Learning to Rank using Linear Regression

Haoquan Zhao

50189594

haoquanz@buffalo.edu

1. Abstract

In this project, we will using machine learning approach to solve the Learning to Rank (LeToR) problems on a real dataset. We train our model using the training set which comprises of 80% of the given data set and validate it using the 10 % validation data set and finally test it using the 10% testing data set.

2. Introduction

In this project, we apply linear regression on a dataset from Microsoft LETOR 4.0, train our model on part of it and evaluate the performance on another part, then by tuning Hyper-Parameters, we get to know with which parameter setting can get the minimum error. And apply the setting into test part of dataset. And finally, by using Root Mean Square error to evaluate the solution on a test part of the dataset, we could find out how the solution works.

The linear regression function is

$$y(\mathbf{x}, \mathbf{w}) = \mathbf{w}^\top \boldsymbol{\phi}(\mathbf{x})$$

where the w is a weight vector to be learn from training samples and Φ is a vector of M basis functions, and the Gaussian radial basis function is shown below:

$$\phi_j(\mathbf{x}) = \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_j)^\top \Sigma_j^{-1}(\mathbf{x} - \boldsymbol{\mu}_j)\right)$$

where μj is the center of the basis function and Σj decides how broadly the basis function spreads.

Closed form solution is calculated using basis function, Sigma, lambda and M. Its equation is:

$$\mathbf{w}_{\mathrm{ML}} = (\lambda \mathbf{I} + \boldsymbol{\Phi}^\top \boldsymbol{\Phi})^{-1} \boldsymbol{\Phi}^\top \mathbf{t}$$

For the Stochastic Gradient Descent Solution，We start with an initial random value of solution and then modify this solution for each data sample.

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} + \Delta \mathbf{w}^{(\tau)}$$

Where

$$\Delta \mathbf{w}^{(\tau)} = -\eta^{(\tau)} \nabla E$$

$$\nabla E = \nabla E_D + \lambda \nabla E_W$$

In which

$$\nabla E_D = -(t_n - \mathbf{w}^{(\tau)\top} \boldsymbol{\phi}(\mathbf{x}_n)) \boldsymbol{\phi}(\mathbf{x}_n)$$

$$\nabla E_W = \mathbf{w}^{(\tau)}$$

Then we can calculate the error by using:

$$E_{\mathrm{RMS}} = \sqrt{2E(\mathbf{w}^{*})/N_{\mathrm{V}}}$$

3. Process Data

In the code below, we do Partition to decide the whole dataset into three parts. The first one is used to train our model, so that we could generate the optimal parameter set as expected. The second part is reserved as test dataset, with which we could evaluate the efficiency of our model. 80% of the dataset is used as train data, 10% of the dataset is used as valid data, and the rest 10% is used for test. In the dataset, there are 69623 query-document pairs each consisting of 46features. The last part is called valid part, which is used as a confirmation that the model we build by training the train part dataset is reliable.

```
def GenerateTrainingTarget(rawTraining,TrainingPercent = 80):
    TrainingLen = int(math.ceil(len(rawTraining)*(TrainingPercent*0.01)))
    t           = rawTraining[:TrainingLen]
    #print(str(TrainingPercent) + "% Training Target Generated..")
    return t

#In this function , we split 80% of my input data for training as Training data and save it into d2
def GenerateTrainingDataMatrix(rawData, TrainingPercent = 80):
    T_len = int(math.ceil(len(rawData[0])*0.01*TrainingPercent))
    d2 = rawData[:,0:T_len]
    #print(str(TrainingPercent) + "% Training Data Generated..")
    return d2

#In this function, we split 10% of my input data for validation as validation data and save it into dataMatrix
def GenerateValData(rawData, ValPercent, TrainingCount):
    valSize = int(math.ceil(len(rawData[0])*ValPercent*0.01))
    V_End = TrainingCount + valSize
    dataMatrix = rawData[:,TrainingCount+1:V_End]
    #print (str(ValPercent) + "% Val Data Generated..")
    return dataMatrix
```

In code below, we reading the dataset we need to use to do LeToR. "Querylevelnorm_x" is the input dataset with 46 features. "Querylevelnorm_t" is the target dataset which has the expected result of the input dataset.

```
In [4]: #Reading the target data and output data
        RawTarget = GetTargetVector('Querylevelnorm_t.csv')
        RawData   = GenerateRawData('Querylevelnorm_X.csv',IsSynthetic)
```

4. Closed Form Solution

Closed form solution is calculated using basis function, Sigma, lambda and M.

$$\mathbf{w}_{\mathrm{ML}} = (\lambda\mathbf{I} + \mathbf{\Phi}^\top\mathbf{\Phi})^{-1}\mathbf{\Phi}^\top\mathbf{t}$$

In the code below, first we pick up M (which is 10 in this case) basis functions, and set centers and spreads of them. This process could also be accomplished by K-means function.

```
In [8]: ErmsArr = []
        AccuracyArr = []
        #Create M number of cluster.
        kmeans = KMeans(n_clusters=M, random_state=0).fit(np.transpose(TrainingData))
        #Set Mu as the center of the cluster
        Mu = kmeans.cluster_centers_
```

Then implement the closed form solution

```
def GetWeightsClosedForm(PHI, T, Lambda):
    Lambda_I = np.identity(len(PHI[0]))
    for i in range(0,len(PHI[0])):
        Lambda_I[i][i] = Lambda
    PHI_T       = np.transpose(PHI)
    PHI_SQR     = np.dot(PHI_T,PHI)
    PHI_SQR_LI  = np.add(Lambda_I,PHI_SQR)
    PHI_SQR_INV = np.linalg.inv(PHI_SQR_LI)
    INTER       = np.dot(PHI_SQR_INV, PHI_T)
    W           = np.dot(INTER, T)
    ##print ("Training Weights Generated..")
    return W
```

Then we are able to calculate the accuracy fir the training, testing and validation accuracy by applying the Erms function ERMS $E_{\mathrm{RMS}} = \sqrt{2E(\mathbf{w}^*)/N_{\mathrm{V}}}$

```
def GetErms(VAL_TEST_OUT,ValDataAct):
    sum = 0.0
    t=0
    accuracy = 0.0
    counter = 0
    val = 0.0
    for i in range (0,len(VAL_TEST_OUT)):
        sum = sum + math.pow((ValDataAct[i] - VAL_TEST_OUT[i]),2)
        if(int(np.around(VAL_TEST_OUT[i], 0)) == ValDataAct[i]):
            counter+=1
    accuracy = (float((counter*100))/float(len(VAL_TEST_OUT)))
    ##print ("Accuracy Generated..")
    ##print ("Validation E_RMS : " + str(math.sqrt(sum/len(VAL_TEST_OUT))))
    return (str(accuracy) + ',' +  str(math.sqrt(sum/len(VAL_TEST_OUT))))
```

```
In [10]: TR_TEST_OUT  = GetValTest(TRAINING_PHI,W)
         VAL_TEST_OUT = GetValTest(VAL_PHI,W)
         TEST_OUT     = GetValTest(TEST_PHI,W)

         TrainingAccuracy   = str(GetErms(TR_TEST_OUT,TrainingTarget))
         ValidationAccuracy = str(GetErms(VAL_TEST_OUT,ValDataAct))
         TestAccuracy       = str(GetErms(TEST_OUT,TestDataAct))
```

5. Stochastic Gradient Descent Solution

Then we define how dose SGD works in the dataset, firstly, we define the Delta_E_D, implement equation $\Delta \mathbf{w}^{(\tau)} = -\eta^{(\tau)} \nabla E$. In which, we also get $\Delta \mathbf{w}^{(\tau)} = -\eta^{(\tau)} \nabla E$, which is called weight update, it goes against with direction of gradient of the error. As well as delta_E equation, then we are able to update the value of w.

```
#print ('---------Iteration: ' + str(i) + '--------------')
#Implement the equation 11
Delta_E_D     = -np.dot((TrainingTarget[i] - np.dot(np.transpose(W_Now),TRAINING_PHI[i])),TRAINI
#Implement  La_Delta_E_W= λ∇E
La_Delta_E_W  = np.dot(La,W_Now)
#Implement ∇E = ∇ED + λ∇E
Delta_E       = np.add(Delta_E_D,La_Delta_E_W)
#Implement  Δw(τ) = −η(τ)∇
Delta_W       = -np.dot(learningRate,Delta_E)

#Implement equation w(τ+1) = w(τ) + Δw(τ)
W_T_Next      = W_Now + Delta_W
W_Now         = W_T_Next
```

Then, we can print out the Gradient Descent result.

```
#-----------------TrainingData Accuracy---------------------#
TR_TEST_OUT   = GetValTest(TRAINING_PHI,W_T_Next)
Erms_TR       = GetErms(TR_TEST_OUT,TrainingTarget)
L_Erms_TR.append(float(Erms_TR.split(',')[1]))

#-----------------ValidationData Accuracy---------------------#
VAL_TEST_OUT  = GetValTest(VAL_PHI,W_T_Next)
Erms_Val      = GetErms(VAL_TEST_OUT,ValDataAct)
L_Erms_Val.append(float(Erms_Val.split(',')[1]))

#-----------------TestingData Accuracy---------------------#
TEST_OUT      = GetValTest(TEST_PHI,W_T_Next)
Erms_Test = GetErms(TEST_OUT,TestDataAct)
L_Erms_Test.append(float(Erms_Test.split(',')[1]))
```

```
print ('----------Gradient Descent Solution--------------------')
print ("M = 15 \nLambda  = 0.0001\neta=0.01")
print ("E_rms Training   = " + str(np.around(min(L_Erms_TR),5)))
print ("E_rms Validation = " + str(np.around(min(L_Erms_Val),5)))
print ("E_rms Testing    = " + str(np.around(min(L_Erms_Test),5)))
```

6. Hyper parameter Tuning

Comparison of different M and λ:

Testing root mean square for LeToR:

M \λ	0.01	0.05	0.1	0.15	0.25	0.35	0.5
2	0.552950332226	0.552951915421	0.552953888716	0.552955855476	0.55295976978	0.552963658754	0.5529654445471
4	0.552950332226	0.552951915421	0.552953888716	0.552955855476	0.55295976978	0.552963658754	0.552969445471
6	0.552950332226	0.552951915421	0.552953888716	0.552955855476	0.55295976978	0.552963658754	0.552969445471
8	0.552950332226	0.552951915421	0.552953128716	0.552955855476	0.55295976978	0.552963658754	0.552969445471
10	0.552950332226	0.552951915421	0.552953888716	0.552955855476	0.55295976978	0.552963658754	0.552969445471

12	0.552950332226	0.552951915491	0.552953888716	0.552955855475	0.55295976978	0.552963685754	0.552969445311
14	0.552950332226	0.552951915491	0.552953888716	0.552955855476	0.55295976978	0.552963658754	0.552969445471
16	0.552950332226	0.552951915491	0.552953888716	0.552955855476	0.55295976978	0.552963658754	0.552969445471

SGD testing root mean square error for LeToR:

M \λ	0.01	0.05	0.1	0.15	0.25	0.35	0.5
2	0.566869312151	0.565565952783	0.564739834612	0.564406149085	0.564781425873	0.566101503304	0.56910009271
4	0.567346231397	0.56555903823	0.564740197552	0.564435132955	0.564781425867	0.566101503304	0.56910009271
6	0.566639926989	0.565926614186	0.5647396514	0.564406097985	0.564781425868	0.566101503304	0.56910009271
8	0.566796787282	0.565572287126	0.564740304495	0.564406141412	0.564781425837	0.566101503304	0.56910009271
10	0.567065466021	0.565561019734	0.564740438979	0.56440610258654	0.564781425828	0.566101503304	0.56910009271
12	0.567024344147	0.565564157065	0.564739847065	0.564406154582	0.564781425848	0.56610150330654	0.56910009271
14	0.566615282005	0.565553023072	0.564739405078	0.56440610672225	0.56478142587	0.566101503304	0.56910009271
16	0.566754299209	0.565547730639	0.564740294215	0.5644061786625	0.564781425837	0.566101503304	0.56910009271

According to the result, the best M and λ setting are M= 12, λ=0.05

Comparison of different learning rate setting if M and lambda is fixed:

η	0.01	0.02	0.05	0.1	0.2	0.3
Erms	0.5650143124	0.5651649951	0.5730526018	0.6068018019	0.6367228218	0.6478911416

Hence, when the M and lambda are fixed, set learning rate to 0.01, we will have the minimum root mean square value.

Comparison of different variance scaling

Variance Scaling	10	50	100	150	200	250
Erms	0.5651649951	0.62787559952	0.6283896363	0.630189622	0.6367228218	0.638296085

According to the result, we know that the mean square value is getting bigger when the variance scaling increase.

SGD testing root mean square error for LeToR:

Variance Scaling	10	50	100	150	200	250
Erms	0.6476	0.64295	0.63293	0.63018	0.63015	0.63012

According to the result, the different variance scaling doesn't seems to have a big influence on the Stochastic Gradient Descent Solution.