DDA Lab

Simran Kaur

311443

Exercise 4

DISTRIBUTED MACHINE LEARNING (SUPERVISED)

1. Dynamic Features of VirusShare Executables Data Set

In this exercise sheet you are going to implement a supervised machine learning algorithm in a distributed setting using MPI (mpi4py). We will pick a simple Linear Regression model and train it using a parallel stochastic gradient algorithm (PSGD).

1 Parallel Linear Regression

The first task in this exercise is to implement a linear regression model and learn it using PSGD learning algorithm explained above. You will implement it using mpi4py. A basic version of PSGD could be though of using one worker as a Master, whose sole responsibility is getting local models from other workers and averaging the model. A slight modification could be though of using all the workers as worker and no separate master worker. [Hint: Think of collective routines that can help you in averaging the models and return the result on each worker]. Once you implement your model, show that your implementation can work for any number of workers i.e. P = {2, 4, 6, 8}.

On the Virus share dataset, first data is read and cleaned before learning linear regression and then the data is splitted as 70% training and 30% test data. For parallel processing, the training data is divided into equal chunks and then scattered among all workers, where each of the workers locally update their parameter for the given number of epochs using stochastic gradient descent. Also the test set is broadcasted to all workers, each worker calcules the traing and test error and each epoch and finally the parameters, train error and test error calculated locally is returned to the master node

which finally gives the parameter as the average received from all workers and so is the training and test error.

In [ ]:

```
#importing Libraries
from numpy import int64
import pandas as pd
import os
import numpy as np
from mpi4py import MPI

#Setting MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
data_chunks = None
TestData = None

epochs =  10       # number of epochs
learning_rate = 10**-(12)               # learning rate


def RMSE(data, beta, target):        # this function calculates root mean squared er
    prediction = np.dot(data, beta)
    return np.sqrt((np.sum((prediction - target.reshape(-1,1))**2))/data.shape[0])

def grad(instnce, beta, target):        # this function calculates gradient
    return ((2*(np.dot(instnce, beta) - target))*instnce).reshape(-1,1)

start_time = MPI.Wtime()

# master node
if rank == 0:
    main = 'C:/Users/simra/DDALab/Exercise4/dataset'
    directory = os.scandir(main)
    files = []
    for i in directory:
        array = []
        newpath = os.path.join(i)
        newpath2 = main + '/' + newpath.split("\\")[-1]
        file = open(newpath2, 'r')
        f = file.read()
        num_lines = sum(1 for line in f.split('\n'))     # the text file contains ta
        np_array = np.zeros((num_lines, 482))          # entries in the text file ar
        target_vector = np.zeros((num_lines, 1))          # index and second entry d
        for idx, line in enumerate(f.split('\n')[:-1]):
            values = {}
            target_vector[idx, 0] = float(line.split(' ')[0])
            array = line.split(' ')[1:-1]
            for k, v in [val.split(':') for val in array]:
                values[int(k)] = int(v)
            for j in range(np_array.shape[1]):        # the index where entries are
                if j in values.keys():
                    np_array[idx, j] = values[j]
        indx = np.where(~np_array.any(axis=0))[0]
        clean_array = np.delete(np_array, indx, axis=1)   # the columns whose all en
        files.append(np.hstack((np_array, target_vector)))
    # from each text file in the directory numpy array is created and then stacked o
    Data = np.vstack(files)
    # bias column is added to the numpy array
    bias_column = np.ones(shape=(Data.shape[0],1))
    Data_biasC = np.append(bias_column, Data ,axis=1)
    TrainData = Data_biasC[: int(0.7*Data_biasC.shape[0]), :]   # Data is splitted i
    TestData = Data_biasC[int(0.7*Data_biasC.shape[0]): , :]      # Data is splitted
```

```
    M, N = TrainData.shape
    step = M//size

    data_chunks = []                    # splitting data for distribution among worker
    for i in range(size - 1):
        data_chunks.append(TrainData[i*step:(i+1)*step, :])
    data_chunks.append(TrainData[(size - 1)*step:, : ])

# the test data is broadcasted among all workers for calculating local RMSE at the t
test = comm.bcast(TestData, root = 0)
# each worker receives its chunk of the training data
worker_chunk = comm.scatter(data_chunks, root = 0)
M_chunk, N_chunk = worker_chunk.shape
# parameters for learning algorithm are initialized to be zero
old_params = np.zeros((N_chunk-1, 1))
new_params = np.zeros((N_chunk-1, 1))
RmseTrain = []
RmseTest = []

for _ in range(epochs):
    np.random.shuffle(worker_chunk)       # data is shuffled in every epoch
    for i in range(M_chunk):              # parameters are updated by calculating sgd
        new_params = old_params - learning_rate*grad(worker_chunk[i, :-1], old_param
        old_params = new_params.copy()
    RmseTrain.append(RMSE(worker_chunk[:, :-1], new_params, worker_chunk[:, -1]))
    RmseTest.append(RMSE(test[:, :-1], new_params, test[:, -1]))        # error on t

# parameters, training error and test error are gathered from all workers
local_params = comm.gather(new_params, root = 0)
local_errorTrain = comm.gather(RmseTrain, root = 0)
local_errorTest = comm.gather(RmseTest, root = 0)

if rank == 0:
    weights = np.hstack(local_params)
    global_params = np.mean(weights, axis=1)  # global parameter is calculated by ta
    errorTrain = np.vstack(local_errorTrain)
    errorTest = np.vstack(local_errorTest)
    global_RmseTrain = np.mean(errorTrain, axis = 0)  # global training error
    global_RmseTest = np.mean(errorTrain, axis = 0)        # global test error
    print('Parameters', global_params)
    print('Training Error', global_RmseTrain)
    print('Test Error', global_RmseTest)
    print('Time:', MPI.Wtime()-start_time)
```

Below snippet gives the training and test error when working with two processors

```
(base) C:\Users\simra\DDALab\Exercise4>mpiexec -n 2 python mlexercise.py
Training Error [0.63082007 0.62010539 0.6122011  0.60577842 0.60035012 0.5956592
Test Error [0.63082007 0.62010539 0.6122011  0.60577842 0.60035012 0.5956592
 0.59150017 0.58778133 0.58441693 0.58136213]
Time: 17.51934490003623
```

The snippet gives the training and test error when working with four processors

```
(base) C:\Users\simra\DDALab\Exercise4>mpiexec -n 4 python mlexercise.py
Training Error [0.63704721 0.62899554 0.62268938 0.61745374 0.61294687 0.60897402
 0.60541668 0.60219014 0.5992324  0.59649985]
Test Error [0.63704721 0.62899554 0.62268938 0.61745374 0.61294687 0.60897402
 0.60541668 0.60219014 0.5992324  0.59649985]
Time: 17.03067969996482
```

2 Performance and convergence of PSGD

The second task is to do some performance analysis and convergence tests.

1. First, you have to check the convergence behavior of your model learned through PSGD and compare it to a sequential version. You will plot the convergence curve (Train/Test score verses the number of epochs) for P = {1, 2, 4, 6, 7}. You have to use any sequential version of Linear Regression for P = 1 (Only for sequential version you can use sklearn).

Training Error calculated over different number of processors

In [17]:
```
Training_Error_1 =  [0.62150259, 0.60783031, 0.59842418, 0.59132093, 0.58553963, 0.5
 0.57669138, 0.57321089, 0.57016935, 0.56752201]
Training_Error_2 = [0.63083864, 0.62015291, 0.61219215, 0.605777, 0.60038236, 0.5956
 0.59150256, 0.58778283, 0.58442463, 0.58135842]
Training_Error_4 = [0.63702724, 0.62899525, 0.62267627, 0.61745004, 0.61294648, 0.60
 0.60543675, 0.60219693, 0.59923974, 0.59651181]
Training_Error_6 = [0.64038611, 0.63363871, 0.62840499, 0.62400533, 0.62019032, 0.61
 0.61374765, 0.61094875, 0.60836731, 0.60596204]
Training_Error_7 = [0.64150478, 0.63530681, 0.63052847, 0.62646605, 0.62289035, 0.61
 0.61678323, 0.61411613, 0.6116383, 0.60933897]
```

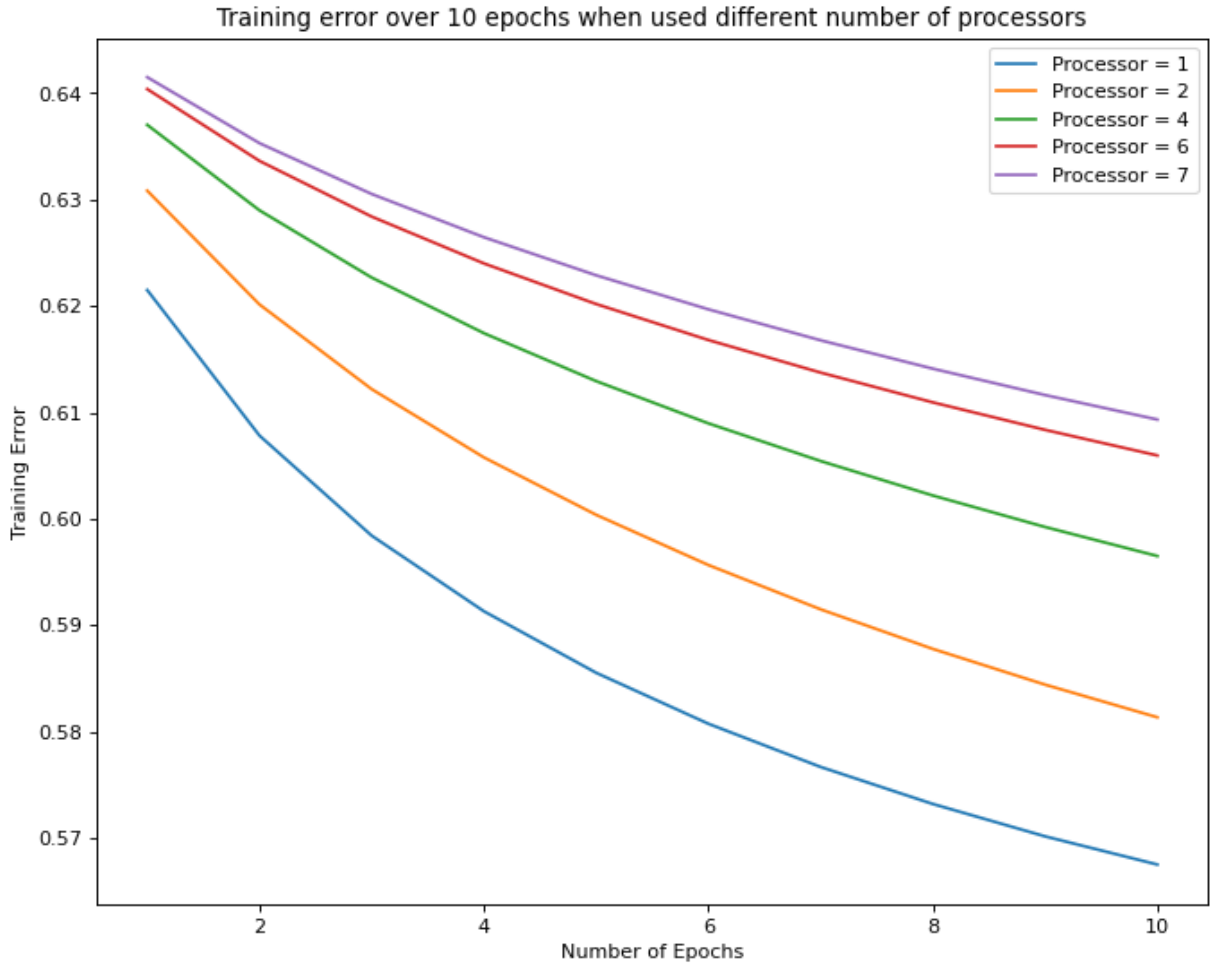
Test Error calculated over different number of processors

In [18]:
```
Test_Error_1 = [0.59405814, 0.58956915, 0.58666688, 0.58464636, 0.58308894, 0.581916
 0.58097914, 0.58023167, 0.57962904, 0.57913425]
Test_Error_2 = [0.59758566, 0.59412593, 0.59162373, 0.5896943, 0.5881704, 0.58690968
 0.58584021, 0.58493244, 0.58413463, 0.58344567]
Test_Error_4 = [0.60064192, 0.59791051, 0.59589193, 0.59431052, 0.59302091, 0.591936
 0.59100365, 0.59016849, 0.58942331, 0.58874936]
Test_Error_6 = [0.60188487, 0.59952835, 0.59785113, 0.59651789, 0.59539525, 0.594416
 0.59355701, 0.59277527, 0.59207214, 0.5914231]
Test_Error_7 = [0.60261857, 0.60041916, 0.59879844, 0.59745821, 0.5963177, 0.5953173
 0.5944432, 0.59366667, 0.59297037, 0.59234795]
```

From the Plot below plotted for training error over a span of 10 epochs for different number of processors shows that with the increase in the number of epochs the training error decreases which is due to the fact that as model goes over the entire data in one epoch and with each increasing epoch it updates its parameters which results in the decreased training error.

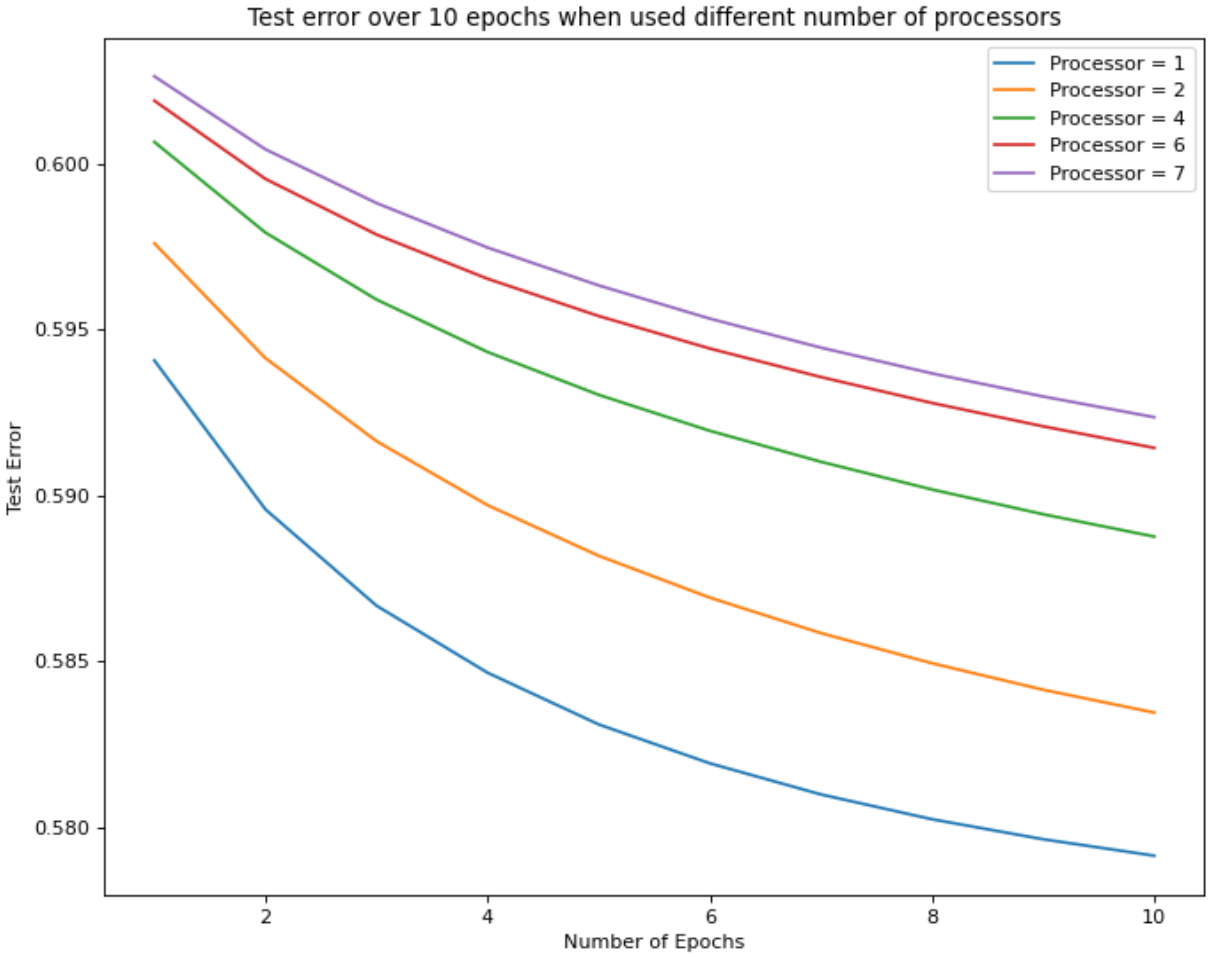
Another important thing seen is that with the increase in the number of processors the training error increases as when more and more processors are used, data is more splitted and since other workers doesn't get to see all data at once, the parameters calculated are not that nicely fitted and hence training error is more.

In [23]:
```
import matplotlib.pyplot as plt
plt.figure(figsize=(10, 8), dpi=80)
import numpy as np
x_axis = np.arange(1, 11)
plt.plot(x_axis, Training_Error_1, label = 'Processor = 1')
plt.plot(x_axis, Training_Error_2, label = 'Processor = 2')
plt.plot(x_axis, Training_Error_4, label = 'Processor = 4')
plt.plot(x_axis, Training_Error_6, label = 'Processor = 6')
plt.plot(x_axis, Training_Error_7, label = 'Processor = 7')
plt.legend()
plt.xlabel('Number of Epochs')
plt.ylabel('Training Error')
plt.title('Training error over 10 epochs when used different number of processors')
plt.show()
```



Same reasoning as above applies to test error.

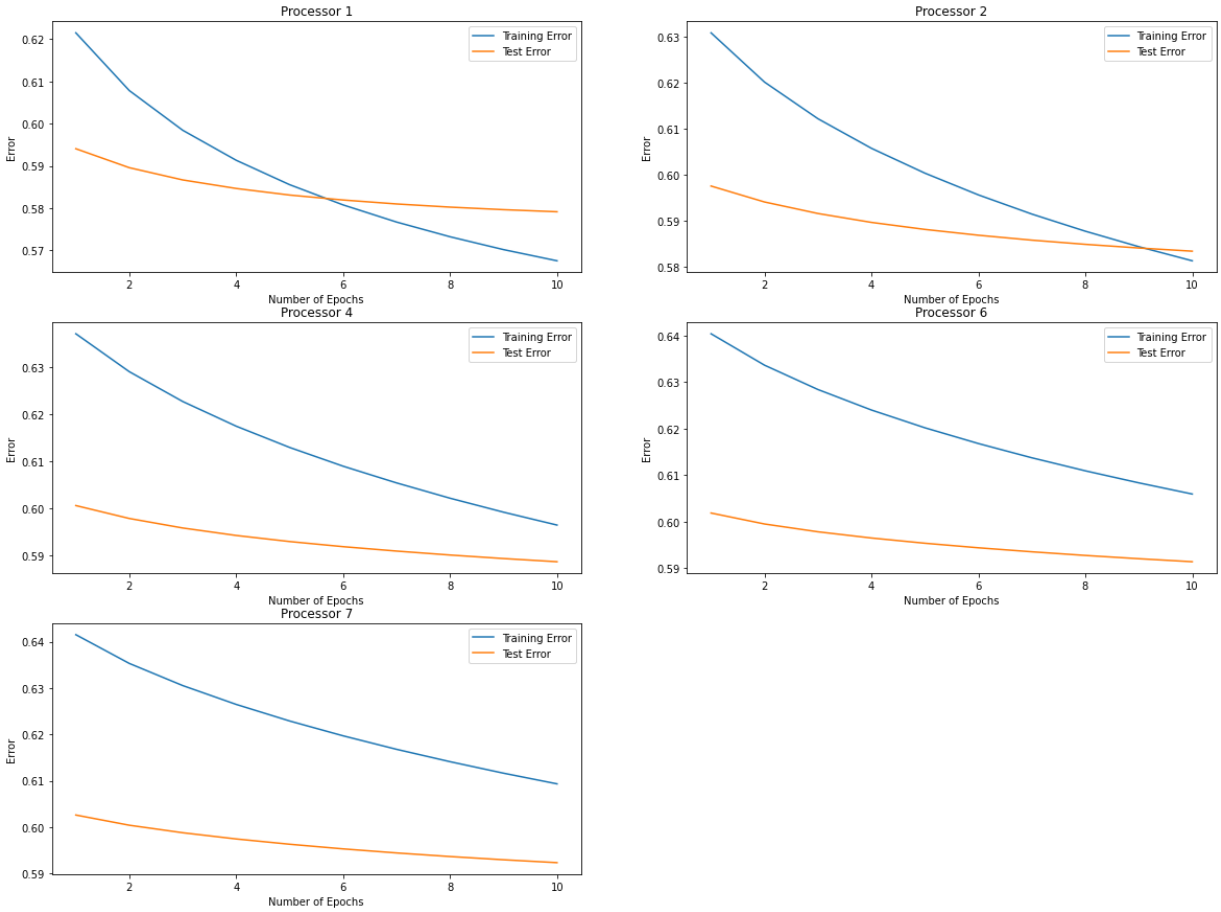
In [24]:
```
plt.figure(figsize=(10, 8), dpi=80)
plt.plot(x_axis, Test_Error_1, label = 'Processor = 1')
plt.plot(x_axis, Test_Error_2, label = 'Processor = 2')
plt.plot(x_axis, Test_Error_4, label = 'Processor = 4')
plt.plot(x_axis, Test_Error_6, label = 'Processor = 6')
plt.plot(x_axis, Test_Error_7, label = 'Processor = 7')
plt.legend()
plt.xlabel('Number of Epochs')
plt.ylabel('Test Error')
plt.title('Test error over 10 epochs when used different number of processors')
plt.show()
```



Below we have plotted training and test error on one plot for different number of processors to show how training and test error behaves, since the testing is done on camparatively smaller set, the test error is smaller than the training error in the beggening and after a number of epochs both tend to converge.

In [21]:
```
fig = plt.figure(figsize=(20, 15))

plt.subplot(3, 2, 1)
plt.plot(x_axis, Training_Error_1, label = 'Training Error')
plt.plot(x_axis, Test_Error_1, label = 'Test Error')
plt.legend()
plt.xlabel('Number of Epochs')
plt.ylabel('Error')
plt.title('Processor 1')

plt.subplot(3, 2, 2)
plt.plot(x_axis, Training_Error_2, label = 'Training Error')
plt.plot(x_axis, Test_Error_2, label = 'Test Error')
plt.legend()
plt.xlabel('Number of Epochs')
plt.ylabel('Error')
plt.title('Processor 2')

plt.subplot(3, 2, 3)
plt.plot(x_axis, Training_Error_4, label = 'Training Error')
plt.plot(x_axis, Test_Error_4, label = 'Test Error')
plt.legend()
plt.xlabel('Number of Epochs')
plt.ylabel('Error')
plt.title('Processor 4')
```

```
plt.subplot(3, 2, 4)
plt.plot(x_axis, Training_Error_6, label = 'Training Error')
plt.plot(x_axis, Test_Error_6, label = 'Test Error')
plt.legend()
plt.xlabel('Number of Epochs')
plt.ylabel('Error')
plt.title('Processor 6')

plt.subplot(3, 2, 5)
plt.plot(x_axis, Training_Error_7, label = 'Training Error')
plt.plot(x_axis, Test_Error_7, label = 'Test Error')
plt.legend()
plt.xlabel('Number of Epochs')
plt.ylabel('Error')
plt.title('Processor 7')

plt.show()
```



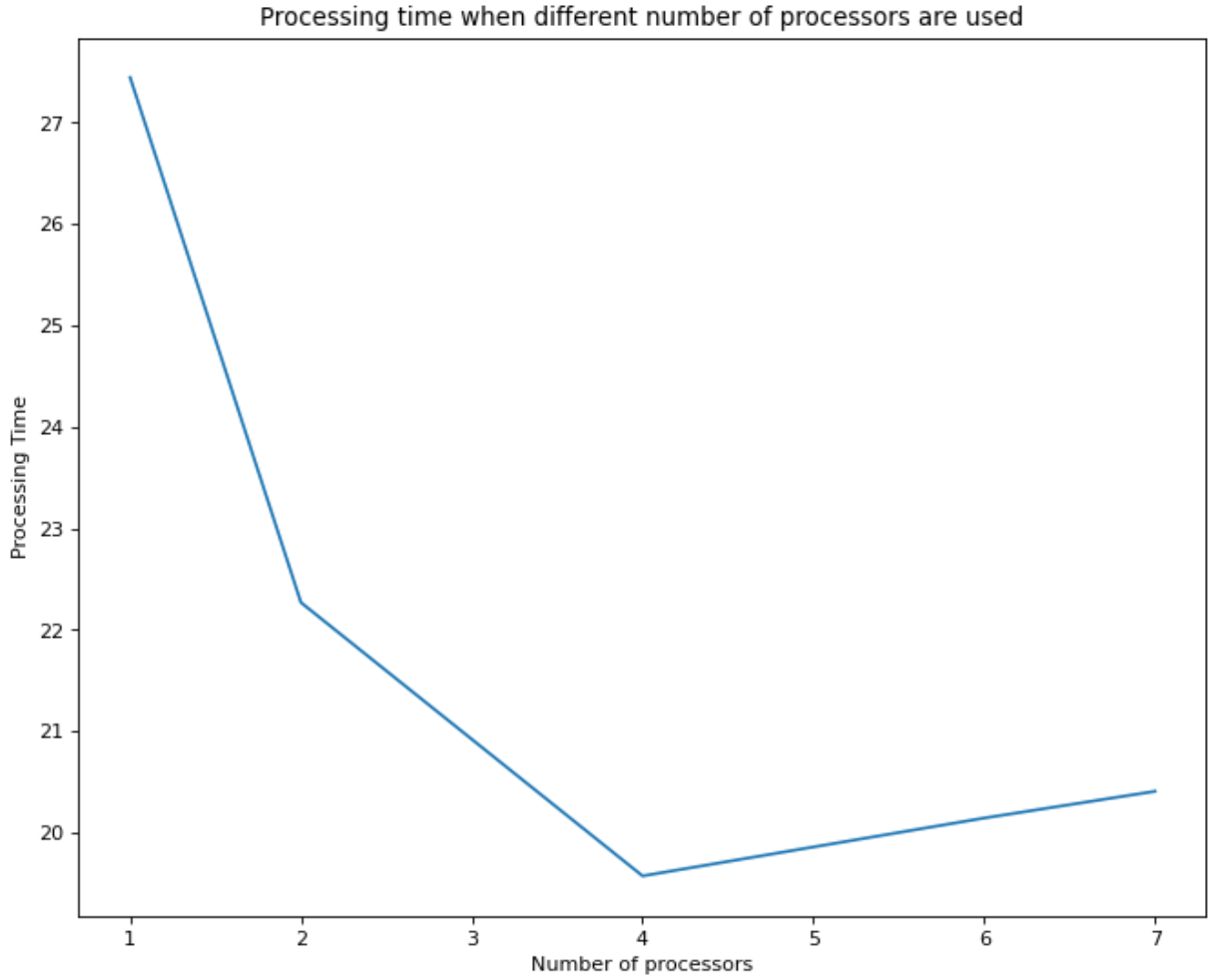
2. Second, you have to do a performance analysis by plotting learning curve (Train/Test scores) verses time. Time your program for P = {1, 2, 4, 6, 7}.

Below we have calculated the processing time when different processors are used and we can see that as we increase the number of processors, the processing time decreases till a particular processor and after that due to increase in the communication cost when more and more number of processors are incorporated, the processing time increases.

In [26]:

```
processors1 = [1, 2, 4, 6, 7]
processor_time1 = [27.445942899999864, 22.270860300000095, 19.57549299999937, 20.146
plt.figure(figsize=(10, 8), dpi=80)
plt.plot(processors1, processor_time1)
plt.xlabel('Number of processors')
```

```
plt.ylabel('Processing Time')
plt.title('Processing time when different number of processors are used')
plt.show()
```



2. KDD Cup 1998 Data Data Set

1 Parallel Linear Regression

The first task in this exercise is to implement a linear regression model and learn it using PSGD learning algorithm explained above. You will implement it using mpi4py. A basic version of PSGD could be though of using one worker as a Master, whose sole responsibility is getting local models from other workers and averaging the model. A slight modification could be though of using all the workers as worker and no separate master worker. [Hint: Think of collective routines that can help you in averaging the models and return the result on each worker]. Once you implement your model, show that your implementation can work for any number of workers i.e. P = {2, 4, 6, 8}.

On the KDD Cup 1998 dataset, first data is read and cleaned before learning linear regression and then the data is splitted as 70% training and 30% test data. For parallel processing, the training data is divided into equal chunks and then scattered among all workers, where each of the workers locally update their parameter for the given number of epochs using stochastic

gradient descent. Also the test set is broadcasted to all workers, each worker calcules the traing and test error and each epoch and finally the parameters, train error and test error calculated locally is returned to the master node which finally gives the parameter as the average received from all workers and so is the training and test error.

In [ ]:

```
# importing libraries
from numpy import int64
import pandas as pd
import os
import numpy as np
from mpi4py import MPI

# Setting MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
data_chunks = None
TestData = None

epochs =  10              # number of epochs
learning_rate = 10**-15           # learning rate


def RMSE(data, beta, target):         # this function calculates root mean squared e
    prediction = np.dot(data, beta)
    return np.sqrt((np.sum((prediction - target.reshape(-1,1))**2))/data.shape[0])

def grad(instnce, beta, target):                    # this function calculates the g
    return ((2*(np.dot(instnce, beta) - target))*instnce).reshape(-1,1)

start_time = MPI.Wtime()

# master node
if rank == 0:
    file = pd.read_csv('cup98LRN.txt', sep = ',')      # reading the txt file throug
    file['MAILCODE']= file['MAILCODE'].replace(' ', 'O')      # this column has spac
    file = file.replace(' ', 'N')             # and rest features with spaces indica
    num_col = file._get_numeric_data().columns              # columns in the datafra
    nonNumColumns = list(set(file.columns)-set(num_col))       # columns whose datat
    new_file = pd.DataFrame([])     # creating an empty dataframe
    for col in nonNumColumns:
        file[col] = file[col].astype('category')     # converting datatype from obje
        file[f'{col}_cat'] = file[col].cat.codes
        new_file = pd.concat([new_file, file[f'{col}_cat']], axis = 1)   # appending
    for col in file.columns:
        if col not in nonNumColumns + ['TARGET_D']:    # the columns having numeric
            new_file = pd.concat([new_file, file[col]], axis = 1)

    new_file = pd.concat([new_file, file['TARGET_D']], axis = 1)   # target column i
    new_file = new_file.fillna(0)      # all nan values are filled with 0s
    Data = new_file.to_numpy()                # converting dataframe to numpy array

    bias_column = np.ones(shape = (Data.shape[0],1))    # adding bias column to the
    Data_biasC = np.append(bias_column, Data ,axis=1)
    TrainData = Data_biasC[: int(0.7*Data_biasC.shape[0]), :]    # splitting 70% dat
    TestData = Data_biasC[int(0.7*Data_biasC.shape[0]): , :]         # splitting 30%
    M, N = TrainData.shape
    step = M//size

    data_chunks = []                  # splitting data for distribution among worker
    for i in range(size - 1):
        data_chunks.append(TrainData[i*step:(i+1)*step, :])
```

```
    data_chunks.append(TrainData[(size - 1)*step:, : ])

# broadcasting the test set over all workers
test = comm.bcast(TestData, root = 0)
# scattering training chunks to all workers
worker_chunk = comm.scatter(data_chunks, root = 0)
M_chunk, N_chunk = worker_chunk.shape

# initializing parameters to zero initially
old_params = np.zeros((N_chunk-1, 1))
new_params = np.zeros((N_chunk-1, 1))
RmseTrain = []
RmseTest = []

for _ in range(epochs):
    np.random.shuffle(worker_chunk)    # shuffling the data in each epoch
    for i in range(M_chunk):       # updating the parameters using sgd taken on rand
        new_params = old_params - learning_rate*grad(worker_chunk[i, :-1], old_param
        old_params = new_params.copy()
    RmseTrain.append(RMSE(worker_chunk[:, :-1], new_params, worker_chunk[:, -1]))
    RmseTest.append(RMSE(test[:, :-1], new_params, test[:, -1]))     # test error

# gathering parameters, train error and test error from each worker
local_params = comm.gather(new_params, root = 0)
local_errorTrain = comm.gather(RmseTrain, root = 0)
local_errorTest = comm.gather(RmseTest, root = 0)

if rank == 0:
    weights = np.hstack(local_params)
    global_params = np.mean(weights, axis=1)   # global parameter is calculated by t
    errorTrain = np.vstack(local_errorTrain)
    errorTest = np.vstack(local_errorTest)
    global_RmseTrain = np.mean(errorTrain, axis = 0)    # global training error
    global_RmseTest = np.mean(errorTest, axis = 0)               # global test error
    # print('Parameters', global_params)
    print('Training Error', global_RmseTrain)
    print('Test Error', global_RmseTest)
    print('Time:', MPI.Wtime()-start_time)
```

Below snippet gives the training and test error when working with seven processors

```
(base) C:\Users\simra\DDALab\Exercise4>mpiexec -n 7 python kddexercise.py
sys:1: DtypeWarning: Columns (8) have mixed types.Specify dtype option on import or set low_memory=False.
C:\Users\simra\DDALab\Exercise4\kddexercise.py:34: PerformanceWarning: DataFrame is highly fragmented.  This is usually the result of calling `frame
.insert` many times, which has poor performance.  Consider joining all columns at once using pd.concat(axis=1) instead.  To get a de-fragmented fram
e, use `newframe = frame.copy()`
  file[f'{col}_cat'] = file[col].cat.codes
Training Error [4.40891642 4.39337342 4.38426667 4.37887368 4.37565249 4.37370786
 4.37248284 4.37168901 4.37115088 4.37076719]
Test Error [4.5851308  4.57028542 4.56161443 4.55649776 4.55345192 4.55162389
 4.55047493 4.54972801 4.54922861 4.54886583]
Time: 52.30884710000094
```

2 Performance and convergence of PSGD

The second task is to do some performance analysis and convergence tests.

1. First, you have to check the convergence behavior of your model learned through PSGD and compare it to a sequential version. You will plot the convergence curve (Train/Test score verses the number of epochs) for P = {1, 2, 4, 6, 7}. You have to

use any sequential version of Linear Regression for P = 1 (Only for sequential version you can use sklearn).

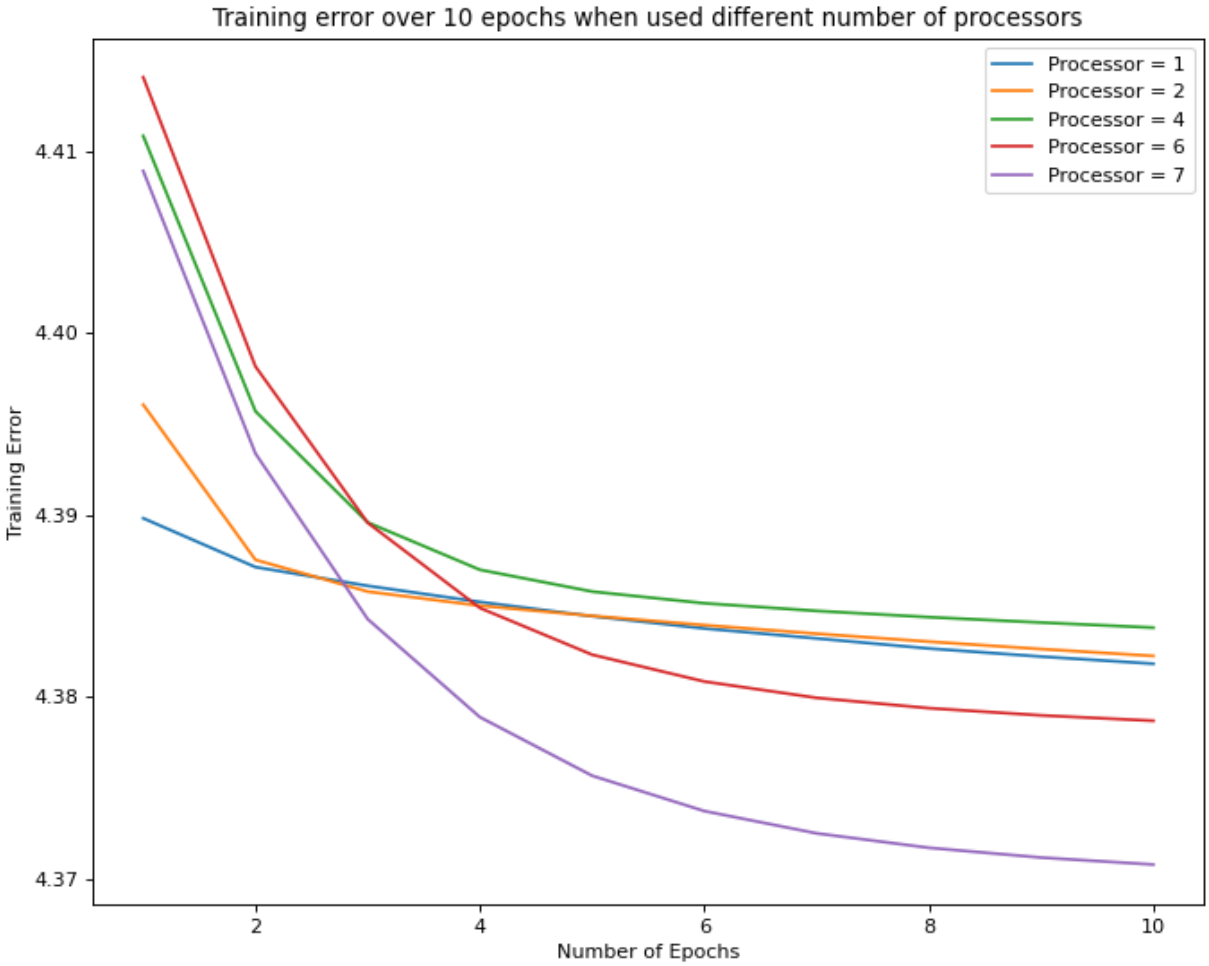
In [28]:
```
TrainingError_1 = [4.38981282, 4.38712817, 4.38609786, 4.38521063, 4.38442598, 4.383
 4.38320346, 4.38265093, 4.38220611, 4.38181268]
TrainingError_2 = [4.39606032, 4.38752707, 4.38578073, 4.38500394, 4.38444296, 4.383
 4.38346413, 4.38302736, 4.38261978, 4.38224027]
TrainingError_4 = [4.41084118, 4.3956968, 4.38958295, 4.38697982, 4.38578147, 4.3851
 4.38471772, 4.38437443, 4.38407754, 4.383798]
TrainingError_6 = [4.41407207, 4.39815974, 4.38956618, 4.38487858, 4.38230893, 4.380
 4.3799355, 4.37936878, 4.37897224, 4.3786754]
TrainingError_7 = [4.40891642, 4.39337342, 4.38426667, 4.37887368, 4.37565249, 4.373
 4.37248284, 4.37168901, 4.37115088, 4.37076719]
```

In [29]:
```
TestError_1 = [4.55040324, 4.54777067, 4.54666679, 4.54569408, 4.54484438, 4.5440908
 4.54346078, 4.54286257, 4.54235429, 4.5418812]
TestError_2 = [4.55843944, 4.55027671, 4.54857945, 4.54779378, 4.54720412, 4.5466728
 4.54616516, 4.5457029, 4.54526099, 4.54484479]
TestError_4 = [4.57317187, 4.55866508, 4.55282508, 4.55035815, 4.54921714, 4.5485956
 4.5481784, 4.54783565, 4.54753156, 4.5472406]
TestError_6 = [4.58204068, 4.56684424, 4.55867055, 4.55423418, 4.55180151, 4.5504082
 4.54955969, 4.54902528, 4.54864328, 4.5483576]
TestError_7 = [4.5851308, 4.57028542, 4.56161443, 4.55649776, 4.55345192, 4.55162389
 4.55047493, 4.54972801, 4.54922861, 4.54886583]
```

From the Plot below plotted for training error over a span of 10 epochs for different number of processors shows that with the increase in the number of epochs the training error decreases which is due to the fact that as model goes over the entire data in one epoch and with each increasing epoch it updates its parameters which results in the decreased training error.

Another important thing seen is that with the increase in the number of processors the training error increases as when more and more processors are used, data is more splitted and since other workers doesn't get to see all data at once, the parameters calculated are not that nicely fitted and hence training error is more.

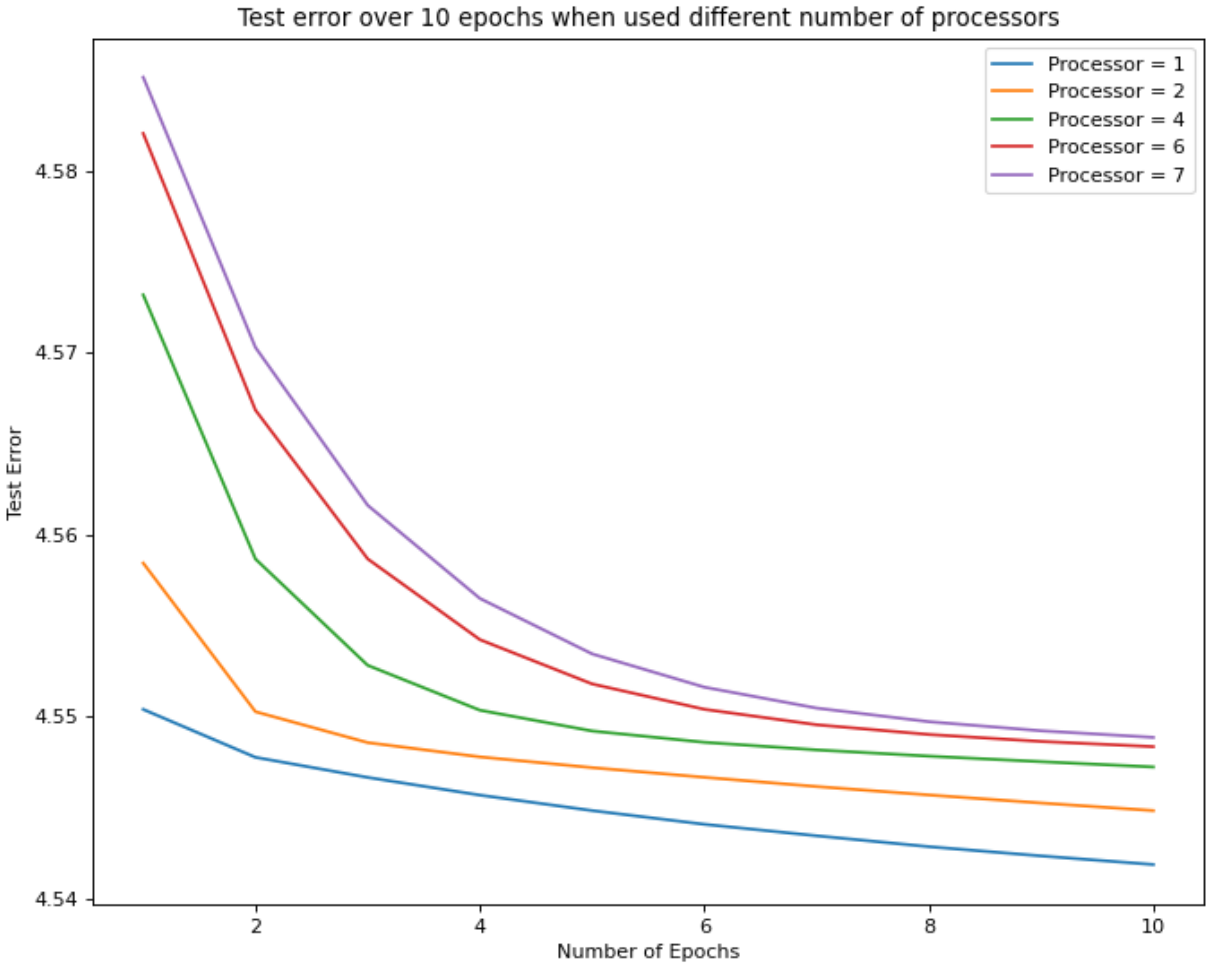
In [32]:
```
import matplotlib.pyplot as plt
plt.figure(figsize=(10, 8), dpi=80)
import numpy as np
x_axis = np.arange(1, 11)
plt.plot(x_axis, TrainingError_1, label = 'Processor = 1')
plt.plot(x_axis, TrainingError_2, label = 'Processor = 2')
plt.plot(x_axis, TrainingError_4, label = 'Processor = 4')
plt.plot(x_axis, TrainingError_6, label = 'Processor = 6')
plt.plot(x_axis, TrainingError_7, label = 'Processor = 7')
plt.legend()
plt.xlabel('Number of Epochs')
plt.ylabel('Training Error')
plt.title('Training error over 10 epochs when used different number of processors')
plt.show()
```



Same reasoning applies to test error.

In [33]:

```
plt.figure(figsize=(10, 8), dpi=80)
plt.plot(x_axis, TestError_1, label = 'Processor = 1')
plt.plot(x_axis, TestError_2, label = 'Processor = 2')
plt.plot(x_axis, TestError_4, label = 'Processor = 4')
plt.plot(x_axis, TestError_6, label = 'Processor = 6')
plt.plot(x_axis, TestError_7, label = 'Processor = 7')
plt.legend()
plt.xlabel('Number of Epochs')
plt.ylabel('Test Error')
plt.title('Test error over 10 epochs when used different number of processors')
plt.show()
```

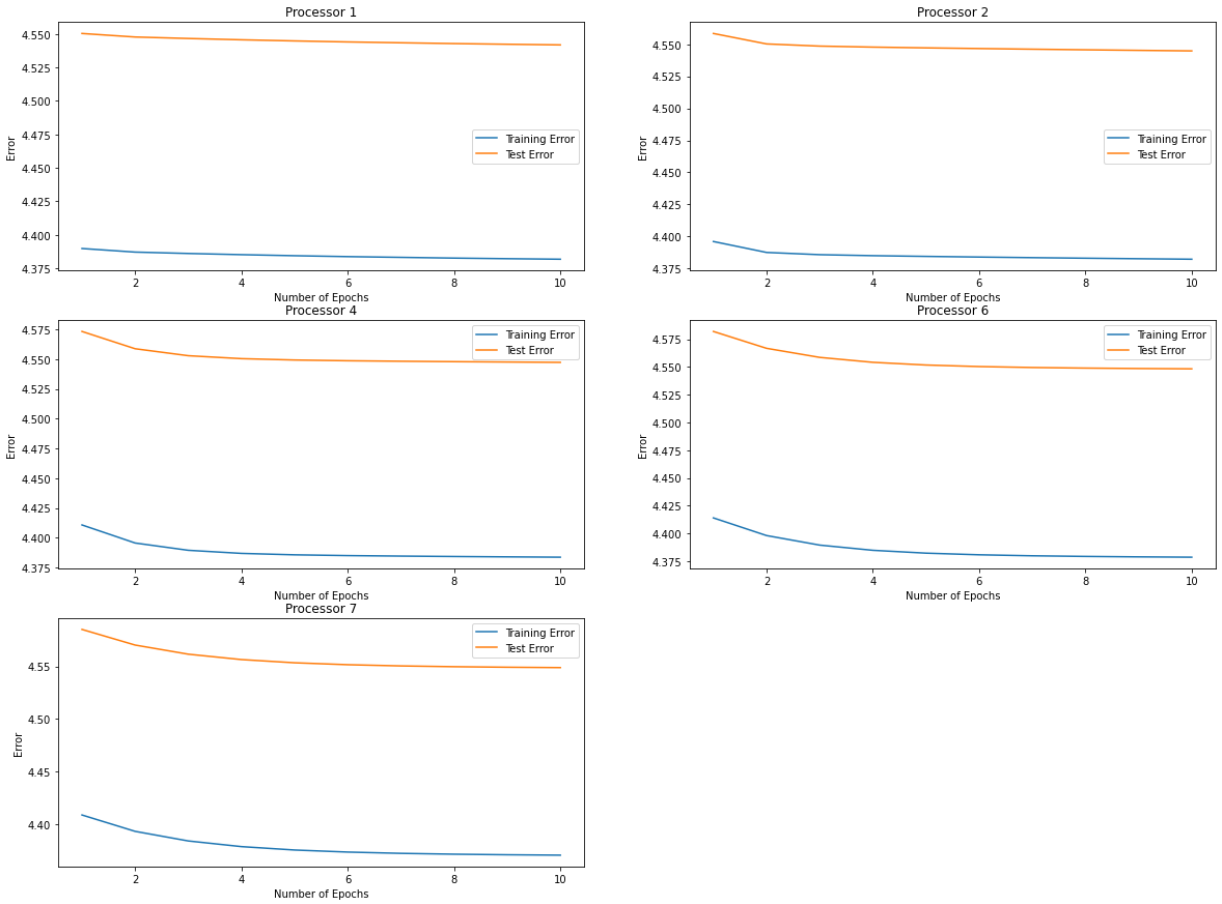


Below we have plotted training and test error on one plot for different number of processors to show how training and test error behaves, since the model is learned over the training set and hence the error is smaller here in the training set as compared to the test set in the begenning, since my machine wasn't working nicely with this big data, I could only plot the result for 10 epochs but it can be showed that as the number of epochs is increased the training and test error would tend to converge.

Another reason could be the label encoding done for this dataset, as label encoding gives different integer values to different instances, the model tend to think that some instances have more weights over the other due to a bigger value they have becasue of encoding and this results in a model learned that is not so efficient.

In [35]:
```
fig = plt.figure(figsize=(20, 15))

plt.subplot(3, 2, 1)
plt.plot(x_axis, TrainingError_1, label = 'Training Error')
plt.plot(x_axis, TestError_1, label = 'Test Error')
plt.legend()
plt.xlabel('Number of Epochs')
plt.ylabel('Error')
plt.title('Processor 1')

plt.subplot(3, 2, 2)
plt.plot(x_axis, TrainingError_2, label = 'Training Error')
plt.plot(x_axis, TestError_2, label = 'Test Error')
plt.legend()
plt.xlabel('Number of Epochs')
plt.ylabel('Error')
plt.title('Processor 2')
```

```
plt.subplot(3, 2, 3)
plt.plot(x_axis, TrainingError_4, label = 'Training Error')
plt.plot(x_axis, TestError_4, label = 'Test Error')
plt.legend()
plt.xlabel('Number of Epochs')
plt.ylabel('Error')
plt.title('Processor 4')

plt.subplot(3, 2, 4)
plt.plot(x_axis, TrainingError_6, label = 'Training Error')
plt.plot(x_axis, TestError_6, label = 'Test Error')
plt.legend()
plt.xlabel('Number of Epochs')
plt.ylabel('Error')
plt.title('Processor 6')

plt.subplot(3, 2, 5)
plt.plot(x_axis, TrainingError_7, label = 'Training Error')
plt.plot(x_axis, TestError_7, label = 'Test Error')
plt.legend()
plt.xlabel('Number of Epochs')
plt.ylabel('Error')
plt.title('Processor 7')

plt.show()
```



2. Second, you have to do a performance analysis by plotting learning curve (Train/Test scores) verses time. Time your program for P = {1, 2, 4, 6, 7}.

Below we have calculated the processing time when different processors are used and we can see that as we increase the number of processors, the processing time decreases till a particular processor and after that due to

increase in the communication cost when more and more number of processors are incorporated, the processing time increases.

In [36]:
```
processors2 = [1, 2, 4, 6, 7]
processor_time2 = [26.966349100000116, 22.100634399999763, 20.241714100000536, 20.14
plt.figure(figsize=(10, 8), dpi=80)
plt.plot(processors2, processor_time2)
plt.xlabel('Number of processors')
plt.ylabel('Processing Time')
plt.title('Processing time when different number of processors are used')
plt.show()
```