CAPSTONE PROJECT

ON

HOUSE PRICE PREDICTION ANALYSIS

Ashish Mohan

RUID: 197005922

Under the guidance of:

Professor Meng Qu

Table of Contents:

1. Introduction

The dataset used in this project is an ongoing Kaggle competition based on the ’Ames Housing Dataset ‘of Dean De Cook. The dataset describes the sale of individual residential property in Ames, Iowa from 2006 to 2010.

Objective:

The prediction of house prices is an essential and important part in the real estate world as there is a lot of money involved in the buying and selling of houses. An accurate prediction of house prices is of high importance for the seller and the prospective buyers.

For a learner and an aspiring data analyst like myself, the presence of the many features present in the dataset, offered a lot of room for data analysis, feature engineering and exploration of algorithms which made the project engaging and allowed to compare different techniques and models present in the machine learning domain.

Target audience:

Being able to predict the real estate prices can help two major demographics viz. the real estate investors and prospective buyers. By creating a model that uses the existing information on houses to predict and assess if the asking price of a house is higher or lower than the actual value of the house can be a great help for people trying to make significant investments in real estate.

2. Problem statement

Given the diverse nature of variables the dataset has, the main goal of this project is to assess and analyze how the price of a house be predicted from a variety of numerical, categorical, correlated, and uncorrelated features. Also, to perform feature engineering and how the different algorithms compare in terms of predictions and accuracy.

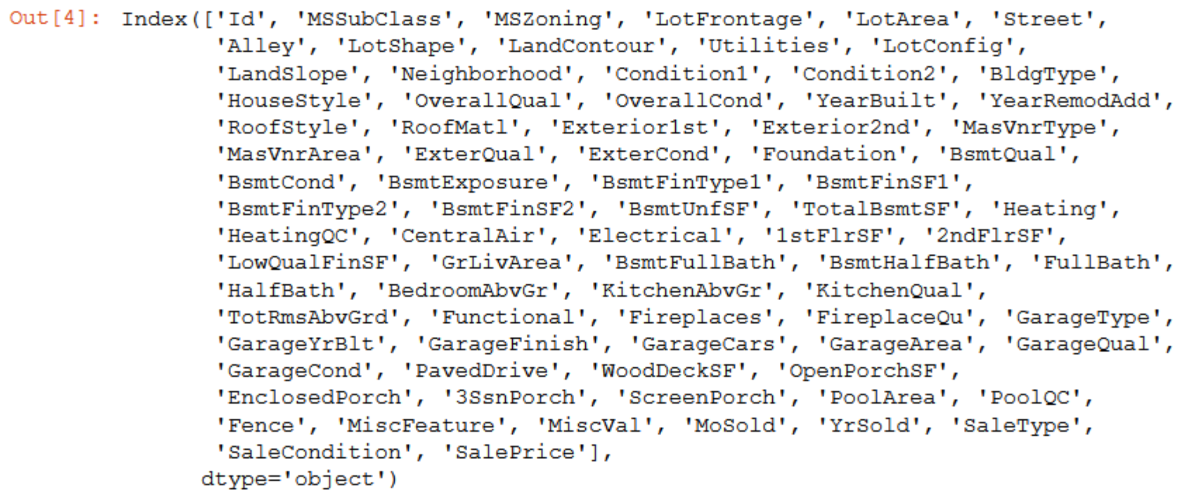
3. Dataset

With 79 explanatory variables describing every aspect of residential homes in Ames, Iowa, the aim is to build a model that predicts the final price of a house.

Here is a brief description of the data file. An extensive description of all the features can be found in the original documentation of the data. (Link in references section)

- SalePrice — the target variable to predict.
- Address (including neighborhood and utilities)
- Sale Condition (including year sold and built)

- House Condition (including type of rooms, measurement of basement, roof, heating system, air conditioner, electrical system, measurement of floor, fireplaces, measurement of garage, pool, etc.)



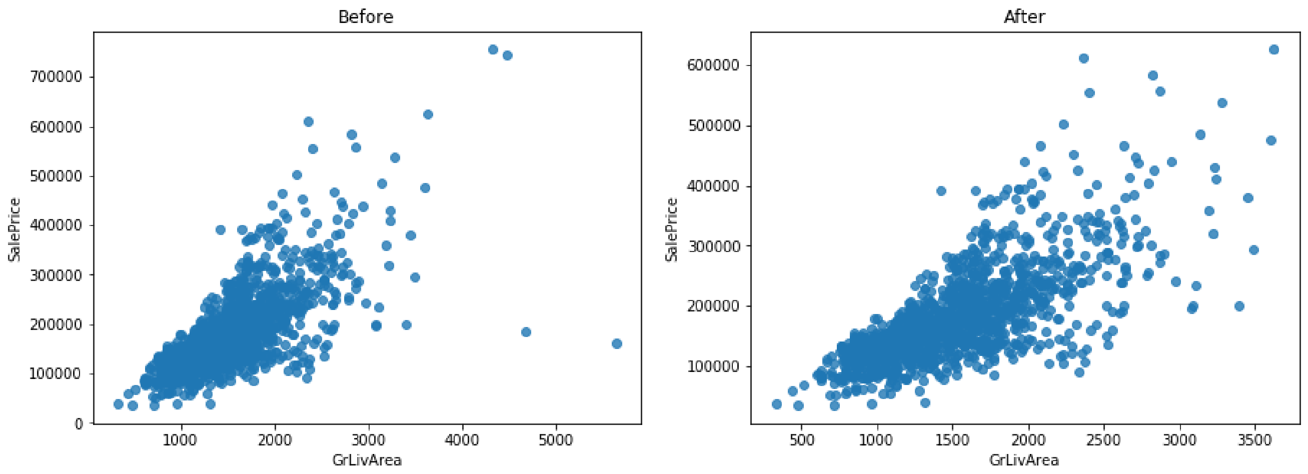
```
Out[4]: Index(['Id', 'MSSubClass', 'MSZoning', 'LotFrontage', 'LotArea', 'Street',
               'Alley', 'LotShape', 'LandContour', 'Utilities', 'LotConfig',
               'LandSlope', 'Neighborhood', 'Condition1', 'Condition2', 'BldgType',
               'HouseStyle', 'OverallQual', 'OverallCond', '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')
```

List of attributes present in the dataset

Because the number of variables is on the higher end, it is important that unwanted or features that aren't useful are removed or combined to prevent possible co-linearity between the variables when applying machine learning models on the data.

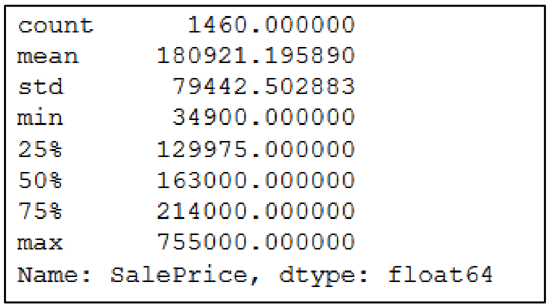
4. Experiment

a) Outlier handling



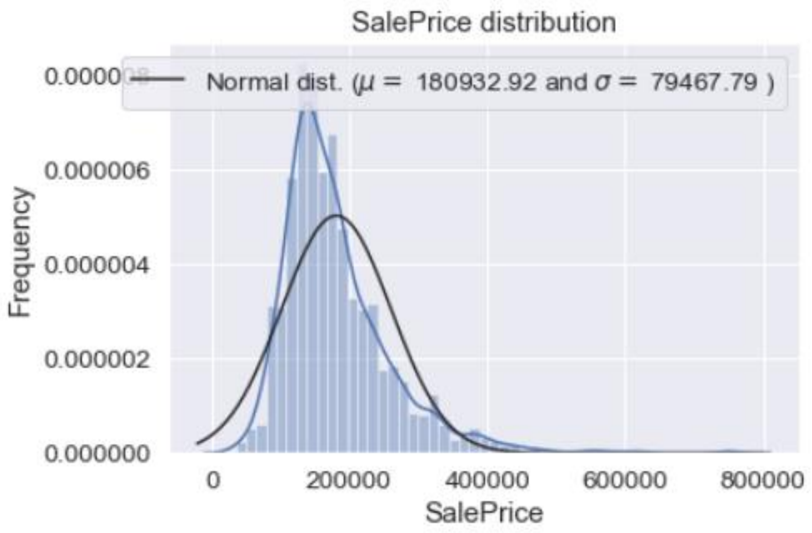
The data description document mentions an obvious presence of outliers on plotting between living area above ground ('GrLivArea') and Saleprice, as the sale price is exceptionally low for a house with living area greater than 4000 sq ft. Hence, removing the points greater than 4000 sq ft. as they have erratic Saleprice.

b) Analysis of target variable 'Saleprice'

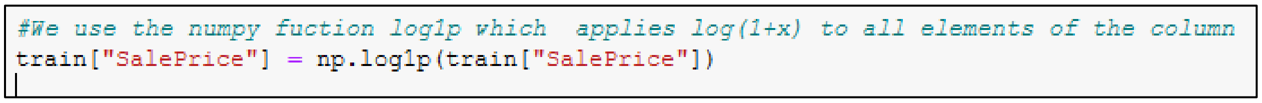


```
count      1460.000000
mean     180921.195890
std       79442.502883
min       34900.000000
25%      129975.000000
50%      163000.000000
75%      214000.000000
max      755000.000000
Name: SalePrice, dtype: float64
```

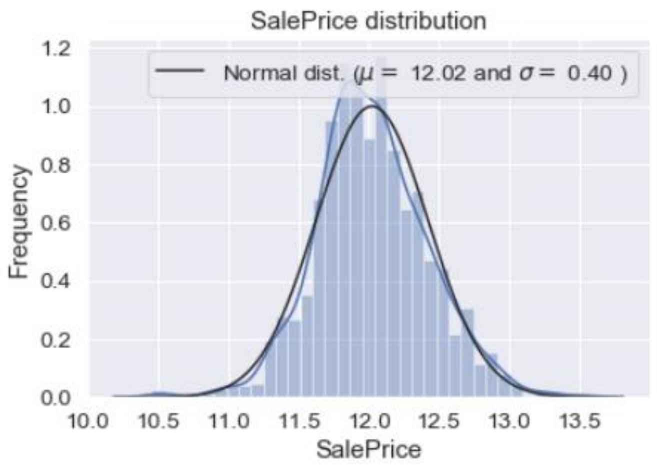
After performing descriptive statistics on the target variable, we see that the median is lesser than the mean, which indicates the presence of outliers. Additionally, it is also seen that the average sale price of a house in our dataset is close to 180,000 USD with most of the values falling within the 130,000 USD to 215,000 USD range.



By visually examining the distribution of the house prices we can see that the target variable is positively or rightly skewed. However, for our analysis we will try to make the distribution a normal one for better performance of the machine learning models. Hence, by transforming it using numpy's log1p function we get the below normal distribution of target variable 'SalePrice'.

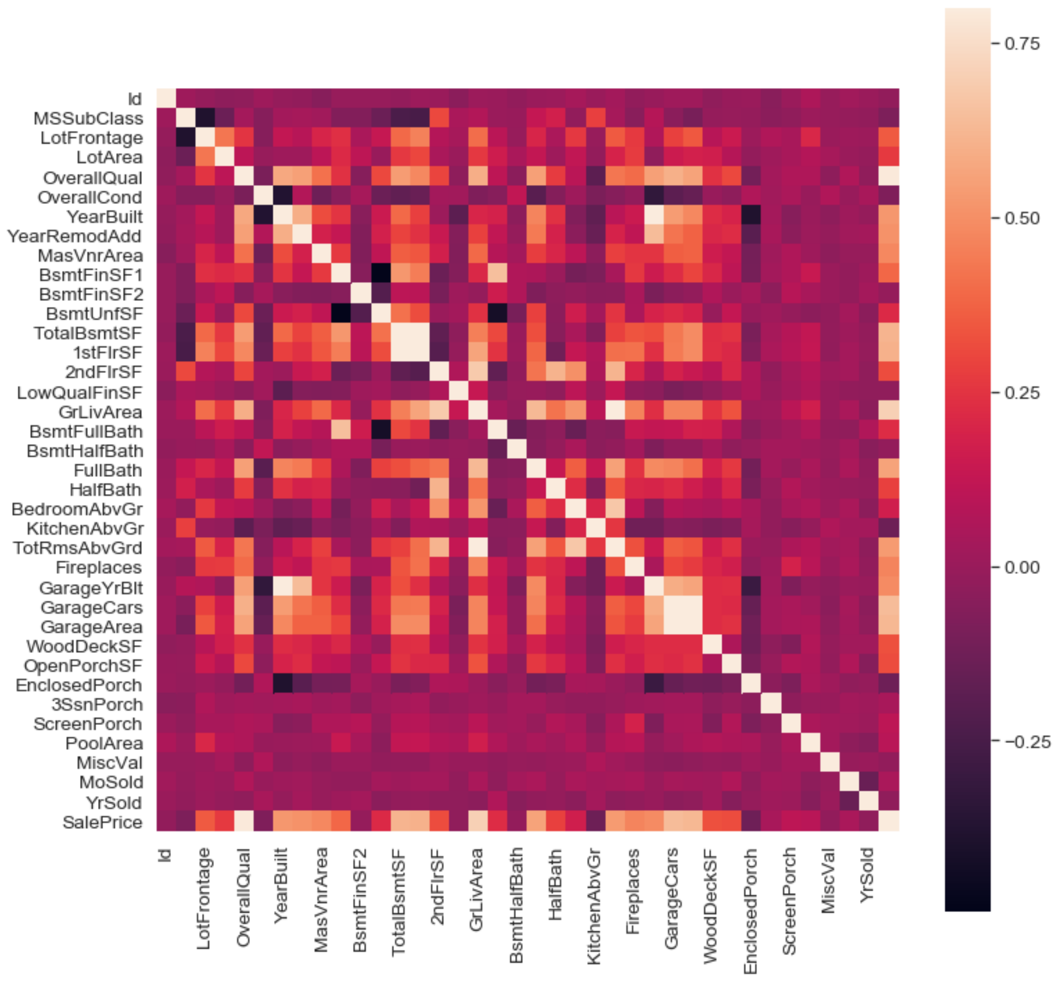


```
#We use the numpy fuction log1p which  applies log(1+x) to all elements of the column
train["SalePrice"] = np.log1p(train["SalePrice"])
```



c) Correlation between different features

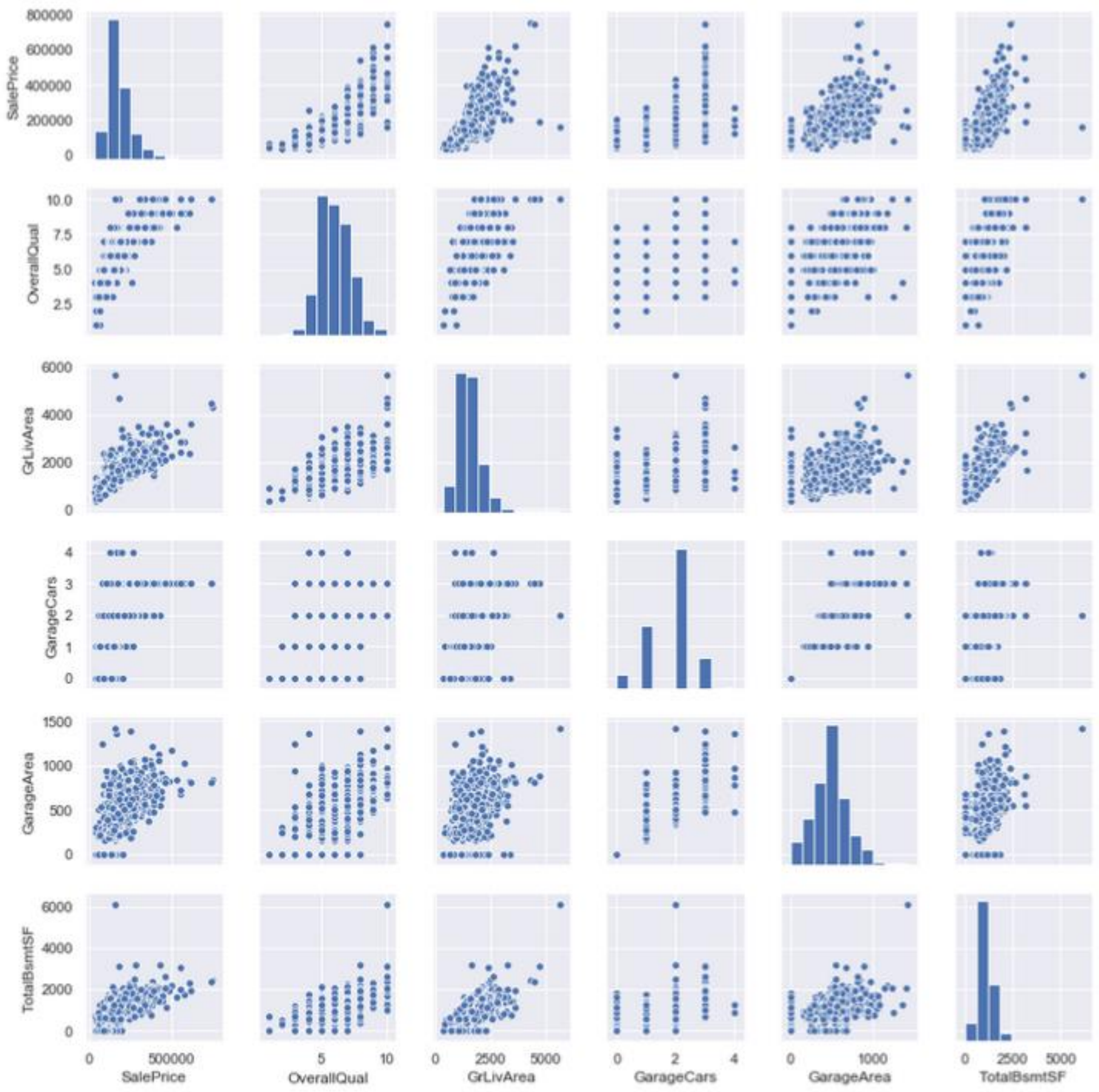
Perfect collinearity between regressors would violate the regression assumptions, hence, we examine the correlation between the independent variables with a correlation matrix/heatmap below.



Strong correlation can be seen in the following cases:

1) We see a significant correlation between variables 'TotalBsmtSF' & '1stFlrSF' variables and also between the 'Garage' variables. This indicates multicollinearity present between these variables which will be removed later through feature engineering.

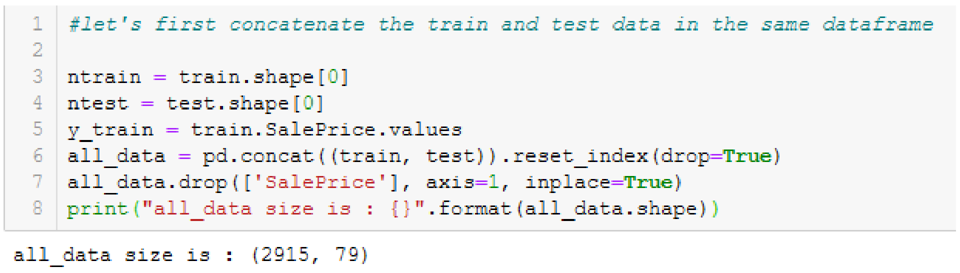
2) Another significant correlation we see is with the target variable 'SalePrice'. Strong correlations are seen with variables 'GrLivArea', 'TotalBsmtSF', and 'OverallQual'.



The above pairplot helps us visualize the relationship of the target variable with five of the most correlated features. As we can see above, there is a general positive correlation trend among the different variables and the target variable SalePrice.

d) Feature Engineering:

For feature engineering, there is a need to concatenate both the training and test datasets into one common dataset, so that feature analysis and modification can be done effectively and consistently. The concatenated dataset will later be split back into test and train datasets when applying machine learning models.



```python
#let's first concatenate the train and test data in the same dataframe

ntrain = train.shape[0]
ntest = test.shape[0]
y_train = train.SalePrice.values
all_data = pd.concat((train, test)).reset_index(drop=True)
all_data.drop(['SalePrice'], axis=1, inplace=True)
print("all_data size is : {}".format(all_data.shape))
```

```
all_data size is : (2915, 79)
```

Concatenating both the train and test data into one common data frame.

Handling missing data:

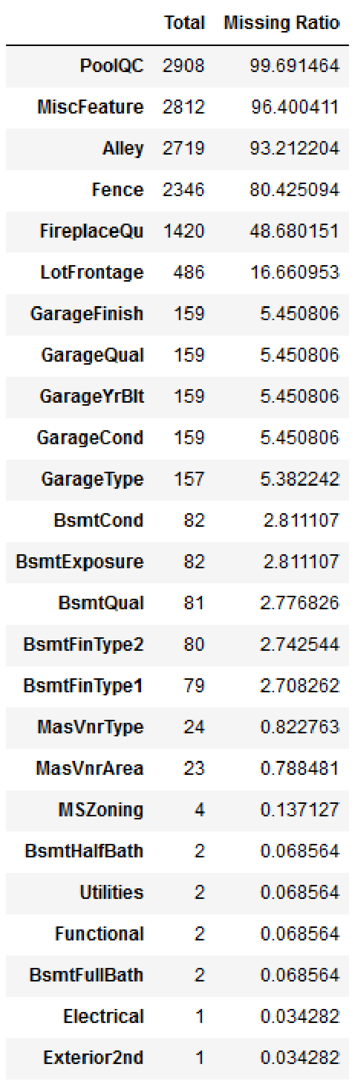
As seen in the adjacent table, the strategy used to handle missing data is as follows:

a) Drop attributes that have more than 15% of their data missing. Also, it is to be seen that the set of variables that have most number of missing data (Eg. 'PoolQC', 'MiscFeature', 'Alley', etc.) aren't important factors when it comes to buying a house.

b) Dropping attributes like 'MasVnrArea' and 'MasVnrType', we can consider that these variables are not essential. Furthermore, they have a strong correlation with 'YearBuilt' and 'OverallQual' which are already considered. Thus, information would not be lost if we drop 'MasVnrArea' and 'MasVnrType'

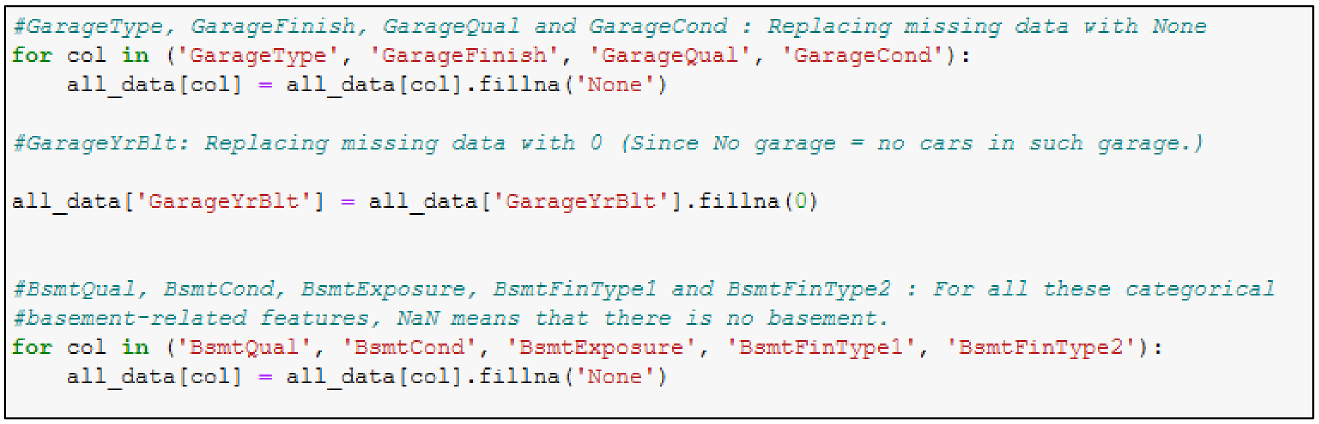
c) Finally, there is one missing observation in 'Electrical'. Since it is just one observation, this observation can be deleted, and the variable kept.

d) For the Garage and Basement variables, the missing values can be replaced according to the type of variables (Numerical or categorical).



	Total	Missing Ratio
PoolQC	2908	99.691464
MiscFeature	2812	96.400411
Alley	2719	93.212204
Fence	2346	80.425094
FireplaceQu	1420	48.680151
LotFrontage	486	16.660953
GarageFinish	159	5.450806
GarageQual	159	5.450806
GarageYrBlt	159	5.450806
GarageCond	159	5.450806
GarageType	157	5.382242
BsmtCond	82	2.811107
BsmtExposure	82	2.811107
BsmtQual	81	2.776826
BsmtFinType2	80	2.742544
BsmtFinType1	79	2.708262
MasVnrType	24	0.822763
MasVnrArea	23	0.788481
MSZoning	4	0.137127
BsmtHalfBath	2	0.068564
Utilities	2	0.068564
Functional	2	0.068564
BsmtFullBath	2	0.068564
Electrical	1	0.034282
Exterior2nd	1	0.034282

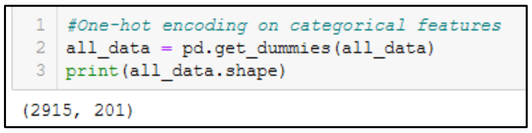
As seen in the below code snippet, the missing values for categorical variables under the Garage and Basement variables can be replaced with 'None'. Whereas, for numerical variables the missing values are simply replaced with 0.



```python
#GarageType, GarageFinish, GarageQual and GarageCond : Replacing missing data with None
for col in ('GarageType', 'GarageFinish', 'GarageQual', 'GarageCond'):
    all_data[col] = all_data[col].fillna('None')

#GarageYrBlt: Replacing missing data with 0 (Since No garage = no cars in such garage.)

all_data['GarageYrBlt'] = all_data['GarageYrBlt'].fillna(0)


#BsmtQual, BsmtCond, BsmtExposure, BsmtFinType1 and BsmtFinType2 : For all these categorical
#basement-related features, NaN means that there is no basement.
for col in ('BsmtQual', 'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFinType2'):
    all_data[col] = all_data[col].fillna('None')
```

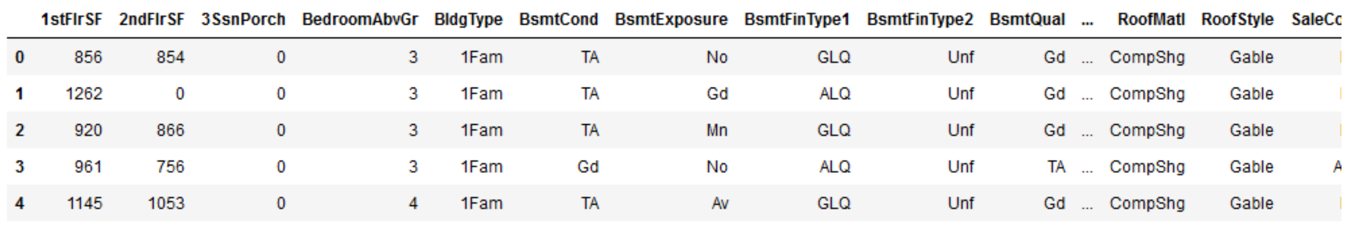
e) Categorical to dummy/indicator variable conversion

Before we proceed with applying our data on the models, it is necessary to convert the available categorical data to numeric data. The get_dummies() feature of pandas helps us to achieve that.



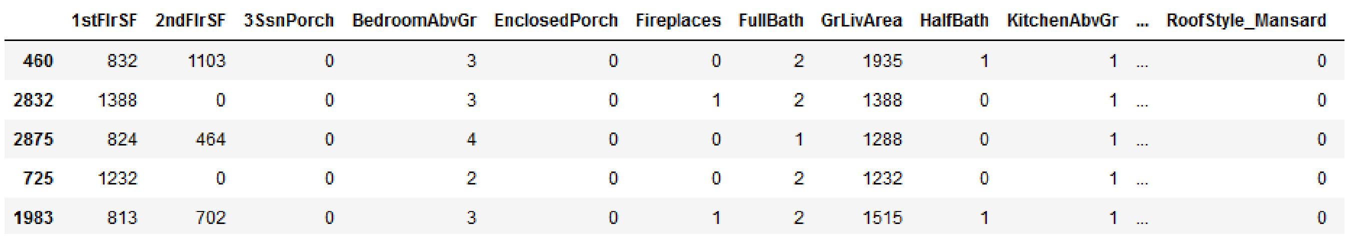
```
#One-hot encoding on categorical features
all_data = pd.get_dummies(all_data)
print(all_data.shape)
```

```
(2915, 201)
```



	1stFlrSF	2ndFlrSF	3SsnPorch	BedroomAbvGr	BldgType	BsmtCond	BsmtExposure	BsmtFinType1	BsmtFinType2	BsmtQual	...	RoofMatl	RoofStyle	SaleCo
0	856	854	0	3	1Fam	TA	No	GLQ	Unf	Gd	...	CompShg	Gable	
1	1262	0	0	3	1Fam	TA	Gd	ALQ	Unf	Gd	...	CompShg	Gable	
2	920	866	0	3	1Fam	TA	Mn	GLQ	Unf	Gd	...	CompShg	Gable	
3	961	756	0	3	1Fam	Gd	No	ALQ	Unf	TA	...	CompShg	Gable	A
4	1145	1053	0	4	1Fam	TA	Av	GLQ	Unf	Gd	...	CompShg	Gable	

Before conversion



	1stFlrSF	2ndFlrSF	3SsnPorch	BedroomAbvGr	EnclosedPorch	Fireplaces	FullBath	GrLivArea	HalfBath	KitchenAbvGr	...	RoofStyle_Mansard
460	832	1103	0	3	0	0	2	1935	1	1	...	0
2832	1388	0	0	3	0	1	2	1388	0	1	...	0
2875	824	464	0	4	0	0	1	1288	0	1	...	0
725	1232	0	0	2	0	0	2	1232	0	1	...	0
1983	813	702	0	3	0	1	2	1515	1	1	...	0

After conversion to numeric values

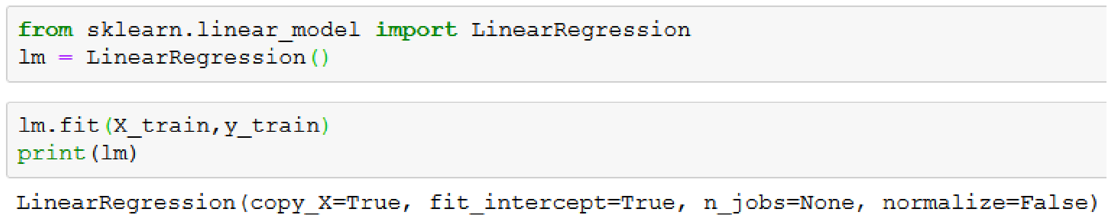
5. Methodology

We will use four prediction models for our analysis. The data is split into a 70:30 ratio for training and testing.

1) Linear Regression (Multiple Linear Regression):

Linear Regression is a statistical method that allows us to summarize and study relationships between continuous (quantitative) variables. The term “linear” in linear regression refers to the fact that the method models data with linear combination of the explanatory/predictor variables (attributes)

Fitting the model with the house prices data.

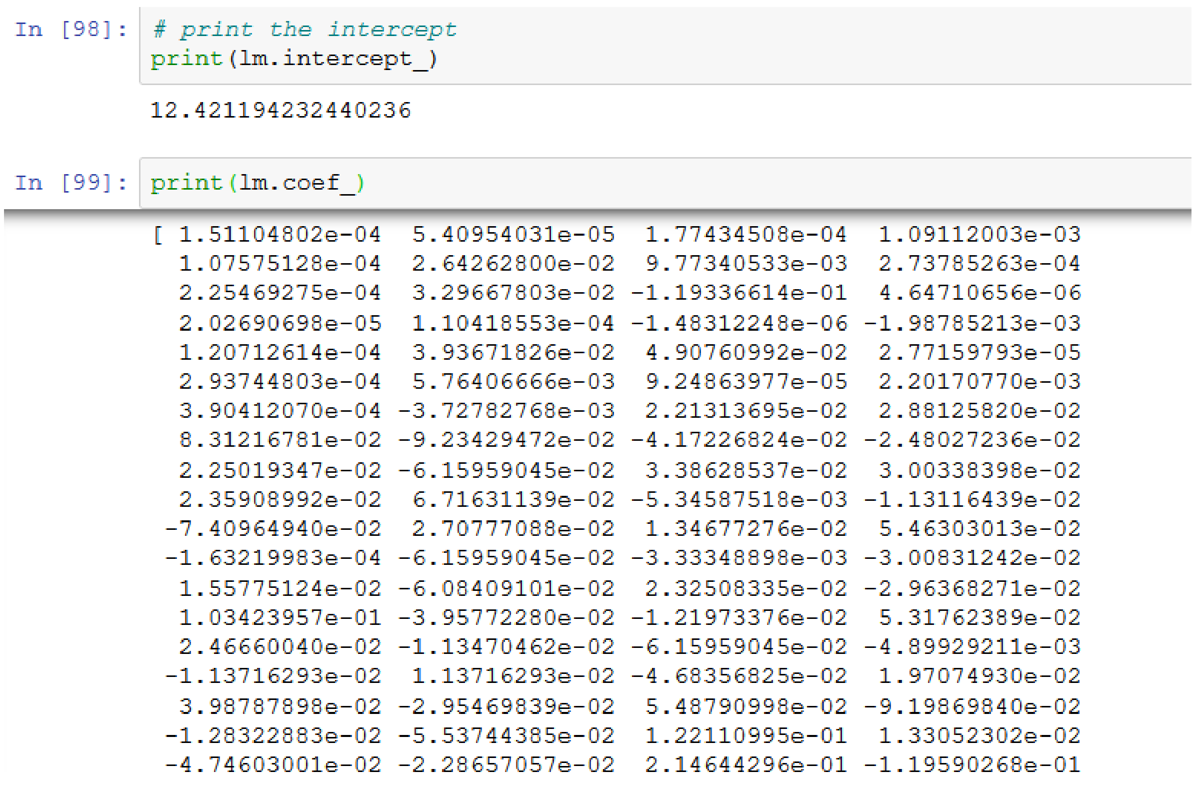


```
from sklearn.linear_model import LinearRegression
lm = LinearRegression()
```

```
lm.fit(X_train,y_train)
print(lm)
```

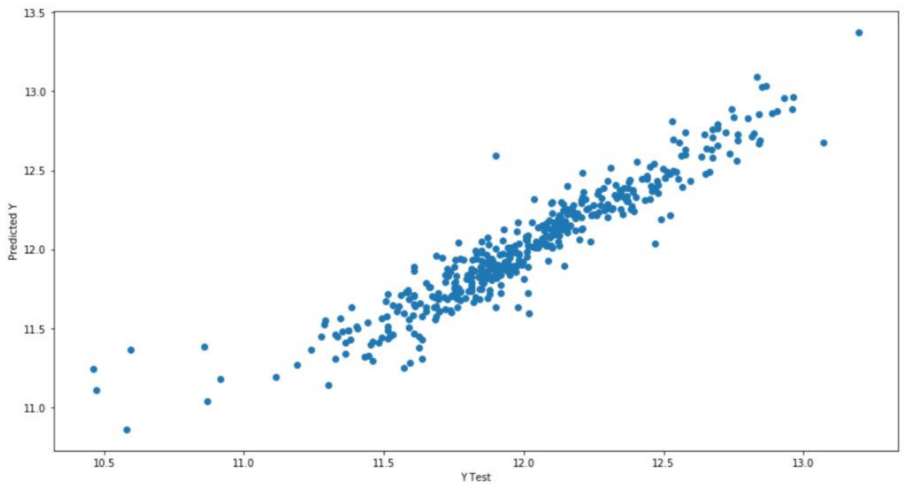
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

Getting the linear model coefficients:



```
In [98]: # print the intercept
         print(lm.intercept_)

         12.421194232440236

In [99]: print(lm.coef_)

         [ 1.51104802e-04  5.40954031e-05  1.77434508e-04  1.09112003e-03
           1.07575128e-04  2.64262800e-02  9.77340533e-03  2.73785263e-04
           2.25469275e-04  3.29667803e-02 -1.19336614e-01  4.64710656e-06
           2.02690698e-05  1.10418553e-04 -1.48312248e-06 -1.98785213e-03
           1.20712614e-04  3.93671826e-02  4.90760992e-02  2.77159793e-05
           2.93744803e-04  5.76406666e-03  9.24863977e-05  2.20170770e-03
           3.90412070e-04 -3.72782768e-03  2.21313695e-02  2.88125820e-02
           8.31216781e-02 -9.23429472e-02 -4.17226824e-02 -2.48027236e-02
           2.25019347e-02 -6.15959045e-02  3.38628537e-02  3.00338398e-02
           2.35908992e-02  6.71631139e-02 -5.34587518e-03 -1.13116439e-02
          -7.40964940e-02  2.70777088e-02  1.34677276e-02  5.46303013e-02
          -1.63219983e-04 -6.15959045e-02 -3.33348898e-03 -3.00831242e-02
           1.55775124e-02 -6.08409101e-02  2.32508335e-02 -2.96368271e-02
           1.03423957e-01 -3.95772280e-02 -1.21973376e-02  5.31762389e-02
           2.46660040e-02 -1.13470462e-02 -6.15959045e-02 -4.89929211e-03
          -1.13716293e-02  1.13716293e-02 -4.68356825e-02  1.97074930e-02
           3.98787898e-02 -2.95469839e-02  5.48790998e-02 -9.19869840e-02
          -1.28322883e-02 -5.53744385e-02  1.22110995e-01  1.33052302e-02
          -4.74603001e-02 -2.28657057e-02  2.14644296e-01 -1.19590268e-01
```

Plotting the predicted values with the actual values :

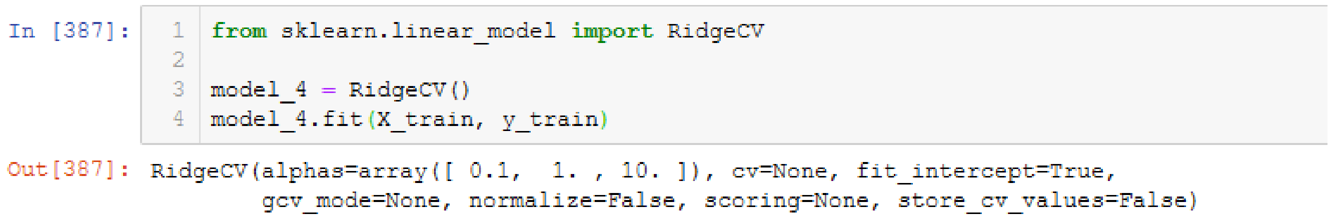


On plotting the predicted with the actual values we see that the datapoints aren't that dispersed from the diagonal and the predictions are fairly accurate.

2) Ridge Regression

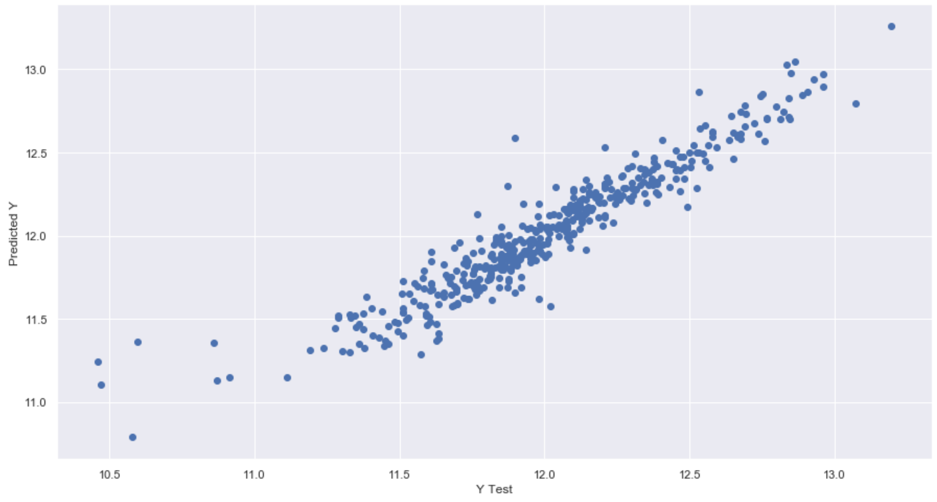
Ridge regression is a variation of linear regression specifically adapted for data that shows heavy multicollinearity. Ridge regression applies shrinking and is well-suited for datasets that have an abundant number of features which are not independent (collinearity) from one another.

Fitting the model with the data.



```
In [387]:  from sklearn.linear_model import RidgeCV

           model_4 = RidgeCV()
           model_4.fit(X_train, y_train)

Out[387]: RidgeCV(alphas=array([ 0.1,  1. , 10. ]), cv=None, fit_intercept=True,
              gcv_mode=None, normalize=False, scoring=None, store_cv_values=False)
```

Plotting the predicted values with the actual data:

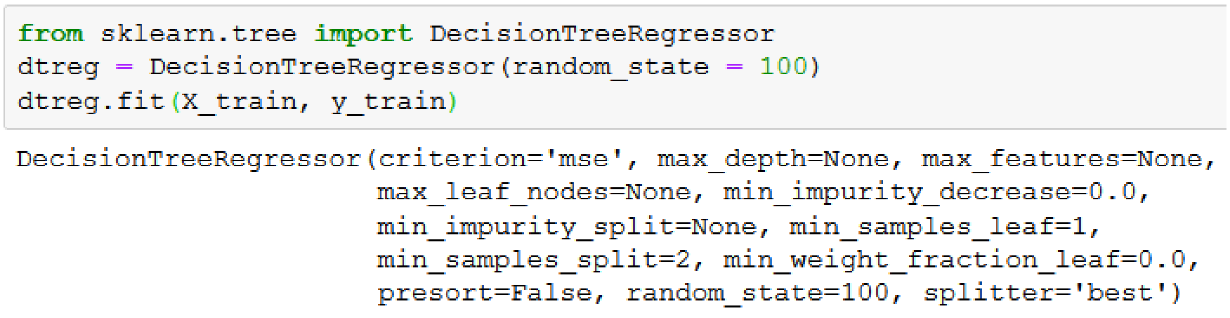


We see the datapoints are almost similar to the dispersion of points when compared to the linear regression model.

3) Decision Tree Regression

The decision tree is a simple machine learning model for getting started with regression tasks. A decision tree is a flow-chart-like structure, where each internal (non-leaf) node denotes a test on an attribute, each branch represents the outcome of a test, and each leaf (or terminal) node holds a class label. The topmost node in a tree is the root node.

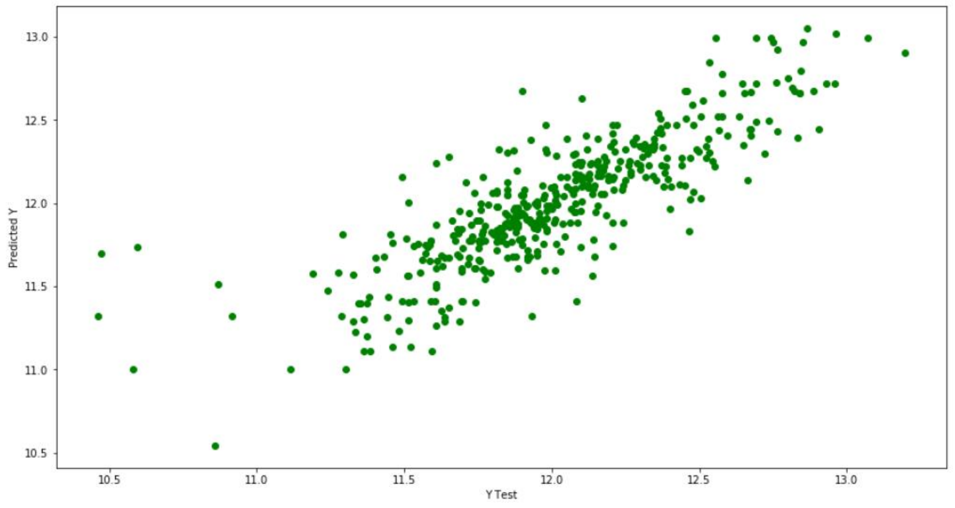
Fitting the data on the model:



```
from sklearn.tree import DecisionTreeRegressor
dtreg = DecisionTreeRegressor(random_state = 100)
dtreg.fit(X_train, y_train)
```

```
DecisionTreeRegressor(criterion='mse', 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=100, splitter='best')
```

Plotting the predicted values with the actual values :



An idea prediction would be a line that is diagonal where y=x, but here we see the datapoints to be scattered around the diagonal, which shows the predictions not to be that close to the actual values.

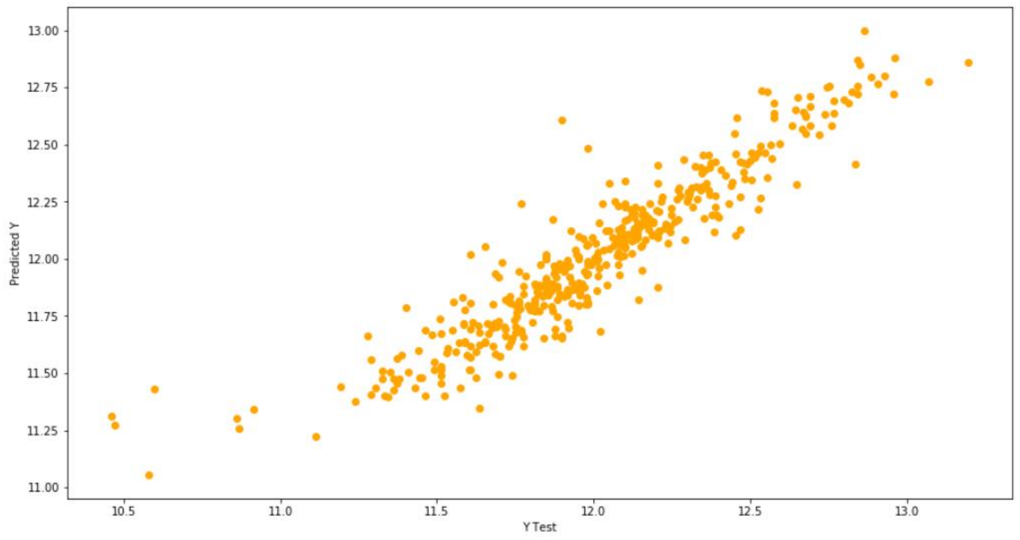
3) Random Forest

A Random Forest is an ensemble technique capable of performing both regression and classification tasks with the use of multiple decision trees and a technique called Bootstrap Aggregation, commonly known as bagging. Bagging, in the Random Forest method, involves training each decision tree on a different data sample where sampling is done with replacement.

```
from sklearn.ensemble import RandomForestRegressor
rfr = RandomForestRegressor(n_estimators = 100, random_state = 0)
rfr.fit(X_train, y_train)
```

```
RandomForestRegressor(bootstrap=True, criterion='mse', 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=100,
                      n_jobs=None, oob_score=False, random_state=0, verbose=0,
                      warm_start=False)
```

Plotting the predicted values with the actual values :



The predictions are pretty accurate as the most of the datapoints are concentrated around the diagonal.

Model Evaluation:

The performance of the models is evaluated through the following metrics:

1)**Mean Absolute Error (MAE)** is the mean of the absolute value of the errors:

$1n\sum i=1n|yi-\hat{y}\, i|$

2)**Mean Squared Error (MSE)** is the mean of the squared errors:

$1n\sum i=1n(yi-\hat{y}\, i)2$

3)**Root Mean Squared Error (RMSE)** is the square root of the mean of the squared errors:

$1n\sum i=1n(yi-\hat{y}\, i)$

From the above metrics the MAE is easiest to understand as it simply the average of errors. Whereas, the MSE can tend to punish the larger errors. RMSE is even more popular than MSE, because RMSE is interpretable in the "y" units.

For our models we see the below results:

	Linear Regression	**Ridge Regression**	**Decision Tree**	**Random Forest**
MAE	0.09493456881208923	0.09012912184873402	0.16563735105629293	0.10391893923108599
MSE	0.018724410351855573	0.01729965865099877	0.05334837517914882	0.02326594618183682
RMSE	0.13683716728964967	0.13152816675905876	0.23097267193144044	0.15253178744719678

By looking at the error rates we see the best performing model to be the Ridge Regression model closely followed by the Linear Regression model and the Random Forest model and the Decision Tree in the third and fourth places.

6. Conclusion

The objective of this was to build models to predict housing prices of different residences in Ames, out of which, the Ridge Regression model performed best, from the select number of features we chose to perform our analysis on.

However, the prediction can further be improved by doing a more thorough analysis and extensive feature engineering be performed on the variables that are most important in predicting SalePrice. This coupled with better models that can contribute to better interpretability (Eg. Ensemble models, gradient boosting models) and significantly improving the performance of the model on real-world data.

References:

Official Documentation of the dataset:

http://jse.amstat.org/v19n3/decock/DataDocumentation.txt

http://jse.amstat.org/v19n3/decock.pdf

Resources:

https://www.kaggle.com/c/house-prices-advanced-regression-techniques

https://realpython.com/linear-regression-in-python/#simple-linear-regression-with-scikit-learn

https://www.nobledesktop.com/learn/python/modality-skewness-and-kurtosis

https://towardsdatascience.com/understanding-gradient-boosting-machines-9be756fe76ab

https://medium.com/swlh/random-forest-and-its-implementation-71824ced454f

Appendix:

Code:

```
#Importing the necessary libraries
import pandas as pd
import numpy as np
import seaborn as sns

from scipy import stats
from scipy.stats import norm, skew #for some statistics

import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')
get_ipython().run_line_magic('matplotlib', 'inline')

train=pd.read_csv("C:/Users/astro/Desktop/Capstone Project -House Price Regression/Data/train.csv")
test=pd.read_csv("C:/Users/astro/Desktop/Capstone Project -House Price Regression/Data/test.csv")

train.shape

# List the number of columns
train.columns

train.sample(5)
# ### Analysing Salesprice

# Performing descriptive statistics on 'Salesprice'
train['SalePrice'].describe()
```

```
# This shows the median being lesser than the mean, which indicates the presence of outliers. Additionally, the average sale price of a house in our dataset is close to 180,000 USD with most of the values falling within the 130,000 USD to 215,000 USD range.


total_test = test.isnull().sum().sort_values(ascending=False)

percent_test = (test.isnull().sum()/test.isnull().count()).sort_values(ascending=False)

missing_data = pd.concat([total_test, percent_test], axis=1, keys=['Total', 'Percent'])

missing_data.head(25)


#Plotting the distribution of SalePrice

sns.distplot(train['SalePrice'])


#Skewness and kurtosis

print("Skewness: %f" % train['SalePrice'].skew())

print("Kurtosis: %f" % train['SalePrice'].kurt())


# <b>Interpretation:</b>

#

# 1.Deviates from the normal distribution by having a positive skew.

#

# 2.High kurtosis confirms peakedness.

#



#Correlation matrix


corrmat=train.corr()

f, ax = plt.subplots(figsize=(12, 12))

sns.heatmap(corrmat, vmax=.8, square=True);


# <b>Strong correlation can be seen in the following cases:</b>
```

```
#

# 1) We see a significant correlation between variables 'TotalBsmtSF' & '1stFlrSF' variables and also between the 'Garage' variables. This indicates multicollinearity present between these variables which will be removed later through feature engineering.

#

# 2) Another significant correlation we see is with the target variable 'SalePrice'. Strong correlations are seen with variables 'GrLivArea', 'TotalBsmtSF', and 'OverallQual'.

#


# Pairplot of the top 5 attributes with SalePrice

sns.set()

cols = ['SalePrice', 'OverallQual', 'GrLivArea', 'GarageCars', 'GarageArea','TotalBsmtSF']

sns.pairplot(train[cols], size = 2)

plt.show();

#Expand analysis on BsmtSF and Grlivarea & saleprice and yearbuilt scatter plot


# ### Categorical and numerical split


#check the numbers of samples and features

print("The train data size before dropping Id feature is : {} ".format(train.shape))

print("The test data size before dropping Id feature is : {} ".format(test.shape))


#Save the 'Id' column

train_ID = train['Id']

test_ID = test['Id']


#Now drop the  'Id' colum since it's unnecessary for  the prediction process.

train.drop("Id", axis = 1, inplace = True)

test.drop("Id", axis = 1, inplace = True)


#check again the data size after dropping the 'Id' variable
```

```
print("\nThe train data size after dropping Id feature is : {} ".format(train.shape))
print("The test data size after dropping Id feature is : {} ".format(test.shape))


# ### Outlier handling:
# Documentation mentions presence of an obvious outlier in the plot between GrLivArea and Saleprice
plt.subplots(figsize=(15, 5))


plt.subplot(1, 2, 1)
g = sns.regplot(x=train['GrLivArea'], y=train['SalePrice'], fit_reg=False).set_title("Before")


# Delete outliers
plt.subplot(1, 2, 2)
train = train.drop(train[(train['GrLivArea']>4000)].index)
g = sns.regplot(x=train['GrLivArea'], y=train['SalePrice'], fit_reg=False).set_title("After")


# ### Target Variable
sns.distplot(train['SalePrice'] , fit=norm);


# Get the fitted parameters used by the function
(mu, sigma) = norm.fit(train['SalePrice'])
print( '\n mu = {:.2f} and sigma = {:.2f}\n'.format(mu, sigma))


#Now plot the distribution
plt.legend(['Normal dist. ($\mu=$ {:.2f} and $\sigma=$ {:.2f} )'.format(mu, sigma)],
            loc='best')
plt.ylabel('Frequency')
plt.title('SalePrice distribution')


#
```

```
# The target variable is right skewed. As (linear) models love normally distributed data , we need to transform this variable and make it more normally distributed.

#


#Log transforming the target variable


#We use the numpy fuction log1p which  applies log(1+x) to all elements of the column

train["SalePrice"] = np.log1p(train["SalePrice"])


#Check the new distribution

sns.distplot(train['SalePrice'] , fit=norm);


# Get the fitted parameters used by the function

(mu, sigma) = norm.fit(train['SalePrice'])

print( '\n mu = {:.2f} and sigma = {:.2f}\n'.format(mu, sigma))


#Now plot the distribution

plt.legend(['Normal dist. ($\mu=$ {:.2f} and $\sigma=$ {:.2f} )'.format(mu, sigma)],

            loc='best')

plt.ylabel('Frequency')

plt.title('SalePrice distribution')



#### Feature Engineering:


x=train['SalePrice'].reset_index(drop=True)

x


#let's first concatenate the train and test data in the same dataframe
```

```
ntrain = train.shape[0]

ntest = test.shape[0]

y_train = train.SalePrice.values

all_data = pd.concat((train, test)).reset_index(drop=True)

all_data.drop(['SalePrice'], axis=1, inplace=True)

print("all_data size is : {}".format(all_data.shape))



total = all_data.isnull().sum().sort_values(ascending=False)

percent = ((all_data.isnull().sum()/all_data.isnull().count())*100).sort_values(ascending=False)

missing_data = pd.concat([total, percent], axis=1, keys=['Total', 'Missing Ratio'])

missing_data.head(25)



# As we can see above, the top four features seem to be filled with missing data(Almost 100%) as a result we
would be droppoing those attributes which have more than 80 percent of missing data.



# Dropping all those features where the missing data is greater than 80

all_data = all_data.drop((missing_data[missing_data['Missing Ratio'] > 15]).index,1)


# # Dropping all those features where the missing data is lesser than 1

# all_data = all_data.drop((missing_data[(missing_data['Missing Ratio'] >0) && (missing_data['Missing Ratio']
>0) ]).index,1)



all_data = all_data.drop((missing_data[(missing_data['Missing Ratio'] >0)&(missing_data['Missing Ratio'] <1)
]).index,1)


#Checking again
```

```
total = all_data.isnull().sum().sort_values(ascending=False)
percent = ((all_data.isnull().sum()/all_data.isnull().count())*100).sort_values(ascending=False)
missing_data = pd.concat([total, percent], axis=1, keys=['Total', 'Missing Ratio'])
missing_data.head(15)

# ### Handling missing data

#GarageType, GarageFinish, GarageQual and GarageCond : Replacing missing data with None
for col in ('GarageType', 'GarageFinish', 'GarageQual', 'GarageCond'):
    all_data[col] = all_data[col].fillna('None')

#GarageYrBlt: Replacing missing data with 0 (Since No garage = no cars in such garage.)

all_data['GarageYrBlt'] = all_data['GarageYrBlt'].fillna(0)



#BsmtQual, BsmtCond, BsmtExposure, BsmtFinType1 and BsmtFinType2 : For all these categorical
#basement-related features, NaN means that there is no basement.
for col in ('BsmtQual', 'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFinType2'):
    all_data[col] = all_data[col].fillna('None')



# Check for any missing values
total = all_data.isnull().sum().sort_values(ascending=False)
percent = (all_data.isnull().sum()/all_data.isnull().count()).sort_values(ascending=False)
missing_data = pd.concat([total, percent], axis=1, keys=['Total', 'Missing Ratio'])
missing_data.head(5)



all_data.columns
all_data.shape

# Getting dummy categorical features
#
all_data.shape



all_data.head()

all_data = pd.get_dummies(all_data)
print(all_data.shape)

aha=all_data
del aha['GarageYrBlt']
aha.sample(5)

all_data.columns

all_data.sample(5)



train = all_data[:ntrain]
test = all_data[ntrain:]

train['SalePrice']=x

train.head()
```

```
# ### Models:


# ###  Linear Regression


# Linear Regression is a statistical method that allows us to summarize and study relationships between
continuous (quantitative) variables. The term “linear” in linear regression refers to the fact that the method models
data with linear combination of the explanatory/predictor variables (attributes)


from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(train.drop('SalePrice', axis=1),train['SalePrice'], test_size=0.3,
random_state=101)


y_train.head()


X_train.shape


from sklearn.linear_model import LinearRegression

lm = LinearRegression()


lm.fit(X_train,y_train)

print(lm)


# Model Evaluation

# print the intercept

print(lm.intercept_)


print(lm.coef_)


# Predictions:


predictions = lm.predict(X_test)
```

```
predictions= predictions.reshape(-1,1)


plt.figure(figsize=(15,8))
plt.scatter(y_test,predictions)
plt.xlabel('Y Test')
plt.ylabel('Predicted Y')
plt.show()


from sklearn import metrics
print('MAE:', metrics.mean_absolute_error(y_test, predictions))
print('MSE:', metrics.mean_squared_error(y_test, predictions))
print('RMSE:', np.sqrt(metrics.mean_squared_error(y_test, predictions)))


# ### Ridge Regression


# Ridge regression is a variation of linear regression specifically adapted for data that shows heavy multicollinearity. Ridge regression applies shrinking and is well-suited for datasets that have an abundant number of features which are not independent (collinearity) from one another.
#
from sklearn.linear_model import RidgeCV


model_4 = RidgeCV()
model_4.fit(X_train, y_train)


# Prediction


y_pred_4 = model_4.predict(X_test)
plt.figure(figsize=(15,8))
plt.scatter(y_test,y_pred_4)
plt.xlabel('Y Test')
plt.ylabel('Predicted Y')
plt.show()

#Evaluating the Model

print('Mean Absolute Error:', metrics.mean_absolute_error(y_test, y_pred_4))
print('Mean Squared Error:', metrics.mean_squared_error(y_test, y_pred_4))
print('Root Mean Squared Error:', np.sqrt(metrics.mean_squared_error(y_test, y_pred_4)))

# ### Decision Tree Regression
from sklearn.tree import DecisionTreeRegressor
dtreg = DecisionTreeRegressor(random_state = 100)
dtreg.fit(X_train, y_train)



from sklearn.tree import DecisionTreeRegressor
dtreg = DecisionTreeRegressor(random_state = 100)
dtreg.fit(X_train, y_train)

print('MAE:', metrics.mean_absolute_error(y_test, dtr_pred))
print('MSE:', metrics.mean_squared_error(y_test, dtr_pred))
print('RMSE:', np.sqrt(metrics.mean_squared_error(y_test, dtr_pred)))



plt.figure(figsize=(15,8))
plt.scatter(y_test,dtr_pred,c='green')
plt.xlabel('Y Test')
plt.ylabel('Predicted Y')
plt.show()

# ###  Random Forest
```

```
from sklearn.ensemble import RandomForestRegressor
rfr = RandomForestRegressor(n_estimators = 100, random_state = 0)
rfr.fit(X_train, y_train)

rfr_pred= rfr.predict(X_test)
rfr_pred = rfr_pred.reshape(-1,1)



print('MAE:', metrics.mean_absolute_error(y_test, rfr_pred))
print('MSE:', metrics.mean_squared_error(y_test, rfr_pred))
print('RMSE:', np.sqrt(metrics.mean_squared_error(y_test, rfr_pred)))

plt.figure(figsize=(15,8))
plt.scatter(y_test,rfr_pred, c='orange')
plt.xlabel('Y Test')
plt.ylabel('Predicted Y')
plt.show()

#plot the error among the different models
error_rate=np.array([metrics.mean_squared_error(y_test, predictions),metrics.mean_squared_error(y_test,
y_pred_4),metrics.mean_squared_error(y_test, dtr_pred),metrics.mean_squared_error(y_test, rfr_pred)])



plt.figure(figsize=(16,5))
plt.plot(error_rate)
```