



Property Price Prediction

Problem Statement

A key challenge for property sellers is to determine the sale price of the property. The ability to predict the exact property value is beneficial for property investors as well as for buyers to plan their finances according to the price trend. The property prices depend on the number of features like the property area, basement square footage, year built, number of bedrooms, and so on. Regression analysis can be useful in predicting the price of the house.

Data Definition

Dwell_Type: Identifies the type of dwelling involved in the sale

Zone_Class: Identifies the general zoning classification of the sale

LotFrontage: Linear feet of street-connected to the property

LotArea: Lot size is the lot or parcel side where it adjoins a street, boulevard or access way

Road_Type: Type of road access to the property

Alley: Type of alley access to the property

Property_Shape: General shape of the property

LandContour: Flatness of the property

LotConfig: Lot configuration

LandSlope: Slope of property

Neighborhood: Physical locations within Ames city limits

Condition1: Proximity to various conditions

Condition2: Proximity to various conditions (if more than one is present)

Dwelling_Type: Type of dwelling

HouseStyle: Style of dwelling

OverallQual: Rates the overall material and finish of the house

OverallCond: Rates the overall condition of the house

YearBuilt: Original construction date

YearRemodAdd: Remodel date (same as construction date if no remodeling or additions)

RoofStyle: Type of roof

RoofMatl: Roof material

Exterior1st: Exterior covering on the house

Exterior2nd: Exterior covering on the house (if more than one material)

MasVnrType: Masonry veneer type

MasVnrArea: Masonry veneer area in square feet

ExterQual: Evaluates the quality of the material on the exterior

ExterCond: Evaluates the present condition of the material on the exterior

Foundation: Type of foundation

BsmtQual: Evaluates the height of the basement

BsmtCond: Evaluates the general condition of the basement

BsmtExposure: Refers to walkout or garden level walls

BsmtFinType1: Rating of basement finished area

BsmtFinSF1: Type 1 finished square feet

BsmtFinType2: Rating of basement finished area (if multiple types)

BsmtFinSF2: Type 2 finished square feet

BsmtUnfSF: Unfinished square feet of the basement area

TotalBsmtSF: Total square feet of the basement area

Heating: Type of heating

HeatingQC: Heating quality and condition

CentralAir: Central air conditioning

Electrical: Electrical system

1stFlrSF: First floor square feet

2ndFlrSF: Second floor square feet

LowQualFinSF: Low quality finished square feet (all floors)

GrLivArea: Above grade (ground) living area square feet

BsmtFullBath: Basement full bathrooms

BsmtHalfBath: Basement half bathrooms

FullBath: Full bathrooms above grade

HalfBath: Half baths above grade

Bedroom: Bedrooms above grade (does NOT include basement bedrooms)

Kitchen: Kitchens above grade

KitchenQual: Kitchen quality

TotRmsAbvGrd: Total rooms above grade (does not include bathrooms)

Functional: Home functionality (Assume typical unless deductions are warranted)

Fireplaces: Number of fireplaces

FireplaceQu: Fireplace quality

GarageType: Garage location

GarageYrBlt: Year garage was built

GarageFinish: Interior finish of the garage

GarageCars: Size of garage in car capacity

GarageArea: Size of garage in square feet

GarageQual: Garage quality

GarageCond: Garage condition

PavedDrive: Paved driveway

WoodDeckSF: Wood deck area in square feet

OpenPorchSF: Open porch area in square feet

EnclosedPorch: Enclosed porch area in square feet

3SsnPorch: Three season porch area in square feet

ScreenPorch: Screen porch area in square feet

PoolArea: Pool area in square feet

PoolQC: Pool quality

Fence: Fence quality

MiscFeature: Miscellaneous feature not covered in other categories

MiscVal: Value of miscellaneous feature

MoSold: Month Sold (MM)

YrSold: Year Sold (YYYY)

SaleType: Type of sale

SaleCondition: Condition of sale

Icon Legend



Inferences from outcome



Additional Reads



Lets do it



Quick Tips

Table of Contents

1. [Import Libraries](#)
2. [Set Options](#)
3. [Read Data](#)
4. [Prepare and Analyze the Data](#)
 - 4.1 - [Understand the Data](#)
 - 4.1.1 - [Data Type](#)
 - 4.1.2 - [Summary Statistics](#)
 - 4.1.3 - [Distribution of Variables](#)
 - 4.1.4 - [Discover Outliers](#)
 - 4.1.5 - [Missing Values](#)
 - 4.1.6 - [Correlation](#)
 - 4.1.7 - [Analyze Relationships Between Target and Categorical Variables](#)
 - 4.2 - [Data Preparation](#)
 - 4.2.1 - [Check for Normality](#)
 - 4.2.2 - [Dummy Encode the Categorical Variables](#)
5. [Linear Regression \(OLS\)](#)
 - 5.1 - [Multiple Linear Regression Full Model with Log Transformed Dependent Variable \(OLS\)](#)
 - 5.2 - [Multiple Linear Regression Full Model without Log Transformed Dependent Variable \(OLS\)](#)
 - 5.3 - [Feature Engineering](#)
 - 5.3.1 - [Multiple Linear Regression \(Using New Feature\) - 1](#)
 - 5.3.2 - [Multiple Linear Regression \(Using New Feature\) - 2](#)
6. [Feature Selection](#)
 - 6.1 - [Variance Inflation Factor](#)
7. [Conclusion and Interpretation](#)

1. Import Libraries

```
In [1]: # We use 'Numpy' for mathematical operations on large, multi-dimensional arrays and matrices
# 'Pandas' is used for data manipulation and analysis
import numpy as np
import pandas as pd

# To check the data type we import 'is_string_dtype' and 'is_numeric_dtype'
from pandas.api.types import is_string_dtype
from pandas.api.types import is_numeric_dtype

# 'Matplotlib' is a data visualization library for 2D and 3D plots
import matplotlib.pyplot as plt
%matplotlib inline

# seaborn is used for plotting statistical graphics
import seaborn as sns

# To build and analyze various statistical models we use 'Statsmodels'
import statsmodels
import statsmodels.api as sm
from statsmodels.compat import lzip
import statsmodels.stats.api as sms
import statsmodels.formula.api as smf
from statsmodels.formula.api import ols
from statsmodels.tools.eval_measures import rmse
from statsmodels.stats.outliers_influence import variance_inflation_factor

# 'Scikit-Learn' (sklearn) emphasizes various regression, classification and clustering algorithms
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.linear_model import LinearRegression
from sklearn import metrics
from sklearn import preprocessing

# To perform scientific computations
from scipy.stats import shapiro
from scipy import stats
```

```
In [2]: # suppress the warnings
from warnings import filterwarnings
filterwarnings('ignore')

# set the plot size using 'rcParams'
# once the plot size is set using 'rcParams', it sets the size of all the forthcoming plots in the
# 15 and 8 are width and height in inches respectively
plt.rcParams['figure.figsize'] = [10,6]
```

2. Set Options



1. Display complete data frames
2. To avoid the exponential number

```
In [3]: # display all columns of the dataframe
pd.options.display.max_columns = None

# display all rows of the dataframe
pd.options.display.max_rows = None

# use below code to convert the 'exponential' values to float
np.set_printoptions(suppress=True)
```

3. Read Data

```
In [4]: # read csv file using pandas
df_property = pd.read_csv('HousePrices.csv')

# display the top 5 rows of the dataframe
df_property.head()
```

Out[4]:

	Id	Dwell_Type	Zone_Class	LotFrontage	LotArea	Road_Type	Alley	Property_Shape	LandContour	Utilities	LotConfig	
0	1	60	RL	65.0	8450	Pave	NaN		Reg	Lvl	AllPub	Inside
1	2	20	RL	80.0	9600	Pave	NaN		Reg	Lvl	AllPub	FR:
2	3	60	RL	68.0	11250	Pave	NaN		IR1	Lvl	AllPub	Inside
3	4	70	RL	60.0	9550	Pave	NaN		IR1	Lvl	AllPub	Corne
4	5	60	RL	84.0	14260	Pave	NaN		IR1	Lvl	AllPub	FR:

Lets take a glance at our dataframe and see how it looks

Dimensions of the data

```
In [5]: # 'shape' function returns a tuple that gives the total number of rows and columns in the data
df_property.shape
```

Out[5]: (1460, 81)

4. Data Analysis and Preparation

Data preparation is the process of cleaning and transforming raw data before building predictive models.

Here, we analyze and perform the following tasks:

1. Check data types. Ensure your data types are correct.
2. We need to change the data types as per requirement if they are not as per business definition
3. Go through the summary statistics
4. Distribution of variables
5. Study the correlation
6. Detect outliers from the data
7. Look for the missing values

4.1 Understand the Dataset

4.1.1 Data Type

The data types in pandas dataframes are the object, float, int64, bool, and datetime64. We should know the data type of each column.



In our dataset, we have a blend of numerical and categorical variables. The numeric variables should have data type 'int'/'float' while categorical variables should have data type 'object'.

1. Check for the data type

```
In [6]: # 'dtypes' provides the data type for each column  
df_property.dtypes
```

www.linkedin.com/in/lajantha-devi-vairamani

```
Out[6]: Id                      int64
Dwell_Type                 int64
Zone_Class                  object
LotFrontage                 float64
LotArea                     int64
Road_Type                   object
Alley                       object
Property_Shape               object
LandContour                 object
Utilities                     object
LotConfig                    object
LandSlope                     object
Neighborhood                object
Condition1                  object
Condition2                  object
Dwelling_Type                object
HouseStyle                   object
OverallQual                  int64
OverallCond                  int64
YearBuilt                     int64
YearRemodAdd                  int64
RoofStyle                     object
RoofMatl                     object
Exterior1st                  object
Exterior2nd                  object
MasVnrType                   object
MasVnrArea                   float64
ExterQual                     object
ExterCond                     object
Foundation                   object
BsmtQual                     object
BsmtCond                     object
BsmtExposure                  object
BsmtFinType1                  object
BsmtFinSF1                   int64
BsmtFinType2                  object
BsmtFinSF2                   int64
BsmtUnfSF                     int64
TotalBsmtSF                   int64
Heating                       object
HeatingQC                     object
CentralAir                    object
Electrical                    object
1stFlrSF                     int64
2ndFlrSF                     int64
LowQualFinSF                  int64
GrLivArea                     int64
BsmtFullBath                  int64
BsmtHalfBath                  int64
FullBath                      int64
HalfBath                      int64
BedroomAbvGr                  int64
KitchenAbvGr                  int64
KitchenQual                   object
TotRmsAbvGrd                  int64
Functional                     object
Fireplaces                     int64
FireplaceQu                  object
GarageType                     object
GarageYrBlt                   float64
GarageFinish                   object
GarageCars                     int64
GarageArea                     int64
GarageQual                     object
GarageCond                     object
PavedDrive                    object
WoodDeckSF                     int64
OpenPorchSF                     int64
EnclosedPorch                  int64
3SsnPorch                     int64
ScreenPorch                    int64
PoolArea                      int64
PoolQC                        object
Fence                          object
MiscFeature                    object
MiscVal                        int64
```

```

MoSold           int64
YrSold          int64
SaleType         object
SaleCondition    object
Property_Sale_Price   int64
dtype: object

```

From the above output, we can see that 'Dwell_Type', 'OverallQual' and 'OverallCond' have data type as 'int64'.



But as per the data definition, 'Dwell_Type', 'OverallQual' and 'OverallCond' are categorical variables, so we need to convert these variables data type to 'object'.

Let us convert 'Dwell_Type', 'OverallQual' and 'OverallCond' to categorical data type

```
In [7]: df_property['Dwell_Type'] = df_property['Dwell_Type'].astype('O')
df_property['OverallQual'] = df_property['OverallQual'].astype('O')
df_property['OverallCond'] = df_property['OverallCond'].astype('O')
```

```
In [8]: # 'dtypes' provides the data type for each column
df_property[['Dwell_Type', 'OverallQual', 'OverallCond']].dtypes
```

```
Out[8]: Dwell_Type      object
OverallQual     object
OverallCond     object
dtype: object
```

Let us now remove the Id column as this will not be necessary for our analysis

```
In [9]: df_property.drop(['Id'], axis=1, inplace=True)
```

```
In [10]: df_property.columns
```

```
Out[10]: Index(['Dwell_Type', 'Zone_Class', 'LotFrontage', 'LotArea', 'Road_Type',
       'Alley', 'Property_Shape', 'LandContour', 'Utilities', 'LotConfig',
       'LandSlope', 'Neighborhood', 'Condition1', 'Condition2',
       'Dwelling_Type', 'HouseStyle', 'OverallQual', 'OverallCond',
       'YearBuilt', 'YearRemodAdd', 'RoofStyle', 'RoofMatl', 'Exterior1st',
       'Exterior2nd', 'MasVnrType', 'MasVnrArea', 'ExterQual', 'ExterCond',
       'Foundation', 'BsmtQual', 'BsmtCond', 'BsmtExposure', 'BsmtFinType1',
       'BsmtFinSF1', 'BsmtFinType2', 'BsmtFinSF2', 'BsmtUnfSF', 'TotalBsmtSF',
       'Heating', 'HeatingQC', 'CentralAir', 'Electrical', '1stFlrSF',
       '2ndFlrSF', 'LowQualFinSF', 'GrLivArea', 'BsmtFullBath', 'BsmtHalfBath',
       'FullBath', 'HalfBath', 'BedroomAbvGr', 'KitchenAbvGr', 'KitchenQual',
       'TotRmsAbvGrd', 'Functional', 'Fireplaces', 'FireplaceQu', 'GarageType',
       'GarageYrBlt', 'GarageFinish', 'GarageCars', 'GarageArea', 'GarageQual',
       'GarageCond', 'PavedDrive', 'WoodDeckSF', 'OpenPorchSF',
       'EnclosedPorch', '3SsnPorch', 'ScreenPorch', 'PoolArea', 'PoolQC',
       'Fence', 'MiscFeature', 'MiscVal', 'MoSold', 'YrSold', 'SaleType',
       'SaleCondition', 'Property_Sale_Price'],
      dtype='object')
```

4.1.2 Summary Statistics

Here we take a look at the summary of each attribute. This includes the count, mean, the minimum and maximum values as well as some percentiles for numeric variables and count, unique, top, frequency for categorical variables.



In our dataset we have numerical as well as categorical variables. Now we check for summary statistics of all the variables.

1. For getting the statistical summary of numerical variables we use the describe()

```
In [11]: # by default the describe function returns the summary of numerical variables
df_property.describe()
```

Out[11]:

	LotFrontage	LotArea	YearBuilt	YearRemodAdd	MasVnrArea	BsmtFinSF1	BsmtFinSF2	BsmtUnfSF	Tot
count	1201.000000	1460.000000	1460.000000	1460.000000	1452.000000	1460.000000	1460.000000	1460.000000	1460.000000
mean	70.049958	10516.828082	1971.267808	1984.865753	103.685262	443.639726	46.549315	567.240411	10:
std	24.284752	9981.264932	30.202904	20.645407	181.066207	456.098091	161.319273	441.866955	4:
min	21.000000	1300.000000	1872.000000	1950.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	59.000000	7553.500000	1954.000000	1967.000000	0.000000	0.000000	0.000000	223.000000	7:
50%	69.000000	9478.500000	1973.000000	1994.000000	0.000000	383.500000	0.000000	477.500000	9:
75%	80.000000	11601.500000	2000.000000	2004.000000	166.000000	712.250000	0.000000	808.000000	12:
max	313.000000	215245.000000	2010.000000	2010.000000	1600.000000	5644.000000	1474.000000	2336.000000	61

The above output displays the summary statistics of all the numeric variables like mean, median, standard deviation, minimum, and the maximum values, the first and third quantiles.



We can see that the LotFrontage ranges from 21 feet to 313 feet, with mean 70 feet. We can see that the minimum pool area is 0 sq.ft. and this means that not all houses have pools and yet have been considered to calculate the mean pool area. Also the count for LotFrontage is less than the total number of observations which indicates the presence of missing values.

2. For getting the statistical summary of categorical features we use the describe(include = object)

```
In [12]: # summary of categorical variables
df_property.describe(include = 'object')
```

Out[12]:

	Dwell_Type	Zone_Class	Road_Type	Alley	Property_Shape	LandContour	Utilities	LotConfig	LandSlope	Neighb
count	1460	1460	1460	91	1460	1460	1460	1460	1460	1460
unique	15	5	2	2	4	4	2	5	3	
top	20	RL	Pave	Grvl	Reg	Lvl	AllPub	Inside	Gtl	
freq	536	1151	1454	50	925	1311	1459	1052	1382	

```
In [13]: df_property["Dwell_Type"].value_counts().count()
```

Out[13]: 15

The summary statistics for categorical variables contains information about the total number of observations, number of unique classes, the most occurring class, and its frequency.:

Lets understand the outputs of the above table using variable 'Property_Shape'

count: Number of observations = 1460

unique: Number of unique classes in the column = 4 classes

top: The most occurring class = Reg

frequency: Frequency of the most repeated class; out of 1460 observations Reg has a frequency of 925

It is visible that some of the variables have count less than total number of observations which indicates the presence of missing values.

As, the variable PoolQC has only 7 non-zero values out of 1460 observations. And also the variable PoolArea contains the area of these 7 pools, we will remove the variables PoolQC and PoolArea .

```
In [14]: # use drop() to drop the redundant variables  
# 'axis = 1' drops the corresponding columns  
df_property = df_property.drop(['PoolQC', 'PoolArea'], axis= 1)  
  
# re-check the shape of the dataframe  
df_property.shape
```

Out[14]: (1460, 78)

4.1.3 Distribution of Variables



Check the distribution of all the variables.

1. Distribution of numeric variables

We plot the histogram to check the distribution of the variables.

```
In [15]: # filter the numerical features in the dataset using select_dtypes()  
# include=np.number is used to select the numeric features  
df_numeric_features = df_property.select_dtypes(include=np.number)
```

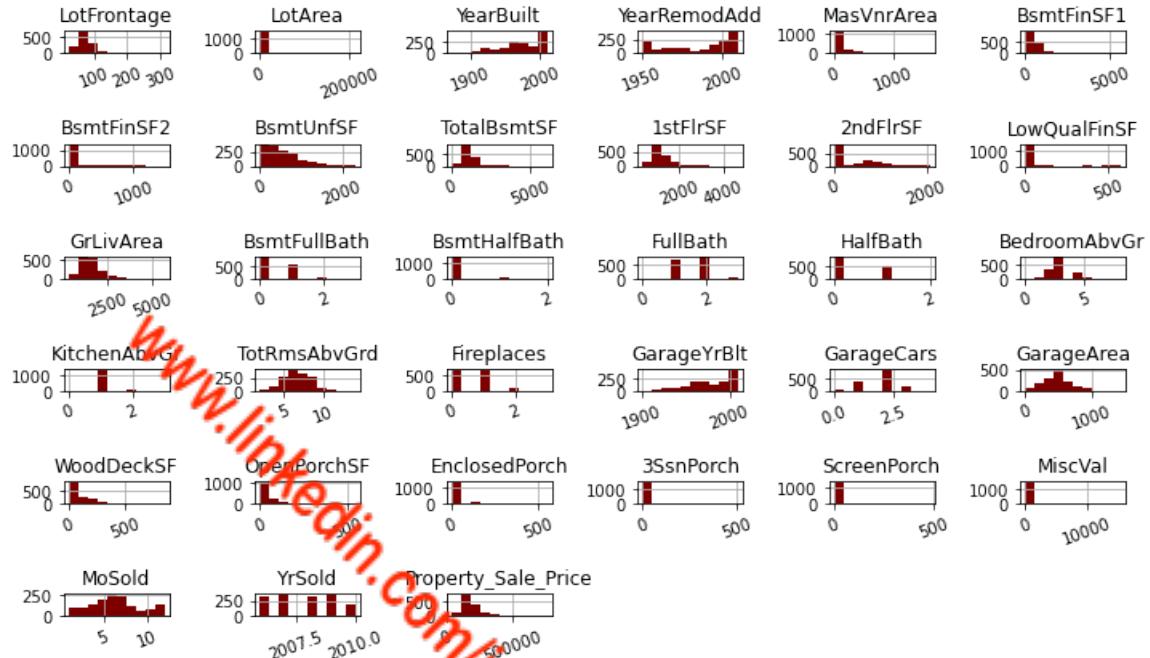
```
# display the numeric features  
df_numeric_features.columns
```

```
Out[15]: Index(['LotFrontage', 'LotArea', 'YearBuilt', 'YearRemodAdd', 'MasVnrArea',  
    'BsmtFinSF1', 'BsmtFinSF2', 'BsmtUnfSF', 'TotalBsmtSF', '1stFlrSF',  
    '2ndFlrSF', 'LowQualFinSF', 'GrLivArea', 'BsmtFullBath', 'BsmtHalfBath',  
    'FullBath', 'HalfBath', 'BedroomAbvGr', 'KitchenAbvGr', 'TotRmsAbvGrd',  
    'Fireplaces', 'GarageYrBlt', 'GarageCars', 'GarageArea', 'WoodDeckSF',  
    'OpenPorchSF', 'EnclosedPorch', 'ScreenPorch', 'ScreenPorch', 'MiscVal',  
    'MoSold', 'YrSold', 'Property_Sale_Price'],  
    dtype='object')
```

```
In [16]: # plot the histogram of numeric variables
# hist() by default considers the numeric variables only,
# rotate the x-axis labels by 20 degree using the parameter, 'xrot'
df_property.hist(xrot = 20, color = "maroon")

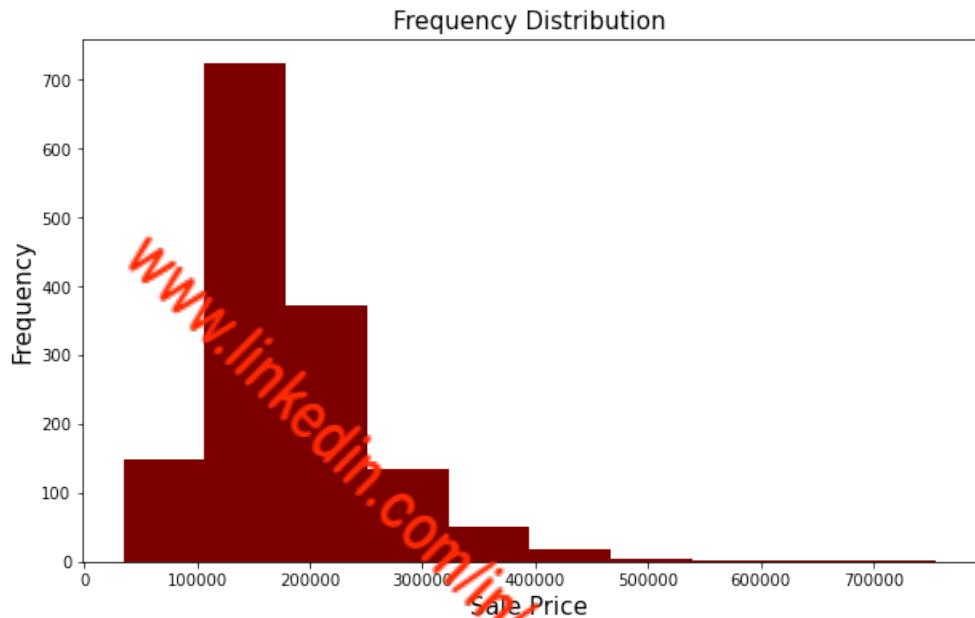
# adjust the subplots
plt.tight_layout()

# display the plot
plt.show()
```



Visualize the target variable

```
In [17]: # Sale Price Frequency Distribution  
# set the xlabel and the fontsize  
plt.xlabel("Sale Price", fontsize=15)  
  
# set the ylabel and the fontsize  
plt.ylabel("Frequency", fontsize=15)  
  
# set the title of the plot  
plt.title("Frequency Distribution", fontsize=15)  
  
# plot the histogram for the target variable  
plt.hist(df_property["Property_Sale_Price"], color = 'maroon')  
plt.show()
```



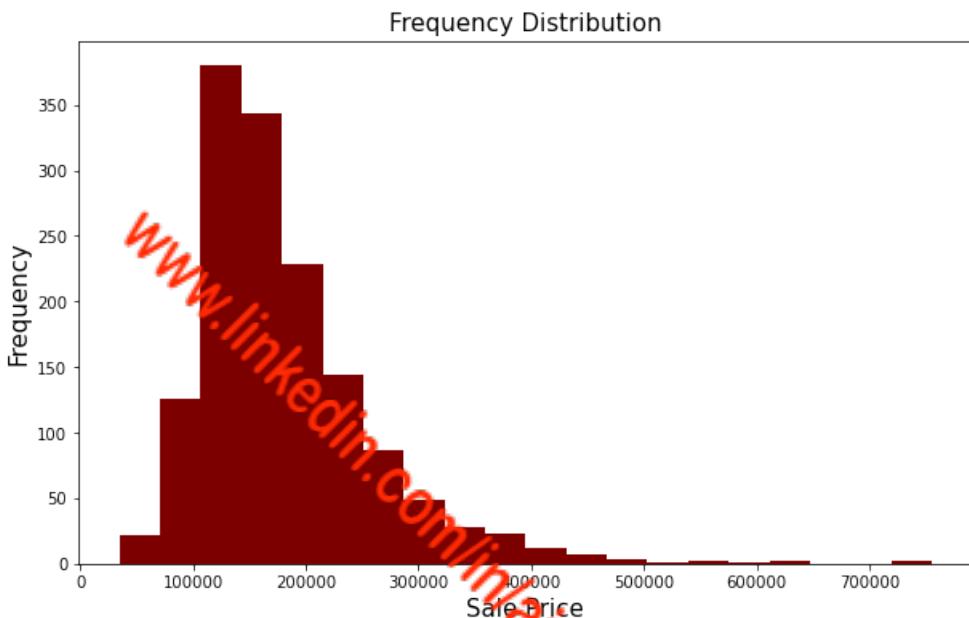
www.linkedin.com/in/vijayantha-devi-vairamani

```
In [18]: # Sale Price Frequency Distribution
#Tableau bins are containers of equal size that store data values corresponding to or fitting in bins
# set the xlabel and the fontsize
plt.xlabel("Sale Price", fontsize=15)

# set the ylabel and the fontsize
plt.ylabel("Frequency", fontsize=15)

# set the title of the plot
plt.title("Frequency Distribution", fontsize=15)

# plot the histogram for the target variable
plt.hist(df_property["Property_Sale_Price"], color = 'maroon',bins=20)
plt.show()
```



The above plot shows that the target variable 'Property_Sale_Price' is right skewed (right tailed).

2. Distribution of categorical variables

For the categoric variables, we plot the countplot (Uni-Variate Analysis)

```
In [19]: # create an empty list to store all the categorical variables
categorical=[]

# check the data type of each variable
for column in df_property:

    # check if the variable has the categorical type
    if is_string_dtype(df_property[column]):

        # append the categorical variables to the list 'categorical'
        categorical.append(column)

# plot the count plot for each categorical variable
# 'figsize' sets the figure size
fig, ax = plt.subplots(nrows=7, ncols=6, figsize = (50, 35))

# plot the count plot using countplot() for each categorical variable
for variable, subplot in zip(categorical, ax.flatten()):
    sns.countplot(df_property[variable], ax = subplot)

# display the plot
plt.show()
```



Boxplot of OverallQuality and Property_Sale_Price (Bi-Variate Analysis)

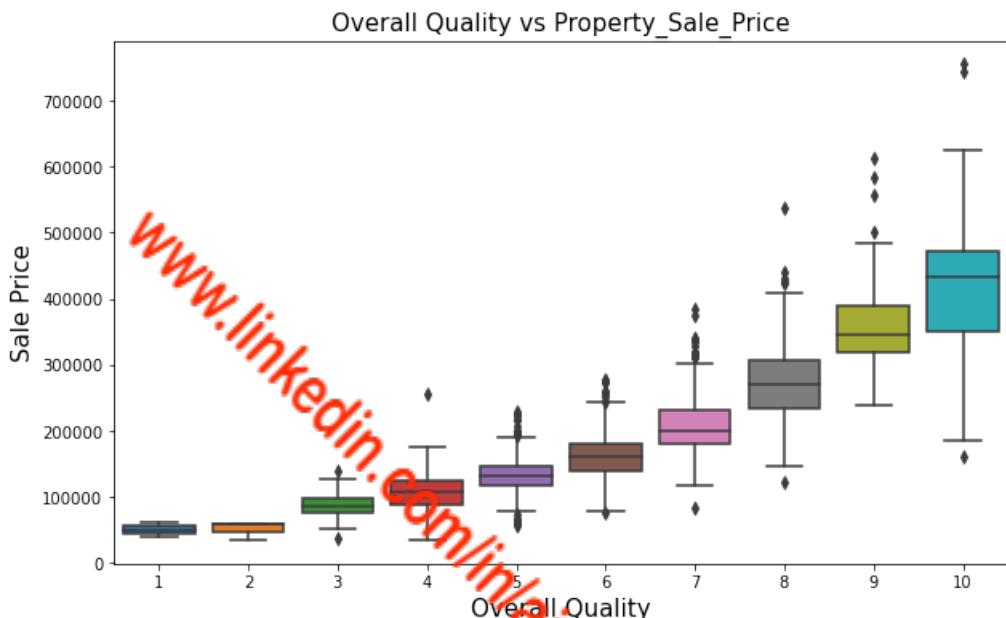
```
In [20]: # draw the boxplot for OverallQuality and the Property_Sale_Price
sns.boxplot(y="Property_Sale_Price", x="OverallQual", data= df_property)

# set the title of the plot and the fontsize
plt.title("Overall Quality vs Property_Sale_Price", fontsize=15)

# set the xlabel and the fontsize
plt.xlabel("Overall Quality", fontsize=15)

# set the ylabel and the fontsize
plt.ylabel("Sale Price", fontsize=15)

# display the plot
plt.show()
```



Boxplot of Overall Condition and Property_Sale_Price

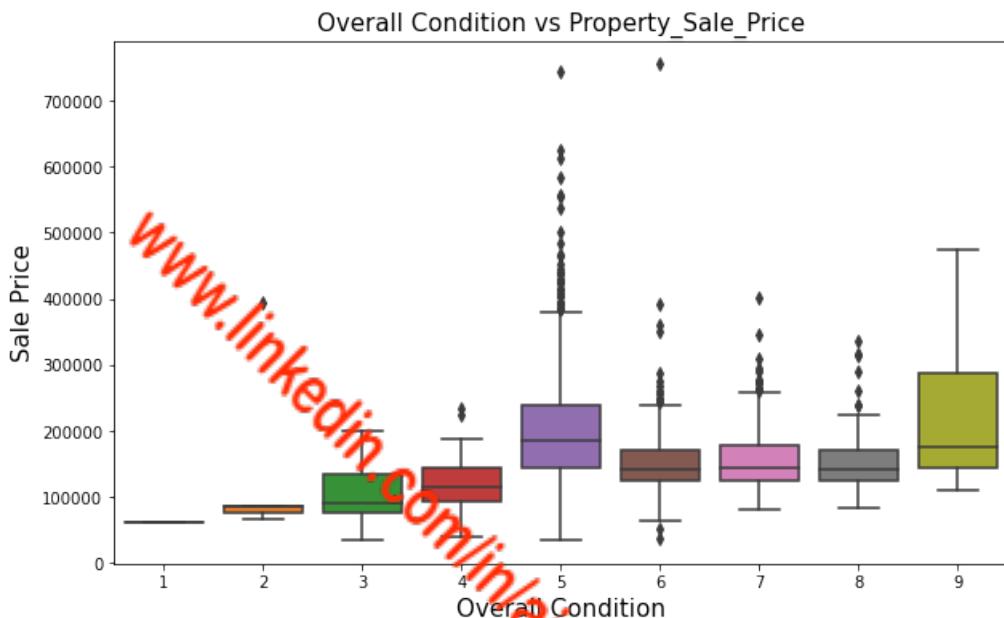
```
In [21]: # draw the boxplot for OverallQuality and the Property_Sale_Price
sns.boxplot(y="Property_Sale_Price", x="OverallCond", data= df_property)

# set the title of the plot and the fontsize
plt.title("Overall Condition vs Property_Sale_Price", fontsize=15)

# set the xlabel and the fontsize
plt.xlabel("Overall Condition", fontsize=15)

# set the ylabel and the fontsize
plt.ylabel("Sale Price", fontsize=15)

# display the plot
plt.show()
```



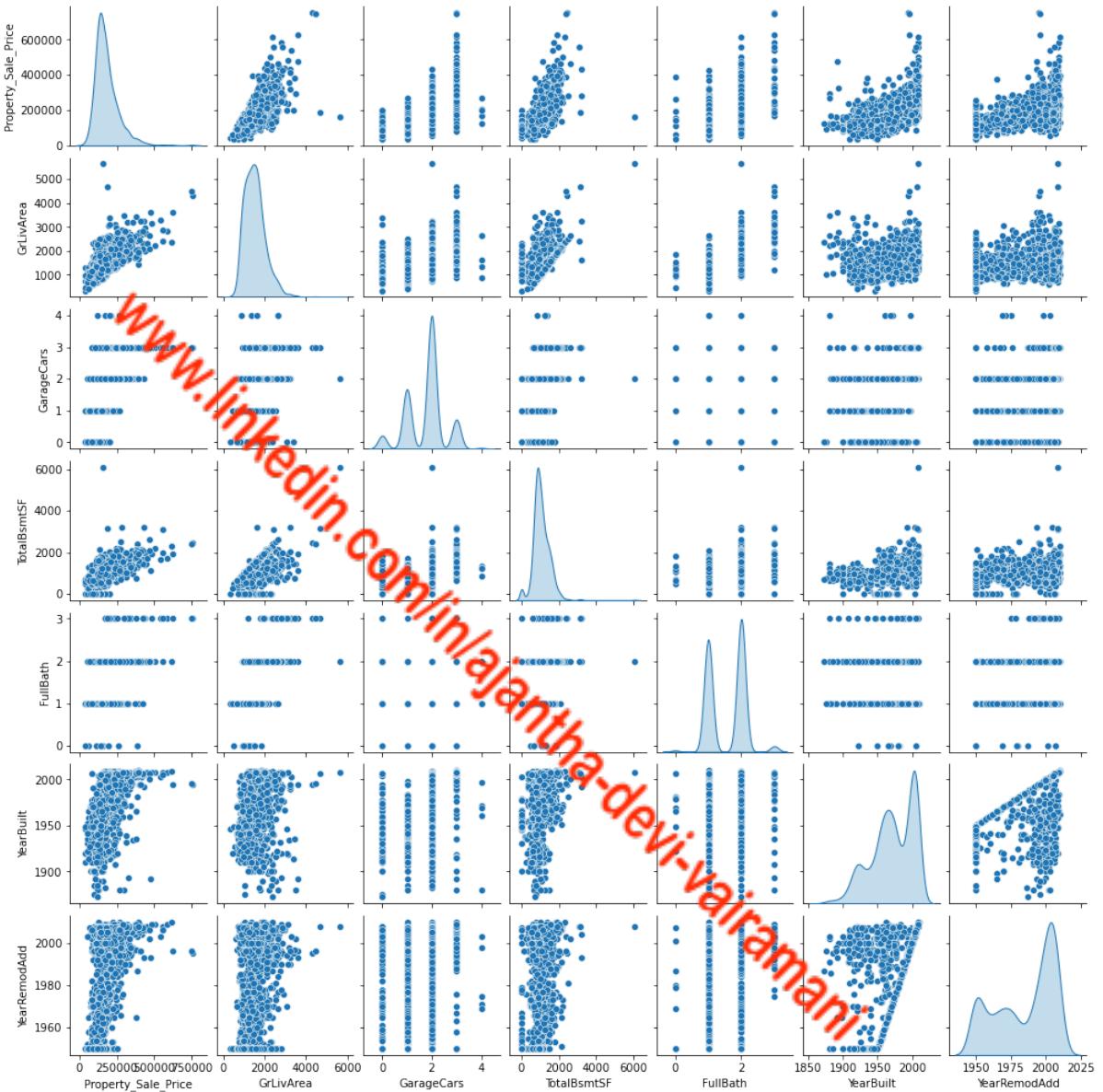
Draw the pairplot of the numeric variables

```
In [22]: # Pairplot of numeric variables
```

```
# select the columns for the pairplot
columns= ["Property_Sale_Price", "GrLivArea", "GarageCars", "TotalBsmtSF", "FullBath", "YearBuilt"]

# draw the pairplot such that the diagonal should be density plot and the other graphs should be scatter
sns.pairplot(df_property[columns], size=2, kind= "scatter", diag_kind="kde")

# display the plot
plt.show()
```

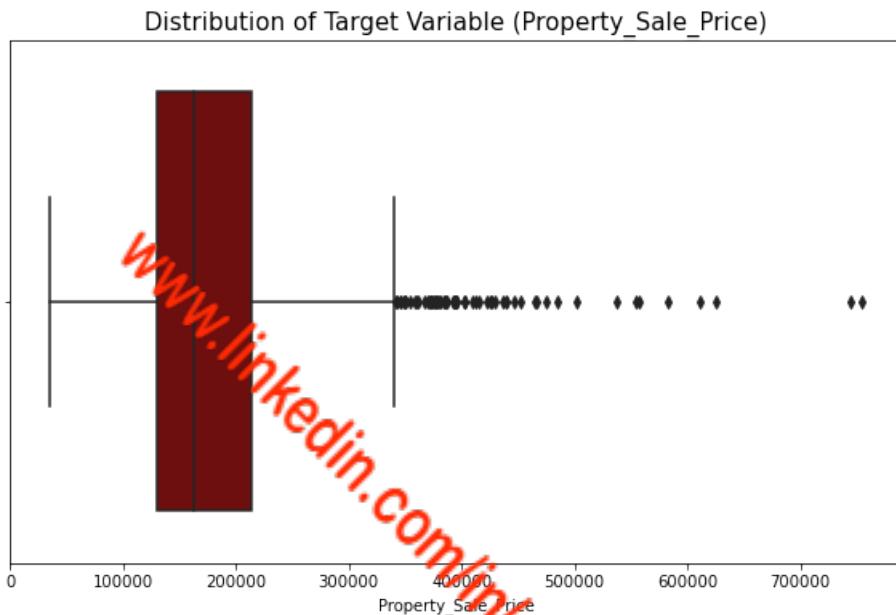


4.1.4 Outliers Discovery

```
In [23]: # plot a boxplot of target variable to detect the outliers
sns.boxplot(df_property['Property_Sale_Price'], color='maroon')

# set plot label
# set text size using 'fontsize'
plt.title('Distribution of Target Variable (Property_Sale_Price)', fontsize = 15)

# display the plot
plt.show()
```



From the above plot we can see that there are outliers present in the target variable 'Property_Sale_Price'. Outliers badly affect the prediction of the regression model and thus, we will remove these outliers.

```
In [24]: # remove the observations with the house price greater than or equal to 500000
df_property = df_property[df_property['Property_Sale_Price'] < 500000]

# check the dimension of the data
df_property.shape
```

Out[24]: (1451, 78)

4.1.5 Missing Values

If we do not handle the missing values properly then we may end up drawing an inaccurate inference about the data.



Look for the missing values and handle the missing values separately for numerical and categorical values.

Look for the missing values

```
In [25]: # 'isnull().sum()' returns the number of missing values in each variable
# 'ascending = False': sorts values in the descending order
total_nulls = df_property.isnull().sum().sort_values(ascending = False)

# calculate the percentage of missing values
# 'ascending = False' sorts values in the descending order
percent_null = (df_property.isnull().sum()*100/df_property.isnull().count())
percent_null = percent_null.sort_values(ascending = False)

# concat the 'total_nulls' and 'percent_null' columns
# 'axis = 1' stands for columns
missing_values = pd.concat([total_nulls, percent_null], axis = 1, keys = ['Total Nulls', 'Percentage'])

# add the column containing data type of each variable
missing_values['Data Type'] = df_property[missing_values.index].dtypes
missing_values
```

Out[25]:

	Total Nulls	Percentage of Missing Values	Data Type
MiscFeature	1397	96.278429	object
Alley	1360	93.728463	object
Fence	1171	80.702963	object
FireplaceQu	690	47.553411	object
LotFrontage	259	17.849759	float64
GarageType	81	5.582357	object
GarageFinish	8	5.582357	object
GarageQual	81	5.582357	object
GarageCond	81	5.582357	object
GarageYrBlt	81	5.582357	float64
BsmtFinType2	38	2.618884	object



We can see that 18 variables contain the missing values.

Handle the missing values for numerical variables

```
In [26]: # filter out the categorical variables and consider only the numeric variables with missing values
num_missing_values = missing_values[(missing_values['Total Nulls'] > 0) & (missing_values['Data Type'] == 'float64')]
num_missing_values
```

Out[26]:

	Total Nulls	Percentage of Missing Values	Data Type
LotFrontage	259	17.849759	float64
GarageYrBlt	81	5.582357	float64
MasVnrArea	8	0.551344	float64

For the numerical variables, we can replace the missing values by their mean, median or mode as per the requirement.

The variable 'LotFrontage' is right skewed and thus we will fill the missing values with its median value

```
In [27]: # use the function fillna() to fill the missing values
df_property['LotFrontage'] = df_property['LotFrontage'].fillna(df_property['LotFrontage'].median())
```

We will replace the missing values in the numeric variable GarageYrBlt by 0. The missing values in this variable indicates that there are 81 observations for which garage facility is not available.

```
In [28]: # use the function fillna() to replace missing values in 'GarageYrBlt' with 0
df_property['GarageYrBlt'] = df_property['GarageYrBlt'].fillna(0)
```

The variable 'MasVnrArea' is positively skewed and thus we will fill the missing values with its median value

```
In [29]: # use the function fillna() to fill the missing values  
df_property['MasVnrArea'] = df_property['MasVnrArea'].fillna(df_property['MasVnrArea'].median())
```

Handle the missing values for categorical variables

```
In [30]: # filter out the numerical variables and consider only the categorical variables with missing value  
cat_missing_values = missing_values[(missing_values['Total Nulls'] > 0) & (missing_values['Data Type'] == 'category')]  
cat_missing_values
```

Out[30]:

	Total Nulls	Percentage of Missing Values	Data Type
MiscFeature	1397	96.278429	object
Alley	1360	93.728463	object
Fence	1171	80.702963	object
FireplaceQu	690	47.553411	object
GarageType	81	5.582357	object
GarageFinish	81	5.582357	object
GarageQual	81	5.582357	object
GarageCond	81	5.582357	object
BsmtFinType2	38	2.618884	object
BsmtExposure	38	2.618884	object
BsmtCond	37	2.549966	object
BsmtFinType1	37	2.549966	object
BsmtQual	37	2.549966	object
MasVnrType	8	0.551374	object
Electrical	1	0.068918	object

```
In [31]: # according to the data definition, 'NA' denotes the absence of miscellaneous feature  
# replace NA values in 'MiscFeature' with a valid value, 'None'  
df_property['MiscFeature'] = df_property['MiscFeature'].fillna('None')  
  
# replace NA values in 'Alley' with a valid value, 'No alley access'  
df_property['Alley'] = df_property['Alley'].fillna('No alley access')  
  
# replace NA values in 'Fence' with a valid value, 'No Fence'  
df_property['Fence'] = df_property['Fence'].fillna('No Fence')  
  
# replace null values in 'FireplaceQu' with a valid value, 'No Fireplace'  
df_property['FireplaceQu'] = df_property['FireplaceQu'].fillna('No Fireplace')
```

```
In [32]: # replace the missing values in the categoric variables representing the garage by 'No Garage'  
for col in ['GarageType', 'GarageFinish', 'GarageCond', 'GarageQual']:  
    df_property[col].fillna('No Garage', inplace = True)
```

```
In [33]: # according to the data definition, 'NA' denotes the absence of basement in the variabels 'BsmtQual'  
# replace the missing values in the categoric variables representing the basement by 'No Basement'  
for col in ['BsmtFinType2', 'BsmtExposure', 'BsmtQual', 'BsmtCond', 'BsmtFinType1']:  
    df_property[col].fillna('No Basement', inplace = True)
```

```
In [34]: # according to the data definition, 'NA' denotes the absence of masonry veneer  
# replace the missing values in the categorical variable 'MasVnrType' with a value, 'None'  
df_property['MasVnrType'] = df_property['MasVnrType'].fillna('None')
```

```
In [35]: # replace the missing values in the categorical variable 'Electrical' with its mode  
mode_electrical = df_property['Electrical'].mode()  
df_property['Electrical'].fillna(mode_electrical[0] , inplace = True)
```

4.1.6 Study correlation



To check the correlation between numerical variables, compute a correlation matrix and plot a heatmap for the correlation matrix

Compute a correlation matrix

www.linkedin.com/in/lajantha-devi-vairamani

```
In [36]: # use the corr() function to generate the correlation matrix of the numeric variables
corrmat = df_property.corr()

# print the correlation matrix
corrmat
```

Out[36]:

	LotFrontage	LotArea	YearBuilt	YearRemodAdd	MasVnrArea	BsmtFinSF1	BsmtFinSF2	BsmtUnfSF
LotFrontage	1.000000	0.306226	0.110575	0.078240	0.168580	0.201974	0.041758	0.122347
LotArea	0.306226	1.000000	0.007052	0.007963	0.079576	0.204725	0.113638	-0.002229
YearBuilt	0.110575	0.007052	1.000000	0.590358	0.308832	0.242388	-0.050381	0.147484
YearRemodAdd	0.078240	0.007963	0.590358	1.000000	0.170015	0.120246	-0.069146	0.179960
MasVnrArea	0.168580	0.079576	0.308832	0.170015	1.000000	0.237426	-0.070851	0.116437
BsmtFinSF1	0.201974	0.204725	0.242388	0.120246	0.237426	1.000000	-0.054443	-0.502348
BsmtFinSF2	0.041758	0.113638	-0.050381	-0.069146	-0.070851	-0.054443	1.000000	-0.211687
BsmtUnfSF	0.122341	-0.002229	0.147484	0.179966	0.116437	-0.502348	-0.211681	1.000000
TotalBsmtSF	0.351744	0.253819	0.385072	0.284008	0.340365	0.506796	0.101582	0.422596
1stFlrSF	0.403928	0.293575	0.272635	0.232100	0.318259	0.427145	0.096551	0.322025
2ndFlrSF	0.055154	0.033593	0.002106	0.136997	0.150374	-0.160550	-0.096758	-0.000495
LowQualFinSF	0.038892	0.005727	-0.183659	-0.061920	-0.069527	-0.064339	0.014921	0.028657
GrLivArea	0.52795	0.248741	0.188173	0.283858	0.358948	0.177440	-0.008304	0.243357
BsmtFullBath	0.081810	0.154564	0.185755	0.117521	0.078313	0.652108	0.158017	-0.421406
BsmtHalfBath	-0.009650	0.047356	-0.039128	-0.012512	0.015343	0.065858	0.072546	-0.099057
FullBath	0.166976	0.111174	0.464424	0.436448	0.251721	0.039019	-0.076080	0.286674
HalfBath	0.039602	0.006773	0.239822	0.180804	0.192394	-0.003163	-0.029227	-0.044902
BedroomAbvGr	0.233933	0.114221	0.073171	-0.041872	0.099591	-0.108698	-0.012647	0.162806
KitchenAbvGr	-0.003219	-0.016372	-0.174064	-0.148847	-0.036336	-0.080020	-0.040788	0.030835
TotRmsAbvGrd	0.311321	0.175347	0.081153	0.181722	0.254011	0.016513	-0.036589	0.248997
Fireplaces	0.223229	0.268811	0.140979	0.106649	0.238226	0.245847	0.044752	0.050762
GarageYrBlt	0.097975	0.071133	0.271118	0.145115	0.133385	0.114548	0.035023	0.042190
GarageCars	0.261254	0.145224	0.533519	0.416036	0.351166	0.209144	-0.039581	0.212905
GarageArea	0.317880	0.171107	0.474181	0.366414	0.357034	0.283615	-0.017447	0.182243
WoodDeckSF	0.073205	0.159650	0.223873	0.205597	0.145807	0.202566	0.071761	-0.006586
OpenPorchSF	0.133322	0.084143	0.186221	0.224186	0.172038	0.112177	0.003118	0.125145
EnclosedPorch	0.013371	-0.015829	-0.386450	-0.192517	-0.108302	-0.099877	0.036869	-0.001607
3SsnPorch	0.063617	0.021493	0.032225	0.046070	0.022121	0.028548	-0.030038	0.021195
ScreenPorch	0.035947	0.032825	-0.057585	-0.045230	0.057424	0.057946	0.085238	-0.017501
MiscVal	0.000464	0.039093	-0.033934	-0.009821	-0.029244	0.004795	0.005002	-0.023749
MoSold	0.015944	0.003327	0.016370	0.025102	0.005738	-0.010615	-0.014359	0.039297
YrSold	0.009395	-0.016991	-0.016129	0.033392	-0.010231	0.008691	0.030290	-0.038680
Property_Sale_Price	0.318744	0.252156	0.543898	0.531708	0.436515	0.357076	-0.014028	0.223997

2. Plot the heatmap for the diagonal correlation matrix

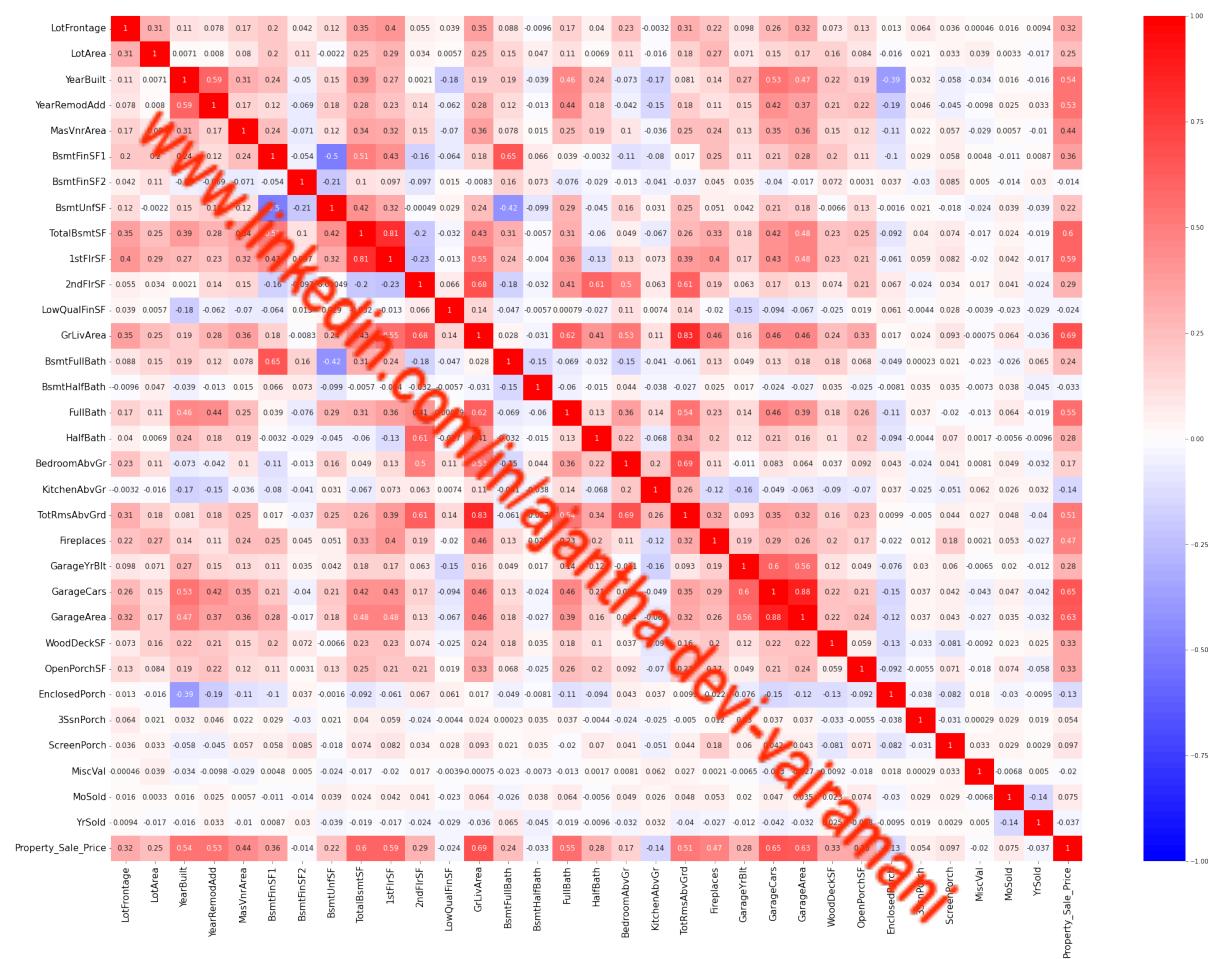
A correlation matrix is a symmetric matrix. Plot only the upper triangular entries using a heatmap.

```
In [37]: # set the plot size
plt.figure(figsize = (35,25))

# plot the heat map
# corr: give the correlation matrix
# cmap: color code used for plotting
# annot_kws: sets the font size of the annotation
# annot: prints the correlation values in the chart
# vmax: gives a maximum range of values for the chart
# vmin: gives a minimum range of values for the chart
sns.heatmap(corrmat, annot = True, vmax = 1.0, vmin = -1.0, cmap = 'bwr', annot_kws = {"size": 11})

# set the size of x and y axes labels using 'fontsize'
plt.xticks(fontsize = 15)
plt.yticks(fontsize = 15)

# display the plot
plt.show()
```



The diagonal represents the correlation of the variable with itself thus all the diagonal entries are '1'. The dark red squares represent the variables with strong positive correlation.



From the above plot we can see that the highest positive correlation (= 0.88) is between the variables 'GarageArea' and 'GarageCars'. Also there is strong positive correlation between the pairs (1StFlrSF, TotalBsmtSF) and (TotRmsAbvGrd, GrlivArea). There may be multicollinearity present.

No two variables have strong negative correlation in the dataset.



Correlation does not imply causation. In other words, if two variables are correlated, it does not mean that one variable caused the other.

4.1.7 Analyze Relationships Between Target and Categorical Variables



Plot the box-and-whisker plot for visualizing relationships between target and categorical variables

www.linkedin.com/in/lajantha-devi-vairamani

```
In [38]: # create an empty list to store all the categorical variables
categorical=[]

# check the data type of each variable
for column in df_property:

    # check if the variable has the categorical type
    if is_string_dtype(df_property[column]):

        # append the categorical variables to the List 'categorical'
        categorical.append(column)

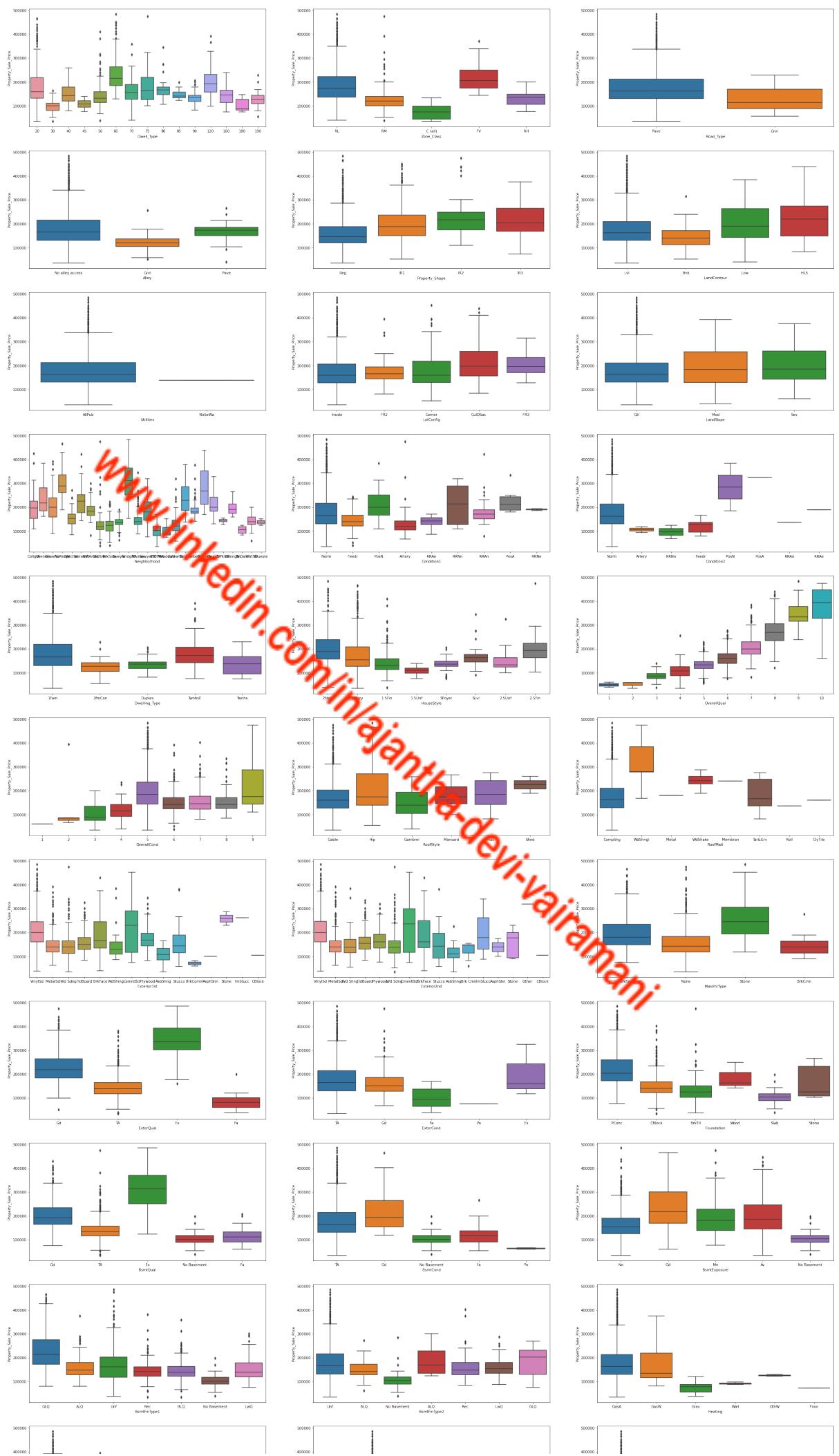
# plot the count plot for each categorical variable
# 'figsize' sets the figure size
fig, ax = plt.subplots(nrows = 14, ncols = 3, figsize = (40, 100))

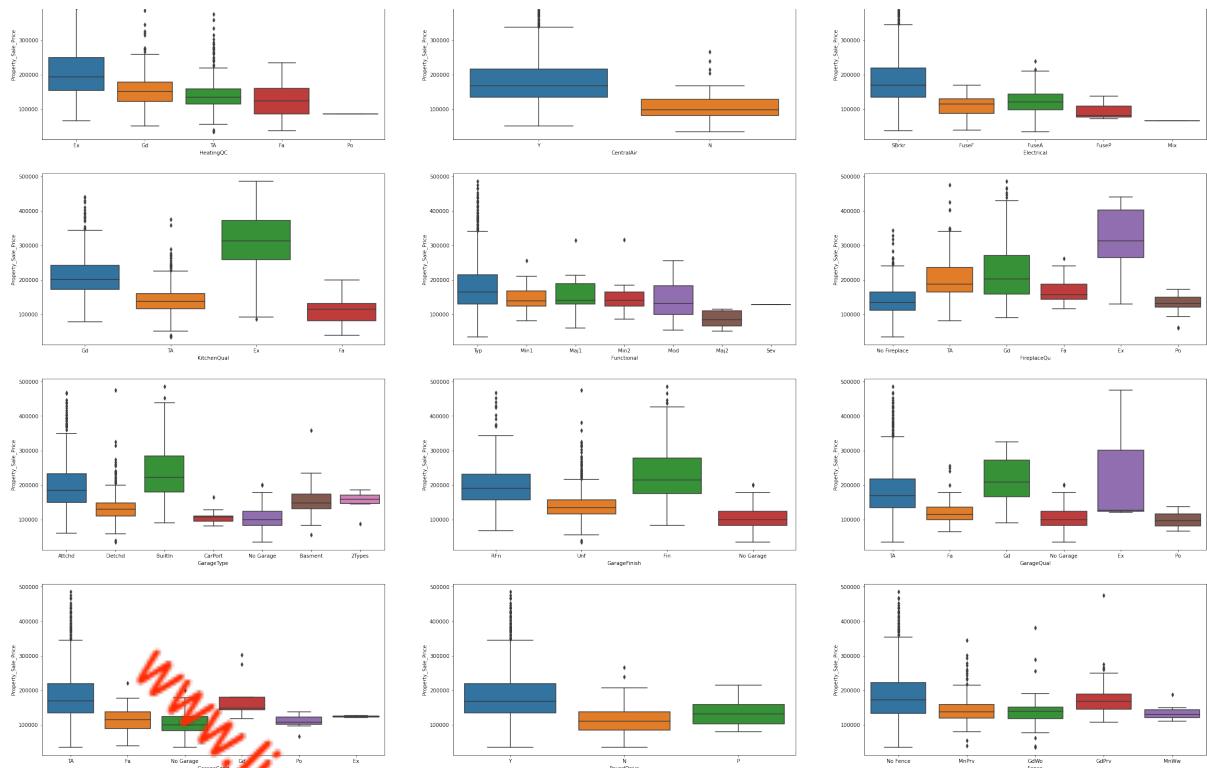
# plot the boxplot for each categoric and target variable
for variable, subplot in zip(categorical, ax.flatten()):
    sns.boxplot(x = variable, y = 'Property_Sale_Price', data = df_property, ax = subplot)

# display the plot
plt.show()
```

www.linkedin.com/in/lajantha-devi-vairamani

www.linkedin.com/in/lajantha-devi-vairamani





It can be seen that most of the categorical variables have an effect on the sale price of the property. The median sale price rises exponentially with respect to the rating of the overall quality of the material used.

4.2 Data Preparation

4.2.1 Check for Normality

Plot a histogram and also perform the Shapiro-Wilk test

We use the function `hist()` from the matplotlib library to plot a histogram.

linkedin.com/in/ajantha-devivairamani

```
In [39]: # check the distribution of target variable
df_property.Property_Sale_Price.hist(color = 'maroon')

# add plot and axes labels
# set text size using 'fontsize'
plt.title('Distribution of Target Variable (Property_Sale_Price)', fontsize = 15)
plt.xlabel('Property Sale Price', fontsize = 15)
plt.ylabel('Frequency', fontsize = 15)

# display the plot
plt.show()
```



We can see that the variable 'Property_Sale_Price' is positively skewed and thus we can say that it is not normally distributed.



We should not only make conclusions through visual representations or only using a statistical test but perform multiple ways to get the best insights.

Let us perform from Shapiro-Wilk test to check the normality of the target variable.

The null and alternate hypothesis of Shapiro-Wilk test is as follows:

H_0 : The data is normally distributed

H_1 : The data is not normally distributed

```
In [40]: # shapiro() returns the the test statistics along with the p-value of the test
stat, p = shapiro(df_property.Property_Sale_Price)

# print the numeric outputs of the Shapiro-Wilk test upto 3 decimal places
print('Statistics=% .3f, p-value=% .3f' % (stat, p))

# set the Level of significance (alpha) to 0.05
alpha = 0.05

# if the p-value is less than alpha print we reject alpha
# if the p-value is greater than alpha print we accept alpha
if p > alpha:
    print('The data is normally distributed (fail to reject H0)')
else:
    print('The data is not normally distributed (reject H0)')
```

Statistics=0.919, p-value=0.000
The data is not normally distributed (reject H0)



We can see that the p-value is less than 0.05 and thus we reject the null hypothesis. It can be concluded that the data is not normally distributed.



Shapiro Wilk Test does not work if the number of observations are more than 5000. However Shapiro Wilk test is more robust than other tests. In case where the observations are more than 5000, other tests like Anderson Darling test or Jarque Bera test may also be used.

If the data is not normally distributed, use log transformation to reduce the skewness and get a near normally distributed data

The log transformation can be used to reduce the skewness. To log transform the 'Property_Sale_Price' variable we use the function `np.log()`.

```
In [41]: # Log transformation using np.log()
df_property['log_Property_Sale_Price'] = np.log(df_property['Property_Sale_Price'])

# display the top 5 rows of the data
df_property.head()
```

Out[41]:

	Dwell_Type	Zone_Class	LotFrontage	LotArea	Road_Type	Alley	Property_Shape	LandContour	Utilities	LotConfig
0	60	RL	65.0	8450	Pave	No alley access	Reg	Lvl	AllPub	Inside
1	20	RL	80.0	9600	Pave	No alley access	Reg	Lvl	AllPub	FR2
2	60	RL	68.0	11250	Pave	No alley access	IR1	Lvl	AllPub	Inside
3	70	RL	60.0	9550	Pave	No alley access	IR1	Lvl	AllPub	Corner
4	60	RL	84.0	14260	Pave	No alley access	IR1	Lvl	AllPub	FR2

Recheck for normality by plotting histogram and performing Shapiro-Wilk test

Let us first plot a histogram of `log_Property_Sale_Price`.

```
In [42]: # recheck for normality
# plot the histogram using hist
df_property.log_Property_Sale_Price.hist(color = 'maroon')

# add plot and axes labels
# set text size using 'fontsize'
plt.title('Distribution of Log-transformed Target Variable (log_Property_Sale_Price)', fontsize = 15)
plt.xlabel('Sale Price (log-transformed)', fontsize = 15)
plt.ylabel('Frequency', fontsize = 15)

# display the plot
plt.show()
```



It can be seen that the variable log_Property_Sale_Price is near normally distributed. Lets confirm it again by using the Shapiro-Wilk test.

Let us perform Shapiro-Wilk test.

```
In [43]: # shapiro() returns the the test statistics along with the p-value of the test
stat, p = shapiro(df_property['log_Property_Sale_Price'])

# print the numeric outputs of the Shapiro-Wilk test upto 3 decimal places
print('Statistics=%.3f, p-value=%.3f' % (stat, p))

# set the level of significance (alpha) to 0.05
alpha = 0.05

# if the p-value is Less than alpha print we reject alpha
# if the p-value is greater than alpha print we accept alpha
if p > alpha:
    print('The data is normally distributed (fail to reject H0)')
else:
    print('The data is not normally distributed (reject H0)')
```

```
Statistics=0.992, p-value=0.000
The data is not normally distributed (reject H0)
```

```
In [44]: # find the skewness of the variable log_Property_Sale_Price
df_property['log_Property_Sale_Price'].skew()
```

```
Out[44]: -0.03836503763294778
```



It can be visually seen that the data has near-normal distribution, but Shapiro-Wilk test does not support the claim.
Note that in reality it might be very tough for your data to adhere to all assumptions your algorithm needs.

4.2.2 Dummy Encode the Categorical Variables



We need to perform dummy encoding on our categorical variables before we proceed; since the method of OLS works only on the numeric data.

Filter numerical and categorical variables

```
In [45]: # filter out the categorical variables and consider only the numeric variables using (include=np.number)
df_numeric_features = df_property.select_dtypes(include=np.number)

# display the numeric features
df_numeric_features.columns
```

```
Out[45]: Index(['LotFrontage', 'LotArea', 'YearBuilt', 'YearRemodAdd', 'MasVnrArea',
       'BsmtFinSF1', 'BsmtFinSF2', 'BsmtUnfSF', 'TotalBsmtSF', '1stFlrSF',
       '2ndFlrSF', 'LowQualFinSF', 'GrLivArea', 'BsmtFullBath', 'BsmtHalfBath',
       'FullBath', 'HalfBath', 'BedroomAbvGr', 'KitchenAbvGr', 'TotRmsAbvGrd',
       'Fireplaces', 'GarageYrBlt', 'GarageCars', 'GarageArea', 'WoodDeckSF',
       'OpenPorchSF', 'EnclosedPorch', '3SsnPorch', 'ScreenPorch', 'MiscVal',
       'MoSold', 'YrSold', 'Property_Sale_Price', 'log_Property_Sale_Price'],
      dtype='object')
```

```
In [46]: # filter out the numerical variables and consider only the categorical variables using (include=object)
df_categorical_features = df_property.select_dtypes(include = object)

# display categorical features
df_categorical_features.columns
```

```
Out[46]: Index(['Dwell_Type', 'Zone_Class', 'Road_Type', 'Alley', 'Property_Shape',
       'LandContour', 'Utilities', 'LotConfig', 'LandSlope', 'Neighborhood',
       'Condition1', 'Condition2', 'Dwelling_Type', 'HouseStyle',
       'OverallQual', 'OverallCond', 'RoofStyle', 'RoofMatl', 'Exterior1st',
       'Exterior2nd', 'MasVnrType', 'ExterQual', 'ExterCond', 'Foundation',
       'BsmtQual', 'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFinType2',
       'Heating', 'HeatingQC', 'CentralAir', 'Electrical', 'KitchenQual',
       'Functional', 'FireplaceQu', 'GarageType', 'GarageFinish', 'GarageQual',
       'GarageCond', 'PavedDrive', 'Fence', 'MiscFeature', 'SaleType',
       'SaleCondition'],
      dtype='object')
```

Dummy encode the categorical variables

```
In [47]: # to create the dummy variables we use 'get_dummies()' from pandas
# to create (n-1) dummy variables we use 'drop_first = True'
dummy_encoded_variables = pd.get_dummies(df_categorical_features, drop_first = True)
```

Concatenate numerical and dummy encoded categorical variables

```
In [48]: # concatenate the numerical and dummy encoded categorical variables column-wise
df_property_dummy = pd.concat([df_numeric_features, dummy_encoded_variables], axis=1)

# display data with dummy variables
df_property_dummy.head()
```

Out[48]:

	LotFrontage	LotArea	YearBuilt	YearRemodAdd	MasVnrArea	BsmtFinSF1	BsmtFinSF2	BsmtUnfSF	TotalBsmtSF	1stFlrSF
0	65.0	8450	2003	2003	196.0	706	0	150	856	
1	80.0	9600	1976	1976	0.0	978	0	284	1262	
2	68.0	11250	2001	2002	162.0	486	0	434	920	
3	60.0	9550	1915	1970	0.0	216	0	540	756	
4	84.0	14260	2000	2000	350.0	655	0	490	1145	



```
In [49]: # check the dimensions of the dataframe  
df_property_dummy.shape
```

Out[49]: (1451, 285)



There are various forms of encoding like n-1 dummy encoding, one hot encoding, label encoding, frequency encoding.



We will now train models by fitting a linear regression model using the method of ordinary least square(OLS).

5. Linear Regression (OLS)

5.1 Multiple Linear Regression Full Model with Log Transformed Dependent Variable (OLS)

Follow the steps in order to build the OLS model:

1. Split the data into training and test sets

```
In [50]: # add the intercept column using 'add_constant()'  
df_property_dummy = sm.add_constant(df_property_dummy)  
  
# separate the independent and dependent variables  
# drop(): drops the specified columns  
X = df_property_dummy.drop(['Property_Sale_Price', 'log_Property_Sale_Price'], axis = 1)  
  
# extract the target variable from the data set  
y = df_property_dummy[['Property_Sale_Price', 'log_Property_Sale_Price']]  
  
# split data into train data and test data  
# what proportion of data should be included in test data is passed using 'test_size'  
# set 'random_state' to get the same data each time the code is executed  
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state = 1)  
  
# check the dimensions of the train & test subset for  
# print dimension of predictors train set  
print("The shape of X_train is:", X_train.shape)  
  
# print dimension of predictors test set  
print("The shape of X_test is:", X_test.shape)  
  
# print dimension of target train set  
print("The shape of y_train is:", y_train.shape)  
  
# print dimension of target test set  
print("The shape of y_test is:", y_test.shape)
```

The shape of X_train is: (1015, 284)
The shape of X_test is: (436, 284)
The shape of y_train is: (1015, 2)
The shape of y_test is: (436, 2)

2. Build model using sm.OLS().fit()

```
In [51]: # build a full model using OLS()
# consider the log of sales price as the target variable
# use fit() to fit the model on train data
linreg_logmodel_full = sm.OLS(y_train['log_Property_Sale_Price'], X_train).fit()

# print the summary output
print(linreg_logmodel_full.summary())
```

OLS Regression Results

Dep. Variable:	log_Property_Sale_Price	R-squared:	0.952			
Model:	OLS	Adj. R-squared:	0.935			
Method:	Least Squares	F-statistic:	56.43			
Date:	Wed, 23 Oct 2024	Prob (F-statistic):	0.00			
Time:	17:11:14	Log-Likelihood:	1089.1			
No. Observations:	1015	AIC:	-1648.			
Df Residuals:	750	BIC:	-343.7			
Df Model:	264					
Covariance Type:	nonrobust					
	coef	std err	t	P> t	[0.025	0.975]
const	6.6542	4.107	1.620	0.106	-1.408	14.716
LotFrontage	0.0003	0.000	1.384	0.167	-0.000	0.001
LotArea	3.336e-06	5.9e-07	5.658	0.000	2.18e-06	4.49e-06
YearBuilt	0.0020	0.000	4.393	0.000	0.001	0.003
YearRemodAdd	0.0010	0.000	3.373	0.001	0.000	0.002
...

This model explains around 95% of the variation in dependent variable log_Property_Sale_Price. The Condition Number 6.16e+19 suggests that there is severe multicollinearity in the data. The Durbin-Watson test statistics is 1.934 i.e. close to 2.0 and thus it indicates that there is no autocorrelation.



Condition Number : Multicollinearity can be checked by computing the condition number(CN). If condition number is between 100 and 1000, there is moderate multicollinearity, if condition number is less than 100, there is no multicollinearity and if condition number is greater 1000 there is severe multicollinearity in the data.



Durbin-Watson : The Durbin-Watson statistic will always have a value between 0 and 4. A value of 2.0 means that there is no autocorrelation detected in the sample. Values from 0 to less than 2 indicate positive autocorrelation and values from 2 to 4 indicate negative autocorrelation.

3. Predict the values using test set

```
In [52]: # predict the 'log_Property_Sale_Price' using predict()
linreg_logmodel_full_predictions = linreg_logmodel_full.predict(x_test)
```



Note that the predicted values are log transformed Property_Sale_Price. In order to get Property_Sale_Price values, we take the antilog of these predicted values by using the function np.exp()

```
In [53]: # take the exponential of predictions using np.exp()
predicted_Property_Sale_Price = np.exp(linreg_logmodel_full_predictions)

# extract the 'Property_Sale_Price' values from the test data
actual_Property_Sale_Price = y_test['Property_Sale_Price']
```

4. Compute accuracy measures

Now we calculate accuracy measures Root-mean-square-error (RMSE), R-squared and Adjusted R-squared.

```
In [54]: # calculate rmse using rmse()
linreg_logmodel_full_rmse = rmse(actual_Property_Sale_Price, predicted_Property_Sale_Price)

# calculate R-squared using rsquared
linreg_logmodel_full_rsquared = linreg_logmodel_full.rsquared

# calculate Adjusted R-Squared using rsquared_adj
linreg_logmodel_full_rsquared_adj = linreg_logmodel_full.rsquared_adj
```

5. Tabulate the results

```
In [55]: # create the result table for all accuracy scores
# accuracy measures considered for model comparision are RMSE, R-squared value and Adjusted R-squared
# create a list of column names
cols = ['Model', 'RMSE', 'R-Squared', 'Adj. R-Squared']

# create a empty dataframe of the colums
# columns: specifies the columns to be selected
result_tabulation = pd.DataFrame(columns = cols)

# compile the required information
linreg_logmodel_full_metrics = pd.Series({'Model': "Linreg full model with log of target variable",
                                           'RMSE': linreg_logmodel_full_rmse,
                                           'R-Squared': linreg_logmodel_full_rsquared,
                                           'Adj. R-Squared': linreg_logmodel_full_rsquared_adj
                                         })

# append our result table using append()
# ignore_index=True: does not use the index labels
# python can only append a series if ignore_index=True or if the Series has a name
result_tabulation = result_tabulation.append(linreg_logmodel_full_metrics, ignore_index = True)

# print the result table
result_tabulation
```

Out[55]:

	Model	RMSE	R-Squared	Adj. R-Squared
0	Linreg full model with log of target variable	37726.465264	0.952066	0.935193



We will also build a linear regression model without performing log transformation on the target variable.

5.2 Multiple Linear Regression Full Model without Log Transformed Target Variable (OLS)

In this section we build a full model with linear regression using OLS (Ordinary Least Square) technique. By full model we indicate that we consider all the independent variables that are present in the dataset.

In this case, we do not consider any kind of transformation on the dependent variable, we use the 'Property_Sale_Price' variable as it is.

We have already done train and test split while building the previous model.

1. Build model using sm.OLS().fit()

```
In [56]: # build a OLS model using function OLS()
# Property_Sale_Price is our target variable
# use fit() to fit the model on train data
linreg_nolog_model = sm.OLS(y_train['Property_Sale_Price'], X_train).fit()

# print the summary output
print(linreg_nolog_model.summary())
```

OLS Regression Results

Dep. Variable:	Property_Sale_Price	R-squared:	0.947			
Model:	OLS	Adj. R-squared:	0.928			
Method:	Least Squares	F-statistic:	50.66			
Date:	Wed, 23 Oct 2024	Prob (F-statistic):	0.00			
Time:	17:16:22	Log-Likelihood:	-11281.			
No. Observations:	1015	AIC:	2.309e+04			
Df Residuals:	750	BIC:	2.440e+04			
Df Model:	264					
Covariance Type:	nonrobust					
	coef	std err	t	P> t	[0.025	0.975]
const	-2.939e+05	8.06e+05	-0.365	0.715	-1.88e+06	1.29e+06
LotFrontage	33.2893	45.735	0.728	0.467	-56.495	123.074
LotArea	0.8280	0.116	7.157	0.000	0.601	1.055
YearBuilt	400.9010	90.470	4.431	0.000	223.296	578.506
YearRemodAdd	106.4820	59.474	1.790	0.074	-10.273	223.237
...



This model explains around 94% of the variation in dependent variable `Property_Sale_Price`.
The Durbin-Watson test statistics is 1.973 and indicates that there is no autocorrelation.
The Condition Number 16e+19 suggests that there is severe multicollinearity in the data.

2. Predict the values using test set

```
In [57]: # predict the 'Property_Sale_Price' using predict()
linreg_nolog_model_predictions = linreg_nolog_model.predict(X_test)
```

3. Compute accuracy measures

Now we calculate accuracy measures Root-mean-square-error (RMSE), R-squared and Adjusted R-squared.

```
In [58]: # calculate rmse using rmse()
linreg_nolog_model_rmse = rmse(actual_Property_Sale_Price, linreg_nolog_model_predictions)

# calculate R-squared using rsquared
linreg_nolog_model_rsquared = linreg_nolog_model.rsquared

# calculate Adjusted R-Squared using rsquared_adj
linreg_nolog_model_rsquared_adj = linreg_nolog_model.rsquared_adj
```

4. Tabulate the results

```
In [59]: # append the result table
# compile the required information
linreg_nolog_model_metrics = pd.Series({'Model': "Linreg full model without log of target variable",
                                         'RMSE':linreg_nolog_model_rmse,
                                         'R-Squared': linreg_nolog_model_rsquared,
                                         'Adj. R-Squared': linreg_nolog_model_rsquared_adj}

# append our result table using append()
# ignore_index=True: does not use the index labels
# python can only append a Series if ignore_index=True or if the Series has a name
result_tabulation = result_tabulation.append(linreg_nolog_model_metrics, ignore_index = True)

# print the result table
result_tabulation
```

Out[59]:

	Model	RMSE	R-Squared	Adj. R-Squared
0	Linreg full model with log of target variable	37726.465264	0.952066	0.935193
1	Linreg full model without log of target variable	54970.829645	0.946898	0.928206



If we compare the results in the table we can see that the linreg model withh log of target variable is performing slightly better than the model without log of target variable. Thus we will continue with the target variable 'log_Property_Sale_Price'.



Let us perform feature engineering and take a look at building a linear regression full model by adding new features to the dataset.

5.3 Feature Engineering

It is the process of creating new features using domain knowledge of the data that provides more insight into the data. Let us create a few features from the existing dataset and build a regression model on the newly created data.

5.3.1 Multiple Linear Regression (Using New Feature) - 1

In order to build the model, we do the following:

1. Create a new feature by using variables 'TotalBsmtSF', '1stFlrSF', '2ndFlrSF', and 'GrLivArea'.

Calculate the complete area of the house.

Create a new variable `TotalSF` representing the total square feet area of the house by adding the area of the first floor, second floor, ground level and basement of the house.

```
In [60]: # create a new variable 'TotalSF' using the variables 'TotalBsmtSF', '1stFlrSF', '2ndFlrSF', and 'GrLivArea'
df_property['TotalSF'] = df_property['TotalBsmtSF'] + df_property['1stFlrSF'] + df_property['2ndFlrSF']

# as we have created a new variable using the variables 'TotalBsmtSF', '1stFlrSF', '2ndFlrSF' and
# use 'drop()' to remove the variables
df_property = df_property.drop(['TotalBsmtSF', "1stFlrSF", "2ndFlrSF", "GrLivArea"], axis=1)
```

```
In [61]: # filter out the categorical variables and consider only the numerical variables using (include=np.number)
df_numeric_features = df_property.select_dtypes(include=np.number)

# filter out the numerical variables and consider only the categorical variables using (include=object)
df_categorical_features = df_property.select_dtypes(include = object)
```

Dummy encode the catergorical variables

```
In [62]: # use 'get_dummies()' from pandas to create dummy variables
# use 'drop_first = True' to create (n-1) dummy variables
dummy_encoded_variables = pd.get_dummies(df_categorical_features, drop_first = True)
```

Concatenate numerical and dummy encoded categorical variables

```
In [63]: # concatenate the numerical and dummy encoded categorical variables column-wise
df_dummy = pd.concat([df_numeric_features, dummy_encoded_variables], axis=1)

# display data with dummy variables
df_dummy.head()
```

Out[63]:

	LotFrontage	LotArea	YearBuilt	YearRemodAdd	MasVnrArea	BsmtFinSF1	BsmtFinSF2	BsmtUnfSF	LowQualFinSF	B
0	65.0	8450	2003	2003	196.0	706	0	150	0	0
1	80.0	9600	1976	1976	0.0	978	0	284	0	0
2	68.0	11250	2001	2002	162.0	486	0	434	0	0
3	60.0	9550	1915	1970	0.0	216	0	540	0	0
4	84.0	14260	2000	2000	350.0	655	0	490	0	0

2. Split the data into train and test sets

```
In [64]: # add the intercept column using 'add_constant()'
df_dummy = sm.add_constant(df_dummy)

# separate the independent and dependent variables
# drop(): drops the specified columns
# axis=1: specifies that the column is to be dropped
X = df_dummy.drop(['Property_Sale_Price', 'log_Property_Sale_Price'], axis = 1)

# extract the target variable from the data set
y = df_dummy[['Property_Sale_Price', 'log_Property_Sale_Price']]

# split data into train data and test data
# what proportion of data should be included in test data is specified using 'test_size'
# set 'random_state' to get the same data each time the code is executed
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state = 1)

# check the dimensions of the train & test subset for
# print dimension of predictors train set
print("The shape of X_train is:", X_train.shape)

# print dimension of predictors test set
print("The shape of X_test is:", X_test.shape)

# print dimension of target train set
print("The shape of y_train is:", y_train.shape)

# print dimension of target test set
print("The shape of y_test is:", y_test.shape)
```

The shape of X_train is: (1015, 281)

The shape of X_test is: (436, 281)

The shape of y_train is: (1015, 2)

The shape of y_test is: (436, 2)

3. Build model using sm.OLS().fit()

```
In [65]: # build a full model using OLS()
# consider the target variable 'log_Property_Sale_Price'
# use fit() to fit the model on train data
linreg_feature_1_model = sm.OLS(y_train['log_Property_Sale_Price'], X_train).fit()

# print the summary output
print(linreg_feature_1_model.summary())
```

OLS Regression Results

Dep. Variable:	log_Property_Sale_Price	R-squared:	0.952			
Model:	OLS	Adj. R-squared:	0.935			
Method:	Least Squares	F-statistic:	56.71			
Date:	Wed, 23 Oct 2024	Prob (F-statistic):	0.00			
Time:	17:21:47	Log-Likelihood:	1089.1			
No. Observations:	1015	AIC:	-1650.			
Df Residuals:	751	BIC:	-350.5			
Df Model:	263					
Covariance Type:	nonrobust					
	coef	std err	t	P> t	[0.025	0.975]
const	6.6868	4.103	1.630	0.104	-1.368	14.741
LotFrontage	0.0003	0.000	1.411	0.159	-0.000	0.001
LotArea	3.349e-06	5.88e-07	5.697	0.000	2.2e-06	4.5e-06
YearBuilt	0.0020	0.000	4.386	0.000	0.001	0.003
YearRemodAdd	0.0010	0.000	3.397	0.001	0.000	0.002

 This model explains around 95% of the variation in dependent variable log_Property_Sale_Price. The Condition Number 1.26e+16 suggests that there is severe multicollinearity in the data. The Durbin-Watson test statistics is 1.934 i.e. close to 2.0 and thus it indicates that there is no autocorrelation.

4. Predict the values using test set

```
In [66]: # predict the 'Log_Property_Sale_Price' using predict()
linreg_feature_1_model_predictions = linreg_feature_1_model.predict(X_test)
```



Note that the predicted values are log transformed Property_Sale_Price. In order to get Property_Sale_Price values, we take the antilog of these predicted values by using the function np.exp()

```
In [67]: # take the exponential of predictions using np.exp()
predicted_Property_Sale_Price = np.exp(linreg_feature_1_model_predictions)

# extract the 'Property_Sale_Price' values from the test data
actual_Property_Sale_Price = y_test['Property_Sale_Price']
```

5. Compute accuracy measures

Now we calculate accuracy measures Root-mean-square-error (RMSE), R-squared and Adjusted R-squared.

```
In [68]: # calculate rmse using rmse()
linreg_feature_1_model_rmse = rmse(actual_Property_Sale_Price, predicted_Property_Sale_Price)

# calculate R-squared using rsquared
linreg_feature_1_model_rsquared = linreg_feature_1_model.rsquared

# calculate Adjusted R-Squared using rsquared_adj
linreg_feature_1_model_rsquared_adj = linreg_feature_1_model.rsquared_adj
```

6. Tabulate the results

```
In [69]: # append the accuracy scores to the table
# compile the required information
linreg_feature_1_model_metrics = pd.Series({'Model': "Linreg with new feature (TotalSF)",

                                             'RMSE': linreg_feature_1_model_rmse,
                                             'R-Squared': linreg_feature_1_model_rsquared,
                                             'Adj. R-Squared': linreg_feature_1_model_rsquared_a

                                         }

                                         # append our result table using append()
                                         # ignore_index=True: does not use the index labels
                                         # python can only append a Series if ignore_index=True or if the Series has a name
                                         result_tabulation = result_tabulation.append(linreg_feature_1_model_metrics, ignore_index = True)

                                         # print the result table
                                         result_tabulation
```

Out[69]:

	Model	RMSE	R-Squared	Adj. R-Squared
0	Linreg full model with log of target variable	37726.465264	0.952066	0.935193
1	Linreg full model without log of target variable	54970.829645	0.946898	0.928206
2	Linreg with new feature (TotalSF)	37781.679484	0.952059	0.935271

5.3.2 Multiple Linear Regression (Using New Feature) - 2

In order to build the model, we do the following:

1. Create two new feature by using variables 'Buiding_age' and 'Remodel_age'

```
In [70]: # 'datetime' is used to perform operations related to date and time
import datetime as dt

# 'now().year' returns the current year
current_year = int(dt.datetime.now().year)
```

```
In [71]: # create 2 new variables 'Buiding_age' and 'Remodel_age'
Buiding_age = current_year - df_property.YearBuilt
Remodel_age = current_year - df_property.YearRemodAdd
```

```
In [72]: # append the newly created variables to the dataframe
df_property['Buiding_age'] = Buiding_age
df_property['Remodel_age'] = Remodel_age

# as we have added a new variable using the variables 'YearBuilt' and 'YearRemodAdd', we will drop
# drop the variables using drop()
df_property = df_property.drop(['YearBuilt', 'YearRemodAdd'], axis=1)
```

```
In [73]: # filter out the categorical variables and consider only the numerical variables using (include=np.number)
df_numeric_features = df_property.select_dtypes(include=np.number)

# filter out the numerical variables and consider only the categorical variables using (include=object)
df_categorical_features = df_property.select_dtypes(include = object)
```

Dummy encode the categorical variables

```
In [74]: # use 'get_dummies()' from pandas to create dummy variables
# use 'drop_first = True' to create (n-1) dummy variables
dummy_encoded_variables = pd.get_dummies(df_categorical_features, drop_first = True)
```

Concatenate numerical and dummy encoded categorical variables

```
In [75]: # concatenate the numerical and dummy encoded categorical variables column-wise
df_dummy = pd.concat([df_numeric_features, dummy_encoded_variables], axis=1)

# display data with dummy variables
df_dummy.head()
```

Out[75]:

	LotFrontage	LotArea	MasVnrArea	BsmtFinSF1	BsmtFinSF2	BsmtUnfSF	LowQualFinSF	BsmtFullBath	BsmtHalfBath
0	65.0	8450	196.0	706	0	150	0	1	0
1	80.0	9600	0.0	978	0	284	0	0	1
2	68.0	11250	162.0	486	0	434	0	1	0
3	60.0	9550	0.0	216	0	540	0	1	0
4	84.0	14260	350.0	655	0	490	0	1	0

2. Split the data into train and test sets

```
In [76]: # add the intercept column using 'add_constant()'
df_dummy = sm.add_constant(df_dummy)

# separate the independent and dependent variables
# drop(): drops the specified columns
# axis=1: specifies that the column is to be dropped
X = df_dummy.drop(['Property_Sale_Price','log_Property_Sale_Price'], axis = 1)

# extract the target variable from the data set
y = df_dummy[['Property_Sale_Price','log_Property_Sale_Price']]

# split data into train data and test data
# what proportion of data should be included in test data is specified using 'test_size'
# set 'random_state' to get the same data each time the code is executed
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state = 1)

# check the dimensions of the train & test subset for
# print dimension of predictors train set
print("The shape of X_train is:",X_train.shape)

# print dimension of predictors test set
print("The shape of X_test is:",X_test.shape)

# print dimension of target train set
print("The shape of y_train is:",y_train.shape)

# print dimension of target test set
print("The shape of y_test is:",y_test.shape)
```

The shape of X_train is: (1015, 281)
The shape of X_test is: (436, 281)
The shape of y_train is: (1015, 2)
The shape of y_test is: (436, 2)

3. Build model using sm.OLS().fit()

```
In [77]: # build a OLS model using the function OLS()
# consider the target variable "log_Property_Sale_Price"
# use fit() to fit the model on train data
linreg_feature_2_model = sm.OLS(y_train['log_Property_Sale_Price'], X_train).fit()

# print the summary output
print(linreg_feature_2_model.summary())
```

OLS Regression Results

Dep. Variable:	log_Property_Sale_Price	R-squared:	0.952			
Model:	OLS	Adj. R-squared:	0.935			
Method:	Least Squares	F-statistic:	56.71			
Date:	Wed, 23 Oct 2024	Prob (F-statistic):	0.00			
Time:	17:26:23	Log-Likelihood:	1089.1			
No. Observations:	1015	AIC:	-1650.			
Df Residuals:	751	BIC:	-350.5			
Df Model:	263					
Covariance Type:	nonrobust					
	coef	std err	t	P> t	[0.025	0.975]
const	11.1054	4.074	2.726	0.007	3.108	19.103
LotFrontage	0.0003	0.000	1.411	0.159	-0.000	0.001
LotArea	3.349e-06	5.88e-07	5.697	0.000	2.2e-06	4.5e-06
MasVnrArea	1.219e-05	3.29e-05	0.371	0.711	-5.24e-05	7.68e-05
BsmtFinSF1	2.461e-05	3.28e-05	0.751	0.453	-3.97e-05	8.9e-05
---	---	---	---	---	---	---



This model explains around 95% of the variation in dependent variable log_Property_Sale_Price. The Condition Number 1.28e+16 suggests that there is severe multicollinearity in the data. The Durbin-Watson test statistic is 1.934 and indicates that there is no autocorrelation.

4. Predict the values using test set

```
In [78]: # predict the 'log_Property_Sale_Price' using predict()
linreg_feature_2_model_predictions = linreg_feature_2_model.predict(X_test)
```



Note that the predicted values are log transformed Property_Sale_Price. In order to get Property_Sale_Price values, we take the antilog of these predicted values by using the function np.exp()

```
In [79]: # take the exponential of predictions using np.exp()
predicted_Property_Sale_Price = np.exp(linreg_feature_2_model_predictions)

# extract the 'Property_Sale_Price' values from the test data
actual_Property_Sale_Price = y_test['Property_Sale_Price']
```

5. Compute accuracy measures

Now we calculate accuracy measures Root-mean-square-error (RMSE), R-squared and Adjusted R-squared.

```
In [80]: # calculate rmse using rmse()
linreg_feature_2_model_rmse = rmse(actual_Property_Sale_Price, predicted_Property_Sale_Price)

# calculate R-squared using rsquared
linreg_feature_2_model_rsquared = linreg_feature_2_model.rsquared

# calculate Adjusted R-Squared using rsquared_adj
linreg_feature_2_model_rsquared_adj = linreg_feature_2_model.rsquared_adj
```

6. Tabulate the results

```
In [81]: # append the accuracy scores to the table
# compile the required information
linreg_feature_2_model_metrics = pd.Series({'Model': "Linreg with new features (Building_age and Remodel_age)", 'RMSE': linreg_feature_2_model_rmse, 'R-Squared': linreg_feature_2_model_rsquared, 'Adj. R-Squared': linreg_feature_2_model_rsquared_adj})
# append our result table using append()
# ignore_index=True: does not use the index labels
# python can only append a Series if ignore_index=True or if the Series has a name
result_tabulation = result_tabulation.append(linreg_feature_2_model_metrics, ignore_index = True)

# print the result table
result_tabulation
```

Out[81]:

	Model	RMSE	R-Squared	Adj. R-Squared
0	Linreg full model with log of target variable	37726.465264	0.952066	0.935193
1	Linreg full model without log of target variable	54970.829645	0.946898	0.928206
2	Linreg with new feature (TotalSF)	37781.679484	0.952059	0.935271
3	Linreg with new features (Building_age and Remodel_age)	38063.992077	0.952059	0.935271



RMSE of the model with new features 'Building_age' and 'Remodel_age' is increased. The value of R-squared and adjusted R-squared is same as the previous model.

6. Feature Selection

6.1 Variance Inflation Factor

The Variance Inflation Factor (VIF) is used to detect the presence of multicollinearity between the features. The value of VIF equal to 1 indicates that no features are correlated. We calculate the VIF of the numerical independent variables. VIF for the variable V_i is given as:

$$VIF = 1 / (1 - R\text{-squared})$$

Where, R-squared is the R-squared of the regression model build by regressing one independent variable (say V_i) on all the remaining independent variables (say V_j , $j \neq i$).

```
In [82]: # consider the independent variables in the dataframe 'df_property'
# remove the target variables 'Property_Sale_Price' and 'Log_Property_Sale_Price' using drop() function
df_property_features = df_property.drop(['Property_Sale_Price', 'log_Property_Sale_Price'], axis = 1)

# filter out the categorical variables and consider only the numerical variables using (include=np.number)
df_numeric_features_vif = df_property_features.select_dtypes(include=[np.number])

# display the first five observations
df_numeric_features_vif.head()
```

Out[82]:

	LotFrontage	LotArea	MasVnrArea	BsmtFinSF1	BsmtFinSF2	BsmtUnfSF	LowQualFinSF	BsmtFullBath	BsmtHalfBath
0	65.0	8450	196.0	706	0	150	0	1	0
1	80.0	9600	0.0	978	0	284	0	0	1
2	68.0	11250	162.0	486	0	434	0	1	0
3	60.0	9550	0.0	216	0	540	0	1	0
4	84.0	14260	350.0	655	0	490	0	1	0

Calculate the VIF for each numeric variable.

```
In [83]: # create an empty dataframe to store the VIF for each variable
vif = pd.DataFrame()

# calculate VIF using list comprehension
# use for Loop to access each variable
# calculate VIF for each variable and create a column 'VIF_Factor' to store the values
vif[ "VIF_Factor" ] = [variance_inflation_factor(df_numeric_features_vif.values, i) for i in range(df_numeric_features_vif.shape[1])]

# create a column of variable names
vif[ "Features" ] = df_numeric_features_vif.columns

# sort the dataframe based on the values of VIF_Factor in descending order
# 'reset_index' resets the index of the dataframe
# 'ascending = False' sorts the data in descending order
# 'drop = True' drops the index that was previously created
vif.sort_values('VIF_Factor', ascending = False).reset_index(drop = True)
```

Out[83]:

	VIF_Factor	Features
0	113.617057	TotalSF
1	92.337837	YrSold
2	85.411954	TotRmsAbvGrd
3	38.944336	GarageCars
4	32.012702	GarageYrBlt
5	31.216275	GarageArea
6	30.844830	KitchenAbvGr
7	29.835340	BedroomAbvGr
8	25.726756	FullBath
9	15.281312	LotFrontage
10	12.389799	Buiding_age
11	12.016293	BsmtUnfSF
12	11.826834	BsmtFinSF1
13	8.395213	Remodel_age
14	6.621128	MoSold
15	3.674500	BsmtFullBath
16	2.851753	Fireplaces
17	2.815314	HalfBath
18	2.613935	LotArea
19	1.873705	WoodDeckSF
20	1.817381	OpenPorchSF
21	1.786236	MasVnrArea
22	1.741937	BsmtFinSF2
23	1.416396	EnclosedPorch
24	1.205339	BsmtHalfBath
25	1.187996	ScreenPorch
26	1.115006	LowQualFinSF
27	1.034145	3SsnPorch
28	1.025051	MiscVal

We can see that the variable 'YrSold' has the highest VIF.

We will remove the variables having VIF greater than 10.

We want to remove the variable for which the remaining variables explain more than 90% of the variation and thus we set the threshold to 10.

The value of threshold is completely experimental i.e. it depends on the business requirements. One can choose the threshold other than 10.



```
In [84]: # we will calculate the VIF for each numerical variable
for ind in range(len(df_numeric_features_vif.columns)):

    # create an empty dataframe
    vif = pd.DataFrame()

    # calculate VIF for each variable and create a column 'VIF_Factor' to store the values
    vif["VIF_Factor"] = [variance_inflation_factor(df_numeric_features_vif.values, i) for i in range(len(df_numeric_features_vif.columns))]

    # create a column of feature names
    vif["Features"] = df_numeric_features_vif.columns

    # filter the variables with VIF greater than 10 and store it in a dataframe 'vif_more_than_10'
    # one can choose the threshold other than 10 (it depends on the business requirements)
    vif_more_than_10 = vif[vif['VIF_Factor'] > 10]

    # if dataframe 'vif_more_than_10' is not empty, then sort the dataframe by VIF values
    # if dataframe 'vif_more_than_10' is empty (i.e. all VIF <= 10), then print the dataframe 'vif'
    # 'by' sorts the data using given variable(s)
    # 'ascending = False' sorts the data in descending order
    if(vif_more_than_10.empty == False):
        df_sorted = vif_more_than_10.sort_values(by = 'VIF_Factor', ascending = False)
    else:
        print(vif)
        break

    # if dataframe 'df_sorted' is not empty, then drop the first entry in the column 'Features' from 'vif'
    # select the variable using 'iloc[]'
    # 'axis=1' drops the corresponding column
    # else print the final dataframe 'vif' with all values after removal of variables with VIF less than 10
    if (df_sorted.empty == False):
        df_numeric_features_vif = df_numeric_features_vif.drop(df_sorted.Features.iloc[0], axis=1)
    else:
        print(vif)
```

	VIF_Factor	Features
0	2.466251	LotArea
1	1.716802	MasVnrArea
2	5.588529	BsmtFinSF1
3	1.333163	BsmtFinSF2
4	4.463709	BsmtUnfSF
5	1.058491	LowQualFinSF
6	3.416672	BsmtFullBath
7	1.173941	BsmtHalfBath
8	1.771201	HalfBath
9	2.475114	Fireplaces
10	7.957664	GarageArea
11	1.828450	WoodDeckSF
12	1.735122	OpenPorchSF
13	1.367396	EnclosedPorch
14	1.026525	3SsnPorch
15	1.170999	ScreenPorch
16	1.014968	MiscVal
17	5.593054	MoSold
18	7.320595	Buiding_age
19	6.843799	Remodel_age



The above dataframe contains all the variables with VIF less than 10.

In order to build the model, we do the following

Now, let us build the model using the categorical variables and the numerical variables obtained from VIF.

1. Concatenate numerical and dummy encoded categorical variables

```
In [85]: # lets consider the variables obtained from VIF
# use the dummy variables created previously
# concatenate the numerical and dummy encoded categorical variables
df_dummy = pd.concat([df_numeric_features_vif, dummy_encoded_variables], axis=1)

# display data with dummy variables
df_dummy.head()
```

Out[85]:

	LotArea	MasVnrArea	BsmtFinSF1	BsmtFinSF2	BsmtUnfSF	LowQualFinSF	BsmtFullBath	BsmtHalfBath	HalfBath	Fir
0	8450	196.0	706	0	150	0	1	0	1	
1	9600	0.0	978	0	284	0	0	1	0	0
2	11250	162.0	486	0	434	0	1	0	1	
3	9550	0.0	216	0	540	0	1	0	0	0
4	14260	350.0	655	0	490	0	1	0	1	

2. Split the data into train and test sets

```
# add the intercept column using 'add_constant()'
df_dummy = sm.add_constant(df_dummy)

# consider independent variables
# create a copy of 'df_dummy' and store it as X
X = df_dummy.copy()

# extract the target variable from the data set
y = df_property[['Property_Sale_Price', 'log_Property_Sale_Price']]

# split data into train data and test data
# what proportion of data should be included in test data is specified using 'test_size'
# set 'random_state' to get the same data each time the code is executed
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state = 1)

# check the dimensions of the train & test subset for
# print dimension of predictors train set
print("The shape of X_train is:", X_train.shape)

# print dimension of predictors test set
print("The shape of X_test is:", X_test.shape)

# print dimension of target train set
print("The shape of y_train is:", y_train.shape)

# print dimension of target test set
print("The shape of y_test is:", y_test.shape)
```

The shape of X_train is: (1015, 272)
The shape of X_test is: (436, 272)
The shape of y_train is: (1015, 2)
The shape of y_test is: (436, 2)

3. Build model using sm.OLS().fit()

```
In [87]: # build a full model using OLS()
# consider the target variable log_Property_Sale_Price
# use fit() to fit the model on train data
linreg_vif_model = sm.OLS(y_train['log_Property_Sale_Price'], X_train).fit()

# print the summary output
print(linreg_vif_model.summary())
```

OLS Regression Results

Dep. Variable:	log_Property_Sale_Price	R-squared:	0.938			
Model:	OLS	Adj. R-squared:	0.917			
Method:	Least Squares	F-statistic:	44.96			
Date:	Wed, 23 Oct 2024	Prob (F-statistic):	0.00			
Time:	17:33:00	Log-Likelihood:	955.27			
No. Observations:	1015	AIC:	-1401.			
Df Residuals:	760	BIC:	-145.3			
Df Model:	254					
Covariance Type:	nonrobust					
	coef	std err	t	P> t	[0.025	0.975]
const	5.4969	0.593	9.274	0.000	4.333	6.661
LotArea	4.111e-06	6.5e-07	6.321	0.000	2.83e-06	5.39e-06
MasVnrArea	2.889e-05	3.7e-05	0.780	0.436	-4.38e-05	0.000
BsmtFinSF1	0.0003	2.6e-05	11.781	0.000	0.000	0.000
BsmtFinSF2	0.0002	5e-05	4.688	0.000	0.000	0.000
...



This model explains around 93% of the variation in dependent variable log_Property_Sale_Price. The Condition Number 5.02e+19 suggests that there is severe multicollinearity in the data. The Durbin-Watson test statistics is 1.946 and indicates that there is no autocorrelation.

4. Predict the values using test set

```
In [88]: # predict the 'log_Property_Sale_Price' using predict()
linreg_vif_model_predictions = linreg_vif_model.predict(X_test)
```



Note that the predicted values are log transformed Property_Sale_Price. In order to get Property_Sale_Price values, we take the antilog of these predicted values by using the function np.exp()

```
In [89]: # take the exponential of predictions using np.exp()
predicted_Property_Sale_Price = np.exp(linreg_vif_model_predictions)

# extract the 'Property_Sale_Price' values from the test data
actual_Property_Sale_Price = y_test['Property_Sale_Price']
```

5. Compute accuracy measures

Now we calculate accuracy measures Root-mean-square-error (RMSE), R-squared and Adjusted R-squared.

```
In [90]: # calculate rmse using rmse()
linreg_vif_model_rmse = rmse(actual_Property_Sale_Price, predicted_Property_Sale_Price)

# calculate R-squared using rsquared
linreg_vif_model_rsquared = linreg_vif_model.rsquared

# calculate Adjusted R-Squared using rsquared_adj
linreg_vif_model_rsquared_adj = linreg_vif_model.rsquared_adj
```

6. Tabulate the results

```
In [91]: # append the accuracy scores to the table
# compile the required information
linreg_vif_model_metrics = pd.Series({'Model': "Linreg with VIF",
                                         'RMSE': linreg_vif_model_rmse,
                                         'R-Squared': linreg_vif_model_rsquared,
                                         'Adj. R-Squared': linreg_vif_model_rsquared_adj})

# append our result table using append()
# ignore_index=True: does not use the index labels
# python can only append a Series if ignore_index=True or if the Series has a name
result_tabulation = result_tabulation.append(linreg_vif_model_metrics, ignore_index = True)

# print the result table
result_tabulation
```

Out[91]:

	Model	RMSE	R-Squared	Adj. R-Squared
0	Linreg full model with log of target variable	37726.465264	0.952066	0.935193
1	Linreg full model without log of target variable	54970.829645	0.946898	0.928206
2	Linreg with new feature (TotalSF)	37781.679484	0.952059	0.935271
3	Linreg with new features (Building_age and Rem...	38063.992077	0.952059	0.935271
4	Linreg with VIF	32980.577734	0.937600	0.916745



From the above table we can see that the linear regression with new features has the lowest RMSE value. Thus, it can be concluded that the linear regression model with new features can be used to predict the price of the house.

LR with OLS Project (Property Price Prediction Case Study) - Detailed Video in my YouTube Channel

(<https://youtu.be/muSQMQHgegQ>).

Link to this Video Tutorial (<https://youtu.be/muSQMQHgegQ>)

The End