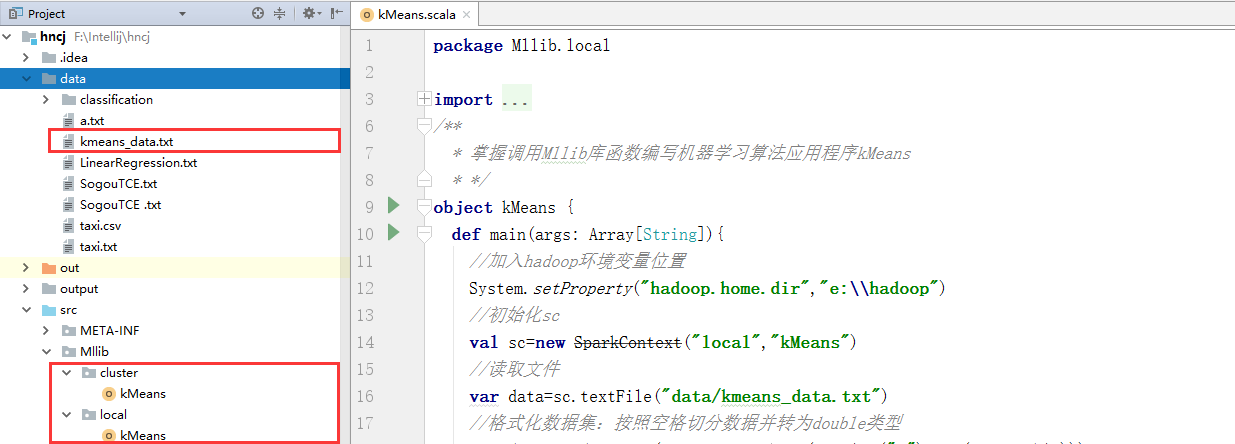
**大数据平台部分实践作业10#**

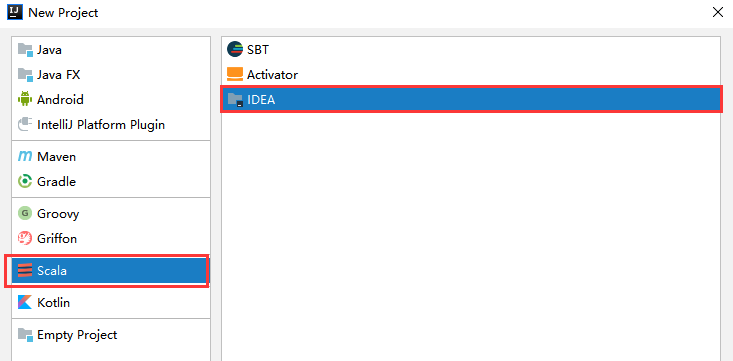
# 调用Mllib库函数

## 1.1、kMeans

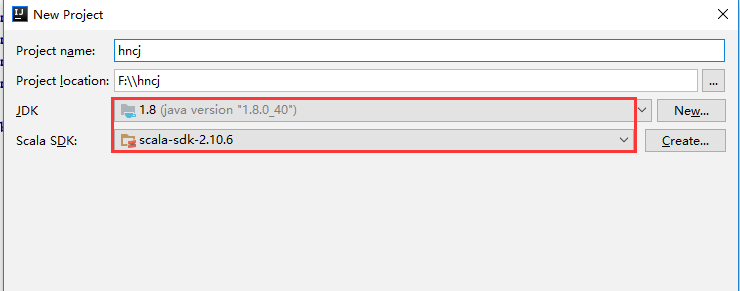
1. 、工程结构图如下所示：



（2）、新建工程/file/new/project

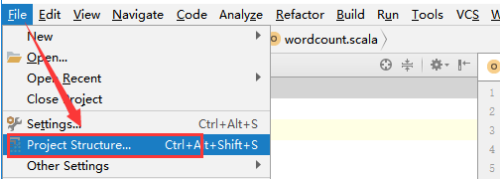


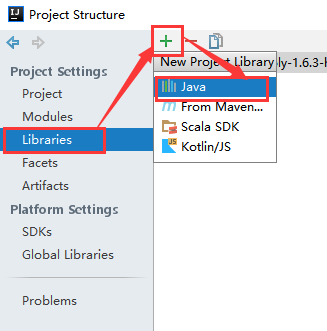
（3）、填写工程相应的名字、位置、导入相应JDK和Scala环境变量位置

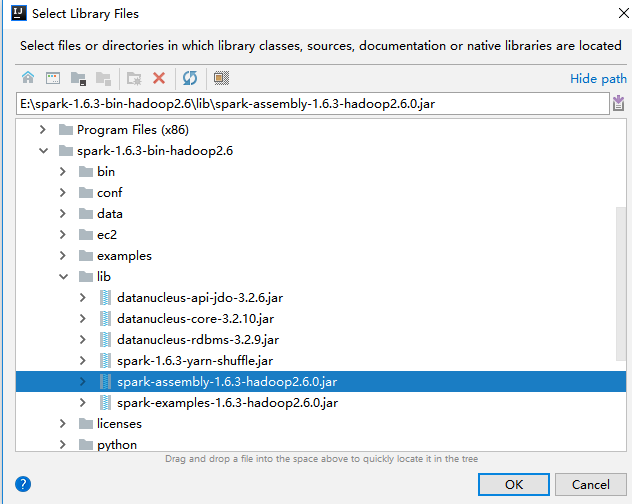


（4）、导入spark的jar包

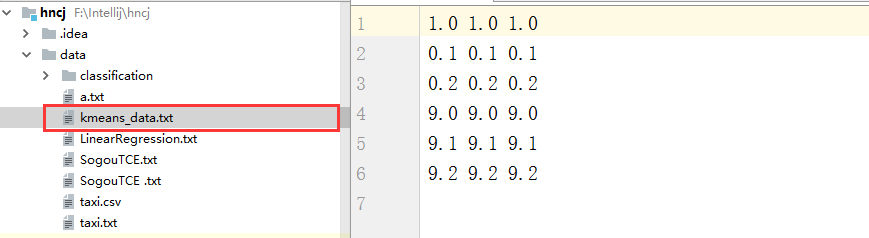
将spark解压包下lib/spark-assembly-1.6.3-hadoop2.6.0.jar导入中







1. 、在hncj/data下目录下放入数据



### 1.1.1、local模式

（1）、读取文件

*//初始化sc* **val** sc=**new** SparkContext(**"local"**,**"kMeans"**)  
 *//读取文件* **var** data=sc.textFile(**"data/kmeans\_data.txt"**)  
 *//格式化数据集：按照空格切分数据并转为double类型* **var** dataset=data.map(x=>Vectors.*dense*(x.split(**" "**).map(\_.toDouble)))

1. 、建立模型  
    *//最大迭代次数* **var** maxIteration=20  
    *//划分最大簇数* **var** maxClass=5  
    *//构建训练模型* **val** model=KMeans.*train*(dataset,maxClass,maxIteration)
2. 、模型评估与预测  
    *//使用误差平方之和估计数据模型* **val** cost=model.computeCost(dataset)  
    *println*(**"Sum of squares of errors="**+cost)  
    *//使用模型测试单点数据  
    println*(**"data 0.2 0.2 0.2 is belongs to cluster:"**+model.predict(Vectors.*dense*(**"0.2 0.2 0.2"**.split(**" "**).map(\_.toDouble))))  
    *println*(**"data 9 9 9 is belongs to cluster:"**+model.predict(Vectors.*dense*(**"9 9 9"**.split(**" "**).map(\_.toDouble))))  
   （4）、结果输出

*//将数据集和结果 打印output/kMeans2* **val** result2=data.map{  
 line=>  
 **val** dataset1=Vectors.*dense*(line.split(**" "**).map(\_.toDouble))  
 **val** predictresult=model.predict(dataset1)  
 line+**" "**+predictresult  
 }.saveAsTextFile(**"output/kMeans2"**)

### 1.1.2、clustet/client模式

clustet/client模式代码和local模式代码相似，只是更改文件读取形式。

（1）、读取文件

*//当文件不存在时候  
if (args.length < 1) {  
 System.err.println("Usage: <file>")  
 System.exit(1)  
}  
val conf=new SparkConf()  
conf.setAppName("kMeans")  
//初始化sc  
val sc=new SparkContext(conf)  
//读取文件  
var data=sc.textFile(args(0))  
//格式化数据集：按照空格切分数据并转为double类型  
var dataset=data.map(x=>Vectors.dense(x.split(" ").map(\_.toDouble)))*

**var** dataset=data.map(x=>Vectors.*dense*(x.split(**" "**).map(\_.toDouble)))

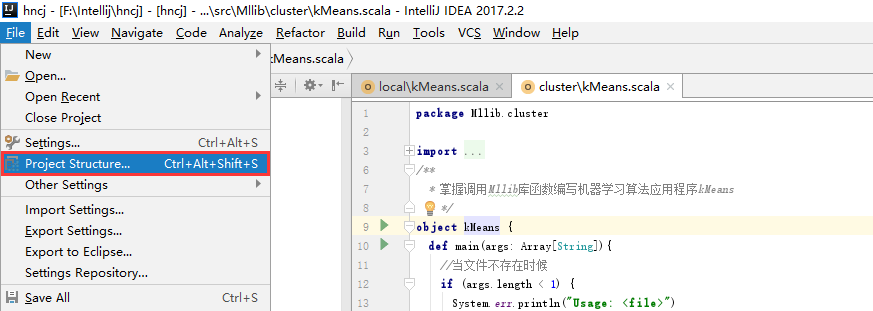
（2）、建立模型  
 *//最大迭代次数* **var** maxIteration=20  
 *//划分最大簇数* **var** maxClass=5  
 *//构建训练模型* **val** model=KMeans.*train*(dataset,maxClass,maxIteration)

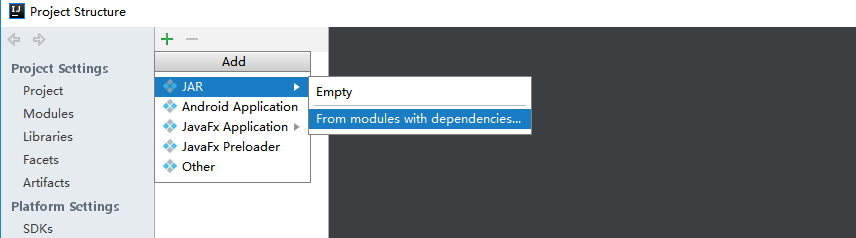
（3）、模型评估与预测  
 *//使用误差平方之和估计数据模型* **val** cost=model.computeCost(dataset)  
 *println*(**"Sum of squares of errors="**+cost)  
 *//使用模型测试单点数据  
 println*(**"data 0.2 0.2 0.2 is belongs to cluster:"**+model.predict(Vectors.*dense*(**"0.2 0.2 0.2"**.split(**" "**).map(\_.toDouble))))  
 *println*(**"data 9 9 9 is belongs to cluster:"**+model.predict(Vectors.*dense*(**"9 9 9"**.split(**" "**).map(\_.toDouble))))  
（4）、结果输出

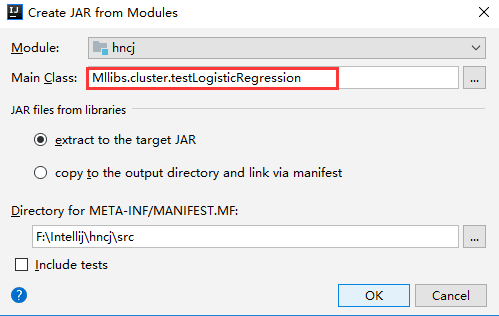
*//将数据集和结果 打印output/kMeans2* **val** result2=data.map{  
 line=>  
 **val** dataset1=Vectors.*dense*(line.split(**" "**).map(\_.toDouble))  
 **val** predictresult=model.predict(dataset1)  
 line+**" "**+predictresult  
 }.saveAsTextFile(**"output/kMeans2"**)

### 1.1.3、导出jar

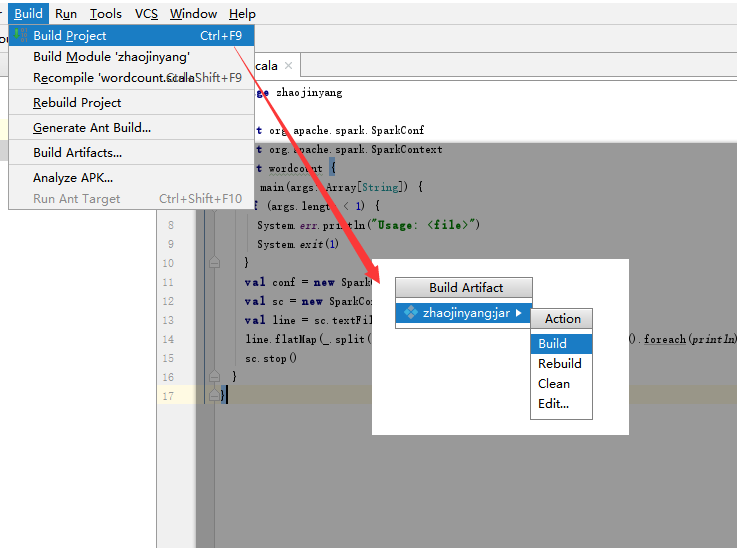
（1）、选择File -> Project Structure -> Artifact, 选择‘+’----->Jar---->From Modeles with dependencies ,选择main函数，之后要指定下输出的位置。







（2）选择Build -> Build Artifacts -> jar包名 -> Build，直到编译器左下角出现completed successfully。



（3）将导入的cqupt.jar上传到hadoop的/usr/local目录下。

### 1.1.3、本地测试

1. 、在intelliJ中运行Mllib.local.kMeans，我们在控制台观察结果运行结果：

输出的簇中心为：

Output cluster center：

[9.0,9.0,9.0]

[0.05,0.05,0.05]

[0.2,0.2,0.2]

[9.2,9.2,9.2]

[9.1,9.1,9.1]

平方误差和为：

Sum of squares of errors=0.01500000000000001

单个模型预测：

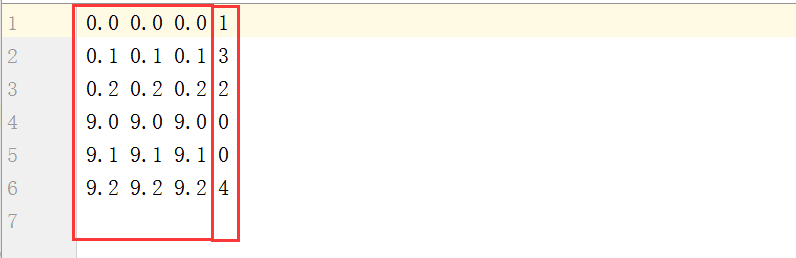
data 0.2 0.2 0.2 is belongs to cluster:2

data 9 9 9 is belongs to cluster:0

（2）、我们在intelliJ左侧看到生成的kMeans文件中，打开我们发现结果如下所示。

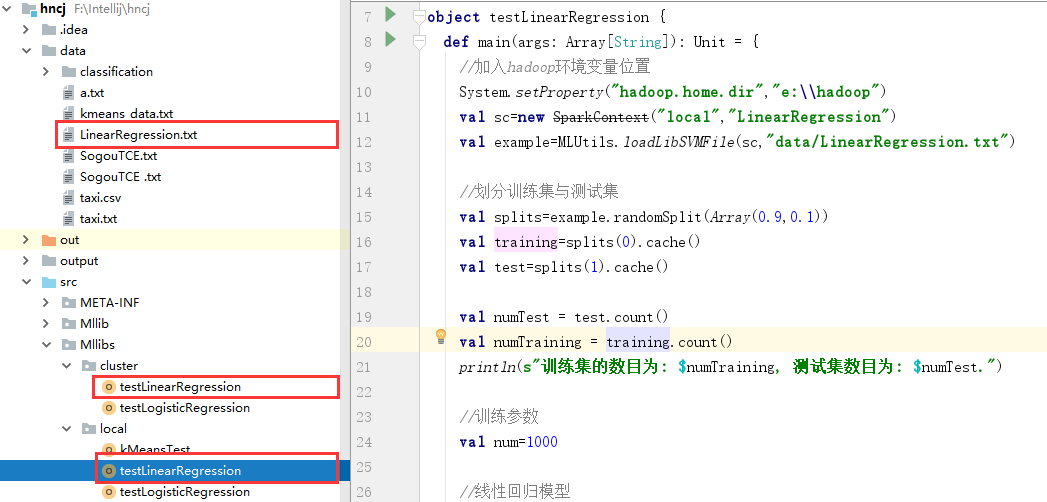


前三列为原始数据，后一列为分的簇

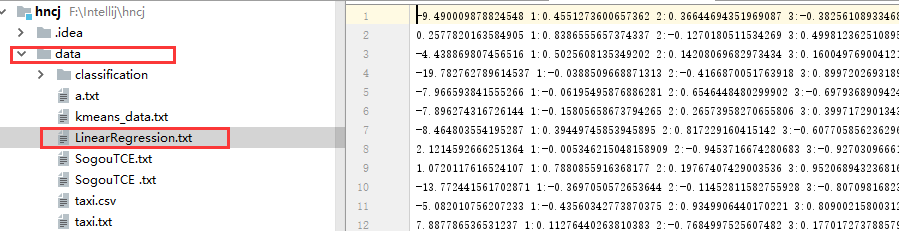


## 1.2、LinearRegression

（1）、工程结构图如下所示：



（2）、在hncj/data下目录下放入数据LinearRegression.txt



### 1.2.1、local模式

（1）、读取文件

*//加入hadoop环境变量位置*System.*setProperty*(**"hadoop.home.dir"**,**"e:\\hadoop"**)  
**val** sc=**new** SparkContext(**"local"**,**"LinearRegression"**)  
**val** example=MLUtils.*loadLibSVMFile*(sc,**"data/LinearRegression.txt"**)

**var** dataset=data.map(x=>Vectors.*dense*(x.split(**" "**).map(\_.toDouble)))

（2）、数据划分

*//划分训练集与测试集***val** splits=example.randomSplit(*Array*(0.9,0.1))  
**val** training=splits(0).cache()  
**val** test=splits(1).cache()  
  
**val** numTest = test.count()  
**val** numTraining = training.count()  
*println*(**s"训练集的数目为: $**numTraining**, 测试集数目为: $**numTest**."**)

（3）、建立模型  
 *//训练参数***val** num=1000  
*//线性回归模型***val** model=LinearRegressionWithSGD.*train*(training,num)  
*println*(**s"Intercept: $**{model.intercept} **Coefficients: $**{model.weights}**"**)

（4）、模型评估与预测  
 *//测试集进行测试***val** prediction = model.predict(test.map(\_.features))  
**val** predictionAndLabel = prediction.zip(test.map(\_.label))  
predictionAndLabel.saveAsTextFile(**"output/LinearRegression/pred"**)  
predictionAndLabel.foreach(*println*(\_))  
*//均方根误差，模型误差评估***val** loss = predictionAndLabel.map {  
  
 **case** (p1, p2) =>  
 **val** err = p1-p2  
 err \* err  
}.reduce(\_ + \_)  
**val** rmse = math.*sqrt*(loss / numTest)  
*println*(**s"均方根误差为 = $**rmse**."**)

### 1.2.2、clustet/client模式

clustet/client模式代码和local模式代码相似，只是更改文件读取形式。

（1）、读取文件

*//当文件不存在时候***if** (args.length < 1) {  
 System.*err*.println(**"Usage: <file>"**)  
 System.*exit*(1)  
}  
**val** conf=**new** SparkConf().setAppName(**"LinearRegression"**)  
**val** sc=**new** SparkContext(conf)  
**val** example=MLUtils.*loadLibSVMFile*(sc,args(0))

（2）、数据划分

*//划分训练集与测试集***val** splits=example.randomSplit(*Array*(0.9,0.1))  
**val** training=splits(0).cache()  
**val** test=splits(1).cache()  
  
**val** numTest = test.count()  
**val** numTraining = training.count()  
*println*(**s"训练集的数目为: $**numTraining**, 测试集数目为: $**numTest**."**)

（3）、建立模型  
 *//训练参数***val** num=1000  
*//线性回归模型***val** model=LinearRegressionWithSGD.*train*(training,num)  
*println*(**s"Intercept: $**{model.intercept} **Coefficients: $**{model.weights}**"**)

（4）、模型评估与预测  
 *//测试集进行测试***val** prediction = model.predict(test.map(\_.features))  
**val** predictionAndLabel = prediction.zip(test.map(\_.label))  
predictionAndLabel.saveAsTextFile(**"output/LinearRegression/pred"**)  
predictionAndLabel.foreach(*println*(\_))  
*//均方根误差，模型误差评估***val** loss = predictionAndLabel.map {  
  
 **case** (p1, p2) =>  
 **val** err = p1-p2  
 err \* err  
}.reduce(\_ + \_)  
**val** rmse = math.*sqrt*(loss / numTest)  
*println*(**s"均方根误差为 = $**rmse**."**)

（5）、将jar包导入到hdfs相应位置

### 1.2.3、本地测试

（1）、在intelliJ中运行Mllibs.local.testLinearRegression，我们在控制台观察结果运行结果：

输出的回归结果为：

(-2.1773980223554745,8.417571532985823)

(0.8875924399883112,9.020048819638827)

(0.060584561967331294,11.13509519160867)

(0.0838825526408973,12.242677118499806)

**... ...**

(-0.7192726076500668,8.688243146888663)

(0.6459526485004842,2.2546997585565296)

(-2.3994427347282845,0.4250502150408626)

均方根误差：

均方根误差为 = 9.543627925514054

Intercept and Coefficients：

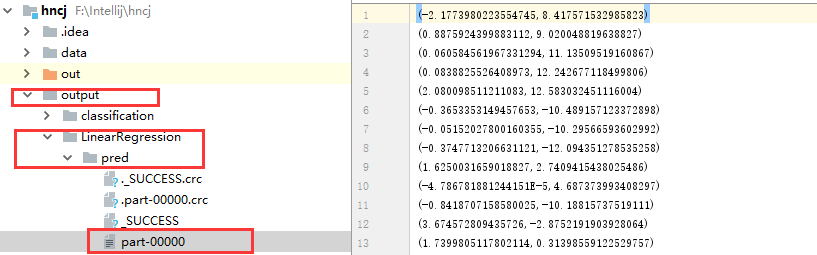
Intercept:

0.0

Coefficients: [-0.5789770669140339,0.9981525607487858,-0.9179271215541233,2.463595831169019,0.6927254988565752,1.127844706491487,-0.2775592272079169,-0.47433252285769517,-0.7208342165065849,0.2826035055542507]

（2）、我们在intelliJ左侧看到生成的outputt/testLinearRegression

文件中，打开我们发现结果如下所示。



打开outputt/testLinearRegression/part-00000的结果与控制台一致。

### 1.2.4、集群测试

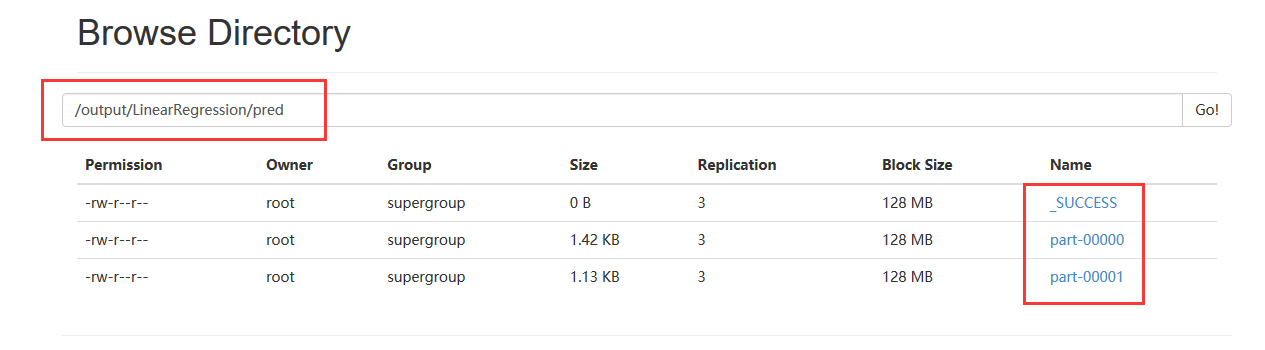
（1）、将数据集LinearRegression.txt文件上传到hdfs中根目录下。线性回归hncj.jar上传到/usr/local目录下。

（2）、执行程序

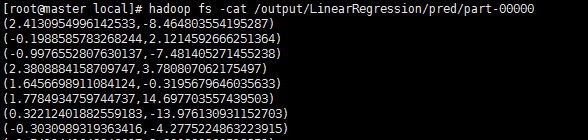
spark-submit --master spark://192.168.56.100:7077 --name LinearRegression --class Mllibs.cluster.testLinearRegression --executor-memory 500m --total-executor-cores 2 /usr/local/hncj.jar hdfs://192.168.56.100:9000/data/LinearRegression.txt

IMG_256

（3）、在hdfs上保存的结果

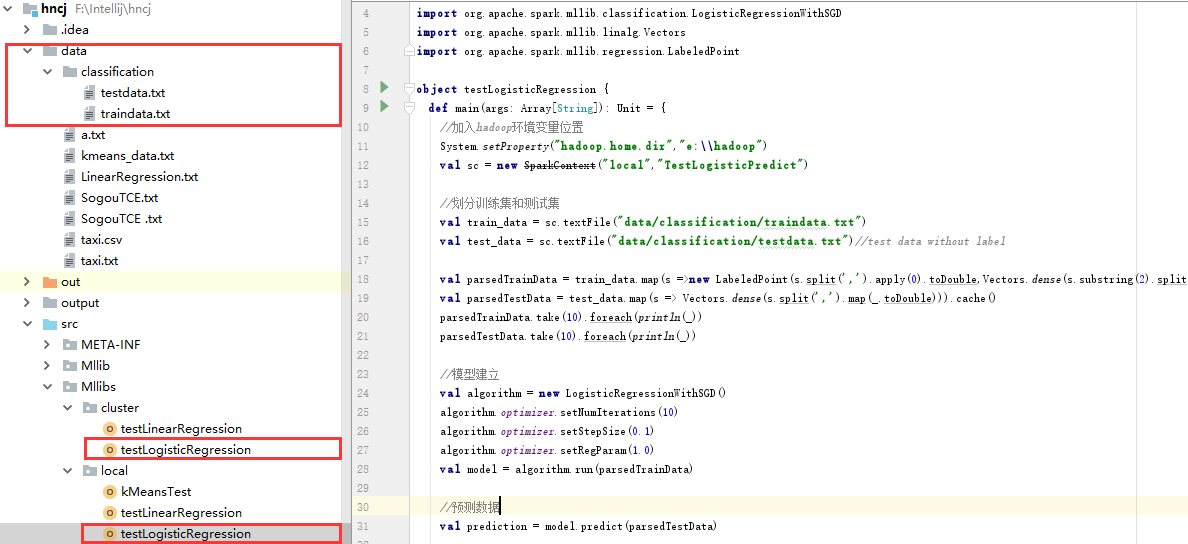


线性回归结果：



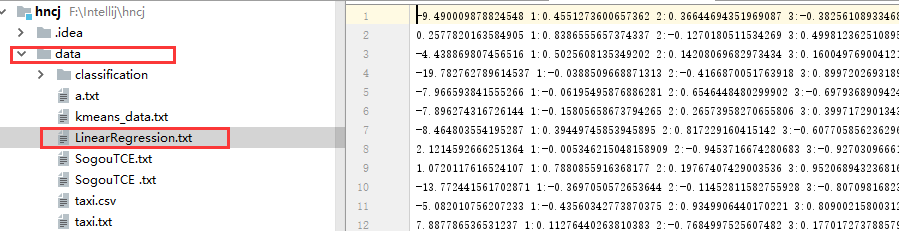
## 1.3、LinearRegression

（1）、工程结构图如下所示：



1. 、在hncj/data/classification下目录下放入数据

traindata.txt  
testdata.txt



### 1.3.1、local模式

（1）、读取文件与划分数据

*//加入hadoop环境变量位置*System.*setProperty*(**"hadoop.home.dir"**,**"e:\\hadoop"**)  
**val** sc = **new** SparkContext(**"local"**,**"TestLogisticPredict"**)  
  
*//划分训练集和测试集***val** train\_data = sc.textFile(**"data/classification/traindata.txt"**)  
**val** test\_data = sc.textFile(**"data/classification/testdata.txt"**)*//test data without label***val** parsedTrainData = train\_data.map(s =>**new** LabeledPoint(s.split(**','**).apply(0).toDouble,Vectors.*dense*(s.substring(2).split(**','**).map(\_.toDouble))) ).cache()  
**val** parsedTestData = test\_data.map(s => Vectors.*dense*(s.split(**','**).map(\_.toDouble))).cache()  
parsedTrainData.take(10).foreach(*println*(\_))  
parsedTestData.take(10).foreach(*println*(\_))

（2）、建立模型  
 *//模型建立***val** algorithm = **new** LogisticRegressionWithSGD()  
algorithm.*optimizer*.setNumIterations(10)  
algorithm.*optimizer*.setStepSize(0.1)  
algorithm.*optimizer*.setRegParam(1.0)  
**val** model = algorithm.run(parsedTrainData)

（3）、模型评估与预测  
 *//预测数据***val** prediction = model.predict(parsedTestData)  
  
prediction.saveAsTextFile(**"output/classification/pred"**)

### 1.3.2、clustet/client模式

clustet/client模式代码和local模式代码相似，只是更改文件读取形式。

（1）、读取文件与划分数据

**if** (args.length < 2) {  
 System.*err*.println(**"Usage: <trainfile>,<testfile>"**)  
 System.*exit*(1)  
}  
**val** conf = **new** SparkConf().setAppName(**"TestLogisticPredict"**)  
**val** sc = **new** SparkContext(conf)  
*//划分训练集和测试集***val** train\_data = sc.textFile(**"data/classification/traindata.txt"**)  
**val** test\_data = sc.textFile(**"data/classification/testdata.txt"**)*//test data without label***val** parsedTrainData = train\_data.map(s =>**new** LabeledPoint(s.split(**','**).apply(0).toDouble,Vectors.*dense*(s.substring(2).split(**','**).map(\_.toDouble))) ).cache()  
**val** parsedTestData = test\_data.map(s => Vectors.*dense*(s.split(**','**).map(\_.toDouble))).cache()  
parsedTrainData.take(10).foreach(*println*(\_))  
parsedTestData.take(10).foreach(*println*(\_))

（2）、建立模型  
 *//模型建立***val** algorithm = **new** LogisticRegressionWithSGD()  
algorithm.*optimizer*.setNumIterations(10)  
algorithm.*optimizer*.setStepSize(0.1)  
algorithm.*optimizer*.setRegParam(1.0)  
**val** model = algorithm.run(parsedTrainData)

（3）、模型评估与预测  
 *//预测数据***val** prediction = model.predict(parsedTestData)  
  
prediction.saveAsTextFile(**"output/classification/pred"**)

（5）、将jar包导入到hdfs相应位置

### **1.3.3、本地测试**

（1）、在intelliJ中运行Mllibs.local.testLinearRegression，我们在控制台观察结果运行结果：

输出部分测试集和训练集：

[22.0,45026.0,2046.0,7465.0,48.0,515773.0,10745.0,199070.0,43.0,44414.0,1032.0,1051.0]

[10.0,18704.0,1870.0,4977.0,30.0,96694.0,3223.0,35907.0,35.0,232537.0,6643.0,174186.0]

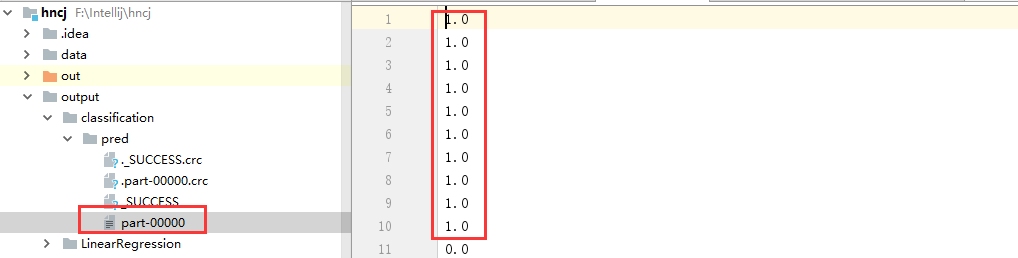
... ...

[25.0,96608.0,3864.0,24884.0,49.0,70959.0,1448.0,24884.0,47.0,393641.0,8375.0,21459.0]

[4.0,22322.0,5580.0,13188.0,11.0,21894.0,1990.0,3484.0,8.0,771236.0,96404.0,765423.0]

（2）、我们在intelliJ左侧看到生成的output/classification/pred

文件中，打开我们显示分类结果如下所示。



### **1.3.4、集群测试**

（1）、将数据集traindata.txt和testdata.txt文件上传到hdfs中/data/classification/。逻辑回归hncj.jar上传到/usr/local目录下。

（2）、从master节点进入到/usr/local/spark/sbin启动Spark

cd /usr/local/spark/sbin

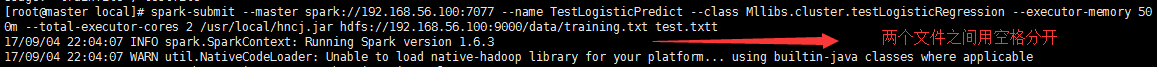
开启：

start-all.sh

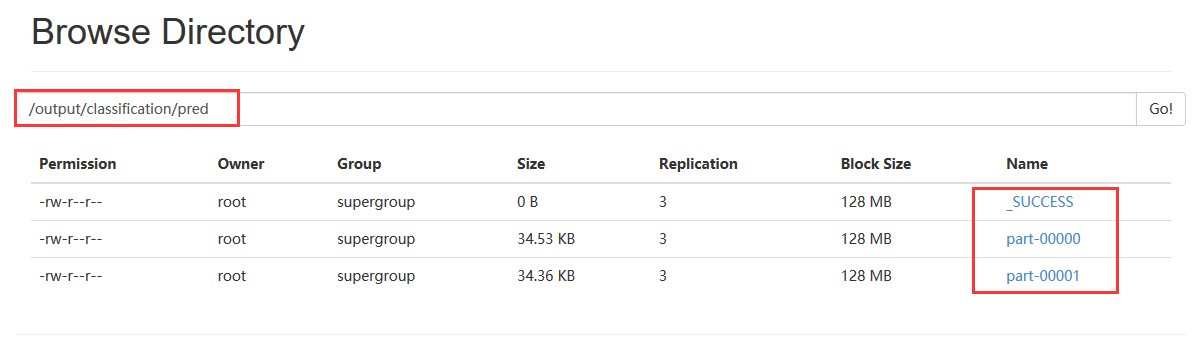
（3）、执行程序

spark-submit --master spark://192.168.56.100:7077 --name TestLogisticPredict --class Mllibs.cluster.testLogisticRegression --executor-memory 500m --total-executor-cores 2 /usr/local/hncj.jar hdfs://192.168.56.100:9000/data/training.txt test.txt

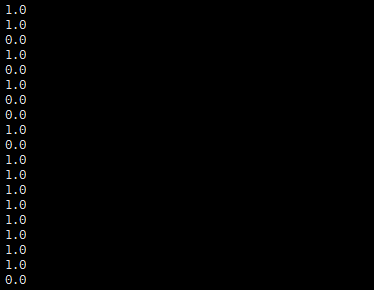
！！！注：不同文件之间用空格分离training.txt test.txt



1. 、在hdfs上保存的结果



显示逻辑线性回归结果



# 文本数据聚类案例

## 2.1、TF-IDF

某个词在一篇文章中出现次数"词频"（TF）。

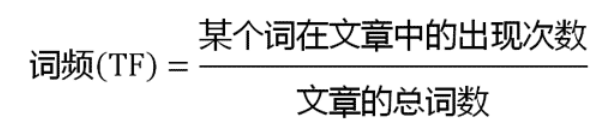
出现次数最多的词是----"的"、"是"、"在"----这一类最常用的词。它们叫做["停用词"](http://baike.baidu.com/view/3784680.htm" \t "http://www.cnblogs.com/haore147/p/_blank)（stop words），表示对找到结果毫无帮助、必须过滤掉的词。

在词频的基础上，要对每个词分配一个"重要性"权重。最常见的词给予最小的权重，较常见的词给予较小的权重，较少见的词给予较大的权重。这个权重叫做"逆文档频率"（IDF），它的大小与一个词的常见程度成反比。

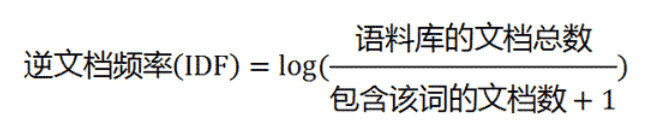
计算词频：



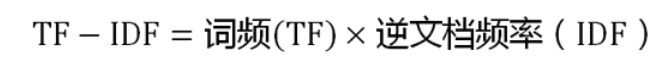
或者



计算逆文档频：



TF-IDF：



## 2.2、读取数据

数据集为搜狗实验室提供数据集，我们将网址和后边标签按照正则表达式进行分离，分离后再剔除stop words，将所有字母变为小写，字符长度要大于2。对网址进行切分，计算每个词的词频，逆文档频，最后算出TF-IDF。对每行的TF-IDF进行排序，取最大的TF-IDF，其即为关键词。



System.*setProperty*(**"hadoop.home.dir"**,**"e:\\hadoop"**)  
**val** sc = **new** SparkContext(**"local"**,**"SogoukMeans"**)  
**val** Num=**"[^0-9]\*"**.r  
**def** qiefen(text:String):Seq[String]={  
 *//按照字符和数字（W+）进行切分，所有字母变为小写，字符长度要大于2* text.split(**"\\W+"**).map(\_.toLowerCase).filter(Num.*pattern*.matcher(\_).matches) .filter(\_.length>2).toSeq  
}  
**val** rdd=sc.textFile(**"data/SogouTCE .txt"**)  
*//切分数据***val** data= rdd.map(qiefen)  
data.saveAsTextFile(**"output/sogou/data"**+System.*currentTimeMillis*)

显示切分后的结果：

WrappedArray(http, www, xinhuanet, com, auto)  
WrappedArray(http, www, xinhuanet, com, fortune)  
WrappedArray(http, www, xinhuanet, com, internet)

... ...  
WrappedArray(http, www, xinhuanet, com, health)  
WrappedArray(http, www, xinhuanet, com, sports)

## 2.3、计算TF-IDF

记录所有单词的下标映射，这里我们使用一个mapWord:Map[Int,String]来保存这层关系。

*//hashing***val** hashingTF = **new** HashingTF()  
*//每个词赋予对应hashing,记录所有单词的下标映射***val** index=data.flatMap(x=>x).map(w=>(hashingTF.indexOf(w),w)).collect.toMap  
*//计算其在文档中的词频（TF）***val** tf=hashingTF.transform(data)  
tf.saveAsTextFile(**"output/sogou/tf"**+System.*currentTimeMillis*)  
**val** word=tf.context.broadcast(index)  
tf.cache()  
*//计算TF-IDF值***val** idf = **new** IDF(2).fit(tf)  
**val** tfidf: RDD[Vector] = idf.transform(tf)  
**val** r = tfidf.map{*///* **case** *SparseVector*(size, indices, values)=>  
 **val** words=indices.map(index=>word.value.getOrElse(index,**"null"**))  
 words.zip(values).sortBy(-\_.\_2).toSeq.head  
}  
rdd.zip(r).saveAsTextFile(**"output/sogou/result"**+System.*currentTimeMillis*)

## 2.3、本地测试

对网址进行切分，计算每个词的词频，逆文档频，最后算出TF-IDF。对每行的TF-IDF进行排序，取最大的TF-IDF，其即为关键词。

**val** r = tfidf.map{  
 **case** *SparseVector*(size, indices, values)=>  
 **val** words=indices.map(index=>bcWords.value.getOrElse(index,**"null"**))  
 words.zip(values).sortBy(-\_.\_2).toSeq.head  
}  
rdd.zip(r).saveAsTextFile(**"output/sogou/result"**+System.*currentTimeMillis*)

显示运行结果：



## 2.4、集群测试

1. 、将数据集SogouTCE.txt文件上传到hdfs中根目录下。搜狗聚类hncj.jar上传到/usr/local目录下。

（2）、从master节点进入到/usr/local/spark/sbin启动Spark

cd /usr/local/spark/sbin

开启：

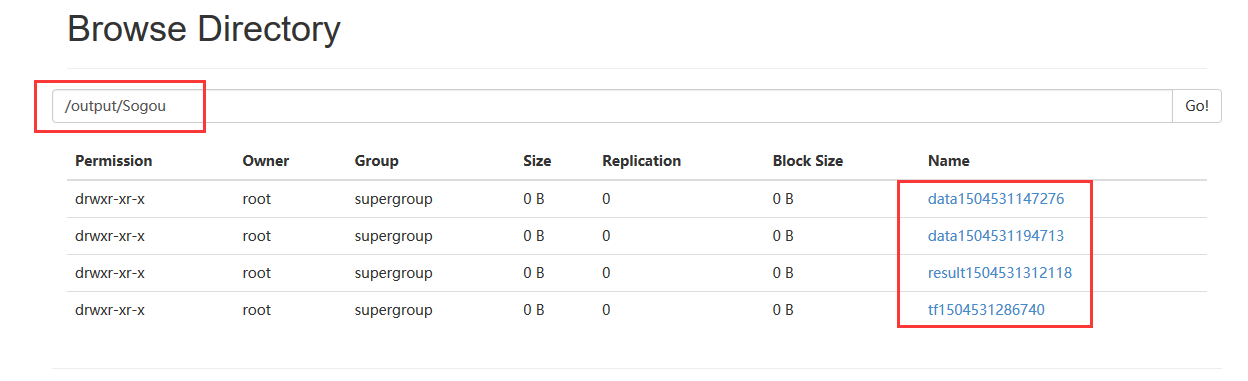
start-all.sh

（3）、执行程序

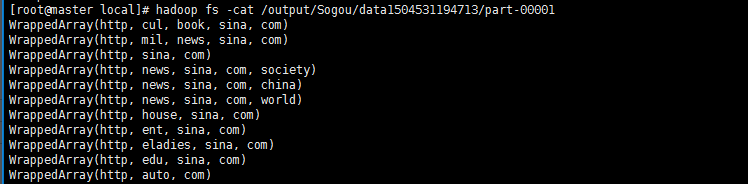
spark-submit --master spark://192.168.56.100:7077 --name SogoukMeans --class SogouTCE.cluster.SogouTFIDF --executor-memory 500m --total-executor-cores 2 /usr/local/hncj.jar hdfs://192.168.56.100:9000 /data/SogouTCE.txt

IMG_256

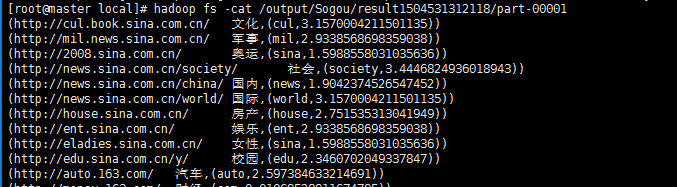
1. 、在hdfs上保存的结果



在hdfs中显示切分结果



TF\_IDF计算后的结果为：



# 出租车数据聚类

## 3.1、数据集

本次出租车数据聚类使用的数据集为taxi.csv,数据大小为24.4M,大约80万条数据，数据之间用逗号分隔。

数据集中的一条记录如下所示：

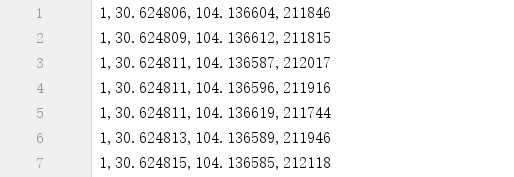
1,30.624848,104.136546,210903

TID：出租车的ID。每辆出租车的TID都是唯一的。

Lat：出租车状态为载客时的纬度。

Lon：出租车状态为载客时的经度。

Time：该条记录的时间戳。如 210903 代表 21 点 09分 03秒。



## 3.2、定义字段格式

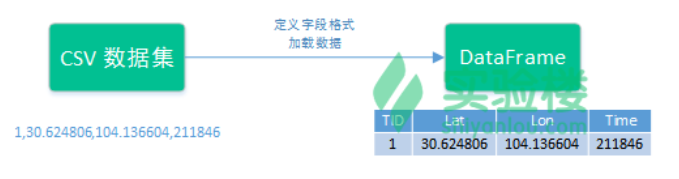
Spark SQL支持两种支持两种不同的方式来将现有的 RDD 转换为数据框（DataFrame）。第一个方法是使用反射机制（Relection），另一个则是通过编程的方式指明字段格式。我们这里通过编程方式指明字段格式。

（1）、主函数外定义RawDataRecord格式

**case class** RawDataRecord(TID:String, Lat:String,Lon:String,Time:String)

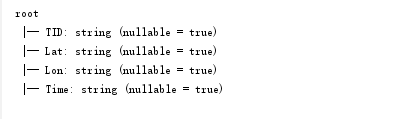
1. 、将数据写入DataFrame

我通过实现定义的RawDataRecord，RawDataRecord中有四个字段，分别是TID:String, Lat:String,Lon:String,Time:String



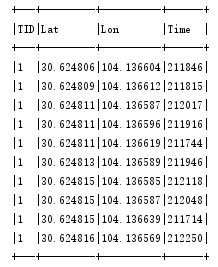
**import** sqlContext.implicits.\_  
*//将原始数据映射到DataFrame中，，通过,分隔***var** taxiDF = sc.textFile(**"data/taxi.csv"**).map {  
 x =>  
 **var** data = x.split(**","**)  
 *RawDataRecord*(data(0),data(1),data(2),data(3))  
}.toDF()

观察taxiDF中表的结构：



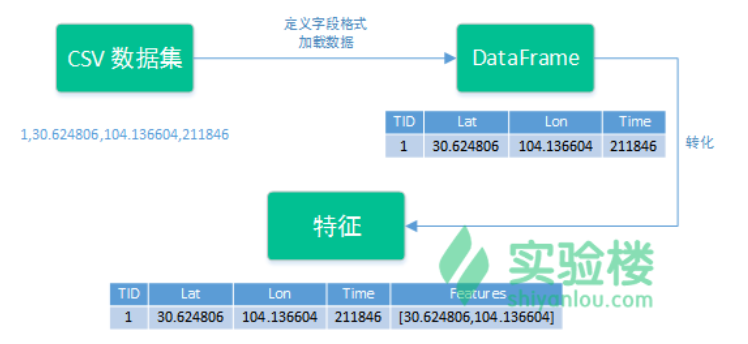
DataFrame中的的三个参数分别为字段名称、字段数据类型和是否不允许为空。

观察taxiDF表中数据：



## 3.3、特征数组

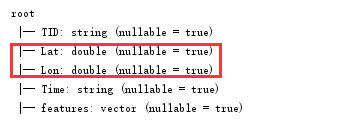
RawDataRecord中有四个字段，分别是TID:String, Lat:String,Lon:String,Time:String，在进行聚类的时候，应该剔除TID和Time字段，仅对出租车状态为载客时的纬度和经度进行聚类，我们在原始数据taxiDF的DataFrame提取新的特征，进行聚类。



1. 创建向量装配器

向量装配器VectorAssembler中的setInputCols方法传入的不支持String类型，要先对表的结构进行改动。

Lat:String,Lon:String----------->Lat:Double,Lon:Double

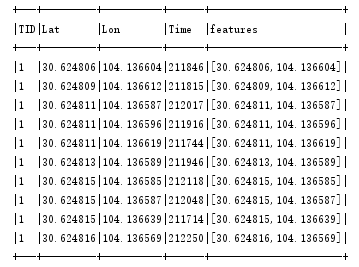


**val** analysisData = taxiDF.withColumn(**"Lat"**, taxiDF(**"Lat"**).cast(DoubleType))  
**val** analysisData1 = analysisData.withColumn(**"Lon"**, analysisData(**"Lon"**).cast(DoubleType)

使用向量装配器VectorAssembler，将转为新的特征列名称放入到setInputCols（）中，生成新特征的名称放在setOutputCol()中，使用向量装配器对象的 transform() 方法对导入的数据（taxiData）进行转化。

**val** va = **new** VectorAssembler().setInputCols(*Array*(**"Lat"**,**"Lon"**)).setOutputCol(**"features"**)  
**val** taxiDF2 = va.transform(analysisData1)

显示特征向量转换后的DataFrame:



## 3.4、划分数据集

将taxiDF2划分为训练集和测试集，比例为7:3

**val** trainTestRatio = *Array*(0.7, 0.3)  
**val** *Array*(trainingData, testData) = taxiDF2.randomSplit(trainTestRatio, 2333)

## 3.5、聚类模型

setK()：聚类的簇数量。

setFeaturesCol()：聚类特征字段名称。

setPredictionCol：预测值名称

fit()：训练模型

**val** km = **new** KMeans().setK(10).setFeaturesCol(**"features"**).setPredictionCol(**"prediction"**)  
*//训练聚类模型***val** kmModel = km.fit(taxiDF2)  
**val** kmResult = kmModel.clusterCenters  
**val** kmRDD1 = sc.parallelize(kmResult)  
**val** kmRDD2 = kmRDD1.map(x => (x(1), x(0)))  
kmRDD2.saveAsTextFile(**"output/kmResult"**)

调用训练集训练的模型kmModel对测试集进行预测。

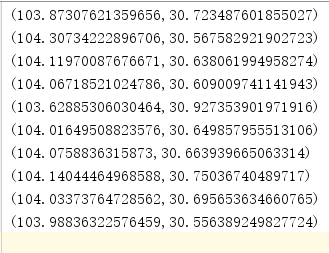
*//预测结果***val** predictions = kmModel.transform(testData)

## 3.6、本地测试

### 3.6.1、簇中心

将生成簇中心打印出来：

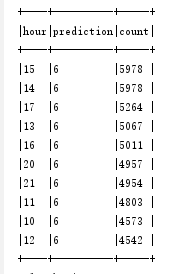
kmRDD2.foreach(println(\_))



### 3.6.2、时段预测

统计每个时段属于哪个簇，并将其个数打印出来

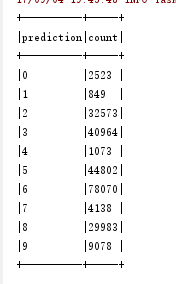
*//对每个小时不同预测类型的数量进行统计*predictions.registerTempTable(**"predictions"**)  
**val** tmpQuery = predictions.select(*substring*(**$"Time"**,0,2).alias(**"hour"**), **$"prediction"**).groupBy(**"hour"**, **"prediction"**)  
**val** predictCount = tmpQuery.agg(*count*(**"prediction"**).alias(**"count"**)).orderBy(*desc*(**"count"**))  
*println*(**"输出对每个小时不同预测类型的数量进行统计："**)  
predictCount.show(10,**false**)



### 3.6.3、聚类数目

统计不同簇包含的个数打印出来

*//统计聚类簇的数目***val** busyZones = predictions.groupBy(**"prediction"**).count()  
*println*(**"输出统计聚类簇的数目："**)  
busyZones.show(10,**false**)



## 3.7、集群测试

（1）、将数据集taxi.txt文件上传到hdfs中根目录下。出租车聚类hncj.jar上传到/usr/local目录下。

（2）、从master节点进入到/usr/local/spark/sbin启动Spark

cd /usr/local/spark/sbin

开启：

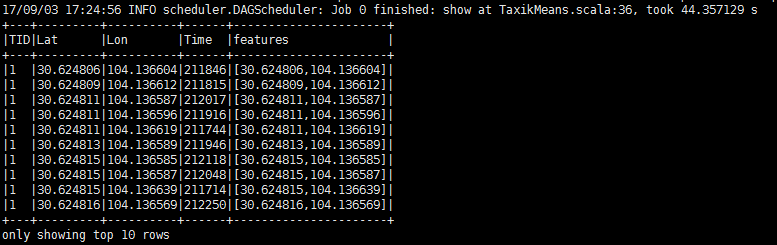
start-all.sh

（3）、执行程序

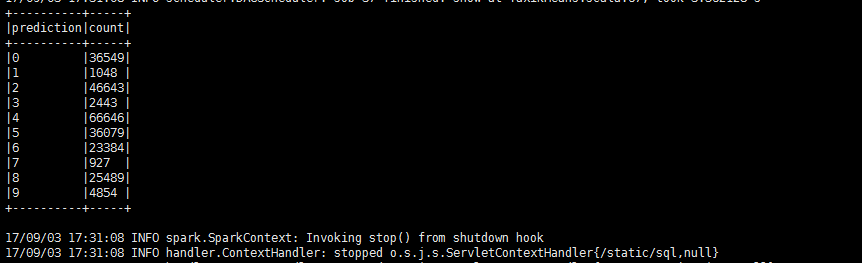
spark-submit --master spark://192.168.56.100:7077 --name TaxikMeans --class TaxikMeans.cluster.TaxikMeans --executor-memory 500m --total-executor-cores 2 /usr/local/hncj.jar hdfs://192.168.56.100:9000/taxi.txt

IMG_256

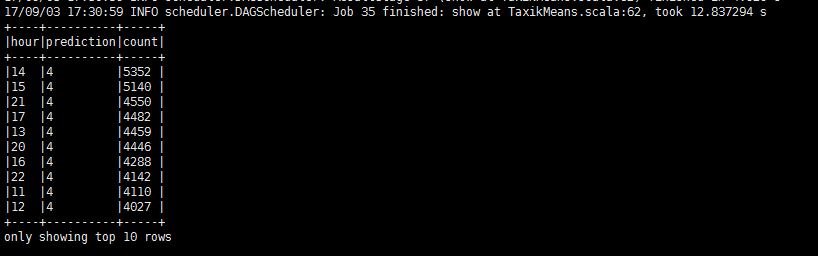
1. 、运行结果



统计不同簇包含的个数打印出来



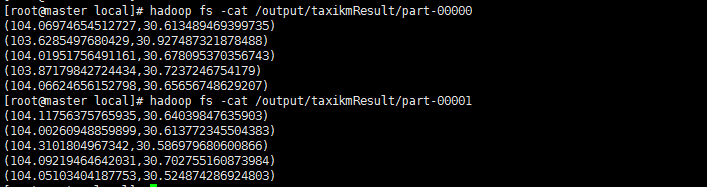
统计每个时段属于哪个簇，并将其个数打印出来



1. 、在hdfs上保存的结果

![IN7UMUT11L$](U{BTF{U4$V](data:image/png;base64,)

将生成簇中心打印出来：



# 错误总结

（1）、抛出异常：

17/08/28 12:14:18 ERROR Shell: Failed to locate the winutils binary in the hadoop binary path  
java.io.IOException: Could not locate executable null\bin\winutils.exe in the Hadoop binaries.

问题分析：

可能没有配置hadoop环境变量，使得运行程序时找不到winutils.exe，解决方法配置hadoop环境变量引入winutils.exe

解决办法一：

System.setProperty("hadoop.home.dir", "E:\\hadoop");

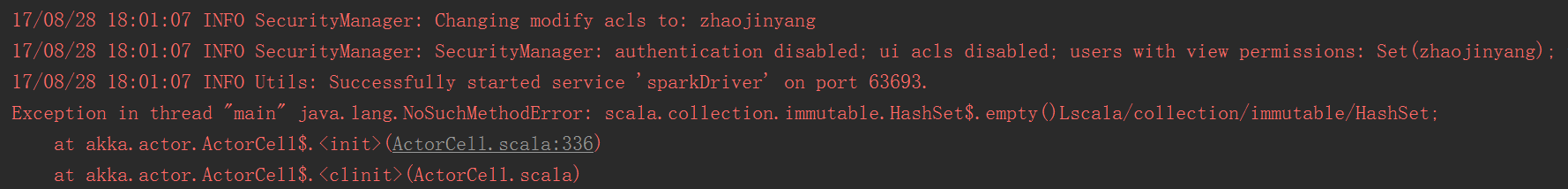
解决方法二：

下载winutils.exe，下载地址是

https://github.com/srccodes/hadoop-common-2.2.0-bin，下载后是文件名是hadoop-common-2.2.0-bin-master.zip,随便解压到一个目录，然后在系统变量path里增加$HADOOP\_HOME\bin 即可。再次运行程序，正常执行。

（2）、抛出异常

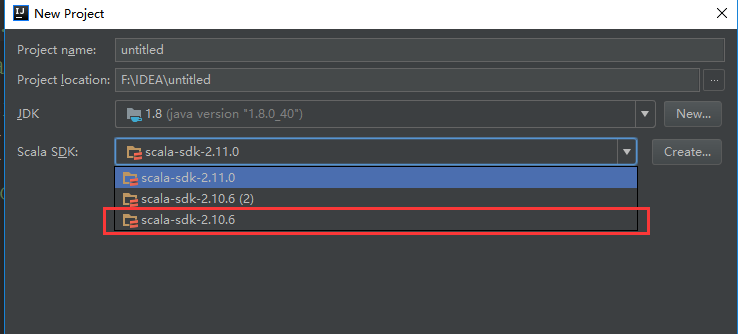
Exception in thread "main" java.lang.NoSuchMethodError: scala.collection.immutable.HashSet$.empty()Lscala/collection/immutable/HashSet;



原因分析：

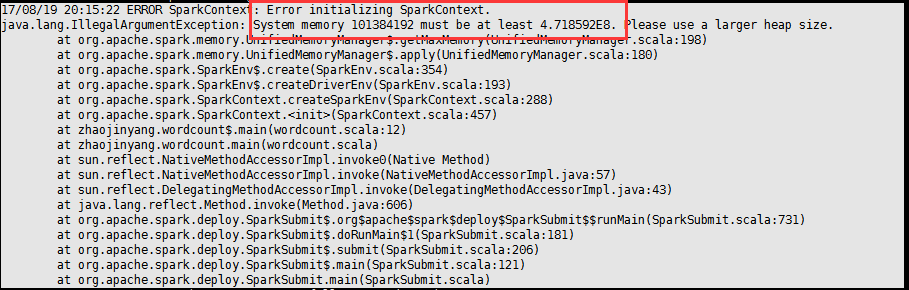
scala-sdk和spark版本不比配问题

解决办法：scala-sdk和spark版本不比配问题，在网上下载新的scala，更改scala版本，可以解决问题。



（3）、抛出异常：

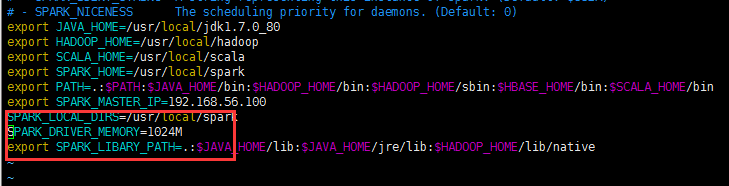
如果出现内存不足的问题，可能使在spark-env.sh配置文档问题。



解决办法：

将SPARK\_DRIVER\_MEMORY修改为1024M

SPARK\_DRIVER\_MEMORY=1024M



（4）、关于IntelliJ运行速度问题

原来未优化IntelliJ之前，运行一个程序，到达下图位置会停止1-2min,运行速度偏慢。

IMG_256

优化IntelliJ方法：

如图用记事本打开

idea.exe.vmoptions

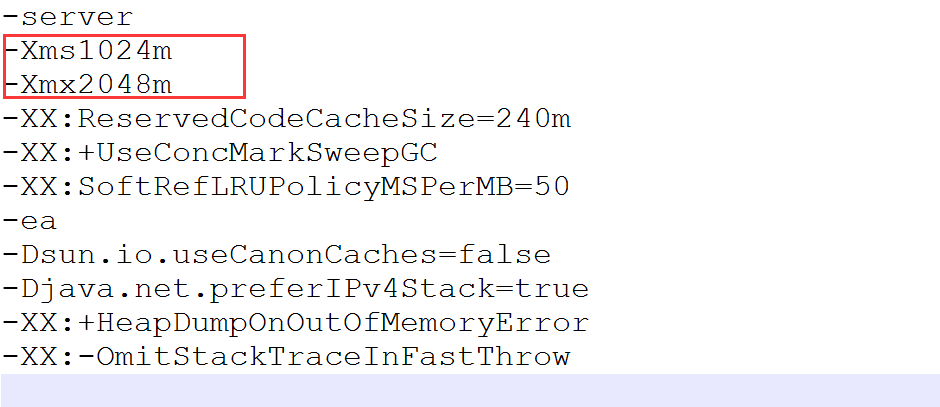
idea64.exe.vmoptions



如下图修改

idea.exe.vmoptions

idea64.exe.vmoptions



# 参考网站

Spark 官方文档

<http://spark.apache.org/docs/1.6.3/api/scala/index.html#org.apache.spark.package>

文本分类评价(SogouTCE)：

<http://www.sogou.com/labs/resource/tce.php>

Spark机器学习：TF-IDF实例讲解 ：

<http://blog.csdn.net/jiangpeng59/article/details/52786344>

Spark MLlib实现的中文文本分类–Naive Bayes

<http://lxw1234.com/archives/2016/01/605.htm>

出租车数据分析：

<http://www.cnblogs.com/mrchige/p/6346885.html>