NYC Property Sales Prediction

Introduction

This report was produced as a project requirement for the course “3252 Big Data Management Systems & Tools” at the University of Toronto, School of continuing studies. A real dataset of New York City property sale was downloaded from Kaggle - the subject of this data analysis is to generate prediction models using spark framework's community databricks.

The objective of this course project is to design and implement a sales prediction model by performing a regression analysis utilizing all sales information provided by NYC property data.

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.

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?

Table of Content

- Loading Dataset
- Data Inspection & Visualization
- Features Engineering
- Predictive Modeling
- Conclusion

Loading Dataset

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

# CSV options
infer_schema = "false"
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.limit(10))
```

_c0	BOROUGH	NEIGHBORHOOD	BUILDING CLASS CATEGORY	TAX CLASS AT PRESENT	BLOCK	LOT	EASE-MENT	BUILDING CLASS AT PRESENT
4	1	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	2A	392	6		C2
5	1	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	2	399	26		C7

Check the Record and Features Count

```
# no of records
print((df.count(), len(df.columns)))
```

```
(84548, 22)
```

Drop Empty Feature and Index Columns

```
data_pd = df.toPandas()

# Dropping column as it is empty
del data_pd['EASE-MENT']

# Dropping as it looks like an iterator
del data_pd['_c0']
# Dropping Sale Date
del data_pd['SALE DATE']

df = spark.createDataFrame(data_pd)


# confirm columns has been removed
display(df.limit(5))
```

BOROUGH	NEIGHBORHOOD	BUILDING CLASS CATEGORY	TAX CLASS AT PRESENT	BLOCK	LOT	BUILDING CLASS AT PRESENT
1	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	2A	392	6	C2
1	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	2	399	26	C7
1	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	2	399	39	C7

Original Count

```
originalcount = df.count()
print("Original Count: ", originalcount)
```

```
Original Count:  84548
```

Remove Duplicates

```
# Checking for duplicated entries
print("Duplicates were found:", sum(data_pd.duplicated(df.columns)))
```

```
Duplicates were found: 1358
```

```
# Delete the duplicates
data_pd = data_pd.drop_duplicates(df.columns, keep='last')
df = spark.createDataFrame(data_pd)
```

Data Inspection & Visualization

Shape of Data

```
print((df.count(), len(df.columns)))
```

```
(83190, 19)
```

Data Description

```
data_pd.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 83190 entries, 0 to 84547
Data columns (total 19 columns):
BOROUGH                          83190 non-null object
NEIGHBORHOOD                     83190 non-null object
BUILDING CLASS CATEGORY          83190 non-null object
TAX CLASS AT PRESENT             83190 non-null object
BLOCK                            83190 non-null object
LOT                              83190 non-null object
BUILDING CLASS AT PRESENT        83190 non-null object
ADDRESS                          83190 non-null object
APARTMENT NUMBER                 83190 non-null object
ZIP CODE                         83190 non-null object
RESIDENTIAL UNITS                83190 non-null object
COMMERCIAL UNITS                 83190 non-null object
TOTAL UNITS                      83190 non-null object
LAND SQUARE FEET                 83190 non-null object
GROSS SQUARE FEET                83190 non-null object
YEAR BUILT                       83190 non-null object
TAX CLASS AT TIME OF SALE        83190 non-null object
BUILDING CLASS AT TIME OF SALE   83190 non-null object
SALE PRICE                       83190 non-null object
dtypes: object(19)
memory usage: 12.7+ MB
```

Data Schema Info

```
df.printSchema()
```

```
root
 |-- BOROUGH: string (nullable = true)
 |-- NEIGHBORHOOD: string (nullable = true)
 |-- BUILDING CLASS CATEGORY: string (nullable = true)
 |-- TAX CLASS AT PRESENT: string (nullable = true)
 |-- BLOCK: string (nullable = true)
```

```
 |-- LOT: string (nullable = true)
 |-- BUILDING CLASS AT PRESENT: string (nullable = true)
 |-- ADDRESS: string (nullable = true)
 |-- APARTMENT NUMBER: string (nullable = true)
 |-- ZIP CODE: string (nullable = true)
 |-- RESIDENTIAL UNITS: string (nullable = true)
 |-- COMMERCIAL UNITS: string (nullable = true)
 |-- TOTAL UNITS: string (nullable = true)
 |-- LAND SQUARE FEET: string (nullable = true)
 |-- GROSS SQUARE FEET: string (nullable = true)
 |-- YEAR BUILT: string (nullable = true)
 |-- TAX CLASS AT TIME OF SALE: string (nullable = true)
 |-- BUILDING CLASS AT TIME OF SALE: string (nullable = true)
 |-- SALE PRICE: string (nullable = true)
```

Fix Schema

```python
import pandas as pd

# Let's convert some of the columns to numeric datatype
data_pd['LAND SQUARE FEET'] = pd.to_numeric(data_pd['LAND SQUARE FEET'], errors='coerce')
data_pd['GROSS SQUARE FEET']= pd.to_numeric(data_pd['GROSS SQUARE FEET'], errors='coerce')
data_pd['TOTAL UNITS']= pd.to_numeric(data_pd['TOTAL UNITS'], errors='coerce')
data_pd['RESIDENTIAL UNITS']= pd.to_numeric(data_pd['RESIDENTIAL UNITS'], errors='coerce')
data_pd['COMMERCIAL UNITS']= pd.to_numeric(data_pd['COMMERCIAL UNITS'], errors='coerce')
data_pd['COMMERCIAL UNITS']= pd.to_numeric(data_pd['COMMERCIAL UNITS'], errors='coerce')
data_pd['LOT']= pd.to_numeric(data_pd['LOT'], errors='coerce')
data_pd['ZIP CODE']= pd.to_numeric(data_pd['ZIP CODE'], errors='coerce')
data_pd['LOT']= pd.to_numeric(data_pd['LOT'], errors='coerce')
data_pd['BLOCK'] = pd.to_numeric(data_pd['BLOCK'], errors='coerce')
data_pd['YEAR BUILT'] = pd.to_numeric(data_pd['YEAR BUILT'], errors='coerce')
data_pd['SALE PRICE'] = pd.to_numeric(data_pd['SALE PRICE'], errors='coerce')
df = spark.createDataFrame(data_pd)
```

Verify Final Schema

```python
df.printSchema()
```

```
root
 |-- BOROUGH: string (nullable = true)
 |-- NEIGHBORHOOD: string (nullable = true)
 |-- BUILDING CLASS CATEGORY: string (nullable = true)
 |-- TAX CLASS AT PRESENT: string (nullable = true)
 |-- BLOCK: long (nullable = true)
 |-- LOT: long (nullable = true)
 |-- BUILDING CLASS AT PRESENT: string (nullable = true)
 |-- ADDRESS: string (nullable = true)
 |-- APARTMENT NUMBER: string (nullable = true)
 |-- ZIP CODE: long (nullable = true)
 |-- RESIDENTIAL UNITS: long (nullable = true)
 |-- COMMERCIAL UNITS: long (nullable = true)
 |-- TOTAL UNITS: long (nullable = true)
 |-- LAND SQUARE FEET: double (nullable = true)
```

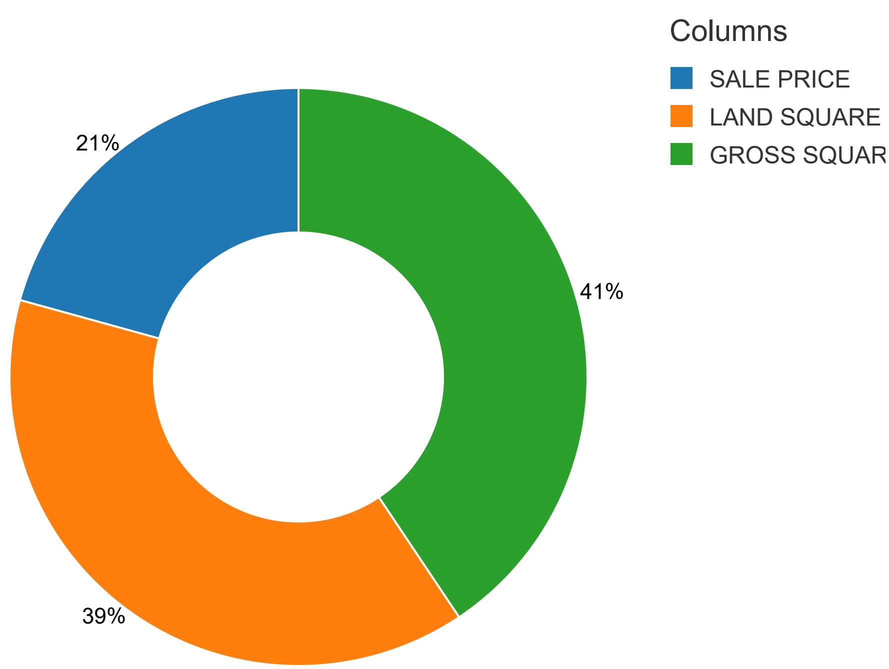
```
 |-- GROSS SQUARE FEET: double (nullable = true)
 |-- YEAR BUILT: long (nullable = true)
 |-- TAX CLASS AT TIME OF SALE: string (nullable = true)
 |-- BUILDING CLASS AT TIME OF SALE: string (nullable = true)
 |-- SALE PRICE: double (nullable = true)
```

Check all Null Values in Columns

```
# checking missing values
data_pd.columns[data_pd.isnull().any()]
```

```
Out[13]: Index(['LAND SQUARE FEET', 'GROSS SQUARE FEET', 'SALE PRICE'], dtype='object')
```

```
miss=data_pd.isnull().sum()/len(data_pd)
miss=miss[miss>0]
miss.sort_values(inplace=True)
miss=miss.to_frame()
miss.columns=['count']
miss.index.names=['Columns']
miss['Columns']=miss.index
display(miss)
```



The "Land Square Feet" and "Gross Square Feet" has around 25K records has null value - pretty large.

The "Sale Price" null records 14K will use as test data

Fix Null Feet Records with Mean

```
data_pd['LAND SQUARE FEET']=data_pd['LAND SQUARE FEET'].fillna(data_pd['LAND SQUARE FEET'].mean())
data_pd['GROSS SQUARE FEET']=data_pd['GROSS SQUARE FEET'].fillna(data_pd['GROSS SQUARE
FEET'].mean())
df = spark.createDataFrame(data_pd)
```

Split data get only Sale Price is not null

```
data_pd = data_pd[~data_pd['SALE PRICE'].isnull()]
data = spark.createDataFrame(data_pd)
df = spark.createDataFrame(data_pd)
```

```
data_pd.shape
```

```
Out[17]: (69281, 19)
```

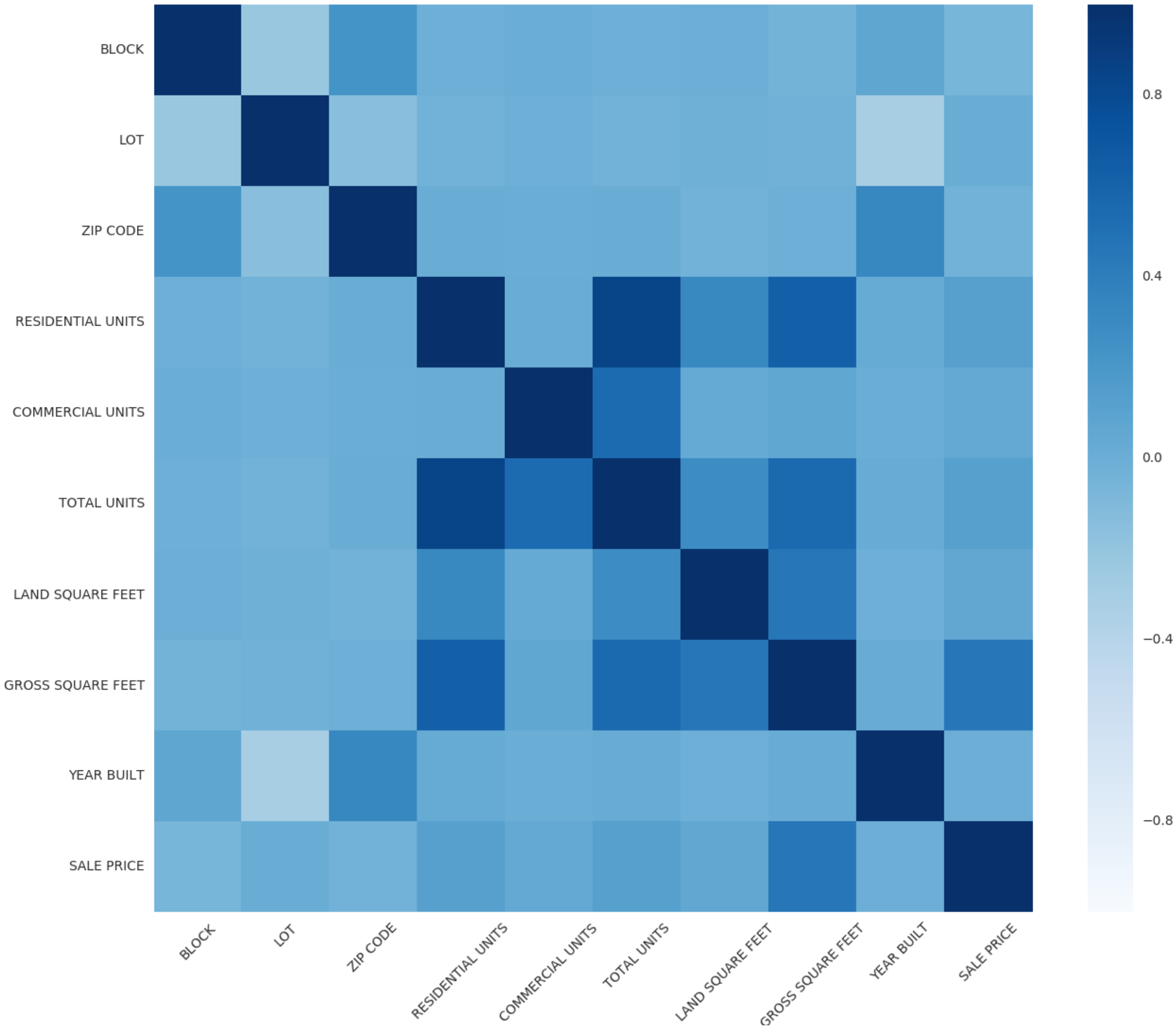
Correlation Between the Features

```
import seaborn as sns
import matplotlib.pyplot as plt

size = (15, 12)

fig, ax = plt.subplots(figsize=size)

corr = data_pd.corr()
ax = sns.heatmap(corr, cmap="Blues")
plt.xticks(rotation=45)
display(fig)
```




```
# numeric correlation
corr['SALE PRICE'].sort_values(ascending=False)
```

```
Out[19]:
SALE PRICE           1.000000
GROSS SQUARE FEET    0.449913
TOTAL UNITS          0.126654
RESIDENTIAL UNITS    0.122566
LAND SQUARE FEET     0.060143
COMMERCIAL UNITS     0.044535
LOT                  0.012266
YEAR BUILT          -0.003779
ZIP CODE            -0.034110
BLOCK               -0.061357
Name: SALE PRICE, dtype: float64
```

```python
import numpy as np

numeric_data=data_pd.select_dtypes(include=[np.number])
numeric_data.describe()
```

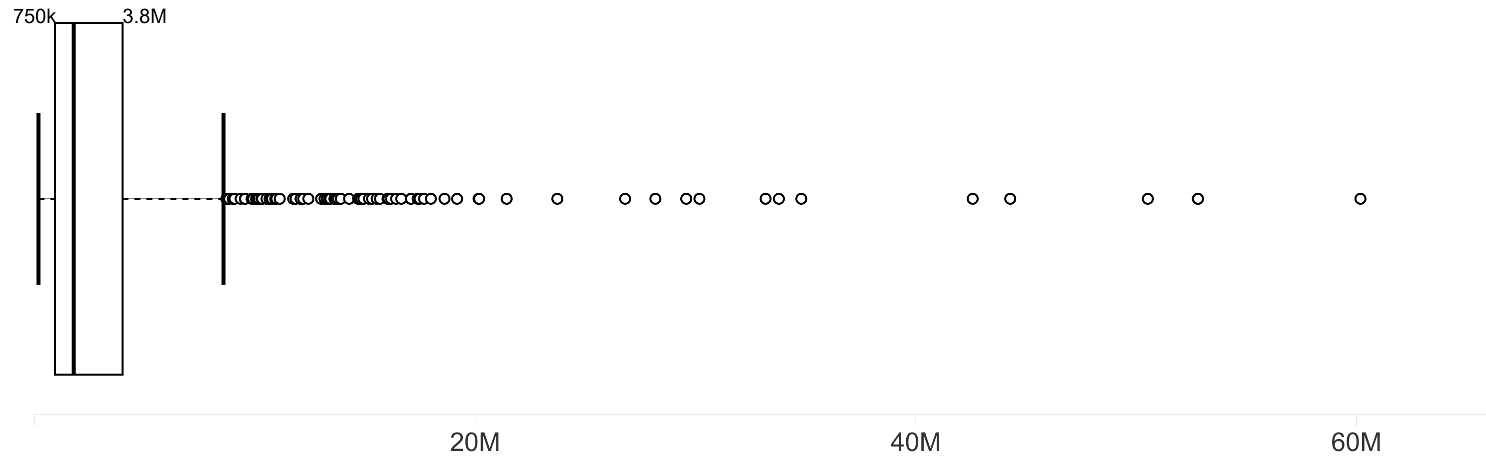
```
Out[20]:
              BLOCK           LOT      ZIP CODE  RESIDENTIAL UNITS  \
count  69281.000000  69281.000000  69281.000000       69281.000000
mean    4200.305437    374.983473  10739.919458           1.870859
std     3434.828427    656.820333   1265.389144          14.317577
min        1.000000      1.000000      0.000000           0.000000
25%     1349.000000     21.000000  10306.000000           0.000000
50%     3377.000000     50.000000  11209.000000           1.000000
75%     6192.000000    879.000000  11249.000000           2.000000
max    16319.000000   9106.000000  11694.000000        1844.000000

       COMMERCIAL UNITS   TOTAL UNITS  LAND SQUARE FEET  GROSS SQUARE FEET  \
count      69281.000000  69281.000000      6.928100e+04       6.928100e+04
mean           0.164244      2.055109      3.643061e+03       3.640300e+03
std            9.018311     17.026435      3.322172e+04       2.427118e+04
min            0.000000      0.000000      0.000000e+00       0.000000e+00
25%            0.000000      0.000000      1.900000e+03       1.268000e+03
50%            0.000000      1.000000      2.970000e+03       2.400000e+03
75%            0.000000      2.000000      3.858418e+03       3.891878e+03
max         2261.000000   2261.000000      4.252327e+06       3.750565e+06

         YEAR BUILT    SALE PRICE
count  69281.000000  6.928100e+04
mean    1800.113451  1.286521e+06
std      519.752668  1.145690e+07
min        0.000000  0.000000e+00
25%     1920.000000  2.350000e+05
50%     1938.000000  5.350000e+05
75%     1965.000000  9.500000e+05
max     2017.000000  2.210000e+09
```

Sale Price

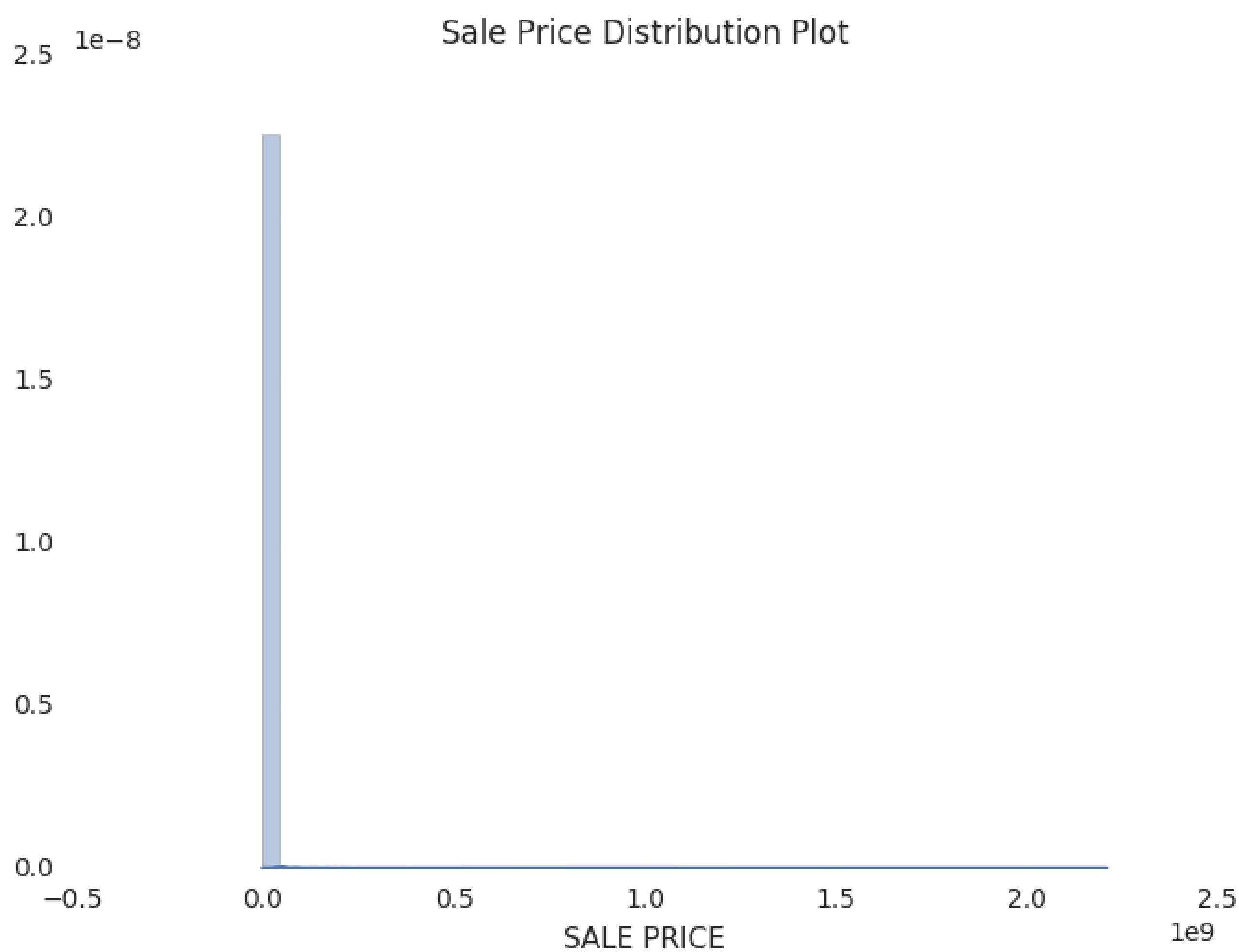
```python
display(data_pd)
```



Showing sample based on the first 1000 rows.

Sale Price Distribution

```
fig, ax = plt.subplots()
ax = sns.distplot(data_pd['SALE PRICE'])
ax.set_title('Sale Price Distribution Plot')
display(fig)
```



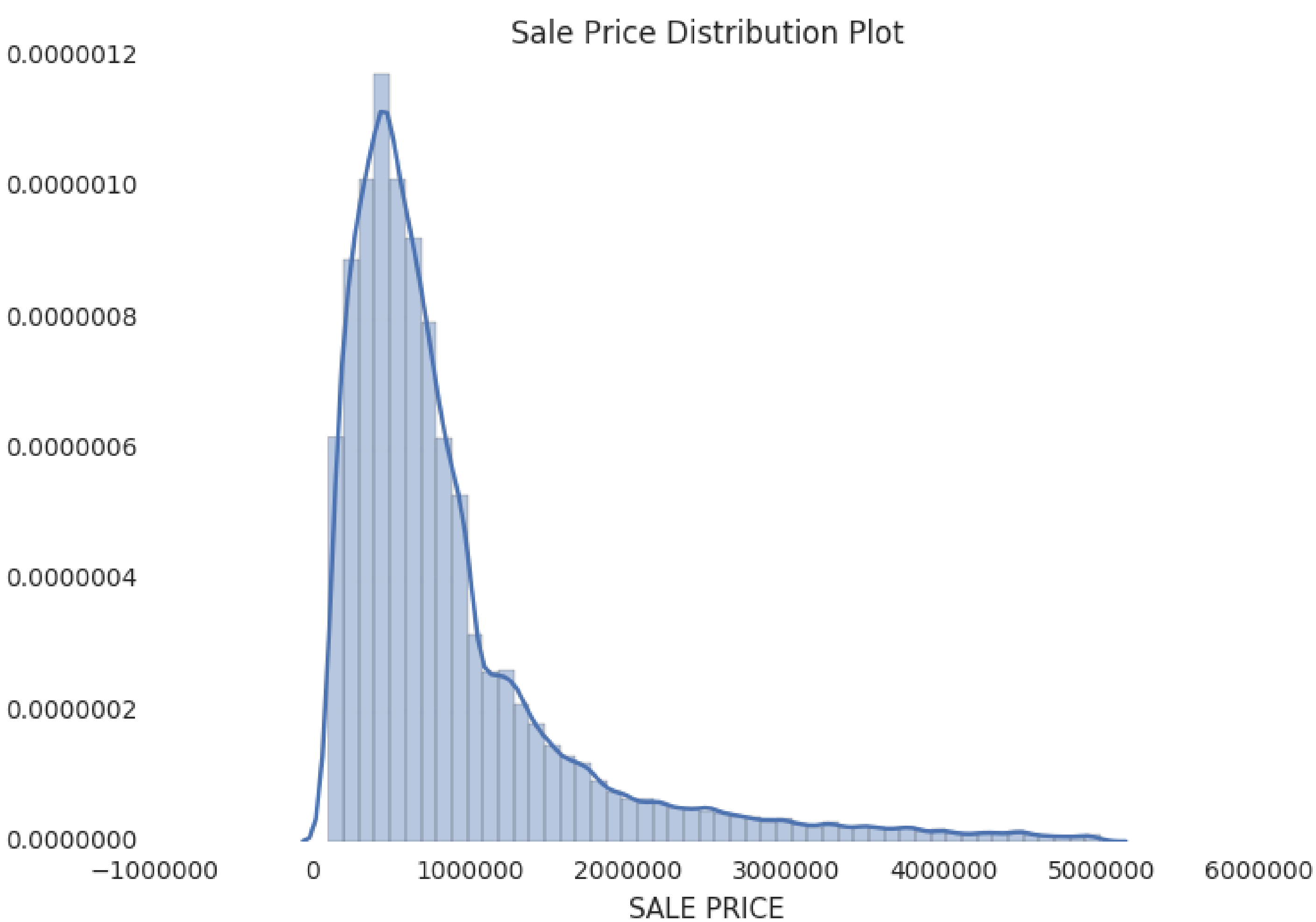
Removing Outside Those Caps

```
from pyspark.sql import functions as F

# Remove observations that fall outside those caps
data = data.filter((F.col('SALE PRICE') > 100000) & (F.col('SALE PRICE') < 5000000))
data_pd = data.toPandas()
```

Sale Price After Changes

```
fig, ax = plt.subplots()
ax = sns.distplot(data_pd['SALE PRICE'])
ax.set_title('Sale Price Distribution Plot')
display(fig)
```



Skewness of Sale Price

```
display(data.select(F.skewness(F.col('SALE PRICE'))))
```

skewness(SALE PRICE)
2.3436810674005444

SALE PRICE is highly right skewed. So, we will log transform it so that it give better results.

```
import numpy as np
sales=np.log(data_pd['SALE PRICE'])
print(sales.skew())
```

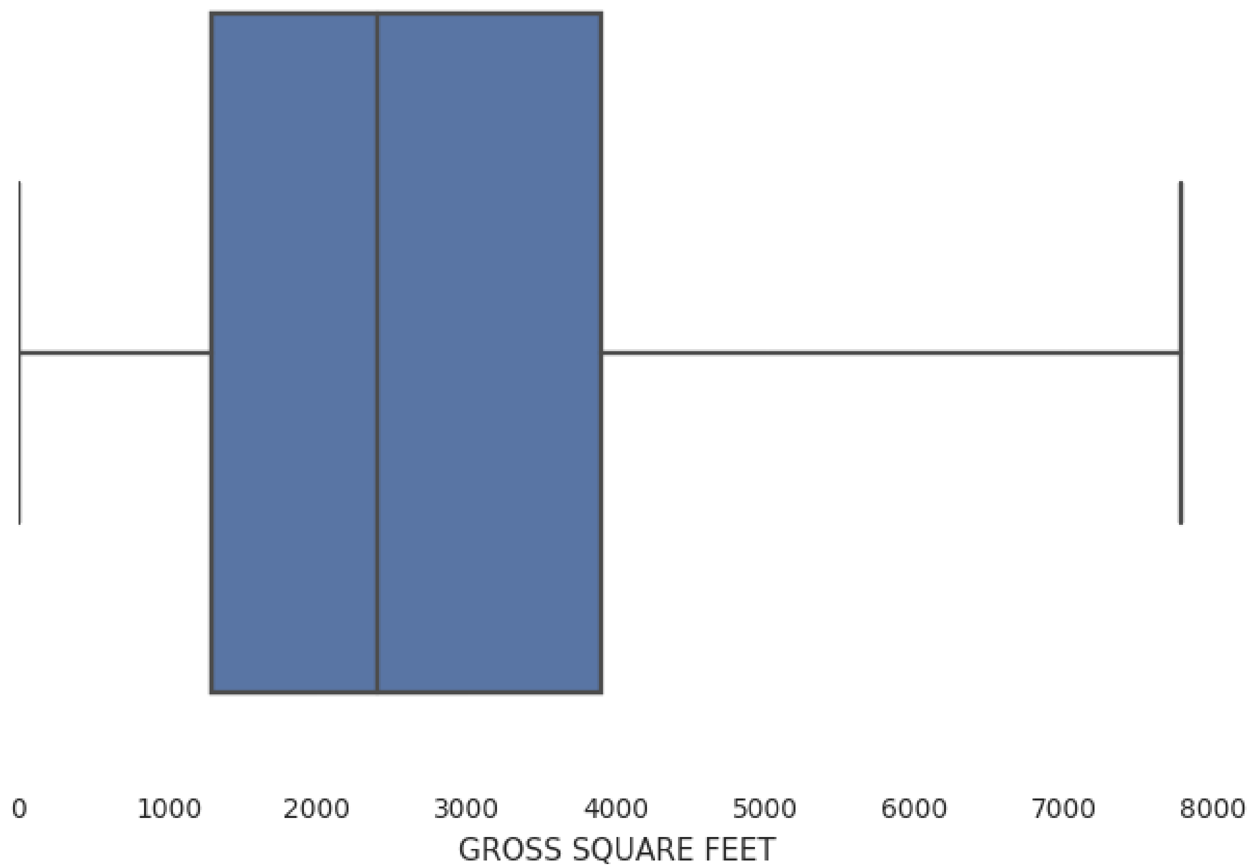
```
0.19896303705
```

```
fig, ax = plt.subplots()
ax = sns.distplot(sales)
display(fig)
```

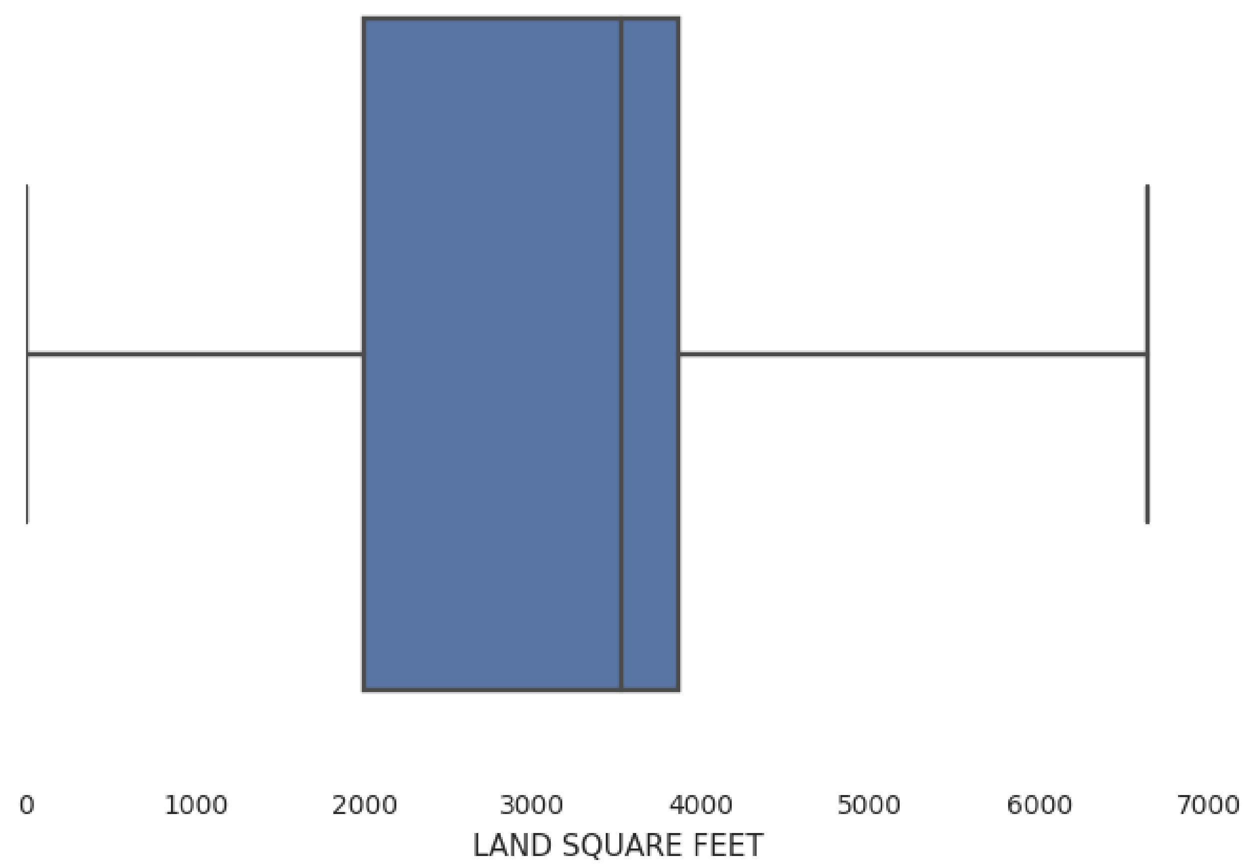
Well now we can see the symmetry and thus it is normalised.

Square Feet

```
fig, ax = plt.subplots()
ax = sns.boxplot(x='GROSS SQUARE FEET', data=data_pd,showfliers=False)
display(fig)
```




```
fig, ax = plt.subplots()
ax = sns.boxplot(x='LAND SQUARE FEET', data=data_pd,showfliers=False)
display(fig)
```

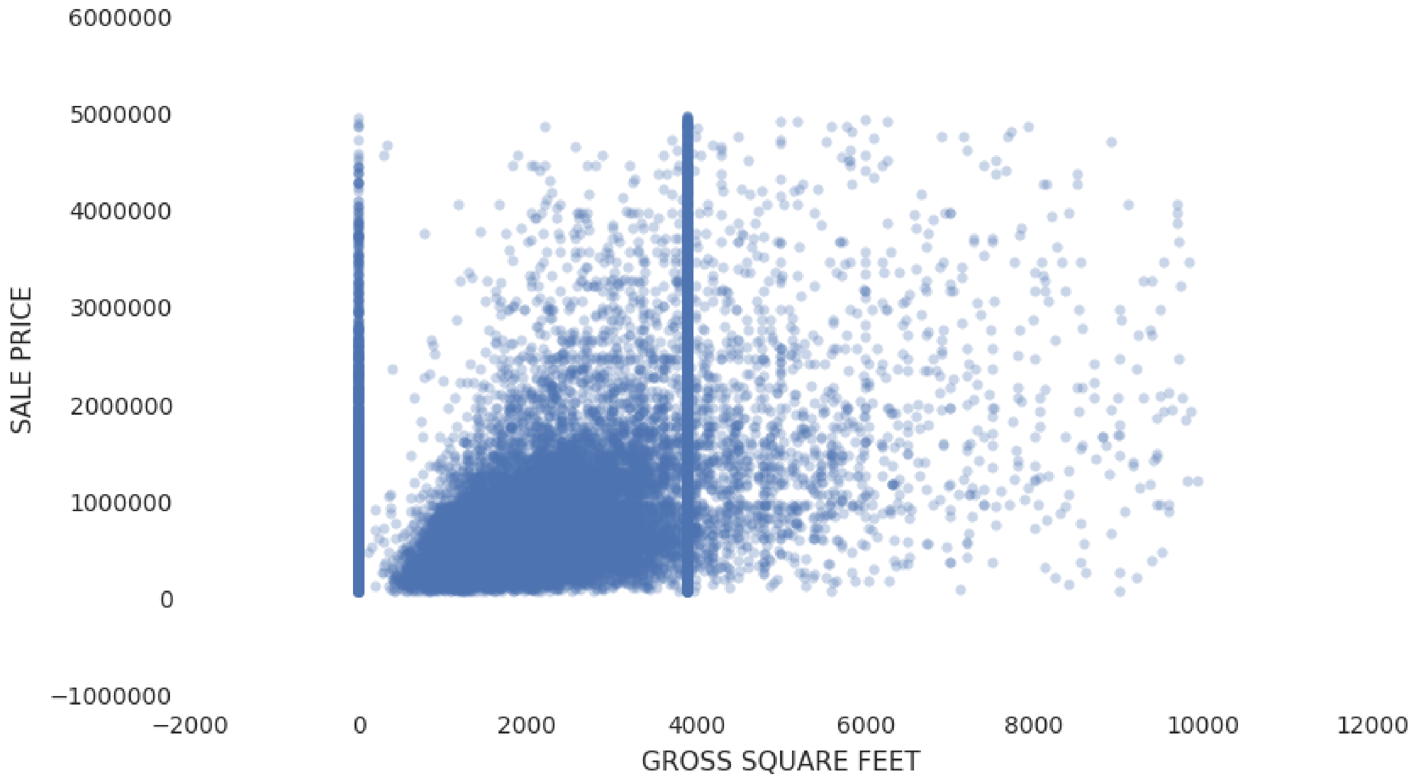


Filter Feet Outliers

```
# Filter feet fields to remove outliers
data = data.filter((F.col('GROSS SQUARE FEET') < 10000))
data = data.filter((F.col('LAND SQUARE FEET') < 10000))
data_pd = data.toPandas()
```

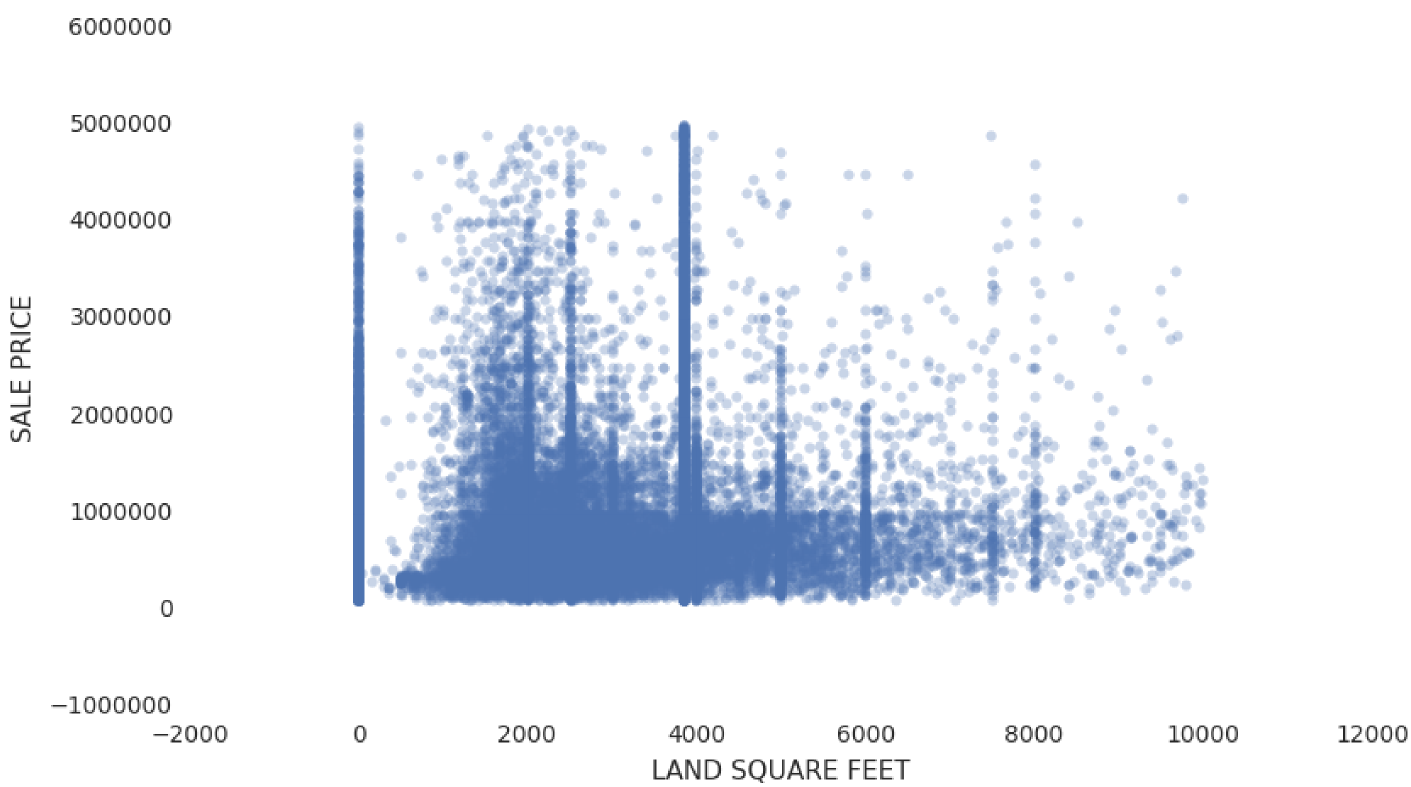
```
size = (9, 5)
fig, ax = plt.subplots(figsize=size)

ax = sns.regplot(x='GROSS SQUARE FEET', y='SALE PRICE', data=data_pd, fit_reg=False, scatter_kws=
{'alpha':0.3})
display(fig)
```




```
size = (9, 5)
fig, ax = plt.subplots(figsize=size)

ax = sns.regplot(x='LAND SQUARE FEET', y='SALE PRICE', data=data_pd, fit_reg=False, scatter_kws=
{'alpha':0.3})
display(fig)
```



Total Units, Commercial Units, Residential Units

```
data_pd[["TOTAL UNITS", "SALE PRICE"]].groupby(['TOTAL UNITS'],
as_index=False).count().sort_values(by='SALE PRICE', ascending=False)
```

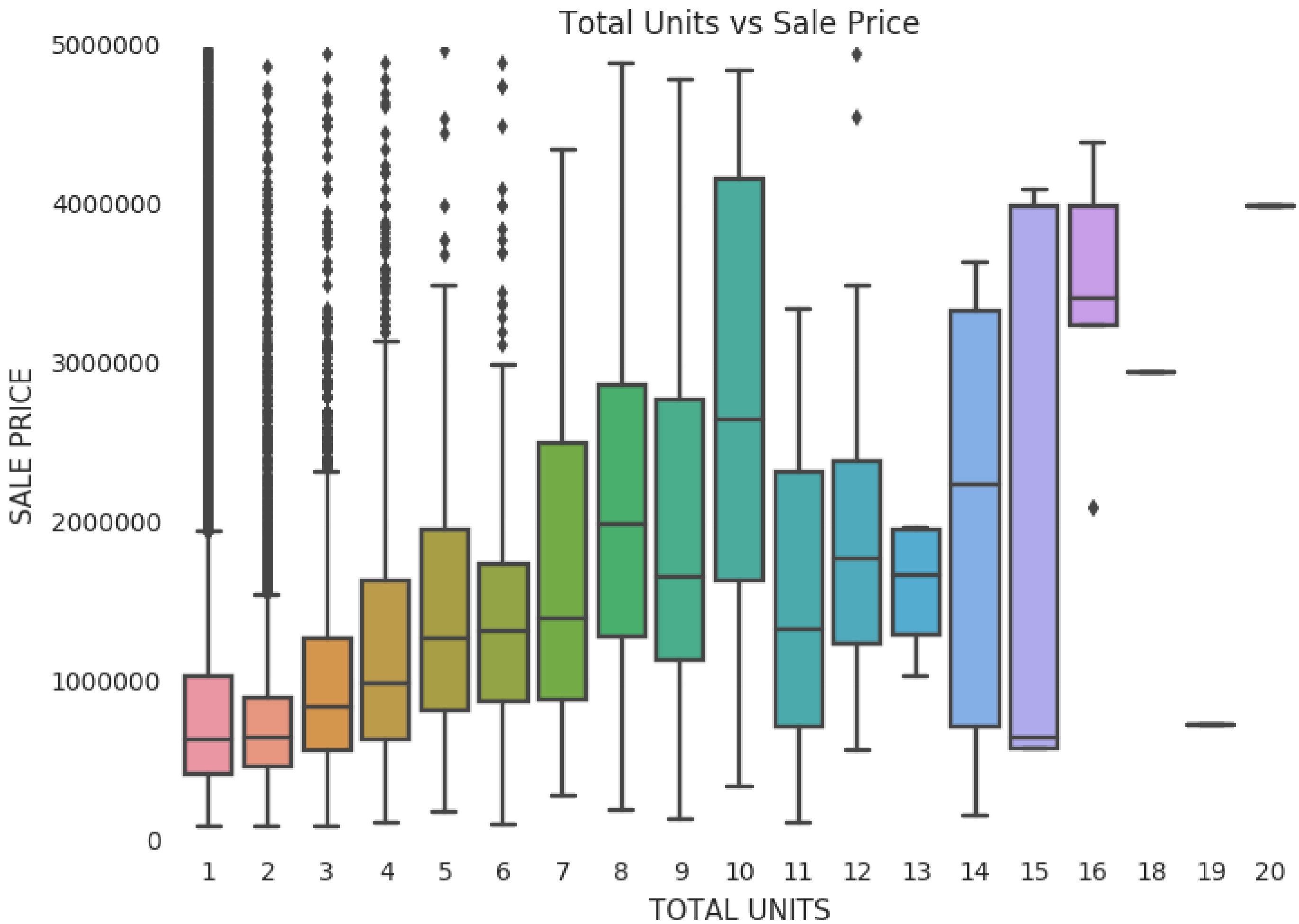
```
Out[33]:
    TOTAL UNITS  SALE PRICE
1             1       24570
0             0       15489
2             2        9473
3             3        2720
4             4         695
6             6         360
5             5         170
8             8         133
7             7          70
9             9          56
10           10          44
12           12          16
11           11           9
16           16           9
14           14           5
15           15           5
13           13           4
17           18           1
18           19           1
19           20           1
20         2261           1
```

Removing rows with TOTAL UNITS == 0 and one outlier with 2261 units

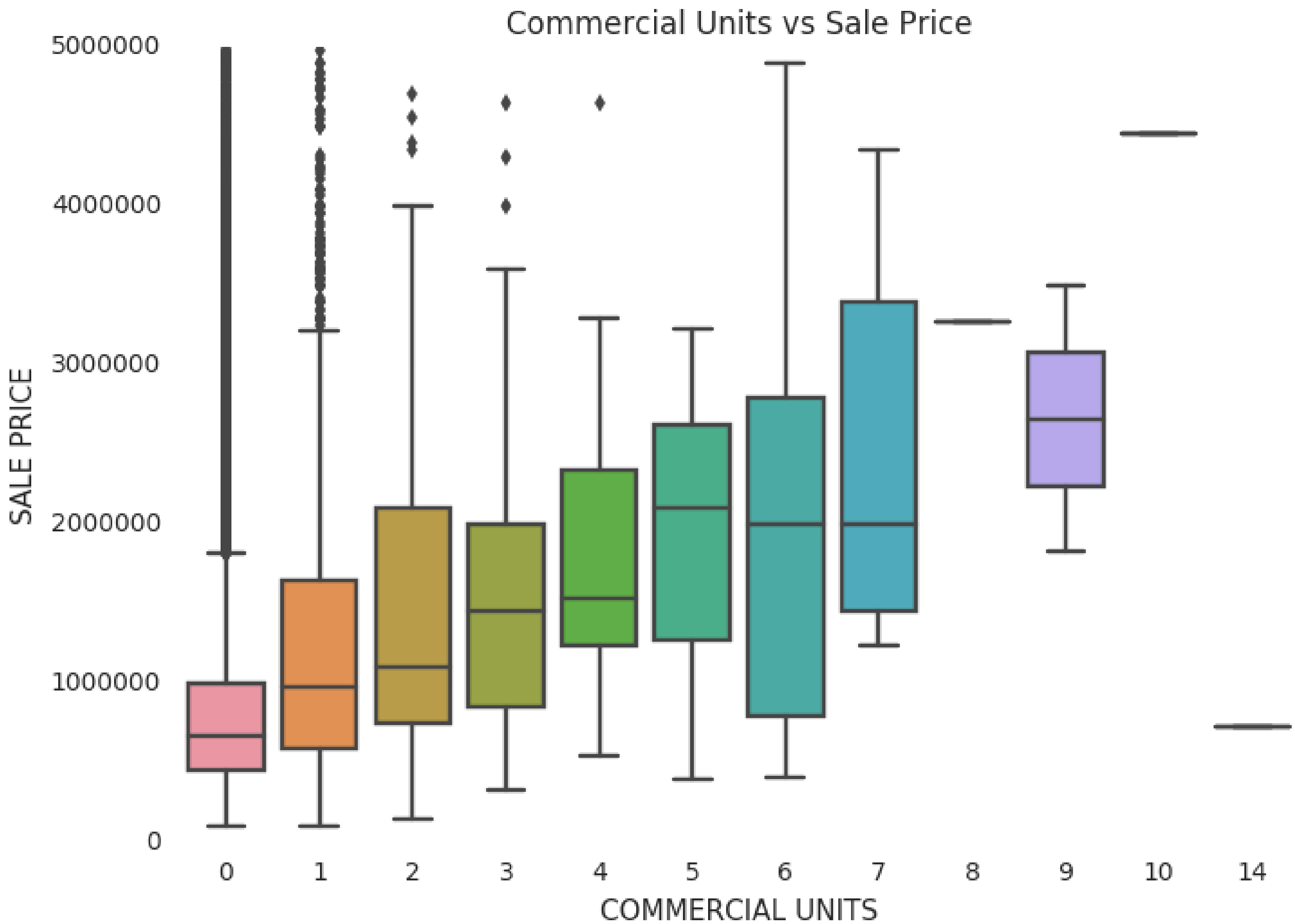
Filter Total Units

```
data = data.filter((F.col('TOTAL UNITS') > 0) & (F.col('TOTAL UNITS') != 2261))
data_pd = data.toPandas()
```

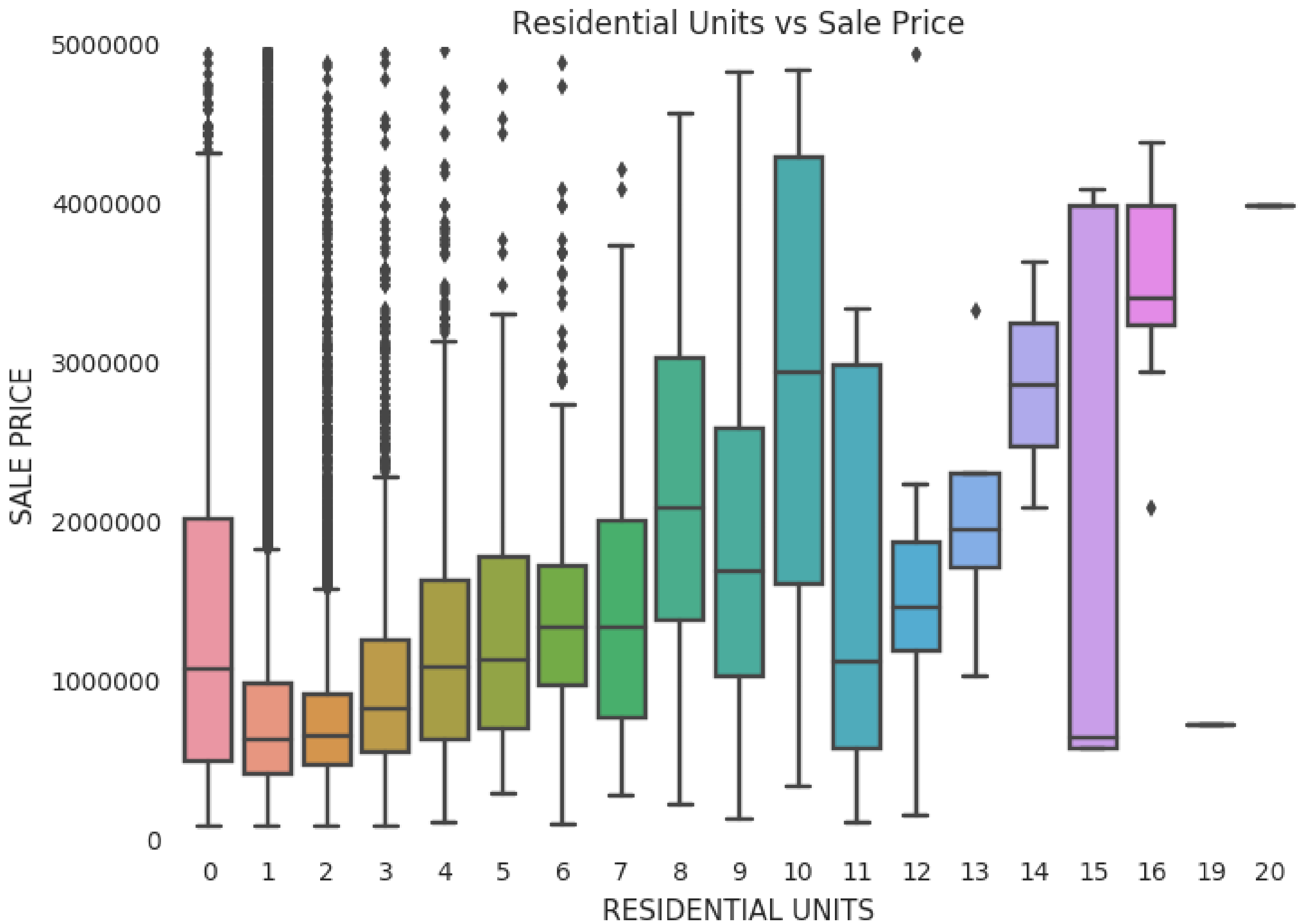
```
fig, ax = plt.subplots()
ax = sns.boxplot(x='TOTAL UNITS', y='SALE PRICE', data=data_pd)
ax.set_title('Total Units vs Sale Price')
display(fig)
```




```
fig, ax = plt.subplots()
ax = sns.boxplot(x='COMMERCIAL UNITS', y='SALE PRICE', data=data_pd)
ax.set_title('Commercial Units vs Sale Price')
display(fig)
```




```
fig, ax = plt.subplots()
ax = sns.boxplot(x='RESIDENTIAL UNITS', y='SALE PRICE', data=data_pd)
ax.set_title('Residential Units vs Sale Price')
display(fig)
```



Top Building Class Category by Average Sale Price

```
display(data.groupBy('BUILDING CLASS CATEGORY').agg(F.mean('SALE PRICE').alias("AVG SALE
PRICE")).orderBy('AVG SALE PRICE', ascending=0))
```

BUILDING CLASS CATEGORY
08 RENTALS - ELEVATOR APARTMENTS
38 ASYLUMS AND HOMES
23 LOFT BUILDINGS
26 OTHER HOTELS
16 CONDOS - 2-10 UNIT WITH COMMERCIAL UNIT
46 CONDO STORE BUILDINGS
32 HOSPITAL AND HEALTH FACILITIES
30 WAREHOUSES
14 RENTALS - 4-10 UNIT
27 FACTORIES
43 CONDO OFFICE BUILDINGS
42 CONDO CULTURAL/MEDICAL/EDUCATIONAL/ETC
07 RENTALS - WALKUP APARTMENTS

Top Tax Class at Present Category by Average Sale Price

```
display(data.groupBy('TAX CLASS AT PRESENT').agg(F.mean('SALE PRICE').alias("SALE PRICE")).orderBy('SALE PRICE', ascending=0))
```

TAX CLASS AT PRESENT
2B
4
2A
2
1C
2C
1B
1
1A

Top Tax Class at Time of Sale Category by Average Sale Pı

```
display(data.groupBy('TAX CLASS AT TIME OF SALE').agg(F.mean('SALE PRICE').alias("SALE PRICE")).orderBy('SALE PRICE', ascending=0))
```

TAX CLASS AT TIME OF SALE
4
2
1

Top Borough Category by Average Sale Price

```
display(data.groupBy('BOROUGH').agg(F.mean('SALE PRICE').alias("SALE PRICE")).orderBy('SALE PRICE', ascending=0))
```

BOROUGH
1
3
4
2
5

Top Neighborhood by Average Sale Price

```
display(data.groupBy('NEIGHBORHOOD').agg(F.mean('SALE PRICE').alias("SALE PRICE")).orderBy('SALE
PRICE', ascending=0))
```

NEIGHBORHOOD
CIVIC CENTER
JAVITS CENTER
BROOKLYN HEIGHTS
LITTLE ITALY
SOHO
FLATIRON
TRIBECA
GREENWICH VILLAGE-CENTRAL
UPPER WEST SIDE (96-116)

```
data = spark.createDataFrame(data_pd)
```

```
display(data)
```

BOROUGH	NEIGHBORHOOD	BUILDING CLASS CATEGORY	TAX CLASS AT PRESENT	BLOCK	LOT	BUILDING CLASS AT PRESENT	ADDRESS
1	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	2B	402	21	C4	154 EAST STREET
1	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	2B	406	32	C4	210 AVEN B

Showing the first 1000 rows.

Features Engineering

Get process data by doing following steps

- Normalization to numeric features
- Index categorial features
- Encode to one hot vectors
- Assemble to a feature vector

```
print((data.count(), len(data.columns)))
```

```
(38342, 19)
```

Normalization (Standard Scaler)

```
# delete some columns
del data_pd['ADDRESS']
del data_pd['APARTMENT NUMBER']
```

```
# transform the numeric features using log(x + 1)
from scipy.stats import skew
skewed = data_pd[numeric_data.columns].apply(lambda x: skew(x.dropna().astype(float)))
skewed = skewed[skewed > 0.75]
skewed = skewed.index
data_pd[skewed] = np.log1p(data_pd[skewed])
```

```
from sklearn.preprocessing import StandardScaler

# apply standard scaler to all numeric feature.
scaler = StandardScaler()
scaler.fit(data_pd[numeric_data.columns])
scaled = scaler.transform(data_pd[numeric_data.columns])

for i, col in enumerate(numeric_data.columns):
       data_pd[col] = scaled[:,i]
```

```
data = spark.createDataFrame(data_pd)
```

```
display(data)
```

BOROUGH	NEIGHBORHOOD	BUILDING CLASS CATEGORY	TAX CLASS AT PRESENT	BLOCK	LOT	BUILDI CLASS AT PRESE
1	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	2B	-1.5528375734493316	-0.8714236123746584	C4

Showing the first 1000 rows.

Apply String Indexer and Hot Encoder to Categorical Featu

```
from pyspark.ml.feature import OneHotEncoderEstimator, StringIndexer
from pyspark.ml import Pipeline, PipelineModel

# Encoded feature text columns
cat_cols = ['BOROUGH', 'NEIGHBORHOOD', 'BUILDING CLASS CATEGORY','TAX CLASS AT PRESENT', 'BUILDING
CLASS AT PRESENT', 'TAX CLASS AT TIME OF SALE', 'BUILDING CLASS AT TIME OF SALE']

stages=[]
for col in cat_cols:
  stringIndexer = StringIndexer().setInputCol(col).setOutputCol(col + "_index")

  encoder =
OneHotEncoderEstimator().setInputCols([stringIndexer.getOutputCol()]).setOutputCols([col +
"_vec"])

  stages += [stringIndexer, encoder]

# encoded features fit into pipeline
pipeline = Pipeline().setStages(stages)

indexer_model = pipeline.fit(data)

# transformed data
data_transformed = indexer_model.transform(data)
```

Check Transformed Data

```
display(data_transformed)
```

BOROUGH	NEIGHBORHOOD	BUILDING CLASS CATEGORY	TAX CLASS AT PRESENT	BLOCK	LOT	BUILDI CLASS AT PRESE
1	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	2B	-1.5528375734493316	-0.8714236123746584	C4
1	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	2B	-1.5450232122589314	-0.6542280352263933	C4

Showing the first 1000 rows.

Assemble to a Feature Vector

```
from pyspark.ml.feature import VectorAssembler

# list of columns to exclude
indexFeatures = [k for k in data_transformed.columns if '_index' in k]

targetCol = ['SALE PRICE']

# remove from features and add vector features
featureCols = [x for x in data_transformed.columns  if x not in (targetCol + cat_cols +
indexFeatures)]

# Create features through vector assembler
vectorAssembler = VectorAssembler().setInputCols(featureCols).setOutputCol("features")

pipelineVectorAssembler = Pipeline().setStages([vectorAssembler])

# Get final data result
result_data = pipelineVectorAssembler.fit(data_transformed).transform(data_transformed)

# visualize data
display(result_data)
```

BOROUGH	NEIGHBORHOOD	BUILDING CLASS CATEGORY	TAX CLASS AT PRESENT	BLOCK	LOT	BUILDI CLASS AT PRESE
1	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	2B	-1.5528375734493316	-0.8714236123746584	C4
1	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	2B	-1.5450232122589314	-0.6542280352263933	C4

Showing the first 1000 rows.

Train/Test Split

```
# Split and cache training and test data
[training, test] = result_data.randomSplit([0.7, 0.3])

# crate cache both training and test data
training.cache()
test.cache()

# print counts so make sure cache works.
print(training.count())
print(test.count())
```

```
26784
11558
```

Predictive Modeling

Linear Regression

```
from pyspark.ml.regression import LinearRegression

# Predict "SALE PRICE" and assign feature column to LinerRegression
lr = LinearRegression().setLabelCol("SALE
PRICE").setFeaturesCol("features").setElasticNetParam(0.5).setRegParam(0.3).setMaxIter(10)

lrModel = lr.fit(training)
```

```
lrholdout = lrModel.transform(test)
```

```
# create a function so every model can use it
from pyspark.mllib.evaluation import RegressionMetrics

def PrintRegressionMatrics(holdout):
  rm = RegressionMetrics(holdout.select("PREDICTION", "SALE PRICE").rdd.map(lambda x: x))

  print("MSE: " + str(rm.meanSquaredError))
  print("MAE: " + str(rm.meanAbsoluteError))
  print("RMSE Squared: " + str(rm.rootMeanSquaredError))
  print("R Squared: " + str(rm.r2))
  print("Explained Variance: " + str(rm.explainedVariance) + "\n")

PrintRegressionMatrics(lrholdout)
```

```
MSE: 0.7408573392691188
MAE: 0.6661243335980449
RMSE Squared: 0.8607307007822591
R Squared: 0.26539753099226493
Explained Variance: 0.08932959895522118
```

Random Forest Regressor

```
from pyspark.ml.regression import RandomForestRegressor
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator
from pyspark.ml.evaluation import RegressionEvaluator

# create model with "SALE PRICE" as target (prediction) variable, use same features variable.
rf = RandomForestRegressor().setLabelCol("SALE PRICE").setFeaturesCol("features")

# create gird with some parameters like maxDepth and num of trees.
paramGrid = ParamGridBuilder().addGrid(rf.maxDepth, [5, 10]).addGrid(rf.numTrees, [20,
60]).build()

# create a pipeline
pipeline = Pipeline().setStages([rf])

# create a crosvalidation with default folds.
cv =
CrossValidator().setEstimator(pipeline).setEstimatorParamMaps(paramGrid).setEvaluator(RegressionEv
aluator().setLabelCol("SALE PRICE"))

# transform tranining data
rfModel = cv.fit(training)
```

```
rfholdout = rfModel.bestModel.transform(test)
```

```
PrintRegressionMatrics(rfholdout)
```

```
MSE: 0.3551006943395346
MAE: 0.4279319956998821
RMSE Squared: 0.5959032592120425
R Squared: 0.6478973305906257
Explained Variance: 0.5520801006520869
```

Gradient-Boosted Tree Regression

```
from pyspark.ml.regression import GBTRegressor

# Train a GBT model.
gbt = GBTRegressor().setLabelCol("SALE PRICE").setFeaturesCol("features").setMaxIter(10)

paramGrid = ParamGridBuilder().addGrid(gbt.maxIter, [5, 10]).build()

# create a pipeline
pipeline = Pipeline().setStages([rf])

# transform tranining data
gbtModel = pipeline.fit(training)
```

```
gbtholdout = gbtModel.transform(test)
```

```
PrintRegressionMatrics(gbtholdout)
```

```
MSE: 0.4979453033631373
MAE: 0.5293566543759911
RMSE Squared: 0.70565239556253
R Squared: 0.5062587223037669
Explained Variance: 0.3585684072584366
```

Decision Tree Regression

```
from pyspark.ml.regression import DecisionTreeRegressor

# Train a GBT model.
dt = DecisionTreeRegressor().setLabelCol("SALE PRICE").setFeaturesCol("features")

# transform tranining data
dtModel = dt.fit(training)
```

```
dtholdout = dtModel.transform(test)
```

```
PrintRegressionMatrics(dtholdout)
```

```
MSE: 0.4930877044193193
MAE: 0.5229372924469503
RMSE Squared: 0.7022020396006546
R Squared: 0.5110753097740327
Explained Variance: 0.5147389037507983
```

Conclusion

In conclusion, only 45% data has been used and lot of data has been removed due to “Sale Price” was null and other factors. The Random Forest Regressor works best for this dataset with **RMSE** score of **0.59** and **R-Square** of **0.64**.