House_Price_Prediction

September 5, 2024

1 Linear Regression Model for Predicting House Prices

This Kaggle dataset consists of roughly 3,000 property listings (observations), each with 79 property attributes, and our target, sale price. The goal is to use EDA, data cleaning, preprocessing and linear model to predict home prices given the features of the home. I will follow these steps to a successful submission:

1. Exploratory Data Analysis
2. Data Preprocessing
    1. Fixing Skewness and Outliers
    2. Encoding Categorical Data
    3. Imputing Missing Values
3. Modelling
4. Submission

[ ]:
```
import numpy as np
import pandas as pd
from math import sqrt
from scipy.stats import skew
import matplotlib.pyplot as plt
from sklearn import linear_model
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_absolute_error, mean_squared_error
```

[ ]:
```
plt.style.use(style='fivethirtyeight')
plt.rcParams['figure.figsize'] = (10, 6)
```

2 Exploratory Data Analysis

In this initial investigations on data will be performed to to develop an understanding of the data, discover patterns and spot anomalies.

[ ]:
```
# load the datasets into dataframe
train = pd.read_csv('./house_prices/train.csv')
test = pd.read_csv('./house_prices/test.csv')
```

```
[ ]: # show the first few records of train set
     train.head()
```

```
[ ]:    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]
```

```
[ ]: # check the number of records and columns in both of datasets
     print('No. of records in train dataset: ', len(train.index))
     print('No. of columns in train dataset: ', len(train.columns))
     print('No. of records in test dataset: ', len(test.index))
     print('No. of columns in test dataset: ', len(test.columns))
```

```
No. of records in train dataset:  1460
No. of columns in train dataset:  81
No. of records in test dataset:  1459
No. of columns in test dataset:  80
```

```
[ ]: # check the missing values
     print ('Total missing values in train set', sum(train.isna().sum()))
     print ('Total missing values in test set', sum(test.isna().sum()))
```

```
Total missing values in train set 7829
Total missing values in test set 7878
```

```
[ ]: train['SalePrice'].describe()
```

```
[ ]: count      1460.000000
     mean     180921.195890
```

```
std       79442.502883
min       34900.000000
25%      129975.000000
50%      163000.000000
75%      214000.000000
max      755000.000000
Name: SalePrice, dtype: float64
```

The aobve line code shows that the average sale price of a house is close to 180,000 with most of the values falling within the 130,000 to 215,000 range. Next step is to show the relationship between the columns to examine the correlations between the features and the target.

```
[ ]: numeric_cols = train.select_dtypes(include = [np.number])
     corr = numeric_cols.corr()
     print ('The Most Correlated Features with SalePrice:'), print␣
      ↪(corr['SalePrice'].sort_values(ascending = False)[:10], '\n')
     print ('The Most Uncorrelated Features with SalePrice:'), print␣
      ↪(corr['SalePrice'].sort_values(ascending = False)[-5:])
```

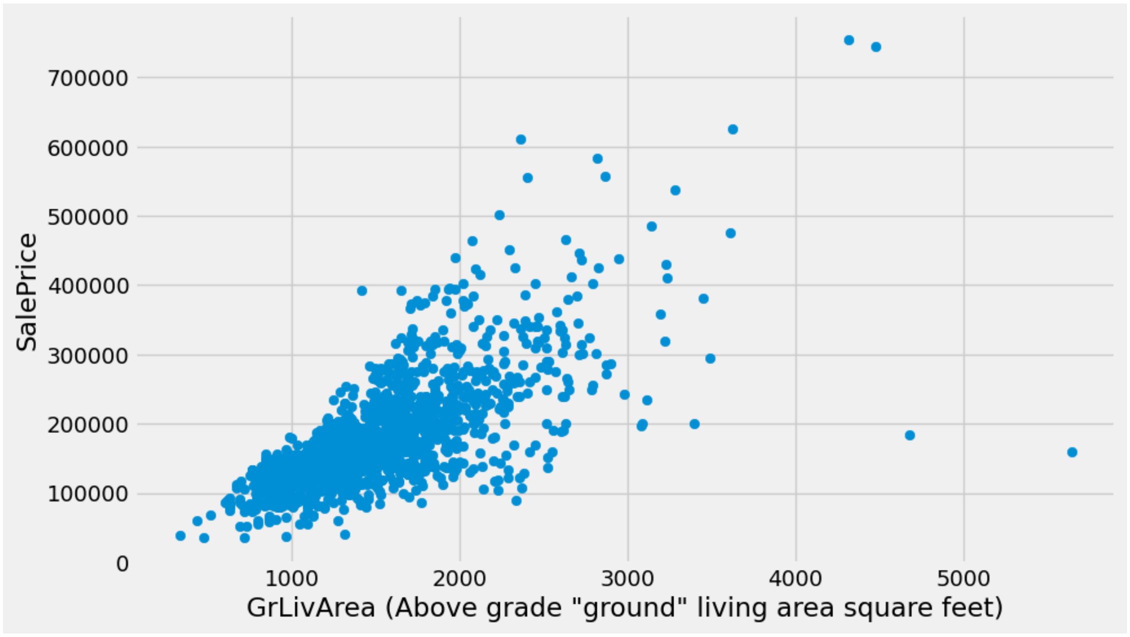
```
The Most Correlated Features with SalePrice:
SalePrice       1.000000
OverallQual     0.790982
GrLivArea       0.708624
GarageCars      0.640409
GarageArea      0.623431
TotalBsmtSF     0.613581
1stFlrSF        0.605852
FullBath        0.560664
TotRmsAbvGrd    0.533723
YearBuilt       0.522897
Name: SalePrice, dtype: float64

The Most Uncorrelated Features with SalePrice:
YrSold          -0.028923
OverallCond     -0.077856
MSSubClass      -0.084284
EnclosedPorch   -0.128578
KitchenAbvGr    -0.135907
Name: SalePrice, dtype: float64
```

```
[ ]: (None, None)
```

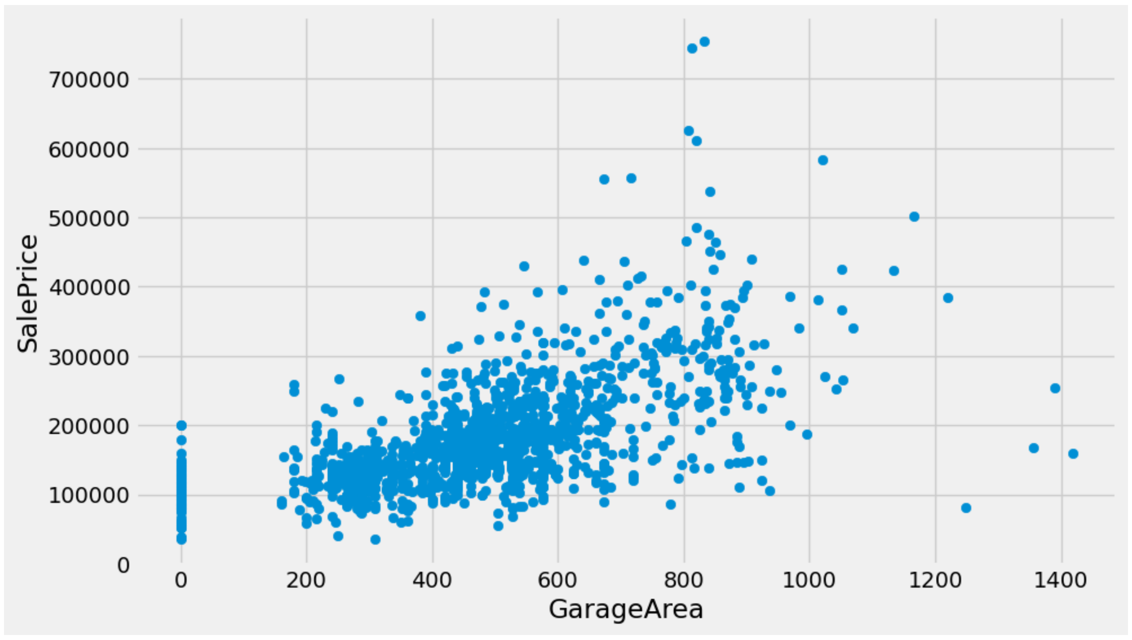
The most correlated features to sale price were the overall quality score (79%), above-ground living area (71%), garage area (64%), and number-of-car garage (62%). Next step is to plot each variable individually against SalePrice in a scatter plot to check outliers as outliers can affect the regression model by pulling the estimated regression line further away from the true population regression line.

```
[ ]: plt.scatter(x = train['GrLivArea'], y = train['SalePrice'])
     plt.ylabel('SalePrice')
     plt.xlabel('GrLivArea (Above grade "ground" living area square feet)')
     plt.show()
```



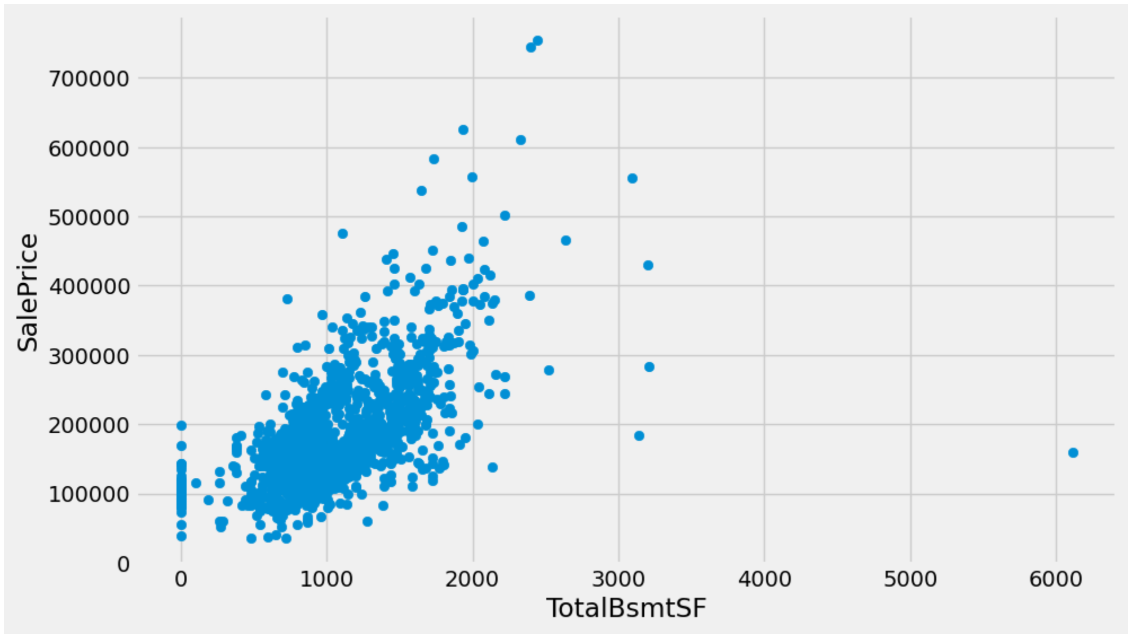
At first glance, there are increases in living area correspond to increases in price, with few outliers.

```
[ ]: plt.scatter(x = train['GarageArea'], y = train['SalePrice'])
     plt.ylabel('SalePrice')
     plt.xlabel('GarageArea')
     plt.show()
```

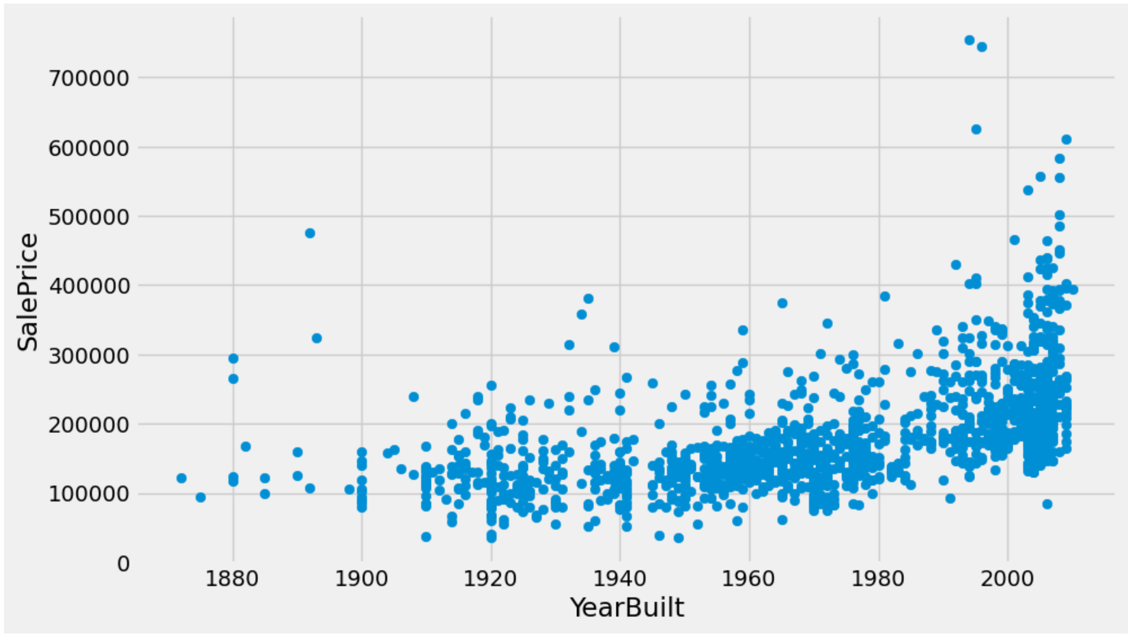


So there are many homes with 0 for GarageArea and there are a few outliers as well!

```
[ ]: plt.scatter(x=train['TotalBsmtSF'] , y=train['SalePrice'])
     plt.ylabel('SalePrice')
     plt.xlabel('TotalBsmtSF')
     plt.show()
```



[ ]:
```
plt.scatter(x=train['YearBuilt'] , y=train['SalePrice'])
plt.ylabel('SalePrice')
plt.xlabel('YearBuilt')
plt.show()
```



3 Data Preprocessing

In this section the data is prepared (transformed, encoded, etc) to make it suitable for a building and training machine learning model. I chose to manually remove certain extreme outliers in the dataset to produce a better fit.

[ ]:
```
# remove GrLivArea outliers
train = train[train['GrLivArea'] < 4500]
```

[ ]:
```
# remove GarageArea outliers
train = train[train['GarageArea'] < 1200]
```

[ ]:
```
# drop columns with percentage of missing values > 80%
train_percentage = train.isnull().sum() / train.shape[0]
print (train_percentage[train_percentage > 0.80])
train = train.drop(train_percentage[train_percentage > 0.80].index, axis = 1)
```

```
Alley          0.937414
PoolQC         0.995873
Fence          0.806740
MiscFeature    0.963549
dtype: float64
```

```
[ ]: # do the same with test data
test_percentage = test.isnull().sum() / test.shape[0]
print (test_percentage[test_percentage > 0.80])
test = test.drop(test_percentage[test_percentage > 0.80].index, axis = 1)
```

```
Alley          0.926662
PoolQC         0.997944
Fence          0.801234
MiscFeature    0.965045
dtype: float64
```

```
[ ]: # encode categorical variables
le = preprocessing.LabelEncoder()
for name in train.columns:
    if train[name].dtypes == 'O':
        train[name] = train[name].astype(str)
        le.fit(train[name])
        train[name] = le.transform(train[name])
```

```
[ ]: # do the same for testset
for name in test.columns:
    if test[name].dtypes == 'O':
        test[name] = test[name].astype(str)
        le.fit(test[name])
        test[name] = le.transform(test[name])
```

There are many ways to handle NaN values, whether to fill with the mean or median, however strings cannot be averaged or median-ed. One way to fill missing values is to impute these missing values according to their probability of occuring in the dataset to avoid single-valued imputation that impacts the quality of inference and prediction.

```
[ ]: # Fill missing values based on probability of occurrence
for column in train.columns:
    if train[column].isnull().sum() > 0:
        # Get non-null values and their counts
        values, counts = np.unique(train[column].dropna(), return_counts=True)
        # Calculate probabilities
        probabilities = counts / counts.sum()
        # Fill NaN values with random choices based on probabilities
        train[column] = train[column].fillna(np.random.choice(values,␣
 ↪p=probabilities))

# Apply the same process to the test set
for column in test.columns:
    if test[column].isnull().sum() > 0:
        values, counts = np.unique(test[column].dropna(), return_counts=True)
        probabilities = counts / counts.sum()
        test[column] = test[column].fillna(np.random.choice(values,␣
↪p=probabilities))
```

[ ]:
```
# apply log transformation to reduce skewness over .75 by taking log(feature +␣
↪1)

skewed_features = train.apply(lambda x: skew(x.dropna())).abs()
skewed_features = skewed_features[skewed_features > 0.75]
skewed_features_names = skewed_features.index

train[skewed_features_names] = np.log1p(train[skewed_features_names])
```

[ ]:
```
# Deal with the skewness in the test data
skewed_features_test = test.apply(lambda x: skew(x.dropna())).abs()
skewed_features_test = skewed_features_test[skewed_features_test > 0.75]
skewed_features_names_test = skewed_features_test.index

test[skewed_features_names_test] = np.log1p(test[skewed_features_names_test])
```

4 Modelling

I will perform a simple linear regression on the dataset to predict house prices. In order to train out the regression model, we need to first split up the data into an X list that contains the features to train on, and a y list with the target variable, in this case, the Price column.

[ ]:
```
X = train.drop(['SalePrice', 'Id'], axis = 1)
y = train['SalePrice']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,␣
↪random_state = 42)
```

Split the data into training and testing set using scikit-learn train_test_split function. We are using 80% of the data for training and 20% for testing, train_test_split() returns four objects:

- **X_train**: the subset of our features used for training
- **X_test**: the subset which will be our 'hold-out' set – what we'll use to test the model
- **y_train**: the target variable SalePrice which corresponds to X_train
- **y_test**: the target variable SalePrice which corresponds to X_test

Now we will import the linear regression class, create an object of that class, which is the linear regression model.

[ ]:
```
lr = linear_model.LinearRegression()
```

Then using the fit method to "fit" the model to the dataset. What this does is nothing but make the regressor "study" the data and "learn" from it.

[ ]:
```
model = lr.fit(X_train, y_train)
```

R-squared is the measure of how close the data are to the fitted regression line, in other words it measures the strength of the relationship between the model and the SalePrice on a convenient 0

– 100% scale.

```
[ ]: # make predictions based on model
     predictions = model.predict(X_test)
```

There are three primary metrics used to evaluate linear models. These are: * Mean absolute error (MAE) * Mean squared error (MSE) * Root mean squared error (RMSE)

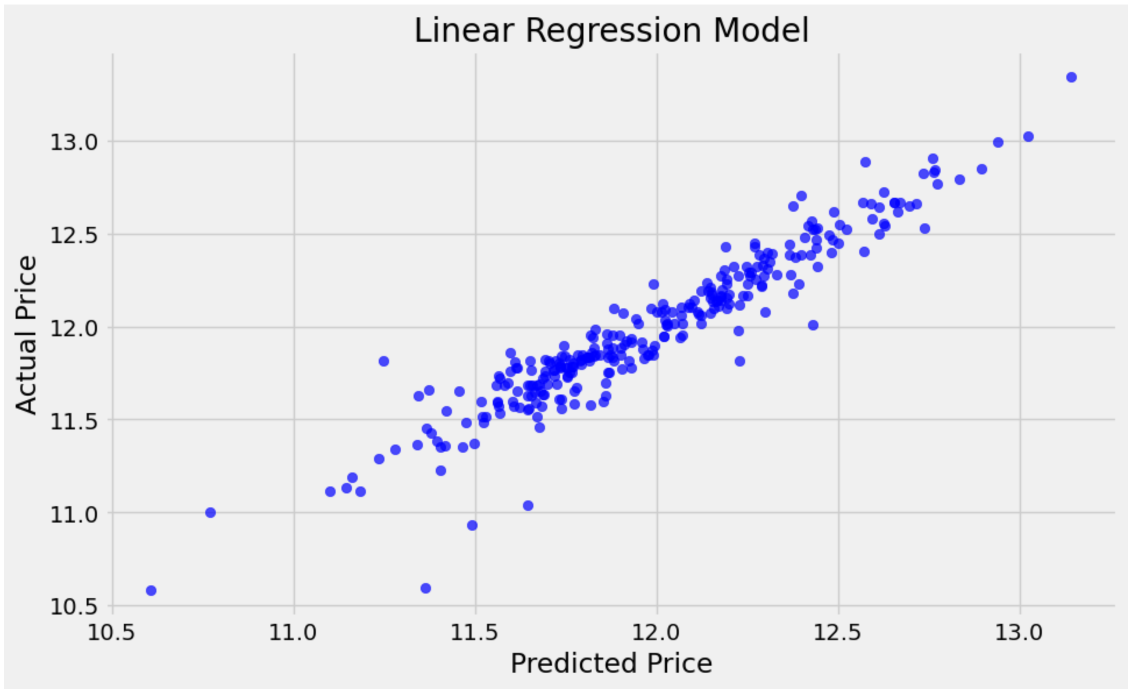
MAE: The easiest to understand. Represents average error. **MSE**: Similar to MAE but noise is exaggerated and larger errors are "punished". It is harder to interpret than MAE as it's not in base units, however, it is generally more popular. **RMSE**: Most popular metric, similar to MSE, however, the result is square rooted to make it more interpretable as it's in base units. It is recommended that RMSE be used as the primary metric to interpret your model.

```
[ ]: print ('MAE is:', mean_absolute_error(y_test, predictions))
     print ('MSE is:', mean_squared_error(y_test, predictions))
     print ('RMSE is:', sqrt(mean_squared_error(y_test, predictions)))
```

```
MAE is: 0.0862277597760611
MSE is: 0.01652730538940944
RMSE is: 0.12855856793465553
```

```
[ ]: # alpha helps to show overlapping data
     plt.scatter(predictions, y_test, alpha = 0.7, color = 'b')
     plt.xlabel('Predicted Price')
     plt.ylabel('Actual Price')
     plt.title('Linear Regression Model')
```

```
[ ]: Text(0.5, 1.0, 'Linear Regression Model')
```



5 Submission

```
[ ]: submission = pd.DataFrame()
     submission['Id'] = test['Id'].astype(int)
```

```
[ ]: temp = test.select_dtypes(include = [np.number]).drop(['Id'], axis = 1).
       ↪interpolate()
```

```
[ ]: predictions = model.predict(temp)
```

```
[ ]: predictions = np.exp(predictions)
     submission['SalePrice'] = predictions
```

```
[ ]: submission.to_csv('submission.csv', index = False)
```