19-综合案例实战：处理加州房屋信息，构建线性回归模型

你好，我是蔡元楠。

今天我要与你分享的主题是“综合案例实战：处理加州房屋信息，构建线性回归模型”。

通过之前的学习，我们对Spark各种API的基本用法有了一定的了解，还通过统计词频的实例掌握了如何从零开始写一个Spark程序。那么现在，让我们从一个真实的数据集出发，看看如何用Spark解决实际问题。

数据集介绍

为了完成今天的综合案例实战，我使用的是美国加州1990年房屋普查的数据集。

数据集中的每一个数据都代表着一块区域内房屋和人口的基本信息，总共包括9项：

1. 该地区中心的纬度（latitude）
2. 该地区中心的经度（longitude）
3. 区域内所有房屋屋龄的中位数（housingMedianAge）
4. 区域内总房间数（totalRooms）
5. 区域内总卧室数（totalBedrooms）
6. 区域内总人口数（population）
7. 区域内总家庭数（households）
8. 区域内人均收入中位数（medianIncome）
9. 该区域房价的中位数（medianHouseValue）

也就是说，我们可以把每一个数据看作一个地区，它含有9项我们关心的信息，也就是上面提到的9个指标。比如下面这个数据：

```
-122.230000,37.880000,41.000000,880.000000,129.000000,322.000000,126.000000,8.325200,452600.000000'
```

这个数据代表该地区的经纬度是（-122.230000,37.880000），这个地区房屋历史的中位数是41年，所有房屋总共有880个房间，其中有129个卧室。这个地区内共有126个家庭和322位居民，人均收入中位数是8.3252万，房价中位数是45.26万。

这里的地域单位是美国做人口普查的最小地域单位，平均一个地域单位中有1400多人。在这个数据集中共有两万多个这样的数据。显然，这样小的数据量我们并“不需要”用Spark来处理，但是，它可以起到一个很好的示例作用。这个数据集可以从网上下载到。这个数据集是在1997年的一篇学术论文中创建的，感兴趣的同学可以去亲自下载，并加以实践。

那么我们今天的目标是什么呢？就是用已有的数据，构建一个**线性回归模型**，来预测房价。

我们可以看到，前8个属性都可能对房价有影响。这里，我们假设这种影响是线性的，我们就可以找到一个类似**A=bB+cC+dD+⋯+iI**的公式，A代表房价，B到I分别代表另外八个属性。这样，对于不在数据集中的房子，我们可以套用这个公式来计算出一个近似的房价。由于专栏的定位是大规模数据处理专栏，所以我们不会细讲统计学的知识。如果你对统计学知识感兴趣，或者还不理解什么是线性回归的话，可以去自行学习一下。

进一步了解数据集

每当我们需要对某个数据集进行处理时，不要急着写代码。你一定要先观察数据集，了解它的特性，并尝试对它做一些简单的预处理，让数据的可读性更好。这些工作我们最好在Spark的交互式Shell上完成，而不是创建python的源文件并执行。因为，在Shell上我们可以非常直观而简便地看到每一步的输出。

首先，让我们把数据集读入Spark。

```
from pyspark.sql import SparkSession

# 初始化SparkSession和SparkContext
spark = SparkSession.builder
  .master("local")
  .appName("California Housing ")
  .config("spark.executor.memory", "1gb")
  .getOrCreate()
sc = spark.sparkContext

# 读取数据并创建RDD
rdd = sc.textFile('/Users/yourName/Downloads/CaliforniaHousing/cal_housing.data')

# 读取数据每个属性的定义并创建RDD
header = sc.textFile('/Users/yourName/Downloads/CaliforniaHousing/cal_housing.domain')
```

这样，我们就把房屋信息数据和每个属性的定义读入了Spark，并创建了两个相应的RDD。你还记得吧？RDD是有一个惰性求值的特性的，所以，我们可以用collect()函数来把数据输出在Shell上。

```
header.collect()

[u'longitude: continuous.', u'latitude: continuous.', u'housingMedianAge: continuous. ', u'totalRooms: cont
```

这样，我们就得到了每个数据所包含的信息，这和我们前面提到的9个属性的顺序是一致的，而且它们都是连续的值，而不是离散的。你需要注意的是，collect()函数会把所有数据都加载到内存中，如果数据很大的话，有可能会造成内存泄漏，所以要小心使用。平时比较常见的方法是用take()函数去只读取RDD中的某几个元素。

由于RDD中的数据可能会比较大，所以接下来让我们读取它的前两个数据。

```
rdd.take(2)

[u'-122.230000,37.880000,41.000000,880.000000,129.000000,322.000000,126.000000,8.325200,452600.000000', u'-
```

由于我们是用SparkContext的textFile函数去创建RDD，所以每个数据其实是一个大的字符串，各个属性之间用逗号分隔开来。这不利于我们之后的处理，因为我们可能会需要分别读取每个对象的各个属性。所以，让我们用map函数把大字符串分隔成数组，这会方便我们的后续操作。

```
rdd = rdd.map(lambda line: line.split(","))
rdd.take(2)

[[u'-122.230000', u'37.880000', u'41.000000', u'880.000000', u'129.000000', u'322.000000', u'126.000000', u
```

我们在前面学过，Spark SQL的DataFrame API在查询结构化数据时更方便使用，而且性能更好。在这个例子中你可以看到，数据的schema是定义好的，我们需要去查询各个列，所以DataFrame API显然更加适用。所以，我们需要先把RDD转换为DataFrame。

具体来说，就是需要把之前用数组代表的对象，转换成为Row对象，再用toDF()函数转换成DataFrame。

```
from pyspark.sql import Row

df = rdd.map(lambda line: Row(longitude=line[0],
                              latitude=line[1],
                              housingMedianAge=line[2],
                              totalRooms=line[3],
                              totalBedRooms=line[4],
                              population=line[5],
                              households=line[6],
                              medianIncome=line[7],
                              medianHouseValue=line[8])).toDF()
```

现在我们可以用show()函数打印出这个DataFrame所含的数据表。

```
df.show()
```

```
+----------+---------------+---------+----------+---------------+------------+----------+-------------+-----------+
| households|housingMedianAge| latitude|  longitude|medianHouseValue|medianIncome| population|totalBedRooms| totalRooms|
+----------+---------------+---------+----------+---------------+------------+----------+-------------+-----------+
| 126.000000|       41.000000|37.880000|-122.230000|   452600.000000|    8.325200| 322.000000|   129.000000| 880.000000|
|1138.000000|       21.000000|37.860000|-122.220000|   358500.000000|    8.301400|2401.000000|  1106.000000|7099.000000|
| 177.000000|       52.000000|37.850000|-122.240000|   352100.000000|    7.257400| 496.000000|   190.000000|1467.000000|
| 219.000000|       52.000000|37.850000|-122.250000|   341300.000000|    5.643100| 558.000000|   235.000000|1274.000000|
| 259.000000|       52.000000|37.850000|-122.250000|   342200.000000|    3.846200| 565.000000|   280.000000|1627.000000|
| 193.000000|       52.000000|37.850000|-122.250000|   269700.000000|    4.036800| 413.000000|   213.000000| 919.000000|
| 514.000000|       52.000000|37.840000|-122.250000|   299200.000000|    3.659100|1094.000000|   489.000000|2535.000000|
| 647.000000|       52.000000|37.840000|-122.250000|   241400.000000|    3.120000|1157.000000|   687.000000|3104.000000|
| 595.000000|       42.000000|37.840000|-122.260000|   226700.000000|    2.080400|1206.000000|   665.000000|2555.000000|
| 714.000000|       52.000000|37.840000|-122.250000|   261100.000000|    3.691200|1551.000000|   707.000000|3549.000000|
| 402.000000|       52.000000|37.850000|-122.260000|   281500.000000|    3.203100| 910.000000|   434.000000|2202.000000|
| 734.000000|       52.000000|37.850000|-122.260000|   241800.000000|    3.270500|1504.000000|   752.000000|3503.000000|
| 468.000000|       52.000000|37.850000|-122.260000|   213500.000000|    3.075000|1098.000000|   474.000000|2491.000000|
| 174.000000|       52.000000|37.840000|-122.260000|   191300.000000|    2.673600| 345.000000|   191.000000| 696.000000|
| 620.000000|       52.000000|37.850000|-122.260000|   159200.000000|    1.916700|1212.000000|   626.000000|2643.000000|
| 264.000000|       50.000000|37.850000|-122.260000|   140000.000000|    2.125000| 697.000000|   283.000000|1120.000000|
| 331.000000|       52.000000|37.850000|-122.270000|   152500.000000|    2.775000| 793.000000|   347.000000|1966.000000|
| 303.000000|       52.000000|37.850000|-122.270000|   155500.000000|    2.120200| 648.000000|   293.000000|1228.000000|
| 419.000000|       50.000000|37.840000|-122.260000|   158700.000000|    1.991100| 990.000000|   455.000000|2239.000000|
| 275.000000|       52.000000|37.840000|-122.270000|   162900.000000|    2.603300| 690.000000|   298.000000|1503.000000|
+----------+---------------+---------+----------+---------------+------------+----------+-------------+-----------+
only showing top 20 rows
```

这里每一列的数据格式都是string，但是，它们其实都是数字，所以我们可以通过cast()函数把每一列的类型转换成float。

```
def convertColumn(df, names, newType)
 for name in names:
    df = df.withColumn(name, df[name].cast(newType))
 return df

columns = ['households', 'housingMedianAge', 'latitude', 'longitude', 'medianHouseValue', 'medianIncome', '

df = convertColumn(df, columns, FloatType())
```

转换成数字有很多优势。比如，我们可以按某一列，对所有对象进行排序，也可以计算平均值等。比如，下面这段代码就可以统计出所有建造年限各有多少个房子。

```
df.groupBy("housingMedianAge").count().sort("housingMedianAge",ascending=False).show()
```

预处理

通过上面的数据分析，你可能会发现这些数据还是不够直观。具体的问题有：

1. 房价的值普遍都很大，我们可以把它调整成相对较小的数字；
2. 有的属性没什么意义，比如所有房子的总房间数和总卧室数，我们更加关心的是平均房间数；
3. 在我们想要构建的线性模型中，房价是结果，其他属性是输入参数。所以我们需要把它们分离处理；
4. 有的属性最小值和最大值范围很大，我们可以把它们标准化处理。

对于第一点，我们观察到大多数房价都是十万起的，所以可以用withColumn()函数把所有房价都除以100000。

```
df = df.withColumn("medianHouseValue", col("medianHouseValue")/100000)
```

对于第二点，我们可以添加如下三个新的列：

- 每个家庭的平均房间数：roomsPerHousehold
- 每个家庭的平均人数：populationPerHousehold
- 卧室在总房间的占比：bedroomsPerRoom

当然，你们可以自由添加你们觉得有意义的列，这里的三个是我觉得比较典型的。同样，用withColumn()函数可以容易地新建列。

```
df = df.withColumn("roomsPerHousehold", col("totalRooms")/col("households"))
  .withColumn("populationPerHousehold", col("population")/col("households"))
  .withColumn("bedroomsPerRoom", col("totalBedRooms")/col("totalRooms"))
```

同样，有的列是我们并不关心的，比如经纬度，这个数值很难有线性的意义。所以我们可以只留下重要的信息列。

```
df = df.select("medianHouseValue",
              "totalBedRooms",
              "population",
              "households",
              "medianIncome",
              "roomsPerHousehold",
              "populationPerHousehold",
              "bedroomsPerRoom")
```

对于第三点，最简单的办法就是把DataFrame转换成RDD，然后用map()函数把每个对象分成两部分：房价和一个包含其余属性的列表，然后在转换回DataFrame。

```
from pyspark.ml.linalg import DenseVector

input_data = df.rdd.map(lambda x: (x[0], DenseVector(x[1:])))
df = spark.createDataFrame(input_data, ["label", "features"])
```

我们重新把两部分重新标记为“label”和“features”，label代表的是房价，features代表包括其余参数的列表。

对于第四点，数据的标准化我们可以借助Spark的机器学习库Spark ML来完成。Spark ML也是基于DataFrame，它提供了大量机器学习的算法实现、数据流水线（pipeline）相关工具和很多常用功能。由于

本专栏的重点是大数据处理，所以我们并没有介绍Spark ML，但是我强烈推荐同学们有空去了解一下它。

在这个AI和机器学习的时代，我们不能落伍。

```
from pyspark.ml.feature import StandardScaler

standardScaler = StandardScaler(inputCol="features", outputCol="features_scaled")
scaler = standardScaler.fit(df)
scaled_df = scaler.transform(df)
```

在第二行，我们创建了一个StandardScaler，它的输入是features列，输出被我们命名为features_scaled。第三、第四行，我们把这个scaler对已有的DataFrame进行处理，让我们看下代码块里显示的输出结果。

```
scaled_df.take(1)

[Row(label=4.526, features=DenseVector([129.0, 322.0, 126.0, 8.3252, 6.9841, 2.5556, 0.1466]), features_sca
```

我们可以清楚地看到，这一行新增了一个features_scaled的列，它里面每个数据都是标准化过的，我们应该用它，而非features来训练模型。

创建模型

上面的预处理都做完后，我们终于可以开始构建线性回归模型了。

首先，我们需要把数据集分为训练集和测试集，训练集用来训练模型，测试集用来评估模型的正确性。DataFrame的randomSplit()函数可以很容易的随机分割数据，这里我们将80%的数据用于训练，剩下20%作为测试集。

```
train_data, test_data = scaled_df.randomSplit([.8,.2],seed=123)
```

用Spark ML提供的LinearRegression功能，我们可以很容易得构建一个线性回归模型，如下所示。

```
from pyspark.ml.regression import LinearRegression

lr = LinearRegression(featuresCol='features_scaled', labelCol="label", maxIter=10, regParam=0.3, elasticNet
linearModel = lr.fit(train_data)
```

LinearRegression可以调节的参数还有很多，你可以去官方API文档查阅，这里我们只是示范一下。

模型评估

现在有了模型，我们终于可以用linearModel的transform()函数来预测测试集中的房价，并与真实情况进行对比。代码如下所示。

```
predicted = linearModel.transform(test_data)
predictions = predicted.select("prediction").rdd.map(lambda x: x[0])
labels = predicted.select("label").rdd.map(lambda x: x[0])
predictionAndLabel = predictions.zip(labels).collect()
```

我们用RDD的zip()函数把预测值和真实值放在一起，这样可以方便地进行比较。比如让我们看一下前两个对比结果。

```
predictionAndLabel.take(2)

[(1.4491508524918457, 1.14999), (1.5831547768979277, 0.964)]
```

这里可以看出，我们的模型预测的结果有些偏小，这可能有多个因素造成。最直接的原因就是房价与我们挑选的列并没有强线性关系，而且我们使用的参数也可能不够准确。

这一讲我只是想带着你一起体验下处理真实数据集和解决实际问题的感觉，想要告诉你的是这种通用的思想，并帮助你继续熟悉Spark各种库的用法，并不是说房价一定就是由这些参数线性决定了。感兴趣的同学可以去继续优化，或者尝试别的模型比如逻辑回归。

小结

这一讲我们通过一个真实的数据集，通过以下步骤解决了一个实际的数据处理问题：

1. 观察并了解数据集
2. 数据清洗
3. 数据的预处理
4. 训练模型
5. 评估模型

其实这里还可以有与“优化与改进”相关的内容，这里没有去阐述是因为我们的首要目的依然是熟悉与使用Spark各类API。相信通过今天的学习，你初步了解了数据处理问题的一般思路，并强化了对RDD、DataFrame和机器学习API的使用。

实践与思考题

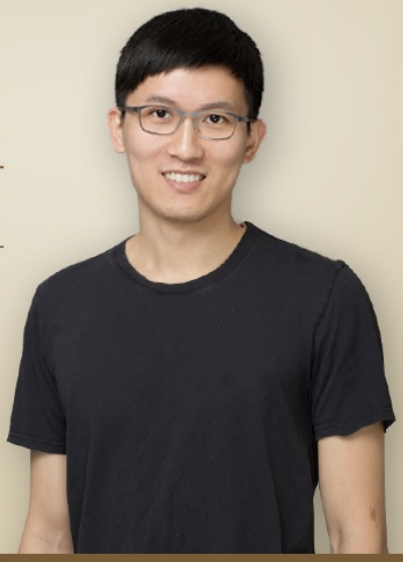
今天请你下载这个数据集，按文章的介绍去动手实践一次。如果有时间的话，还可以对这个过程的优化和改进提出问题并加以解决。

欢迎你在留言板贴出自己的idea。如果你觉得有所收获，也欢迎你把文章分享给朋友。

极客时间

大规模数据处理实战

Google一线工程师的大数据架构实战经验



蔡元楠

Google Brain 资深工程师