Kaggle Competition:
House Prices
--Advanced Regression Techniques

--- Zhongyu YAO

Our Best Ranking(Top 15%):

1 Active Competition

House Prices: Advanced Regression Techniques

Predict sales prices and practice feature engineering, RFs, and gradient boosting

Getting Started · Ongoing · tabular data, regression

667/4696
Top 15%

667	▲1040	**CodeMonkeys**	0.11729	20	17h

Your Best Entry ↑

Your submission scored 0.11729, which is not an improvement of your best score. Keep trying!

First of all, in the competition, our team ranks around 15%

1.0-3.0 Version:

- 1.0 version: - Only one adding feature “TotalArea”;
  -Falsely using mean values to fill in the numeric attributes, e.g, the house has no garage, but fill in the garage with the average value of this group(do group by first);
  -One hot encoding;
  -Linar Regression Model;

- 2.0 version: - Adding more features;
  - Fill in the missing value with relatively correct method;
  - One-hot encoding;
  - Linear regression model,Elastic Net Regression, Lasso Regression.

- 3.0 version: - Model Enhancement on the basis of 2.0 version;
  - Add more features;
  - Gradient Boosting Regression, Ridge Regression;
  - Stack the models with relatively good performances together to realize the prediction.

Data Description

Lets look at our work more specifically. To begin with we put the data in pandas dataframe to observed the data types, the number, the distribution and features of these data.

```
train.info();
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1460 entries, 0 to 1459
Data columns (total 81 columns):
```

```
SalePrice         1460 non-null int64
dtypes: float64(3), int64(35), object(43)
memory usage: 924.0+ KB
```

```
test.info();
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1459 entries, 0 to 1458
Data columns (total 80 columns):
```

```
dtypes: float64(11), int64(26), object(43)
memory usage: 912.0+ KB
```

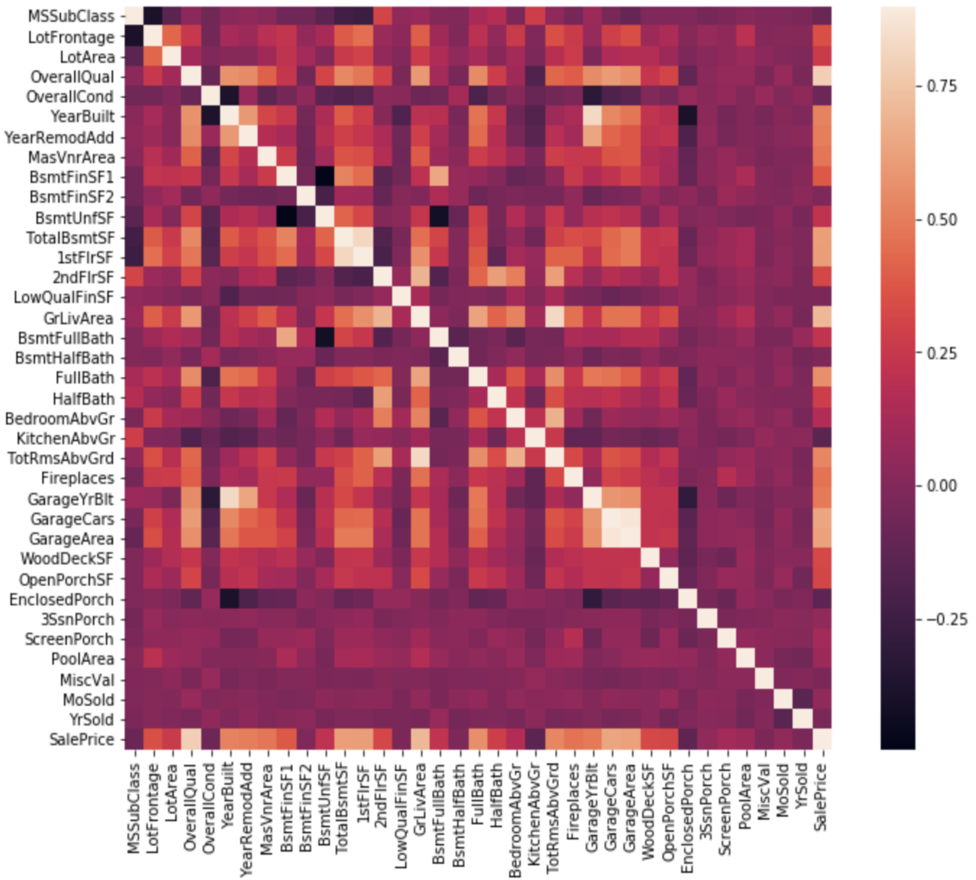
Correlation Analysis

Only Numerical values!

After that, we discussed how to find the outliers effectively. And Finding that only deleting outliers highly correlated with saleprice matters!
So we use function of corr to find out how features are correlated with SalePrice. As the heatmap showing, the red deeper, the correlation higher.
Btw, Correlation Analysis is only suitable for numerical values.

```python
#do correlation analysis
corrmat = train.corr()
plt.subplots(figsize=(12,9))
sns.heatmap(corrmat, vmax=0.9, square=True)
```

<matplotlib.axes._subplots.AxesSubplot at 0x1a2c4cf438>



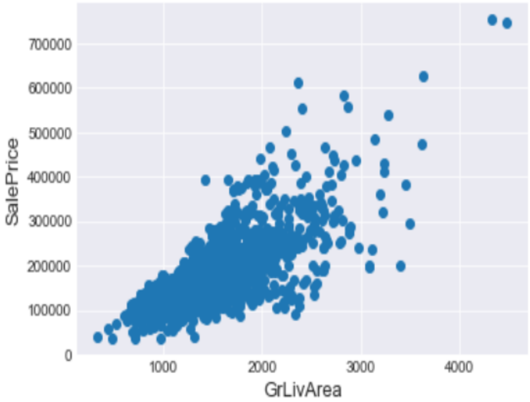
Discard Outliers --- e.g ‘GrLivArea’

In [5]:
```
import matplotlib.pyplot as plt
%matplotlib inline

fig, ax = plt.subplots()
ax.scatter(x = train['GrLivArea'], y = train['SalePrice'])
plt.ylabel('SalePrice', fontsize=13)
plt.xlabel('GrLivArea', fontsize=13)
plt.show()
```

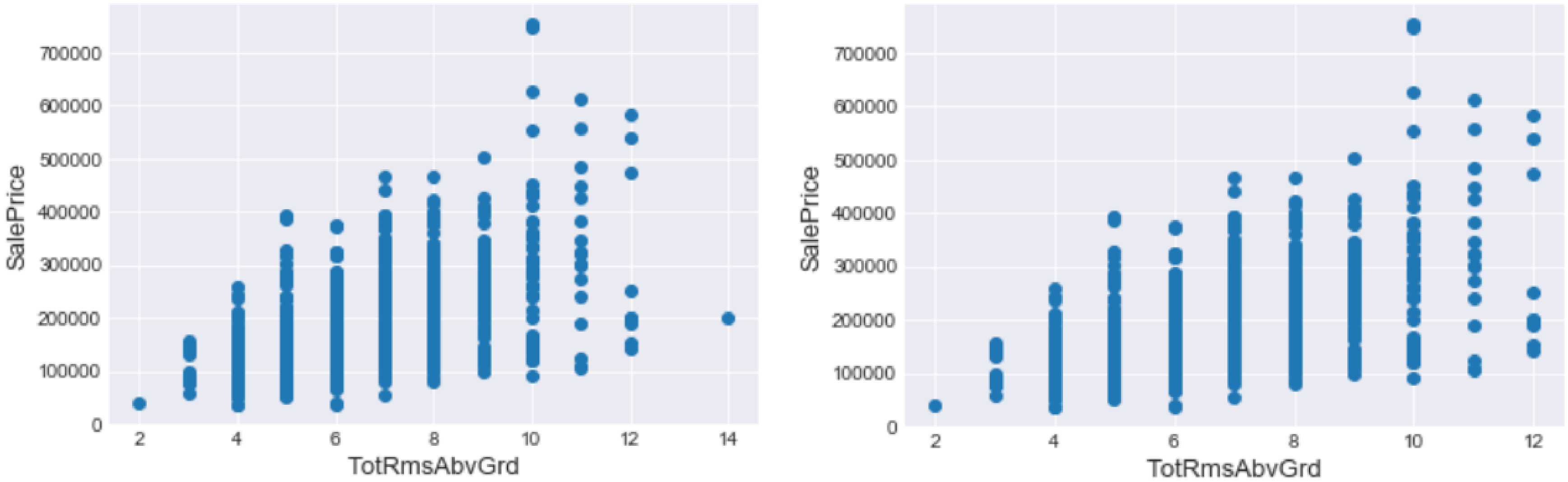
In [6]:
```
train = train.drop(train[(train['GrLivArea']>4000) & (train['SalePrice']<300000)].index)

#Check the graphic again
fig, ax = plt.subplots()
ax.scatter(train['GrLivArea'], train['SalePrice'])
plt.ylabel('SalePrice', fontsize=13)
plt.xlabel('GrLivArea', fontsize=13)
plt.show()
```



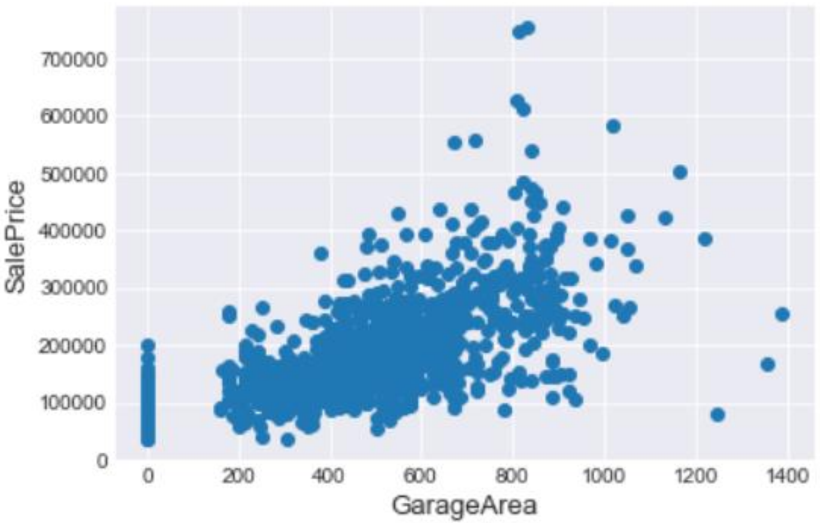
Then, we choose five features based on Correlation Analysis to explore the outliers and to delete them. (the value of the five are larger than 0.5)
In the correlation between ground living area square feet and saleprice, We can see at the bottom right two with extremely large GrLivArea that are of a low price. These values are huge oultliers. Therefore, we can safely delete them.

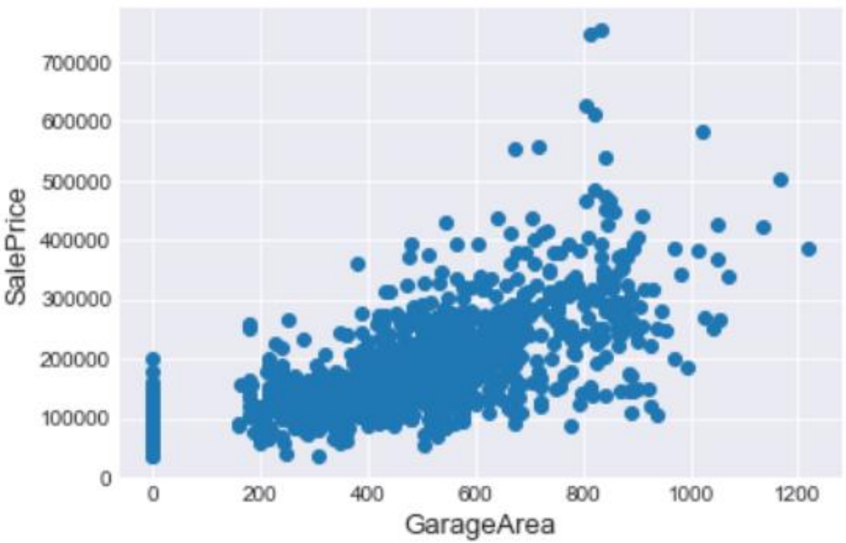
Discard Outliers---TotRmsAbvGrd



And for Total rooms above ground, we deleted this point.

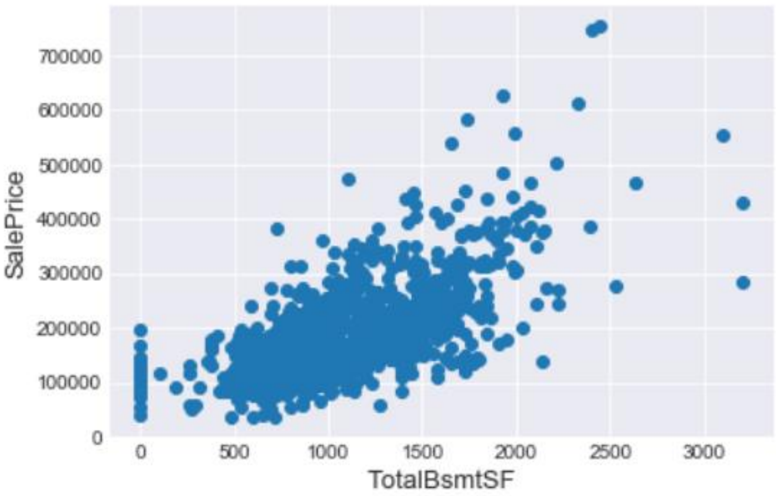
Discard Outliers---GarageArea

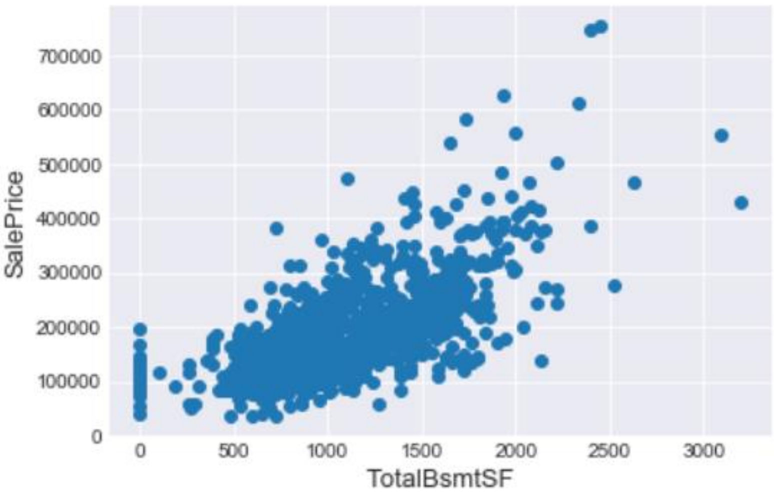




And for Size of garage, we deleted these three points.

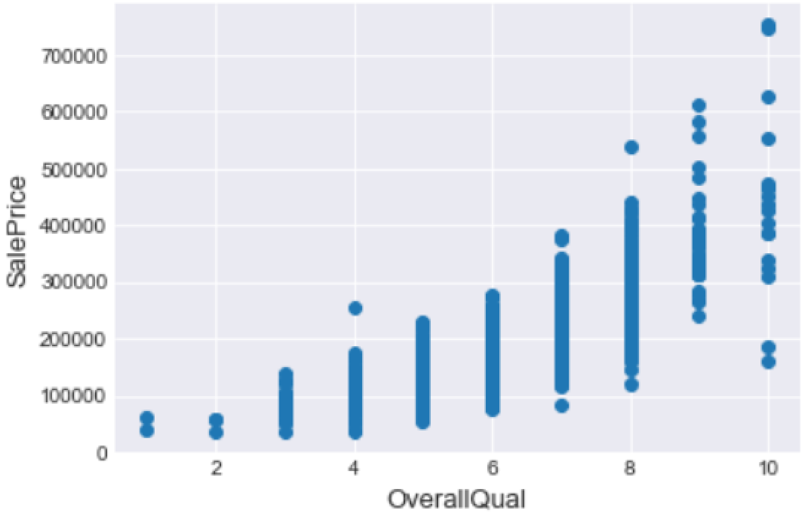
Discard Outliers---TotalBsmtSF

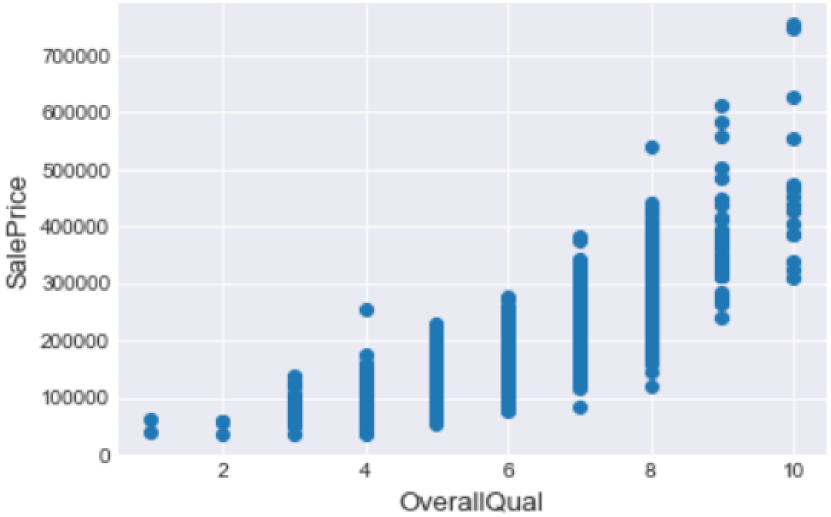




For Total square feet of basement area, we deleted this points.

Discard Outliers---OverallQual





For Overall material and finish quality, we deleted these two points.

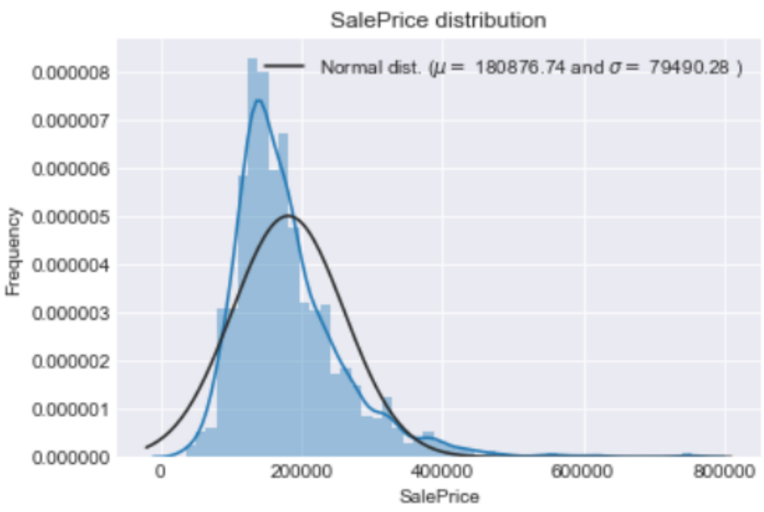
Skewness Adjustment

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

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

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

As linear models love normally distributed date so we analyzed how the saleprice distribute but found that it is right skewed



Skewness Adjustment

Remember converting back after prediction!

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

#Check the new distribution
sns.distplot(train['SalePrice'] , fit=norm);

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

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

So we use Log-transformation to pull it at the center.
And there is one thing I have to mention that do
Remember converting back after prediction!
Because at first time we forget to convert the normally distributed to the origional one we got very pool score.

Filling In Missing Value -- Merge train & test

Calculate Missing Ratio

```
all_data_na = (all_data.isnull().sum() / len(all_data)) * 100
all_data_na = all_data_na.drop(all_data_na[all_data_na == 0].index).sort_values(ascending=False)[:30]
missing_data = pd.DataFrame({'Missing Ratio' :all_data_na})
```

Then we try to find the missing value and to fill in them. Here we use pandas function: point is_null divided by the length of all data to calculate the missing ratio

	Missing Ratio
PoolQC	99.690934
MiscFeature	96.428571
Alley	93.234890
Fence	80.391484
FireplaceQu	48.695055
LotFrontage	16.655220
GarageQual	5.425824
GarageCond	5.425824
GarageFinish	5.425824
GarageYrBlt	5.425824

Filling In Missing Value --- Merge train & test

Fill in non-numeric value

NA has meaning: fill with “None”

```
all_data["PoolQC"] = all_data["PoolQC"].fillna("None")
all_data["MiscFeature"] = all_data["MiscFeature"].fillna("None")
all_data["Alley"] = all_data["Alley"].fillna("None")
all_data["Fence"] = all_data["Fence"].fillna("None")
all_data["FireplaceQu"] = all_data["FireplaceQu"].fillna("None")
for col in ('GarageType', 'GarageFinish', 'GarageQual', 'GarageCond'):
    all_data[col] = all_data[col].fillna('None')
for col in ('BsmtQual', 'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFinType2'):
    all_data[col] = all_data[col].fillna('None')
all_data["MasVnrType"] = all_data["MasVnrType"].fillna("None")
all_data['MSSubClass'] = all_data['MSSubClass'].fillna("None")
```

```
for col in ('BsmtQual', 'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFinType2'):
    all_data[col] = all_data[col].fillna('None')
```

Specifically, for non-numeric missing value, if missing value has meaning, we fill with “None”. For example, missing value of PoolQC means no pool so we fill with none

Filling In Missing Value --- Merge train & test

Fill in non-numeric value

NA has no meaning: fill with mode

```
#categorical attributes that NA has no meaning
for col2 in ('MSZoning', 'Utilities', 'Functional', 'Electrical', 'Exterior1st', 'Exterior2nd', 'SaleType', 'KitchenQual'):
    all_data[col2] = all_data[col2].fillna(all_data[col2].mode()[0])
```

```
all_data['MSZoning'] = all_data['MSZoning'].fillna(all_data['MSZoning'].mode()[0])
all_data["Functional"] = all_data["Functional"].fillna("Typ")
all_data['Electrical'] = all_data['Electrical'].fillna(all_data['Electrical'].mode()[0])
all_data['KitchenQual'] = all_data['KitchenQual'].fillna(all_data['KitchenQual'].mode()[0])
all_data['Exterior1st'] = all_data['Exterior1st'].fillna(all_data['Exterior1st'].mode()[0])
all_data['Exterior2nd'] = all_data['Exterior2nd'].fillna(all_data['Exterior2nd'].mode()[0])
all_data['SaleType'] = all_data['SaleType'].fillna(all_data['SaleType'].mode()[0])
```

and, for non-numeric missing value, if missing value has no meaning, we fill with mode.

Filling In Missing Value --- Merge train & test

Fill in numeric value

Group by → fill with median

```
#Group by neighborhood and fill in missing value by the median LotFrontage of all the neighborhood
all_data["LotFrontage"] = all_data.groupby("Neighborhood")["LotFrontage"].transform(
    lambda x: x.fillna(x.median()))
```

The house has no garage or no basement, its numeric values will be filled with 0:

```
for col in ('GarageYrBlt', 'GarageArea', 'GarageCars'):
    all_data[col] = all_data[col].fillna(0)
```

```
for col in ('BsmtFinSF1', 'BsmtFinSF2', 'BsmtUnfSF','TotalBsmtSF', 'BsmtFullBath', 'BsmtHalfBath'):
    all_data[col] = all_data[col].fillna(0)
```

```
all_data["MasVnrArea"] = all_data["MasVnrArea"].fillna(0)
```

and, for numeric missing value, if missing value has no meaning, we fill with median and if it has mean we fill with 0. For example, here, a house has no basement, its numeric values will be filled with 0.

Filling In Missing Value --- Merge train & test

Re-check Missing Ratio

```
#Check remaining missing values if any
all_data_na = (all_data.isnull().sum() / len(all_data)) * 100
all_data_na = all_data_na.drop(all_data_na[all_data_na == 0].index).sort_values(ascending=False)
missing_data = pd.DataFrame({'Missing Ratio' :all_data_na})
missing_data.head()
```

Missing Ratio

Lastly we check the missing value again to make sure we add all the missing values.

Feature Engineering:

```python
def transform(X):
    #地下室总面积+1 2 层面积
    X["TotalHouse"] = X["TotalBsmtSF"] + X["1stFlrSF"] + X["2ndFlrSF"]
    #地下室总面积+1 2 层面积 + 车库+ 泳池
    X["TotalArea"] = X["TotalBsmtSF"] + X["1stFlrSF"] + X["2ndFlrSF"] + X["GarageArea"] + X["PoolArea"]
    X["+_GrLivArea_OverallQual"] = X["GrLivArea"] * X["OverallQual"]
    X["+_BsmtFinSF1_OverallQual"] = X["BsmtFinSF1"] * X["OverallQual"]
    X["-_LotArea_OverallQual"] = X["LotArea"] * X["OverallQual"]
    X["-_TotalHouse_LotArea"] = X["TotalHouse"] + X["LotArea"]
    #地上总房间数
    X["Rooms"] = X["FullBath"]+X["TotRmsAbvGrd"] + X["HalfBath"]
    #总门廊面积
    X["PorchArea"] = X["OpenPorchSF"]+X["EnclosedPorch"]+X["3SsnPorch"]+X["ScreenPorch"]
    #总的所有面积
    X["TotalPlace"] = X["TotalBsmtSF"] + X["1stFlrSF"] + X["2ndFlrSF"] + X["GarageArea"] + X["PoolArea"]
    + X["OpenPorchSF"]+X["EnclosedPorch"]+X["3SsnPorch"]+X["ScreenPorch"]
    #总卫生间数
    X["TotalBaths"] = X["BsmtFullBath"] + X["BsmtHalfBath"] + X["FullBath"] +X["HalfBath"]
    #已完工高质量面积
    X["FinishedHQArea"] = X["BsmtFinSF1"] + X["BsmtFinSF2"] - X["LowQualFinSF"]
    #设施面积
    X["FacilityArea"]=X["GarageArea"] + X["PoolArea"]
    #地下总房间数
    X["BaseTotalRoom"]= X["BsmtFullBath"] + X["BsmtHalfBath"]
    return X
```

Furthermore, in feature engineering period, we added 13 more features. For example, Here we regard the sum of Unfinished square feet of basement area, First Floor square feet and second Floor square feet as total house square. Similar ideal can be found at the remaining 12 features

One-hot Encoding:

Data after adding features:

```
X=transform(all_data)
X.shape
```

(2913, 92)

Data after one-hot encoding:

```
X = pd.get_dummies(X)
print(X.shape)
```

(2913, 314)

To make the data fit our models and make our models more efficiently, we use one hot encoding to convert the categorical values to the continuous or numeric values. Specifically, we use pandas function get dummies here to realize.

Train Model:

Cross Validation & Definition of RMSE

```
#Validation function
n_folds = 10

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

Finally, we start to train the model. Before training, we write the same evaluation method of this competition, Root-mean-square deviation. As we all know, it is a frequently used measurement of the differences between values predicted by a model and the values observed.

In the method, we use 10_folds cross validation.

Train Model:

1. Linear Regression

Linear Regression without normalizatio
n feature scaling:

2. Measure Metrics --- mean_squared_error & RMSE

```
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

x_train, x_test, y_train, y_test = train_test_split(X_lr, y_lr, random_state = 42, test_size = 0.3)
reg = LinearRegression().fit(x_train, y_train)
predictions = reg.predict(x_test)
mean_squared_error(y_test, predictions)
```

```
3.4960272674390116
```

```
kf = KFold(n_folds, shuffle=True, random_state=42).get_n_splits(X_lr)
score= np.sqrt(-cross_val_score(reg, X_lr, y_lr, scoring="neg_mean_squared_error", cv = kf))
score1 = np.delete(score, 6)
print("\nLR score: {:.4f} ({:.4f})\n".format(score1.mean(), score1.std()))
```

```
LR score: 0.1249 (0.0112)
```

Our first model is linear regression, which can be imported from sklearn. The result is 0.12sth, just so so. So we try to improve it by using normalizing the feature.

Train Model: 1. Linear Regression

Linear Regression **AFTER normalization feature scaling:**

2 Measure Metrics --- mean_squared_error & RMSE

```
X_lr_sc
```

```
array([[0.18037319, 0.41355932, 0.        , ..., 0.58333333, 0.0593963 ,
        0.38242024],
       [0.32066344, 0.        , 0.        , ..., 0.41666667, 0.        ,
        0.35671019],
       [0.20248791, 0.41937046, 0.        , ..., 0.41666667, 0.04089581,
        0.40678422],
       ...,
       [0.2950933 , 0.55786925, 0.        , ..., 0.66666667, 0.05842259,
        0.46708844],
       [0.25708362, 0.        , 0.        , ..., 0.25      , 0.1090555 ,
        0.29263696],
       [0.31859019, 0.        , 0.        , ..., 0.33333333, 0.06621227,
        0.33948041]])
```

Train Model: 1. Linear Regression

Linear Regression **AFTER normalization feature scaling:**

2 Measure Metrics --- mean_squared_error & RMSE

min max scaling

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
scaler.fit(X_lr)
X_lr_sc = scaler.transform(X_lr)
```

training lr model

```
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

x_train, x_test, y_train, y_test = train_test_split(X_lr_sc, y_lr, random_state = 42, test_size = 0.3)
reg = LinearRegression().fit(x_train, y_train)
predictions = reg.predict(x_test)
mean_squared_error(y_test, predictions)
```

```
1.8003338020965856e+16
```

```
kf = KFold(n_folds, shuffle=True, random_state=42).get_n_splits(X_lr_sc)
score= np.sqrt(-cross_val_score(reg, X_lr_sc, y_lr, scoring="neg_mean_squared_error", cv = kf))
score1 = np.delete(score, 6)
print("\nLR score: {:.4f} ({:.4f})\n".format(score1.mean(), score1.std()))
```

```
LR score: 0.1249 (0.0113)
```

As we can see, mean_squared_error DOWN but the score didn’t change much. So we continued to try other models.

Train Model: 2. Lasso Regression

```
Lasso score: 0.1146 (0.0149)

Lasso score: 0.1128 (0.0150)

Lasso score: 0.1118 (0.0152)

Lasso score: 0.1111 (0.0153)

Lasso score: 0.1106 (0.0153)

Lasso score: 0.1104 (0.0152)

Lasso score: 0.1103 (0.0151)

Lasso score: 0.1103 (0.0151)

Lasso score: 0.1104 (0.0150)
```

```
for i in (0.0001,0.00015,0.0002,0.00025,0.0003,0.00035,0.0004,0.00045,0.0005):
    lasso = make_pipeline(RobustScaler(), Lasso(alpha =i, random_state=1))
    score = rmsle_cv(lasso)
    print("\nLasso score: {:.4f} ({:.4f})\n".format(score.mean(), score.std()))
```

```
lasso = make_pipeline(RobustScaler(), Lasso(alpha =0.00045, random_state=1))
score = rmsle_cv(lasso)
print("\nLasso score: {:.4f} ({:.4f})\n".format(score.mean(), score.std()))
```

```
Lasso score: 0.1103 (0.0151)
```

Our second model is lasso regression. We wrote a loop to check the scores of different parameters. The best score is 0.1103

Train Model: 3. Elastic Net Regression

```
alpha:  0.0005
l1_ratio:  0.01
\ENet score: 0.1172 (0.0142)

alpha:  0.0005
l1_ratio:  0.1
\ENet score: 0.1155 (0.0146)

alpha:  0.0005
l1_ratio:  0.5
\ENet score: 0.1110 (0.0152)

alpha:  0.0005
l1_ratio:  0.9
\ENet score: 0.1103 (0.0150)

alpha:  0.0005
l1_ratio:  0.99
\ENet score: 0.1103 (0.0150)
```

```python
for i in (0.0001,0.00015,0.0002,0.00025,0.0003,0.00035,0.0004,0.00045,0.0005):
    for j in (.01, .1, .5, .9, .99):
        lENet = make_pipeline(RobustScaler(), ElasticNet(alpha=i, l1_ratio=j, random_state=3))
        score = rmsle_cv(lENet)
        print('alpha: ', i, '\nl1_ratio: ', j)
        print("\ENet score: {:.4f} ({:.4f})\n".format(score.mean(), score.std()))
```

```python
ENet = make_pipeline(RobustScaler(), ElasticNet(alpha=0.0005, l1_ratio=.9, random_state=3))
score = rmsle_cv(ENet)
print("\ENet score: {:.4f} ({:.4f})\n".format(score.mean(), score.std()))
```

```
\ENet score: 0.1103 (0.0150)
```

Our third model is Elastic Net Regression. The best score is 0.1103

Train Model: 4. Gradient Boosting Regression

```
for i in (1000, 1500, 2000, 2500, 3000, 3500, 4000, 4500):
    GBoost = GradientBoostingRegressor(n_estimators=i , learning_rate=0.05,
                                       max_depth=3, max_features='sqrt',
                                       min_samples_leaf=15, min_samples_split=10,
                                       loss='huber', random_state =5)
    score = rmsle_cv(GBoost)
    print("Gradient Boosting score: {:.4f} ({:.4f})\n".format(score.mean(), score.std()))
```

```
Gradient Boosting score: 0.1105 (0.0148)

Gradient Boosting score: 0.1105 (0.0148)

Gradient Boosting score: 0.1106 (0.0150)

Gradient Boosting score: 0.1109 (0.0149)

Gradient Boosting score: 0.1113 (0.0149)

Gradient Boosting score: 0.1115 (0.0149)

Gradient Boosting score: 0.1118 (0.0149)

Gradient Boosting score: 0.1121 (0.0149)
```

Our forth model is Gradient Boosting Regression.
The best score is 0.1105

Train Model: 5. Ridge Regression

```
for i in (11,11.5,12,12.5,13,13.5,14,14.5,15):
    ridge = Ridge(alpha=i, copy_X=True, fit_intercept=True, max_iter=None,
                  normalize=False, random_state=None, solver='auto', tol=0.0001)
    score = rmsle_cv(ridge)
    print("ridge score: {:.4f} ({:.4f})\n".format(score.mean(), score.std()))
```

```
ridge score: 0.1127 (0.0138)

ridge score: 0.1126 (0.0137)

ridge score: 0.1126 (0.0137)

ridge score: 0.1126 (0.0137)

ridge score: 0.1126 (0.0137)

ridge score: 0.1126 (0.0137)

ridge score: 0.1126 (0.0137)

ridge score: 0.1126 (0.0137)

ridge score: 0.1126 (0.0137)
```

Our fifth model is ridge Regression. The best score is 0.1126

Stack Model

```
class AveragingModels(BaseEstimator, RegressorMixin, TransformerMixin):
    def __init__(self, models):
        self.models = models

    # we define clones of the original models to fit the data in
    def fit(self, X, y):
        self.models_ = [clone(x) for x in self.models]

        # Train cloned base models
        for model in self.models_:
            model.fit(X, y)

        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 (predictions[:,0]*predictions[:,1])   ** (1/2)
```

Finally, considering the performance of these models, we choose Ridge Regression and Gradient Boosting Regression with the best parameters to build the stacked model.
To be mentioned, there are 2 ways to stack the models. The first one is adding two results and divide by two.
The second one is multiplying the two results and then square root.
And the second one performed well in our result.

```
averaged_models = AveragingModels(models = (GBoost, ridge))
MODEL = averaged_models.fit(train, Price)
Price_pred =averaged_models.predict(test.values)
score = rmsle_cv(averaged_models)
print(" Averaged base models score: {:.4f} ({:.4f})\n".format(score.mean(), score.std()))
```

Averaged base models score: 0.1068 (0.0150)

Predict test data and output .csv file

Remember converting normalization form back after prediction!

```
final_predictions = np.exp(Price_pred)-1
print(final_predictions)
```

```
test = pd.read_csv('test.csv')
test_id = test['Id']
submission = pd.DataFrame()
submission['Id'] = test_id
submission['SalePrice'] = final_predictions
```

Finally, as mentioned before, we convert the saleprice to original one. Then write the result to csv file, upload, finish.

Thanks!