README.md

Distributed Stochastic Gradient descent

1 Stochastic gradient descent for Linear regression

Gradient descent is iterative algorithm to find the function minimum using first order derivative of the function. For regression problem the prediction defined as

yPrediction = XB

X is feature set and B is weight vector

The first order derivative for RSS *(yTrue - yPrediction)^2* is given by

*-1 * X.T(Y - XB)*

For large dataset calculating same could be very expensive and so stochastic gradient descent is used where idea is to perform gradient update one document at a time.

2 Distributed stochastic gradient descent for Linear regression

This project tries to extends the above idea of stochastic gradient descent for regression for parallel computing using message passing interface(MPI). The general idea for distributed stochastic gradient descent is that each worker will have its own chunk of data and will minimize the function using stochastic gradient descent for the available sub dataset. However the final output should be the function minimal for entire dataset and not n minimals from n workers. So at end of each epoch an approximate minimum is calculated by averaging the local minimum of each workers.

3 Project Structure

There are multiple modules in this project. This section describes each of this section in depth.

```
foo@bar:~$ ll
drwxrwxr-x  9 foo foo   4096 May 11 06:46 ./
drwxrwxr-x 11 foo foo   4096 May 10 22:02 ../
-rw-rw-r--  1 foo foo  28985 May 10 20:15 Classification.ipynb
drwxrwxr-x  4 foo foo   4096 May  5 10:32 Data/
-rw-rw-r--  1 foo foo   2745 May 11 06:45 kddcup98.py
drwxrwxr-x  3 foo foo   4096 May  5 22:27 Logger/
drwxrwxr-x  2 foo foo   4096 May  5 22:27 Out/
drwxrwxr-x  3 foo foo   4096 May 10 17:57 PerformanceMetric/
drwxrwxr-x  3 foo foo   4096 May  5 22:58 Regression/
-rw-rw-r--  1 foo foo   2414 May 10 22:04 SGDVirusDataset.py
-rw-rw-r--  1 foo foo 145323 May 11 06:46 timeit.log
drwxrwxr-x  3 foo foo   4096 May 10 15:40 Utils/
```

As shown above the project is broken in multiple modules.

3.1 Logger

This module includes script for storing the time taken for execution.

```
import time
import logging

logger = logging.getLogger('timeLogger')
logger.setLevel(logging.DEBUG)
fh = logging.FileHandler('timeit.log')
fh.setLevel(logging.DEBUG)
ch = logging.StreamHandler()
ch.setLevel(logging.ERROR)
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
fh.setFormatter(formatter)
ch.setFormatter(formatter)
logger.addHandler(fh)
logger.addHandler(ch)


def timeit(f):
        def timed(*args, **kw):
                ts = time.time()
                result = f(*args, **kw)
```

```
                te = time.time()
                logger.info('Function:%r  Execution Time: %2.4f sec' %(f.__name__, te-ts))
                return result
        return timed
```

For MPI processes it is necessary to calculate time taken by some function to execute. For this the function is wrapped between timers and difference between end and start time is the total execution time by that function. This increases unnecessary code foot print which is reduced by this module. The above method aids in implementation of **Decorator** design pattern. The function that needs to be timed should be decorated as shown below

```
@timeit
def foo():
        return 0
```

Each time this function is called its execution time is calculated and stored in timeit.log file.

3.2 Utils

This module includes all frequently used helper methods.

3.2.1 Data Reader

```
import scipy.sparse as sp
import numpy as np

from sklearn.datasets import load_svmlight_file
from os import  listdir
from os.path import basename, join
from Logger.TimeIt import timeit
"""This module incudes common
utility functions for IO"""

@timeit
def listAllFiles(basePath):
        pathList = []
        for file in listdir(basePath):
                pathList.append(join(basePath, file))
        return pathList

@timeit
def svmLightToNPVectors(pathList, featureCount, dType, includeBias = False):
        csrMatrixList = []
        y = []
        for file in pathList:
                output = load_svmlight_file(file, n_features = featureCount, dtype = dType)
                csrMatrixList.append(output[0])
                y.extend(output[1])
        x = sp.vstack(csrMatrixList)
        if includeBias:
                x = sp.hstack((np.ones((x.shape[0], 1)), x))
        return x.tocsr(), np.array(y).reshape(-1, 1)
```

There are two methods in data reader module. First method listAllFiles takes input as base path and provides output as list of all the files in the base path.
Second method converts multiple files in SVM light format to a single numpy array. Additionally this method also adds bias to the data if required.
The shape of x will be (totalDocuments, totalFeatures) and shape of y will be (totalDocuments, 1).

3.2.2 Preprocessing

```
import numpy as np

from random import shuffle
from Logger.TimeIt import timeit

"""
This module includes all the methods
for data preprocessing
"""
@timeit
def splitDataSet(data, testSetSize):
        testSetSplit = round(len(data)*testSetSize)
        shuffle(data)
        return data[testSetSplit:], data[:testSetSplit]

@timeit
def distributedMaxScaler(data, communicator = None, root = None):
```

```
        maximum = data.max(axis = 0)

        if communicator != None:
                globalMaximum = communicator.gather(maximum, root)
                if communicator.Get_rank() == root:
                        globalMaximum = np.vstack(globalMaximum)
                        globalMaximum = globalMaximum.max(axis = 0).reshape(1, -1)
                maximum = communicator.bcast(globalMaximum, root)

        scaledData = (data - maximum)
        return scaledData


@timeit
def distributedMinMaxScaler(data, communicator = None, root = None):
        minimum = data.min(axis = 0)
        maximum = data.max(axis = 0)
        if communicator != None:
                globalMinimum = communicator.gather(minimum, root)
                globalMaximum = communicator.gather(maximum, root)
                if communicator.Get_rank() == root:
                        globalMaximum = np.vstack(globalMaximum)
                        globalMaximum = globalMaximum.max(axis = 0).reshape(1, -1)
                        globalMinimum = np.vstack(globalMinimum)
                        globalMinimum = globalMinimum.min(axis = 0).reshape(1, -1)
                maximum = communicator.bcast(globalMaximum, root)
                minimum = communicator.bcast(globalMinimum, root)

        scaledData = (data - minimum)/(maximum - minimum + 1)
        return scaledData
```

This module includes methods for data preprocessing. The first method is splitDataSet. This method takes input as a numpy array and provides output as a split in train and test set. Testset size is provided as percentage.
The second method is distributedMaxScaler. This method takes input as data and optionally communicator and root can also be provided if the data scaler is to be executed in distributed environment. If communicator is provided then the maximum data points are calculated within the workers and sent to root. Root will calculate the maximum data point value across the workers and sent it to all workers.
Scaling algorithm : *data - max(data)*
Third method is distributedMinMaxScaler which is also a data scaling method. It works in a similar way as the previous method.
Scaling algorithm : *(data - minimum)/(maximum - minimum + 1)*
1 is added to prevent division by 0. Both the scaling methods stated above can work for non distributed and distributed environment.

3.3 Performance Metric

```
import numpy as np
import math

from Logger.TimeIt import timeit

def SquaredError(yTrue, yPrediction):
        return np.sum((yTrue - yPrediction) ** 2)

def MSE(yTrue, yPrediction):
        return SquaredError(yTrue, yPrediction)/len(yTrue)

def RMSE(yTrue, yPrediction):
        return math.sqrt(MSE(yTrue, yPrediction))

@timeit
def distributedMSE(root, communicator, yTrue, yPrediction):
        localSquaredError = SquaredError(yTrue, yPrediction)
        assimilatedSquaredError = communicator.gather([localSquaredError, len(yTrue)])
        if communicator.Get_rank() == root:
                globalSquaredSum = 0
                globalCount = 0
                for localSquaredError in assimilatedSquaredError:
                        globalSquaredSum = globalSquaredSum + localSquaredError[0]
                        globalCount = globalCount + localSquaredError[1]
                return (globalSquaredSum/globalCount)


@timeit
def distributedRMSE(root, communicator, yTrue, yPrediction):
        globalMSE = distributedMSE(root, communicator, yTrue, yPrediction)
        if communicator.Get_rank() == root:
                return math.sqrt(globalMSE)
```

This module includes methods to calculate the performance of the gradient descent methods. The first method calculates Squared Error which is sum of square of residuals of actual value and prediction. Second method is MSE which averages the squared error. Third method calculates RMSE which is root of MSE.
Next two methods calculate MSE and RMSE but for distributed environment. All workers calculate local errors and send it to root. Root calculates the global error using the local errors calculated by other workers.

3.4 Regression

```
import numpy as np
import math

from mpi4py import MPI
from random import shuffle

from Logger.TimeIt import timeit
from PerformanceMetric import Performance

"""
prediction = XB
"""
def prediction(X, B):
        return np.dot(X, B)

"""The RSS is defined as
RSS  = (y - yPrediction)^2

X = (n, m)
Y = (n, 1)
B = (m, 1)

where y is  a vector of dimension n X 1
B are the weights of dimension 1 X m
and X is of dimension n X m
where m is no of features and n is total no of samples

The gradient of the RSS is
-X.T(y - yprediction)
"""
def getGradient(X, Y, B):
        residual = Y - prediction(X, B)
        return (-1) * np.dot(X.T, residual)


def L2Regularization(regularizationParameter, penalty):
        loss = 2 * regularizationParameter * penalty
        "Remove impact on bias"
        loss[0] = 0
        return loss

"""Calculate stochastic gradient descent
, if a communicator is provided the method will
calculate PSGD else will calculate SGD

Additionally for faster computation for reshuffiling of data
train set and test set is not split, instead
their indices are split"""
@timeit
def calculatePSGD(featureSet, targetSet, trainingIndices, beta, learningRate, maxEpochs,
        regularization = None, penalty = None, communicator = None, tolerance = 1.0e-10):
        prevRMSE = 0
        for i in range(0, maxEpochs):
                print("Epoch-"+str(i+1))
                """shuffle data"""
                shuffle(trainingIndices)
                """Perform gradient descent for each sample"""
                for trainingIndex in trainingIndices:
                        x = featureSet[trainingIndex, None]
                        y = targetSet[trainingIndex, None]
                        gradient = getGradient(x, y, beta)
                        if regularization != None:
                                gradient = gradient + regularization(beta, penalty)
                        beta = beta - (learningRate * gradient)

                if communicator != None:
                        """Get new gradient as average of centroids"""
                        communicator.barrier()
                        beta = communicator.allreduce(beta, op = MPI.SUM)
                        beta = beta/communicator.Get_size()

                predictions = prediction(featureSet[trainingIndices], beta)
                RMSE = Performance.RMSE(targetSet[trainingIndices], predictions)
```

```
                if i%20 == 0:
                        if communicator != None:
                                """Test convergence among all workers"""
                                localConvergence = False
                                if tolerance > math.fabs(prevRMSE - RMSE):
                                        localConvergence = True
                                        print("Converged locally")
                                globalConvergenceList = communicator.allgather(localConvergence)
                                communicator.barrier()
                                globalConvergence = True
                                for localConvergence in globalConvergenceList:
                                        if localConvergence == False:
                                                """Atleast one worker has not converged"""
                                                globalConvergence = False
                                                break
                                if globalConvergence == True:
                                        print("Converged globally")
                                        return beta
                        else:
                                """Test convergence locally"""
                                if tolerance > math.fabs(prevRMSE - RMSE):
                                        print("Converged")
                                        return beta
                prevRMSE = RMSE
                print("RMSE "+str(RMSE))
        return beta
```

Before looking into the methods, we must first understand the expected data dimensions.

Variable Name	Description	Dimensions
X	Represents numpy matrix of features	(documentSize, featureSize)
B	Represents numpy matrix/vector the weight for features also knows as beta	(featureSize, 1)
Y	Represents numpy matrix/vector dependent variable that is to be predicted	(documentSize, 1)

The first method prediction calculates the prediction as dot(X, B). The prediction will have dimension
((documentSize, featureSize) . (featureSize, 1)) = (documentSize, 1)

Second method calculates gradient of the regression model. The gradient will be
*-1 * dot(X.T, residual)*,
where residual is *y - yPrediction*.
The dimensions of the gradient will be
((featureSize, documentSize) . (documentSize, 1)) = (featureSize, 1)

The third method L2Regularization.The regularization(derivative) that is to be added to the gradient is given by
*2 * regularizationParemeter * penalty*
where regularizationParameter is the model parameters that are to be regularized and penalty is hyperparameter for determining the intensity of penalty It is important the bias is not penalized and for this the first term in regularization is set to 0.

Fourth method is calculatePSGD. This method excepts all the parameters as standard SGD but in addition this method also excepts communicator. This is optional parameter and if no communicator is provided then this method will calculate standard SGD. With communicator SGD will run in parallel across multiple workers.
For each SGD step it is necessary that the data is shuffled. It can be expensive task to shuffle the data. PSGD handles this by shuffling the indices rather than the actual numpy array containing the data. This improves the performance of SGD algorithm. Once the indices are shuffled, one entire pass over these indices is performed also known as one epoch. Here the same methods for calculating gradient are used. However as we are traversing one document at a time the documentSize will be 1 for SGD/PSGD. If regularization required, regularization will be added to the gradient. The beta values are updated using this gradient subject to the provided learning rate. This will end one epoch of SGD. However if communicator is provided then all workers will average out the beta value using local values of all workers.

There are two different convergence conditions. If the PSDG is running with no communicator, then the convergence condition will check if gradient has changed in two consecutive epochs subject to tolerance rate. However if the communicator is present, the convergence is when all workers gradient remains unchanged in consecutive epochs. Furthermore the convergence for parallel SGD is checked every 20 iterations. This is done to prevent unnecessary checks as convergence test for PSGD is expensive. This check frequency can be changed as needed.

4 Dataset#1

4.1 Dataset description

The first data set is Dynamic Features of Virus Share Executables. Here we will try to predict the value of the target(continous). The data is present in libsvm format.

4.2 Assumptions

It is assumed that all workers have access to the dataset using DFS. Each worker should be able to read and write to the DFS.

4.3 Execution

```
import sys
import numpy as np
import os

from mpi4py import MPI
from Utils import DataReader
from Utils import Preprocessing
from Regression import SGD
from PerformanceMetric import Performance

"""Command line arguments"""
inputPath = sys.argv[1]
outputPath = sys.argv[2]
maxEpochs = int(sys.argv[3])

featureCount = 482

"""MPI variables"""
comm = MPI.COMM_WORLD
processorCount = comm.Get_size()
currentRank = comm.Get_rank()
root = 0


"""For storing the assigned task to the processor"""
processorTasks = []

if currentRank == root:
        if not os.path.exists(outputPath):
                os.makedirs(outputPath)
        processorTasks = np.array_split(DataReader.listAllFiles(inputPath), processorCount)

processorTasks = comm.scatter(processorTasks, root)
features, target = DataReader.svmLightToNPVectors(processorTasks, featureCount ,np.float32, True)

features = features.toarray().reshape(-1, featureCount + 1)
"""Split data set into train and test set locally.
Each processor has its own copy of train set and
test set"""
totalData = list(range(0, len(target)))
trainSetIndices, testSetIndices = Preprocessing.splitDataSet(totalData, 0.3)


modelParams = None
if currentRank == root:
        modelParams = np.random.uniform(-1e-4, 1e-4, (featureCount + 1, 1))


modelParams = comm.bcast(modelParams, root)



"""Min max scaler"""
features = Preprocessing.distributedMaxScaler(features, comm, 0)

startTime = MPI.Wtime()
modelParams = SGD.calculatePSGD(features, target, trainSetIndices, modelParams,
        1e-12, maxEpochs, communicator = comm)
endTime = MPI.Wtime()
print(endTime - startTime)

predictions = SGD.prediction(features[testSetIndices], modelParams)

globalTestMSE = Performance.distributedMSE(root, comm, target[testSetIndices], predictions)
globalTestRMSE = Performance.distributedRMSE(root, comm, target[testSetIndices], predictions)

predictions = SGD.prediction(features[trainSetIndices], modelParams)
```

```
globalTrainMSE = Performance.distributedMSE(root, comm, target[trainSetIndices], predictions)
globalTrainRMSE = Performance.distributedRMSE(root, comm, target[trainSetIndices], predictions)

if currentRank == root:
        print("Total Train set RMSE is "+str(globalTrainRMSE))
        print("Total Test set RMSE is "+str(globalTestRMSE))
```

The above snippet uses all the modules described before. First the root process will list all the file that are present in the input path. Then the root process will split the task among all the workers.

4.3.1 Parallel Execution

All workers will read the data from the files assigned to them by the root workers. As the data is there in libSVM format, this need to be converted to numpy array for further processing. This is done using svmLightToNPVectors method. This methods expects list of libsvm format files and outputs one numpy matrix for it. Also addBias boolean variable is set to true so this method will add bias additionally to the data.

Now the next task is to split this data in train and test split. Each worker will split the available dataset in train and test set which means that the test set of 30% is also distributed on all workers. The algorithm will not slice the dataset directly instead it will slice the indices. This operation is optimized and is much faster than the alternative. Furthermore these indices will aid in improving the performance of PSGD algorithm as discussed before.

Next step is data preprocessing/scaling. For this dataset we will use max scaler. This needs to be executed across all items in data and the max data/feature used for scaling should be globally maximum and not locally maximum. To achive this distributedMaxScaler is used.

Having done this final step is to execute PSDG in parallel. But for this initial values for weights are required. Furthermore these weights should be same for all the workers. The root worker generates the beta value from uniform distribution and then broadcasts to other workers. Now each workers have same beta values and they can execute the PSGD.

4.4 Hyper Parameters

Hyperparameters are tuned for this dataset using random search and following are their values. Rest are using the default values.

Variable	Value
Beta	Uniform Distribution in range (-1e4, 1e4)
Learning rate	1e-10

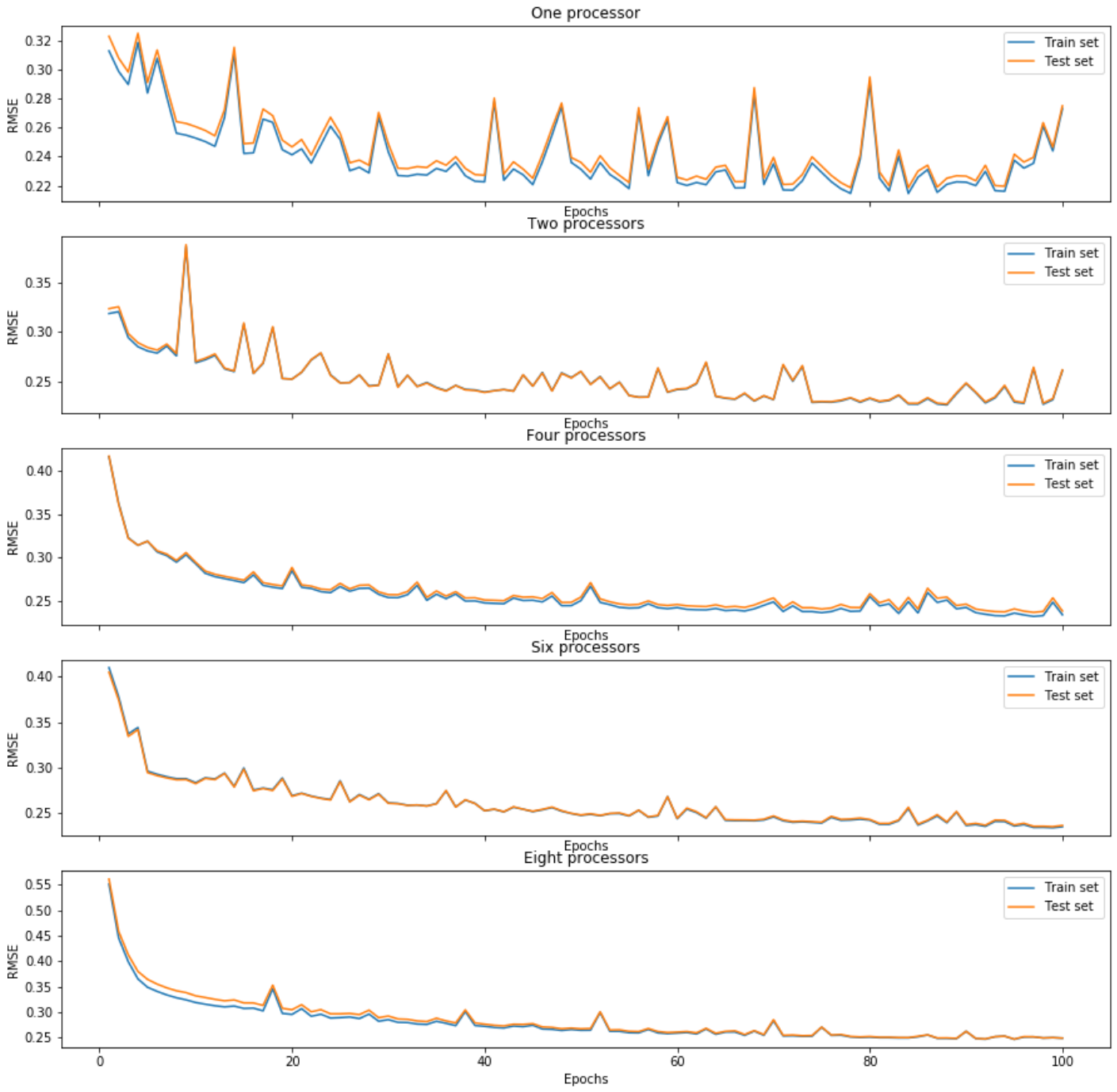
4.5 Output

For this dataset, this algorithm is able to achive RMSE of 0.24 +/- 1 on test set.

4.6 Performance

Processor count	Time taken for one epoch(sec)
1	0.87
2	0.58
4	0.47
6	0.50
8	0.54

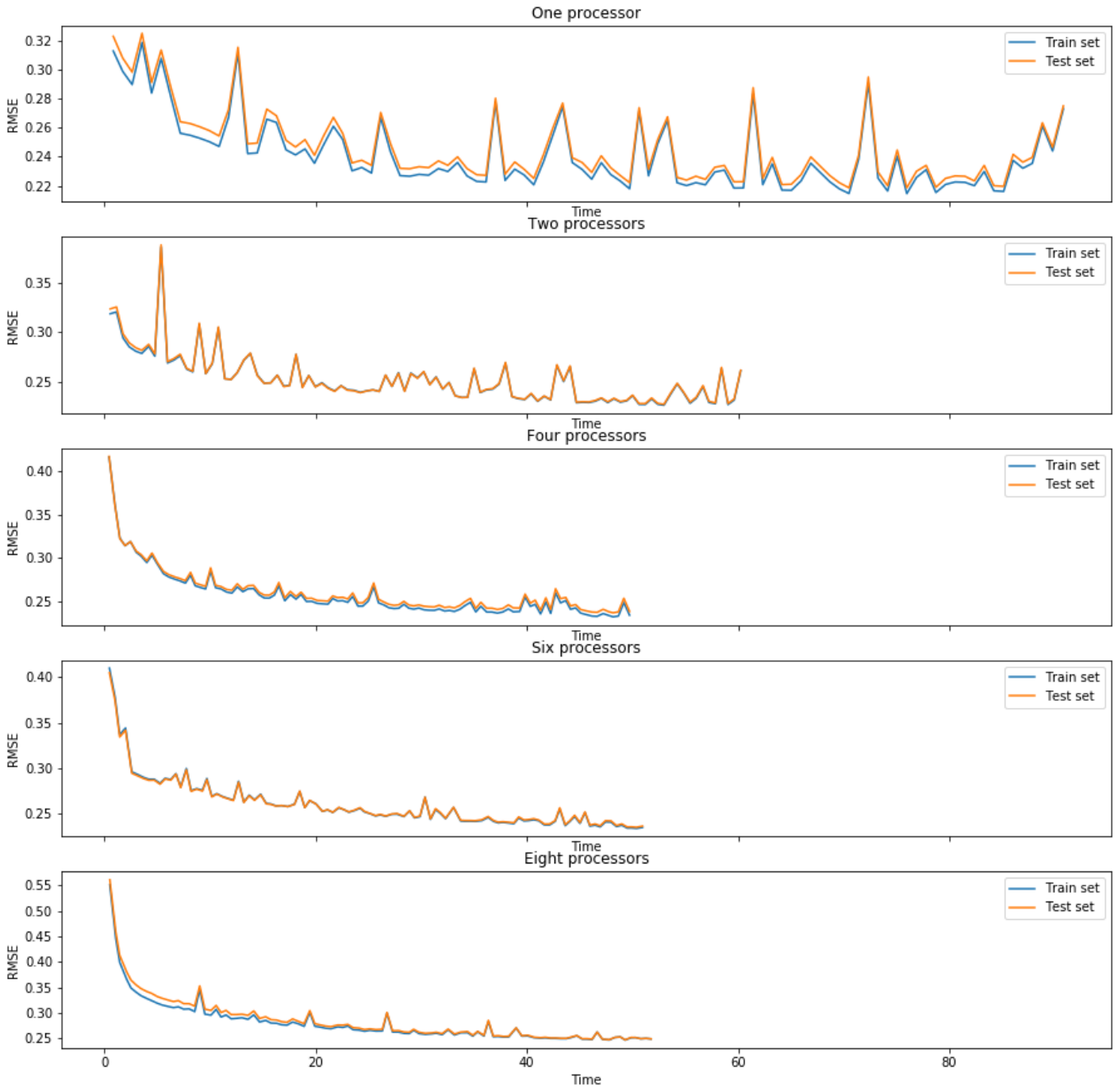
4.6.1 Epoch vs RMSE

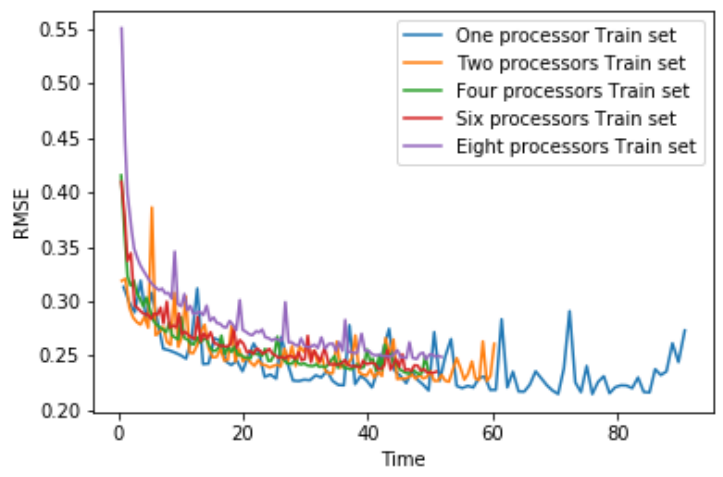
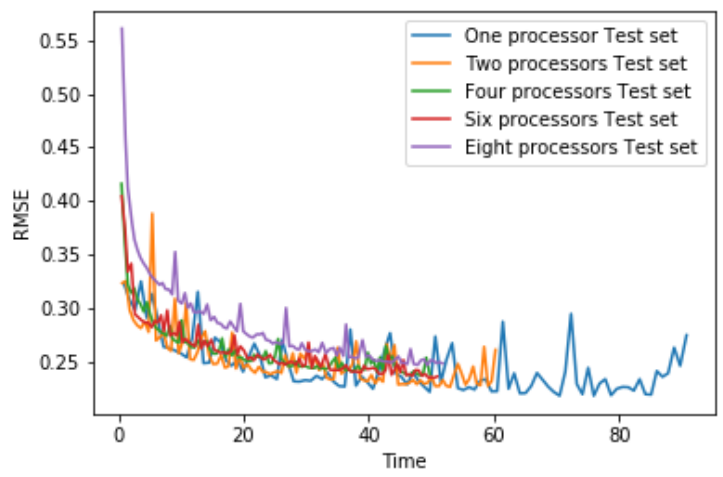


Train Set	Test Set

The above figures describe epochs required vs RMSE values.

4.6.2 Time vs RMSE



Train Set	Test Set
	

The above figures compares RMSE vs Time take to achieve that score.

4.7 Execution Scripts

To execute PSDG for Virus share dataset call shell script as

```
foo@bar:~$ sh virus.sh {ProcessorCount} {InputPath} {OutputPath} {MaxIterations}
```

The above shell script includes following code

```
if test "$#" -ne 4; then
    echo "Illegal number of parameters"
else
    mpiexec -n $1 python3 SGDVirusDataset.py $2 $3 $4
fi
```

5 Dataset#2

5.1 Dataset description

The second dataset is from KDD cup 98 challenge. Here objective is to use regression methods to predict the value of possible donation. Furthermore the dataset has lot of features but many can be dropped.

5.2 Assumptions

Here it is assumed that data is available on only one worker(master/root). All workers must get the data from root worker.

5.3 Execution

```
import sys
import numpy as np
import os
import pandas as pd

from mpi4py import MPI
from Utils import Preprocessing
from Regression import SGD
from PerformanceMetric import Performance

"""Command line arguments"""
inputPath = sys.argv[1]
outputPath = sys.argv[2]
maxEpochs = int(sys.argv[3])

"""MPI variables"""
comm = MPI.COMM_WORLD
processorCount = comm.Get_size()
currentRank = comm.Get_rank()
root = 0


"""For storing the assigned task to the processor"""
processorTasks = []
xTrain = None
yTrain = None
xTest = None
yTest = None


if currentRank == root:
        if not os.path.exists(outputPath):
                os.makedirs(outputPath)
        trainData = pd.read_csv(os.path.join(inputPath, "Train", "cup98LRN.txt"))
        """Selected Features"""
        featureList = ['STATE', 'PVASTATE', 'MDMAUD', 'CLUSTER', 'GENDER', 'HIT',
        'DATASRCE', 'MALEMILI', 'MALEMILI', 'VIETVETS', 'WWIIVETS', 'LOCALGOV', 'STATEGOV',
        'FEDGOV','CARDPROM', 'NUMPROM', 'RAMNTALL', 'NGIFTALL', 'CARDGIFT', 'AVGGIFT', 'TARGET_D']
        trainData = trainData[featureList]
        trainData = pd.get_dummies(trainData)
        processorTasks = np.array_split(trainData, processorCount)


trainData = comm.scatter(processorTasks, root)

"""Feature cleanup"""


features = np.array(trainData.loc[:, trainData.columns != 'TARGET_D'])
target = np.array(trainData['TARGET_D']).reshape(-1, 1)

"""Add bias"""
features = np.insert(features, 0, 1, axis = 1)


"""Split in train and test set"""
totalData = list(range(0, len(trainData)))
trainSetIndices, testSetIndices = Preprocessing.splitDataSet(totalData, 0.3)

modelParams = None
if currentRank == root:
        #modelParams = np.random.uniform(0, 1.5, (features.shape[1], 1))
        modelParams = np.random.uniform(-1.5, 1.5, (features.shape[1], 1))


modelParams = comm.bcast(modelParams, root)
```

```
"""Min max scaler"""
features = Preprocessing.distributedMinMaxScaler(features, comm, 0)
startTime = MPI.Wtime()
modelParams = SGD.calculatePSGD(features, target, trainSetIndices, modelParams,
        4e-5, maxEpochs, communicator = comm, regularization = SGD.L2Regularization, penalty = 0.1)
endTime = MPI.Wtime()
print(endTime - startTime)


predictions = SGD.prediction(features[testSetIndices], modelParams)
globalTestMSE = Performance.distributedMSE(root, comm, target[testSetIndices], predictions)
globalTestRMSE = Performance.distributedRMSE(root, comm, target[testSetIndices], predictions)

predictions = SGD.prediction(features[trainSetIndices], modelParams)
globalTrainMSE = Performance.distributedMSE(root, comm, target[trainSetIndices], predictions)
globalTrainRMSE = Performance.distributedRMSE(root, comm, target[trainSetIndices], predictions)

if currentRank == root:
        print("Total Train set RMSE is "+str(globalTrainRMSE))
        print("Total Test set RMSE is "+str(globalTestRMSE))
```

First task is to read the data on root worker. For this pandas module is used. Furthermore only few features were selected and rest were dropped based upon dataset description. Not all data in data frame is numeric. As a part of preprocessing one hot encoding is generated for selected features.

Having done this the data is divided into chunks and sent to all workers.

5.3.1 Parallel Execution

Once each processor gets their share of data, they seperate the features from target values. The target for regression in this case is *TARGET_D*. Next step is to add bias and split the data set in train and test set. As before each worker has 30% of data as test data. Furthermore the data is slit using indices which is faster and allows PSGD to shuffle data efficiently.

For this dataset MinMaxScaler is used for scaling the data using a distributed varient available in preprocessing module. After this PSGD method called. The initial beta values are required to be same for all workers. Root worker calculates initial random beta value and broadcasts to all the other workers.

Additionally this dataset has lot of imbalance and to prevent overfitting of data L2 regularization is used.

5.4 Hyperparameters

Variable	Value
Beta	Uniform Distribution in range (-1.5, 1.5)
Learning rate	4e-5
L2 penalty	0.1

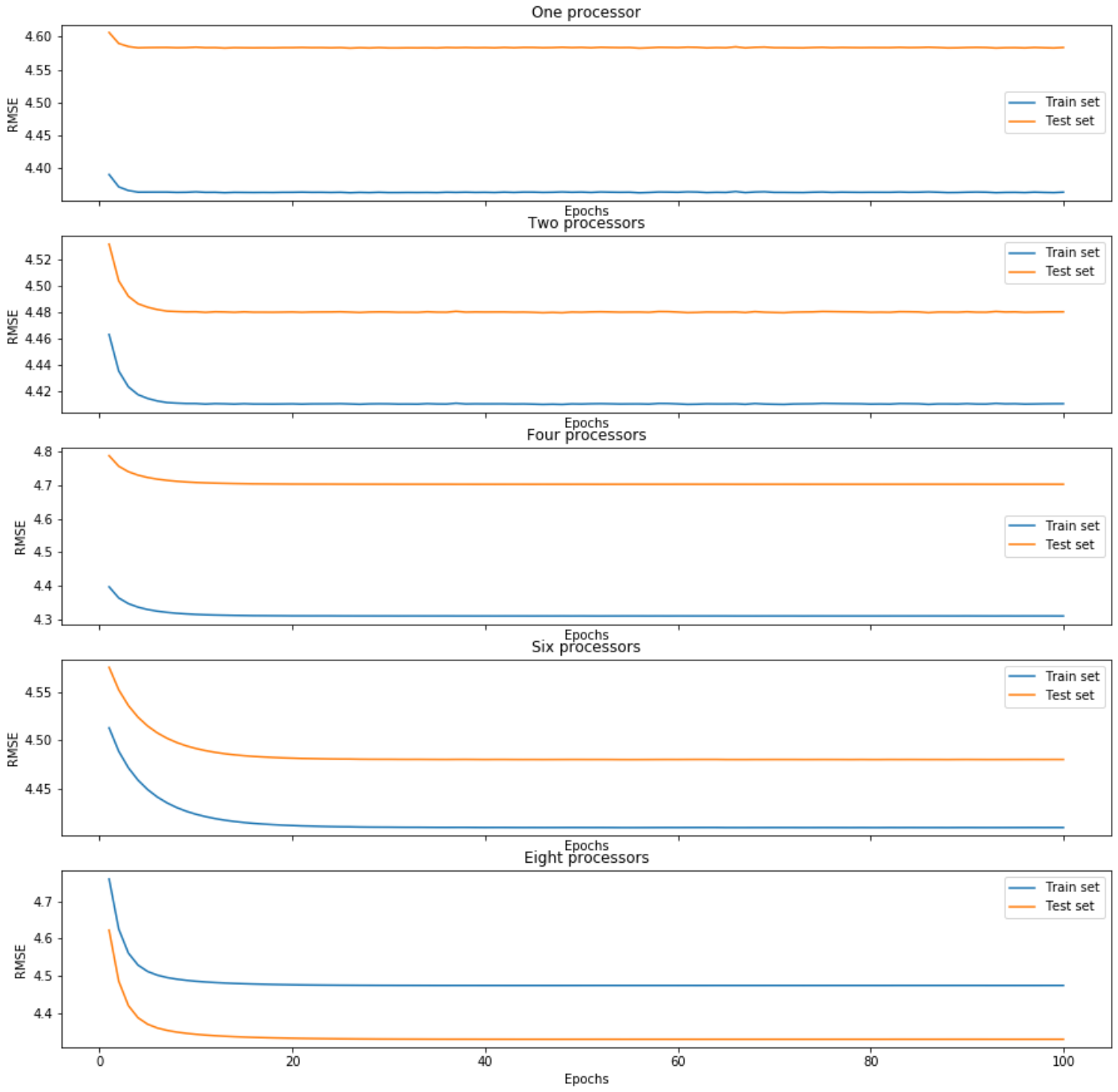
5.5 Output

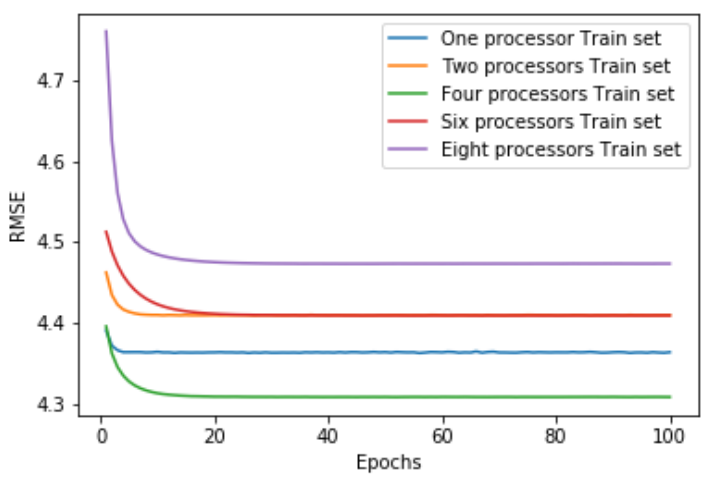
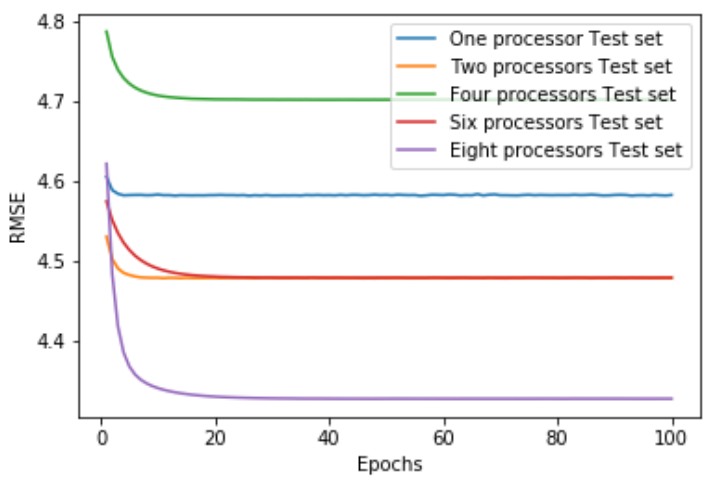
The RMSE for this dataset is 4.4 +/- 1

5.6 Performance

Processor count	Time taken for one epoch(sec)
1	1.06
2	0.73
4	0.47
6	0.55
8	0.611

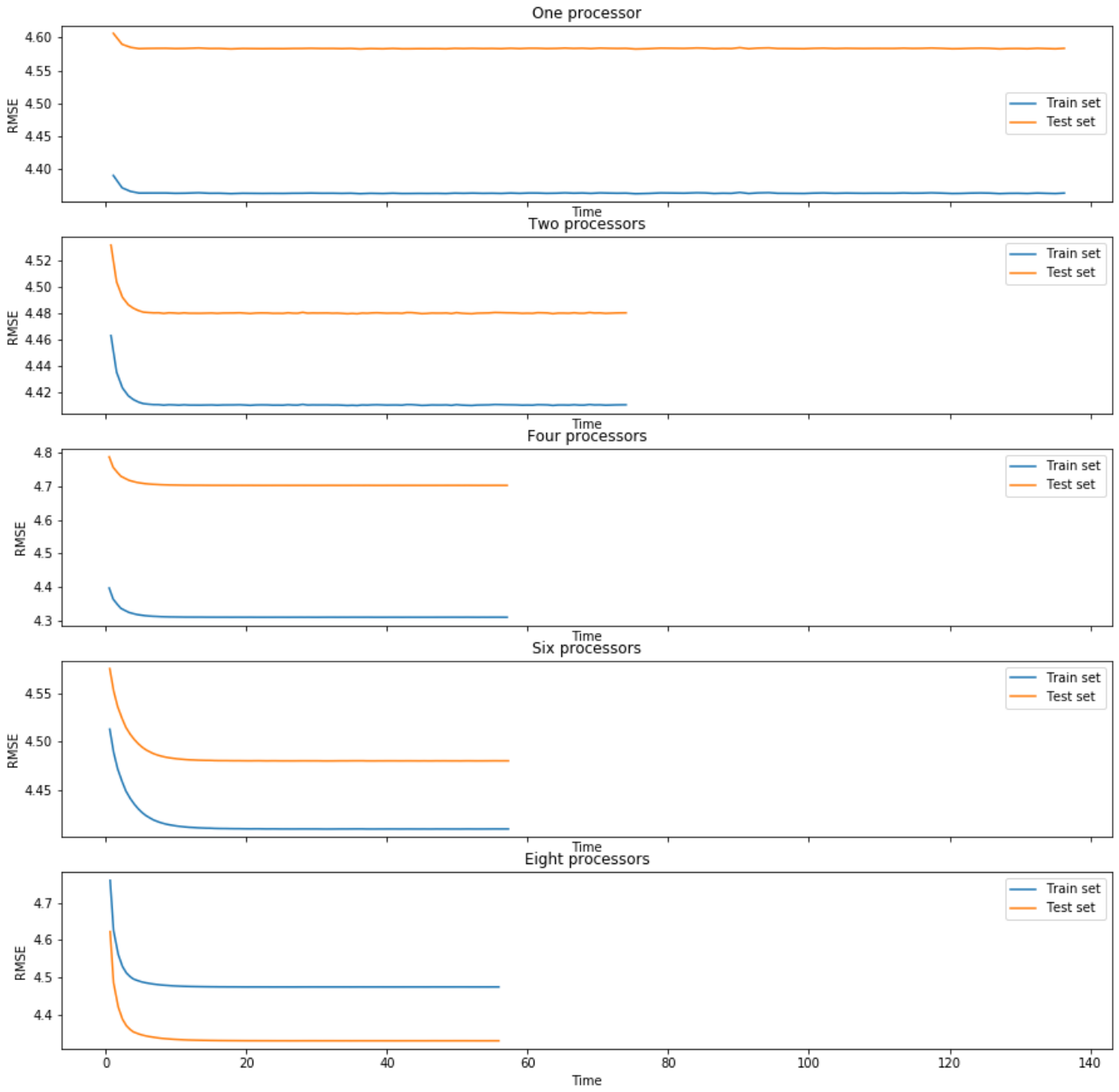
5.6.1 Epoch vs RMSE

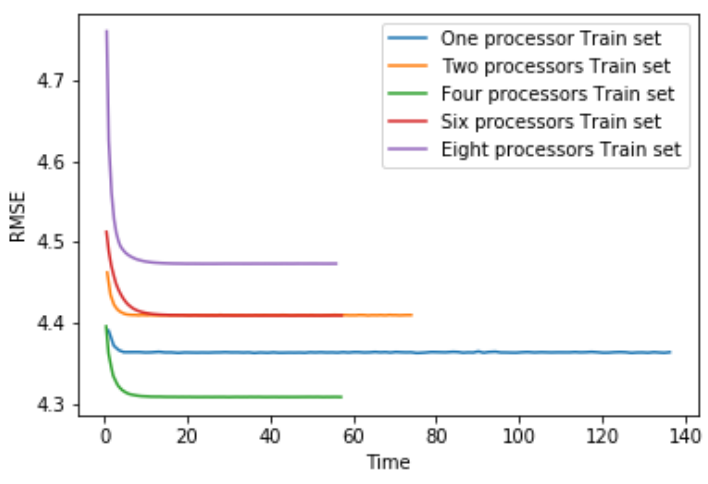
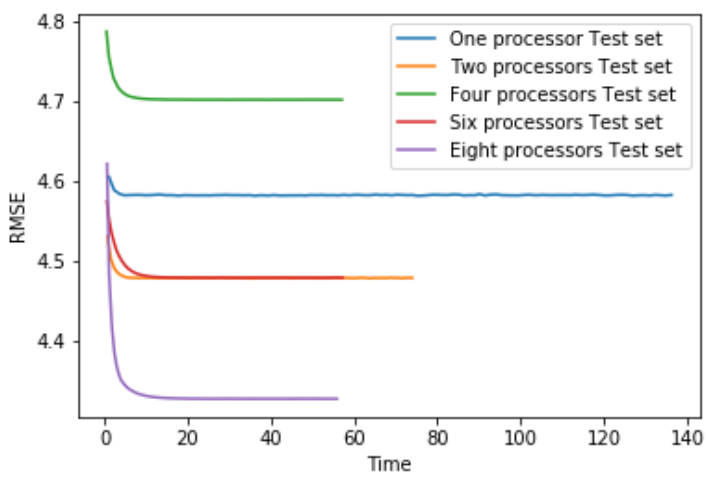


Train Set	Test Set
	

The above figures describe epochs required vs RMSE values.

5.6.2 Time vs RMSE



Train Set	Test Set
	

The above figures compares RMSE vs Time take to achieve that score.

5.7 Execution Scripts

To execute PSDG for KDD cup 98 dataset call shell script as

```
foo@bar:~$ sh kddcup.sh {ProcessorCount} {InputPath} {OutputPath} {MaxIterations}
```

The above shell script includes following code

```
if test "$#" -ne 4; then
    echo "Illegal number of parameters"
else
    mpiexec -n $1 python3 kddcup98.py $2 $3 $4
fi
```

5.8 Optimal solution

This code includes lot of 0 and hence it is difficult to solve it using just linear regression. The optimal solution is to first do classification to filter out potential doners and do the regression on donation amount. See the Ipython notebook.