Stacked_Ensemble_Regression

February 28, 2024

[1]:
```
import pandas as pd
import numpy as np
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
from scipy.stats import norm, probplot, boxcox, skew, kurtosis, shapiro
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
from sklearn.metrics import mean_squared_error
from skopt import BayesSearchCV
from sklearn.linear_model import Lasso, ElasticNet
from sklearn.kernel_ridge import KernelRidge
from sklearn.model_selection import cross_val_score
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import RobustScaler
import warnings
from time import time
import pprint
from xgboost import XGBRegressor
from catboost import CatBoostRegressor
from sklearn.base import BaseEstimator, RegressorMixin, TransformerMixin
from sklearn.utils.validation import check_is_fitted
from sklearn.base import clone
from skopt.space import Real, Integer
from itertools import combinations
from joblib import Parallel, delayed
from sklearn.linear_model import LinearRegression
from skopt.callbacks import DeadlineStopper, DeltaYStopper
from sklearn.metrics import make_scorer, mean_squared_error
from functools import partial
from scipy.special import inv_boxcox
from itertools import combinations
xgb.set_config(verbosity=0)
np.int = int # To avoid error `np.int` was a deprecated alias for the builtin␣
↪`int`
```

[44]:
```python
df = pd.read_csv("data.csv")
```

[45]:
```python
df.head()
```

[45]:

	date	price	bedrooms	bathrooms	sqft_living	sqft_lot
0	2014-05-02 00:00:00	313000.0	3.0	1.50	1340	7912
1	2014-05-02 00:00:00	2384000.0	5.0	2.50	3650	9050
2	2014-05-02 00:00:00	342000.0	3.0	2.00	1930	11947
3	2014-05-02 00:00:00	420000.0	3.0	2.25	2000	8030
4	2014-05-02 00:00:00	550000.0	4.0	2.50	1940	10500

	floors	waterfront	view	condition	sqft_above	sqft_basement	yr_built
0	1.5	0	0	3	1340	0	1955
1	2.0	0	4	5	3370	280	1921
2	1.0	0	0	4	1930	0	1966
3	1.0	0	0	4	1000	1000	1963
4	1.0	0	0	4	1140	800	1976

	yr_renovated	street	city	statezip	country
0	2005	18810 Densmore Ave N	Shoreline	WA 98133	USA
1	0	709 W Blaine St	Seattle	WA 98119	USA
2	0	26206-26214 143rd Ave SE	Kent	WA 98042	USA
3	0	857 170th Pl NE	Bellevue	WA 98008	USA
4	1992	9105 170th Ave NE	Redmond	WA 98052	USA

[47]:
```python
df.drop(columns = ['street','city','statezip','country', 'date'], inplace =
 True)
df.head()
```

[47]:

	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront
0	313000.0	3.0	1.50	1340	7912	1.5	0
1	2384000.0	5.0	2.50	3650	9050	2.0	0
2	342000.0	3.0	2.00	1930	11947	1.0	0
3	420000.0	3.0	2.25	2000	8030	1.0	0
4	550000.0	4.0	2.50	1940	10500	1.0	0

	view	condition	sqft_above	sqft_basement	yr_built	yr_renovated
0	0	3	1340	0	1955	2005
1	4	5	3370	280	1921	0
2	0	4	1930	0	1966	0
3	0	4	1000	1000	1963	0
4	0	4	1140	800	1976	1992

1 Data Processing

[42]:
```
df.dtypes
```

[42]:
```
price            float64
bedrooms         float64
bathrooms        float64
sqft_living        int64
sqft_lot           int64
floors           float64
waterfront         int64
view               int64
condition          int64
sqft_above         int64
sqft_basement      int64
yr_built           int64
yr_renovated       int64
dtype: object
```

As the range is between -2,147,483,648 and 2,147,483,647 we will convert int64 and float64 to their respective 32-bit format

[40]:
```
int_columns = ['bedrooms', 'price']

df[int_columns] = df[int_columns].astype('int32')

for col in df.select_dtypes(include=['int64']).columns:
    # Convert int64 columns to int32
    df[col] = df[col].astype('int32')

for col in df.select_dtypes(include=['float64']).columns:
    # Convert float64 columns to float32
    df[col] = df[col].astype('float32')
```

[42]:
```
df.isna().sum()
```

[42]:
```
price            0
bedrooms         0
bathrooms        0
sqft_lot         0
floors           0
waterfront       0
view             0
condition        0
sqft_above       0
sqft_basement    0
yr_built         0
yr_renovated     0
```

```
total_sqft       0
dtype: int64
```

The absence of missing values is advantageous as it eliminates the need for methods like multiple imputation or mean filling, which could potentially introduce additional variance to the data

[5]:
```
df['total_sqft'] = df['sqft_living'] + df['sqft_above'] + df['sqft_basement']
```

Although sqft_lot exists, it shows the entire area of the land and not of the house, so we create a new feature and we will test it's correlation with the target variable afterwards, should be high as big houses usually are more expensive

1.1 Target variable analysis

[48]:
```
# Check for instances with negative or zero prices
instances_with_negative_or_zero_price = df[(df['price'] <= 0)]

if not instances_with_negative_or_zero_price.empty:
    print("There are instances with negative or zero prices:")
    print(instances_with_negative_or_zero_price.head()) #head() to save in pdf␣
 ↪format
else:
    print("There are no instances with negative or zero prices.")
```

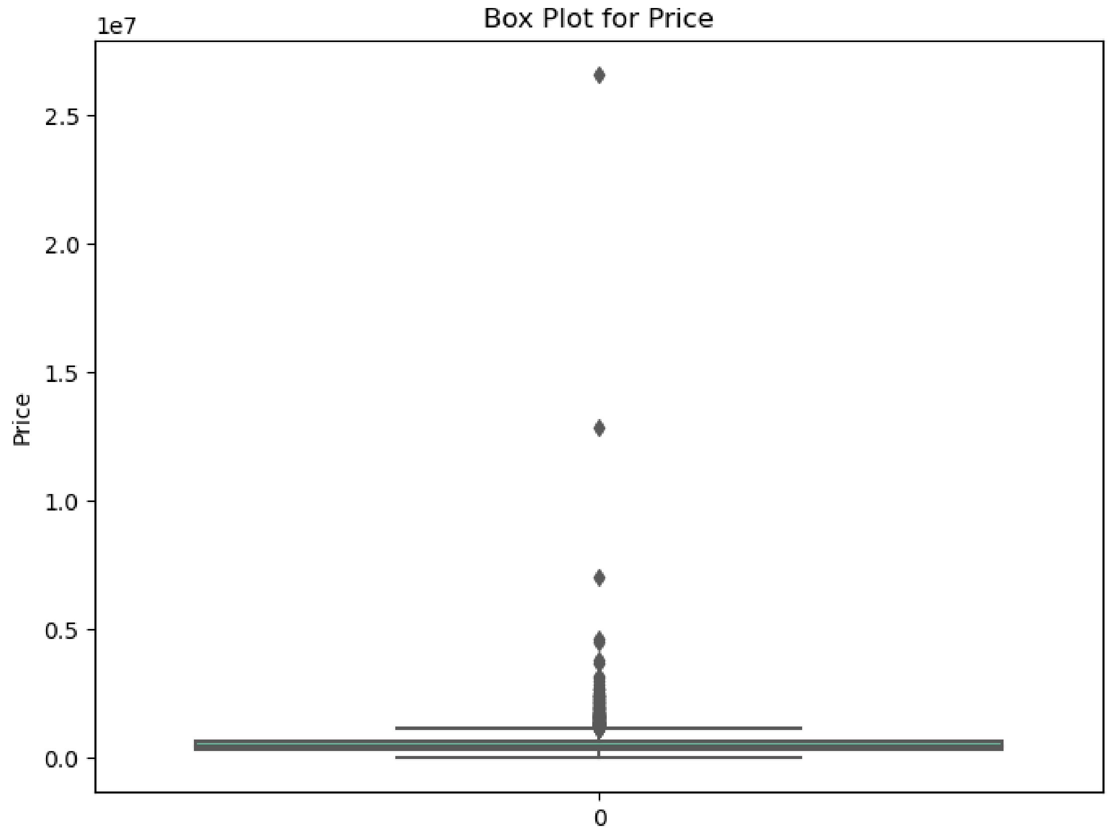
```
There are instances with negative or zero prices:
      price  bedrooms  bathrooms  sqft_living  sqft_lot  floors  waterfront  \
4354    0.0       3.0       1.75         1490     10125     1.0           0
4356    0.0       4.0       2.75         2600      5390     1.0           0
4357    0.0       6.0       2.75         3200      9200     1.0           0
4358    0.0       5.0       3.50         3480     36615     2.0           0
4361    0.0       5.0       1.50         1500      7112     1.0           0

      view  condition  sqft_above  sqft_basement  yr_built  yr_renovated
4354     0          4        1490              0      1962             0
4356     0          4        1300           1300      1960          2001
4357     2          4        1600           1600      1953          1983
4358     0          4        2490            990      1983             0
4361     0          5         760            740      1920             0
```

There are not negative prices but there are several houses with price set to zero, we can delete these instances.

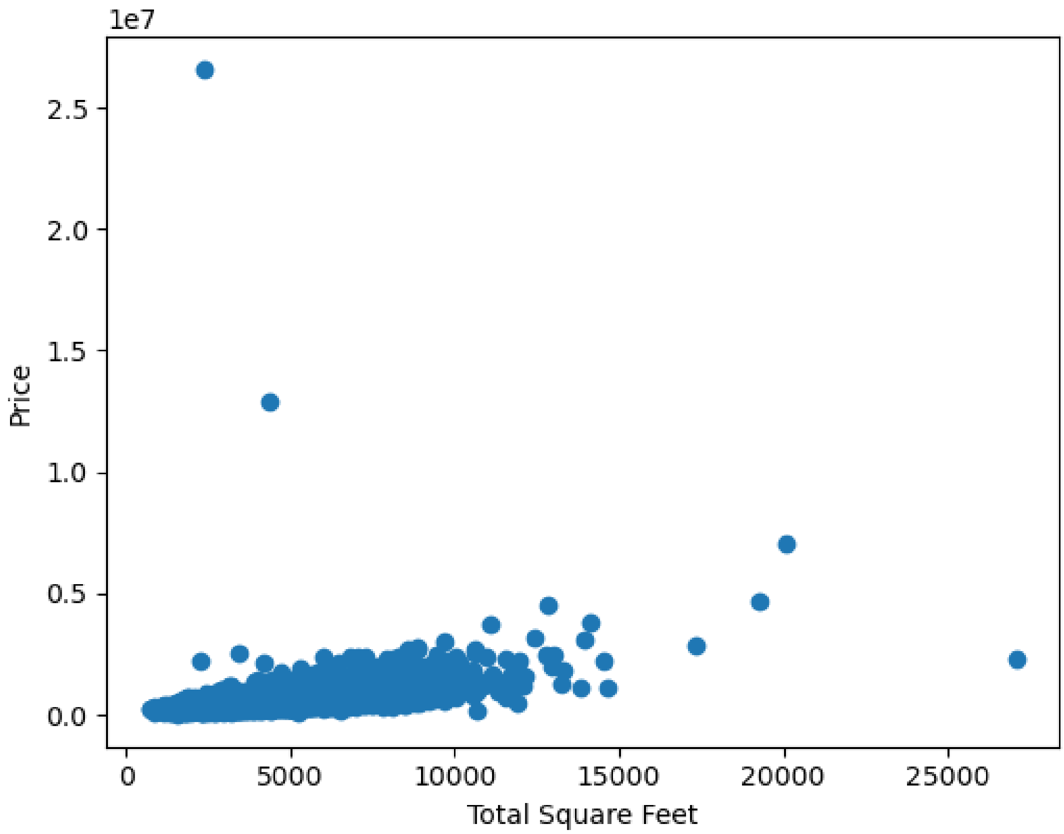
[6]:
```
indices_to_drop = df[df['price'] == 0].index

df.drop(indices_to_drop, inplace=True)
```

[85]:
```
plt.figure(figsize=(8, 6))
sns.boxplot(data=df['price'], orient='v', palette='Set2')
plt.title('Box Plot for Price')
```

```
plt.ylabel('Price')
plt.show()
```



[86]:
```
plt.scatter(x=df['total_sqft'], y=df['price'])
plt.ylabel('Price')
plt.xlabel('Total Square Feet')
plt.show()
```



Here there are 3 outliers that don't make sense, first two very expensive houses with very little square feets, and a house with a lot of total square feets but a very low price. Let's check into them.

[43]:
```
# Get the rows with the 2 highest values in the 'price' column
top_2_highest = df.nlargest(2, 'price')

print("Instances with the highest price:")

print(top_2_highest)
```

```
Instances with the highest price:
        price  bedrooms  bathrooms  sqft_living  sqft_lot  floors  waterfront  \
2286  7062500         5       4.50        10040     37325     2.0           1
2654  4668000         5       6.75         9640     13068     1.0           1

      view  condition  sqft_above  sqft_basement  yr_built  yr_renovated  \
2286     2          3        7680           2360      1940          2001
2654     4          3        4820           4820      1983          2009

      total_sqft
```

```
2286        20080
2654        19280
```

As we can see the only value that could perhaps justify the highest data entry is it's high sqft_lot

[14]:
```
df['sqft_lot'].describe()
```

[14]:
```
count    4.551000e+03
mean     1.483528e+04
std      3.596408e+04
min      6.380000e+02
25%      5.000000e+03
50%      7.680000e+03
75%      1.097800e+04
max      1.074218e+06
Name: sqft_lot, dtype: float64
```

We can confidently remove the instance with a sqft_lot exceeding the 50% threshold since it doesn't seem justifiable for its price. Additionally, it's safe to drop the second highest instance as its specifications don't match the price of the house.

[7]:
```
instances_with_highest_price = df[(df['price'] > 12e6) & (df['total_sqft'] <
 ↪5000)].index

df.drop(instances_with_highest_price, inplace=True)
```

Now for the entry with the highest total_sqft

[24]:
```
top_2_highest_sqft = df.nlargest(2, 'total_sqft')

print("Instances with the highest Total_sqft:")
print(top_2_highest_sqft)
```

```
Instances with the highest Total_sqft:
        price  bedrooms  bathrooms  sqft_living  sqft_lot  floors  waterfront  \
2286  7062500         5       4.50        10040     37325     2.0           1
2654  4668000         5       6.75         9640     13068     1.0           1

      view  condition  sqft_above  sqft_basement  yr_built  yr_renovated  \
2286     2          3        7680           2360      1940          2001
2654     4          3        4820           4820      1983          2009

      total_sqft
2286       20080
2654       19280
```

[8]:
```
indice_to_drop = df[(df['total_sqft'] > 27000) & (df['price'] < 2.5e6)].index

df.drop(indice_to_drop, inplace=True)
```

Now to check for very low prices that may be an entry mistake

[89]:
```
# Instances with a price less than 1 million
filtered_df = df[df['price'] < 1e6]

sorted_df = filtered_df.sort_values(by='price', ascending=True)

sorted_df.head()
```

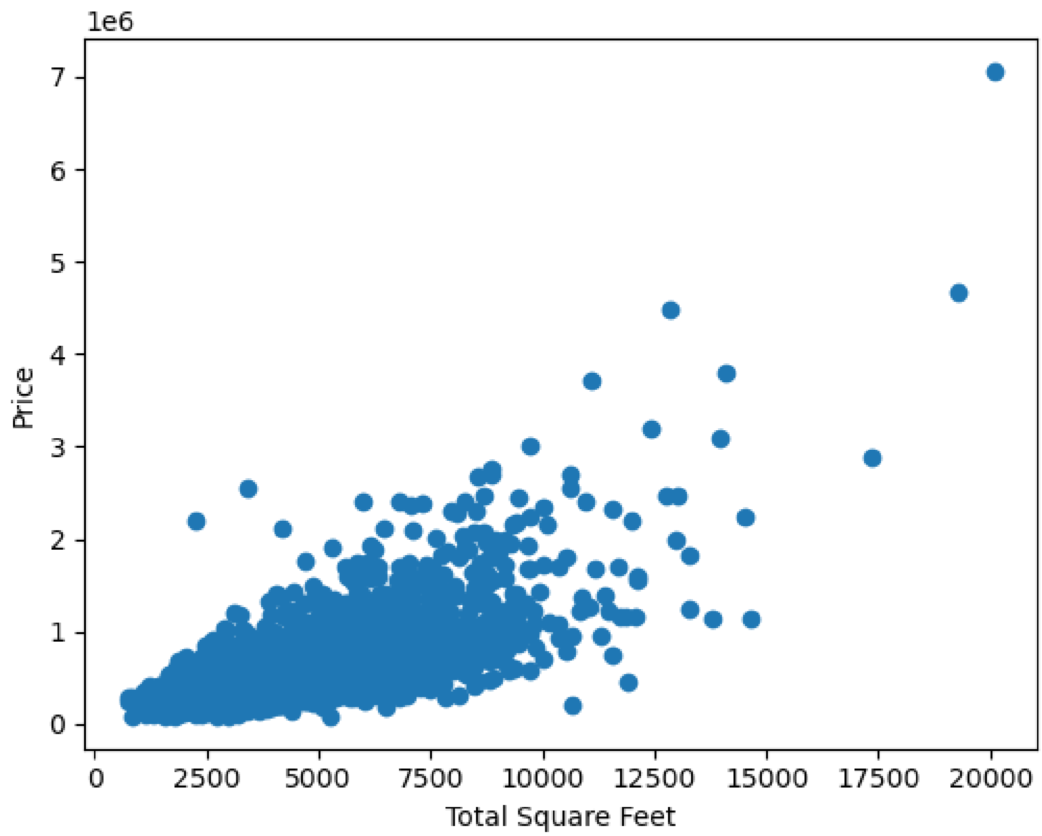
[89]:
```
       price  bedrooms  bathrooms  sqft_living  sqft_lot  floors  waterfront  \
4351    7800         2       1.00          780     16344     1.0           0
1219   80000         1       0.75          430      5050     1.0           0
1587   83000         2       1.00          900      8580     1.0           0
4407   83300         3       2.00         1490      7770     1.0           0
4415   83300         3       2.00         1370     78408     1.0           0

      view  condition  sqft_above  sqft_basement  yr_built  yr_renovated  \
4351     0          1         780              0      1942             0
1219     0          2         430              0      1912             0
1587     0          3         900              0      1918             0
4407     0          4        1490              0      1990             0
4415     0          5        1370              0      1964             0

      total_sqft
4351        1560
1219         860
1587        1800
4407        2980
4415        2740
```

As you can see there is a house with a price of 7800, we can also safely delete that instance

[9]:
```
# Indice of row with price less than 7900
low_price_instance = df[df['price'] < 7900].index

df.drop(low_price_instance, inplace=True)
```

Let's check the scatterplot again

[91]:
```
plt.scatter(x=df['total_sqft'], y=df['price'])
plt.ylabel('Price')
plt.xlabel('Total Square Feet')
plt.show()
```

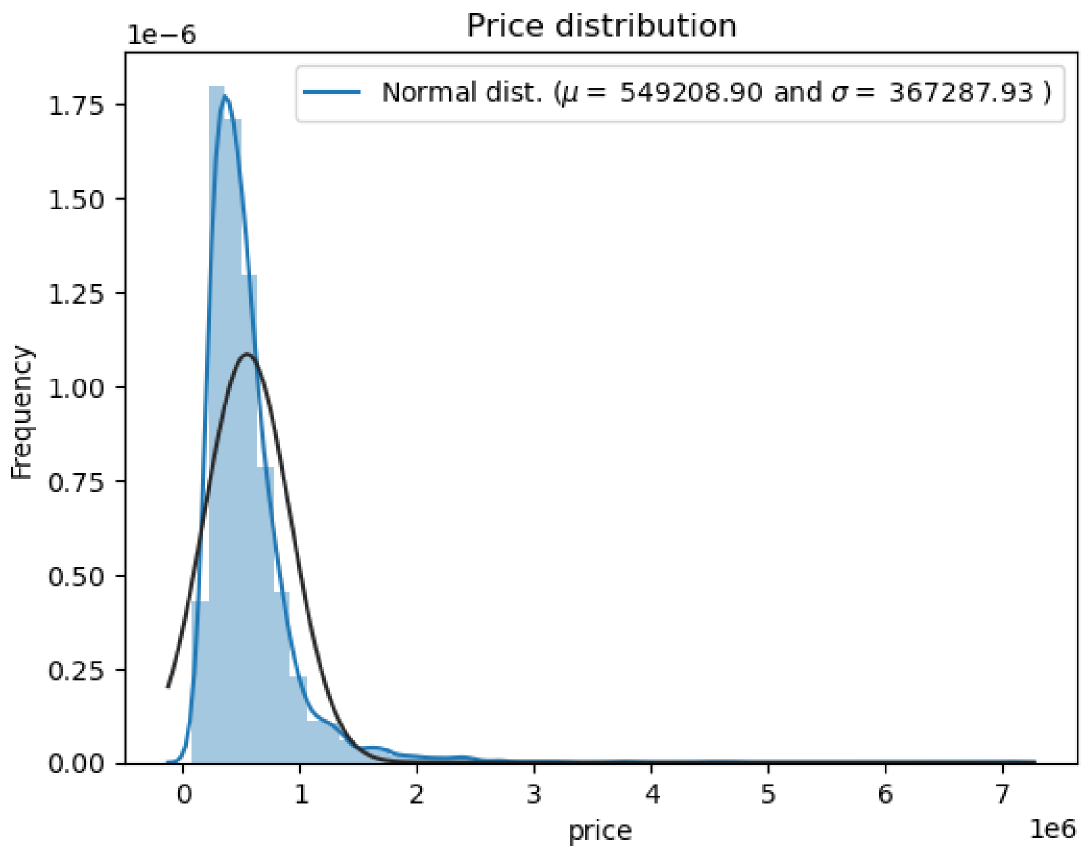


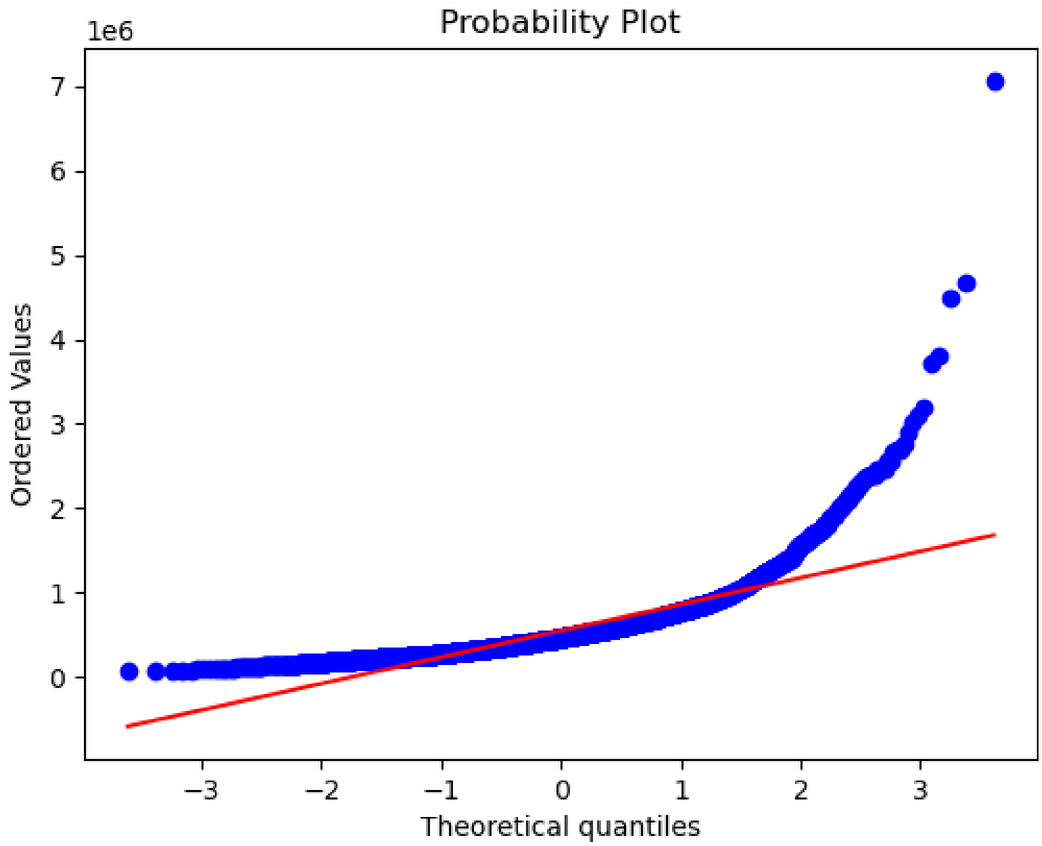
Now, the scatterplot reveals a clearer pattern, with price increasing as total square footage increases, suggesting a more linear relationship between the two variables.

[27]:
```
sns.distplot(df['price'] , fit=norm);

# Get the fitted parameters used by the function
(mu, sigma) = norm.fit(df['price'])
print( '\n mu = {:.2f} and sigma = {:.2f}\n'.format(mu, sigma))

#Now plot the distribution
plt.legend(['Normal dist. ($\mu=$ {:.2f} and $\sigma=$ {:.2f} )'.format(mu,␣
 ↪sigma)],
            loc='best')
plt.ylabel('Frequency')
plt.title('Price distribution')

#Get also the QQ-plot
fig = plt.figure()
res = stats.probplot(df['price'], plot=plt)
plt.show()
```

```
C:\Users\User\anaconda3\lib\site-packages\seaborn\distributions.py:2619:
FutureWarning: `distplot` is a deprecated function and will be removed in a
future version. Please adapt your code to use either `displot` (a figure-level
function with similar flexibility) or `histplot` (an axes-level function for
histograms).
  warnings.warn(msg, FutureWarning)
```

```
 mu = 549208.90 and sigma = 367287.93
```





[22]:
```
print("Skewness: %f" % df['price'].skew())
print("Kurtosis: %f" % df['price'].kurt())
```

```
Skewness: 3.998881
Kurtosis: 36.681189
```

A positive skewness value (3.998881) indicates a significant right skew, meaning that there is a long tail of high values on the right side of the distribution.

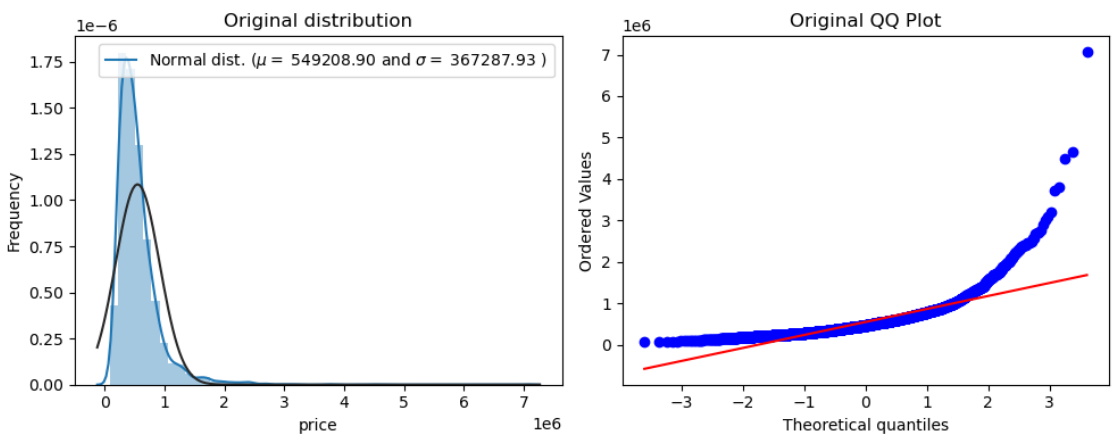
A high positive kurtosis value (36.681189) suggests heavy tails and an unusually sharp peak, indicating that extreme values are more likely to occur than in a normal distribution.

The target variable exhibits right skewness. Linear models tend to perform better with data that follows a normal distribution. Therefore, we need to transform this variable to achieve a more normal distribution.

[77]:
```
transformation_methods = {
    'Original': lambda x: x,
    'Log Transformation': np.log1p,
    'Box-Cox Transformation': lambda x: boxcox(x)[0],
}
```

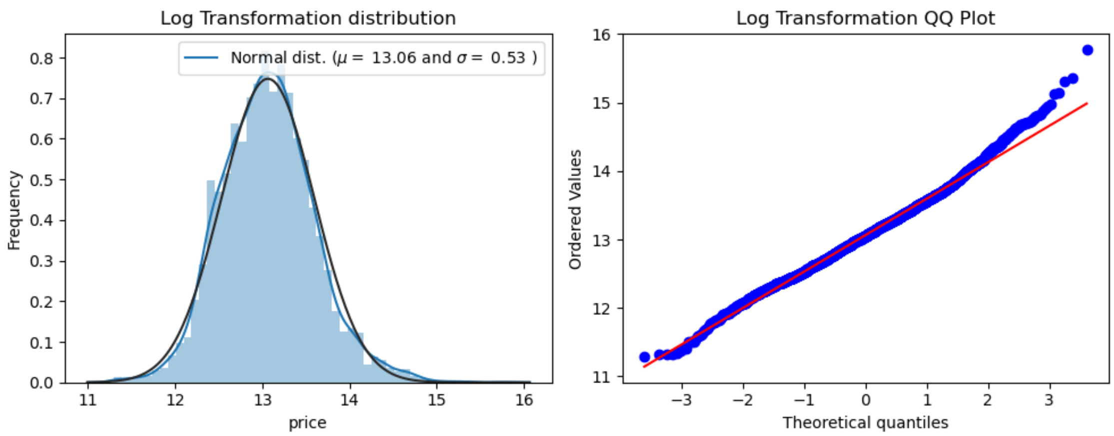
```
for name, transform in transformation_methods.items():
    plt.figure(figsize=(10, 4))

    transformed_data = transform(df['price'])

    plt.subplot(1, 2, 1)

    sns.distplot(transformed_data, fit=norm);
    mu, sigma = norm.fit(transformed_data)
    plt.legend(['Normal dist. ($\mu=$ {:.2f} and $\sigma=$ {:.2f} )'.format(mu,␣
 ↪sigma)],
                loc='best')
    plt.ylabel('Frequency')
    plt.title('{} distribution'.format(name))

    plt.subplot(1, 2, 2)
    probplot(transformed_data, plot=plt) # QQ plot
    plt.title('{} QQ Plot'.format(name))

    plt.tight_layout()
    plt.show()
```

```
C:\Users\User\anaconda3\lib\site-packages\seaborn\distributions.py:2619:
FutureWarning: `distplot` is a deprecated function and will be removed in a
future version. Please adapt your code to use either `displot` (a figure-level
function with similar flexibility) or `histplot` (an axes-level function for
histograms).
  warnings.warn(msg, FutureWarning)
```

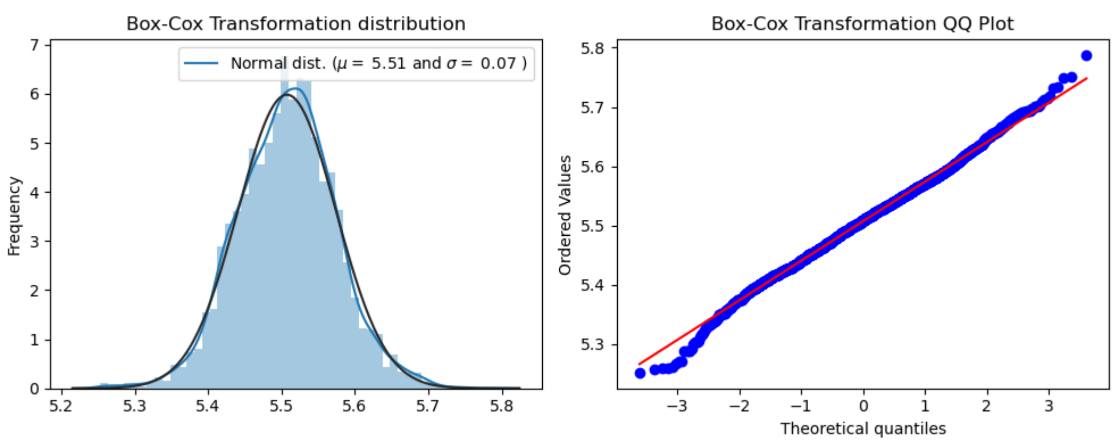



```
C:\Users\User\anaconda3\lib\site-packages\seaborn\distributions.py:2619:
FutureWarning: `distplot` is a deprecated function and will be removed in a
future version. Please adapt your code to use either `displot` (a figure-level
```

```
function with similar flexibility) or `histplot` (an axes-level function for
histograms).
  warnings.warn(msg, FutureWarning)
```




```
C:\Users\User\anaconda3\lib\site-packages\seaborn\distributions.py:2619:
FutureWarning: `distplot` is a deprecated function and will be removed in a
future version. Please adapt your code to use either `displot` (a figure-level
function with similar flexibility) or `histplot` (an axes-level function for
histograms).
  warnings.warn(msg, FutureWarning)
```



[33]:
```python
# log1p transformation
log_transformed_data = np.log1p(df['price'])

# Box-Cox transformation
boxcox_transformed_data, _ = boxcox(df['price'] + 0.01)
```

```
print("Log Transformation:")
print("Skewness:", skew(log_transformed_data))
print("Kurtosis:", kurtosis(log_transformed_data))
shapiro_log = shapiro(log_transformed_data)
print("Shapiro-Wilk test p-value:", shapiro_log[1])

print("\nBox-Cox Transformation:")
print("Skewness:", skew(boxcox_transformed_data))
print("Kurtosis:", kurtosis(boxcox_transformed_data))
shapiro_boxcox = shapiro(boxcox_transformed_data)
print("Shapiro-Wilk test p-value:", shapiro_boxcox[1])
```

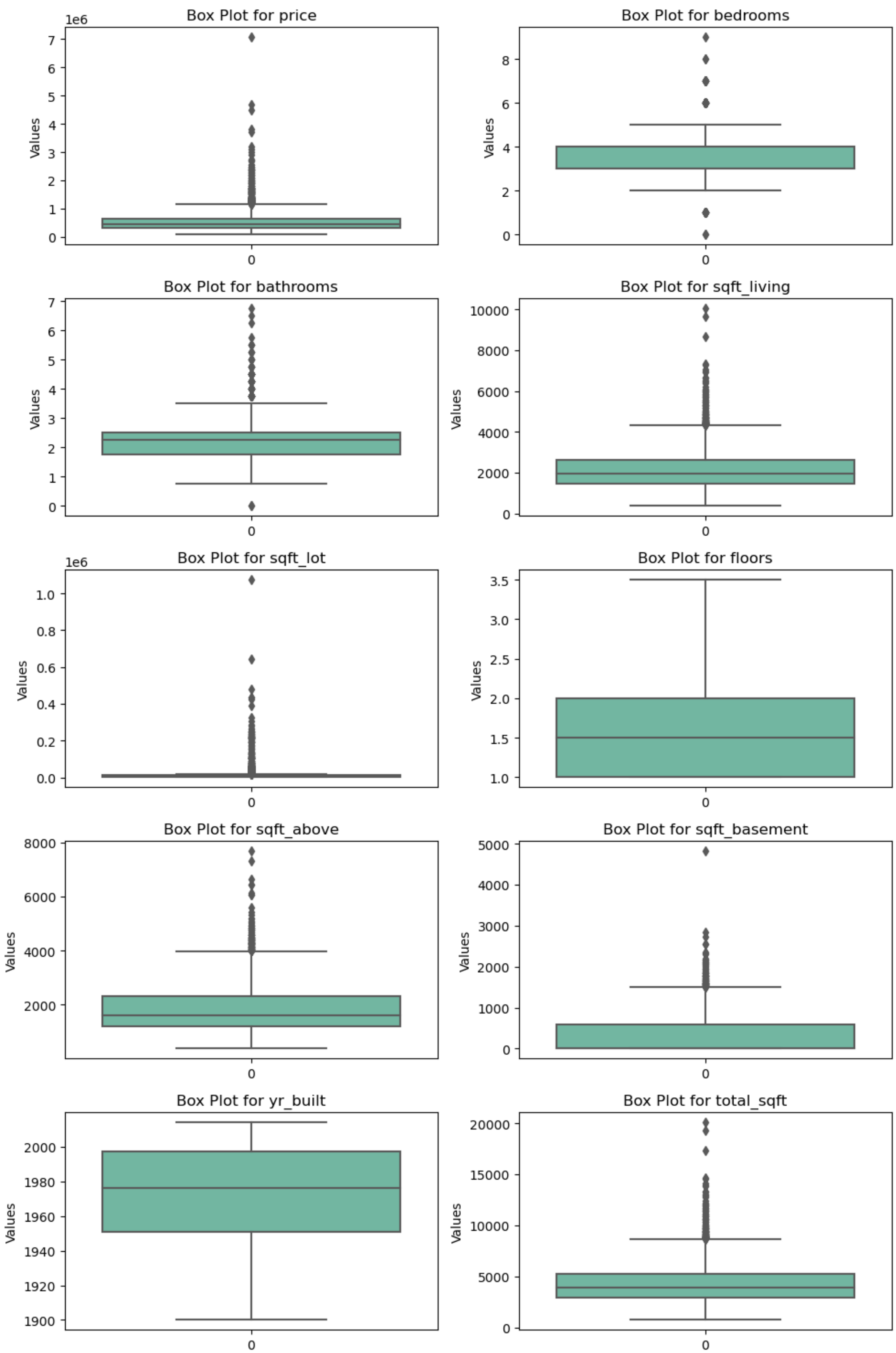
```
Log Transformation:
Skewness: 0.3007119380080463
Kurtosis: 0.6221470776891205
Shapiro-Wilk test p-value: 5.806631759620873e-14

Box-Cox Transformation:
Skewness: -0.011647219310031083
Kurtosis: 0.41685110477825704
Shapiro-Wilk test p-value: 1.9845242604787927e-06
```

Both transformations have effectively reduced skewness and achieved distributions with moderate tail heaviness. However, neither transformation fully normalizes the data, as indicated by the low p-values from the Shapiro-Wilk tests.

The Box-Cox transformation produces data with skewness and kurtosis closer to 0, indicating a more symmetrical distribution and tails resembling a normal distribution, supported by the Shapiro-Wilk test showing stronger evidence of normality compared to the Log transformation.

2 EDA

[54]:
```
fig, axes = plt.subplots(5, 2, figsize=(10, 15))

axes = axes.flatten()

# Filter columns excluding 'view', 'waterfront', 'condition' and 'yr_renovated'
cols_to_plot = [col for col in df.columns if col not in ['view', 'waterfront',␣
    ↪'condition', 'yr_renovated']]

for i, col in enumerate(cols_to_plot):
    sns.boxplot(data=df[col], ax=axes[i], orient='v', palette='Set2')
    axes[i].set_title('Box Plot for {}'.format(col))
    axes[i].set_ylabel('Values')

plt.tight_layout()
plt.show()
```


Box Plot for price
1e6
Values
Box Plot for bedrooms
Values
Box Plot for bathrooms
Values
Box Plot for sqft_living
Values
Box Plot for sqft_lot
1e6
Values
Box Plot for floors
Values
Box Plot for sqft_above
Values
Box Plot for sqft_basement
Values
Box Plot for yr_built
Values
Box Plot for total_sqft
Values

For better visualization we can split yr_built and yr_renovated into same length bins with np.linspace to have a better visualization and data presentation

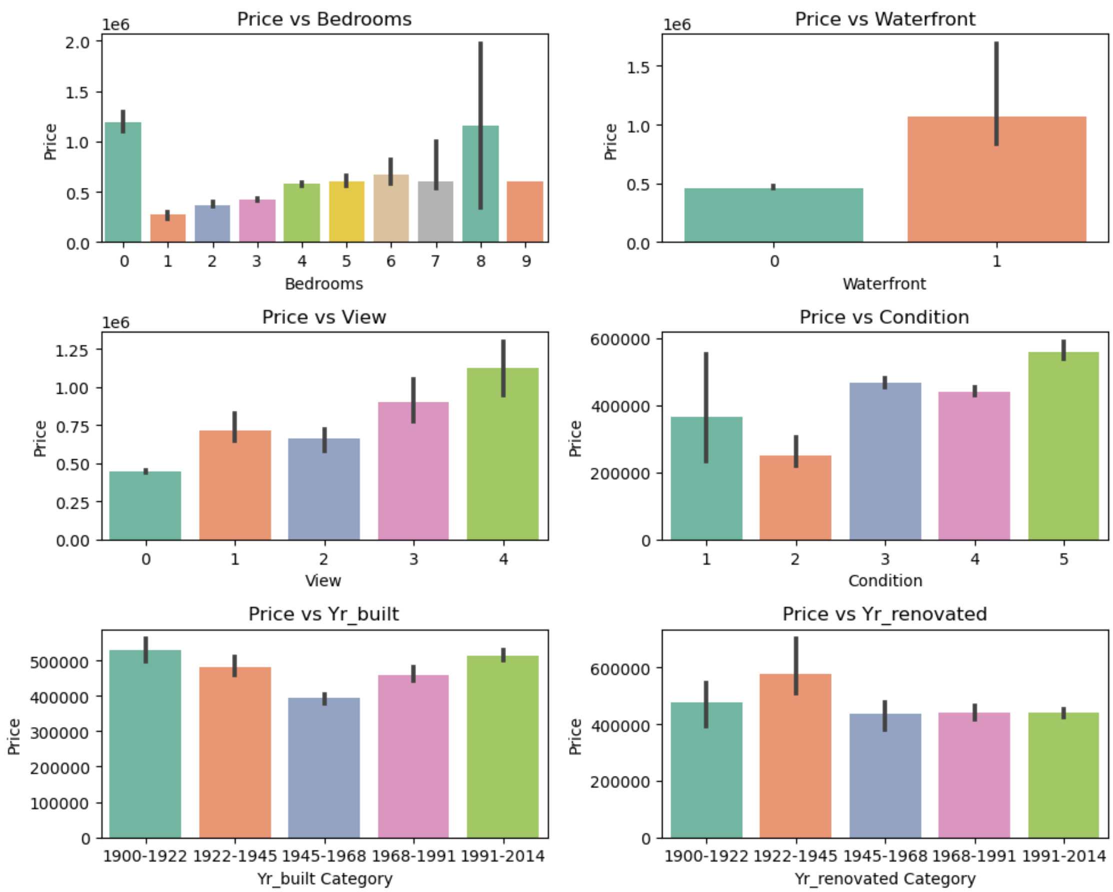
[59]:
```
filtered_yr_renovated = df[df['yr_renovated'] != 0]['yr_renovated']
description = pd.concat([filtered_yr_renovated.describe(), df['yr_built'].
 ↪describe()], axis=1)
print(description)
```

```
       yr_renovated     yr_built
count   1844.000000  4547.000000
mean    1994.454447  1970.794370
std       21.341590    29.764691
min     1912.000000  1900.000000
25%     1990.000000  1951.000000
50%     2001.000000  1976.000000
75%     2006.000000  1997.000000
max     2014.000000  2014.000000
```

Values range from ~1900 to 2014

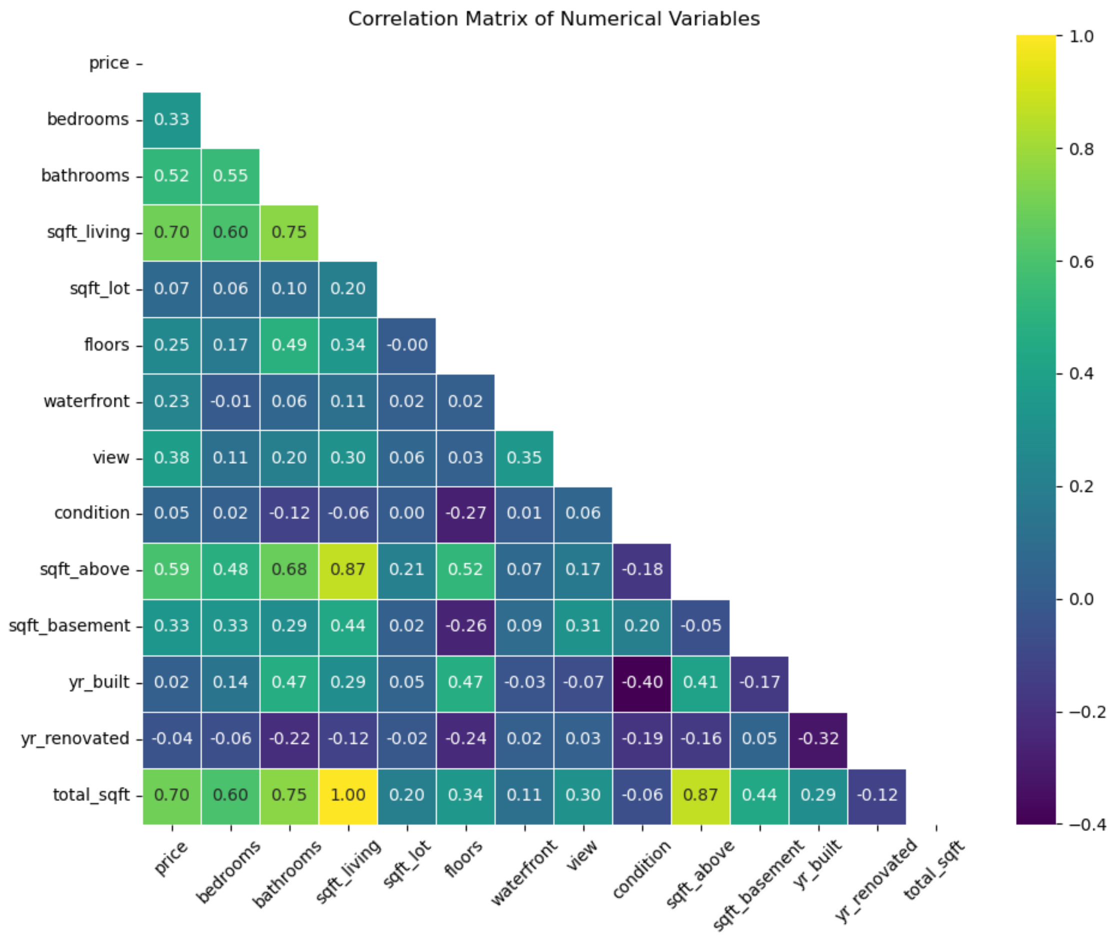
[61]:
```
num_bins = 5

# Generate equally spaced bins
bins = np.linspace(1900, 2014, num_bins + 1)

columns = ['bedrooms', 'waterfront', 'view', 'condition', 'yr_built',␣
 ↪'yr_renovated']

fig, axes = plt.subplots(3, 2, figsize=(10, 8))

axes = axes.flatten()

year_categories = {
    'yr_built': pd.cut(df['yr_built'], bins=bins,␣
 ↪labels=[f'{int(bins[i])}-{int(bins[i+1])}' for i in range(len(bins)-1)]),
    'yr_renovated': pd.cut(df['yr_renovated'], bins=bins,␣
 ↪labels=[f'{int(bins[i])}-{int(bins[i+1])}' for i in range(len(bins)-1)])
}

for i, column in enumerate(columns):
    if column in ['yr_built', 'yr_renovated']:
        sns.barplot(x=year_categories[column], y='price', data=df, estimator=np.
 ↪median, ax=axes[i], palette='Set2')
        axes[i].set_title(f'Price vs {column.capitalize()}')
        axes[i].set_xlabel(f'{column.capitalize()} Category')
    else:
```

```
        sns.barplot(x=column, y='price', data=df, estimator=np.median,␣
↪ax=axes[i], palette='Set2')
        axes[i].set_title(f'Price vs {column.capitalize()}')
        axes[i].set_xlabel(column.capitalize())
    axes[i].set_ylabel('Price')

plt.tight_layout()
plt.show()
```



[126]:
```
corr = df.corr()

mask = np.triu(np.ones_like(corr, dtype=bool))

plt.figure(figsize=(10, 8))

sns.heatmap(corr, mask=mask, cmap='viridis', annot=True, fmt=".2f",␣
↪annot_kws={"size": 10}, linewidths=.5)
plt.title('Correlation Matrix of Numerical Variables')
plt.xticks(rotation=45)
```

```
plt.yticks(rotation=0)
plt.tight_layout()
plt.show()
```



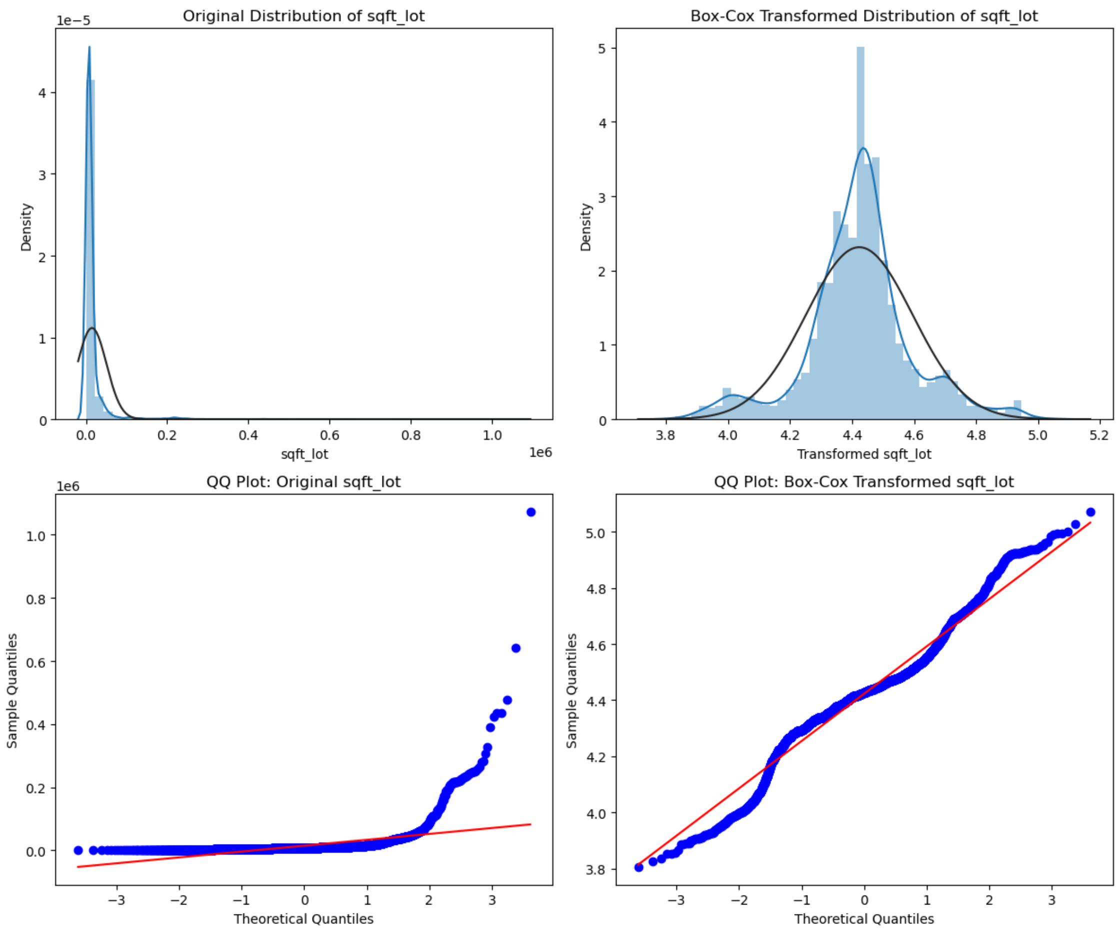
The correlation analysis highlights low correlations among most features, with a notable exception found between "total_sqft" and "sqft_living," indicating multicollinearity. The target variable, "price," exhibits a significant correlation (0.7) with "total_sqft," reflecting the common trend of higher prices associated with increased square footage. Features such as "sqft_above," "bathrooms," "bedrooms," and "sqft_basement" also show correlations with price, reinforcing their influence on housing prices. Furthermore, features like "view," "floors," and "waterfront" demonstrate moderate correlations with price, suggesting their relevance in determining property values.

[10]:
```
df.drop(columns = 'sqft_living', inplace = True)
```

3 Data Transformation with Box-Cox

[128]:
```python
# Box-Cox transformation on 'sqft_lot'
transformed_sqft_lot, lambda_value = boxcox(df['sqft_lot'])

fig, axes = plt.subplots(2, 2, figsize=(12, 10))

# Plot the original distribution of 'sqft_lot'
sns.distplot(df['sqft_lot'], fit=norm, ax=axes[0, 0])
axes[0, 0].set_title('Original Distribution of sqft_lot')
axes[0, 0].set_xlabel('sqft_lot')
axes[0, 0].set_ylabel('Density')

# Plot the Box-Cox transformed distribution of 'sqft_lot'
sns.distplot(transformed_sqft_lot, fit=norm, ax=axes[0, 1])
axes[0, 1].set_title('Box-Cox Transformed Distribution of sqft_lot')
axes[0, 1].set_xlabel('Transformed sqft_lot')
axes[0, 1].set_ylabel('Density')

# QQ plot for original 'sqft_lot'
res = stats.probplot(df['sqft_lot'], plot=axes[1, 0])
axes[1, 0].set_title('QQ Plot: Original sqft_lot')
axes[1, 0].set_xlabel('Theoretical Quantiles')
axes[1, 0].set_ylabel('Sample Quantiles')

# QQ plot for Box-Cox transformed 'sqft_lot'
res = stats.probplot(transformed_sqft_lot, plot=axes[1, 1])
axes[1, 1].set_title('QQ Plot: Box-Cox Transformed sqft_lot')
axes[1, 1].set_xlabel('Theoretical Quantiles')
axes[1, 1].set_ylabel('Sample Quantiles')

plt.tight_layout()
plt.show()
```

```
C:\Users\User\anaconda3\lib\site-packages\seaborn\distributions.py:2619:
FutureWarning: `distplot` is a deprecated function and will be removed in a
future version. Please adapt your code to use either `displot` (a figure-level
function with similar flexibility) or `histplot` (an axes-level function for
histograms).
  warnings.warn(msg, FutureWarning)
C:\Users\User\anaconda3\lib\site-packages\seaborn\distributions.py:2619:
FutureWarning: `distplot` is a deprecated function and will be removed in a
future version. Please adapt your code to use either `displot` (a figure-level
function with similar flexibility) or `histplot` (an axes-level function for
histograms).
  warnings.warn(msg, FutureWarning)
```

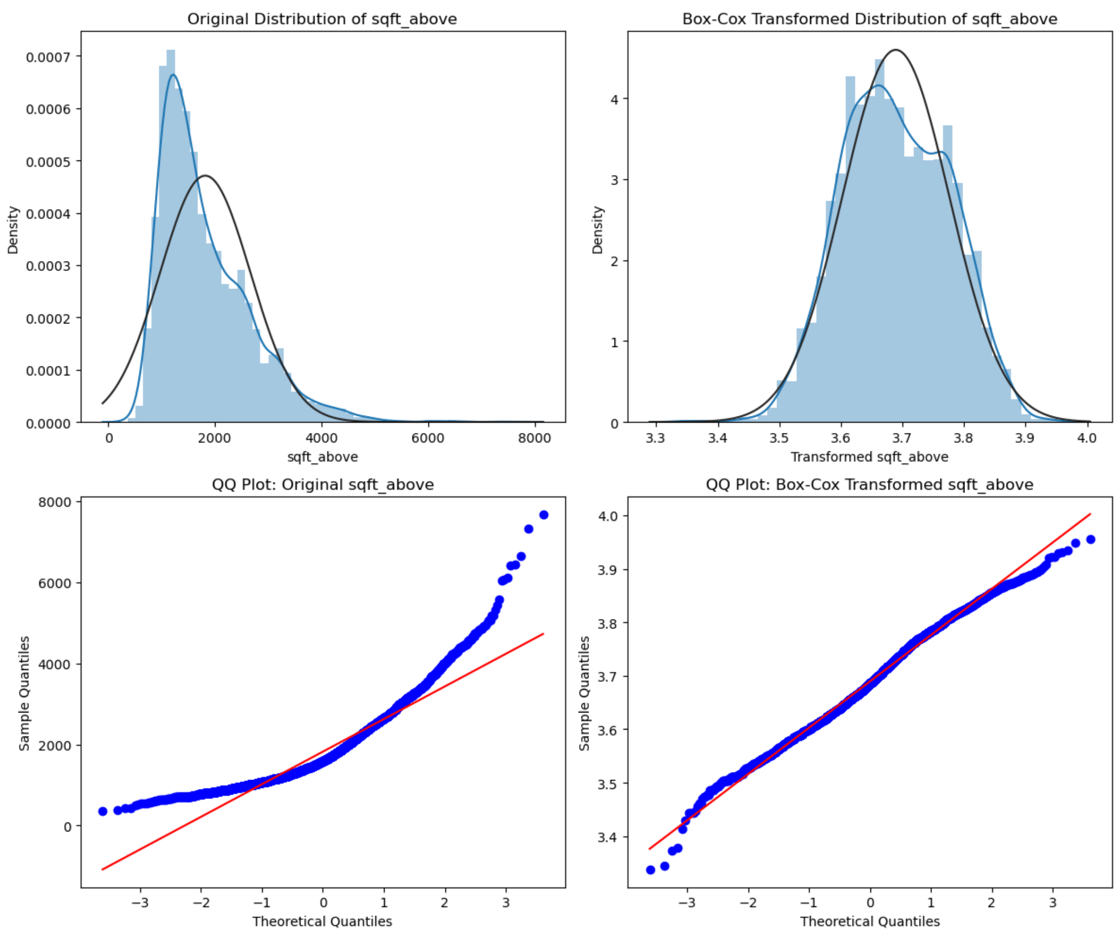


[99]:

```
transformed_sqft_above, lambda_value = boxcox(df['sqft_above'])

fig, axes = plt.subplots(2, 2, figsize=(12, 10))

sns.distplot(df['sqft_above'], fit=norm, ax=axes[0, 0])
axes[0, 0].set_title('Original Distribution of sqft_above')
axes[0, 0].set_xlabel('sqft_above')
axes[0, 0].set_ylabel('Density')

sns.distplot(transformed_sqft_above, fit=norm, ax=axes[0, 1])
axes[0, 1].set_title('Box-Cox Transformed Distribution of sqft_above')
axes[0, 1].set_xlabel('Transformed sqft_above')
axes[0, 1].set_ylabel('Density')

res = stats.probplot(df['sqft_above'], plot=axes[1, 0])
axes[1, 0].set_title('QQ Plot: Original sqft_above')
axes[1, 0].set_xlabel('Theoretical Quantiles')
axes[1, 0].set_ylabel('Sample Quantiles')
```

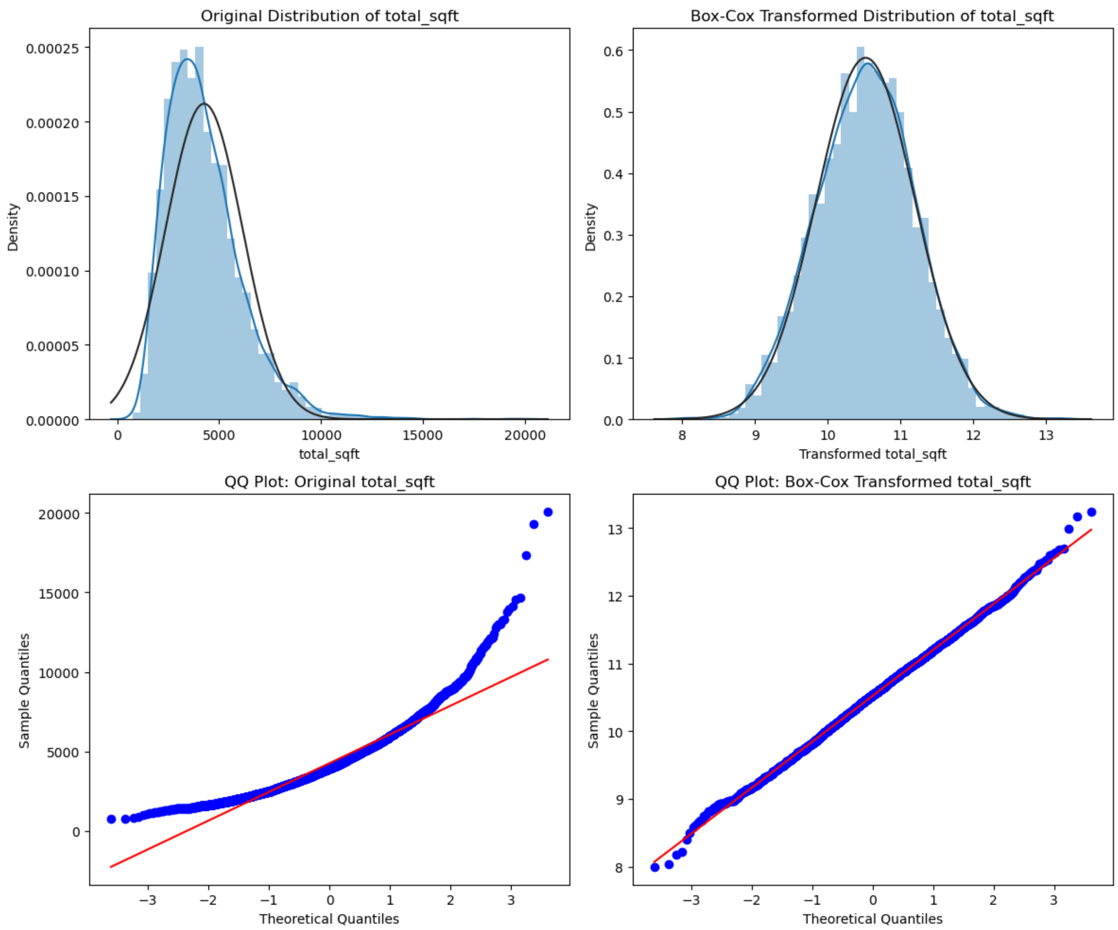
```
res = stats.probplot(transformed_sqft_above, plot=axes[1, 1])
axes[1, 1].set_title('QQ Plot: Box-Cox Transformed sqft_above')
axes[1, 1].set_xlabel('Theoretical Quantiles')
axes[1, 1].set_ylabel('Sample Quantiles')

plt.tight_layout()
plt.show()
```

```
C:\Users\User\anaconda3\lib\site-packages\seaborn\distributions.py:2619:
FutureWarning: `distplot` is a deprecated function and will be removed in a
future version. Please adapt your code to use either `displot` (a figure-level
function with similar flexibility) or `histplot` (an axes-level function for
histograms).
  warnings.warn(msg, FutureWarning)
C:\Users\User\anaconda3\lib\site-packages\seaborn\distributions.py:2619:
FutureWarning: `distplot` is a deprecated function and will be removed in a
future version. Please adapt your code to use either `displot` (a figure-level
function with similar flexibility) or `histplot` (an axes-level function for
histograms).
  warnings.warn(msg, FutureWarning)
```



[101]:
```
transformed_total_sqft, lambda_value = boxcox(df['total_sqft'])

fig, axes = plt.subplots(2, 2, figsize=(12, 10))

sns.distplot(df['total_sqft'], fit=norm, ax=axes[0, 0])
axes[0, 0].set_title('Original Distribution of total_sqft')
axes[0, 0].set_xlabel('total_sqft')
axes[0, 0].set_ylabel('Density')

sns.distplot(transformed_total_sqft, fit=norm, ax=axes[0, 1])
axes[0, 1].set_title('Box-Cox Transformed Distribution of total_sqft')
axes[0, 1].set_xlabel('Transformed total_sqft')
axes[0, 1].set_ylabel('Density')

res = stats.probplot(df['total_sqft'], plot=axes[1, 0])
axes[1, 0].set_title('QQ Plot: Original total_sqft')
axes[1, 0].set_xlabel('Theoretical Quantiles')
axes[1, 0].set_ylabel('Sample Quantiles')

res = stats.probplot(transformed_total_sqft, plot=axes[1, 1])
axes[1, 1].set_title('QQ Plot: Box-Cox Transformed total_sqft')
axes[1, 1].set_xlabel('Theoretical Quantiles')
axes[1, 1].set_ylabel('Sample Quantiles')

plt.tight_layout()
plt.show()
```

```
C:\Users\User\anaconda3\lib\site-packages\seaborn\distributions.py:2619:
FutureWarning: `distplot` is a deprecated function and will be removed in a
future version. Please adapt your code to use either `displot` (a figure-level
function with similar flexibility) or `histplot` (an axes-level function for
histograms).
  warnings.warn(msg, FutureWarning)
C:\Users\User\anaconda3\lib\site-packages\seaborn\distributions.py:2619:
FutureWarning: `distplot` is a deprecated function and will be removed in a
future version. Please adapt your code to use either `displot` (a figure-level
function with similar flexibility) or `histplot` (an axes-level function for
histograms).
  warnings.warn(msg, FutureWarning)
```



Now let's apply the changes to the original dataset and some more final data preprocessing

[11]:
```
from scipy.stats import boxcox

columns_to_transform = ['price', 'sqft_lot', 'sqft_above', 'total_sqft']

lambda_values = {}

for column in columns_to_transform:
    transformed_data, lambda_value = boxcox(df[column])
    df[column] = transformed_data
    lambda_values[column] = lambda_value

for column, lambda_value in lambda_values.items():
    print(f"Lambda value for {column}: {lambda_value}")

df.head()
```

```
Lambda value for price: -0.15864308256570642
```

```
Lambda value for sqft_lot: -0.18122901712209788
Lambda value for sqft_above: -0.21547599254162927
Lambda value for total_sqft: 0.05601305582156565
```

[11]:

```
      price  bedrooms  bathrooms  sqft_lot  floors  waterfront  view  \
0  5.456721         3       1.50  4.433230     1.5           0     0
1  5.689889         5       2.50  4.459327     2.0           0     4
2  5.468540         3       2.00  4.511285     1.0           0     0
3  5.495313         3       2.25  4.436136     1.0           0     0
4  5.529157         4       2.50  4.487456     1.0           0     0

   condition  sqft_above  sqft_basement  yr_built  yr_renovated  total_sqft
0          3    3.657364              0      1955          2005    9.926942
1          5    3.834616            280      1921             0   11.530772
2          4    3.731724              0      1966             0   10.500504
3          4    3.593342           1000      1963             0   10.557143
4          4    3.622504            800      1976          1992   10.508713
```

Now that all the data has been appropriately processed, they are ready for further modeling and analysis.

4 Modelling

[12]:

```python
X = df.drop(columns=['price'])
y = df['price']

# 80% train and 20% test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

print("Train shape:", X_train.shape)
print("Test shape:", y_test.shape)
```

```
Train shape: (3637, 12)
Test shape: (910,)
```

4.1 Creating the base models

[13]:

```python
# Cross-validation
def cv_rmse(model):
    kf = KFold(n_splits=5, shuffle=True, random_state=42)
    n_splits = kf.get_n_splits(X_train)

    rmse = np.sqrt(-cross_val_score(model, X_train, y_train,␣
 ↪scoring="neg_mean_squared_error", cv=kf))

    return rmse, n_splits
```

```
n_folds = 5

skf = KFold(n_splits=5, shuffle=True, random_state=42)

cv_strategy = list(skf.split(X_train, y_train))
```

[14]:

```
# Creating a scoring function for Root Mean Squared Error (RMSE) for Bayesian␣
 ↪optimization
scoring = make_scorer(partial(mean_squared_error, squared=False),
                      greater_is_better=False)
```

[38]:

```
lasso_params = {
    'alpha': Real(0.001, 1),
    'max_iter': Integer(100, 300),
    'selection': ['cyclic', 'random'],
    'tol': Real(1e-5, 1e-1)
}

enet_params = {
    'alpha': Real(0.001, 10),
    'l1_ratio': Real(0.1, 1.0),
    'max_iter': Integer(100, 500),
    'selection': ['cyclic', 'random'],
    'tol': Real(1e-5, 1e-1)
}

kr_params = {
    'alpha': Real(0.001, 10.0),
    'kernel': ['linear', 'polynomial', 'rbf'],
    'gamma': Real(0.001, 0.1),
    'degree': Integer(1, 10),
    'coef0': Real(0.0, 1.0),
}

xgb_params = {'learning_rate': (0.01, 1.0),
              'n_estimators': (100, 3000),
              'max_depth': (1, 10),
              'min_child_weight': (1, 10),
              'gamma': (0.01, 1.0),
              'subsample': (0.5, 1.0),
              'colsample_bytree': (0.5, 1.0),
              'reg_alpha': (0.01, 1.0),
              'reg_lambda': (0.01, 1.0)}

catboost_params = {'learning_rate': (0.01, 1.0),
                   'iterations': (100, 1000),
                   'depth': (3, 10),
```

```
                    'l2_leaf_reg': (1, 10),
                    'subsample': (0.5, 1.0),
                    'colsample_bylevel': (0.5, 1.0),
                    'bagging_temperature': (0.0, 10.0),
                    'border_count': (1, 255),
                    'leaf_estimation_iterations': (1, 10)}

lasso = Lasso()
enet = ElasticNet()
kr = KernelRidge()
xgb = XGBRegressor()
catboost = CatBoostRegressor()

lasso_opt = BayesSearchCV(lasso, lasso_params, cv=cv_strategy, scoring=scoring,␣
 ↪random_state=42, n_jobs=-1, verbose=0, iid=False, return_train_score=False,␣
 ↪refit=False, optimizer_kwargs={'base_estimator': 'GP'}, n_iter=200)

enet_opt = BayesSearchCV(enet, enet_params, cv=cv_strategy, scoring=scoring,␣
 ↪random_state=42, n_jobs=-1, verbose=0, iid=False, return_train_score=False,␣
 ↪refit=False, optimizer_kwargs={'base_estimator': 'GP'}, n_iter=200)

kr_opt = BayesSearchCV(kr, kr_params, cv=cv_strategy, scoring=scoring,␣
 ↪random_state=42, n_jobs=-1, verbose=0, iid=False, return_train_score=False,␣
 ↪refit=False, optimizer_kwargs={'base_estimator': 'GP'}, n_iter=200)

xgb_opt = BayesSearchCV(xgb, xgb_params, cv=cv_strategy, scoring=scoring,␣
 ↪random_state=42, n_jobs=-1, verbose=0, iid=False, return_train_score=False,␣
 ↪refit=False, optimizer_kwargs={'base_estimator': 'GP'}, n_iter=200)

catboost_opt = BayesSearchCV(catboost, catboost_params, cv=cv_strategy,␣
 ↪scoring=scoring, random_state=42, n_jobs=-1, verbose=0, iid=False,␣
 ↪return_train_score=False, refit=False, optimizer_kwargs={'base_estimator':␣
 ↪'GP'}, n_iter=200)
```

```
C:\Users\User\anaconda3\lib\site-packages\skopt\searchcv.py:300: UserWarning:
The `iid` parameter has been deprecated and will be ignored.
  warnings.warn("The `iid` parameter has been deprecated "
```

[ ]:
```
def report_perf(optimizer, X, y, title="model", callbacks=None):

    start = time()

    if callbacks is not None:
        optimizer.fit(X, y, callback=callbacks)
    else:
        optimizer.fit(X, y)
```

```
    d=pd.DataFrame(optimizer.cv_results_)
    best_score = optimizer.best_score_
    best_score_std = d.iloc[optimizer.best_index_].std_test_score
    best_params = optimizer.best_params_

    print(f"The {title} process took {time() - start:.2f} seconds, with "
      f"{len(optimizer.cv_results_['params'])} candidates checked. "
      f"The best cross-validation score achieved was {best_score:.3f} ±␣
 ↪{best_score_std:.3f}")
    print('Best parameters:')
    pprint.pprint(best_params)
    print()
    return best_params
```

[62]:
```
warnings.filterwarnings("ignore", message="The objective has been evaluated",␣
 ↪category=UserWarning)

early_stopping = DeltaYStopper(delta=0.00001)

bayesian_opt = [(lasso_opt, "Lasso Regression"), (enet_opt, "ElasticNet␣
 ↪Regression"), (kr_opt, "Kernel Ridge Regression"),
               (xgb_opt, "XGBRegressor"), (catboost_opt, "CatBoost Regression")]

for model, name in bayesian_opt:
    best_params = report_perf(model, X_train, y_train, name, ␣
 ↪callbacks=early_stopping)
```

```
The Lasso Regression process took 45.85 seconds, with 28 candidates checked. The
best cross-validation score achieved was -0.046 ± 0.001
Best parameters:
OrderedDict([('alpha', 0.001), ('max_iter', 100)])

The ElasticNet Regression process took 68.49 seconds, with 36 candidates
checked. The best cross-validation score achieved was -0.045 ± 0.002
Best parameters:
OrderedDict([('alpha', 0.001), ('l1_ratio', 0.1), ('max_iter', 140)])

The Kernel Ridge Regression process took 514.34 seconds, with 51 candidates
checked. The best cross-validation score achieved was -0.059 ± 0.002
Best parameters:
OrderedDict([('alpha', 0.001),
             ('degree', 9),
             ('gamma', 0.1),
             ('kernel', 'linear')])

The XGBRegressor process took 4649.62 seconds, with 200 candidates checked. The
best cross-validation score achieved was -0.044 ± 0.001
```

```
Best parameters:
OrderedDict([('colsample_bytree', 0.5),
             ('gamma', 0.01),
             ('learning_rate', 0.01),
             ('max_depth', 10),
             ('min_child_weight', 1),
             ('n_estimators', 3000),
             ('reg_alpha', 0.01),
             ('reg_lambda', 1.0),
             ('subsample', 0.5)])

The CatBoost Regression process took 9088.73 seconds, with 200 candidates
checked. The best cross-validation score achieved was -0.044 ± 0.001
Best parameters:
OrderedDict([('bagging_temperature', 7.909623291326059),
             ('border_count', 221),
             ('colsample_bylevel', 0.5365373220208844),
             ('depth', 5),
             ('iterations', 945),
             ('l2_leaf_reg', 5),
             ('leaf_estimation_iterations', 8),
             ('learning_rate', 0.016419932455436083),
             ('subsample', 0.9737118326453182)])
```

For Lasso, Elastic Net and Kernel Ridge regression the parameters were updated, the final parameters are shown below

[15]:
```python
lasso_params = {'alpha': 0.001, 'max_iter': 300, 'selection': 'random', 'tol':␣
 ↪0.1}
enet_params = {'alpha': 0.001, 'l1_ratio': 0.1, 'max_iter': 408, 'selection':␣
 ↪'random', 'tol': 1e-05}
kr_params = {'alpha': 4.753469290519291, 'coef0': 1.0, 'degree': 2, 'gamma': 0.
 ↪04643060076679006, 'kernel': 'polynomial'}
xgb_params = {'colsample_bytree': 0.5, 'gamma': 0.01, 'learning_rate': 0.01,␣
 ↪'max_depth': 10,
              'min_child_weight': 1, 'n_estimators': 3000, 'reg_alpha': 0.01,␣
 ↪'reg_lambda': 1.0, 'subsample': 0.5}
catboost_params = {'bagging_temperature': 7.909623291326059, 'border_count':␣
 ↪221,
                   'colsample_bylevel': 0.5365373220208844, 'depth': 5,␣
 ↪'iterations': 945,
                   'l2_leaf_reg': 5, 'leaf_estimation_iterations': 8,␣
 ↪'learning_rate': 0.016419932455436083,
                   'subsample': 0.9737118326453182, 'verbose': 0}

# models with updated parameters
```

```
lasso_model = Lasso(**lasso_params)
enet_model = ElasticNet(**enet_params)
kr_model = KernelRidge(**kr_params)
xgb_model = XGBRegressor(**xgb_params)
catboost_model = CatBoostRegressor(**catboost_params)
```

[20]:
```
models = [lasso_model, xgb_model, kr_model, enet_model, catboost_model]

for model in models:
    rmse, n_splits = cv_rmse(model)
    print("{} rmse: {:.4f} ± {:.4f}".format(type(model).__name__, rmse.mean(),␣
↪rmse.std()))
```

```
Lasso rmse: 0.0455 ± 0.0009
XGBRegressor rmse: 0.0436 ± 0.0010
KernelRidge rmse: 0.0448 ± 0.0009
ElasticNet rmse: 0.0448 ± 0.0009
CatBoostRegressor rmse: 0.0433 ± 0.0011
```

4.2 Averaging Models

Below we will create the weighted averaging models where models with lower rmse affect the score more

[16]:
```
class AveragingModels(BaseEstimator, RegressorMixin, TransformerMixin):
    def __init__(self, models, n_jobs=None):
        self.models = models
        self.n_jobs = n_jobs

    def fit(self, X, y):
        self.models_ = [clone(model).fit(X, y) for model in self.models]
        return self

    def predict(self, X):
        check_is_fitted(self, 'models_')

        predictions = Parallel(n_jobs=self.n_jobs)(
            delayed(model.predict)(X)
            for model in self.models_
        )

        # Calculating average predictions from individual models
        y_pred = np.mean(predictions, axis=0)

        # Error of each model
        errors = [np.sqrt(np.mean((y_pred - pred) ** 2)) for pred in␣
↪predictions]
```

```
        # weights based on errors
        weights = [1 / error for error in errors]

        # Normalizing weights to sum to 1
        total_weight = sum(weights)
        normalized_weights = [weight / total_weight for weight in weights]

        # weighted average of predictions
        weighted_predictions = np.average(predictions, axis=0,␣
 ↪weights=normalized_weights)

        return weighted_predictions
```

[50]:
```
best_score = float('inf')
best_models = None
best_std = None

all_models = [enet_model, xgb_model, kr_model, lasso_model, catboost_model]

for r in range(2, 6):
    for model_combination in combinations(all_models, r):
        averaged_models = AveragingModels(models=model_combination)

        scores, n_splits = cv_rmse(averaged_models)

        mean_score = scores.mean()
        std = scores.std()

        if mean_score < best_score:
            best_score = mean_score
            best_models = model_combination
            best_std = std

best_model_names = [model.__class__.__name__ for model in best_models]

print("Best score for weighted AveragingModels: {:.4f} ± {:.4f}".
 ↪format(best_score, best_std))
print("Best models for AveragingModels:", best_model_names)
```

```
Best score for weighted AveragingModels: 0.0433 ± 0.0010
Best models for AveragingModels: ['XGBRegressor', 'KernelRidge',
'CatBoostRegressor']
```

4.3 Stacking Averaged Models

[17]:
```
class StackingAveragedModels(BaseEstimator, RegressorMixin, TransformerMixin):
    def __init__(self, base_models, meta_model, n_folds=5):
        self.base_models = base_models
        self.meta_model = meta_model
        self.n_folds = n_folds

    def fit(self, X, y):
        self.base_models_ = [list() for _ in self.base_models]
        self.meta_model_ = clone(self.meta_model)
        kfold = KFold(n_splits=self.n_folds, shuffle=True, random_state=42)

        out_of_fold_predictions = np.zeros((X.shape[0], len(self.base_models)))
        for i, model in enumerate(self.base_models):
            for train_index, holdout_index in kfold.split(X, y):
                instance = clone(model)
                self.base_models_[i].append(instance)
                instance.fit(pd.DataFrame(X).iloc[train_index], pd.Series(y).
 ↪iloc[train_index])
                y_pred = instance.predict(pd.DataFrame(X).iloc[holdout_index])
                out_of_fold_predictions[holdout_index, i] = y_pred

        self.meta_model_.fit(out_of_fold_predictions, y)
        return self

    def predict(self, X):
        meta_features = np.column_stack([
            np.column_stack([model.predict(X) for model in base_models]).
 ↪mean(axis=1)
            for base_models in self.base_models_ ])
        return self.meta_model_.predict(meta_features)
```

[55]:
```
all_models = [xgb_model, kr_model, catboost_model, enet_model, lasso_model]

scores = []

for r_base in range(2, len(all_models) + 1):
    for base_model_combination in combinations(all_models, r_base):
        remaining_models = [model for model in all_models if model not in␣
 ↪base_model_combination]

        for meta_model in remaining_models:
            stacked_averaged_models =␣
 ↪StackingAveragedModels(base_models=base_model_combination,␣
 ↪meta_model=meta_model)
```

```
            score, _ = cv_rmse(stacked_averaged_models)

            scores.append((base_model_combination, meta_model.__class__.
 ↪__name__, score.mean(), score.std()))

best_combination, best_meta_model, best_mean_score, best_std_score =␣
 ↪min(scores, key=lambda x: x[2])

print("Optimal Combination: {}, Meta Model: {}, Mean Score: {:.4f}, Std Score:␣
 ↪{:.4f}".format(best_combination, best_meta_model, best_mean_score,␣
 ↪best_std_score))
```

```
Optimal Combination: (KernelRidge(alpha=4.753469290519291, coef0=1.0, degree=2,
            gamma=0.04643060076679006, kernel='polynomial'),
<catboost.core.CatBoostRegressor object at 0x000002C0C503E970>,
ElasticNet(alpha=0.001, l1_ratio=0.1, max_iter=408, selection='random',
           tol=1e-05), Lasso(alpha=0.001, max_iter=300, selection='random',
tol=0.1)), Meta Model: XGBRegressor, Mean Score: 0.0435, Std Score: 0.0010
```

5 Testing

The Averaged Models method exhibits lower mean MSE, mean RMSE, mean MAE, and higher mean R-squared and modified R-squared values compared to the Stacked Averaged Models method. Now lets compare it to a baseline model, and to the lowest rmse bayesian hyperparameter tuned model CatBoost

[18]:
```
averaged_models = AveragingModels(models=[xgb_model,kr_model,catboost_model])
```

[19]:
```
lambda_param_price = -0.15864308256570642

# Revert true target values from Box-Cox transformation
y_test_reverted = inv_boxcox(y_test, lambda_param_price)
```

[34]:
```
def mape(y_true, y_pred):
    return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
```

[33]:
```
models = [
    ("Linear Regression", LinearRegression()),
    ("CatBoost", catboost_model),
    ("Averaged Models", averaged_models)
]

rmse_list = []
mae_list = []
r2_list = []
mape_list = []
```

```python
for name, model in models:

    model.fit(X_train, y_train)

    y_test_pred = model.predict(X_test)

    # Revert the predicted values from Box-Cox transformation
    y_test_pred_reverted = inv_boxcox(y_test_pred, lambda_param_price)

    rmse = np.sqrt(mean_squared_error(y_test_reverted, y_test_pred_reverted))
    mae = mean_absolute_error(y_test_reverted, y_test_pred_reverted)
    r2 = r2_score(y_test_reverted, y_test_pred_reverted)
    mape_value = mape(y_test_reverted, y_test_pred_reverted)

    rmse_list.append(rmse)
    mae_list.append(mae)
    r2_list.append(r2)
    mape_list.append(mape_value)

    print(f"{name}:")
    print(f"RMSE: {rmse:.4f}")
    print(f"MAE: {mae:.4f}")
    print(f"R-squared: {r2:.4f}")
    print(f"MAPE: {mape_value:.4f}")
    print()
```

```
Linear Regression:
RMSE: 247466.1419
MAE: 154147.4988
R-squared: 0.5088
MAPE: 31.5419

CatBoost:
RMSE: 245510.0785
MAE: 147458.3447
R-squared: 0.5165
MAPE: 29.5251

Averaged Models:
RMSE: 241854.9708
MAE: 147411.3170
R-squared: 0.5308
MAPE: 29.5789
```