Report_Ex04

May 12, 2018

Lab Course: Distributed Data Analytics - Exercise Sheet 4
Krithika Murugesan - 277537
Parallel Stochastic Gradient Descent

Linear Regression has to be performed using MPI interface, here the root process or process with rank 0 is considered as the master; It distributes the work among the other workers. The two datasets provided are large and using a manual implementation sequentially will take a lot of time. Even in parallel processing only a part of the data was considered due to time and hardware constraints. A sample of 6000 rows were considered. Though the model is not a perfect one due to the limitaion in the usuage of data, the performance increase with number of prcesses can be seen; which is the main goal of this exercise. The steps are as follows

1. Data Reading and Prepossesing

```
In [ ]: def virus():
            path = "/home/kritz/Downloads/dataset/"
            label = []
            data = np.array([])
            #Listing all the files in directory
            for rootdir, subdirs, files in os.walk(path):
                if files:
                    for file in sorted(files):
                        #Reading the data into sparse matrix from libsvm
                        x,y = load_svmlight_file(rootdir+'/'+file)
                        #Adding empty rows to compensate for missing columns in these four fil
                        if file in ["2011-09.txt","2013-07.txt","2013-09.txt","2014-01.txt"]:
                            update = np.zeros((x.shape[0],1))
                            x = scipy.sparse.hstack((x,np.array((update))))
                        label.extend(y)
                        #Adding all files to same csr matrix
                        data = scipy.sparse.vstack((data,x))
            print(len(y),data.shape)
            data = data.tocsr()
            return(data[1:,].tocsr(),label)
```

This code snippet, is to read the virus data. Four files had one column less of data which has to be dealt with before merging them into a sparse matrix. As the data is present in libsvm format, load_svmlight_file is used to read the file directly in the form of sparse matrix thereby reducing the overhead of calculations in using a dense matrix and saving the memory used as well

```
In [ ]: def Kdd():
            path = "/home/kritz/Downloads/cup98LRN.txt"
            df = pd.read_csv(path,low_memory=False)

            df = df.iloc[1:10000,:]
            #Getting the numerical and categorical columns
            cols = df.columns
            numCol = df._get_numeric_data().columns
            catCol = list(set(cols) - set(numCol))
            #Type casting for float and string
            for each in numCol:
                df[each] = df[each].astype('float')
            for each in catCol:
                 df[each] = df[each].astype('category')

            #Filling mean values to Nans
            newDf = pd.get_dummies(df,drop_first = True)
            newDf = newDf.fillna(newDf.mean())

            #Reducing the dimension of predictor space based on variable importance
            X, y = newDf.loc[:,newDf.columns != 'TARGET_D'], newDf.loc[:,df.columns == 'TARGET_
            XNew = SelectKBest(chi2, k=200).fit_transform(X, y)

            return(scipy.sparse.csr_matrix(XNew),y)
```

The target variable here is considered as TARGET_D, this is a pretty huge dataset; Working with all the features was not possible in the system. So, I have used only the top 200 features by selecting the significant variables by a chi sqaure test. Also, it has categorical data which are encoded into one hot encoding, making them numerical without loosing the information they have. This is also converted to a sparse matrix as to save any modifications in the code later on

```
In [ ]: if rank == 0:
            updatedBeta,betaPart =  None,None
            data,y = virus()
            train,test = trainTestSplit(data[:6000,:],y[:6000])
            beta = np.zeros(((train.shape[1])-1,1))
            print("Shape of train at root ",train.shape)
            dataDistribute(train)

        else:
            partFile = comm.recv(source = 0,tag=1)
            beta = None
            test = None
            print("At rank ",rank, " size of data received is ",partFile.shape)

        comm.Barrier()
```

This part of the code is used to intially distribute the data among all the workers equally, each worker has a part of the x train data and a global copy of the beta is broadcast, after each epoch.

```
In [ ]: def dataDistribute(data):
            '''All the send functions in root process'''
            rows = data.shape[0]
            for i in range(1,size):
                startIndex = int((i-1)*(rows/(size-1)))
                endIndex = int(((rows/(size-1))*i))
                ##print("Index",startIndex,endIndex)
                #May cause index out of range due to index strting from 0, but we are counting
                if endIndex > rows:
                    endIndex = int(rows)
                comm.send(data[startIndex:endIndex,:],dest = i,tag =1)
```

Since our data is in CSR matrix, to be able to broadcast it to the workers the datDistribute function is used, it more or else distributes the data among all the workers

```
In [ ]: def trainTestSplit(data,y):
            #3. Split the data into test and train
            #Add bias
            print("Data raw ", data.shape)
            bias = np.ones((data.shape[0],1))
            dataBias = scipy.sparse.hstack([data,np.array(bias)])
            dataBias = dataBias.tocsr()
            print("After bias ",dataBias.shape,len(y))
            #Adding target
            y = np.reshape(y,(len(y),1))
            dataXY = scipy.sparse.hstack((dataBias,y)).tocsr()
            print("After y ",dataXY.shape)
            #Generate a random list of true and false and assign train and test based on those
            split = np.random.rand(dataXY.shape[0]) < 0.7
            #Assign train to true
            train= dataXY[split]
            #Assign test to false
            test = dataXY[~split]
            return(train,test)
```

The above given function is used to split the whole data into test and train, since sending xTrain and yTrain to the workers requires two communications, the bias and y is added to the whole dataset. Later they are seperated at the worker functions as and when needed, reducing the communication overhead involved

```
In [ ]: while flag == False :
            tic = time.time()
            #Ensuring all workers have updated betas
            beta = comm.bcast(beta,root = 0)
            test = comm.bcast(test,root = 0)

            comm.Barrier()

            #Workers compute local beta updates
```

```
if rank != 0:
    betaPart = SGD(partFile,beta,0.00000000001)

comm.Barrier()
#Sending to worker for global update
updatedBeta = comm.gather(betaPart,0)
comm.Barrier()

if rank == 0:
    #Calculating loss and appending results
    lossOld = leastSquareLoss(train,beta)
    updatedBeta = np.sum(np.array(updatedBeta)[1:,])/(size-1)
    lossNew = leastSquareLoss(train,updatedBeta)
    if lossOld - lossNew <= 0.0000000000000001:
        print("Algorithm converged")
        flag = True
    if iteration > 100:
        print("Algorithm did not converge in 100 iterations")
        flag = True
    iteration = iteration+1
    temp = leastSquareLoss(test,beta)
    testRmse.append(np.sqrt(temp/test.shape[0]))
    trainRmse.append(np.sqrt(lossOld/train.shape[0]))
    trainloss.append(lossOld)
    testloss.append(temp)
    times.append(time.time()-tic)
    beta = updatedBeta

#Updating all workers with new betas
comm.Barrier()
root = comm.bcast(flag,root = 0)
comm.Barrier()
print("Iteration ",iteration)

#Writing results to a csv file
if flag==True:
    loss = pd.DataFrame({'trainloss': trainloss,'testloss': testloss,
                         'trainRmse':trainRmse,'testRmse': testRmse,'time':times})
    loss.to_csv('ResidualsVirus1.csv', sep='\t',index = False)
    #Forcefully exiting since break did not kill all the processes
    comm.Abort()
```

This is the main section of the code, where the epochs are done, this is seperated from the first call of workers to avoid transfer of same data in every epoch. The beta is initialized to a zero matrix of the needed dimensions and is bcast to all the workers. In each worker for every element in the partfile gradient is calculated, which can be used to update the beta. Once the full iteration is over, the updated betas are gathered and ther mean is computed at the root process.New beta is bcast to all workers and the loop is executed till convergence of the algorithm

The newloss is checked with old loss against epsilon value, if this criteia is met the learning is done. Considering, the practicality of this. Only 100 epochs are computed.The algorithm does not convrege within this given epoch.Also, at every epoch the loss in calculated and appended to a dataframe which is written to a csv. To better present the graphs of all processes. Break stopped any computation and wrote the output file after 100 epochs but the processes did not terminate. So comm.Abort was used to do the same.

```
In [ ]: #Loss function
        def leastSquareLoss(train,beta):
            yPredicted = train[:,:-1].dot(beta)
            leastSquareLoss = (np.square(train[:,-1]-yPredicted)).sum()
            return(leastSquareLoss)
```

The squared loss is used as the cost function, the loss is used to check for convergence

```
In [ ]: #Function for derivative
        def derivative(train,beta):
            x,y = train[:,:-1],train[:,-1]
            predicted = y - (x.dot(beta))
            #print((-2 * x.T.dot(predicted)).shape)
            return (-2 * x.T.dot(predicted))
```

The derivative function computes the derivative for the loss function in each iteration

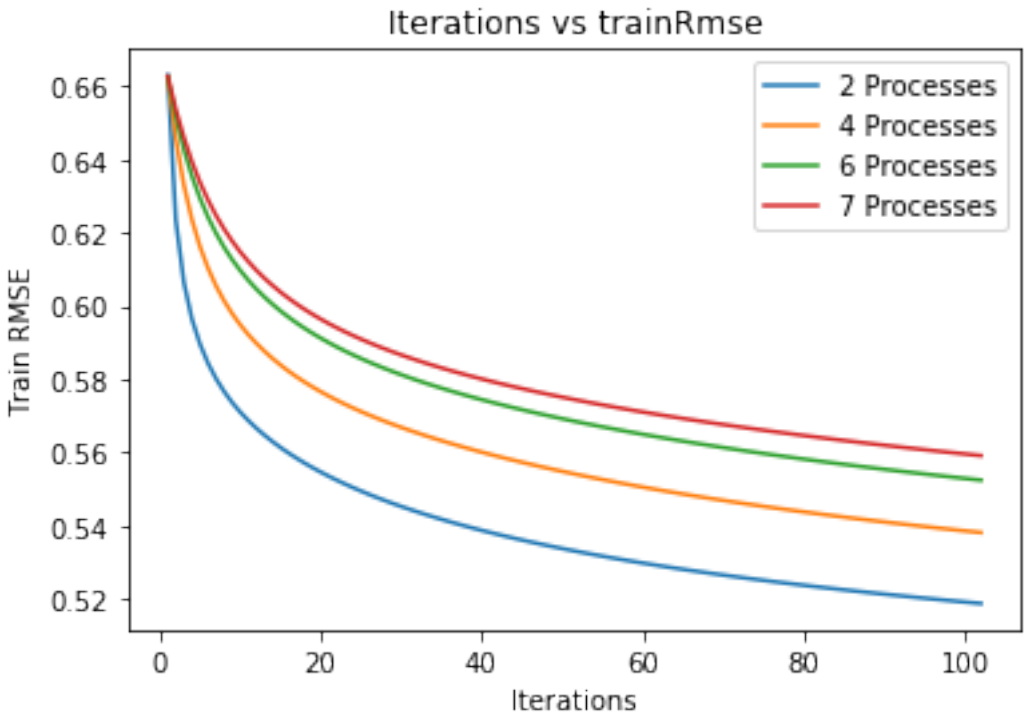
```
In [ ]: #Stochastic gradient calculation
        def SGD(train,beta,alpha):
            #Shuffling csr
            index = np.arange(np.shape(train)[0])
            np.random.shuffle(index)
            train = train[index,:]
            #Looping through each row in part file
            for each in train:
                #Computing next beta
                betaNext = beta - (alpha*derivative(each,beta))
                beta = betaNext
            return(beta)
```

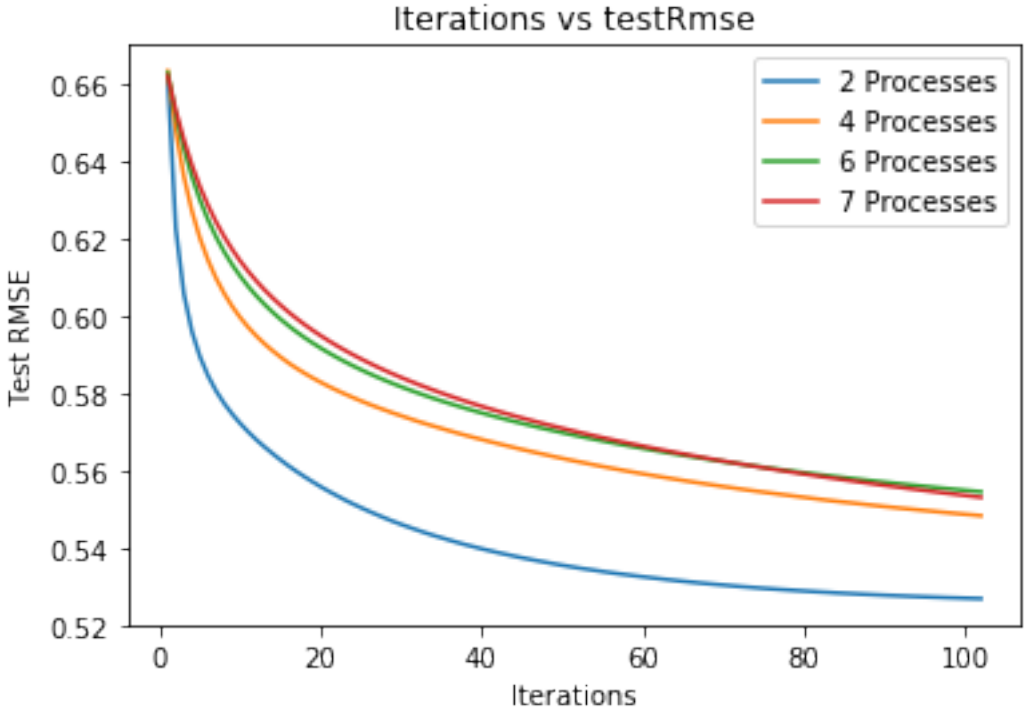
In SGD the data is shuffled at first and gone through in the order to ensure randomness of selection od data point at each iteration. This is used by all the workers

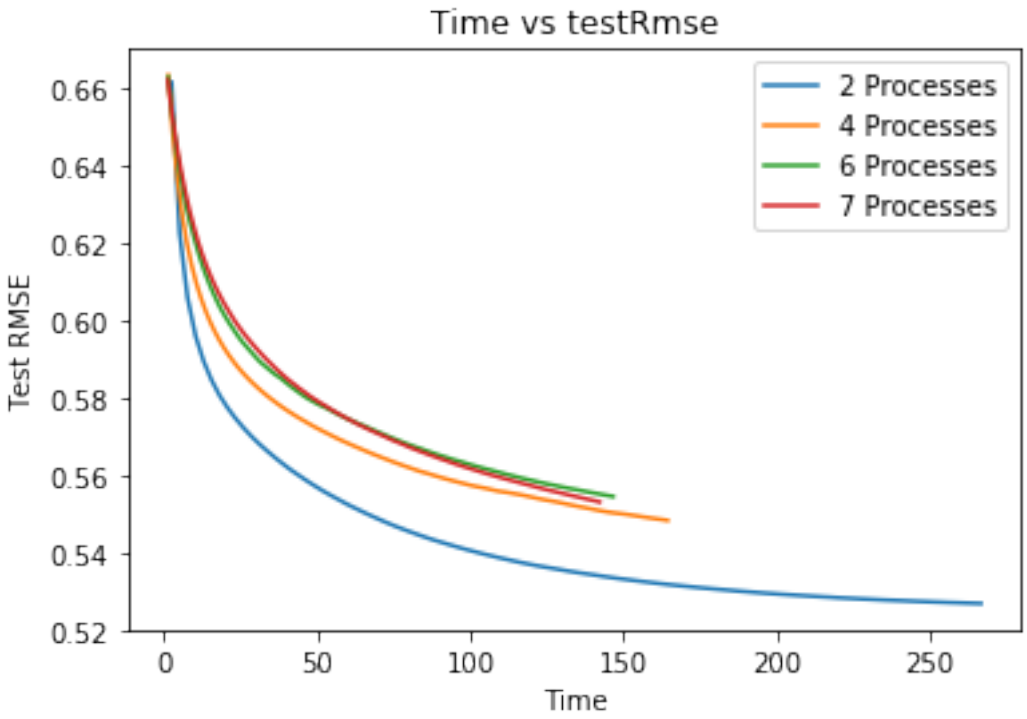
The computation bottleneck here, is every epoch in itself, which has lots of computations like gradient and beta updation. Using multiple processes should reduce this problem. The results obtained from 2,4,6 and 7 workers are as follows for Virus dataset

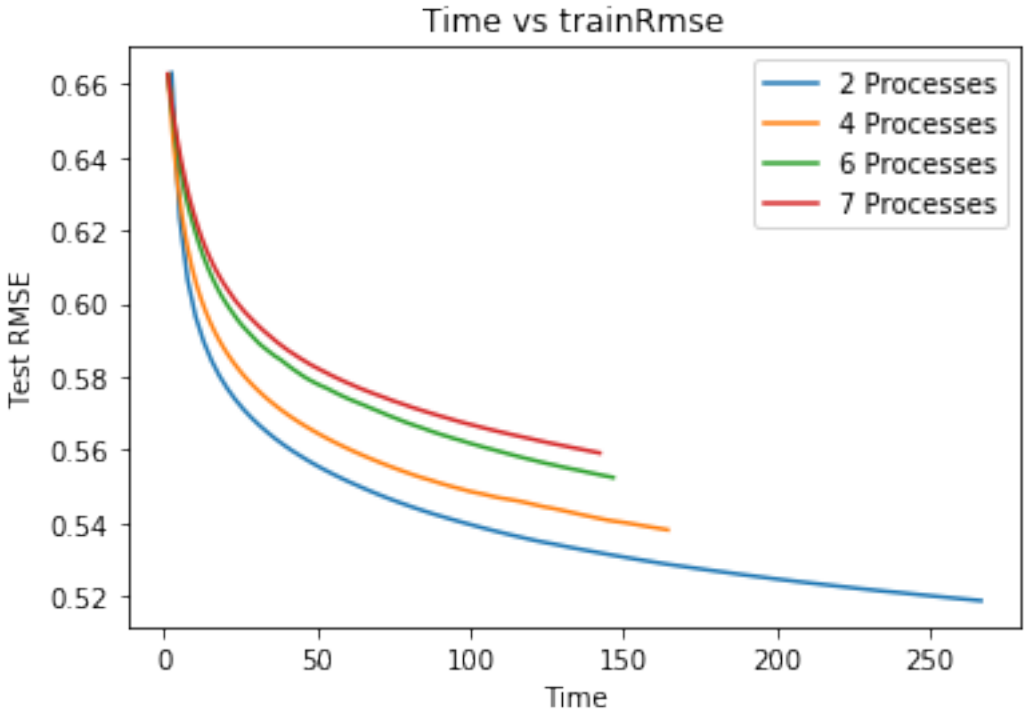
```
In [16]: #Plotting efficiency graphs
         import pandas as pd
         import matplotlib.pyplot as plt
         df2 = pd.read_csv('ResidualsVirus2.csv',sep = '\t')
         df4 = pd.read_csv('ResidualsVirus4.csv',sep = '\t')
         df6 = pd.read_csv('ResidualsVirus6.csv',sep = '\t')
```

```
df7 = pd.read_csv('ResidualsVirus7.csv',sep = '\t')
#Iterations
cnt = [i for i in range(1,103)]
time = df2['time'].cumsum()

#Iterations vs trainRmse
plt.plot(cnt,df2['trainRmse'],label = "2 Processes")
plt.plot(cnt,df4['trainRmse'],label = "4 Processes")
plt.plot(cnt,df6['trainRmse'],label = "6 Processes")
plt.plot(cnt,df7['trainRmse'],label = "7 Processes")
plt.xlabel("Iterations")
plt.ylabel("Train RMSE")
plt.title("Iterations vs trainRmse")
plt.legend()
plt.show()

#Iterations vs testRmse
plt.plot(cnt,df2['testRmse'],label = "2 Processes")
plt.plot(cnt,df4['testRmse'],label = "4 Processes")
plt.plot(cnt,df6['testRmse'],label = "6 Processes")
plt.plot(cnt,df7['testRmse'],label = "7 Processes")
plt.xlabel("Iterations")
plt.ylabel("Test RMSE")
plt.title("Iterations vs testRmse")
plt.legend()
plt.show()

#Time vs testRmse
plt.plot(df2['time'].cumsum(),df2['testRmse'],label = "2 Processes")
plt.plot(df4['time'].cumsum(),df4['testRmse'],label = "4 Processes")
plt.plot(df6['time'].cumsum(),df6['testRmse'],label = "6 Processes")
plt.plot(df7['time'].cumsum(),df7['testRmse'],label = "7 Processes")
plt.xlabel("Time")
plt.ylabel("Test RMSE")
plt.title("Time vs testRmse")
plt.legend()
plt.show()

#Time vs trainRmse
plt.plot(df2['time'].cumsum(),df2['trainRmse'],label = "2 Processes")
plt.plot(df4['time'].cumsum(),df4['trainRmse'],label = "4 Processes")
plt.plot(df6['time'].cumsum(),df6['trainRmse'],label = "6 Processes")
plt.plot(df7['time'].cumsum(),df7['trainRmse'],label = "7 Processes")
plt.xlabel("Time")
plt.ylabel("Test RMSE")
plt.title("Time vs trainRmse")
plt.legend()
plt.show()
```


Iterations vs trainRmse
Train RMSE
Iterations
2 Processes
4 Processes
6 Processes
7 Processes
0.66
0.64
0.62
0.60
0.58
0.56
0.54
0.52
0
20
40
60
80
100


Iterations vs testRmse
Test RMSE
Iterations
2 Processes
4 Processes
6 Processes
7 Processes
0.66
0.64
0.62
0.60
0.58
0.56
0.54
0.52
0
20
40
60
80
100


Time vs testRmse
2 Processes
4 Processes
6 Processes
7 Processes
Test RMSE
0.66
0.64
0.62
0.60
0.58
0.56
0.54
0.52
0
50
100
150
200
250
Time

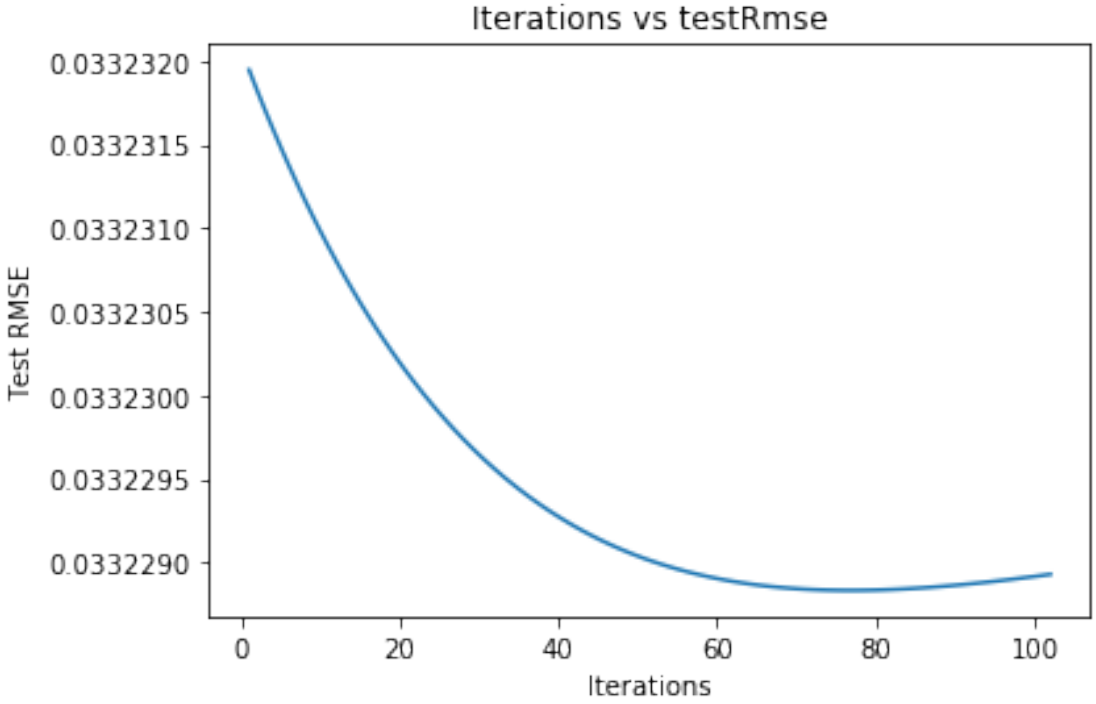

Time vs trainRmse
2 Processes
4 Processes
6 Processes
7 Processes
Test RMSE
0.66
0.64
0.62
0.60
0.58
0.56
0.54
0.52
0
50
100
150
200
250
Time

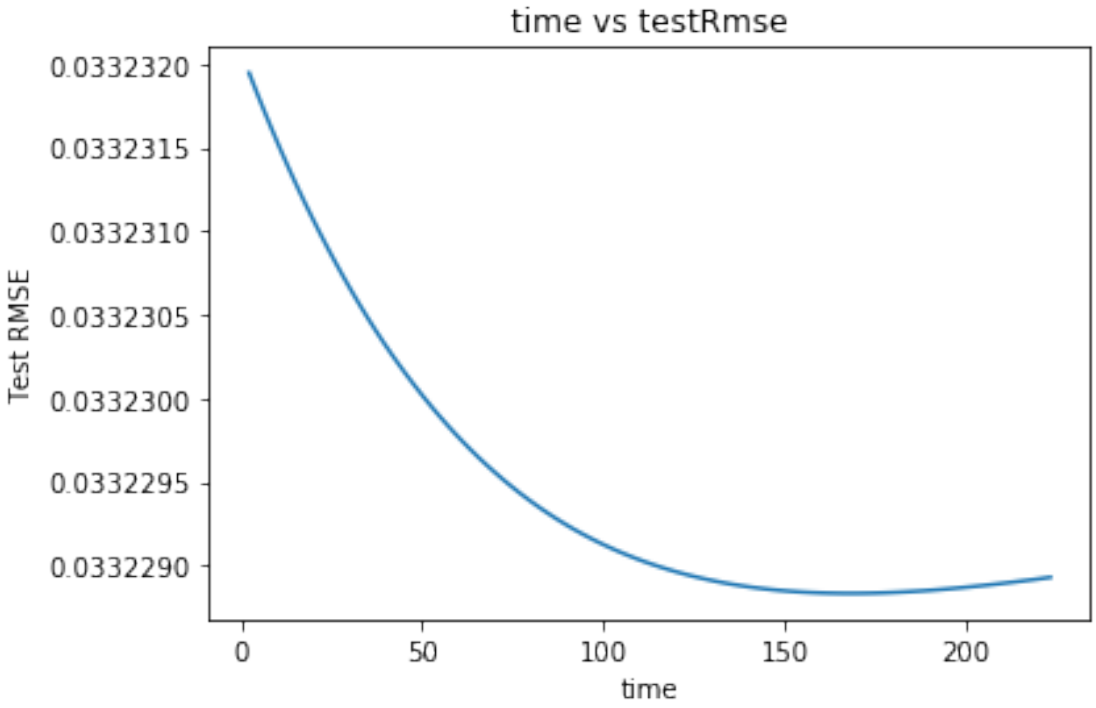
It can be seen that for more number of process the convergence is faster, also the time takes for the dip in loss is also considerably less. Giving more epochs a clear picture of the same could have been established but due to time and hardware limitations it could not be done. Nevertheless, the graphs represent the working of parallel SGD. The same number of iterations take lesser time for more processes

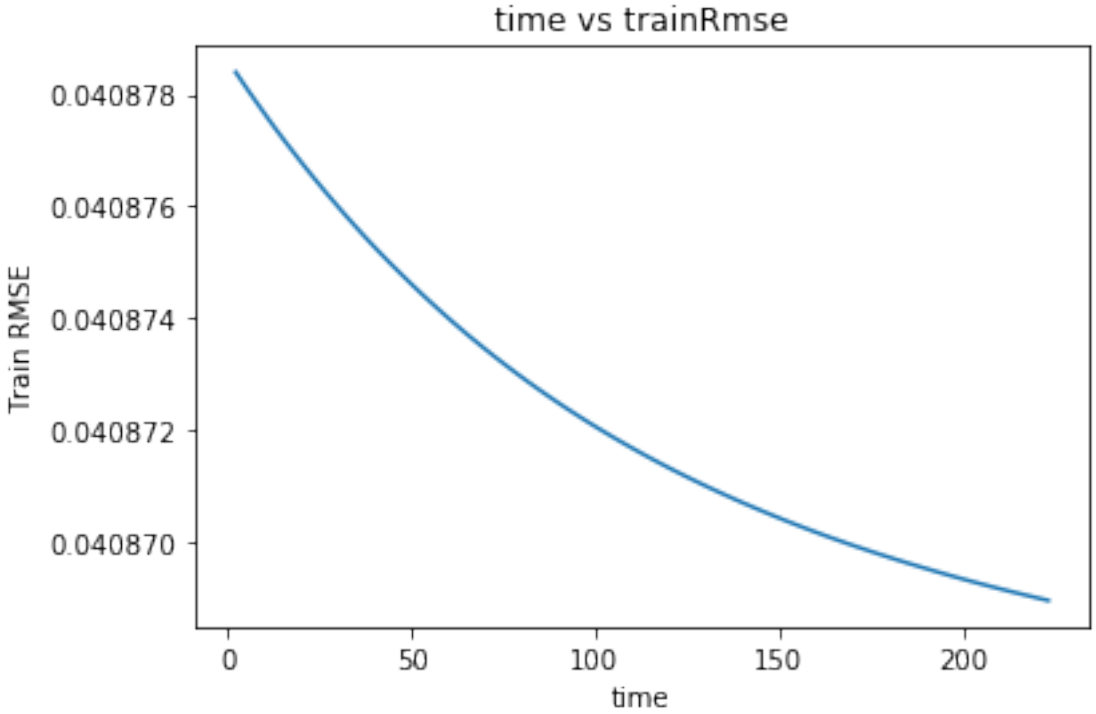
```
In [15]: #Plotting efficiency graphs for Kdd dataset
         import pandas as pd
         import matplotlib.pyplot as plt
         df2 = pd.read_csv('ResidualsKdd2Actual.csv',sep = '\t')
         df4 = pd.read_csv('ResidualsKdd2.csv',sep = '\t')
         df6 = pd.read_csv('ResidualsKdd6.csv',sep = '\t')
         df7 = pd.read_csv('ResidualsKdd7.csv',sep = '\t')
         #Iterations
         cnt = [i for i in range(1,103)]

         print("2 Processes")
         #Iterations vs trainRmse
         plt.plot(cnt,df2['trainRmse'])
         plt.xlabel("Iterations")
         plt.ylabel("Train RMSE")
         plt.title("Iterations vs trainRmse")
         plt.show()

         plt.plot(cnt,df2['testRmse'])
         plt.xlabel("Iterations")
         plt.ylabel("Test RMSE")
         plt.title("Iterations vs testRmse")
         plt.show()

         time = df2['time'].cumsum()
         plt.plot(time,df2['testRmse'])
         plt.xlabel("time")
         plt.ylabel("Test RMSE")
         plt.title("time vs testRmse")
         plt.show()

         plt.plot(time,df2['trainRmse'])
         plt.xlabel("time")
         plt.ylabel("Train RMSE")
         plt.title("time vs trainRmse")
         plt.show()
```

```
2 Processes
```


Iterations vs trainRmse
Train RMSE
0.040878
0.040876
0.040874
0.040872
0.040870
0
20
40
60
80
100
Iterations

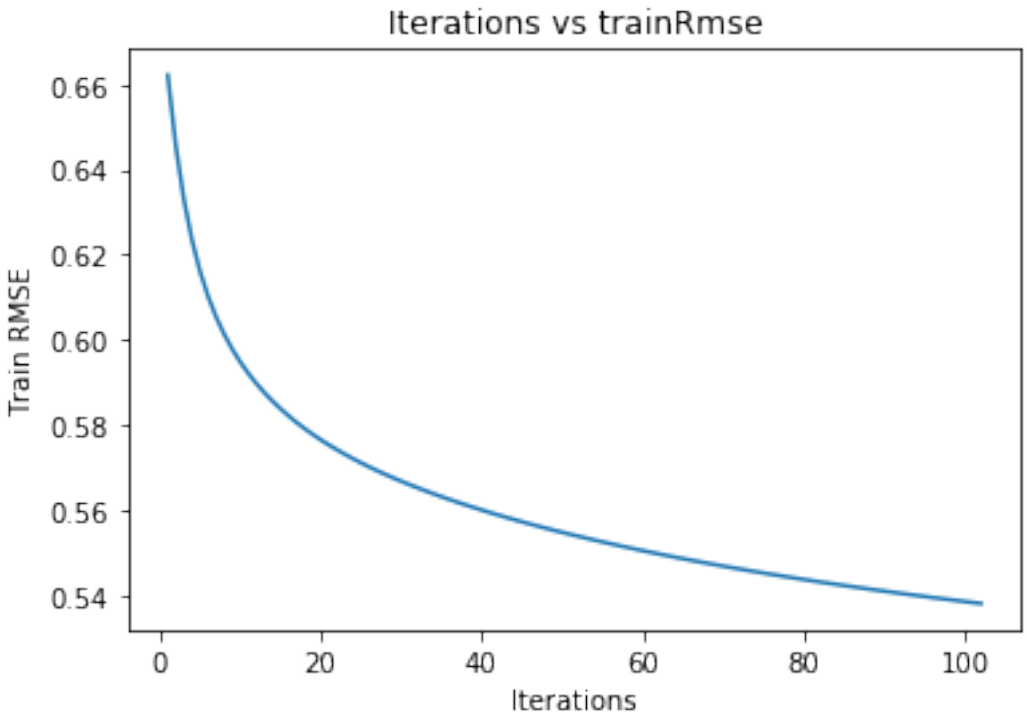

Iterations vs testRmse
Test RMSE
0.0332320
0.0332315
0.0332310
0.0332305
0.0332300
0.0332295
0.0332290
0
20
40
60
80
100
Iterations

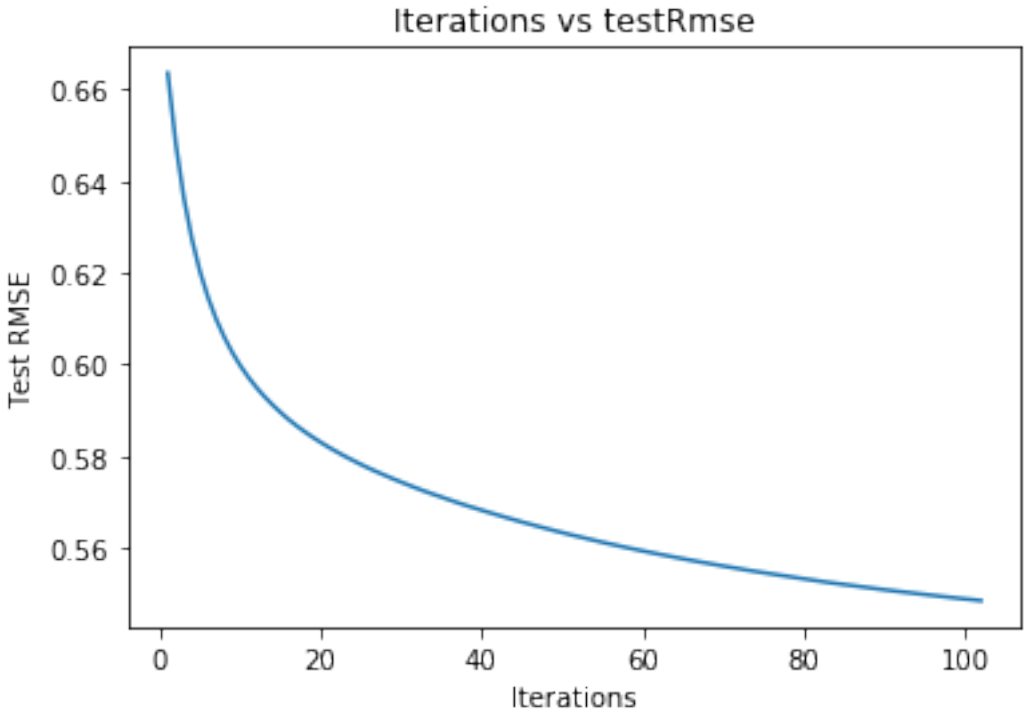

time vs testRmse
Test RMSE
0.0332320
0.0332315
0.0332310
0.0332305
0.0332300
0.0332295
0.0332290
0
50
100
150
200
time

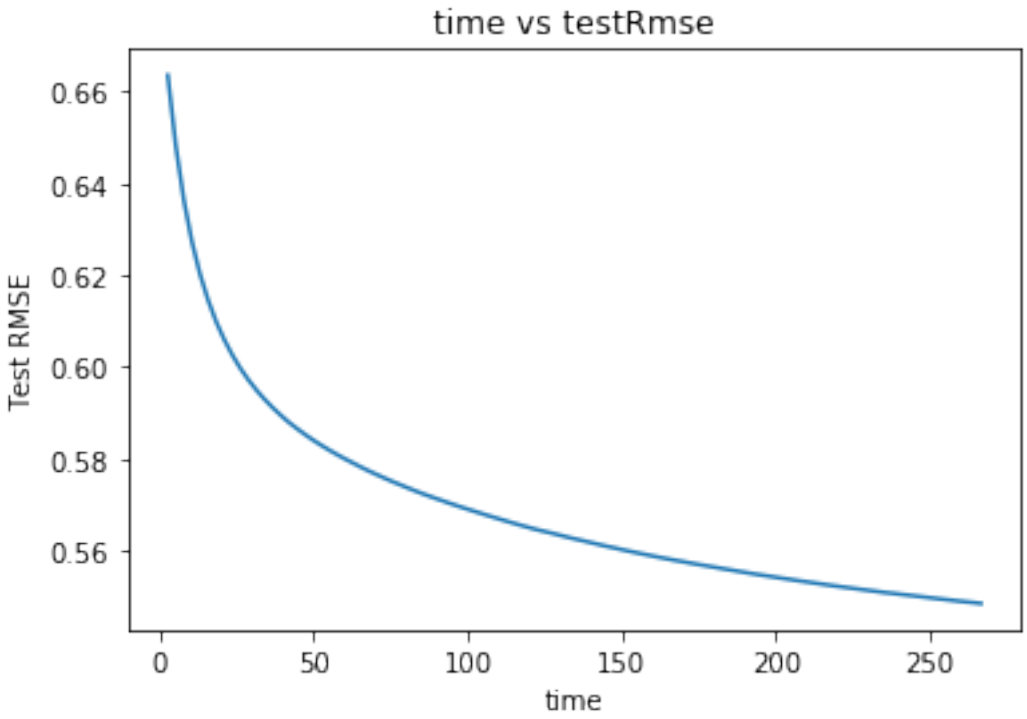

time vs trainRmse
Train RMSE
0.040878
0.040876
0.040874
0.040872
0.040870
0
50
100
150
200
time

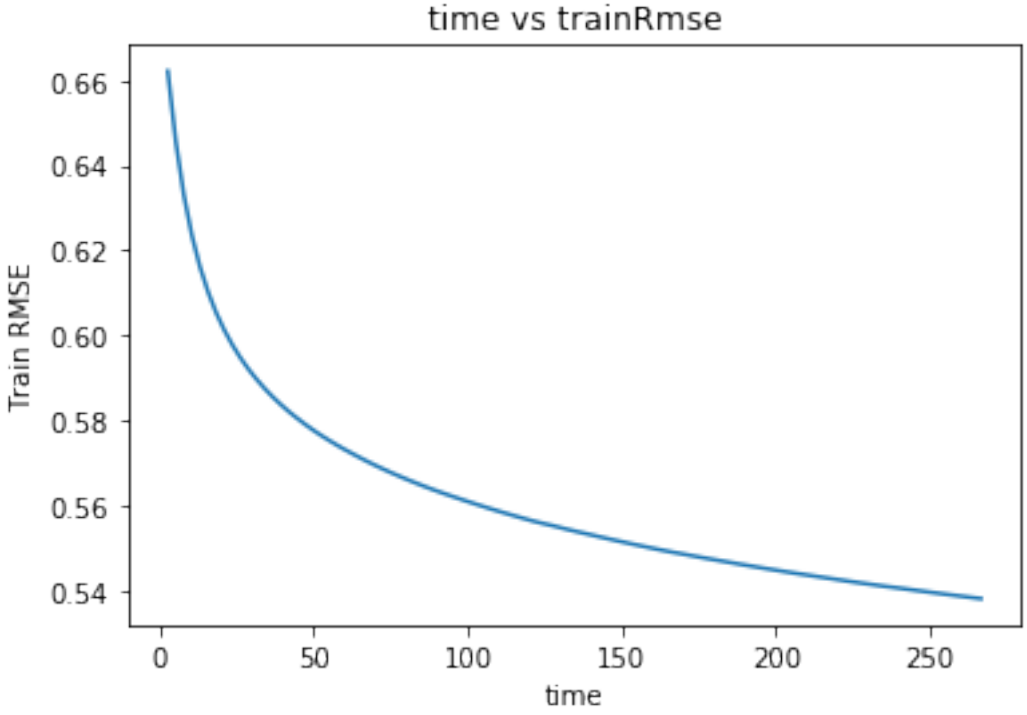
In [14]:
```
#Plotting efficiency graphs for Kdd dataset
#Iterations
cnt = [i for i in range(1,103)]

print("4 Processes")
#Iterations vs trainRmse
plt.plot(cnt,df4['trainRmse'])
plt.xlabel("Iterations")
plt.ylabel("Train RMSE")
plt.title("Iterations vs trainRmse")
plt.show()

plt.plot(cnt,df4['testRmse'])
plt.xlabel("Iterations")
plt.ylabel("Test RMSE")
plt.title("Iterations vs testRmse")
plt.show()

time = df2['time'].cumsum()
plt.plot(time,df4['testRmse'])
plt.xlabel("time")
plt.ylabel("Test RMSE")
plt.title("time vs testRmse")
plt.show()

plt.plot(time,df4['trainRmse'])
plt.xlabel("time")
plt.ylabel("Train RMSE")
plt.title("time vs trainRmse")
plt.show()
```

```
4 Processes
```


Iterations vs trainRmse
Train RMSE
0.66
0.64
0.62
0.60
0.58
0.56
0.54
0
20
40
60
80
100
Iterations

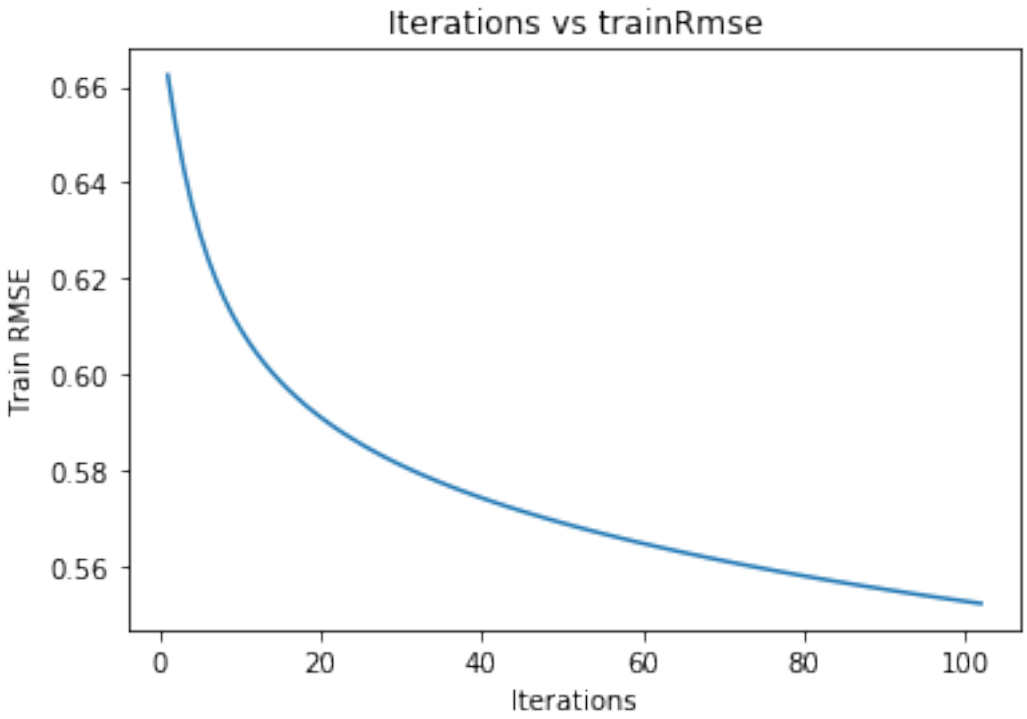

Iterations vs testRmse
Test RMSE
0.66
0.64
0.62
0.60
0.58
0.56
0
20
40
60
80
100
Iterations

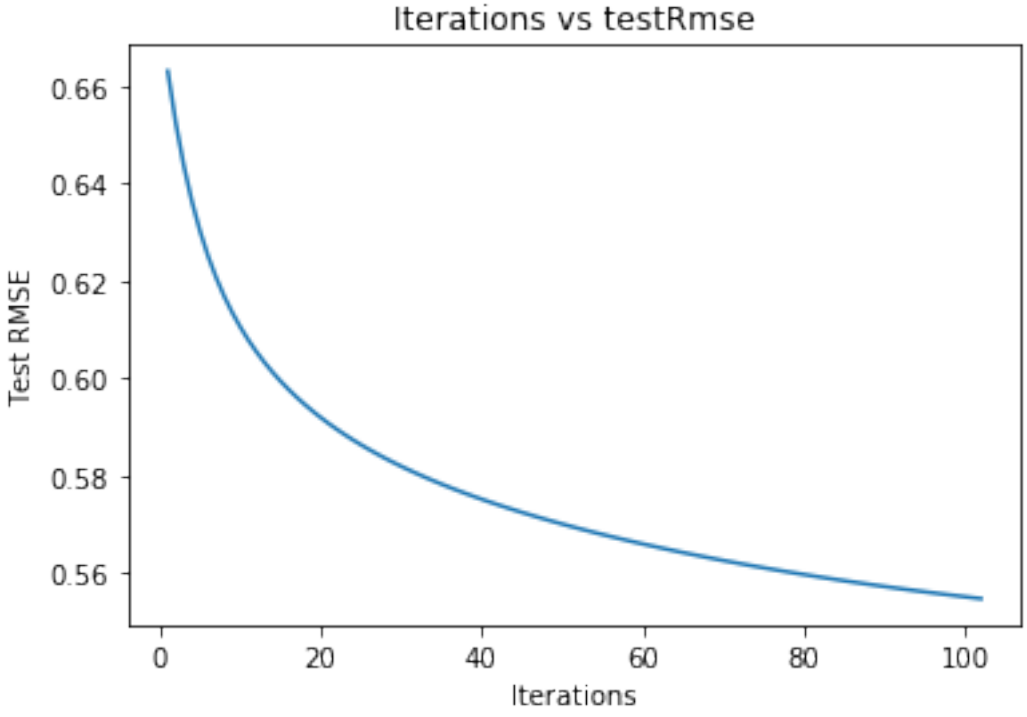

time vs testRmse
Test RMSE
0.66
0.64
0.62
0.60
0.58
0.56
0
50
100
150
200
250
time

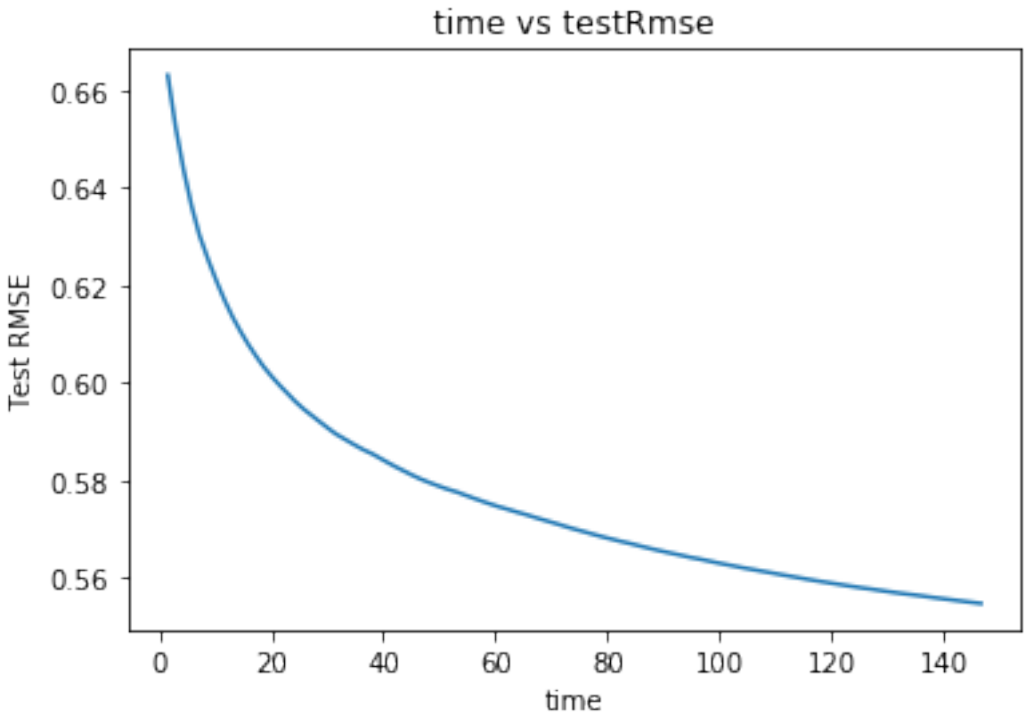

time vs trainRmse
Train RMSE
0.66
0.64
0.62
0.60
0.58
0.56
0.54
0
50
100
150
200
250
time

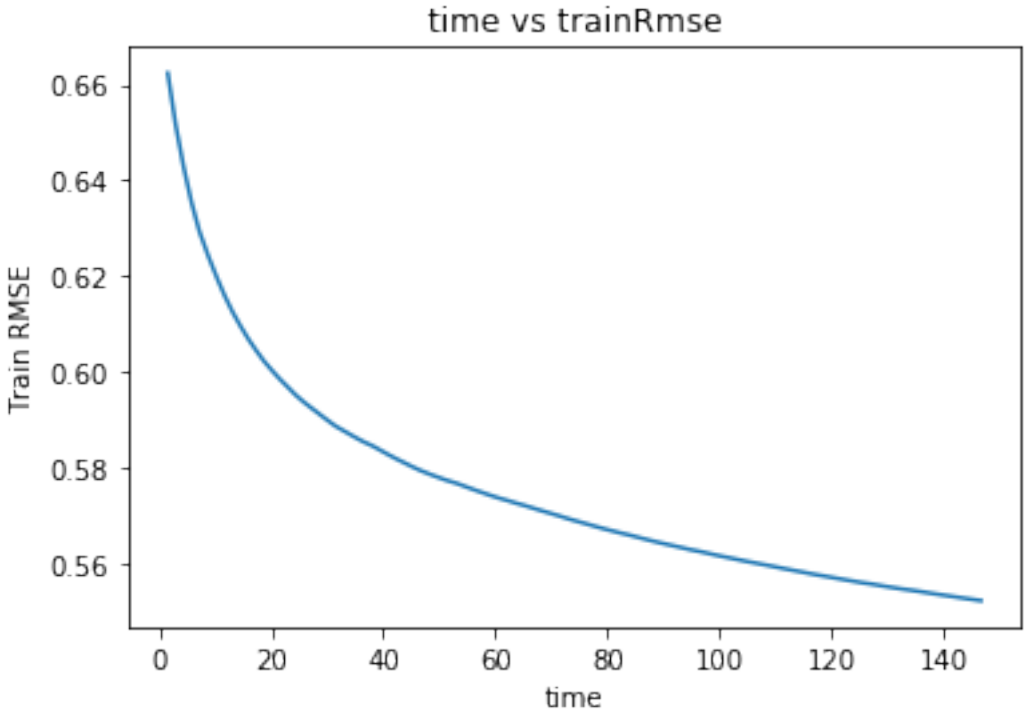
In [13]:
```
#Plotting efficiency graphs for Kdd dataset
#Iterations
cnt = [i for i in range(1,103)]

print("6 Processes")
#Iterations vs trainRmse
plt.plot(cnt,df6['trainRmse'])
plt.xlabel("Iterations")
plt.ylabel("Train RMSE")
plt.title("Iterations vs trainRmse")
plt.show()

plt.plot(cnt,df6['testRmse'])
plt.xlabel("Iterations")
plt.ylabel("Test RMSE")
plt.title("Iterations vs testRmse")
plt.show()

time = df6['time'].cumsum()
plt.plot(time,df6['testRmse'])
plt.xlabel("time")
plt.ylabel("Test RMSE")
plt.title("time vs testRmse")
plt.show()

plt.plot(time,df6['trainRmse'])
plt.xlabel("time")
plt.ylabel("Train RMSE")
plt.title("time vs trainRmse")
plt.show()
```

```
6 Processes
```


Iterations vs trainRmse
Train RMSE
0.66
0.64
0.62
0.60
0.58
0.56
0
20
40
60
80
100
Iterations

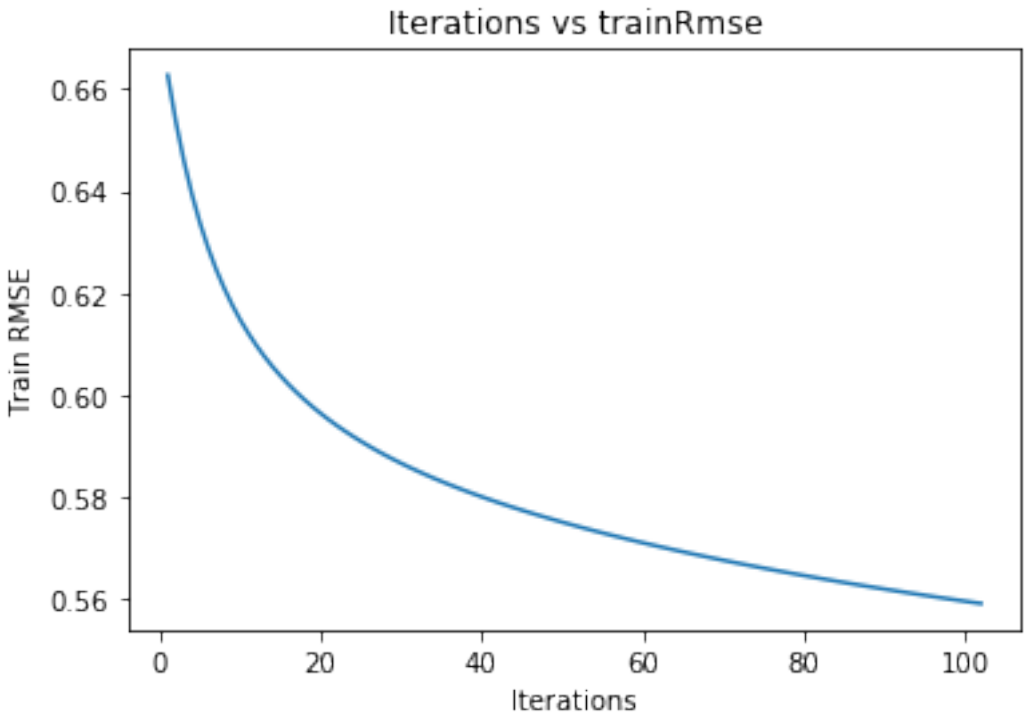

Iterations vs testRmse
Test RMSE
0.66
0.64
0.62
0.60
0.58
0.56
0
20
40
60
80
100
Iterations

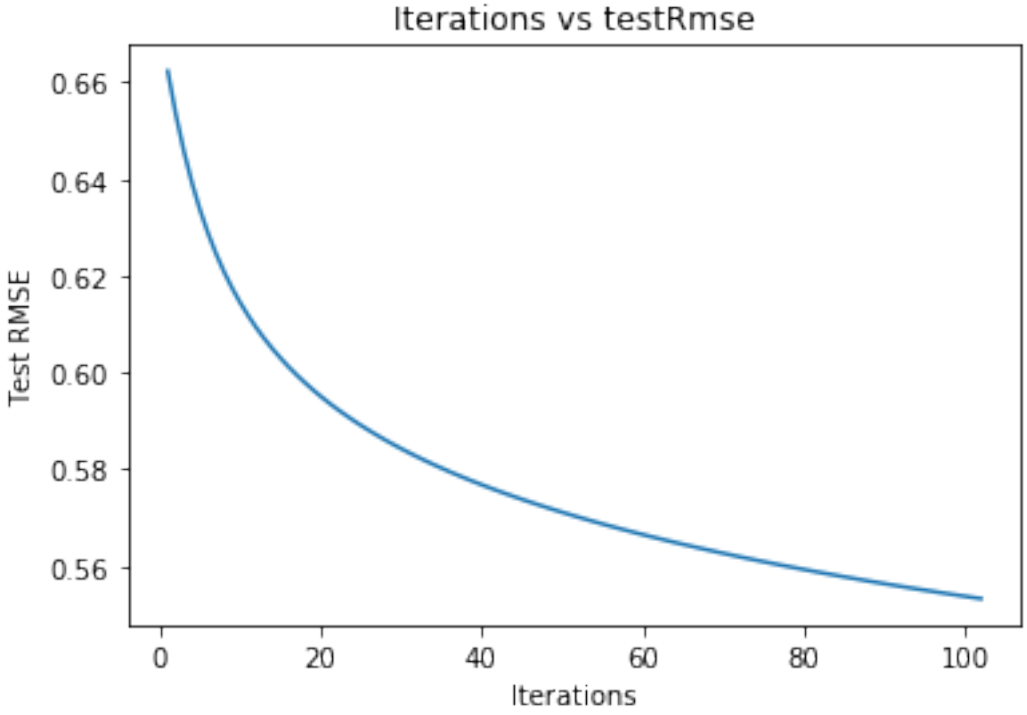

time vs testRmse
Test RMSE
0.66
0.64
0.62
0.60
0.58
0.56
0
20
40
60
80
100
120
140
time

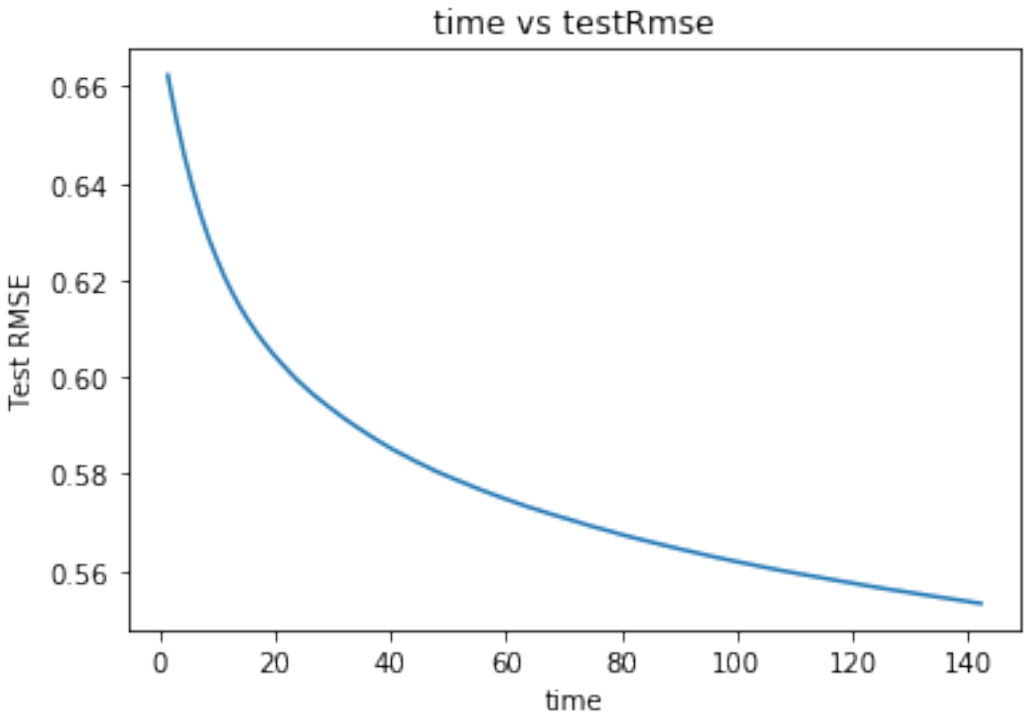

time vs trainRmse
Train RMSE
0.66
0.64
0.62
0.60
0.58
0.56
0
20
40
60
80
100
120
140
time

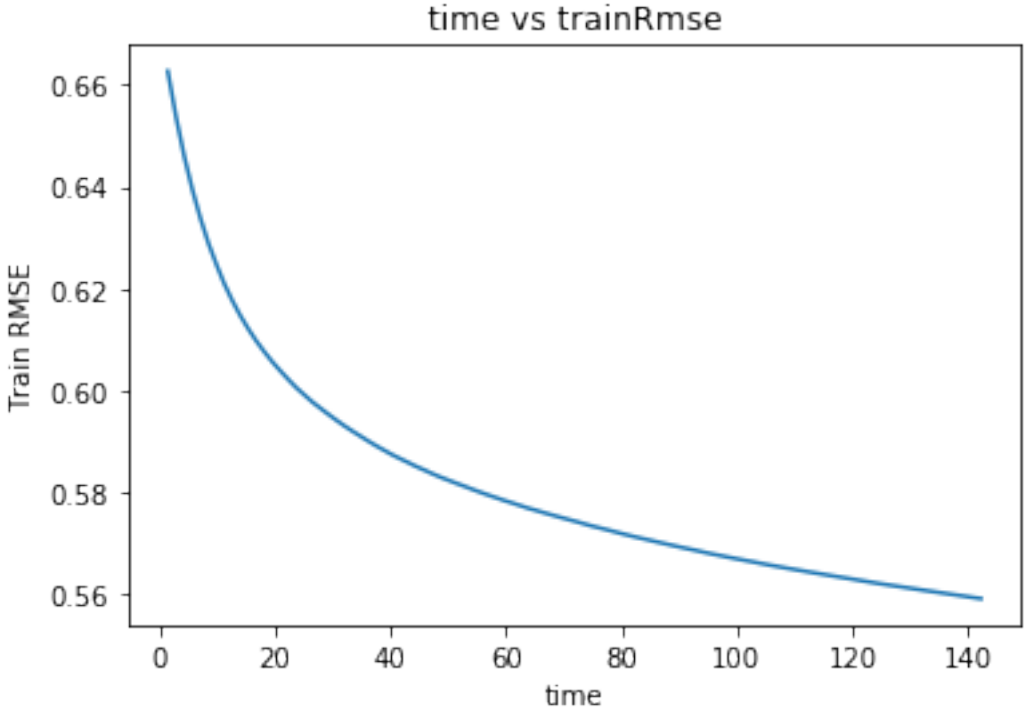
In [12]:
```
#Plotting efficiency graphs for Kdd dataset
#Iterations
cnt = [i for i in range(1,103)]

print("7 Processes")
#Iterations vs trainRmse
plt.plot(cnt,df7['trainRmse'])
plt.xlabel("Iterations")
plt.ylabel("Train RMSE")
plt.title("Iterations vs trainRmse")
plt.show()

plt.plot(cnt,df7['testRmse'])
plt.xlabel("Iterations")
plt.ylabel("Test RMSE")
plt.title("Iterations vs testRmse")
plt.show()

time = df7['time'].cumsum()
plt.plot(time,df7['testRmse'])
plt.xlabel("time")
plt.ylabel("Test RMSE")
plt.title("time vs testRmse")
plt.show()

plt.plot(time,df7['trainRmse'])
plt.xlabel("time")
plt.ylabel("Train RMSE")
plt.title("time vs trainRmse")
plt.show()
```

```
7 Processes
```


Iterations vs trainRmse
Train RMSE
0.66
0.64
0.62
0.60
0.58
0.56
0
20
40
60
80
100
Iterations


Iterations vs testRmse
Test RMSE
0.66
0.64
0.62
0.60
0.58
0.56
0
20
40
60
80
100
Iterations


time vs testRmse
Test RMSE
0.66
0.64
0.62
0.60
0.58
0.56
0
20
40
60
80
100
120
140
time


time vs trainRmse
Train RMSE
0.66
0.64
0.62
0.60
0.58
0.56
0
20
40
60
80
100
120
140
time

Similarly, for KDD better performance was observed in more number of processes. Due to scale constraints seperate graphs have to be plotted

For sequential timing, the following code snippet was used. However it is not a one to one comparison with the paralle code, as we use scikit for this. Also, loss graphs could not be plotted for the same due to non-availability in sklearn. This performs faster than the others, still only due to optimizations used in sklearn

```
In [ ]: from sklearn.datasets import load_svmlight_file
        from scipy.sparse import csr_matrix, vstack
        from sklearn.feature_selection import SelectKBest
        from sklearn.feature_selection import chi2
        import matplotlib.pyplot as plt
        from sklearn import linear_model
        from sklearn.svm import SVC
        from sklearn.datasets import load_svmlight_file
        from sklearn.datasets import dump_svmlight_file
        from sklearn.model_selection import train_test_split
        from mpi4py import MPI

        import pandas as pd
        import numpy as np
        import time
        import scipy
        import os
        import re

        #Initialize communicators
        comm = MPI.COMM_WORLD
        size = comm.Get_size()
        rank = comm.Get_rank()



        def virus():
            path = "/home/kritz/Downloads/dataset/"
            label = []
            data = np.array([])
            #Listing all the files in directory
            for rootdir, subdirs, files in os.walk(path):
                if files:
                    for file in sorted(files):
                        #Reading the data into sparse matrix from libsvm
                        x,y = load_svmlight_file(rootdir+'/'+file)
                        #Adding empty rows to compensate for missing columns in these four fil
                        if file in ["2011-09.txt","2013-07.txt","2013-09.txt","2014-01.txt"]:
                            update = np.zeros((x.shape[0],1))
                            x = scipy.sparse.hstack((x,np.array((update))))
                        label.extend(y)
```

```
                    #Adding all files to same csr matrix
                    data = scipy.sparse.vstack((data,x))
        print(len(y),data.shape)
        data = data.tocsr()
        return(data[1:,].tocsr(),label)

    #Pre-processing Kdd data
    def Kdd():
        path = "/home/kritz/Downloads/cup98LRN.txt"
        df = pd.read_csv(path,low_memory=False)

        #Getting the numerical and categorical columns
        cols = df.columns
        numCol = df._get_numeric_data().columns
        catCol = list(set(cols) - set(numCol))
        #Type casting for float and string
        for each in numCol:
            df[each] = df[each].astype('float')
        for each in catCol:
             df[each] = df[each].astype('category')

        #Filling mean values to Nans
        newDf = pd.get_dummies(df,drop_first = True)
        newDf = newDf.fillna(newDf.mean())

        #Reducing the dimension of predictor space based on variable importance
        X, y = newDf.loc[:,newDf.columns != 'TARGET_D'], newDf.loc[:,df.columns == 'TARGET_
        XNew = SelectKBest(chi2, k=200).fit_transform(X, y)

        return(scipy.sparse.csr_matrix(XNew),y)

    if rank == 0:
        tic = time.time()
        data,y = Kdd()
        x = data[:,:6000]
        y = np.array(y[:6000]).ravel()

        xTrain, xTest, yTrain, yTest = train_test_split(x, y, test_size=0.30, random_state=

        clf = linear_model.SGDRegressor()
        clf.fit(xTrain,yTrain)
        print(time.time()-tic)
```

Virus Data : Sequential time : 8.08 s Kdd Data : Sequential time : 12.87 s

Note : Alpha and epsilon are hyper-parameters, they can be optimized by cross validation, which could not be done due to the amount of data and time constraints. An arbitray value is observed here, which may not lead to optimal results