

19 | 综合案例实战：处理加州房屋信息，构建线性回归模型

2019-05-31 蔡元楠 来自北京

《大规模数据处理实战》



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：

 复制代码

```
1  -122.230000,37.880000,41.000000,880.000000,129.000000,322.000000,126.000000,8.325
```

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 $A = bB + cC + dD + \dots + iI$ 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。

 复制代码

```
1 from pyspark.sql import SparkSession
2
3 # 初始化SparkSession和SparkContext
4 spark = SparkSession.builder
5     .master("local")
6     .appName("California Housing ")
```

```
7     .config("spark.executor.memory", "1gb")
8     .getOrCreate()
9  sc = spark.sparkContext
10
11 # 读取数据并创建RDD
12 rdd = sc.textFile('/Users/yourName/Downloads/CaliforniaHousing/cal_housing.data')
13
14 # 读取数据每个属性的定义并创建RDD
15 header = sc.textFile('/Users/yourName/Downloads/CaliforniaHousing/cal_housing.dom
```


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。

 复制代码

```
1 header.collect()
2
3 [u'longitude: continuous.', u'latitude: continuous.', u'housingMedianAge: continu
```

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。

 复制代码

```
1 rdd.take(2)
2
3 [u'-122.230000,37.880000,41.000000,880.000000,129.000000,322.000000,126.000000,8.
```

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。

[复制代码](#)

```
1 rdd = rdd.map(lambda line: line.split(","))
2 rdd.take(2)
3
4 [[u'-122.230000', u'37.880000', u'41.000000', u'880.000000', u'129.000000', u'322
```

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。

[复制代码](#)

```
1 from pyspark.sql import Row
2
3 df = rdd.map(lambda line: Row(longitude=line[0],
4                               latitude=line[1],
5                               housingMedianAge=line[2],
6                               totalRooms=line[3],
7                               totalBedRooms=line[4],
8                               population=line[5],
9                               households=line[6],
10                              medianIncome=line[7],
11                              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。

[复制代码](#)

```
1 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	201500.000000	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。

复制代码

```
1 def convertColumn(df, names, newType)
2     for name in names:
3         df = df.withColumn(name, df[name].cast(newType))
4     return df
5
6 columns = ['households', 'housingMedianAge', 'latitude', 'longitude', 'medianHous
7
8 df = convertColumn(df, columns, FloatType())
```

转换成数字有很多优势。比如，我们可以按某一行，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。

复制代码

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

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。

 复制代码

```
1 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。

 复制代码

```
1 df = df.withColumn("roomsPerHousehold", col("totalRooms")/col("households"))
2     .withColumn("populationPerHousehold", col("population")/col("households"))
3     .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。

[📄 复制代码](#)

```
1 df = df.select("medianHouseValue",
2               "totalBedRooms",
3               "population",
4               "households",
5               "medianIncome",
6               "roomsPerHousehold",
7               "populationPerHousehold",
8               "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。

[📄 复制代码](#)

```
1 from pyspark.ml.linalg import DenseVector
2
3 input_data = df.rdd.map(lambda x: (x[0], DenseVector(x[1:])))
4 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。

[📄 复制代码](#)

```
1 from pyspark.ml.feature import StandardScaler
2
3 standardScaler = StandardScaler(inputCol="features", outputCol="features_scaled")
4 scaler = standardScaler.fit(df)
5 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。

 复制代码


```
1 scaled_df.take(1)
2
3 [Row(label=4.526, features=DenseVector([129.0, 322.0, 126.0, 8.3252, 6.9841, 2.55
```

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。

 复制代码

```
1 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。


 复制代码

```
1 from pyspark.ml.regression import LinearRegression
2
3 lr = LinearRegression(featuresCol='features_scaled', labelCol="label", maxIter=10
4 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。

 复制代码

```
1 predicted = linearModel.transform(test_data)
2 predictions = predicted.select("prediction").rdd.map(lambda x: x[0])
3 labels = predicted.select("label").rdd.map(lambda x: x[0])
4 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。

 复制代码

```
1 predictionAndLabel.take(2)
2
3 [(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。

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。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

精选留言 (19)



coder

2019-05-31

老师的代码可以po到Github上，这样大家都可以学习了😊😊😊



👍 17



JustDoDT

2019-09-02

终于跑通了，不容易啊，刚开始数据集没下载正确。有空值，老师给的数据集较干净。别的数据集，要记得去除空值。

下面是实践代码jupyter

<https://github.com/LearningChanging/spark-exercise/blob/master/19/CaliforniaHousing.ipynb>

作者回复: 给你点赞👍加油

共 3 条评论 >

👍 12



kylin

2019-06-01

请问为什么不用dataset进行数据处理而是用dateFrame?

作者回复: PySpark的SQL库只有DataFrame，并没有DataSet。不过在Scala和Java中，DataSet已经成为了统一的SQL入口。

共 2 条评论 >

👍 12



hufox

2019-06-16

最后一句的代码改成 predictionAndLabel[:2]，可以了！

共 2 条评论 >

👍 3



咸鱼与果汁

2020-02-12

spark df的数据处理还是略显复杂，感觉大部分的算法人员还是使用pandas进行数据预处理，请问使用pandas是不是就无法发挥spark RDD的威力了？这种情况下spark就相当于是一个异步任务处理框架？

共 1 条评论 >

👍 2



henry

2019-09-18

最后一步，“模型预测的结果有些偏小”，这一点，从结果上看，不是应该预测的结果要大一些吗？



👍 1



黄智寿

2019-08-13

老师，你好，数据集的下载地址能发一下吗？



1

**毛毛**

2019-06-23

老师，建议在案例讲解时对用到的算法大概解释下，比如srandscaler，这不是太清楚什么用处？谢谢

共 1 条评论 >



1

**hufox**

2019-06-16

在执行最后一句代码predictionAndLabel.take(2)时报错：

AttributeError Traceback (most recent call last)

<ipython-input-35-0700ca2381fb> in <module>

----> 1 predictionAndLabel.take(2)

AttributeError: 'list' object has no attribute 'take'

共 3 条评论 >



2

**gotojeff**

2019-06-05

dataset不支持python, 所以在python里只有DF, 这算不算python的一大劣势? scala是更好的选择?

作者回复: 感谢提问。PySpark现在不支持DataSet, 只有Scala和Java支持。这是由语言特性决定的, Python是动态类型的语言, 而DataSet是强类型的, 要求在编译时检测类型安全。所以, 在所有用Python的代码例子中, 我用的都是DataFrame。

大部分人都同意在Spark中, Scala和Python是优于Java和R的。至于在Spark生态中, Scala和Python孰优孰劣, 这是个很主观的问题, 我们不能只因为不支持DataSet这一点就说Python比Scala差。

Scala确实很优秀, Spark原生的实现就是用Scala写的, 所以任何新发布的功能肯定支持Scala, 官方文档也都是用Scala来举例子。而且Scala的性能要优于Python。但是Python也有很多优点, 比如容易学习、应用场景广。这两种语言在Spark的世界中都可以满足我们绝大多数的需求, 选择任何一个都不是错误的。

共 2 条评论 >



2



2019-06-03

老师代码可以上github吗，这样同学们可以参考下



1



Zoe

2019-05-31

看前两篇文章时还在想，没什么练手的机会啊，今天就推送了实战练习，有一种终于跟上大神思维的幻觉，开心！

作者回复: 👍



1



Geek5350

2021-05-21

请问数据集在哪里下载？



寻水的小鱼

2021-01-11

```
>>>
```

```
>>> def convertColumn(df, names, newType)
```

```
File "<stdin>", line 1
```

```
    def convertColumn(df, names, newType)
```

```
        ^
```

SyntaxError: invalid syntax



之渊

2020-08-21

java 版的代码demo : <https://gitee.com/oumin12345/daimademojihe/tree/master/cloudx/bigdata/src/main/java/test/spark>

建议初学者没写过的可以自己敲一下。

机器学习的入门可以看看：

<https://my.oschina.net/ouminzy/blog/4437101>





xianhai
2020-05-11

最好还是给完整的代码，节省初学者的时间。



Chloe
2020-02-15

报错：

"

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name 'FloatType' is not defined

"

Google了一下: <https://stackoverflow.com/questions/40701122/unexpected-type-class-pyspark-sql-types-datatype-singleton-when-casting-to-i>

加了这句就好了：

```
from pyspark.sql.types import FloatType
```

大家还有人也遇到这个错误吗？



JustDoDT
2019-09-02

StandardScaler 归一化之后，两列变成 NaN 了搞不明白

```
+-----+-----+-----+
|label|      features|  features_scaled|
+-----+-----+-----+
|4.526|[129.0,322.0,126....|[NaN,0.2843362208...|
|3.585|[1106.0,2401.0,11...|[NaN,2.1201592122...|
+-----+-----+-----+
```

```
scaled_df.take(2)
```

```
[Row(label=4.526, features=DenseVector([129.0, 322.0, 126.0, 8.3252, 6.9841, 2.5556, 0.1466]), features_scaled=DenseVector([nan, 0.2843, 0.3296, 4.3821, 2.8228, 0.2461, nan])),
 Row(label=3.585, features=DenseVector([1106.0, 2401.0, 1138.0, 8.3014, 6.2381, 2.1098, 0.1558]), features_scaled=DenseVector([nan, 2.1202, 2.9765, 4.3696, 2.5213, 0.2031, nan]))]
```

共 1 条评论 >



西北偏北

2019-08-05

一些实际的大数据处理，确实需要数学啊.....怎么才能把数学学好？

作者回复：在实践中看到需要可以回头看一下基础知识

