(Kaggle) House Prices: Advanced Regression Techniques

1. Introduction

House Price Prediction project was being launched two years ago, and it will be closed after two years from now. According to the current statistics, 4543 teams have been joining in this competition, counting up to 4730 competitors for now. As one of three competitions for starters, House Prices problem requires us to build a predictive model with advanced regression techniques to predict the sale price of a house in Ames, Iowa, based on many features of this house (location, size, condition, etc).

In this problem, we are given a training set and a testing data set. The training data set has 1460 rows and 81 columns, in which every row is a record of house sales. The testing data set has 1459 rows, where every row is a house with different features, and 80 columns with each row representing one single feature of a house. The main purpose of this project is to train a model based on analysis of training dataset, and apply this model to predict the sale price of houses in our testing dataset.

In the real world, some companies have developed their own models to predict the house price, such as Zillow. In our common knowledge, some influential factors in deciding the house price are obvious, such as community, area of house, amenities, construction materials, etc, while some factors might hard to locate when people might even not realize their importance. This is the purpose of data analysis: to find all correlation, association, or interactions between factors, and factors to sale price, which is hiding in the numbers. Based on findings we gathered, a proper model would be produced as a result.

This problem is of great importance in the real world as we discussed in last paragraph. Despite the reduced data records, this competition, however, is still a complicated case. Although its complexity, this problem can be breaking into several small pieces: exploring the data, cleaning data, selecting features, selecting models. For each of these puzzles, we need to use data analysis skills and programming skills to solve this problem step by step.

2 Research and methodology used in similiar problems

2.1 Business and industry

There are different approaches to predict the price of houses in real estate industry and academic world. Since the academic approaches are more focused on statistic models and methods, real world industries tend to take in more factors, such as time series, policies, economic factors, etc. In the following part, we would like to briefly introduce some approached in industries and academics.

In real estate industry, as we know, house price prediction is usually completed by property agencies. Followings are some traditional ways property agencies use in China to evaluate the house price:

(1) The cost-product algorithm, which counting the cost of acquiring land or cost of land development achieved, removing the value of abnormal factors, and taking in certain amount of capital interest and reasonable investment profits after accumulating the normal costs, then deriving an estimate value of land use rights as a result. This method is often used for the evaluation of land acquired through normal procedures.

(2) Replacement cost method is also a common method in real world. First, it measures the cost of reconstructing a house under an existing market standard for an existing house. It then takes into account the interest on the funds and takes a certain amount of development (or construction profit) to deriving a fully replacement cost price. Then according to the actual situation and legal norms to determine the new rate of housing, At the end, it will multiple the fully replacement price and the new rate of housing to obtain the value of the house.

(3) In the market, picking up real estate cases, which have already been traded or evaluated, with the same use and other similar conditions according to conditions of the real estate to be assessed. Then quantifying the indexes of each factor, through the accurate index comparison and adjustment, to find out the value of houses. This method is popular because of its practical significance and accuracy. It is usually used when the market is mature, transactions are transparent, and comparison cases are easy to find.

2.2 Academics

In academic world, there are various ways to predict the house price from different academic fields. For computing and statistic, as well as in machine learning fields, the feature selection and model selection are of great importance since features are closely related to accuracy of models and better models usually performs better when make predictions.

Linear Regression As the simplest regression model, linear regression uses a linear combination of independent variables to estimate a continuous dependent variable (C. M. Bishop, 2006). As Alex Seutin and Ian Jones did in their work, they put selected features which can represent the linear relationship best, then used vanilla linear regression predictor to take in raw data. They found the linear regression had potential to perform well given well-chosen features (Alex Seutin, Ian Jones, 2016). Furthermore, they tried random forest model from scikitlearn. They use package of sklearn.ensemble.RandomForestRegressor, which produce the best output achieved by a single algorithm(Alex Seutin, Ian Jones, 2016).

XGBoost Another commonly used model in analyzing the house price is the XGBoost model, which is an implementation of gradient boosted decision tree designed for speed and performance (Jason Brownlee, 2016). It is the abbreviation for extreme gradient boosting, referring to the engineering goal to push the limit of computations resources for boosted tree algorithms (Jason Brownlee, 2016). This model is focused on execution speed and model performance since it would make the best use of available resources to train the model and it would handle missing data value automatically. This model is becoming popular through these years and we can identify this model in many Kaggle winner's algorithms.

Ridge regression and Lasso Other models, such as ridge regression and lasso method, are also frequently be used to do a multivariate analysis. Ridge regression is a regularized and is also an extension for linear regression model. It will eliminate the irrelevant features' influence on train models, which is useful when people try to optimize their models. Lasso is also an extension for linear regression but the difference is that the regulation term is in absolute value (Ofir Chakon, 2017) and it improve the ridge regression by setting coefficient to zero when they are not relevant.

3. Descriptive Analytics

In [6]:
```
%matplotlib inline
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import xgboost as xgb
import lightgbm as lgb

from scipy import stats
from scipy.stats import norm, skew
from sklearn.linear_model import ElasticNet, Lasso
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.kernel_ridge import KernelRidge
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import RobustScaler, LabelEncoder
from sklearn.base import BaseEstimator, TransformerMixin, RegressorMixin, c
from sklearn.model_selection import KFold, cross_val_score
from sklearn.metrics import mean_squared_error
from sklearn.metrics import accuracy_score
from scipy.special import boxcox1p

import warnings
def ignore_warn(*args, **kwargs):
    pass
warnings.warn = ignore_warn
```

3.1 Basic Statistics

In [7]:
```
path = 'https://raw.githubusercontent.com/cooldoggo/kaggle1/master'
df_train = pd.read_csv(path + '/train.csv')
df_train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1460 entries, 0 to 1459
Data columns (total 81 columns):
Id               1460 non-null int64
MSSubClass       1460 non-null int64
MSZoning         1460 non-null object
LotFrontage      1201 non-null float64
LotArea          1460 non-null int64
Street           1460 non-null object
Alley            91 non-null object
LotShape         1460 non-null object
LandContour      1460 non-null object
Utilities        1460 non-null object
LotConfig        1460 non-null object
LandSlope        1460 non-null object
Neighborhood     1460 non-null object
Condition1       1460 non-null object
Condition2       1460 non-null object
BldgType         1460 non-null object
HouseStyle       1460 non-null object
OverallQual      1460 non-null int64
OverallCond      1460 non-null int64
YearBuilt        1460 non-null int64
YearRemodAdd     1460 non-null int64
RoofStyle        1460 non-null object
RoofMatl         1460 non-null object
Exterior1st      1460 non-null object
Exterior2nd      1460 non-null object
MasVnrType       1452 non-null object
MasVnrArea       1452 non-null float64
ExterQual        1460 non-null object
ExterCond        1460 non-null object
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
HeatingQC        1460 non-null object
CentralAir       1460 non-null object
Electrical       1459 non-null object
1stFlrSF         1460 non-null int64
2ndFlrSF         1460 non-null int64
LowQualFinSF     1460 non-null int64
GrLivArea        1460 non-null int64
BsmtFullBath     1460 non-null int64
BsmtHalfBath     1460 non-null int64
FullBath         1460 non-null int64
```

```
HalfBath         1460 non-null int64
BedroomAbvGr     1460 non-null int64
KitchenAbvGr     1460 non-null int64
KitchenQual      1460 non-null object
TotRmsAbvGrd     1460 non-null int64
Functional       1460 non-null object
Fireplaces       1460 non-null int64
FireplaceQu      770 non-null object
GarageType       1379 non-null object
GarageYrBlt      1379 non-null float64
GarageFinish     1379 non-null object
GarageCars       1460 non-null int64
GarageArea       1460 non-null int64
GarageQual       1379 non-null object
GarageCond       1379 non-null object
PavedDrive       1460 non-null object
WoodDeckSF       1460 non-null int64
OpenPorchSF      1460 non-null int64
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
```

The train set contains 1460 rows: each of these represents one house sold.

In [8]:
```
df_test = pd.read_csv(path + '/test.csv')
df_test.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1459 entries, 0 to 1458
Data columns (total 80 columns):
Id               1459 non-null int64
MSSubClass       1459 non-null int64
MSZoning         1455 non-null object
LotFrontage      1232 non-null float64
LotArea          1459 non-null int64
Street           1459 non-null object
Alley            107 non-null object
LotShape         1459 non-null object
LandContour      1459 non-null object
Utilities        1457 non-null object
LotConfig        1459 non-null object
LandSlope        1459 non-null object
Neighborhood     1459 non-null object
Condition1       1459 non-null object
Condition2       1459 non-null object
BldgType         1459 non-null object
HouseStyle       1459 non-null object
OverallQual      1459 non-null int64
OverallCond      1459 non-null int64
YearBuilt        1459 non-null int64
YearRemodAdd     1459 non-null int64
RoofStyle        1459 non-null object
RoofMatl         1459 non-null object
Exterior1st      1458 non-null object
Exterior2nd      1458 non-null object
MasVnrType       1443 non-null object
MasVnrArea       1444 non-null float64
ExterQual        1459 non-null object
ExterCond        1459 non-null object
Foundation       1459 non-null object
BsmtQual         1415 non-null object
BsmtCond         1414 non-null object
BsmtExposure     1415 non-null object
BsmtFinType1     1417 non-null object
BsmtFinSF1       1458 non-null float64
BsmtFinType2     1417 non-null object
BsmtFinSF2       1458 non-null float64
BsmtUnfSF        1458 non-null float64
TotalBsmtSF      1458 non-null float64
Heating          1459 non-null object
HeatingQC        1459 non-null object
CentralAir       1459 non-null object
Electrical       1459 non-null object
1stFlrSF         1459 non-null int64
2ndFlrSF         1459 non-null int64
LowQualFinSF     1459 non-null int64
GrLivArea        1459 non-null int64
BsmtFullBath     1457 non-null float64
BsmtHalfBath     1457 non-null float64
FullBath         1459 non-null int64
HalfBath         1459 non-null int64
```

```
BedroomAbvGr     1459 non-null int64
KitchenAbvGr     1459 non-null int64
KitchenQual      1458 non-null object
TotRmsAbvGrd     1459 non-null int64
Functional       1457 non-null object
Fireplaces       1459 non-null int64
FireplaceQu      729 non-null object
GarageType       1383 non-null object
GarageYrBlt      1381 non-null float64
GarageFinish     1381 non-null object
GarageCars       1458 non-null float64
GarageArea       1458 non-null float64
GarageQual       1381 non-null object
GarageCond       1381 non-null object
PavedDrive       1459 non-null object
WoodDeckSF       1459 non-null int64
OpenPorchSF      1459 non-null int64
EnclosedPorch    1459 non-null int64
3SsnPorch        1459 non-null int64
ScreenPorch      1459 non-null int64
PoolArea         1459 non-null int64
PoolQC           3 non-null object
Fence            290 non-null object
MiscFeature      51 non-null object
MiscVal          1459 non-null int64
MoSold           1459 non-null int64
YrSold           1459 non-null int64
SaleType         1458 non-null object
SaleCondition    1459 non-null object
dtypes: float64(11), int64(26), object(43)
memory usage: 912.0+ KB
```

The test set contains 1459 rows.

The train set contains 81 columns. The first 80 of these also appear in the test set: these will be the features on which we will base our predictions. The final column, SalePrice, is our target variable.

In [9]:
```
#Save the test data set 'Id' column for final submission
test_ID = df_test['Id']

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

print("The train data size after dropping Id feature is : {} ".format(df_tr
print("The test data size after dropping Id feature is : {} ".format(df_tes
```

```
The train data size after dropping Id feature is : (1460, 80)
The test data size after dropping Id feature is : (1459, 79)
```

3.2 Variables' Distributions

In order to get deeper understanding of our data, we use three types of graphs to show the distribution of all variables in this dataset: kde graph, bar graph, and box graph. In this step, we write a function to describe data, if the data type is object and the value count is lower than 20, we use bar graph, if not, we used box and kde graph to show the value distribution.

3.2.1 Attributes variables

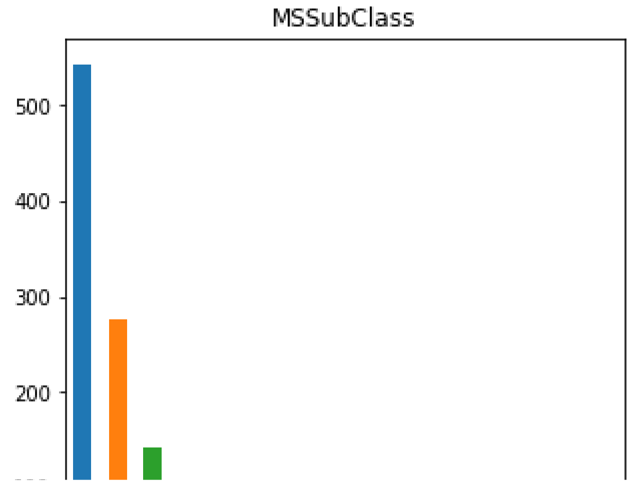
In [10]:
```
def describeData(DataFrame):
    feature_name=list(DataFrame.keys())
    data_type=[]
    for i in DataFrame.dtypes:
        data_type.append(str(i))
    for i in range(len(feature_name)):
        if data_type[i]=='object' or len(DataFrame.iloc[:,i].value_counts()
            print(DataFrame.iloc[:,i].describe())
            print(DataFrame.iloc[:,i].value_counts())
            DataFrame.iloc[:,i].value_counts().plot(kind='bar',figsize=(5,5
            plt.show()
        else:
            print(DataFrame.iloc[:,i].describe())
            DataFrame.iloc[:,i].plot(kind='box',figsize=(5,5),title=feature
            plt.show()
            DataFrame.iloc[:,i].plot(kind='kde',figsize=(5,5),title=feature
            plt.show()
        print('\n')
```

In [11]:
```
describeData(df_train)
```

```
count    1460.000000
mean       56.897260
std        42.300571
min        20.000000
25%        20.000000
50%        50.000000
75%        70.000000
max       190.000000
Name: MSSubClass, dtype: float64
20     536
60     299
50     144
120     87
30      69
160     63
70      60
80      58
90      52
190     30
```

In [12]:
```
describeData(df_test)
```

```
75         7
45         6
40         2
150        1
Name: MSSubClass, dtype: int64
```

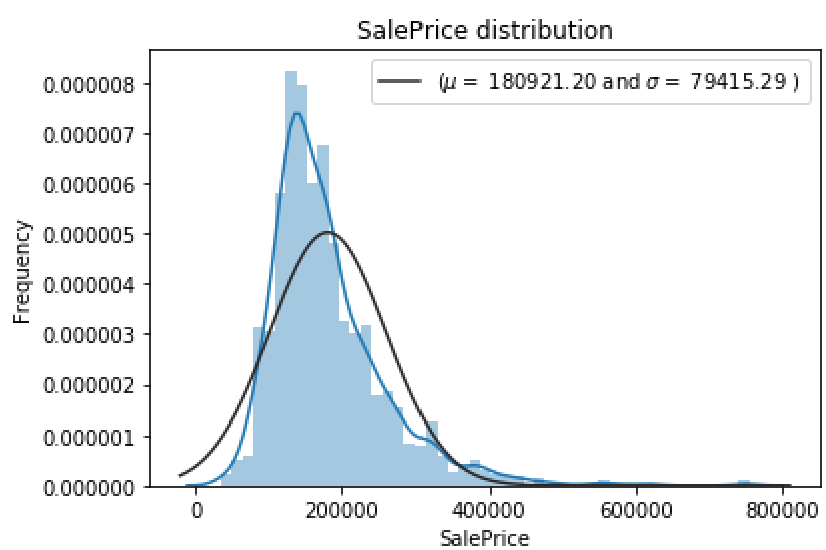


3.2.2 Target Variable: Sale Price

In [13]:
```
sns.distplot(df_train['SalePrice'], fit = norm);
(mu, sigma) = norm.fit(df_train['SalePrice'])
print( '\n mu = {:.2f} and sigma = {:.2f}\n'.format(mu, sigma))
plt.legend(['($\mu=$ {:.2f} and $\sigma=$ {:.2f} )'.format(mu, sigma)], loc
plt.ylabel('Frequency')
plt.title('SalePrice distribution')
```

```
 mu = 180921.20 and sigma = 79415.29
```

Out[13]: `Text(0.5,1,'SalePrice distribution')`



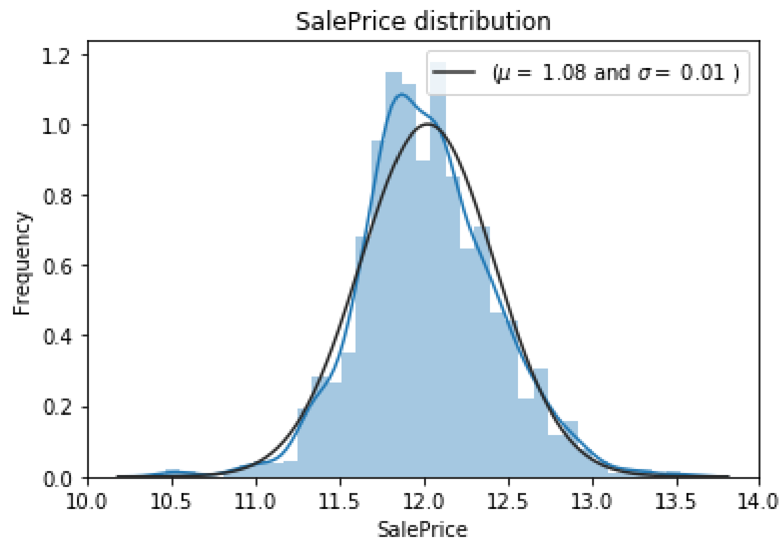
The distribution shows a preference to cheaper homes. To make the distribution more

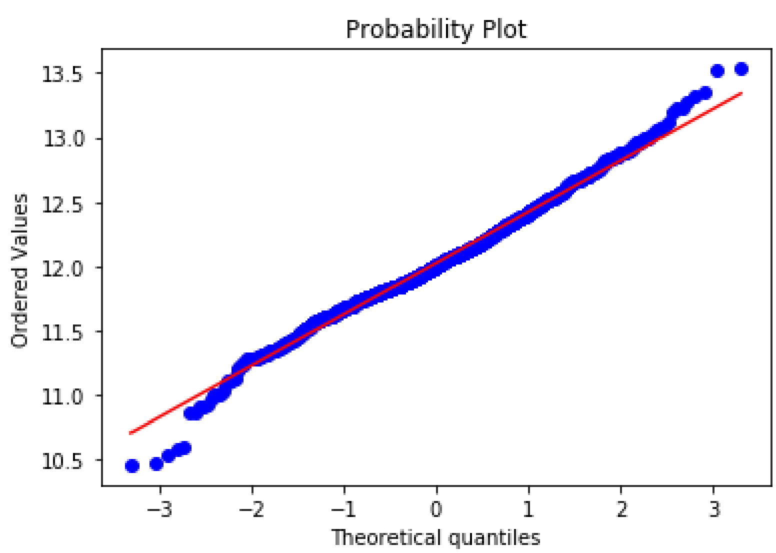
symmetric, we try taking its logarithm.

In [14]:
```
df_train["SalePrice"] = np.log1p(df_train["SalePrice"])
sns.distplot(df_train['SalePrice'], fit = norm);
(mu, sigma) = norm.fit(np.log10(df_train['SalePrice']))
print( '\n mu = {:.2f} and sigma = {:.2f}\n'.format(mu, sigma))
plt.legend(['($\mu=$ {:.2f} and $\sigma=$ {:.2f} )'.format(mu, sigma)], loc
plt.ylabel('Frequency')
plt.title('SalePrice distribution')

fig = plt.figure()
res = stats.probplot(df_train['SalePrice'], plot=plt)
plt.show()
```

```
 mu = 1.08 and sigma = 0.01
```



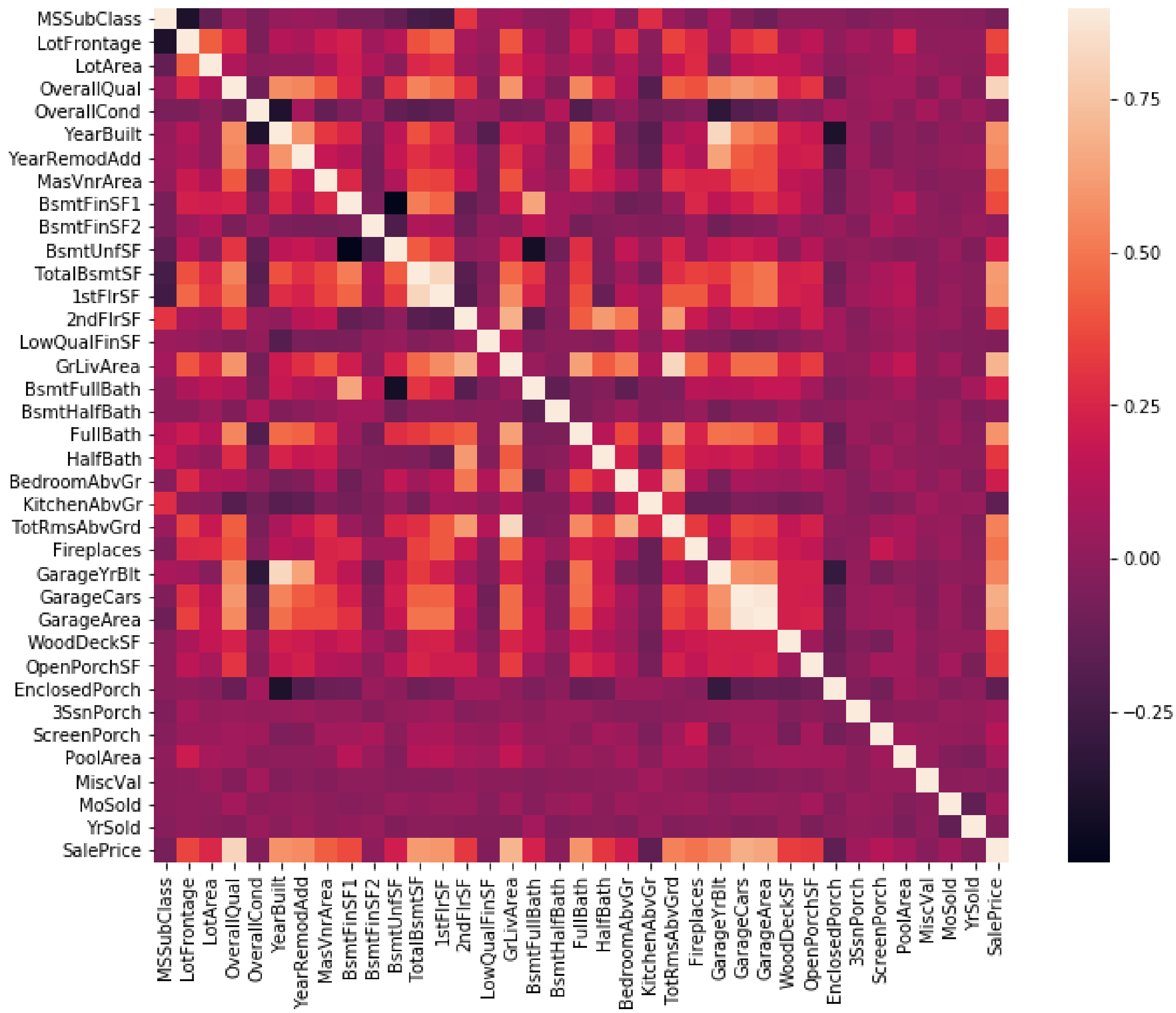


The data appears more normally distributed. Thus, we'll take log10(SalePrice) as target variable.

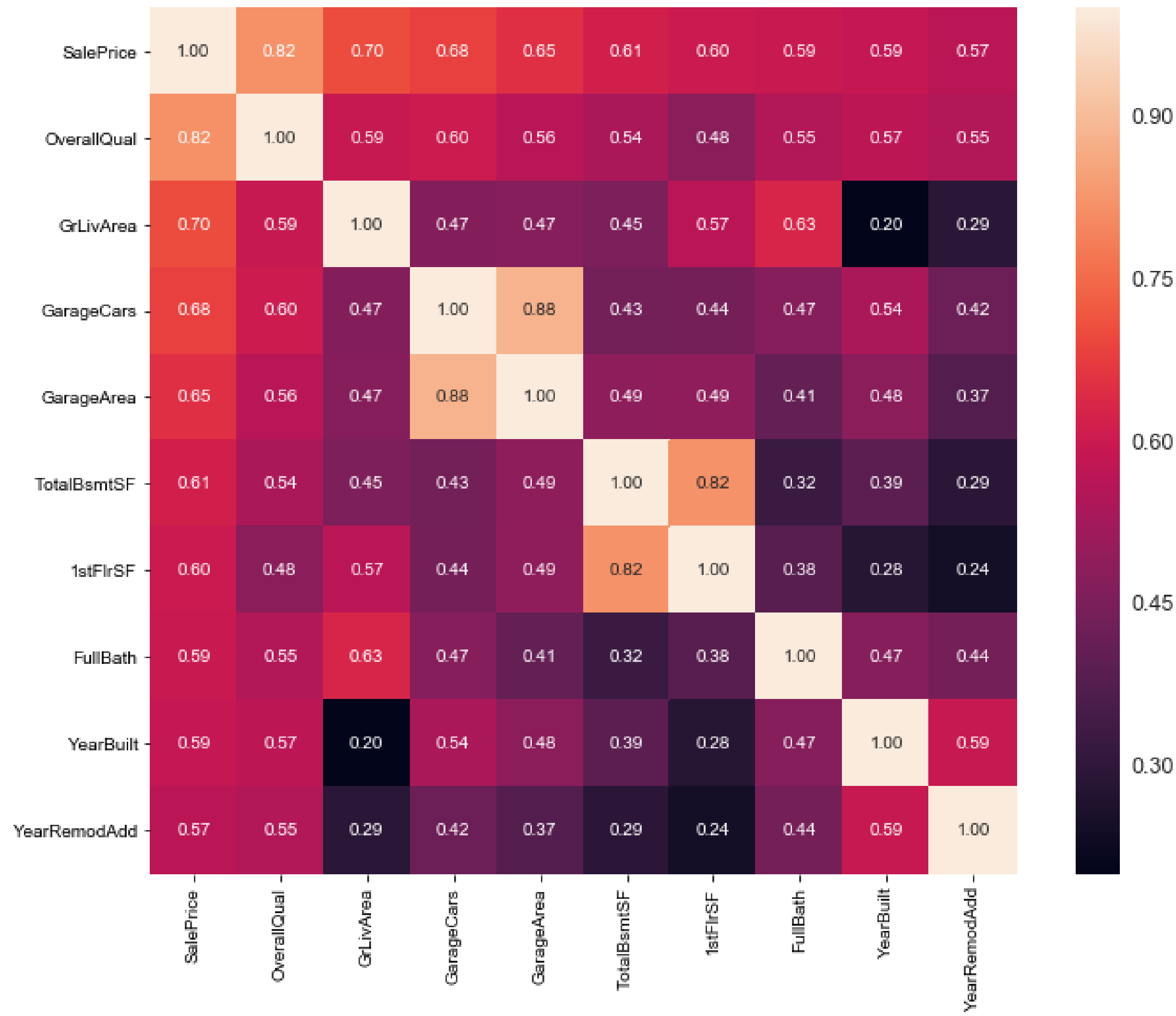
3.3 Correlation and Interactions

In [15]:
```
#Correlation map to see how features are correlated
corrmat = df_train.corr()
plt.subplots(figsize=(12,9))
sns.heatmap(corrmat, vmax=0.9, square=True)
```

Out[15]: <matplotlib.axes._subplots.AxesSubplot at 0x1a0ba004e0>



In [16]:
```
#saleprice correlation matrix
k = 10 #number of variables for heatmap
cols = corrmat.nlargest(k, 'SalePrice')['SalePrice'].index
plt.subplots(figsize=(12,9))
cm = np.corrcoef(df_train[cols].values.T)
sns.set(font_scale=1.25)
hm = sns.heatmap(cm, cbar=True, annot=True, square=True, fmt='.2f', annot_k
```



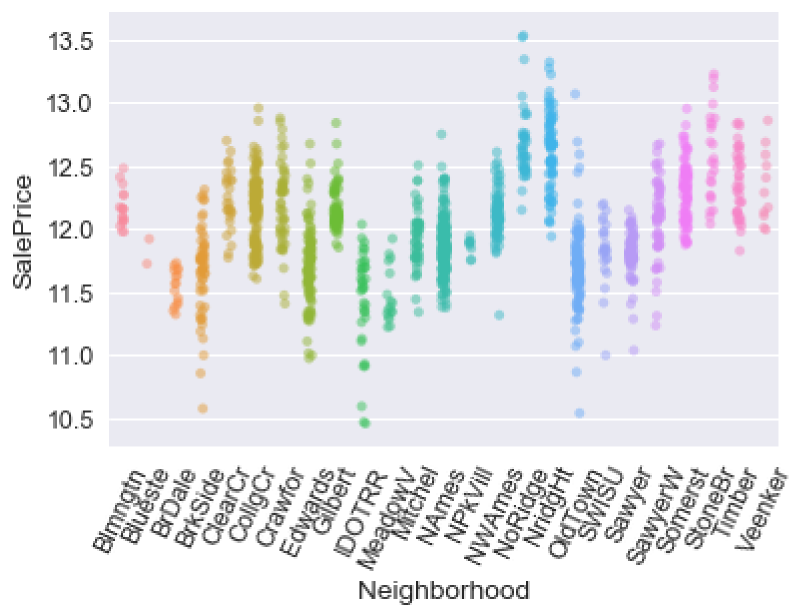
We can observe from the heatmap that the top 9 attributes which related to SalePrice mostly are OverallQual, GrLivArea, GarageCars, GarageArea, TotalBsmtSF, 1stFlrSF, FullBath, YearBuilt and YearRemodAdd.

Neighborhood vs. SalePrice

In [17]:
```
sns.stripplot(x = df_train.Neighborhood, y = df_train.SalePrice,
              order = np.sort(df_train.Neighborhood.unique()),
              jitter=0.1, alpha=0.5)

plt.xticks(rotation=65)
```

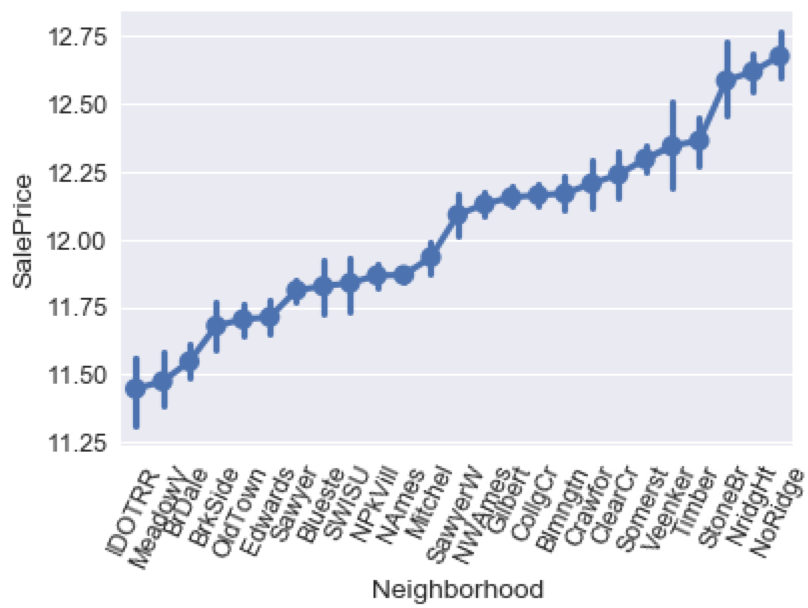
Out[17]:
```
(array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 1
6,
        17, 18, 19, 20, 21, 22, 23, 24]),
 <a list of 25 Text xticklabel objects>)
```



In [18]:
```
Neighborhood_meanSalePrice = df_train.groupby('Neighborhood')['SalePrice'].
Neighborhood_meanSalePrice = Neighborhood_meanSalePrice.sort_values()
sns.pointplot(x = df_train.Neighborhood, y = df_train.SalePrice,
              order = Neighborhood_meanSalePrice.index)

plt.xticks(rotation = 65)
```

Out[18]:
```
(array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 1
6,
        17, 18, 19, 20, 21, 22, 23, 24]),
 <a list of 25 Text xticklabel objects>)
```



3.4 Outliers

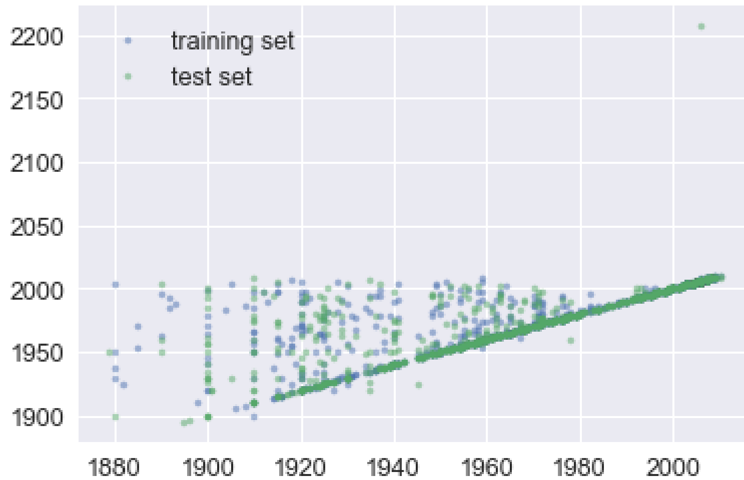
Original Construction Date vs Year Garage Was Built

In [19]:
```
plt.plot(df_train.YearBuilt, df_train.GarageYrBlt,
         '.', alpha=0.5, label = 'training set')

plt.plot(df_test.YearBuilt, df_test.GarageYrBlt,
         '.', alpha=0.5, label = 'test set')

plt.legend()
```

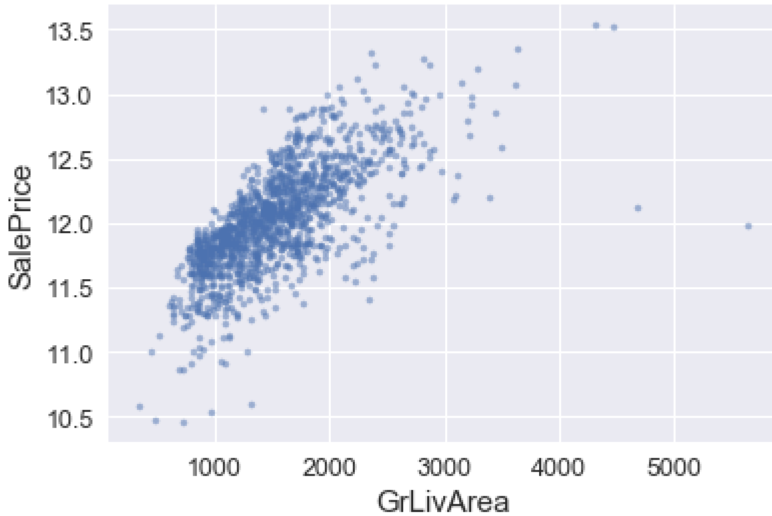
Out[19]: <matplotlib.legend.Legend at 0x1a179d26a0>



From the figure we can know that the majority of garages were built at the same time as the houses they belong to. We also see a number of strange points. In both train and test sets, we have several garages that were built as many as 10 years earlier than their houses, and in the test set we have a garage from later than 2200.

Ground Living Area Square Feet vs. Sale Price

In [20]:
```
plt.plot(df_train.GrLivArea, df_train.SalePrice, '.', alpha = 0.5)
plt.ylabel('SalePrice', fontsize=15)
plt.xlabel('GrLivArea', fontsize=15)
plt.show()
```

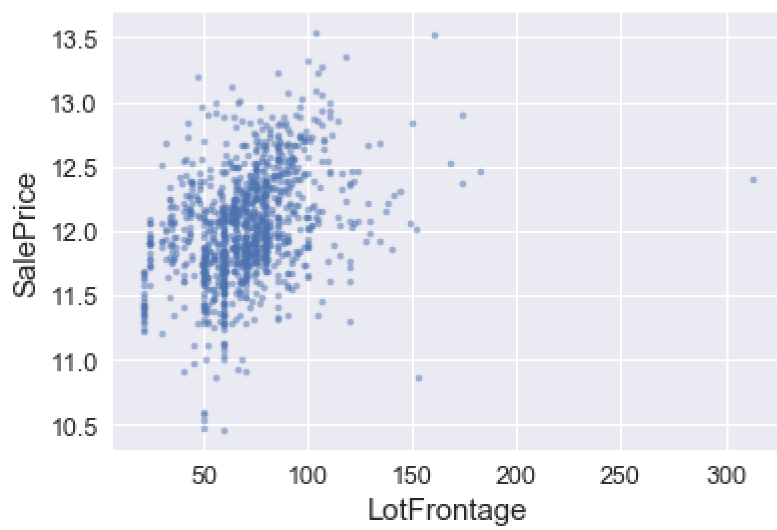


There is a strong dependence of sale price on the total living area. The larger the house, the more expensive it tends to be.
We can see there are two points towards the lower right part of the plot that don't seem to fit in with the rest. These two very large houses (bigger than 4000 sqft) with low sale prices. We can treat them as outliers and exclude them.

In [21]:
```
df_train = df_train.drop(df_train[(df_train['GrLivArea'] > 4000) & (df_trai
plt.plot(df_train.GrLivArea, df_train.SalePrice, '.', alpha = 0.5)
plt.ylabel('SalePrice', fontsize=15)
plt.xlabel('GrLivArea', fontsize=15)
plt.show()
```

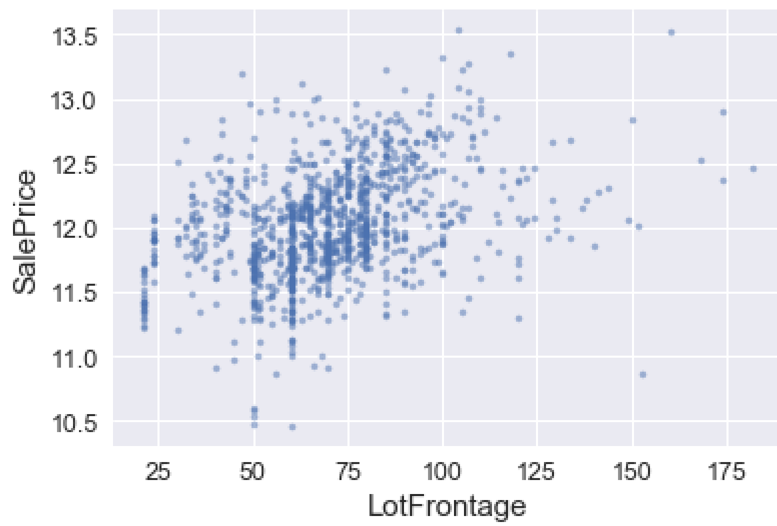
Lot Frontage vs. Sale Price

In [22]:
```
plt.plot(df_train.LotFrontage, df_train.SalePrice, '.', alpha = 0.5)
plt.ylabel('SalePrice', fontsize=15)
plt.xlabel('LotFrontage', fontsize=15)
plt.show()
```



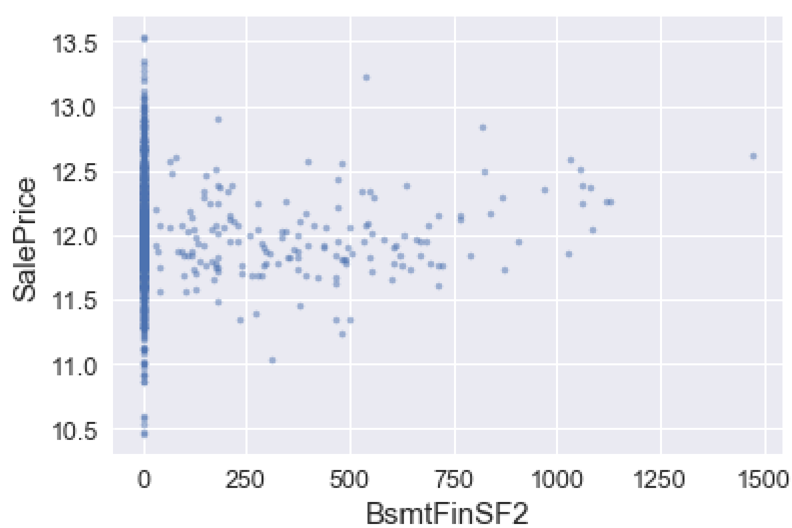
There is no strong dependence of sale price on the lot frontage.
We can see there are one point towards the upper right part of the plot that don't seem to fit in with the rest. We can treat it as a outlier and exclude it.

In [23]:
```
df_train = df_train.drop(df_train[(df_train['LotFrontage'] > 300) & (df_tra
plt.plot(df_train.LotFrontage, df_train.SalePrice, '.', alpha = 0.5)
plt.ylabel('SalePrice', fontsize=15)
plt.xlabel('LotFrontage', fontsize=15)
plt.show()
```

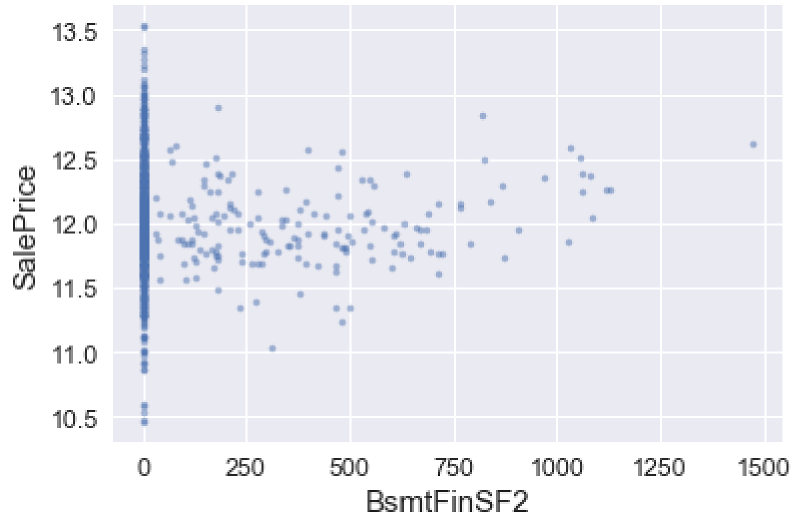


Type 2 Finished Basement Square Feet vs. Sale Price

In [24]:
```
plt.plot(df_train.BsmtFinSF2, df_train.SalePrice, '.', alpha = 0.5)
plt.ylabel('SalePrice', fontsize=15)
plt.xlabel('BsmtFinSF2', fontsize=15)
plt.show()
```



In [25]:
```
df_train = df_train.drop(df_train[(df_train['BsmtFinSF2'] > 400) & (df_trai
plt.plot(df_train.BsmtFinSF2, df_train.SalePrice, '.', alpha = 0.5)
plt.ylabel('SalePrice', fontsize=15)
plt.xlabel('BsmtFinSF2', fontsize=15)
plt.show()
```



3.5 Figure Out Missing Data

3.5.1 Find missing data

In [26]:
```
# Save data for redividing train and test data sets
original_train = df_train.shape[0]
y_train = df_train.SalePrice.values
```

In [27]:
```
df_whole = pd.concat((df_train, df_test)).reset_index(drop=True)
df_whole.drop(['SalePrice'], axis=1, inplace=True)
print("df_whole size is : {}".format(df_whole.shape))
```

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

In [28]:
```
def count_missing(data):
    null_cols = data.columns[data.isnull().any(axis=0)]
    X_null = data[null_cols].isnull().sum()
    print(X_null)
```

In [29]:
```
print(count_missing(df_whole))
```

```
Alley           2717
BsmtCond          82
BsmtExposure      82
BsmtFinSF1         1
BsmtFinSF2         1
BsmtFinType1      79
BsmtFinType2      80
BsmtFullBath       2
BsmtHalfBath       2
BsmtQual          81
BsmtUnfSF          1
Electrical         1
Exterior1st        1
Exterior2nd        1
Fence           2344
FireplaceQu     1420
Functional         2
GarageArea         1
GarageCars         1
GarageCond       159
GarageFinish     159
GarageQual       159
GarageType       157
GarageYrBlt      159
KitchenQual        1
LotFrontage      486
MSZoning           4
MasVnrArea        23
MasVnrType        24
MiscFeature     2810
PoolQC          2906
SaleType           1
TotalBsmtSF        1
Utilities          2
dtype: int64
None
```

In [30]:
```
df_whole_na = (df_whole.isnull().sum() / len(df_whole)) * 100
df_whole_na = df_whole_na.drop(df_whole_na[df_whole_na == 0].index).sort_va
missing_data = pd.DataFrame({'Missing Ratio' :df_whole_na})
missing_data.head(20)
```

Out[30]:

	Missing Ratio
PoolQC	99.691252
MiscFeature	96.397942
Alley	93.207547
Fence	80.411664
FireplaceQu	48.713551
LotFrontage	16.672384
GarageQual	5.454545
GarageCond	5.454545
GarageFinish	5.454545
GarageYrBlt	5.454545
GarageType	5.385935
BsmtExposure	2.813036
BsmtCond	2.813036
BsmtQual	2.778731
BsmtFinType2	2.744425
BsmtFinType1	2.710120
MasVnrType	0.823328
MasVnrArea	0.789022
MSZoning	0.137221
BsmtFullBath	0.068611

We can find that there is a large percentage of missing data in the dataset. However, most of those missing data is not meaningless. Basesd on our findings, we decided to fill up those data primaryly in four ways. First, we would fill up some missing value with "none", which means that this house does not have such attribute. For example, if the value for 'BsmtQual' is missing, it indicates that this house has no basement. The second way to deal with missing value is filling them with '0'. For exapmle, if this house has no basement, we would fill the 'BsmtFinSF1' and related quantitative missing value with 0, simply due to its absence of basement. The third way to fill up with missing value is to fill them with mode, which is the most frequent value in such attribute. Moreover, there are two exceptions. One is the grage year built, another one is lot frontage. We will illustrate how we deal with these two special case in the following part.

3.5.2 Make Up Missing Data

In [31]:
```
feats_fillnaNA = ['Alley', 'BsmtQual', 'BsmtCond', 'BsmtExposure', 'BsmtFin
    'FireplaceQu', 'GarageType', 'GarageFinish', 'GarageQual', 'GarageCond'

df_whole.loc[:, feats_fillnaNA] = df_whole[feats_fillnaNA].fillna('None')
```

In [32]:
```
feats_fillnaZERO = ['BsmtFullBath', 'BsmtHalfBath', 'TotalBsmtSF', 'BsmtFin
     'GarageArea', 'GarageCars', 'MasVnrArea']

df_whole.loc[:,feats_fillnaZERO] = df_whole[feats_fillnaZERO].fillna(0)
```

The missing values for garage, pool or basement-related features simply imply that the house does not have a garage, pool or basement respectively. Thus, it makes sense to fill these missing values with None or 0.

We'll make up the missing values of MSZoning, LotFrontage, Utilities, Exterior1st, Exterior2nd, Electrical, KitchenQual, Functional, GarageYrBlt, SaleType with further consideration.

For the house has no garage, we fill it with YearBuilt.

In [33]:
```
df_whole.loc[:,'GarageYrBlt'] = df_whole['GarageYrBlt'].fillna(df_whole.Yea
```

In [34]:
```
feats_fillnamode = ['Electrical', 'MSZoning', 'Exterior1st', 'Exterior2nd',
                       'KitchenQual', 'SaleType']

df_whole.loc[:, feats_fillnamode] = df_whole[feats_fillnamode].fillna(df_wh
```

For LotFrontage: since the area of each street connected to the house property most likely have a similar area to other houses in its neighborhood, we can fill in missing values by the median LotFrontage of the neighborhood.

In [35]:
```
df_whole["LotFrontage"] = df_whole.groupby("Neighborhood")["LotFrontage"].t
    lambda x: x.fillna(x.median()))
```

*** For Utilities: all records are "AllPub" in test data set, this feature won't help in predictive modelling. We can remove it.***

In [36]:
```
df_whole = df_whole.drop(['Utilities'], axis=1)
```

In [37]:
```
df_whole["Functional"] = df_whole["Functional"].fillna("Typ")
```

In [38]:
```
print(count_missing(df_whole))
```

```
Series([], dtype: float64)
None
```

3.6. Feature Engineering

Transform Numerical Variables

In [39]:
```
df_whole['MSSubClass'] = df_whole['MSSubClass'].apply(str)
df_whole['OverallCond'] = df_whole['OverallCond'].astype(str)
df_whole['YrSold'] = df_whole['YrSold'].astype(str)
df_whole['MoSold'] = df_whole['MoSold'].astype(str)
```

Apply LabelEncoder to Categorical Features

In [40]:
```
categorical_features = ('FireplaceQu', 'BsmtQual', 'BsmtCond', 'GarageQual'
        'ExterQual', 'ExterCond','HeatingQC', 'PoolQC', 'KitchenQual', 'Bsm
        'BsmtFinType2', 'Functional', 'Fence', 'BsmtExposure', 'GarageFinis
        'LotShape', 'PavedDrive', 'Street', 'Alley', 'CentralAir', 'MSSubCl
        'YrSold', 'MoSold')
for feat in categorical_features:
    lb = LabelEncoder()
    lb.fit(list(df_whole[feat].values))
    df_whole[feat] = lb.transform(list(df_whole[feat].values))

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

```
Shape all_data: (2915, 78)
```

***Adding total square feet feature ***

In [41]:
```
df_whole['TotalSF'] = df_whole['TotalBsmtSF'] + df_whole['1stFlrSF'] + df_w
```

Skewed Features

In [42]:
```
numeric_feats = df_whole.dtypes[df_whole.dtypes != "object"].index

skewed_feats = df_whole[numeric_feats].apply(lambda x: skew(x.dropna())).sc
print("\nSkew in numerical features: \n")
skewness = pd.DataFrame({'Skew' :skewed_feats})
skewness.head(10)
```

```
Skew in numerical features:
```

Out[42]:

	Skew
MiscVal	21.932147
PoolArea	17.682542
LotArea	13.141138
LowQualFinSF	12.080315
3SsnPorch	11.368094
LandSlope	4.993598
KitchenAbvGr	4.298845
BsmtFinSF2	4.155517
EnclosedPorch	4.000796
ScreenPorch	3.954650

In [43]:
```
skewness = skewness[abs(skewness) > 0.75]

skewed_features = skewness.index
lam = 0.15
for feat in skewed_features:
    df_whole[feat] = boxcox1p(df_whole[feat], lam)
```

Transform Object Features by get_dummies

In [44]:
```
df_whole = pd.get_dummies(df_whole)
print(df_whole.shape)
```

```
(2915, 220)
```

Divide df_whole into Train and Test

In [45]:
```
df_train = df_whole[:original_train]
df_test = df_whole[original_train:]
```

4. Methodology and Settings

In [46]:
```
n_folds = 5

def checkAccuracy(model):
    kf = KFold(n_folds, shuffle = True, random_state = 50).get_n_splits(df_
    accuracy_score = np.sqrt(-cross_val_score(model, df_train.values, y_tra
    return(accuracy_score)
```

Lasso

In [47]:
```
lasso = make_pipeline(RobustScaler(), Lasso(alpha = 0.0005, random_state =
score = checkAccuracy(lasso)
print("Lasso score: {:.4f} ({:.4f})\n".format(score.mean(), score.std()))
```

```
Lasso score: 0.1117 (0.0073)
```

XGBoost

In [48]:
```
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,
                             random_state = 7, nthread = -1)
score = checkAccuracy(model_xgb)
print("XGBoost score: {:.4f} ({:.4f})\n".format(score.mean(), score.std()))
```

```
XGBoost score: 0.1155 (0.0062)
```

LightGBM

In [49]:
```
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
score = checkAccuracy(model_lgb)
print("LightGBM score: {:.4f} ({:.4f})\n".format(score.mean(), score.std())
```

```
LightGBM score: 0.1158 (0.0064)
```

Gradient Boosting Regression

In [50]:
```
GBoost = GradientBoostingRegressor(n_estimators = 3000, learning_rate = 0.0
                                   max_depth = 4, max_features = 'sqrt',
                                   min_samples_leaf = 15, min_samples_split
                                   loss = 'huber', random_state = 5)
score = checkAccuracy(GBoost)
print("Gradient Boosting Regression score: {:.4f} ({:.4f})\n".format(score.
```

```
Gradient Boosting Regression score: 0.1175 (0.0086)
```

Elastic Net Regression

In [51]:
```
ENet = make_pipeline(RobustScaler(), ElasticNet(alpha = 0.0005, l1_ratio =
score = checkAccuracy(ENet)
print("Elastic Net Regression score: {:.4f} ({:.4f})\n".format(score.mean()
```

```
Elastic Net Regression score: 0.1117 (0.0074)
```

Kernel Ridge Regression

In [52]:
```
KRR = KernelRidge(alpha = 0.6, kernel = 'polynomial', degree = 2, coef0 = 2
score = checkAccuracy(KRR)
print("Kernel Ridge Regression score: {:.4f} ({:.4f})\n".format(score.mean(
```

```
Kernel Ridge Regression score: 0.1149 (0.0081)
```

Stacking Models

Stacking averaged Models Class

In [53]:
```
class StackingAveragedModels(BaseEstimator, RegressorMixin, TransformerMixi
    def __init__(self, base_models, meta_model, n_folds = 5):
        self.base_models = base_models
        self.meta_model = meta_model
        self.n_folds = n_folds

    def fit(self, x, y):
        self.base_models_ = [list() for x in self.base_models]
        self.meta_model_ = clone(self.meta_model)
        kfold = KFold(n_splits = self.n_folds, shuffle = True, random_state

        out_of_fold_predictions = np.zeros((x.shape[0], len(self.base_model
        for i, model in enumerate(self.base_models):
            for train_index, holdout_index in kfold.split(x, y):
                instance = clone(model)
                self.base_models_[i].append(instance)
                instance.fit(x[train_index], y[train_index])
                y_pred = instance.predict(x[holdout_index])
                out_of_fold_predictions[holdout_index, i] = y_pred

        self.meta_model_.fit(out_of_fold_predictions, y)
        return self

    def predict(self, x):
        meta_features = np.column_stack([
            np.column_stack([model.predict(x) for model in base_models]).me
            for base_models in self.base_models_ ])
        return self.meta_model_.predict(meta_features)
```

In [ ]:
```
stacked_averaged_models = StackingAveragedModels(base_models = (ENet, GBoos
                                                 meta_model = lasso)

score = checkAccuracy(stacked_averaged_models)
print("Stacking Averaged models score: {:.4f} ({:.4f})".format(score.mean()
```

5. Prediction Results

5.1 Candidate models

In [ ]:
```
def getSquaredError(y, y_pred):
    return np.sqrt(mean_squared_error(y, y_pred))
```

StackedRegressor

In [ ]:
```
stacked_averaged_models.fit(df_train.values, y_train)
stacked_train_pred = stacked_averaged_models.predict(df_train.values)
stacked_pred = np.expm1(stacked_averaged_models.predict(df_test.values))
print(getSquaredError(y_train, stacked_train_pred))
```

XGBoost

In [ ]:
```
model_xgb.fit(df_train, y_train)
xgb_train_pred = model_xgb.predict(df_train)
xgb_pred = np.expm1(model_xgb.predict(df_test))
print(getSquaredError(y_train, xgb_train_pred))
```

LightGBM

In [ ]:
```
model_lgb.fit(df_train, y_train)
lgb_train_pred = model_lgb.predict(df_train)
lgb_pred = np.expm1(model_lgb.predict(df_test.values))
print(getSquaredError(y_train, lgb_train_pred))
```

In [ ]:
```
print('score on train data set:')
print(getSquaredError(y_train, stacked_train_pred * 0.65 +
               xgb_train_pred * 0.15 + lgb_train_pred * 0.2 ))
```

5.2 Ensemble models

In [ ]:
```
ensemble = stacked_pred * 0.65 + xgb_pred * 0.15 + lgb_pred * 0.2
```

5.3 Submission and results

In [ ]:
```
sub = pd.DataFrame()
sub['Id'] = test_ID
sub['SalePrice'] = ensemble
sub.to_csv('submission.csv',index=False)
```

6. Discussion

As our original design, the overal performance is about 0.12 according to kaggle score. In order to improve the performance and score, we compared our design with some designs on the Kernels. After comparison, we found our weakness is in the step of dealing with features. We just did log to correct the skewness of features. It seems that this is not enoght to improve its normalization. Therefore, we adopted box cox, which is a way to eliminate the skewness, to improve the performance. After we used the box cox, the overal score is improved.

199	new	Yesi Xie	0.11521	8	now

7. Conclussion

As we can see from the analysis, there are lots of factors contributing to the sale price of houses.

Some of them might be easy to identify, while others would be revealed by data analysis. Ensembling models is a important step when we want to improve the performance of models. Through the different ways of ensembling, it will lead to different performance and scores. For our design, the stacked model, xgboost, and light gbm models have the best performance when ensembling together. Moreover, the box cox is also an important way to deal with the skewness of features, which would improve the accuracy of our predictions.

8. Bibliography

(n.d.). Retrieved March 26, 2018, from https://www.kaggle.com/wiki/Home (https://www.kaggle.com/wiki/Home)

Brett Romero, Data Science: A Kaggle Walkthrough - Introduction. Retrieved March 26, 2018, from http://brettromero.com/data-science-a-kaggle-walkthrough-introduction/ (http://brettromero.com/data-science-a-kaggle-walkthrough-introduction/)

C. M. Bishop, Pattern Recognition and Machine Learning. Springer, 2006.

Alex Seutin, Ian Jones, Using Machine Learning to Predict Housing Prices Given Multivariate Input. Fall, 2016

Jason Brownlee, (2016, September 21). A Gentle Introduction to XGBoost for Applied Machine Learning. Retrieved March 26, 2018, from https://machinelearningmastery.com/gentle-introduction-xgboost-applied-machine-learning/ (https://machinelearningmastery.com/gentle-introduction-xgboost-applied-machine-learning/)

Ofir Chakon, Practical machine learning: Ridge regression vs. Lasso. (2017, August 10). Retrieved March 26, 2018, from https://codingstartups.com/practical-machine-learning-ridge-regression-vs-lasso/ (https://codingstartups.com/practical-machine-learning-ridge-regression-vs-lasso/)

Scott, D. (n.d.). Box-Cox Transformations. Retrieved from http://onlinestatbook.com/2/transformations/box-cox.html (http://onlinestatbook.com/2/transformations/box-cox.html)

Serigne. (n.d.). Stacked Regressions to predict House Prices. Retrieved April 26, 2018, from https://www.kaggle.com/serigne/stacked-regressions-top-4-on-leaderboard (https://www.kaggle.com/serigne/stacked-regressions-top-4-on-leaderboard)