Predicting House Prices with Apache Spark

LINEAR REGRESSION

In this we'll make use of the California Housing (http://www.dcc.fc.up.pt/~ltorgo/Regression/cal_housing.html) data set. Note, of course, that this is actually 'small' data and that using Spark in this context might be overkill; This notebook is for educational purposes only and is meant to give us an idea of how we can use PySpark to build a machine learning model.

1. Understanding the Data Set

The California Housing data set appeared in a 1997 paper titled *Sparse Spatial Autoregressions*, written by Pace, R. Kelley and Ronald Barry and published in the Statistics and Probability Letters journal. The researchers built this data set by using the 1990 California census data.

The data contains one row per census block group. A block group is the smallest geographical unit for which the U.S. Census Bureau publishes sample data (a block group typically has a population of 600 to 3,000 people). In this sample a block group on average includes 1425.5 individuals living in a geographically compact area.

These spatial data contain 20,640 observations on housing prices with 9 economic variables:

> **Longitude:**refers to the angular distance of a geographic place north or south of the earth’s equator for each block group
> **Latitude :**refers to the angular distance of a geographic place east or west of the earth’s equator for each block group
> **Housing Median Age:**is the median age of the people that belong to a block group. Note that the median is the value that lies at the midpoint of a frequency distribution of observed values
> **Total Rooms:**is the total number of rooms in the houses per block group
> **Total Bedrooms:**is the total number of bedrooms in the houses per block group
> **Population:**is the number of inhabitants of a block group
> **Households:**refers to units of houses and their occupants per block group
> **Median Income:**is used to register the median income of people that belong to a block group
> **Median House Value:**is the dependent variable and refers to the median house value per block group

What's more, we also learn that all the block groups have zero entries for the independent and dependent variables have been excluded from the data.

The Median house value is the dependent variable and will be assigned the role of the target variable in our ML model.

In [1]:

```
#!pip install pyspark
```

In [2]:

```
import os
import numpy as np
import pandas as pd


#eda
from pyspark.sql import SparkSession, Row
from pyspark.sql.functions import udf,col
from pyspark.sql.types import *
import pyspark.sql.functions as F

from pyspark.ml.feature import VectorAssembler, StandardScaler
from pyspark.ml.regression import LinearRegression
import seaborn as sns
```

In [3]:

```
import matplotlib.pyplot as plt
%matplotlib inline


pd.set_option('display.max_columns', 200)
pd.set_option('display.max_colwidth', 400)

#圖像細節
from matplotlib import rcParams
sns.set(context='notebook', style='whitegrid', rc={'figure.figsize': (18,4)})
rcParams['figure.figsize'] = 18,4

%matplotlib inline

#記得提高圖的解析度
%config InlineBackend.figure_format = 'retina'
```

In [4]:

```
#read data
sparkcontext = SparkSession.builder.master("spark://master:7077").appName("cali_
housing_price").getOrCreate()
```

In [5]:

```
HOUSING_DATA = '/user/curtis0982/data/cal_housing.data'
```

In [6]:

```
schema = StructType([
    StructField("long", FloatType(), nullable=True),
    StructField("lat", FloatType(), nullable=True),
    StructField("medage", FloatType(), nullable=True),
    StructField("totrooms", FloatType(), nullable=True),
    StructField("totbdrms", FloatType(), nullable=True),
    StructField("pop", FloatType(), nullable=True),
    StructField("houshlds", FloatType(), nullable=True),
    StructField("medinc", FloatType(), nullable=True),
    StructField("medhv", FloatType(), nullable=True)]
)
```

2. Creating the Spark Session

In [7]:

```
# Visualization

#多行輸出


#圖像細節


#提高圖的解析度
```

In [8]:

```
# setting random seed for notebook reproducability
```

3. Load The Data From a File Into a Dataframe

In [9]:

```
#記得常用的cache()
sdf = spark.read.csv(path =HOUSING_DATA ,schema=schema).cache()
```

In [10]:

```
sdf.printSchema()
```

```
root
 |-- long: float (nullable = true)
 |-- lat: float (nullable = true)
 |-- medage: float (nullable = true)
 |-- totrooms: float (nullable = true)
 |-- totbdrms: float (nullable = true)
 |-- pop: float (nullable = true)
 |-- houshlds: float (nullable = true)
 |-- medinc: float (nullable = true)
 |-- medhv: float (nullable = true)
```

In [ ]:

```
for i in range(len(sdf.columns)):
    print("col name:",sdf[i].names,"\tnull count:",sdf.filter(sdf[i].isNull()).c
ount())
```

In [ ]:

```
sdf.select('medhv').describe().show()
```

4. Data Exploration

In [ ]:

```
#4分位數
quantile = sdf.approxQuantile(['medhv'], [0.25, 0.5, 0.75], 0)
print("quantile_25 ", quantile[0][0],
'\nquantile_50', quantile[0][1],
'\nquantile_75', quantile[0][2])

#觀察發現中位數比平均數低
```

In [ ]:

```
#資料很少的畫圖法
plt.figure(figsize=(25,10))
plt.title('medhv distribution')
sns.distplot(sdf.select('medhv').toPandas())
plt.show()

#資料很多要歸為bin
```

In [ ]:

```
sdf.select("medhv").show(10)
```

In [ ]:

```
from pyspark.ml.feature import Bucketizer
from pyspark.sql.functions import udf

var = "medhv"
# create the split list ranging from 0 to  21, interval of 0.5
split_list = [float(i) for i in np.arange(0,510000,10)]
# initialize buketizer
bucketizer = Bucketizer(splits=split_list,inputCol=var, outputCol="buckets")
# transform
sdf_buck = bucketizer.setHandleInvalid("keep").transform(sdf.select(var).dropna
())

# the "buckets" column gives the bucket rank, not the acctual bucket value(rang
e),
# use dictionary to match bucket rank and bucket value
bucket_names = dict(zip([float(i) for i in range(len(split_list[1:]))],split_lis
t[1:]))
# user defined function to update the data frame with the bucket value
udf_foo = udf(lambda x: bucket_names[x], FloatType())
```

In [ ]:

```
#seaborn字體大小
sns.set(font_scale=2)

bins = sdf_buck.withColumn("bins", udf_foo("buckets")).groupBy("bins").count().s
ort("bins")
plt.figure(figsize=(30,10))
plt.title('medhv distribution')
sns.distplot(bins.select('bins').toPandas())
plt.show()
```

Most of the residents are either in their youth or they settle here during their senior years. Some data are showing median age < 10 which seems to be out of place.

In [ ]:

```
print(sdf.groupBy("medage").agg(F.count("*").alias("freq")).orderBy(F.desc("fre
q")).show(5))
```

In [ ]:

```
#alias 和 orderBy不對count(無參數)作用
#所以要用agg(F.count(*).alias("count_1").orderBy(F.desc(count_1)))
sdf.groupby("medage").count().alias("medage_freq").show(20)
```

In [ ]:

```
#sdf.toPandas().plot(kind='bar',x='medage',figsize=(20, 6),fontsize=12)
```

In [ ]:

```
sdf = sdf.withColumn("medhv",sdf.medhv/100000)
```

4.2 Summary Statistics:

Spark DataFrames include some built-in functions for statistical processing. The describe() function performs summary statistics calculations on all numeric columns and returns them as a DataFrame.

Look at the minimum and maximum values of all the (numerical) attributes. We see that multiple attributes have a wide range of values: we will need to normalize your dataset.

5. Data Preprocessing

With all this information that we gathered from our small exploratory data analysis, we know enough to preprocess our data to feed it to the model.

- we shouldn't care about missing values; all zero values have been excluded from the data set.
- We should probably standardize our data, as we have seen that the range of minimum and maximum values is quite big.
- There are possibly some additional attributes that we could add, such as a feature that registers the number of bedrooms per room or the rooms per household.
- Our dependent variable is also quite big; To make our life easier, we'll have to adjust the values slightly.

5.1 Preprocessing The Target Values

First, let's start with the `medianHouseValue`, our dependent variable. To facilitate our working with the target values, we will express the house values in units of 100,000. That means that a target such as `452600.000000` should become `4.526`:

We can clearly see that the values have been adjusted correctly when we look at the result of the show() method:

6. Feature Engineering

Now that we have adjusted the values in medianHouseValue, we will now add the following columns to the data set:

- Rooms per household which refers to the number of rooms in households per block group;
- Population per household, which basically gives us an indication of how many people live in households per block group; And
- Bedrooms per room which will give us an idea about how many rooms are bedrooms per block group;

As we're working with DataFrames, we can best use the `select()` method to select the columns that we're going to be working with, namely `totalRooms`, `households`, and `population`. Additionally, we have to indicate that we're working with columns by adding the `col()` function to our code. Otherwise, we won't be able to do element-wise operations like the division that we have in mind for these three variables:

6.1 Feature Extraction

Now that we have re-ordered the data, we're ready to normalize the data. We will choose the features to be normalized.

Use a VectorAssembler to put features into a feature vector column:

All the features have transformed into a Dense Vector.

6.2 Standardization

Next, we can finally scale the data using `StandardScaler`. The input columns are the `features`, and the output column with the rescaled that will be included in the scaled_df will be named `"features_scaled"`:

In [ ]:

```
fea = ['long','lat','medage','totrooms','totbdrms','pop','houshlds','medinc']
```

In [ ]:

```
va_tool = VectorAssembler(inputCols = fea, outputCol='va_vector')
```

In [ ]:

```
sdf_vaed = va_tool.transform(sdf)
```

In [ ]:

```
ss_tool = StandardScaler(inputCol = 'va_vector', outputCol='va_vector_ssed')
sdf_4ml = ss_tool.fit(sdf_vaed).transform(sdf_vaed)
```

In [ ]:

```
# Add the new columns to `df`
sdf = (sdf.withColumn("rmsperhh", F.round(col("totrooms")/col("houshlds"), 2))
                        .withColumn("popperhh", F.round(col("pop")/col("houshlds"
), 2))
                        .withColumn("bdrmsperrm", F.round(col("totbdrms")/col("to
trooms"), 2)))
```

7. Building A Machine Learning Model With Spark ML

With all the preprocessing done, it's finally time to start building our Linear Regression model! Just like always, we first need to split the data into training and test sets. Luckily, this is no issue with the `randomSplit()` method:

In [ ]:

```
train_sdf,test_sdf = sdf_4ml.randomSplit([0.8,0.2])
regressor = LinearRegression(featuresCol='va_vector_ssed',labelCol='medhv')
regressor = regressor.fit(train_sdf)
```

We pass in a list with two numbers that represent the size that we want your training and test sets to have and a seed, which is needed for reproducibility reasons.

Note that the argument `elasticNetParam` corresponds to α or the vertical intercept and that the `regParam` or the regularization paramater corresponds to λ.

8. Evaluating the Model

With our model in place, we can generate predictions for our test data: use the `transform()` method to predict the labels for our `test_data`. Then, we can use RDD operations to extract the predictions as well as the true labels from the DataFrame.

8.1 Inspect the Model Co-efficients

8.2 Generating Predictions

8.3 Inspect the Metrics

Looking at predicted values is one thing, but another and better thing is looking at some metrics to get a better idea of how good your model actually is.

Using the `LinearRegressionModel.summary` attribute:

Next, we can also use the `summary` attribute to pull up the `rootMeanSquaredError` and the `r2`.

- The RMSE measures how much error there is between two datasets comparing a predicted value and an observed or known value. The smaller an RMSE value, the closer predicted and observed values are.
- The R2 ("R squared") or the coefficient of determination is a measure that shows how close the data are to the fitted regression line. This score will always be between 0 and a 100% (or 0 to 1 in this case), where 0% indicates that the model explains none of the variability of the response data around its mean, and 100% indicates the opposite: it explains all the variability. That means that, in general, the higher the R-squared, the better the model fits our data.

Using the RegressionEvaluator from pyspark.ml package:

Using the RegressionMetrics from pyspark.mllib package:

In [ ]:

```
pred = regressor.transform(test_sdf)
```

In [ ]:

```
regressor.summary.rootMeanSquaredError
```

In [ ]:

```
regressor.summary.r2
```