Linux下hadoop2+spark2的安装教程

教程包含以下部分：

- VMware下载/安装/使用
- linux下的java8环境配置（给定的虚拟机中已经配好）
- Hadoop2下载/安装（虚拟机中已安装）
- Hadoop2单机模式配置
- Hadoop2伪分布式模式配置（虚拟机中的配置为伪分布式）
- Hadoop在伪分布式模式下的wordCount示例于HDFS文件上传示例
- Spark2下载/安装/配置（虚拟机中已安装）
- 使用spark.mllib对鸢尾花数据集进行聚类（Scala）

VMware下载

部署hadoop最好在linux上部署，因此需要虚拟机软件来使用linux系统。可以在vmware的官网https://www.vmware.com/cn.html下载vmware workstation

如果机房有vmware 的话就不用下载了，直接使用即可。

VMware安装/使用镜像

点击创建虚拟机

导入iso文件

VMware安装/使用镜像

选择ubuntu 64位

处理器可以稍微

多一点，免得卡顿

剩下的选择默认配置即可，至此我们便创建出了新的系统。

Java安装与配置

首先到oracle官网下载jdk8

Linux x64 Compressed Archive	136.51 MB	jdk-8u271-linux-x64.tar.gz

把下载下来的jdk压缩包复制到“/home/hadoop/Downloads/”目录下，然后执行以下命令：

```
cd /usr/lib
sudo mkdir jvm                #创建/usr/lib/jvm目录用来存放JDK文件
cd ~                          #进入hadoop用户的主目录
cd Downloads                  #注意区分大小写字母
sudo tar -zxvf ./jdk-8u162-linux-x64.tar.gz -C /usr/lib/jvm
#把JDK文件解压到/usr/lib/jvm目录下
```

Java安装与配置

然后需要配置环境变量，输入以下命令：

cd ~

vim ~/.bashrc　（对vim不熟可以用gedit ~/.bashrc）

在文件开头添加如下内容：

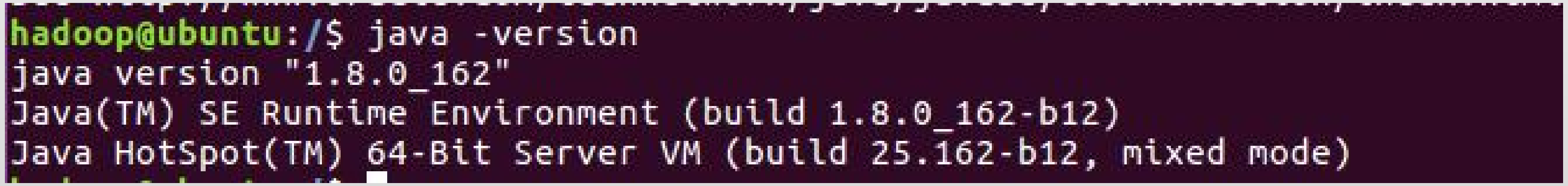
export JAVA_HOME=/usr/lib/jvm/jdk1.8.0_162

export JRE_HOME=${JAVA_HOME}/jre

export CLASSPATH=.:${JAVA_HOME}/lib:${JRE_HOME}/lib

export PATH=${JAVA_HOME}/bin:$PATH

然后输入java -version，如果显示版本则配置成功



Hadoop2下载/安装

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，一般选择下载最新的稳定版本，即下载 “stable” 下的 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。

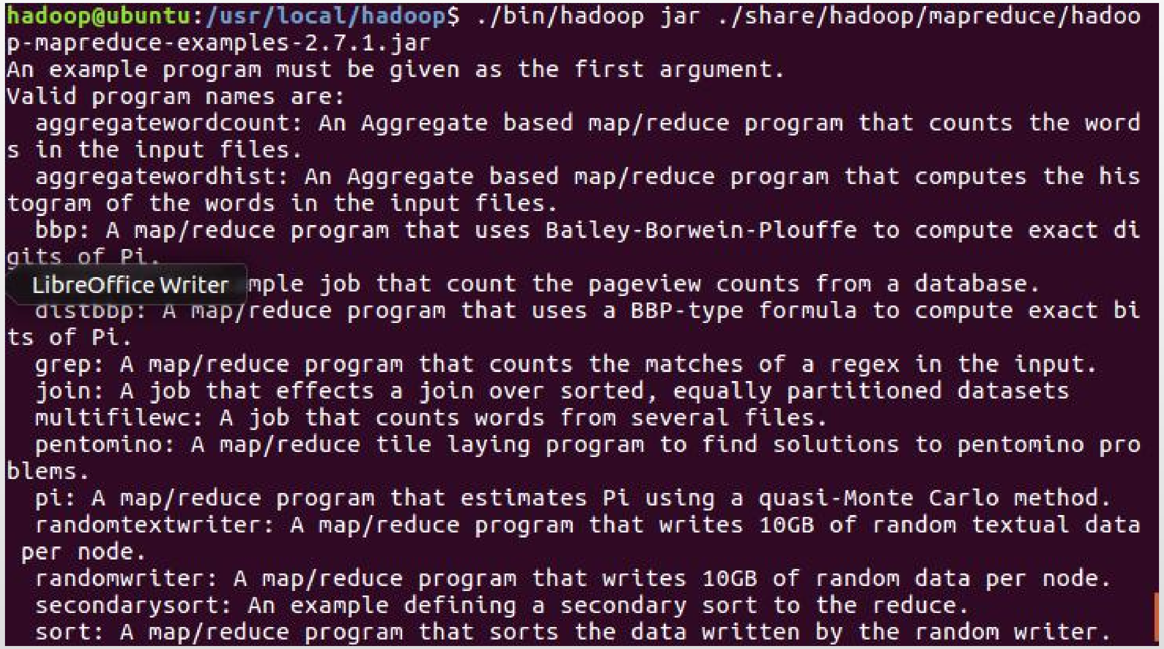
我们将Hadoop2安装在/usr/local中，先将压缩包粘贴到usr/Downloads中，然后执行如下命令：

```
sudo tar -zxf ~/Downloads/hadoop-2.6.0.tar.gz -C /usr/local          # 解压到/usr/local中
cd /usr/local/
sudo mv ./hadoop-2.6.0/ ./hadoop                                      # 将文件夹名改为hadoop
sudo chown -R hadoop ./hadoop                                         # 修改文件权限
```

Hadoop单机配置(非分布式)

- Hadoop 默认模式为非分布式模式（本地模式），无需进行其他配置即可运行。非分布式即单 Java 进程，方便进行调试。
- 例1：./bin/hadoop jar ./share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.1.jar

  显示了hadoop-mapreduce-examples-2.7.1.jar文件里所包含的函数



Hadoop单机配置(非分布式)

- Hadoop 默认模式为非分布式模式（本地模式），无需进行其他配置即可运行。非分布式即单 Java 进程，方便进行调试。
- 例2：在此我们选择运行 grep 例子，我们将 input 文件夹中的所有文件作为输入，筛选当中符合正则表达式 dfs[a-z.]+ 的单词并统计出现的次数，最后输出结果到 output 文件夹中

```
cd /usr/local/hadoop

mkdir ./input

cp ./etc/hadoop/*.xml ./input   # 将配置文件作为输入文件

./bin/hadoop jar ./share/hadoop/mapreduce/hadoop-mapreduce-examples-*.jar grep ./input ./output 'dfs[a-z.]+'

cat ./output/*
```

Hadoop伪分布式配置

- Hadoop 可以在单节点上以伪分布式的方式运行，Hadoop 进程以分离的 Java 进程来运行，节点既作为 NameNode 也作为 DataNode，同时，读取的是 HDFS 中的文件。
- Hadoop 的配置文件位于 /usr/local/hadoop/etc/hadoop/ 中，伪分布式需要修改2个配置文件 core-site.xml 和 hdfs-site.xml 。Hadoop的配置文件是 xml 格式，每个配置以声明 property 的 name 和 value 的方式来实现。
- 修改配置文件 core-site.xml (通过 gedit 编辑会比较方便: gedit ./etc/hadoop/core-site.xml)，将当中的

```xml
<configuration>
</configuration>
```

```xml
<configuration>
    <property>
        <name>hadoop.tmp.dir</name>
        <value>file:/usr/local/hadoop/tmp</value>
        <description>Abase for other temporary directories.</description>
    </property>
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://localhost:9000</value>
    </property>
</configuration>
```

Hadoop伪分布式配置

- 修改配置文件 hdfs-site.xml

```xml
<configuration>
</configuration>
```

```xml
<configuration>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
    <property>
        <name>dfs.namenode.name.dir</name>
        <value>file:/usr/local/hadoop/tmp/dfs/name</value>
    </property>
    <property>
        <name>dfs.datanode.data.dir</name>
        <value>file:/usr/local/hadoop/tmp/dfs/data</value>
    </property>
</configuration>
```

Hadoop伪分布式配置

- Hadoop配置文件说明
- Hadoop 的运行方式是由配置文件决定的（运行 Hadoop 时会读取配置文件），因此如果需要从伪分布式模式切换回非分布式模式，需要删除 core-site.xml 中的配置项。

- 此外，伪分布式虽然只需要配置 fs.defaultFS 和 dfs.replication 就可以运行（官方教程如此），不过若没有配置 hadoop.tmp.dir 参数，则默认使用的临时目录为 /tmp/hadoo-hadoop，而这个目录在重启时有可能被系统清理掉，导致必须重新执行 format 才行。所以我们进行了设置，同时也指定 dfs.namenode.name.dir 和 dfs.datanode.data.dir，否则在接下来的步骤中可能会出错。

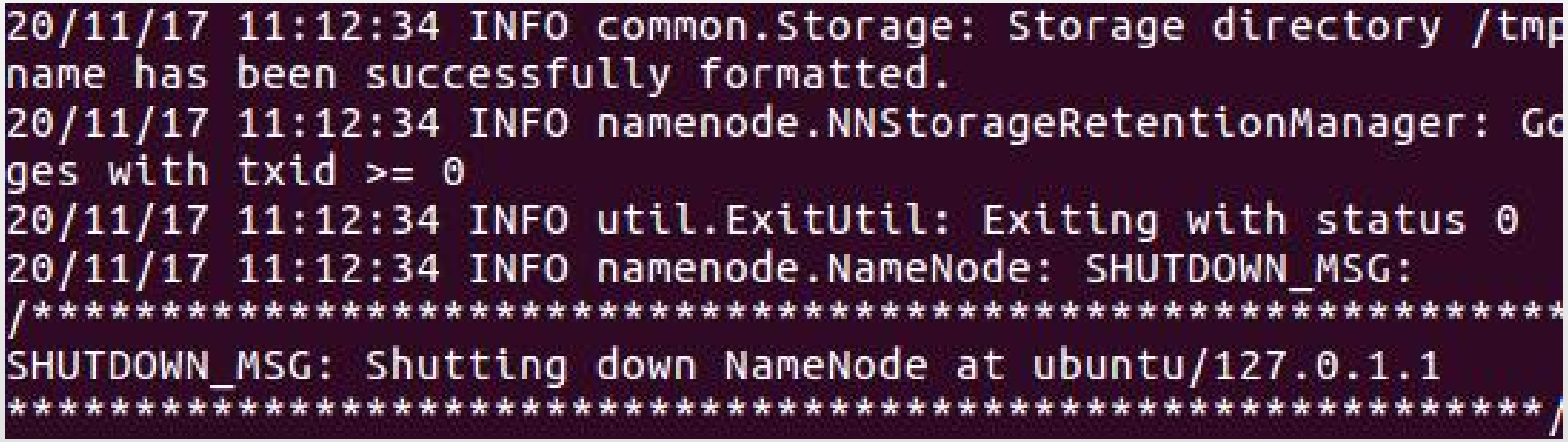
Hadoop伪分布式配置

● 配置完成后，执行 NameNode 的格式化

cd /usr/local/hadoop

./bin/hdfs namenode -format

成功的话，会看到 “successfully formatted” 和 “Exitting with status 0” 的提示，若为 “Exitting with status 1” 则是出错。



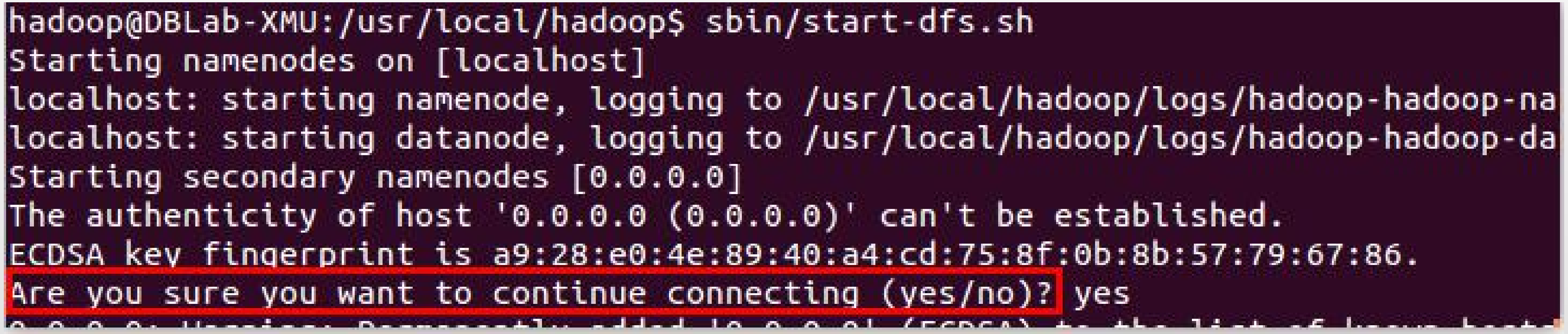
Hadoop伪分布式配置

● 接着开启 NameNode 和 DataNode 守护进程。

cd /usr/local/hadoop

./sbin/start-dfs.sh

若出现如下SSH提示，输入yes即可。



Hadoop伪分布式配置

- 启动完成后，可以通过命令 jps 来判断是否成功启动，若成功启动则会列出如下进程:“NameNode”、” DataNode” 和 “SecondaryNameNode” （如果 SecondaryNameNode 没有启动，请运行 sbin/stop-dfs.sh 关闭进程，然后再次尝试启动尝试）。如果没有 NameNode 或 DataNode ，那就是配置不成功，请仔细检查之前步骤，或通过查看启动日志排查原因。

```
hadoop@powerxing-M1:/usr/local/hadoop$ jps
7100 Jps
6867 SecondaryNameNode
6445 NameNode
6594 DataNode
```

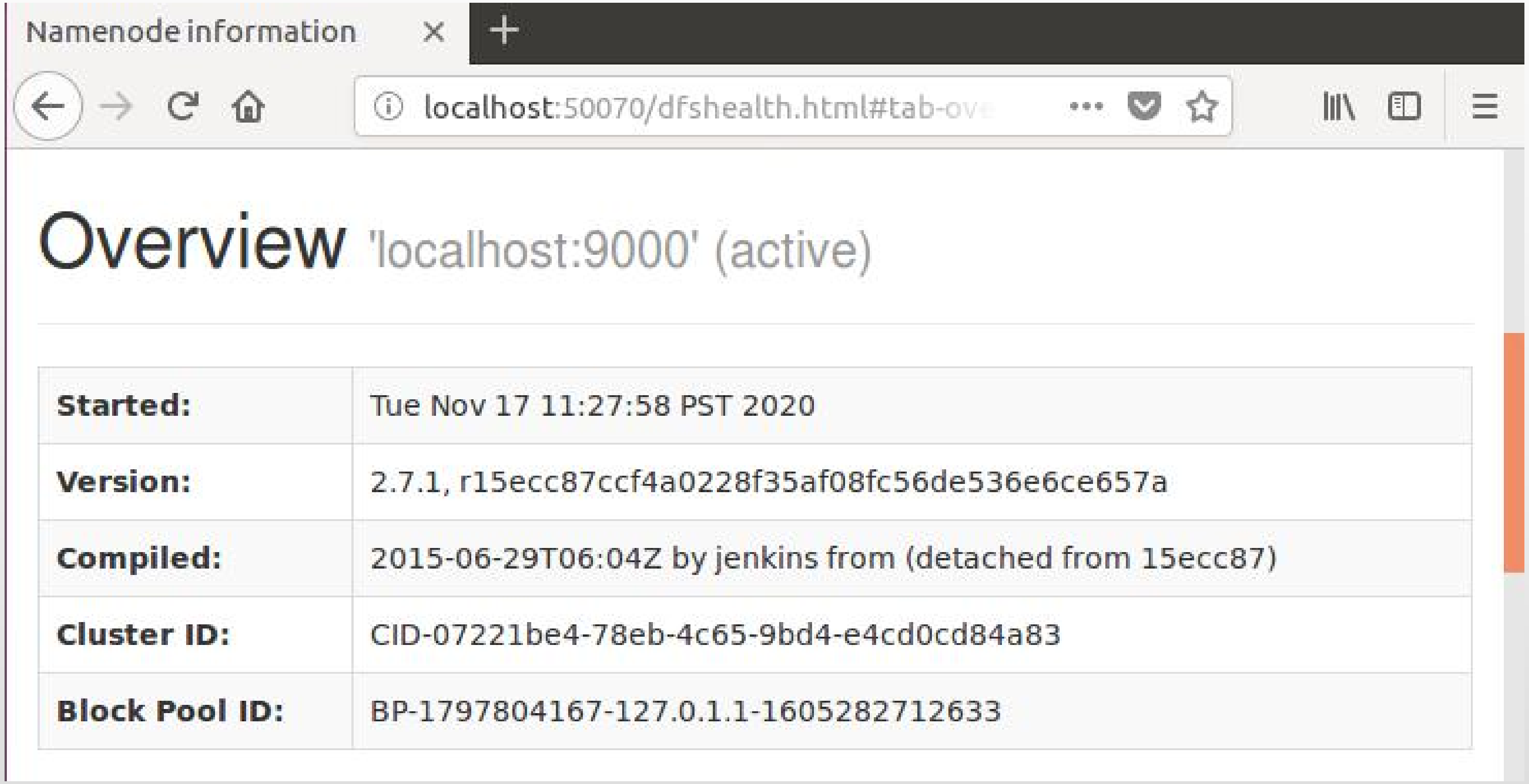
Hadoop伪分布式配置

- 若是 DataNode 没有启动，可尝试如下的方法（注意这会删除 HDFS 中原有的所有数据，如果原有的数据很重要请不要这样做）：

```
# 针对 DataNode 没法启动的解决方法
cd /usr/local/hadoop
./sbin/stop-dfs.sh          # 关闭
rm -r ./tmp             # 删除 tmp 文件，注意这会删除 HDFS 中原有的所有数据
./bin/hdfs namenode -format          # 重新格式化 NameNode
./sbin/start-dfs.sh       # 重启
```

Hadoop伪分布式配置

- 成功启动后，可以访问 Web 界面 http://localhost:50070 查看 NameNode 和 Datanode 信息，还可以在线查看 HDFS 中的文件。



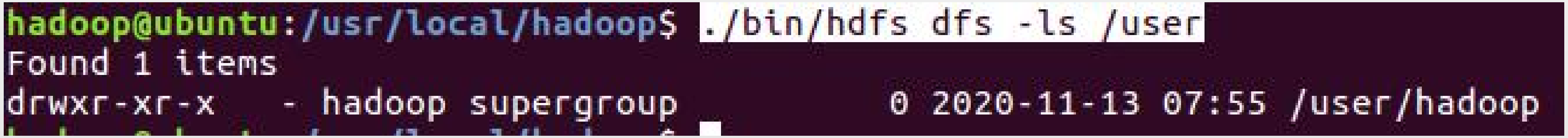
运行Hadoop伪分布式实例

- 上面的单机模式，grep 例子读取的是本地数据，伪分布式读取的则是 HDFS 上的数据。要使用 HDFS，首先需要在 HDFS 中创建用户目录：

./bin/hdfs dfs -mkdir -p /user/hadoop

运行成功后

./bin/hdfs dfs -ls /user #查看创建结果



运行Hadoop伪分布式实例

- 接着将 ./etc/hadoop 中的 xml 文件作为输入文件复制到分布式文件系统中，即将 /usr/local/hadoop/etc/hadoop 复制到分布式文件系统中的 /user/hadoop/input 中。我们使用的是 hadoop 用户，并且已创建相应的用户目录 /user/hadoop ，因此在命令中就可以使用相对路径如 input，其对应的绝对路径就是 /user/hadoop/input:

./bin/hdfs dfs -mkdir input

./bin/hdfs dfs -put ./etc/hadoop/*.xml input

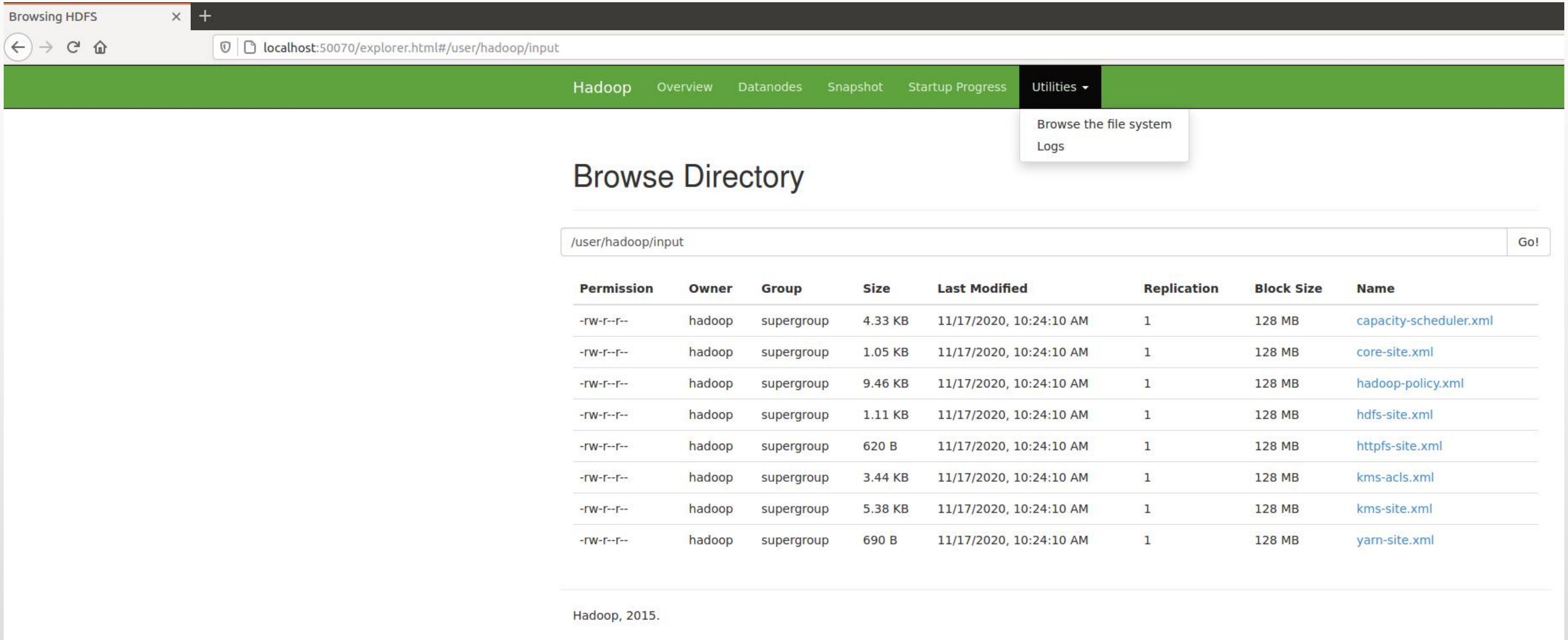
复制完成后，可以通过如下命令查看文件列表

./bin/hdfs dfs -ls input

```
hadoop@ubuntu:/usr/local/hadoop$ ./bin/hdfs dfs -ls input
Found 8 items
-rw-r--r--   1 hadoop supergroup       4436 2020-11-13 07:54 input/capacity-sche
duler.xml
-rw-r--r--   1 hadoop supergroup       1075 2020-11-13 07:54 input/core-site.xml
-rw-r--r--   1 hadoop supergroup       9683 2020-11-13 07:54 input/hadoop-policy
.xml
-rw-r--r--   1 hadoop supergroup       1133 2020-11-13 07:54 input/hdfs-site.xml
-rw-r--r--   1 hadoop supergroup        620 2020-11-13 07:54 input/httpfs-site.x
ml
-rw-r--r--   1 hadoop supergroup       3518 2020-11-13 07:54 input/kms-acls.xml
-rw-r--r--   1 hadoop supergroup       5511 2020-11-13 07:54 input/kms-site.xml
-rw-r--r--   1 hadoop supergroup        690 2020-11-13 07:54 input/yarn-site.xml
```

运行Hadoop伪分布式实例

也可以在浏览器里看，打开localhost:50070，点开Utilities里的file system，就能看到刚刚上传的文件。（这些配置文件在虚拟机中已经上传）



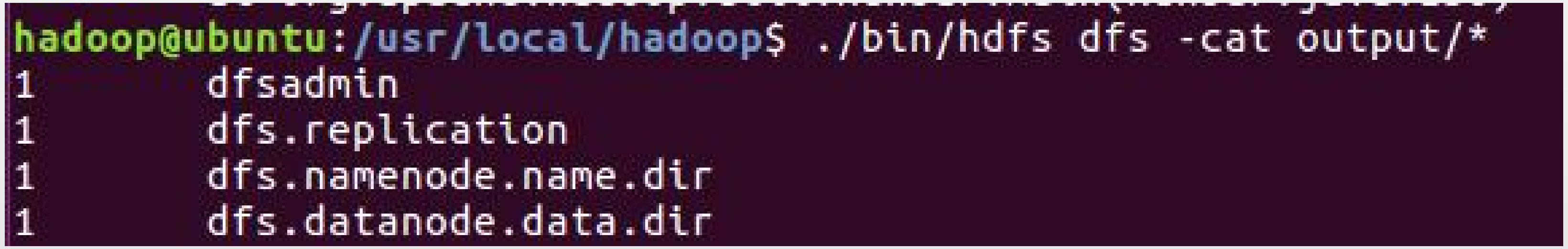
Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
-rw-r--r--	hadoop	supergroup	4.33 KB	11/17/2020, 10:24:10 AM	1	128 MB	capacity-scheduler.xml
-rw-r--r--	hadoop	supergroup	1.05 KB	11/17/2020, 10:24:10 AM	1	128 MB	core-site.xml
-rw-r--r--	hadoop	supergroup	9.46 KB	11/17/2020, 10:24:10 AM	1	128 MB	hadoop-policy.xml
-rw-r--r--	hadoop	supergroup	1.11 KB	11/17/2020, 10:24:10 AM	1	128 MB	hdfs-site.xml
-rw-r--r--	hadoop	supergroup	620 B	11/17/2020, 10:24:10 AM	1	128 MB	httpfs-site.xml
-rw-r--r--	hadoop	supergroup	3.44 KB	11/17/2020, 10:24:10 AM	1	128 MB	kms-acls.xml
-rw-r--r--	hadoop	supergroup	5.38 KB	11/17/2020, 10:24:10 AM	1	128 MB	kms-site.xml
-rw-r--r--	hadoop	supergroup	690 B	11/17/2020, 10:24:10 AM	1	128 MB	yarn-site.xml

运行Hadoop伪分布式实例

- 伪分布式运行 MapReduce 作业的方式跟单机模式相同，区别在于伪分布式读取的是HDFS中的文件（可以将单机步骤中创建的本地 input 文件夹，输出结果 output 文件夹都删掉来验证这一点）。

./bin/hadoop jar ./share/hadoop/mapreduce/hadoop-mapreduce-examples-*.jar grep input output 'dfs[a-z.]+'

查看运行结果的命令（查看的是位于 HDFS 中的输出结果）：



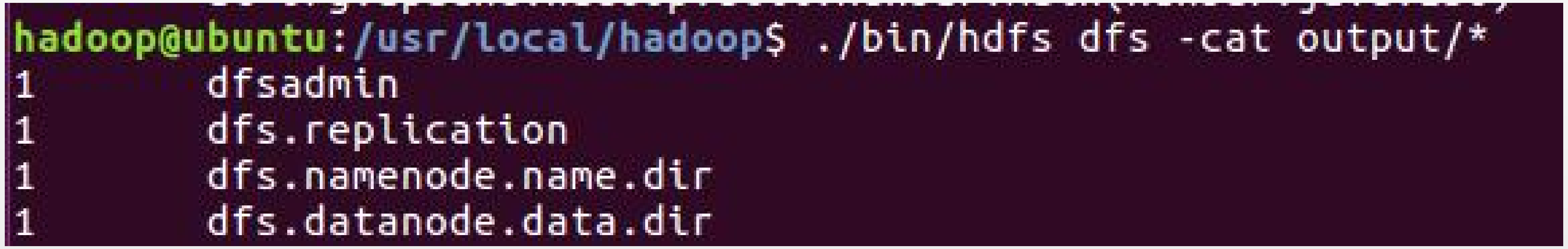
运行Hadoop伪分布式实例

- 我们也可以将运行结果取回到本地：

rm -r ./output   # 先删除本地的 output 文件夹（如果存在）

./bin/hdfs dfs -get output ./output   # 将 HDFS 上的 output 文件夹拷贝到本机

cat ./output/*



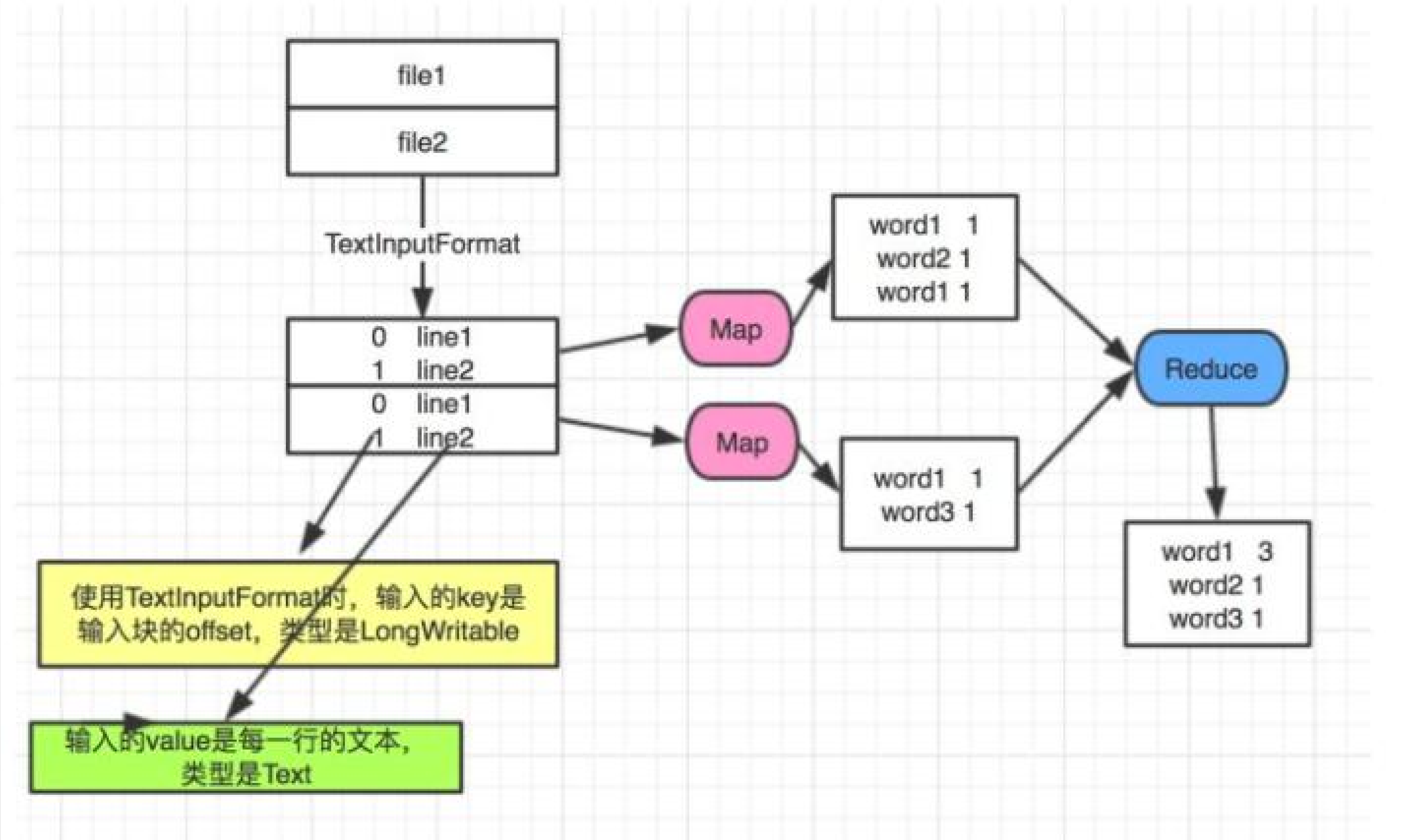
若要关闭 Hadoop，则运行

./sbin/stop-dfs.sh

运行Hadoop伪分布式实例

wordcount源码：

运行wordcount函数统计给定的test.txt文档中的单词数。



运行Hadoop伪分布式实例

wordcount源码：

Mapper(14-26行)中的map方法(18-25行)通过指定的 TextInputFormat(49行)一次处理一行。然后，它通过StringTokenizer 以空格为分隔符将一行切分为若干tokens，之后，输出< <word>，1> 形式的键值对。

WordCount还指定了一个combiner (46行)。因此，每次map运行之后，会对输出按照key进行排序，然后把输出传递给本地的combiner（按照作业的配置与Reducer一样），进行本地聚合。

Reducer(28-36行)中的reduce方法(29-35行) 仅是将每个key（本例中就是单词）出现的次数求和。

代码中的run方法中指定了作业的几个方面， 例如：通过命令行传递过来的输入/输出路径、key/value的类型、输入/输出的格式等等JobConf中的配置信息。

随后程序调用了obClient.runJob(55行)来提交作业并且监控它的执行。

core-site.xml

```
<configuration>
    <property>
        <name>hadoop.tmp.dir</name>
        <value>file:/usr/local/hadoop/tmp</value>
        <description>Abase for other temporary directories.</description>
    </property>
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://localhost:9000</value>
    </property>
</configuration>
```

hdfs-site.xml

```xml
<configuration>
  <property>
    <name>dfs.replication</name>
    <value>1</value>
  </property>
  <property>
    <name>dfs.namenode.name.dir</name>
    <value>file:/usr/local/hadoop/tmp/dfs/name</value>
  </property>
  <property>
    <name>dfs.datanode.data.dir</name>
```

Spark安装/配置

访问官网http://spark.apache.org/downloads.html，下载Spark2，由于我们已经装了Hadoop2，所以选择pre-built版本

下载完了放到Downloads文件夹里，并执行如下命令，把Spark安装到/usr/local：

```
sudo tar -zxf ~/Downloads/spark-2.1.0-bin-without-hadoop.tgz -C /usr/local/
cd /usr/local
sudo mv ./spark-2.1.0-bin-without-hadoop/ ./spark
sudo chown -R hadoop:hadoop ./spark        # 此处的 hadoop 为你的用户名
```

Spark安装/配置

修改过权限以后（上页最后一条命令），需要修改配置文件，执行以下命令：

cd /usr/local/spark

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

vim ./conf/spark-env.sh（或者gedit）

在文件的第一行加上：

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

然后执行以下spark自带的demo，看是否安装成功：

cd /usr/local/spark

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

```
hadoop@ubuntu:/usr/local/spark$ bin/run-example SparkPi 2>&1 | grep "Pi is"
Pi is roughly 3.149395746978735
```

利用Spark中的机器学习库Mllib对鸢尾花数据进行聚类

示例中所使用的数据集为鸢尾花数据集，包含150条数据，每条数据包含4个特征和一个标签（分类结果）

```
6.3,2.9,5.6,1.8,Iris-virginica
6.5,3.0,5.8,2.2,Iris-virginica
7.6,3.0,6.6,2.1,Iris-virginica
4.9,2.5,4.5,1.7,Iris-virginica
7.3,2.9,6.3,1.8,Iris-virginica
6.7,2.5,5.8,1.8,Iris-virginica
7.2,3.6,6.1,2.5,Iris-virginica
6.5,3.2,5.1,2.0,Iris-virginica
6.4,2.7,5.3,1.9,Iris-virginica
6.8,3.0,5.5,2.1,Iris-virginica
5.7,2.5,5.0,2.0,Iris-virginica
5.8,2.8,5.1,2.4,Iris-virginica
6.4,3.2,5.3,2.3,Iris-virginica
6.5,3.0,5.5,1.8,Iris-virginica
7.7,3.8,6.7,2.2,Iris-virginica
7.7,2.6,6.9,2.3,Iris-virginica
6.0,2.2,5.0,1.5,Iris-virginica
6.9,3.2,5.7,2.3,Iris-virginica
5.6,2.8,4.9,2.0,Iris-virginica
7.7,2.8,6.7,2.0,Iris-virginica
```

```
7.0,3.2,4.7,1.4,Iris-versicolor
6.4,3.2,4.5,1.5,Iris-versicolor
6.9,3.1,4.9,1.5,Iris-versicolor
5.5,2.3,4.0,1.3,Iris-versicolor
6.5,2.8,4.6,1.5,Iris-versicolor
5.7,2.8,4.5,1.3,Iris-versicolor
6.3,3.3,4.7,1.6,Iris-versicolor
4.9,2.4,3.3,1.0,Iris-versicolor
6.6,2.9,4.6,1.3,Iris-versicolor
5.2,2.7,3.9,1.4,Iris-versicolor
5.0,2.0,3.5,1.0,Iris-versicolor
5.9,3.0,4.2,1.5,Iris-versicolor
6.0,2.2,4.0,1.0,Iris-versicolor
```

```
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,Iris-setosa
4.8,3.0,1.4,0.1,Iris-setosa
```

150条数据被划分为3类，每类有50条数据

1.开启spark-shell

在命令行中输入/usr/local/spark/bin/spark-shell，开启spark-shell，使用scala语言编写程序

```
hadoop@ubuntu:~$ /usr/local/spark/bin/spark-shell
20/11/18 10:22:13 WARN util.Utils: Your hostname, ubuntu resolves to a loopback address: 127.0.1.1; using 192.168.237.133 instead (on interface ens33)
20/11/18 10:22:13 WARN util.Utils: Set SPARK_LOCAL_IP if you need to bind to another address
20/11/18 10:22:14 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Spark context Web UI available at http://192.168.237.133:4040
Spark context available as 'sc' (master = local[*], app id = local-1605723742294).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.4.7
      /_/

Using Scala version 2.11.12 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_162)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

2.在交互式命令行中输入程序

导入spark的机器学习库Mllib

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

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

KMeans是用到的模型，Vectors是模型所需要的向量

读取鸢尾花数据集

```
scala> val rawData = sc.textFile("file:///home/hadoop/Desktop/iris.txt")
rawData: org.apache.spark.rdd.RDD[String] = file:///home/hadoop/Desktop/iris.txt MapPartitionsRDD[3] at textFile at <console>:27
```

此处假定数据放在hadoop用户的桌面上，文件名为iris.txt

使用sc.textFile读取文件并形成RDD
从本地读取文件的URL格式为“file:/// + 文件位置”，如果不加”file:///”则默认从hdfs中读取

对原始数据进行处理

```
scala> val trainingData = rawData.map(line => {Vectors.dense(line.split(",").filter(p => p.matches("\\d*(\\.?)\\d*")).map(_.toDouble))}).cache()
trainingData: org.apache.spark.rdd.RDD[org.apache.spark.mllib.linalg.Vector] = MapPartitionsRDD[6] at map at <console>:28
```

通过filter算子过滤掉源文件中的分类结果；正则表达式\\d*(\\.?)\\d*可以用于匹配实数类型的数字，\\d*使用了*限定符，表示匹配0次或多次的数字字符，\\.?使用了?限定符，表示匹配0次或1次的小数点

因为K-means属于无监督学习，所以说此处去掉标签数据训练模型

```
val trainingData = rawData.map(                  //对rawData的每一个元素做以下操作
    line => {                                    //对rawData的每一行（用testFile获取的RDD每一行为一个元素）
        Vectors.dense(                           //把每一行变成一个Vector，这个Vector用Vectors.dense()函数生成
            line.split(",").filter(              //向量中的元素由line.split(",")经过filter过滤后剩下的元素构成
                p => p.matches("\\d*(\\.?)\\d*") //过滤掉非实数的元素
            ).map(_.toDouble)                    //把Vector中的每一个元素变为double型
        )
    }
).cache()                                        //该RDD存放在内存中
```

RDD.map()	参数是函数，函数应用于RDD每一个元素，返回值是新的RDD	Vectors.dense()	参数为一个数组(列表)，返回相应的稠密向量
RDD.filter()	参数是函数，函数会过滤掉不符合条件的元素，返回值是新的RDD	String.matches()	参数为正则表达式，返回字符串中符合正则表达式部分的内容

查看原始数据和去除标签之后的数据

```
scala> rawData.collect().foreach{println}
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,Iris-setosa
4.8,3.0,1.4,0.1,Iris-setosa
4.3,3.0,1.1,0.1,Iris-setosa
```

```
scala> trainingData.collect().foreach {println}
[5.1,3.5,1.4,0.2]
[4.9,3.0,1.4,0.2]
[4.7,3.2,1.3,0.2]
[4.6,3.1,1.5,0.2]
[5.0,3.6,1.4,0.2]
[5.4,3.9,1.7,0.4]
[4.6,3.4,1.4,0.3]
[5.0,3.4,1.5,0.2]
[4.4,2.9,1.4,0.2]
[4.9,3.1,1.5,0.1]
[5.4,3.7,1.5,0.2]
[4.8,3.4,1.6,0.2]
[4.8,3.0,1.4,0.1]
[4.3,3.0,1.1,0.1]
[5.8,4.0,1.2,0.2]
[5.7,4.4,1.5,0.4]
```

RDD.collect()	返回RDD中的所有元素	RDD.foreach()	对 RDD 的每个元素都使用特定函数

设定超参数并训练

```
scala> val numClusters = 2
numClusters: Int = 2

scala> val numIterations = 5
numIterations: Int = 5

scala> val model : KMeansModel = KMeans.train(trainingData, numClusters, numIterations)
20/11/18 10:43:07 WARN netlib.BLAS: Failed to load implementation from: com.github.fommil.netlib.NativeSystemBLAS
20/11/18 10:43:07 WARN netlib.BLAS: Failed to load implementation from: com.github.fommil.netlib.NativeRefBLAS
model: org.apache.spark.mllib.clustering.KMeansModel = org.apache.spark.mllib.clustering.KMeansModel@67174f09
```

此处设定聚类结果中类的个数（聚类中称为簇）为2，迭代次数为5，输入到KMeans的train方法中进行训练，模型参数保存在model中。
（中间的两行warn中无法加载的工具是加快运算速度的，不影响结果）

查看训练好的模型中每个类别的中心点

```
scala> for (c <- model.clusterCenters) {
     |     println("  " + c.toString)
     | }
  [6.305208333333331,2.885416666666667,4.957291666666667,1.6947916666666663]
  [5.005660377358491,3.3603773584905667,1.562264150943396,0.28867924528301875]
```

对数据集中每一个数据分类

```
scala> trainingData.collect().foreach(
     |      sample => {
     |        val predictedCluster = model.predict(sample)
     |        println(sample.toString + " belongs to cluster " + predictedCluster)
     | })
[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
[5.4,3.9,1.7,0.4] belongs to cluster 1
[4.6,3.4,1.4,0.3] belongs to cluster 1
```

对新输入的点进行分类

```
scala> println("Vectors 5.1,3.5,1.4,0.2 is belongs to clusters:" + model.predict(Vectors.dense("5.1,3.5,1.4,0.2".split(',').map(_.toDouble))))
Vectors 5.1,3.5,1.4,0.2 is belongs to clusters:1
```

使用Vector.dense创建新的向量，输入到模型的predict方法中进行分类

参考技术博客：https://blog.csdn.net/sartinl/article/details/108060242

利用Spark中的机器学习库Mllib对数据进行回归

```
-0.4307829,-1.63735562648104 -2.00621178480549 -1.86242597251066 -1.02470580167082 -0.522940888712441 -0.863171185425945 -1.04215728919298 -0.864466507337306
-0.1625189,-1.98898046126935 -0.722008756122123 -0.787896192088153 -1.02470580167082 -0.522940888712441 -0.863171185425945 -1.04215728919298 -0.864466507337306
-0.1625189,-1.57881887548545 -2.1887840293994 1.36116336875686 -1.02470580167082 -0.522940888712441 -0.863171185425945 0.342627053981254 -0.155348103855541
-0.1625189,-2.16691708463163 -0.807993896938655 -0.787896192088153 -1.02470580167082 -0.522940888712441 -0.863171185425945 -1.04215728919298 -0.864466507337306
0.3715636,-0.507874475300631 -0.458834049396776 -0.250631301876899 -1.02470580167082 -0.522940888712441 -0.863171185425945 -1.04215728919298 -0.864466507337306
0.7654678,-2.03612849966376 -0.933954647105133 -1.86242597251066 -1.02470580167082 -0.522940888712441 -0.863171185425945 -1.04215728919298 -0.864466507337306
0.8544153,-0.557312518810673 -0.208756571683607 -0.787896192088153 0.990146852537193 -0.522940888712441 -0.863171185425945 -1.04215728919298 -0.864466507337306
1.2669476,-0.929360463147704 -0.0578991819441687 0.152317365781542 -1.02470580167082 -0.522940888712441 -0.863171185425945 -1.04215728919298 -0.864466507337306
1.2669476,-2.28833047634983 -0.0706369432557794 -0.116315079324086 0.80409888772376 -0.522940888712441 -0.863171185425945 -1.04215728919298 -0.864466507337306
1.2669476,0.223498042876113 -1.41471935455355 -0.116315079324086 -1.02470580167082 -0.522940888712441 -0.29928234305568 0.342627053981254 0.199211097885341
1.3480731,0.107785900236813 -1.47221551299731 0.420949810887169 -1.02470580167082 -0.522940888712441 -0.863171185425945 0.342627053981254 -0.687186906466865
1.446919,0.162180092313795 -1.32557369901905 0.286633588334355 -1.02470580167082 -0.522940888712441 -0.863171185425945 -1.04215728919298 -0.864466507337306
1.4701758,-1.49795329918548 -0.263601072284232 0.823898478545609 0.788388310173035 -0.522940888712441 -0.29928234305568 0.342627053981254 0.199211097885341
1.4929041,0.796247055396743 0.0476559407005752 0.286633588334355 -1.02470580167082 -0.522940888712441 0.394013435896129 -1.04215728919298 -0.864466507337306
1.5581446,-1.62233848461465 -0.843294091975396 -3.07127197548598 -1.02470580167082 -0.522940888712441 -0.863171185425945 -1.04215728919298 -0.864466507337306
1.5993876,-0.990720665490831 0.458513517212311 0.823898478545609 1.07379746308195 -0.522940888712441 -0.863171185425945 -1.04215728919298 -0.864466507337306
1.6389967,-0.171901281967138 -0.489197399065355 -0.65357996953534 -1.02470580167082 -0.522940888712441 -0.863171185425945 -1.04215728919298 -0.864466507337306
1.6956156,-1.60758252338831 -0.590700340358265 -0.65357996953534 -0.619561070667254 -0.522940888712441 -0.863171185425945 -1.04215728919298 -0.864466507337306
1.7137979,0.366273918511144 -0.414014962912583 -0.116315079324086 0.232904453212813 -0.522940888712441 0.971228997418125 0.342627053981254 1.26288870310799
1.8000583,-0.710307384579833 0.211731938156277 0.152317365781542 -1.02470580167082 -0.522940888712441 -0.442797990776478 0.342627053981254 1.61744790484887
1.8484548,-0.262791728113881 -1.16708345615721 0.420949810887169 0.0846342590816532 -0.522940888712441 0.163172393491611 0.342627053981254 1.97200710658975
1.8946169,0.899043117369237 -0.590700340358265 0.152317365781542 -1.02470580167082 -0.522940888712441 1.28643254437683 -1.04215728919298 -0.864466507337306
1.9242487,-0.903451690500615 1.07659722048274 0.152317365781542 1.28380453408541 -0.522940888712441 -0.442797990776478 -1.04215728919298 -0.864466507337306
2.008214,-0.0633337899773081 -1.38088970920094 0.958214701098423 0.80409888772376 -0.522940888712441 -0.863171185425945 -1.04215728919298 -0.864466507337306
2.0476928,-1.15393789990757 -0.961853075398404 -0.116315079324086 -1.02470580167082 -0.522940888712441 -0.442797990776478 -1.04215728919298 -0.864466507337306
```

所使用的数据为mllib中的示例数据，每行为一条数据

任务是根据以往的数据来拟合出未来的数据

模型为线性回归模型

2.在交互式命令行中输入程序

导入spark的机器学习库Mllib

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

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

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

LabeledPoint将数据划分成feature和label

读取数据集并预处理

```
scala> val data = sc.textFile("file:///usr/local/spark/data/mllib/ridge-data/lpsa.data")
data: org.apache.spark.rdd.RDD[String] = file:///usr/local/spark/data/mllib/ridge-data/lpsa.data MapPartitionsRDD[1424] at textFile at <console>:57

scala> val parsedData = data.map { line =>{
     |       val parts = line.split(',')
     |       LabeledPoint(parts(0).toDouble, Vectors.dense(parts(1).split(' ').map(x => x.toDouble)))
     |       }
     | }
parsedData: org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.LabeledPoint] = MapPartitionsRDD[1425] at map at <console>:58
```

数据集中的第一个数字为预测结果(label)，其他的为预测输入(feature)
处理后输入到LabeledPoint(label, feature)中

设定超参数并训练

```
scala> val numIterations = 20
numIterations: Int = 20

scala> val model = LinearRegressionWithSGD.train(parsedData, numIterations)
warning: there was one deprecation warning; re-run with -deprecation for details
20/11/22 13:35:03 WARN regression.LinearRegressionWithSGD: The input data is not directly cached, which may hurt performance if
20/11/22 13:35:03 WARN regression.LinearRegressionWithSGD: The input data was not directly cached, which may hurt performance if
model: org.apache.spark.mllib.regression.LinearRegressionModel = org.apache.spark.mllib.regression.LinearRegressionModel: interc
```

此处设定迭代次数为20次

查看模型的训练结果

```
scala> val valuesAndPreds = parsedData.map { point =>
     |        val prediction = model.predict(point.features)
     |        (point.label, prediction)
     | }
valuesAndPreds: org.apache.spark.rdd.RDD[(Double, Double)] = MapPartitionsRDD[1468] at map at <console>:60

scala> val MSE = valuesAndPreds.map{ case(v, p) => math.pow((v - p), 2)}.reduce(_ + _)/valuesAndPreds.count
MSE: Double = 6.225349396263681
```

此处的训练结果用MSE来表示，MSE的公式为：

$$MSE = \frac{1}{M}\sum_{m=1}^{M}(y_m - \hat{y}_m)^2$$

可以表明预测值与真实值之间的平均距离（的平方）

利用Spark中的机器学习库Mllib对数据进行分类


```
1 0 2.52078447201548 0 0 0 2.004684436494304 2.000347299268466 0 2.228387042742021 2.228387042742023 0 0 0 0 0 0
0 2.857738033247042 0 0 2.619965104088255 0 2.004684436494304 2.000347299268466 0 2.228387042742021 2.228387042742023 0 0 0 0 0 0
0 2.857738033247042 0 2.061393766919624 0 0 2.004684436494304 0 0 2.228387042742021 2.228387042742023 0 0 0 0 0 0
1 0 0 2.061393766919624 2.619965104088255 0 2.004684436494304 2.000347299268466 0 0 0 0 2.055002875864414 0 0 0 0
1 2.857738033247042 0 2.061393766919624 2.619965104088255 0 2.004684436494304 0 0 0 0 0 2.055002875864414 0 0 0 0
0 2.857738033247042 0 2.061393766919624 2.619965104088255 0 2.004684436494304 2.000347299268466 0 2.228387042742021 2.228387042742023 0 0 0 0 0 0
1 0 0 0 2.619965104088255 0 2.004684436494304 0 0 2.228387042742021 2.228387042742023 0 2.055002875864414 0 0 0 0
1 0 0 0 2.619965104088255 0 2.004684436494304 0 0 2.228387042742021 2.228387042742023 0 2.055002875864414 0 0 0 0
0 2.857738033247042 0 2.061393766919624 2.619965104088255 0 2.004684436494304 2.000347299268466 2.122974378789621 2.228387042742021 2.228387042742023 0 0 0 0 12.72816758217773 0
0 2.857738033247042 0 0 2.619965104088255 0 0 0 0 2.228387042742021 2.228387042742023 0 2.055002875864414 0 0 0 0
1 2.857738033247042 0 0 2.619965104088255 0 0 2.000347299268466 0 2.228387042742021 2.228387042742023 0 0 0 0 0 0
1 2.857738033247042 0 0 2.619965104088255 0 2.004684436494304 2.000347299268466 2.122974378789621 0 0 0 0 0 0 0 0
1 0 0 0 0 4.745052855503306 2.004684436494304 0 2.122974378789621 2.228387042742021 2.228387042742023 0 0 0 0 0 0
1 2.857738033247042 0 0 2.619965104088255 0 2.004684436494304 0 2.122974378789621 2.228387042742021 2.228387042742023 0 2.055002875864414 0 0 0 0
0 2.857738033247042 0 0 2.619965104088255 0 0 0 0 2.228387042742021 2.228387042742023 0 2.055002875864414 0 0 0 0
0 0 0 2.061393766919624 2.619965104088255 0 0 0 2.122974378789621 2.228387042742021 2.228387042742023 0 0 0 0 0 0
0 2.857738033247042 0 0 0 0 2.004684436494304 0 2.122974378789621 2.228387042742021 2.228387042742023 0 0 0 0 0 0
0 2.857738033247042 0 0 2.619965104088255 0 0 2.000347299268466 0 2.228387042742021 2.228387042742023 0 2.055002875864414 0 0 0 0
0 2.857738033247042 0 0 2.619965104088255 0 2.004684436494304 0 0 2.228387042742021 2.228387042742023 0 2.055002875864414 0 0 0 0
1 2.857738033247042 0 0 2.619965104088255 0 0 2.000347299268466 0 2.228387042742021 2.228387042742023 0 0 0 0 0 0
```


所使用的数据为mllib中的示例数据，每行为一条数据，第一个数字0/1为标签，剩下的是特征

任务是根据特征来给数据打上0/1的标签

模型为支持向量机SVM

```
import org.apache.spark.mllib.classification.SVMWithSGD
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.linalg.Vectors
//数据加载和预处理
val data = sc.textFile(__1__)
val parsedData = data.map { line => {
   val parts = line.split(' ')
   LabeledPoint(parts(__2__).toDouble, Vectors.dense(parts.tail.map(x =>x.toDouble)))
   }
}
//输出数据
__3__
//训练模型
val model = __4__
//将数据输出到模型中分类
val labelAndPreds = parsedData.map { point =>{
     val prediction = model.predict(point.features)
     (point.label, prediction)
     }
}
//输出分类错误的样本比例
val trainErr = labelAndPreds.filter(r => r._1 != r._2).count.toDouble / parsedData.count
println("Training Error = " + trainErr)
```

__1__: 读取文件，文件位置为usr/local/spark/data/mllib/sample_svm_data.txt

__2__: 括号中的数字为下标

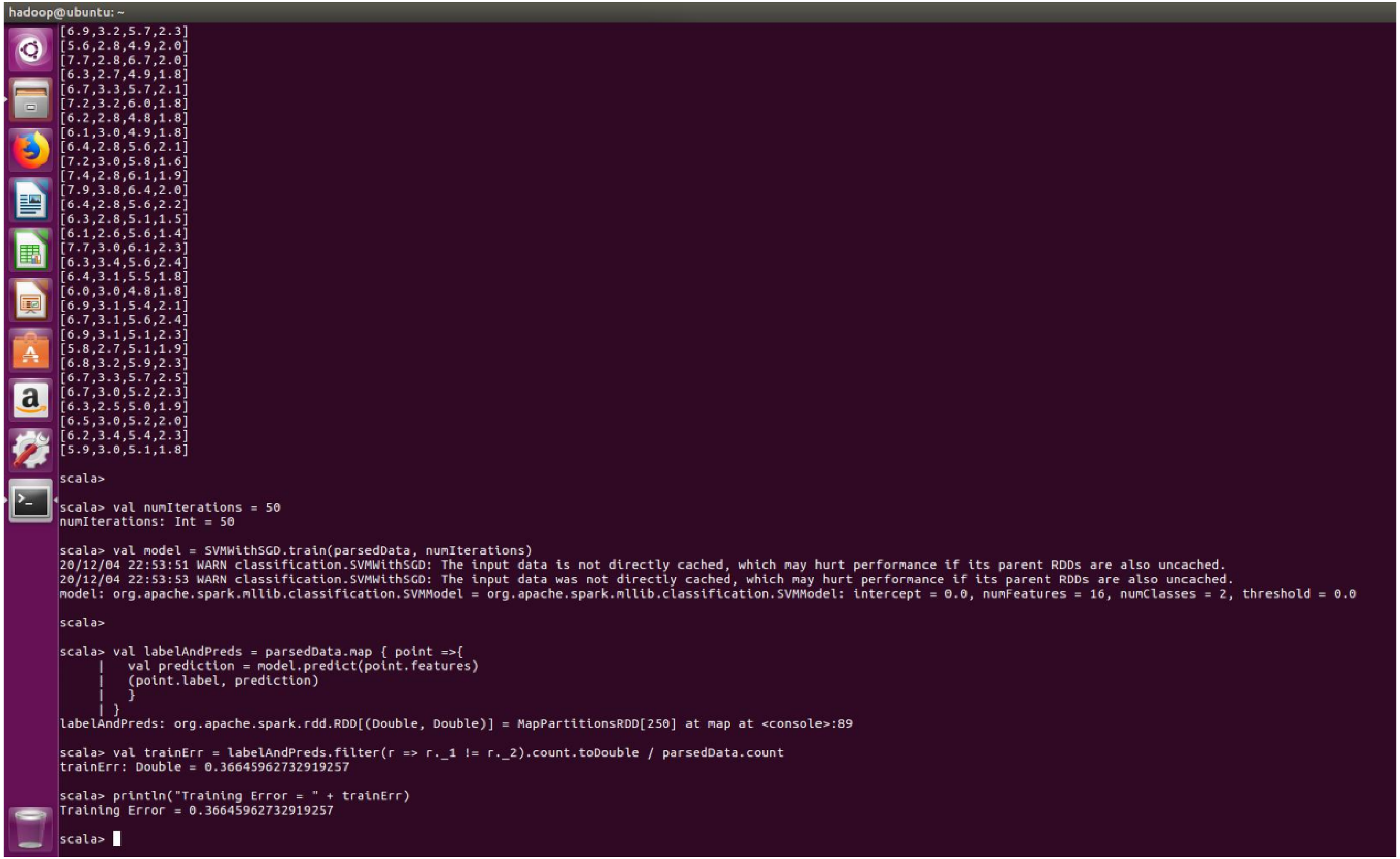
__3__: 写一条语句输出你处理好的数据

__4__: 写一条语句训练SVM模型（格式类似于前两个），SVM模型的训练函数接受的参数为(数据，迭代次数)

```
import org.apache.spark.mllib.classification.SVMWithSGD
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.linalg.Vectors
val data = sc.textFile("file:///usr/local/spark/data/mllib/sample_svm_data.txt")
val parsedData = data.map { line => {
  val parts = line.split(' ')

  LabeledPoint(parts(0).toDouble, Vectors.dense(parts.tail.map(x =>x.toDouble)))
  }
}

trainingData.collect().foreach {println}

val numIterations = 50
val model = SVMWithSGD.train(parsedData, numIterations)

val labelAndPreds = parsedData.map { point =>{
  val prediction = model.predict(point.features)
  (point.label, prediction)
  }
}
val trainErr = labelAndPreds.filter(r => r._1 != r._2).count.toDouble / parsedData.count
println("Training Error = " + trainErr)
```

1. 改写第一个程序（聚类），令簇的数目为3，迭代次数为10，然后给出点(6.3, 2.8, 5.1, 2.0)和点(6.8, 3.3, 5.0, 1.6)的聚类结果。

2. 补全第三个程序（分类），并给出分类错误的样本比例。


hadoop@ubuntu: ~
[6.9,3.2,5.7,2.3]
[5.6,2.8,4.9,2.0]
[7.7,2.8,6.7,2.0]
[6.3,2.7,4.9,1.8]
[6.7,3.3,5.7,2.1]
[7.2,3.2,6.0,1.8]
[6.2,2.8,4.8,1.8]
[6.1,3.0,4.9,1.8]
[6.4,2.8,5.6,2.1]
[7.2,3.0,5.8,1.6]
[7.4,2.8,6.1,1.9]
[7.9,3.8,6.4,2.0]
[6.4,2.8,5.6,2.2]
[6.3,2.8,5.1,1.5]
[6.1,2.6,5.6,1.4]
[7.7,3.0,6.1,2.3]
[6.3,3.4,5.6,2.4]
[6.4,3.1,5.5,1.8]
[6.0,3.0,4.8,1.8]
[6.9,3.1,5.4,2.1]
[6.7,3.1,5.6,2.4]
[6.9,3.1,5.1,2.3]
[5.8,2.7,5.1,1.9]
[6.8,3.2,5.9,2.3]
[6.7,3.3,5.7,2.5]
[6.7,3.0,5.2,2.3]
[6.3,2.5,5.0,1.9]
[6.5,3.0,5.2,2.0]
[6.2,3.4,5.4,2.3]
[5.9,3.0,5.1,1.8]
scala>
scala> val numIterations = 50
numIterations: Int = 50
scala> val model = SVMWithSGD.train(parsedData, numIterations)
20/12/04 22:53:51 WARN classification.SVMWithSGD: The input data is not directly cached, which may hurt performance if its parent RDDs are also uncached.
20/12/04 22:53:53 WARN classification.SVMWithSGD: The input data was not directly cached, which may hurt performance if its parent RDDs are also uncached.
model: org.apache.spark.mllib.classification.SVMModel = org.apache.spark.mllib.classification.SVMModel: intercept = 0.0, numFeatures = 16, numClasses = 2, threshold = 0.0
scala>
scala> val labelAndPreds = parsedData.map { point =>{
| val prediction = model.predict(point.features)
| (point.label, prediction)
| }
| }
labelAndPreds: org.apache.spark.rdd.RDD[(Double, Double)] = MapPartitionsRDD[250] at map at <console>:89
scala> val trainErr = labelAndPreds.filter(r => r._1 != r._2).count.toDouble / parsedData.count
trainErr: Double = 0.36645962732919257
scala> println("Training Error = " + trainErr)
Training Error = 0.36645962732919257
scala>

hadoop@ubuntu: ~

```
[7.6,3.0,6.6,2.1] belongs to cluster 0
[4.9,2.5,4.5,1.7] belongs to cluster 2
[7.3,2.9,6.3,1.8] belongs to cluster 0
[6.7,2.5,5.8,1.8] belongs to cluster 0
[7.2,3.6,6.1,2.5] belongs to cluster 0
[6.5,3.2,5.1,2.0] belongs to cluster 0
[6.4,2.7,5.3,1.9] belongs to cluster 0
[6.8,3.0,5.5,2.1] belongs to cluster 0
[5.7,2.5,5.0,2.0] belongs to cluster 2
[5.8,2.8,5.1,2.4] belongs to cluster 2
[6.4,3.2,5.3,2.3] belongs to cluster 0
[6.5,3.0,5.5,1.8] belongs to cluster 0
[7.7,3.8,6.7,2.2] belongs to cluster 0
[7.7,2.6,6.9,2.3] belongs to cluster 0
[6.0,2.2,5.0,1.5] belongs to cluster 2
[6.9,3.2,5.7,2.3] belongs to cluster 0
[5.6,2.8,4.9,2.0] belongs to cluster 2
[7.7,2.8,6.7,2.0] belongs to cluster 0
[6.3,2.7,4.9,1.8] belongs to cluster 2
[6.7,3.3,5.7,2.1] belongs to cluster 0
[7.2,3.2,6.0,1.8] belongs to cluster 0
[6.2,2.8,4.8,1.8] belongs to cluster 2
[6.1,3.0,4.9,1.8] belongs to cluster 2
[6.4,2.8,5.6,2.1] belongs to cluster 0
[7.2,3.0,5.8,1.6] belongs to cluster 0
[7.4,2.8,6.1,1.9] belongs to cluster 0
[7.9,3.8,6.4,2.0] belongs to cluster 0
[6.4,2.8,5.6,2.2] belongs to cluster 0
[6.3,2.8,5.1,1.5] belongs to cluster 2
[6.1,2.6,5.6,1.4] belongs to cluster 0
[7.7,3.0,6.1,2.3] belongs to cluster 0
[6.3,3.4,5.6,2.4] belongs to cluster 0
[6.4,3.1,5.5,1.8] belongs to cluster 0
[6.0,3.0,4.8,1.8] belongs to cluster 2
[6.9,3.1,5.4,2.1] belongs to cluster 0
[6.7,3.1,5.6,2.4] belongs to cluster 0
[6.9,3.1,5.1,2.3] belongs to cluster 0
[5.8,2.7,5.1,1.9] belongs to cluster 2
[6.8,3.2,5.9,2.3] belongs to cluster 0
[6.7,3.3,5.7,2.5] belongs to cluster 0
[6.7,3.0,5.2,2.3] belongs to cluster 0
[6.3,2.5,5.0,1.9] belongs to cluster 2
[6.5,3.0,5.2,2.0] belongs to cluster 0
[6.2,3.4,5.4,2.3] belongs to cluster 0
[5.9,3.0,5.1,1.8] belongs to cluster 2

scala> println("Vectors 6.3,2.8,5.1,2.0 is belongs to clusters:" + model.predict(Vectors.dense("6.3,2.8,5.1,2.0".split(',').map(_.toDouble))))
Vectors 6.3,2.8,5.1,2.0 is belongs to clusters:0

scala>

scala> println("Vectors 6.8,3.3,5.0,1.6 is belongs to clusters:" + model.predict(Vectors.dense("6.8,3.3,5.0,1.6".split(',').map(_.toDouble))))
Vectors 6.8,3.3,5.0,1.6 is belongs to clusters:0

scala>

scala> 
```