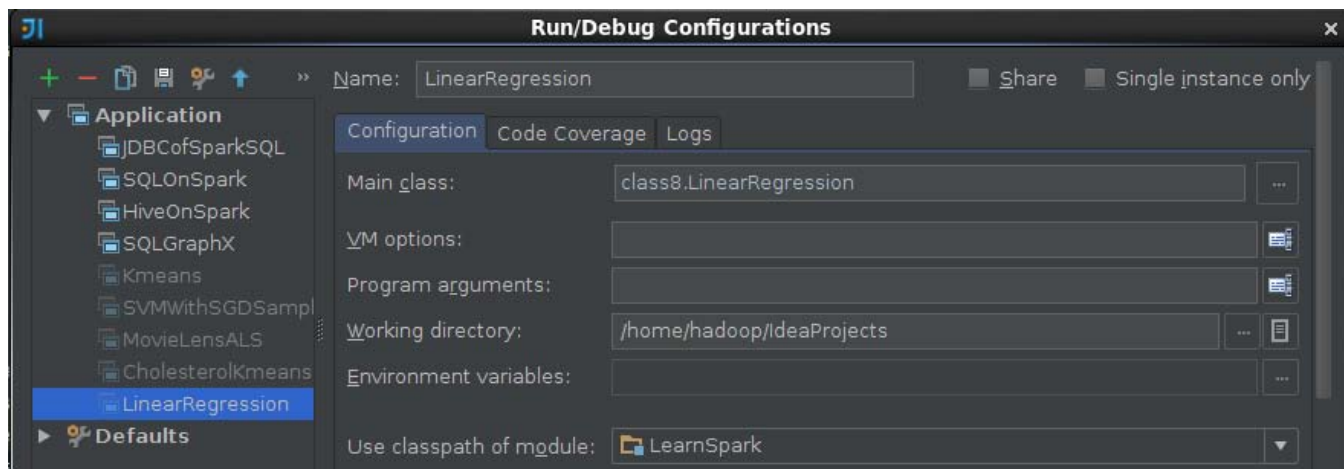
第一步 启动 Spark 集群

```
$cd /app/hadoop/spark-1.1.0
```

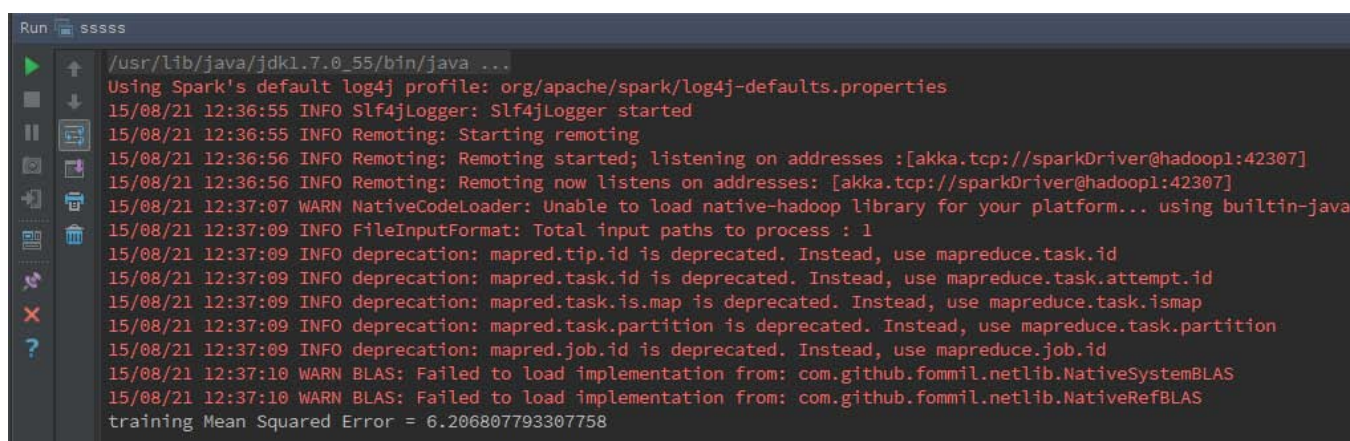
```
$sbin/start-all.sh
```

第二步 在 IDEA 中设置运行环境

在 IDEA 运行配置中设置 LinearRegression 运行配置，由于读入的数据已经在程序中指定，故在该设置界面中不需要设置输入参数



第三步 执行并观察输出



1.3 协同过滤实例

1.3.1 算法说明

协同过滤（Collaborative Filtering，简称 CF，WIKI 上的定义是：简单来说是利用某个兴趣相投、拥有共同经验之群体的喜好来推荐感兴趣的资讯给使用者，个人透过合作的机制给予资讯相当程度的回应（如评分）并记录下来以达到过滤的目的，进而帮助别人筛选资讯，回应不一定局限于特别感兴趣的，特别不感兴趣资讯的纪录也相当重要。

协同过滤常被应用于推荐系统。这些技术旨在补充用户—商品关联矩阵中所缺失的部分。

MLlib 当前支持基于模型的协同过滤，其中用户和商品通过一小组隐性因子进行表达，并且这些因子也用于预测缺失的元素。MLlib 使用交替最小二乘法（ALS）来学习这些隐性因子。

用户对物品或者信息的偏好，根据应用本身的不同，可能包括用户对物品的评分、用户查看物品的记录、用户的购买记录等。其实这些用户的偏好信息可以分为两类：

- **显式的用户反馈** 这类是用户在网站上自然浏览或者使用网站以外，显式地提供反馈信息，例如用户对物品的评分或者对物品的评论。
- **隐式的用户反馈** 这类是用户在使用网站是产生的数据，隐式地反映了用户对物品的喜好，例如用户购买了某物品，用户查看了某物品的信息，等等。

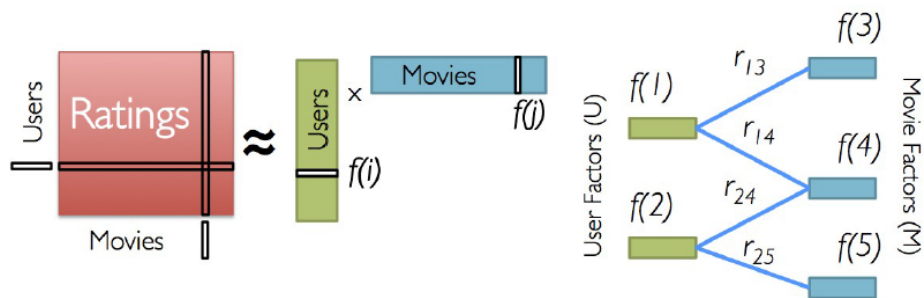
显式的用户反馈能准确地反映用户对物品的真实喜好，但需要用户付出额外的代价；而隐式的用户行为，通过一些分析和处理，也能反映用户的喜好，只是数据不是很精确，有些行为的分析存在较大的噪音。但只要选择正确的行为特征，隐式的用户反馈也能得到很好的效果，只是行为特征的选择可能在不同的应用中有很大的不同，例如在电子商务的网站上，购买行为其实就是一个能很好表现用户喜好的隐式反馈。

推荐引擎根据不同的推荐机制可能用到数据源中的一部分，然后根据这些数据，分析出一定的规则或者直接对用户对其他物品的喜好进行预测计算。这样推荐引擎可以在用户进入时给他推荐他可能感兴趣的物品。

MLlib 目前支持基于协同过滤的模型，在这个模型里，用户和产品被一组可以用来预测缺失项目的潜在因子来描述。特别是我们实现交替最小二乘（ALS）算法来学习这些潜在的因子，在 MLlib 中的实现有如下参数：

- **numBlocks** 是用于并行化计算的分块个数（设置为-1时 为自动配置）；
- **rank** 是模型中隐性因子的个数；
- **iterations** 是迭代的次数；
- **lambda** 是 ALS 的正则化参数；
- **implicitPrefs** 决定了是用显性反馈 ALS 的版本还是用隐性反馈数据集的版本；
- **alpha** 是一个针对于隐性反馈 ALS 版本的参数，这个参数决定了偏好行为强度的基准。

Low-Rank Matrix Factorization:



Iterate:

$$f[i] = \arg \min_{w \in \mathbb{R}^d} \sum_{j \in \text{Nbrs}(i)} (r_{ij} - w^T f[j])^2 + \lambda \|w\|_2^2$$

1.3.2 实例介绍

在本实例中将使用协同过滤算法对 GroupLens Research (<http://grouplens.org/datasets/movielens/>) 提供的数据进行分析，该数据为一组从 20 世纪 90 年末到 21 世纪初由 MovieLens 用户提供的电影评分数据，这些数据中包括电影评分、电影元数据（风格类型和年代）以及关于用户的人口统计学数据（年龄、邮编、性别和职业等）。根据不同需求该组织提供了不同大小的样本数据，不同样本信息中包含三种数据：评分、用户信息和电影信息。

对这些数据分析进行如下步骤：

1. 装载如下两种数据：
 - a) 装载样本评分数据，其中最后一列时间戳除 10 的余数作为 key，Rating 为值；
 - b) 装载电影目录对照表（电影 ID->电影标题）
2. 将样本评分表以 key 值切分成 3 个部分，分别用于训练（60%，并加入用户评分），校验（20%），and 测试（20%）
3. 训练不同参数下的模型，并再校验集中验证，获取最佳参数下的模型
4. 用最佳模型预测测试集的评分，计算和实际评分之间的均方根误差
5. 根据用户评分的数据，推荐前十部最感兴趣的电影（注意要剔除用户已经评分的电影）

1.3.3 测试数据说明

在 MovieLens 提供的电影评分数据分为三个表：评分、用户信息和电影信息，在该系列提供的附属数据提供大概 6000 位读者和 100 万个评分数据，具体位置为 /data/class8/movielens/data 目录下，对三个表数据说明可以参考该目录下 README 文档。

1. 评分数据说明 (ratings.data)

该评分数据总共四个字段 格式为 UserID::MovieID::Rating::Timestamp 分为为用户编号 :: 电影编号 :: 评分 :: 评分时间戳，其中各个字段说明如下：

- 用户编号范围 1~6040
- 电影编号 1~3952
- 电影评分为五星评分，范围 0~5
- 评分时间戳单位秒
- 每个用户至少有 20 个电影评分

使用的 ratings.dat 的数据样本如下所示：

```
1::1193::5::978300760
1::661::3::978302109
1::914::3::978301968
1::3408::4::978300275
1::2355::5::978824291
1::1197::3::978302268
1::1287::5::978302039
1::2804::5::978300719
```

2. 用户信息(users.dat)

用户信息五个字段，格式为 UserID::Gender::Age::Occupation::Zip-code，分为为用户编号 :: 性别 :: 年龄 :: 职业::邮编，其中各个字段说明如下：

- 用户编号范围 1~6040
- 性别，其中 M 为男性，F 为女性
- 不同的数字代表不同的年龄范围，如：25 代表 25~34 岁范围
- 职业信息，在测试数据中提供了 21 中职业分类
- 地区邮编

使用的 users.dat 的数据样本如下所示：

```
1::F::1::10::48067
2::M::56::16::70072
3::M::25::15::55117
```


4::M::45::7::02460
5::M::25::20::55455
6::F::50::9::55117
7::M::35::1::06810
8::M::25::12::11413

3. 电影信息(movies.dat)

电影数据分为三个字段，格式为 MovieID::Title::Genres，分为为电影编号 :: 电影名 :: 电影类别，其中各个字段说明如下：

- 电影编号 1~3952
- 由 IMDB 提供电影名称，其中包括电影上映年份
- 电影分类，这里使用实际分类名非编号，如：Action、Crime 等

使用的 movies.dat 的数据样本如下所示：

1::Toy Story (1995)::Animation/Children's/Comedy
2::Jumanji (1995)::Adventure/Children's/Fantasy
3::Grumpier Old Men (1995)::Comedy/Romance
4::Waiting to Exhale (1995)::Comedy/Drama
5::Father of the Bride Part II (1995)::Comedy
6::Heat (1995)::Action/Crime/Thriller
7::Sabrina (1995)::Comedy/Romance
8::Tom and Huck (1995)::Adventure/Children's

1.3.4 程序代码

```
import java.io.File
import scala.io.Source
import org.apache.log4j.{Level, Logger}
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.rdd._
import org.apache.spark.mllib.recommendation.{ALS, Rating, MatrixFactorizationModel}

object MovieLensALS {
```



```

def main(args: Array[String]) {
  // 屏蔽不必要的日志显示在终端上
  Logger.getLogger("org.apache.spark").setLevel(Level.WARN)
  Logger.getLogger("org.eclipse.jetty.server").setLevel(Level.OFF)

  if (args.length != 2) {
    println("Usage: /path/to/spark/bin/spark-submit --driver-memory 2g --class
    week7.MovieLensALS " +
      "week7.jar movieLensHomeDir personalRatingsFile")
    sys.exit(1)
  }

  // 设置运行环境
  val conf = new SparkConf().setAppName("MovieLensALS").setMaster("local[4]")
  val sc = new SparkContext(conf)

  // 装载用户评分，该评分由评分器生成
  val myRatings = loadRatings(args(1))
  val myRatingsRDD = sc.parallelize(myRatings, 1)

  // 样本数据目录
  val movieLensHomeDir = args(0)

  // 装载样本评分数据，其中最后一列 Timestamp 取除 10 的余数作为 key，Rating 为值，
  即(Int, Rating)
  val ratings = sc.textFile(new File(movieLensHomeDir, "ratings.dat").toString).map
  { line =>
    val fields = line.split("::")
    (fields(3).toLong % 10, Rating(fields(0).toInt, fields(1).toInt, fields(2).toDouble))
  }

  // 装载电影目录对照表 ( 电影 ID->电影标题 )
  val movies = sc.textFile(new File(movieLensHomeDir, "movies.dat").toString).map
  { line =>
    val fields = line.split("::")

```

```

(fields(0).toInt, fields(1))
}.collect().toMap

val numRatings = ratings.count()
val numUsers = ratings.map(_._2.user).distinct().count()
val numMovies = ratings.map(_._2.product).distinct().count()

println("Got " + numRatings + " ratings from " + numUsers + " users on " +
numMovies + " movies.")

// 将样本评分表以 key 值切分成 3 个部分，分别用于训练 (60%，并加入用户评分)，校验
// (20%)，and 测试 (20%)
// 该数据在计算过程中要多次应用到，所以 cache 到内存
val numPartitions = 4
val training = ratings.filter(x => x._1 < 6)
    .values
    .union(myRatingsRDD) //注意 ratings 是(Int,Rating)，取 value 即可
    .repartition(numPartitions)
    .cache()
val validation = ratings.filter(x => x._1 >= 6 && x._1 < 8)
    .values
    .repartition(numPartitions)
    .cache()
val test = ratings.filter(x => x._1 >= 8).values.cache()

val numTraining = training.count()
val numValidation = validation.count()
val numTest = test.count()

println("Training: " + numTraining + ", validation: " + numValidation + ", test: " +
numTest)

// 训练不同参数下的模型，并在校验集中验证，获取最佳参数下的模型
val ranks = List(8, 12)
val lambdas = List(0.1, 10.0)

```

```

val numIters = List(10, 20)
var bestModel: Option[MatrixFactorizationModel] = None
var bestValidationRmse = Double.MaxValue
var bestRank = 0
var bestLambda = -1.0
var bestNumIter = -1
for (rank <- ranks; lambda <- lambdas; numIter <- numIters) {
  val model = ALS.train(training, rank, numIter, lambda)
  val validationRmse = computeRmse(model, validation, numValidation)
  println("RMSE (validation) = " + validationRmse + " for the model trained with
    rank = "
      + rank + ", lambda = " + lambda + ", and numIter = " + numIter + ".")
  if (validationRmse < bestValidationRmse) {
    bestModel = Some(model)
    bestValidationRmse = validationRmse
    bestRank = rank
    bestLambda = lambda
    bestNumIter = numIter
  }
}

```

// 用最佳模型预测测试集的评分，并计算和实际评分之间的均方根误差

```

val testRmse = computeRmse(bestModel.get, test, numTest)

```

```

println("The best model was trained with rank = " + bestRank + " and lambda = " +
bestLambda + ", and numIter = " + bestNumIter + ", and its RMSE on the test set is
" + testRmse + ".")

```

// create a naive baseline and compare it with the best model

```

val meanRating = training.union(validation).map(_.rating).mean
val baselineRmse =

```

```

  math.sqrt(test.map(x => (meanRating - x.rating) * (meanRating - x.rating)).mean)

```

```

val improvement = (baselineRmse - testRmse) / baselineRmse * 100

```

```

println("The best model improves the baseline by " + "%1.2f".format(improvement)
+ "%.")

```

```

// 推荐前十部最感兴趣的电影，注意要剔除用户已经评分的电影
val myRatedMovieIds = myRatings.map(_product).toSet
val candidates =
sc.parallelize(movies.keys.filter(!myRatedMovieIds.contains(_)).toSeq)
val recommendations = bestModel.get
    .predict(candidates.map((0, _)))
    .collect()
    .sortBy(_rating)
    .take(10)

var i = 1
println("Movies recommended for you:")
recommendations.foreach { r =>
    println("%2d".format(i) + ": " + movies(r.product))
    i += 1
}

sc.stop()
}

/** 校验集预测数据和实际数据之间的均方根误差 */
def computeRmse(model: MatrixFactorizationModel, data: RDD[Rating], n: Long):
Double = {
    val predictions: RDD[Rating] = model.predict(data.map(x => (x.user, x.product)))
    val predictionsAndRatings = predictions.map(x => ((x.user, x.product), x.rating))
        .join(data.map(x => ((x.user, x.product), x.rating)))
        .values
    math.sqrt(predictionsAndRatings.map(x => (x._1 - x._2) * (x._1 - x._2)).reduce(_ + _) /
n)
}

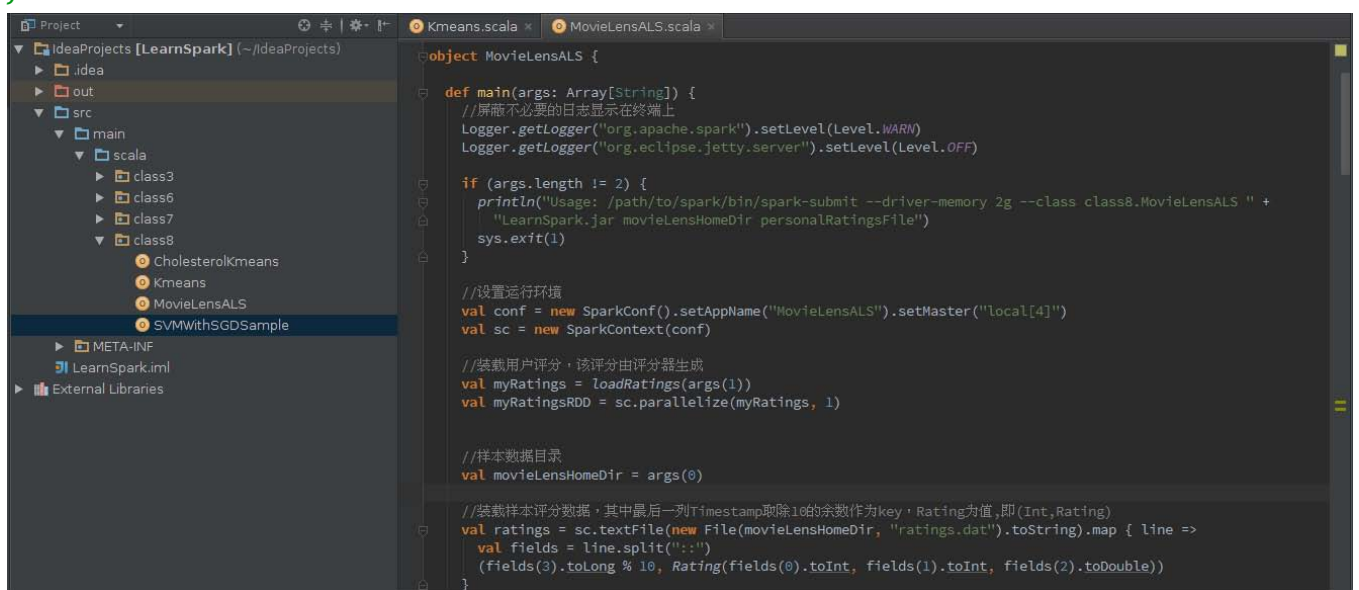
/** 装载用户评分文件 */
def loadRatings(path: String): Seq[Rating] = {
    val lines = Source.fromFile(path).getLines()

```

```

val ratings = lines.map { line =>
    val fields = line.split("::")
    Rating(fields(0).toInt, fields(1).toInt, fields(2).toDouble)
}.filter(_.rating > 0.0)
if (ratings.isEmpty) {
    sys.error("No ratings provided.")
} else {
    ratings.toSeq
}
}
}
}

```



1.3.5 IDEA 执行情况

第一步 使用如下命令启动 Spark 集群

```
$cd /app/hadoop/spark-1.1.0
```

```
$sbin/start-all.sh
```

第二步 进行用户评分，生成用户样本数据

由于该程序中最终推荐给用户十部电影，这需要用户提供对样本电影数据的评分，然后根据生成的最佳模型获取当前用户推荐电影。用户可以使用 /home/hadoop/upload/class8/movielens/bin/rateMovies 程序进行评分，最终生成 personalRatings.txt 文件：

```
hadoop1 | hadoop2 | hadoop3
[hadoop@hadoop1 ~]$ cd /home/hadoop/upload/class8/movielens
[hadoop@hadoop1 movielens]$ bin/rateMovies
Please rate the following movie (1-5 (best), or 0 if not seen):
Toy Story (1995): 5
Independence Day (a.k.a. ID4) (1996): 4
Dances with wolves (1990): 5
Star Wars: Episode VI - Return of the Jedi (1983): 3
Mission: Impossible (1996): 4
Ace Ventura: Pet Detective (1994): 5
Die Hard: With a Vengeance (1995): 5
Batman Forever (1995): 4
Pretty woman (1990): 3
Men in Black (1997): 5
Dumb & Dumber (1994): 3
[hadoop@hadoop1 movielens]$ ll
total 16
drwxrwxrwx 2 hadoop hadoop 4096 Mar 12 17:03 data
-rw-rw-r-- 1 hadoop hadoop 242 Aug 21 10:54 personalRatings.txt
-rwxr--r-- 1 hadoop hadoop 561 Jul 2 2014 personalRatings.txt.template
[hadoop@hadoop1 movielens]$
```

第三步 在 IDEA 中设置运行环境

在 IDEA 运行配置中设置 MovieLensALS 运行配置，需要设置输入数据所在文件夹和用户的评分文件路径：

- 输入数据所在目录：输入数据文件目录，在该目录中包含了评分信息、用户信息和电影信息，这里设置为/home/hadoop/upload/class8/movielens/data/
- 用户的评分文件路径：前一步骤中用户对十部电影评分结果文件路径，在这里设置为/home/hadoop/upload/class8/movielens/personalRatings.txt

第五步 执行并观察输出

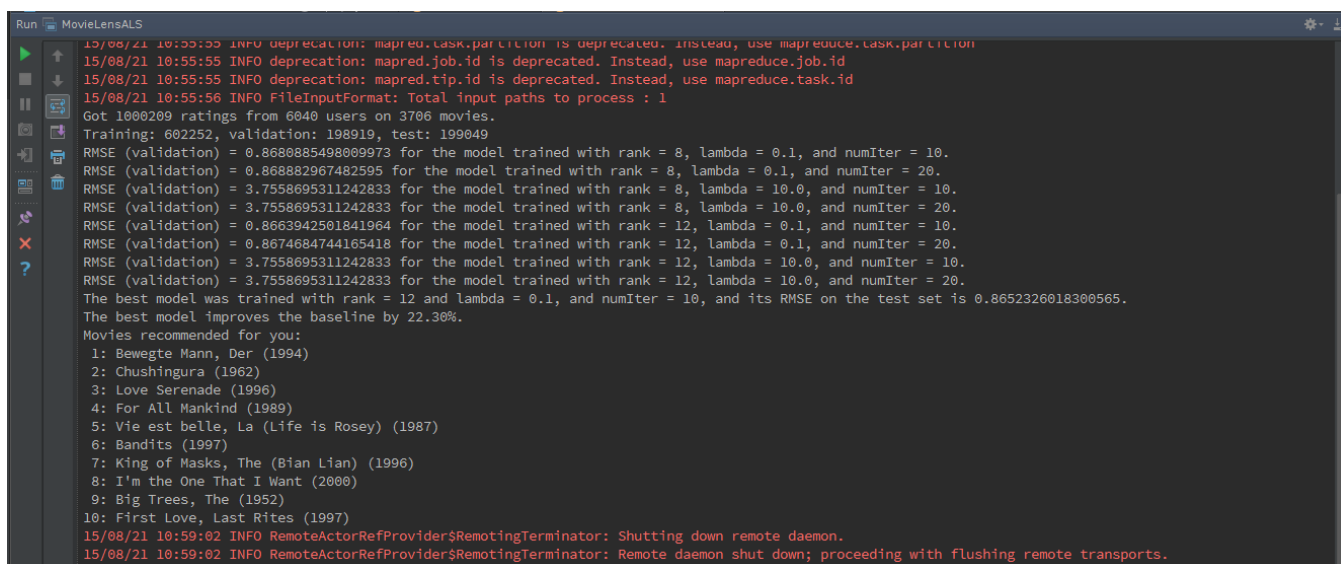
- 输出 *Got 1000209 ratings from 6040 users on 3706 movies*，表示本算法中计算数据包括大概 100 万评分数据、6000 多用户和 3706 部电影；
- 输出 Training: 602252, validation: 198919, test: 199049，表示对评分数据进行拆分为训练数据、校验数据和测试数据，大致占比为 6:2:2；
- 在计算过程中选择 8 种不同模型对数据进行训练，然后从中选择最佳模型，其中最佳模型比基准模型提供 22.30%

RMSE (validation) = 0.8680885498009973 for the model trained with rank = 8, lambda = 0.1, and numIter = 10.
RMSE (validation) = 0.868882967482595 for the model trained with rank = 8, lambda = 0.1, and numIter = 20.
RMSE (validation) = 3.7558695311242833 for the model trained with rank = 8, lambda = 10.0, and numIter = 10.
RMSE (validation) = 3.7558695311242833 for the model trained with rank = 8, lambda = 10.0, and numIter = 20.
RMSE (validation) = 0.8663942501841964 for the model trained with rank = 12, lambda = 0.1, and numIter = 10.
RMSE (validation) = 0.8674684744165418 for the model trained with rank = 12, lambda = 0.1, and numIter = 20.
RMSE (validation) = 3.7558695311242833 for the model trained with rank = 12, lambda = 10.0, and numIter = 10.
RMSE (validation) = 3.7558695311242833 for the model trained with rank = 12, lambda = 10.0, and numIter = 20.
The best model was trained with rank = 12 and lambda = 0.1, and numIter = 10, and its RMSE on the test set is 0.8652326018300565.
The best model improves the baseline by 22.30%.

- 利用前面获取的最佳模型，结合用户提供的样本数据，最终推荐给用户如下影片：

Movies recommended for you:

- 1: *Bewegte Mann, Der* (1994)
- 2: *Chushingura* (1962)
- 3: *Love Serenade* (1996)
- 4: *For All Mankind* (1989)
- 5: *Vie est belle, La (Life is Rosey)* (1987)
- 6: *Bandits* (1997)
- 7: *King of Masks, The (Bian Lian)* (1996)
- 8: *I'm the One That I Want* (2000)
- 9: *Big Trees, The* (1952)
- 10: *First Love, Last Rites* (1997)



```
Run MovieLensALS
15/08/21 10:55:55 INFO deprecation: mapred.task.partition is deprecated. Instead, use mapreduce.task.partition
15/08/21 10:55:55 INFO deprecation: mapred.job.id is deprecated. Instead, use mapreduce.job.id
15/08/21 10:55:55 INFO deprecation: mapred.tip.id is deprecated. Instead, use mapreduce.task.id
15/08/21 10:55:56 INFO FileInputFormat: Total input paths to process : 1
Got 1000209 ratings from 6040 users on 3706 movies.
Training: 602252, validation: 198919, test: 199049
RMSE (validation) = 0.8680885498009973 for the model trained with rank = 8, lambda = 0.1, and numIter = 10.
RMSE (validation) = 0.868882967482595 for the model trained with rank = 8, lambda = 0.1, and numIter = 20.
RMSE (validation) = 3.7558695311242833 for the model trained with rank = 8, lambda = 10.0, and numIter = 10.
RMSE (validation) = 3.7558695311242833 for the model trained with rank = 8, lambda = 10.0, and numIter = 20.
RMSE (validation) = 0.8663942501841964 for the model trained with rank = 12, lambda = 0.1, and numIter = 10.
RMSE (validation) = 0.8674684744165418 for the model trained with rank = 12, lambda = 0.1, and numIter = 20.
RMSE (validation) = 3.7558695311242833 for the model trained with rank = 12, lambda = 10.0, and numIter = 10.
RMSE (validation) = 3.7558695311242833 for the model trained with rank = 12, lambda = 10.0, and numIter = 20.
The best model was trained with rank = 12 and lambda = 0.1, and numIter = 10, and its RMSE on the test set is 0.8652326018300565.
The best model improves the baseline by 22.30%.
Movies recommended for you:
1: Bewegte Mann, Der (1994)
2: Chushingura (1962)
3: Love Serenade (1996)
4: For All Mankind (1989)
5: Vie est belle, La (Life is Rosey) (1987)
6: Bandits (1997)
7: King of Masks, The (Bian Lian) (1996)
8: I'm the One That I Want (2000)
9: Big Trees, The (1952)
10: First Love, Last Rites (1997)
15/08/21 10:59:02 INFO RemoteActorRefProvider$RemotingTerminator: Shutting down remote daemon.
15/08/21 10:59:02 INFO RemoteActorRefProvider$RemotingTerminator: Remote daemon shut down; proceeding with flushing remote transports.
```

2 参考资料

- (1) Spark 官网 mllib 说明 <http://spark.apache.org/docs/1.1.0/mllib-guide.html>
- (2) 《机器学习常见算法分类汇总》 <http://www.ctocio.com/hotnews/15919.html>