House Price Prediction

```
# This notebook is going to be focused on solving the problem of predicting house prices for house buyers and house sellers.

A house value is simply more than location and square footage. Like the features that make up a person, an educated party would want to know all aspects that give a house its value.

We are going to take advantage of all of the feature variables available to use and use it to analyze and predict house prices.

We are going to break everything into logical steps that allow us to ensure the cleanest, most realistic data for our model to make accurate predictions from.

1. Load Data and Packages
2. Analyzing the Test Variable (Sale Price)
3. Multivariable Analysis
4. Impute Missing Data and Clean Data
5. Feature Transformation/Engineering
6. Modeling and Predictions
```

About Dataset

```
A benefit to this study is that we can have two clients at the same time! (Think of being a divorce lawyer for both interested parties) However, in this case, we can have both clients with no conflict of interest!

Client Housebuyer: This client wants to find their next dream home with a reasonable price tag. They have their locations of interest ready. Now, they want to know if the house price matches the house value. With this study, they can understand which features (ex. Number of bathrooms, location, etc.) influence the final price of the house. If all matches, they can ensure that they are getting a fair price.

Client Houseseller: Think of the average house-flipper. This client wants to take advantage of the features that influence a house price the most. They typically want to buy a house at a low price and invest on the features that will give the highest return. For example, buying a house at a good location but small square footage. The client will invest on making rooms at a small cost to get a large return.
```

In [2]:

```
import warnings
warnings.filterwarnings('ignore')
import numpy as np
import pandas as pd
```

In [4]:

```
advertising = pd.read_csv(r"C:\Users\Saima Sheikh\Downloads\data.csv")
advertising.head()
```

Out[4]:

	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	condition	sqft_above	sqft_b
0	2014-05-02 00:00:00	313000.0	3.0	1.50	1340	7912	1.5	0	0	3	1340	
1	2014-05-02 00:00:00	2384000.0	5.0	2.50	3650	9050	2.0	0	4	5	3370	
2	2014-05-02 00:00:00	342000.0	3.0	2.00	1930	11947	1.0	0	0	4	1930	
3	2014-05-02 00:00:00	420000.0	3.0	2.25	2000	8030	1.0	0	0	4	1000	
4	2014-05-02 00:00:00	550000.0	4.0	2.50	1940	10500	1.0	0	0	4	1140	

In [5]:

```
advertising.shape
```

Out[5]:

```
(4600, 18)
```

In [6]:

```
advertising.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4600 entries, 0 to 4599
Data columns (total 18 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   date           4600 non-null   object 
 1   price          4600 non-null   float64
 2   bedrooms       4600 non-null   float64
 3   bathrooms      4600 non-null   float64
 4   sqft_living    4600 non-null   int64  
 5   sqft_lot       4600 non-null   int64  
 6   floors         4600 non-null   float64
 7   waterfront     4600 non-null   int64  
 8   view           4600 non-null   int64  
 9   condition      4600 non-null   int64  
 10  sqft_above     4600 non-null   int64  
 11  sqft_basement  4600 non-null   int64  
 12  yr_built       4600 non-null   int64  
 13  yr_renovated   4600 non-null   int64  
 14  street         4600 non-null   object 
 15  city           4600 non-null   object 
 16  statezip       4600 non-null   object 
 17  country        4600 non-null   object 
dtypes: float64(4), int64(9), object(5)
memory usage: 647.0+ KB
```

In [8]:

```
advertising.describe()
```

Out[8]:

edrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	condition	sqft_above	sqft_
0.000000	4600.000000	4600.000000	4.600000e+03	4600.000000	4600.000000	4600.000000	4600.000000	4600.000000	46
3.400870	2.160815	2139.346957	1.485252e+04	1.512065	0.007174	0.240652	3.451739	1827.265435	3
0.908848	0.783781	963.206916	3.588444e+04	0.538288	0.084404	0.778405	0.677230	862.168977	4
0.000000	0.000000	370.000000	6.380000e+02	1.000000	0.000000	0.000000	1.000000	370.000000	
3.000000	1.750000	1460.000000	5.000750e+03	1.000000	0.000000	0.000000	3.000000	1190.000000	
3.000000	2.250000	1980.000000	7.683000e+03	1.500000	0.000000	0.000000	3.000000	1590.000000	
4.000000	2.500000	2620.000000	1.100125e+04	2.000000	0.000000	0.000000	4.000000	2300.000000	6
9.000000	8.000000	13540.000000	1.074218e+06	3.500000	1.000000	4.000000	5.000000	9410.000000	48

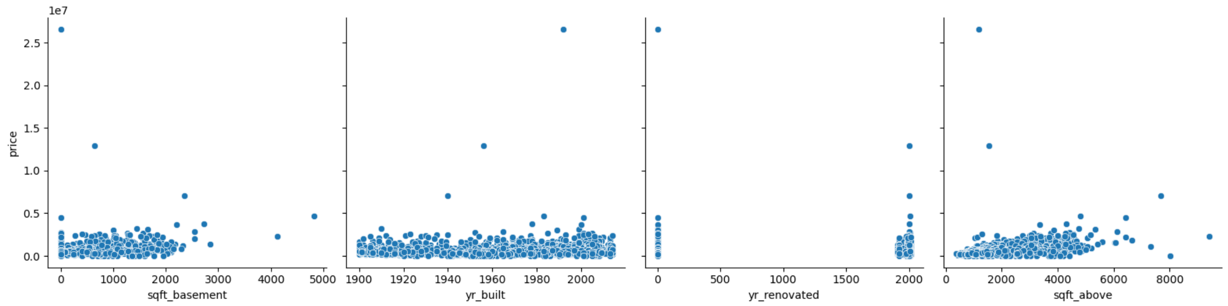
```
# Let's now visualise our data using seaborn. We'll first make a pairplot of all the variables present to
visualise which variables are most correlated
```

In [7]:

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

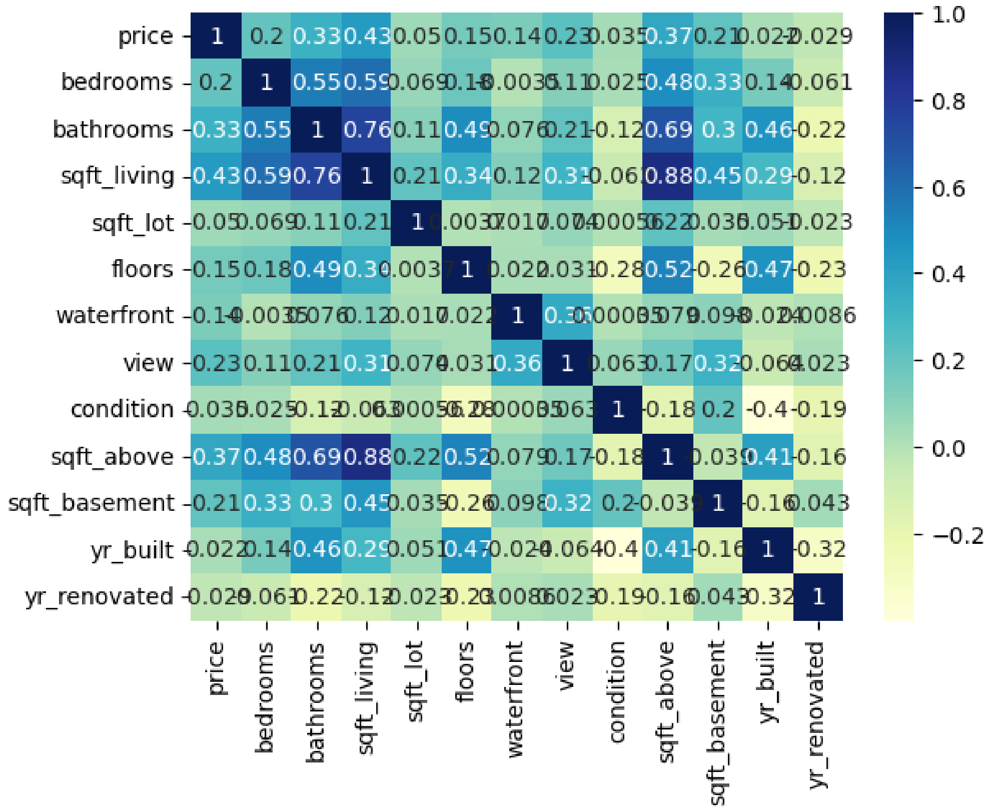
In [13]:

```
sns.pairplot(advertising, x_vars=['sqft_basement', 'yr_built', 'yr_renovated','sqft_above'], y_vars='price',siz
plt.show()
```



In [14]:

```
sns.heatmap(advertising.corr(), cmap="YlGnBu", annot = True)
plt.show()
```



Step 3: Performing Simple Linear Regression

Generic Steps in model building using statsmodels

In [34]:

```
X = advertising['price']
y = advertising['yr_built']
```

Train-Test Split
You now need to split our variable into training and testing sets. You'll perform this by importing train_test_split from the sklearn.model_selection library. It is usually a good practice to keep 70% of the data in your train dataset and the rest 30% in your test dataset

In [35]:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size = 0.7, test_size = 0.3, random_state = 100
```

In [36]:

```
# Let's now take a look at the train dataset

X_train.head()
```

Out[36]:

```
2867    4.400000e+05
3748    1.695000e+06
4475    4.586639e+05
428     5.660000e+05
474     9.270000e+05
Name: price, dtype: float64
```

In [37]:

```
y_train.head()
```

Out[37]:

```
2867    1968
3748    1924
4475    1941
428     1980
474     1953
Name: yr_built, dtype: int64
```

Building a Linear Model

You first need to import the statsmodel.api library using which you'll perform the linear regression.

In [38]:

```
import statsmodels.api as sm
```

In [39]:

```
# Add a constant to get an intercept
X_train_sm = sm.add_constant(X_train)

# Fit the resgression line using 'OLS'
lr = sm.OLS(y_train, X_train_sm).fit()
# Print the parameters, i.e. the intercept and the slope of the regression line fitted
lr.params
```

Out[39]:

```
const    1.970369e+03
price    6.793451e-07
dtype: float64
```

In [40]:

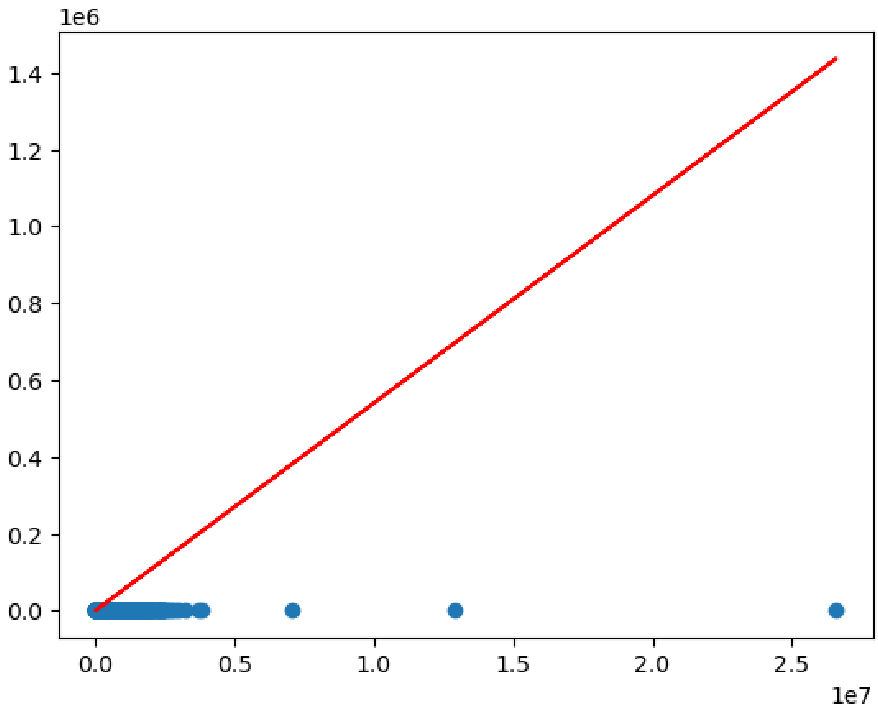
```
# Performing a summary operation lists out all the different parameters of the regression line fitted
print(lr.summary())
```

```
                            OLS Regression Results                            
==============================================================================
Dep. Variable:               yr_built   R-squared:                       0.000
Model:                            OLS   Adj. R-squared:                 -0.000
Method:                 Least Squares   F-statistic:                    0.6578
Date:                Thu, 27 Apr 2023   Prob (F-statistic):              0.417
Time:                        22:14:20   Log-Likelihood:                -15509.
No. Observations:                3220   AIC:                         3.102e+04
Df Residuals:                    3218   BIC:                         3.103e+04
Df Model:                           1                                         
Covariance Type:            nonrobust                                         
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const       1970.3692      0.704   2796.956      0.000    1968.988    1971.750
price       6.793e-07   8.38e-07      0.811      0.417   -9.63e-07    2.32e-06
==============================================================================
Omnibus:                      291.114   Durbin-Watson:                   2.029
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              198.881
Skew:                          -0.496   Prob(JB):                     6.51e-44
Kurtosis:                       2.295   Cond. No.                     1.12e+06
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 1.12e+06. This might indicate that there are
strong multicollinearity or other numerical problems.
```

In [41]:

```
plt.scatter(X_train, y_train)
plt.plot(X_train, 6.948 + 0.054*X_train, 'r')
plt.show()
```



Step 4: Residual analysis

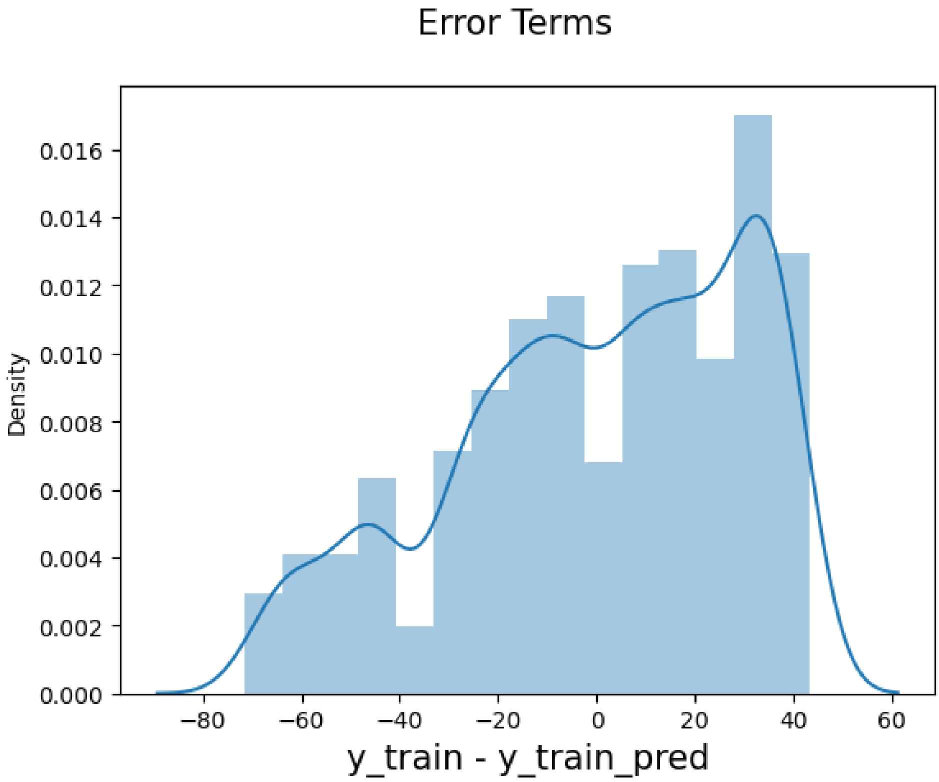
To validate assumptions of the model, and hence the reliability for inference

In [42]:

```
y_train_pred = lr.predict(X_train_sm)
res = (y_train - y_train_pred)
```

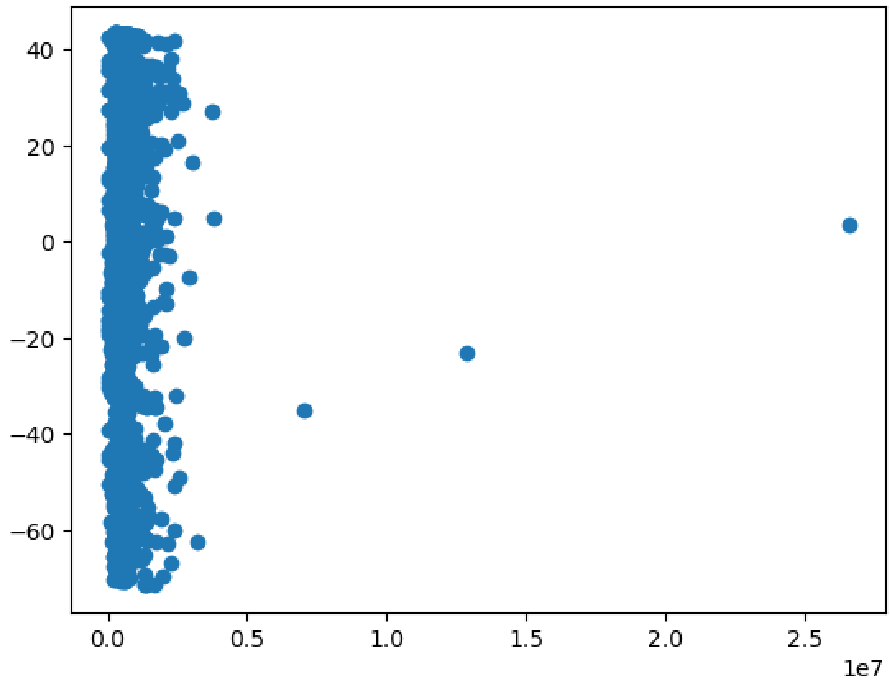
In [43]:

```
fig = plt.figure()
sns.distplot(res, bins = 15)
fig.suptitle('Error Terms', fontsize = 15)                  # Plot heading
plt.xlabel('y_train - y_train_pred', fontsize = 15)         # X-label
plt.show()
```



In [44]:

```
plt.scatter(X_train,res)
plt.show()
```



Step 5: Predictions on the Test Set

Now that you have fitted a regression line on your train dataset, it's time to make some predictions on the test data. For this, you first need to add a constant to the X_test data like you did for X_train and then you can simply go on and predict the y values corresponding to X_test using the predict attribute of the fitted regression lin

In [45]:

```
# Add a constant to X_test
X_test_sm = sm.add_constant(X_test)

# Predict the y values corresponding to X_test_sm
y_pred = lr.predict(X_test_sm)
```

In [46]:

```
y_pred.head()
```

Out[46]:

```
3386    1970.721777
918     1970.669807
338     1971.075037
4501    1970.527598
2856    1970.664712
dtype: float64
```

In [47]:

```
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score
```

Looking at the RMSE

In [48]:

```
#Returns the mean squared error; we'll take a square root
np.sqrt(mean_squared_error(y_test, y_pred))
```

Out[48]:

```
29.32943989630104
```

Checking the R-squared on the test set

In [49]:

```
r_squared = r2_score(y_test, y_pred)
r_squared
```

Out[49]:

```
0.0008443477993202997
```

Visualizing the fit on the test set

In [50]:

```
plt.scatter(X_test, y_test)
plt.plot(X_test, 6.948 + 0.054 * X_test, 'r')
plt.show()
```