HOUSING PRICE PREDICTION PROJECT

NAME OF THE PROJECT

Submitted by:

Kartik Khodke

House Price Prediction Projects

Table of Contents

- Literature Review
- Data Preparation
- Exploratory Data Analysis
- Prediction Type and Modeling Techniques
- Model Building and Evaluation
- Analysis and Comparison
- Conclusion
- References

Introduction:

Thousands of houses are sold everyday. There are some questions every buyer asks himself like: What is the actual price that this house deserves? Am I paying a fair price? In this paper, a, the year it was built in, etc.). During the development and evaluation of our model, we will show the code machine learning model is proposed to predict a house price based on data related to the house (its size used for each step followed by its output. This will facilitate the reproducibility of our work. In this study, Python programming language with a number of Python packages will be used.

Problem Statement:

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 data is provided in the CSV file
below.

The company is looking at prospective properties to buy houses to enter the market. You 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?

Business Goal:

You are required 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.

Technical Requirements:

- Data contains 1460 entries each having 81 variables.
- Data contains Null values. You need to treat them using the domain knowledge and your own understanding.
- Extensive EDA has to be performed to gain relationships of important variable and price.
- Data contains numerical as well as categorical variable. You need to handle them accordingly.
- You have to build Machine Learning models, apply regularization and determine the optimal values of Hyper Parameters.
- You need to find important features which affect the price positively or negatively.
- Two datasets are being provided to you (test.csv, train.csv). You will train on train.csv dataset and predict on test.csv file.

Breakdown of the Problem Statement:

- Supervised machine learning problem.
- Linear Regression.
- The target value will be SalePrice.

Goals of the Study

The main objectives of this study are as follows:

- To apply data preprocessing and preparation techniques in order to obtain clean data
- To build machine learning models able to predict house price based on house features
- To analyze and compare models performance in order to choose the best model

Literature Review

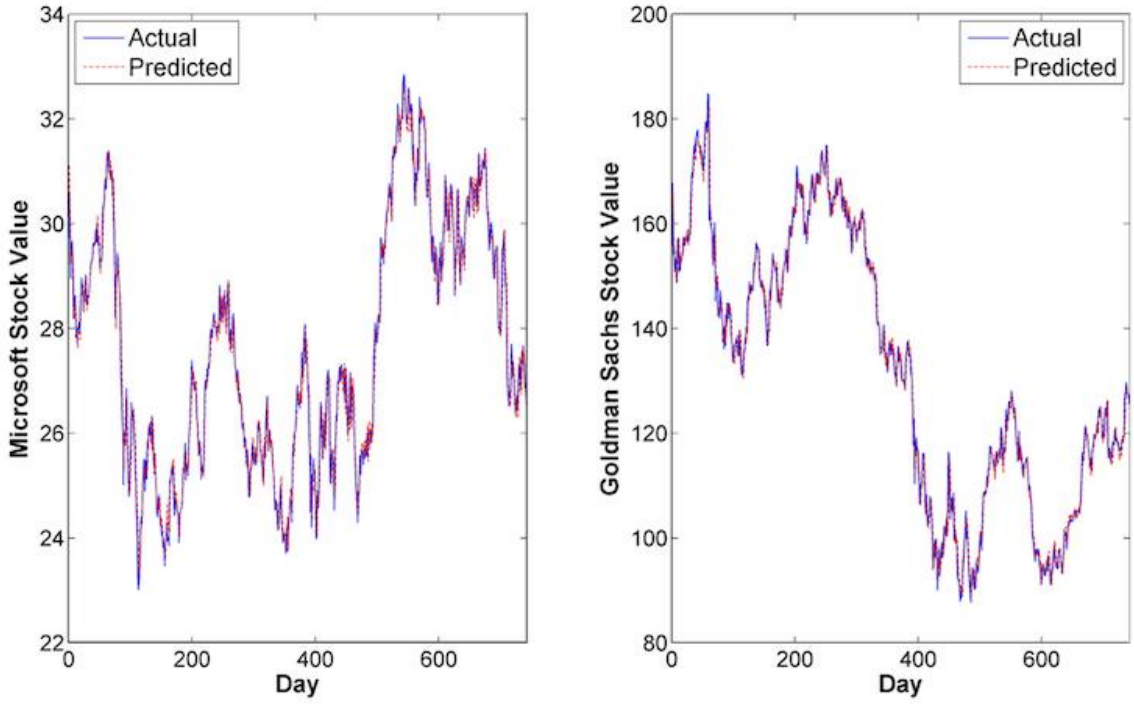
In this section, we look at five recent studies that are related to our topic and see how models were built and what results were achieved in these studies.

Stock Market Prediction Using Bayesian-Regularized Neural Networks

In a study done by Ticknor (2013), he used Bayesian regularized artificial neural network to predict the future operation of financial market. Specifically, he built a model to predict future stock prices. The input of the model is previous stock statistics in addition to some financial technical data. The output of the model is the next-day closing price of the corresponding stocks.
The model proposed in the study is built using Bayesian regularized neural network. The weights of this type of networks are given a probabilistic nature. This allows the network to penalize very complex models (with many hidden layers) in an automatic manner. This in turn will reduce the overfitting of the model.
The model consists of a feedforward neural network which has three layers: an input layer, one hidden layer, and an output layer. The author chose the number of neurons in the hidden layer based on experimental methods.The input data of the model is normalized to be between -1 and 1, and this opertion is reversed for the output so the predicted price appears in the appropriate scale.
The data that was used in this study was obtained from Goldman Sachs Group (GS), Inc. and Microsoft Corp. (MSFT) . The data covers 734 trading days (4 January 2010 to 31 December 2012). Each instance of the data consisted of daily statistics: low price, high price, opening price, close price, and trading volume. To facilitate the training and testing of the model, this data was split into training data and test data with 80% and 20% of the original data, respectively. In addition to the daily-statistics variables in the data, six more variables were created to reflect financial indicators.
The performance of the model were evaluated using mean absolute percentage error (MAPE) performance metric. MAPE was calculated using this formula

$$MAPE=\sum_{ri=1}(abs(y_i-p_i)/y_i)r\times 100$$

where p_i is the predicted stock price on day i, y_i is the actual stock price on day i, and r is the number of trading days.
When applied on the test data, The model achieved a MAPE score of `1.0561` for MSFT part, and `1.3291` for GS part. Figure 1 shows the actual values and predicted values for both GS and MSFT data.

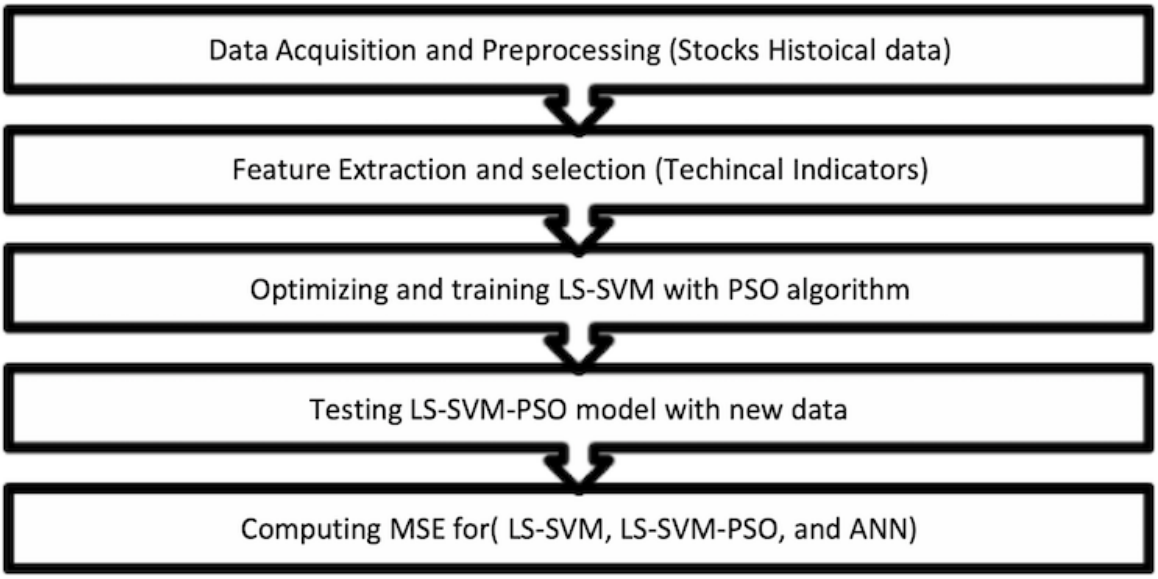


Stock Market Prediction Using A Machine Learning Model

another study done by Hegazy, Soliman, and Salam (2014), a system was proposed to predict daily stock market prices. The system combines particle swarm optimization (PSO) and least square support vector machine (LS-SVM), where PSO was used to optimize LV-SVM.

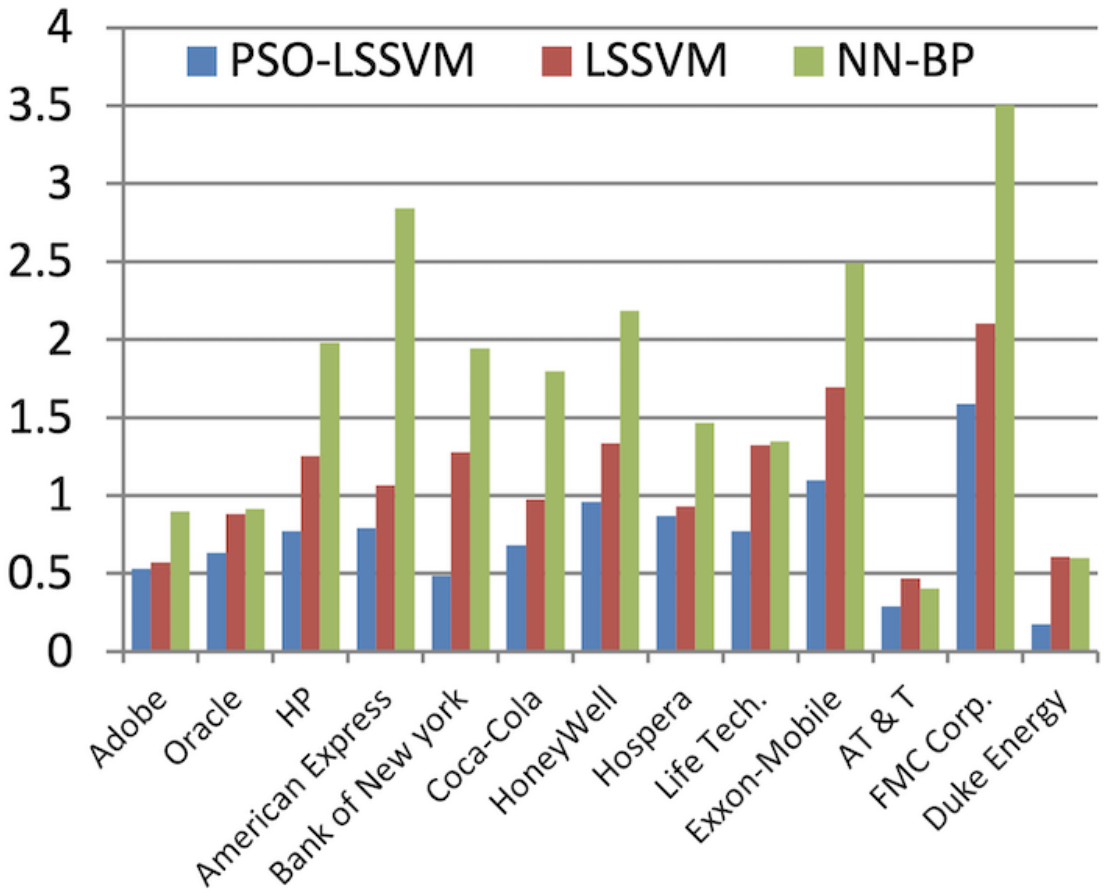
The authors claim that in most cases, artificial neural networks (ANNs) are subject to the overfitting problem. They state that support vector machines algorithm (SVM) was developed as an alternative that doesn't suffer from overfitting. They attribute this advantage to SVMs being based on the solid foundations of VC-theory. They further elaborate that LS-SVM method was reformulation of traditional SVM method that uses a regularized least squares function with equality constraints to obtain a linear system that satisfies Karush-Kuhn-Tucker conditions for getting an optimal solution.

The authors describe PSO as a popular evolutionary optimization method that was inspired by organism social behavior like bird flocking. They used it to find the optimal parameters for LS-SVM. These parameters are the cost penalty CC, kernel parameter γγ, and insensitive loss function єє.
The model proposed in the study was based on the analysis of historical data and technical financial indicators and using LS-SVM optimized by PSO to predict future daily stock prices. The model input was six vectors representing the historical data and the technical financial indicators. The model output was the future price. The model used is represented in Figure 2.



Regarding the technical financial indicators, five were derived from the raw data: relative strength index (RSI), money flow index (MFI), exponential moving average (EMA), stochastic oscillator (SO), and moving average convergence/divergence (MACD). These indicators are known in the domain of stock market.

The model was trained and tested using datasets taken from https://finance.yahoo.com/. The datasets were from Jan 2009 to Jan 2012 and include stock data for many companies like Adobe and HP. All datasets were partitioned into a training set with 70% of the data and a test set with 30% of the data. Three models were trained and tested: LS-SVM-PSO model, LS-SVM model, and ANN model. The results obtained in the study showed that LS-SVM-PSO model had the best performance. Figure 3 shows a comparison between the mean square error (MSE) of the three models for the stocks of many companies.



House Price Prediction Using Multilevel Model and Neural Networks

The multilevel model integrates the micro-level that specifies the relationships between houses within a given neighbourhood, and the macro-level equation which specifies the relationships between neighbouhoods. The hedonic price model is a model that estimates house prices using some attributes such as the number of bedrooms in the house, the size of the house, etc.

The data used in the study contains house prices in Greater Bristol area between 2001 and 2013. Secondary data was obtained from the Land Registry, the Population Census and Neighbourhood Statistics to be used in order to make the models suitable for national usage. The authors listed many reasons on why they chose the Greater Bristol area such as its diverse urban and rural blend and its different property types. Each record in the dataset contains data about a house in the area: it contains the address, the unit postcode, property type, the duration (freehold or leasehold), the sale price, the date of the sale, and whether the house was newly-built when it was sold. In total, the dataset contains around $_{65,000}$ entries. To enable model training and testing, the dataset was divided into a training set that contains data about house sales from 2001 to 2012, and a test set that contains data about house sales in 2013.

The three models (MLM, ANN, and HPM) were tested using three senarios. In the first senario, locational and measured neighbourhood attributes were not included in the data. In the second senario, grid references of house location were included in the data. In the third senario, measured neighbourhood attributes were included in the data. The models were compared in goodness of fit where R_2R_2 was the metric, predictive accuracy where mean absolute error (MAE) and mean absolute percentage error (MAPE) were the metrics, and explanatory power. HPM and MLM models were fitted using MLwiN software, and ANN were fitted using IBM SPSS software. Figure 4 shows the performance of each model regarding fit goodness and predictive accuracy. It shows that MLM model has better performance in general than other models.

COMPARISONS OF GOODNESS-OF-FIT

	R^2 (training set)	R^2 (test set)
HPM1	0.39	0.23
MLM1	0.75	0.75
ANN1	0.39	0.23
HPM2	0.43	0.3
MLM2	0.75	0.75
ANN2	0.41	0.26
HPM3	0.68	0.65
MLM3	0.75	0.74
ANN3	0.69	0.67

COMPARISON OF PREDICTIVE ACCURACY

Test set	MAE (lnP)	MAPE (lnP)	MAE (raw price)	MAPE (raw price)
HPM1	0.319	5.89%	80.4	30.9%
MLM1	0.178	3.29%	48.6	17.5%
ANN1	0.318	5.85%	80.1	30.0%
HPM2	0.304	5.61%	77.0	29.4%
MLM2	0.178	3.29%	48.6	17.5%
ANN2	0.313	5.76%	79.0	29.8%
HPM3	0.210	3.89%	25.3	20.7%
MLM3	0.178	3.30%	48.8	17.6%
ANN3	0.216	4.00%	55.7	20.9%

Composition of Models and Feature Engineering to Win Algorithmic Trading Challenge

A study done by de Abril and Sugiyama (2013) introduced the techniques and ideas used to win Algorithmic Trading Challenge, a competition held on Kaggle. The goal of the competition was to develop a model that can predict the short-term response of order-driven markets after a big liquidity shock. A liquidity shock happens when a trade or a sequence of trades causes an acute shortage of liquidity (cash for example).

The challenge data contains a training dataset and a test dataset. The training dataset has around $754{,}000$ records of trade and quote observations for many securities of London Stock Exchange before and after a liquidity shock. A trade event happens when shares are sold or bought, whereas a quote event happens when the ask price or the best bid changes.

A separate model was built for bid and another for ask. Each one of these models consists of κ random-forest sub-models. The models predict the price at a particular future time.

The authors spent much effort on feature engineering. They created more than 150 features. These features belong to four categories: price features, liquidity-book features, spread features (bid/ask

spread), and rate features (arrival rate of orders/quotes). They applied a feature selection algorithm to obtain the optimal feature set (F_bF_b) for bid sub-models and the optimal feature set (F_aF_a) of all ask sub-models. The algorithm applied eliminates features in a backward manner in order to get a feature set with reasonable computing time and resources.

Three instances of the final model proposed in the study were trained on three datasets; each one of them consists of $50{,}000$ samples sampled randomly from the training dataset. Then, the three models were applied to the test dataset. The predictions of the three models were then averaged to obtain the final prediction. The proposed method achieved a RMSE score of 0.77 approximately.

Using K-Nearest Neighbours for Stock Price Prediction

Alkhatib, Najadat, Hmeidi, and Shatnawi (2013) have done a study where they used the k-nearest neighbours (KNN) algorithm to predict stock prices. In this study, they expressed the stock prediction problem as a similarity-based classification, and they represented the historical stock data as well as test data by vectors.

·The authors listed the steps of predicting the closing price of stock market using KNN as follows:

.The number of neaerest neighbours is chosen

.The distance between the new record and the training data is computed

.Training data is sorted according to the calculated distance

.Majority voting is applied to the classes of the k nearest neighbours to determine the predicted value of the new record

The data used in the study is stock data of five companies listed on the Jordanian stock exchange. The data range is from 4 June 2009 to 24 December 2009. Each of the five companies has around 200 records in the data. Each record has three variables: closing price, low price, and high price. The author stated that the closing price is the most important feature in determining the prediction value of a stock using KNN.

After applying KNN algorithm, the authors summarized the prediction performance evaluation using different metrics in a the table shown in Figure 5.

	kNN algorithm for K = 5		
Company	**Total squared RMS error**	**RMS error**	**Average error**
AIEI	0.263151	0.0378176	-5.43E-09
AFIN	0.2629177	0.0363482	-1.01E-08
APOT	22.74533	0.3372338	2.50E-08
IREL	1.2823397	0.1046908	4.27E-08
JOST	0.17963	0.0300444	1.508E-08

The authors used lift charts also to evaluate the performance of their model. Lift chart shows the improvement obtained by using the model compared to random estimation. As an example, the lift graph for AIEI company is shown in Figure 6. The area between the two lines in the graph is an indicator of the goodness of the model.

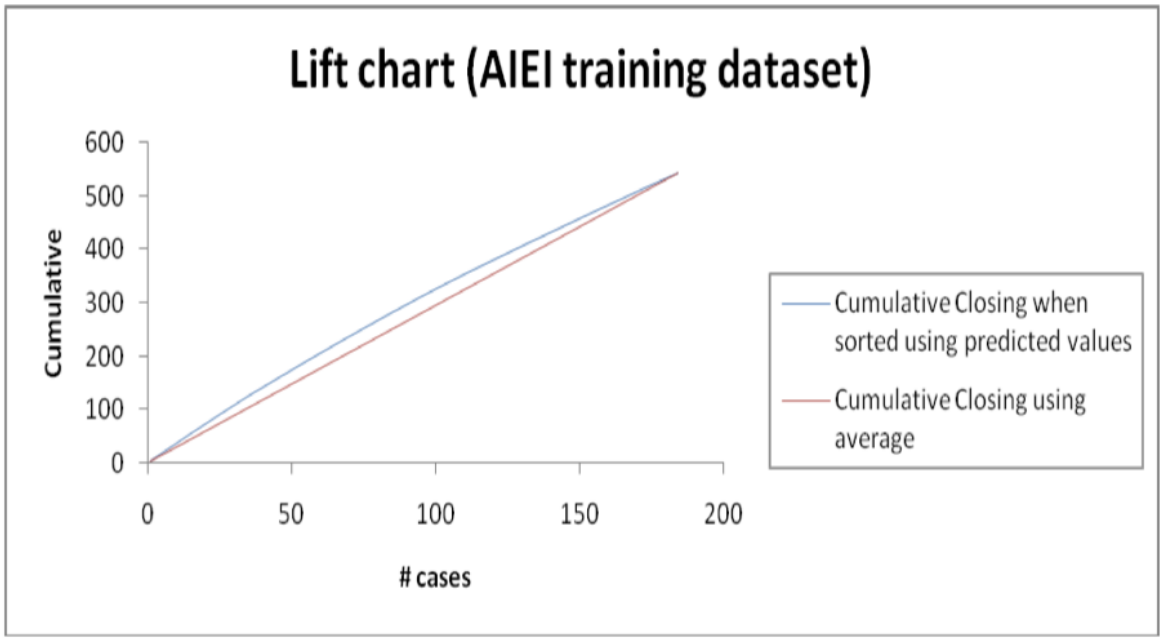
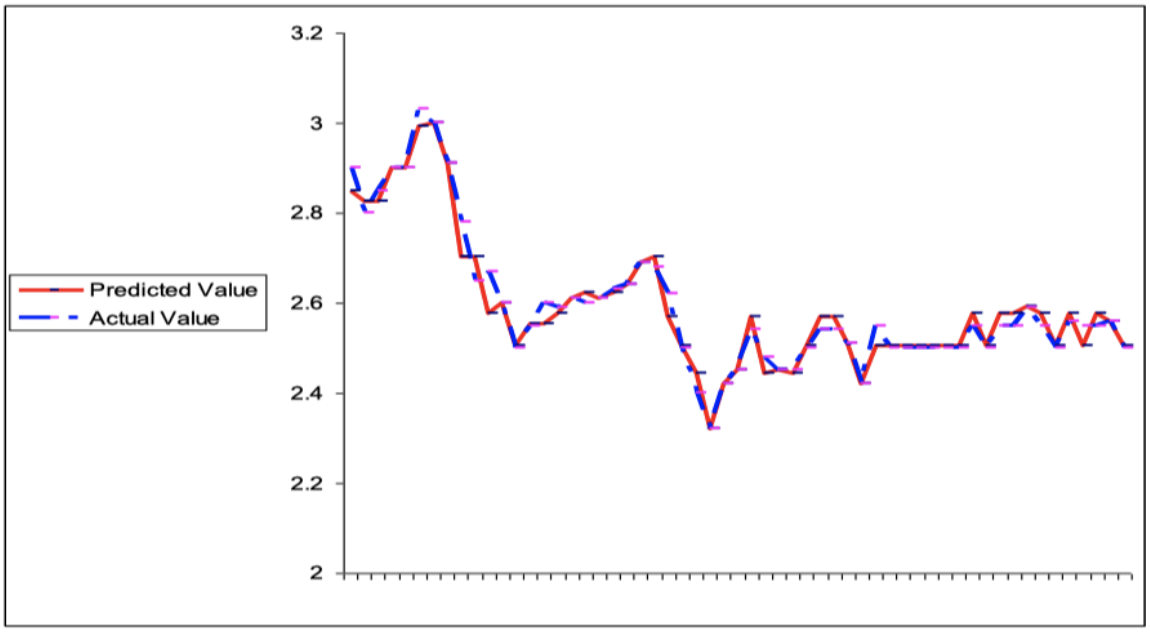


Figure 7 shows the relationship between the actual price and predicted price for one year for the same company.



Data Preparation

In this study, we will use a housing dataset presented by De Cock (2011). This dataset describes the sales of residential units in Ames, Iowa starting from 2006 until 2010. The dataset contains a large number of variables that are involved in determining a house price. We obtained a csv copy of the data from https://www.kaggle.com/prevek18/ames-housing-dataset.

Data Description

The dataset contains 1160 records (rows) and 81 features (columns).

Here, we will provide a brief description of dataset features. Since the number of features is large (81), we will attach the original data description file to this paper for more information about the dataset (It can be downloaded also from https://www.kaggle.com/c/house-prices-advanced-regression-techniques/data). Now, we will mention the feature name with a short description of its meaning.

Feature	Description
MSSubClass	The type of the house involved in the sale
MSZoning	The general zoning classification of the sale
LotFrontage	Linear feet of street connected to the house
LotArea	Lot size in square feet
Street	Type of road access to the house
Alley	Type of alley access to the house
LotShape	General shape of the house
LandContour	House flatness
Utilities	Type of utilities available
LotConfig	Lot configuration

LandSlope	House Slope
Neighborhood	Locations within Ames city limits
Condition1	Proximity to various conditions
Condition2	Proximity to various conditions (if more than one is present)
BldgType	House type
HouseStyle	House style
OverallQual	Overall quality of material and finish of the house
OverallCond	Overall condition of the house
YearBuilt	Construction year
YearRemodAdd	Remodel year (if no remodeling nor addition, same as YearBuilt)
RoofStyle	Roof type
RoofMatl	Roof material
Exterior1st	Exterior covering on house
Exterior2nd	Exterior covering on house (if more than one material)
MasVnrType	Type of masonry veneer
MasVnrArea	Masonry veneer area in square feet
ExterQual	Quality of the material on the exterior
ExterCond	Condition of the material on the exterior
Foundation	Foundation type

BsmtQual	Basement height
BsmtCond	Basement Condition
BsmtExposure	Refers to walkout or garden level walls
BsmtFinType1	Rating of basement finished area
BsmtFinSF1	Type 1 finished square feet
BsmtFinType2	Rating of basement finished area (if multiple types)
BsmtFinSF2	Type 2 finished square feet
BsmtUnfSF	Unfinished basement area in square feet
TotalBsmtSF	Total basement area in square feet
Heating	Heating type
HeatingQC	Heating quality and condition
CentralAir	Central air conditioning
Electrical	Electrical system type
1stFlrSF	First floor area in square feet
2ndFlrSF	Second floor area in square feet
LowQualFinSF	Low quality finished square feet in all floors
GrLivArea	Above-ground living area in square feet
BsmtFullBath	Basement full bathrooms
BsmtHalfBath	Basement half bathrooms

FullBath	Full bathrooms above ground
HalfBath	Half bathrooms above ground
Bedroom	Bedrooms above ground
Kitchen	Kitchens above ground
KitchenQual	Kitchen quality
TotRmsAbvGrd	Total rooms above ground (excluding bathrooms)
Functional	Home functionality
Fireplaces	Number of fireplaces
FireplaceQu	Fireplace quality
GarageType	Garage location
GarageYrBlt	Year garage was built in
GarageFinish	Interior finish of the garage
GarageCars	Size of garage (in car capacity)
GarageArea	Garage size in square feet
GarageQual	Garage quality
GarageCond	Garage condition
PavedDrive	How driveway is paved
WoodDeckSF	Wood deck area in square feet
OpenPorchSF	Open porch area in square feet

EnclosedPorch	Enclosed porch area in square feet
3SsnPorch	Three season porch area in square feet
ScreenPorch	Screen porch area in square feet
PoolArea	Pool area in square feet
PoolQC	Pool quality
Fence	Fence quality
MiscFeature	Miscellaneous feature
MiscVal	Value of miscellaneous feature
MoSold	Sale month
YrSold	Sale year
SaleType	Sale type
SaleCondition	Sale condition

Reading the Dataset

The first step is reading the dataset from the csv file we downloaded. We will use the `read_csv()` function from `Pandas` Python package:

```
import pandas as pd

df=pd.read_csv("train.csv")

df.head()
```

Getting A Feel of the Dataset

Let's display the first few rows of the dataset to get a feel of it:

```
df=pd.read_csv("train.csv")
df.head()
```

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Utilities	...	PoolArea	PoolQC	Fence	MiscFeature	MiscVal	Mo
0	127	120	RL	NaN	4928	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	NaN	NaN	0	
1	889	20	RL	95.0	15865	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	NaN	NaN	0	
2	793	60	RL	92.0	9920	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	NaN	NaN	0	
3	110	20	RL	105.0	11751	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	MnPrv	NaN	0	
4	422	20	RL	NaN	16635	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	NaN	NaN	0	

5 rows × 81 columns

Same way read the test data also

```
test_df=pd.read_csv("test.csv")
test_df.head()
```

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Utilities	...	ScreenPorch	PoolArea	PoolQC	Fence	MiscFeatu
0	337	20	RL	86.0	14157	Pave	NaN	IR1	HLS	AllPub	...	0	0	NaN	NaN	Na
1	1018	120	RL	NaN	5814	Pave	NaN	IR1	Lvl	AllPub	...	0	0	NaN	NaN	Na
2	929	20	RL	NaN	11838	Pave	NaN	Reg	Lvl	AllPub	...	0	0	NaN	NaN	Na
3	1148	70	RL	75.0	12000	Pave	NaN	Reg	Bnk	AllPub	...	0	0	NaN	NaN	Na
4	1227	60	RL	86.0	14598	Pave	NaN	IR1	Lvl	AllPub	...	0	0	NaN	NaN	Na

5 rows × 80 columns

```
test_df.shape
```

```
(292, 80)
```

Exploratory data analysis:

Exploratory Data Analysis or EDA is very crucial for the success of all data science projects. It is an approach to analyze and understand the various aspects of the data.

Check dtypes of train data:

```
df.dtypes
```

```
Id                int64
MSSubClass        int64
MSZoning         object
LotFrontage     float64
LotArea           int64
                 ...
MoSold            int64
YrSold            int64
SaleType         object
SaleCondition    object
SalePrice         int64
Length: 81, dtype: object
```

Check the dtypes of test data

```
test_df.dtypes
```

```
Id                int64
MSSubClass        int64
MSZoning         object
LotFrontage     float64
LotArea           int64
                 ...
MiscVal           int64
MoSold            int64
YrSold            int64
SaleType         object
SaleCondition    object
Length: 80, dtype: object
```

Check nunique values of train data

```
df.nunique()
```

```
Id               1168
MSSubClass         15
MSZoning            5
LotFrontage       106
LotArea           892
                  ...
MoSold             12
YrSold              5
SaleType            9
SaleCondition       6
SalePrice         581
Length: 81, dtype: int64
```

Check the unique value of test data

```
test_df.nunique()
```

```
Id               292
MSSubClass        15
MSZoning           4
LotFrontage       65
LotArea          249
                 ...
MiscVal            8
MoSold            12
YrSold             5
SaleType           6
SaleCondition      4
Length: 80, dtype: int64
```

Check the null value counts of train data

```
df.isnull().sum()
```

```
Id                0
MSSubClass        0
MSZoning          0
LotFrontage     214
LotArea           0
                 ...
MoSold            0
YrSold            0
SaleType          0
SaleCondition     0
SalePrice         0
Length: 81, dtype: int64
```

Check the null value counts of test data

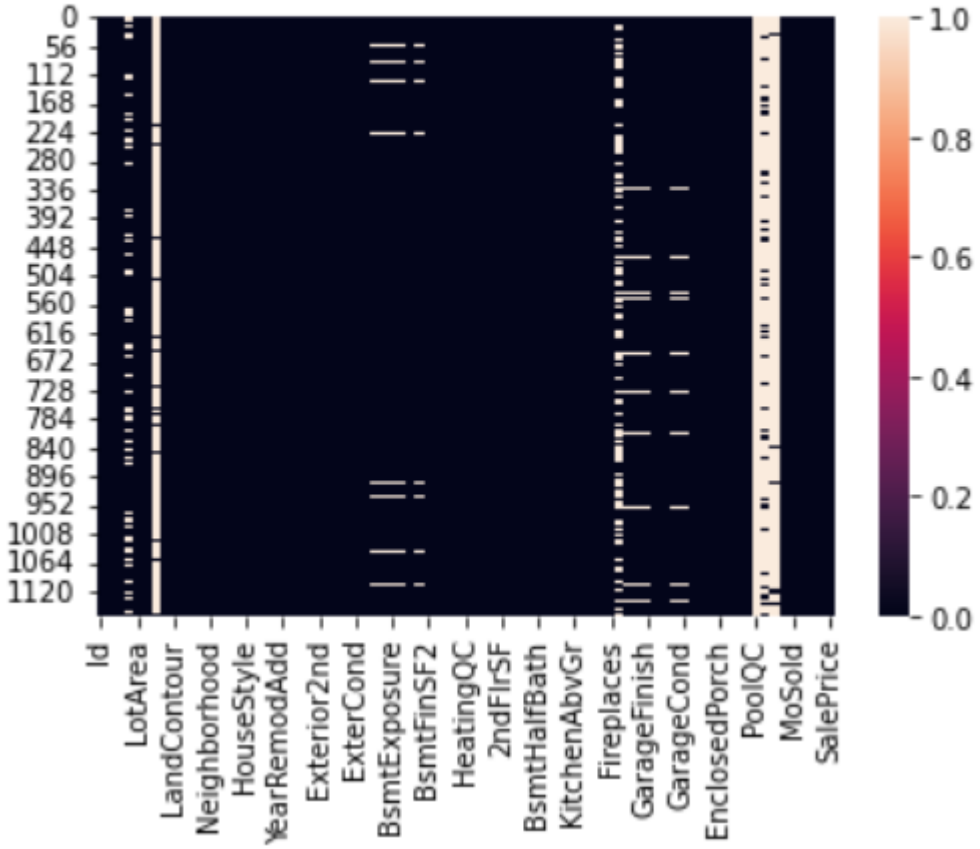
```
test_df.isnull().sum()
```

```
Id                0
MSSubClass        0
MSZoning          0
LotFrontage      45
LotArea           0
                 ..
MiscVal           0
MoSold            0
YrSold            0
SaleType          0
SaleCondition     0
Length: 80, dtype: int64
```

Check with the help of heatmap null value of train data

```
import seaborn as sns
sns.heatmap(df.isnull())
```

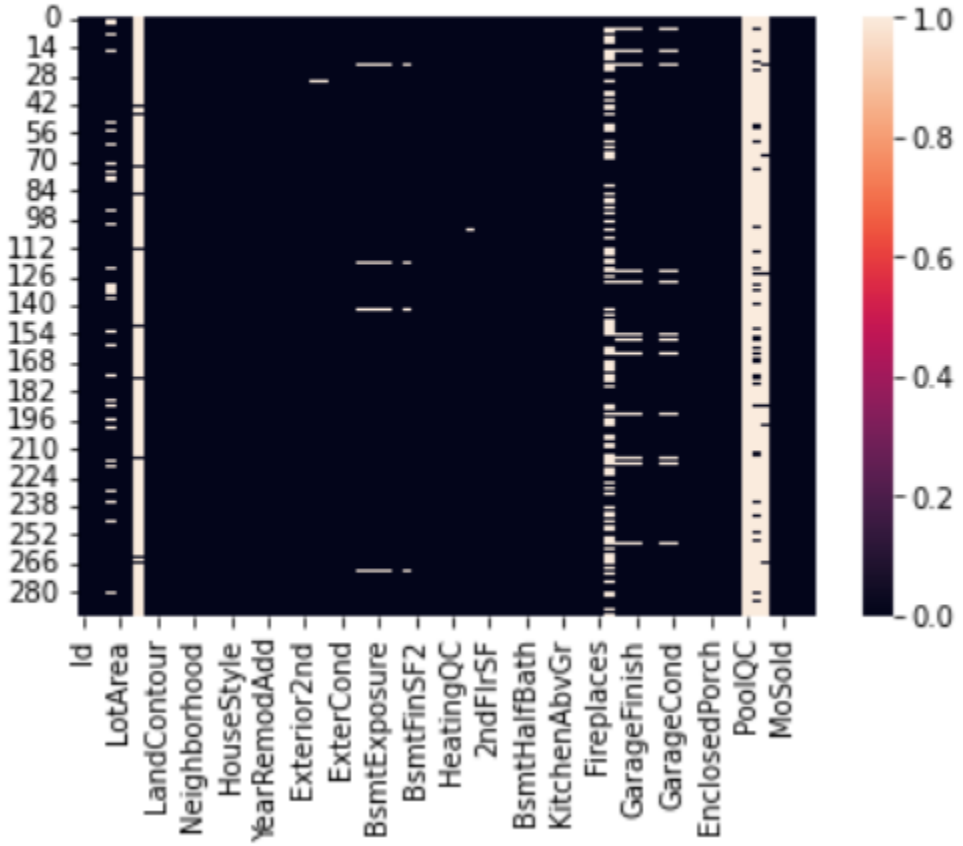
```
<AxesSubplot:>
```



Check with the help of heatmap null value of test data

```
import seaborn as sns
sns.heatmap(test_df.isnull())
```

```
<AxesSubplot:>
```



Check the percentage of null values of train data

Dealing with Missing Values

We should deal with the problem of missing values because some machine learning models don't accept data with missing values. Firstly, let's see the number of missing values in our dataset. We want to see the number and the percentage of missing values for each column that actually contains missing values.

```
#frist step to find out the  which columns has null value is presented
features_with_na=[features for features in df.columns if df[features].isnull().sum()>1]
#step2 to print the columns name and the percentage of missing values
for features in features_with_na:
    print(features,np.round(df[features].isnull().mean(),4),  '% missing values' )
```

```
LotFrontage 0.1832 % missing values
Alley 0.9341 % missing values
MasVnrType 0.006 % missing values
MasVnrArea 0.006 % missing values
BsmtQual 0.0257 % missing values
BsmtCond 0.0257 % missing values
BsmtExposure 0.0265 % missing values
BsmtFinType1 0.0257 % missing values
BsmtFinType2 0.0265 % missing values
FireplaceQu 0.4717 % missing values
GarageType 0.0548 % missing values
GarageYrBlt 0.0548 % missing values
GarageFinish 0.0548 % missing values
GarageQual 0.0548 % missing values
GarageCond 0.0548 % missing values
PoolQC 0.994 % missing values
Fence 0.7971 % missing values
MiscFeature 0.9623 % missing values
```

Check the percentage of null values of test data

```
#frist step to find out the  which columns has null value is presented
features_with_na=[features for features in test_df.columns if test_df[features].isnull().sum()
#step2 to print the columns name and the percentage of missing values
for features in features_with_na:
    print(features,np.round(test_df[features].isnull().mean(),4),  '% missing values' )
```

```
LotFrontage 0.1541 % missing values
Alley 0.9521 % missing values
BsmtQual 0.024 % missing values
BsmtCond 0.024 % missing values
BsmtExposure 0.024 % missing values
BsmtFinType1 0.024 % missing values
BsmtFinType2 0.024 % missing values
FireplaceQu 0.476 % missing values
GarageType 0.0582 % missing values
GarageYrBlt 0.0582 % missing values
GarageFinish 0.0582 % missing values
GarageQual 0.0582 % missing values
GarageCond 0.0582 % missing values
PoolQC 1.0 % missing values
Fence 0.8493 % missing values
MiscFeature 0.9658 % missing values
```

To fill up the null values of train data

```
df["LotFrontage"]=df["LotFrontage"].fillna(df["LotFrontage"].mean())
```

```
#this columns having a lot of null values present so drop thid columns
df.drop(["Alley","PoolQC","MiscFeature","Fence"],axis=1,inplace=True)
```

```
df["BsmtQual"]=df["BsmtQual"].fillna(df["BsmtQual"].mode()[0])
df["BsmtCond"]=df["BsmtCond"].fillna(df["BsmtCond"].mode()[0])
df["FireplaceQu"]=df["FireplaceQu"].fillna(df["FireplaceQu"].mode()[0])
df["GarageType"]=df["GarageType"].fillna(df["GarageType"].mode()[0])
df["GarageFinish"]=df["GarageFinish"].fillna(df["GarageFinish"].mode()[0])
df["GarageQual"]=df["GarageQual"].fillna(df["GarageQual"].mode()[0])
df["GarageCond"]=df["GarageCond"].fillna(df["GarageCond"].mode()[0])
```

here iam fill with most frequent value with the help of mode function

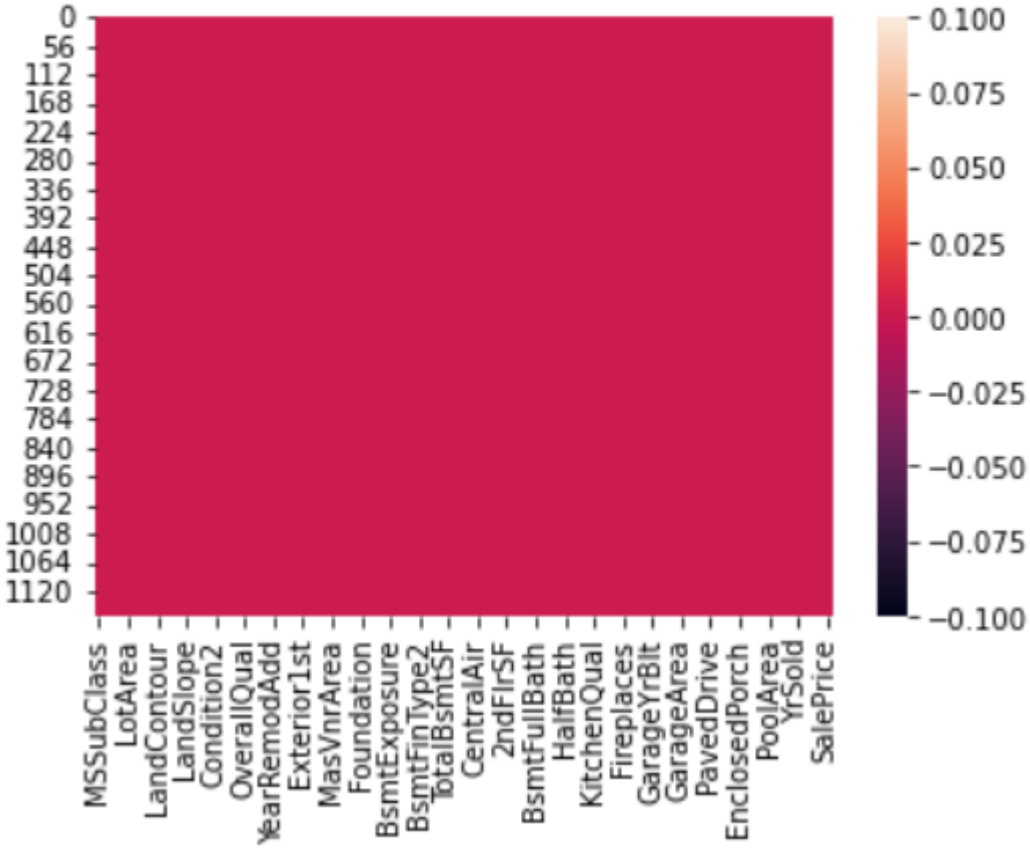
```
df["GarageYrBlt"]=df["GarageYrBlt"].fillna(df["GarageYrBlt"].mean())
```

```
df["MasVnrType"]=df["MasVnrType"].fillna(df["MasVnrType"].mode()[0])
df["MasVnrArea"]=df["MasVnrArea"].fillna(df["MasVnrArea"].mode()[0])
df["BsmtExposure"]=df["BsmtExposure"].fillna(df["BsmtExposure"].mode()[0])
df["BsmtFinType1"]=df["BsmtFinType1"].fillna(df["BsmtFinType1"].mode()[0])
df["BsmtFinType2"]=df["BsmtFinType2"].fillna(df["BsmtFinType2"].mode()[0])
```

To check again having a any of the columns null values remaining in train data

```
import seaborn as sns
sns.heatmap(df.isnull())
```

```
<AxesSubplot:>
```



here showing that non ot the features has missing value so complete the first step here to fill the nan values

To fill up the null values of test data

```
test_df["LotFrontage"]=test_df["LotFrontage"].fillna(test_df["LotFrontage"].mean())
```

```
test_df["Alley"].isnull().sum()
```

```
278
```

```
test_df["PoolQC"].isnull().sum()
```

```
292
```

```
test_df["MiscFeature"].isnull().sum()
```

```
282
```

```
test_df["Fence"].isnull().sum()
```

```
248
```

---

above columns have a lot of null values present so drop this columns

---

```
test_df.drop(["Alley","PoolQC","MiscFeature","Fence"],axis=1,inplace=True)
```

```
test_df["BsmtQual"]=test_df["BsmtQual"].fillna(test_df["BsmtQual"].mode()[0])
test_df["BsmtCond"]=test_df["BsmtCond"].fillna(test_df["BsmtCond"].mode()[0])
test_df["FireplaceQu"]=test_df["FireplaceQu"].fillna(test_df["FireplaceQu"].mode()[0])
test_df["GarageType"]=test_df["GarageType"].fillna(test_df["GarageType"].mode()[0])
test_df["GarageFinish"]=test_df["GarageFinish"].fillna(test_df["GarageFinish"].mode()[0])
test_df["GarageQual"]=test_df["GarageQual"].fillna(test_df["GarageQual"].mode()[0])
test_df["GarageCond"]=test_df["GarageCond"].fillna(test_df["GarageCond"].mode()[0])
```

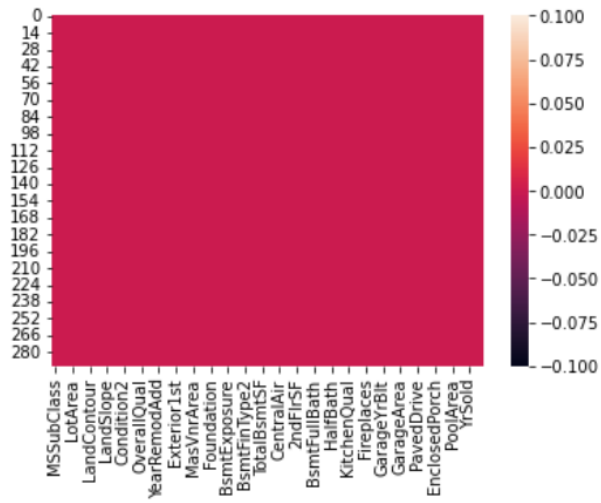
```
test_df["MasVnrType"]=test_df["MasVnrType"].fillna(test_df["MasVnrType"].mode()[0])
test_df["MasVnrArea"]=test_df["MasVnrArea"].fillna(test_df["MasVnrArea"].mode()[0])
test_df["BsmtExposure"]=test_df["BsmtExposure"].fillna(test_df["BsmtExposure"].mode()[0])
test_df["BsmtFinType1"]=test_df["BsmtFinType1"].fillna(test_df["BsmtFinType1"].mode()[0])
test_df["BsmtFinType2"]=test_df["BsmtFinType2"].fillna(test_df["BsmtFinType2"].mode()[0])
test_df["Electrical"]=test_df["Electrical"].fillna(test_df["Electrical"].mode()[0])
```

```
test_df["GarageYrBlt"]=test_df["GarageYrBlt"].fillna(test_df["GarageYrBlt"].mean())
```

To check again having a any of the columns null values remaining in test data

```
import seaborn as sns
sns.heatmap(test_df.isnull())
```

```
<AxesSubplot:>
```



here showing that non the features has missing value is present so here handle the missing value first step is completed

Now, let's get statistical information about the numeric columns in our train dataset. We want to know the mean, the standard deviation, the minimum, the maximum, and the 50th percentile (the median) for *each numeric column* in the dataset:

```
df.describe()
```

	MSSubClass	LotFrontage	LotArea	OverallQual	OverallCond	YearBuilt	YearRemodAdd	MasVnrArea	BsmtFinSF1	BsmtFinSF2	...	WoodDeck
count	1168.000000	1168.000000	1168.000000	1168.000000	1168.000000	1168.000000	1168.000000	1168.000000	1168.000000	1168.000000	...	1168.000(
mean	56.767979	70.988470	10484.749144	6.104452	5.595890	1970.930651	1984.758562	101.696918	444.726027	46.647260	...	96.206
std	41.940650	22.437056	8957.442311	1.390153	1.124343	30.145255	20.785185	182.218483	462.664785	163.520016	...	126.158
min	20.000000	21.000000	1300.000000	1.000000	1.000000	1875.000000	1950.000000	0.000000	0.000000	0.000000	...	0.000(
25%	20.000000	60.000000	7621.500000	5.000000	5.000000	1954.000000	1966.000000	0.000000	0.000000	0.000000	...	0.000(
50%	50.000000	70.988470	9522.500000	6.000000	5.000000	1972.000000	1993.000000	0.000000	385.500000	0.000000	...	0.000(
75%	70.000000	79.250000	11515.500000	7.000000	6.000000	2000.000000	2004.000000	160.000000	714.500000	0.000000	...	171.000(
max	190.000000	313.000000	164660.000000	10.000000	9.000000	2010.000000	2010.000000	1600.000000	5644.000000	1474.000000	...	857.000(

8 rows × 37 columns

describe():- The describe() method is used for calculating some statistical data like percentile, mean and std of the numerical values of the Series or DataFrame. It analyzes both numeric and object series and also the DataFrame column sets of mixed data types.

key observation:

YearBuilt,LotArea, yearsRemoveAdd, BsmtFinSF1 WoodDeckSF yrSold Saleprice,MasVnrArea,MSSubClass,LotFrontage,BsmtFinSF2,OpenPorchSF,MiscVal

this columns having a very high std.

MSSubClass,LotArea,YearBuilt,OpenPorchSF,SalePrice this columns having a right skewed data because in this columns mean is greather than median

YearRemodAdd,YrSold this columns having a left skewed data because this columns having a mean is less than meadian

there is a difference between min value and 25th percentile in this columns like LotFrontage,LotArea,OverallQual,OverallCond,YearBuilt,YearRemodAdd

MoSold,YrSold,SalePric it mean possibilities is having a outliers are present in this columns

there is a difference between max value and 75th percentile in this columns like LotFrontage,LotArea,OverallQual,OverallCond,YearBuilt,YearRemodAdd

MoSold,YrSold,SalePric it mean possibilities is having a outliers are present in this columns

Now, let's get statistical information about the numeric columns in our test dataset. We want to know the mean, the standard deviation, the minimum, the maximum, and the 50th percentile (the median) for *each numeric column* in the dataset:

```
test_df.describe()
```

	MSSubClass	LotFrontage	LotArea	OverallQual	OverallCond	YearBuilt	YearRemodAdd	MasVnrArea	BsmtFinSF1	BsmtFinSF2	...	GarageAre
count	292.000000	292.000000	292.000000	292.000000	292.000000	292.000000	292.000000	292.000000	292.000000	292.000000	...	292.00000
mean	57.414384	66.425101	10645.143836	6.078767	5.493151	1972.616438	1985.294521	108.797945	439.294521	46.157534	...	457.45890
std	43.780649	19.975962	13330.669795	1.356147	1.063267	30.447016	20.105792	174.845785	429.559675	152.467119	...	210.78559
min	20.000000	21.000000	1526.000000	3.000000	3.000000	1872.000000	1950.000000	0.000000	0.000000	0.000000	...	0.00000
25%	20.000000	57.750000	7200.000000	5.000000	5.000000	1954.000000	1968.000000	0.000000	0.000000	0.000000	...	300.00000
50%	50.000000	66.425101	9200.000000	6.000000	5.000000	1976.000000	1994.000000	0.000000	369.500000	0.000000	...	467.50000
75%	70.000000	76.000000	11658.750000	7.000000	6.000000	2001.000000	2003.250000	180.000000	700.500000	0.000000	...	569.75000
max	190.000000	150.000000	215245.000000	10.000000	9.000000	2009.000000	2010.000000	1031.000000	1767.000000	1085.000000	...	1052.00000

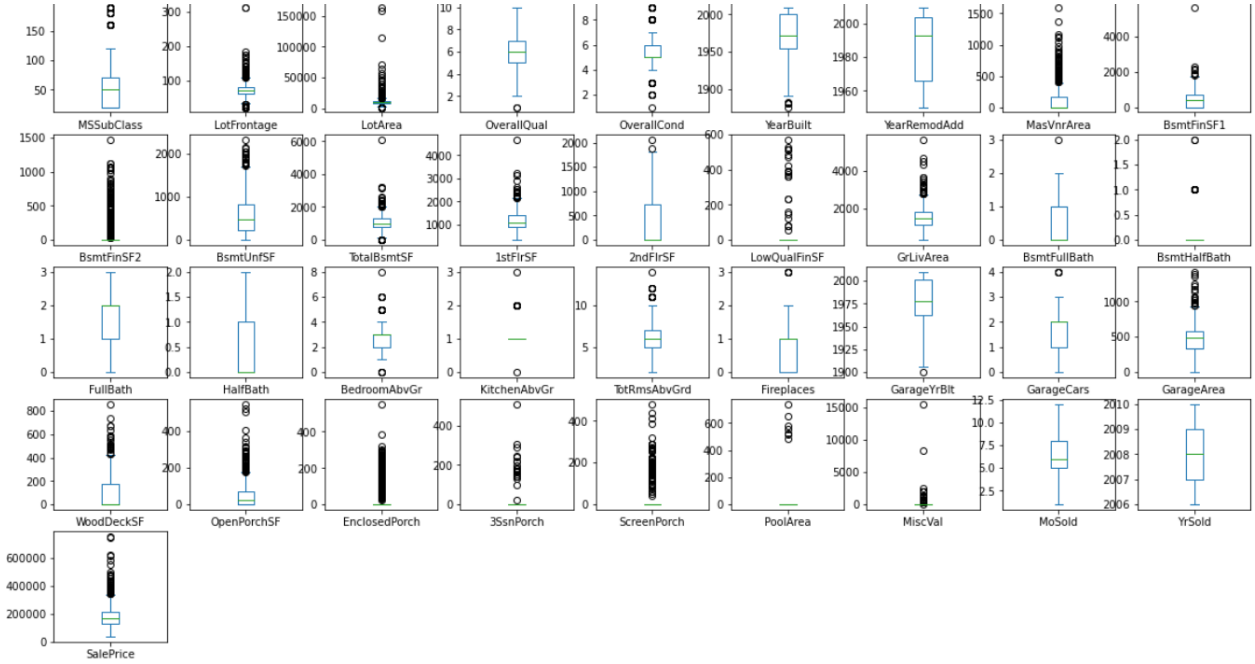
8 rows × 36 columns

key observation:

YearBuilt,LotArea, yearsRemoveAdd, BsmtFinSF1 WoodDeckSF yrSold Saleprice,MasVnrArea,MSSubClass,LotFrontage,BsmtFinSF2,OpenPorchSF,MiscVal
this columns having a very high std.
MSSubClass,LotArea,YearBuilt,OpenPorchSF,SalePrice this columns having a right skewed data because in this columns mean is greather than median
YearRemodAdd,YrSold this columns having a left skewed data because this columns having a mean is less than meadian
there is a difference between min value and 25th percentile in this columns like LotFrontage,LotArea,OverallQual,OverallCond,YearBuilt,YearRemodAdd
MoSold,YrSold,SalePric it mean possibilities is having a outliers are present in this columns

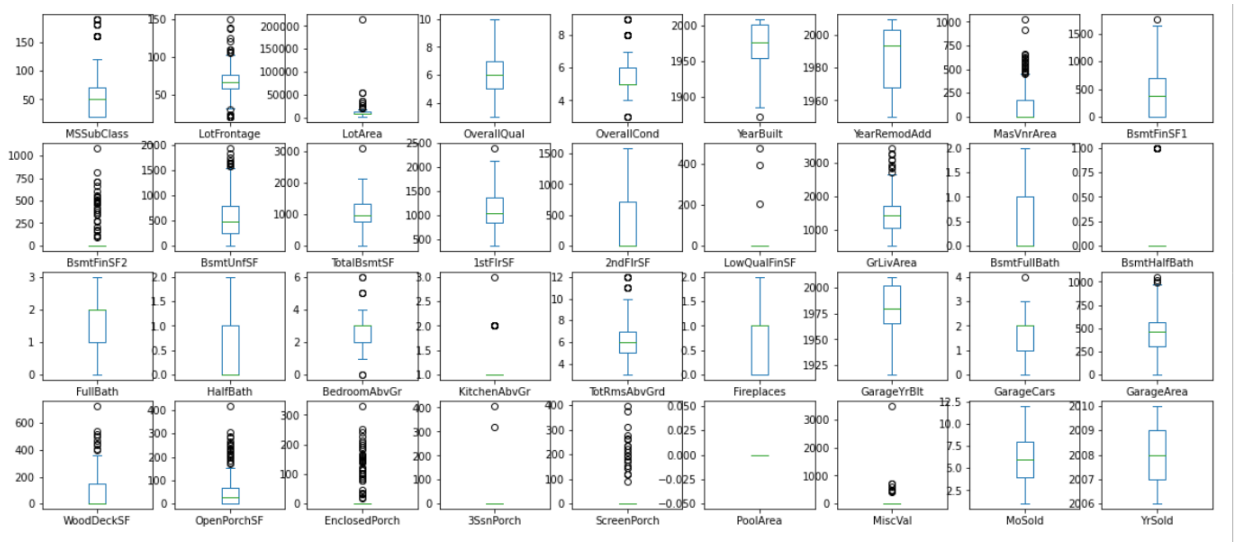
To detect the outliers with the help of boxplot of a train data

```
df.plot(kind='box',subplots=True,layout=(9,9),figsize=(20,20))
```



To detect the outliers with the help of boxplot of a test data

```
test_df.plot(kind='box',subplots=True,layout=(9,9),figsize=(20,20))
```



To check the numerical columns of train dataset

Numerical variables

```
#separate the list of numerical variables and print out the numerical features
numerical_features =[feature for feature in df.columns if df[feature].dtypes != 'O' ]
print("number of numerical variables: " , len(numerical_features))
#head of numerical features
df[numerical_features].head()
```

```
number of numerical variables:  37
```

	MSSubClass	LotFrontage	LotArea	OverallQual	OverallCond	YearBuilt	YearRemodAdd	MasVnrArea	BsmtFinSF1	BsmtFinSF2	...	WoodDeckSF	OpenPorc
0	120	70.98847	4928	6	5	1976	1976	0.0	120	0	...	0	
1	20	95.00000	15865	8	6	1970	1970	0.0	351	823	...	81	
2	60	92.00000	9920	7	5	1996	1997	0.0	862	0	...	180	
3	20	105.00000	11751	6	6	1977	1977	480.0	705	0	...	0	
4	20	70.98847	16635	6	7	1977	2000	126.0	1246	0	...	240	

5 rows × 37 columns

Temporal variables eg.datetime variables

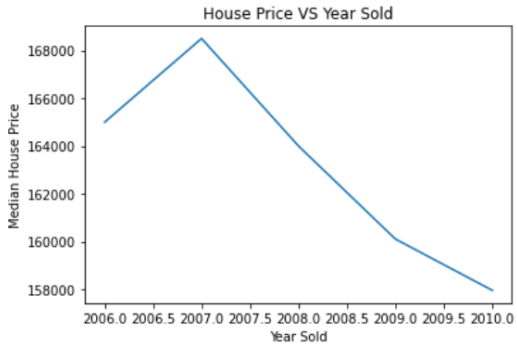
from the datasets we have the 4 years variables we have to extaract the informations from this datetime variables like no of years or no of days here we to perfrom the analysis with this datatime variables

one of the examples in a specific scenario can be a difference between the year the house was built and the year the house was sold so we will be performed the analysis with this features

```
#list of years features that contain the year information
year_features=[feature for feature in numerical_features if 'Yr' in feature or 'Year' in feature]
year_features
```

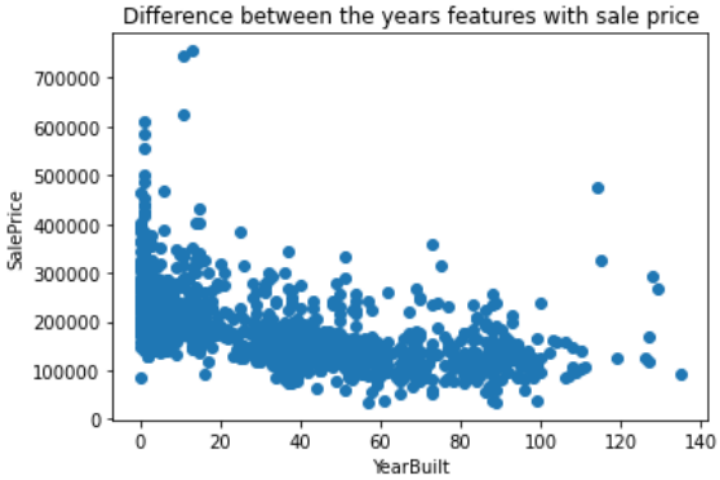
```
['YearBuilt', 'YearRemodAdd', 'GarageYrBlt', 'YrSold']
```

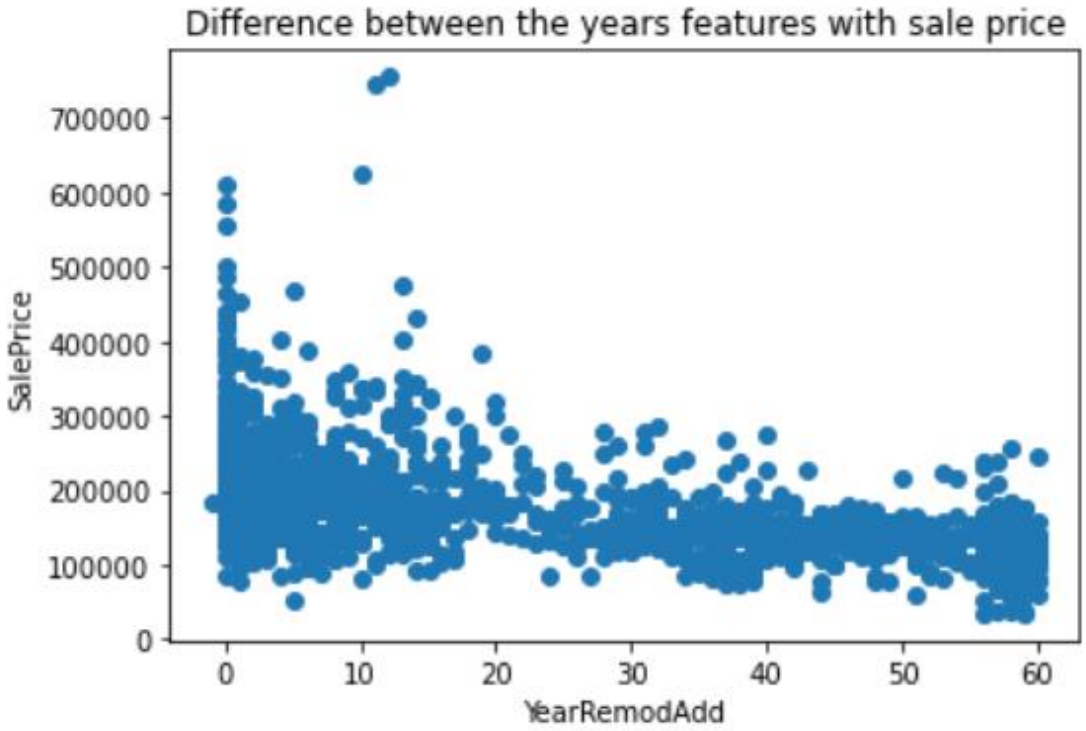
```
#lets analysis the temporal datatiime variables
# we will check the weather any relations between year the houes is sold and price of the house is incresing or decreasing
import matplotlib.pyplot as plt
df.groupby('YrSold')['SalePrice'].median().plot()
plt.xlabel("Year Sold")
plt.ylabel("Median House Price")
plt.title('House Price VS Year Sold')
plt.show()
```

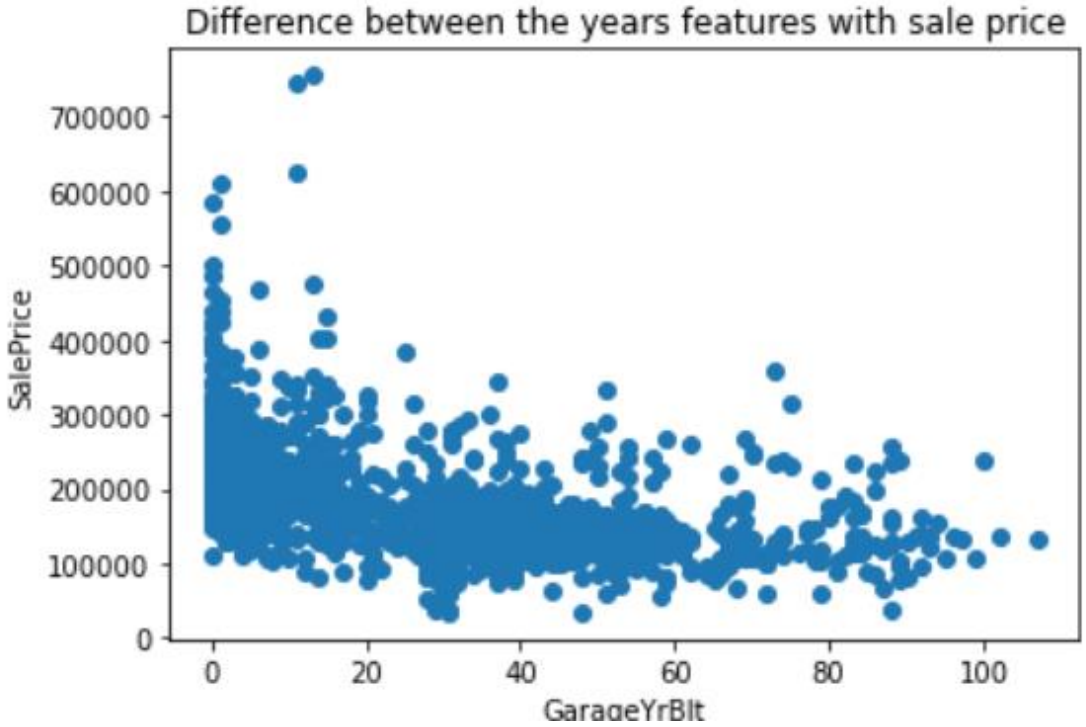


here show that increasing the year house price is decresing so iam not consider that ie true because incersing the year house price is also incresing

```
#here we will compare the difference between the years features with sale price
for feature in year_features:
    if feature != 'YrSold':
        data=df.copy()
        data[feature]=data['YrSold']-data[feature]
        plt.scatter(data[feature],data['SalePrice'])
        plt.xlabel(feature )
        plt.ylabel('SalePrice')
        plt.title('Difference between the years features with sale price')
        plt.show()
```







we will capture the difference between year variables and year the house was sold for it will give more informations about the year variables.here every graph is show that house was new the price of house is more and and house was older the price is less

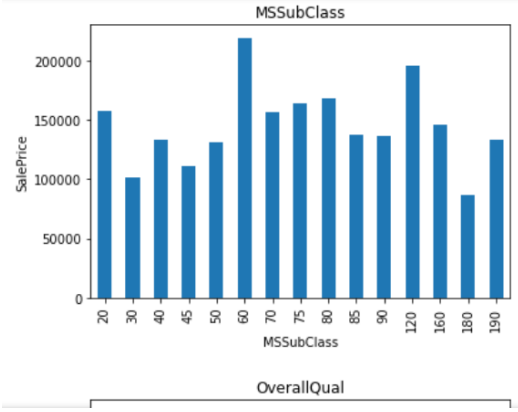
```
#here iam separate the discrete feature from numerical feature because numerical features has two types 1) is continous feature
#2) is discrete feature ,discreate feature has fixed value ie countable eg. age
discreate_features=[feature for feature in numerical_features if len(df[feature].unique())<25 and feature not in year_features]
print("Discreate feature count: {}".format(len(discreate_features)))
```

```
Discreate feature count: 17
```

```
#visulize the discreate feature with head method
df[discreate_features].head()
```

	MSSubClass	OverallQual	OverallCond	LowQualFinSF	BsmtFullBath	BsmtHalfBath	FullBath	HalfBath	BedroomAbvGr	KitchenAbvGr	TotRmsAbvGrd	Firep
0	120	6	5	0	0	0	2	0	2	1	5	
1	20	8	6	0	1	0	2	0	4	1	8	
2	60	7	5	0	1	0	2	1	3	1	8	
3	20	6	6	0	0	0	2	0	3	1	7	
4	20	6	7	0	0	1	2	0	3	1	8	

```
#lets find out the relationships between discreate features and SalePrice to make vislizations with this features
for feature in discreate_features:
    data=df.copy()
    data.groupby(feature)['SalePrice'].median().plot.bar()
    plt.xlabel(feature)
    plt.ylabel('SalePrice')
    plt.title(feature)
    plt.show()
```



Continuous variables

```
#lets find out the contionous features to make the analysis with saleprice
continuous_features=[feature for feature in numerical_features if feature not in discreate_features+year_features ]
print("continuous feature count {}".format(len(continuous_features )))
```

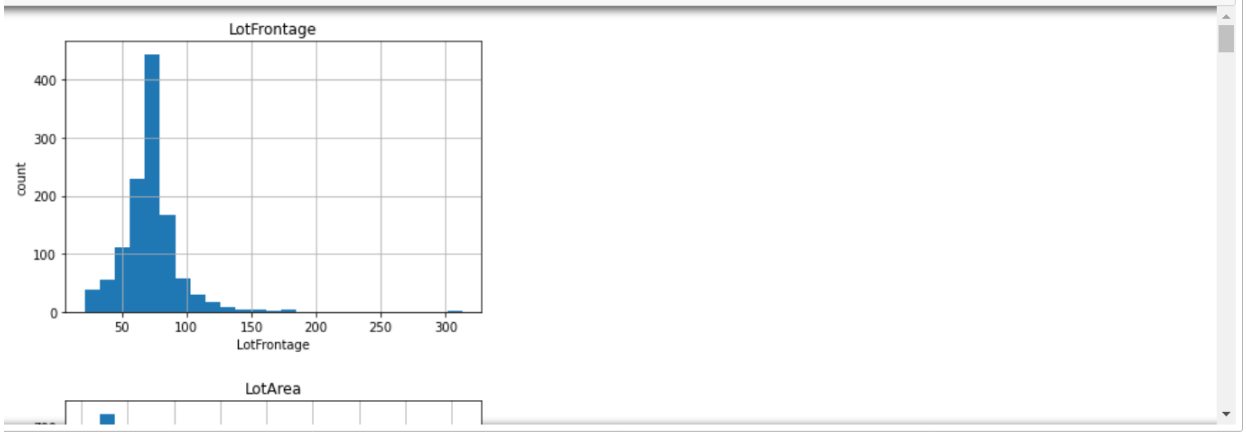
```
continuous feature count 16
```

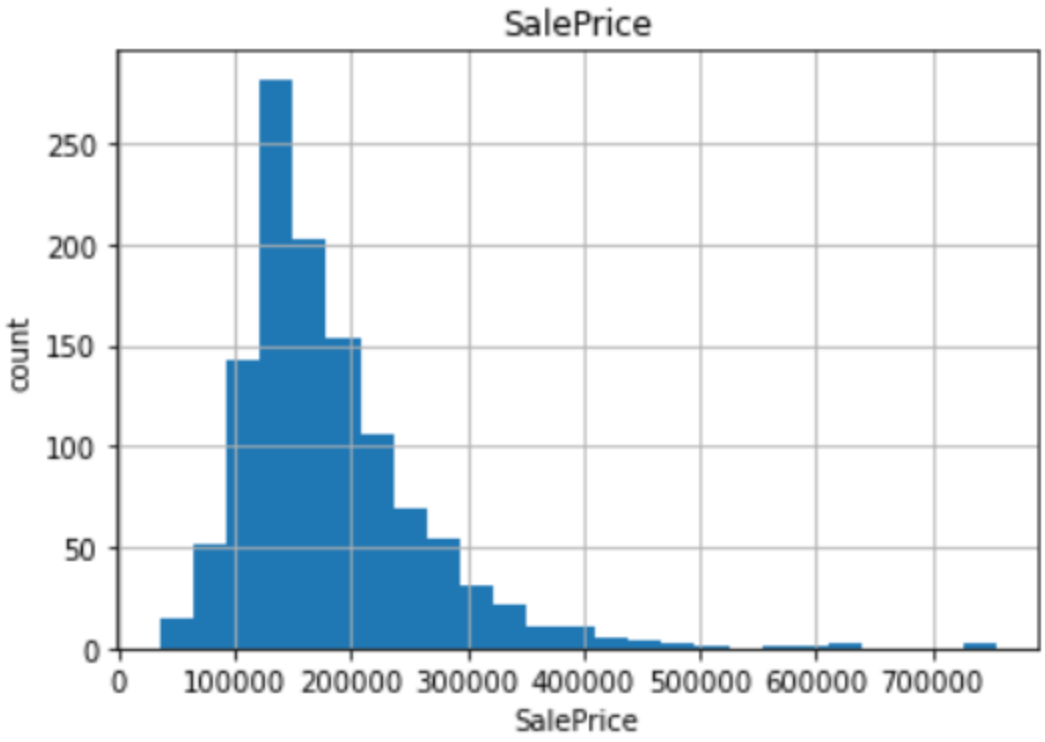
```
#visulize the continuous features
df[continuous_features].head()
```

	LotFrontage	LotArea	MasVnrArea	BsmtFinSF1	BsmtFinSF2	BsmtUnfSF	TotalBsmtSF	1stFlrSF	2ndFlrSF	GrLivArea	GarageArea	WoodDeckSF	OpenPorc
0	70.98847	4928	0.0	120	0	958	1078	958	0	958	440	0	
1	95.00000	15865	0.0	351	823	1043	2217	2217	0	2217	621	81	
2	92.00000	9920	0.0	862	0	255	1117	1127	886	2013	455	180	
3	105.00000	11751	480.0	705	0	1139	1844	1844	0	1844	546	0	
4	70.98847	16635	126.0	1246	0	356	1602	1602	0	1602	529	240	

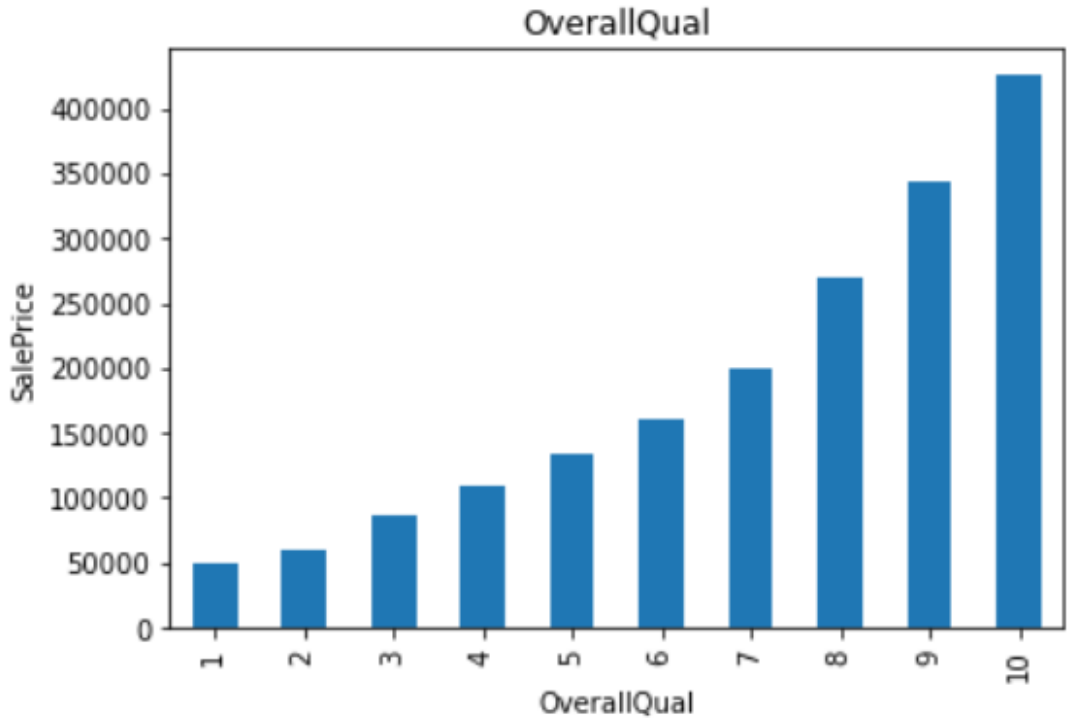
```
#Lets analyses the continuous features with histogram to find out the  in each continous columns having a skewness or notand it
#is normal distribution or not or data is left skewed or right skewed data
```

```
for feature in continuous_features:
    data=df.copy()
    data[feature].hist(bins=25)
    plt.xlabel(feature)
    plt.ylabel("count")
    plt.title(feature)
    plt.show()
```





We can see that most house prices fall between 100,000 and 200,000. We see also that there is a number of expensive houses to the right of the plot. Now, we move to see the distribution of `Overall Qual` variable:



We see that `Overall Qual` takes an integer value between 1 and 10, and that most houses have an overall quality between 5 and 10.

Categorical Feature

```
#separate the categorical feature from datasets for analysis
categorical_features=[feature for feature in df.columns if df[feature].dtype=='O']
print("number of categorical features:", len(categorical_features))
```

```
number of categorical features: 39
```

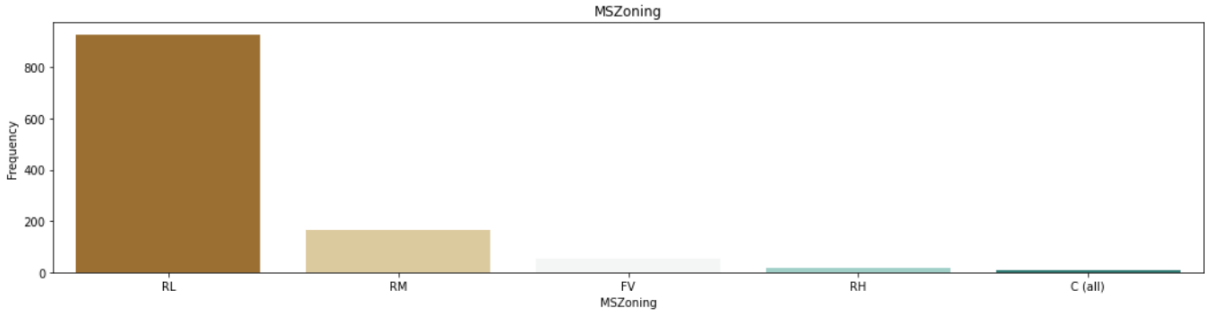
```
df[categorical_features].head()
```

	MSZoning	Street	LotShape	LandContour	Utilities	LotConfig	LandSlope	Neighborhood	Condition1	Condition2	...	KitchenQual	Functional	FireplaceQu
0	RL	Pave	IR1	Lvl	AllPub	Inside	Gtl	NPkVill	Norm	Norm	...	TA	Typ	TA
1	RL	Pave	IR1	Lvl	AllPub	Inside	Mod	NAmes	Norm	Norm	...	Gd	Typ	TA
2	RL	Pave	IR1	Lvl	AllPub	CulDSac	Gtl	NoRidge	Norm	Norm	...	TA	Typ	TA
3	RL	Pave	IR1	Lvl	AllPub	Inside	Gtl	NWAmes	Norm	Norm	...	TA	Typ	TA
4	RL	Pave	IR1	Lvl	AllPub	FR2	Gtl	NWAmes	Norm	Norm	...	Gd	Typ	TA

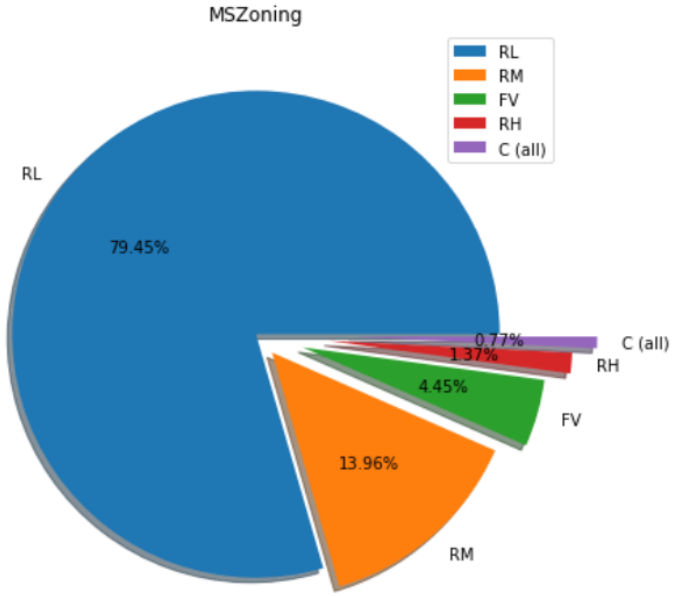
5 rows × 39 columns

```
#lets analysis the categorical features with the help of countplot,countplot is count the values of each categories is predented
#in the each columns it will shows the value counts of each category is presented in the columns it will shows the frequency
#here i created function of categorical features to plot the countplot
def plot_categorical(categorical_features):
    plt.subplots(figsize=(18,4))
    sns.countplot(categorical_features,data=df,dodge=True,palette='BrBG')
    plt.xlabel(categorical_features)
    plt.ylabel('Frequency')
    plt.title(categorical_features)
    plt.show()
```

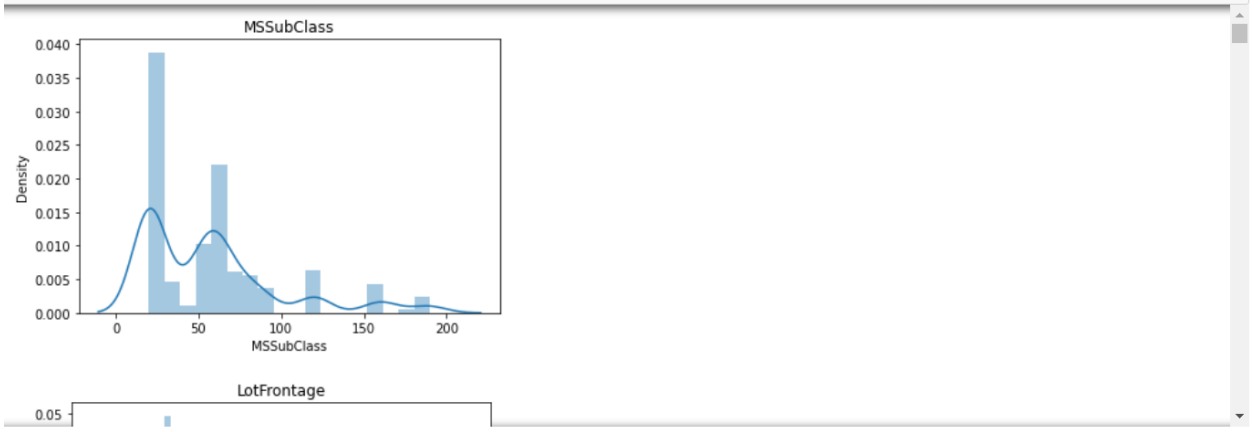
```
import warnings
warnings.filterwarnings('ignore')
plot_categorical('MSZoning')
```



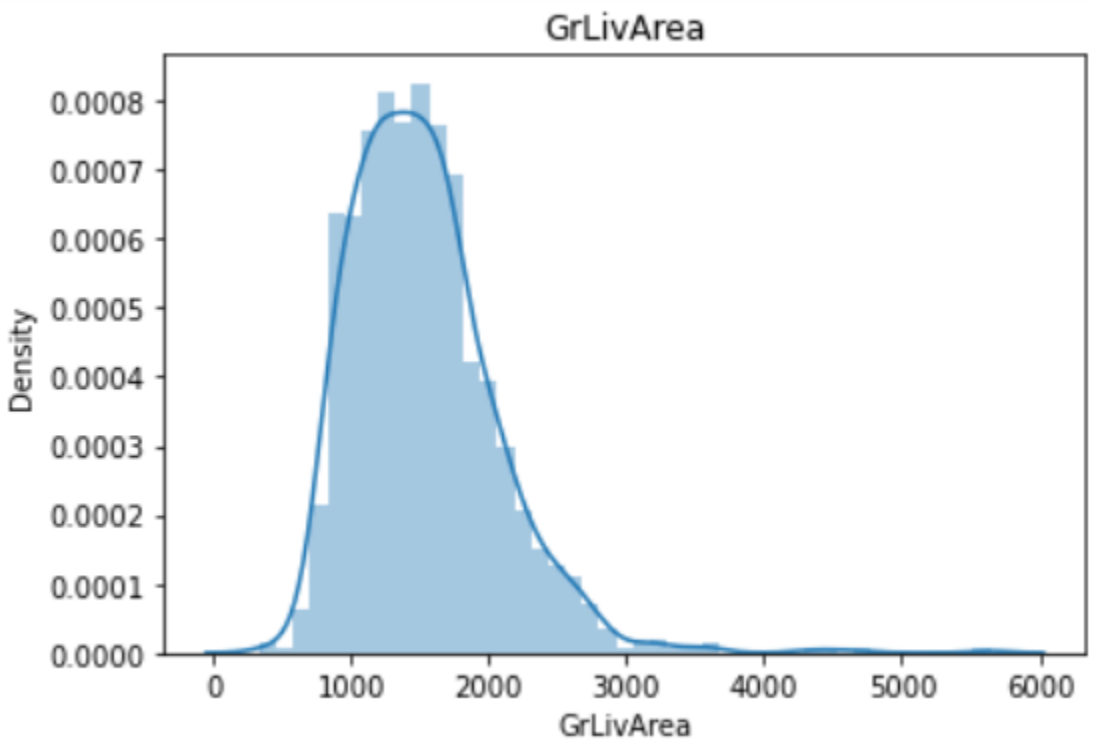
```
]: label=df['MSZoning'].value_counts().index
   data=df['MSZoning'].value_counts().values
   fig=plt.figure(figsize=(10,7))
   plt.pie(data,labels =label,autopct="%.2f%%",shadow=True,explode=(0,0.1,0.2,0.3,0.4))
   plt.title('MSZoning')
   plt.legend()
   plt.show()
```




```
for feature in numerical_features:
    sns.distplot(df[feature])
    plt.xlabel(feature)
    plt.ylabel("Density")
    plt.title(feature)
    plt.show()
```



the sns distplot shows the data are normally distributed are not it shows the data are left skewed or right skewed here showtha some of the numerical fearures are not normally distributed like LotFrontage,LotArea,1stFlrSF,GrLivArea SalePrice this features has data is grether 3std

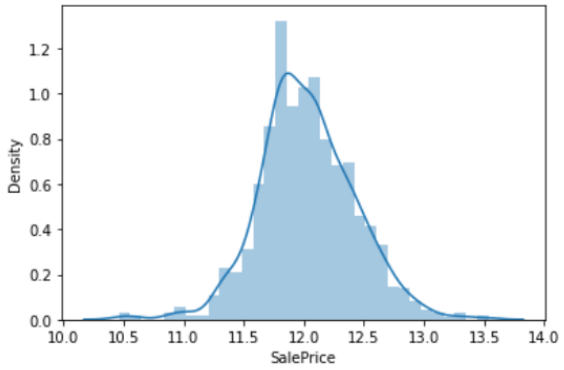


We can see that the above-ground living area falls approximately between 800 and 1800 ft.

```
#here remove the skewness is present in the data with the help of log transform
import numpy as np
num_feature=['LotFrontage','LotArea','1stFlrSF','GrLivArea','SalePrice']
for feature in num_feature:
    df[feature]=np.log(df[feature])
```

```
sns.distplot(df['SalePrice'])
```

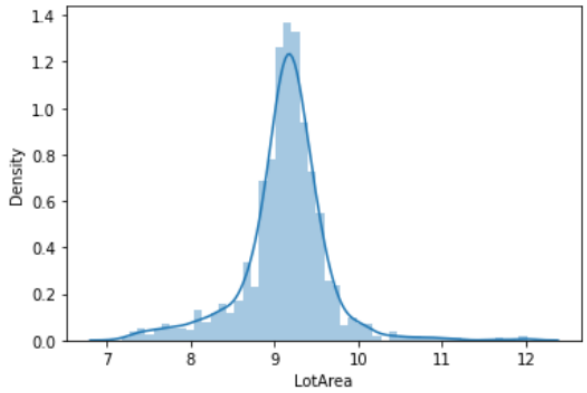
```
<AxesSubplot:xlabel='SalePrice', ylabel='Density'>
```



so here remove the skewness, The Data in the columns is normalized .The building blocks is not out of the normalised curve

```
sns.distplot(df['LotArea'])
```

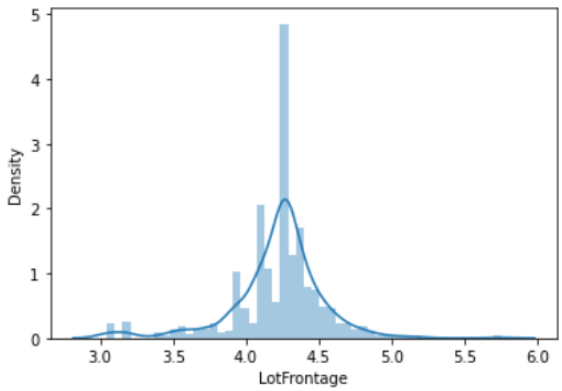
```
<AxesSubplot:xlabel='LotArea', ylabel='Density'>
```



so here remove the skewness, The Data in the columns is normalized .The building blocks is not out of the normalised curve

```
sns.distplot(df['LotFrontage'])
```

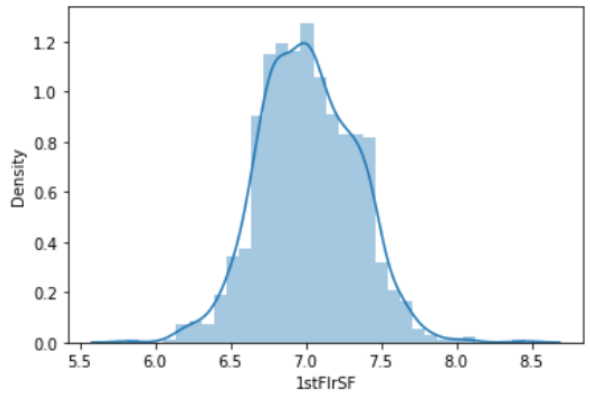
```
<AxesSubplot:xlabel='LotFrontage', ylabel='Density'>
```



so here remove the skewness, The Data in the columns is normalized .The building blocks is not out of the normalised curve

```
sns.distplot(df['1stFlrSF'])
```

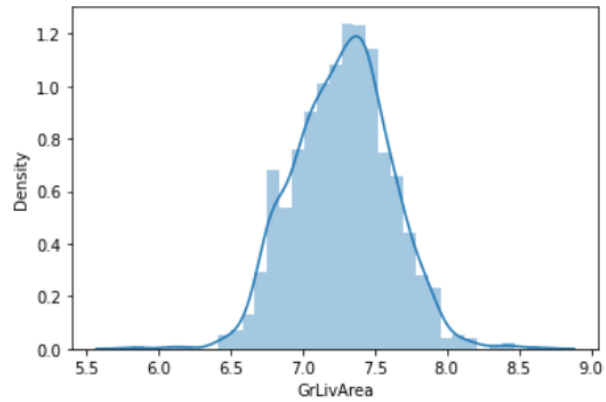
```
<AxesSubplot:xlabel='1stFlrSF', ylabel='Density'>
```



so here remove the skewness, The Data in the columns is normalized .The building blocks is not out of the normalised curve

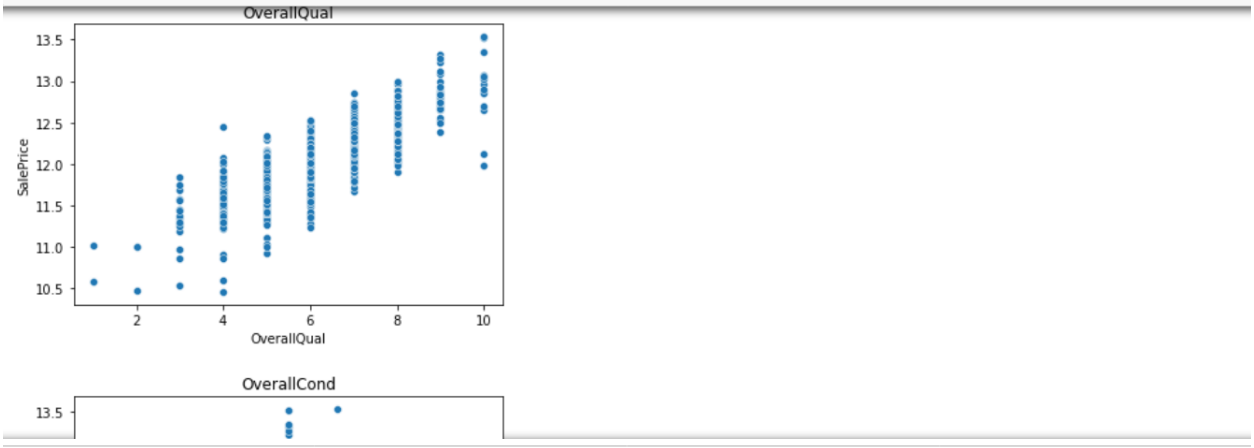
```
sns.distplot(df['GrLivArea'])
```

```
<AxesSubplot:xlabel='GrLivArea', ylabel='Density'>
```

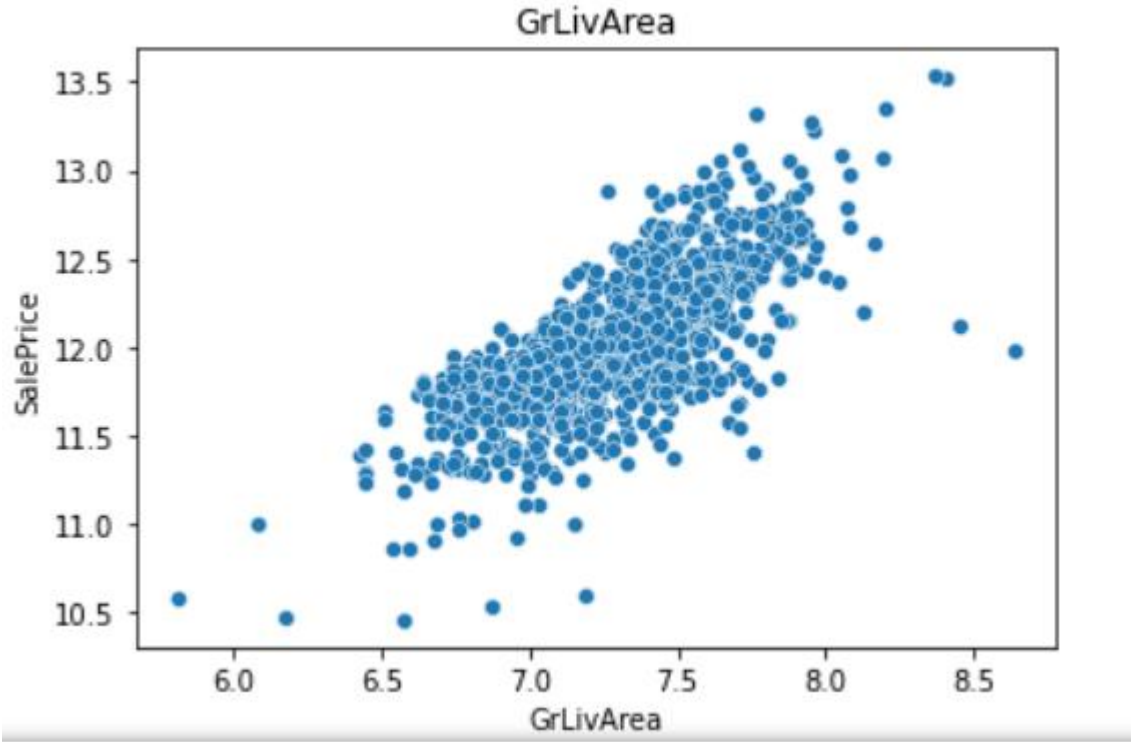


so here remove the skewness, The Data in the columns is normalized .The building blocks is not out of the normalised curve

```
for feature in numerical_features:
    sns.scatterplot(x=df[feature],y='SalePrice',data=df)
    plt.xlabel(feature)
    plt.ylabel('SalePrice')
    plt.title(feature)
    plt.show()
```



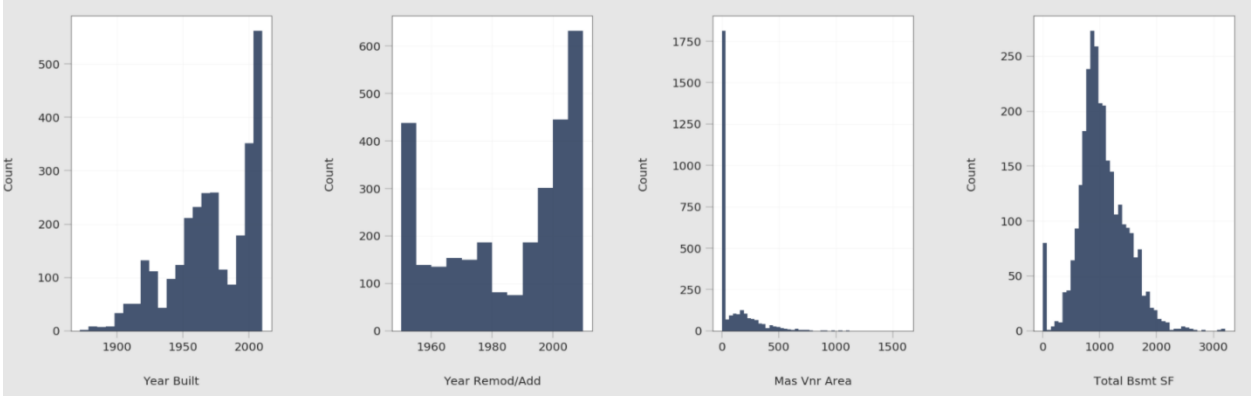
We can see that they are truly positively correlated; generally, as the overall quality increases, the sale price increases too. This verfies what we got from the heatmap

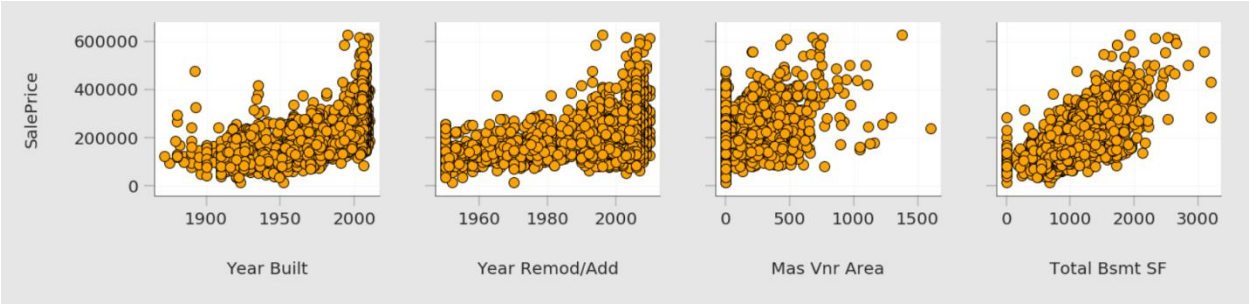


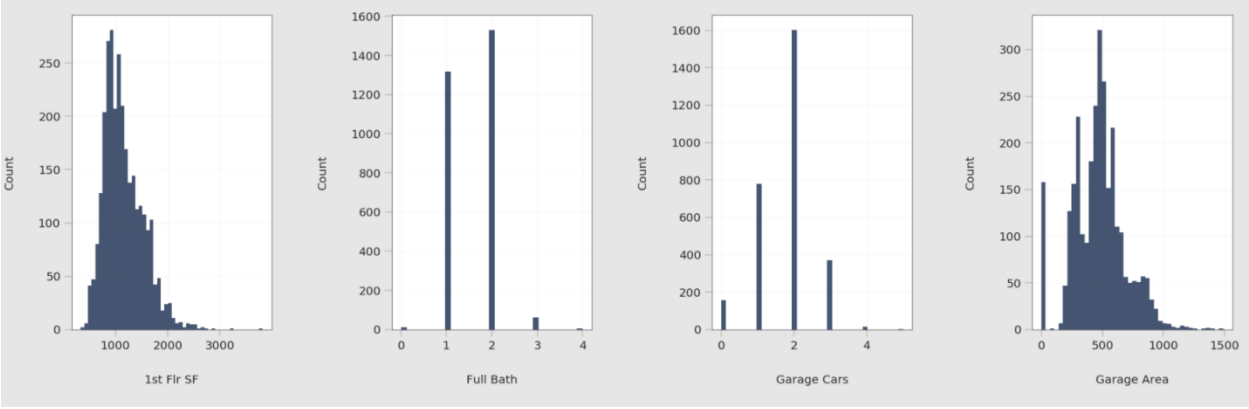
The scatter plot above shows clearly the strong positive correlation between `Gr Liv Area` and `SalePrice`

Moderate Positive Correlation

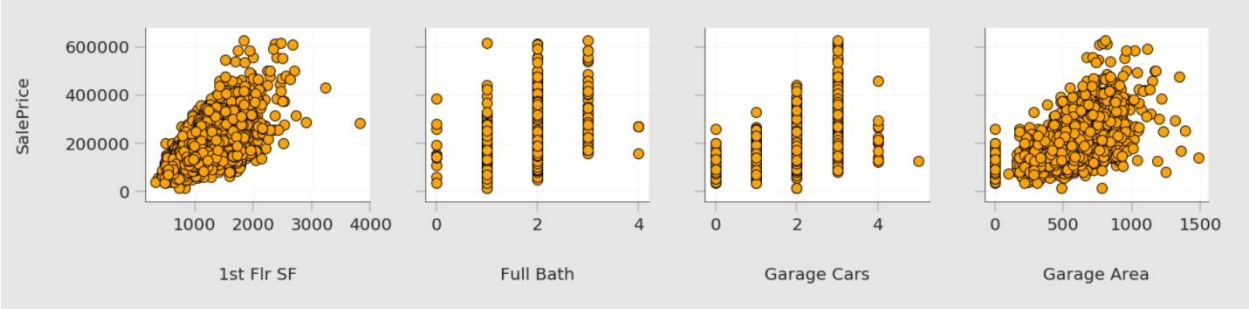
Next, we want to visualize the relationship between the target variable and the variables that are positively correlated with it, but the correlation is not very strong. Namely, these variables are `Year Built`, `Year Remod/Add`, `Mas Vnr Area`, `Total Bsmt SF`, `1st Flr SF`, `Full Bath`, `Garage Cars`, and `Garage Area`. We start with the first four. Let us see the distribution of each of them:







And now let us see their relationships with the target variable:

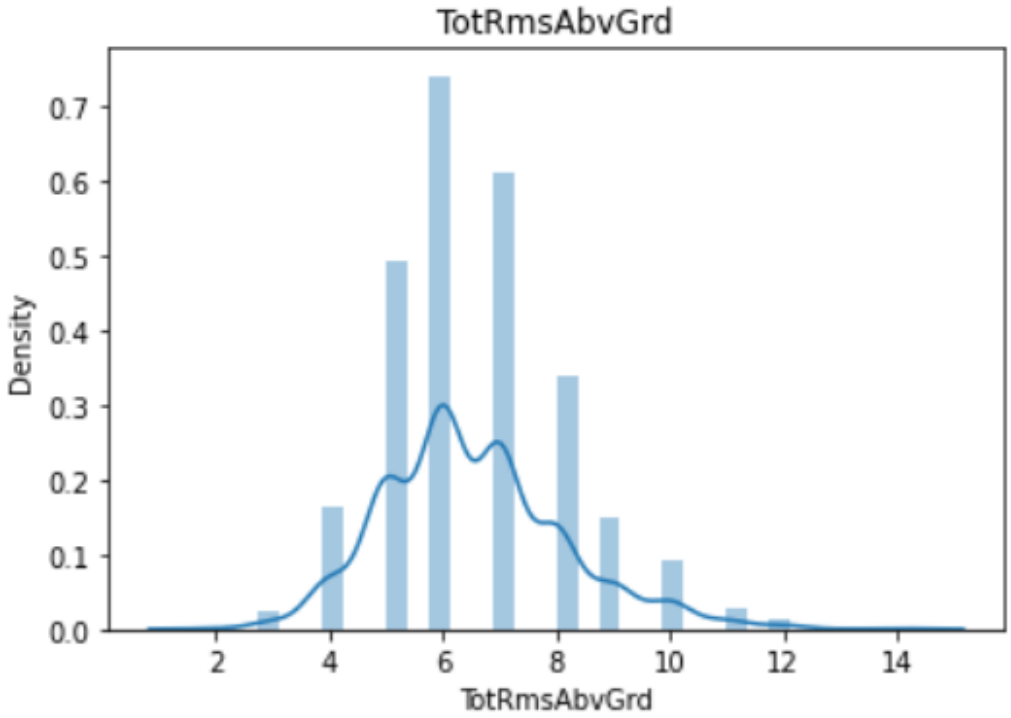


From the plots above, we can see that these eight variables are truly positively correlated with the target variable. However, it's apparent that they are not as highly correlated as `Overall Qual` and `Gr Liv Area`.

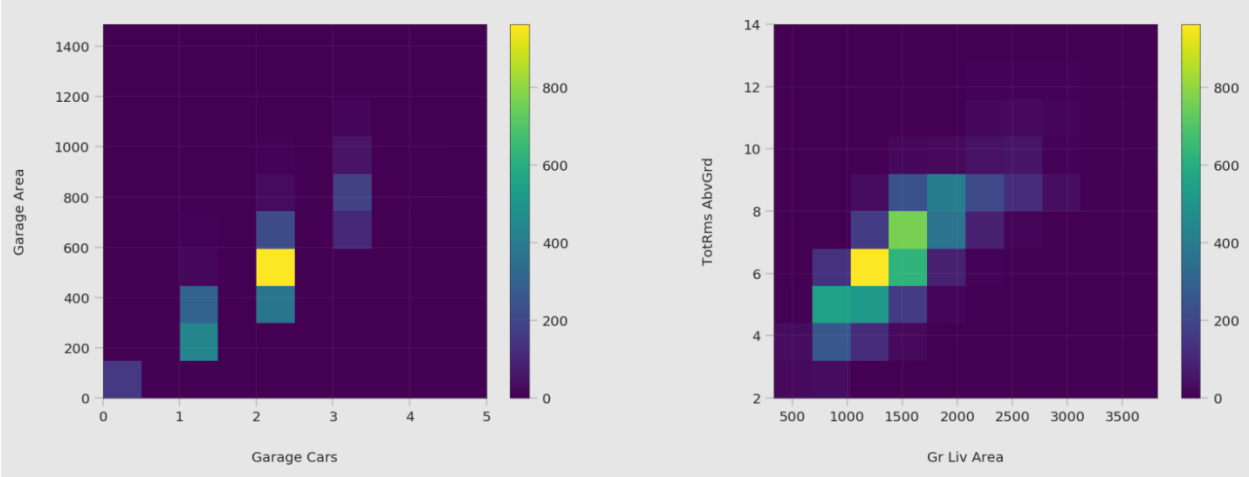
Relatioships Between Predictor Variables¶

Positive Correlation

Apart from the target variable, when we plotted the heatmap, we discovered a high positive correlation between `Garage Cars` and `Garage Area` and between `Gr Liv Area` and `TotRms AbvGrd`. We want to visualize these correlations also. We've already seen the distribution of each of them except for `TotRms AbvGrd`. Let us see the distribution of `TotRms AbvGrd` first:



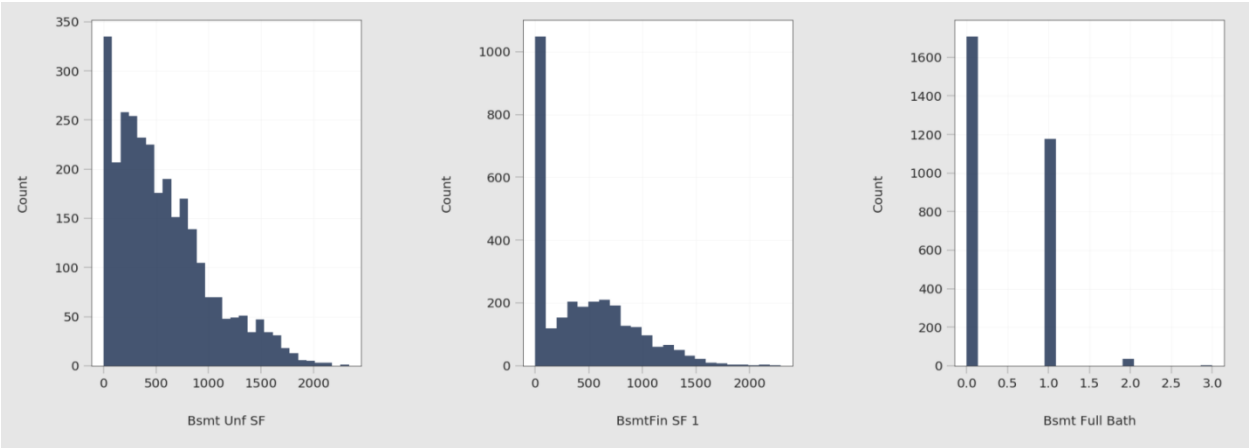
Now, we visualize the relationship between `Garage Cars` and `Garage Area` and between `Gr Liv Area` and `TotRms AbvGrd`:



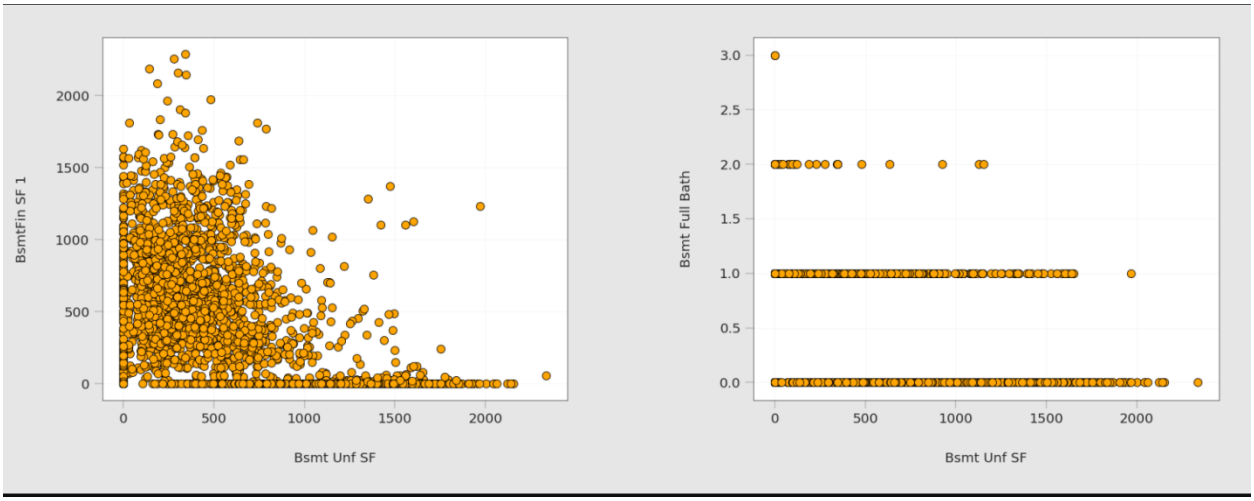
We can see the strong correlation between each pair. For `Garage Cars` and `Garage Area`, we see that the highest concentration of data is when `Garage Cars` is 2 and `Garage Area` is approximately between 450 and 600 ft^2. For `Gr Liv Area` and `TotRms AbvGrd`, we notice that the highest concentration is when `Garage Liv Area` is roughly between 800 and 2000 ft^2 and `TotRms AbvGrd` is 6.

Negative Correlation

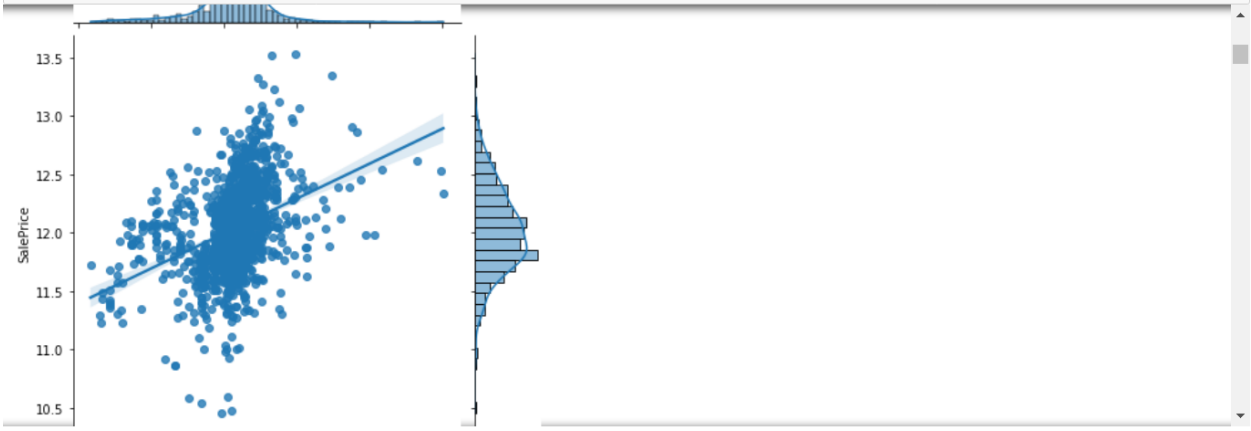
When we plotted the heatmap, we also discovered a significant negative correlation between `Bsmt Unf SF` and `BsmtFin SF 1`, and between `Bsmt Unf SF` and `Bsmt Full Bath`. We also want to visualize these correlations. Let us see the distribution of these variables first:



Now, we visualize the relationship between each pair using scatter plots:




```
for feature in numerical_features:
    sns.jointplot(x=df[feature],y='SalePrice',kind='reg',data=df)
    plt.xlabel(feature)
    plt.ylabel('SalePrice')
    plt.show()
```



YearBuilt,TotalBsmtSF,GrLivArea,OpenPorchSF,WooDeckSF,GarageArea,GarageYrBlt,2ndflrsf,1stflrsf,tottalBsmtsf,Bsmtunfsf,Bamtfinsf1,MasvnrArea,YearRemo
this features is play very important role to predict the saleprice of house because in above jointplot in this variables more points is tuch to the reg line more points can pass this line so this features is important to predict price of house

Correlation Between Variables

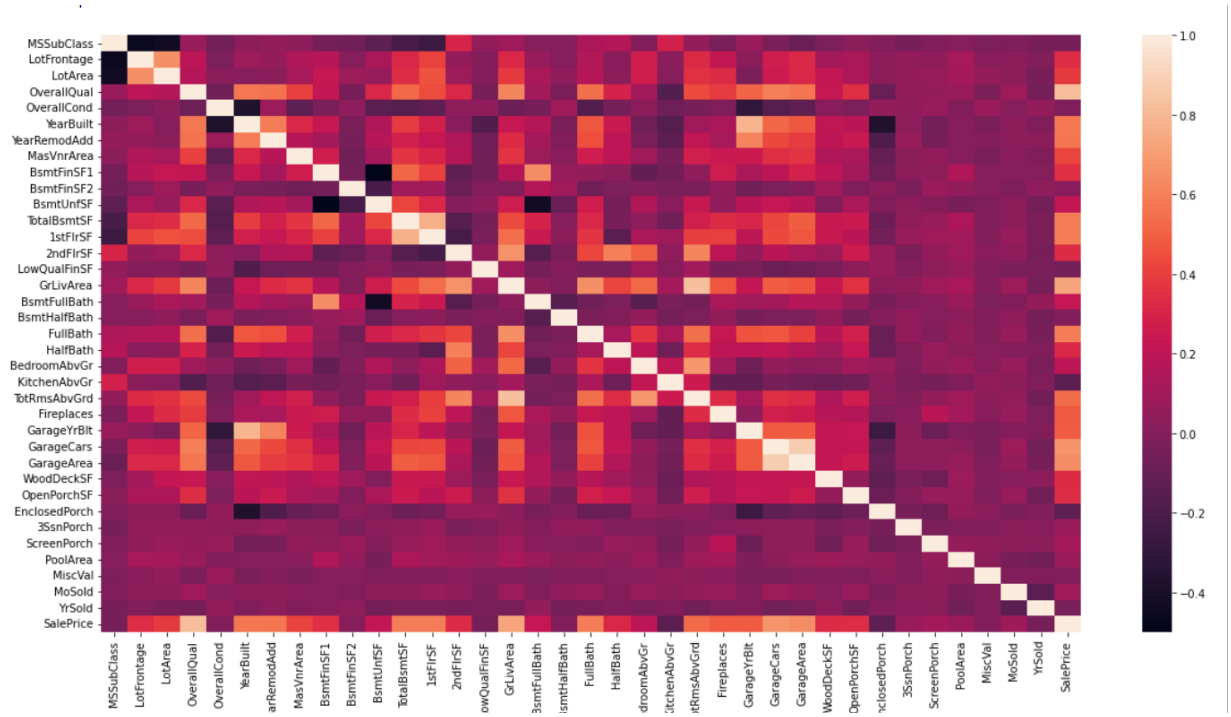
We want to see how the dataset variables are correlated with each other and how predictor variables are correlated with the target variable. For example, we would like to see how `Lot Area` and `SalePrice` are correlated: Do they increase and decrease together (positive correlation)? Does one of them increase when the other decrease or vice versa (negative correlation)? Or are they not correlated?

Correlation is represented as a value between -1 and +1 where +1 denotes the highest positive correlation, -1 denotes the highest negative correlation, and 0 denotes that there is no correlation.

We will show correlation between our dataset variables (numerical and boolean variables only) using a heatmap graph:

plt.figure(figsize=(20,10))

sns.heatmap(df.corr())



We can see that there are many correlated variables in our dataset. Wwe notice that `Garage Cars` and `Garage Area` have high positive correlation which is reasonable because when the garage area increases, its car capacity increases too. We see also that `Gr Liv Area` and `TotRms AbvGrd` are highly positively correlated which also makes sense because when living area above ground increases, it is expected for the rooms above ground to increase too.

Regarding negative correlation, we can see that `Bsmt Unf SF` is negatively correlated with `BsmtFin SF 1`, and that makes sense because when we have more unfinished area, this means that we have less finished area. We note also that `Bsmt Unf SF` is negatively correlated with `Bsmt Full Bath` which is reasonable too.

Most importantly, we want to look at the predictor variables that are correlated with the target variable (`SalePrice`). By looking at the last row of the heatmap, we see that the target variable is highly positively correlated with `Overall Qual` and `Gr Liv Area`. We see also that the target variable is positively correlated with `Year Built`, `Year Remod/Add`, `Mas Vnr Area`, `Total Bsmt SF`, `1st Flr SF`, `Full Bath`, `Garage Cars`, and `Garage Area`.

Observations:

TotalBsmtSF' and '1stFlrSF columns are highly correlated with each other 0.77 and 0.77

GarageCars and GarageArea columns are highly correlated with each other 0.88 and 0.88

OverallQual and SalePrice columns are highly correlated with each other 0.82 and 0.82

GrLivArea and TotRmsAbvGrd colums are highly correlated with each other 0.83 and 0.83

GrLivArea and SalePrice colums are highly correlated with each other 0.73 and 0.73

YearBuilt and GareageYrBlt colums are highly correlated with each other 0.78 and 0.78

Feature Engineering for train data

```
from sklearn.preprocessing import LabelEncoder,OneHotEncoder
```

```
for val in categorical_features:
    le=LabelEncoder()
    df[val]=le.fit_transform(df[val].astype(str))
```

```
df[categorical_features].head()
```

	MSZoning	Street	LotShape	LandContour	Utilities	LotConfig	LandSlope	Neighborhood	Condition1	Condition2	...	KitchenQual	Functional	FireplaceQu
0	3	1	0	3	0	4	0	13	2	2	...	3	6	4
1	3	1	0	3	0	4	1	12	2	2	...	2	6	4
2	3	1	0	3	0	1	0	15	2	2	...	3	6	4
3	3	1	0	3	0	4	0	14	2	2	...	3	6	4
4	3	1	0	3	0	2	0	14	2	2	...	2	6	4

5 rows × 39 columns

Feature Engineering for test data

```
for val in categorical_features:
    le=LabelEncoder()
    test_df[val]=le.fit_transform(test_df[val].astype(str))
```

```
test_df[categorical_features].head()
```

	MSZoning	Street	LotShape	LandContour	Utilities	LotConfig	LandSlope	Neighborhood	Condition1	Condition2	...	KitchenQual	Functional	FireplaceQu
0	2	1	0	1	0	0	0	21	2	0	...	2	5	2
1	2	1	0	3	0	1	0	21	2	0	...	2	5	0
2	2	1	3	3	0	4	0	4	2	0	...	0	5	4
3	2	1	3	0	0	4	0	5	2	0	...	1	5	2
4	2	1	0	3	0	1	0	20	1	0	...	2	5	2

5 rows × 39 columns

Outlier Removal for train data

```
from scipy.stats import zscore
import numpy as np
```

```
z=np.abs(zscore(df))
z
```

	MSSubClass	MSZoning	LotFrontage	LotArea	Street	LotShape	LandContour	Utilities	LotConfig	LandSlope	...	EnclosedPorch	3SsnPorch	Screer
0	1.508301	0.021646	0.148280	1.217608	0.058621	1.373107	0.318473	NaN	0.606420	0.226126	...	0.364375	0.125172	0.2
1	0.877042	0.021646	1.073586	1.097402	0.058621	1.373107	0.318473	NaN	0.606420	3.295414	...	0.364375	0.125172	3.1
2	0.077095	0.021646	0.971679	0.167657	0.058621	1.373107	0.318473	NaN	1.220661	0.226126	...	0.364375	0.125172	0.2
3	0.877042	0.021646	1.391434	0.503044	0.058621	1.373107	0.318473	NaN	0.606420	0.226126	...	0.364375	0.125172	0.2
4	0.877042	0.021646	0.148280	1.191243	0.058621	1.373107	0.318473	NaN	0.611634	0.226126	...	0.364375	0.125172	0.2
...	...	...	...	...	...	...	...	...	...	...	...	...	...	
1163	0.877042	0.021646	0.148280	0.147394	0.058621	1.373107	0.318473	NaN	0.606420	0.226126	...	0.364375	0.125172	0.2
1164	0.877042	0.021646	0.035362	0.074735	0.058621	0.752055	0.318473	NaN	0.606420	0.226126	...	0.364375	0.125172	0.2
1165	2.462438	0.021646	3.295786	2.743728	0.058621	0.752055	0.318473	NaN	0.611634	0.226126	...	0.364375	0.125172	0.2
1166	0.315629	4.762117	0.964829	0.138231	0.058621	0.752055	0.318473	NaN	0.606420	0.226126	...	2.358693	0.125172	0.2
1167	0.077095	0.021646	0.148280	0.292975	0.058621	1.373107	0.318473	NaN	0.606420	0.226126	...	0.364375	0.125172	0.2

1168 rows × 76 columns

The abs() function of Pythons standard library returns the absolute value of the given number. Absolute value of a number is the value without considering its sign. Hence absolute of 10 is 10, -10 is also 10. If the number is a complex number, abs() returns its magnitude.
It will make all the data positive

```
threshold=3
print(np.where(z>3))
```

```
(array([   1,    1,    1, ..., 1166, 1166, 1166], dtype=int64), array([ 9, 20, 34, ..., 62, 63, 75], dtype=int64))
```

```
Q1=df.quantile(0.25)
Q1
```

```
MSSubClass          20.000000
MSZoning             3.000000
LotFrontage          4.094345
LotArea              8.938727
Street               1.000000
                        ...
MoSold               5.000000
YrSold            2007.000000
SaleType             8.000000
SaleCondition        4.000000
SalePrice           11.778169
Name: 0.25, Length: 76, dtype: float64
```

```
Q3=df.quantile(0.75)
Q3
```

```
MSSubClass          70.000000
MSZoning             3.000000
LotFrontage          4.372593
LotArea              9.351449
Street               1.000000
                        ...
MoSold               8.000000
YrSold            2009.000000
SaleType             8.000000
SaleCondition        4.000000
SalePrice           12.278393
Name: 0.75, Length: 76, dtype: float64
```

```
IQR=Q3-Q1
IQR
```

```
MSSubClass        50.000000
MSZoning           0.000000
LotFrontage        0.278248
LotArea            0.412722
Street             0.000000
                      ...
MoSold             3.000000
YrSold             2.000000
SaleType           0.000000
SaleCondition      0.000000
SalePrice          0.500224
Length: 76, dtype: float64
```

```
# REMOVING OUTLIERS USING IQR METHOD
df_new = df[~((df < (Q1 - 1.5 * IQR)) |(df > (Q3 + 1.5 * IQR))).any(axis=1)]
df_new
```

	MSSubClass	MSZoning	LotFrontage	LotArea	Street	LotShape	LandContour	Utilities	LotConfig	LandSlope	...	EnclosedPorch	3SsnPorch	ScreenPo
45	20	3	4.330733	9.025696	1	3	3	0	0	0	...	0	0	
62	20	3	4.094345	8.857088	1	3	3	0	4	0	...	0	0	
70	60	3	4.174387	9.031931	1	0	3	0	4	0	...	0	0	
92	20	3	4.317488	9.161675	1	3	3	0	4	0	...	0	0	
110	60	3	4.262517	9.602585	1	0	3	0	1	0	...	0	0	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	
1040	60	3	4.369448	9.427063	1	3	3	0	4	0	...	0	0	
1043	60	3	4.248495	9.076809	1	3	3	0	4	0	...	0	0	
1070	20	3	4.382027	9.169518	1	3	3	0	0	0	...	0	0	
1109	60	3	4.219508	9.097731	1	0	3	0	4	0	...	0	0	
1167	60	3	4.262517	8.969669	1	0	3	0	4	0	...	0	0	

71 rows × 76 columns

```
print("shape before and after")
print("shape before".ljust(20),":", df.shape)
print("shape after".ljust(20),":", df_new.shape)
print("Percentage Loss".ljust(20),":", (df.shape[0]-df_new.shape[0])/df.shape[0])
```

```
shape before and after
shape before         : (1168, 76)
shape after          : (71, 76)
Percentage Loss      : 0.9392123287671232
```

Outlier Removal for test data

```
from scipy.stats import zscore
import numpy as np
```

```
z=np.abs(zscore(test_df))
z
```

	MSSubClass	MSZoning	LotFrontage	LotArea	Street	LotShape	LandContour	Utilities	LotConfig	LandSlope	...	OpenPorchSF	EnclosedPorch	3Ss
0	0.856054	0.287006	0.981605	0.263894	0.083045	1.402669	2.566101	0.058621	2.001106	0.226274	...	0.059897	0.341845	0
1	1.431981	0.287006	0.000000	0.363030	0.083045	1.402669	0.299297	0.058621	1.351136	0.226274	...	0.715738	0.341845	0
2	0.856054	0.287006	0.000000	0.089636	0.083045	0.745474	0.299297	0.058621	0.598774	0.226274	...	1.580750	0.341845	0
3	0.287963	0.287006	0.429998	0.101809	0.083045	0.745474	3.998799	0.058621	0.598774	0.226274	...	0.715738	0.341845	0
4	0.059160	0.287006	0.981605	0.297033	0.083045	1.402669	0.299297	0.058621	1.351136	0.226274	...	0.441985	0.341845	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	
287	0.856054	0.287006	0.580436	0.032999	0.083045	0.745474	0.299297	0.058621	0.598774	0.226274	...	0.789906	0.341845	0
288	0.856054	0.287006	0.472632	0.105212	0.083045	0.686621	0.299297	0.058621	0.598774	0.226274	...	0.715738	0.341845	0
289	0.856054	0.287006	0.000000	0.044775	0.083045	1.402669	0.299297	0.058621	1.351136	0.226274	...	0.715738	0.341845	0
290	0.169644	1.808136	0.823655	0.424197	0.083045	0.745474	0.299297	0.058621	2.001106	0.226274	...	0.350734	0.353083	0
291	2.347195	1.808136	2.277892	0.653160	0.083045	0.745474	0.299297	0.058621	0.598774	0.226274	...	0.715738	0.341845	0

292 rows × 75 columns

```
threshold=3
print(np.where(z>3))
```

```
Q1=test_df.quantile(0.25)
Q1
```

```
MSSubClass          20.00
MSZoning             2.00
LotFrontage         57.75
LotArea           7200.00
Street               1.00
                    ...
MiscVal              0.00
MoSold               4.00
YrSold            2007.00
SaleType             5.00
SaleCondition        2.00
Name: 0.25, Length: 75, dtype: float64
```

```
Q3=test_df.quantile(0.75)
Q3
```

```
MSSubClass          70.00
MSZoning             2.00
LotFrontage         76.00
LotArea          11658.75
Street               1.00
                    ...
MiscVal              0.00
MoSold               8.00
YrSold            2009.00
SaleType             5.00
SaleCondition        2.00
Name: 0.75, Length: 75, dtype: float64
```

```
IQR=Q3-Q1
IQR
```

```
MSSubClass          50.00
MSZoning             0.00
LotFrontage         18.25
LotArea           4458.75
Street               0.00
                    ...
MiscVal              0.00
MoSold               4.00
YrSold               2.00
SaleType             0.00
SaleCondition        0.00
Length: 75, dtype: float64
```

```
# REMOVING OUTLIERS USING IQR METHOD (be carefull with the variables used here)
df_new_test = test_df[~((test_df < (Q1 - 1.5 * IQR)) |(test_df > (Q3 + 1.5 * IQR))).any(axis=1)]
df_new_test.head()
```

	MSSubClass	MSZoning	LotFrontage	LotArea	Street	LotShape	LandContour	Utilities	LotConfig	LandSlope	...	OpenPorchSF	EnclosedPorch	3SsnPorc
16	20	2	66.425101	8780	1	0	3	0	0	0	...	0	0	
25	20	2	37.000000	6951	1	0	3	0	1	0	...	0	0	
27	20	2	78.000000	7800	1	3	3	0	4	0	...	98	0	
31	20	2	67.000000	10083	1	3	3	0	4	0	...	41	0	
57	60	2	63.000000	7875	1	3	3	0	4	0	...	75	0	

5 rows × 75 columns

```
print("shape before and after")
print("shape before".ljust(20),":", test_df.shape)
print("shape after".ljust(20),":", df_new_test.shape)
print("Percentage Loss".ljust(20),":", (test_df.shape[0]-df_new_test.shape[0])/test_df.shape[0])
```

```
shape before and after
shape before         : (292, 75)
shape after          : (24, 75)
Percentage Loss      : 0.9178082191780822
```

Now divide the train data in x,y

```
y=df_new["SalePrice"]
y
```

```
45      11.732061
62      12.078239
70      12.061047
92      11.877569
110     12.154779
           ...
1040    12.345835
1043    12.208570
1070    11.732061
1109    12.180755
1167    12.118334
Name: SalePrice, Length: 71, dtype: float64
```

```
x=df_new.drop(['SalePrice'],axis=1)
x
```

	MSSubClass	MSZoning	LotFrontage	LotArea	Street	LotShape	LandContour	Utilities	LotConfig	LandSlope	...	OpenPorchSF	EnclosedPorch	3SsnP
45	20	3	4.330733	9.025696	1	3	3	0	0	0	...	0	0	
62	20	3	4.094345	8.857088	1	3	3	0	4	0	...	64	0	
70	60	3	4.174387	9.031931	1	0	3	0	4	0	...	45	0	
92	20	3	4.317488	9.161675	1	3	3	0	4	0	...	0	0	
110	60	3	4.262517	9.602585	1	0	3	0	1	0	...	66	0	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	
1040	60	3	4.369448	9.427063	1	3	3	0	4	0	...	45	0	
1043	60	3	4.248495	9.076809	1	3	3	0	4	0	...	77	0	
1070	20	3	4.382027	9.169518	1	3	3	0	0	0	...	88	0	
1109	60	3	4.219508	9.097731	1	0	3	0	4	0	...	68	0	
1167	60	3	4.262517	8.969669	1	0	3	0	4	0	...	75	0	

71 rows × 75 columns

Feature selection of a train data

```
from sklearn.linear_model import Lasso
from sklearn.feature_selection import SelectFromModel
```

```
#apply fearure selection and specifying the lasso Regression model and selected a suitable alpha
#bigger the alpha less fearure that will be selected
# then i use the selectfrommodel object from sklearn which will select the fearures which cofficients are non-zero

feature_sel_model=SelectFromModel(Lasso(alpha=0.005,random_state=0)) #remember that to set the seed value of random s
feature_sel_model.fit(x,y)
```

```
SelectFromModel(estimator=Lasso(alpha=0.005, random_state=0))
```

```
feature_sel_model.get_support()
```

```
array([ True, False, False, False, False,  True, False, False,  True,
       False,  True, False, False, False, False, False,  True,  True,
        True, False, False,  True,  True, False,  True, False, False,
       False, False, False, False,  True,  True, False, False,  True,
        True, False,  True, False, False, False,  True, False, False,
       False, False, False, False, False, False, False,  True, False,
        True, False, False,  True,  True, False,  True, False, False,
       False,  True,  True, False, False, False, False, False,  True,
        True, False, False])
```

```
#lets print the number of total and selected features
selected_feat=x.columns[(feature_sel_model.get_support())]
print('total features: {}'.format((x.shape[1])))
print('selected features: {}'.format(len(selected_feat)))
print('features with coefficients shrank to zero: {}'.format(np.sum(feature_sel_model.estimator_.coef_ == 0)))
```

```
total features: 75
selected features: 25
features with coefficients shrank to zero: 50
```

```
selected_feat
```

```
Index(['MSSubClass', 'LotShape', 'LotConfig', 'Neighborhood', 'OverallCond',
       'YearBuilt', 'YearRemodAdd', 'Exterior1st', 'Exterior2nd', 'MasVnrArea',
       'BsmtFinType1', 'BsmtFinSF1', 'BsmtUnfSF', 'TotalBsmtSF', 'HeatingQC',
       '2ndFlrSF', 'TotRmsAbvGrd', 'Fireplaces', 'GarageYrBlt', 'GarageFinish',
       'GarageArea', 'WoodDeckSF', 'OpenPorchSF', 'MoSold', 'YrSold'],
      dtype='object')
```

```
x_train=x[selected_feat]
x_train
```

	MSSubClass	LotShape	LotConfig	Neighborhood	OverallCond	YearBuilt	YearRemodAdd	Exterior1st	Exterior2nd	MasVnrArea	...	2ndFlrSF	TotRmsAb
45	20	3	0	11	7	1982	1982	5	6	0.0	...	0	
62	20	3	4	7	5	2005	2005	11	12	0.0	...	0	
70	60	0	4	20	5	2004	2004	11	12	0.0	...	842	
92	20	3	4	5	6	1995	2006	11	12	0.0	...	0	
110	60	0	1	14	5	1971	1971	5	5	252.0	...	896	
...	...	...	...	...	...	...	...	...	...	...	...	...	
1040	60	3	4	5	5	2001	2001	11	12	0.0	...	896	
1043	60	3	4	5	5	1998	1998	11	12	116.0	...	878	
1070	20	3	0	12	6	1959	1959	7	7	132.0	...	0	
1109	60	0	4	5	5	2002	2002	11	12	95.0	...	829	
1167	60	0	4	8	5	2002	2003	11	12	0.0	...	702	

71 rows × 25 columns

Same feature can selected for test data

here i am selected the same features as selected the training data because train data and test data has indentical features

so here i am selectd the same features from test data also for predications the final results of a model

so here done all stepd with test data only last steps is predicting the final results of a model so all things is done up to here

test_x=df_new_test[selected_feat]

test_x

Remove the skewness' of train data

```
from sklearn.preprocessing import power_transform
```

```
df_new=power_transform(x,method="yeo-johnson")
df_new
```

```
array([[-0.93840956,  0.        ,  0.36167011, ..., -0.84224008,
         0.        ,  0.        ],
       [-0.93840956,  0.        , -1.12791802, ..., -0.02489336,
         0.        ,  0.        ],
       [ 1.10752374,  0.        , -0.6204652 , ..., -0.02489336,
         0.        ,  0.        ],
       ...,
       [-0.93840956,  0.        ,  0.68133883, ...,  0.79491131,
         0.        ,  0.        ],
       [ 1.10752374,  0.        , -0.33581058, ...,  0.79491131,
         0.        ,  0.        ],
       [ 1.10752374,  0.        , -0.06539644, ..., -1.65713502,
         0.        ,  0.        ]])
```

Remove the skewness' of test data

```
df_new_test=power_transform(df_new_test,method="yeo-johnson")
df_new_test
```

```
array([[-0.91752283,  0.        , -0.18502982, ...,  0.        ,
         0.        ,  0.        ],
       [-0.91752283,  0.        , -2.76877873, ...,  0.        ,
         0.        ,  0.        ],
       [-0.91752283,  0.        ,  1.24972001, ...,  0.        ,
         0.        ,  0.        ],
       ...,
       [ 1.051025  ,  0.        ,  0.85494962, ...,  0.        ,
         0.        ,  0.        ],
       [-0.91752283,  0.        , -0.34521635, ...,  0.        ,
         0.        ,  0.        ],
       [-0.91752283,  0.        ,  0.11352885, ...,  0.        ,
         0.        ,  0.        ]])
```

Now find out the vif of a train data

```
def vif_calc(x):
    vif=pd.DataFrame()
    vif['VIF Factor']=[variance_inflation_factor(x.values,i)for i in range(x.shape[1])]
    vif['variables']=x.columns
    return(vif)
```

```
vif_calc(x)
```

	VIF Factor	variables
0	40.077126	MSSubClass
1	0.000000	MSZoning
2	3.246895	LotFrontage
3	3.827019	LotArea
4	0.000000	Street
...	...	...
70	NaN	MiscVal
71	2.056773	MoSold
72	3.440631	YrSold
73	0.000000	SaleType
74	0.000000	SaleCondition

75 rows × 2 columns

The Variance Inflation Factor (VIF) is a measure of colinearity among predictor variables within a multiple regression. It is calculated by taking the the ratio of the variance of all a given models betas divide by the variane of a single beta if it were fit alone. The numerical value for VIF tells you (in decimal form) what percentage the variance (i.e. the standard error squared) is inflated for each coefficient. ... A rule of thumb for interpreting the variance inflation factor: 1 = not correlated. Between 1 and 5 = moderately correlated. Greater than 5 = highly correlated.

Observations:

here showing that in this columns YrSold,GarageCond,Garagequal,GarageYrBlt,TotRmsAbvGrd,YearRemodAdd,YearBlt,Condion2,Street,

LotArea,LotFrontage the vif is very very high this feature is highly correlated among predictor variables

Feature Scaling of a train data

In order to make all algorithms work properly with our data, we need to scale the features in our dataset. For that, we will use a helpful function named `StandardScaler()` from the popular Scikit-Learn Python package. This function standardizes features by subtracting the mean and scaling to unit variance. It works on each feature independently.

```
from sklearn.preprocessing import StandardScaler
```

```
sc=StandardScaler()
x=sc.fit_transform(x)
x
```

```
array([[-0.91516464,  0.        ,  0.35631787, ..., -0.84169781,
         0.        ,  0.        ],
       [-0.91516464,  0.        , -1.12611568, ..., -0.02306021,
         0.        ,  0.        ],
       [ 1.10745604,  0.        , -0.62415448, ..., -0.02306021,
         0.        ,  0.        ],
       ...,
       [-0.91516464,  0.        ,  0.6779867 , ...,  0.79557738,
         0.        ,  0.        ],
       [ 1.10745604,  0.        , -0.34119668, ...,  0.79557738,
         0.        ,  0.        ],
       [ 1.10745604,  0.        , -0.0714753 , ..., -1.6603354 ,
         0.        ,  0.        ]])
```

StandardScaler removes the mean and scales each feature/variable to unit variance. This operation is performed feature-wise in an independent way. StandardScaler can be influenced by outliers (if they exist in the dataset) since it involves the estimation of the empirical mean and standard deviation of each feature.

Feature Scaling of a test data

```
from sklearn.preprocessing import StandardScaler
```

```
sc=StandardScaler()
df_new_test=sc.fit_transform(df_new_test)
df_new_test
```

```
array([[-0.91752283,  0.        , -0.18502982, ...,  0.        ,
         0.        ,  0.        ],
       [-0.91752283,  0.        , -2.76877873, ...,  0.        ,
         0.        ,  0.        ],
       [-0.91752283,  0.        ,  1.24972001, ...,  0.        ,
         0.        ,  0.        ],
       ...,
       [ 1.051025  ,  0.        ,  0.85494962, ...,  0.        ,
         0.        ,  0.        ],
       [-0.91752283,  0.        , -0.34521635, ...,  0.        ,
         0.        ,  0.        ],
       [-0.91752283,  0.        ,  0.11352885, ...,  0.        ,
         0.        ,  0.        ]])
```

Prediction Type and Modeling Techniques

In this section, we choose the type of machine learning prediction that is suitable to our problem. We want to determine if this is a ragression problem or a classification problem. In this project, we want to predict the *price* of a house given information about it. The price we want to predict is a continuous

value; it can be any real number. This can be seen by looking at the target vatiable in our dataset `SalePrice`:

That means that the prediction type that is appropriate to our problem is **regression.**

Now we move to choose the modeling techniques we want to use. There are a lot of techniques available f or regression problems like Linear Regression, Ridge Regression, Artificial Neural Networks, Decision Trees, Random Forest, etc. In this project, we will test many modeling techniques, and then choose the technique(s) that yield the best results. The techniques that we will try are:

learning Machine models

Assumptions:

- This is a Linear Regression problem so we will use Regression methods.
- Train test split will be a 75:25 ratio respectively.

Models we will use:

- **LinearRegression**
- **Lasso**
- **Ridge**
- **ElasticNet**
- **DecisionTreeRegressor**
- **KNeighborRegressor**
- **SupportVectorRegressor**
- **SGDRegressor**
- **RandomForestRegressor**
- **AdaBoostRegressor**
- **GradientBoostRegressor**
- **XGBRegressor**

r2_score:

(coefficient of determination) regression score function. Best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a score of 0

In statistics, the coefficient of determination, denoted R^2 or r^2 and pronounced "R squared", is the proportion of the variation in the dependent variable that is predictable from the independent variable.

The coefficient of determination can also be found with the following formula: R2 = MSS/TSS = (TSS − RSS)/TSS, where MSS is the model sum of squares (also known as ESS, or explained sum of squares), which is the sum of the squares of the prediction from the linear regression minus the mean for that variable; TSS is the total sum of squares associated with the outcome variable, which is the sum of the squares of the measurements minus their mean; and RSS is the residual sum of squares, which is the sum of the squares of the measurements minus the prediction from the linear regression

coss_val_score:

Cross-validation is a statistical method used to estimate the skill of machine learning models. ... That k-fold cross validation is a procedure used to estimate the skill of the model on new data. There are common tactics that you can use to select the value of k for your dataset.

Mean_squared_error:

The mean squared error (MSE) tells you how close a regression line is to a set of points. It does this by taking the distances from the points to the regression line (these distances are the “errors”) and squaring them. The squaring is necessary to remove any negative signs. ... The lower the MSE, the better the forecast.

Mean_absolute_error:

In statistics, mean absolute error (MAE) is a measure of errors between paired observations expressing the same phenomenon. ... This is known as a scale-dependent accuracy measure and therefore cannot be used to make comparisons between series using different scales.

Root_mean_squared_error:

Root mean squared error (RMSE) is the square root of the mean of the square of all of the error. The use of RMSE is very common, and it is considered an excellent general-purpose error metric for numerical predictions.

LinearRegression:

Linear regression analysis is used to predict the value of a variable based on the value of another variable. The variable you want to predict is called the dependent variable. The variable you are using to predict the other variable's value is called the independent variable.In statistics, linear regression is a linear approach for modelling the relationship between a scalar response and one or more explanatory variables. The case of one explanatory variable is called simple linear regression; for more than one, the process is called multiple linear regression.

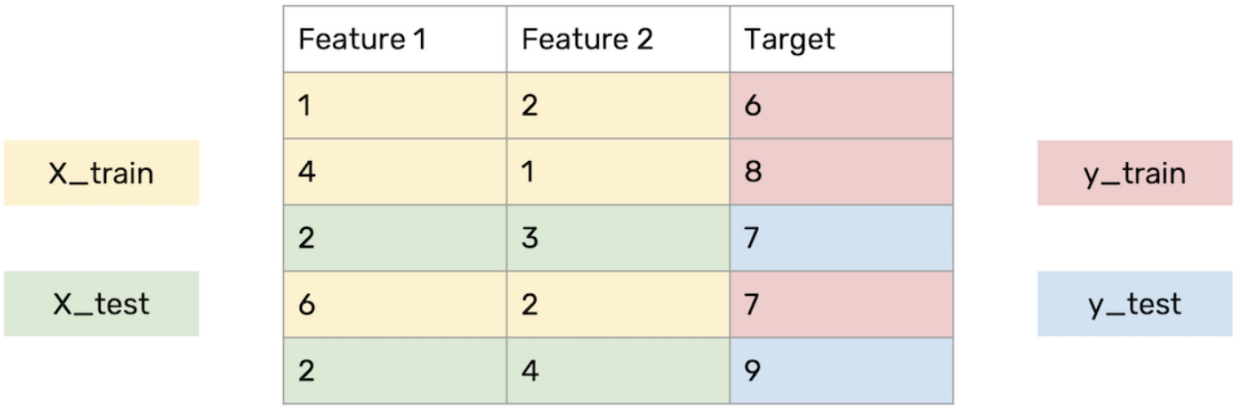
Splitting the Dataset

As usual for supervised machine learning problems, we need a training dataset to train our model and a test dataset to evaluate the model. So we will split our dataset randomly into two parts, one for training and the other for testing. For that, we will use another function from Scikit-Learn called `train_test_split()`:

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error,mean_absolute_error,r2_score
from sklearn.model_selection import cross_val_score
```

```
x_train,x_test,y_train,y_test=train_test_split(x_train,y_train,test_size=0.25,random_state=1)
```

We specified the size of the test set to be 25% of the whole dataset. This leaves 75% for the training dataset. Now we have four subsets: `X_train`, `X_test`, `y_train`, and `y_test`. Later we will use `X_train` and `y_train` to train our model, and `X_test` and `y_test` to test and evaluate the model. `X_train` and `X_test` represent features (predictors); `y_train` and `y_test` represent the target. From now on, we will refer to `X_train` and `y_train` as the training dataset, and to `X_test` and `y_test` as the test dataset. Figure 10 shows an example of what `train_test_split()` does.



Modeling Approach

For each one of the techniques mentioned in the previous section (Linear Regression, Nearest Neighbor, Support Vector Machines, etc.), we will follow these steps to build a model:

- Choose an algorithm that implements the corresponding technique
- Search for an effective parameter combination for the chosen algorithm
- Create a model using the found parameters
- Train (fit) the model on the training dataset
- Test the model on the test dataset and get the results

Searching for Effective Parameters

```
from sklearn.model_selection import GridSearchCV
```

```
parameters={'fit_intercept': (False,True),'normalize': (True, False),'copy_X': (False, True),'positive': (True,False)}
lr=LinearRegression()
gds=GridSearchCV(lr,parameters)
gds.fit(x_train,y_train)
```

```
GridSearchCV(estimator=LinearRegression(),
             param_grid={'copy_X': (False, True),
                         'fit_intercept': (False, True),
                         'normalize': (True, False),
                         'positive': (True, False)})
```

```
#sorted(gds,cv_results_.keys())
print(gds.best_params_)
```

```
{'copy_X': False, 'fit_intercept': False, 'normalize': True, 'positive': True}
```

```
param_grid={
    'n_jobs': list(range(1,100))}
```

```
lr=LinearRegression()
gds=GridSearchCV(lr,param_grid)
gds.fit(x_train,y_train)
```

```
GridSearchCV(estimator=LinearRegression(),
             param_grid={'n_jobs': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
                                    14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
                                    25, 26, 27, 28, 29, 30, ...]})
```

```
#sorted(gds,cv_results_.keys())
print(gds.best_params_)
```

```
{'n_jobs': 1}
```

```
lr=LinearRegression(copy_X=False,fit_intercept=True,normalize=True,positive=True,n_jobs=1)
lr.fit(x_train,y_train)
lr.coef_
```

```
array([5.18488594e-03, 0.00000000e+00, 0.00000000e+00, 4.11117187e-04,
       8.02536821e-02, 4.55510269e-03, 2.40533761e-03, 0.00000000e+00,
       0.00000000e+00, 0.00000000e+00, 1.43663331e-02, 1.45532687e-04,
       0.00000000e+00, 4.36930615e-04, 0.00000000e+00, 6.35283100e-05,
       2.69778101e-02, 1.00535728e-01, 0.00000000e+00, 0.00000000e+00,
       2.75117561e-04, 0.00000000e+00, 2.77530030e-04, 7.67894772e-03,
       1.37518104e-02])
```

```
lr.intercept_
```

```
-31.004170948244997
```

```
pred=lr.predict(x_test)
```

```
print("predicted value",pred)
print("actual value",y_test)
```

```
predicted value [12.23417299 12.14732595 12.16781632 12.01268224 12.22426216 12.216992
 12.25029354 11.69059571 11.91433653 12.46964762 12.16232805 12.68339529
 12.13324412 11.83449184 12.43605545 12.23815052 12.18806713 11.87210967]
actual value 1014    12.165251
424     12.273731
1043    12.208570
844     11.998433
427     12.247694
948     12.119970
750     12.332705
627     11.699405
748     11.970350
536     12.449019
70      12.061047
666     12.631014
829     12.139399
594     11.845820
724     12.449019
924     12.220469
996     12.146853
805     11.941456
Name: SalePrice, dtype: float64
```

```
print("error")
print("Mean absolute error of lr:",mean_absolute_error(y_test,pred))
print("Mean squared error of lr:",mean_squared_error(y_test,pred))
print("Root Mean squared error of lr:",np.sqrt(mean_squared_error(y_test,pred)))
```

```
error
Mean absolute error of lr: 0.04727769768497955
Mean squared error of lr: 0.003507961864515908
Root Mean squared error of lr: 0.05922804964301887
```

```
print(r2_score(y_test,pred))
print(lr.score(x_train,y_train))
```

```
0.9280885747686548
0.9597033651725712
```

```
scores = cross_val_score(lr, x_train, y_train, cv=5)
scores
```

```
array([0.90035096, 0.94701406, 0.68739392, 0.86104549, 0.76336172])
```

```
print("%0.2f accuracy with a standard deviation of %0.2f" % (scores.mean(), scores.std()))
```

```
0.83 accuracy with a standard deviation of 0.09
```

```
print(scores.mean())
print(scores.std())
```

```
0.8318332298760955
0.09419736819615351
```

RidgeRegression:

Ridge regression is a model tuning method that is used to analyse any data that suffers from multicollinearity. This method performs L2 regularization. When the issue of multicollinearity occurs, least-squares are unbiased, and variances are large, this results in predicted values to be far away from the actual value.

```
from sklearn.linear_model import Lasso,Ridge,ElasticNet
```

```
alphavalue={'alpha':[1,0.1,0.01,0.001,0.0001,0.005,0,1.0,1.1,1.5,1.2,1.3,1.4,1.6,1.7,1.8,1.9,1.10,0.10]}
rdg=Ridge()
gds=GridSearchCV(rdg,alphavalue)
gds.fit(x_train,y_train)
```

```
GridSearchCV(estimator=Ridge(),
             param_grid={'alpha': [1, 0.1, 0.01, 0.001, 0.0001, 0.005, 0, 1.0,
                                   1.1, 1.5, 1.2, 1.3, 1.4, 1.6, 1.7, 1.8, 1.9,
                                   1.1, 0.1]})
```

```
#sorted(gds,cv_results_.keys())
print(gds.best_params_)
```

```
{'alpha': 1.9}
```

```
parameters={'fit_intercept': [False,True],'normalize': [True, False],'copy_X': [False, True],'solver':['auto', 'svd', 'cholesky',
rdg=Ridge()
gds=GridSearchCV(rdg,parameters)
gds.fit(x_train,y_train)
```

```
GridSearchCV(estimator=Ridge(),
             param_grid={'copy_X': [False, True],
                         'fit_intercept': [False, True],
                         'normalize': [True, False],
                         'solver': ['auto', 'svd', 'cholesky', 'lsqr',
                                    'sparse_cg', 'sag', 'saga', 'lbfgs']})
```

```
#sorted(gds,cv_results_.keys())
print(gds.best_params_)
```

```
{'copy_X': True, 'fit_intercept': True, 'normalize': True, 'solver': 'sag'}
```

```
param_grid={
    'max_iter':[100,300,600,1000,1200,1500,2000],
    'random_state': list(range(1,100)),
    'tol':[0.1,0.01,0.001,0.0001,0.00001,1.0,0.14,0.18]
}
```

```
rdg=Ridge()
gds=GridSearchCV(rdg,param_grid)
gds.fit(x_train,y_train)
```

```
GridSearchCV(estimator=Ridge(),
             param_grid={'max_iter': [100, 300, 600, 1000, 1200, 1500, 2000],
                         'random_state': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
                                          13, 14, 15, 16, 17, 18, 19, 20, 21,
                                          22, 23, 24, 25, 26, 27, 28, 29, 30, ...],
                         'tol': [0.1, 0.01, 0.001, 0.0001, 1e-05, 1.0, 0.14,
                                 0.18]})
```

```
#sorted(gds,cv_results_.keys())
print(gds.best_params_)
```

```
{'max_iter': 100, 'random_state': 1, 'tol': 0.1}
```

```
rdg=Ridge(alpha=0.005,copy_X=True,fit_intercept=True,normalize=True,solver='sag',random_state=1,tol=0.1,max_iter=100)
rdg.fit(x_train,y_train)
rdg.score(x_train,y_train)
```

```
0.969646300572874
```

```
rdg.coef_
```

```
array([ 3.40106311e-03,  8.88425785e-03, -1.38056112e-03, -2.73743888e-03,
        7.81153041e-02,  3.10377007e-03,  3.36831129e-03, -1.92221494e-02,
        1.22626330e-02,  2.07834007e-04,  2.41252963e-02,  1.36181472e-04,
       -3.30613671e-05,  3.07937653e-04, -7.60855301e-03,  4.77294248e-05,
        2.13637895e-02,  9.60335477e-02,  1.16573492e-03, -2.82112200e-02,
        2.95524204e-04, -2.86357956e-04,  3.72874254e-04,  4.55582926e-03,
        1.88573517e-02])
```

```
pred=rdg.predict(x_test)
pred
```

```
array([12.20169657, 12.11472931, 12.22167005, 11.90840482, 12.29049961,
       12.2045616 , 12.25610117, 11.70539068, 11.83409511, 12.44913238,
       12.07513512, 12.67605029, 12.15003888, 11.65356393, 12.36488884,
       12.17958702, 12.18978694, 11.89941892])
```

```
r2_score(y_test,pred)
```

```
0.8633297421012975
```

```
print("Mean squared error of rdg:",mean_squared_error(y_test,pred))
print('Mean absolute error of rdg:',mean_absolute_error(y_test,pred))
print('Root Mean squared error of rdg:',np.sqrt(mean_squared_error(y_test,pred)))
```

```
Mean squared error of rdg: 0.006667008075279033
Mean absolute error of rdg: 0.06205198641074697
Root Mean squared error of rdg: 0.08165174875824176
```

```
scores = cross_val_score(rdg, x_train, y_train, cv=5)
scores
```

```
array([0.79384319, 0.94986076, 0.81988723, 0.71925444, 0.78864795])
```

```
print("%0.2f accuracy with a standard deviation of %0.2f" % (scores.mean(), scores.std()))
```

```
0.81 accuracy with a standard deviation of 0.08
```

```
print(scores.mean())
print(scores.std())
```

```
0.8142987158930023
0.07552231924965726
```

LassoRegresion:

Lasso regression is a regularization technique. It is used over regression methods for a more accurate prediction. This model uses shrinkage. Shrinkage is where data values are shrunk towards a central point as the mean. The lasso procedure encourages simple, sparse models (i.e. models with fewer parameters).

```
alphavalue={'alpha':[1,0.1,0.01,0.001,0.0001,0.005,0,1.0,1.1,1.2,1.3,1.4,1.5,1.6,1.8,1.9,1.0,0.10],'precompute':['auto',True,Fals
lso=Lasso()
gds=GridSearchCV(lso,alphavalue)
gds.fit(x_train,y_train)
```

```
GridSearchCV(estimator=Lasso(),
             param_grid={'alpha': [1, 0.1, 0.01, 0.001, 0.0001, 0.005, 0, 1.0,
                                   1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.8, 1.9, 1.0,
                                   0.1],
                         'precompute': ['auto', True, False]})
```

```
#sorted(gds,cv_results_.keys())
print(gds.best_params_)
```

```
{'alpha': 0.005, 'precompute': False}
```

```
parameters={'fit_intercept': (False,True),'normalize': (True, False),'copy_X': (False, True),'positive': (True,False),'selection'
lso=Lasso()
gds=GridSearchCV(lso,parameters)
gds.fit(x_train,y_train)
```

```
GridSearchCV(estimator=Lasso(),
             param_grid={'copy_X': (False, True),
                         'fit_intercept': (False, True),
                         'normalize': (True, False), 'positive': (True, False),
                         'selection': ('cyclic', 'random')})
```

```
#sorted(gds,cv_results_.keys())
print(gds.best_params_)
```

```
{'copy_X': True, 'fit_intercept': False, 'normalize': True, 'positive': False, 'selection': 'random'}
```

```
param_grid={
    'max_iter':[100,300,600,1000,1200,1500,2000],
    'random_state': list(range(1,100)),
    'tol':[0.1,0.01,0.001,0.0001,0.00001,1.0,0.14,0.18],
    'warm_start':[True,False]
}
```

```
lso=Lasso()
gds=GridSearchCV(lso,param_grid)
gds.fit(x_train,y_train)
```

```
GridSearchCV(estimator=Lasso(),
             param_grid={'max_iter': [100, 300, 600, 1000, 1200, 1500, 2000],
                         'random_state': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
                                          13, 14, 15, 16, 17, 18, 19, 20, 21,
                                          22, 23, 24, 25, 26, 27, 28, 29, 30, ...],
                         'tol': [0.1, 0.01, 0.001, 0.0001, 1e-05, 1.0, 0.14,
                                 0.18],
                         'warm_start': [True, False]})
```

```
#sorted(gds,cv_results_.keys())
print(gds.best_params_)
```

```
{'max_iter': 100, 'random_state': 1, 'tol': 0.1, 'warm_start': True}
```

```
lso=Lasso(alpha=0.005,copy_X=True,fit_intercept=True,normalize=True,positive=False,selection='random',random_state=1,tol=0.1,warm
lso.fit(x_train,y_train)
lso.score(x_train,y_train)
predlso=lso.predict(x_test)
r2_score(y_test,predlso)
```

```
0.7902340996316474
```

```
print(lso.score(x_train,y_train))
```

```
0.8511069768064317
```

```
print("Mean squared error of lso:",mean_squared_error(y_test,predlso))
print('Mean absolute error of lso:',mean_absolute_error(y_test,predlso))
print('Root Mean squared error of lso:',np.sqrt(mean_squared_error(y_test,predlso)))
```

```
Mean squared error of lso: 0.010232738074662408
Mean absolute error of lso: 0.07734423634748418
Root Mean squared error of lso: 0.1011569971611574
```

```
scores = cross_val_score(lso, x_train, y_train, cv=5)
scores
```

```
array([0.71903393, 0.67229744, 0.89358927, 0.85206363, 0.55604184])
```

```
print("%0.2f accuracy with a standard deviation of %0.2f" % (scores.mean(), scores.std()))
```

```
0.74 accuracy with a standard deviation of 0.12
```

```
print(scores.mean())
print(scores.std())
```

```
0.7386052210244775
0.12247613125449826
```

ElasticNetRegression:

Elastic net is a popular type of regularized linear regression that combines two popular penalties, specifically the L1 and L2 penalty functions. ... Elastic Net is an extension of linear regression that adds regularization penalties to the loss function during training.

```
alphavalue={'alpha':[1,0.1,0.01,0.001,0.0001,0.005],'precompute':[True,False],'l1_ratio':[1.0,1.1,1.2,1.3,1.4,1.5,0.0,0.1,0.2,0.
enr=ElasticNet()
gds=GridSearchCV(enr,alphavalue)
gds.fit(x_train,y_train)
```

```
GridSearchCV(estimator=ElasticNet(),
             param_grid={'alpha': [1, 0.1, 0.01, 0.001, 0.0001, 0.005],
                         'l1_ratio': [1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 0.0, 0.1,
                                      0.2, 0.3, 0.4, 0.5, 0.6],
                         'precompute': [True, False]})
```

```
#sorted(gds,cv_results_.keys())
print(gds.best_params_)
```

```
{'alpha': 0.1, 'l1_ratio': 0.0, 'precompute': False}
```

```
parameters={'fit_intercept': (False,True),'normalize': (True, False),'copy_X': (False, True),'positive': (True,False),'selectior
enr=ElasticNet()
gds=GridSearchCV(enr,parameters)
gds.fit(x_train,y_train)
```

```
GridSearchCV(estimator=ElasticNet(),
             param_grid={'copy_X': (False, True),
                         'fit_intercept': (False, True),
                         'normalize': (True, False), 'positive': (True, False),
                         'selection': ('cyclic', 'random')})
```

```
#sorted(gds,cv_results_.keys())
print(gds.best_params_)
```

```
{'copy_X': False, 'fit_intercept': False, 'normalize': True, 'positive': False, 'selection': 'random'}
```

```
param_grid={
    'max_iter':[100,300,600,1000,1200,1500,2000],
    'random_state': list(range(1,100)),
    'tol':[0.1,0.01,0.001,0.0001,0.00001,1.0,0.14,0.18],
    'warm_start':[bool,False]
}
```

```
enr=ElasticNet()
gds=GridSearchCV(enr,param_grid)
gds.fit(x_train,y_train)
```

```
GridSearchCV(estimator=ElasticNet(),
             param_grid={'max_iter': [100, 300, 600, 1000, 1200, 1500, 2000],
                         'random_state': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
                                          13, 14, 15, 16, 17, 18, 19, 20, 21,
                                          22, 23, 24, 25, 26, 27, 28, 29, 30, ...],
                         'tol': [0.1, 0.01, 0.001, 0.0001, 1e-05, 1.0, 0.14,
                                 0.18],
                         'warm_start': [<class 'bool'>, False]})
```

```
#sorted(gds,cv_results_.keys())
print(gds.best_params_)
```

```
{'max_iter': 100, 'random_state': 1, 'tol': 0.18, 'warm_start': <class 'bool'>}
```

```
enr=ElasticNet(alpha=0.005,copy_X=False,fit_intercept=True,normalize=True,positive=False,selection='cyclic',random_state=1,max_it
#enr=ElasticNet()
enr.fit(x_train,y_train)
enrpred=enr.predict(x_test)
print(enr.score(x_train,y_train))
```

```
0.9528778232281973
```

```
r2_score(y_test,enrpred)
```

```
0.891300021784052
```

```
enr.coef_
```

```
array([ 1.43439087e-03,  2.58858977e-03, -1.61896821e-03,  6.54608933e-04,
        5.22936225e-02,  2.41825534e-03,  2.57765991e-03, -6.52836937e-03,
        3.00618266e-03,  1.94688461e-04,  1.19217075e-02,  8.59553995e-05,
       -6.33043379e-06,  2.37491838e-04, -1.61667053e-02,  9.04706397e-05,
        3.34553379e-02,  6.37928398e-02,  1.67660533e-03, -4.03673349e-02,
        2.75945780e-04, -1.62350739e-04,  4.02000471e-04,  3.85401949e-03,
        7.21571942e-03])
```

```
print("Mean squared error of enr:",mean_squared_error(y_test,enrpred))
print('Mean absolute error of enr:',mean_absolute_error(y_test,enrpred))
print('Root Mean squared error of enr:',np.sqrt(mean_squared_error(y_test,enrpred)))
```

```
Mean squared error of enr: 0.005302570169184268
Mean absolute error of enr: 0.05326748128308228
Root Mean squared error of enr: 0.07281874874772477
```

```
scores = cross_val_score(enr, x_train, y_train, cv=5)
scores
```

```
array([0.92697198, 0.89296222, 0.90325543, 0.80311694, 0.75490718])
```

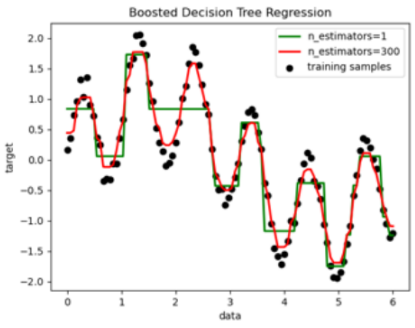
```
print("%0.2f accuracy with a standard deviation of %0.2f" % (scores.mean(), scores.std()))
```

```
0.86 accuracy with a standard deviation of 0.07
```

Approaching more regressor

DecisionTreeRegressor:

Decision Tree - Regression. Decision tree builds regression or classification models in the form of a tree structure. It breaks down a dataset into smaller and smaller subsets while at the same time an associated decision tree is incrementally developed. The final result is a tree with decision nodes and leaf nodes.



```
from sklearn.tree import DecisionTreeRegressor
from sklearn.neighbors import KNeighborsRegressor
from sklearn.svm import SVR
```

```
param_grid = {'max_depth': [3, 5, 10],
              'min_samples_split': [2, 5, 10],
              'min_samples_leaf': [1,2,3],
              'random_state': list(range(1,100))}
```

```
dtr=DecisionTreeRegressor()
gds=GridSearchCV(dtr,param_grid)
gds.fit(x_train,y_train)
```

```
GridSearchCV(estimator=DecisionTreeRegressor(),
             param_grid={'max_depth': [3, 5, 10], 'min_samples_leaf': [1, 2, 3],
                         'min_samples_split': [2, 5, 10],
                         'random_state': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
                                          13, 14, 15, 16, 17, 18, 19, 20, 21,
                                          22, 23, 24, 25, 26, 27, 28, 29, 30, ...]})
```

```
#sorted(gds,cv_results_.keys())
print(gds.best_params_)
```

```
{'max_depth': 5, 'min_samples_leaf': 1, 'min_samples_split': 2, 'random_state': 55}
```

```
parameters={'criterion':('mse', 'friedman_mse', 'mae', 'poisson'),'splitter':('best', 'random'),'max_features': ('auto
dtr=DecisionTreeRegressor()
gds=GridSearchCV(dtr,parameters)
gds.fit(x_train,y_train)
```

```
GridSearchCV(estimator=DecisionTreeRegressor(),
             param_grid={'criterion': ('mse', 'friedman_mse', 'mae', 'poisson'),
                         'max_features': ('auto', 'sqrt', 'log2'),
                         'splitter': ('best', 'random')})
```

```
#sorted(gds,cv_results_.keys())
print(gds.best_params_)
```

```
{'criterion': 'poisson', 'max_features': 'auto', 'splitter': 'random'}
```

```
param_grid={
    'max_leaf_nodes':[10,20,30,40,50,60,70,80,90,100,None],
    'min_weight_fraction_leaf':[0.0,0.1,0.001,0.0001],
    'min_impurity_decrease':[0.0,0.1,0.001,0.0001],
    'ccp_alpha':[0.1,0.001,0.0001,.1,1.0]
}
```

```
dtr=DecisionTreeRegressor()
gds=GridSearchCV(dtr,param_grid)
gds.fit(x_train,y_train)
```

```
dtr=DecisionTreeRegressor()
gds=GridSearchCV(dtr,param_grid)
gds.fit(x_train,y_train)
```

```
GridSearchCV(estimator=DecisionTreeRegressor(),
             param_grid={'ccp_alpha': [0.1, 0.001, 0.0001, 0.1, 1.0],
                         'max_leaf_nodes': [10, 20, 30, 40, 50, 60, 70, 80, 90,
                                            100, None],
                         'min_impurity_decrease': [0.0, 0.1, 0.001, 0.0001],
                         'min_weight_fraction_leaf': [0.0, 0.1, 0.001, 0.0001]})
```

```
#sorted(gds,cv_results_.keys())
print(gds.best_params_)
```

```
{'ccp_alpha': 0.0001, 'max_leaf_nodes': 80, 'min_impurity_decrease': 0.0, 'min_weight_fraction_leaf': 0.0}
```

```
dtr=DecisionTreeRegressor(criterion='poisson',max_features='auto',splitter='random',max_depth=10,min_samples_split=2,min_samples_
dtr.fit(x_train,y_train)
```

```
DecisionTreeRegressor(ccp_alpha=0.0001, criterion='poisson', max_depth=10,
                      max_features='auto', max_leaf_nodes=80, random_state=55,
                      splitter='random')
```

```
print('dtr_score:',dtr.score(x_train,y_train))
dtrpredict=dtr.predict(x_test)
```

```
dtr_score: 0.5506832920015432
```

```
print('dtr r2_score:',r2_score(y_test,dtrpredict))
```

```
dtr r2_score: 0.5358983759793055
```

```
print("Mean squared error of dtr:",mean_squared_error(y_test,dtrpredict))
print('Mean absolute error of dtr:',mean_absolute_error(y_test,dtrpredict))
print('Root Mean squared error of dtr:',np.sqrt(mean_squared_error(y_test,dtrpredict)))
```

```
Mean squared error of dtr: 0.022639668078986326
Mean absolute error of dtr: 0.11913352470287414
Root Mean squared error of dtr: 0.15046484000917398
```

```
scores = cross_val_score(dtr, x_train, y_train, cv=5)
scores
```

```
array([0.60539678, 0.4382592 , 0.84654741, 0.66115659, 0.73251637])
```

```
print("%0.2f accuracy with a standard deviation of %0.2f" % (scores.mean(), scores.std()))
```

```
0.66 accuracy with a standard deviation of 0.14
```

```
print(scores.mean())
print(scores.std())
```

```
0.6567752714707283
0.13576332176834868
```

KNeighborRegressor:

KNN works by finding the distances between a query and all the examples in the data, selecting the specified number examples (K) closest to the query, then votes for the most frequent label (in the case of classification) or averages the labels (in the case of regression).

```
param_grid = {'n_neighbors': list(range(1,30)),
              'leaf_size': list(range(1,50)),
               'p': [1,2,3]}
```

```
knr=KNeighborsRegressor()
gds=GridSearchCV(knr,param_grid)
gds.fit(x_train,y_train)
```

```
GridSearchCV(estimator=KNeighborsRegressor(),
             param_grid={'leaf_size': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
                                       13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
                                       23, 24, 25, 26, 27, 28, 29, 30, ...],
                         'n_neighbors': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
                                         13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
                                         23, 24, 25, 26, 27, 28, 29],
                         'p': [1, 2, 3]})
```

```
#sorted(gds,cv_results_.keys())
print(gds.best_params_)
```

```
{'leaf_size': 1, 'n_neighbors': 3, 'p': 2}
```

```
parameters={'weights':('uniform', 'distance'),'algorithm':('auto', 'ball_tree', 'kd_tree', 'brute')}
knr=KNeighborsRegressor()
gds=GridSearchCV(knr,parameters)
gds.fit(x_train,y_train)
```

```
GridSearchCV(estimator=KNeighborsRegressor(),
             param_grid={'algorithm': ('auto', 'ball_tree', 'kd_tree', 'brute'),
                         'weights': ('uniform', 'distance')})
```

```
#sorted(gds,cv_results_.keys())
print(gds.best_params_)
```

```
{'algorithm': 'ball_tree', 'weights': 'distance'}
```

```
param_grid={
    'n_jobs':[1,2,3,4,5,6,7,8,9,10,-1,None],
    'metric_params':[dict,None],
    'metric' :['standard Euclidean metric','minkowski','DistanceMetric']
}
```

```
knr=KNeighborsRegressor()
gds=GridSearchCV(knr,param_grid)
gds.fit(x_train,y_train)
```

```
GridSearchCV(estimator=KNeighborsRegressor(),
             param_grid={'metric': ['standard Euclidean metric', 'minkowski',
                                    'DistanceMetric'],
                         'metric_params': [<class 'dict'>, None],
                         'n_jobs': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, -1, None]})
```

```
#sorted(gds,cv_results_.keys())
print(gds.best_params_)
```

```
{'metric': 'minkowski', 'metric_params': None, 'n_jobs': 1}
```

```
knr=KNeighborsRegressor(algorithm='ball_tree',weights='distance',n_neighbors=3,leaf_size=1,p=2,metric_params=None,n_jobs=1,metric
knr.fit(x_train,y_train)
print('knr score:',knr.score(x_train,y_train))
```

```
knr score: 1.0
```

```
knrpredict=knr.predict(x_test)
print('knr r2_score:',r2_score(y_test,knrpredict))
```

```
knr r2_score: 0.5243377166948167
```

```
enr.coef_
```

```
array([ 1.33570651e-03,  5.77566667e-04,  2.25212532e-03, -1.12853754e-04,
        4.07784138e-02,  2.53104769e-03,  2.58239236e-03, -5.60406956e-03,
        2.03330036e-03,  2.09540182e-04,  6.34295665e-03,  8.56505598e-05,
        6.42403240e-06,  2.02285342e-04, -1.46439129e-02,  8.47188348e-05,
        3.35293541e-02,  7.00670453e-02,  1.21505810e-03, -3.73129264e-02,
        2.88021288e-04, -1.64859998e-04,  5.22163774e-04,  4.63577404e-03,
        1.05121372e-02])
```

```
print("Mean squared error of enr:",mean_squared_error(y_test,enrpred))
print('Mean absolute error of enr:',mean_absolute_error(y_test,enrpred))
print('Root Mean squared error of enr:',np.sqrt(mean_squared_error(y_test,enrpred)))
```

```
Mean squared error of enr: 0.005467694740823721
Mean absolute error of enr: 0.056014605419296454
Root Mean squared error of enr: 0.07394386209026224
```

```
scores = cross_val_score(knr, x_train, y_train, cv=5)
scores
```

```
array([0.59387528, 0.69952573, 0.67885008, 0.70539061, 0.38492017])
```

```
print("%0.2f accuracy with a standard deviation of %0.2f" % (scores.mean(), scores.std()))
```

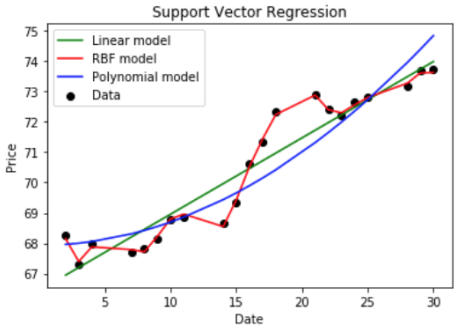
```
0.61 accuracy with a standard deviation of 0.12
```

```
print(scores.mean())
print(scores.std())
```

```
0.6125123765114524
0.1206183516952927
```

SupportVectorRegressor:

Support Vector Regression is a supervised learning algorithm that is used to predict discrete values. Support Vector Regression uses the same principle as the SVMs. The basic idea behind SVR is to find the best fit line. In SVR, the best fit line is the hyperplane that has the maximum number of points.



```
parameters={'kernel':('linear','rbf','poly'),'C':[1, 10,20,30,40,50,60,70,80,90,100],'gamma':('scale', 'auto'),"epsilon":[0.1,0.2
svr=SVR()
gds=GridSearchCV(svr,parameters)
gds.fit(x_train,y_train)
```

```
GridSearchCV(estimator=SVR(),
             param_grid={'C': [1, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100],
                         'epsilon': [0.1, 0.2, 0.3, 0.4, 0.5],
                         'gamma': ('scale', 'auto'),
                         'kernel': ('linear', 'rbf', 'poly')})
```

```
#sorted(gds,cv_results_.keys())
print(gds.best_params_)
```

```
{'C': 1, 'epsilon': 0.1, 'gamma': 'scale', 'kernel': 'poly'}
```

```
param_grid={
    'max_iter':[100,300,600,1000,1200,1500,2000,-1],
    'tol':[0.1,0.01,0.001,0.0001,0.00001,1.0],
    'cache_size':[100,200,300,400,500,600,700,800,900,1000],
    'shrinking':[bool,True],
    'coef0':[0.0,0.1,0.001,0.0001,1.0,1]
}
```

```
svr=SVR()
gds=GridSearchCV(svr,param_grid)
gds.fit(x_train,y_train)
```

```
GridSearchCV(estimator=SVR(),
```

```
svr=SVR()
gds=GridSearchCV(svr,param_grid)
gds.fit(x_train,y_train)
```

```
GridSearchCV(estimator=SVR(),
             param_grid={'cache_size': [100, 200, 300, 400, 500, 600, 700, 800,
                                        900, 1000],
                         'coef0': [0.0, 0.1, 0.001, 0.0001, 1.0, 1],
                         'max_iter': [100, 300, 600, 1000, 1200, 1500, 2000,
                                      -1],
                         'shrinking': [<class 'bool'>, True],
                         'tol': [0.1, 0.01, 0.001, 0.0001, 1e-05, 1.0]})
```

```
#sorted(gds,cv_results_.keys())
print(gds.best_params_)
```

```
{'cache_size': 100, 'coef0': 0.0, 'max_iter': 100, 'shrinking': True, 'tol': 1e-05}
```

```
svr=SVR(kernel="poly",C=1,gamma='scale',epsilon=0.1,degree=3,cache_size=100,coef0=0.0,max_iter=100,shrinking=True,tol=1e-05)
svr.fit(x_train,y_train)
print('svr score:',svr.score(x_train,y_train))
```

```
svr score: 0.8035618059062937
```

```
svrpredict=svr.predict(x_test)
print('svr r2_score:',r2_score(y_test,svrpredict))
```

```
svr r2_score: 0.9236812501885632
```

```
print('Mean absolute error of svr:',mean_absolute_error(y_test,svrpredict))
print('Mean squared error of svr:',mean_squared_error(y_test,svrpredict))
print('Root Mean squared error of svr:',np.sqrt(mean_squared_error(y_test,svrpredict)))
```

```
Mean absolute error of svr: 0.048080581128920365
Mean squared error of svr: 0.0037229586678996112
Root Mean squared error of svr: 0.06101605254274985
```

```
scores = cross_val_score(svr, x_train, y_train, cv=5)
scores
```

```
array([0.76569642, 0.79948498, 0.72502004, 0.8580113 , 0.52942971])
```

```
print("%0.2f accuracy with a standard deviation of %0.2f" % (scores.mean(), scores.std()))
```

```
print("%0.2f accuracy with a standard deviation of %0.2f" % (scores.mean(), scores.std()))
```

```
0.74 accuracy with a standard deviation of 0.11
```

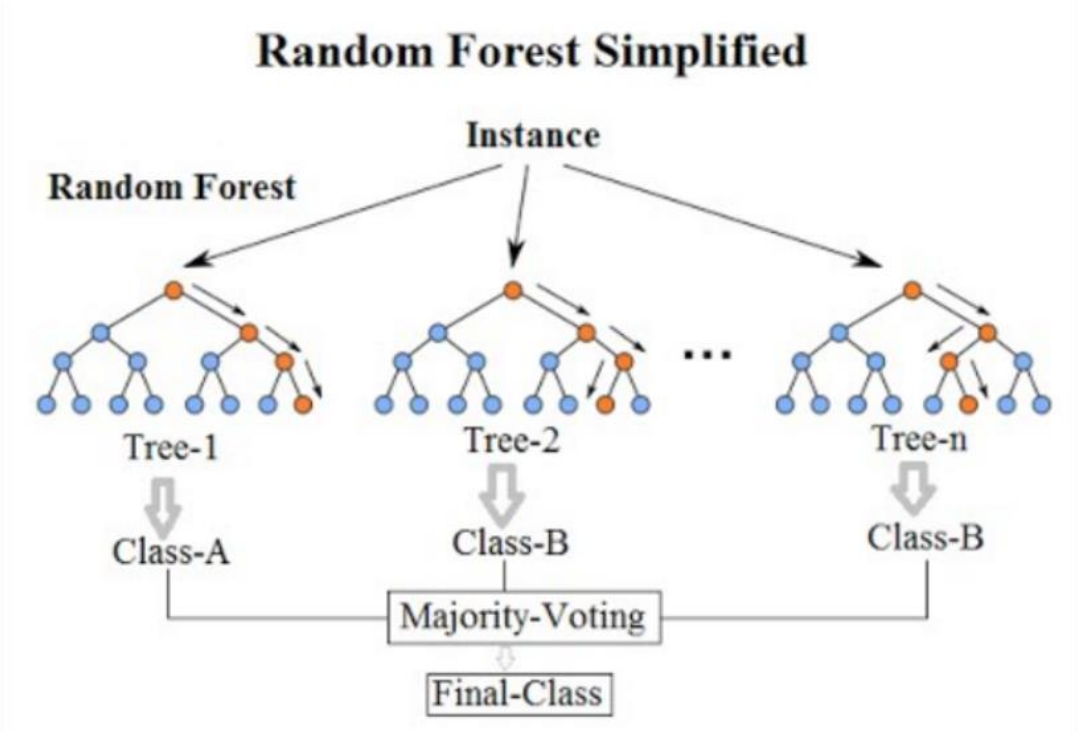
```
print(scores.mean())
print(scores.std())
```

```
0.7355284881798585
0.11188360875969454
```

Approaching Ensemble Method

RandomForestRegressor:

Random forest is a flexible, easy to use machine learning algorithm that produces, even without hyper-parameter tuning, a great result most of the time. It is also one of the most used algorithms, because of its simplicity and diversity (it can be used for both classification and regression tasks).



```
from sklearn.ensemble import RandomForestRegressor
```

```
param_grid={
 'bootstrap': [True, False],
 'max_depth': [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, None],
 'max_features': ['auto', 'sqrt','log2'],
 'min_samples_leaf': [1, 2, 4]}
```

```
rfr=RandomForestRegressor()
gds=GridSearchCV(rfr,param_grid)
gds.fit(x_train,y_train)
```

```
GridSearchCV(estimator=RandomForestRegressor(),
             param_grid={'bootstrap': [True, False],
                         'max_depth': [10, 20, 30, 40, 50, 60, 70, 80, 90, 100,
                                       None],
                         'max_features': ['auto', 'sqrt', 'log2'],
                         'min_samples_leaf': [1, 2, 4]})
```

```
#sorted(gds,cv_results_.keys())
print(gds.best_params_)
```

```
{'bootstrap': False, 'max_depth': 30, 'max_features': 'log2', 'min_samples_leaf': 1}
```

```
param_grid={
 'min_samples_split': [2, 5, 10],
 'n_estimators': [200, 400, 600, 800, 1000, 1200, 1400, 1600, 1800, 2000]}
```

```
rfr=RandomForestRegressor()
gds=GridSearchCV(rfr,param_grid)
gds.fit(x_train,y_train)
```

```
GridSearchCV(estimator=RandomForestRegressor(),
             param_grid={'min_samples_split': [2, 5, 10],
                         'n_estimators': [200, 400, 600, 800, 1000, 1200, 1400,
                                          1600, 1800, 2000]})
```

```
#sorted(gds,cv_results_.keys())
print(gds.best_params_)
```

```
{'min_samples_split': 2, 'n_estimators': 400}
```

```
param_grid={
    'criterion' :['squared_error', 'mse', 'absolute_error', 'poisson']}
```

```
rfr=RandomForestRegressor()
gds=GridSearchCV(rfr,param_grid)
gds.fit(x_train,y_train)
```

```
rfr=RandomForestRegressor()
gds=GridSearchCV(rfr,param_grid)
gds.fit(x_train,y_train)
```

```
GridSearchCV(estimator=RandomForestRegressor(),
             param_grid={'criterion': ['squared_error', 'mse', 'absolute_error',
                                       'poisson']})
```

```
#sorted(gds,cv_results_.keys())
print(gds.best_params_)
```

```
{'criterion': 'mse'}
```

```
param_grid={
    'max_leaf_nodes':[10,20,30,40,50,60,70,80,90,100,None],
    'n_jobs':[1,2,3,4,5,6,7,8,9.10,None]}
```

```
rfr=RandomForestRegressor()
gds=GridSearchCV(rfr,param_grid)
gds.fit(x_train,y_train)
```

```
GridSearchCV(estimator=RandomForestRegressor(),
             param_grid={'max_leaf_nodes': [10, 20, 30, 40, 50, 60, 70, 80, 90,
                                            100, None],
                         'n_jobs': [1, 2, 3, 4, 5, 6, 7, 8, 9.1, None]})
```

```
#sorted(gds,cv_results_.keys())
print(gds.best_params_)
```

```
{'max_leaf_nodes': 80, 'n_jobs': 5}
```

```
param_grid={
    'random_state': list(range(1,100)),
    'warm_start':[bool,False],
    'max_samples':[0,1,None]
}
```

```
rfr=RandomForestRegressor()
gds=GridSearchCV(rfr,param_grid)
gds.fit(x_train,y_train)
```

```
GridSearchCV(estimator=RandomForestRegressor(),
             param_grid={'max_samples': [0, 1, None],
                         'random_state': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
                                          13, 14, 15, 16, 17, 18, 19, 20, 21,
                                          22, 23, 24, 25, 26, 27, 28, 29, 30, ...],
                         'warm_start': [<class 'bool'>, False]})
```

```
#sorted(gds,cv_results_.keys())
print(gds.best_params_)
```

```
{'max_samples': None, 'random_state': 77, 'warm_start': <class 'bool'>}
```

```
param_grid={
    'min_weight_fraction_leaf':[0.0,0.1,0.001,0.0001],
    'min_impurity_decrease':[0.0,0.1,0.001,0.0001],
    'oob_score':[True,False],
    'ccp_alpha':[0.1,0.001,0.0001,.1,1.0]


}
```

```
rfr=RandomForestRegressor()
gds=GridSearchCV(rfr,param_grid)
gds.fit(x_train,y_train)
```

```
GridSearchCV(estimator=RandomForestRegressor(),
             param_grid={'ccp_alpha': [0.1, 0.001, 0.0001, 0.1, 1.0],
                         'min_impurity_decrease': [0.0, 0.1, 0.001, 0.0001],
                         'min_weight_fraction_leaf': [0.0, 0.1, 0.001, 0.0001],
                         'oob_score': [<class 'bool'>, False]})
```

```
#sorted(gds,cv_results_.keys())
print(gds.best_params_)
```

```
{'ccp_alpha': 0.0001, 'min_impurity_decrease': 0.0001, 'min_weight_fraction_leaf': 0.001, 'oob_score': False}
```

```
rfr=RandomForestRegressor(criterion='mse',max_features='log2',n_estimators=400,max_leaf_nodes=80,n_jobs=5,min_samples_split=2,boo
#RandomForestRegressor(100)by default
```

```
rfr.fit(x_train,y_train)
print('rfr score:',rfr.score(x_train,y_train))
```

```
rfr score: 0.9965012242051895
```

```
perdrfr=rfr.predict(x_test)
print('rfr r2_score:',r2_score(y_test,perdrfr))
```

```
rfr r2_score: 0.5916280015101456
```

```
perdrfr=rfr.predict(x_test)
print('rfr r2_score:',r2_score(y_test,perdrfr))
```

```
rfr r2_score: 0.5916280015101456
```

```
print("Mean absolue error of rfr:",mean_absolute_error(y_test,perdrfr))
print("Mean squared error of rfr:",mean_squared_error(y_test,perdrfr))
print("Root Mean squared error of rfr:",np.sqrt(mean_squared_error(y_test,perdrfr)))
```

```
Mean absolue error of rfr: 0.0985751128758653
Mean squared error of rfr: 0.019921081978696742
Root Mean squared error of rfr: 0.14114206310911268
```

```
scores = cross_val_score(rfr, x_train, y_train, cv=5)
scores
```

```
array([0.78019829, 0.64147083, 0.94882371, 0.88991829, 0.53697158])
```

```
print("%0.2f accuracy with a standard deviation of %0.2f" % (scores.mean(), scores.std()))
```

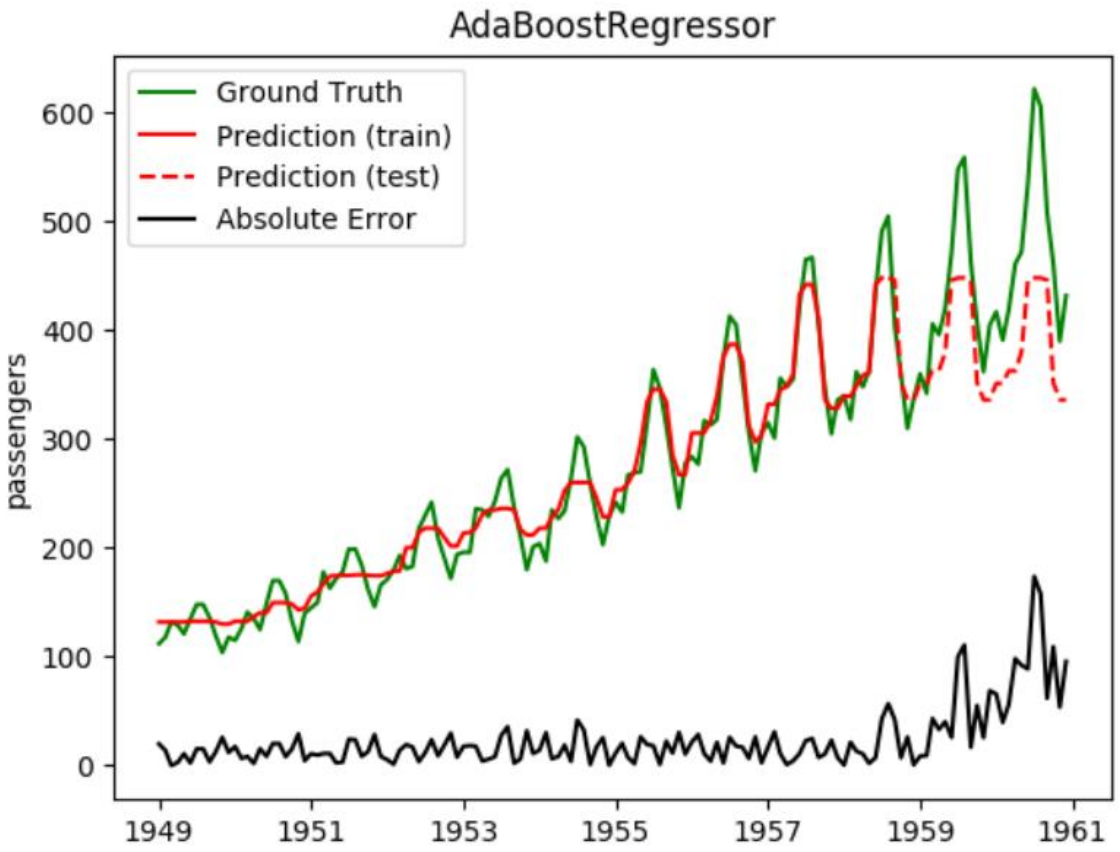
```
0.76 accuracy with a standard deviation of 0.15
```

```
print(scores.mean())
print(scores.std())
```

```
0.7594765394992852
0.15279436679229255
```

AdaBoostRegressor:

Adaboost stands for Adaptive Boosting and it is widely used ensemble learning algorithm in machine learning. Weak learners are boosted by improving their weights and make them vote in creating a combined final model. In this post, we'll learn how to use AdaBoostRegressor class for the regression problem. AdaboostRegressor starts fitting the regressor with the dataset and adjusts the weights according to error rate.




```
from sklearn.ensemble import AdaBoostRegressor
```

```
param_grid={'n_estimators':[50,100,500,1000,1500,2000],'learning_rate':[.0001,.001,0.01,.1,1.0],'loss': ['linear', 'square', 'exp
```

```
ada=AdaBoostRegressor()
gds=GridSearchCV(ada,param_grid)
gds.fit(x_train,y_train)
```

```
GridSearchCV(estimator=AdaBoostRegressor(),
             param_grid={'learning_rate': [0.0001, 0.001, 0.01, 0.1, 1.0],
                         'loss': ['linear', 'square', 'exponential'],
                         'n_estimators': [50, 100, 500, 1000, 1500, 2000]})
```

```
#sorted(gds,cv_results_.keys())
print(gds.best_params_)
```

```
{'learning_rate': 0.0001, 'loss': 'linear', 'n_estimators': 1000}
```

```
param_grid={
    'random_state': list(range(1,100))}
```

```
ada=AdaBoostRegressor()
gds=GridSearchCV(ada,param_grid)
gds.fit(x_train,y_train)
```

```
GridSearchCV(estimator=AdaBoostRegressor(),
             param_grid={'random_state': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
                                          13, 14, 15, 16, 17, 18, 19, 20, 21,
                                          22, 23, 24, 25, 26, 27, 28, 29, 30, ...]})
```

```
#sorted(gds,cv_results_.keys())
print(gds.best_params_)
```

```
{'random_state': 61}
```

```
ada=AdaBoostRegressor(loss='square',n_estimators=1000,random_state=61,learning_rate=1.0)
```

```
ada.fit(x_train,y_train)
predada=ada.predict(x_test)
print(predada)
```

```
[12.16768589 12.20344073 12.28185969 11.88018531 12.33973308 12.17441079
 12.28352903 11.68610769 11.89811785 12.34583459 12.15714076 12.28352903
 12.17976818 11.74782789 12.34028488 12.19870896 12.15477935 11.90415617]
```

```
print('ada score:',ada.score(x_train,y_train))
print('ada r2_score:',r2_score(y_test,predada))
```

```
ada score: 0.9697870118697627
ada r2_score: 0.7617986844430252
```

```
print("Mean absolue error of ada:",mean_absolute_error(y_test,predada))
print("Mean squared error of ada:",mean_squared_error(y_test,predada))
print("Root Mean squared error of ada:",np.sqrt(mean_squared_error(y_test,predada)))
```

```
Mean absolue error of ada: 0.07812739562890109
Mean squared error of ada: 0.011619866083354457
Root Mean squared error of ada: 0.10779548266673541
```

```
scores = cross_val_score(ada, x_train, y_train, cv=5)
scores
```

```
array([0.58975038, 0.64272275, 0.89014343, 0.91238353, 0.32369411])
```

```
print("%0.2f accuracy with a standard deviation of %0.2f" % (scores.mean(), scores.std()))
```

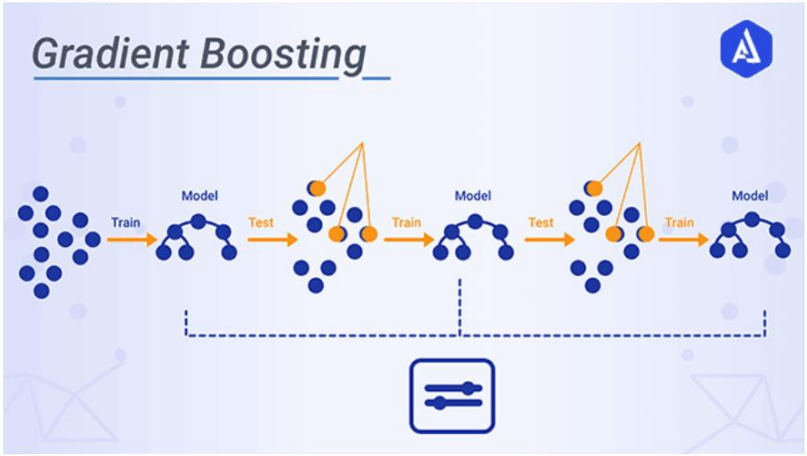
```
0.67 accuracy with a standard deviation of 0.22
```

```
print(scores.mean())
print(scores.std())
```

```
0.6717388404790269
0.2164761445846564
```

GradientBoostRegressor:

Gradient Boosting algorithm is used to generate an ensemble model by combining the weak learners or weak predictive models. Gradient boosting algorithm can be used to train models for both regression and classification problem. Gradient Boosting Regression algorithm is used to fit the model which predicts the continuous value.




```
from sklearn.ensemble import GradientBoostingRegressor
```

```
param_grid={
 'max_depth': [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, None],
 'max_features': ['auto', 'sqrt','log2'],
 'min_samples_leaf': [1, 2, 4]}
```

```
grid=GradientBoostingRegressor()
gds=GridSearchCV(grid,param_grid)
gds.fit(x_train,y_train)
```

```
GridSearchCV(estimator=GradientBoostingRegressor(),
             param_grid={'max_depth': [10, 20, 30, 40, 50, 60, 70, 80, 90, 100,
                                       None],
                         'max_features': ['auto', 'sqrt', 'log2'],
                         'min_samples_leaf': [1, 2, 4]})
```

```
#sorted(gds,cv_results_.keys())
print(gds.best_params_)
```

```
{'max_depth': 70, 'max_features': 'sqrt', 'min_samples_leaf': 1}
```

```
param_grid={
 'min_samples_split': [2, 5, 10],
  'n_estimators': [200, 400, 600, 800, 1000, 1200, 1400, 1600, 1800, 2000],
  'subsample':[0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9],
   'init':['estimator','zero',None]
}
```

```
grid=GradientBoostingRegressor()
gds=GridSearchCV(grid,param_grid)
gds.fit(x_train,y_train)
```

```
GridSearchCV(estimator=GradientBoostingRegressor(),
             param_grid={'init': ['estimator', 'zero', None],
                         'min_samples_split': [2, 5, 10],
                         'n_estimators': [200, 400, 600, 800, 1000, 1200, 1400,
                                          1600, 1800, 2000],
                         'subsample': [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8,
                                       0.9]})
```

```
#sorted(gds,cv_results_.keys())
print(gds.best_params_)
```

```
{'init': None, 'min_samples_split': 5, 'n_estimators': 400, 'subsample': 0.3}
```

```
param_grid={
    'max_leaf_nodes':[10,20,30,40,50,60,70,80,90,100,None],
    'criterion' :['mse','mae','friedman_mse', 'squared_error'],
    'alpha':[0.1,0.001,0.0001,.1,1.0]}
```

```
grid=GradientBoostingRegressor()
gds=GridSearchCV(grid,param_grid)
gds.fit(x_train,y_train)
```

```
GridSearchCV(estimator=GradientBoostingRegressor(),
             param_grid={'alpha': [0.1, 0.001, 0.0001, 0.1, 1.0],
                         'criterion': ['mse', 'mae', 'friedman_mse',
                                       'squared_error'],
                         'max_leaf_nodes': [10, 20, 30, 40, 50, 60, 70, 80, 90,
                                            100, None]})
```

```
#sorted(gds,cv_results_.keys())
print(gds.best_params_)
```

```
{'alpha': 0.0001, 'criterion': 'friedman_mse', 'max_leaf_nodes': 20}
```

```
param_grid={
    'random_state': list(range(1,100)),
    'warm_start':[True,False]}
```

```
grid=GradientBoostingRegressor()
gds=GridSearchCV(grid,param_grid)
gds.fit(x_train,y_train)
```

```
GridSearchCV(estimator=GradientBoostingRegressor(),
             param_grid={'random_state': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
                                          13, 14, 15, 16, 17, 18, 19, 20, 21,
                                          22, 23, 24, 25, 26, 27, 28, 29, 30, ...],
                         'warm_start': [True, False]})
```

```
#sorted(gds,cv_results_.keys())
print(gds.best_params_)
```

```
{'random_state': 70, 'warm_start': True}
```

```
param_grid={'ccp_alpha':[0.1,0.001,0.0001,.1,1.0],
    'tol':[0.1,0.14,0.18,0.01,0.001,0.0001,0.00001,1.0,1e-3]}
```

```
grid=GradientBoostingRegressor()
gds=GridSearchCV(grid,param_grid)
gds.fit(x_train,y_train)
```

```
GridSearchCV(estimator=GradientBoostingRegressor(),
             param_grid={'ccp_alpha': [0.1, 0.001, 0.0001, 0.1, 1.0],
                         'tol': [0.1, 0.14, 0.18, 0.01, 0.001, 0.0001, 1e-05,
                                 1.0, 0.001]})
```

```
#sorted(gds,cv_results_.keys())
print(gds.best_params_)
```

```
{'ccp_alpha': 0.0001, 'tol': 1e-05}
```

```
param_grid={
    'min_weight_fraction_leaf':[0.0,0.1,0.001,0.0001],
    'n_iter_no_change':[10,50,100,200,500,1000,None]}
```

```
grid=GradientBoostingRegressor()
gds=GridSearchCV(grid,param_grid)
gds.fit(x_train,y_train)
```

```
GridSearchCV(estimator=GradientBoostingRegressor(),
             param_grid={'min_weight_fraction_leaf': [0.0, 0.1, 0.001, 0.0001],
                         'n_iter_no_change': [10, 50, 100, 200, 500, 1000,
                                              None]})
```

```
#sorted(gds,cv_results_.keys())
print(gds.best_params_)
```

```
{'min_weight_fraction_leaf': 0.1, 'n_iter_no_change': 500}
```

```
param_grid={'loss':['ls', 'lad', 'huber', 'quantile','squared_error','absolute_error'],
    'min_impurity_decrease':[0.0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,0.10]}
```

```
grid=GradientBoostingRegressor()
gds=GridSearchCV(grid,param_grid)
gds.fit(x_train,y_train)
```

```
GridSearchCV(estimator=GradientBoostingRegressor(),
             param_grid={'loss': ['ls', 'lad', 'huber', 'quantile',
                                  'squared_error', 'absolute_error'],
                         'min_impurity_decrease': [0.0, 0.1, 0.2, 0.3, 0.4, 0.5,
                                                   0.6, 0.7, 0.8, 0.9, 0.1]})
```

```
#sorted(gds,cv_results_.keys())
print(gds.best_params_)
```

```
{'loss': 'lad', 'min_impurity_decrease': 0.1}
```

```
grid=GradientBoostingRegressor(criterion='friedman_mse',loss='lad',max_features='sqrt',random_state=70,max_depth=70,min_samples_l
grid.fit(x_train,y_train)
predgrid=grid.predict(x_test)
print(predgrid)
```

```
[12.23883672 12.16387818 12.21852607 11.93964912 12.3464265  12.19174964
 12.2618335  11.67328532 11.96909159 12.39735607 12.15634676 12.30399671
 12.19257088 11.84940873 12.32912848 12.33419032 12.08983382 11.9467972 ]
```

```
print('Gradient score:',grid.score(x_train,y_train))
print('Gradient r2_score:',r2_score(y_test,predgrid))
```

```
Gradient score: 0.9584857403465429
Gradient r2_score: 0.7793324418060696
```

```
print("Mean absolue error of grid:",mean_absolute_error(y_test,predgrid))
print("Mean squared error of grid:",mean_squared_error(y_test,predgrid))
print("Root Mean squared error of grid:",np.sqrt(mean_squared_error(y_test,predgrid)))
```

```
Mean absolue error of grid: 0.0748697580998478
Mean squared error of grid: 0.01076453952052582
Root Mean squared error of grid: 0.10375229886863144
```

```
scores = cross_val_score(grid, x_train, y_train, cv=5)
scores
```

```
array([0.64346653, 0.69604932, 0.6338264 , 0.71386528, 0.7095141 ])
```

```
print("%0.2f accuracy with a standard deviation of %0.2f" % (scores.mean(), scores.std()))
```

```
0.68 accuracy with a standard deviation of 0.03
```

```
print(scores.mean())
print(scores.std())
```

```
0.6793443259336268
0.03388234221873383
```

```
from xgboost import XGBRegressor
```

```
from xgboost.sklearn import XGBRegressor
```

```
param_grid={  'n_estimators': [200, 400, 600, 800, 1000, 1200, 1400, 1600, 1800, 2000]}
```

```
xg = XGBRegressor()
gds=GridSearchCV(xg,param_grid)
gds.fit(x_train,y_train)
```

```
GridSearchCV(estimator=XGBRegressor(base_score=None, booster=None,
                                    colsample_bylevel=None,
                                    colsample_bynode=None,
                                    colsample_bytree=None,
                                    enable_categorical=False, gamma=None,
                                    gpu_id=None, importance_type=None,
                                    interaction_constraints=None,
                                    learning_rate=None, max_delta_step=None,
                                    max_depth=None, min_child_weight=None,
                                    missing=nan, monotone_constraints=None,
                                    n_estimators=100, n_jobs=None,
                                    num_parallel_tree=None, predictor=None,
                                    random_state=None, reg_alpha=None,
                                    reg_lambda=None, scale_pos_weight=None,
                                    subsample=None, tree_method=None,
                                    validate_parameters=None, verbosity=None),
             param_grid={'n_estimators': [200, 400, 600, 800, 1000, 1200, 1400,
                                          1600, 1800, 2000]})
```

```
#sorted(gds,cv_results_.keys())
print(gds.best_params_)
```

```
{'n_estimators': 200}
```

```
param_grid={'max_depth': [3,5,7,9,11,13,15,17,18,19,20,30,50,60,70,80,90,100 ,None],
      'subsample':[0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9]}
```

```
xg = XGBRegressor()
gds=GridSearchCV(xg,param_grid)
gds.fit(x_train,y_train)
```

```
GridSearchCV(estimator=XGBRegressor(base_score=None, booster=None,
                                    colsample_bylevel=None,
                                    colsample_bynode=None,
                                    colsample_bytree=None,
                                    enable_categorical=False, gamma=None,
                                    gpu_id=None, importance_type=None,
                                    interaction_constraints=None,
                                    learning_rate=None, max_delta_step=None,
                                    max_depth=None, min_child_weight=None,
                                    missing=nan, monotone_constraints=None,
                                    n_estimators=100, n_jobs=None,
                                    num_parallel_tree=None, predictor=None,
                                    random_state=None, reg_alpha=None,
                                    reg_lambda=None, scale_pos_weight=None,
                                    subsample=None, tree_method=None,
                                    validate_parameters=None, verbosity=None),
             param_grid={'max_depth': [3, 5, 7, 9, 11, 13, 15, 17, 18, 19, 20,
                                       30, 50, 60, 70, 80, 90, 100, None],
                         'subsample': [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8,
                                       0.9]})
```

```
#sorted(gds,cv_results_.keys())
print(gds.best_params_)
```

```
{'max_depth': 9, 'subsample': 0.5}
```

```
param_grid={'min_child_weight':[1,2,3,4,5,6,7,8,9,10],
    'gamma':[0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,0.10]}
```

```
param_grid={ 'colsample_bytree':[0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,0.10,0.20,0.30,0.40,0.50,0.60,0.70,0.75,0.80,0.95,0.100],
     'reg_alpha':[1e-5, 1e-2, 0.1, 1, 100],
      'eta': [.3, .2, .1, .05, .01, .005]}
```

```
xg = XGBRegressor()
gds=GridSearchCV(xg,param_grid)
gds.fit(x_train,y_train)
```

```
GridSearchCV(estimator=XGBRegressor(base_score=None, booster=None,
                                    colsample_bylevel=None,
                                    colsample_bynode=None,
                                    colsample_bytree=None,
                                    enable_categorical=False, gamma=None,
                                    gpu_id=None, importance_type=None,
                                    interaction_constraints=None,
                                    learning_rate=None, max_delta_step=None,
                                    max_depth=None, min_child_weight=None,
                                    missing=nan, monotone_constraints=None,
                                    n_esti...
                                    num_parallel_tree=None, predictor=None,
                                    random_state=None, reg_alpha=None,
                                    reg_lambda=None, scale_pos_weight=None,
                                    subsample=None, tree_method=None,
                                    validate_parameters=None, verbosity=None),
             param_grid={'colsample_bytree': [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7,
                                              0.8, 0.9, 0.1, 0.2, 0.3, 0.4, 0.5,
                                              0.6, 0.7, 0.75, 0.8, 0.95, 0.1],
                         'eta': [0.3, 0.2, 0.1, 0.05, 0.01, 0.005],
                         'reg_alpha': [1e-05, 0.01, 0.1, 1, 100]})
```

```
#sorted(gds,cv_results_.keys())
print(gds.best_params_)
```

```
#sorted(gds,cv_results_.keys())
print(gds.best_params_)
```

```
{'colsample_bytree': 0.6, 'eta': 0.1, 'reg_alpha': 1e-05}
```

```
param_grid={'learning_rate':[.0001,.001,0.01,.1,1.0],
        'random_state': list(range(1,100))}
```

```
xg = XGBRegressor()
gds=GridSearchCV(xg,param_grid)
gds.fit(x_train,y_train)
```

```
GridSearchCV(estimator=XGBRegressor(base_score=None, booster=None,
                                    colsample_bylevel=None,
                                    colsample_bynode=None,
                                    colsample_bytree=None,
                                    enable_categorical=False, gamma=None,
                                    gpu_id=None, importance_type=None,
                                    interaction_constraints=None,
                                    learning_rate=None, max_delta_step=None,
                                    max_depth=None, min_child_weight=None,
                                    missing=nan, monotone_constraints=None,
                                    n_esti...obs=None,
                                    num_parallel_tree=None, predictor=None,
                                    random_state=None, reg_alpha=None,
                                    reg_lambda=None, scale_pos_weight=None,
                                    subsample=None, tree_method=None,
                                    validate_parameters=None, verbosity=None),
             param_grid={'learning_rate': [0.0001, 0.001, 0.01, 0.1, 1.0],
                         'random_state': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
                                          13, 14, 15, 16, 17, 18, 19, 20, 21,
                                          22, 23, 24, 25, 26, 27, 28, 29, 30, ...]})
```

```
#sorted(gds,cv_results_.keys())
print(gds.best_params_)
```

```
{'learning_rate': 0.1, 'random_state': 1}
```

```
xg = XGBRegressor(n_estimators= 200,max_depth= 9, subsample= 0.5,gamma= 0.1, min_child_weight= 10,colsample_bytree= 0.6, eta= 0.1
xg.fit(x_train,y_train)
predxg=xg.predict(x_test)
print(predxg)
```

```
[12.181967 12.200254 12.263726 11.925402 12.301541 12.21789  12.260482
 11.801532 11.848296 12.39394  12.146341 12.237664 12.181004 11.717038
 12.301582 12.167384 12.102448 11.9929  ]
```

```
print('xgboost score:',xg.score(x_train,y_train))
print('xgboost r2_score:',r2_score(y_test,predxg))
```

```
xgboost score: 0.8378618362823388
xgboost r2_score: 0.6929549784896121
```

```
print("Mean absolue error of xg:",mean_absolute_error(y_test,predxg))
print("Mean squared error of xg:",mean_squared_error(y_test,predxg))
print("Root Mean squared error of xg:",np.sqrt(mean_squared_error(y_test,predxg)))
```

```
Mean absolue error of xg: 0.09261286849858921
Mean squared error of xg: 0.0149781793739003
Root Mean squared error of xg: 0.122385372385348
```

```
scores = cross_val_score(xg, x_train, y_train, cv=5)
scores
```

```
array([0.59339817, 0.52802823, 0.85385441, 0.81789704, 0.47635292])
```

```
print("%0.2f accuracy with a standard deviation of %0.2f" % (scores.mean(), scores.std()))
```

```
0.65 accuracy with a standard deviation of 0.15
```

```
print(scores.mean())
print(scores.std())
```

```
0.6539061545470846
0.1535603709344387
```

Conclusion:

```
import numpy as np
a=np.array(y_test)
predicted=np.array(enr.predict(x_test))
df_com=pd.DataFrame({"original":a,"predicted":predicted},index=range(len(a)))
df_com
```

	original	predicted
0	12.165251	12.190971
1	12.273731	12.179875
2	12.208570	12.188725
3	11.998433	11.923819
4	12.247694	12.270135
5	12.119970	12.225900
6	12.332705	12.225104
7	11.699405	11.705694
8	11.970350	11.877244
9	12.449019	12.414550
10	12.061047	12.131887
11	12.631014	12.604979
12	12.139399	12.140956
13	11.845820	11.648651
14	12.449019	12.444062
15	12.220469	12.256113
16	12.146853	12.176115
17	11.941456	11.950933

Model_performance

```
model_name=['LinearRegression','Lasso','Ridge','ElasticNet','DecisionTreeRegressor','KNeighborsRegressor','SupportVectorRegressor
accuracy_model=[0.83,0.74,0.81,0.86,0.66,0.61,0.74,0.76,0.67,0.68,0.65]
Model_performance=pd.DataFrame({})
Model_performance['model_name']=model_name
Model_performance['accuracy_model']=accuracy_model
Model_performance
```

	model_name	accuracy_model
0	LinearRegression	0.83
1	Lasso	0.74
2	Ridge	0.81
3	ElasticNet	0.86
4	DecisionTreeRegressor	0.66
5	KNeighborsRegressor	0.61
6	SupportVectorRegressor	0.74
7	RandomForestRegressor	0.76
8	AdaBoostRegressor	0.67
9	GradientBoostRegressor	0.68
10	XGBRegressor	0.65

Analysis and Comparison

In the previous section, we created many models: for each model, we searched for good parameters then we constructed the model using those parameters, then trained (fitted) the model to our training data (`X_train` and `y_train`), then tested the model on our test data (`X_test`) and finally, we evaluated the model performance by comparing the model predictions with the true values in `y_test`. We used the Accuracy of the model to evaluate model performance.

Model Saving and predictions of a test data

```
import pickle
```

```
# Save to file in the current working directory
pkl_filename = "house_sales_price.pkl"
with open(pkl_filename, 'wb') as file:
    pickle.dump(enr, file)

# Load from file
with open(pkl_filename, 'rb') as file:
    pickle_model = pickle.load(file)

# Calculate the accuracy score and predict test data
score = pickle_model.score(x_test, y_test)
print("Test score: {0:.2f} %".format(100 * score))
Ypredict = pickle_model.predict(test_x) #here predict the test data
```

```
Test score: 89.13 %
```

```
#prediction of a test data of a house price predictions
Ypredict
```

```
array([11.65553742, 11.6569354 , 11.96315532, 12.12611124, 12.25517636,
       12.38890654, 12.05280088, 12.31685059, 12.21154884, 11.70351271,
       12.25849656, 11.81033018, 11.91039406, 12.20238013, 12.10438307,
       11.74490547, 12.3039182 , 12.1794525 , 11.82145963, 12.15807406,
       12.16502736, 12.39596801, 12.26340403, 11.78636396])
```

```
#To save the files
df=pd.DataFrame(Ypredict)
df.to_csv("house_price_pred_submission.csv")
```

Conclusion

In this project, we built serveral regression models to predict the price of some house given some of the house features. We eveluated and compared each model to determine the one with highest performance. We also looked at how some models rank the features according to their importance. In this project, we followed the data science process starting with getting the data, then cleaning and preprocessing the data, followed by exploring the data and building models, then evaluating the results and communicating them with visualizations.

As a recommendation, we advise to use this model (or a version of it trained with more recent data) by people who want to buy a house in the area covered by the dataset to have an idea about the actual price. The model can be used also with datasets that cover different cities and areas provided that they contain the same features. We also suggest that people take into consideration the features that were deemed as most important as seen in the previous section; this might help them estimate the house price better.

References

Alkhatib, K., Najadat, H., Hmeidi, I., & Shatnawi, M. K. A. (2013). Stock price prediction using k-nearest neighbor (kNN) algorithm. International Journal of Business, Humanities and Technology, 3(3), 32-44.

de Abril, I. M., & Sugiyama, M. (2013). Winning the kaggle algorithmic trading challenge with the composition of many models and feature engineering. IEICE transactions on information and systems, 96(3), 742-745.

Feng, Y., & Jones, K. (2015, July). Comparing multilevel modelling and artificial neural networks in house price prediction. In Spatial Data Mining and Geographical Knowledge Services (ICSDM), 2015 2nd IEEE International Conference on (pp. 108-114). IEEE.

Hegazy, O., Soliman, O. S., & Salam, M. A. (2014). A machine learning model for stock market prediction. arXiv preprint arXiv:1402.7351.

Ticknor, J. L. (2013). A Bayesian regularized artificial neural network for stock market forecasting. Expert Systems with Applications, 40(14), 5501-550

De Cock, D. (2011). Ames, Iowa: Alternative to the Boston housing data as an end of semester regression project. Journal of Statistics Education, 19(3).