Wake County Dataset

Load Dataset

In [1]:
```
import pandas as pd
import numpy as np

df = pd.read_csv('./WakeCountyHousing.csv')
df.head()
```

Out[1]:

	Real_Estate_Id	Deeded_Acreage	Total_Sale_Price	Total_Sale_Date	Month_Year_of_Sale	Year_of_Sale	Year_Built	Year_Remodeled	Heated_Area	Num_Stories	Design_Style	Bath
0	19	0.21	34500	1/1/1974	January 1974	1974	1964	1964	1828	One Story	Split level	Ba
1	20	0.46	35500	5/18/1983	May 1983	1983	1970	1970	1240	One Story	Conventional	Ba
2	22	0.46	37500	9/16/2004	September 2004	2004	1900	1900	2261	One Story	Conventional	Ba
3	25	0.96	70000	1/1/1971	January 1971	1971	1971	1971	3770	One Story	Conventional	Othe
4	30	0.47	360000	8/12/2015	August 2015	2015	1946	2017	1789	One Story	Conventional	Ba

Drop Unnecessary Columns

We decided to drop the unnecessary columns for model prediction. Those dropped will not be used for prediction.

In [2]:
```
df.drop(columns=['Deeded_Acreage', 'Total_Sale_Date', 'Month_Year_of_Sale'], inplace=True)
df.head()
```

Out[2]:

	Real_Estate_Id	Total_Sale_Price	Year_of_Sale	Year_Built	Year_Remodeled	Heated_Area	Num_Stories	Design_Style	Bath	Utilities	Physical_City	Physical_Zip
0	19	34500	1974	1964	1964	1828	One Story	Split level	2 Bath	ALL	Raleigh	27610.0
1	20	35500	1983	1970	1970	1240	One Story	Conventional	1 Bath	E	Raleigh	27610.0
2	22	37500	2004	1900	1900	2261	One Story	Conventional	2 Bath	WSE	Wendell	27591.0
3	25	70000	1971	1971	1971	3770	One Story	Conventional	Other	WSE	Raleigh	27613.0
4	30	360000	2015	1946	2017	1789	One Story	Conventional	2 Bath	ALL	Raleigh	27607.0

Check for null values

It was important to see what rows had null values. Categorical data, instead of trying to replace drop columns because there is not too many. Will not make a difference for modelling.

In [3]:
```
df.info()
df.isnull()
print(df.isnull().sum())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 300237 entries, 0 to 300291
Data columns (total 12 columns):
 #   Column           Non-Null Count   Dtype
---  ------           --------------   -----
 0   Real_Estate_Id   300292 non-null  int64
 1   Total_Sale_Price  300292 non-null  int64
 2   Year_of_Sale     300292 non-null  int64
 3   Year_Built       300292 non-null  int64
 4   Year_Remodeled   300292 non-null  int64
 5   Heated_Area      300292 non-null  int64
 6   Num_Stories      300292 non-null  object
 7   Design_Style     300292 non-null  object
 8   Bath             300292 non-null  object
 9   Utilities        300324 non-null  object
 10  Physical_City    300183 non-null  object
 11  Physical_Zip     300148 non-null  float64
dtypes: float64(1), int64(6), object(5)
memory usage: 28.2+ MB
Real_Estate_Id      0
Total_Sale_Price    0
Year_of_Sale        0
Year_Built          0
Year_Remodeled      0
Heated_Area         0
Num_Stories         0
Design_Style        0
Bath                0
Utilities        1968
Physical_City     109
Physical_Zip      146
dtype: int64
```

Remove rows with missing values

Fix the missing value by dropping.

In [4]:
```
df = df.dropna(subset=['Bath'])
df = df.dropna(subset=['Utilities'])
df = df.dropna(subset=['Physical_City'])
df = df.dropna(subset=['Physical_Zip'])
print(df.isnull().sum())
```

```
Real_Estate_Id    0
Total_Sale_Price    0
Year_of_Sale        0
Year_Built          0
Year_Remodeled      0
Heated_Area         0
Num_Stories         0
Design_Style        0
Bath                0
Utilities           0
Physical_City       0
Physical_Zip        0
dtype: int64
```

One-Hot Encode

Will first create data frame by one-hot-encoding the categorical data. Will later compare versus ordinal encoding.

In [5]:
```
#use get_dummies
df_hot_encoded = pd.get_dummies(df, drop_first=True)
df_hot_encoded.head()
```

Out[5]:

	Real_Estate_Id	Total_Sale_Price	Year_of_Sale	Year_Built	Year_Remodeled	Heated_Area	Physical_Zip	Num_Stories_Other	Num_Stories_Two Story	Design_Style_Colonial	...	Physical
0	19	34500	1974	1964	1964	1828	27610.0	0	0	0	...	
1	20	35500	1983	1970	1970	1240	27610.0	0	0	0	...	
2	22	37500	2004	1900	1900	2261	27591.0	0	0	0	...	
3	25	70000	1971	1971	1971	3770	27613.0	0	0	0	...	
4	30	360000	2015	1946	2017	1789	27607.0	0	0	0	...	

5 rows × 60 columns

Sale Price Range

In [6]:
```
#check range for regression test
min_value = df_hot_encoded['Total_Sale_Price'].min()
max_value = df_hot_encoded['Total_Sale_Price'].max()
print(max_value-min_value)
```

```
619830
```

Create prediciton of sale price

The prediction will be based off predicting the total sale price of a house. Every column that has not been dropped will go into the model prediction.

In [7]:
```
df_hot_encoded_copy = df_hot_encoded
y = df_hot_encoded_copy['Total_Sale_Price'] #make y the total sale price
del df_hot_encoded_copy['Total_Sale_Price'] #remove from column
y = np.array(y)

X = df_hot_encoded.copy.to_numpy()

print(X.shape)
print(y.shape)

print(300192*.8)
```

```
(300161, 59)
(300161,)
240028.80000000002
```

Train/Test Split

Doing an 80/20 split will provide a good split for model prediction.

In [8]:
```
X_train, X_test, y_train, y_test = X[:240028], X[240028:], y[:240028], y[240028:]
```

Linear Regression

We use linear regression to predict our model and then test how accurate our regression is.

In [9]:
```
from sklearn.linear_model import LinearRegression

#run regression
lin_reg = LinearRegression()
lin_reg.fit(X_train, y_train)
```

Out[9]:
```
LinearRegression()
```

In [10]:
```
predictions = lin_reg.predict(X_test)
```

Mean squared Error

In [11]:
```
from sklearn.metrics import mean_squared_error
predictions = lin_reg.predict(X_test)
lin_mse = mean_squared_error(y_test, predictions)
lin_rmse = np.sqrt(lin_mse)
lin_rmse
```

Out[11]:
```
107948.15122136101
```

R squared

In [12]:
```
from sklearn.metrics import r2_score

r2_score(y_test, predictions)
```

Out[12]:
```
0.6855985788723932
```

Model created a score of 0.60

R2 score perfect = 1.00
Our Model = 0.60

Ordinal Encoding

Now that one-hot-encoding has been done, we will test the data set using ordinal encoding for the categorical data.

In [13]:
```
from sklearn.preprocessing import OrdinalEncoder

#ordinal encoding has to manually be done
#encode_1 = 0
encode_1 = OrdinalEncoder()

#num of stories
stories_reshaped = np.array(df_ordinal['Num_Stories']).reshape(-1, 1)
stories_values = encode_1.fit_transform(stories_reshaped)

#design style
design_reshaped = np.array(df_ordinal['Design_Style']).reshape(-1, 1)
design_values = encode_1.fit_transform(design_reshaped)

#bath
bath_reshaped = np.array(df_ordinal['Bath']).reshape(-1, 1)
bath_values = encode_1.fit_transform(bath_reshaped)

#utilities
util_reshaped = np.array(df_ordinal['Utilities']).reshape(-1, 1)
util_values = encode_1.fit_transform(util_reshaped)
```

In [14]:
```
#check unique values for each to create column headers
uniquevalues4 = df_ordinal['Num_Stories'].unique()
print(uniquevalues4)
uniquevalues3 = df_ordinal['Design_Style'].unique()
print(uniquevalues3)
uniquevalues2 = df_ordinal['Bath'].unique()
print(uniquevalues2)
uniquevalues1 = df_ordinal['Utilities'].unique()
print(uniquevalues1)
```

```
['One Story' 'Two Story' 'Other']
['Split Foyer' 'Conventional' 'Bungalow' 'Townhouse' 'Split Foyer'
 'Contemporary' 'Modular' 'Colonial' 'Conventional' 'Condo' 'Log' 'Duplex'
 'Manuf Multi' 'Cape']
['2 Bath' '1 Bath' 'Other' '3 Bath' '3+ Bath' '1 ½ Bath' '2½ Bath']
['ALL' 'E' 'WSE' 'WSE' 'WE' 'GE' 'S' 'WSG' 'W' 'SSE' 'G' 'SE' 'SG' 'WG'
 'WG']
```

Create Headers

In [15]:
```
#num stories
stories = pd.DataFrame(stories_values, columns=['Stories Value'])

#design style
design_style = pd.DataFrame(design_values, columns=['Design Value'])

#baths
baths = pd.DataFrame(bath_values, columns=['Bath Value'])

#utilities
util = pd.DataFrame(util_values, columns=['Utility Value'])

df_ordinal_cats = pd.concat([stories,design_style,baths,utilities], axis=1)
df_ordinal_cats.head()
```

Out[15]:

	Stories Value	Design Value	Bath Value	Utility Value
0	0.0	12.0	2.0	0.0
1	0.0	4.0	0.0	1.0
2	0.0	4.0	2.0	13.0
3	0.0	4.0	6.0	11.0
4	0.0	4.0	2.0	0.0

In [16]:
```
#dataframe everything else but the ordinal categories
df_others = df[['Real_Estate_Id', 'Total_Sale_Price', 'Year_of_Sale', 'Year_Built', 'Year_Remodeled', 'Heated_Area', 'Physical_Zip']]

#concatin data frame
df_ordinal_final = pd.concat([df_others, df_ordinal_cats], axis=1)
df_ordinal_final.head()
```

Out[16]:

	Real_Estate_Id	Total_Sale_Price	Year_of_Sale	Year_Built	Year_Remodeled	Heated_Area	Physical_Zip	Stories Value	Design Value	Bath Value	Utility Value
0	19.0	34500.0	1974.0	1964.0	1964.0	1828.0	27610.0	0.0	12.0	2.0	0.0
1	20.0	35500.0	1983.0	1970.0	1970.0	1240.0	27610.0	0.0	4.0	0.0	1.0
2	22.0	37500.0	2004.0	1900.0	1900.0	2261.0	27591.0	0.0	4.0	2.0	13.0
3	25.0	70000.0	1971.0	1971.0	1971.0	3770.0	27613.0	0.0	4.0	6.0	11.0
4	30.0	360000.0	2015.0	1946.0	2017.0	1789.0	27607.0	0.0	4.0	2.0	0.0

In [17]:
```
#check for nan
df_ordinal_final.info()
df_ordinal_final.isnull()
#df_ordinal_final.isnull()
df_ordinal_final.replace([np.inf, -np.inf], np.nan, inplace=True)
#dp_ca bad values
print(df_ordinal_final.isnull().sum())
print(df_ordinal_final.isnull().sum().sum())
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 300237 entries, 0 to 300291
Data columns (total 11 columns):
 #   Column          Non-Null Count   Dtype
---  ------          --------------   -----
 0   Real_Estate_Id  300161 non-null  float64
 1   Total_Sale_Price  300161 non-null  float64
 2   Year_of_Sale    300161 non-null  float64
 3   Year_Built      300161 non-null  float64
 4   Year_Remodeled  300161 non-null  float64
 5   Heated_Area     300161 non-null  float64
 6   Physical_Zip    300161 non-null  float64
 7   Stories Value   300161 non-null  float64
 8   Design Value    300161 non-null  float64
 9   Bath Value      300161 non-null  float64
 10  Utility Value   300161 non-null  float64
dtypes: float64(11)
memory usage: 28.2 MB
Real_Estate_Id      0
Total_Sale_Price    0
Year_of_Sale        0
Year_Built          0
Year_Remodeled      0
Heated_Area         0
Physical_Zip        0
Stories Value       0
Design Value        0
Bath Value          0
Utility Value       0
dtype: int64
```

Training Data

Will once again predict the total sale price.

In [18]:
```
y = df_ordinal_final['Total_Sale_Price']
del df_ordinal_final['Total_Sale_Price']
y = np.array(y)

X = df_ordinal_final.to_numpy()

print(X.shape)
print(y.shape)

print(300237*.8)
```

```
(300480, 10)
(300480,)
240189.6
```

In [19]:
```
X_train, X_test, y_train, y_test = X[:240509], X[240509:], y[:240509], y[240509:]
```

In [20]:
```
#ensure there is no nan or inf data points. Problem with ordinal encoding
print(np.any(np.isnan(X_train)))
np.all(np.isfinite(X_train))
```

```
False
True
```

Out[20]:

Linear Regression

Will do linear regression on ordinal encoding based data frame.

In [21]:
```
lin_reg = LinearRegression()
lin_reg.fit(X_train, y_train)
```

Out[21]:
```
LinearRegression()
```

MSE and R2 score check

In [22]:
```
#MSE on dataframe
predictions = lin_reg.predict(X_test)
predictions = lin_reg.predict(X_test)
lin_mse = mean_squared_error(y_test, predictions)
lin_rmse = np.sqrt(lin_mse)
lin_rmse
```

Out[22]:
```
110852.71342328213
```

In [23]:
```
#R2 score check
r2_score(y_test, predictions)
```

Out[23]:
```
0.5835198226046148
```

Model created a score of 0.58

R2 score perfect = 1.00
Our Model = 0.58

Check on Different Encodings

Based on our different models, we found one hot encoding the data provided slightly more accurate predictions. The ordinal encoded data provided very well; however, the one hot gave a slightly less error and a better/closer to 1 value for R2. It is a very slight difference in output, but one hot encoding is slightly better to do in this case.