Regularized Linear Least Squares

Swetha Varadarajan

October 9, 2014

Contents

Abstract

Linear models are being well sorted for its ease of view and many other special properties. Thus, linear regression becomes an essential topic of study. In this report, Regularized Linear Least Squares method has been presented. Its implementation in Python language is explained. Demonstration with a simple function and also using the dataset selected from (Bishop, 2006) UCI Machine Learning Repository UCI is also presented.

1 Introduction

Linear regression model aims at developing or predicting a model for a given problem. Suppose there is an outcome(f(x)) expected from a situation(x), the linear regression model does its best to predict the outcome accurately so as to be cautious or preventive. It is called linear because the model predicts the outcome as a linear function(straight line) of the situation.

1.1 Linear regression

Viewing it from mathematical point in a multi-dimensional space,

$$w = (X^T X)^{-1} X^T T \tag{1}$$

Equation-1 gives the parameters of the models in terms of the situation(X) and outcome(T). Here, w is called the weights since it weighs the situation's parameters to produce the outcome. There are many variants of the model. The one which we are interested is the Regularised least squares. When dealing with a large data samples, the equation-1 becomes inefficient. Hence, we resort to a sequential method of modelling. But, this may result in under-sampling of data and thereby creating noise. In order to avoid it, we introduce some artificial constant called λ weighted over the model weights to the above equation and it turns out after rigorous mathematics as equation-2. So, the ultimate aim is to minimize this artificial terms and also to reduce the noise level in developing the best predictable linear model.

$$w = (X^T X + \lambda I)^{-1} X^T T \tag{2}$$

1.2 Objective

To create such a model, we need to know the behaviour of the system. So, we need to train the model with sample data and then use it to predict the original data. The algorithm is as follows.

1. Select a set of data to train the model
2. Normalize these data to reduce variations
3. Pick up a random λ value
4. Calculate the model parameters(weight matrix) using equation 2
5. Test it on original data
6. Check for accuracy. Determine best λ value.

The further sections demonstrates this on a toy data and real data using the code implemented in python language. The results and discussions are also presented.

2 Python Implementation

This section gives a detailed description of **python code** used for reading the data, constructing input and target matrices, visualizations of the data, random partitioning of data, standardization, **makeLLS** (regularized)function and **useLLS**function, finding good value of lambda and also the codes used for plotting.

2.1 Reading,constructing and visualization

The data set Concrete Slump Test Data Set from the UCI Machine Learning Repository was taken. The code for reading the data is shown in the following listing. The pandas package was used to read data.

```
1 # reading data
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import pandas
5 d = pandas.read_csv('slump_test.data')
```

The first column which contains the serial number of the data was excluded and the remaining columns were taken for further implementation. So, a total of 103 samples with 10 attributes were selected and divided into input and target matrices as shown in the following listing. The selected data set contains 7 input variables and 3 output variables. So, the input matrix dimension will be $103X7$ and the target matrix dimension will be $103X3$. But, when including the bias column, the input matrix changes to $103X8$.The corresponding names were also extracted and stored.

```
# constructing
T = d.iloc[:,8:].values
X = d.iloc[:,1:8].values
names =  list(d.columns.values)
Xnames = names[1:8]
Tname = names[8:]
# include bias in input matrix
X1 = np.hstack((np.ones((X.shape[0],1)), X))
Xnames.insert(0, 'bias')
```

To visualize the 3 target variables as a function of the 7 input variables, 3*7 plots were drawn and the code corresponding to it is shown below.

```
# visualization
for bb in range(T.shape[1]):
    plt.figure(bb,figsize=(10,10))
    for c in range(X.shape[1]):
        plt.subplot(3,3, c+1)
        plt.plot(X[:,c], T[:,bb], 'o', alpha=0.5)
        plt.ylabel(Tname[bb])
        plt.xlabel(Xnames[c])
                plt.savefig('visualization{bb}.png'.format(bb=bb))
```

2.2 Random partitioning of data and standardization

The data set is divided into 80 percentage of training and 20 percentage of testing data as shown in the following listing. The data is partitioned in a random manner by shuffling the row indices and grouping. So, at the end we get 4 sets which is shown in table 1.

Data set name	size
XTrain(Input training data)	(82, 8)
TTrain(Target training data)	(82, 3)
XTest(Input testing data)	(21, 8)
TTest(Target testing data)	(21, 8)

Table 1: Partitioned data sets and size

```
#random data partitioning
nrows = X1.shape[0]
nTrain = int(round(nrows*0.8))
nTest = nrows - nTrain
rows = np.arange(nrows)
np.random.shuffle(rows)
trainIndices = rows[:nTrain]
testIndices = rows[nTrain:]
Xtrain = X1[trainIndices,:]
Ttrain = T[trainIndices,:]
Xtest = X1[testIndices,:]
Ttest = T[testIndices,:]
```

The standardization of data is done using the function defined in the following listing. The formula for standardization is given in equation-3 in which z is the standardized data, x is the original data, μ is the mean and σ is the standard deviation. Mean and standard deviation are found using numpy package functions numpy.mean and numpy.std respectively.

$$z = \frac{x - \mu}{\sigma} \tag{3}$$

```
# standardization of data
def makeStandardize(X):
    means = X.mean(axis=0)
    stds = X.std(axis=0)
    def standardize(origX):
        return (origX - means) / stds
    def unStandardize(stdX):
        return stds * stdX + means
    return (standardize, unStandardize)
```

2.3 makeLLS (regularized) and useLLS

The makeLLS function is defined as shown below. It takes the input data, target data and λ values as input and produces the weights, standardized and unstandardized data as results. Inside the function, it standardizes the X value using the *make standardize* function described before. After this, the bias data(ones) are stacked. This is because, if we standardize the bias value, we get weight value with a large deviation from the mean(zero). Hence, the bias data is kept un-standardized.Now, the λ matrix is formed. It is nothing but the identity matrix of dimension equal to the number of columns(attributes) in the input data. Here, it is 8. The [0,0] element of λ matrix is assigned zero in order as not to disturb the bias data. Finally, the weights are calculated by using the **np.linalg.lstsq()** in-built function. This function returns the least square solution for the set of matrices mentioned. It returns solution for each column of the second matrix. Here, equation-1 is being implemented. The second matrix is nothing but the dot product of input and target matrix. So, the dimension of this will be 8*3. Dimension of the first term (np.dot(X.T,X)+penalty) is 8*8. So, the dimension of weight matrix will be 8*3. These are the predicted weights.

```
# makeLLS function
def makeLLS(X,T,lambdaw):
    (standardizeF, unstandardizeF) = makeStandardize(X)
    X = standardizeF(X)
    (nRows,nCols) = X.shape
    X = np.hstack((np.ones((X.shape[0],1)), X))
    penalty = lambdaw * np.eye(nCols+1)
    penalty[0,0]  =0  # don't penalize the bias weight
    w = np.linalg.lstsq(np.dot(X.T,X)+penalty, np.dot(X.T,T))[0]
    return (w, standardizeF, unstandardizeF)
```

The predicted weights obtained from the *makeLLS* function is passed to the *useLLS* in order to make use of these weights to calculate the target data. This is done on the 20 percentage of testing data that we had partitioned initially. This is done simply by taking the dot product of the weights and the testing input data. The code is shown below. It can be seen that again standardization and bias data inclusion is done for the testing data too.

```
# useLLS function
def useLLS(model,X):
    w, standardizeF, _ = model
    X = standardizeF(X)
    X = np.hstack((np.ones((X.shape[0],1)), X))
    return np.dot(X,w)
```

2.4 Best lambda finding

Finding the best λ value is nothing but to observe the *root mean square errors* **(RMSE)** obtained for different values of λ. The one which corresponds to the minimum error is the best value. To do this, we take 2 variations. One is obvious to vary λ and the other is to randomize the set of training and testing data we select simultaneously maintaining the partition percentage as 80-20.

```
#FINDING BEST LAMBDAW VALUE

#defining lambdaw and model value
lambdaw=np.linspace(0,50,20)*20
nModels=1000

# creating list
RMSEtt=[]
RMSEtn=[]

# iteration-1
for lamb in lambdaw:
    models = []
    YAll=[]
    ZAll=[]

    #iteration-2
    for modeli in range(nModels):
        rows = np.arange(nrows)
        np.random.shuffle(rows)
        trainIndices = rows[:nTrain]
        testIndices = rows[nTrain:]
        Xtrain = X[trainIndices,:]
        Ttrain = T[trainIndices,:]
        Xtest = X[testIndices,:]
        Ttest = T[testIndices,:]
        model = makeLLS(Xtrain,Ttrain,lamb)
        models.append(model)

    #iteration-3
    for model in models:
        YAll.append(useLLS(model,Xtest))
        ZAll.append(useLLS(model,Xtrain))

    # Finding the means
    YAll = np.array(YAll).squeeze().T
    Ytest = np.mean(YAll,axis=2).flatten()
    Ttest = Ttest.flatten()
    ZAll = np.array(ZAll).squeeze().T
    Ztest = np.mean(ZAll,axis=2).flatten()
    Ttrain = Ttrain.flatten()

    #Finding RMSE values
    RMSEtest = np.sqrt(np.mean((Ytest - Ttest)**2))
    RMSEtt.append(RMSEtest)
    RMSEtrain = np.sqrt(np.mean((Ztest - Ttrain)**2))
    RMSEtn.append(RMSEtrain)

#Finding the best value through plot
a=np.array(RMSEtt)
b=np.array(RMSEtn)
rmses=np.vstack((a,b)).T
plt.plot(lambdaw,a,'o-')
plt.plot(lambdaw,b,'*-')
plt.legend(('test','train'))
plt.ylabel('RMSE')
plt.xlabel('$\lambda$')
```

In the above listing, the lines 4 and 5 is used to initialize some values for the two iteration. Iteration 2 and 3 can be nested in a single one. So, the two iterations are,

1. Iterate over λ value
2. Iterate over number of model value thereby creating different randomized data sets.

Lines 8 and 9 are used to create a list for storing the RMSE values of training and testing data. Lines 12-47 is the iteration-1. It involves creating a list every time to store the *makeLLS* and *useLLS* function results. Iteration 2 and 3 performs the randomization of data sets(lines 19-26) followed by the 2 function calls. Here, we call *useLLS* twice one for testing and the other for training data. The results of this predicted values are stored in YAll[] and ZAll[] list.Further, the mean and RMSE values are found for these predicted values using formulas given in equation-3. Mean over the 1000(number of models) corresponding to each λ value is taken to calculate the mean and RMSE values. The axis is taken as 2 since YAll is a 3-dimensional list that has the iteration number and the 2D weight matrix. This is being transposed. So, the iteration comes as the 3rd axis. Hence, the mean is taken along this axis. To find the best λ value, a plot is drawn with λ along the x-axis and the two RMSE arrays along the y-axis. By observing this graph the value of λ corresponding to the minimum error on the testing data can be taken as the best value. The weights corresponding to this best value has to selected.

2.5 Plotting codes

The list of plots used are

1. Visualization of data sets(Target versus Input data): This is discussed in section 2.1
2. RMSE plots: This is discussed in section 2.4
3. Actual versus predicted results with the regression line.

The code for 3rd plot is as follows. It includes 6 sub-plots, 2 for each target and each target has their predicted training and testing data plotted against their actual training and testing data. In addition to it, the regression line is drawn by taking a slope of it.

```
1  plt.figure(figsize=(10,10))
2  plt.subplot(3,2,1)
3  plt.plot(Ttest[:,0],predictedtt[:,0],'o')
4  a,b = max(min(Ttest[:,0]),min(predictedtt[:,0])),
5                    min(max(Ttest[:,0]),max(predictedtt[:,0]))
6  plt.plot([a,b],[a,b],'r',linewidth=3)
7  plt.title('Testing SLUMP (cm) result, with lambda = 257')
8  #plt.xlim(-40,50)
9  #plt.ylim(-40,50)
10 plt.xlabel('Actual')
11 plt.ylabel('Predicted')
12
13 plt.subplot(3,2,2)
14 plt.plot(Ttest[:,1],predictedtt[:,1],'o')
15 a,b = max(min(Ttest[:,1]),min(predictedtt[:,1])),
16                   min(max(Ttest[:,1]),max(predictedtt[:,1]))
17 plt.plot([a,b],[a,b],'r',linewidth=3)
18 plt.title('Testing FLOW (cm) result, with lambda = 257')
19 #plt.xlim(-40,50)
20 #plt.ylim(-40,50)
21 plt.xlabel('Actual')
22 plt.ylabel('Predicted')
23
24 plt.subplot(3,2,3)
25 plt.plot(Ttest[:,2],predictedtt[:,2],'o')
26 a,b = max(min(Ttest[:,2]),min(predictedtt[:,2])),
```

```
                    min(max(Ttest[:,2]),max(predictedtt[:,2]))
plt.plot([a,b],[a,b],'r',linewidth=3)
plt.title('Testing 28-day Compressive Strength (Mpa)
                          result, with lambda = 257')
#plt.xlim(-40,50)
#plt.ylim(-40,50)
plt.xlabel('Actual')
plt.ylabel('Predicted')

plt.subplot(3,2,4)
plt.plot(Ttrain[:,0],predictedtn[:,0],'o')
a,b = max(min(Ttrain[:,0]),min(predictedtn[:,0])),
                   min(max(Ttrain[:,0]),max(predictedtn[:,0]))
plt.plot([a,b],[a,b],'r',linewidth=3)
plt.title('Training SLUMP (cm)result, with lambda = 257')
#plt.xlim(-40,50)
#plt.ylim(-40,50)
plt.xlabel('Actual')
plt.ylabel('Predicted')

plt.subplot(3,2,5)
plt.plot(Ttrain[:,1],predictedtn[:,1],'o')
a,b = max(min(Ttrain[:,1]),min(predictedtn[:,1])),
                   min(max(Ttrain[:,1]),max(predictedtn[:,1]))
plt.plot([a,b],[a,b],'r',linewidth=3)
plt.title('Training FLOW (cm)result, with lambda = 257')
#plt.xlim(-40,50)
#plt.ylim(-40,50)
plt.xlabel('Actual')
plt.ylabel('Predicted')

plt.subplot(3,2,6)
plt.plot(Ttrain[:,2],predictedtn[:,2],'o')
a,b = max(min(Ttrain[:,2]),min(predictedtn[:,2])),
                   min(max(Ttrain[:,2]),max(predictedtn[:,2]))
plt.plot([a,b],[a,b],'r',linewidth=3)
plt.title('Training 28-day Compressive Strength (Mpa)
                          result, with lambda = 257')
#plt.xlim(-40,50)
#plt.ylim(-40,50)
plt.xlabel('Actual')
plt.ylabel('Predicted')
```

3 Toy data Demonstration

This section explains the concept for a simple example taken from the notebook[]. The code is shown in the following listing.

```
Xtrain = np.random.uniform(0,10,(nSamples,3))
Ttrain = 3 * X[:,0:1] - 2 * X[:,1:2] + np.random.normal(0, 10, (N,1))
```

The X value is taken as a random 3 dimensional data and the target value is is taken as a function of X as shown above. Later, 80 percent of the data were divided into training and the remaining as testing. Standardization of the data was done using the functions described in the previous section.Then, for different values of λ, the makeLLS and useLLS functions were called. The results were recorded to calculate the RMSEs. The following results were obtained. The first 4 sub-plots show the plot of actual versus predicted data with the regression line. It can be seen that for λ= 5000, the data points lie closer to the regression line

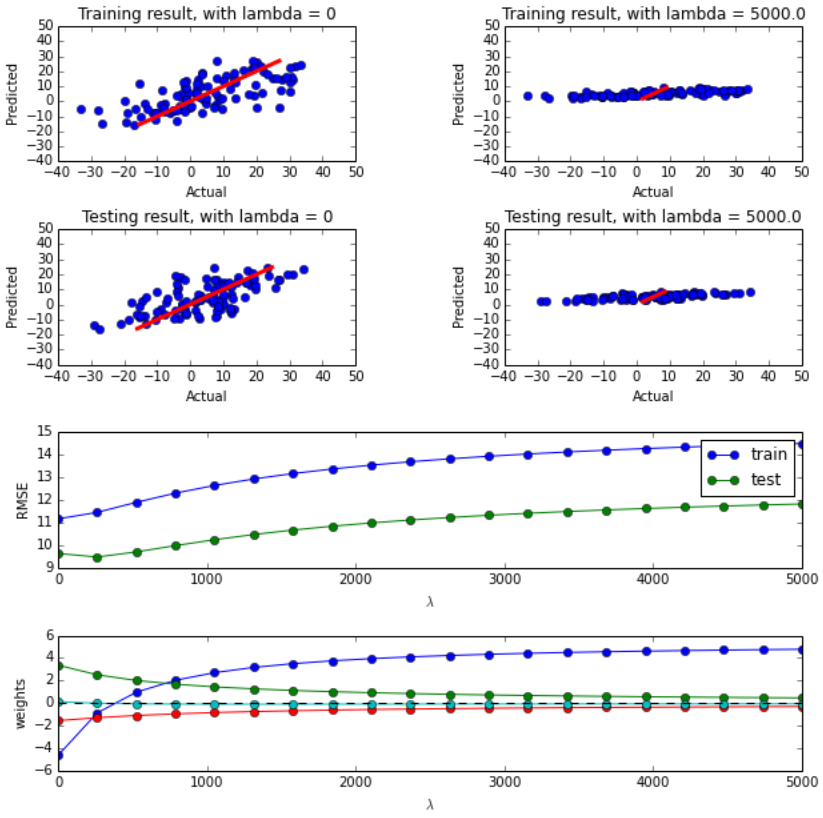

Figure 1: Demonstration

whereas for zero λ value, it is scattered. This shows the effect of under sampling of data. But, we cannot arbitrarily assume a higher value of it. So, in the next sub-plot, RMSE versus λ, we can see that the least error is obtained for some value around 200-300. The corresponding weights form the model parameters. Thus, this is the generalized method to develop a linear regression model. It can also be seen that the testing data's RMSE is constantly less than the training data's. This means that the noise introduced was uniform and the model prediction was accurate.

4 Discussion

This section discuss the results obtained from the initial data plots, about the linear model obtained, effect of λ variation and also few other observed results.

4.1 Initial data set

The data set taken is []. It has 7 inputs and 3 output variables. These 7 input variables when combined in a certain ratio , gives the concrete texture having the 3 output variable's attribute.This sub-section will describe the visualization of the target variables as a function of the input variables.

4.1.1 Slump data versus input data

The Slump(cm) is a measure of workability of the concrete obtained. It depends on various factors. There is a classification of slump according to the European Standard EN 206-1:2000from wikipedia and they have specific application. The following table gives the classification. For Slump value less than 1 cm,

Class	value in (cm)	application
S1	1-4	Foundations
S2	5-9	Normal reinforcement with vibration
S3	10-15	High workability
S4	16-21	High workability
S5	>=22	High workability

Table 2: Slump data classification

the application lies in road laying. The Figure-2 gives Slump as a function of various input data and the inferences obtained are discussed.

1. **Cement:**Seems to form cluster for cement value 150 and 300. Most of the slump value are about 10 indicating that they are high workability. Less than 20 percent of the data are less than 10.
2. **Slag:** Clustered near zero. For slag less than 100, high workability of slump data obtained. For slag greater than 100, it is observed to be distributed randomly. Another clustering also found near slag=120.But, it is may not depend only on slag value.
3. **Fly ash:** Similar result as seen for slag
4. **Water:** Increases with increase in water value
5. **SP:** Decreases abruptly for SP>14
6. **Coarse Aggregate:** Similar to water result. Cannot be deduced with respect to a single attribute.
7. **Fine Aggregate:** High values of Slump data

On average, the slump value is greater than 10 with respect to all the input variables.

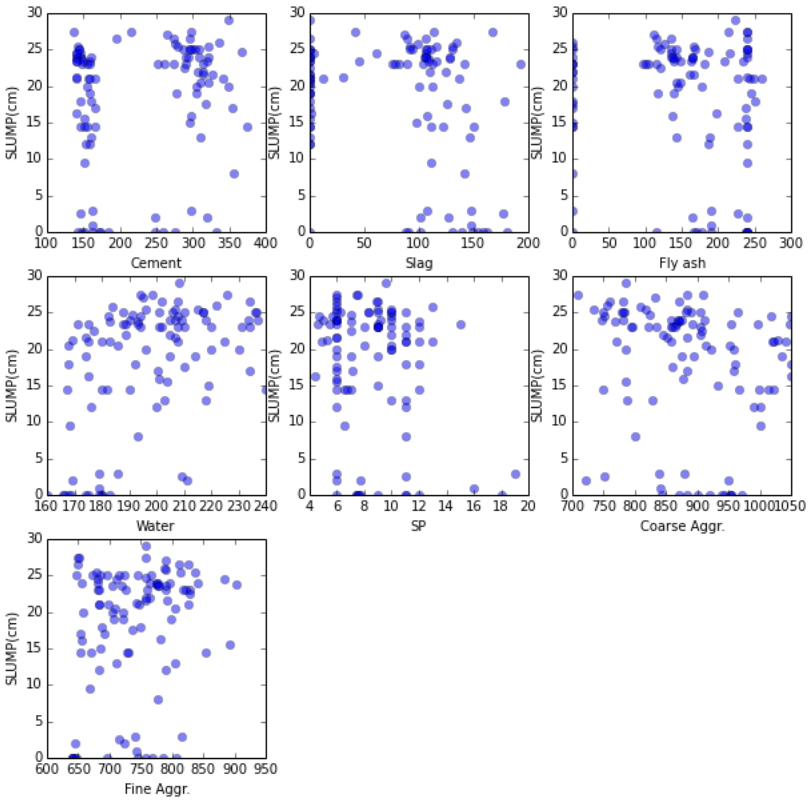

Figure 2: Slump result versus input data

Class	value in (cm)	application
SF1	55-65	slightly reinforced or non-reinforced structures
SF2	66-75	common walls and columns
SF3	76-85	complex structures

Table 3: Flow data classification

4.1.2 Flow data versus input data

Flow can be defined as the flow-ability of the concrete mixture.[reference].The following table gives the classification. For Flow value greater than 85 cm are used in special cases with care. The Figure-3 gives Flow as a function of various input data and the inferences obtained are discussed.

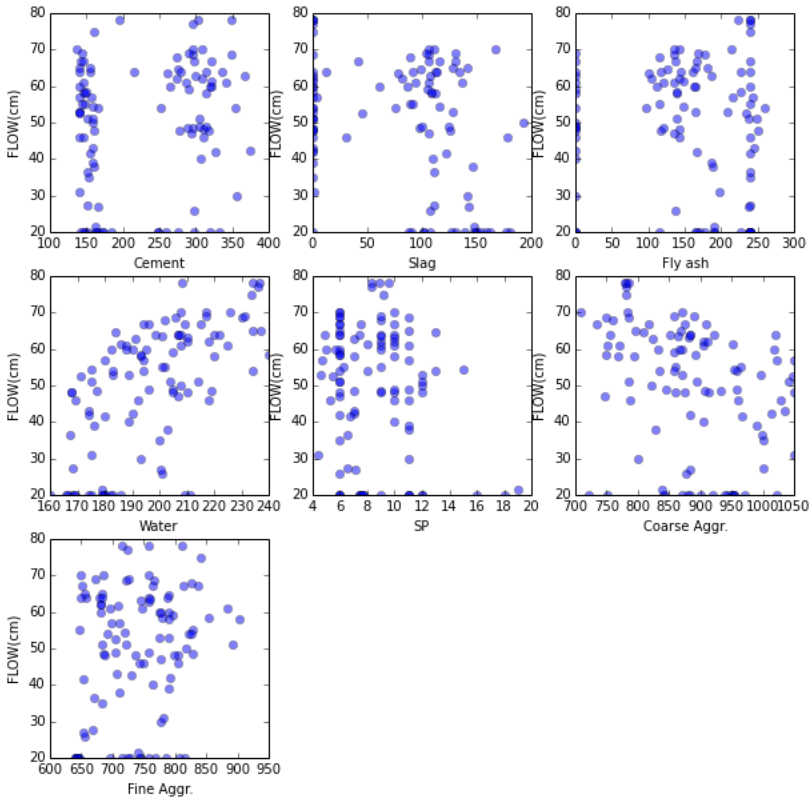

Figure 3: Flow result versus input data

1. **Cement:**Seems to form cluster for cement value 150 and 300.
2. **Slag:** Clustered near zero. For slag less than 100, high flow of data obtained. For slag greater than 100, it is observed to be distributed randomly.
3. **Fly ash:** values seen only at zero and between 100 and 250.
4. **Water:** Seems like a polynomial function. Increases with increase in water.
5. **SP:** Decreases abruptly for SP>14
6. **Coarse Aggregate:** Similar to water result.
7. **Fine Aggregate:** similar to the one obtained for slump result.

4.1.3 Compressive strength 28-day versus input data

Compressive strength can be defined as the strength of the concrete to withstand any load without getting compressed. The Figure-4 gives Compressive strength as a function of various input data and the inferences obtained are discussed.

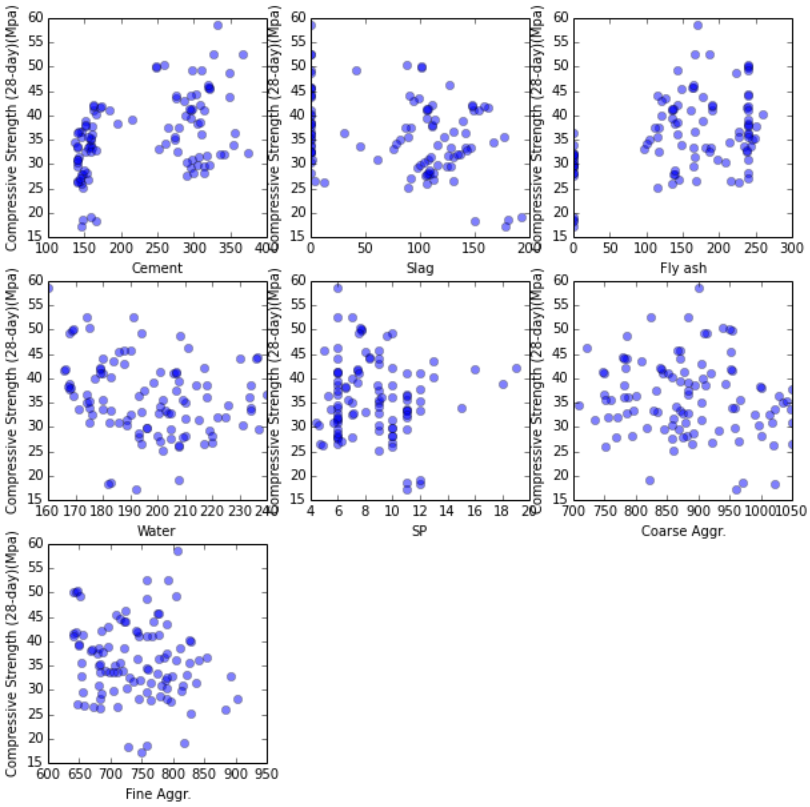

Figure 4: 28-day Compressive Strength result versus input data

1. **Cement:**Seems to form cluster for cement value 150 and 300.
2. **Slag:** Clustered near zero. Diamond shape in the middle
3. **Fly ash:** Clustered near zero and between 100 and 250.
4. **Water:** Seems to decrease with increase in water.
5. **SP:**Remains constant after SP=14.
6. **Coarse Aggregate:** seems like a bell curve
7. **Fine Aggregate:** Increases and then decreases.

On average, the slump value is greater than 10 with respect to all the input variables.

4.2 Measure of Linear model efficiency

To know how good the model, the result of code discussed in section 2.5 is shown in figure-5. The model is said to be good depending on how well the linear regression line fits with the data sets. For λ value equals 185, the following graphs have been derived. The two plots show the variation of each of the target value for

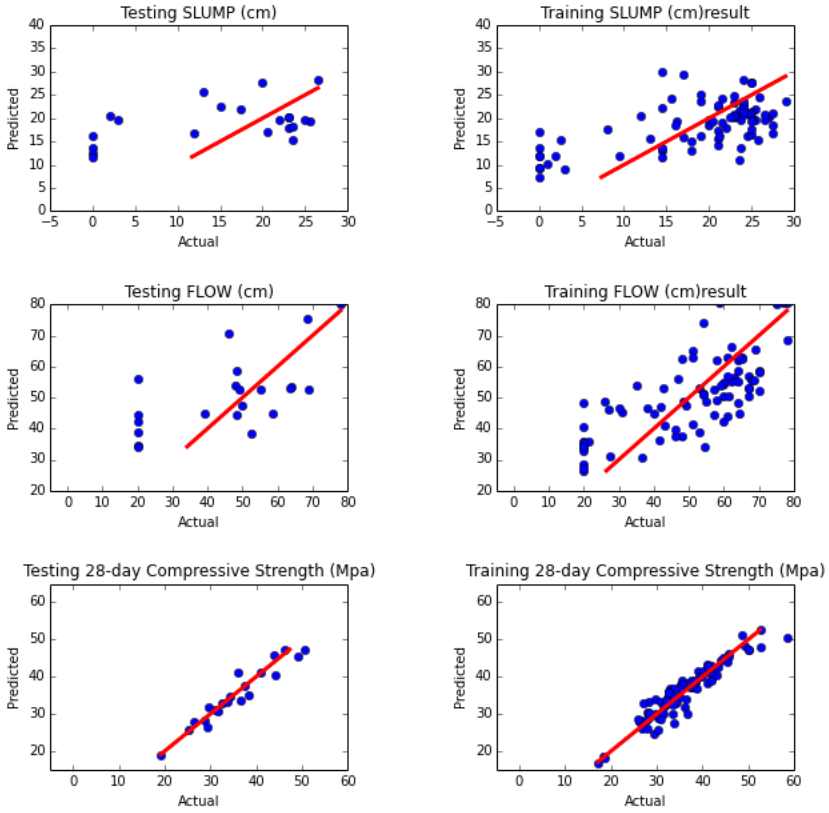

Figure 5: Predicted versus actual data plots when lambda=0

the best value and the zero value of λ and also for the training and the testing data sets. It can be observed that the variance or the scattering of the data for different values of λ is been reduced when it increases. Also, for slump and flow result, the some of the data points came closer to the regression line while some lay at the same place(left-most). These can be the boundary or the extreme points. the regression line for the 28-day Compressive Strength seems to be the best prediction. The value 185 is been chosen corresponding to the best value obtained(comparing the RMSE) which is explained in the next section. It can also be observed that, the horizontal line decreases in length and tends to become more horizontal with increase in λ value. I don't observe in the variation of the training and testing data. The pattern seems to be same. Since I have taken 80-20 percentage partition, the data points are dense for training data compared with the testing data. It can also be inferred that as λ value increases, the model becomes flatter. The main objective is to do good on the testing data. From the plot, I can see that it is evenly performed in both training and testing data. This will be evident in the next section's discussion on figure-7.

4.3 Variation of lambda

To know how good the model, the result of code discussed in section 2.5 is shown in figure-7. In this figure,

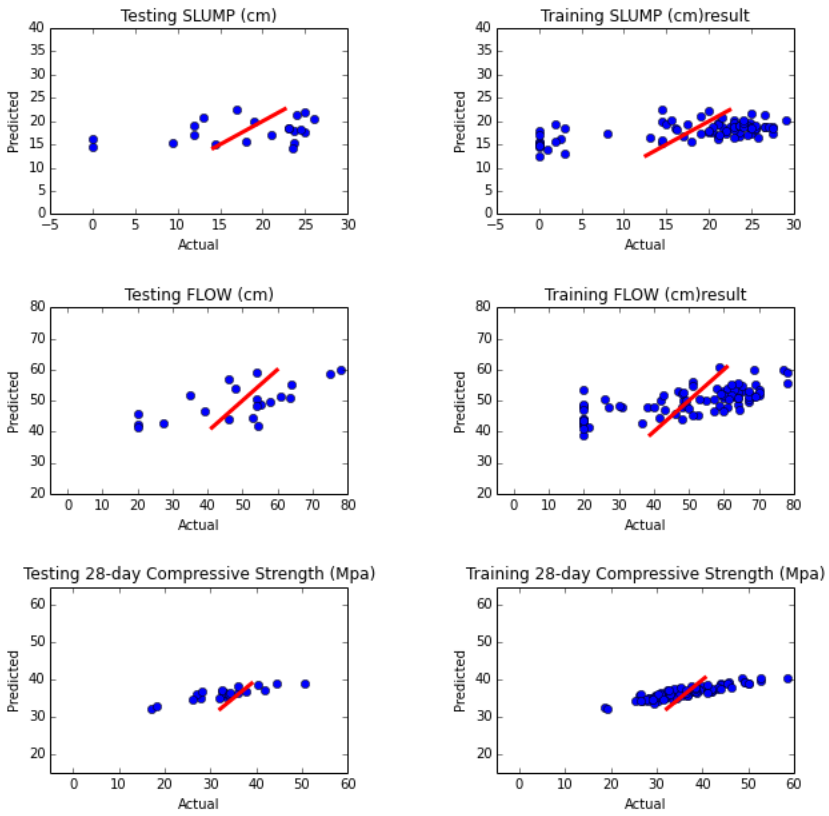

Figure 6: Predicted versus actual data plots when lambda=185

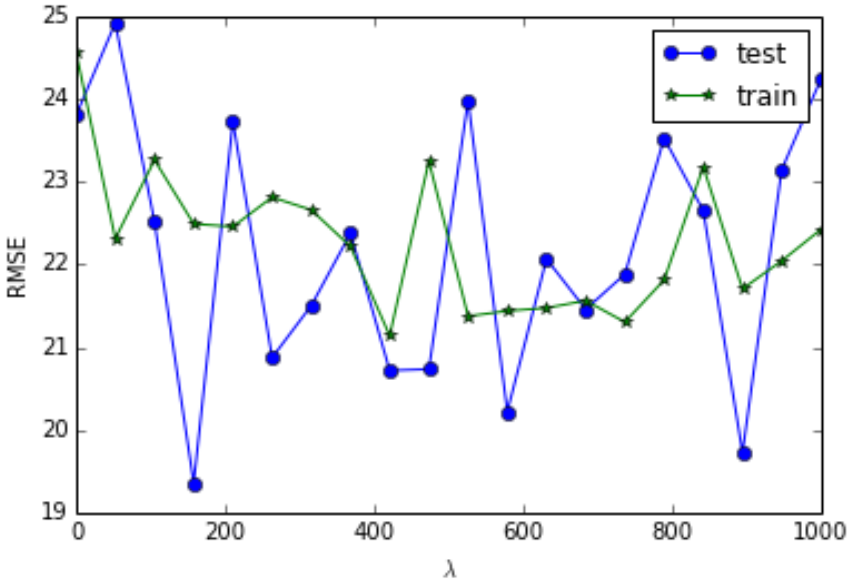

Figure 7: RMSE versus λ

the blue legend represents the performance of test data and the green line represents that of the training data. I had taken 20 sample point in the range of 0 to 1000 for λ. From the figure, it can be seen that testing data performs well than the training data for 11 values of λ. This means that 55 percent of the test data follow the predicted model and the remaining 45 percent is influenced by the λ value. This is due to the RMSE taken over all the 3 target values. But, when we take RMSE for individual target value, the commonality observed is the minimum error that we get is around 19 to 20. The best value of penalty to be selected is nothing but the one corresponding to the **least square error**. From this graph, it can be observed that a value around 180-185 can be selected. This variation can also be due to the fact that the noise in testing data is more when compared to that in the training data. But, this data has been taken for 1000 different random partitioning of data samples.So, the reason can only be due to taking RMSE over all the 3 target variables leads to a fairly accurate linear model. But, this graph doesn't remain constant and varies when I tried various other times. But the ratio of 45 percent and 55 percent for training and testing performance remained constant although the λ corresponding to the least error varied.

4.4 Additional question

Why isn't there a unique λ for best prediction model even after iterating over 1000 different partitioned sets? Why does it vary every time ? I am not sure why there is a huge difference in the λ value every other time. It can be said that variance is huge or the noise varies. But, when we are having just a 103 samples of data, why cannot we find a fixed best value of λ is the additional question that I have. I probably don't understand or have intuition on this. But, the one observation I am sure is that the test data performs only 55 percent times better than the training data. So, I guess it might improve if we carefully select the input data set instead of taking all the attribute. For example, cement and water are the main ingredients. So, forming a prediction model based on this can be more reliable.

5 Conclusion

Thus, the regularized linear least squares has been implemented using python language. A clear demonstration on a sample data and also on a real set data has been provided. It can be concluded that for better

model prediction, the selection of λ , training data are critical and also the useful or most favourable input dependence should be considered.

Bibliography

Christopher Bishop. *Pattern Recognition and Machine Learning.* Springer, 2006.