HOUSE PRICE PREDICTION

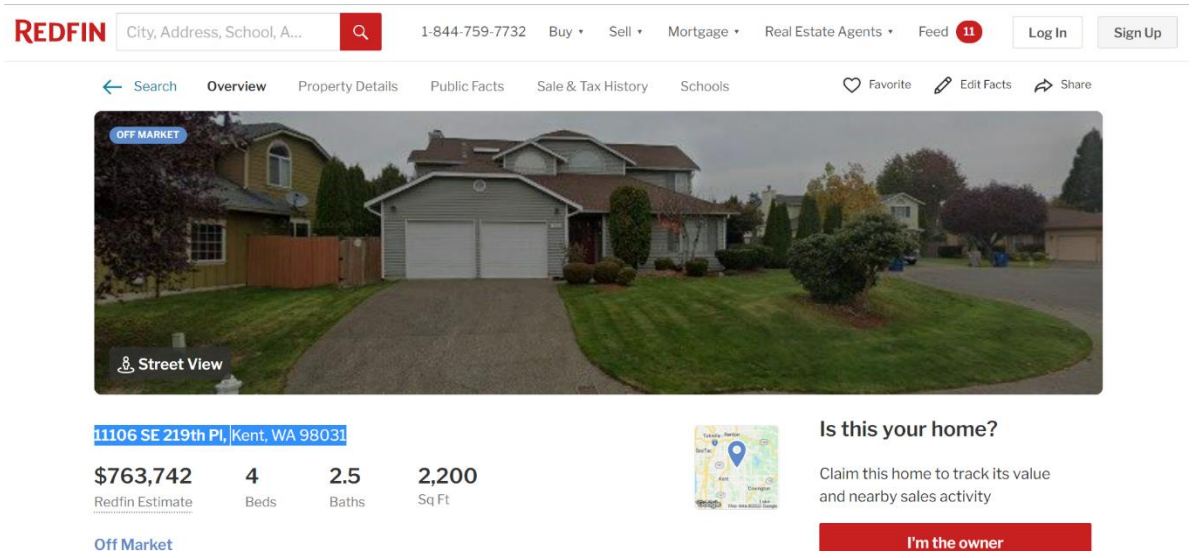
Project Report

Filed by Aditya Rishi of PGP-DSBA-Feb 21

For Great Learning Capstone Project

March 6, 2022

Contents



A house on sale in King County, Washington, US.
Picture from the website of US property portal Redfin.com

King County real-estate market information from Redfin.com

Problem statement

The aim is to build a model to predict the selling price of the house for the real estate customers with respect to their budget and priorities. The house price is determined by using various factors such as geographical location, number of floors in the house, quality and current condition of the house, total area, furnished or non-furnished, number of bedrooms and bathrooms and so on. Here we make use of Machine Learning algorithms to build the price prediction model for the houses. We have around 21k records spread around 100 years of time span; this will fecilitate us to build an effective model.

	cid	dayhours	price	room_bed	room_bath	living_measure	lot_measure	ceil	coast	sight	condition	quality	ceil_measure
0	3876100940	20150427T000000	600000	4.000000	1.750000	3050.000000	9440.000000	1	0	0.000000	3	8.000000	1800.000000
1	3145600250	20150317T000000	190000	2.000000	1.000000	670.000000	3101.000000	1	0	0.000000	4	6.000000	670.000000
2	7129303070	20140820T000000	735000	4.000000	2.750000	3040.000000	2415.000000	2	1	4.000000	3	8.000000	3040.000000
3	7338220280	20141010T000000	257000	3.000000	2.500000	1740.000000	3721.000000	2	0	0.000000	3	8.000000	1740.000000
4	7950300670	20150218T000000	450000	2.000000	1.000000	1120.000000	4590.000000	1	0	0.000000	3	7.000000	1120.000000

basement	yr_built	yr_renovated	zipcode	lat	long	living_measure15	lot_measure15	furnished	total_area
1250.000000	1966	0	98034	47.722800	-122.183000	2020.000000	8660.000000	0.000000	12490
0.000000	1948	0	98118	47.554600	-122.274000	1660.000000	4100.000000	0.000000	3771
0.000000	1966	0	98118	47.518800	-122.256000	2620.000000	2433.000000	0.000000	5455
0.000000	2009	0	98002	47.336300	-122.213000	2030.000000	3794.000000	0.000000	5461
0.000000	1924	0	98118	47.566300	-122.285000	1120.000000	5100.000000	0.000000	5710

Data dictionary

Objective:

Take advantage of all of the feature variables available below, use it to analyse and predict house prices.

1. cid: a notation for a house
2. dayhours: Date house was sold
3. price: Price is prediction target
4. room_bed: Number of Bedrooms/House
5. room_bath: Number of bathrooms/bedrooms
6. living_measure: square footage of the home
7. lot_measure: quare footage of the lot
8. ceil: Total floors (levels) in house
9. coast: House which has a view to a waterfront
10. sight: Has been viewed
11. condition: How good the condition is (Overall)
12. quality: grade given to the housing unit, based on grading system
13. ceil_measure: square footage of house apart from basement
14. basement_measure: square footage of the basement
15. yr_built: Built Year
16. yr_renovated: Year when house was renovated
17. zipcode: zip
18. lat: Latitude coordinate
19. long: Longitude coordinate
20. living_measure15: Living room area in 2015(implies-- some renovations) This might or might not have affected the lotsize area
21. lot_measure15: lotSize area in 2015(implies-- some renovations)
22. furnished: Based on the quality of room
23. total_area: Measure of both living and lot

```
The dataset has 21613 rows or records and 23 columns or variables
```

Business Problem Understanding

What we are trying to solve

Predicting house prices in King County, US, with the help of ML.

Constraints

The dataset has a mix of small, medium, large houses. The data span is just one year.

Scope

Finding fair price and what features of the property determine it.

Objectives

Giving house buyers, sellers, and intermediaries such as banks fair and scientifically deduced prices for profitable deal-making

Need of this study

Predict and win a million-dollar prize

- Predicting house price is so tough that Zillow has a $1 million prize in the US for improving their house evaluations
- Most buyers are inexperienced amateurs, and know little about the market.
- They tend to believe that prices will always rise, so they pay more than the house is worth.
- People in US cities were asked: “What percentage do you expect house prices to rise next year?” The mean expected rise was 38%. The actual was 5.7%.
- Even experienced property investors overestimate the property value, while banks miscalculate the mortgage value they should set.
- We will use regression techniques to try and build an optimal model that explains the variation in house prices.

Source: https://www.ibef.org/industry/real-estate-india.aspx

American residential real estate

The residential sector focuses on the buying and selling of properties used as homes or for non-professional purposes. The residential real estate sector is comprised of single-family homes, apartments, condominiums, planned unit developments, and more. As of 2021, this segment of the real estate industry has been steadily increasing in value for several years. In May 2021, median home prices were up at approximately $337,400, while the residential real estate market as a whole was valued at more than $33.8 trillion.

While there has been a lot of discussion on whether younger generations such as Millennials and Gen Z are still buying property, the trend towards urbanization and growth of large cities will definitely inform residential real estate prices for years to come.

Real estate sector and statistics

Can house prices be predicted using statistical techniques? Or are the prices driven by exuberant human behaviour. John Maynard Keynes developed the term 'Animal Spirits' to explain irrational behaviour in an economy.

This term has been used to explain many behaviours that cannot be explained by behavioural economics [1]. From this, several studies have been published on 'herd'-like mentality showed that an individual deciding is likely to override information to conform to the popular trend of thinking

Buying a house is one of the largest and most expensive purchases of the lifetime, usually, so being able to track the value of these assets gains importance. The existing solutions on predicting house prices include using domain knowledge as independent features and feeding them into machine learning algorithms as input to get prediction of the house value. However, these models do not evaluate the effects of feature engineering on the performance of the model, which in this project, we will do.

Price influencers

Economic environment, housing market, neighborhood, proximity to amenities, location, home size, usable space, age, condition, upgrades

Type of statistical problem

The objective of using the housing transaction data, including housing property such as the number of bedrooms, total area, and location to forecast the property price makes it becomes a classical regression problem. We will take the housing property data to not only train the traditional LR (linear regression) model but also develop some gradient boosting models to see if we can enhance the model's performance.

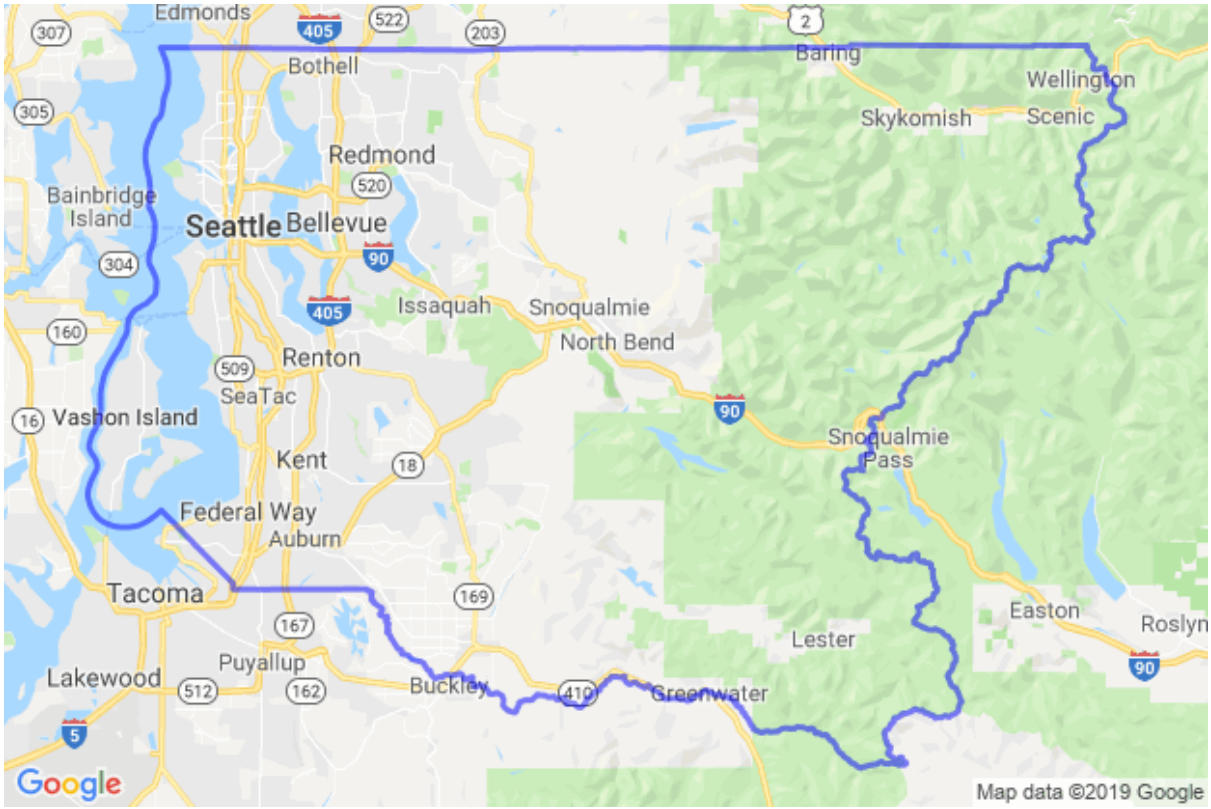
The difficulty

Linear regression assumes all the features in a dataset to be independent. However, in the house price prediction data, we have the problem of collinearity, which means some of the independent or predictor variables are highly correlated, which will affect the accuracy of the final prediction. Moreover, it is more complicated to deal with categorical variables in linear regression. We can't take these into the model and must do pre-processing before the training.

Generally, a linear regression model is limited to linear relationships. It only looks at linear relationships between dependent and independent variables and assumes there is a straight-line relationship between them. However, real-life scenarios are more complicated than that. A linear regression model is also sensitive to outliers. Surprising data points may affect the overall performance of the model.

Site of investigation

The longitude and latitude variables helped us detect that these house transaction figures are from King County of Washington area in the US (boundary map given), which is home to some of the world's priciest real estate, being the residence of the rich and the famous. The biggest city of the region is Seattle, which being on the coast, has quite a few waterfront houses.



Even though one might not be looking for a "cheap" property, there will always be cheap properties around in secondary locations. Everyone is looking for the right property at a good price. Properties to consider may be ones that are a little ugly or untidy but have good "bones" and are in good or superior locations. Look for an area that has a long, proven history of strong capital growth and is

one that is likely to continue to outperform the averages. This is largely because of the demographics in the area. These suburbs tend to be those where a large number of owner-occupier's desire to live in the area, because of lifestyle choices of the offer. People also look for suburbs where wages (and hence disposable incomes) are increasing above average. An ideal investment is one in which you can manufacture capital growth through refurbishment, renovations, or redevelopment. In January 2022, King County home prices were up 10.7% compared to last year, selling for a median price of $731K. On average, homes in King County sell after 7 days on the market compared to 14 days last year. There were 1,561 homes sold in January this year, down from 1,981 last yea

(Source: propertyupdate.com.au and Redfin.com)

Methodology

This part of the project includes:

1. Scrubbing data
2. Preparing data
3. Understanding data and answering business questions based on initial understanding

DATA SCRUBBING

1. Casting columns to the appropriate data types
2. Identifying and dealing with null and duplicated values appropriately
3. Removing columns that aren't required for modeling
4. Checking for normality with distplot
5. Checking for linearity with boxplot, correlation coefficient
6. Removing outliers that are more than 3 standard deviations away from the mean
7. Checking for and dealing with multicollinearity with heatmap
8. Select potential features for modeling
9. Normalizing the continuous
10. On-hot encoding categorical data

5-point summary

	count	mean	std	min	25%	50%	75%	max
cid	21613.000000	4580301520.864988	2876565571.312048	1000102.000000	2123049194.000000	3904930410.000000	7308900445.000000	9900000190.000000
price	21613.000000	540182.158793	367362.231718	75000.000000	321950.000000	450000.000000	645000.000000	7700000.000000
room_bed	21505.000000	3.371355	0.930289	0.000000	3.000000	3.000000	4.000000	33.000000
room_bath	21505.000000	2.115171	0.770248	0.000000	1.750000	2.250000	2.500000	8.000000
living_measure	21596.000000	2079.860761	918.496121	290.000000	1429.250000	1910.000000	2550.000000	13540.000000
lot_measure	21571.000000	15104.583283	41423.619385	520.000000	5040.000000	7618.000000	10684.500000	1651359.000000
sight	21556.000000	0.234366	0.766438	0.000000	0.000000	0.000000	0.000000	4.000000
quality	21612.000000	7.656857	1.175484	1.000000	7.000000	7.000000	8.000000	13.000000
ceil_measure	21612.000000	1788.366556	828.102535	290.000000	1190.000000	1560.000000	2210.000000	9410.000000
basement	21612.000000	291.522534	442.580840	0.000000	0.000000	0.000000	560.000000	4820.000000
yr_renovated	21613.000000	84.402258	401.679240	0.000000	0.000000	0.000000	0.000000	2015.000000
zipcode	21613.000000	98077.939805	53.505026	98001.000000	98033.000000	98065.000000	98118.000000	98199.000000
lat	21613.000000	47.560053	0.138564	47.155900	47.471000	47.571800	47.678000	47.777600
living_measure15	21447.000000	1987.065557	685.519629	399.000000	1490.000000	1840.000000	2360.000000	6210.000000
lot_measure15	21584.000000	12766.543180	27286.987107	651.000000	5100.000000	7620.000000	10087.000000	871200.000000
furnished	21584.000000	0.196720	0.397528	0.000000	0.000000	0.000000	0.000000	1.000000

The maximum price is 7,700,000 (7.7 million) and the minimum is 75,000, dollars we assume, since the houses are in the US.

The average price is $5,40,182 in that case, which is more than half a million.

It can also be the negotiating base price for the buyers as well as the sellers. The median price is 4,50,000, which means that half the deals were negotiated below this level and half the deals were sealed above this value.

The average living area you get in King County is 2079 square feet. The average price for each square feet is close to 260 (dollars).

Current situation

In December 2021, King County home prices were up 11.7% compared with last year, selling for a median price of $755K. On an average, homes in King County sell after 7 days on the market compared with 10 days last year (2021). There were 2,593 homes sold in December 2021, 14.9% down from 3,046 last year. From the 2014-2015 sales data that we have, this is quite a jump.

Approximately 85,128 homes (12%) are already at risk in King County, and within 30 years, close to 90,317 homes (13%) will be at risk. Flood risk in King County is increasing slower than the national average.

(Source: https://www.redfin.com/county/118/WA/King-County/housing-market)

The overall housing market is a collection of several connected sub markets. An increase in housing demand which increase the cost of housing is driven by biased speculation. Opposing the general theory of supply and demand, in housing, when prices rise demand increases and when price fall demand decreases. This can be explained by the psychological factors of buyers.

The global housing bubble was driven by emotion rather that sound investment decisions. Overconfidence in decision making coupled with the 'herd'-like behaviour exasperated the housing crash that started in 2008. A bubble cannot exist if buyers are making rational decisions. This would indicate that buyers are not making rational decisions and are simple conforming to the trend. B. Hedonic price theory and its limitations; In most studies, price modelling is used to explain the variation in price. The characteristics of a property is used as explanatory variables.

The main theory in this field is hedonic price theory. This theory can be traced back to the work done in 1966 and 1974. It is the main theory of reference to explain price in the housing market. In this theory the price of a good, in this instance a house, depends on its characteristics.

Because this theory uses the basic characteristics of a house such as number of bedroom and size, numerous research has been published on its limitations. Notable research focuses on the effect of location on price. Location factors include distance to work, transport systems, accessibility to schools and other amenities.

One caveat in using hedonic pricing models is that the results are location-specific and are difficult to generalize across different geographic regions suggests that that location is a key factor in determining price. Similarly, buyers were willing to pay more for properties with quick accessibility to public transport. However, various researchers found out that housing is a multidimensional commodity separated into a package of attributes that vary in both quantity and quality.

Data types

```
RangeIndex: 21613 entries, 0 to 21612
Data columns (total 23 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   cid               21613 non-null  int64
 1   dayhours          21613 non-null  object
 2   price             21613 non-null  int64
 3   room_bed          21505 non-null  float64
 4   room_bath         21505 non-null  float64
 5   living_measure    21596 non-null  float64
 6   lot_measure       21571 non-null  float64
 7   ceil              21571 non-null  object
 8   coast             21612 non-null  object
 9   sight             21556 non-null  float64
 10  condition         21556 non-null  object
 11  quality           21612 non-null  float64
 12  ceil_measure      21612 non-null  float64
 13  basement          21612 non-null  float64
 14  yr_built          21612 non-null  object
 15  yr_renovated      21613 non-null  int64
 16  zipcode           21613 non-null  int64
 17  lat               21613 non-null  float64
 18  long              21613 non-null  object
 19  living_measure15  21447 non-null  float64
 20  lot_measure15     21584 non-null  float64
 21  furnished         21584 non-null  float64
 22  total_area        21584 non-null  object
dtypes: float64(12), int64(4), object(7)
memory usage: 3.8+ MB
```

The data has 12 decimal, 4 integer, and 7 string values, while the set takes up more than 3.8 megabytes of computer memory space. We'll need to change the data types of some variables for the sake of our analysis.

Missing values

```
The dataset has 689 missing values.
```

Checking the count of missing values in each variable

living_measure15	166
room_bath	108
room_bed	108
condition	57
sight	57
ceil	42
lot_measure	42
total_area	29
furnished	29
lot_measure15	29
living_measure	17
yr_built	1
basement	1
ceil_measure	1
quality	1
coast	1

The maximum missing values are in living_measure15, followed by the 'room_bath' and 'room_bed' variables. There are single missing values in five variables. We'll treat all the missing values, later.

```
The size of the dataset is 497099

The dataset has 689 missing values

Percentage of missing values in the dataset is 0.14
```

	Skewness
lot_measure	13.072364
lot_measure15	9.519083
yr_renovated	4.549493
price	4.021716
sight	3.395181
room_bed	1.980749
basement	1.577898
furnished	1.525972
living_measure	1.472565
ceil_measure	1.446747
living_measure15	1.107979
quality	0.771129
room_bath	0.511689
zipcode	0.405661
cid	0.243329
lat	-0.485270

- If the skewness is between -0.5 and 0.5, the data are fairly symmetrical
- If the skewness is between -1 and — 0.5 or between 0.5 and 1, the data are moderately skewed
- If the skewness is less than -1 or greater than 1, the data are highly skewed

Kurtosis

```
cid                -1.260542
zipcode            -0.853479
lat                -0.676313
furnished           0.328622
quality             1.190813
room_bath           1.286012
living_measure15    1.598975
basement            2.715338
ceil_measure        3.402404
living_measure      5.246808
sight              10.887637
yr_renovated       18.701152
price              34.522444
room_bed           49.254149
lot_measure15     151.222905
lot measure       285.492738
```

- If the distribution is tall and thin it is called a leptokurtic distribution (Kurtosis > 3). Values in a leptokurtic distribution are near the mean or at the extremes.
- A flat distribution where the values are moderately spread out (unlike leptokurtic) is called platykurtic (Kurtosis <3) distribution.
- A distribution whose shape is in between a leptokurtic and a platykurtic is called a mesokurtic (Kurtosis=3) distribution, closer to a normal distribution.
- High kurtosis in a data set is an indicator that data has heavy outliers.
- Low kurtosis in a data set is an indicator that data has lack of outliers

Initial look at target variable price

```
count     21613.000000
mean     540182.158793
std      367362.231718
min       75000.000000
25%      321950.000000
50%      450000.000000
75%      645000.000000
max     7700000.000000
```

Mean value is higher than median because the data has extreme values on the higher side.

Unique values of price

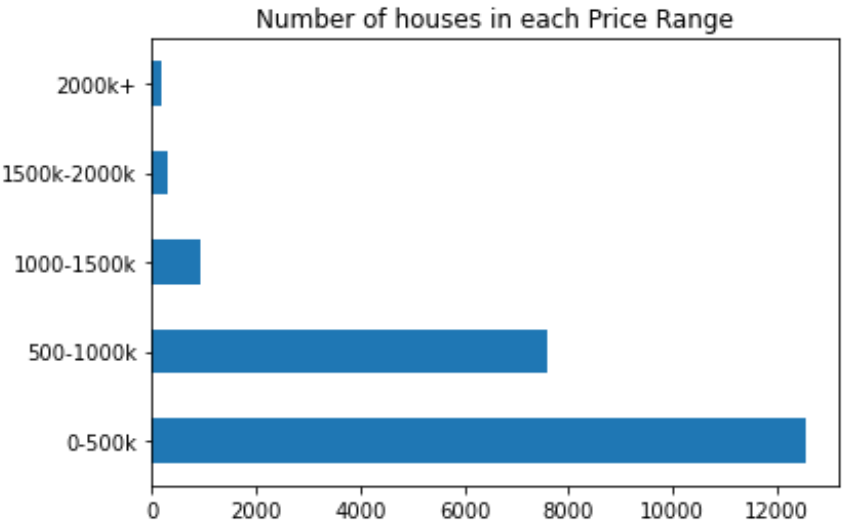
```
array([600000, 190000, 735000, ..., 725126, 332100, 685530], dtype=int64)
```

```
75000      1
78000      1
80000      1
81000      1
82000      1
          ..
5350000    1
5570000    1
6890000    1
7060000    1
7700000    1
```

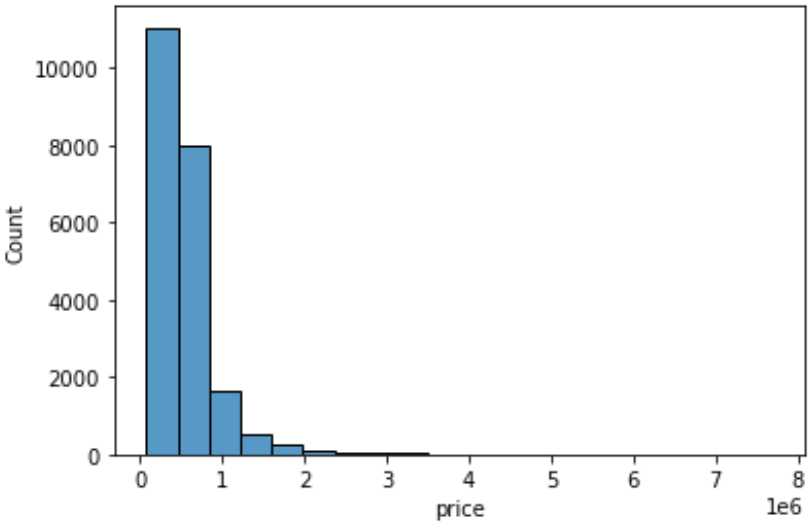
Seems to be all fine by the look of it. All values are single and unique.

Distribution of price at different percentiles

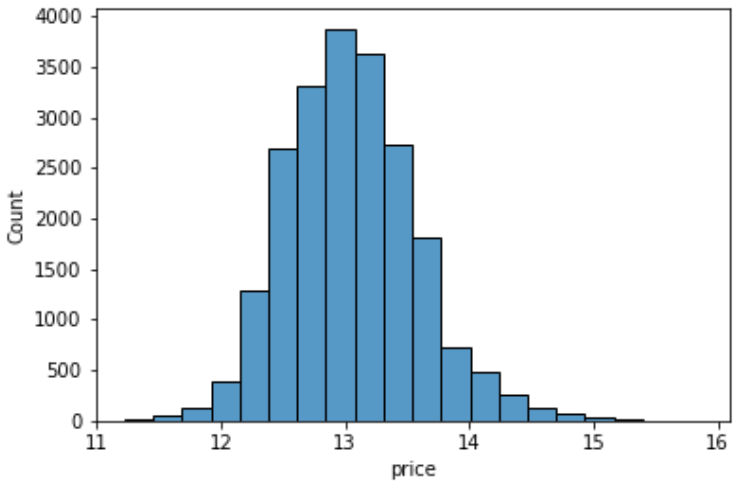
```
0.5% properties have a price lower than  133436.00
  1% properties have a price lower than  153500.36
  5% properties have a price lower than  210000.00
 10% properties have a price lower than  245000.00
 90% properties have a price lower than  887000.00
 95% properties have a price lower than  1160000.00
 99% properties have a price lower than  1968800.00
99.5% properties have a price lower than  2450000.00
```



Distribution of price



Temporary log transformation of price



Most prices are between 0 and 1 million. Log transformation normalises distribution.

Changing data types and treating missing values

Total area

$' value was replaced with NaNs or null values for imputation. The column was imputed with the output of the formula living_measure + lot_measure because it is given to be the sum of the outside area and the living area. The data type was changed to float.

Living measure and lot measure

We imputed the living_measure with output of the formula total_area- lot_measure, since if we subtract outside are from total area, we get the living area (refer to the data dictionary). Similarly, we imputed lot_measure with the output of the formula total_area - living_measure. Then we check again for the missing values and find that the two variables are no longer on that list. The missing values have been treated.

living_measure15	166
room_bath	108
room_bed	108
condition	57
sight	57
ceil	42
furnished	29
lot_measure15	29
yr_built	1
basement	1
ceil_measure	1
quality	1
coast	1

Living measure15 and lot measure15

Since both values were missing, we couldn't compute by the previous method, so we imputed living_measure15 with living_measure and lot_measure15 with lot_measure, assuming the area to be unchanged for these values. Then we check for the missing value and find none for these variables.

room_bath	108
room_bed	108
condition	57
sight	57
ceil	42
furnished	29
yr_built	1
basement	1
ceil_measure	1
quality	1
coast	1

House condition¶

3	13978
4	5655
5	1694
2	171
1	30
$	28

This feature takes values from only 1 to 5, so changing $ to modal value 3. Its instances are only 28, so adding it to 3 won't make a big difference.

3	14006
4	5655
5	1694
2	171
1	30

After this replacement, we impute the null values as well with modal value 3. Then, we check for the missing values and find the column taken care of.

3	14063
4	5655
5	1694
2	171
1	30

It might result in a high bias if we train the ML algorithms with missing data filled with mean. Mode is not impacted by the presence of outliers in the data.

Ceil measure and basement

Ceil_measure was imputed with the output of the formula total_area – basement, and likewise basement with total_area - ceil_measure. This is because one is house area with basement and the other is house area without basement. Then we check for missing values. One missing value remain in each case. It's because both values are missing in that record.

room_bath	108
room_bed	108
sight	57
ceil	42
furnished	29
yr_built	1
basement	1
ceil_measure	1
quality	1
coast	1

Description of ceil measure

```
count    21612.000000
mean      1788.366556
std        828.102535
min        290.000000
25%       1190.000000
50%       1560.000000
75%       2210.000000
max       9410.000000
```

We impute ceil_measure with output of the formula total_area - median of basement, and then we impute basement with output of the formula total_area - ceil_measure, since we have one of the two values already. Median is also robust to outliers. Rechecking for missing values, we find the problem solved.

```
room_bath     108
room_bed      108
sight          57
ceil           42
furnished      29
yr_built        1
quality         1
coast           1
```

Dayhours

Day-hour was changed to DateTime data type, string T000000 was removed, and month, day, year were extracted from the variable. The column was renamed to sell_date.

	cid	sell_date	price	room_bed	room_bath
0	3876100940	04-27-2015	600000	4.000000	1.750000
1	3145600250	03-17-2015	190000	2.000000	1.000000

Ceil

1	10647
2	8210
1.5	1905
3	610
2.5	161
$	30
3.5	8

Changing $ to modal value 1

1.000000	10677
2.000000	8210
1.500000	1905
3.000000	610
2.500000	161
3.500000	8

We imputed ceil with median value and rechecked for the missing values. The variable is treated for missing values successfully. The datatype was changed to float.

room_bath	108
room_bed	108
sight	57
furnished	29
yr_built	1
quality	1
coast	1

House quality

7.000000	8981
8.000000	6067
9.000000	2615
6.000000	2038
10.000000	1134
11.000000	399
5.000000	242
12.000000	90
4.000000	29
13.000000	13
3.000000	3
1.000000	1

Quality is the 13-point overall grade given to the house, based on King County's grading system

From grade 1 to 13: -

1-3 = Falls short of minimum building standards. Normally cabin or inferior structure.

4 = Generally older, low-quality construction. Does not meet building code.

5 = Low construction costs and workmanship. Small, simple design.

6 = Lowest grade currently meeting building code. Low quality materials and simple designs.

7 = Average grade of construction and design. Commonly seen in plats and older sub-divisions.

8 = Just above average in construction and design. Usually, better material in both the exterior and the interior finish work.

9 = Better architectural design with extra interior and exterior design and quality.

10 = Homes of this quality generally have high quality features. Finish work is better and more design quality is seen in the floor plans. Generally, have a larger square footage.

11 = Custom design and higher quality finish work with added amenities of solid woods, bathroom fixtures and more luxurious options.

12 = Custom design and excellent builders. All materials are of the highest quality and all conveniences are present.

13 = Generally custom designed and built. Mansion level. Large amount of highest quality cabinet work, wood trim, marble, entryways etc.

Checking for the missing value's features

```
             cid   sell_date   price  room_bed  room_bath  living_measure  \
21226  2474400250  06-30-2014  327500  3.000000   2.250000     2310.000000

       lot_measure     ceil coast  sight  condition  quality  ceil_measure  \
21226  7200.000000 1.500000   NaN    nan          3      nan   9510.000000

       basement yr_built  yr_renovated  zipcode       lat        long  \
21226  0.000000      NaN             0    98031 47.405100 -122.193000

       living_measure15  lot_measure15  furnished  total_area
21226       1960.000000    7201.000000   0.000000 9510.000000
```

Not a renovated property, but its condition is given as 3. The living area is average and it has no basement.

Checking for zipcode 98031

	lot_measure	ceil	coast	sight	condition	quality	ceil_measure
9	10506.000000	1.000000	0	0.000000	3	7.000000	1460.000000
40	7297.000000	1.000000	0	0.000000	3	7.000000	1130.000000
76	9618.000000	2.000000	0	0.000000	3	7.000000	1950.000000
116	7700.000000	1.000000	0	0.000000	3	7.000000	1240.000000
209	7314.000000	1.000000	0	0.000000	3	7.000000	1100.000000
...	...	...	...	...	...	...	...
21010	7128.000000	1.000000	0	0.000000	4	7.000000	940.000000
21226	7200.000000	1.500000	NaN	nan	3	nan	9510.000000
21294	7289.000000	2.000000	0	0.000000	3	7.000000	2020.000000
21394	6956.000000	1.000000	0	0.000000	4	7.000000	1400.000000
21607	6900.000000	1.000000	0	0.000000	4	7.000000	1130.000000

Neighbourhood property has been graded 7, generally, which is also the modal value. So we'll impute the missing values with mode, and that solves our problem. The data type was changed ton integer.

```
room_bath     108
room_bed      108
sight          57
furnished      29
yr_built        1
coast           1
dtype: int64
```

Year built

Unique values for yr_built: -

```
array([1966, 1948, 2009, 1924, 1994, 2005, 1978, 1983, 2012, 1912, 1990,
       1967, 1919, 1908, 1950, 2000, 2013, 1943, 1922, 1977, 2004, 1935,
       1964, 1945, 1987, 2008, 1940, 2003, 1988, 1985, 1998, 1995, 1946,
       1984, 1958, 1963, 1942, 2014, 1971, 1936, 1954, 1923, 2002, 1972,
       2007, 1930, 1962, 1999, 1953, 1965, 2010, 1997, 2006, 1979, 1996,
       1992, 1968, 1980, 1981, 1969, 2001, 1929, 1952, 1916, 1976, 1974,
       1920, 1931, 1975, 1960, 1900, '$', 1986, 1989, 1906, 1955, 1956,
       1915, 1941, 1993, 2011, 1925, 1947, 1991, 1926, 1927, 1951, 1961,
       1932, 1917, 1928, 1959, 1921, 1911, 1949, 1982, 1913, 1957, 1914,
       1938, 1973, 1937, 1944, 1970, 1901, 1907, 1939, 1918, 1934, 1904,
       2015, 1909, 1910, 1905, 1902, 1933, 1903, nan], dtype=object)
```

We detect the presence of a '$' sign in the column, which we change to NaN to prepare it for imputation. NaN, because it can take both object and numerical values.

Checking for house location and zip code: -

```
             cid   sell_date   price  room_bed  room_bath  living_measure  \
1906  7905200205  10-21-2014  410000       nan        nan     1230.000000

      lot_measure     ceil coast    sight  condition  quality  ceil_measure  \
1906  7020.000000 1.000000     0 0.000000          3        7   1090.000000

       basement yr_built  yr_renovated  zipcode       lat        long  \
1906 140.000000      NaN             0    98116 47.571900 -122.390000

      living_measure15  lot_measure15  furnished  total_area
1906       1230.000000    5850.000000   0.000000 8250.000000
```

```
          basement yr_built  yr_renovated  zipcode       lat        long  \
38        0.000000     1946             0    98116 47.577200 -122.381000
47     1020.000000     2014             0    98116 47.579000 -122.389000
160       0.000000     1915             0    98116 47.570400 -122.409000
183    1220.000000     1948             0    98116 47.570900 -122.406000
269     690.000000     1948             0    98116 47.568300 -122.396000
...            ...      ...           ...      ...       ...         ...
21471  1040.000000     1926             0    98116 47.579100 -122.392000
21491   100.000000     1941             0    98116 47.584500 -122.395000
21504  2180.000000     2005             0    98116 47.574400 -122.406000
21534     0.000000     2007             0    98116 47.560500 -122.396000
21551     0.000000     1927             0    98116 47.572800 -122.388000
```

Within the neighbourbood, the year built has a wide variation, so we'll impute the missing value with median. Checking for missing values, we find that it solves the problem. The data type was changed to integer.

```
room_bath     108
room_bed      108
sight          57
furnished      29
coast           1
```

Year renovated

Unique values: - 70

```
array([   0, 1993, 2014, 1983, 1992, 2000, 2011, 1994, 2009, 1944, 1971,
       2003, 1955, 1985, 2008, 2015, 2005, 1979, 1998, 1968, 2010, 1989,
       2002, 1987, 1999, 1996, 1940, 1986, 1988, 1969, 1995, 2004, 2007,
       2013, 2001, 1990, 1958, 2012, 1967, 1991, 1970, 1984, 2006, 1982,
       1951, 1960, 1956, 1997, 1980, 1959, 1974, 1973, 1975, 1981, 1963,
       1957, 1976, 1948, 1945, 1977, 1978, 1972, 1965, 1964, 1953, 1950,
       1962, 1946, 1934, 1954], dtype=int64)
```

Zipcode

Unique values: - 70

```
array([98034, 98118, 98002, 98030, 98103, 98006, 98042, 98031, 98065,
       98109, 98058, 98001, 98105, 98115, 98032, 98033, 98199, 98053,
       98056, 98102, 98038, 98092, 98003, 98075, 98059, 98008, 98011,
       98014, 98023, 98116, 98198, 98126, 98052, 98108, 98133, 98074,
       98077, 98106, 98045, 98146, 98155, 98117, 98027, 98040, 98072,
       98005, 98055, 98070, 98028, 98166, 98019, 98136, 98107, 98004,
       98125, 98112, 98024, 98177, 98122, 98168, 98029, 98007, 98178,
       98010, 98188, 98039, 98144, 98022, 98148, 98119], dtype=int64)
```

Longitude

Unique values: -

```
array([-122.183, -122.274, -122.256, -122.213, -122.285, '$', -122.333,
       -122.165, -122.15, -122.178, -121.87, -122.352, -122.122, -122.275,
       -122.234, -122.324, -122.321, -122.277, -122.196, -122.398,
       -122.019, -122.181, -122.325, -122.191, -122.026, -122.084, -122.3,
       -122.287, -122.011, -122.127, -122.281, -122.221, -121.859,
       -121.913, -122.39, -121.871, -122.013, -122.381, -122.279,
       -122.194, -122.201, -122.125, -122.379, -122.121, -122.024,
...
```

Longitude has a special character, so we change it to NaN first.

```
array([-122.183, -122.274, -122.256, -122.213, -122.285, nan, -122.333,
       -122.165, -122.15, -122.178, -121.87, -122.352, -122.122, -122.275,
       -122.234, -122.324, -122.321, -122.277, -122.196, -122.398,
       -122.019, -122.181, -122.325, -122.191, -122.026, -122.084, -122.3,
       -122.287, -122.011, -122.127, -122.281, -122.221, -121.859,
       -121.913, -122.39, -121.871, -122.013, -122.381, -122.279,
       -122.194, -122.201, -122.125, -122.379, -122.121, -122.024,
...
```

We impute the NaN with median and see that it solves the problem. The data type of longitude was changed to float.

```
room_bath     108
room_bed      108
sight          57
furnished      29
coast           1
```

Furnished

Unique values: -

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

```
0.000000    17338
1.000000     4246
```

Since 0 or non-furnished house is the most common value, we fill the missing value with 0, assuming it to be non-furnished. It is the most probable class to which the data point can belong.

```
room_bath    108
room_bed     108
sight         57
coast          1
```

Sight

Unique values: -

```
array([ 0.,  4.,  2.,  3.,  1., nan])
```

We assume the property to be non-viewed and impute 0, because most property in the area is also non-viewed

```
room_bath    108
room_bed     108
coast          1
```

Coast

. Unique values: -

```
array([0, 1, '$', nan], dtype=object)
```

Replaced '$' with NanNs and imputed those missing values with 0, assuming those to be non-coastal or non-waterfront houses.

```
room_bath    108
room_bed     108
```

Bathrooms per bedroom

. Unique values: -

```
array([1.75, 1.  , 2.75, 2.5 , 1.5 , 3.5 , 2.  , 2.25, 3.  , 4.  , 3.25,
       3.75,  nan, 5.  , 0.75, 5.5 , 4.25, 4.5 , 4.75, 8.  , 6.75, 5.25,
       6.  , 0.  , 1.25, 5.75, 7.5 , 6.5 , 0.5 , 7.75, 6.25])
```

Value counts: -

2.500000	5358
1.000000	3829
1.750000	3031
2.250000	2039
2.000000	1917
1.500000	1439
2.750000	1178
3.000000	750
3.500000	726
3.250000	588
3.750000	155
4.000000	135
4.500000	100
4.250000	78
0.750000	72
4.750000	23
5.000000	21
5.250000	13
0.000000	10
5.500000	10
1.250000	9
6.000000	6
0.500000	4
5.750000	4
8.000000	2
6.250000	2
6.750000	2
6.500000	2
7.500000	1
7.750000	1

The column was imputed with modal value 2.5. Checking for any more missing value.

```
room_bed      108
```

Bedrooms per house

Unique values: -

```
array([ 4.,  2.,  3.,  1.,  5.,  6., nan,  7., 10.,  8.,  0.,  9., 33.,
       11.])
```

3.000000	9767
4.000000	6854
2.000000	2747
5.000000	1595
6.000000	270
1.000000	197
7.000000	38
0.000000	13
8.000000	13
9.000000	6
10.000000	3
11.000000	1
33.000000	1

We see that the 1620-sqft house with 33 bedrooms has only 1.75 bathrooms, which makes little sense. We can assume that there '33' is a typo, maybe 3 bedrooms with 1.75 bathrooms, which is the neighbourhood norm, since 13 bedrooms with 1.75 bathrooms or 33 bedrooms with 17.5 bathrooms are both improbable looking. Since this is the outlier that impacts our data spread, we will replace 33 with 3, the modal value of 'bedrooms'. Also 1620-sqft house can't have 33 bathrooms, while 3 is the norm for this specification.

	cid	sell_date	price	room_bed	room_bath	living_measure	lot_measure	ceil	coast	sight	condition	quality
16913	2402100895	06-25-2014	640000	33.000000	1.750000	1620.000000	6000.000000	1.000000	0	0.000000	5	7

3.000000	9768
4.000000	6854
2.000000	2747
5.000000	1595
6.000000	270
1.000000	197
7.000000	38
0.000000	13
8.000000	13
9.000000	6
10.000000	3
11.000000	1

```
Missing values in the house pricing dataset: 0
```

New data info

```
RangeIndex: 21613 entries, 0 to 21612
Data columns (total 23 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   cid               21613 non-null  int64
 1   sell_date         21613 non-null  object
 2   price             21613 non-null  int64
 3   room_bed          21613 non-null  float64
 4   room_bath         21613 non-null  float64
 5   living_measure    21613 non-null  float64
 6   lot_measure       21613 non-null  float64
 7   ceil              21613 non-null  float64
 8   coast             21613 non-null  int64
 9   sight             21613 non-null  float64
 10  condition         21613 non-null  int64
 11  quality           21613 non-null  int64
 12  ceil_measure      21613 non-null  float64
 13  basement          21613 non-null  float64
 14  yr_built          21613 non-null  int32
 15  yr_renovated      21613 non-null  int64
 16  zipcode           21613 non-null  object
 17  lat               21613 non-null  float64
 18  long              21613 non-null  float64
 19  living_measure15  21613 non-null  float64
 20  lot_measure15     21613 non-null  float64
 21  furnished         21613 non-null  float64
 22  total_area        21613 non-null  float64
dtypes: float64(14), int32(1), int64(6), object(2)
memory usage: 3.7+ MB
```

Missing values by columns

```
cid               0
sell_date         0
price             0
room_bed          0
room_bath         0
living_measure    0
lot_measure       0
ceil              0
coast             0
sight             0
condition         0
quality           0
ceil_measure      0
basement          0
yr_built          0
yr_renovated      0
zipcode           0
lat               0
long              0
living_measure15  0
lot_measure15     0
furnished         0
total_area        0
dtype: int64
```

The duplicate mystery

```
Duplicate values in housing dataset: 0
```

There are **21613** samples, while 'cid' has only **21436** counts, which means there are duplicates in the 'cid' column or, in other words, these **177** houses were perhaps sold more than once.

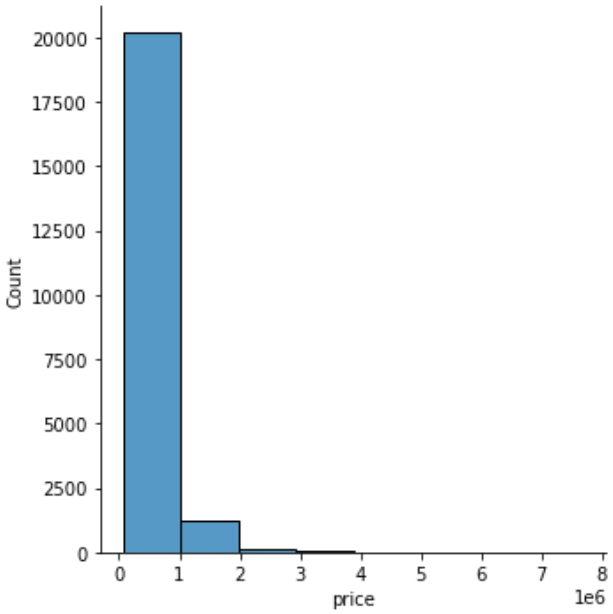
Renaming some columns

	cid	sell_date	price	bedrooms	bathrooms	house_in_sqft	lot_in_sqft	floors	coast	viewed	condition	quality	size_sans_basement
0	3876100940	04-27-2015	600000	4.000000	1.750000	3050.000000	9440.000000	1.000000	0	0.000000	3	8	1800.000000
1	3145600250	03-17-2015	190000	2.000000	1.000000	670.000000	3101.000000	1.000000	0	0.000000	4	6	670.000000

basement_size	yr_built	yr_renovated	zipcode	lat	long	house_size_2015	lot_size_2015	furnished	total_area
1250.000000	1966	0	98034	47.722800	-122.183000	2020.000000	8660.000000	0.000000	12490.000000
0.000000	1948	0	98118	47.554600	-122.274000	1660.000000	4100.000000	0.000000	3771.000000

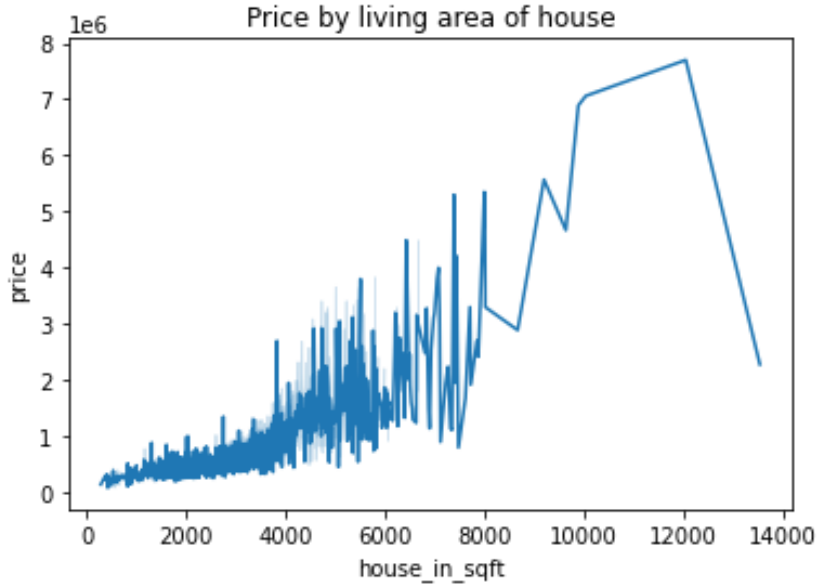
```
RangeIndex: 21613 entries, 0 to 21612
Data columns (total 23 columns):
 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   cid                  21613 non-null  int64
 1   sell_date            21613 non-null  object
 2   price                21613 non-null  int64
 3   bedrooms             21613 non-null  float64
 4   bathrooms            21613 non-null  float64
 5   house_in_sqft        21613 non-null  float64
 6   lot_in_sqft          21613 non-null  float64
 7   floors               21613 non-null  float64
 8   coast                21613 non-null  int64
 9   viewed               21613 non-null  float64
 10  condition            21613 non-null  int64
 11  quality              21613 non-null  int64
 12  size_sans_basement   21613 non-null  float64
 13  basement_size        21613 non-null  float64
 14  yr_built             21613 non-null  int32
 15  yr_renovated         21613 non-null  int64
 16  zipcode              21613 non-null  object
 17  lat                  21613 non-null  float64
 18  long                 21613 non-null  float64
 19  house_size_2015      21613 non-null  float64
 20  lot_size_2015        21613 non-null  float64
 21  furnished            21613 non-null  float64
 22  total_area           21613 non-null  float64
dtypes: float64(14), int32(1), int64(6), object(2)
memory usage: 3.7+ MB
```

Price against other variables



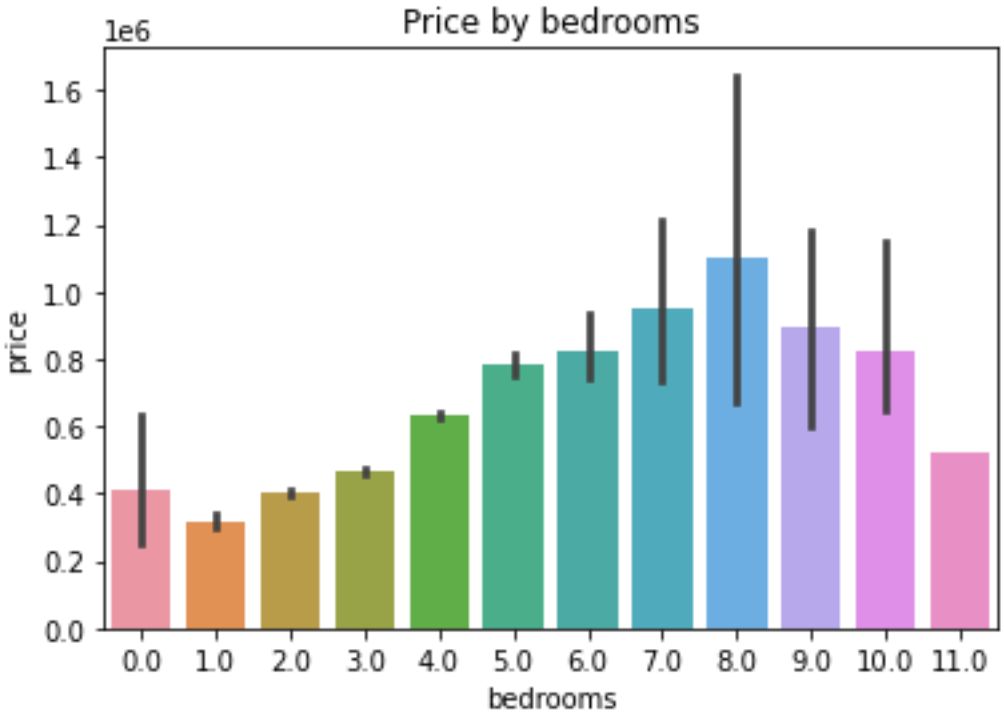
You can see that most of the houses are priced between 0 and 1 million. Next, let' see if there is any relationship between the area of the house in square feet and the price. We will use the "lineplot" from the "Seaborn" library to view this relationship.

Price by living area



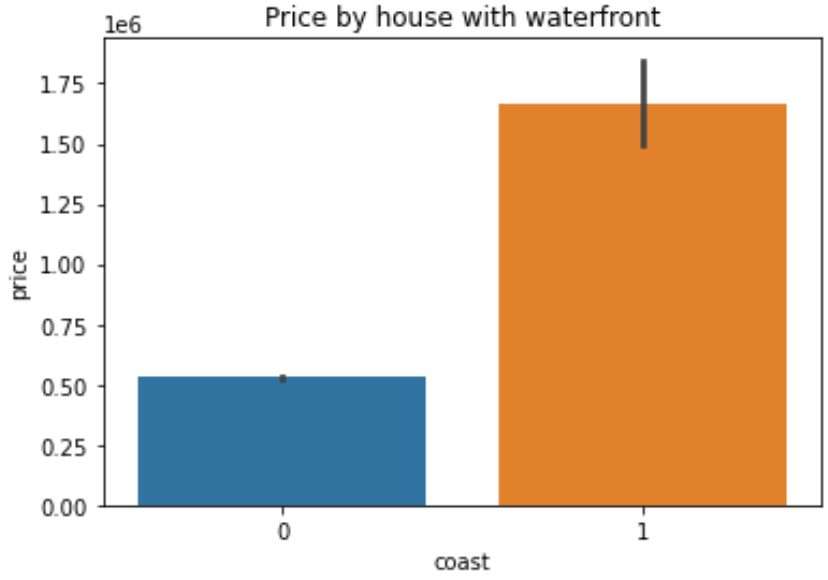
From the output, it can be seen that there is a positive correlation between the area of the house and the price. However, if the area is too big, the price starts to decrease. One of the reasons can be the fact that there are very few buyers of the very big houses since huge houses are too expensive to maintain. Next, let's find the relationship between the number of bedrooms and the price. Since the unique values for the bedroom columns are not too many, we can use a bar plot to draw this relationship.

Price by bedrooms



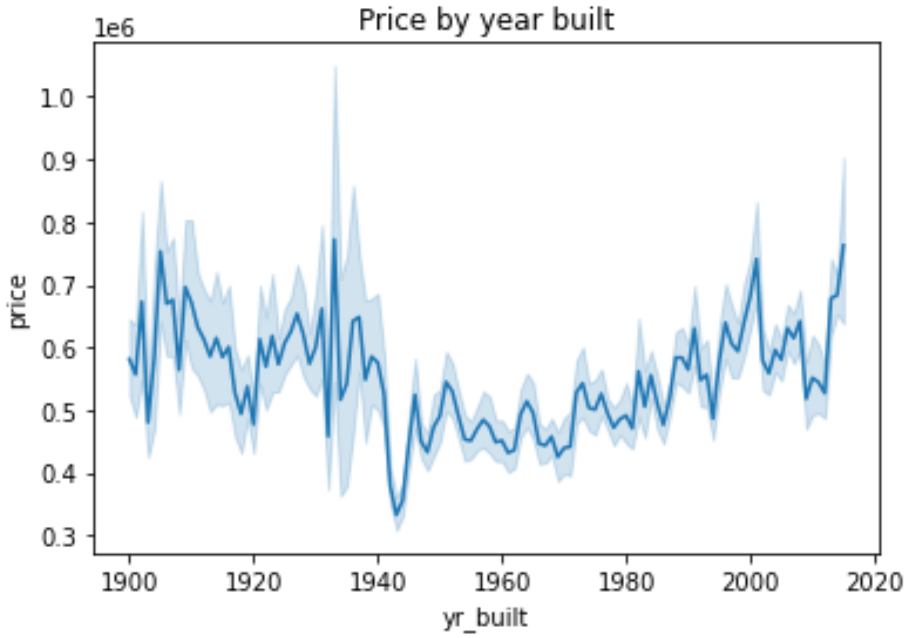
The price of the house increases with the increase in the number of bedrooms and decreases in case of too many bedrooms. The reason can be the same, the houses with too many bedrooms are big and not so easy to maintain. Next, let's see if we can find any difference between the prices of the houses with a waterfront view and the houses without a waterfront view.

Price by house with waterfront



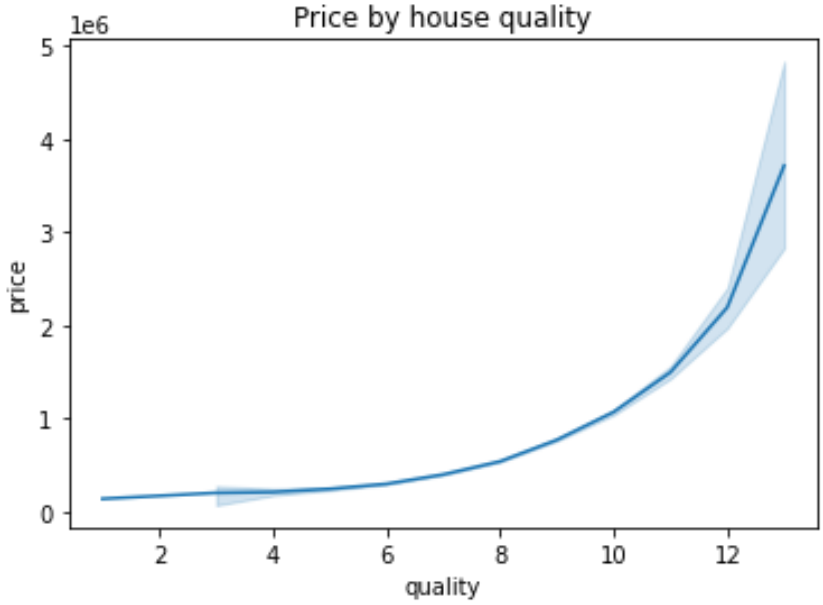
The houses with waterfront (orange bar) are far more expensive than those without the waterfront (blue bar). This shows that coast can really be a useful feature to predict the house price. Now we plot the relationship between the built year and the price of the house.

Price by year built



The houses that are too old are expensive may be due to historical value. Similarly, the houses that are relatively newer are expensive too, which is self-explanatory. However, the houses that are neither too old nor too new have a lower price value, since those have neither any historical value nor the novelty factor. Next, let's find the relationship between quality and the price of the house. The grade is the value assigned by the King County Administration based on various factors. Let's plot a line plot between grade and house price.

Price by quality



The quality and the price of the house have a clear positive correlation as evident from the output.

Feature engineering

We build new features and compare those as well against price.

House age

house_size_2015	lot_size_2015	furnished	total_area	house_age
2020.000000	8660.000000	0.000000	12490.000000	56
1660.000000	4100.000000	0.000000	3771.000000	74
2620.000000	2433.000000	0.000000	5455.000000	56
2030.000000	3794.000000	0.000000	5461.000000	13
1120.000000	5100.000000	0.000000	5710.000000	98

House age = 2022 – year built

```
Variable house_age has 0.0 negative values
```

2015 area

house_size_2015	lot_size_2015	furnished	total_area	house_age	2015_area
2020.000000	8660.000000	0.000000	12490.000000	56	10680.000000
1660.000000	4100.000000	0.000000	3771.000000	74	5760.000000
2620.000000	2433.000000	0.000000	5455.000000	56	5053.000000
2030.000000	3794.000000	0.000000	5461.000000	13	5824.000000
1120.000000	5100.000000	0.000000	5710.000000	98	6220.000000

2015_area= house_size_2015 + lot_size_2015

Has basement

has_basement = 1 where basement_size > 0, else it is 0

Renovated

Renovated = 1 where yr_renovated >0, else it is 0

Diff_living (difference to living after modification to house)

diff_living = house_in_sqft - house_size_2015

has_basement	renovated	diff_living
1	0	1030.000000
0	0	-990.000000
0	0	420.000000
0	0	-290.000000
0	0	0.000000

Square feet per floor

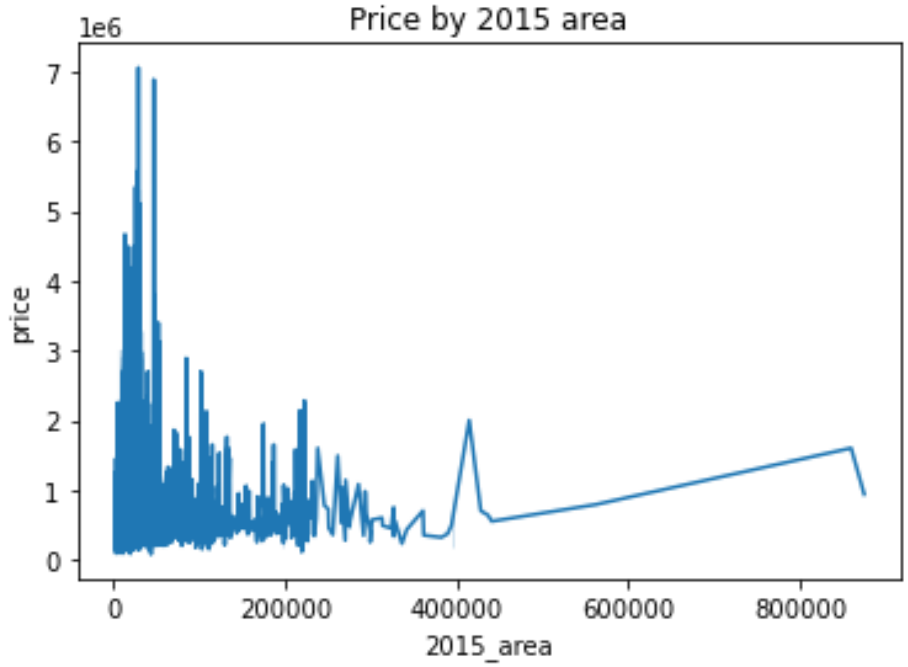
sqft_per_flr = size_sans_basement + basement_size/floors

Outside space

outside_space = 'lot_in_sqft' - sqft_per_flr'

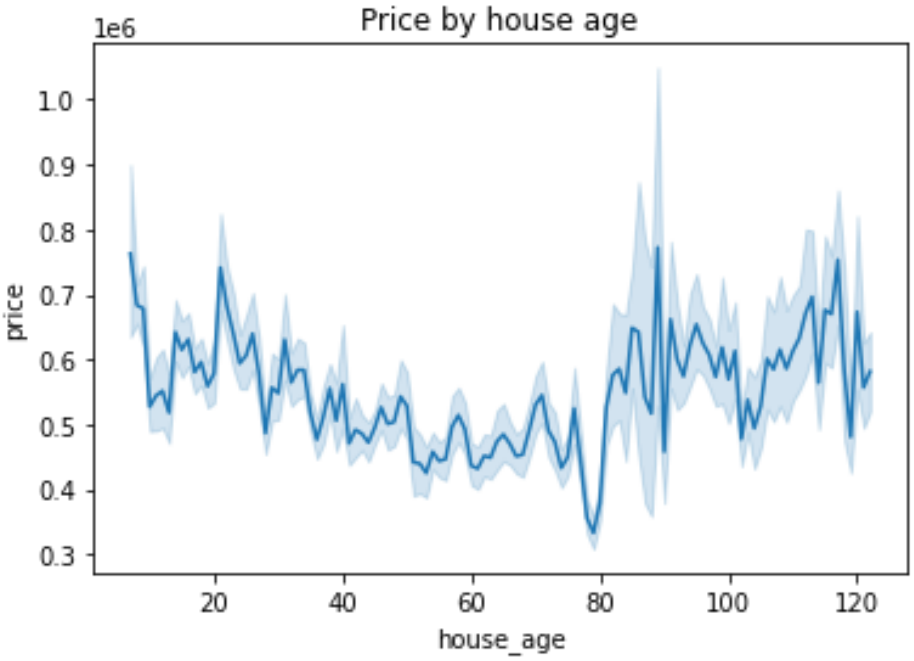
sqft_per_flr	outside_space
3050.000000	6390.000000
670.000000	2431.000000
1520.000000	895.000000
870.000000	2851.000000
1120.000000	3470.000000

Price by 2015 area



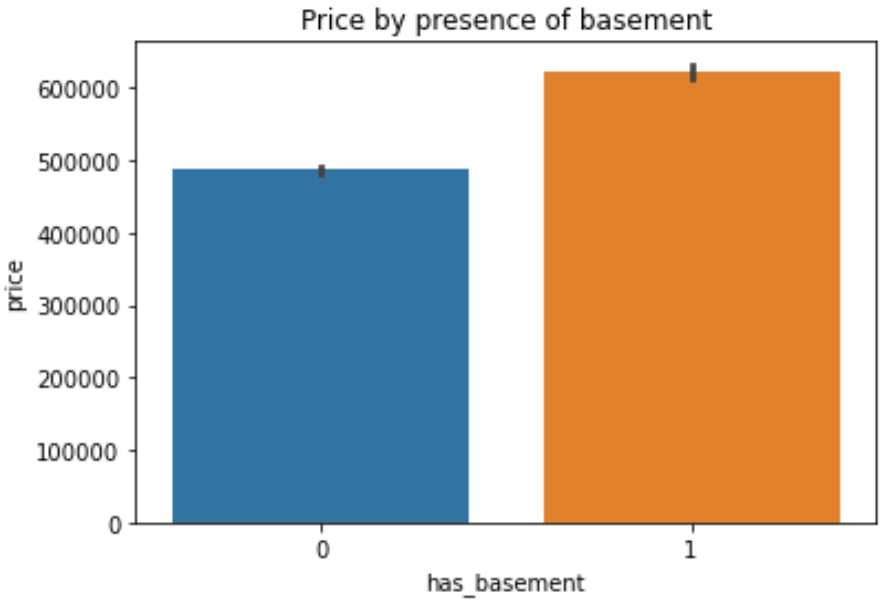
he 2015 area doesn't have much impact on the house price.

Price by house age



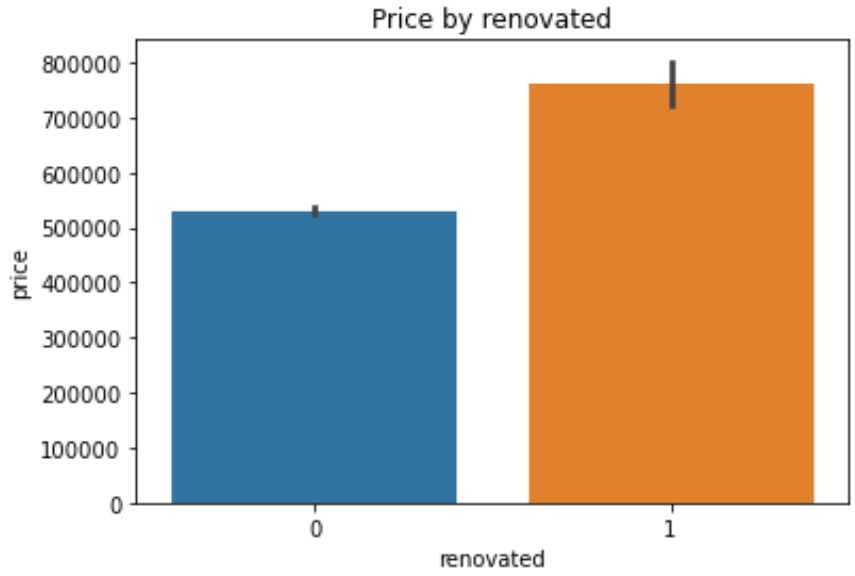
The houses that are too old are expensive may be due to historical value. Similarly, the houses that are relatively newer are expensive too, which is self-explanatory. However, the houses that are neither too old nor new have a lower price value since they have neither any historical value associated with them, nor they are new. The feature is quite similar to year built.

Price by presence of basement



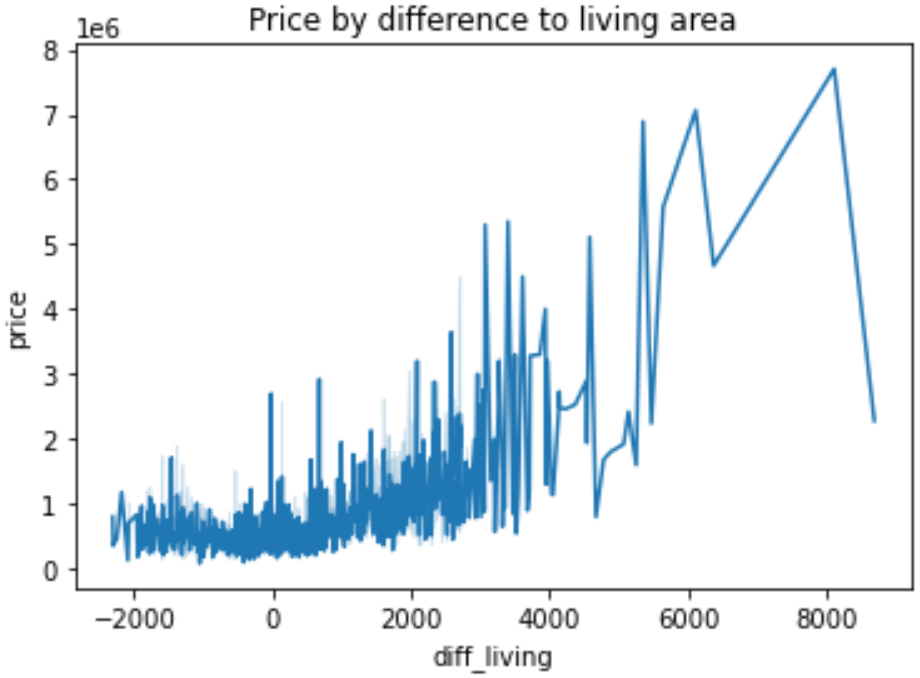
Presence of basement does increase house value.

Price by renovated



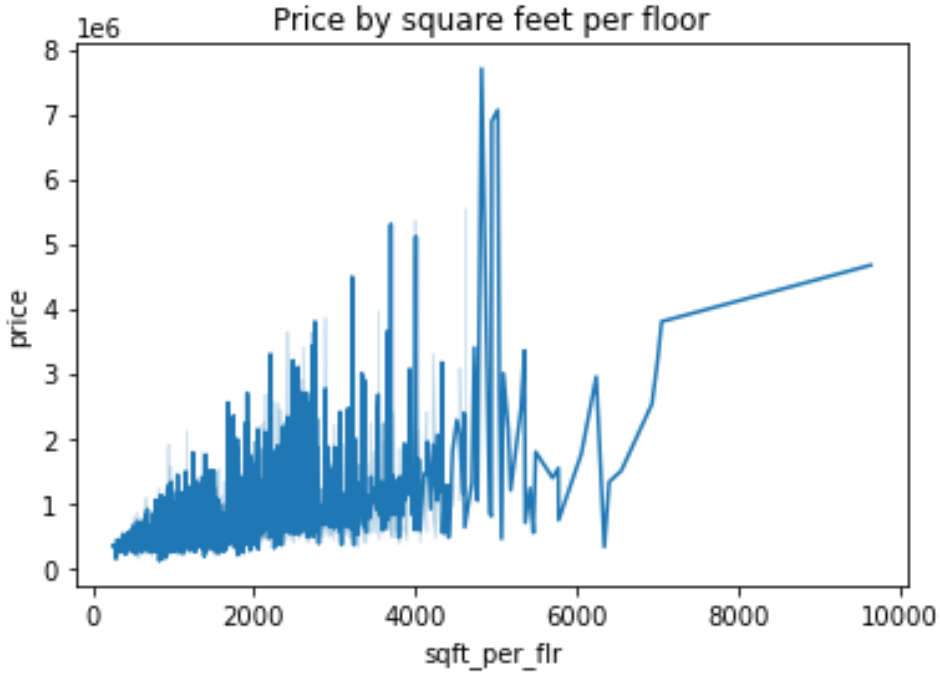
Renovated property does get a higher price.

Price by difference to living area



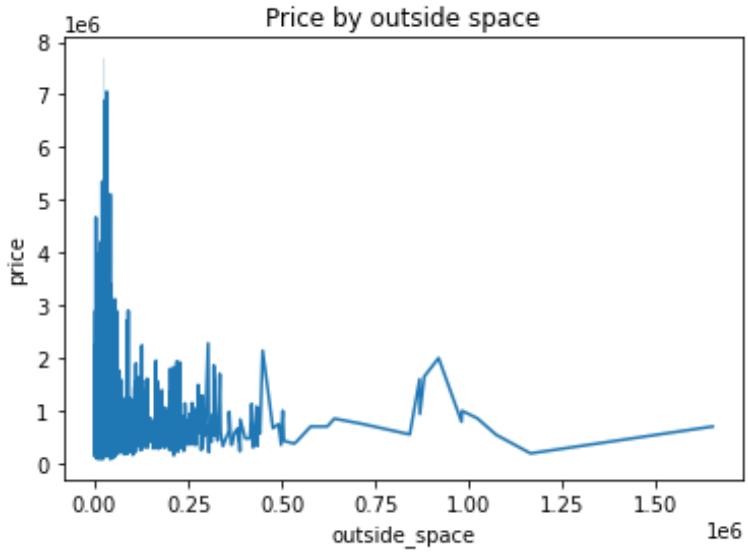
The difference made to the living area after renovation has a clear impact on price.

Price by square feet per floor



Price increases with square feet per floor, generally, so it's a good feature to look at

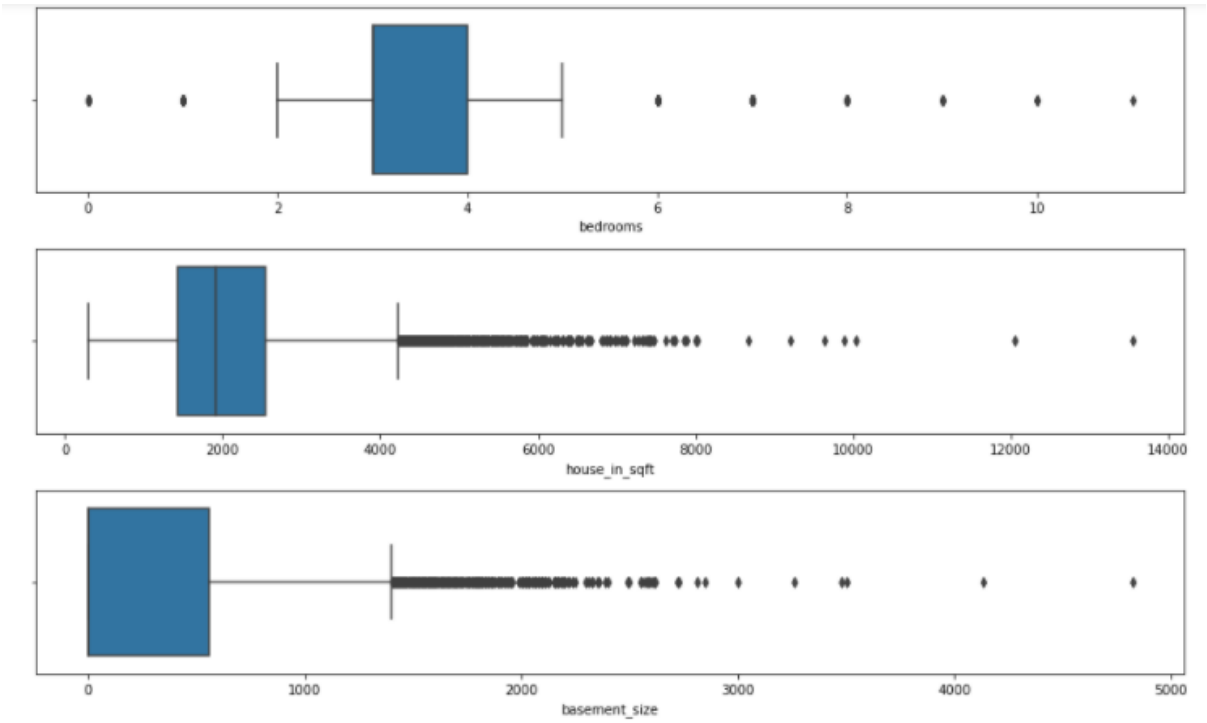
Price by outside space

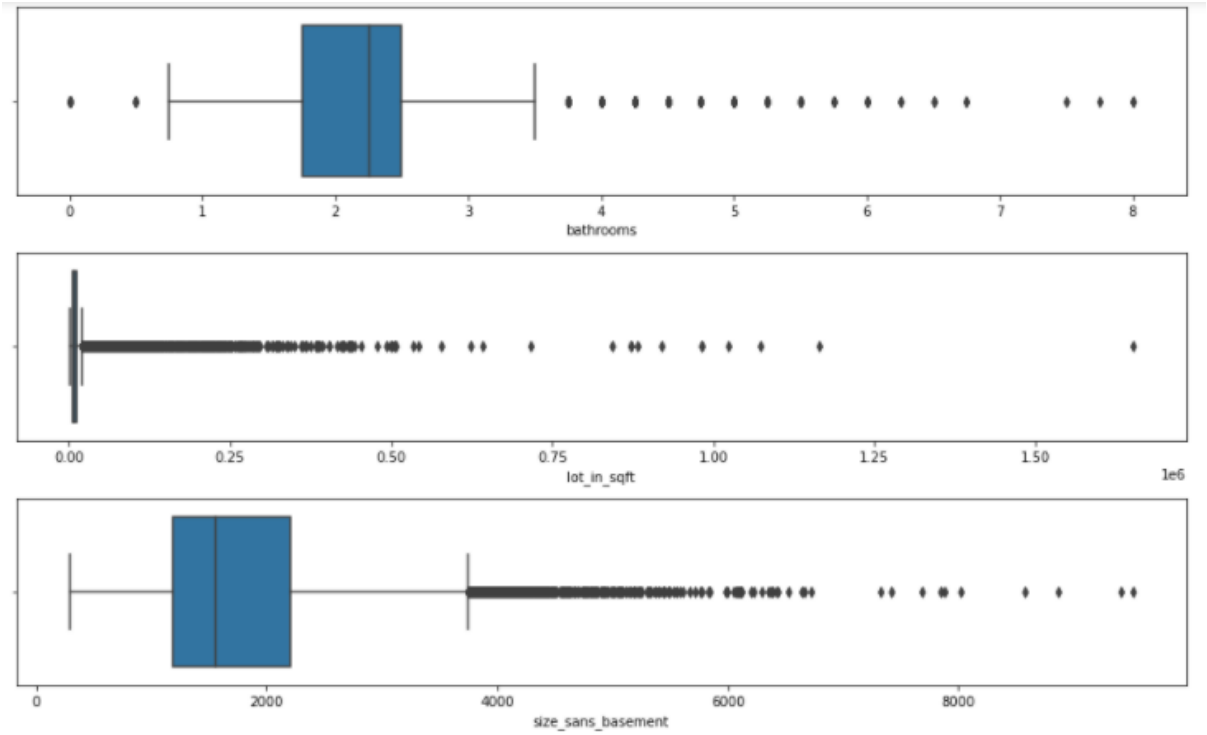


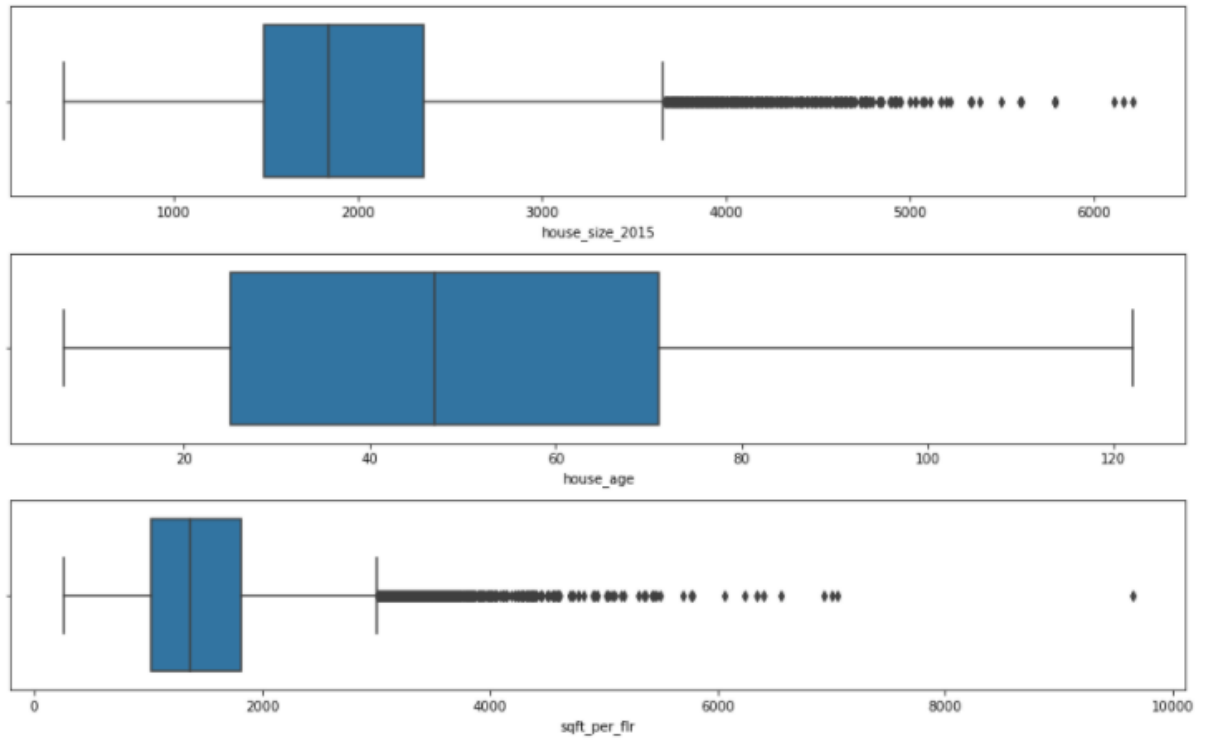
Outside space hasn't mattered to the customers and has little impact on price

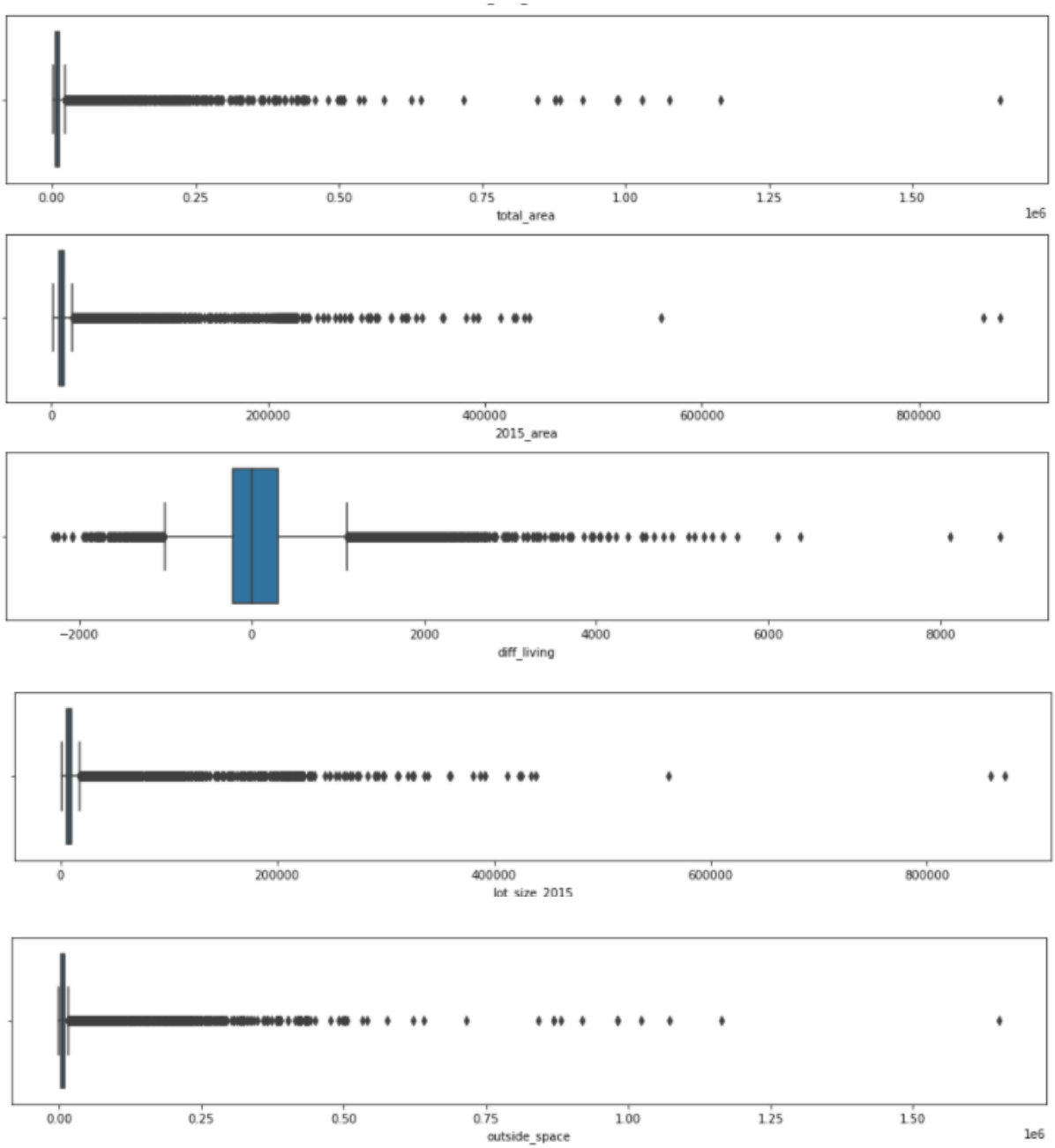
Handling outliers

Before treatment: -




0
1
2
3
4
5
6
7
8
bathrooms
0.00
0.25
0.50
0.75
1.00
1.25
1.50
lot_in_sqft
1e6
0
2000
4000
6000
8000
size_sans_basement


1000
2000
3000
4000
5000
6000
house_size_2015
20
40
60
80
100
120
house_age
0
2000
4000
6000
8000
10000
sqft_per_flr



All attributes have outliers but variation is expected, since not all houses will be similar.

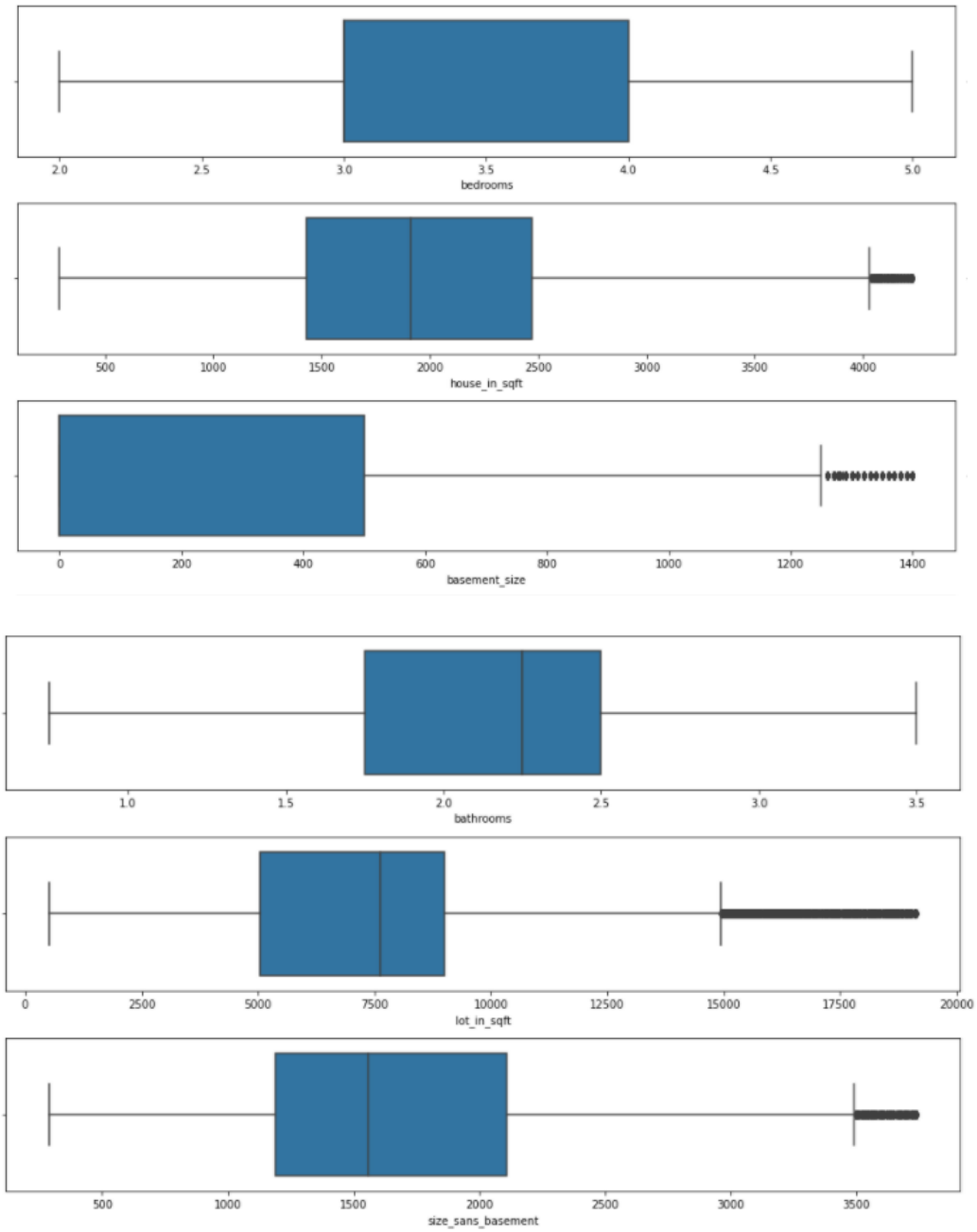
Let's check the number of outliers per column

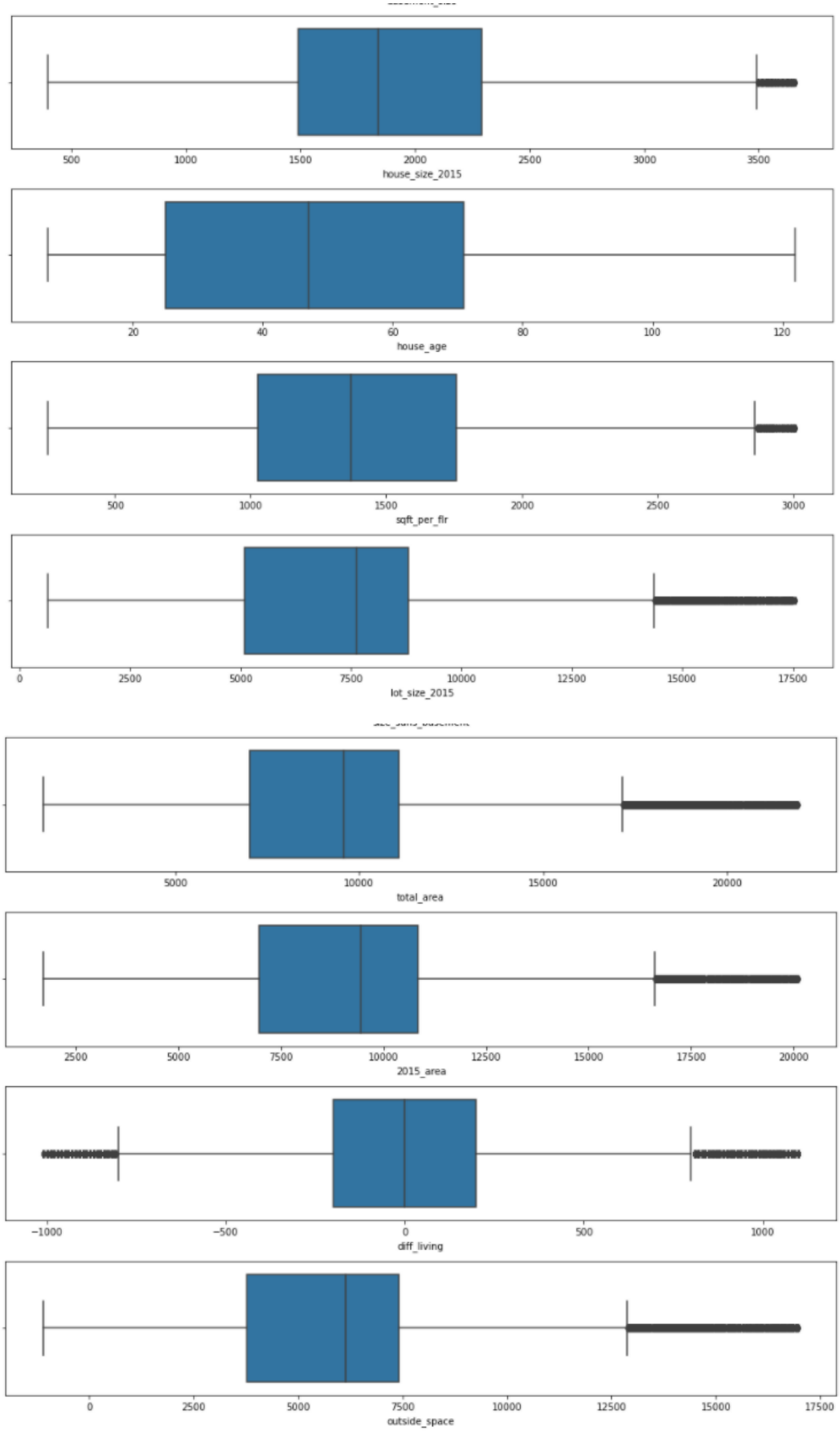
2015_area	2199	house_in_sqft	572		
basement_size	496	house_size_2015	546		
bathrooms	569	lat	2		
bedrooms	541	long	256		
cid	0	lot_in_sqft	2425		
coast	161	lot_size_2015	2196	sqft_per_flr	577
condition	30	outside_space	2455	total_area	2419
diff_living	1545	price	1159	viewed	2119
floors	0	quality	1911	yr_built	0
furnished	4246	renovated	914	yr_renovated	914
has_basement	0	sell_date	0	zipcode	0
house_age	0	size_sans_basement	612	dtype: int64	

```
Total outliers in the dataset are 28864

The size of the housing dataset is 648390

Percentage of outliers in the dataset is 4.30804041175935
```

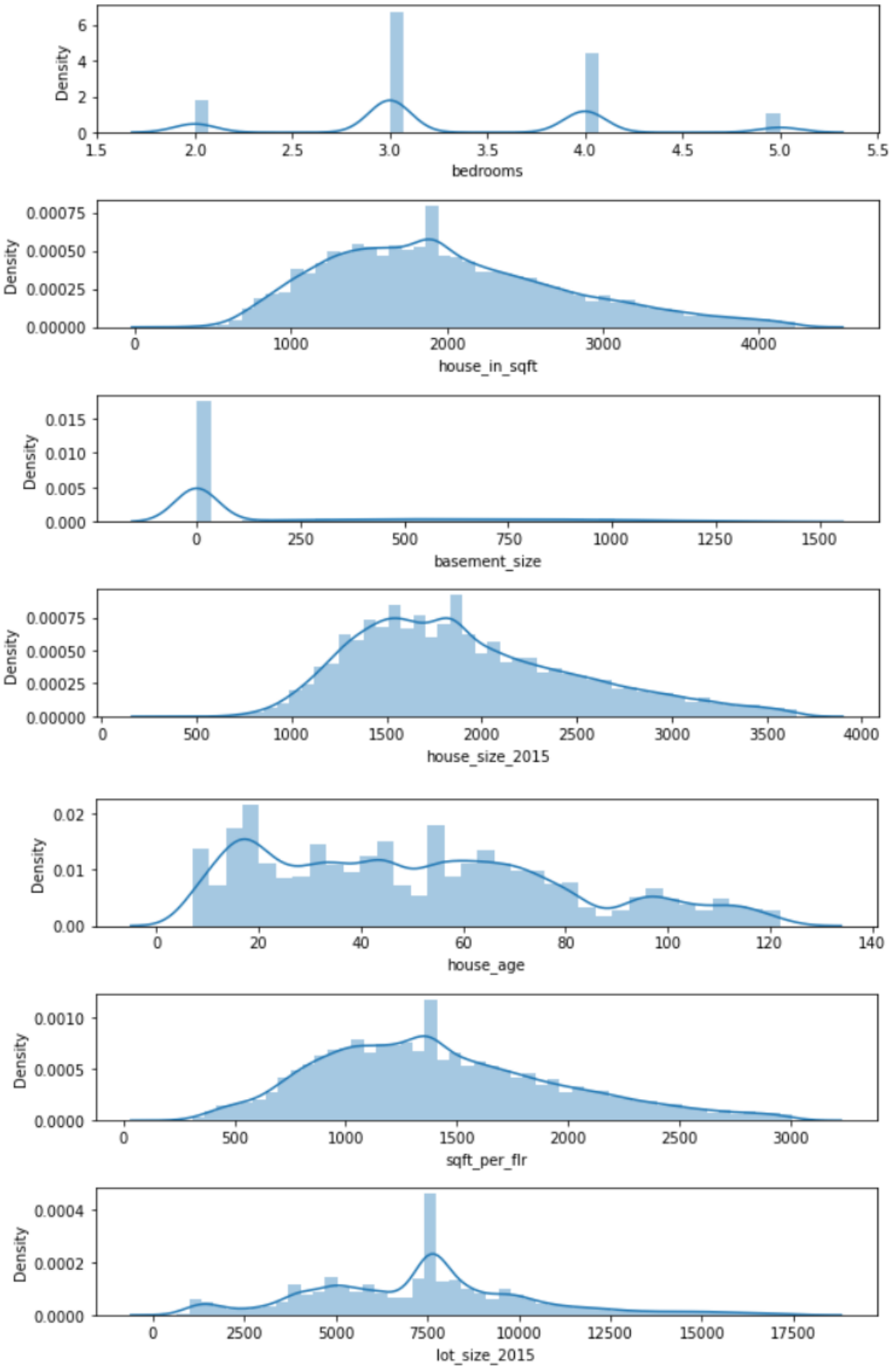
4% is a small number, but for the sake of keeping all information, we'll convert those outliers to median

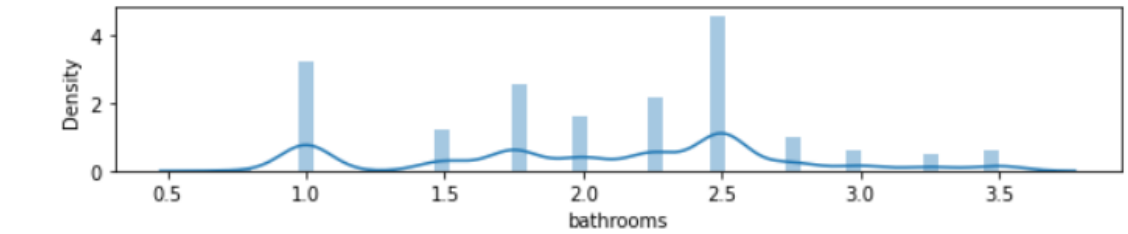
Boxplots after treating outliers

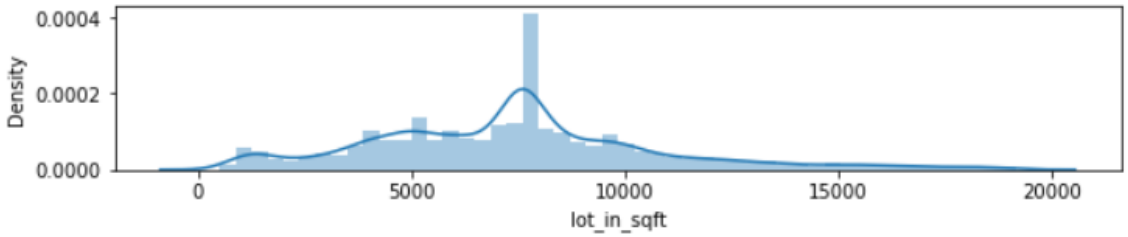


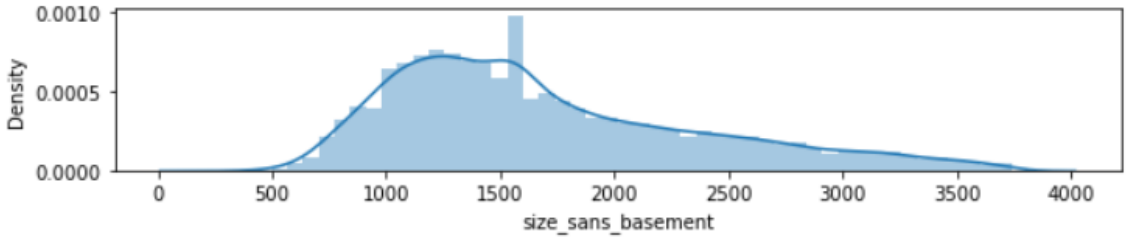

500
1000
1500
2000
2500
3000
3500
house_size_2015
20
40
60
80
100
120
house_age
500
1000
1500
2000
2500
3000
sqft_per_flr
0
2500
5000
7500
10000
12500
15000
17500
lot_size_2015
5000
10000
15000
20000
total_area
2500
5000
7500
10000
12500
15000
17500
20000
2015_area
−1000
−500
0
500
1000
diff_living
0
2500
5000
7500
10000
12500
15000
17500
outside_space

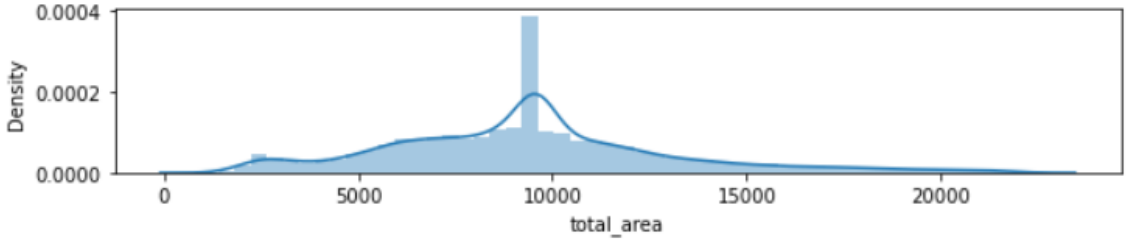
Checking for distribution plots

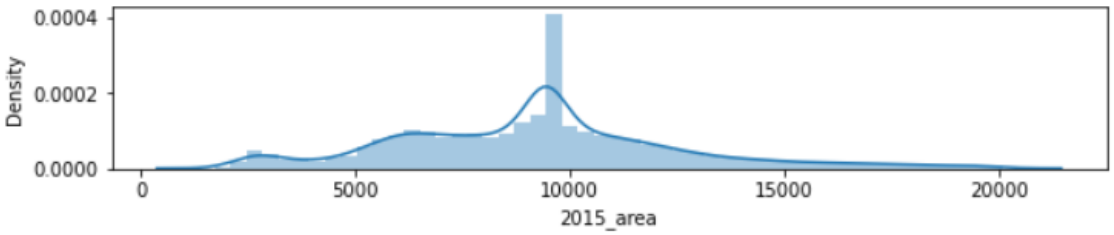


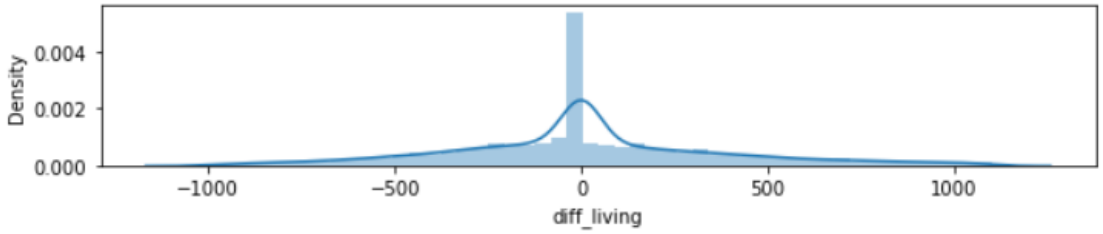

Density
0
2
4
0.5
1.0
1.5
2.0
2.5
3.0
3.5
bathrooms

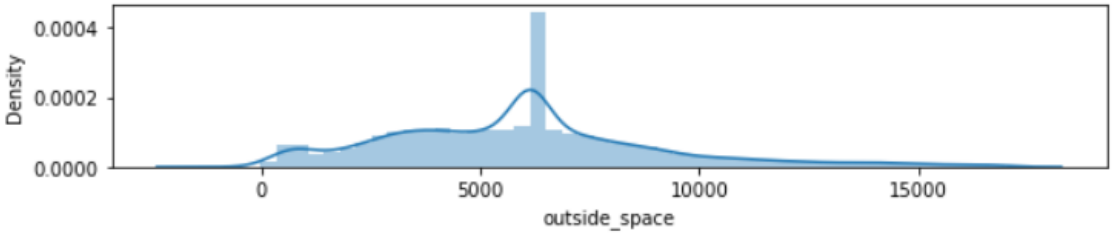

Density
0.0000
0.0002
0.0004
0
5000
10000
15000
20000
lot_in_sqft


Density
0.0000
0.0005
0.0010
0
500
1000
1500
2000
2500
3000
3500
4000
size_sans_basement


Density
0.0000
0.0002
0.0004
0
5000
10000
15000
20000
total_area

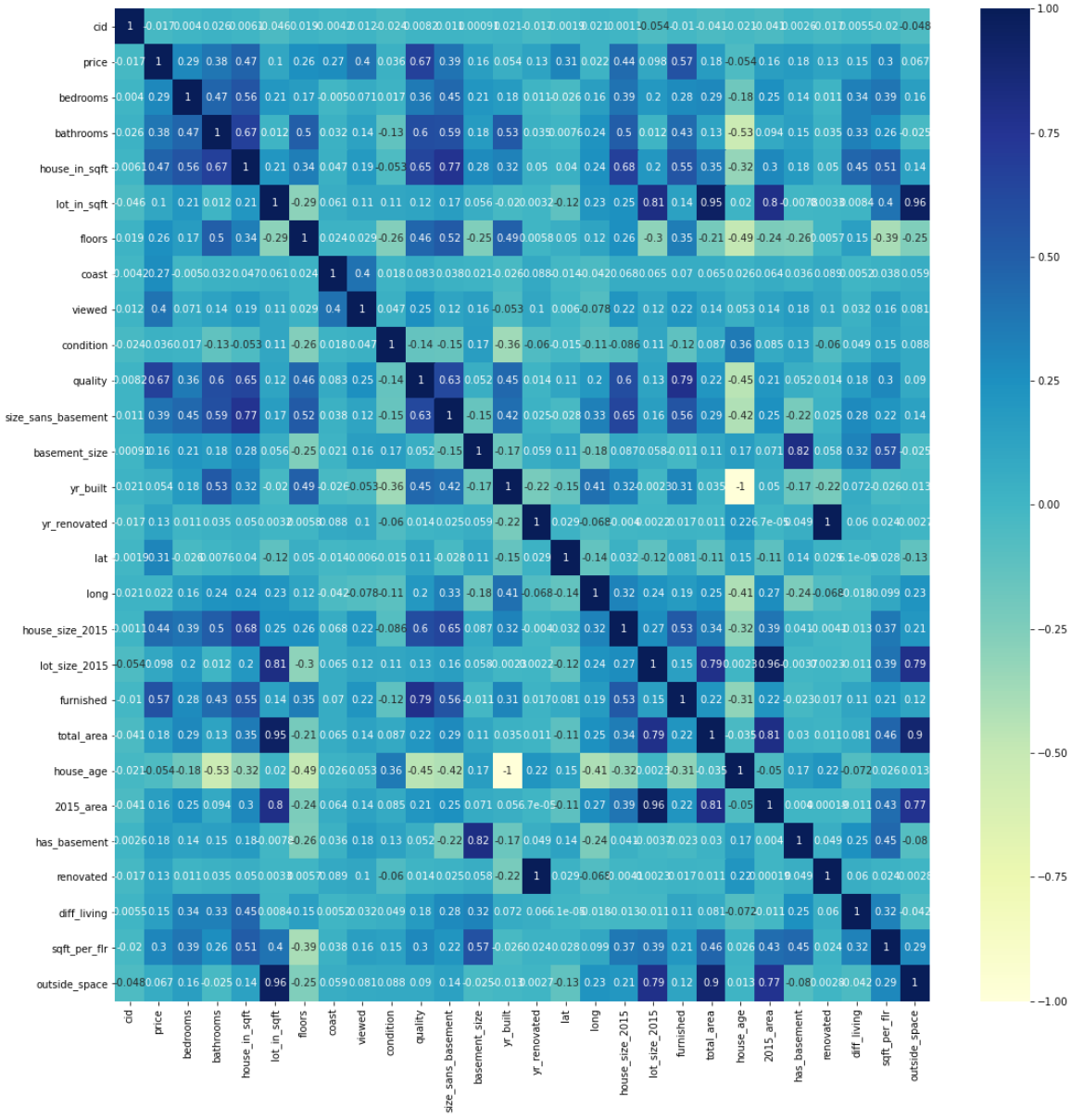

Density
0.0000
0.0002
0.0004
0
5000
10000
15000
20000
2015_area


Density
0.000
0.002
0.004
−1000
−500
0
500
1000
diff_living


Density
0.0000
0.0002
0.0004
0
5000
10000
15000
outside_space

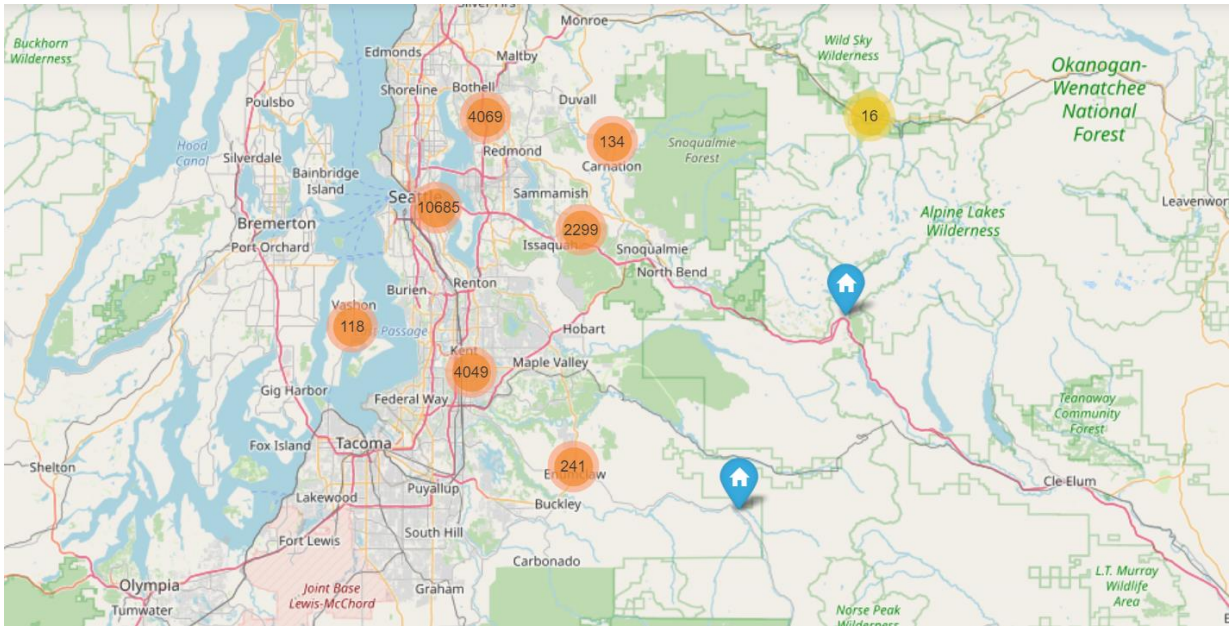
- Bedrooms and bathrooms have clusters.
- Variables house_in_sqft, lot_in_sqft and house_size_2015 are normally distributed but right skewed, slightly.
- Basement size and total area are not normally distributed and are heavily skewed.
- House_size_2015 and sqft per floor are almost normally distributed but right skewed, slightly.
- House_size without basement, difference to living area, 2015 area, lot size in 2015, and outside space are not normally distributed.

Correlation analysis

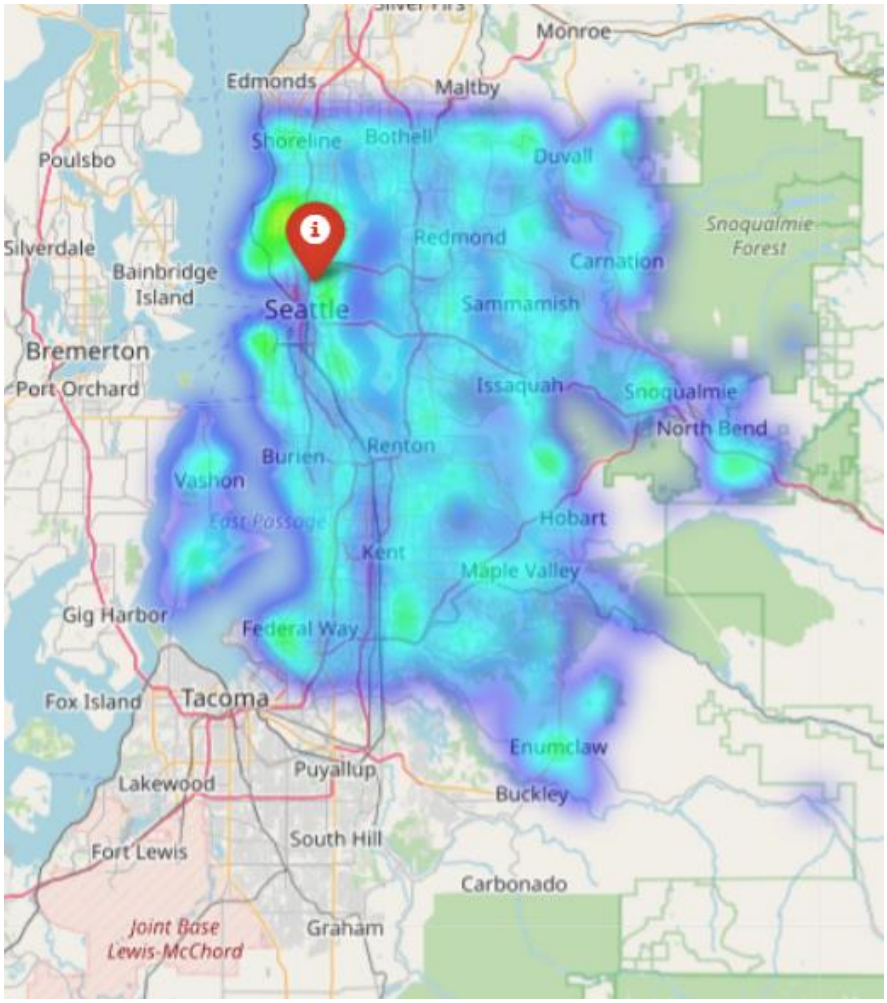


- Price' is moderately correlated with 'furninshed','house_in-sqft', 'size_sans_basement','quality','house_size_2015', and 'bathrooms'.
- Also, 'lot_in_sqft' and 'total_area' are completely correlated.
- Even 'size_sans_basement' and 'house_in-sqft' are highly correlated.

Maximum house density in Seattle



That's where prices are bound to be high, since there is more competition for less space.



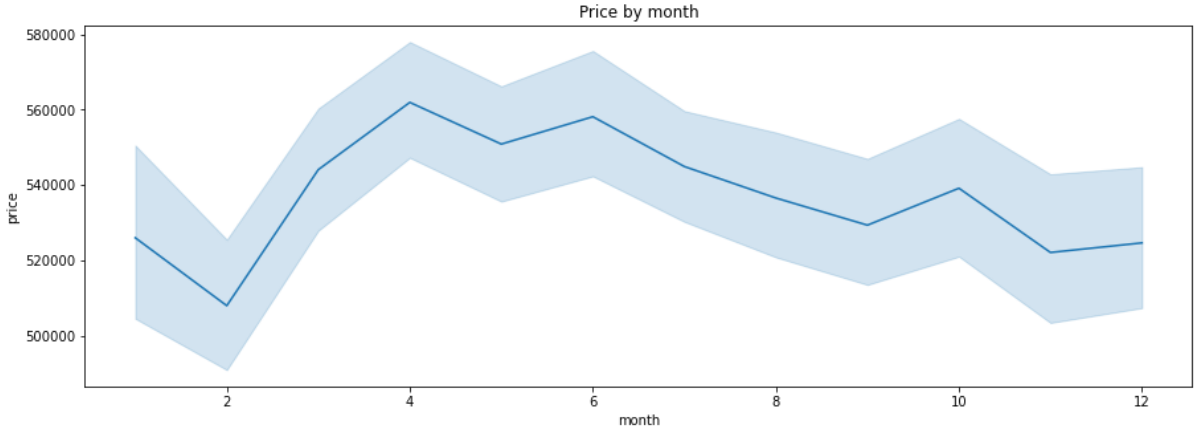
Heat map of King County, based on house price

Date

- Make datetime
- Extract year
- Extract month

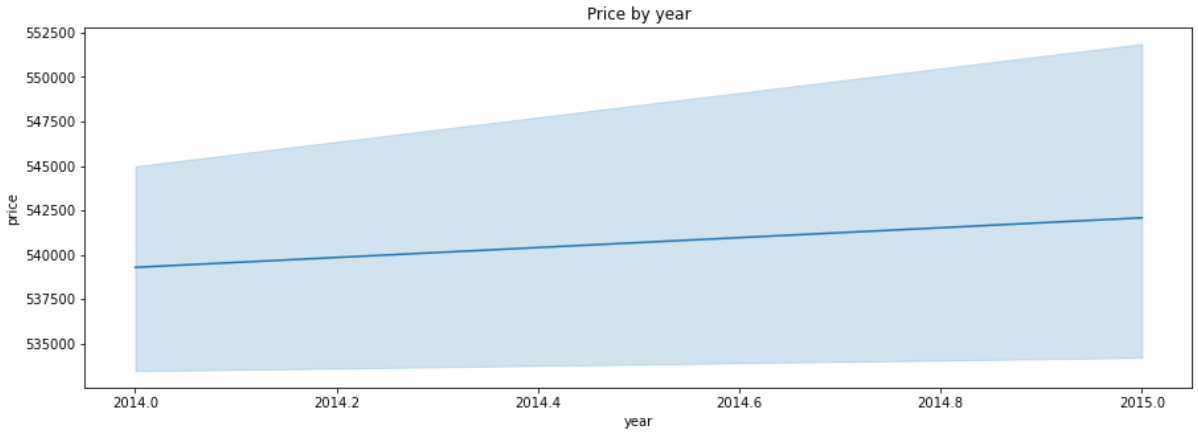
	sell_date	day_week	month	year
0	2015-04-27	Monday	4	2015
1	2015-03-17	Tuesday	3	2015
2	2014-08-20	Wednesday	8	2014
3	2014-10-10	Friday	10	2014
4	2015-02-18	Wednesday	2	2015

Month



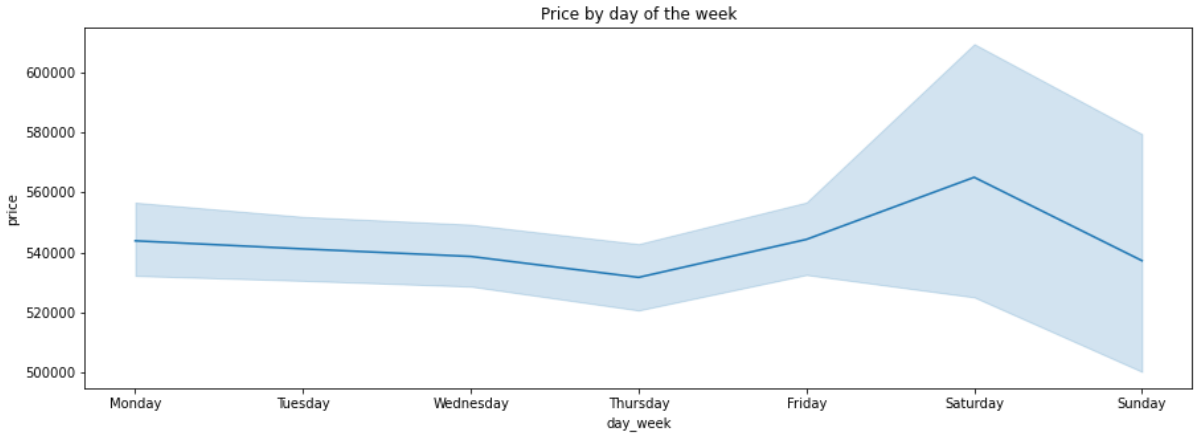
House price goes up in April and is least in February. April is the best time for the sellers and February for the buyers. The steepest fall is from December to February. Now trending upwards.

Year

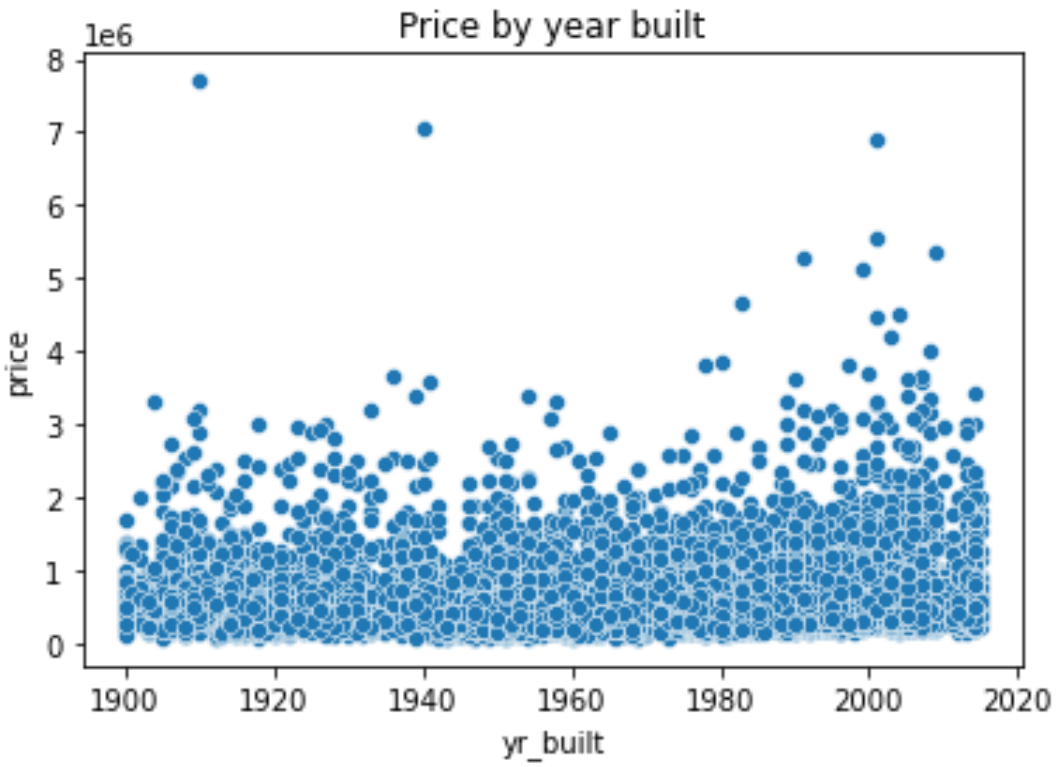


The house price has been rising steadily since 2014 but the rise is not so steep. Trending upwards.

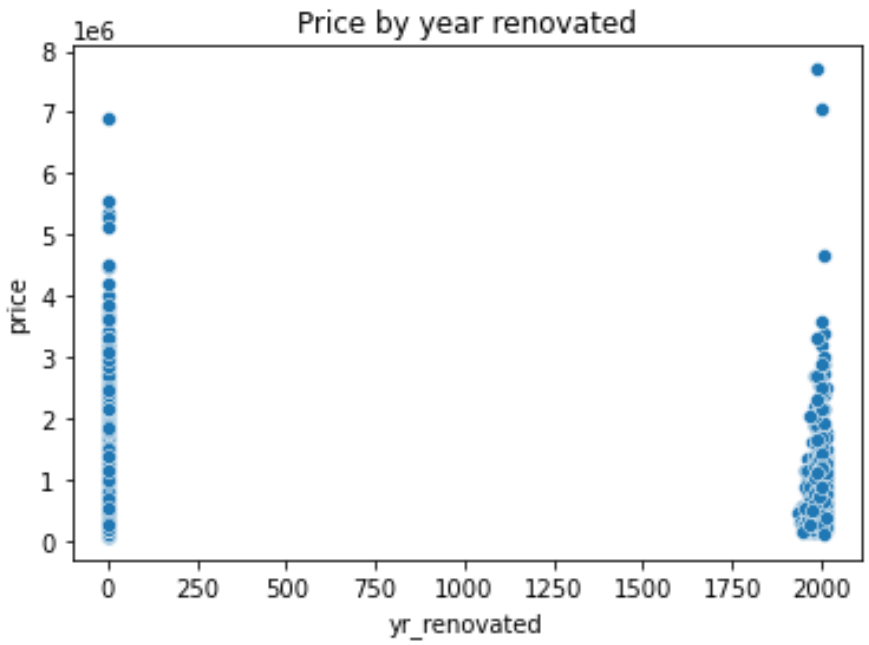
Day of the week



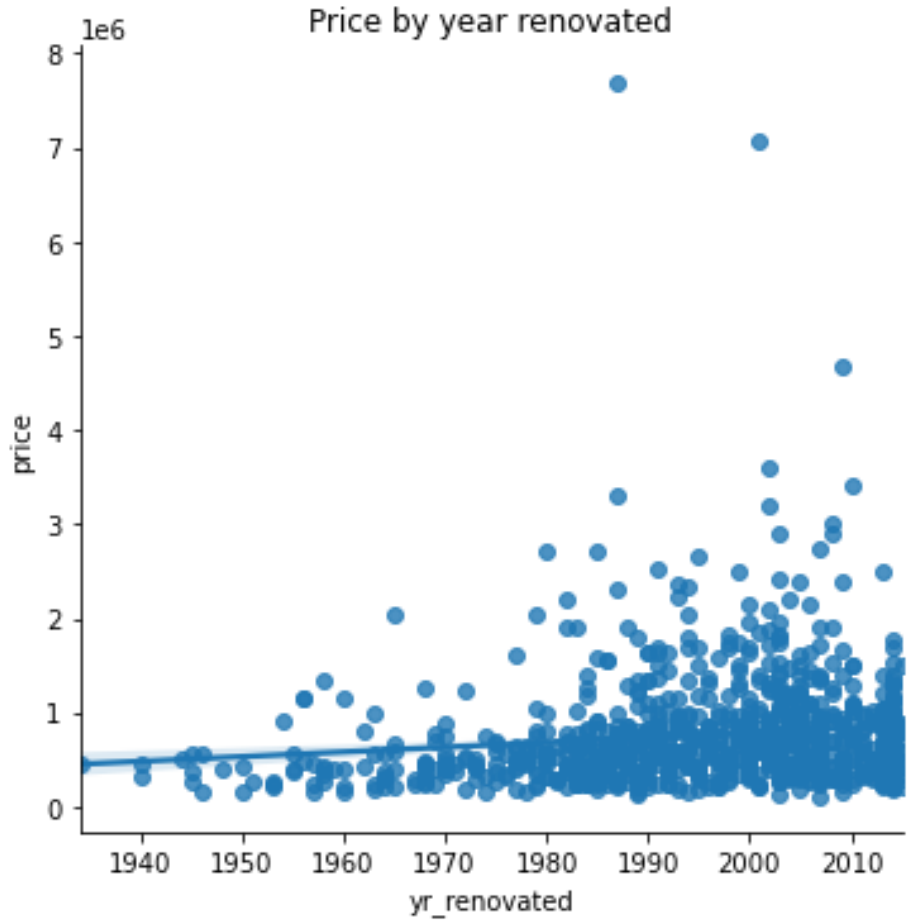
Sunday is the best time to buy a house, while Saturday is the worst. It's the other way round for the buyers.



Some of the property built after 2000 or between 1900 and 1920 is the most sought after and the costliest in the county.

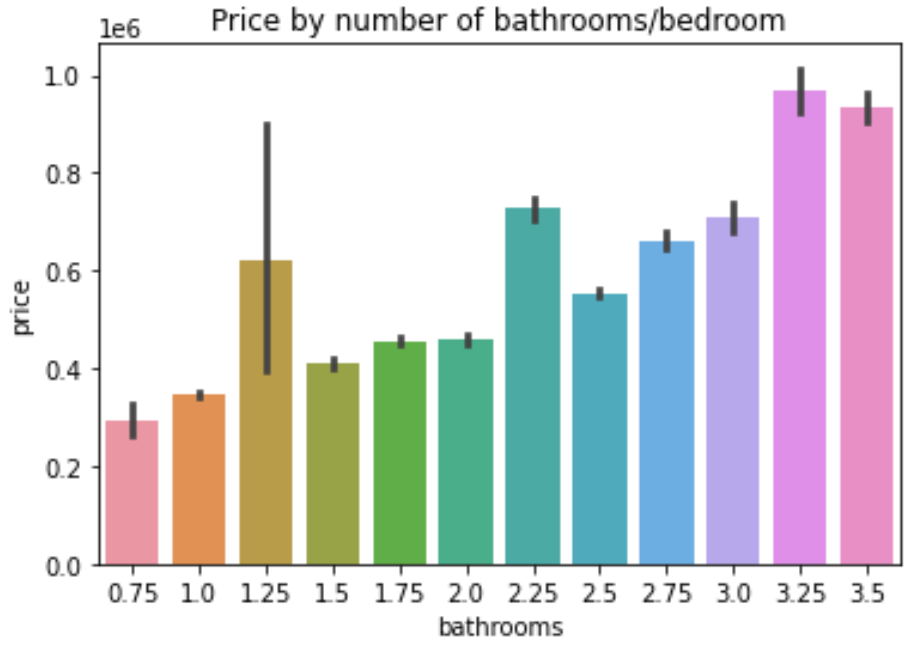


0 means it was never renovated, so let's check the variable without the zeroes.



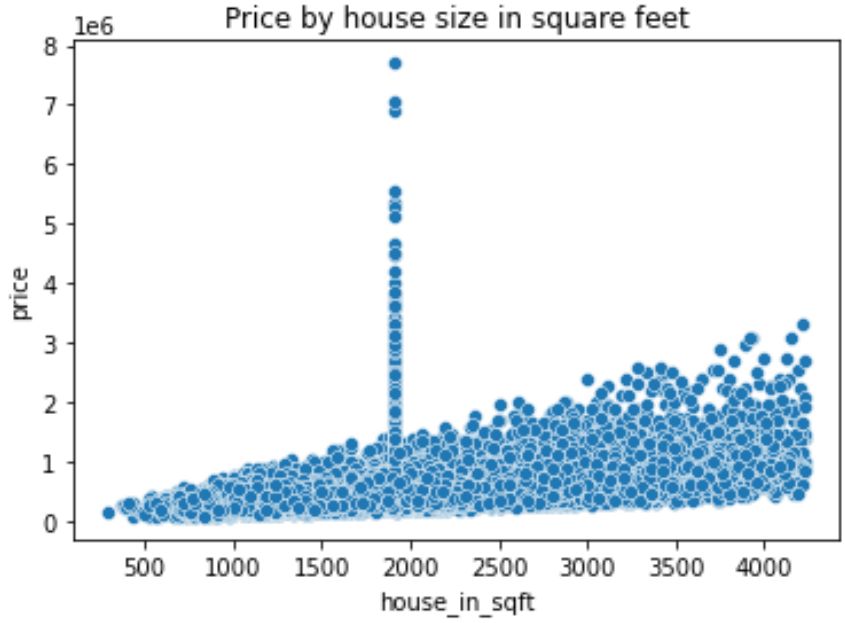
Checking relation between target variable price and other features

Price by number of bathrooms/bedroom



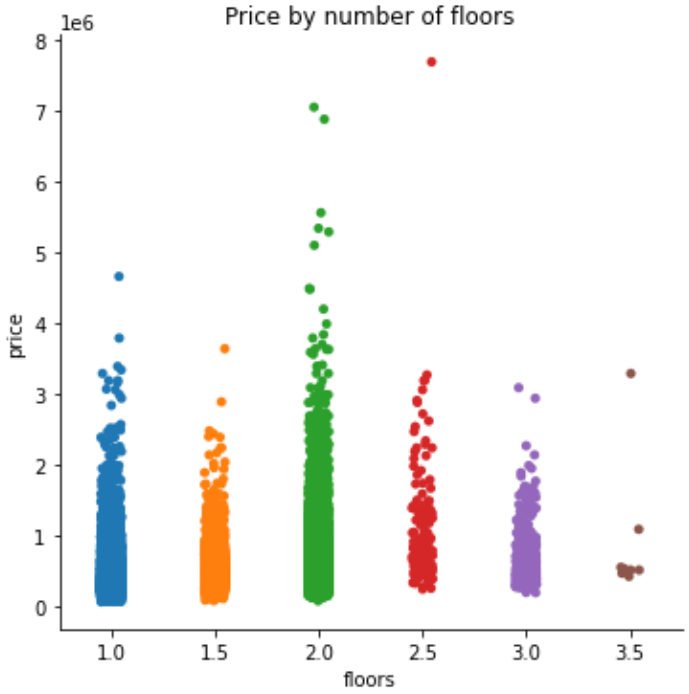
Price increases as the number of bathrooms goes up.

Price by house size in square feet



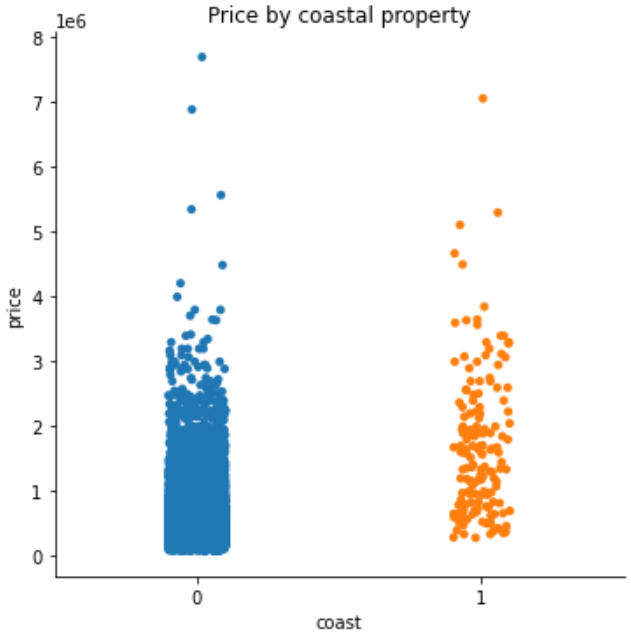
Price increases linearly with the living area but for a certain space in the range of 1,800 square feet, the price can go very high. Is it the area? We have to check.

Price by number of floors



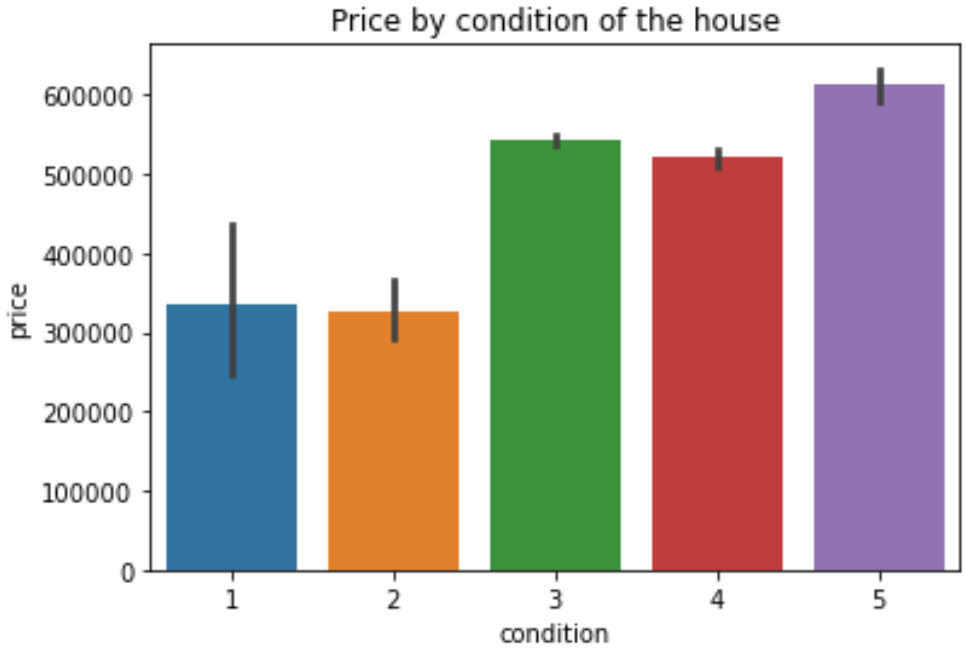
Two-floor houses have a higher price.

Price by coastal property



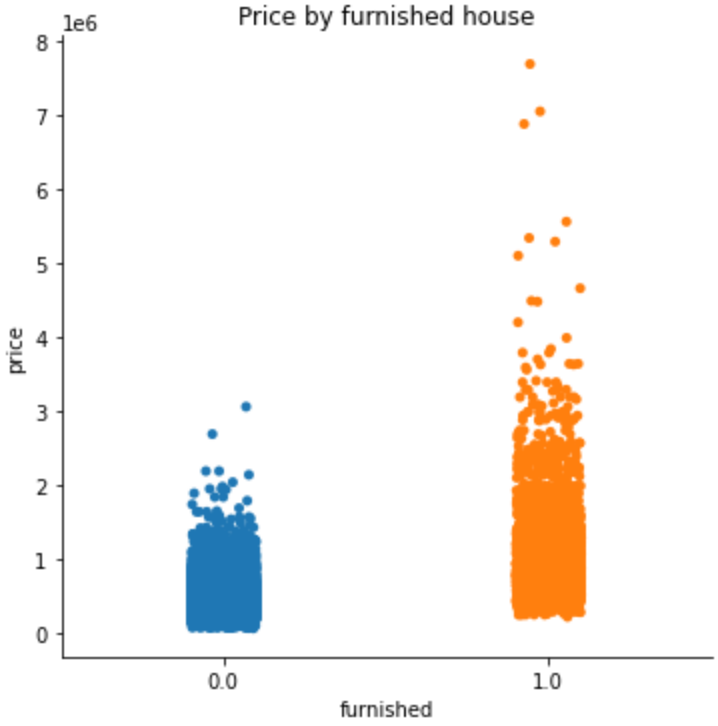
The number of houses with no coast or waterfront view is much higher than of houses with coast, but waterfront property is pricier, generally.

Price by condition of the house



Houses in better condition fetch a better price.

Price by furnished house



Price for a furnished property is way better.

In-demand property (Price by views)

The houses that are viewed more fetch a better price for being more in demand.

Binning the attributes

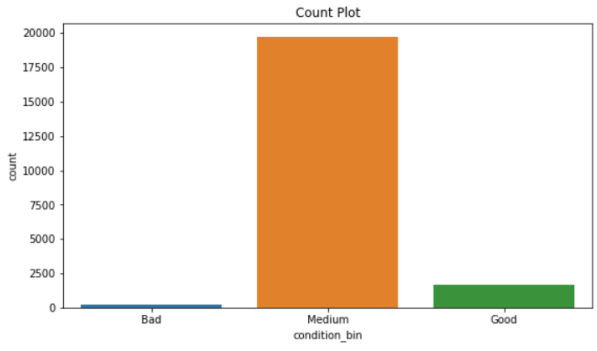
First check the unique values of the features to see how to group those

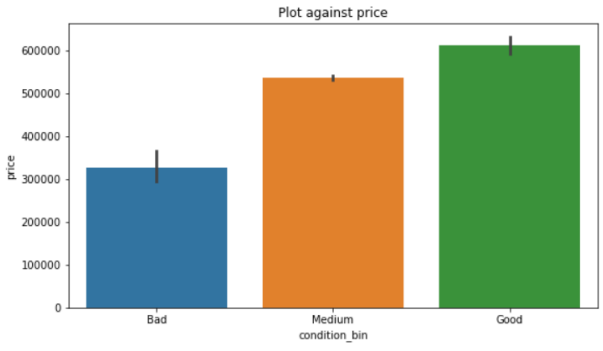
```
Condition :-
[1 2 3 4 5]

Quality :-
[ 1  3  4  5  6  7  8  9 10 11 12 13]

Bedrooms :-
[2. 3. 4. 5.]

Bathrooms :-
[0.75 1.   1.25 1.5  1.75 2.   2.25 2.5  2.75 3.   3.25 3.5 ]

Floors :-
[1.  1.5 2.  2.5 3.  3.5]

Viewed :-
[0. 1. 2. 3. 4.]
```

```
Year Built:-
[1900 1901 1902 1903 1904 1905 1906 1907 1908 1909 1910 1911 1912 1913
 1914 1915 1916 1917 1918 1919 1920 1921 1922 1923 1924 1925 1926 1927
 1928 1929 1930 1931 1932 1933 1934 1935 1936 1937 1938 1939 1940 1941
 1942 1943 1944 1945 1946 1947 1948 1949 1950 1951 1952 1953 1954 1955
 1956 1957 1958 1959 1960 1961 1962 1963 1964 1965 1966 1967 1968 1969
 1970 1971 1972 1973 1974 1975 1976 1977 1978 1979 1980 1981 1982 1983
 1984 1985 1986 1987 1988 1989 1990 1991 1992 1993 1994 1995 1996 1997
 1998 1999 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011
 2012 2013 2014 2015]

Year Renovated :-
[   0 1934 1940 1944 1945 1946 1948 1950 1951 1953 1954 1955 1956 1957
 1958 1959 1960 1962 1963 1964 1965 1967 1968 1969 1970 1971 1972 1973
 1974 1975 1976 1977 1978 1979 1980 1981 1982 1983 1984 1985 1986 1987
 1988 1989 1990 1991 1992 1993 1994 1995 1996 1997 1998 1999 2000 2001
 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012 2013 2014 2015]

House Age :-
[  7   8   9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24
  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42
  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60
  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78
  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96
  97  98  99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114
 115 116 117 118 119 120 121 122]
```

Binning the variable 'condition'

	condition	condition_bin
0	3	Medium
1	4	Medium
2	3	Medium
3	3	Medium
4	3	Medium

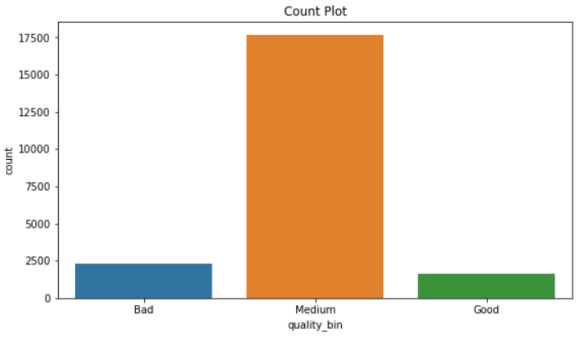


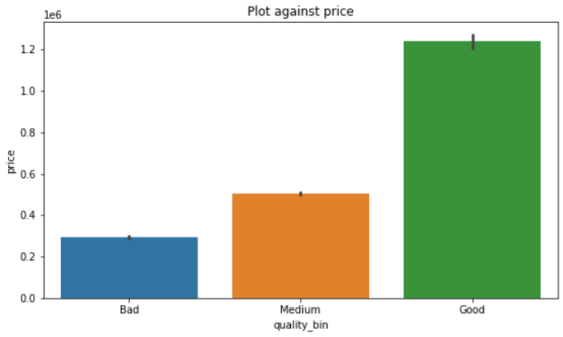



```
* Data was grouped on 'condition' like this: (1 & 2)= Good,
                                             (3 & 4)= Medium,
(                                                (5)= Good.

* Number of houses in 'medium' condition are in majority.
* Price of the house in 'good' condition is higher.
```

Binning the variable 'quality'

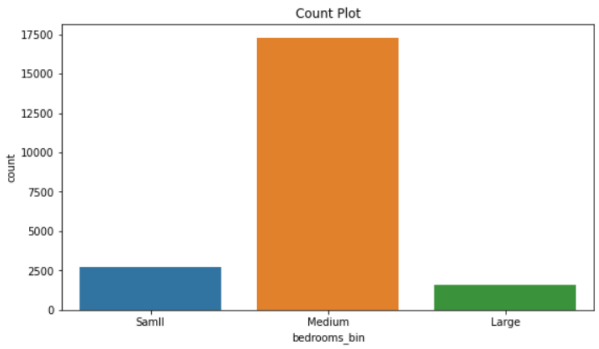
	quality	quality_bin
0	8	Medium
1	6	Bad
2	8	Medium
3	8	Medium
4	7	Medium

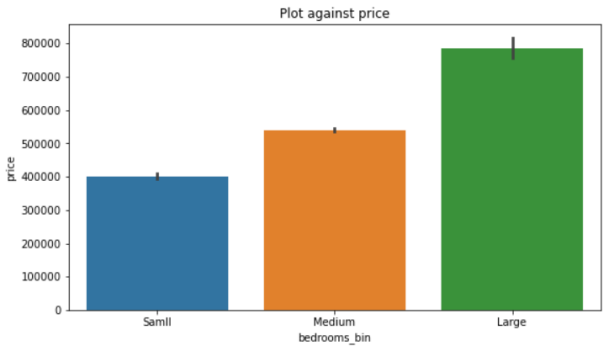




- The data was grouped on 'quality' in this manner: (1 to 6) = Good,
  (7 to 9) = Medium,
  (10 to 12) = Good.
- Medium-quality houses are in majority.
- Price of houses of good quality is higher.

Binning 'bedrooms'

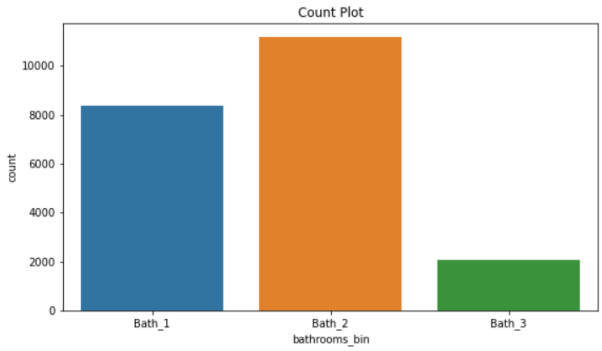


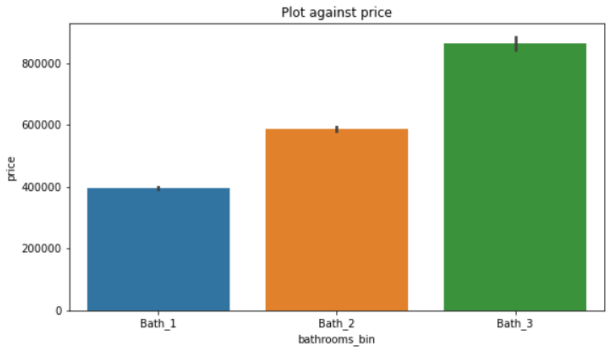


- The data was grouped on 'bedrooms' as: (1 to 2) = small,
  (3 & 4) = Medium,
  (5 =<) = Large.
- Houses with 'medium' number of bedrooms (3 to 4) are in majority.
- Houses with a large number of rooms are in the highest price bracket.

Binning bathrooms

	bathrooms	bathrooms_bin
0	1.750000	Bath_1
1	1.000000	Bath_1
2	2.750000	Bath_2
3	2.500000	Bath_2
4	1.000000	Bath_1

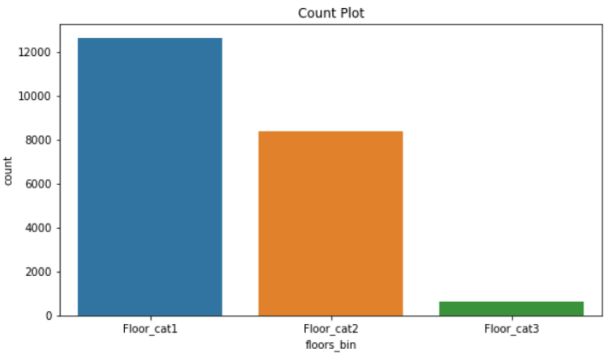


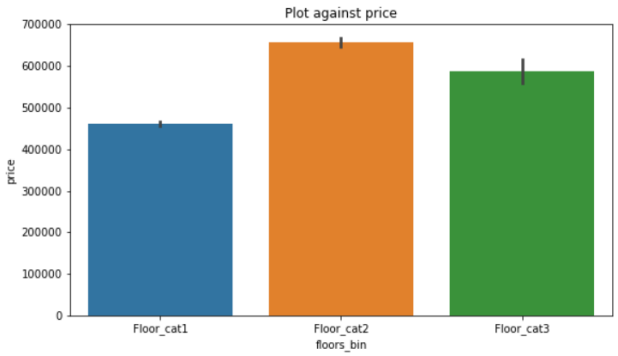


- Data was grouped on bathrooms as: (0 to 1) = Bath_1,
  (2 to 3) = Bath_2,
  (4 to 5) = Bath_grp3.
- Houses with 2 to 3 bathrooms in majority. Houses with 4 to 5 bathrooms are costliest.

Binning floors

	floors	floors_bin
0	1.000000	Floor_cat1
1	1.000000	Floor_cat1
2	2.000000	Floor_cat2
3	2.000000	Floor_cat2
4	1.000000	Floor_cat1

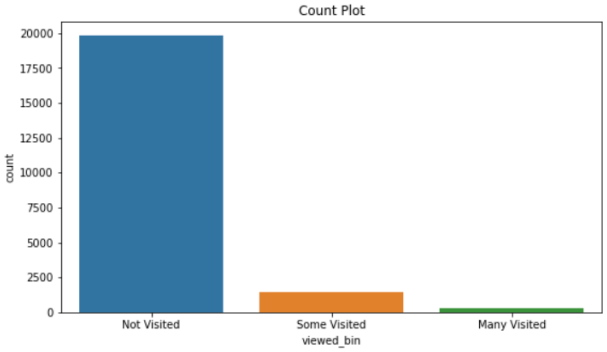


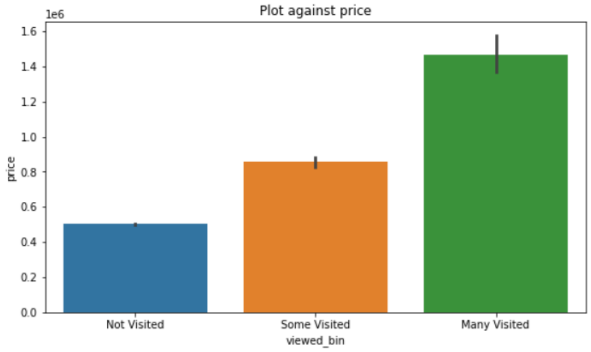


- The data was grouped on floors as: (0 to 1) = Floor_cat1,
  (2 to 3) = Floor_cat2,
  (3 to 4) = Floor_cat3.
- Houses with 0 to 1 floor are in majority. 0 floors are cabins, perhaps.
- Price of houses with 2 to 3 floors is highest.

Binning viewed

	viewed	viewed_bin
0	0.000000	Not Visited
1	0.000000	Not Visited
2	4.000000	Many Visited
3	0.000000	Not Visited
4	0.000000	Not Visited

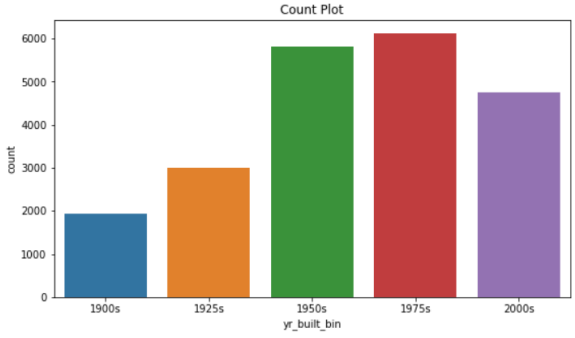




- Data was group on viewed as: (0 to 1) = Not Visited,
- (2 to 3) = Some Visited,
  (4 to 5) = Many visited.
- Unvisited houses are in majority and price of houses with many visitors is the highest.

Binning year_built

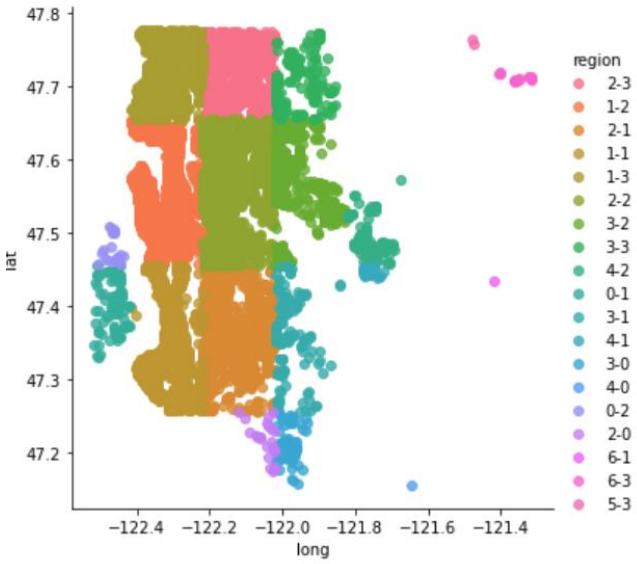
	yr_built	yr_built_bin
0	1966	1950s
1	1948	1925s
2	1966	1950s
3	2009	2000s
4	1924	1900s



- The variable yr_built' is divided into 5 groups.

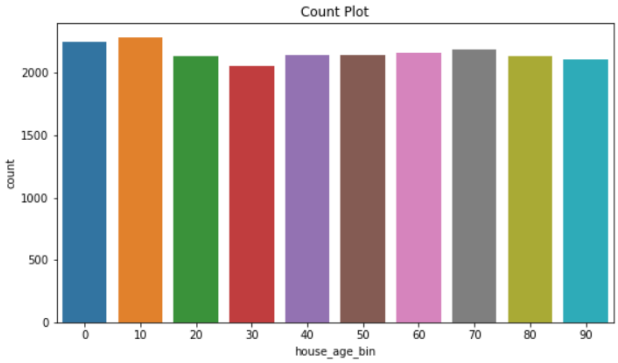
- Most of the houses in the county were built in the 1950s.
- The price of houses built in the 1900s is the highest because of antique value.

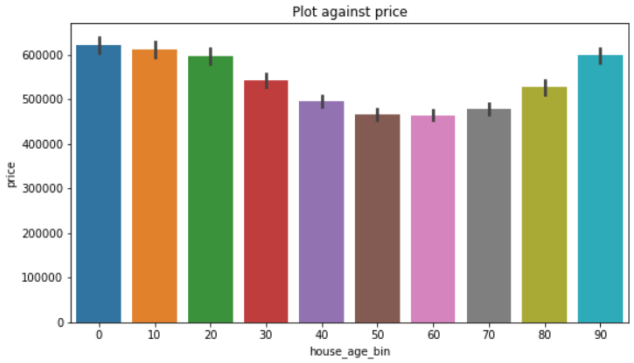
Lat, Long binning



Binning 'house_age'

	house_age	house_age_bin
0	56	50
1	74	70
2	56	50
3	13	0
4	98	90

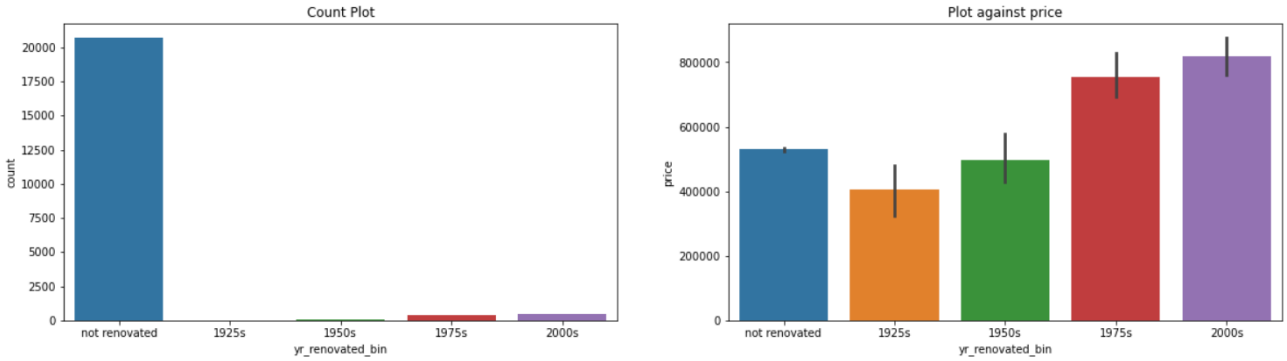




House_age is distributed equally across the 10 groups.

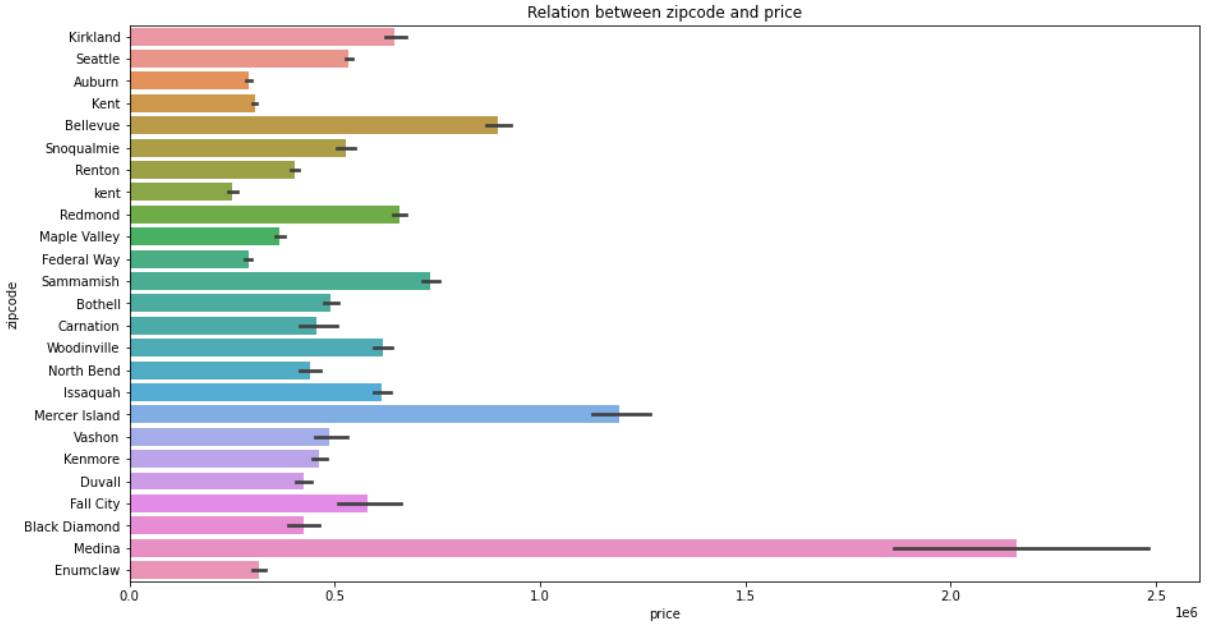
Binning 'year_renovated'

	yr_renovated	yr_renovated_bin
0	0	not renovated
1	0	not renovated
2	0	not renovated
3	0	not renovated
4	0	not renovated



- The 'yr_renovated' data was divided into 5 groups.
- Maximum number of houses in the county are not renovated.
- Price of renovated houses is better. Houses renovated in the 2000's fetch the best price, followed by ones renovated in the 1975s.

Binning Zipcodes (Gates' residence)



Medina seems to have the most expensive houses in the county, pricier than even Seattle. Google tells us that billionaire Bill Gates, along with a number of Microsoft executives, or other associates of Gates, have homes in Medina.

Unique zip codes

```
array(['Kirkland', 'Seattle', 'Auburn', 'Kent', 'Bellevue', 'Snoqualmie',
       'Renton', 'kent', 'Redmond', 'Maple Valley', 'Federal Way',
       'Sammamish', 'Bothell', 'Carnation', 'Woodinville', 'North Bend',
       'Issaquah', 'Mercer Island', 'Vashon', 'Kenmore', 'Duvall',
       'Fall City', 'Black Diamond', 'Medina', 'Enumclaw'], dtype=object)
```

Zip code description

```
count       21613
unique         25
top       Seattle
freq          8977
Name: zipcode, dtype: object
```

long_bin	lat_bin	region	Region_name	house_age_bin	yr_renovated_bin
2	3	2-3	Region-2-3	50	not renovated
1	2	1-2	Region-1-2	70	not renovated
1	2	1-2	Region-1-2	50	not renovated
2	1	2-1	Region-2-1	0	not renovated
1	2	1-2	Region-1-2	90	not renovated

```
Correlation between house_in_sqft and price:
0.4698618477335935

Correlation between house_size_2015 and price:
0.4363436591322249

Correlation between lot_in_sqft and price:
0.10370394435639411

Correlation between lot_size_2015 and price:
0.09798229078122614

Correlation between basement size and price:
0.16086228217888976
```

Looking at the above correlation values, we decide to retain house_in_sqft since its correlation with 'price' is higher and drop the other house size. And between lot_in_sqft and lot_size_2015; we can retain lot_in_sqft since its correlation with 'price' is little higher and drop the other one.

Running the encoder

```
bedrooms_bin
Samll     406233.367713
Medium    539497.326342
Large     772187.133852
dtype: float64
bathrooms_bin
Bath_1    398297.614373
Bath_2    587948.493201
Bath_3    847438.218983
dtype: float64
condition_bin
Bad       398237.723852
Medium    536155.143550
Good      608483.531705
dtype: float64
```

```
quality_bin
Bad        305198.109357
Medium     505929.081563
Good      1198493.559419
dtype: float64
floors_bin
Floor_cat1    460994.144128
Floor_cat2    655663.596728
Floor_cat3    581052.685069
dtype: float64
viewed_bin
Not Visited      502221.096250
Some Visited     834526.246577
Many Visited    1244933.973396
dtype: float64
```

```
yr_built_bin
1900s    595050.010725
1925s    514564.009337
1950s    472679.050462
1975s    539075.553678
2000s    616908.625876
dtype: float64
house_age_bin
0     616675.901818
10    607971.299740
20    593758.022755
30    541198.089545
40    496610.116050
50    469222.879572
60    467455.158887
70    480564.783164
80    527434.017843
90    595967.919220
dtype: float64
```

Columns before selection

```
Index(['cid', 'sell_date', 'price', 'bedrooms', 'bathrooms', 'house_in_sqft',
       'lot_in_sqft', 'floors', 'coast', 'viewed', 'condition', 'quality',
       'size_sans_basement', 'basement_size', 'yr_built', 'yr_renovated',
       'zipcode', 'lat', 'long', 'house_size_2015', 'lot_size_2015',
       'furnished', 'total_area', 'house_age', '2015_area', 'has_basement',
       'renovated', 'diff_living', 'sqft_per_flr', 'outside_space',
       'condition_bin', 'quality_bin', 'bedrooms_bin', 'bathrooms_bin',
       'floors_bin', 'viewed_bin', 'yr_built_bin', 'long_bin', 'lat_bin',
       'region', 'Region_name', 'house_age_bin', 'yr_renovated_bin',
       'bedrooms_bin_enc', 'bathrooms_bin_enc', 'condition_bin_enc',
       'quality_bin_enc', 'floors_bin_enc', 'viewed_bin_enc',
       'yr_built_bin_enc', 'house_age_bin_enc', 'yr_renovated_bin_enc',
       'zipcode_enc', 'Region_name_enc', 'furnished_enc', 'coast_enc'],
      dtype='object')
```

Running the encoder

```
yr_renovated_bin
not renovated   530494.759887
1925s           527856.735267
1950s           521594.697650
1975s           709494.866979
2000s           767963.748888
dtype: float64
```

```
zipcode
Auburn          316056.650078
Bellevue        874691.436549
Black Diamond   481924.074397
Bothell         507260.175862
Carnation       493369.365533
Duvall          464596.820274
Enumclaw        382916.745148
Fall City       558286.662317
Federal Way     317922.681319
Issaquah        606125.772964
Kenmore         482774.323445
Kent            325097.336909
Kirkland        636667.184660
Maple Valley    391992.898376
Medina         1080554.772529
Mercer Island  1023488.434239
```

```
North Bend      470870.170964
Redmond         647905.524448
Renton          411524.659917
Sammamish       711416.690977
Seattle         535125.845971
Snoqualmie      530973.631413
Vashon          511655.559080
Woodinville     603957.465638
kent            379689.981686
dtype: float64
```

```
Region_name
Region-0-1   515027.458141
Region-0-2   528154.105568
Region-1-1   317437.478806
Region-1-2   586292.650521
Region-1-3   549455.883688
Region-2-0   511904.079672
Region-2-1   331809.121868
Region-2-2   695417.642081
Region-2-3   618337.728882
Region-3-0   380693.461653
Region-3-1   471365.212931
Region-3-2   608552.335212
Region-3-3   532039.573488
Region-4-0   538596.196825
Region-4-1   490861.319196
```

```
Region-4-2   481554.766130
Region-5-3   532972.704699
Region-6-1   540031.840389
Region-6-3   501493.121749
dtype: float64
furnished
0.000000   437992.387123
1.000000   950892.867897
dtype: float64
coast
0    531754.768554
1   1236071.486128
dtype: float64
```

Dropped columns

```
dropped column: condition_bin
dropped column: quality_bin
dropped column: bedrooms_bin
dropped column: bathrooms_bin
dropped column: floors_bin
dropped column: viewed_bin
dropped column: yr_built_bin
dropped column: house_age_bin
dropped column: yr_renovated_bin
dropped column: Region_name
dropped column: region
dropped column: furnished
dropped column: coast
dropped column: lat_bin
dropped column: long_bin
```

```
dropped column: cid
dropped column: lat
dropped column: long
dropped column: sell_date
dropped column: bedrooms
dropped column: bathrooms
dropped column: floors
dropped column: condition
dropped column: zipcode
dropped column: quality
dropped column: yr_built
dropped column: house_size_2015
dropped column: lot_size_2015
```

```
dropped column: size_sans_basement
dropped column: sqft_per_flr
dropped column: outside_space
dropped column: renovated
```

```
dropped column: viewed
dropped column: 2015_area
```

Data shape:

```
(21613, 22)
```

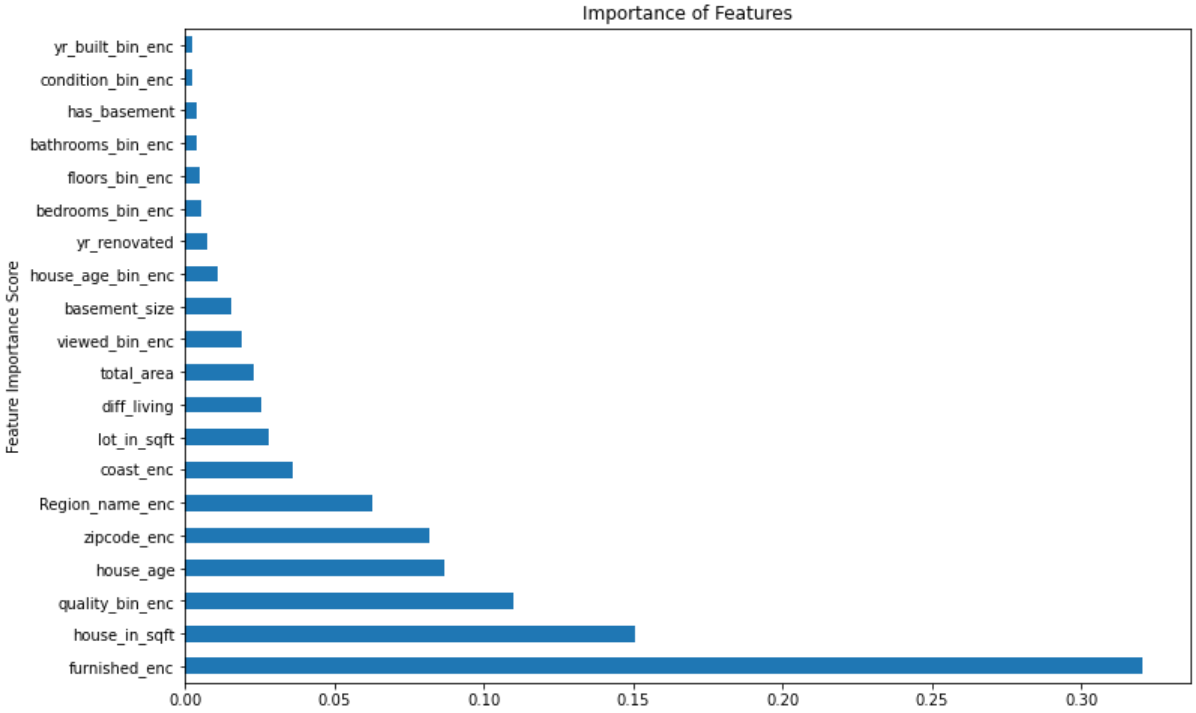
Missing values (corrected with mean imputing)

```
price                  0
house_in_sqft          0
lot_in_sqft            0
basement_size          0
yr_renovated           0
total_area             0
house_age              0
has_basement           0
diff_living            0
bedrooms_bin_enc       0
bathrooms_bin_enc      0
condition_bin_enc      0
quality_bin_enc       13
floors_bin_enc         0
viewed_bin_enc         0
yr_built_bin_enc       0
house_age_bin_enc      0
yr_renovated_bin_enc   0
zipcode_enc            0
Region_name_enc        0
furnished_enc          0
coast_enc              0
dtype: int64
```

```
price                  0
house_in_sqft          0
lot_in_sqft            0
basement_size          0
yr_renovated           0
total_area             0
house_age              0
has_basement           0
diff_living            0
bedrooms_bin_enc       0
bathrooms_bin_enc      0
condition_bin_enc      0
quality_bin_enc        0
floors_bin_enc         0
viewed_bin_enc         0
yr_built_bin_enc       0
house_age_bin_enc      0
yr_renovated_bin_enc   0
zipcode_enc            0
Region_name_enc        0
furnished_enc          0
coast_enc              0
dtype: int64
```

Finding feature importance

Extracting top 12 features that contribute to predicting our target variable.



Feature importance

```
furnished_enc          0.320494
house_in_sqft          0.150329
quality_bin_enc        0.109912
house_age              0.086790
zipcode_enc            0.081527
Region_name_enc        0.062355
coast_enc              0.036140
lot_in_sqft            0.027903
diff_living            0.025630
total_area             0.022724
viewed_bin_enc         0.018832
basement_size          0.015250
house_age_bin_enc      0.010764
yr_renovated           0.007548
bedrooms_bin_enc       0.005215
floors_bin_enc         0.004768
bathrooms_bin_enc      0.003606
has_basement           0.003560
condition_bin_enc      0.002458
yr_built_bin_enc       0.002359
yr_renovated_bin_enc   0.001837
dtype: float64
```

We drop the last 3 columns, since those variables have the least feature importance. We select the remaining variables for model building.

```
dropped column: yr_renovated_bin_enc
dropped column: yr_built_bin_enc
dropped column: condition_bin_enc
```

Furnished house is the most important feature, followed by square feet area of the living space, quality of the building, and age of the house. Zip code is also important.

```
Index(['price', 'house_in_sqft', 'lot_in_sqft', 'basement_size',
       'yr_renovated', 'total_area', 'house_age', 'has_basement',
       'diff_living', 'bedrooms_bin_enc', 'bathrooms_bin_enc',
       'quality_bin_enc', 'floors_bin_enc', 'viewed_bin_enc',
       'house_age_bin_enc', 'zipcode_enc', 'Region_name_enc', 'furnished_enc',
       'coast_enc'],
      dtype='object')
```

At this point, we have completed our feature engineering. Now we can start building our models.

Insights gained from EDA

There are 70 *zipcodes* in Kings County. This high cardinality might make it difficult to use encoding schemes. Using one-hot encoding increases our feature count by 68.

Extra bedrooms add extra values to the home. But after 6, the price starts to plateau or even decrease. Too many bedrooms to crowd square footage of the home will have less value.

Waterfront can really be a useful feature to predict the house price.

Bedrooms and bathrooms have clusters.

What type of problem is this

It is a case for regression predictive modelling. Regression is the problem of predicting a continuous quantity output. - Regression predictive modeling is the task of approximating a mapping function (f) from input variables (X) to a continuous output (target) variable (y). - In this case we are predicting price to be paid to for a house. - Target Variable (Y) = price.

Quality and condition can be subjected to ANOVA test to check how different factors impact the grading.

Feature floors' is positively correlated to 'price'. Higher number of floors can add value to houses that have smaller square footage. Higher number of floors doesn't add more value to houses that have big square footage. Higher number of floors with small square footage decreases the value of a home. 2.5 floors is ideal to have, more than that is unnecessary.

There are more houses without a basement than with a basement.
The presence of a basement increases the price of a house but not always: there are houses without a basement still make to Above Median price and there are houses with a basement stay behind in Below Median price.

Most of the King County houses are in condition 3 (which is average) and still they perform as well as those with condition 4 (good) and 5 (very good). This will be interesting to study.

PROBLEM SET 2

1). Model building and interpretation.
a. Build various models (You can choose to build models for either or all of descriptive, predictive or prescriptive purposes)
b. Test your predictive model against the test set using various appropriate performance metrics
c.Interpretation of the model(s)
2). Model Tuning and business implication
a.Ensemble modelling, wherever applicable
b. Any other model tuning measures(if applicable)
c. Interpretation of the most optimum model and its implication on the business

Modelling approach

In this section, we try out widely used regression techniques. Linear regression, K- nearest neighbour, support vector machines, and gradient bosting, which is and an ensemble method to boost results by stacking.

Multi-linear regression - Most common form of regression, multi linear regression is used to explain the relationship between one continuous dependant variable and two or more explanatory variables. Linear regression has several assumptions such as:- regression residuals must be normally distributed, residuals are homoscedastic and approximately rectangular-shaped, absence of multicollinearity is assumed in the model.

Boosting - Gradient boosting machines is a black-box boosting algorithm, means that it doesn't reveal the important feature that help it get to a prediction. It produces a prediction model in the form of an ensemble of weak prediction models, typically decision trees

K Nearest Neighbour - is a non-parametric lazy learning algorithm. KNN uses a similarity measure such as the Euclidian distance to find a heuristically optimal number of K nearest neighbours based on Root mean square error to label new observations. The output is the property value for the object. This value is the average of the values of its k nearest neighbours. KNN is sensitive to high dimensional data and outliers.

Support Vector Machine - Support vector regression uses the same method as classification. The objective is to find the maximal margin to explain the data whilst minimizing the error rate or soft margin (cost). The gaussian radial base function transforms the feature space to make a linear separation possible.

Random forest: - It is a supervised learning algorithm which is used for both classification as well as regression. But however, it is mainly used for classification problems. As we know that a forest is

made up of trees and more trees means more robust forest. Similarly, random forest algorithm creates decision trees on data samples and then gets the prediction from each of them and finally selects the best solution by means of voting. It is an ensemble method which is better than a single decision tree.

Lasso: - It is a technique that performs L1 regularisation. It modifies the loss function by adding the penalty (shrinkage quantity) equivalent to the summation of the absolute value of coefficients.

Ridge regression: - The Ridge regression is a technique which is specialized to analyse multiple regression data which is multicollinear in nature like our King County house price dataset.

The Elastic-Net: - It is a regularised regression method that linearly combines both penalties i.e. L1 and L2 of the Lasso and Ridge regression methods. It is useful when there are multiple correlated features. The difference between Lass and Elastic-Net lies in the fact that Lasso is likely to pick one of these features at random while elastic-net is likely to pick both at once.

XGBoost: - It stands for "Extreme Gradient Boosting". XGBoost is an optimized distributed gradient boosting library designed to be highly efficient, flexible and portable. It implements Machine Learning algorithms under the Gradient Boosting framework. It provides a parallel tree boosting to solve many data science problems in a fast and accurate way. Boosting is an ensemble learning technique to build a strong classifier from several weak classifiers in series. Boosting algorithms play a crucial role in dealing with bias-variance trade-off. Unlike bagging algorithms, which only controls for high variance in a model, boosting controls both the aspects (bias & variance) and is considered to be more effective. This algorithm is the champion of may data science competitions.

Gradient Boosting: - It is a popular boosting algorithm. In gradient boosting, each predictor corrects its predecessor's error. In contrast to Adaboost, the weights of the training instances are not tweaked, instead, each predictor is trained using the residual errors of predecessor as labels. There is a technique called the Gradient Boosted Trees whose base learner is CART (Classification and Regression Trees).

Decision tree: - It is a type of flowchart that shows a clear pathway to a decision. In terms of data analytics, it is a type of algorithm that includes conditional 'control' statements to classify data. A decision tree starts at a single point (or 'node') which then branches (or 'splits') in two or more directions. Each branch offers different possible outcomes, incorporating a variety of decisions and chance events until a final outcome is achieved. When shown visually, their appearance is tree-like.

Bagging: - Bagging Ensemble Algorithm Bootstrap Aggregation, or Bagging for short, is an ensemble machine learning algorithm. Specifically, it is an ensemble of decision tree models, although the bagging technique can also be used to combine the predictions of other types of models.

Metrics:- Our initial approach is to implement the linear regression and ensemble models such as gradient boosting along with using grid search CV. We will then evaluate the performance of these models using R-squared, adjusted r-squared, root mean square error, and mean absolute percentage error.

Approach:- We will try out this dataset first on non-scaled base models for all the above-listed techniques. Then, we will have another round of modelling on the non-scaled dataset, and then we will take the best models and try to improve those with grid search and hyperparameter tuning. We'll use scatter plots to study the train-test outputs and do normality check for each model. So, here we begin by splitting the data.

Checking if the train-test split happened right

```
SPLIT FOR INDEPENDENT VARIABLES
Shape of train data for independent variables: (15129, 18)
Shape of test data for independent variables: (6484, 18)
Percentage of data in train set for independent variables: 70 %
Percentage of data in test set for independent variables: 30 %

SPLIT FOR TARGET VARIABLE
Shape of train data for target variable price: (15129,)
Shape of test data for target variable price: (6484,)
Percentage of data in train set for target variable price: 70 %
Percentage of data in test set for target variable price: 30 %
```

Linear regression

Created a linear regressor and train the model using train sets

```
LinearRegression()
```

Linear Regression: Model Prediction on train data

```
array([407267.26406208, 451404.83186217, 481887.81219973, ...,
       372977.23729567, 367351.8061875 , 633386.51297598])
```

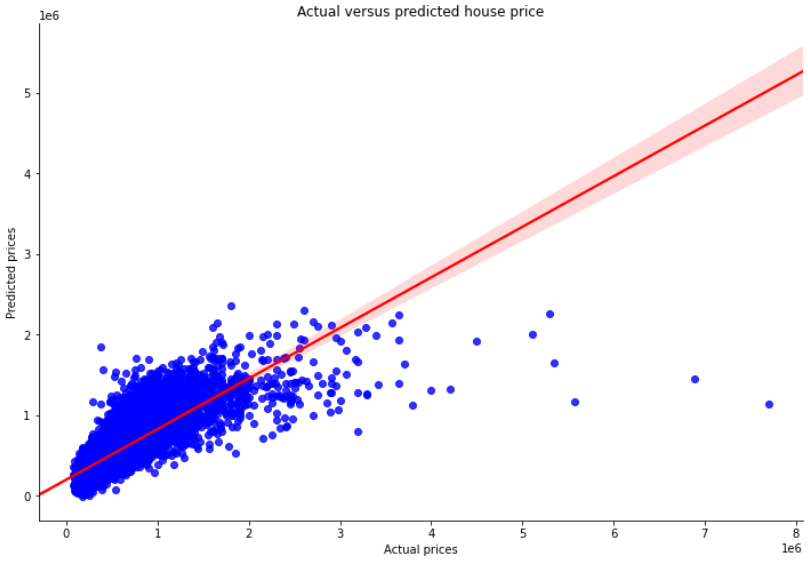
Actual target train data

```
17639    310000
11482    297000
14834    340000
17221    269000
7112     565000
          ...
5520     825000
3046     838000
20463    415250
18638    349950
2915     430000
Name: price, Length: 15129, dtype: int64
```

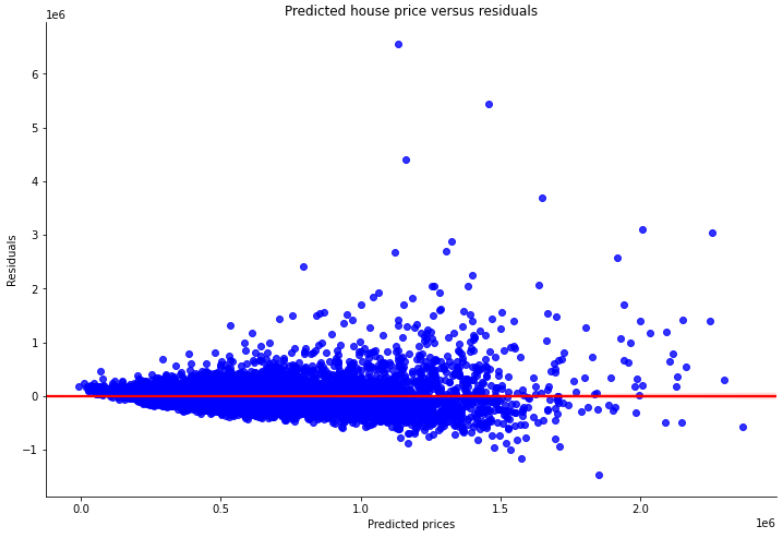
Linear Regression: Model evaluation (train data)

```
r^2: 0.6273528776391732
Adjusted r^2: 0.6269089565139254
Root mean squared error or RMSE: 220610.4038793652
Mean absolute percentage error or MAPE: 0.24434118200216937
```

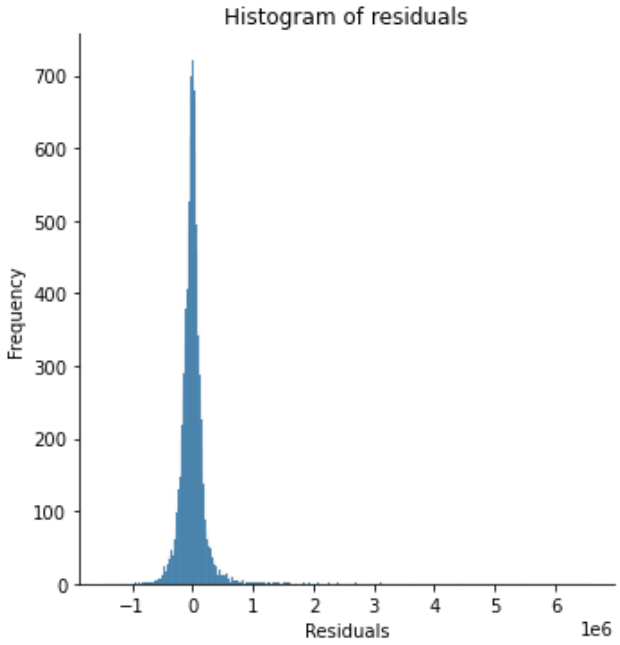
Linear regression:
Visualising the difference between actual and predicted prices



Linear regression: Inspecting residuals



Linear regression: Checking normality of errors¶



Linear regression: Model evaluation for test data

```
r^2: 0.6398587545623272
Adjusted r^2: 0.6388560411179531
Root mean squared error or RMSE: 228587.232399803
Mean absolute percentage error or MAPE: 0.24803115635728687
```

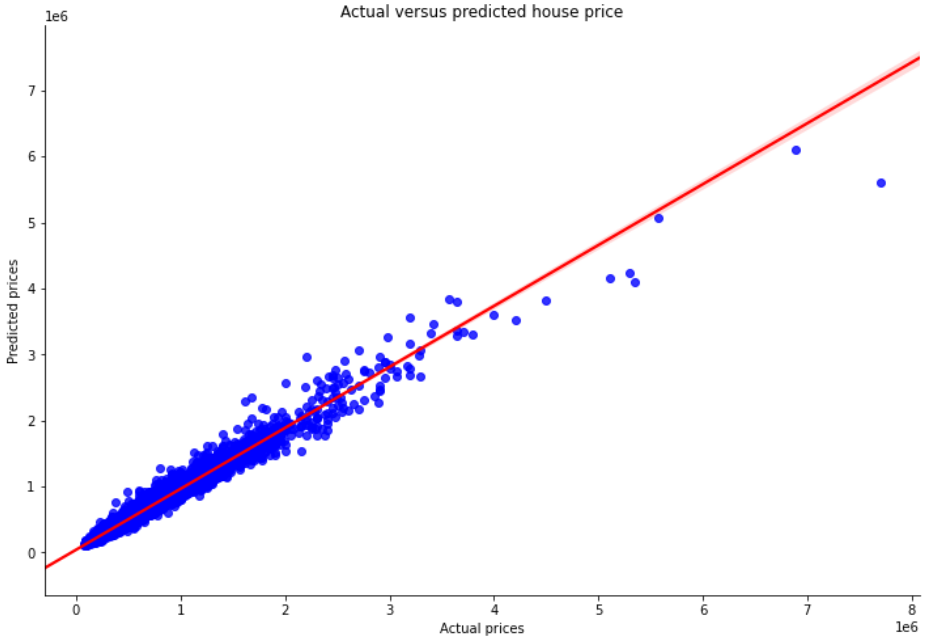
Random Forest: Create and train the model

```
RandomForestRegressor()
```

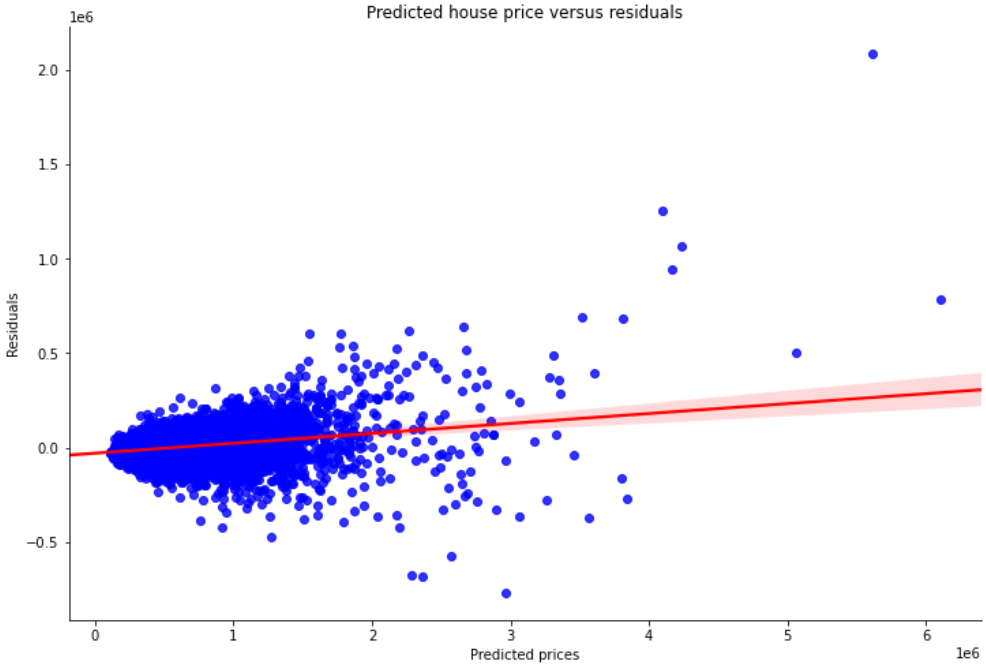
Random Forest: Model evaluation (train)

```
r^2: 0.9686838971220311
Adjusted r^2: 0.9686465913740626
Root mean squared error or RMSE: 63953.00552353083
Mean absolute percentage error or MAPE: 0.06619318896898202
```

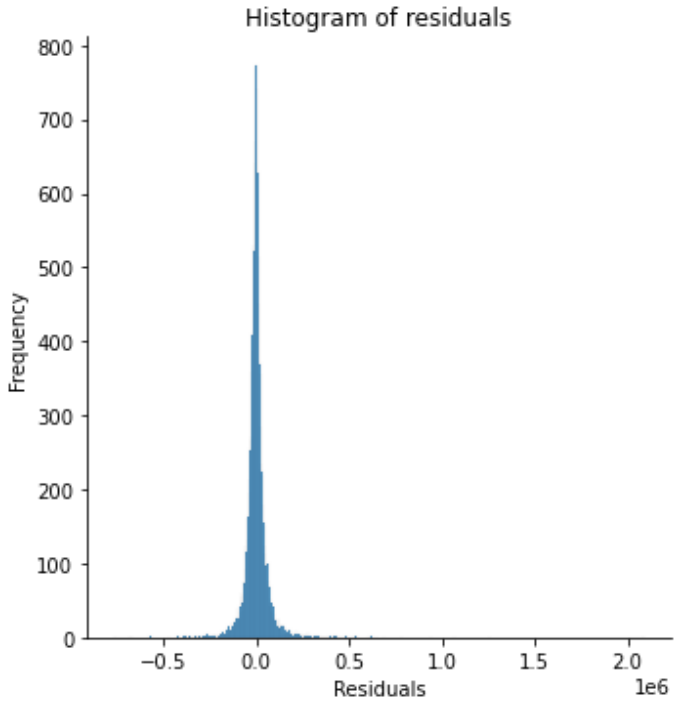
Random Forest:
Visualising the difference between actual and predicted house prices



Random Forest: Checking residuals



Random forest: Checking normality of errors



Random Forest: Model evaluation (test)

```
r^2: 0.7978383329355616
Adjusted r^2: 0.7972754698254054
Root mean squared error or RMSE: 171263.5506193193
Mean absolute percentage error or MAPE: 0.17235625865132734
```

XGBoost regressor

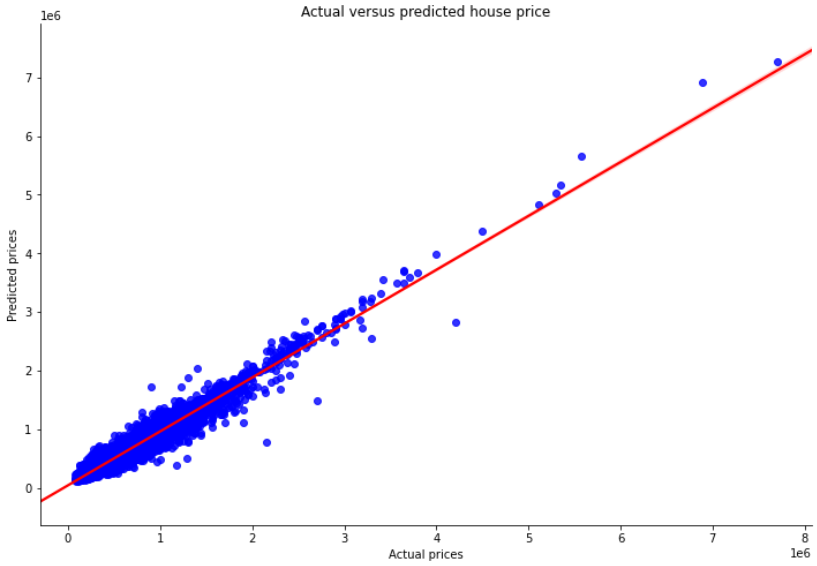
Create and train

```
XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
             colsample_bynode=1, colsample_bytree=1, gamma=0, gpu_id=-1,
             importance_type='gain', interaction_constraints='',
             learning_rate=0.300000012, max_delta_step=0, max_depth=6,
             min_child_weight=1, missing=nan, monotone_constraints='()',
             n_estimators=100, n_jobs=8, num_parallel_tree=1, random_state=0,
             reg_alpha=0, reg_lambda=1, scale_pos_weight=1, subsample=1,
             tree_method='exact', validate_parameters=1, verbosity=None)
```

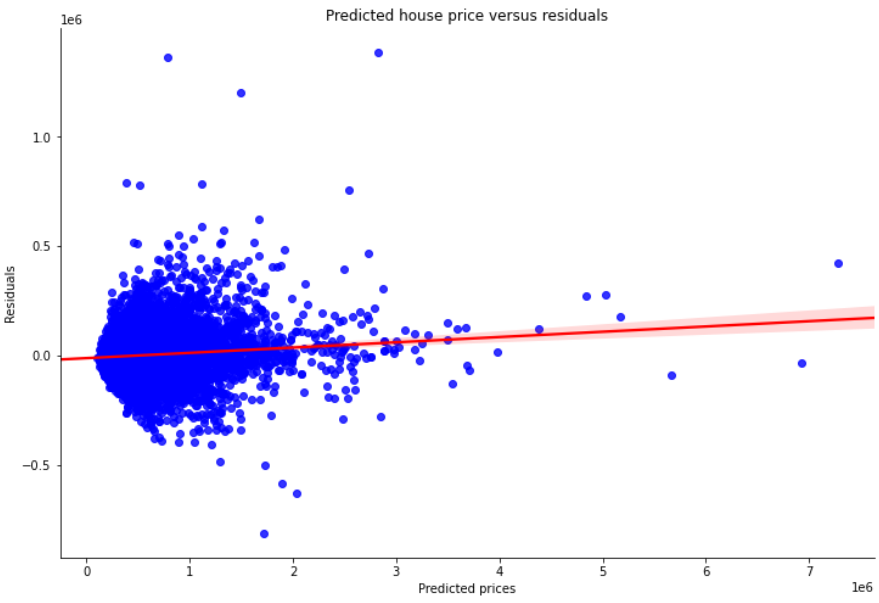
XGBoost Model evaluation (train)

```
r^2: 0.9404692782499442
Adjusted r^2: 0.9403312291370275
Root mean squared error or RMSE: 88175.44857718151
Mean absolute percentage error or MAPE: 0.1293323960730077
```

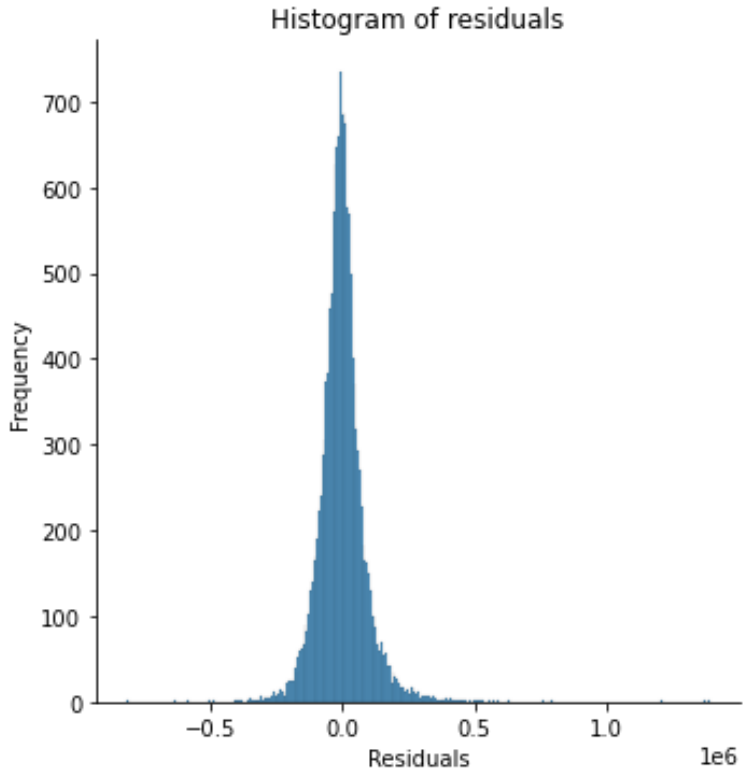
XG Boost:
Visualising the difference between actual and predicted prices



XGBoost: Checking residuals



XGBoost: Checking normality of errors



XGBoost: Model evaluation (test)

```
r^2: 0.8081558686340758
Adjusted r^2: 0.8071145310723812
Root mean squared error or RMSE: 166836.01106931968
Mean absolute percentage error or MAPE: 0.17678435645333032
```

Decision tree: Import decision tree regressor

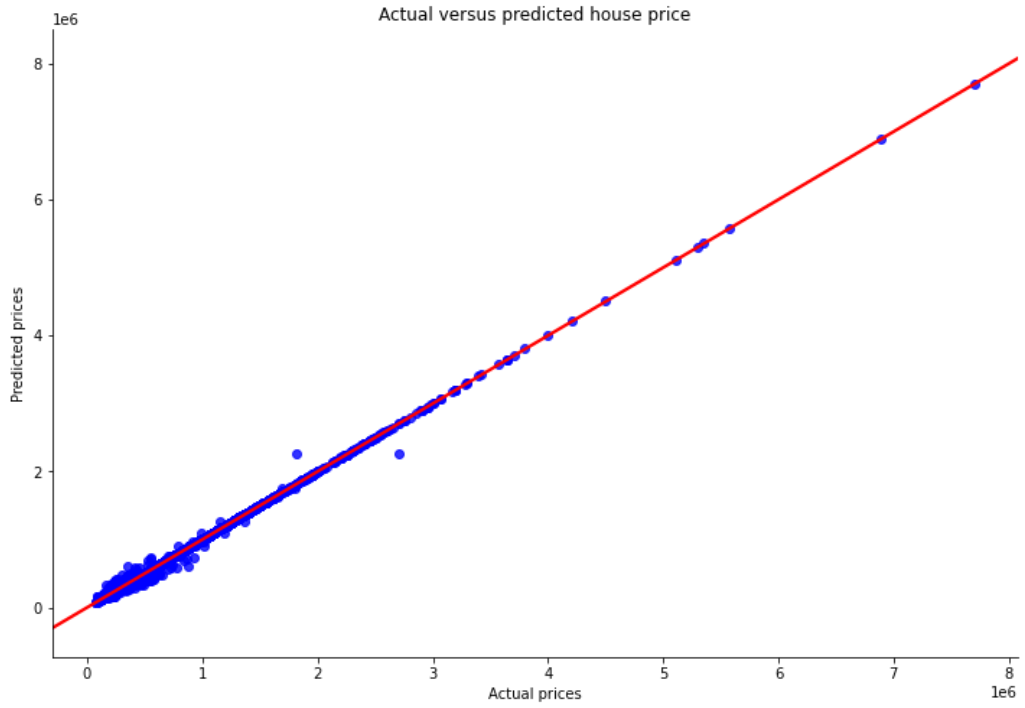
Create and train

```
DecisionTreeRegressor()
```

Decision tree: Model evaluation (train)¶

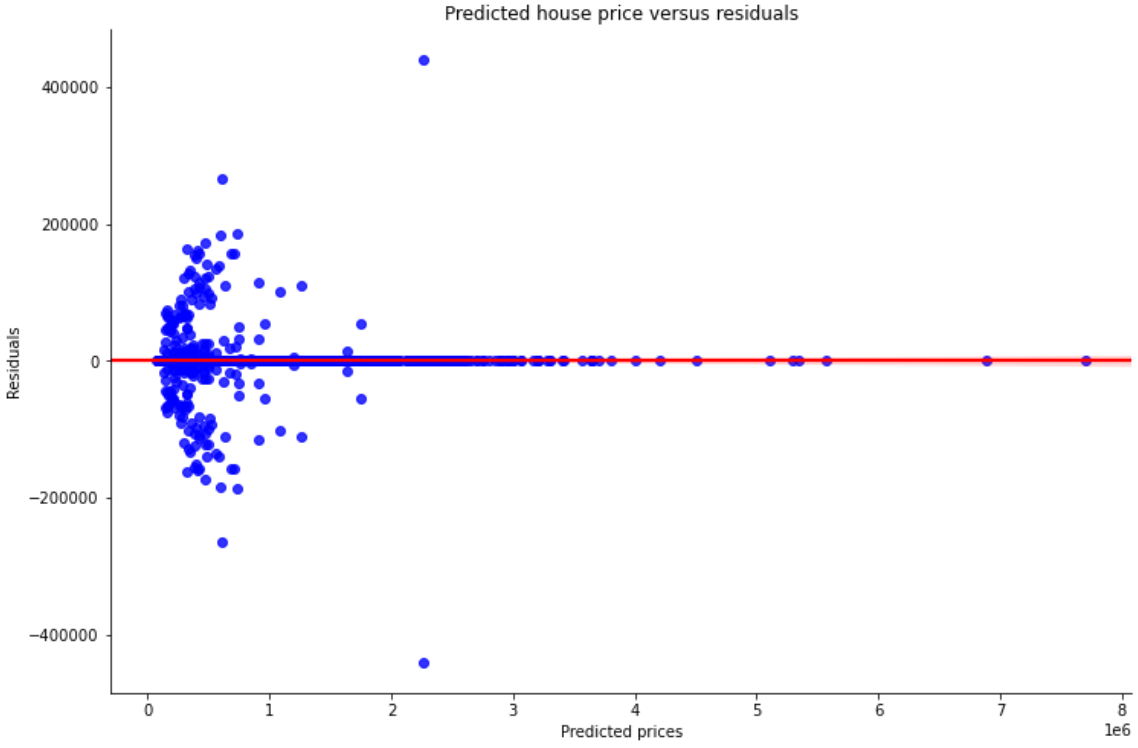
```
r^2: 0.999014565353217
Adjusted r^2: 0.9990133914403353
Root mean squared error or RMSE: 11344.647799122395
Mean absolute percentage error or MAPE: 0.002850832910236522
```

Decision tree:
Visualising the difference between actual and predicted house prices

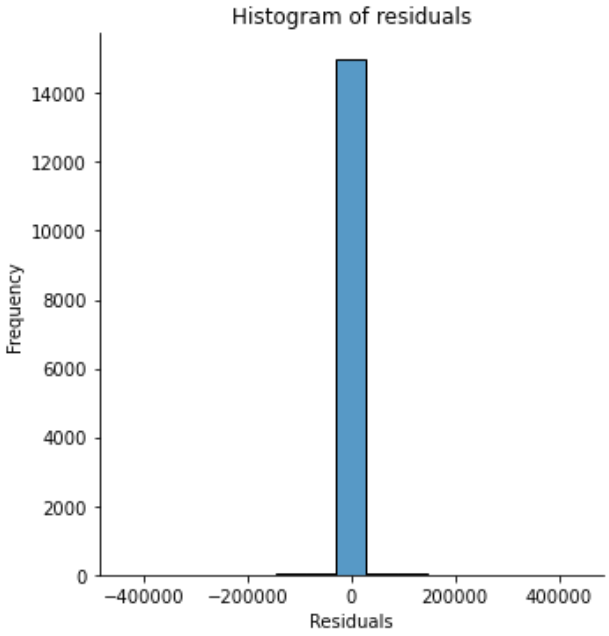


Decision tree: Checking residuals

¶



Decision tree: Checking normality of errors



Decision tree: Model evaluation (test)

```
r^2: 0.6535531478100132
Adjusted r^2: 0.6525885626067001
Root mean squared error or RMSE: 224199.09142904685
Mean absolute percentage error or MAPE: 0.23911582387336983
```

Bagging

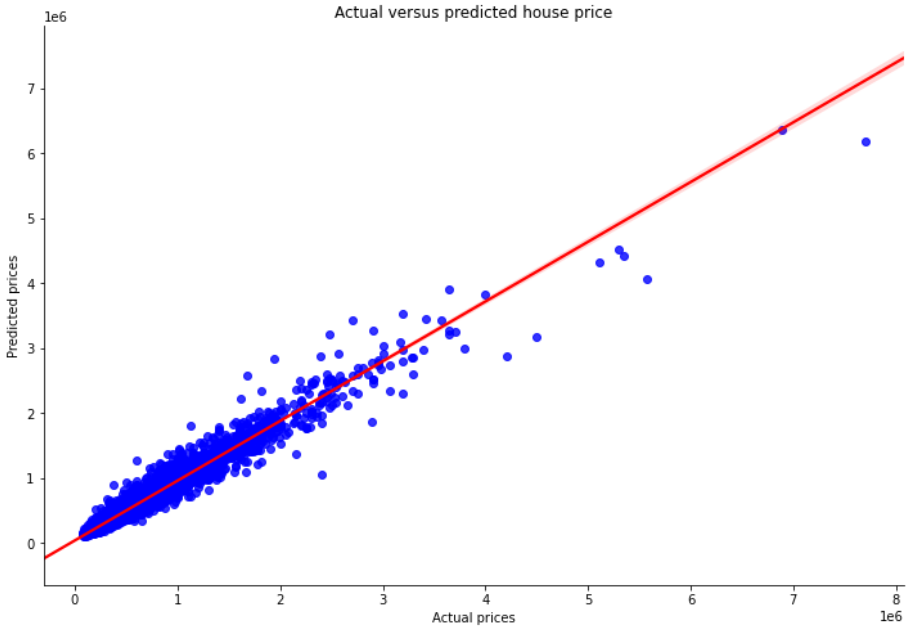
Create and train

```
BaggingRegressor()
```

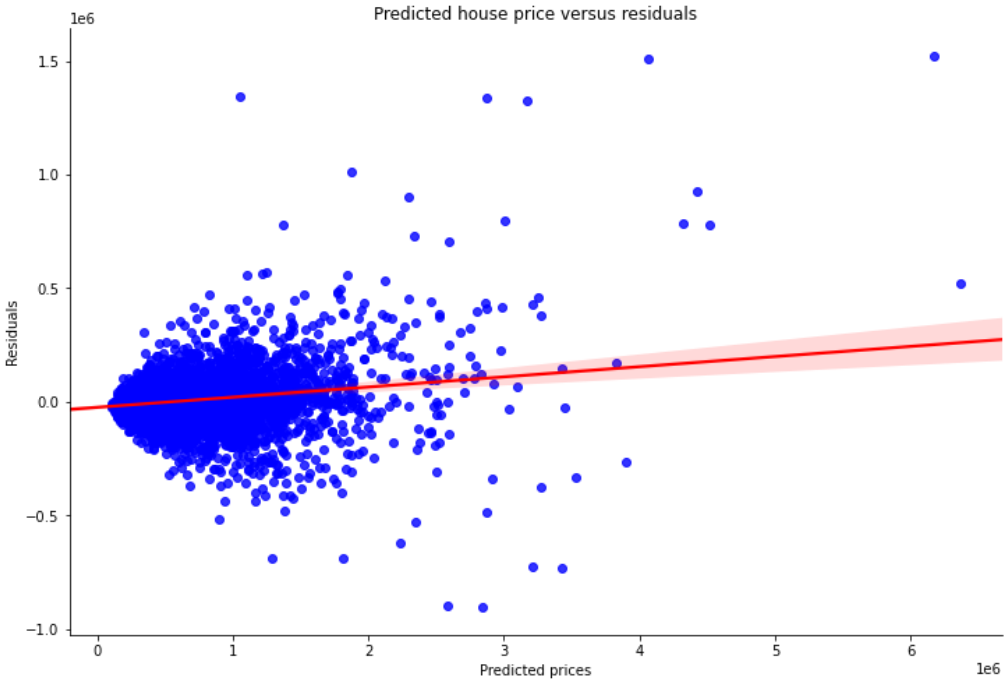
Bagging: Model evaluation (train)

```
r^2: 0.9583529889308012
Adjusted r^2: 0.9583033763431609
Root mean squared error or RMSE: 73751.18842600402
Mean absolute percentage error or MAPE: 0.07329438743935351
```

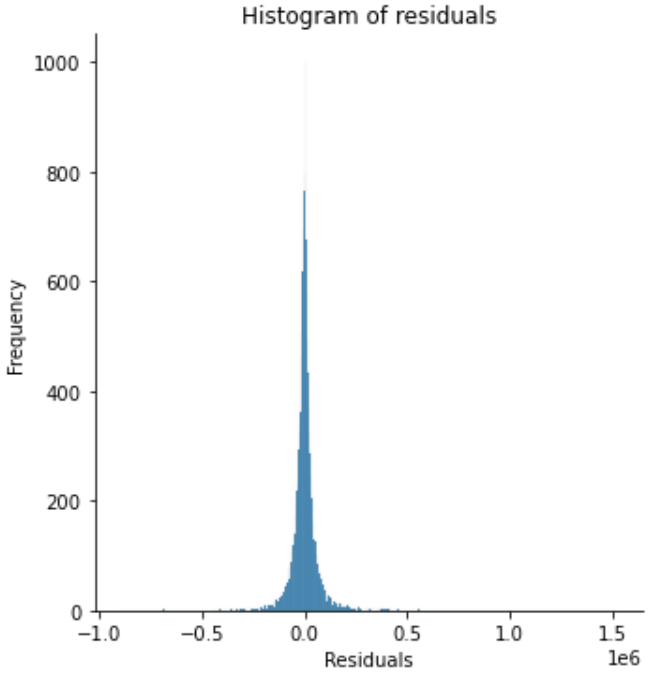
Bagging:
Visualising the difference between actual and predicted house prices



Bagging: Checking residuals



Bagging: Checking for normality of errors



Bagging: Predicting test data with the model

```
r^2: 0.7867552455411096
Adjusted r^2: 0.7861615246470246
Root mean squared error or RMSE: 175895.4951375854
Mean absolute percentage error or MAPE: 0.18107279235250243
```

Gradient Boost regressor

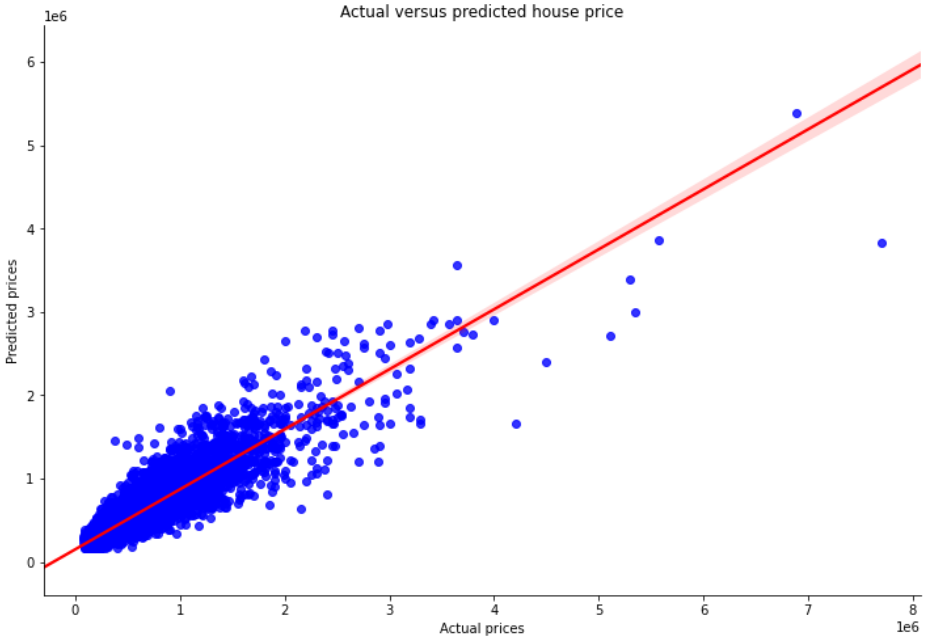
Create and train

```
GradientBoostingRegressor(n_estimators=50)
```

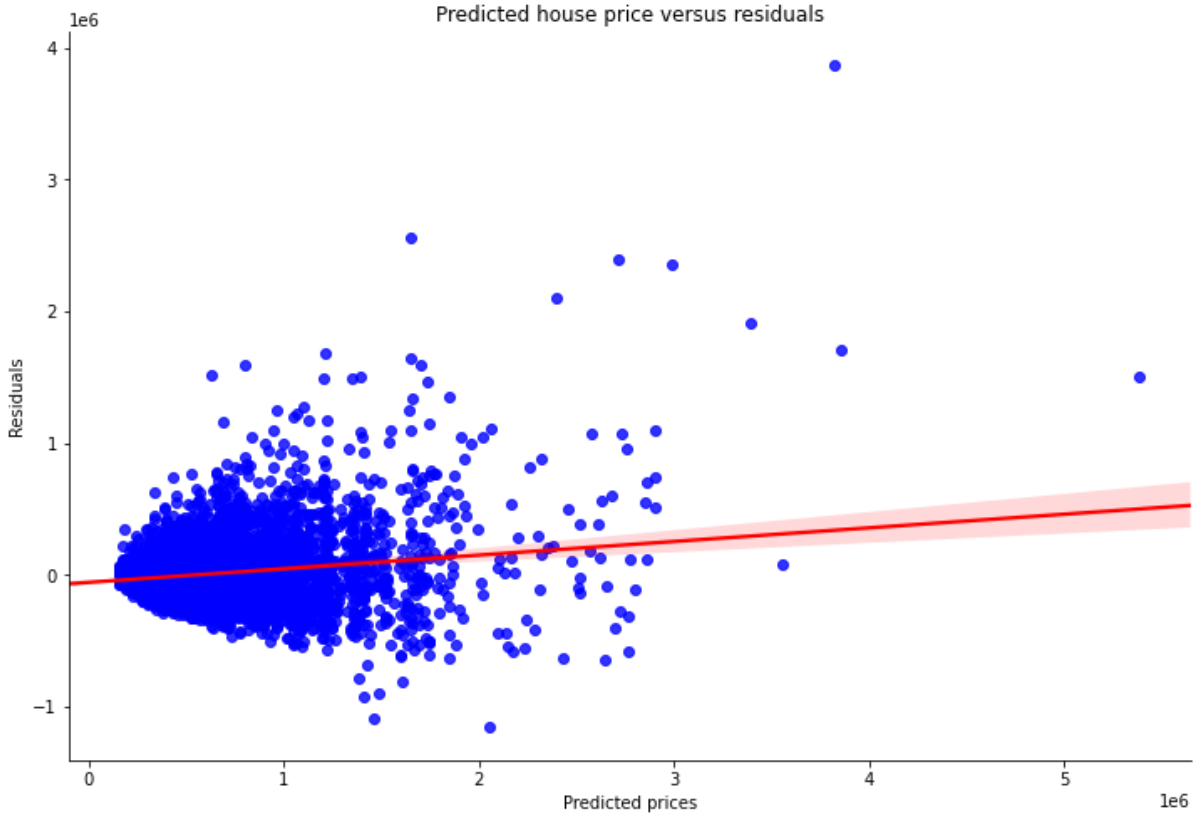
Gradient Boost: Model evaluation (train)

```
r^2: 0.7874810140823862
Adjusted r^2: 0.7872278478516439
Root mean squared error or RMSE: 166600.37369182028
Mean absolute percentage error or MAPE: 0.2045360022551658
```

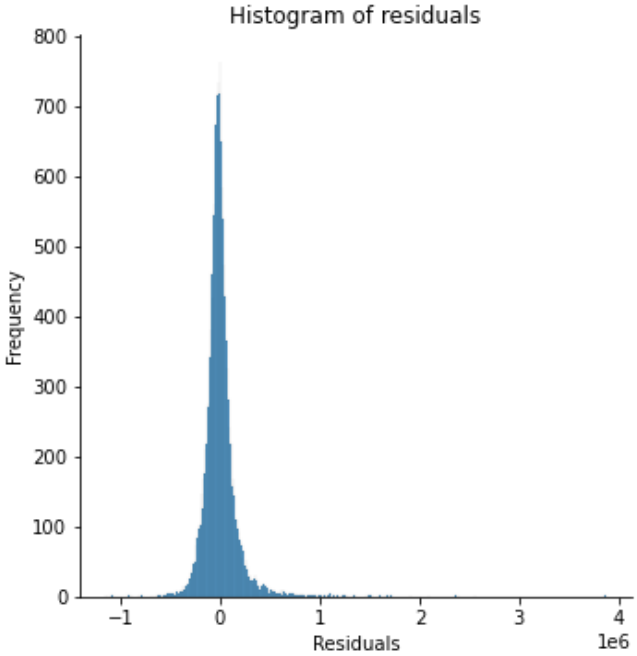
Gradient Boost:
Visualising the difference between actual and predicted house prices



Gradient Boost: Checking residuals



Gradient Boost: Checking for normality of errors



Gradient Boost: Model evaluation (test)

```
r^2: 0.761253443585104
Adjusted r^2: 0.7605887199941577
Root mean squared error or RMSE: 186116.16559844464
Mean absolute percentage error or MAPE: 0.20705725399318028
```

XGBoost: Tuned model

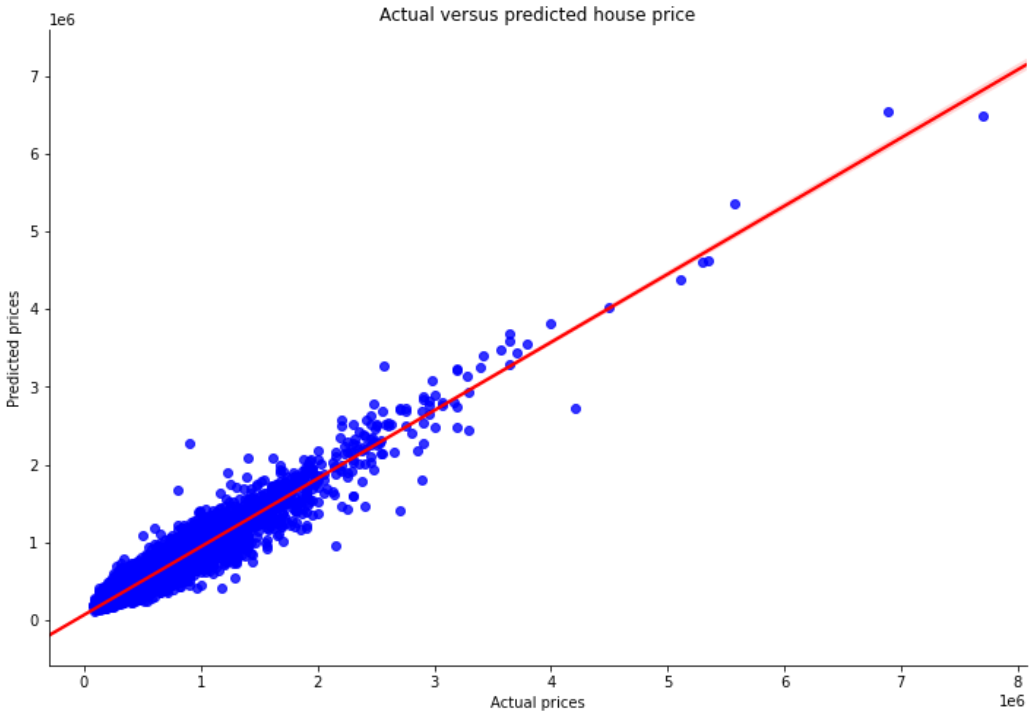
Create and train

```
XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
             colsample_bynode=1, colsample_bytree=0.5, gamma=0, gpu_id=-1,
             importance_type='gain', interaction_constraints='',
             learning_rate=0.01, max_delta_step=0, max_depth=7,
             min_child_weight=1, missing=nan, monotone_constraints='()',
             n_estimators=1000, n_jobs=8, num_parallel_tree=1, random_state=0,
             reg_alpha=0, reg_lambda=1, scale_pos_weight=1, subsample=0.6,
             tree_method='exact', validate_parameters=1, verbosity=None)
```

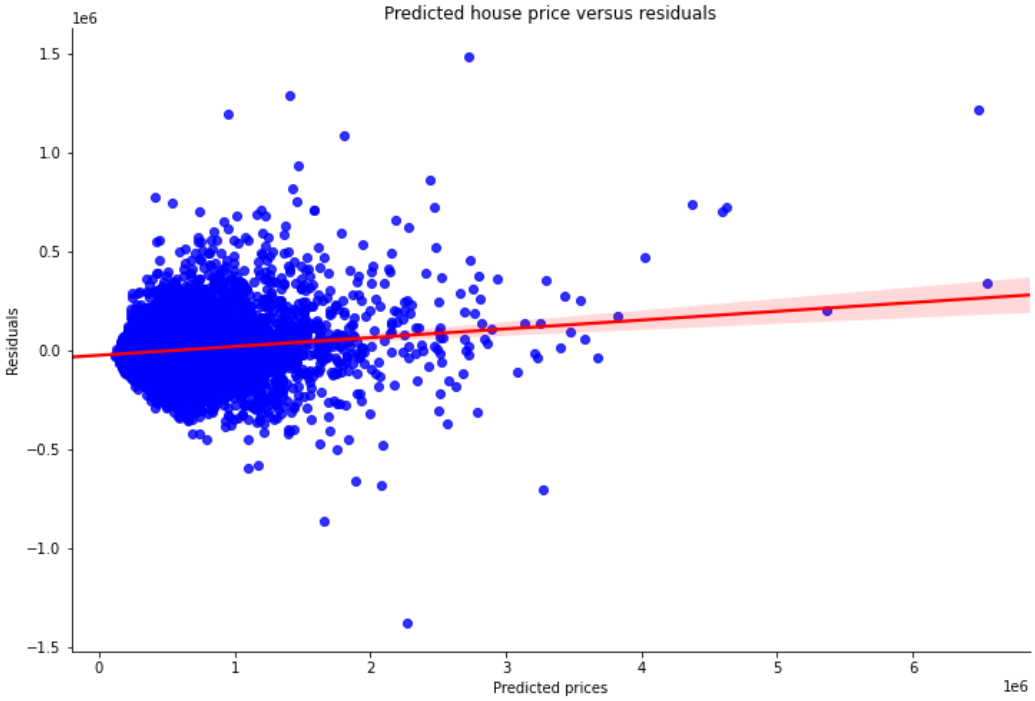
XGBoost tuned: Model evaluation on train set

```
r^2: 0.9137794464147787
Adjusted r^2: 0.9135795047613312
Root mean squared error or RMSE: 106116.39978410782
Mean absolute percentage error or MAPE: 0.1443156095718864
```

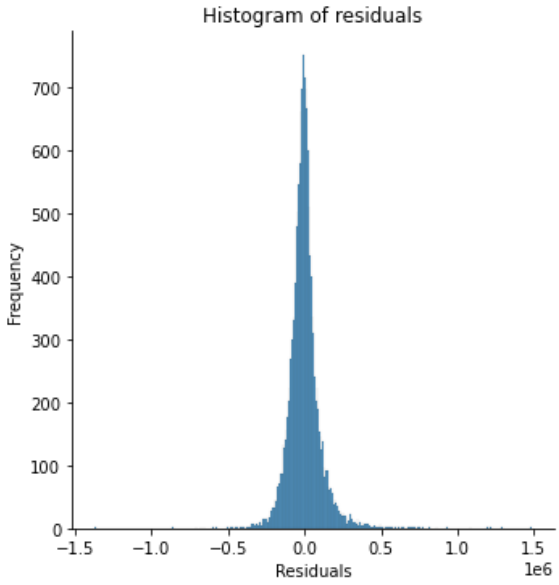
XGBoost tuned:
Visualising the difference between actual and predicted house prices



XGBoost tuned: Checking residuals



XGBoost tuned: checking for normality of errors



XGBoost tuned: Model evaluation (test)

```
r^2: 0.8159721276027667
Adjusted r^2: 0.814973217005077
Root mean squared error or RMSE: 163401.98988329218
Mean absolute percentage error or MAPE: 0.1703578743816115
```

Ridge

Create and train

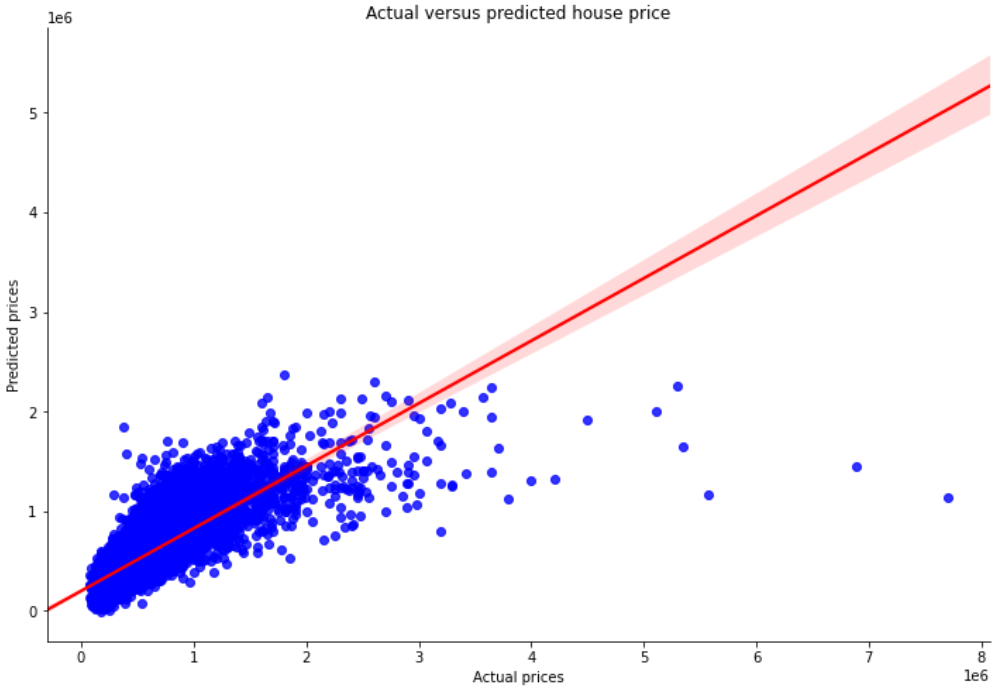
```
Ridge(alpha=0.3)
```

Ridge: Model evaluation (train)

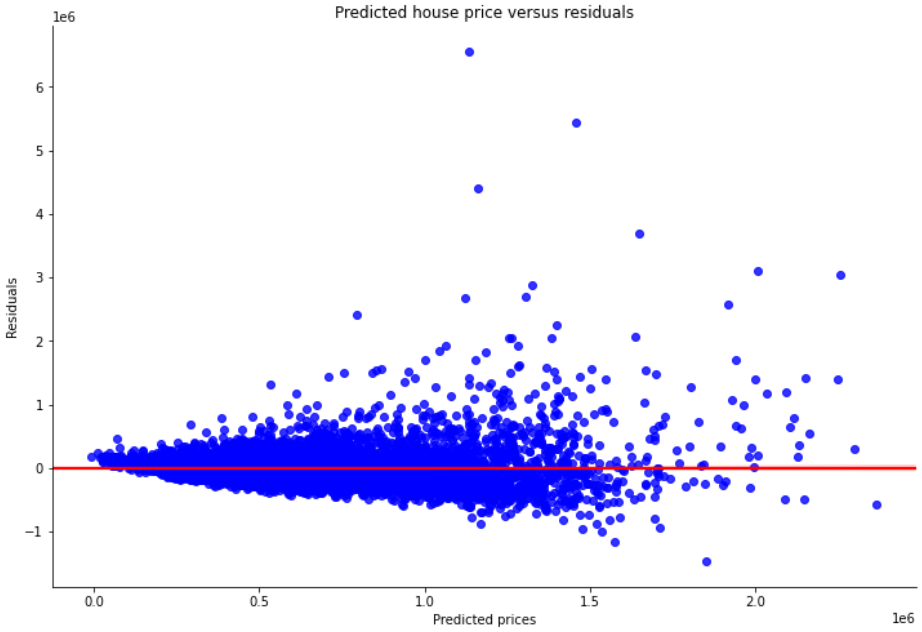
```
r^2: 0.6273528774491531
Adjusted r^2: 0.6269089563236789
Root mean squared error or RMSE: 220610.403935612
Mean absolute percentage error or MAPE: 0.244340262675872 78
```

Ridge:

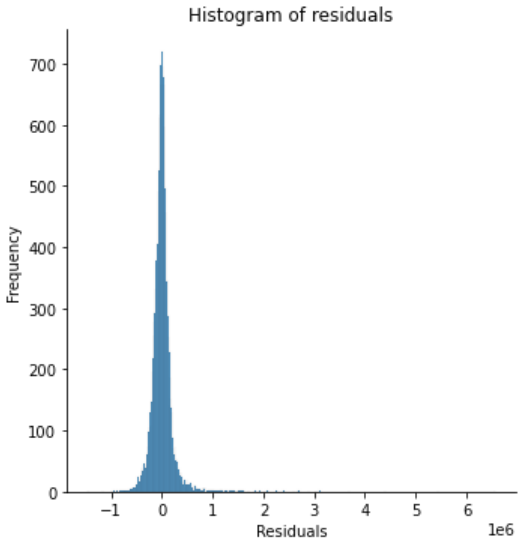
Visualising the difference between actual and predicted prices



Ridge: Checking residuals



Ridge: Checking for normality of residuals



Ridge: Model evaluation

```
r^2: 0.6398583343857234
Adjusted r^2: 0.6388556197714841
Root mean squared error or RMSE: 228587.3657460667
Mean absolute percentage error or MAPE: 0.24803032230313385
```

Lasso

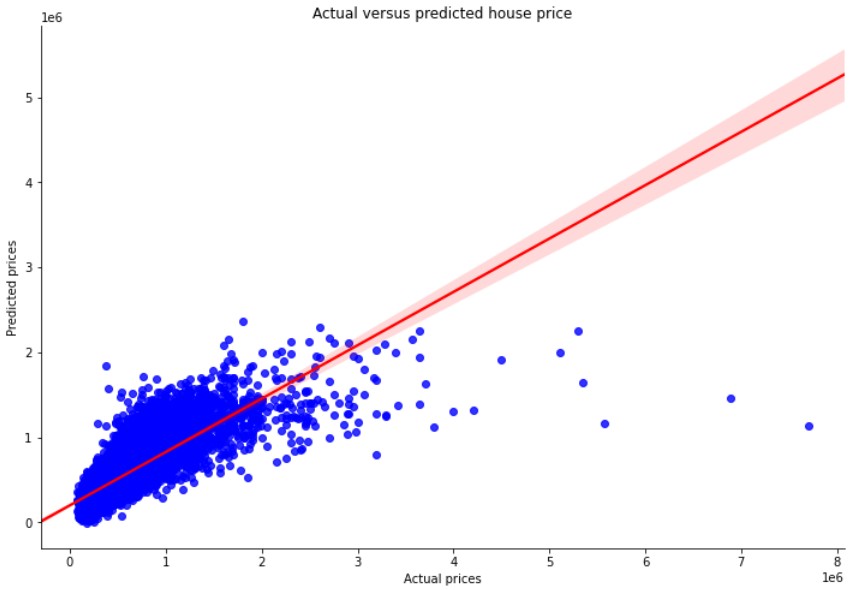
Create and train

```
Lasso(alpha=0.1)
```

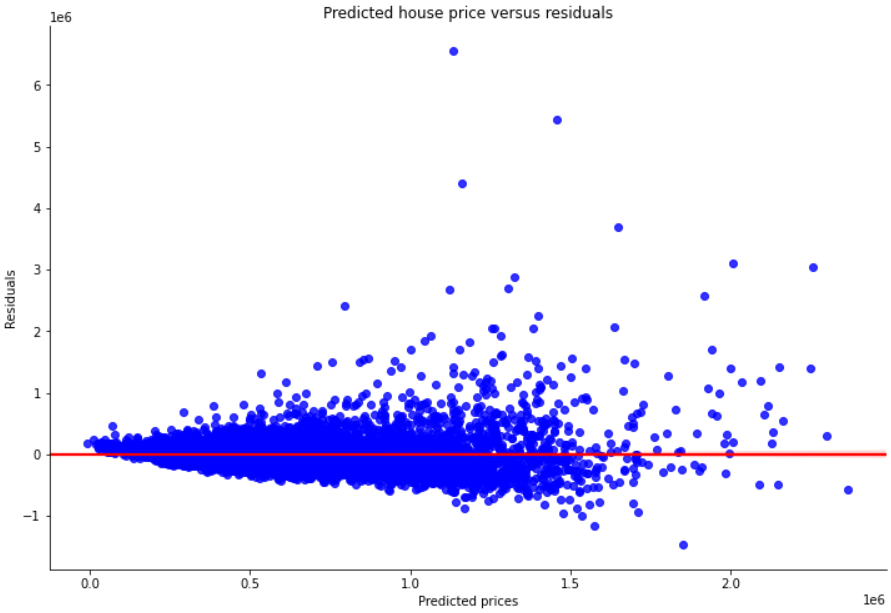
Lasso: Model evaluation (train)

```
r^2: 0.6273528776381445
Adjusted r^2: 0.6269089565128954
Root mean squared error or RMSE: 220610.4038796697
Mean absolute percentage error or MAPE: 0.24434111325199256
```

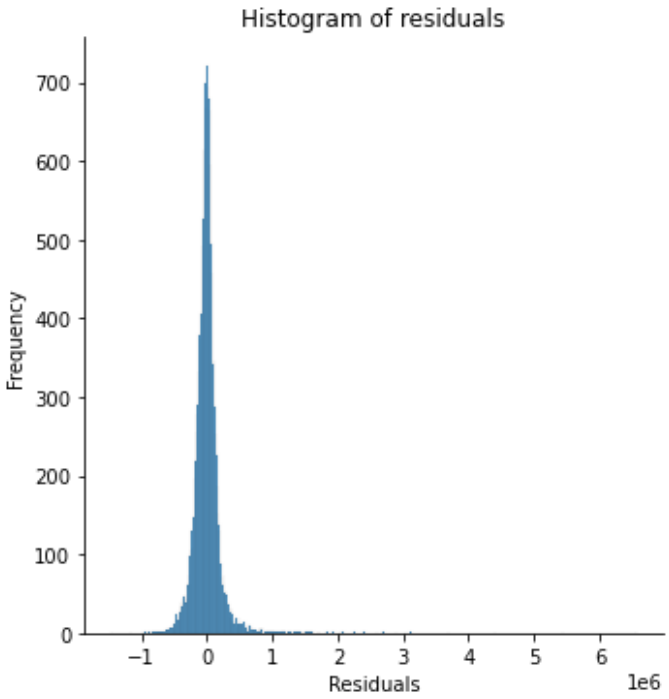
Lasso:
Visualising the difference between actual and predicted prices¶



Lasso: Checking residuals



Lasso: Checking for normality of errors



Lasso: Model evaluation (test)

```
r^2: 0.639858724122464
Adjusted r^2: 0.6388560105933386
Root mean squared error or RMSE: 228587.24206012936
Mean absolute percentage error or MAPE: 0.24803109365612538
```

Support vector regressor

Create and train

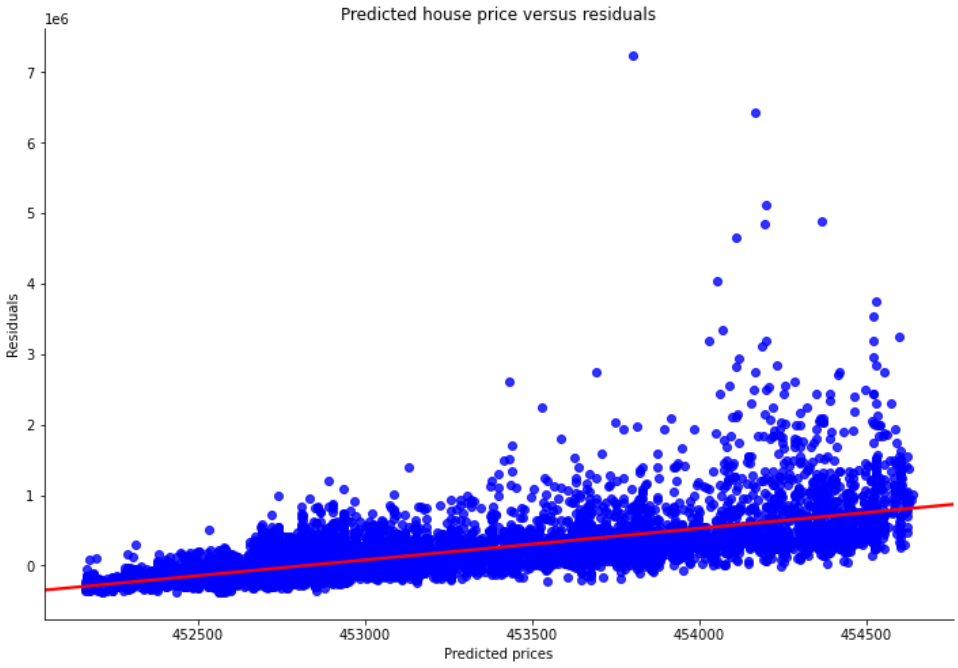
```
SVR()
```

Support vector regressor: Model evaluation (train)

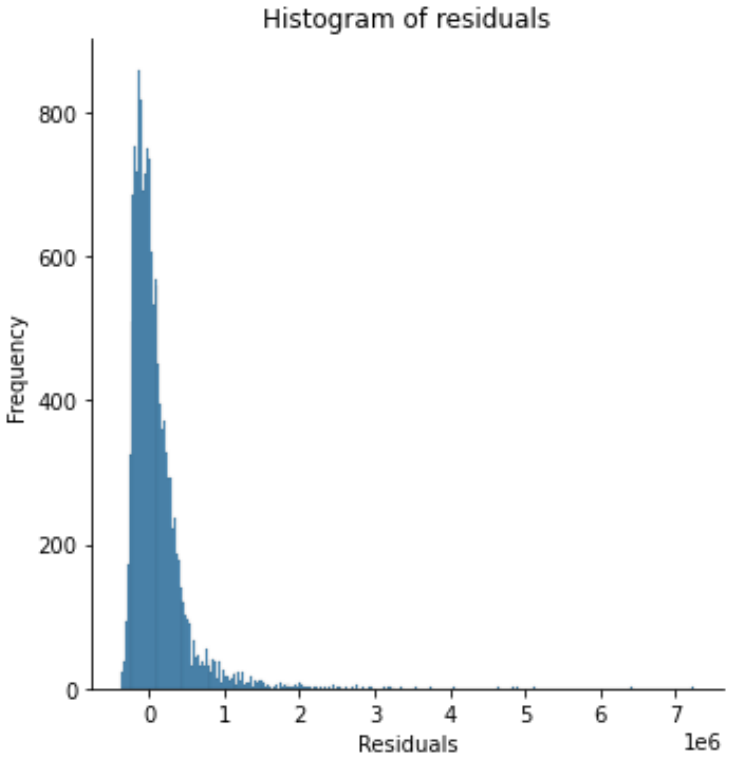
```
r^2: -0.05585952235188274
Adjusted r^2: -0.05711732985700069
Root mean squared error or RMSE: 371347.2176333003
Mean absolute percentage error or MAPE: 0.42459050529749787
```

SVR:
Visualising the difference between actual and predicted house prices

SVR: Checking residuals



SVR: Checking for normality or errors



SVR: Model evaluation (test)

```
r^2: -0.050550766791321866
Adjusted r^2: -0.05347573412345552
Root mean squared error or RMSE: 390412.8112790113
Mean absolute percentage error or MAPE: 0.4334109414536356
```

KNN

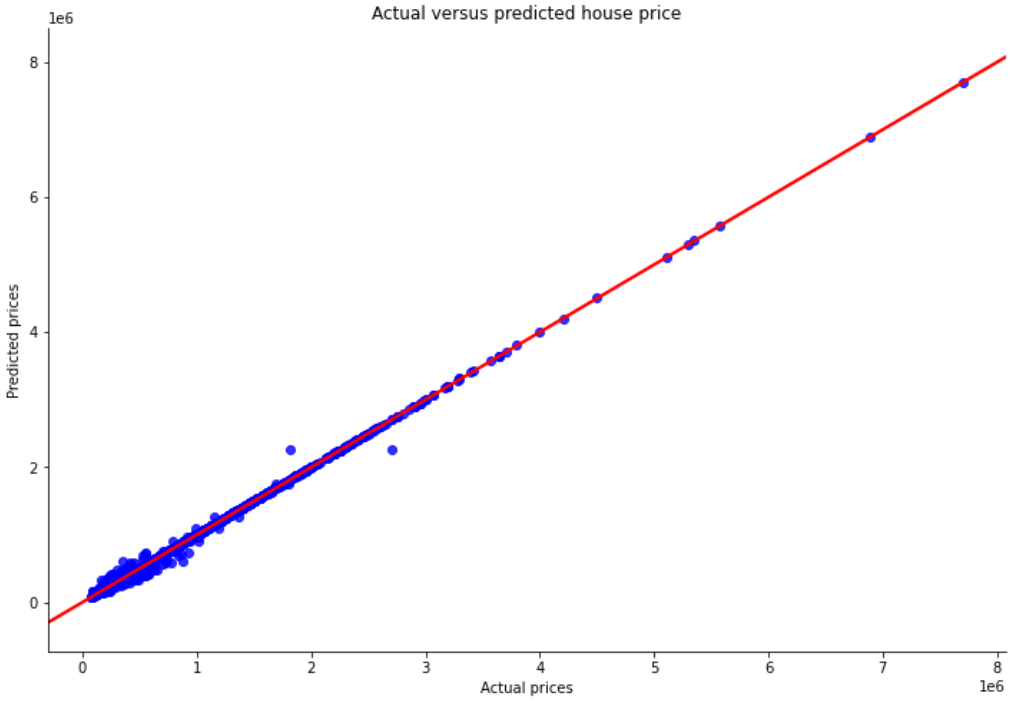
Create and train

```
KNeighborsRegressor(n_neighbors=4, weights='distance')
```

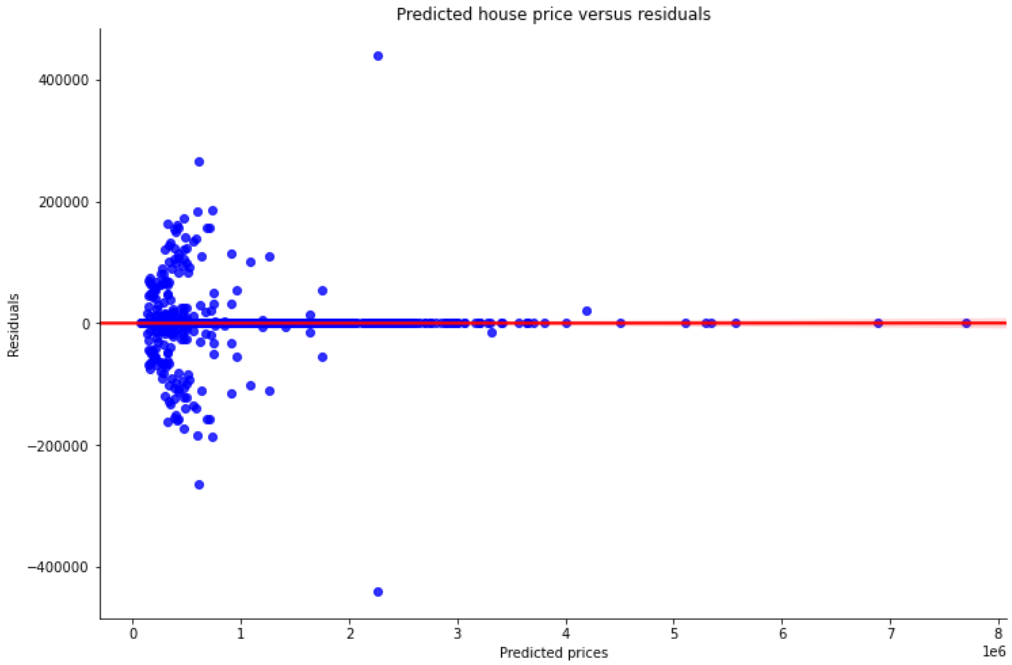
KNN: Model evaluation (train)

```
r^2: 0.9990141897396491
Adjusted r^2: 0.9990130153793124
Root mean squared error or RMSE: 11346.80968660604
Mean absolute percentage error or MAPE: 0.0028600720035641455
```

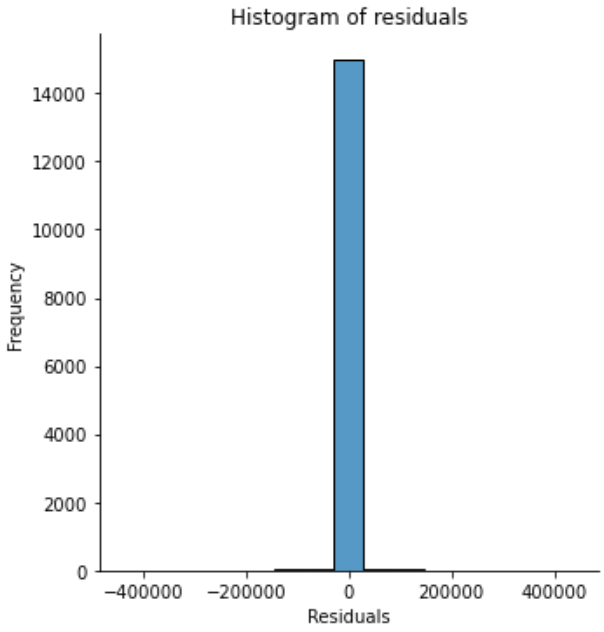
KNN:
Visualising the difference between actual and predicted house prices



KNN: Checking residuals



KNN: Checking for normality of errors



KNN: Model evaluation (test)

```
r^2: 0.6942518305589358
Adjusted r^2: 0.6934005595535314
Root mean squared error or RMSE: 210618.963505483
Mean absolute percentage error or MAPE: 0.20979355850654846
```

Elastic Net regressor

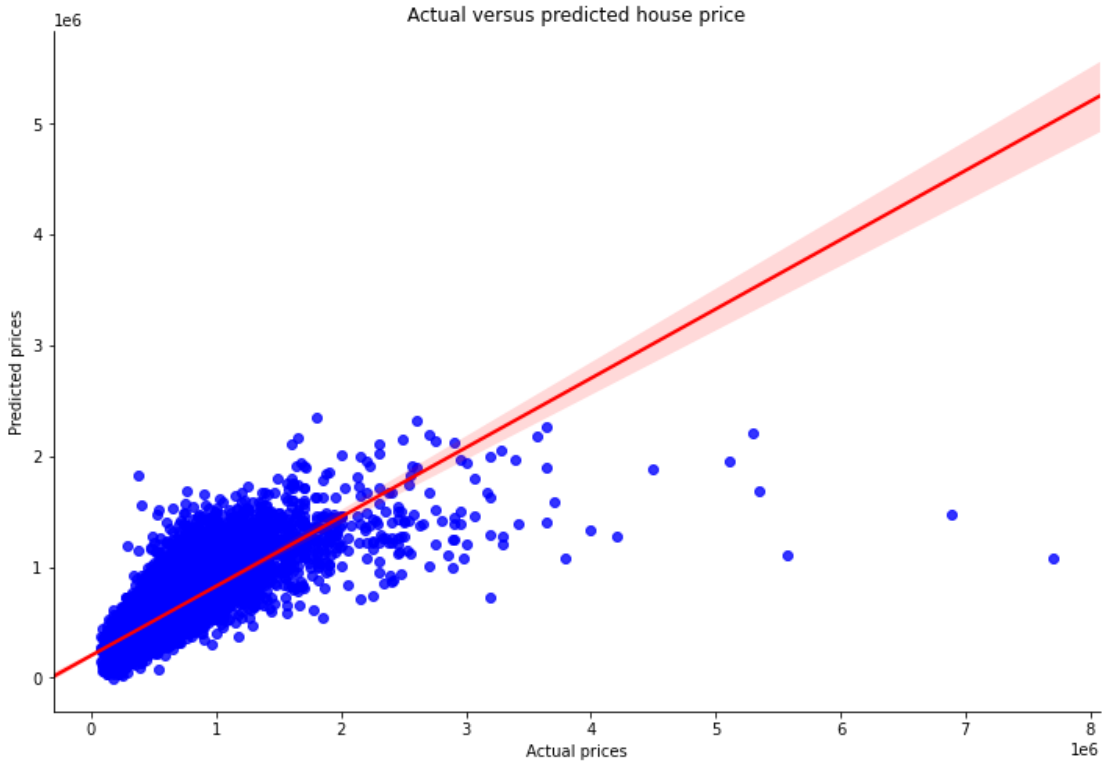
Create and train

```
ElasticNet()
```

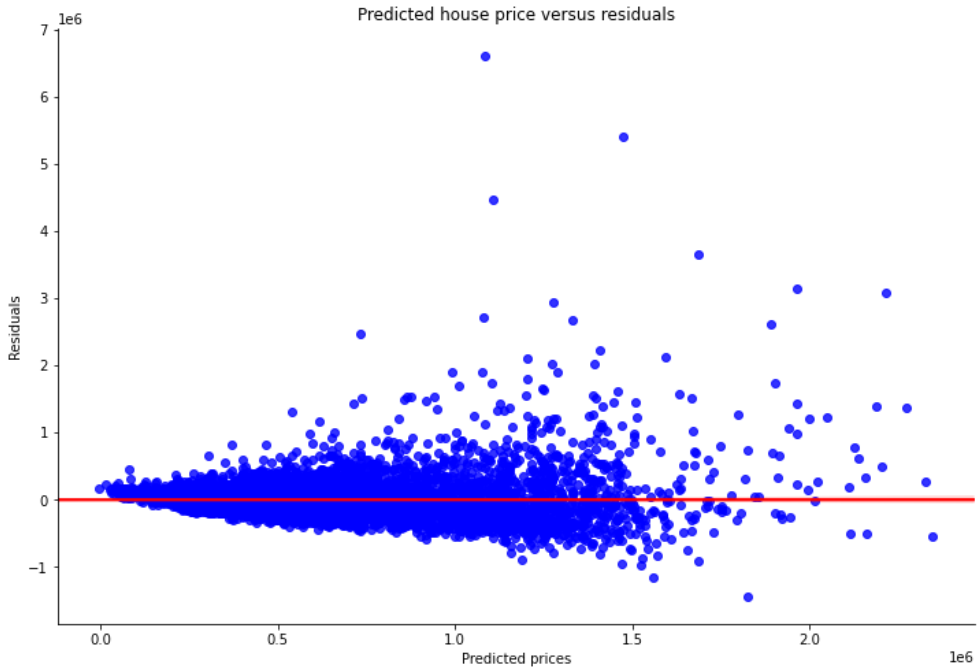
Elastic Net: Model evaluation (train)

```
r^2: 0.6253153750629636
Adjusted r^2: 0.6248690267341174
Root mean squared error or RMSE: 221212.69153222992
Mean absolute percentage error or MAPE: 0.24330046865225075
```

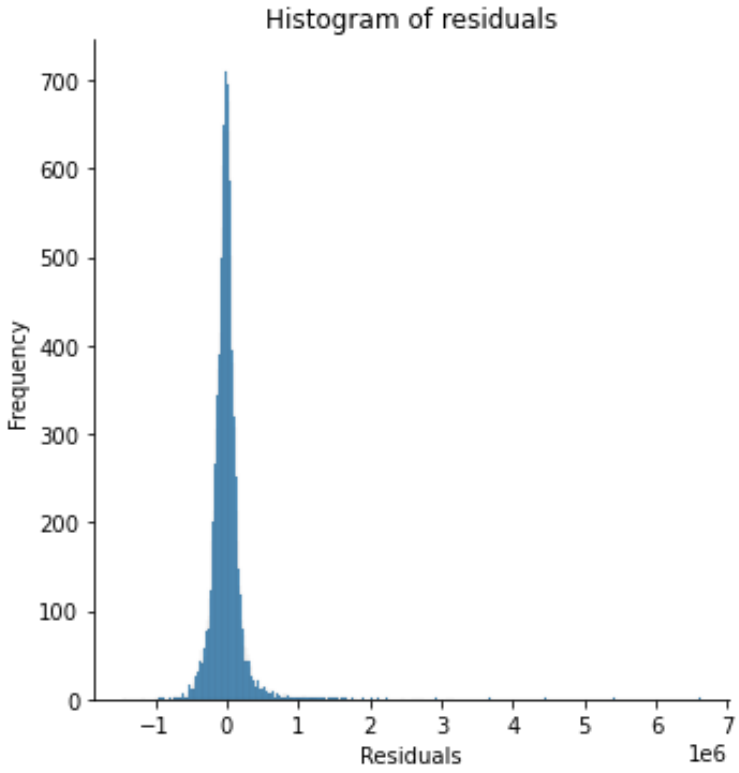
Elastic Net:
Visualising the difference between actual and predicted house prices



Elastic Net: Checking residuals



Elastic Net: Checking for normality of errors



Elastic Net: Model evaluation (test)

```
r^2: 0.6366379006220471
Adjusted r^2: 0.6356262196028972
Root mean squared error or RMSE: 229607.1201503086
Mean absolute percentage error or MAPE: 0.24743567528379723
```

Models on scaled data

Scaled train set

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	-0.636615	-0.268199	0.722608	-0.207925	-0.375347	0.444542	1.236563	0.004337	0.000091	-1.055994	-0.157354	-0.836817	-0.279621	-1.275236
1	-0.281026	-0.606782	1.271653	-0.207925	-0.615531	0.682942	1.236563	0.596984	0.000091	-1.055994	-0.157354	-0.836817	-0.279621	-1.045968
2	0.759403	0.004654	1.820697	-0.207925	0.163381	0.274256	1.236563	1.653441	0.000091	0.359859	-0.157354	-0.836817	-0.279621	-1.275236
3	-0.741975	-1.056960	-0.663077	-0.207925	-1.126430	-1.156145	-0.808693	-1.206723	0.000091	0.359859	-0.157354	1.229028	-0.279621	1.182184
4	0.680383	1.676243	1.951422	-0.207925	1.693106	-0.270659	1.236563	1.163863	0.000091	0.359859	-0.157354	-0.836817	-0.279621	-0.765359
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
15124	2.313461	-0.630445	0.722608	-0.207925	-0.105172	-1.053974	1.236563	-0.047197	0.000091	0.359859	-0.157354	1.229028	-0.279621	1.182184
15125	1.720812	0.058406	-0.663077	-0.207925	0.044774	-1.019917	-0.808693	-0.150266	0.000091	0.359859	-0.157354	1.229028	-0.279621	1.182184
15126	-0.794655	0.038541	-0.663077	-0.207925	-0.124085	0.614828	-0.808693	-0.897516	0.000091	-1.055994	-0.157354	-0.836817	-0.279621	-1.045968
15127	-0.315268	2.390218	-0.663077	-0.207925	2.149160	0.989457	-0.808693	0.658825	0.000091	0.359859	-0.157354	-0.836817	-0.279621	-0.226294
15128	-0.412725	0.668966	-0.663077	-0.207925	0.537301	0.580771	-0.808693	0.262010	0.000091	-1.055994	-0.157354	-0.836817	-0.279621	-1.045968

Scaled test set

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	-0.939525	-0.248918	-0.663077	-0.207925	-0.419655	-0.168487	-0.808693	-1.077887	0.000091	-1.055994	-0.157354	-0.836817	-0.279621	-0.765359
1	1.733982	1.670985	-0.663077	-0.207925	1.904382	0.240199	1.236563	1.241165	2.954403	0.359859	-0.157354	-0.836817	2.484486	-1.275236
2	-0.386385	0.446652	-0.663077	-0.207925	0.337102	0.512656	-0.808693	0.854656	0.000091	-1.055994	-0.157354	-0.836817	-0.279621	-1.275236
3	1.544334	1.744895	0.513448	-0.207925	1.933831	-0.475002	1.236563	-0.804754	0.000091	0.359859	-0.157354	1.229028	2.484486	0.014419
4	-0.320536	-0.985095	1.428523	-0.207925	-0.973512	1.364086	1.236563	0.622751	0.000091	0.359859	-0.157354	-0.836817	-0.279621	-0.226294
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
6479	-0.909234	-1.115387	-0.663077	4.848745	-1.214777	2.283629	-0.808693	0.553180	-1.691875	0.359859	-0.157354	-0.836817	-0.279621	0.972263
6480	-1.137075	-1.260285	-0.663077	-0.207925	-1.395524	-0.577173	-0.808693	-0.072964	0.000091	-1.055994	-0.157354	1.229028	-0.279621	0.014419
6481	-0.439065	-0.297412	-0.663077	-0.207925	-0.361838	1.772772	-0.808693	-0.665611	2.954403	-1.055994	-0.157354	-0.836817	-0.279621	0.972263
6482	-0.610275	-0.437637	-0.663077	-0.207925	-0.526644	1.227857	-0.808693	-0.279102	0.000091	0.359859	-0.157354	-0.836817	-0.279621	-0.226294
6483	1.378392	0.590674	2.605047	-0.207925	0.832331	0.172085	1.236563	2.606829	0.000091	-1.055994	-0.157354	-0.836817	-0.279621	-1.244322

Linear regression scaled

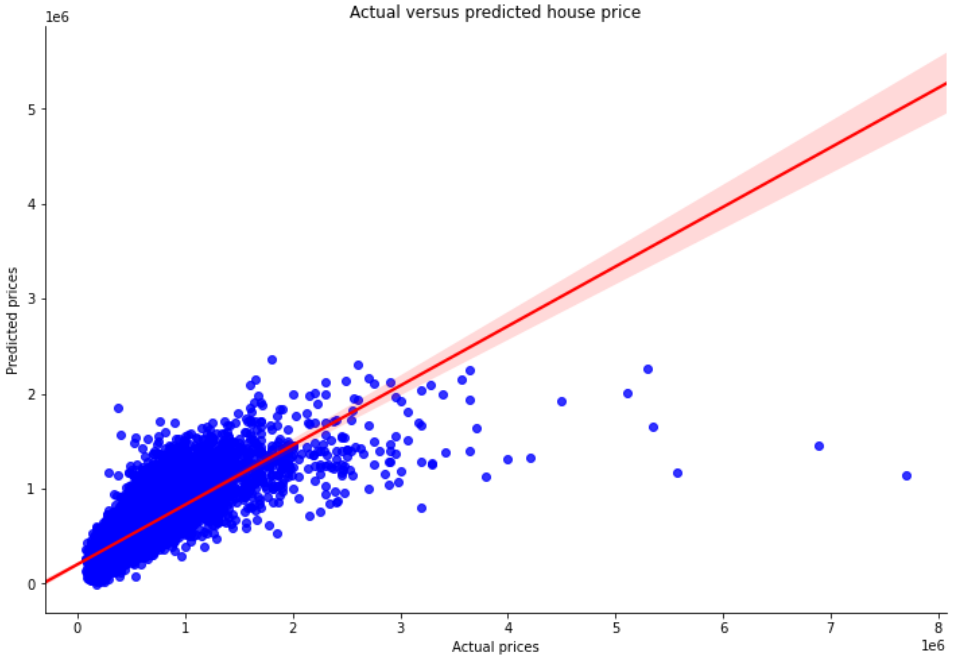
Create and train

```
LinearRegression()
```

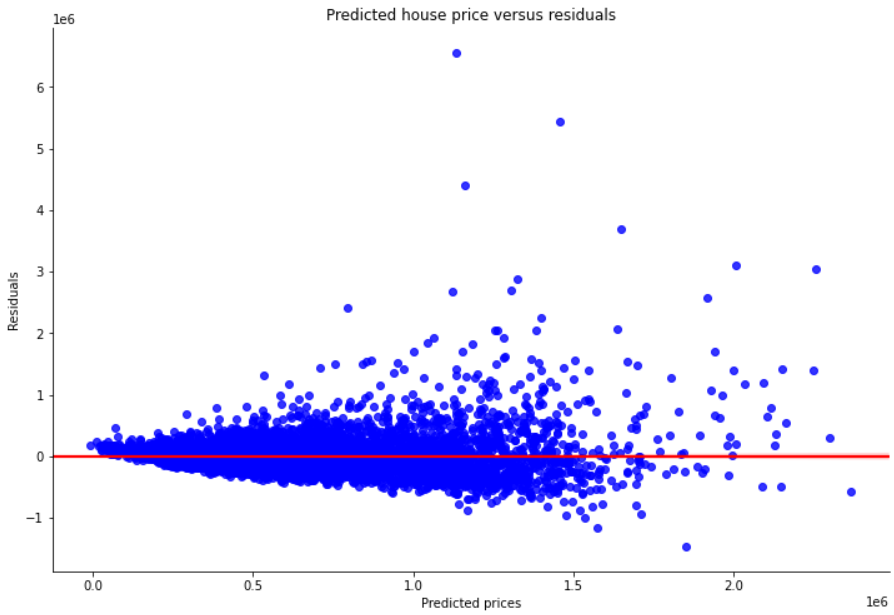
Linear Regression scaled: Model evaluation (train)

```
r^2: 0.6273528776391732
Adjusted r^2: 0.6269089565139254
Root mean squared error or RMSE: 220610.4038793652
Mean absolute percentage error or MAPE: 0.24434118200225677
```

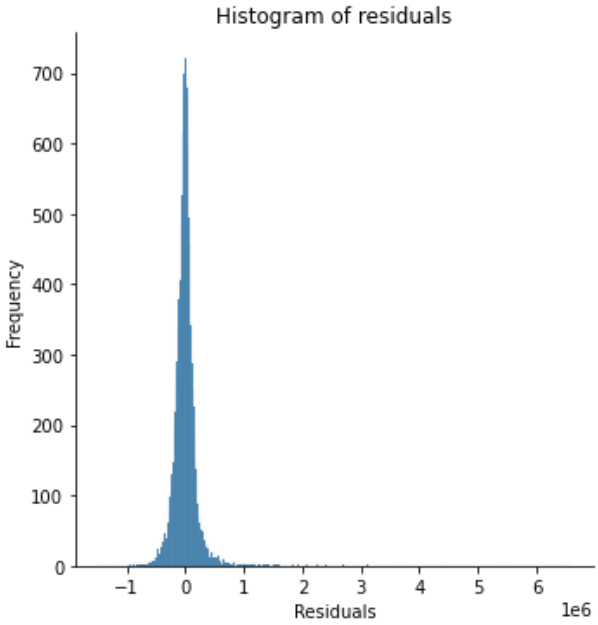
Linear regression scaled:
Visualising the difference between actual and predicted prices



Linear regression scaled: Inspecting residuals



Linear regression scaled: Checking normality of errors



Linear regression scaled: Model evaluation (test)

```
r^2: 0.6398587545622838
Adjusted r^2: 0.6388560411179096
Root mean squared error or RMSE: 228587.23239981677
Mean absolute percentage error or MAPE: 0.24803115635729867
```

Random forest scaled

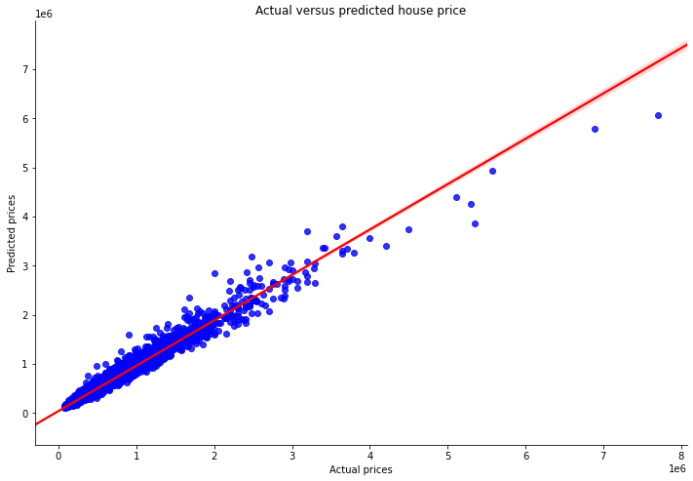
Create and train

```
RandomForestRegressor()
```

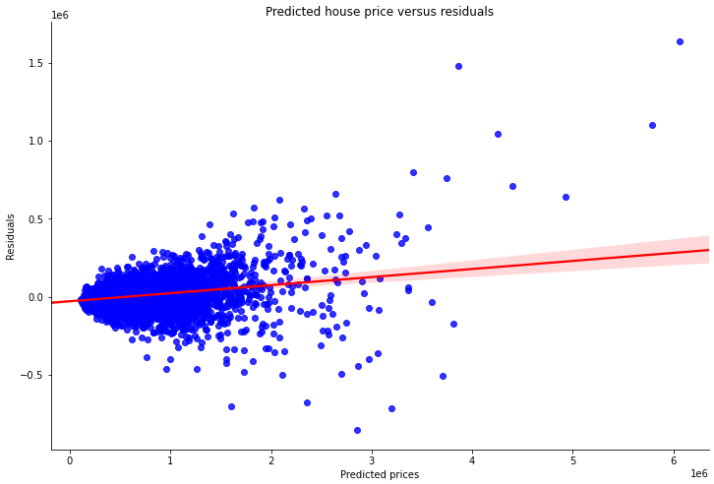
Random forest scaled: Model evaluation (train data)

```
r^2: 0.9685752095621305
Adjusted r^2: 0.9685377743385778
Root mean squared error or RMSE: 64063.88898081909
Mean absolute percentage error or MAPE: 0.0663024320349983
```

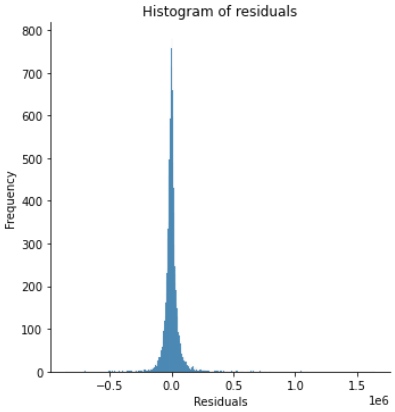
Random forest scaled:
Visualising the difference between actual and predicted prices



Random forest scaled: Inspecting residuals



Random forest scaled: Checking normality of errors



Random forest scaled: Model evaluation (test)

```
r^2: 0.800588208392578
Adjusted r^2: 0.8000330015481955
Root mean squared error or RMSE: 170094.768360897
Mean absolute percentage error or MAPE: 0.17285611292848962
```

XGBoost scaled

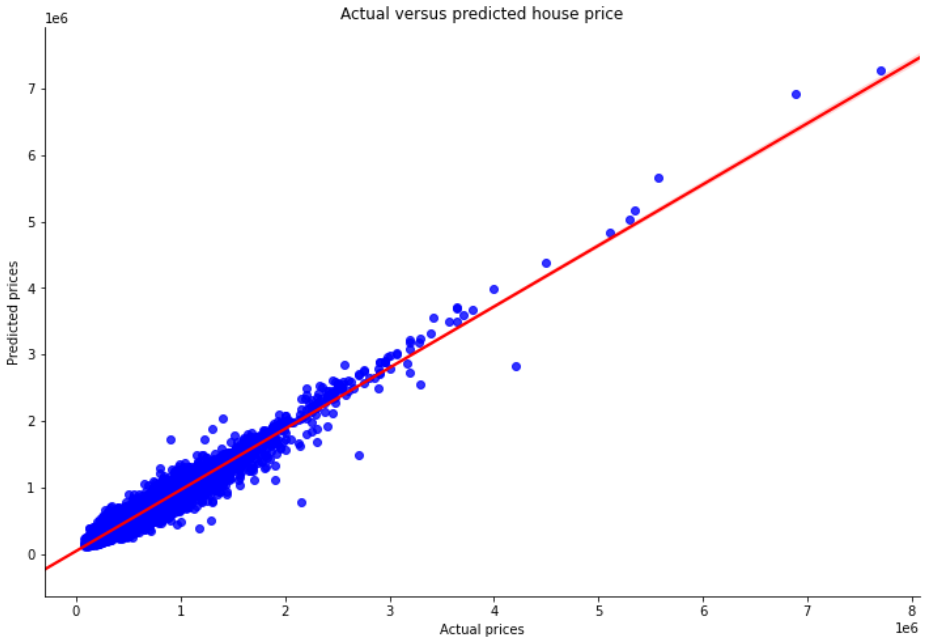
Create and train

```
XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
             colsample_bynode=1, colsample_bytree=1, gamma=0, gpu_id=-1,
             importance_type='gain', interaction_constraints='',
             learning_rate=0.300000012, max_delta_step=0, max_depth=6,
             min_child_weight=1, missing=nan, monotone_constraints='()',
             n_estimators=100, n_jobs=8, num_parallel_tree=1, random_state=0,
             reg_alpha=0, reg_lambda=1, scale_pos_weight=1, subsample=1,
             tree_method='exact', validate_parameters=1, verbosity=None)
```

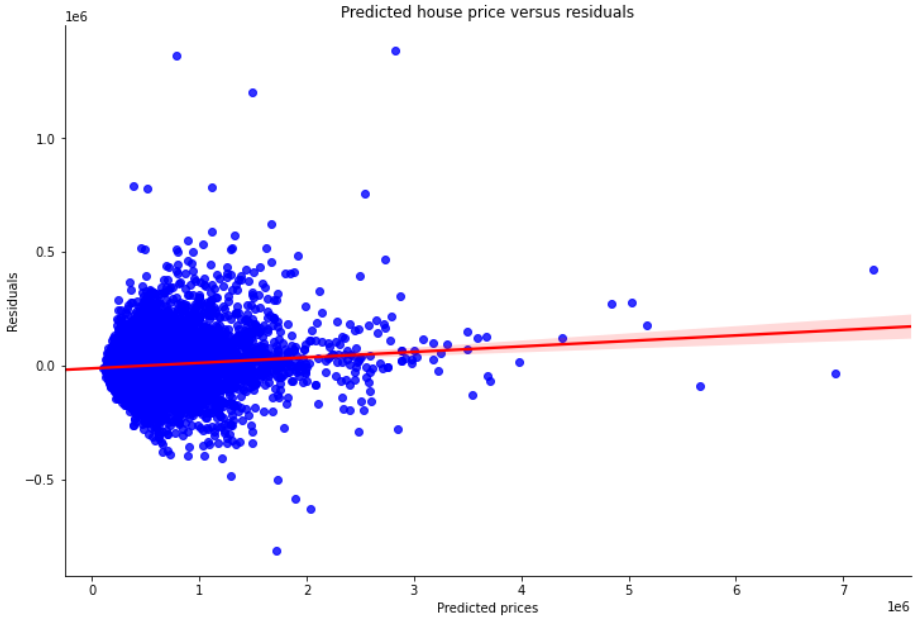
XGBoost scaled: Model evaluation on train data

```
r^2: 0.9404692782499442
Adjusted r^2: 0.9403312291370275
Root mean squared error or RMSE: 88175.44857718151
Mean absolute percentage error or MAPE: 0.1293323960730077
```

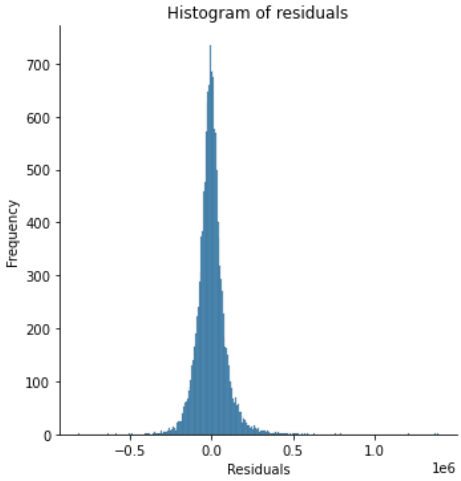
XGBoost scaled:
Visualising the difference between actual and predicted prices



XGBoost scaled: Inspecting residuals



XGBoost: scaled Checking normality of errors



XGBoost scaled: Model evaluation (test data)

```
r^2: 0.8080835517387697
Adjusted r^2: 0.8070418216380961
Root mean squared error or RMSE: 166867.45306724054
Mean absolute percentage error or MAPE: 0.17677506195663434
```

Decision tree scaled

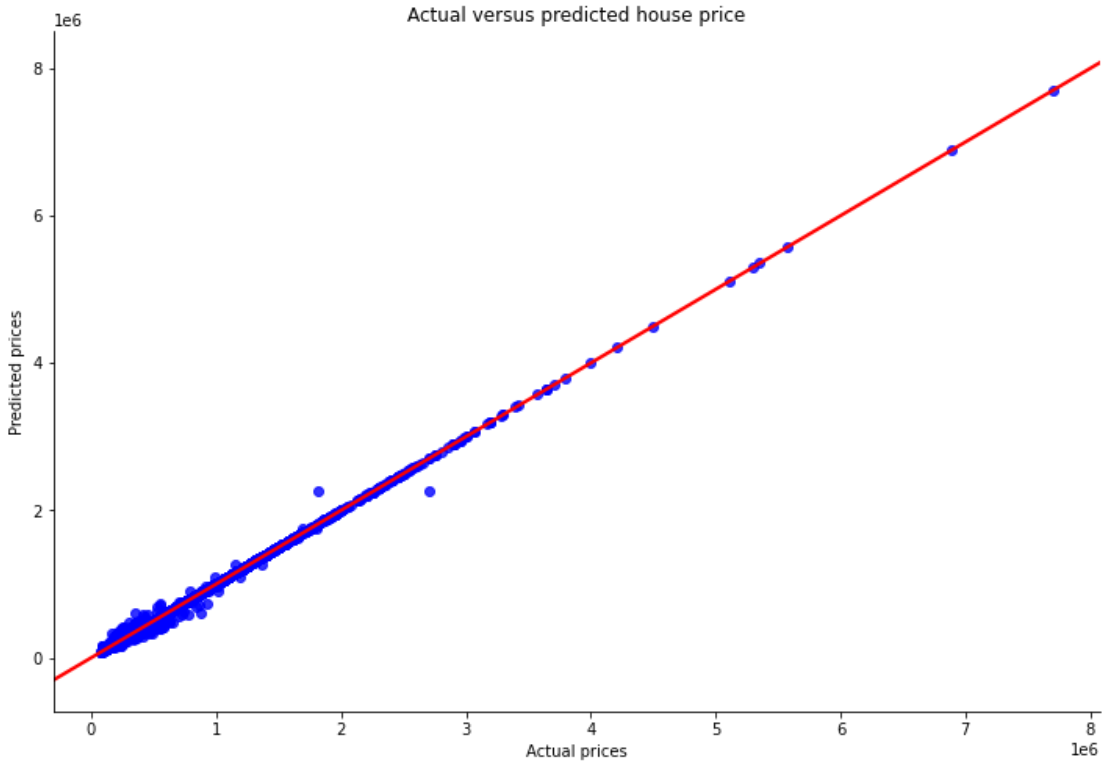
Create and train

```
DecisionTreeRegressor()
```

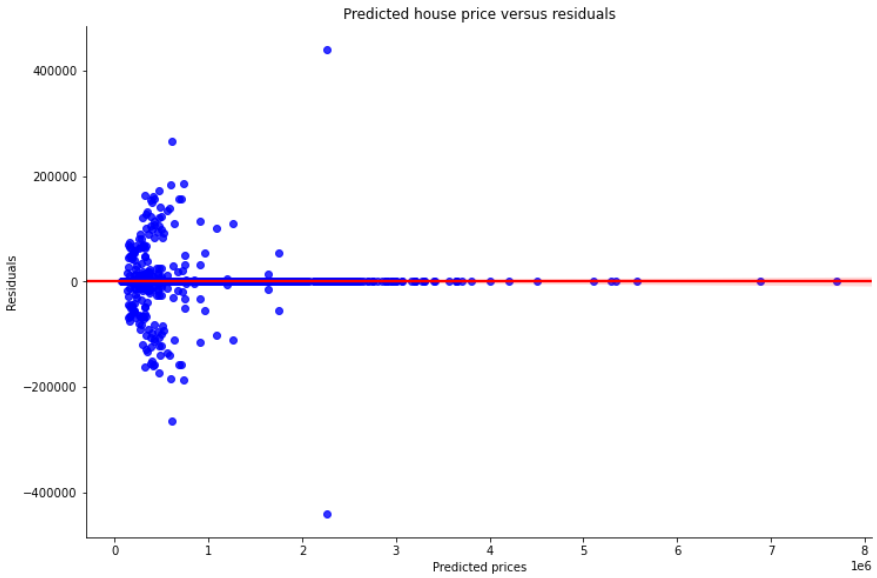
Decision tree scaled: Model evaluation (train)

```
r^2: 0.999014565353217
Adjusted r^2: 0.9990133914403353
Root mean squared error or RMSE: 11344.647799122395
Mean absolute percentage error or MAPE: 0.002850832910236522
```

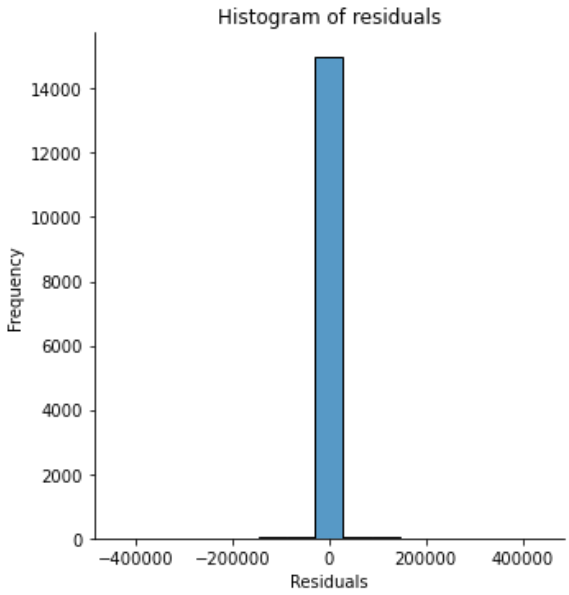
Decision tree scaled:
Visualising the difference between actual and predicted house prices



Decision tree scaled: Checking residuals



Decision tree scaled: Checking normality of errors



Decision tree scaled: Model evaluation (test)

```
r^2: 0.6691976902820225
Adjusted r^2: 0.6682766629695827
Root mean squared error or RMSE: 219078.52336594055
Mean absolute percentage error or MAPE: 0.23772202699399753
```

Bagging scaled

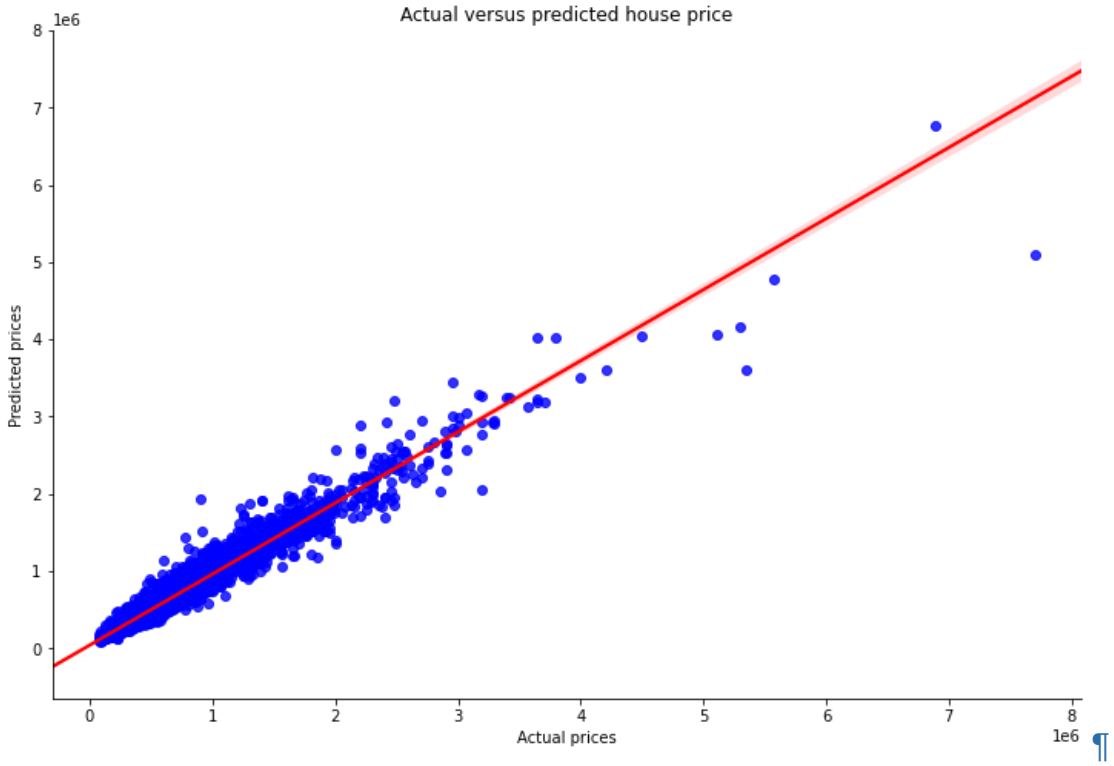
Create and train

```
BaggingRegressor()
```

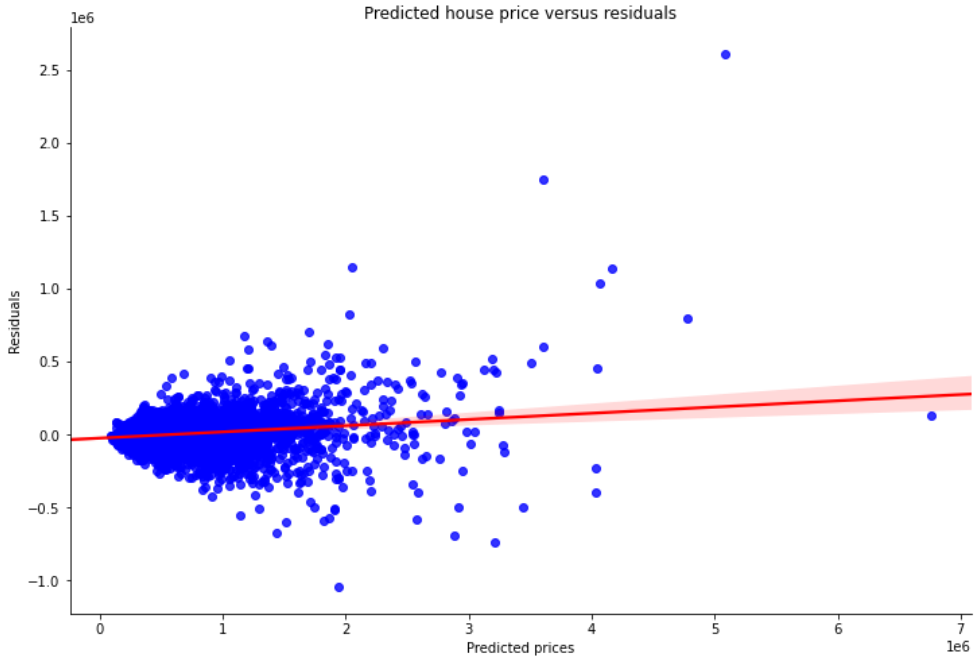
Bagging scaled: Model evaluation (train)

```
r^2: 0.9573837166125085
Adjusted r^2: 0.9573329493655877
Root mean squared error or RMSE: 74604.4769046783
Mean absolute percentage error or MAPE: 0.0739875874404801
```

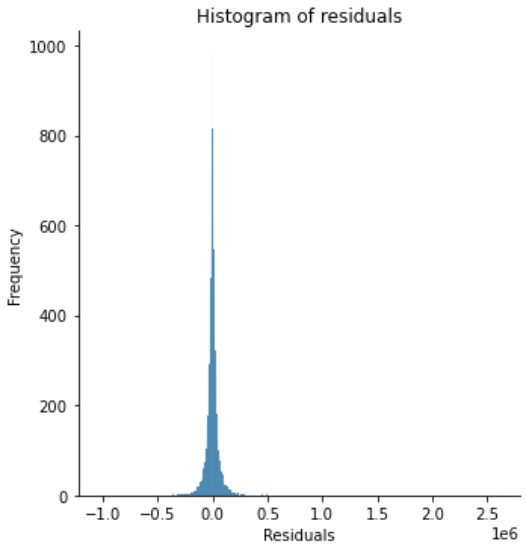
Bagging scaled:
Visualising the difference between actual and predicted house prices



Bagging scaled: Checking residuals



Bagging scaled: Checking normality of errors



Bagging scaled: Model evaluation (test)

```
r^2: 0.7768246959080032
Adjusted r^2: 0.7762033261518306
Root mean squared error or RMSE: 179944.51203244942
Mean absolute percentage error or MAPE: 0.18144741777419024
```

Gradient Boost scaled

Create and train

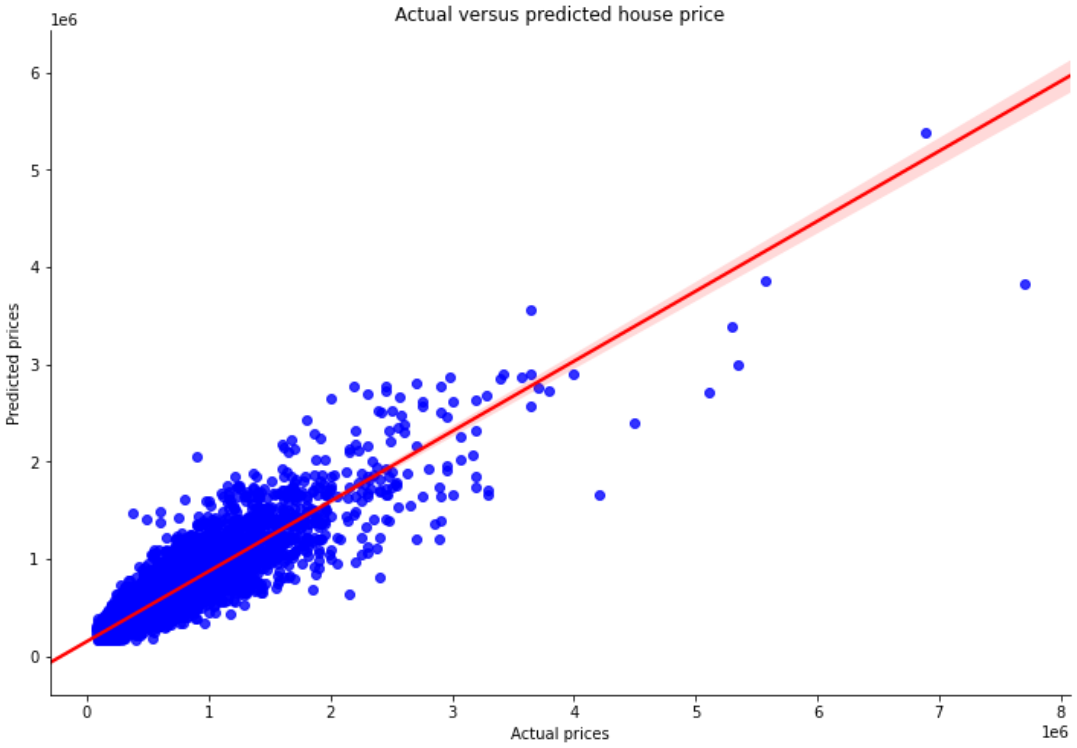
```
GradientBoostingRegressor(n_estimators=50)
```

Gradient Boost scaled: Model evaluation (train)

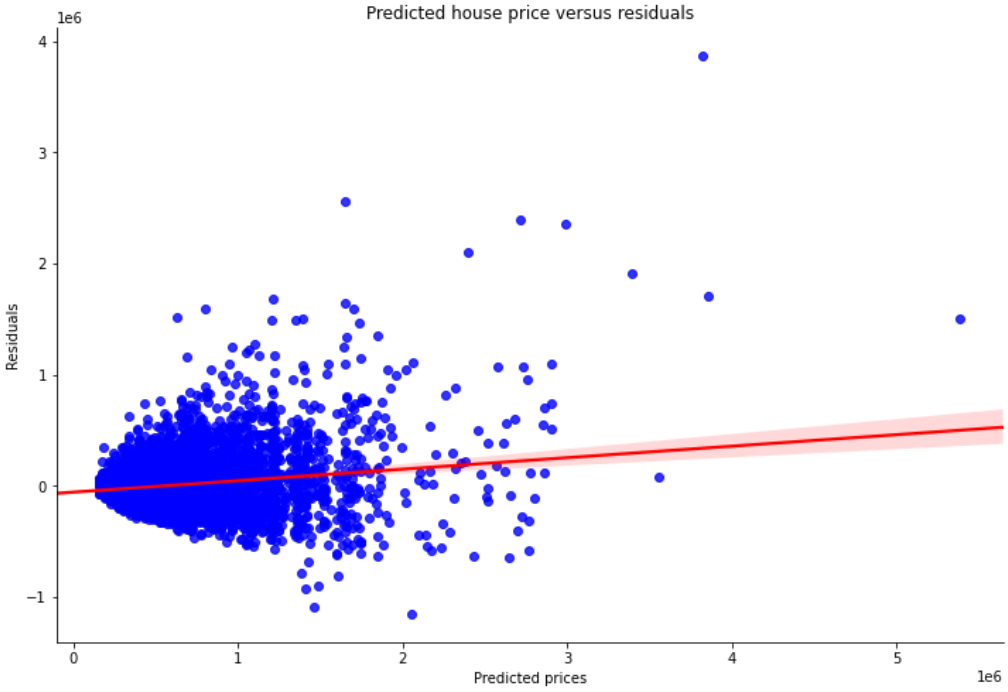
```
r^2: 0.7874810140823862
Adjusted r^2: 0.7872278478516439
Root mean squared error or RMSE: 166600.37369182028
Mean absolute percentage error or MAPE: 0.20453600225516583
```

¶

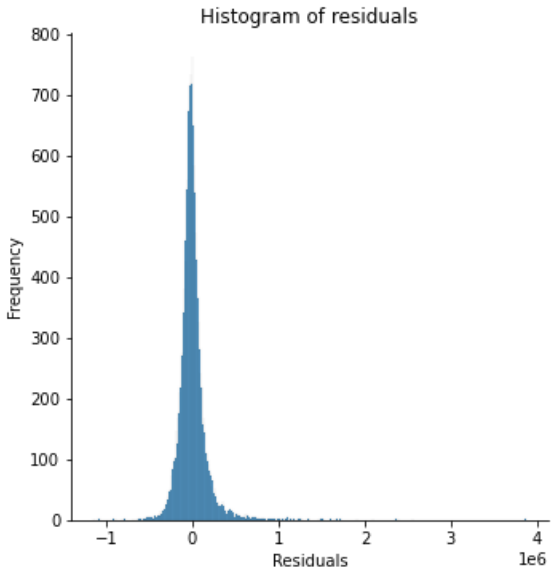
Gradient Boost scaled:
Visualising the difference between actual and predicted house prices



Gradient Boost scaled: Checking residuals



Gradient Boost scaled: Checking normality of errors



Gradient Boost scaled: Model evaluation (test data)

```
r^2: 0.761194994834335
Adjusted r^2: 0.7605301085090477
Root mean squared error or RMSE: 186138.9462237062
Mean absolute percentage error or MAPE: 0.20706015473385714
```

XGBoost tuned scaled

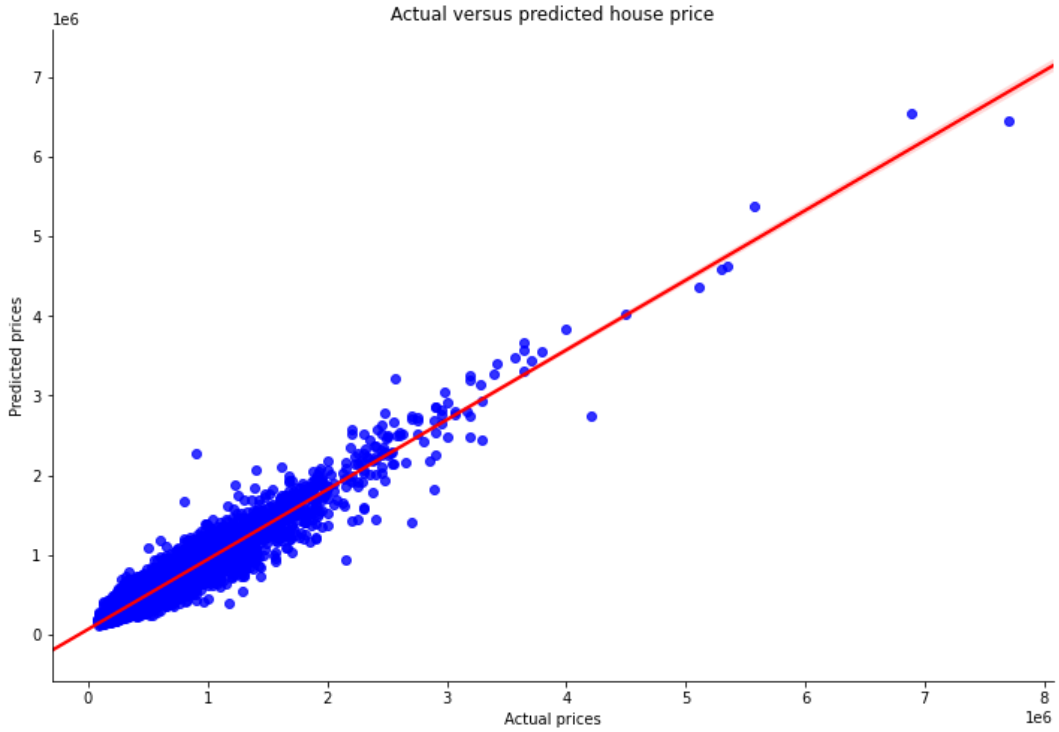
Create and train

```
XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
             colsample_bynode=1, colsample_bytree=0.5, gamma=0, gpu_id=-1,
             importance_type='gain', interaction_constraints='',
             learning_rate=0.01, max_delta_step=0, max_depth=7,
             min_child_weight=1, missing=nan, monotone_constraints='()',
             n_estimators=1000, n_jobs=8, num_parallel_tree=1, random_state=0,
             reg_alpha=0, reg_lambda=1, scale_pos_weight=1, subsample=0.6,
             tree_method='exact', validate_parameters=1, verbosity=None)
```

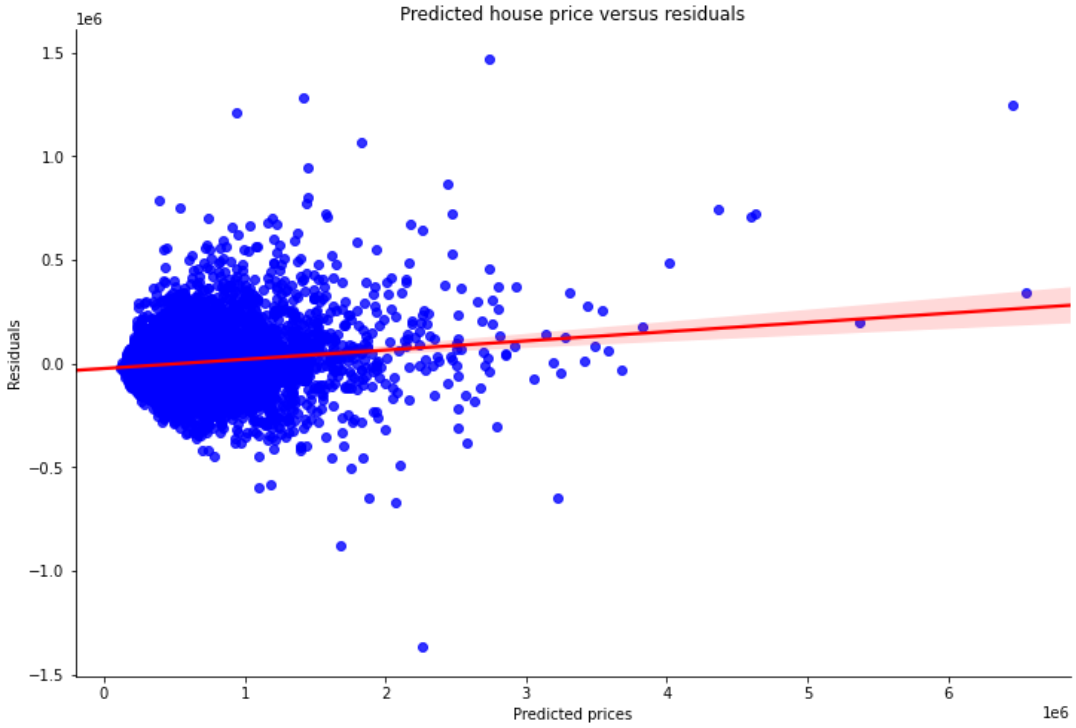
XGBoost tuned scaled: Model evaluation (train)

```
r^2: 0.9136373498833467
Adjusted r^2: 0.9134370787143224
Root mean squared error or RMSE: 106203.80681264914
Mean absolute percentage error or MAPE: 0.14443566085297666
```

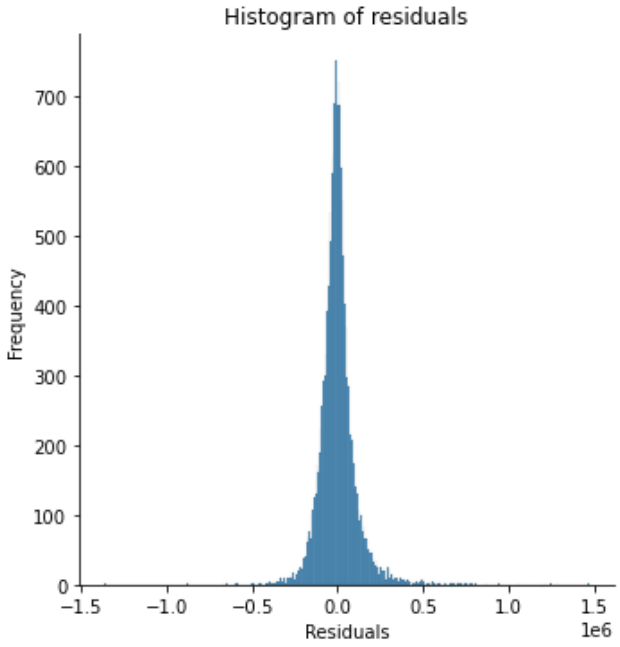
XGBoost tuned scaled:
Visualising the difference between actual and predicted prices



XGBoost tuned scaled: Inspecting residuals



XGBoost tubed scaled: Checking normality of errors



XGBoost tuned scaled: Model evaluation (test)

```
r^2: 0.8161917488058963
Adjusted r^2: 0.8151940303208166
Root mean squared error or RMSE: 163304.45776926583
Mean absolute percentage error or MAPE: 0.17030197582140608
```

Ridge scaled

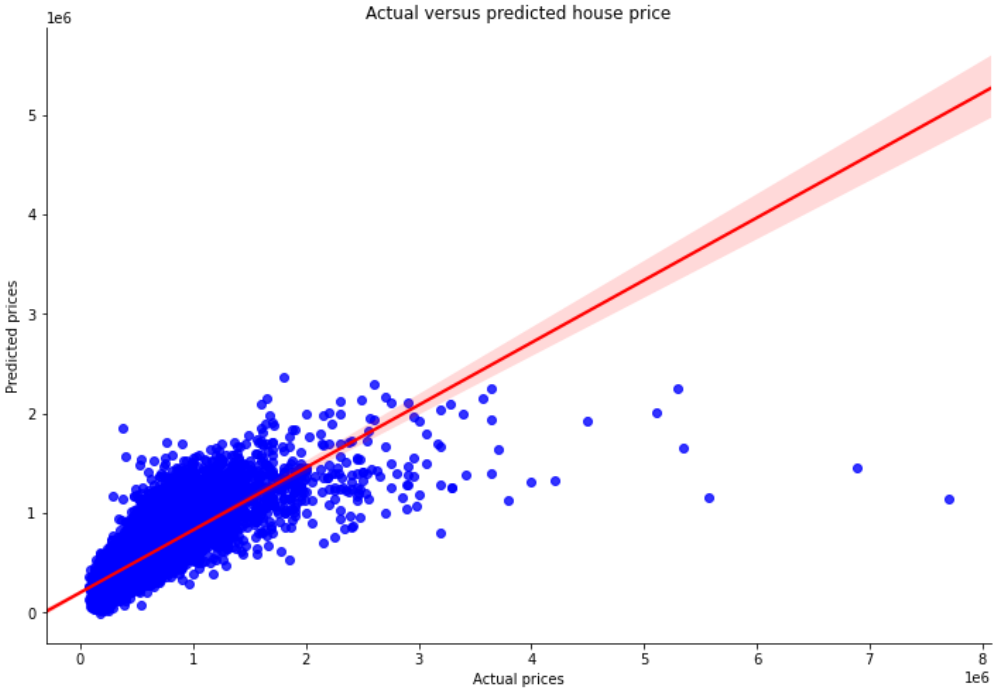
Create and train

```
Ridge(alpha=0.3)
```

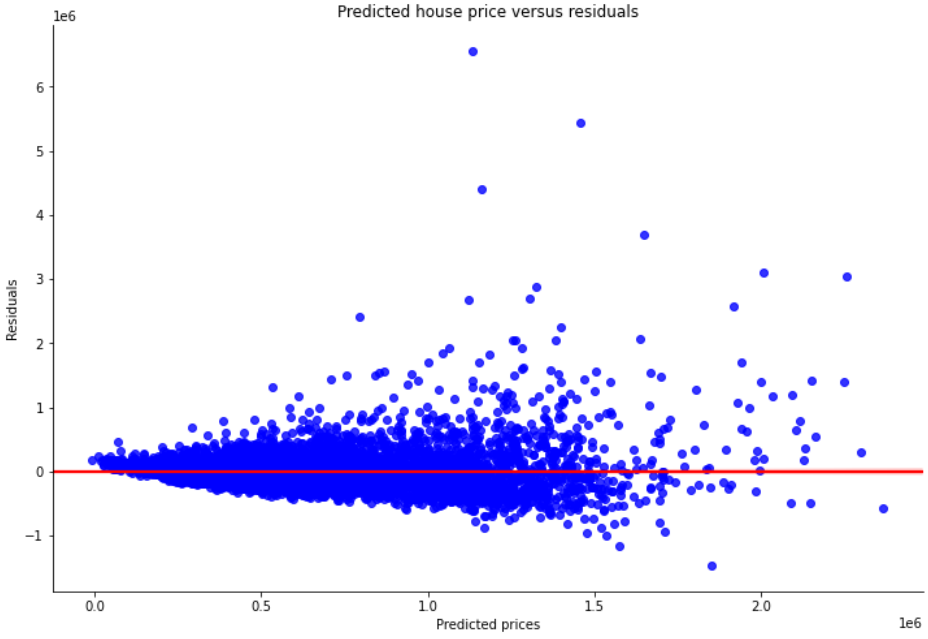
Ridge scaled: Model evaluation (train)

```
r^2: 0.627352877411266
Adjusted r^2: 0.6269089562857466
Root mean squared error or RMSE: 220610.40394682676
Mean absolute percentage error or MAPE: 0.24433931695632244
```

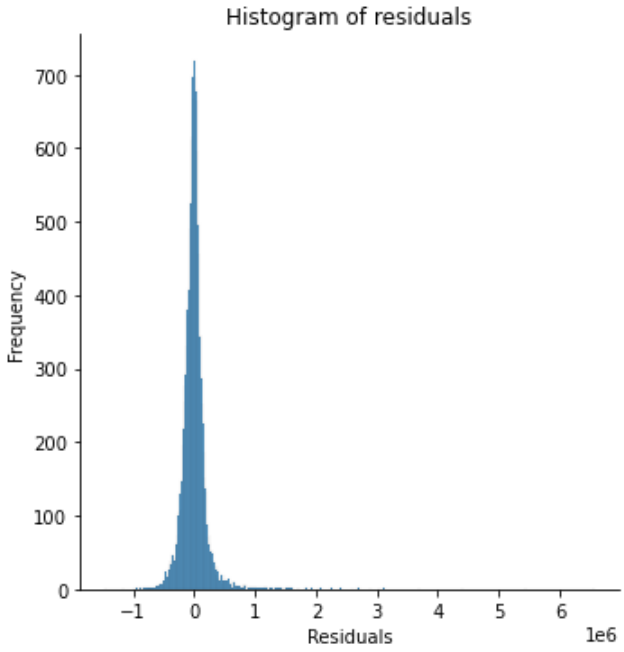
Ridge scaled:
Visualising the difference between actual and predicted house prices



Ridge scaled: Checking residuals¶



Ridge scaled: Checking normality of errors



Ridge scaled: Model evaluation (test)

```
r^2: 0.639858303015848
Adjusted r^2: 0.6388555883142679
Root mean squared error or RMSE: 228587.37570153383
Mean absolute percentage error or MAPE: 0.24802930953676197
```

Lasso scaled

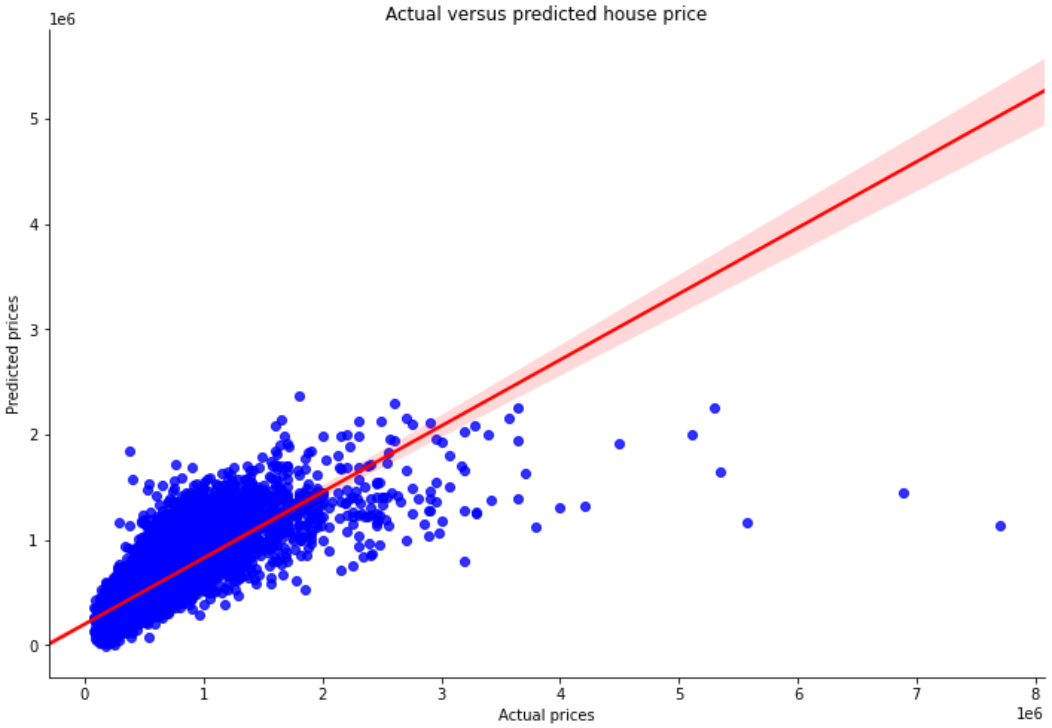
Create and train

```
Lasso(alpha=0.1)
```

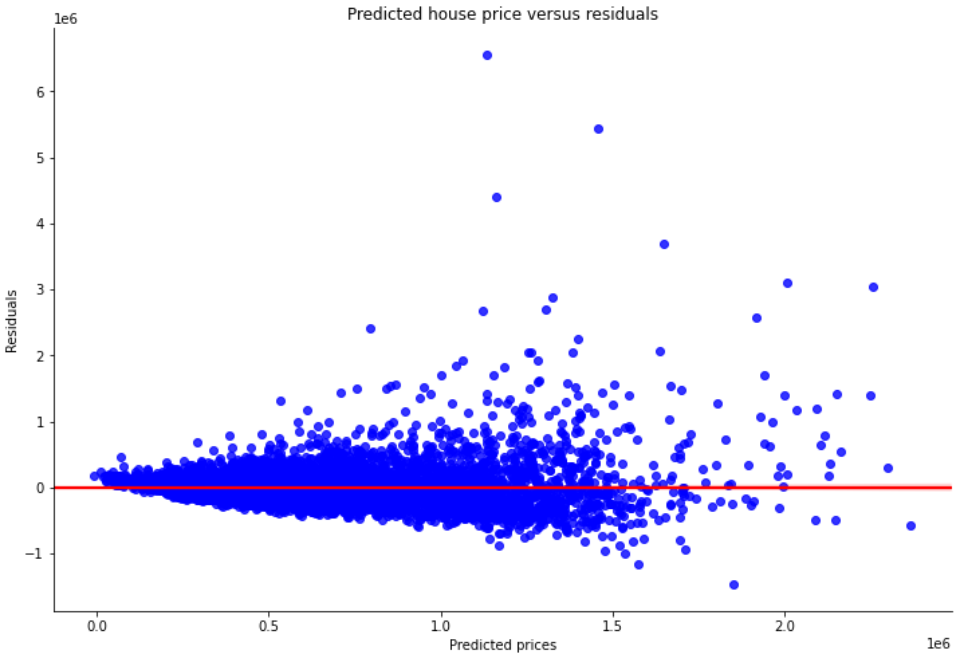
Lasso scaled: Model evaluation (train)

```
r^2: 0.6273528776339805
Adjusted r^2: 0.6269089565087265
Root mean squared error or RMSE: 220610.40388090227
Mean absolute percentage error or MAPE: 0.24434101143333417
```

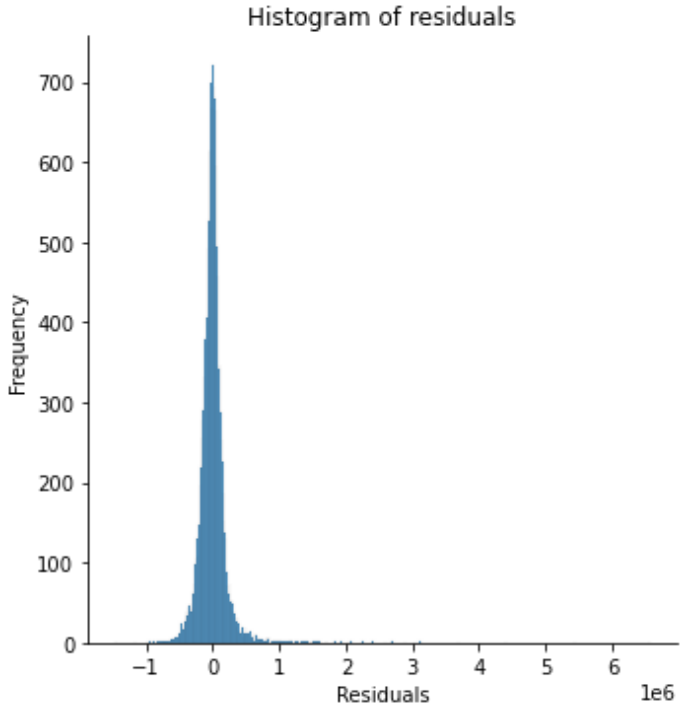
Lasso scaled:
Visualising the difference between actual and predicted house prices



Lasso scaled: Checking residuals



Lasso scaled: Checking normality of errors



Lasso scaled: Model evaluation (test)

```
r^2: 0.6398587258915132
Adjusted r^2: 0.6388560123673133
Root mean squared error or RMSE: 228587.24149870794
Mean absolute percentage error or MAPE: 0.24803098275369945
```

Support vector scaled

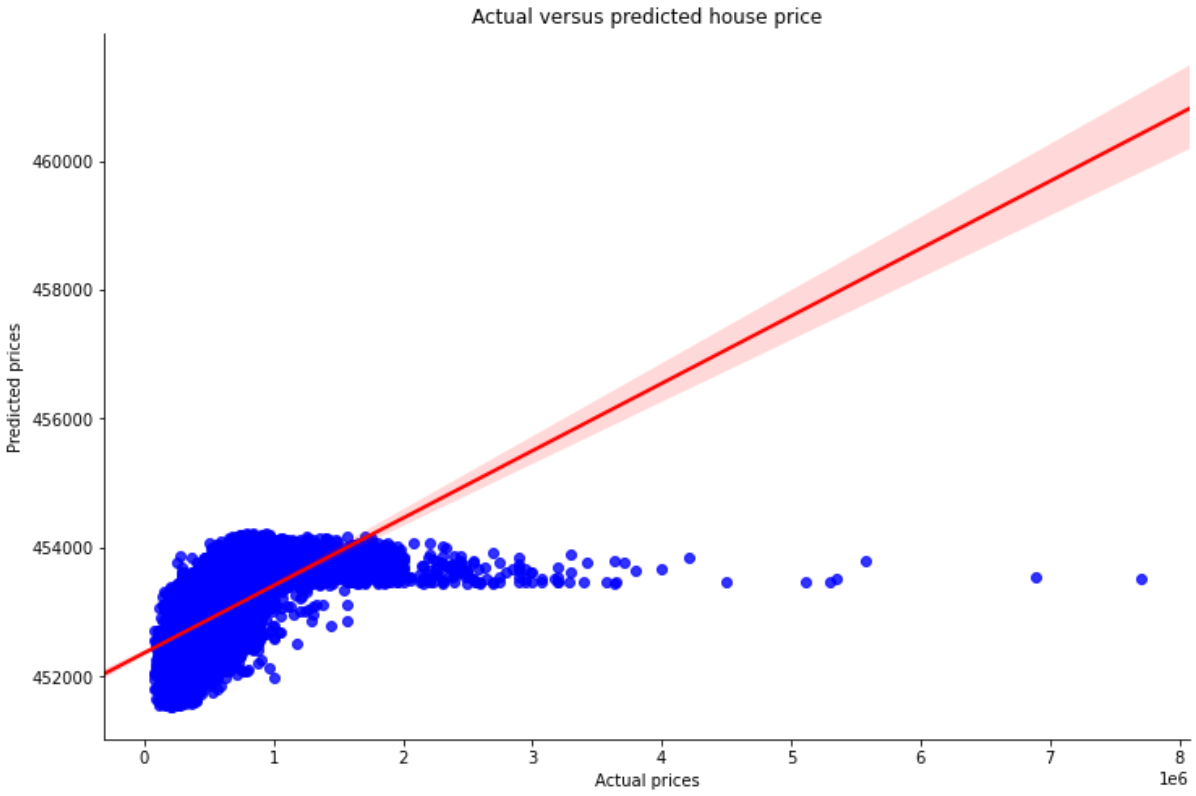
Create and train

```
SVR()
```

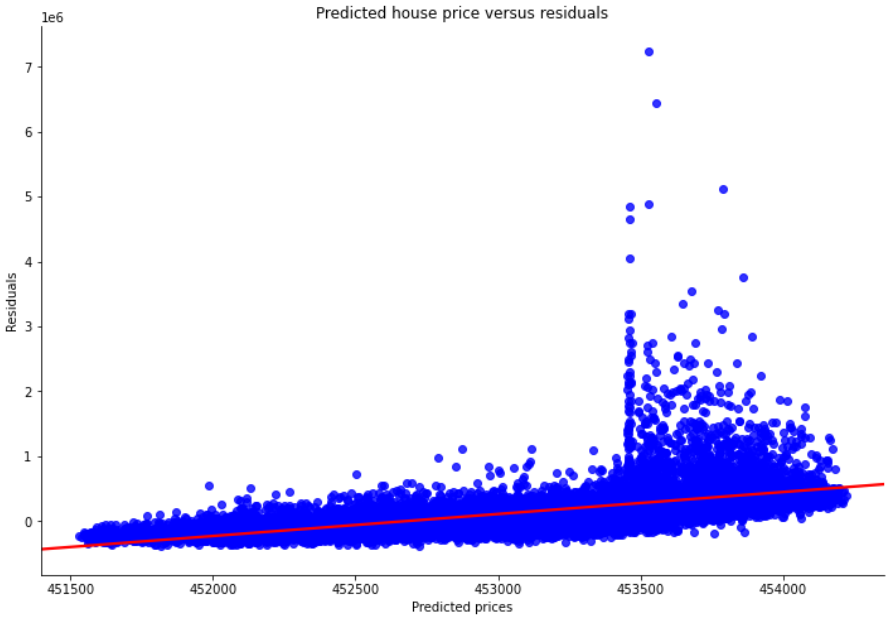
Support vector scaled: Model evaluation (train)

```
r^2: -0.05608353617401507
Adjusted r^2: -0.0573416105387492
Root mean squared error or RMSE: 371386.6085257349
Mean absolute percentage error or MAPE: 0.42425141945516726
```

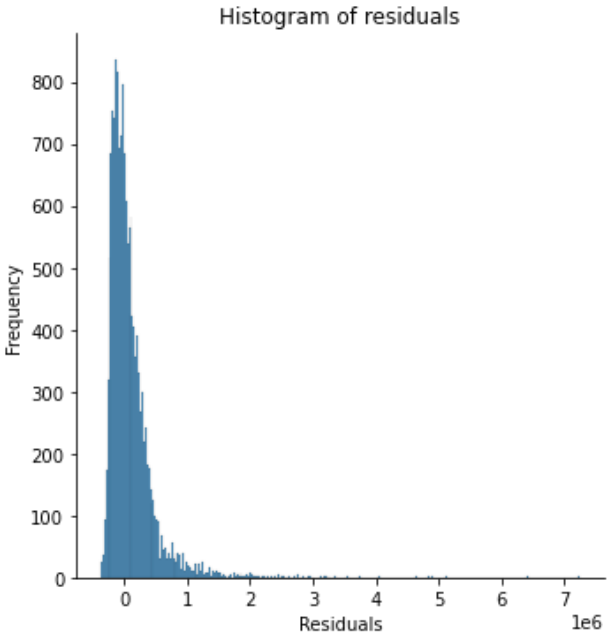
Support vector scaled:
Visualising the difference between actual and predicted house prices



Support vector scaled: Checking residuals



Support vector scaled: Checking normality of errors



Normality not satisfied. A failed model

Support vector scaled: Model evaluation (test data)

```
r^2: -0.05078395605228225
Adjusted r^2: -0.05370957263525855
Root mean squared error or RMSE: 390456.1385633087
Mean absolute percentage error or MAPE: 0.43304212525276164
```

KNN scaled

Create and train

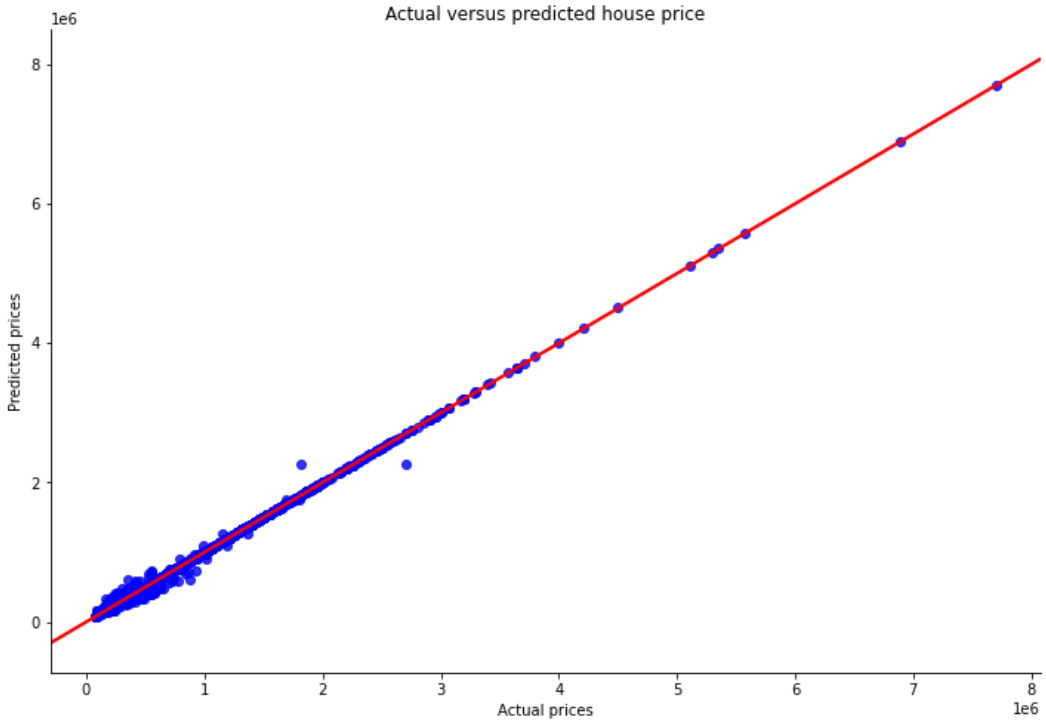
```
KNeighborsRegressor(n_neighbors=4, weights='distance')
```

KNN scaled: Model evaluation (train)

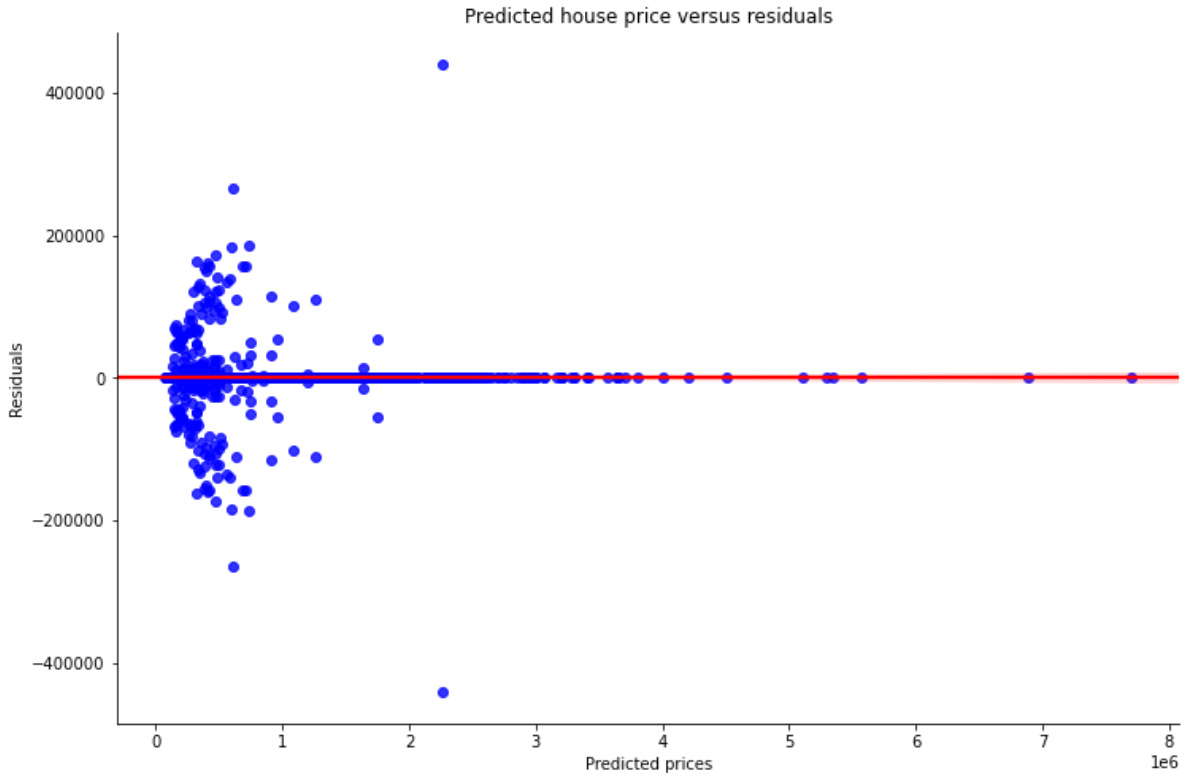
```
r^2: 0.9990145653532138
Adjusted r^2: 0.9990133914403321
Root mean squared error or RMSE: 11344.64779914104
Mean absolute percentage error or MAPE: 0.002850841259216476
```

KNN Scaled:

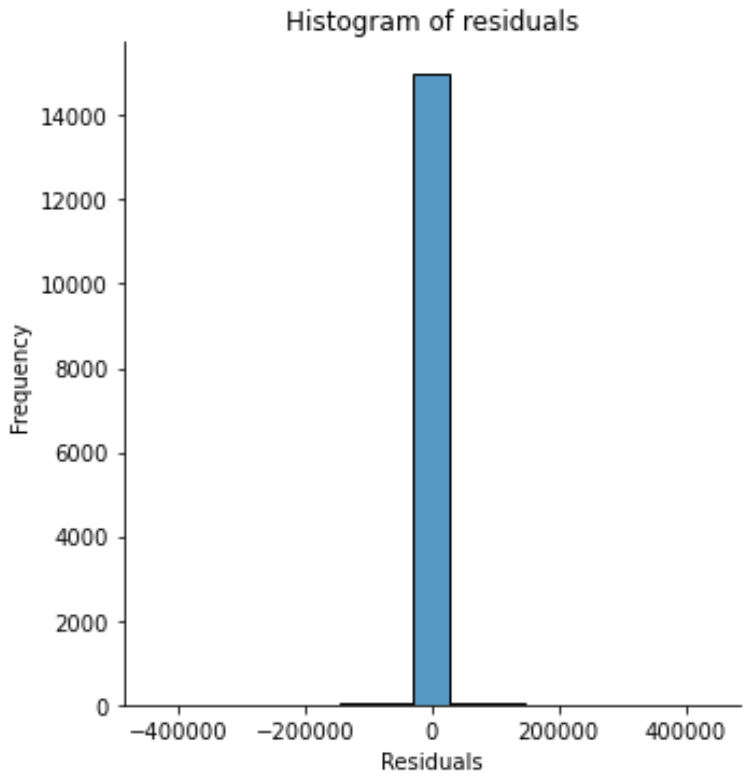
Visualising the difference between actual and predicted house prices



KNN scaled: Checking residuals



KNN scaled: Checking normality of errors



KNN scaled: Model evaluation (test data)

```
r^2: 0.6792358178376651
Adjusted r^2: 0.6783427389082108
Root mean squared error or RMSE: 215728.97104624554
Mean absolute percentage error or MAPE: 0.20949292954113602
```

Elastic Net scaled

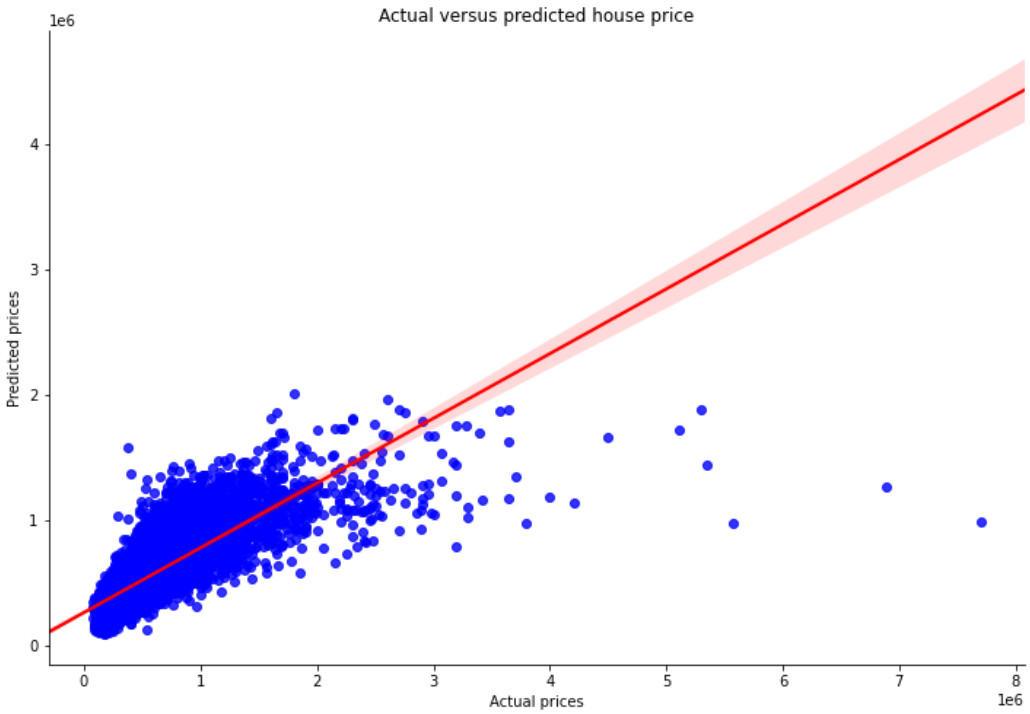
Create and train

```
ElasticNet()
```

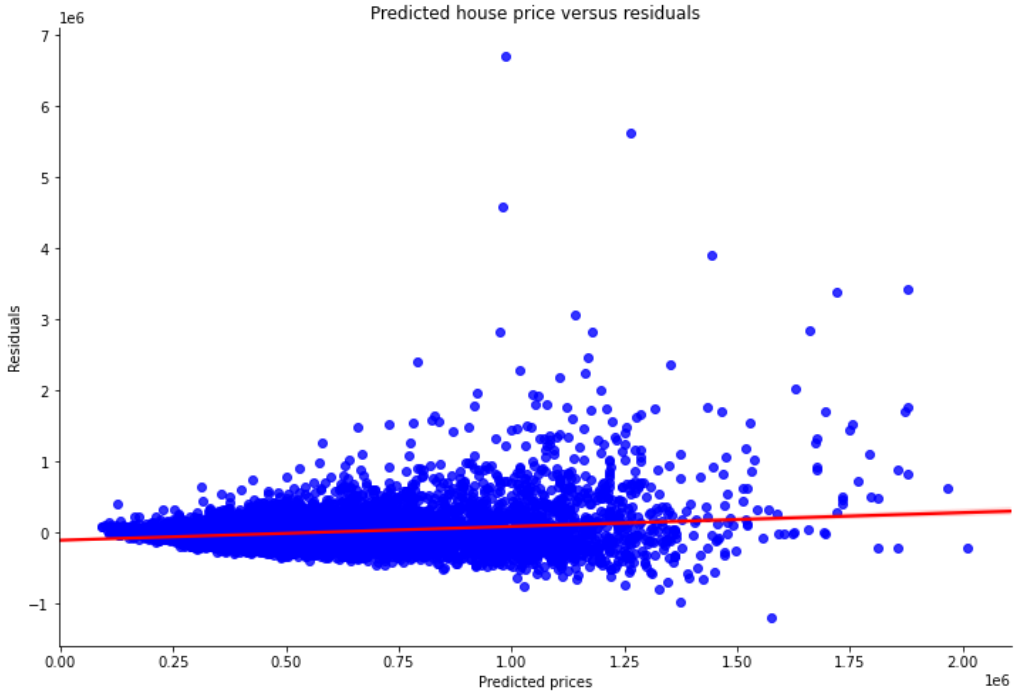
Elastic Net scaled: Model evaluation (train data)

```
r^2: 0.6001734652347371
Adjusted r^2: 0.5996971662522239
Root mean squared error or RMSE: 228514.05146624992
Mean absolute percentage error or MAPE: 0.244319728832206
```

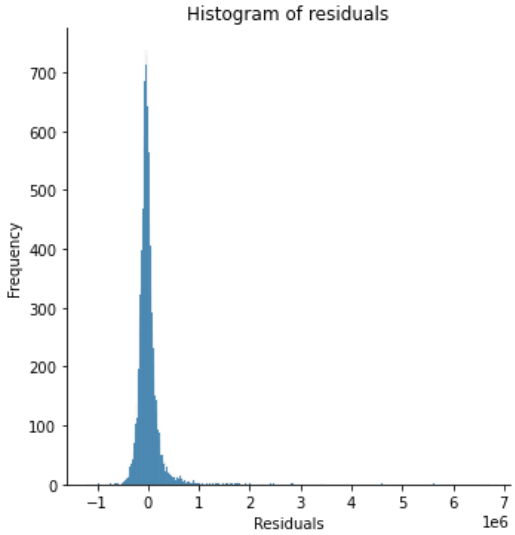
Elastic Net scaled:
Visualising the difference between actual and predicted house prices



Elastic Net scaled: Checking residuals



Elastic Net scaled: Checking normality of errors



Elastic Net scaled: Model evaluation (test)

```
r^2: 0.6041035022634971
Adjusted r^2: 0.6030012382326762
Root mean squared error or RMSE: 239665.967291215
Mean absolute percentage error or MAPE: 0.24893063268187854
```

Hypertuning with Gridsearch CV

Since we have best performance from XGB0oost models but most robust performance from the gradient boosting model, we will first hyper-tune the gradient boosting model for improving the score.

Following are the parameters we tune for the gradient boosting model: -

Loss	= ls, lad, huber
Bootstrap	= True, False
Max_depth	= range (5,11,1)
Max_features	= auto, sqrt
Learning_rate	= 0.05, 0.1, 0.2, 0.25,
Min_samples_leaf	= 4, 10, 20
Min_samples_split	= 5, 10, 1,000,
n_estimators	= 10, 50, 100, 150, 200,
Subsample	= 0.8, 1

GradientBoostingRegressor(random_state=22)

About hyper-tuning parameters

max_depth: The maximum depth per tree. A deeper tree might increase the performance, but also the complexity and chances to overfit.
The value must be an integer greater than 0. Default is 6.

learning_rate: The learning rate determines the step size at each iteration while your model optimizes toward its objective. A low learning rate makes computation slower, and requires more rounds to achieve the same reduction in residual error as a model with a high learning rate. But it optimizes the chances to reach the best optimum.
The value must be between 0 and 1. Default is 0.3.

n_estimators: The number of trees in our ensemble. Equivalent to the number of boosting rounds.
The value must be an integer greater than 0. Default is 100.
NB: In the standard library, this is referred as num_boost_round.
colsample_bytree: Represents the fraction of columns to be randomly sampled for each tree. It might improve overfitting.
The value must be between 0 and 1. Default is 1.

subsample: Represents the fraction of observations to be sampled for each tree. A lower values prevent overfitting but might lead to under-fitting.
The value must be between 0 and 1. Default is 1.

Regularization parameters:

alpha (reg_alpha): L1 regularization on the weights (Lasso Regression). When working with a large number of features, it might improve speed performances. It can be any integer. Default is 0.

lambda (reg_lambda): L2 regularization on the weights (Ridge Regression). It might help to reduce overfitting. It can be any integer. Default is 1.

gamma: Gamma is a pseudo-regularisation parameter (Lagrangian multiplier), and depends on the other parameters. The higher Gamma is, the higher the regularization. It can be any integer. Default is 0.

Grid search 1

First will tune each parameter separately

n_estimators = range (50, 401, 50)

```
Fitting 3 folds for each of 8 candidates, totalling 24 fits
{'n_estimators': 400}
```

Best parameters and best score

```
({'n_estimators': 400}, 0.7926021137891736)
```

n_estimators of 400 is best in range 50 to 400. Will test same until 1000

Grid search 2

```
Fitting 3 folds for each of 4 candidates, totalling 12 fits
GridSearchCV(cv=3, estimator=GradientBoostingRegressor(random_state=22),
             n_jobs=2, param_grid={'n_estimators': range(400, 1001, 200)},
             verbose=1)
({'mean_fit_time': array([ 4.44222633,  6.82679367,  9.22287838, 11.70107849]),
  'std_fit_time': array([0.04498105, 0.07504604, 0.04808188, 0.18023442]),
  'mean_score_time': array([0.02100094, 0.03311102, 0.04004836, 0.04432591]),
  'std_score_time': array([0.00081644, 0.00081867, 0.00143496, 0.00125031]),
  'param_n_estimators': masked_array(data=[400, 600, 800, 1000],
               mask=[False, False, False, False],
         fill_value='?',
              dtype=object),
  'params': [{'n_estimators': 400},
   {'n_estimators': 600},
   {'n_estimators': 800},
   {'n_estimators': 1000}],
  'split0_test_score': array([0.78354239, 0.78332145, 0.78256757, 0.78225603]),
  'split1_test_score': array([0.7950637 , 0.79485951, 0.79358607, 0.79160006]),
  'split2_test_score': array([0.79920025, 0.79860088, 0.79786431, 0.79640088]),
  'mean_test_score': array([0.79260211, 0.79226061, 0.79133932, 0.79008566]),
  'std_test_score': array([0.00662504, 0.00650287, 0.00644378, 0.00587306]),
  'rank_test_score': array([1, 2, 3, 4])},
 {'n_estimators': 400},
 0.7926021137891736)
```

```
Fitting 5 folds for each of 4 candidates, totalling 20 fits

GridSearchCV(cv=5, estimator=GradientBoostingRegressor(random_state=22),
             n_jobs=3, param_grid={'n_estimators': range(1000, 2000, 300)},
             verbose=1)
```

```
({'n_estimators': 1000}, 0.7989234256247523)
```

n_estimators of 1000 has given the best result in the range 400 to 1000

Grid search 3

```
Fitting 5 folds for each of 36 candidates, totalling 180 fits

GridSearchCV(cv=5, estimator=GradientBoostingRegressor(random_state=22),
             n_jobs=3,
             param_grid={'learning_rate': [0.1, 0.2],
                         'min_samples_leaf': [5, 10, 20],
                         'min_samples_split': [5, 10, 20],
                         'n_estimators': [500, 1000]},
             verbose=1)
```

```
({'learning_rate': 0.1,
  'min_samples_leaf': 5,
  'min_samples_split': 5,
  'n_estimators': 500},
 0.7996279352013775)
```

The four above parameters in combination have given us the best result, and we see that n_estimators of 1000 is the best again. Now, we will change the ranges of other three parameters.

Grid search 4

```
Fitting 5 folds for each of 16 candidates, totalling 80 fits

GridSearchCV(cv=5, estimator=GradientBoostingRegressor(random_state=22),
             n_jobs=3,
             param_grid={'learning_rate': [0.1, 0.15], 'max_depth': [5, 10],
                         'min_samples_leaf': [5, 8],
                         'min_samples_split': [20, 30],
                         'n_estimators': [1000]},
             verbose=1)
```

```
({'learning_rate': 0.1,
  'max_depth': 5,
  'min_samples_leaf': 5,
  'min_samples_split': 30,
  'n_estimators': 1000},
 0.7914017051966626)
```

The score has reduced from the earlier run.

Grid search 5

```
Fitting 5 folds for each of 4 candidates, totalling 20 fits

GridSearchCV(cv=5, estimator=GradientBoostingRegressor(random_state=22),
             n_jobs=2,
             param_grid={'learning_rate': [0.1], 'max_depth': [5],
                         'min_samples_leaf': [8, 10],
                         'min_samples_split': [30, 40],
                         'n_estimators': [1000]},
             verbose=1)

({'learning_rate': 0.1,
  'max_depth': 5,
  'min_samples_leaf': 10,
  'min_samples_split': 40,
  'n_estimators': 1000},
 0.7892094409903724)
```

Score has reduced from even the previous run.

Grid search 6

```
Fitting 5 folds for each of 2 candidates, totalling 10 fits

GridSearchCV(cv=5, estimator=GradientBoostingRegressor(random_state=22),
             n_jobs=2,
             param_grid={'learning_rate': [0.1], 'max_depth': [5],
                         'min_samples_leaf': [8], 'min_samples_split': [40, 50],
                         'n_estimators': [1000]},
             verbose=1)
```

```
({'learning_rate': 0.1,
  'max_depth': 5,
  'min_samples_leaf': 8,
  'min_samples_split': 50,
  'n_estimators': 1000},
 0.7861773409172611)
```

Score has dropped further. The best score is at min_samples_split of 5 among 5,10,20, so we will tune the final set of parameters along those finalised lines.

Grid search 7

```
Fitting 5 folds for each of 12 candidates, totalling 60 fits

GridSearchCV(cv=5, estimator=GradientBoostingRegressor(random_state=22),
             n_jobs=2,
             param_grid={'learning_rate': [0.1], 'loss': ['ls', 'lad', 'huber'],
                         'max_depth': [5], 'max_features': ['auto', 'sqrt'],
                         'min_samples_leaf': [5], 'min_samples_split': [5],
                         'n_estimators': [500], 'subsample': [0.8, 1]},
             verbose=1)
```

```
({'learning_rate': 0.1,
  'loss': 'huber',
  'max_depth': 5,
  'max_features': 'auto',
  'min_samples_leaf': 5,
  'min_samples_split': 5,
  'n_estimators': 500,
  'subsample': 0.8},
 0.8058864269240585)
```

Our best score has improved, so will try one more iteration by changing other parameters.

Grid search 8

```
Fitting 5 folds for each of 4 candidates, totalling 20 fits

({'learning_rate': 0.1,
  'loss': 'huber',
  'max_depth': 5,
  'max_features': 'sqrt',
  'min_samples_leaf': 5,
  'min_samples_split': 5,
  'n_estimators': 1000,
  'subsample': 1},
 0.807302321993196)
```

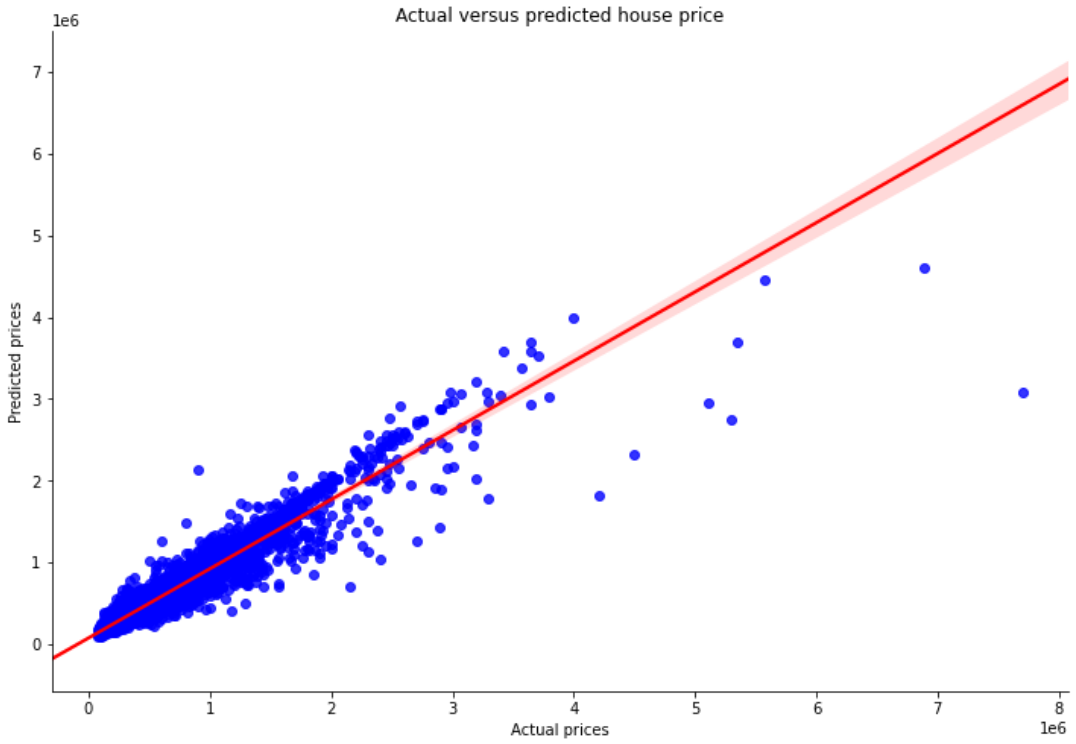
There is an improvement in the score. The final parameters that have given us the best result on training set are loss as huber, max_features as sqrt, learning_rate as 0.1, max_depth as 5, min_samples_leaf as 5, min_samples_split as 5, 'n_estimators as 1000, and subsample as 1.

```
GradientBoostingRegressor(loss='huber', max_depth=5, min_samples_leaf=5,
                          min_samples_split=5, n_estimators=1000,
                          random_state=22)
```

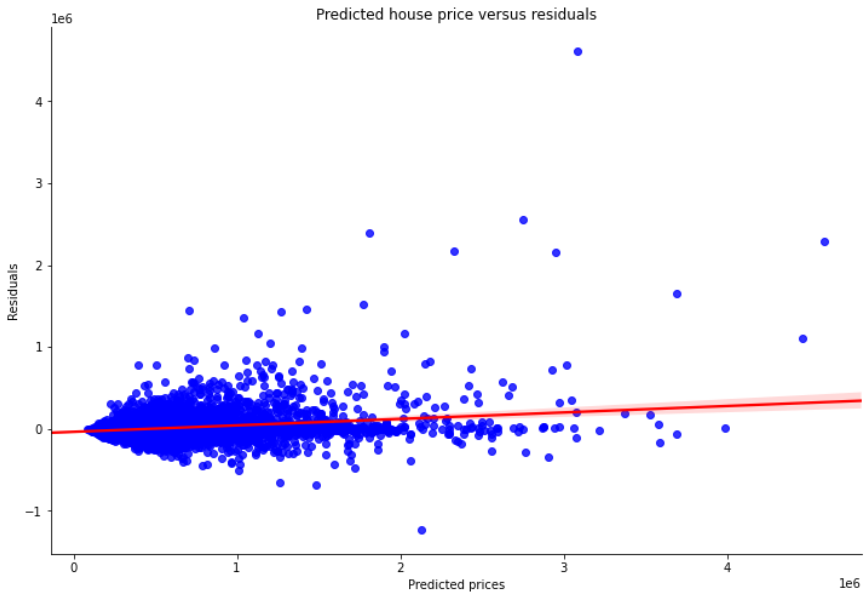
Grid search CV (best parameter): Model evaluation (train)

```
r^2: 0.9081392608293641
Adjusted r^2: 0.9080298304319405
Root mean squared error or RMSE: 109532.26579338525
Mean absolute percentage error or MAPE: 0.10577620590237073
```

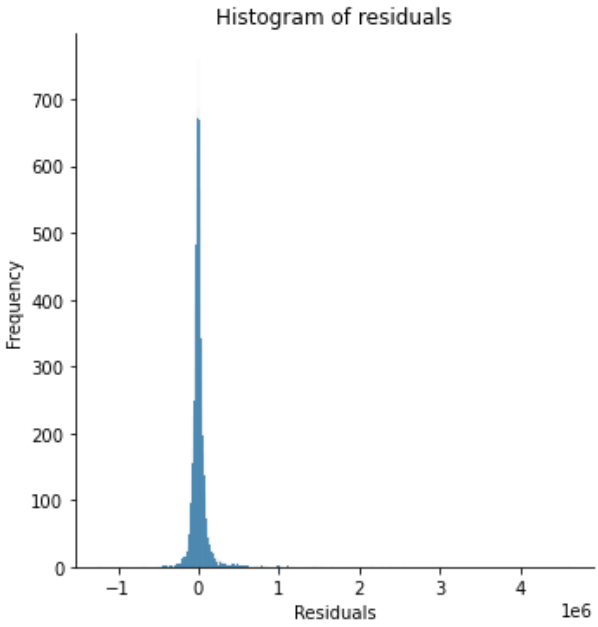
Grid-search CV:
Visualising the difference between actual and predicted house prices



Grid search: Checking residuals



Grid search: Checking normality of errors



Grid search scaled: Model evaluation (test)

```
r^2: 0.8076307949147785
Adjusted r^2: 0.8070951961999241
Root mean squared error or RMSE: 167064.16854644497
Mean absolute percentage error or MAPE: 0.16795887512524255
```

Similarly, we'll try another iterative process to improve the score of the XGBoost model

Regularization XGBoost using GridSearchCV - 1st Iteration

```
Fitting 5 folds for each of 216 candidates, totalling 1080 fits

GridSearchCV(cv=KFold(n_splits=5, random_state=None, shuffle=False),
             estimator=XGBRegressor(base_score=None, booster='gbtree',
                                    colsample_bylevel=None,
                                    colsample_bynode=None,
                                    colsample_bytree=None, gamma=None,
                                    gpu_id=None, importance_type='gain',
                                    interaction_constraints=None,
                                    learning_rate=None, max_delta_step=None,
                                    max_depth=None, min_child_weight=None,
                                    missing=na...
                                    num_parallel_tree=None, random_state=None,
                                    reg_alpha=None, reg_lambda=None,
                                    scale_pos_weight=None, subsample=None,
                                    tree_method=None, validate_parameters=None,
                                    verbosity=None),
             n_jobs=2,
             param_grid={'colsample_bytree': [0.66, 0.68, 0.7, 0.72],
                         'learning_rate': [0.2, 0.22, 0.24],
                         'n_estimators': [185, 188, 1],
                         'subsample': [0.62, 0.63, 0.64, 0.65, 0.66, 0.67]},
             return_train_score=True, verbose=1)
```

```
{'colsample_bytree': 0.7,
 'learning_rate': 0.22,
 'n_estimators': 185,
 'subsample': 0.65}
```

Regularization using GridSearchCV - 2nd Iteration

```
Fitting 5 folds for each of 15 candidates, totalling 75 fits

{'max_depth': 4, 'min_child_weight': 6}
```

Regularization using GridSearchCV - 3rd Iteration

```
Fitting 5 folds for each of 5 candidates, totalling 25 fits

{'gamma': 50.0}
```

Using last iteration to build a model

```
XGBRegressor(base_score=None, booster=None, colsample_bylevel=None,
             colsample_bynode=None, colsample_bytree=0.7, gamma=50, gpu_id=None,
             importance_type='gain', interaction_constraints=None,
             learning_rate=0.22, max_delta_step=None, max_depth=4,
             min_child_weight=6, missing=nan, monotone_constraints=None,
             n_estimators=186, n_jobs=None, num_parallel_tree=None,
             random_state=None, reg_alpha=None, reg_lambda=None,
             scale_pos_weight=None, subsample=0.65, tree_method=None,
             validate_parameters=None, verbosity=None)
```

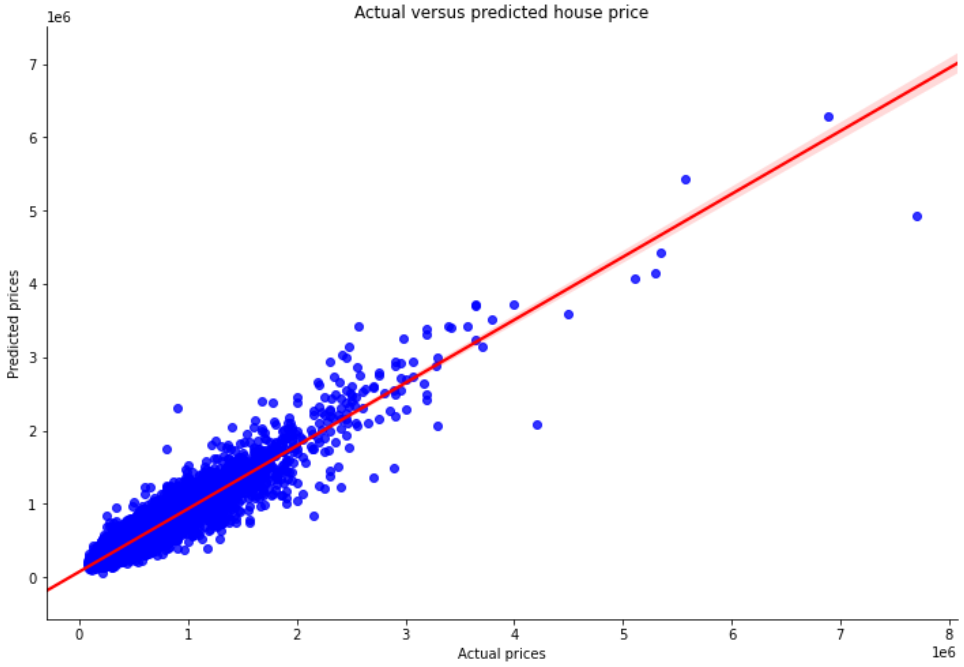
Fitting the model to scaled train data that has hot-encoded categorical features

```
XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
             colsample_bynode=1, colsample_bytree=0.7, gamma=50, gpu_id=-1,
             importance_type='gain', interaction_constraints='',
             learning_rate=0.22, max_delta_step=0, max_depth=4,
             min_child_weight=6, missing=nan, monotone_constraints='()',
             n_estimators=186, n_jobs=8, num_parallel_tree=1, random_state=0,
             reg_alpha=0, reg_lambda=1, scale_pos_weight=1, subsample=0.65,
             tree_method='exact', validate_parameters=1, verbosity=None)
```

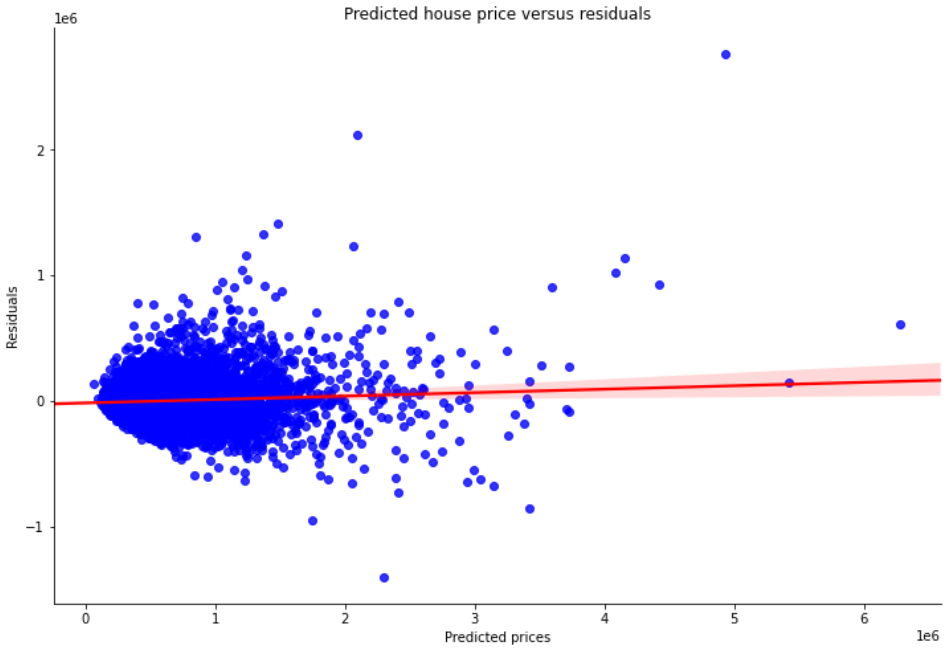
XGBoost grid: Model evaluation (train data)

```
r^2: 0.8811977560827012
Adjusted r^2: 0.8809222589292456
Root mean squared error or RMSE: 124563.1050442301
Mean absolute percentage error or MAPE: 0.16224134158917222
```

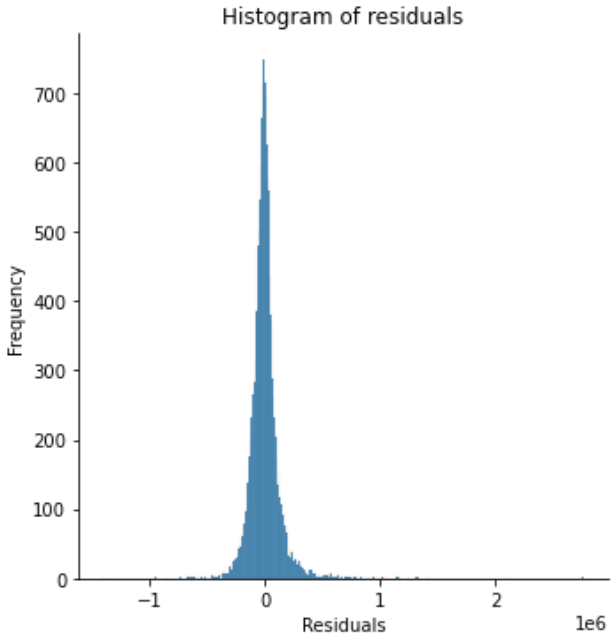
XGBoost grid: Visualising the difference between actual and predicted prices



XGBoost grid: Inspecting residuals



XGBoost grid: Checking normality of errors



XGBoost: Model evaluation (test data)

```
r^2: 0.808492698094226
Adjusted r^2: 0.8074531888562139
Root mean squared error or RMSE: 166689.48594245594
Mean absolute percentage error or MAPE: 0.18135939825251698
```

XG Boost grid2

```
XGBRegressor(base_score=None, booster=None, colsample_bylevel=None,
             colsample_bynode=None, colsample_bytree=0.7, gamma=None,
             gpu_id=None, importance_type='gain', interaction_constraints=None,
             learning_rate=0.22, max_delta_step=None, max_depth=4,
             min_child_weight=6, missing=nan, monotone_constraints=None,
             n_estimators=186, n_jobs=None, num_parallel_tree=None,
             random_state=None, reg_alpha=None, reg_lambda=None,
             scale_pos_weight=None, subsample=0.65, tree_method=None,
             validate_parameters=None, verbosity=None)
```

Model fitted on train data

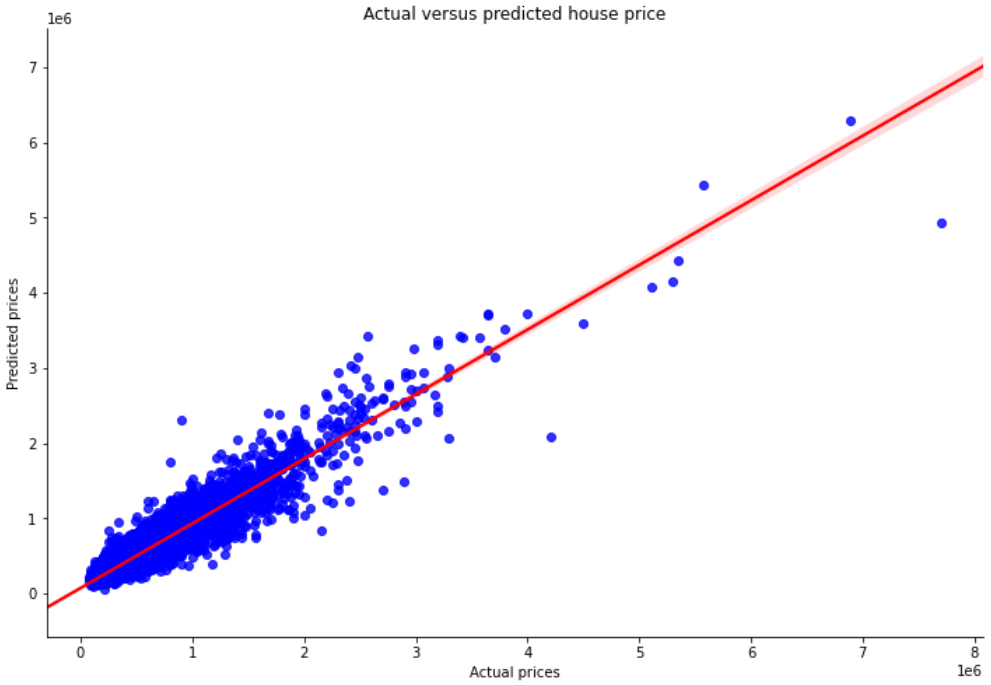
```
XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
             colsample_bynode=1, colsample_bytree=0.7, gamma=0, gpu_id=-1,
             importance_type='gain', interaction_constraints='',
             learning_rate=0.22, max_delta_step=0, max_depth=4,
             min_child_weight=6, missing=nan, monotone_constraints='()',
             n_estimators=186, n_jobs=8, num_parallel_tree=1, random_state=0,
             reg_alpha=0, reg_lambda=1, scale_pos_weight=1, subsample=0.65,
             tree_method='exact', validate_parameters=1, verbosity=None)
```

XGBoost grid 2: Model evaluation (train data)

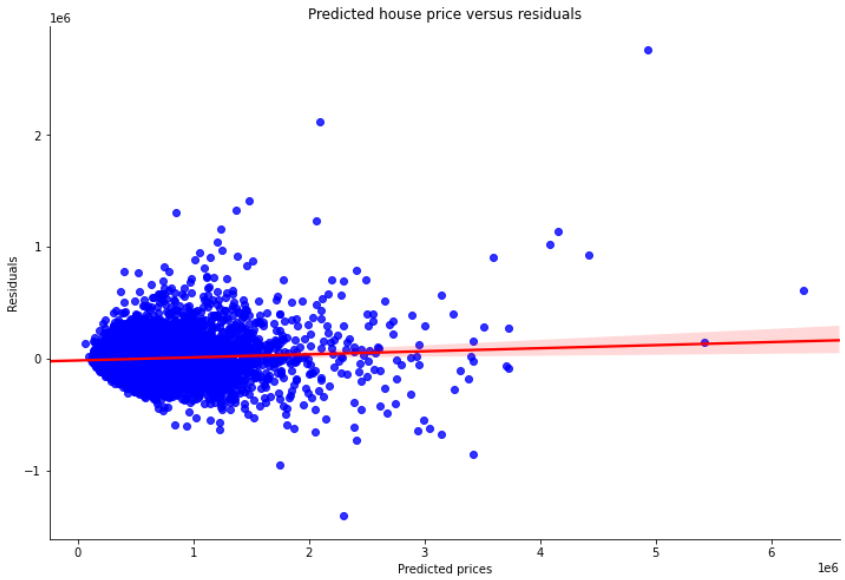
```
r^2: 0.8811977560827012
Adjusted r^2: 0.8809222589292456
Root mean squared error or RMSE: 124563.1050442301
Mean absolute percentage error or MAPE: 0.16224134158917222
```

XGBoost grid2:

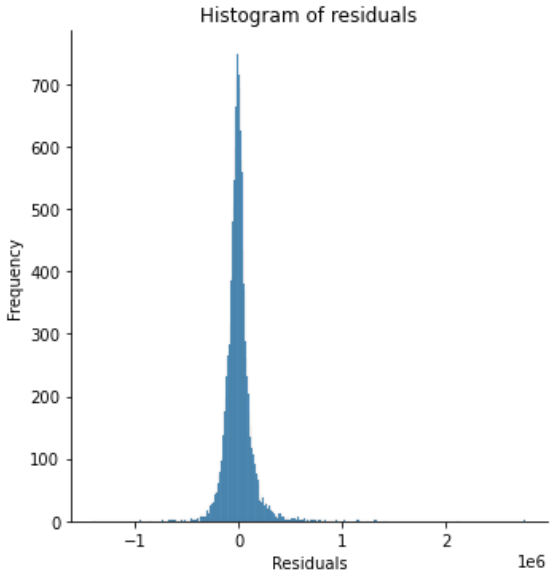
Visualising the difference between actual and predicted prices¶



XGBoost grid2: Inspecting residuals



XGBoost grid2: Checking normality of errors



XGBoost grid2: Model evaluation (test data)¶

```
r^2: 0.808492698094226
Adjusted r^2: 0.8074531888562139
Root mean squared error or RMSE: 166689.48594245594
Mean absolute percentage error or MAPE: 0.18135939825251698
```

XGBoost grid3

```
XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
             colsample_bynode=1, colsample_bytree=0.7, gamma=0, gpu_id=-1,
             importance_type='gain', interaction_constraints='',
             learning_rate=0.22, max_delta_step=0, max_depth=6,
             min_child_weight=1, missing=nan, monotone_constraints='()',
             n_estimators=185, n_jobs=8, num_parallel_tree=1, random_state=0,
             reg_alpha=0, reg_lambda=1, scale_pos_weight=1, subsample=0.65,
             tree_method='exact', validate_parameters=1, verbosity=None)
```

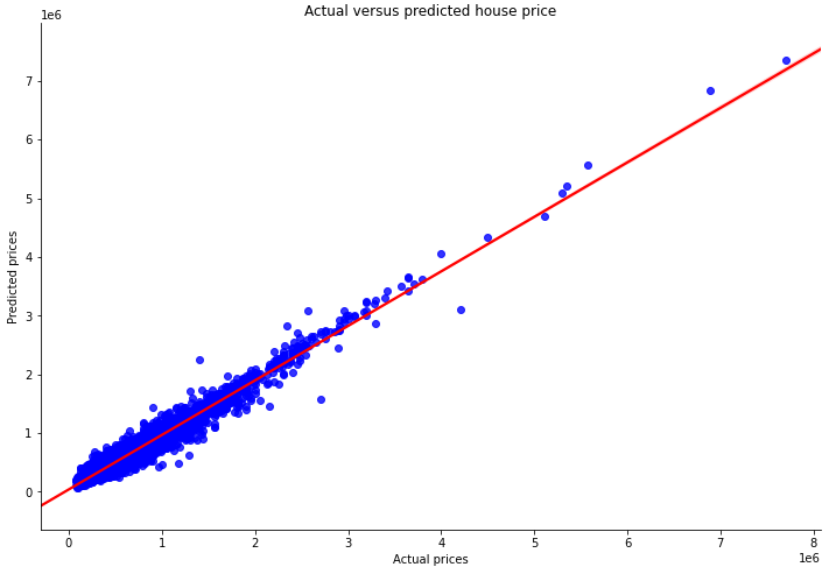
Fitting the model to train data

```
XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
             colsample_bynode=1, colsample_bytree=0.7, gamma=0, gpu_id=-1,
             importance_type='gain', interaction_constraints='',
             learning_rate=0.22, max_delta_step=0, max_depth=6,
             min_child_weight=1, missing=nan, monotone_constraints='()',
             n_estimators=185, n_jobs=8, num_parallel_tree=1, random_state=0,
             reg_alpha=0, reg_lambda=1, scale_pos_weight=1, subsample=0.65,
             tree_method='exact', validate_parameters=1, verbosity=None)
```

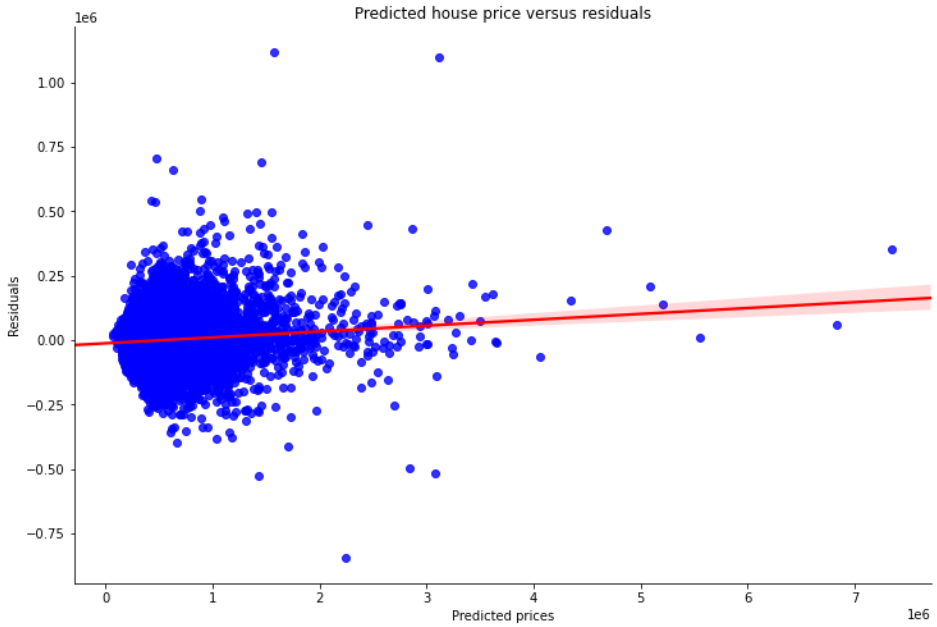
XGBoost grid 3: Model evaluation (train data)

```
r^2: 0.949477215516367
Adjusted r^2: 0.9493600554118863
Root mean squared error or RMSE: 81230.8023121788
Mean absolute percentage error or MAPE: 0.1246893977425414
```

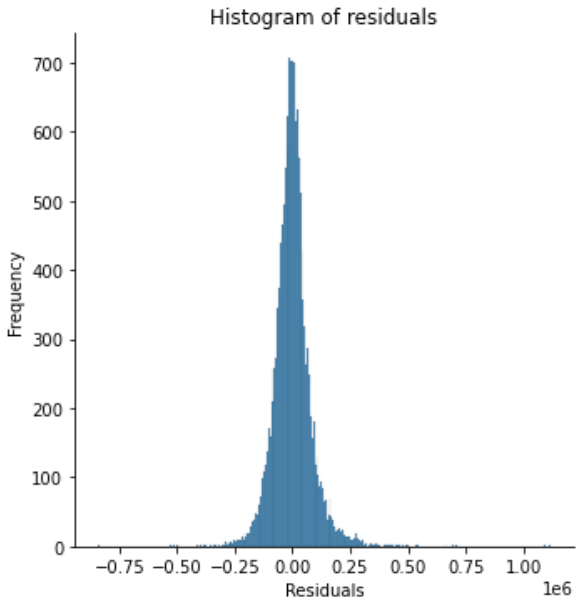
XGBoost grid3:
Visualising the difference between actual and predicted prices



XGBoost grid3: Inspecting residuals



XGBoost grid3: Checking normality of errors



XGBoost grid3: Model evaluation (test)

```
r^2: 0.797288954589753
Adjusted r^2: 0.796188630987185
Root mean squared error or RMSE: 171496.09878825894
Mean absolute percentage error or MAPE: 0.1774863221205069
```

Sorting models by r^2 test (highest first)

	Model	R-squared train	R-squared test	Adjusted r^2 train	Adjusted r^2 test	RMSE train	RMSE test	MAPE train	MAPE test
18	XGB tuned scaled	91.363735	81.619175	91.343708	81.519403	106203.806813	163304.457769	14.443566	17.030198
6	XGBoost tuned	91.377945	81.597213	91.367673	81.497322	106116.399784	163401.989883	14.431561	17.035787
26	XGB grid2	88.119776	80.849270	88.092226	80.745319	124563.105044	166689.485942	16.224134	18.135940
25	XGB grid1	88.119776	80.849270	88.092226	80.745319	124563.105044	166689.485942	16.224134	18.135940
2	XGBoost	94.046928	80.815587	94.039836	80.711453	88175.448577	166836.011069	12.933240	17.678436
14	XGB scaled	94.046928	80.808355	94.033123	80.704182	88175.448577	166867.453067	12.933240	17.677506
24	GBR grid search	90.813926	80.763079	90.802983	80.709520	109532.265793	167064.168546	10.577621	16.795888
13	RF scaled	96.857521	80.058821	96.853777	80.003300	64063.888981	170094.768361	6.630243	17.285611
1	Random forest	96.868390	79.783833	96.864659	79.727547	63953.005524	171263.550619	6.619319	17.235626
27	XGB grid3	94.947722	79.728895	94.936006	79.618863	81230.802312	171496.098788	12.468940	17.748632
4	Bagging	95.835299	78.675525	95.830338	78.616152	73751.188426	175895.495138	7.329439	18.107279
16	Bagging scaled	95.738372	77.682470	95.733295	77.620333	74604.476905	179944.512032	7.398759	18.144742
5	Gradient boost	78.748101	76.125344	78.722785	76.058872	166600.373692	186116.165598	20.453600	20.705725
17	GBR scaled	78.748101	76.119499	78.722785	76.053011	166600.373692	186138.946224	20.453600	20.706015
10	KNN	99.901419	69.425183	99.901302	69.340056	11346.809687	210618.963505	0.286007	20.979356
22	KNN scaled	99.901457	67.923582	99.901339	67.834274	11344.647799	215728.971046	0.285084	20.949293
22	KNN scaled	99.901457	67.923582	99.901339	67.834274	11344.647799	215728.971046	0.285084	20.949293
15	DT scaled	99.901457	66.919769	99.901339	66.827666	11344.647799	219078.523366	0.285083	23.772203
3	Decision tree	99.901457	65.355315	99.901339	65.258856	11344.647799	224199.091429	0.285083	23.911582
0	Linear regression	62.735288	63.985875	62.690896	63.885604	220610.403879	228587.232400	24.434118	24.803116
12	LR scaled	62.735288	63.985875	62.690896	63.885604	220610.403879	228587.232400	24.434118	24.803116
20	Lasso scaled	62.735288	63.985873	62.690896	63.885601	220610.403881	228587.241499	24.434101	24.803098
8	Lasso	62.735288	63.985872	62.690896	63.885601	220610.403880	228587.242060	24.434111	24.803109
7	Ridge	62.735288	63.985833	62.690896	63.885562	220610.403936	228587.365746	24.434026	24.803032
19	Ridge scaled	62.735288	63.985830	62.690896	63.885559	220610.403947	228587.375702	24.433932	24.802931
11	Elastic net	62.531538	63.663790	62.486903	63.562622	221212.691532	229607.120150	24.330047	24.743568
23	Elastic Net scaled	60.017347	60.410350	59.969717	60.300124	228514.051466	239665.967291	24.431973	24.893063
9	Support vector	-5.585952	-5.055077	-5.711733	-5.347573	371347.217633	390412.811279	42.459051	43.341094
21	SV scaled	-5.608354	-5.078396	-5.734161	-5.370957	371386.608526	390456.138563	42.425142	43.304213

Sorting models by MAPE test (lowest first)

	Model	R-squared train	R-squared test	Adjusted r^2 train	Adjusted r^2 test	RMSE train	RMSE test	MAPE train	MAPE test
24	GBR grid search	90.813926	80.763079	90.802983	80.709520	109532.265793	167064.168546	10.577621	16.795888
18	XGB tuned scaled	91.363735	81.619175	91.343708	81.519403	106203.806813	163304.457769	14.443566	17.030198
6	XGBoost tuned	91.377945	81.597213	91.367673	81.497322	106116.399784	163401.989883	14.431561	17.035787
1	Random forest	96.868390	79.783833	96.864659	79.727547	63953.005524	171263.550619	6.619319	17.235626
13	RF scaled	96.857521	80.058821	96.853777	80.003300	64063.888981	170094.768361	6.630243	17.285611
14	XGB scaled	94.046928	80.808355	94.033123	80.704182	88175.448577	166867.453067	12.933240	17.677506
2	XGBoost	94.046928	80.815587	94.039836	80.711453	88175.448577	166836.011069	12.933240	17.678436
27	XGB grid3	94.947722	79.728895	94.936006	79.618863	81230.802312	171496.098788	12.468940	17.748632
4	Bagging	95.835299	78.675525	95.830338	78.616152	73751.188426	175895.495138	7.329439	18.107279
25	XGB grid1	88.119776	80.849270	88.092226	80.745319	124563.105044	166689.485942	16.224134	18.135940
26	XGB grid2	88.119776	80.849270	88.092226	80.745319	124563.105044	166689.485942	16.224134	18.135940
16	Bagging scaled	95.738372	77.682470	95.733295	77.620333	74604.476905	179944.512032	7.398759	18.144742
5	Gradient boost	78.748101	76.125344	78.722785	76.058872	166600.373692	186116.165598	20.453600	20.705725
17	GBR scaled	78.748101	76.119499	78.722785	76.053011	166600.373692	186138.946224	20.453600	20.706015
22	KNN scaled	99.901457	67.923582	99.901339	67.834274	11344.647799	215728.971046	0.285084	20.949293
10	KNN	99.901419	69.425183	99.901302	69.340056	11346.809687	210618.963505	0.286007	20.979356

15	DT scaled	99.901457	66.919769	99.901339	66.827666	11344.647799	219078.523366	0.285083	23.772203
3	Decision tree	99.901457	65.355315	99.901339	65.258856	11344.647799	224199.091429	0.285083	23.911582
11	Elastic net	62.531538	63.663790	62.486903	63.562622	221212.691532	229607.120150	24.330047	24.743568
19	Ridge scaled	62.735288	63.985830	62.690896	63.885559	220610.403947	228587.375702	24.433932	24.802931
7	Ridge	62.735288	63.985833	62.690896	63.885562	220610.403936	228587.365746	24.434026	24.803032
20	Lasso scaled	62.735288	63.985873	62.690896	63.885601	220610.403881	228587.241499	24.434101	24.803098
8	Lasso	62.735288	63.985872	62.690896	63.885601	220610.403880	228587.242060	24.434111	24.803109
0	Linear regression	62.735288	63.985875	62.690896	63.885604	220610.403879	228587.232400	24.434118	24.803116
12	LR scaled	62.735288	63.985875	62.690896	63.885604	220610.403879	228587.232400	24.434118	24.803116
23	Elastic Net scaled	60.017347	60.410350	59.969717	60.300124	228514.051466	239665.967291	24.431973	24.893063
21	SV scaled	-5.608354	-5.078396	-5.734161	-5.370957	371386.608526	390456.138563	42.425142	43.304213
9	Support vector	-5.585952	-5.055077	-5.711733	-5.347573	371347.217633	390412.811279	42.459051	43.341094

For business use, **MAPE** is often preferred because apparently managers understand percentages better than squared errors.

R-squared is a goodness-of-fit measure for linear regression models. This statistic indicates the percentage of the variance in the dependent variable that the independent variables explain collectively. R-squared measures the strength of the relationship between your model and the dependent variable on a convenient 0 – 100% scale.

Sorting models by Adjusted r^2 test (highest first)

	Model	R-squared train	R-squared test	Adjusted r^2 train	Adjusted r^2 test	RMSE train	RMSE test	MAPE train	MAPE test
18	XGB tuned scaled	91.363735	81.619175	91.343708	81.519403	106203.806813	163304.457769	14.443566	17.030198
6	XGBoost tuned	91.377945	81.597213	91.367673	81.497322	106116.399784	163401.989883	14.431561	17.035787
26	XGB grid2	88.119776	80.849270	88.092226	80.745319	124563.105044	166689.485942	16.224134	18.135940
25	XGB grid1	88.119776	80.849270	88.092226	80.745319	124563.105044	166689.485942	16.224134	18.135940
2	XGBoost	94.046928	80.815587	94.039836	80.711453	88175.448577	166836.011069	12.933240	17.678436
24	GBR grid search	90.813926	80.763079	90.802983	80.709520	109532.265793	167064.168546	10.577621	16.795888
14	XGB scaled	94.046928	80.808355	94.033123	80.704182	88175.448577	166867.453067	12.933240	17.677506
13	RF scaled	96.857521	80.058821	96.853777	80.003300	64063.888981	170094.768361	6.630243	17.285611
1	Random forest	96.868390	79.783833	96.864659	79.727547	63953.005524	171263.550619	6.619319	17.235626
27	XGB grid3	94.947722	79.728895	94.936006	79.618863	81230.802312	171496.098788	12.468940	17.748632
4	Bagging	95.835299	78.675525	95.830338	78.616152	73751.188426	175895.495138	7.329439	18.107279
16	Bagging scaled	95.738372	77.682470	95.733295	77.620333	74604.476905	179944.512032	7.398759	18.144742
5	Gradient boost	78.748101	76.125344	78.722785	76.058872	166600.373692	186116.165598	20.453600	20.705725
17	GBR scaled	78.748101	76.119499	78.722785	76.053011	166600.373692	186138.946224	20.453600	20.706015
10	KNN	99.901419	69.425183	99.901302	69.340056	11346.809687	210618.963505	0.286007	20.979356
22	KNN scaled	99.901457	67.923582	99.901339	67.834274	11344.647799	215728.971046	0.285084	20.949293
15	DT scaled	99.901457	66.919769	99.901339	66.827666	11344.647799	219078.523366	0.285083	23.772203
3	Decision tree	99.901457	65.355315	99.901339	65.258856	11344.647799	224199.091429	0.285083	23.911582
0	Linear regression	62.735288	63.985875	62.690896	63.885604	220610.403879	228587.232400	24.434118	24.803116
12	LR scaled	62.735288	63.985875	62.690896	63.885604	220610.403879	228587.232400	24.434118	24.803116
20	Lasso scaled	62.735288	63.985873	62.690896	63.885601	220610.403881	228587.241499	24.434101	24.803098
8	Lasso	62.735288	63.985872	62.690896	63.885601	220610.403880	228587.242060	24.434111	24.803109
7	Ridge	62.735288	63.985833	62.690896	63.885562	220610.403936	228587.365746	24.434026	24.803032
19	Ridge scaled	62.735288	63.985830	62.690896	63.885559	220610.403947	228587.375702	24.433932	24.802931
11	Elastic net	62.531538	63.663790	62.486903	63.562622	221212.691532	229607.120150	24.330047	24.743568
23	Elastic Net scaled	60.017347	60.410350	59.969717	60.300124	228514.051466	239665.967291	24.431973	24.893063
9	Support vector	-5.585952	-5.055077	-5.711733	-5.347573	371347.217633	390412.811279	42.459051	43.341094
21	SV scaled	-5.608354	-5.078396	-5.734161	-5.370957	371386.608526	390456.138563	42.425142	43.304213

Ridge model with polynomial features

We performed a second-order polynomial transform on both the training data and testing data. We created and fitted a Ridge regression object using the training data, set the regularisation parameter to 0.1, and calculated the R^2 utilising the test data.¶

```
The R^2 Score Value for the training data is :0.7319120267024092
The R^2 Score Value for the testing data is :0.7346131083639942
```

Neural Network model

Zipcode

Categorical versus numerical

Zipcode equates to a specific area in King County, so we should consider this a categorical feature and create dummies or else group them into categories. This is a lot of numbers to create dummy variables, so we can potentially group them using external information

- Found a good website called zipdatamaps, which labels the zipcodes by area

	ZIP Code	ZIP Code Name	Population	ZIP Type
0	98001.000000	Auburn	31911.000000	Non-Unique
1	98002.000000	Auburn	31647.000000	Non-Unique
2	98003.000000	Federal Way	44151.000000	Non-Unique
3	98004.000000	Bellevue	27946.000000	Non-Unique
4	98005.000000	Bellevue	17714.000000	Non-Unique

Checking for null values

```
ZIP Code          0
ZIP Code Name     0
Population        0
ZIP Type          0
dtype: int64
```

How the data looks

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 21613 entries, 0 to 21612
Data columns (total 24 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   cid               21613 non-null  int64
 1   sell_date         21613 non-null  object
 2   price             21613 non-null  int64
 3   room_bed          21613 non-null  float64
 4   room_bath         21613 non-null  float64
 5   living_measure    21613 non-null  float64
 6   lot_measure       21613 non-null  float64
 7   ceil              21613 non-null  float64
 8   coast             21613 non-null  int64
 9   sight             21613 non-null  float64
 10  condition         21613 non-null  int64
 11  quality           21613 non-null  int64
 12  ceil_measure      21613 non-null  float64
 13  basement          21613 non-null  float64
 14  yr_built          21613 non-null  int32
 15  yr_renovated      21613 non-null  int64
 16  lat               21613 non-null  float64
 17  long              21613 non-null  float64
 18  living_measure15  21613 non-null  float64
 19  lot_measure15     21613 non-null  float64
 20  furnished         21613 non-null  float64
 21  total_area        21613 non-null  float64
 22  ZIP Code Name     21613 non-null  object
 23  Population        21613 non-null  float64
dtypes: float64(15), int32(1), int64(6), object(2)
memory usage: 4.0+ MB
```

Creating dummies with zipcode names creates a lot of additional columns. The data is then scaled using MinMaxScaler.

Neural Network Regression

Sequential()

```
X_train.shape: (15129, 21)
```

NN Evaluation

	loss	val_loss
0	435578863616.000000	406133997568.000000
1	435578929152.000000	406133997568.000000
2	435578863616.000000	406133997568.000000
3	435578863616.000000	406133997568.000000
4	435578863616.000000	406133997568.000000

```
473/473 [==============================] - 0s 299us/step - loss: 4355789619
20.0000

659984.0618681636
```

Data shape with dummies

(15129, 415)

Model

Epochs = 400

Batch size = 128

Evaluation

r^2 score = 0.6970340024972224 = 69.70%

Explained variance score = 0.6985811280993701

The ridge model with polynomial feature was robust but not to advantage, while the neural network model didn't perform as expected

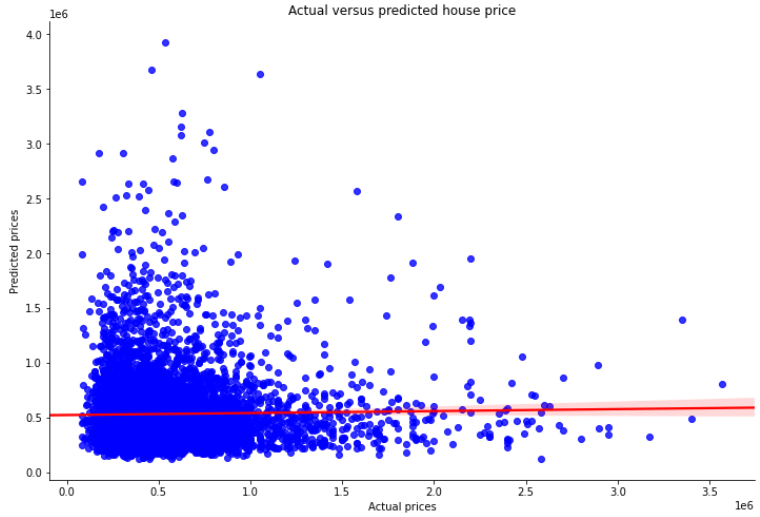
Final model: XGB grid2

The MAPE is close to that of the best MAPE model, while looking at the adjusted R^2, the model is a little bit more robust than the best model in that category

Predicted prices by the final model

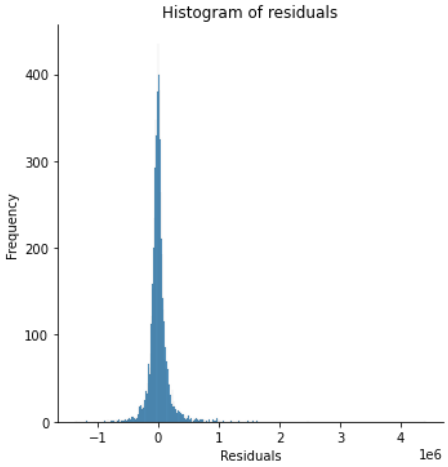
```
array([226558.98, 774862.2 , 446262.5 , ..., 554687.3 , 544902.44,
       593101.6 ], dtype=float32)
```

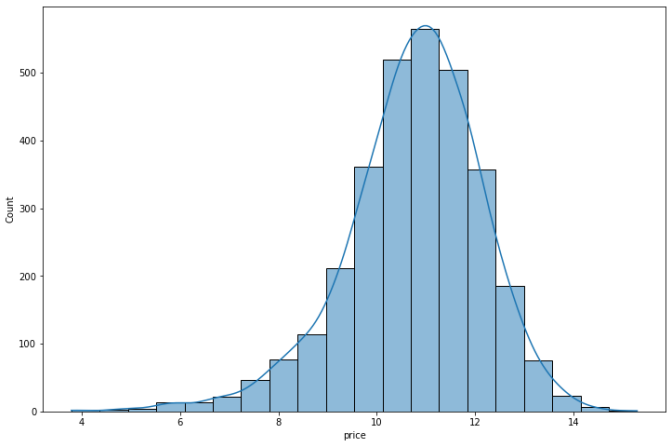
Final model: Actual versus predicted prices



This model will get more accurate as the houses become more high-end

Checking for normality



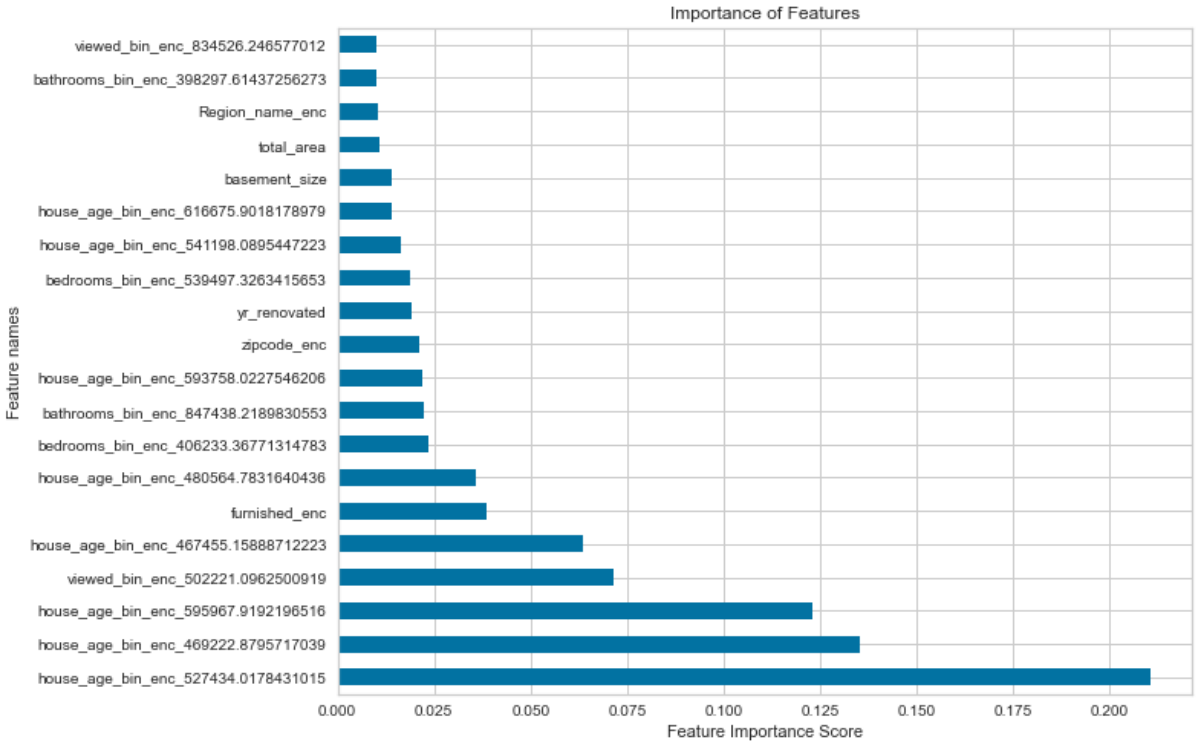


The curve is bell shaped. Normality satisfied, the model is valid

XGBoost is a popular implementation of Gradient Boosting because of its speed and performance.

Internally, XGBoost models represent all problems as a regression predictive modeling problem that only takes numerical values as input. If your data is in a different form, it must be prepared into the expected format.

Feature importance of best model



	Feature	Value
33	house_age_bin_enc_527434.0178431015	0.210782
30	house_age_bin_enc_469222.8795717039	0.135436
34	house_age_bin_enc_595967.9192196516	0.123176
22	viewed_bin_enc_502221.0962500919	0.071547
31	house_age_bin_enc_467455.15888712223	0.063723
11	furnished_enc	0.038405
32	house_age_bin_enc_480564.7831640436	0.035736
19	bedrooms_bin_enc_406233.36771314783	0.023521
18	bathrooms_bin_enc_847438.2189830553	0.022140
27	house_age_bin_enc_593758.0227546206	0.022102
9	zipcode_enc	0.021096
3	yr_renovated	0.018975
20	bedrooms_bin_enc_539497.3263415653	0.018602
28	house_age_bin_enc_541198.0895447223	0.016323
25	house_age_bin_enc_616675.9018178979	0.014190
2	basement_size	0.013898
4	total_area	0.010967
10	Region_name_enc	0.010241
16	bathrooms_bin_enc_398297.61437256273	0.009868
23	viewed_bin_enc_834526.246577012	0.009857
12	coast_enc	0.008796
26	house_age_bin_enc_607971.2997397617	0.008317

Insights from analysis

- Price increases with the number of bathrooms.
- Price increases linearly with the living area but for a certain space in the range of 1,800 sqft, the price can go very high.
- Two-floor houses have a higher price.
- The number of houses with no coast or waterfront view is much higher than of houses with coast, but waterfront property is pricier, generally.
- Houses in better condition fetch a better price.
- Price for a furnished property is way better.
- The houses that are viewed more fetch a better price for being more in demand.
- Houses with a large number of rooms are in the highest price bracket.
- Price of houses with 4 to 5 bathrooms is highest.
- The price of houses built in the 1900s is the highest because of antique value.

Recommendations

For buyers and sellers

- April is the best time to sell and February to buy
- Sunday is the best time to buy, while Saturday is the worst. Other way round for the sellers.
- Square footage of living area, bathroom count, latitudes lift price
- Longitude and year built important but not related linearly to price
- Buy in winter, sell in late spring or early summer

For investors

- **Go for the property** built after 2000 or between 1900 and 1920
- Medina, home to Bill Gates and Jeff Bezos, is pricier than Seattle

For intermediaries such as banks and evaluators

- For every year added to the age of a latest-built house, raise house value by 2,10,782 *
- For every year added to the age of a 90-year-old house, raise house value by 1,35,436*
- For every year added to the age of an 80-year-old house, raise house value by 1,23,176*
- Every time the house is viewed, raise house value by 71,547*
- If the house is furnished, house value increases by 38,405*

- If the house is renovated, house value increases by 18,975*
- If the house has 5 bedrooms, house value increases by 18,602*
- Each unit added to the basement size takes value up by 13,898*
- Each unit added to the total area takes value up by 10,967*
- For each one unit added to a house in the Medina region, house value goes up by 10,241*
- If house is on the coast (or for each unit of coast covered), the value goes up 8,796*
- For each unit added to the living space, raise house value by 7,423*
- Difference to living parameters takes the house value up by 7,206*
- For every level of quality 6 and up, rase the house value by 6,577*
- For each unit of outside space added, the house value goes up by 6,435*
- Having a basement should add only 6,392* to the house value
- For every new floor, the house value should appreciate only by 5,738*

- All other variables held constant.
- P-values below 0.05 for all, predictors significant.

The ensemble models have performed better than the linear regression, KNN, SVR models

XGBoost models have outperformed all other, with the exception of gradient boosting in onc case.

The best performance is given by X Gradient boosting tuned scaled model with training score = adjusted r^2 = 91.34% , RMSE-106203, and testing score = 81.51%, RMSE-163304.

The least mean absolute percentage error is given by Gradient boosting grid search model.

The chosen model was X Gradient boosting grid search model XGB grid2, with mean adjusted r^2 80.74, r^2 80.84, RMSE 166689.98, and MAPE = 18.13 based on robustness as compared with the top models.

The top key features that drive the price of the property are: 'furnished_1', 'yr_built', 'living_measure','quality_8', ' sqft_per_flr', 'lot_measure15', 'quality_9', 'ceil_measure', and 'total_area'.

The above data is also reinforced by the analysis done during bivariate analysis.

For further analysis, more dataset can be made by treating outliers in different ways and hyper-tuning the ensemble models.

Thank You