AML2019

Challenge 1

House Pricing Prediction

22th March 2019

Corentin RAFFLIN

Intro

Defining the overall price of a house can be challenging. A lot of factors influence more or less the price, one can think of several factors : size of the house, presence of a pool, neighboorhood... We will try to predict the price of a house by detecting the most important features and trying several predictive models.

To realize this kernel, we followed guidelines and took part of codes from different kernels in kaggle :

- **For the visualisation :** https://www.kaggle.com/jowin92/house-price-prediction
- **Processing and model by massquantity :** https://www.kaggle.com/massquantity/all-you-need-is-pca-lb-0-11421-top-4
- **Processing and visualisation by janiobachmann :** https://www.kaggle.com/janiobachmann/predicting-house-prices-regression-techniques
- **Other things :** https://www.kaggle.com/juliencs/a-study-on-regression-applied-to-the-ames-dataset

We tried to question choices, explain why some things are necessary, what alternatives we could have used and also optimize parameters/models the best we could.

In [ ]:

```
#Diverses libraries
%matplotlib inline
import os
import sys
import re
import random
from time import time

# Data and plotting imports
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import seaborn as sns

from scipy import stats

#statistical libraries
from sklearn.base import BaseEstimator, TransformerMixin, RegressorMixin, clone
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import RobustScaler
from sklearn.metrics import mean_squared_error, r2_score
```

```
from sklearn.pipeline import Pipeline, make_pipeline
from scipy.stats import skew, norm, probplot, normaltest
from sklearn.decomposition import PCA
from sklearn.preprocessing import Imputer
from sklearn.model_selection import cross_val_score, GridSearchCV, train_test_split, KFold
from sklearn.linear_model import LinearRegression, Ridge, Lasso, ElasticNet, SGDRegressor,
BayesianRidge
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor, ExtraTreesReg
ressor
from sklearn.svm import SVR, LinearSVR
from sklearn.kernel_ridge import KernelRidge
```

1. Data Exploration

Importing and getting to know the data
First let us import the data and get some general knowledge about it.

In [ ]:

```
# training data
trainDF = pd.read_csv('challenge_data/train.csv', sep=',', header=0)
# test data
testDF = pd.read_csv('challenge_data/test.csv', sep=',', header=0)
```

In [ ]:

```
#Printing info on the training data
trainDF.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1200 entries, 0 to 1199
Data columns (total 81 columns):
Id               1200 non-null int64
MSSubClass       1200 non-null int64
MSZoning         1200 non-null object
LotFrontage      990 non-null float64
LotArea          1200 non-null int64
Street           1200 non-null object
Alley            75 non-null object
LotShape         1200 non-null object
LandContour      1200 non-null object
Utilities        1200 non-null object
LotConfig        1200 non-null object
LandSlope        1200 non-null object
Neighborhood     1200 non-null object
Condition1       1200 non-null object
Condition2       1200 non-null object
BldgType         1200 non-null object
HouseStyle       1200 non-null object
OverallQual      1200 non-null int64
OverallCond      1200 non-null int64
YearBuilt        1200 non-null int64
YearRemodAdd     1200 non-null int64
RoofStyle        1200 non-null object
RoofMatl         1200 non-null object
Exterior1st      1200 non-null object
Exterior2nd      1200 non-null object
MasVnrType       1194 non-null object
MasVnrArea       1194 non-null float64
ExterQual        1200 non-null object
ExterCond        1200 non-null object
Foundation       1200 non-null object
BsmtQual         1168 non-null object
```

```
BsmtCond         1168 non-null object
BsmtExposure     1167 non-null object
BsmtFinType1     1168 non-null object
BsmtFinSF1       1200 non-null int64
BsmtFinType2     1167 non-null object
BsmtFinSF2       1200 non-null int64
BsmtUnfSF        1200 non-null int64
TotalBsmtSF      1200 non-null int64
Heating          1200 non-null object
HeatingQC        1200 non-null object
CentralAir       1200 non-null object
Electrical       1200 non-null object
1stFlrSF         1200 non-null int64
2ndFlrSF         1200 non-null int64
LowQualFinSF     1200 non-null int64
GrLivArea        1200 non-null int64
BsmtFullBath     1200 non-null int64
BsmtHalfBath     1200 non-null int64
FullBath         1200 non-null int64
HalfBath         1200 non-null int64
BedroomAbvGr     1200 non-null int64
KitchenAbvGr     1200 non-null int64
KitchenQual      1200 non-null object
TotRmsAbvGrd     1200 non-null int64
Functional       1200 non-null object
Fireplaces       1200 non-null int64
FireplaceQu      636 non-null object
GarageType       1133 non-null object
GarageYrBlt      1133 non-null float64
GarageFinish     1133 non-null object
GarageCars       1200 non-null int64
GarageArea       1200 non-null int64
GarageQual       1133 non-null object
GarageCond       1133 non-null object
PavedDrive       1200 non-null object
WoodDeckSF       1200 non-null int64
OpenPorchSF      1200 non-null int64
EnclosedPorch    1200 non-null int64
3SsnPorch        1200 non-null int64
ScreenPorch      1200 non-null int64
PoolArea         1200 non-null int64
PoolQC           4 non-null object
Fence            227 non-null object
MiscFeature      47 non-null object
MiscVal          1200 non-null int64
MoSold           1200 non-null int64
YrSold           1200 non-null int64
SaleType         1200 non-null object
SaleCondition    1200 non-null object
SalePrice        1200 non-null int64
dtypes: float64(3), int64(35), object(43)
memory usage: 759.5+ KB
```

In [ ]:

```
#Stocking the IDs
train_id = trainDF['Id']
test_id = testDF['Id']
#Dropping the IDs
trainDF = trainDF.drop('Id', axis = 1)
testDF = testDF.drop('Id', axis = 1)

#Number of features in the training set
print("After removing ID in the data set, there are ", trainDF.columns.size, " columns in th
e training set.")

#Quantitative features
```

```
quantitative = [f for f in trainDF.columns if trainDF.dtypes[f] != 'object']
print("\nquantitative :", quantitative)
print("There are ", len(quantitative), "quantitative columns")
#Removing the output we are looking for
quantitative.remove('SalePrice')

#Qualitative features
qualitative = [f for f in trainDF.columns if trainDF.dtypes[f] == 'object']
print("\nqualitative :", qualitative)
print("There are ", len(qualitative), "qualitative columns")
```

```
After removing ID in the data set, there are  80  columns in the training set.

quantitative : ['MSSubClass', 'LotFrontage', 'LotArea', 'OverallQual', 'OverallCond', 'YearB
uilt', 'YearRemodAdd', 'MasVnrArea', 'BsmtFinSF1', 'BsmtFinSF2', 'BsmtUnfSF', 'TotalBsmtSF',
'1stFlrSF', '2ndFlrSF', 'LowQualFinSF', 'GrLivArea', 'BsmtFullBath', 'BsmtHalfBath', 'FullBa
th', 'HalfBath', 'BedroomAbvGr', 'KitchenAbvGr', 'TotRmsAbvGrd', 'Fireplaces', 'GarageYrBlt'
, 'GarageCars', 'GarageArea', 'WoodDeckSF', 'OpenPorchSF', 'EnclosedPorch', '3SsnPorch', 'Sc
reenPorch', 'PoolArea', 'MiscVal', 'MoSold', 'YrSold', 'SalePrice']
There are  37 quantitative columns

qualitative : ['MSZoning', 'Street', 'Alley', 'LotShape', 'LandContour', 'Utilities', 'LotCo
nfig', 'LandSlope', 'Neighborhood', 'Condition1', 'Condition2', 'BldgType', 'HouseStyle', 'R
oofStyle', 'RoofMatl', 'Exterior1st', 'Exterior2nd', 'MasVnrType', 'ExterQual', 'ExterCond',
'Foundation', 'BsmtQual', 'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFinType2', 'Heati
ng', 'HeatingQC', 'CentralAir', 'Electrical', 'KitchenQual', 'Functional', 'FireplaceQu', 'G
arageType', 'GarageFinish', 'GarageQual', 'GarageCond', 'PavedDrive', 'PoolQC', 'Fence', 'Mi
scFeature', 'SaleType', 'SaleCondition']
There are  43 qualitative columns
```

Knowledge about the output : SalePrice

Let us get knowledge about the 'SalePrice' which is the output we want to get from our test set.

In [ ]:

```
#Describing the output : mean, quantiles and so on
trainDF['SalePrice'].describe().to_frame()
```

Out[ ]:

	SalePrice
count	1200.000000
mean	181414.628333
std	81070.908544
min	34900.000000
25%	129900.000000
50%	163700.000000
75%	214000.000000
max	755000.000000

From the previous table, everything seems normal, for one the minimum is not negative. Yet, the standard deviation is pretty high compared to the mean, meaning the data deviates a lot from the mean (many outliers). Additionaly we noticed that 50% of SalePrice are less or equal to 163,700. As maximum price is 755.000, we can expect skeweness in SalePrice.

Some (personal) notes to understand the analysis of data

> **Residual :** In statistical models, a residual is the difference between the observed value and the mean value that the model predicts for that observation. Residual values are especially useful in regression and ANOVA procedures because they indicate the extent to which a model accounts for the variation in the observed data.

https://statisticsbyjim.com/glossary/residuals/

> **Heteroscedasticity :** Heteroscedasticity means unequal scatter. In regression analysis, we talk about heteroscedasticity in the context of the residuals or error term. Specifically, heteroscedasticity is a systematic change in the spread of the residuals over the range of measured values. Heteroscedasticity is a problem because ordinary least squares (OLS) regression assumes that all residuals are drawn from a population that has a constant variance (homoscedasticity).

https://statisticsbyjim.com/regression/heteroscedasticity-regression/

> **Skewedness :** (Janiobachman kernel)
>
> - A skewness of zero or near zero indicates a symmetric distribution.
> - A negative value for the skewness indicate a left skewness (tail to the left)
> - A positive value for te skewness indicate a right skewness (tail to the right)
>
> **Kurtosis :** (Janiobachman kernel)
>
> - Kurtosis is a measure of how extreme observations are in a dataset.
> - The greater the kurtosis coefficient , the more peaked the distribution around the mean is.
> - Greater coefficient also means fatter tails, which means there is an increase in tail risk (extreme results)

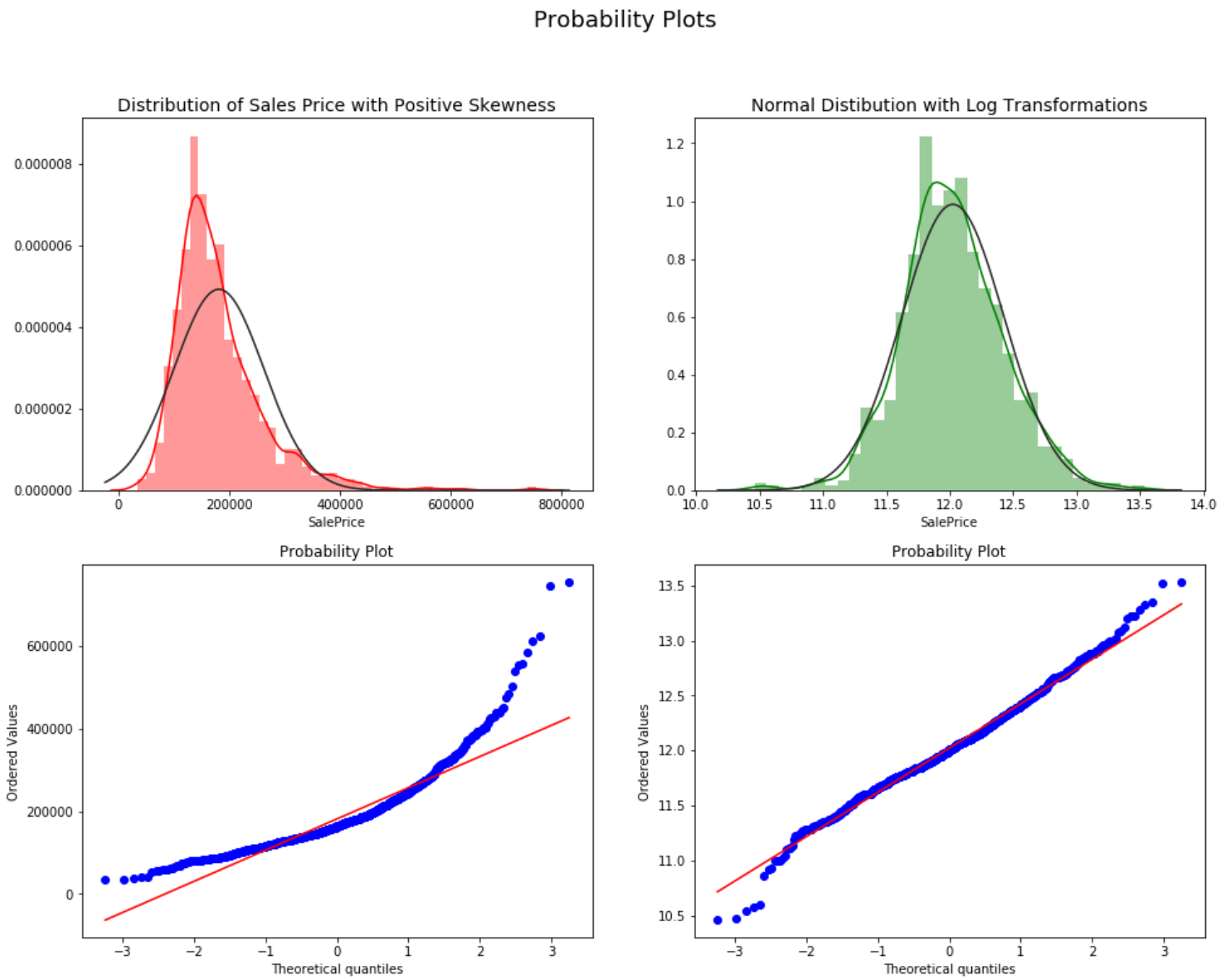
> **Applying a log transformation**
>
> - To remove positive skewness, to have a symmetry
> - To make it easier to interpret patterns of our data
> - For possible statistical analysis that require the data to be normalized
> - To remove outliers : a log transformation could reduce the influence of those observations
> - To eliminate heteroscedasticity (when the variance of the regression residuals are increasing with the regression predictions)
> - To make residuals normal
> - To transform a non-linear model into a linear model

https://www.quora.com/Why-do-we-log-variables-in-regression-model

In [ ]:

```
#Computing the log of the saleprice
salePrice_log = np.log(trainDF['SalePrice'])

fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows=2, ncols=2, figsize=(16,12))
plt.suptitle('Probability Plots', fontsize=18)

#Plotting the distribution of SalePrice (before log transformation) and comparing it with a
normal distribution
ax1 = sns.distplot(trainDF['SalePrice'], color="r", ax=ax1, fit=norm)
```

```
ax1.set_title("Distribution of Sales Price with Positive Skewness", fontsize=14)

#Plotting the distribution of log of salePrice and comparing it with a normal distribution
ax2 = sns.distplot(salePrice_log, color="g", ax=ax2, fit=norm)
ax2.set_title("Normal Distibution with Log Transformations", fontsize=14)

#Plotting the corresponding probability plot
ax3 = probplot(trainDF['SalePrice'], plot=ax3)
ax4 = probplot(salePrice_log, plot=ax4)

plt.show()
```



Before the log transformation, we notice a positive skewness. There is a higher frequency of occurence to the left of the distribution plot leading to more exceptions (outliers) to the right at the higher cost. Nevertheless, there is a way to transform this histogram into a normal distributions by using log transformations. And indeed, after applying the log transform, there is less skewness.

In [ ]:

```
print('Skewness for Normal D.: %f'% trainDF['SalePrice'].skew())
print('Skewness for Log D.: %f'% salePrice_log.skew())
print('Kurtosis for Normal D.: %f' % trainDF['SalePrice'].kurt())
print('Kurtosis for Log D.: %f' % salePrice_log.kurt())
```

```
Skewness for Normal D.: 1.967215
Skewness for Log D.: 0.132714
Kurtosis for Normal D.: 7.033907
```

```
Kurtosis for Log D.: 0.888850
```

Non-meaningful features

Let us look at the most meaningful information to know which plots we should look at in priority. We will still print plots on most features to notice anything suspiscious.

In [ ]:

```
#Printing features that are composed of only few values
countDF = pd.DataFrame()
for column in testDF.columns:
    s = pd.Series([trainDF[column].unique().size, testDF[column].unique().size ])
    countDF[column] = s

countDF = countDF.rename(index={0:"In train", 1:"In test"}).T.sort_values(by="In train")
display(countDF.head())
```

	In train	In test
Street	2	1
CentralAir	2	2
Utilities	2	1
Alley	3	3
HalfBath	3	3

In [ ]:

```
#Printing these values for some of these features, group by elements with non-missing (see l
ater)
display(trainDF.groupby('Utilities', as_index=False).SalePrice.count())
display(testDF.groupby('Utilities', as_index=False).YearBuilt.count())
display(trainDF.groupby('Street', as_index=False).SalePrice.count())
display(testDF.groupby('Street', as_index=False).YearBuilt.count())
```

	Utilities	SalePrice
0	AllPub	1199
1	NoSeWa	1

	Utilities	YearBuilt
0	AllPub	260

	Street	SalePrice
0	Grvl	6
1	Pave	1194

	Street	YearBuilt
0	Pave	260

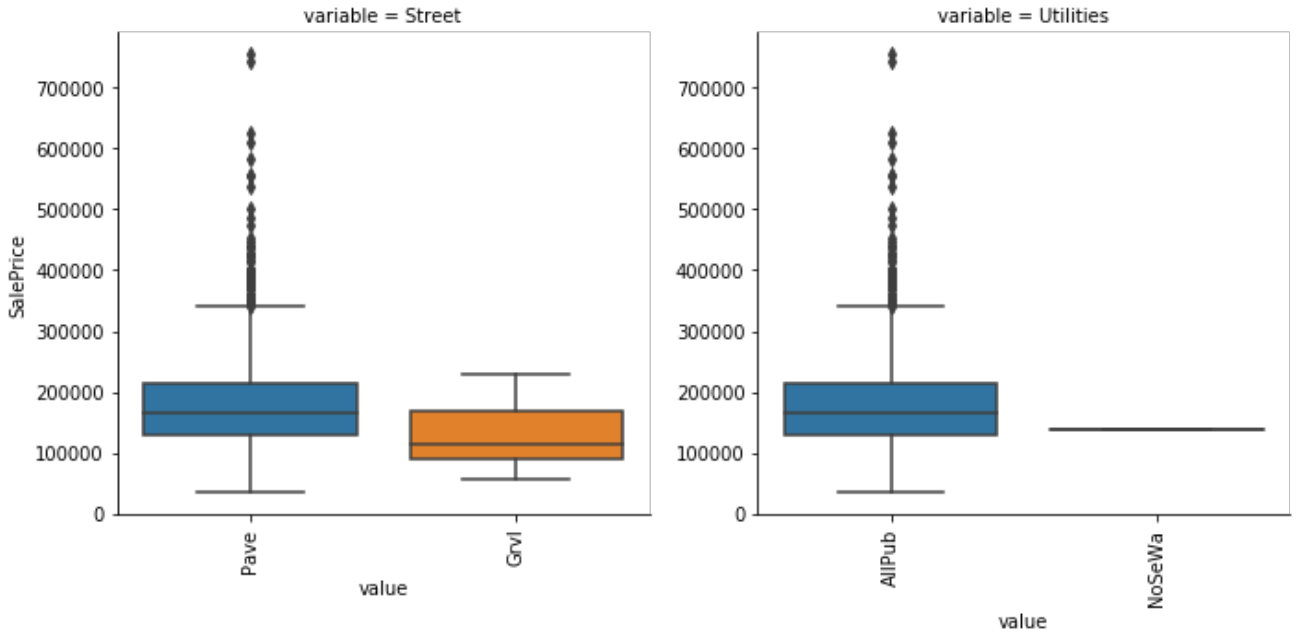
In [ ]:

```
#Defining box plot useful to plot categorical features together
def boxplot(x, y, **kwargs):
    sns.boxplot(x=x, y=y)
```

```
    x=plt.xticks(rotation=90)
```

In [ ]:

```
#Plotting repartition of values of Street and Utilities in regard of SalePrice
f=  pd.melt(trainDF, id_vars=['SalePrice'], value_vars=['Street', 'Utilities'])
g = sns.FacetGrid(f, col="variable", col_wrap=2, sharex=False, sharey=False, height=5)
g = g.map(boxplot, "value", "SalePrice")
```



It seems that Street and Utilities will not play a big part to our model as the testing set contains only one identical value, which is also the most common one in the training set. Therefore we decided to remove it without impacting on our model.

In [ ]:

```
#Dropping these two features
trainDF.drop(['Street', 'Utilities'], axis=1, inplace=True)
testDF.drop(['Street', 'Utilities'], axis=1 , inplace=True)

#Also removing them from the previous list so we do not print more info on them
qualitative.remove('Street')
qualitative.remove('Utilities')
```

NB : After trying with and without dropping these two features, dropping them indeed improve the model prediction (less loss).

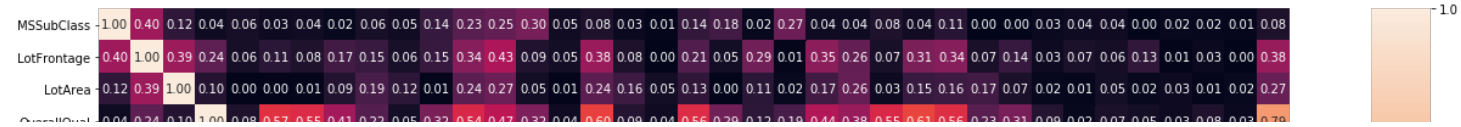
Meaningful features

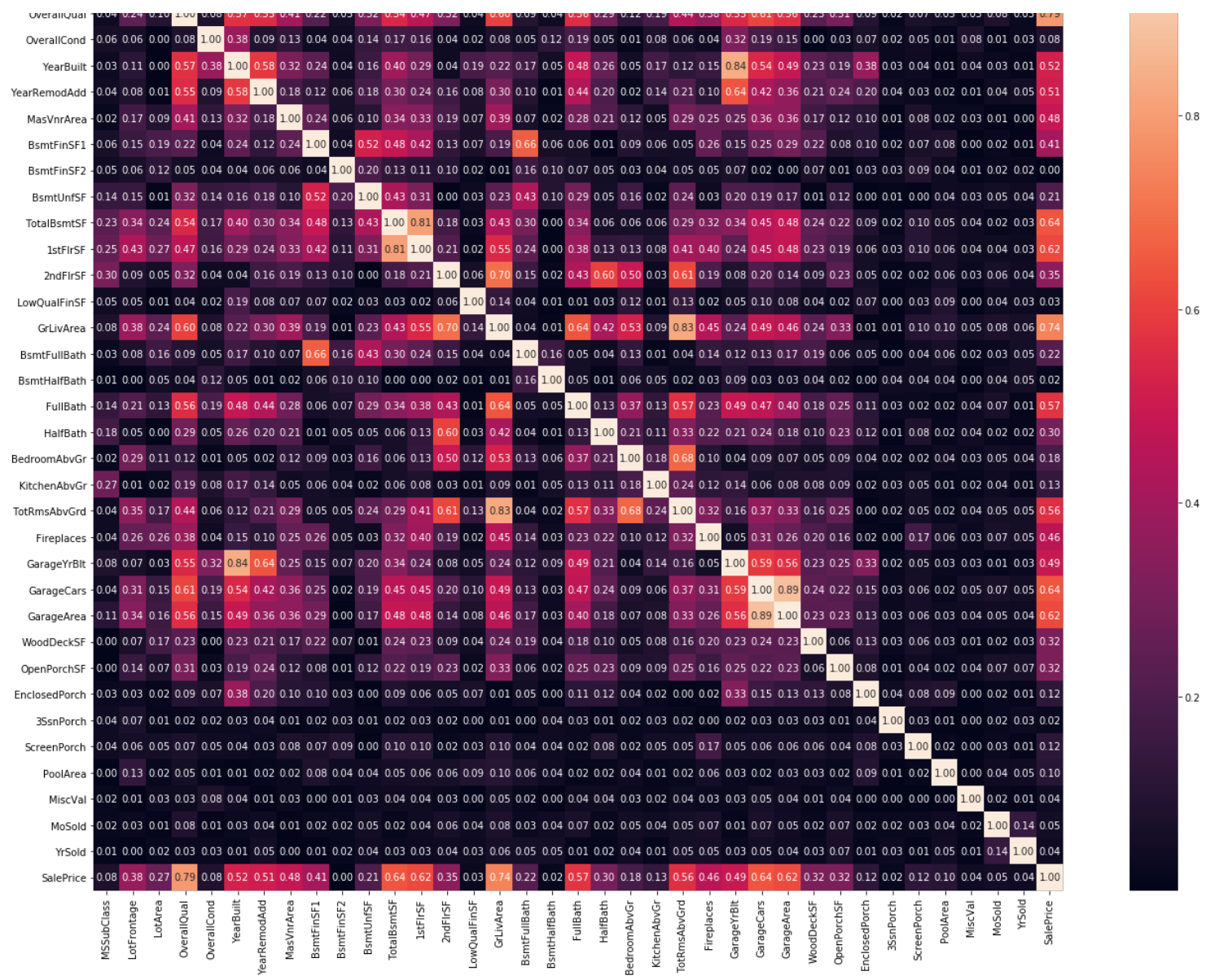
In [ ]:

```
#Correlation matrix
corrmat = abs(trainDF.corr())
f, ax = plt.subplots(figsize=(24, 18))
sns.heatmap(corrmat, vmax=1, square=True, annot=True, annot_kws={'size': 10}, fmt='.2f')
```

Out[ ]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fd7eceba7b8>
```





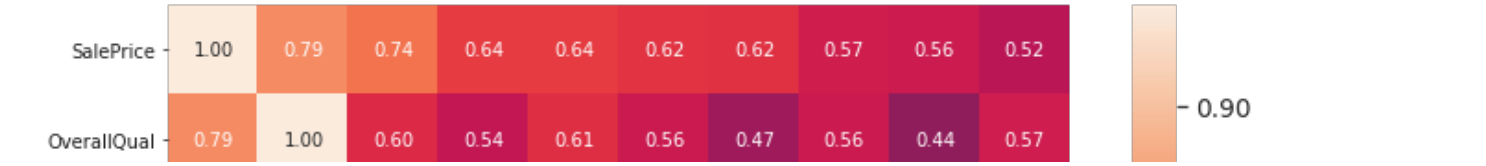
With this correlation matrix, we can notice which features are correlated or not. Some examples of 'strongly' correlated (coefficient greater than 0.7) features :

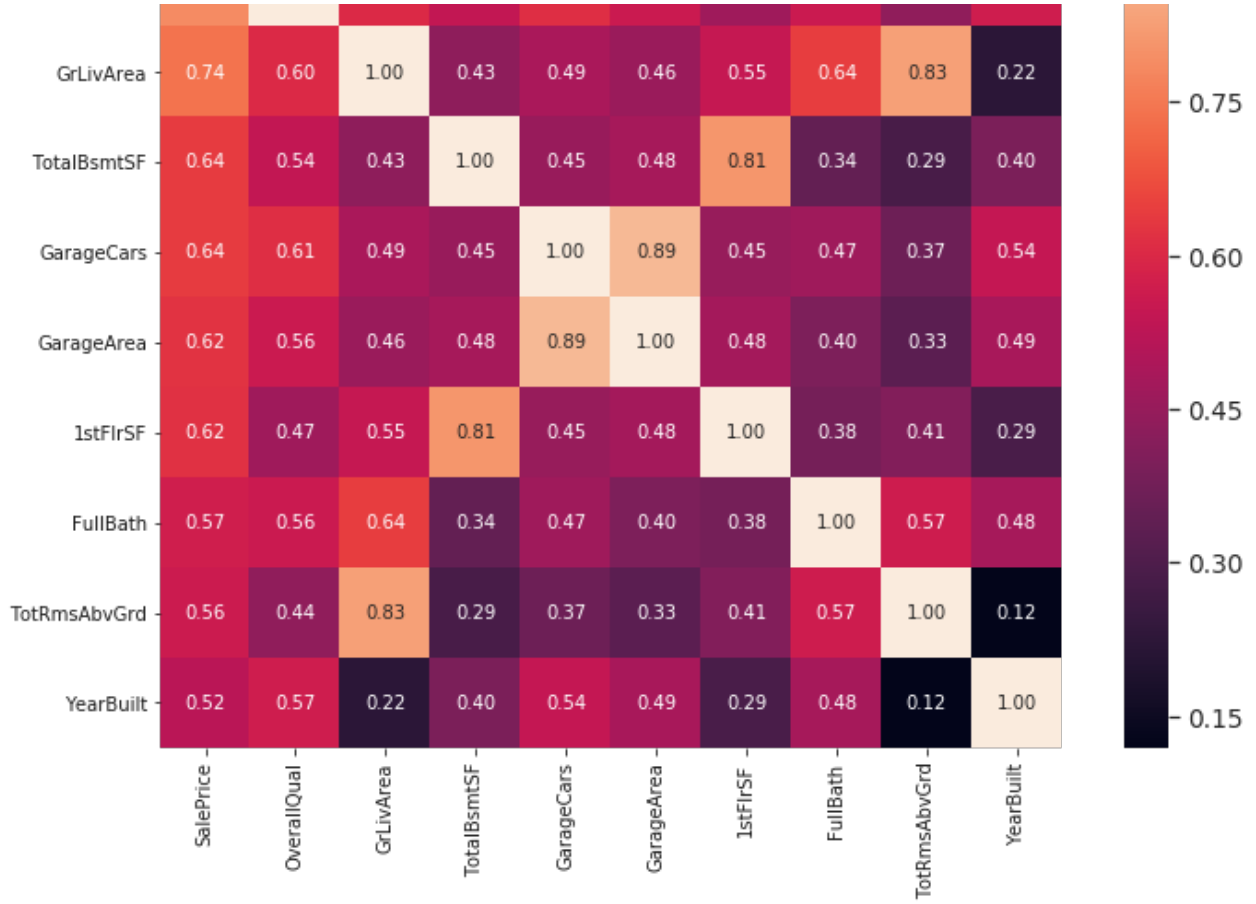
- **'GarageYrBlt' and 'YearBuilt' which makes sense**
- **'TotRmsAbvGrd' and 'GrLiveArea'**
- **'GrLivArea' and '2ndFlrSF'.**

Let us see the correlation matrix for the features most correlated with 'SalePrice' (this is more interesting for us).

In [ ]:

```
#Saleprice correlation matrix
f, ax = plt.subplots(figsize=(12, 9))
cols = corrmat.nlargest(10, 'SalePrice')['SalePrice'].index #getting the 10 most correlated features
cm = np.corrcoef(trainDF[cols].values.T)
sns.set(font_scale=1.25)
hm = sns.heatmap(cm, cbar=True, annot=True, square=True, fmt='.2f', annot_kws={'size': 10}, yticklabels=cols.values, xticklabels=cols.values)
plt.show()
```





Now that we have the most meaningful features for the 'SalePrice' we can try to reduce the dimension further by looking at the correlation between them.

- **'GarageArea' and 'GarageCars' have a strong correlation (0.89) which makes sense. As 'GarageCars' is more correlated with 'SalePrice' we will keep this feature and not 'GarageArea'.**
- **'TotalBsmtSF' and '1stFlrSF' have also a strong correlation (0.81) which again makes sense. With the same argument as before we will keep 'TotalBsmtSF'.**
- **'TotRmsABvGrd' and 'GrLiveArea' have also a sgtrong correlation (0.83). We will keep 'GrLiveArea' with still the same argument.**

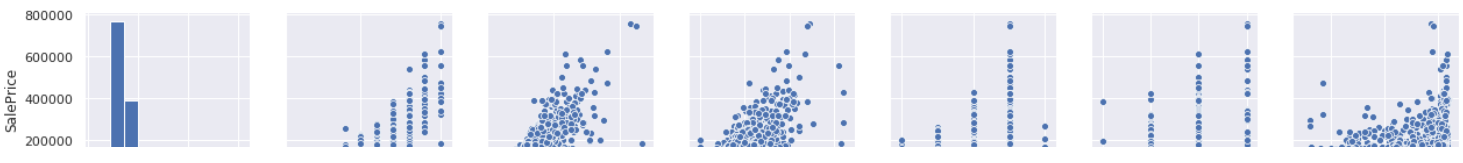
We can therefore say with this evaluation that the most meaningful features to focus on are ['SalePrice', 'OverallQual', 'GrLiveArea', 'TotalBsmtSF', 'GarageCars', 'FullBath', 'YearBuilt']. It seems quite normal that the overall quality is important (yet it is not clear what it represents for a house), the year built also seems for us a good indication of the valor of the house. The feature 'FullBath' however is quite surprising.

Plotting relation (by pair) between these important features

Now that we have an idea of the most meaningful information, we can plot the figures to show the relation between these elements. For categorical features, we will need to make other plots with previous function bbox.

In [ ]:

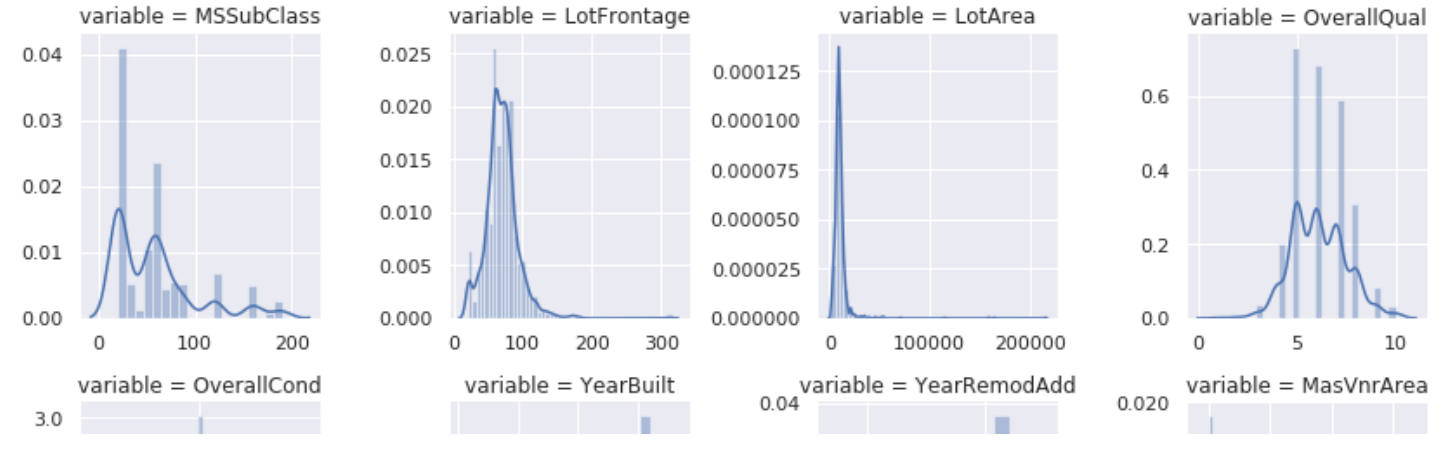
```
sns.set()
cols = ['SalePrice', 'OverallQual', 'GrLivArea', 'TotalBsmtSF', 'GarageCars', 'FullBath', '
YearBuilt']
sns.pairplot(trainDF[cols], height = 2.5)
plt.show();
```

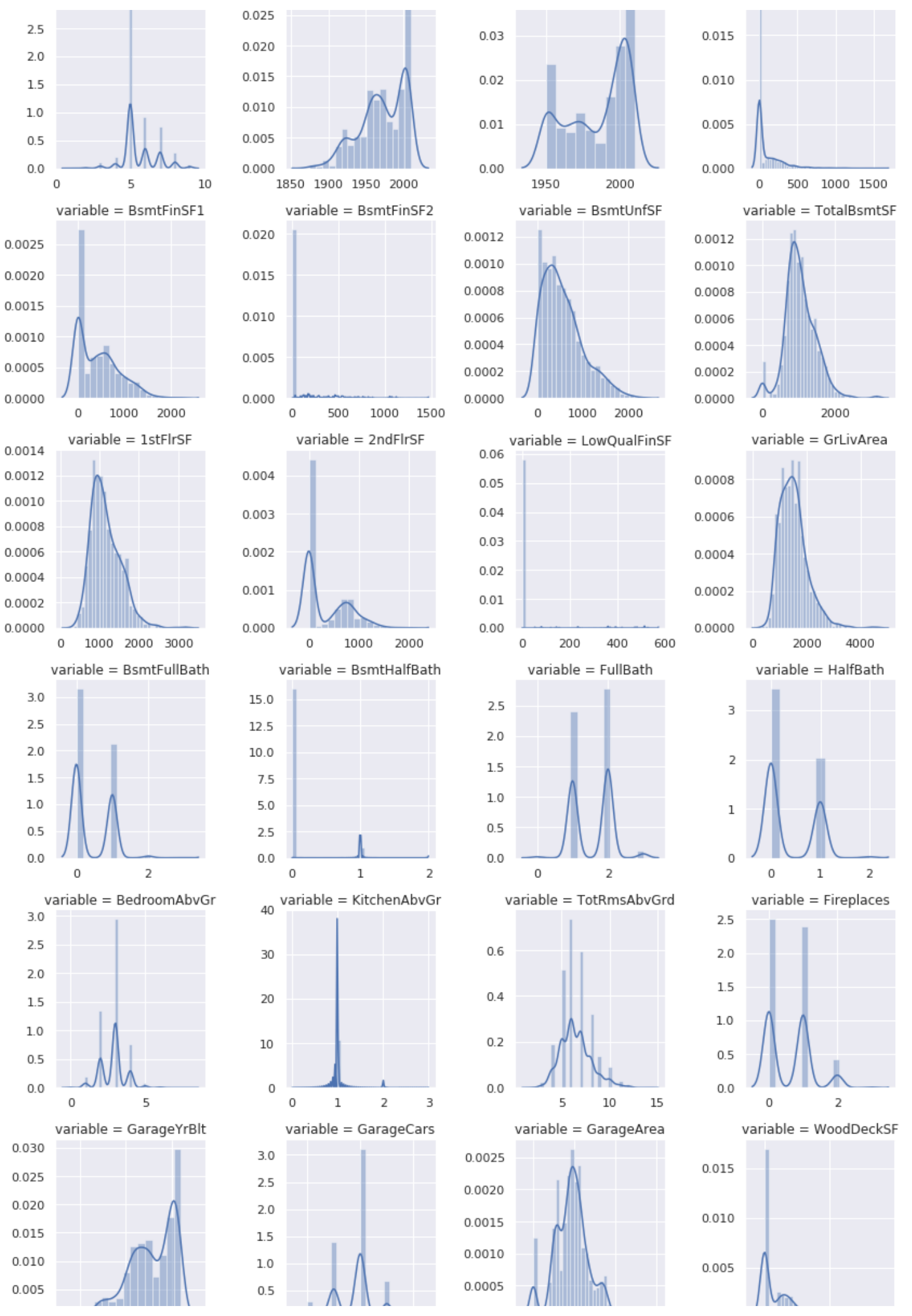


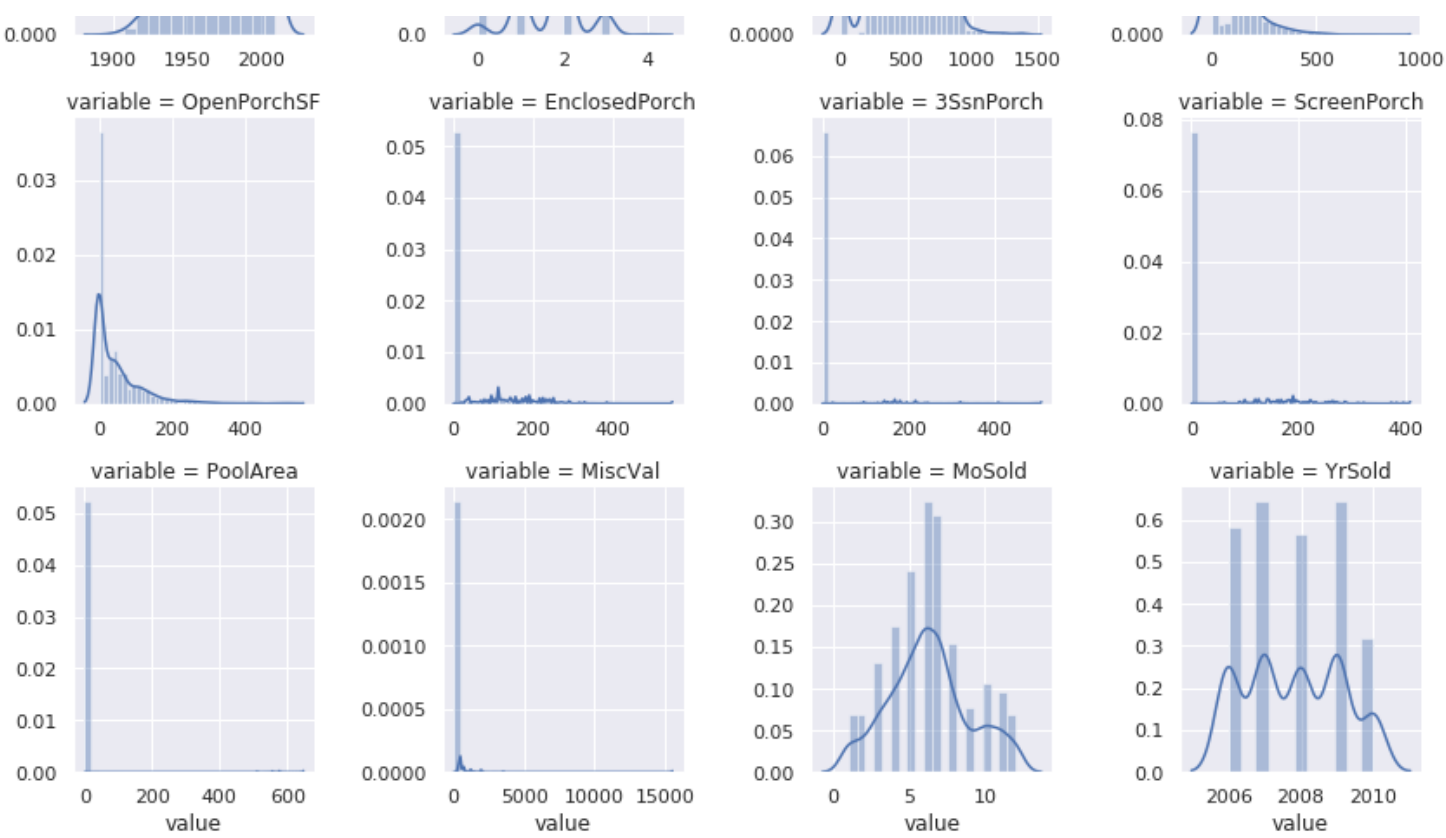
Plotting quantitative features distribution

In [ ]:

```
#plotting univariate distribution of observations of the quantitative features
f = pd.melt(trainDF, value_vars=quantitative)
g = sns.FacetGrid(f, col="variable", col_wrap=4, sharex=False, sharey=False)
g = g.map(sns.distplot, "value")
```




variable = BsmtFinSF1
variable = BsmtFinSF2
variable = BsmtUnfSF
variable = TotalBsmtSF
variable = 1stFlrSF
variable = 2ndFlrSF
variable = LowQualFinSF
variable = GrLivArea
variable = BsmtFullBath
variable = BsmtHalfBath
variable = FullBath
variable = HalfBath
variable = BedroomAbvGr
variable = KitchenAbvGr
variable = TotRmsAbvGrd
variable = Fireplaces
variable = GarageYrBlt
variable = GarageCars
variable = GarageArea
variable = WoodDeckSF

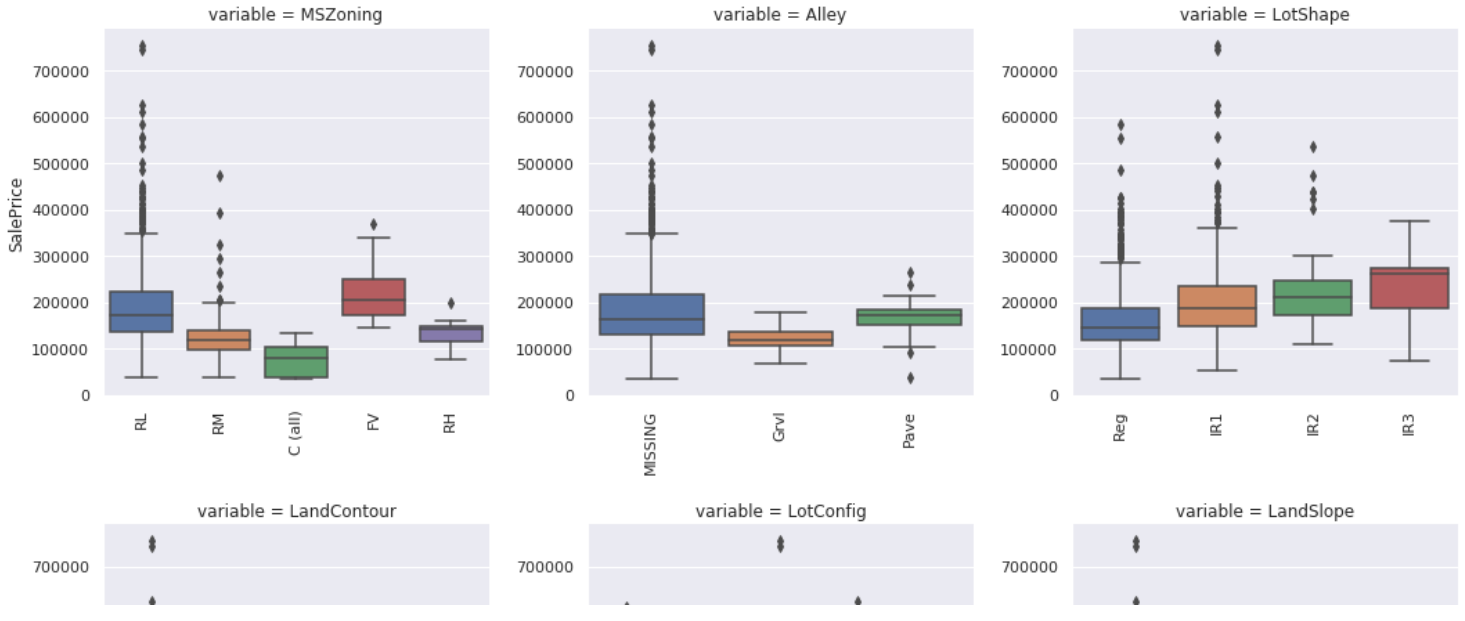


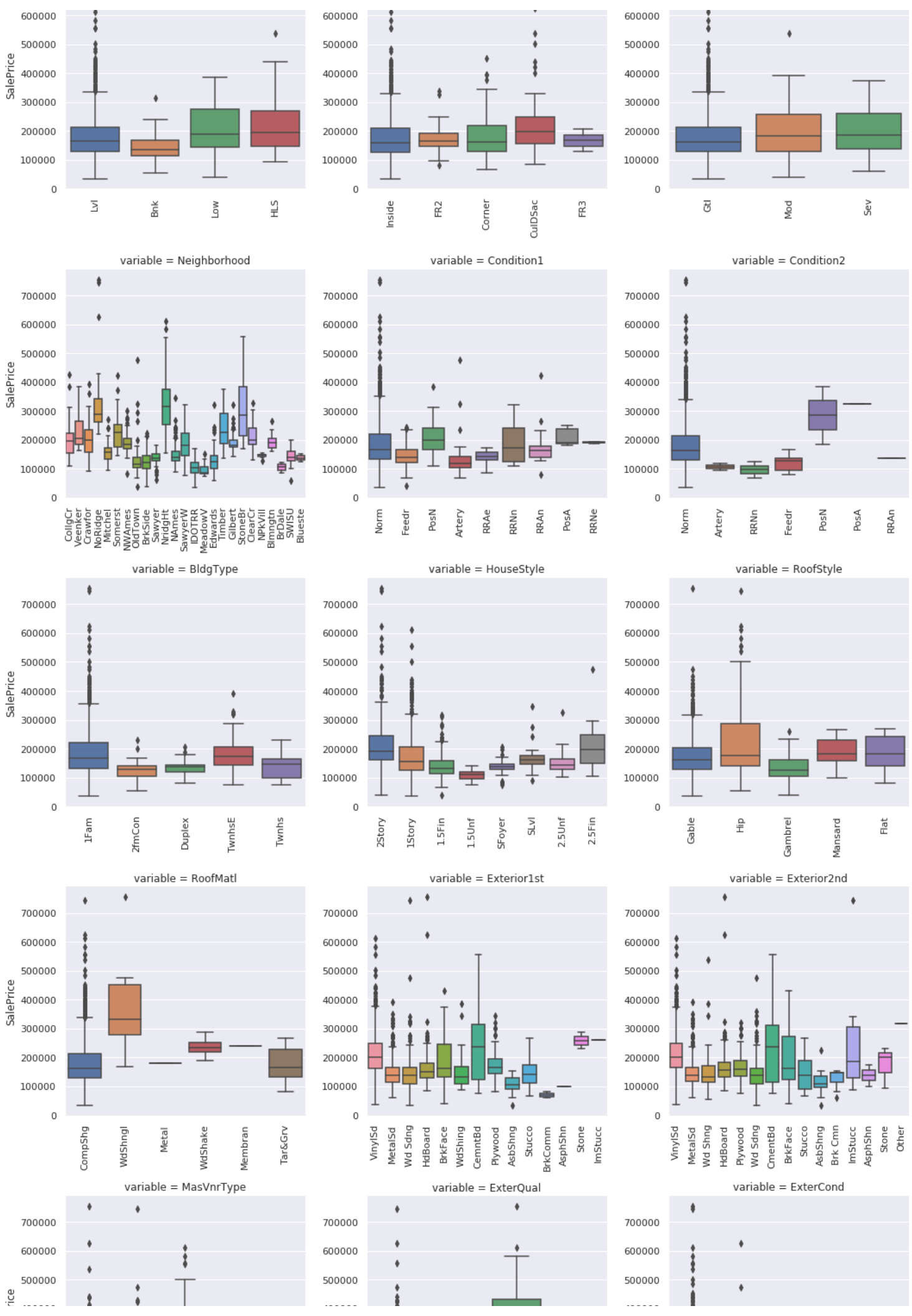
We can notice on the last figure (distribution of 'YrSold') a decrease at the year 2009-2010 that we will notice also later on.

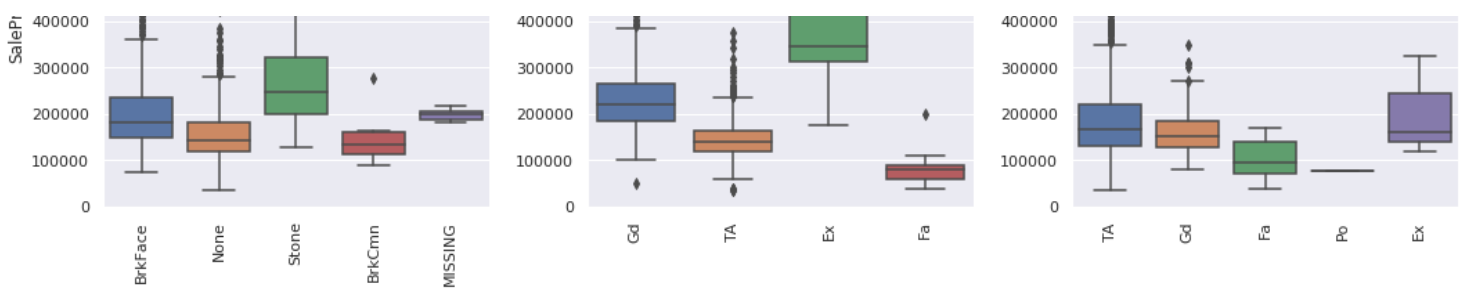
Plotting relation of SalePrice with qualitative features

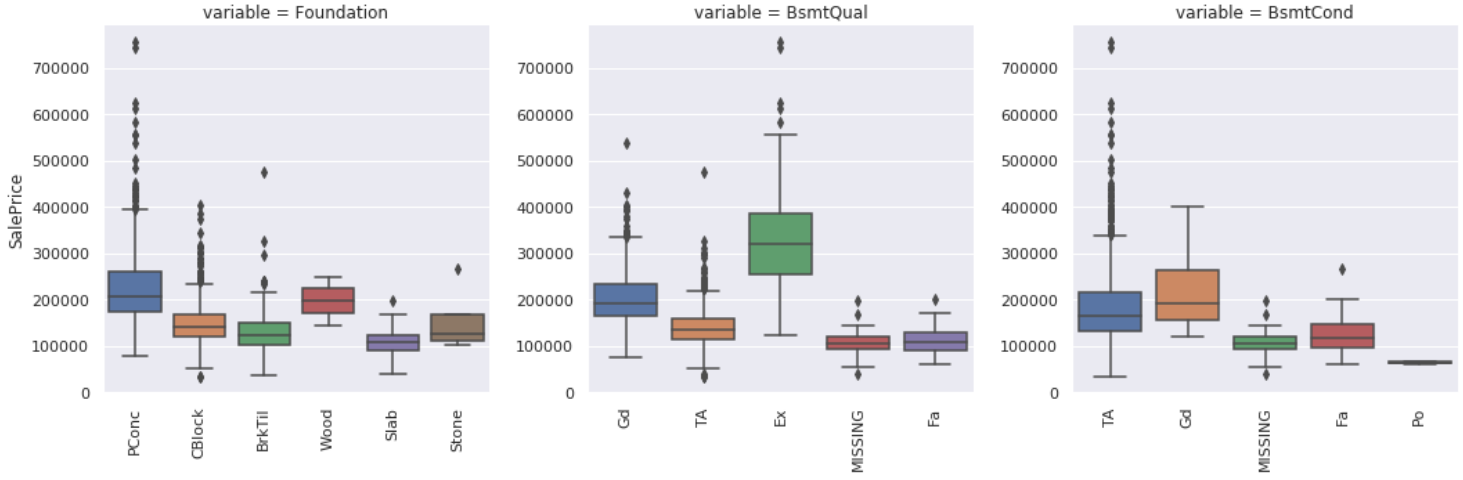
In [ ]:

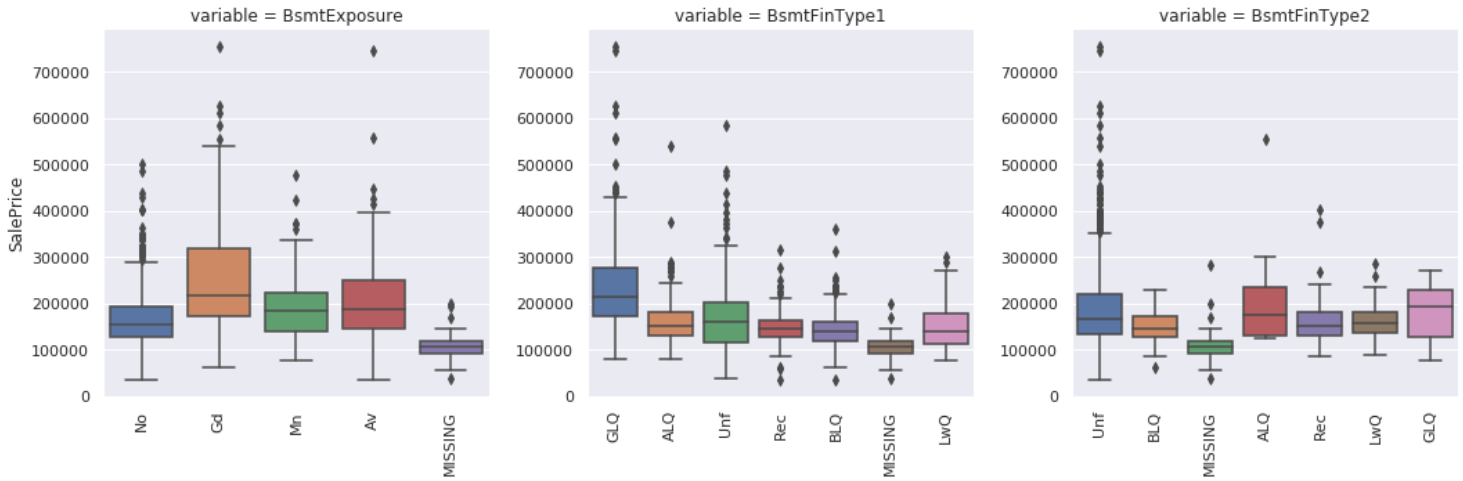
```
for c in qualitative:
    trainDF[c]= trainDF[c].astype('category')
    if trainDF[c].isnull().any():
        trainDF[c] = trainDF[c].cat.add_categories(['MISSING'])
        trainDF[c] = trainDF[c].fillna('MISSING') #we need not to forget to deal with this
later

f=  pd.melt(trainDF, id_vars=['SalePrice'], value_vars=qualitative)
g = sns.FacetGrid(f, col="variable", col_wrap=3, sharex=False, sharey=False, height=5)
g = g.map(boxplot, "value", "SalePrice")
```

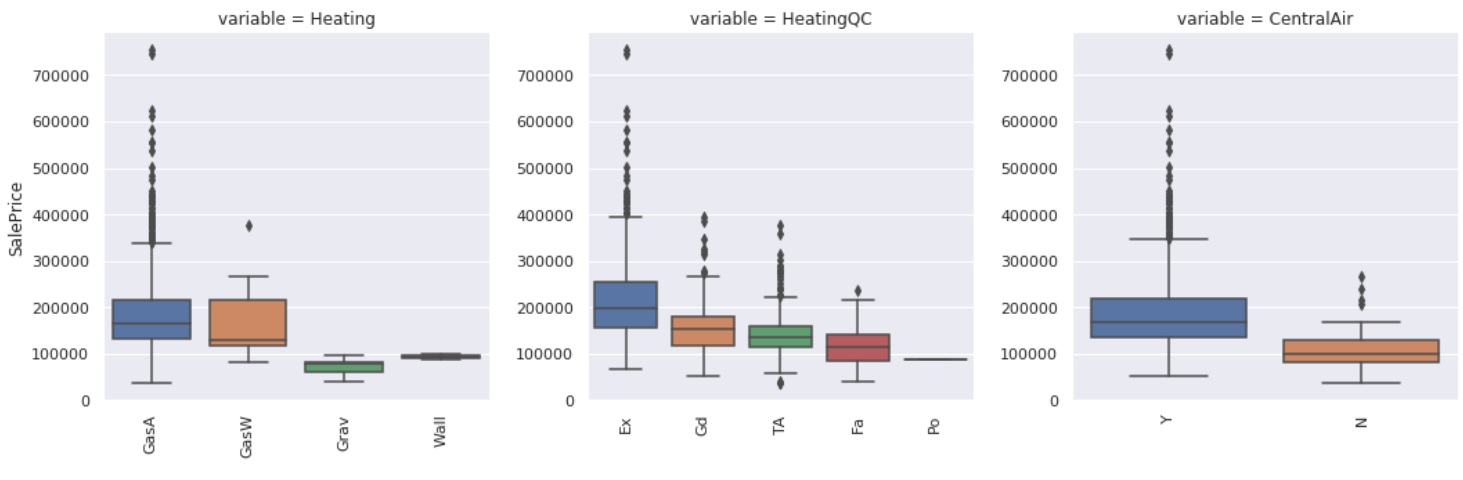


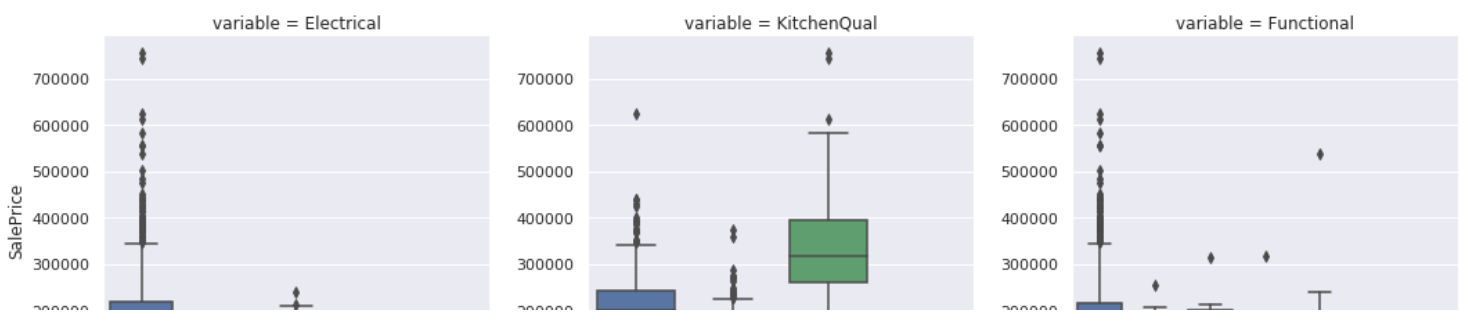

SalePrice
Lvl
Bnk
Low
HLS
Inside
FR2
Corner
CulDSac
FR3
Gtl
Mod
Sev
variable = Neighborhood
CollgCr
Veenker
Crawfor
NoRidge
Mitchel
Somerst
NWAmes
OldTown
BrkSide
Sawyer
NridgHt
NAmes
SawyerW
IDOTRR
MeadowV
Edwards
Timber
Gilbert
StoneBr
ClearCr
NPkVill
Blmngtn
BrDale
SWISU
Blueste
variable = Condition1
Norm
Feedr
PosN
Artery
RRAe
RRNn
RRAn
PosA
RRNe
variable = Condition2
Norm
Artery
RRNn
Feedr
PosN
PosA
RRAn
variable = BldgType
1Fam
2fmCon
Duplex
TwnhsE
Twnhs
variable = HouseStyle
2Story
1Story
1.5Fin
1.5Unf
SFoyer
SLvl
2.5Unf
2.5Fin
variable = RoofStyle
Gable
Hip
Gambrel
Mansard
Flat
variable = RoofMatl
CompShg
WdShngl
Metal
WdShake
Membran
Tar&Grv
variable = Exterior1st
VinylSd
MetalSd
Wd Sdng
HdBoard
BrkFace
WdShing
CemntBd
Plywood
AsbShng
Stucco
BrkComm
AsphShn
Stone
ImStucc
variable = Exterior2nd
VinylSd
MetalSd
Wd Shng
HdBoard
Plywood
Wd Sdng
CmentBd
BrkFace
Stucco
AsbShng
Brk Cmn
ImStucc
AsphShn
Stone
Other
variable = MasVnrType
variable = ExterQual
variable = ExterCond

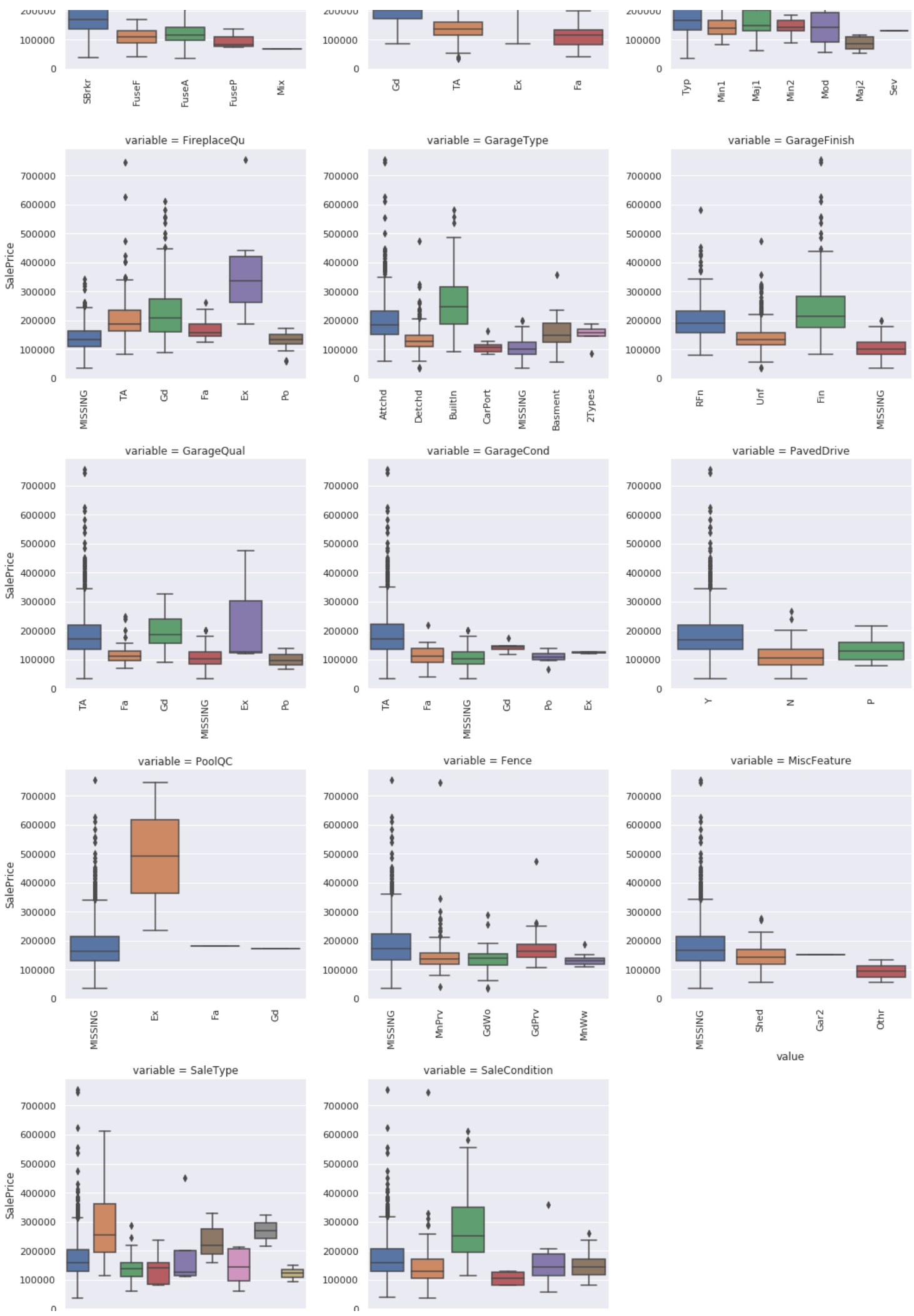

SalePr
400000
300000
200000
100000
0
BrkFace
None
Stone
BrkCmn
MISSING
Gd
TA
Ex
Fa
TA
Gd
Fa
Po
Ex


variable = Foundation
variable = BsmtQual
variable = BsmtCond
SalePrice
700000
600000
500000
400000
300000
200000
100000
0
PConc
CBlock
BrkTil
Wood
Slab
Stone
Gd
TA
Ex
MISSING
Fa
TA
Gd
MISSING
Fa
Po


variable = BsmtExposure
variable = BsmtFinType1
variable = BsmtFinType2
SalePrice
700000
600000
500000
400000
300000
200000
100000
0
No
Gd
Mn
Av
MISSING
GLQ
ALQ
Unf
Rec
BLQ
MISSING
LwQ
Unf
BLQ
MISSING
ALQ
Rec
LwQ
GLQ


variable = Heating
variable = HeatingQC
variable = CentralAir
SalePrice
700000
600000
500000
400000
300000
200000
100000
0
GasA
GasW
Grav
Wall
Ex
Gd
TA
Fa
Po
Y
N


variable = Electrical
variable = KitchenQual
variable = Functional
SalePrice
700000
600000
500000
400000
300000


variable = FireplaceQu
variable = GarageType
variable = GarageFinish
variable = GarageQual
variable = GarageCond
variable = PavedDrive
variable = PoolQC
variable = Fence
variable = MiscFeature
variable = SaleType
variable = SaleCondition
SalePrice
value

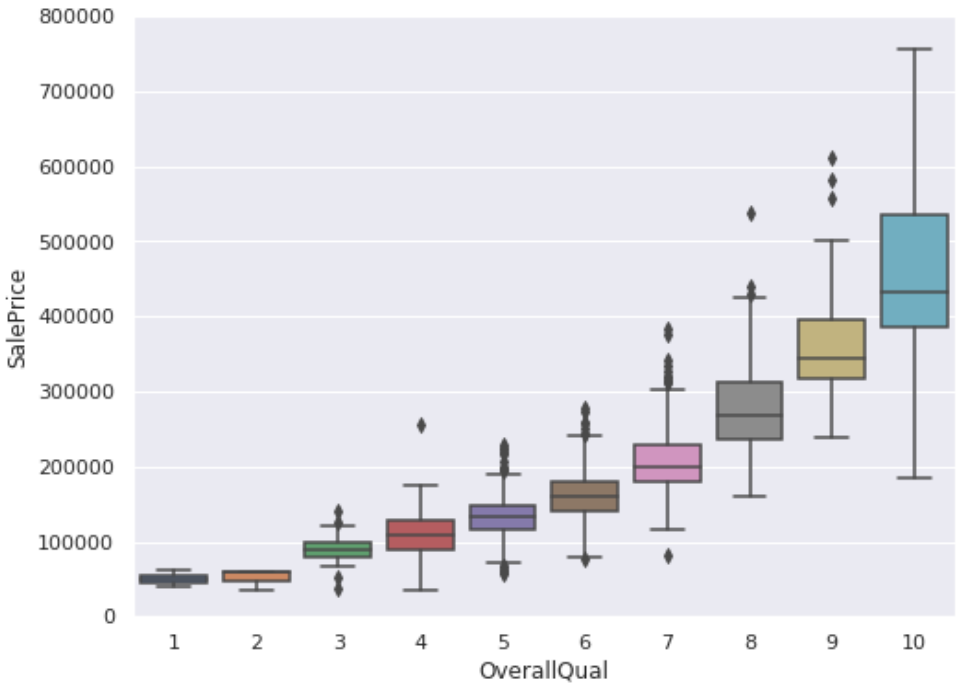
We can already notice that we will have to deal with missing data, for example for the 'Alley' feature. We could be tempted to see what is behind 'CentralAir' and see if we can remove it like we did for 'Street' and 'Utilites', but we already saw that both values appear in the test set so we should keep it.

NB : Test with the final model showed indeed that removing it would decrease performance.

Additionaly we noticed a few plots which are very flat. Example is 'RoofMatl' where for 'Metal' and 'memebran' we can see a line. It is caused by the fact that in our data we have only one observation for each of this type. Whenever in above box plots we see only one line representing the plot, we are suspecting only one observation for the category.

In [ ]:

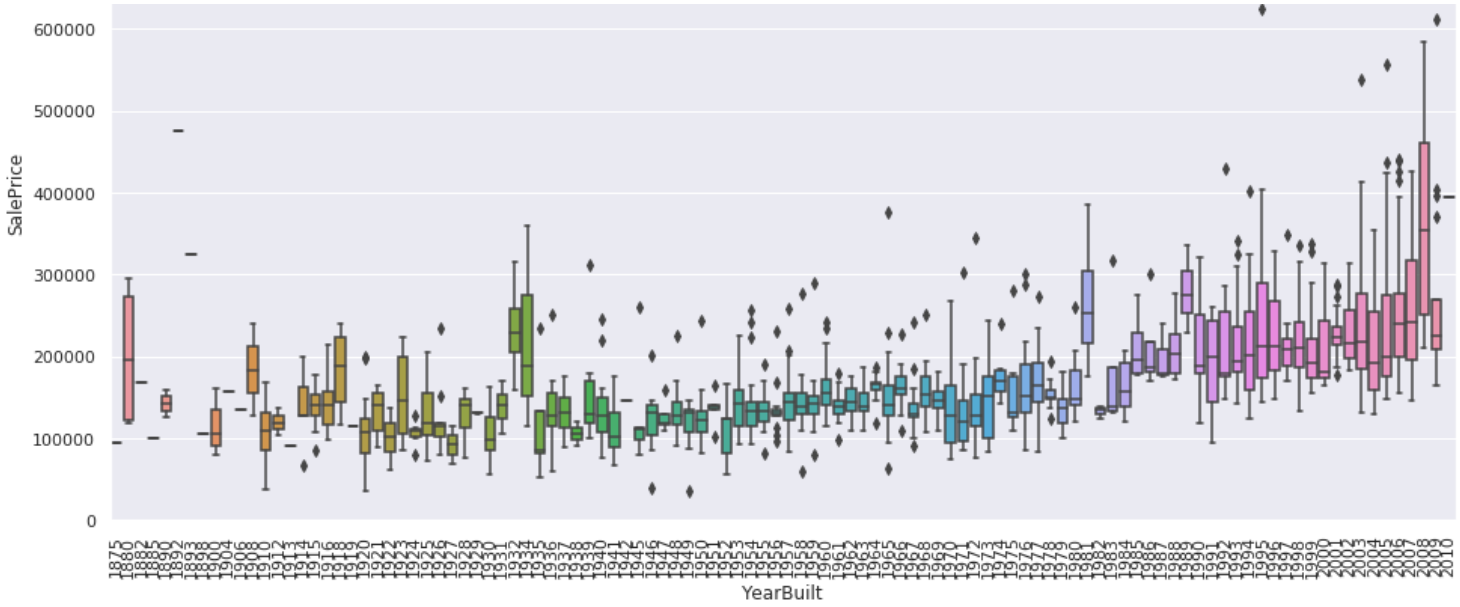
```
#box plot overallqual/saleprice
var = 'OverallQual'
data = pd.concat([trainDF['SalePrice'], trainDF[var]], axis=1)
f, ax = plt.subplots(figsize=(8, 6))
fig = sns.boxplot(x=var, y="SalePrice", data=data)
fig.axis(ymin=0, ymax=800000);
```



We see that median, Q1 and Q2 are growing with higher quality as it would be expected.

In [ ]:

```
#box plot yearbuilt/saleprice
var = 'YearBuilt'
data = pd.concat([trainDF['SalePrice'], trainDF[var]], axis=1)
f, ax = plt.subplots(figsize=(16, 8))
fig = sns.boxplot(x=var, y="SalePrice", data=data)
fig.axis(ymin=0, ymax=800000);
plt.xticks(rotation=90);
```

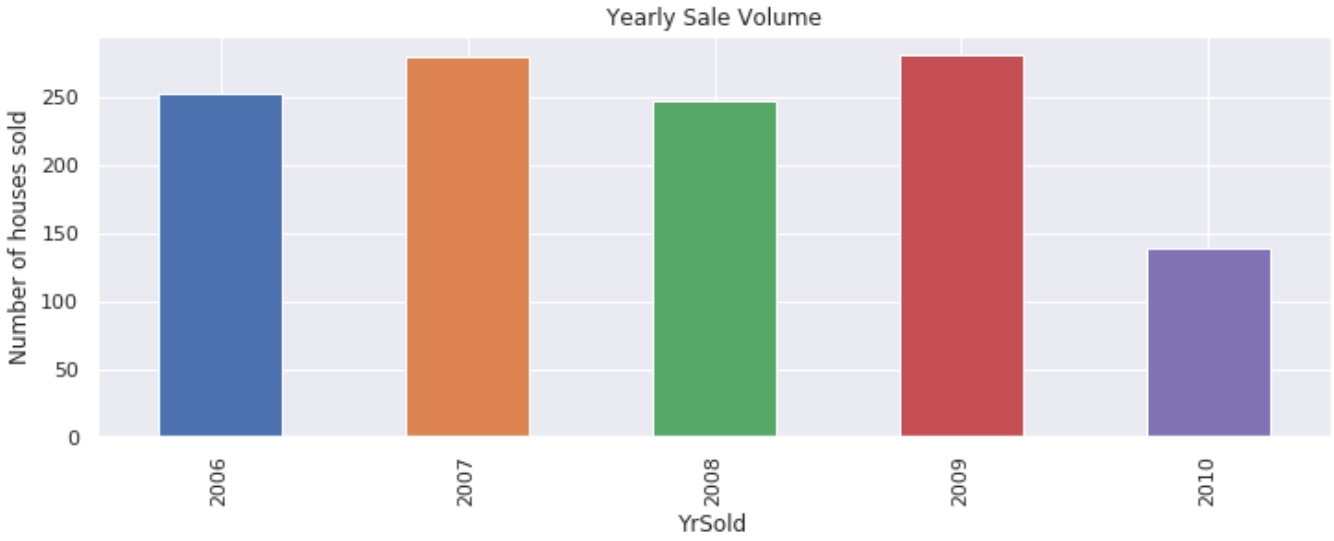


It seems that the price of recent-built houses are higher. Except for the gap at 2009 that we can explain by the crisis. Later we'll use labelencoder for three "Year" feature.

More analysis on the features related to years

In [ ]:

```
#Plot of yearly sale volume
fig = plt.figure(figsize=(12,4))
YearlySale = trainDF.groupby('YrSold').YrSold.size()
YearlySale.plot(x='YrSold', y='Number of houses sold', kind='bar')

plt.ylabel('Number of houses sold')
plt.title('Yearly Sale Volume')
plt.show()
```

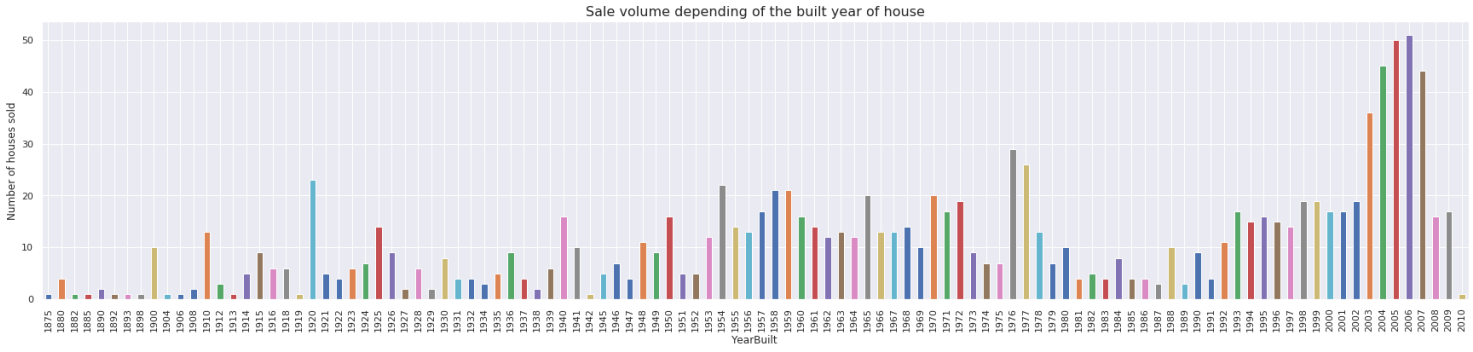


Indeed in 2010 the number of houses sold dropped almost twice.

In [ ]:

```
#Plot of Sale volume depending of the built year of house
fig = plt.figure(figsize=(30,6))
Sale = trainDF.groupby('YearBuilt').YearBuilt.size()
Sale.plot(x='YearBuilt', y='number of houses sold', kind='bar')
```

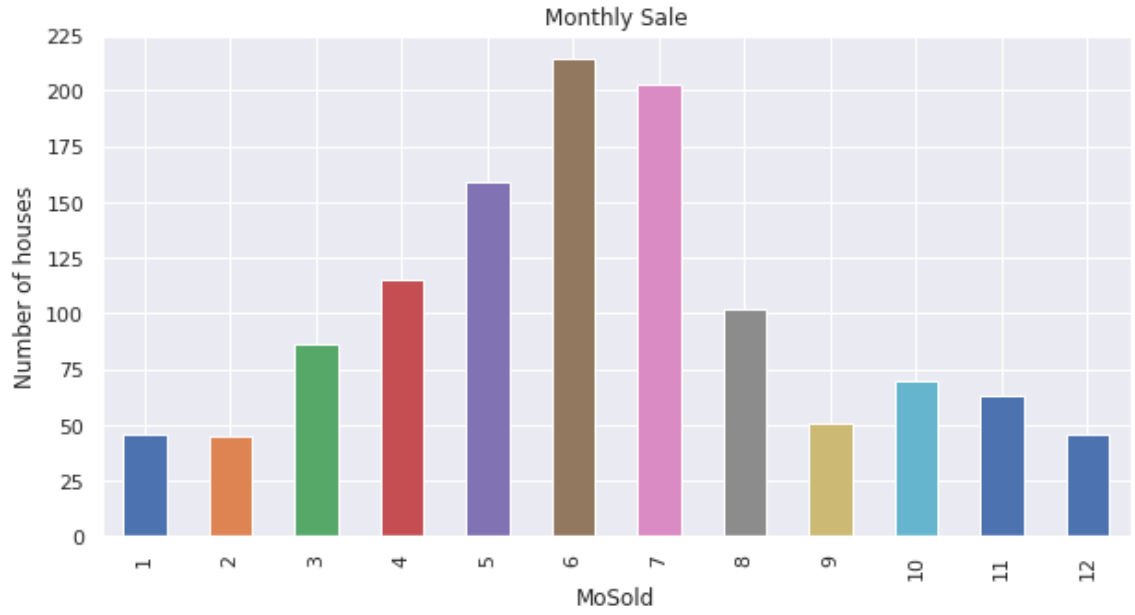
```
plt.ylabel('Number of houses sold')
plt.title('Sale volume depending of the built year of house', fontsize=16)
plt.show()
```



On above figure it's visisble that before the crisis (2003-2007) newly builded houses were sold more easily. It was caused by the speculative bubble which created bigger supply of new houses. Houses built in 2008 and 2009 were sold less easily and only one house from 2010 was sold.

In [ ]:

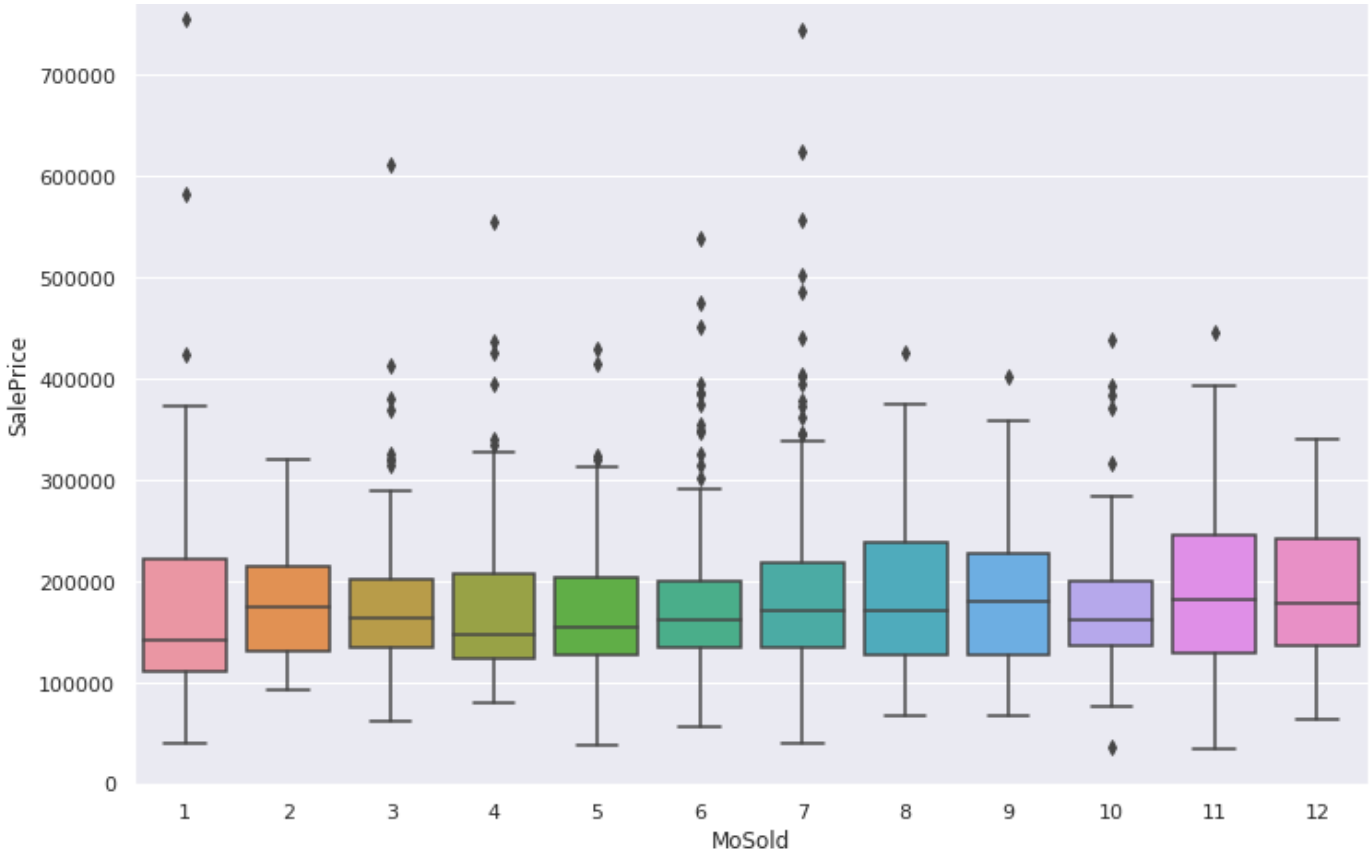
```
#Number of houses sold per month
fig = plt.figure(figsize=(10,5))
MonthlySesonality = trainDF.groupby('MoSold').MoSold.size()
MonthlySesonality.plot(x='MoSold', y='number of houses', kind='bar')

plt.ylabel('Number of houses')
plt.title('Monthly Sale')
plt.show()
```



In the spring, the number of houses sold starts to grow. In June and July, monthly sale reaches the maximum. Starting from August, monthly sale drops and reaches minimum in winter months.

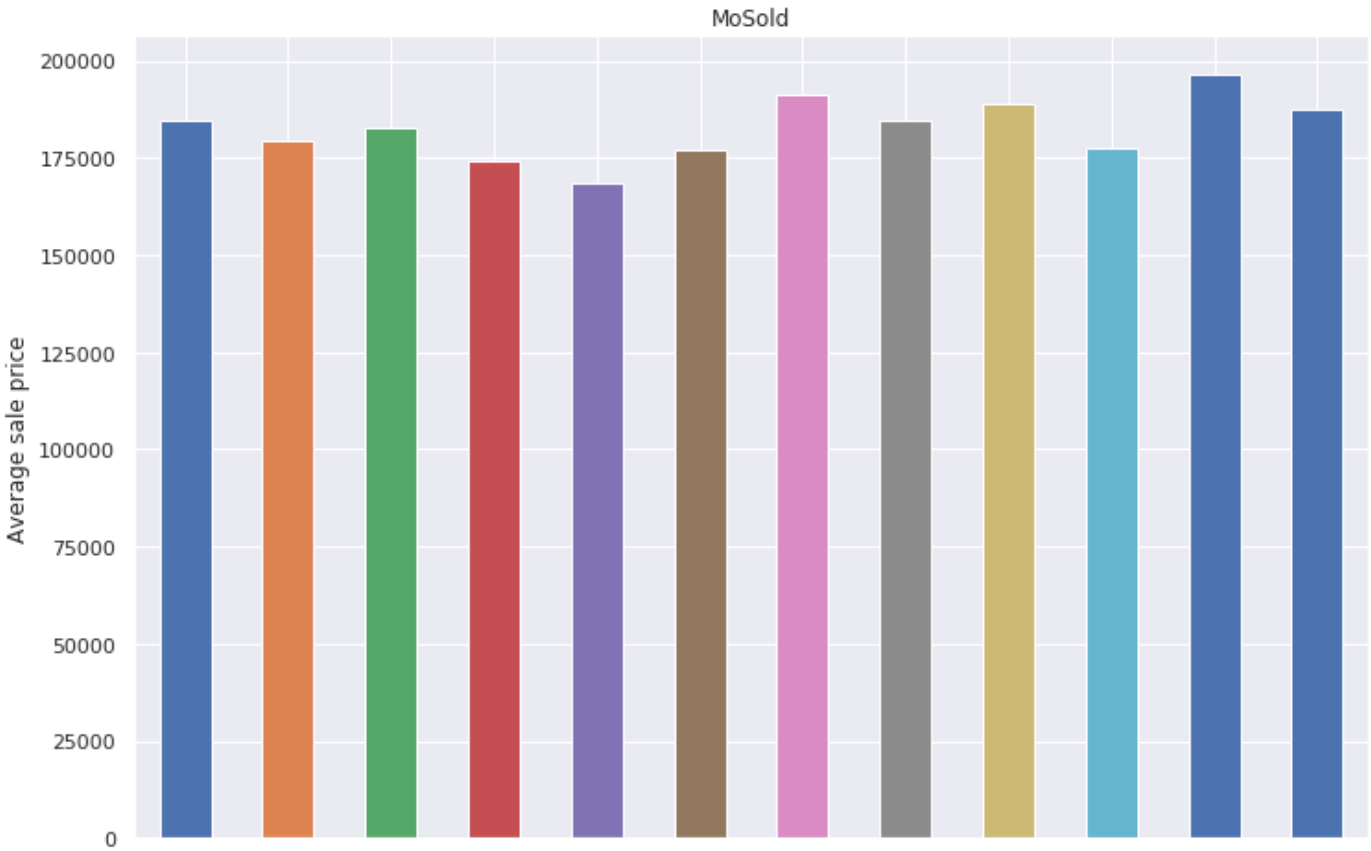
In [ ]:

```
#Months and prices - box plot
fig = plt.figure(figsize=(12,8))
var = 'MoSold'
data = pd.concat([trainDF['SalePrice'], trainDF[var]], axis=1)
ax = sns.boxplot(x=var, y="SalePrice", data=data)
plt.show()
```



In [ ]:

```
#Monts and prices - average
fig = plt.figure(figsize=(12,8))
Monthly = trainDF.groupby('MoSold').SalePrice.mean()
Monthly.plot(x='MoSold', y='SalePrice', kind='bar')

plt.ylabel('Average sale price')
plt.title('MoSold')
plt.show()
```



There is visible seasonality in the monthly price sale. Prices are higher during summer and year end. Prices do not depend on number of houses sold (popularity of the month for sales). The highest prices are in November when the number of houses sold is 4 time less than during June.

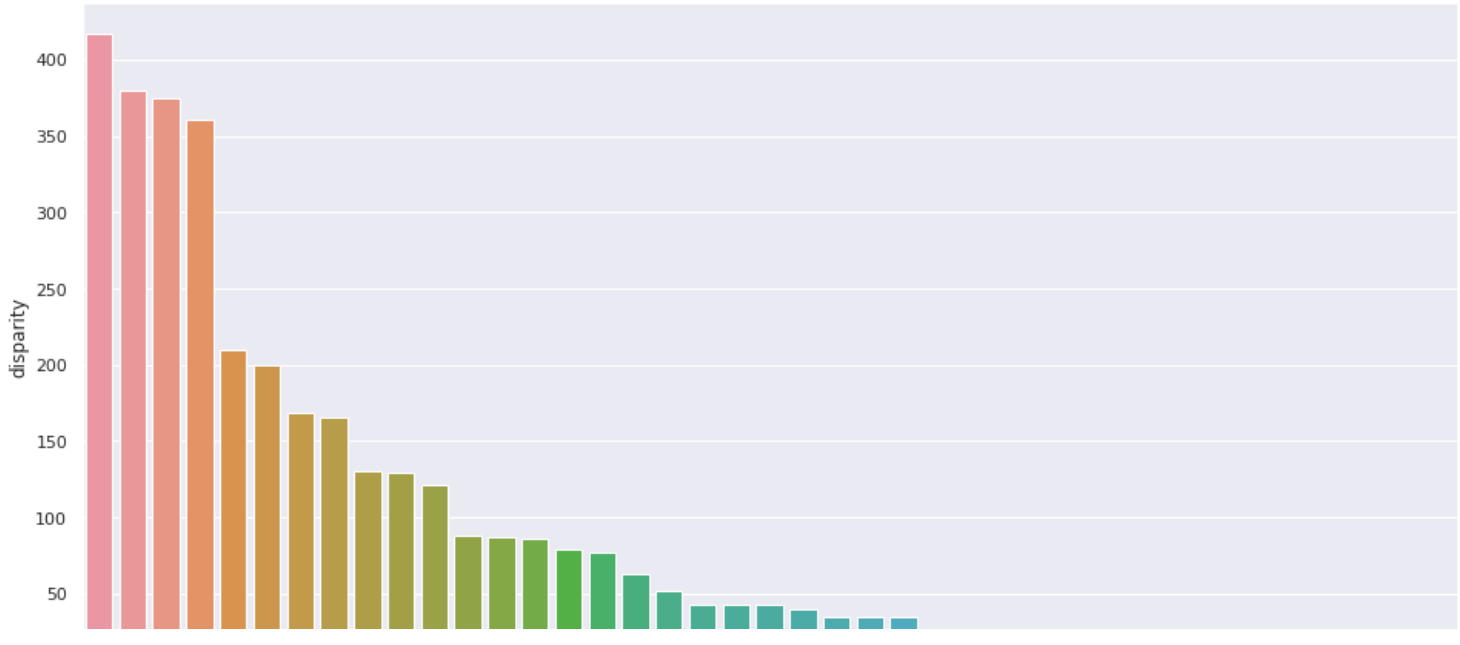
Visualizing disparity on qualitative features

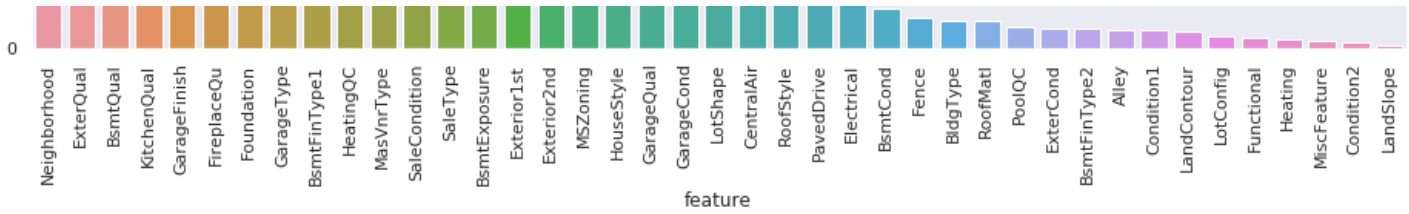
> **Analysis of variance (ANOVA)** is an analysis tool used in statistics that splits the aggregate variability found inside a data set into two parts: systematic factors and random factors. The analysis of variance test is the initial step in analyzing factors that affect a given data set.
>
> The one-way ANOVA tests the null hypothesis that two or more groups have the same population mean. The test is applied to samples from two or more groups, possibly with differing sizes.

https://www.quora.com/How-are-ANOVA-methods-used-in-Data-Science

In [ ]:

```
def anova(frame):
    anv=pd.DataFrame()
    anv['feature'] = qualitative #only testing disparity of qualitative features here
    pvals =[]
    for c in qualitative:
        samples = []
        for cls in frame[c].unique():
            s = frame[frame[c] == cls]['SalePrice'].values
            samples.append(s)
        pval = stats.f_oneway(*samples)[1]  #.
        pvals.append(pval)

    anv['pval'] = pvals
    return anv.sort_values('pval')

f, ax = plt.subplots(figsize=(16, 8))
a = anova(trainDF)
a['disparity']= np.log(1./a['pval'].values)
sns.barplot(data=a, x='feature',y='disparity')
x=plt.xticks(rotation=90)
```





It is difficult to say if we should remove the features with the less disparity. As we are not trying to reduce the dimension but make a good prediction, we decided to keep the features as they are.

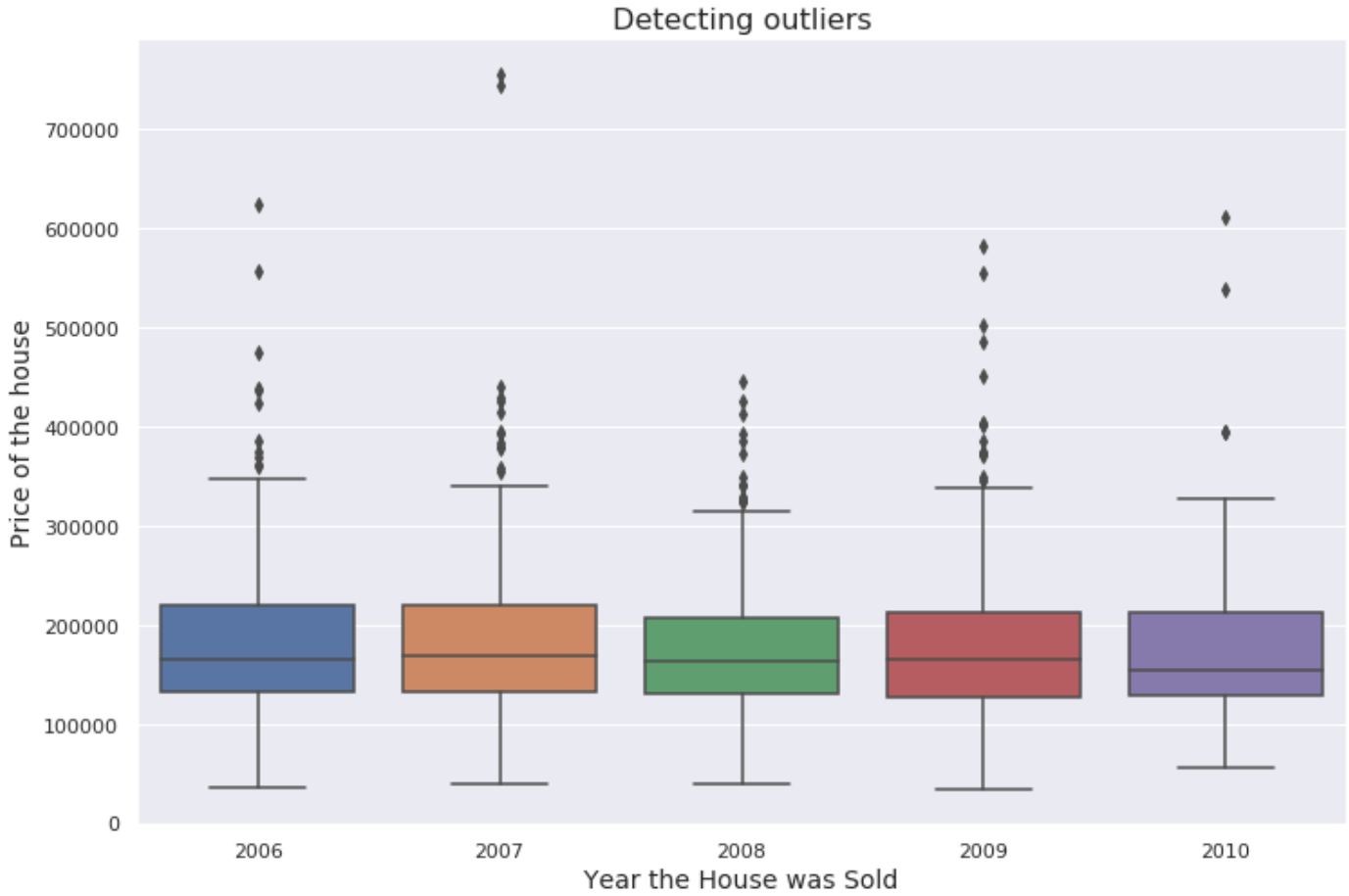
A lot more analysis could be done as did in the kernel of janioobachmann (influence of zone/neighborhood, regrouping features in new categories like space features etc.)

2. Data Pre-processing

Outliers : univariate analysis

In [ ]:

```
fig = plt.figure(figsize=(12,8))
ax = sns.boxplot(x="YrSold", y="SalePrice", data=trainDF)
plt.title('Detecting outliers', fontsize=16)
plt.xlabel('Year the House was Sold', fontsize=14)
plt.ylabel('Price of the house', fontsize=14)
plt.show()
```

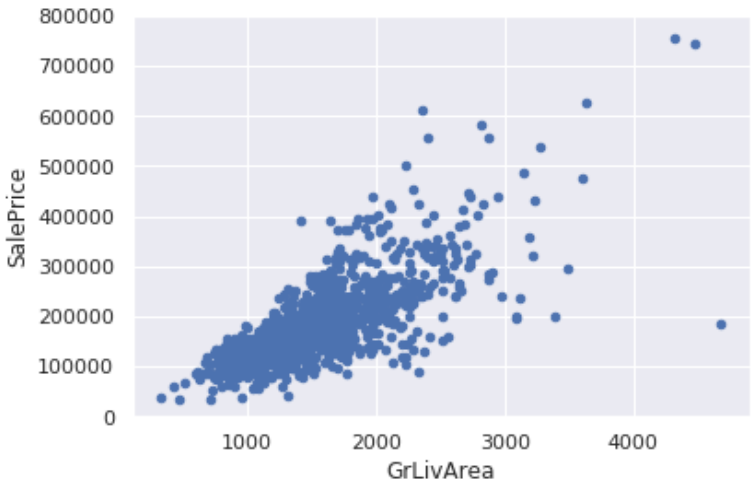


The year of 2007 had the highest outliers (peak of the housing market before collapse). We still observe a drop at 2010 and much less outliers.

Outliers : bivariate analysis (Detecting outliers through visualizations)

In [ ]:

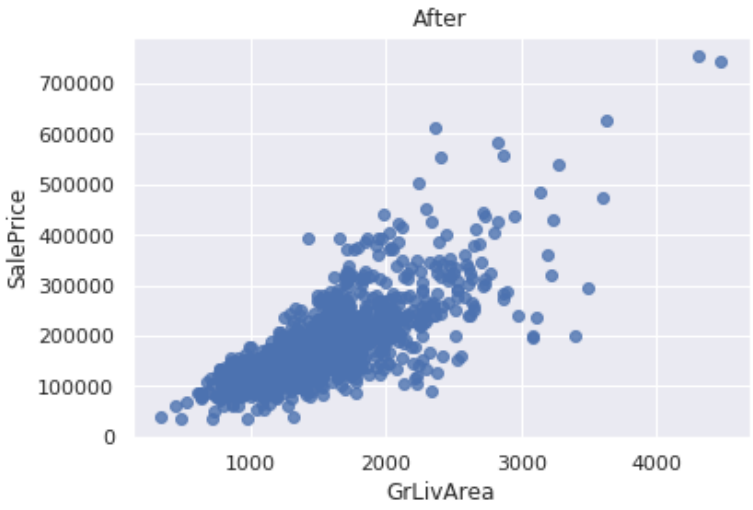
```
#bivariate analysis saleprice/GrLivArea
data = pd.concat([trainDF['SalePrice'], trainDF['GrLivArea']], axis=1)
data.plot.scatter(x='GrLivArea', y='SalePrice', ylim=(0,800000));
```



There are two far points on the top right corner but they seem to follow the trend so we will keep them. We notice one outlier at the bottom right corner that we will remove now. This can be an isolated house (agricultural area) which would explain why though it is really big the price is low.

In [ ]:

```
#After removing the outlier
trainDF = trainDF.drop(trainDF[((trainDF.GrLivArea>4000) & (trainDF.SalePrice<600000))].ind
ex)
g = sns.regplot(x=trainDF['GrLivArea'],y=trainDF['SalePrice'],fit_reg=False).set_title("Aft
er")
```



Combining data

In [ ]:

```
full=pd.concat([trainDF,testDF], ignore_index=True, sort=True)
print(full.shape)
```

```
(1459, 78)
```

Missing data

Missing data

In [ ]:

```
def printMissing(dataframe):
    total = dataframe.isnull().sum().sort_values(ascending=False)
    percent = (dataframe.isnull().sum()/dataframe.isnull().count()).sort_values(ascending=F
alse)
    missing_data = pd.concat([total, percent], axis=1, keys=['Total', 'Percent'])
    display(missing_data[missing_data['Total']>0])

printMissing(full)
```

	Total	Percent
SalePrice	260	0.178204
LotFrontage	259	0.177519
PoolQC	257	0.176148
MiscFeature	253	0.173406
Alley	244	0.167238
Fence	206	0.141193
FireplaceQu	126	0.086361
GarageYrBlt	81	0.055517
GarageFinish	14	0.009596
GarageQual	14	0.009596
GarageType	14	0.009596
GarageCond	14	0.009596
MasVnrArea	8	0.005483
BsmtFinType2	5	0.003427
BsmtExposure	5	0.003427
BsmtFinType1	5	0.003427
BsmtCond	5	0.003427
BsmtQual	5	0.003427
MasVnrType	2	0.001371
Electrical	1	0.000685

There are a lot of missing data. But most of them are just interpreted as missing though they give information, like for the 'PoolQC' NA is not a missing information but it means that there is no pool. For 'LotFrontage' we need to calculate it approximately.

In [ ]:

```
# When NA should be 0
cols=["MasVnrArea"]
for col in cols:
    full[col].fillna(0, inplace=True)
```

In [ ]:

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

In [ ]:

```
# When NA means None (no garage for example)
cols1 = ["PoolQC" , "MiscFeature", "Alley", "Fence", "FireplaceQu",  "GarageType", "GarageF
inish", "GarageQual", "GarageCond",
         "GarageYrBlt", "BsmtExposure", "BsmtFinType2", "BsmtCond", "BsmtQual", "BsmtFinTyp
e1", "MasVnrType"]

for col in cols1:
    full[col].fillna("None", inplace=True)
    full.replace({"MISSING":"None"}, inplace=True) #do not forget that we replaced na with
MISSING before for the plot
```

In [ ]:

```
#When we need to specify a category, set it to the most frequent
cols2 = ["Electrical"]
for col in cols2:
    full[col].fillna(testDF[col].mode()[0], inplace=True) #most frequent of the test set (w
here it is missing)
```

As described in the comment of the kernel of massquantity :

> **The correct way would be to fill in training dataset with the mode and after do the same for the test data, separately. By doing so, you don't take into account the observations of the test set.**

Fortunately there is only one missing for 'Electrical' and so we can just take the most frequent of the test set (where it is missing)

In [ ]:

```
printMissing(full)
```

	Total	Percent
SalePrice	260	0.178204

No more missing data, except for the value we want to predict !

Feature Engineering

Converting some numerical features into categorical features to use get_dummies for these features.

In [ ]:

```
NumStr = ["MSSubClass","BsmtFullBath","BsmtHalfBath","HalfBath","BedroomAbvGr","KitchenAbvGr
","MoSold",
          "YrSold","YearBuilt","YearRemodAdd","LowQualFinSF","GarageYrBlt"]
for col in NumStr:
    full[col]=full[col].astype(str)
```

Next we need to map categorical features to values. We used the mapping made by 'massquantity' where he added a small "o" in front of the features to keep the original ones so that we can still use get_dummies.

In [ ]:

```
#Sort and print to determine a first mapping for MSSubClass
full.groupby(['MSSubClass'])[['SalePrice']].agg(['mean','median','count']).sort_values(by=(
'SalePrice', 'mean'))
```

Out[ ]:

	SalePrice		
	mean	median	count
MSSubClass			
180	92285.714286	84500.0	7
30	97425.910714	100000.0	56
45	108591.666667	107500.0	12
40	121500.000000	133000.0	3
190	128922.222222	129000.0	27
90	132849.658537	136500.0	41
160	139202.509434	147400.0	53
50	140901.589286	130750.0	112
85	151113.333333	147000.0	15
70	167094.851064	155000.0	47
80	168726.136364	164200.0	44
20	183355.488688	159000.0	442
75	196633.333333	164000.0	15
120	200819.287671	192500.0	73
60	243890.333333	219500.0	252

In [ ]:

```
# Encode some categorical features as ordered numbers when there is information in the order
def map_values():
    full["oMSSubClass"] = full.MSSubClass.map({'180':1,
                                        '30':2, '45':2,
                                        '190':3, '50':3, '90':3,
                                        '85':4, '40':4, '160':4,
                                        '70':5, '20':5, '75':5, '80':5, '150':5,
                                        '120': 6, '60':6})

    full["oMSZoning"] = full.MSZoning.map({'C (all)':1, 'RH':2, 'RM':2, 'RL':3, 'FV':4})

    full["oNeighborhood"] = full.Neighborhood.map({'MeadowV':1,
                                                   'IDOTRR':2, 'BrDale':2,
                                                   'OldTown':3, 'Edwards':3, 'BrkSide':3,
                                                   'Sawyer':4, 'Blueste':4, 'SWISU':4, 'NAmes'
:4,
                                                   'NPkVill':5, 'Mitchel':5,
                                                   'SawyerW':6, 'Gilbert':6, 'NWAmes':6,
                                                   'Blmngtn':7, 'CollgCr':7, 'ClearCr':7, 'Cra
wfor':7,
                                                   'Veenker':8, 'Somerst':8, 'Timber':8,
                                                   'StoneBr':9,
                                                   'NoRidge':10, 'NridgHt':10})

    full["oCondition1"] = full.Condition1.map({'Artery':1,
                                           'Feedr':2, 'RRAe':2,
                                           'Norm':3, 'RRAn':3,
                                           'PosN':4, 'RRNe':4,
                                           'PosA':5 ,'RRNn':5})

    full["oBldgType"] = full.BldgType.map({'2fmCon':1, 'Duplex':1, 'Twnhs':1, '1Fam':2, 'Tw
nhsE':2})

    full["oHouseStyle"] = full.HouseStyle.map({'1.5Unf':1,
```

```
                                           '1.5Fin':2, '2.5Unf':2, 'SFoyer':2,
                                           '1Story':3, 'SLvl':3,
                                           '2Story':4, '2.5Fin':4})

    full["oExterior1st"] = full.Exterior1st.map({'BrkComm':1,
                                             'AsphShn':2, 'CBlock':2, 'AsbShng':2,
                                             'WdShing':3, 'Wd Sdng':3, 'MetalSd':3, 'Stucc
o':3, 'HdBoard':3,
                                             'BrkFace':4, 'Plywood':4,
                                             'VinylSd':5,
                                             'CemntBd':6,
                                             'Stone':7, 'ImStucc':7})

    full["oMasVnrType"] = full.MasVnrType.map({'BrkCmn':1, 'None':1, 'BrkFace':2, 'Stone':3
})

    full["oExterQual"] = full.ExterQual.map({'Fa':1, 'TA':2, 'Gd':3, 'Ex':4})

    full["oFoundation"] = full.Foundation.map({'Slab':1,
                                           'BrkTil':2, 'CBlock':2, 'Stone':2,
                                           'Wood':3, 'PConc':4})

    full["oBsmtQual"] = full.BsmtQual.map({'Fa':2, 'None':1, 'TA':3, 'Gd':4, 'Ex':5})

    full["oBsmtExposure"] = full.BsmtExposure.map({'None':1, 'No':2, 'Av':3, 'Mn':3, 'Gd':4
})

    full["oHeating"] = full.Heating.map({'Floor':1, 'Grav':1, 'Wall':2, 'OthW':3, 'GasW':4,
'GasA':5})

    full["oHeatingQC"] = full.HeatingQC.map({'Po':1, 'Fa':2, 'TA':3, 'Gd':4, 'Ex':5})

    full["oKitchenQual"] = full.KitchenQual.map({'Fa':1, 'TA':2, 'Gd':3, 'Ex':4})

    full["oFunctional"] = full.Functional.map({'Maj2':1, 'Maj1':2, 'Min1':2, 'Min2':2, 'Mod
':2, 'Sev':2, 'Typ':3})

    full["oFireplaceQu"] = full.FireplaceQu.map({'None':1, 'Po':1, 'Fa':2, 'TA':3, 'Gd':4,
'Ex':5})

    full["oGarageType"] = full.GarageType.map({'CarPort':1, 'None':1,
                                           'Detchd':2,
                                           '2Types':3, 'Basment':3,
                                           'Attchd':4, 'BuiltIn':5})

    full["oGarageFinish"] = full.GarageFinish.map({'None':1, 'Unf':2, 'RFn':3, 'Fin':4})

    full["oPavedDrive"] = full.PavedDrive.map({'N':1, 'P':2, 'Y':3})

    full["oSaleType"] = full.SaleType.map({'COD':1, 'ConLD':1, 'ConLI':1, 'ConLw':1, 'Oth':
1, 'WD':1,
                                       'CWD':2, 'Con':3, 'New':3})

    full["oSaleCondition"] = full.SaleCondition.map({'AdjLand':1, 'Abnorml':2, 'Alloca':2,
'Family':2, 'Normal':3, 'Partial':4})

    return 'Done'
```

In [ ]:

```
map_values()
```

Out[ ]:

```
'Done'
```

In [ ]:

```
#Drop the column SalePrice
full.drop(['SalePrice'],axis=1,inplace=True)
```

Functions for the pipeline

Now we can build a pipeline so it will be easier to choose the features. The purpose of the pipeline is to assemble several steps that can be cross-validated together while setting different parameters.

In [ ]:

```
#Normalizing YearBuilt, YearRemodAdd and GarageYrBlt
class labelenc(BaseEstimator, TransformerMixin):
    def __init__(self):
        pass

    def fit(self,X,y=None):
        return self

    def transform(self,X):
        lab=LabelEncoder()  #encoder to encode labels with value between 0 and shape-1. sha
pe of the fitting array
        X["YearBuilt"] = lab.fit_transform(X["YearBuilt"]) #Fit label encoder and return en
coded labels
        X["YearRemodAdd"] = lab.fit_transform(X["YearRemodAdd"])
        X["GarageYrBlt"] = lab.fit_transform(X["GarageYrBlt"])
        return X
```

NB of the use of this encoder : There is another way apparenty find in the kernel of "janiobachmann" that uses "one-hot encoder" and no "labelEncoder", then the author separates into categorical and numerical features and uses two pipeline. We kept the first way (using Label Encoder) as we get good result with this method.

> **This is a way to encode our features in a way that it avoids the assumption that two nearby values are more similar than two distant values. This is the reason we should avoid using LabelEncoder to scale features (inputs) in our dataset and in addition the word LabelEncoder is used for scaling labels (outputs). This could be used more often in binary classification problems were no association exists between the outputs.**

In [ ]:

```
#Applying log to the features with skewness and applying get_dummies to treat categorical fe
atures for which the
#skewness is greater than threshold
class skew_dummies(BaseEstimator, TransformerMixin):
    def __init__(self,skew=0.5): #a skewness with an absolute value > 0.5 can be considered
at least moderately skewed
        self.skew = skew

    def fit(self,X,y=None):
        return self

    def transform(self,X):
        X_numeric=X.select_dtypes(exclude=["object"]) #keep only quantitative columns
        skewness = X_numeric.apply(lambda x: skew(x)) #Computes the skewness of a data set.
        skewness_features = skewness[abs(skewness) >= self.skew].index # features with skew
greater than threshold
        X[skewness_features] = np.log1p(X[skewness_features]) #Apply log to the skewed feat
ures
        X = pd.get_dummies(X) #Convert categorical variable into dummy/indicator variables
        return X
```

In [ ]:

```
# save the original data for later use
full2 = full.copy()
```

A first pipeline for the features selection

Next we define a first pipeline that will allow us to execute in order the previous functions to process the features.

NB choice of skewness threshold : Applying logarithm to features with an absolute skewness greater than 0.5 gives better result than when the threshold is 1. Yet, this is true only for the final prediction, meaning that when taking the predictions of the models apart, the loss could be greater. As we only care about the final result, we kept this threshold of 0.5.

In [ ]:

```
#Pipeline with a list of (name, transform) tuples (implementing fit/transform) that are chai
ned in order
pipe = Pipeline([
    ('labenc', labelenc()),
    ('skew_dummies', skew_dummies(skew=0.5)),
    ])

data_pipe = pipe.fit_transform(full2)
print(data_pipe.shape)
display(data_pipe.head())
```

```
(1459, 389)
```

	1stFlrSF	2ndFlrSF	3SsnPorch	BsmtFinSF1	BsmtFinSF2	BsmtUnfSF	EnclosedPorch	Fireplaces	FullBath	GarageArea	...	Sa
0	6.753438	6.751101	0.0	6.561031	0.0	5.017280	0.000000	0.000000	2	548	...	
1	7.141245	0.000000	0.0	6.886532	0.0	5.652489	0.000000	0.693147	2	460	...	
2	6.825460	6.765039	0.0	6.188264	0.0	6.075346	0.000000	0.693147	2	608	...	
3	6.869014	6.629363	0.0	5.379897	0.0	6.293419	5.609472	0.693147	1	642	...	
4	7.044033	6.960348	0.0	6.486161	0.0	6.196444	0.000000	0.693147	2	836	...	

5 rows × 389 columns

Standardization

Standardization of a dataset is a common requirement for many machine learning estimators. We use the RobustScaler more robust to outliers, and it is possible that we have many outliers as we only removed one manually. The centering and scaling statistics of this scaler are based on percentiles and are therefore not influenced by a few number of very large marginal outliers. Note that the outliers themselves are still present in the transformed data.

> **Typically this is done by removing the mean and scaling to unit variance. However, outliers can often influence the sample mean / variance in a negative way. In such cases, the median and the interquartile range often give better results.**

https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.RobustScaler.html

In [ ]:

```
scaler = RobustScaler() #Scale features using statistics that are robust to outliers.

#Splitting our process data into train and test sets
```

```
n_train=trainDF.shape[0]
X = data_pipe[:n_train]
print("Shape of X : ", X.shape)
test_X = data_pipe[n_train:]
y= trainDF.SalePrice

#fit computes the median and quantiles to be used for scaling, transform centers and scales the data.
X_scaled = scaler.fit(X).transform(X)
print("Shape of X_scaled : ", X_scaled.shape)
y_log = np.log(trainDF.SalePrice) #scaling SalePrice
test_X_scaled = scaler.transform(test_X) #scaler already fitted
```

```
Shape of X :  (1199, 389)
Shape of X_scaled :  (1199, 389)
```

Features selection

We will first use Lasso to choose which features we should combine. Then we will try with Ridge (we could also try with RandomForest).

In [ ]:

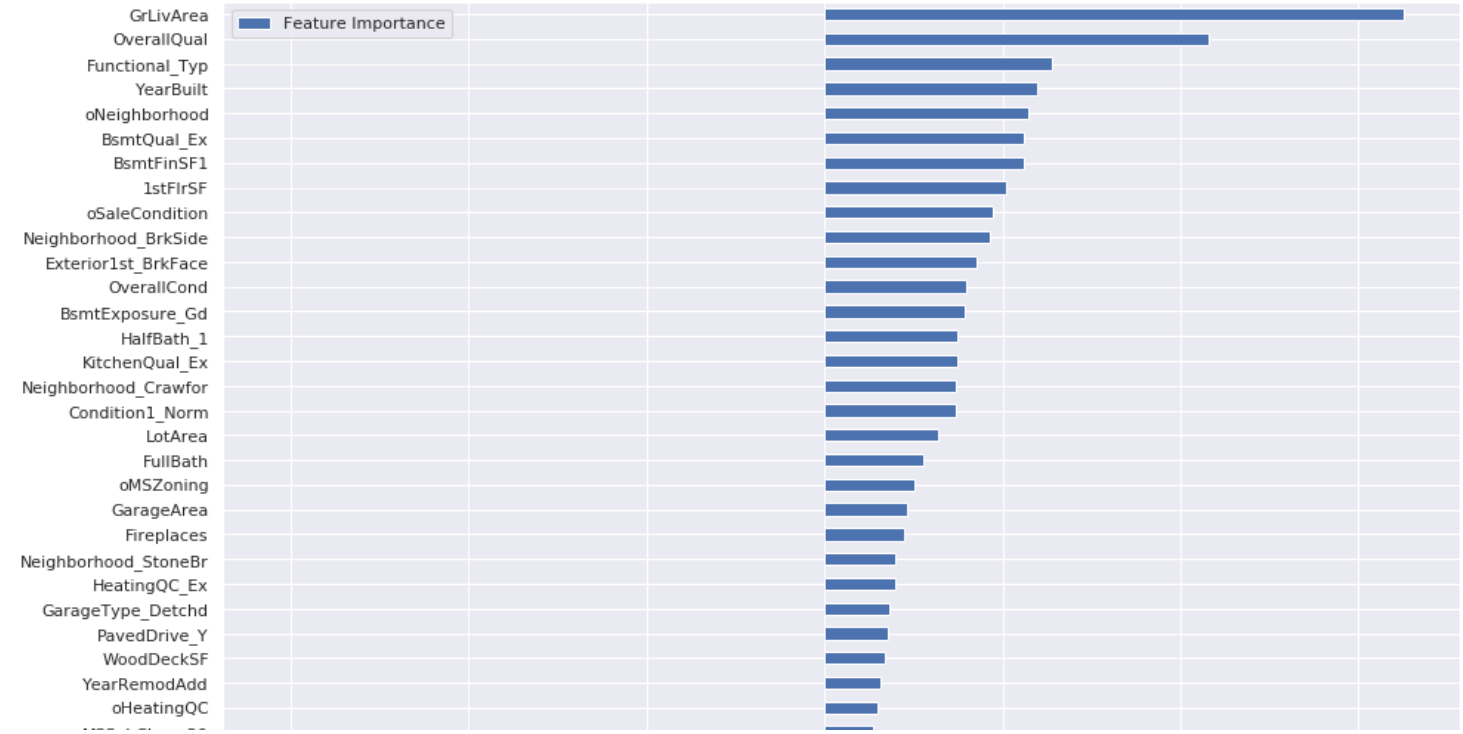
```
lasso=Lasso(alpha=0.001) #quite equivalent to an ordinary least square
lasso.fit(X_scaled,y_log)
```

Out[ ]:

```
Lasso(alpha=0.001, copy_X=True, fit_intercept=True, max_iter=1000,
   normalize=False, positive=False, precompute=False, random_state=None,
   selection='cyclic', tol=0.0001, warm_start=False)
```

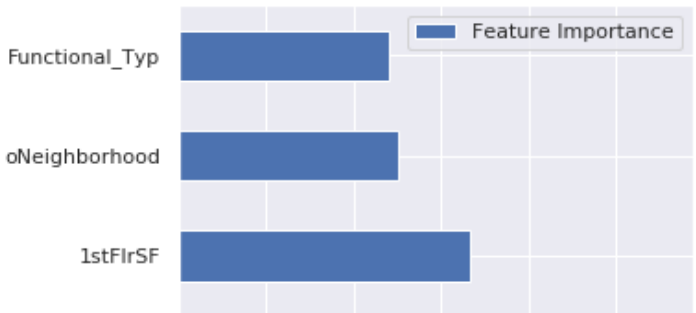
In [ ]:

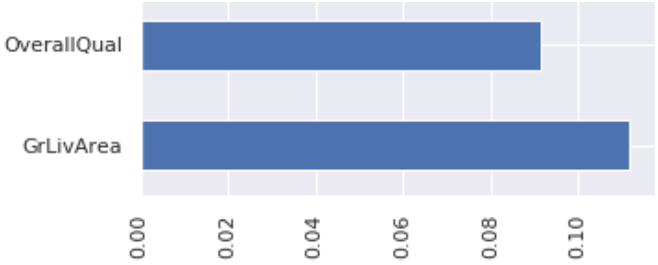
```
FI_lasso = pd.DataFrame({"Feature Importance":lasso.coef_}, index=data_pipe.columns)
FI_lasso.sort_values("Feature Importance",ascending=False)

FI_lasso[FI_lasso["Feature Importance"]!=0].sort_values("Feature Importance").plot(kind="barh",figsize=(15,25))
plt.xticks(rotation=90)
plt.show()
```



In [ ]:

```
ridge = Ridge(alpha=25)
ridge.fit(X_scaled,y_log)

# Plot important coefficients
FI_ridge = pd.DataFrame({"Feature Importance":ridge.coef_}, index=data_pipe.columns)
FI_ridge[FI_ridge["Feature Importance"]!=0].sort_values("Feature Importance", ascending=Fals
e).head().plot(kind="barh", figsize=(5,5))
plt.xticks(rotation=90)
plt.show()
```





There is much more non-zero coefficients displayed for Ridge, meaning that there is much more features described as 'useful' by Ridge. Lasso can set some coefficients to zero, thus performing variable selection, while ridge regression cannot. We only printed the first five most important (positively correlated) features given by Ridge to compare with those given by Lasso. We can see that we do not find the same result (the same order), but as the models use different norm it is not surprising.

> **The main difference between ridge and lasso regression is a shape of the constraint region. Ridge regression use $L2$ norm for a constraint. Lasso uses $L1$ norm for a constraint. In case of Lasso, there is a great propensity for Gradient Descent to intersect at the axes i.e. making the coefficients to zero.**
>
> **Lasso regression helps for feature selection. The main advantage of using Lasso regression for feature selection over other subset selection method (Forward , backward regression) is that it uses convex optimisation to find out the best features. So, it converges faster compared to other method.**

https://www.quora.com/Why-is-it-that-the-lasso-unlike-ridge-regression-results-in-coefficient-estimates-that-are-exactly-equal-to-zero

Features combination

We used the combination of features still from the same kernel, the author based these choices on the "features importance and try-and-error". A lot of these added features make sense : totalArea, adding/multiplying "overallQual" (an important feature) to other features. We tried to add other features (taken from a lot of "good" kernels) but it kept on reducing the performance of the final model.

In [ ]:

```
class add_feature(BaseEstimator, TransformerMixin):
    def __init__(self,additional=1):
        self.additional = additional

    def fit(self,X,y=None):
        return self

    def transform(self,X):
        if self.additional==1:
            X["TotalHouse"] = X["TotalBsmtSF"] + X["1stFlrSF"] + X["2ndFlrSF"]
            X["TotalArea"] = X["TotalBsmtSF"] + X["1stFlrSF"] + X["2ndFlrSF"] + X["GarageAr
ea"]

        else:
            X["TotalHouse"] = X["TotalBsmtSF"] + X["1stFlrSF"] + X["2ndFlrSF"]
            X["TotalArea"] = X["TotalBsmtSF"] + X["1stFlrSF"] + X["2ndFlrSF"] + X["GarageAr
ea"]

            X["+_TotalHouse_OverallQual"] = X["TotalHouse"] * X["OverallQual"]
            X["+_GrLivArea_OverallQual"] = X["GrLivArea"] * X["OverallQual"]
            X["+_oMSZoning_TotalHouse"] = X["oMSZoning"] * X["TotalHouse"]
            X["+_oMSZoning_OverallQual"] = X["oMSZoning"] + X["OverallQual"]
            X["+_oMSZoning_YearBuilt"] = X["oMSZoning"] + X["YearBuilt"]
            X["+_oNeighborhood_TotalHouse"] = X["oNeighborhood"] * X["TotalHouse"]
            X["+_oNeighborhood_OverallQual"] = X["oNeighborhood"] + X["OverallQual"]
```

```
            X["+_oNeighborhood_YearBuilt"] = X["oNeighborhood"] + X["YearBuilt"]
            X["+_BsmtFinSF1_OverallQual"] = X["BsmtFinSF1"] * X["OverallQual"]

            X["-_oFunctional_TotalHouse"] = X["oFunctional"] * X["TotalHouse"]
            X["-_oFunctional_OverallQual"] = X["oFunctional"] + X["OverallQual"]
            X["-_LotArea_OverallQual"] = X["LotArea"] * X["OverallQual"]
            X["-_TotalHouse_LotArea"] = X["TotalHouse"] + X["LotArea"]
            X["-_oCondition1_TotalHouse"] = X["oCondition1"] * X["TotalHouse"]
            X["-_oCondition1_OverallQual"] = X["oCondition1"] + X["OverallQual"]


            X["Bsmt"] = X["BsmtFinSF1"] + X["BsmtFinSF2"] + X["BsmtUnfSF"]
            X["Rooms"] = X["FullBath"]+X["TotRmsAbvGrd"]
            X["PorchArea"] = X["OpenPorchSF"]+X["EnclosedPorch"]+X["3SsnPorch"]+X["ScreenPo
rch"]
            X["TotalPlace"] = X["TotalBsmtSF"] + X["1stFlrSF"] + X["2ndFlrSF"] + X["GarageA
rea"] + X["OpenPorchSF"]+X["EnclosedPorch"]+X["3SsnPorch"]+X["ScreenPorch"]


            return X
```

Final pipeline

We define a new pipeline that this time also add the new combination of features. Then we do the standardization as before.

In [ ]:

```
pipe = Pipeline([
    ('labenc', labelenc()), #Encode feature with years
    ('add_feature', add_feature(additional=2)), #Add new features to the data
    ('skew_dummies', skew_dummies(skew=0.5)), #Apply log to skew features and get_dummies fr
om the data
    ])
```

In [ ]:

```
full_pipe = pipe.fit_transform(full)
print(full_pipe.shape)
```

```
(1459, 410)
```

In [ ]:

```
#As before we split the data
X = full_pipe[:n_train]
test_X = full_pipe[n_train:]
y= trainDF.SalePrice
#and we apply some transformation
X_scaled = scaler.fit(X).transform(X)
y_log = np.log(trainDF.SalePrice)
test_X_scaled = scaler.transform(test_X)
```

Using PCA

> **The purpose of PCA is to perform de-correlation between the dimensions. Thus, the different features of the feature-vector obtained after PCA are uncorrelated and each dimension contributes a specific (and probably interpretable) characteristic of the data. Adding the original features increases the correlation between the dimensions.**
> **The second reason for PCA is dimensionality reduction. After PCA, it is often enough to retain the first k dimensions ranked in terms of their variance. These dimensions are derived from the underlying**

> data and retain most of the important information contained by the data.

https://www.quora.com/Is-there-any-benefit-from-adding-the-original-features-along-with-the-reduced-ones-by-PCA

We made several tests for dimensionality reduction, none of which improve the results in the final model using stacking. Nonetheless sometimes it does improve some models while penalizing others. It does help to reduce the training time but to keep a minimum loss at the end we kept the same number of features and just use the decorrelation properties of PCA.

In [ ]:

```
pca = PCA()
X_scaled=pca.fit_transform(X_scaled)
test_X_scaled = pca.transform(test_X_scaled)
```

In [ ]:

```
print(X_scaled.shape, test_X_scaled.shape)
```

```
(1199, 410) (260, 410)
```

In [ ]:

```
# Checking normal distribution of features in train data
k2, p_array = normaltest(X_scaled, axis=0)
count_normal = 0
count_not_normal = 0
for p in p_array:
    if p > 1e-3: #threshold used in the documentation of scipy.
        count_normal+=1
    else:
        count_not_normal+=1
print("Number of normal features :", count_normal)
print("Number of not normal features :", count_not_normal)
```

```
Number of normal features : 47
Number of not normal features : 363
```

Most of the features are not following a normal distribution. We were not expecting every feature to followed one seeing that some of our features act like booleans for example (for 'YrSold_2006' that we got for example). As we do not know the number of features that should follow a normal distribution it is difficult to say if we did enough to normalize. Still we tried our best when we applied log transform to the features with the most skewness in the pipeline.

3. Model Selection

There are a lot of models possible for regression, we first tried to understand some of them and how we could use them using this website https://scikit-learn.org/stable/supervised_learning.html. More details about algorithms are explained on this website as well as example for how to use the corresponding libraries.

We tried several models as did in the kernel of 'massquantity', then as in general combining models gives better result we will try to combine the best models (based on the result from the cross-validation).

Cross-validation and RMSE

Cross-validation We used cross-validation to compare models. For the choice of K for the K-fold-cross validation we used 5 after some researches, though it depends on the size of the training set.
https://machinelearningmastery.com/k-fold-cross-validation/

> **As a general rule, most authors, and empirical evidence, suggest that 5- or 10- fold cross validation should be preferred to LOO. https://scikit-learn.org/stable/modules/cross_validation.html**
>
> **"To summarize, there is a bias-variance trade-off associated with the choice of k in k-fold cross-validation. Typically, given these considerations, one performs k-fold cross-validation using k = 5 or k = 10, as these values have been shown empirically to yield test error rate estimates that suffer neither from excessively high bias nor from very high variance." — Page 184, An Introduction to Statistical Learning, 2013.**

Root Mean Square Error : We found also several score/loss model, 'neg_mean_squared_log_error' seems to be the closest to what we want but as we already transform the output, 'neg_mean_squared_error' is the model we kept. Functions ending with _error like "neg_mean_squared_error" return a value to minimize, the lower the better. https://scikit-learn.org/stable/modules/model_evaluation.html

In [ ]:

```
# define cross validation strategy
n_folds=5

#Root Mean Square Error
def rmse_cv(model,features,labels):
    rmse = np.sqrt(-cross_val_score(model, features, labels, scoring="neg_mean_squared_erro
r", cv=n_folds))
    return rmse
```

In [ ]:

```
#all the models, kept parameters as define by the kernel of massquantity for now
models = [LinearRegression(),
          Ridge(),
          Lasso(alpha=0.01,max_iter=10000),
          RandomForestRegressor(),
          GradientBoostingRegressor(),
          SVR(),
          LinearSVR(),
          ElasticNet(alpha=0.001,max_iter=10000),
          SGDRegressor(max_iter=1000,tol=1e-3),
          BayesianRidge(),
          KernelRidge(alpha=0.6, kernel='polynomial', degree=2, coef0=2.5),
          ExtraTreesRegressor()]
```

In [ ]:

```
names = ["LR", "Ridge", "Lasso", "RF", "GBR", "SVR", "LinSVR", "Ela","SGD","Bay","Ker","Ext
ra"]

scoreDF = pd.DataFrame(columns=['model', 'mean_test_score','std_test_score', 'time'])
#obtaining the score (which is a loss) for each model
for name, model in zip(names, models):
    start_time = time()
    rmse = rmse_cv(model, X_scaled, y_log)
    df = pd.DataFrame([[name, format(rmse.mean(), '.6f'), format(rmse.std(), '.6f'), time()
-start_time]], columns= scoreDF.columns)
    scoreDF = scoreDF.append(df)

display(scoreDF.sort_values(by='mean_test_score'))
```

	model	mean_test_score	std_test_score	time
0	Ker	0.110600	0.007070	1.087605
0	Bay	0.112161	0.007120	4.303633
0	Ela	0.112619	0.007304	0.192736

	model	mean_test_score	std_test_score	time
0	SVR	0.118332	0.011896	0.858514
0	Ridge	0.121106	0.009347	0.386245
0	Lasso	0.123605	0.009873	0.294430
0	LinSVR	0.127935	0.014107	8.306013
0	GBR	0.134682	0.013953	11.458673
0	Extra	0.150964	0.012606	2.679502
0	RF	0.162894	0.016581	6.469246
0	SGD	0.183114	0.007429	0.690207
0	LR	3413665532.979381	1891686983.656648	2.173264

We can already decide which models give the best result and so the ones we should keep when we will combine the models : KernelRidge, Bayesian Ridge, ElasticNet, SVR, Ridge and Lasso. We can see that LinearRegression gives a really big error, this is not that surprising as this is a 'basic' model (ordinary least squares linear regression).

When looking at the time the models take, we can see that all our selected models take less than a second to complete (with the selected parameters : the K in the K-fold cross validation etc.), BayesianRidge and SVR are the ones that are the slowest in these 6 models. We see that there is no real need to do features selection to decrease the time of computation in these models. When looking at all tested models, we see that GradientBoostingRegressor, LinearSVR and RandomForestRegressor take much more time, from 6 to 12sec, these models are more complex and yet they give worse results in our case.

4. Parameter Optimisation

Let us do some parameters tuning to improve each model. For that we will define a grid function that basically used the GridSearchCV to try several combination of parameters and add the corresponding score in a dataframe.

In [ ]:

```
#Defining a grid to search for the optimal parameters for each model
class grid():
    def __init__(self,model):
        self.model = model

    def grid_get(self,X,y,param_grid):
        grid_search = GridSearchCV(self.model,param_grid,cv=5, scoring="neg_mean_squared_er
ror", return_train_score=True)
        grid_search.fit(X,y)
        print("First best param_grid :", grid_search.best_params_, np.sqrt(-grid_search.bes
t_score_))
        grid_search.cv_results_['mean_test_score'] = np.sqrt(-grid_search.cv_results_['mean
_test_score'])
        display(pd.DataFrame(grid_search.cv_results_)[['params','mean_test_score','std_test
_score']].sort_values(by='mean_test_score').head())
```

In [ ]:

```
#Lasso
grid(Lasso()).grid_get(X_scaled,y_log,{'alpha': [0.0004,0.0005,0.0006,0.0008],'max_iter':[1
0000]})
```

```
First best param_grid : {'alpha': 0.0005, 'max_iter': 10000} 0.11291282723748745
```

	params	mean_test_score	std_test_score
1	{'alpha': 0.0005, 'max_iter': 10000}	0.112913	0.001648
2	{'alpha': 0.0006, 'max_iter': 10000}	0.112981	0.001696

	params	mean_test_score	std_test_score
0	{'alpha': 0.0004, 'max_iter': 10000}	0.113190	0.001596
3	{'alpha': 0.0008, 'max_iter': 10000}	0.113355	0.001758

In [ ]:

```
#Ridge
alpha = grid(Ridge()).grid_get(X_scaled,y_log,{'alpha':[25,30,34,35,36,40]})
```

```
First best param_grid : {'alpha': 35} 0.11215491322641323
```

	params	mean_test_score	std_test_score
3	{'alpha': 35}	0.112155	0.001641
2	{'alpha': 34}	0.112155	0.001637
4	{'alpha': 36}	0.112156	0.001644
5	{'alpha': 40}	0.112167	0.001658
1	{'alpha': 30}	0.112168	0.001624

In [ ]:

```
#SVR Epsilon-Support Vector Regression.
grid(SVR()).grid_get(X_scaled,y_log,{'C':[14,15,16],"gamma":[0.0004,0.0005,0.0006],"epsilon
":[0.009,0.01, 0.11]})
```

```
First best param_grid : {'epsilon': 0.01, 'gamma': 0.0005, 'C': 15} 0.10971718120663239
```

	params	mean_test_score	std_test_score
13	{'epsilon': 0.01, 'gamma': 0.0005, 'C': 15}	0.109717	0.002291
4	{'epsilon': 0.01, 'gamma': 0.0005, 'C': 14}	0.109720	0.002279
22	{'epsilon': 0.01, 'gamma': 0.0005, 'C': 16}	0.109745	0.002305
5	{'epsilon': 0.01, 'gamma': 0.0006, 'C': 14}	0.109776	0.002320
14	{'epsilon': 0.01, 'gamma': 0.0006, 'C': 15}	0.109796	0.002332

In [ ]:

```
#KernelRidge
grid(KernelRidge()).grid_get(X_scaled,y_log,{'alpha':[0.1, 0.2,0.3], 'kernel':["polynomial",
"linear"], 'degree':[3,4],'coef0':[0.9,1.0,1.1]})
```

```
First best param_grid : {'alpha': 0.2, 'coef0': 1.0, 'kernel': 'polynomial', 'degree': 3} 0.
10927436659935798
```

	params	mean_test_score	std_test_score
16	{'alpha': 0.2, 'coef0': 1.0, 'kernel': 'polyno...	0.109274	0.001520
34	{'alpha': 0.3, 'coef0': 1.1, 'kernel': 'polyno...	0.109285	0.001548
20	{'alpha': 0.2, 'coef0': 1.1, 'kernel': 'polyno...	0.109317	0.001528
12	{'alpha': 0.2, 'coef0': 0.9, 'kernel': 'polyno...	0.109322	0.001517
30	{'alpha': 0.3, 'coef0': 1.0, 'kernel': 'polyno...	0.109364	0.001546

NB : https://scikit-learn.org/stable/modules/generated/sklearn.kernel_ridge.KernelRidge.html

> **The form of the model learned by KRR is identical to support vector regression (SVR). However, different loss functions are used: KRR uses squared error loss while support vector regression uses epsilon-insensitive loss, both combined with l2 regularization. In contrast to SVR, fitting a KRR model**

> **can be done in closed-form and is typically faster for medium-sized datasets. On the other hand, the learned model is non-sparse and thus slower than SVR, which learns a sparse model for epsilon > 0, at prediction-time.**

In [ ]:

```
#ElasticNet
grid(ElasticNet()).grid_get(X_scaled,y_log,{'alpha':[0.006,0.007, 0.008, 0.009],'l1_ratio':
[0.05, 0.06, 0.07],'max_iter':[10000]})
```

```
First best param_grid : {'alpha': 0.008, 'l1_ratio': 0.05, 'max_iter': 10000} 0.112549901361
30114
```

	params	mean_test_score	std_test_score
6	**{'alpha': 0.008, 'l1_ratio': 0.05, 'max_iter':...**	**0.112550**	**0.001624**
9	**{'alpha': 0.009, 'l1_ratio': 0.05, 'max_iter':...**	**0.112560**	**0.001652**
3	**{'alpha': 0.007, 'l1_ratio': 0.05, 'max_iter':...**	**0.112564**	**0.001607**
4	**{'alpha': 0.007, 'l1_ratio': 0.06, 'max_iter':...**	**0.112575**	**0.001628**
7	**{'alpha': 0.008, 'l1_ratio': 0.06, 'max_iter':...**	**0.112585**	**0.001665**

In [ ]:

```
#BayesianRidge, alpha : precision of the noise, lambda : precision of the weights
grid(BayesianRidge()).grid_get(X_scaled,y_log,{'alpha_1':[0.000001, 0.001],'alpha_2':[0.0000
01, 0.01, 1],
                                               'lambda_1':[0.000001, 0.001, 0.1, 1],'lambd
a_2':[0.000001, 0.001],'n_iter':[10000]})
```

```
First best param_grid : {'alpha_1': 1e-06, 'n_iter': 10000, 'alpha_2': 1, 'lambda_1': 1, 'la
mbda_2': 1e-06} 0.11220705477653321
```

	params	mean_test_score	std_test_score
22	**{'alpha_1': 1e-06, 'n_iter': 10000, 'alpha_2':...**	**0.112207**	**0.001633**
46	**{'alpha_1': 0.001, 'n_iter': 10000, 'alpha_2':...**	**0.112207**	**0.001633**
20	**{'alpha_1': 1e-06, 'n_iter': 10000, 'alpha_2':...**	**0.112210**	**0.001630**
44	**{'alpha_1': 0.001, 'n_iter': 10000, 'alpha_2':...**	**0.112210**	**0.001630**
18	**{'alpha_1': 1e-06, 'n_iter': 10000, 'alpha_2':...**	**0.112211**	**0.001630**

We did improve the performance of our models. Without giving the exact calculation, we can see for example that we reduced the loss of Lasso passing from 0.123 to 0.113, the loss of KernelRidge from 0.11 to 0.109 etc. While some models did not improve like Bayesian Ridge when looking at the mean of the loss, we still see that the standard deviation has been reduced (from 0.007 to 0.002 approximately).

5. Model Evaluation

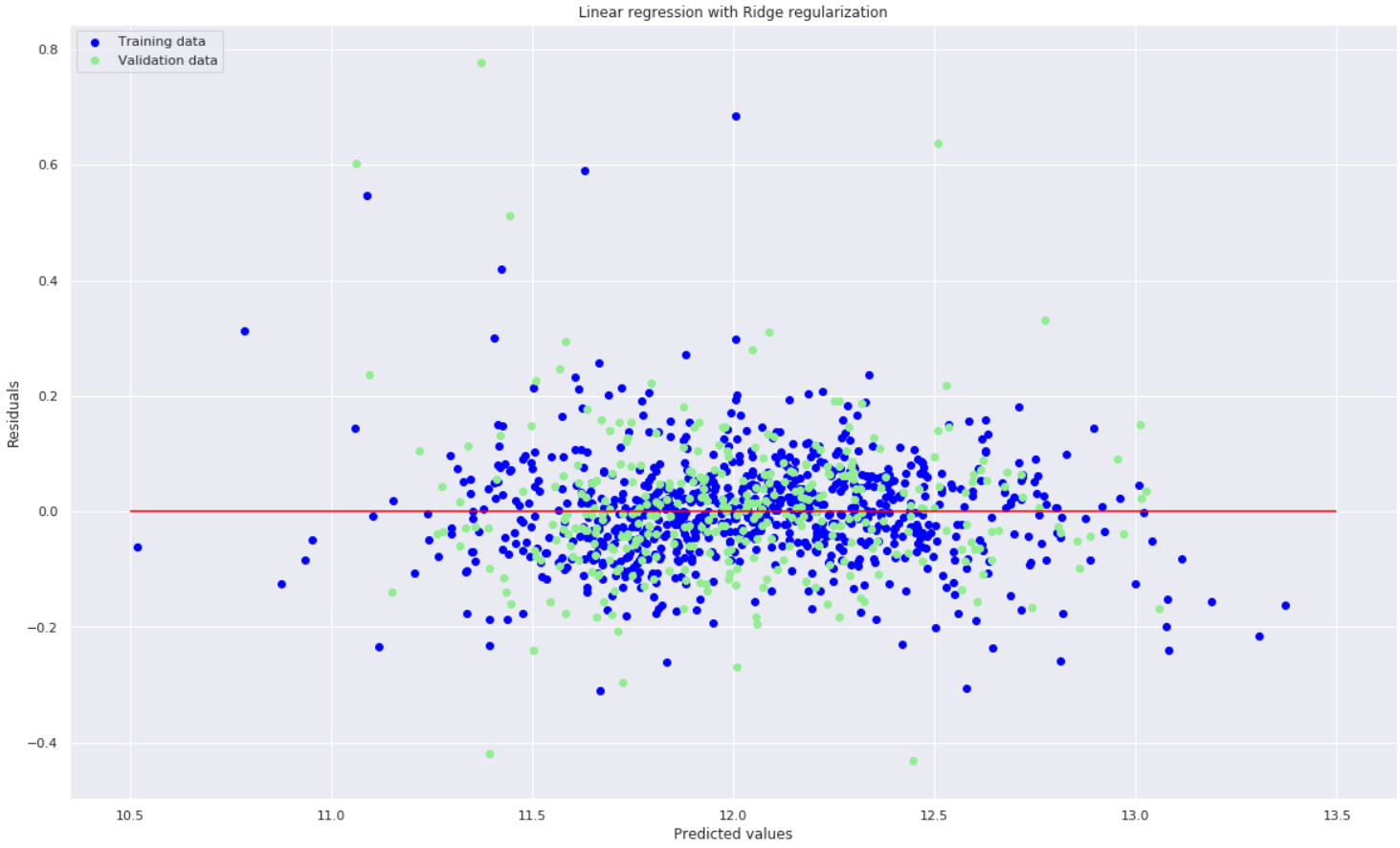
Analysing a single model : Ridge

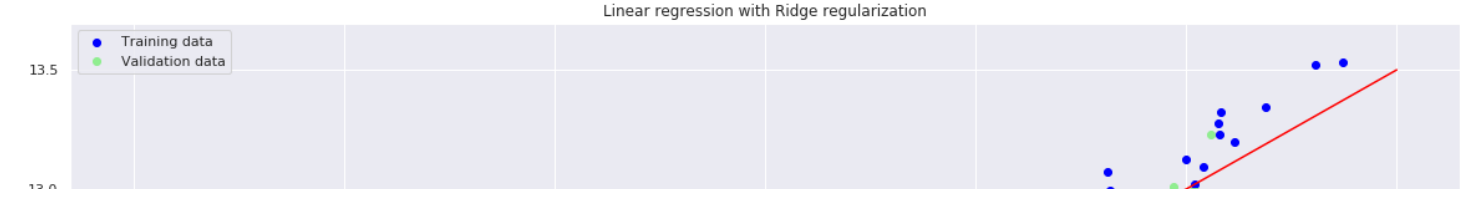
Let us focus on a single model, we took Ridge just for the example but we could have taken others. We only will try to see what this model can give us after splitting our 'X_scaled' into a new training and validation set.

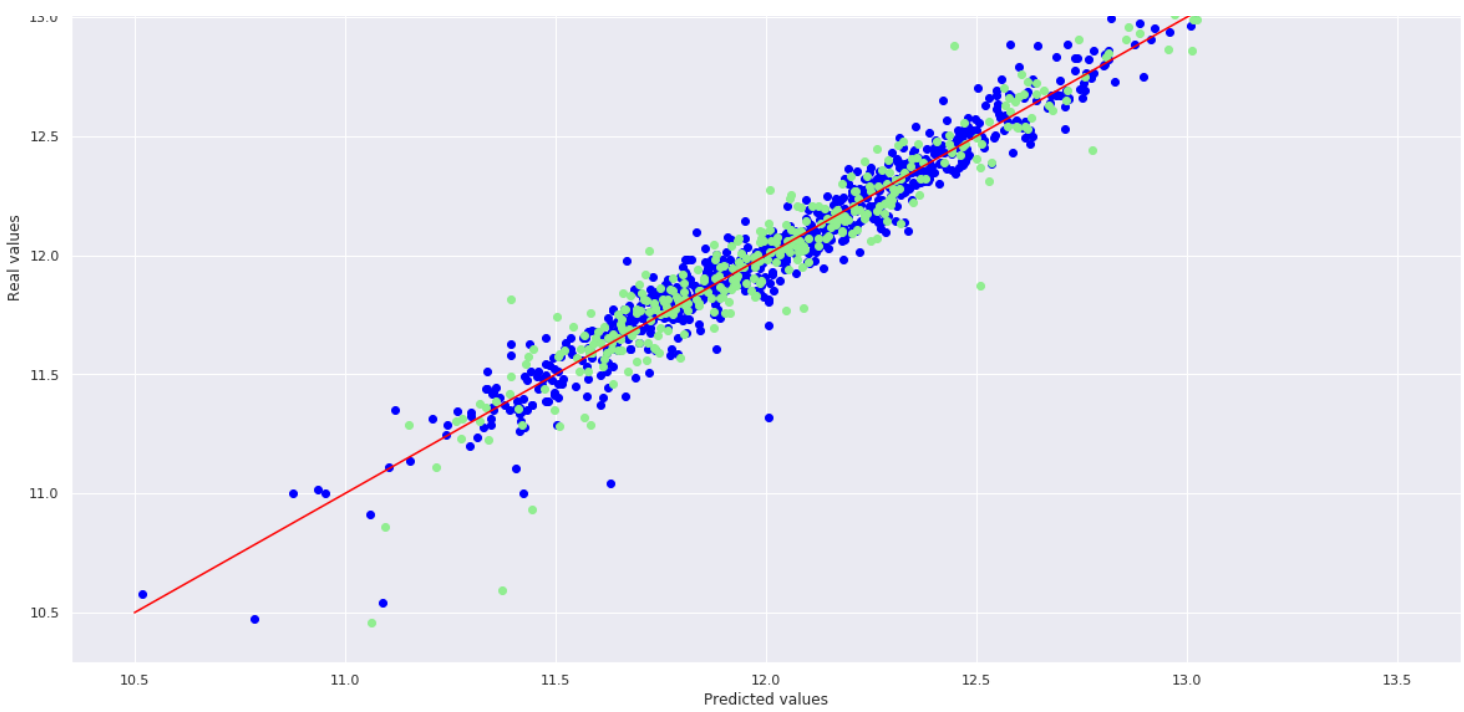
In [ ]:

```
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y_log, test_size = 0.3, rando
```

```
m_state = 0)

#kept optimal parameters we'll find later
ridge = Ridge(alpha=35)
ridge.fit(X_train, y_train)

y_train_rdg = ridge.predict(X_train)
y_test_rdg = ridge.predict(X_test)

plt.figure(figsize=(20,12))

# Plot residuals
plt.scatter(y_train_rdg, y_train_rdg - y_train, c = "blue", label = "Training data")
plt.scatter(y_test_rdg, y_test_rdg - y_test, c = "lightgreen",  label = "Validation data")
plt.title("Linear regression with Ridge regularization")
plt.xlabel("Predicted values")
plt.ylabel("Residuals")
plt.legend(loc = "upper left")
plt.hlines(y = 0, xmin = 10.5, xmax = 13.5, color = "red")
plt.show()

# Plot predictions
plt.figure(figsize=(20,12))
plt.scatter(y_train_rdg, y_train, c = "blue", label = "Training data")
plt.scatter(y_test_rdg, y_test, c = "lightgreen", label = "Validation data")
plt.title("Linear regression with Ridge regularization")
plt.xlabel("Predicted values")
plt.ylabel("Real values")
plt.legend(loc = "upper left")
plt.plot([10.5, 13.5], [10.5, 13.5], c = "red")
plt.show()
```







The regression does not seem bad when looking at this residual plot and prediction plot, most of the points are aligned.

Ensembles

> **Having trouble picking the best model to use? Often you can combine the outputs from the different models and get better accuracy. There are two steps for every one of these algorithms.**
>
> - **Producing a distribution of simple ML models on subsets of the original data**
> - **Combining the distribution into one "Aggregated" model**
>
> **Bagging (Combined Models/Views) : In this method, you train a few different models, which are different in some way, on the same data and you average out the outputs to create the final output. Bagging has the effect of reducing variance in the model. You can intuitively think of it as having multiple people with different backgrounds thinking about the same problem but with different starting positions. Just as on a team this can be a potent tool for getting the right answer.**
> **Stacking : Its similar to bagging the difference here is that you don't have an empirical formula for your combined output. You create a meta-level learner that based on the input data chooses how to weight the answers from your different models to produce the final output.**

https://towardsdatascience.com/deep-learning-performance-cheat-sheet-21374b9c4f45

> **Benefits :**
>
> - **The resulting committee is almost always less complex than a single network which would achieve the same level of performance**
> - **The resulting committee can be trained more easily on smaller input sets**
> - **The resulting committee often has improved performance over any single network**
> - **The risk of overfitting is lessened, as there are fewer parameters (weights) which need to be set**

https://en.wikipedia.org/wiki/Ensemble_averaging_(machine_learning)

Bagging

In [ ]:

```
class AverageWeight(BaseEstimator, RegressorMixin):
    def __init__(self,mod,weight):
        self.mod = mod
        self.weight = weight

    def fit(self,X,y):
        self.models_ = [clone(x) for x in self.mod]
        for model in self.models_:
            model.fit(X,y)  #train in model separately
        return self

    def predict(self,X):
        w = list()
        pred = np.array([model.predict(X) for model in self.models_])
        # for every data point, single model prediction times weight, then add them togethe
r
        for data in range(pred.shape[1]):
            single = [pred[model,data]*weight for model,weight in zip(range(pred.shape[0]),
self.weight)]
            w.append(np.sum(single))
        return w
```

In [ ]:

```
#keeping the best parameters given by the previous section
lasso = Lasso(alpha=0.0005,max_iter=10000)
ridge = Ridge(alpha=35)
svr = SVR(gamma= 0.0005,C=15,epsilon=0.01)
ker = KernelRidge(alpha=0.2 ,kernel='polynomial',degree=3 , coef0=1.0)
ela = ElasticNet(alpha=0.008,l1_ratio=0.05,max_iter=10000)
bay = BayesianRidge(alpha_2=1, lambda_1=1, n_iter=10000)
```

For the next part of code, we did several tries, every combination tested is not visible in the dictionary below, we know from previous test that this averaged model is better without Lasso and BayesianRidge for example, and we also know that giving greater importance to SVR is best.

In [ ]:

```
from sklearn.model_selection import ParameterGrid

# Define the hyperparameters
hyperparameters = {
    'w1': [0, 0.1],
    'w2': [0.1, 0.2],
    'w3': [0.4, 0.5, 0.6],
    'w4': [0.2, 0.3, 0.4],
    'w5': [0, 0.1],
    'w6': [0, 0.1],
    }


# Compute all combinations
parameter_grid = list(ParameterGrid(hyperparameters))
resultsDF = pd.DataFrame(columns=['w1', 'w2', 'w3', 'w4', 'w5', 'w6'])
for p in parameter_grid:
    weight_avg = AverageWeight(mod = [lasso,ridge,svr,ker,ela,bay],weight=[p['w1'],p['w2'],
p['w3'],p['w4'],p['w5'],p['w6']])
    rmse = rmse_cv(weight_avg,X_scaled,y_log)
    p['mean_loss']=rmse.mean()
    resultsDF = resultsDF.append(p, ignore_index=True)

display(resultsDF.sort_values(by='mean_loss', ascending=True).head())
```

	w1	w2	w3	w4	w5	w6	mean_loss
	w1	w2	w3	w4	w5	w6	mean_loss
20	0.0	0.1	0.5	0.4	0.0	0.0	0.107674
28	0.0	0.1	0.6	0.3	0.0	0.0	0.107778
80	0.1	0.1	0.4	0.4	0.0	0.0	0.107785
44	0.0	0.2	0.4	0.4	0.0	0.0	0.107797
52	0.0	0.2	0.5	0.3	0.0	0.0	0.107801

In [ ]:

```
# assign arbitrary weights based on their gridsearch score
# better score = better weight
w1 = 0.0
w2 = 0.1
w3 = 0.5
w4 = 0.4
w5 = 0.0
w6 = 0.0
```

In [ ]:

```
#testing
weight_avg = AverageWeight(mod = [lasso,ridge,svr,ker,ela,bay],weight=[w1,w2,w3,w4,w5,w6])
rmse = rmse_cv(weight_avg,X_scaled,y_log)
print(rmse.mean())
```

```
0.10767423253073108
```

The averaging model is indeed better than any of the model taken separately !

Stacking

> **In stacking regressor we combine different models and use the predicted values in the training set to make further predictions. In a nutshell stacking uses as a first-level (base), the predictions of a few basic classifiers and then uses another model at the second-level to predict the output from the earlier first-level predictions. Having now obtained our first-level predictions, we are therefore having as our new columns the first-level predictions from our earlier classifiers and we train the next classifier on this.**
>
> **All three so-called "meta-algorithms" combine several machine learning techniques into one predictive model in order to decrease the variance (bagging), bias (boosting) or improving the predictive force (stacking alias ensemble). Thus we are expecting to find the best prediction with stacking.**

https://stats.stackexchange.com/questions/18891/bagging-boosting-and-stacking-in-machine-learning

In [ ]:

```
class stacking(BaseEstimator, RegressorMixin, TransformerMixin):
    def __init__(self,mod,meta_model):
        self.mod = mod
        self.meta_model = meta_model
        self.kf = KFold(n_splits=5, random_state=42, shuffle=True)

    def fit(self,X,y):
        self.saved_model = [list() for i in self.mod]
        oof_train = np.zeros((X.shape[0], len(self.mod)))  #OOF = Out Of fold (predictions)

        for i,model in enumerate(self.mod):
```

```
            for train_index, val_index in self.kf.split(X,y):
                renew_model = clone(model)
                renew_model.fit(X[train_index], y[train_index])
                self.saved_model[i].append(renew_model)
                oof_train[val_index,i] = renew_model.predict(X[val_index])

        self.meta_model.fit(oof_train,y)  #the prediction of each fold are used as data for
the new models
        return self

    def predict(self,X):
        whole_test = np.column_stack([np.column_stack(model.predict(X) for model in single_
model).mean(axis=1)
                                      for single_model in self.saved_model])
        return self.meta_model.predict(whole_test)

    def get_oof(self,X,y,test_X):
        oof = np.zeros((X.shape[0],len(self.mod)))
        test_single = np.zeros((test_X.shape[0],5))
        test_mean = np.zeros((test_X.shape[0],len(self.mod)))
        for i,model in enumerate(self.mod):
            for j, (train_index,val_index) in enumerate(self.kf.split(X,y)):
                clone_model = clone(model)
                clone_model.fit(X[train_index],y[train_index])
                oof[val_index,i] = clone_model.predict(X[val_index])
                test_single[:,j] = clone_model.predict(test_X)
            test_mean[:,i] = test_single.mean(axis=1)
        return oof, test_mean
```

In [ ]:

```
# must do imputer first, otherwise stacking won't work : Imputation transformer for completi
ng missing values but in theory there is not
X_train_imputer = Imputer().fit_transform(X_scaled)
y_train_imputer = Imputer().fit_transform(y_log.values.reshape(-1,1)).ravel()
# -1 unknown dimension that numpy will figure out so that the new shape is compatible with t
he old one
# ravel() returns a contiguous flattened array.
```

For the models in the stacking, we kept the six ones we optimised previously. Adding the average model from the previous part only increased the computation time without really changing the prediction. The order in the stacking influences the performance of the stack model. Similarly, if we try to add more than one time a same model (like three times each model) in the stack the loss changes and not always in good, but it increases the computation time. We try different order and some other combinations to see if we could improve the result but at the end we kept a simple combination of the six models. For the meta_model it seems obvious that it has to be the best of our previous model, therefore KernelRidge (we test some of the others and it increased the loss, also testing with the average model gave poor result 0.104 instead of 0.099 even though it is a better model than KernelRidge).

In [ ]:

```
stack_model = stacking(mod=[lasso,ridge,svr,ker,ela,bay],meta_model=ker)

#Score before adding new features
rmse = rmse_cv(stack_model,X_train_imputer, y_train_imputer)
print(rmse.mean())
```

```
0.10713918845849539
```

In [ ]:

```
#get the new features generated by stacking
X_train_stack, X_test_stack = stack_model.get_oof(X_train_imputer,y_train_imputer,test_X_sc
aled)
print(X_train_stack.shape, X_train_imputer.shape)
```

```
#add new features to both training and testing data
X_train_add = np.hstack((X_train_imputer,X_train_stack))
X_test_add = np.hstack((test_X_scaled,X_test_stack))
print(X_train_add.shape, X_test_add.shape)
```

```
(1199, 6) (1199, 410)
(1199, 416) (260, 416)
```

In [ ]:

```
#calcul the new score (rmse)
rmse = rmse_cv(stack_model,X_train_add,y_train_imputer)
print(rmse.mean())
```

```
0.09929331527001825
```

In [ ]:

```
#fit the final model with the final training data
stack_model.fit(X_train_add, y_train_imputer)
y_predict = stack_model.predict(X_train_add)

#calcul R^2 (coefficient of determination) regression score function
score = r2_score(y_train_imputer, y_predict)
print(score)
```

```
0.9473544645939743
```

Finally this is our final model, it gives the best result in term of loss comparing the previous models. Also when using another scoring function, the R^2 regression score function, we obtain a score of almost 95%, which appears to be pretty good (100% indicates that the model explains all the variability of the response data around its mean) but does not mean that our model is perfect.

> **R-squared cannot determine whether the coefficient estimates and predictions are biased, which is why you must assess the residual plots. R-squared does not indicate whether a regression model is adequate. You can have a low R-squared value for a good model, or a high R-squared value for a model that does not fit the data!**

https://blog.minitab.com/blog/adventures-in-statistics-2/regression-analysis-how-do-i-interpret-r-squared-and-assess-the-goodness-of-fit

But testing both the RMSE and R^2 should be enough to say that our model is pretty good.

Submission

In [ ]:

```
pred = np.exp(stack_model.predict(X_test_add))
result=pd.DataFrame({'Id':test_id, 'SalePrice':pred})
result.to_csv("submission.csv",index=False)
```