Lab Course Machine Learning
Exercise 6

1.1 Data preprocessing:

Function name : read_data
Parameter : Filename and column names

This function reads the data and returns the data in the format of data frame along with the headings.

```
def read_data(filename,columns):
    if columns == None:
        data = pd.read_csv(filename, sep='\s+', delimiter = ";")
        return data
    data = pd.read_csv(filename, sep='\s+',header = None)
    data.columns = columns
    return data
```

Output:

Wine data

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
706	7.0	0.780	0.08	2.0	0.093	10.0	19.0	0.99560	3.40	0.47	10.0	5
67	6.6	0.705	0.07	1.6	0.076	6.0	15.0	0.99620	3.44	0.58	10.7	5
1089	11.6	0.410	0.54	1.5	0.095	22.0	41.0	0.99735	3.02	0.76	9.9	7
382	8.3	0.260	0.42	2.0	0.080	11.0	27.0	0.99740	3.21	0.80	9.4	6
589	10.2	0.290	0.49	2.6	0.059	5.0	13.0	0.99760	3.05	0.74	10.5	7

Function name : check_na_null
Parameter : Dataframe and column names

This function checks the presence of null values or NA or nan values within the dataset and removes the same. This function also generates dummies for non-numeric values and returns the same.

```
def check_na_null(data,column_dummies):
    data.isnull().values.any()
    data.dropna(inplace = True)
    data = pd.get_dummies(data=data, columns=column_dummies)
    return data
```

Function name : text_to_number
Parameter : Dataframe and column names

This function converts the text within a column to numbers. For example, a city with name ABQ will be converted to a specific number for all the available instances.

```
def text_to_number(data,column):
    for i in column:
        Airdata[i] = pd.Categorical(Airdata[i])
        Airdata[i] = Airdata[i].cat.codes
    return data
```

Function name : normalise_data
Parameter : Data to be normalised

Reference: http://mathworld.wolfram.com/FrobeniusNorm.html

Numpy.linalg.norm function returns the normalisation of the data. By default, it returns Forbenius norm of the matrix.

$$A_{norm} = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|A_{i,j}|^2}$$

```
def normalise_data(data):
    norm_data = data.copy()
    for columns in norm_data:
        norm_data[columns] = norm_data[columns]/np.linalg.norm(norm_
    return norm_data
```

Output:

Normalilsed Wine data

	volatile acidity	citric acid	density	sulphates	alcohol	quality
706	0.034998	0.005995	0.024979	0.017295	0.023869	0.021962
67	0.031633	0.005246	0.024994	0.021342	0.025539	0.021962
1089	0.018396	0.040469	0.025023	0.027966	0.023630	0.030746
382	0.011666	0.031476	0.025024	0.029438	0.022436	0.026354
589	0.013012	0.036722	0.025029	0.027230	0.025062	0.030746

Function name : create_Test_Train_data
Parameter : Data of input variables and target

This function splits the input variables to Train and Test data with 80% and 20% of the entire data respectively.

```
def create_Test_Train_data(X_data,Y_data):
    Y_train = Y_data[:math.ceil(0.8*len(Y_data))]
    Y_test = Y_data[math.ceil(0.8*len(Y_data)):]
    X_train = X_data[:math.ceil(0.8*len(X_data))]
    X_test = X_data[math.ceil(0.8*len(X_data)):]
    return Y_train,Y_test,X_train,X_test
```

Output:

Wine data

```
Xtrain:  (1280, 5)
Ytrain:  (1280,)
Xtest:  (319, 5)
Ytest:  (319,)
```

Generalized Linear Models with Scikit Learn:

Wine dataset:

To begin with the problem, first we've to find the correlation between the input variables and the quality of the red wine. As the dataset contains numerical values, we can use the Pearson co-efficient correlation between two variables.

Function name : pearson_coefficient
Parameter : Ranked data for correlation

Reference: *https://statistics.laerd.com/statistical-guides/pearson-correlation-coefficient-statistical-guide.php*

The Pearson correlation coefficient (PCC) is a measure of the linear correlation between two variables X and Y. PCC or 'r' has a value between +1 and −1.

Assumptions:

1. The variables must be either interval or ratio measurements.
2. The variables must be approximately normally distributed.
3. There is a linear relationship between the two variables
4. Outliers are either kept to a minimum or are removed entirely.
5. There is homoscedasticity of the data.

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

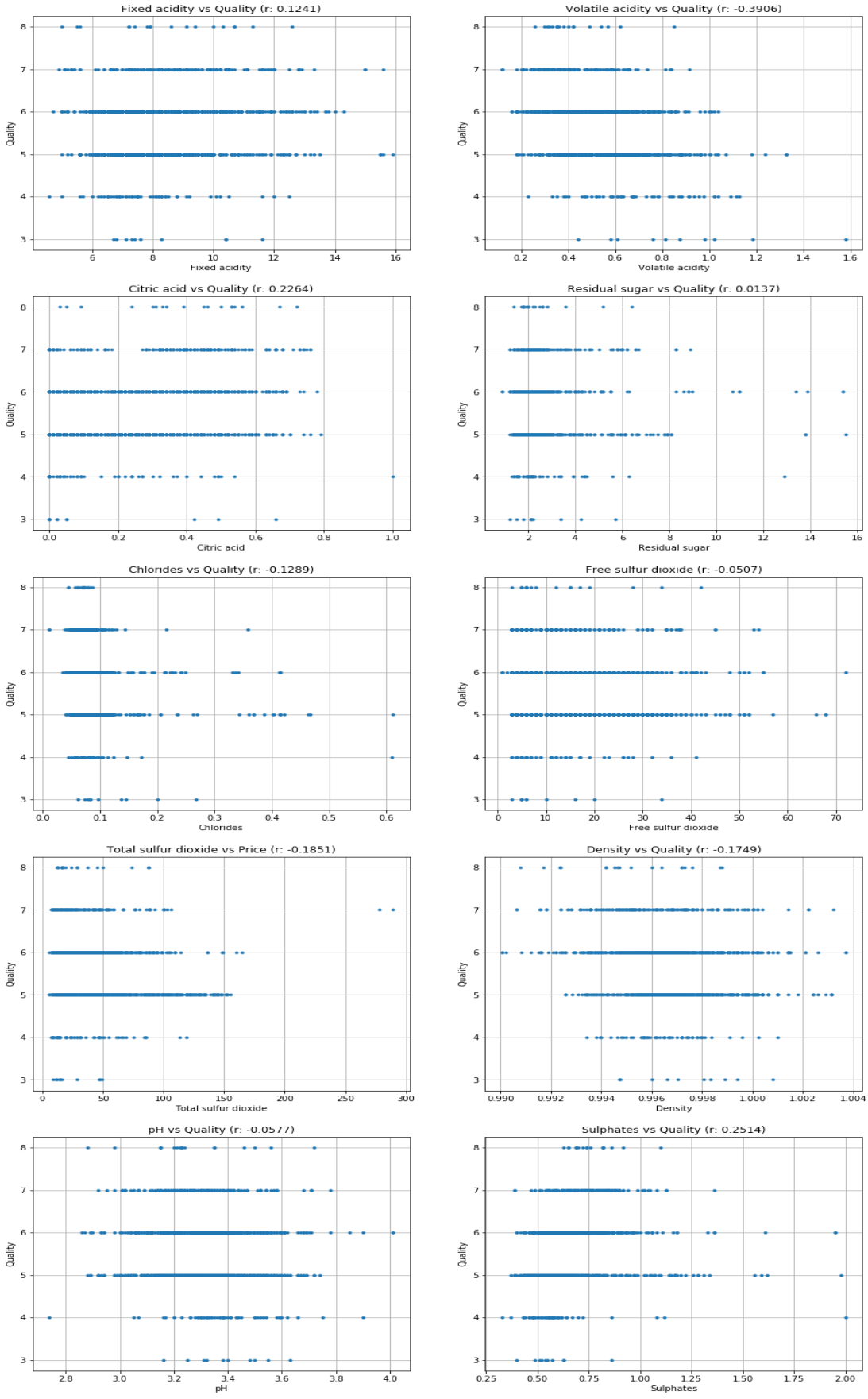
```
def pearson_coefficient(x,y):
    x = x - np.mean(x)
    y = y - np.mean(y)
    return (np.sum(x*y)/np.sqrt(np.sum(x*x)*np.sum(y*y)))
```

Output:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	0.1241	-0.3906	0.2264	0.0137	-0.1289	-0.0507	-0.1851	-0.1749	-0.0577	0.2514	0.4762	NaN

From the correlation between the parameters and quality, we can conclude that there is no high relation between residual sugar, free sulphur dioxide, pH and quality. Hence, I am dropping out these columns from the dataset to be used for creating the model. After creating the model, as it could be seen that there's poor correlation between fixed acidity, chlorides, total sulphur dioxide and quality. At last, these columns are also dropped down from the dataset.

Correlation between variables and Quality



Data after dropping columns

	volatile acidity	citric acid	density	sulphates	alcohol	quality
0	0.70	0.00	0.9978	0.56	9.4	5
1	0.88	0.00	0.9968	0.68	9.8	5
2	0.76	0.04	0.9970	0.65	9.8	5
3	0.28	0.56	0.9980	0.58	9.8	6
4	0.70	0.00	0.9978	0.56	9.4	5

Normalized wine data

	volatile acidity	citric acid	density	sulphates	alcohol	quality
706	0.034998	0.005995	0.024979	0.017295	0.023869	0.021962
67	0.031633	0.005246	0.024994	0.021342	0.025539	0.021962
1089	0.018396	0.040469	0.025023	0.027966	0.023630	0.030746
382	0.011666	0.031476	0.025024	0.029438	0.022436	0.026354
589	0.013012	0.036722	0.025029	0.027230	0.025062	0.030746

Creating a Pipeline:

Reference: https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html

The purpose of the pipeline is to assemble several steps that can be cross-validated together while setting different parameters. For this, it enables setting parameters of the various steps using their names and the parameter name separated by a '__'. A step's estimator may be replaced entirely by setting the parameter with its name to another estimator, or a transformer removed by setting to None.

```
pipe = Pipeline([('sgd', linear_model.SGDRegressor())])
param = {"sgd__learning_rate":"constant", "sgd__penalty":None}
```

Pipeline:

```
Pipeline(memory=None,
     steps=[('sgd', SGDRegressor(alpha=0.0001, average=False, early_stopping=False, epsilon=0.1,
       eta0=0.01, fit_intercept=True, l1_ratio=0.15,
       learning_rate='invscaling', loss='squared_loss', max_iter=None,
       n_iter=None, n_iter_no_change=5, penalty='l2', power_t=0.25,
       random_state=None, shuffle=True, tol=None, validation_fraction=0.1,
       verbose=0, warm_start=False))])
```

Non – Regularized:

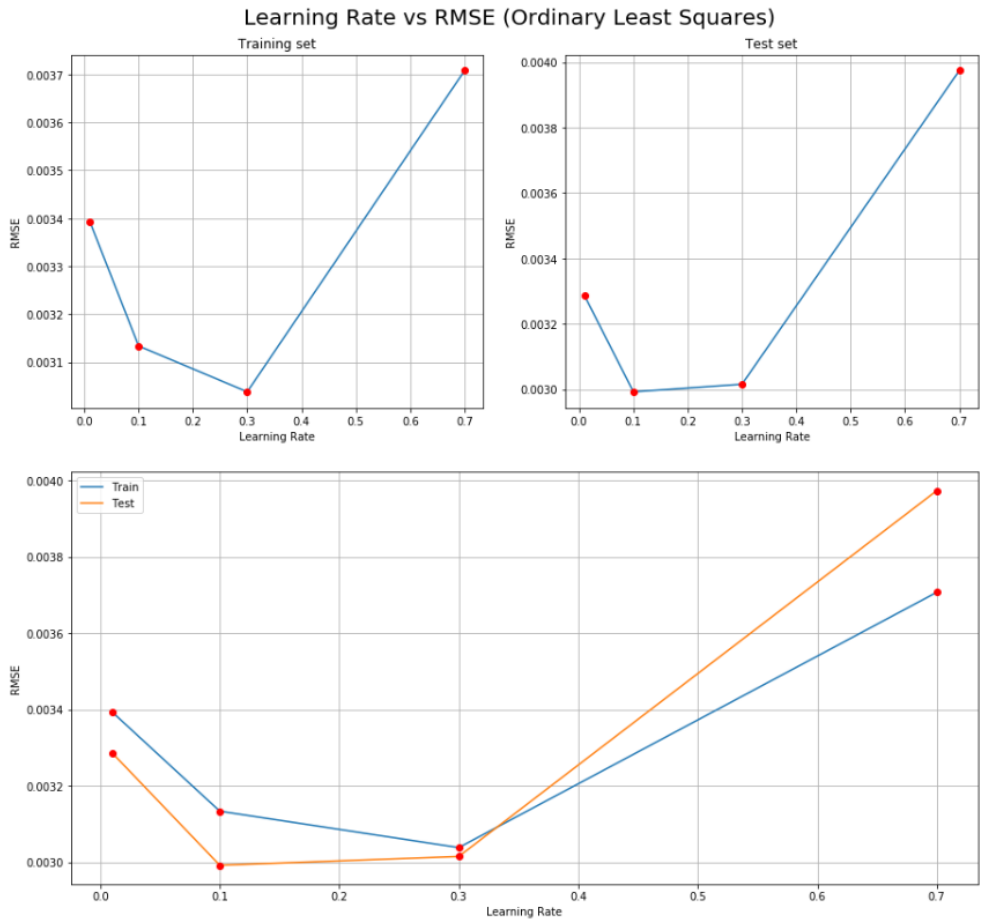
Training the model with Ordinary Least Squares (Linear Regression) method:

To train the model with linear regression, Learning Rate is used as a hyper paramter and 3 sets of iterations (100,300,700) were performed

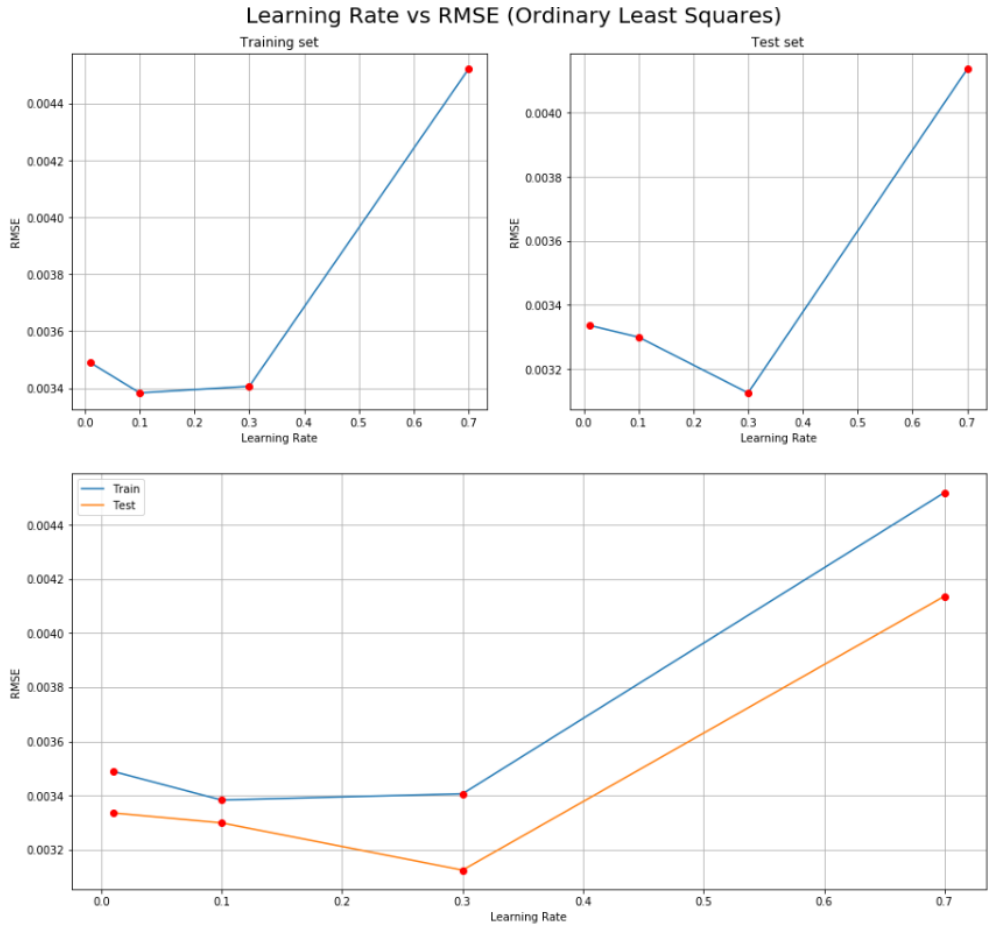
```
RMSE_train_linear = list()
RMSE_test_linear = list()
alpha = [0.01,0.1,0.3,0.7]##Learning Rate as Hyperparameter
```

```
for i in alpha:
        param["sgd__eta0"] = i
        param["sgd__max_iter"] = 700
        pipe.set_params(**param)##Set parameters to the pipeline
        pipe.fit(X_train,Y_train)
        y_pred_train = pipe.predict(X_train)
        y_pred_test = pipe.predict(X_test)
        RMSE_train_linear.append(np.sqrt(mean_squared_error(Y_train,y_pred_train)))
        RMSE_test_linear.append(np.sqrt(mean_squared_error(Y_test,y_pred_test)))
```

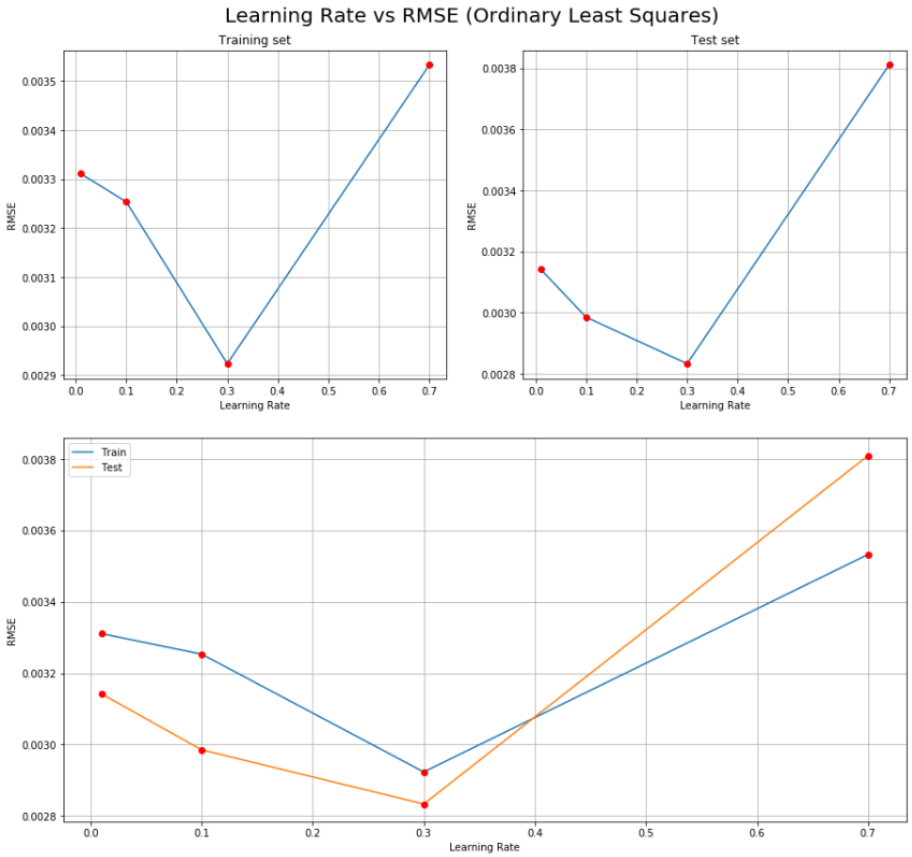
Iteration : 100



Iteration : 300



Iteration: 700



Regularized:

Training the model with Ridge Regression method:

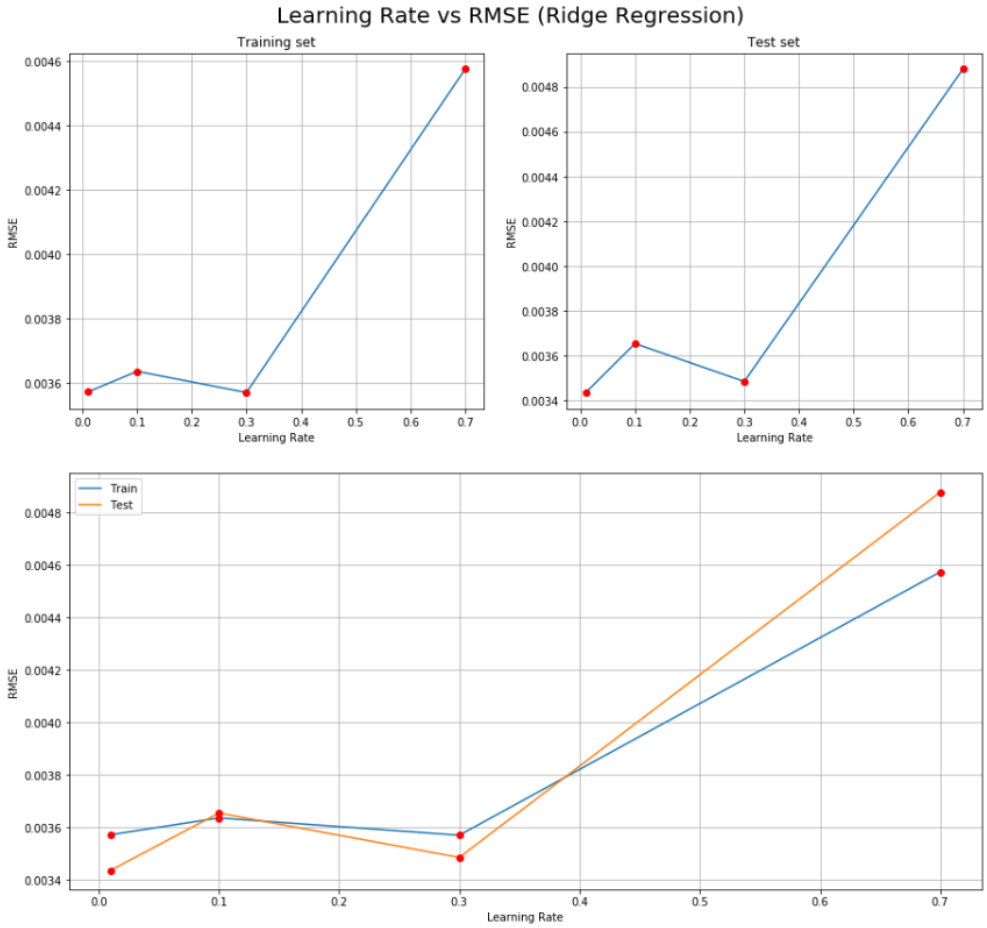
To train the model with ridge regression (L2 Loss), Learning Rate and Regularization constant are used as a hyper paramter.

Learning Rate: [0.01,0.1,0.3,0.7]
Regularization constant: [0.1,0.001,0.0001]

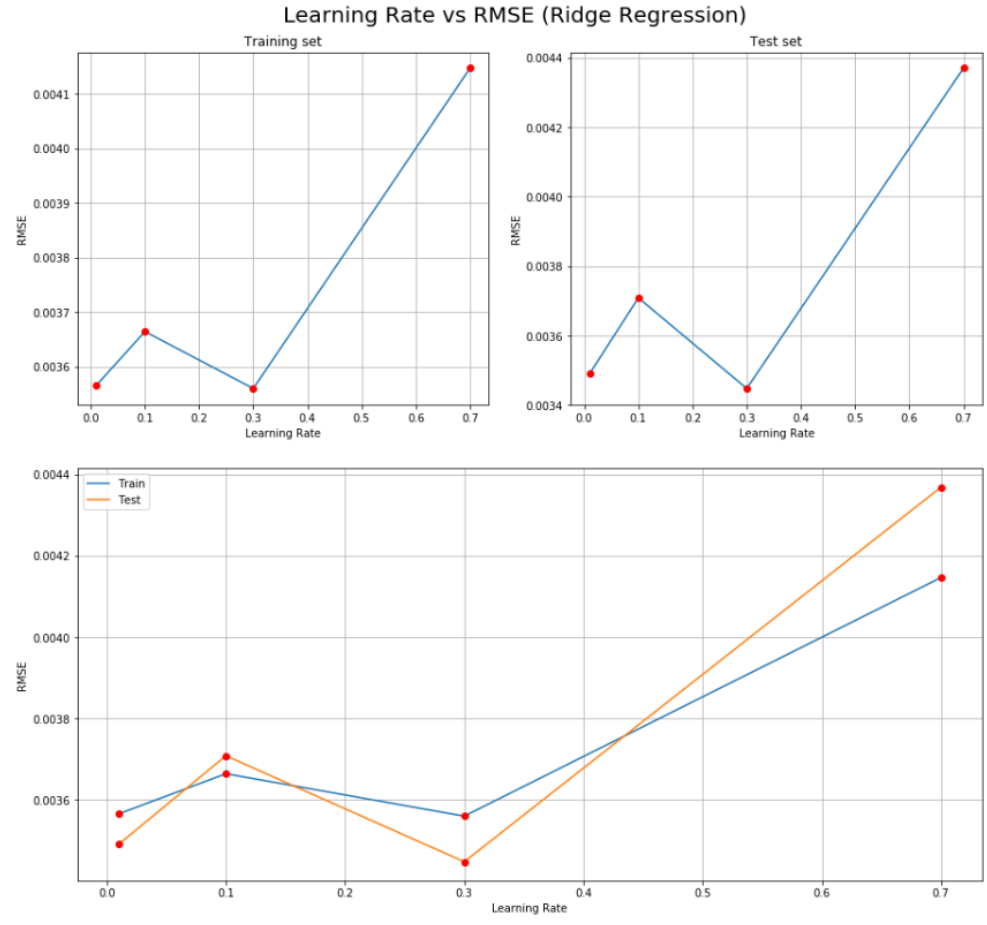
```
##Reference:https://stackoverflow.com/questions/27122757/
###sklearn-set-params-takes-exactly-1-argument/29028601
alpha = [0.01,0.1,0.3,0.7]
```

```
RMSE_train_ridge = list()
RMSE_test_ridge = list()
for i in alpha:
        param["sgd__penalty"] = 'l2'
        param["sgd__alpha"] = 0.1
        param["sgd__eta0"] = i
        param["sgd__max_iter"] = 700
        pipe.set_params(**param)
        pipe.fit(X_train,Y_train)
        y_pred_train = pipe.predict(X_train)
        y_pred_test = pipe.predict(X_test)
        RMSE_train_ridge.append(np.sqrt(mean_squared_error(Y_train,y_pred_train)))
        RMSE_test_ridge.append(np.sqrt(mean_squared_error(Y_test,y_pred_test)))
```

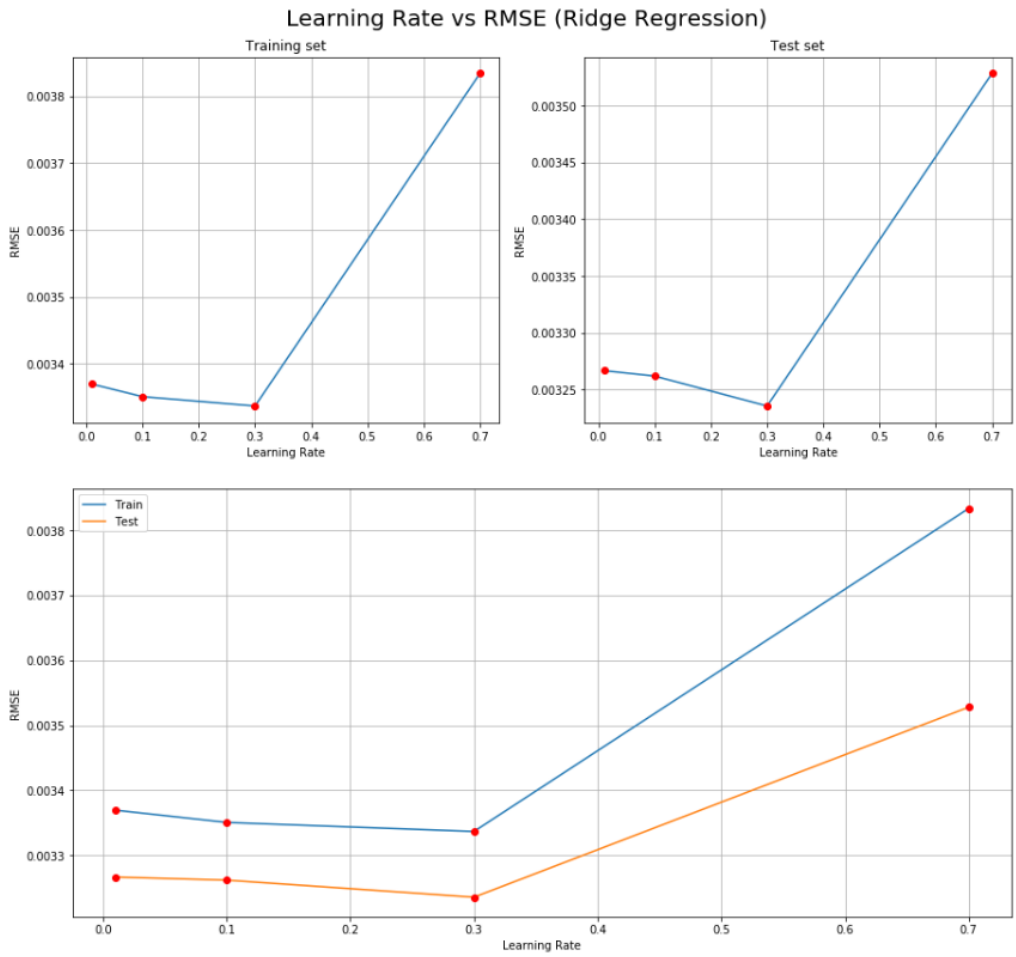
Regularization Constant : 0.1



Regularization Constant : 0.01



Regularization Constant : 0.0001



Training the model with Ridge Regression method:

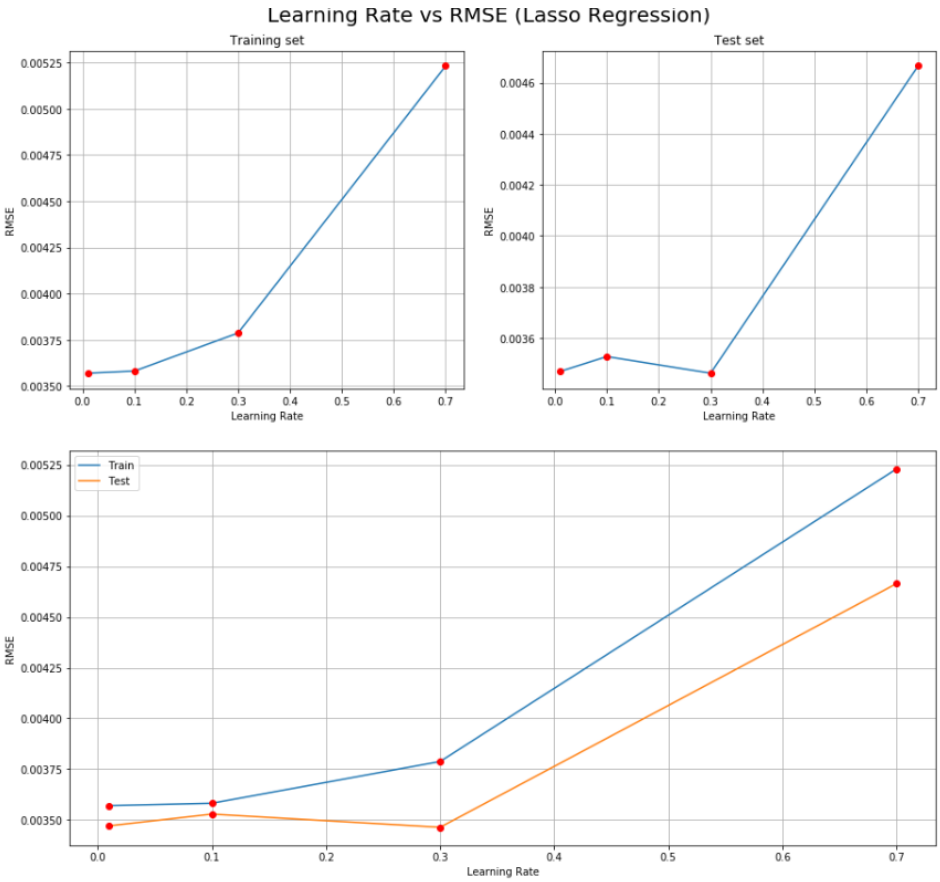
To train the model with ridge regression (L1 Loss), Learning Rate and Regularization constant are used as hyper paramters.

Learning Rate: [0.01,0.1,0.3,0.7]
Regularization constant: [0.1,0.001,0.0001]

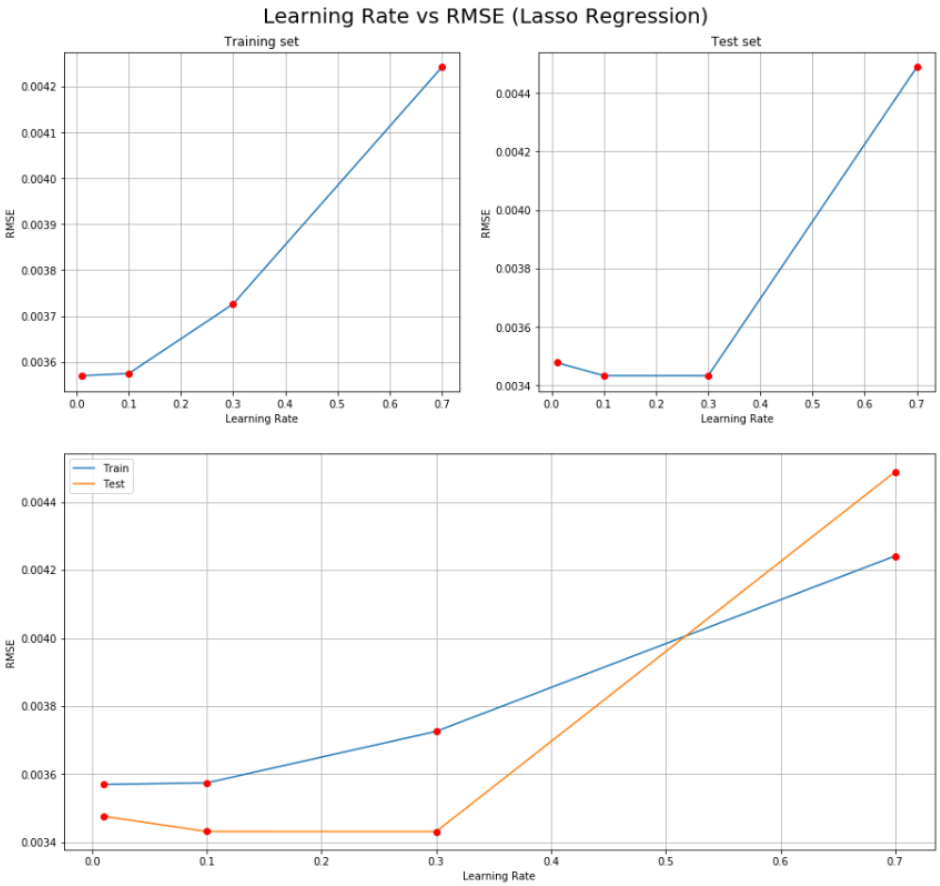
```
RMSE_train_lasso = list()
RMSE_test_lasso = list()
alpha = [0.01,0.1,0.3,0.7]
```

```
for i in alpha:
        param["sgd__penalty"] = 'l1'
        param["sgd__alpha"] = 0.1
        param["sgd__eta0"] = i
        param["sgd__max_iter"] = 500
        pipe.set_params(**param)
        pipe.fit(X_train,Y_train)
        y_pred_train = pipe.predict(X_train)
        y_pred_test = pipe.predict(X_test)
        RMSE_train_lasso.append(np.sqrt(mean_squared_error(Y_train,y_pred_train)))
        RMSE_test_lasso.append(np.sqrt(mean_squared_error(Y_test,y_pred_test)))
```

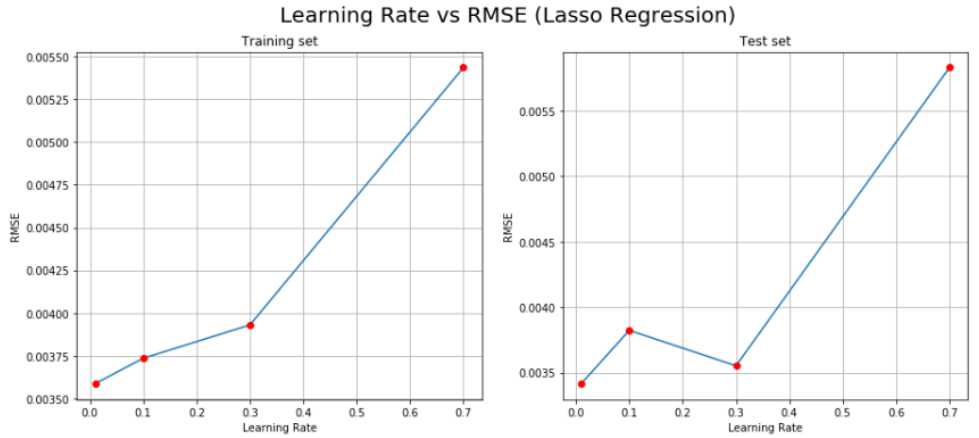
Regularization Constant : 0.1

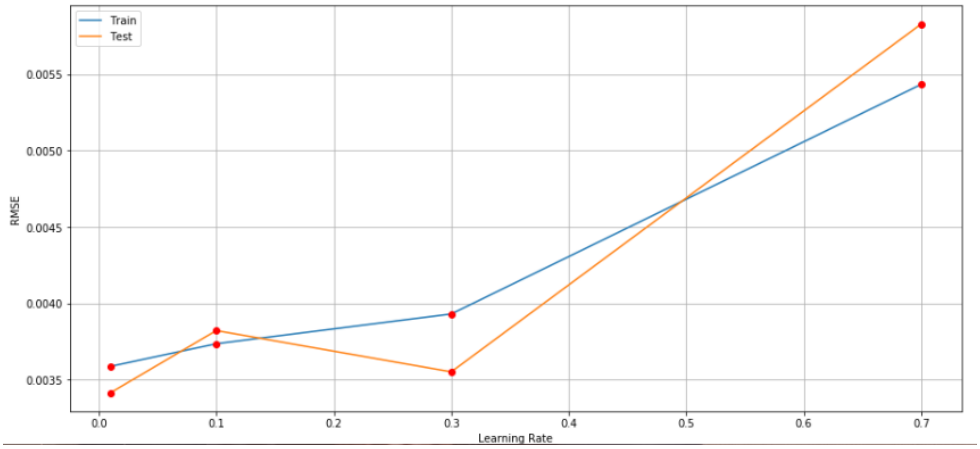


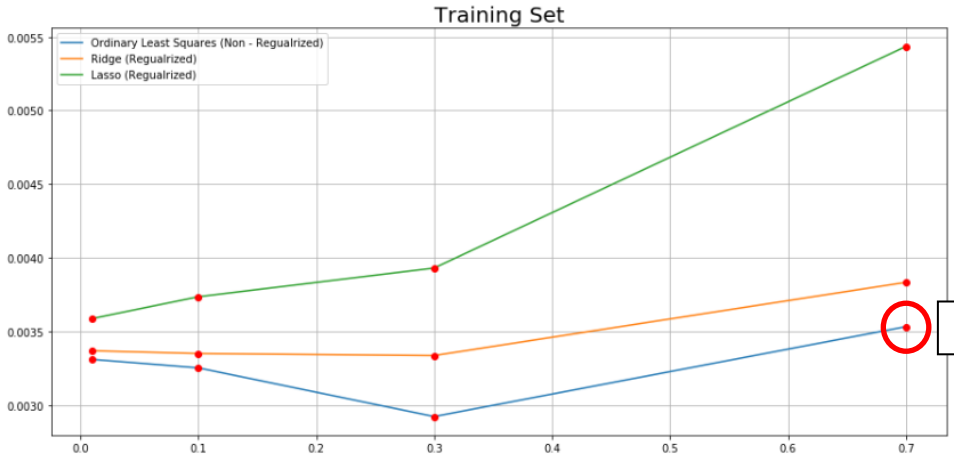
Regularization Constant : 0.01



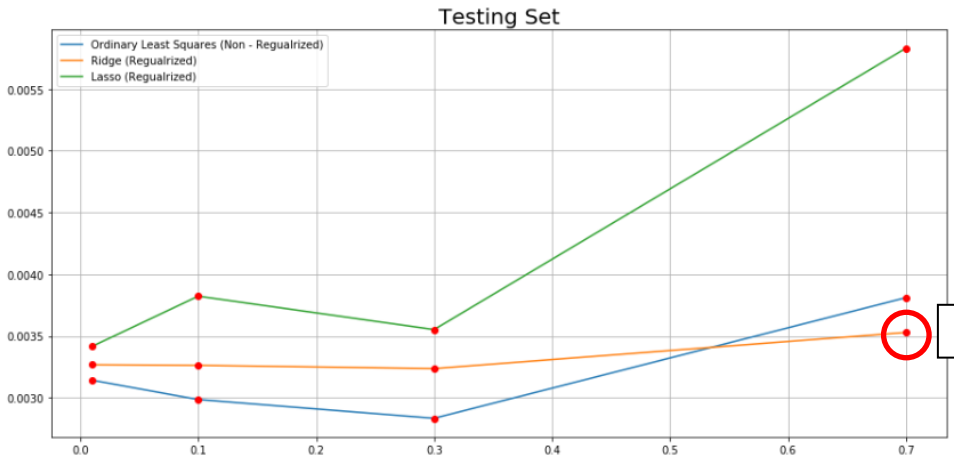
Regularization Constant : 0.0001







Best Model



Best Model

Combining all the 3 models generated using Linear Regression, Ridge Regression and Lasso Regression for both the training and testing set, we can see that for the training set, the linear regression model performs well than the other two models but on the contrary, it has failed to perform well for the testing set. This might be because the model is overfit and as a result of which it performs well for the training data. Comparing the regularized models, Ridge Regression with L2 loss performs better for both the training and testing datase. Hence, it can be seen that regularized models perfom well than non-regularized models and can be used to avoid overfitting or underfitting of the data.

Tuning the hyperparameters using scikit learn GridSearchCV:

Grid Parameters:

```
gridparameters= [{"sgd__learning_rate" : ["constant"],"sgd__eta0" :[0.01,0.1,0.3,0.7],"sgd__penalty" : [None]},
    {"sgd__learning_rate":["constant"],"sgd__penalty":["l2"],"sgd__alpha":[0.1,0.0001,0.001],"sgd__eta0":[0.01,0.1,0.3,0.7]},
    {"sgd__learning_rate":["constant"],"sgd__penalty":["l1"],"sgd__alpha":[0.1,0.0001,0.001],"sgd__eta0":[0.01,0.1,0.3,0.7]}]
```

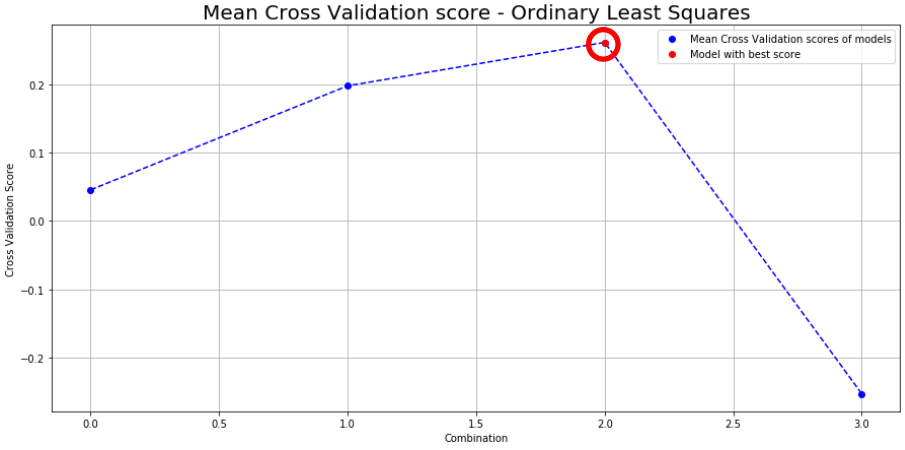
Grid Search for Linear Regression:

```
gridlinreg= GridSearchCV(pipe,  param_grid = gridparameters[0], cv = 4, n_jobs = 2) ##Refit is always True
gridlinreg.fit(X_train, Y_train)
```

```
Results:

mean_test_score : [ 0.0456405   0.19792477  0.26148466 -0.2528563 ]
rank_test_score : [3 2 1 4]
split1_train_score : [ 0.13127694  0.24401041  0.33884247 -0.05995475]
std_fit_time : [0.01144841 0.02814929 0.01353072 0.01351795]
split0_test_score : [-0.01285899  0.08748123  0.09828425  0.10032937]
mean_train_score : [ 0.12323913  0.21696385  0.30765503 -0.14375937]
std_test_score : [0.06879347 0.07175803 0.10075136 0.34159381]
params : [{'sgd__eta0': 0.01, 'sgd__learning_rate': 'constant', 'sgd__penalty': None}, {'sgd__eta0': 0.1, 'sgd__learning_rate':
'constant', 'sgd__penalty': None}, {'sgd__eta0': 0.3, 'sgd__learning_rate': 'constant', 'sgd__penalty': None}, {'sgd__eta0': 0.
7, 'sgd__learning_rate': 'constant', 'sgd__penalty': None}]
std_score_time : [0.00173254 0.         0.00676624 0.00676954]
std_train_score : [0.01045482 0.04045446 0.02915582 0.36696036]
split0_train_score : [0.13576074 0.25131512 0.31104016 0.33482876]
param_sgd__learning_rate : ['constant' 'constant' 'constant' 'constant']
split2_train_score : [ 0.11146424  0.22334512  0.32052252 -0.69368146]
mean_fit_time : [0.20301372 0.1530363  0.14843744 0.1294933 ]
mean_score_time : [0.00100029 0.         0.00390649 0.0039084 ]
split3_train_score : [ 0.11445461  0.14918476  0.26021497 -0.15623001]
param_sgd__eta0 : [0.01 0.1 0.3 0.7]
param_sgd__penalty : [None None None None]
split2_test_score : [ 0.13356287  0.21667582  0.34406499 -0.79010531]
split3_test_score : [-0.0296712   0.28761007  0.34595162 -0.02586341]
split1_test_score : [ 0.09152931  0.19993194  0.25763777 -0.29578583]
```

Output:

```
Best model after tuning the hyper parameters using Grid Search:
{'sgd__eta0': 0.3, 'sgd__learning_rate': 'constant', 'sgd__penalty': None}


Best score: 0.26148465875980553

Number of Folds: 4
```



Grid Search with Ridge Regression:

```
gridridreg= GridSearchCV(pipe,  param_grid = gridparameters[1], cv = 4, n_jobs = 2) ##Refit is always True
gridridreg.fit(X_train, Y_train)
```

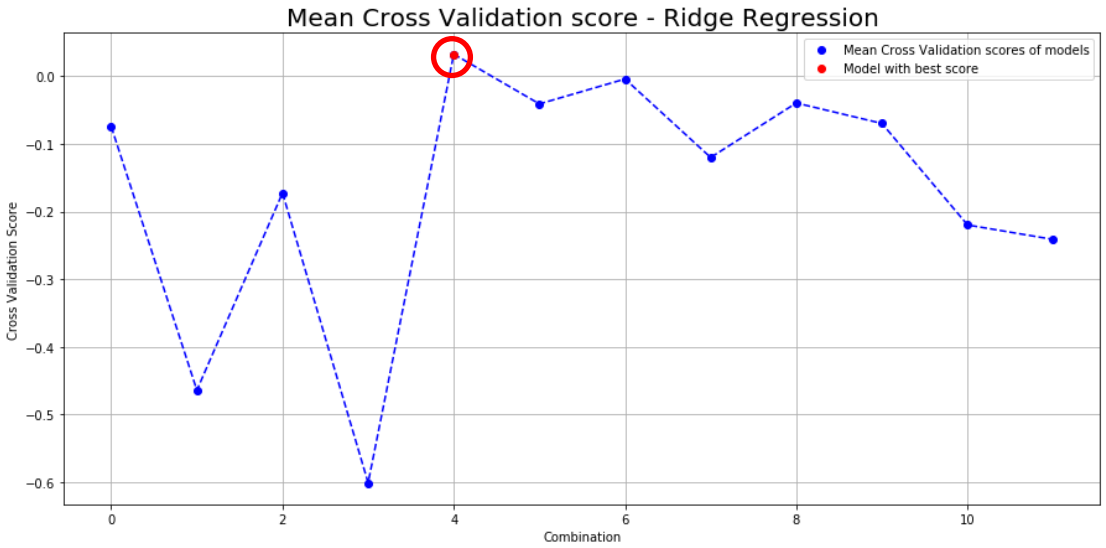
Results:

```
rank_test_score : [ 6 11  8 12  1  4  2  7  3  5  9 10]

std_train_score : [0.00575877 0.08168936 0.09027264 0.5601989  0.00982469 0.0474237
0.03099    0.14111969 0.00275688 0.05872465 0.11578679 0.2190932 ]

split1_train_score : [-0.00748801 -0.17516151 -0.18851941 -0.19472431  0.11485024  0.11372189
 0.14361932  0.09376448  0.03755298  0.00410718 -0.18749633 -0.39217   ]

param_sgd__learning_rate : ['constant' 'constant' 'constant' 'constant' 'constant' 'constant'
'constant' 'constant' 'constant' 'constant' 'constant' 'constant']

mean_fit_time : [0.19396579 0.16115004 0.1249994  0.15111613 0.19801235 0.15509337
0.2036404  0.14101702 0.15792185 0.14249957 0.15827239 0.13956088]

mean_score_time : [0.0020026  0.         0.         0.00100094 0.00200725 0.0049051
0.00490868 0.00200635 0.00494158 0.00099909 0.00782734 0.00884533]

split2_train_score : [-0.01582306 -0.03659344 -0.20615762 -1.35111966  0.09192048  0.01746801
 0.06652366  0.11407233  0.03384064  0.00806591  0.03179668 -0.01202846]

split0_train_score : [ 4.42508797e-04 -2.32253147e-01 -3.20080597e-02 -5.01561054e-03
 1.12864903e-01  1.44802889e-01  1.40433214e-01 -1.93732826e-01
 3.68149074e-02  2.70121056e-02 -2.84264322e-01 -6.20419507e-01]

param_sgd__alpha : [0.1 0.1 0.1 0.1 0.0001 0.0001 0.0001 0.0001 0.001 0.001 0.001 0.001]

mean_test_score : [-0.07453174 -0.46426892 -0.17256145 -0.60113308  0.03260744 -0.04092735
-0.00406663 -0.12008379 -0.03958437 -0.06977096 -0.21988399 -0.24132539]

std_fit_time : [0.01759626 0.0217023  0.01104947 0.01060604 0.05079661 0.0146761
0.00504524 0.00998933 0.03087171 0.01372003 0.0180539  0.02512663]

split0_test_score : [-0.20605598 -1.00595732 -0.05689184 -0.13074904 -0.02529629 -0.11093416
-0.13814614 -0.01405518 -0.10944598 -0.27602182 -0.02468534 -0.23719576]

mean_train_score : [-0.00779164 -0.12482793 -0.10780663 -0.38995386  0.1042484   0.08811361
 0.11859116 -0.0365183   0.03469482 -0.02047256 -0.13675104 -0.35759206]

std_test_score : [0.08273575 0.32859583 0.12451422 0.62292267 0.04188668 0.10979369
0.08203693 0.30363453 0.04847233 0.12814316 0.23171673 0.22087171]

param_sgd__penalty : ['l2' 'l2' 'l2' 'l2' 'l2' 'l2' 'l2' 'l2' 'l2' 'l2' 'l2' 'l2']

split1_test_score : [-7.02371748e-03 -3.48839056e-01 -6.54870643e-02 -3.75917767e-01
 7.05102738e-02  3.48791639e-02  8.40181187e-02  7.72374438e-02
 1.44636474e-04 -7.82859463e-02 -3.49269296e-01 -6.02151270e-01]

split2_test_score : [-8.35387122e-04 -1.20177175e-01 -3.62557026e-01 -1.66942505e+00
 7.44551187e-02  9.32024048e-02  2.68752937e-02  9.74210522e-02
 1.07327521e-02  3.55418908e-02  2.80667791e-02 -8.85581181e-02]

split3_test_score : [-0.08421189 -0.38210211 -0.20530987 -0.22844045  0.01076067 -0.1808568
 0.01098621 -0.64093849 -0.05976889  0.03968204 -0.53364811 -0.0373964 ]

split3_train_score : [-0.00829799 -0.05530361 -0.00454143 -0.00895585  0.09735799  0.07646165
 0.12378845 -0.1601772   0.03057074 -0.12107543 -0.10704018 -0.40575029]
```

Output:

```
Best model after tuning the hyper parameters using Grid Search:
{'sgd__eta0': 0.3, 'sgd__learning_rate': 'constant', 'sgd__penalty': None}


Best score: 0.032607443328035834

Number of Folds: 4
```

```
plt.figure(figsize = (15, 7))
plt.title("Mean Cross Validation score - Lasso Regression",fontsize = 20)
plt.plot(scores,'b--')
plt.plot(scores, "bo", label = "Mean Cross Validation scores of models")
plt.plot(scores.index(bestscore),bestscore, "ro",label = "Model with best score")
plt.xlabel('Combination')
plt.ylabel('Cross Validation Score')
plt.legend()
plt.grid()
plt.show()
```



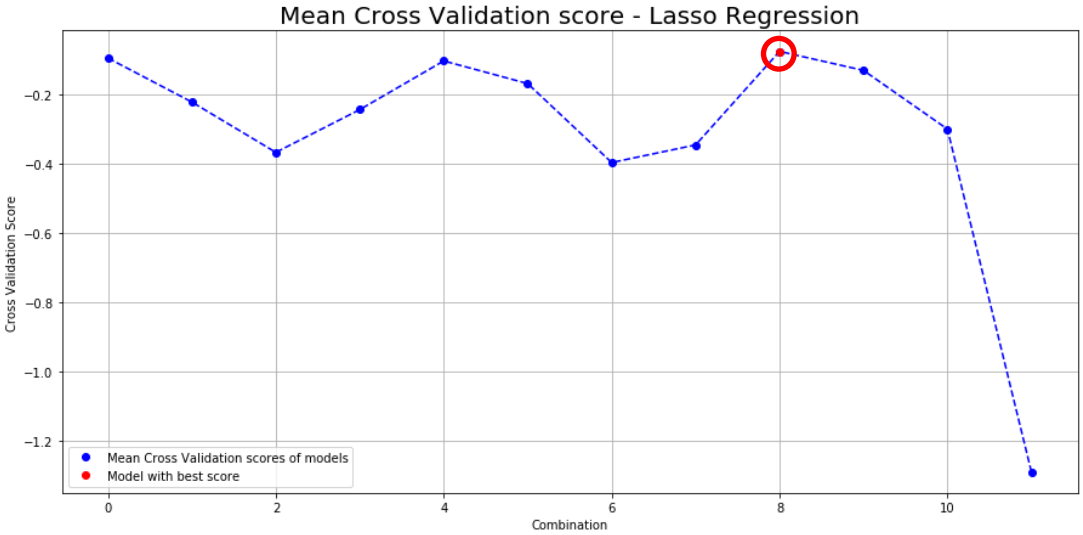
Grid search with LASSO regression:

```
gridlasreg= GridSearchCV(pipe,  param_grid = gridparameters[2], cv = 4, n_jobs = 2) ##Refit is always True
gridlasreg.fit(X_train, Y_train)
```

Output:

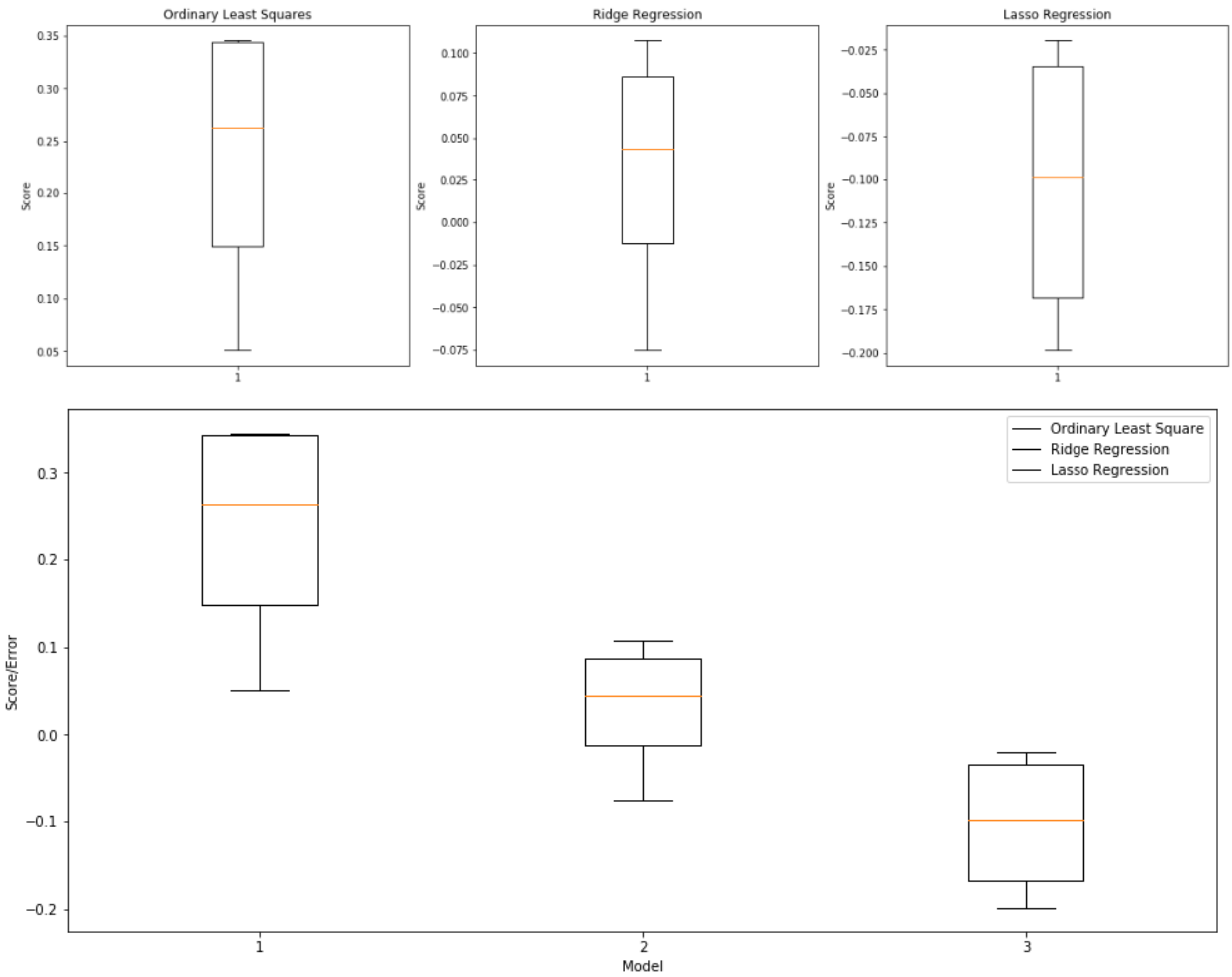
```
Best model after tuning the hyper parameters using Grid Search:
{'sgd__eta0': 0.3, 'sgd__learning_rate': 'constant', 'sgd__penalty': None}


Best score: -0.07711011791633715

Number of Folds: 4
```

```
plt.figure(figsize = (15, 7))
plt.title("Mean Cross Validation score - Lasso Regression",fontsize = 20)
plt.plot(scores,'b--')
plt.plot(scores, "bo", label = "Mean Cross Validation scores of models")
plt.plot(scores.index(bestscore),bestscore, "ro",label = "Model with best score")
plt.xlabel('Combination')
plt.ylabel('Cross Validation Score')
plt.legend()
plt.grid()
plt.show()
```



Evaluate each model using Optimal parameters:

```
cv_linreg = cross_val_score(pipe.set_params(**gridlinreg.best_params_),X_train,Y_train,cv=4)
cv_ridreg = cross_val_score(pipe.set_params(**gridridreg.best_params_),X_train,Y_train,cv=4)
cv_lasreg = cross_val_score(pipe.set_params(**gridlasreg.best_params_),X_train,Y_train,cv=4)
```



From the boxplots, it can be seen that the cross validation score for the model with Ridge Regression(L2 Loss) performs well than the other two models.The variance of error is less for regularized models and the opposite for non-regularized model. Here, L2 loss model is better than models with L1 and no regularization.

Polynomial Regression:

```
start,end = uniform_values(1,0.05)
D1_x,D1_y = create_dataset(start,end)
degree = [1,2,7,10,16,100]
Y_pred = []
RMSE = []
poly_D1_x = create_poly(D1_x,degree)##A dict of polynomial feature for data
for ind,i in enumerate(degree):
    ## Linear Regression model
    linreg = linear_model.LinearRegression()
    linreg.fit(poly_D1_x[i],D1_y)##Fit linear regression model
    Y_pred.append(linreg.predict(poly_D1_x[i]))
    RMSE.append(np.sqrt(mean_squared_error(D1_y,Y_pred[ind])))
```

Firstly, the start and end of the range with the given mean and standard deviation is found for which the uniform distribution is to be generated with. Secondly, the dataset is created with the start and end of the range. Using the degrees of polynomial, Polynomial features are generated for the dataset D1. Lastly, the model is fit for all the polynomial features and tested on the same and the error is also noted.

Function name : uniform_values
Parameter : mean and standard deviation

```
def uniform_values(mean,sd):
    ##To find the end of the range
    end = np.sqrt(12) * sd / 2 + mean
    ## To find the start of the range
    start = 2 * mean - end
    return start,end
```

```
Start: 0.9133974596215562 End: 1.0866025403784438
```

For the given mean and standard deviation, this function finds the starting and ending value of the range. The result is used for creating the dataset with uniform distribution.

Function name : create_dataset
Parameter : Start and end of the range for which uniform distribution is to be created

```
def create_dataset(start,end):
    ##Create the input values for D1
    D1_x = np.random.uniform(start,end,(100,1))
    D1_x = sorted(D1_x)

    D1_y = generate_target(D1_x)
    return D1_x,D1_y
```

From the found start and end values for the given mean and standard deviation, this function creates an uniform distribution for the Dataset D1 of size (100X1). The output shows the first 10 values.

Output:

```
D1_X: [array([0.91369959]), array([0.9182404]), array([0.91894628]), array([0.9193962]), array([0.92031634]), array([0.9209644
1]), array([0.92610762]), array([0.92698167]), array([0.92707757]), array([0.92924431])]

D1_Y: [array([13.96785952]), array([13.67669382]), array([13.89079244]), array([13.81184302]), array([13.89629011]), array([13.
94565833]), array([14.02395875]), array([13.88879677]), array([13.9949125]), array([13.85269043])]
```

Function name : generate_target
Parameter : X data of dataset

```
def generate_target(x):
    y = []
    ## Creating random additive values
    additive = np.random.uniform(0,0.5,(100,1))
    for i in range (0,len(x)):
        ##Generating the target values
        y.append(1.3*pow(x[i],2) + 4.8*x[i] + 8 + additive[i])
    return y
```

The generated dataset D1 with X data is fed into this function to get the target Y data.

Function name : create_poly
Parameter : X data of dataset

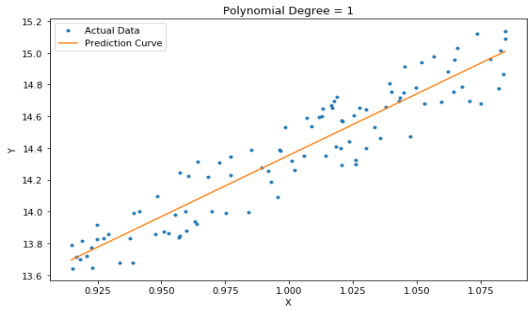
```
def create_poly(D1_x,degree):
    poly = {}
    for i in degree:
        #Generating Polynomial feature for the data with given degrees
        poly[i] = PolynomialFeatures(i,include_bias =False).fit_transform(D1_x)
    return poly
```

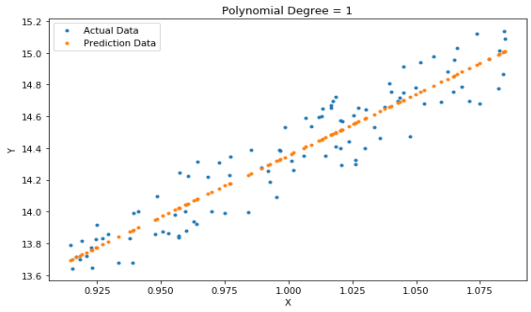
For the given degrees of polynomial, this function genarates the polynomial features for the dataset D1 and returns a list of datasets.

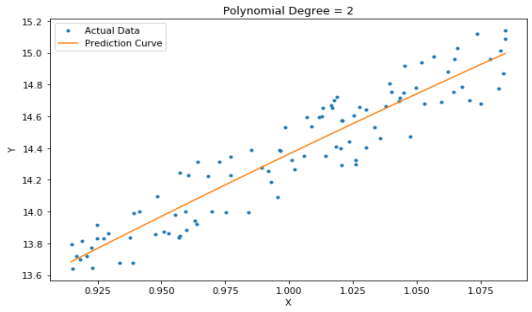
Output:

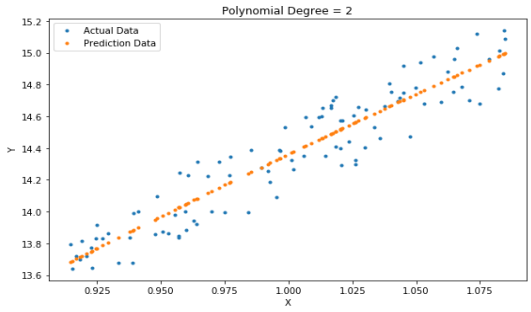
```
Degree 1: [0.91369959]
Degree 2: [0.91369959 0.83484694]
Degree 7: [0.91369959 0.83484694 0.76279931 0.69696941 0.63682067 0.58186278
 0.53164779]
Degree 10: [0.91369959 0.83484694 0.76279931 0.69696941 0.63682067 0.58186278
 0.53164779 0.48576636 0.44384453 0.40554056]
Degree 16: [0.91369959 0.83484694 0.76279931 0.69696941 0.63682067 0.58186278
 0.53164779 0.48576636 0.44384453 0.40554056 0.37054225 0.3385643
 0.30934606 0.28264937 0.25825661 0.23596896]
Degree 100: [9.13699590e-01 8.34846941e-01 7.62799307e-01 6.96969414e-01
 6.36820668e-01 5.81862783e-01 5.31647786e-01 4.85766364e-01
 4.43844528e-01 4.05540563e-01 3.70542246e-01 3.38564298e-01
 3.09346061e-01 2.82649369e-01 2.58256612e-01 2.35968961e-01
 2.15604743e-01 1.96997965e-01 1.79996960e-01 1.64463148e-01
 1.50269911e-01 1.37301556e-01 1.25452376e-01 1.14625784e-01
 1.04733532e-01 9.56949852e-02 8.74364687e-02 7.98906656e-02
 7.29960684e-02 6.66964777e-02 6.09405444e-02 5.56813504e-02
 5.08760270e-02 4.64854050e-02 4.24736955e-02 3.88081982e-02
 3.54590347e-02 3.23989055e-02 2.96028667e-02 2.70481271e-02
 2.47138627e-02 2.25810462e-02 2.06322926e-02 1.88517173e-02
 1.72248064e-02 1.57382985e-02 1.43800769e-02 1.31390704e-02
 1.20051632e-02 1.09691127e-02 1.00224738e-02 9.15753019e-03
 8.36723157e-03 7.64513606e-03 6.98535768e-03 6.38251845e-03
 5.83170449e-03 5.32842600e-03 4.86858065e-03 4.44842014e-03
 4.06451966e-03 3.71374995e-03 3.39325180e-03 3.10041278e-03
 2.83284589e-03 2.58837013e-03 2.36499272e-03 2.16089288e-03
 1.97440694e-03 1.80401481e-03 1.64832759e-03 1.50607624e-03
 1.37610125e-03 1.25734315e-03 1.14883392e-03 1.04968908e-03
 9.59100480e-04 8.76329715e-04 8.00702101e-04 7.31601182e-04
 6.68463700e-04 6.10775008e-04 5.58064875e-04 5.09903647e-04
 4.65898753e-04 4.25691500e-04 3.88954149e-04 3.55387246e-04
 3.24717181e-04 2.96693955e-04 2.71089145e-04 2.47694041e-04
 2.26317943e-04 2.06786612e-04 1.88940843e-04 1.72635170e-04
 1.57736684e-04 1.44123944e-04 1.31685988e-04 1.20321434e-04]
```

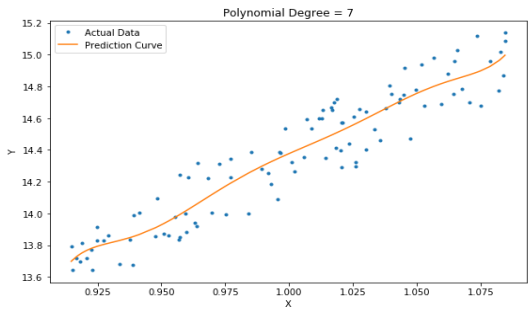
Prediction Curves for each reprocessed data

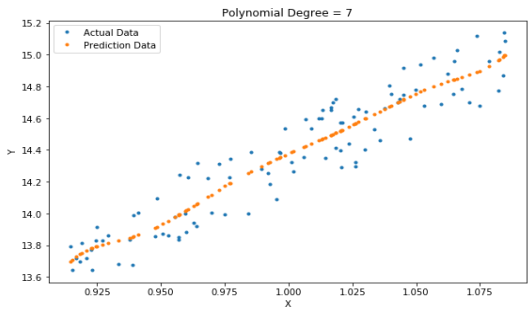


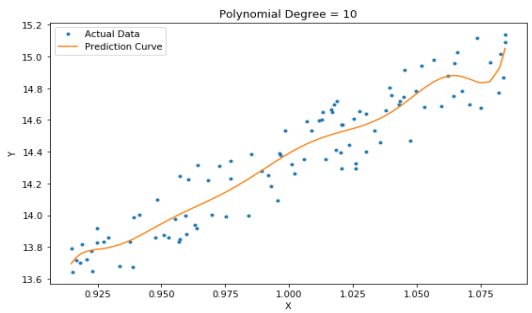


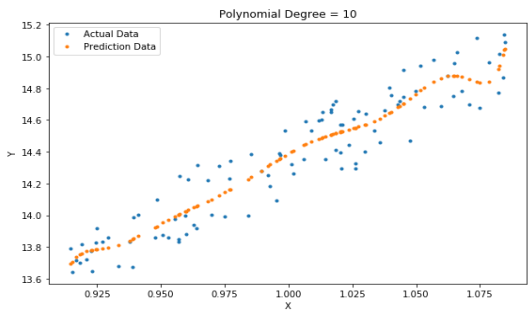


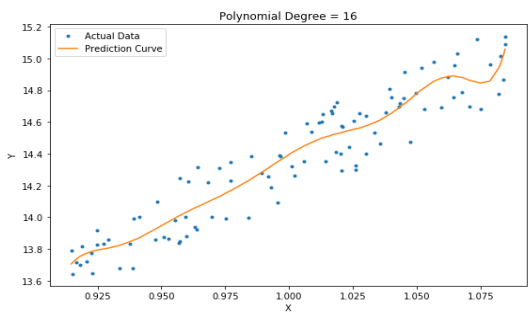


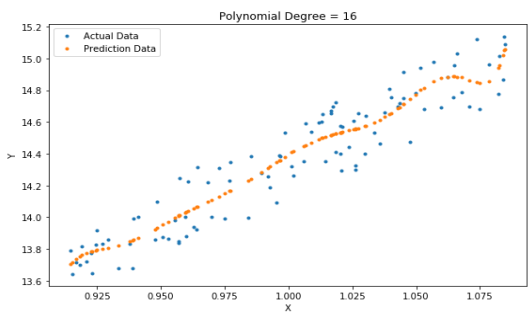


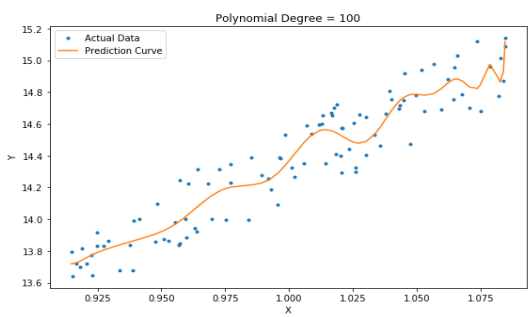


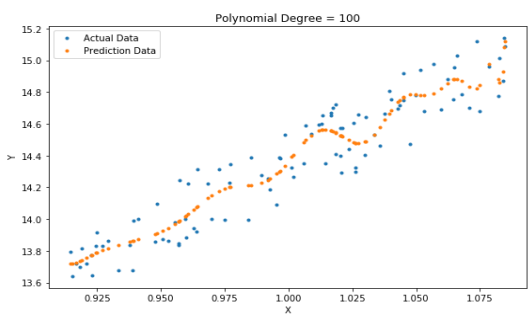






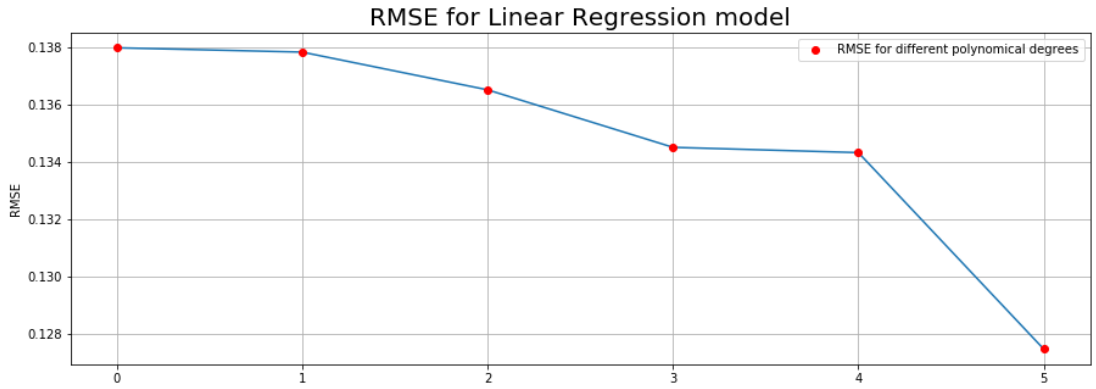






```
for i in range(0,len(RMSE)):
    print("RMSE for Polynomal degree "+str(degree[i])+": " + str(round(RMSE[i],5)))
plt.figure(figsize=(15,5))
plt.title("RMSE for Linear Regression model",fontsize=20)
plt.plot(RMSE)
plt.plot(RMSE,'ro',label ="RMSE for different polynomical degrees")
plt.ylabel("RMSE")
plt.grid()
plt.legend()
plt.show()
```

```
RMSE for Polynomal degree 1: 0.13798
RMSE for Polynomal degree 2: 0.13783
RMSE for Polynomal degree 7: 0.13651
RMSE for Polynomal degree 10: 0.13451
RMSE for Polynomal degree 16: 0.13432
RMSE for Polynomal degree 100: 0.12748
```



For different degrees of polynomial, the prediction curve tries to fit exactly for increase in degrees. This will lead to overfitting of the model which results in very low error. For different prediction curves, I observed that without regularization, the Root Mean Squared Error (RMSE) decreases with increase in the number of degrees of the polynomial. The error of the polynomial feature of degree 100 is better compared to the other degrees. In other words, higher the degree of polynomial, lower will be the error in the case of non-regularized model. This may lead to overfitting of the model.

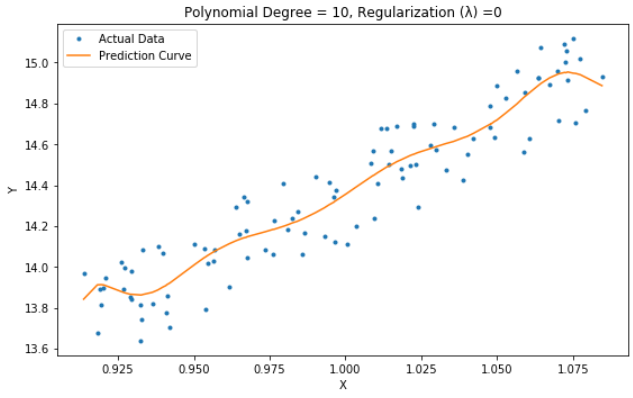
Effect of Regularization:

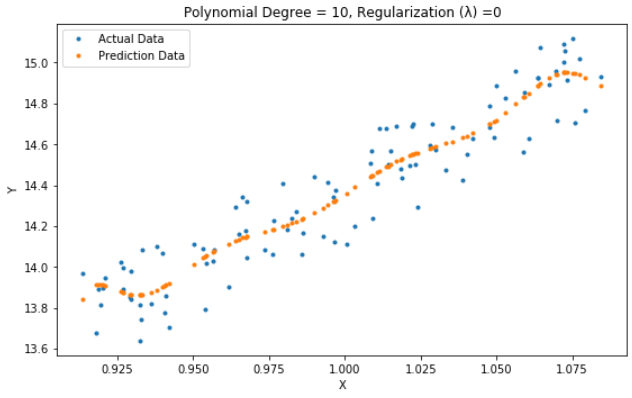
In order to check the effect of regularization, the degree of the polynomial is set to zero and different regularization constants, 0,10e-6,10e-2 and 1, are used to genarate the model with ridge regression (L2 Loss)

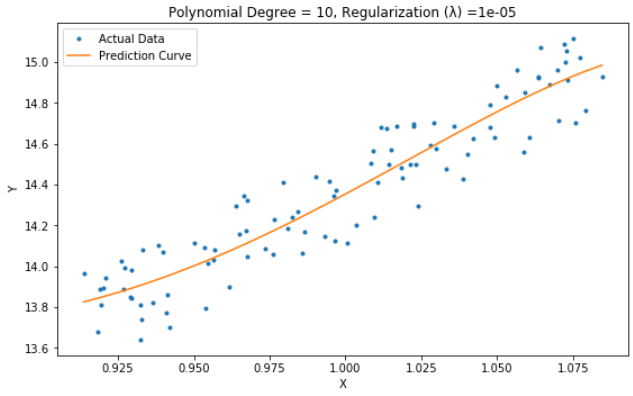
```
ridge_alpha = [0,10e-6,10e-2,1]
Y_pred_ridge = []
RMSE_ridge = []
for ind,i in enumerate(ridge_alpha):
    ridreg = linear_model.Ridge(alpha = i)##Initialize Ridge regression with custom regularization constant
    ridreg.fit(poly_D1_x[10],D1_y)##Fitting the model
    Y_pred_ridge.append(ridreg.predict(poly_D1_x[10]))
    RMSE_ridge.append(np.sqrt(mean_squared_error(D1_y,Y_pred_ridge[ind])))
```

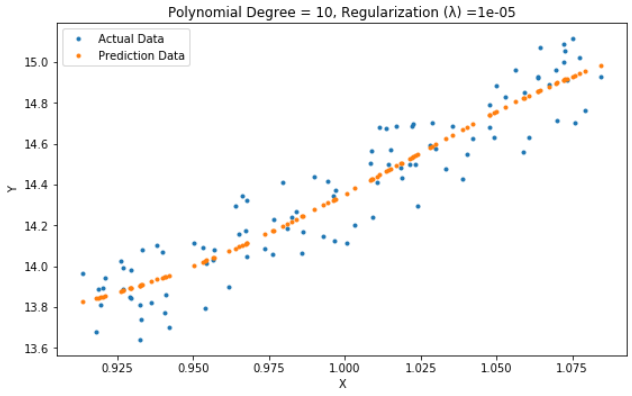
```
fig, ax = plt.subplots(4, 2, figsize=(20, 25))
axis = 0
for i in range(0,len(Y_pred_ridge)):
    ax[axis][0].plot(D1_x,D1_y,'.',label = "Actual Data")
    ax[axis][0].plot(D1_x,Y_pred_ridge[i],label = "Prediction Curve")
    ax[axis][0].set_xlabel("X")
    ax[axis][0].set_title("Polynomial Degree = " + str(degree[i]))
    ax[axis][0].set_ylabel("Y")
    ax[axis][0].legend()
    ax[axis][1].plot(D1_x,D1_y,'.',label = "Actual Data")
    ax[axis][1].plot(D1_x,Y_pred_ridge[i],'.',label = "Prediction Data")
    ax[axis][1].set_title("Polynomial Degree = " + str(degree[i]))
    ax[axis][1].set_xlabel("X")
    ax[axis][1].set_ylabel("Y")
    ax[axis][1].legend()
    axis+=1
```

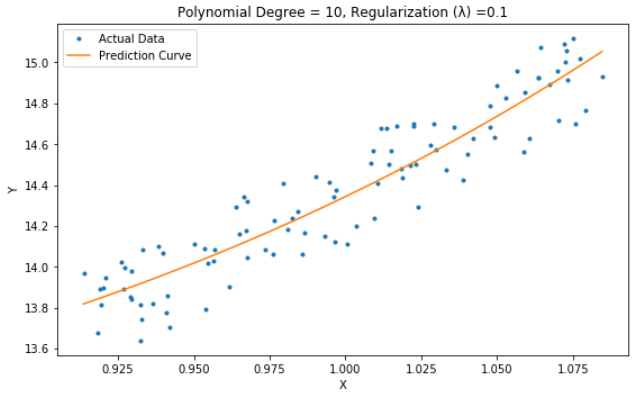
Prediction Curves using different regularization constants

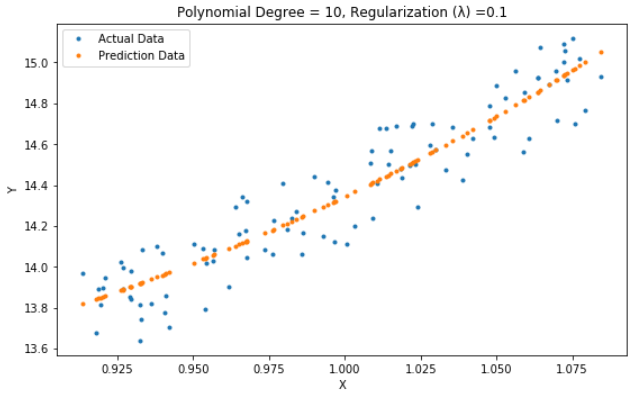


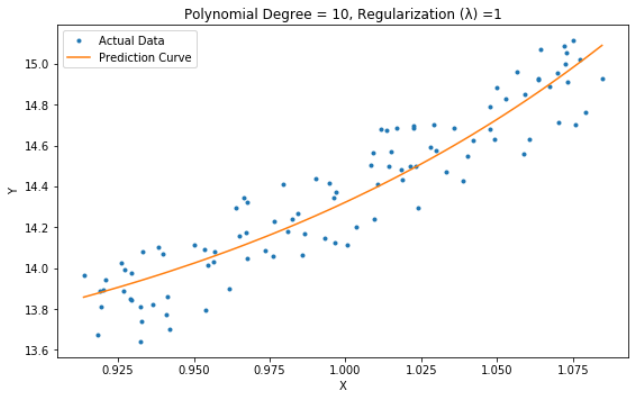


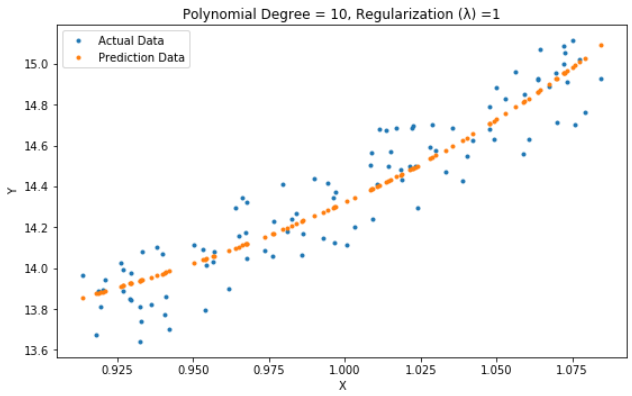






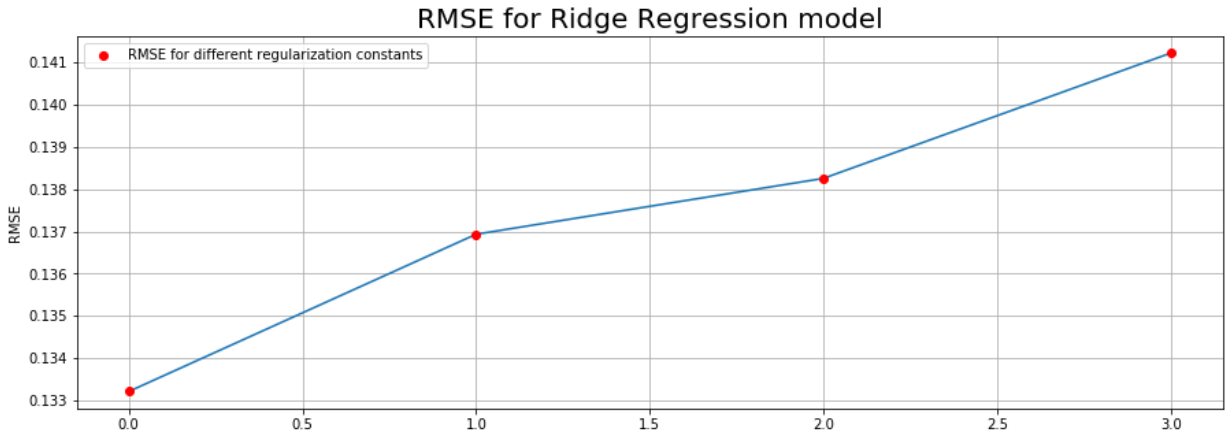






```
for i in range(0,len(RMSE_ridge)):
    print("RMSE for Regularization (λ:" + str(ridge_alpha[i])+") = " + str(round(RMSE_ridge[i],5)))
plt.figure(figsize=(15,5))
plt.title("RMSE for Ridge Regression model",fontsize=20)
plt.plot(RMSE_ridge)
plt.plot(RMSE_ridge,'ro',label ="RMSE for different regularization constants")
plt.ylabel("RMSE")
plt.grid()
plt.legend()
plt.show()
```

```
RMSE for Regularization (λ:0) = 0.13321
RMSE for Regularization (λ:1e-05) = 0.13693
RMSE for Regularization (λ:0.1) = 0.13825
RMSE for Regularization (λ:1) = 0.14122
```



For different regularization constants, the prediction curves varies a lot. As the value of regularization constant increses, the prediction curve reduces its spread around the data and it can be seen that for a reg constant of 1, it's almost a straight line which makes the model a poor fit and may lead to underfitting of the model. On the other hand, a very low regularization constant leads to overfitting of the model. Also comparing the RMSE values for different regularization constants, it can be observed that the error increases as the regularization value increases and vice-versa. Hence it's preferred to use an optimum regularization constant.