NYC Property Price - Regression Models - FIN

March 31, 2020

NYC Property Price Model - Linear Regresssion and RandomTree Regression Input dataset: https://www.kaggle.com/new-york-city/nyc-property-sales

Using the Apache Spark ML pipeline (python), build a model to predict price of property on the available features for New York City.

Load Data

[3]:
```
# File location and type
file_location = "/FileStore/tables/nyc_rolling_sales-3aec7.csv"
file_type = "csv"

# CSV options
infer_schema = "true"
first_row_is_header = "true"
delimiter = ","

# The applied options are for CSV files. For other file types, these will be␣
 ↪ignored.
df = spark.read.format(file_type) \
  .option("inferSchema", infer_schema) \
  .option("header", first_row_is_header) \
  .option("sep", delimiter) \
  .load(file_location)

display(df)
```

Data Description Context

This dataset is a record of every building or building unit (apartment, etc.) sold in the New York City property market over a 12-month period (2016 & 2017).

Content

This dataset contains the location, address, type, sale price, and sale date of building units sold. A reference on the trickier fields:

BOROUGH: A digit code for the borough the property is located in; in order these are Manhattan (1), Bronx (2), Brooklyn (3), Queens (4), and Staten Island (5). BLOCK; LOT: The combination of borough, block, and lot forms a unique key for property in New York City. Commonly called a BBL. BUILDING CLASS AT PRESENT and BUILDING CLASS AT TIME OF SALE: The type of building at various points in time. See the glossary linked to below. For further reference on individual fields see the Glossary of Terms. For the building classification codes see the Building Classifications Glossary: https://www1.nyc.gov/assets/finance/downloads/pdf/07pdf/glossary_rsf071607.pdf

Note that because this is a financial transaction dataset, there are some points that need to be kept in mind:

- Many sales occur with a nonsensically small dollar amount: $0 most commonly. These sales are actually transfers of deeds between parties: for example, parents transferring ownership to their home to a child after moving out for retirement.
- This dataset uses the financial definition of a building/building unit, for tax purposes. In case a single entity owns the building in question, a sale covers the value of the entire building. In case a building is owned piecemeal by its residents (a condominium), a sale refers to a single apartment (or group of apartments) owned by some individual.

Acknowledgements

This dataset is a concatenated and slightly cleaned-up version of the New York City Department of Finance's Rolling Sales dataset. ###### Inspiration What can you discover about New York City real estate by looking at a year's worth of raw transaction records? Can you spot trends in the market, or build a model that predicts sale value in the future?

[5]:
```
df.printSchema()
```

[6]:
```
# descriptions of data
df.describe().show()
```

Preliminary Data Summary

- There are 84548 rows in the data set
- According to Kaggle data summary, ease_ment is empty and the first column is irrelevant
- Some of the features in the dataset are uploaded with incorrect datatype inferred by databricks and needs to be updated, example price appears as string but should be an integer.
- If there are duplicates, the rows need to be deleted from dataset
- Based on the description provided, the price column needs to be reviewed and several rows with no price or $0 price for property will need to be deleted.
- There are also several high value properties which is the sale of entire building and price for these sales is very high, in billions of dollars, this data will also need to be deleted to remove outliers.
- Other data in the dataset needs to be explored

Dataset Updates and Analysis

[9]:
```
# remove columns ease_ment as it is empty and _c0 is not useful

df = df.drop("ease_ment", "_c0")
display(df)
```

[10]:
```
# remove duplicate rows if any from dataset

from pyspark.sql import Row
```

```
df0 = df.dropDuplicates()
df0.count()
```

- Removed 765 duplicate rows from the dataset and updated the dataframe

Next Data Updates

- Update the datatype of various columns that are incorrect
- Update borough columns so it is easy to identify them for analysis
- Review and conduct various data analysis to determine if there is any interesting information
- Review and update price column and prepare for linear model creation
- Review other columns and make adjustments as needed to dataset

[13]:
```
# change datatype of some features

from pyspark.sql.types import StringType, IntegerType, DateType, FloatType
from pyspark.sql.functions import unix_timestamp
from pyspark.sql.functions import to_date, col

df0 = df0.withColumn("borough", df0["borough"].cast(StringType()))
df0 = df0.withColumn("price", df0["price"].cast(IntegerType()))
df0 = df0.withColumn("land_sqft", df0["land_sqft"].cast(FloatType()))
df0 = df0.withColumn("gross_sqft", df0["gross_sqft"].cast(FloatType()))
df0 = df0.withColumn("price", df0["price"].cast(IntegerType()))
df0 = df0.withColumn("sale_date", to_date(col("sale_date"),"yyyy-MM-DD").
 ↪cast(DateType()))
df0.printSchema()
```

[14]:
```
# update the column value based on data description

from pyspark.sql.functions import when

df1 = df0.withColumn("borough", when(df0["borough"] == '1', 'Manhattan').
 ↪otherwise(df0['borough']))
df2 = df1.withColumn("borough", when(df1["borough"] == '2', 'Bronx').
 ↪otherwise(df1['borough']))
df3 = df2.withColumn("borough", when(df2["borough"] == '3', 'Brooklyn').
 ↪otherwise(df2['borough']))
df4 = df3.withColumn("borough", when(df3["borough"] == '4', 'Queens').
 ↪otherwise(df3['borough']))
df5 = df4.withColumn("borough", when(df4["borough"] == '5', 'Staten Island').
 ↪otherwise(df4['borough']))
```

[15]:
```
# register spark SQL tables for analysis purposes

df5.createOrReplaceTempView("df_nyc")
```

[16]:
```
%sql
Select count(*) from df_nyc
```

[17]:
```
%sql
Select borough, count(*) from df_nyc group by borough;
```

- Queens seems to have majority of the sales followed by Brooklyn and Manhattan.

[19]:
```
%sql
Select tax_class, count(*) from df_nyc group by tax_class;
```

- Most of the data consists of sales for tax_class 1 and 2
- tax_class 3 seems to be absent from the dataset
- Below is the description of the Tax Classes from: https://www1.nyc.gov/assets/finance/downloads/pdf/(

Class 1: Includes most residential property of up to three units (such as one-, two-, and three-family homes and small stores or offices with one or two attached apartments), vacant land that is zoned for residential use, and most condominiums that are not more than three stories.

Class 2: Includes all other property that is primarily residential, such as cooperatives and condominiums.

Class 3: Includes property with equipment owned by a gas, telephone or electric company.

Class 4: Includes all other properties not included in class 1, 2, and 3, such as offices, factories, warehouses, garage buildings, etc.

[21]:
```
%sql
Select sale_date, count(*) from df_nyc group by sale_date;
```

- Based on the above plots, there is no specific seasonal pattern in the sale_date other than a weekly pattern.
- On Sunday there are no property sales in NYC as expected. As the week progresses, sales of property in NYC increase and peaks on Wednesday, Thursday, Friday based on visual review of plot above only.

Price

- Review and update dataset using price column
- This column is the label for regression model and so must be scrutinized and dataset adjusted

[24]:
```
%sql
Select price, count(*) from df_nyc group by price;
```

[25]:
```
%sql
Select price, count(*) from df_nyc group by price;
```

[26]:
```
%sql
Select price, count(*) from df_nyc group by price;
```

Based on the above three plots, “Price” data column seems to contain a lot of low values. This was also mentioned in the data description as follows: “Many sales occur with a nonsensically small dollar amount: $0 most commonly. These sales are actually transfers of deeds between parties: for example, parents transferring ownership to their home to a child after moving out for retirement."

Need to remove all null priced properties and then $0 values properties from the dataset.

Based on the BoxPlot and Quantile plots, on the higher side of the scale, there are some significant outlier prices as well that will need to be adjusted.

[28]:
```
# identify all null values for price

from pyspark.sql.functions import isnan
df5.filter(df5["price"].isNull()).count()
```

[29]:
```
# remove 14,177 null values from dataframe

df6 = df5.filter(df5.price.isNotNull())
df6.count()
```

[30]:
```
# review the descriptive stats including min, max for price
# to determine what to additional values will require removal
df6.describe("price").show()
df6.select("price").summary("count", "min", "25%", "75%", "max").show()
```

After removing the null values and based on the descriptive summary of price column, almost 50% of the prices fall between $230,000 and $950,000.

A judgetment call of minimum reasonable price in NYC area needs to be determined and it would be fair to assume that it is $100,000 USD.

On the higher side of the price, there are several outliers but note that in the description it does indicates that the higher price values the value can go as high as say $5,000,000 USD. This would remove any extreme outlier values and preserve majority of the dataset.

[32]:
```
df7 = df6.filter(df6.price>100000)
df8 = df7.filter(df7.price<5000000)
df8.describe("price").show()
df8.select("price").summary("count", "min", "25%", "75%", "max").show()
```

[33]:
```
# register spark SQL tables for analysis purposes

df8.createOrReplaceTempView("df_nyc2")
```

[34]:
```
%sql
Select price, count(*) from df_nyc2 group by price;
```

- After cleaning up the price above, the distribution seems to be normal as per box plot above
- Total dataset is now 54579

Yr_Built

[37]:
```
%sql
Select yr_built, count(*) from df_nyc2 group by yr_built;
```

- Looking at the data for year built, the top years when the properties sold in NYC were constructed as follows:

Year - Bldg Constructed
1910 - 2,131
1920 - 3,802
1925 - 2,760
1930 - 3,176
1940 - 1,607
1950 - 2,072
1960 - 1,756

- Overall, it is evident that the properties in NYC are relatively old in origin but they have been upgraded several times and at the end of the day location, location, location still applies to purchase of property in any city
- Note that the dataset does contain almost 3,692 properties with no information on year building was constructed

[39]:
```
# remove 6,885 value with year_built value of '0'

df9 = df8.filter(df8.yr_built>0)
df9.count()
```

[40]:
```
# add new column age_build using the yr_build column to df

df10 = df9.withColumn("age_built", 2017-col("yr_built"))
```

- After removing the yr_built = 0, the dataset is at 50,887 rows
- Converted the year_built with age_built which will be more useful/meaningful in the model

[42]:
```
# review gross_sqft & land_sqft for null values

from pyspark.sql.functions import isnan
df10.filter((df10["gross_sqft"] == "") | df10["gross_sqft"].isNull() |␣
 ↪isnan(df10["gross_sqft"])).count()
```

[43]:
```
df10.filter((df10["land_sqft"] == "") | df10["land_sqft"].isNull() |␣
 ↪isnan(df10["land_sqft"])).count()
```

- As we want to preserve as much data as possible, both “gross_sqft” and “land_sqft” will need to be imputed to ensure null values are replaced for regression modelling
- Imputation estimator will be used for completing missing values, either using the mean or the median of the columns in which the missing values are located (see below - Regression Modelling, Data Preparation for execution)

[45]:
```
%sql
Select res_unts, count(*) from df_nyc2 group by res_unts;
```

[46]:
```
%sql
Select com_unts, count(*) from df_nyc2 group by com_unts;
```

[47]:
```
%sql
Select tot_unts, count(*) from df_nyc2 group by tot_unts;
```

- There seem to be outliers in residential, commercial and total units data that need to be deleted to ensure a good distribution
- Also the dataset is skewed towards 0, so tot_unts that have 0 value and tot_unts = res_unts + com_unts is incorrect will be removed

[49]:
```
df11 = df10.filter(df10.tot_unts>0)
df12 = df10.filter(df11.tot_unts<500)
df13 = df12.filter(df12.tot_unts == df12.res_unts + df11.com_unts)
df13.count()
```

[50]:
```
df13.createOrReplaceTempView("df_nyc3")
```

[51]:
```
%sql
Select tot_unts, count(*) from df_nyc3 group by tot_unts;
```

- The boxplot of tot_unts now looks more usable with outliers and additional incorrect rows are deleted
- Total rows in dataset is: 50575

[53]:
```
%sql
Select blg_class, count(*) from df_nyc3 group by blg_class;
```

- There doesn't seem to be a need to do anything for blg_class = Building Class Category
- All values are complete and useful

Correlation

[56]:
```
display(df13)
```

- Based on the correlation map above, price is positively correlated with gross square feet and there is some degree of correlation between price and age_built
- The remainder of the features do not have any specific pattern that can be identified visually based on correlation map above

Regression Models

Feature Selection See https://www1.nyc.gov/assets/finance/downloads/pdf/07pdf/glossary_rsf071607.pdf for detailed overview of the features.

For regression modelling not all features provided are useful. So I will be including the following features only:

- "borough" - location can influence the price of property in NYC
- "blg_class" - building class represents the type and size of property (https://www1.nyc.gov/assets/finance/jump/hlpbldgcode.html)

- "res_unts" - number of residential units in the building
- "com_unts" - number of commercial units in the building
- "tot_unts" - number of total units in the building
- "land_sqft", - sqft of land
- "gross_sqft" - gross sqft of property
- "yr_built">"building age" - converted to building age and may influence price
- "price" - label value
- "tax_class" - taxes on the property may infulence decision of purchase

I will not be including the following:

- "neighborhood" - covered with "borough"
- "block" - covered with "borough"
- "lot" - covered with "borough"
- "bldg_class" - covered with "borough"
- "address" - not relevant to pricing
- "apt" - not relevant to pricing
- "zip" - not relevant to pricing
- "tax_class_sale" - covered with tax_class
- "bldg_class_sale" - covered with blg_class (Building Class Category)
- "sale_date" - irrelevant based on plot above

Data Preparation

[61]:
```
# impute gross_sqft and land_sqft to complete the null values
# drop old columns of gross_sqft, land_sqft and yr_built as it was update to␣
 ↪age_built
# dropping all columns that won't be used, "gross_sqft" + "land_sqft" original␣
 ↪columns and "yr_built"

from pyspark.ml.feature import Imputer

imputer = Imputer(inputCols=["gross_sqft", "land_sqft"],␣
 ↪outputCols=["gross_sqft_impt", "land_sqft_impt"])
model = imputer.fit(df13)
df14 = model.transform(df13)
```

```
df15 = df14.drop("neighborhood", "block", "lot", "bldg_class", "address",␣
 ↪"apt", "zip", "tax_class_sale", "bldg_class_sale", "sale_date", "land_sqft",␣
 ↪"gross_sqft", "yr_built")
display(df15)
```

[62]:
```
# convert categorical features into vectors for the model

from pyspark.ml.feature import StringIndexer, OneHotEncoder

indexer1 = StringIndexer(inputCol="borough", outputCol="borough_index")
df16 = indexer1.fit(df15).transform(df15)

encoder1 = OneHotEncoder(dropLast=False, inputCol="borough_index",␣
 ↪outputCol="borough_vec")
df17 = encoder1.transform(df16)

indexer2 = StringIndexer(inputCol="blg_class", outputCol="blg_class_index")
df18 = indexer2.fit(df17).transform(df17)

encoder2 = OneHotEncoder(dropLast=False, inputCol="blg_class_index",␣
 ↪outputCol="blg_class_vec")
df19 = encoder2.transform(df18)

indexer3 = StringIndexer(inputCol="tax_class", outputCol="tax_class_index")
df20 = indexer3.fit(df19).transform(df19)

encoder3 = OneHotEncoder(dropLast=False, inputCol="tax_class_index",␣
 ↪outputCol="tax_class_vec")
df21 = encoder3.transform(df20)

df22 = df21.drop("borough", "blg_class", "tax_class", "borough_index",␣
 ↪"blg_class_index", "tax_class_index")

display(df22)
```

[63]:
```
from pyspark.ml.feature import VectorAssembler

vectorAssembler = VectorAssembler(inputCols = ["res_unts", "com_unts",␣
 ↪"tot_unts", "age_built", "gross_sqft_impt", "land_sqft_impt",␣
 ↪"borough_vec","blg_class_vec", "tax_class_vec"], outputCol = 'features')
df23 = vectorAssembler.transform(df22)
df23 = df23.select(['features', 'price'])
display(df23)
```

[64]:
```
# split the dataframe into 70% training dataframe and 30% testing dataframe

splits = df23.randomSplit([0.7, 0.3])
train_df = splits[0]
```

```
test_df = splits[1]
```

Linear Regression

[66]:
```
from pyspark.ml.regression import LinearRegression

lr = LinearRegression(featuresCol = "features", labelCol="price", maxIter=10,␣
 ↪regParam=0.3, elasticNetParam=0.8)
lr_model = lr.fit(train_df)

print("Coefficients: " + str(lr_model.coefficients))
print("Intercept: " + str(lr_model.intercept))
```

[67]:
```
trainingSummary = lr_model.summary

print("RMSE: %f" % trainingSummary.rootMeanSquaredError)
print("r2: %f" % trainingSummary.r2)
```

[68]:
```
lr_predictions = lr_model.transform(test_df)
lr_predictions.select("prediction", "price", "features").show(5)

from pyspark.ml.evaluation import RegressionEvaluator
lr_evaluator = RegressionEvaluator(predictionCol="prediction",␣
 ↪labelCol="price",metricName="r2")

print("R Squared (R2) on test data = %g" % lr_evaluator.
 ↪evaluate(lr_predictions))
```

[69]:
```
test_result = lr_model.evaluate(test_df)
rmse_lr = test_result.rootMeanSquaredError
print("Root Mean Squared Error (RMSE) on test data = %g" % rmse_lr)
```

Random Tree Regression

[71]:
```
from pyspark.ml.regression import RandomForestRegressor

rf = RandomForestRegressor(featuresCol = "features", labelCol = "price")
rf_model = rf.fit(train_df)

rf_predictions = rf_model.transform(test_df)
rf_predictions.select("prediction", "price", "features").show(5)
```

[72]:
```
rf_evaluator = RegressionEvaluator(labelCol="price",␣
 ↪predictionCol="prediction", metricName="rmse")
rmse_rf = rf_evaluator.evaluate(rf_predictions)

print("Root Mean Squared Error (RMSE) on test data = %g" % rmse_rf)
```

Decision Tree Regression

[74]:
```
from pyspark.ml.regression import DecisionTreeRegressor

dt = DecisionTreeRegressor(featuresCol ="features", labelCol = "price")
dt_model = dt.fit(train_df)

dt_predictions = dt_model.transform(test_df)
dt_evaluator = RegressionEvaluator(
    labelCol="price", predictionCol="prediction", metricName="rmse")

rmse_dt = dt_evaluator.evaluate(dt_predictions)

print("Root Mean Squared Error (RMSE) on test data = %g" % rmse_dt)
```

Gradient-Boosted Tree Regression

[76]:
```
from pyspark.ml.regression import GBTRegressor

gbt = GBTRegressor(featuresCol = "features", labelCol = "price", maxIter=10)
gbt_model = gbt.fit(train_df)

gbt_predictions = gbt_model.transform(test_df)
gbt_predictions.select("prediction", "price", "features").show(5)
```

[77]:
```
gbt_evaluator = RegressionEvaluator(labelCol="price",␣
  ↪predictionCol="prediction", metricName="rmse")
rmse_gbt = gbt_evaluator.evaluate(gbt_predictions)

print("Root Mean Squared Error (RMSE) on test data = %g" % rmse_gbt)
```

[78]:
```
print("SUMMARY")
print()
print("Root Mean Squared Error (RMSE) on test data for Linear Regression = %g"␣
  ↪% rmse_lr)
print("Root Mean Squared Error (RMSE) on test data for Random Tree Regression =␣
  ↪%g" % rmse_rf)
print("Root Mean Squared Error (RMSE) on test data for Decision Tree Regression␣
  ↪= %g" % rmse_dt)
print("Root Mean Squared Error (RMSE) on test data for Gradient Boosted Tree␣
  ↪Regression = %g" % rmse_gbt)
```

Summary Linear Regression - $608K
Random Tree Regression - $587K
Decision Tree Regression - $585K
Gradient Boosted Tree Regression - $568K

The regression model for NYC property data remains signaificantly error prone but there was improvement based on subsequent usage of different statistical models: from Linear Regression $608K to Gradient Boosted Tree Regression - $568K.

If there is to be improvement in the model, further work will be required in the following areas:

- Improvement in the data fields being collect, perhaps number of rooms in the property, etc.
- A lot of data cleansing was required, so perhaps the data can be more cautiously collected or even automated. It might also make sense to separate the data for residential properties from the commercial but this may be difficult as the two types of properties may exist in the same building
- Further data wrangling may definitely be reuquired of existing data
- Other types statistical models can also be tried to improve the modelling results, like classification modelling