MA336 : Artificial Intelligence and Machine Learning with Applications Final Project

Name :- Nagadithya Bathala

Registration Number :- 2310244

Introduction

Now a days estimating house prices accurately is very essential. It helps many groups of the community such as house owners, buyers, real estate professionals, sellers, and stakeholders to buy or sell the house with reasonable and true market price. As we are aware that the house prices are usually estimated using traditional methods such as based on the demand, area and with the help of experts. These methods are time consuming and sometimes there might be bias in estimating the price. To overcome these problems, machine learning offers methods to predict the house prices based on the previous and present data.

The objective of this project is to predict the house price by making use of different machine learning and deep learning algorithms. By analysing the data, we aim to create machine learning models and train them to predict the price using unseen. data. The dataset I have used in this project is taken from Kaggle. The source of the data set is https://www.kaggle.com/datasets/shivachandel/kc-house-data. It has many important features which helps to predict the output feature price.

Different techniques that were used to analyse the data and build models are data cleaning, data preparation, exploratory data analysis, feature engineering, model selection, model evaluation and hyperparameter tuning. By the end of this project, the models built and trained using machine learning algorithms will be able to predict the house prices accurately.

Importing libraries

The libraries that are required to do data analysis tasks such as data cleaning, data preparation, exploratory data analysis (EDA), feature selection, and the modules required to build different machine learning and deep learning algorithms, and libraries related to evaluation metrics were imported here.

In [1]:
```
# Basic libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
import tensorflow

# Cross validation and Grid Serach libraries
from sklearn.model_selection import cross_val_score, KFold
from sklearn.model_selection import GridSearchCV

# Data scaling and data splitting libraries
from sklearn.preprocessing import OrdinalEncoder
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from sklearn.model_selection import train_test_split

#Evaluation metrics libraries
from sklearn.metrics import r2_score
from sklearn.metrics import mean_squared_error

# Manchine learning libraries
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.svm import SVR
from sklearn.neighbors import KNeighborsRegressor
from sklearn.ensemble import GradientBoostingRegressor
from xgboost import XGBRegressor
from sklearn.cluster import KMeans

#Deep learning libraries
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from keras.optimizers import Adam
```

```
WARNING:tensorflow:From C:\Users\nagad\anaconda3\Lib\site-packages\keras\src\losses.py:2976: The name tf.losses.sparse_softmax_cross_entropy is deprecated. Please use tf.compat.v1.losses.sparse_softmax_cross_entropy instead.
```

Dataset

Loading dataset into pandas dataframe

KC house pricing dataset has been imported into jupyter notebook using pandas data frame. Analysis was carried out using the complete dataset and sample dataset. The reason behind using sample dataset is to reduce computational costs. But the complete dataset was giving better results compared to sample of the dataset. So, it was decided to use the complete dataset to perform further analysis.

In [2]:
```
#Reading dataset
my_df=pd.read_csv('kc_house_data.csv')
my_df
```

Out[2]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	...	grade	sqft_above	sqft_basem
0	7129300520	20141013T000000	221900.0	3	1.00	1180	5650	1.0	0	0	...	7	1180.0	
1	6414100192	20141209T000000	538000.0	3	2.25	2570	7242	2.0	0	0	...	7	2170.0	4
2	5631500400	20150225T000000	180000.0	2	1.00	770	10000	1.0	0	0	...	6	770.0	
3	2487200875	20141209T000000	604000.0	4	3.00	1960	5000	1.0	0	0	...	7	1050.0	9
4	1954400510	20150218T000000	510000.0	3	2.00	1680	8080	1.0	0	0	...	8	1680.0	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	
21608	263000018	20140521T000000	360000.0	3	2.50	1530	1131	3.0	0	0	...	8	1530.0	
21609	6600060120	20150223T000000	400000.0	4	2.50	2310	5813	2.0	0	0	...	8	2310.0	
21610	1523300141	20140623T000000	402101.0	2	0.75	1020	1350	2.0	0	0	...	7	1020.0	
21611	291310100	20150116T000000	400000.0	3	2.50	1600	2388	2.0	0	0	...	8	1600.0	
21612	1523300157	20141015T000000	325000.0	2	0.75	1020	1076	2.0	0	0	...	7	1020.0	

21613 rows × 21 columns

Data Description

In [3]:
```
#Extracting all variable names
my_df.columns.tolist()
```

Out[3]:
```
['id',
 'date',
 'price',
 'bedrooms',
 'bathrooms',
 'sqft_living',
 'sqft_lot',
 'floors',
 'waterfront',
 'view',
 'condition',
 'grade',
 'sqft_above',
 'sqft_basement',
 'yr_built',
 'yr_renovated',
 'zipcode',
 'lat',
 'long',
 'sqft_living15',
 'sqft_lot15']
```

Description of each feature in the dataset-

- id - Unique identification number of each house
- date - Date on which the house was sold
- price - Price of the house
- bedrooms - Number of bedrooms
- bathrooms - Number of bathrooms
- sqft_livingsquare - Size of the living area
- sqft_lotsquare - Size of the lot
- floors - Number of floors in the house
- waterfront - House which has a view to a waterfront
- view - Shows now many times house has been viewed
- condition - How good the condition of the house is
- grade - Grade given to the housing unit based on King County grading system
- sqft_above - Size of the house apart from basement
- sqft_basement - Size of the basement

- yr_built - Year the house was built
- yr_renovated - Year the house was renovated
- zipcode - Zipcode of the area the house was located
- lat - Latitude coordinate of the house
- long - Longitude coordinate of the house
- sqft_living15 - Average size of the living space of the 15 nearest neighbors
- sqft_lot15 - Average size of the lot space of the 15 nearest neighbors

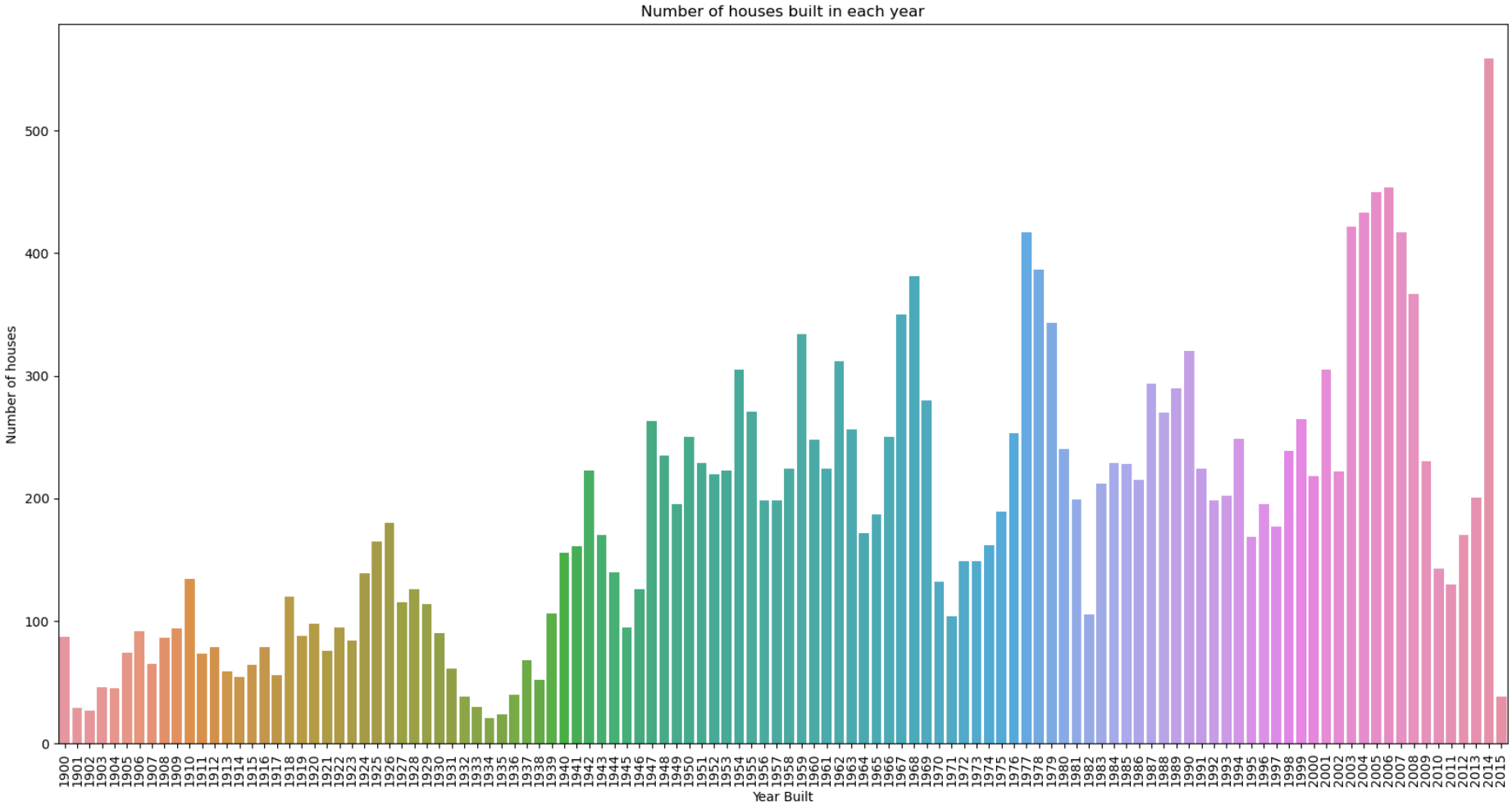
Preliminary Analysis

Exploratory Data Analysis(EDA)

EDA abbreviated as Exploratory Data Analysis is an essential step in any machine learning projects. It involves exploring data using different visualization and statistical techniques to understand the data that helps to perform further analysis. EDA can be performed on single feature to understand the data distribution and it is called as univariate analysis. Similarly, bivariate analysis is performed between two variables to understand the relationship between them. Likewise multivariate analysis is performed between multiple variables. Visualizing data helps to understand the data and the steps that needs to be performed to enrich the data.

Univariate Analysis on yr_built feature

In [4]:
```
#Setting figure size
plt.figure(figsize=(20,10))
#Plotting the countplot
sns.countplot(x='yr_built', data=my_df)
#Setting rotation for x-axis values
plt.xticks(rotation=90)
plt.xlabel("Year Built")
plt.ylabel("Number of houses")
plt.title("Number of houses built in each year")
plt.show()
```

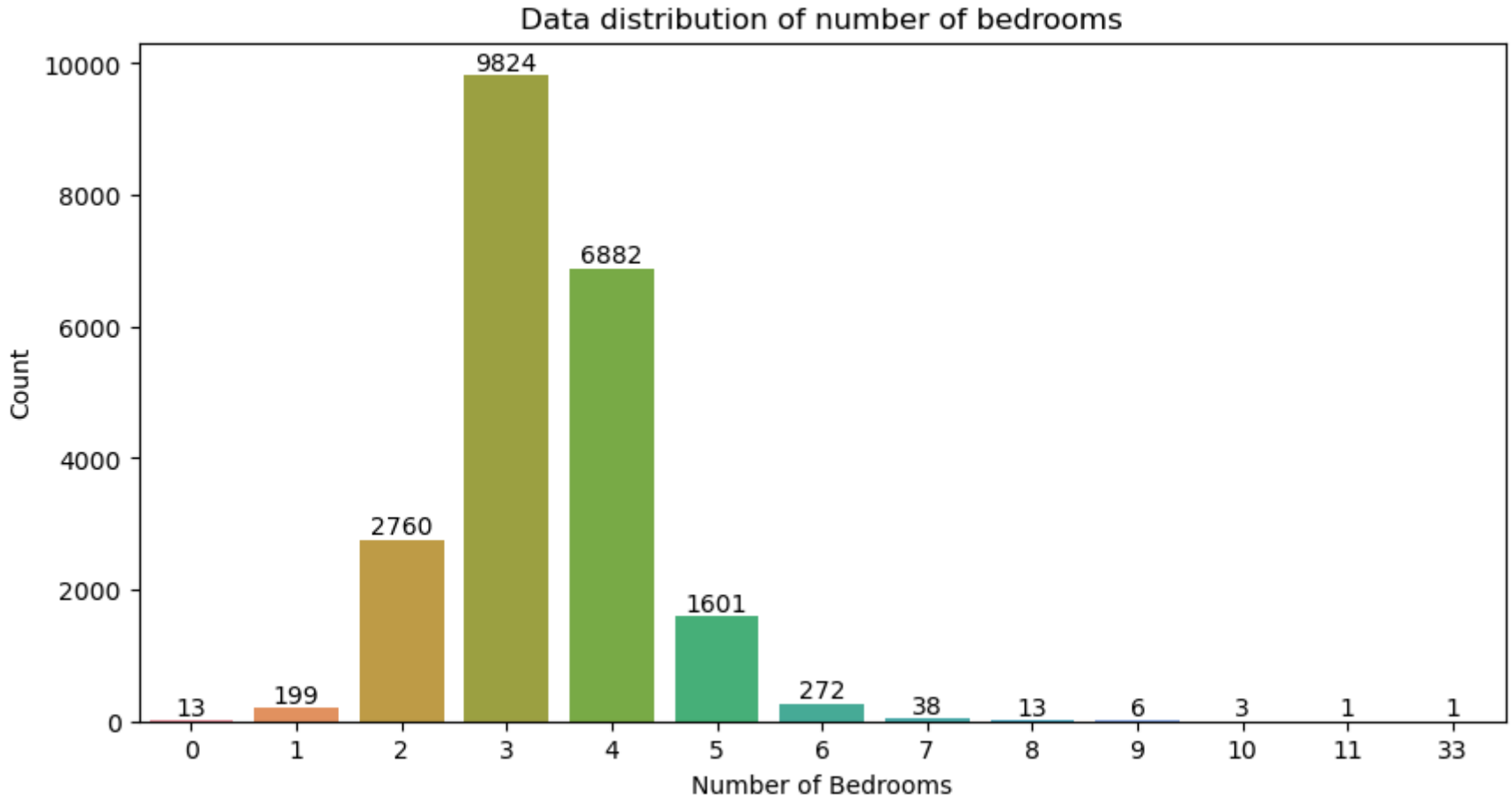


The count plot above shows the number houses built in each year. It is observed that the given dataset has houses that were built in the years between 1900 and 2015. The dataset has more number house that were built in 2014 compared to other years.

Univariate analysis on bedrooms feature

In [5]:
```
#Setting figure size
plt.figure(figsize=(10,5))
#Plotting the countplot
ax=sns.countplot(x='bedrooms', data=my_df)
plt.xlabel("Number of Bedrooms")
```

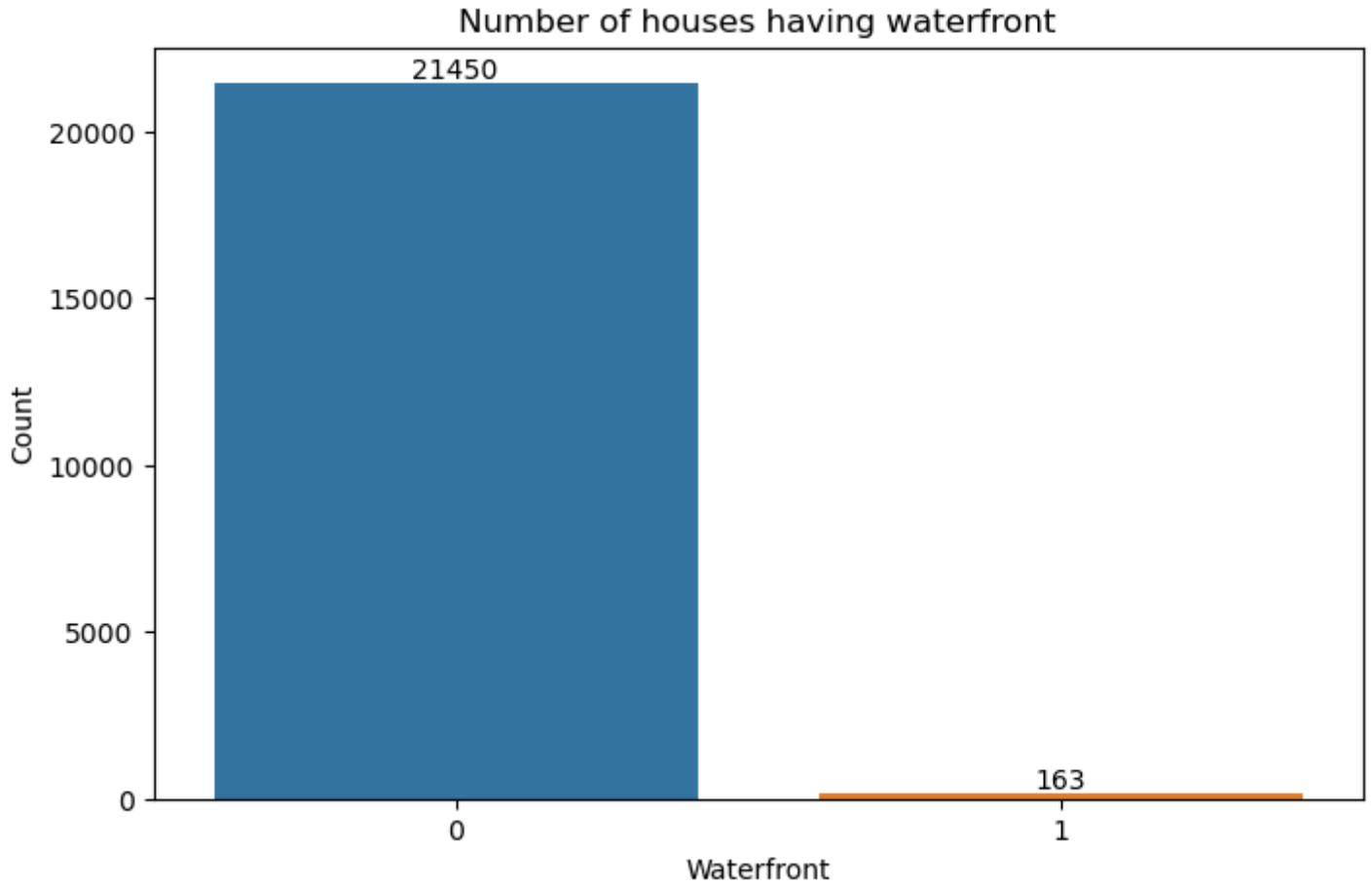
```
plt.ylabel("Count")
plt.title("Data distribution of number of bedrooms")
for i in ax.containers:
    ax.bar_label(i,)
plt.show()
```



The data distribution of number of bedrooms is explored using count plot. It is observed that most of the houses has at least 3 or 4 bedrooms. It also seen that 13 houses have 0 bedrooms and one house has 33 bedrooms which can be seen as an outlier.

Univariate analysis on waterfront feature

In [6]:
```
#Setting figure size
plt.figure(figsize=(8,5))
#Plotting the countplot
ax=sns.countplot(x='waterfront', data=my_df)
plt.xlabel("Waterfront")
plt.ylabel("Count")
plt.title("Number of houses having waterfront")
for i in ax.containers:
    ax.bar_label(i,)
plt.show()
```

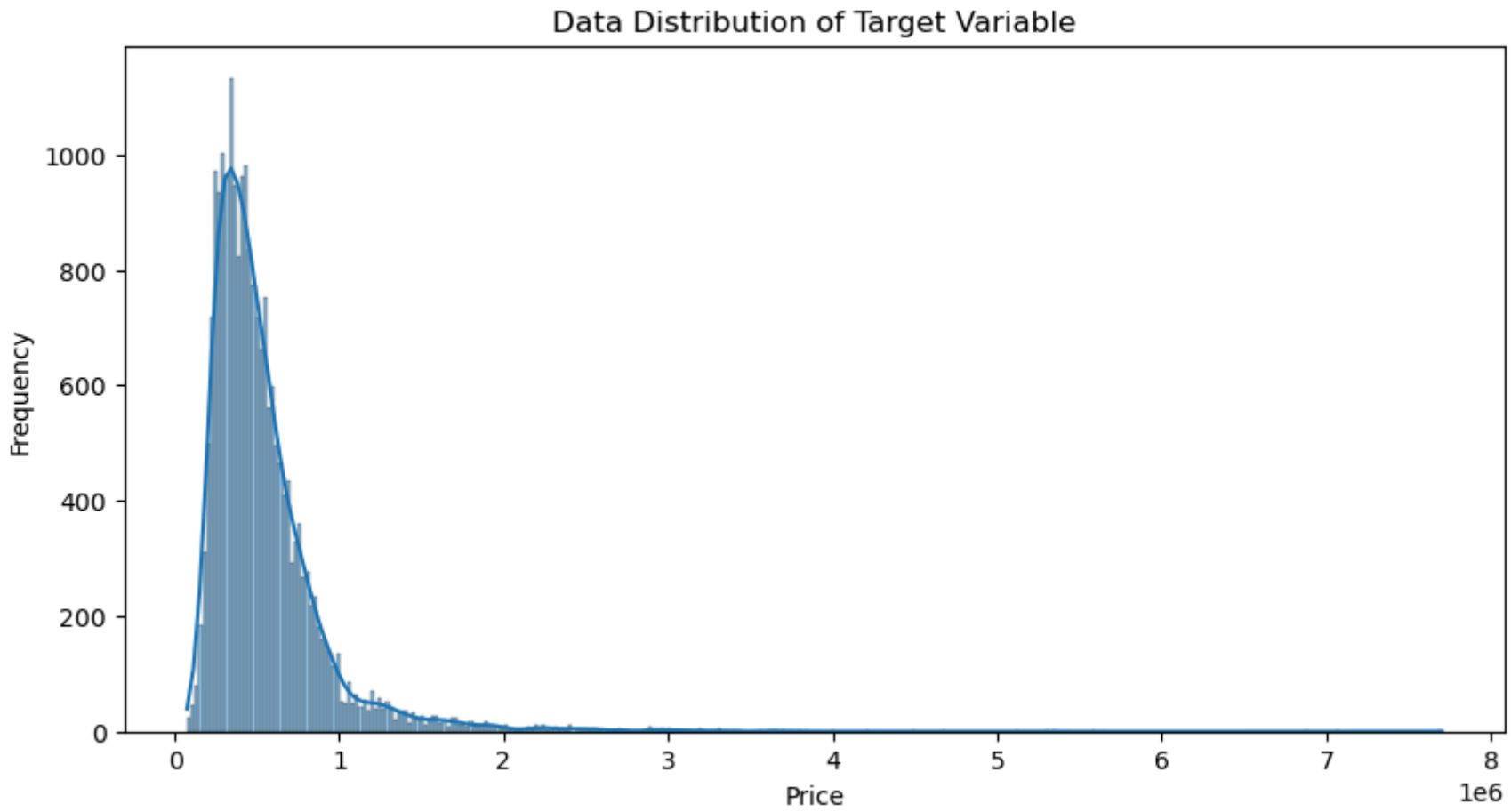


The plot above shows the number of houses has waterfront and number of houses doesn't have. From the plot we can observe that only 163 houses has waterfront whereas 21450 houses doesn't have waterfront.

Checking data distribution of the target variable

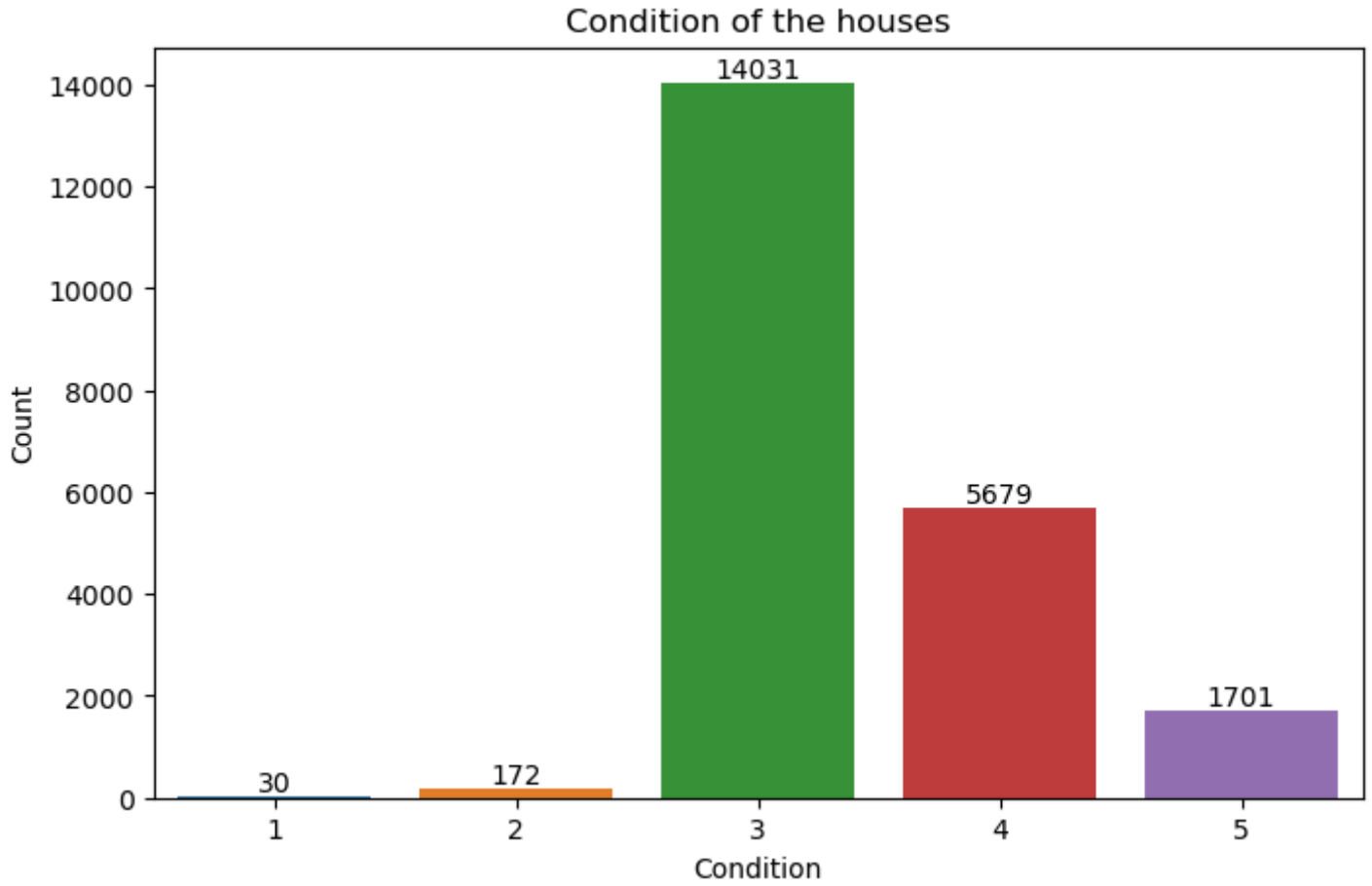
As predicting house price based on the input features is a regression problem, it is necessary to check for the data distribution of the target variable to see if the data is normally distributed. Many machine learning algorithms assumes that the data is normally distributed. Even though, the normality of the data is not always strictly necessary, it has a benefit of improving model performance. Here, we can see that the data is not normally distributed, and it is skewed towards right. Models were trained and tested without transforming the target variable and as well as by transforming the target variable.

In [7]:
```
#Using histplot in sns to check data distribution
plt.figure(figsize=(10, 5))
sns.histplot(my_df['price'], kde=True) #
plt.title("Data Distribution of Target Variable")
plt.xlabel('Price')
plt.ylabel('Frequency')
plt.show()
```



Univariate analysis on condition feature

In [8]:
```
#Setting figure size
plt.figure(figsize=(8,5))
#Plotting the countplot
ax=sns.countplot(x='condition', data=my_df)
plt.xlabel("Condition")
plt.ylabel("Count")
plt.title("Condition of the houses")
for i in ax.containers:
    ax.bar_label(i,)
plt.show()
```



The plot above shows the data distribution of condition of the houses. It is observed that more than half of the houses are in condition 3 whereas only 30 houses are in condition 1.

Data Preprocessing

Data preprocessing is the most essential part of any machine learning project. The basic steps involved in data preprocessing are checking for null values, na values, duplicate values, checking for outliers. If missing values were present in the data, data imputation techniques can be used to fill the missing values. Target variable distribution can be checked based on the type of problem we are dealing with. For regression problem, target variable distribution can be checked and data transformation can be performed to make sure data is normally distributed and if it is a

classification problem, response variable class imbalance can be checked. If the class imbalance exists, techniques such as under sampling, over sampling (SMOOTE) can be sed to balance the classes in response variable. These are all some basic steps of the data preprocessing.

In [9]:
```
#Checking for na values
my_df.isna().sum()
```

Out[9]:
```
id               0
date             0
price            0
bedrooms         0
bathrooms        0
sqft_living      0
sqft_lot         0
floors           0
waterfront       0
view             0
condition        0
grade            0
sqft_above       2
sqft_basement    0
yr_built         0
yr_renovated     0
zipcode          0
lat              0
long             0
sqft_living15    0
sqft_lot15       0
dtype: int64
```

In [10]:
```
#Checking for null values
my_df.isnull().all()
```

Out[10]:
```
id               False
date             False
price            False
bedrooms         False
bathrooms        False
sqft_living      False
sqft_lot         False
floors           False
waterfront       False
view             False
condition        False
grade            False
sqft_above       False
sqft_basement    False
yr_built         False
yr_renovated     False
zipcode          False
lat              False
long             False
sqft_living15    False
sqft_lot15       False
dtype: bool
```

In [11]:
```
#Checking for duplicate values
my_df.duplicated().sum()
```

Out[11]:
```
0
```

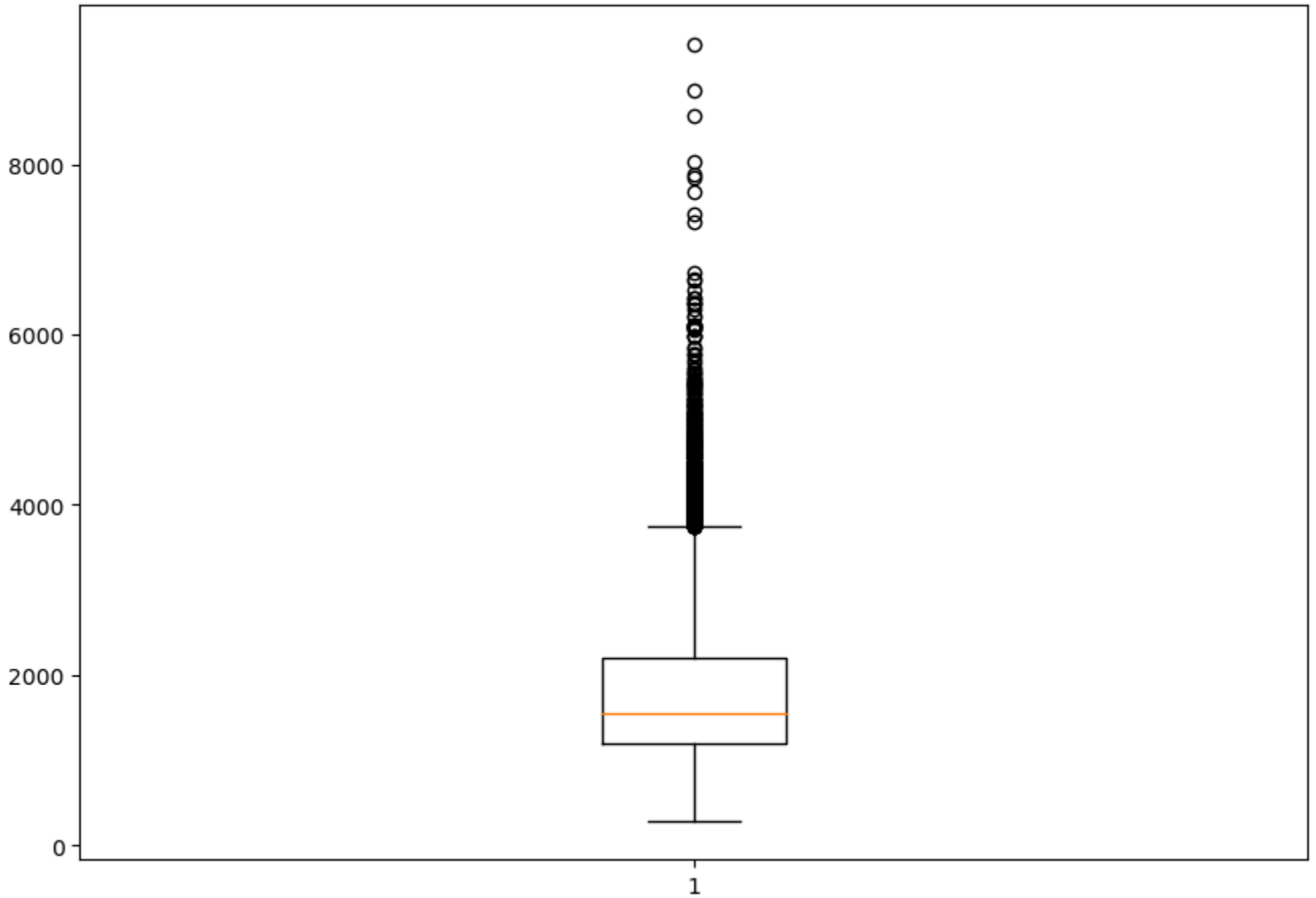
After checking for null values, na values and duplicates, it is observed that the data doesn't have any null values or duplicate values. But it has two na values. These null values are imputed using data imputation techniques as shown below.

Missing value Imputation

Missing values imputation techniques are the methods used to fill missing values based on the nature of the data. There are many methods available to impute missing values. Among them, the basic methods are mean, median and mode. Categorical data is imputed using mode. Numeric data is imputed using mean if the data doesn't have any outliers. If outliers were present in the data, then imputation can be done using median.

In [12]:
```
#Plotting box plot for feature having null values to check for outliers
fig = plt.figure(figsize =(10, 7))
```

```
plt.boxplot(my_df['sqft_above'].dropna())
plt.show()
```



In [13]:
```
#Imputing missing values using media as the data has outliers
my_df['sqft_above']=my_df['sqft_above'].fillna(my_df['sqft_above'].median())
```

The box plot above clearly shows that there are outliers present in the 'sqt_above' feature. So, using mean to impute the missing values can lead to biased imputation. So, imputing using median is the reliable way in this case. Therefore, missing values were imputed using median as shown above.

Extracting Year, Month and Day from Date object column

Dataset contains a date feature, and the datatype of this column is object. To extract the day, month, and year from the date feature, it was converted to datetime format using pandas 'to_datetime' library. Then day, month and year were extracted using pandas apply and lambda functions for further analysis.

In [14]:
```
# Using pandas to_datetime
my_df['date'] = pd.to_datetime(my_df.date)
#Extracting day, month and year using apply and lambda functions
my_df['Year'] = my_df.date.apply(lambda x:x.year)
my_df['Month'] = my_df.date.apply(lambda x:x.month)
my_df['Day'] = my_df.date.apply(lambda x:x.day)
```

Converting yr_renovated column into binary data column

Dataset contains a feature 'yr_renovated' which shows the year of renovation if the house is renovated else 0. To make this feature more informative, it was converted into binary data column i.e., 1 if house is renovated and 0 if it is not renovated. To perform this, a manual function was created and then applied this function on 'yr_renovated' feature in the data frame using apply function as shown below.

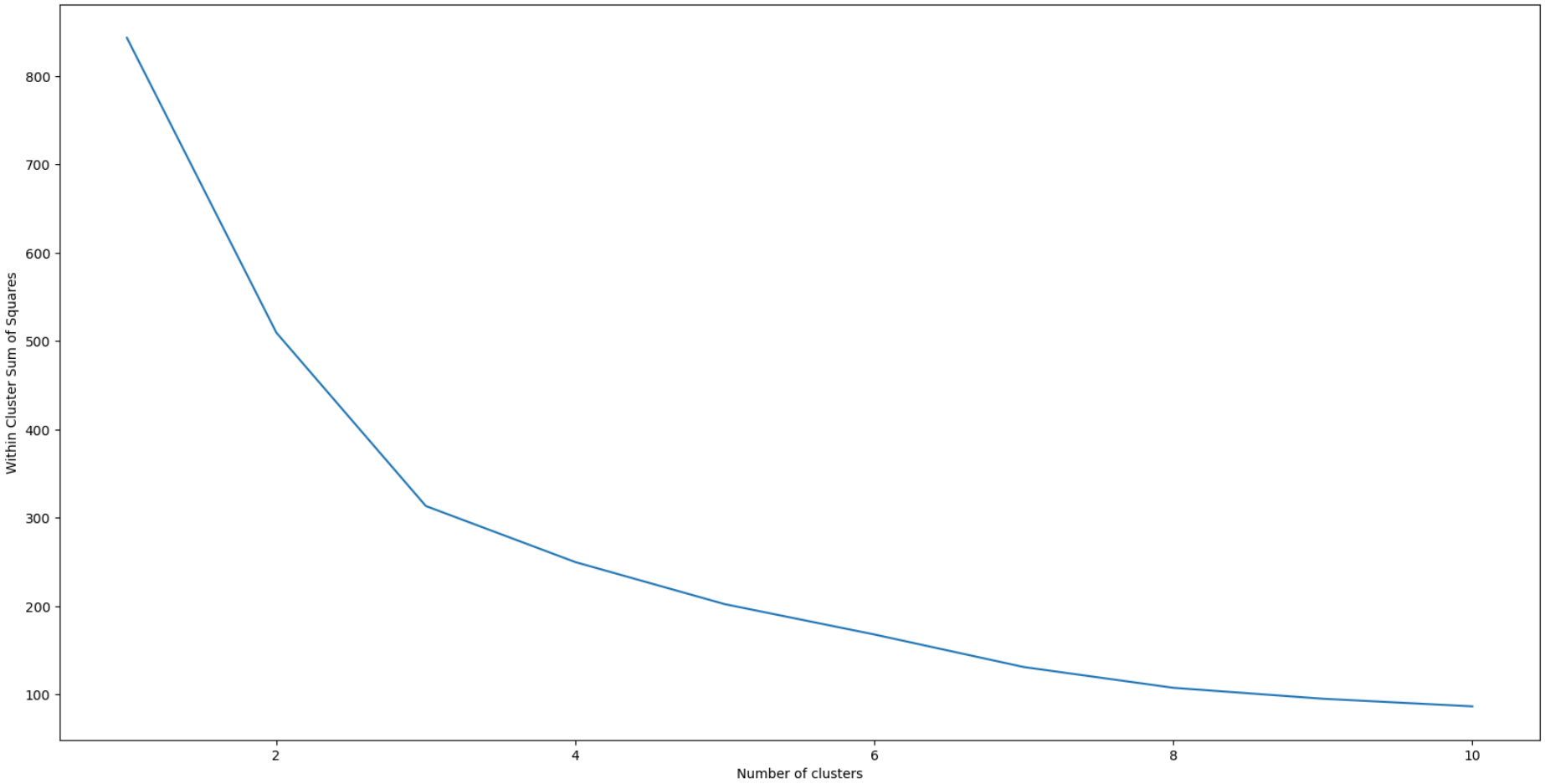
In [15]:
```
#Function to convert yr_renovated feature
def func_renovated(x):
    if x == 0:
        return 0
    return 1
#Applying the fuction to covert the 'yr_renovated' feature to binary feature 'renovated'
my_df['renovated'] = my_df['yr_renovated'].apply(func_renovated)
```

Binning latitude and longitude columns using k-means clustering

The chosen dataset has geospatial data such as latitude and longitude. It is not the best practice to just scale the geospatial data as we scale the other numeric data to apply machine learning algorithms. If we scale geospatial data, the information present in that will be lost. So, there are

other ways to deal with geospatial data. The most common method among them is binning the geospatial columns. I have grouped the coordinates i.e., latitude and longitude using k-means clustering algorithm. Elbow method was used to find out the optimal value of n_clusters.

In [16]:
```
#Finding optimal value of k for k-means clustering using elbow method
coords=my_df[['lat','long']].values

#Finding the optimal value of n_clusters
within_cluster_sum_of_sqrs=[]
for i in range(1,11):
    k_means=KMeans(n_clusters=i, init='k-means++', random_state=42)
    k_means.fit(coords)
    within_cluster_sum_of_sqrs.append(k_means.inertia_)

#Plotting the results
fig = plt.figure(figsize =(20, 10))
plt.plot(range(1, 11), within_cluster_sum_of_sqrs)
plt.xlabel('Number of clusters')
plt.ylabel('Within Cluster Sum of Squares')
plt.show()
```



From elbow method it is observed that the optimal value for n_clusters in 3. So, k-means clustering algorithm was applied on the coordinate features(Latitude and Longitude) to group the data points.

In [17]:
```
#Binning Latitude and Longitude columns
k_means=KMeans(n_clusters=3, init='k-means++', random_state=64)
# Adding the clusters to the dataframe
my_df['lat_long_cluster'] = k_means.fit_predict(coords)
my_df
```

Out[17]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	...	zipcode	lat	long	sqft_living15
0	7129300520	2014-10-13	221900.0	3	1.00	1180	5650	1.0	0	0	...	98178	47.5112	-122.257	1340
1	6414100192	2014-12-09	538000.0	3	2.25	2570	7242	2.0	0	0	...	98125	47.7210	-122.319	1690
2	5631500400	2015-02-25	180000.0	2	1.00	770	10000	1.0	0	0	...	98028	47.7379	-122.233	2720
3	2487200875	2014-12-09	604000.0	4	3.00	1960	5000	1.0	0	0	...	98136	47.5208	-122.393	1360
4	1954400510	2015-02-18	510000.0	3	2.00	1680	8080	1.0	0	0	...	98074	47.6168	-122.045	1800
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
21608	263000018	2014-05-21	360000.0	3	2.50	1530	1131	3.0	0	0	...	98103	47.6993	-122.346	1530
21609	6600060120	2015-02-23	400000.0	4	2.50	2310	5813	2.0	0	0	...	98146	47.5107	-122.362	1830
21610	1523300141	2014-06-23	402101.0	2	0.75	1020	1350	2.0	0	0	...	98144	47.5944	-122.299	1020
21611	291310100	2015-01-16	400000.0	3	2.50	1600	2388	2.0	0	0	...	98027	47.5345	-122.069	1410
21612	1523300157	2014-10-15	325000.0	2	0.75	1020	1076	2.0	0	0	...	98144	47.5941	-122.299	1020

21613 rows × 26 columns

Dropping unwanted columns from data frame

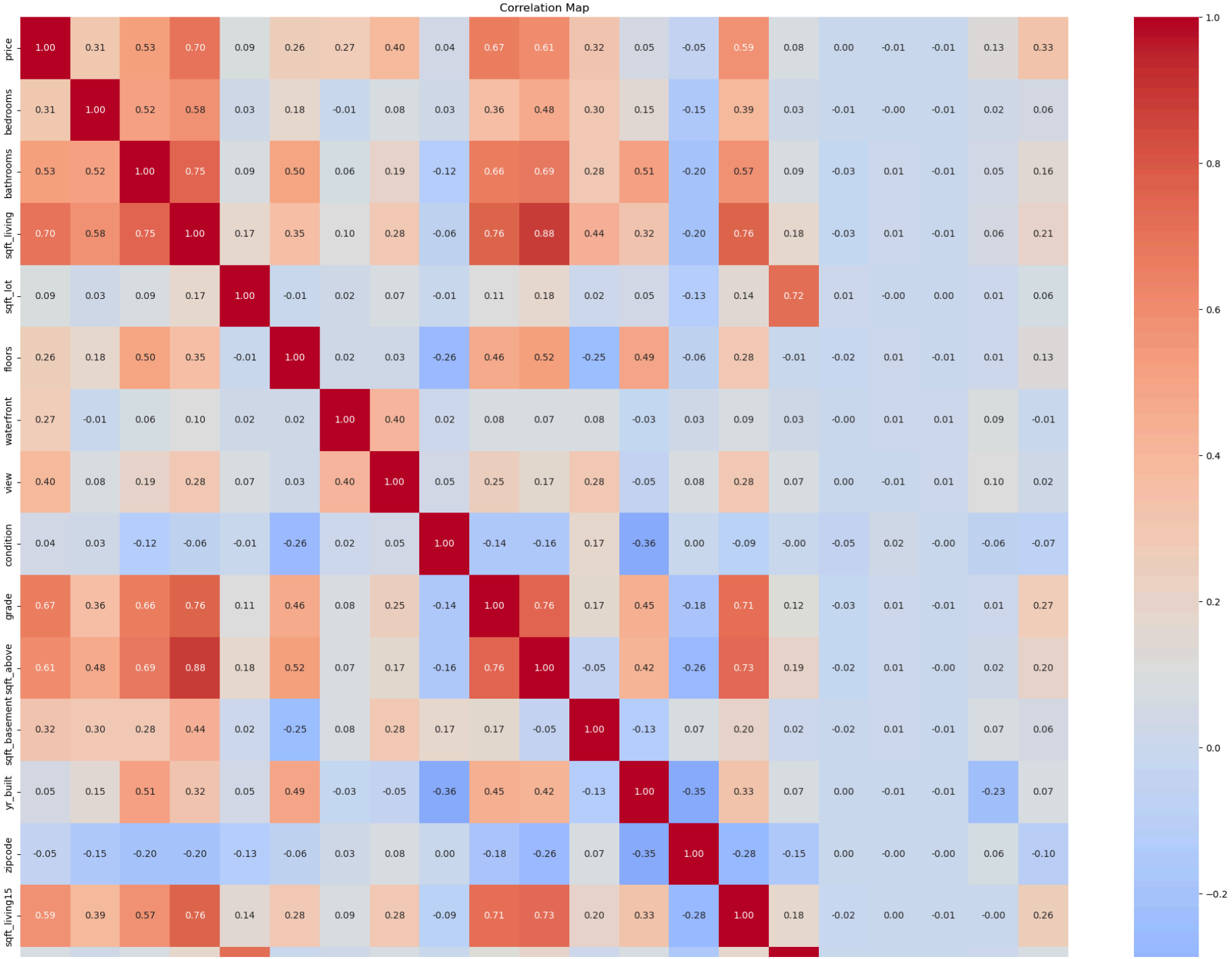
There were some unwanted features present in the data frame after applying data cleaning and transformation techniques on the initial data. These columns are not useful for model training and prediction. So, these unwanted columns were removed from the data frame.

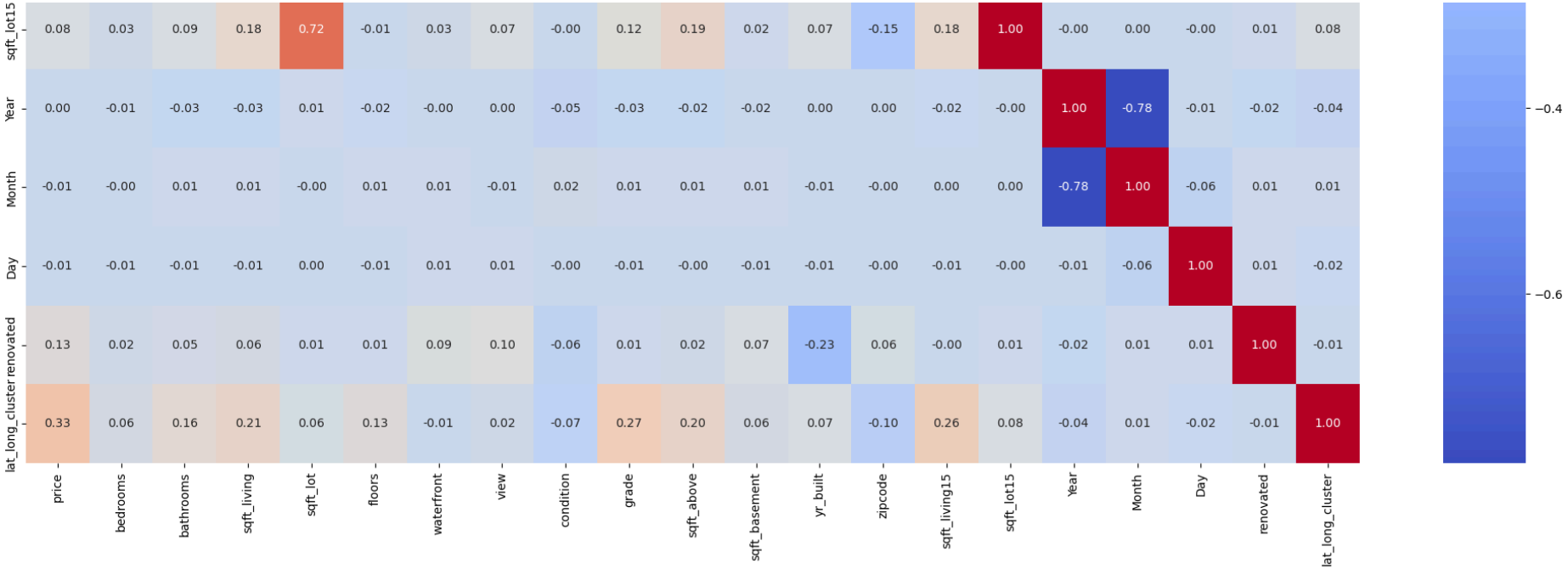
In [18]:
```
#Dropping columns
my_df=my_df.drop(['date', 'id', 'lat', 'long', 'yr_renovated'], axis=1)
```

Heatmap

Heatmap is a graphical representation of the data. It gives a correlation matrix. Heatmap is mainly used to check the correlation between the columns. It is plotted using sns library for the feature and response variables as shown below.

In [19]:
```
#Setting figure size
plt.figure(figsize=(25,25))
#Plotting Heat map
sns.heatmap(my_df.corr(), annot=True, cmap='coolwarm', fmt=".2f")
plt.title('Correlation Map')
plt.show()
```


Correlation Map
price 1.00 0.31 0.53 0.70 0.09 0.26 0.27 0.40 0.04 0.67 0.61 0.32 0.05 -0.05 0.59 0.08 0.00 -0.01 -0.01 0.13 0.33
bedrooms 0.31 1.00 0.52 0.58 0.03 0.18 -0.01 0.08 0.03 0.36 0.48 0.30 0.15 -0.15 0.39 0.03 -0.01 -0.00 -0.01 0.02 0.06
bathrooms 0.53 0.52 1.00 0.75 0.09 0.50 0.06 0.19 -0.12 0.66 0.69 0.28 0.51 -0.20 0.57 0.09 -0.03 0.01 -0.01 0.05 0.16
sqft_living 0.70 0.58 0.75 1.00 0.17 0.35 0.10 0.28 -0.06 0.76 0.88 0.44 0.32 -0.20 0.76 0.18 -0.03 0.01 -0.01 0.06 0.21
sqft_lot 0.09 0.03 0.09 0.17 1.00 -0.01 0.02 0.07 -0.01 0.11 0.18 0.02 0.05 -0.13 0.14 0.72 0.01 -0.00 0.00 0.01 0.06
floors 0.26 0.18 0.50 0.35 -0.01 1.00 0.02 0.03 -0.26 0.46 0.52 -0.25 0.49 -0.06 0.28 -0.01 -0.02 0.01 -0.01 0.01 0.13
waterfront 0.27 -0.01 0.06 0.10 0.02 0.02 1.00 0.40 0.02 0.08 0.07 0.08 -0.03 0.03 0.09 0.03 -0.00 0.01 0.01 0.09 -0.01
view 0.40 0.08 0.19 0.28 0.07 0.03 0.40 1.00 0.05 0.25 0.17 0.28 -0.05 0.08 0.28 0.07 0.00 -0.01 0.01 0.10 0.02
condition 0.04 0.03 -0.12 -0.06 -0.01 -0.26 0.02 0.05 1.00 -0.14 -0.16 0.17 -0.36 0.00 -0.09 -0.00 -0.05 0.02 -0.00 -0.06 -0.07
grade 0.67 0.36 0.66 0.76 0.11 0.46 0.08 0.25 -0.14 1.00 0.76 0.17 0.45 -0.18 0.71 0.12 -0.03 0.01 -0.01 0.01 0.27
sqft_above 0.61 0.48 0.69 0.88 0.18 0.52 0.07 0.17 -0.16 0.76 1.00 -0.05 0.42 -0.26 0.73 0.19 -0.02 0.01 -0.00 0.02 0.20
sqft_basement 0.32 0.30 0.28 0.44 0.02 -0.25 0.08 0.28 0.17 0.17 -0.05 1.00 -0.13 0.07 0.20 0.02 -0.02 0.01 -0.01 0.07 0.06
yr_built 0.05 0.15 0.51 0.32 0.05 0.49 -0.03 -0.05 -0.36 0.45 0.42 -0.13 1.00 -0.35 0.33 0.07 0.00 -0.01 -0.01 -0.23 0.07
zipcode -0.05 -0.15 -0.20 -0.20 -0.13 -0.06 0.03 0.08 0.00 -0.18 -0.26 0.07 -0.35 1.00 -0.28 -0.15 0.00 -0.00 -0.00 0.06 -0.10
sqft_living15 0.59 0.39 0.57 0.76 0.14 0.28 0.09 0.28 -0.09 0.71 0.73 0.20 0.33 -0.28 1.00 0.18 -0.02 0.00 -0.01 -0.00 0.26
1.0
0.8
0.6
0.4
0.2
0.0
−0.2



Checking unique values of categorical data columns to perform data encoding

There were some categorical columns present in the data. Categorical data can be encoded using ordinal encoding technique. Here, unique values in each of the categorical columns were checked to decide on which columns to perform ordinal encoding. Unique () function was used to find out the unique values present in each column.

In [20]:
```
# bedrooms feature unique values
my_df['bedrooms'].unique()
```

Out[20]:
```
array([ 3,  2,  4,  5,  1,  6,  7,  0,  8,  9, 11, 10, 33], dtype=int64)
```

In [21]:
```
# bathrooms feature unique values
my_df['bathrooms'].unique()
```

Out[21]:
```
array([1.  , 2.25, 3.  , 2.  , 4.5 , 1.5 , 2.5 , 1.75, 2.75, 3.25, 4.  ,
       3.5 , 0.75, 4.75, 5.  , 4.25, 3.75, 0.  , 1.25, 5.25, 6.  , 0.5 ,
       5.5 , 6.75, 5.75, 8.  , 7.5 , 7.75, 6.25, 6.5 ])
```

In [22]:
```
# floors feature unique values
my_df['floors'].unique()
```

Out[22]:
```
array([1. , 2. , 1.5, 3. , 2.5, 3.5])
```

In [23]:
```
# waterfront feature unique values
my_df['waterfront'].unique()
```

Out[23]:
```
array([0, 1], dtype=int64)
```

In [24]:
```
# view feature unique values
my_df['view'].unique()
```

Out[24]:
```
array([0, 3, 4, 2, 1], dtype=int64)
```

In [25]:
```
# condition feature unique values
my_df['condition'].unique()
```

Out[25]:
```
array([3, 5, 4, 1, 2], dtype=int64)
```

In [26]:
```
# grade feature unique values
my_df['grade'].unique()
```

Out[26]:
```
array([ 7,  6,  8, 11,  9,  5, 10, 12,  4,  3, 13,  1], dtype=int64)
```

Data Preparation

Data preparation is one of the crucial steps in any machine learning project. It involves data encoding depending on the nature of the data, normalizing data using different scaling techniques such as minmax scalar or standardization according to the requirement. Data encoding can be performed using different techniques. One Hot Encoding is used when the categorical data has unordered levels. Ordinal Encoding is used when the categorical data has ordered levels. In this project I have used both the techniques to perform data encoding. The categorical features present in the data are all ordered data. Initially, Ordinal encoding was used to convert the data which is suitable to apply machine learning algorithms on it. Then, One Hot Encoding was used, and the results were not great compared to ordinal encoding. So, it was decided to continue using ordinal encoded data to do further analysis.

Data Scaling can be done using different techniques. Most commonly used ones are MinMaxScalar and StandardScalar. MinMaxScalar scales the data between 0 and 1. StandardScalar scales the data such that it has mean of 0 and standard deviation of 1. In this project, machine learning models were fitted with the data that is scaled using both the techniques i.e., mixmax scalar and standard scalar. Minmax scalar scales the data between 0 and 1. Using minmax scalar is suitable when the algorithm doesn't assume the data normality. Standard scalar scales the data such that the data has a mean of 0 and standard deviation of 1. Some models were not giving good results when standard scalar was used. Data normalized using minmax scalar gave good results compared to standard scalar. So, in this project minmax scalar was used to normalize the data which is suitable for machine learning algorithms to train and predict the output accurately.

Ordinal encoding to convert Categorical data to Numeric data

Ordinal encoding is a data encoding technique used to convert the categorical data with ordered levels into numeric data. It assigns a unique value for each unique category in the feature.

In [27]:
```
#Ordinal encoding
ordinal_encoder = OrdinalEncoder()
ordinal_cols = ['bedrooms','bathrooms','floors','grade','zipcode','yr_built','condition','Year','Month','Day']
my_df[ordinal_cols] = ordinal_encoder.fit_transform(my_df[ordinal_cols])
```

Normalizing data

Data normalization or feature scaling is a technique which is used to scale the data such that the data is suitable for machine learning algorithms to learn from the data and predict the results precisely. Here minmax scalar was used to normalize the data as discussed above.

In [28]:
```
#Data scaling using MinMaxScaler
scaler = MinMaxScaler()
normalize_cols=['sqft_living','sqft_above','sqft_basement','sqft_living15','price','yr_built','zipcode','sqft_lot','sqft_lot15',
                'bedrooms','bathrooms','floors','grade','condition','lat_long_cluster','view','Year','Month','Day']
#Creating a deep copy of the df
my_df_1=my_df.copy()
#Scaling data
my_df_1[normalize_cols] = scaler.fit_transform(my_df_1[normalize_cols])
my_df_1
```

Out[28]:

	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	condition	grade	...	sqft_basement	yr_built	zipcode	sqft
0	0.019266	0.250000	0.103448	0.067170	0.003108	0.0	0	0.0	0.5	0.454545	...	0.000000	0.478261	0.956522	
1	0.060721	0.250000	0.275862	0.172075	0.004072	0.4	0	0.0	0.5	0.454545	...	0.082988	0.443478	0.797101	
2	0.013770	0.166667	0.103448	0.036226	0.005743	0.0	0	0.0	0.5	0.363636	...	0.000000	0.286957	0.231884	
3	0.069377	0.333333	0.379310	0.126038	0.002714	0.0	0	0.0	1.0	0.454545	...	0.188797	0.565217	0.840580	
4	0.057049	0.250000	0.241379	0.104906	0.004579	0.0	0	0.0	0.5	0.545455	...	0.000000	0.756522	0.536232	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
21608	0.037377	0.250000	0.310345	0.093585	0.000370	0.8	0	0.0	0.5	0.545455	...	0.000000	0.947826	0.608696	
21609	0.042623	0.333333	0.310345	0.152453	0.003206	0.4	0	0.0	0.5	0.545455	...	0.000000	0.991304	0.869565	
21610	0.042898	0.166667	0.068966	0.055094	0.000503	0.4	0	0.0	0.5	0.454545	...	0.000000	0.947826	0.855072	
21611	0.042623	0.250000	0.310345	0.098868	0.001132	0.4	0	0.0	0.5	0.545455	...	0.000000	0.904348	0.217391	
21612	0.032787	0.166667	0.068966	0.055094	0.000337	0.4	0	0.0	0.5	0.454545	...	0.000000	0.939130	0.855072	

21613 rows × 21 columns

Splitting data

Data splitting is a fundamental step in every machine learning project. Usually, the data is divided into train and test sets using train_test_split library. In this project I have made use of k-fold cross validation to get accurate output. I have used train test split data to train and test neural networks. The variables 'feature_vars' has all feature variables data and 'response_var' has target variable data. These two variables were used in k-fold cross validation to perform further analysis.

In [29]:
```
# Features and response data variables for k-fold cross validation
feature_vars=my_df_1.drop(columns='price', axis=1)
response_var=my_df_1['price']

#Dividing train data into train and test sets
x_train, x_test, y_train, y_test = train_test_split(feature_vars,response_var,test_size=0.2,random_state=42)
```

In [30]:
```
#Shapes of the train and test data
print("Dim of x_train : ",x_train.shape)
print("Dim of y_train : ",y_train.shape)
print("Dim of x_test : ",x_test.shape)
print("Dim of y_test : ",y_test.shape)
```

```
Dim of x_train :  (17290, 20)
Dim of y_train :  (17290,)
Dim of x_test :  (4323, 20)
Dim of y_test :  (4323,)
```

K-fold Cross validation

It is a technique used to evaluate the performance of the machine learning algorithms using different metrics. It helps in determining how well the model is performing. This method divides the data into k folds where in each iteration it uses k-1 folds for training and 1-fold for testing and this process continues k times. The average of these scores is considered as robust and reliable estimate of model performance compared to simple train test split method.

Tuning number of folds (cv) value for K-fold cross validation

In [31]:
```
#Model with k-fold cross validation
R2_Scores={}
for i in range(3,21):
    model_LinearRegression = LinearRegression()
    LR_r2_scores=cross_val_score(model_LinearRegression, feature_vars, response_var, cv=i, scoring='r2')
    R2_Scores[i]=np.average(LR_r2_scores)
print("R2 scores of different folds : ",R2_Scores)
max_r2=max(R2_Scores.values())
optimal_fold = list(R2_Scores.values()).index(max_r2)+3
print("\nMax R2 score is ",max(R2_Scores.values()), " at number of folds equal to",optimal_fold)
```

```
R2 scores of different folds :  {3: 0.671871794505334, 4: 0.6720920584847165, 5: 0.6718871319040994, 6: 0.6722488121073701, 7:
0.6731089783187721, 8: 0.6710893387860672, 9: 0.6733336202014188, 10: 0.6725499304625446, 11: 0.6724719948788525, 12: 0.67422857
40397511, 13: 0.6739022234876443, 14: 0.6747209529629676, 15: 0.6747915602138369, 16: 0.6736730963973501, 17: 0.674723239292460
7, 18: 0.6745527676281133, 19: 0.6744412385158771, 20: 0.6754438416259373}

Max R2 score is  0.6754438416259373  at number of folds equal to 20
```

Note :- The R2 scores are almost similar for different number of folds as observed from the above performed test. In machine learning projects three, five and ten number of folds are considered as optimal folds for larger datasets to reduce the time consumption. **So, to reduce the computational costs, cv value was taken as 5 to perform further analysis**.

Analysis using Machine Learning Algorithms

Machine learning algorithms are widely used in many industries to solve different problems because of their ability to find patterns, automate decision making process and make the predictions accurately. They are used to predict the response based on the input features. There are many machine learning algorithms available and each ones have different strengths and weaknesses. There are three types of machine learning algorithms i.e., supervised learning, unsupervised learning and reinforcement learning algorithms. Out of these types the most common and widely used are supervised and unsupervised based on the problem statement. In supervised we have two types of algorithms. They are Classification algorithms and Regression algorithms. Classification algorithms are used when the response variable have classes or response variable has categorical data. Some examples of classification problems are credit card default payment, predicting sentiment from the text, predicting the class from image data. Regression algorithms are used when the response variable has numeric values. Some examples of regression problems are predicting sales of a company, predicting house prices, and predicting car prices.

In this project the problem statement is to predict the price of the houses based on different input features. So, this is a regression problem. Models like Linear Regression, Decision Tree, Random Forest, Support Vector Machines (SVM), K-Nearest Neighbours (KNN), Gradient Boosting Machines (GBM) and Extreme Gradient Boosting (XGB) were used to predict the house prices.

Evaluation Metrics

In machine learning the model performance is evaluated using evaluation metrics. There are many types of evaluation metrics in machine learning depending on the type of the problem. For classification problems model performance is measured using f1 score, precision, recall, misclassification error and accuracy. Regression models evaluated using R-Squared score, mean squared error (MSE), mean absolute error (MAE), root mean squared error (RMSE) and mean absolute percentage error (MAPE). The problem statement in this project is regression. So, R-Squared score and Mean Squared Error (MSE) were used to measure the performance of the machine learning algorithms.

Machine Learning models

Linear Regression

Linear Regression is the most basic foundational algorithm in machine learning, and it is used for predictive analysis. It is used to predict the relationship between two or more variables. In other words, we say that it is a statistical method used to model the relationship between the dependent and independent variables. Linear Regression has some basic assumptions such as assuming the linear relationship between response and predictor variables, assuming the data is a representation of a sample and assuming all the variables in the data are normally distributed. It works on a simple straight-line equation which is represented as y=mx+b where, m is the slope of the line, b is the intercept, y is dependent variable and x is independent variable. The aim of the linear regression is to find a line that fits best and minimizes the error which is the difference between predicted and actual values.

Model

In [32]:
```
#Linear Regression model with k-fold cross validation
def model_LR(feature_vars,response_var):
    model_LinearRegression = LinearRegression()

    LR_r2_scores=cross_val_score(model_LinearRegression, feature_vars, response_var, cv=5, scoring='r2')
    print("LR  -> R2 score : ",round(np.average(LR_r2_scores),2))
```

Decision Tree

Decision tree is a popular machine learning algorithm that is used for both classification and regression analysis. It uses conditional control statements, and it is a tree based and non-parametric supervised learning algorithm. It works by dividing the features into nodes. It has a hierarchical tree structure where it contains root nodes, branches, internal nodes, and leaf nodes. It starts with all the features and selects the features that best divides the data into groups based on the Gini impurity value. In training the feature based split continues until max depth value exceeds. During prediction the decision tree traverses the starting from root node until leaf node based on the input features and the decision rules that it learned during the training. The benefits of decision tree are it can identify the complex non-linear relationships between the variables, and they are easy to interpret because of their tree structure. However, it has the disadvantage of overfitting if the data is noisy and irrelevant features exist in the dataset. The algorithm used for classification is DecisonTreeClassifier and for regression it is DecisionTreeRegressor. Gene impurity value plays a key role in decision making process in this algorithm.

In [33]:
```
#Decision Tree model with k-fold cross validation
def model_DT(feature_vars,response_var, max_depth):
    model_DecisionTree = DecisionTreeRegressor(max_depth=max_depth, random_state=64)
```

```
    DT_r2_scores=cross_val_score(model_DecisionTree, feature_vars, response_var, cv=5, scoring='r2')
    print("DT  -> R2 score : ",round(np.average(DT_r2_scores),2))
```

Random Forest

Random Forest algorithm is one of the most widely used and user friendly algorithm in machine learning. It is a type to ensemble learning methods and is a tree based algorithm. It works for both classification and regression problems. It is a group of decision trees where it has n number of decision trees, and the output of the random forest is the aggregate of the results of decision trees. It combines the results of decision trees to make the prediction more accurate and robust. The important feature of random forest is bootstrapping. It divided the dataset into multiple bootstrap samples and each of the sample is created by randomly sampling the dataset with replacement. For each bootstrap sample a decision tree is constructed and trained them with that sample. Then predictions are made on unseen data and the response of these models are aggregated to get the final response. This nature of the random forest makes it robust against the overfitting and it is not sensitive to outliers. Some of the advantages of random forest are handling higher dimension data, robust against overfitting and performs well on a variety of datasets. However it has disadvantages like it is computationally expensive when dealing with large datasets and the interpretation is difficult.

In [34]:
```
#Random Forest model with k-fold cross validation
def model_RF(feature_vars,response_var,n_estimators,max_depth):
    model_RandomForest = RandomForestRegressor(n_estimators=n_estimators, max_depth=max_depth, random_state=64)

    RF_r2_scores=cross_val_score(model_RandomForest, feature_vars, response_var, cv=5, scoring='r2')
    print("RF  -> R2 score : ",round(np.average(RF_r2_scores),2))
```

Support Vector Machines(SVM)

Support Vector Machines is a supervised machine learning algorithm, and it is one of the most widely used algorithm. It works on a concept of a hyperplane. It finds a hyperplane that best separates the classes. Hyperplane is decided such that the distance is maximum between the two classes. One of the key parameters of SVM is C. It is used to control the difference between minimizing the error and maximizing the margin. In SVM, large margin is considered as good and it suitable for both linear and non-linear data. It works best for small, complex, and high dimensional datasets. It is most widely used algorithm in fields like bioinformatics, image, and text classification. SVM works for both classification (SVC) and regression problems (SVR). Advantages of SVM are it is not sensitive to outliers, works for high dimensional data and image classification. Drawbacks are it doesn't perform well on large datasets, computationally costly and parameter tuning is difficult.

In [35]:
```
#SVM model with k-fold cross validation
def model_SVM(feature_vars,response_var,C):
```

```
    model_svm = SVR(C=C, epsilon=0.01)

    SVM_r2_scores=cross_val_score(model_svm, feature_vars, response_var, cv=5)
    print("SVM -> R2 score : ",round(np.average(SVM_r2_scores),2))
```

KNN Algorithm

K-Nearest Neighbours(KNN) algorithm is a supervised machine learning algorithm which is used for both classification and regression problems. It is a non-parametric algorithm and assumes that similar data points tend to have similar data labels. It memorises the whole dataset and its class labels and it doesn't require any further training. While making predictions it calculates the distance between the unseen data point and all other data points in the memory. Then the algorithm finds the k nearest neighbours of the new data point based on the distance. For classification it considers the mode of the k nearest neighbours as the final prediction and for regression problems it calculates the weighted average of the target variables of k nearest neighbours as prediction to the new data point. Advantages are interpretability, robust to noisy data. Disadvantages are computationally expensive for large datasets as it involves calculating the distance between all of the data points and it doesn't perform better on high dimensional data.

In [36]:
```
#KNN model with k-fold cross validation
def model_KNN(feature_vars,response_var,n_neighbors):
    model_KNN = KNeighborsRegressor(n_neighbors=n_neighbors)

    KNN_r2_scores=cross_val_score(model_KNN, feature_vars, response_var, cv=5)
    print("KNN -> R2 score : ",round(np.average(KNN_r2_scores),2))
```

Gradient Boosting Machines(GBM)

Gradient Boosting algorithm commonly known as Gradient Boosting Machines is a popular supervised machine learning algorithm. It is a type of ensemble learning techniques and a tree based algorithm which can be used for both classification and regression problems. It builds the model with multiple decision trees and combines them sequentially to predict the final output. GBM has decision tree as fixed base estimator. Initially it assigns the same weights to all the observations in the data and feeds the data to the first model. Then it increases the weights of the misclassified data points and reduces the weights of the correctly classified data points. This method improves the chances of misclassified values to be classified correctly in the next model. The aim of GBM is to improve the overall performance of the model by increasing the weights based on the errors of the previous models and gradually increase the model performance. Advantages of GBM are prediction speed, accuracy, ability to handle complex and large datasets. Disadvantages are it is sensitive to noisy data and outliers, computationally expensive.

In [37]:
```
#GBM model with k-fold cross validation
def model_GBM(feature_vars,response_var,n_estimators):
    model_GBM = GradientBoostingRegressor(n_estimators=n_estimators,random_state=64)

    GBM_r2_scores=cross_val_score(model_GBM, feature_vars, response_var, cv=5)
    print("GBM -> R2 score : ",round(np.average(GBM_r2_scores),2))
```

XG Boosting

Extreme Gradient Boosting Machines is a supervised machine learning algorithm and it can be used to deal with both classification and regression problems. It is a type of ensemble learning techniques and a tree based model. It works similar to GBM model with has multiple decision tress which are connected sequentially to improve the performance of the model. The main difference between these two models is that XGB uses the regularization technique which is used to minimize the loss function and improve the model performance.

In [38]:
```
#XGB model with k-fold cross validation
def model_XGB(feature_vars,response_var,n_estimators):
    model_XGB = XGBRegressor(n_estimators=n_estimators)

    XGB_r2_scores=cross_val_score(model_XGB, feature_vars, response_var, cv=5)
    print("XGB -> R2 score : ",round(np.average(XGB_r2_scores),2))
```

Hyperparameter Tuning

Hyperparameter tuning is essential for many machine learning algorithms to improve the model performance. It is the process of selecting the optimal parameters for machine learning and deep learning algorithms by training and evaluating the models with a range of values. Parameters values are selected such that the model performance is maximum, and errors are minimum. These parameters help to improve the models performance on unseen data. These parameters are not learned during the training process. They should be set before the training starts. There are many methods available to do parameter tuning. In this project, GridSearchCV library from sklearn module was used as it is easy to use and interpret the results.

In machine learning most of the algorithms require parameter tuning to get maximum performance from the models. Each algorithm has different parameters. But sometimes it is not possible to tune all the parameters because of the computational costs and time constraints. Every parameter has a default value assigned in the backend. So, we can focus more the parameters that makes a big impact after tuning them. So, in this project at least 1 parameter for each algorithm was tuned and discussed about the impact they made after tuning the each parameter.

Decision Tree

Decision tree has a key parameter 'max_depth'. It refers to the maximum depth of the tree. The complexity of the model increases with increase in the max depth value. A deeper tree learns better from the training data. But it is also prone to over fitting.

In [39]:
```
#Grid search using GridSearchCV
def model_DT_GridSearch(feature_vars,response_var):
    model_DecisionTree = DecisionTreeRegressor(random_state=64)
    decision_params = [{'max_depth': list(range(1, 21))}]
    grid_decision = GridSearchCV(model_DecisionTree, decision_params, cv=5, scoring='r2')
    grid_decision.fit(feature_vars, response_var)
    print("Decision Tree -> Best parameters : ",grid_decision.best_params_)
```

Random Forest

Random Forest algorithm contains n number of decision trees. The key parameters of the random forest are 'n_estimators' and 'max_depth'. The depth of each decision tree is controlled by max depth parameter and the number of decision trees in the forest is controlled by n_estimators parameter. Both of these parameters play a key role in maximizing the model performance.

In [40]:
```
#Grid search using GridSearchCV library
def model_RF_GridSearch(feature_vars,response_var):
    model_RandomForest = RandomForestRegressor(random_state=64)
    RF_params = [{'n_estimators':list(range(1,21)),'max_depth': list(range(1, 21))}]
    grid_RF = GridSearchCV(model_RandomForest, RF_params, cv=5, scoring='r2')
    grid_RF.fit(feature_vars, response_var)
    print("RF  -> Best parameters : ",grid_RF.best_params_)
```

SVM

Support vector machines(SVM) has some key parameters such as C and epsilon. The C value helps to avoid the misclassification of each training data point. The model takes larger margin hyperplane if the C values is small and vice versa. Epsilon parameter in SVM is a loss function that indicates the epsilon intensive loss. It is useful when dealing with regression problems and it is used in support vector regressor. Because of the computational costs only C parameter was tuned.

In [41]:
```
#Grid search using GridSearchCV library
def model_SVM_GridSearch(feature_vars,response_var):
    model_svm = SVR(epsilon=0.01)
    SVM_params = [{'C':list(range(1,10,1))}]
    grid_SVM = GridSearchCV(model_svm, SVM_params, cv=5, scoring='r2')
    grid_SVM.fit(feature_vars, response_var)
    print("SVM -> Best parameters : ",grid_SVM.best_params_)
```

KNN

K-Nearest Neighbours algorithm predicts the response based on the nearest neighbours. It has 'n_neighbours' parameter which indicates the number of neighbours and it plays an important role in predicting the response on unseen data accurately.

In [42]:
```
#Grid search using GridSearchCV library
def model_KNN_GridSearch(feature_vars,response_var):
    model_KNN = KNeighborsRegressor()
    KNN_params = [{'n_neighbors':list(range(1,21))}]
    grid_KNN = GridSearchCV(model_KNN, KNN_params, cv=5, scoring='r2')
    grid_KNN.fit(feature_vars, response_var)
    print("KNN -> Best parameters : ",grid_KNN.best_params_)
```

GBM

The key parameter of Gradient Boosting Machines is 'n_estimators'. GBM is also a tree based algorithm and it contains n number of decision trees that are connected sequentially and trained to make predictions on unseen data. This parameter controls the number of decision trees in the model.

In [43]:
```
#Grid search using GridSearchCV library
def model_GBM_GridSearch(feature_vars,response_var):
    model_gbm = GradientBoostingRegressor(random_state=64)
    GBM_params = [{'n_estimators':list(range(0,101,5))}]
    grid_GBM = GridSearchCV(model_gbm, GBM_params, cv=5, scoring='r2')
    grid_GBM.fit(feature_vars, response_var)
    print("GBM -> Best parameters : ",grid_GBM.best_params_)
```

XGB

Extreme Gradient Boosting Machines has a key parameter 'n_estimators'. XGB is also a tree based algorithm and it contains n number of decision trees that are connected sequentially and trained using regularization technique to make predictions on unseen data. This parameter controls the number of decision trees in the model.

In [44]:
```
#Grid search using GridSearchCV library
def model_XGB_GridSearch(feature_vars,response_var):
    model_xgb = XGBRegressor()
    XGB_params = [{'n_estimators':list(range(0,101,5))}]
    grid_XGB = GridSearchCV(model_xgb, XGB_params, cv=5, scoring='r2')
    grid_XGB.fit(feature_vars, response_var)
    print("XGB -> Best parameters : ",grid_XGB.best_params_)
```

Model Evaluation

Model evaluation is process of the evaluating the performance of machine learning algorithm on unseen data and observe how well the model is able to generalize the new observations. It involves training the models on train dataset and testing the models on unseen data and comparing the predictions made by the model with true values in the test dataset. The evaluation is done using different evaluation metrics based on the problem statement.

In this project model performance was evaluated using different types of pre-processed data such as:

1. Model evaluation with all features
2. Model evaluation with features selected using correlation matrix
3. Model evaluation with transformed target variable

Model evaluation with all features

Parameter Tuning

In [45]:
```
# Ingoning the warnings
warnings.filterwarnings("ignore")
# Descision Tree
model_DT_GridSearch(feature_vars,response_var)
#Random Forest
model_RF_GridSearch(feature_vars,response_var)
```

```
#SVM
model_SVM_GridSearch(feature_vars,response_var)
#KNN
model_KNN_GridSearch(feature_vars,response_var)
#GBM
model_GBM_GridSearch(feature_vars,response_var)
#XGB
model_XGB_GridSearch(feature_vars,response_var)
```

```
Decision Tree -> Best parameters :  {'max_depth': 10}
RF  -> Best parameters :  {'max_depth': 18, 'n_estimators': 17}
SVM -> Best parameters :  {'C': 2}
KNN -> Best parameters :  {'n_neighbors': 5}
GBM -> Best parameters :  {'n_estimators': 100}
XGB -> Best parameters :  {'n_estimators': 100}
```

Fitting and Evaluating models with k-fold cross validation

In [46]:

```
# Ingoning the warnings
warnings.filterwarnings("ignore")
#Linear Regression
model_LR(feature_vars,response_var)
#Decision Tree
model_DT(feature_vars,response_var,max_depth=10)
#Random Forest
model_RF(feature_vars,response_var,n_estimators=17,max_depth=18)
#SVM
model_SVM(feature_vars,response_var,C=2)
#KNN
model_KNN(feature_vars,response_var,n_neighbors=5)
#GBM
model_GBM(feature_vars,response_var,n_estimators=100)
#XGB
model_XGB(feature_vars,response_var,n_estimators=100)
```

```
LR  -> R2 score :  0.67
DT  -> R2 score :  0.76
RF  -> R2 score :  0.83
SVM -> R2 score :  0.84
KNN -> R2 score :  0.71
GBM -> R2 score :  0.84
XGB -> R2 score :  0.86
```

After performing initial data preprocessing tasks such as data cleaning and data preparation tasks the dataset has 20 features. For initial analysis all the 20 features were used to training and evaluating the models. Model training and evaluation were performed using K-Fold Cross Validation. Before training the models, hyperparameters of each model were tuned to get the best parameters using GridsearchCV library. After fine tuning the hyperparameters, each model was trained and then evaluated using r2_score as evaluation metric.

Some models were performing better and some were not giving great r2 scores. Models like Random Forest, GBM and XGB gave good results compared to other models. This is because of their robustness to noisy data and they can handle the preprocessing steps internally. KNN, Decision Tree and Linear Regression didn't perform well and further data preparation is required to improve the model performance.

Model evaluation on features selected based on correlation matrix

In [47]:
```
#Correlation of all features with price variable
my_df.corr()['price'].sort_values()
```

Out[47]:
```
zipcode            -0.050889
Day                -0.014670
Month              -0.010081
Year                0.003576
condition           0.036362
yr_built            0.054012
sqft_lot15          0.082447
sqft_lot            0.089661
renovated           0.126092
floors              0.256794
waterfront          0.266369
bedrooms            0.314906
sqft_basement       0.323816
lat_long_cluster    0.327695
view                0.397293
bathrooms           0.524510
sqft_living15       0.585379
sqft_above          0.605561
grade               0.667529
sqft_living         0.702035
price               1.000000
Name: price, dtype: float64
```

In [48]:
```
#Making a deep copy of df
my_df_2=my_df.copy()
```

```
#Dropping features that have correlation value less that 0.1
my_df_2=my_df_2.drop(['Year', 'Month', 'Day'], axis=1)
my_df_2=my_df_2.drop(['condition'], axis=1)
my_df_2=my_df_2.drop(['sqft_lot', 'sqft_lot15'], axis=1)
my_df_2=my_df_2.drop(['zipcode'], axis=1)
my_df_2=my_df_2.drop(['yr_built'], axis=1)

# Scaling data
scaler = MinMaxScaler()
normalize_cols=['sqft_living','sqft_above','sqft_basement','sqft_living15','price','bedrooms','bathrooms',
                'floors','grade','lat_long_cluster','view']
my_df_2[normalize_cols] = scaler.fit_transform(my_df_2[normalize_cols])

# Features variable and Response variable
feature_vars_1=my_df_2.drop(columns='price', axis=1)
response_var_1=my_df_2['price']
```

Parameter tuning

In [49]:
```
# Ingoning the warnings
warnings.filterwarnings("ignore")
# Descision Tree
model_DT_GridSearch(feature_vars_1,response_var_1)
#Random Forest
model_RF_GridSearch(feature_vars_1,response_var_1)
#SVM
model_SVM_GridSearch(feature_vars_1,response_var_1)
#KNN
model_KNN_GridSearch(feature_vars_1,response_var_1)
#GBM
model_GBM_GridSearch(feature_vars_1,response_var_1)
#XGB
model_XGB_GridSearch(feature_vars_1,response_var_1)
```

```
Decision Tree -> Best parameters :  {'max_depth': 8}
RF  -> Best parameters :  {'max_depth': 11, 'n_estimators': 16}
SVM -> Best parameters :  {'C': 1}
KNN -> Best parameters :  {'n_neighbors': 10}
GBM -> Best parameters :  {'n_estimators': 100}
XGB -> Best parameters :  {'n_estimators': 20}
```

Fitting and Evaluating models

In [50]:
```
# Ingoning the warnings
warnings.filterwarnings("ignore")
#Linear Regression
model_LR(feature_vars_1,response_var_1)
#Decision Tree
model_DT(feature_vars_1,response_var_1,max_depth=8)
#Random Forest
model_RF(feature_vars_1,response_var_1,n_estimators=16,max_depth=11)
#SVM
model_SVM(feature_vars_1,response_var_1,C=1)
#KNN
model_KNN(feature_vars_1,response_var_1,n_neighbors=10)
#GBM
model_GBM(feature_vars_1,response_var_1,n_estimators=100)
#XGB
model_XGB(feature_vars_1,response_var_1,n_estimators=20)
```

```
LR  -> R2 score :  0.62
DT  -> R2 score :  0.75
RF  -> R2 score :  0.79
SVM -> R2 score :  0.79
KNN -> R2 score :  0.76
GBM -> R2 score :  0.79
XGB -> R2 score :  0.79
```

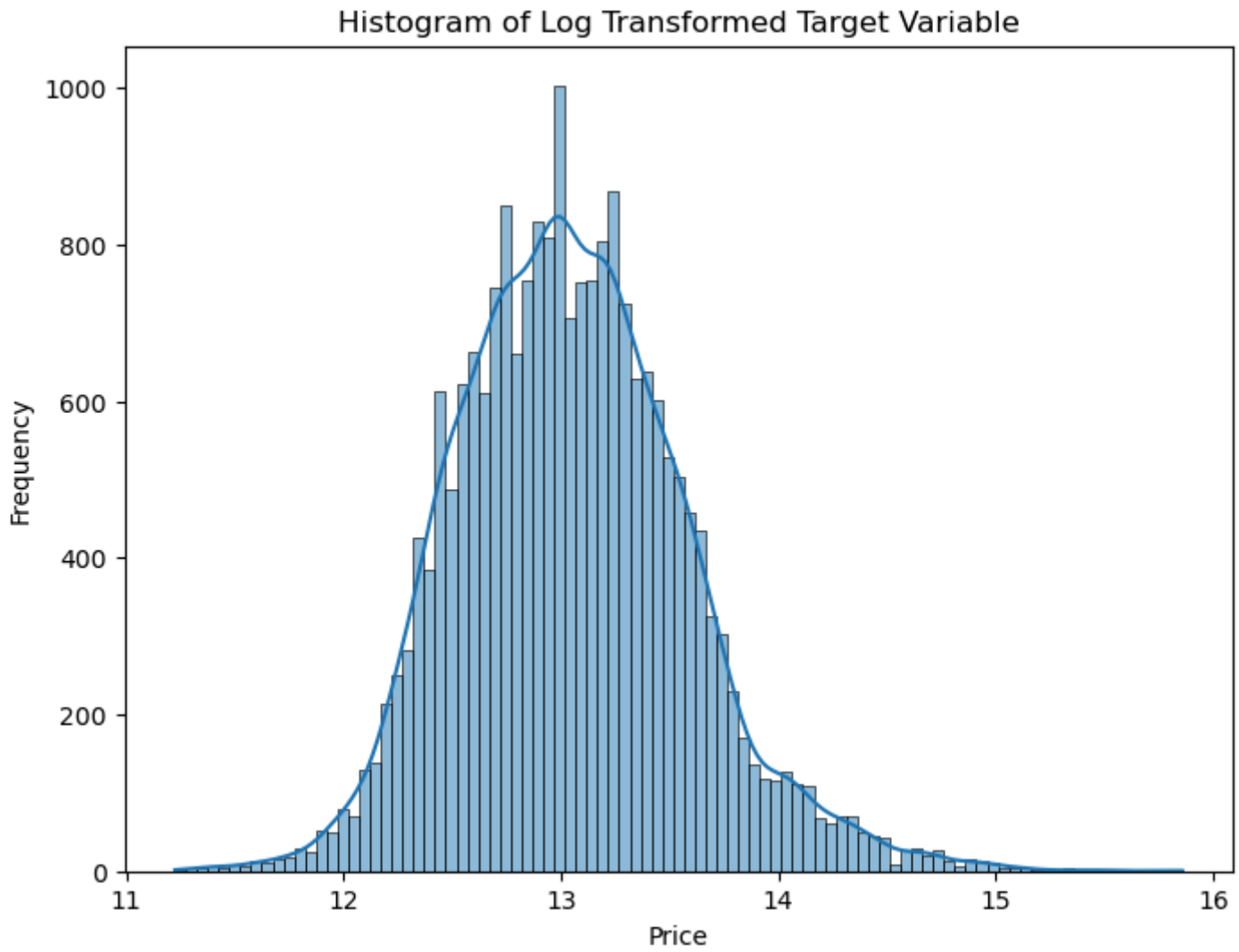
The process of selecting features based on their correlation with response variable is known as feature selection using correlation matrix. Here the hypothesis is that the features with less correlation with response variable will not contribute in predicting the response variable. But it is just an assumption and to decide on that analysis need to be performed by training the model and making predictions on unseen data. The features with correlation value less than 0.1 with response variable were removed from the data frame. The analysis above shows that the model performance was reduced for features selected using correlation matrix. It implies that the features with small correlation value are also contributing in predicting the target variable. Performance of models like RF, GBM and XGB was reduced which performed good in previous analysis.

Model evaluation with all features and transformed target variable

Many machine learning algorithms assume the data is normally distributed. Even though, data normality is not strictly necessary it helps to improve the model performance. As discussed previously in data preprocessing section, the target variable is not normally distributed, and the data is skewed towards the right. So, here the target variable was transformed using three types of transformation namely log transformation, square root transformation and cube root transformation to check for which method the data is normally distributed.

Log Transformation

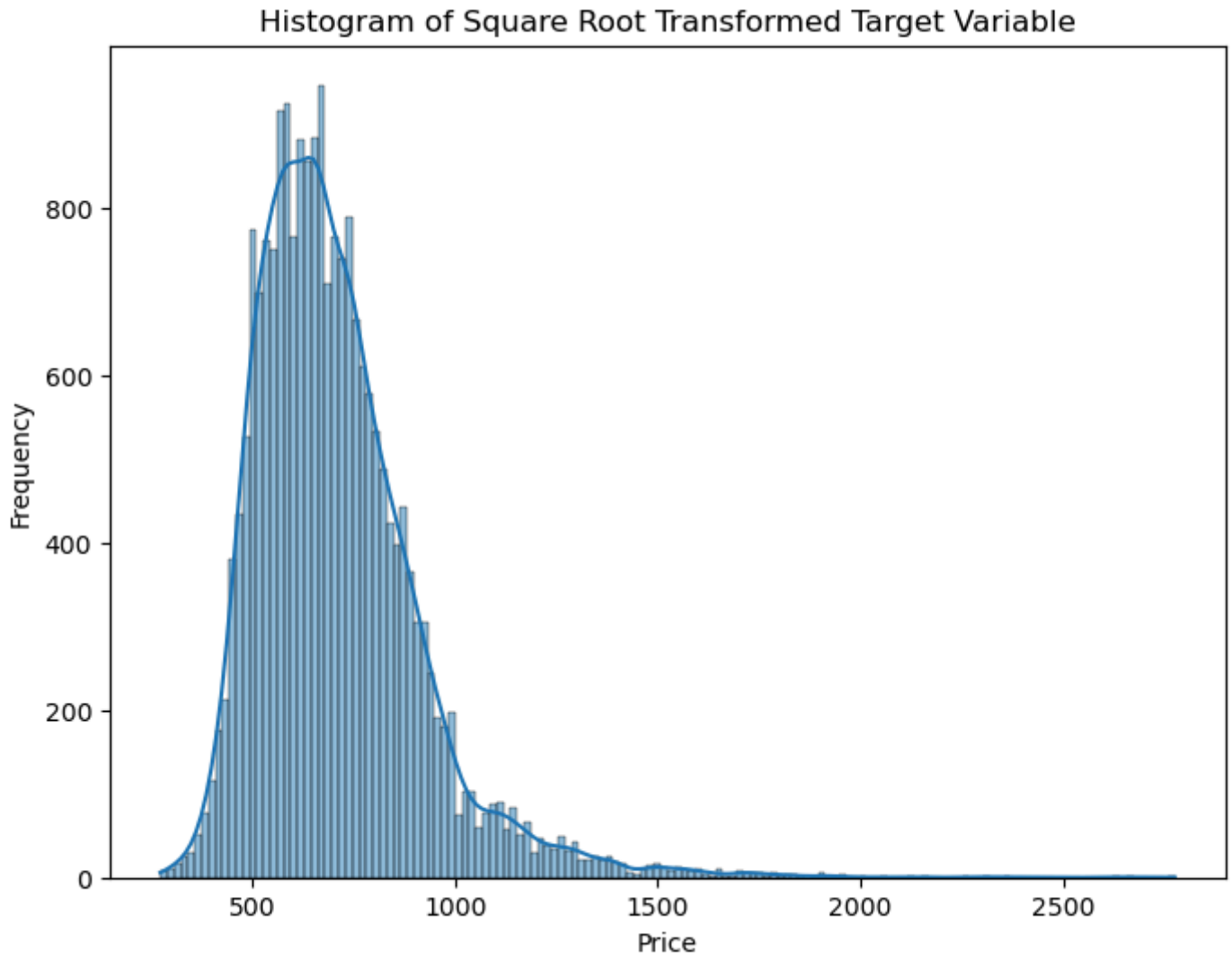
In [51]:
```
#Making a deep copy of the df
my_df_log=my_df.copy()
my_df_sqrt=my_df.copy()
my_df_cbrt=my_df.copy()

#Performing log transormation on the price column
my_df_log['price']=np.log(my_df_log['price'])

#Plotting the results
plt.figure(figsize=(8, 6))
sns.histplot(my_df_log['price'], kde=True)
plt.title("Histogram of Log Transformed Target Variable")
plt.xlabel('Price')
plt.ylabel('Frequency')
plt.show()
```



Square Root Transformation

In [52]:
```
#Performing square root transormation on the price column
my_df_sqrt['price']=np.sqrt(my_df_sqrt['price'])
#Plotting the results
plt.figure(figsize=(8, 6))
sns.histplot(my_df_sqrt['price'], kde=True)
plt.title("Histogram of Square Root Transformed Target Variable")
```

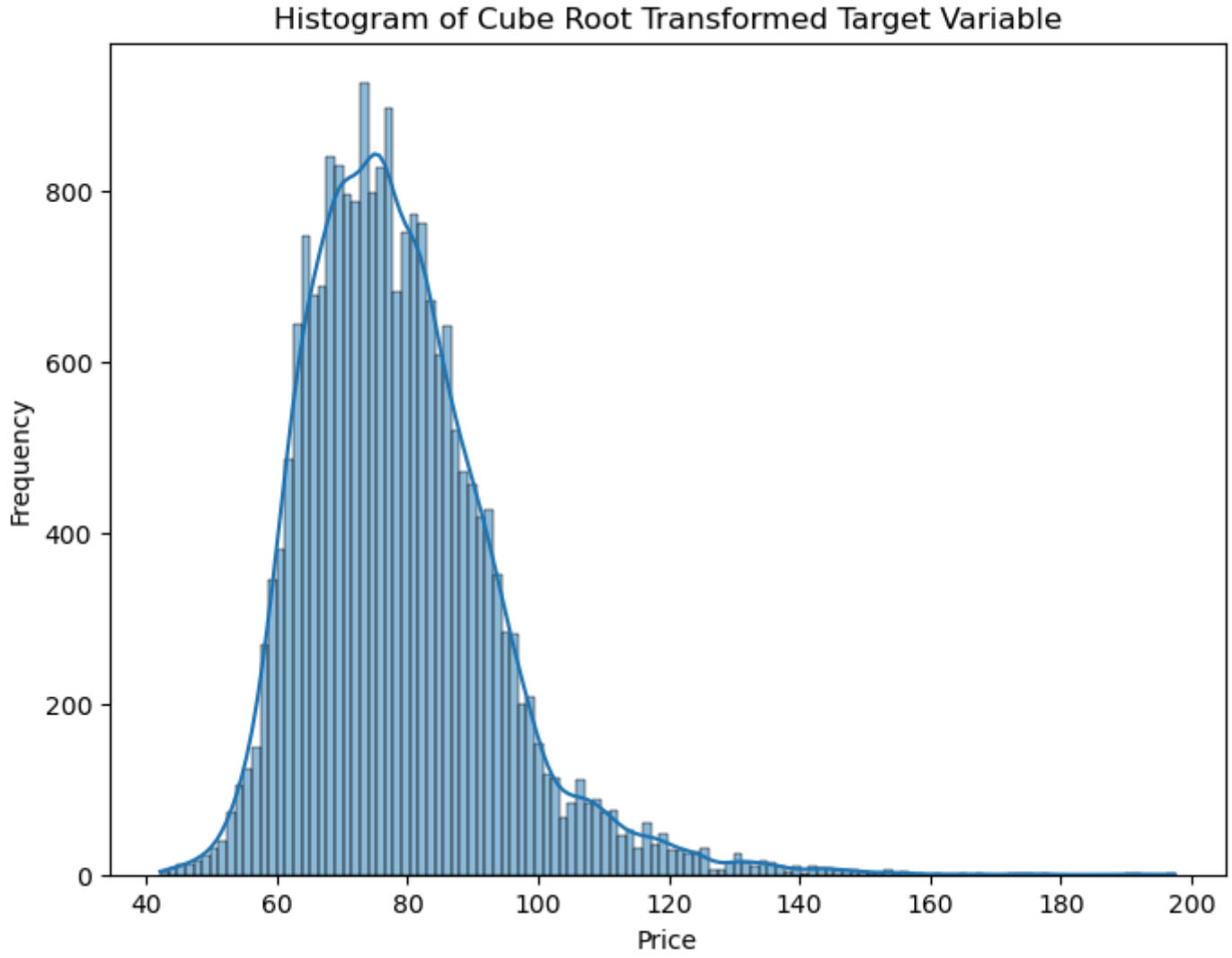
```
plt.xlabel('Price')
plt.ylabel('Frequency')
plt.show()
```



Cube Root Transformation

In [53]:
```
#Performing square root transormation on the price column
my_df_cbrt['price']=np.cbrt(my_df_cbrt['price'])
#Plotting the results
```

```
plt.figure(figsize=(8, 6))
sns.histplot(my_df_cbrt['price'], kde=True)
plt.title("Histogram of Cube Root Transformed Target Variable")
plt.xlabel('Price')
plt.ylabel('Frequency')
plt.show()
```



In [54]:
```
# Data scaling using MinMaxScaler
scaler = MinMaxScaler()
```

```
normalize_cols=['sqft_living','sqft_above','sqft_basement','sqft_living15','price','yr_built','zipcode','sqft_lot','sqft_lot15',
                'bedrooms','bathrooms','floors','grade','condition','lat_long_cluster','view','Year','Month','Day']
my_df_log[normalize_cols] = scaler.fit_transform(my_df_log[normalize_cols])

#Dropping unwanted features
my_df_log=my_df_log.drop(['Year', 'Month', 'Day'], axis=1)

#Features and response data variables for k-fold cross validation
feature_vars_2=my_df_log.drop(columns='price', axis=1)
response_var_2=my_df_log['price']
```

Parameter tuning

In [55]:
```
# Ingoning the warnings
warnings.filterwarnings("ignore")
# Descision Tree
model_DT_GridSearch(feature_vars_2,response_var_2)
#Random Forest
model_RF_GridSearch(feature_vars_2,response_var_2)
#SVM
model_SVM_GridSearch(feature_vars_2,response_var_2)
#KNN
model_KNN_GridSearch(feature_vars_2,response_var_2)
#GBM
model_GBM_GridSearch(feature_vars_2,response_var_2)
#XGB
model_XGB_GridSearch(feature_vars_2,response_var_2)
```

```
Decision Tree -> Best parameters :  {'max_depth': 9}
RF  -> Best parameters :  {'max_depth': 19, 'n_estimators': 20}
SVM -> Best parameters :  {'C': 1}
KNN -> Best parameters :  {'n_neighbors': 9}
GBM -> Best parameters :  {'n_estimators': 100}
XGB -> Best parameters :  {'n_estimators': 100}
```

Fitting and Evaluating models

In [56]:
```
# Ingoning the warnings
warnings.filterwarnings("ignore")
#Linear Regression
model_LR(feature_vars_2,response_var_2)
#Decision Tree
```

```
model_DT(feature_vars_2,response_var_2,max_depth=9)
#Random Forest
model_RF(feature_vars_2,response_var_2,n_estimators=20,max_depth=19)
#SVM
model_SVM(feature_vars_2,response_var_2,C=1)
#KNN
model_KNN(feature_vars_2,response_var_2,n_neighbors=9)
#GBM
model_GBM(feature_vars_2,response_var_2,n_estimators=100)
#XGB
model_XGB(feature_vars_2,response_var_2,n_estimators=100)
```

```
LR  -> R2 score :  0.73
DT  -> R2 score :  0.79
RF  -> R2 score :  0.85
SVM -> R2 score :  0.83
KNN -> R2 score :  0.81
GBM -> R2 score :  0.84
XGB -> R2 score :  0.88
```

After applying transformation techniques on the target variable, it is observed that the data transformed using log transformation is normally distributed compared to other techniques. The data transformed using square root and cube root techniques is skewed towards right. So, model training and prediction were performed on log transformed data. In previous analysis, the performance of Linear Regression, Decision Tree and KNN were not great. From above results, we can see there is an improvement in performance of these models and considerable amount of increase in r2 scores of other models as well. Overall, the models performance has improved compared to previous analysis.

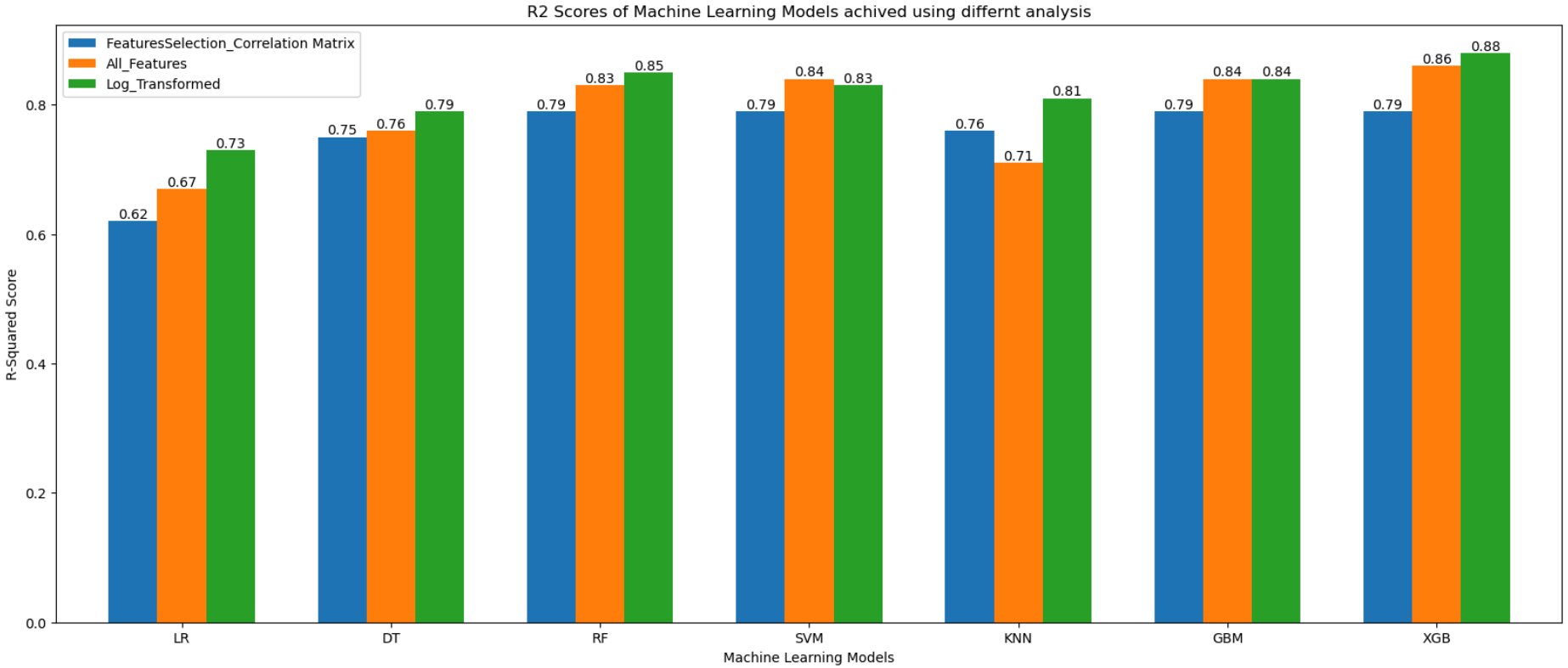
Based on all different analysis carried out above, it is observed that the models trained using data with all the input features and log transformed response variable performed better compared to others. **The best performing machine learning model for this house pricing dataset is Extreme Gradient Boosting Machines (XGB) algorithm with a R2 score of 0.88**.

Plotting results achieved with different analysis

In [57]:
```
#Creating the list of models
Models=['LR', 'DT', 'RF', 'SVM', 'KNN', 'GBM', 'XGB']
#Creating the list of r2 scores of models of different analysis
All_Features=[0.67,0.76,0.83,0.84,0.71,0.84,0.86]
Features_CorrelationMatrix=[0.62,0.75,0.79,0.79,0.76,0.79,0.79]
Log_Transformed=[0.73,0.79,0.85,0.83,0.81,0.84,0.88]
#Creating data frame
r2_scores_=pd.DataFrame({
```

```
    'FeaturesSelection_Correlation Matrix' : Features_CorrelationMatrix,
    'All_Features' : All_Features,
    'Log_Transformed' : Log_Transformed},
    index=Models
)

#Plotting the results
plt.figure(figsize = (20, 10))
ax=r2_scores_.plot(kind='bar', width=0.7, figsize=(20,8))

plt.xlabel("Machine Learning Models")
plt.ylabel("R-Squared Score")
plt.title("R2 Scores of Machine Learning Models achived using differnt analysis")
#Setting roration for x-axis values
plt.xticks(rotation=0)
for i in ax.containers:
    ax.bar_label(i,)
plt.show()
```

```
<Figure size 2000x1000 with 0 Axes>
```



Deep Learning (Neural Networks)

Deep learning is a part of machine learning. It has models which are based on Artificial Neuarl Networks(ANN) with multiple layers. The term deep refers to hidden layers present between input and output layers. These neural networks are capable of uncovering the complex patterns present in the data and learn from them. Every neural network has one input layer and output layer. To understand the complex patterns hidden layers are required and the complexity of the model increases with increase in number of hidden layers. Neural networks have many hyperparameters that needs to be tuned to increase model performance and avoid issues like overfitting and underfitting. The evaluation metrics are same as we use to evaluate machine learning models such as R-Squared score, MSE, MAE, F1 score, Precision, Recall, Accuracy depending on the problem statement. In this project a simple neural network was trained to make prediction on the unseen data. Evaluation metrics used are mean squared error(MSE), mean absolute error(MAE) and R-Squared score.

Hyperparameter tuning in deep learning makes a significant impact on the model performance. It involves finding optimal values of different parameters. Tuning the parameters needs to be done before the model training as different parameters performs well on different datasets. Right parameters need to be picked to let the model learn effectively and make predictions on unseen data accurately. But, neural networks has many parameters and tuning all the parameters takes a lot of time. So, to reduce the computational costs selected parameters were tuned in this project.

Optimizer - Optimizer in deep learning is used to fine tune the parameters during the training process. It is a crucial component in artificial neural networks. The main aim of optimizer is to minimise the loss during the training by improving the convergence speed . It updates the weights of inputs and biases throughout the training process iteratively to minimize the loss and improve the model performance based on the feedback received from the data. There are different types of optimizers available each one its own way of improving model performance. In this project Adam optimizer also called as Adaptive Moment Estimation optimizer was used. Because, it is the most widely used optimizer in deep learning tasks because of its robustness. It adjusts the learning rate dynamically based on the individual weights.

Data Splitting

In [58]:
```
#Dividing train data into train and test sets
x_train_2, x_test_2, y_train_2, y_test_2 = train_test_split(feature_vars_2,response_var_2,test_size=0.2,random_state=42)

#Shapes of the train and test data
print("Dim of x_train_2 : ",x_train_2.shape)
print("Dim of y_train_2 : ",y_train_2.shape)
print("Dim of x_test_2 : ",x_test_2.shape)
print("Dim of y_test_2 : ",y_test_2.shape)
```

```
Dim of x_train_2 :  (17290, 17)
Dim of y_train_2 :  (17290,)
Dim of x_test_2 :  (4323, 17)
Dim of y_test_2 :  (4323,)
```

Early Stopping Criteria

In [59]:
```
#Early stopping
tensorflow.keras.callbacks.Callback()
el=tensorflow.keras.callbacks.EarlyStopping(monitor='val_loss',patience=5)
```

Grid search for number of neurons

In neural networks, the number of neurons in each layers needs to be provided. Output layer neurons depends on the problem statement. For regression problems output layer has one neuron and for classification problems number layers depends on the number of classes in the response variable. There is no limitation for number of neurons in input and hidden layers. But, it is important to tune the number of neurons of neurons in the input layer. The computational cost of the model increases with increase in number of neurons. Parameter tuning helps to avoid this problem by finding the optimal number of neurons.

In [62]:
```
numOfNeurons=[17,34,51,68,136,272,544,1088,2176]
r2_scores={}
for i in range(len(numOfNeurons)):

    #Model Architecture
    model_NN= Sequential()
    model_NN.add(Dense(numOfNeurons[i],input_dim =17 ,activation="relu"))
    model_NN.add(Dense(1,activation = "sigmoid"))

    #Configure the model
    model_NN.compile(optimizer=Adam(learning_rate=0.0001),loss="mean_squared_error", metrics=["mean_absolute_error"])
    #Train the model
    history=model_NN.fit(x_train_2,y_train_2, validation_split=0.2,epochs=100,batch_size=16,callbacks=[el], verbose=0)
    #Predicting results
    pred = model_NN.predict(x_test_2)
    #R2 Score
    r2_scores[numOfNeurons[i]]=r2_score(y_test_2, pred)
print("R2 Scores : ",r2_scores)
max_r2=max(r2_scores.values())
r2_values = list(r2_scores.values())
indx=r2_values.index(max_r2)
print("\nMax R2 score is ",max(r2_scores.values()), " at number of neurons equal to",numOfNeurons[indx])
```

```
136/136 [==============================] - 0s 2ms/step
136/136 [==============================] - 0s 2ms/step
136/136 [==============================] - 0s 2ms/step
136/136 [==============================] - 0s 2ms/step
136/136 [==============================] - 0s 2ms/step
136/136 [==============================] - 0s 2ms/step
136/136 [==============================] - 0s 2ms/step
136/136 [==============================] - 0s 2ms/step
136/136 [==============================] - 0s 2ms/step
R2 Scores :  {17: 0.815689231840584, 34: 0.8182544479630147, 51: 0.8251973991894889, 68: 0.8230700157583792, 136: 0.829010448396
2838, 272: 0.8332615301134305, 544: 0.8339562920078656, 1088: 0.8325133296431709, 2176: 0.8270687354220834}

Max R2 score is  0.8339562920078656  at number of neurons equal to 544
```

Grid search for batch size

Batch size is a key hyperparameter in neural networks. It refers to the number of samples that are taken in one feedforward and backpropagation process during the model training. Tuning batch size is very important as it has a greater impact on training process. Model trains faster when the batch size is larger but sometimes larger batch size gives memory error and there is a possibility of overfitting. So, tuning batch size and finding optimal number of parameters plays a key role in model performance.

In [63]:
```
size=[8,16,32,64,128]
r2_scores={}
for i in range(len(size)):

    #Model Architecture
    model_NN= Sequential()
    model_NN.add(Dense(544,input_dim =17 ,activation="relu"))
    model_NN.add(Dense(1,activation = "sigmoid"))

    #Configure the model
    model_NN.compile(optimizer=Adam(learning_rate=0.0001),loss="mean_squared_error", metrics=["mean_absolute_error"])
    #Train the model
    history=model_NN.fit(x_train_2,y_train_2, validation_split=0.2,epochs=100,batch_size=size[i],callbacks=[el], verbose=0)
    #Predicting results
    pred = model_NN.predict(x_test_2)
    #R2 Score
    r2_scores[size[i]]=r2_score(y_test_2, pred)
print("R2 Scores : ",r2_scores)
max_r2=max(r2_scores.values())
r2_values = list(r2_scores.values())
```

```
indx=r2_values.index(max_r2)
print("\nMax R2 score is ",max(r2_scores.values()), " with batch size equal to",size[indx])
```

```
136/136 [==============================] - 0s 2ms/step
136/136 [==============================] - 0s 2ms/step
136/136 [==============================] - 0s 2ms/step
136/136 [==============================] - 0s 2ms/step
136/136 [==============================] - 0s 2ms/step
R2 Scores :  {8: 0.8365026749477438, 16: 0.8359804338124639, 32: 0.8298694747959509, 64: 0.8314538540731918, 128: 0.829127852526
0228}

Max R2 score is  0.8365026749477438  with batch size equal to 8
```

Grid search for learning rate

Learning rate in neural networks is used to control rate at which the model weights are updated and the rate at which the model learns during the training process. It is a key parameter of neural networks to optimize the model performance. It impacts the speed of the model learning during training. A higher learning rate helps the model to learn faster, but it might lead to overfitting. Conversely smaller learning rate may result in getting stuck in local minima. So, it is necessary to tune the learning rate to find the optimal value at which the model learns better while making predictions on unseen data.

In [64]:
```
#Tuning learning rate
rate=[0.0001, 0.0002, 0.0005, 0.0008, 0.001, 0.005, 0.01]
r2_scores={}
for i in range(len(rate)):

    #Model Architecture
    model_NN= Sequential()
    model_NN.add(Dense(544,input_dim =17 ,activation="relu"))
    model_NN.add(Dense(1,activation = "sigmoid"))

    #Configure the model
    model_NN.compile(optimizer=Adam(learning_rate=rate[i]),loss="mean_squared_error", metrics=["mean_absolute_error"])
    #Train the model
    history=model_NN.fit(x_train_2,y_train_2, validation_split=0.2,epochs=100,batch_size=8,callbacks=[el], verbose=0)
    #Predicting results
    pred = model_NN.predict(x_test_2)
    #R2 Score
    r2_scores[rate[i]]=r2_score(y_test_2, pred)
print("R2 Scores : ",r2_scores)
max_r2=max(r2_scores.values())
```

```
r2_values = list(r2_scores.values())
indx=r2_values.index(max_r2)
print("\nMax R2 score is ",max(r2_scores.values()), " with learning rate equal to",rate[indx])
```

```
136/136 [==============================] - 0s 2ms/step
136/136 [==============================] - 0s 1ms/step
136/136 [==============================] - 0s 2ms/step
136/136 [==============================] - 0s 2ms/step
136/136 [==============================] - 0s 2ms/step
136/136 [==============================] - 0s 2ms/step
136/136 [==============================] - 0s 2ms/step
R2 Scores :  {0.0001: 0.8367365141638851, 0.0002: 0.83080389238115, 0.0005: 0.8356886420375892, 0.0008: 0.8339675044265374, 0.00
1: 0.8352873723713439, 0.005: 0.8262057653170823, 0.01: 0.8018124242287239}

Max R2 score is  0.8367365141638851  with learning rate equal to 0.0001
```

Grid search for Activation Function

In neural networks, activation function is a key component, and it is also called as transfer function. It is a mathematical function used to calculate the output of each neuron. It gives the output as weighted sum of inputs and bias inputs. There are different types of activation functions available based on the problem statement such as ReLU, Sigmoid, Linear, SoftMax and Tanh. In regression problems, the most widely used activation functions in input and hidden layers are ReLU, Sigmoid, Tanh and in output layer Sigmoid is used. Sigmoid function outputs the value between 0 and 1. ReLU activation function is a simple non-linear function that can identify complex patterns in the data and learn from them. ReLU gives the x if the value is positive and 0 if it is negative. In this project grid search was performed to decide on which activation function to use.

In [65]:
```
#Hidden Layer activation function tuning
hidden_activation_func=['sigmoid', 'relu', 'tanh', 'linear']
r2_scores={}
for i in range(len(hidden_activation_func)):

    #Model Architecture
    model_NN= Sequential()
    model_NN.add(Dense(544,input_dim =17 ,activation=hidden_activation_func[i]))
    model_NN.add(Dense(1,activation = "sigmoid"))

    #Configure the model
    model_NN.compile(optimizer=Adam(learning_rate=0.0001),loss="mean_squared_error", metrics=["mean_absolute_error"])
    #Train the model
    history=model_NN.fit(x_train_2,y_train_2, validation_split=0.2,epochs=100,batch_size=8,callbacks=[el], verbose=0)
```

```
    #Predicting results
    pred = model_NN.predict(x_test_2)
    #R2 Score
    r2_scores[hidden_activation_func[i]]=r2_score(y_test_2, pred)
print("R2 Scores : ",r2_scores)
max_r2=max(r2_scores.values())
r2_values = list(r2_scores.values())
indx=r2_values.index(max_r2)
print("\nMax R2 score is ",max(r2_scores.values()), " with hidden layer activation function as ",hidden_activation_func[indx])
```

```
136/136 [==============================] - 0s 2ms/step
136/136 [==============================] - 0s 2ms/step
136/136 [==============================] - 0s 2ms/step
136/136 [==============================] - 0s 2ms/step
R2 Scores :  {'sigmoid': 0.732044126538575, 'relu': 0.8354723425413147, 'tanh': 0.714314010732988, 'linear': 0.7303593647427435}

Max R2 score is  0.8354723425413147  with hidden layer activation function as  relu
```

In [66]:

```
#Output layer activation function tuning
opt_activation_func=['linear', 'sigmoid']
r2_scores={}
for i in range(len(opt_activation_func)):

    #Model Architecture
    model_NN= Sequential()
    model_NN.add(Dense(544,input_dim =17 ,activation="relu"))
    model_NN.add(Dense(1,activation = opt_activation_func[i]))

    #Configure the model
    model_NN.compile(optimizer=Adam(learning_rate=0.0001),loss="mean_squared_error", metrics=["mean_absolute_error"])
    #Train the model
    history=model_NN.fit(x_train_2,y_train_2, validation_split=0.2,epochs=100,batch_size=8,callbacks=[el], verbose=0)
    #Predicting results
    pred = model_NN.predict(x_test_2)
    #R2 Score
    r2_scores[opt_activation_func[i]]=r2_score(y_test_2, pred)
print("R2 Scores : ",r2_scores)
max_r2=max(r2_scores.values())
r2_values = list(r2_scores.values())
indx=r2_values.index(max_r2)
print("\nMax R2 score is",max(r2_scores.values()), "with output layer activation function as",opt_activation_func[indx])
```

```
136/136 [==============================] - 1s 3ms/step
136/136 [==============================] - 0s 2ms/step
R2 Scores :  {'linear': 0.8252251675268769, 'sigmoid': 0.8355576168609908}

Max R2 score is 0.8355576168609908 with output layer activation function as sigmoid
```

It is observed from all the experiments conducted above that, the model with ReLU activation function in hidden layers and Sigmoid activation function in output layer performs better compared to others. The model performance is maximum with 544 neurons in hidden layers, with a batch size of 8 and models learning rate of 0.0001.

Neural Networks Final Model

In [67]:
```
#Model Architecture
model_NN= Sequential()
model_NN.add(Dense(544,input_dim =17 ,activation="relu"))
model_NN.add(Dense(1,activation = "sigmoid"))
#Summary
model_NN.summary()
```

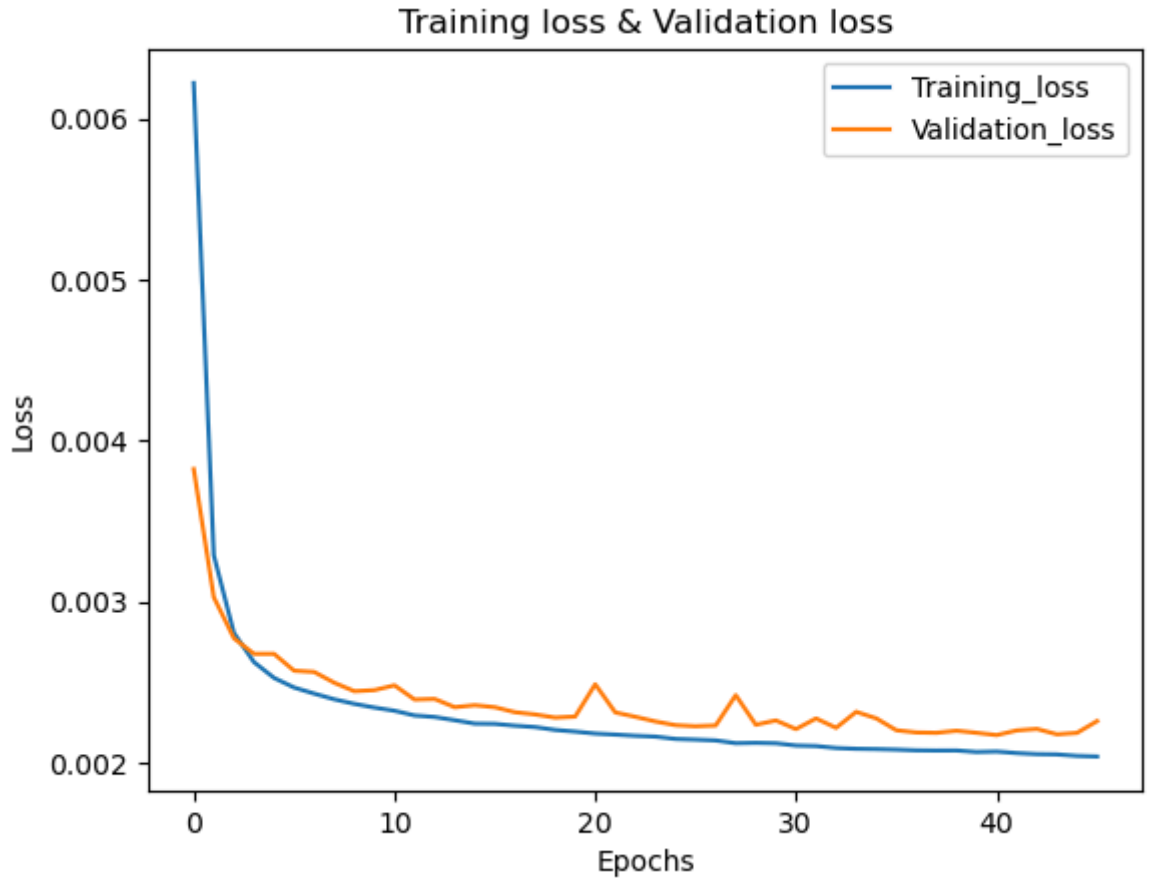
```
Model: "sequential_45"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_90 (Dense)            (None, 544)               9792

 dense_91 (Dense)            (None, 1)                 545

=================================================================
Total params: 10337 (40.38 KB)
Trainable params: 10337 (40.38 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

In [68]:
```
#Configure the model
model_NN.compile(optimizer=Adam(learning_rate=0.0001),loss="mean_squared_error", metrics=["mean_absolute_error"])
#Train the model
history=model_NN.fit(x_train_2,y_train_2, validation_split=0.2,epochs=100,batch_size=8,callbacks=[el], verbose=0)
#Use the model's evaluate method to predict and evaluate the test datasets
result = model_NN.evaluate(x_test_2,y_test_2)
```

```
136/136 [==============================] - 0s 3ms/step - loss: 0.0023 - mean_absolute_error: 0.0357
```

In [69]:
```
#Print the results
for i in range(len(model_NN.metrics_names)):
    print("Metric ",model_NN.metrics_names[i],":",str(round(result[i],4)))

#Ploting the results
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title("Training loss & Validation loss")
plt.ylabel('Loss')
plt.xlabel('Epochs')
plt.legend(['Training_loss', 'Validation_loss'], loc='upper right')
plt.show()
```

```
Metric  loss : 0.0023
Metric  mean_absolute_error : 0.0357
```



In [70]:
```
#Model prediction
pred = model_NN.predict(x_test_2)
```

```
136/136 [==============================] - 0s 2ms/step
```

In [72]:
```
# R2 Score
print("R2 Score:", round(r2_score(y_test_2, pred),2))
```

```
R2 Score: 0.83
```

The Neural Networks final model is trained and tested with tuned hyperparameters such as number of neurons, batch size, learning rate and activation functions. In hidden layer, relu activation function is used and in output layer sigmoid function is used as activation function. Mean Squared Error (MSE) is used as loss function and Mean Absolute Error(MAE) and R2 score are the evaluation metrics used to measure the performance of the model. Early stopping is implemented in the model to avoid the model from overfitting. The graph above shows the model's training loss vs validation loss. We can see from the plot that the difference between training loss and validation loss is very small which indicates that the model is not overfitting. Predictions were made on unseen data using the model trained with the above mentioned tuned hyperparameters and it performed better for the chosen house pricing data with 0.0023 loss and R2 score of 0.83.

Conclusion

Predicting the house prices accurately based on the input features is an essential task as it helps different groups of people in the community. The data analysis and machine learning modelling carried out in this project has helped to successfully achieve the task of predicting house prices. Different data preprocessing techniques were used to prepare the data that is suitable to train machine learning and deep learning models and make predictions on unseen data. Machine learning algorithms such as Linear Regression, Decision Tree, KNN, Random Forest, SVM, GBM and XGB were used to perform analysis on the chosen dataset. The hyperparameters of each model were tuned to maximize the model performance. Models were evaluated using R2 score evaluation metric. The results of the machine learning models indicate that XGB model performance was better compared to other models for this specific dataset with a R-Squared score of 0.88. Data cleaning and data transformation has helped in improving model's overall performance. Models like KNN, SVM, GBM and Random Forest have also performed better with scores of 0.81, 0.83, 0.84 and 0.85 respectively. The analysis was further carried out using deep learning. A simple Artificial Neural Network (ANN) was used to predict the house prices. The hyperparameters of neural network were tuned to maximize the model performance and it made good predictions on the unseen data with a R-Squared score of 0.83. In conclusion, the objective of predicting the house prices was achieved in this project by making use of different machine learning and deep learning models with well pre-processed data. The best performing model for this house pricing dataset is Extreme Gradient Boosting Machines (XGB) algorithm with a R-Squared score of 0.88.

References

[1]. https://www.analyticsvidhya.com/blog/2021/06/defining-analysing-and-implementing-imputation-techniques/

[2]. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

[3]. https://machinelearningmastery.com/how-to-configure-k-fold-cross-validation/

[4]. https://medium.com/@khadijamahanga/using-latitude-and-longitude-data-in-my-machine-learning-problem-541e2651e08c