Advanced Regression Assignment

PART-1 : Importing Data and Taking glimpse at it

In [1]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import Ridge
from sklearn.linear_model import Lasso
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LinearRegression
from sklearn.feature_selection import RFE
from sklearn.model_selection import KFold
from sklearn.metrics import r2_score

pd.set_option('display.max_columns', 500)
```

In [2]:

```
df = pd.read_csv('train.csv')
df.head()
```

Out[2]:

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Ut
0	1	60	RL	65.0	8450	Pave	NaN	Reg	Lvl	/
1	2	20	RL	80.0	9600	Pave	NaN	Reg	Lvl	/
2	3	60	RL	68.0	11250	Pave	NaN	IR1	Lvl	/
3	4	70	RL	60.0	9550	Pave	NaN	IR1	Lvl	/
4	5	60	RL	84.0	14260	Pave	NaN	IR1	Lvl	/

In [ ]:

```
```

In [3]:

```
df.shape
```

Out[3]:

```
(1460, 81)
```

In [4]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1460 entries, 0 to 1459
Data columns (total 81 columns):
Id                1460 non-null int64
MSSubClass        1460 non-null int64
MSZoning          1460 non-null object
LotFrontage       1201 non-null float64
LotArea           1460 non-null int64
Street            1460 non-null object
Alley             91 non-null object
LotShape          1460 non-null object
LandContour       1460 non-null object
Utilities         1460 non-null object
LotConfig         1460 non-null object
LandSlope         1460 non-null object
Neighborhood      1460 non-null object
Condition1        1460 non-null object
Condition2        1460 non-null object
BldgType          1460 non-null object
HouseStyle        1460 non-null object
```

There are missing values in the dataset

In [5]:

```
df.describe()
```

Out[5]:

	Id	MSSubClass	LotFrontage	LotArea	OverallQual	OverallCond	Ye
count	1460.000000	1460.000000	1201.000000	1460.000000	1460.000000	1460.000000	1460.(
mean	730.500000	56.897260	70.049958	10516.828082	6.099315	5.575342	1971.:
std	421.610009	42.300571	24.284752	9981.264932	1.382997	1.112799	30.:
min	1.000000	20.000000	21.000000	1300.000000	1.000000	1.000000	1872.(
25%	365.750000	20.000000	59.000000	7553.500000	5.000000	5.000000	1954.(
50%	730.500000	50.000000	69.000000	9478.500000	6.000000	5.000000	1973.(
75%	1095.250000	70.000000	80.000000	11601.500000	7.000000	6.000000	2000.(
max	1460.000000	190.000000	313.000000	215245.000000	10.000000	9.000000	2010.(

It looks like there are a few outliers, we will look at those in part-2

PART - 2: Understanding and Cleaning Data

Treating missing values

Taking a look at the percentage amount of missing data in every column

In [6]:

```
percent_missing = df.isnull().sum() * 100 / len(df)
missing_value_df = pd.DataFrame({'column_name': df.columns,
                                 'percent_missing': percent_missing})
missing_value_df.sort_values('percent_missing', inplace=True,ascending=False)
missing_value_df
```

GarageQual	GarageQual	5.547945
BsmtFinType2	BsmtFinType2	2.602740
BsmtExposure	BsmtExposure	2.602740
BsmtQual	BsmtQual	2.534247
BsmtCond	BsmtCond	2.534247
BsmtFinType1	BsmtFinType1	2.534247
MasVnrArea	MasVnrArea	0.547945
MasVnrType	MasVnrType	0.547945
Electrical	Electrical	0.068493
Id	Id	0.000000
Functional	Functional	0.000000
Fireplaces	Fireplaces	0.000000
KitchenQual	KitchenQual	0.000000

Data is very less in this case so we cannot afford to lose a lot of rows therefore we are taking the removal `threshold to 5%` for column removal and less than that for row removal. If in the end we don't get a good model then we will try to tweek these values

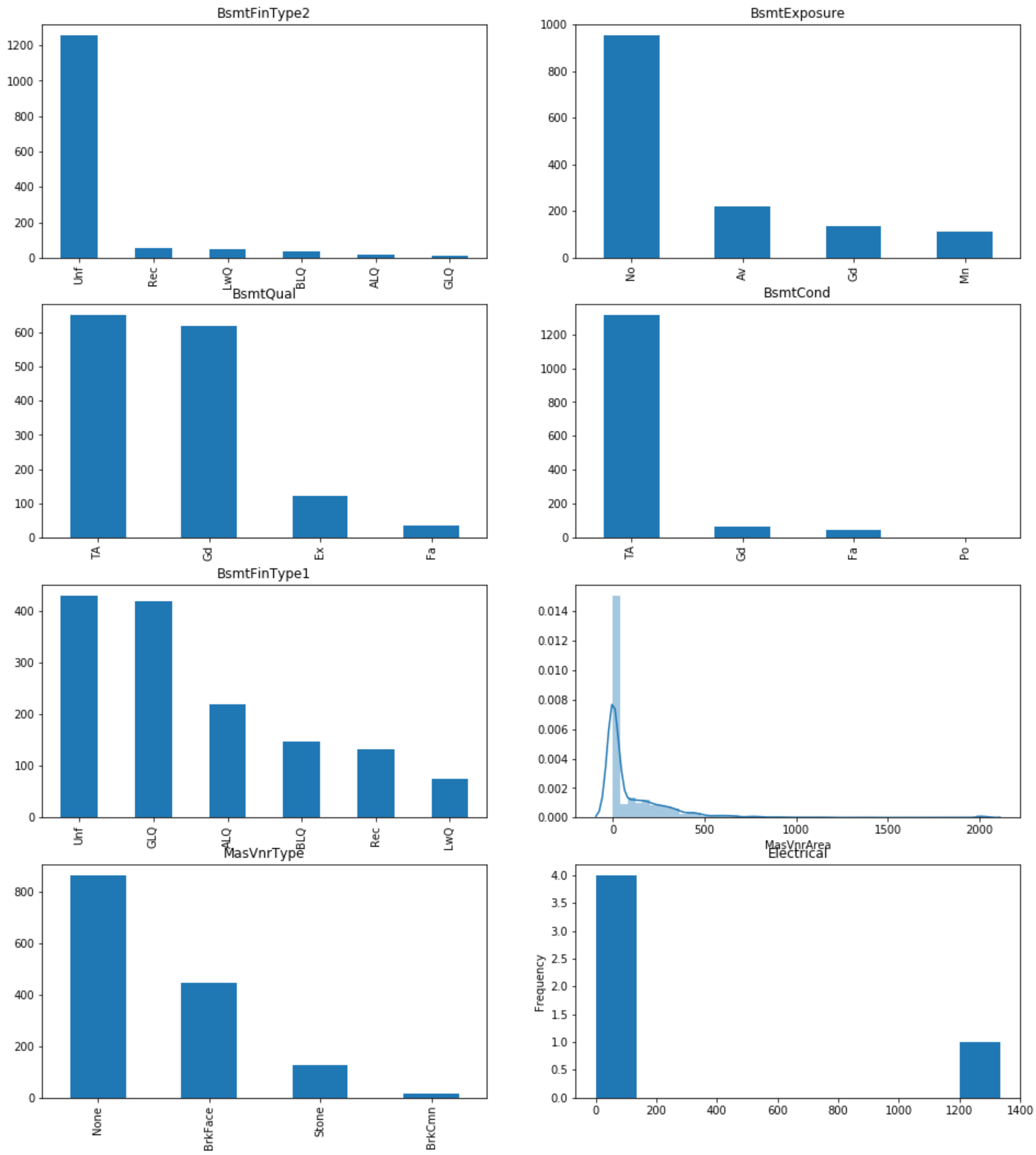
Let's visualize columns whose missing percentage is below 5%

In [7]:

```
fig, axes = plt.subplots(nrows=4, ncols=2,figsize=(16,18))
df['BsmtFinType2'].value_counts().plot.bar(ax=axes[0,0],title='BsmtFinType2')
df['BsmtExposure'].value_counts().plot.bar(ax=axes[0,1],title='BsmtExposure')
df['BsmtQual'].value_counts().plot.bar(ax=axes[1,0],title='BsmtQual')
df['BsmtCond'].value_counts().plot.bar(ax=axes[1,1],title='BsmtCond')
df['BsmtFinType1'].value_counts().plot.bar(ax=axes[2,0],title='BsmtFinType1')
sns.distplot(df['MasVnrArea'].fillna(2021),ax=axes[2,1])
#df['MasVnrArea'].value_counts().plot.hist(ax=axes[2,1],title='MasVnrArea')
df['MasVnrType'].value_counts().plot.bar(ax=axes[3,0],title='MasVnrType')
df['Electrical'].value_counts().plot.hist(ax=axes[3,1],title='Electrical')
plt.plot()
```

Out[7]:

```
[]
```



As we can see that columns `BsmtFinType2` , `BsmtExposure` , `BsmtCond` , `Electrical` are having very high percentage of one particular value in each. column `MasVnrType` also have a mean value which is quite steep in nature. So we will fill categorical columns `BsmtFinType2` , `BsmtExposure` , `BsmtCond` , `Electrical` with ***mode.*** And numerical column `MasVnrArea` with ***mean value***

But for the columns `BsmtQual` , `BsmtFinType1` , `MasVnrType` , the data is spread in considerable amount in all the classes. Randomly adding values may lead to false data insertion. For these column the probability of data to falsify is very high.

BsmtQual : 2.534247 BsmtFinType1 : 2.534247 MasVnrType : 0.547945

Since we don't have enough data to train on we will drop columns `BsmtQual` and `BsmtFinType1` And we will delete the null value rows for `MasVnrType` If in the end we don't get a good model then we will try to tweek these values

In [8]:

```
column_mode =['BsmtFinType2', 'BsmtExposure', 'BsmtCond', 'Electrical']
column_mean = ['MasVnrArea']
column_remove = ['BsmtQual','BsmtFinType1']
```

In [9]:

```
for i in column_mode:
    df[i].fillna(df[i].mode()[0], inplace=True)
for i in column_mean:
    df[i].fillna(df[i].mean(), inplace=True)
```

Now we have removed all the NA values with mode and mean

In [10]:

```
df.drop(column_remove,axis=1,inplace=True)
```

Dropped all the columns which we cannot handle.

Now we will be droping rest of the columns with missing percentage greater than 5%

In [11]:

```
columns_to_drop = missing_value_df[missing_value_df['percent_missing']>5].index
df2 = df.drop(columns_to_drop,axis=1)
```

Now dropping a few remaining nan values left

In [12]:

```
df2 = df2.dropna()
```

In [13]:

```
#cols = ['BsmtFinType2', 'BsmtExposure', 'BsmtCond']
#for i in cols:
#    df2.pop(i)
```

We can also drop `id` column as it is of no use

In [14]:

```
df2 = df2.drop('Id',axis=1)
```

In [15]:

```
df2.shape
```

Out[15]:

```
(1452, 67)
```

df2 is our final dataframe

In [16]:

```
df2.head()
```

Out[16]:

	MSSubClass	MSZoning	LotArea	Street	LotShape	LandContour	Utilities	LotConfig	LandS
0	60	RL	8450	Pave	Reg	Lvl	AllPub	Inside	
1	20	RL	9600	Pave	Reg	Lvl	AllPub	FR2	
2	60	RL	11250	Pave	IR1	Lvl	AllPub	Inside	
3	70	RL	9550	Pave	IR1	Lvl	AllPub	Corner	
4	60	RL	14260	Pave	IR1	Lvl	AllPub	FR2	

Outlier treatment

In [17]:

```
percentile = [0.85,0.90,0.95,1.00]
df2.describe(percentiles = percentile)
```

Out[17]:

	MSSubClass	LotArea	OverallQual	OverallCond	YearBuilt	YearRemodAdd	Ma
count	1452.000000	1452.000000	1452.000000	1452.000000	1452.000000	1452.000000	14
mean	56.949036	10507.276171	6.092975	5.579201	1971.116391	1984.775482	1
std	42.340097	9989.563592	1.381289	1.113136	30.193761	20.652466	1
min	20.000000	1300.000000	1.000000	1.000000	1872.000000	1950.000000	
50%	50.000000	9478.500000	6.000000	5.000000	1972.000000	1993.000000	
85%	90.000000	13141.450000	8.000000	7.000000	2004.350000	2006.000000	2
90%	120.000000	14373.900000	8.000000	7.000000	2006.000000	2006.000000	3
95%	160.000000	17299.350000	8.000000	8.000000	2007.000000	2007.000000	4
100%	190.000000	215245.000000	10.000000	9.000000	2010.000000	2010.000000	16
max	190.000000	215245.000000	10.000000	9.000000	2010.000000	2010.000000	16

`'LotArea','MasVnrArea','BsmtFinSF1','BsmtFinSF2','BsmtUnfSF',TotalBsmtSF','1stFlrSF', '2ndFlrSF','LowQualFinSF', 'GrLivArea', 'OpenPorchSF','EnclosedPorch','3SsnPorch','PoolArea','MiscVal','SalePrice'` shows steep increase in values when measured at 95 percentile and 100 percentile

Since there are many columns. The 5% statistical outlier might not be inclusive for all the above columns. Treating these columns might lead to loss of data. Because of limited data, we are skipping this step as of now, But if we do not get a better result in the end, we might end up tweeking these values

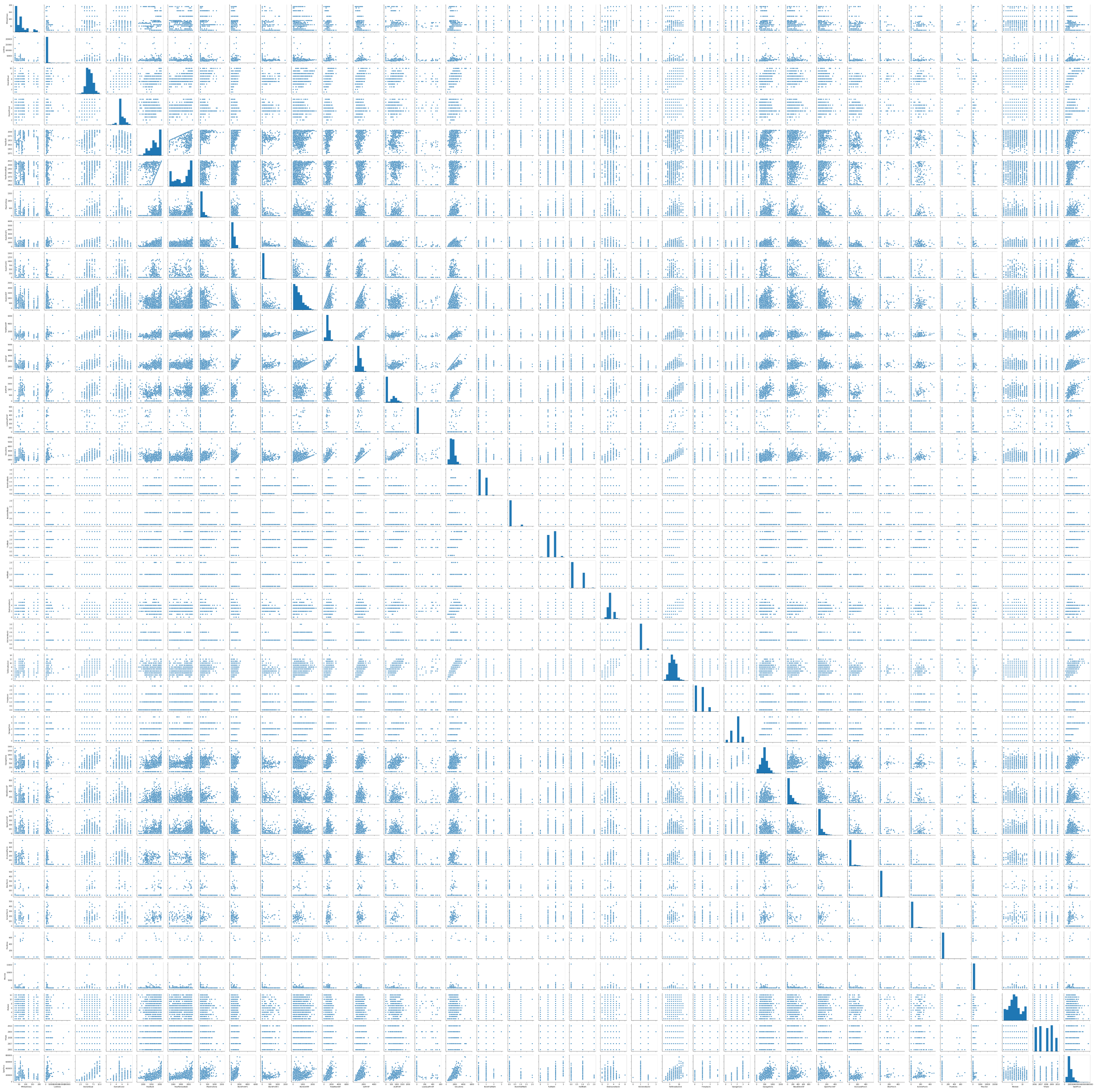
Now let's see the scatter plot of all the columns, and see if we can pick out any information or not

In [18]:

```
# paiwise scatter plot

plt.figure(figsize=(20, 10))
sns.pairplot(df2)
plt.show()
```

<Figure size 1440x720 with 0 Axes>



This scatter plot is a little difficult to comprehend because of very large number of variables

In [19]:

```
# correlation matrix
cor = df2.corr()
cor
```

Out[19]:

	MSSubClass	LotArea	OverallQual	OverallCond	YearBuilt	YearRemodAdd
MSSubClass	1.000000	-0.138054	0.034491	-0.061330	0.028397	0.041047
LotArea	-0.138054	1.000000	0.106324	-0.002269	0.015639	0.015126
OverallQual	0.034491	0.106324	1.000000	-0.090628	0.571111	0.549573
OverallCond	-0.061330	-0.002269	-0.090628	1.000000	-0.376763	0.075121
YearBuilt	0.028397	0.015639	0.571111	-0.376763	1.000000	0.590674
YearRemodAdd	0.041047	0.015126	0.549573	0.075121	0.590674	1.000000
MasVnrArea	0.022936	0.104160	0.411876	-0.128101	0.315707	0.179618
BsmtFinSF1	-0.069575	0.213063	0.236823	-0.041927	0.249239	0.127609
BsmtFinSF2	-0.066137	0.111686	-0.058039	0.039333	-0.047816	-0.066672
BsmtUnfSF	-0.138922	-0.004227	0.309602	-0.136934	0.149810	0.181828
TotalBsmtSF	-0.236906	0.258409	0.537122	-0.167230	0.392562	0.291492
1stFlrSF	-0.250050	0.295919	0.476936	-0.138814	0.284570	0.242488
2ndFlrSF	0.308104	0.052935	0.298543	0.027473	0.009566	0.140225
LowQualFinSF	0.046413	0.004904	-0.029998	0.025140	-0.183749	-0.062045
GrLivArea	0.076930	0.261159	0.594417	-0.076541	0.199343	0.288279
BsmtFullBath	0.003807	0.157702	0.108505	-0.051567	0.186305	0.118169
BsmtHalfBath	-0.002633	0.048377	-0.039207	0.117290	-0.037072	-0.011312
FullBath	0.136306	0.122457	0.552266	-0.190396	0.469625	0.440329
HalfBath	0.176165	0.016290	0.271466	-0.061434	0.240417	0.181063
BedroomAbvGr	-0.021651	0.117778	0.105900	0.014274	-0.068619	-0.038429
KitchenAbvGr	0.286572	-0.024697	-0.184642	-0.081254	-0.173951	-0.148527
TotRmsAbvGrd	0.042406	0.187990	0.430549	-0.055964	0.097440	0.193988
Fireplaces	-0.044466	0.269643	0.400398	-0.020120	0.150148	0.114806
GarageCars	-0.039043	0.154739	0.599734	-0.184866	0.537492	0.419815
GarageArea	-0.098141	0.180778	0.560543	-0.151062	0.478439	0.370674
WoodDeckSF	-0.012634	0.173167	0.240652	-0.004530	0.226891	0.207464
OpenPorchSF	-0.005462	0.086301	0.303482	-0.031172	0.185081	0.223491
EnclosedPorch	-0.010571	-0.023094	-0.112950	0.074731	-0.386839	-0.192367
3SsnPorch	-0.044049	0.020574	0.031029	0.025163	0.032037	0.045907
ScreenPorch	-0.026414	0.043511	0.066403	0.054016	-0.049169	-0.037656
PoolArea	0.008214	0.077888	0.065743	-0.002229	0.005310	0.006145
MiscVal	-0.007805	0.038226	-0.031129	0.068642	-0.034048	-0.009927
MoSold	-0.013840	0.003203	0.068760	-0.004034	0.009362	0.018588

	MSSubClass	LotArea	OverallQual	OverallCond	YearBuilt	YearRemodAdd
YrSold	-0.021529	-0.012977	-0.025186	0.043433	-0.014441	0.035352
SalePrice	-0.082813	0.264674	0.789997	-0.076294	0.522896	0.507158

In [20]:

```
'''
# plotting correlations on a heatmap

# figure size
plt.figure(figsize=(16,8))

# heatmap
sns.heatmap(cor, cmap="YlGnBu", annot=True)
plt.show()
'''
```

Out[20]:

```
'\n# plotting correlations on a heatmap\n\n# figure size\nplt.figure(figsize
=(16,8))\n\n# heatmap\nsns.heatmap(cor, cmap="YlGnBu", annot=True)\nplt.show
()\n'
```

Heatmap will be difficult to comprehend. Therefore just looking at minimum and maximum values of correlation

In [21]:

```
corunstack =cor.unstack()
corunstack[corunstack<1].sort_values()[::2]
```

Out[21]:

```
BsmtFinSF1    BsmtUnfSF       -0.496137
BsmtFullBath  BsmtUnfSF       -0.422231
YearBuilt     EnclosedPorch   -0.386839
OverallCond   YearBuilt       -0.376763
MSSubClass    1stFlrSF        -0.250050
                                 ...
SalePrice     GrLivArea        0.710080
              OverallQual      0.789997
TotalBsmtSF   1stFlrSF         0.818246
GrLivArea     TotRmsAbvGrd     0.825476
GarageCars    GarageArea       0.882332
Length: 595, dtype: float64
```

Now we can see that `BsmtFinSF1` is mostly negatively correlated with `BsmtUnfSF` with a value of -0.49, And next to which is `BsmtFullBath and BsmtUnfSF` = -0.422 On the other side `GarageArea and GarageCars` are 0.882 are the most positive correlations

PART-3 : Preparing Data

Now lets create dummy variables for the categorical variables

In [22]:

```
categoricalvar = ['MSZoning','Street','LotShape','LandContour','Utilities','LotConfig','Lar
                  'Condition1','Condition2','BldgType','HouseStyle','RoofStyle','RoofMatl','
                  'MasVnrType','ExterQual','ExterCond','Foundation','Heating','HeatingQC','C
                  'Functional','PavedDrive','SaleType','SaleCondition','BsmtFinType2', 'Bsmt
numericalvar =['LotArea','YearBuilt','YearRemodAdd','MasVnrArea','BsmtFinSF1','BsmtFinSF2',
               'BsmtUnfSF','TotalBsmtSF','1stFlrSF','2ndFlrSF','LowQualFinSF','GrLivArea','E
               'KitchenAbvGr','TotRmsAbvGrd','Fireplaces','GarageCars','GarageArea','WoodDec
               'ScreenPorch','PoolArea','MiscVal','MoSold','YrSold','SalePrice','MSSubClass'
```

variables `'OverallQual','OverallCond','MSSubClass'` are categorical variables, so we will use label encoder for these.

In [23]:

```
df2[['OverallQual','OverallCond','MSSubClass']].head()
```

Out[23]:

	OverallQual	OverallCond	MSSubClass
0	7	5	60
1	6	8	20
2	7	5	60
3	7	5	70
4	8	5	60

In [24]:

```
from sklearn import preprocessing
le = preprocessing.LabelEncoder()
df2['MSSubClass'] = le.fit_transform(df2['MSSubClass'])
```

We have not converted `'OverallQual','OverallCond'` because they are already in range of 1-10

Creating Dummy variables for the columns

In [25]:

```
# creating dummy variables for categorical variables

# convert into dummies
housing_dummies = pd.get_dummies(df2[categoricalvar], drop_first=True)
housing_dummies.head()
```

Out[25]:

	MSZoning_FV	MSZoning_RH	MSZoning_RL	MSZoning_RM	Street_Pave	LotShape_IR2	LotS
0	0	0	1	0	1	0	
1	0	0	1	0	1	0	
2	0	0	1	0	1	0	
3	0	0	1	0	1	0	
4	0	0	1	0	1	0	

Our Dummy variables are created as above, and now we will concatenate above dummy variables with original dataframe

In [26]:

```
df3 = pd.concat([df2,housing_dummies],axis=1)
df3.head()
```

Out[26]:

	MSSubClass	MSZoning	LotArea	Street	LotShape	LandContour	Utilities	LotConfig	LandS
0	5	RL	8450	Pave	Reg	Lvl	AllPub	Inside	
1	0	RL	9600	Pave	Reg	Lvl	AllPub	FR2	
2	5	RL	11250	Pave	IR1	Lvl	AllPub	Inside	
3	6	RL	9550	Pave	IR1	Lvl	AllPub	Corner	
4	5	RL	14260	Pave	IR1	Lvl	AllPub	FR2	

In [27]:

```
df3.shape
```

Out[27]:

```
(1452, 240)
```

Now dropping original variables which are no longer needed

In [28]:

```
df3.drop(categoricalvar,axis=1,inplace=True)
df3.head()
```

Out[28]:

	MSSubClass	LotArea	OverallQual	OverallCond	YearBuilt	YearRemodAdd	MasVnrArea	Bs
0	5	8450	7	5	2003	2003	196.0	
1	0	9600	6	8	1976	1976	0.0	
2	5	11250	7	5	2001	2002	162.0	
3	6	9550	7	5	1915	1970	0.0	
4	5	14260	8	5	2000	2000	350.0	

In [29]:

```
df3.shape
```

Out[29]:

```
(1452, 208)
```

In [ ]:

```
```

In [30]:

```
#popcols = ['MSSubClass','OverallQual','OverallCond']
#for i in popcols:
#    df3.pop(i)
```

Now df3 is our final dataframe

Dividing data into Train_Test split

In [71]:

```
# split into train and test

df_train, df_test = train_test_split(df3,train_size=0.9,test_size = 0.1, random_state=100)
```

We will be solving problem using k fold cross validation. That is the reason for only 10% of test data

Scalling the training variables

In [72]:

```
# Standardising the values
scaler = StandardScaler()
df_train[numericalvar] = scaler.fit_transform(df_train[numericalvar])
```

```
c:\users\ankit.chaturvedi\appdata\local\programs\python\python36\lib\site-pa
ckages\ipykernel_launcher.py:3: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/s
table/user_guide/indexing.html#returning-a-view-versus-a-copy (http://panda
s.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-ve
rsus-a-copy)
  This is separate from the ipykernel package so we can avoid doing imports
 until
c:\users\ankit.chaturvedi\appdata\local\programs\python\python36\lib\site-pa
ckages\pandas\core\indexing.py:480: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/s
table/user_guide/indexing.html#returning-a-view-versus-a-copy (http://panda
s.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-ve
rsus-a-copy)
  self.obj[item] = s
```

In [73]:

```
df_train.head()
```

Out[73]:

	MSSubClass	LotArea	OverallQual	OverallCond	YearBuilt	YearRemodAdd	MasVnArea
531	0.426447	-0.419932	-0.070818	2.135747	-1.692991	0.674263	-0.573257
1104	1.857555	-0.815832	-0.793380	-0.525785	-0.037645	-0.744382	1.131970
685	1.857555	-0.524478	0.651744	-0.525785	0.425853	-0.059519	-0.573257
1051	-1.004661	0.060237	0.651744	-0.525785	1.187312	1.065613	-0.573257
1347	-1.004661	0.455372	1.374307	-0.525785	1.154205	1.065613	-0.012327

In [74]:

```
df_train.shape
```

Out[74]:

```
(1306, 208)
```

In [75]:

```
y_train = df_train.pop('SalePrice')
X_train = df_train
```

We have now created `X_train and y_train` , which is our clean dataset

Setting up test dataset

Transforming Test dataset

In [76]:

```
df_test[numericalvar] = scaler.transform(df_test[numericalvar])
df_test.head()
```

```
c:\users\ankit.chaturvedi\appdata\local\programs\python\python36\lib\site-pa
ckages\ipykernel_launcher.py:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/s
table/user_guide/indexing.html#returning-a-view-versus-a-copy (http://panda
s.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-ve
rsus-a-copy)
  """Entry point for launching an IPython kernel.
c:\users\ankit.chaturvedi\appdata\local\programs\python\python36\lib\site-pa
ckages\pandas\core\indexing.py:480: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/s
table/user_guide/indexing.html#returning-a-view-versus-a-copy (http://panda
s.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-ve
rsus-a-copy)
  self.obj[item] = s
```

Out[76]:

	MSSubClass	LotArea	OverallQual	OverallCond	YearBuilt	YearRemodAdd	MasVnrArea
157	0.187929	0.139436	1.374307	-0.525785	1.253526	1.212369	-0.102075
337	-1.004661	-0.134891	0.651744	-0.525785	1.021777	0.869938	0.060594
1120	-0.766143	-0.218299	-0.070818	-0.525785	-1.692991	-1.722758	-0.573257
563	-0.050589	1.074617	-0.070818	1.248570	-1.759205	-1.722758	-0.573257
371	-0.050589	0.628883	-1.515942	-1.412963	-0.401821	-1.282489	-0.573257

In [77]:

```
y_test = df_test.pop('SalePrice')
X_test = df_test
```

PART - 4 : Model Building

Using Ridge and Lasso regression

Now lets predict house price using both Ridge and Lasso Regression using `Grid Search Cross Validation`

But first let's see how this perform in linear regression model

Linear Regression with RFE

In [78]:

```
features=25
# first model with an arbitrary choice of n_features
# running RFE with number of features=10

lm = LinearRegression()
lm.fit(X_train, y_train)

rfe = RFE(lm, n_features_to_select=features)
rfe = rfe.fit(X_train, y_train)
# tuples of (feature name, whether selected, ranking)
# note that the 'rank' is > 1 for non-selected features
list(zip(X_train.columns,rfe.support_,rfe.ranking_))
# predict prices of X_test
```

```
 ('Neighborhood_SWISU', False, 70),
 ('Neighborhood_Sawyer', False, 60),
 ('Neighborhood_SawyerW', False, 101),
 ('Neighborhood_Somerst', False, 127),
 ('Neighborhood_StoneBr', True, 1),
 ('Neighborhood_Timber', False, 93),
 ('Neighborhood_Veenker', False, 145),
 ('Condition1_Feedr', False, 147),
 ('Condition1_Norm', False, 73),
 ('Condition1_PosA', False, 126),
 ('Condition1_PosN', False, 76),
 ('Condition1_RRAe', False, 11),
 ('Condition1_RRAn', False, 74),
 ('Condition1_RRNe', False, 160),
 ('Condition1_RRNn', False, 166),
 ('Condition2_Feedr', False, 136),
 ('Condition2_Norm', False, 137),
 ('Condition2_PosA', False, 4),
 ('Condition2_PosN', True, 1),
 ('Condition2_RRAe', True, 1),
```

In [79]:

```
list1 = []
for i in zip(X_train.columns,rfe.support_):
    if i[1]==True:
        list1.append(i[0])
```

In [ ]:

In [80]:

```
y_train_pred =rfe.predict(X_train)
r2 = r2_score(y_train, y_train_pred)
print(r2)
```

```
0.8340329817358673
```

In [112]:

```
# predict prices of X_test
y_pred = rfe.predict(X_test)

# evaluate the model on test set
r2 = r2_score(y_test, y_pred)
print(r2)
```

```
0.8571248845745755
```

In case of Linear regression without RFE

In [113]:

```
# predict prices of X_test

# evaluate the model on test set
r2 = r2_score(y_test, lm.predict(X_test))
print(r2)
```

```
-205913403599308.47
```

In [82]:

```
range(len(y_test))
```

Out[82]:

```
range(0, 146)
```

In [83]:

```
mse = np.mean((y_pred - y_test)**2)
mse
```

Out[83]:

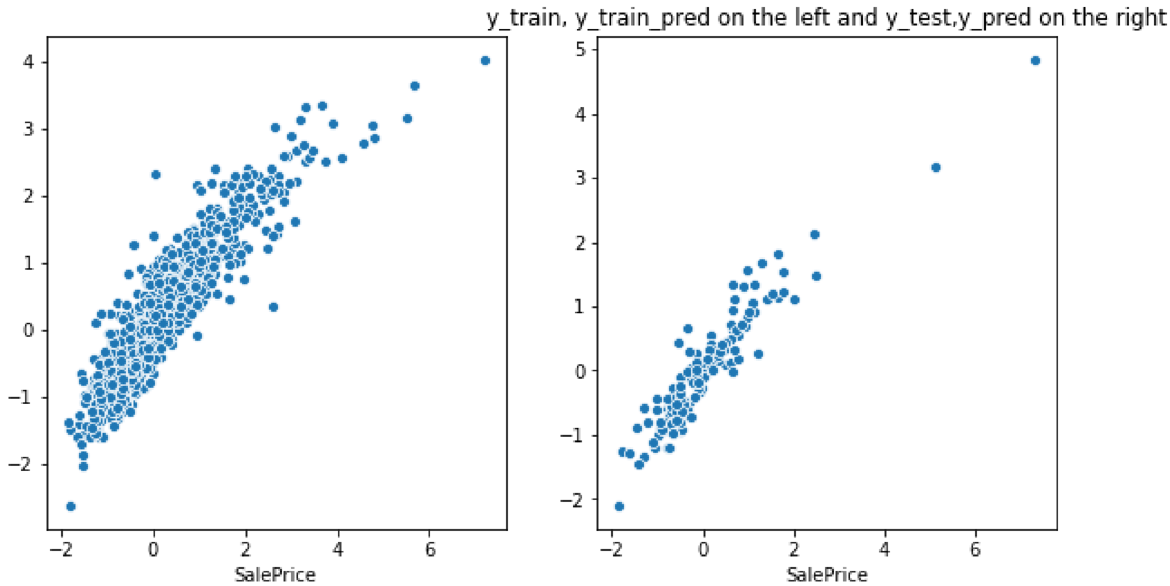
```
0.17347039478378112
```

In [84]:

```
fig,ax = plt.subplots(nrows = 1,ncols=2, figsize=(10,5))
sns.scatterplot(x=y_train, y=y_train_pred,ax=ax[0])
sns.scatterplot(x=y_test, y=y_pred,ax=ax[1])
plt.title("y_train, y_train_pred on the left and y_test,y_pred on the right")
#plt.scatter(x=y_pred, y=range(len(y_test)),c='#ff7f0e')
#['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b', '#e377c2', '#7f7f7f', '
```

Out[84]:

```
Text(0.5, 1.0, 'y_train, y_train_pred on the left and y_test,y_pred on the r
ight')
```



As, we can see that training data is well alligned towards the origin however test data is deviating from the center

The Prediction R^2 Score is around `85%` for Linear Regression using RFE with number of features = 25. And Train dataset R^2 is `83%` . But if we don't use RFE then test R^2 dips down to `-20` . Clearly overfitting is there

Furthermore Linear regression is comutationally more intensive than `Ridge and Lasso` . Grid Search for Linear model using rfe hyperparameter was not possible because of large number of columns which made it comutationally difficult to perform

Ridge Regression

In [85]:

```
# list of alphas to tune
params = {'alpha': [0.0001, 0.001, 0.01, 0.05, 0.1,
 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 2.0, 3.0,
 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 20, 50, 100, 500, 1000 ]}


ridge = Ridge()

# cross validation
folds = 10
model_cvr = GridSearchCV(estimator = ridge,
                        param_grid = params,
                        scoring= 'neg_mean_absolute_error',
                        cv = folds,
                        return_train_score=True,
                        verbose = 1)
model_cvr.fit(X_train, y_train)
```

```
Fitting 10 folds for each of 28 candidates, totalling 280 fits

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent work
ers.
[Parallel(n_jobs=1)]: Done 280 out of 280 | elapsed:    2.9s finished
```

Out[85]:

```
GridSearchCV(cv=10, error_score='raise-deprecating',
             estimator=Ridge(alpha=1.0, copy_X=True, fit_intercept=True,
                             max_iter=None, normalize=False, random_state=No
ne,
                             solver='auto', tol=0.001),
             iid='warn', n_jobs=None,
             param_grid={'alpha': [0.0001, 0.001, 0.01, 0.05, 0.1, 0.2, 0.3,
                                   0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 2.0,
3.0,
                                   4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 20, 5
0,
                                   100, 500, 1000]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
             scoring='neg_mean_absolute_error', verbose=1)
```

In [86]:

```
cv_results_r = pd.DataFrame(model_cvr.cv_results_)
cv_results_r = cv_results_r[cv_results_r['param_alpha']<=200]
cv_results_r.head()
```

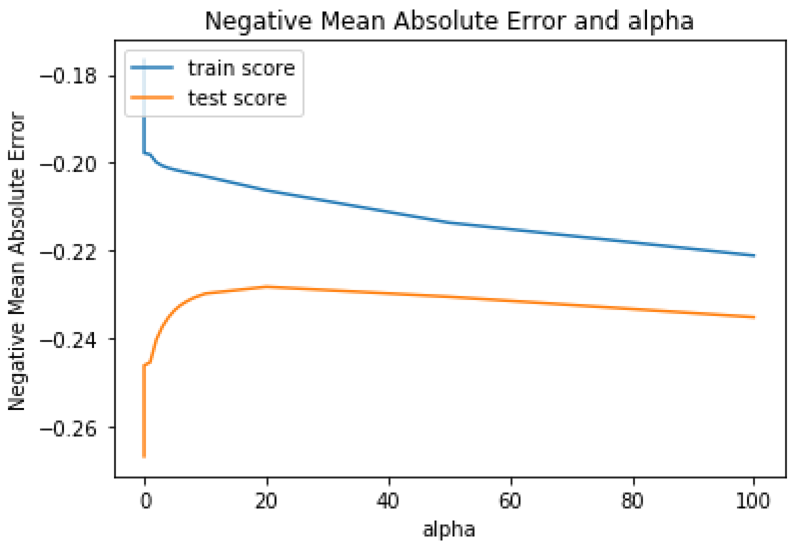
Out[86]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_alpha	params	split0_t
0	0.007978	0.000630	0.000898	2.991836e-04	0.0001	{'alpha': 0.0001}	
1	0.007675	0.000454	0.000903	3.013520e-04	0.001	{'alpha': 0.001}	
2	0.008378	0.001558	0.001098	2.989196e-04	0.01	{'alpha': 0.01}	
3	0.007978	0.000446	0.000899	2.996364e-04	0.05	{'alpha': 0.05}	
4	0.007779	0.000399	0.000997	9.536743e-08	0.1	{'alpha': 0.1}	

In [87]:

```
# plotting mean test and train scoes with alpha
cv_results_r['param_alpha'] = cv_results_r['param_alpha'].astype('int32')

# plotting
plt.plot(cv_results_r['param_alpha'], cv_results_r['mean_train_score'])
plt.plot(cv_results_r['param_alpha'], cv_results_r['mean_test_score'])
plt.xlabel('alpha')
plt.ylabel('Negative Mean Absolute Error')
plt.title("Negative Mean Absolute Error and alpha")
plt.legend(['train score', 'test score'], loc='upper left')
plt.show()
```



In [88]:

```
max_ridge_alpha = max(cv_results_r['mean_test_score'])
cv_results_r[cv_results_r['mean_test_score']==max_ridge_alpha]
```

Out[88]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_alpha	params	split0_
23	0.007779	0.000399	0.000898	0.000299	20	{'alpha': 20}	

In [121]:

```
alpha = 10
ridge = Ridge(alpha=alpha)

ridge.fit(X_train, y_train)
ridge.coef_
```

```
       -0.02271326, -0.10256521,  0.17173597, -0.21138027, -0.15509398,
       -0.06024345, -0.03962833, -0.14785189, -0.12035916,  0.02208872,
       -0.13384952,  0.3512959 ,  0.3621209 , -0.09968632, -0.05153044,
       -0.08448426, -0.01771345,  0.06693226,  0.36215224, -0.05606225,
        0.04557308, -0.07102426,  0.11642547,  0.01338193, -0.05827234,
       -0.09886556,  0.07546105, -0.01741098, -0.02256401,  0.02650485,
        0.12307904,  0.07006704, -0.30497616, -0.00524663,  0.0234353 ,
        0.02152078,  0.05287189,  0.03274411, -0.13193374, -0.08947752,
        0.07269442,  0.10862447, -0.03000053, -0.06047075, -0.11239276,
        0.0505151 ,  0.06578292, -0.05872217,  0.02045294, -0.00165115,
        0.04234719,  0.03314074,  0.16467362,  0.02947051,  0.01133335,
        0.00791212, -0.04505676,  0.01793012,  0.24791438,  0.        ,
        0.00309275,  0.15393083, -0.00964039,  0.04731228, -0.0592186 ,
       -0.03193885,  0.01930712, -0.00465966, -0.01524652, -0.08005205,
       -0.00760854, -0.04809623,  0.0342129 ,  0.00256761,  0.00422461,
        0.03262482, -0.00964039,  0.0492034 ,  0.00352939,  0.16143935,
       -0.00979052, -0.01433063, -0.04140513, -0.01768239, -0.10259113,
        0.02490097,  0.00693233, -0.07266276,  0.01715764,  0.06933904,
        0.05730589, -0.04528918, -0.09395618, -0.14225863,  0.01156594,
       -0.04240775, -0.00497788, -0.03071963,  0.03175648,  0.07445382,
```

In [122]:

```
ridge_pred = ridge.predict(X_test)
```

In [123]:

```
ridge_pred_train = ridge.predict(X_train)
```

Prediction R^2 is below

In [124]:

```
ridge.score(X_test,y_test)
```

Out[124]:

```
0.8721268196617036
```

Training R^2 is below

In [125]:

```
ridge.score(X_train,y_train)
```

Out[125]:

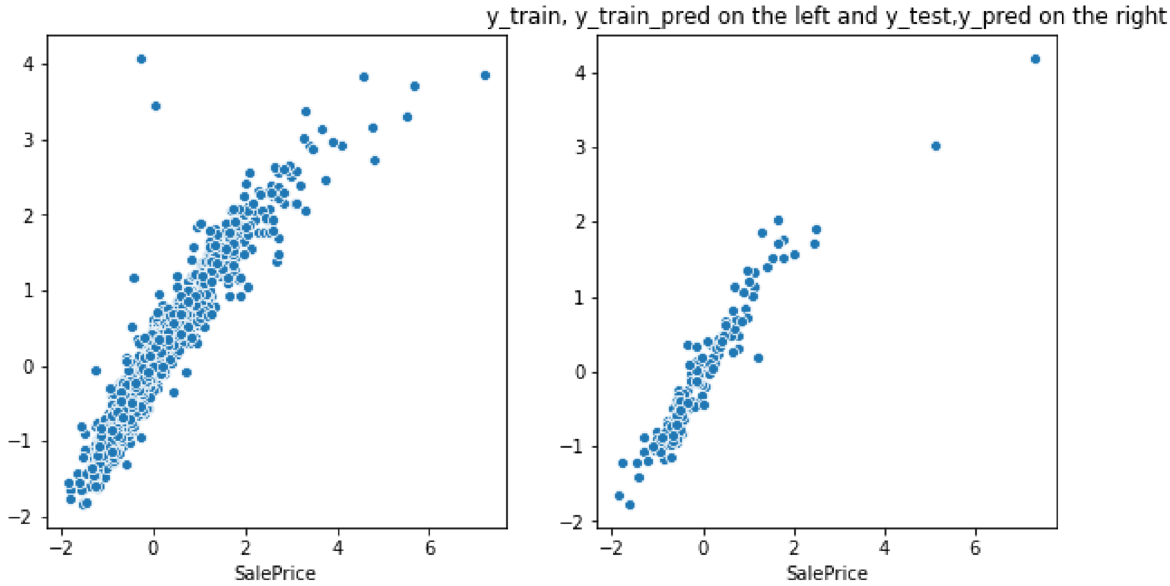
```
0.886926032880223
```

In [94]:

```
fig,ax = plt.subplots(nrows = 1,ncols=2, figsize=(10,5))
sns.scatterplot(x=y_train, y=ridge_pred_train,ax=ax[0])
sns.scatterplot(x=y_test, y=ridge_pred,ax=ax[1])
plt.title("y_train, y_train_pred on the left and y_test,y_pred on the right")
```

Out[94]:

```
Text(0.5, 1.0, 'y_train, y_train_pred on the left and y_test,y_pred on the r
ight')
```



Both the scatter plots looks quite similar, we can say model is well fitted

In [95]:

```
mse = np.mean((ridge_pred - y_test)**2)
mse
```

Out[95]:

```
0.15525594509227333
```

The Prediction R^2 Score is around `87%` for Ridge Regression and Train R^2 Score is 88% with `mean square error is 0.155`

Lasso Regression

In [96]:

```
lasso = Lasso()

# cross validation
model_cvl = GridSearchCV(estimator = lasso,
                        param_grid = params,
                        scoring= 'neg_mean_absolute_error',
                        cv = folds,
                        return_train_score=True,
                        verbose = 1)

model_cvl.fit(X_train, y_train)
```

```
Fitting 10 folds for each of 28 candidates, totalling 280 fits

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent work
ers.
c:\users\ankit.chaturvedi\appdata\local\programs\python\python36\lib\site-pa
ckages\sklearn\linear_model\coordinate_descent.py:475: ConvergenceWarning: O
bjective did not converge. You might want to increase the number of iteratio
ns. Duality gap: 0.7151724178520595, tolerance: 0.11568332170727916
  positive)
[Parallel(n_jobs=1)]: Done 280 out of 280 | elapsed:    3.3s finished
```

Out[96]:

```
GridSearchCV(cv=10, error_score='raise-deprecating',
             estimator=Lasso(alpha=1.0, copy_X=True, fit_intercept=True,
                             max_iter=1000, normalize=False, positive=False,
                             precompute=False, random_state=None,
                             selection='cyclic', tol=0.0001, warm_start=Fals
e),
             iid='warn', n_jobs=None,
             param_grid={'alpha': [0.0001, 0.001, 0.01, 0.05, 0.1, 0.2, 0.3,
                                   0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 2.0,
3.0,
                                   4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 20, 5
0,
                                   100, 500, 1000]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
             scoring='neg_mean_absolute_error', verbose=1)
```

In [97]:

```
cv_results_l = pd.DataFrame(model_cvl.cv_results_)
cv_results_l = cv_results_l[cv_results_l['param_alpha']<=200]
cv_results_l.head()
```

Out[97]:

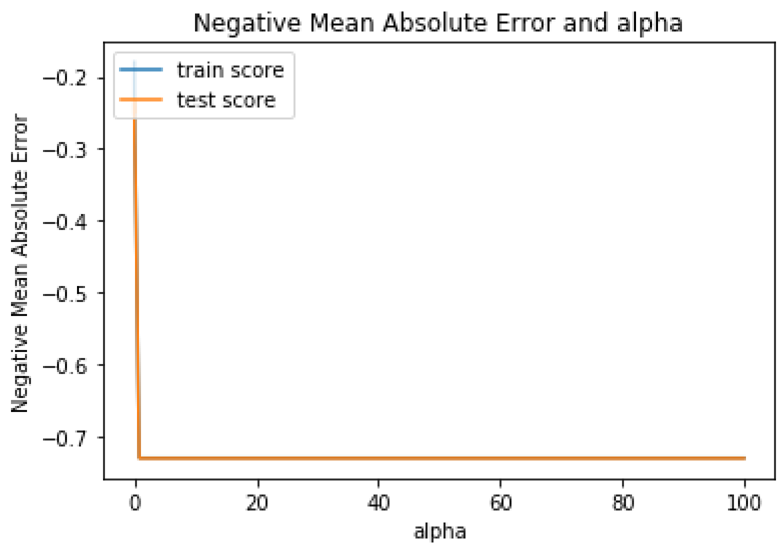
	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_alpha	params	split0_t
0	0.061137	0.031030	0.000997	0.000001	0.0001	{'alpha': 0.0001}	
1	0.042782	0.003928	0.001000	0.000008	0.001	{'alpha': 0.001}	
2	0.008871	0.000698	0.001195	0.000385	0.01	{'alpha': 0.01}	
3	0.006587	0.000481	0.000898	0.000300	0.05	{'alpha': 0.05}	
4	0.006389	0.000483	0.000995	0.000013	0.1	{'alpha': 0.1}	

In [98]:

```
# plotting mean test and train scoes with alpha
cv_results_l['param_alpha'] = cv_results_l['param_alpha'].astype('float32')

# plotting
plt.plot(cv_results_l['param_alpha'], cv_results_l['mean_train_score'])
plt.plot(cv_results_l['param_alpha'], cv_results_l['mean_test_score'])
plt.xlabel('alpha')
plt.ylabel('Negative Mean Absolute Error')

plt.title("Negative Mean Absolute Error and alpha")
plt.legend(['train score', 'test score'], loc='upper left')
plt.show()
```



In [99]:

```
max_lasso_alpha = max(cv_results_l['mean_test_score'])
cv_results_l[cv_results_l['mean_test_score']==max_lasso_alpha]
```

Out[99]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_alpha	params	split0_t
1	0.042782	0.003928	0.001	0.000008	0.001	{'alpha': 0.001}	

In [100]:

```
alpha =0.001

lasso = Lasso(alpha=alpha)

lasso.fit(X_train, y_train)
```

Out[100]:

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

In [101]:

```
lasso.coef_
```

Out[101]:

```
array([-7.77034254e-02,  4.91964013e-02,  1.96952971e-01,  6.67650222e-02,
        7.82141426e-02,  2.84833563e-02,  4.43005377e-02,  4.81483607e-02,
        8.49743048e-03, -4.35166754e-03,  0.00000000e+00,  0.00000000e+00,
        7.07889050e-02, -1.16687897e-02,  2.69937555e-01,  6.11175594e-02,
        7.49648492e-04,  7.69179512e-02,  3.50491548e-02, -4.73508782e-02,
       -2.69554787e-02,  7.69195492e-02,  2.75336474e-02,  8.96588447e-02,
       -9.65390933e-03,  2.40655864e-02,  0.00000000e+00,  5.14371864e-03,
        1.25313871e-02,  2.72192005e-02, -3.22810044e-03, -7.03609373e-03,
       -5.05851479e-03, -8.76123763e-03,  5.92576396e-02,  0.00000000e+00,
        5.94935970e-02, -0.00000000e+00,  1.70221706e-01,  6.50309215e-02,
       -2.33474065e-01,  0.00000000e+00,  1.50243596e-01,  2.42862183e-02,
        1.34621665e-01, -0.00000000e+00,  1.54994437e-01, -3.20733280e-02,
       -0.00000000e+00, -0.00000000e+00,  1.03431739e-01, -0.00000000e+00,
       -0.00000000e+00,  0.00000000e+00,  6.00568311e-02,  0.00000000e+00,
       -1.84287928e-02,  2.46552701e-01, -1.48797792e-01, -6.32915177e-02,
       -0.00000000e+00, -0.00000000e+00, -9.46830039e-02, -6.44672909e-02,
        0.00000000e+00, -8.06262110e-02,  5.13329944e-01,  4.93358784e-01,
       -6.38996680e-02, -0.00000000e+00, -2.77424263e-02,  1.47986686e-02,
```

In [102]:

```
lasso_pred = lasso.predict(X_test)
```

In [103]:

```
lasso_pred_train = lasso.predict(X_train)
```

In [104]:

```
lasso.score(X_test,y_test)
```

Out[104]:

```
0.8857195627490981
```

In [105]:

```
lasso.score(X_train,y_train)
```

Out[105]:

```
0.8906758668277343
```

In [106]:

```
mse = np.mean((lasso_pred - y_test)**2)
mse
```

Out[106]:

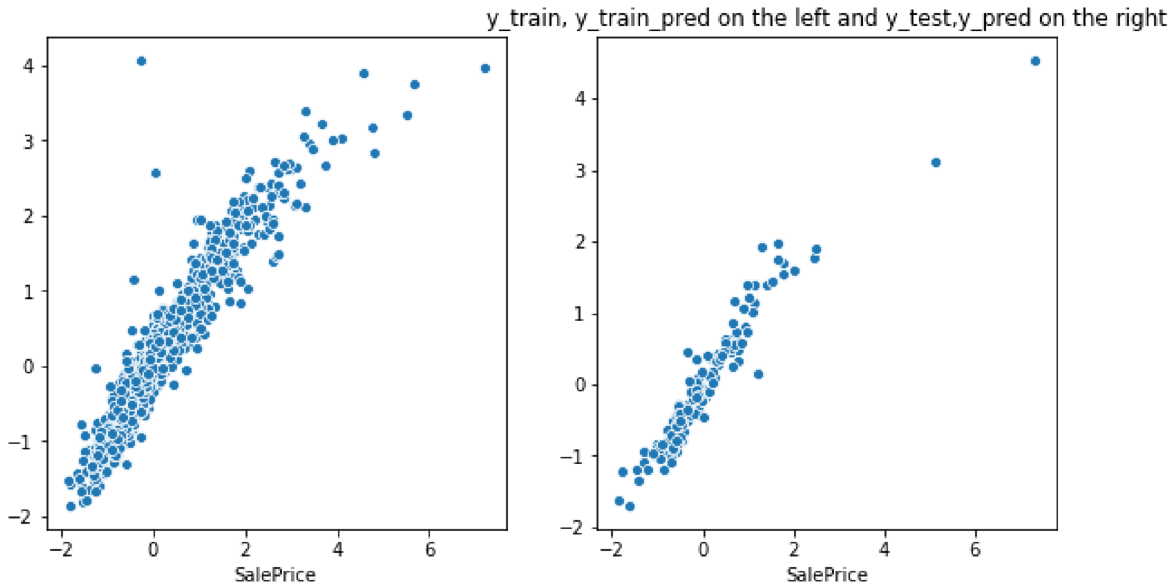
```
0.13875245179644036
```

In [107]:

```
fig,ax = plt.subplots(nrows = 1,ncols=2, figsize=(10,5))
sns.scatterplot(x=y_train, y=lasso_pred_train,ax=ax[0])
sns.scatterplot(x=y_test, y=lasso_pred,ax=ax[1])
plt.title("y_train, y_train_pred on the left and y_test,y_pred on the right")
```

Out[107]:

```
Text(0.5, 1.0, 'y_train, y_train_pred on the left and y_test,y_pred on the r
ight')
```



Because all the coefficients turn out to zero , lasso is not able to predict the model

The Prediction R^2 Score is around `88%` for Lasso Regression and Train R^2 Score is `89%` with `mean square error is 0.13`

Lasso is the best performing moddel compared to Ridge and Linear regression with RFE. When Linear regression without RFE is used then it shows clear sign of overfitting with test R^2 equal to -20

Here, the key fact about LASSO regression is that it minimizes sum of squared error, under the constraint that the sum of absolute values of coefficients is less than some constant c.So, for all of the coefficients to be zero, there must be no vector of coefficients with summed absolute value less than c that improves error.

For another view, consider the LASSO loss function:

$\sum_{i=1}^{n}(Yi - XTi\beta) + \lambda \sum_{j=1}^{p} |\beta j|$
"If λ is sufficiently large, some of the coefficients are driven to zero, leading to a sparse model." For it to be the case that zero coefficients minimize this function, λ must be large enough that any improvement in error (the left term) is less than the added loss from the increased norm (the right term).

In [ ]:

```

```

In [ ]:

```

```

So our Final model is Lasso regression model with R^2 value equal to 88% on Test Data with mean square error equal to 0.13

In [114]:

```
lasso
```

Out[114]:

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

Our selection for lasso is also because of automatic feature selection

In [ ]:

```

```