Lab Course Machine Learning
Exercise Sheet 5
Krithika Murugesan(277537)

Preprocessing of dataset, the wine dataset has no categorical variables, hence one hot encoding is not required, while the categorical variables in Bank dataset are converted to numerical using One-hot encoding. There are no missing values in the dataset. Both the datasets are normalized using min-max normalization so that all variables have the same scale and does not scale up the process.

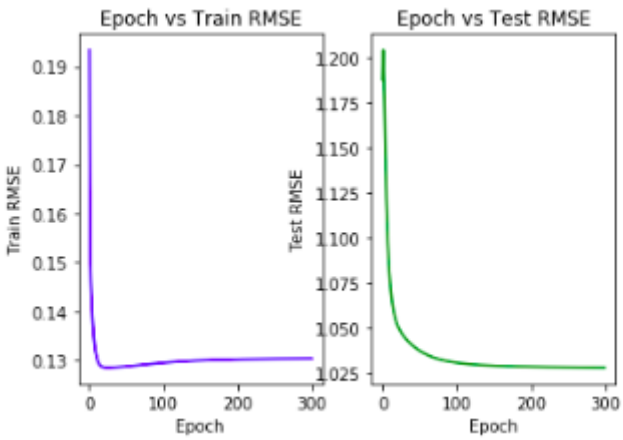
Exercise 1: Regularization

The three hyper parameters are learning rate, regularization constant and batch size. To implement the mini batch gradient descent first the train data is shuffled for each epoch and split into mini batches of size 50, later the parameter is updated using the gradient of this batches. The bias is not added to the x data as in ridge regression there may be some discrepancies in the regularization. The following is the code for the same. The given linear regression Bank dataset is used here.

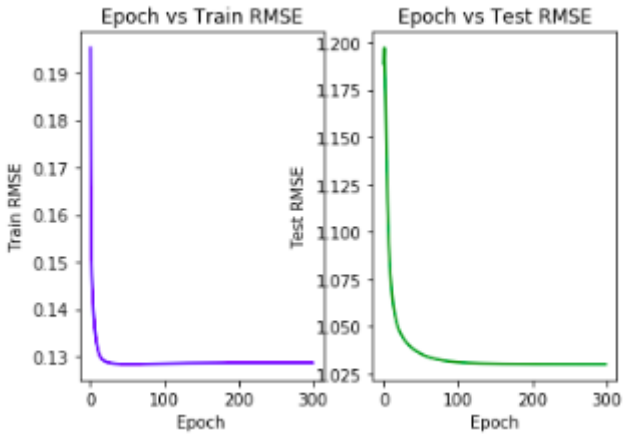
```
#Mini batch Gradient Descent
def mini_BGD(trainGd,xTest,yTest,alpha,lamda,batchsize,epoch):
    beta = np.zeros((trainGd.shape[1]-1,1))
    rmse,ii,rmseTest = [],[],[]
    for each in range(epoch):
        minibatch = shuffleAndSplit(trainGd,int(trainGd.shape[0]/batchsize))
        for every in minibatch:
            x,y = trainSplit(every)
            lossOld = loss(x,y,beta)
            betaNew = beta - alpha*gradient(x,y,beta,lamda)
            lossNew = loss(x,y,betaNew)
            #alpha = boldDriver(alpha,lossOld,lossNew)
            if float(lossOld-lossNew) < 0.00001:
                print("Optimal beta found : \n",betaNew)
                break
            if each == epoch :
                print("Did not converge")
            beta = betaNew
        ii.append(each)
        rmse.append(np.sqrt(lossNew/xTrain.shape[0]))
        temp = loss(xTest,yTest,beta)
        rmseTest.append(np.sqrt(temp/xTest.shape[0]))
    return (ii,rmse,rmseTest)
```

After each iteration of the epoch, RMSE of train data and RMSE of test data are appended with respective lists which are used to plot the graphs. The following are the graphs for different values of alpha and lambda,using grid search.

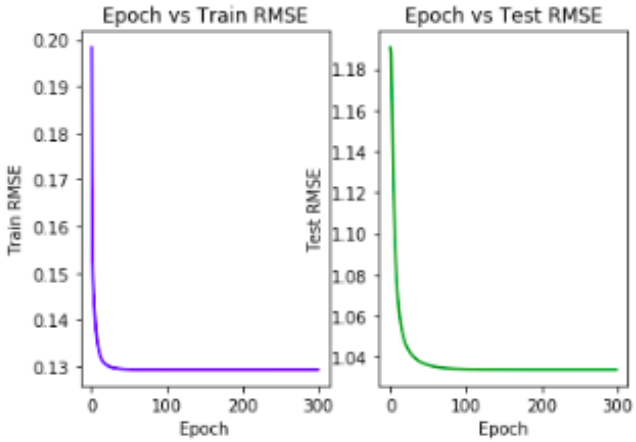
Learning rate = 0.001 and Regression co-efficient = 0.1



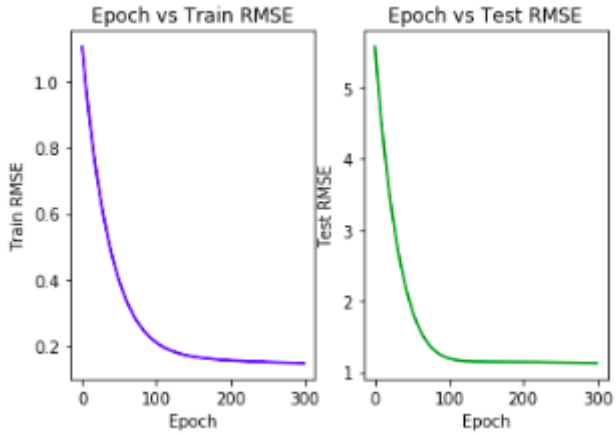
Learning rate = 0.001 and Regression co-efficient = 0.3



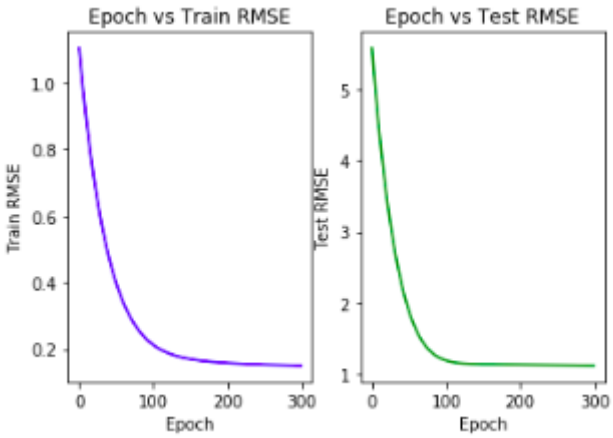
Learning rate = 0.001 and Regression co-efficient = 0.6



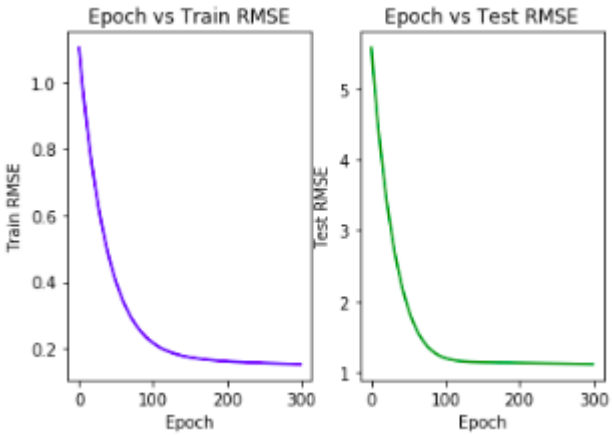
Learning rate = 0.00001 and Regression co-efficient = 0.1



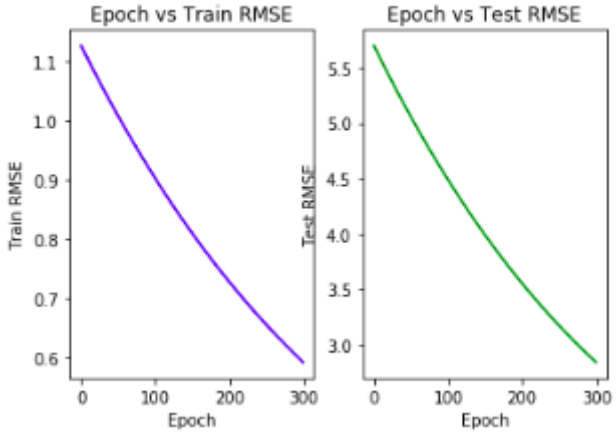
Learning rate = 0.00001 and Regression co-efficient = 0.3



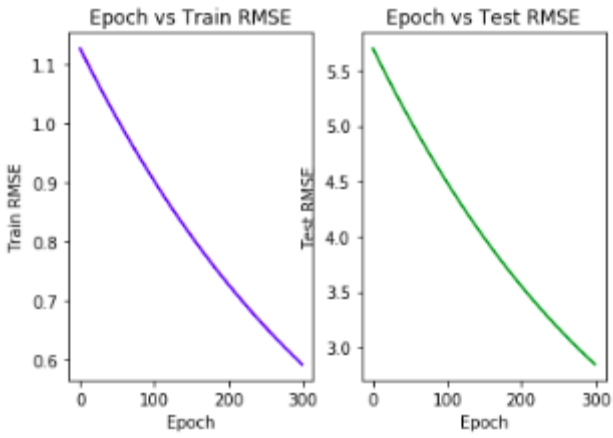
Learning rate = 0.00001 and Regression co-efficient = 0.6



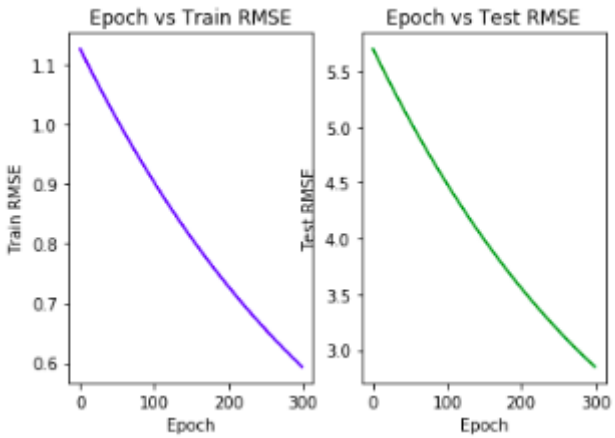
Learning rate = 0.000001 and Regression co-efficient = 0.1



Learning rate = 0.000001 and Regression co-efficient = 0.3



Learning rate = 0.000001 and Regression co-efficient = 0.6



It can be observed that by using smaller learning rate, the number of iterations required by the algorithm to converge are more. The train and test RMSE graphs are almost similar but there is a change in the error value relatively. It can also be seen higher values of the regression co-efficient enables faster convergence of the descent.

Exercise 2: Hyper-parameter tuning and Cross validation

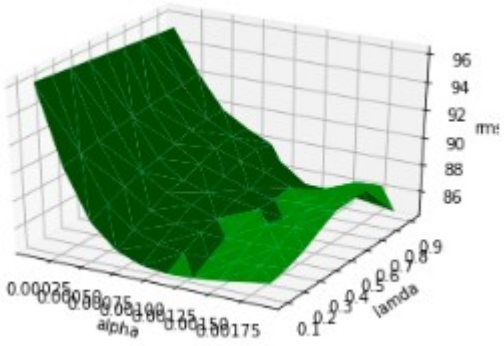
Cross validation has to be done in order to determine the hyper parameters. This is done by splitting the train data in k parts and learning the parameter for each combination of the parameters (grid search). The data used for validation is a part of the training data itself as the use of Test data will result in complete replication of the testing data and lead to over-fitting of data. To perform the 5 fold cross validation in this case, we split the data into 5 equal parts and treat each set at one point of time as the test data and remaining four parts as traind ata. This is implemented by the following snippet of code.

```
#returns the test and train data for each fold of cross validation
def cvTestTrain(trainA,testFold,totalFolds):
    trainCv = pd.DataFrame([])
    testCv = pd.DataFrame([])
    cols = trainA.columns.values
    trainA.columns = [''] * len(trainA.columns)
    batch = np.array_split(trainA,totalFolds)
    key = 1
    trainDict = {}
    for each in batch:
        trainDict[key] = each
        key = key + 1
    testCv = trainDict[testFold]
    testCv.columns = cols
    for key in range(1,6):
        if(key != testFold):
            trainCv = trainCv.append(trainDict[key])
    trainCv.columns = cols
    return(trainCv,testCv)
```

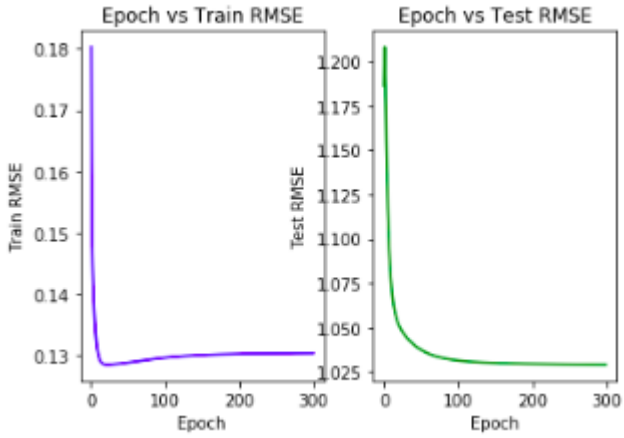
Later for each combination, the RMSE is estimated and the final hyper parameters are chosen based on the model with least errors. The code snippet for the same is as follows.

```
#CrossValidation
def cv(trainCvalid,alphaGrid,lamdaGrid,totalFolds,batchsize,epoch):
    alpha,lamda,rmseCv = [],[],[]
    for eachAlpha in alphaGrid:
        for eachLamda in lamdaGrid:
            #alpha.append(eachAlpha)
            alpha.append(eachAlpha)
            lamda.append(eachLamda)
            eachFoldRmse = []
            for each in range(1,totalFolds+1):
                print("alpha,lambda",eachAlpha,eachLamda)
                trainC = trainCvalid.copy()
                trainCv,testCv = cvTestTrain(trainC,each,totalFolds)
                xTest = testRed.loc[:,testCv.columns != 'quality']
                yTest = testRed.loc[:,testCv.columns == 'quality']
                #print(trainCv.shape)
                ii,rmse,rmseTest = mini_BGD(trainCv,xTest,yTest,eachAlpha,eachLamda,batchsize,epoch)
                eachFoldRmse.append(np.mean(rmseTest))
            temp = (np.mean(eachFoldRmse))
            print(float(temp))
            rmseCv.append(float(temp))
            print(len(rmse))
    #cvResult = pd.DataFrame({'alpha': alpha,'lambda': lamda,'rmse': rmse})
    return(alpha,lamda,rmseCv)
```

The 3D plot between the lambda, alpha and RMSE is as follows,



Once the optimal point with least RMSE is chosen, the optimal learning rate and regression co-efficient values are 0.0011 and 0.1 respectively. The performance of the Ridge regression for these parameters are:

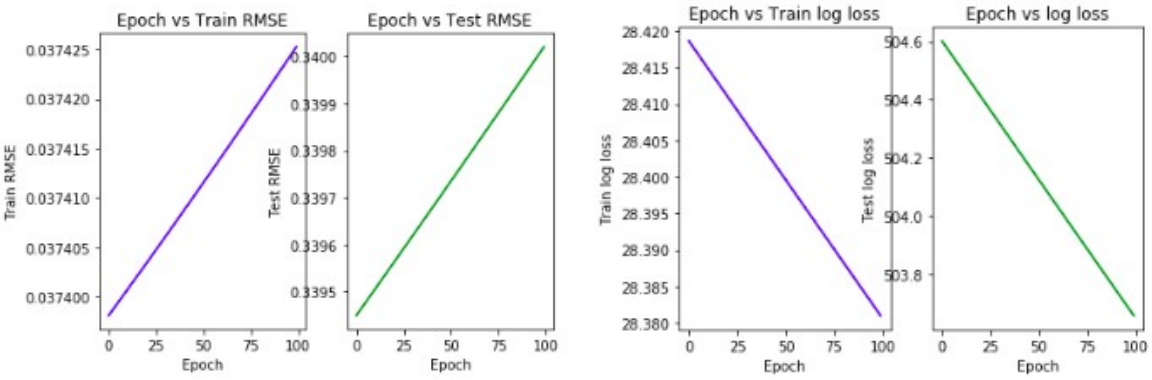


It can be seen, that for this model the convergence is faster!

Bonus: Implement Newton's Method

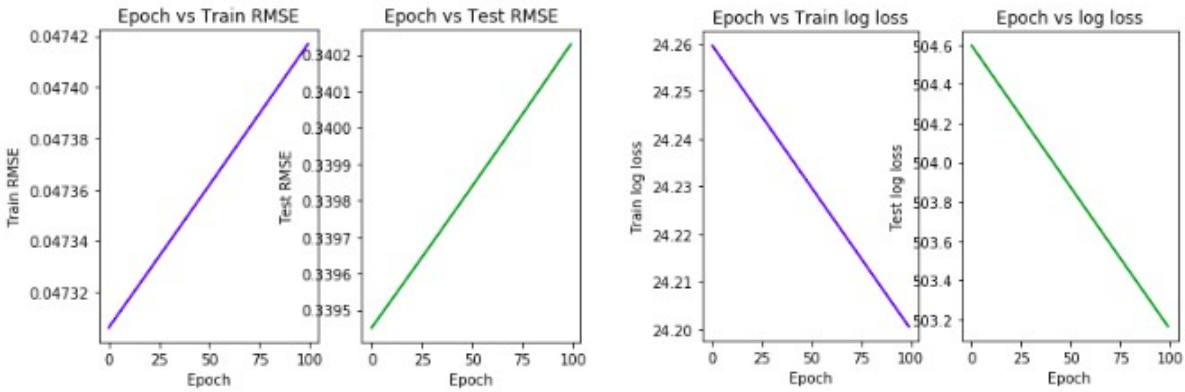
The newton's method is used for logistic regression, thus the Bank dataset is used. the logistic regression makes use of log-likelihood and loss function. Comparing the model by RMSE is not the preferred method. Here, the computation is based on log-likelihood, but the RMSE is also plotted. The following are the graphs for different with Alpha and Lambda.

Learning rate = 0.000001 and Regression co-efficient = 0.1

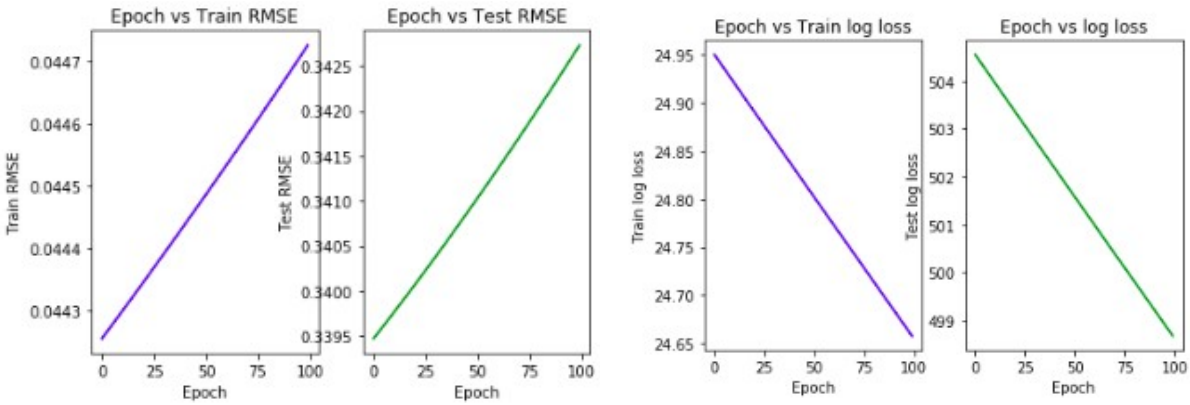


We can see that the log-likelihood and log loss are decreasing while RMSE is increasing as it is not an appropriate method to validate a model.

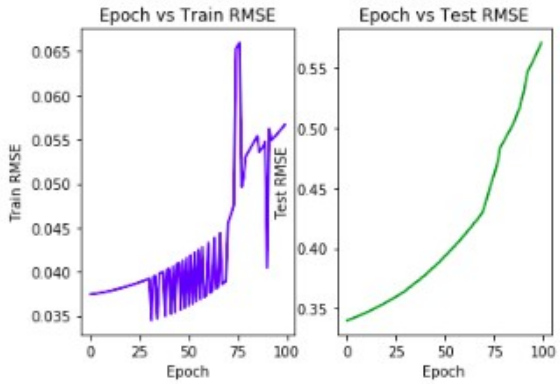
Learning rate = 0.000001 and Regression co-efficient = 0.01

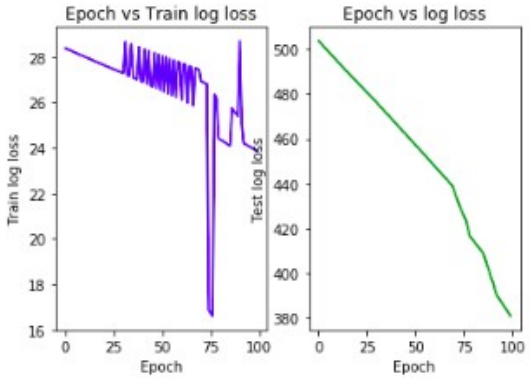


Learning rate = 0.000001 and Regression co-efficient = 0.001

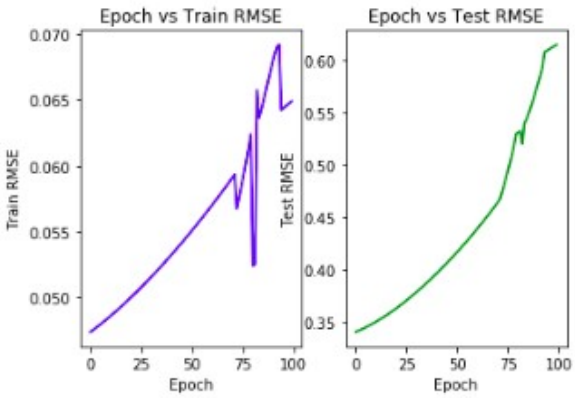


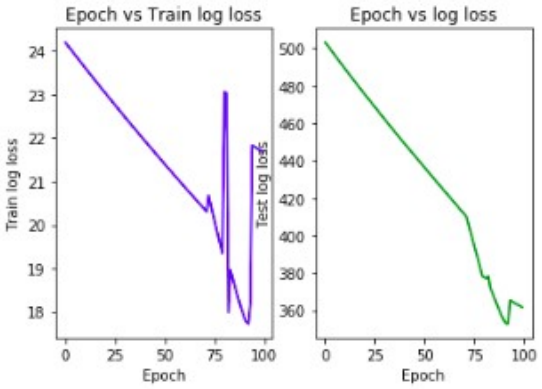
Learning rate = 0.0001 and Regression co-efficient = 0.1



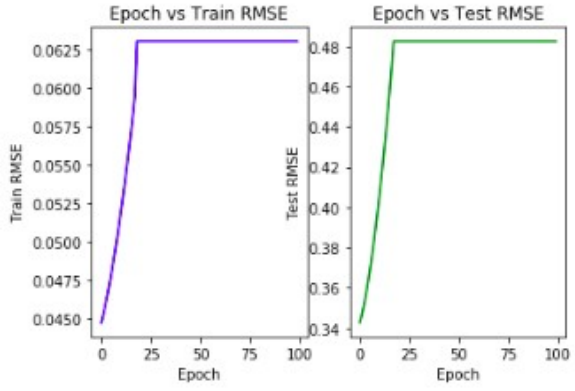


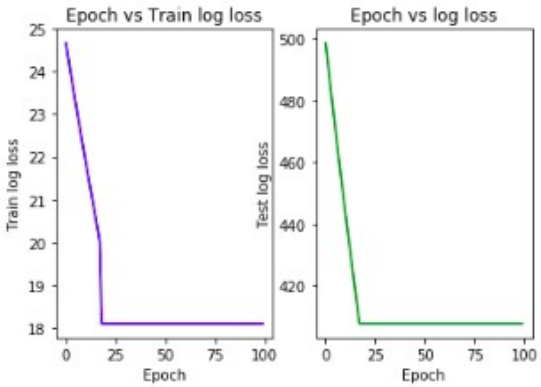
Learning rate = 0.0001 and Regression co-efficient = 0.01



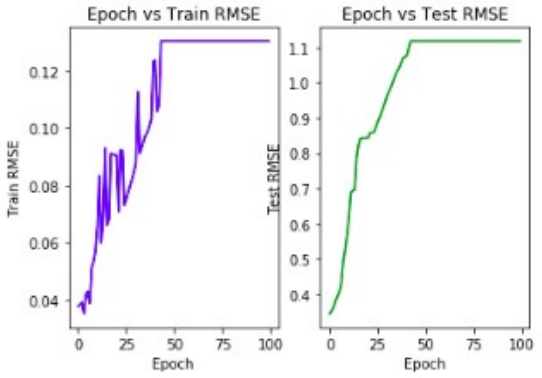


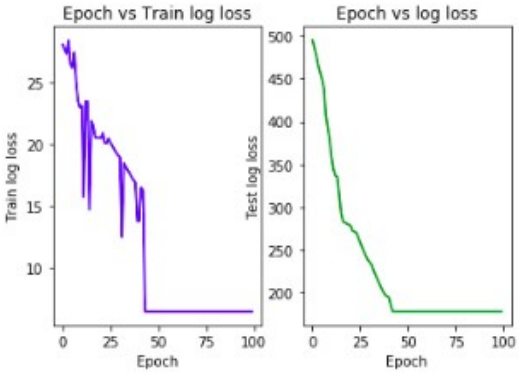
Learning rate = 0.0001 and Regression co-efficient = 0.001



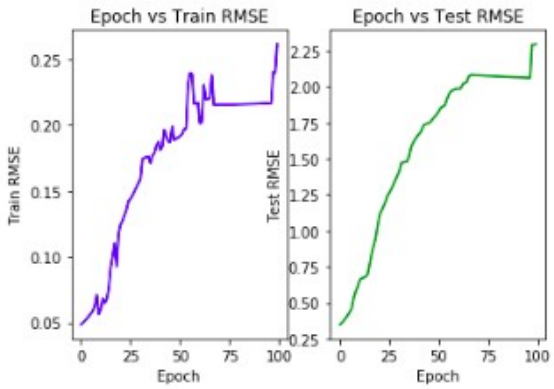


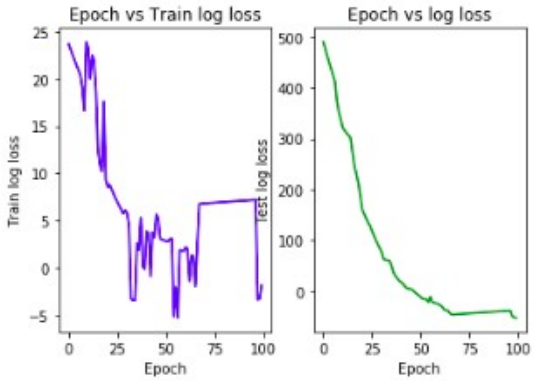
Learning rate = 0.001 and Regression co-efficient = 0.1





Learning rate = 0.001 and Regression co-efficient = 0.01





Learning rate = 0.001 and Regression co-efficient = 0.001