

# REAL ESTATE PROBLEM

## 1. Business Problem

### 1.1 Problem Context

Our client is a large Real Estate Investment Trust (REIT).

- They invest in houses, apartments, and condos(complex of buildings) within a small county in New York state.
- As part of their business, they try to predict the fair transaction price of a property before it's sold.
- They do so to calibrate their internal pricing models and keep a pulse on the market.

### 1.2 Problem Statement ¶

The REIT has hired us to find a data-driven approach to valuing properties.

- They currently have an untapped dataset of transaction prices for previous properties on the market.
- The data was collected in 2016.
- Our task is to build a real-estate pricing model using that dataset.
- If we can build a model to predict transaction prices with an average error of under US Dollars 70,000, then our client will be very satisfied with the our resultant model.

### 1.3 Business Objectives and Constraints

- Deliverable: Trained model file
- Win condition: Avg. prediction error < \$70,000
- Model Interpretability will be useful
- No latency requirement

## 2. Machine Learning Problem

### 2.1 Data Overview

For this project:

1. The dataset has 1883 observations in the county where the REIT operates.
2. Each observation is for the transaction of one property only.
3. Each transaction was between \$200,000 and \$800,000.

#### Target Variable

- 'tx\_price' - Transaction price in USD

#### Features of the data:

Public records:

- 'tx\_year' - Year the transaction took place
- 'property\_tax' - Monthly property tax
- 'insurance' - Cost of monthly homeowner's insurance

Property characteristics:

- 'beds' - Number of bedrooms
- 'baths' - Number of bathrooms
- 'sqft' - Total floor area in squared feet
- 'lot\_size' - Total outside area in squared feet
- 'year\_built' - Year property was built
- 'active\_life' - Number of gyms, yoga studios, and sports venues within 1 mile
- 'basement' - Does the property have a basement?
- 'exterior\_walls' - The material used for constructing walls of the house
- 'roof' - The material used for constructing the roof

Location convenience scores:

- 'restaurants' - Number of restaurants within 1 mile
- 'groceries' - Number of grocery stores within 1 mile
- 'nightlife' - Number of nightlife venues within 1 mile
- 'cafes' - Number of cafes within 1 mile
- 'shopping' - Number of stores within 1 mile
- 'arts\_entertainment' - Number of arts and entertainment venues within 1 mile
- 'beauty\_spas' - Number of beauty and spa locations within 1 mile
- 'active\_life' - Number of gyms, yoga studios, and sports venues within 1 mile

Neighborhood demographics:

- 'median\_age' - Median age of the neighborhood
- 'married' - Percent of neighborhood who are married
- 'college\_grad' - Percent of neighborhood who graduated college

Schools:

- 'num\_schools' - Number of public schools within district
- 'median\_school' - Median score of the public schools within district, on the range 1 - 10

## 2.2 Mapping business problem to ML problem

### 2.2.1 Type of Machine Learning Problem

It is a regression problem, where given the above set of features, we need to predict the transaction price of the house.

### 2.2.2 Performance Metric (KPI)

Since it is a regression problem, we will use the following regression metrics:

$$\text{RMSE} = \sqrt{\frac{1}{n} \sum_{j=1}^n (y_j - \hat{y}_j)^2}$$


#### 2.2.2.2 R-squared

Sum Squared Regression Error

$$R^2 = 1 - \frac{SS_{Regression}}{SS_{Total}}$$

Sum Squared Total Error

### 2.2.2.3 Mean Absolute Error (MAE):


$$\text{MAE} = \frac{1}{n} \sum_{j=1}^n |y_j - \hat{y}_j|$$

©easycalculation.com

- Remember, our win-condition for this project is predicting within \$70,000 of true transaction prices, on average.
- Mean absolute error (or MAE) is the average absolute difference between predicted and actual values for our target variable. That exactly aligns with the terms of our win condition!
- So we're aiming to get MAE below 70,000

## 2.3 Train-Test Splits

80-20

## 3. Exploratory Data Analysis

Import the libraries

In [1]:

```
# NumPy for numerical computing
import numpy as np

# Pandas for DataFrames
import pandas as pd

# Matplotlib for visualization
from matplotlib import pyplot as plt
# display plots in the notebook
%matplotlib inline
# import color maps
from matplotlib.colors import ListedColormap

# Seaborn for easier visualization
import seaborn as sns

# Ignore Warnings
import warnings
warnings.filterwarnings("ignore")

from math import sqrt

# Function for splitting training and test set
from sklearn.model_selection import train_test_split

# Function to perform data standardization
from sklearn.preprocessing import StandardScaler

# Libraries to perform hyperparameter tuning
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RandomizedSearchCV

# Import classes for ML Models
from sklearn.linear_model import Ridge ## Linear Regression + L2 regularization
from sklearn.linear_model import Lasso ## Linear Regression + L1 regularization

from sklearn.svm import SVR ## Support Vector Regressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.tree import DecisionTreeRegressor

# Evaluation Metrics
from sklearn.metrics import mean_squared_error as mse
from sklearn.metrics import r2_score
from sklearn.metrics import mean_absolute_error as mae
## import xgboost
import os
os.environ['PATH'] = os.environ['PATH'] + ';C:\\Program Files\\mingw-w64\\x86_64-5.3.0-
posix-seh-rt_v4-rev0\\mingw64\\bin'

from xgboost import XGBRegressor
from xgboost import plot_importance ## to plot feature importance

# To save the final model on disk
from sklearn.externals import joblib ## Reference http://scikit-learn.org/stable/modules/model\_persistence.html

from sklearn.cross_validation import cross_val_score
from sklearn.neighbors import KNeighborsClassifier
```

```
from matplotlib.colors import ListedColormap
from sklearn.metrics import accuracy_score
```

C:\ProgramData\Anaconda3\lib\site-packages\sklearn\cross\_validation.py:41: DeprecationWarning: This module was deprecated in version 0.18 in favor of the model\_selection module into which all the refactored classes and functions are moved. Also note that the interface of the new CV iterators are different from that of this module. This module will be removed in 0.20.

"This module will be removed in 0.20.", DeprecationWarning)

In [2]:

```
np.set_printoptions(precision=2, suppress=True)
```

### 3.1 Load real estate data from CSV

In [3]:

```
df = pd.read_csv('Files/real_estate_data.csv')
```

Display the dimensions of the dataset.

In [4]:

```
df.shape
```

Out[4]:

```
(1883, 26)
```

Columns of the dataset

In [5]:

```
df.columns
```

Out[5]:

```
Index(['tx_price', 'beds', 'baths', 'sqft', 'year_built', 'lot_size',
       'property_type', 'exterior_walls', 'roof', 'basement', 'restaurant',
       'groceries', 'nightlife', 'cafes', 'shopping', 'arts_entertainment',
       'beauty_spas', 'active_life', 'median_age', 'married', 'college_grad',
       'property_tax', 'insurance', 'median_school', 'num_schools', 'tx_year'],
      dtype='object')
```

Display the first 5 rows to see example observations.

In [6]:

```
pd.set_option('display.max_columns', 100) ## display max 100 columns
df.head()
```

Out[6]:

	tx_price	beds	baths	sqft	year_built	lot_size	property_type	exterior_walls	
0	295850	1	1	584	2013	0	Apartment / Condo / Townhouse	Wood Siding	Na
1	216500	1	1	612	1965	0	Apartment / Condo / Townhouse	Brick	Co Sh
2	279900	1	1	615	1963	0	Apartment / Condo / Townhouse	Wood Siding	Na
3	379900	1	1	618	2000	33541	Apartment / Condo / Townhouse	Wood Siding	Na
4	340000	1	1	634	1992	0	Apartment / Condo / Townhouse	Brick	Na

Some feaures are numeric and some are categorical

Filtering the categorical features:

In [7]:

```
df.dtypes[df.dtypes=='object']
```

Out[7]:

```
property_type    object
exterior_walls   object
roof             object
dtype: object
```

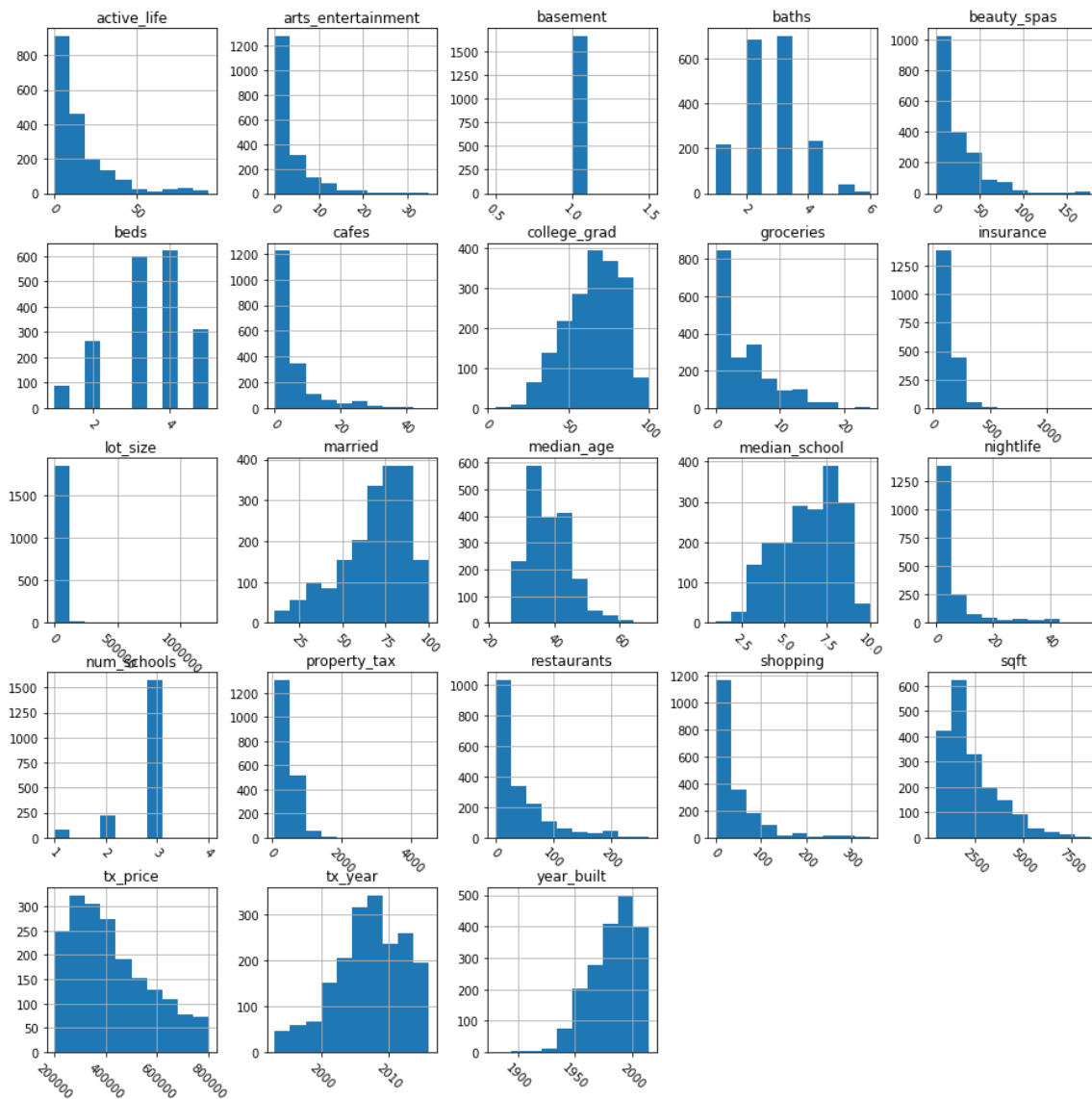
## 3.2 Distributions of numeric features

One of the most enlightening data exploration tasks is plotting the distributions of the features.

In [8]:

```
# Plot histogram grid
df.hist(figsize=(16,16), xrot=-45) ## Display the labels rotated by 45 degress

# Clear the text "residue"
plt.show()
```





**Observations:** We can make out quite a few observations:

For example, consider the histogram for beds:

- Over 600 houses have 4 bedrooms, and similar number of houses have 3 bedrooms.
- Less than 100 houses have one bedroom and so on.

Also, look at the plot for the 'year\_built' feature:

- The last bin in the histogram represents the range 2010-2020.
- Since this dataset was pulled in 2016, we should not have properties built in 2019.
- A property built after 2016 would be a measurement error.
- However, because of the 10-year bins, it's hard to tell if there's a measurement error just from the plot above.

Display summary statistics for the numerical features.

In [9]:

```
df.describe()
```

Out[9]:

	tx_price	beds	baths	sqft	year_built	basement
count	1883.000000	1883.000000	1883.000000	1883.000000	1883.000000	1.883000
mean	422839.807754	3.420605	2.579926	2329.398832	1982.963887	1.339200
std	151462.593276	1.068554	0.945576	1336.991858	20.295945	4.494900
min	200000.000000	1.000000	1.000000	500.000000	1880.000000	0.000000
25%	300000.000000	3.000000	2.000000	1345.000000	1970.000000	1.542000
50%	392000.000000	3.000000	3.000000	1907.000000	1986.000000	6.098000
75%	525000.000000	4.000000	3.000000	3005.000000	2000.000000	1.176100
max	800000.000000	5.000000	6.000000	8450.000000	2015.000000	1.220500

**Obeservation:**

- Look at the 'year\_built' column, we can see that its max value is 2015.
- The 'basement' feature has some missing values, also its standard deviation is 0.0, while its min and max are both 1.0.

### 3.3 Distributions of categorical features

Display summary statistics for categorical features.

In [10]:

```
df.describe(include=['object'])
```

Out[10]:

	property_type	exterior_walls	roof
count	1883	1660	1529
unique	2	16	16
top	Single-Family	Brick	Composition Shingle
freq	1080	687	1179

### Observation:

- 'exterior\_walls' and 'roof' have missing values
- There are 16 unique classes for 'exterior\_walls', and the most frequent one is 'Brick'.

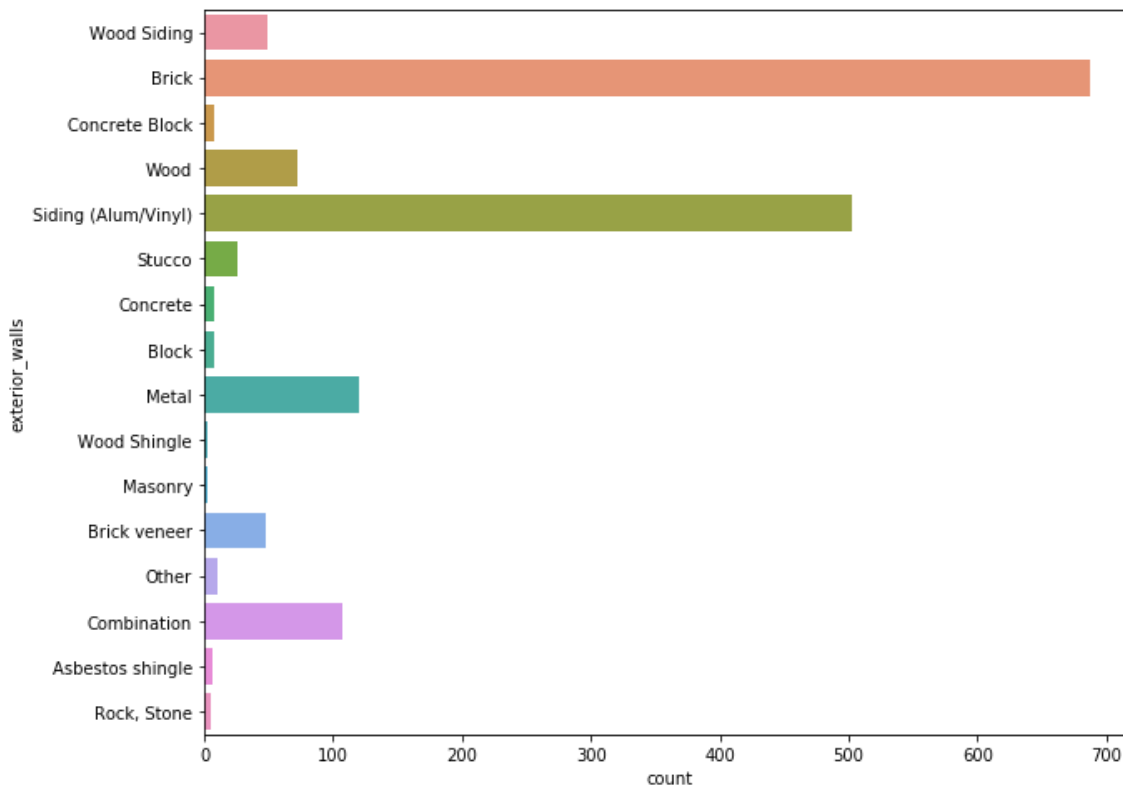
Plot bar plot for the 'exterior\_walls' feature.

In [11]:

```
plt.figure(figsize=(10,8))
sns.countplot(y='exterior_walls', data=df)
```

Out[11]:

<matplotlib.axes.\_subplots.AxesSubplot at 0x291d62584e0>



**Observations:** Take a look at the frequencies of the classes.

Several classes are quite prevalent in the dataset. They have long bars. Those include:

- 'Brick'
- 'Siding (Alum/Vinyl)'
- 'Metal'
- 'Combination'

On the flipside, some classes have really short bars. For example:

- 'Concrete Block'
- 'Concrete'
- 'Block'
- 'Wood Shingle'
- etc...
- These don't have many observations, and they are called sparse classes.

## 3.4 Sparse Classes

Sparse classes are classes in categorical features that have a very small number of observations.

They tend to be problematic when we get to building models.

- In the best case, they don't influence the model much.
- In the worst case, they can cause the model to be overfit.

Let's make a mental note to combine or reassign some of these classes later.

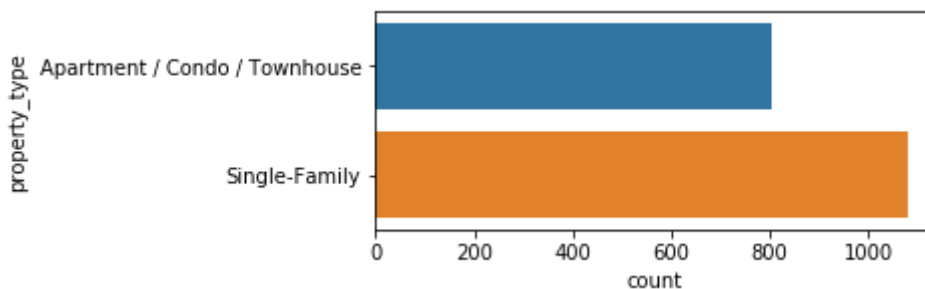
### Bar plot for each categorical feature

In [12]:

```
plt.figure(figsize=(5,2))
sns.countplot(y='property_type', data=df)
```

Out[12]:

<matplotlib.axes.\_subplots.AxesSubplot at 0x291d62a8da0>

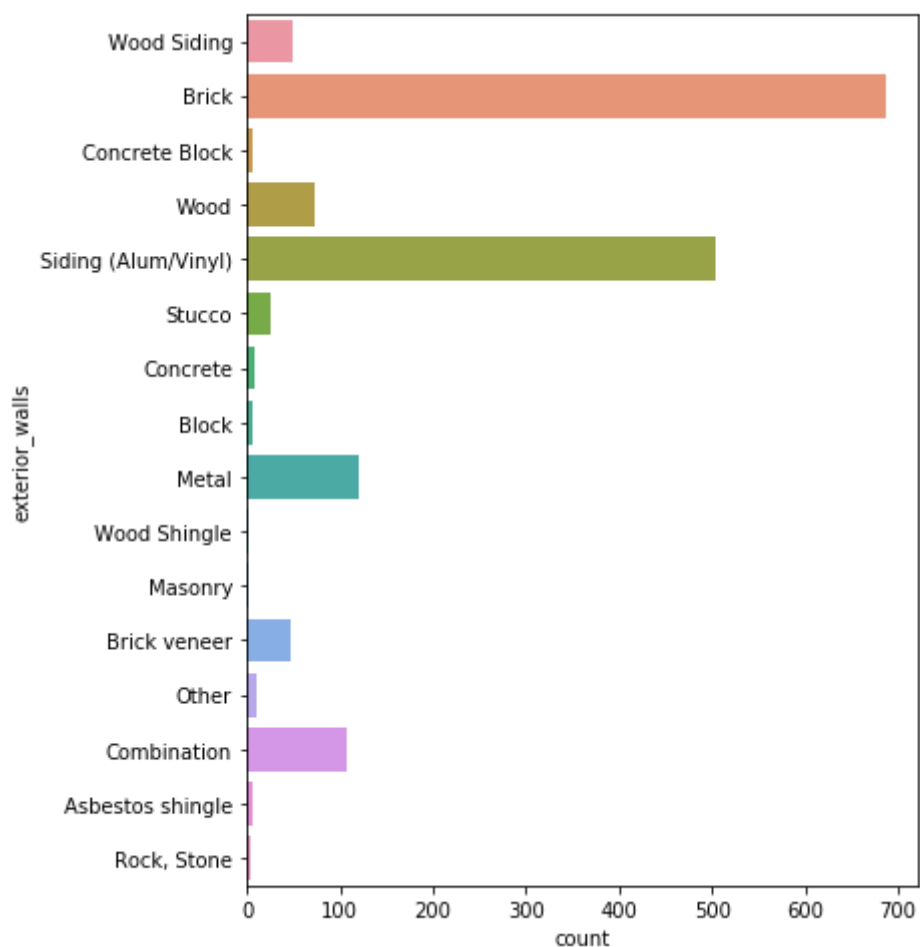


In [13]:

```
plt.figure(figsize=(6,8))  
sns.countplot(y='exterior_walls', data=df)
```

Out[13]:

<matplotlib.axes.\_subplots.AxesSubplot at 0x291d6608c18>

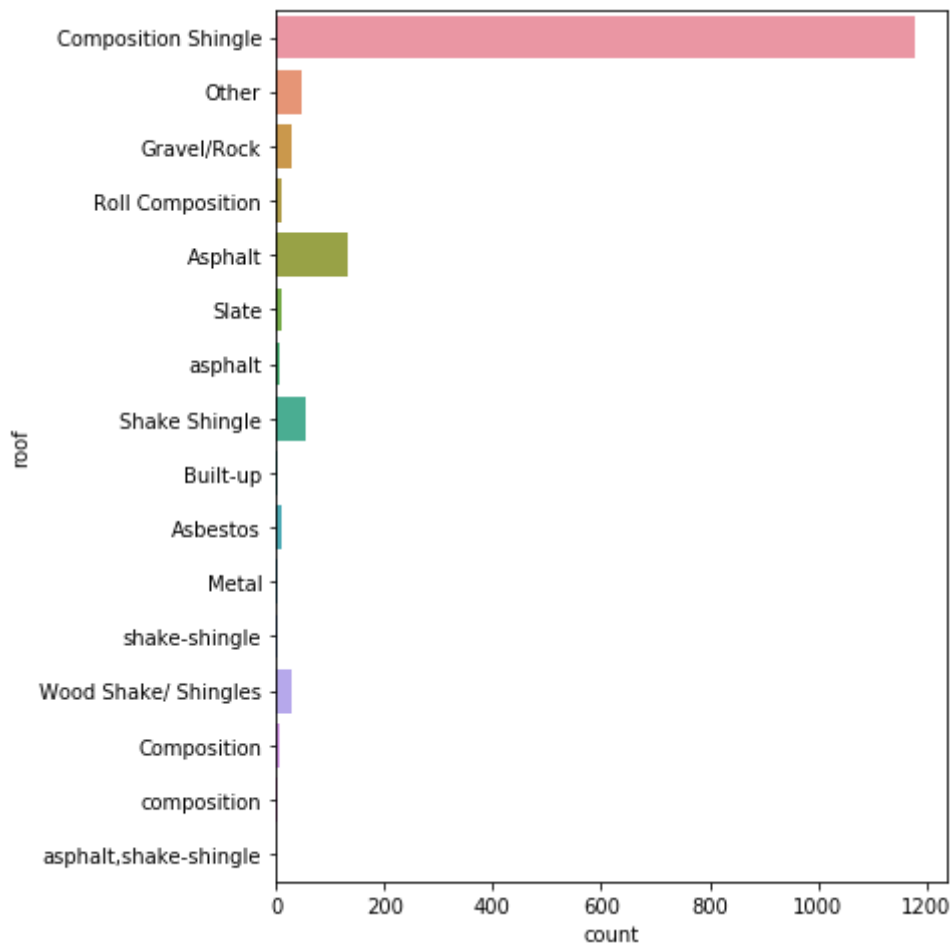


In [14]:

```
plt.figure(figsize=(6,8))  
sns.countplot(y='roof', data=df)
```

Out[14]:

<matplotlib.axes.\_subplots.AxesSubplot at 0x291d6d8c6a0>



**Observations:** In both 'exterior\_walls' and 'roof' as well we have many sparse classes

### 3.5 Segmentations

Segmentations are powerful ways to cut the data to observe the relationship between categorical features and numeric features.

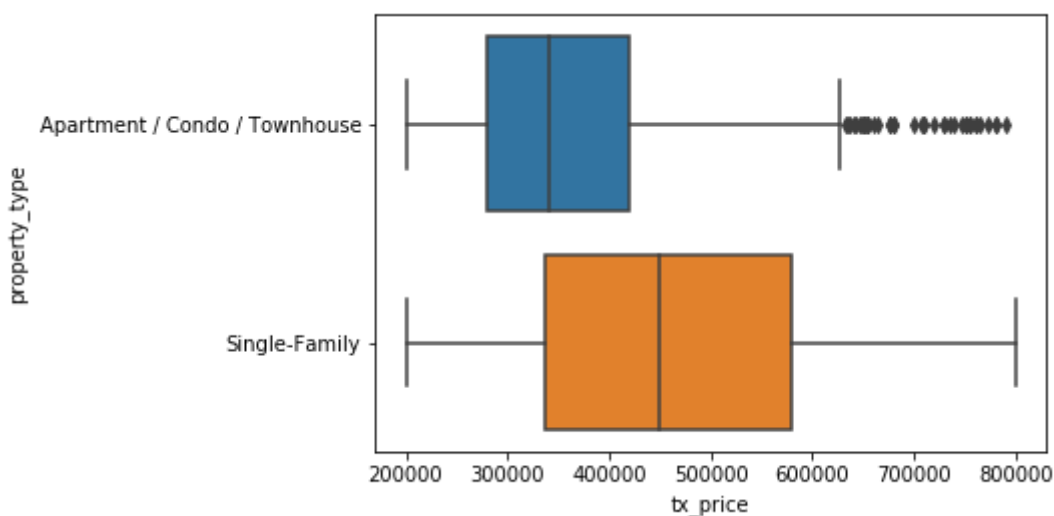
Segmenting the target variable by key categorical features.

In [15]:

```
sns.boxplot(y='property_type', x='tx_price', data=df)
```

Out[15]:

<matplotlib.axes.\_subplots.AxesSubplot at 0x291d6746940>



**Observation:** In general, it looks like single family homes are more expensive.

Comparing the two property types across other features as well

In [16]:

```
df.groupby('property_type').mean()
```

Out[16]:

	tx_price	beds	baths	sqft	year_built	lot_s
property_type						
Apartment / Condo / Townhouse	366614.034869	2.601494	2.200498	1513.727273	1988.936488	3944
Single-Family	464644.711111	4.029630	2.862037	2935.865741	1978.523148	2041

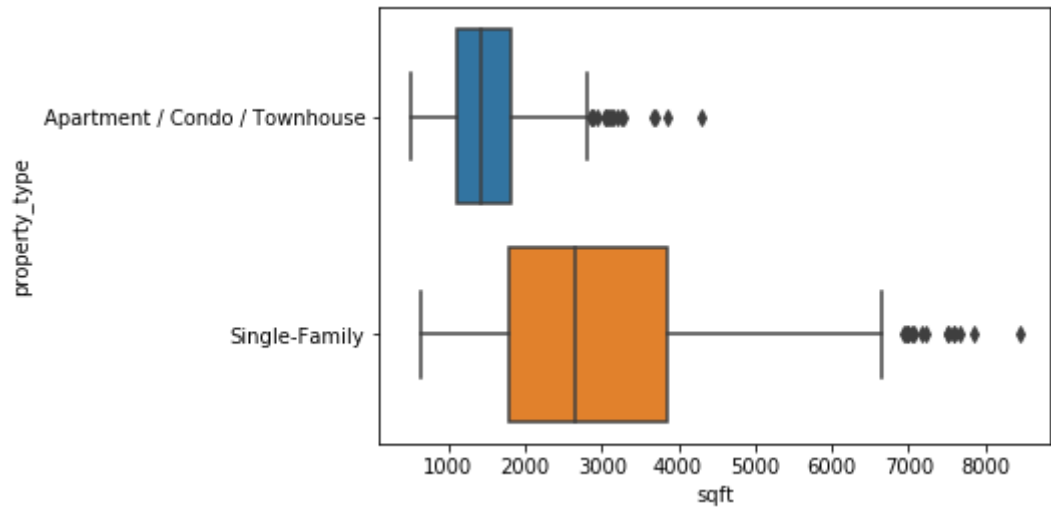
Segment 'sqft' by 'property\_type' and plot the boxplots.

In [17]:

```
sns.boxplot(y='property_type', x='sqft', data=df)
```

Out[17]:

<matplotlib.axes.\_subplots.AxesSubplot at 0x291d6741a58>



Segment by property\_type and display the means and standard deviations within each class

In [18]:

```
df.groupby('property_type').agg([np.mean, np.std])
```

Out[18]:

	tx_price		beds		baths	
	mean	std	mean	std	mean	std
property_type						
Apartment / Condo / Townhouse	366614.034869	121784.490486	2.601494	0.810220	2.200498	0.8150
Single-Family	464644.711111	157758.739013	4.029630	0.795639	2.862037	0.9375

## 3.6 Correlations

- Finally, let's take a look at the relationships between numeric features and other numeric features.
- **Correlation** is a value between -1 and 1 that represents how closely values for two separate features move in unison.
- Positive correlation means that as one feature increases, the other increases; eg. a child's age and her height.....(monotonically increasing)
- Negative correlation means that as one feature increases, the other decreases; eg. hours spent studying and number of parties attended.
- Correlations near -1 or 1 indicate a strong relationship.
- Those closer to 0 indicate a weak relationship.
- 0 indicates no relationship.



In [19]:

df.corr()

Out[19]:

	<b>tx_price</b>	<b>beds</b>	<b>baths</b>	<b>sqft</b>	<b>year_built</b>	<b>lot_size</b>
<b>tx_price</b>	1.000000	0.384046	0.389375	0.469573	0.033138	0.130558
<b>beds</b>	0.384046	1.000000	0.644572	0.691039	-0.011844	0.164399
<b>baths</b>	0.389375	0.644572	1.000000	0.682060	0.206141	0.132976
<b>sqft</b>	0.469573	0.691039	0.682060	1.000000	0.254589	0.246016
<b>year_built</b>	0.033138	-0.011844	0.206141	0.254589	1.000000	0.000068
<b>lot_size</b>	0.130558	0.164399	0.132976	0.246016	0.000068	1.000000
<b>basement</b>	NaN	NaN	NaN	NaN	NaN	NaN
<b>restaurants</b>	-0.038027	-0.495834	-0.350210	-0.353759	-0.106948	-0.113887
<b>groceries</b>	-0.094314	-0.421412	-0.340024	-0.371167	-0.222443	-0.118574
<b>nightlife</b>	0.009361	-0.440844	-0.306686	-0.281540	-0.080006	-0.072224
<b>cafes</b>	-0.001398	-0.464289	-0.316836	-0.300969	-0.102209	-0.098182
<b>shopping</b>	-0.038246	-0.388670	-0.259603	-0.275586	-0.121684	-0.116560
<b>arts_entertainment</b>	-0.021076	-0.442168	-0.305885	-0.293402	-0.195141	-0.067189
<b>beauty_spas</b>	-0.054349	-0.419832	-0.282008	-0.310465	-0.163670	-0.121075
<b>active_life</b>	-0.001165	-0.486957	-0.329736	-0.332663	-0.134919	-0.092668
<b>median_age</b>	0.126335	0.133711	0.095844	0.109811	-0.237152	0.099140
<b>married</b>	0.200494	0.643240	0.442225	0.480167	0.038208	0.122028
<b>college_grad</b>	0.268577	-0.082354	0.016097	0.065343	-0.014204	-0.030725
<b>property_tax</b>	0.535148	0.547643	0.525776	0.660264	-0.046504	0.165800
<b>insurance</b>	0.532947	0.485776	0.475430	0.594049	-0.109521	0.146579
<b>median_school</b>	0.175762	0.137309	0.163365	0.220669	0.155835	0.096551
<b>num_schools</b>	-0.014380	-0.124359	-0.088299	-0.102032	-0.188106	-0.031535
<b>tx_year</b>	0.108782	-0.174081	-0.132110	-0.236190	-0.043301	-0.048207

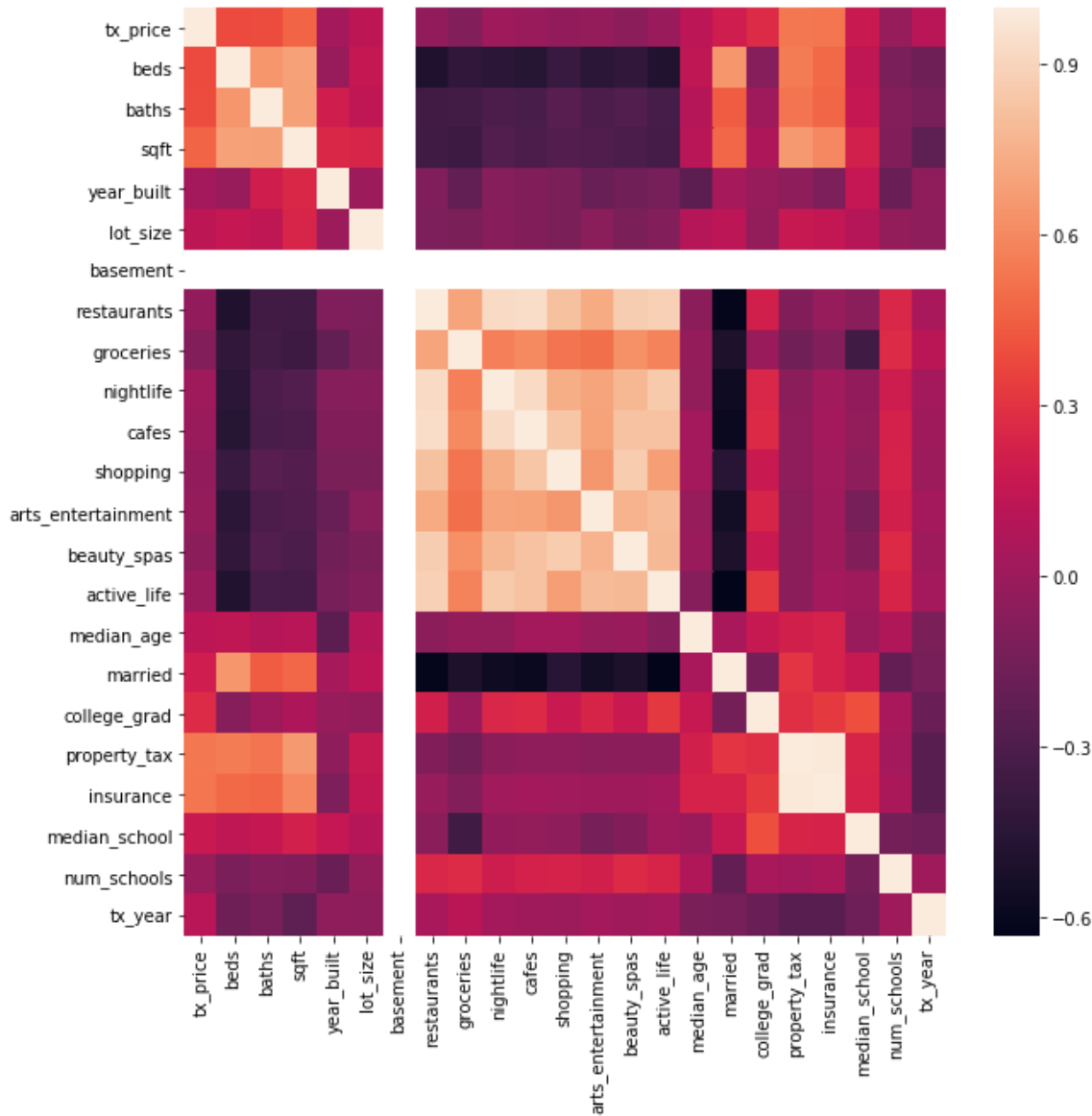
Let's visualize this.

In [20]:

```
plt.figure(figsize=(10,10))
sns.heatmap(df.corr())
```

Out[20]:

<matplotlib.axes.\_subplots.AxesSubplot at 0x291d69e3ef0>

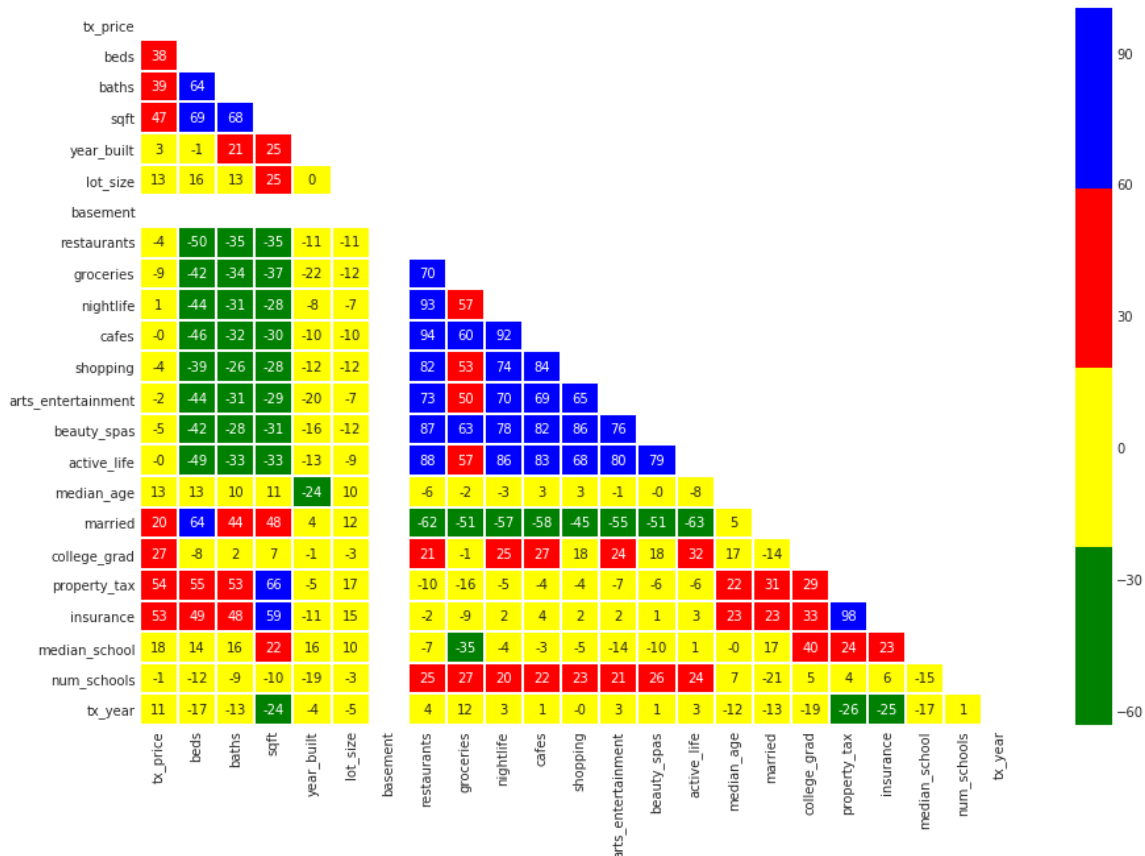


### What to look for?

- The colorbar on the right explains the meaning of the heatmap - Dark colors indicate **strong negative correlations** and light colors indicate **strong positive correlations**.
- Perhaps the most helpful way to interpret this correlation heatmap is to first find features that are correlated with our target variable by scanning the first column.
- In this case, it doesn't look like many features are strongly correlated with the target variable.
- There is a weak positive correlation between 'tx\_price' and 'property\_tax'.

In [21]:

```
mask=np.zeros_like(df.corr())
mask[np.triu_indices_from(mask)] = True
plt.figure(figsize=(15,10))
with sns.axes_style("white"):
    ax = sns.heatmap(df.corr()*100, mask=mask, fmt='.0f', annot=True, lw=1, cmap=Listed
Colormap(['green', 'yellow', 'red', 'blue']))
```



## 4. Data Cleaning

### 4.1 Dropping the duplicates (De-duplication)

Duplicate observations most frequently arise during data collection, such as when we:

- Combine datasets from multiple places
- Scrape data
- Receive data from clients/other departments

In [22]:

```
df = df.drop_duplicates()
print( df.shape )
```

(1883, 26)

There are no duplicates in original dataset.

## 4.2 Fix structural errors

- Recall, the basement features had some nan values

In [23]:

```
df.basement.unique()
```

Out[23]:

```
array([ nan,   1.])
```

- Even though NaN represents "missing" values, those are actually meant to indicate properties without basements.
- Fill missing 'basement' values with the value 0 to turn 'basement' into a true indicator variable.

In [24]:

```
df.basement.fillna(0, inplace=True)
```

In [25]:

```
df.basement.unique()
```

Out[25]:

```
array([ 0.,   1.])
```

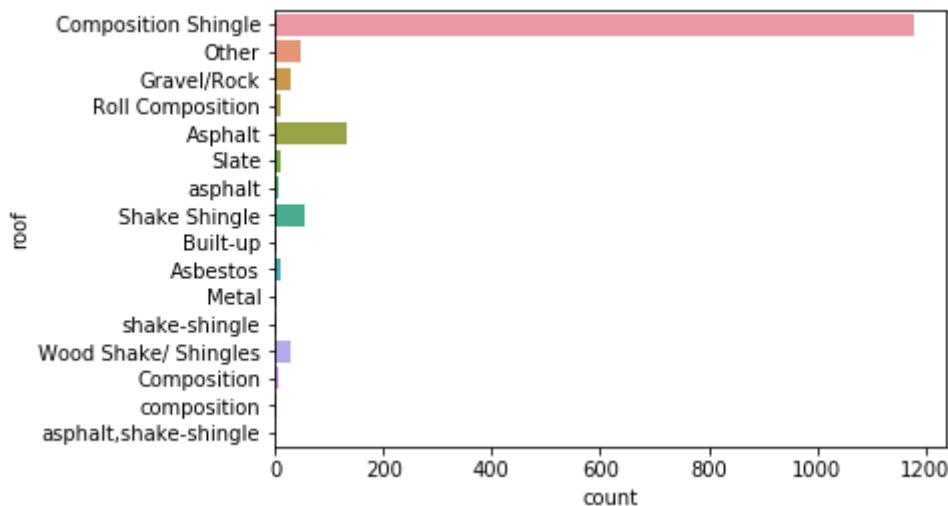
## 4.3 Typos and capitalization

In [26]:

```
# Class distributions for 'roof'
sns.countplot(y='roof', data=df)
```

Out[26]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x291d8275048>
```



Using this plot, we can easily catch typos and inconsistent capitalization. For example:

- 'composition' should be 'Composition'
- 'asphalt' should be 'Asphalt'
- 'shake-shingle' should be 'Shake Shingle'
- 'asphalt,shake-shingle' could probably just be 'Shake Shingle'

In [27]:

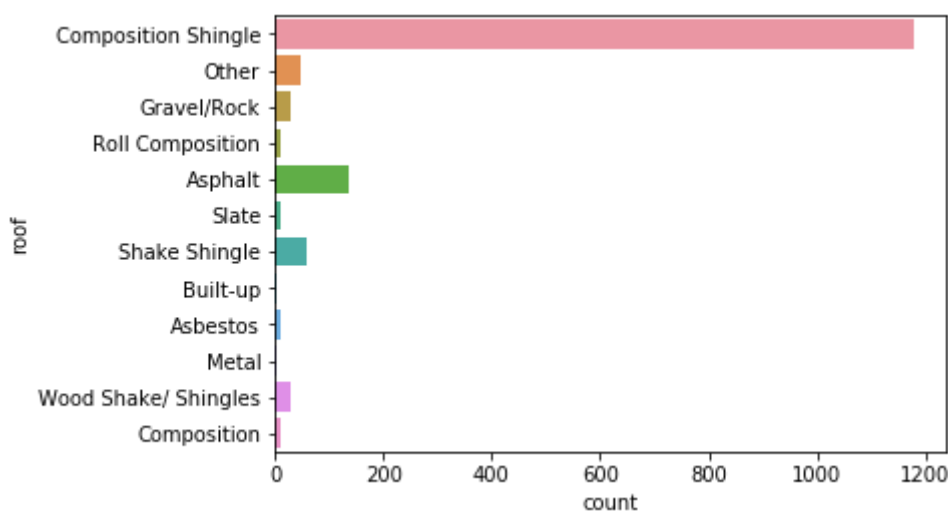
```
df.roof.replace('composition', 'Composition', inplace=True)
df.roof.replace('asphalt', 'Asphalt', inplace=True)
df.roof.replace(['shake-shingle', 'asphalt,shake-shingle'], 'Shake Shingle', inplace=True)
```

In [28]:

```
# Class distribution for 'roof' after the modifications
sns.countplot(y='roof', data=df)
```

Out[28]:

<matplotlib.axes.\_subplots.AxesSubplot at 0x291d8594be0>



## 4.4 Mislabeled classes

Finally, we'll check for classes that are labeled as separate classes when they should really be the same.

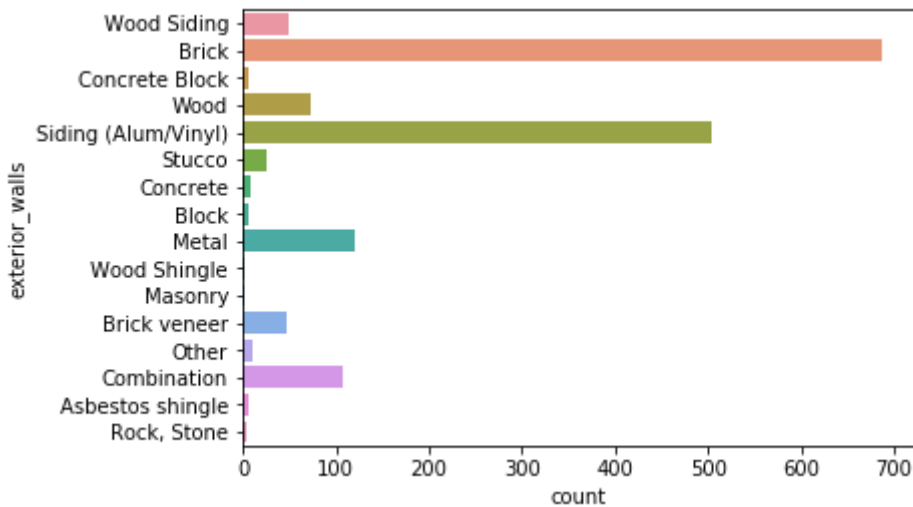
- e.g. If 'N/A' and 'Not Applicable' appear as two separate classes, we should combine them.
- For example, let's plot the class distributions for 'exterior\_walls':

In [29]:

```
sns.countplot(y='exterior_walls', data=df)
```

Out[29]:

<matplotlib.axes.\_subplots.AxesSubplot at 0x291d6342fd0>



Using that chart, we can easily catch mislabeled classes. For example

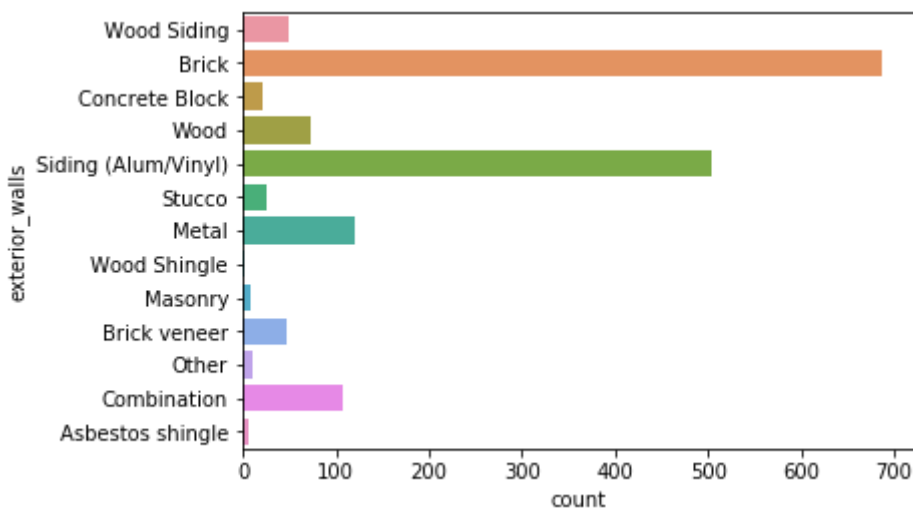
- 'Rock, Stone' should be 'Masonry'
- 'Concrete' and 'Block' should both just be 'Concrete Block'

In [30]:

```
df.exterior_walls.replace(['Rock, Stone'], 'Masonry', inplace=True)
df.exterior_walls.replace(['Concrete', 'Block'], 'Concrete Block', inplace=True)
sns.countplot(y='exterior_walls', data=df)
```

Out[30]:

<matplotlib.axes.\_subplots.AxesSubplot at 0x291d6338438>



## 4.5 Removing Outliers

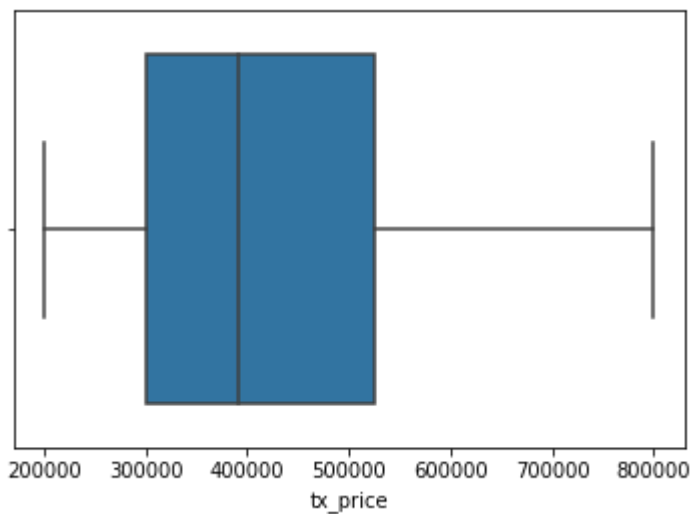
- Outliers can cause problems with certain types of models.
- Boxplots are a nice way to detect outliers
- Let's start with a box plot of the target variable, since that's what we're actually trying to predict.

In [31]:

```
sns.boxplot(df.tx_price)
```

Out[31]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x291d8b19358>
```



### Interpretation

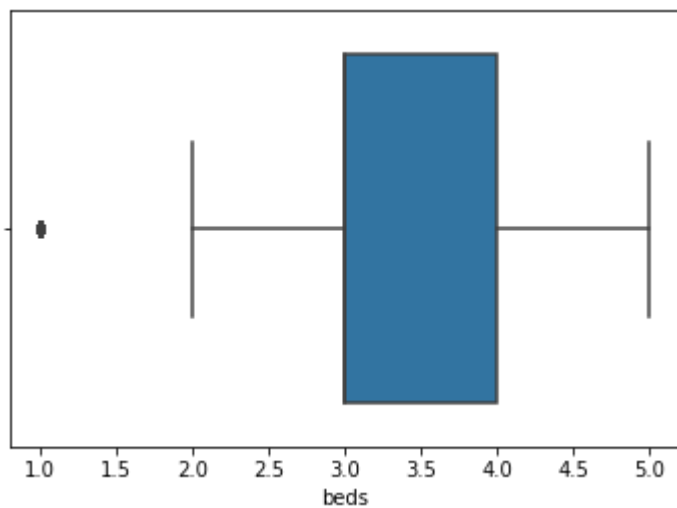
- The two vertical bars on the ends are the min and max values. All properties sold for between \$200,000 and \$800,000.
- The box in the middle is the interquartile range (25th percentile to 75th percentile).
- Half of all observations fall in that box.
- Finally, the vertical bar in the middle of the box is the median.

In [32]:

```
## Checking outliers in number of bedrooms  
sns.boxplot(df.beds)
```

Out[32]:

<matplotlib.axes.\_subplots.AxesSubplot at 0x291d642c978>

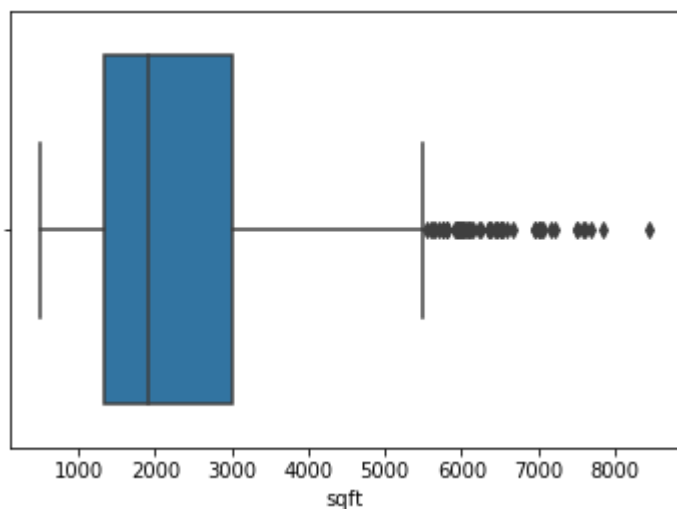


In [33]:

```
## Checking outliers in size of the house  
sns.boxplot(df.sqft)
```

Out[33]:

<matplotlib.axes.\_subplots.AxesSubplot at 0x291d6cd7710>



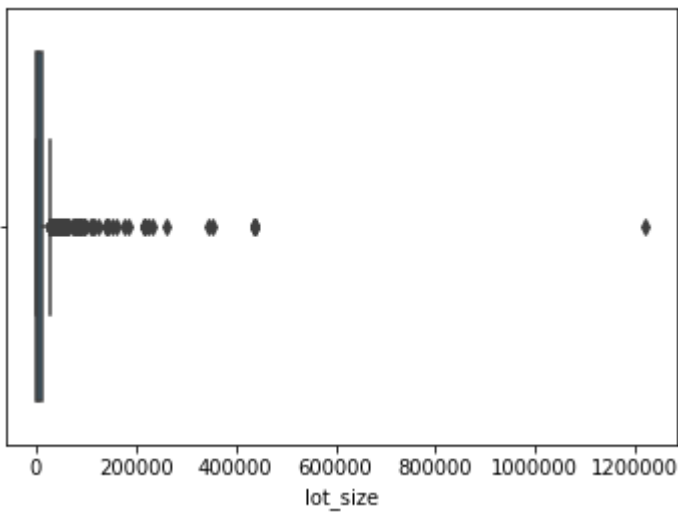


In [34]:

```
## Checking outliers in lot size
sns.boxplot(df.lot_size)
```

Out[34]:

<matplotlib.axes.\_subplots.AxesSubplot at 0x291d5bd6438>



Look at the dot on the extreme right. This might indicate some erroneous point.

Let's look at the largest 5 lot sizes just to confirm.

In [35]:

```
df.lot_size.sort_values(ascending=False).head()
```

Out[35]:

```
102      1220551
1111     436471
1876     436035
1832     436035
1115     435600
Name: lot_size, dtype: int64
```

The largest property has a lot\_size of 1,220,551 sqft. The next largest has a lot\_size of only 436,471 sqft.

Because it's unlikely the REIT will ever invest in properties with lots that large, and because it's the only one in the dataset, let's remove it so it doesn't interfere with our ability to model normal size properties.

In [36]:

```
## Remove observations with lot_size greater than 500,000 sqft.
df = df[df.lot_size <= 500000]
df.shape
```

Out[36]:

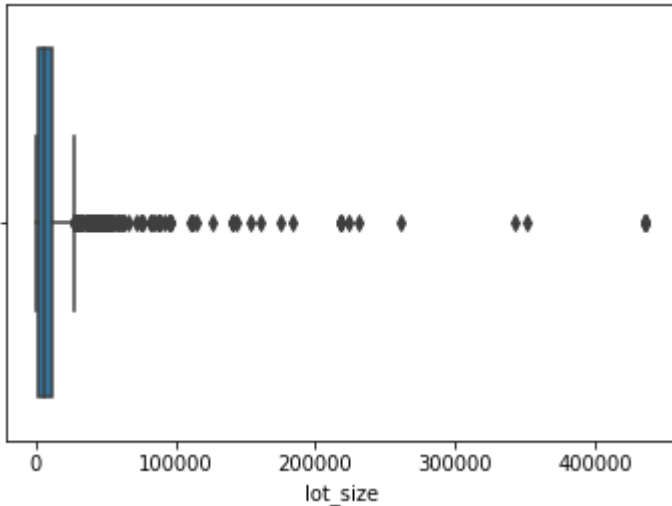
```
(1882, 26)
```

In [37]:

```
## Plotting the boxplot of lot size after the change
sns.boxplot(df.lot_size)
```

Out[37]:

<matplotlib.axes.\_subplots.AxesSubplot at 0x291d5b34dd8>



**Note:** Even though visually the plot looks the same but the x-axis dimensions have changed

## 4.6 Label missing categorical data

You cannot simply ignore missing values in your dataset. You must handle them in some way for the very practical reason that Scikit-Learn algorithms do not accept missing values.

In [38]:

```
# Display number of missing values by categorical feature
df.select_dtypes(include=['object']).isnull().sum()
```

Out[38]:

```
property_type      0
exterior_walls    223
roof              353
dtype: int64
```

**The best way to handle missing data for categorical features is to simply label them as 'Missing'!**

In [39]:

```
df['exterior_walls'] = df['exterior_walls'].fillna('Missing')
df['roof'] = df['roof'].fillna('Missing')
df.select_dtypes(include=['object']).isnull().sum()
```

Out[39]:

```
property_type      0
exterior_walls     0
roof               0
dtype: int64
```

## 4.7 Flag and fill missing numeric data

In [40]:

```
# Display number of missing values by numeric feature
df.select_dtypes(exclude=['object']).isnull().sum()
```

Out[40]:

```
tx_price      0
beds          0
baths        0
sqft          0
year_built    0
lot_size      0
basement      0
restaurants   0
groceries     0
nightlife     0
cafes         0
shopping      0
arts_entertainment 0
beauty_spas   0
active_life   0
median_age    0
married       0
college_grad  0
property_tax  0
insurance     0
median_school 0
num_schools   0
tx_year       0
dtype: int64
```

Well, it looks like we don't have any numerical features with missing values in this dataset.

**Before we move on to the next module, let's save the new dataframe we worked hard to clean.**

This makes sure we don't have to re-do all the cleaning after closing the session

In [41]:

```
# Save cleaned dataframe to new file  
df.to_csv('Files/cleaned_df.csv', index=None)
```

## 5. Feature Engineering

- Feature engineering is the practice of creating new features from existing ones
- The engineered features are often more specific or isolate key information.
- Often, feature engineering is one of the most valuable tasks a data scientist can do to improve model effectiveness.

**Note:** There are limitless possibilities for Feature Engineering, and it's a skill that will naturally improve as you gain more experience and domain expertise.

### 5.1 Indicator variables

- For example, let's say you knew that homes with 2 bedrooms and 2 bathrooms are especially popular for investors.
- Maybe you suspect these types of properties command premium prices. (You don't need to know for sure.)
- Create an indicator variable to flag properties with 2 beds and 2 baths and name it 'two\_and\_two'.

In [42]:

```
df['two_and_two'] = ((df.beds == 2) & (df.baths == 2)).astype(int)
```

In [43]:

```
# Display percent of rows where two_and_two == 1  
df[df['two_and_two']==1].shape[0]/df.shape[0]
```

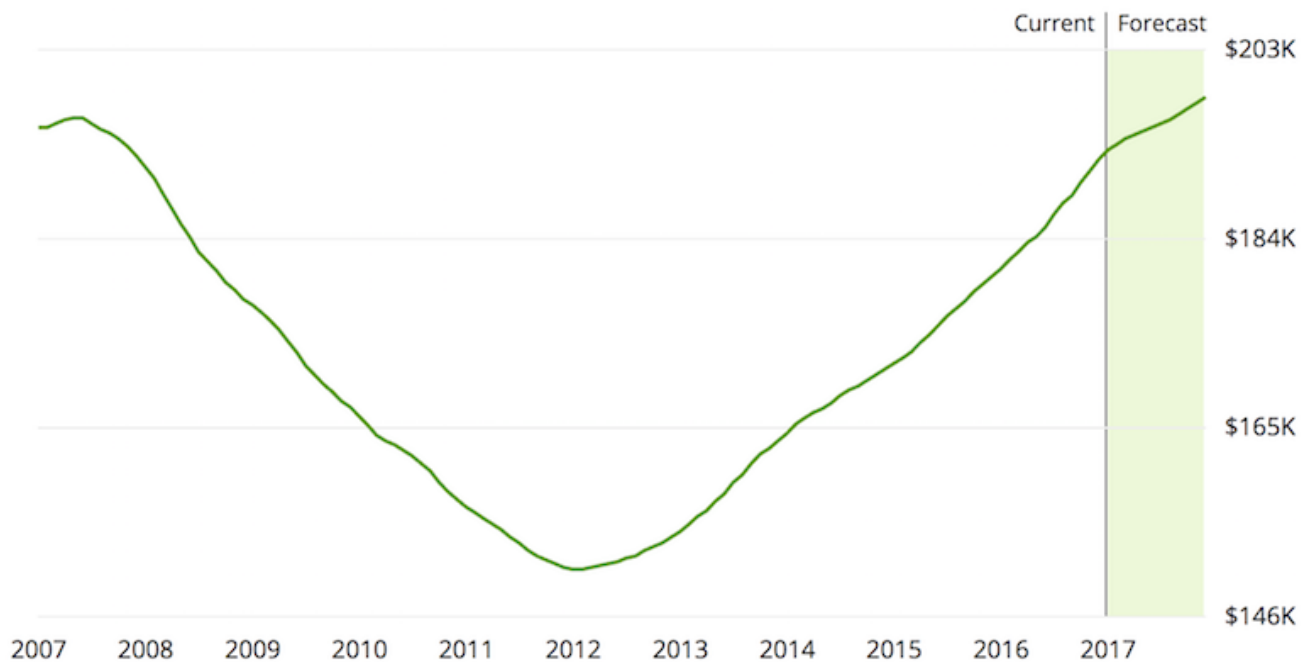
Out[43]:

```
0.09458023379383634
```

The interpretation is that almost 9.5% of the observations in our dataset were for properties with 2 beds and 2 baths.

## Example: housing market recession

According to data from Zillow, the lowest housing prices were from 2010 to end of 2013 (country-wide).



Create a new feature called 'during\_recession' to indicate if a transaction falls between 2010 and 2013

In [44]:

```
df['during_recession'] = ((df.tx_year >= 2010) & (df.tx_year <= 2013)).astype(int)
```

In [45]:

```
# Print percent of transactions where during_recession == 1
df[df['during_recession']==1].shape[0]/df.shape[0]
```

Out[45]:

```
0.2635494155154091
```

## 5.2 Interaction features

- Interaction features are operations between two or more other features.
- In some contexts, "interaction terms" must be products of two variables.
- In our context, interaction features can be products, sums, or differences between two features.

- For example, in our dataset, we know the transaction year and the year the property was built in.
- However, the more useful piece of information that combining these two features provides is the age of the property at the time of the transaction.

In [46]:

```
# Create a property age feature
df['property_age'] = df.tx_year - df.year_built
```

**Note:** 'property\_age' denotes the age of the property when it was sold and not how old it is today, since we want to predict the price at the time when the property is sold.

In [47]:

```
print(df.property_age.min())
```

-8

In [48]:

```
# Number of observations with 'property_age' < 0
print(sum(df.property_age < 0))
```

19

- On second thought, it's possible that some home owners bought houses before the construction company built them.
- This is not uncommon, especially with single-family homes.
- However, for this problem, **we are only interested in houses that already exist** because the REIT only buys existing ones!

In [49]:

```
# Remove rows where property_age is less than 0
df = df[df.property_age >= 0]
df.shape
```

Out[49]:

(1863, 29)

Now, let's add another interaction feature. How about the number of quality schools nearby?

- Well, we do know the number of schools nearby ('num\_schools')
- We also have their median quality score ('median\_schools')
- But what if it's the **interaction** of those two factors that's really important?
- In other words, what if it's good to have many school options, but only if they are good?

We can represent this with an interaction feature.

In [50]:

```
# Create a school score feature that num_schools * median_school
df['school_score'] = df.num_schools * df.median_school
```

In [51]:

```
# Display median school score
df.school_score.median()
```

Out[51]:

18.0

## 5.3 Handling Sparse Classes

- Sparse classes are those that have very few total observations.
- As mentioned earlier, they can be problematic for certain machine learning algorithms.
- At best, they are ignored.
- At worst, they can cause models to be overfit.

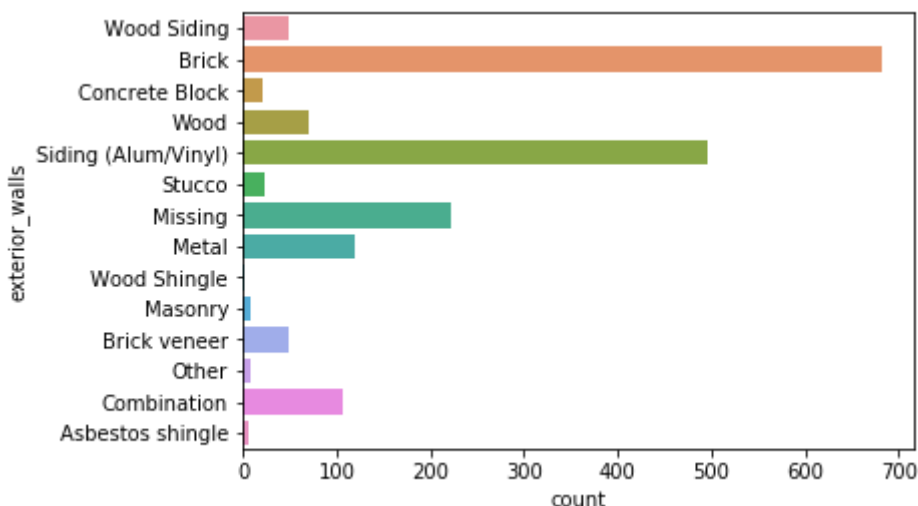
The easiest way to check for sparse classes is simply by plotting the distributions of your categorical features. We already did this during exploratory analysis, but since we've done some data cleaning since then, let's plot them again.

In [52]:

```
# Bar plot for exterior_walls
sns.countplot(y='exterior_walls', data=df)
```

Out[52]:

<matplotlib.axes.\_subplots.AxesSubplot at 0x291d6c7fdd8>



Group 'Wood Siding', 'Wood Shingle', and 'Wood' together. Label all of them as 'Wood'.

In [53]:

```
df.exterior_walls.replace(['Wood Siding', 'Wood Shingle', 'Wood'], 'Wood', inplace=True)
```

Next, we can group the remaining sparse classes into a single 'Other' class, even though there's already an 'Other' class.

Let's label 'Stucco', 'Other', 'Asbestos shingle', 'Concrete Block', and 'Masonry' as 'Other':

In [54]:

```
other_exterior_walls = ['Concrete Block', 'Stucco', 'Masonry', 'Other', 'Asbestos shingle']
df.exterior_walls.replace(other_exterior_walls, 'Other', inplace=True)
```

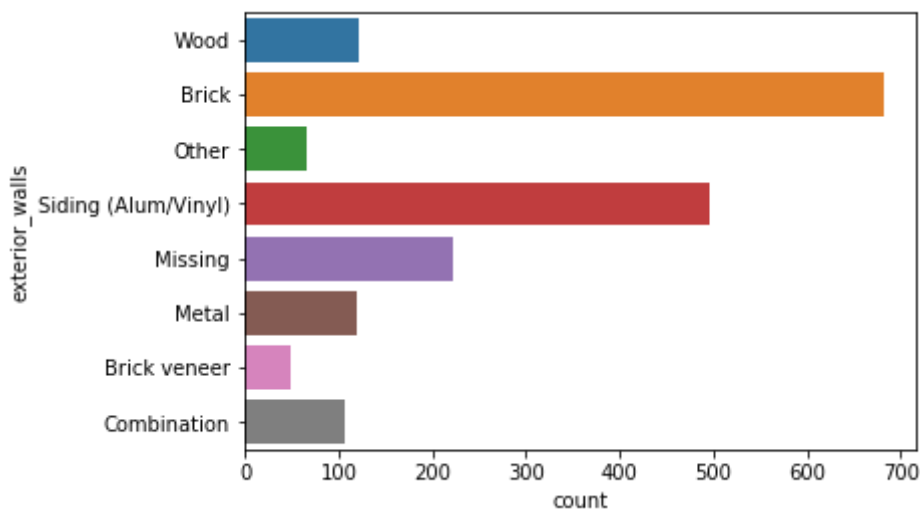
Finally, now that we've grouped together sparse classes, let's look at the bar plot for exterior walls again.

In [55]:

```
sns.countplot(y='exterior_walls', data=df)
```

Out[55]:

<matplotlib.axes.\_subplots.AxesSubplot at 0x291d6d0d048>



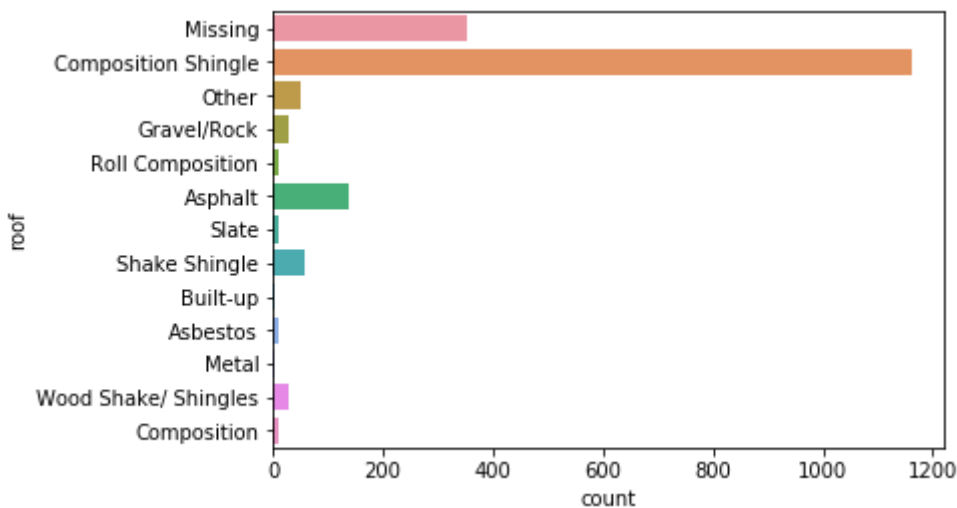
Similarly we check for 'roof'

In [56]:

```
sns.countplot(y='roof', data=df)
```

Out[56]:

<matplotlib.axes.\_subplots.AxesSubplot at 0x291d6066400>





Group 'Composition' and 'Wood Shake/ Shingles' into the 'Composition Shingle' class.

In [57]:

```
df.roof.replace(['Composition', 'Wood Shake/ Shingles'], 'Composition Shingle', inplace=True)
```

Next, let's group remaining sparse classes into a single 'Other' class.

Label 'Other', 'Gravel/Rock', 'Roll Composition', 'Slate', 'Built-up', 'Asbestos', and 'Metal' as 'Other'.

In [58]:

```
other_roof = ['Other', 'Gravel/Rock', 'Roll Composition', 'Slate', 'Built-up', 'Asbestos', 'Metal']
df.roof.replace(other_roof, 'Other', inplace=True)
```

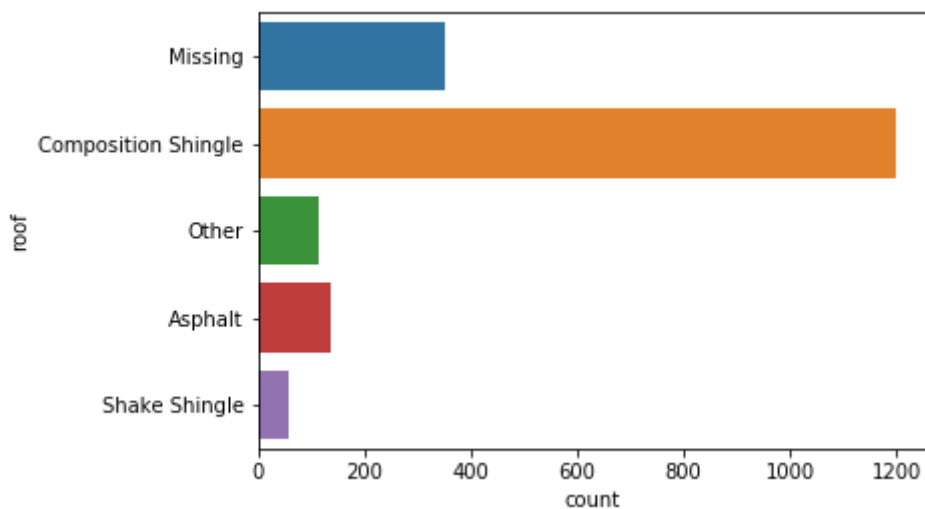
Finally, display bar plot again.

In [59]:

```
sns.countplot(y='roof', data=df)
```

Out[59]:

<matplotlib.axes.\_subplots.AxesSubplot at 0x291d64d2080>



Now these plots look much nicer

## 5.4 Encode dummy variables (One Hot Encoding)

- Machine learning algorithms cannot directly handle categorical features. Specifically, they cannot handle text values.
- Therefore, we need to create dummy variables for our categorical features.
- Dummy variables* are a set of binary (0 or 1) features that each represent a single class from a categorical feature.

In [60]:

```
# Create a new dataframe with dummy variables for for our categorical features.
df = pd.get_dummies(df, columns=['exterior_walls', 'roof', 'property_type'])
```

**Note:** There are many ways to perform one-hot encoding, you can also use LabelEncoder and OneHotEncoder classes in SKLEARN or use the above pandas function.

Let's display the first 5 rows again to see these new features

Click the table and scroll over to the right.

Check dummy variables created for 'exterior\_walls', 'roof' and 'property\_type'

In [61]:

```
df.head()
```

Out[61]:

	tx_price	beds	baths	sqft	year_built	lot_size	basement	restaurants	groceries
0	295850	1	1	584	2013	0	0.0	107	9
1	216500	1	1	612	1965	0	1.0	105	15
2	279900	1	1	615	1963	0	0.0	183	13
3	379900	1	1	618	2000	33541	0.0	198	9
4	340000	1	1	634	1992	0	0.0	149	7

## 5.5 Remove unused or redundant features

- Unused features are those that don't make sense to pass into our machine learning algorithms.
- Example ID columns, Features that wouldn't be available at the time of prediction (e.g. price of the property 5 years after the transaction), Other text descriptions, etc.
- For this dataset, we don't have any unused features.
- Redundant features would typically be those that have been **replaced by other features** that you've added.
- For example, since we used 'tx\_year' and 'year\_built' to create the 'property\_age' feature, we might consider removing them.

In [62]:

```
# Drop 'tx_year' and 'year_built' from the dataset
df = df.drop(['tx_year', 'year_built'], axis=1) ## axis=1 because we are dropping columns.
```

In [63]:

```
# Save analytical base table
df.to_csv('Files/analytical_base_table.csv', index=None)
```

## 6. Machine Learning Models

### 6.1 Data Preparation

In [64]:

```
df = pd.read_csv("Files/analytical_base_table.csv")
```

In [65]:

```
print(df.shape)
```

```
(1863, 40)
```

#### 6.1.1 Train and Test Splits

Separate your dataframe into separate objects for the target variable (y) and the input features (X) and perform the train and test split

In [66]:

```
# Create separate object for target variable  
y = df.tx_price  
# Create separate object for input features  
X = df.drop('tx_price', axis=1)
```

In [67]:

```
# Split X and y into train and test sets: 80-20  
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1234)
```

Let's confirm we have the right number of observations in each subset.

In [68]:

```
print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
```

```
(1490, 39) (373, 39) (1490,) (373,)
```

### 6.2 Data standardization

- In Data Standardization we perform zero mean centring and unit scaling; i.e. we make the mean of all the features as zero and the standard deviation as 1.
- Thus we use **mean** and **standard deviation** of each feature.
- It is very important to save the **mean** and **standard deviation** for each of the feature from the **training set**, because we use the same mean and standard deviation in the test set.

In [69]:

```
train_mean = X_train.mean()
train_std = X_train.std()
```

In [70]:

```
## Standardize the train data set
X_train = (X_train - train_mean) / train_std
```

In [71]:

```
## Check for mean and std dev.
X_train.describe()
```

Out[71]:

	beds	baths	sqft	lot_size	basement
<b>count</b>	1.490000e+03	1.490000e+03	1.490000e+03	1.490000e+03	1.490000e+03
<b>mean</b>	-1.902281e-16	-4.254613e-17	7.663519e-17	3.911860e-17	9.746119e-17
<b>std</b>	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00
<b>min</b>	-2.268801e+00	-1.697190e+00	-1.405276e+00	-3.662250e-01	-2.688343e+00
<b>25%</b>	-4.047185e-01	-6.224713e-01	-7.491974e-01	-3.219217e-01	3.717266e-01
<b>50%</b>	5.273226e-01	4.522474e-01	-3.155383e-01	-1.885809e-01	3.717266e-01
<b>75%</b>	5.273226e-01	4.522474e-01	5.334699e-01	-2.831904e-02	3.717266e-01
<b>max</b>	1.459364e+00	3.676403e+00	4.255036e+00	1.217405e+01	3.717266e-01

In [72]:

```
## Note: We use train_mean and train_std_dev to standardize test data set
X_test = (X_test - train_mean) / train_std
```

In [73]:

```
## Check for mean and std dev. - not exactly 0 and 1
X_test.describe()
```

Out[73]:

	beds	baths	sqft	lot_size	basement	restaurants	gro
count	373.000000	373.000000	373.000000	373.000000	373.000000	373.000000	373.
mean	-0.117360	-0.080790	-0.090918	-0.032233	0.010753	0.091066	0.14
std	0.958651	0.989343	1.001612	1.034313	0.988393	1.003586	0.99
min	-2.268801	-1.697190	-1.261108	-0.366225	-2.688343	-0.840593	-0.97
25%	-0.404719	-0.622471	-0.803935	-0.324824	0.371727	-0.627763	-0.75
50%	-0.404719	-0.622471	-0.386851	-0.266126	0.371727	-0.287235	-0.08
75%	0.527323	0.452247	0.306233	-0.062883	0.371727	0.500236	0.58
max	1.459364	3.676403	4.127830	12.149022	0.371727	4.820685	3.91

## 6.3 Modelling

### Model 1 - Baseline Model

- In this model, for every test data point, we will simply predict the average of the train labels as the output.
- We will use this simple model to perform hypothesis testing for other complex models.

In [74]:

```
## Predict Train results
y_train_pred = np.ones(y_train.shape[0])*y_train.mean()
```

In [75]:

```
## Predict Test results
y_pred = np.ones(y_test.shape[0])*y_train.mean()
```

In [76]:

```
print("Train Results for Baseline Model:")
print("*****")
print("Root mean squared error: ", sqrt(mse(y_train.values, y_train_pred)))
print("R-squared: ", r2_score(y_train.values, y_train_pred))
print("Mean Absolute Error: ", mae(y_train.values, y_train_pred))
```

Train Results for Baseline Model:

\*\*\*\*\*

Root mean squared error: 153791.70506675562

R-squared: 0.0

Mean Absolute Error: 127271.757171

In [77]:

```
print("Results for Baseline Model:")
print("*****")
print("Root mean squared error: ", sqrt(mse(y_test, y_pred)))
print("R-squared: ", r2_score(y_test, y_pred))
print("Mean Absolute Error: ", mae(y_test, y_pred))
```

Results for Baseline Model:

\*\*\*\*\*

Root mean squared error: 143268.37228905046

R-squared: -0.00731881601388

Mean Absolute Error: 120855.475979

## Model-2 Ridge Regression

In [78]:

```
tuned_params = {'alpha': [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000, 100000]}
model = GridSearchCV(Ridge(), tuned_params, scoring = 'neg_mean_absolute_error', cv=10,
n_jobs=-1)
model.fit(X_train, y_train)
```

Out[78]:

```
GridSearchCV(cv=10, error_score='raise',
            estimator=Ridge(alpha=1.0, copy_X=True, fit_intercept=True, max_iter=None,
            normalize=False, random_state=None, solver='auto', tol=0.001),
            fit_params=None, iid=True, n_jobs=-1,
            param_grid={'alpha': [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000, 100000]},
            pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
            scoring='neg_mean_absolute_error', verbose=0)
```

In [79]:

```
model.best_estimator_
```

Out[79]:

```
Ridge(alpha=100, copy_X=True, fit_intercept=True, max_iter=None,
      normalize=False, random_state=None, solver='auto', tol=0.001)
```

In [80]:

```
## Predict Train results
y_train_pred = model.predict(X_train)
```

In [81]:

```
## Predict Test results
y_pred = model.predict(X_test)
```

In [82]:

```
print("Train Results for Ridge Regression:")
print("*****")
print("Root mean squared error: ", sqrt(mse(y_train.values, y_train_pred)))
print("R-squared: ", r2_score(y_train.values, y_train_pred))
print("Mean Absolute Error: ", mae(y_train.values, y_train_pred))
```

Train Results for Ridge Regression:

\*\*\*\*\*

Root mean squared error: 119527.54751388216

R-squared: 0.395953748637

Mean Absolute Error: 93312.1223675

In [83]:

```
print("Test Results for Ridge Regression:")
print("*****")
print("Root mean squared error: ", sqrt(mse(y_test, y_pred)))
print("R-squared: ", r2_score(y_test, y_pred))
print("Mean Absolute Error: ", mae(y_test, y_pred))
```

Test Results for Ridge Regression:

\*\*\*\*\*

Root mean squared error: 109547.05458559998

R-squared: 0.411064541575

Mean Absolute Error: 84903.7040005

## Feature Importance

In [84]:

```
## Building the model again with the best hyperparameters
model = Ridge(alpha=1000)
model.fit(X_train, y_train)
```

Out[84]:

```
Ridge(alpha=1000, copy_X=True, fit_intercept=True, max_iter=None,
      normalize=False, random_state=None, solver='auto', tol=0.001)
```

In [85]:

```
indices = np.argsort(-abs(model.coef_))
print("The features in order of importance are:")
print(50*'-')
for feature in X.columns[indices]:
    print(feature)
```

The features in order of importance are:

-----

sqft  
insurance  
property\_tax  
college\_grad  
baths  
beds  
property\_type\_Apartment / Condo / Townhouse  
property\_type\_Single-Family  
lot\_size  
during\_recession  
exterior\_walls\_Brick veneer  
nightlife  
cafes  
active\_life  
exterior\_walls\_Wood  
two\_and\_two  
restaurants  
groceries  
exterior\_walls\_Missing  
num\_schools  
exterior\_walls\_Combination  
shopping  
beauty\_spas  
roof\_Composition Shingle  
roof\_Asphalt  
arts\_entertainment  
exterior\_walls\_Siding (Alum/Vinyl)  
exterior\_walls\_Brick  
median\_age  
median\_school  
roof\_Missing  
married  
basement  
school\_score  
exterior\_walls\_Metal  
roof\_Shake Shingle  
roof\_Other  
exterior\_walls\_Other  
property\_age

## Model-3 Support Vector Regression

With an RBF Kernel



In [86]:

```
tuned_params = {'C': [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000, 100000], 'gamma': [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000, 100000]}
model = GridSearchCV(SVR(), tuned_params, scoring = 'neg_mean_absolute_error', cv=5, n_jobs=-1)
model.fit(X_train, y_train)
```

Out[86]:

```
GridSearchCV(cv=5, error_score='raise',
             estimator=SVR(C=1.0, cache_size=200, coef0=0.0, degree=3, epsilon=0.1, gamma='auto',
                           kernel='rbf', max_iter=-1, shrinking=True, tol=0.001, verbose=False),
             fit_params=None, iid=True, n_jobs=-1,
             param_grid={'C': [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000, 100000], 'gamma': [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000, 100000]}},
             pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
             scoring='neg_mean_absolute_error', verbose=0)
```

In [87]:

```
model.best_estimator_
```

Out[87]:

```
SVR(C=100000, cache_size=200, coef0=0.0, degree=3, epsilon=0.1, gamma=0.0001,
    kernel='rbf', max_iter=-1, shrinking=True, tol=0.001, verbose=False)
```

In [88]:

```
## Building the model again with the best hyperparameters
model = SVR(C=100000, gamma=0.0001)
model.fit(X_train, y_train)
```

Out[88]:

```
SVR(C=100000, cache_size=200, coef0=0.0, degree=3, epsilon=0.1, gamma=0.0001,
    kernel='rbf', max_iter=-1, shrinking=True, tol=0.001, verbose=False)
```

In [89]:

```
## Predict Train results
y_train_pred = model.predict(X_train)
```

In [90]:

```
## Predict Test results
y_pred = model.predict(X_test)
```

In [91]:

```
print("Train Results for Support Vector Regression:")
print("*****")
print("Root mean squared error: ", sqrt(mse(y_train.values, y_train_pred)))
print("R-squared: ", r2_score(y_train.values, y_train_pred))
print("Mean Absolute Error: ", mae(y_train.values, y_train_pred))
```

```
Train Results for Support Vector Regression:
*****
Root mean squared error: 133691.95752049028
R-squared: 0.244308146054
Mean Absolute Error: 106379.145056
```

In [92]:

```
print("Test Results for Support Vector Regression:")
print("*****")
print("Root mean squared error: ", sqrt(mse(y_test, y_pred)))
print("R-squared: ", r2_score(y_test, y_pred))
print("Mean Absolute Error: ", mae(y_test, y_pred))
```

```
Test Results for Support Vector Regression:
*****
Root mean squared error: 121622.0808526227
R-squared: 0.274076005158
Mean Absolute Error: 97424.4827257
```

## Model-4 Random Forest Regression

In [132]:

```
## Reference for random search on random forest
## https://towardsdatascience.com/hyperparameter-tuning-the-random-forest-in-python-using-scikit-learn-28d2aa77dd74
tuned_params = {'n_estimators': [100, 200, 300, 400, 500], 'min_samples_split': [2, 5, 10], 'min_samples_leaf': [1, 2, 4]}
model = RandomizedSearchCV(RandomForestRegressor(), tuned_params, n_iter=20, scoring = 'neg_mean_absolute_error', cv=5, n_jobs=-1)
model.fit(X_train, y_train)
```

Out[132]:

```
RandomizedSearchCV(cv=5, error_score='raise',
                  estimator=RandomForestRegressor(bootstrap=True, criterion='mse',
max_depth=None,
                  max_features='auto', max_leaf_nodes=None,
                  min_impurity_decrease=0.0, min_impurity_split=None,
                  min_samples_leaf=1, min_samples_split=2,
                  min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=1,
                  oob_score=False, random_state=None, verbose=0, warm_start=False),
                  fit_params=None, iid=True, n_iter=20, n_jobs=-1,
                  param_distributions={'n_estimators': [100, 200, 300, 400, 500],
'min_samples_split': [2, 5, 10], 'min_samples_leaf': [1, 2, 4]},
                  pre_dispatch='2*n_jobs', random_state=None, refit=True,
                  return_train_score='warn', scoring='neg_mean_absolute_error',
                  verbose=0)
```

In [133]:

```
model.best_estimator_
```

Out[133]:

```
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None,
                      max_features='auto', max_leaf_nodes=None,
                      min_impurity_decrease=0.0, min_impurity_split=None,
                      min_samples_leaf=4, min_samples_split=10,
                      min_weight_fraction_leaf=0.0, n_estimators=300, n_jobs=1,
                      oob_score=False, random_state=None, verbose=0, warm_start=False)
e)
```

In [134]:

```
## Predict Train results
y_train_pred = model.predict(X_train)
```

In [135]:

```
## Predict Test results
y_pred = model.predict(X_test)
```

In [136]:

```
print("Train Results for Random Forest Regression:")
print("*****")
print("Root mean squared error: ", sqrt(mse(y_train.values, y_train_pred)))
print("R-squared: ", r2_score(y_train.values, y_train_pred))
print("Mean Absolute Error: ", mae(y_train.values, y_train_pred))
```

```
Train Results for Random Forest Regression:
*****
Root mean squared error:  66319.06809566794
R-squared:  0.814043653035
Mean Absolute Error:  48496.7163274
```

In [137]:

```
print("Test Results for Random Forest Regression:")
print("*****")
print("Root mean squared error: ", sqrt(mse(y_test, y_pred)))
print("R-squared: ", r2_score(y_test, y_pred))
print("Mean Absolute Error: ", mae(y_test, y_pred))
```

```
Test Results for Random Forest Regression:
*****
Root mean squared error:  94244.78734368536
R-squared:  0.564105920359
Mean Absolute Error:  69317.3029288
```

## Feature Importance

In [99]:

```
## Building the model again with the best hyperparameters
model = RandomForestRegressor(n_estimators=300, min_samples_split=10, min_samples_leaf=
4)
model.fit(X_train, y_train)
```

Out[99]:

```
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None,
    max_features='auto', max_leaf_nodes=None,
    min_impurity_decrease=0.0, min_impurity_split=None,
    min_samples_leaf=4, min_samples_split=10,
    min_weight_fraction_leaf=0.0, n_estimators=300, n_jobs=1,
    oob_score=False, random_state=None, verbose=0, warm_start=False)
```

e)

In [100]:

```
indices = np.argsort(-model.feature_importances_)
print("The features in order of importance are:")
print(50*'-' )
for feature in X.columns[indices]:
    print(feature)
```

The features in order of importance are:

```
-----
insurance
property_tax
property_age
sqft
college_grad
lot_size
median_age
during_recession
married
beauty_spas
active_life
shopping
restaurants
school_score
baths
median_school
arts_entertainment
nightlife
groceries
cafes
beds
exterior_walls_Siding (Alum/Vinyl)
exterior_walls_Brick
roof_Composition Shingle
exterior_walls_Metal
roof_Missing
exterior_walls_Missing
roof_Shake Shingle
roof_Asphalt
num_schools
exterior_walls_Brick veneer
exterior_walls_Combination
property_type_Apartment / Condo / Townhouse
basement
property_type_Single-Family
two_and_two
roof_Other
exterior_walls_Wood
exterior_walls_Other
```

## 6.6 Model-5 XGBoost Regression

In [126]:

```
## Reference for random search on xgboost
## https://gist.github.com/wrwr/3f6b66bf4ee01bf48be965f60d14454d
tuned_params = {'max_depth': [1, 2, 3, 4, 5], 'learning_rate': [0.01, 0.05, 0.1], 'n_estimators': [100, 200, 300, 400, 500], 'reg_lambda': [0.001, 0.1, 1.0, 10.0, 100.0]}
model = RandomizedSearchCV(XGBRegressor(), tuned_params, n_iter=20, scoring = 'neg_mean_absolute_error', cv=5, n_jobs=-1)
model.fit(X_train, y_train)
```

Out[126]:

```
RandomizedSearchCV(cv=5, error_score='raise',
                  estimator=XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
                  colsample_bytree=1, gamma=0, learning_rate=0.1, max_delta_step=0,
                  max_depth=3, min_child_weight=1, missing=None, n_estimators=100,
                  n_jobs=1, nthread=None, objective='reg:linear', random_state=0,
                  reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
                  silent=True, subsample=1),
                  fit_params=None, iid=True, n_iter=20, n_jobs=-1,
                  param_distributions={'max_depth': [1, 2, 3, 4, 5], 'learning_rate': [0.01, 0.05, 0.1], 'n_estimators': [100, 200, 300, 400, 500], 'reg_lambda': [0.001, 0.1, 1.0, 10.0, 100.0]},
                  pre_dispatch='2*n_jobs', random_state=None, refit=True,
                  return_train_score='warn', scoring='neg_mean_absolute_error',
                  verbose=0)
```

In [127]:

```
model.best_estimator_
```

Out[127]:

```
XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
             colsample_bytree=1, gamma=0, learning_rate=0.05, max_delta_step=0,
             max_depth=3, min_child_weight=1, missing=None, n_estimators=200,
             n_jobs=1, nthread=None, objective='reg:linear', random_state=0,
             reg_alpha=0, reg_lambda=1.0, scale_pos_weight=1, seed=None,
             silent=True, subsample=1)
```

In [128]:

```
## Predict Train results
y_train_pred = model.predict(X_train)
```

In [129]:

```
## Predict Test results
y_pred = model.predict(X_test)
```

In [130]:

```
print("Train Results for XGBoost Regression:")
print("*****")
print("Root mean squared error: ", sqrt(mse(y_train.values, y_train_pred)))
print("R-squared: ", r2_score(y_train.values, y_train_pred))
print("Mean Absolute Error: ", mae(y_train.values, y_train_pred))
```

Train Results for XGBoost Regression:

\*\*\*\*\*

Root mean squared error: 86656.24315626375

R-squared: 0.682507193329

Mean Absolute Error: 65516.7485319

In [131]:

```
print("Test Results for XGBoost Regression:")
print("*****")
print("Root mean squared error: ", sqrt(mse(y_test, y_pred)))
print("R-squared: ", r2_score(y_test, y_pred))
print("Mean Absolute Error: ", mae(y_test, y_pred))
```

Test Results for XGBoost Regression:

\*\*\*\*\*

Root mean squared error: 96067.81938302648

R-squared: 0.54707931179

Mean Absolute Error: 70163.0759886

## Feature Importance

In [107]:

```
## Building the model again with the best hyperparameters
model = XGBRegressor(max_depth=4, learning_rate=0.1, n_estimators=400, reg_lambda=0.1)
model.fit(X_train, y_train)
```

Out[107]:

```
XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
             colsample_bytree=1, gamma=0, learning_rate=0.1, max_delta_step=0,
             max_depth=4, min_child_weight=1, missing=None, n_estimators=400,
             n_jobs=1, nthread=None, objective='reg:linear', random_state=0,
             reg_alpha=0, reg_lambda=0.1, scale_pos_weight=1, seed=None,
             silent=True, subsample=1)
```

In [108]:

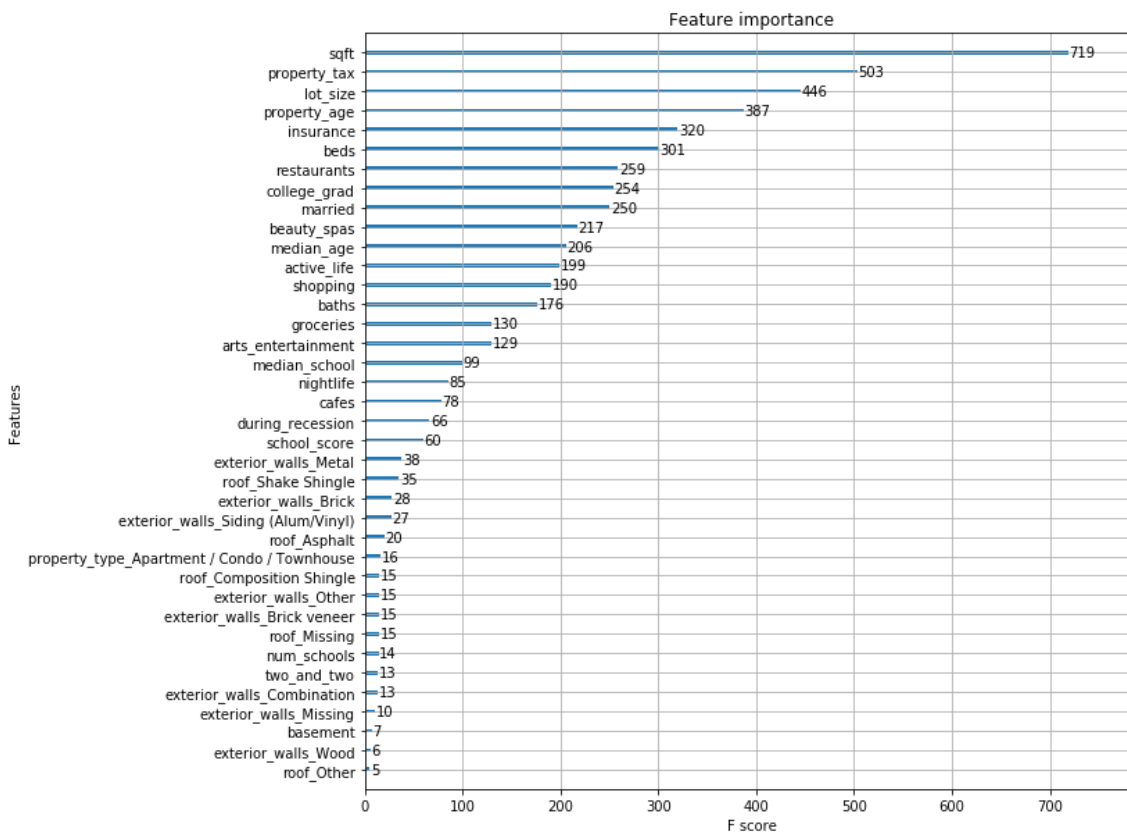
```
## Function to include figsize parameter
## Reference: https://stackoverflow.com/questions/40081888/xgboost-plot-importance-figure-size
def my_plot_importance(booster, figsize, **kwargs):
    from matplotlib import pyplot as plt
    from xgboost import plot_importance
    fig, ax = plt.subplots(1,1,figsize=figsize)
    return plot_importance(booster=booster, ax=ax, **kwargs)
```

In [109]:

```
my_plot_importance(model, (10,10))
```

Out[109]:

<matplotlib.axes.\_subplots.AxesSubplot at 0x291d61807f0>



## Model-6 KNN Regression

In [110]:

```
from sklearn.neighbors import KNeighborsRegressor
tuned_params = {'n_neighbors': range(2,100)}
model = GridSearchCV(KNeighborsRegressor(), tuned_params, scoring = 'neg_mean_absolute_error', cv=5, n_jobs=-1)
model.fit(X_train, y_train)
```

Out[110]:

```
GridSearchCV(cv=5, error_score='raise',
             estimator=KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='minkowski',
             metric_params=None, n_jobs=1, n_neighbors=5, p=2,
             weights='uniform'),
             fit_params=None, iid=True, n_jobs=-1,
             param_grid={'n_neighbors': range(2, 100)}, pre_dispatch='2*n_jobs',
             refit=True, return_train_score='warn',
             scoring='neg_mean_absolute_error', verbose=0)
```



In [111]:

```
model.best_estimator_
```

Out[111]:

```
KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='minkowski',
                    metric_params=None, n_jobs=1, n_neighbors=12, p=2,
                    weights='uniform')
```

In [112]:

```
## Predict Train results
y_train_pred = model.predict(X_train)
```

In [113]:

```
## Predict Test results
y_pred = model.predict(X_test)
```

In [114]:

```
print("Train Results for KNN Regression:")
print("*****")
print("Root mean squared error: ", sqrt(mse(y_train.values, y_train_pred)))
print("R-squared: ", r2_score(y_train.values, y_train_pred))
print("Mean Absolute Error: ", mae(y_train.values, y_train_pred))
```

Train Results for KNN Regression:

\*\*\*\*\*

Root mean squared error: 115747.48626601043

R-squared: 0.433555567874

Mean Absolute Error: 90732.0924497

In [115]:

```
print("Test Results for KNN Regression:")
print("*****")
print("Root mean squared error: ", sqrt(mse(y_test, y_pred)))
print("R-squared: ", r2_score(y_test, y_pred))
print("Mean Absolute Error: ", mae(y_test, y_pred))
```

Test Results for KNN Regression:

\*\*\*\*\*

Root mean squared error: 109868.13784675737

R-squared: 0.407607133411

Mean Absolute Error: 86279.5444593

## Model-7 Lasso Regression

In [138]:

```
tuned_params = {'alpha': [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000, 100000]}
model = GridSearchCV(Lasso(), tuned_params, scoring = 'neg_mean_absolute_error', cv=10,
    n_jobs=-1)
model.fit(X_train, y_train)
```

Out[138]:

```
GridSearchCV(cv=10, error_score='raise',
    estimator=Lasso(alpha=1.0, copy_X=True, fit_intercept=True, max_iter=1000,
    normalize=False, positive=False, precompute=False, random_state=None,
    selection='cyclic', tol=0.0001, warm_start=False),
    fit_params=None, iid=True, n_jobs=-1,
    param_grid={'alpha': [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000, 100000]},
    pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
    scoring='neg_mean_absolute_error', verbose=0)
```

In [139]:

```
model.best_estimator_
```

Out[139]:

```
Lasso(alpha=1000, copy_X=True, fit_intercept=True, max_iter=1000,
    normalize=False, positive=False, precompute=False, random_state=None,
    selection='cyclic', tol=0.0001, warm_start=False)
```

In [140]:

```
## Predict Train results
y_train_pred = model.predict(X_train)
```

In [141]:

```
## Predict Test results
y_pred = model.predict(X_test)
```

In [143]:

```
print("Train Results for Lasso Regression:")
print("*****")
print("Root mean squared error: ", sqrt(mse(y_train.values, y_train_pred)))
print("R-squared: ", r2_score(y_train.values, y_train_pred))
print("Mean Absolute Error: ", mae(y_train.values, y_train_pred))
```

Train Results for Lasso Regression:

\*\*\*\*\*

Root mean squared error: 119645.16203019097

R-squared: 0.394764406716

Mean Absolute Error: 93505.4315353

In [144]:

```
print("Test Results for Lasso Regression:")
print("*****")
print("Root mean squared error: ", sqrt(mse(y_test, y_pred)))
print("R-squared: ", r2_score(y_test, y_pred))
print("Mean Absolute Error: ", mae(y_test, y_pred))
```

Test Results for Lasso Regression:

\*\*\*\*\*

Root mean squared error: 109600.69393795903

R-squared: 0.410487659783

Mean Absolute Error: 84704.4651886

## Feature Importance

In [126]:

```
## Building the model again with the best hyperparameters
model = Lasso(alpha=1000)
model.fit(X_train, y_train)
```

Out[126]:

```
Lasso(alpha=1000, copy_X=True, fit_intercept=True, max_iter=1000,
      normalize=False, positive=False, precompute=False, random_state=None,
      selection='cyclic', tol=0.0001, warm_start=False)
```

In [127]:

```
indices = np.argsort(-abs(model.coef_))
print("The features in order of importance are:")
print(50*'-')
for feature in X.columns[indices]:
    print(feature)
```

The features in order of importance are:

```
-----
sqft
insurance
college_grad
baths
beds
during_recession
property_type_Apartment / Condo / Townhouse
cafes
exterior_walls_Wood
groceries
school_score
lot_size
restaurants
nightlife
beauty_spas
property_tax
exterior_walls_Brick veneer
shopping
property_age
exterior_walls_Combination
exterior_walls_Brick
two_and_two
roof_Composition Shingle
exterior_walls_Missing
roof_Asphalt
num_schools
active_life
median_age
property_type_Single-Family
married
exterior_walls_Metal
exterior_walls_Other
exterior_walls_Siding (Alum/Vinyl)
arts_entertainment
basement
roof_Missing
roof_Other
roof_Shake Shingle
median_school
```

## Model-8 Decision Tree Regression

In [118]:

```
from sklearn.tree import DecisionTreeRegressor
tuned_params = {'max_depth': range(2,1000)}
model = GridSearchCV(DecisionTreeRegressor(), tuned_params, scoring = 'neg_mean_absolut
e_error', cv=10, n_jobs=-1)
model.fit(X_train, y_train)
```

Out[118]:

```
GridSearchCV(cv=10, error_score='raise',
             estimator=DecisionTreeRegressor(criterion='mse', max_depth=None, ma
x_features=None,
             max_leaf_nodes=None, min_impurity_decrease=0.0,
             min_impurity_split=None, min_samples_leaf=1,
             min_samples_split=2, min_weight_fraction_leaf=0.0,
             presort=False, random_state=None, splitter='best'),
             fit_params=None, iid=True, n_jobs=-1,
             param_grid={'max_depth': range(2, 1000)}, pre_dispatch='2*n_jobs',
             refit=True, return_train_score='warn',
             scoring='neg_mean_absolute_error', verbose=0)
```

In [119]:

```
model.best_estimator_
```

Out[119]:

```
DecisionTreeRegressor(criterion='mse', max_depth=4, max_features=None,
                      max_leaf_nodes=None, min_impurity_decrease=0.0,
                      min_impurity_split=None, min_samples_leaf=1,
                      min_samples_split=2, min_weight_fraction_leaf=0.0,
                      presort=False, random_state=None, splitter='best')
```

In [120]:

```
## Predict Train results
y_train_pred = model.predict(X_train)
```

In [121]:

```
## Predict Test results
y_pred = model.predict(X_test)
```

In [122]:

```
print("Train Results for Decision Tree Regression:")
print("*****")
print("Root mean squared error: ", sqrt(mse(y_train.values, y_train_pred)))
print("R-squared: ", r2_score(y_train.values, y_train_pred))
print("Mean Absolute Error: ", mae(y_train.values, y_train_pred))
```

```
Train Results for Decision Tree Regression:
*****
```

```
Root mean squared error: 104690.25034604006
R-squared: 0.536610004923
Mean Absolute Error: 78413.798199
```

In [123]:

```
print("Test Results for Decision Tree Regression:")
print("*****")
print("Root mean squared error: ", sqrt(mse(y_test, y_pred)))
print("R-squared: ", r2_score(y_test, y_pred))
print("Mean Absolute Error: ", mae(y_test, y_pred))
```

```
Test Results for Decision Tree Regression:
*****
Root mean squared error: 101455.55435878348
R-squared: 0.49485280098
Mean Absolute Error: 76563.0867353
```

## 7. Save the winning model to disk

***Win condition: Avg. prediction error \$69,317.3029 < \$70,000 has been achieved using Random Forest Regression***

In [145]:

```
win_model = RandomForestRegressor(n_estimators=300, min_samples_split=10, min_samples_leaf=4)
win_model.fit(X_train, y_train)
joblib.dump(win_model, 'Save/rfr_real_estate.pkl')
```

Out[145]:

```
['Save/rfr_real_estate.pkl']
```

## 8. Final Results Tabulation

Sr. No.	Model	Root Mean Squared Error		R Squared		Mean absolute Error	
		Train	Test	Train	Test	Train	Test
1.	Baseline Model	153791.7050	143268.3722	0.0000	-0.0073	127271.7571	120855.4759
2.	Ridge Regression	119527.5475	109547.0545	0.3959	0.4110	93312.1223	84903.7040
3.	Support Vector Regression	133691.9575	121622.0808	0.2443	0.2740	106379.1450	97424.4827
4.	<b>Random Forest Regression</b>	<b>66319.0680</b>	<b>94244.7873</b>	<b>0.8140</b>	<b>0.5641</b>	<b>48496.7163</b>	<b>69317.3029</b>
5.	XGBoost Regression	86656.2431	96067.8193	0.6825	0.5470	65516.7483	70163.0759
6.	KNN Regression	115747.4862	109868.1378	0.4335	0.4076	90732.0924	86279.5444
7.	Lasso Regression	119645.1620	109600.6939	0.3947	0.4104	93505.4315	84704.4651
8.	Decision Tree Regression	104690.2503	101455.5543	0.5366	0.4948	78413.7981	76563.0867