Housing Prices Project

Sulayman Desai

Abstract

This project required me to predict sale prices from the features of a housing dataset. I used a number of data processing techniques such as imputing missing values, encoding categorical variables, normalising features and removing outliers to clean up the training and testing data sets. Using these cleaned data sets, I fitted various machine learning models such as: Ordinary Least Squares Regression, Ridge Regression, Lasso Regression, Elastic Net Regression, Decision Tree Regressor, Random Forest Regressor and XGBoost Regressor. The **Ridge Regression** model returned the lowest **mean absolute error** of **0.07871660662991475**. It was found that the tree models demanded great computational cost for minimal improvements in accuracy. If this project were to be improved, I would implement feature generation to generate more useful features, used different encoding techniques for greater usage of categorical variables and would develop an ensemble of models to get the best predictions.

Contents

1 **Housing Prices Competition** 3

2 **Importing Libraries** 3

3 **Importing Data** 4

4 **Examining the Datasets** 4

5 **Processing Data** 11
- 5.0.1 Checking skewness and kurtosis . . . 11
- 5.0.2 Sale Price log-transformation . . . 13
- 5.0.3 Missing Values . . . 14
- 5.0.4 Imputation . . . 18
- 5.0.5 Dealing with Outliers . . . 19
- 5.0.6 Encoding Categorical Variables . . . 23
- 5.0.7 Checking skewness and kurtosis of rest of dataset . . . 25
- 5.0.8 And beyond… . . . 29

6 **Cleaning Test Data** 31
- 6.0.1 Missing Values . . . 32
- 6.0.2 Outliers . . . 38
- 6.0.3 Checking skewness . . . 42
- 6.0.4 Categorical Variables . . . 44
- 6.0.5 Comparing Test and Train datasets . . . 46

7 **Modelling** 58
- 7.0.1 Linear Regression . . . 58
- 7.0.2 Ridge Regression . . . 61
- 7.0.3 Lasso Regression . . . 62
- 7.0.4 Elastic Net . . . 63
- 7.0.5 Decision Tree Regressor . . . 64
- 7.0.6 Random Forest Regressor . . . 66
- 7.0.7 XGBoost Regressor . . . 67

8 **Conclusion** 68

1 Housing Prices Competition

In this notebook, I will be using the Housing Prices dataset (https://www.kaggle.com/c/house-prices-advanced-regression-techniques/data) to predict future housing prices by applying different linear regression and decision tree models.

2 Importing Libraries

I am importing the relavant libraries which will be used.

[1]:
```
# Importing packages to deal with dataframes and numerical functions
import pandas as pd
import numpy as np
from scipy import stats

# Importing modules for graphing data
import matplotlib.pyplot as plt
import matplotlib as mpl
import seaborn as sns

# stores plot within notebook
%matplotlib inline

# Importing package which makes plots have same theme as notebook, using jtplot.
↪style()
#from jupyterthemes import jtplot

# corrects behaviour between panda and matplotlib libraries
pd.plotting.register_matplotlib_converters()

# Python Machine Learning (ML) library Scikit-Learn
import sklearn
import xgboost
import category_encoders as ce

# Importing ML models
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Ridge, Lasso
from sklearn.ensemble import GradientBoostingRegressor, RandomForestRegressor
from sklearn.tree import DecisionTreeRegressor
from xgboost import XGBRegressor

# Importing error metrics
from sklearn.metrics import mean_squared_error, mean_absolute_error
from scipy.stats import kurtosis, skew

# Importing data manipulation tools
```

```
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_validate, cross_val_score,␣
 ↪StratifiedKFold
from sklearn.model_selection import KFold
from sklearn.preprocessing import StandardScaler
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import LabelEncoder, OneHotEncoder, LabelBinarizer,␣
 ↪MultiLabelBinarizer
from sklearn.preprocessing import PolynomialFeatures
from sklearn import preprocessing
from scipy.stats import norm

# ensures cross platform functionality
import os
```

3 Importing Data

Now I will import the train, test and sample data.

[2]:
```
# importing training data
train_df = pd.read_csv(r'train.csv')

# importing test data
test_df = pd.read_csv(r'test.csv')

#importing sample submission data
sample_df = pd.read_csv(r'sample_submission.csv')
```

4 Examining the Datasets

I am going to check the structure of the training set, discovering questions such as, how large the data is, what the data contains, are there any missing values etc.

First, let me get a quick look at the first few rows of the data.

[3]:
```
# displaying the first 5 rows of data
train_df.head()
```

[3]:
```
   Id  MSSubClass MSZoning  LotFrontage  LotArea Street Alley LotShape  \
0   1          60       RL         65.0     8450   Pave   NaN      Reg
1   2          20       RL         80.0     9600   Pave   NaN      Reg
2   3          60       RL         68.0    11250   Pave   NaN      IR1
3   4          70       RL         60.0     9550   Pave   NaN      IR1
4   5          60       RL         84.0    14260   Pave   NaN      IR1
```

```
  LandContour Utilities  ... PoolArea PoolQC Fence MiscFeature MiscVal MoSold  \
0         Lvl    AllPub  ...        0    NaN   NaN         NaN       0      2
1         Lvl    AllPub  ...        0    NaN   NaN         NaN       0      5
2         Lvl    AllPub  ...        0    NaN   NaN         NaN       0      9
3         Lvl    AllPub  ...        0    NaN   NaN         NaN       0      2
4         Lvl    AllPub  ...        0    NaN   NaN         NaN       0     12

  YrSold  SaleType  SaleCondition  SalePrice
0   2008        WD         Normal     208500
1   2007        WD         Normal     181500
2   2008        WD         Normal     223500
3   2006        WD        Abnorml     140000
4   2008        WD         Normal     250000

[5 rows x 81 columns]
```

I want to get some general information about the data set which may be useful later on.

[4]:
```
# displaying various statistical information about training data
train_df.describe()
```

[4]:
```
                Id    MSSubClass   LotFrontage        LotArea   OverallQual  \
count  1460.000000   1460.000000   1201.000000    1460.000000   1460.000000
mean    730.500000     56.897260     70.049958   10516.828082      6.099315
std     421.610009     42.300571     24.284752    9981.264932      1.382997
min       1.000000     20.000000     21.000000    1300.000000      1.000000
25%     365.750000     20.000000     59.000000    7553.500000      5.000000
50%     730.500000     50.000000     69.000000    9478.500000      6.000000
75%    1095.250000     70.000000     80.000000   11601.500000      7.000000
max    1460.000000    190.000000    313.000000  215245.000000     10.000000

       OverallCond    YearBuilt  YearRemodAdd   MasVnrArea   BsmtFinSF1  ...  \
count  1460.000000  1460.000000   1460.000000  1452.000000  1460.000000  ...
mean      5.575342  1971.267808   1984.865753   103.685262   443.639726  ...
std       1.112799    30.202904     20.645407   181.066207   456.098091  ...
min       1.000000  1872.000000   1950.000000     0.000000     0.000000  ...
25%       5.000000  1954.000000   1967.000000     0.000000     0.000000  ...
50%       5.000000  1973.000000   1994.000000     0.000000   383.500000  ...
75%       6.000000  2000.000000   2004.000000   166.000000   712.250000  ...
max       9.000000  2010.000000   2010.000000  1600.000000  5644.000000  ...

        WoodDeckSF  OpenPorchSF  EnclosedPorch    3SsnPorch  ScreenPorch  \
count  1460.000000  1460.000000    1460.000000  1460.000000  1460.000000
mean     94.244521    46.660274      21.954110     3.409589    15.060959
std     125.338794    66.256028      61.119149    29.317331    55.757415
min       0.000000     0.000000       0.000000     0.000000     0.000000
```

```
25%       0.000000     0.000000       0.000000     0.000000     0.000000
50%       0.000000    25.000000       0.000000     0.000000     0.000000
75%     168.000000    68.000000       0.000000     0.000000     0.000000
max     857.000000   547.000000     552.000000   508.000000   480.000000

          PoolArea       MiscVal       MoSold       YrSold      SalePrice
count  1460.000000   1460.000000  1460.000000  1460.000000    1460.000000
mean      2.758904     43.489041     6.321918  2007.815753  180921.195890
std      40.177307    496.123024     2.703626     1.328095   79442.502883
min       0.000000      0.000000     1.000000  2006.000000   34900.000000
25%       0.000000      0.000000     5.000000  2007.000000  129975.000000
50%       0.000000      0.000000     6.000000  2008.000000  163000.000000
75%       0.000000      0.000000     8.000000  2009.000000  214000.000000
max     738.000000  15500.000000    12.000000  2010.000000  755000.000000

[8 rows x 38 columns]
```

As the dataset has a large number of columns, not all of them are showing when displaying the dataframe, so I will print all the column names. It could also be useful to know the shape of the dataframe.

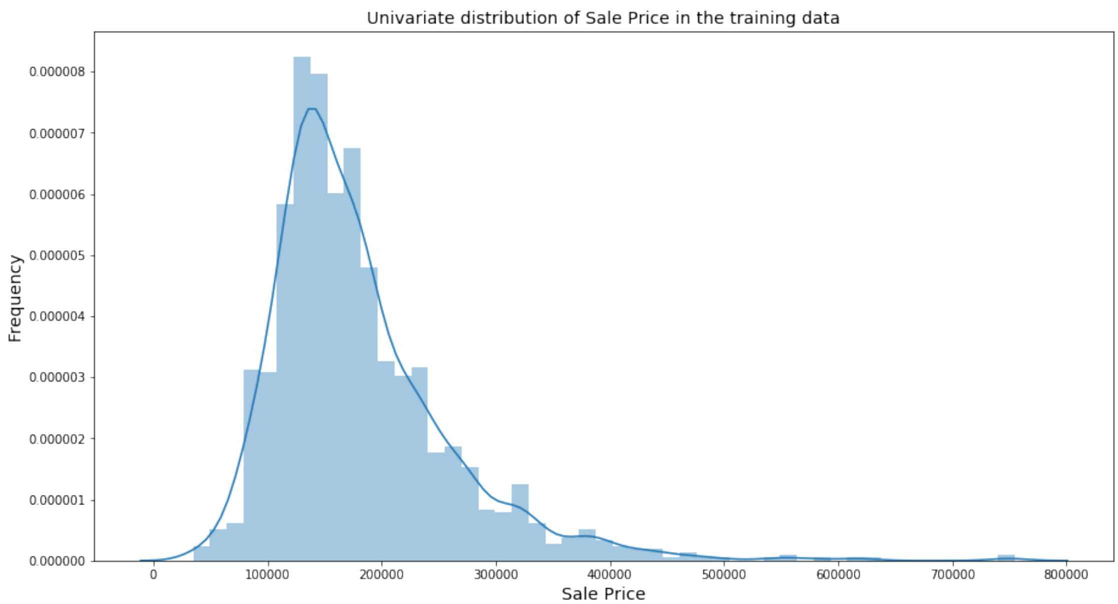
[5]:
```python
# Printing all the column names
print(train_df.columns)
train_df.shape
```

```
Index(['Id', 'MSSubClass', 'MSZoning', 'LotFrontage', 'LotArea', 'Street',
       'Alley', 'LotShape', 'LandContour', 'Utilities', 'LotConfig',
       'LandSlope', 'Neighborhood', 'Condition1', 'Condition2', 'BldgType',
       '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', 'SalePrice'],
      dtype='object')
```

[5]: (1460, 81)

I want to see how varied the instances of Sale Price are in the training data, so I will plot a univariate distribution which shows the probability of occurrence of Sale Price.

[6]:
```
# Setting the size of the figure
plt.figure("sale_price_histogram", figsize=(15,8))

# Giving the graph a title
plt.title("Univariate distribution of Sale Price in the training data", fontsize␣
 ↪= 14)

plt.ylabel("Frequency", fontsize = 14)

plt.xlabel("Sale Price", fontsize = 14)

# Plotting univariate distribution of sale price
sns.distplot(train_df.SalePrice, axlabel = False)

plt.show;
```

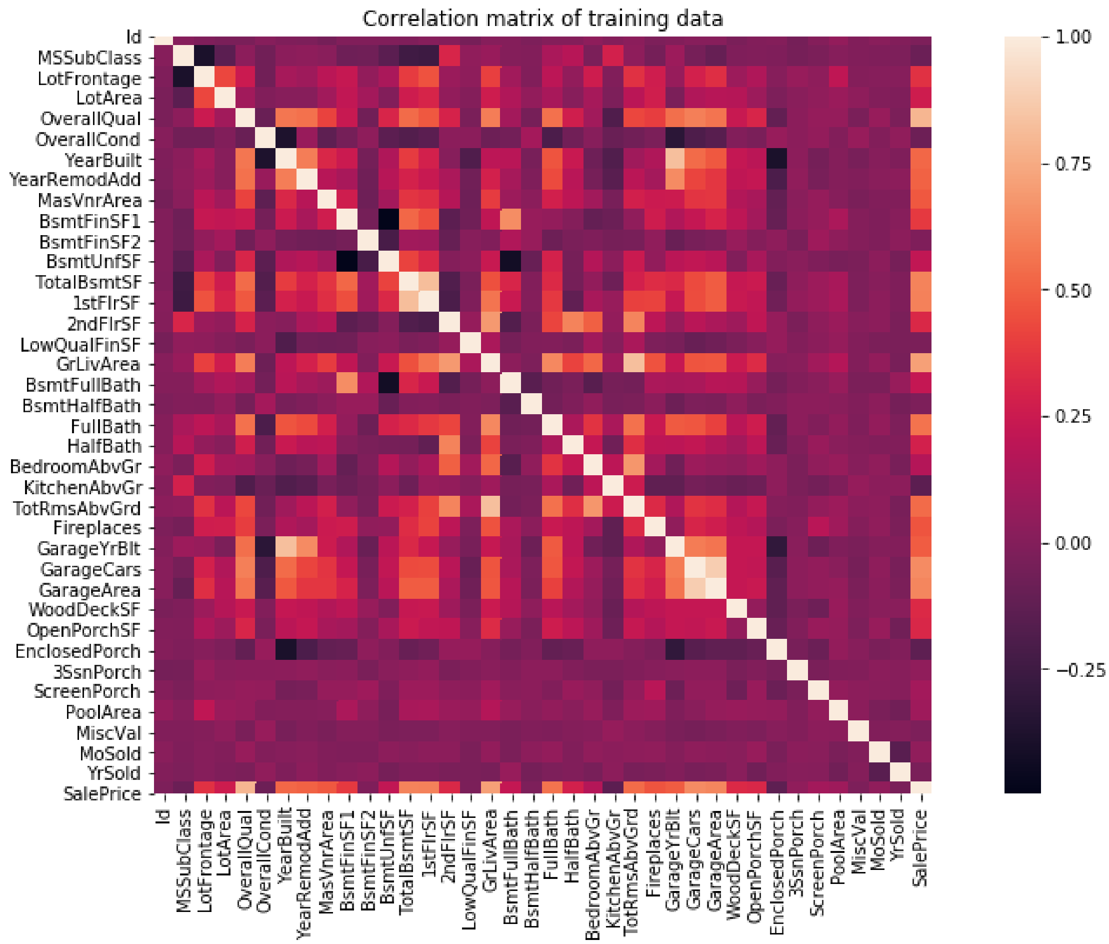


This graph clearly shows that the distrbution peaks somewhere inbetween 100,000-200,000 dollars. The SalePrice has a positive skewness away from the normal distribution.

I think it would be worth to just get a general overview of how correlated all the features are with each other.

[7]:
```
plt.figure('correlation_matrix_training', figsize = (15,8))

# Calculating the correlation matrix of the dataframe train_df
corr_train = train_df.corr()

sns.heatmap(data=corr_train, square = True)
```

```
plt.title('Correlation matrix of training data')

plt.show;
```



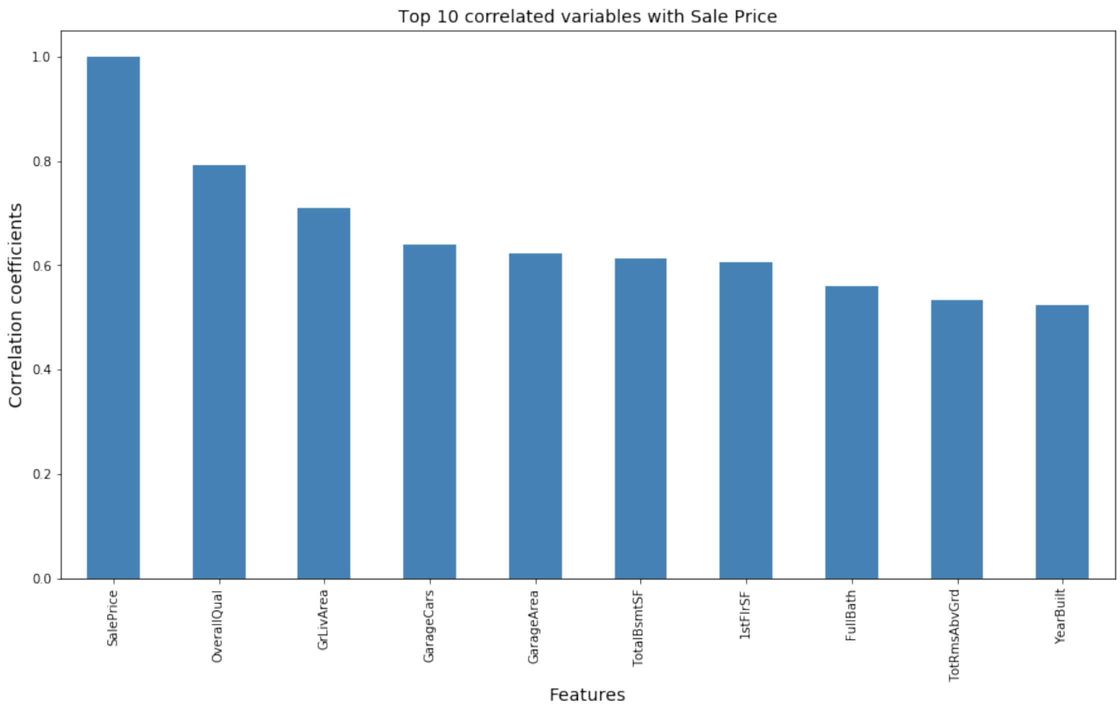
Clearly, this does not provide too much information about the dataset as it is difficult to read. Hence, I will check which features have the most impact on Sale Price by finding the top 10 correlated features.

[8]:
```
# Creating a function which returns the n top correlations with SalePrice
def get_top_correlations(df, n):
    #df.corrwith returns the correlation with SalePrice
    a_corr = df.corrwith(df['SalePrice'])
    # implementing a loop through the rows of the series au_corr to pick the n␣
 ↪largest correlations
    for row in a_corr:
        n_largest = a_corr.nlargest(n)
        return n_largest
```

[9]:
```python
# Calling function get_top_correlations to find the top 10 correlating features␣
↪with SalePrice
correlations_top_10 = get_top_correlations(train_df, 10)

# Printing the correlation coefficients
print(correlations_top_10)
```

```
SalePrice       1.000000
OverallQual     0.790982
GrLivArea       0.708624
GarageCars      0.640409
GarageArea      0.623431
TotalBsmtSF     0.613581
1stFlrSF        0.605852
FullBath        0.560664
TotRmsAbvGrd    0.533723
YearBuilt       0.522897
dtype: float64
```

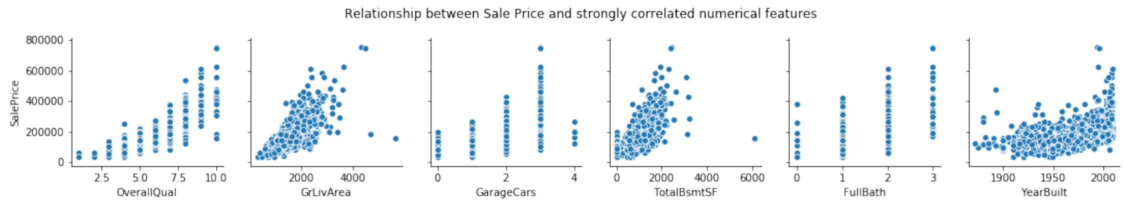
[10]:
```python
#Plotting a bar chart to easily identify which features are most correlated with␣
↪SalePrice

plt.figure('top_10_correlated_saleprice',figsize=(15,8))

plt.title("Top 10 correlated variables with Sale Price", fontsize = 14)

plt.ylabel("Correlation coefficients", fontsize = 14)

plt.xlabel("Features", fontsize = 14)

correlations_top_10.plot.bar(color = 'steelblue')

plt.show;
```



As seen in the barplot, OverallQual, GrLivArea and GarageCars are all strongly correlated with Sale Price. The rest of the features are moderatly correlated with Sale Price.

The above graph shows the top 10 correlated variables with Sale Price. But it would be nice to see scatter graphs of the strongly correlated numerical features with Sale Price. Categorical features (for the moment) cannot demonstrate any correlation with Sale Price as they do not have any values assigned to them.

[11]:
```
# Selecting only the columns which have numerical values and plotting a scatter␣
↪plot against SalePrice
top_correlated_feat = ['SalePrice', 'OverallQual', 'GrLivArea', 'GarageCars',␣
↪'TotalBsmtSF', 'FullBath', 'YearBuilt']

sale_price_scatter = sns.pairplot(train_df[top_correlated_feat], y_vars =␣
↪['SalePrice'],
             x_vars = ['OverallQual', 'GrLivArea', 'GarageCars', 'TotalBsmtSF',␣
↪'FullBath', 'YearBuilt'],
             height = 2.5)

sale_price_scatter.fig.suptitle("Relationship between Sale Price and strongly␣
↪correlated numerical features", y=1.08)

plt.show;
```



The above scatter plots give a very easy visual representation of the relationship of different numerical features with sale price.

As expected, the higher the overall quality of the housebuild, the higher the sale price. This is also the case for ground living area. We had confirmed both of these before from our correlation coefficients, but now we can visually see the relationship.

I also find interesting that houses built after 2000 increase in price a lot more than before 2000. I think it may be worth checking what is different about these houses, and possibly speaking to a housing expert.

5 Processing Data

5.0.1 Checking skewness and kurtosis

In the univariate distribution plot of SalePrice, we saw that the training data had a postive skewness. I want to check the actual value of the skewness and to check another piece of information about the SalePrice, the kurtosis. Skewness is a measure of how symmetric a distribution is, whilst kurtosis is a measure of the heaviness of the tail of the distribution i.e. a measure of the number of outliers in the data.

Skewness can be calculated using the following formula:

$$s = \frac{\sum_{i=1}^{N} (x_i - \bar{x})^3}{\sigma^3}$$

whilst, kurtosis can be calculated using the following formula:

$$K = \frac{1}{N} \sum_{i=1}^{N} \left(\frac{x_i - \mu}{\sigma} \right)^4$$

[12]:
```
# Calculating skewness and kurtosis

skew_SalePrice = skew(train_df['SalePrice'])
print('Skewness of sale price is:', skew_SalePrice)

kurt_SalePrice = kurtosis(train_df['SalePrice'])
print('Kurtosis of sale price is:', kurt_SalePrice)
```

```
Skewness of sale price is: 1.880940746034036
Kurtosis of sale price is: 6.509812011089439
```

As seen in the values of skewness and kurtosis, SalePrice demonstrates it has positive skewness and a significant positive kurtosis. This means the data is not normally distributed, as normally distributed data will have a skewness of 0 and a kurtosis of 0 (Python uses Fisher's definition as default).

Additionally, I will create a probability plot of sale price to see how the data compares to its best fit line.

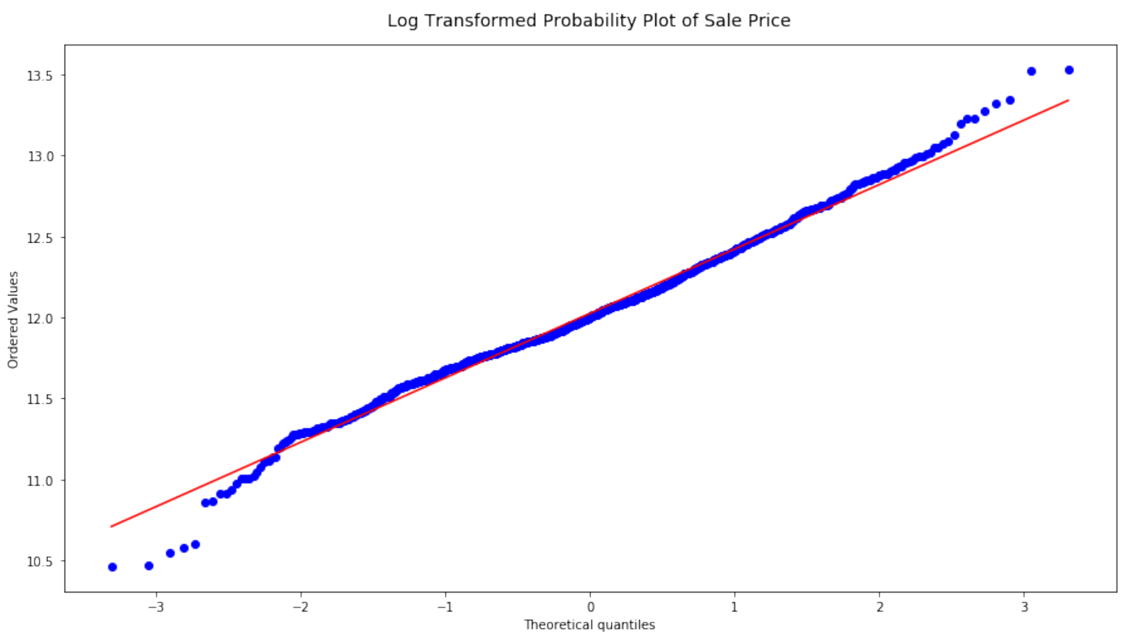
[13]:
```
plt.figure(figsize = (15,8))

# Plotting quantiles of probability of SalePrice
stats.probplot(train_df['SalePrice'], plot = plt)

plt.title("Probability Plot of Sale Price", fontsize = 14, y=1.02)

plt.show;
```

As we can see in the above graph, SalePrice does not follow the diagonal line, further demonstrating that the data is not normally distributed. One may ask about my obesession with comparing the data to a normal distribution? It is because the simplicity of normally distributed data makes it much easier to build more accurate models. Skewed data can violate inbuilt model assumptions and distort which features are more important in model building. For example, in our housing data set, we may train the model on a much larger number of expensive homes, so when making predictions on moderately priced homes, our model will be less accurate.

5.0.2 Sale Price log-transformation

To transform the distribution of SalePrice into a normal distribution, I will use a log-transformation. This is a simple transformation using the log function from numpy

[14]:
```
plt.figure(figsize = (15,8))

#Applying the log function from numpy to all values in the SalePrice column.␣
↪Storing it back in the DF at SalePrice
train_df['SalePrice'] = np.log(train_df.SalePrice)

stats.probplot(train_df['SalePrice'], plot = plt)

plt.title("Log Transformed Probability Plot of Sale Price", fontsize = 14, y=1.
↪02)

plt.show;
```



Clearly, the data follows the straight line much better indicating SalePrice has been normalised. We can verify this by calculating the skewness and kurtosis again.

[15]:
```
skew_SalePrice = skew(train_df['SalePrice'])
print('Skewness of sale price is:', skew_SalePrice)

kurt_SalePrice = kurtosis(train_df['SalePrice'])
print('Kurtosis of sale price is:', kurt_SalePrice)
```

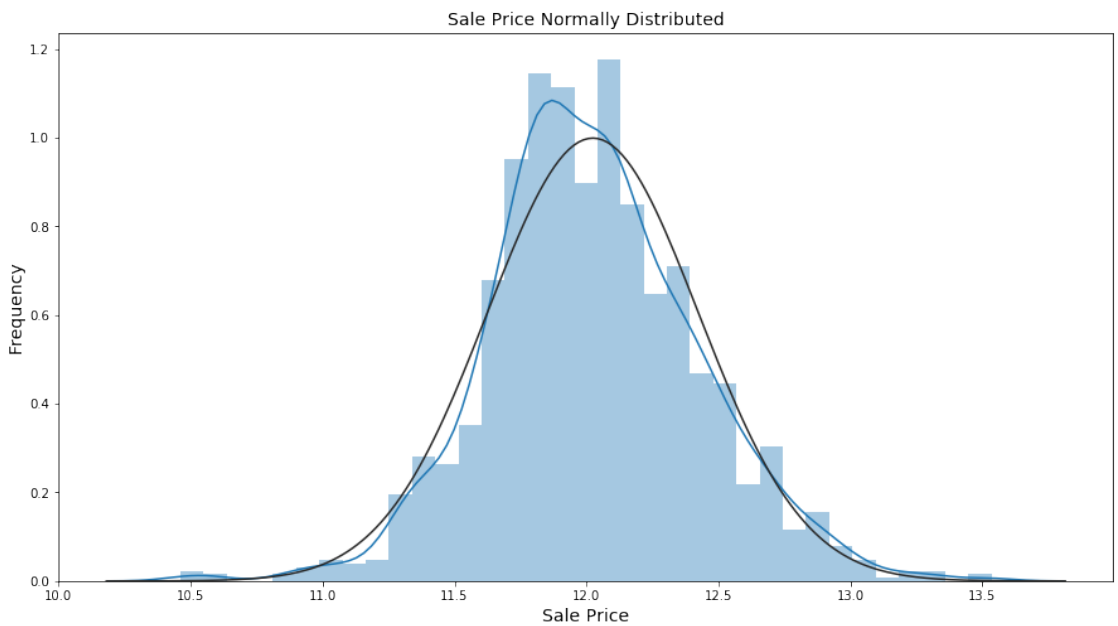
```
Skewness of sale price is: 0.1212103673013655
```

```
Kurtosis of sale price is: 0.8026555069117713
```

Therefore, SalePrice is much closer to a normal distribution and is now more useful for when we build models.

Finally, let me replot the distribution of the SalePrice to see its newly normally didstributed self in all its glory. In this case, I have compared it to a perfect standard normal distribution of the data

[16]:
```
plt.figure(figsize=(15,8))

plt.title("Sale Price Normally Distributed", fontsize = 14)

plt.ylabel("Frequency", fontsize = 14)

plt.xlabel("Sale Price", fontsize = 14)

sns.distplot(train_df.SalePrice, axlabel = False, fit=norm)

plt.show;
```



5.0.3 Missing Values

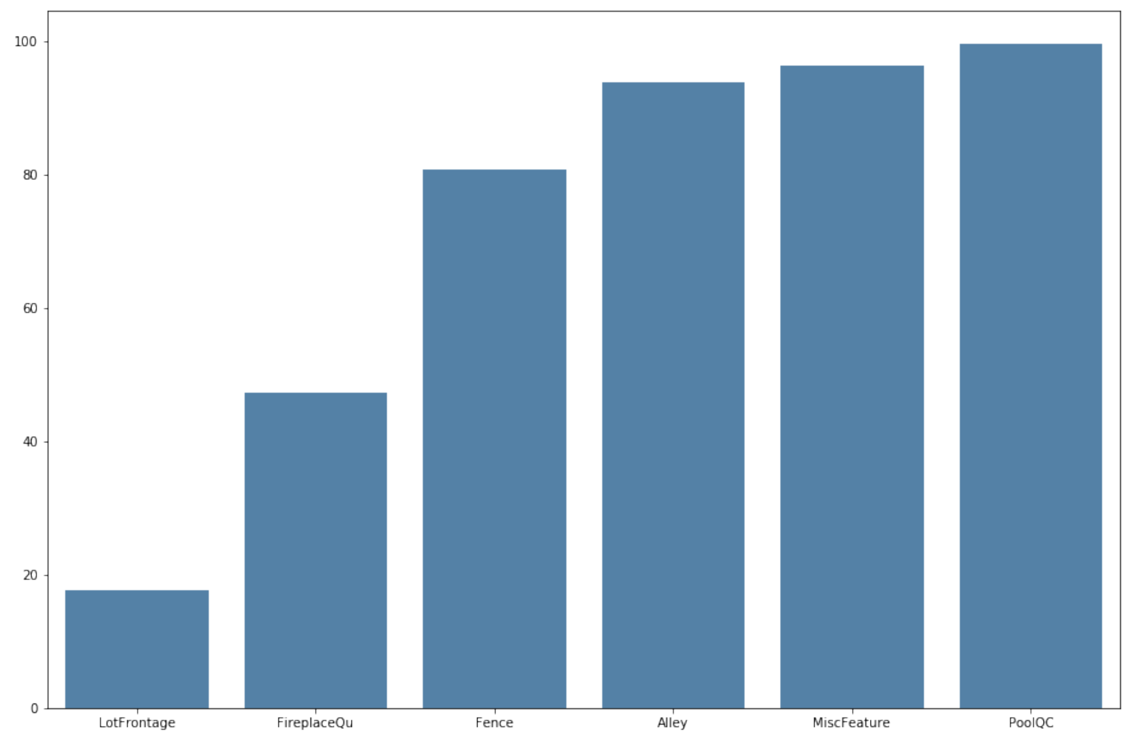
Now I am going to check which columns have lots of missing values, so I can remove those columns from the dataframe. Columns with lots of missing values shouldn't affect SalePrice much.

[17]:
```
#Printing percentage of values missing in each column
missing_lots = train_df.isna().mean() * 100
```

```
#Dropping features with percentage missing values < 6% as these still hold lots␣
↪of information
missing_lots_reduced = missing_lots.drop(missing_lots[missing_lots < 6].index).
↪sort_values()
print(missing_lots_reduced)
```

```
LotFrontage    17.739726
FireplaceQu    47.260274
Fence          80.753425
Alley          93.767123
MiscFeature    96.301370
PoolQC         99.520548
dtype: float64
```

[18]:
```
# Plotting percentage of missing values
plt.figure(figsize = (15,10))

sns.barplot(x = missing_lots_reduced.index, y = missing_lots_reduced, color =␣
↪'steelblue')

plt.show;
```



As we can see in the plot, the features Alley, PoolQC, Fence and MiscFeatures are missing over

80% of their values. This does not necessarily mean that this data was not collected, but most houses just do not have these features. Because of this, these features will not contribute much to our prediction of SalePrice, hence we can drop them from the Dataframe.

I have decided to drop any features missing more than 15% of their values, as we can conclude that there is not enough information left in that feature to build an accurate model. Furthermore, by looking at the individual features which are missing lots of values, I can assume that they are not very important factors to consider when buying a house, therefore I will drop the above features.

[19]:
```
# Dropping features which are missing over 15% of their values
train_df = train_df.drop(columns = missing_lots_reduced.index)
train_df.head()
```

[19]:
```
   Id  MSSubClass MSZoning  LotArea Street LotShape LandContour Utilities  \
0   1          60       RL     8450   Pave      Reg         Lvl    AllPub
1   2          20       RL     9600   Pave      Reg         Lvl    AllPub
2   3          60       RL    11250   Pave      IR1         Lvl    AllPub
3   4          70       RL     9550   Pave      IR1         Lvl    AllPub
4   5          60       RL    14260   Pave      IR1         Lvl    AllPub

  LotConfig LandSlope  ... EnclosedPorch 3SsnPorch ScreenPorch PoolArea  \
0    Inside       Gtl  ...             0         0           0        0
1       FR2       Gtl  ...             0         0           0        0
2    Inside       Gtl  ...             0         0           0        0
3    Corner       Gtl  ...           272         0           0        0
4       FR2       Gtl  ...             0         0           0        0

  MiscVal  MoSold  YrSold  SaleType  SaleCondition  SalePrice
0       0       2    2008        WD         Normal  12.247694
1       0       5    2007        WD         Normal  12.109011
2       0       9    2008        WD         Normal  12.317167
3       0       2    2006        WD        Abnorml  11.849398
4       0      12    2008        WD         Normal  12.429216

[5 rows x 75 columns]
```

Now I will deal with features with only a few missing values.

[20]:
```
missing_small = train_df.isna().mean() * 100

missing_small_reduced = missing_small.drop(missing_small[missing_small == 0].
 ↪index).sort_values()
print(missing_small_reduced)
```

```
Electrical      0.068493
MasVnrType      0.547945
MasVnrArea      0.547945
BsmtQual        2.534247
BsmtCond        2.534247
```

```
BsmtFinType1    2.534247
BsmtExposure    2.602740
BsmtFinType2    2.602740
GarageType      5.547945
GarageYrBlt     5.547945
GarageFinish    5.547945
GarageQual      5.547945
GarageCond      5.547945
dtype: float64
```

[21]:
```
#Counting the number of missing values for each feature
missing_small_count = train_df.isnull().sum()
missing_small_count_reduced = missing_small_count.
 ↪drop(missing_small_count[missing_small_count == 0].index).sort_values()
print(missing_small_count_reduced)
```

```
Electrical       1
MasVnrType       8
MasVnrArea       8
BsmtQual        37
BsmtCond        37
BsmtFinType1    37
BsmtExposure    38
BsmtFinType2    38
GarageType      81
GarageYrBlt     81
GarageFinish    81
GarageQual      81
GarageCond      81
dtype: int64
```

Electrical: this feature only has one missing value, so we will just remove that row from the dataframe.

MasVnrType, MasVnrArea, BsmtQual, BsmtCond, BsmtFinType1, BsmtExposure, and BsmtFinType2: all of these features do not have many missing values so we will remove them from the dataframe. Many of these rows overlap so we would only be removing a maximum of 46 rows which only represents 2.6% of the dataframe. So the effect on our model will be minimal.

GarageType: this feature holds useful information about the garage location and if there is no garage so I will impute these 81 values later on.

GarageYrBlt, GarageFinish, GarageQual and GarageCond: these features are all closely related. Hence I will keep GarageQual and remove the rest of the features. The 81 missing values will be imputed later on.

[22]:
```
#Removing various rows and features from dataframe
train_df = train_df.dropna(subset = ['Electrical', 'MasVnrType', 'MasVnrArea',␣
 ↪'BsmtQual', 'BsmtCond',
```

```
                                               'BsmtFinType1', 'BsmtExposure',␣
 ↪'BsmtFinType2'])
train_df = train_df.drop(columns = ['GarageYrBlt', 'GarageFinish', 'GarageCond'])
```

[23]:
```
train_df.head()
```

[23]:
```
   Id  MSSubClass MSZoning  LotArea Street LotShape LandContour Utilities  \
0   1          60       RL     8450   Pave      Reg         Lvl    AllPub
1   2          20       RL     9600   Pave      Reg         Lvl    AllPub
2   3          60       RL    11250   Pave      IR1         Lvl    AllPub
3   4          70       RL     9550   Pave      IR1         Lvl    AllPub
4   5          60       RL    14260   Pave      IR1         Lvl    AllPub

  LotConfig LandSlope  ... EnclosedPorch 3SsnPorch ScreenPorch PoolArea  \
0    Inside       Gtl  ...             0         0           0        0
1       FR2       Gtl  ...             0         0           0        0
2    Inside       Gtl  ...             0         0           0        0
3    Corner       Gtl  ...           272         0           0        0
4       FR2       Gtl  ...             0         0           0        0

  MiscVal  MoSold  YrSold  SaleType  SaleCondition  SalePrice
0       0       2    2008        WD         Normal  12.247694
1       0       5    2007        WD         Normal  12.109011
2       0       9    2008        WD         Normal  12.317167
3       0       2    2006        WD        Abnorml  11.849398
4       0      12    2008        WD         Normal  12.429216

[5 rows x 72 columns]
```

5.0.4 Imputation

When I was removing missing values, I was still left with missing values in GarageType and GarageQual. I will now impute these values. Imputation is the process by which you replace a missing value with some other value e.g. mean or in a categorical features case, the most popular type of feature. First, lets get some more information about GarageType and GarageQual.

[24]:
```
# Counting the number of instances in a feature
print('No. of instances of values in GarageType: \n', train_df['GarageType'].
 ↪value_counts(dropna = False))
print('\n No. of instances of values in GarageQual: \n', train_df['GarageQual'].
 ↪value_counts(dropna = False))
```

```
No. of instances of values in GarageType:
 Attchd     852
Detchd     369
BuiltIn     85
NaN         74
```

```
Basment      19
CarPort       7
2Types        6
Name: GarageType, dtype: int64

 No. of instances of values in GarageQual:
 TA     1270
NaN      74
Fa       48
Gd       14
Ex        3
Po        3
Name: GarageQual, dtype: int64
```

It seems most houses have garages. Therefore for GarageType, I will impute the missing values with Attchd and for GarageQual I will impute the missing values with TA.

[25]:
```
#Filling missing values in GarageType with Attchd
train_df['GarageType'].fillna(value = 'Attchd', inplace = True)
```

[26]:
```
#Filling missing values in GarageQUal with TA
train_df['GarageQual'].fillna(value = 'TA', inplace = True)
```

[27]:
```
#Checking that I have dealt with all of the missing values
missing_check = train_df.isna().mean() * 100
missing_check = missing_check.drop(missing_check[missing_check == 0].index).
 ↪sort_values()
print(missing_check)
```

```
Series([], dtype: float64)
```

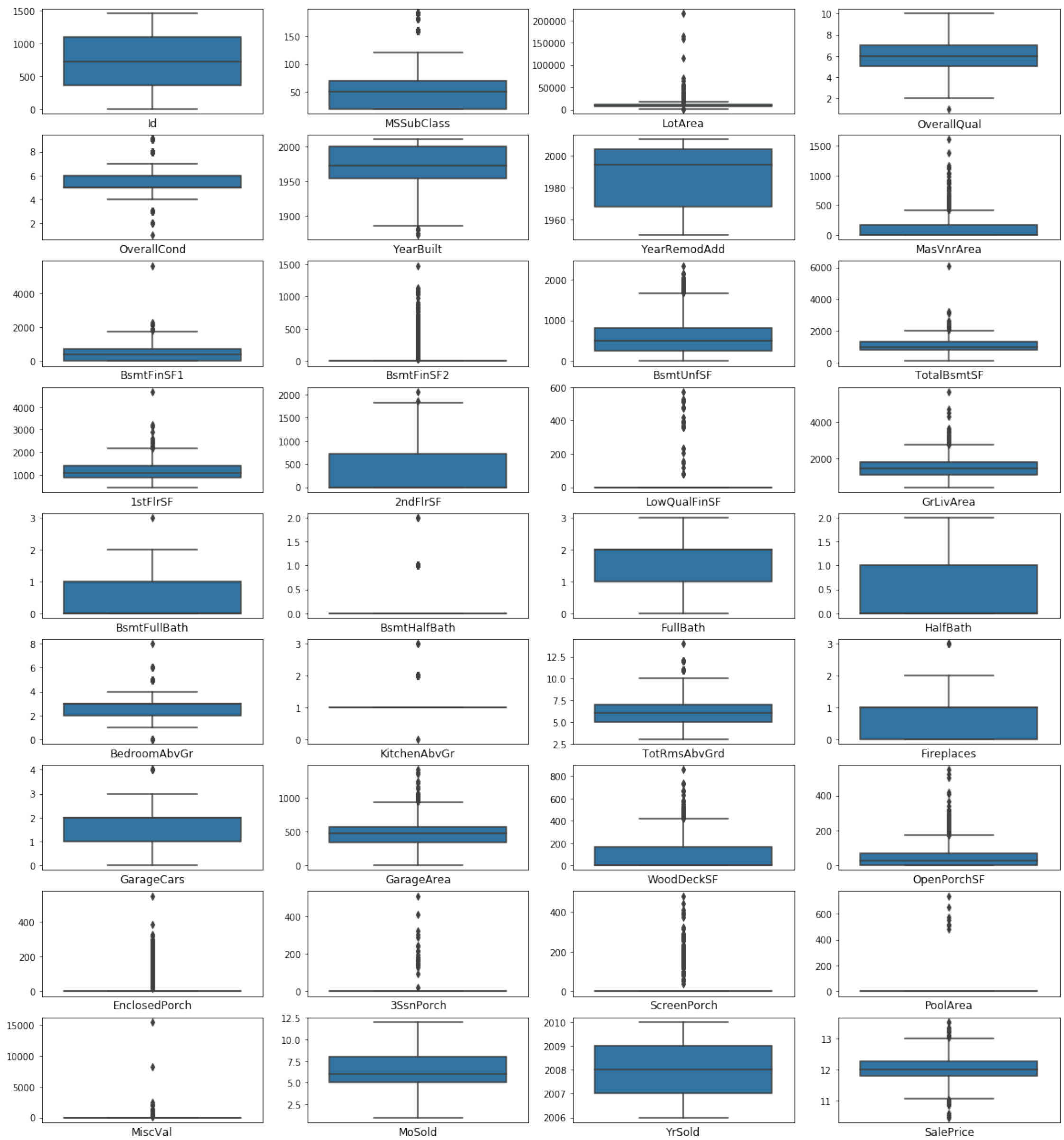
Clearly, there are no longer any missing values.

5.0.5 Dealing with Outliers

Outliers are pieces of data which do not match the overall distribution of its dataset. These datapoints can arise due to mistakes or variances in the data, and if they detract from the overall information of the dataset then we need to identify and remove them. We can tell if a particular datapoint is an outlier if it lies outside the range of a histogram of all the datapoints of that particular feature. This process is known as univariate analysis.

[28]:
```
plt.figure(figsize=(20,25))

numerical_cols = train_df.select_dtypes(exclude = 'object')

#This 'for' loop allows me to loop over numerical_cols using enumerate, whilst␣
 ↪keeping track of the index in i
for i, col in enumerate(numerical_cols.columns):
```

```
    # axes allows you to plot multiple axes and do things to the individual axes (i.e. label them)
    axes = plt.subplot(10,4,i+1)
    sns.boxplot(data = train_df[col])
    axes.set_xlabel(col, fontsize = 12)
    axes.tick_params(axis='x', which='both', bottom=False, top=False, labelbottom=False)

plt.tight_layout
plt.show;
```



As we can see in the graphs above, many of the features display outliers. I am going to remove some of them from the data set to build a more accurate model. However, I will not remove all of the outliers. Some features only have one or two outliers so they have a very minimal impact on the dataset, so it is safe to leave them in. Some outliers are still quite close to the other values in that feature, so do not need to be removed.

For some of the histograms, it seems they do not have enough data to produce a histogram and demonstrate many outliers (e.g. EnclosedPorch, 3SsnPorch, ScreenPorch, PoolArea, and MiscVal etc). These points are misleading, as they do not really represent outliers, but rather most houses do not have these features hence there not being enough data to produce a histogram. Therefore, these datapoints are not outliers and can remain as part of our dataset.

To choose which outliers from a particular feature which need to be removed, I still need some more information. As the business question is related to SalePrice, I am interested in how these features affect SalePrice. Therefore, I am going to conduct some bivariate analysis and plot scatter plots of the numerical features against SalePrice, to visually see how SalePrice is dependent on each feature.

[29]:
```
plt.figure(figsize=(20,25))

for i, col in enumerate(numerical_cols.columns):
    axes = plt.subplot(10,4,i+1)
    sns.scatterplot(x = train_df[col], y = train_df['SalePrice'])
    axes.set_ylabel('SalePrice', fontsize = 12)

plt.tight_layout(pad = 1.0)
plt.show;
```

Based on looking at the number of outliers from the univariate analysis and checking how points vary with SalePrice using bivariate analysis, I have concluded that only the following features need outliers to be removed:

- GrLivArea
- SalePrice

To decide which outliers to remove and to keep in these two features, I will use the 1.5 Interquartile Range (IQR) rule, where any points below (First Quartile - 1.5IQR) and above (Third Quartile + 1.5IQR) will be removed.

[30]:
```
#Calculating IQR for GrLivArea and SalePrice
q1_GrLivArea, q3_GrLivArea = np.percentile(train_df['GrLivArea'], [25,75])
iqr_GrLivArea = q3_GrLivArea - q1_GrLivArea

q1_SalePrice, q3_SalePrice = np.percentile(train_df['SalePrice'], [25,75])
iqr_SalePrice = q3_SalePrice - q1_SalePrice
```

[31]:
```
#Calculating lower and upper bound thresholds for outliers
lower_bound_GrLivArea = q1_GrLivArea -(1.5 * iqr_GrLivArea)
upper_bound_GrLivArea = q3_GrLivArea +(1.5 * iqr_GrLivArea)

lower_bound_SalePrice = q1_SalePrice -(1.5 * iqr_SalePrice)
upper_bound_SalePrice = q3_SalePrice +(1.5 * iqr_SalePrice)
```

[32]:
```
# Creating booleans to find outliers
outliers = train_df[(train_df['GrLivArea'] < lower_bound_GrLivArea) |␣
 ↪(train_df['GrLivArea'] > upper_bound_GrLivArea)
                     | (train_df['SalePrice'] < lower_bound_SalePrice)
                     | (train_df['SalePrice'] > upper_bound_SalePrice)].index

train_df = train_df.drop(outliers)
```

[33]:
```
# Checking outlier rows have been dropped
train_df.shape
```

[33]:
```
(1362, 72)
```

5.0.6 Encoding Categorical Variables

Most Machine Learning models cannot pass categorical variables into the predictions, therefore I am going to covert them into a usable format. But first, I need to change the type of data represented in some features as they are misleading.

[34]:
```
# Changing these features back into strings as currently the dataframe sees them␣
 ↪as integers
train_df['MSSubClass'] = str(train_df['MSSubClass'])
train_df['OverallCond'] = str(train_df['OverallCond'])
train_df['OverallQual'] = str(train_df['OverallQual'])
train_df['YrSold'] = str(train_df['YrSold'])
train_df['MoSold'] = str(train_df['MoSold'])
```

[35]:
```
train_df.select_dtypes(exclude = 'object').columns # will need later
```

[35]:
```
Index(['Id', 'LotArea', 'YearBuilt', 'YearRemodAdd', 'MasVnrArea',
       'BsmtFinSF1', 'BsmtFinSF2', 'BsmtUnfSF', 'TotalBsmtSF', '1stFlrSF',
       '2ndFlrSF', 'LowQualFinSF', 'GrLivArea', 'BsmtFullBath', 'BsmtHalfBath',
       'FullBath', 'HalfBath', 'BedroomAbvGr', 'KitchenAbvGr', 'TotRmsAbvGrd',
```

```
       'Fireplaces', 'GarageCars', 'GarageArea', 'WoodDeckSF', 'OpenPorchSF',
       'EnclosedPorch', '3SsnPorch', 'ScreenPorch', 'PoolArea', 'MiscVal',
       'SalePrice'],
      dtype='object')
```

I will use a strategy called one-hot encoding which gives a value of 1 for each type which occurs in a particular categorical feature and gives a value of 0 if that type is not present. This is a simple way to encode all of the categorical features and should prove useful

[36]:
```
train_df = pd.get_dummies(train_df)
train_df.head()
```

[36]:
```
   Id  LotArea  YearBuilt  YearRemodAdd  MasVnrArea  BsmtFinSF1  BsmtFinSF2  \
0   1     8450       2003          2003       196.0         706           0
1   2     9600       1976          1976         0.0         978           0
2   3    11250       2001          2002       162.0         486           0
3   4     9550       1915          1970         0.0         216           0
4   5    14260       2000          2000       350.0         655           0

   BsmtUnfSF  TotalBsmtSF  1stFlrSF  ...  SaleType_ConLw  SaleType_New  \
0        150          856       856  ...               0             0
1        284         1262      1262  ...               0             0
2        434          920       920  ...               0             0
3        540          756       961  ...               0             0
4        490         1145      1145  ...               0             0

   SaleType_Oth  SaleType_WD  SaleCondition_Abnorml  SaleCondition_AdjLand  \
0             0            1                      0                      0
1             0            1                      0                      0
2             0            1                      0                      0
3             0            1                      1                      0
4             0            1                      0                      0

   SaleCondition_Alloca  SaleCondition_Family  SaleCondition_Normal  \
0                     0                     0                     1
1                     0                     0                     1
2                     0                     0                     1
3                     0                     0                     0
4                     0                     0                     1

   SaleCondition_Partial
0                      0
1                      0
2                      0
3                      0
4                      0
```

```
[5 rows x 255 columns]
```

5.0.7 Checking skewness and kurtosis of rest of dataset

Only the numerical features demonstrate skewness and kurtosis, so we will select these columns first. And then we will check their skewness and kurtosis to decide whether we will log transform the feature. NB: we will not do this for categorical features which have been encoded. I will only use the numerical features which are not discrete (e.g. YearBuilt etc.)

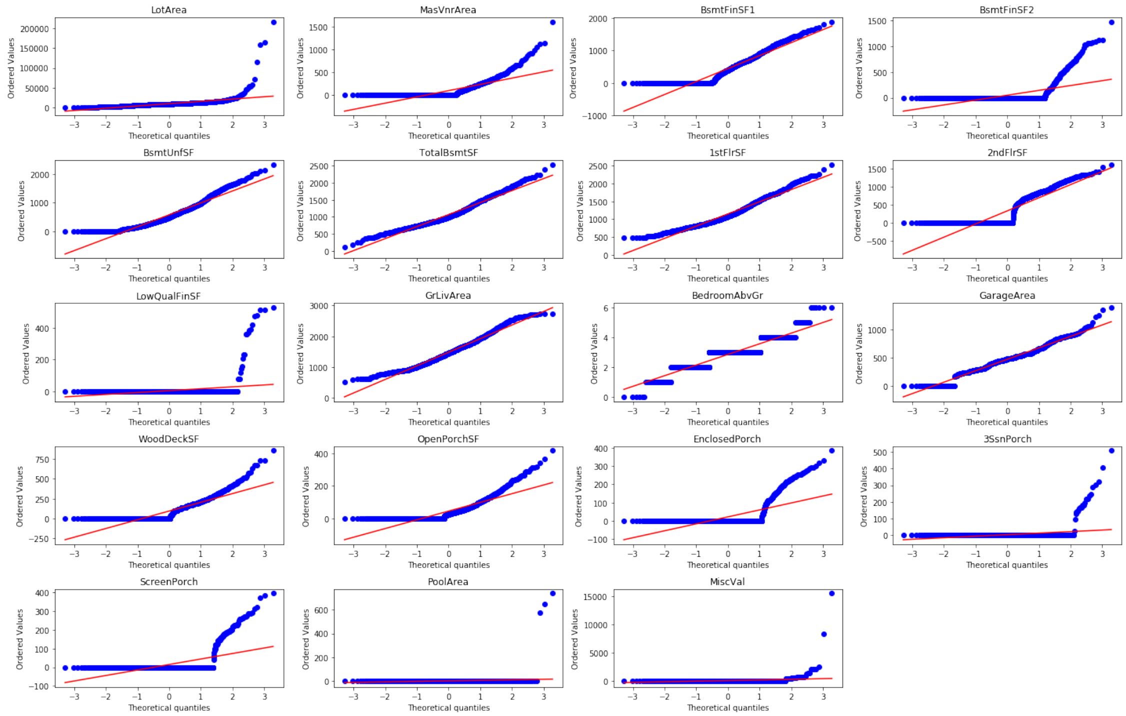
[37]:
```
numerical_columns = ['LotArea',  'MasVnrArea',
       'BsmtFinSF1', 'BsmtFinSF2', 'BsmtUnfSF', 'TotalBsmtSF', '1stFlrSF',
       '2ndFlrSF', 'LowQualFinSF', 'GrLivArea',
       'BedroomAbvGr', 'GarageArea', 'WoodDeckSF', 'OpenPorchSF',
       'EnclosedPorch', '3SsnPorch', 'ScreenPorch', 'PoolArea', 'MiscVal']


for i, col in enumerate(numerical_columns):
    skewness = train_df[col].skew()
    kurt = train_df[col].kurtosis()
    print('Skewness of', col, 'is', skewness)
    print('Kurtosis of', col, 'is', kurt, '\n')
```

```
Skewness of LotArea is 12.728694101002754
Kurtosis of LotArea is 213.67938372752775

Skewness of MasVnrArea is 2.590881997294897
Kurtosis of MasVnrArea is 10.073190763893653

Skewness of BsmtFinSF1 is 0.6687200389155128
Kurtosis of BsmtFinSF1 is -0.3804752689892852

Skewness of BsmtFinSF2 is 4.222610786234709
Kurtosis of BsmtFinSF2 is 19.68631550787231

Skewness of BsmtUnfSF is 0.9304248401095043
Kurtosis of BsmtUnfSF is 0.5284944356251353

Skewness of TotalBsmtSF is 0.69746052606669
Kurtosis of TotalBsmtSF is 0.26837006104123384

Skewness of 1stFlrSF is 0.697982873735981
Kurtosis of 1stFlrSF is 0.10848807187995524

Skewness of 2ndFlrSF is 0.7119746132221992
Kurtosis of 2ndFlrSF is -0.9668470217148708

Skewness of LowQualFinSF is 10.161640363702475
```

```
Kurtosis of LowQualFinSF is 107.23029942840147

Skewness of GrLivArea is 0.4829127905098185
Kurtosis of GrLivArea is -0.1948566522712274

Skewness of BedroomAbvGr is 0.008206513816296715
Kurtosis of BedroomAbvGr is 1.6499527281015074

Skewness of GarageArea is 0.17071879537395862
Kurtosis of GarageArea is 0.90699530930583

Skewness of WoodDeckSF is 1.5377435286917205
Kurtosis of WoodDeckSF is 3.113487147788757

Skewness of OpenPorchSF is 1.874472931805437
Kurtosis of OpenPorchSF is 4.1642698234699544

Skewness of EnclosedPorch is 2.9089863253866723
Kurtosis of EnclosedPorch is 7.913576535857889

Skewness of 3SsnPorch is 10.189015567275886
Kurtosis of 3SsnPorch is 120.48183060568954

Skewness of ScreenPorch is 3.8333486778336594
Kurtosis of ScreenPorch is 14.847027314873504

Skewness of PoolArea is 21.586982953398458
Kurtosis of PoolArea is 469.29174240478625

Skewness of MiscVal is 25.334782804784542
Kurtosis of MiscVal is 726.9251286699115
```

[38]:

```
plt.figure(figsize = (20,25))

for i, col in enumerate(numerical_columns):
    axes = plt.subplot(10,4,i+1)
    stats.probplot(train_df[col], plot = plt)
    plt.title(col)

plt.tight_layout(pad = 1.0)
```



Ok, now we have the skewness, kurtosis and probability plots of the numerical features, lets decide which features need to be log-transformed. As ideal values of skewness and kurtosis are both 0, some features have skewness and kurtosis quite close to this value. Hence I will define a threshold of skewness of above 0.75 for features to be log transformed. As features who have low skewness also have a low kurtosis, we do not need to set a threshold for kurtosis.

[39]:
```
numerical_columns = ['LotArea',  'MasVnrArea',
        'BsmtFinSF1', 'BsmtFinSF2', 'BsmtUnfSF', 'TotalBsmtSF', '1stFlrSF',
        '2ndFlrSF', 'LowQualFinSF', 'GrLivArea',
        'BedroomAbvGr', 'GarageArea', 'WoodDeckSF', 'OpenPorchSF',
        'EnclosedPorch', '3SsnPorch', 'ScreenPorch']


def high_skew(dataframe, colls):
    """
    This function will only keep features which have high skewness

    Input: Dataframe

    Output: Array with features which have high skew

    """

    #Creating empty array to store high skewed features
```

```
    high_skewed_features = []

    for i, col in enumerate(colls):

        skewness_df = dataframe[col].skew()

        #Condition for which features to keep
        if skewness_df > 1:

            #Adding the feature name to array
            high_skewed_features.append(col)

    return high_skewed_features
```

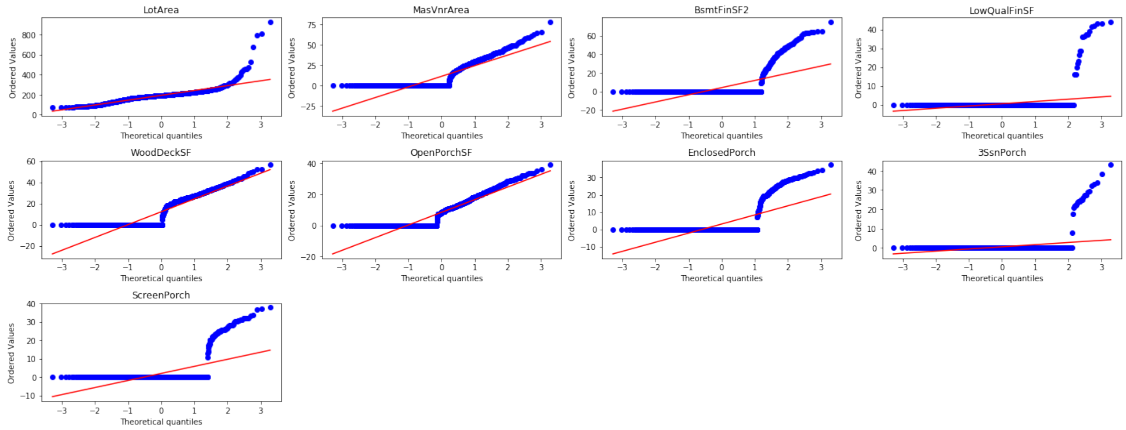
[40]:
```
#Calling function on train_df
high_skewed_features_train_df = high_skew(train_df, numerical_columns)
print(high_skewed_features_train_df)
```

```
['LotArea', 'MasVnrArea', 'BsmtFinSF2', 'LowQualFinSF', 'WoodDeckSF',
'OpenPorchSF', 'EnclosedPorch', '3SsnPorch', 'ScreenPorch']
```

[41]:
```
import scipy

#Performing a boxcox transformation to manage skewness
high_skewed_features_train_df_box = scipy.special.
 ↪boxcox1p(train_df[high_skewed_features_train_df], 0.5)
```

Unlike before when I used a log transformation, now my features have negative values hence a log transformation does not work. Therefore, I use a boxcox transformation. For more information about how boxcox transformations: https://docs.scipy.org/doc/scipy/reference/generated/scipy.special.boxcox1p.html

[42]:
```
plt.figure(figsize = (20,25))

#Checking new lines of best fit
for i, col in enumerate(high_skewed_features_train_df):
    axes = plt.subplot(10,4,i+1)
    stats.probplot(high_skewed_features_train_df_box[col], plot = plt)
    plt.title(col)

plt.tight_layout(pad = 1.0)
```



[43]:
```python
#Checking skewness is reduced
for i, col in enumerate(high_skewed_features_train_df):
    skewness = high_skewed_features_train_df_box[col].skew()
    print('Skewness of', col, 'is', skewness)
```

```
Skewness of LotArea is 4.368044693263819
Skewness of MasVnrArea is 1.0535272749685074
Skewness of BsmtFinSF2 is 3.0588149311390076
Skewness of LowQualFinSF is 9.017708889454473
Skewness of WoodDeckSF is 0.5064592144802054
Skewness of OpenPorchSF is 0.6400877188563249
Skewness of EnclosedPorch is 2.381578795644092
Skewness of 3SsnPorch is 8.270539021633104
Skewness of ScreenPorch is 3.30897987124542
```

As we can see the skewness has been reduced for the high skew features. Although some still have high skews (skewness > 5), this could be because of a number of reasons, such as less houses having this feature or the houses which have these features are very different to the rest of the dataset. Therefore, as only a few features demonstrate this high skewness, we can just ignore it as it won't have a large effect on our model.

[44]:
```python
#Dropping columns with skewed values and replacing them with transformed values

train_df[high_skewed_features_train_df] = high_skewed_features_train_df_box
```

5.0.8 And beyond...

Now my training dataset is finally ready for modelling. Let me show my dataframe one last time.

[45]:
```python
train_df
```

[45]:
```
        Id     LotArea  YearBuilt  YearRemodAdd  MasVnrArea  BsmtFinSF1  \
0        1  181.858641       2003          2003   26.071338         706
```

```
1        2  193.969385       1976          1976    0.000000          978
2        3  210.141462       2001          2002   23.534291          486
3        4  193.458435       1915          1970    0.000000          216
4        5  236.838858       2000          2000   35.469988          655
...    ...         ...        ...           ...         ...          ...
1455  1456  175.966289       1999          2000    0.000000            0
1456  1457  227.573518       1978          1988   19.908902          790
1457  1458  188.189379       1941          2006    0.000000          275
1458  1459  195.159834       1950          1996    0.000000           49
1459  1460  197.379036       1965          1965    0.000000          830

      BsmtFinSF2  BsmtUnfSF  TotalBsmtSF  1stFlrSF  ...  SaleType_ConLw  \
0       0.000000        150          856       856  ...               0
1       0.000000        284         1262      1262  ...               0
2       0.000000        434          920       920  ...               0
3       0.000000        540          756       961  ...               0
4       0.000000        490         1145      1145  ...               0
...          ...        ...          ...       ...  ...             ...
1455    0.000000        953          953       953  ...               0
1456   23.612497        589         1542      2073  ...               0
1457    0.000000        877         1152      1188  ...               0
1458   62.187226          0         1078      1078  ...               0
1459   32.117444        136         1256      1256  ...               0

      SaleType_New  SaleType_Oth  SaleType_WD  SaleCondition_Abnorml  \
0                0             0            1                      0
1                0             0            1                      0
2                0             0            1                      0
3                0             0            1                      1
4                0             0            1                      0
...            ...           ...          ...                    ...
1455             0             0            1                      0
1456             0             0            1                      0
1457             0             0            1                      0
1458             0             0            1                      0
1459             0             0            1                      0

      SaleCondition_AdjLand  SaleCondition_Alloca  SaleCondition_Family  \
0                         0                     0                     0
1                         0                     0                     0
2                         0                     0                     0
3                         0                     0                     0
4                         0                     0                     0
...                     ...                   ...                   ...
1455                      0                     0                     0
1456                      0                     0                     0
1457                      0                     0                     0
```

```
1458                        0                     0                     0
1459                        0                     0                     0

      SaleCondition_Normal  SaleCondition_Partial
0                        1                      0
1                        1                      0
2                        1                      0
3                        0                      0
4                        1                      0
...                    ...                    ...
1455                     1                      0
1456                     1                      0
1457                     1                      0
1458                     1                      0
1459                     1                      0

[1362 rows x 255 columns]
```

6 Cleaning Test Data

This section will use many of the techniques and decisions I made when processing the training dataset.

[46]:

```
test_df
```

[46]:

```
        Id  MSSubClass MSZoning  LotFrontage  LotArea Street Alley LotShape  \
0     1461          20       RH         80.0    11622   Pave   NaN      Reg
1     1462          20       RL         81.0    14267   Pave   NaN      IR1
2     1463          60       RL         74.0    13830   Pave   NaN      IR1
3     1464          60       RL         78.0     9978   Pave   NaN      IR1
4     1465         120       RL         43.0     5005   Pave   NaN      IR1
...    ...         ...      ...          ...      ...    ...   ...      ...
1454  2915         160       RM         21.0     1936   Pave   NaN      Reg
1455  2916         160       RM         21.0     1894   Pave   NaN      Reg
1456  2917          20       RL        160.0    20000   Pave   NaN      Reg
1457  2918          85       RL         62.0    10441   Pave   NaN      Reg
1458  2919          60       RL         74.0     9627   Pave   NaN      Reg

     LandContour Utilities  ... ScreenPorch PoolArea PoolQC  Fence  \
0            Lvl    AllPub  ...         120        0    NaN  MnPrv
1            Lvl    AllPub  ...           0        0    NaN    NaN
2            Lvl    AllPub  ...           0        0    NaN  MnPrv
3            Lvl    AllPub  ...           0        0    NaN    NaN
4            HLS    AllPub  ...         144        0    NaN    NaN
...          ...       ...  ...         ...      ...    ...    ...
1454         Lvl    AllPub  ...           0        0    NaN    NaN
```

```
1455          Lvl    AllPub  ...            0         0    NaN    NaN
1456          Lvl    AllPub  ...            0         0    NaN    NaN
1457          Lvl    AllPub  ...            0         0    NaN  MnPrv
1458          Lvl    AllPub  ...            0         0    NaN    NaN

     MiscFeature MiscVal MoSold  YrSold  SaleType  SaleCondition
0            NaN       0      6    2010        WD         Normal
1           Gar2   12500      6    2010        WD         Normal
2            NaN       0      3    2010        WD         Normal
3            NaN       0      6    2010        WD         Normal
4            NaN       0      1    2010        WD         Normal
...          ...     ...    ...     ...       ...            ...
1454         NaN       0      6    2006        WD         Normal
1455         NaN       0      4    2006        WD        Abnorml
1456         NaN       0      9    2006        WD        Abnorml
1457        Shed     700      7    2006        WD         Normal
1458         NaN       0     11    2006        WD         Normal

[1459 rows x 80 columns]
```

Lets convert the values which are currently not in their correct format

[47]:
```
test_df['MSSubClass'] = str(test_df['MSSubClass'])
test_df['OverallCond'] = str(test_df['OverallCond'])
test_df['OverallQual'] = str(test_df['OverallQual'])
test_df['YrSold'] = str(test_df['YrSold'])
test_df['MoSold'] = str(test_df['MoSold'])
```

6.0.1 Missing Values

[48]:
```
test_df.isnull().sum()
```

[48]:
```
Id                 0
MSSubClass         0
MSZoning           4
LotFrontage      227
LotArea            0
                ...
MiscVal            0
MoSold             0
YrSold             0
SaleType           1
SaleCondition      0
Length: 80, dtype: int64
```

This is not that useful, lets only retrieve the columns with missing values and what percentage of values are missing.

[49]:
```
test_missing = test_df.isna().mean() * 100

test_missing_drop = test_missing.drop(test_missing[test_missing == 0].index).
 ↪sort_values()
print(test_missing_drop)
```

```
TotalBsmtSF      0.068540
GarageArea       0.068540
GarageCars       0.068540
KitchenQual      0.068540
BsmtUnfSF        0.068540
BsmtFinSF2       0.068540
BsmtFinSF1       0.068540
SaleType         0.068540
Exterior1st      0.068540
Exterior2nd      0.068540
Functional       0.137080
Utilities        0.137080
BsmtHalfBath     0.137080
BsmtFullBath     0.137080
MSZoning         0.274160
MasVnrArea       1.028101
MasVnrType       1.096642
BsmtFinType2     2.878684
BsmtFinType1     2.878684
BsmtQual         3.015764
BsmtExposure     3.015764
BsmtCond         3.084304
GarageType       5.209047
GarageFinish     5.346127
GarageQual       5.346127
GarageCond       5.346127
GarageYrBlt      5.346127
LotFrontage     15.558602
FireplaceQu     50.034270
Fence           80.123372
Alley           92.666210
MiscFeature     96.504455
PoolQC          99.794380
dtype: float64
```

As we want to pass the same features from the test set into our model which was fitted on the training set, lets see which columns in the test set we can remove straight away.

[50]:
```
train_df[train_df.columns[0:30]].columns
```

```
[50]: Index(['Id', 'LotArea', 'YearBuilt', 'YearRemodAdd', 'MasVnrArea',
       'BsmtFinSF1', 'BsmtFinSF2', 'BsmtUnfSF', 'TotalBsmtSF', '1stFlrSF',
       '2ndFlrSF', 'LowQualFinSF', 'GrLivArea', 'BsmtFullBath', 'BsmtHalfBath',
       'FullBath', 'HalfBath', 'BedroomAbvGr', 'KitchenAbvGr', 'TotRmsAbvGrd',
       'Fireplaces', 'GarageCars', 'GarageArea', 'WoodDeckSF', 'OpenPorchSF',
       'EnclosedPorch', '3SsnPorch', 'ScreenPorch', 'PoolArea', 'MiscVal'],
      dtype='object')
```

```
[51]: test_df.select_dtypes(include = np.number).columns
```

```
[51]: Index(['Id', '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', 'PoolArea', 'MiscVal'],
      dtype='object')
```

Ok, so we can drop 'Id', 'LotFrontage', 'GarageYrBlt'.

```
[52]: test_df = test_df.drop(columns = ['Id', 'LotFrontage', 'GarageYrBlt'])
```

```
[53]: test_missing = test_df.isna().mean() * 100

test_missing_drop = test_missing.drop(test_missing[test_missing == 0].index).
 ↪sort_values()
print(test_missing_drop)
```

```
TotalBsmtSF      0.068540
GarageArea       0.068540
GarageCars       0.068540
KitchenQual      0.068540
BsmtUnfSF        0.068540
BsmtFinSF2       0.068540
BsmtFinSF1       0.068540
SaleType         0.068540
Exterior1st      0.068540
Exterior2nd      0.068540
Functional       0.137080
Utilities        0.137080
BsmtHalfBath     0.137080
BsmtFullBath     0.137080
MSZoning         0.274160
MasVnrArea       1.028101
MasVnrType       1.096642
BsmtFinType2     2.878684
BsmtFinType1     2.878684
```

```
BsmtQual          3.015764
BsmtExposure      3.015764
BsmtCond          3.084304
GarageType        5.209047
GarageFinish      5.346127
GarageQual        5.346127
GarageCond        5.346127
FireplaceQu      50.034270
Fence            80.123372
Alley            92.666210
MiscFeature      96.504455
PoolQC           99.794380
dtype: float64
```

We can remove 'Fence', 'Alley', 'MiscFeature', 'PoolQC' as they have lots of missing values so do not contribute much to the model.

[54]:
```
test_df = test_df.drop(columns = ['Fence', 'Alley', 'MiscFeature', 'PoolQC',␣
 ↪'FireplaceQu'])
```

[55]:
```
test_missing = test_df.isna().mean() * 100

test_missing_drop = test_missing.drop(test_missing[test_missing == 0].index).
 ↪sort_values()
print(test_missing_drop)
```

```
BsmtFinSF2      0.068540
GarageArea      0.068540
GarageCars      0.068540
KitchenQual     0.068540
TotalBsmtSF     0.068540
BsmtUnfSF       0.068540
BsmtFinSF1      0.068540
SaleType        0.068540
Exterior1st     0.068540
Exterior2nd     0.068540
BsmtHalfBath    0.137080
Utilities       0.137080
Functional      0.137080
BsmtFullBath    0.137080
MSZoning        0.274160
MasVnrArea      1.028101
MasVnrType      1.096642
BsmtFinType2    2.878684
BsmtFinType1    2.878684
BsmtQual        3.015764
BsmtExposure    3.015764
BsmtCond        3.084304
```

```
GarageType      5.209047
GarageCond      5.346127
GarageFinish    5.346127
GarageQual      5.346127
dtype: float64
```

Now I am going to check which of these columns I have already removed from the training set so I can remoove these columns. Then I will impute the rest of the missing values with 0.

These columns can be removed as they are not in our training set 'BsmtQual', 'GarageCond', 'GarageFinish'.

[56]:
```
test_df = test_df.drop(columns = ['BsmtQual', 'GarageCond', 'GarageFinish'])
```

[57]:
```
test_missing = test_df.isna().mean() * 100

test_missing_drop = test_missing.drop(test_missing[test_missing == 0].index).
 ↪sort_values()
print(test_missing_drop)
```

```
BsmtFinSF2      0.068540
GarageArea      0.068540
GarageCars      0.068540
KitchenQual     0.068540
TotalBsmtSF     0.068540
BsmtUnfSF       0.068540
BsmtFinSF1      0.068540
SaleType        0.068540
Exterior2nd     0.068540
Exterior1st     0.068540
BsmtFullBath    0.137080
BsmtHalfBath    0.137080
Functional      0.137080
Utilities       0.137080
MSZoning        0.274160
MasVnrArea      1.028101
MasVnrType      1.096642
BsmtFinType1    2.878684
BsmtFinType2    2.878684
BsmtExposure    3.015764
BsmtCond        3.084304
GarageType      5.209047
GarageQual      5.346127
dtype: float64
```

[58]:
```
test_df = test_df.fillna(0)
```

[59]:
```
test_missing = test_df.isna().mean() * 100

test_missing_drop = test_missing.drop(test_missing[test_missing == 0].index).
 ↪sort_values()
print(test_missing_drop)
```

```
Series([], dtype: float64)
```

Now we no longer have any missing values.

[60]:
```
test_df
```

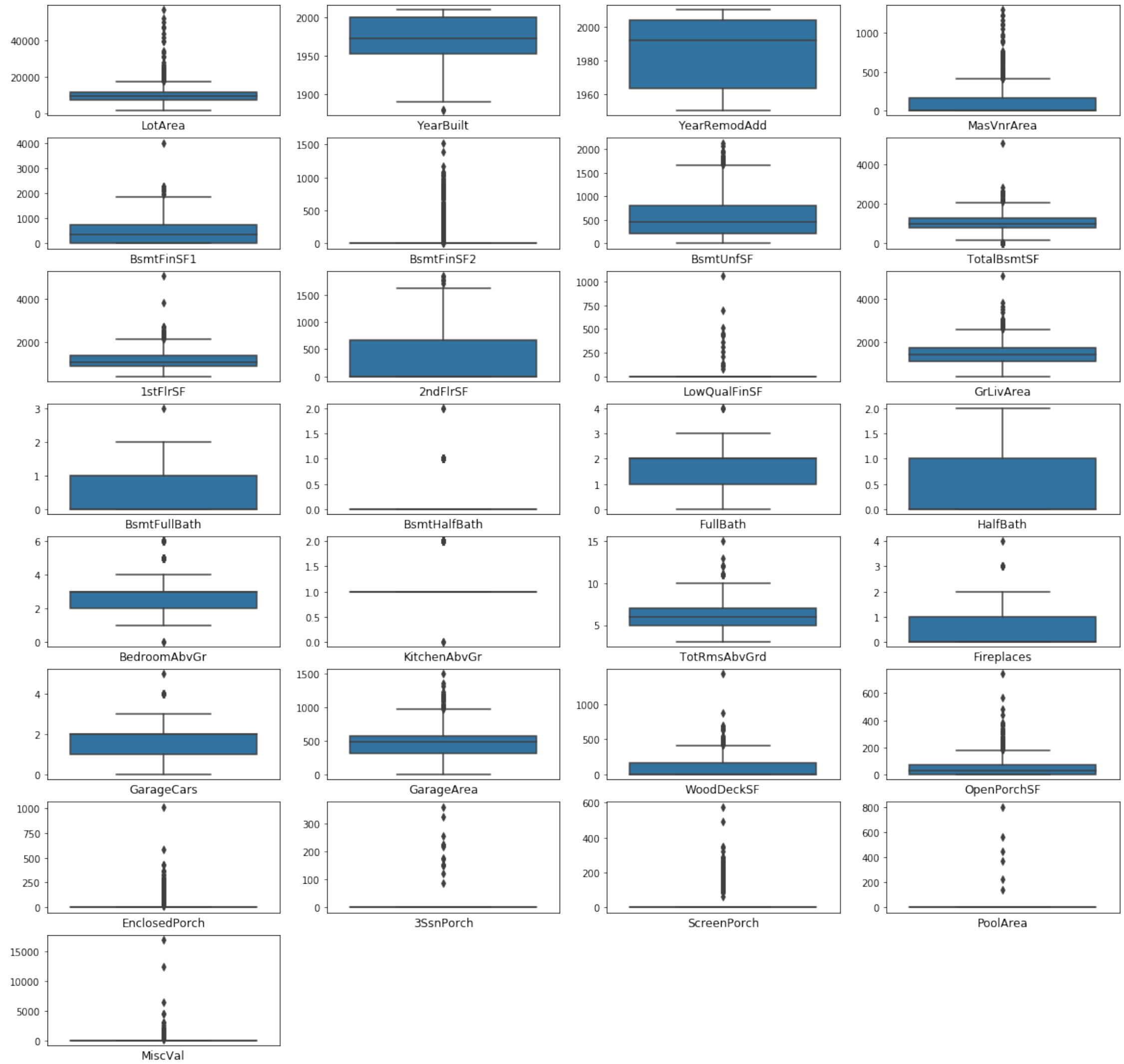
[60]:
```
                                                    MSSubClass MSZoning  LotArea  \
0     0         20\n1         20\n2         60\n3         ...       RH    11622
1     0         20\n1         20\n2         60\n3         ...       RL    14267
2     0         20\n1         20\n2         60\n3         ...       RL    13830
3     0         20\n1         20\n2         60\n3         ...       RL     9978
4     0         20\n1         20\n2         60\n3         ...       RL     5005
...                                                     ...      ...      ...
1454  0         20\n1         20\n2         60\n3         ...       RM     1936
1455  0         20\n1         20\n2         60\n3         ...       RM     1894
1456  0         20\n1         20\n2         60\n3         ...       RL    20000
1457  0         20\n1         20\n2         60\n3         ...       RL    10441
1458  0         20\n1         20\n2         60\n3         ...       RL     9627

     Street LotShape LandContour Utilities LotConfig LandSlope Neighborhood  \
0      Pave      Reg         Lvl    AllPub    Inside       Gtl        NAmes
1      Pave      IR1         Lvl    AllPub    Corner       Gtl        NAmes
2      Pave      IR1         Lvl    AllPub    Inside       Gtl      Gilbert
3      Pave      IR1         Lvl    AllPub    Inside       Gtl      Gilbert
4      Pave      IR1         HLS    AllPub    Inside       Gtl      StoneBr
...     ...      ...         ...       ...       ...       ...          ...
1454   Pave      Reg         Lvl    AllPub    Inside       Gtl      MeadowV
1455   Pave      Reg         Lvl    AllPub    Inside       Gtl      MeadowV
1456   Pave      Reg         Lvl    AllPub    Inside       Gtl      Mitchel
1457   Pave      Reg         Lvl    AllPub    Inside       Gtl      Mitchel
1458   Pave      Reg         Lvl    AllPub    Inside       Mod      Mitchel

      ... OpenPorchSF EnclosedPorch 3SsnPorch ScreenPorch PoolArea MiscVal  \
0     ...           0             0         0         120        0       0
1     ...          36             0         0           0        0   12500
2     ...          34             0         0           0        0       0
3     ...          36             0         0           0        0       0
4     ...          82             0         0         144        0       0
...   ...         ...           ...       ...         ...      ...     ...
1454  ...           0             0         0           0        0       0
1455  ...          24             0         0           0        0       0
1456  ...           0             0         0           0        0       0
```

```
1457  ...        32            0         0           0        0      700
1458  ...        48            0         0           0        0        0

                                                 MoSold  \
0     0       6\n1        6\n2        3\n3         6...
1     0       6\n1        6\n2        3\n3         6...
2     0       6\n1        6\n2        3\n3         6...
3     0       6\n1        6\n2        3\n3         6...
4     0       6\n1        6\n2        3\n3         6...
...                                                 ...
1454  0       6\n1        6\n2        3\n3         6...
1455  0       6\n1        6\n2        3\n3         6...
1456  0       6\n1        6\n2        3\n3         6...
1457  0       6\n1        6\n2        3\n3         6...
1458  0       6\n1        6\n2        3\n3         6...

                                                 YrSold SaleType SaleCondition
0     0      2010\n1      2010\n2      2010\n3     ...       WD        Normal
1     0      2010\n1      2010\n2      2010\n3     ...       WD        Normal
2     0      2010\n1      2010\n2      2010\n3     ...       WD        Normal
3     0      2010\n1      2010\n2      2010\n3     ...       WD        Normal
4     0      2010\n1      2010\n2      2010\n3     ...       WD        Normal
...                                                 ...      ...           ...
1454  0      2010\n1      2010\n2      2010\n3     ...       WD        Normal
1455  0      2010\n1      2010\n2      2010\n3     ...       WD       Abnorml
1456  0      2010\n1      2010\n2      2010\n3     ...       WD       Abnorml
1457  0      2010\n1      2010\n2      2010\n3     ...       WD        Normal
1458  0      2010\n1      2010\n2      2010\n3     ...       WD        Normal

[1459 rows x 69 columns]
```

6.0.2 Outliers

[61]:
```
plt.figure(figsize=(20,25))

numerical_cols = test_df.select_dtypes(exclude = 'object')

#This 'for' loop allows me to loop over numerical_cols using enumerate, whilst␣
↪keeping track of the index in i
for i, col in enumerate(numerical_cols.columns):
    # axes allows you to plot multiple axes and do things to the individual axes␣
↪(i.e. label them)
    axes = plt.subplot(10,4,i+1)
    sns.boxplot(data = test_df[col])
    axes.set_xlabel(col, fontsize = 12)
    axes.tick_params(axis='x', which='both', bottom=False, top=False,␣
↪labelbottom=False)
```

```
plt.tight_layout
plt.show;
```



Clearly, 'LotArea', 'MasVnrArea', 'GrLivArea', 'GarageArea', 'OpenPorchSF' have outliers which can be removed.

[62]:
```
def outlier_drop(data_frame, columns_outliers):
    """
    This function will drop all of the outliers in features which are specified

    Input: dataframe and features with outliers that need removing

    Output: updated dataframe with outliers removed
```

```
    """

    for i, col in enumerate(columns_outliers):
        q1_col, q3_col = np.percentile(data_frame[col], [25,75])
        iqr_col = q3_col - q1_col

        lowerbound_col = q1_col - (1.5 * iqr_col)
        upperbound_col = q3_col + (1.5 * iqr_col)

        outliers = (data_frame[(data_frame[col] < lowerbound_col)
                              | (data_frame[col] > upperbound_col)].index)

    data_frame = data_frame.drop(outliers)

    return data_frame
```

[63]:
```
outliers_columns = ['LotArea', 'MasVnrArea', 'GrLivArea', 'GarageArea',␣
 ↪'OpenPorchSF']

#Removing outliers from test dataframe
outlier_drop(test_df, outliers_columns)
```

[63]:
```
                                                    MSSubClass MSZoning  LotArea  \
0     0        20\n1         20\n2         60\n3        ...        RH    11622
1     0        20\n1         20\n2         60\n3        ...        RL    14267
2     0        20\n1         20\n2         60\n3        ...        RL    13830
3     0        20\n1         20\n2         60\n3        ...        RL     9978
4     0        20\n1         20\n2         60\n3        ...        RL     5005
...                                                     ...       ...      ...
1454  0        20\n1         20\n2         60\n3        ...        RM     1936
1455  0        20\n1         20\n2         60\n3        ...        RM     1894
1456  0        20\n1         20\n2         60\n3        ...        RL    20000
1457  0        20\n1         20\n2         60\n3        ...        RL    10441
1458  0        20\n1         20\n2         60\n3        ...        RL     9627

     Street LotShape LandContour Utilities LotConfig LandSlope Neighborhood  \
0      Pave      Reg         Lvl    AllPub    Inside       Gtl        NAmes
1      Pave      IR1         Lvl    AllPub    Corner       Gtl        NAmes
2      Pave      IR1         Lvl    AllPub    Inside       Gtl      Gilbert
3      Pave      IR1         Lvl    AllPub    Inside       Gtl      Gilbert
4      Pave      IR1         HLS    AllPub    Inside       Gtl      StoneBr
...     ...      ...         ...       ...       ...       ...          ...
1454   Pave      Reg         Lvl    AllPub    Inside       Gtl      MeadowV
1455   Pave      Reg         Lvl    AllPub    Inside       Gtl      MeadowV
1456   Pave      Reg         Lvl    AllPub    Inside       Gtl      Mitchel
1457   Pave      Reg         Lvl    AllPub    Inside       Gtl      Mitchel
1458   Pave      Reg         Lvl    AllPub    Inside       Mod      Mitchel
```

```
     ... OpenPorchSF EnclosedPorch 3SsnPorch ScreenPorch PoolArea MiscVal  \
0    ...           0             0         0         120        0       0
1    ...          36             0         0           0        0   12500
2    ...          34             0         0           0        0       0
3    ...          36             0         0           0        0       0
4    ...          82             0         0         144        0       0
...  ...         ...           ...       ...         ...      ...     ...
1454 ...           0             0         0           0        0       0
1455 ...          24             0         0           0        0       0
1456 ...           0             0         0           0        0       0
1457 ...          32             0         0           0        0     700
1458 ...          48             0         0           0        0       0

                                               MoSold  \
0     0        6\n1        6\n2        3\n3      6...
1     0        6\n1        6\n2        3\n3      6...
2     0        6\n1        6\n2        3\n3      6...
3     0        6\n1        6\n2        3\n3      6...
4     0        6\n1        6\n2        3\n3      6...
...                                               ...
1454  0        6\n1        6\n2        3\n3      6...
1455  0        6\n1        6\n2        3\n3      6...
1456  0        6\n1        6\n2        3\n3      6...
1457  0        6\n1        6\n2        3\n3      6...
1458  0        6\n1        6\n2        3\n3      6...

                                               YrSold SaleType SaleCondition
0     0      2010\n1      2010\n2      2010\n3    ...       WD        Normal
1     0      2010\n1      2010\n2      2010\n3    ...       WD        Normal
2     0      2010\n1      2010\n2      2010\n3    ...       WD        Normal
3     0      2010\n1      2010\n2      2010\n3    ...       WD        Normal
4     0      2010\n1      2010\n2      2010\n3    ...       WD        Normal
...                                               ...      ...           ...
1454  0      2010\n1      2010\n2      2010\n3    ...       WD        Normal
1455  0      2010\n1      2010\n2      2010\n3    ...       WD       Abnorml
1456  0      2010\n1      2010\n2      2010\n3    ...       WD       Abnorml
1457  0      2010\n1      2010\n2      2010\n3    ...       WD        Normal
1458  0      2010\n1      2010\n2      2010\n3    ...       WD        Normal

[1380 rows x 69 columns]
```

6.0.3 Checking skewness

[64]:
```
#Choosing numerical columns
test_numerical_columns = test_df.select_dtypes(include = np.number).columns



for i, col in enumerate(test_numerical_columns):
    skewness = test_df[col].skew()
    kurt = test_df[col].kurtosis()
    print('Skewness of', col, 'is', skewness)
    print('Kurtosis of', col, 'is', kurt, '\n')
```

```
Skewness of LotArea is 3.115216613500925
Kurtosis of LotArea is 20.746548709480273

Skewness of YearBuilt is -0.5876566078696325
Kurtosis of YearBuilt is -0.579320618279759

Skewness of YearRemodAdd is -0.39990598881237394
Kurtosis of YearRemodAdd is -1.4125856813078284

Skewness of MasVnrArea is 2.549568580084032
Kurtosis of MasVnrArea is 8.479206132512642

Skewness of BsmtFinSF1 is 1.1663296764605604
Kurtosis of BsmtFinSF1 is 2.673291224416369

Skewness of BsmtFinSF2 is 4.0429539042374705
Kurtosis of BsmtFinSF2 is 17.68216539285966

Skewness of BsmtUnfSF is 0.9199232287642486
Kurtosis of BsmtUnfSF is 0.33244656805494754

Skewness of TotalBsmtSF is 0.8050662652894462
Kurtosis of TotalBsmtSF is 5.173387914148094

Skewness of 1stFlrSF is 1.558194572983426
Kurtosis of 1stFlrSF is 8.053863335994567

Skewness of 2ndFlrSF is 0.9128826344911904
Kurtosis of 2ndFlrSF is -0.27544098409544926

Skewness of LowQualFinSF is 16.167254030990343
Kurtosis of LowQualFinSF is 308.67690650153196

Skewness of GrLivArea is 1.1304024140503506
Kurtosis of GrLivArea is 2.9203451470914166
```

```
Skewness of BsmtFullBath is 0.6518652012126019
Kurtosis of BsmtFullBath is -0.6425404276091546

Skewness of BsmtHalfBath is 3.782975779434155
Kurtosis of BsmtHalfBath is 13.575839867420619

Skewness of FullBath is 0.2958386394763122
Kurtosis of FullBath is -0.232339259823799

Skewness of HalfBath is 0.7147275486062835
Kurtosis of HalfBath is -0.9887003982031177

Skewness of BedroomAbvGr is 0.43662327937443507
Kurtosis of BedroomAbvGr is 1.6859653302114266

Skewness of KitchenAbvGr is 4.079055022333038
Kurtosis of KitchenAbvGr is 17.471691151777893

Skewness of TotRmsAbvGrd is 0.8425974464281701
Kurtosis of TotRmsAbvGrd is 1.5225956466723316

Skewness of Fireplaces is 0.8198582704555165
Kurtosis of Fireplaces is 0.3873261355352815

Skewness of GarageCars is -0.10999345834943444
Kurtosis of GarageCars is 0.24648590262300463

Skewness of GarageArea is 0.29629030596790457
Kurtosis of GarageArea is 0.9610046428994408

Skewness of WoodDeckSF is 2.130759950574929
Kurtosis of WoodDeckSF is 10.249278055421557

Skewness of OpenPorchSF is 2.6877788503224047
Kurtosis of OpenPorchSF is 13.010835704951141

Skewness of EnclosedPorch is 4.669172309715744
Kurtosis of EnclosedPorch is 40.129017354477156

Skewness of 3SsnPorch is 12.524215926005567
Kurtosis of 3SsnPorch is 170.20011269208473

Skewness of ScreenPorch is 3.7882443811743776
Kurtosis of ScreenPorch is 17.239542060670402

Skewness of PoolArea is 20.196887591116827
Kurtosis of PoolArea is 445.66110156800477
```

```
Skewness of MiscVal is 20.075188353860344
Kurtosis of MiscVal is 471.51738779462795
```

[65]:
```
# Selecting features with high skew
test_high_skew = high_skew(test_df, test_numerical_columns)
```

[66]:
```
# Performing boxcox transformation for features with high skew
high_skewed_features_test_df_box = scipy.special.
↪boxcox1p(test_df[test_high_skew], 0.5)
```

[67]:
```
for i, col in enumerate(test_high_skew):
    skewness = high_skewed_features_test_df_box[col].skew()
    print('Skewness of', col, 'is', skewness)
```

```
Skewness of LotArea is 0.7363390828698114
Skewness of MasVnrArea is 1.149356822074124
Skewness of BsmtFinSF1 is 0.04578881428414585
Skewness of BsmtFinSF2 is 2.980868196365515
Skewness of 1stFlrSF is 0.6486423873019207
Skewness of LowQualFinSF is 12.029002531472997
Skewness of GrLivArea is 0.538179736134212
Skewness of BsmtHalfBath is 3.6894768259301016
Skewness of KitchenAbvGr is 3.757919518725453
Skewness of WoodDeckSF is 0.5858574655441303
Skewness of OpenPorchSF is 0.747153597002874
Skewness of EnclosedPorch is 2.3182627997359506
Skewness of 3SsnPorch is 11.085387021724689
Skewness of ScreenPorch is 2.9726015541097697
Skewness of PoolArea is 17.173516060992487
Skewness of MiscVal is 9.611681553124736
```

Clearly skewness has been reduced.

[68]:
```
# Replacing features with high skew with transformed values
test_df[test_high_skew] = high_skewed_features_test_df_box
```

6.0.4 Categorical Variables

Lets encode all of the categorical variables.

[69]:
```
test_df = pd.get_dummies(test_df)
test_df
```

[69]:

	LotArea	YearBuilt	YearRemodAdd	MasVnrArea	BsmtFinSF1	BsmtFinSF2	\
0	213.620036	1961	1961	0.000000	41.312816	22.083189	
1	236.897468	1958	1958	18.880613	58.794737	0.000000	
2	233.210544	1997	1998	0.000000	54.284989	0.000000	

```
3     197.789890       1998           1998    7.165151   47.112117    0.000000
4     139.506184       1992           1992    0.000000   30.496154    0.000000
...          ...        ...            ...         ...         ...         ...
1454   86.022724       1970           1970    0.000000    0.000000    0.000000
1455   85.063195       1970           1970    0.000000   29.811947    0.000000
1456  280.849783       1960           1996    0.000000   68.000000    0.000000
1457  202.372209       1992           1992    0.000000   34.769553    0.000000
1458  194.244745       1993           1994   17.493589   53.099909    0.000000

      BsmtUnfSF  TotalBsmtSF   1stFlrSF  2ndFlrSF  ...  SaleType_ConLw  \
0         270.0        882.0  57.899917         0  ...               0
1         406.0       1329.0  70.938330         0  ...               0
2         137.0        928.0  58.959003       701  ...               0
3         324.0        926.0  58.893349       678  ...               0
4        1017.0       1280.0  69.582121         0  ...               0
...         ...          ...        ...       ...  ...             ...
1454      546.0        546.0  44.776062       546  ...               0
1455      294.0        546.0  44.776062       546  ...               0
1456        0.0       1224.0  68.000000         0  ...               0
1457      575.0        912.0  60.321746         0  ...               0
1458      238.0        996.0  61.150614      1004  ...               0

      SaleType_New  SaleType_Oth  SaleType_WD  SaleCondition_Abnorml  \
0                0             0            1                      0
1                0             0            1                      0
2                0             0            1                      0
3                0             0            1                      0
4                0             0            1                      0
...            ...           ...          ...                    ...
1454             0             0            1                      0
1455             0             0            1                      1
1456             0             0            1                      1
1457             0             0            1                      0
1458             0             0            1                      0

      SaleCondition_AdjLand  SaleCondition_Alloca  SaleCondition_Family  \
0                         0                     0                     0
1                         0                     0                     0
2                         0                     0                     0
3                         0                     0                     0
4                         0                     0                     0
...                     ...                   ...                   ...
1454                      0                     0                     0
1455                      0                     0                     0
1456                      0                     0                     0
1457                      0                     0                     0
1458                      0                     0                     0
```

```
      SaleCondition_Normal  SaleCondition_Partial
0                        1                      0
1                        1                      0
2                        1                      0
3                        1                      0
4                        1                      0
...                    ...                    ...
1454                     1                      0
1455                     0                      0
1456                     0                      0
1457                     1                      0
1458                     1                      0

[1459 rows x 254 columns]
```

6.0.5 Comparing Test and Train datasets

Let me double check the test and train sets are similar so the test set can pass through the model.

[70]:
```python
print(train_df.shape)
print(test_df.shape)
```

```
(1362, 255)
(1459, 254)
```

I am going to examine the column names to see if there are any columns which are in the test set but not in the training set and I will remove these columns

[71]:
```python
list(test_df.columns)
```

[71]:
```
['LotArea',
 'YearBuilt',
 'YearRemodAdd',
 'MasVnrArea',
 'BsmtFinSF1',
 'BsmtFinSF2',
 'BsmtUnfSF',
 'TotalBsmtSF',
 '1stFlrSF',
 '2ndFlrSF',
 'LowQualFinSF',
 'GrLivArea',
 'BsmtFullBath',
 'BsmtHalfBath',
 'FullBath',
 'HalfBath',
 'BedroomAbvGr',
```

```
 'KitchenAbvGr',
 'TotRmsAbvGrd',
 'Fireplaces',
 'GarageCars',
 'GarageArea',
 'WoodDeckSF',
 'OpenPorchSF',
 'EnclosedPorch',
 '3SsnPorch',
 'ScreenPorch',
 'PoolArea',
 'MiscVal',
 'MSSubClass_0        20\n1        20\n2        60\n3        60\n4       120\n
... \n1454    160\n1455    160\n1456     20\n1457     85\n1458     60\nName:
MSSubClass, Length: 1459, dtype: int64',
 'MSZoning_0',
 'MSZoning_C (all)',
 'MSZoning_FV',
 'MSZoning_RH',
 'MSZoning_RL',
 'MSZoning_RM',
 'Street_Grvl',
 'Street_Pave',
 'LotShape_IR1',
 'LotShape_IR2',
 'LotShape_IR3',
 'LotShape_Reg',
 'LandContour_Bnk',
 'LandContour_HLS',
 'LandContour_Low',
 'LandContour_Lvl',
 'Utilities_0',
 'Utilities_AllPub',
 'LotConfig_Corner',
 'LotConfig_CulDSac',
 'LotConfig_FR2',
 'LotConfig_FR3',
 'LotConfig_Inside',
 'LandSlope_Gtl',
 'LandSlope_Mod',
 'LandSlope_Sev',
 'Neighborhood_Blmngtn',
 'Neighborhood_Blueste',
 'Neighborhood_BrDale',
 'Neighborhood_BrkSide',
 'Neighborhood_ClearCr',
 'Neighborhood_CollgCr',
```

```
 'Neighborhood_Crawfor',
 'Neighborhood_Edwards',
 'Neighborhood_Gilbert',
 'Neighborhood_IDOTRR',
 'Neighborhood_MeadowV',
 'Neighborhood_Mitchel',
 'Neighborhood_NAmes',
 'Neighborhood_NPkVill',
 'Neighborhood_NWAmes',
 'Neighborhood_NoRidge',
 'Neighborhood_NridgHt',
 'Neighborhood_OldTown',
 'Neighborhood_SWISU',
 'Neighborhood_Sawyer',
 'Neighborhood_SawyerW',
 'Neighborhood_Somerst',
 'Neighborhood_StoneBr',
 'Neighborhood_Timber',
 'Neighborhood_Veenker',
 'Condition1_Artery',
 'Condition1_Feedr',
 'Condition1_Norm',
 'Condition1_PosA',
 'Condition1_PosN',
 'Condition1_RRAe',
 'Condition1_RRAn',
 'Condition1_RRNe',
 'Condition1_RRNn',
 'Condition2_Artery',
 'Condition2_Feedr',
 'Condition2_Norm',
 'Condition2_PosA',
 'Condition2_PosN',
 'BldgType_1Fam',
 'BldgType_2fmCon',
 'BldgType_Duplex',
 'BldgType_Twnhs',
 'BldgType_TwnhsE',
 'HouseStyle_1.5Fin',
 'HouseStyle_1.5Unf',
 'HouseStyle_1Story',
 'HouseStyle_2.5Unf',
 'HouseStyle_2Story',
 'HouseStyle_SFoyer',
 'HouseStyle_SLvl',
 'OverallQual_0       5\n1       6\n2       5\n3       6\n4       8\n
..\n1454    4\n1455    4\n1456    5\n1457    5\n1458    7\nName: OverallQual,
```

```
Length: 1459, dtype: int64',
 'OverallCond_0       6\n1       6\n2       5\n3       6\n4       5\n
..\n1454    7\n1455    5\n1456    7\n1457    5\n1458    5\nName: OverallCond,
Length: 1459, dtype: int64',
 'RoofStyle_Flat',
 'RoofStyle_Gable',
 'RoofStyle_Gambrel',
 'RoofStyle_Hip',
 'RoofStyle_Mansard',
 'RoofStyle_Shed',
 'RoofMatl_CompShg',
 'RoofMatl_Tar&Grv',
 'RoofMatl_WdShake',
 'RoofMatl_WdShngl',
 'Exterior1st_0',
 'Exterior1st_AsbShng',
 'Exterior1st_AsphShn',
 'Exterior1st_BrkComm',
 'Exterior1st_BrkFace',
 'Exterior1st_CBlock',
 'Exterior1st_CemntBd',
 'Exterior1st_HdBoard',
 'Exterior1st_MetalSd',
 'Exterior1st_Plywood',
 'Exterior1st_Stucco',
 'Exterior1st_VinylSd',
 'Exterior1st_Wd Sdng',
 'Exterior1st_WdShing',
 'Exterior2nd_0',
 'Exterior2nd_AsbShng',
 'Exterior2nd_AsphShn',
 'Exterior2nd_Brk Cmn',
 'Exterior2nd_BrkFace',
 'Exterior2nd_CBlock',
 'Exterior2nd_CmentBd',
 'Exterior2nd_HdBoard',
 'Exterior2nd_ImStucc',
 'Exterior2nd_MetalSd',
 'Exterior2nd_Plywood',
 'Exterior2nd_Stone',
 'Exterior2nd_Stucco',
 'Exterior2nd_VinylSd',
 'Exterior2nd_Wd Sdng',
 'Exterior2nd_Wd Shng',
 'MasVnrType_0',
 'MasVnrType_BrkCmn',
 'MasVnrType_BrkFace',
```

```
'MasVnrType_None',
'MasVnrType_Stone',
'ExterQual_Ex',
'ExterQual_Fa',
'ExterQual_Gd',
'ExterQual_TA',
'ExterCond_Ex',
'ExterCond_Fa',
'ExterCond_Gd',
'ExterCond_Po',
'ExterCond_TA',
'Foundation_BrkTil',
'Foundation_CBlock',
'Foundation_PConc',
'Foundation_Slab',
'Foundation_Stone',
'Foundation_Wood',
'BsmtCond_0',
'BsmtCond_Fa',
'BsmtCond_Gd',
'BsmtCond_Po',
'BsmtCond_TA',
'BsmtExposure_0',
'BsmtExposure_Av',
'BsmtExposure_Gd',
'BsmtExposure_Mn',
'BsmtExposure_No',
'BsmtFinType1_0',
'BsmtFinType1_ALQ',
'BsmtFinType1_BLQ',
'BsmtFinType1_GLQ',
'BsmtFinType1_LwQ',
'BsmtFinType1_Rec',
'BsmtFinType1_Unf',
'BsmtFinType2_0',
'BsmtFinType2_ALQ',
'BsmtFinType2_BLQ',
'BsmtFinType2_GLQ',
'BsmtFinType2_LwQ',
'BsmtFinType2_Rec',
'BsmtFinType2_Unf',
'Heating_GasA',
'Heating_GasW',
'Heating_Grav',
'Heating_Wall',
'HeatingQC_Ex',
'HeatingQC_Fa',
```

```
 'HeatingQC_Gd',
 'HeatingQC_Po',
 'HeatingQC_TA',
 'CentralAir_N',
 'CentralAir_Y',
 'Electrical_FuseA',
 'Electrical_FuseF',
 'Electrical_FuseP',
 'Electrical_SBrkr',
 'KitchenQual_0',
 'KitchenQual_Ex',
 'KitchenQual_Fa',
 'KitchenQual_Gd',
 'KitchenQual_TA',
 'Functional_0',
 'Functional_Maj1',
 'Functional_Maj2',
 'Functional_Min1',
 'Functional_Min2',
 'Functional_Mod',
 'Functional_Sev',
 'Functional_Typ',
 'GarageType_0',
 'GarageType_2Types',
 'GarageType_Attchd',
 'GarageType_Basment',
 'GarageType_BuiltIn',
 'GarageType_CarPort',
 'GarageType_Detchd',
 'GarageQual_0',
 'GarageQual_Fa',
 'GarageQual_Gd',
 'GarageQual_Po',
 'GarageQual_TA',
 'PavedDrive_N',
 'PavedDrive_P',
 'PavedDrive_Y',
 'MoSold_0        6\n1        6\n2        3\n3        6\n4        1\n
..\n1454     6\n1455     4\n1456     9\n1457     7\n1458    11\nName: MoSold,
Length: 1459, dtype: int64',
 'YrSold_0       2010\n1       2010\n2       2010\n3       2010\n4       2010\n
... \n1454    2006\n1455    2006\n1456    2006\n1457    2006\n1458
2006\nName: YrSold, Length: 1459, dtype: int64',
 'SaleType_0',
 'SaleType_COD',
 'SaleType_CWD',
 'SaleType_Con',
```

```
 'SaleType_ConLD',
 'SaleType_ConLI',
 'SaleType_ConLw',
 'SaleType_New',
 'SaleType_Oth',
 'SaleType_WD',
 'SaleCondition_Abnorml',
 'SaleCondition_AdjLand',
 'SaleCondition_Alloca',
 'SaleCondition_Family',
 'SaleCondition_Normal',
 'SaleCondition_Partial']
```

[72]:
```
list(train_df.columns)
```

[72]:
```
['Id',
 'LotArea',
 'YearBuilt',
 'YearRemodAdd',
 'MasVnrArea',
 'BsmtFinSF1',
 'BsmtFinSF2',
 'BsmtUnfSF',
 'TotalBsmtSF',
 '1stFlrSF',
 '2ndFlrSF',
 'LowQualFinSF',
 'GrLivArea',
 'BsmtFullBath',
 'BsmtHalfBath',
 'FullBath',
 'HalfBath',
 'BedroomAbvGr',
 'KitchenAbvGr',
 'TotRmsAbvGrd',
 'Fireplaces',
 'GarageCars',
 'GarageArea',
 'WoodDeckSF',
 'OpenPorchSF',
 'EnclosedPorch',
 '3SsnPorch',
 'ScreenPorch',
 'PoolArea',
 'MiscVal',
 'SalePrice',
 'MSSubClass_0       60\n1        20\n2        60\n3        70\n4        60\n
```

```
..\n1455    60\n1456    20\n1457    70\n1458    20\n1459    20\nName:
MSSubClass, Length: 1362, dtype: int64',
 'MSZoning_C (all)',
 'MSZoning_FV',
 'MSZoning_RH',
 'MSZoning_RL',
 'MSZoning_RM',
 'Street_Grvl',
 'Street_Pave',
 'LotShape_IR1',
 'LotShape_IR2',
 'LotShape_IR3',
 'LotShape_Reg',
 'LandContour_Bnk',
 'LandContour_HLS',
 'LandContour_Low',
 'LandContour_Lvl',
 'Utilities_AllPub',
 'Utilities_NoSeWa',
 'LotConfig_Corner',
 'LotConfig_CulDSac',
 'LotConfig_FR2',
 'LotConfig_FR3',
 'LotConfig_Inside',
 'LandSlope_Gtl',
 'LandSlope_Mod',
 'LandSlope_Sev',
 'Neighborhood_Blmngtn',
 'Neighborhood_Blueste',
 'Neighborhood_BrDale',
 'Neighborhood_BrkSide',
 'Neighborhood_ClearCr',
 'Neighborhood_CollgCr',
 'Neighborhood_Crawfor',
 'Neighborhood_Edwards',
 'Neighborhood_Gilbert',
 'Neighborhood_IDOTRR',
 'Neighborhood_MeadowV',
 'Neighborhood_Mitchel',
 'Neighborhood_NAmes',
 'Neighborhood_NPkVill',
 'Neighborhood_NWAmes',
 'Neighborhood_NoRidge',
 'Neighborhood_NridgHt',
 'Neighborhood_OldTown',
 'Neighborhood_SWISU',
 'Neighborhood_Sawyer',
```

```
 'Neighborhood_SawyerW',
 'Neighborhood_Somerst',
 'Neighborhood_StoneBr',
 'Neighborhood_Timber',
 'Neighborhood_Veenker',
 'Condition1_Artery',
 'Condition1_Feedr',
 'Condition1_Norm',
 'Condition1_PosA',
 'Condition1_PosN',
 'Condition1_RRAe',
 'Condition1_RRAn',
 'Condition1_RRNe',
 'Condition1_RRNn',
 'Condition2_Artery',
 'Condition2_Feedr',
 'Condition2_Norm',
 'Condition2_PosN',
 'Condition2_RRAe',
 'Condition2_RRAn',
 'Condition2_RRNn',
 'BldgType_1Fam',
 'BldgType_2fmCon',
 'BldgType_Duplex',
 'BldgType_Twnhs',
 'BldgType_TwnhsE',
 'HouseStyle_1.5Fin',
 'HouseStyle_1.5Unf',
 'HouseStyle_1Story',
 'HouseStyle_2.5Fin',
 'HouseStyle_2.5Unf',
 'HouseStyle_2Story',
 'HouseStyle_SFoyer',
 'HouseStyle_SLvl',
 'OverallQual_0       7\n1       6\n2       7\n3       7\n4       8\n
..\n1455    6\n1456    6\n1457    7\n1458    5\n1459    5\nName: OverallQual,
Length: 1362, dtype: int64',
 'OverallCond_0       5\n1       8\n2       5\n3       5\n4       5\n
..\n1455    5\n1456    6\n1457    9\n1458    6\n1459    6\nName: OverallCond,
Length: 1362, dtype: int64',
 'RoofStyle_Flat',
 'RoofStyle_Gable',
 'RoofStyle_Gambrel',
 'RoofStyle_Hip',
 'RoofStyle_Mansard',
 'RoofStyle_Shed',
 'RoofMatl_CompShg',
```

```
'RoofMatl_Membran',
'RoofMatl_Metal',
'RoofMatl_Roll',
'RoofMatl_Tar&Grv',
'RoofMatl_WdShake',
'RoofMatl_WdShngl',
'Exterior1st_AsbShng',
'Exterior1st_BrkFace',
'Exterior1st_CBlock',
'Exterior1st_CemntBd',
'Exterior1st_HdBoard',
'Exterior1st_ImStucc',
'Exterior1st_MetalSd',
'Exterior1st_Plywood',
'Exterior1st_Stone',
'Exterior1st_Stucco',
'Exterior1st_VinylSd',
'Exterior1st_Wd Sdng',
'Exterior1st_WdShing',
'Exterior2nd_AsbShng',
'Exterior2nd_AsphShn',
'Exterior2nd_Brk Cmn',
'Exterior2nd_BrkFace',
'Exterior2nd_CBlock',
'Exterior2nd_CmentBd',
'Exterior2nd_HdBoard',
'Exterior2nd_ImStucc',
'Exterior2nd_MetalSd',
'Exterior2nd_Other',
'Exterior2nd_Plywood',
'Exterior2nd_Stone',
'Exterior2nd_Stucco',
'Exterior2nd_VinylSd',
'Exterior2nd_Wd Sdng',
'Exterior2nd_Wd Shng',
'MasVnrType_BrkCmn',
'MasVnrType_BrkFace',
'MasVnrType_None',
'MasVnrType_Stone',
'ExterQual_Ex',
'ExterQual_Fa',
'ExterQual_Gd',
'ExterQual_TA',
'ExterCond_Ex',
'ExterCond_Fa',
'ExterCond_Gd',
'ExterCond_Po',
```

```
'ExterCond_TA',
'Foundation_BrkTil',
'Foundation_CBlock',
'Foundation_PConc',
'Foundation_Stone',
'Foundation_Wood',
'BsmtQual_Ex',
'BsmtQual_Fa',
'BsmtQual_Gd',
'BsmtQual_TA',
'BsmtCond_Fa',
'BsmtCond_Gd',
'BsmtCond_Po',
'BsmtCond_TA',
'BsmtExposure_Av',
'BsmtExposure_Gd',
'BsmtExposure_Mn',
'BsmtExposure_No',
'BsmtFinType1_ALQ',
'BsmtFinType1_BLQ',
'BsmtFinType1_GLQ',
'BsmtFinType1_LwQ',
'BsmtFinType1_Rec',
'BsmtFinType1_Unf',
'BsmtFinType2_ALQ',
'BsmtFinType2_BLQ',
'BsmtFinType2_GLQ',
'BsmtFinType2_LwQ',
'BsmtFinType2_Rec',
'BsmtFinType2_Unf',
'Heating_GasA',
'Heating_GasW',
'Heating_Grav',
'Heating_OthW',
'HeatingQC_Ex',
'HeatingQC_Fa',
'HeatingQC_Gd',
'HeatingQC_Po',
'HeatingQC_TA',
'CentralAir_N',
'CentralAir_Y',
'Electrical_FuseA',
'Electrical_FuseF',
'Electrical_FuseP',
'Electrical_Mix',
'Electrical_SBrkr',
'KitchenQual_Ex',
```

```
 'KitchenQual_Fa',
 'KitchenQual_Gd',
 'KitchenQual_TA',
 'Functional_Maj1',
 'Functional_Maj2',
 'Functional_Min1',
 'Functional_Min2',
 'Functional_Mod',
 'Functional_Sev',
 'Functional_Typ',
 'GarageType_2Types',
 'GarageType_Attchd',
 'GarageType_Basment',
 'GarageType_BuiltIn',
 'GarageType_CarPort',
 'GarageType_Detchd',
 'GarageQual_Ex',
 'GarageQual_Fa',
 'GarageQual_Gd',
 'GarageQual_Po',
 'GarageQual_TA',
 'PavedDrive_N',
 'PavedDrive_P',
 'PavedDrive_Y',
 'MoSold_0        2\n1        5\n2        9\n3        2\n4       12\n
..\n1455     8\n1456     2\n1457     5\n1458     4\n1459     6\nName: MoSold,
Length: 1362, dtype: int64',
 'YrSold_0       2008\n1       2007\n2       2008\n3       2006\n4       2008\n
... \n1455    2007\n1456    2010\n1457    2010\n1458    2010\n1459
2008\nName: YrSold, Length: 1362, dtype: int64',
 'SaleType_COD',
 'SaleType_CWD',
 'SaleType_Con',
 'SaleType_ConLD',
 'SaleType_ConLI',
 'SaleType_ConLw',
 'SaleType_New',
 'SaleType_Oth',
 'SaleType_WD',
 'SaleCondition_Abnorml',
 'SaleCondition_AdjLand',
 'SaleCondition_Alloca',
 'SaleCondition_Family',
 'SaleCondition_Normal',
 'SaleCondition_Partial']
```

'Utilities_0' is a feature which should be removed

[73]:
```
test_df = test_df.drop(columns = 'Utilities_0')
```

Now the test set is ready to pass through my models

7 Modelling

First lets split our data up into the features we want to learn from and the features we want to test on.

[74]:
```
#Dropping the column Id as it is just a label for the row. We can use inbuilt␣
↪Python index
train_df = train_df.drop(columns = 'Id')
```

[75]:
```
#These are the features which our model will use to make predictions about␣
↪SalePrice
X = train_df.drop(columns = 'SalePrice')
```

[76]:
```
#Our target variable
y = train_df['SalePrice']
```

I will now split X and y into train and test splits using StratifiedKFold which preserves the distribution of the data and uses all the data to train and test models.

[77]:
```
skf = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)
train_df_split = skf.get_n_splits(X,y)
```

7.0.1 Linear Regression

I will first fit a simple linear regression model and find the model accuracy using the error metric cross_val_score. For more information on how the LinearRegression model operates: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html

[78]:
```
lr = LinearRegression()

#Calculating cross_val_score for the 10 folds
train_lr_scores = (-1) * cross_val_score(lr, X, y, cv = train_df_split, scoring␣
↪= 'neg_mean_absolute_error')

print(train_lr_scores)
```

```
[0.07766194 0.08720395 0.08578021 0.08860303 0.09179059 0.07992134
 0.07895251 0.07925374 0.08040423 0.08601455]
```

[79]:
```
#Calculating the mean of these scores
train_lr_scores_mean = np.mean(train_lr_scores)
print(train_lr_scores_mean)
```

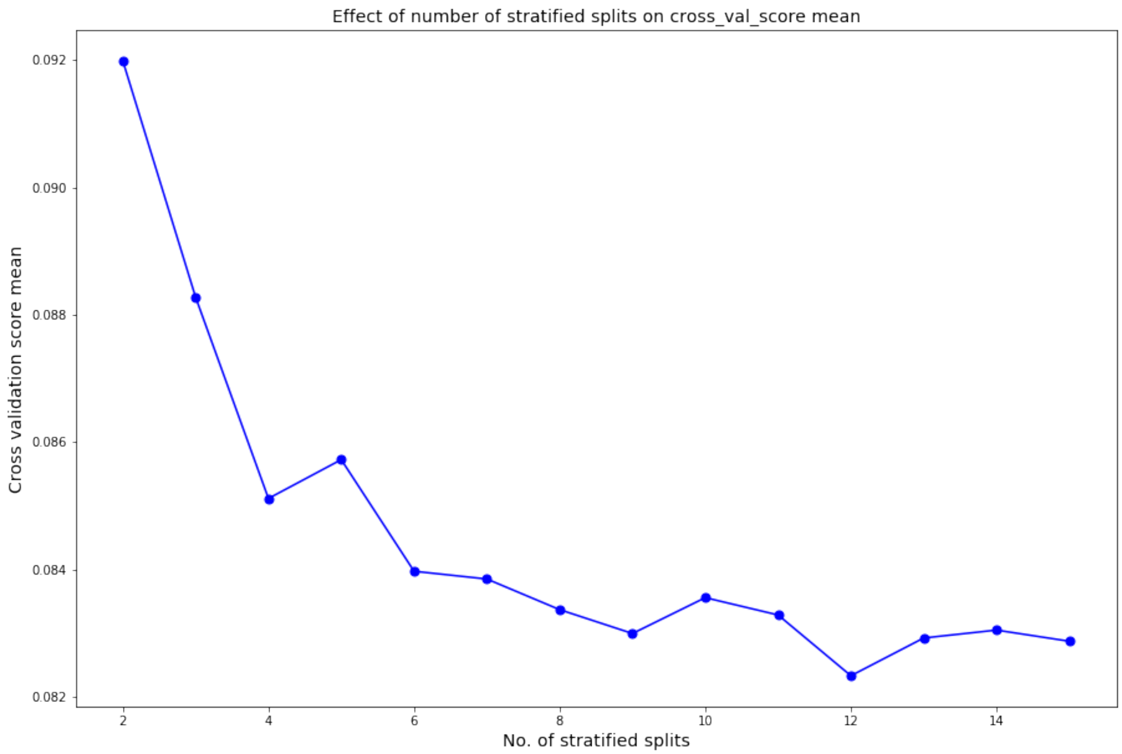
0.08355861009621796

An MAE of 0.084! This is quite good. However we just picked a random number of folds. Lets see how this value varies with the number of folds.

[80]:
```
def best_cross_val_score(model, data_X, data_y, splits):

    """
    This function will give the best cross validation score from a range of␣
 ↪stratified splits

    Input: machine learning model, input features, target features, no. of␣
 ↪desired splits

    Output: Best mean cross validation score, plot of mean cross validation␣
 ↪score with no. of splits

    """

    data_mean_array = []
    split_array = []

    for i, split in enumerate(range(2, splits+1)):

        skf_loop = StratifiedKFold(n_splits=split, shuffle=True, random_state=42)
        data_split = skf_loop.get_n_splits(data_X, data_y)
        data_scores = (-1) * cross_val_score(model, data_X, data_y, cv =␣
 ↪data_split, scoring = 'neg_mean_absolute_error')
        data_scores_mean = np.mean(data_scores)
        data_mean_array.append(data_scores_mean)
        split_array.append(split)

    print('The most accurate mean cross validation score is',␣
 ↪max(data_mean_array))

    plt.figure(figsize = (15,10))
    plt.plot(split_array, data_mean_array,'b', marker='.', markersize=14)
    plt.title("Effect of number of stratified splits on cross_val_score mean",␣
 ↪fontsize = 14)
    plt.xlabel("No. of stratified splits", fontsize = 14)
    plt.ylabel("Cross validation score mean", fontsize = 14)
    plt.show()
```

[81]:
```
#Finding the number of splits which results in the best cross validation score
best_cross_val_score(lr, X, y, 15)
```

The most accurate mean cross validation score is 0.09198970379208748



So we can see that as the number of folds the CVS mean increases. Therefore our accuracy increases. I used splits = 15 as any greater, it becomes computationally expensive to run the code whilst the accuracy does not improve by much. Therefore the 12 is the best number of stratified splits to use.

[82]:
```
# Storing the most accurate model
skf = StratifiedKFold(n_splits=12, shuffle=True, random_state=42)
train_df_split = skf.get_n_splits(X,y)
```

The number of splits will be useful for when I start to hypertune parameters to build more accurate models. Now I will split my training data so that I can fit in a simple linear regression model.

[83]:
```
# Splitting data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size = 0.2,␣
↪random_state = 42)
```

[84]:
```
# Fitting model to training split data
lr.fit(X_train,y_train)

# Predicting SalePrice using model on test split data
y_pred_lr = lr.predict(X_test)

# Calculating mean absolute error between predicted values and actual values
```

```python
print(mean_absolute_error(y_pred_lr, y_test))
```

```
0.08945059791046515
```

I achieve a low mean absolute error of 0.0895. This is impressive. However I think we can reduce this by finding a better model.

Now, let me use this model to predict on my test set

[85]:
```python
# Storing prediction on test set
lr_pred = lr.predict(test_df)
```

7.0.2 Ridge Regression

I will now use a ridge regression model which should hopefully manage some of the overfitting of a standard ordinary least squares model. For more information: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Ridge.html

[86]:
```python
rr = Ridge(alpha = 1.0, random_state=42)
```

[87]:
```python
rr.fit(X_train, y_train)

y_pred_rr = rr.predict(X_test)

print(mean_absolute_error(y_pred_rr, y_test))
```

```
0.0852248591197156
```

Now we have used the ridge regression model we have reduced the mean absolute error. Although the reduction is minimal.

However, we have only done this for 1 alpha value, let me do this for a range of alpha values. I will use the GridSearchCV function for this. GridSearchCV allows me to do an exhaustive search over a range of parameter values. For more information: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

[88]:
```python
# Some warnings were appearing so I will ignore them for now
from sklearn.model_selection import GridSearchCV
import warnings

warnings.filterwarnings('ignore')
```

As I am tuning hyperparameters I will use a cross validation splitting strategy using my best stratified k folds splitting strategy found previously.

[89]:
```python
# Defining range of parameters to use
parameters = {'alpha': [1e-15, 1e-10, 1e-8, 1e-4, 1e-3, 1e-2, 1, 5, 10, 20]}

ridge_regressor = GridSearchCV(rr, parameters, scoring =␣
 ↪'neg_mean_absolute_error', cv = train_df_split)
```

```
ridge_regressor.fit(X, y)
```

```
[89]: GridSearchCV(cv=12, error_score='raise-deprecating',
                   estimator=Ridge(alpha=1.0, copy_X=True, fit_intercept=True,
                                   max_iter=None, normalize=False, random_state=42,
                                   solver='auto', tol=0.001),
                   iid='warn', n_jobs=None,
                   param_grid={'alpha': [1e-15, 1e-10, 1e-08, 0.0001, 0.001, 0.01, 1,
                                         5, 10, 20]},
                   pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
                   scoring='neg_mean_absolute_error', verbose=0)
```

```
[90]: print('The best parameter for', ridge_regressor.best_params_)
      print('The best mean absolute error:', (-1) * ridge_regressor.best_score_)
```

```
The best parameter for {'alpha': 5}
The best mean absolute error: 0.07871660662991475
```

Clearly, now the mean absolute error has reduced by a much greater amount and we have found a better model. Therefore, I will use this model to make my predictions with the test data

```
[91]: rr_pred = ridge_regressor.predict(test_df)
```

7.0.3 Lasso Regression

Now I will use a lasso regression model to check whether anymore improvements can be made. For more information: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Lasso.html

```
[92]: lasso = Lasso(random_state = 42)

      lasso_regressor = GridSearchCV(lasso, parameters, scoring =␣
       ↪'neg_mean_absolute_error', cv = train_df_split)

      lasso_regressor.fit(X, y)
```

```
[92]: GridSearchCV(cv=12, error_score='raise-deprecating',
                   estimator=Lasso(alpha=1.0, copy_X=True, fit_intercept=True,
                                   max_iter=1000, normalize=False, positive=False,
                                   precompute=False, random_state=42,
                                   selection='cyclic', tol=0.0001, warm_start=False),
                   iid='warn', n_jobs=None,
                   param_grid={'alpha': [1e-15, 1e-10, 1e-08, 0.0001, 0.001, 0.01, 1,
                                         5, 10, 20]},
                   pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
                   scoring='neg_mean_absolute_error', verbose=0)
```

[93]:
```
print('The best parameter for', lasso_regressor.best_params_)
print('The best mean absolute error:', (-1) * lasso_regressor.best_score_)
```

```
The best parameter for {'alpha': 0.0001}
The best mean absolute error: 0.07947369397324025
```

The lasso regression model does not improve on the mean absolute error hence I will not make predictions using this model.

7.0.4 Elastic Net

Now I will use the Elastic Net model which uses both ridge regression and lasso regression to build a model which should help manage overfitting and underfitting. For documentation on this model: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.ElasticNet.html

[94]:
```
from sklearn.linear_model import ElasticNet
```

[95]:
```
parameters_grid = {'alpha': [1e-15, 1e-10, 1e-8, 1e-4, 1e-3, 1e-2, 1, 5, 10, 20],
                   'l1_ratio': [0, 0.1, 1]}
```

[96]:
```
elastic = ElasticNet(random_state = 42)

elastic_regressor = GridSearchCV(elastic, parameters_grid, scoring =␣
 ↪'neg_mean_absolute_error', cv = train_df_split)

elastic_regressor.fit(X, y)
```

[96]:
```
GridSearchCV(cv=12, error_score='raise-deprecating',
             estimator=ElasticNet(alpha=1.0, copy_X=True, fit_intercept=True,
                                  l1_ratio=0.5, max_iter=1000, normalize=False,
                                  positive=False, precompute=False,
                                  random_state=42, selection='cyclic',
                                  tol=0.0001, warm_start=False),
             iid='warn', n_jobs=None,
             param_grid={'alpha': [1e-15, 1e-10, 1e-08, 0.0001, 0.001, 0.01, 1,
                                   5, 10, 20],
                         'l1_ratio': [0, 0.1, 1]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring='neg_mean_absolute_error', verbose=0)
```

[97]:
```
print('The best parameter for', elastic_regressor.best_params_)
print('The best mean absolute error:', (-1) * elastic_regressor.best_score_)
```

```
The best parameter for {'alpha': 0.01, 'l1_ratio': 0}
The best mean absolute error: 0.07883602214880273
```

As we can see the mean absolute error does not improve. Also, this is a computationally expensive model to run for only a likely minimally improved model hence it will not be used.

7.0.5 Decision Tree Regressor

A decision tree aims to create a predictive model by learning decision rules which are inferred from the features of the dataset. For more information on the model: https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html and https://scikit-learn.org/stable/modules/tree.html#tree

[98]:
```
dt = DecisionTreeRegressor(criterion = 'mae', random_state = 42)

parameters_dt = {'max_depth': [3, 5],
                 'max_leaf_nodes': [3, 5]}

dt_regressor = GridSearchCV(dt, parameters_dt, cv = train_df_split, scoring =␣
 ↪'neg_mean_absolute_error')

dt_regressor.fit(X, y)
```

[98]:
```
GridSearchCV(cv=12, error_score='raise-deprecating',
             estimator=DecisionTreeRegressor(criterion='mae', max_depth=None,
                                             max_features=None,
                                             max_leaf_nodes=None,
                                             min_impurity_decrease=0.0,
                                             min_impurity_split=None,
                                             min_samples_leaf=1,
                                             min_samples_split=2,
                                             min_weight_fraction_leaf=0.0,
                                             presort=False, random_state=42,
                                             splitter='best'),
             iid='warn', n_jobs=None,
             param_grid={'max_depth': [3, 5], 'max_leaf_nodes': [3, 5]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring='neg_mean_absolute_error', verbose=0)
```

[99]:
```
print('The best parameter for', dt_regressor.best_params_)
print('The best mean absolute error:', (-1) * dt_regressor.best_score_)
```

```
The best parameter for {'max_depth': 3, 'max_leaf_nodes': 5}
The best mean absolute error: 0.17712525691570757
```

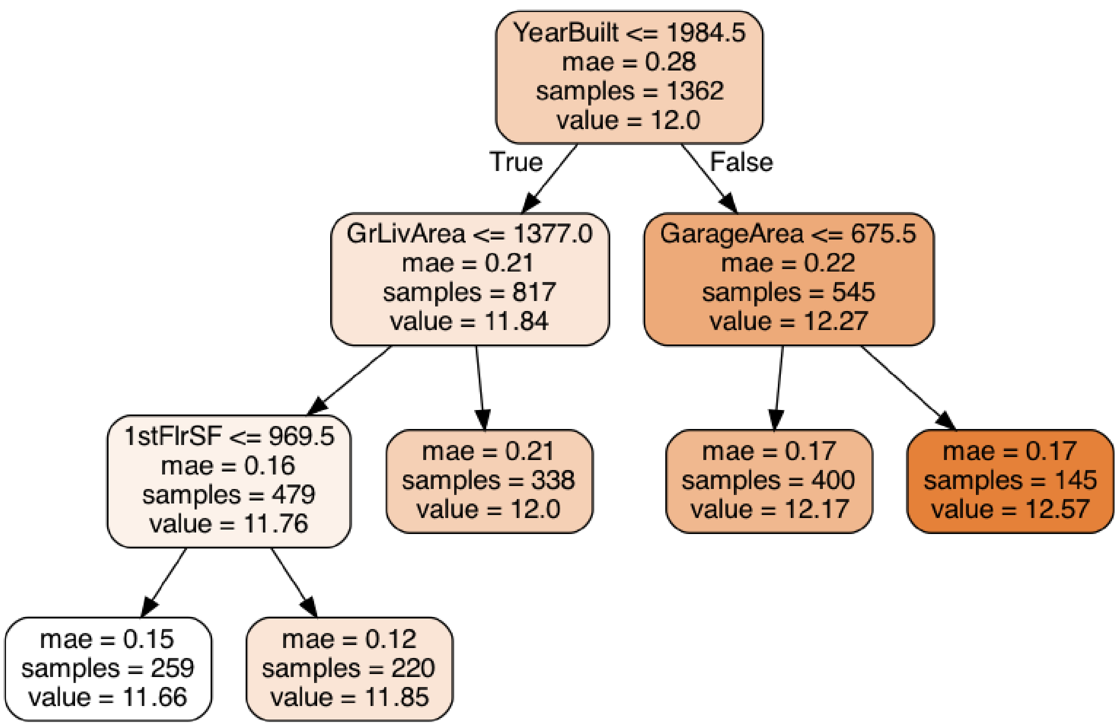
If we add more parameters to further hypertune this model, it is very computationally expensive and runs very slowly. Also, the mean absolute error is a lot greater than the ones acheived using ridge regression hence this is not an ideal model to use. Lets take a deeper look as to why this model has not worked as well.

In the plot below, I do not use the same splitting of the training data as I did in my calculation of the mean absolute error. This is because I am not conerned so much about the accuracy of the following model, but to demonstrate where this type of model fails when working with the training data I use.

[100]:

```
from sklearn import tree

dt_plot = DecisionTreeRegressor(criterion = 'mae', random_state = 42, max_depth␣
 ↪= 3, max_leaf_nodes = 5)

dt_plot = dt_plot.fit(X, y)


from sklearn.tree import export_graphviz

# Export as dot file
export_graphviz(dt_plot, out_file='tree.dot',
                feature_names = X.columns,
                rounded = True, proportion = False,
                precision = 2, filled = True)

import pydot

(graph,) = pydot.graph_from_dot_file(r'tree.dot')
graph.write_png(r'tree.png')

# Display in jupyter notebook
from IPython.display import Image
Image(filename = r'tree.png')
```

[100]:



As we can see from the decision tree plot above, this is not a great model for our dataset as we have lots of features which makes the decision tree computationally expensive to run. Additionally, if we do use a decision tree, I will only use a small number of features, making my model prone to underfitting. Therefore, this model leads to a high mean absolute error.

7.0.6 Random Forest Regressor

Now I will use a random forest tree, as this model uses many trees and makes predictions by averaging the predictions of each component tree. This will help use as many features as possible when training a model hence managing underfitting and overfitting. For more information: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html

Instead of using GridSearchCV, I use RandomizedSearchCV as I want to do a random search on the hyperparameters used to reduced computational cost. For more information: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.Randomiz

[101]:
```
from sklearn.model_selection import RandomizedSearchCV

rf = RandomForestRegressor(criterion = 'mae', random_state = 42)

parameters_rf = {'n_estimators': [10],
                 'max_depth': [3]}

rf_regressor = RandomizedSearchCV(rf, parameters_rf, cv = train_df_split,␣
 ↪random_state = 42, n_jobs = -1,
                                  scoring = 'neg_mean_absolute_error')
```

[102]:
```
rf_regressor.fit(X, y)
```

[102]:
```
RandomizedSearchCV(cv=12, error_score='raise-deprecating',
                   estimator=RandomForestRegressor(bootstrap=True,
                                                   criterion='mae',
                                                   max_depth=None,
                                                   max_features='auto',
                                                   max_leaf_nodes=None,
                                                   min_impurity_decrease=0.0,
                                                   min_impurity_split=None,
                                                   min_samples_leaf=1,
                                                   min_samples_split=2,
                                                   min_weight_fraction_leaf=0.0,
                                                   n_estimators='warn',
                                                   n_jobs=None, oob_score=False,
                                                   random_state=42, verbose=0,
                                                   warm_start=False),
                   iid='warn', n_iter=10, n_jobs=-1,
                   param_distributions={'max_depth': [3], 'n_estimators': [10]},
```

```
                       pre_dispatch='2*n_jobs', random_state=42, refit=True,
                       return_train_score=False, scoring='neg_mean_absolute_error',
                       verbose=0)
```

[103]:
```
print('The best parameter for', rf_regressor.best_params_)
print('The best mean absolute error:',(-1) * rf_regressor.best_score_)
```

```
The best parameter for {'n_estimators': 10, 'max_depth': 3}
The best mean absolute error: 0.13293277300624817
```

Again, we have a high mean absolute error with a high computational cost model hence this is not an ideal model to use

7.0.7 XGBoost Regressor

I will now use an agressive gradient boosting regressor to iterate over many random models to find the best model. For more information: https://xgboost.readthedocs.io/en/latest/python/python_api.html#module-xgboost.sklearn

[104]:
```
xgb = XGBRegressor(random_state = 42)

parameters_xgb = {'n_estimators': [10, 50, 100],
                  'learning_rate': [0.01, 0.025, 0.05]}

xgb_regressor = RandomizedSearchCV(xgb, parameters_xgb, cv = train_df_split,␣
 ↪random_state = 42,
                                  scoring = 'neg_mean_absolute_error')
```

[105]:
```
xgb_regressor.fit(X, y)
```

[105]:
```
RandomizedSearchCV(cv=12, error_score='raise-deprecating',
                   estimator=XGBRegressor(base_score=None, booster=None,
                                          colsample_bylevel=None,
                                          colsample_bynode=None,
                                          colsample_bytree=None, gamma=None,
                                          gpu_id=None, importance_type='gain',
                                          interaction_constraints=None,
                                          learning_rate=None,
                                          max_delta_step=None, max_depth=None,
                                          min_child_weight=None, missing=nan,
                                          monotone_co...
                                          random_state=42, reg_alpha=None,
                                          reg_lambda=None,
                                          scale_pos_weight=None, subsample=None,
                                          tree_method=None,
                                          validate_parameters=None,
                                          verbosity=None),
                   iid='warn', n_iter=10, n_jobs=None,
```

```
                   param_distributions={'learning_rate': [0.01, 0.025, 0.05],
                                        'n_estimators': [10, 50, 100]},
                   pre_dispatch='2*n_jobs', random_state=42, refit=True,
                   return_train_score=False, scoring='neg_mean_absolute_error',
                   verbose=0)
```

[106]:
```
print('The best parameter for', xgb_regressor.best_params_)
print('The best mean absolute error:',(-1) * xgb_regressor.best_score_)
```

```
The best parameter for {'n_estimators': 100, 'learning_rate': 0.05}
The best mean absolute error: 0.11441886818121688
```

Whilst this again reduced the mean absolute error compared to the other tree models, it is still worse than the mean absolute error achieved compared to the ones obtained from the regression models and the tree models have much greater computational cost.

8 Conclusion

In conclusion, the best model to predict SalePrice was the Ridge Regression model which has a mean absolute error of 0.07871660662991475, the lowest of all the models which were trained.

Throughout this notebook, various data science techniques have been used and I have discovered the huge computational cost of using Tree models for very large datasets. The regression models performed well in trying to predict the feature SalePrice.

Now I have completed this project, I am able to evaluate the outcome, more specifically my implementation of the various machine learning techniques used and available to me.

What Went Well:

- Developed many iterative plots and functions which helped speed up exploratory data analysis
- Improved familiarity with Python modules: NumPy, Pandas, SciPy, SkLearn and Seaborn
- Built a machine learning model with an acceptable mean absolute errror

Even Better If:

- Used more complex encoding techniques to make better use of categorical features
- used feature engineering to create more useful features to train model on
- Stacked models together and used more hyperparameter tuning techniques

One final thing which would have made my job a lot easier would be to join the training and test files right at the beginning and cleaning both of these datasets together. This would have allowed me to apply the exact same transformations to both datasets. Then I would be able to halve the conjoined dataframe so I could run separate training and testing sets into my models. But the added benefit would be the processing of both dataframes would be the same.

In further projects I will aim to develop on this improvements I have identified.

I will now store the best model in a submission file. As I log transformed the sale price values, I will reverse this so I retrieve the actual predicted Sale Price values without scaling.

[107]:
```
samplesubmission = pd.read_csv(r'sample_submission.csv')

output = pd.DataFrame({'Id': samplesubmission.Id, 'SalePrice': (np.
 ↪exp(rr_pred))})
output.to_csv('submission.csv', index=False)
```