TITLE OF THE PROJECT

House Prices: Advanced Regression Techniques

Table of Content

List of tables :

List of figures

1. Introduction and Background Information

We choose to work on the **Colab** notebooks environment.

The project provides us with two inputs as two datasets. One is called *'train_data'* which will be used to train model; The other is called *'test_data'* which will be used as the dataset to test our model and to make the prediction. In this project First we applied Simple linear regression with multiple variables , then we try to improve the result and we did more analysis to the dataset And applied regression teqnieqes see the effect on the result or not.

we will try to predict the house prices for 1459 houses in the test dataset. We vertical stacking of GBoost, LightGBM, and Lasso.

Data Exploration

Data Exploration is the key to getting insights from data. Practitioners say a good data exploration strategy can solve even complicated problems in a few hours. A good data exploration strategy comprises the following:

1. Univariate Analysis - It is used to visualize one variable in one plot. Examples: histogram, density plot, etc.
2. Bivariate Analysis - It is used to visualize two variables (x and y axis) in one plot. Examples: bar chart, line chart, area chart, etc.
3. Multivariate Analysis - As the name suggests, it is used to visualize more than two variables at once. Examples: stacked bar chart, dodged bar chart, etc.
4. Cross Tables -They are used to compare the behavior of two categorical variables.

2. Problem Statement

Predict the sale price of houses in Ames, Iowa, given 79 explanatory variables describing (almost) every aspect of residential homes in Ames, Iowa, to predict the final price of each home.

3. Used Dataset.

The data set used in this project describes the sale of individual residential property in Ames, Iowa from 2006 to 2010. The data set contains 2930 observations and a large number of explanatory variables (23 nominal, 23 ordinal, 14 discrete, and 20 continuous) involved in assessing home values.

```
train.info()
```

```
Foundation       1460 non-null object
BsmtQual         1423 non-null object
BsmtCond         1423 non-null object
BsmtExposure     1422 non-null object
BsmtFinType1     1423 non-null object
BsmtFinSF1       1460 non-null int64
BsmtFinType2     1422 non-null object
BsmtFinSF2       1460 non-null int64
BsmtUnfSF        1460 non-null int64
TotalBsmtSF      1460 non-null int64
Heating          1460 non-null object
```

```
EnclosedPorch    1460 non-null int64
3SsnPorch        1460 non-null int64
ScreenPorch      1460 non-null int64
PoolArea         1460 non-null int64
PoolQC           7 non-null object
Fence            281 non-null object
MiscFeature      54 non-null object
MiscVal          1460 non-null int64
MoSold           1460 non-null int64
YrSold           1460 non-null int64
SaleType         1460 non-null object
SaleCondition    1460 non-null object
SalePrice        1460 non-null int64
dtypes: float64(3), int64(35), object(43)
memory usage: 924.0+ KB
```

5.

We import both the "train" and "test" dataset.

*As we can see, the Training dataset has 1460 observations and 81 columns, while the Test dataset has 1459 observations and 80 columns (since it's missing the target variable).

We use the *'read.csv'* syntax in R to read in these two datasets.

After a quick scan of these two datasets, we choose to combine these two datasets together, even if they will be used in different ways, just because we want to clean the whole data firstly. We delete the column *'Id'* both in train and test and delete the column *'SalePrice'* in train which is our target variable and does not appear in test dataset. Now there are 78 variables remained in the combined dataset called *'df.combined'*.

4. Proposed Solution

The project provides us with two inputs as two datasets. One is called *'train_data'* which will be used to train model; The other is called *'test_data'* which will be used as the dataset to test our model and to make the prediction. We use the *'read.csv'* syntax in to read in these two datasets.

After a quick scan of these two datasets, we choose to combine these two datasets together, even if they will be used in different ways, just because we want to clean the whole data firstly. We delete the column *'Id'* both in train and test and delete the column *'SalePrice'* in train which is our target variable and does not appear in test dataset.Normalizing Response Variable

The response variable is converted to its logarithmic form to normalize it. Now there are 78 variables remained in the combined dataset called *'df.combined'*.

5. Evaluation metrics

Root-Mean-Square-Error (RMSE) the log price is to reduce the impact of biased higher price,Taking logs means that errors in predicting expensive houses and cheap houses will affect the result equally.

6. Project Design

We did more than one experiment in this project First we applied the multiple linear regression then we did another experiments.

1) Import dependent libraries
2) Read and Exploratory analysis of the data
3) Data cleaning (treating missing values, Replace null values with median ,dealing with wrong data types)
4) Apply normalization
5) Feature Engineering - Find out the most correlated columns with SalesPrice and use only those columns in models.
6)Transformations of skewed features, create dummy variables for different categorical levels)
7) Split train-testing data
8) Model Stacking Regression (Linear Regression, Lasso, GBoost, LightGBM
9) Neural Networks with Keras.

Multiple linear regression :

First step is load the data and read it.

```
# read train and testing data

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

testID = test['Id']
```

```
train.head()
```

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Utilities	LotConfig	LandSlope	Nei
0	1	60	RL	65.0	8450	Pave	NaN	Reg	Lvl	AllPub	Inside	Gtl	
1	2	20	RL	80.0	9600	Pave	NaN	Reg	Lvl	AllPub	FR2	Gtl	
2	3	60	RL	68.0	11250	Pave	NaN	IR1	Lvl	AllPub	Inside	Gtl	
3	4	70	RL	60.0	9550	Pave	NaN	IR1	Lvl	AllPub	Corner	Gtl	
4	5	60	RL	84.0	14260	Pave	NaN	IR1	Lvl	AllPub	FR2	Gtl	

5 rows × 81 columns

Now Checking the NAN values, because, we want to use the variables that has not missing values. Also there is a thing which needs to be considered. In regression analyses all the variables we will use needs to be numerical.

So in the next line the variables chosen are all numerical.

```
[14] missing_val_count_by_column = train.isnull().sum()
     missing_val_count_by_column
```

```
Id                0
MSSubClass        0
MSZoning          0
LotFrontage     259
LotArea           0
               ...
MoSold            0
YrSold            0
SaleType          0
SaleCondition     0
SalePrice         0
Length: 81, dtype: int64
```

Here we determined the variables we will use .

```
##variables)
year_built = list(train['YearBuilt'])
overall_condition = list(train['OverallCond'])
liv_area = list(train['GrLivArea'])
lot_area = list(train['LotArea'])
overall_qual = list(train['OverallQual'])
# Y variable
house_price = list(train['SalePrice'])
# X variables of test data
year_built_test = list(test['YearBuilt'])
overall_condition_test = list(test['OverallCond'])
liv_area_test = list(test['GrLivArea'])
lot_area_test = list(test['LotArea'])
overall_qual_test = list(test['OverallQual'])
print("Variables are: year_built, overall_condition liv_area, lot_area, overall_qual")
```

```
Variables are: year_built, overall_condition liv_area, lot_area, overall_qual
```

In this part we concatenated the variables in order to create the array.

Model has been trained and ready for prediction , and there is the Predicted values.

```
[32] difference_list = []
     for i in range(len(y_pred)):
         difference = 0
         difference = df_sample_submission['SalePrice'][i] - y_pred[i]
         difference_list.append(abs(difference))
     difference_list.sort(reverse = True)
     average_difference = sum(difference_list)/len(difference_list)
     print("Average difference between predicted values and results: ", average_difference)
```

```
Average difference between predicted values and results:  48530.44937147013
```

Difference between predicted values and sample submission results.

Before we finish the analysis, we need to check the accuracy. We will find the coefficient of determination which will be a values between 0-1. As much as it is close to 1.00, the result can be improved .

```
model.score(array_variables,house_price)
```

```
0.6804595198465027
```

Now will start with some exploration of the data, as we start applying another machine learning methods without first gaining some understanding of the nature of the problem and its features.

```
[9] train_data.shape, test_data.shape
```

```
((1460, 80), (1459, 79))
```

As we can see, the Training dataset has 1460 observations and 81 columns, while the Test dataset has 1459 observations and 80 columns (since it's missing the target variable).

```
[7] train_id = train_data['Id']
    test_id = test_data['Id']
```

```
[8] for dataset in combine:
        dataset.drop('Id',axis=1,inplace=True)
```

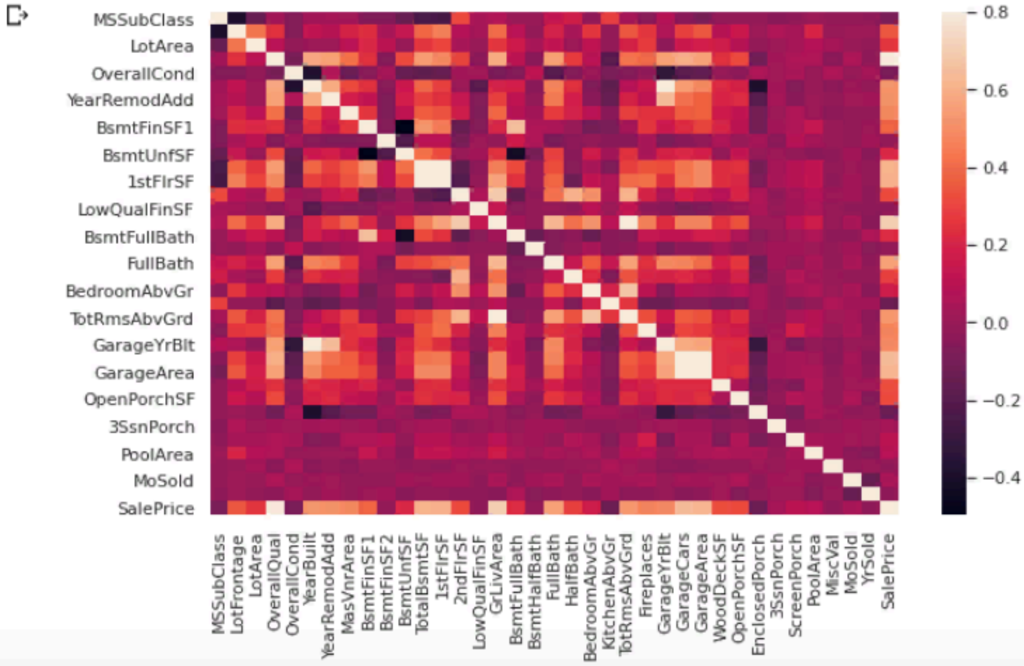
Zoomed correlation map

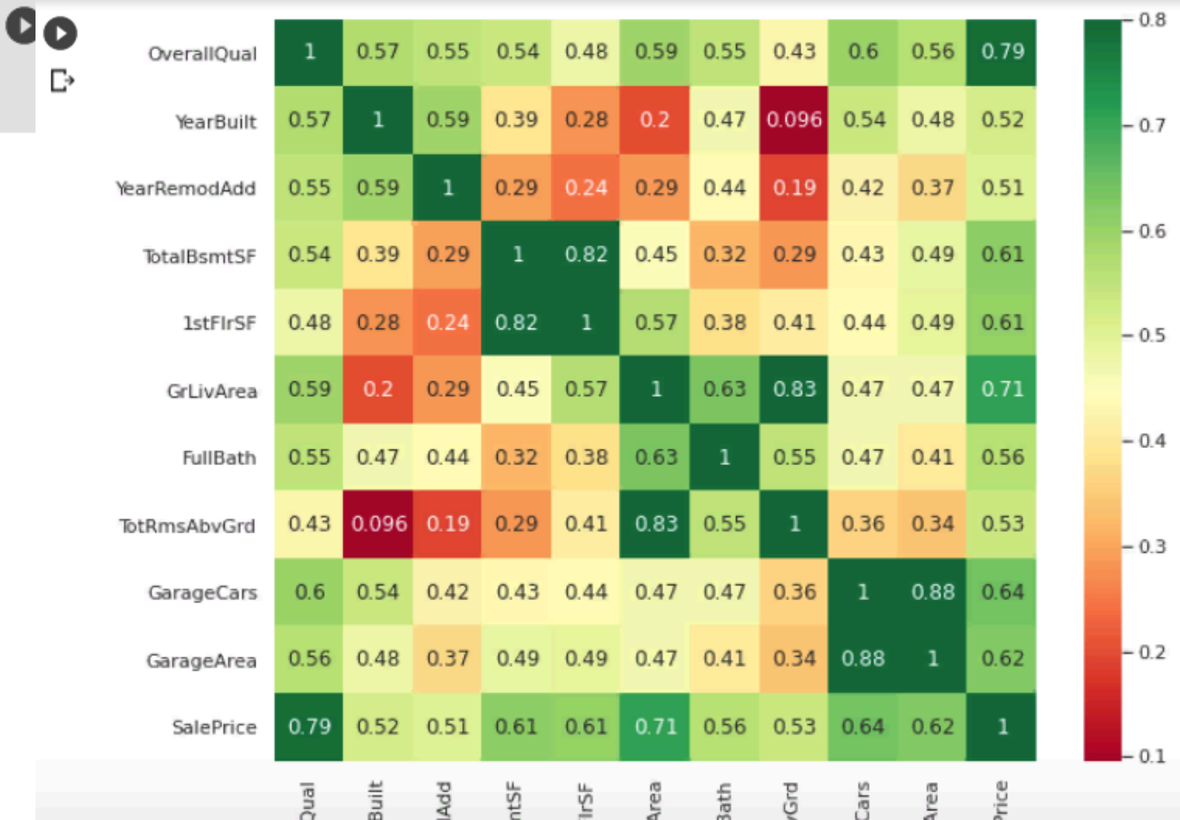
Dropping the Id column as it not useful .

Exploratory data analysis

Correlation map

```
plt.figure(figsize=(10,7))
sns.heatmap(train_data.corr(),vmax=0.8)
```





From the above, we can see that ‘OverallQual’ has the highest correlation with ‘SalePrice’. There are few other features that have high correlation with ‘SalePrice’. Another observation is that there are some variables that are highly correlated with each other, therefore, we might have multicollinearity problem. For example, ‘GarageArea’ and ‘GarageCar’. We will create a pair plot with our new list of variables that avoid having two variables that are highly correlated.

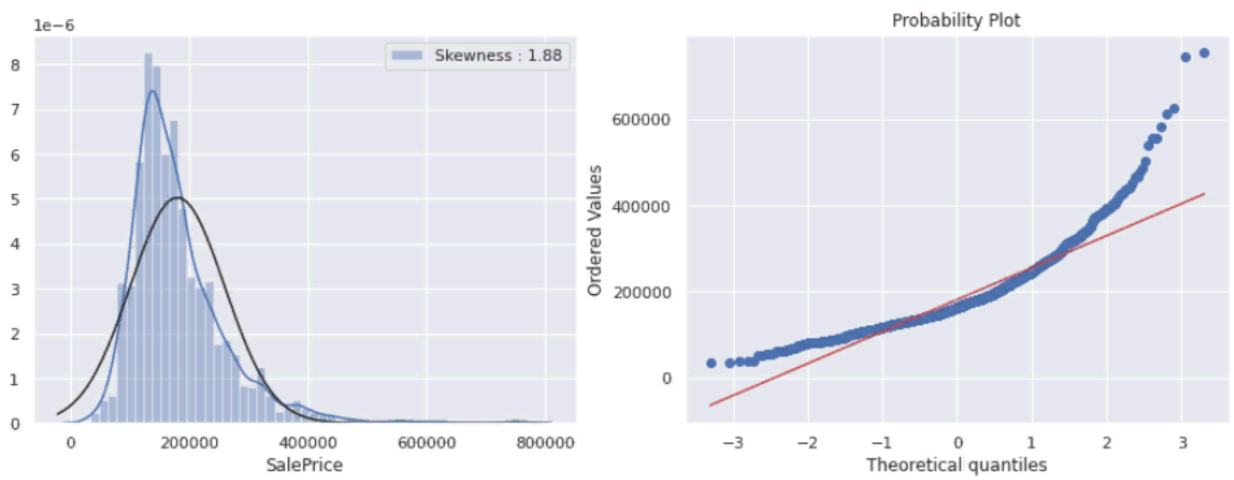
Exploring target variable: SalePrice

```
[25] plt.subplots(figsize=(15, 5))

     plt.subplot(1, 2, 1)
     g = sns.distplot(train_data['SalePrice'], fit=norm, label = "Skewness : %.2f"%(train_data['SalePrice'].s
     g = g.legend(loc="top-right")

     plt.subplot(1, 2, 2)
     res = stats.probplot(train_data['SalePrice'], plot=plt)
     plt.show()
```

```
[25] This will raise an exception in 3.3.
     """
```

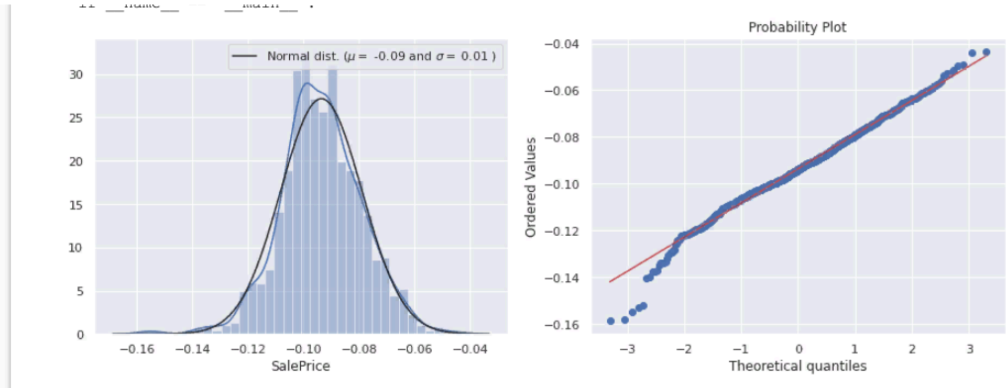


As we can see from the graphs, our response (SalePrice) has a large positive skewness is not normal. This might be a problem for linear-based model (does not follow the diagonal line). We will transform the variable ,we will appl a log transformation and see if this corrects the normality of the variable.

```
[31]
    train_data["SalePrice"] = np.log(train_data["SalePrice"])
    (mu, sigma) = norm.fit(train_data['SalePrice'])

    plt.subplots(figsize=(15, 5))
    plt.subplot(1, 2, 1)
    g = sns.distplot(train_data['SalePrice'], fit=norm)
    g.legend(['Normal dist. ($\mu=$ {:.2f} and $\sigma=$ {:.2f} )'.format(mu, sigma)],
              loc='top-right')

    plt.subplot(1, 2, 2)
    res = stats.probplot(train_data['SalePrice'], plot=plt)
    plt.show()
```



As the graph above we see the log transformation has largely improved the normality of our distribution

```
] y_train = np.log1p(train_data['SalePrice'])
```

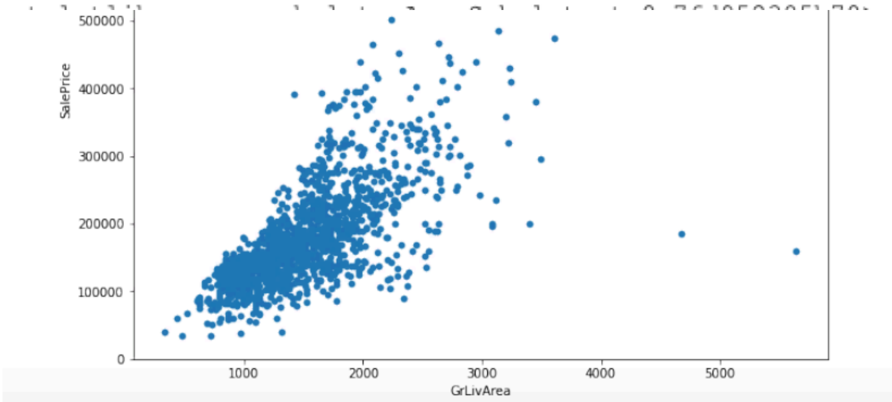
Apply log transformation.

Data Cleaning

Outlier

```
var = 'GrLivArea'

data = pd.concat([train_data[var],train_data['SalePrice']],axis=1)
data.plot.scatter(x = var, y ='SalePrice',ylim=(0,800000))
```



The two values on the bottom right seems to be outlier. It doesn’t follow the crowd. This might be due to the data points reflecting the agricultural area (explaining the low price)

Remove the two points.

```
train = train_data.drop(train_data[(train_data['GrLivArea'] > 4000) & (train_data['SalePrice'] < 300000)].index)
```

Dealing with missing values

The second part of data cleaning process, is dealing with missing values.
First, we get an idea about the variables that have missing values.
Then, with a little help from the `data_description.txt` file that containts description for each variable, we will choose how to deal with each one individually, in order to replace the missing values with something reasonable.

The most important par at this stage, is to make sure that whatever way we choose to deal with missing values in the Training dataset, we keep it consistent in the Test dataset. **For that reason, we will concatenate the 2 datasets**, perform the changes in the joint dataset and then split it again before we start the Model Formulation.

Concatenating the test + train datasets and Dropping the 'SalePrice' column, because it has values only for the train dataset

```
ntrain = train_data.shape[0]
ntest = test_data.shape[0]
y_train = train_data['SalePrice'].values
```

```
all_data = pd.concat([train_data,test_data]).reset_index(drop=True)
```

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

```
all_data.shape
```

```
(2915, 79)
```

PoolQC	1453	99.52
MiscFeature	1406	96.30
Alley	1369	93.77
Fence	1179	80.75
FireplaceQu	690	47.26
LotFrontage	259	17.74
GarageType	81	5.55
GarageCond	81	5.55
GarageFinish	81	5.55
GarageQual	81	5.55
GarageYrBlt	81	5.55
BsmtFinType2	38	2.60
BsmtExposure	38	2.60
BsmtQual	37	2.53
BsmtCond	37	2.53

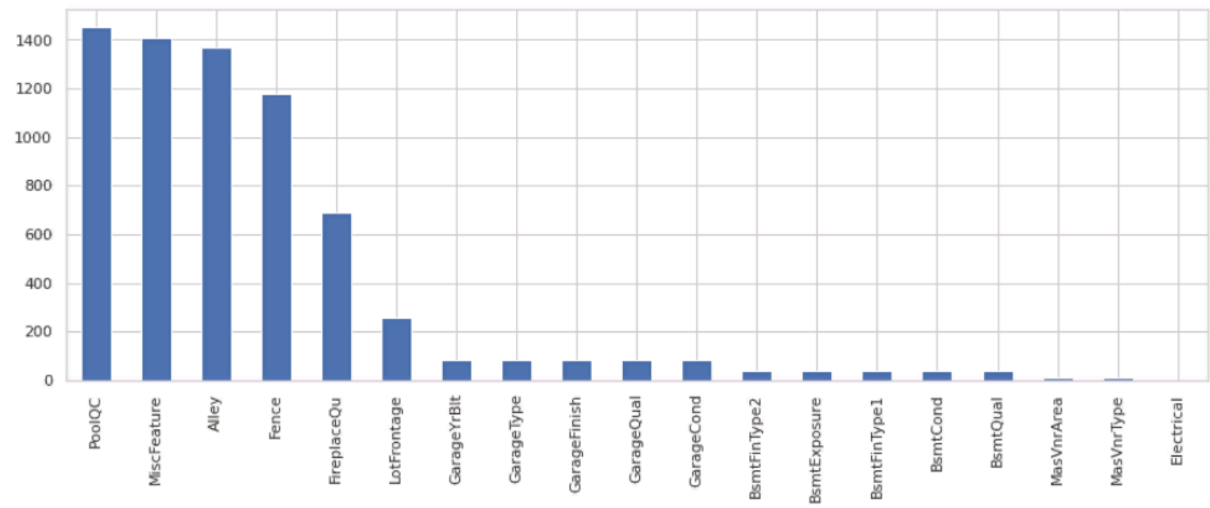
Getting an idea about

```
trair          lues(as
perce          ()/trai          lues(ascending=False)
trair          ,axis=1
trair_
```

Missing Values

First we checked which variables have missing values and how many each.

<matplotlib.axes._subplots.AxesSubplot at 0x7fde3f313438>



Plotting the count bars to get an idea of the missing values each column has
We can observe that the variables `PoolQC`, `MiscFeature`, `Alley`, `Fence` and `FireplaceQu` have a huge amount of missing values. Also, there are quite a few

columns that have only 1 or 2 missing values (the ones that have a non-visible bar). In total, there's missing values in 34 columns of our dataframe.

```
[48] col = ("PoolQC", "MiscFeature", "Alley", "Fence", "FireplaceQu", "GarageType", "GarageFinish",
        "GarageQual", "GarageCond", "BsmtQual", "BsmtCond", "BsmtExposure", "BsmtFinType1", "BsmtFinType2",
        "MasVnrType", "MSSubClass")


    for i in col:

        all_data[i] = all_data[i].fillna("None")
```

Variables to fill with “None"

There are certain variables (mainly categorical variables that concern features of the house), where the existence of a missing value or an NA, indicates that the house does not have that particular feature. We will fill these missing values with "None"

Variables to fill with 0

For numeric variables, the meaning of a null value, is that this value is equal to zero (0). That's why we will replace missing values with 0.

```
[49] col = ("GarageArea", "GarageCars", "BsmtFinSF1", "BsmtFinSF2", "BsmtUnfSF",
            "TotalBsmtSF", "MasVnrArea", "BsmtFullBath", "BsmtHalfBath")

    for i in col:
        all_data[i] = all_data[i].fillna(all_data[i].fillna(0))
```

Variables with very low missing values (strings)

For variables that have a very low number of missing values (mostly 1 value missing), we will replace them with the most common value (string) in the whole column , since it will keep the proportions of the values fairly unchanged.

```
[50] col = ("MSZoning", "Electrical", "KitchenQual", "Exterior1st", "Exterior2nd", "SaleType", "Functional")
    for i in col:
        all_data[i] = all_data[i].fillna(all_data[i].mode()[0])
```

```
[51] all_data['Utilities'].value_counts()
```

```
AllPub    2912
NoSeWa       1
Name: Utilities, dtype: int64
```

Other Variables

For the GarageYrBlt variable, a sensible assumption is that it was built the same year as the house, or that even if the Garage is younger than the house, it would have a large impact on the price of the house.

For the LotFrontage variable, a sensible assumption is that the Lot Frontage of a house is similar to the other houses in the same neighborhood.

That's why we will replace the missing values with the median value of the LotFrontage for the specific neighborhood. We don't use the Average, because it might be influenced by some extreme values. Here, the median is a better option to get realistic values.

The Utilities variable, seems to be quite unhelpful, since all the houses have full utilities (AllPub), instead of 2 houses in the test dataset that have NoSeWa (=Electricity and Gas Only) and 2 missing values. There is no way to make this feature helpful for the predictive model, so we will drop it entirely.

```
[52] # Fixing missing values for GarageYrBlt
     all_data["GarageYrBlt"] = all_data["GarageYrBlt"].fillna(all_data["YearBuilt"])

     # Fixing missing values for LotFrontage
     all_data["LotFrontage"] = all_data.groupby("Neighborhood")["LotFrontage"].transform(  lambda x: x.fillna(x.median()))

     # Dropping the Utilities variable
     all_data = all_data.drop(['Utilities'], axis=1)
```

Checking to see if there are any missing values

```
print("# of missing values = " + str(joint.isnull().sum().sum()))
```

```
# of missing values = 0
```

After our treatment of missing values has been completed, we do a final check to see if we have any missing values left.
Since, we don't have any missing values, we are good to go to the next step.

Dealing with wrong data types

Some of the variables have the wrong data type.Specifically, the variables `MSSubClass`, `YrSold`, `MoSold` and `OverallCond` are treated as numbers from the dataset, while in reality they are structured in a categorical way (with different levels). That's why we will convert them to strings.

```
[54] # Converting to categorical features
     col = ("YrSold", "MoSold", "OverallCond")
     for i in col:
          all_data[i] = all_data[i].astype(str)

     all_data['MSSubClass'] = all_data['MSSubClass'].apply(str)
```

Feature Engineering

Transforming numerical variables that are categorical

```
[55] from sklearn.preprocessing import LabelEncoder
     cols = ('FireplaceQu', 'BsmtQual', 'BsmtCond', 'GarageQual', 'GarageCond',
             'ExterQual', 'ExterCond','HeatingQC', 'PoolQC', 'KitchenQual', 'BsmtFinType1',
             'BsmtFinType2', 'Functional', 'Fence', 'BsmtExposure', 'GarageFinish', 'LandSlope',
             'LotShape', 'PavedDrive', 'Street', 'Alley', 'CentralAir', 'MSSubClass', 'OverallCond'
             'YrSold', 'MoSold')
     # process columns, apply LabelEncoder to categorical features
     for i in cols:
         lbl = LabelEncoder()
         lbl.fit(list(all_data[i].values))
         all_data[i] = lbl.transform(list(all_data[i].values))

     # shape
     print('Shape : {}'.format(all_data.shape))

     Shape : (2915, 80)
```

Since area related features are very important to determine house prices, we decided to add one more feature, which is the total area of basement, first and second floor areas of each house

```
[203] all_data['TotalSF'] = all_data['TotalBsmtSF'] + all_data['1stFlrSF'] + all_data['2ndFlrSF']
```

Treating skewed features

The dataset contains some extremely skewed features. These may largely affect the model's predictive ability. One solution would be to transform them, using the Box-Cox Transformation.

To do that, we will separate the numeric columns and use the `boxcox1p()` function on the variables that have an extremely large skewness (>0.75).

```
[205] # We will transform only the variables that have an extremely large skewness (>0.75)
      skewness = skewed[abs(skewed) > 0.75]

      skewed = skewness.index
      lam = 0.15
      for i in skewed:
          all_data[i] = boxcox1p(all_data[i], lam)

      print(skewness.shape[0],  "skewed numerical features have been Box-Cox transformed")

      37 skewed numerical features have been Box-Cox transformed
```

Code Text

And finally Creating dummy variables for different categorical lavels.

```
[58] all_data = pd.get_dummies(all_data)
     print(all_data.shape)
     all_data.head()

     (2915, 221)
```

	1stFlrSF	2ndFlrSF	3SsnPorch	Alley	BedroomAbvGr	BsmtCond	BsmtExposure	BsmtFinSF1	BsmtFinSF2	BsmtFinType1	Bsı
0	11.692623	11.686189	0.0	1	3	1.820334	1.540963	11.170327	0.0	2	
1	12.792276	0.000000	0.0	1	3	1.820334	0.730463	12.062832	0.0	0	
2	11.892039	11.724598	0.0	1	3	1.820334	1.194318	10.200343	0.0	2	
3	12.013683	11.354094	0.0	1	3	0.730463	1.540963	8.274266	0.0	0	
4	12.510588	12.271365	0.0	1	4	1.820334	0.000000	10.971129	0.0	2	

5 rows × 221 columns

Splitting the dataset into train and test

```
[209] train_data = all_data[:ntrain]
      test_data = all_data[ntrain:]
```

```
[210] train_data.shape
```

```
(1460, 221)
```

```
[211] test_data.shape
```

```
(1459, 221)
```

Now the data is ready for next step.

Model Formulation :

Cross-validation parameters
First, we define a cross-validation function to get the RMSE of each model, using 5-fold cross-validation.

```
[212] n_folds = 5

      def rmse_cv(model):
          kf = KFold(n_folds,shuffle=True, random_state=42).get_n_splits(train_data.values)
          rmse = np.sqrt(-cross_val_score(model, train_data.values, y_train, scoring="neg_mean_squared_error", cv = kf))
          return (rmse)
```

LASSO model

First is to use a penalized LASSO model, which will select the best variables to use and produce the coefficients used for the predictions

```
[215] lasso = Lasso(alpha =0.0005, random_state=1)
      lasso.fit(train_data.values, y_train)
      lasso_pred = np.exp(lasso.predict(test_data.values))
      score = rmse_cv(lasso)
      print("Lasso score: {:.4f} ({:.4f})\n".format(score.mean(), score.std()))
```

```
Lasso score: 0.1238 (0.0174)
```

To avoid being influenced by extreme values, we apply the RobustScaler() function, to make the model more robust to outliers.

```
[217] lasso1 = make_pipeline(RobustScaler(), Lasso(alpha =0.0005, random_state=1))
      lasso1.fit(train_data.values, y_train)
      lasso1_pred = np.exp(lasso1.predict(test_data.values))
      score = rmse_cv(lasso1)
      print("Lasso (Robust Scaled) score: {:.4f} ({:.4f})\n".format(score.mean(), score.std()))
```

```
Lasso (Robust Scaled) score: 0.1237 (0.0172)
```

We see that the Lasso Model, gives quite good results.

Elastic Net regression

Performing Elastic Net

```
[219] ENet = make_pipeline(RobustScaler(), ElasticNet(alpha = 0.0005, l1_ratio=.9, random_state=3))
```

```
[220] score = rmse_cv(ENet)
      print("ElasticNet score: {:.4f} ({:.4f})\n".format(score.mean(), score.std()))
```

```
ElasticNet score: 0.1237 (0.0172)
```

Kernel ridge regression

Performing KRR

```
[221] KRR = KernelRidge(alpha=0.6, kernel='polynomial',degree=2,coef0=2.5)
```

```
[222] score = rmse_cv(KRR)
      print("Kernel Ridge score: {:.4f} ({:.4f})\n".format(score.mean(), score.std()))
```

```
Kernel Ridge score: 0.1681 (0.0122)
```

We now move to Boosting methods.

GradientBoostingRegressor

```
[223] GBoost = GradientBoostingRegressor(n_estimators=3000, learning_rate=0.05,
                                       max_depth=4, max_features='sqrt',
                                       min_samples_leaf=15, min_samples_split=10,
                                       loss='huber', random_state =5)
```

```
[224] score = rmse_cv(GBoost)
      print("Gradient Boosting score: {:.4f} ({:.4f})\n".format(score.mean(), score.std()))
```

```
Gradient Boosting score: 0.1239 (0.0123)
```

XGBRegressor

```
[227] model_lgb = lgb.LGBMRegressor(objective='regression',num_leaves=5,
                              learning_rate=0.05, n_estimators=720,
                              max_bin = 55, bagging_fraction = 0.8,
                              bagging_freq = 5, feature_fraction = 0.2319,
                              feature_fraction_seed=9, bagging_seed=9,
                              min_data_in_leaf =6, min_sum_hessian_in_leaf = 11)
```

```
[228] score = rmse_cv(model_lgb)
      print("LGBM score: {:.4f} ({:.4f})\n" .format(score.mean(), score.std()))
```

```
LGBM score: 0.1246 (0.0093)
```

LGBMRegressor

```
[225] model_xgb = xgb.XGBRegressor(colsample_bytree=0.4603, gamma=0.0468,
                             learning_rate=0.05, max_depth=3,
                             min_child_weight=1.7817, n_estimators=2200,
                             reg_alpha=0.4640, reg_lambda=0.8571,
                             subsample=0.5213, silent=1,
                             nthread = -1)
```

```
[226] score = rmse_cv(model_xgb)
      print("Xgboost score: {:.4f} ({:.4f})\n".format(score.mean(), score.std()))
```

```
Xgboost score: 0.1222 (0.0093)
```

Stacking models – Averaging base models (Simplest)

Averaging base models score

Trining and predicting

And here the RMSLE score on train data

```
[231] def rmsle(y_test, y_pred):
          return np.sqrt(mean_squared_error(y_test, y_pred))
```

```
RMSLE score on train data:
0.07323736430507158
```

```
        return self

    #Now we do the predictions for cloned models and average them
    def predict(self, X):
        predictions = np.column_stack([
            model.predict(X) for model in self.models_
        ])
        return np.mean(predictions, axis=1)
```

Stacking

Another approach we applied Stacked Regression Model. To implement it:

- We will split the training dataset in a 80% training and a 20% validation set
- We will use the training set to fit the XGBRegresso LGBMRegressor models and predict the values for the newly-created validation datasetset, as well as the actual test dataset.
- After that, knowing the actual values of the target variable for the validation dataset, we will train Lasso as a meta-model, in order to create predictions for the actual test dataset, using the predictions from the other models.

First we perform a train_test_split, in order to split the Training dataset in 80%

```
# Making predictions for the actual test dataset
tpred1 = model_xgb.predict(test_data.values)
tpred2 = model_lgb.predict(test_data.values)


# Stacking the prediction outcomes in the 2 data frames
stacked_validation = np.column_stack((predictions1, predictions2))
```

```
[237] x_trn, x_val, y_trn, y_val = model_selection.train_test_split(train_data.values, y_train, test_size=0.2, random_state=42)
      print('x_trn: ', x_trn.shape, '\nx_val: ', x_val.shape, '\ny_trn: ', y_trn.shape, '\ny_val: ', y_val.shape)

x_trn:  (1168, 221)
x_val:  (292, 221)
y_trn:  (1168,)
y_val:  (292,)
```

```
final_predictions = np.exp(lasso.predict(stacked_test))
```

training set and a 20% validation set for the stacking algorithm.

- x_trn = predictor features for estimation dataset
- x_val = predictor variable for estimation dataset
- y_trn = target variable for the estimation dataset
- y_val = target variable for the validation dataset

```
[239] !pip install vecstack

Requirement already satisfied: vecstack in /usr/local/l
Requirement already satisfied: scipy in /usr/local/lib/
Requirement already satisfied: numpy in /usr/local/lib/
Requirement already satisfied: scikit-learn>=0.18 in /u
Requirement already satisfied: joblib>=0.11 in /usr/loc

[240] from vecstack import stacking

[241] models = [model_xgb, model_lgb,lasso]

[242] from sklearn.metrics import mean_absolute_error
```

```
[243] S_train[:5]
```

```
array([[12.23087502, 12.26605888, 12.23359789],
       [12.07153416, 12.04243962, 12.13119556],
       [12.27294922, 12.26570433, 12.29656078],
       [12.0340929 , 12.1165991 , 12.09099424],
       [12.60389233, 12.60957013, 12.61080667]])
```

```
[244] S_test[:5]
```

```
array([[11.73193645, 11.73128138, 11.66589168],
       [11.98745632, 11.97279951, 11.92752843],
       [12.11256123, 12.12728554, 12.11111792],
       [12.18240452, 12.19350907, 12.19119989],
       [12.15938091, 12.16720705, 12.2065225 ]])
```

```
[245] S_train, S_test = stacking(models,                    # list of models
                                 train_data.values, y_train, test_data.values,   # data
                                 regression=True,           # regression task (if you need
                                                            #     classification - set to False)
                                 mode='oof_pred',       # mode: oof for train set, predict test
                                                            #     set in each fold and find mean
                                 save_dir=None,             # do not save result and log (to save
                                                            #     in current dir - set to '.')
                                 metric=mean_absolute_error, # metric: callable
                                 n_folds=5,                 # number of folds
                                 shuffle=True,              # shuffle the data
                                 random_state=0,            # ensure reproducibility
                                 verbose=2)                 # print all info
```

Output :

```
task:          [regression]
metric:        [mean_absolute_error]
mode:          [oof_pred]
n_models:      [3]

model  0:      [XGBRegressor]
    fold  0:   [0.08363998]
    fold  1:   [0.07407641]
    fold  2:   [0.09621934]
    fold  3:   [0.07745524]
    fold  4:   [0.07567507]
    ----
    MEAN:      [0.08141321] + [0.00808208]
    FULL:      [0.08141321]

    Fitting on full train set...
```

Model predicted values :

Model score :

Models Scores:

Model	Metric	Data preproseing	Score
Multiple Linear Regression	-	Not scaled	0.680459519846503
averaged_models	mean squared error	scaled	0.0755300703829717
XGBRegressor	mean squared error	scaled	0.0790106774228878
LGBMRegressor	mean squared error	scaled	0.0753505998790645
Stacking models Simplest)	RMSLE	scaled	0.0732373643050716
Stacked Regression Model	mean absolute error (MAE)	scaled	0.90955289248104

```
model  1:     [LGBMRegressor]
    fold  0:  [0.08687292]
    fold  1:  [0.07616563]
    fold  2:  [0.09263632]
    fold  3:  [0.07854066]
    fold  4:  [0.07632750]
    ----
    MEAN:     [0.08210861] + [0.00655747]
    FULL:     [0.08210861]

    Fitting on full train set...

model  2:     [Lasso]
    fold  0:  [0.08526387]
    fold  1:  [0.07348467]
    fold  2:  [0.09393151]
    fold  3:  [0.07479957]
    fold  4:  [0.07350193]
    ----
    MEAN:     [0.08019631] + [0.00816482]
    FULL:     [0.08019631]

    Fitting on full train set...
```

Neural Networks with Keras

We did another and final experiment Trying to see if Neural Networks (with Keras) can produce a better result.

```
[246] from sklearn.linear_model import LinearRegression
      reg = LinearRegression()
      models = reg.fit(S_train,y_train)
      y_pred = models.predict(S_test)
      print("Predicted values: ", y_pred)

Predicted values:  [11.71256952 11.96592079 12.11781444 ... 12.02009535 11.68417008
 12.33133268]
```

- We create a sequential model with 3 layers.
- We use the adam optimizer with mse as loss measure and as a metric, too.

```
[247] model
      0.909
```

```
[48] !pip install tensorflow==1.12.0
     import tensorflow as tf
     print(tf.__version__)

Requirement already satisfied: ten
Requirement already satisfied: ker
Requirement already satisfied: whe
```

```
[51] import numpy as np
     import pandas as pd
     import keras
```

```
[52] train = pd.read_csv('train.csv')
     test = pd.read_csv('test.csv')
```

Select the most related feature to train model with and Preprocess data.

```
[60]
    features = ['OverallQual', 'GrLivArea', 'GarageCars', 'TotalBsmtSF', 'FullBath', 'YearBuilt']
    train_targets = train.SalePrice
    train_data = train[features]
    test_data = test[features]
```

```
[61] mean = train_data.mean(axis=0)
     std = train_data.std(axis=0)
     train_data = (train_data - mean) / std
     test_data = (test_data - mean) / std
```

Handling NaN values

```
[62]
    mean2 = test_data['GarageCars'].mean()
    mean3 = test_data['TotalBsmtSF'].mean()

    test_data = np.array(test_data)
    test_data[1116][2] = mean2
    test_data[660][3] = mean3
```

```
[63]
    from keras import models
    from keras import layers
    from keras.layers.normalization import BatchNormalization
    from keras import optimizers

    model = models.Sequential()
    model.add(layers.Dense(32, activation='relu', input_shape=(train_data.shape[1],)))
    model.add(BatchNormalization())
    model.add(layers.Dense(32, activation='relu'))
    model.add(BatchNormalization())
    model.add(layers.Dense(1))

    model.compile(optimizer='adam', loss='mse', metrics=['mae', 'acc'])
```

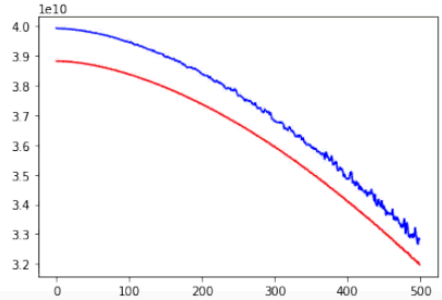
Build the model

First try was with 500 epochs

```
[64] history = model.fit(train_data, train_targets, validation_split=0.2, epochs=500,
```

```
import matplotlib.pyplot as plt
plt.plot(history.history['loss'], 'r')
plt.plot(history.history['val_loss'], 'b')
```

```
[<matplotlib.lines.Line2D at 0x7fbb942cab70>]
```

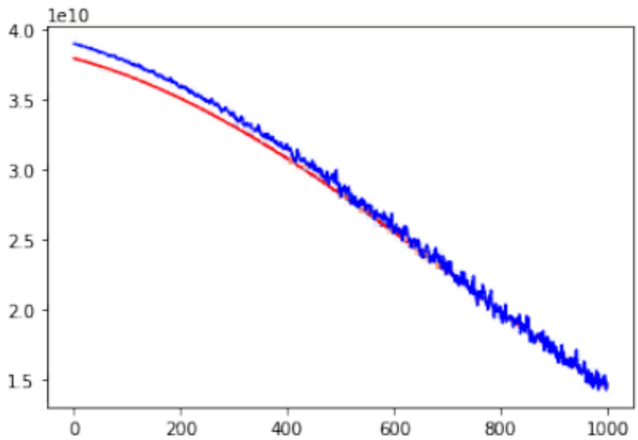


Second is with 1000 and the model become best at 2000 epochs.

```
[41] # Train model.
     history = model.fit(train_data, train_targets, validation_split=0.2, epochs=1000,
```

```
# Plot learning history.
import matplotlib.pyplot as plt
plt.plot(history.history['loss'], 'r')
plt.plot(history.history['val_loss'], 'b')
```

```
[<matplotlib.lines.Line2D at 0x7f309ae4a668>]
```



With 2000 epochs

```
[43] # Train model.
     history = model.fit(train_data, train_targets, validation_split=0.2, epochs=2000, batch_size=32, verbo
```

```
# Plot learning history.
import matplotlib.pyplot as plt
plt.plot(history.history['loss'], 'r')
plt.plot(history.history['val_loss'], 'b')
```

```
[<matplotlib.lines.Line2D at 0x7f309ad8f390>]
```