All About ML — Part 2: Lasso and Ridge Regularization

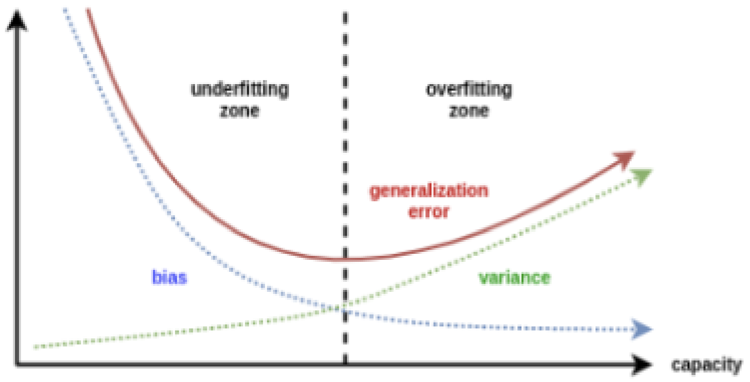
Dharani J Follow
Mar 6, 2020 · 9 min read

1.1 Introduction

In machine learning when we use supervised learning algorithms on a data set, there will be situations where the model performs really well on train data and when tested on new data it might not perform well and also has high error. This is due to multiple reasons like collinearity, bias-variance decomposition and over modeling on train data. Dealing with collinearity is discussed in ***my previous blog*** ***:)***

1.2 Bias Variance Trade off

Bias and **Variance** are the measures which helps us understand how deviation of the function is varied. Bias is the measure of deviation or error from actual value of the function. Variance measures deviation in response variable function if we estimated it with a different training sample of data set.
- From the definitions it can be inferred that while modelling we must keep bias as low as possible that implies accuracy is high
- Also by changing samples in training data set, one should not get highly varied results of the output. Therefore low variance is preferred for a good performing model

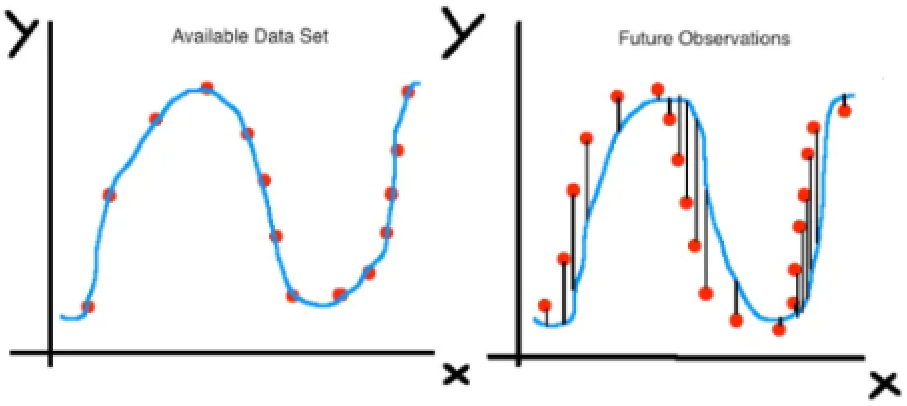


optimal capacity

Source

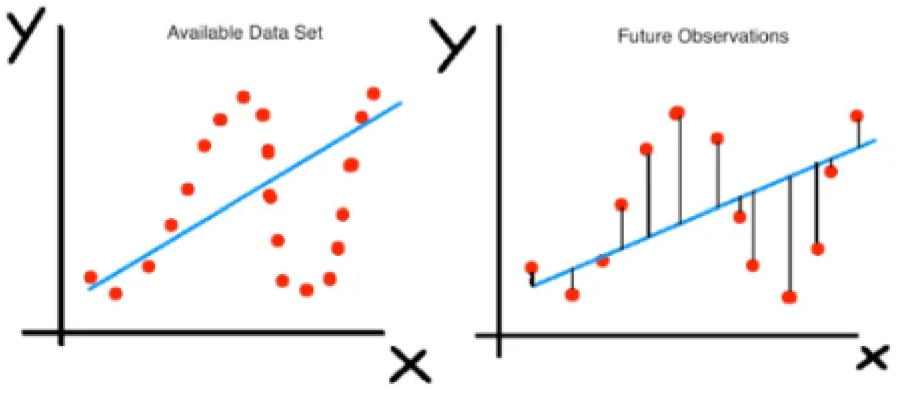
But here comes a catch,

- If we try to **reduce the bias**, then the model would fit exactly well on that specific sample of training data and it cannot find the underlying patterns in the data set that it has never seen. So it is very likely that the model will have deviated output when another sample is used for training. This then results in **high variance**.



source

- Similarly when we want to have less deviation or **low variance** when different samples are used then, the model will not fit exactly on the data points and results in **high bias**



source

1.3 Overfitting

The situation where we had **low bias and high variance** is called **overfitting** as the model fits absolutely well with high accuracy on available data and when it sees a new data it fails to predict, leading to high test error. This generally happens with data that has many features and the model considers the contribution of the estimated coefficients of all of them and tries to over estimate the actual value. But in reality it might be the case that only few features of the data set are really important and impact the predictions. So if the less impactful features are more in number they tend to add value to the function in training data and when new data comes up that has nothing to do with these features then the predictions goes wrong.

1.4 Regularization

So it is highly important to restrict the features while modelling to minimize the risk of overfitting and this process is called **regularization**.

In **regression** we know that the features are estimated using **coefficients** and these estimates are the real game changes in modelling. If there is a possibility to 'restrict' or 'shrink' or 'regularize' the estimates towards zero, then the effect of the non-impactful features is reduced and it saves the model from high variance with a stable fit. In terms of a typical linear regression model using ordinary least squares, this is done by modifying our typical loss function (Residual Sum of Squares, RSS) by adding a **penalty** for higher magnitude coefficient values.

There are few things to keep in mind while using regularization. There needs to be a constant lookup on bias Vs. variance trade off while using the shrinkage parameter. The more we shrink the coefficients the more we reduce the variance which might pitch in high bias. Noting all the trade offs we now proceed to learn the regularization techniques

2.1 Ridge and Lasso Regularization

Ridge Regularization:

Recall from the ***previous blog*** where we discussed about RSS and how it helps in estimating the coefficients by reducing RSS.

$$\mathrm{RSS} = \sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j x_{ij}\right)^2$$

- **Ridge** regression is quite similar to RSS except that there is also a **shirnkage parametre 'λ'** that minimizes their value. 'λ' is also called as '**tuning parametre**' and it is determined separately using **cross-validation** technique

$$\sum_{i=1}^{n}\left(y_i-\beta_0-\sum_{j=1}^{p}\beta_j x_{ij}\right)^2+\lambda\sum_{j=1}^{p}\beta_j^2=\mathrm{RSS}+\lambda\sum_{j=1}^{p}\beta_j^2$$

- Suppose the coefficients β1, . . . , βp are having some values and out of them few must have values already close to zero which as discussed above where features that do no have much impact on the response variable. When we add the shrinkage parametre these values which are already having small value will tend to zero in the equation shown above. So, the second term after RSS is called shrinkage penalty or **l2 norm**.
- If λ=0 then the equation is as normal as RSS, but if λ → ∞, the impact of shrinkage penalty increases and the ridge regression estimate coefficients will approach to zero.
- Ridge regression's advantage over least squares is rooted in the bias-variance trade-off. As λ increases, the flexibility of the ridge regression fit decreases, leading to decreased variance but increased bias.
- Ridge regression does have one disadvantage. Ridge regression will include all p predictors in the final model. The shrinkage penalty will shrink all of the coefficients towards zero, but it will not set any of them exactly to zero (unless λ = ∞). So we need to resort to **step wise selection** models again to pick up the important features.

Lasso Regularization:

To overcome the problem that ridge has, Lasso(Least Absolute Shrinkage and Selection Operator) is an alternative that can pick relevant features that will be useful for modelling. Lasso also has the shrinkage parametre but the difference that has with Ridge is that there is no squared term of the estimated coefficient but only an absolute value.

$$n \quad \Big(\qquad p \qquad \Big)^{2} \qquad p \qquad\qquad p$$

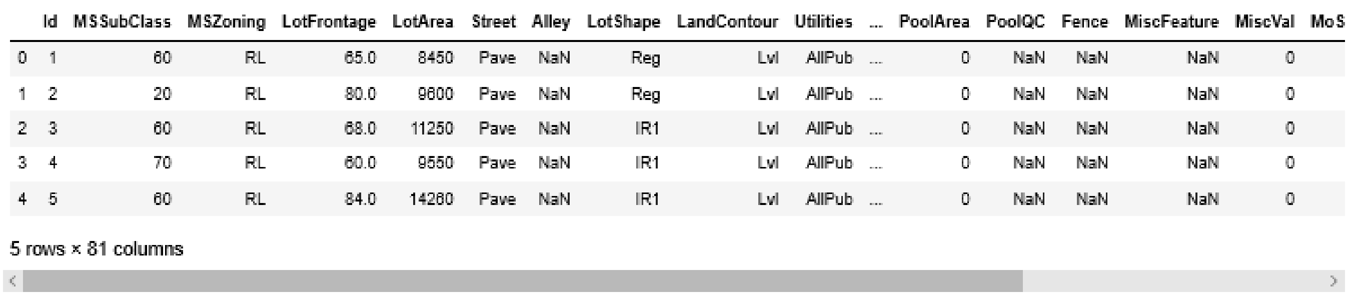
$$\sum_{i=1} \left(y_i - \beta_0 - \sum_{j=1} \beta_j x_{ij} \right) + \lambda \sum_{j=1} |\beta_j| = \mathrm{RSS} + \lambda \sum_{j=1} |\beta_j|$$

- Like in Ridge regression, lasso also shrinks the estimated coefficients to zero but the penalty effect will forcefully make the coefficients equal to zero if the tuning parameter is large enough. Hence, much like best subset selection, the lasso performs **feature selection**. As a result, models generated from the lasso are generally much easier to interpret.
- The term after RSS is called the shrinkage penalty or **l1 norm**

3. Python Tutorial on Lasso & Ridge Regularization

Now lets get some hands on with house prices data set. We need to predict the prices of houses given some features. Lets import the dependencies and data that we need to use

```
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
import seaborn as sns
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn import metrics
from sklearn import ensemble
from sklearn.linear_model import Lasso,Ridge

#load train data
df_data=pd.read_csv("data_price.csv")
df_data.head()
```

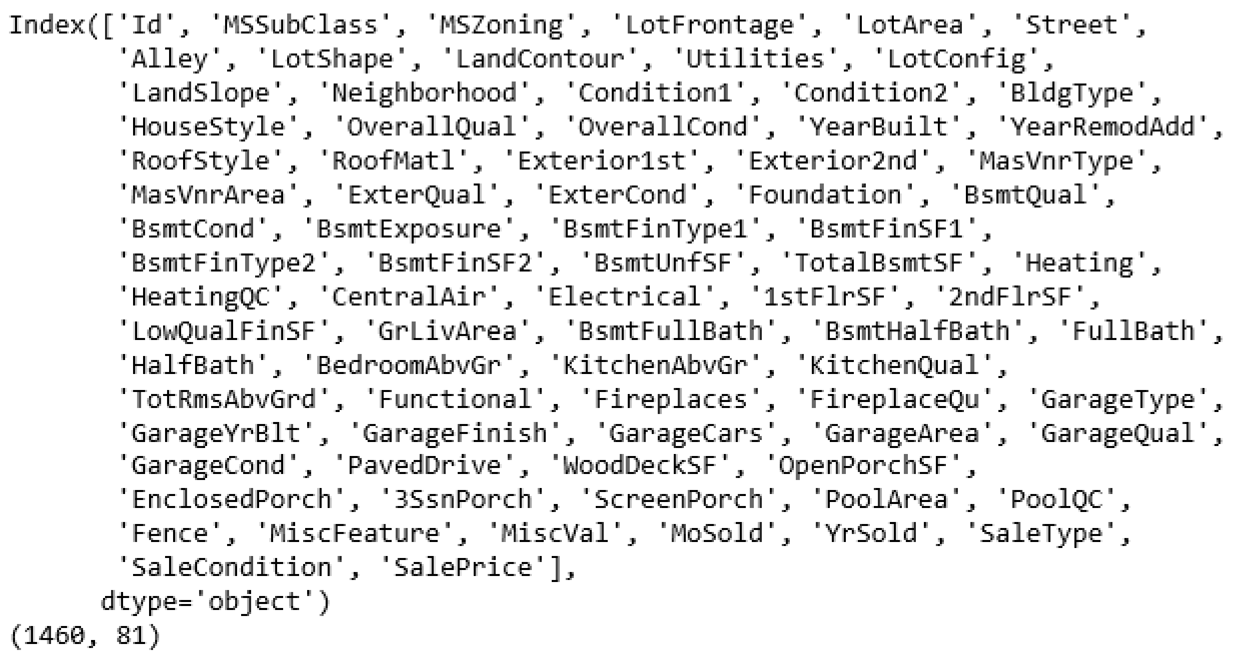


	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Utilities	...	PoolArea	PoolQC	Fence	MiscFeature	MiscVal	MoS
0	1	60	RL	65.0	8450	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	NaN	NaN	0	
1	2	20	RL	80.0	9600	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	NaN	NaN	0	
2	3	60	RL	68.0	11250	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	NaN	NaN	0	
3	4	70	RL	60.0	9550	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	NaN	NaN	0	
4	5	60	RL	84.0	14260	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	NaN	NaN	0	

5 rows × 81 columns

81 columns

```
#to know each and every column execute the following
print(df_data.columns)
print(df_data.shape)
```

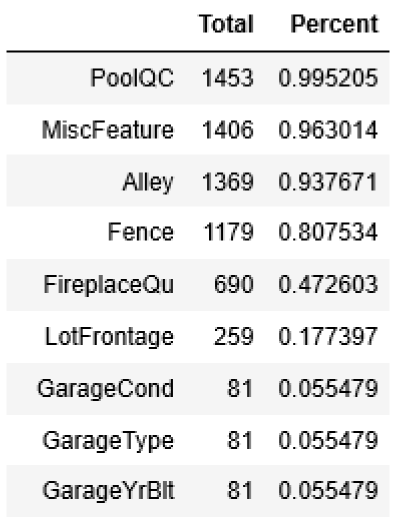


```
Index(['Id', 'MSSubClass', 'MSZoning', 'LotFrontage', 'LotArea', 'Street',
       'Alley', 'LotShape', 'LandContour', 'Utilities', 'LotConfig',
       'LandSlope', 'Neighborhood', 'Condition1', 'Condition2', 'BldgType',
       'HouseStyle', 'OverallQual', 'OverallCond', 'YearBuilt', 'YearRemodAdd',
       'RoofStyle', 'RoofMatl', 'Exterior1st', 'Exterior2nd', 'MasVnrType',
       'MasVnrArea', 'ExterQual', 'ExterCond', 'Foundation', 'BsmtQual',
       'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFinSF1',
       'BsmtFinType2', 'BsmtFinSF2', 'BsmtUnfSF', 'TotalBsmtSF', 'Heating',
       'HeatingQC', 'CentralAir', 'Electrical', '1stFlrSF', '2ndFlrSF',
       'LowQualFinSF', 'GrLivArea', 'BsmtFullBath', 'BsmtHalfBath', 'FullBath',
       'HalfBath', 'BedroomAbvGr', 'KitchenAbvGr', 'KitchenQual',
       'TotRmsAbvGrd', 'Functional', 'Fireplaces', 'FireplaceQu', 'GarageType',
       'GarageYrBlt', 'GarageFinish', 'GarageCars', 'GarageArea', 'GarageQual',
       'GarageCond', 'PavedDrive', 'WoodDeckSF', 'OpenPorchSF',
       'EnclosedPorch', '3SsnPorch', 'ScreenPorch', 'PoolArea', 'PoolQC',
       'Fence', 'MiscFeature', 'MiscVal', 'MoSold', 'YrSold', 'SaleType',
       'SaleCondition', 'SalePrice'],
      dtype='object')
(1460, 81)
```

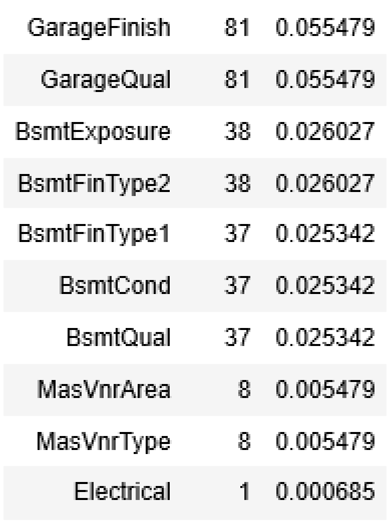
1460 rows and 81 columns

As we have a basic idea on the data lets see how to deal with Null values in it

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



	Total	Percent
PoolQC	1453	0.995205
MiscFeature	1406	0.963014
Alley	1369	0.937671
Fence	1179	0.807534
FireplaceQu	690	0.472603
LotFrontage	259	0.177397
GarageCond	81	0.055479
GarageType	81	0.055479
GarageYrBlt	81	0.055479

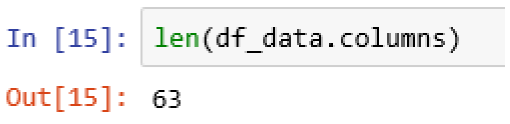


GarageFinish	81	0.055479
GarageQual	81	0.055479
BsmtExposure	38	0.026027
BsmtFinType2	38	0.026027
BsmtFinType1	37	0.025342
BsmtCond	37	0.025342
BsmtQual	37	0.025342
MasVnrArea	8	0.005479
MasVnrType	8	0.005479
Electrical	1	0.000685

Arranged in descending order

We can drop the columns having more than 15% of null values. So the columns till 'LotFrontage' can be removed. If we check the columns like 'GarageXXXX' they seem to be related with Garage area and we can remove them as well are they are collinear. Similar case with 'BsmtXXX' and 'MasVnrXXXX'. For 'Electrical' there is only one datapoint that is null. So we can remove that specific row from the data.

```
df_data=
df_data.drop(missing_data[missing_data['Total']>1].index.values,1)
df_data=
df_data.drop(df_data.loc[df_data['Electrical'].isnull()].index)
```

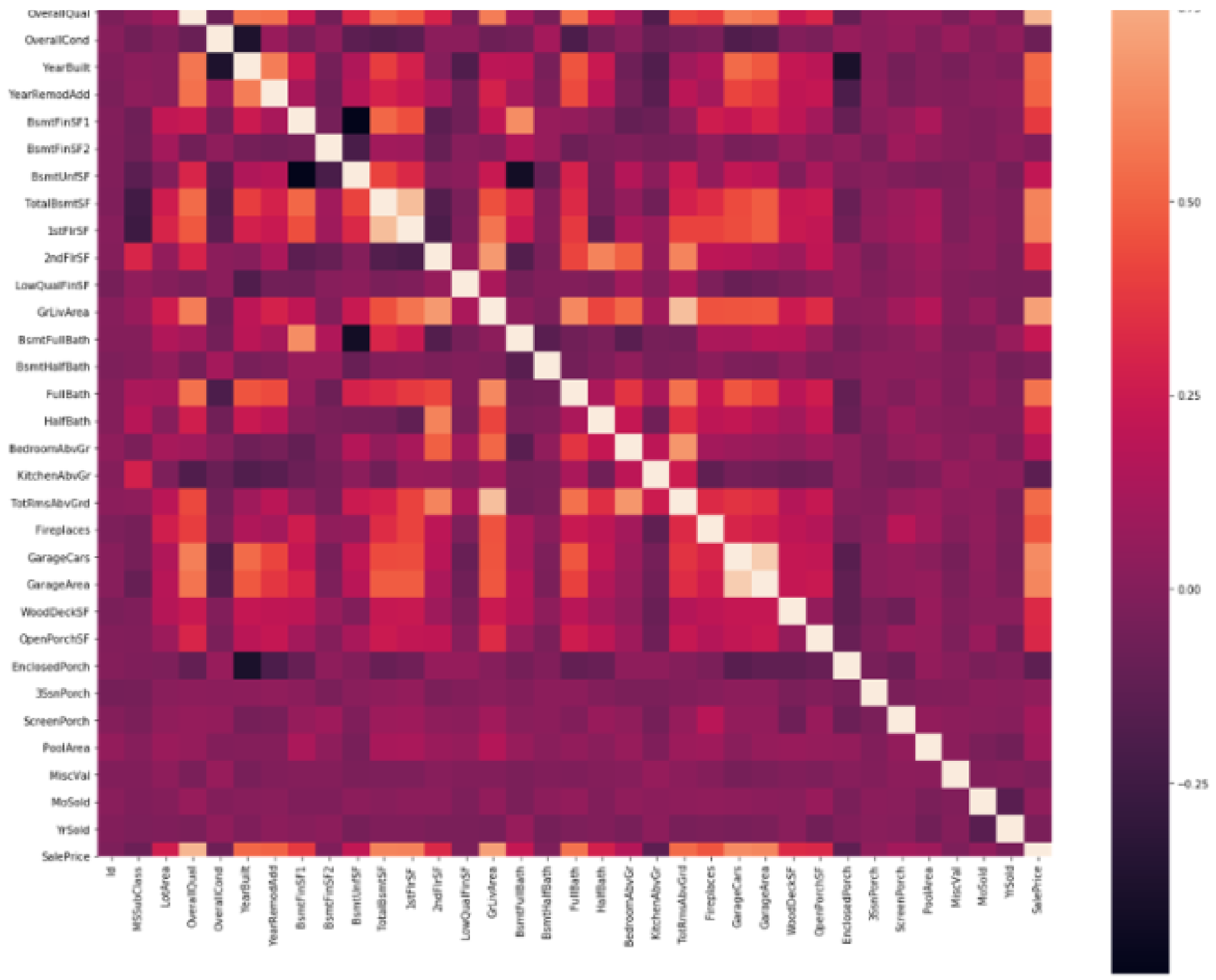


```
In [15]: len(df_data.columns)
Out[15]: 63
```

After null value treatment there are 63 columns

Lets check the correlation matrix to find any unforeseen relations

```
corr_mat=df_data.corr()

fi,ax=plt.subplots(figsize=(20,20))
sns.heatmap(corr_mat,square=True)
```

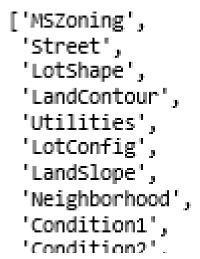


Though column 'ID' seems to have no correlation to all the columns but there is no use in modelling as its just row number. So we can remove it too.

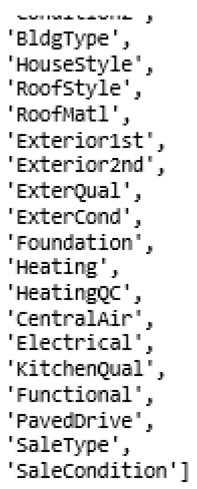
```
del df_data['Id']
```

There are many columns that are categorical. We need to one hot encode them.

```
le=LabelEncoder()
cat_mask= df_data.dtypes=='object'
cat_cols= df_data.columns[cat_mask].tolist()
cat_cols
```



```
['MSZoning',
 'Street',
 'LotShape',
 'LandContour',
 'Utilities',
 'LotConfig',
 'LandSlope',
 'Neighborhood',
 'Condition1',
```



```
'BldgType',
'HouseStyle',
'RoofStyle',
'RoofMatl',
'Exterior1st',
'Exterior2nd',
'ExterQual',
'ExterCond',
'Foundation',
'Heating',
'HeatingQC',
'CentralAir',
'Electrical',
'KitchenQual',
'Functional',
'PavedDrive',
'SaleType',
'SaleCondition']
```

All categorical columns in the data

```
#Lets convert the columns to one hot encoding
df_data[cat_cols]=df_data[cat_cols].apply(lambda x:
le.fit_transform(x.astype(str)))
df_data_c = df_data.copy()

#get_dummies is used for one hot encoding
df_data_c = pd.get_dummies(df_data_c,columns=cat_cols)
```

Now the data is ready for modelling. Before that we need to split the data into train and test.

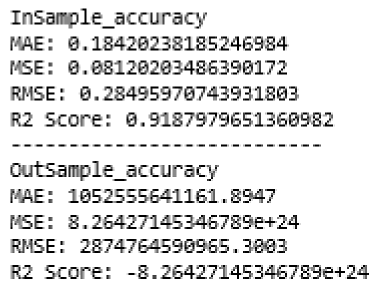
```
x_train, x_test, y_train, y_test =
train_test_split(df_data_c.drop(‘SalePrice’,axis=1),df_data_c[‘SaleP
rice’], test_size =0.25,random_state=120)
y_train= y_train.values.reshape(-1,1)
y_test= y_test.values.reshape(-1,1)
```

Normalize the values in train and test using Standard Scaler function

```
sc_X = StandardScaler()
sc_y = StandardScaler()
x_train = sc_X.fit_transform(x_train)
x_test = sc_X.fit_transform(x_test)
y_train = sc_X.fit_transform(y_train)
y_test = sc_y.fit_transform(y_test)
```

Let’s now fit a linear regression model on the data

```
lm = LinearRegression()
lm.fit(x_train,y_train)

#predictions on train data
x_pred = lm.predict(x_train)
x_pred = x_pred.reshape(-1,1)

#Prediction of validation data
y_predictions = lm.predict(x_test)
y_predictions= predictions.reshape(-1,1)

def scores_(y,x):
    print('MAE:', metrics.mean_absolute_error(y, x))
    print('MSE:', metrics.mean_squared_error(y, x))
    print('RMSE:', np.sqrt(metrics.mean_squared_error(y, x)))
    print('R2 Score:' ,metrics.r2_score(y,x))

print('InSample_accuracy')
scores_(y_train, x_pred)

print('---------------------------')
print('OutSample_accuracy')
scores_(y_test,y_pred)
```



```
InSample_accuracy
MAE: 0.18420238185246984
MSE: 0.08120203486390172
RMSE: 0.28495970743931803
R2 Score: 0.9187979651360982
---------------------------
OutSample_accuracy
MAE: 1052555641161.8947
MSE: 8.26427145346789e+24
RMSE: 2874764590965.3003
R2 Score: -8.26427145346789e+24
```

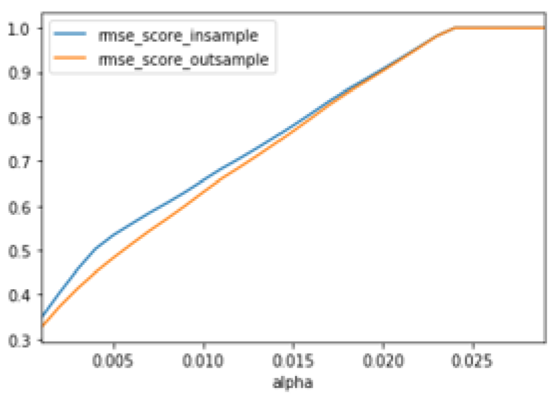
The model performed really well on training data with a good 0.92 r2 score and <1 RMSE score but with test data the performance is no where near good. This is a clear **overfitting** model. Reason might me because of numerous features. To tackle this we can perform Ridge and Lasso regularization.

Lasso or l1 regularization:

For a given range of alpha lets try to find out the RMSE scores of training(In sample) and test(Out sample) data sets.

```
def regularization(model,alpha_range):
    rmse_score_insample=[]
    rmse_score_outsample=[]
    r2_score_insample=[]
    r2_score_outsample=[]
    for i in alpha_range:
        regularization = model(alpha=i,normalize=True)
        regularization.fit(x_train,y_train)
        y_pred_train = regularization.predict(x_train)
        y_pred_train = y_pred_train.reshape(-1,1)
        y_pred_test=regularization.predict(x_test)
        y_pred_test = y_pred_test.reshape(-1,1)

rmse_score_insample.append(np.sqrt(metrics.mean_squared_error(y_trai
n,y_pred_train )))

rmse_score_outsample.append(np.sqrt(metrics.mean_squared_error(y_tes
t, y_pred_test)))
        r2_score_insample.append(metrics.r2_score(y_train,
y_pred_train))
        r2_score_outsample.append(metrics.r2_score(y_test,
y_pred_test))
    df=pd.DataFrame()
    df['alpha']=alpha_range
    df['rmse_score_insample'] = rmse_score_insample
    df['rmse_score_outsample']= rmse_score_outsample
    df['r2_score_insample'] = r2_score_insample
    df['r2_score_outsample'] = r2_score_outsample
    return df.plot(x = 'alpha', y = ['rmse_score_insample',
'rmse_score_outsample'])

alpha_range_lasso = np.arange(0.001,0.03,0.001)
print(regularization(Lasso,alpha_range_lasso))
```

```
AxesSubplot(0.125,0.125;0.775x0.755)
```



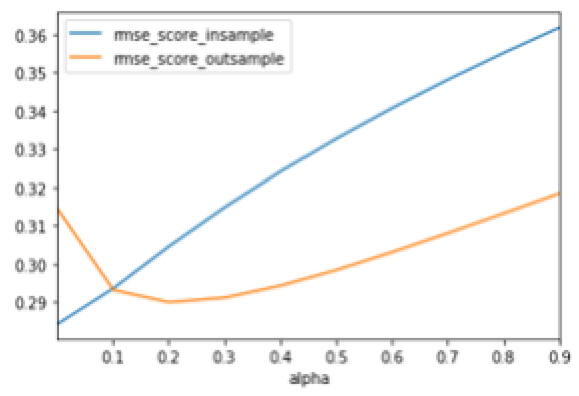
Convergence at alpha=0.024

We can see that there is no huge difference in in sample and out sample RMSE scores so Lasso has resolved overfitting. One observation here is that after alpha= 0.017 there is no difference in RMSE scores of In sample and Out sample. Let us also check for Ridge

Ridge or l2 regularization:

```
alpha_range_ridge = np.arange(0.001,1,0.1)
print(regularization(Ridge,alpha_range_ridge))

#writing functions helps reduce redundant lines of code as seen
#above we can just input the parametre Ridge or Lasso
```

```
AxesSubplot(0.125,0.125;0.775x0.755)
```



At alpha = 0.1 the model performs good

We see in the graph that around alpha=0.1 there is no much difference in the RMSE scores and clearly there is no sign of over fitting as there is very less difference of insample and outsample RMSE scores as compared to huge difference in Linear Regression.

By comparing Lasso and Ridge RMSE or R2 and we can pick the model that has good score as desired for the problem statement.

3. Final Code

```
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
import seaborn as sns
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn import metrics
from sklearn import ensemble
from sklearn.linear_model import Lasso,Ridge

#load train data
df_data=pd.read_csv("data_price.csv")
df_data.head()

#to know each and every column execute the following
print(df_data.columns)
print(df_data.shape)

total = df_data.isnull().sum().sort_values(ascending=False)
percent =
(df_data.isnull().sum()/df_data.isnull().count()).sort_values(ascend
ing=False)
missing_data = pd.concat([total, percent], axis=1, keys=['Total',
'Percent'])
missing_data.head(20)

df_data=
df_data.drop(missing_data[missing_data['Total']>1].index.values,1)
df_data=
df_data.drop(df_data.loc[df_data['Electrical'].isnull()].index)

corr_mat=df_data.corr()

fi,ax=plt.subplots(figsize=(20,20))
sns.heatmap(corr_mat,square=True)

del df_data['Id']

le=LabelEncoder()
cat_mask= df_data.dtypes=='object'
cat_cols= df_data.columns[cat_mask].tolist()
cat_cols

#Lets convert the columns to one ht encoding
df_data[cat_cols]=df_data[cat_cols].apply(lambda x:
le.fit_transform(x.astype(str)))
df_data_c = df_data.copy()

#get_dummies is used for one hot encoding
df_data_c = pd.get_dummies(df_data_c,columns=cat_cols)

x_train, x_test, y_train, y_test =
train_test_split(df_data_c.drop('SalePrice',axis=1),df_data_c['SaleP
rice'], test_size =0.25,random_state=120)
y_train= y_train.values.reshape(-1,1)
y_test= y_test.values.reshape(-1,1)

sc_X = StandardScaler()
sc_y = StandardScaler()
x_train = sc_X.fit_transform(x_train)
x_test = sc_X.fit_transform(x_test)
y_train = sc_X.fit_transform(y_train)
y_test = sc_y.fit_transform(y_test)

#Linear Regression
lm = LinearRegression()
lm.fit(x_train,y_train)

#predictions on train data
x_pred = lm.predict(x_train)
x_pred = x_pred.reshape(-1,1)
```

```
#Prediction of test data
y_pred = lm.predict(x_test)
y_pred= y_pred.reshape(-1,1)

def scores_(y,x):
    print('MAE:', metrics.mean_absolute_error(y, x))
    print('MSE:', metrics.mean_squared_error(y, x))
    print('RMSE:', np.sqrt(metrics.mean_squared_error(y, x)))
    print('R2 Score:' ,metrics.r2_score(y,x))

print('InSample_accuracy')
scores_(y_train, x_pred)
print('---------------------------')

print('OutSample_accuracy')
scores_(y_test,y_pred)

def regularization_model(model,alpha_range):
 rmse_score_insample=[]
 rmse_score_outsample=[]
 r2_score_insample=[]
 r2_score_outsample=[]
 for i in alpha_range:
 regularization = model(alpha=i,normalize=True)
 regularization.fit(x_train,y_train)
 y_pred_train = regularization.predict(x_train)
 y_pred_train = y_pred_train.reshape(-1,1)
 y_pred_test=regularization.predict(x_test)
 y_pred_test = y_pred_test.reshape(-1,1)

rmse_score_insample.append(np.sqrt(metrics.mean_squared_error(y_trai
n,y_pred_train )))

rmse_score_outsample.append(np.sqrt(metrics.mean_squared_error(y_tes
t, y_pred_test)))
 r2_score_insample.append(metrics.r2_score(y_train, y_pred_train))
 r2_score_outsample.append(metrics.r2_score(y_test, y_pred_test))

df=pd.DataFrame()
 df[‘alpha’]=alpha_range
 df[‘rmse_score_insample’] = rmse_score_insample
 df[‘rmse_score_outsample’]= rmse_score_outsample
 df[‘r2_score_insample’] = r2_score_insample
 df[‘r2_score_outsample’] = r2_score_outsample
 return df.plot(x = ‘alpha’, y =
[‘rmse_score_insample’,’rmse_score_outsample’])

alpha_range_lasso = np.arange(0.001,0.03,0.001)
print(regularization_model(Lasso,alpha_range_lasso))

alpha_range_ridge = np.arange(0.001,1,0.1)
print(regularization_model(Ridge,alpha_range))
```

References: An Introduction to Statistical Learning: With Applications in R

Thank You!

Machine Learning Bias Variance Tradeoff Overfitting Ridge Regularization