ID A1812

Submission Date : 11/06/2019

House Prices: Advanced Regression Techniques

Abstract:

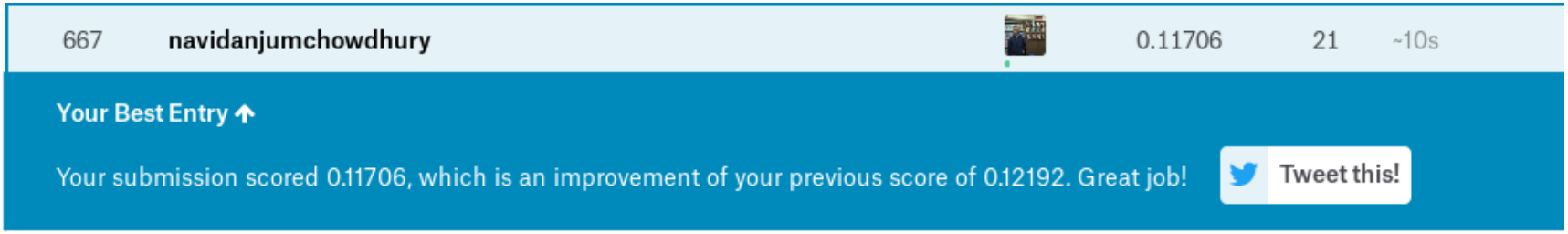
House Price prediction is a very popular dataset for data science competition. In this dataset 79 explanatory variables describing (almost) every aspect of residential homes in Ames and Iowa. This competition challenges competitor to predict the final price of each home.

In this report my main focus is how artificial neural network performs for this kind of problems and how to improve performance of the prediction using artificial neural network. So my elaboration on that section will be much more detailed.I have divided my work in four part and they are

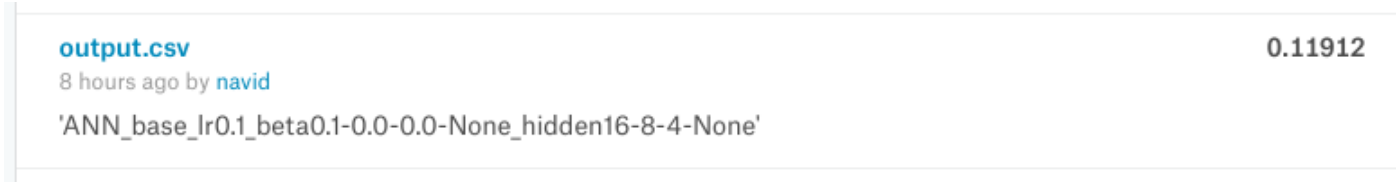
- **Data processing** where I have visualized, cleaned, handled missing data, carefully modified , removed and merged some features.
- **Testing multiple model** In this part I have used gradient boosting, decision tree, random forest regression , lasso and Artificial neural network on my pre processed data.
- **Artificial neural network implementation** In this section I have implemented ANN , performed parameter tuning, training, used grid search inside training and validate test score.
- **Cross Validation** In this part I have used k fold cross validation on my artificial neural network model to make sure if the Data is actually independent and to fine tune few parameters on whole dataset if the cross validation score is not same as validation score.
- **Ensemble learning** I have used bagging method for this section to improve my kaggle score.

Score:

Best Score : 0.11706 (using custom Stacked model)

- 

Best score without Ensemble : 0.11912 (ANN only)

- 

Imports:

In the following section, I have imported all the necessary libraries that I will need to properly complete the assignment.

- The 'Pandas' library will be used to store the 'Train' and 'Test' datasets. The particular data storing format is called a 'Dataframe'.
- The 'Numpy' library is used to make mathematical calculations easier and faster to do.
- 'Matplotlib' and 'Seaborn' are used to plot graphs
- From the 'Scikit learn'(sklearn) library, I have imported some data processing methods, some evaluation metrics and some predictive models.

Gpu testing

In [1]:
```
import tensorflow as tf
device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
  raise SystemError('GPU device not found')
print('Found GPU at: {}'.format(device_name))
```

```
Found GPU at: /device:GPU:0
```

In [2]:
```
import tensorflow as tf
import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestRegressor
from sklearn.impute import SimpleImputer
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.metrics import  r2_score
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import StandardScaler
from IPython.display import Image
from sklearn.preprocessing import normalize,MinMaxScaler
import matplotlib.pyplot as plt
from sklearn.utils import shuffle
import seaborn as sns
# %matplotlib widget
%matplotlib inline
```

Data Pre-processing

Load Data

The following block of code reads the two CSV (Comma Separated Values) files and then stores the data inside them in two separate Dataframes named 'train' and 'test'.

In [3]:
```
train = pd.read_csv('train.csv')#.select_dtypes(exclude=['object'])
test = pd.read_csv('test.csv')#.select_dtypes(exclude=['object'])

#look into datatypes of the file
print("data types count")
train.dtypes.groupby(train.dtypes).count()
```

```
data types count
```

Out[3]:
```
int64      35
float64     3
object     43
dtype: int64
```

Looking into data

Here I am printing the first five entries in the train dataset to look into the actual data that I will be working with. I gives me some insight about the data I am working with.

In [4]:
```
print('show sample')
pd.set_option('display.max_column', None)
train.head()
```

```
show sample
```

Out[4]:

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Utilities	LotConfig	LandSlope	Neighborhood	Condition1	Condition2	BldgType	H
0	1	60	RL	65.0	8450	Pave	NaN	Reg	Lvl	AllPub	Inside	Gtl	CollgCr	Norm	Norm	1Fam	
1	2	20	RL	80.0	9600	Pave	NaN	Reg	Lvl	AllPub	FR2	Gtl	Veenker	Feedr	Norm	1Fam	
2	3	60	RL	68.0	11250	Pave	NaN	IR1	Lvl	AllPub	Inside	Gtl	CollgCr	Norm	Norm	1Fam	
3	4	70	RL	60.0	9550	Pave	NaN	IR1	Lvl	AllPub	Corner	Gtl	Crawfor	Norm	Norm	1Fam	
4	5	60	RL	84.0	14260	Pave	NaN	IR1	Lvl	AllPub	FR2	Gtl	NoRidge	Norm	Norm	1Fam	

Here I have used one of the built-in functions of pandas dataframe to display descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding NaN values.

In [5]:
```
print('description of data')
train.describe()
```

```
description of data
```

Out[5]:

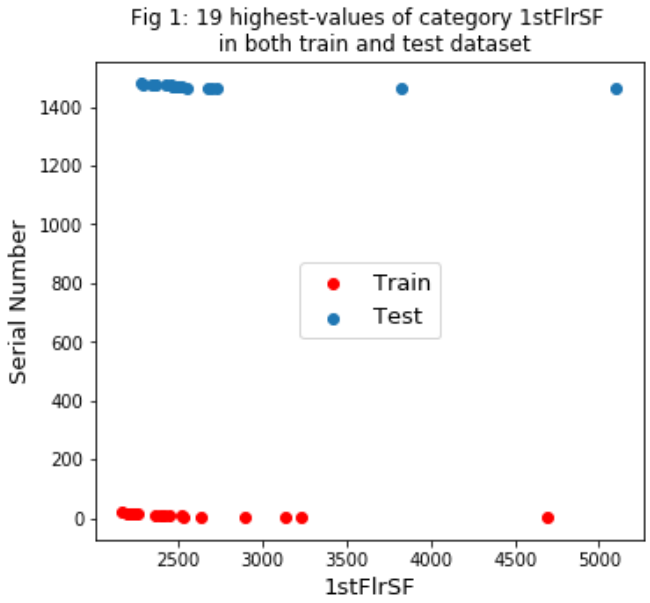
	Id	MSSubClass	LotFrontage	LotArea	OverallQual	OverallCond	YearBuilt	YearRemodAdd	MasVnrArea	BsmtFinSF1	BsmtFinSF2	BsmtUnfSF	Tota
count	1460.000000	1460.000000	1201.000000	1460.000000	1460.000000	1460.000000	1460.000000	1460.000000	1452.000000	1460.000000	1460.000000	1460.000000	146
mean	730.500000	56.897260	70.049958	10516.828082	6.099315	5.575342	1971.267808	1984.865753	103.685262	443.639726	46.549315	567.240411	105
std	421.610009	42.300571	24.284752	9981.264932	1.382997	1.112799	30.202904	20.645407	181.066207	456.098091	161.319273	441.866955	43
min	1.000000	20.000000	21.000000	1300.000000	1.000000	1.000000	1872.000000	1950.000000	0.000000	0.000000	0.000000	0.000000	
25%	365.750000	20.000000	59.000000	7553.500000	5.000000	5.000000	1954.000000	1967.000000	0.000000	0.000000	0.000000	223.000000	79
50%	730.500000	50.000000	69.000000	9478.500000	6.000000	5.000000	1973.000000	1994.000000	0.000000	383.500000	0.000000	477.500000	99
75%	1095.250000	70.000000	80.000000	11601.500000	7.000000	6.000000	2000.000000	2004.000000	166.000000	712.250000	0.000000	808.000000	129
max	1460.000000	190.000000	313.000000	215245.000000	10.000000	9.000000	2010.000000	2010.000000	1600.000000	5644.000000	1474.000000	2336.000000	611

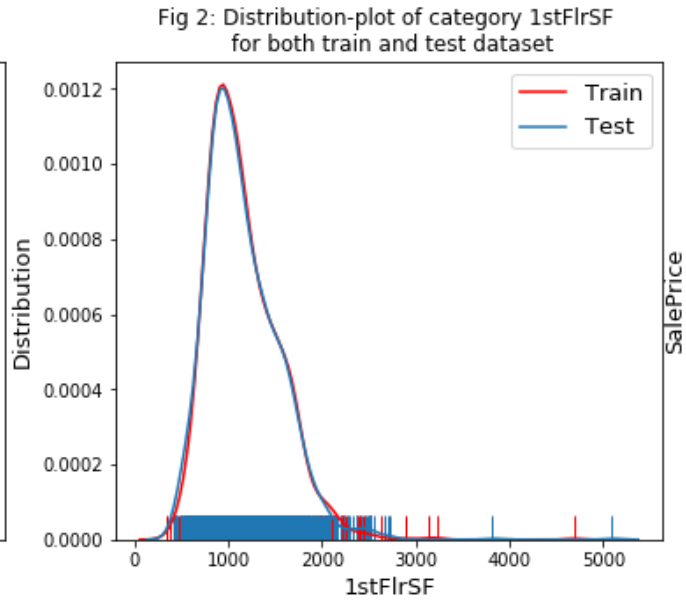
This function shows scatter-plot and distribution plot. I am going to use it to see few of the features of the dataset and observe how it changes while I process the data. I will try not to remove data so instead of removing any data point I will observe them until all my data processing is complete. If I found out after all the processing some data points are really causing problem then I will drop it.

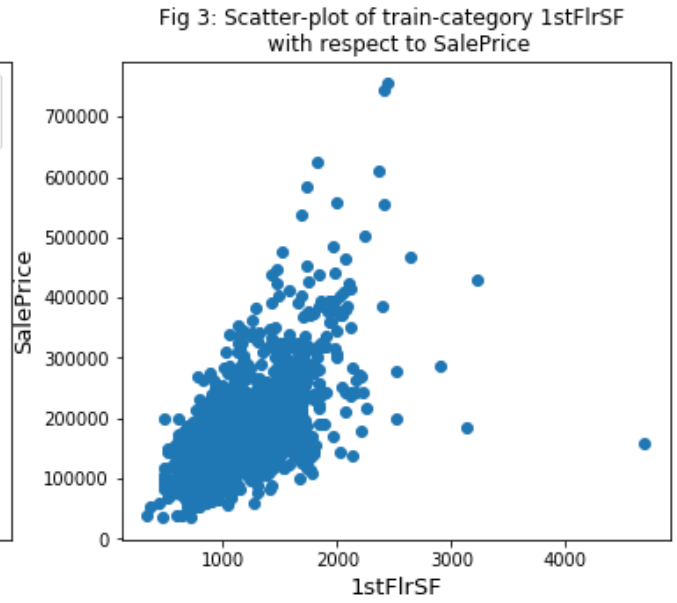
In [6]:
```
#For showing diffrence
old_train_outlier_flag =train.copy()
old_test_outlier_flag =test.copy()
old_target_outlier_flag =train.SalePrice.copy()

# A FUNCTION THAT SHOWS SCATTER-PLOT AND DISTRIBUTION-PLOT
def outlier_check_plot(column, train_data_flag=train , test_data_flag=test , target=train.SalePrice ):
    plt.subplots(figsize=(19, 5))
    # SCATTER PLOT OF THE 19 HIGHEST-VALUES OF A COLUMN
    plt.subplot(1, 3, 1)
    plt.scatter(x = train_data_flag[column].sort_values(ascending=False)[:19], y = train.Id[:19], color='red', label='Train' )
    plt.scatter(x = test_data_flag[column].sort_values(ascending=False)[:19], y = test.Id[:19], label='Test')
    plt.ylabel('Serial Number', fontsize=13)
    plt.xlabel(column, fontsize=13)
    plt.title('Fig 1: 19 highest-values of category {} \n in both train and test dataset'.format(column))
    plt.legend(loc='center',fontsize=13)
    # DISTRIBUTION-PLOT OF THE COLUMN
    plt.subplot(1, 3, 2)
    sns.distplot(train_data_flag[column],color='red', rug=True, hist=False, label='Train')
    sns.distplot(test_data_flag[column], rug=True, hist=False, label='Test')
    plt.ylabel('Distribution', fontsize=13)
    plt.xlabel(column, fontsize=13)
    plt.title('Fig 2: Distribution-plot of category {} \n for both train and test dataset'.format(column))
    plt.legend(fontsize=13)
    # SCATTER-PLOT OF THE COLUMN WITH RESPECT TO SALEPRICE
    plt.subplot(1, 3, 3)
    plt.scatter(x = train_data_flag[column], y = target)
    plt.ylabel('SalePrice', fontsize=13)
    plt.xlabel(column, fontsize=13)
    plt.title('Fig 3: Scatter-plot of train-category {} \n with respect to SalePrice'.format(column))
    plt.show()
```

In [7]:
```
print('Before outlier-removal of 1stFlrSF: ')
outlier_check_plot('1stFlrSF')
```

```
Before outlier-removal of 1stFlrSF:
```







We can see one value in train set that is highly contradictory with SalePrice (1stFlrSF is too high but SalePrice is too low). And there is only one such high-value point available in test dataset. So we might want to remove this outlier.

In [8]:
```
print('Before outlier-removal of BsmtFinSF1: ')
outlier_check_plot('BsmtFinSF1')
```

```
Before outlier-removal of BsmtFinSF1:

/home/navid/anaconda3/envs/tf/lib/python3.6/site-packages/statsmodels/nonparametric/kde.py:448: RuntimeWarning: invalid value enco
untered in greater
  X = X[np.logical_and(X > clip[0], X < clip[1])] # won't work for two columns.
/home/navid/anaconda3/envs/tf/lib/python3.6/site-packages/statsmodels/nonparametric/kde.py:448: RuntimeWarning: invalid value enco
untered in less
  X = X[np.logical_and(X > clip[0], X < clip[1])] # won't work for two columns.
```

Fig 1: 19 highest-values of category BsmtFinSF1 in both train and test dataset

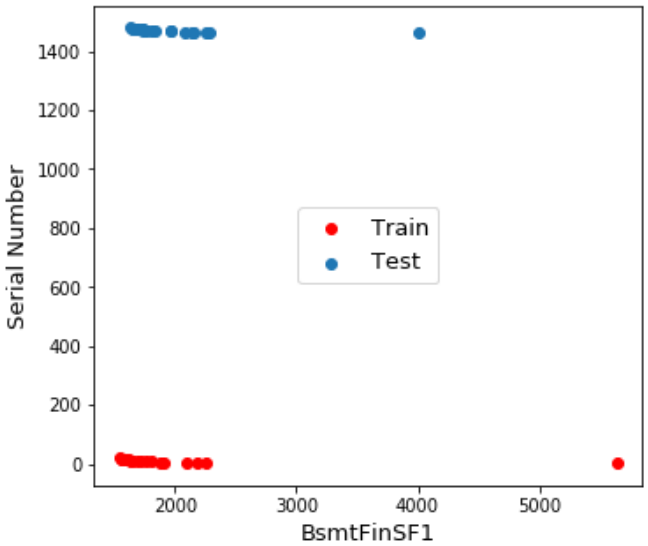


Fig 2: Distribution-plot of category BsmtFinSF1 for both train and test dataset

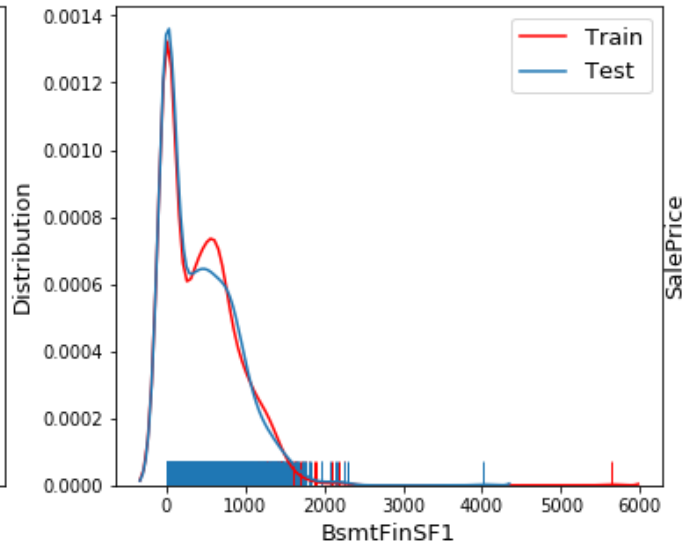
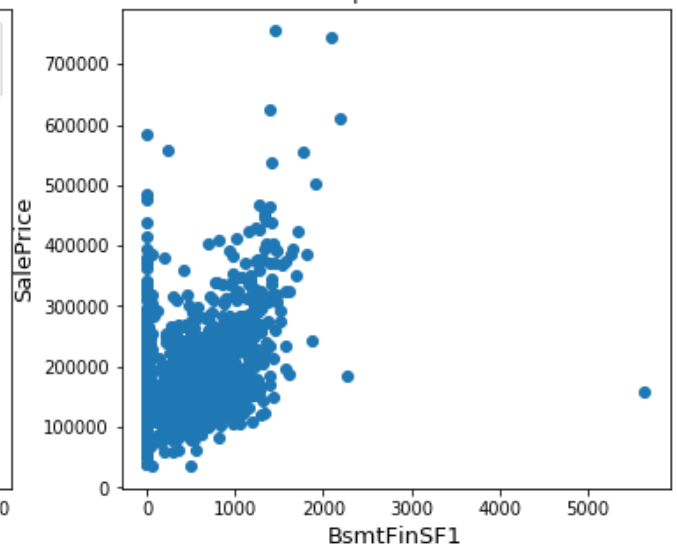


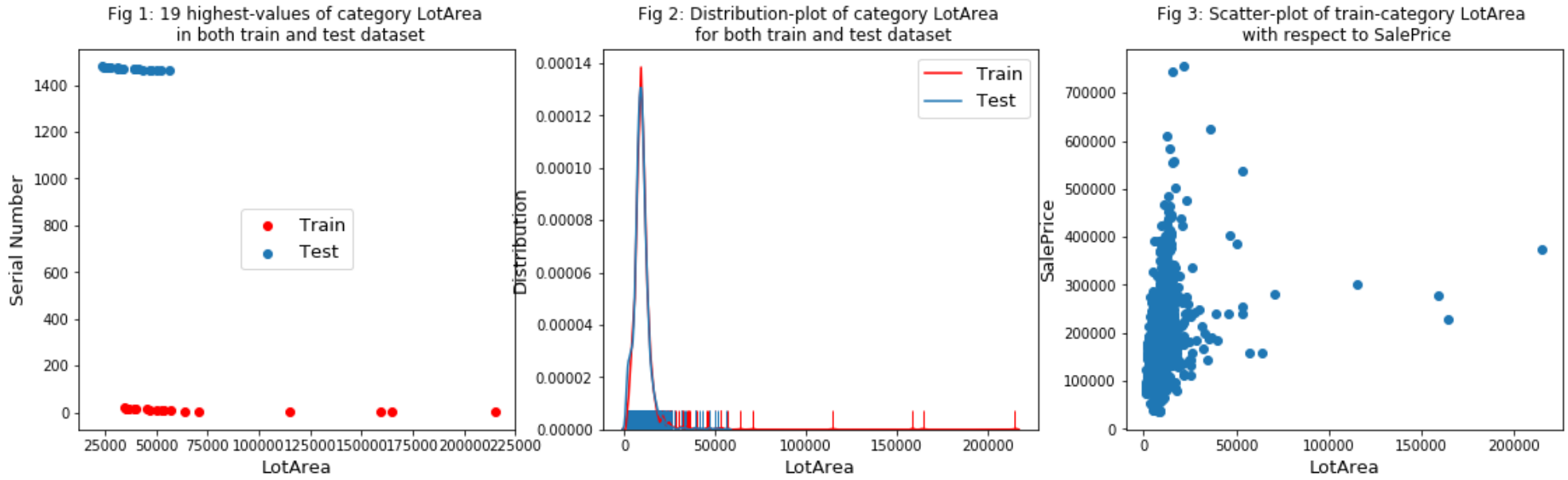
Fig 3: Scatter-plot of train-category BsmtFinSF1 with respect to SalePrice



We can also see the same outlier here.

In [9]:
```
print('Before outlier-removal of LotArea: ')
outlier_check_plot('LotArea')
```

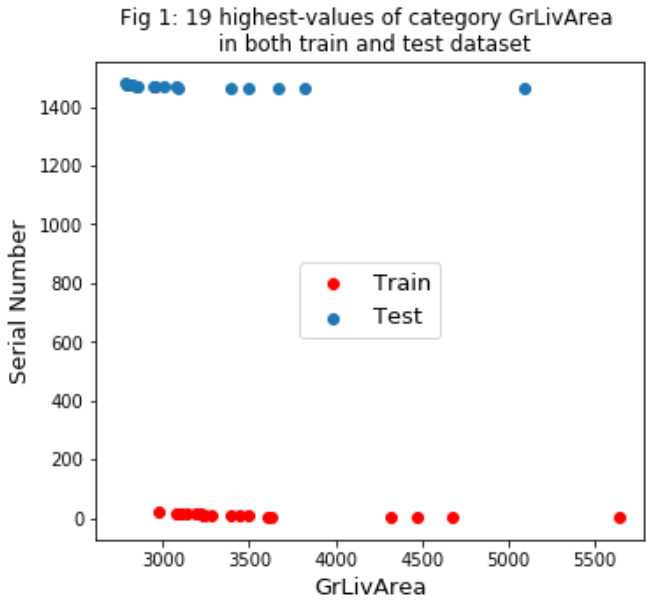
```
Before outlier-removal of LotArea:
```

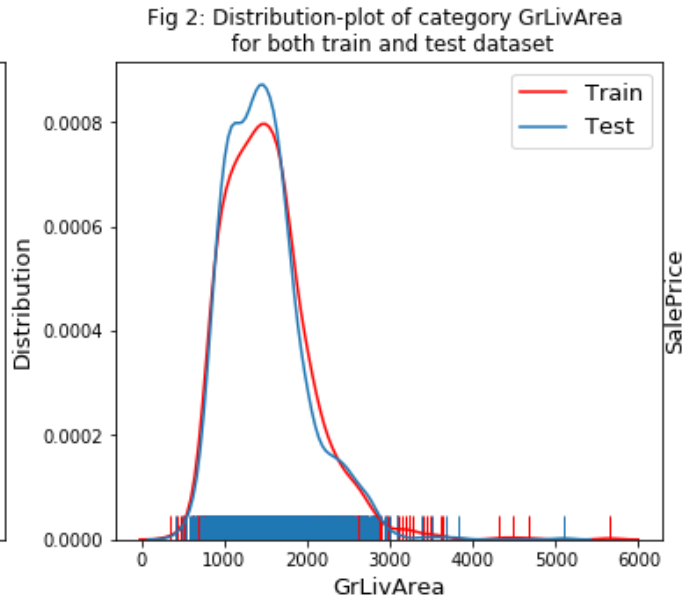


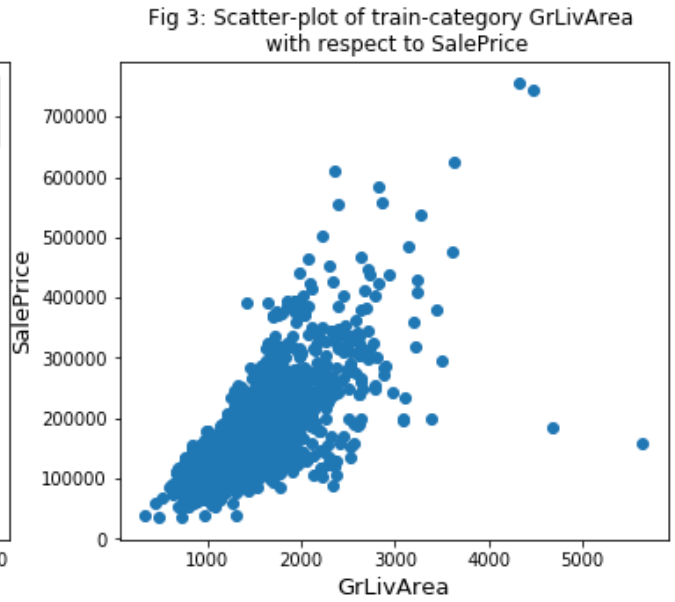
We can see in Fig 3 that there are 4 LotArea train-samples above 80000 that are very high in size but comperatively very low in SalePrice. Also there are no such values present in test-data: Fig 1. So we can drop them

In [10]:
```
print('Before outlier-removal of GrLivArea: ')
outlier_check_plot('GrLivArea')
```

```
Before outlier-removal of GrLivArea:
```



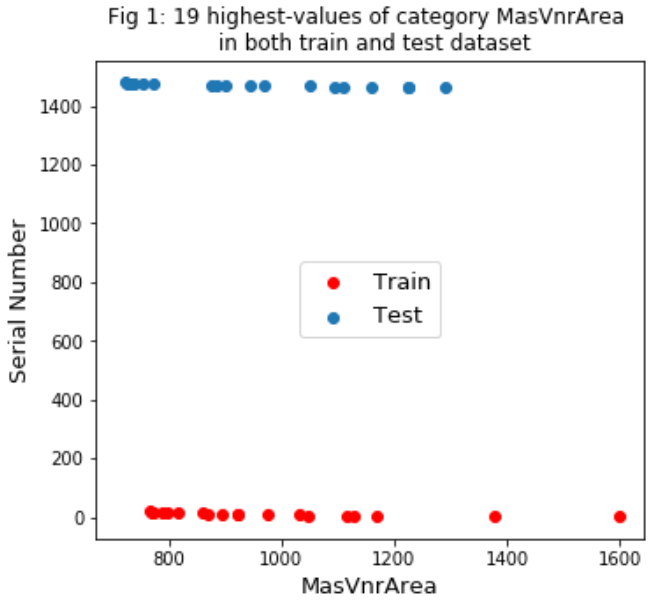


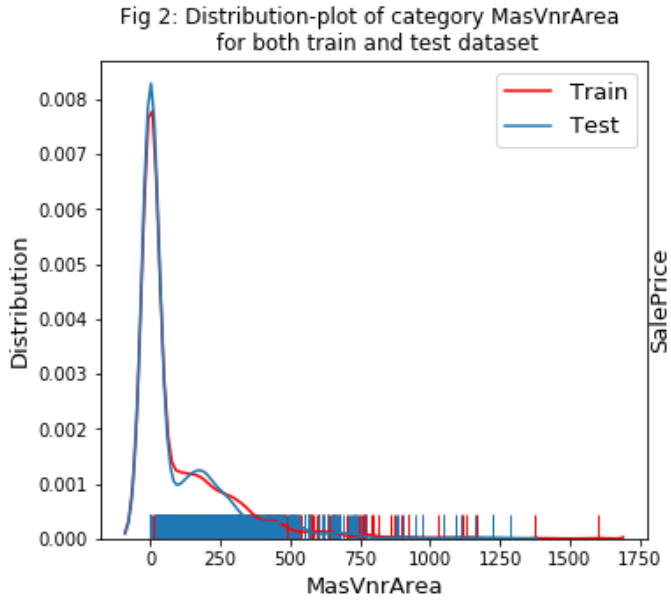


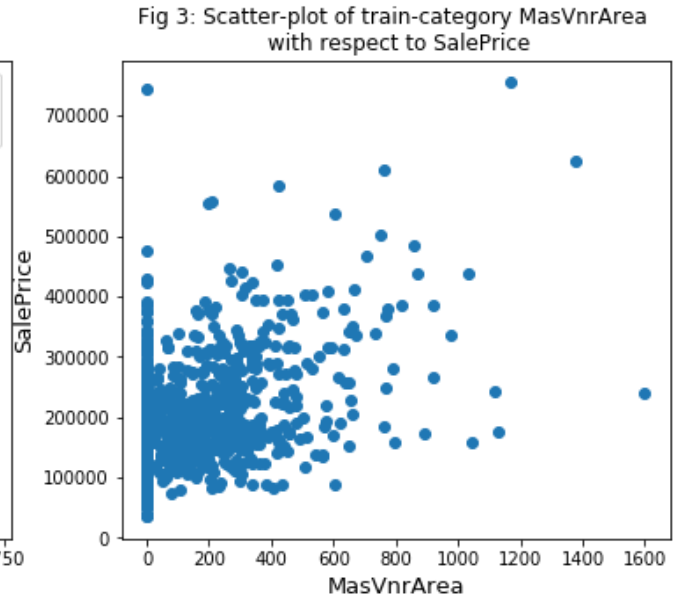
If we compare Fig. 3 with code-cell 13 we can see that two outliers are already common in GrLivArea. These two outliers of GrLivArea train-samples were above 4000 with very low SalePrice (below 300000). We are seeing same outlier again and again.

In [11]:
```
print('Before outlier-removal of MasVnrArea: ')
outlier_check_plot('MasVnrArea')
```

```
Before outlier-removal of MasVnrArea:
```



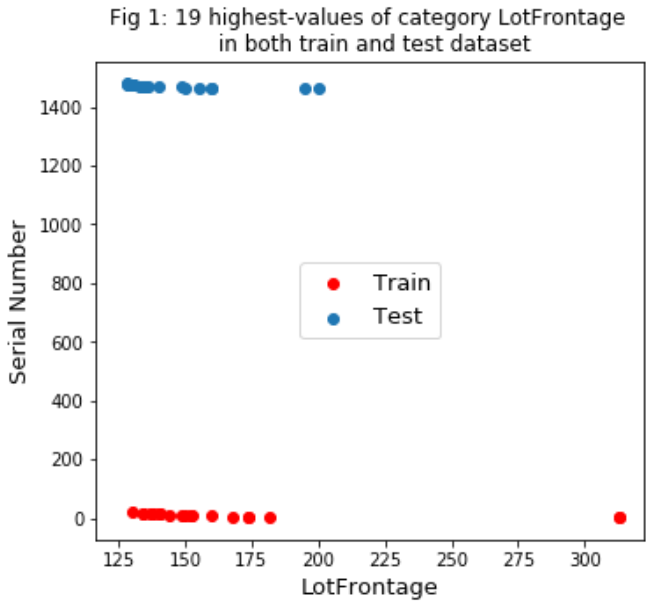


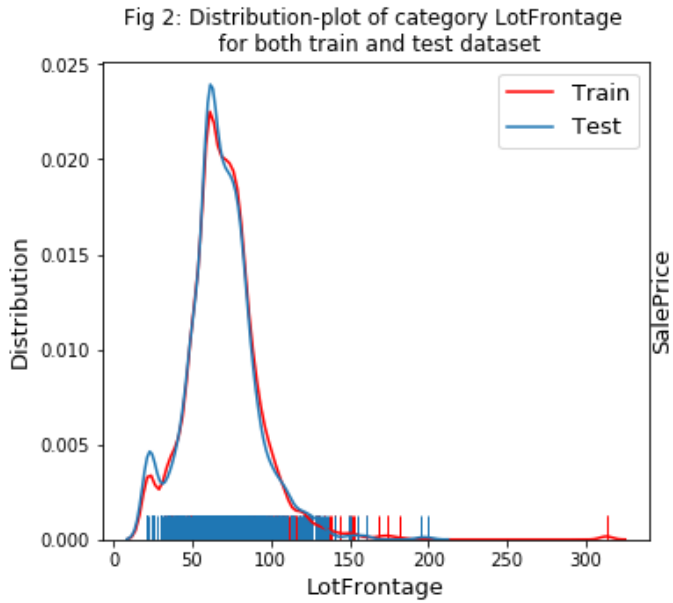


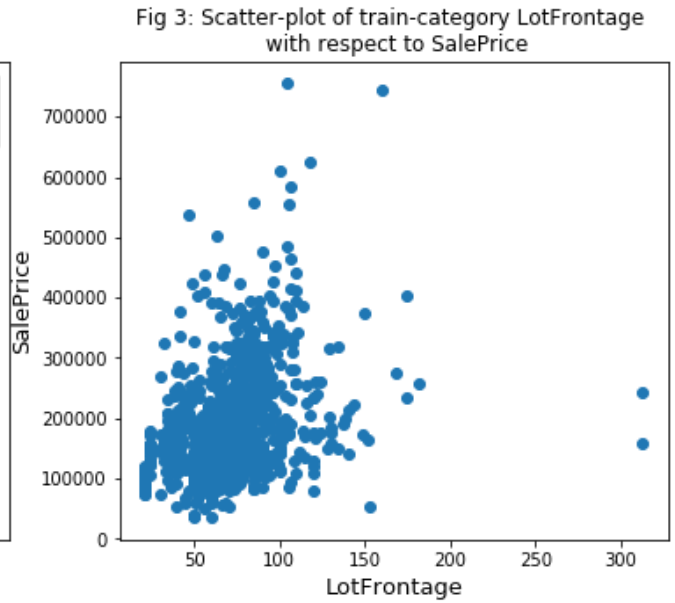
As we can see in Fig 3 that above 1500 there is 1 MasVnrArea train-samples that are very high in size but comperatively very low in SalePrice (below 300000) and there is no such values present in test-data: Fig 1. But this case is not so common outlier in other sections so keeping it would be safe for now.

In [12]:
```
print('Before outlier-removal of LotFrontage: ')
outlier_check_plot('LotFrontage')
```

```
Before outlier-removal of LotFrontage:
```



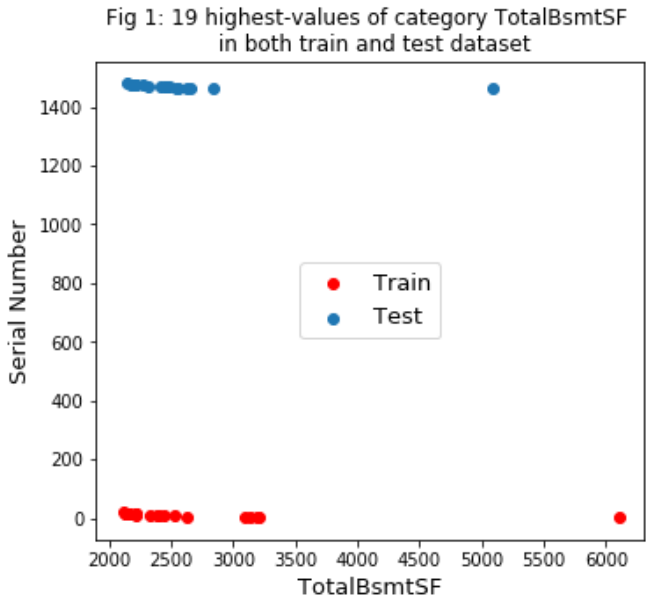


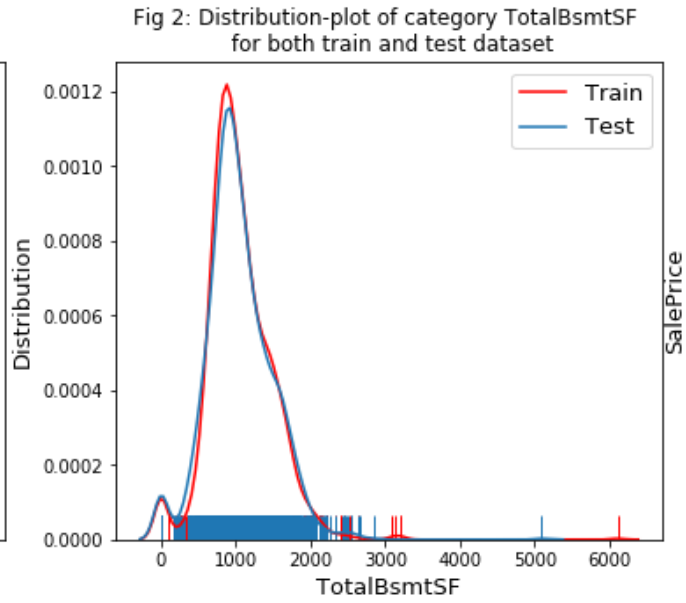


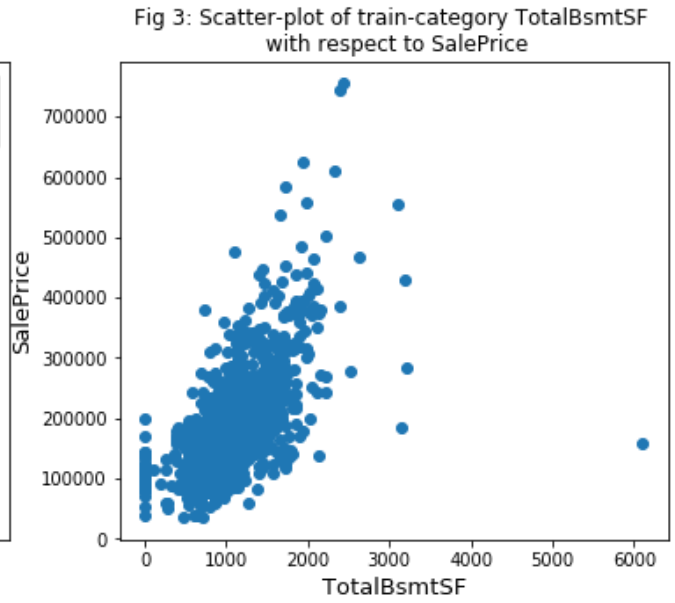
As we can see in Fig 3 that above 200 there is 1 LotFrontage train-samples that is very high in size but comperatively very low in SalePrice (below 300000) and there is no such value present in test-data. But one of them seems to be the common outlier which is below 20000(saleprice). We should remove the common one and observe the other.

In [13]:
```
print('Before outlier-removal of TotalBsmtSF: ')
outlier_check_plot('TotalBsmtSF')
```

```
Before outlier-removal of TotalBsmtSF:
```





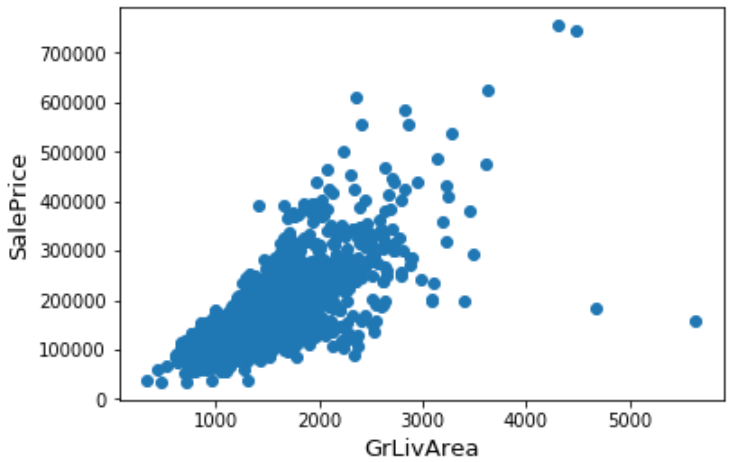


We can also see the common outlier and we would be removing the common outlier in the next section.

Common Outlier Remove

Saleprice vs GrLivArea

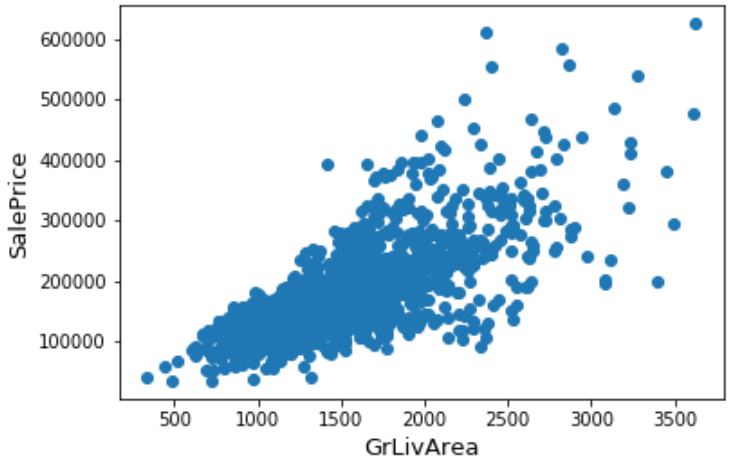
In [14]:
```
fig, ax = plt.subplots()
ax.scatter(x = train['GrLivArea'], y = train['SalePrice'])
plt.ylabel('SalePrice', fontsize=13)
plt.xlabel('GrLivArea', fontsize=13)
plt.show()
```



There are a few houses with more than 4000 sq ft living area that are outliers, so we drop them from the training data.

In [15]:
```
train.drop(train[ (train["GrLivArea"] > 4000) ].index, inplace=True)
```

In [16]:
```
#Check the graph again
fig, ax = plt.subplots()
ax.scatter(train['GrLivArea'], train['SalePrice'])
plt.ylabel('SalePrice', fontsize=13)
plt.xlabel('GrLivArea', fontsize=13)
plt.show()
```

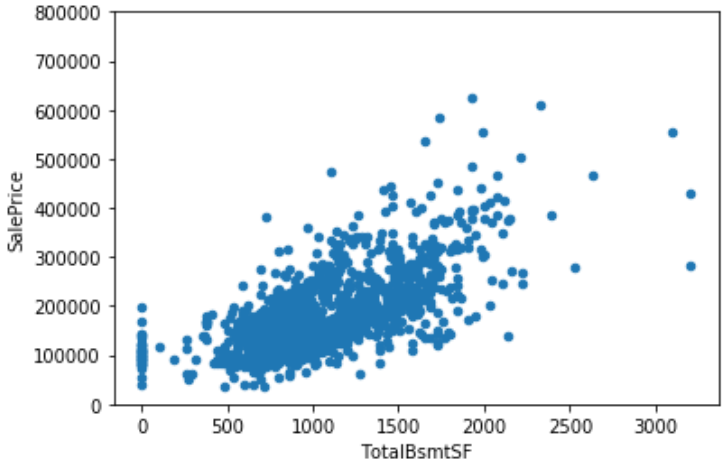


Its a linear relation so this feature is helpful to predict the price.

SalePrice vs TotalBsmSf

This relationship is also linear so we can expect that it also have great impact on the price.

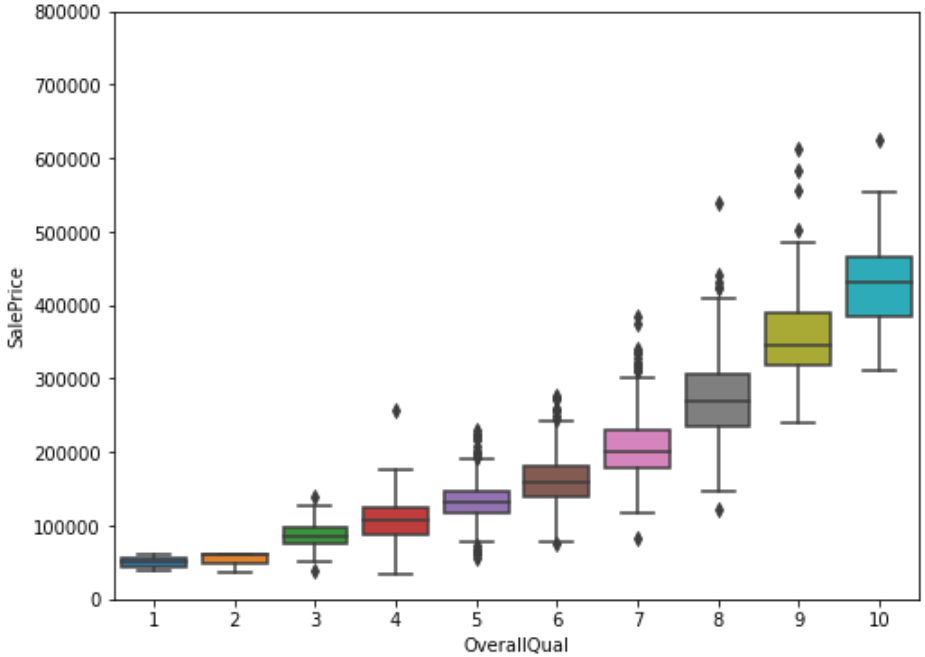
In [17]:
```
#scatter plot totalbsmtsf/saleprice
var = 'TotalBsmtSF'
data = pd.concat([train['SalePrice'], train[var]], axis=1)
data.plot.scatter(x=var, y='SalePrice', ylim=(0,800000));
```



We have removed the common outlier and now the graph seems better and we will follow up later after all the data pre processing. If any outlier remains after all processing I will remove them.

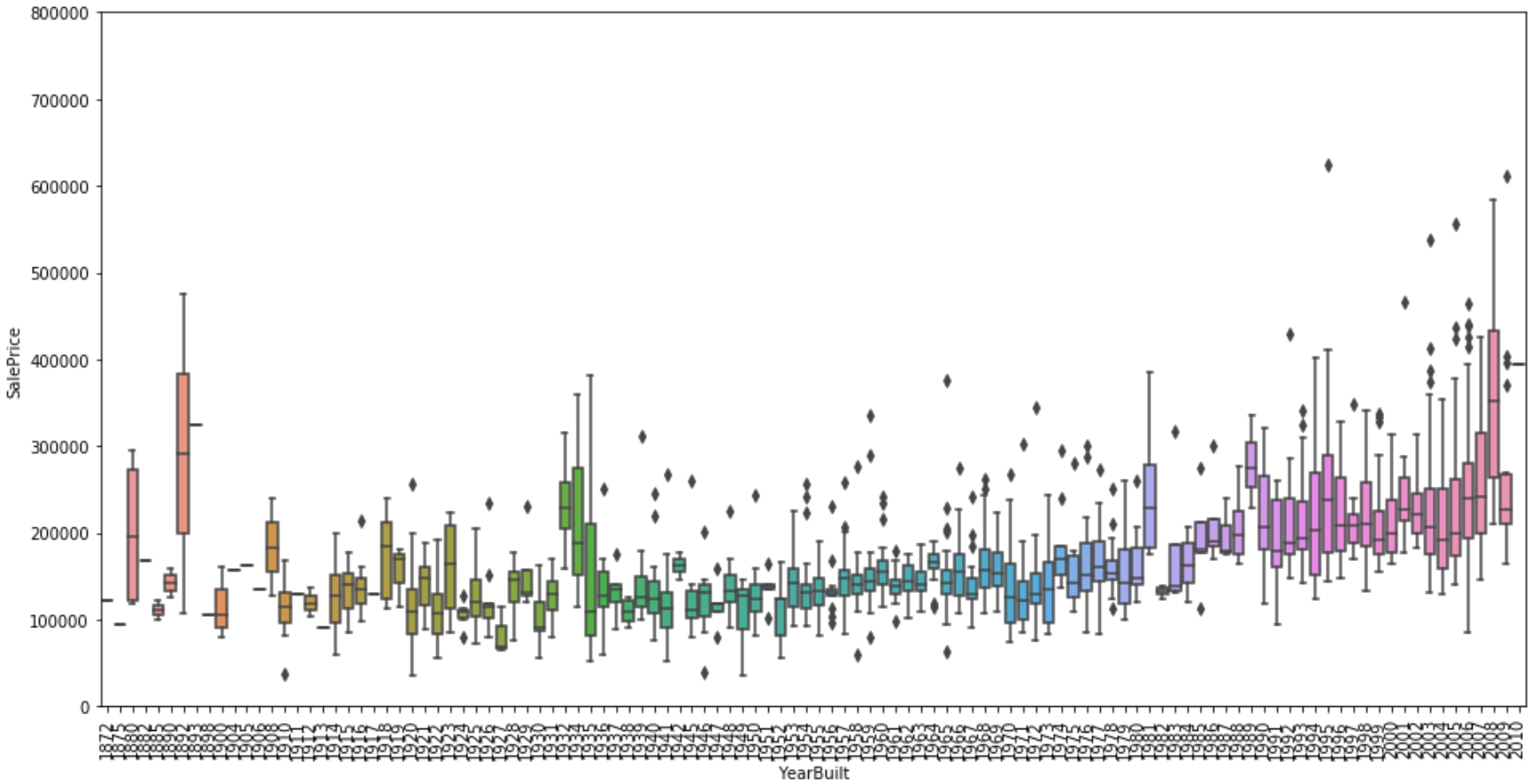
Relationship with categorical features

In [18]:
```
#box plot overallqual/saleprice
import seaborn as sns

var = 'OverallQual'
data = pd.concat([train['SalePrice'], train[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);
```



As expected saleprice increases when overall quality increases.

In [19]:
```
var = 'YearBuilt'
data = pd.concat([train['SalePrice'], train[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);
```



We can see that people tends to spend more for newly built houses. Although its does not seems really a storong feature acording to plot but its really importent if we consider other parameters too.

Note

- 'GrLivArea' and 'TotalBsmtSF' seem to be linearly related with 'SalePrice'. Both relationships are positive, which means that as one variable increases, the other also increases. In the case of 'TotalBsmtSF', we can see that the slope of the linear relationship is particularly high.
- 'OverallQual' and 'YearBuilt' also seem to be related with 'SalePrice'. The relationship seems to be stronger in the case of 'OverallQual', where the box plot shows how sales prices increase with the overall quality.

We have analised four variables, but there are many other that we should analyse. The trick here seems to be the choice of the right features (feature selection) and not the definition of complex relationships between them (feature engineering).

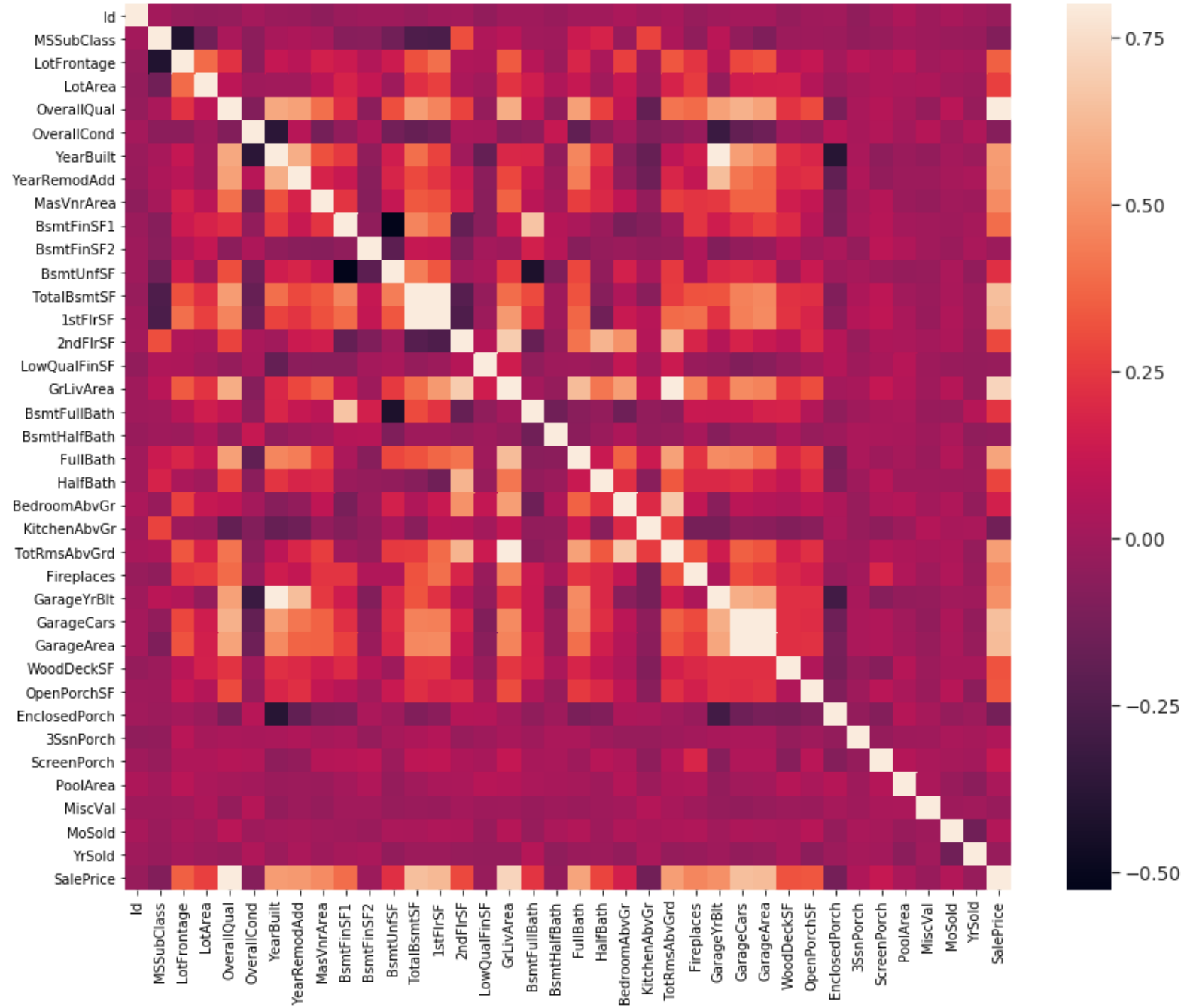
Correlation matrix (heatmap)

The correlation coefficient is a statistical calculation that is used to examine the relationship between two sets of data. The value of the correlation coefficient tells us about the strength and the nature of the relationship.

Correlation coefficient values can range between +1.00 to -1.00. If the value is exactly +1.00, it means that there is a "perfect" positive relationship between two numbers, while a value of exactly -1.00 indicates a "perfect" negative relationship.

If correlation is Positive then the values increase together and if the correlation is Negative, one value decreases as the other increases. When two sets of data are strongly linked together we say they have a High Correlation.

In [20]:
```
#correlation matrix
corrmat = train.corr()
f, ax = plt.subplots(figsize=(15, 12))
sns.set(font_scale=1.25)
sns.heatmap(corrmat, vmax=.8, square=True);
```

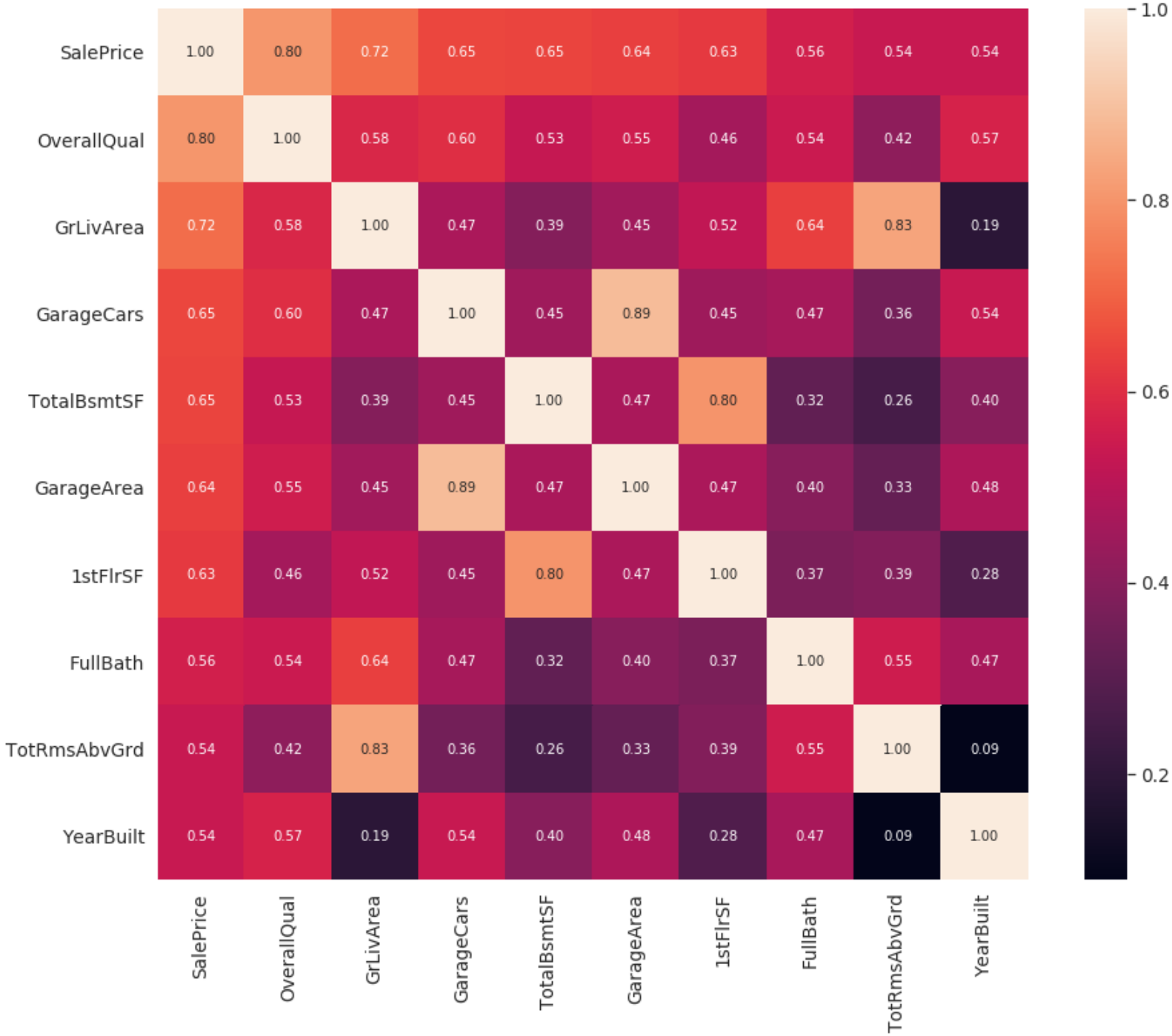


In my opinion, this heatmap is the best way to get a quick overview the relationships of a dataset.

At first sight, there are two red colored squares that get my attention. The first one refers to the 'TotalBsmtSF' and '1stFlrSF' variables, and the second one refers to the 'GarageX' variables. Both cases show how significant the correlation is between these variables. Actually, this correlation is so strong that it can indicate a situation of multicollinearity. If we think about these variables, we can conclude that they give almost the same information so multicollinearity really occurs. Heatmaps are great to detect this kind of situations and in problems dominated by feature selection, like ours, they are an essential tool.

Another thing that got my attention was the 'SalePrice' correlations. We can see our well-known 'GrLivArea', 'TotalBsmtSF', and 'OverallQual' is closely related to salePrice, but we can also see many other variables that should be taken into account. So we are zooming in.

In [21]:
```
#saleprice correlation matrix
k = 10 #number of variables for heatmap
cols = corrmat.nlargest(k, 'SalePrice')['SalePrice'].index
cm = np.corrcoef(train[cols].values.T)
f, ax = plt.subplots(figsize=(15, 12))
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=c
ols.values)
plt.show()
```


SalePrice 1.00 0.80 0.72 0.65 0.65 0.64 0.63 0.56 0.54 0.54
OverallQual 0.80 1.00 0.58 0.60 0.53 0.55 0.46 0.54 0.42 0.57
GrLivArea 0.72 0.58 1.00 0.47 0.39 0.45 0.52 0.64 0.83 0.19
GarageCars 0.65 0.60 0.47 1.00 0.45 0.89 0.45 0.47 0.36 0.54
TotalBsmtSF 0.65 0.53 0.39 0.45 1.00 0.47 0.80 0.32 0.26 0.40
GarageArea 0.64 0.55 0.45 0.89 0.47 1.00 0.47 0.40 0.33 0.48
1stFlrSF 0.63 0.46 0.52 0.45 0.80 0.47 1.00 0.37 0.39 0.28
FullBath 0.56 0.54 0.64 0.47 0.32 0.40 0.37 1.00 0.55 0.47
TotRmsAbvGrd 0.54 0.42 0.83 0.36 0.26 0.33 0.39 0.55 1.00 0.09
YearBuilt 0.54 0.57 0.19 0.54 0.40 0.48 0.28 0.47 0.09 1.00
SalePrice
OverallQual
GrLivArea
GarageCars
TotalBsmtSF
GarageArea
1stFlrSF
FullBath
TotRmsAbvGrd
YearBuilt
1.0
0.8
0.6
0.4
0.2

Explanation

- 'OverallQual', 'GrLivArea' and 'TotalBsmtSF' are strongly correlated with 'SalePrice'.
- 'GarageCars' and 'GarageArea' are also some of the most strongly correlated variables.The number of cars that fit into the garage is a consequence of the garage area. 'GarageCars' and 'GarageArea' are really close.Therefore, we just need one of these variables in our analysis (we can keep 'GarageCars' since its correlation with 'SalePrice' is higher).
- 'TotalBsmtSF' and '1stFloor' also seem to be really close. We can keep 'TotalBsmtSF'
- 'FullBath' is really seems to be a important features.
- 'TotRmsAbvGrd' and 'GrLivArea' also seems very close we will decide later which to keep.
- 'YearBuilt' is slightly correlated with 'SalePrice'.

Missing Data

Important questions when thinking about missing data:

- How prevalent is the missing data?
- Is missing data random or does it have a pattern?

The answer to these questions is important for practical reasons because missing data can imply a reduction of the sample size. This can prevent us from proceeding with the analysis. Moreover, from a substantive perspective, we need to ensure that the missing data process is not biased and hiding an inconvenient truth.

In [22]:
```
total = train.isnull().sum().sort_values(ascending=False)
percent = (train.isnull().sum()/train.isnull().count()).sort_values(ascending=False)
missing_data = pd.concat([total, percent], axis=1, keys=['Total', 'Percent'])
missing_data.head(20)
```

Out[22]:

	Total	Percent
PoolQC	1451	0.996566
MiscFeature	1402	0.962912
Alley	1365	0.937500
Fence	1176	0.807692
FireplaceQu	690	0.473901
LotFrontage	259	0.177885
GarageCond	81	0.055632
GarageType	81	0.055632
GarageYrBlt	81	0.055632
GarageFinish	81	0.055632
GarageQual	81	0.055632
BsmtExposure	38	0.026099
BsmtFinType2	38	0.026099
BsmtFinType1	37	0.025412
BsmtCond	37	0.025412
BsmtQual	37	0.025412
MasVnrArea	8	0.005495
MasVnrType	8	0.005495
Electrical	1	0.000687
Utilities	0	0.000000

Data processing

I have tried few approaches to data preprocessing. The current one was the best one for all the models.Below are the steps I have taken to preprocess the data.

- I have filled missing values of some data features with zero because these missing value means it does not exist in the house.
- I have label encoded the ordinal value containing features. Ordinal values are which are used something along the line of "Good","Average","Bad"
- I have label encodded object type data which are not ordinal in nature
- I have also done some feature engineering, meaning I have created some new features from already existing features.

Lable Encoding

Label Encoding refers to converting the labels into numeric form so as to convert it into the machine-readable form. Machine learning algorithms can then decide in a better way on how those labels must be operated.

In this dataset, there are lot of features which don't represent a quantitative value but rather is actually a label of some sort. For this particular dataset, almost all of the labeled values are in the form of 'string' or words. Only a couple of the labels are represented with numbers. For example, lets check the feature 'Alley', which denotes the type of alley access to the property using the following labels. The meaning of the labels are also given

```
Grvl    Gravel
Pave    Paved
NA  No alley access
```

In the real world, labels are in the form of words, because words are human readable. So it makes sense from that perspective. But when it comes tho machine learning models, which works with numbers, we hit a bit of a roadblock. To remedy this, there is a need to use Label Encoding. Label encoding refers to the process of transforming the word labels into numerical form. This enables the algorithms to operate on data that have textual labels

In case of the labels there are two distinct types, "nominal" and "ordinal". The terms "nominal" and "ordinal" refer to different types of categorizable data.

"Nominal" data assigns names to each data point without placing it in some sort of order. For example, the results of a test could be each classified nominally as a "pass" or "fail."

"Ordinal" data groups data according to some sort of ranking system: it orders the data. For example, this dataset has a very common ranking system which is as follows

```
Ex  Excellent
Gd  Good
TA  Average/Typical
Fa  Fair
Po  Poor
```

Merge Train and Test Data for processing

In [23]:
```
ntrain = train.shape[0]
ntest = test.shape[0]
target = train.SalePrice
all_data = pd.concat((train, test), sort=False).reset_index(drop=True)
all_data.drop(['SalePrice'], axis=1, inplace=True)
print("all_data size is : {}".format(all_data.shape))
```

```
all_data size is : (2915, 80)
```

Function for Imputing missing data

Two of these following part would be used in the common data processing section to impute missing data.

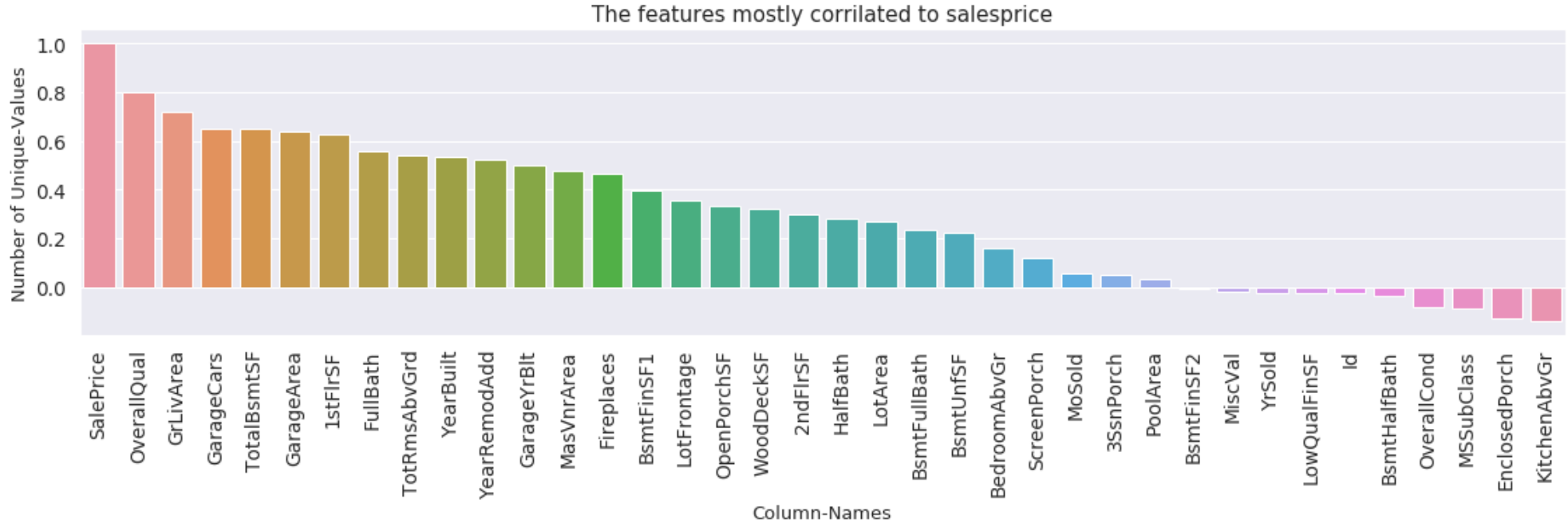
In [24]:
```
lot_frontage_by_neighborhood = all_data["LotFrontage"].groupby(all_data["Neighborhood"])
```

Following function will be used to convert categorical features as number.

In [25]:
```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()

def factorize(df, factor_df, column, fill_na=None):
    factor_df[column] = df[column]
    if fill_na is not None:
        factor_df[column].fillna(fill_na, inplace=True)
    le.fit(factor_df[column].unique())
    factor_df[column] = le.transform(factor_df[column])
    return factor_df
```

In the following section we are looking into the features that are more related to saleprice. we should handle mostly corrilated features carefully because they will contribute more for our prediction.

In [26]:
```
 # THE FEATURES MOSTLY CORRELATED TO SALESPRICE
cols = train.dtypes[train.dtypes != 'object'].index
corrs=[]
for item in cols:
    corrs.append((train[item].corr(train['SalePrice'])))
ist = pd.DataFrame(
    {'cols': cols,
    'corrs': corrs
    })
ist = ist.sort_values(by='corrs', ascending=False)
#ist.head()
plt.subplots(figsize=(19, 4))
sns.barplot(x=ist['cols'], y=ist['corrs'])
plt.xticks(rotation=90)
plt.ylabel('Number of Unique-Values', fontsize=13)
plt.xlabel('Column-Names', fontsize=13)
plt.title('The features mostly corrilated to salesprice')
plt.show()
```



Making Function for Feature-Analysis

All features of house-price dataset have been separated into 3 different categories: Type, Size and Period(Year/Month). Type or size based features represent type or size of the sample respectively. Type-based features are generally non-numeric and have very few categories.

In [27]:
```
def type_based_feature_analysis(column, rotation = None):
    order_all_data = all_data[column].value_counts().index
    order_train = train[column].value_counts().index

    if rotation is None: rotation = 90

    plt.subplots(figsize =(19, 4))
    plt.subplot(1, 5, 1)
    sns.barplot(x=train[column], y=train['SalePrice'], order = order_train)
    plt.xlabel(column+' (train)')
    plt.xticks(rotation=rotation)

    plt.subplot(1, 5, 2)
    sns.countplot(x=train[column], order = order_train)
    plt.xlabel(column+' (train)')
    plt.xticks(rotation=rotation)

    plt.subplot(1, 5, 3)
    sns.countplot(x=all_data[column], order = order_all_data)
    plt.xlabel(column+' (all_data)')
    plt.xticks(rotation=rotation)

    plt.subplot(1, 5, 4)
    sns.stripplot(x=train[column], y=train['SalePrice'], jitter = True, order = order_train)
    plt.xlabel(column+' (train)')
    plt.xticks(rotation=rotation)

    plt.subplot(1, 5, 5)
    sns.boxplot(x=train[column], y=train['SalePrice'], order = order_train)
    plt.xlabel(column+' (train)')
    plt.xticks(rotation=rotation)

    plt.show()
```

Size based features show the area: square-feet that are continuous numeric values.

In [28]:
```
from scipy.stats.stats import pearsonr
def size_based_feature_analysis(column):

    grid = plt.GridSpec(3, 4)
    plt.subplots(figsize =(19, 7))
    zero = 0
    rotation = 90

    plt.subplot(grid[zero, 0])
    sns.barplot(x=train['HouseStyle'],y=train[column])
    plt.xticks(rotation=rotation)

    plt.subplot(grid[zero,1])
    sns.barplot(x=train['BldgType'],y=train[column])
    plt.xticks(rotation=rotation)

    plt.subplot(grid[zero, 2])
    sns.barplot(x=train['LotShape'],y=train[column])
    plt.xticks(rotation=rotation)

    plt.subplot(grid[zero, 3])
    g = sns.regplot(x=train[column], y=train['SalePrice'], fit_reg=False,
                    label = "corr: %2f"%(pearsonr(train[column], train['SalePrice'])[0]))
    g = g.legend(loc='best', fontsize=12)
    plt.xticks(rotation=rotation)

    plt.subplot(grid[2, 0:])
    sns.boxplot(x=train['Neighborhood'],y=train[column])
    plt.xticks(rotation=rotation)

    plt.show()
```

Period/Year based features show duration or periods.

In [29]:
```
def year_based_feature_analysis(column, rotation = None, box = True):
    if rotation is None: rotation = 90

    if(box == True):
        n = 3
        a = 2
        b = 3
        plt.subplots(figsize =(19, 16))
    else:
        n = 2
        b = 2
        plt.subplots(figsize =(19, 9))

    plt.subplot(n, 1, 1)
    sns.barplot(x=train[column], y=train['SalePrice'])
    plt.xlabel(column+' (train)')
    plt.xticks(rotation=rotation)

    if(box == True):
        plt.subplot(n, 1, a)
        sns.boxplot(x=train[column], y=train['SalePrice'])
        plt.xlabel(column+' (train)')
        plt.xticks(rotation=rotation)

    plt.subplot(n, 1, b)
    sns.stripplot(x=train[column], y=train['SalePrice'], jitter = True)
    plt.xlabel(column+' (train)')
    plt.xticks(rotation=rotation)

    plt.show()
```

All these 3 categories have their own unique pattern of values. Therefore they have been presented in 3 separeate ways all through the karnel. Lets have a look at the graph-plotting of the functions and analyze then one-by-one.

common data processing:

In this part we have label encoded some of the columns because some features are ordinal. I have replaced some null value with zero because in those case they probably meant that it may not exist . Finally I have merged some of the features to get a better feature.

Befor starting following block its important to understand which feature means what so that describing my work would be easier

- SalePrice - the property's sale price in dollars. This is the target variable that you're trying to predict.
- MSSubClass: The building class
- MSZoning: The general zoning classification
- LotFrontage: Linear feet of street connected to property
- LotArea: Lot size in square feet
- Street: Type of road access
- Alley: Type of alley access
- LotShape: General shape of property
- LandContour: Flatness of the property
- Utilities: Type of utilities available
- LotConfig: Lot configuration
- LandSlope: Slope of property
- Neighborhood: Physical locations within Ames city limits
- Condition1: Proximity to main road or railroad
- Condition2: Proximity to main road or railroad (if a second is present)
- BldgType: Type of dwelling
- HouseStyle: Style of dwelling
- OverallQual: Overall material and finish quality
- OverallCond: Overall condition rating
- YearBuilt: Original construction date
- YearRemodAdd: Remodel date
- RoofStyle: Type of roof
- RoofMatl: Roof material
- Exterior1st: Exterior covering on house
- Exterior2nd: Exterior covering on house (if more than one material)
- MasVnrType: Masonry veneer type
- MasVnrArea: Masonry veneer area in square feet
- ExterQual: Exterior material quality
- ExterCond: Present condition of the material on the exterior
- Foundation: Type of foundation
- BsmtQual: Height of the basement
- BsmtCond: General condition of the basement
- BsmtExposure: Walkout or garden level basement walls
- BsmtFinType1: Quality of basement finished area
- BsmtFinSF1: Type 1 finished square feet
- BsmtFinType2: Quality of second finished area (if present)
- BsmtFinSF2: Type 2 finished square feet
- BsmtUnfSF: Unfinished square feet of basement area
- TotalBsmtSF: Total square feet of basement area
- Heating: Type of heating
- HeatingQC: Heating quality and condition

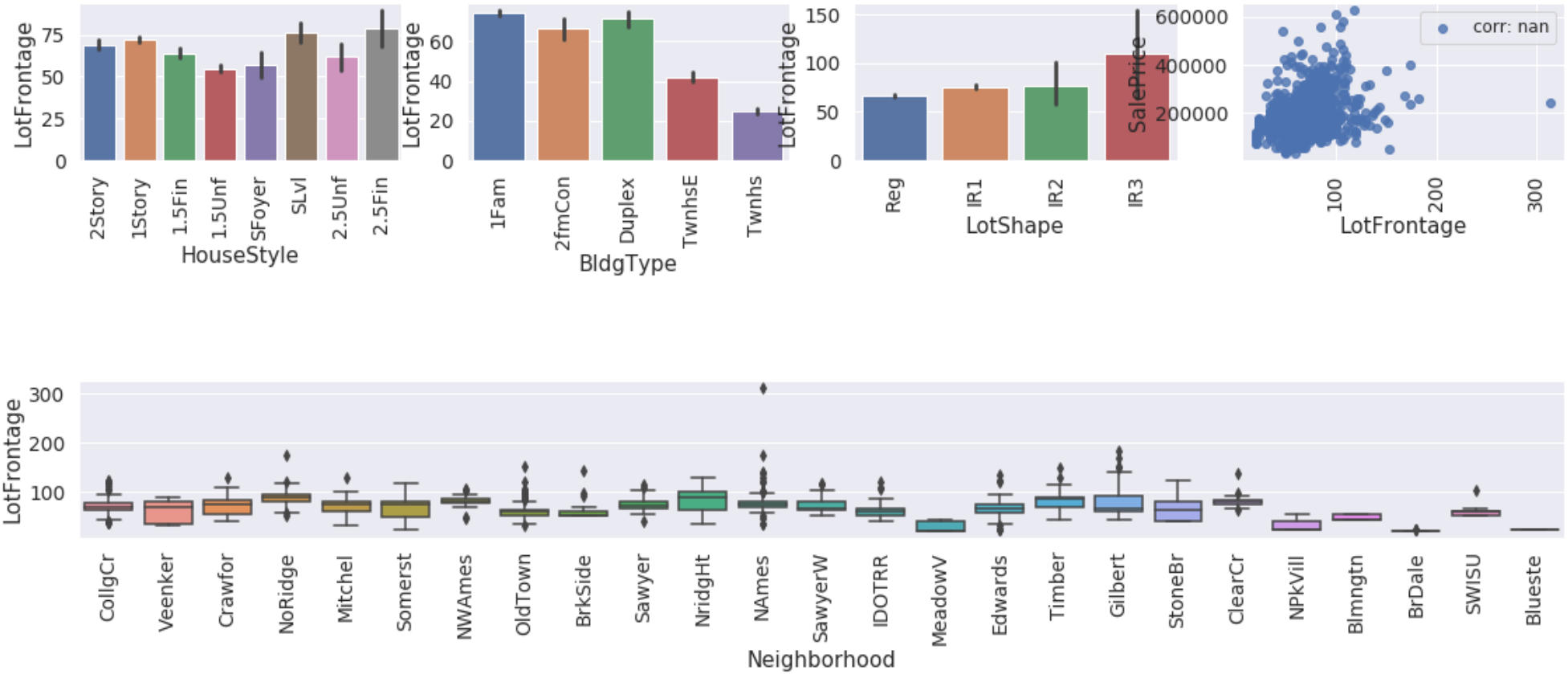
- CentralAir: Central air conditioning
- Electrical: Electrical system
- 1stFlrSF: First Floor square feet
- 2ndFlrSF: Second floor square feet
- LowQualFinSF: Low quality finished square feet (all floors)
- GrLivArea: Above grade (ground) living area square feet
- BsmtFullBath: Basement full bathrooms
- BsmtHalfBath: Basement half bathrooms
- FullBath: Full bathrooms above grade
- HalfBath: Half baths above grade
- Bedroom: Number of bedrooms above basement level
- Kitchen: Number of kitchens
- KitchenQual: Kitchen quality
- TotRmsAbvGrd: Total rooms above grade (does not include bathrooms)
- Functional: Home functionality rating
- Fireplaces: Number of fireplaces
- FireplaceQu: Fireplace quality
- -GarageType: Garage location
- GarageYrBlt: Year garage was built
- GarageFinish: Interior finish of the garage
- GarageCars: Size of garage in car capacity
- GarageArea: Size of garage in square feet
- GarageQual: Garage quality
- GarageCond: Garage condition
- PavedDrive: Paved driveway
- WoodDeckSF: Wood deck area in square feet
- OpenPorchSF: Open porch area in square feet
- EnclosedPorch: Enclosed porch area in square feet
- 3SsnPorch: Three season porch area in square feet
- ScreenPorch: Screen porch area in square feet
- PoolArea: Pool area in square feet
- PoolQC: Pool quality
- Fence: Fence quality
- MiscFeature: Miscellaneous feature not covered in other categories
- MiscVal: $Value of miscellaneous feature
- MoSold: Month Sold
- YrSold: Year Sold
- SaleType: Type of sale
- SaleCondition: Condition of sale

Missing data handle

In [30]:
```
all_df = pd.DataFrame(index =all_data.index)
```

LotFrontage: Linear feet of street connected to property. Now this property of the house is most likely going to be similiar to the other ones in the neighbourhood. ´SO let us group and impute with the median (due to potential outliers)

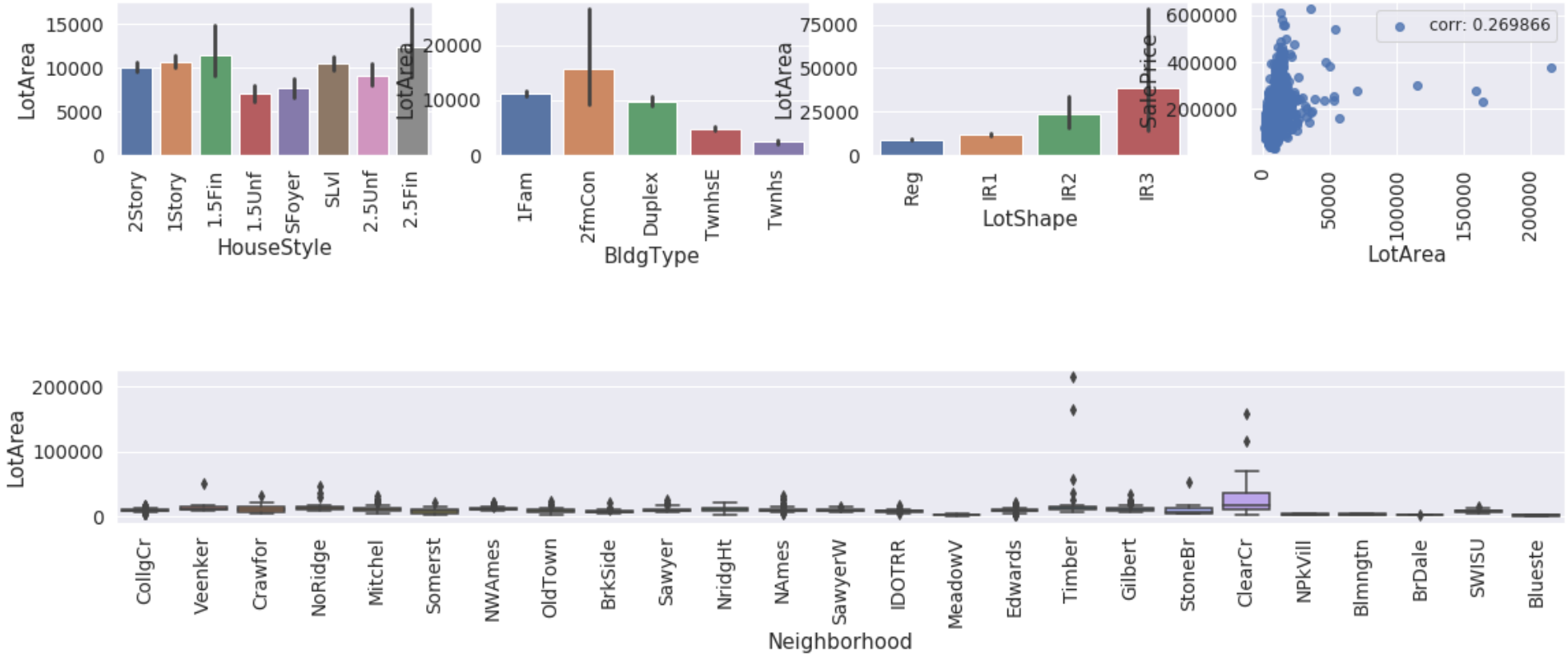
In [31]:
```
size_based_feature_analysis("LotFrontage")
```



In [32]:
```
all_df["LotFrontage"] =all_data["LotFrontage"]
for key, group in lot_frontage_by_neighborhood:
    #Filling in missing LotFrontage values by the median
    idx = (all_data["Neighborhood"] == key) & (all_data["LotFrontage"].isnull())
    all_df.loc[idx, "LotFrontage"] = group.median()
```

LotArea is a numeric value and not required to change so I am keeping as it is.

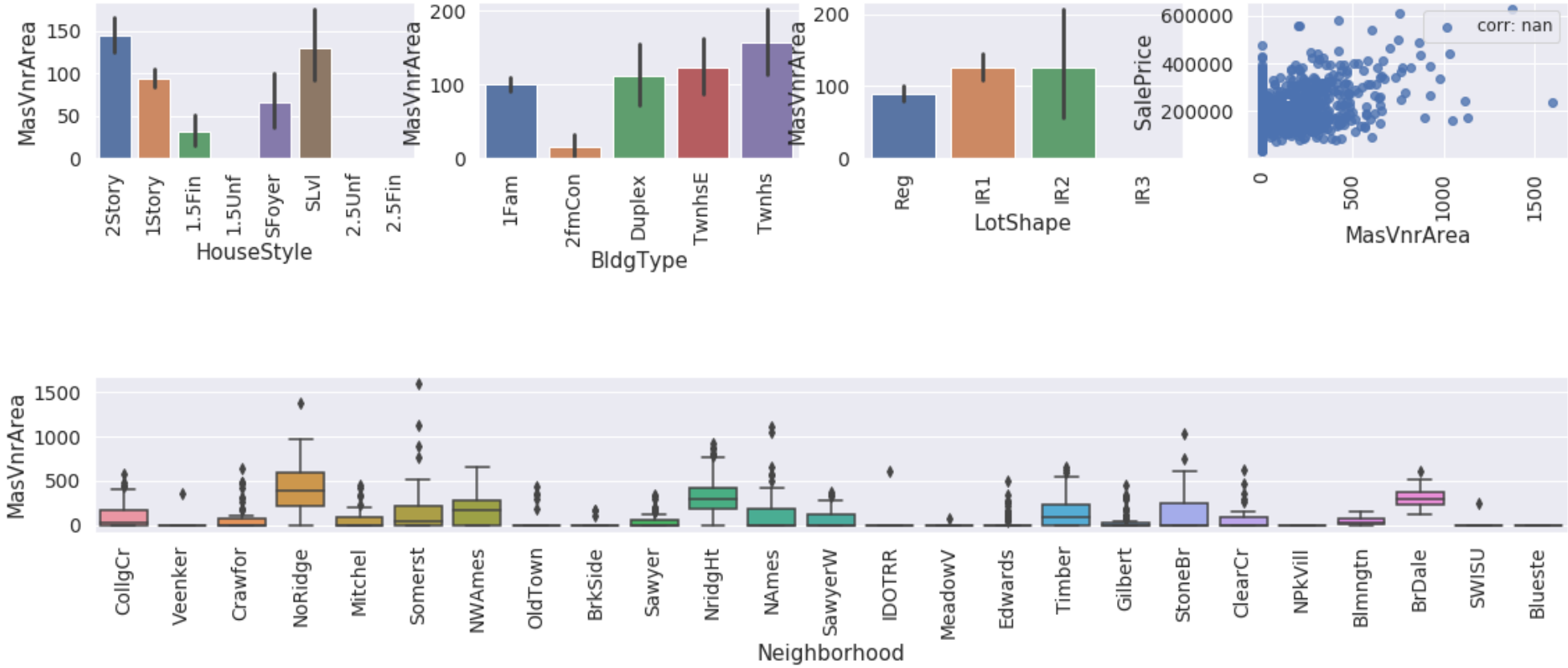
In [33]:
```
size_based_feature_analysis("LotArea")
```



In [34]:
```
all_df["LotArea"] =all_data["LotArea"]
```

MasVnrArea : NA most likely means no masonry veneer for these houses. We can fill 0 for the area and None for the type.

In [35]:
```
size_based_feature_analysis("MasVnrArea")
```



In [36]:
```
all_df["MasVnrArea"] =all_data["MasVnrArea"]
all_df["MasVnrArea"].fillna(0, inplace=True)
```

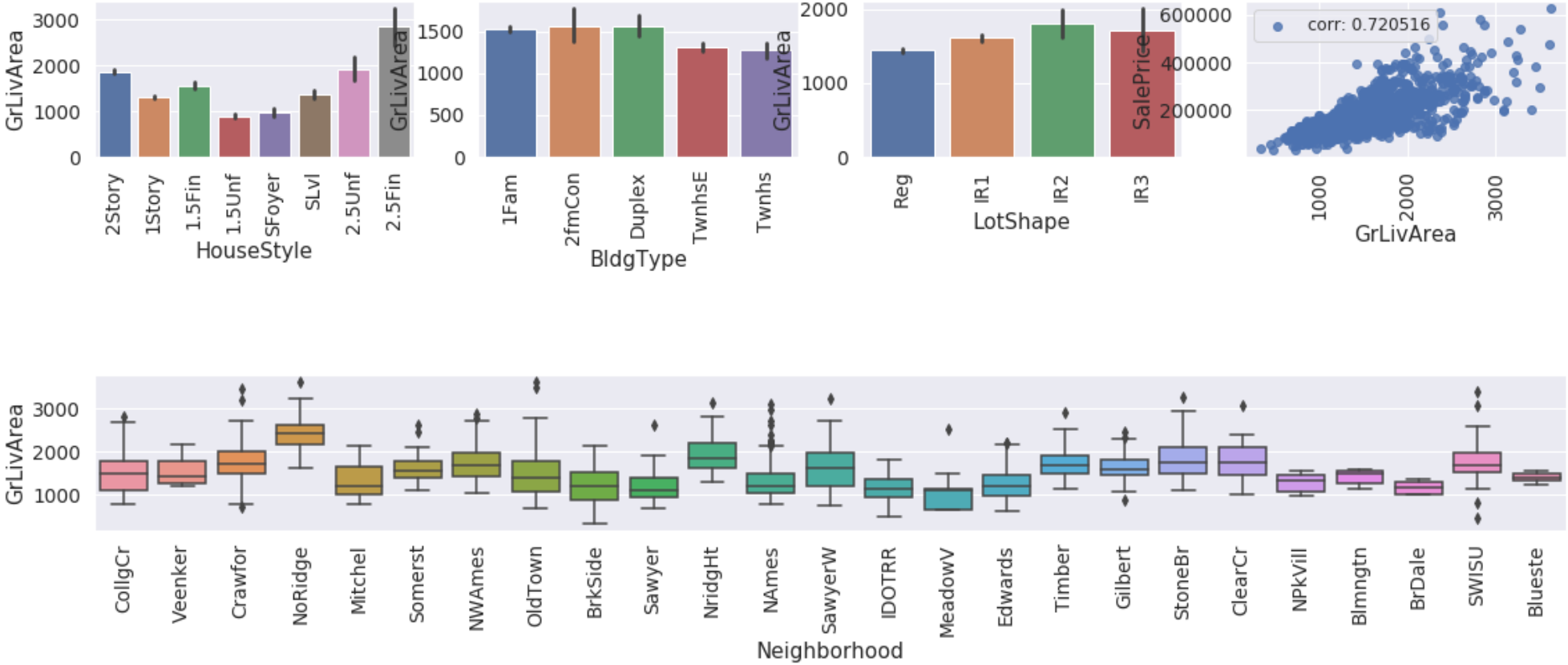
BsmtFinSF1: Type 1 finished square feet BsmtFinType2: Rating of basement finished area (if multiple types) GLQ Good Living Quarters ALQ Average Living Quarters BLQ Below Average Living Quarters Rec Average Rec Room LwQ Low Quality Unf Unfinshed NA No Basement BsmtFinSF2: Type 2 finished square feet BsmtUnfSF: Unfinished square feet of basement area TotalBsmtSF: Total square feet of basement area

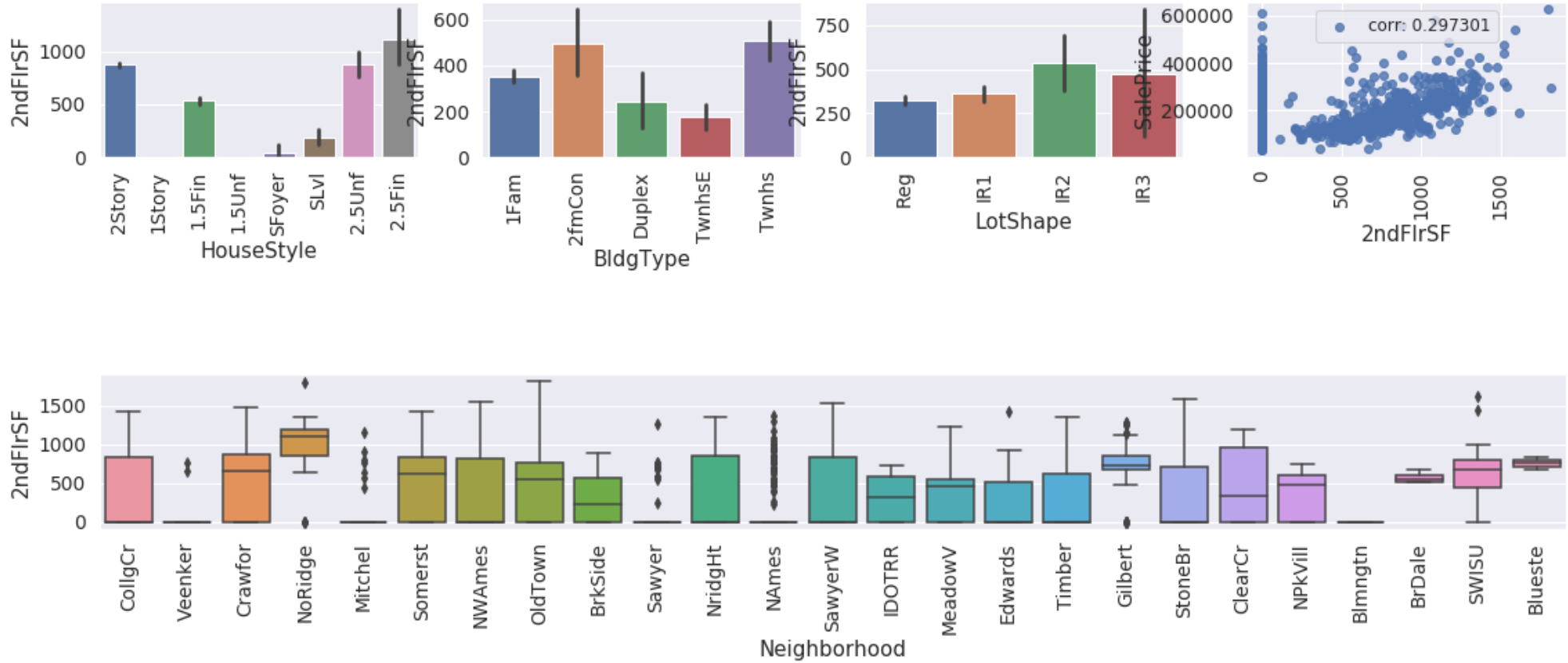
We can notice is that some variables will inherit the imputed value due to the fact that we do not have the object at hand. For example havign no garage implies for the following 3 variables that NaN means 0.

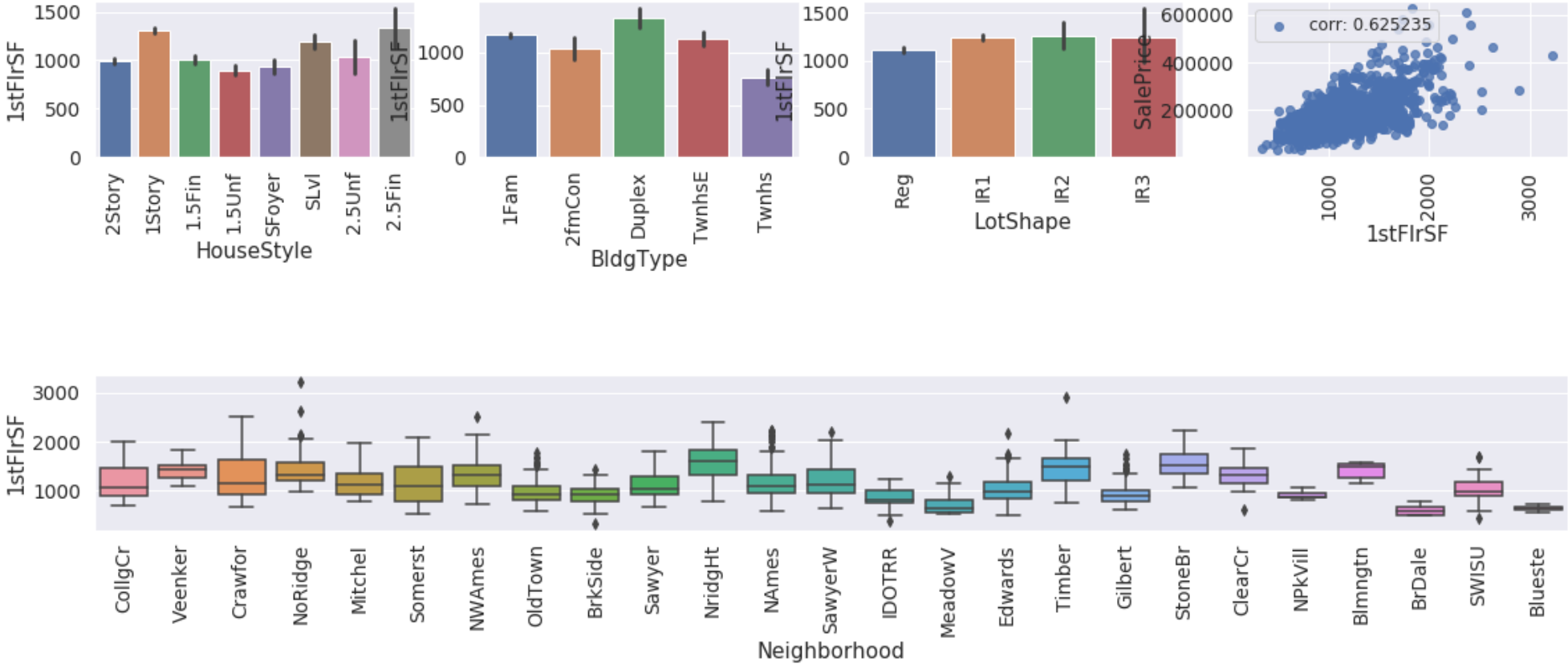
In [37]:
```
all_df["BsmtFinSF1"] =all_data["BsmtFinSF1"]
all_df["BsmtFinSF1"].fillna(0, inplace=True)

all_df["BsmtFinSF2"] =all_data["BsmtFinSF2"]
all_df["BsmtFinSF2"].fillna(0, inplace=True)

all_df["BsmtUnfSF"] =all_data["BsmtUnfSF"]
all_df["BsmtUnfSF"].fillna(0, inplace=True)

all_df["TotalBsmtSF"] =all_data["TotalBsmtSF"]
all_df["TotalBsmtSF"].fillna(0, inplace=True)
```

Following sections does not have any null values and all of them are live area so I am keeping them as it is

In [38]:
```
size_based_feature_analysis('GrLivArea')
size_based_feature_analysis('2ndFlrSF')
size_based_feature_analysis('1stFlrSF')
```


GrLivArea
HouseStyle
2Story
1Story
1.5Fin
1.5Unf
SFoyer
SLvl
2.5Unf
2.5Fin
BldgType
1Fam
2fmCon
Duplex
TwnhsE
Twnhs
LotShape
Reg
IR1
IR2
IR3
SalePrice
corr: 0.720516
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

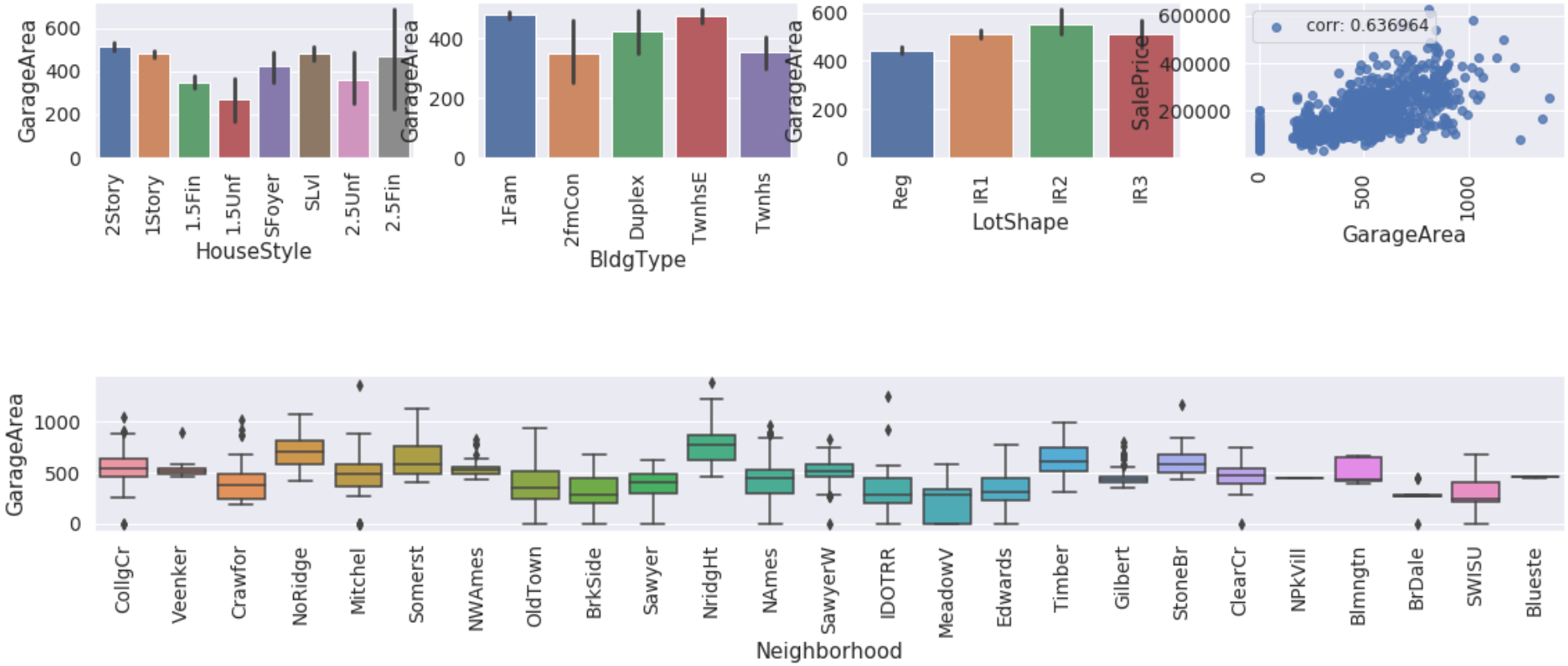

2ndFlrSF
HouseStyle
2Story
1Story
1.5Fin
1.5Unf
SFoyer
SLvl
2.5Unf
2.5Fin
BldgType
1Fam
2fmCon
Duplex
TwnhsE
Twnhs
LotShape
Reg
IR1
IR2
IR3
SalePrice
corr: 0.297301
Neighborhood
CollgCr
Veenker
Crawfor
NoRidge
Mitchel
Somerst
NWAmes
OldTown
BrkSide
Sawyer
NridgHt
NAmes
SawyerW
IDOTRR
MeadowV
Edwards
Timber
Gilbert
StoneBr
ClearCr
NPkVill
Blmngtn
BrDale
SWISU
Blueste



In [39]:
```
all_df["1stFlrSF"] =all_data["1stFlrSF"]
all_df["2ndFlrSF"] =all_data["2ndFlrSF"]
all_df["GrLivArea"] =all_data["GrLivArea"]
```

GarageArea: Size of garage in square feet. NA means no garage available. so putting '0' to make it numeric.

In [40]:
```
size_based_feature_analysis('GarageArea')
```



In [41]:
```
all_df["GarageArea"] =all_data["GarageArea"]
all_df["GarageArea"].fillna(0, inplace=True)
```

Following features are neumeric and no null is present so we can keep them as they are.

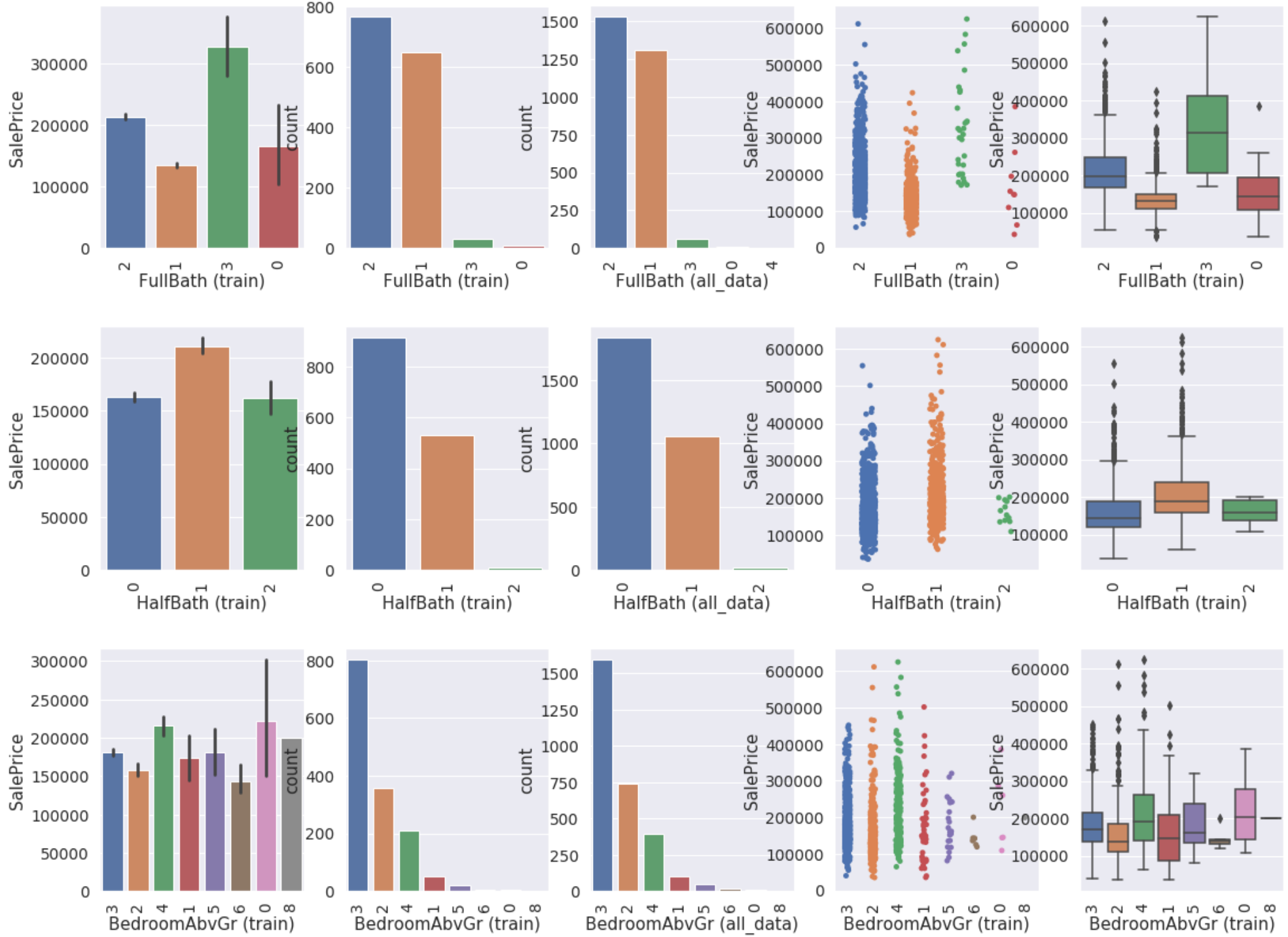
In [42]:
```
all_df["WoodDeckSF"] =all_data["WoodDeckSF"]
all_df["OpenPorchSF"] =all_data["OpenPorchSF"]
all_df["EnclosedPorch"] =all_data["EnclosedPorch"]
all_df["3SsnPorch"] =all_data["3SsnPorch"]
all_df["ScreenPorch"] =all_data["ScreenPorch"]
```

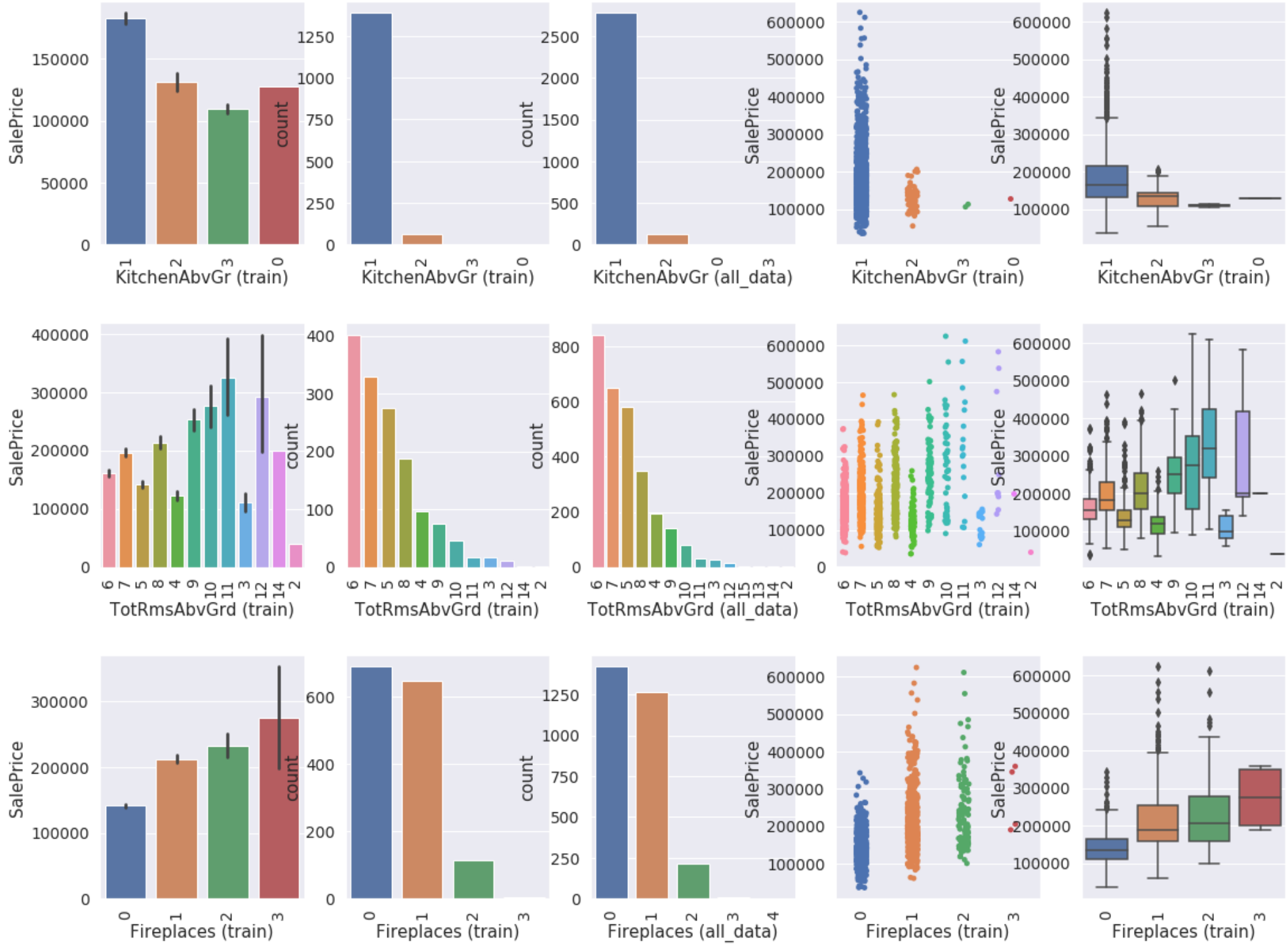
In the following features Null means no GarageCars or no BsmFullBath so we can fill null with zero

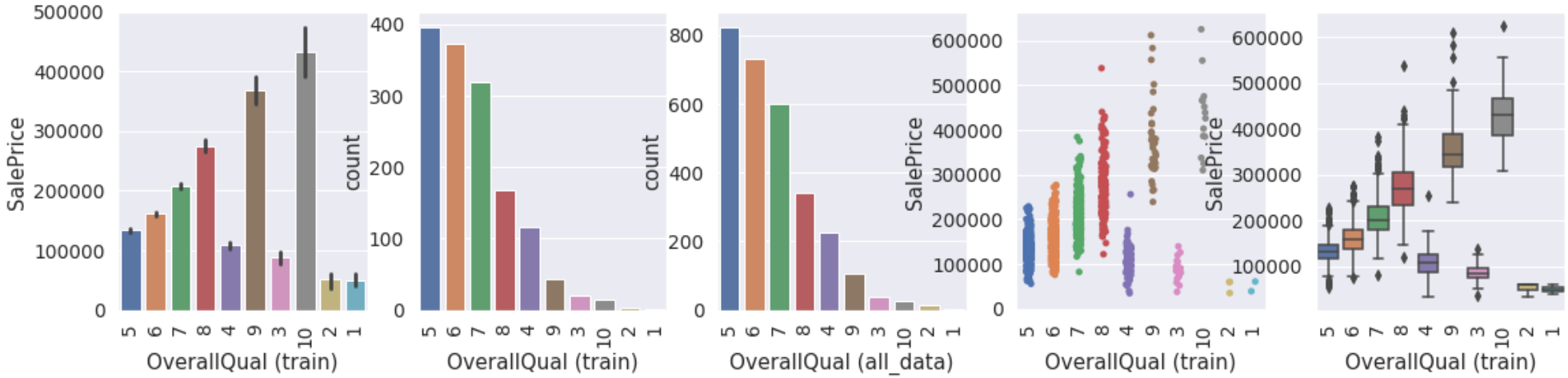
In [43]:
```
all_df["BsmtFullBath"] =all_data["BsmtFullBath"]
all_df["BsmtFullBath"].fillna(0, inplace=True)

all_df["BsmtHalfBath"] =all_data["BsmtHalfBath"]
all_df["BsmtHalfBath"].fillna(0, inplace=True)

all_df["GarageCars"] =all_data["GarageCars"]
all_df["GarageCars"].fillna(0, inplace=True)
```

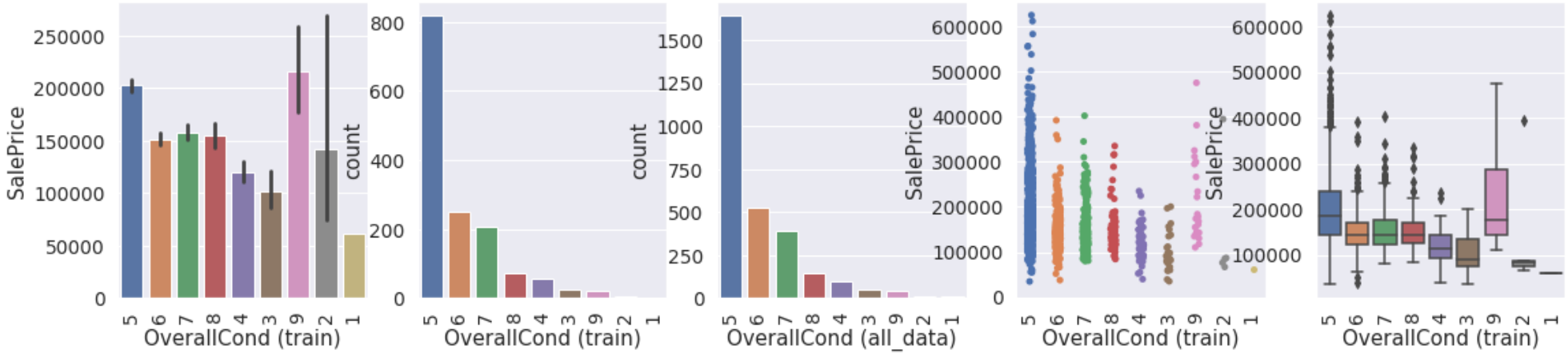
Following section dont have any missing value and the values are numerical so we can keep them untouched.

In [44]:
```
type_based_feature_analysis('FullBath')
type_based_feature_analysis("HalfBath")
type_based_feature_analysis("BedroomAbvGr")
type_based_feature_analysis("KitchenAbvGr")
type_based_feature_analysis("TotRmsAbvGrd")
type_based_feature_analysis("Fireplaces")
type_based_feature_analysis("OverallQual")
type_based_feature_analysis("OverallCond")
```


SalePrice
count
FullBath (train)
FullBath (all_data)
HalfBath (train)
HalfBath (all_data)
BedroomAbvGr (train)
BedroomAbvGr (all_data)


SalePrice
count
KitchenAbvGr (train)
KitchenAbvGr (train)
KitchenAbvGr (all_data)
KitchenAbvGr (train)
KitchenAbvGr (train)
SalePrice
count
TotRmsAbvGrd (train)
TotRmsAbvGrd (train)
TotRmsAbvGrd (all_data)
TotRmsAbvGrd (train)
TotRmsAbvGrd (train)
SalePrice
count
Fireplaces (train)
Fireplaces (train)
Fireplaces (all_data)
Fireplaces (train)
Fireplaces (train)

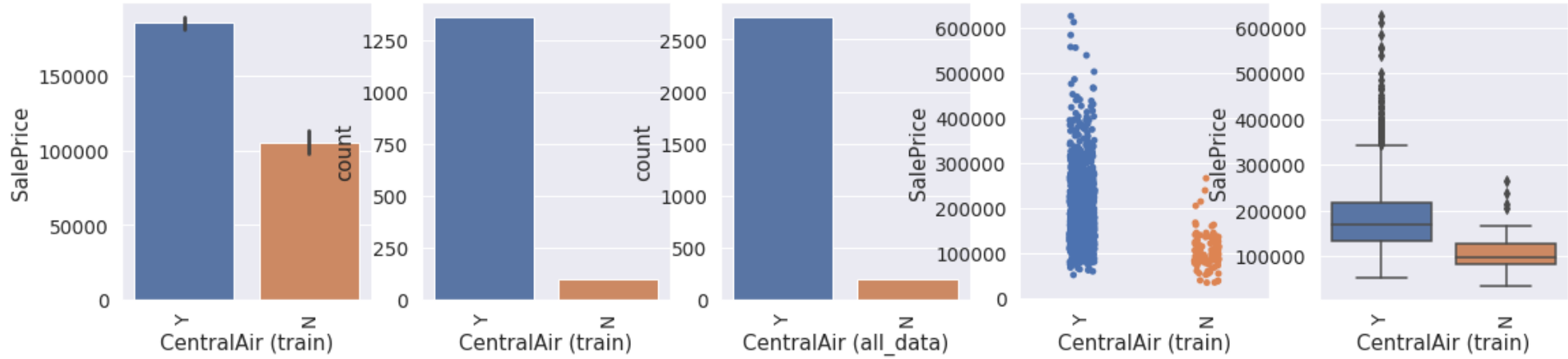




In [45]:
```
all_df["FullBath"] =all_data["FullBath"]
all_df["HalfBath"] =all_data["HalfBath"]
all_df["BedroomAbvGr"] =all_data["BedroomAbvGr"]
all_df["KitchenAbvGr"] =all_data["KitchenAbvGr"]
all_df["TotRmsAbvGrd"] =all_data["TotRmsAbvGrd"]
all_df["Fireplaces"] =all_data["Fireplaces"]
all_df["OverallQual"] =all_data["OverallQual"]
all_df["OverallCond"] =all_data["OverallCond"]
```

In this feature there is only two option either yes or no so converting it to binary will help.

In [46]:
```
type_based_feature_analysis('CentralAir')
```



In [47]:
```
all_df["CentralAir"] = (all_data["CentralAir"] == "Y") * 1.0
```

Lable Encoding for ordinal Data

Following case are ordinal so we are performing label encoding here. In the following section meaning of the orders are:

Ex Excellent

Gd Good

TA Average/Typical

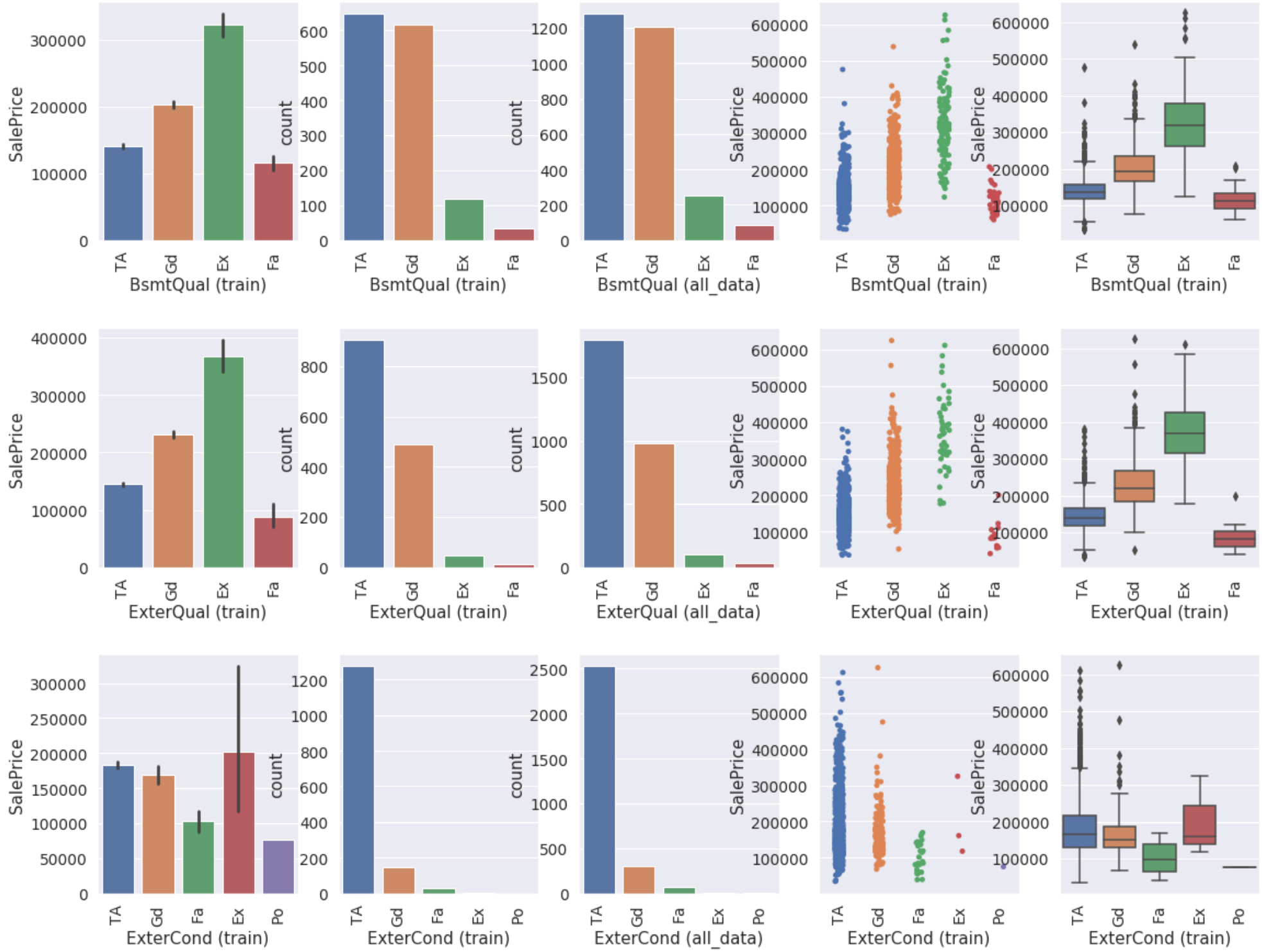
Fa Fair

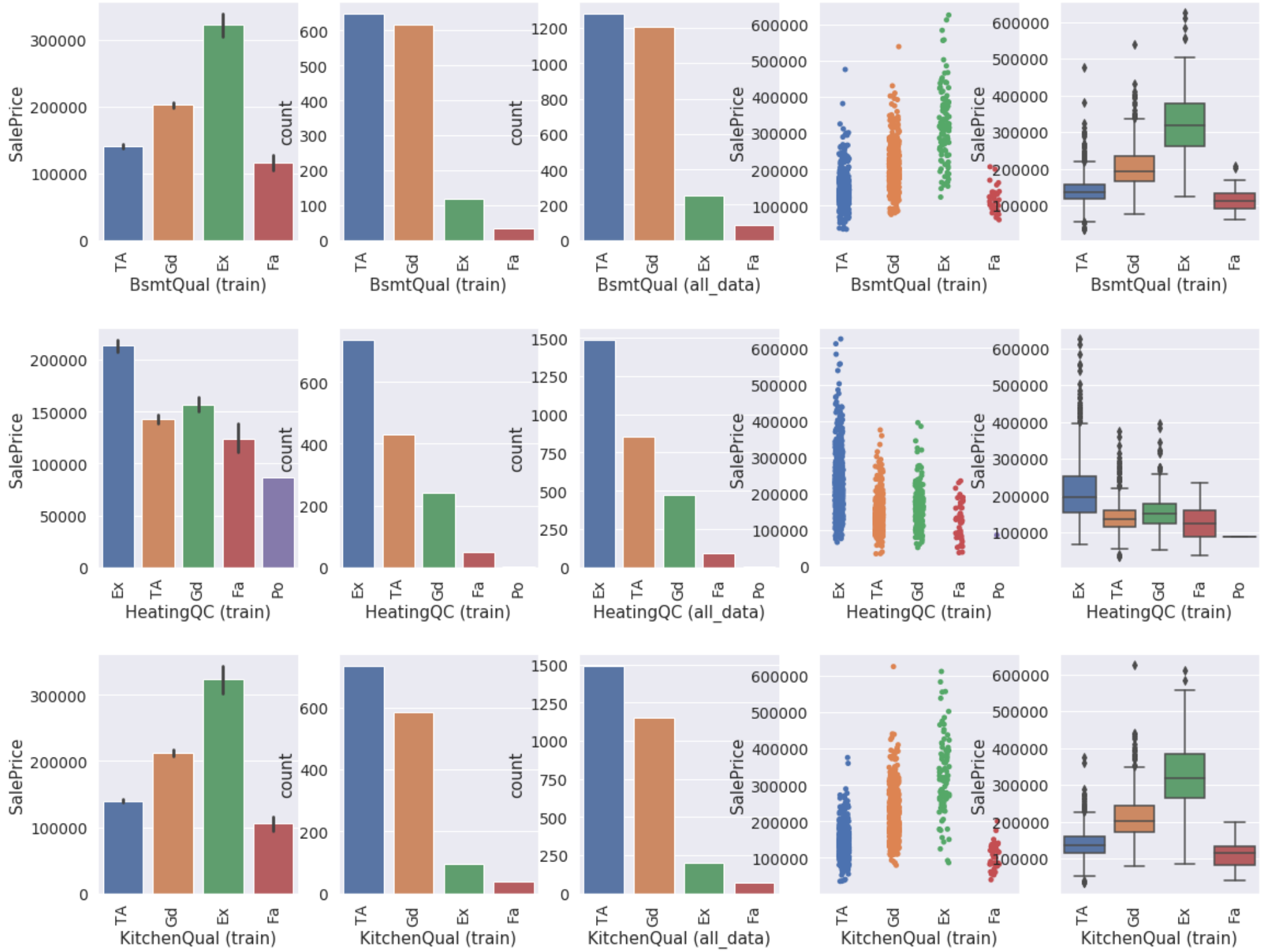
Po Poor

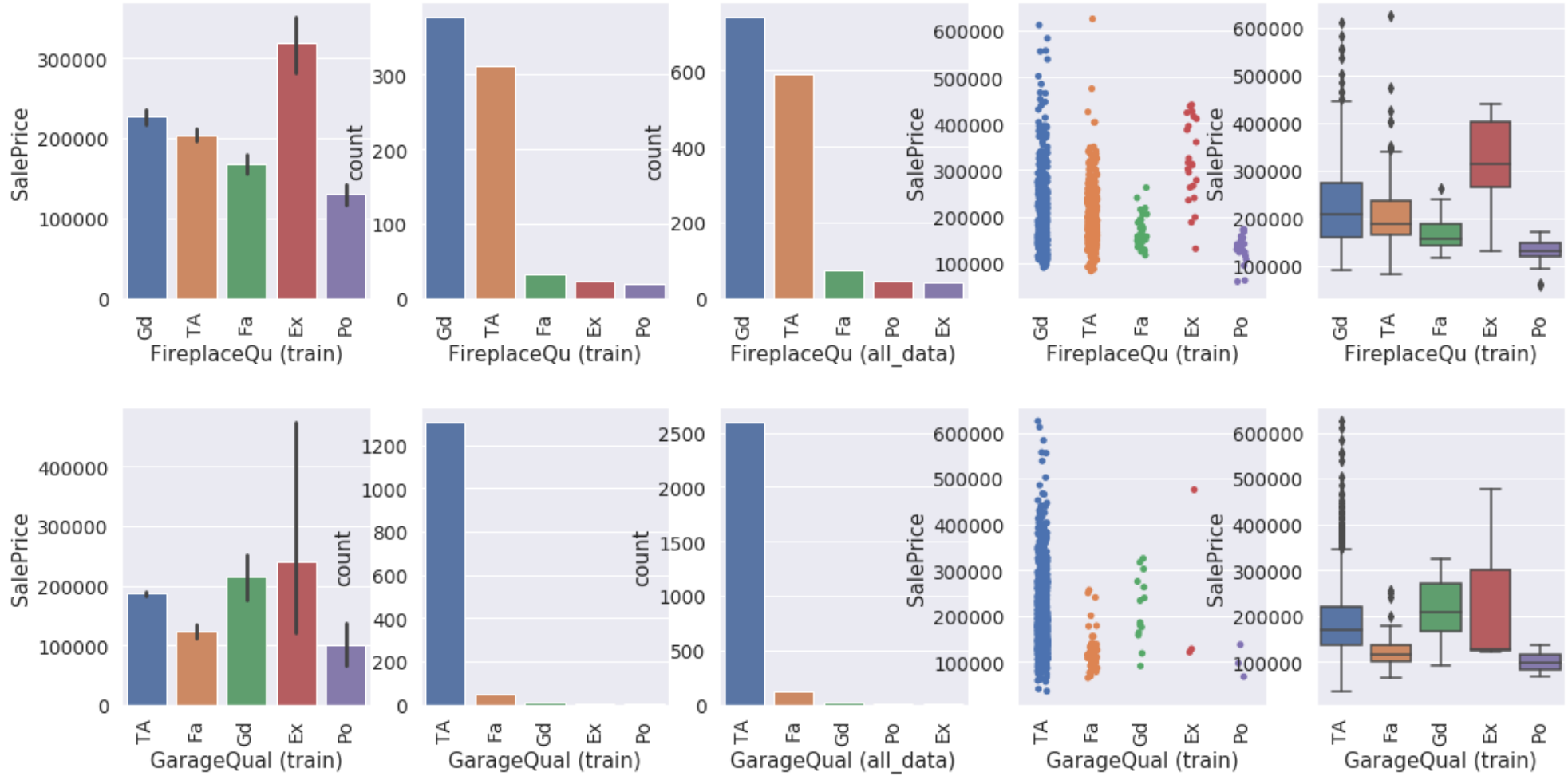
now we can easily convert them to 0 to 5

In the following graph we can see relationship with saleprice

In [48]:
```
type_based_feature_analysis('BsmtQual')
type_based_feature_analysis('ExterQual')
type_based_feature_analysis('ExterCond')
type_based_feature_analysis('BsmtQual')
type_based_feature_analysis('HeatingQC')
type_based_feature_analysis('KitchenQual')
type_based_feature_analysis('FireplaceQu')
type_based_feature_analysis('GarageQual')
```


SalePrice
count
BsmtQual (train)
BsmtQual (all_data)
ExterQual (train)
ExterQual (all_data)
ExterCond (train)
ExterCond (all_data)
TA
Gd
Ex
Fa
Po

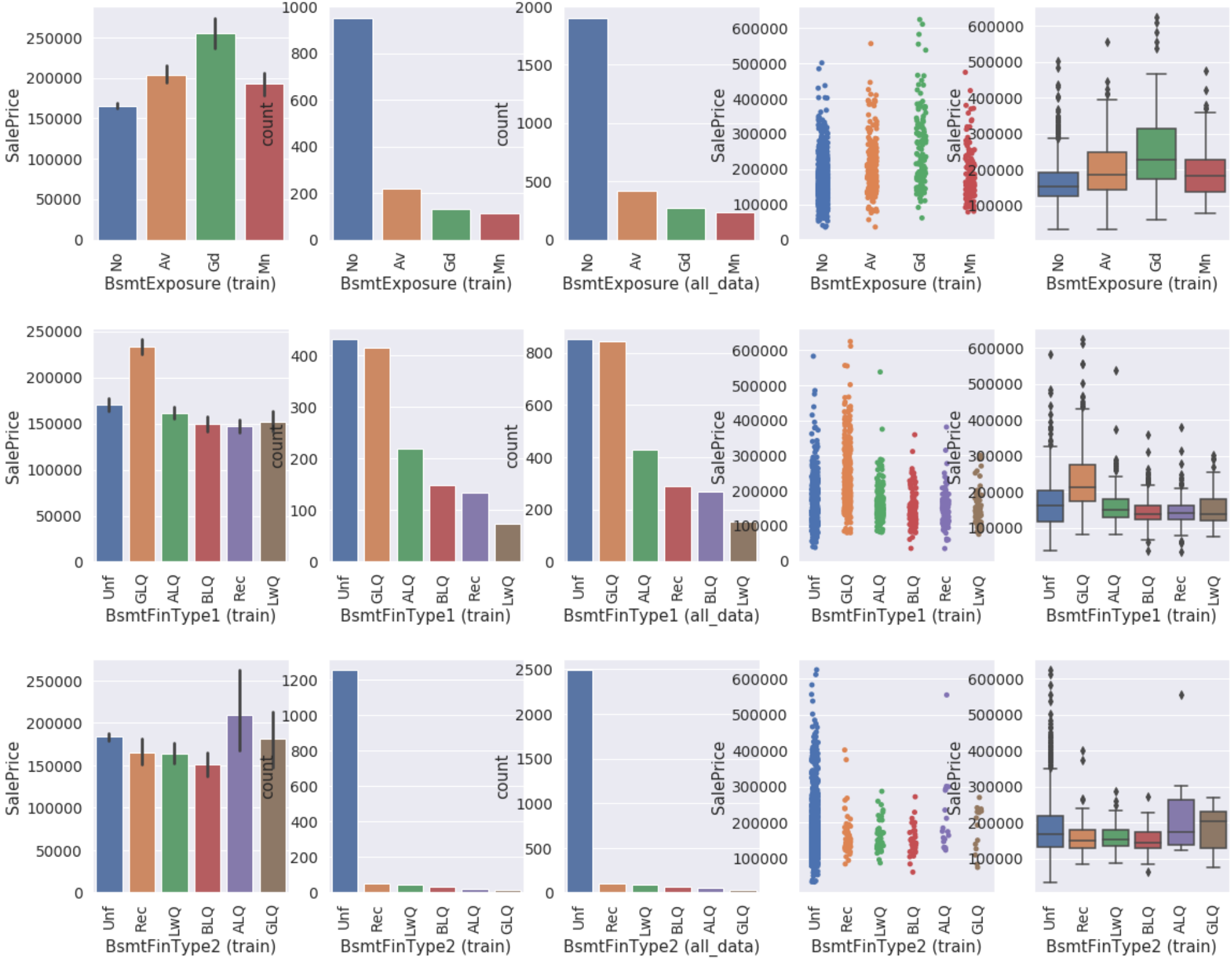

SalePrice
count
TA
Gd
Ex
Fa
BsmtQual (train)
BsmtQual (train)
BsmtQual (all_data)
BsmtQual (train)
BsmtQual (train)
HeatingQC (train)
HeatingQC (train)
HeatingQC (all_data)
HeatingQC (train)
HeatingQC (train)
Po
KitchenQual (train)
KitchenQual (train)
KitchenQual (all_data)
KitchenQual (train)
KitchenQual (train)

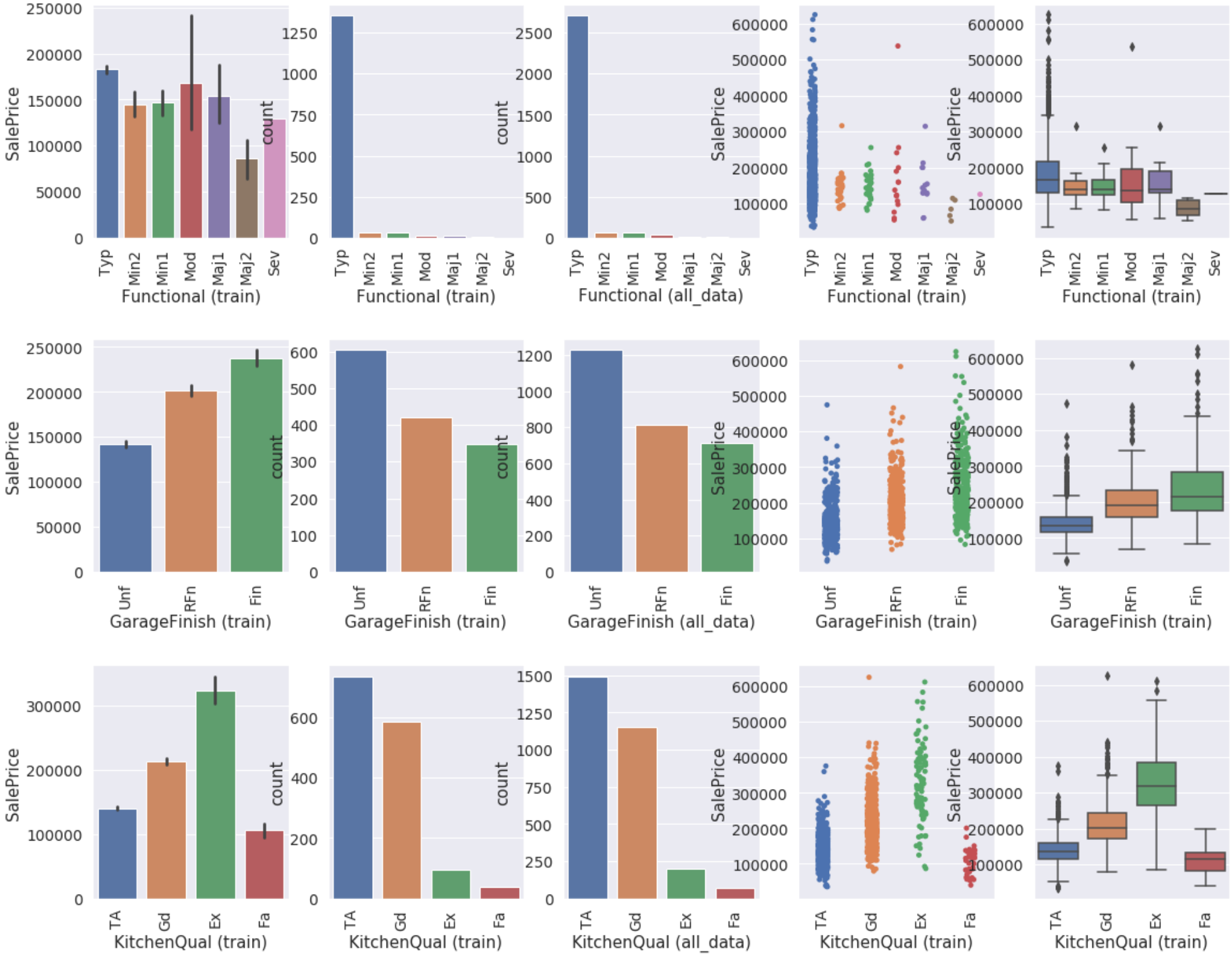


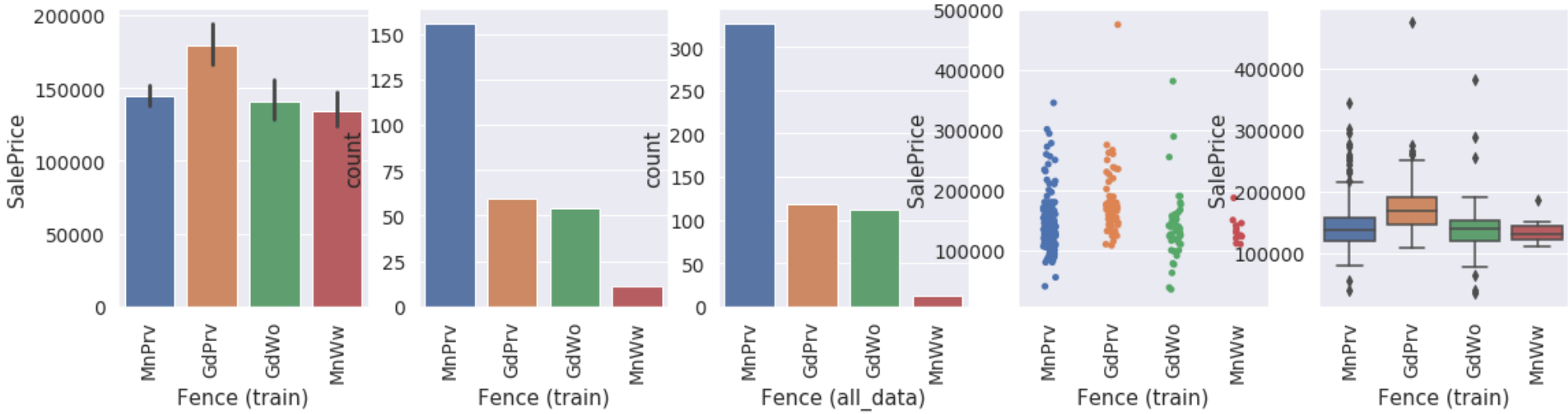
In [49]:
```
nan = float('nan')
qual_dict = {nan: 0, "NA": 0, "Po": 1, "Fa": 2, "TA": 3, "Gd": 4, "Ex": 5}
all_df["ExterQual"] =all_data["ExterQual"].map(qual_dict).astype(int)
all_df["ExterCond"] =all_data["ExterCond"].map(qual_dict).astype(int)
all_df["BsmtQual"] =all_data["BsmtQual"].map(qual_dict).astype(int)
all_df["BsmtCond"] =all_data["BsmtCond"].map(qual_dict).astype(int)
all_df["HeatingQC"] =all_data["HeatingQC"].map(qual_dict).astype(int)
all_df["KitchenQual"] =all_data["KitchenQual"].map(qual_dict).astype(int)
all_df["FireplaceQu"] =all_data["FireplaceQu"].map(qual_dict).astype(int)
all_df["GarageQual"] =all_data["GarageQual"].map(qual_dict).astype(int)
all_df["GarageCond"] =all_data["GarageCond"].map(qual_dict).astype(int)
```

I have converted few more ordinal features to neumerical feature by performing lable encoding. Lets look at their relation with saleprice and the occurance.

In [50]:
```
type_based_feature_analysis('BsmtExposure')
type_based_feature_analysis('BsmtFinType1')
type_based_feature_analysis('BsmtFinType2')
type_based_feature_analysis('Functional')
type_based_feature_analysis('GarageFinish')
type_based_feature_analysis('KitchenQual')
type_based_feature_analysis('Fence')
```


SalePrice
count
No
Av
Gd
Mn
BsmtExposure (train)
BsmtExposure (all_data)
Unf
GLQ
ALQ
BLQ
Rec
LwQ
BsmtFinType1 (train)
BsmtFinType1 (all_data)
BsmtFinType2 (train)
BsmtFinType2 (all_data)

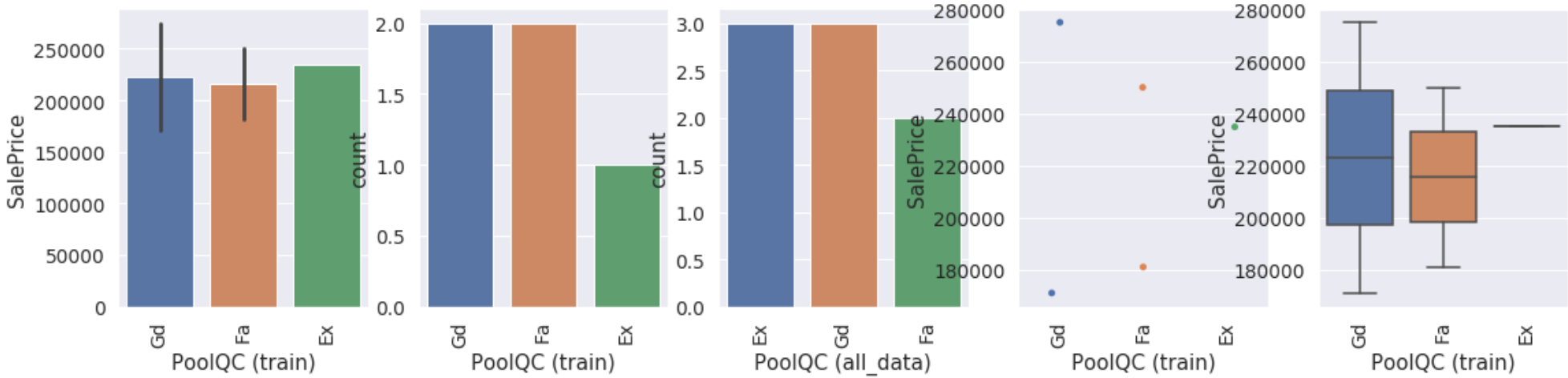

SalePrice
count
Typ
Min2
Min1
Mod
Maj1
Maj2
Sev
Functional (train)
Functional (all_data)
Unf
RFn
Fin
GarageFinish (train)
GarageFinish (all_data)
TA
Gd
Ex
Fa
KitchenQual (train)
KitchenQual (all_data)



In [51]:
```
all_df["BsmtExposure"] =all_data["BsmtExposure"].map(
    {nan: 0, "No": 1, "Mn": 2, "Av": 3, "Gd": 4}).astype(int)

bsmt_fin_dict = {nan: 0, "Unf": 1, "LwQ": 2, "Rec": 3, "BLQ": 4, "ALQ": 5, "GLQ": 6}
all_df["BsmtFinType1"] =all_data["BsmtFinType1"].map(bsmt_fin_dict).astype(int)
all_df["BsmtFinType2"] =all_data["BsmtFinType2"].map(bsmt_fin_dict).astype(int)

all_df["Functional"] =all_data["Functional"].map(
    {nan: 0, "Sal": 1, "Sev": 2, "Maj2": 3, "Maj1": 4,
     "Mod": 5, "Min2": 6, "Min1": 7, "Typ": 8}).astype(int)

all_df["GarageFinish"] =all_data["GarageFinish"].map(
    {nan: 0, "Unf": 1, "RFn": 2, "Fin": 3}).astype(int)

all_df["Fence"] =all_data["Fence"].map(
    {nan: 0, "MnWw": 1, "GdWo": 2, "MnPrv": 3, "GdPrv": 4}).astype(int)
```

PoolQC: NA means "No Pool"

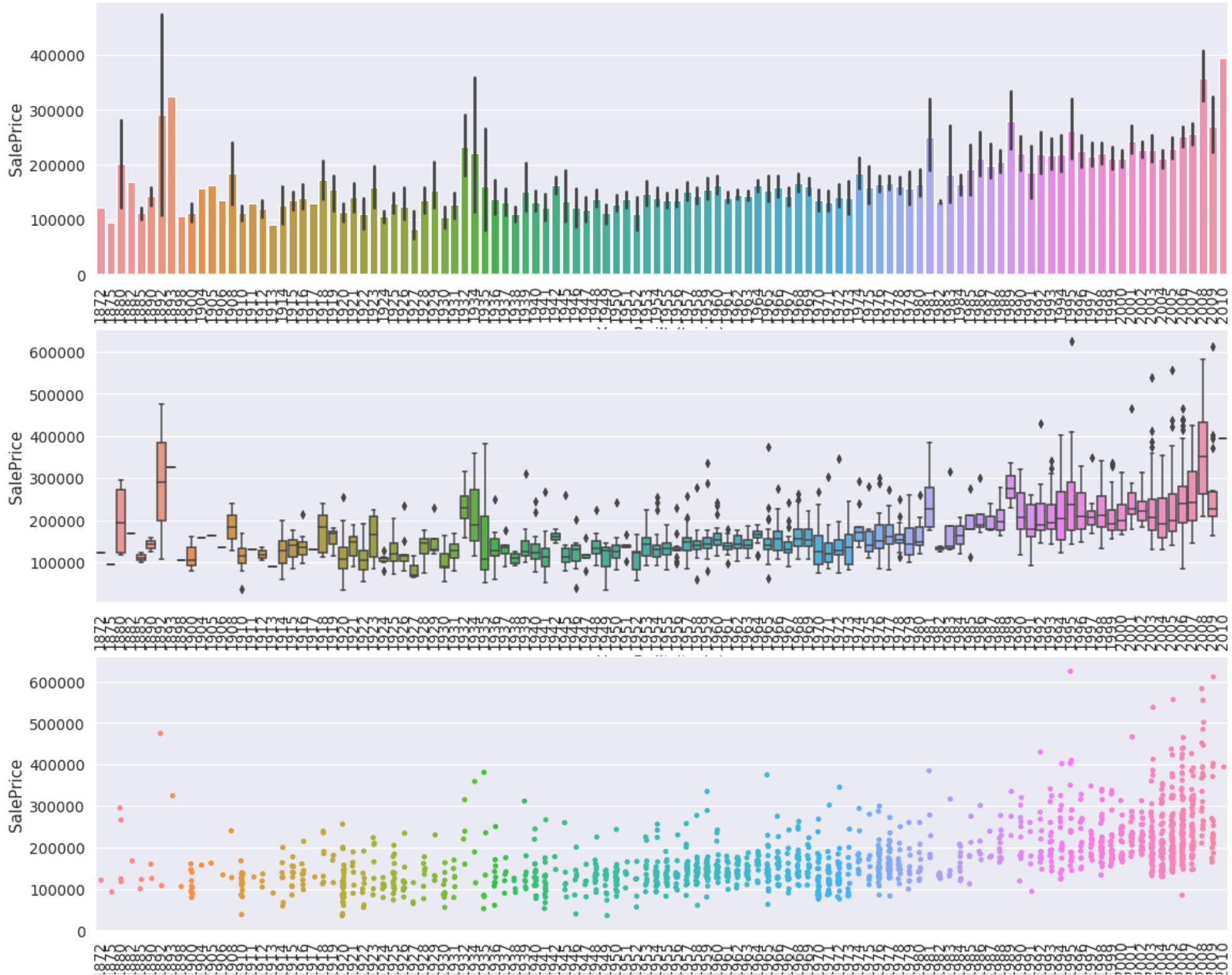
In [52]:
```
type_based_feature_analysis('PoolQC')
```

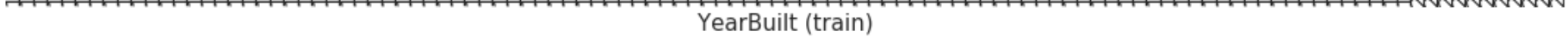


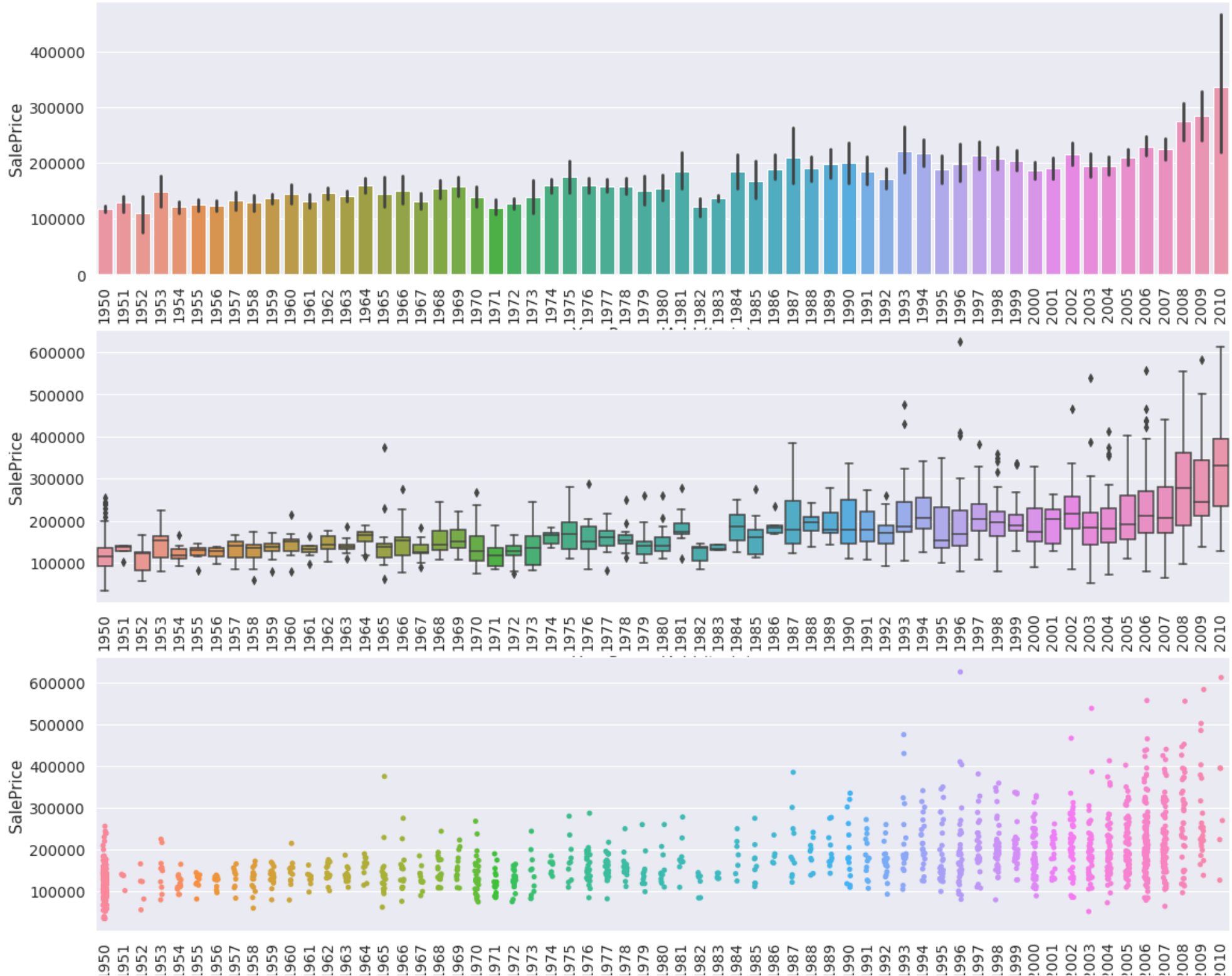
In [53]:
```
all_df["PoolQC"] =all_data["PoolQC"].map(qual_dict).astype(int)
```

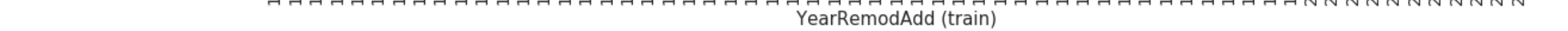
Following features are year based except MoSold so lets see how they relate to Sale price. If we use type based graph it would be easy to realize which month the selling is higher and costlier.

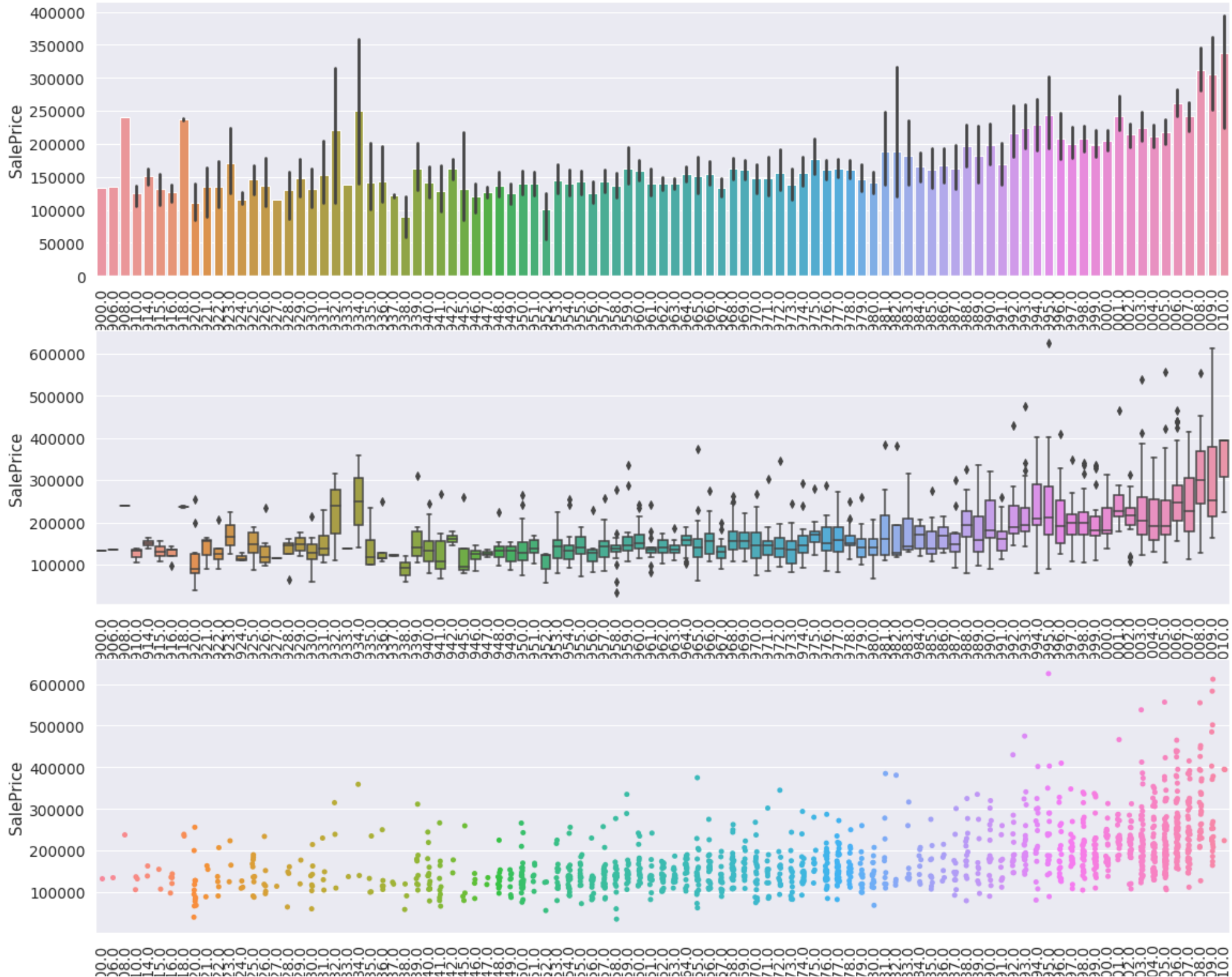
In [54]:
```
year_based_feature_analysis('YearBuilt')
year_based_feature_analysis('YearRemodAdd')
year_based_feature_analysis('GarageYrBlt')
type_based_feature_analysis('MoSold')
year_based_feature_analysis('YrSold')
```

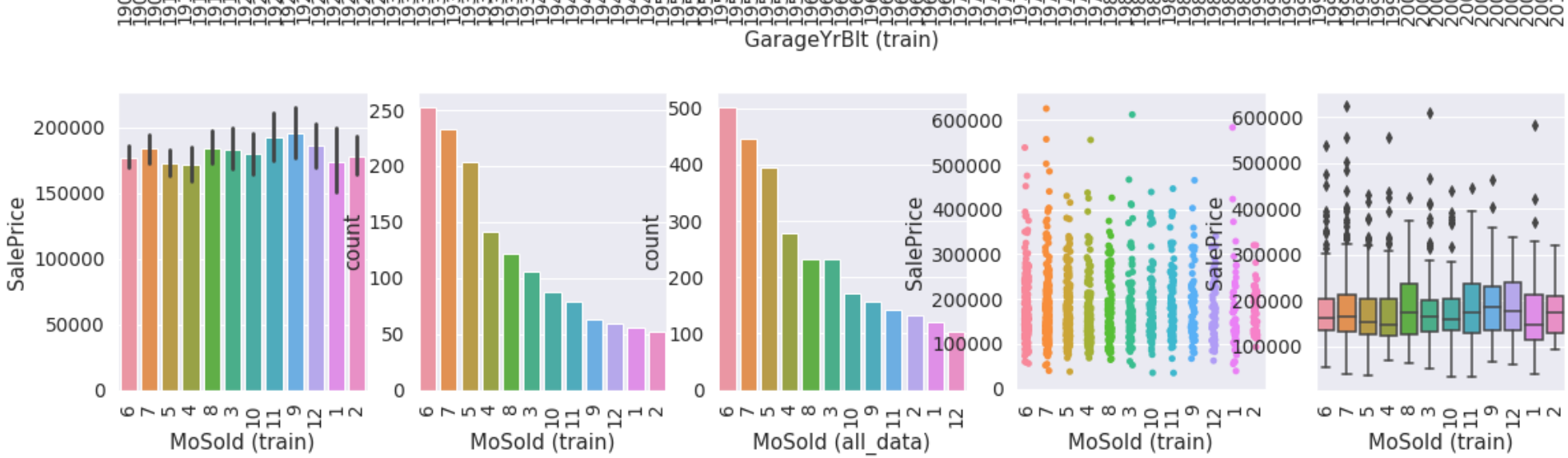

SalePrice
0
100000
200000
300000
400000
SalePrice
100000
200000
300000
400000
500000
600000
SalePrice
0
100000
200000
300000
400000
500000
600000

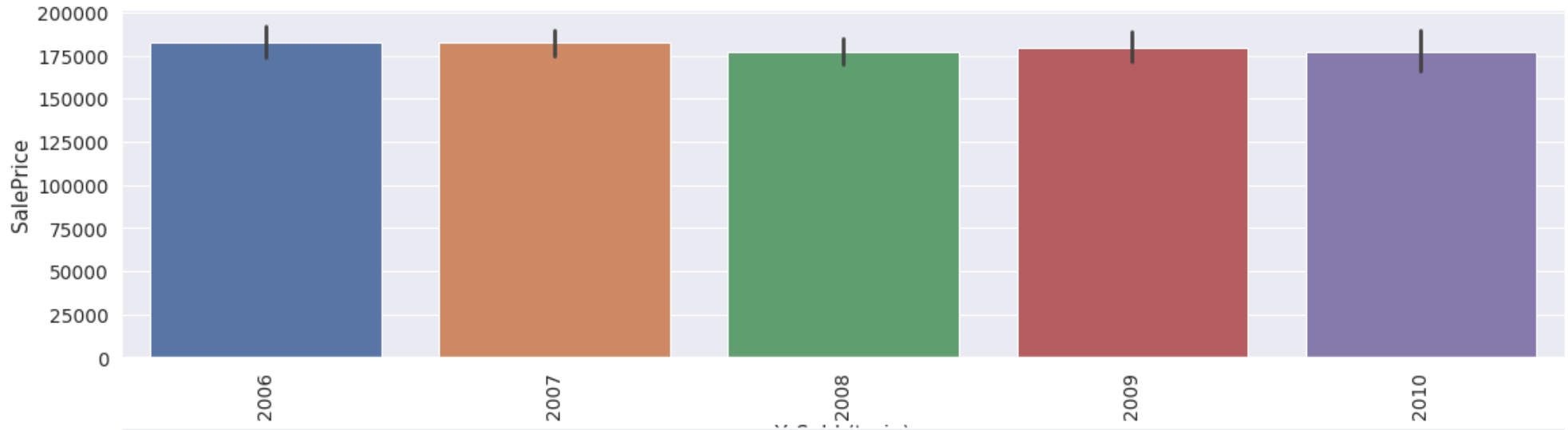

YearBuilt (train)

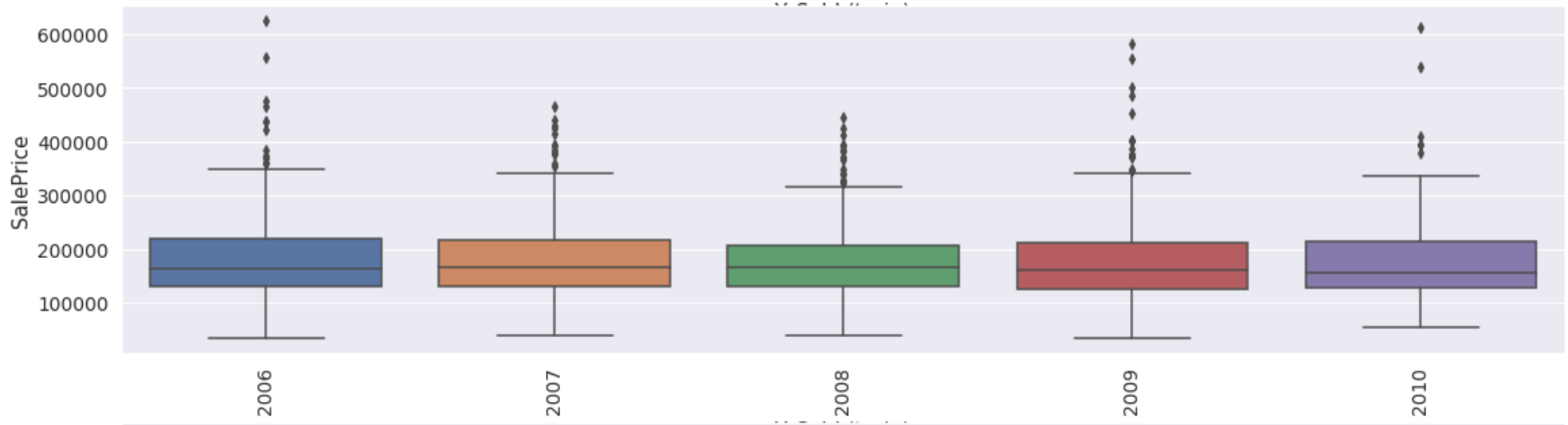

SalePrice
0
100000
200000
300000
400000
500000
600000
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010

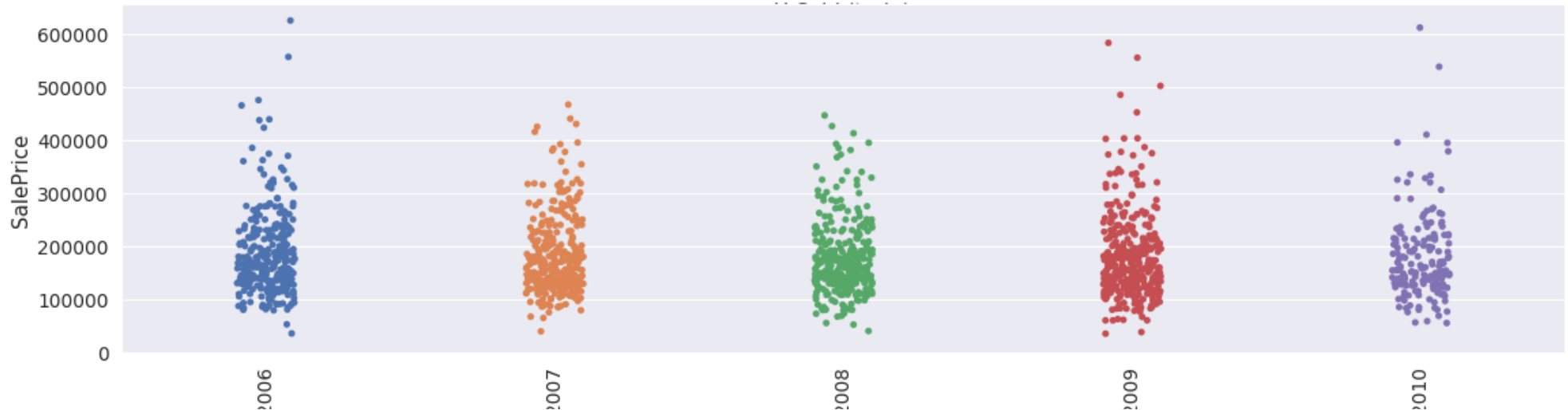

YearRemodAdd (train)


SalePrice
400000
350000
300000
250000
200000
150000
100000
50000
0
SalePrice
600000
500000
400000
300000
200000
100000
SalePrice
600000
500000
400000
300000
200000
100000


GarageYrBlt (train)
SalePrice
MoSold (train)
count
MoSold (train)
count
MoSold (all_data)
SalePrice
MoSold (train)
SalePrice
MoSold (train)


SalePrice
200000
175000
150000
125000
100000
75000
50000
25000
0
2006
2007
2008
2009
2010


SalePrice
600000
500000
400000
300000
200000
100000
2006
2007
2008
2009
2010

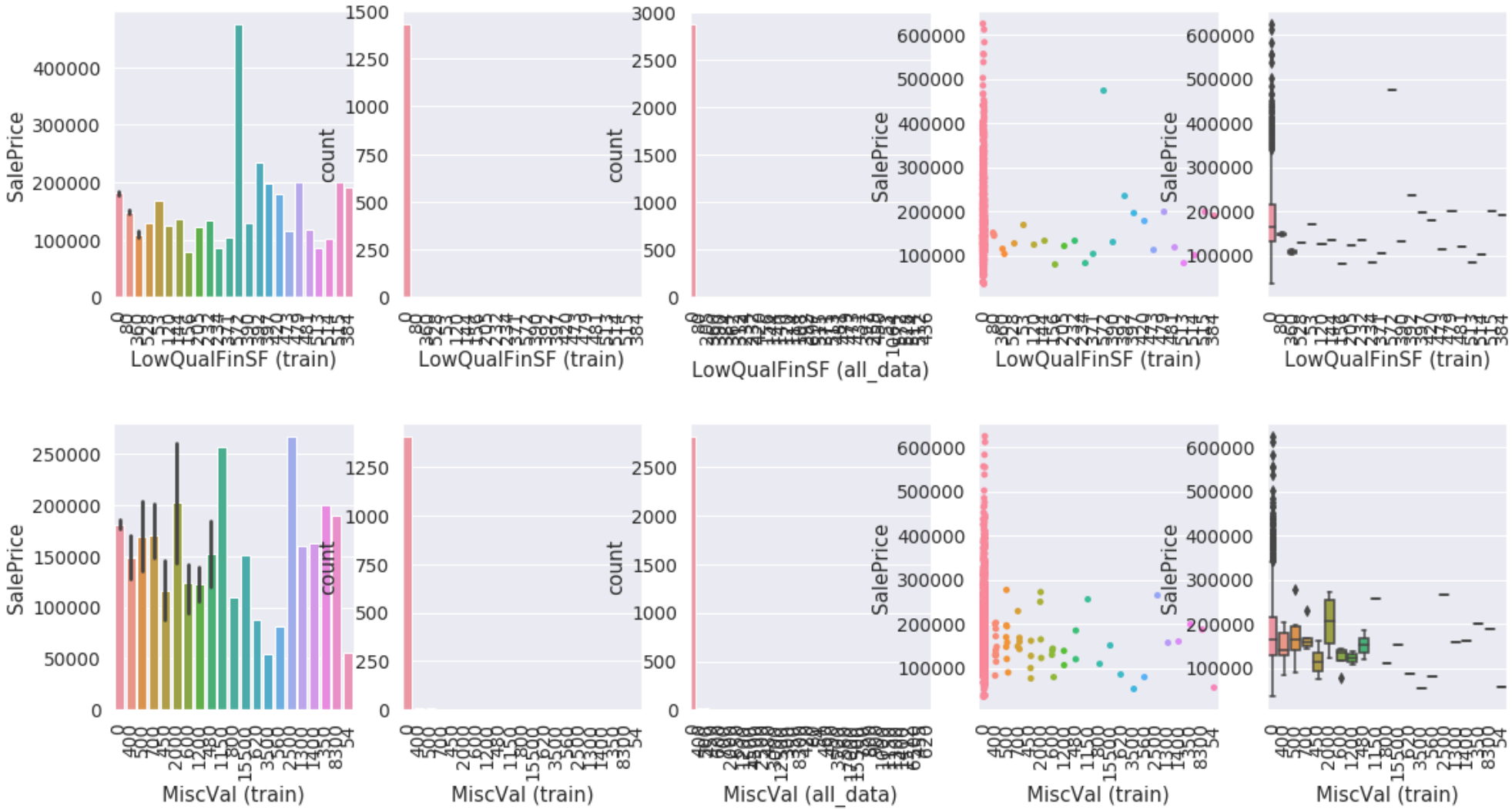

SalePrice
600000
500000
400000
300000
200000
100000
0
2006
2007
2008
2009
2010

In above section we can see that year does not matter much on Saleprice but in the month of June and July selling is higher and property is more costlier.

Now we need to change only GarageYrBlt because there is null value present in dataset. If there is no garageYrBlt then we can simply put 0 in the section of year.

In [55]:
```
all_df["YearBuilt"] =all_data["YearBuilt"]
all_df["YearRemodAdd"] =all_data["YearRemodAdd"]

all_df["GarageYrBlt"] =all_data["GarageYrBlt"]
all_df["GarageYrBlt"].fillna(0.0, inplace=True)

all_df["MoSold"] =all_data["MoSold"]
all_df["YrSold"] =all_data["YrSold"]
```

In [56]:
```
type_based_feature_analysis('LowQualFinSF')
type_based_feature_analysis('MiscVal')
```



It seems that MiscVal and LowQualFinSF is little bit skewed we will check them later when we adjust skewness. Lets keep them as they are until then.

In [57]:
```
all_df["LowQualFinSF"] =all_data["LowQualFinSF"]
all_df["MiscVal"] =all_data["MiscVal"]
```

If pool area is not available or null then that means no pool is available

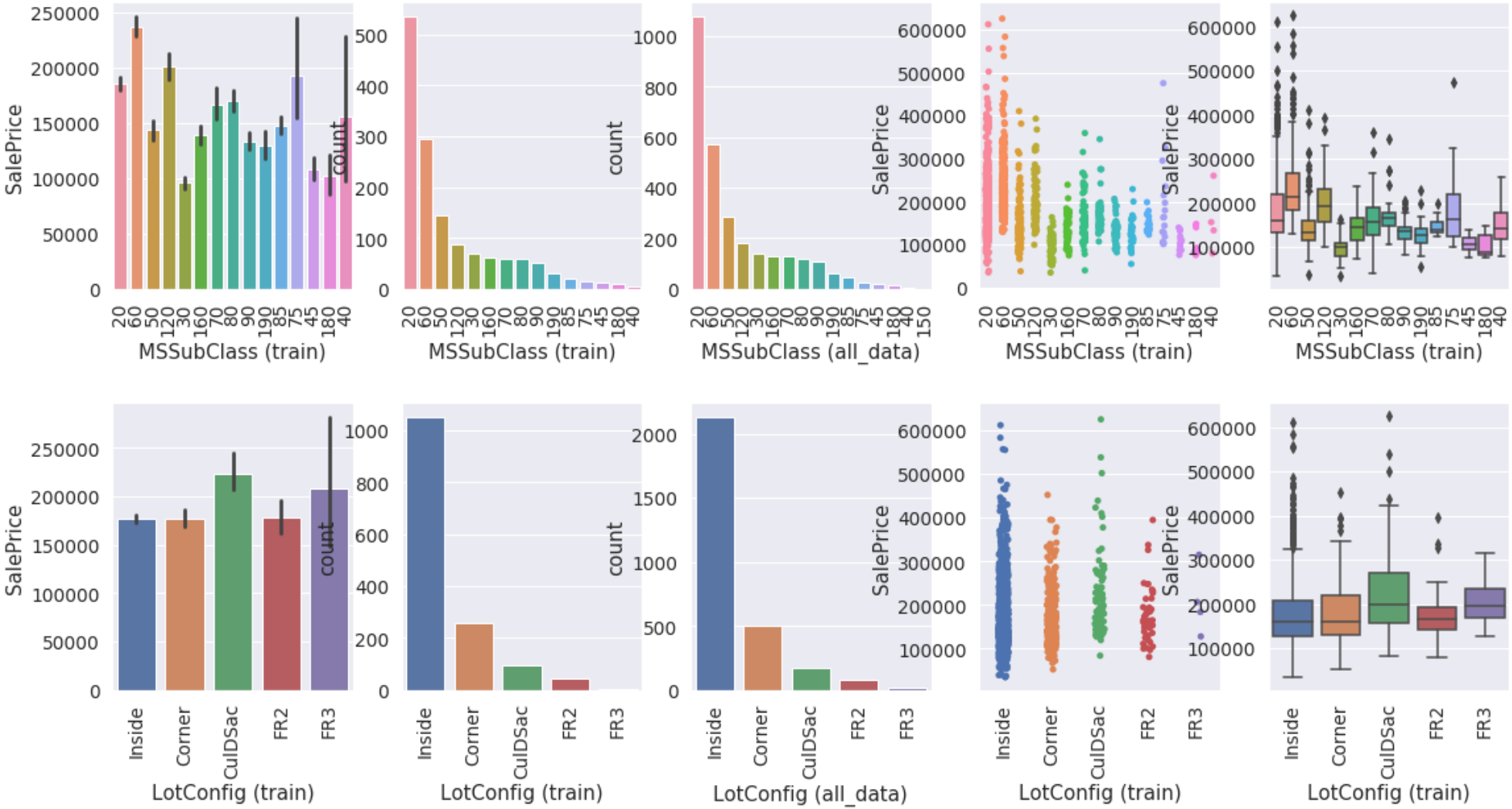
In [58]:
```
all_df["PoolQC"] =all_data["PoolQC"].map(qual_dict).astype(int)

all_df["PoolArea"] =all_data["PoolArea"]
all_df["PoolArea"].fillna(0, inplace=True)
```

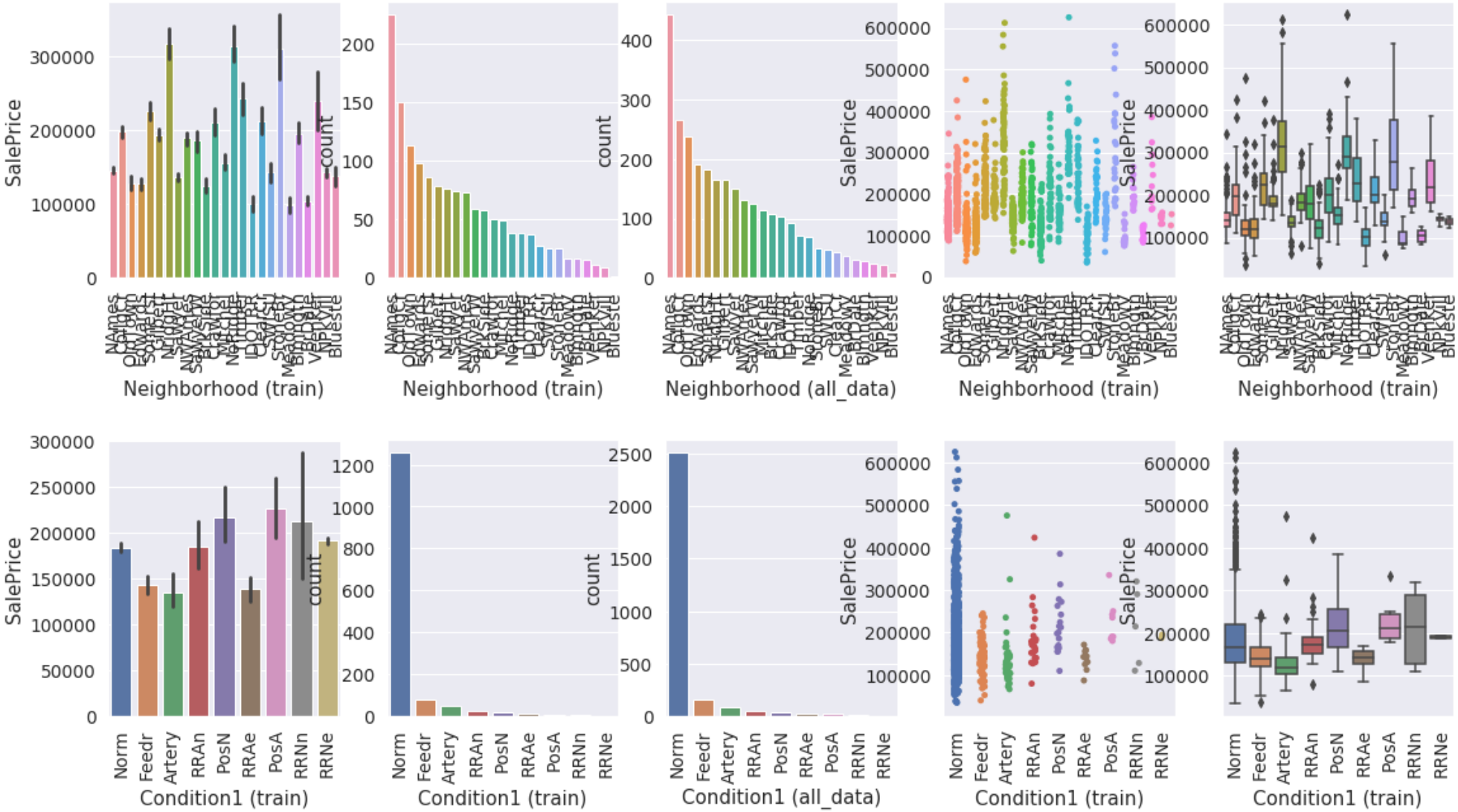
In the following section we have categorical features. For example, our first column MSSUbClass should be actually categorical, and not only that but with some hiararchy also (since there is difference whether it is 120 or 20)

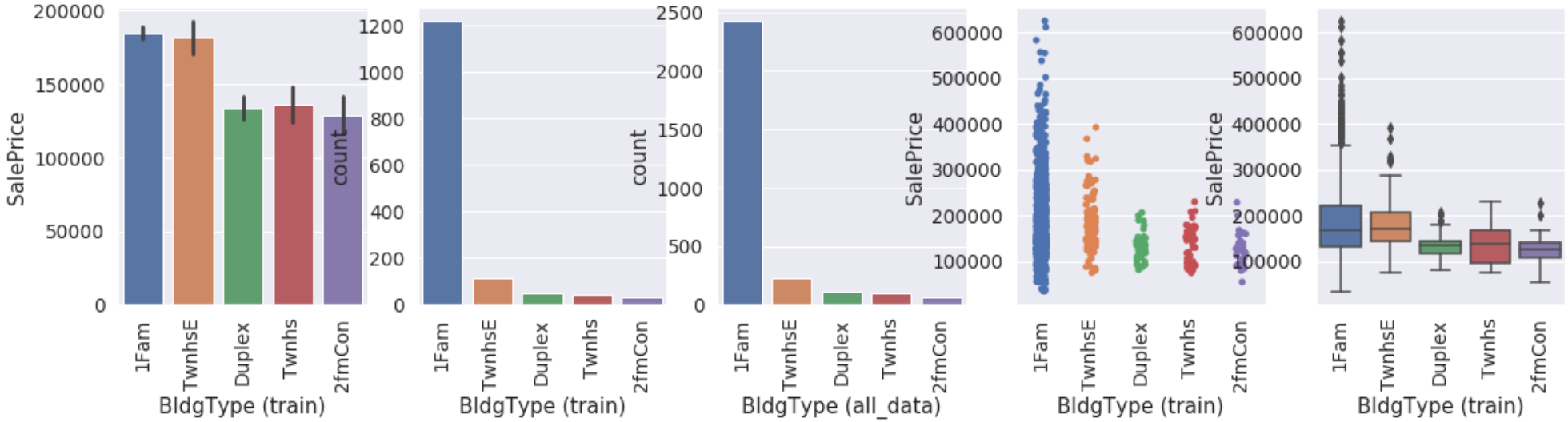
NOTE if we do not labelENCODE numerical variables BEFORE we apply dummy encoding, than these variables will never be encoded. Since dummy encoding works only on categorical variables.

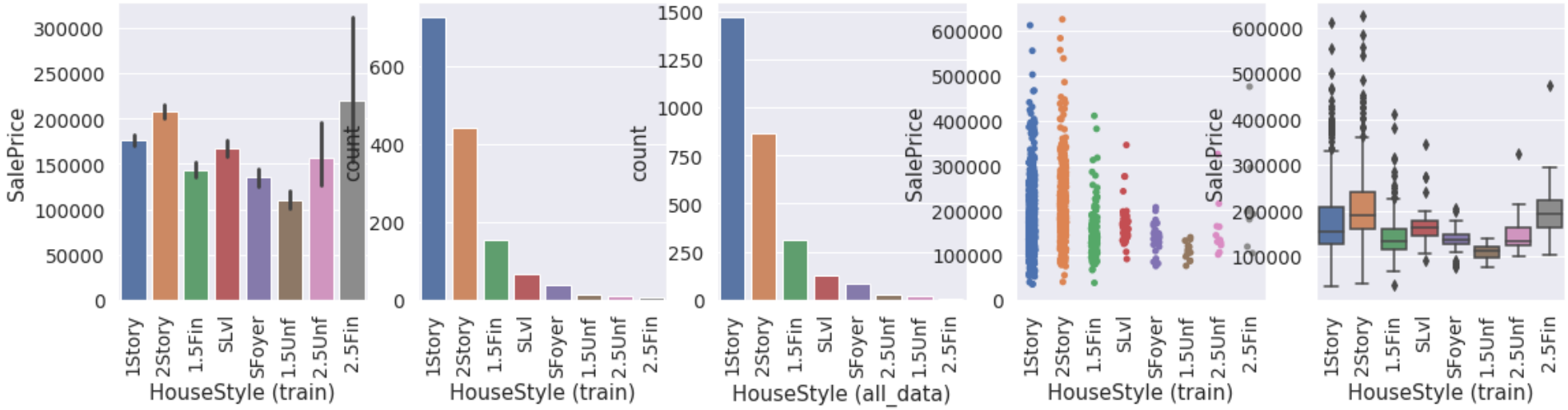
Again in factorize function we are actually filling null values and then lable encoded them.

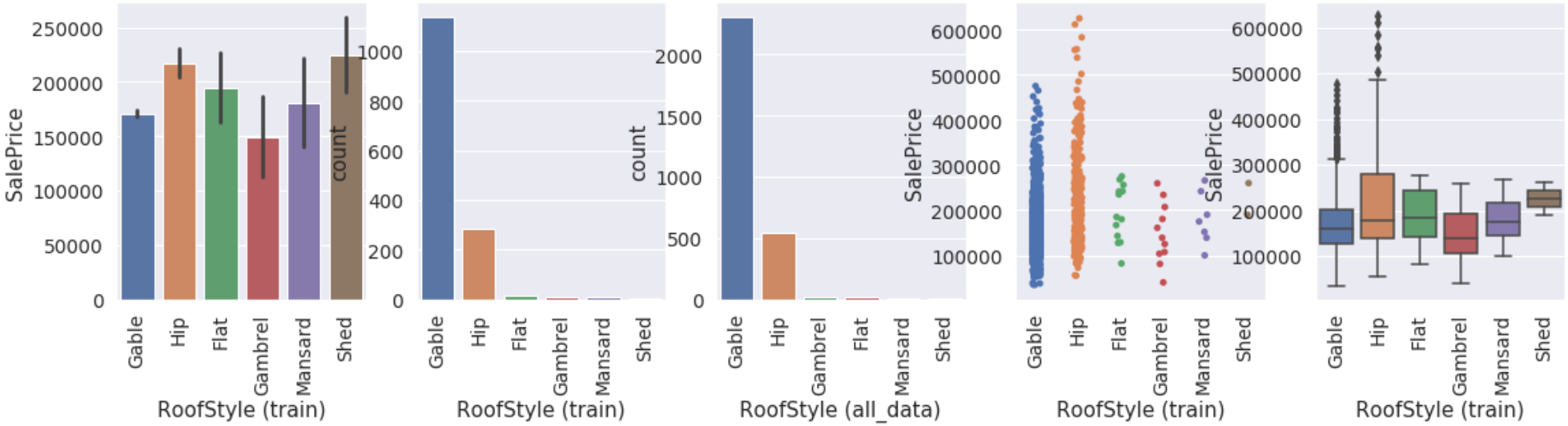
In [59]:
```
type_based_feature_analysis('MSSubClass')

type_based_feature_analysis('LotConfig')
type_based_feature_analysis('Neighborhood')
type_based_feature_analysis('Condition1')
type_based_feature_analysis('BldgType')
type_based_feature_analysis('HouseStyle')
type_based_feature_analysis('RoofStyle')

type_based_feature_analysis('Foundation')
type_based_feature_analysis('SaleCondition')
```

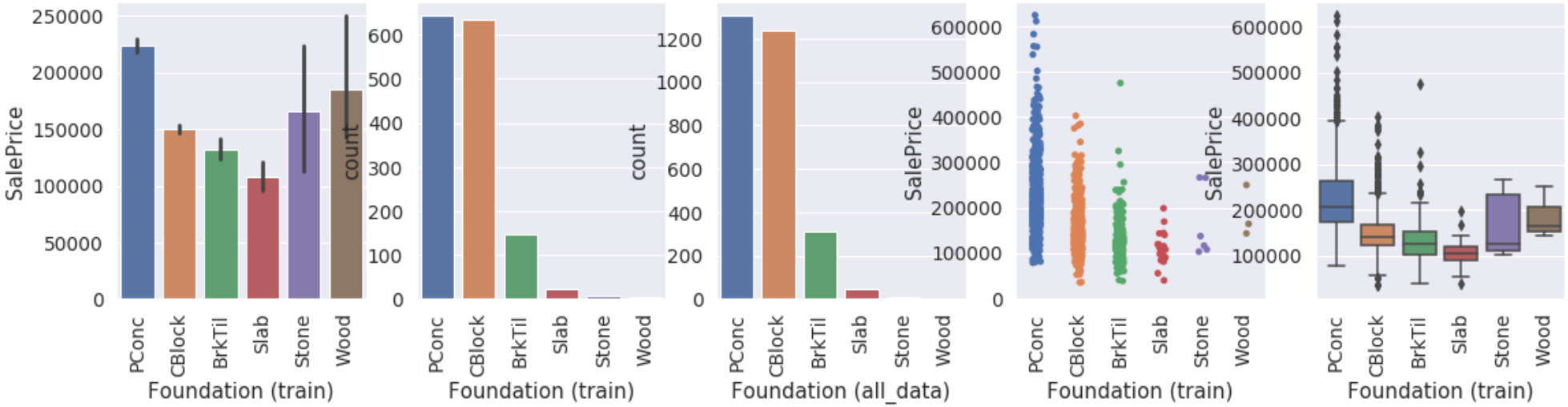

SalePrice
count
MSSubClass (train)
MSSubClass (all_data)
LotConfig (train)
LotConfig (all_data)
Inside
Corner
CulDSac
FR2
FR3

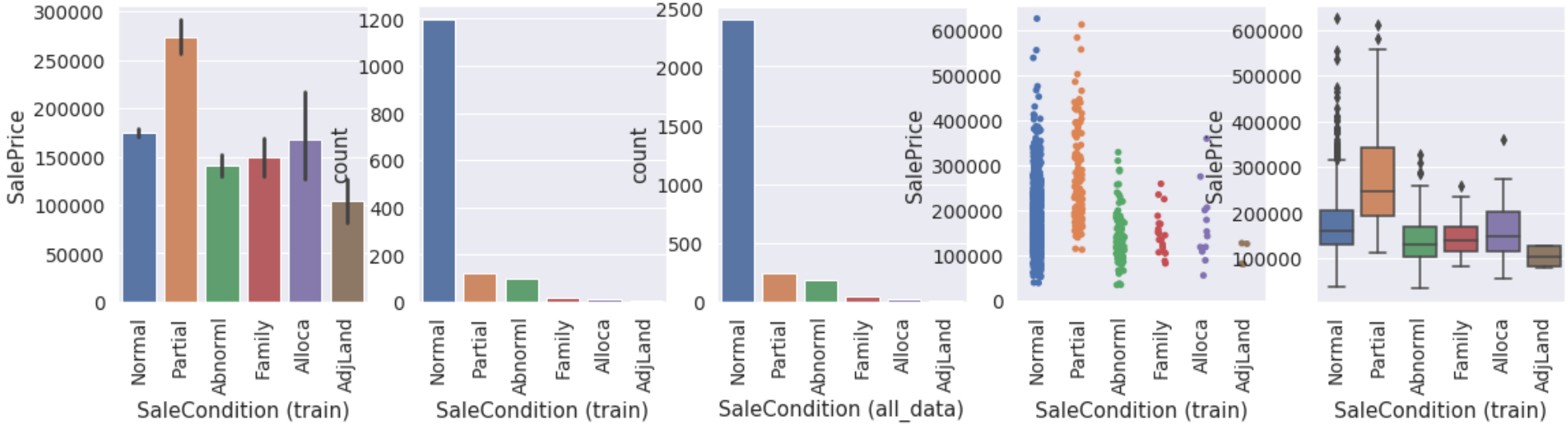

SalePrice
count
Neighborhood (train)
Neighborhood (train)
Neighborhood (all_data)
Neighborhood (train)
Neighborhood (train)
SalePrice
count
Condition1 (train)
Condition1 (train)
Condition1 (all_data)
Condition1 (train)
Condition1 (train)
Norm
Feedr
Artery
RRAn
PosN
RRAe
PosA
RRNn
RRNe


SalePrice
count
1Fam
TwnhsE
Duplex
Twnhs
2fmCon
BldgType (train)
BldgType (all_data)


SalePrice
count
1Story
2Story
1.5Fin
SLvl
SFoyer
1.5Unf
2.5Unf
2.5Fin
HouseStyle (train)
HouseStyle (all_data)

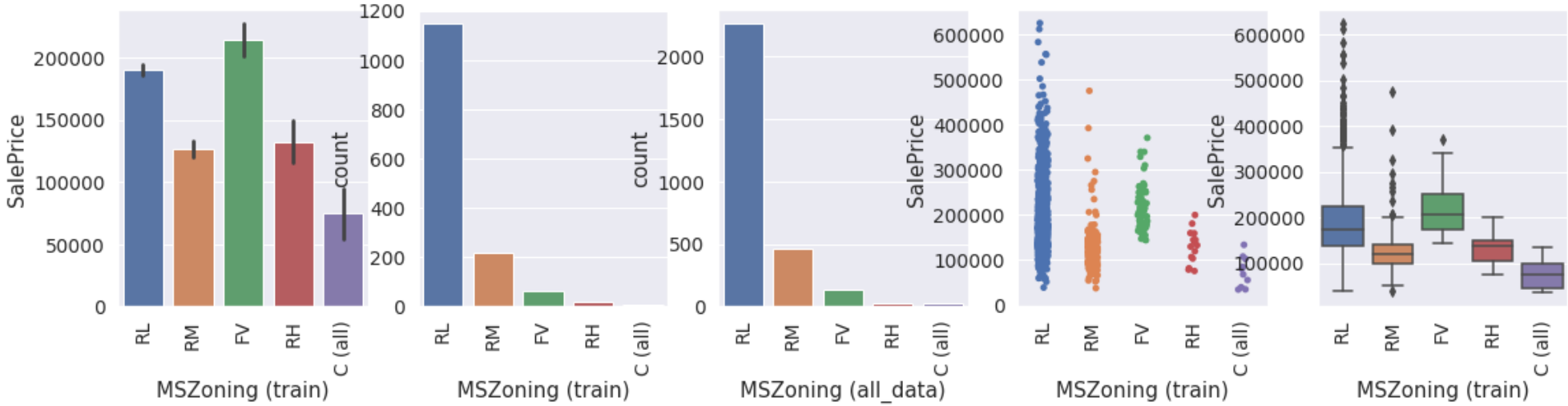

SalePrice
count
RoofStyle (train)
RoofStyle (train)
RoofStyle (all_data)
RoofStyle (train)
RoofStyle (train)
Gable
Hip
Flat
Gambrel
Mansard
Shed


SalePrice
count
Foundation (train)
Foundation (train)
Foundation (all_data)
Foundation (train)
Foundation (train)
PConc
CBlock
BrkTil
Slab
Stone
Wood



In [60]:
```
# Add categorical features as numbers too. It seems to help a bit.
all_df = factorize(all_data, all_df, "MSSubClass")

all_df = factorize(all_data, all_df, "LotConfig")
all_df = factorize(all_data, all_df, "Neighborhood")
all_df = factorize(all_data, all_df, "Condition1")
all_df = factorize(all_data, all_df, "BldgType")
all_df = factorize(all_data, all_df, "HouseStyle")
all_df = factorize(all_data, all_df, "RoofStyle")
all_df = factorize(all_data, all_df, "Foundation")
all_df = factorize(all_data, all_df, "SaleCondition")
```

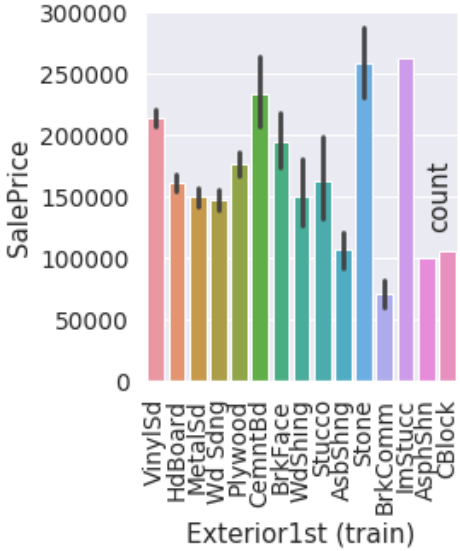
In [61]:
```
type_based_feature_analysis('MSZoning')
```

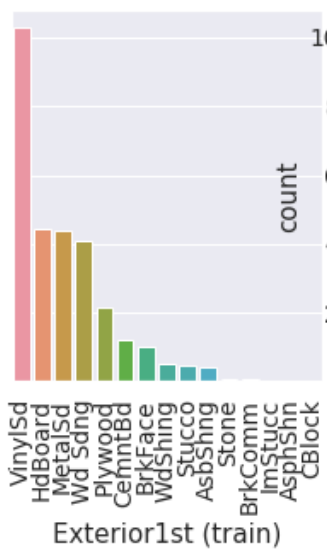


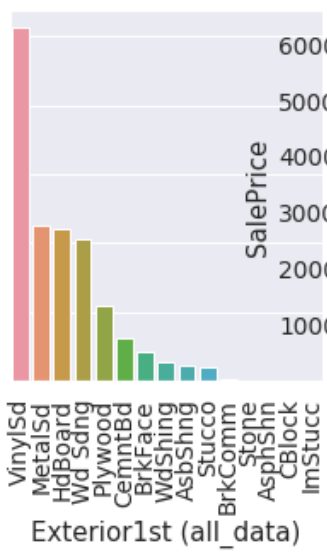
For MSZoning: since "RL" is the most common values, we are going to use mode to impute it.

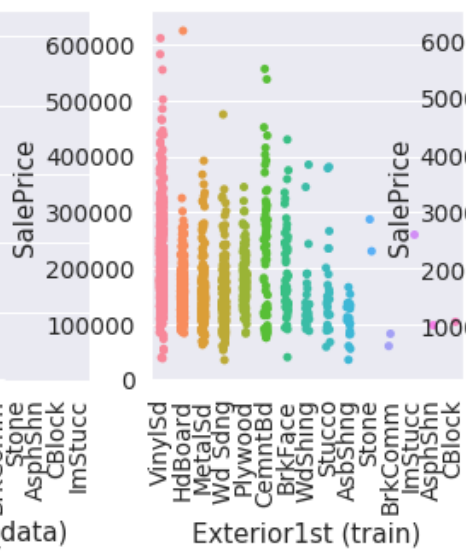
In [62]:
```
all_df = factorize(all_data, all_df, "MSZoning", "RL")
```

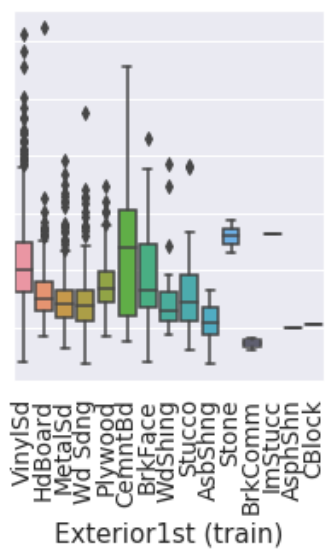
In [63]:
```
type_based_feature_analysis('Exterior1st')
type_based_feature_analysis('Exterior2nd')
type_based_feature_analysis('SaleType')
```

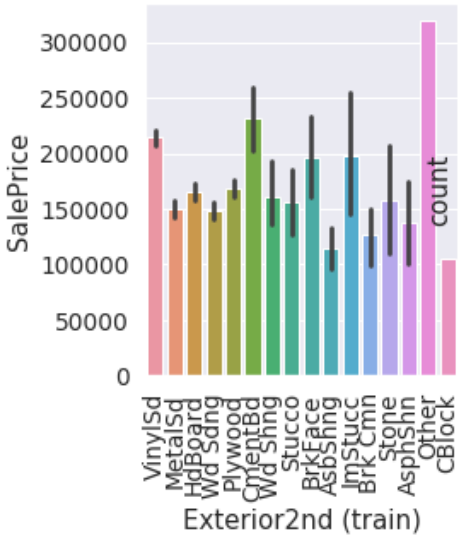

SalePrice
VinylSd
HdBoard
MetalSd
Wd Sdng
Plywood
CmentBd
BrkFace
WdShing
Stucco
AsbShng
Stone
BrkComm
ImStucc
AsphShn
CBlock
Exterior1st (train)

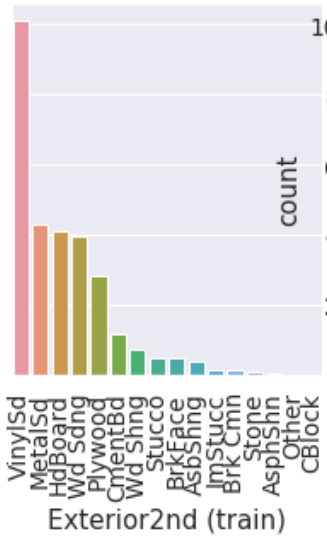

count
VinylSd
HdBoard
MetalSd
Wd Sdng
Plywood
CmentBd
BrkFace
WdShing
Stucco
AsbShng
Stone
BrkComm
ImStucc
AsphShn
CBlock
Exterior1st (train)

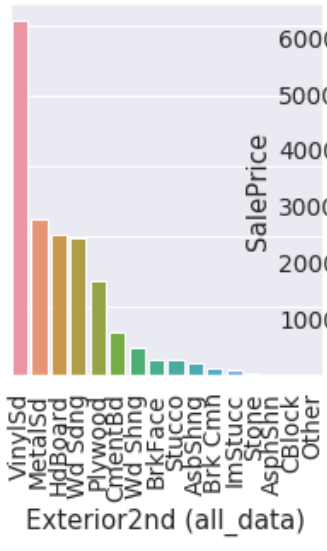

count
VinylSd
HdBoard
MetalSd
Wd Sdng
Plywood
CmentBd
BrkFace
WdShing
Stucco
AsbShng
Stone
BrkComm
ImStucc
AsphShn
CBlock
ImStucc
Exterior1st (all_data)

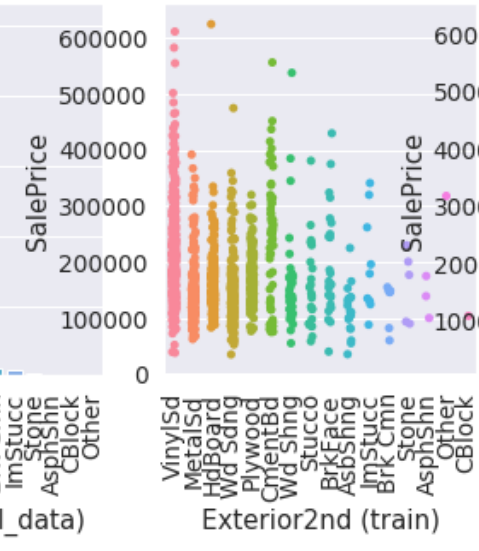

SalePrice
VinylSd
HdBoard
MetalSd
Wd Sdng
Plywood
CmentBd
BrkFace
WdShing
Stucco
AsbShng
Stone
BrkComm
ImStucc
AsphShn
CBlock
Exterior1st (train)

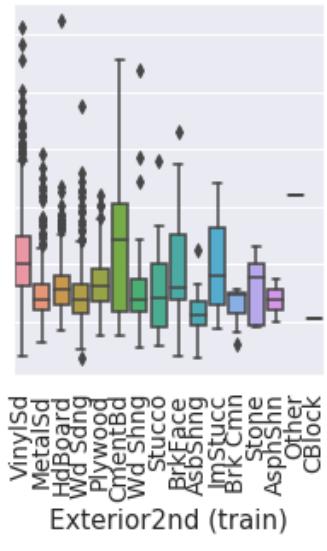

SalePrice
VinylSd
HdBoard
MetalSd
Wd Sdng
Plywood
CmentBd
BrkFace
WdShing
Stucco
AsbShng
Stone
BrkComm
ImStucc
AsphShn
CBlock
Exterior1st (train)

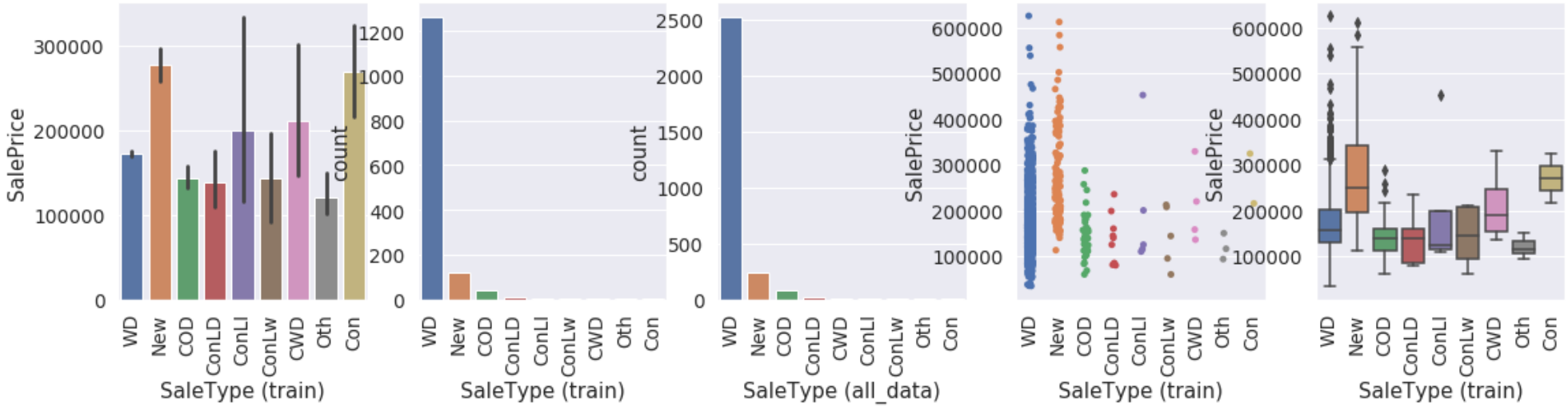

SalePrice
VinylSd
MetalSd
HdBoard
Wd Sdng
Plywood
CmentBd
Wd Shng
Stucco
BrkFace
AsbShng
ImStucc
Brk Cmn
Stone
AsphShn
Other
CBlock
Exterior2nd (train)


count
VinylSd
MetalSd
HdBoard
Wd Sdng
Plywood
CmentBd
Wd Shng
Stucco
BrkFace
AsbShng
ImStucc
Brk Cmn
Stone
AsphShn
Other
CBlock
Exterior2nd (train)


count
VinylSd
MetalSd
HdBoard
Wd Sdng
Plywood
CmentBd
Wd Shng
BrkFace
Stucco
AsbShng
ImStucc
Brk Cmn
Stone
AsphShn
CBlock
Other
Exterior2nd (all_data)


SalePrice
VinylSd
MetalSd
HdBoard
Wd Sdng
Plywood
CmentBd
Wd Shng
Stucco
BrkFace
AsbShng
ImStucc
Brk Cmn
Stone
AsphShn
Other
CBlock
Exterior2nd (train)

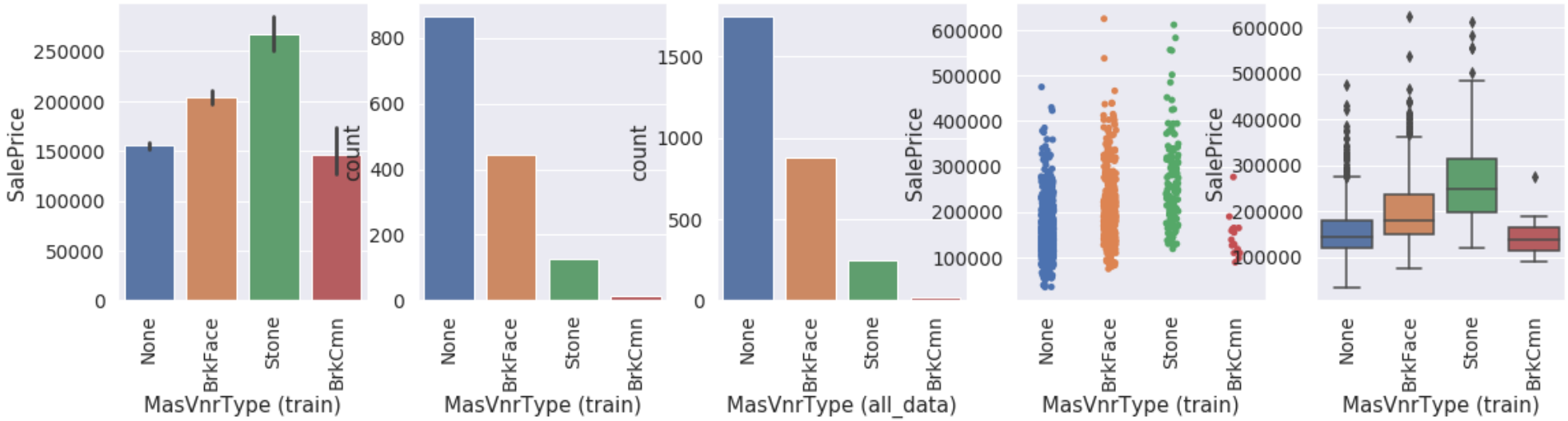

SalePrice
VinylSd
MetalSd
HdBoard
Wd Sdng
Plywood
CmentBd
Wd Shng
Stucco
BrkFace
AsbShng
ImStucc
Brk Cmn
Stone
AsphShn
Other
CBlock
Exterior2nd (train)



Exterior1st and Exterior2nd and SaleType have just one missing values (both of them are strings!) so we are just going to impute with the most common string.

In [64]:
```
all_df = factorize(all_data, all_df, "Exterior1st", "Other")
all_df = factorize(all_data, all_df, "Exterior2nd", "Other")
all_df = factorize(all_data, all_df, "SaleType", "Oth")
```

In [65]:
```
type_based_feature_analysis('MasVnrType')
```



MasVnrType: NA most likely means no masonry veneer for these houses. We can fill 0 for the area and None for the type.

In [66]:
```
all_df = factorize(all_data, all_df, "MasVnrType", "None")
```

In following code I am converting values of those features as 0 or 1

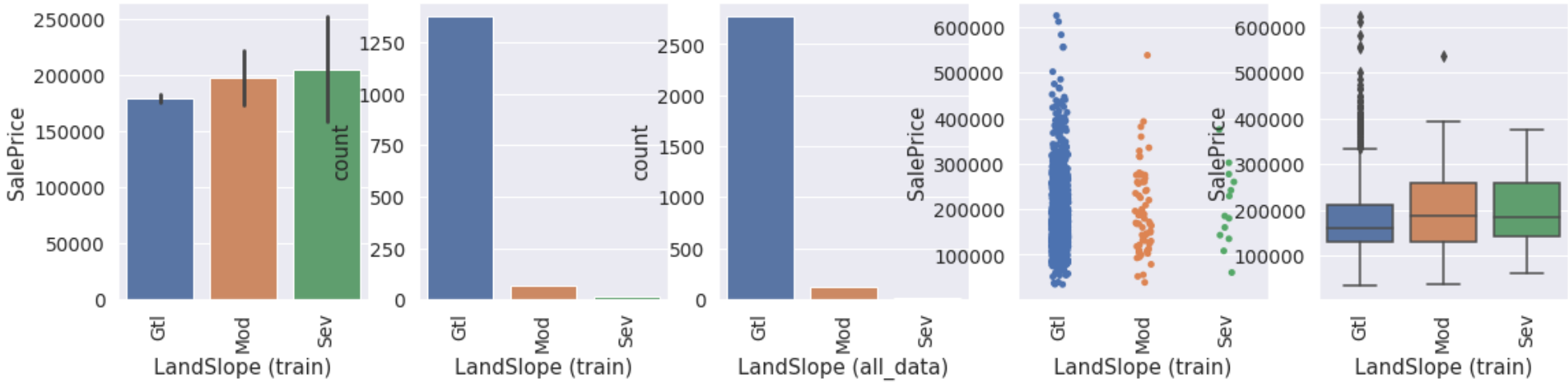
IR2 and IR3 don't appear that often, so just make a distinction between regular and irregular.

In [67]:
```
all_df["IsRegularLotShape"] = (all_data["LotShape"] == "Reg") * 1
```

Most properties are level; bin the other possibilities together as "not level".

In [68]:
```
all_df["IsLandLevel"] = (all_data["LandContour"] == "Lvl") * 1
```

In [69]:
```
type_based_feature_analysis('LandSlope')
```



Most land slopes are gentle; treat the others as "not gentle".

In [70]:
```
all_df["IsLandSlopeGentle"] = (all_data["LandSlope"] == "Gtl") * 1
```

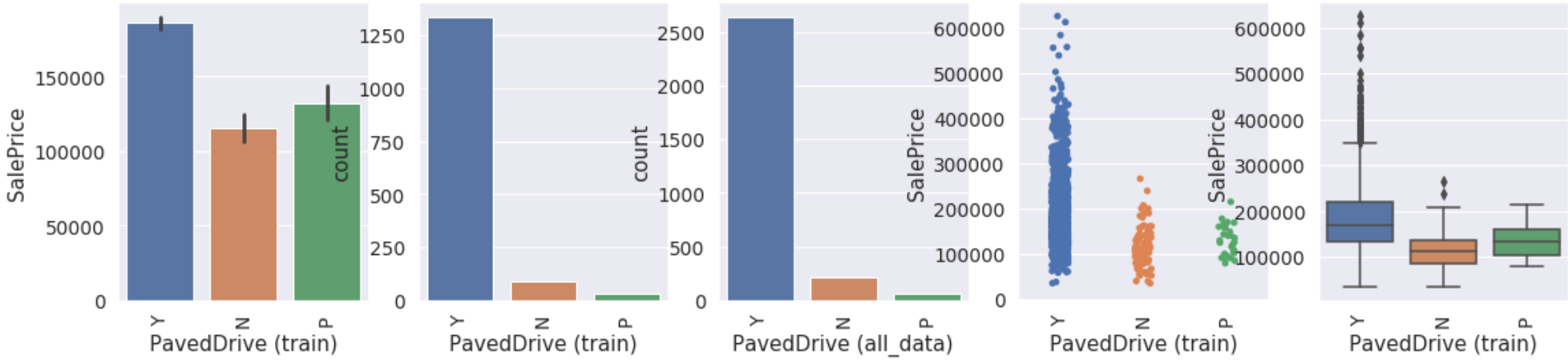
Most properties use standard circuit breakers.

In [71]:
```
all_df["IsElectricalSBrkr"] = (all_data["Electrical"] == "SBrkr") * 1
```

About 2/3rd have an attached garage.

In [72]:
```
all_df["IsGarageDetached"] = (all_data["GarageType"] == "Detchd") * 1
```

In [73]:
```
type_based_feature_analysis('PavedDrive')
```



Most have a paved drive. Treat dirt/gravel and partial pavement as "not paved".

In [74]:
```
all_df["IsPavedDrive"] = (all_data["PavedDrive"] == "Y") * 1
```

The only interesting "misc. feature" is the presence of a shed.

In [75]:
```
all_df["HasShed"] = (all_data["MiscFeature"] == "Shed") * 1.
```

If YearRemodAdd != YearBuilt, then a remodeling took place at some point.

In [76]:
```
all_df["Remodeled"] = (all_df["YearRemodAdd"] != all_df["YearBuilt"]) * 1
```

Did a remodeling happen in the year the house was sold?

In [77]:
```
all_df["RecentRemodel"] = (all_df["YearRemodAdd"] == all_df["YrSold"]) * 1
```

Was this house sold in the year it was built?

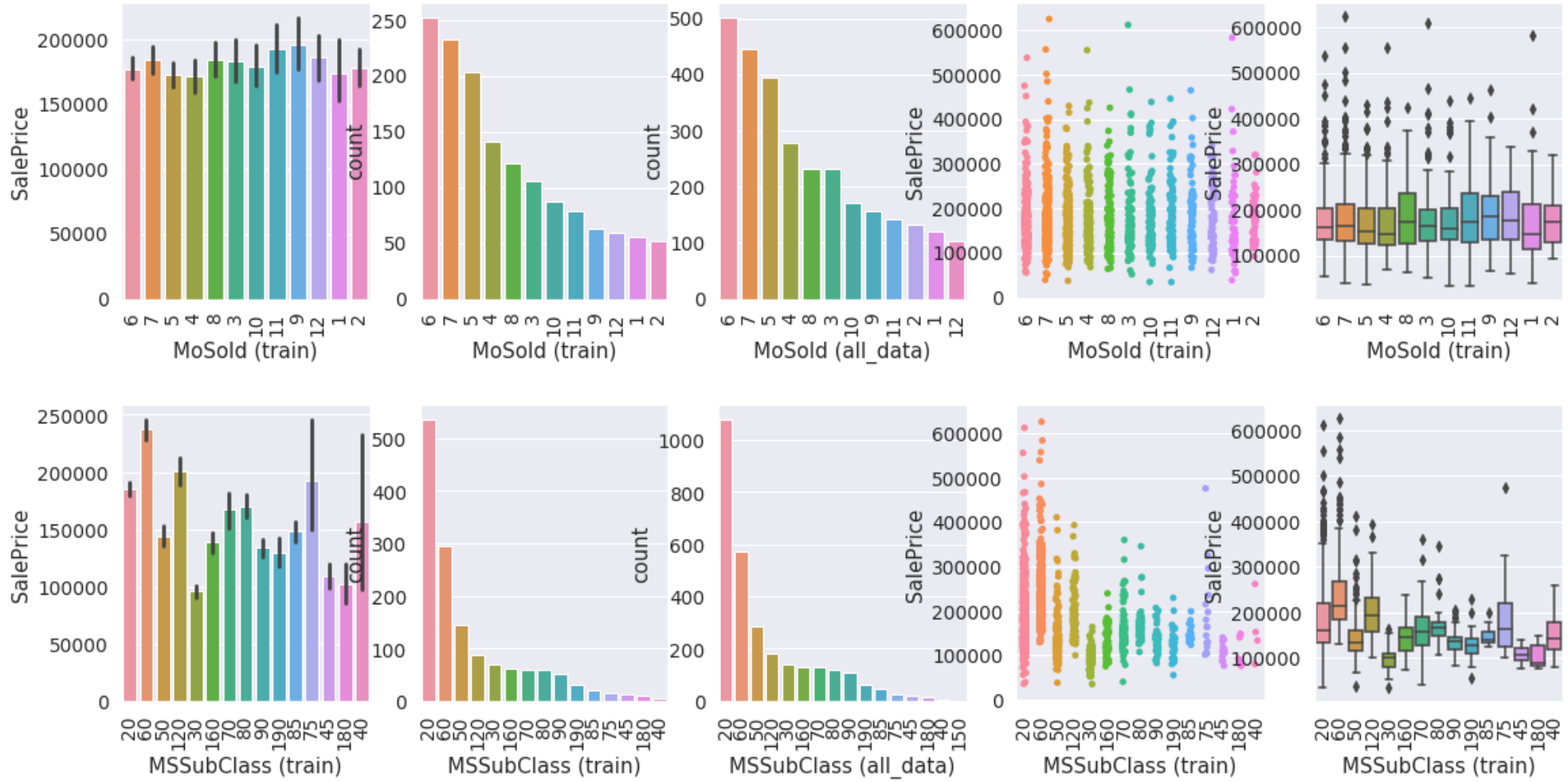
In [78]:
```
all_df["VeryNewHouse"] = (all_df["YearBuilt"] == all_df["YrSold"]) * 1
```

converting following features similarly

In [79]:
```
all_df["Has2ndFloor"] = (all_df["2ndFlrSF"] == 0) * 1
all_df["HasMasVnr"] = (all_df["MasVnrArea"] == 0) * 1
all_df["HasWoodDeck"] = (all_df["WoodDeckSF"] == 0) * 1
all_df["HasOpenPorch"] = (all_df["OpenPorchSF"] == 0) * 1
all_df["HasEnclosedPorch"] = (all_df["EnclosedPorch"] == 0) * 1
all_df["Has3SsnPorch"] = (all_df["3SsnPorch"] == 0) * 1
all_df["HasScreenPorch"] = (all_df["ScreenPorch"] == 0) * 1
```

Feature Engineering

In [80]:
```
type_based_feature_analysis('MoSold')
type_based_feature_analysis('MSSubClass')
```



Following portion was calculated with the commented part of the code and by observing graph. Instead of the fraction value putting binary value helps for generalization.

We can see that most of the selling happens in April, may , june and July. For MSSubClass 20 , 60, 120 is most costly and we put 1 for them.

In [81]:
```
# Months with the largest number of deals may be significant.
#     mx = max(train["MoSold"].groupby(train["MoSold"]).count())
#     all_df["HighSeason"] =all_data["MoSold"].replace(
#         train["MoSold"].groupby(train["MoSold"]).count()/mx)

#     mx = max(train["MSSubClass"].groupby(train["MSSubClass"]).count())
#     all_df["NewerDwelling"] =all_data["MSSubClass"].replace(
#         train["MSSubClass"].groupby(train["MSSubClass"]).count()/mx)


all_df["HighSeason"] =all_data["MoSold"].replace(
    {1: 0, 2: 0, 3: 0, 4: 1, 5: 1, 6: 1, 7: 1, 8: 0, 9: 0, 10: 0, 11: 0, 12: 0})

all_df["NewerDwelling"] =all_data["MSSubClass"].replace(
    {20: 1, 30: 0, 40: 0, 45: 0,50: 0, 60: 1, 70: 0, 75: 0, 80: 0, 85: 0,
     90: 0, 120: 1, 150: 0, 160: 0, 180: 0, 190: 0})
```

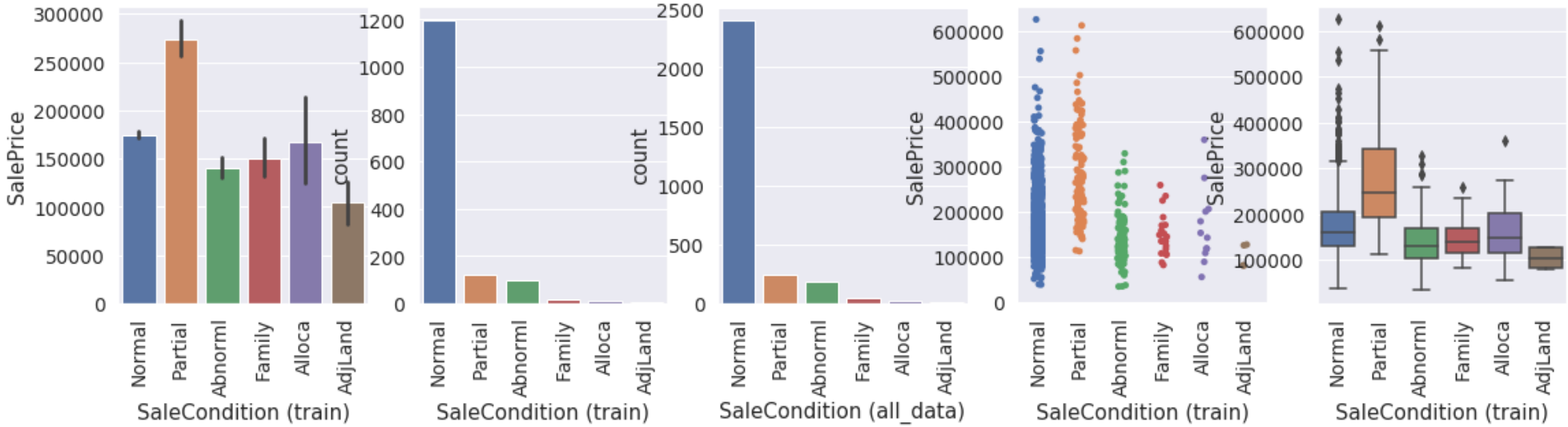
In [82]:
```
year_based_feature_analysis('Neighborhood')
```


SalePrice
0
50000
100000
150000
200000
250000
300000
350000
SalePrice
100000
200000
300000
400000
500000
600000
SalePrice
0
100000
200000
300000
400000
500000
600000

According to graph we put top 5 costly place to neighborhood as good place to live in. And make others zero.

In [83]:
```
all_df.loc[all_data.Neighborhood == 'NridgHt', "Neighborhood_Good"] = 1
all_df.loc[all_data.Neighborhood == 'Crawfor', "Neighborhood_Good"] = 1
all_df.loc[all_data.Neighborhood == 'StoneBr', "Neighborhood_Good"] = 1
all_df.loc[all_data.Neighborhood == 'Somerst', "Neighborhood_Good"] = 1
all_df.loc[all_data.Neighborhood == 'NoRidge', "Neighborhood_Good"] = 1
all_df["Neighborhood_Good"].fillna(0, inplace=True)
```

In [84]:
```
type_based_feature_analysis('SaleCondition')
```



House completed before sale or not if partially completed then put 0.

In [85]:
```
# House completed before sale or not
all_df["SaleCondition_PriceDown"] =all_data.SaleCondition.replace(
    {'Abnorml': 1, 'Alloca': 1, 'AdjLand': 1, 'Family': 1, 'Normal': 0, 'Partial': 0})
```

In [86]:
```
# House completed before sale or not
all_df["BoughtOffPlan"] =all_data.SaleCondition.replace(
    {"Abnorml" : 0, "Alloca" : 0, "AdjLand" : 0, "Family" : 0, "Normal" : 0, "Partial" : 1})

all_df["BadHeating"] =all_data.HeatingQC.replace(
    {'Ex': 0, 'Gd': 0, 'TA': 0, 'Fa': 1, 'Po': 1})
```

Total area covered by the property is TotalArea and total liveable place is tatalara1st2nd . These features are here because they are more normal and thus easy to work with.

In [87]:
```
area_cols = ['LotFrontage', 'LotArea', 'MasVnrArea', 'BsmtFinSF1', 'BsmtFinSF2', 'BsmtUnfSF',
             'TotalBsmtSF', '1stFlrSF', '2ndFlrSF', 'GrLivArea', 'GarageArea', 'WoodDeckSF',
             'OpenPorchSF', 'EnclosedPorch', '3SsnPorch', 'ScreenPorch', 'LowQualFinSF', 'PoolArea' ]
all_df["TotalArea"] = all_df[area_cols].sum(axis=1)

all_df["TotalArea1st2nd"] = all_df["1stFlrSF"] + all_df["2ndFlrSF"]
```

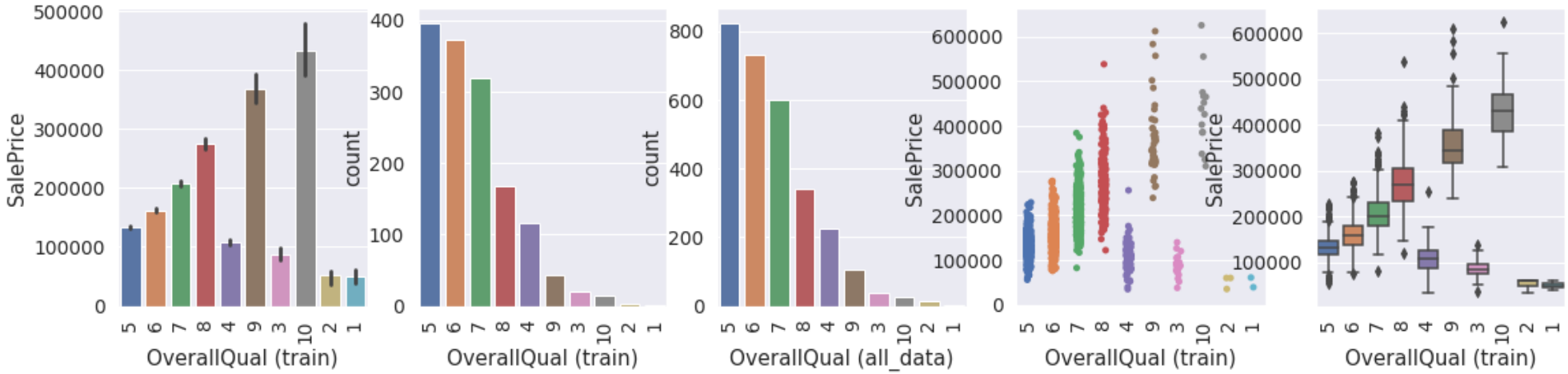
Price usually drops when property gets old so generating this feature might help.

In [88]:
```
all_df["Age"] = 2010 - all_df["YearBuilt"]
all_df["TimeSinceSold"] = 2010 - all_df["YrSold"]

all_df["SeasonSold"] = all_df["MoSold"].map({12:0, 1:0, 2:0, 3:1, 4:1, 5:1,
                                             6:2, 7:2, 8:2, 9:3, 10:3, 11:3}).astype(int)

all_df["YearsSinceRemodel"] = all_df["YrSold"] - all_df["YearRemodAdd"]
```

Simplifications of existing features into bad/average/good base on the graphs.

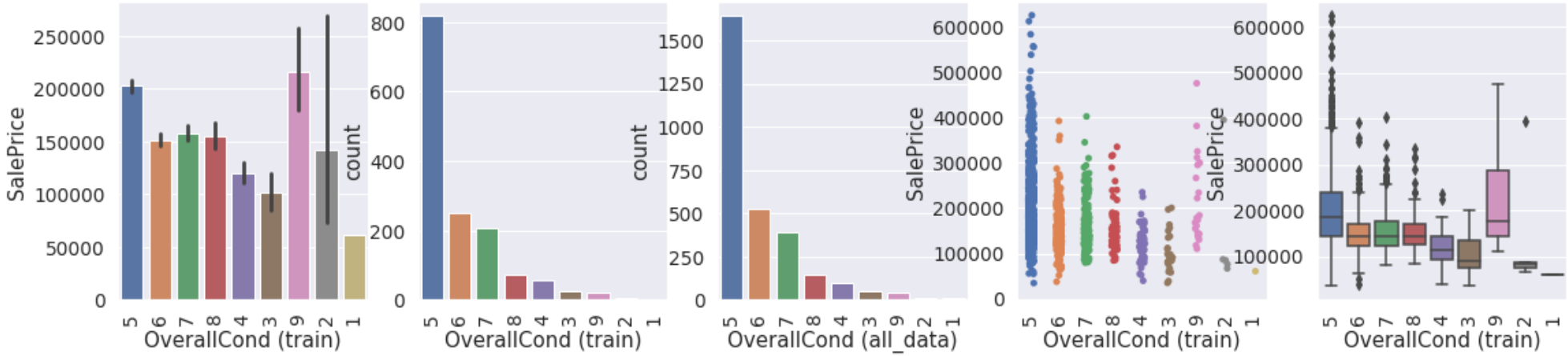
In this section I have created some new feature based on my understanding of the graph. After this simplification I have found that my score in kaggle improved a lot. This section is done through graph analysis only so no explanation added due to that reason.

In [89]:
```
type_based_feature_analysis('OverallQual')
```



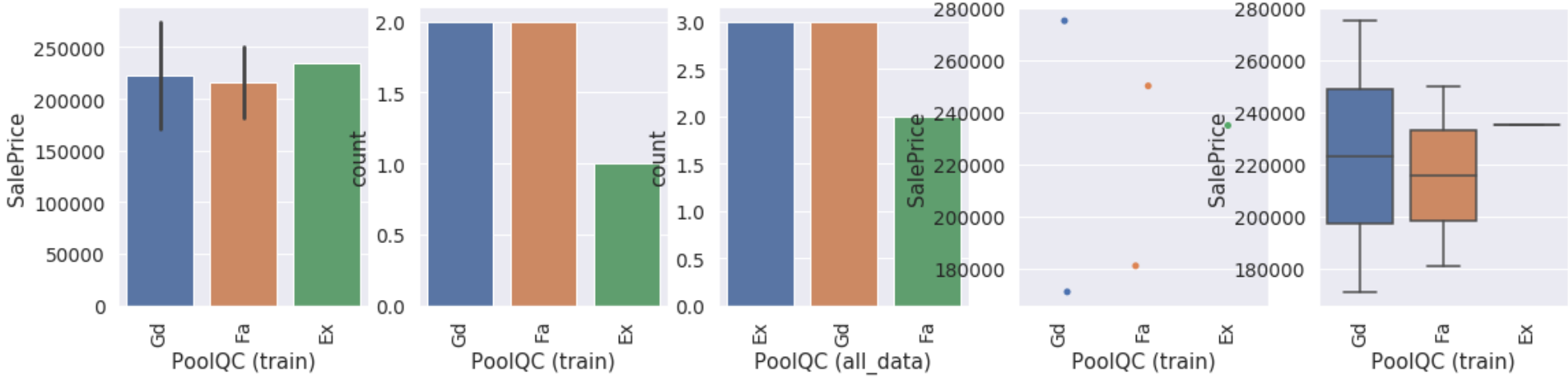
In [90]:
```
all_df["SimplOverallQual"] = all_df.OverallQual.replace(
    {1 : 1, 2 : 1, 3 : 1, 4 : 2, 5 : 2, 6 : 2, 7 : 3, 8 : 3, 9 : 3, 10 : 3})
```

In [91]:
```
type_based_feature_analysis('OverallCond')
```



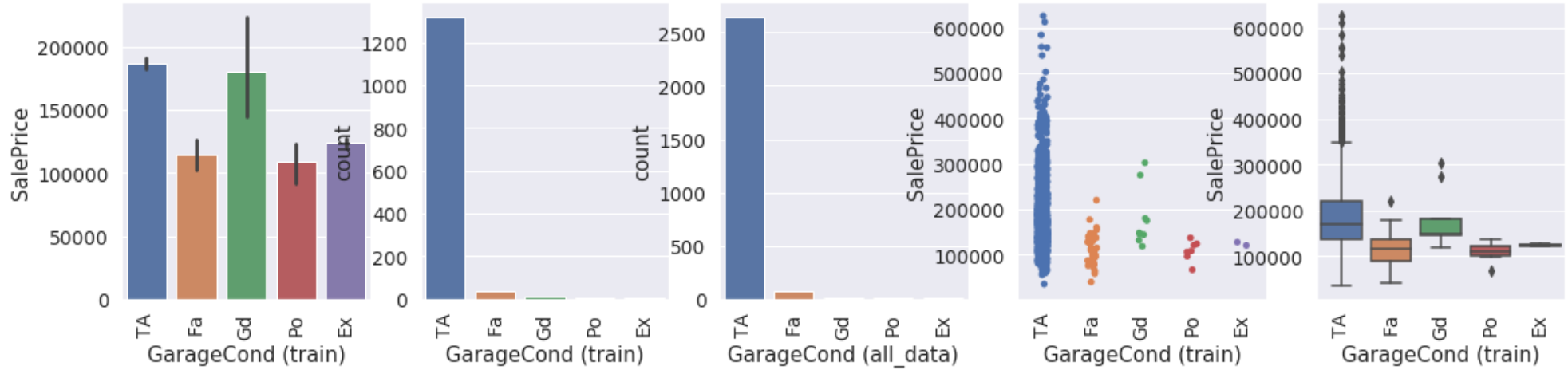
In [92]:
```
all_df["SimplOverallCond"] = all_df.OverallCond.replace(
    {1 : 1, 2 : 1, 3 : 1, 4 : 2, 5 : 2, 6 : 2, 7 : 3, 8 : 3, 9 : 3, 10 : 3})
```

In [93]:
```
type_based_feature_analysis('PoolQC')
```



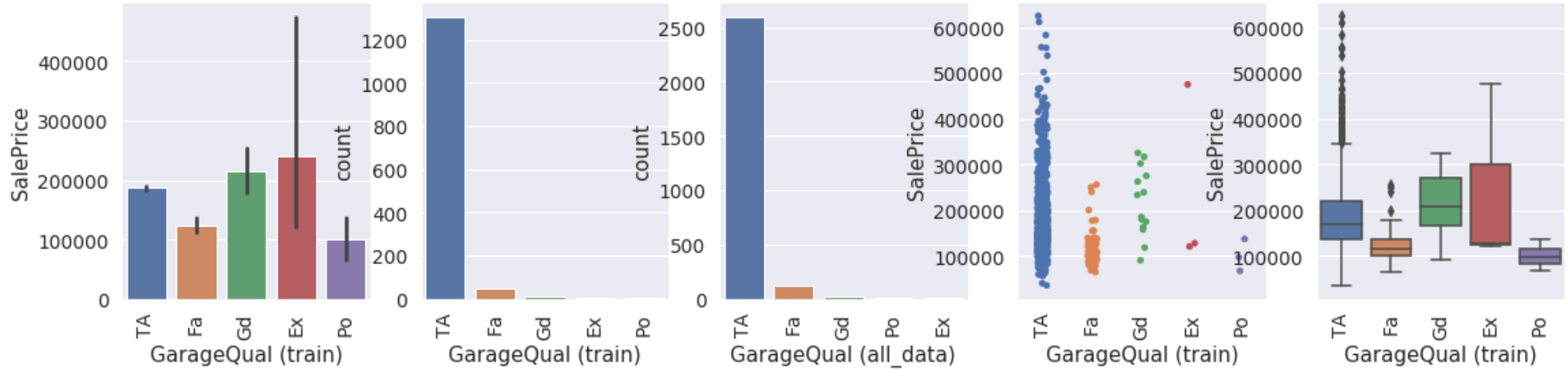
In [94]:
```
all_df["SimplPoolQC"] = all_df.PoolQC.replace(
    {1 : 1, 2 : 1, 3 : 2, 4 : 2})
```

In [95]:
```
type_based_feature_analysis('GarageCond')
```



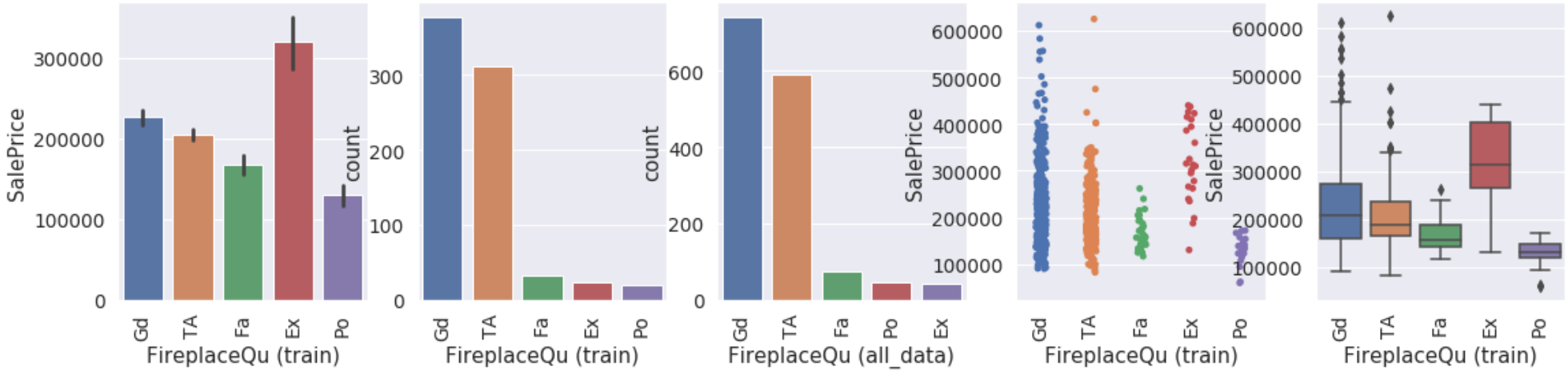
In [96]:
```
all_df["SimplGarageCond"] = all_df.GarageCond.replace(
    {1 : 1, 2 : 1, 3 : 1, 4 : 2, 5 : 2})
```

In [97]:
```
type_based_feature_analysis('GarageQual')
```



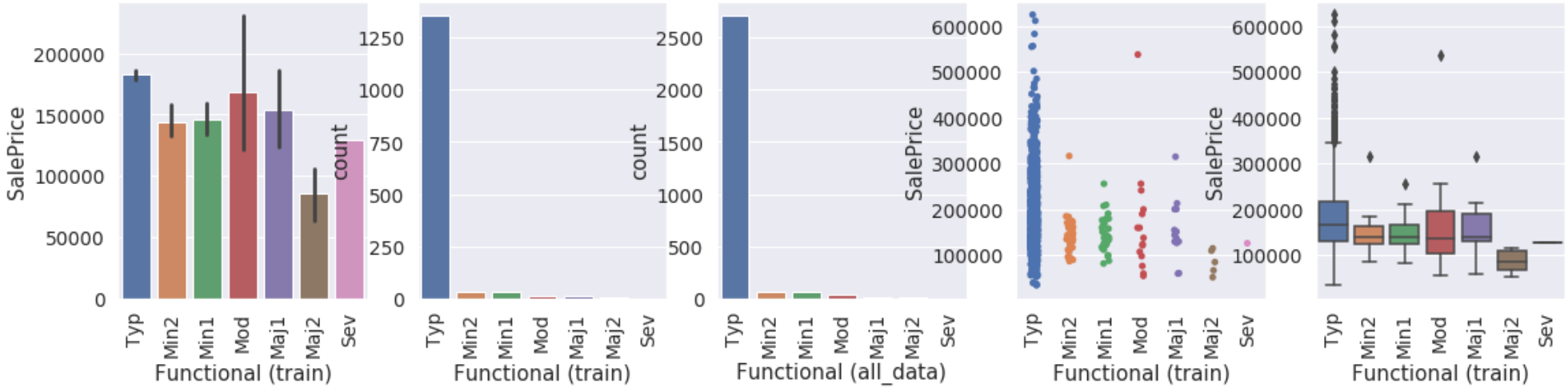
In [98]:
```
all_df["SimplGarageQual"] = all_df.GarageQual.replace(
    {1 : 1, 2 : 1, 3 : 1, 4 : 2, 5 : 2})
```

In [99]:
```
type_based_feature_analysis('FireplaceQu')
```



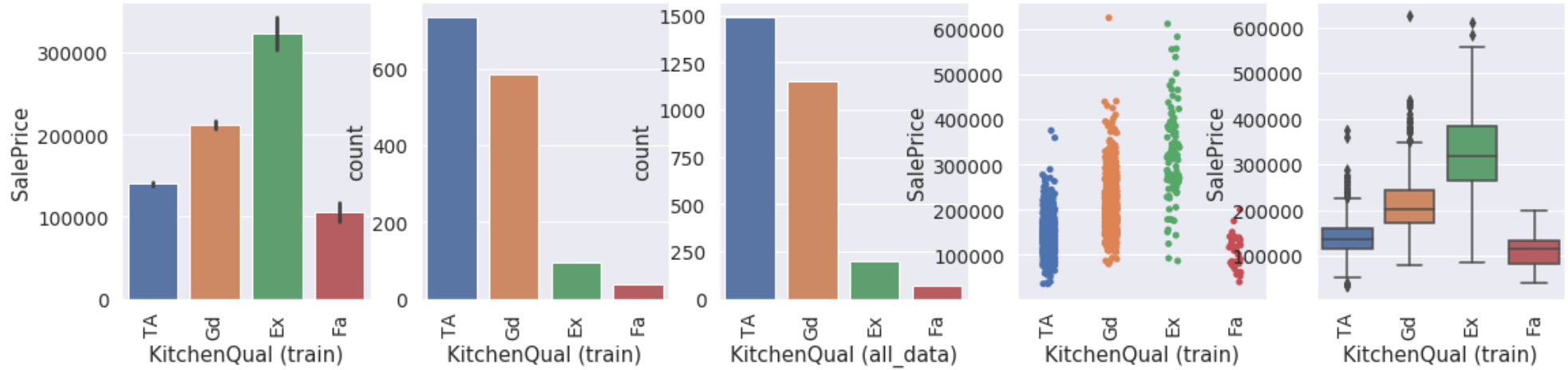
In [100]:
```
all_df["SimplFireplaceQu"] = all_df.FireplaceQu.replace(
    {1 : 1, 2 : 1, 3 : 1, 4 : 2, 5 : 2})
```

In [101]:
```
type_based_feature_analysis('Functional')
```



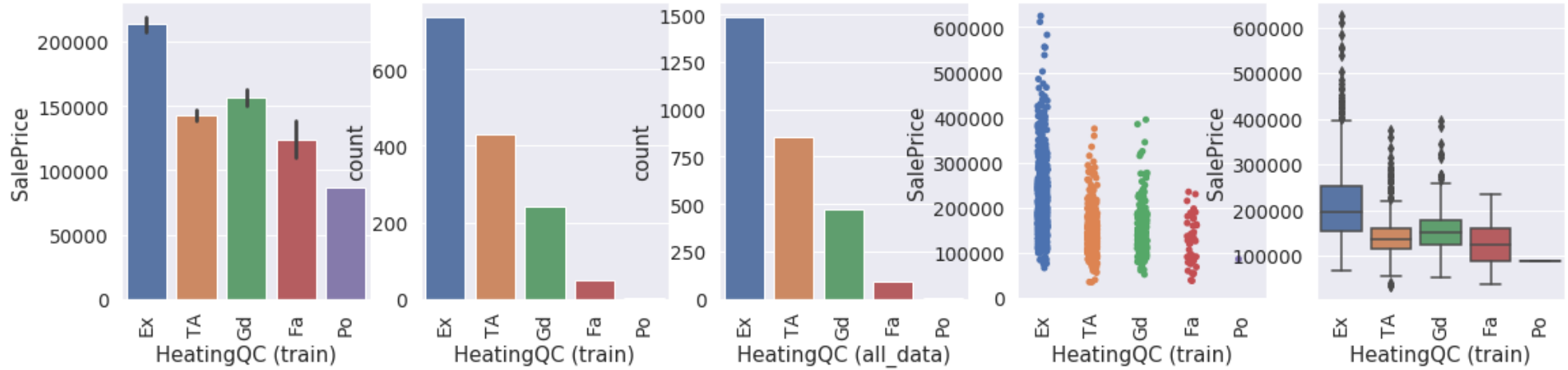
In [102]:
```
all_df["SimplFunctional"] = all_df.Functional.replace(
    {1 : 1, 2 : 1, 3 : 2, 4 : 2, 5 : 3, 6 : 3, 7 : 3, 8 : 4})
```

In [103]:
```
type_based_feature_analysis('KitchenQual')
```



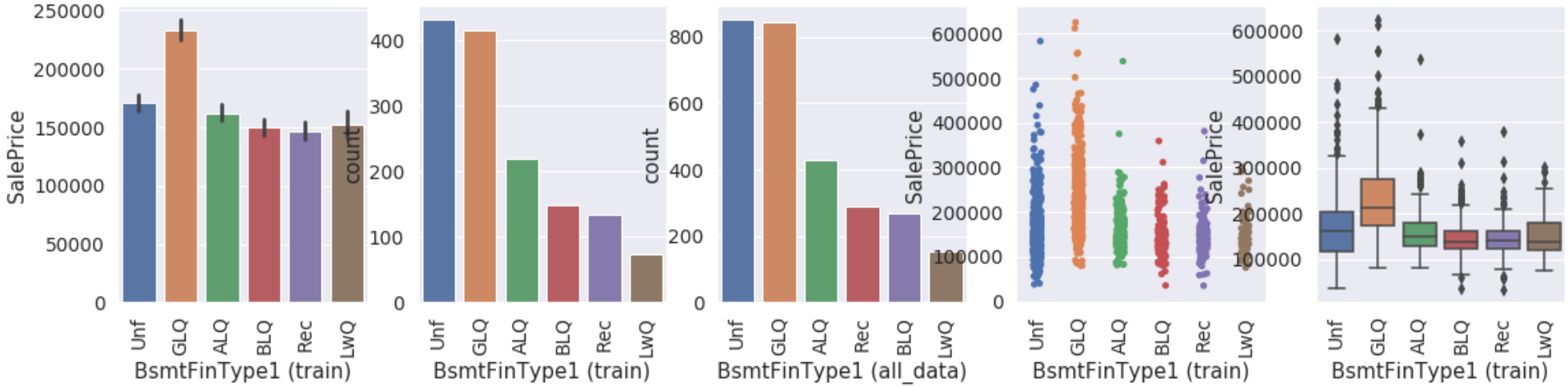
In [104]:
```
all_df["SimplKitchenQual"] = all_df.KitchenQual.replace(
    {1 : 1, 2 : 1, 3 : 1, 4 : 2, 5 : 2})
```

In [105]:
```
type_based_feature_analysis('HeatingQC')
```



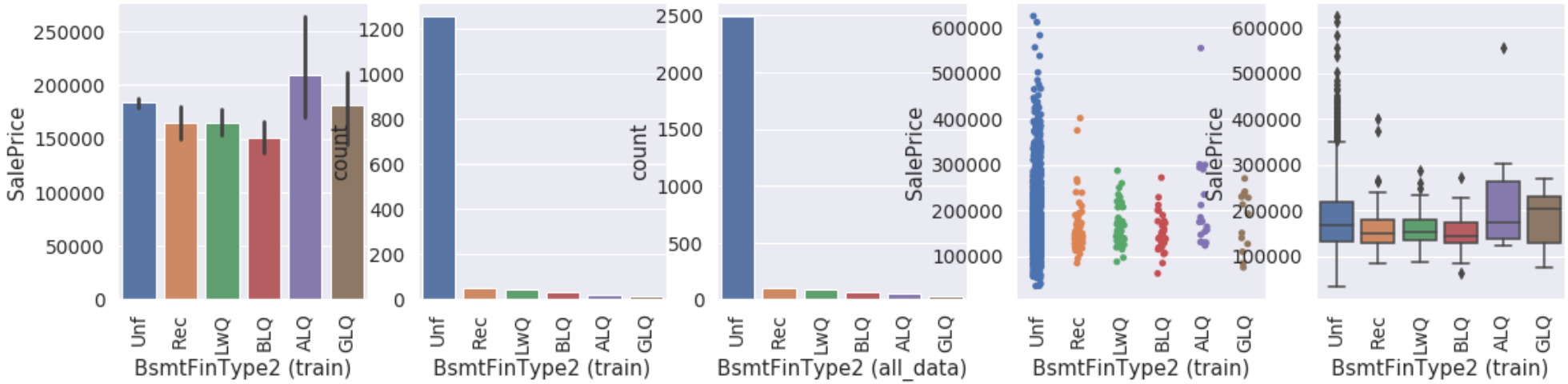
In [106]:
```
all_df["SimplHeatingQC"] = all_df.HeatingQC.replace(
    {1 : 1, 2 : 1, 3 : 1, 4 : 2, 5 : 2})
```

In [107]:
```
type_based_feature_analysis('BsmtFinType1')
```



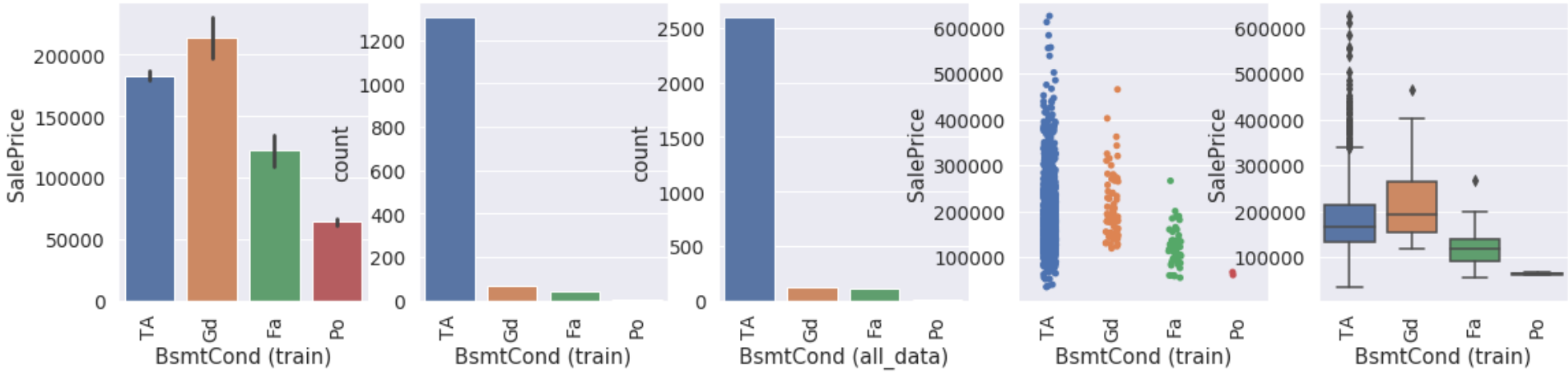
In [108]:
```
all_df["SimplBsmtFinType1"] = all_df.BsmtFinType1.replace(
    {1 : 1, 2 : 1, 3 : 1, 4 : 2, 5 : 2, 6 : 2})
```

In [109]:
```
type_based_feature_analysis('BsmtFinType2')
```



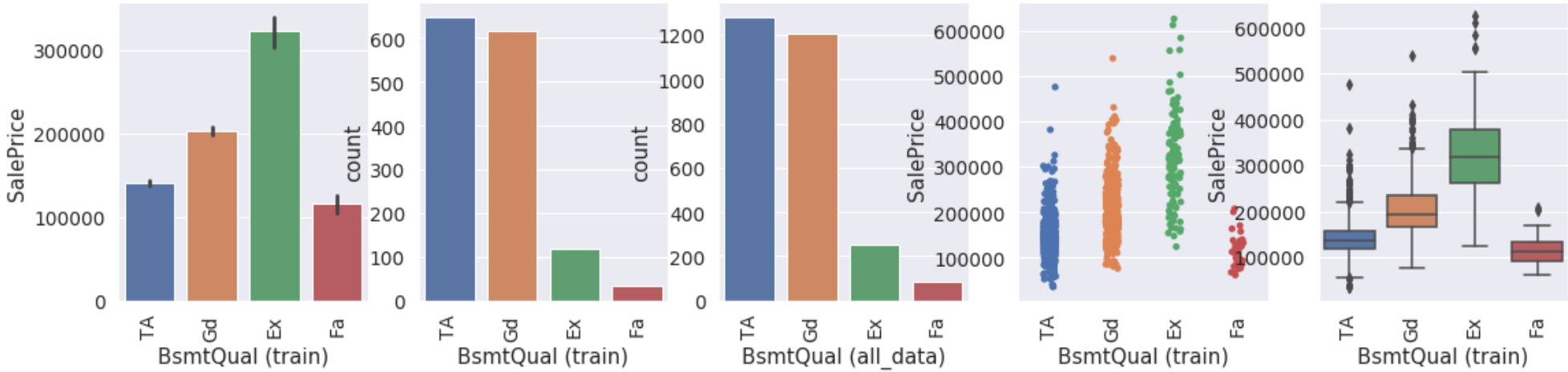
In [110]:
```
all_df["SimplBsmtFinType2"] = all_df.BsmtFinType2.replace(
    {1 : 1, 2 : 1, 3 : 1, 4 : 2, 5 : 2, 6 : 2})
```

In [111]:
```
type_based_feature_analysis('BsmtCond')
```



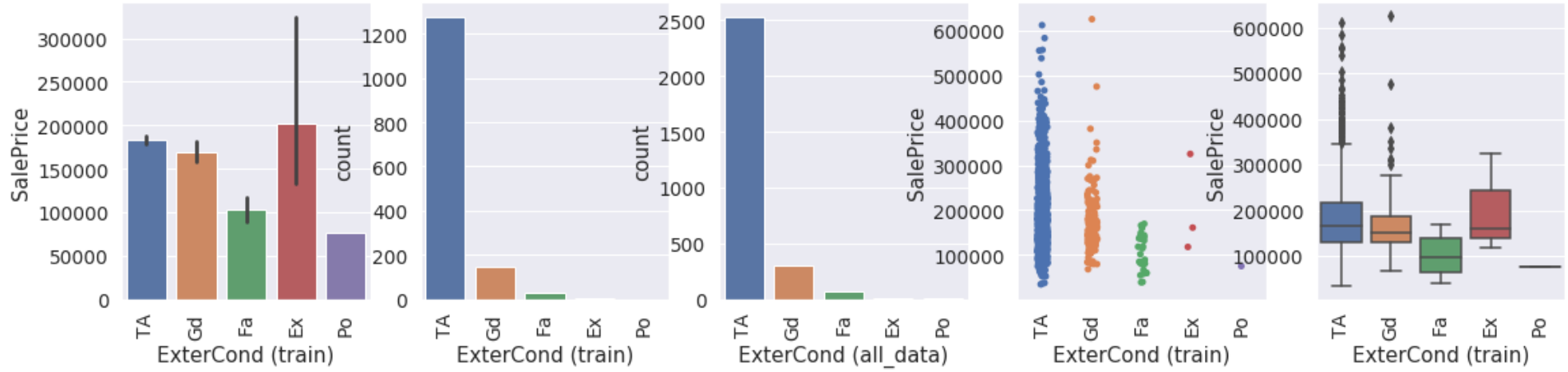
In [112]:
```
all_df["SimplBsmtCond"] = all_df.BsmtCond.replace(
    {1 : 1, 2 : 1, 3 : 1, 4 : 2, 5 : 2})
```

In [113]:
```
type_based_feature_analysis('BsmtQual')
```



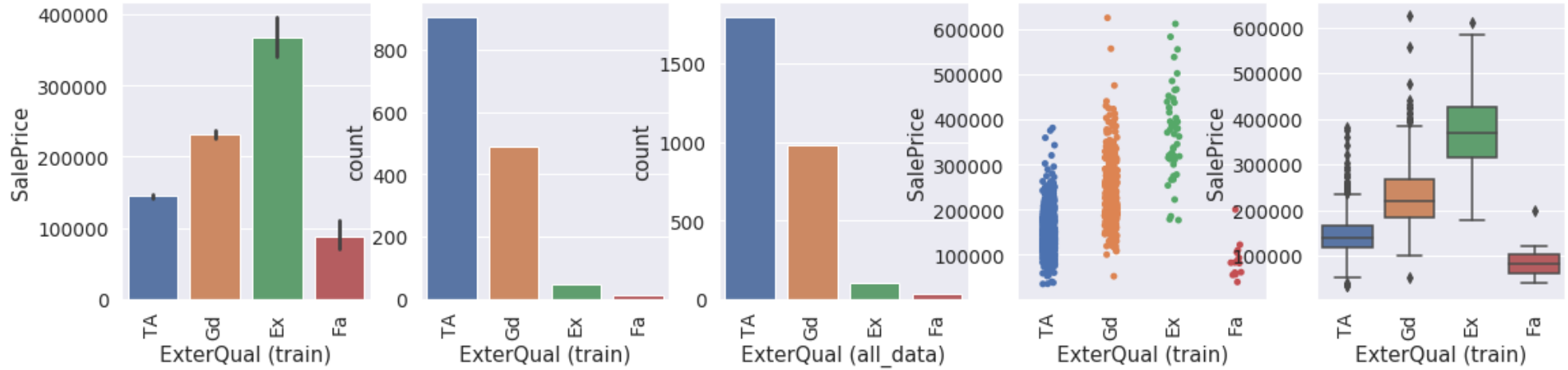
In [114]:
```
all_df["SimplBsmtQual"] = all_df.BsmtQual.replace(
    {1 : 1, 2 : 1, 3 : 1, 4 : 2, 5 : 2})
```

In [115]:
```
type_based_feature_analysis('ExterCond')
```



In [116]:
```
all_df["SimplExterCond"] = all_df.ExterCond.replace(
    {1 : 1, 2 : 1, 3 : 1, 4 : 2, 5 : 2})
```

In [117]:
```
type_based_feature_analysis('ExterQual')
```



In [118]:
```
all_df["SimplExterQual"] = all_df.ExterQual.replace(
    {1 : 1, 2 : 1, 3 : 1, 4 : 2, 5 : 2})
```

In [119]:
```
year_based_feature_analysis('Neighborhood')
```


SalePrice
0
50000
100000
150000
200000
250000
300000
350000
SalePrice
100000
200000
300000
400000
500000
600000
SalePrice
0
100000
200000
300000
400000
500000
600000

Bin by neighborhood (a little arbitrarily). Values were computed by:

train_df["SalePrice"].groupby(train_df["Neighborhood"]).median().sort_values()

In [120]:
```
neighborhood_map = {
    "MeadowV" : 0,  #  88000
    "IDOTRR" : 1,   # 103000
    "BrDale" : 1,   # 106000
    "OldTown" : 1,  # 119000
    "Edwards" : 1,  # 119500
    "BrkSide" : 1,  # 124300
    "Sawyer" : 1,   # 135000
    "Blueste" : 1,  # 137500
    "SWISU" : 2,    # 139500
    "NAmes" : 2,    # 140000
    "NPkVill" : 2,  # 146000
    "Mitchel" : 2,  # 153500
    "SawyerW" : 2,  # 179900
    "Gilbert" : 2,  # 181000
    "NWAmes" : 2,   # 182900
    "Blmngtn" : 2,  # 191000
    "CollgCr" : 2,  # 197200
    "ClearCr" : 3,  # 200250
    "Crawfor" : 3,  # 200624
    "Veenker" : 3,  # 218000
    "Somerst" : 3,  # 225500
    "Timber" : 3,   # 228475
    "StoneBr" : 4,  # 278000
    "NoRidge" : 4,  # 290000
    "NridgHt" : 4,  # 315000
}

all_df["NeighborhoodBin"] =all_data["Neighborhood"].map(neighborhood_map)
```

Summery

- Filled with 0 for some features like "MasVnrArea", "BsmtFinSF1" "BsmtFinSF2" "BsmtUnfSF" "TotalBsmtSF" "GarageArea" "BsmtFullBath" "BsmtHalfBath" "GarageCars" "PoolArea" "GarageYrBlt" .According to the documentation of the dataset if these features have any field empty then that means the feature is not available. So I have done this operation according to documentation of the dataset.
- CentralAir feature was given has two field only 'Y' or 'N' so I have converted that to 0 or 1
- For some ordinal features I ave performed lable encoding to map data to a numerical features. Those features are ExterQual , ExterCond, BsmtQual, BsmtCond, HeatingQC, KitchenQual etc
- I have converted some features from categorical to numerical and those features are MSSubClass, MSZoning , LotConfig, RL , LotConfig,Neighborhood, Condition1 ,BldgType, HouseStyle , HouseStyle, Exterior1st, Other, Exterior2nd, MasVnrType, Foundation, SaleType and SaleCondition
- Converted fields of some Features to 0 or 1 based on the understanding of the dataset and a little bit research. What I have done is that I have made simplified versions of existing features. For example, the Land Slope feature lets us know what type of slope the property has. Even though is has multiple labels, it all comes down to if the slope is gentle or not. Hence I have created a new feature called IsLandSlopeGentle, which is effectively tells us if the slope is gentle (==1) or is it not gentle (==0).Those features with the changing reasons are given below
  - IsRegularLotShape : Field IR2 and IR3 don't appear that often, so just make a distinction between regular and irregular.
  - IsLandLevel : Most land slopes are gentle; treat the others as "not gentle".
  - IsElectricalSBrkr : Most properties use standard circuit breakers.
  - IsGarageDetached : About 2/3rd have an attached garage.
  - IsPavedDrive : Most have a paved drive. Treat dirt/gravel and partial pavement as "not paved".
  - HasShed : The only interesting "misc. feature" is the presence of a shed.
  - Remodeled : If YearRemodAdd != YearBuilt, then a remodeling took place at some point.
  - RecentRemodel : Did a remodeling happen in the year the house was sold?
  - VeryNewHouse : Was this house sold in the year it was built?
  - sofe other features dont need to describe they are self explanatory Has2ndFloor , HasMasVnr , HasWoodDeck .HasOpenPorch ,HasEnclosedPorch ,Has3SsnPorch , HasScreenPorch
- Simplifications of existing features into bad/average/good. Features : SimplOverallQual, SimplOverallCond, SimplPoolQC ,SimplGarageCond, SimplGarageQual, SimplFireplaceQu ,SimplFunctional ,SimplKitchenQual, SimplHeatingQC ,SimplBsmtFinType1, SimplBsmtFinType2 ,SimplBsmtCond , SimplBsmtQual ,SimplExterCond ,SimplExterQual.
- mapped neighborhood based on their quality.The mapping is as followed:

```
"MeadowV" : 0,  #  88000
"IDOTRR" : 1,   # 103000
"BrDale" : 1,   # 106000
"OldTown" : 1,  # 119000
"Edwards" : 1,  # 119500
"BrkSide" : 1,  # 124300
"Sawyer" : 1,   # 135000
"Blueste" : 1,  # 137500
"SWISU" : 2,    # 139500
"NAmes" : 2,    # 140000
"NPkVill" : 2,  # 146000
"Mitchel" : 2,  # 153500
"SawyerW" : 2,  # 179900
"Gilbert" : 2,  # 181000
"NWAmes" : 2,   # 182900
"Blmngtn" : 2,  # 191000
"CollgCr" : 2,  # 197200
"ClearCr" : 3,  # 200250
"Crawfor" : 3,  # 200624
"Veenker" : 3,  # 218000
"Somerst" : 3,  # 225500
"Timber" : 3,   # 228475
"StoneBr" : 4,  # 278000
"NoRidge" : 4,  # 290000
"NridgHt" : 4,  # 315000
- the number after hash is actually median priece of that location.
```

Keeping NeighborhoodBin into a temporary DataFrame because we want to use the unscaled version later on (to one-hot encode it).

In [121]:
```
# Keeping NeighborhoodBin into a temporary DataFrame because we want to use the
# unscaled version later on (to one-hot encode it).
neighborhood_bin = pd.DataFrame(index = all_df.index)
neighborhood_bin["NeighborhoodBin"] = all_df["NeighborhoodBin"]
```

Skewness, Normalization & Standardization

According to Hair et al. (2013), four assumptions should be tested:

- **Normality** - When we talk about normality what we mean is that the data should look like a normal distribution. This is important because several statistic tests rely on this (e.g. t-statistics). In this exercise we'll just check univariate normality for 'SalePrice' (which is a limited approach). Remember that univariate normality doesn't ensure multivariate normality (which is what we would like to have), but it helps. Another detail to take into account is that in big samples (>200 observations) normality is not such an issue. However, if we solve normality, we avoid a lot of other problems (e.g. heteroscedacity) so that's the main reason why we are doing this analysis.
- **Homoscedasticity** - Homoscedasticity refers to the 'assumption that dependent variable(s) exhibit equal levels of variance across the range of predictor variable(s)' (Hair et al., 2013). Homoscedasticity is desirable because we want the error term to be the same across all values of the independent variables.
- **Linearity** - The most common way to assess linearity is to examine scatter plots and search for linear patterns. If patterns are not linear, it would be worthwhile to explore data transformations. However, we'll not get into this because most of the scatter plots we've seen appear to have linear relationships.
- **standardization** is the process of putting different variables on the same scale. This process allows you to compare scores between different types of variables. Typically, to standardize variables, you calculate the mean and standard deviation for a variable. Then, for each observed value of the variable, you subtract the mean and divide by the standard deviation.

Skewness, in basic terms, implies off-centre, so does in statistics, it means lack of symmetry. With the help of skewness, one can identify the shape of the distribution of data.

In the simplest cases, normalization of ratings means adjusting values measured on different scales to a notionally common scale, often prior to averaging.Some types of normalization involve only a rescaling, to arrive at values relative to some size variable.

We will remove skewness through normalization and then scale all the numeric features using standardization technique (Except SalePrice).

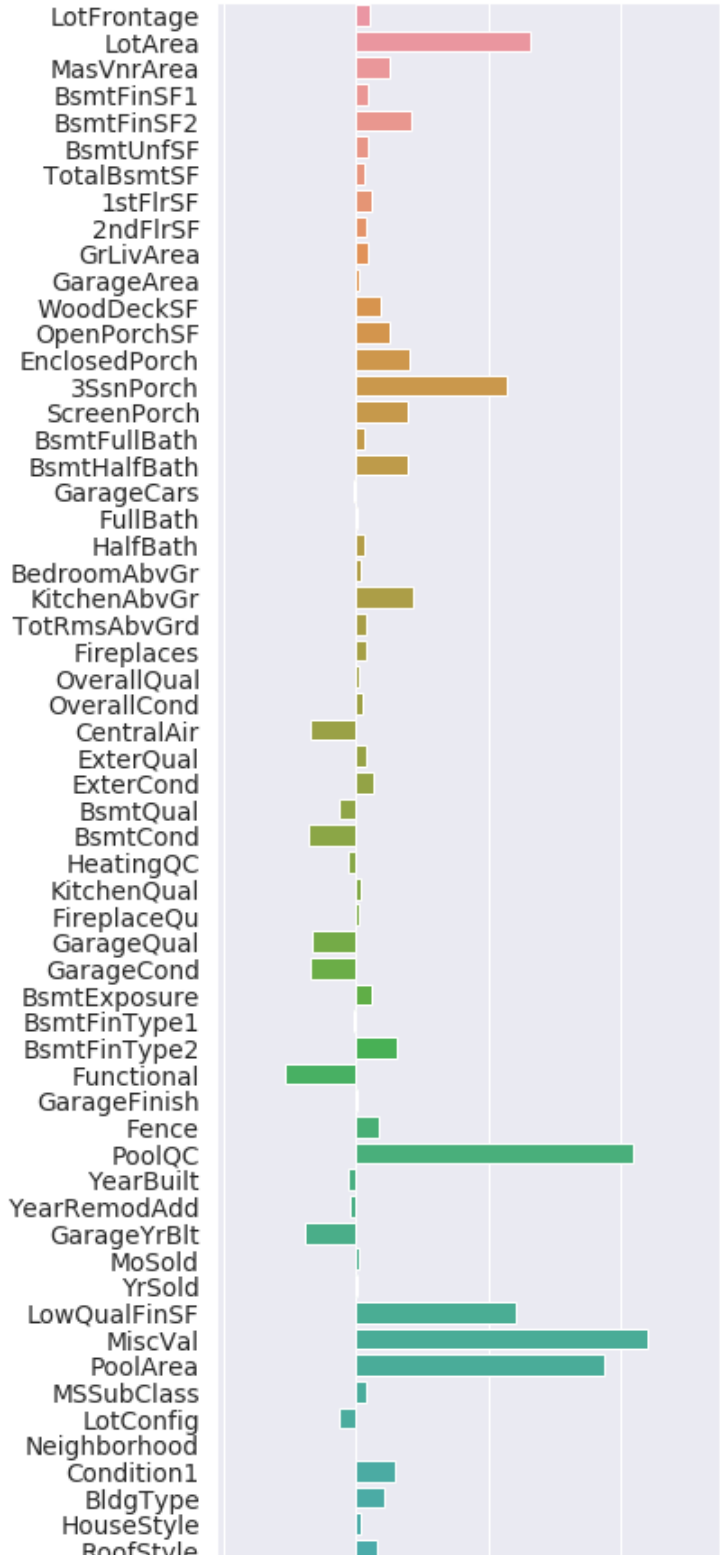
skewness

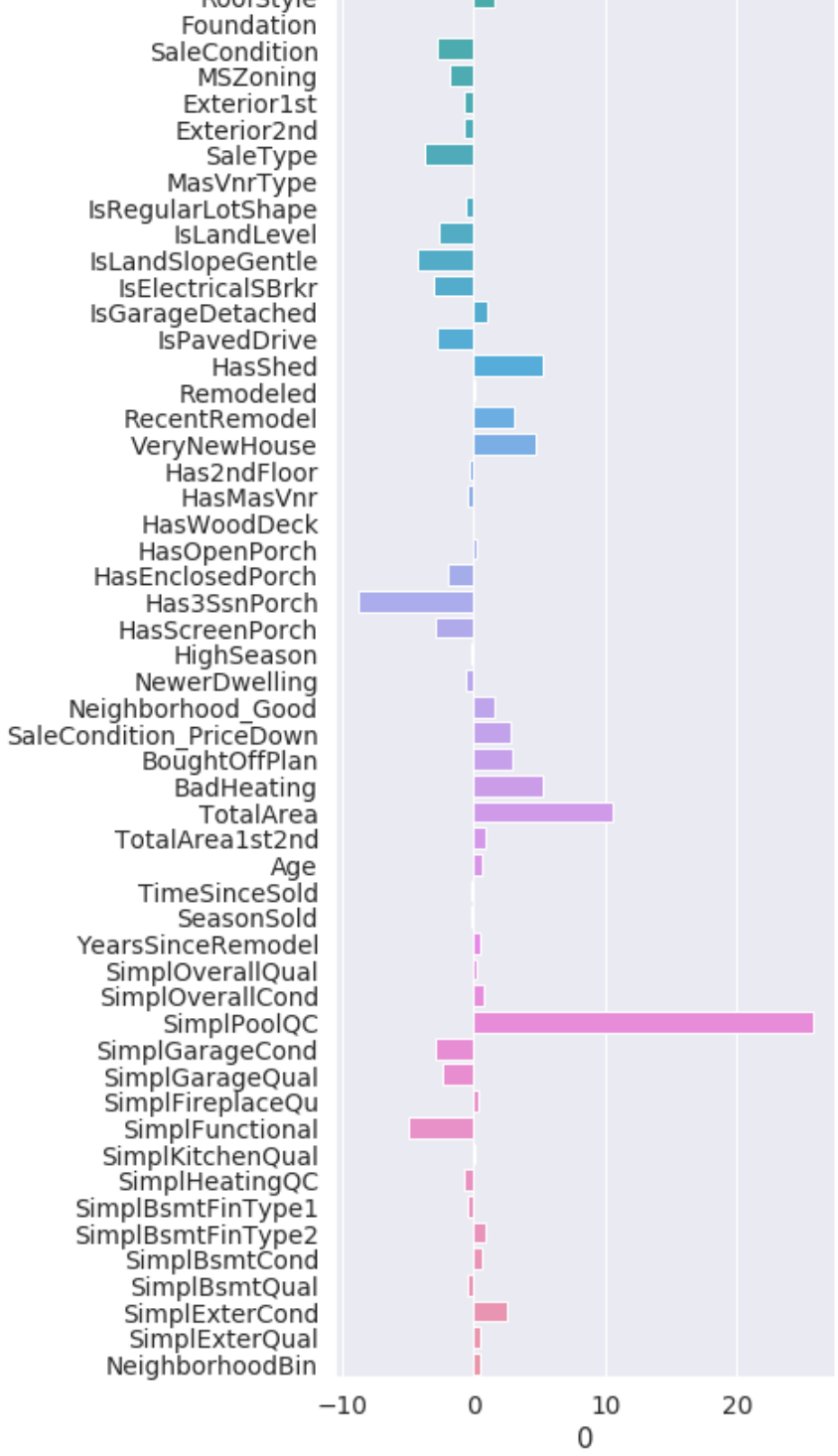
In the following part we are looking at skewness of dataset and we can see that many features are highly skewed. We will be solving it with log transformation.

The log transformation is, arguably, the most popular among the different types of transformations used to transform skewed data to approximately conform to normality. If the original data follows a log-normal distribution or approximately so, then the log-transformed data follows a normal or near normal distribution.

In [122]:
```
# keeping train and test data in a flag for comparison purpose
old_train_skewness_flag = all_df[:ntrain].copy()
old_test_skewness_flag = all_df[ntrain:].copy()
old_target_skewness_flag= train["SalePrice"].copy()
# old_target_skewness_flag
```

In [123]:
```
from scipy.stats import skew
numeric_features = all_df.dtypes[all_df.dtypes != "object"].index
skewness = all_df[numeric_features].skew(axis=0 , skipna =True)
skewness = pd.DataFrame(skewness)
plt.figure(figsize=[5,30])
# skw = sns.load_dataset(skewness)
ax = sns.barplot( y= skewness.index , x=skewness[0] , data = skewness)
plt.show()
```


LotFrontage
LotArea
MasVnrArea
BsmtFinSF1
BsmtFinSF2
BsmtUnfSF
TotalBsmtSF
1stFlrSF
2ndFlrSF
GrLivArea
GarageArea
WoodDeckSF
OpenPorchSF
EnclosedPorch
3SsnPorch
ScreenPorch
BsmtFullBath
BsmtHalfBath
GarageCars
FullBath
HalfBath
BedroomAbvGr
KitchenAbvGr
TotRmsAbvGrd
Fireplaces
OverallQual
OverallCond
CentralAir
ExterQual
ExterCond
BsmtQual
BsmtCond
HeatingQC
KitchenQual
FireplaceQu
GarageQual
GarageCond
BsmtExposure
BsmtFinType1
BsmtFinType2
Functional
GarageFinish
Fence
PoolQC
YearBuilt
YearRemodAdd
GarageYrBlt
MoSold
YrSold
LowQualFinSF
MiscVal
PoolArea
MSSubClass
LotConfig
Neighborhood
Condition1
BldgType
HouseStyle
RoofStyle


RoofStyle
Foundation
SaleCondition
MSZoning
Exterior1st
Exterior2nd
SaleType
MasVnrType
IsRegularLotShape
IsLandLevel
IsLandSlopeGentle
IsElectricalSBrkr
IsGarageDetached
IsPavedDrive
HasShed
Remodeled
RecentRemodel
VeryNewHouse
Has2ndFloor
HasMasVnr
HasWoodDeck
HasOpenPorch
HasEnclosedPorch
Has3SsnPorch
HasScreenPorch
HighSeason
NewerDwelling
Neighborhood_Good
SaleCondition_PriceDown
BoughtOffPlan
BadHeating
TotalArea
TotalArea1st2nd
Age
TimeSinceSold
SeasonSold
YearsSinceRemodel
SimplOverallQual
SimplOverallCond
SimplPoolQC
SimplGarageCond
SimplGarageQual
SimplFireplaceQu
SimplFunctional
SimplKitchenQual
SimplHeatingQC
SimplBsmtFinType1
SimplBsmtFinType2
SimplBsmtCond
SimplBsmtQual
SimplExterCond
SimplExterQual
NeighborhoodBin
−10
0
10
20
0

Observation

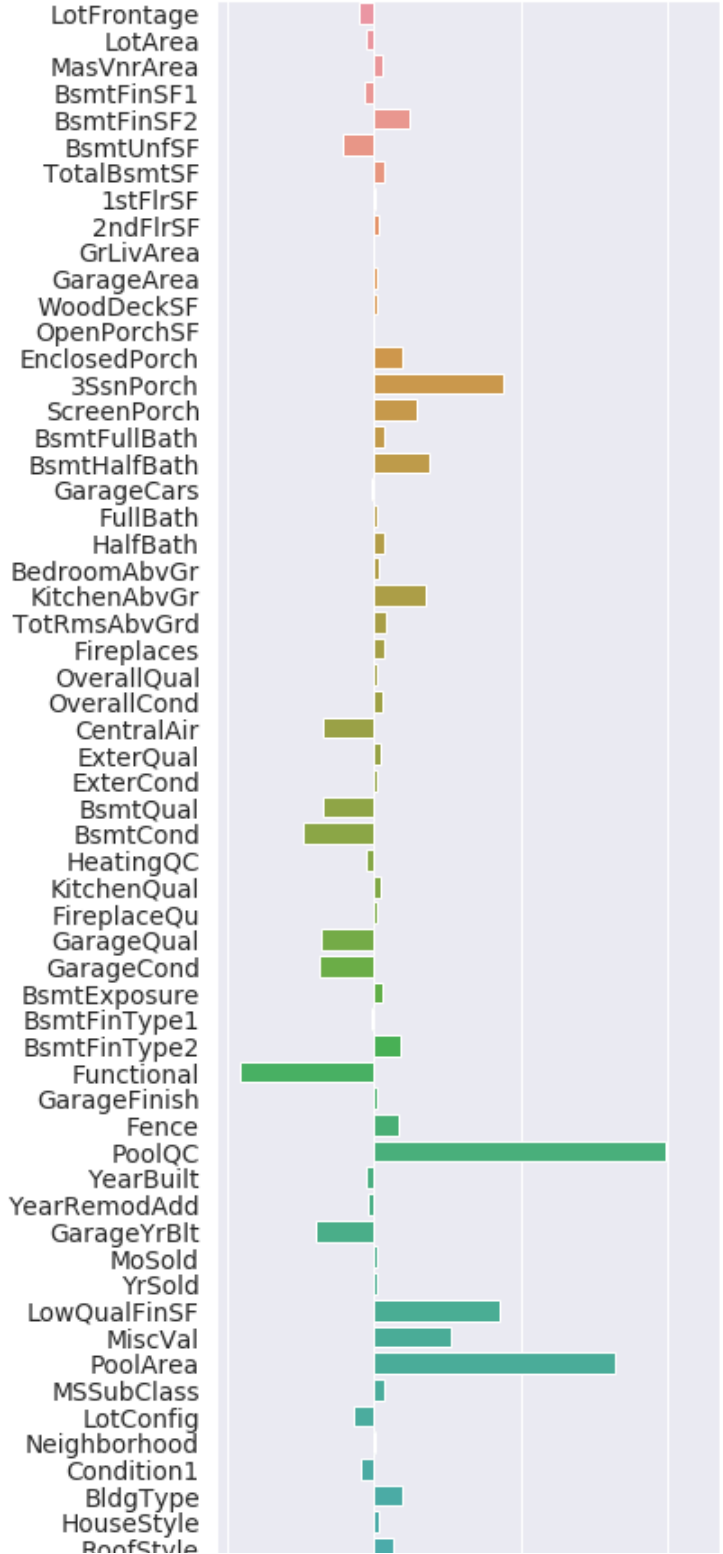
- A significant number of observations with value zero (houses without basement).
- A big problem because the value zero doesn't allow us to do log transformations.

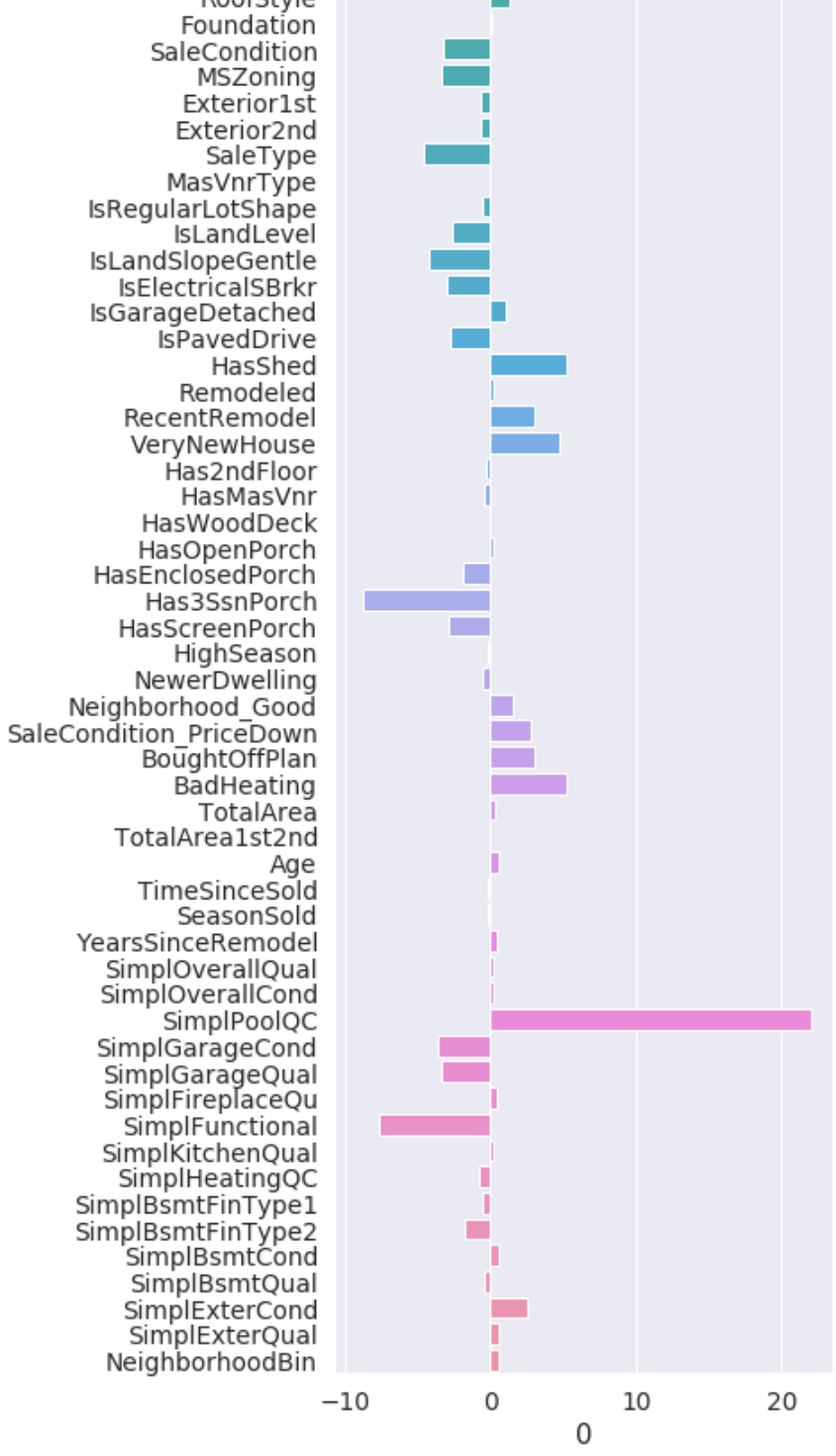
To apply a log transformation here, we need to add 1 and then perform log transform operation. **Note** : For real-valued input, log1p is accurate also for x so small that 1 + x == 1 in floating-point accuracy.

In [124]:
```
numeric_features = all_df.dtypes[all_df.dtypes != "object"].index

# Transform the skewed numeric features by taking log(feature + 1).
# This will make the features more normal.
from scipy.stats import skew

skewed = all_df[numeric_features].apply(lambda x: skew(x.dropna().astype(float)))
skewed = skewed[(skewed < -0.75) | (skewed > 0.75)]
skewed = skewed.index

all_df[skewed] = np.log1p(all_df[skewed])


# Additional processing: scale the data.
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaled = scaler.fit_transform(all_df[numeric_features])

for i, col in enumerate(numeric_features):
    all_df[col] = scaled[:, i]
```

```
/home/navid/anaconda3/envs/tf/lib/python3.6/site-packages/sklearn/preprocessing/data.py:645: DataConversionWarning: Data with inpu
t dtype int64, float64 were all converted to float64 by StandardScaler.
  return self.partial_fit(X, y)
/home/navid/anaconda3/envs/tf/lib/python3.6/site-packages/sklearn/base.py:464: DataConversionWarning: Data with input dtype int64,
float64 were all converted to float64 by StandardScaler.
  return self.fit(X, **fit_params).transform(X)
```

In [125]:
```
from scipy.stats import skew
numeric_features = all_df.dtypes[all_df.dtypes != "object"].index
skewness = all_df[numeric_features].skew(axis=0 , skipna =True)
skewness = pd.DataFrame(skewness)
plt.figure(figsize=[5,30])
# skw = sns.load_dataset(skewness)
ax = sns.barplot( y= skewness.index , x=skewness[0] , data = skewness)
plt.show()
```


LotFrontage
LotArea
MasVnrArea
BsmtFinSF1
BsmtFinSF2
BsmtUnfSF
TotalBsmtSF
1stFlrSF
2ndFlrSF
GrLivArea
GarageArea
WoodDeckSF
OpenPorchSF
EnclosedPorch
3SsnPorch
ScreenPorch
BsmtFullBath
BsmtHalfBath
GarageCars
FullBath
HalfBath
BedroomAbvGr
KitchenAbvGr
TotRmsAbvGrd
Fireplaces
OverallQual
OverallCond
CentralAir
ExterQual
ExterCond
BsmtQual
BsmtCond
HeatingQC
KitchenQual
FireplaceQu
GarageQual
GarageCond
BsmtExposure
BsmtFinType1
BsmtFinType2
Functional
GarageFinish
Fence
PoolQC
YearBuilt
YearRemodAdd
GarageYrBlt
MoSold
YrSold
LowQualFinSF
MiscVal
PoolArea
MSSubClass
LotConfig
Neighborhood
Condition1
BldgType
HouseStyle


RoofStyle
Foundation
SaleCondition
MSZoning
Exterior1st
Exterior2nd
SaleType
MasVnrType
IsRegularLotShape
IsLandLevel
IsLandSlopeGentle
IsElectricalSBrkr
IsGarageDetached
IsPavedDrive
HasShed
Remodeled
RecentRemodel
VeryNewHouse
Has2ndFloor
HasMasVnr
HasWoodDeck
HasOpenPorch
HasEnclosedPorch
Has3SsnPorch
HasScreenPorch
HighSeason
NewerDwelling
Neighborhood_Good
SaleCondition_PriceDown
BoughtOffPlan
BadHeating
TotalArea
TotalArea1st2nd
Age
TimeSinceSold
SeasonSold
YearsSinceRemodel
SimplOverallQual
SimplOverallCond
SimplPoolQC
SimplGarageCond
SimplGarageQual
SimplFireplaceQu
SimplFunctional
SimplKitchenQual
SimplHeatingQC
SimplBsmtFinType1
SimplBsmtFinType2
SimplBsmtCond
SimplBsmtQual
SimplExterCond
SimplExterQual
NeighborhoodBin
−10
0
10
20
0

We can see that skewness of the following features decreased a lot:

- LotArea
- WoodDeskSf
- OpenPorch
- Extencond
- MiscVal
- TotalArea

But other numeric features also improved its skewness a little bit.

splitting into train and test dataset

In [126]:
```
train_processed = all_df[:ntrain]
test_processed = all_df[ntrain:]

print("shape of train :" , train_processed.shape)
print("shape of test :" , test_processed.shape)
```

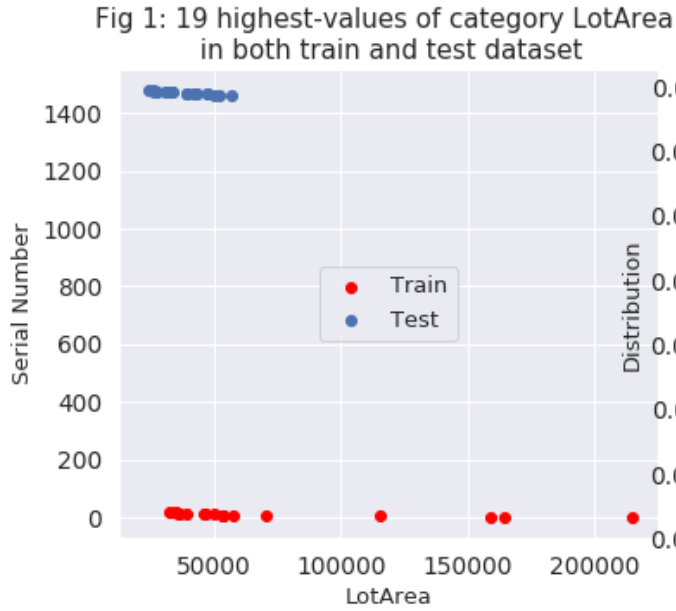
```
shape of train : (1456, 111)
shape of test : (1459, 111)
```

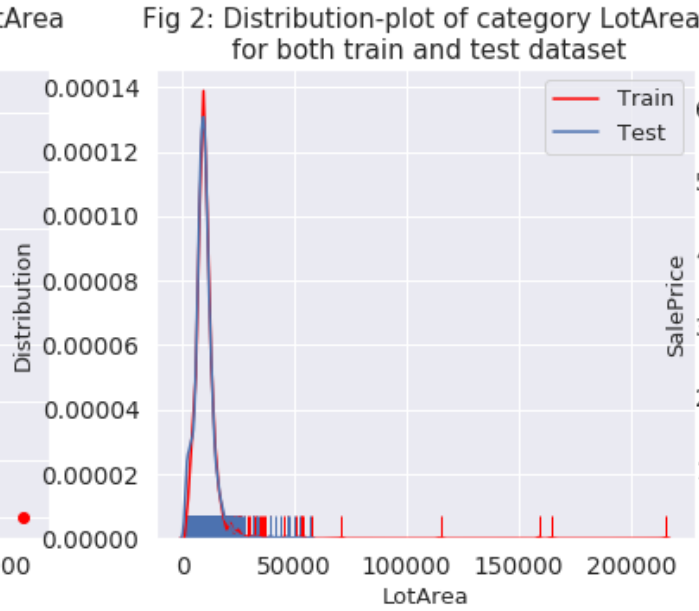
Observation

In this section we will observe how Distribution plot changes due to normalization and standardization of the numeric features. In the first line of plot we would be able to see the distribution before skewness section starts and every second line we will see how it changes due to skewness removal and standardization. Fig-2 is the distribution plot so we should observe it carefully. We can observe that how much skewness of the data is lost due to normalization. Fig-3 will show the relation between SalePrice and and the feature. If the relation between them is linear or close to linear then that will help us in training.

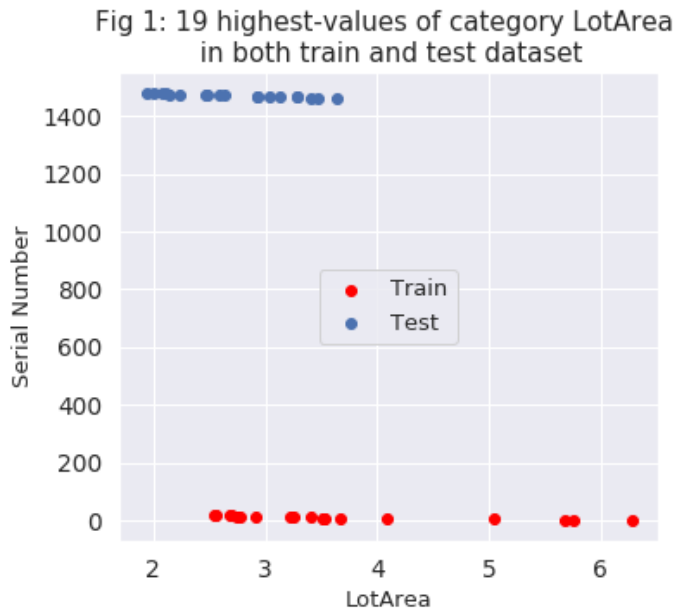
In [127]:
```
from IPython.display import Markdown, display
def printmd(string):
    display(Markdown("***"+string+"***"))

printmd('Before skewness removal:')
outlier_check_plot('LotArea',old_train_skewness_flag, old_test_skewness_flag, old_target_skewness_flag)
printmd('After skewness removal:')
outlier_check_plot('LotArea' , train_processed, test_processed, old_target_skewness_flag)

printmd('Before skewness removal:')
outlier_check_plot('WoodDeckSF',old_train_skewness_flag, old_test_skewness_flag, old_target_skewness_flag)
printmd('After skewness removal:')
outlier_check_plot('WoodDeckSF', train_processed, test_processed, old_target_skewness_flag)

printmd('Before skewness removal:')
outlier_check_plot('OpenPorchSF',old_train_skewness_flag, old_test_skewness_flag, old_target_skewness_flag)
printmd('After skewness removal:')
outlier_check_plot('OpenPorchSF', train_processed, test_processed, old_target_skewness_flag)

printmd('Before skewness removal:')
outlier_check_plot('ExterCond',old_train_skewness_flag, old_test_skewness_flag, old_target_skewness_flag)
printmd('After skewness removal:')
outlier_check_plot('ExterCond', train_processed, test_processed, old_target_skewness_flag)

printmd('Before skewness removal:')
outlier_check_plot('MiscVal',old_train_skewness_flag, old_test_skewness_flag, old_target_skewness_flag)
printmd('After skewness removal:')
outlier_check_plot('MiscVal', train_processed, test_processed, old_target_skewness_flag)

printmd('Before skewness removal:')
outlier_check_plot('TotalArea',old_train_skewness_flag, old_test_skewness_flag, old_target_skewness_flag)
printmd('After skewness removal:')
outlier_check_plot('TotalArea', train_processed, test_processed, old_target_skewness_flag)
```

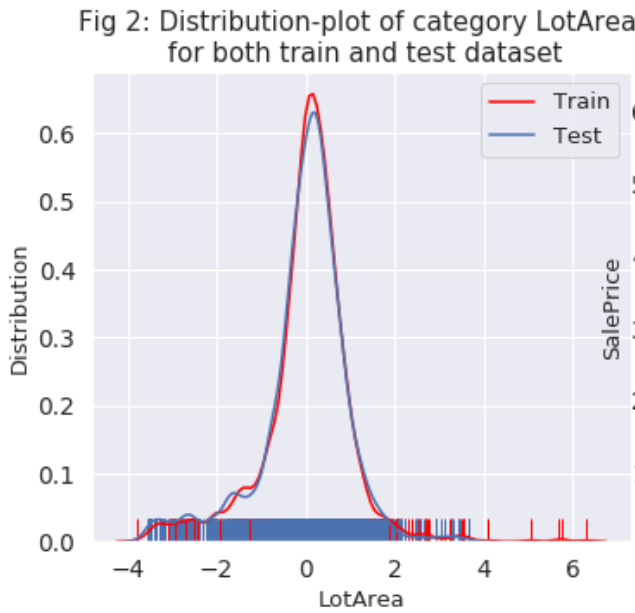
Before skewness removal:



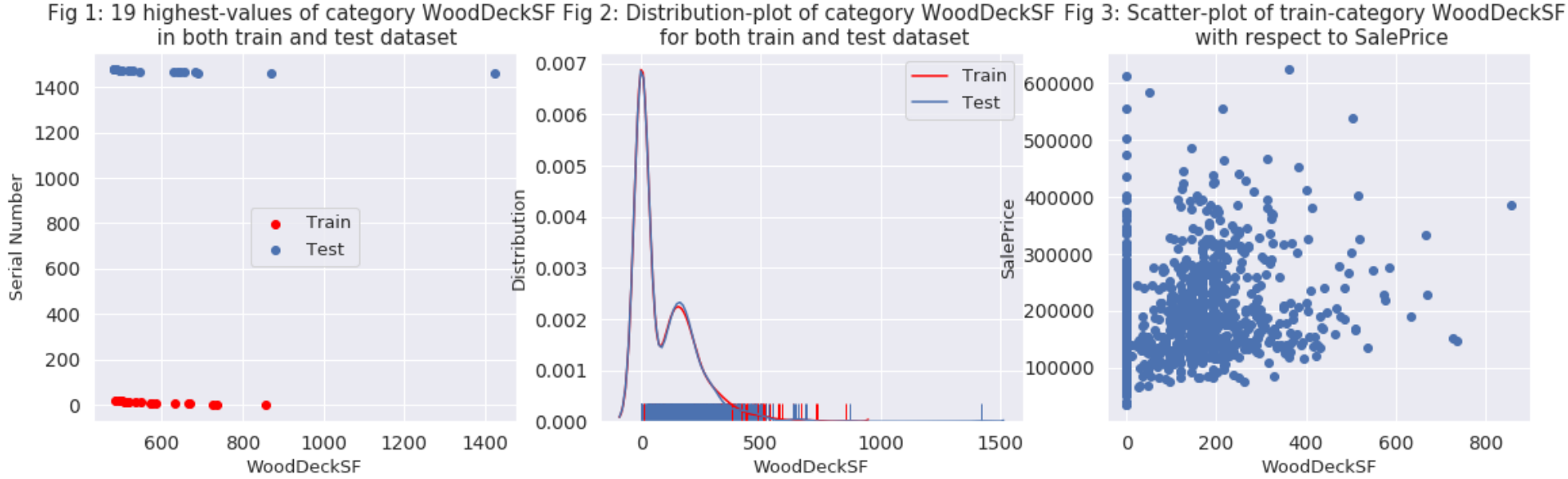


After skewness removal:

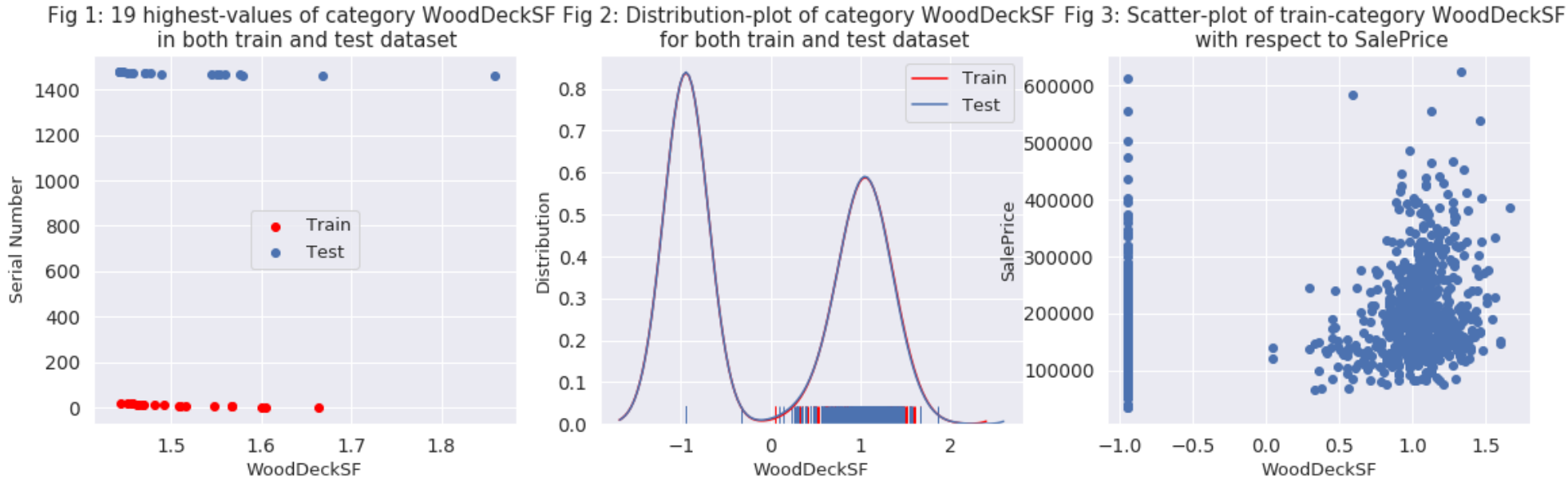




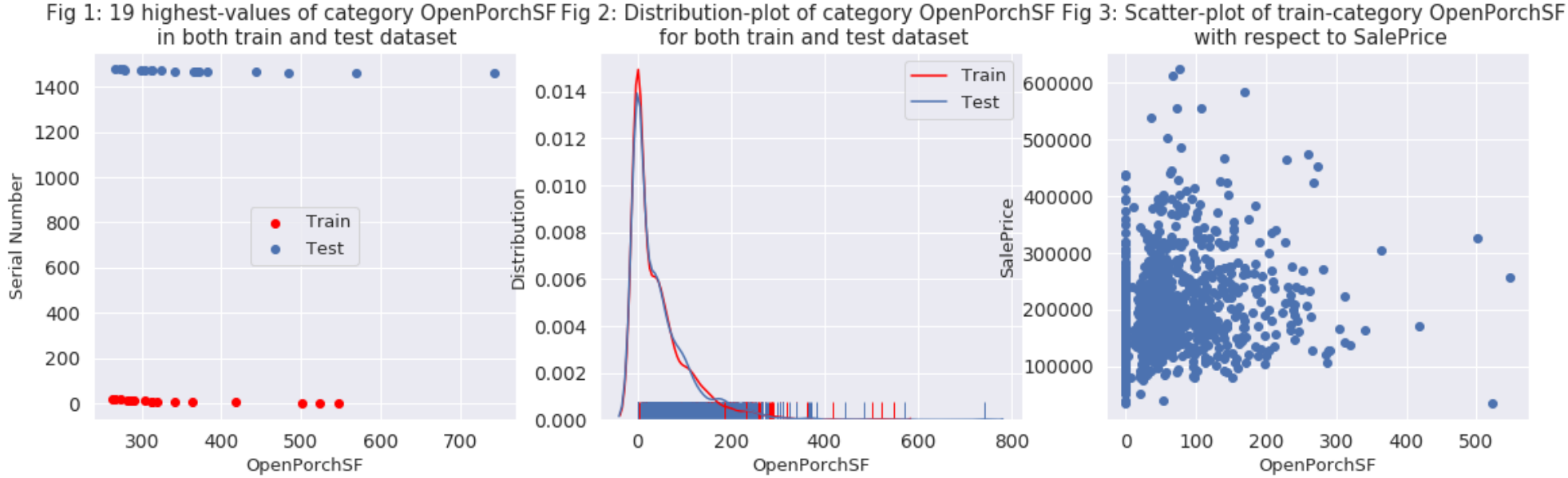
Before skewness removal:



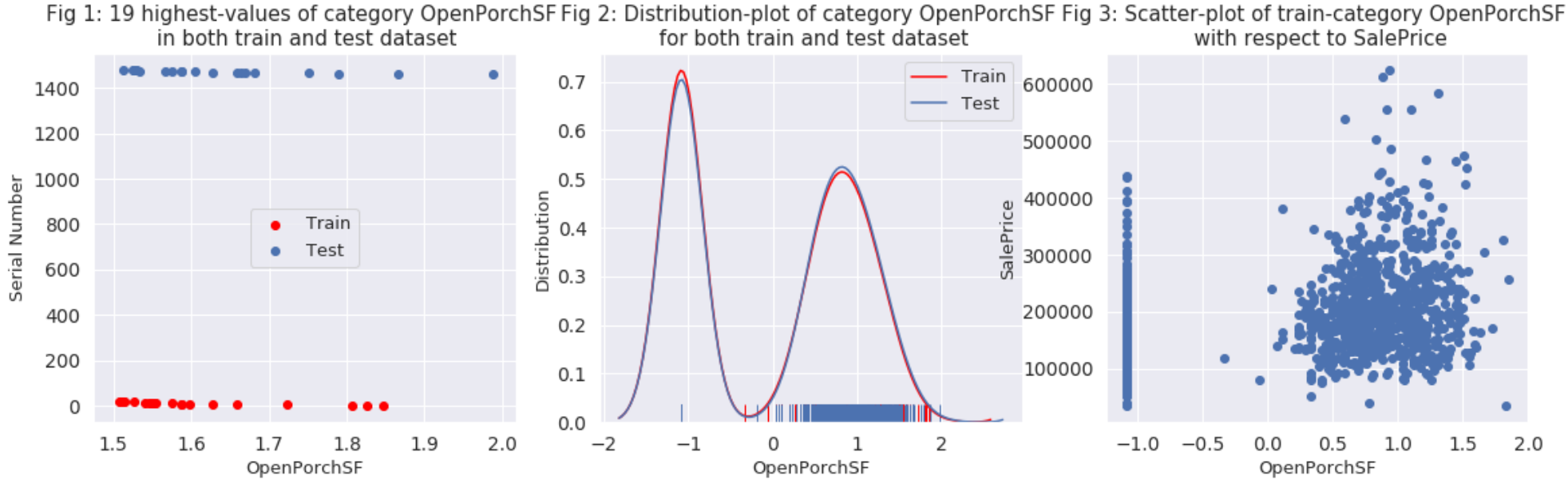
After skewness removal:



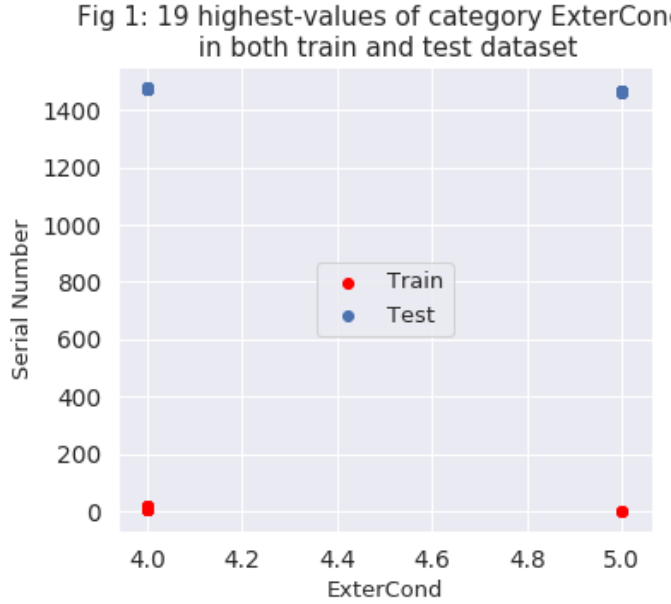
Before skewness removal:

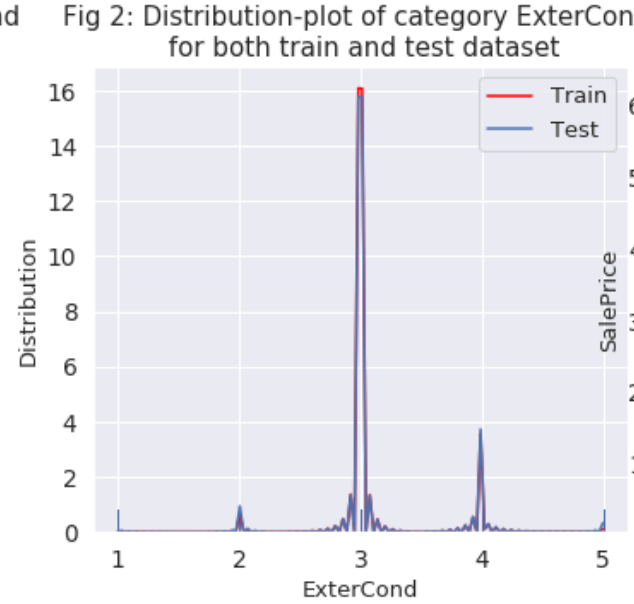


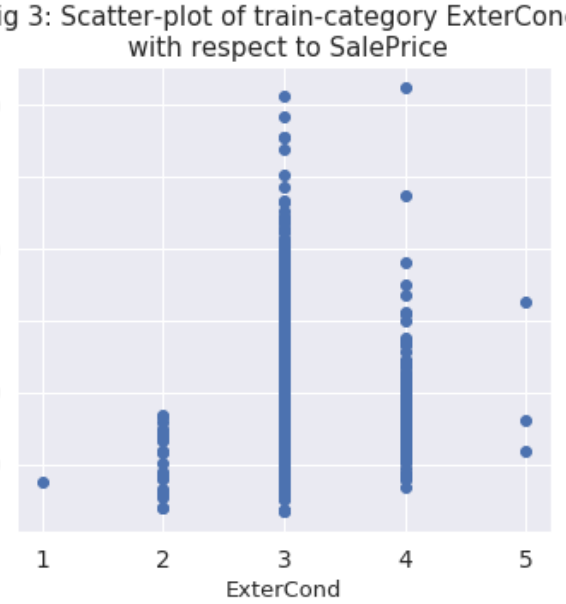
After skewness removal:



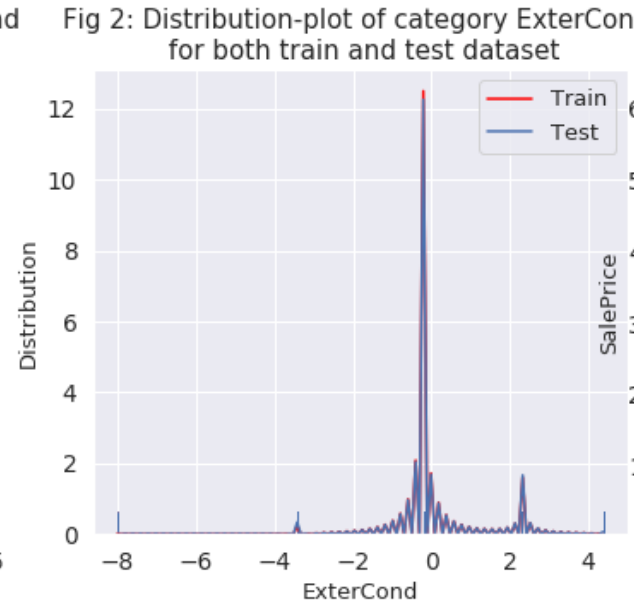
Before skewness removal:

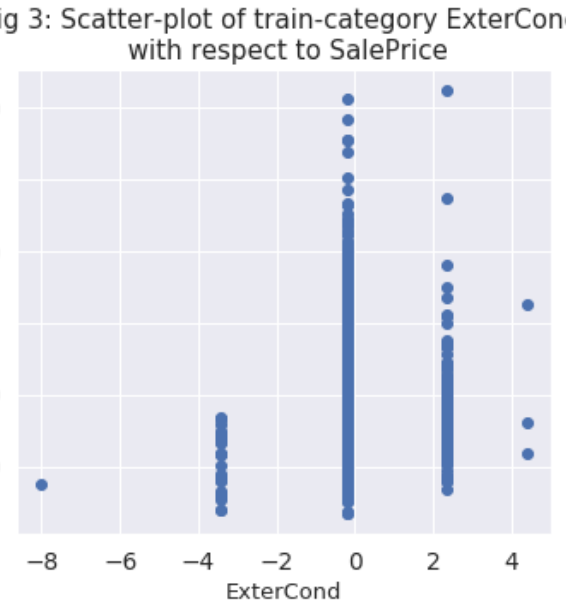




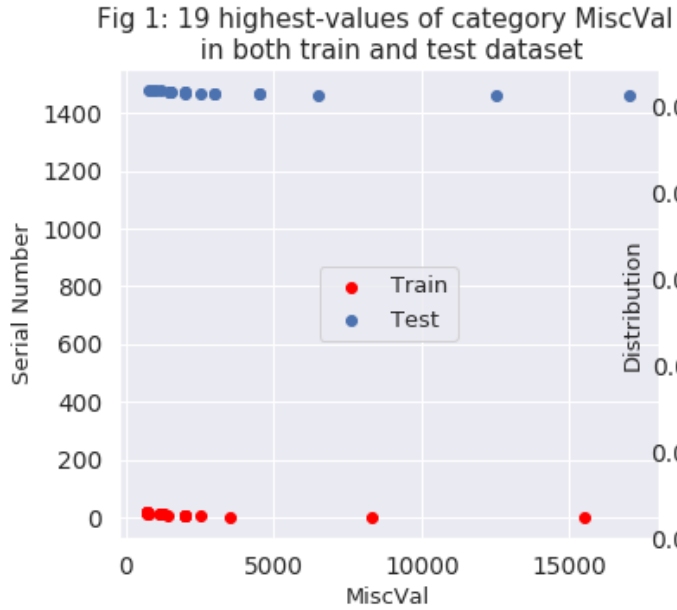


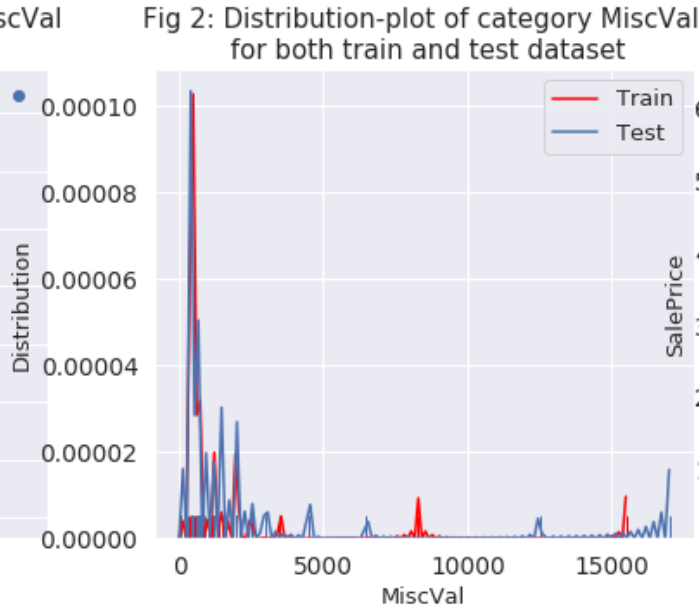
After skewness removal:



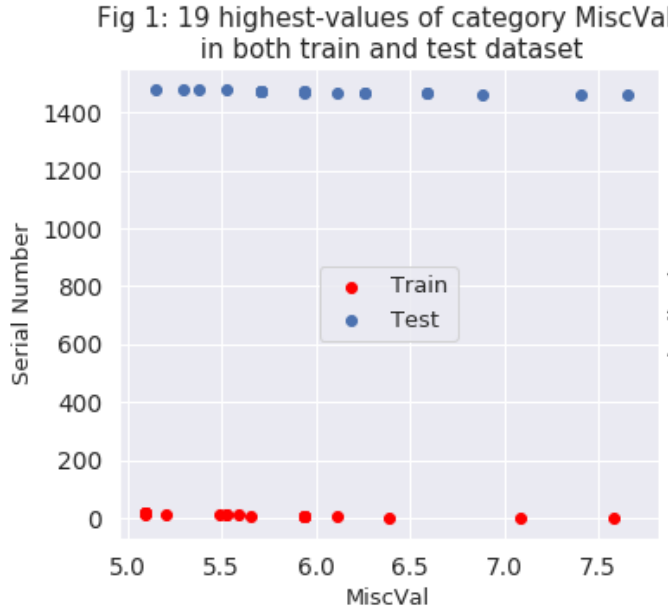


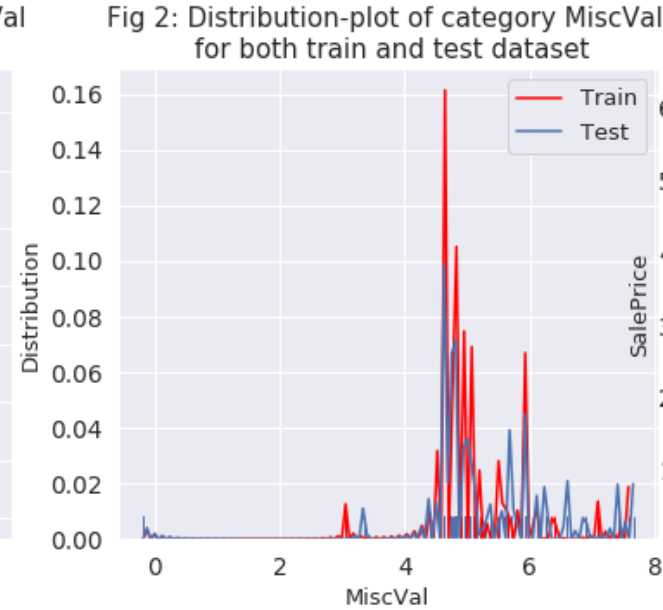
Before skewness removal:



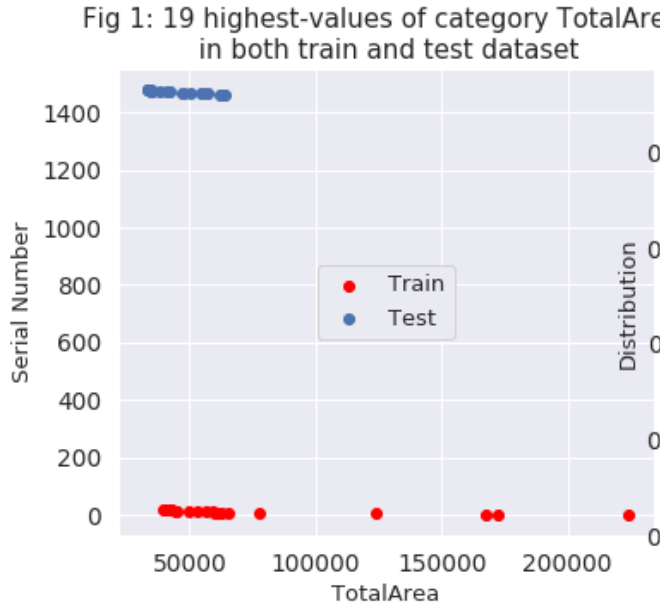


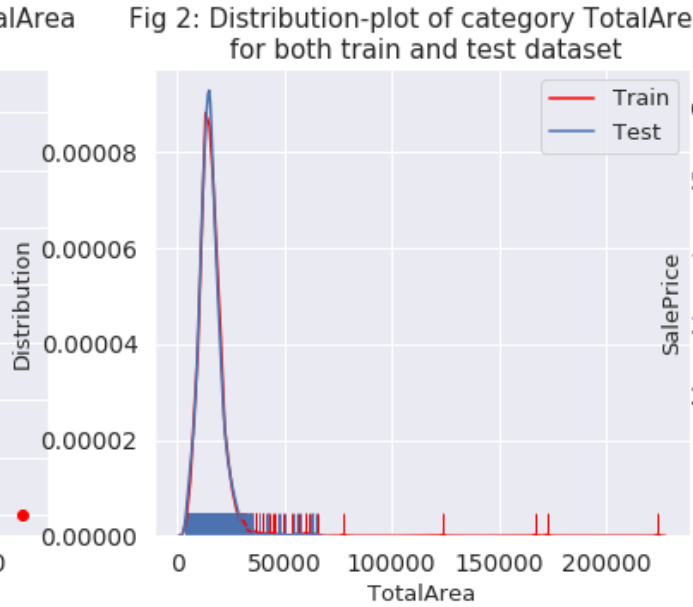
After skewness removal:



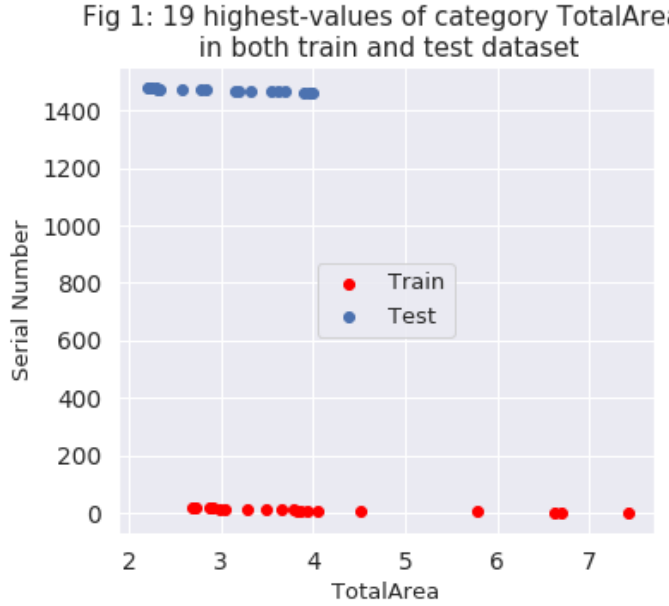


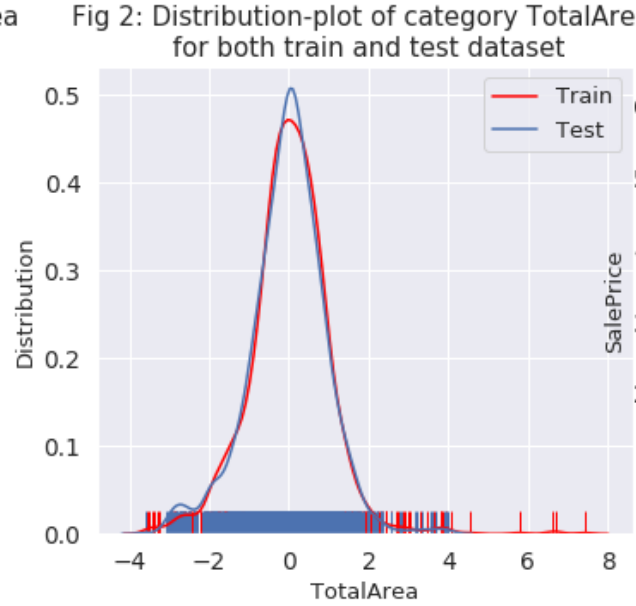
Before skewness removal:





After skewness removal:





Most of the scatterplot now seems that they have more linear relationship with saleprice and the distribution graphs are less skewed and close to normal distribution. Finally due to standarization all of the features are now in same scale this will also help us to converge. We can see that the distribution improved a little bit due to log transformation.

Missing Value Check

In [128]:
```
total = all_df.isnull().sum().sort_values(ascending=False)
percent = (all_df.isnull().sum()/all_df.isnull().count()).sort_values(ascending=False)


missing_data = pd.concat([total, percent], axis=1,
                         keys=['Total', 'Percent' ])
missing_data.head()
```

Out[128]:

	Total	Percent
NeighborhoodBin	0	0.0
Functional	0	0.0
ExterCond	0	0.0
BsmtQual	0	0.0
BsmtCond	0	0.0

Now there is no missing data in any of the train or test dataset so we can proceed further.

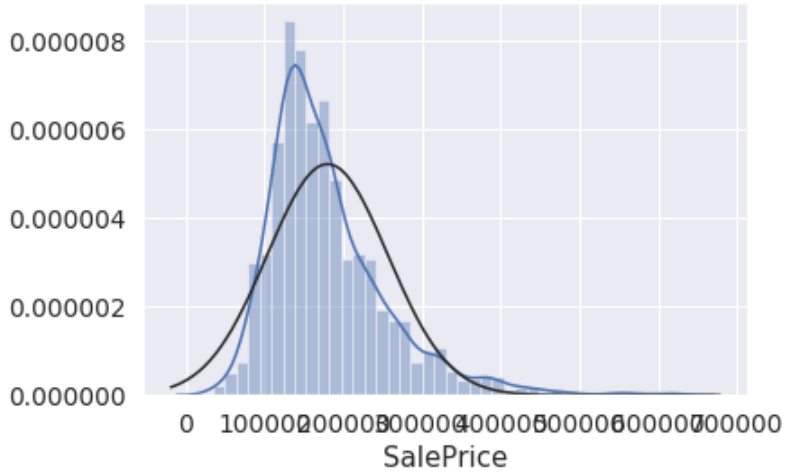
log transform

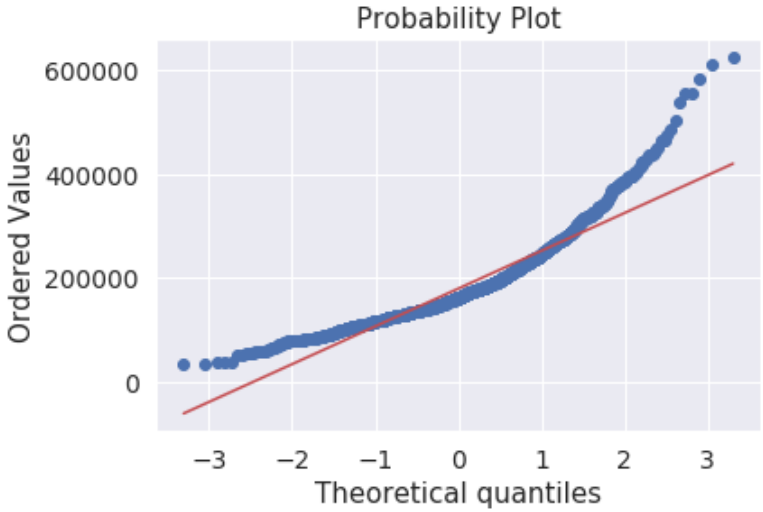
According to Hair et al. (2013), four assumptions should be tested:

- **Normality** - When we talk about normality what we mean is that the data should look like a normal distribution. This is important because several statistic tests rely on this (e.g. t-statistics). In this exercise we'll just check univariate normality for 'SalePrice' (which is a limited approach). Remember that univariate normality doesn't ensure multivariate normality (which is what we would like to have), but it helps. Another detail to take into account is that in big samples (>200 observations) normality is not such an issue. However, if we solve normality, we avoid a lot of other problems (e.g. heteroscedacity) so that's the main reason why we are doing this analysis.
- **Homoscedasticity** - Homoscedasticity refers to the 'assumption that dependent variable(s) exhibit equal levels of variance across the range of predictor variable(s)' (Hair et al., 2013). Homoscedasticity is desirable because we want the error term to be the same across all values of the independent variables.
- **Linearity** - The most common way to assess linearity is to examine scatter plots and search for linear patterns. If patterns are not linear, it would be worthwhile to explore data transformations. However, we'll not get into this because most of the scatter plots we've seen appear to have linear relationships.

'SalePrice' is not normal. It shows 'peakedness', positive skewness and does not follow the diagonal line. But a simple data transformation can solve the problem.

In [129]:
```
from scipy.stats import norm
from scipy import stats
#histogram and normal probability plot
sns.distplot(target, fit=norm);
fig = plt.figure()
res = stats.probplot(target, plot=plt)
```





We take the log here because the error metric is between the log of the SalePrice and the log of the predicted price. That does mean we need to exp() the prediction to get an actual sale price.

In [130]:
```
temp_var = target.values.copy()
temp_var = np.log(temp_var)
target = pd.DataFrame(temp_var, columns=['SalePrice'])
# target["SalePrice"] = np.log(temp_var)
# train_processed.drop(["SalePrice"], axis=1, inplace=True)

print("Training set size:", train_processed.shape)
print("Test set size:", test_processed.shape)
```

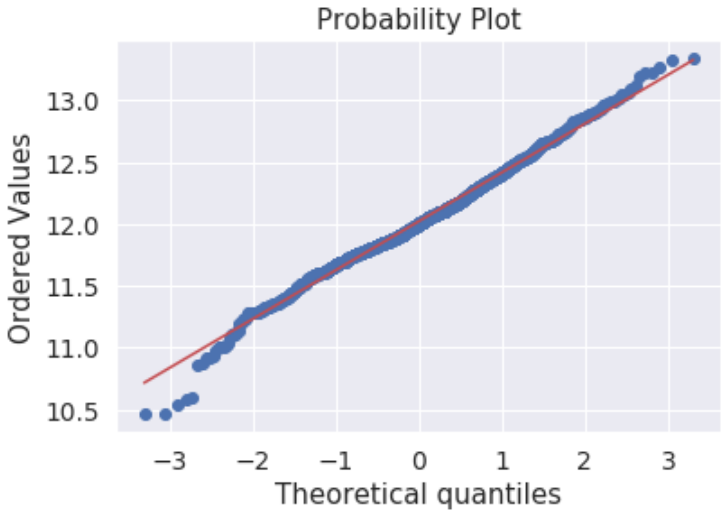
```
Training set size: (1456, 111)
Test set size: (1459, 111)
```

Now we can see the following graph is normal and the probability plot reflects linearity.

In [131]:
```
print(train_processed.shape)
print(target.shape)
# print( test_processed.shape)
# print(train.shape)
```

```
(1456, 111)
(1456, 1)
```

In [132]:
```
from scipy.stats import norm
from scipy import stats
#histogram and normal probability plot
sns.distplot(target['SalePrice'], fit=norm);
fig = plt.figure()
res = stats.probplot(target['SalePrice'], plot=plt)
```



splitting into train and test dataset

In [133]:
```
train_processed = all_df[:ntrain]
test_processed = all_df[ntrain:]

print("shape of train :" , train_processed.shape)
print("shape of test :" , test_processed.shape)
```

```
shape of train : (1456, 111)
shape of test : (1459, 111)
```

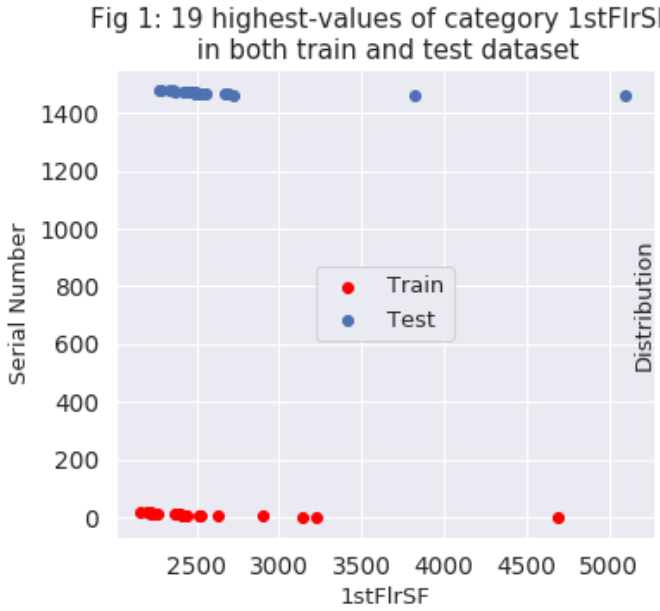
Outlier Crosscheck

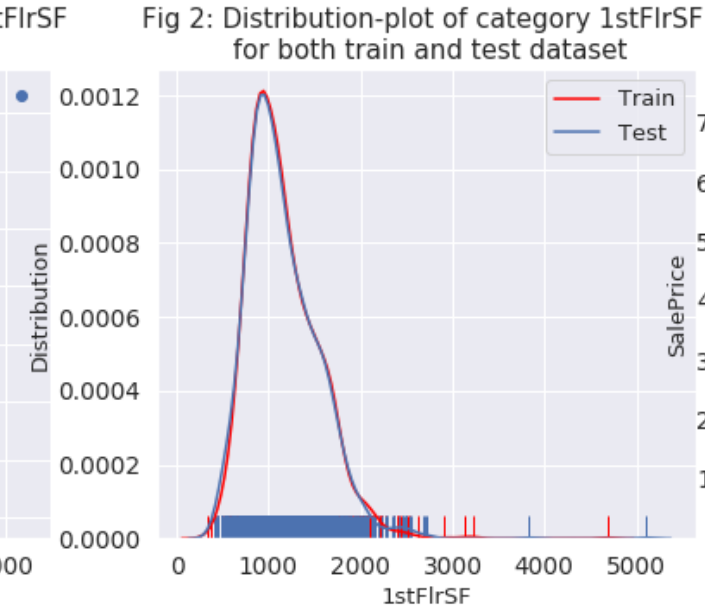
In this section we are checking again If any outlier remains after all the data processing. And the distribution plot will help us to realize the difference before and after normalization. Most of them became more close to normal distribution and less skewed after the processing. So we are not going to normalize them again.

In [134]:

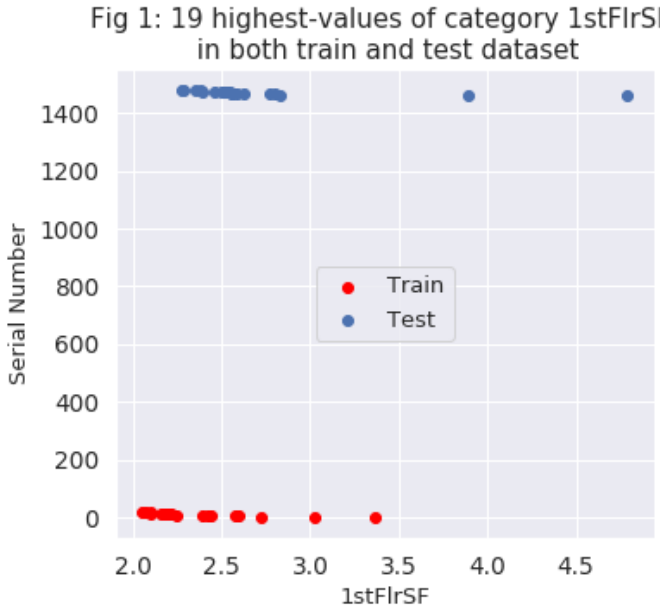
```
from IPython.display import Markdown, display
def printmdmd(string):
    display(Markdown("***"+string+"***"))

printmdmd('Before outlier-removal:')
outlier_check_plot('1stFlrSF',old_train_outlier_flag, old_test_outlier_flag, old_train_outlier_flag.SalePrice)
printmd('After outlier-removal:')
outlier_check_plot('1stFlrSF' , train_processed, test_processed, target.SalePrice)

printmd('Before outlier-removal:')
outlier_check_plot('BsmtFinSF1',old_train_outlier_flag, old_test_outlier_flag, old_target_outlier_flag)
printmd('After outlier-removal:')
outlier_check_plot('BsmtFinSF1', train_processed, test_processed, target.SalePrice)

printmd('Before outlier-removal:')
outlier_check_plot('LotArea',old_train_outlier_flag, old_test_outlier_flag, old_target_outlier_flag)
printmd('After outlier-removal:')
outlier_check_plot('LotArea', train_processed, test_processed, target.SalePrice)

printmd('Before outlier-removal:')
outlier_check_plot('GrLivArea',old_train_outlier_flag, old_test_outlier_flag, old_target_outlier_flag)
printmd('After outlier-removal:')
outlier_check_plot('GrLivArea', train_processed, test_processed, target.SalePrice)

printmd('Before outlier-removal:')
outlier_check_plot('MasVnrArea',old_train_outlier_flag, old_test_outlier_flag, old_target_outlier_flag)
printmd('After outlier-removal:')
outlier_check_plot('MasVnrArea', train_processed, test_processed, target.SalePrice)

printmd('Before outlier-removal:')
outlier_check_plot('TotalBsmtSF',old_train_outlier_flag, old_test_outlier_flag, old_target_outlier_flag)
printmd('After outlier-removal:')
outlier_check_plot('TotalBsmtSF', train_processed, test_processed, target.SalePrice)

printmd('Before outlier-removal:')
outlier_check_plot('TotalBsmtSF',old_train_outlier_flag, old_test_outlier_flag, old_target_outlier_flag)
printmd('After outlier-removal:')
outlier_check_plot('TotalBsmtSF', train_processed, test_processed, target.SalePrice)
```

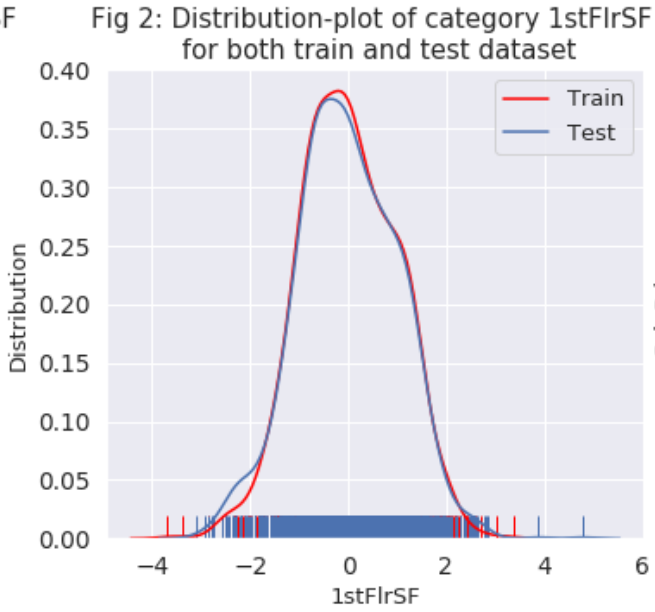
Before outlier-removal:

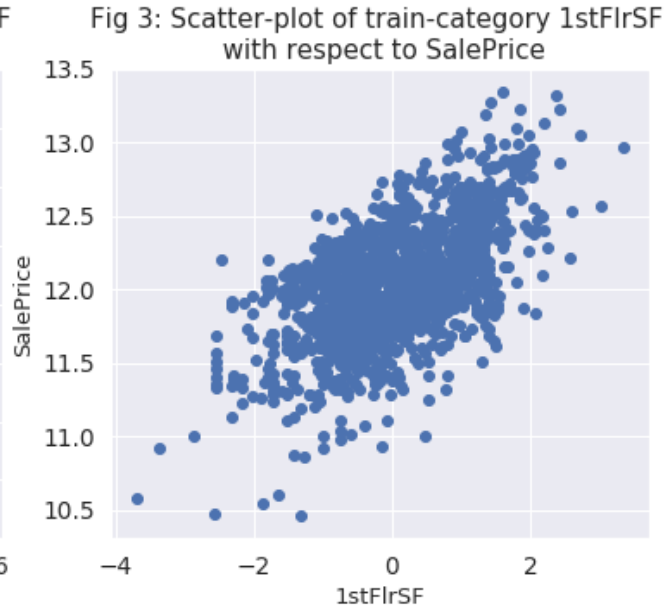




After outlier-removal:

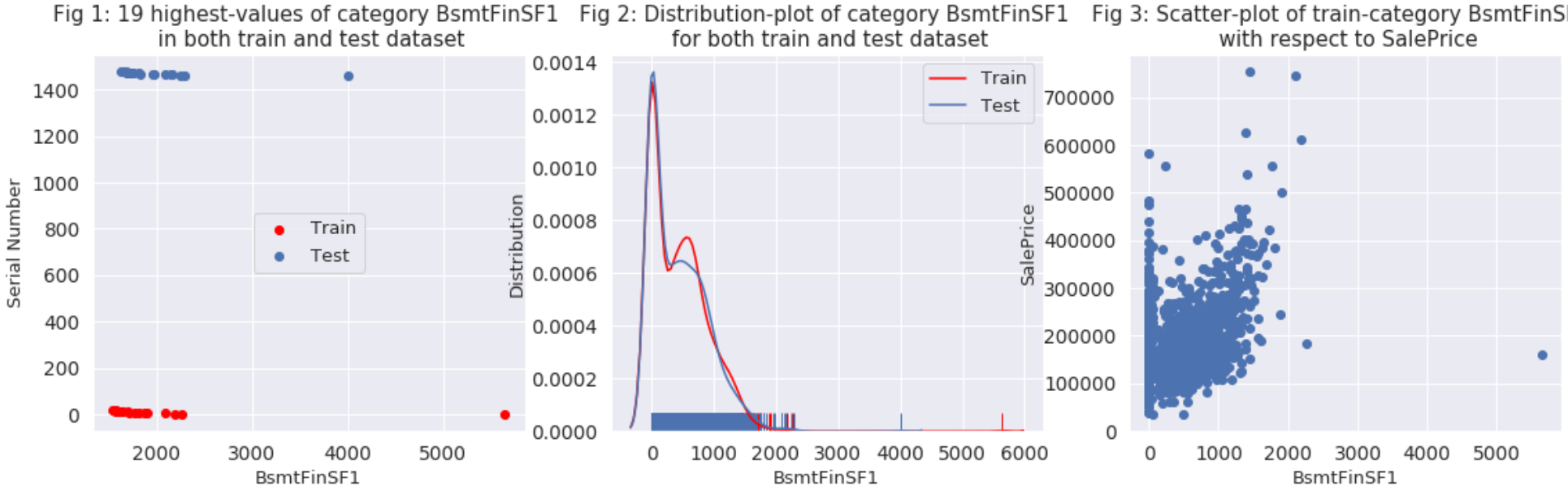




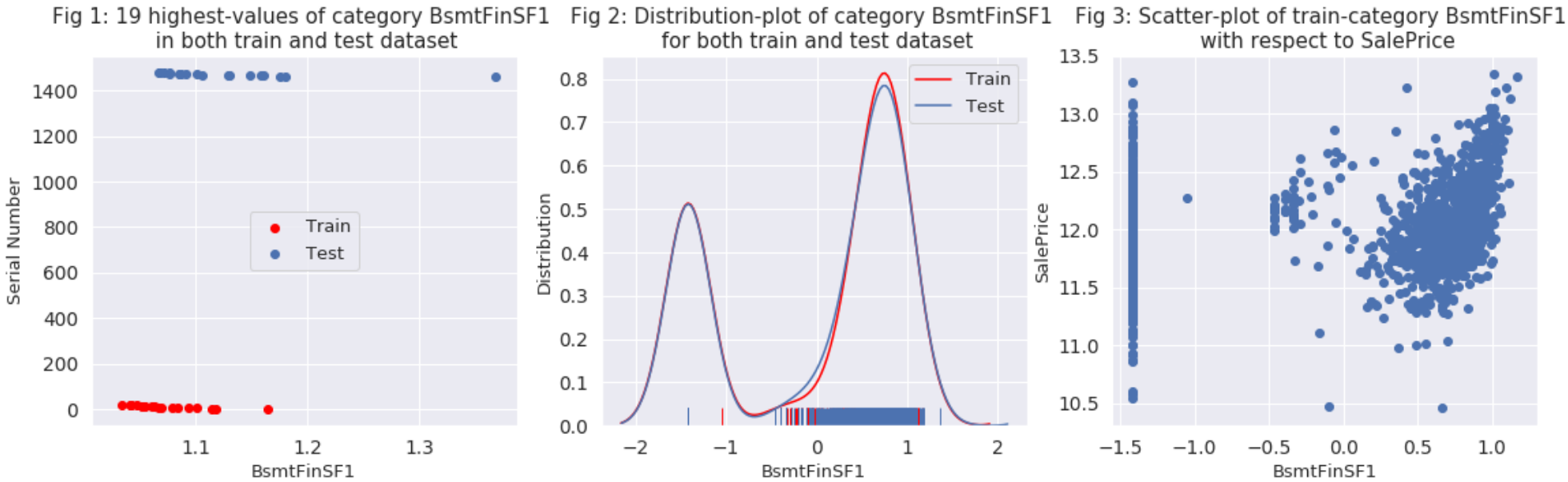


Before outlier-removal:

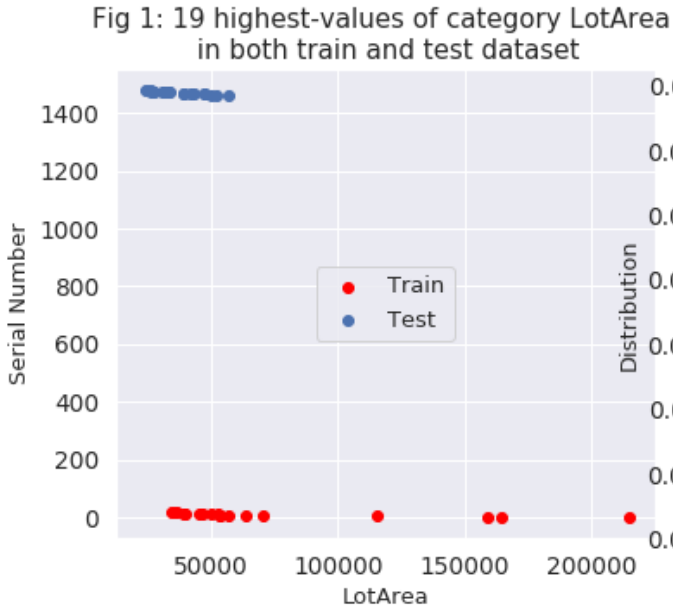
```
/home/navid/anaconda3/envs/tf/lib/python3.6/site-packages/statsmodels/nonparametric/kde.py:448: RuntimeWarning: invalid value enco
untered in greater
  X = X[np.logical_and(X > clip[0], X < clip[1])] # won't work for two columns.
/home/navid/anaconda3/envs/tf/lib/python3.6/site-packages/statsmodels/nonparametric/kde.py:448: RuntimeWarning: invalid value enco
untered in less
  X = X[np.logical_and(X > clip[0], X < clip[1])] # won't work for two columns.
```

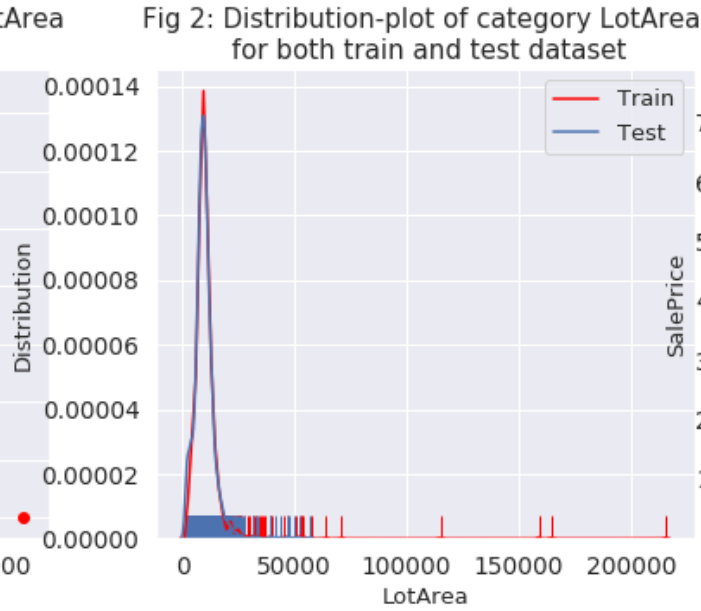


After outlier-removal:

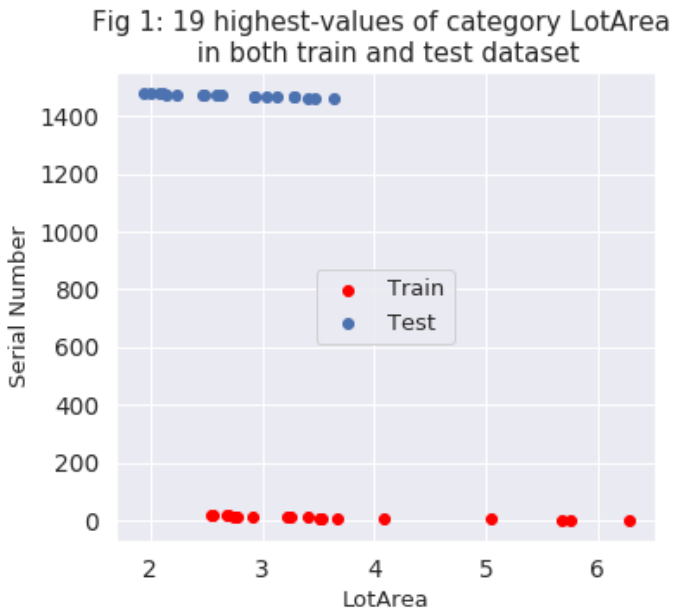


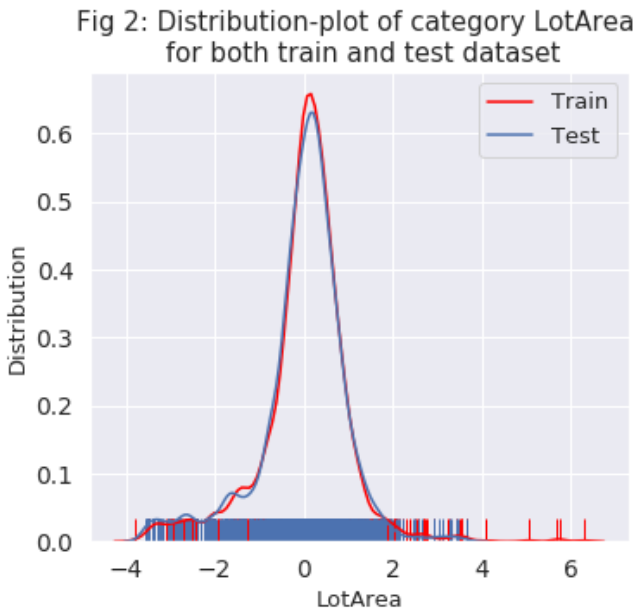
Before outlier-removal:

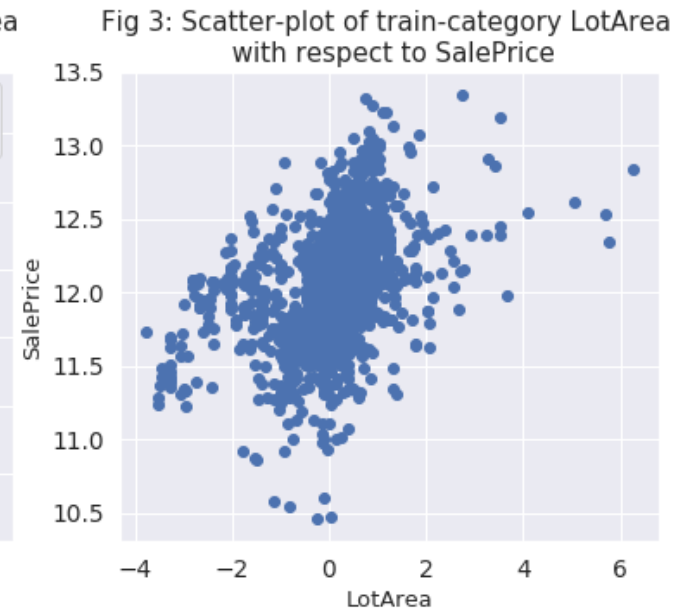




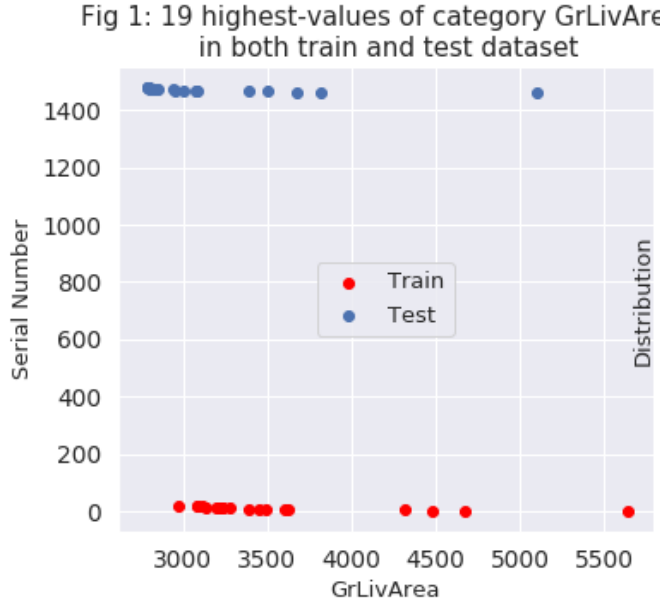
After outlier-removal:

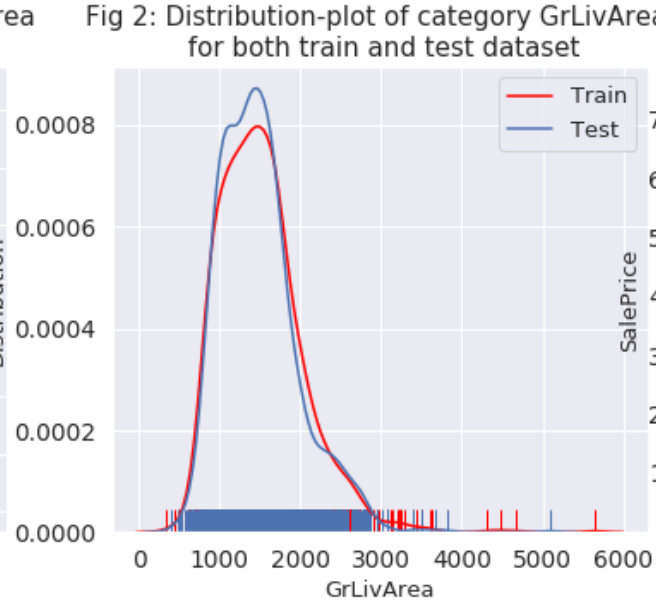


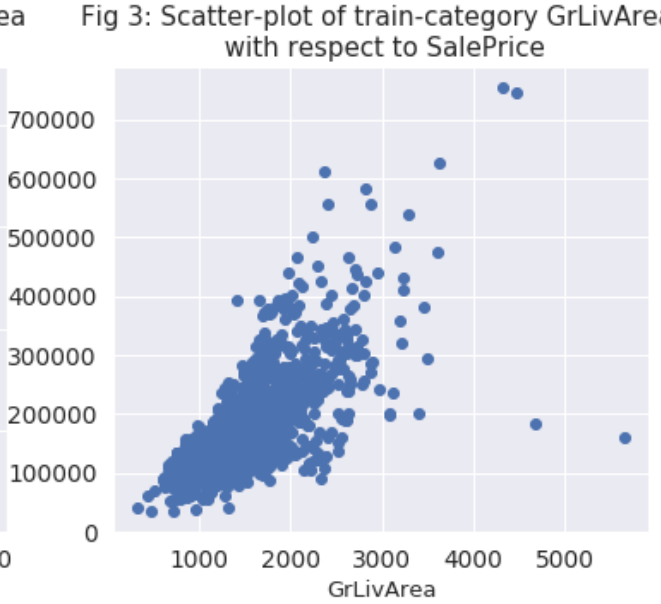




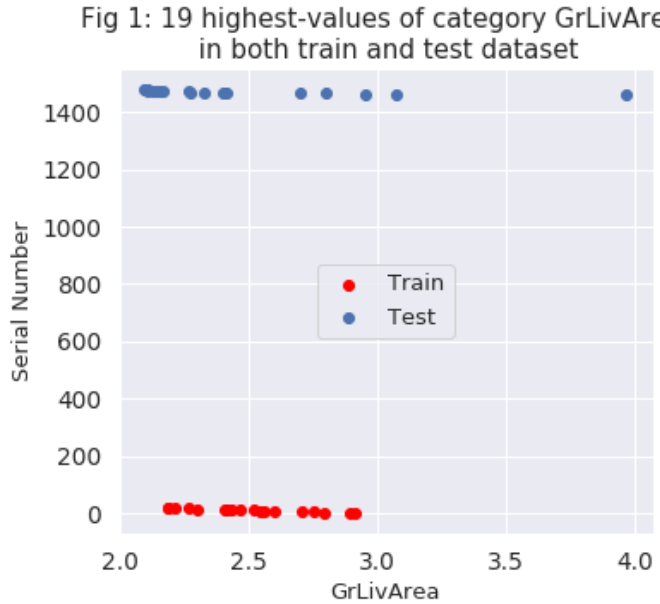
Before outlier-removal:

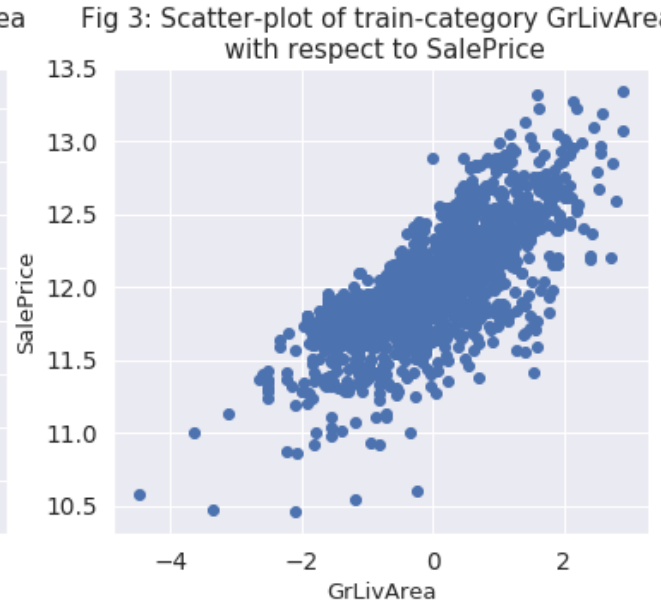




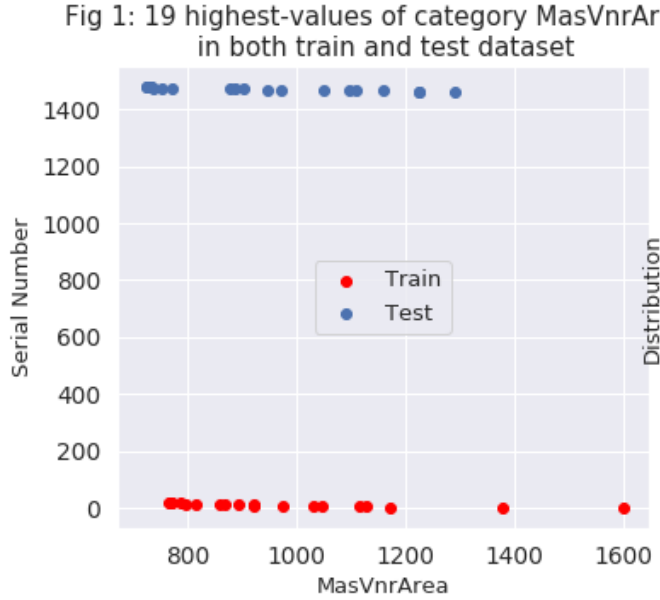


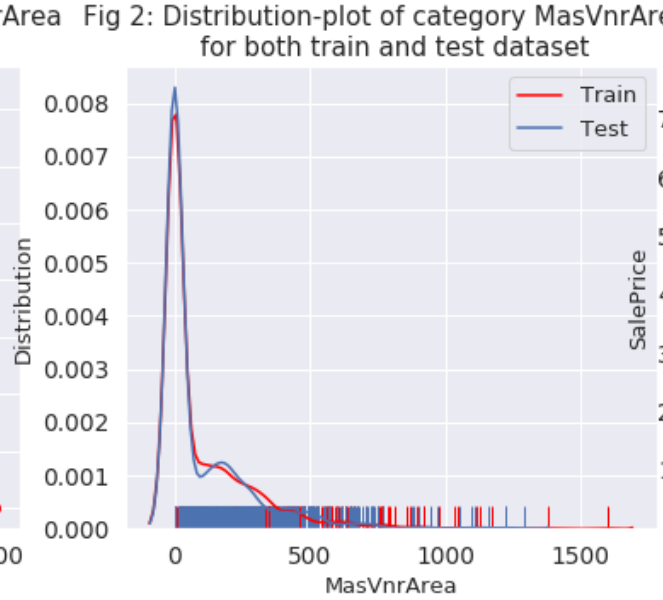
After outlier-removal:

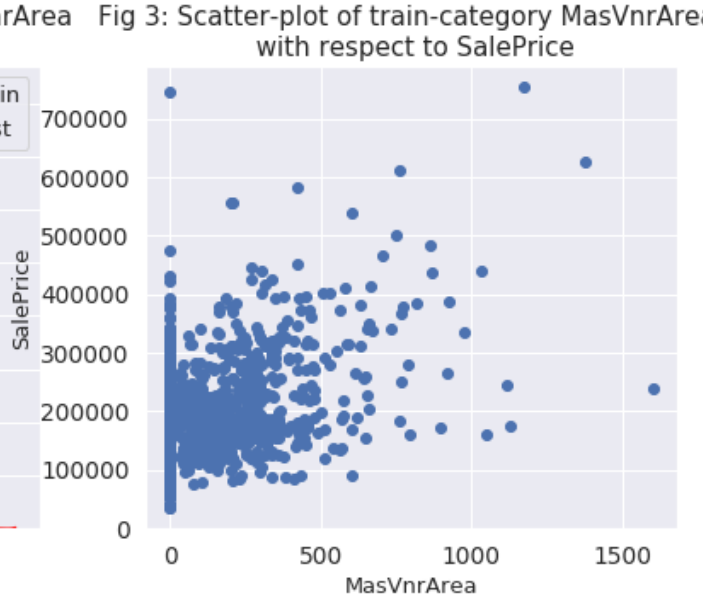




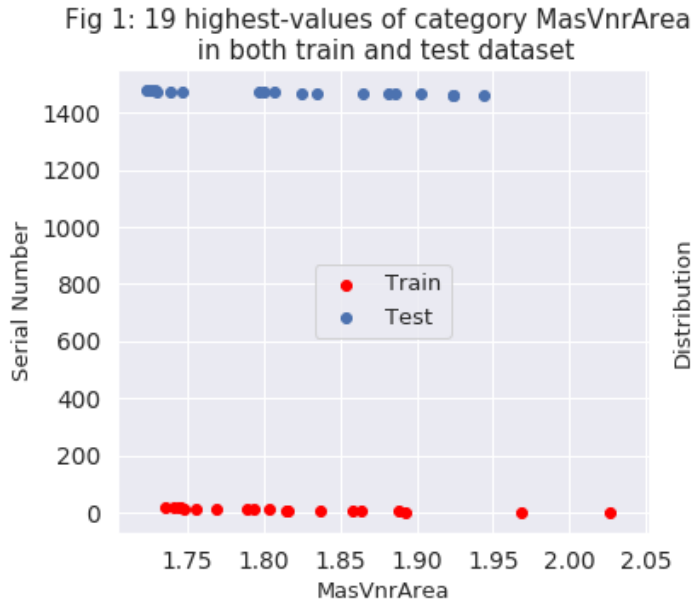
Before outlier-removal:

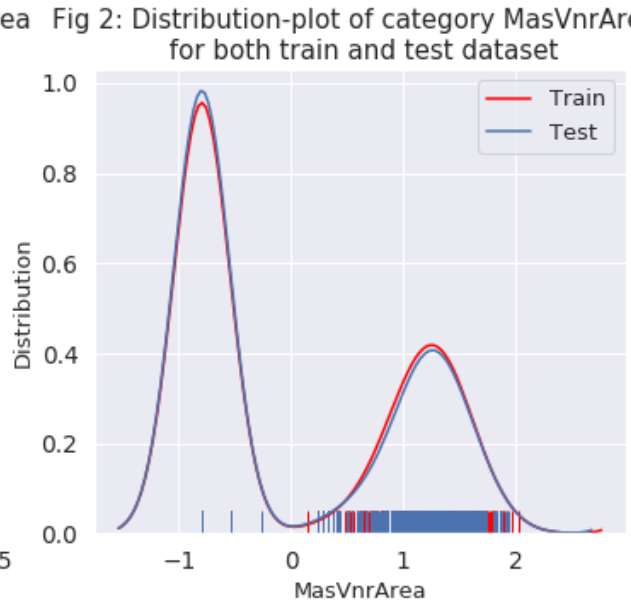


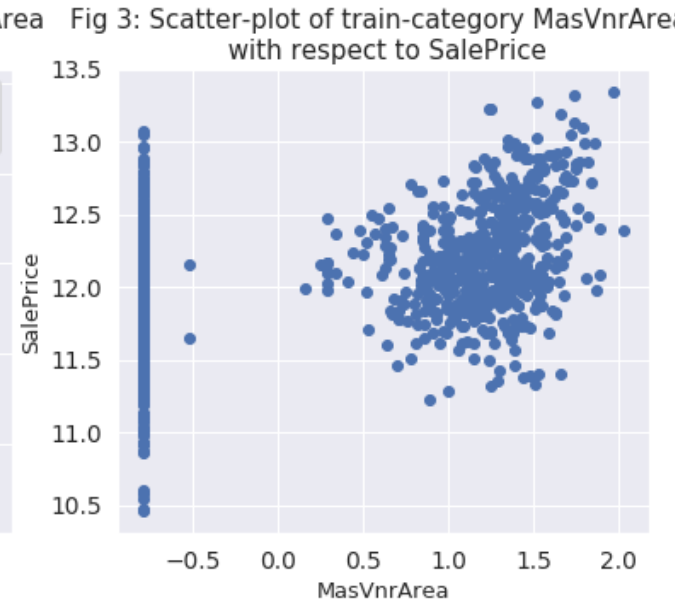




After outlier-removal:

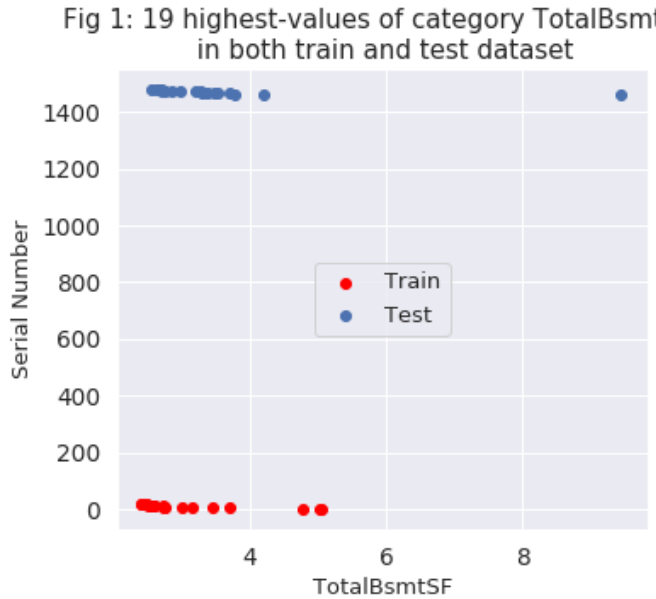


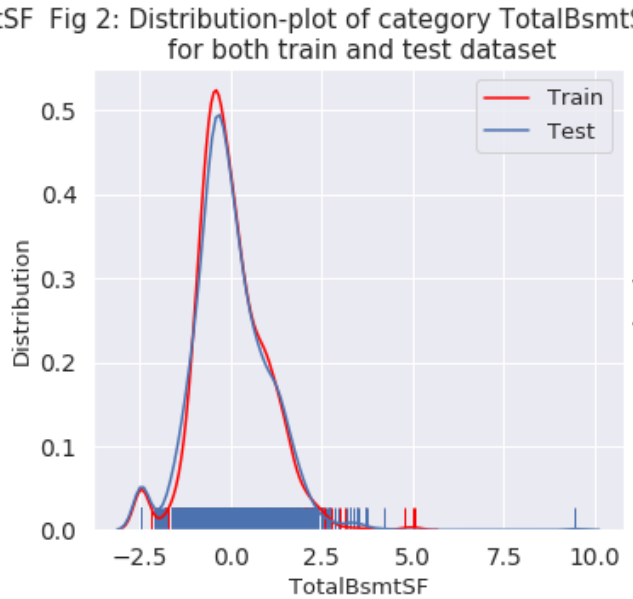


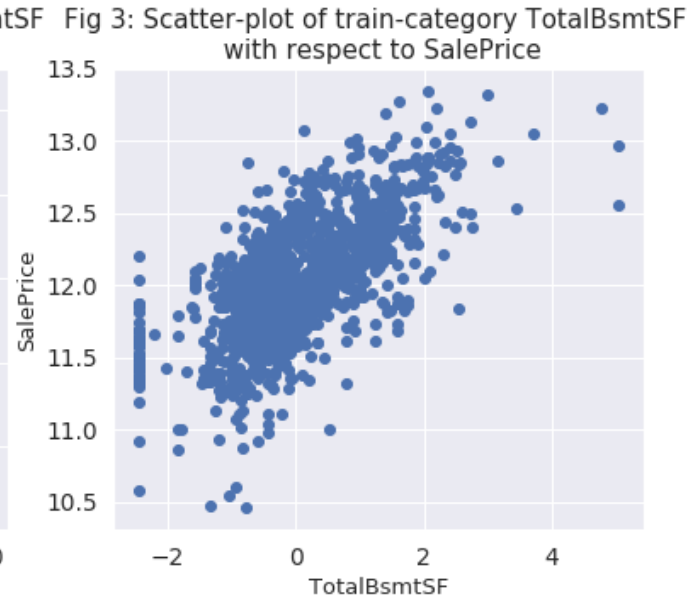


Before outlier-removal:

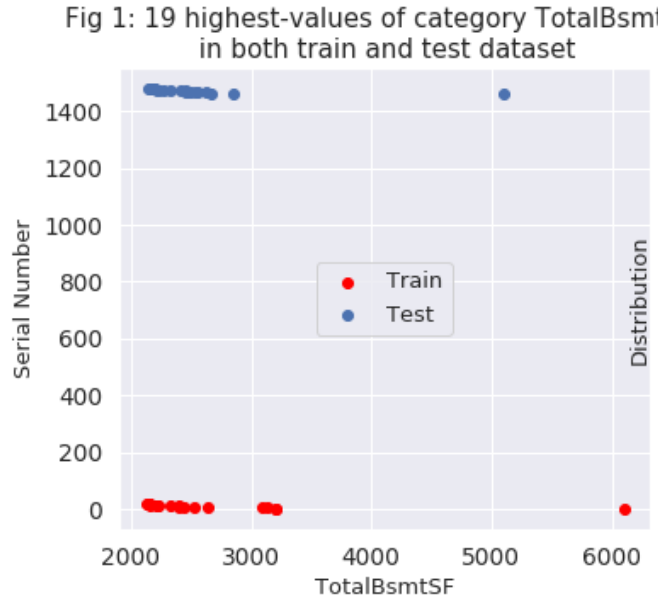
After outlier-removal:

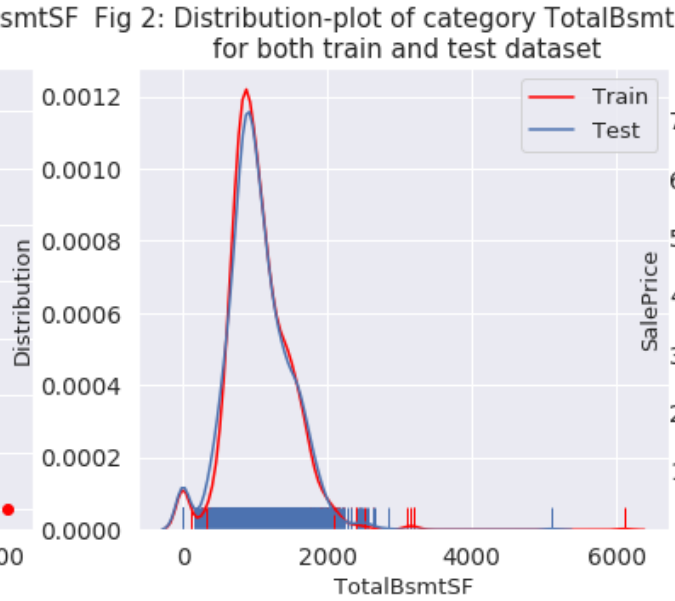


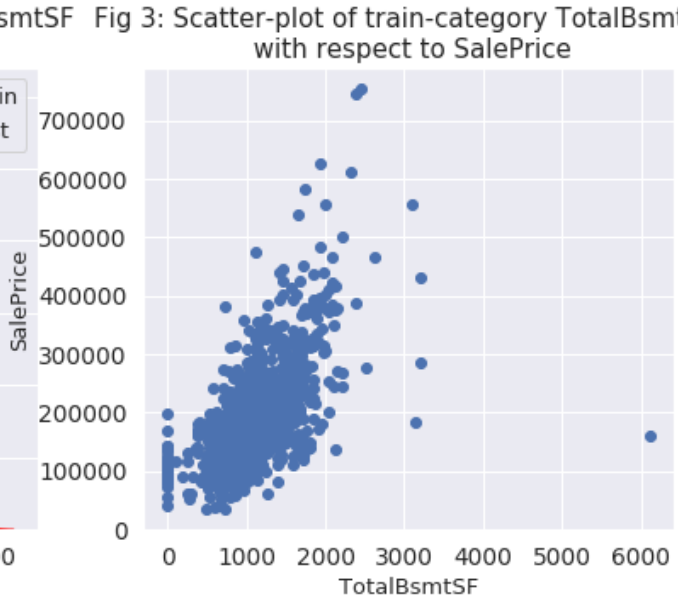




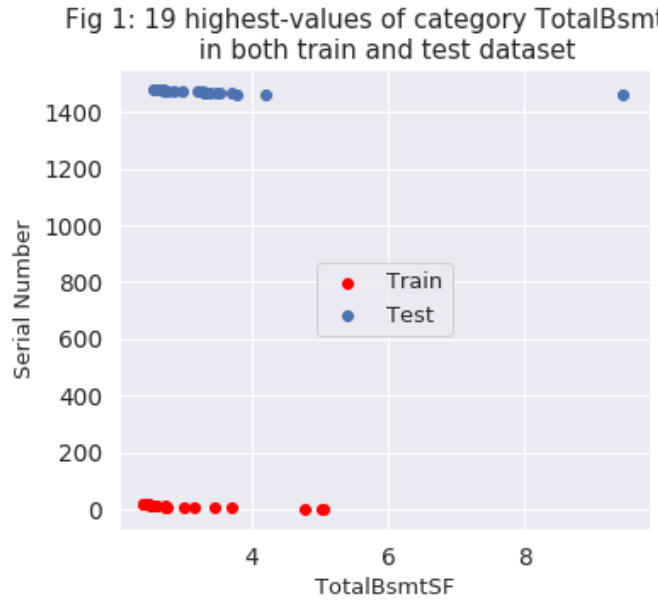
Before outlier-removal:

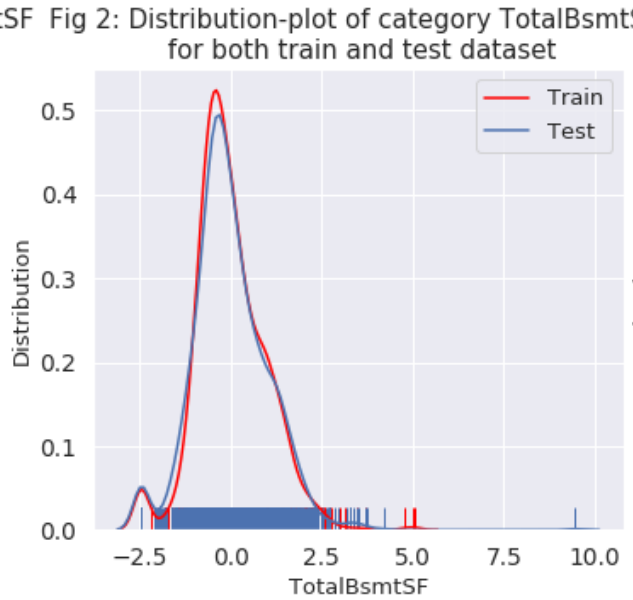


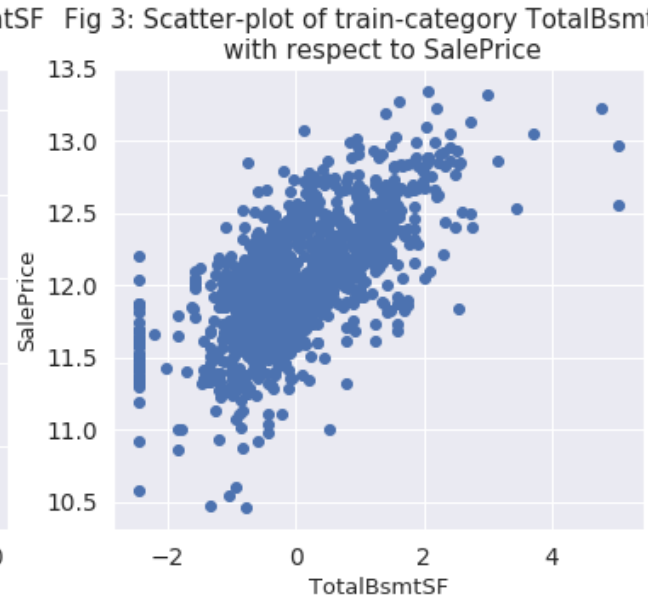




After outlier-removal:







Most of the scatterplot now seems that they have linear relationship with saleprice and the distribution graphs are less skewed and close to normal distribution. Finally due to standarization all of the features are now in same scale this will also help us to converge.

This time we can see that the distribution improved a little bit due to log transformation and I was expecting that few outliers we observed earlier are no longer seems to be a outlier. Only the common outlier was the actual source of the problem. So we can now proceed to feed these data to our model.

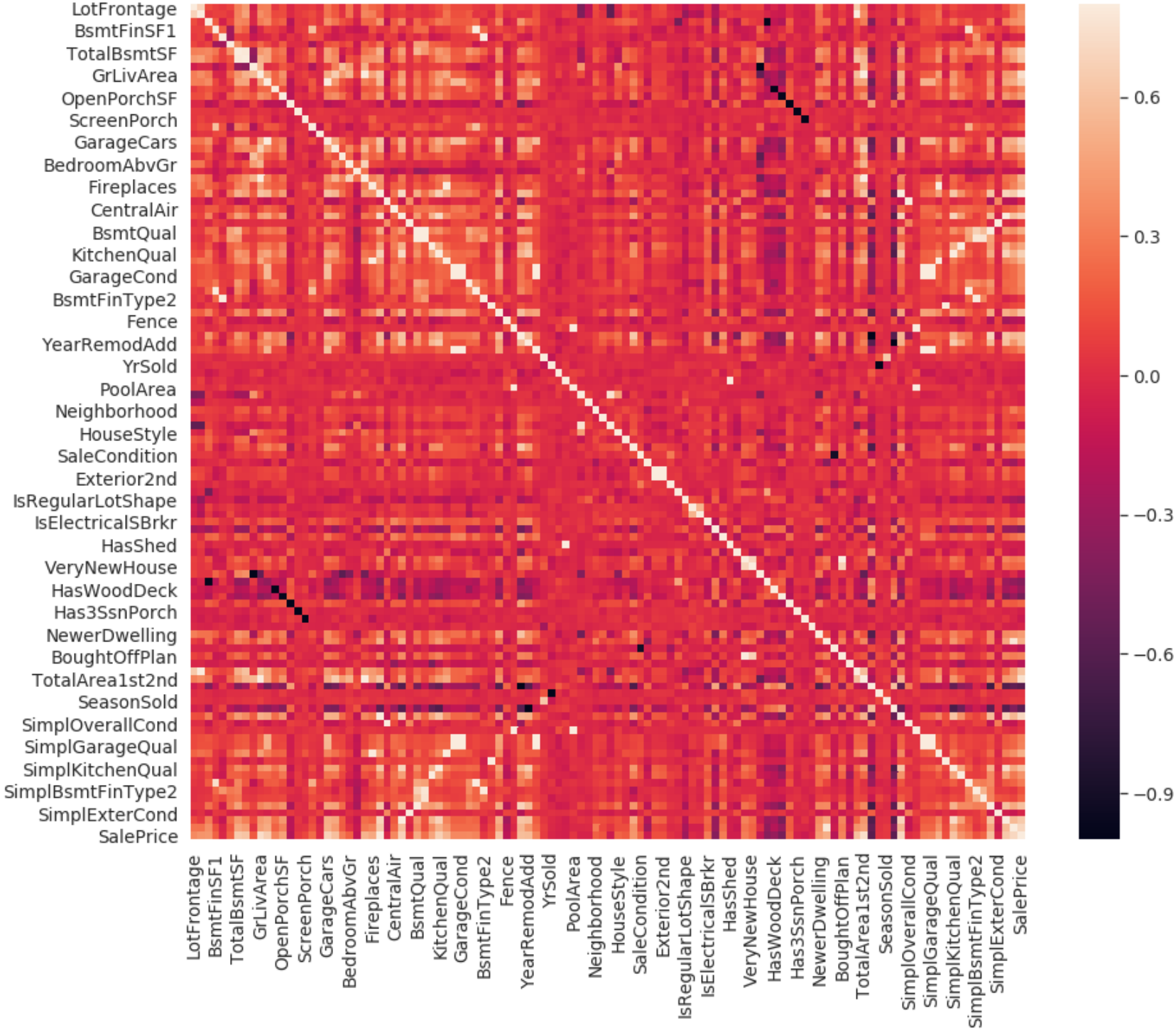
Corelation Matrix after procesing

In [135]:
```
print(train_processed.shape)
print(target.shape)
# print( test_processed.shape)
# print(train.shape)
```

```
(1456, 111)
(1456, 1)
```

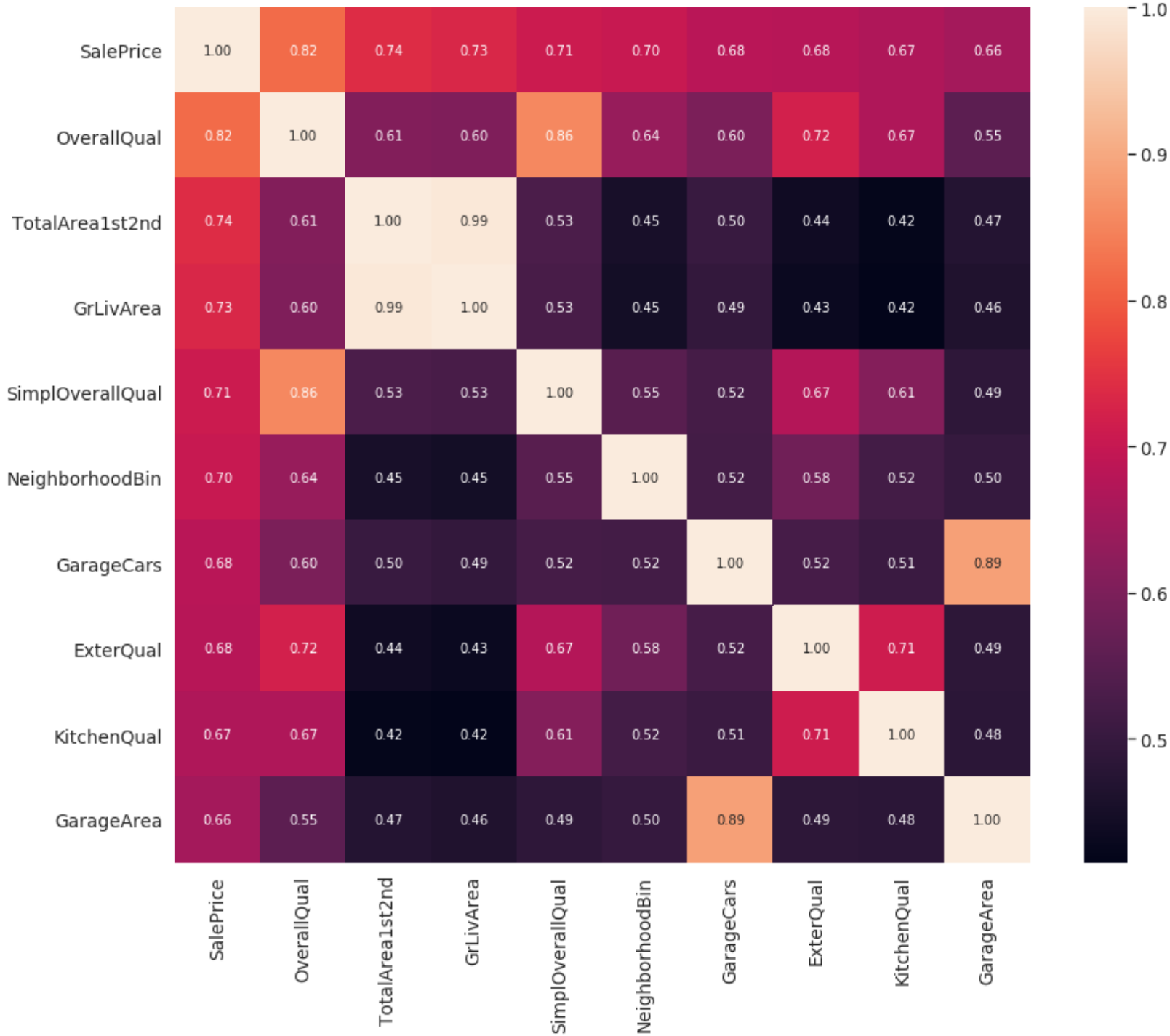
In [136]:
```
abc = train_processed.copy()
abc['SalePrice'] = target.SalePrice.copy()
```

In [137]:
```
#correlation matrix
corrmat = abc.corr()
f, ax = plt.subplots(figsize=(15, 12))
sns.set(font_scale=1.25)
sns.heatmap(corrmat, vmax=.8, square=True);
```


LotFrontage
BsmtFinSF1
TotalBsmtSF
GrLivArea
OpenPorchSF
ScreenPorch
GarageCars
BedroomAbvGr
Fireplaces
CentralAir
BsmtQual
KitchenQual
GarageCond
BsmtFinType2
Fence
YearRemodAdd
YrSold
PoolArea
Neighborhood
HouseStyle
SaleCondition
Exterior2nd
IsRegularLotShape
IsElectricalSBrkr
HasShed
VeryNewHouse
HasWoodDeck
Has3SsnPorch
NewerDwelling
BoughtOffPlan
TotalArea1st2nd
SeasonSold
SimplOverallCond
SimplGarageQual
SimplKitchenQual
SimplBsmtFinType2
SimplExterCond
SalePrice
0.6
0.3
0.0
−0.3
−0.6
−0.9

We can see that above graph is almost completely red that means no feature have any relation with another feature. That means all the features are now independent. So our data processing part should be good enough to get good results.

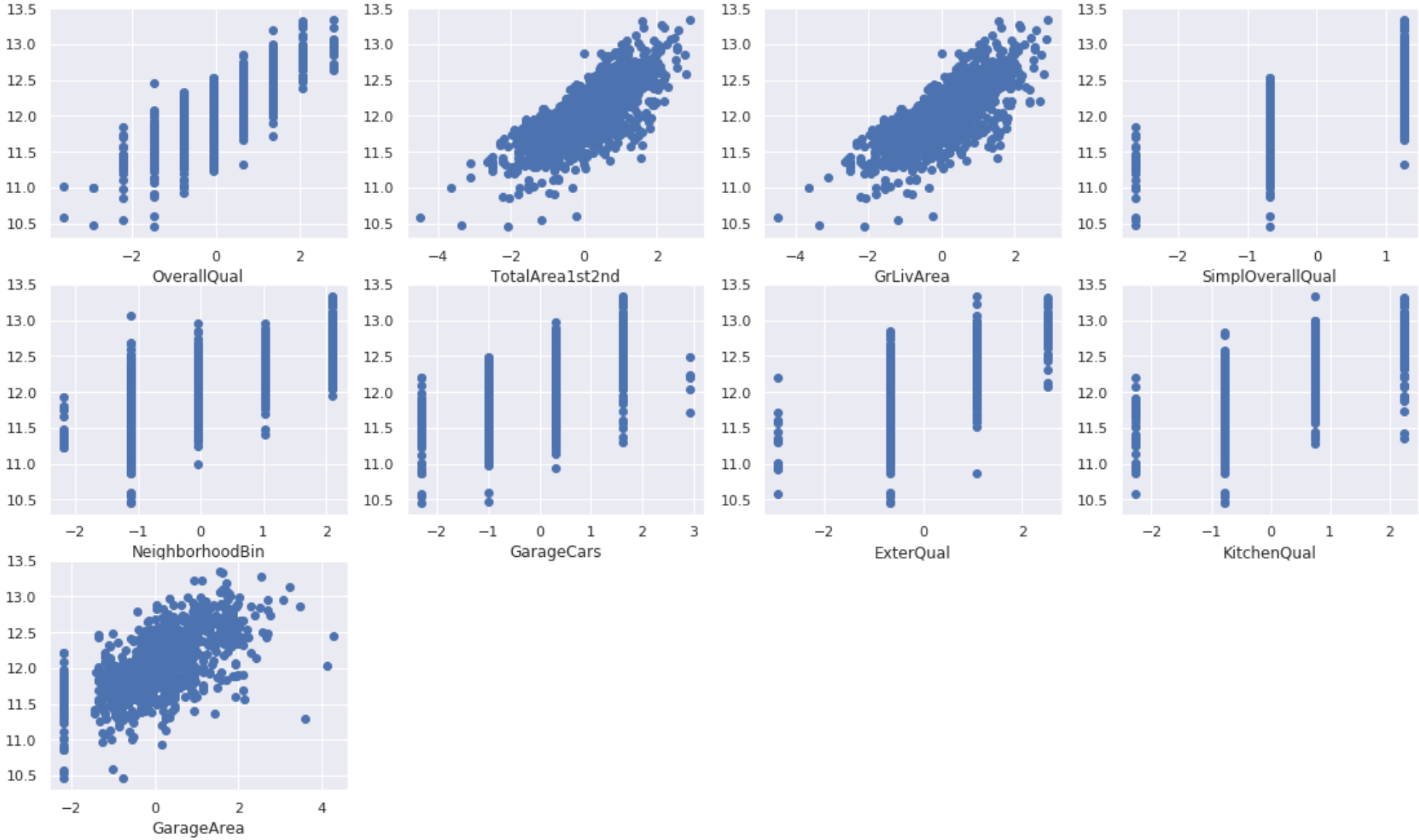
In [138]:
```
#saleprice correlation matrix
k = 10 #number of variables for heatmap
cols = corrmat.nlargest(k, 'SalePrice')['SalePrice'].index
cm = np.corrcoef(abc[cols].values.T)
f, ax = plt.subplots(figsize=(15, 12))
sns.set(font_scale=1.25)
hm = sns.heatmap(cm, cbar=True, annot=True, square=True, fmt='.2f', annot_kws={'size': 10}, yticklabels=cols.values, xticklabels=c
ols.values)
plt.show()
```


SalePrice 1.00 0.82 0.74 0.73 0.71 0.70 0.68 0.68 0.67 0.66
OverallQual 0.82 1.00 0.61 0.60 0.86 0.64 0.60 0.72 0.67 0.55
TotalArea1st2nd 0.74 0.61 1.00 0.99 0.53 0.45 0.50 0.44 0.42 0.47
GrLivArea 0.73 0.60 0.99 1.00 0.53 0.45 0.49 0.43 0.42 0.46
SimplOverallQual 0.71 0.86 0.53 0.53 1.00 0.55 0.52 0.67 0.61 0.49
NeighborhoodBin 0.70 0.64 0.45 0.45 0.55 1.00 0.52 0.58 0.52 0.50
GarageCars 0.68 0.60 0.50 0.49 0.52 0.52 1.00 0.52 0.51 0.89
ExterQual 0.68 0.72 0.44 0.43 0.67 0.58 0.52 1.00 0.71 0.49
KitchenQual 0.67 0.67 0.42 0.42 0.61 0.52 0.51 0.71 1.00 0.48
GarageArea 0.66 0.55 0.47 0.46 0.49 0.50 0.89 0.49 0.48 1.00
SalePrice
OverallQual
TotalArea1st2nd
GrLivArea
SimplOverallQual
NeighborhoodBin
GarageCars
ExterQual
KitchenQual
GarageArea
1.0
0.9
0.8
0.7
0.6
0.5

We can see that GrLivArea and TotalArea1st2nd is very close and in the following graph we will see that the graph is also same for both of the feature. But by dropping one of them performance does not improves rather decreases sometime so I am keeping both of them.

The following 9 features are the most important feature for determining the SalePrice and they also don't have any outlier

In [139]:
```
 # A FUNCTION TO SCATTER-PLOT ALL SELECTED FEATURES AGAINST SALEPRICE
def relation_with_SalePrice(c,column):
    plt.subplot(5, 4, c)
    plt.scatter(x = train_processed[column], y = target.SalePrice)
    plt.xlabel(column)
c=1
sns.set(font_scale=1)
plt.subplots(figsize=(19, 19))

if 'SalePrice' in cols:
    cols = cols.drop('SalePrice')

for item in cols:
    relation_with_SalePrice(c,item)
    c=c+1
plt.show()
```



Split Data for training and testing

In this Section We have split the training dataset into two part. First one is called train and other is called val (means validation set).Training set contains X_train and y_train.Validation set also contains X_val and y_val. X means this SalePrice is excluded. Again Y means this portion only contains Saleprice. I have used 80-20 split where training contains 80% data and validation contains 20% data. I have used kaggle testing set for testing them (variable name is test_processed) and the result of the kaggle testing is also included as a screenshot after accuracy section.

In [149]:
```
X_train, X_val, y_train, y_val = train_test_split(train_processed,
                                                  target,
#                                                   train_size = 0.99,
                                                  test_size = 0.2,
                                                  random_state = 0,
                                                  shuffle = True
                                                 )
```

Following section changes the training set to 100% when we set submit=True. The reason behind it is when we train with full dataset then we use to get better accuracy. But we will set that True only when we are going to submit the prediction of the tess_proceed to kaggle.

In [150]:
```
prediction_dict = dict()
submit_prediction_dict = dict()

submit = False
save_score = False

if submit :
    X_train = train_processed
    y_train = target
else:
    X_train = X_train
    y_train = y_train
```

Testing different models

RMSE

Following function calculates root mean squire error

What is RMSE ?

The root-mean-square deviation (RMSD) or root-mean-square error (RMSE) (or sometimes root-mean-squared error) is a frequently used measure of the differences between values (sample or population values) predicted by a model or an estimator and the values observed. The RMSD represents the square root of the second sample moment of the differences between predicted values and observed values or the quadratic mean of these differences. These deviations are called residuals when the calculations are performed over the data sample that was used for estimation and are called errors (or prediction errors) when computed out-of-sample. The RMSD serves to aggregate the magnitudes of the errors in predictions for various times into a single measure of predictive power. RMSD is a measure of accuracy, to compare forecasting errors of different models for a particular dataset and not between datasets, as it is scale-dependent.[1]

In [142]:
```
def rmse(y_true, y_pred):
    return np.sqrt(mean_squared_error(y_true, y_pred))
```

Random Forest Regressor

A random forest is a meta estimator that fits a number of classifying decision trees on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting.

In [143]:
```
my_model = RandomForestRegressor(n_estimators=500,n_jobs=-1)


my_model.fit(X_train, y_train)
prediction = my_model.predict(X_val)
if submit:
    submit_prediction = my_model.predict(test_processed)
    submit_prediction_dict['Random Forest Regressor'] = submit_prediction

prediction_dict['Random Forest Regressor'] = prediction

print('root mean absolute error: ',rmse(y_val, prediction))
print('accuracy score: ',  r2_score(np.array(y_val),prediction) )
```

```
/home/navid/anaconda3/envs/tf/lib/python3.6/site-packages/ipykernel_launcher.py:4: DataConversionWarning: A column-vector y was pa
ssed when a 1d array was expected. Please change the shape of y to (n_samples,), for example using ravel().
  after removing the cwd from sys.path.

root mean absolute error:  0.11885556163719953
accuracy score:  0.9065381567947393
```

DecisionTree

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

In [144]:
```
from sklearn.tree import DecisionTreeRegressor
my_model = DecisionTreeRegressor()

my_model.fit(X_train, y_train)
prediction = my_model.predict(X_val)
prediction_dict['DecisionTree'] = prediction
if submit:
    submit_prediction = my_model.predict(test_processed)
    submit_prediction_dict['DecisionTree'] = submit_prediction

print('root mean absolute error: ',rmse(y_val, prediction))
print('accuracy score: ',  r2_score(np.array(y_val),prediction) )
```

```
root mean absolute error:  0.19970749763486026
accuracy score:  0.7361336167239957
```

Xgboost

XGBoost stands for eXtreme Gradient Boosting. It is an optimized distributed gradient boosting library designed to be highly efficient, flexible and portable. It implements machine learning algorithms under the Gradient Boosting framework. XGBoost provides a parallel tree boosting (also known as GBDT, GBM) that solve many data science problems in a fast and accurate way.

In [145]:
```
from xgboost import XGBRegressor
my_model = XGBRegressor(n_estimators=500, learning_rate=0.05)

my_model.fit(X_train, y_train)
prediction = my_model.predict(X_val)
prediction_dict['Xgboost'] = prediction

if submit:
    submit_prediction = my_model.predict(test_processed)
    submit_prediction_dict['Xgboost'] = submit_prediction


print('root mean absolute error: ',rmse(y_val, prediction))
print('accuracy score: ',  r2_score(np.array(y_val),prediction) )
```

```
root mean absolute error:  0.10851927351325137
accuracy score:  0.9220871519593594
```

Lasso

Lasso (least absolute shrinkage and selection operator; also Lasso or LASSO) is a regression analysis method that performs both variable selection and regularization in order to enhance the prediction accuracy and interpretability of the statistical model it produces.Lasso was originally formulated for least squares models and this simple case reveals a substantial amount about the behavior of the estimator, including its relationship to ridge regression and best subset selection and the connections between lasso coefficient estimates and so-called soft thresholding. It also reveals that (like standard linear regression) the coefficient estimates need not be unique if covariates are collinear.

In [146]:
```
from sklearn.linear_model import Lasso
my_model = Lasso(alpha=5e-3, max_iter=50000)


my_model.fit(X_train, y_train)
prediction = my_model.predict(X_val)
prediction_dict['Lasso'] = prediction

if submit:
    submit_prediction = my_model.predict(test_processed)
    submit_prediction_dict['Lasso'] = submit_prediction

print(' root mean absolute error: ',rmse(y_val, prediction))
print('accuracy score: ',  r2_score(np.array(y_val),prediction) )
```

```
 root mean absolute error:  0.10492714420014548
accuracy score:  0.9271598166676635
```

In the above model alpha is Constant that multiplies the L1 term. For numerical reason we cant set alpha to 0 but keeping alpha low provides good accuracy for out dataset. I have found 5e-4 provides good accuracy.

for 5e-5: root mean absolute error: 0.10973737757187135 accuracy score: 0.9289433650407954

for 1e-5: root mean absolute error: 0.11426822609093419 accuracy score: 0.9229546464396043

for 1e-3: root mean absolute error: 0.10466883446067998 accuracy score: 0.9353556969018821

for 1e-4: root mean absolute error: 0.10658498063306822 accuracy score: 0.9329671780226085

for 5e-3: root mean absolute error: 0.10794617678311977 accuracy score: 0.9312440935471524

ANN

Theory and Basics:

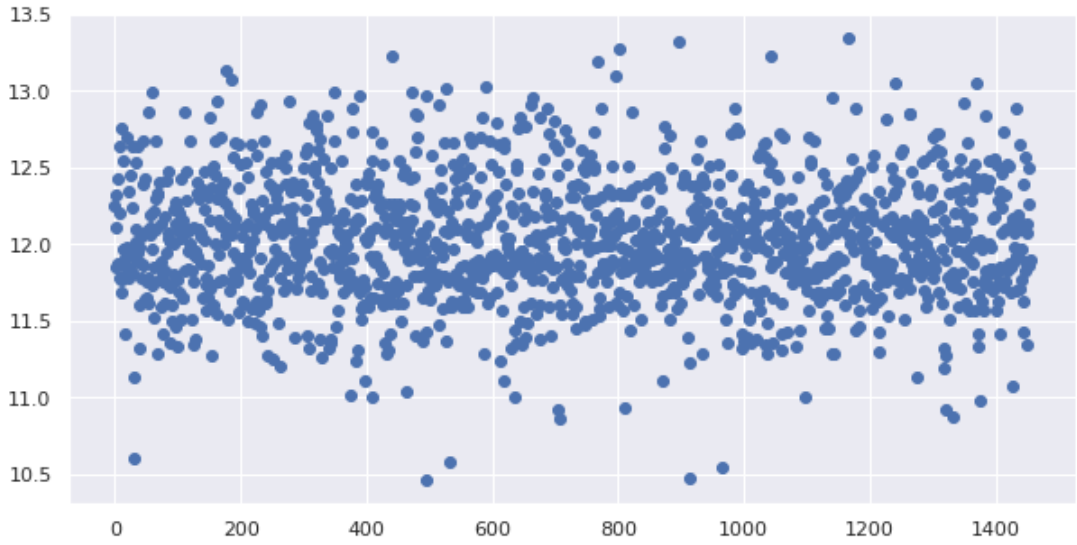
An Artificial Neurol Network (ANN) is a computational model. It is based on the structure and functions of biological neural networks. It works like the way human brain processes information. ANN includes a large number of connected processing units that work together to process information. They also generate meaningful results from it.

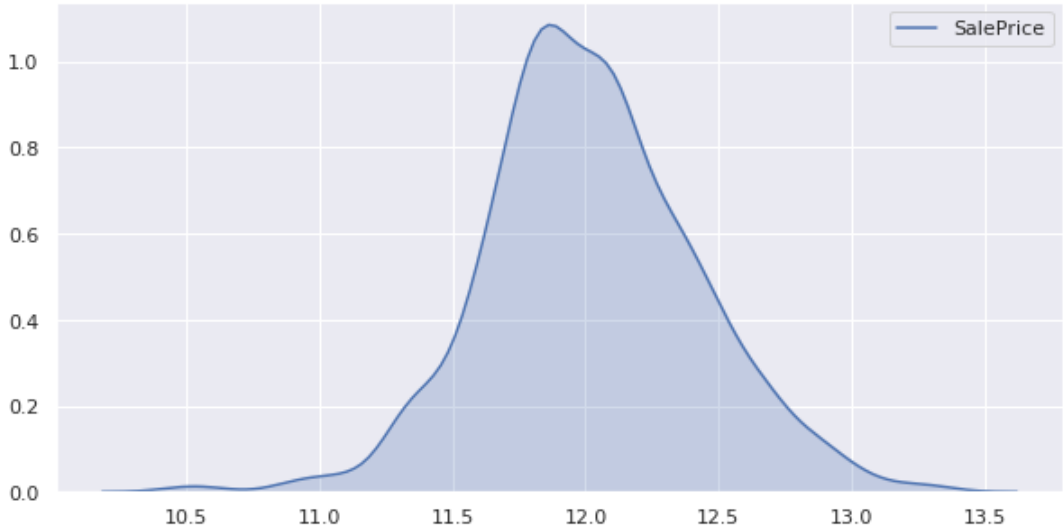
An artificial neuron is a mathematical function conceived as a model of biological neurons, a neural network. Usually each input is separately weighted, and the sum is passed through a non-linear function known as an activation function or transfer function.

The artificial Neural network is typically organized in layers. Layers are being made up of many interconnected ‘nodes’ which contain an ‘activation function’. A neural network may contain the following 3 layers:

- **Input layer** The purpose of the input layer is to receive as input the values of the explanatory attributes for each observation. Usually, the number of input nodes in an input layer is equal to the number of explanatory variables. ‘input layer’ presents the patterns to the network, which communicates to one or more ‘hidden layers’. The nodes of the input layer are passive, meaning they do not change the data. They receive a single value on their input and duplicate the value to their many outputs. From the input layer, it duplicates each value and sent to all the hidden nodes.
- **Hidden Layer** The Hidden layers apply given transformations to the input values inside the network. In this, incoming arcs that go from other hidden nodes or from input nodes connected to each node. It connects with outgoing arcs to output nodes or to other hidden nodes. In hidden layer, the actual processing is done via a system of weighted ‘connections’. There may be one or more hidden layers. The values entering a hidden node multiplied by weights, a set of predetermined numbers stored in the program. The weighted inputs are then added to produce a single number.
- **Output layer** The hidden layers then link to an ‘output layer‘. Output layer receives connections from hidden layers or from input layer. It returns an output value that corresponds to the prediction of the response variable. In classification problems, there is usually only one output node. The active nodes of the output layer combine and change the data to produce the output values.

In [147]:
```
import seaborn as sns
import matplotlib.pyplot as plt
plt.figure(figsize=[10,5])
plt.scatter(range(len(train)),list(target.SalePrice.values))
plt.show()
plt.figure(figsize=[10,5])
sns.kdeplot(target.SalePrice, shade= True)
plt.show()
```





In the above graph we can see that the price range is in a normal distribution. If we provide tf.random.normal while initializing the weight it should be more helpful for training. And this initialization should provide better validation with low amount of epoches. In my kaggle score rmse 0.123 is found through random normal while uniform distribution provided rmse 0.127 score. Again Uniform distribution takes 3 times more epoches to reach rmse score 0.127. But for uniform distribution no improvement cant found after 16000 epoch and for normal distribution no improvement can't found after 6000 epoch.

Target

By observing the span of the data and the data distribution we can conclude that logistic regression should perform well for this kind of problem. So we can safely say that starting with single neuron in a single hidden layer should perform well and we should look for simpler solution. Again from theoretical perspective single neurone and single layer ANN is nothing but a logistic regression and after adding layers and neurons we can regularize them so that they behave more like a logistic regression model and then we can tune parameter such a way so that it can handle little bit more complexity than a logistic regression. Finally my target is to make sure that it performs well as a logistic regression model and then improve it with more neuron/layers and proper tuning of parameters.

In [148]:
```
# log_df = pd.DataFrame(columns=['learning_rate', 'num_steps', 'beta1','beta2','beta3', 'hidden_1' , 'hidden_2', 'hidden_3','input
_dim' , 'test_rmse_score', 'test_r2_score'])
# log_df.to_csv("diffrent_training_results.csv", index=False)
```

Ann parameters

Variables

A brief explanation of the variables used is given below. Some terminologies are explained in more detail when their usage comes up.

learning_rate: On a intuition level, learning rate means how fast the network will learn something new and discard the old one. On a technical level, learning rate determines how fast the **'weights'** will be updated. Learning rate should be high enough so that it won't take too long to converge, and it should be low enough so that it is able to find the minima.

epoch: The number of times the model will be trained. After each run, the **'weights'** will be updated by the means of **'optimizer'**

beta1/2/3 : These variables control how much penalty to add to the model's loss function.

hidden_1/2/3 = Determines how many neurons a layer has. The number after the **'hidden_'** part denotes the layer number. i.e. 2 means second hidden layer

input_dim: Determines the shape of the input matrix. The input size is the same as the number of features the dataset has.

output_dim: Determines the shape of the final output. As this is a regression problem the ouput is of size **one**.

X_tf/y_tf: These two are tensorflow placeholder variables. They take input during the training period.

loss for loss function I have used mean squared error.

The following ANN is build with 3 hidden layers. Output dimention is 1 because its a regration problem.

In [149]:
```
tf.reset_default_graph()
learning_rate = 0.1
num_steps = 8000
#for regularize weight matrix
beta1 = 0.1
beta2 = 0.0
beta3 = 0.0
beta4 = None

hidden_1 = 16
hidden_2 = 8
hidden_3 = 4
hidden_4 = None



# minimum_validation_loss is to control model saving locally
minimum_validation_loss = 0.0190000

input_dim = X_train.shape[1]  # Number of features
output_dim = 1              # Because it is a regression problem
#tf graph input
X_tf = tf.placeholder("float" )
y_tf = tf.placeholder("float" )
```

Weight and Bias

A weight decides how much influence the input will have on the output. A weight represent the strength of the connection between units. When a value arrives at a neuron, the value gets multiplied by a weight value.

Bias is an extra input to neurons and has it’s own connection weight. But a bias node is not connected to any node in the previous layer, only connected to the next layer. This makes sure that even when all the inputs are none (all 0’s) there’s gonna be an activation in the neuron.

Here I have initialized the "weight" and "bias" variables as "random normal", which takes some random values from a normal distribution to use. Now there is also the option to set them all to "zero". But there is a problem to that. If all of the weights are the same, they will all have the same error and the model will not learn anything - there is no source of asymmetry between the neurons.That's why the better method is to keep the weights very close to zero but make them different by initializing them to small, non-zero numbers. With default parameters, "random normal" chooses values from a nomal distribution whose mean is 0 (zero) and has a standard deviation of 1 (one).

In [150]:
```
weights = {
    'w1': tf.Variable(tf.random_normal([input_dim, hidden_1])),
    'w2': tf.Variable(tf.random_normal([hidden_1, hidden_2])),
    'w3': tf.Variable(tf.random_normal([hidden_2, hidden_3])),
    'out': tf.Variable(tf.random_normal([hidden_3, output_dim]))
}
biases = {
    'b1': tf.Variable(tf.random_normal([hidden_1])),
    'b2': tf.Variable(tf.random_normal([hidden_2])),
    'b3': tf.Variable(tf.random_normal([hidden_3])),
    'out': tf.Variable(tf.random_normal([output_dim]))
}
```

```
WARNING:tensorflow:From /home/navid/anaconda3/envs/tf/lib/python3.6/site-packages/tensorflow/python/framework/op_def_library.py:26
3: colocate_with (from tensorflow.python.framework.ops) is deprecated and will be removed in a future version.
Instructions for updating:
Colocations handled automatically by placer.
```

Model

The following block of code is what the actual ANN model looks like. Each layer, a matrix multiplication happens and then the layer is activated by a activation function. The final output layer does not have any activation function because we are performing a regression a here.

Here, the activation function is our main concern. Currently the most popular types of Activation functions are as follows:

- Sigmoid
- Tanh - Hyperbolic tangent
- ReLu - Rectified linear units

"Sigmoid" activation function is mathematically represented by this equation,f(x) = 1 / 1 + exp(-x) . Its output range is between 0 to 1 and it has an S-shaped curve. It is easy to understand and apply but it has "vanishing gradient" problem as well as being slow to converg. So, I have avoided using it.

"**Tanh**" activation function is mathematically represented by this equation,f(x) = 1- exp(-2x) / 1 + exp(-2x). It's output range is in between -1 to 1 i.e -1 < output < 1 . As such optimization is easier in this method but still it suffers from Vanishing gradient problem.

"**ReLu**" is a very popular currently due to its simplicity and ease of use. Mathematically, ReLu can be defined as follows-

R(x) = max(0,x) i.e if x < 0 , R(x) = 0 and if x >= 0 , R(x) = x.

From the mathamatical function it can be seen that it is very simple and efficinent. It also avoids and rectifies vanishing gradient problem . It is also relatively easier to optimize.

In the dataset Sales price are non negative number so our model is expected to return positive values so as a activation function I have used relu as it gives positive values. Again relu is easy to optimize because they are similar to linear units. The only difference is that a rectified linear unit outputs zero across half its domain. Thus derivatives through a rectified linear unit remain large whenever the unit is activate. The gradients are not only large but also consistent.

In [151]:
```
def ann_model(X_input):
  # Hidden layers
    layer_1 = tf.add(tf.matmul(X_input, weights['w1']), biases['b1'])
    layer_1 = tf.nn.relu(layer_1)


    layer_2 = tf.add(tf.matmul(layer_1, weights['w2']), biases['b2'])
    layer_2 = tf.nn.relu(layer_2)

    layer_3 = tf.add(tf.matmul(layer_2, weights['w3']), biases['b3'])
    layer_3 = tf.nn.relu(layer_3)


    # Output layer
    layer_out = tf.matmul(layer_3, weights['out']) + biases['out']

    return layer_out
```

For optimization I have used Adam optimizer. Adam derives from phrase “adaptive moments”. Its a varient of RMSProp. I have used adam instead of RMSProp for couple of reasons. First, in Adam, momentum is incorporated directly as an estimate of the first-order moment (with exponential weighting) of the gradient. The most straightforward way to add momentum to RMSProp is to apply momentum to the rescaled gradients. The use of momentum in combination with rescaling does not have a clear theoretical motivation. Second, Adam includes bias corrections to the estimates of both the first-order moments (the momentumterm) and the (uncentered) second-order moments to account for their initializationat the origin. RMSProp also incorporates an estimate of the (uncentered) second-order moment; however, it lacks the correction factor. Thus,unlike in Adam, the RMSProp second-order moment estimate may have high bias early in training. Adam is generally regarded as being fairly robust to the choice of hyperparameters, though the learning rate sometimes needs to be changed from the suggested default. Usually default rate is .001 but for our case I have used 0.1 as it gives better optimization results.

Following segment is actually initializing different parameters. From the dataset we can see that the estimation of sale price is a regression problem and neural network used here was overfitting most of the time due to higher variance. So for making it simpler I have penalized weight matrix of hidden layers with l2 regularization. Again I have found that single hidden layer with single neuron performs well and that means the prediction model don't need to be too complex. Thus I became ensured that regularization is going to improve performance.

In [152]:
```
# Model Construct
model = ann_model(X_tf)

# Mean Squared Error function
# loss = tf.reduce_mean(tf.square(y_tf - model))
loss = tf.losses.mean_squared_error(y_tf , model , reduction=tf.losses.Reduction.SUM_BY_NONZERO_WEIGHTS)

# loss = tf.square(y_tf - model)
regularizer_1 = tf.nn.l2_loss(weights['w1'])
regularizer_2 = tf.nn.l2_loss(weights['w2'])
regularizer_3 = tf.nn.l2_loss(weights['w3'])
loss = tf.reduce_mean(loss + beta1*regularizer_1 + beta2*regularizer_2 + beta3*regularizer_3)
# loss = loss + beta1*regularizer_1 + beta2*regularizer_2 + beta3*regularizer_3

# Adam optimizer will update weights and biases after each step
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(loss)


# Initialize variables
init = tf.global_variables_initializer()

# Add ops to save and restore all the variables.
saver = tf.train.Saver()
```

```
WARNING:tensorflow:From /home/navid/anaconda3/envs/tf/lib/python3.6/site-packages/tensorflow/python/ops/losses/losses_impl.py:667:
to_float (from tensorflow.python.ops.math_ops) is deprecated and will be removed in a future version.
Instructions for updating:
Use tf.cast instead.
WARNING:tensorflow:From /home/navid/anaconda3/envs/tf/lib/python3.6/site-packages/tensorflow/python/ops/math_ops.py:3066: to_int32
(from tensorflow.python.ops.math_ops) is deprecated and will be removed in a future version.
Instructions for updating:
Use tf.cast instead.
```

Training

The **training_block()** is the function where all the work finally happens. The constructed model gets the training data and the training process begins. In each epoch the code is calculating the loss function and trying to minimize that value. Training loss and validation loss of each epoch gets stored in **train_loss** and **val_loss** respectively. After each 50 epochs the current loss values are added to two lists **train_LC** and **val_LC**, which is used to plot the learning curve after the training is finished. Also after each 500 epochs, I have printed the **Training loss and validation loss.**

During the training phase, I have run a shuffle function on the input data. This is so that when the data is input into the model, there are some variation to the serial the data gets inside the model. The reason why i have done it is so that it can have the effect of training on mini batches.

In [153]:
```
train_LC = []
val_LC = []
# session_var = None
```

Above train LC and val Lc variable keeps track of the learning rate so that learning curve can be drwan. In the following training block I have shuffled the training data in each epoch. This helps to reduce the loss difference of the validation and training. Thus it reduces the chance for over-fitting and under-fitting.

In [154]:
```
def training_block(X_train,y_train, X_val,y_val):
    #reseting variables
    session_var = None
    save_path = None
    with tf.Session() as sess:

        #running initializer
        sess.run(init)

#         minimum_validation_loss = 0.0190000

        global minimum_validation_loss
        for i in range(num_steps):
            if submit :
                X_train , y_train = shuffle(train_processed,target )
            else:
                X_train,y_train = shuffle(X_train,y_train )

            sess.run(optimizer, feed_dict={X_tf:X_train, y_tf:y_train})
            train_loss = sess.run(loss, feed_dict={X_tf:X_train, y_tf:y_train})
            val_loss = sess.run(loss, feed_dict={X_tf:X_val, y_tf:y_val})
            if submit :
                new_minimum_validation_loss = np.min(train_loss)
            else:
                new_minimum_validation_loss = np.min(val_loss)

#             if (i+1)%50 == 0:
            train_LC.append(train_loss)
            val_LC.append(val_loss)

            if (i+1)%500 == 0:
                print("epoch no : ",i+1, "  training loss: ",train_loss, "  validation loss: ", val_loss, "    minimum_validation_
loss" , minimum_validation_loss)

            if new_minimum_validation_loss  < minimum_validation_loss :
                minimum_validation_loss = new_minimum_validation_loss
#                 global session_var
#                 session_var = sess
#                 Save the variables to disk.
                save_path = saver.save(sess, "model/model.ckpt")


        if bool(save_path):
            sess.close()
            print("Model saved in path: %s" % save_path)



training_block(X_train,y_train, X_val,y_val)
```

```
epoch no :  500   training loss:  0.0154222   validation loss:  0.01753993     minimum_validation_loss 0.0153078
epoch no :  1000   training loss:  0.044814564   validation loss:  0.083696276     minimum_validation_loss 0.013667234
epoch no :  1500   training loss:  0.02136956   validation loss:  0.023081107     minimum_validation_loss 0.013120597
epoch no :  2000   training loss:  0.01474503   validation loss:  0.01468334     minimum_validation_loss 0.013120597
epoch no :  2500   training loss:  0.014489401   validation loss:  0.014471456     minimum_validation_loss 0.013120597
epoch no :  3000   training loss:  0.014190603   validation loss:  0.014224969     minimum_validation_loss 0.013120597
epoch no :  3500   training loss:  0.013883083   validation loss:  0.013974995     minimum_validation_loss 0.013120597
epoch no :  4000   training loss:  0.06976974   validation loss:  0.07462383     minimum_validation_loss 0.013120597
epoch no :  4500   training loss:  0.014431743   validation loss:  0.014358748     minimum_validation_loss 0.013120597
epoch no :  5000   training loss:  0.014715004   validation loss:  0.01466368     minimum_validation_loss 0.013120597
epoch no :  5500   training loss:  0.023093486   validation loss:  0.022327803     minimum_validation_loss 0.013120597
epoch no :  6000   training loss:  0.015900424   validation loss:  0.01563737     minimum_validation_loss 0.013120597
epoch no :  6500   training loss:  0.020474296   validation loss:  0.020973198     minimum_validation_loss 0.013120597
epoch no :  7000   training loss:  0.01923947   validation loss:  0.018714586     minimum_validation_loss 0.013120597
epoch no :  7500   training loss:  0.01636867   validation loss:  0.015925562     minimum_validation_loss 0.013120597
epoch no :  8000   training loss:  0.022346398   validation loss:  0.022731695     minimum_validation_loss 0.013120597
Model saved in path: model/model.ckpt
```

Grid search on epoch:

In the above block I have saved the model when validation loss is lowest. To do that I have kept another parameter called minimum_validation_loss. When validation loss reach lower I save the model, update minimum_validation_loss and continue running it. If it finds another lower validation loss it saves the model again and update minimum_validation_loss. Thus when I get the lowest validation loss my model saves again and that is the most optimum result. But when I run using all the data to predict kaggle test dataset then I use training loss to do the same.

As I mentioned earlier the epoch to reach the best validation accuracy is not fixed. Rather we can find it in 3 different range of epoch. The reason behind this is mostly because of random initializing of the weight and if we have fixed the seed value then it might change into only one single epoch range. But doing so we loose chance to improve our model further. Again if we want to ensemble different ANN model it woun't help when we use same seed and state. I have tried 1000+ parameters and combination from the start and used graph to visualize how to improve that but with grid search I might not get the exact idea why certain things provide good results or not and looking into every search result and graph is also too much so applying on the epoch seems to me more reasonable solution because the epoch for best validation result will be different in every run.

Trick

I have shuffled the data in every epoch and this trick improved the validation accuracy. On the other hand I did't use batch because according to my previous experience this kind of logistic regression problem works better when its given as a whole set rather than batch or mini-batch. But if its overfitting then passing the data in a batch / mini-batch would perform better as it helps to generalize more. We can say its more like a dropout effect. And I have tried to do dropout to reduce distance of training and validation accuracy but that didn't worked well.

In [155]:
```
def Prediction_block(X_val):
    with tf.Session() as sess:
        try:
            # Restore variables from disk.
            saver.restore(sess, "model/model.ckpt")
            print("Model restored.")
        except:
            print("------------ available checkpoint is for different model --------------")
            return
        # Check the values of the variables
        pred =  sess.run(model, feed_dict={X_tf: X_val})
        prediction = pred.squeeze()
        sess.close()
        return prediction
    #     print(np.exp(prediction))

prediction = Prediction_block(X_val)

pred_str = 'ANN_base_lr'+str(learning_rate)+'_beta'+str(beta1)+'-'+str(beta2)+'-'+str(beta3)+'-'+str(beta4)+'_hidden'+str(hidden_1
)+'-'+str(hidden_2)+'-'+str(hidden_3)+'-'+str(hidden_4)
prediction_dict[pred_str] = prediction

if submit:
    submit_prediction = Prediction_block(test_processed)
    submit_prediction_dict[pred_str] = submit_prediction
```

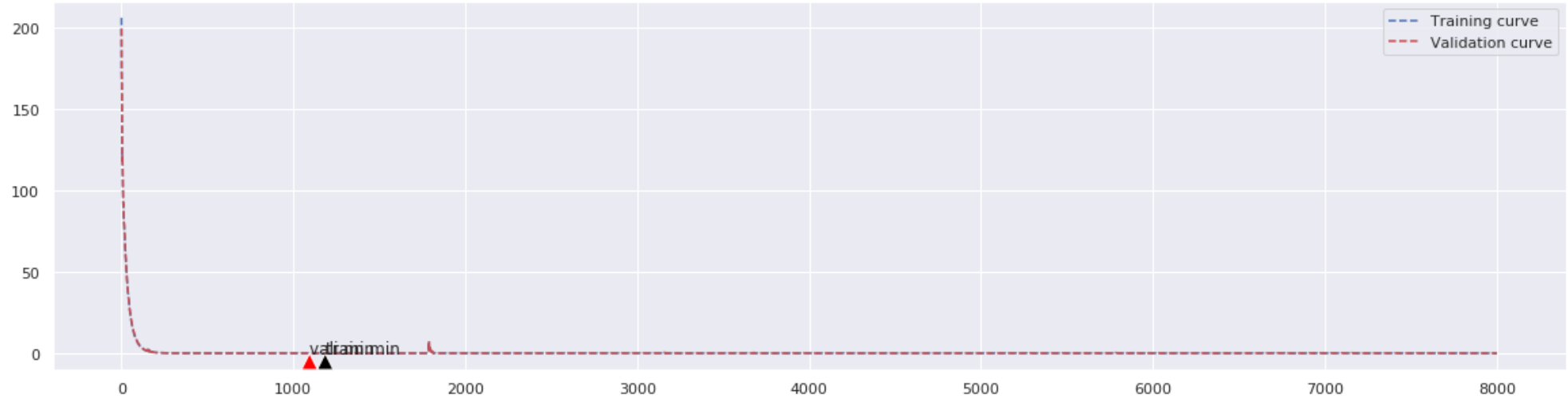
```
WARNING:tensorflow:From /home/navid/anaconda3/envs/tf/lib/python3.6/site-packages/tensorflow/python/training/saver.py:1266: checkp
oint_exists (from tensorflow.python.training.checkpoint_management) is deprecated and will be removed in a future version.
Instructions for updating:
Use standard file APIs to check for files with this prefix.
INFO:tensorflow:Restoring parameters from model/model.ckpt
Model restored.
```

Learning curve

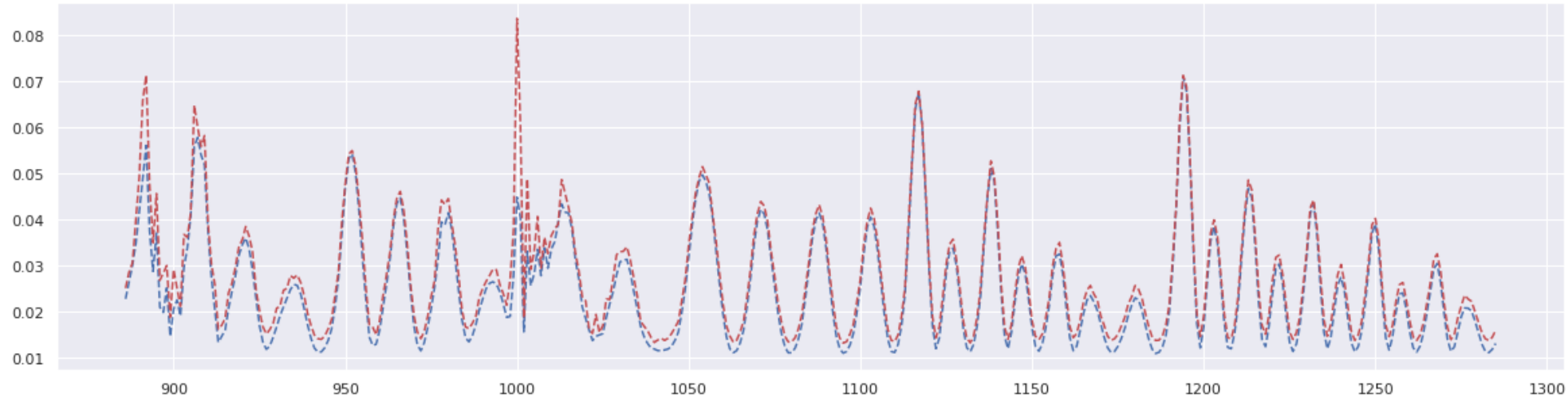
Following variables are only used to zoom into the graph

- start_observation_flag = starts point to zoom in
- end_observation_flag = end point to zoom in

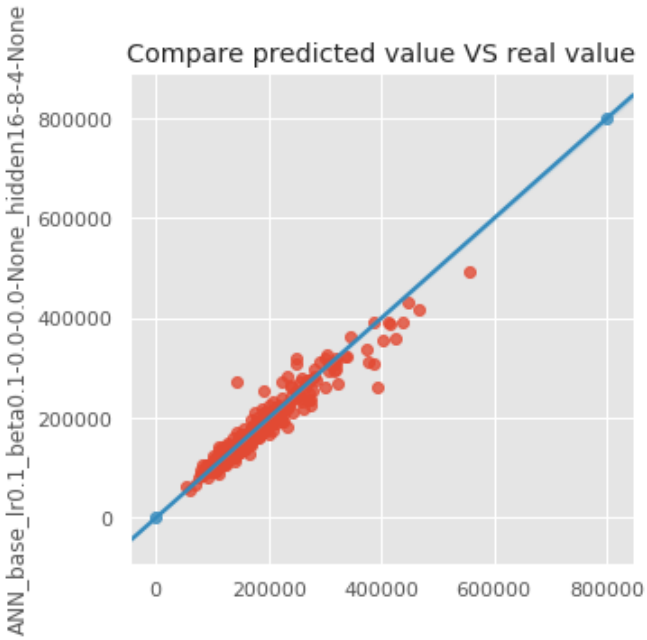
In [156]:
```
def learning_curve(start_observation_flag,end_observation_flag):
    xdata = list(range(1,len(train_LC)+1))
    minimum = min(train_LC)

    plt.figure(figsize=[20,5])
    plt.plot(xdata, train_LC, 'b--', label='Training curve')
    plt.annotate('train min', xy=(xdata[train_LC.index(minimum)], minimum),
                 arrowprops=dict(facecolor='black', shrink=0.05))

    minimum = min(val_LC)
    plt.plot(xdata, val_LC, 'r--' , label='Validation curve')
    plt.annotate('vali min', xy=(xdata[val_LC.index(minimum)], minimum),
                 arrowprops=dict(facecolor='red', shrink=0.05))
    plt.legend()
    plt.show()

    print("If we zoom into the curve we would have seen the following")

    plt.figure(figsize=[20,5])
    plt.plot(xdata[start_observation_flag:end_observation_flag], train_LC[start_observation_flag:end_observation_flag], 'b--')
    plt.plot(xdata[start_observation_flag:end_observation_flag], val_LC[start_observation_flag:end_observation_flag],'r--')

    plt.show()

#Following variables are only used to zoom into the graph
start_observation_flag = train_LC.index( min(train_LC)) - 300
end_observation_flag =  train_LC.index( min(train_LC)) + 100
learning_curve(start_observation_flag,end_observation_flag)
```



If we zoom into the curve we would have seen the following



In [205]:
```
def plot_prediction(y_val, prediction_val):
    plt.figure(figsize=[5,5])
    plt.title('Compare predicted value VS real value')
    sns.regplot(x= np.exp(y_val), y = np.exp(prediction_val), fit_reg=False)
    sns.regplot(x=np.array([10,800000]), y=np.array([10,800000]),fit_reg=True)
    plt.show()


plot_prediction( y_val, pred_df['ANN_base_lr0.1_beta0.1-0.0-0.0-None_hidden16-8-4-None'])
```



Both of the curve actually seems to be on top of each other.The reason is:

- I have applied log transformation on the SalePrice and I have also transformed all my numerical data thats why the difference between the training loss and validation loss seems to be very small and very stable.
- For loss function I have used Mean Squared Error (MSE). For reducing MSE I have used SUM_BY_NONZERO_WEIGHTS which divided scalar sum by number of non-zero weights. MSE calculates squared error for all the data and then calculate the mean. Now, all my SalePrice is very small due to normalization (between 10 to 13.5). Where mean of saleprice is 12.02 . Suppose in nth epoch if
  - for training loss
    - a saleprice is 11.5 and prediction is 12 Squred error .25
    - another saleprice is 13 and prediction is 12 Squred error 1
    - another saleprice is 12.5 and prediction is 12 Squred error .25
    - another saleprice is 11.5 and prediction is 12 Squred error .25
    - another saleprice is 10.3 and prediction is 12 Squred error 1.7

    MSE = (.25+1+.25+.25+1.7)/5 = .69
  - for validation loss
    - another saleprice is 12.9 and prediction is 12 Squred error .81
    - another saleprice is 13.3 and prediction is 12 Squred error 1.69
    - another saleprice is 10.8 and prediction is 12 Squred error 1.44
    - another saleprice is 11.3 and prediction is 12 Squred error .49
    - another saleprice is 11.8 and prediction is 12 Squred error .04

    MSE = (.81+1.69+1.44+.49+.04)/5 = .894

    Difference between validation loss and training loss is .204

    Usually in regression problem neural network stats to predicts the average value within 5-20 epoch so very quickly the difference between val_loss and training_loss gets much lower. In our dummy example difference is already .204 and if its epoch no is 10, by the time it reaches to 500 epoch the difference could go as low as 10^-4.

Acuracy Score

In [157]:
```
def accuracy(y_val,prediction):
    test_rmse_score = rmse(y_val, prediction)
    test_r2_score =  r2_score(np.array(y_val),prediction)
    return test_rmse_score, test_r2_score

test_rmse_score, test_r2_score = accuracy(y_val,prediction)

print('ann root mean absolute error: ', test_rmse_score)
print('accuracy score: ', test_r2_score  )
```

```
ann root mean absolute error:  0.10422349247961371
accuracy score:  0.9281334877446001
```

kaggle rmse:

In kaggle ranking the above ANN model provides the best rmse score and the score is 0.11912

Description on Learning curve and Accuracy:

We can observe where overfitting occurs. Overfitting actually occurs if the training loss goes under the validation loss even though the validation is still dropping. It is the sign that network is learning the patterns in the train set that are not applicable in the validation done. In a short note we can say::

Overfitting : training loss << validation loss

Underfitting : training loss >> validation loss

Just right : training loss ~ validation loss

According to this theory our both learning curve is exactly top of one another so in our case validation loss and training loss is almost same so we can say that our model is doing just the right thing. Again In validation score .1054 is impressive compared to other models.

Save score

In [158]:
```
if save_score:
    log_df = pd.read_csv("diffrent_training_results.csv")
    log_df = log_df.append({'learning_rate' : learning_rate, 'num_steps' : num_steps, 'beta1' : beta1, 'beta2' : beta2, 'beta3' :
beta3, 'beta4' : beta4, 'hidden_1' : hidden_1 , 'hidden_2' : hidden_2, 'hidden_3' : hidden_3, 'hidden_4' : hidden_4, 'input_dim' :
input_dim , 'test_rmse_score' : test_rmse_score , 'test_r2_score' : test_r2_score}, ignore_index=True)
    log_df.to_csv("diffrent_training_results.csv", encoding='utf-8',index=False)
```

Cross validation

When we perform a random train-test split of our data, we assume that our examples are independent. That means that by knowing/seeing some instance will not help us understand other instances. However, that’s not always the case. So to make sure if the Data is actually independent, to get more metrics and to use fine tuning my parameters on whole dataset I am performing cross validation.

In [180]:

```
from sklearn.model_selection import KFold
from sklearn.model_selection import RepeatedKFold
kf = KFold(n_splits=10, shuffle=True)

kf_rmse_list = []
kf_r2_list = []

# train_processed['SalePrice'] = target.values
for train_index, test_index in kf.split(train_processed):
    X_train, X_val = train_processed.iloc[train_index] , train_processed.iloc[test_index]
    y_train, y_val = target.iloc[train_index], target.iloc[test_index]

    training_block(X_train,y_train, X_val,y_val)
    prediction = Prediction_block(X_val)
    test_rmse_score, test_r2_score = accuracy(y_val, prediction)

    kf_rmse_list.append(test_rmse_score)
    kf_r2_list.append(test_r2_score)

    print("r2 list print", kf_r2_list)
    print('rmse list print',kf_rmse_list)

print("r2 mean print", np.mean(kf_r2_list))
print('rmse mean print', np.mean(kf_rmse_list))
```

```
epoch no :  500   training loss:  0.013619974   validation loss:  0.011485558     minimum_validation_loss 0.009141851
epoch no :  1000   training loss:  0.012036915   validation loss:  0.009576037     minimum_validation_loss 0.009141851
epoch no :  1500   training loss:  0.012212438   validation loss:  0.010224601     minimum_validation_loss 0.009141851
epoch no :  2000   training loss:  0.0175557   validation loss:  0.014995297     minimum_validation_loss 0.009141851
epoch no :  2500   training loss:  0.0120672025   validation loss:  0.010273805     minimum_validation_loss 0.009141851
epoch no :  3000   training loss:  0.011971448   validation loss:  0.009877532     minimum_validation_loss 0.009141851
epoch no :  3500   training loss:  0.012855139   validation loss:  0.010995634     minimum_validation_loss 0.009141851
epoch no :  4000   training loss:  0.012983421   validation loss:  0.010518662     minimum_validation_loss 0.009141851
epoch no :  4500   training loss:  0.018259497   validation loss:  0.017239176     minimum_validation_loss 0.009141851
epoch no :  5000   training loss:  0.015164167   validation loss:  0.012945443     minimum_validation_loss 0.009141851
epoch no :  5500   training loss:  0.014592094   validation loss:  0.012027358     minimum_validation_loss 0.009141851
epoch no :  6000   training loss:  0.02006973   validation loss:  0.01937655     minimum_validation_loss 0.009141851
epoch no :  6500   training loss:  0.020192016   validation loss:  0.017559156     minimum_validation_loss 0.009141851
epoch no :  7000   training loss:  0.015621353   validation loss:  0.013175331     minimum_validation_loss 0.009141851
epoch no :  7500   training loss:  0.023095477   validation loss:  0.022388548     minimum_validation_loss 0.009141851
epoch no :  8000   training loss:  0.02218112   validation loss:  0.021205228     minimum_validation_loss 0.009141851
INFO:tensorflow:Restoring parameters from model/model.ckpt
Model restored.
r2 list print [0.966034794679135]
rmse list print [0.07918232636080008]
epoch no :  500   training loss:  8.106482   validation loss:  7.971449     minimum_validation_loss 0.009141851
epoch no :  1000   training loss:  1.7433393   validation loss:  1.710667     minimum_validation_loss 0.009141851
epoch no :  1500   training loss:  0.61207014   validation loss:  0.57976574     minimum_validation_loss 0.009141851
epoch no :  2000   training loss:  0.29504254   validation loss:  0.26273856     minimum_validation_loss 0.009141851
epoch no :  2500   training loss:  0.19452232   validation loss:  0.16221894     minimum_validation_loss 0.009141851
epoch no :  3000   training loss:  0.16488338   validation loss:  0.13258037     minimum_validation_loss 0.009141851
epoch no :  3500   training loss:  0.15798438   validation loss:  0.12568174     minimum_validation_loss 0.009141851
epoch no :  4000   training loss:  0.15688232   validation loss:  0.12457993     minimum_validation_loss 0.009141851
epoch no :  4500   training loss:  0.15677679   validation loss:  0.12447465     minimum_validation_loss 0.009141851
epoch no :  5000   training loss:  0.1567716   validation loss:  0.12446962     minimum_validation_loss 0.009141851
epoch no :  5500   training loss:  0.15677151   validation loss:  0.12446962     minimum_validation_loss 0.009141851
epoch no :  6000   training loss:  0.1567715   validation loss:  0.12446976     minimum_validation_loss 0.009141851
epoch no :  6500   training loss:  0.1567715   validation loss:  0.12446976     minimum_validation_loss 0.009141851
epoch no :  7000   training loss:  0.1567715   validation loss:  0.12446989     minimum_validation_loss 0.009141851
epoch no :  7500   training loss:  0.15677151   validation loss:  0.12446989     minimum_validation_loss 0.009141851
epoch no :  8000   training loss:  0.1567715   validation loss:  0.12446989     minimum_validation_loss 0.009141851
INFO:tensorflow:Restoring parameters from model/model.ckpt
Model restored.
r2 list print [0.966034794679135, 0.9147768660572881]
rmse list print [0.07918232636080008, 0.1011852163700275]
epoch no :  500   training loss:  0.96950114   validation loss:  0.99083734     minimum_validation_loss 0.009141851
epoch no :  1000   training loss:  0.17070965   validation loss:  0.16137826     minimum_validation_loss 0.009141851
epoch no :  1500   training loss:  0.1568672   validation loss:  0.14752576     minimum_validation_loss 0.009141851
epoch no :  2000   training loss:  0.15677167   validation loss:  0.1474298     minimum_validation_loss 0.009141851
epoch no :  2500   training loss:  0.1567715   validation loss:  0.1474292     minimum_validation_loss 0.009141851
epoch no :  3000   training loss:  0.1567715   validation loss:  0.14742897     minimum_validation_loss 0.009141851
epoch no :  3500   training loss:  0.15677151   validation loss:  0.14742875     minimum_validation_loss 0.009141851
epoch no :  4000   training loss:  0.1567715   validation loss:  0.14742853     minimum_validation_loss 0.009141851
epoch no :  4500   training loss:  0.1567715   validation loss:  0.14742841     minimum_validation_loss 0.009141851
epoch no :  5000   training loss:  0.1567715   validation loss:  0.1474283     minimum_validation_loss 0.009141851
epoch no :  5500   training loss:  0.1567715   validation loss:  0.14742818     minimum_validation_loss 0.009141851
```

```
epoch no :  6000   training loss:  0.15677153   validation loss:  0.14742818     minimum_validation_loss 0.009141851
epoch no :  6500   training loss:  0.1567715   validation loss:  0.14742808     minimum_validation_loss 0.009141851
epoch no :  7000   training loss:  0.1567715   validation loss:  0.14742808     minimum_validation_loss 0.009141851
epoch no :  7500   training loss:  0.1567715   validation loss:  0.14742808     minimum_validation_loss 0.009141851
epoch no :  8000   training loss:  0.1567715   validation loss:  0.14742795     minimum_validation_loss 0.009141851
INFO:tensorflow:Restoring parameters from model/model.ckpt
Model restored.
r2 list print [0.966034794679135, 0.9147768660572881, 0.9537090047645579]
rmse list print [0.07918232636080008, 0.1011852163700275, 0.08164930890918835]
epoch no :  500   training loss:  0.33438882   validation loss:  0.32751098     minimum_validation_loss 0.009141851
epoch no :  1000   training loss:  0.030355968   validation loss:  0.026787056     minimum_validation_loss 0.009141851
epoch no :  1500   training loss:  0.013811676   validation loss:  0.01231014     minimum_validation_loss 0.009141851
epoch no :  2000   training loss:  0.026015006   validation loss:  0.023583803     minimum_validation_loss 0.009141851
epoch no :  2500   training loss:  0.011827763   validation loss:  0.01097246     minimum_validation_loss 0.009141851
epoch no :  3000   training loss:  0.020729046   validation loss:  0.02133848     minimum_validation_loss 0.009141851
epoch no :  3500   training loss:  0.014867802   validation loss:  0.014430715     minimum_validation_loss 0.009141851
epoch no :  4000   training loss:  0.013321041   validation loss:  0.013425821     minimum_validation_loss 0.009141851
epoch no :  4500   training loss:  0.012930628   validation loss:  0.012989472     minimum_validation_loss 0.009141851
epoch no :  5000   training loss:  0.0980646   validation loss:  0.09261147     minimum_validation_loss 0.009141851
epoch no :  5500   training loss:  0.012606918   validation loss:  0.012399212     minimum_validation_loss 0.009141851
epoch no :  6000   training loss:  0.012885423   validation loss:  0.012565173     minimum_validation_loss 0.009141851
epoch no :  6500   training loss:  0.0127938995   validation loss:  0.012595723     minimum_validation_loss 0.009141851
epoch no :  7000   training loss:  0.013260676   validation loss:  0.013041337     minimum_validation_loss 0.009141851
epoch no :  7500   training loss:  0.013235789   validation loss:  0.013062529     minimum_validation_loss 0.009141851
epoch no :  8000   training loss:  0.051659744   validation loss:  0.049887307     minimum_validation_loss 0.009141851
INFO:tensorflow:Restoring parameters from model/model.ckpt
Model restored.
r2 list print [0.966034794679135, 0.9147768660572881, 0.9537090047645579, 0.9427842533694809]
rmse list print [0.07918232636080008, 0.1011852163700275, 0.08164930890918835, 0.08959322377764997]
epoch no :  500   training loss:  51.094097   validation loss:  51.08056     minimum_validation_loss 0.009141851
epoch no :  1000   training loss:  24.44586   validation loss:  24.435799     minimum_validation_loss 0.009141851
epoch no :  1500   training loss:  12.07077   validation loss:  12.064256     minimum_validation_loss 0.009141851
epoch no :  2000   training loss:  6.1601996   validation loss:  6.156398     minimum_validation_loss 0.009141851
epoch no :  2500   training loss:  3.2237465   validation loss:  3.2210124     minimum_validation_loss 0.009141851
epoch no :  3000   training loss:  1.7077777   validation loss:  1.706599     minimum_validation_loss 0.009141851
epoch no :  3500   training loss:  0.89746666   validation loss:  0.89750814     minimum_validation_loss 0.009141851
epoch no :  4000   training loss:  0.4566344   validation loss:  0.45683303     minimum_validation_loss 0.009141851
epoch no :  4500   training loss:  0.21879756   validation loss:  0.21893464     minimum_validation_loss 0.009141851
epoch no :  5000   training loss:  0.09778543   validation loss:  0.09830833     minimum_validation_loss 0.009141851
epoch no :  5500   training loss:  0.042145047   validation loss:  0.04107195     minimum_validation_loss 0.009141851
epoch no :  6000   training loss:  0.020071626   validation loss:  0.018836489     minimum_validation_loss 0.009141851
epoch no :  6500   training loss:  0.019281756   validation loss:  0.021210428     minimum_validation_loss 0.009141851
epoch no :  7000   training loss:  0.010780502   validation loss:  0.011776745     minimum_validation_loss 0.009141851
epoch no :  7500   training loss:  0.012393176   validation loss:  0.011037876     minimum_validation_loss 0.009141851
epoch no :  8000   training loss:  0.015808862   validation loss:  0.015582579     minimum_validation_loss 0.009141851
INFO:tensorflow:Restoring parameters from model/model.ckpt
Model restored.
r2 list print [0.966034794679135, 0.9147768660572881, 0.9537090047645579, 0.9427842533694809, 0.9472118138230641]
rmse list print [0.07918232636080008, 0.1011852163700275, 0.08164930890918835, 0.08959322377764997, 0.08633177828950171]
epoch no :  500   training loss:  42.113926   validation loss:  42.221367     minimum_validation_loss 0.009141851
epoch no :  1000   training loss:  14.996103   validation loss:  14.996328     minimum_validation_loss 0.009141851
```

```
epoch no :  1500   training loss:  6.2828336   validation loss:  6.285405     minimum_validation_loss 0.009141851
epoch no :  2000   training loss:  2.66433   validation loss:  2.665118     minimum_validation_loss 0.009141851
epoch no :  2500   training loss:  1.1055615   validation loss:  1.1072863     minimum_validation_loss 0.009141851
epoch no :  3000   training loss:  0.43362805   validation loss:  0.43509766     minimum_validation_loss 0.009141851
epoch no :  3500   training loss:  0.16143301   validation loss:  0.16308072     minimum_validation_loss 0.009141851
epoch no :  4000   training loss:  0.058468625   validation loss:  0.06037449     minimum_validation_loss 0.009141851
epoch no :  4500   training loss:  0.023950096   validation loss:  0.02570058     minimum_validation_loss 0.009141851
epoch no :  5000   training loss:  0.014344843   validation loss:  0.015967095     minimum_validation_loss 0.009141851
epoch no :  5500   training loss:  0.0121255005   validation loss:  0.013745706     minimum_validation_loss 0.009141851
epoch no :  6000   training loss:  0.011622955   validation loss:  0.01326978     minimum_validation_loss 0.009141851
epoch no :  6500   training loss:  0.014613305   validation loss:  0.01599727     minimum_validation_loss 0.009141851
epoch no :  7000   training loss:  0.0122690005   validation loss:  0.014133809     minimum_validation_loss 0.009141851
epoch no :  7500   training loss:  0.012315406   validation loss:  0.013934883     minimum_validation_loss 0.009141851
epoch no :  8000   training loss:  0.011970366   validation loss:  0.01381484     minimum_validation_loss 0.009141851
INFO:tensorflow:Restoring parameters from model/model.ckpt
Model restored.
r2 list print [0.966034794679135, 0.9147768660572881, 0.9537090047645579, 0.9427842533694809, 0.9472118138230641, 0.94158670353409
46]
rmse list print [0.07918232636080008, 0.1011852163700275, 0.08164930890918835, 0.08959322377764997, 0.08633177828950171, 0.0994237
8517653855]
epoch no :  500   training loss:  0.016125906   validation loss:  0.015609428     minimum_validation_loss 0.009141851
epoch no :  1000   training loss:  0.012974413   validation loss:  0.013580626     minimum_validation_loss 0.009141851
epoch no :  1500   training loss:  0.01868954   validation loss:  0.019193621     minimum_validation_loss 0.009141851
epoch no :  2000   training loss:  0.011979641   validation loss:  0.011745248     minimum_validation_loss 0.009141851
epoch no :  2500   training loss:  0.012151112   validation loss:  0.011877935     minimum_validation_loss 0.009141851
epoch no :  3000   training loss:  0.021042723   validation loss:  0.018614208     minimum_validation_loss 0.009141851
epoch no :  3500   training loss:  0.012318071   validation loss:  0.012170627     minimum_validation_loss 0.009141851
epoch no :  4000   training loss:  0.012264705   validation loss:  0.012285118     minimum_validation_loss 0.009141851
epoch no :  4500   training loss:  0.012562231   validation loss:  0.013113246     minimum_validation_loss 0.009141851
epoch no :  5000   training loss:  0.0125289755   validation loss:  0.012318526     minimum_validation_loss 0.009141851
epoch no :  5500   training loss:  0.012627416   validation loss:  0.012321671     minimum_validation_loss 0.009141851
epoch no :  6000   training loss:  0.024291316   validation loss:  0.024922403     minimum_validation_loss 0.009141851
epoch no :  6500   training loss:  0.012883588   validation loss:  0.012206572     minimum_validation_loss 0.009141851
epoch no :  7000   training loss:  0.012608926   validation loss:  0.012091969     minimum_validation_loss 0.009141851
epoch no :  7500   training loss:  0.012720086   validation loss:  0.011958711     minimum_validation_loss 0.009141851
epoch no :  8000   training loss:  0.012726533   validation loss:  0.012433439     minimum_validation_loss 0.009141851
INFO:tensorflow:Restoring parameters from model/model.ckpt
Model restored.
r2 list print [0.966034794679135, 0.9147768660572881, 0.9537090047645579, 0.9427842533694809, 0.9472118138230641, 0.94158670353409
46, 0.9471667465299047]
rmse list print [0.07918232636080008, 0.1011852163700275, 0.08164930890918835, 0.08959322377764997, 0.08633177828950171, 0.0994237
8517653855, 0.09571442974640998]
epoch no :  500   training loss:  0.03303396   validation loss:  0.036419123     minimum_validation_loss 0.009141851
epoch no :  1000   training loss:  0.011525056   validation loss:  0.011891428     minimum_validation_loss 0.009141851
epoch no :  1500   training loss:  0.011402027   validation loss:  0.011600805     minimum_validation_loss 0.009141851
epoch no :  2000   training loss:  0.30597335   validation loss:  0.32709798     minimum_validation_loss 0.009141851
epoch no :  2500   training loss:  0.033091083   validation loss:  0.04023156     minimum_validation_loss 0.009141851
epoch no :  3000   training loss:  0.011892995   validation loss:  0.012220785     minimum_validation_loss 0.009141851
epoch no :  3500   training loss:  0.011965128   validation loss:  0.012517272     minimum_validation_loss 0.009141851
epoch no :  4000   training loss:  0.014375218   validation loss:  0.013712658     minimum_validation_loss 0.009141851
epoch no :  4500   training loss:  0.012115744   validation loss:  0.012684421     minimum_validation_loss 0.009141851
```

```
epoch no :  5000   training loss:  0.0122325625   validation loss:  0.01282599     minimum_validation_loss 0.009141851
epoch no :  5500   training loss:  0.015073455   validation loss:  0.014146421     minimum_validation_loss 0.009141851
epoch no :  6000   training loss:  0.012272882   validation loss:  0.012861219     minimum_validation_loss 0.009141851
epoch no :  6500   training loss:  0.012311014   validation loss:  0.012877526     minimum_validation_loss 0.009141851
epoch no :  7000   training loss:  0.012787523   validation loss:  0.012934981     minimum_validation_loss 0.009141851
epoch no :  7500   training loss:  0.012524765   validation loss:  0.013145186     minimum_validation_loss 0.009141851
epoch no :  8000   training loss:  0.014117582   validation loss:  0.013683337     minimum_validation_loss 0.009141851
INFO:tensorflow:Restoring parameters from model/model.ckpt
Model restored.
r2 list print [0.966034794679135, 0.9147768660572881, 0.9537090047645579, 0.9427842533694809, 0.9472118138230641, 0.94158670353409
46, 0.9471667465299047, 0.9546846375877138]
rmse list print [0.07918232636080008, 0.1011852163700275, 0.08164930890918835, 0.08959322377764997, 0.08633177828950171, 0.0994237
8517653855, 0.09571442974640998, 0.08534581770550757]
epoch no :  500   training loss:  0.012883391   validation loss:  0.01513976     minimum_validation_loss 0.009141851
epoch no :  1000   training loss:  0.013675831   validation loss:  0.016082974     minimum_validation_loss 0.009141851
epoch no :  1500   training loss:  0.014122789   validation loss:  0.016568564     minimum_validation_loss 0.009141851
epoch no :  2000   training loss:  0.04238956   validation loss:  0.048448827     minimum_validation_loss 0.009141851
epoch no :  2500   training loss:  0.03981378   validation loss:  0.04100083     minimum_validation_loss 0.009141851
epoch no :  3000   training loss:  0.015559063   validation loss:  0.0166934     minimum_validation_loss 0.009141851
epoch no :  3500   training loss:  0.013838102   validation loss:  0.016097013     minimum_validation_loss 0.009141851
epoch no :  4000   training loss:  0.013466299   validation loss:  0.015488665     minimum_validation_loss 0.009141851
epoch no :  4500   training loss:  0.01631156   validation loss:  0.017679408     minimum_validation_loss 0.009141851
epoch no :  5000   training loss:  0.013915047   validation loss:  0.014991083     minimum_validation_loss 0.009141851
epoch no :  5500   training loss:  0.060332302   validation loss:  0.06441878     minimum_validation_loss 0.009141851
epoch no :  6000   training loss:  0.01535064   validation loss:  0.016800927     minimum_validation_loss 0.009141851
epoch no :  6500   training loss:  0.014203868   validation loss:  0.015041616     minimum_validation_loss 0.009141851
epoch no :  7000   training loss:  0.013530094   validation loss:  0.014638897     minimum_validation_loss 0.009141851
epoch no :  7500   training loss:  0.012752774   validation loss:  0.013857335     minimum_validation_loss 0.009141851
epoch no :  8000   training loss:  0.021882962   validation loss:  0.02362502     minimum_validation_loss 0.009141851
INFO:tensorflow:Restoring parameters from model/model.ckpt
Model restored.
r2 list print [0.966034794679135, 0.9147768660572881, 0.9537090047645579, 0.9427842533694809, 0.9472118138230641, 0.94158670353409
46, 0.9471667465299047, 0.9546846375877138, 0.9554347882474149]
rmse list print [0.07918232636080008, 0.1011852163700275, 0.08164930890918835, 0.08959322377764997, 0.08633177828950171, 0.0994237
8517653855, 0.09571442974640998, 0.08534581770550757, 0.09000086249482751]
epoch no :  500   training loss:  0.012399824   validation loss:  0.012301879     minimum_validation_loss 0.009141851
epoch no :  1000   training loss:  0.018696032   validation loss:  0.018220013     minimum_validation_loss 0.009141851
epoch no :  1500   training loss:  0.013902478   validation loss:  0.013285283     minimum_validation_loss 0.009141851
epoch no :  2000   training loss:  0.012967689   validation loss:  0.012797864     minimum_validation_loss 0.009141851
epoch no :  2500   training loss:  0.012606605   validation loss:  0.012293708     minimum_validation_loss 0.009141851
epoch no :  3000   training loss:  0.012743736   validation loss:  0.012476213     minimum_validation_loss 0.009141851
epoch no :  3500   training loss:  0.05576414   validation loss:  0.055643518     minimum_validation_loss 0.009141851
epoch no :  4000   training loss:  0.013507971   validation loss:  0.013204421     minimum_validation_loss 0.009141851
epoch no :  4500   training loss:  0.012656722   validation loss:  0.012303873     minimum_validation_loss 0.009141851
epoch no :  5000   training loss:  0.02071242   validation loss:  0.020102337     minimum_validation_loss 0.009141851
epoch no :  5500   training loss:  0.037119415   validation loss:  0.03741987     minimum_validation_loss 0.009141851
epoch no :  6000   training loss:  0.016682925   validation loss:  0.016533     minimum_validation_loss 0.009141851
epoch no :  6500   training loss:  0.0155230565   validation loss:  0.015346852     minimum_validation_loss 0.009141851
epoch no :  7000   training loss:  0.01452904   validation loss:  0.01431609     minimum_validation_loss 0.009141851
epoch no :  7500   training loss:  0.01439276   validation loss:  0.01410942     minimum_validation_loss 0.009141851
epoch no :  8000   training loss:  0.021082401   validation loss:  0.02032368     minimum_validation_loss 0.009141851
```

```
INFO:tensorflow:Restoring parameters from model/model.ckpt
Model restored.
r2 list print [0.966034794679135, 0.9147768660572881, 0.9537090047645579, 0.9427842533694809, 0.9472118138230641, 0.94158670353409
46, 0.9471667465299047, 0.9546846375877138, 0.9554347882474149, 0.9480120117143849]
rmse list print [0.07918232636080008, 0.1011852163700275, 0.08164930890918835, 0.08959322377764997, 0.08633177828950171, 0.0994237
8517653855, 0.09571442974640998, 0.08534581770550757, 0.09000086249482751, 0.08550332181874735]
r2 mean print 0.9471401620307038
rmse mean print 0.08939300706491984
```

Observation

In the cross validation section we can see that 10 fold cross validation on our best ANN model provides similar rmse to 80-20 split rmse score. So we can relay on 80-20 split on this dataset. Thus we can say that the data in the dataset is independent.

Observing Few Other well performed ANN models

In this section We are observing the few other models and their learning curve. After that some of them will be used for Ensemble learning section for further improvement. In this model I have only changed the size of hidden layer, amount of neuron in each hidden layers , number of steps and learning rates. Rest of the part is same as the ANN described above.

ANN with 4 layers

Initialization of models

In [159]:
```
tf.reset_default_graph()
def weight_bais():
    global weights, biases
    weights = None
    biases = None

    weights = {
        'w1': tf.Variable(tf.random_normal([input_dim, hidden_1])),
        'w2': tf.Variable(tf.random_normal([hidden_1, hidden_2])),
        'w3': tf.Variable(tf.random_normal([hidden_2, hidden_3])),
        'w4': tf.Variable(tf.random_normal([hidden_3, hidden_4])),
        'out': tf.Variable(tf.random_normal([hidden_4, output_dim]))
    }
    biases = {
        'b1': tf.Variable(tf.random_normal([hidden_1])),
        'b2': tf.Variable(tf.random_normal([hidden_2])),
        'b3': tf.Variable(tf.random_normal([hidden_3])),
        'b4': tf.Variable(tf.random_normal([hidden_4])),
        'out': tf.Variable(tf.random_normal([output_dim]))
    }
```

In [160]:
```
def ann_model(X_input):
  # Hidden layers
    layer_1 = tf.add(tf.matmul(X_input, weights['w1']), biases['b1'])
    layer_1 = tf.nn.relu(layer_1)


    layer_2 = tf.add(tf.matmul(layer_1, weights['w2']), biases['b2'])
    layer_2 = tf.nn.relu(layer_2)

    layer_3 = tf.add(tf.matmul(layer_2, weights['w3']), biases['b3'])
    layer_3 = tf.nn.relu(layer_3)

    layer_4 = tf.add(tf.matmul(layer_3, weights['w4']), biases['b4'])
    layer_4 = tf.nn.relu(layer_4)

    # Output layer
#     layer_out = tf.add(tf.matmul(layer_4, weights['out']), biases['out'])
    layer_out = tf.matmul(layer_4, weights['out']) + biases['out']

    return layer_out
```

In [161]:
```
regularizer_4 = None
def miscellaneous_initialization():
    global model, loss , regularizer_1 , regularizer_2 ,regularizer_3, regularizer_4, optimizer , init , saver
    # Model Construct
    model = ann_model(X_tf)

    # Mean Squared Error loss function
    loss = tf.losses.mean_squared_error(y_tf , model , reduction=tf.losses.Reduction.SUM_BY_NONZERO_WEIGHTS)

    # loss = tf.square(y_tf - model)
    regularizer_1 = tf.nn.l2_loss(weights['w1'])
    regularizer_2 = tf.nn.l2_loss(weights['w2'])
    regularizer_3 = tf.nn.l2_loss(weights['w3'])
    regularizer_4 = tf.nn.l2_loss(weights['w4'])
    # loss = tf.reduce_mean(loss + beta1*regularizer_1 + beta2*regularizer_2 + beta3*regularizer_3)
    loss = tf.reduce_mean(loss + beta1*regularizer_1 + beta2*regularizer_2 + beta3*regularizer_3 + beta4*regularizer_4)

    # Adam optimizer will update weights and biases after each step
    optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(loss)


    # Initialize variables
    init = tf.global_variables_initializer()

    # Add ops to save and restore all the variables.
    saver = tf.train.Saver()
```

Training

ANN 1

In this section changed variables are

- learning rate = .01

layer name	Neuron	value of beta for l2 regularization
1st hidden layer	76 Neuron	.1
2nd hidden layer	48 Neuron	.05
3rd hidden layer	32 Neuron	0
4th hidden layer	16 Neuron	0

In [164]:

```
tf.reset_default_graph()
learning_rate = 0.1
num_steps = 25000
#for regularize weight matrix
beta1 = 0.1
beta2 = 0.05
beta3 = 0.0
beta4 = 0.0

hidden_1 = 76
hidden_2 = 48
hidden_3 = 32
hidden_4 = 16

minimum_validation_loss = .02101000

input_dim = X_train.shape[1]  # Number of features
output_dim = 1              # Because it is a regression problem

#tf graph input
X_tf = tf.placeholder("float" )
y_tf = tf.placeholder("float" )

weight_bais()
miscellaneous_initialization()
train_LC = []
val_LC = []
```

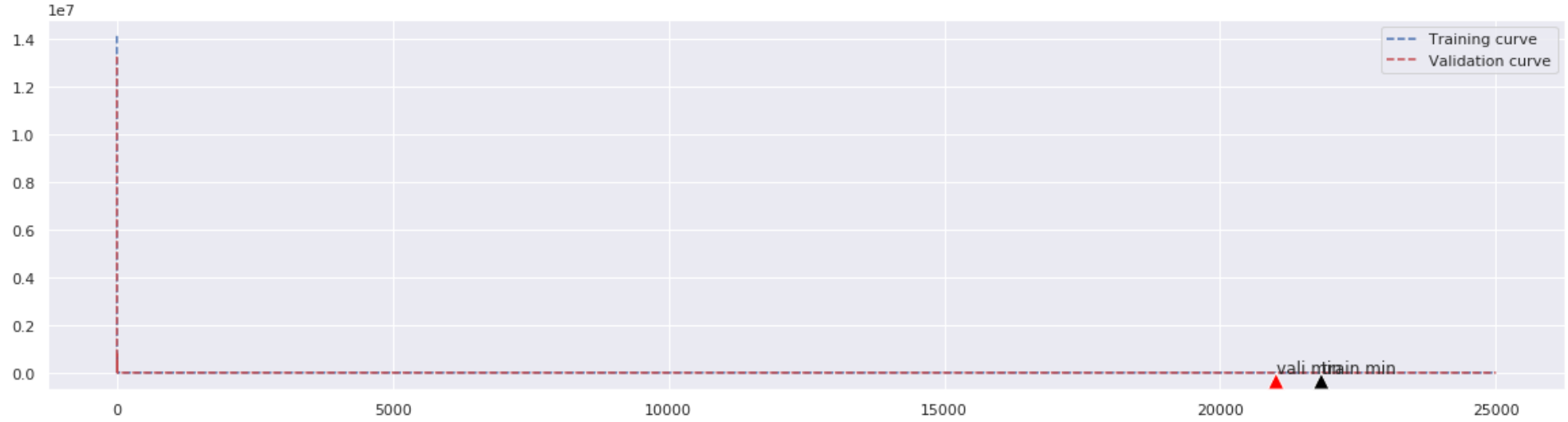
In [165]:

```
training_block(X_train,y_train, X_val,y_val)
prediction = Prediction_block(X_val)
test_rmse_score, test_r2_score = accuracy(y_val,prediction)

print('ann root mean absolute error: ', test_rmse_score)
print('accuracy score: ', test_r2_score  )


pred_str = 'ANN_lr'+str(learning_rate)+'_beta'+str(beta1)+'-'+str(beta2)+'-'+str(beta3)+'-'+str(beta4)+'_hidden'+str(hidden_1)+'-'
+str(hidden_2)+'-'+str(hidden_3)+'-'+str(hidden_4)
prediction_dict[pred_str] = prediction

if submit:
    submit_prediction = Prediction_block(test_processed)
    submit_prediction_dict[pred_str] = submit_prediction
# Data Save
if save_score:
    log_df = pd.read_csv("diffrent_training_results.csv")
    log_df = log_df.append({'learning_rate' : learning_rate, 'num_steps' : num_steps, 'beta1' : beta1, 'beta2' : beta2, 'beta3' :
beta3, 'beta4' : beta4, 'hidden_1' : hidden_1 , 'hidden_2' : hidden_2, 'hidden_3' : hidden_3, 'hidden_4' : hidden_4, 'input_dim' :
input_dim , 'test_rmse_score' : test_rmse_score , 'test_r2_score' : test_r2_score}, ignore_index=True)
    log_df.to_csv("diffrent_training_results.csv", encoding='utf-8',index=False)
```

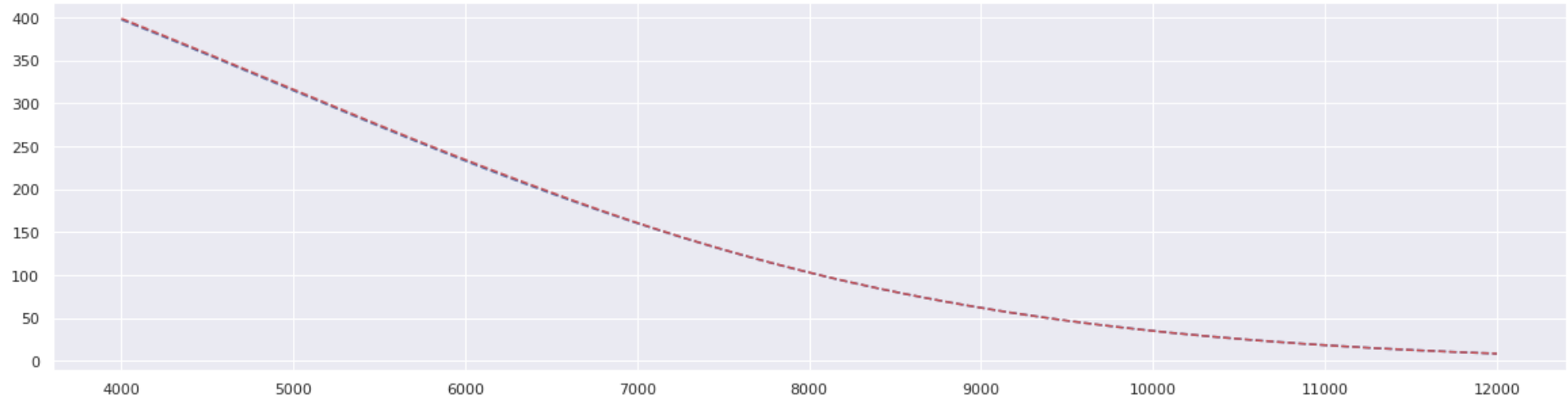
```
epoch no :  500   training loss:  689.5297   validation loss:  697.3612     minimum_validation_loss 0.02101
epoch no :  1000   training loss:  616.1182   validation loss:  621.04333     minimum_validation_loss 0.02101
epoch no :  1500   training loss:  567.1678   validation loss:  570.49713     minimum_validation_loss 0.02101
epoch no :  2000   training loss:  534.4771   validation loss:  536.7388     minimum_validation_loss 0.02101
epoch no :  2500   training loss:  504.62146   validation loss:  506.37347     minimum_validation_loss 0.02101
epoch no :  3000   training loss:  472.07227   validation loss:  473.5253     minimum_validation_loss 0.02101
epoch no :  3500   training loss:  436.13715   validation loss:  437.49036     minimum_validation_loss 0.02101
epoch no :  4000   training loss:  397.3561   validation loss:  398.62814     minimum_validation_loss 0.02101
epoch no :  4500   training loss:  356.56857   validation loss:  357.82224     minimum_validation_loss 0.02101
epoch no :  5000   training loss:  314.80746   validation loss:  316.06567     minimum_validation_loss 0.02101
epoch no :  5500   training loss:  273.2062   validation loss:  274.486     minimum_validation_loss 0.02101
epoch no :  6000   training loss:  232.91711   validation loss:  234.31366     minimum_validation_loss 0.02101
epoch no :  6500   training loss:  195.01007   validation loss:  196.22589     minimum_validation_loss 0.02101
epoch no :  7000   training loss:  160.33734   validation loss:  160.87321     minimum_validation_loss 0.02101
epoch no :  7500   training loss:  129.51091   validation loss:  129.74931     minimum_validation_loss 0.02101
epoch no :  8000   training loss:  102.8486   validation loss:  103.049095     minimum_validation_loss 0.02101
epoch no :  8500   training loss:  80.37189   validation loss:  80.46686     minimum_validation_loss 0.02101
epoch no :  9000   training loss:  61.864494   validation loss:  61.891174     minimum_validation_loss 0.02101
epoch no :  9500   training loss:  46.91856   validation loss:  46.914936     minimum_validation_loss 0.02101
epoch no :  10000   training loss:  35.031296   validation loss:  35.025192     minimum_validation_loss 0.02101
epoch no :  10500   training loss:  25.684698   validation loss:  25.678593     minimum_validation_loss 0.02101
epoch no :  11000   training loss:  18.408863   validation loss:  18.402756     minimum_validation_loss 0.02101
epoch no :  11500   training loss:  12.811649   validation loss:  12.805543     minimum_validation_loss 0.02101
epoch no :  12000   training loss:  8.583126   validation loss:  8.577021     minimum_validation_loss 0.02101
epoch no :  12500   training loss:  5.4819746   validation loss:  5.475868     minimum_validation_loss 0.02101
epoch no :  13000   training loss:  3.3098226   validation loss:  3.3037167     minimum_validation_loss 0.02101
epoch no :  13500   training loss:  1.883859   validation loss:  1.877753     minimum_validation_loss 0.02101
epoch no :  14000   training loss:  1.022428   validation loss:  1.016322     minimum_validation_loss 0.02101
epoch no :  14500   training loss:  0.55073935   validation loss:  0.5446333     minimum_validation_loss 0.02101
epoch no :  15000   training loss:  0.18164402   validation loss:  0.18109342     minimum_validation_loss 0.02101
epoch no :  15500   training loss:  0.075847864   validation loss:  0.07612695     minimum_validation_loss 0.02101
epoch no :  16000   training loss:  0.03628137   validation loss:  0.036539715     minimum_validation_loss 0.02101
epoch no :  16500   training loss:  0.023247756   validation loss:  0.023018867     minimum_validation_loss 0.02101
epoch no :  17000   training loss:  0.0223353   validation loss:  0.021694785     minimum_validation_loss 0.018920038
epoch no :  17500   training loss:  0.021896552   validation loss:  0.021579536     minimum_validation_loss 0.017993538
epoch no :  18000   training loss:  0.024100091   validation loss:  0.022835236     minimum_validation_loss 0.017712258
epoch no :  18500   training loss:  0.020218534   validation loss:  0.019985007     minimum_validation_loss 0.017706223
epoch no :  19000   training loss:  0.036242306   validation loss:  0.03316517     minimum_validation_loss 0.017685074
epoch no :  19500   training loss:  0.019369708   validation loss:  0.019250423     minimum_validation_loss 0.0175728
epoch no :  20000   training loss:  0.017847683   validation loss:  0.017864604     minimum_validation_loss 0.01706047
epoch no :  20500   training loss:  0.028777868   validation loss:  0.029321767     minimum_validation_loss 0.016584832
epoch no :  21000   training loss:  0.016673645   validation loss:  0.016915029     minimum_validation_loss 0.01614659
epoch no :  21500   training loss:  0.017190948   validation loss:  0.017391069     minimum_validation_loss 0.015914697
epoch no :  22000   training loss:  0.017947957   validation loss:  0.017825836     minimum_validation_loss 0.015914697
epoch no :  22500   training loss:  0.0267585   validation loss:  0.027188936     minimum_validation_loss 0.015914697
epoch no :  23000   training loss:  0.02010389   validation loss:  0.019641459     minimum_validation_loss 0.015914697
epoch no :  23500   training loss:  0.020483635   validation loss:  0.020583564     minimum_validation_loss 0.015914697
epoch no :  24000   training loss:  0.019735338   validation loss:  0.019650808     minimum_validation_loss 0.015914697
epoch no :  24500   training loss:  0.021665059   validation loss:  0.021613348     minimum_validation_loss 0.015914697
epoch no :  25000   training loss:  0.05241191   validation loss:  0.051599044     minimum_validation_loss 0.015914697
Model saved in path: model/model.ckpt
```

```
INFO:tensorflow:Restoring parameters from model/model.ckpt
Model restored.
ann root mean absolute error:  0.10759422408562268
accuracy score:  0.9234097934900477
```

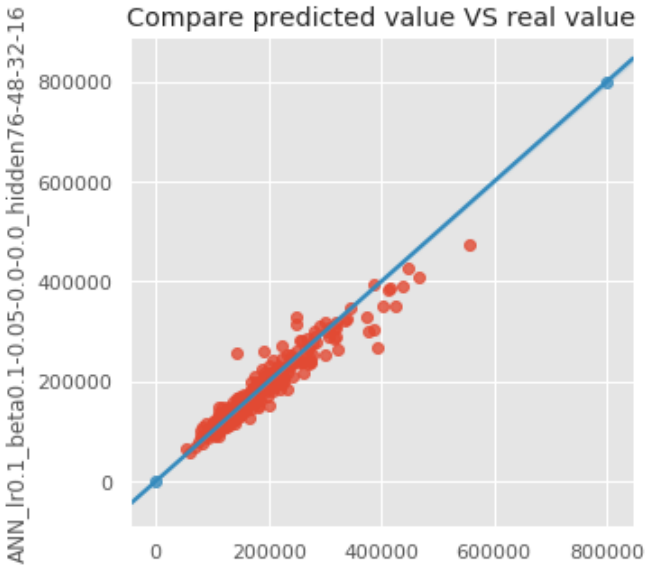
In [166]:
```
#Following variables are only used to zoom into the graph
start_observation_flag = 4000
end_observation_flag = 12000
learning_curve(start_observation_flag,end_observation_flag)
```



If we zoom into the curve we would have seen the following



In [206]:
```
plot_prediction( y_val, pred_df['ANN_lr0.1_beta0.1-0.05-0.0-0.0_hidden76-48-32-16'])
```



Both of the curve actually seems to be on top of each other.The reason is:

- I have applied log transformation on the SalePrice and I have also transformed all my numerical data thats why the difference between the training loss and validation loss seems to be very small and very stable.
- For loss function I have used Mean Squared Error (MSE). For reducing MSE I have used SUM_BY_NONZERO_WEIGHTS which divided scalar sum by number of non-zero weights. MSE calculates squared error for all the data and then calculate the mean. Now, all my SalePrice is very small due to normalization (between 10 to 13.5). Where mean of saleprice is 12.02 . Suppose in nth epoch if
  - for training loss
    - a saleprice is 11.5 and prediction is 12 Squred error .25
    - another saleprice is 13 and prediction is 12 Squred error 1
    - another saleprice is 12.5 and prediction is 12 Squred error .25
    - another saleprice is 11.5 and prediction is 12 Squred error .25
    - another saleprice is 10.3 and prediction is 12 Squred error 1.7

    MSE = (.25+1+.25+.25+1.7)/5 = .69
  - for validation loss
    - another saleprice is 12.9 and prediction is 12 Squred error .81
    - another saleprice is 13.3 and prediction is 12 Squred error 1.69
    - another saleprice is 10.8 and prediction is 12 Squred error 1.44
    - another saleprice is 11.3 and prediction is 12 Squred error .49
    - another saleprice is 11.8 and prediction is 12 Squred error .04

    MSE = (.81+1.69+1.44+.49+.04)/5 = .894

    Difference between validation loss and training loss is .204

    Usually in regression problem neural network stats to predicts the average value within 5-20 epoch so very quickly the difference between val_loss and training_loss gets much lower. In our dummy example difference is already .204 and if its epoch no is 10, by the time it reaches to 500 epoch the difference could go as low as 10^-4.

ANN 2

In this section changed variables are

- learning rate = .05

layer name	Neuron	value of beta for l2 regularization
1st hidden layer	8 Neuron	.005
2nd hidden layer	32 Neuron	.1
3rd hidden layer	16 Neuron	0.05
4th hidden layer	8 Neuron	0

In [167]:

```
tf.reset_default_graph()
learning_rate = 0.05
num_steps = 25000
#for regularize weight matrix
beta1 = 0.005
beta2 = 0.1
beta3 = 0.05
beta4 = 0.0

hidden_1 = 8
hidden_2 = 32
hidden_3 = 16
hidden_4 = 8


minimum_validation_loss = 0.02101000
#tf graph input
X_tf = tf.placeholder("float" )
y_tf = tf.placeholder("float" )

weight_bais()
miscellaneous_initialization()
train_LC = []
val_LC = []
```

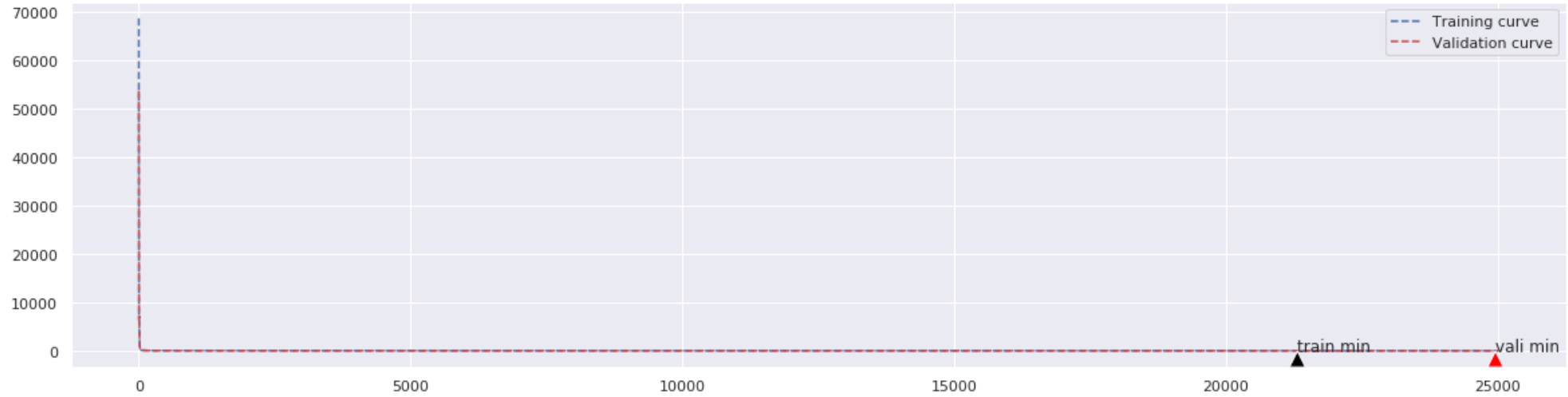
In [168]:

```
training_block(X_train,y_train, X_val,y_val)
prediction = Prediction_block(X_val)

test_rmse_score, test_r2_score = accuracy(y_val,prediction)

print('ann root mean absolute error: ', test_rmse_score)
print('accuracy score: ', test_r2_score  )
# learning_curve(start_observation_flag,end_observation_flag)



pred_str = 'ANN_lr'+str(learning_rate)+'_beta'+str(beta1)+'-'+str(beta2)+'-'+str(beta3)+'-'+str(beta4)+'_hidden'+str(hidden_1)+'-'
+str(hidden_2)+'-'+str(hidden_3)+'-'+str(hidden_4)
prediction_dict[pred_str] = prediction

if submit:
    submit_prediction = Prediction_block(test_processed)
    submit_prediction_dict[pred_str] = submit_prediction
# Data Save
if save_score:
    log_df = pd.read_csv("diffrent_training_results.csv")
    log_df = log_df.append({'learning_rate' : learning_rate, 'num_steps' : num_steps, 'beta1' : beta1, 'beta2' : beta2, 'beta3' :
beta3, 'beta4' : beta4, 'hidden_1' : hidden_1 , 'hidden_2' : hidden_2, 'hidden_3' : hidden_3, 'hidden_4' : hidden_4, 'input_dim' :
input_dim , 'test_rmse_score' : test_rmse_score , 'test_r2_score' : test_r2_score}, ignore_index=True)
    log_df.to_csv("diffrent_training_results.csv", encoding='utf-8',index=False)
```

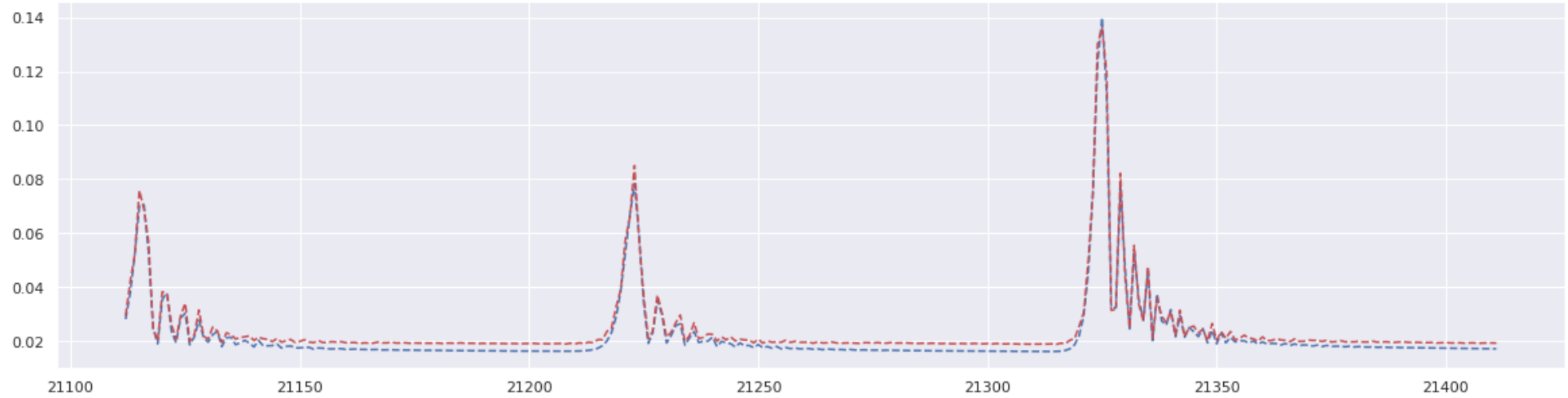
```
epoch no :  500   training loss:  37.6267   validation loss:  38.978355     minimum_validation_loss 0.02101
epoch no :  1000   training loss:  27.510735   validation loss:  27.943092     minimum_validation_loss 0.02101
epoch no :  1500   training loss:  23.682922   validation loss:  23.832542     minimum_validation_loss 0.02101
epoch no :  2000   training loss:  20.943321   validation loss:  21.053318     minimum_validation_loss 0.02101
epoch no :  2500   training loss:  18.664837   validation loss:  18.741512     minimum_validation_loss 0.02101
epoch no :  3000   training loss:  16.61599   validation loss:  16.669695     minimum_validation_loss 0.02101
epoch no :  3500   training loss:  14.703579   validation loss:  14.734726     minimum_validation_loss 0.02101
epoch no :  4000   training loss:  12.837566   validation loss:  12.860882     minimum_validation_loss 0.02101
epoch no :  4500   training loss:  11.050297   validation loss:  11.093956     minimum_validation_loss 0.02101
epoch no :  5000   training loss:  9.367546   validation loss:  9.442308     minimum_validation_loss 0.02101
epoch no :  5500   training loss:  7.799547   validation loss:  7.8686986     minimum_validation_loss 0.02101
epoch no :  6000   training loss:  6.3637867   validation loss:  6.4025187     minimum_validation_loss 0.02101
epoch no :  6500   training loss:  5.085393   validation loss:  5.2863364     minimum_validation_loss 0.02101
epoch no :  7000   training loss:  3.9699533   validation loss:  4.1281457     minimum_validation_loss 0.02101
epoch no :  7500   training loss:  3.0310469   validation loss:  3.074746     minimum_validation_loss 0.02101
epoch no :  8000   training loss:  2.2571797   validation loss:  2.2608483     minimum_validation_loss 0.02101
epoch no :  8500   training loss:  1.6329634   validation loss:  1.6376603     minimum_validation_loss 0.02101
epoch no :  9000   training loss:  1.1535571   validation loss:  1.1554539     minimum_validation_loss 0.02101
epoch no :  9500   training loss:  0.78964365   validation loss:  0.7916806     minimum_validation_loss 0.02101
epoch no :  10000   training loss:  0.52650845   validation loss:  0.5281081     minimum_validation_loss 0.02101
epoch no :  10500   training loss:  0.34919658   validation loss:  0.3502559     minimum_validation_loss 0.02101
epoch no :  11000   training loss:  0.20667128   validation loss:  0.20794587     minimum_validation_loss 0.02101
epoch no :  11500   training loss:  0.12266491   validation loss:  0.12416926     minimum_validation_loss 0.02101
epoch no :  12000   training loss:  0.0751102   validation loss:  0.07606231     minimum_validation_loss 0.02101
epoch no :  12500   training loss:  0.049861707   validation loss:  0.051290333     minimum_validation_loss 0.02101
epoch no :  13000   training loss:  0.02827235   validation loss:  0.029945958     minimum_validation_loss 0.02101
epoch no :  13500   training loss:  0.04991171   validation loss:  0.048592627     minimum_validation_loss 0.02101
epoch no :  14000   training loss:  0.028847948   validation loss:  0.029059194     minimum_validation_loss 0.02101
epoch no :  14500   training loss:  0.020585075   validation loss:  0.022505341     minimum_validation_loss 0.02101
epoch no :  15000   training loss:  0.018651668   validation loss:  0.020893693     minimum_validation_loss 0.020906245
epoch no :  15500   training loss:  0.01882105   validation loss:  0.021152975     minimum_validation_loss 0.020680087
epoch no :  16000   training loss:  0.01809584   validation loss:  0.020696577     minimum_validation_loss 0.020539427
epoch no :  16500   training loss:  0.019732365   validation loss:  0.021994274     minimum_validation_loss 0.020291
epoch no :  17000   training loss:  0.018076668   validation loss:  0.02047741     minimum_validation_loss 0.020051764
epoch no :  17500   training loss:  0.017927157   validation loss:  0.020241098     minimum_validation_loss 0.019826028
epoch no :  18000   training loss:  0.047517974   validation loss:  0.0517275     minimum_validation_loss 0.019683957
epoch no :  18500   training loss:  0.027257835   validation loss:  0.030908309     minimum_validation_loss 0.019573662
epoch no :  19000   training loss:  0.017325712   validation loss:  0.01978569     minimum_validation_loss 0.019429322
epoch no :  19500   training loss:  0.017351553   validation loss:  0.019733     minimum_validation_loss 0.01934588
epoch no :  20000   training loss:  0.016998997   validation loss:  0.019527124     minimum_validation_loss 0.01911965
epoch no :  20500   training loss:  0.019064577   validation loss:  0.020734243     minimum_validation_loss 0.01911965
epoch no :  21000   training loss:  0.023555102   validation loss:  0.023552503     minimum_validation_loss 0.018758774
epoch no :  21500   training loss:  0.019659821   validation loss:  0.021613965     minimum_validation_loss 0.018758774
epoch no :  22000   training loss:  0.03497848   validation loss:  0.03297078     minimum_validation_loss 0.018667541
epoch no :  22500   training loss:  0.017689392   validation loss:  0.018813692     minimum_validation_loss 0.018278366
epoch no :  23000   training loss:  0.019469261   validation loss:  0.020195637     minimum_validation_loss 0.0179243
epoch no :  23500   training loss:  0.0179356   validation loss:  0.019082114     minimum_validation_loss 0.0179243
epoch no :  24000   training loss:  0.016888814   validation loss:  0.018200478     minimum_validation_loss 0.017910173
epoch no :  24500   training loss:  0.016851548   validation loss:  0.01783916     minimum_validation_loss 0.01786506
epoch no :  25000   training loss:  0.016798453   validation loss:  0.018016163     minimum_validation_loss 0.017572396
Model saved in path: model/model.ckpt
```

```
INFO:tensorflow:Restoring parameters from model/model.ckpt
Model restored.
ann root mean absolute error:  0.10801976770634326
accuracy score:  0.9228027548322302
```

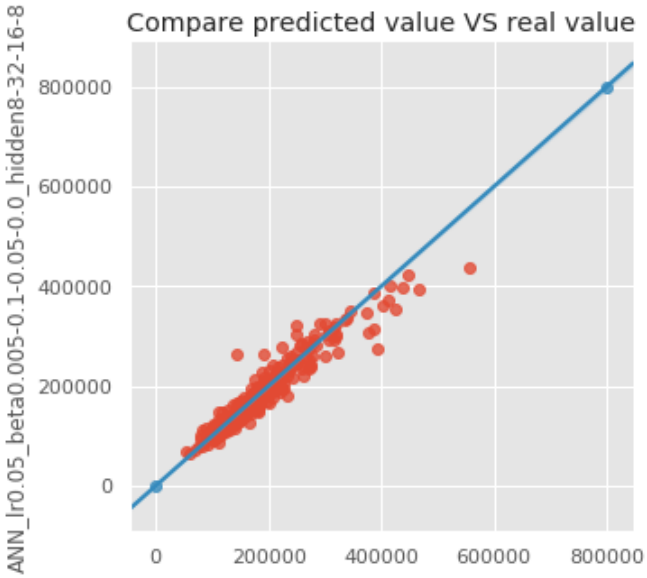
In [169]:
```
#Following variables are only used to zoom into the graph
start_observation_flag = train_LC.index( min(train_LC)) - 200
end_observation_flag =  train_LC.index( min(train_LC)) + 100
learning_curve(start_observation_flag,end_observation_flag)
```



If we zoom into the curve we would have seen the following



In [207]:
```
plot_prediction( y_val, pred_df['ANN_lr0.05_beta0.005-0.1-0.05-0.0_hidden8-32-16-8'])
```



ANN 3

- learning rate = .05

layer name	Neuron	value of beta for l2 regularization
1st hidden layer	16 Neuron	.1
2nd hidden layer	8 Neuron	.0
3rd hidden layer	4 Neuron	0.0
4th hidden layer	2 Neuron	0

In [170]:

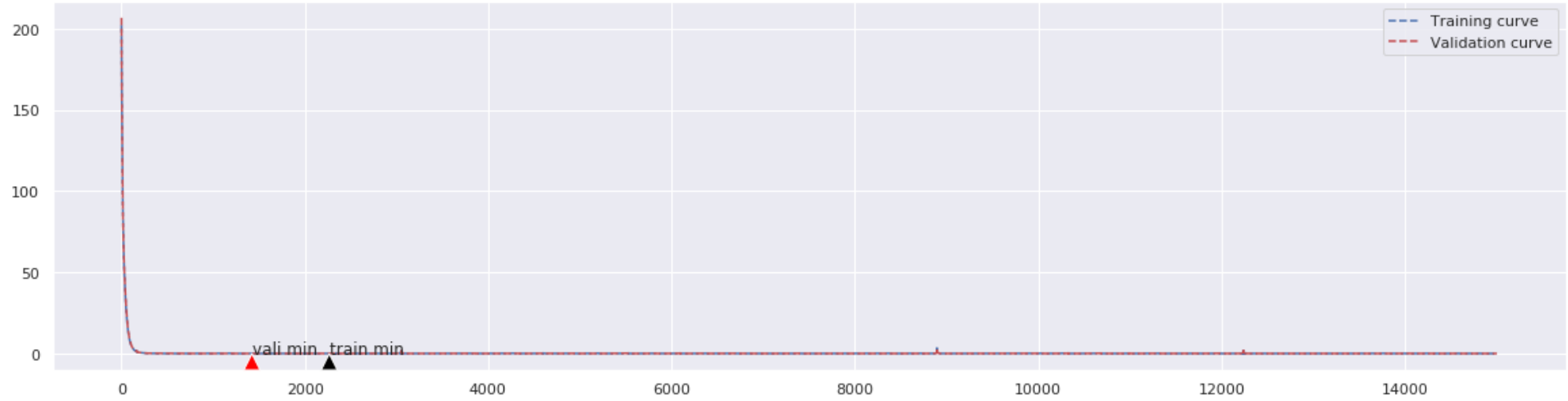
```
tf.reset_default_graph()
learning_rate = 0.05
num_steps = 15000
#for regularize weight matrix
beta1 = 0.1
beta2 = 0.0
beta3 = 0.0
beta4 = 0.0

hidden_1 = 16
hidden_2 = 8
hidden_3 = 4
hidden_4 = 2

minimum_validation_loss = 0.01901000

#tf graph input
X_tf = tf.placeholder("float" )
y_tf = tf.placeholder("float" )


weight_bais()
miscellaneous_initialization()
train_LC = []
val_LC = []
```

In [171]:

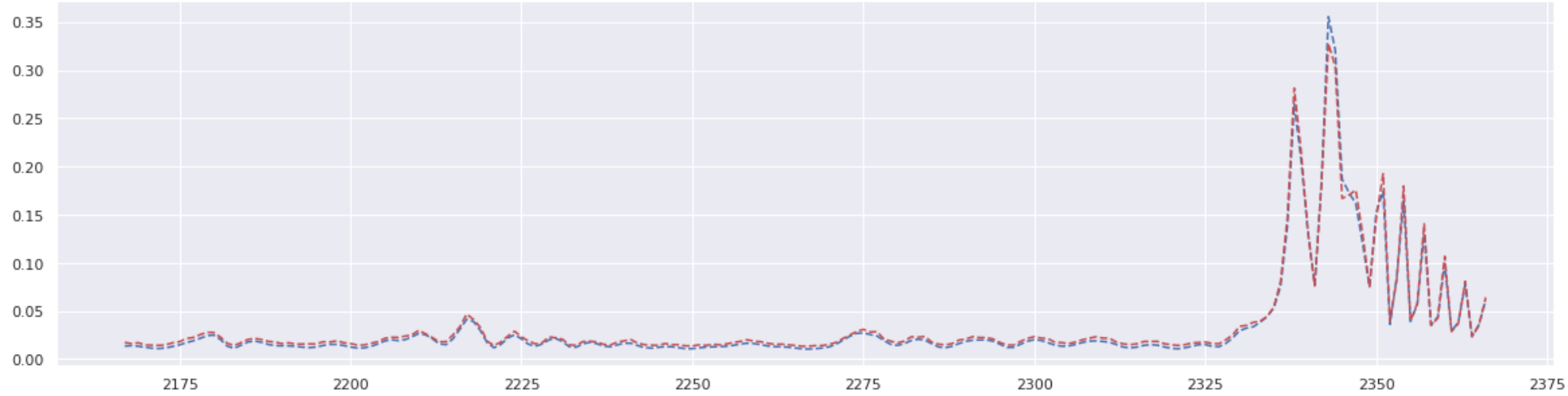
```
training_block(X_train,y_train, X_val,y_val)
prediction = Prediction_block(X_val)
test_rmse_score, test_r2_score = accuracy(y_val,prediction)

print('ann root mean absolute error: ', test_rmse_score)
print('accuracy score: ', test_r2_score  )
# learning_curve(start_observation_flag,end_observation_flag)

pred_str = 'ANN_lr'+str(learning_rate)+'_beta'+str(beta1)+'-'+str(beta2)+'-'+str(beta3)+'-'+str(beta4)+'_hidden'+str(hidden_1)+'-'
+str(hidden_2)+'-'+str(hidden_3)+'-'+str(hidden_4)
prediction_dict[pred_str] = prediction

if submit:
    submit_prediction = Prediction_block(test_processed)
    submit_prediction_dict[pred_str] = submit_prediction
# Data Save
if save_score:
    log_df = pd.read_csv("diffrent_training_results.csv")
    log_df = log_df.append({'learning_rate' : learning_rate,
                            'num_steps' : num_steps, 'beta1' : beta1,
                            'beta2' : beta2, 'beta3' : beta3, 'beta4' : beta4,
                            'hidden_1' : hidden_1 , 'hidden_2' : hidden_2,
                            'hidden_3' : hidden_3, 'hidden_4' : hidden_4, 'input_dim' : input_dim ,
                            'test_rmse_score' : test_rmse_score ,
                            'test_r2_score' : test_r2_score}, ignore_index=True)

    log_df.to_csv("diffrent_training_results.csv", encoding='utf-8',index=False)
```

```
epoch no :  500   training loss:  0.021373276   validation loss:  0.023275886     minimum_validation_loss 0.01901
epoch no :  1000   training loss:  0.012365959   validation loss:  0.014759562     minimum_validation_loss 0.012888228
epoch no :  1500   training loss:  0.012171449   validation loss:  0.014154333     minimum_validation_loss 0.012697128
epoch no :  2000   training loss:  0.014479765   validation loss:  0.016879402     minimum_validation_loss 0.012697128
epoch no :  2500   training loss:  0.011750519   validation loss:  0.014684602     minimum_validation_loss 0.012697128
epoch no :  3000   training loss:  0.011854692   validation loss:  0.014007693     minimum_validation_loss 0.012697128
epoch no :  3500   training loss:  0.013063474   validation loss:  0.0133137135     minimum_validation_loss 0.012697128
epoch no :  4000   training loss:  0.01293024   validation loss:  0.013209097     minimum_validation_loss 0.012697128
epoch no :  4500   training loss:  0.012487516   validation loss:  0.012863866     minimum_validation_loss 0.012697128
epoch no :  5000   training loss:  0.01277885   validation loss:  0.013111608     minimum_validation_loss 0.012697128
epoch no :  5500   training loss:  0.012645144   validation loss:  0.012987481     minimum_validation_loss 0.012697128
epoch no :  6000   training loss:  0.012862484   validation loss:  0.013154824     minimum_validation_loss 0.012697128
epoch no :  6500   training loss:  0.012916587   validation loss:  0.013200009     minimum_validation_loss 0.012697128
epoch no :  7000   training loss:  0.012861615   validation loss:  0.013189734     minimum_validation_loss 0.012697128
epoch no :  7500   training loss:  0.013094541   validation loss:  0.013334069     minimum_validation_loss 0.012697128
epoch no :  8000   training loss:  0.013130042   validation loss:  0.013430863     minimum_validation_loss 0.012697128
epoch no :  8500   training loss:  0.0130512575   validation loss:  0.013298386     minimum_validation_loss 0.012697128
epoch no :  9000   training loss:  0.018664096   validation loss:  0.01802725     minimum_validation_loss 0.012697128
epoch no :  9500   training loss:  0.014034299   validation loss:  0.014094248     minimum_validation_loss 0.012697128
epoch no :  10000   training loss:  0.0130161205   validation loss:  0.013274895     minimum_validation_loss 0.012697128
epoch no :  10500   training loss:  0.0135711385   validation loss:  0.013546365     minimum_validation_loss 0.012697128
epoch no :  11000   training loss:  0.014884353   validation loss:  0.015181476     minimum_validation_loss 0.012697128
epoch no :  11500   training loss:  0.013366242   validation loss:  0.013626482     minimum_validation_loss 0.012697128
epoch no :  12000   training loss:  0.014005462   validation loss:  0.014176171     minimum_validation_loss 0.012697128
epoch no :  12500   training loss:  0.016458847   validation loss:  0.016159832     minimum_validation_loss 0.012697128
epoch no :  13000   training loss:  0.013437911   validation loss:  0.013710307     minimum_validation_loss 0.012697128
epoch no :  13500   training loss:  0.013294996   validation loss:  0.013504602     minimum_validation_loss 0.012697128
epoch no :  14000   training loss:  0.013023082   validation loss:  0.013275907     minimum_validation_loss 0.012697128
epoch no :  14500   training loss:  0.013095467   validation loss:  0.013344189     minimum_validation_loss 0.012697128
epoch no :  15000   training loss:  0.012866392   validation loss:  0.013178272     minimum_validation_loss 0.012697128
Model saved in path: model/model.ckpt
INFO:tensorflow:Restoring parameters from model/model.ckpt
Model restored.
ann root mean absolute error:  0.10418463979323321
accuracy score:  0.9281870589031296
```

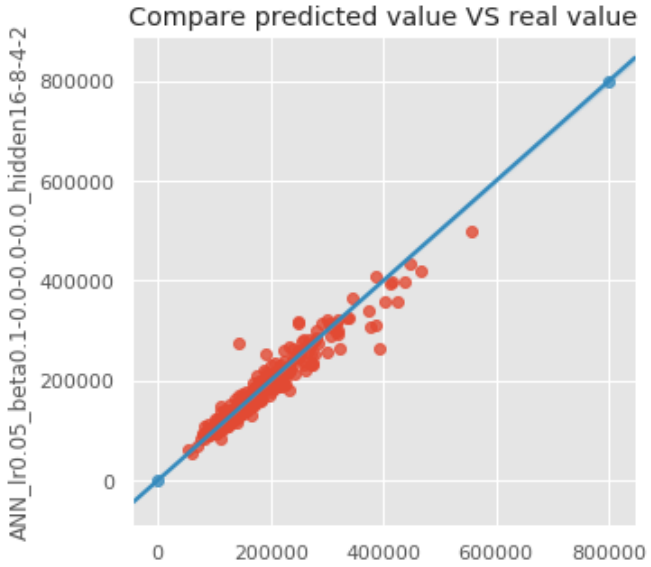
In [172]:
```
#Following variables are only used to zoom into the graph
start_observation_flag = train_LC.index( min(train_LC)) - 100
end_observation_flag =  train_LC.index( min(train_LC)) + 100
learning_curve(start_observation_flag,end_observation_flag)
```



If we zoom into the curve we would have seen the following



In [208]:
```
plot_prediction( y_val, pred_df['ANN_lr0.05_beta0.1-0.0-0.0-0.0_hidden16-8-4-2'])
```



In [173]:
```
with tf.Session() as sess:
    try:
        # Restore variables from disk.
        saver.restore(sess, "model/model.ckpt")
        saver.save(sess, "model/model_ext/model.ckpt")
        print("Model Saved for ensemble.")
    except:
        print("------------ available checkpoint is for different model --------------")
```

```
INFO:tensorflow:Restoring parameters from model/model.ckpt
Model Saved for ensemble.
```

Description on Learning curve and Accuracy:

We can observe where overfitting occurs. Overfitting actually occurs if the training loss goes under the validation loss even though the validation is still dropping. It is the sign that network is learning the patterns in the train set that are not applicable in the validation done. In a short note we can say::

Overfitting : training loss << validation loss

Underfitting : training loss >> validation loss

Just right : training loss ~ validation loss

According to this theory, for ANN 1,2 and 3 our both learning curve (validation loss and training loss) is exactly top of one another so in our case validation loss and training loss is almost same so we can say that our model is doing just the right thing. Again In validation score .11,.1081 and .1050 is impressive compared to other models.

Both of the curve actually seems to be on top of each other.The reason is:

- I have applied log transformation on the SalePrice and I have also transformed all my numerical data thats why the difference between the training loss and validation loss seems to be very small and very stable.
- For loss function I have used Mean Squared Error (MSE). For reducing MSE I have used SUM_BY_NONZERO_WEIGHTS which divided scalar sum by number of non-zero weights. MSE calculates squared error for all the data and then calculate the mean. Now, all my SalePrice is very small due to normalization (between 10 to 13.5). Where mean of saleprice is 12.02 . Suppose in nth epoch if
  - for training loss
    - a saleprice is 11.5 and prediction is 12 Squred error .25
    - another saleprice is 13 and prediction is 12 Squred error 1
    - another saleprice is 12.5 and prediction is 12 Squred error .25
    - another saleprice is 11.5 and prediction is 12 Squred error .25
    - another saleprice is 10.3 and prediction is 12 Squred error 1.7

    MSE = (.25+1+.25+.25+1.7)/5 = .69
  - for validation loss
    - another saleprice is 12.9 and prediction is 12 Squred error .81
    - another saleprice is 13.3 and prediction is 12 Squred error 1.69
    - another saleprice is 10.8 and prediction is 12 Squred error 1.44
    - another saleprice is 11.3 and prediction is 12 Squred error .49
    - another saleprice is 11.8 and prediction is 12 Squred error .04

    MSE = (.81+1.69+1.44+.49+.04)/5 = .894

    Difference between validation loss and training loss is .204

    Usually in regression problem neural network stats to predicts the average value within 5-20 epoch so very quickly the difference between val_loss and training_loss gets much lower. In our dummy example difference is already .204 and if its epoch no is 10, by the time it reaches to 500 epoch the difference could go as low as 10^-4.

ANN single hidden layer

In [174]:
```
tf.reset_default_graph()
def weight_bais():
    global weights, biases
    weights = {
        'w1': tf.Variable(tf.random_normal([input_dim, hidden_1])),
        'out': tf.Variable(tf.random_normal([hidden_1, output_dim]))
    }
    biases = {
        'b1': tf.Variable(tf.random_normal([hidden_1])),
        'out': tf.Variable(tf.random_normal([output_dim]))
    }
```

In [175]:
```
def ann_model(X_input):
  # Hidden layers
    layer_1 = tf.add(tf.matmul(X_input, weights['w1']), biases['b1'])
    layer_1 = tf.nn.relu(layer_1)

    # Output layer
    layer_out = tf.matmul(layer_1, weights['out'])+ biases['out']

    return layer_out
```

In [176]:
```
def miscellaneous_initialization():
    global model, loss , regularizer_1 , regularizer_2 ,regularizer_3, regularizer_4, optimizer , init , saver
    # Model Construct
    model = ann_model(X_tf)

    # Mean Squared Error loss function
    loss = tf.losses.mean_squared_error(y_tf , model , reduction=tf.losses.Reduction.SUM_BY_NONZERO_WEIGHTS)

    # loss = tf.square(y_tf - model)
    regularizer_1 = tf.nn.l2_loss(weights['w1'])

    # loss = tf.reduce_mean(loss + beta1*regularizer_1 + beta2*regularizer_2 + beta3*regularizer_3)
    loss = tf.reduce_mean(loss + beta1*regularizer_1 )

    # Adam optimizer will update weights and biases after each step
    optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(loss)


    # Initialize variables
    init = tf.global_variables_initializer()

    # Add ops to save and restore all the variables.
    saver = tf.train.Saver()
```

ANN 4

- learning rate = .1

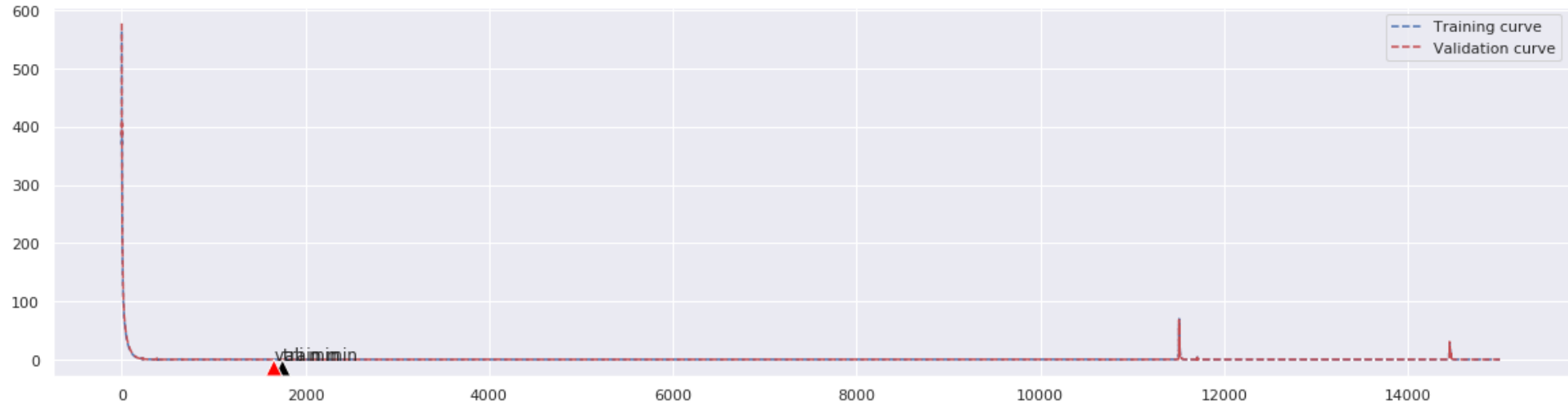
layer name	Neuron	value of beta for l2 regularization
1st hidden layer	16 Neuron	.1

In [177]:

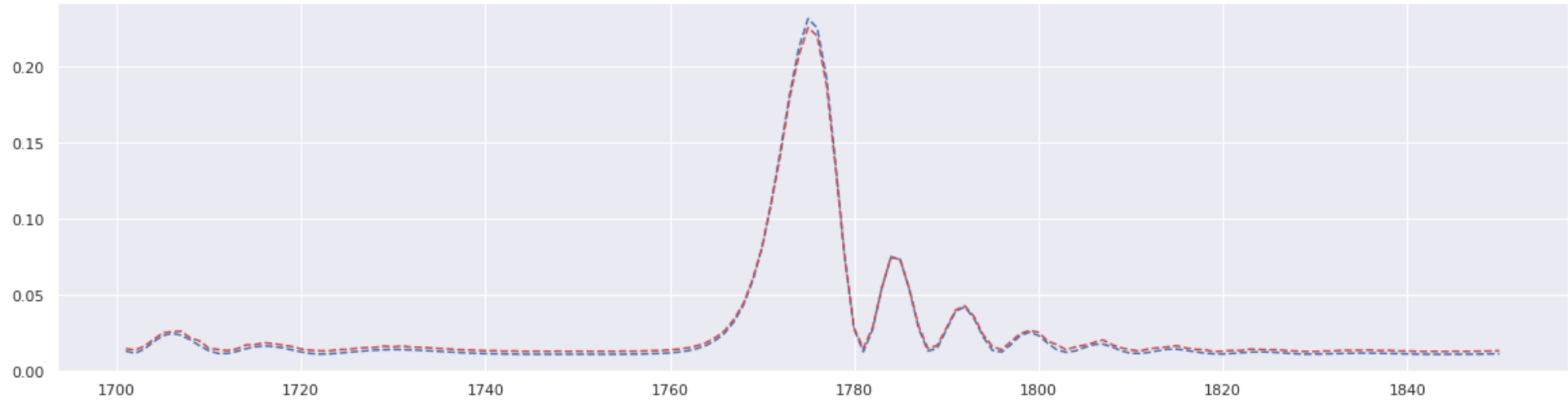
```
tf.reset_default_graph()
learning_rate = 0.1
num_steps = 15000
#for regularize weight matrix
beta1 = 0.1
beta2 = None
beta3 = None
beta4 = None
minimum_validation_loss = 0.01901000
hidden_1 = 16
hidden_2 = None
hidden_3 = None
hidden_4 = None

#tf graph input
X_tf = tf.placeholder("float" )
y_tf = tf.placeholder("float" )

weight_bais()
miscellaneous_initialization()
train_LC = []
val_LC = []
training_block(X_train,y_train, X_val,y_val)
prediction = Prediction_block(X_val)

test_rmse_score, test_r2_score = accuracy(y_val,prediction)

print('ann root mean absolute error: ', test_rmse_score)
print('accuracy score: ', test_r2_score  )
# learning_curve(start_observation_flag,end_observation_flag)


pred_str = 'ANN_lr'+str(learning_rate)+'_beta'+str(beta1)+'-'+str(beta2)+'-'+str(beta3)+'-'+str(beta4)+'_hidden'+str(hidden_1)+'-'
+str(hidden_2)+'-'+str(hidden_3)+'-'+str(hidden_4)
prediction_dict[pred_str] = prediction

if submit:
    submit_prediction = Prediction_block(test_processed)
    submit_prediction_dict[pred_str] = submit_prediction
# Data Save
if save_score:
    log_df = pd.read_csv("diffrent_training_results.csv")
    log_df = log_df.append({'learning_rate' : learning_rate, 'num_steps' : num_steps, 'beta1' : beta1, 'beta2' : beta2, 'beta3' :
beta3, 'beta4' : beta4, 'hidden_1' : hidden_1 , 'hidden_2' : hidden_2, 'hidden_3' : hidden_3, 'hidden_4' : hidden_4, 'input_dim' :
input_dim , 'test_rmse_score' : test_rmse_score , 'test_r2_score' : test_r2_score}, ignore_index=True)
    log_df.to_csv("diffrent_training_results.csv", encoding='utf-8',index=False)
```

```
epoch no :  500   training loss:  0.015023127   validation loss:  0.016788356     minimum_validation_loss 0.016885752
epoch no :  1000   training loss:  0.5317435   validation loss:  0.5155618     minimum_validation_loss 0.012825298
epoch no :  1500   training loss:  0.011146396   validation loss:  0.012717476     minimum_validation_loss 0.01268091
epoch no :  2000   training loss:  0.043104995   validation loss:  0.04419604     minimum_validation_loss 0.012603634
epoch no :  2500   training loss:  0.026202142   validation loss:  0.027487325     minimum_validation_loss 0.012603634
epoch no :  3000   training loss:  0.024001976   validation loss:  0.024989085     minimum_validation_loss 0.012603634
epoch no :  3500   training loss:  0.01209567   validation loss:  0.013693035     minimum_validation_loss 0.012603634
epoch no :  4000   training loss:  0.034115065   validation loss:  0.036607474     minimum_validation_loss 0.012603634
epoch no :  4500   training loss:  0.012893941   validation loss:  0.013268495     minimum_validation_loss 0.012603634
epoch no :  5000   training loss:  0.045774493   validation loss:  0.046305384     minimum_validation_loss 0.012603634
epoch no :  5500   training loss:  0.013231731   validation loss:  0.013570556     minimum_validation_loss 0.012603634
epoch no :  6000   training loss:  0.014462775   validation loss:  0.014696076     minimum_validation_loss 0.012603634
epoch no :  6500   training loss:  0.014413257   validation loss:  0.014290851     minimum_validation_loss 0.012603634
epoch no :  7000   training loss:  0.019640908   validation loss:  0.019496439     minimum_validation_loss 0.012603634
epoch no :  7500   training loss:  0.013458842   validation loss:  0.013642586     minimum_validation_loss 0.012603634
epoch no :  8000   training loss:  0.013325111   validation loss:  0.013581785     minimum_validation_loss 0.012603634
epoch no :  8500   training loss:  0.013964789   validation loss:  0.013985435     minimum_validation_loss 0.012603634
epoch no :  9000   training loss:  0.0135058835   validation loss:  0.013669483     minimum_validation_loss 0.012603634
epoch no :  9500   training loss:  0.014692438   validation loss:  0.014379504     minimum_validation_loss 0.012603634
epoch no :  10000   training loss:  0.015387091   validation loss:  0.014829331     minimum_validation_loss 0.012603634
epoch no :  10500   training loss:  0.0139906565   validation loss:  0.014061734     minimum_validation_loss 0.012603634
epoch no :  11000   training loss:  0.014063163   validation loss:  0.014120329     minimum_validation_loss 0.012603634
epoch no :  11500   training loss:  0.11804387   validation loss:  0.11690733     minimum_validation_loss 0.012603634
epoch no :  12000   training loss:  0.024423271   validation loss:  0.02316959     minimum_validation_loss 0.012603634
epoch no :  12500   training loss:  0.01884674   validation loss:  0.018186796     minimum_validation_loss 0.012603634
epoch no :  13000   training loss:  0.016614432   validation loss:  0.016255913     minimum_validation_loss 0.012603634
epoch no :  13500   training loss:  0.015299929   validation loss:  0.015145771     minimum_validation_loss 0.012603634
epoch no :  14000   training loss:  0.014419476   validation loss:  0.01441915     minimum_validation_loss 0.012603634
epoch no :  14500   training loss:  0.15296379   validation loss:  0.14995757     minimum_validation_loss 0.012603634
epoch no :  15000   training loss:  0.014584897   validation loss:  0.014598845     minimum_validation_loss 0.012603634
Model saved in path: model/model.ckpt
INFO:tensorflow:Restoring parameters from model/model.ckpt
Model restored.
ann root mean absolute error:  0.10573123461204115
accuracy score:  0.9260391435723435
```

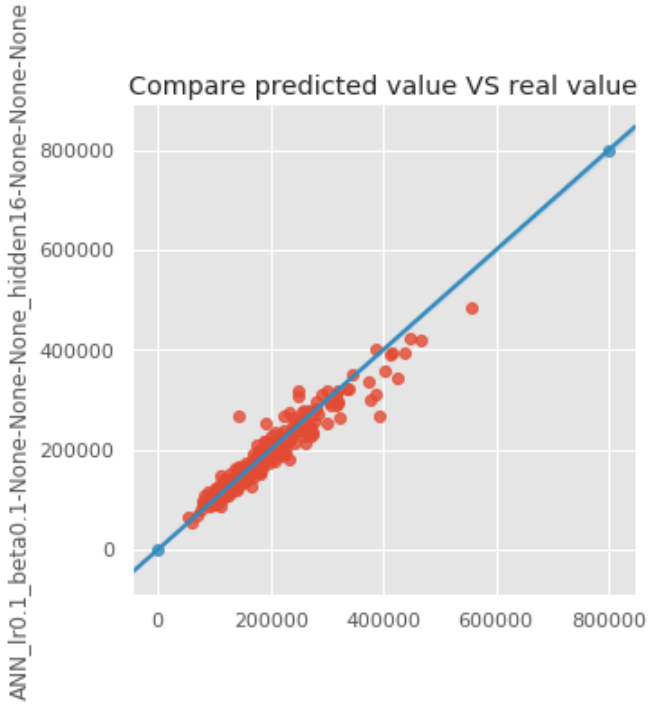
In [178]:
```
#Following variables are only used to zoom into the graph
start_observation_flag = train_LC.index( min(train_LC)) - 50
end_observation_flag =  train_LC.index( min(train_LC)) + 100
learning_curve(start_observation_flag,end_observation_flag)
```



If we zoom into the curve we would have seen the following



In [209]:
```
plot_prediction( y_val, pred_df['ANN_lr0.1_beta0.1-None-None-None_hidden16-None-None-None'])
```



ANN 5

- learning rate = .1

layer name	Neuron	value of beta for l2 regularization
1st hidden layer	4 Neuron	.1

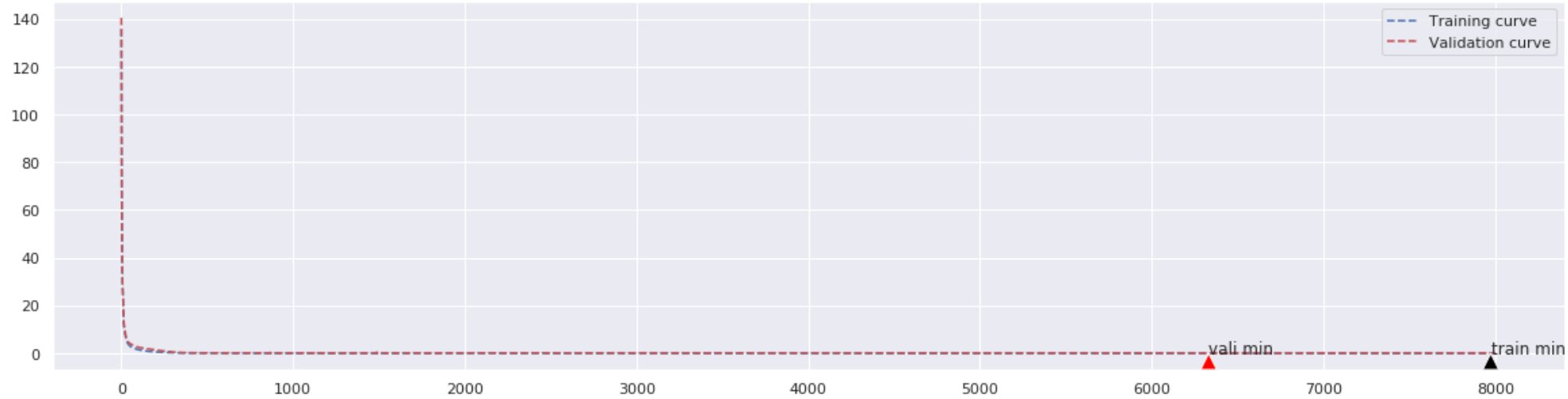
In [179]:

```
tf.reset_default_graph()
learning_rate = 0.1
num_steps = 8000
#for regularize weight matrix
beta1 = 0
beta2 = None
beta3 = None
beta4 = None

hidden_1 = 4
hidden_2 = None
hidden_3 = None
hidden_4 = None
minimum_validation_loss = 0.1701000
#tf graph input
X_tf = tf.placeholder("float" )
y_tf = tf.placeholder("float" )


weight_bais()
miscellaneous_initialization()
train_LC = []
val_LC = []
```

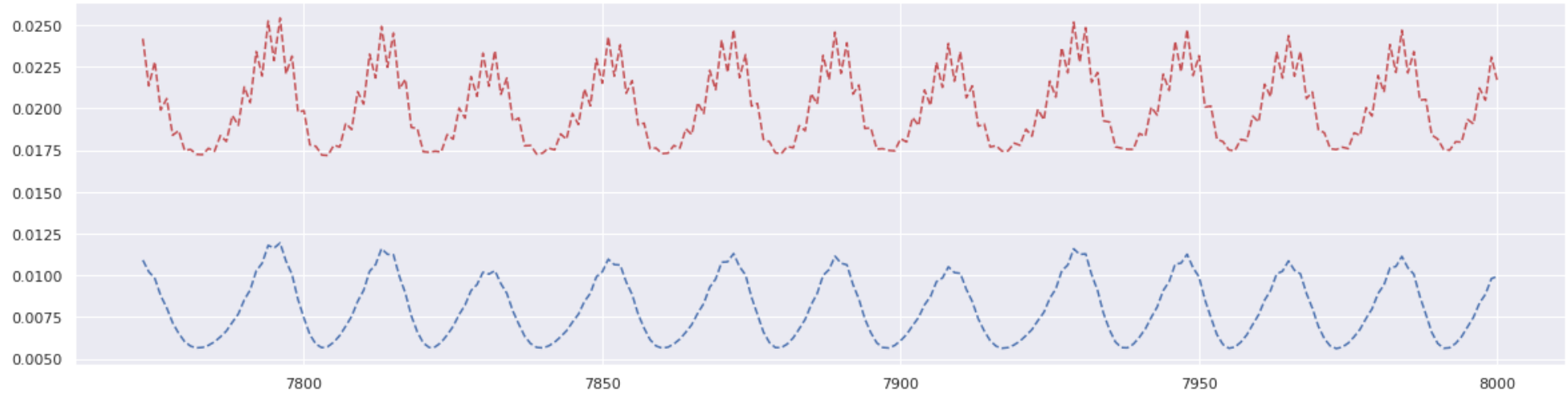
In [180]:
```
training_block(X_train,y_train, X_val,y_val)
prediction = Prediction_block(X_val)

test_rmse_score, test_r2_score = accuracy(y_val,prediction)

print('ann root mean absolute error: ', test_rmse_score)
print('accuracy score: ', test_r2_score  )
# learning_curve(start_observation_flag,end_observation_flag)

pred_str = 'ANN_lr'+str(learning_rate)+'_beta'+str(beta1)+'-'+str(beta2)+'-'+str(beta3)+'-'+str(beta4)+'_hidden'+str(hidden_1)+'-'
+str(hidden_2)+'-'+str(hidden_3)+'-'+str(hidden_4)
prediction_dict[pred_str] = prediction

if submit:
    submit_prediction = Prediction_block(test_processed)
    submit_prediction_dict[pred_str] = submit_prediction
# Data Save
if save_score:
    log_df = pd.read_csv("diffrent_training_results.csv")
    log_df = log_df.append({'learning_rate' : learning_rate, 'num_steps' : num_steps, 'beta1' : beta1, 'beta2' : beta2, 'beta3' :
beta3, 'beta4' : beta4, 'hidden_1' : hidden_1 , 'hidden_2' : hidden_2, 'hidden_3' : hidden_3, 'hidden_4' : hidden_4, 'input_dim' :
input_dim , 'test_rmse_score' : test_rmse_score , 'test_r2_score' : test_r2_score}, ignore_index=True)
    log_df.to_csv("diffrent_training_results.csv", encoding='utf-8',index=False)
```

```
epoch no :  500   training loss:  0.036642373   validation loss:  0.10017427     minimum_validation_loss 0.100658976
epoch no :  1000   training loss:  0.013610061   validation loss:  0.033793807     minimum_validation_loss 0.03379584
epoch no :  1500   training loss:  0.032505546   validation loss:  0.042651974     minimum_validation_loss 0.024058858
epoch no :  2000   training loss:  0.009605219   validation loss:  0.021479186     minimum_validation_loss 0.021180516
epoch no :  2500   training loss:  0.009214793   validation loss:  0.019643791     minimum_validation_loss 0.019576933
epoch no :  3000   training loss:  0.008593406   validation loss:  0.018116053     minimum_validation_loss 0.018103816
epoch no :  3500   training loss:  0.008415474   validation loss:  0.017403528     minimum_validation_loss 0.017070297
epoch no :  4000   training loss:  0.007904806   validation loss:  0.016301744     minimum_validation_loss 0.016226111
epoch no :  4500   training loss:  0.0076621245   validation loss:  0.016145162     minimum_validation_loss 0.01588522
epoch no :  5000   training loss:  0.007776446   validation loss:  0.016533012     minimum_validation_loss 0.015800262
epoch no :  5500   training loss:  0.00672438   validation loss:  0.0154764475     minimum_validation_loss 0.015379271
epoch no :  6000   training loss:  0.0074231895   validation loss:  0.016876103     minimum_validation_loss 0.015195948
epoch no :  6500   training loss:  0.014151724   validation loss:  0.026047615     minimum_validation_loss 0.015188072
epoch no :  7000   training loss:  0.010674133   validation loss:  0.022655414     minimum_validation_loss 0.015188072
epoch no :  7500   training loss:  0.0070397044   validation loss:  0.018908782     minimum_validation_loss 0.015188072
epoch no :  8000   training loss:  0.009947814   validation loss:  0.021635845     minimum_validation_loss 0.015188072
Model saved in path: model/model.ckpt
INFO:tensorflow:Restoring parameters from model/model.ckpt
Model restored.
ann root mean absolute error:  0.12323991132579472
accuracy score:  0.89951573068098
```

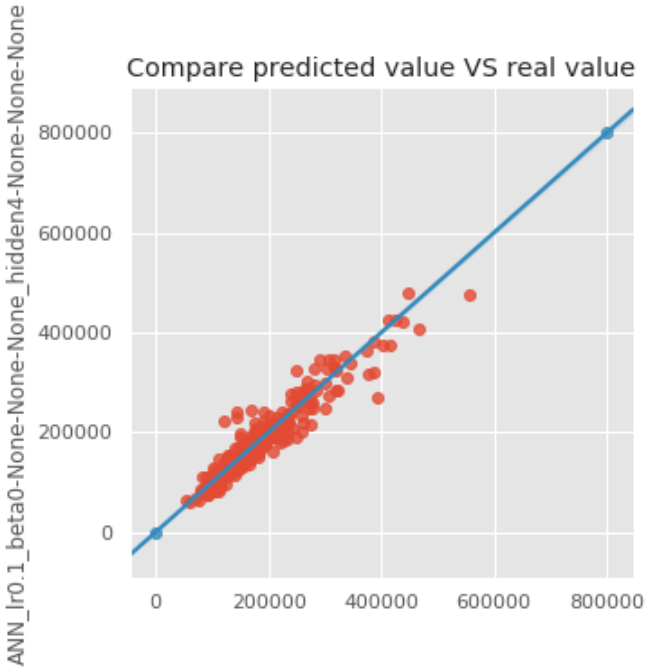
In [181]:
```
#Following variables are only used to zoom into the graph
start_observation_flag = train_LC.index( min(train_LC)) - 200
end_observation_flag =  train_LC.index( min(train_LC)) + 100
learning_curve(start_observation_flag,end_observation_flag)
```



If we zoom into the curve we would have seen the following



In [210]:
```
plot_prediction( y_val, pred_df['ANN_lr0.1_beta0-None-None-None_hidden4-None-None-None'])
```



ANN 6

- learning rate = .1

layer name	Neuron	value of beta for l2 regularization
1st hidden layer	2 Neuron	.1

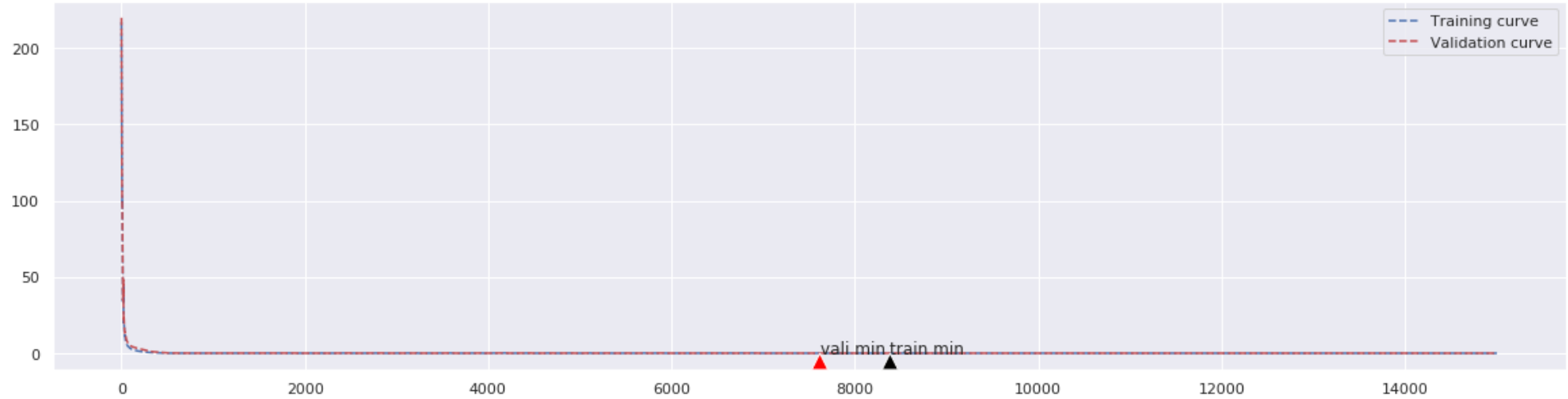
In [182]:

```
tf.reset_default_graph()
learning_rate = 0.1
num_steps = 15000
#for regularize weight matrix
beta1 = 0
beta2 = None
beta3 = None
beta4 = None

hidden_1 = 2
hidden_2 = None
hidden_3 = None
hidden_4 = None
minimum_validation_loss = 0.01901000
#tf graph input
X_tf = tf.placeholder("float" )
y_tf = tf.placeholder("float" )


weight_bais()
miscellaneous_initialization()
train_LC = []
val_LC = []
```

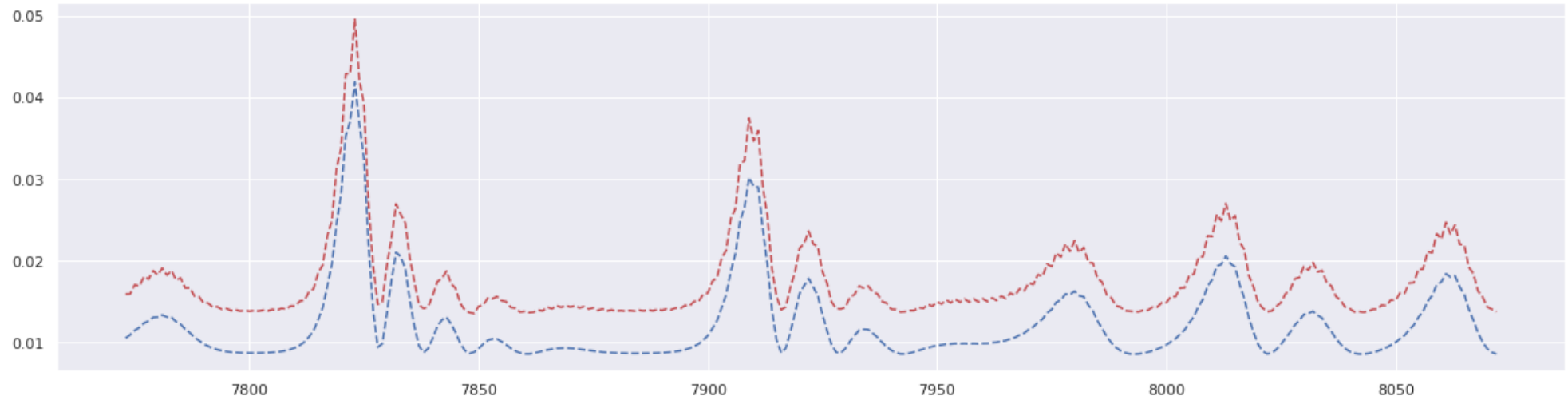
In [183]:

```
training_block(X_train,y_train, X_val,y_val)
prediction = Prediction_block(X_val)

test_rmse_score, test_r2_score = accuracy(y_val,prediction)

print('ann root mean absolute error: ', test_rmse_score)
print('accuracy score: ', test_r2_score  )
learning_curve(start_observation_flag,end_observation_flag)

pred_str = 'ANN_lr'+str(learning_rate)+'_beta'+str(beta1)+'-'+str(beta2)+'-'+str(beta3)+'-'+str(beta4)+'_hidden'+str(hidden_1)+'-'
+str(hidden_2)+'-'+str(hidden_3)+'-'+str(hidden_4)
prediction_dict[pred_str] = prediction

if submit:
    submit_prediction = Prediction_block(test_processed)
    submit_prediction_dict[pred_str] = submit_prediction
# Data Save
if save_score:
    log_df = pd.read_csv("diffrent_training_results.csv")
    log_df = log_df.append({'learning_rate' : learning_rate, 'num_steps' : num_steps, 'beta1' : beta1, 'beta2' : beta2, 'beta3' :
beta3, 'beta4' : beta4, 'hidden_1' : hidden_1 , 'hidden_2' : hidden_2, 'hidden_3' : hidden_3, 'hidden_4' : hidden_4, 'input_dim' :
input_dim , 'test_rmse_score' : test_rmse_score , 'test_r2_score' : test_r2_score}, ignore_index=True)
    log_df.to_csv("diffrent_training_results.csv", encoding='utf-8',index=False)
```

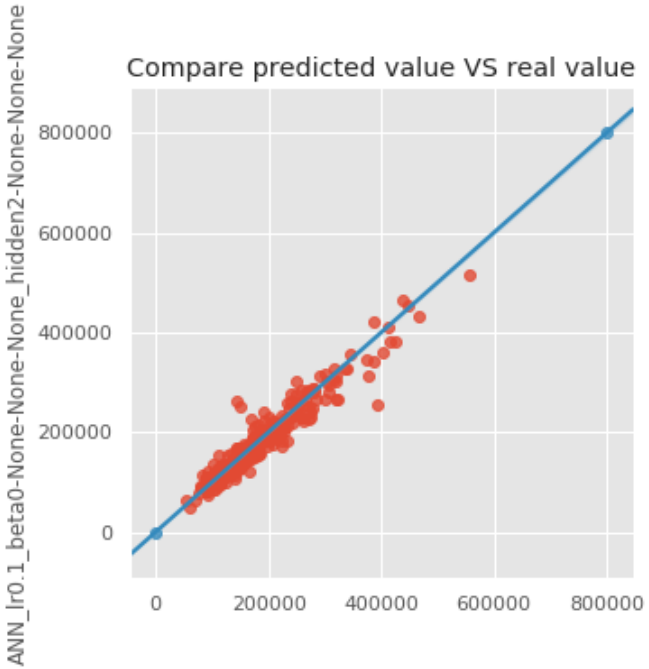
```
epoch no :  500   training loss:  0.055006728   validation loss:  0.17353632     minimum_validation_loss 0.01901
epoch no :  1000   training loss:  0.015379146   validation loss:  0.029070914     minimum_validation_loss 0.01901
epoch no :  1500   training loss:  0.0123758875   validation loss:  0.020271016     minimum_validation_loss 0.01901
epoch no :  2000   training loss:  0.010911904   validation loss:  0.01765618     minimum_validation_loss 0.017668478
epoch no :  2500   training loss:  0.16415106   validation loss:  0.16801585     minimum_validation_loss 0.0166092
epoch no :  3000   training loss:  0.009792692   validation loss:  0.016327215     minimum_validation_loss 0.01632495
epoch no :  3500   training loss:  0.009519739   validation loss:  0.016003141     minimum_validation_loss 0.016005693
epoch no :  4000   training loss:  0.009366161   validation loss:  0.016087644     minimum_validation_loss 0.015710045
epoch no :  4500   training loss:  0.012510911   validation loss:  0.019836118     minimum_validation_loss 0.01516767
epoch no :  5000   training loss:  0.009057123   validation loss:  0.014516197     minimum_validation_loss 0.014396217
epoch no :  5500   training loss:  0.008869484   validation loss:  0.013886349     minimum_validation_loss 0.013850695
epoch no :  6000   training loss:  0.008743159   validation loss:  0.013634956     minimum_validation_loss 0.0135872
epoch no :  6500   training loss:  0.008732803   validation loss:  0.013922713     minimum_validation_loss 0.013583174
epoch no :  7000   training loss:  0.008631376   validation loss:  0.01368356     minimum_validation_loss 0.013519419
epoch no :  7500   training loss:  0.020626204   validation loss:  0.026981445     minimum_validation_loss 0.013519419
epoch no :  8000   training loss:  0.009646868   validation loss:  0.014814932     minimum_validation_loss 0.013489423
epoch no :  8500   training loss:  0.0392291   validation loss:  0.047384102     minimum_validation_loss 0.013489423
epoch no :  9000   training loss:  0.010370756   validation loss:  0.016018813     minimum_validation_loss 0.013489423
epoch no :  9500   training loss:  0.011301058   validation loss:  0.016292341     minimum_validation_loss 0.013489423
epoch no :  10000   training loss:  0.011543658   validation loss:  0.01676923     minimum_validation_loss 0.013489423
epoch no :  10500   training loss:  0.015369514   validation loss:  0.020863634     minimum_validation_loss 0.013489423
epoch no :  11000   training loss:  0.013260584   validation loss:  0.019189946     minimum_validation_loss 0.013489423
epoch no :  11500   training loss:  0.010662286   validation loss:  0.016263     minimum_validation_loss 0.013489423
epoch no :  12000   training loss:  0.011147232   validation loss:  0.01676373     minimum_validation_loss 0.013489423
epoch no :  12500   training loss:  0.008560448   validation loss:  0.013835487     minimum_validation_loss 0.013489423
epoch no :  13000   training loss:  0.009443357   validation loss:  0.0148845045     minimum_validation_loss 0.013489423
epoch no :  13500   training loss:  0.009587998   validation loss:  0.015142298     minimum_validation_loss 0.013489423
epoch no :  14000   training loss:  0.011867486   validation loss:  0.017389275     minimum_validation_loss 0.013489423
epoch no :  14500   training loss:  0.008639899   validation loss:  0.013910311     minimum_validation_loss 0.013489423
epoch no :  15000   training loss:  0.009500151   validation loss:  0.014887231     minimum_validation_loss 0.013489423
Model saved in path: model/model.ckpt
INFO:tensorflow:Restoring parameters from model/model.ckpt
Model restored.
ann root mean absolute error:  0.11614399988937986
accuracy score:  0.9107539750018678
```



If we zoom into the curve we would have seen the following



In [211]:
```
plot_prediction( y_val, pred_df['ANN_lr0.1_beta0-None-None-None_hidden2-None-None-None'])
```



Description on Learning curve and Accuracy:

We can observe where overfitting occurs. Overfitting actually occurs if the training loss goes under the validation loss even though the validation is still dropping. It is the sign that network is learning the patterns in the train set that are not applicable in the validation done. In a short note we can say::

Overfitting : training loss << validation loss

Underfitting : training loss >> validation loss

Just right : training loss ~ validation loss

According to this theory, for ANN 4 our both learning curve (validation loss and training loss) is exactly top of one another so in our case validation loss and training loss is almost same so we can say that our model is doing just the right thing. Again In validation score .1059 is impressive compared to other models.

But for ANN 5 and 6 training loss << validation loss so we can say that this two model overfit data due to lower amount of neuron but ANN 4 have just the right amount of neuron thats why with similar parameter this overfit occered.

Sometimes Both of the curve actually seems to be on top of each other.The reason is:

- I have applied log transformation on the SalePrice and I have also transformed all my numerical data thats why the difference between the training loss and validation loss seems to be very small and very stable.
- For loss function I have used Mean Squared Error (MSE). For reducing MSE I have used SUM_BY_NONZERO_WEIGHTS which divided scalar sum by number of non-zero weights. MSE calculates squared error for all the data and then calculate the mean. Now, all my SalePrice is very small due to normalization (between 10 to 13.5). Where mean of saleprice is 12.02 . Suppose in nth epoch if
  - for training loss
    - a saleprice is 11.5 and prediction is 12 Squred error .25
    - another saleprice is 13 and prediction is 12 Squred error 1
    - another saleprice is 12.5 and prediction is 12 Squred error .25
    - another saleprice is 11.5 and prediction is 12 Squred error .25
    - another saleprice is 10.3 and prediction is 12 Squred error 1.7

    MSE = (.25+1+.25+.25+1.7)/5 = .69
  - for validation loss
    - another saleprice is 12.9 and prediction is 12 Squred error .81
    - another saleprice is 13.3 and prediction is 12 Squred error 1.69
    - another saleprice is 10.8 and prediction is 12 Squred error 1.44
    - another saleprice is 11.3 and prediction is 12 Squred error .49
    - another saleprice is 11.8 and prediction is 12 Squred error .04

    MSE = (.81+1.69+1.44+.49+.04)/5 = .894

    Difference between validation loss and training loss is .204

    Usually in regression problem neural network stats to predicts the average value within 5-20 epoch so very quickly the difference between val_loss and training_loss gets much lower. In our dummy example difference is already .204 and if its epoch no is 10, by the time it reaches to 500 epoch the difference could go as low as 10^-4.

Hyperparameeter tuning

Few of my hyperparameeter tuning is shown in the following block. In this data if a hidden layer value is 0 then it means that the hidden layer is turned off. For example if hidden_3 = 0 then that means hidden layer 3 is removed from the model and the model have only 2 hidden layer. And all the score is done on a validation set which is not seen by the model while training. For most of the case it was a 80-20 split. In the following results I didint kept any cross validation results but I have used diffrent seed while splitting data due to diffrent seed sometimes good hyperparameeter also provided so so accuracy.

In [184]:

```
import pandas as pd
log_df = pd.read_csv("diffrent_training_results.csv")
# print(log_df.to_string())
pd.set_option('display.max_rows', None)
log_df
```

Out[184]:

	learning_rate	num_steps	beta1	beta2	beta3	hidden_1	hidden_2	hidden_3	input_dim	test_rmse_score	test_r2_score	hidden_4	beta4
0	0.050	7500.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.128456	8.949361e-01	NaN	NaN
1	0.040	2500.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.250470	6.005558e-01	NaN	NaN
2	0.050	7500.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.152580	8.517686e-01	NaN	NaN
3	0.050	7500.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.143409	8.690530e-01	NaN	NaN
4	0.050	7500.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.127356	8.967284e-01	NaN	NaN
5	0.050	7900.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.126758	8.976948e-01	NaN	NaN
6	0.050	1500.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.162495	8.318785e-01	NaN	NaN
7	0.050	1500.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.177628	7.991050e-01	NaN	NaN
8	0.050	7500.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.139909	8.753655e-01	NaN	NaN
9	0.050	7500.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.143775	8.683835e-01	NaN	NaN
10	0.100	7500.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.138477	8.779036e-01	NaN	NaN
11	0.100	7500.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.138477	8.779036e-01	NaN	NaN
12	0.010	7500.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.154219	8.485668e-01	NaN	NaN
13	0.010	7500.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.154219	8.485668e-01	NaN	NaN
14	0.010	7500.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.661086	-1.782662e+00	NaN	NaN
15	0.100	3500.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.131423	8.900259e-01	NaN	NaN
16	0.100	2800.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.389771	-5.111744e-03	NaN	NaN
17	0.100	2500.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.390269	-7.679426e-03	NaN	NaN
18	0.100	8500.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.102641	9.302995e-01	NaN	NaN
19	0.050	8500.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.135519	8.830642e-01	NaN	NaN
20	0.100	2500.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.304550	4.094407e-01	NaN	NaN
21	0.100	9500.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.124793	9.008422e-01	NaN	NaN
22	0.001	9500.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	14.246912	-1.291369e+03	NaN	NaN
23	0.005	9500.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.710445	-2.213707e+00	NaN	NaN
24	0.100	7500.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.129652	8.929701e-01	NaN	NaN
25	0.100	12500.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.142223	8.712101e-01	NaN	NaN
26	0.100	11500.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.126000	8.989146e-01	NaN	NaN
27	0.050	11500.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.124008	9.020864e-01	NaN	NaN
28	0.050	11000.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.147886	8.607480e-01	NaN	NaN
29	0.050	11000.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.105610	9.262080e-01	NaN	NaN
30	0.050	11000.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.131099	8.905686e-01	NaN	NaN
31	0.010	11000.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	2.365656	-3.463267e+01	NaN	NaN

	learning_rate	num_steps	beta1	beta2	beta3	hidden_1	hidden_2	hidden_3	input_dim	test_rmse_score	test_r2_score	hidden_4	beta4
32	0.100	11000.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.135869	8.824604e-01	NaN	NaN
33	0.100	11000.0	0.005	0.0050	0.000000	16.0	8.0	4.0	403.0	0.132422	8.883490e-01	NaN	NaN
34	0.050	13000.0	0.005	0.0050	0.000000	16.0	8.0	4.0	403.0	0.126124	8.987160e-01	NaN	NaN
35	0.100	23000.0	0.005	0.0050	0.000000	16.0	8.0	4.0	403.0	0.123945	9.021856e-01	NaN	NaN
36	0.100	23000.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.321211	3.430598e-01	NaN	NaN
37	0.050	23000.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.126705	8.977802e-01	NaN	NaN
38	0.050	23000.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.126705	8.977802e-01	NaN	NaN
39	0.050	23000.0	0.000	0.0000	0.000000	16.0	8.0	4.0	403.0	0.173484	8.083707e-01	NaN	NaN
40	0.050	23000.0	0.000	0.0000	0.000000	16.0	8.0	4.0	403.0	0.396362	-2.986747e-04	NaN	NaN
41	0.100	23000.0	0.000	0.0000	0.000000	16.0	8.0	4.0	403.0	0.396368	-3.297183e-04	NaN	NaN
42	0.100	17000.0	0.000	0.0000	0.000000	16.0	8.0	4.0	403.0	0.396362	-2.981759e-04	NaN	NaN
43	0.001	3000.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	2.654960	-4.388086e+01	NaN	NaN
44	0.001	3000.0	0.050	0.0000	0.000000	16.0	8.0	4.0	403.0	7.177147	-3.269811e+02	NaN	NaN
45	0.100	3000.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.124692	9.010024e-01	NaN	NaN
46	0.100	3000.0	0.010	0.0000	0.000000	16.0	8.0	4.0	403.0	0.126720	8.977569e-01	NaN	NaN
47	0.100	13000.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.396362	-2.986747e-04	NaN	NaN
48	0.100	2500.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.136049	8.821482e-01	NaN	NaN
49	0.100	3000.0	0.100	0.1000	0.000000	16.0	8.0	4.0	403.0	0.127360	8.967204e-01	NaN	NaN
50	0.100	4000.0	0.100	0.1000	0.000000	16.0	8.0	4.0	403.0	0.260455	5.680732e-01	NaN	NaN
51	0.100	3500.0	0.100	0.0100	0.000000	16.0	8.0	4.0	403.0	0.146195	8.639153e-01	NaN	NaN
52	0.100	3500.0	0.100	0.0010	0.000000	16.0	8.0	4.0	403.0	0.396370	-3.376305e-04	NaN	NaN
53	0.100	3500.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.113379	9.149525e-01	NaN	NaN
54	0.100	3500.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.117934	9.079823e-01	NaN	NaN
55	0.100	8500.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.111485	9.177711e-01	NaN	NaN
56	0.100	7600.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.106919	9.243687e-01	NaN	NaN
57	0.050	7600.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.119165	9.060501e-01	NaN	NaN
58	0.100	7600.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.376772	6.081402e-02	NaN	NaN
59	0.100	3500.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.107788	9.231341e-01	NaN	NaN
60	0.050	8500.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.122025	9.014877e-01	NaN	NaN
61	0.100	7600.0	0.005	0.0050	0.000000	16.0	8.0	4.0	403.0	0.389771	-5.109640e-03	NaN	NaN
62	0.100	7600.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.109705	9.203748e-01	NaN	NaN
63	0.100	7600.0	0.100	0.0050	0.005000	200.0	100.0	30.0	403.0	0.389676	-4.622793e-03	NaN	NaN

	learning_rate	num_steps	beta1	beta2	beta3	hidden_1	hidden_2	hidden_3	input_dim	test_rmse_score	test_r2_score	hidden_4	beta4
64	0.100	9600.0	0.000	0.0000	0.000000	200.0	100.0	30.0	403.0	0.389825	-5.391521e-03	NaN	NaN
65	0.100	3600.0	0.100	0.0000	0.000000	200.0	100.0	30.0	403.0	1.764792	-1.960547e+01	NaN	NaN
66	0.100	7600.0	0.100	0.0000	0.000000	200.0	100.0	30.0	403.0	0.389774	-5.127892e-03	NaN	NaN
67	0.100	17600.0	0.100	0.0000	0.000000	200.0	100.0	30.0	403.0	0.389772	-5.119113e-03	NaN	NaN
68	0.100	15600.0	0.100	0.0000	0.000000	200.0	100.0	30.0	403.0	0.149463	8.522031e-01	NaN	NaN
69	0.100	15600.0	0.100	0.0000	0.000000	32.0	16.0	8.0	403.0	0.389750	-5.006016e-03	NaN	NaN
70	0.100	3600.0	0.100	0.0000	0.000000	1.0	0.0	0.0	403.0	0.104633	9.275680e-01	NaN	NaN
71	0.100	7500.0	0.100	0.0000	0.000000	1.0	0.0	0.0	403.0	0.105433	9.264551e-01	NaN	NaN
72	0.100	7500.0	0.010	0.0000	0.000000	1.0	0.0	0.0	403.0	0.104463	9.278026e-01	NaN	NaN
73	0.100	6000.0	0.010	0.0000	0.000000	1.0	0.0	0.0	403.0	0.158149	8.345259e-01	NaN	NaN
74	0.100	86000.0	0.010	0.0000	0.000000	1.0	0.0	0.0	403.0	0.389882	-5.682449e-03	NaN	NaN
75	0.100	8600.0	0.010	0.0000	0.000000	1.0	0.0	0.0	403.0	0.389771	-5.109640e-03	NaN	NaN
76	0.100	8600.0	0.100	0.0000	0.000000	1.0	0.0	0.0	403.0	0.111076	9.183720e-01	NaN	NaN
77	0.100	3600.0	0.100	0.0000	0.000000	4.0	0.0	0.0	403.0	0.105270	9.266836e-01	NaN	NaN
78	0.100	3600.0	0.100	0.0000	0.000000	1.0	0.0	0.0	403.0	0.104898	9.271996e-01	NaN	NaN
79	0.100	3600.0	0.100	0.0000	0.000000	32.0	0.0	0.0	403.0	0.198056	7.404795e-01	NaN	NaN
80	0.100	3600.0	0.100	0.0000	0.000000	16.0	0.0	0.0	403.0	0.607723	-1.443464e+00	NaN	NaN
81	0.100	3600.0	0.100	0.0000	0.000000	16.0	0.0	0.0	403.0	0.106043	9.256030e-01	NaN	NaN
82	0.100	7600.0	0.100	0.0000	0.000000	1.0	0.0	0.0	403.0	0.105914	9.257826e-01	NaN	NaN
83	0.100	7600.0	0.100	0.0000	0.000000	16.0	0.0	0.0	403.0	0.107050	9.241823e-01	NaN	NaN
84	0.100	3900.0	0.100	0.0000	0.000000	16.0	0.0	0.0	403.0	0.107679	9.232893e-01	NaN	NaN
85	0.100	2000.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.157928	8.411961e-01	NaN	NaN
86	0.100	7600.0	0.100	0.0050	0.005000	16.0	8.0	4.0	403.0	0.138570	8.777409e-01	NaN	NaN
87	0.100	8600.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.133445	8.866164e-01	NaN	NaN
88	0.100	8600.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.129291	8.935660e-01	NaN	NaN
89	0.100	3600.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.119097	9.096871e-01	NaN	NaN
90	0.100	3500.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.512748	-6.739936e-01	NaN	NaN
91	0.100	9600.0	0.100	0.0050	0.005000	16.0	8.0	4.0	403.0	0.124896	9.006785e-01	NaN	NaN
92	0.100	9600.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.205646	7.307324e-01	NaN	NaN
93	0.100	19600.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.132143	8.888180e-01	NaN	NaN
94	0.100	19600.0	0.100	0.0100	0.001000	16.0	8.0	4.0	403.0	0.235628	6.464929e-01	NaN	NaN
95	0.100	19600.0	0.100	0.0005	0.000005	16.0	8.0	4.0	403.0	0.128857	8.942789e-01	NaN	NaN

	learning_rate	num_steps	beta1	beta2	beta3	hidden_1	hidden_2	hidden_3	input_dim	test_rmse_score	test_r2_score	hidden_4	beta4
96	0.100	19600.0	0.000	0.0000	0.000000	16.0	8.0	4.0	403.0	0.396391	-4.451594e-04	NaN	NaN
97	0.100	29600.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.148700	8.592123e-01	NaN	NaN
98	0.100	4000.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.172257	8.110704e-01	NaN	NaN
99	0.100	1750.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.396362	-2.980097e-04	NaN	NaN
100	0.100	3600.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.127960	8.957456e-01	NaN	NaN
101	0.100	4000.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.122812	9.039655e-01	NaN	NaN
102	0.100	7600.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.127266	8.968736e-01	NaN	NaN
103	0.100	5500.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.127831	8.959562e-01	NaN	NaN
104	0.100	7600.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	399.635052	-1.016885e+06	NaN	NaN
105	0.100	17600.0	0.100	0.0005	0.000005	16.0	8.0	4.0	403.0	0.122934	9.037754e-01	NaN	NaN
106	0.100	17100.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.123996	9.021051e-01	NaN	NaN
107	0.100	19500.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.122128	9.050330e-01	NaN	NaN
108	0.100	29500.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.121803	9.055369e-01	NaN	NaN
109	0.100	49500.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.120712	9.072223e-01	NaN	NaN
110	0.100	9500.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.104178	9.281964e-01	NaN	NaN
111	0.100	19500.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.089601	9.468848e-01	NaN	NaN
112	0.100	19500.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.118142	9.076568e-01	NaN	NaN
113	0.100	19500.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.118142	9.076568e-01	NaN	NaN
114	0.100	9500.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.090853	9.453897e-01	NaN	NaN
115	0.100	9500.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.090915	9.453158e-01	NaN	NaN
116	0.100	19500.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.091638	9.444420e-01	NaN	NaN
117	0.050	39500.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.076999	9.607748e-01	NaN	NaN
118	0.100	49500.0	0.005	0.0050	0.005000	16.0	8.0	4.0	403.0	0.073579	9.641818e-01	NaN	NaN
119	0.100	49500.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.087605	9.492243e-01	NaN	NaN
120	0.100	9500.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.159089	8.325543e-01	NaN	NaN
121	0.100	29500.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.093281	9.424324e-01	NaN	NaN
122	0.100	29500.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.127969	8.689350e-01	NaN	NaN
123	0.100	9500.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.117637	8.892446e-01	NaN	NaN
124	0.100	9500.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.104068	9.283477e-01	NaN	NaN
125	0.100	9500.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.103677	9.288849e-01	NaN	NaN
126	0.100	9500.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.103494	9.291363e-01	NaN	NaN
127	0.100	9500.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.103775	9.364546e-01	NaN	NaN

	learning_rate	num_steps	beta1	beta2	beta3	hidden_1	hidden_2	hidden_3	input_dim	test_rmse_score	test_r2_score	hidden_4	beta4
128	0.100	26000.0	0.100	0.0000	0.000000	200.0	100.0	50.0	403.0	0.112221	9.256909e-01	NaN	NaN
129	0.100	26000.0	0.100	0.0000	0.000000	200.0	100.0	50.0	403.0	0.111169	9.270777e-01	12.0	NaN
130	0.100	26000.0	0.100	0.0000	0.000000	200.0	100.0	50.0	403.0	0.111169	9.270777e-01	12.0	NaN
131	0.100	46000.0	0.100	0.0000	0.000000	200.0	100.0	50.0	403.0	0.411770	-4.727709e-04	12.0	NaN
132	0.100	26000.0	0.000	0.0000	0.000000	200.0	96.0	32.0	403.0	0.411734	-2.970530e-04	4.0	NaN
133	0.100	6000.0	0.000	0.0000	0.000000	200.0	96.0	32.0	403.0	0.412326	-3.176449e-03	4.0	0.0
134	0.100	26000.0	0.100	0.0000	0.000000	200.0	96.0	32.0	403.0	0.411762	-4.307332e-04	4.0	0.0
135	0.100	26000.0	0.100	0.0000	0.000000	32.0	16.0	8.0	403.0	0.411745	-3.482792e-04	4.0	0.0
136	0.100	46000.0	0.100	0.1000	0.000000	200.0	100.0	50.0	403.0	0.411750	-3.748811e-04	25.0	0.0
137	0.100	46000.0	0.100	0.1000	0.000000	200.0	100.0	50.0	403.0	0.411750	-3.748811e-04	25.0	0.0
138	0.100	26000.0	0.100	0.0000	0.000000	200.0	100.0	50.0	403.0	0.411739	-3.223233e-04	12.0	0.0
139	0.050	26000.0	0.100	0.0000	0.000000	200.0	100.0	50.0	403.0	0.412324	-3.166794e-03	12.0	0.0
140	0.100	15000.0	0.100	NaN	NaN	16.0	NaN	NaN	403.0	0.115072	9.218665e-01	NaN	NaN
141	0.100	15000.0	0.000	NaN	NaN	1.0	NaN	NaN	403.0	0.116185	9.203481e-01	NaN	NaN
142	0.100	15000.0	0.000	NaN	NaN	2.0	NaN	NaN	403.0	0.150994	8.654721e-01	NaN	NaN
143	0.100	15000.0	NaN	NaN	NaN	NaN	NaN	NaN	403.0	0.162648	8.439027e-01	NaN	NaN
144	0.100	35000.0	NaN	NaN	NaN	NaN	NaN	NaN	403.0	0.147050	8.724072e-01	NaN	NaN
145	0.100	8000.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.184813	7.984608e-01	NaN	NaN
146	0.100	20000.0	0.100	0.0500	0.000000	76.0	48.0	32.0	403.0	0.111441	9.267196e-01	16.0	0.0
147	0.100	25000.0	0.100	0.0500	0.000000	128.0	64.0	32.0	403.0	0.412339	-3.237055e-03	16.0	0.0
148	0.100	8000.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.111108	9.271576e-01	NaN	NaN
149	0.100	25000.0	0.100	0.0500	0.000000	128.0	64.0	32.0	403.0	0.150787	8.658403e-01	16.0	0.0
150	0.050	35000.0	0.100	0.0000	0.000000	128.0	64.0	32.0	403.0	0.412324	-3.166794e-03	16.0	0.0
151	0.100	25000.0	0.100	0.0000	0.000000	256.0	128.0	32.0	403.0	0.412324	-3.166794e-03	8.0	0.0
152	0.050	25000.0	0.100	0.0000	0.000000	256.0	128.0	32.0	403.0	0.412324	-3.167316e-03	8.0	0.0
153	0.100	15000.0	0.100	0.0000	0.000000	4.0	16.0	16.0	403.0	0.412324	-3.167055e-03	4.0	0.0
154	0.100	35000.0	0.100	0.0000	0.000000	256.0	128.0	64.0	403.0	0.412325	-3.170185e-03	32.0	0.0
155	0.100	25000.0	0.100	0.1000	0.000000	256.0	128.0	32.0	403.0	0.123334	9.102441e-01	8.0	0.0
156	0.050	25000.0	0.100	0.0000	0.000000	256.0	128.0	64.0	403.0	0.412324	-3.167055e-03	8.0	0.0
157	0.050	25000.0	0.100	0.0000	0.000000	128.0	64.0	16.0	403.0	0.412319	-3.141554e-03	4.0	0.0
158	0.100	35000.0	0.100	0.0000	0.000000	16.0	32.0	48.0	403.0	0.416205	-2.213767e-02	76.0	0.0
159	0.100	15000.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.412324	-3.166794e-03	2.0	0.0

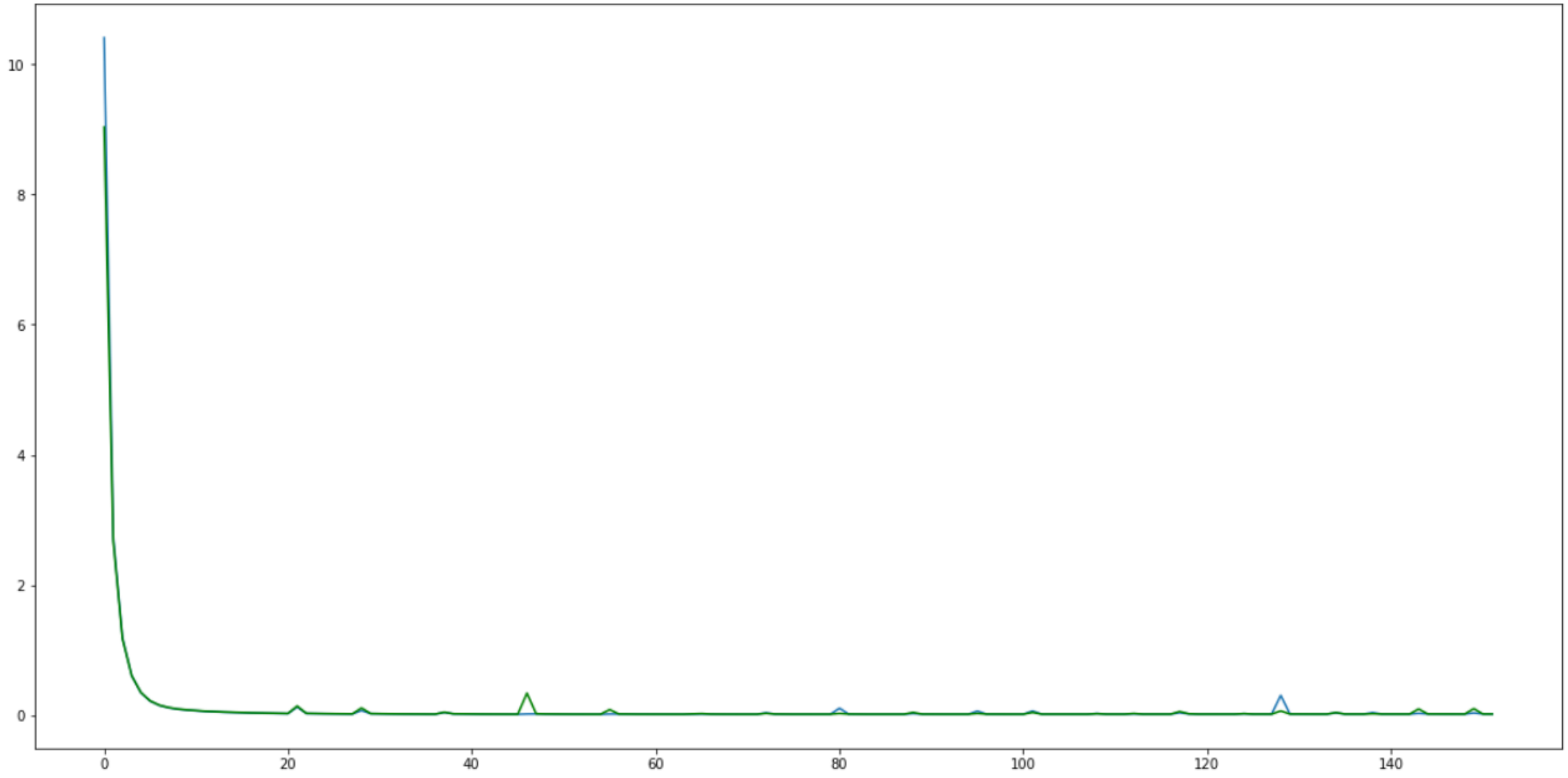
	learning_rate	num_steps	beta1	beta2	beta3	hidden_1	hidden_2	hidden_3	input_dim	test_rmse_score	test_r2_score	hidden_4	beta4
160	0.050	25000.0	0.100	0.0500	0.000000	128.0	64.0	16.0	403.0	0.412324	-3.167837e-03	4.0	0.0
161	0.050	25000.0	0.100	0.0000	0.000000	190.0	90.0	30.0	403.0	0.412324	-3.165491e-03	3.0	0.0
162	0.100	8000.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.412324	-3.166794e-03	2.0	0.0
163	0.100	6000.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.412324	-3.165230e-03	NaN	NaN
164	0.100	6000.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.105436	9.344041e-01	NaN	NaN
165	0.100	20000.0	0.100	0.0500	0.000000	76.0	48.0	32.0	403.0	0.108646	9.303500e-01	16.0	0.0
166	0.100	6000.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.103792	9.364344e-01	NaN	NaN
167	0.100	20000.0	0.100	0.0500	0.000000	76.0	48.0	32.0	403.0	0.106511	9.330597e-01	16.0	0.0
168	0.100	25000.0	0.100	0.0500	0.000000	128.0	64.0	32.0	403.0	0.411699	-1.253480e-04	16.0	0.0
169	0.100	25000.0	0.100	0.1000	0.000000	256.0	128.0	32.0	403.0	0.411756	-4.026479e-04	8.0	0.0
170	0.050	25000.0	0.100	0.0500	0.000000	256.0	128.0	16.0	403.0	0.112563	9.252368e-01	4.0	0.0
171	0.050	25000.0	0.100	0.0000	0.000000	190.0	90.0	30.0	403.0	0.411974	-1.465819e-03	3.0	0.0
172	0.100	8000.0	0.100	0.0000	0.000000	16.0	8.0	4.0	403.0	0.103529	9.367565e-01	2.0	0.0
173	0.100	15000.0	0.100	NaN	NaN	16.0	NaN	NaN	403.0	0.106009	9.336896e-01	NaN	NaN
174	0.100	15000.0	0.000	NaN	NaN	1.0	NaN	NaN	403.0	0.120667	9.140850e-01	NaN	NaN
175	0.100	15000.0	0.000	NaN	NaN	2.0	NaN	NaN	403.0	0.116812	9.194859e-01	NaN	NaN
176	0.100	35000.0	NaN	NaN	NaN	NaN	NaN	NaN	403.0	0.138819	8.862908e-01	NaN	NaN
177	0.050	25000.0	0.100	0.0100	0.000000	180.0	90.0	30.0	403.0	0.411965	-1.419538e-03	6.0	0.0
178	0.100	20000.0	0.100	0.0500	0.000000	76.0	48.0	32.0	403.0	0.110364	9.281301e-01	16.0	0.0

Observation and discovery :

- In the above parameter we can see that index 44 shows that for .001 learning parameter the model does not predict anything so I have changed it slowly and finally What I have found that learning parameter .1 and .05 provides the best results.
- Beta1, Beta2, Beta3, Beta4 represents the regularization parameter for hidden layer 1 ,2 ,3 and 4. Sometimes in the above table we can see that hidden layer 2,3,4 is 0 or NaN but there is some value for beta 2,3,4 that means the layer is actually off so those values actually means nothing.
- For 3 layer model when beta1, beta2, beta3 is .005, model shows significant amount of improvement while learning rate is .1 or .05 . But when learning rate is .1 and beta1=.1 , beta2=0, beta3=0 then the model performs even better most of the time and it also takes less epochs to train for the best validation accuracy
- From index 63 to 69 I have tried to use 200 , 100 , 30 neurons because the data have 403 features and its a common practice to use half amount of the neuron in the first hidden layer and this strategy does not work good enough but with my selected parameter it improved a little bit. I have used 16-8-4 combination of neuron because of this common practice. for our case 16 neuron in the first layer provided better accuracy and adding 8 and 4 in the next 2 layer improved the stability of the model and now it gives good validation accuracy after 2000 epoch and the best validation accuracy remains between the epoch range of 2000-2500 , 3300-3600 or 5000-5400 .
- From index 70 to 78 we can see that single neuron with single hidden layer performs well according to the plan stated in the target section. Then I have increased neurons and the learning curve for them is in the following block. Where y axis shows rmse and x axis shows i and i*50 represents the epoch no. Again blue curve is for training accuracy and green for validation accuracy
- In the table index 169 and 155 the model is exactly same with same parameter but one of them is providing .123 and other is providing .41 and that shows how inconsistent model become when we increase the neuron of the first hidden layer.
- We can see that even after adding another layer ANN does not perform well when we are increasing neurons in the first layer. The reason behind it is that this type of regression problem usually do well with logistic regression. By increasing neurons we cant do much improvement and all we need to do is properly regularize small amount of neurons so that they can perform well.

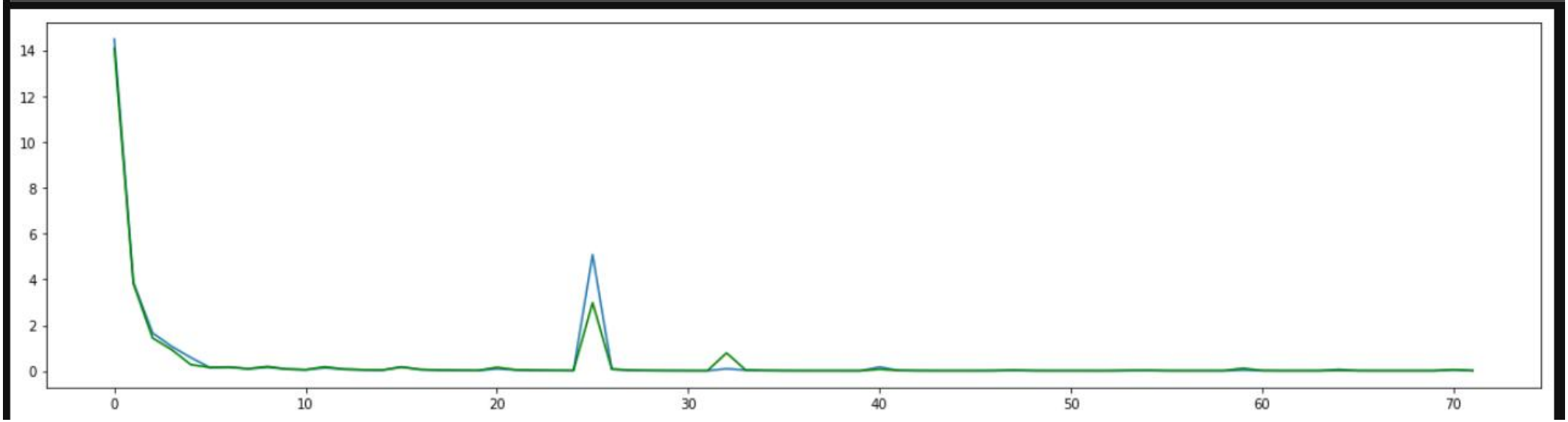
For single neuron learning rate

-



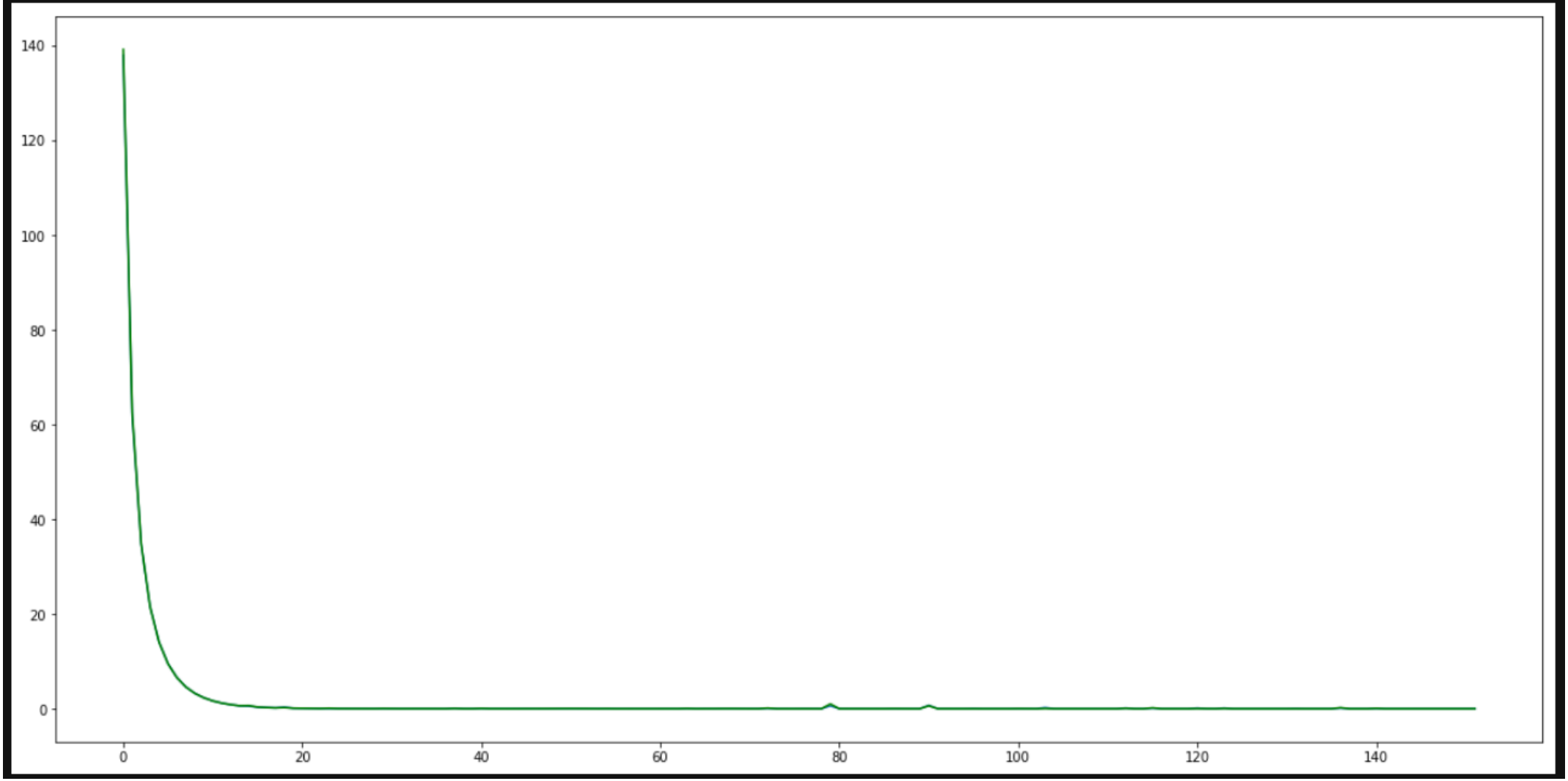
For 4 neuron learning curve

-



For 16 neuron learning curve

-



For 32 neuron learning curve

-

Ensemble

I am using bagging method for this section. Usually in this technique we add different models results and average them. But instead of averaging I am taking different fraction from different models result. Finally making sure that it sums up to 1.

I have tried different combinations of ensemble learning to improve performance. Kaggle has a certain limitation on uploading submission files. So what I have tried is that before submitting it to kaggle, I have made 80-20 split. I made prediction on the 20% data. Then I have tried ensemble learning so that before submission I can confirm which combination might work well.

Following 3 section arranges diffrent prediction results for ensembling.

In [185]:
```
x = ['Random Forest Regressor',
     'DecisionTree','Xgboost','Lasso',
     'ANN_base_lr0.1_beta0.1-0.0-0.0-None_hidden16-8-4-None',
     'ANN_lr0.1_beta0.1-0.05-0.0-0.0_hidden76-48-32-16',
     'ANN_lr0.05_beta0.005-0.1-0.05-0.0_hidden8-32-16-8',
     'ANN_lr0.05_beta0.1-0.0-0.0-0.0_hidden16-8-4-2',
     'ANN_lr0.1_beta0.1-None-None-None_hidden16-None-None-None',
     'ANN_lr0.1_beta0-None-None-None_hidden4-None-None-None',
     'ANN_lr0.1_beta0-None-None-None_hidden2-None-None-None']
```

In [186]:
```
d = dict()
for k in x:
    if not submit:
        d[k] = prediction_dict[k]
    if submit:
        d[k] = submit_prediction_dict[k]
if not submit:
    prediction_dict = d
if submit:
    submit_prediction_dict = d
```

In [187]:
```
if submit :
    pred_df = pd.read_csv("diffrent_pred_results.csv")
else:
    pred_df = pd.read_csv("pred_results.csv")


if not submit:
    pd.set_option('display.max_colwidth', -1)
    pred_df = pd.DataFrame(prediction_dict)
    pred_df.to_csv("pred_results.csv", encoding='utf-8',index=False)

else:
    pd.set_option('display.max_colwidth', -1)
    pred_df = pd.DataFrame(submit_prediction_dict)
    pred_df.to_csv("diffrent_pred_results.csv", encoding='utf-8',index=False)

pd.DataFrame(pred_df.columns)
```

Out[187]:

	0
0	Random Forest Regressor
1	DecisionTree
2	Xgboost
3	Lasso
4	ANN_base_lr0.1_beta0.1-0.0-0.0-None_hidden16-8-4-None
5	ANN_lr0.1_beta0.1-0.05-0.0-0.0_hidden76-48-32-16
6	ANN_lr0.05_beta0.005-0.1-0.05-0.0_hidden8-32-16-8
7	ANN_lr0.05_beta0.1-0.0-0.0-0.0_hidden16-8-4-2
8	ANN_lr0.1_beta0.1-None-None-None_hidden16-None-None-None
9	ANN_lr0.1_beta0-None-None-None_hidden4-None-None-None
10	ANN_lr0.1_beta0-None-None-None_hidden2-None-None-None

Naming explanation of above table

Name	learning rate	beta1	beta 2	beta 3	beta 4	hidden layer 1	hidden layer 2	hidden layer 3	hidden layer 4
ANN_base_lr0.1_beta0.1-0.0-0.0-None_hidden16-8-4-None	0.1	0.1	0.0	0.0	None	16	8	4	None
ANN_lr0.05_beta0.005-0.1-0.05-0.0_hidden8-32-16-8	0.05	0.005	0.1	0.05	0	8	32	16	6
ANN_lr0.05_beta0.1-0.0-0.0-0.0_hidden16-8-4-2	.05	0.1	0.0	0.0	0.0	16	8	4	2
ANN_lr0.1_beta0-None-None-None_hidden2-None-None-None	0.1	0	None	None	None	2	None	None	None
ANN_lr0.1_beta0.1-0.05-0.0-0.0_hidden76-48-32-16	0.1	.1	0.05	0.0	0.0	76	48	32	16
ANN_lr0.1_beta0.1-None-None-None_hidden16-None-None-None	0.1	0.1	None	None	None	16	None	None	None
ANN_lr0.1_beta0-None-None-None_hidden4-None-None-None	0.1	0	None	None	None	4	None	None	None

Ensemble Combination 1

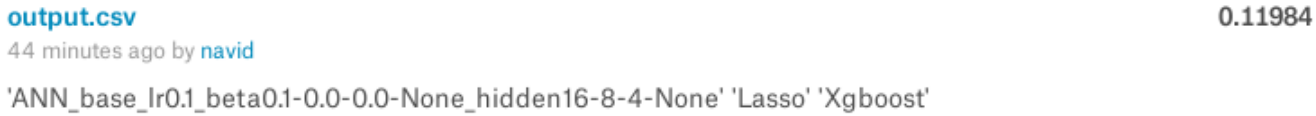
In [188]:
```
# pred_df[pred_df.columns[[1,3,5]]] * [1,2,30]

print('Using  ' , pred_df.columns[[4,3,2]].values)
```

```
Using   ['ANN_base_lr0.1_beta0.1-0.0-0.0-None_hidden16-8-4-None' 'Lasso' 'Xgboost']
```

In [189]:
```
prediction = pred_df[pred_df.columns[[4,3,2]]] * [.4,.2,.4]
prediction = prediction.sum(axis = 1)

if not submit:
    test_rmse_score, test_r2_score = accuracy(y_val, prediction)

    print('ann root mean absolute error: ', test_rmse_score)
    print('accuracy score: ', test_r2_score  )
```

```
ann root mean absolute error:  0.09971740714812104
accuracy score:  0.9342134245735598
```

Kaggle score

- 

In [190]:
```
prediction = pred_df[pred_df.columns[[4,3,2]]] * [.4,.3,.3]
prediction = prediction.sum(axis = 1)

if not submit:
    test_rmse_score, test_r2_score = accuracy(y_val, prediction)

    print('ann root mean absolute error: ', test_rmse_score)
    print('accuracy score: ', test_r2_score  )
```

```
ann root mean absolute error:  0.09979859123790498
accuracy score:  0.9341062617924164
```

Ensemble Combination 2

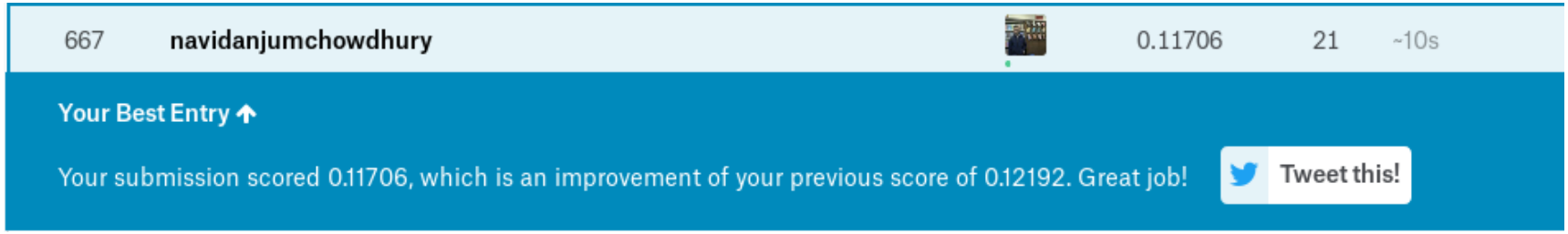
In [191]:
```
# pred_df[pred_df.columns[[1,3,5]]] * [1,2,30]

print('Using  ' , pred_df.columns[[2,4,5,6,7]].values)
```

```
Using   ['Xgboost' 'ANN_base_lr0.1_beta0.1-0.0-0.0-None_hidden16-8-4-None'
 'ANN_lr0.1_beta0.1-0.05-0.0-0.0_hidden76-48-32-16'
 'ANN_lr0.05_beta0.005-0.1-0.05-0.0_hidden8-32-16-8'
 'ANN_lr0.05_beta0.1-0.0-0.0-0.0_hidden16-8-4-2']
```

In [192]:
```
prediction = pred_df[pred_df.columns[[2,4,5,6,7]]] * [.25,.2,.2 ,.15 , .2]
prediction = prediction.sum(axis = 1)

if not submit:
    test_rmse_score, test_r2_score = accuracy(y_val, prediction)

    print('ann root mean absolute error: ', test_rmse_score)
    print('accuracy score: ', test_r2_score  )
```

```
ann root mean absolute error:  0.10113887360156766
accuracy score:  0.9323244880849749
```

Kaggle score

- 

Ensemble Combination 3

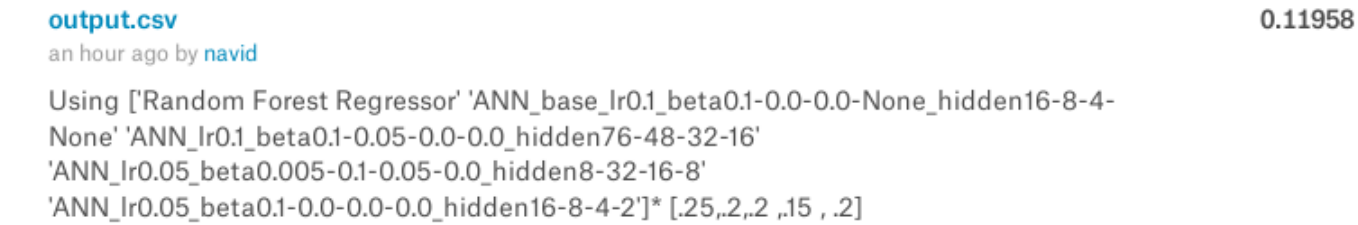
In [193]:
```
print('Using  ' , pred_df.columns[[0,4,5,6,7]].values)
```

```
Using   ['Random Forest Regressor'
 'ANN_base_lr0.1_beta0.1-0.0-0.0-None_hidden16-8-4-None'
 'ANN_lr0.1_beta0.1-0.05-0.0-0.0_hidden76-48-32-16'
 'ANN_lr0.05_beta0.005-0.1-0.05-0.0_hidden8-32-16-8'
 'ANN_lr0.05_beta0.1-0.0-0.0-0.0-0.0_hidden16-8-4-2']
```

In [194]:
```
prediction = pred_df[pred_df.columns[[0,4,5,6,7]]] * [.25,.2,.2 ,.15 , .2]
prediction = prediction.sum(axis = 1)

if not submit:
    test_rmse_score, test_r2_score = accuracy(y_val, prediction)

    print('ann root mean absolute error: ', test_rmse_score)
    print('accuracy score: ', test_r2_score  )
```

```
ann root mean absolute error:  0.10200280259637177
accuracy score:  0.9311633806499339
```

Kaggle score

- 

Ensemble Combination 4

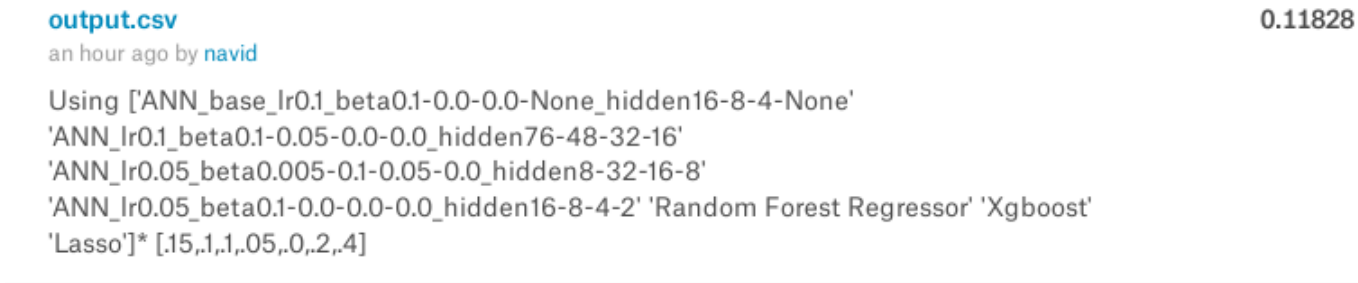
In [195]:
```
print('Using  ' , pred_df.columns[[4,5,6,7,0,2,3]].values)
```

```
Using   ['ANN_base_lr0.1_beta0.1-0.0-0.0-None_hidden16-8-4-None'
 'ANN_lr0.1_beta0.1-0.05-0.0-0.0_hidden76-48-32-16'
 'ANN_lr0.05_beta0.005-0.1-0.05-0.0_hidden8-32-16-8'
 'ANN_lr0.05_beta0.1-0.0-0.0-0.0_hidden16-8-4-2' 'Random Forest Regressor'
 'Xgboost' 'Lasso']
```

In [196]:
```
prediction = pred_df[pred_df.columns[[4,5,6,7,0,2,3]]] * [.15,.1,.1,.05,.0,.2,.4]
prediction = prediction.sum(axis = 1)

if not submit:
    test_rmse_score, test_r2_score = accuracy(y_val, prediction)

    print('ann root mean absolute error: ', test_rmse_score)
    print('accuracy score: ', test_r2_score  )
```

```
ann root mean absolute error:  0.10109235671145589
accuracy score:  0.9323867258832664
```

Kaggle score

- 

Ensemble combination 4 provides the score 0.12192. Currently combination 4 is showing that rmse value is .1004 and there is a worse value present in combination 1 which is 0.1007. The reason behind the difference is ANN does not perform exactly same each time. That means if I currently submit with Combination 1 I might get similar result.

In Combination 4 used models with their parameters:

Name	learning rate	beta1	beta 2	beta 3	beta 4	hidden layer 1	hidden layer 2	hidden layer 3	hidden layer 4	Fraction taken
ANN_base_lr0.1_beta0.1-0.0-0.0-None_hidden16-8-4-None	0.1	0.1	0.0	0.0	None	16	8	4	None	.15
ANN_lr0.05_beta0.005-0.1-0.05-0.0_hidden8-32-16-8	0.05	0.005	0.1	0.05	0	8	32	16	6	.1
ANN_lr0.05_beta0.1-0.0-0.0-0.0_hidden16-8-4-2	.05	0.1	0.0	0.0	0.0	16	8	4	2	.05
ANN_lr0.1_beta0.1-0.05-0.0-0.0_hidden76-48-32-16	0.1	.1	0.05	0.0	0.0	76	48	32	16	.1
Xgboost	0.05	Not applicable	Not applicable	Not applicable	Not applicable	Not applicable	Not applicable	NonNot applicablee	Not applicable	.2
Lasso	alpha = 5e-4	Not applicable	Not applicable	Not applicable	Not applicable	Not applicable	Not applicable	NonNot applicablee	Not applicable	.4

Obesrvation

In the learning curve graph if the minimum of training and validation is close to each other then its good to use that model. Again if training minimum and validation minimum is no where near each other then using them does not help much most of the case. When both of them are close we can use the epoch no of the train_min loss as val_min loss epoch no and then we can train over all the dataset without depending on the epoch number. The model does not give same result in same epoch every time. This is the main reason behind removing the epoch dependency.

Prepare Submission File

To use this section please uncomment the last line of split data section and comment accuracy section.

In [197]:
```
pd.DataFrame(pred_df.columns)
```

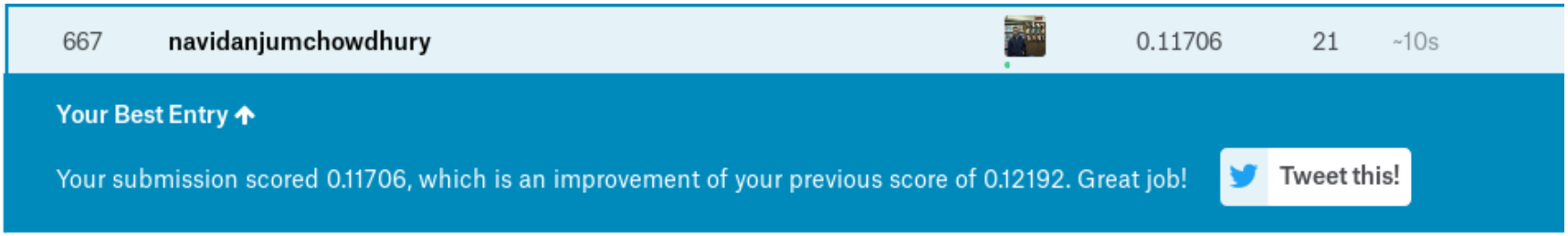
Out[197]:

	0
0	Random Forest Regressor
1	DecisionTree
2	Xgboost
3	Lasso
4	ANN_base_lr0.1_beta0.1-0.0-0.0-None_hidden16-8-4-None
5	ANN_lr0.1_beta0.1-0.05-0.0-0.0_hidden76-48-32-16
6	ANN_lr0.05_beta0.005-0.1-0.05-0.0_hidden8-32-16-8
7	ANN_lr0.05_beta0.1-0.0-0.0-0.0_hidden16-8-4-2
8	ANN_lr0.1_beta0.1-None-None-None_hidden16-None-None-None
9	ANN_lr0.1_beta0-None-None-None_hidden4-None-None-None
10	ANN_lr0.1_beta0-None-None-None_hidden2-None-None-None

In [ ]:
```
use_ensemble = True
#if ensemble =false then chose a model
choose_model = 6
#if want to use given test data
if submit:
#     X_val = test_processed
    if not use_ensemble:
        prediction = pred_df[pred_df.columns[[choose_model]]]

    prediction = np.exp(prediction.values)

    pred_out_df = pd.DataFrame(prediction, index=test["Id"], columns=["SalePrice"])
    pred_out_df.to_csv('output.csv', header=True, index_label='Id')
```

Conclusion & Kaggle score Discussion:

My target of this report was to improve ANN model and show how well it can perform with ANN model. In the beginning of the report I have build a ANN model that performs better or similar to other ANN model I have showed in my report. I have performed cross validation on that model and that model scored 0.11912 in kaggle. Then I have showed some other models that performs well but can't beat the score 0.11912 . Then I have explained why some models with certain parameter works well. After that I showed a table where different models performance is listed and added my analysis and observation. Then I have have Showed four combination of Ensemble and their kaggle score is also attached with them. In the 2nd combination of Ensemble method I have found the best kaggle score which is 0.11706. This score is achived through combining 4 ann models and xgboost. I have used the first 4 ANN models for this Ensemble.

- 

Reference

xgboost:

https://www.kaggle.com/dansbecker/xgboost (https://www.kaggle.com/dansbecker/xgboost)

https://medium.com/@gabrieltseng/gradient-boosting-and-xgboost-c306c1bcfaf5 (https://medium.com/@gabrieltseng/gradient-boosting-and-xgboost-c306c1bcfaf5)

regression + graph :

https://www.kaggle.com/janiobachmann/predicting-house-prices-regression-techniques (https://www.kaggle.com/janiobachmann/predicting-house-prices-regression-techniques)

Selecting and Filtering Data

https://www.kaggle.com/dansbecker/selecting-and-filtering-in-pandas (https://www.kaggle.com/dansbecker/selecting-and-filtering-in-pandas)

Handling Missing Values

https://www.kaggle.com/dansbecker/handling-missing-values (https://www.kaggle.com/dansbecker/handling-missing-values)

why use conditional probability coding

https://medium.com/airbnb-engineering/designing-machine-learning-models-7d0048249e69 (https://medium.com/airbnb-engineering/designing-machine-learning-models-7d0048249e69)

one hot encoding

https://hackernoon.com/what-is-one-hot-encoding-why-and-when-do-you-have-to-use-it-e3c6186d008f (https://hackernoon.com/what-is-one-hot-encoding-why-and-when-do-you-have-to-use-it-e3c6186d008f)

https://medium.com/@rajatgupta310198/getting-started-with-neural-network-for-regression-and-tensorflow-58ad3bd75223 (https://medium.com/@rajatgupta310198/getting-started-with-neural-network-for-regression-and-tensorflow-58ad3bd75223)

class example

https://colab.research.google.com/drive/1MExQ52bvHSPaUrGe8RvHZifvE6K6a0qh?fbclid=IwAR2EUWi4q6_q0mFbXQwGh4GNgB2Ex_WpP3K0L12182PdzszWSsEfzHf0REo#forceEdit=true&offline=true&sandboxMode=true&scrollTo=-Rh3-Vt9Nev9 (https://colab.research.google.com/drive/1MExQ52bvHSPaUrGe8RvHZifvE6K6a0qh?fbclid=IwAR2EUWi4q6_q0mFbXQwGh4GNgB2Ex_WpP3K0L12182PdzszWSsEfzHf0REo#forceEdit=true&offline=true&sandboxMode=true&scrollTo=-Rh3-Vt9Nev9)

Why cross validation

https://towardsdatascience.com/5-reasons-why-you-should-use-cross-validation-in-your-data-science-project-8163311a1e79 (https://towardsdatascience.com/5-reasons-why-you-should-use-cross-validation-in-your-data-science-project-8163311a1e79)

Standardize or Normalize?

https://medium.com/@rrfd/standardize-or-normalize-examples-in-python-e3f174b65dfc (https://medium.com/@rrfd/standardize-or-normalize-examples-in-python-e3f174b65dfc)

Decision Tree - Regression

https://www.saedsayad.com/decision_tree_reg.htm (https://www.saedsayad.com/decision_tree_reg.htm)

Some more

https://www.kaggle.com/klyusba/house-prices-advanced-regression-techniques/lasso-model-for-regression-problem/notebook (https://www.kaggle.com/klyusba/house-prices-advanced-regression-techniques/lasso-model-for-regression-problem/notebook)

https://www.kaggle.com/juliencs/house-prices-advanced-regression-techniques/a-study-on-regression-applied-to-the-ames-dataset/ (https://www.kaggle.com/juliencs/house-prices-advanced-regression-techniques/a-study-on-regression-applied-to-the-ames-dataset/)

https://www.kaggle.com/apapiu/house-prices-advanced-regression-techniques/regularized-linear-models (https://www.kaggle.com/apapiu/house-prices-advanced-regression-techniques/regularized-linear-models)

https://www.kaggle.com/juliencs/a-study-on-regression-applied-to-the-ames-dataset (https://www.kaggle.com/juliencs/a-study-on-regression-applied-to-the-ames-dataset)

For descriptive section

I have inspired form Ian Goodfellows book and used his way of explanation to explain my choice. His book can be found here: https://www.deeplearningbook.org/ (https://www.deeplearningbook.org/)

I have also followed data flatter for definition and their lessons can be found here: https://data-flair.training/blogs/neural-network-for-machine-learning/ (https://data-flair.training/blogs/neural-network-for-machine-learning/)