Housing Sale Price Prediction Using Machine Learning Algorithms

Submitted by:

Yash Bhardwaj

ACKNOWLEDGMENT

I whole heartedly thank our SME Sapna Verma, flip robo technologies for their support towards me to complete this project.

I also thank my family for supporting me.

Yash Bhardwaj

Contents

INTRODUCTION

Business Problem

Houses are one of the necessary need of each and every person around the globe and therefore housing and real estate market is one of the markets which is one of the major contributors in the world's economy. It is a very large market and there are various companies working in the domain. Data science comes as a very important tool to solve problems in the domain to help the companies increase their overall revenue, profits, improving their marketing strategies and focusing on changing trends in house sales and purchases. Predictive modelling, Market mix modelling, recommendation systems are some of the machine learning techniques used for achieving the business goals for housing companies. Our problem is related to one such housing company.
A US-based housing company named **Surprise Housing** has decided to enter the Australian market. The company uses data analytics to purchase houses at a price below their actual values and flip them at a higher price. For the same purpose, the company has collected a data set from the sale of houses in Australia.
The company is looking at prospective properties to buy houses to enter the market. We are required to build a model using Machine Learning in order to predict the actual value of the prospective properties and decide whether to invest in them or not. For this company wants to know:

- Which variables are important to predict the price of variable?
- How do these variables describe the price of the house?

Review of Literature

Ask a home buyer to describe their dream house, and they probably won't begin with the height of the basement ceiling or the proximity to an east-west railroad. But this Surprise Housing's dataset proves that much more influences price negotiations than the number of bedrooms or a white-picket fence. However, this dataset related to the Surprise Housing proves that may have more effects on the housing price than the number of bedrooms or floors. Also, I want to predict the reasonable housing price with these aspects of the houses by using this dataset. This dataset contains 81 explanatory variables which related to almost every aspect of residential homes. In the following steps, I will explore this dataset, do feature engineering, fit some machine learning models to predict the housing prices and find which aspects of the house influence the housing prices mostly. Machine learning is closely related to computational statistics, which focus on using mathematical optimization to deliver methods, theory and application domains to solve medical, industry, social and business problems in the real world. We will try multiple linear regression, regularization and finally the ensemble technique. This model also gives us which aspects have big effects on 'HousingSale' price.

Motivation for the Problem Undertaken

The objective or motivation is to model the price of houses with the available independent variables. This model will then be used by the management to understand how exactly the prices vary with the variables. They can accordingly manipulate the strategy of the firm and concentrate on areas that will yield high returns. Further, the model will be a good way for the management to understand the pricing dynamics of a new market. Moreover, this might give the management team an insight of the real time world.

Exploratory Data Analysis

The response variable –SalePrice

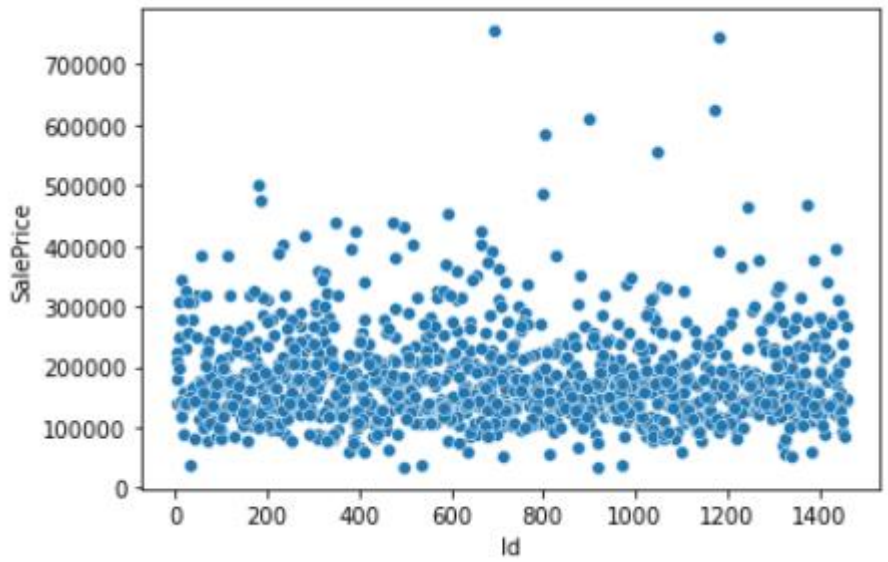


Figure 1 gives us the scatter plot of the sale price. Most of the points are assembled on the bottom. And there seems to be no large outliers in the sale price variable.

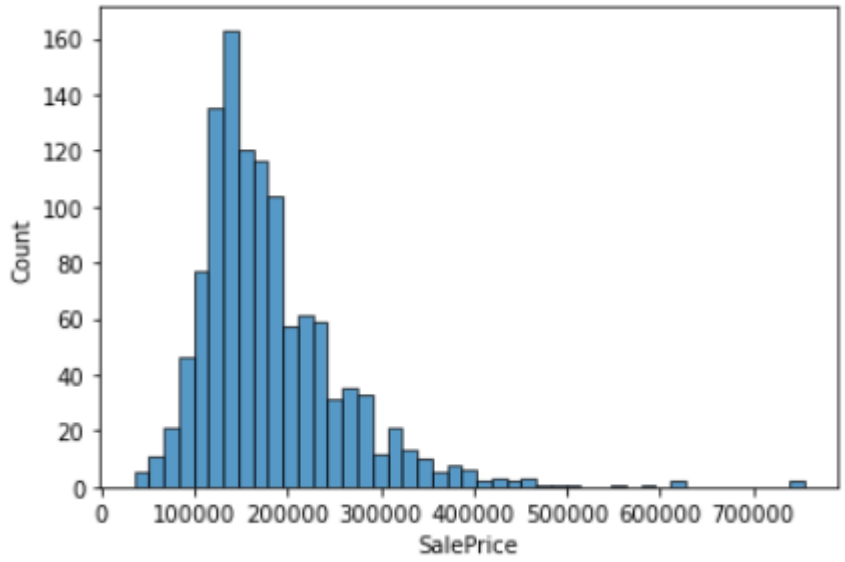


Figure 2 shows that the distribution of sale prices are right skewed, which shows the distribution of the sale prices isn't normal. It is reasonable because few people can afford very expensive houses. We need to take transformation to the sale prices variable before model fitting.

The most related numeric predictors

There are 10 numeric variables with correlations of at least 0.5 with SalePrice. All those correlations are positive. We will visualize the relation between SalePrice and the two predictors with the highest correlation with SalePrice. Overall Quality and the Above Grade Living Area. It also becomes clear the multicollinearity is an issue. For example: the correlation between GarageCars and GarageArea is very high (0.89), and both have similar (high) correlations with SalePrice. The other 6 six variables with a correlation higher than 0.5 with SalePrice are: -TotalBsmtSF: Total square feet of basement area -1stFlrSF: First Floor square feet -FullBath: Full bathrooms abovegrade -TotRmsAbvGrd: Total rooms above grade (does not include
5bathrooms) -YearBuilt: Original construction date -YearRemodAdd: Remodel date (same as construction date if no remodeling or additions).

	SalePrice
OpenPorchSF	0.339500
BsmtFinSF1	0.362874
Foundation	0.374169
GarageYrBlt	0.458007
Fireplaces	0.459611
MasVnrArea	0.463626
YearRemodAdd	0.507831
YearBuilt	0.514408
TotRmsAbvGrd	0.528363
FullBath	0.554988
1stFlrSF	0.587642
TotalBsmtSF	0.595042
GarageArea	0.619000
GarageCars	0.628329
GrLivArea	0.707300
OverallQual	0.789185
SalePrice	1.000000

Figure 3 shows the top 16 highest correlation value of feature variables with the response variable.

Overall Quality (OverallQual)

We find that the highest correlation 0.79 which is between the overall quality and sale price. This overall quality variable rates the overall material and finish of the house as follows:

OverallQual: Rates the overall material and finish of the house

10	Very Excellent
9	Excellent
8	Very Good
7	Good
6	Above Average
5	Average
4	Below Average
3	Fair
2	Poor
1	Very Poor

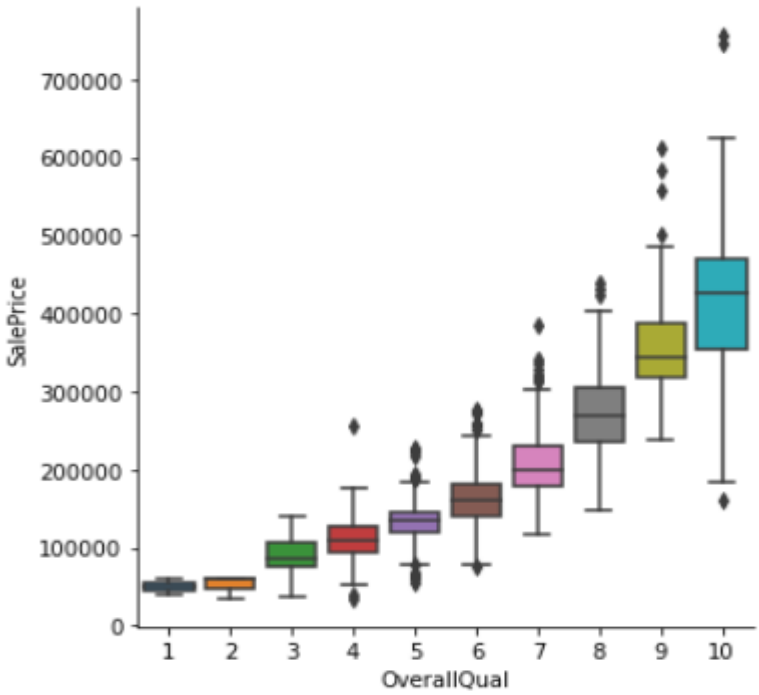


Figure 5 shows the relationship between overall quality and sales price.

We find that there is a positive relationship between the Overall Quality and Sale Price. And it seems like a quadratic relationship or something else like that rather than the linear relationship. This relationship seems easy to be understood. If a house keeper want to improve the overall quality of his house from very poor to poor, he will only need to spend a little money and buy a few items. However, if the house keeper want to improve the overall quality of his house from excellent to very excellent, it will be very difficult and costs him much money.

Above Grade (Ground) Living Area (square feet)

The correlation between this numeric variable and sale price is 0.71 which the second highest. We can give interpretations to the high correlations. Large above grade (ground) living area means large house, and large house means expensive sale price. This makes sense a lot.

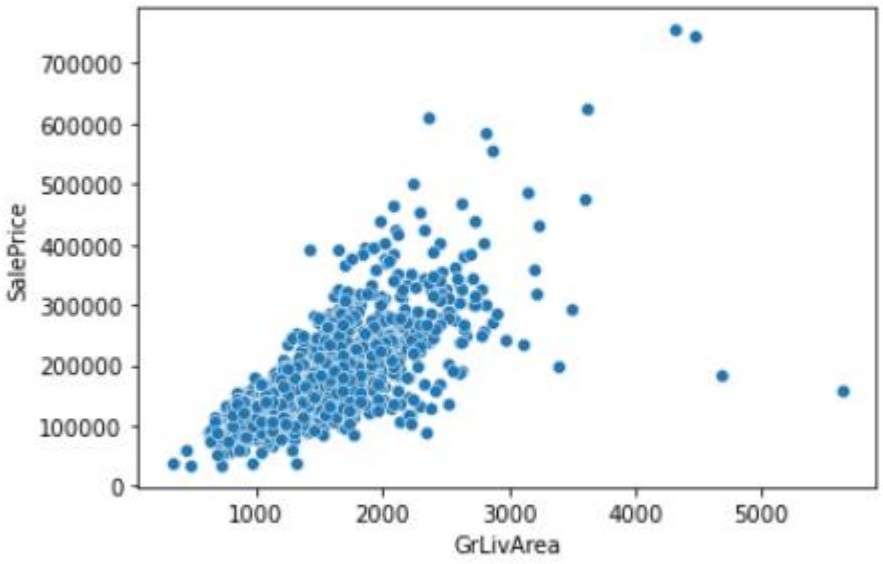


Figure 6 shows the relationship between above grade living area and sale prices.

There are two obvious outliers with high above grade living area but low sale price. Actually, I will not easily delete these two outliers. Because there may be some reasons accounting for the low sale price.

Data Preprocessing

First, we had dropped ‘Id’ from the train set, because it does not contribute to the prediction of sales price.

Missing data

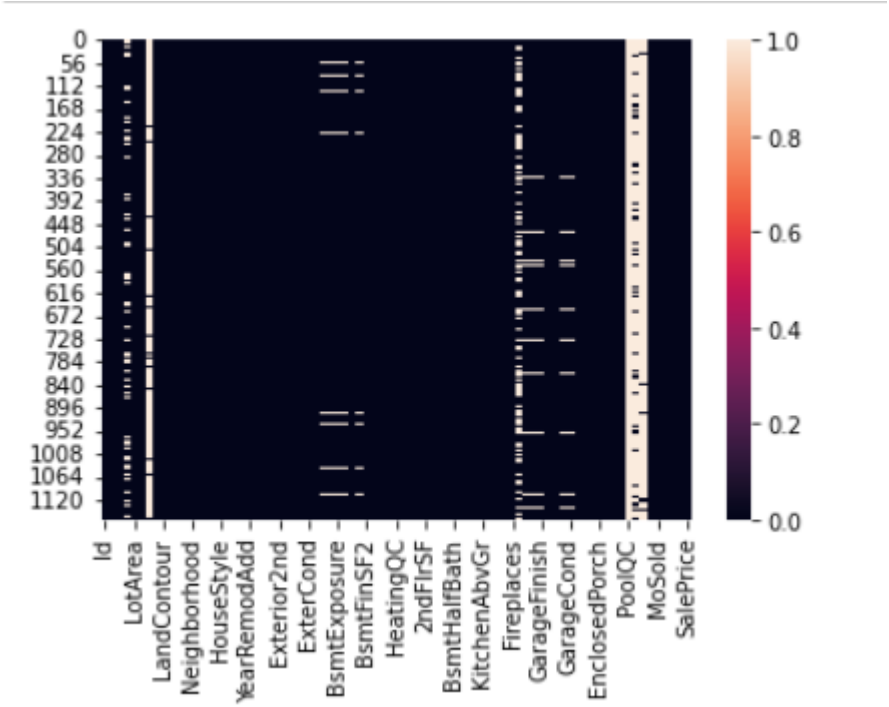


Figure 7 shows missing values present in the dataset.

The dataset has an count of 1168 examples , MiscFeature , PoolQC , Alley , Fence have equal to more than 80% missing data , so we had dropped these from both training and test dataset , as there is no point in imputing them with their respective mode or most frequent value as the data will be biased.

```
from sklearn.impute import SimpleImputer     #importing SimpleImputer

for i in miss_cat:
    train_df[i]=SimpleImputer(strategy='most_frequent').fit_transform(train_df[i].values.reshape(-1,1))

for j in miss_int:
    train_df[j]=SimpleImputer(strategy='mean').fit_transform(train_df[j].values.reshape(-1,1))

for k in miss_cat_test:
    test_df[k]=SimpleImputer(strategy='most_frequent').fit_transform(test_df[k].values.reshape(-1,1))

for l in miss_int_test:
    test_df[l]=SimpleImputer(strategy='mean').fit_transform(test_df[l].values.reshape(-1,1))
```

Here we have used SimpleImputer from sci-kit learn to fill those missing values.

For categorical features SimpleImputer with strategy as most frequent or mode of the data has been used.

For numerical features SimpleImputer with strategy as mean of the data has been used.

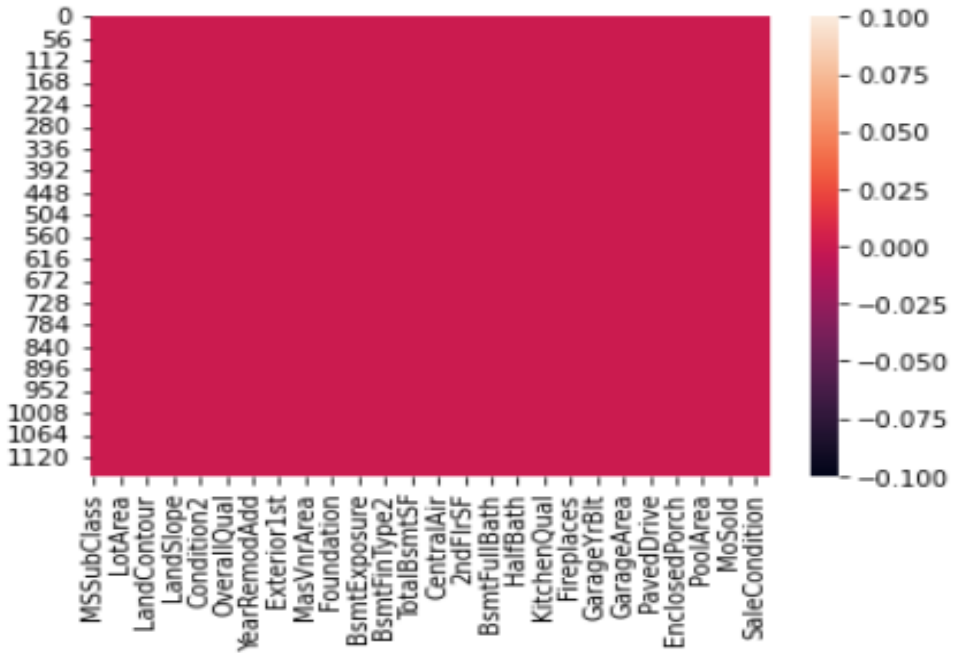


Figure 8 shows that the null values has been replaced.

Converting variables

Before going for the model building phase we need to convert a lot of features into numeric ones later on, so that the machine learning algorithms can process them.

For this we had used Label Encoder from sci-kit learn library. In label encoding, we replace the categorical value with a numeric value between 0 and the number of classes minus 1.

So, as the dataset has more than 35 feature variables of datatype 'object', we had first created a list of those categorical variables and then passed it through label encoder using for loop.

```
cat=[]                      #creating empty list for categorical variable from training set
for i in train_df.columns:
    if train_df[i].dtypes=='object':
        cat.append(i)

cat_test=[]                 #creating empty list for categorical variable from training set
for j in test_df.columns:
    if test_df[j].dtypes=='object':
        cat_test.append(j)
```

```
from sklearn.preprocessing import LabelEncoder    #importing LabelEncoder

for i in cat:
    train_df[i]=LabelEncoder().fit_transform(train_df[i])

for j in cat_test:
    test_df[j]=LabelEncoder().fit_transform(test_df[j])
```

Now, as we have done the preprocessing part, imputed missing data, converted categorical variables into numerical ones, we can go onto building machine learning models using different algorithm.

Building Machine Learning Models

Now we will train several Machine Learning models and compare their results. Note that because the dataset does not provide labels for their testing-set, we need to use the predictions on the training set to compare the algorithms with each other. Later on, we will use cross validation and other evaluation metrics.

So, we had tried in total 8 different algorithms which include, linear regression, 2 regularization methods, k-neighbors, decision tree and 3 ensemble techniques. While trying out different models we had compare the accuracy score for the best fit model.

Best Model?

Score	Model
97.973641	Random Forest
96.741687	Gradient Boosting
87.134516	Ada Boost
83.631103	Linear Regression
83.631092	Lasso
83.629442	Ridge
76.209918	KNN

As we can see, the Random Forest regressor goes on the first place. But first, we must check, how random-forest performs, when we use cross validation.

K-Fold Cross Validation

K-Fold Cross Validation randomly splits the training data into K subsets called folds. Let's image we would split our data into 4 folds (K = 4). Our random forest model would be trained and evaluated 4 times, using a different fold for evaluation every time, while it would be trained on the remaining 3 folds.

The code below perform K-Fold Cross Validation on our random forest model, using 10 folds (K = 10). Therefore it outputs an array with 10 different scores. We then need to compute the mean and the standard deviation for these scores.

```
#Cross validation for Random Forest

rf = RandomForestRegressor()
scores = cross_val_score(rf, X_train, Y_train, cv=10)
print("Scores:", scores,'\n')
print("Mean:", scores.mean(),'\n')
print("Standard Deviation:", scores.std())
```

```
Scores: [0.87786402 0.89258741 0.86888924 0.65629995 0.91224612 0.74175333
 0.86003129 0.92597692 0.85211491 0.82860826]

Mean: 0.841637145713736

Standard Deviation: 0.07854574936370462
```

This looks much more realistic than before. Our model has an average accuracy of 84% with a standard deviation of 7 %. The standard deviation shows us, how precise the estimates are.

This means in our case that the accuracy of our model can differ **+** — 7%.

The accuracy is still really good and since random forest is an easy to use model, we had try to increase its performance even further in the following section.

Random Forest

Random Forest is a supervised learning algorithm. Like you can already see from its name, it creates a forest and makes it somehow random. The 'forest' it builds, is an ensemble of Decision Trees, most of the time trained with the "bagging" method. The general idea of the bagging method is that a combination of learning models increases the overall result.

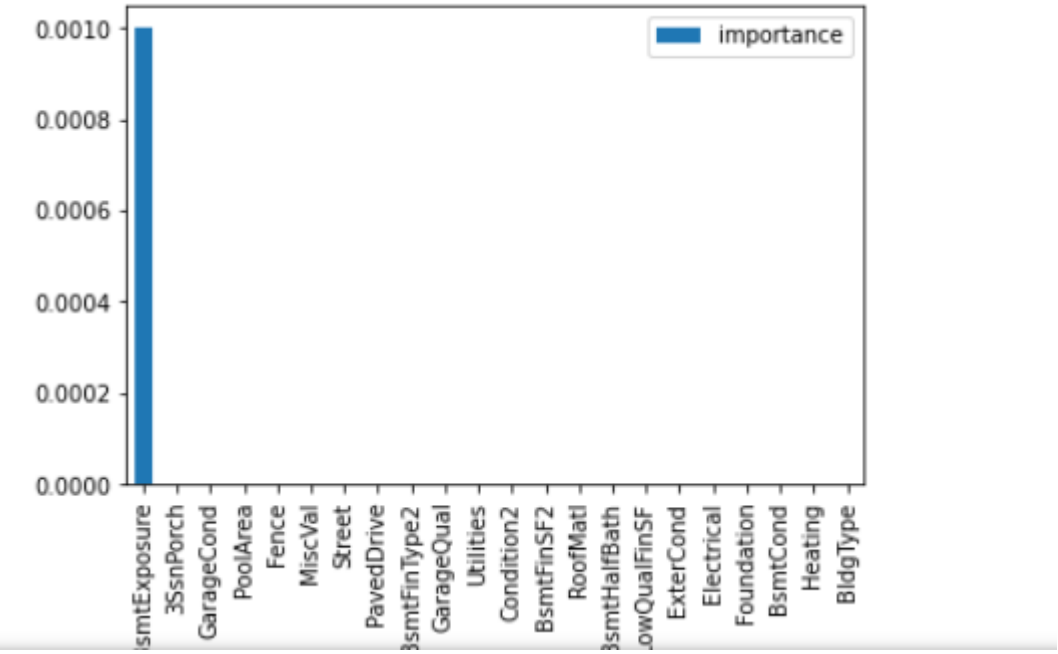
To say it in simple words: Random forest builds multiple decision trees and merges them together to get a more accurate and stable prediction.

One big advantage of random forest is, that it can be used for both classification and regression problems, which form the majority of current machine learning systems. With a few exceptions a random-forest classifier has all the hyperparameters of a decision-tree classifier and also all the hyperparameters of a bagging classifier, to control the ensemble itself.

The random-forest algorithm brings extra randomness into the model, when it is growing the trees. Instead of searching for the best feature while splitting a node, it searches for the best feature among a random subset of features. This process creates a wide diversity, which generally results in a better model. Therefore when you are growing a tree in random forest, only a random subset of the features is considered for splitting a node. You can even make trees more random, by using random thresholds on top of it, for each feature rather than searching for the best possible thresholds (like a normal decision tree does).

Feature Importance

Another great quality of random forest is that they make it very easy to measure the relative importance of each feature. Sci-kit learn measure a features importance by looking at how much the tree nodes, that use that feature, reduce impurity on average (across all trees in the forest). It computes this score automatically for each feature after training and scales the results so that the sum of all importance is equal to 1.



It came out to be that 21 feature variables don't play a significant role in our random forest regressor prediction process. Because of that we will drop them from the dataset and train the regressor again. We could also remove more or less features, but this would need a more detailed investigation of the features effect on our model.

Training the Random Forest model again after dropping the zero feature importance variables:

```
# Random Forest

random_forest = RandomForestRegressor(n_estimators=100, oob_score = True)
random_forest.fit(X_train, Y_train)
Y_prediction = random_forest.predict(X_test)

random_forest.score(X_train, Y_train)

acc_random_forest = round(random_forest.score(X_train, Y_train) * 100, 2)
print(round(acc_random_forest,2,), "%")
```

```
98.16 %
```

Our random forest model predicts as good as it did before.

There is also another way to evaluate a random-forest regressor, which is probably much more accurate than the score we used before. What I am talking about is the **out-of-bag samples** to estimate the generalization accuracy. It is as accurate as using a test set of the same size as the training set. Therefore, using the out-of-bag error estimate removes the need for a set aside test set.

```
#out of bag sample score

print("oob score:", round(random_forest.oob_score_, 4)*100, "%")
```

```
oob score: 85.99 %
```

Now we can start tuning the hyperameters of random forest.

Hyperparameter Tuning

Below is the code of the hyperparamter tuning for the parameters criterion, min_samples_leaf and min_samples_split.

```
param_grid = { "criterion" : ["squared_error", "absolute_error", "poisson"], "min_samples_leaf" : [1, 5, 10], "min_samples_split"
from sklearn.experimental import enable_halving_search_cv
from sklearn.model_selection import HalvingGridSearchCV
rf = RandomForestRegressor(n_estimators=100, max_features='auto', oob_score=True, random_state=1, n_jobs=-1)
clf = HalvingGridSearchCV(estimator=rf, param_grid=param_grid)
clf.fit(X_train, Y_train)
clf.best_params_
```

```
{'criterion': 'squared_error', 'min_samples_leaf': 1, 'min_samples_split': 2}
```

Now that we have optimum values for a few parameters, we are going to test these parameters and find out the o-o-b score:

```
# Random Forest
random_forest = RandomForestRegressor(criterion = 'squared_error',
                                      min_samples_leaf = 1,
                                      min_samples_split = 2,
                                      max_features='auto',
                                      oob_score=True,
                                      random_state=1,
                                      n_jobs=-1)

random_forest.fit(X_train, Y_train)
Y_prediction = random_forest.predict(X_test)

random_forest.score(X_train, Y_train)

print("oob score:", round(random_forest.oob_score_, 4)*100, "%")
```

```
oob score: 86.81 %
```

Out-of-bag score for the model has been increased after hyperparameter tuning.

Now that we have a proper model, we can start evaluating its performance in a more accurate way. Previously we only used accuracy and the out-of-bag score, which is just another form of accuracy.

Evaluation

Mean Squared Error

Mean Squared Error or MSE for short, is a popular error metric for regression problems.

It is also an important loss function for algorithms fit or optimized using the least squares framing of a regression problem. Here "*least squares*" refers to minimizing the mean squared error between predictions and expected values. The MSE is calculated as the mean or average of the squared differences between predicted and expected target values in a dataset.

MSE = 1 / N * sum for i to N (y_i – yhat_i)^2
Where *y_i* is the i'th expected value in the dataset and *yhat_i* is the i'th predicted value. The difference between these two values is squared, which has the effect of removing the sign, resulting in a positive error value.
The squaring also has the effect of inflating or magnifying large errors. That is, the larger the difference between the predicted and expected values, the larger the resulting squared positive error. This has the effect of "*punishing*" models more for larger errors when MSE is used as a loss function. It also has the effect of "*punishing*" models by inflating the average error score when used as a metric.

```
print('MSE:',mean_squared_error(Y_train,predictions))
```

```
MSE: 890304588.7224373
```

Root Mean Squared Error

The Root Mean Squared Error MSE, is an extension of the mean squared error. Importantly, the square root of the error is calculated, which means that the units of the RMSE are the same as the original units of the target value that is being predicted.

For example, if your target variable has the units "*dollars*," then the RMSE error score will also have the unit "*dollars*" and not "*squared dollars*" like the MSE.

As such, it may be common to use MSE loss to train a regression predictive model, and to use RMSE to evaluate and report its performance.

The RMSE can be calculated as follows:

RMSE = sqrt(1 / N * sum for i to N (y_i – yhat_i)^2)
Where *y_i* is the i'th expected value in the dataset, *yhat_i* is the i'th predicted value, and *sqrt()* is the square root function.
We can restate the RMSE in terms of the MSE as:

RMSE = sqrt(MSE)
Note that the RMSE cannot be calculated as the average of the square root of the mean squared error values. This is a common error made by beginners and is an example of Jensen's Inequality.
You may recall that the square root is the inverse of the square operation. MSE uses the square operation to remove the sign of each error value and to punish large errors. The square root reverses this operation, although it ensures that the result remains positive.

The root mean squared error between your expected and predicted values can be calculated using the mean_squared_error function() from the scikit-learn library.
By default, the function calculates the MSE, but we can configure it to calculate the square root of the MSE by setting the "*squared*" argument to *False*.
The function takes a one-dimensional array or list of expected values and predicted values and returns the mean squared error value.

```
print('RMSE:',np.sqrt(mean_squared_error(Y_train,predictions)))
```

```
RMSE: 29837.972262243915
```

Mean Absolute Error

Mean Absolute Error or MAE, is a popular metric because, like RMSE, the units of the error score match the units of the target value that is being predicted.

Unlike the RMSE, the changes in MAE are linear and therefore intuitive.

That is, MSE and RMSE punish larger errors more than smaller errors, inflating or magnifying the mean error score. This is due to the square of the error value. The MAE does not give more or less weight to different types of errors and instead the scores increase linearly with increases in error.

As its name suggests, the MAE score is calculated as the average of the absolute error values. Absolute or *abs()* is a mathematical function that simply makes a number positive. Therefore, the difference between an expected and predicted value may be positive or negative and is forced to be positive when calculating the MAE.
The MAE can be calculated as follows:

- MAE = 1 / N * sum for i to N abs(y_i – yhat_i)
  Where *y_i* is the i'th expected value in the dataset, *yhat_i* is the i'th predicted value and *abs()* is the absolute function.

```
print('MAE:',mean_absolute_error(Y_train,predictions))
```

```
MAE: 18169.7277739726
```

R Squared score

R-squared (R^2) is a statistical measure that represents the proportion of the variance for a dependent variable that's explained by an independent variable or variables in a regression model. Whereas correlation explains the strength of the relationship between an independent and dependent variable, R-squared explains to what extent the variance of one variable explains the variance of the second variable. So, if the R^2 of a model is 0.50, then approximately half of the observed variation can be explained by the model's inputs.

R-squared values range from 0 to 1 and are commonly stated as percentages from 0% to 100%. An R-squared of 100% means that all movements of a security (or another dependent variable) are completely explained by movements in the index (or the independent variable(s) you are interested in).

```
print('r2_score:',r2_score(Y_train,predictions)*100)
```

```
r2_score: 85.76044934360606
```

The Random Forest model is giving us a high r-squared score. 85% of R-squared means up to 85% of feature variables movements are completely explained by movements of label variable.

Conclusion

The objective of this project is to fit models to predict the housing sale price and find some important aspects of the house .In order to achieve this goal, I had fit eight different regression models to the dataset: linear regression, lasso regression, ridge regression, k-nearest neighbors, decision tree, ada boost, gradient boosting and random forest.

Findings

As for the first model -linear regression, it doesn't meet the assumption of equality of the variances. Therefore we can't use the linear model as the candidate of our final model.

In order to deal with this problem, I try the second and third model -lasso regression and ridge regression but the accuracy score looked not so good.

The fourth and fifth model were k-nearest neighbors and decision tree both these model had a very low accuracy score.

Afterwards, we tried three ensemble algorithms ada boost was the sixth model with a very low accuracy score so we moved ahead. Gradient boosting was the seventh model with a very high accuracy score up to now. But we had to try the last model also, so finally, we had tried random forest which gave us the highest accuracy score. But before, selecting random forest as the model to go on with, we had use cross validation score to check how the random forest model is really performing. Cross validation score of random forest model came out to be much realistic than its accuracy score. We also check for out-of-bag score for a better understanding that how the model is working.

Thus, we had used hyperparameter tuning to find the optimum value of model's parameters. After re-training our model with tuned parameters we again checked for out-of-bag score and it had increased.

What's more, from the feature importance plot of the random forest, we could know that the overall quality had the very high feature importance when compared to all the other feature variables. Above grade living area also had reasonable feature importance when compared to all the other features but very low with respect to overall quality. We also saw that around 21 feature variables had a feature importance value of zero.

Limitations of this work and Scope for Future Work

Of course there is still a big room for improvement as we in this project did not do much extensive feature engineering. We had taken the more simple way of looking into data, analysing it and making some models.

The limitations of this project is that it lacks extensive feature engineering, no reduction of multicollinearity and intensive hyperparameter tuning. Moreover, some other regressors like Xgboost could have been use in this case i.e. regression problem.

The scope for future work with this project can be so wide. I had suggest like doing a more extensive feature engineering, by comparing and plotting the features against each other and identifying and removing the noisy features. Feature engineering also includes creation of new variables using existing feature variables, an example would be that the dataset has many variables related to bathrooms inside a house, so one can create a new variable by analysing all these variables. So, extensive feature engineering has very great future scope to further extend the study of this project. Thus, improving all over results.

Before building regression models multicollinearity between feature has to be checked using various methods. One way is using variance inflation factor. A variance inflation factor higher than 4-10, results in multicollinearity. Higher the variance inflation factor higher the multicollinearity. It can be reduced by dropping one of the multicollinear features with lowest correlation and high variance inflation factor. Multicollinearity can be reduced by doing an extensive feature engineering too.

Another thing that can improve the overall result of the model would be a more extensive hyperparameter tuning on several machine learning models. Finding optimum value for several parameters of an algorithm model optimizes the overall performance of the model by some extent.

References

- https://towardsdatascience.com/predicting-the-survival-of-titanic-passengers-30870ccc7e8
- https://escholarship.org/uc/item/3ft2m7z5
- https://www.investopedia.com/terms/r/r-squared.asp#:~:text=R%2Dsquared%20values%20range%20from,)%20you%20are%20interested%20in
- https://machinelearningmastery.com/regression-metrics-for-machine-learning/