

1.实验介绍

1.1 实验目的

掌握排列组合与古典概型的概念，熟悉统计与分布、线性代数基础的相关知识，了解回归与拟合、聚类和相关分类的相关概念。

1.2 实验要求

- 1) 掌握
- 2) 熟悉
- 3) 熟悉
- 4) 了解

1.3 实验项目与课时安排

| 序号 | 实验名称 | 课时 |
|----|------|----|
| 1 | | |
| 2 | | |
| 3 | | |
| 4 | | |
| 5 | | |
| | 合计 | |

1. SparkMLlib 基础（4 课时）

1.1 Spark 的安装

1.1.1 Spark 的安装

Spark 的机器学习库 MLlib 组件属于 Spark 生态的一部分，MLlib 提供了常用机器学习算法的实现，包括聚类、分类、回归、协同过滤等，降低了机器学习的门槛，开发人员只要具备一定的理论知识就能进行机器学习的工作。学习 SparkMLlib 就要先安装 Spark。

为了方便 Spark 使用 Hadoop 的 HDFS 存取数据，可以采用和 Hadoop 一起安装的方法。当安装好 Spark 后，自带有 scala 环境。本教程使用的具体环境如下：

CentOS 7 64 位

Hadoop2.7.6

Java JDK 1.8

Spark2.1.0

推荐上述环境的配置在 VMWare 虚拟机中完成，虚拟机和 CentOS7 系统的安装不再赘述。

（1）Hadoop 的安装

创建 hadoop 用户，shell 命令如下：

```
$ su # 以 root 用户登录
```

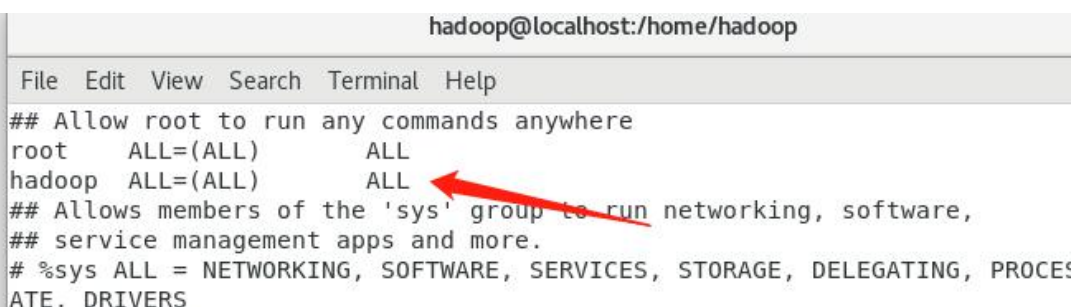
```
$ useradd -m hadoop -s /bin/bash # 创建新用户 hadoop
```

```
$ passwd hadoop #修改密码，按提示输入两次密码
```

为 hadoop 用户增加权限，shell 命令如下：

```
$ visudo
```

找到 root ALL=(ALL) ALL 一行，按 i 键，输入箭头所示一行内容：



```
hadoop@localhost:/home/hadoop
File Edit View Search Terminal Help
## Allow root to run any commands anywhere
root    ALL=(ALL)        ALL
hadoop  ALL=(ALL)        ALL
## Allows members of the 'sys' group to run networking, software,
## service management apps and more.
# %sys ALL = NETWORKING, SOFTWARE, SERVICES, STORAGE, DELEGATING, PROCES
ATE. DRIVERS
```

完成后，按下 ESC 键，输入:wq 保存退出。最后注销当前用户，重新以 hadoop 用户身份登陆。

（2）安装 Java 环境

Java 选择 OpenJDK，系统自带 Java JRE，需要通过 yum 安装 JDK，安装过程输入 y 即可。输入以下命令安装 OpenJDK，注意版本要与系统自带 java 版本对应

```
$ sudo yum install java-1.8.0-openjdk java-1.8.0-openjdk-devel
```

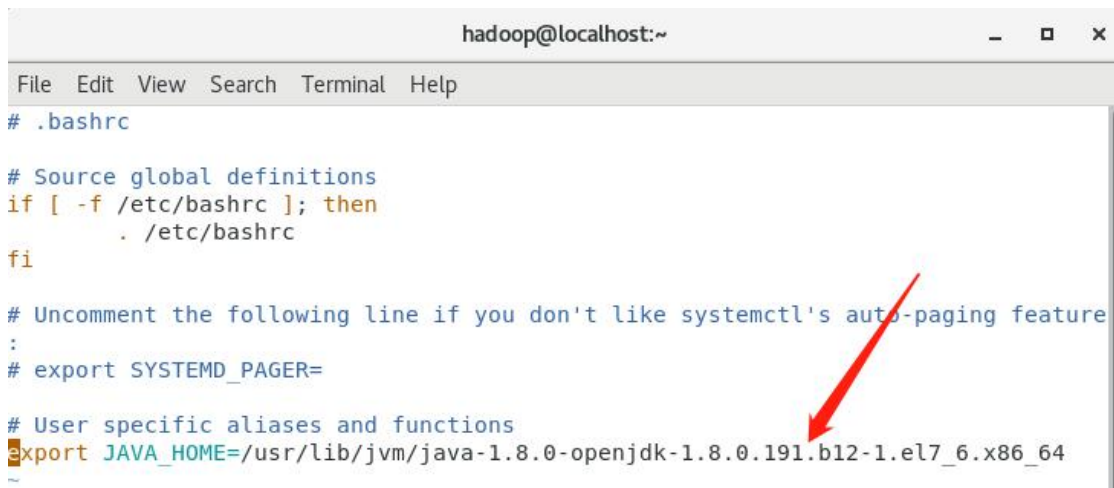
通过执行 `rpm -ql java-1.8.0-openjdk-devel | grep '/bin/javac'` 命令确定默认安装路径，执行后会输出一个路径，除去路径末尾的 “/bin/javac”，剩下的就是正确的路径了。输入以下命令配置 `JAVA_HOME` 环境变量：

```
$ vim ~/.bashrc
```

在最后一行添加如下代码指向 JDK 安装位置：

```
export JAVA_HOME=/usr/lib/jvm/java-1.8.0-openjdk
```

如图所示：



```
hadoop@localhost:~  
File Edit View Search Terminal Help  
# .bashrc  
  
# Source global definitions  
if [ -f /etc/bashrc ]; then  
    . /etc/bashrc  
fi  
  
# Uncomment the following line if you don't like systemctl's auto-paging feature  
:  
# export SYSTEMD_PAGER=  
  
# User specific aliases and functions  
export JAVA_HOME=/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.191.b12-1.el7_6.x86_64
```

输入以下命令使该环境生效并检验设置是否正确：

```
$ source ~/.bashrc    # 使变量设置生效
```

```
$ echo $JAVA_HOME    # 检验变量值
```

```
$ java -version
```

```
$ JAVA_HOME/bin/java -version # 与直接执行 java -version 一样
```

(3) 安装 Hadoop 2

Hadoop 2 可以通过 <http://mirror.bit.edu.cn/apache/hadoop/common/> 或者 <http://mirrors.cnnic.cn/apache/hadoop/common/> 下载，本教程选择的是 2.7.6 版本，下载时请下载 `hadoop-2.x.y.tar.gz` 这个格式的文件，这是编译好的，另一个包含 `src` 的则是 Hadoop 源代码，需要进行编译才可使用。

输入以下命令将 Hadoop 安装至 `/usr/local` 中，解压即可，并检查是否可用，成功会显示 Hadoop 的版本信息：

```
$ sudo tar -zxf ~/下载/hadoop-2.7.6.tar.gz -C /usr/local    # 解压到/usr/local  
中
```

```
$ cd /usr/local/
```

```
$ sudo mv ./hadoop-2.7.6/ ./hadoop    # 将文件夹名改为 hadoop
```

```
$ sudo chown -R hadoop:hadoop ./hadoop    # 修改文件权限
```

```
$ cd /usr/local/hadoop
```


```
$ ./bin/hadoop version    显示版本信息
```

(4) 安装 Spark

在 CentOS 中打开 Spark 官网 <http://spark.apache.org/downloads.html>，按下图所示结合自己 Hadoop 版本选择，点击箭头所指安装包下载即可，也可以在页面

最下方选择以前版本的下载。

Download Apache Spark™

1. Choose a Spark release: **2.2.3 (Jan 11 2019)** ▼
2. Choose a package type: **Pre-built for Apache Hadoop 2.7 and later** ▼
3. Download Spark: [spark-2.2.3-bin-hadoop2.7.tgz](#) 
4. Verify this release using the [2.2.3 signatures and checksums](#) and [project release KEYS](#).

这里使用 Local 模式即单机模式进行安装。

在 shell 中输入以下命令：

```
$ sudo tar -zxf ~/下载/spark-2.1.0-bin-without-hadoop.tgz -C /usr/local/
```

#此处路径为读者下载的实际路径

```
$ cd /usr/local
```

```
$ sudo mv ./spark-2.1.0-bin-without-hadoop/ ./spark
```

```
$ sudo chown -R hadoop:hadoop ./spark #此处的 hadoop 为用户名
```

输入以下命令修改 Spark 的配置文件 spark-env.sh

```
$ cd /usr/local/spark
```

```
$ cp ./conf/spark-env.sh.template ./conf/spark-env.sh
```

输入命令 vim ./conf/spark-env.sh 编辑 spark-env.sh 文件，在第一行添加以下配置信息：

```
export SPARK_DIST_CLASSPATH=$(/usr/local/hadoop/bin/hadoop classpath)
```

配置完成后，Spark 就可以用 Hadoop 的 HDFS 分布式文件系统读取存储数据了，否则只能在本地读写数据。

最后输入以下命令，运行 Spark 自带例子，检验安装是否成功：

```
$ cd /usr/local/spark
```

```
$ bin/run-example SparkPi
```

```
$ bin/run-example SparkPi 2>&1 | grep "Pi is"
```

结果可以得到圆周率的近似值，如下图所示：

```
19/01/23 23:28:32 INFO spark.SparkContext: Successfully stopped SparkContext
19/01/23 23:28:32 INFO util.ShutdownHookManager: Shutdown hook called
19/01/23 23:28:32 INFO util.ShutdownHookManager: Deleting directory /tmp/spark-8
75d3a2e-aef4-4aa0-b2ff-2d14ad3b1923
[hadoop@localhost spark]$ bin/run-example SparkPi 2>&1 | grep "Pi is"
Pi is roughly 3.1407357036785184 
[hadoop@localhost spark]$
```

1.1.2 使用 Spark 编写简单的应用程序

我们首先使用 spark shell 进行开发，这是一种交互式执行环境。输入语句立即执行并返回结果，可以实时查看中间结果，并进行修改。对初学者很友好。本教程使用 scala 进行讲解。

spark-shell 命令及其常用的参数如下：

```
./bin/spark-shell --master <master-url>
```

Master -url 决定了 Spark 的运行模式，在此处我们使用本地模式 local 运行。

- master: 这个参数表示当前的 Spark Shell 要连接到哪个 master，如果是 local[*]，就是使用本地模式启动 spark-shell，其中，中括号内的星号表示需要使用几个 CPU 核心(core)。当然在此处，我们可以直接用如下的命令启动：

此命令相当于 `bin/spark-shell -master local[*]`，使用本地所有 CPU 核心运行。启动以后，进入 `scala` 命令提示符状态，如图所示：

接下来就如同最初学习 C 语言，java 一样，可以试着用 scala 编写一些简单的程序。示例如下，最后使用:quit 退出 Spark shell。

```
scala> print("Hello world!")
Hello world!
scala> :quit
[hadoop@localhost spark]$
```

1.2.1 RDD 操作

Spark 基于 Scala 语言实现了 RDD 的应用程序接口 API, 用户可以直接调用 API 实现操作 RDD。执行过程如下:

- (1) RDD 读取数据并创建;
- (2) RDD 进行“转换”操作, 每次转换产生不同的 RDD, 留给下一次“转换”使用;
- (3) 最后一个 RDD 进行“行动”操作, 输出结果, 一般为数据或 Scala 集合或标量。

如下 ex1 为基于 RDD 进行一系列计算操作的 Spark 程序，计算 “Hello Spark” 中的个数：

```
//创建 SparkContext 对象
val sc= new SparkContext("spark://localhost:7077","Hello World",
"YOUR_SPARK_HOME","YOUR_APP_JAR")
//从 HDFS 文件读取数据创建 RDD
val fileRDD = sc.textFile("hdfs://192.168.0.103:9000/examplefile")
//对上一行产生的对象进行转换操作得到新 RDD
val filterRDD = fileRDD.filter(_.contains("Hello World"))
//对新产生的 RDD 持久化
filterRDD.cache()
//进行行动操作，根据之前的转换形成的依赖关系进行计算，输出元素个数
filterRDD.count()
```

1.2.2 数据处理

Spark 支持对多种常见类型的文件进行读写，除了本地文件系统，还有文本文件、JSON、SequenceFile 等，以及 HDFS、Amazon S3 等文件系统和 MySQL、HBase、Hive 等数据库。以下对数据处理部分进行简单的介绍，只涉及本地文件的读取。

在 CenrOs 中打开终端，输入以下指令打开 Spark-shell：

```
$ cd /usr/local/spark
$ ./bin/spark-shell
$ ....#中间为输出的信息
$ scala>
```

另打开一个终端进行文件编辑和查看。并输入以下命令，完成数据准备：

```
cd /usr/local/spark/mycode #进入对应目录，没有则用 mkdir 创建
mkdir wordcount #创建文件夹
echo "hadoop hello spark hello world" >> word.txt #把自己创建的数据存入 txt
数据准备完毕。
```

回到第一个终端，在 spark-shell 中输入如下指令，完成数据加载：

```
val textFile = sc.textFile("file:///usr/local/spark/mycode/wordcount/word.txt")
```

执行行动操作，即可看到输出：

```
textFile.first() #first()是一个 Action 类型操作，开始计算，输出加载的数据
```

接下来是把变量中的数据写入文件，代码如下：

```
textFile.saveAsTextFile("file:///usr/local/spark/mycode/wordcount/writeback.txt")
)#saveAsTextFile()是一个 Action 类型的操作，把变量中数据写入文件
```

查看 writeback.txt 文件

```
cd /usr/local/spark/mycode/wordcount/writeback.txt/
ls
```

可以看到三个文件如图所示：

```
[hadoop@localhost wordcount]$ cd /usr/local/spark/mycode/wordcount/writeback.txt  
/  
[hadoop@localhost writeback.txt]$ ls  
part-000000 part-000001 _SUCCESS  
[hadoop@localhost writeback.txt]$ █
```

用 cat 命令查看，与 word.txt 内容一样

然后用如下代码可以将数据再次加载回 RDD：

```
val textFile = sc.textFile("file:///usr/local/spark/mycode/wordcount  
/writeback.txt")
```

1.3 SparkMLlib 矩阵基础

Spark 中向量矩阵运算基础简介，MLlib 底层的向量矩阵运算使用了 Breeze 库，提供了 Vector/Matrix 的实现与计算接口。但是 MLlib 函数的参数传递使用自己的 Vector，在函数内的矩阵计算又使用 ToBreeze.ToDenseVector 变成 Breeze 形式计算，以此达到保证函数接口稳定性的目的。而 BLAS（基础线性代数程序集）是 API 标准，规范基础线性代数操作的数值库，分为向量-向量运算，矩阵-向量运算，矩阵-矩阵运算。

在 SparkMLlib 中均封装上述库，实现了自己的矩阵相关的类，以下做出介绍。

（1）MLlib 向量

封装了 Breeze 向量方法，实现了 Vector 类，包含 ToBreeze 方法，可将 Vector 类转换为 Breeze 向量。org.apache.spark.mllib.linalg.Vectors 定义了 Vector 接口，伴生对象，实现了 DenseVector、SparseVector 类。

Vector 接口最终由 DenseVector、SparseVector 实现稀疏向量和密集向量，主要方法有：

- toArray：向量转数组；
- toBreeze：向量转 Breeze 向量；
- toSparse：转稀疏向量；
- toDense：转密集向量；
- copy：向量复制。

（2）MLlib 矩阵

封装了 Breeze 矩阵方法实现了自己的 Matrix 类，与 Vector 类相似，也定义了接口：Matrix，静态对象 Matrixs，实现了 DenseVector、SparseVector 类。主要方法有：

- toArray：矩阵转密集数组；
- toBreeze：向量转 Breeze 矩阵；
- transpose：矩阵转置；
- multiply：矩阵乘法。

2. SparkMLlib 回归算法（2 课时）

2.1 SparkMLlib 线性回归算法

2.1.1 线性回归算法

线性回归 **Linear Regression** 利用最小均方函数对一个或多个自变量和因变量之间的关系进行建模并分析。这种函数是线性回归方程，是一个或多个由回归系数作为模型参数的线性组合。在这种回归分析中，分为一元线性回归分析和多元线性回归分析，一元是指只包括一个自变量和一个因变量，且二者关系能用一条直线近似，二元是指包括两个或两个以上的自变量，且与因变量之间是线性关系。

而两种分析都可以用下面的形式统一表示：

$$h_{\theta}(x) = \sum_{i=0}^n \theta_i x_i = \theta^T X$$

θ 就是回归系数等式最后表示回归系数矩阵与 x 矩阵相乘，我们的目标就是求取这个回归系数矩阵。

在此我们如下的损失函数去评估所求的回归系数是否最优，也就是描述我们预测的结果有多好：

$$J(\theta) = \frac{1}{2} \sum_{i=0}^n (h_{\theta}(x_i) - y_i)^2$$

损失函数是预测值与真实值之差的平方和， $\frac{1}{2}$ 的系数是为了后续求导的方便。

现在的目标就是寻找使得损失函数最小的回归系数 θ 。常用的方法有梯度下降法和最小二乘法。同时，回归分析中也常常会存在过拟合的问题，这是由于训练样本过于侧重某些点，训练得到的曲线对待测样本数据的预测精度会下降，即拟合程度不够。通常的做法是对数据进行正则化处理。其基本思想是防止由拟合算法得到的回归公式对某些特定的特征值更加敏感，对增加的系数使其系数为零进行消除从而消除过拟合。

2.1.2 算法源码分析

MLlib 中用随机梯度下降算法优化损失函数，梯度下降的意义就是，用损失函数对参数求偏导数，也就是梯度，沿着梯度方向，损失函数减小的最快，当达到收敛时，也就求到了最优的参数。此处的随机梯度下降方法是：进行多轮迭代计算，每一次随机抽取一定样本进行计算；对所计算的每一个样本计算梯度；再将样本梯度进行累加，求得平均梯度和损失；最后根据这次的梯度和上一次迭代的权重对梯度和权重进行更新。

实现过程如下：

(1) 建立线性回归模型

建立模型的入口是线性回归伴生对象 **LinearRegressionWithSGD**，定义了训练模型的 **train** 方法，设置训练参数进行模型训练，主要参数如下：

***Input**——标签和特征序列组成的 **RDD** 训练样本，每对有一列数据和其对应的标签；

***numiterations**——要运行的渐变下降迭代次数，默认 100；

***step size**——步进大小，用于每次梯度下降迭代，默认 1；

*minibatchfraction——每次迭代使用的样本比例，默认 1，即全部样本
*weights 要使用的初始权重集，数组的大小应等于数据中的特征数量。

线性回归类 LinearRegressionWithSGD 是基于梯度下降法生成线性回归模型，完成了初始化梯度下降、梯度更新、优化计算等功能，根据初始化的方法调用集成了广义回归类 GeneralizedLinearAlgorithm 的 run 方法训练模型。

(2) run 方法训练模型

此方法首先对样本加偏置，然后进行初始化权重，调用 optimizer.optimize 方法进行权重优化，最后获取最优权重。

(3) 权重优化

optimizer.optimize 方法是调用了 GradientDescent 伴生对象的 runMiniBatchSGD 方法来实现最终梯度下降算法的实现，通过迭代计算样本的梯度及误差返回最优梯度和误差。而 gradient.compute 基于最小二乘法完成了每个样本的梯度及误差计算。最后使用 updata.compute 进行权重更新。

(4) 模型生成

模型训练完成后，生成线性回归模型类，参数包含：各个特征的权重向量及偏置，可以完成预测、保存模型、加载模型等方法。

2.1.3 应用实战

(1) 以下是根据一个应用 LinearRegression 进行数据处理的实例，用到的训练数据格式如下：

```
-9.490009878824548      1:0.4551273600657362      2:0.36644694351969087
3:-0.38256108933468047  4:-0.4458430198517267    5:0.33109790358914726
6:0.8067445293443565    7:-0.2624341731773887    8:-0.44850386111659524
9:-0.07269284838169332 10:0.5658035575800715
```

第一个是数据的标签，“:”前的数字是特征的维数序号，后边的数字是具体的特征值，一共是十维。本例的目标是根据训练数据训练一个正则化的线性模型，并且提取一些模型的概要统计量。

(2) 代码详解

第一步是导入 MLlib 中线性模型的 API：

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

第二步加载我们上文中所述的训练数据：

```
val training = spark.read.format("libsvm")
.load("/mnt/hgfs/thunderdownload/MLlib_rep/data/sample_linear_regression_data.
txt")
```

第三步是设置模型的参数：

```
val lr = new LinearRegression().setMaxIter(10).setRegParam(0.3)
.setElasticNetParam(0.8)
```

第四步是将数据输入设置好的模型：

```
val lrModel = lr.fit(training)
```

第五步就是输出根据训练数据得到的线性模型的回归系数和截距:

```
println(s"Coefficients: ${lrModel.coefficients} Intercept: ${lrModel.intercept}")
```

输出结果如下:

```
Coefficients: [0.0,0.32292516677405936,-0.3438548034562218,1.9156017023458414,0.05288058680386263,0.765962720459771,0.0,-0.15105392669186682,-0.2158793036090464,2,0.22025369188813426] Intercept: 0.1598936844239736
```

最后就是概括模型在训练集上的表现, 并且打印出模型的训练结果参数:

```
val trainingSummary = lrModel.summary
```

```
//模型的迭代次数
```

```
println(s"numIterations: ${trainingSummary.totalIterations}")
```

```
//每次迭代的目标函数的结果, 即标度损失+正则化
```

```
println(s"objectiveHistory: [${trainingSummary.objectiveHistory.mkString(",")}"])
```

```
numIterations: 7
objectiveHistory: [0.49999999999999994,0.4967620357443381,0.4936361664340463,0.4936351537897608,0.4936351214177871,0.49363512062528014,0.4936351206216114]
```

```
//输出预测标签值的残差
```

```
trainingSummary.residuals.show()
```

```
+-----+
| residuals |
+-----+
| -9.889232683103197 |
| 0.5533794340053554 |
| -5.204019455758823 |
| -20.566686715507508 |
| -9.4497405180564 |
| -6.909112502719486 |
| -10.00431602969873 |
| 2.062397807050484 |
| 3.1117508432954772 |
| -15.893608229419382 |
| -5.036284254673026 |
| 6.483215876994333 |
| 12.429497299109002 |
| -20.32003219007654 |
| -2.0049838218725005 |
| -17.867901734183793 |
| 7.646455887420495 |
| -2.2653482182417406 |
| -0.10308920436195645 |
| -1.380034070385301 |
+-----+
```

//输出均方根误差和相关系数, 数值越小, 预测模型对实验数据的预测精度更高

```
println(s"RMSE: ${trainingSummary.rootMeanSquaredError}")
```

```
println(s"r2: ${trainingSummary.r2}")
```

```
RMSE: 10.189077167598475  
r2: 0.022861466913958184
```

3. SparkMLlib 分类算法（8 课时）

3.1 SparkMLlib 贝叶斯分类算法（2 课时）

3.1.1 贝叶斯分类算法

首先简要介绍一下贝叶斯算法的重要定理——贝叶斯定理。

$P(A)$ 是随机事件 A 的先验概率， $P(A|B)$ 是已知随机事件 B 发生后 A 的条件概率，也称作 A 的后验概率。随机事件 B 同理。有公式：

$$P(A|B) = \frac{P(AB)}{P(B)}$$

也就引出了贝叶斯定理：

$$P(B|A) = \frac{P(A|B)P(B)}{P(A)}$$

基于此，我们在这里提出了一种简单的分类器——朴素贝叶斯分类器。对于待分类样本，计算在此样本发生时各个类别出现的概率，找出最大的概率，就可以判定这个待分类样本属于哪个类别。朴素贝叶斯的分类过程包括了数据准备，训练和预测阶段。分类过程如下：

（1）定义待分类项 x 的特征向量 (a_1, a_2, \dots, a_k) ，分类的类别为 (y_1, y_2, \dots, y_m) ；

（2）计算 $P(y_j|x)$ 即 x 发生时，属于 y_1, y_2, \dots, y_m 的概率分别为多少，根据贝叶斯公式可将此问题转化为求待分类样本属于具体分类类别的概率 $p(x|y_j)p(y_j)$ ，即

$$P(y_j|x) = \frac{P(x|y_j)P(y_j)}{P(x)}$$

（3）再根据条件概率公式：

$$P(x|y_j)P(y_j) = \prod P(a_i|y_j)P(y_j)$$

依据此概率最大项来判断 x 所属的类别。

3.1.2 算法源码分析

MLlib 中的贝叶斯分类模型是基于朴素贝叶斯，先计算各个类别的先验概率和各类别下各个特征的条件概率。其实现的步骤如下：

（1）对训练样本统计所有标签出现次数和对应特征之和；

（2）对（标签，样本特征）形式的样本进行聚合操作，统计属于同一标签的数据；

（3）由以上的统计结果计算先验概率和条件概率，得到朴素贝叶斯分类样本；

（4）进行预测，根据模型的先验和条件概率来计算待测样本属于每个类别的概率，取最大概率作为分类依据。

MLlib 中对朴素贝叶斯的实现如下：

(1) 贝叶斯分类伴生对象: **NativeBayes**, 含有 **train** 静态方法, 可以设置参数创建朴素贝叶斯分类类, 执行 **run** 方法进行训练, **train** 方法主要参数如下:

* **input**——训练样本格式为 **RDD(label,features)**;

* **lambda**——平滑参数,

(2) 贝叶斯分类模型: **NativeBayes** 类, 含有 **run** 方法, 训练贝叶斯模型, 计算各个类别的先验概率和各个特征属于各个类别的条件概率;

(3) 训练模型: **aggregated** 对样本各个类别下每一个特征值之和和次数进行统计, **pi** 根据统计结果计算各类别先验概率, **theta** 根据统计结果计算各个特征属于各个类别的条件概率;

(4) 贝叶斯模型: **NativeBayesModel** 类, 由模型的先验和条件概率计算样本属于各个类别的概率, 以最大概率作为判决依据, 主要参数包括:

* **labels**——类别标签列表

* **pi**——类别先验概率

* **theta**——各个特征在各个类别中的条件概率

* **modelType**——多项式或伯努利模型

含有 **predict** 方法根据贝叶斯分类模型返回样本预测值 **RDD[Double]**类型。

3.1.3 应用实战

(1) 本次实战选择是 **sample_libsvm_data** 数据, 以后也会经常用到, 其样本格式为:

label index1:value1 index2:value2 index3:value3 ...

label 是样本的标签值, 一对 **index:value** 表示一个特征和特征值。

(2) 代码详解

//首先导入用到的机器学习包, **NativeBayes** 算法包和多分类器评价包

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

```
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator
```

//以 **DataFrame** 形式加载以 **libsvm** 格式存储的数据

```
val data = spark.read.format("libsvm").load("/mnt/hgfs/thunder-  
download/Mllib_rep/data/sample_libsvm_data.txt")
```

输出结果为:

```
org.apache.spark.sql.DataFrame = [label: double, features: vector]
```

//把数据切分为训练集和测试集, 比例为 7:3

```
val Array(trainingData, testData) = data.randomSplit(Array(0.7, 0.3), seed = 1234L)
```

输出结果为:

```
trainingData: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [label:  
double, features: vector]
```

```
testData: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [label: double,  
features: vector]
```

//加载训练数据, 训练一个朴素贝叶斯模型

```
val model = new NaiveBayes().fit(trainingData)
```

```
//展示预测结果
```

```
val predictions = model.transform(testData)
```

```
predictions.show()
```

输出预测结果如下：

| label | features | rawPrediction | probability | prediction |
|-------|----------------------|----------------------|-------------|------------|
| 0.0 | (692,[95,96,97,12... | [-173678.60946628... | [1.0,0.0] | 0.0 |
| 0.0 | (692,[98,99,100,1... | [-178107.24302988... | [1.0,0.0] | 0.0 |
| 0.0 | (692,[100,101,102... | [-100020.80519087... | [1.0,0.0] | 0.0 |
| 0.0 | (692,[124,125,126... | [-183521.85526462... | [1.0,0.0] | 0.0 |
| 0.0 | (692,[127,128,129... | [-183004.12461660... | [1.0,0.0] | 0.0 |
| 0.0 | (692,[128,129,130... | [-246722.96394714... | [1.0,0.0] | 0.0 |
| 0.0 | (692,[152,153,154... | [-208696.01108598... | [1.0,0.0] | 0.0 |
| 0.0 | (692,[153,154,155... | [-261509.59951302... | [1.0,0.0] | 0.0 |
| 0.0 | (692,[154,155,156... | [-217654.71748256... | [1.0,0.0] | 0.0 |
| 0.0 | (692,[181,182,183... | [-155287.07585335... | [1.0,0.0] | 0.0 |
| 1.0 | (692,[99,100,101,... | [-145981.83877498... | [0.0,1.0] | 1.0 |
| 1.0 | (692,[100,101,102... | [-147685.13694275... | [0.0,1.0] | 1.0 |
| 1.0 | (692,[123,124,125... | [-139521.98499849... | [0.0,1.0] | 1.0 |
| 1.0 | (692,[124,125,126... | [-129375.46702012... | [0.0,1.0] | 1.0 |
| 1.0 | (692,[126,127,128... | [-145809.08230799... | [0.0,1.0] | 1.0 |
| 1.0 | (692,[127,128,129... | [-132670.15737290... | [0.0,1.0] | 1.0 |
| 1.0 | (692,[128,129,130... | [-100206.72054749... | [0.0,1.0] | 1.0 |
| 1.0 | (692,[129,130,131... | [-129639.09694930... | [0.0,1.0] | 1.0 |
| 1.0 | (692,[129,130,131... | [-143628.65574273... | [0.0,1.0] | 1.0 |
| 1.0 | (692,[129,130,131... | [-129238.74023248... | [0.0,1.0] | 1.0 |

only showing top 20 rows

```
//计算模型的准确度
```

```
val evaluator = new MulticlassClassificationEvaluator()
```

```
.setLabelCol("label")
```

```
.setPredictionCol("prediction")
```

```
.setMetricName("accuracy")
```

```
val accuracy = evaluator.evaluate(predictions)
```

```
println("Test set accuracy = " + accuracy)
```

输出结果，预测精度为 100%：

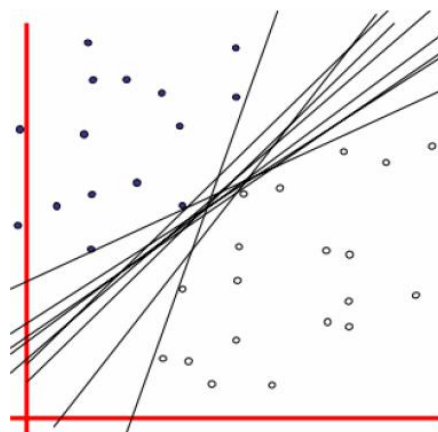
```
Test set accuracy = 1.0
```

3.2SparkMLlib SVM 支持向量机算法（2 课时）

3.2.1 支持向量机算法

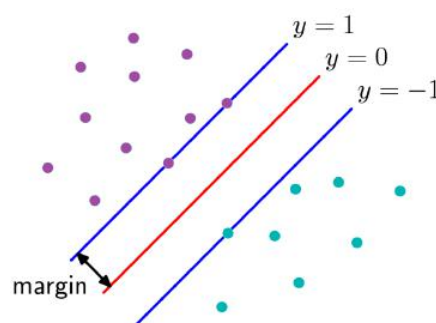
支持向量机是数据挖掘中一个很经典的算法，因为其推导过程涉及很多数学概念且其核函数的变化，在此将用尽量通俗的语言来描述这一算法，从其功能性出发进行讲解。支持向量机不仅对分类问题有良好的处理效果，对回归问题也有很好的解决方案。

SVM 分类器可以在样本空间中对属于不同类别的样本进行区分，用来作为区分的分隔面就是分隔超平面。对于一个 **SVM** 算法，输入带有标签的训练样本，输出的是一个最好的分隔超平面。如下图所示：



这是一个二维平面上属于两个类别上的点，我们试图寻找一条直线进行分隔，可以发现有很多条直线满足要求。但是一条直线如果距离样本太近，会对数据中的噪声敏感性高，就会导致泛化能力下降，所以我们的目标是找到一条尽可能远离所有样本点的直线。

支持向量机的实质就是找出一个分隔超平面使得该超平面距离所有训练样本（这里的点可以简化为是离超平面最近的点）的最小距离在所有分隔面中是最大的，这个最小距离叫做间隔 **margin**，这些距离超平面最近的点叫做支持向量，该超平面也就是最优超平面。



如上图所示，我们的目标就是在超平面将所有样本正确分类的前提下，求出这个最优超平面的 **margin**，然后最大化这个 **margin**。这个问题可以转化为一个拉格朗日优化问题，可以使用拉格朗日乘数法得到最优超平面的相关参数。这里就不再赘述正面过程。此外 **svm** 还可以通过改变核函数，将原始空间线性不可分的数据映射到高维空间变成线性可分的。

3.2.2 算法源码分析

在 MLlib 中，实现了线性 SVM 分类模型，使用随机梯度下降算法来优化目标函数。其实现过程如下：

(1) SVMWithSGD:SVM 线性分类伴生对象：基于随机梯度下降，含有 train 静态方法，设置 SVM 分类参数创建 SVM 分类，执行 run 方法训练模型，train 方法主要参数如下：

- * input——训练样本格式为 RDD(label,features)
- * numIterations——运行的迭代次数，默认 100
- * stepSize——每次迭代的步长，默认 1
- * miniBatchFraction——每次迭代参与计算的样本比例，默认 1
- * initialWeights——初始化权重

(2) SVMWithSGD 类：SVM 线性分类，包含 run 方法，先对样本加偏置再初始化权重，最后用 optimizer.optimize 方法计算优化权重从而获取最优权重；

(3) runMiniBatchSGD：优化计算权重：根据训练样本迭代计算随机梯度从而得到最优权重，每次迭代计算样本梯度并更新，分别使用 gradient.compute 和 updater.compute 方法；

(4) SVMModel：SVM 线性分类模型，含有 predict 方法，根据分类模型计算样本预测值，返回 RDD[double]。

3.2.3 应用实战

(1) 此次实战使用 sample_libsvm_data 样本数据。

(2) 代码详解

//导入需要的包文件

```
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.mllib.classification.{SVMModel, SVMWithSGD}
import org.apache.spark.mllib.evaluation.BinaryClassificationMetrics
import org.apache.spark.mllib.util.MLUtils
```

//构建 Spark 对象

```
val conf = new SparkConf().setAppName("SVM")
val sc = new SparkContext(conf)
```

//读取样本数据

```
val data = MLUtils.loadLibSVMFile(sc, "/mnt/hgfs/thunder-
download/MLlib_rep/data/sample_libsvm_data.txt")
```

//对样本数据划分为训练与测试样本，比例为 6:4

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

//训练样本

```
val training = splits(0).cache()
```

输出结果：

```
training: org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.LabeledPoint]
```


= MapPartitionsRDD[112] at randomSplit at <console>:37

//测试样本

```
val test = splits(1)
```

输出结果:

```
test: org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.LabeledPoint] =  
MapPartitionsRDD[113] at randomSplit at <console>:37
```

//配置训练参数

```
val numIterations = 100
```

输出结果: numIterations: Int = 100

//训练模型

```
val model = SVMWithSGD.train(training, numIterations)
```

输出结果:

```
model: org.apache.spark.mllib.classification.SVMModel =  
org.apache.spark.mllib.classification.SVMModel: intercept = 0.0, numFeatures = 692,  
numClasses = 2, threshold = 0.0
```

//对测试样本进行预测

```
val scoreAndLabels = test.map { point =>  
    | val score = model.predict(point.features)  
    | (score, point.label)  
    | }
```

输出结果:

```
scoreAndLabels: org.apache.spark.rdd.RDD[(Double, Double)] =  
MapPartitionsRDD[317] at map at <console>:47
```

返回数据类型 RDD

//计算误差

```
val metrics = new BinaryClassificationMetrics(scoreAndLabels)
```

```
val auROC = metrics.areaUnderROC()
```

```
println("Area under ROC = " + auROC)
```

输出结果:

```
Area under ROC = 1.0
```

分类精度 100%

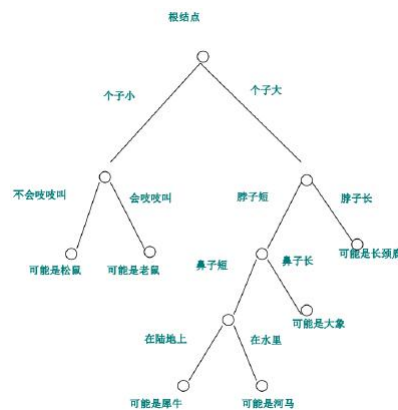
3.3SparkMLlib 决策树算法（2 课时）

3.3.1 决策树算法

决策树 `DecisionTree` 的结构是树型的，由节点和有向边组成。节点由内部节点和叶子节点组成，内部节点表示一个特征的度量，叶子节点表示一个具体的分类，每个分支表示度量的输出结果。决策树算法采用的是自顶向下的递归方法，

其基本思想是以信息熵为度量构造一颗熵值减少最快的树，当到达叶子节点出时熵为零，这事就可以判定每个叶子节点中的样本都属于同一类别。

决策树算法进行学习的过程其实是采用启发式的方法从训练数据集上归纳一组分类规则，求出局部最优即可。每次在内部节点进行度量时都选择最优的那个特征作为划分依据。通常分为三个步骤：特征选择，决策树生成和剪枝。下图就是一个决策树的例子：



（1）特征选择

特征选择的目的是找出局部最优特征，依据这个特征对样本数据进行分类达到最好的效果。这个分类的好坏取决于分类后的节点数据集合有序程度是否更好。分类后数据越有序，这个氟利昂就越好。在使用决策树进行分类的时候，通常选择信息熵作为节点数据有序性的度量。信息熵就是描述信息量的混乱程度，熵越小，信息越有序，我们使用信息增益来表示信息熵的改变。

（2）生成决策树

决策树的生成算法有 ID3 和 C4.5。ID3 算法是基于信息熵来构建决策树，是一种贪心算法。在每个节点选择还没被用来作为划分依据的、有最高信息增益的特征来作为划分标准，也就是以信息熵下降最快作为选取度量特征的标准，然后迭代重复这个过程，指导最后的叶子节点。其核心就是信息增益的计算，也就是一个事件中前后发生的不同信息之间的差值。C4.5 算法使用了信息增益比作为度量标准，克服了 ID3 划分过于充分的缺点。

（3）剪枝

与其他的机器学习算法类似，决策树也存在过拟合的问题，可以通过剪枝来提高泛化能力，就是在决策树模型对训练数据的预测误差和树的复杂度之间寻找一个最优的解决方法。提出了一个由预测误差和树复杂度组成的损失函数，剪枝算法从叶子节点往上寻找，比较剪掉该叶子节点前后损失函数的变化，如果剪掉后损失函数更小，就进行剪枝。

3.3.2 算法源码分析

MLlib 中的决策树分类模型可以完成二元及多元分类任务。Split 为分裂点，bin 为划分数。一个 split 把数据集分为两部分，最后统计产生多少个划分数。在使用剪枝方面，MLlib 采用前向剪枝，即提前设定参数，当满足一定条件时候就停止构建树，主要是树的最大深度：maxDepth，每个字节节点最少的训练样本数：minInstancePerNode，最小的信息增益：minInfoGain。

MLlib 中决策树模型的构建有以下几个部分：

(1) **DecisionTree**: 决策树伴生对象, 包括了 **train**, **trainClassifier** 方法, 设置决策树参数, 新建决策树类, 执行 **run** 方法训练, **train** 方法参数如下:

- * **input**: 训练数据集, 格式为 **RDD(label,features)**;
- * **algo**: 回归或分类算法;
- * **impurity**: 有序度计算方法, 也就是信息增益的计算;
- * **numClasses**: 分类数量, 默认 2;
- * **maxBins**: 分裂特征的最大划分数量。

本教程使用 **train** 方法或 **trainClassifier** 方法均可进行分类器的训练。

(2) **DecisionTree** 类: 包含 **run** 方法;

(3) **RandomForest** 类: 一棵树的随机森林就是决策树;

(4) 决策树训练准备:

建立原数据——**buildMeatdata**, 包括特征数量、样本数量、分类数量、划分数、分类特征信息、无序特征信息等;

查找特征的分裂与划分——使用 **findSplitsBins** 找出每个特征可能出现的 **split** 和 **bin**;

转换样本为 **TreePoint** 格式即把样本点按分裂点条件分到所属的 **bin** 中;

(5) 训练决策树模型

取得决策树分裂所需的节点;

查找最优的分裂顺序;

(6) 决策树模型: **DecisionTreeModel** 类, 包含 **predict** 方法。

3.3.3 应用实战

(1) 本次实战所用数据是 **iris** 数据集, 以鸢尾花的特征作为数据来源, 包含了 150 个数据样本, 分为 3 类, 每个类别 50 个数据样本, 每条数据样本有 4 个维度的特征和 1 个分类标签, 决策树可以用来做分类和回归, 这个实例将依据这个数据集做出分类和回归两个模型。数据示例如下:

5.1,3.5,1.4,0.2,Iris-setosa

4.9,3.0,1.4,0.2,Iris-setosa

4.7,3.2,1.3,0.2,Iris-setosa

4.6,3.1,1.5,0.2,Iris-setosa

5.0,3.6,1.4,0.2,Iris-setosa

(2) 代码详解

首先进行数据的处理, 对标签和特征进行处理并分组:

//导入需要的机器学习包

```
import org.apache.spark.sql.Session
```

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

```
import org.apache.spark.ml.Pipeline
```

```
import org.apache.spark.ml.feature.{IndexToString, StringIndexer, VectorIndexer}
```

```
//读取数据
```

```
//导入 spark.implicits._, 将 RDD 数据转换为 DataFrame
```

```
import spark.implicits._
```

```
//定义一个 schema: Iris
```

```
case class Iris(features: org.apache.spark.ml.linalg.Vector, label: String)
```

```
//从文件中读取数据，第一个 map 用 “,” 把每行数据分开，前四部分是特征，  
//最后是花的分类。
```

```
//把这些特征存储在 Vector 中，创建为 Iris(RDD)，再转化为 DataFrame
```

```
val data = spark.sparkContext.textFile("file:///mnt/hgfs/thunder-  
download/MMLib_rep/data/iris.txt").map(_._split(",")).map(p =>  
Iris(Vectors.dense(p(0).toDouble,p(1).toDouble,p(2).toDouble,  
p(3).toDouble),p(4).toString))).toDF()
```

```
输出结果如下：
```

```
data: org.apache.spark.sql.DataFrame = [features: vector, label: string]
```

```
//将之前得到的数据注册成表 iris
```

```
data.createOrReplaceTempView("iris")
```

```
//使用 sql 查询语句查询数据
```

```
val df = spark.sql("select * from iris")
```

```
//打印出查询的所有数据
```

```
df.map(t => t(1)+":"+t(0)).collect().foreach(println)
```

```
输出结果为：
```

```
Iris-setosa:[5.1,3.5,1.4,0.2]
```

```
Iris-setosa:[4.9,3.0,1.4,0.2]
```

```
Iris-setosa:[4.7,3.2,1.3,0.2]
```

```
Iris-setosa:[4.6,3.1,1.5,0.2]
```

```
Iris-setosa:[5.0,3.6,1.4,0.2]
```

```
Iris-setosa:[5.4,3.9,1.7,0.4]
```

```
Iris-setosa:[4.6,3.4,1.4,0.3]
```

```
Iris-setosa:[5.0,3.4,1.5,0.2]
```

```
Iris-setosa:[4.4,2.9,1.4,0.2]
```

```
Iris-setosa:[4.9,3.1,1.5,0.1]
```

```
Iris-setosa:[5.4,3.7,1.5,0.2]
```

```
Iris-setosa:[4.8,3.4,1.6,0.2]
```

```
.....
```

```
//获取标签列和特征列，索引并重命名
```

```
val labelIndexer = new StringIndexer().setInputCol("label").setOutputCol(  
"indexedLabel").fit(df)
```

```
输出结果：
```

```
labelIndexer: org.apache.spark.ml.feature.StringIndexerModel =
```

```
strIdx_2a3442136787
```

```
创建成功
```

```
val featureIndexer = new  
VectorIndexer().setInputCol("features").setOutputCol("indexedFeatures").setMaxCategories(4).fit(df)
```

输出结果:

```
featureIndexer: org.apache.spark.ml.feature.VectorIndexerModel =  
vecIdx_89a5b864820f
```

//把预测的类别重新转换为字符型

```
val labelConverter = new  
IndexToString().setInputCol("prediction").setOutputCol("predictedLabel").setLabels(labelIndexer.labels)
```

//数据样本随机划分为训练集和测试集，比例 7:3

```
val Array(trainingData, testData) = data.randomSplit(Array(0.7, 0.3))
```

输出结果为:

```
trainingData: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [features:  
vector, label: string]
```

```
testData: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [features: vector,  
label: string]
```

//构建决策树分类模型

//导入构建模型所需要的包

```
import org.apache.spark.ml.classification.DecisionTreeClassificationModel  
import org.apache.spark.ml.classification.DecisionTreeClassifier  
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator
```

//使用 setter 方法设置决策树模型的参数

```
val dtClassifier = new  
DecisionTreeClassifier().setLabelCol("indexedLabel").setFeaturesCol("indexedFeatures")
```

输出结果:

```
dtClassifier: org.apache.spark.ml.classification.DecisionTreeClassifier =  
dtc_bf350ce09979
```

构建模型成功

//在 pipeline 也就是机器学习工作流中设置参数

```
val pipelinedClassifier = new Pipeline().setStages(Array(labelIndexer, featureIndexer,  
dtClassifier, labelConverter))
```

//训练决策树分类模型

```
val modelClassifier = pipelinedClassifier.fit(trainingData)
```

输出结果:

```
modelClassifier: org.apache.spark.ml.PipelineModel = pipeline_c321e4346ddf
```

```
//预测
```

```
val predictionsClassifier = modelClassifier.transform(testData)
```

输出结果:

```
predictionsClassifier: org.apache.spark.sql.DataFrame = [features: vector, label: string ... 6 more fields]
```

```
//选择预测结果的前十个显示
```

```
predictionsClassifier.select("predictedLabel", "label", "features").show(10)
```

输出结果:

```
+-----+-----+-----+
|predictedLabel|    label|    features|
+-----+-----+-----+
|  Iris-setosa| Iris-setosa|[4.6,3.1,1.5,0.2]|
| Iris-virginica| Iris-versicolor|[4.9,2.4,3.3,1.0]|
|  Iris-setosa|  Iris-setosa|[4.9,3.1,1.5,0.1]|
|  Iris-setosa|  Iris-setosa|[4.9,3.1,1.5,0.1]|
|  Iris-setosa|  Iris-setosa|[5.0,3.0,1.6,0.2]|
|  Iris-setosa|  Iris-setosa|[5.0,3.4,1.5,0.2]|
|  Iris-setosa|  Iris-setosa|[5.0,3.4,1.6,0.4]|
|  Iris-setosa|  Iris-setosa|[5.0,3.5,1.3,0.3]|
|  Iris-setosa|  Iris-setosa|[5.0,3.5,1.6,0.6]|
|  Iris-setosa|  Iris-setosa|[5.1,3.5,1.4,0.3]|
+-----+-----+-----+
```

only showing top 10 rows

```
//对决策树分类模型的预测结果进行评估
```

```
val evaluatorClassifier = new
```

```
MulticlassClassificationEvaluator().setLabelCol("indexedLabel").setPredictionCol("pre  
diction").setMetricName("accuracy")
```

```
//计算精确度和测试误差
```

```
val accuracy = evaluatorClassifier.evaluate(predictionsClassifier)
```

输出结果:

准确率 accuracy: Double = 0.9333333333333333

Test Error = 0.06666666666666665

```
//输出决策树模型的结构
```

```
val treeModelClassifier =
```

```
modelClassifier.stages(2).asInstanceOf[DecisionTreeClassificationModel]
```

```
println("Learned classification tree model:\n" + treeModelClassifier.toDebugString)
```

输出结果:

Learned classification tree model:

DecisionTreeClassificationModel (uid=dtc_bf350ce09979) of depth 4 with 13 nodes

```

If (feature 2 <= 1.9)
  Predict: 2.0
Else (feature 2 > 1.9)
  If (feature 3 <= 1.7)
    If (feature 2 <= 5.1)
      If (feature 0 <= 4.9)
        Predict: 1.0
      Else (feature 0 > 4.9)
        Predict: 0.0
    Else (feature 2 > 5.1)
      Predict: 1.0
  Else (feature 3 > 1.7)
    If (feature 2 <= 4.8)
      If (feature 0 <= 5.9)
        Predict: 0.0
      Else (feature 0 > 5.9)
        Predict: 1.0
    Else (feature 2 > 4.8)
      Predict: 1.0

```

3.4 SparkMLlib 逻辑回归算法（2 课时）

3.4.1 逻辑回归算法

逻辑回归——**logistic regression**，直译为对数几率，又译为逻辑斯蒂回归，逻辑回归名为回归其实是分类算法。其数学模型本质是线性回归，在特征到结果的映射中加了一层函数映射，先对所有的特征线性求和，到这一步的输出 $y = f(x) = wx$ (w 是权重参数向量， x 是特征向量)，就是前边所述的线性回归，然后再用一个函数 $g(z)$ 计算，这个函数是将连续值映射到 $[0,1]$ 区间内，也就是输出 $y = g(f(x)) = g(wx)$ 。

在实际应用中，常常使用 **Sigmoid** 函数作为 $g(z)$ 函数：

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

这个函数图形是一个幅度在区间 $[0,1]$ 之间，两头分别在趋向负无穷和正无穷时逼近 0 和 1 的“S”形曲线。代入线性回归的输出后，还是将问题转化为对回归系数的求解上来，这个时候就可以使用梯度下降算法等方法进行求解。

逻辑回归这样做的优点就是：将原本很大的线性回归的输出范围映射到 $[0,1]$ 区间内，一方面消除了突变数值对预测结果的影响，更重要的一方面是可以直接对分类可能性建模，无需假设数据分布，而且 **Sigmoid** 函数是一个凸函数，可以直接用数值优化算法求解最优解。

3.4.2 算法源码分析

SparkMLlib 中对逻辑回归模型的目标函数优化提供了随机梯度下降和拟牛顿法两种算法，本次分析针对随机梯度下降算法。

(1) `LogisticRegressionWithSGD(object)`: 基于随机梯度下降的逻辑回归伴生对象，含有 `train` 静态方法，根据设置的逻辑回归模型参数创建逻辑回归类，执行 `run` 方法进行模型训练，`train` 方法需要设置的参数如下：

- * `input`——训练样本，格式为 `RDD(label,features)`;
- * `numIterations`——迭代次数，默认 100;
- * `stepSize`——每次迭代的步长，默认 1;
- * `miniBatchFraction`——每次迭代计算时的样本比例，默认 1.0;
- * `initialWeights`——初始化权重

(2) `LogisticRegressionWithSGD(class)`: 该类含有 `run` 方法，对权重进行优化计算;

(3) `runMiniBatchSGD`: 该方法根据训练样本迭代计算随机梯度，最后获得最优的权重，在每次迭代中计算样本梯度的同时更新梯度，`gradient.compute` 方法计算样本的梯度值，`updater.compute` 方法更新梯度;

(4) `LogisticRegressionModel`: 逻辑回归模型类，含有 `predict` 方法预测样本。

3.4.3 应用实战

(1) 本次实例使用的是 iris 鸢尾花数据集，数据与之前用到的相同。

(2) 代码详解

//导入所需的包文件

```
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.mllib.classification.{LogisticRegressionWithLBFGS,
LogisticRegressionModel}
import org.apache.spark.mllib.evaluation.MulticlassMetrics
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.linalg.{Vectors,Vector}
```

//读取数据

//先读取文本文件到 data 中

```
val data = sc.textFile("/mnt/hgfs/thunder-download/MLlib_rep/data/iris.txt")
```

//使用 `map` 函数，用 “,” 将每行数据划分为五部分，前四个花的特征，最后一

//个是花的分类，并且将标签和特征列存入 `LabeledPoint`

```
val parsedData = data.map { line =>
  | val parts = line.split(',')
  | LabeledPoint(if(parts(4)=="Iris-setosa") 0.toDouble
  | else if (parts(4)=="Iris-versicolor") 1.toDouble else
  | 2.toDouble, Vectors.dense(parts(0).toDouble,parts(1).toDouble,parts
  | (2).toDouble,parts(3).toDouble))
  | }
```

输出结果:

`parsedData`:

```
org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.LabeledPoint] =  
MapPartitionsRDD[398] at map at <console>:63
```

//输出数据

```
parsedData.foreach { x => println(x) }
```

输出结果为:

```
1.0,[6.0,2.9,4.5,1.5])  
(0.0,[5.1,3.5,1.4,0.2])  
(1.0,[5.7,2.6,3.5,1.0])  
(0.0,[4.9,3.0,1.4,0.2])  
(0.0,[4.7,3.2,1.3,0.2])  
.....
```

//划分数据为训练集和测试集，比例 6:4

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

输出结果:

splits:

```
Array[org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.LabeledPoint]] =  
Array(MapPartitionsRDD[399] at randomSplit at <console>:65,  
MapPartitionsRDD[400] at randomSplit at <console>:65)
```

//划分的第一部分存入训练集

```
val training = splits(0).cache()
```

输出结果:

```
training: org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.LabeledPoint]  
= MapPartitionsRDD[399] at randomSplit at <console>:65
```

//划分的第二部分存入测试集

```
val test = splits(1)
```

输出结果:

```
test: org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.LabeledPoint] =  
MapPartitionsRDD[400] at randomSplit at <console>:65
```

//使用上边得到的数据，用 set 方法设置训练模型的参数

//设置分类数目为 3，这个是根据你自己的目标或者数据样本本身来确定的

//训练构建三分类逻辑回归模型

//这里的 LogisticRegressionWithLBFGS 模型是因为求解非线性优化问题 ($L(w)$ 求极//大值) 的方法使用了 LBFGS(Limited-memory BFGS)

```
val model = new LogisticRegressionWithLBFGS().
```

```
  | setNumClasses(3).
```

```
  | run(training)
```

输出结果:

```
model: org.apache.spark.mllib.classification.LogisticRegressionModel =
```

```
org.apache.spark.mllib.classification.LogisticRegressionModel: intercept = 0.0,
```

```
numFeatures = 8, numClasses = 3, threshold = 0.5
```

//使用模型对测试数据进行预测，使用的是 `predict` 方法。`test` 数据每一行分为标签和特征，使用 `map` 方法对每一行数据进行 `model.predict(features)`，根据特征得到对应的预测值，将此预测值和样本真实标签值一起存入 `predictionAndLabels`。

```
val predictionAndLabels = test.map { case LabeledPoint(label, features) =>
    | val prediction = model.predict(features)
    | (prediction, label)
    | }
```

输出结果：

```
predictionAndLabels: org.apache.spark.rdd.RDD[(Double, Double)] =
MapPartitionsRDD[461] at map at <console>:73
```

//输出预测值

```
predictionAndLabels.foreach { x => println(x) }
```

输出结果：

```
(1.0,1.0)
(2.0,1.0)
(2.0,1.0)
(1.0,1.0)
(1.0,1.0)
(1.0,1.0)
(1.0,1.0)
(1.0,1.0)
(1.0,1.0)
(1.0,1.0)
.....
```

这里选取了前十行预测标签和真实标签的对比，可以发现有两个预测错误，接下来我们整体评估一下模型的预测能力

//对逻辑回归模型的预测能力进行评估

//将预测结果存入 `MulticlassMetrics`，计算预测准确度

```
val metrics = new MulticlassMetrics(predictionAndLabels)
```

```
val precision = metrics.precision
```

```
println("Precision = " + precision)
```

输出结果：

```
Precision = 0.9180327868852459
```

4. SparkMLlib 聚类算法（4 课时）

4.1 SparkMLlib KMeans 聚类算法（2 课时）

4.1.1 KMeans 聚类算法

KMeans 聚类算法属于划分类型的聚类方法，其求解过程是迭代计算，基本思想是在开始时随机选择 k 个簇的中心，依据最近邻规则，把待分类样本点分给每个簇。按照平均计算的方法再计算每个簇的质心，对簇心的位置进行更新，开始新一轮的迭代，直到结果收敛于簇心的移动距离小于事先给定的阈值。

其算法的过程如下所示：

- (1) 根据给定的 k 值，选取 k 个样本点作为初始划分中心；
 - (2) 计算所有样本点到每一个划分中心的距离，并将所有样本点划分到距离最近的划分中心；
 - (3) 计算每个划分中样本点的平均值，将其作为新的中心；
- 循环进行 2~3 步直至达到最大迭代次数，或划分中心的变化小于某一预定阈值即聚类的中心不再进行大范围的移动。

4.1.2 算法源码分析

MLlib 中对 KMeans 聚类算法的实现主要包括以下几个部分：

- (1) KMeans 聚类伴生对象

KMeans 聚类伴生对象 `object KMeans` 是建立聚类模型的入口，定义了训练模型的 `train` 方法，该方法的参数及其意义如下：

- *data——以 `RDD(Vector)` 形式输入的训练数据；
- *k——创建的聚类簇的个数；
- *maxIterations——允许迭代的最大次数；
- *initializationMode——初始化聚类中心点的算法，可以选择随机或 `k-means++`；
- *seed——聚类初始化的随机种子，默认是基于系统时间生成种子。

KMeans 聚类类是 `class KMeans`，`run` 方法根据初始化的参数训练模型。

- (2) 在上述的 `run` 方法中，MLlib 使用 `runAlgorithm` 方法计算中心点，其算法步骤如下：

初始化中心；进行迭代计算各个样本点分别属于哪个中心点，累加中心点的样本值并完成计数；由中心点的样本数据更新中心点；与更新前的数值比较判断是否完成。`runs` 参数表示并行度，在计算中心点时分多组进行计算，最后以最好的一组为聚类中心。

- (3) `runAlgorithm` 中 MLlib 支持 `k-means++`：`initKMeansParallel` 和随机初始化：`initRandom`，`k-means++` 算法基于的是初始聚类中心之间相互距离尽可能远的思想，而随机初始化就是随机选择样本数据点作为中心点。

- (4) MLlib 中还提出了一种快速找到样本点与离其最近的聚类中心的方法：`findClosest`，还有一种快速计算两样本点之间距离的方法 `fastSquaredDistance`。

- (5) 训练数据输入，得到一个 KMeans 聚类模型，模型参数为中心点向量，方法为：预测、保存和加载模型。其中预测方法 `predict` 支持 RDD 和向量格式数据的计算。

4.1.3 应用实战

- (1) 本次实战使用的是 UCI 数据集中很经典的鸢尾花数据 `Iris`，一共有 150 条数

据，示例如下：

5.4,3.0,4.5,1.5,versicolor

其中四个实数值表示花的四个部位的大小尺寸，最后对应的是对应花所属的亚种类型即标签值，一共有三种。

（2）代码详解

//导入 Vectors 包和 KMeans 算法包

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

```
import org.apache.spark.mllib.clustering.{KMeans, KMeansModel}
```

//添加自己的数据集路径，读取数据文件

```
val rawData = sc.textFile("/mnt/hgfs/thunder-download/MLlib_rep/data/iris.txt")
```

此处输出的信息如下：

```
rawData: org.apache.spark.rdd.RDD[String] = /mnt/hgfs/thunder-download/
```

```
MLlib_rep/data/iris.txt MapPartitionsRDD[44] at textFile at <console>:29
```

可以看到是 RDD[String]格式的数据

//把读取的数据转化为训练数据

```
val trainingData = rawData.map(line => {Vectors.dense(line.split(",").filter(p =>
p.matches("\\d*(\\.?)\\d*"))).map(_toDouble)}}).cache()
```

此处输出的信息如下：

```
trainingData: org.apache.spark.rdd.RDD[org.apache.spark.mllib.linalg.Vector] =
MapPartitionsRDD[45] at map at <console>:31
```

训练数据为 Vector 类型，并且使用了 filter 方法，使用正则表达式将花的类别标签去掉。

//调用 KMeans 的 run(RDD[Vector])方法训练 KMeans 模型

//该方法支持重载，这里使用 KMeans.train(data, k, maxIterations, runs)的形式，传

//入训练样本和参数。k 为聚类数目，默认 2；maxIterations 为最大迭代次数，

//默认 20；runs 为运行的次数，默认为 1。

```
val model : KMeansModel = KMeans.train(trainingData, 3, 100, 5)
```

这一步就是 RDD 操作的 action 型操作，模型创建成功

//使用 KMeansModel 类的 clusterCenters 属性得到模型中聚类的中心

```
model.clusterCenters.foreach({
```

```
  | center => {
```

```
    | Display all 664 possibilities? (y or n)
```

```
    | println("Clustering Center:"+center)
```

```
  | })
```

输出结果如下：

Clustering

```
Center:[5.901612903225806,2.7483870967741932,4.393548387096774,1.43387096
7741935]
```

Clustering

```
Center:[5.005999999999999,3.4180000000000006,1.4640000000000002,0.2439999
```

```

999999999]
Clustering
Center:[6.85,3.0736842105263147,5.742105263157893,2.071052631578947]
//用 predict()方法预测每个样本所属聚类
trainingData.collect().foreach(
    | sample => {
    | val predictedCluster = model.predict(sample)
    | println(sample.toString + " belongs to cluster " + predictedCluster)
    | })

```

输出的结果如下：

```

[5.1,3.5,1.4,0.2] belongs to cluster 1
[4.9,3.0,1.4,0.2] belongs to cluster 1
[4.7,3.2,1.3,0.2] belongs to cluster 1
[4.6,3.1,1.5,0.2] belongs to cluster 1
[5.0,3.6,1.4,0.2] belongs to cluster 1
...
[7.0,3.2,4.7,1.4] belongs to cluster 0
[6.4,3.2,4.5,1.5] belongs to cluster 0
[6.9,3.1,4.9,1.5] belongs to cluster 2
[5.5,2.3,4.0,1.3] belongs to cluster 0
[6.5,2.8,4.6,1.5] belongs to cluster 0
...

```

//最后可以计算下 WSSSE 即集合内误差平方和，此项数据是用来测量聚类的有效性

```
val wssse = model.computeCost(trainingData)
```

输出结果为：

wssse: Double = 78.94084142614622，改变 k 值可以改变 wssse，也就可以对无法明确分类的数据选择最优的分类 k 值即聚类数。

4.2 Spark MLlib GMM 高斯混合模型算法（2 课时）

4.2.1 GMM 高斯混合模型算法

在 MLlib 的聚类算法中，高斯混合模型算法也是一种很重要的聚类算法，其基于单高斯模型，而两者的数学基础都是高斯分布。

在统计学中，若随机变量 x 服从数学期望为 μ 、方差为 σ^2 的高斯分布，则记为 $N(\mu, \sigma^2)$ 。数学表达式如下：

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x - \mu)^2}{2\sigma^2}\right)$$

高斯分布的期望值 μ 和标准差 σ 分别决定了分布的位置和幅度。而最为我们熟

知的就是期望值为 0，标准差为 1 的标准高斯分布——正态分布。

以高斯分布为基础的单高斯分布聚类模型，原理是根据已有数据建立一个分布模型，再向模型中代入样本数据计算其值，设定一个阈值范围，如果计算值在阈值以内，则判定该数据与高斯分布相匹配，否则认定该数据不属于该高斯模型的聚类，考虑到实际应用中，特征维度是多维的，多维高斯分布公式如下， \mathbf{x} 为样本数据：

$$F(\mathbf{x}, \mu, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(\mathbf{x} - \mu)^2}{2\sigma^2}\right)$$

混合高斯模型 **GMM** 是在单高斯模型基础上发展得到的，是一种概率式的聚类方法，属于生成式模型，它假设所有的数据样本都是由某一个给定参数的多元高斯分布所生成的。可以有效解决单高斯模型聚合混合数据不理想的问题。**GMM** 的原理就是对于任何样本数据来说，都可以使用多个高斯分布模型表示，其数学表达式如下：

$$\Pr(\mathbf{x}) = \sum \pi F(\mathbf{x}, \mu, \sigma)$$

对于此表达式，我们要做的就是使用样本数据，通过极大似然估计法训练获得模型参数。可以通过增加 **GMM** 中模型的数量，任意的逼近任何连续的概率密度分布函数，而这个数量值的选择就是训练模型好坏的关键。

4.2.2 算法源码分析

(1) 高斯混合伴生对象：object GaussianMixture，需要设置的参数有：

* k：聚类的数目

* d：特征数目

当 k 不是很小且 d 大于 25 时，对多元高斯的计算需要使用启发式分布。

(2) 高斯混合类：class GaussianMixture，抽象 **GMM** 的超参数并进行训练，创建此类，使用 set 方法设置参数，调用 fit 方法训练一个 **GMM** 模型，需要设置的参数如下：

* K——聚类数目，默认为 2

* maxIter——最大迭代次数，默认为 100

* seed——随机数种子，默认为随机 Long 值

* Tol——对数似然函数收敛阈值，默认为 0.01

(3) 高斯混合模型：GaussianMixtureModel，所含参数主要是：

Weights——Array[Double] 每个多元高斯分布所占权重

gaussians:——Array[MultivariateGaussian] 一个 **GMM** 由多个多元高斯分布组成
含有 predict 方法预测样本，还有函数 computeSoftAssignments，给定一个新的样本点，计算它属于各个聚类的概率

(4) 模型训练

run 方法：数据和参数准备好后用来训练模型；

updateWeightsAndGaussians：更新高斯模型的权重

4.2.3 应用实战

(1) 本次实战使用的数据是 iris 数据集，与之前教程所用的一样。

(2) 代码详解

//导入所需的包文件

```
import org.apache.spark.ml.clustering.{GaussianMixture,GaussianMixtureModel}
import org.apache.spark.ml.linalg.Vectors
```

//为后续 RDD 隐式转换做准备

```
import spark.implicits._
```

//定义 case class 作为生成的 DataFrame 中每一个数据样本的数据类型

```
case class model_instance(features: Vector)
```

输出结果:

```
defined class model_instance
```

//数据读入 RDD 中，并转换为 DataFrame

```
val rawData = sc.textFile("/mnt/hgfs/thunder-download/MLlib_rep/data/iris.txt")
```

输出结果:

```
rawData: org.apache.spark.rdd.RDD[String] = iris.csv MapPartitionsRDD[48] at
textFile at <console>:33
```

//过滤类标签

```
val df = rawData.map(line =>
  | { model_instance( Vectors.dense(line.split(",").filter(p =>
    | p.matches("\\d*(\\.?)\\d*"))
    | .map(_toDouble)) )}).toDF()
```

//创建 GaussianMixture 类，设置训练所需参数，聚类数目取 3

```
val gm = new GaussianMixture().setK(3)
  |.setPredictionCol("Prediction")
  |.setProbabilityCol("Probability")
```

输出结果:

```
gm: org.apache.spark.ml.clustering.GaussianMixture =
GaussianMixture_53916e2247ae
```

```
val gmm = gm.fit(df)
```

输出结果:

```
gmm: org.apache.spark.ml.clustering.GaussianMixtureModel =
GaussianMixture_53916e2247ae
```

这里设置 Probability，相比于 KMeans 聚类，可以得到样本属于每个聚类的概率

//使用 transform 方法处理数据集并输出，可以看到每个样本的预测聚类和其概
//率分布向量

```
val result = gmm.transform(df)
result.show(150, false)
```

```
+-----+-----+-----+
|features      |Prediction|Probability|
+-----+-----+-----+
|[5.1,3.5,1.4,0.2]|0        |[0.9999999999999951,4.682229962936943E-
17,4.868372929920407E-15] |
|.....|..      |.....|
|[5.6,2.8,4.9,2.0]|1        |[8.920203149708086E-
16,0.5988576194515217,0.4011423805484774] |
|.....|..      |.....|
|[6.3,2.7,4.9,1.8]|2        |[5.703158630226758E-
16,0.022033640207248576,0.9779663597927509] |
+-----+-----+-----+
```

//查看模型的相关参数，即各个混合高斯分布的参数，这里主要是各个混合成分
//的权重和混合成分的均值向量和协方差矩阵

```
for (i <- 0 until gmm.getK) {
  | println("Component %d : weight is %f \n mu vector is %s \n sigma matrix
is %s" format
  | (i, gmm.weights(i), gmm.gaussians(i).mean, gmm.gaussians(i).cov))
  | }
```

输出结果如下：

```
Component 0 : weight is 0.333333
mu vector is
[5.006000336585284,3.41800074359835,1.4640001090120234,0.243999962786779
1]
sigma matrix is 0.12176391071215485 0.09829168918600302
0.01581595534223468 0.01033602571352466
0.09829168918600302 0.14227526345684152 0.011447885703674401
0.01120804907975396
0.01581595534223468 0.011447885703674401 0.02950400173292353
0.005584009823879005
0.01033602571352466 0.01120804907975396 0.005584009823879005
0.01126400540784641
Component 1 : weight is 0.158358
mu vector is
[6.683368733405807,2.86961545411428,5.6462886220107515,2.005673427136211
]
sigma matrix is 0.49328013505428253 0.050374713498113975
0.3573203540815462 0.050018569392196975
0.050374713498113975 0.04009423452907058 0.00416971505937197
0.02000523766170409
```

```

0.3573203540815462      0.00416971505937197      0.33772537665488306
0.017006917604832562
0.050018569392196975      0.02000523766170409      0.017006917604832562
0.06935869650451881
Component 2 : weight is 0.508309
mu vector is
[6.130726266791161,2.872742630634873,4.675369349848198,1.573293136253829
8]
sigma      matrix      is      0.34423978263401117      0.14332952213838432
0.3498831855148551      0.1447023418962832
0.14332952213838432      0.13127254135549662      0.1848327285944271
0.09799971374720898
0.3498831855148551      0.1848327285944271      0.5558476131836437
0.2698797562441122
0.1447023418962832      0.09799971374720898      0.2698797562441122
0.16825697031717957

```

5. SparkMLlib 关联规则挖掘算法（2 课时）

5.1 关联规则算法

研究规则挖掘算法的目的是发下商品交易数据库中不同类型商品之间的潜在联系，找出客户在购买商品时的行为模式。例如客户购买了某个产品，会对客户购买其他产品产生何种影响，通过分析诸如此类的结果，可以应用到实际的商业规划中，比如超市商品货架的分布排放、商品库存的计划以及如何跟客户的购买习惯对客户进行分类从而进行有针对性的精准营销。

在 MLlib 中，主要实现了 FP-Tree 关联规则算法，这种方法基于 Apriori 算法，对频繁项集数据进行挖掘，优点是效率和鲁棒性上有很大的提高。通过一个简单的例子介绍关联规则算法—啤酒和尿布：上世纪沃尔玛超市销售人员发现在某些日子，啤酒和尿布这两样表面看起来毫无关联的东西出现在同一份购物清单上，经调查发现是，有些日子是父亲外出工作回家前会去超市购买婴儿用品，同时也会购买啤酒之类的产品，于是根据此发现，超市将啤酒鱼尿布放的尽可能远，在中间的通道上摆放吸引这些年轻父亲的产品，促进购买。在这里项就是指的一个个物品，项集就是所有物品中某几项组成的集合，在本教程中是两两组合。

据此案例，Apriori 算法是一种挖掘关联规则的频繁项集算法，核心思想是通过候选集生成和情节的向下封闭检测两个阶段来挖掘频繁项集。其过程是计算一份购物清单中所有商品中每两件商品同时出现的次数，从而可以算出两件商品同时出现的概率即支持度，也可以算出其中一件商品出现时另一件商品出现的概率即置信度。然后根据计算结果，跑去数据支持个数过少的物品项。重新相互组合商品开始下一轮的计算，属于候选消除算法。

MLlib 中支持的 FP-growth 频繁模式增长算法，将提供频繁项集的数据库压缩到 FP 树即频繁模式树，保留物品集合的关联信息。可以解决多次扫描数据库带来的大量小频繁项集的问题，在理论上只对数据库进行两次扫描，直接压缩数据

库生成一个频繁模式树从而生成关联规则。主要分以下两步骤完成：

- (1) 利用数据库中已有样本数据建立 FP 树；
- (2) 建立频繁项集规则。

5.2 算法源码分析

MLlib 中是使用并行 FPGrowth 算法挖掘频繁项集的方法完成 FPGrowth 关联规则的挖掘，分为频繁项的挖掘和频繁项集的挖掘。

(1) class FPGrowth: 含有 run 方法基于 FPGrowth 算法计算频繁项集，需要设置的参数为：

- * minSupport——频繁模式的最小支持度，默认为 0.3；

- * numPartitions——分区数，应用于 FPGrowth 并行计算，默认为输入的样本数据的分区数

(2) 频繁项集挖掘：

genFreqItems: 频繁项计算；genFreqItemsets: 频繁项集计算；

(3) FPTree 类：

Add 方法:树增加；

Merge: 树合并；

Project: 取后缀树；

GetTransactions: 取节点下的所有事务；

Extract: 提取频繁项集

(4) FPGrowthModel: FPGrowth 模型

FreqItemsets: 频繁项集，格式为 RDD (freqItemset)

5.3 应用实战

(1) 数据集为用英文小写字母代表的物品组成的购物清单，如下所示：

```
rz hkp
zyxwvuts
sx onr
xzy mtsqe
z
```

```
xzyrqtp
```

每个字母代表一个物品

(2) 代码详解

//导入需要的机器学习包和相关基础包

```
import org.apache.log4j.{ Level, Logger }
```

```
import org.apache.spark.{ SparkConf, SparkContext }
```

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

```
import org.apache.spark.mllib.fpm.{ FPGrowth, FPGrowthModel }
```

```
//构建 Spark 对象
```

```
val conf = new SparkConf().setAppName("fpg")
```

```
val sc = new SparkContext(conf)
```

```

Logger.getRootLogger.setLevel(Level.WARN)

//读取样本数据
val data_path = "/mnt/hgfs/thunder-download/MLlib_rep/
data/sample_fpgrowth.txt"
val data = sc.textFile(data_path)
val examples = data.map(_._split(" ")).cache()

//建立模型
val minSupport = 0.2
val numPartition = 10
val model = new FPGrowth().
    setMinSupport(minSupport).
    setNumPartitions(numPartition).
    run(examples)

//打印结果
println(s"Number of frequent itemsets: ${model.freqItemsets.count()}")
model.freqItemsets.collect().foreach { itemset =>
    println(itemset.items.mkString("[", ",", "]") + ", " + itemset.freq)
}

```

最后输出关联规则：

```

[z], 5
[x], 4
[x,z], 3
[y], 3
[y,x], 3
[y,x,z], 3
[y,z], 3
[r], 3
[r,x], 2
[r,z], 2
.....

```

6. SparkMLlib 推荐算法（4 课时）

6.1 SparkMLlib 协同过滤推荐算法（2 课时）

6.1.1 协同过滤推荐算法

协同过滤算法是一种经典的推荐算法，推荐算法的基础是计算两个对象之间的相关度，其有两种实现形式：基于用户的推荐和基于物品的推荐。

基于用户的推荐思想是基于用户对某项物品的喜好找到具有相同喜好的相

邻用户，然后将相邻用户喜欢的物品推荐给该用户。在这里，将一个用户对所有物品的喜好作为一个向量，计算不同用户之间的相似度，计算出若干位邻居用户后，根据这些邻居用户的相似度权重和其喜欢的物品，从该用户没有喜欢的物品中挑选出物品进行排序作为推荐。基本原理如下：

| 用户/物品 | 物品 1 | 物品 2 | 物品 3 | 物品 4 |
|----------------------------------|------|------|------|------|
| 用户 1 | 喜好 | | 喜好 | 推荐 |
| 用户 2 | | 喜好 | | |
| 用户 3 | 喜好 | | 喜好 | 喜好 |
| 计算相似度，用户 3 为 1 的相邻用户推荐物品 4 给用户 1 | | | | |

基于物品的推荐思想与基于用户的相似，计算相似度时是计算物品之间的，基于用户喜欢的物品找到相似的物品，做为推荐物品推荐给用户。在这里，就是将所有用户对某个物品的喜好作为一个向量，计算物品之间的相似度，得到物品的相似物品后，就可以根据用户以前喜欢的所有物品预测出用户还没有喜欢的物品，同样也可以得到一个排序后的推荐列表。基本原理如下：

| 用户/物品 | 物品 1 | 物品 2 | 物品 3 |
|---------------------------|------|------|------|
| 用户 1 | 喜好 | | 喜好 |
| 用户 2 | 喜好 | 喜好 | 喜好 |
| 用户 3 | 喜好 | | 推荐 |
| 分析物品 1 3，预测用户 3 有可能喜欢物品 3 | | | |

通过以上的简单描述，我们明确需要计算的三个地方：用户对物品的喜好，用户与用户之间相似度的计算，物品与物品之间相似度的计算。接下来将对这三点做一个简单的介绍。

推荐算法中决定推荐效果的最重要的因素是用户的喜好信息，比如我们浏览商品时的点击，转发，停留时间等等。这些用户行为决定了用户对某项物品的喜好程度，通常推荐算法中对众多的用户行为有两种处理方式：（1）将不同用户行为分组；（2）对不同用户行为进行加权，降噪和归一化处理后根据用户行为的重要性设置不同权重。1·

相似度计算是推荐算法中的关键一环。无论是基于用户还是基于物品的计算，其计算方法都是一样的。常用计算方式有基于欧氏距离，基于余弦角度，皮尔逊相关系数以及由余弦相似度扩展而来的 Tanimoto 系数。

6.1.2 算法源码分析

在 SparkMLlib 中并没有我们上述的协同过滤推荐算法。在这里就简单介绍如何基于前述的理论知识，实现基于物品的协同过滤推荐算法。

算法的实现过程主要包括以下几个过程：

（1）相似度的计算，用 `new ItemSimilarity` 建立物品相似度计算，设置模型参数，再用 `Similarity` 方法计算相似度，最后返回的结果是物品之间的相似度，其格式是 RDD。具体实现代码如下：

```
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.
   *  $w(i,j) = N(i) \cap N(j) / \sqrt{N(i) * N(j)}$ 
   * @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]) = {
    // 0 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))
    // 1 (用户: 物品) 笛卡尔积 (用户: 物品) => 物品:物品组合
    val user_rdd3 = user_rdd2.join(user_rdd2)
    val user_rdd4 = user_rdd3.map(f => (f._2, 1))
    // 2 物品:物品:频次
    val user_rdd5 = user_rdd4.reduceByKey((x, y) => x + y)
    // 3 对角矩阵
    val user_rdd6 = user_rdd5.filter(f => f._1._1 == f._1._2)
    // 4 非对角矩阵
    val user_rdd7 = user_rdd5.filter(f => f._1._1 != f._1._2)
    // 5 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))))
    // 6 结果返回
    user_rdd12.map(f => ItemSimi(f._1, f._2, f._3))
}

/**
 * 余弦相似度矩阵计算.
 *  $T(x,y) = \frac{\sum x(i)y(i)}{\sqrt{\sum x(i)*x(i)} * \sqrt{\sum y(i)*y(i)}}$ 
 * @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]) = {
    // 0 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)))
    // 1 (用户,物品,评分) 笛卡尔积 (用户,物品,评分) => (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)))
    // 2 (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(_
+ _)
    // 3 对角矩阵
    val user_rdd6 = user_rdd5.filter(f => f._1._1 == f._1._2)
    // 4 非对角矩阵
    val user_rdd7 = user_rdd5.filter(f => f._1._1 != f._1._2)
    // 5 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))))

```

```

// 6 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.
 *  $d(x, y) = \sqrt{\sum ((x(i)-y(i)) * (x(i)-y(i)))}$ 
 *  $sim(x, y) = n / (1 + d(x, y))$ 
 * @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]) = {
  // 0 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)))
  // 1 (用户,物品,评分) 笛卡尔积 (用户,物品,评分) => (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)))
  // 2 (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) * (f._2._1 -
f._2._2))).reduceByKey(_ + _)
  // 3 (物品 1,物品 2,评分 1,评分 2)组合 => (物品 1,物品 2,1) 组合 并
累加 计算重叠数
  val user_rdd6 = user_rdd4.map(f => (f._1, 1)).reduceByKey(_ + _)
  // 4 非对角矩阵
  val user_rdd7 = user_rdd5.filter(f => f._1._1 != f._1._2)
  // 5 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))))
  // 6 结果返回
  user_rdd9.map(f => ItemSimi(f._1, f._2, f._3))
}
}

```

(2) 协同过滤推荐

RecommendedItem 类，计算物品推荐，设置模型参数后，调用 **Recommend** 方法，对推荐物品进行计算，最后返回推荐给用户的物品，格式为 **RDD**。其计算原理是根据物品之间的相似度和用户对物品的评分对推荐物品计算，并且对用户

已经喜好的物品进行过滤。

```
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]) = {
    // 0 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))
    // 1 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))))
    // 2 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))
    // 3 矩阵计算——用户：元素累加求和
    val rdd_app1_R4 = rdd_app1_R3.reduceByKey((x, y) => x + y)
    // 4 矩阵计算——用户：对结果过滤已有 I2
    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)))
    // 5 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]) = {
    // 0 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))
    // 1 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))))
    // 2 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))
    // 3 矩阵计算——用户：元素累加求和
    val rdd_app1_R4 = rdd_app1_R3.reduceByKey((x, y) => x + y)
    // 4 矩阵计算——用户：对结果过滤已有 I2
    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)))
    // 5 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))
  }
}

```

6.1.3 应用实战

(1) 本次实战的数据集使用的是自建数据，数据格式是用户序号，物品序号以及用户对物品的评分：

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

(2) 代码详解

//导入需要的 spark 包，注意没有涉及到 MLlib 包

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

//构建 Spark 对象

```
val conf = new SparkConf().setAppName("ItemCF")  
val sc = new SparkContext(conf)  
Logger.getRootLogger.setLevel(Level.WARN)
```

//读取样本数据

```
val data_path = "/mnt/hgfs/thunder-download/MLlib_rep/data/itemcf.txt"  
val data = sc.textFile(data_path)  
val userdata = data.map(_.split(",")).map(f => (ItemPref(f(0), f(1),  
f(2).toDouble))).cache()
```

//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)
```

//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)  
}
```

输出结果：

计算得到的相似度矩阵（物品，物品，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
```

计算得到给用户推荐的物品表

```
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
```

6.2 SparkMLlib ALS 交替最小二乘算法（2 课时）

6.2.1 交替最小二乘算法

ALS 全称 **alternating least squares** 交替最小二乘。在推荐算法中，是指基于 ALS 求解的一种协同推荐算法。ALS 算法是统计分析中一种常用的逼近计算的算法，其计算结果能够最大程度逼近真实的结果。以下简单介绍下这种算法，我们还是将重点放在这个算法在协同推荐算法中的实际应用。

ALS 的基础是 LS——最小二乘，本质属于数学优化，也常常用在机器学习中，其原理是通过最小化误差的平方和，从而找到数据的最优函数匹配。通过 LS 可以便捷求得未知的数据，并且使得预测的数据与真实数据之间的误差值的平方和最小。所以，我们也常常在回归任务中看到它的身影。公式如下：

$$f(x) = ax + b$$
$$J = \sum (f(x_i) - y_i)^2$$

$f(x)$ 为拟合公式，即所求目标函数， y_i 就是实际值，在这里我们就是希望预测与实际值间的误差最小。这个误差就是 ALS 的优化目标，即下文中通过 U 和 V

重构 W 产生的误差

而我们所说的 ALS 就涉及到矩阵的知识，一个基于用户名和物品表的用户评分矩阵 W 可以被分解为两个更小的矩阵 U ：用户的矩阵和 V ：物品的矩阵。ALS 在 MLlib 中的应用如下：

- (1) 对 U 或 V 随机生成矩阵中的某个特定对象，例如是 $U(0)$ ；
- (2) 固定随机生成的对象 $U(0)$ ，求未随机生成的矩阵对象 $V(0)$ ；
- (3) 得到求得的矩阵对象计算随机化矩阵对象；
- (4) 两矩阵对象相互交替计算，迭代这个过程，每次迭代都降低重构误差，求得与实际矩阵差值达到提前设定的最小阈值位置。

6.2.2 算法源码分析

在 MLlib 中，ALS 算法采用的是高效分解矩阵的计算，设计数据分区和 RDD 缓存来减少数据交换，这对迭代计算非常友好。对 ALS 的源码分析如下：

- (1) object ALS: ALS 伴生对象，含有 train 静态方法，需要设置的参数如下：
 - * ratings——用户评分，格式 RDD(userID,productID,rating)；
 - * rank——特征数量；
 - * iterations——迭代数量；
 - * lambda——正则因子，一般为 0.01；
 - * blocks——数据分割，用来进行并行计算；
 - * seed——随机种子
- (2) class ALS: ALS 类，run 方法，调用 NewALS 类的 train 方法计算；
- (3) 矩阵分解计算
 - makeBlocks——创建用户和物品的 Block 块，是用来进行交替计算的；
 - initialize——初始化用户和物品的特征矩阵；
 - computeFactors——使用 ALS 求解用户和物品特征矩阵，进行迭代计算，现根据用户特征矩阵求解物品特征矩阵，再根据结果进行交替计算；
- (4) MatrixFactorizationModel: 矩阵分解模型，矩阵分解计算完成后生成矩阵分解模型：用户和物品特征矩阵。包括了如下方法：predict: 预测用户对物品评分；recommendProducts: 对用户推荐物品的列表；recommendUsers: 对物品推荐用户的列表；recommendProductsForUsers: 在所有用户推荐物品中选取前 k 个物品；recommendUsersForProducts: 在所有物品推荐用户中选取前 k 个用户。

6.2.3 应用实战

(1) 本次实战使用的数据集 sample_movielens_ratings，样本示例如下，每行样本数据由一个用户、一部电影、该用户对电影的评分和时间戳组成。这是一种明确表示对物品喜好的行为，称作显性反馈；与此对应的，没有明确反映用户喜好的行为就是隐性反馈行为。

```
0::2::3::1424380312
0::3::1::1424380312
0::5::2::1424380312
0::9::4::1424380312
0::11::1::1424380312
0::12::2::1424380312
0::15::1::1424380312
```


0::17::1::1424380312

0::19::1::1424380312

0::21::1::1424380312

(2) 代码详解

//导入需要的机器学习包

```
import org.apache.spark.ml.evaluation.RegressionEvaluator
```

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

//创建读取数据的规则 Rating 类型[int, int, float, long]

```
case class Rating(userId: Int, movieId: Int, rating: Float, timestamp: Long)
```

//创建函数把数据集中的数据转化为 Rating 类型

```
def parseRating(str: String): Rating = {  
    |         val fields = str.split("::")  
    |         assert(fields.size == 4)  
    |         Rating(fields(0).toInt, fields(1).toInt, fields(2).toFloat,  
fields(3).toLong)  
    |     }  
}
```

输出结果:

```
parseRating: (str: String)Rating
```

//导入需要的包文件 implicits

//并取数据集

```
import spark.implicits._
```

```
import spark.implicits._
```

```
val ratings =
```

```
spark.sparkContext.textFile("file:///mnt/hgfs/thunder-download/MLlib_rep/data/  
sample_movielens_ratings.txt").map(parseRating).toDF()
```

输出结果:

```
ratings: org.apache.spark.sql.DataFrame = [userId: int, movieId: int ... 2 more fields]
```

//把处理后的数据打印出来

```
ratings.show()
```

```
+-----+-----+-----+-----+  
|userId|movieId|rating|timestamp|  
+-----+-----+-----+-----+  
|      0|      2|    3.0|1424380312|  
|      0|      3|    1.0|1424380312|  
|      0|      5|    2.0|1424380312|  
|      0|      9|    4.0|1424380312|  
|      0|     11|    1.0|1424380312|  
|      0|     12|    2.0|1424380312|  
|      0|     15|    1.0|1424380312|  
|      0|     17|    1.0|1424380312|
```

| | | | | |
|--|---|----|-----|------------|
| | 0 | 19 | 1.0 | 1424380312 |
| | 0 | 21 | 1.0 | 1424380312 |
| | 0 | 23 | 1.0 | 1424380312 |
| | 0 | 26 | 3.0 | 1424380312 |
| | 0 | 27 | 1.0 | 1424380312 |
| | 0 | 28 | 1.0 | 1424380312 |
| | 0 | 29 | 1.0 | 1424380312 |
| | 0 | 30 | 1.0 | 1424380312 |
| | 0 | 31 | 1.0 | 1424380312 |
| | 0 | 34 | 1.0 | 1424380312 |
| | 0 | 37 | 1.0 | 1424380312 |
| | 0 | 41 | 2.0 | 1424380312 |

+-----+-----+-----+-----+

only showing top 20 rows

//划分数据集为训练集和测试集

```
val Array(training, test) = ratings.randomSplit(Array(0.8, 0.2))
```

输出结果:

```
training: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [userId: int,
movieId: int... 2 more fields]
```

```
test: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [userId: int, movieId:
int ... 2 more fields]
```

//建立 ALS 推荐模型

//显性反馈模型

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

输出结果:

```
alsExplicit: org.apache.spark.ml.recommendation.ALS = als_05fe5d78ccd4
```

//隐性反馈模型，setImplicitPrefs 设置为 true

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

输出结果:

```
alsImplicit: org.apache.spark.ml.recommendation.ALS = als_8f0d989bfdea
```

/**

在 MLlib 中可以对模型参数进行如下调整:

numBlocks 是用于并行化计算的用户和商品的分块个数 默认 10;

rank 是模型中隐语义因子的个数 默认 10;

maxIter 是迭代的次数 默认 10;

regParam 是 ALS 的正则化参数 默认 1.0;

implicitPrefs 决定了是用显性反馈 ALS 的版本还是用适用隐性反馈数据集的版本 (默认是 false, 即用显性反馈)。

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 默认 1.0；

nonnegative 决定是否对最小二乘法使用非负的限制 默认 **false**。

调整 **maxiter** 增加，**regParam** 减小，均方差会减小，得到的推荐结果会更准确。

*** /

//使用推荐模型导入训练数据进行训练

//显性模型

val modelExplicit = alsExplicit.fit(training)

//隐性模型

val modelImplicit = alsImplicit.fit(training)

//使用训练好的模型进行预测，对测试数据中的用户物品预测评分，获得预测评

//分数据集

//显性模型预测

val predictionsExplicit = modelExplicit.transform(test)

输出结果：

predictionsExplicit: org.apache.spark.sql.DataFrame = [userId: int, movieId: int ... 3 more fields]

//隐性模型预测

val predictionsImplicit = modelImplicit.transform(test)

输出结果：

predictionsImplicit: org.apache.spark.sql.DataFrame = [userId: int, movieId: int ... 3 more fields]

//输出结果，对比预测与真实结果

//显性模型

predictionsExplicit.show()

输出结果：

```
+-----+-----+-----+-----+
|userId|movieId|rating|timestamp| prediction|
+-----+-----+-----+-----+
|    13|     31|    1.0|1424380312|  0.86262053|
|     5|     31|    1.0|1424380312|-0.033763513|
|    24|     31|    1.0|1424380312|  2.3084288|
|    29|     31|    1.0|1424380312|  1.9081671|
|     0|     31|    1.0|1424380312|  1.6470298|
|    28|     85|    1.0|1424380312|  5.7112412|
|    13|     85|    1.0|1424380312|  2.4970412|
|    20|     85|    2.0|1424380312|  1.9727222|
|     4|     85|    1.0|1424380312|  1.8414592|
|     8|     85|    5.0|1424380312|  3.2290685|
```

| | | | | | |
|--|----|----|-----|------------|-----------|
| | 7 | 85 | 4.0 | 1424380312 | 2.8074787 |
| | 29 | 85 | 1.0 | 1424380312 | 0.7150749 |
| | 19 | 65 | 1.0 | 1424380312 | 1.7827456 |
| | 4 | 65 | 1.0 | 1424380312 | 2.3001173 |
| | 2 | 65 | 1.0 | 1424380312 | 4.8762875 |
| | 12 | 53 | 1.0 | 1424380312 | 1.5465991 |
| | 20 | 53 | 3.0 | 1424380312 | 1.903692 |
| | 19 | 53 | 2.0 | 1424380312 | 2.6036916 |
| | 8 | 53 | 5.0 | 1424380312 | 3.1105173 |
| | 23 | 53 | 1.0 | 1424380312 | 1.0042696 |

```
+-----+-----+-----+-----+-----+
```

only showing top 20 rows

//隐性模型

predictionsImplicit.show()

输出结果:

| | | | | | |
|--|----|----|-----|------------|-------------|
| | 13 | 31 | 1.0 | 1424380312 | 0.33150947 |
| | 5 | 31 | 1.0 | 1424380312 | -0.24669354 |
| | 24 | 31 | 1.0 | 1424380312 | -0.22434244 |
| | 29 | 31 | 1.0 | 1424380312 | 0.15776125 |
| | 0 | 31 | 1.0 | 1424380312 | 0.51940984 |
| | 28 | 85 | 1.0 | 1424380312 | 0.88610375 |
| | 13 | 85 | 1.0 | 1424380312 | 0.15872183 |
| | 20 | 85 | 2.0 | 1424380312 | 0.64086926 |
| | 4 | 85 | 1.0 | 1424380312 | -0.06314563 |
| | 8 | 85 | 5.0 | 1424380312 | 0.2783457 |
| | 7 | 85 | 4.0 | 1424380312 | 0.1618208 |
| | 29 | 85 | 1.0 | 1424380312 | -0.19970453 |
| | 19 | 65 | 1.0 | 1424380312 | 0.11606887 |
| | 4 | 65 | 1.0 | 1424380312 | 0.068018675 |
| | 2 | 65 | 1.0 | 1424380312 | 0.28533924 |
| | 12 | 53 | 1.0 | 1424380312 | 0.42327875 |
| | 20 | 53 | 3.0 | 1424380312 | 0.17345423 |
| | 19 | 53 | 2.0 | 1424380312 | 0.33321634 |
| | 8 | 53 | 5.0 | 1424380312 | 0.10090684 |
| | 23 | 53 | 1.0 | 1424380312 | 0.06724724 |

```
+-----+-----+-----+-----+-----+
```

only showing top 20 rows

//根据模型预测结果进行评估，评估标准是计算模型预测结果的均方根误差，误差越小，模型预测的准确度越高

```
val evaluator = new  
RegressionEvaluator().setMetricName("rmse").setLabelCol("rating").  
setPredictionCol("prediction")
```

```
//rmse: Root-mean-square error, 均方根误差  
//计算显性模型的 rmse  
val rmseExplicit = evaluator.evaluate(predictionsExplicit)  
输出结果:  
rmseExplicit: Double = 1.6995189118765517
```

```
//计算隐性模型的 rmse  
val rmseImplicit = evaluator.evaluate(predictionsImplicit)  
输出结果:  
rmseImplicit: Double = 1.8011620822359165
```

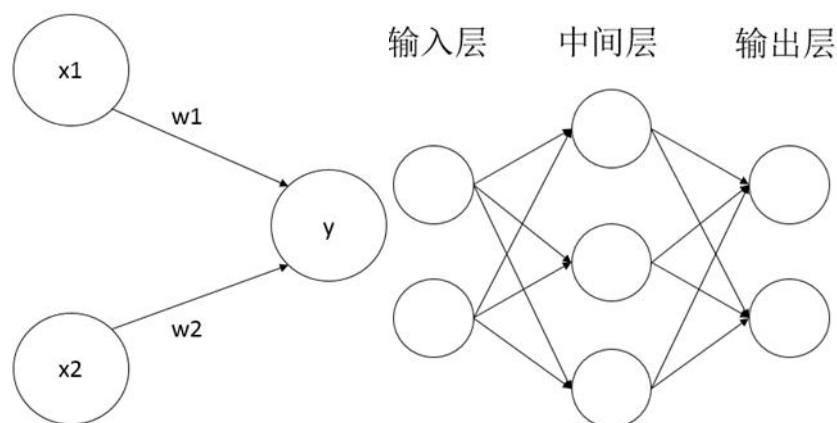
7. SparkMLlib 神经网络算法（2 课时）

7.1 人工神经网络算法

（1）人工神经网络

人工神经网络（Artificial Neural Networks——ANNs）提供了一种普遍而且实用的方法，来从样本中学习值为实数、离散或向量的函数。用反向传播（BackPropagation）这样的算法使用梯度下降来调节网络参数以最佳拟合由输入-输出对组成的训练集合。**ANN** 是基于神经元模型构建的，是一种由许多神经元组成的信息处理网络，具有并行分布结构。每一个神经元具有单一输出，且与其他神经元相连接，有多种输出连接方式，每种方式对应一个连接权重系数。

人工神经网络的研究在一定程度上受到了生物学的启发，因为生物的学习系统是由相互连接的神经元（neuron）组成的异常复杂的网络。而人工神经网络与此大体相似，它是由一系列简单单元相互密集连接构成，其中每一个单元有一定数量的实值输入（可能是其他单元的输出），并产生单一的实数值输出（可能成为其他很多单元的输入）。下图就是一个最简单的神经元的示意图，图右是由神经元前后连接构成的一个三层前馈神经网络，分为输入层，中间层也即隐藏层，输出层。从图中可以看到，输入为 x_1 、 x_2 ，输入权重分别为 w_1 、 w_2 ，代表输入对输出的影响程度。 y 为神经元的处理单元，加入激活函数，完成从输入到输出的映射。



常用的激活函数有 **sigmoid** 函数和双曲正切函数即 **tanh** 函数，若采用 **sigmoid** 函数则神经元的输入输出映射关系是一个逻辑回归。注意，这里是有一次见到这个神奇的函数了。

(2) 误差逆传播算法 BP

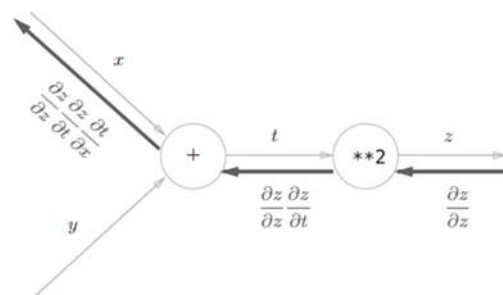
上面介绍了人工神经网络的基本原理和示意图，我们可以知道如果设定多个输入，增加层数和中间的隐藏层数，就可以构建一个复杂的从输入到输出的映射拓扑结构。其中最关键的部分就是层与层之间的权重系数。训练数据的输入，不断地去迭代就是去确定这些权重系数的方法，也就是神经网络的自我学习。神经网络从数据中进行学习，由数据自动决定网络参数的值。这里给出了一个寻找最优权重参数的指标——损失函数，常用均方误差来表示。神经网络对数据进行学习的目的就是不断调整参数，降低这个损失函数：

$$E = \frac{1}{2} \sum_k (y_k - t_k)^2$$

y_k 是预测值， t_k 是真实值， k 为输出值个数

均方误差会计算神经网络输出与数据真实值之差的平方和，每一次训练数据的输入到输出，都可以计算出一个均方误差，神经网络学习的目标就是使得这个均方误差的值越来越小，达到一定范围，即完成训练。神经网络中使用梯度下降的思想来寻找损失函数最小值进行最优化。

误差逆传播算法是最常用的 **ANN** 学习技术，它是从神经网络正向传播的反向来计算输出关于输入和权重的梯度值，从而找到如何调整权重来使得误差函数减小。误差逆传播法可高效计算权重参数的梯度，它基于链式法则，将局部导数进行反向传播，每经过一个结点，把次节点的局部导数乘以上游传过来的值，再传递给前面的节点，最后传递到整个网络的输入端。将逆传播法应用到神经网络中最小损失函数的求解中，即可得到损失函数对输入层每个输入的敏感程度，同时也可以得到网络中权重参数对损失函数的影响程度。下图即为用计算图来表示一个反向传播计算导数的过程。



而我们这里说的误差逆传播，就是指的损失函数关于网络权重的逆传播。

(3) 基于误差逆传播神经网络的实现

基于误差逆传播算法我们就可以完成一个神经网络的实现，如图是一个误差逆传播神经网络。其训练过程如下：

准备：在神经网络中设置好层数，输入神经元个数和输出神经元个数，并设置好合适的权重和偏置；

第一步：从训练数据中随机选择一部分数据；

第二步：使用误差逆传播法计算损失函数关于各个权重参数的梯度；

第三步：将权重参数沿梯度方向进行微小的更新；

第四步：对上述第一、二、三步进行重复

当损失函数减小到一定程度时停止迭代

输出：参数确定的逆差误传播人工神经网络

7.2 算法源码分析

MLlib 对神经网络部分的支持较为充分，源码部分分析如下：

(1) `class NeuralNet`: ANN 神经网络类，需要设置的参数：

- * `size`——数据格式: `Array[Int]`，神经网络每层的节点个数；
- * `layer`——神经网络层数；
- * `activation_function`: 隐含层函数，包括 `sigmoid` 和 `tanh`；
- * `learnRate`——学习率；
- * `momentum`——权重更新参数；
- * `inputZeroMaskedFraction`——训练数据中加入噪声；
- * `scaling_learningRate`——学习缩放因子；
- * `dropoutFraction`——隐含层节点加入噪声；
- * `testing`——内部变量，用于测试；
- * `output_fraction`——输出函数支持 `sigmoid`, `softmax`, `linear`；
- * `initW`——初始化权重

(2)

`Nntrain`: 训练模型的方法，基于梯度下降法进行权重优化计算；

`InitialWeight`: 初始化权重；

`NNtrain` 方法调用的方法；

`NNff`: 前向传播算法实现；

`NNbp`: 后向传播算法实现；

`Nnapplygrads`: 权重更新；

(3) `NeuralNetModel`: 神经网络模型类；

`Predict`: 预测计算；

`Loss`: 计算输出平均误差；

包含参数：权重、配置参数。

7.3 应用实战

(1) 测试数据

测试数据使用的是优化算法中的经典测试函数：

1.Sphere Model

函数表达式如下：

$$f_1(x) = \sum_{i=1}^n x_i^2$$

搜索范围： $-100 \leq x_i \leq 100$

全局最优值： $\min(f_1) = f_1(0, \dots, 0) = 0$

此函数是非线性对称单峰函数主要测试算法的寻优精度。

2.Generalized Rosenbrock

函数表达式如下：

$$f_2(x) = \sum_{i=1}^n \left[100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2 \right]$$

搜索范围： $-30 \leq x_i \leq 30$

全局最优值： $\min(f_2) = f_2(0, \dots, 0) = 0$

此函数是难于进行极小优化的病态二次函数，在其函数图像中，全局最优值与可达局部最优值之间有一道山谷，曲面山谷的点的最大下降方向与函数最小值的最好方向垂直。所以该函数在搜索过程中的优化方向难以确定，会使算法难以辨别正确的搜索方向。

3.Generalized Rastrigin

函数表达式如下：

$$f_3(x) = \sum_{i=1}^n [x_i^2 - 10 \cos(2\pi x_i) + 10]$$

搜索范围： $-5.12 \leq x_i \leq 5.12$

全局最优值： $\min(f_3) = f_3(0, \dots, 0) = 0$

此函数基于 Sphere 函数，通过余弦函数产生大量局部最小值，通过此函数的图像可以发现这是一个典型的具有大量局部最优点的复杂多峰函数，优化算法很容易陷入局部最优值从而得不到全局最优解。

(2) 测试函数代码

```
import java.util.Random
import breeze.linalg.{
  Matrix => BM,
  CSCMatrix => BSM,
  DenseMatrix => BDM,
  Vector => BV,
  DenseVector => BDV,
  SparseVector => BSV,
  axpy => brzAxy,
  svd => brzSvd
```



```

}
import breeze.numerics.{
  exp => Bexp,
  cos => Bcos,
  tanh => Btanh
}
import scala.math.Pi

object RandSampleData extends Serializable {
  // Rosenbrock:
  //  $\sum (100 \cdot (x(i+1) - x(i))^2 + (x(i) - 1)^2)$ 
  // Rastrigin:
  //  $\sum (x(i)^2 - 10 \cdot \cos(2 \cdot 3.14 \cdot x(i)) + 10)$ 
  // Sphere :
  //  $\sum (x(i)^2)$ 
  /**
   * 测试函数: Rosenbrock, Rastrigin
   * 随机生成 n2 维数据, 并根据测试函数计算 Y
   * n1 行, n2 列, b1 上限, b2 下限, function 计算函数
   */
  def RandM(
    n1: Int,
    n2: Int,
    b1: Double,
    b2: Double,
    function: String): BDM[Double] = {
    // val n1 = 2
    // val n2 = 3
    // val b1 = -30
    // val b2 = 30
    val bdm1 = BDM.rand(n1, n2) * (b2 - b1).toDouble + b1.toDouble
    val bdm_y = function match {
      case "rosenbrock" =>
        val xi0 = bdm1(:, 0 to (bdm1.cols - 2))
        val xi1 = bdm1(:, 1 to (bdm1.cols - 1))
        val xi2 = (xi0 :* xi0)
        val m1 = ((xi1 - xi2) :* (xi1 - xi2)) * 100.0 + ((xi0 - 1.0) :* (xi0 - 1.0))
        val m2 = m1 * BDM.ones[Double](m1.cols, 1)
        m2
      case "rastrigin" =>
        val xi0 = bdm1
        val xi2 = (xi0 :* xi0)
        val sicos = Bcos(xi0 * 2.0 * Pi) * 10.0
        val m1 = xi2 - sicos + 10.0
    }
  }

```

```

        val m2 = m1 * BDM.ones[Double](m1.cols, 1)
        m2
    case "sphere" =>
        val xi0 = bdm1
        val xi2 = (xi0 :* xi0)
        val m1 = xi2
        val m2 = m1 * BDM.ones[Double](m1.cols, 1)
        m2
    }
    val randm = BDM.horzcata(bdm_y, bdm1)
    randm
}
}

```

(3) 示例代码详解

基于上述的优化算法测试函数随机生成样本，使用神经网络寻找最优值，观察输出结果：

//导入所需的机器学习包和 Spark 基础的支持库

```

import org.apache.log4j.{ Level, Logger }
import org.apache.spark.{ SparkConf, SparkContext }
import org.apache.spark.storage.StorageLevel
import org.apache.spark.mllib.util.MLUtils
import org.apache.spark.mllib.linalg.{ Vector, Vectors }
import org.apache.spark.mllib.linalg.distributed.RowMatrix
import org.apache.spark.mllib.regression.LabeledPoint
import breeze.linalg.{
    Matrix => BM,
    CSCMatrix => BSM,
    DenseMatrix => BDM,
    Vector => BV,
    DenseVector => BDV,
    SparseVector => BSV,
    axpy => brzAxy,
    svd => brzSvd,
    max => Bmax,
    min => Bmin,
    sum => Bsum
}
import scala.collection.mutable.ArrayBuffer
import NN.NeuralNet
import util.RandSampleData

```

```

object Test_example_NN {

```

```

def main(args: Array[String]) {
  //1 构建 Spark 对象
  val conf = new SparkConf().setAppName("NNtest")
  val sc = new SparkContext(conf)

  //基于经典优化算法测试函数随机生成样本
  //2 随机生成测试数据
  // 随机数生成
  Logger.getRootLogger.setLevel(Level.WARN)
  val sample_n1 = 1000
  val sample_n2 = 5
  //在此处选择所需的随机数生成函数，从前述的函数中
  val randsamp1 = RandSampleData.RandM(sample_n1, sample_n2, -10, 10,
"sphere")
  // 归一化[0 1]
  val normmax = Bmax(randsamp1(:, breeze.linalg.*))
  val normmin = Bmin(randsamp1(:, breeze.linalg.*))
  val norm1 = randsamp1 - (BDM.ones[Double](randsamp1.rows, 1)) * normmin
  val norm2 = norm1 ./ ((BDM.ones[Double](norm1.rows, 1)) * (normmax -
normmin))
  // 转换样本 train_d
  val randsamp2 = ArrayBuffer[BDM[Double]]()
  for (i <- 0 to sample_n1 - 1) {
    val mi = norm2(i, :)
    val mi1 = mi.inner
    val mi2 = mi1.toArray
    val mi3 = new BDM(1, mi2.length, mi2)
    randsamp2 += mi3
  }
  val randsamp3 = sc.parallelize(randsamp2, 10)
  sc.setCheckpointDir("/user/local/checkpoint")
  randsamp3.checkpoint()
  val train_d = randsamp3.map(f => (new BDM(1, 1, f(:, 0).data), f(:, 1 to -1)))
  //3 设置训练参数，建立模型
  // opts:迭代步长，迭代次数，交叉验证比例
  val opts = Array(100.0, 50.0, 0.0)
  train_d.cache
  val numExamples = train_d.count()
  println(s"numExamples = $numExamples.")
  val NNmodel = new NeuralNet().
    setSize(Array(5, 7, 1)).
    setLayer(3).
    setActivation_function("tanh_opt").
    setLearningRate(2.0).

```

```

    setScaling_learningRate(1.0).
    setWeightPenaltyL2(0.0).
    setNonSparsityPenalty(0.0).
    setSparsityTarget(0.05).
    setInputZeroMaskedFraction(0.0).
    setDropoutFraction(0.0).
    setOutput_function("sigmoid").
    NNtrain(train_d, opts)

//4 模型测试
val NNforecast = NNmodel.predict(train_d)
val NNerror = NNmodel.Loss(NNforecast)
println(s"NNerror = $NNerror.")
val printf1 = NNforecast.map(f => (f.label.data(0),
f.predict_label.data(0))).take(20)
println("预测结果——实际值: 预测值: 误差")
for (i <- 0 until printf1.length)
    println(printf1(i)._1 + "\t" + printf1(i)._2 + "\t" + (printf1(i)._2 - printf1(i)._1))
println("权重 W{1}")
val tmpw0 = NNmodel.weights(0)
for (i <- 0 to tmpw0.rows - 1) {
    for (j <- 0 to tmpw0.cols - 1) {
        print(tmpw0(i, j) + "\t")
    }
    println()
}
println("权重 W{2}")
val tmpw1 = NNmodel.weights(1)
for (i <- 0 to tmpw1.rows - 1) {
    for (j <- 0 to tmpw1.cols - 1) {
        print(tmpw1(i, j) + "\t")
    }
    println()
}
}
}
}

```