Project Report
Linear regression on synthetic and LETOR datasets

Submitted by
Hima Sujani Adike, 50246828
Anjali Sujatha Nair ,50248735
Soumya Venkatesan, 50246599

Problem Statement:

Objective is to implement a linear regression model to rank successfully on LETOR dataset and synthetic datasets using closed form solution and stochastic gradient descent methods.

Approach:

There are 2 different approaches as per the requirement

1. Linear regression using closed form
2. Linear regression using stochastic gradient descent

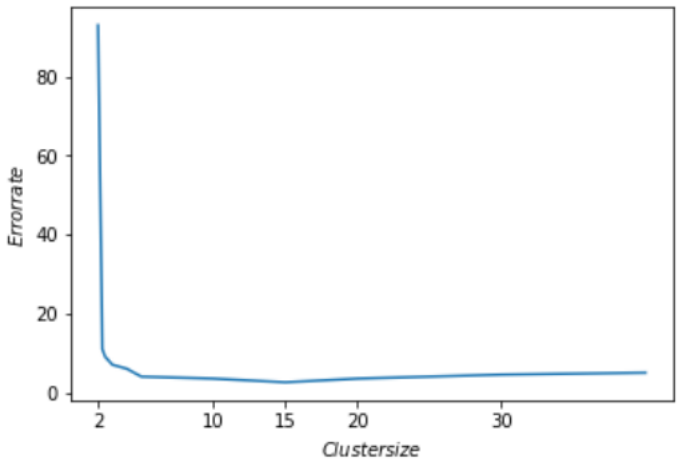
Method 1:

By observing the results Method 1 is preferred only for the smaller datasets like synthetic data of 16,000 training samples. In closed form we use

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

to obtain the weight vector, $\Phi^T\Phi$ and calculating design matrix using gaussian function $(x-\mu_j)^T\Sigma_j^{-1}(x-\mu_j)$ will involve lot of computations and require lot of memory to hold the values.

For M=15, lambda=0.2, learning rate=0.01 we got the least error rate for synthetic dataset.

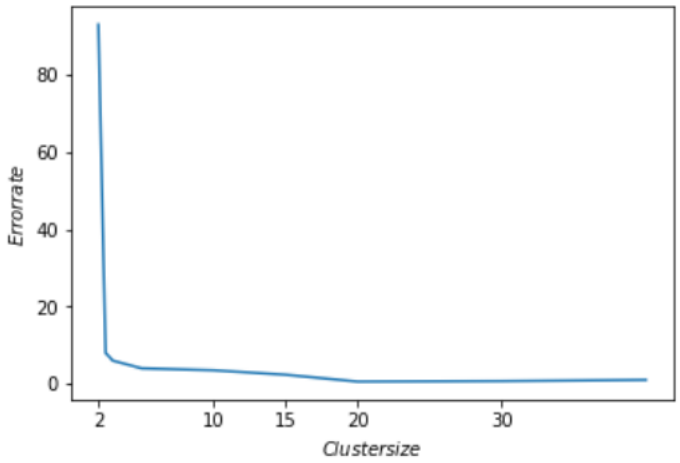


Method 2:

Method 2 is preferred for large datasets like for LeToR dataset. The processing of entire dataset (in case of larger datasets) to obtain the model will be costly and the number of computations required will increase as the dataset increases in method 1 and whereas in stochastic, we are updating model for smaller sets of training samples there is much computational performance gain over method 1. It converges faster as the weights are updated frequently by calculating error.

And using the Early stop algorithm make model better fit and also used to avoid overfitting by calculating both training and validation error and comparing it, such that if training error decreases and validation error starts to increase we stop training the model indicating it is overfitting. In code we are starting with a patience value of 10 and doubling the parameters starting with lambda 0.1 and learning rate of 0.01.

For M=20, lambda=0.1, learningrate=0.01 we got the least error rate for synthetic dataset.



Implementation

The code involves the following files

main.py : The main file to be executed. Calls the other files and reports the root mean squared error for LETOR and synthetic test datasets using closed form solution and SGD
train_closedform_letor.py : Train model using linear regression and update weights using closed form solution for LETOR dataset
train_closedform_syn.py : Train model using linear regression and update weights using closed form solution for synthetic dataset
train_stochastic.py : Train model using linear regression and update weights using stochastic gradient descent for LETOR dataset
train_stochastic_synthetic.py : Train model using linear regression and update weights using stochastic gradient descent for synthetic dataset
testLR.py : Once the model is trained using closed form, evaluates the model on the test set for LETOR and synthetic data sets and reports the error back to main
teststo.py : Once the model is trained using stochastic, evaluates the model on the test set for LETOR and synthetic data sets reports the error back to main

Detailed descriptions of each class follows

Main.py

Class to be called for execution of linear regression. Reads the csv files input.csv,output.csv, Querylevelnorm_X.csv and Querylevelnorm_t.csv required for synthetic and LETOR datasets.
Trains the Linear regression model for both LETOR and synthetic datasets and obtain weights, lambda, cluster size, centroids, spreads and learning rate.
Prints the root mean squared error after testing.

train_closedform_letor.py

Used for obtaining closed form solution for linear regression for LETOR dataset.
main methods are
def getSpreads(data,M,clusterIndexes)
def getDesignMatrix(data,M)
def getWeights(lambdaval,M,phi,trainingOutput)
def trainClosedForm(data,outputLabels)

def trainClosedForm(data,outputLabels) :
The main method is the trainClosedForm , called from main.py. The input LETOR dataset and the output LETOR dataset are passed to the function as arguments. In trainClosedForm, each of the input and the output data are split into training set, validation set and test set.
The regularization term lamda is initially set to 0.01 . M is iterated over a range of values 10 to 20. For each value of M, the design matrix phi, the mean and spreads are calculated for training set and validation set. Starting from lambda value of 0.01 to <10, the weights are calculated using closed form solution. If the error calculated on validation is lesser than the previously recorded error, then the newly obtained values for M, designmatrix, lambda, and weights are considered optimal.

Pseudocode for trainClosedForm

```
def trainClosedForm(data,outputLabels):
        lambdaval = 0.01  /*inital regularization value */
        preverror = 0;

         for M in range(10,20):   /*M chosen from 10 to 20 */
            phi,mu,spread=getDesignMatrix(trainingData,M)  /*calculate the design matrix, mu, spread for training data */
            phiValidate,muval,spreadval=getDesignMatrix(validationData,M) /*calculate the design matrix,mu, spread for validation data */
           while(lambdaVal<10):
               w=  getWeights(lambdaval,M,phi,trainingOutput) /* weights calculated using moore penrose pseudo inverse */
               validationerror = sqrt(error/len(validationOutput)
               if(validation error < preverror)
                  weights=w
                  preverror=validationError
                  finalMu=mu                    /*optimal solution taken since validation error less */
                  finalSpread=spread
                  finalLambda=lambdaval
                  clusterSize=M
              lambdaval*=2     /*lambda value changed by a factor of 2 */
        return weights,finalLambda,clusterSize,finalMu,finalSpread
```

def getDesignMatrix(data,M):
This method is invoked from trainClosedForm solution to calculate the designmatrix each time for the different value of M. Divides the data into M clusters using kmeans from sklearn.cluster. Uses kmeans to calculate the centers . Obtains the M centroid points for the entire data set. These centroid points are then used to calculate the spreads of the data using another method getSpreads. Once the spreads are obtained, the design matrix phi is calculated .

Code for getDesignMatrix

```
def getDesignMatrix(data,M):
   kmeans = KMeans(n_clusters=M, random_state=0).fit(data) /*kmeans data into M clusters */
   centroids=kmeans.cluster_centers_  /*obtains the centroids for each cluster */
   spreads = getSpreads(data,M,kmeans.labels_) /*spreads are calculated using centroids on data */
  /*calculation of design matrix begins */
   phi=[]
   for i in range(0,M):
      temp=[]
      for j in range(1,len(kmeans.labels_)+1):
         t=matrix(data[j-1:j]-centroids[i])
         temp.append(exp((-
1*matmul(matmul(t,matrix(pinv(spreads[i]*identity(len(spreads[i])))))),t.transpose()).tolist()[0][0])/2)
)
      phi.append(temp)
   phi=matrix(phi).transpose()  /*phi the design matrix */
   return phi,centroids,spreads  /*returns the calculated design matrix phi */
```

def getSpreads(data,M,clusterIndexes):

This method is used to calculate the spreads of the data once the centroid points are obtained using kmeans. Basically derives the covariance matrix for a cluster of the data based on the cluster indices obtained from kmeans. The covariance matrix per cluster is added to the spreads[] array, there will be M covariance matrixes since there are M clusters . Returns the spreads to the calling method getDesignMatrix.

Code for getSpreads

```
def getSpreads(data,M,clusterIndexes):
   spreads = []  /*initialise the spreads array */
   temp=[]
   for i in range(0,M):
      temp=[]
      for j in range(1,len(clusterIndexes)+1): /*finds the data belonging to each cluster */
         if clusterIndexes[j-1]==i:
            temp.append(data[j-1:j])
      ds=vstack((temp))
      spreads.append(cov(ds,rowvar=False))  /*determines the covariance matrix for each cluster */
   return spreads
```

train_stochastic.py

Used for obtaining stochastic gradient descent solution for LETOR dataset.

Main methods are

def trainStochastic(data,outputLabels)

def getWeights(M,phi,phiValidate,clusterIndexes,trainingOutput,validationOutput)

def getDesignMatrix(data,M)

def getSpreads(data,M,clusterIndexes)

def trainStochastic(data,outputLabels):

Method is invoked from main.py for calculating the stochastic gradient descent solution for LETOR dataset. In a loop M values are varied from 10 to 20. The design matrix and the weights are then calculated for each value of M by calling method getDesignMatrix and getWeights.

def getWeights(M,phi,phiValidate,clusterIndexes,trainingOutput,validationOutput):
Main method in stochastic gradient descent calculation. Early stop algorithm is also employed in this. Patience val is fixed as 10, lamdaval is initialised to 0.1 and learning rate is initialised to 0.01. minibatch_size is initialised to 128.
In each epoch, for each mini batch, the error derivate E_D is calculated. The E_D value is then used to increment the weight value using the formula

E = (E_D + 0.1 * weights)
E = (E_D + lambdaval * weights.transpose())
weights = weights - learning_rate * E

After obtaining the weights, the weights are tested on the validation data and the validation error is calculated and recorded into variable prevError. This validation error is compared to a pre existing prevError value , and if it is found to be less, then the current weights , lambda, learningrate are considered to be optimal. Between minibatches, Lambdaval is varied between 0.1 and <10, and incremented by a factor of 2, learning rate is varied between 0.01 and < 5 and incremented by a factor of 2. If the patience limit is crossed, the early stop makes the algorithm stop, and the optimal weights, and the optimal hyperparameters lambda and learning rate found are reported back.

Pseudocode for getWeights

```
def getWeights(M,phi,phiValidate,clusterIndexes,trainingOutput,validationOutput):
    j=0
    preverror=float("inf")
    prevTrainingError=float("inf")
    patience=10  /*patience parameter fixed */
    lambdaval=0.1 /*regularisation factor initialised to 0.1 */
    learning_rate=0.01  /*learning rate initialised to 0.01 */
    weights=[]
    minibatch_size=128  /*mini batch size set as 128 */
    for epoch in range(patience):  /* each epoch */
        for i in range(0,int(len(clusterIndexes) / minibatch_size)):  /*each mini batch in epoch */
            lower_bound = i * minibatch_size
            upper_bound = min((i+1)*minibatch_size, len(clusterIndexes))
            Phi = phi[lower_bound: upper_bound]
            t=trainingOutput[lower_bound:upper_bound]
            if te==0:
                E_D = matmul((matmul(Phi,weights)-t).transpose(),Phi)
                te=1
            else:
                E_D = matmul((matmul(Phi,weights.transpose())-t).transpose(),Phi)
            E = (E_D + 0.1 * weights)
            E = (E_D + lambdaval * weights.transpose()) /*error derivate calculation */

            E=E[0]
            weights = weights - learning_rate * E /*incrementing weight based on learning rate ,error */

            weights=weights[0]
            error=0
            for row in range(0,len(validationOutput)):
                t=0
                for j in range(1,M):
```

```
                t+=weights.tolist()[0][j]*phiValidate.item(row,j)
            error+= (validationOutput[0][row+int(55698)]-t)**2
            error+=lambdaval*matmul(weights,weights.transpose())
        validationError=sqrt(error/len(validationOutput))  /*obtained validation error */
        if validationError<preverror:
                finalWeights=weights
                finalLambda=lambdaval    /*optimal weights and hyperparemeters so far */
                learningrate=learning_rate
                preverror=error
        else:
            j=j+1
        if lambdaval<10:
            lambdaval*=2  /*varying the regularization term */
        if learning_rate<5:
            learning_rate*=2  /*varying the learning rate */
        if j>=patience:  /*stopping in early stop */
            break
    return weights,finalLambda,learningrate
```

train_closedform_syn.py and train_stochastic_synthetic.py are same as above train_closedform_letor and train_stochastic. The only difference is the use of the synthetic data set in place of the letor dataset.

testLR.py

The main method in the class is test_LR. Once the weights and the design matrix are obtained by calling the train_closedform_letor.py and the train_closedform_syn.py for LETOR dataset and the synthetic dataset, the model is tested on the respective output data for LETOR and synthetic dataset. The root mean squared error value is found using the method getRootMeanSquareError method in the class. This error is a scalar value and is reported back to the main.py

Code for test_LR

```
def test_LR(weights,lambdaval,M,mu,spread,learningrate,data,n):
    testData=data
    size=len(data)
    phi=getDesignMatrix(testData,M,mu,spread)
    rms=getRootMeanSquareError(testData,M,phi,lambdaval,weights,len(testData),n)
    /* returns the scalar rms value */
    return rms
```

Code for getRootMeanSquareError

```
def getRootMeanSquareError(data,M,phi,lambdaval,weights,size,n):
    error=0
    for i in range(0,len(data)):
        t=0
        for j in range(1,M):
            t+=float(weights.tolist()[j][0])*phi.item(i,j)
        error+= (data[0][i+int(n)]-t)**2
        t=0
        for j in range(0,M):
            t+=weights[j]**2
        error+=lambdaval*t
```

```
    rms=sqrt(error/len(data)) /*actual error calculation */
    return rms
```

teststo.py similar to testLR.py, used to calculate the root mean square error value after training the model for LETOR and synthetic dataset for stochastic solutions. Applies the trained model on the test output and returns the error.

Observations:

UBitName = himasuja
personNumber = 50246828
('Testing error for Letor Dataset using closed form solution:', 2.088497020314503)
('Testing error for Letor Dataset using stochastic gradient descent:', 1.9105418456682792)
('Testing error for Synthetic Dataset using closed form solution:', 1.7524394515149715)
('Testing error for Synthetic Dataset using stochastic gradient descent:', 0.571340876511075)