Project 1.2: Learning to Rank using Linear Regression

Sahil Suhas Pathak
Department of Computer Science and Engineering
University at Buffalo
Buffalo, NY, 14214
sahilsuh@buffalo.edu
UBITName: sahilsuh, Person Number: 50289739

Abstract

The goal of this project is to use machine learning to solve a problem that arises in Information Retrieval, one known as the Learning to Rank (LeToR) problem. We formulate this as a problem of linear regression where we train a linear regression model on LeToR dataset using a closed-form solution and gradient descent solution.

1 Introduction

Our goal is to estimate the relevance of the web search results given a query and document pair. The LETOR training data consists of pairs of input values x and target values t. The input values are real-valued vectors i.e. features, which are derived from query document pairs. The target values are scalars, also called as relevance labels, they take values ranging from 0 to 2. The larger the relevance label, the better is the match between query and document.

Following are the steps illustrating how the implementation is carried out:

1. Importing dataset. Namely two files Querylevelnorm_X.csv and Querylevelnorm_t.csv. Where Querylevelnorm_X.csv stores (n-samples, m-features) and Querylevelnorm_t.csv stores the target values ranging from 0 to 2.

2. Partition the dataset. We partition the dataset into 3 categories, Training, Validation and testing. The default case is when the Training data has 80% share and 10% share for both validation and testing data.

3. For a particular set of hyper parameters, we find the updated weights, where we train a regression model on the Training dataset using a closed-form solution and using Stochastic Gradient Descent Solution.

4. Tune the hyper parameters to have a better performance on the Validation set.

5. Apply the model on the test set and evaluate the accuracy and the error.

2 Problem Statement – Overview of Linear Regression

We want a general way of obtaining a linear model that fits to observed data. Mathematically, we have input values x and target values t, (x_n, t_n) and we have to learn a function y to minimize some error function, say E.

Our linear regression function $y(x, w) = w^T \emptyset(x)$

Where, $w = (w_0, w_1, w_2, \dots w_{m-1})$ which is a weight vector to be learnt from training samples. Also, $\emptyset = (\emptyset_0 \dots \emptyset_{m-1})^T$ is a vector of M basis functions. Here in the default case, we are considering M as 10. Each basis function converts the input vector x into a scalar value. We are using Gaussian radial basis function $\emptyset_i(x) = \exp(-1/2$ * (x - $u_j)^T \sum_j^{-1}(x - u_j))$

3 Closed Form Solution

The closed form solution i.e. sum of squared error without regularization is of the form:

$$w = (\Phi^T \Phi)^{-1} \Phi^T t$$

Where t is the target vector, and $\emptyset$ is the design matrix used to compute the weights.

The design matrix is of the order (55699, 10).

The above form is called as Moore Penrose pseudo inverse of the matrix $\emptyset$.

The closed form solution with least squared regularization is defined as:

$$w = (\lambda I + \Phi^T \Phi)^{-1} \Phi^T t$$

Where λ governs the relative importance of the regularization term. The regularization parameter reduces overfitting.

3.1 Steps for Calculation

We do the following computations in the closed form solution:

1. We find the clusters for the given data.

2. We find the spreads i.e. μ for the given data.

3. We compute the design matrix i.e. ϕ.

4. We start the Training by having initial random set of hyper-parameters and compute W.

5. The ERMS i.e. the Root mean squared error value for the training, validation and test is determined.

6. We tune the hyper-parameters to obtain a better performance and try to reduce ERMS value.

3.2 Understanding the Source Code of Closed Form Solution

The calculation for Weights W is based on these three statements,

3.2.1. BigSigma = GenerateBigSigma(RawData, Mu, TrainingPercent,IsSynthetic)

'''GenerateBigSigma takes RawData, Mu, TrainingPercent and IsSynthetic as input. However, Mu and IsSynthetic does not play any role in the code mentioned below.'''

def GenerateBigSigma(Data, MuMatrix,TrainingPercent,IsSynthetic):

''' Basically we are calculating Covariance Matrix and extracting Variance from the diagonal elements of that Covariance Matrix'''

'''Covariance matrix is where we calculate variance of a feature with every other feature'''

'''We are creating a numpy array called BigSigma of the order (41, 41) as len(Data) = 41 nad len(Data[0])= 69623'''

BigSigma = np.zeros((len(Data),len(Data)))

'''DataT is of the order (69623, 41)'''

DataT = np.transpose(Data)

*'''TrainingLen = 69623 * 80 * 0.01 ~ 55699'''*

```
    TrainingLen = math.ceil(len(DataT)*(TrainingPercent*0.01))
```

'''For every iteration of i, which takes values from 0 to 40, varVect will store the variance of features ranging from 0 to 41 i.e it will store the variance of all 41 features'''

```
    varVect = []
```

'''DataT[0] = 41'''

```
    for i in range(0,len(DataT[0])):
```

'''For every feature, vct will store the number of training samples from RawData. It will store values ranging from 0 to 55699 as TrainingLen is 55699'''

'''In every iteration vct is getting initialized thus considering new feature in every iteration'''

```
        vct = []
        for j in range(0,int(TrainingLen)):
            vct.append(Data[i][j])
    varVect.append(np.var(vct))
```

#Variance is how far a set of numbers are spread out from their average value

'''Up until this point, varVect will have 41 values corresponding to variances of 41 features'''

'''The following loop will store all the variances of 41 features into the diagonal elements of the matrix BigSigma'''

'''We are doing so because in order to calculate the gaussain radial basis function, we want to invert this matrix BigSigma'''

'''BigSigma is of the order (41, 41)'''

'''Since IsSynthetic is always false, Following code randomly multiplies a scalar value 200 with BigSigma '''

```
    for j in range(len(Data)):
        BigSigma[j][j] = varVect[j]
    if IsSynthetic == True:
        BigSigma = np.dot(3,BigSigma)
    else:
        BigSigma = np.dot(200,BigSigma)
    #print ("BigSigma Generated..")
    return BigSigma
```

3.2.2. TRAINING_PHI = GetPhiMatrix(RawData, Mu, BigSigma, TrainingPercent)

```
def GetPhiMatrix(Data, MuMatrix, BigSigma, TrainingPercent = 80):
```

'''Example: DataT is of the order (69623, 10) for TrainingData'''

```
    DataT = np.transpose(Data)
```

'''For TrainingData, TrainingLen = 55699'''

```
    TrainingLen = math.ceil(len(DataT)*(TrainingPercent*0.01))
```

'''PHI is of the order (55699, 10)'''

```
    PHI = np.zeros((int(TrainingLen),len(MuMatrix)))
```

'''Calculating BigSigInv to compute the term Sigma Inverse from the Gaussian radial basis function vector form'''

```
    BigSigInv = np.linalg.inv(BigSigma)
```

'''len(MuMatrix) = 10 and TrainingLen = 55699'''

```
    for  C in range(0,len(MuMatrix)):
    for R in range(0,int(TrainingLen)):
```

'''R takes the values from 0 to 55698 and C takes the values from 0 to 9'''

''' GetRadialBasisOut *calculates the vector form of Gaussian radial basis function and returns a scalar value'''*

```
        PHI[R][C] = GetRadialBasisOut(DataT[R], MuMatrix[C], BigSigInv)
    return PHI
```

```
def GetRadialBasisOut(DataRow,MuRow, BigSigInv):
```

'''phi_x calculates the vector form of gaussian radial basis function'''

```
    phi_x = math.exp(-0.5*GetScalar(DataRow,MuRow,BigSigInv))
     return phi_x
```

'''GetScalar accepts DataRow i.e x in terms of our gaussain radial basis function vector form, MuRow i.e Mu, and BigSigInv which is nothing but the inverse of BigSigma'''

```
def GetScalar(DataRow,MuRow, BigSigInv):
```

'''R is calculating (x-Mu)'''

```
    R = np.subtract(DataRow,MuRow)
```

'''T is calculating (BigSigmaInverse.transpose(x-Mu))'''

```
    T = np.dot(BigSigInv,np.transpose(R))
    L = np.dot(R,T)
    return L
```

3.3.3. W = GetWeightsClosedForm(TRAINING_PHI,TrainingTarget,(C_Lambda))

'''GetWeightsClosedForm gives us the values of the updated weights. It accepts PHI, T as TrainingTarget, Lambda = 0.03 PHI is of the order (55699, 10) for TrainingData. This basically calculates the Weights using Moore- Penrose pseudo- Inverse Matrix notation '''

```
def GetWeightsClosedForm(PHI, T, Lambda):
```

'''Considering an identity matrix of the order (10, 10)'''

```
    Lambda_I = np.identity(len(PHI[0]))
    for i in range(0,len(PHI[0])):
        Lambda_I[i][i] = Lambda
```

'''Example: Calculating PHI_T which is of the order (10, 55699) for TrainingData'''

```
    PHI_T       = np.transpose(PHI)
```

'''Calculating PHI_SQR which is the dot product of PHI_T and PHI, order is (10, 10)'''

```
    PHI_SQR     = np.dot(PHI_T,PHI)
```

*'''Calculating PHI_SQR_LI which corresponds to the term ((lambda*I)+(transpose(PHI).PHI)). It is of the order (10, 10)'''*

```
    PHI_SQR_LI = np.add(Lambda_I,PHI_SQR)
```

'''Calculating inverse of PHI_SQR_LI'''

```
    PHI_SQR_INV = np.linalg.inv(PHI_SQR_LI)
```

'''Calculating INTER which is nothing but (PHI_SQR_INV.PHI_T), it is of the order (10, 55699)'''

```
    INTER = np.dot(PHI_SQR_INV, PHI_T)
```

'''Calculating updated weights, of the order (10, 1)'''

```
    W = np.dot(INTER, T)
    return W
```

gives us the updated weights for Training that we can use in subsequent iterations. Likewise, we calculate VAL_PHI for Validation set and TEST_PHI for Test set.

After varying multiple lambda values, we determine optimal W for the data. We do that by determining minimum error for the validation set i.e. choose lambda which yields minimum error for validation data set and thus compute optimal W for the same.

Let us consider the following illustration where we compute the order of W for TRAINING_PHI.

We have $w = (\phi^T\phi)^{-1}\phi^T t$; this form is also called as Moore Penrose pseudo inverse of Matrix ϕ.

let us consider Training Data which is of the order $(55,000 \times 10)$

Here we approximate 55699 as 55,000 for our calculations below.

Now, we have $\phi \approx (\underline{55,000} \times 10)$

Thus $\phi^T = (10 \times 55,\underline{000})$

$(\phi^T\phi) = (10 \times 10)$

& $(\phi^T\phi)^{-1}\phi^T = (10 \times 55,000)$

Also, $t = (55,000 \times 1)$

Thus, $(\phi^T\phi)^{-1}\phi^T t = 10 \times 1$

Thus, $\boxed{W = (10 \times 1)}$

This W gives us the updated weights, & t is our target Vector ranging from 0 to 2.

But suppose we consider;

$\phi \approx (10 \times 55,000)$

Then $\phi^T \approx (55,000 \times 10)$

i-e $(\phi^T\phi) \approx (55,000 \times 55,000)$

& $(\phi^T\phi)^{-1}\phi^T \approx (55,000 \times 10)$

Also, $t = (55,000 \times 1)$

But since, $(\phi^T\phi)^{-1}\phi^T$ is $(55,000 \times 10)$ & $t = (55,000 \times 1)$

We cannot really obtain their dot product, thus we have to consider ϕ of the order $(55,000 \times 10)$ only.

Illustrating the calculation of the order of W for two different possible orders of TRAINING_PHI i.e. (55699, 10) and (10, 55699)

4 Stochastic Gradient Descent Solution

The Stochastic Gradient Descent Algorithm first takes a random initial weight. Then it updates the weight value as:

$$w^{\tau+1} = w^{\tau} + \Delta w^{\tau}$$

Where, $\Delta w^{\tau} = -\eta^{\tau}\nabla E$, Δw^{τ} is the weight update parameter. The minus sign indicates that the computations go along the opposite direction of the gradient of the error.

η^{τ} is the learning rate and $\nabla E = \nabla E_D + \lambda \nabla E_w$

Also, $\nabla E_D = -(t_n - w^T \emptyset(x_n))\emptyset(x_n)$, is nothing but the derivative of the

$E_D = \frac{1}{2}\sum_1^n (t_n - w^T \emptyset(x_n))^2$ and $\nabla E_w = w$

4.1 Steps for Calculation

1. We have the Design matrix $\emptyset$ from Closed Form Solution.
2. We calculate the following terms in the order, ∇E_D, $\lambda\nabla E_w$, ∇E, Δw^{τ}, $w^{\tau+1}$.
4. Using these values, we train our model with initial set of hyper-parameters.
5. The ERMS i.e. the Root mean squared error value for the training, validation and test is determined.

6. We tune the hyper-parameters to obtain a better performance and try to reduce ERMS value.

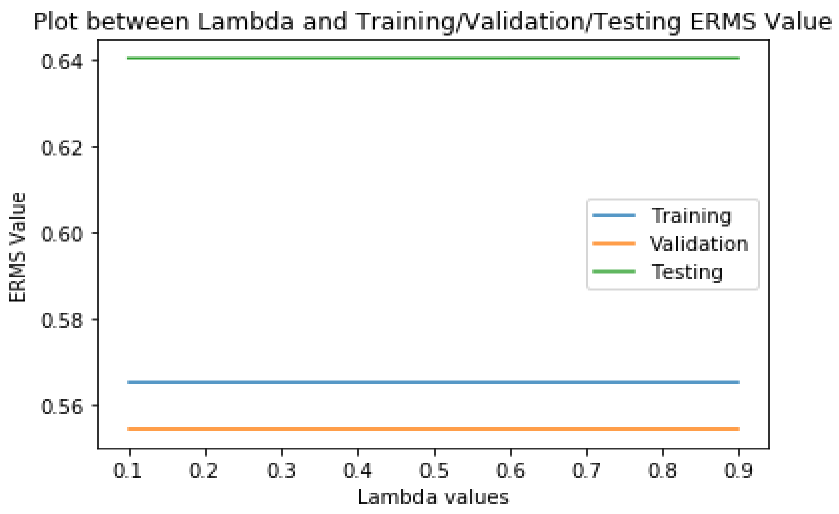
5 Experiments

For a particular value of M evaluating the effect of λ on Training, Validation and Testing ERMS Value

	M = 1		
λ	Training ERMS	Validation ERMS	Testing ERMS
0.1	0.5651378688423746	0.5543829406814273	0.6402356829691717
0.2	0.5651378688435513	0.5543829279319872	0.6402357601204745
0.3	0.565137868845509	0.5543829151834043	0.6402358372721239
0.4	0.5651378688482578	0.5543829024356742	0.6402359144241183
0.5	0.5651378688517864	0.5543828896888023	0.640235991576458
0.6	0.5651378688560998	0.554382876942785	0.6402360687291448
0.7	0.5651378688611969	0.5543828641976237	0.6402361458821784
0.8	0.5651378688670756	0.5543828514533176	0.6402362230355565
0.9	0.5651378688737423	0.5543828387098652	0.6402363001892816

Figure 1

Inference: For M = 1, we calculated the ERMS value for Training, Validation and for Testing set. But the values are fairly constant even when we increase the λ in subsequent iteration.

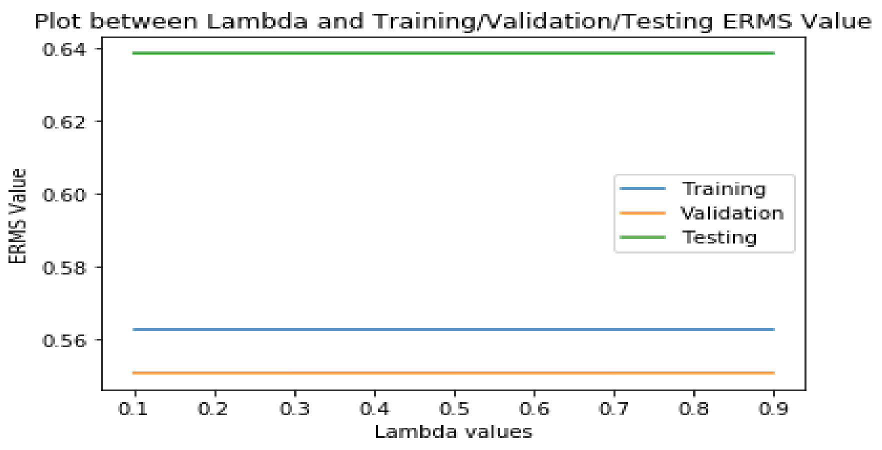


When M = 1, Relationship between Lambda and ERMS for Training Set, Validation Set and Testing Set

	M = 2		
λ	**Training ERMS**	**Validation EMRS**	**Testing ERMS**
0.1	**0.5627948149944119**	**0.5509387527165319**	**0.638449182307033**
0.2	**0.5627948183333238**	**0.5509395553129974**	**0.6384489991213015**
0.3	**0.5627948238859702**	**0.5509403588784759**	**0.6384488181998508**
0.4	**0.5627948316424212**	**0.5509411634059046**	**0.6384486395332776**
0.5	**0.5627948415927972**	**0.5509419688882601**	**0.6384484631122302**
0.6	**0.5627948537272447**	**0.5509427753185476**	**0.6384482889273918**
0.7	**0.5627948680359659**	**0.5509435826898135**	**0.6384481169694886**
0.8	**0.5627948845092149**	**0.5509443909951363**	**0.6384479472292846**
0.9	**0.5627949031372693**	**0.5509452002276315**	**0.6384477796975866**

Figure 2

Inference: For M = 2, the Validation ERMS and the testing ERMS are lesser when compared to the values from M = 1, but again the values are fairly constant with respect to every other value in M = 2.

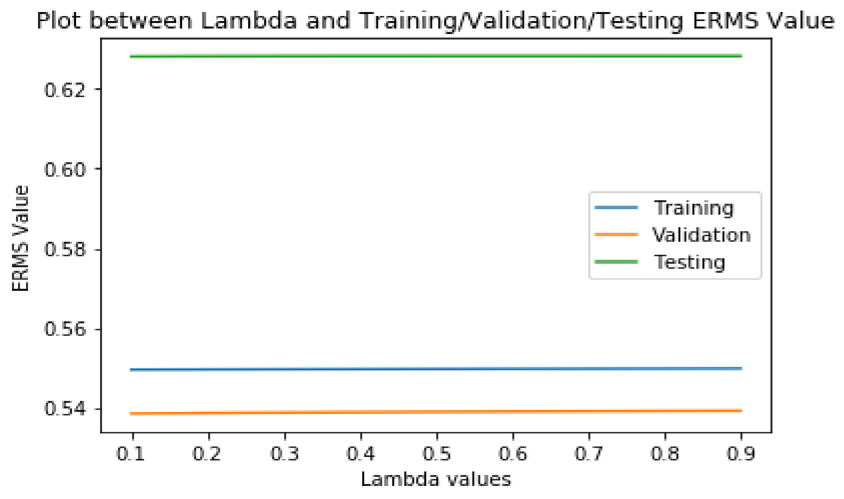


When M = 2, Relationship between Lambda and ERMS for Training Set, Validation Set and Testing Set

	M = 10		
λ	**Training ERMS**	**Validation EMRS**	**Testing ERMS**
0.1	**0.5495138127085464**	**0.5385150200433642**	**0.6281151520201143**
0.2	**0.5495684048344792**	**0.5386478517405763**	**0.6281846747729138**
0.3	**0.5496184028685661**	**0.5387635628826766**	**0.6282134745004091**
0.4	**0.5496652683214609**	**0.5388629310082516**	**0.6282266300255681**
0.5	**0.549709369084412**	**0.5389490530962439**	**0.6282321603009604**
0.6	**0.5497509568481201**	**0.5390246156543649**	**0.6282335875446705**
0.7	**0.5497902629351704**	**0.5390916977152266**	**0.6282327270435593**
0.8	**0.549827498063084**	**0.5391518774092154**	**0.6282306006752241**
0.9	**0.5498628487549053**	**0.5392063570107043**	**0.6282278135552045**

Figure 3

Inference: For M=10, the Validation ERMS and the testing ERMS are lesser when compared to the values from M = 2, but again the values are fairly constant with respect to every other value in M = 10.



When M = 10, Relationship between Lambda and ERMS for Training Set, Validation Set and Testing Set

Note: But for the configuration M = 10 and Lambda = 0.1, when we change the statement BigSigma = np.dot(200, BigSigma) to BigSigma = np.dot(0.1, BigSigma), BigSigma = np.dot(0.2,BigSigma) we observe a drastic change in the ERMS values.

Let us consider k as the parameter that is being multiplied with BigSigma:

	M = 10 and Lambda = 0.1		
k	**Training ERMS**	**Validation ERMS**	**Testing ERMS**
0.1	**0.6427529010663371**	**0.6282628128039963**	**0.7408882396348179**
0.2	**0.642730757436021**	**0.6282427363557429**	**0.7408596692354525**
0.5	**0.6402282222806086**	**0.6248672710796914**	**0.7387994459911353**
1	**0.6321809884034857**	**0.6169407483801693**	**0.730934536665152**

Figure 4

Gradient Descent Observations for the above configuration also W_Now = np.dot (200, W), Lambda = 0.7, learningRate = 0.01:

	M = 10 and Lambda = 0.7		
k	**Training ERMS**	**Validation ERMS**	**Testing ERMS**
0.1	**0.64275**	**0.62826**	**0.74089**
0.2	**0.64253**	**0.6272**	**0.74082**
0.5	**1.38197**	**1.28548**	**1.72439**
1	**2.65627**	**2.72893**	**2.83129**

Figure 5

Inference: From figure 4, we can observe that when BigSigma which stores the variances of all the features, is multiplied by a small constant say 0.1, 0.2 .. there is an adverse effect on the ERMS value. Probably because multiplying by 1/10 or 1/20 to a variance value makes it even more smaller. Also, in figure 5, we can see for k = 0.5, 1 there are very vague ERMS values observed.

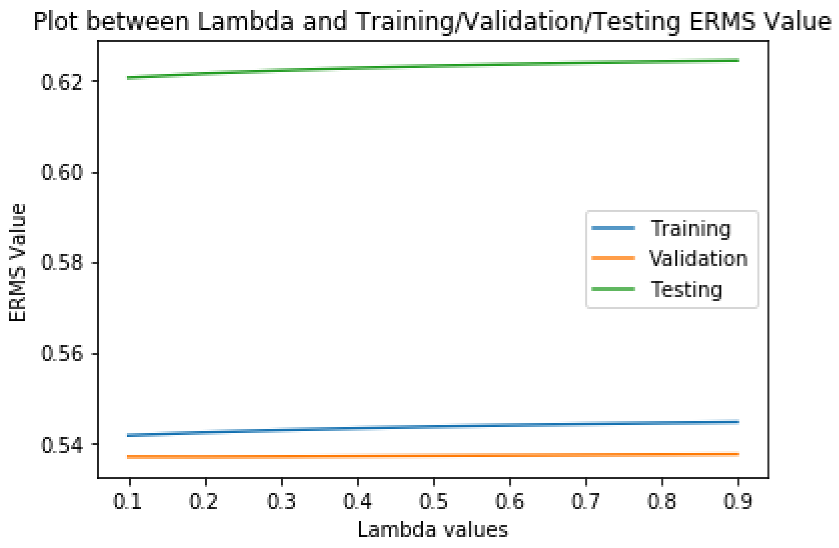
Note: Generally, the Closed Form solution would give us the exact or relatively better estimate of ERMS value when compared to the Stochastic Gradient Descent Solution. This is because the Closed Form solution is based purely on mathematical basis. They involve certain number of functions and computations and they are finite. Whereas, the Stochastic Gradient Descent solution involves little bit of randomization as well of approximation. Because of this uncertainty we can observe that the ERMS value of Closed Form solution is generally lesser than that of Stochastic Gradient Descent Solution.

	M = 50		
λ	**Training ERMS**	**Validation ERMS**	**Testing ERMS**
0.1	**0.5415495064646513**	**0.5368214467110219**	**0.6207069259022228**
0.2	**0.542222244387187**	**0.5368020591528851**	**0.6216302197012522**
0.3	**0.5427325742967881**	**0.5368616443639317**	**0.622316736279007**
0.4	**0.5431523159417327**	**0.5369482117437645**	**0.6228627657197855**
0.5	**0.5435085763629482**	**0.5370427685339039**	**0.6233118328573474**
0.6	**0.5438166648151262**	**0.5371374924689388**	**0.6236894427669977**

0.7	**0.5440866977755379**	**0.5372290214437269**	**0.6240122824831443**
0.8	**0.5443258928802301**	**0.5373159683058648**	**0.6242919659854715**
0.9	**0.5445396340935544**	**0.5373978726541355**	**0.6245369147281961**

Figure 6

Inference: Even when M = 50 and calculating multiple ERMS values for different λ's, the values within the table are fairly constant. Only when we change the scaling factor for BigSigma in the statement BigSigma = np.dot(200, BigSigma) we can see changes in the ERMS value. Being said, the ERMS values for M = 50 are lesser when compared to the values when M was 10 (Figure 3).



When M = 50, Relationship between Lambda and ERMS for Training Set, Validation Set and Testing Set

Note: For M = 50, Lambda = 0.1 if BigSigma = np.dot(**0**,BigSigma) then we get a **Singular Matrix error**. In this case, BigSigma has a determinant which is 0 thus we cannot compute the inverse of BigSigma, consequently we are unable to find the PHI matrix.

	M = 100		
λ	**Training ERMS**	**Validation ERMS**	**Testing ERMS**
0.1	**0.5396590932342948**	**0.53618499346096**	**0.6193171138462322**
0.2	**0.5402240815968818**	**0.5363055236548694**	**0.6196939051918788**

Figure 7

Inference: Interestingly, when we gradually increased M from 1, 2, 10, 50 and now to 10 we have seen a gradual decrease in the Testing ERMS value (Refer Figure 1, 2, 3, and 6).

	M = 56		
λ	**Training ERMS**	**Validation ERMS**	**Testing ERMS**
0.1	**0.5405196564387902**	**0.5364211360479836**	**0.6197675649990406**
0.9	**0.5435138669862053**	**0.5371603510749552**	**0.623470954172003**

Figure 8

Vague computations:

Case 1 - M = 25

Lambda = 1000

E_rms Training = 0.555508447089771

E_rms Validation = 0.5441542959250263

E_rms Testing = 0.6322224740890616

Case 2 - M = 25

Lambda = 1000000

E_rms Training = 0.585379610712308

E_rms Validation = 0.5721964397353346

E_rms Testing = 0.6751122806240362

Case 3 - M = 1

Lambda = 1000

E_rms Training = 0.5651754254657633

E_rms Validation = 0.5542964905268477

E_rms Testing = 0.6410233028780147

Note: In all the above cases, we have increased λ to a very big number but still the ERMS values are relatively similar when we compare them to the ERMS values we got when M was 1, 2, 10, 50 and 100. Moreover, in Case 3, when M = 1 and Lambda = 1000, we were expecting a very low ERMS value but still it remained as 0.6410.

6 Execution

Running main.ipynb file will print the ERMS error value for partitioned data i.e. for Training, Validation and Testing set, for closed-form solution as well as for Stochastic Gradient Descent solution. Also, it will plot the necessary graphs so as to illustrate how EMRS value remains relatively constant for M = 1, 2, 10 ,50 and 100.

7 References

Teaching Assistants – CSE 574 – Introduction to Machine Learning

https://stats.stackexchange.com/questions/117556/understanding-gaussian-basis-function-parameters-to-be-used-in-linear-regression

https://pythonprogramming.net/matplotlib-python-3-basics-tutorial/