DDA Exercise 04

Farjad Ahmed - 1747371
First Semester - Group 1

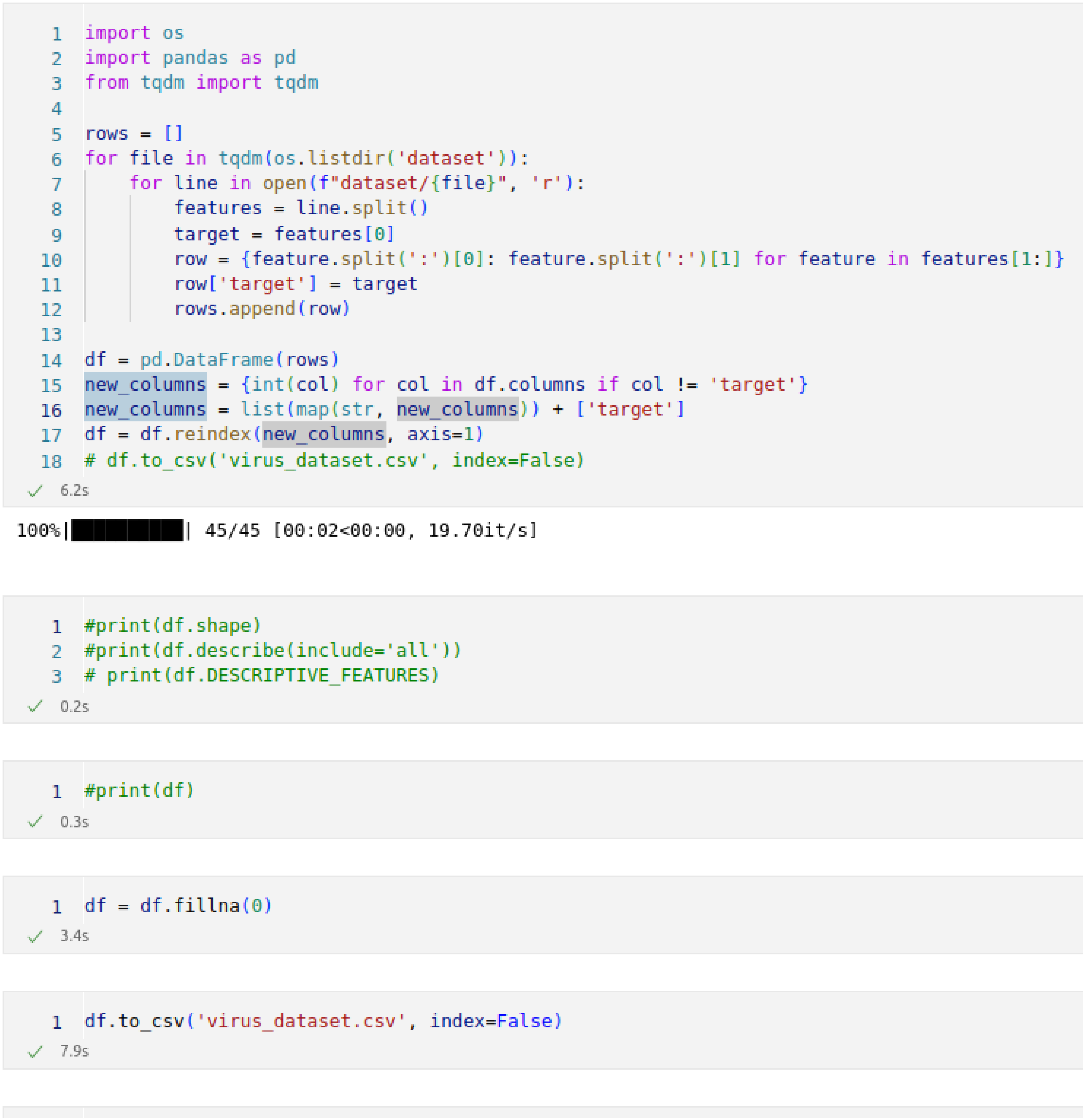
System Information

```
CPU op-mode(s):       32-bit, 64-bit
  Address sizes:        39 bits physical, 48 bits virtual
  Byte Order:           Little Endian
CPU(s):               8
  On-line CPU(s) list:  0-7
Vendor ID:            GenuineIntel
  Model name:           Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
    CPU family:         6
    Model:              142
    Thread(s) per core: 2
    Core(s) per socket: 4
```

This report contains the parallel code for Stochastic Gradient Descent, followed by the tabular answers to the exercise sheet questions.

Code Explanation

The virus dataset was read by first appending all the data files into a dataframe using pandas. Since this created a sparsely populated matrix, a simple approach was applied to fill nan values by zero for simplicity. This dataframe was exported as a csv for the parallel SGD to read and process it further.



```python
import os
import pandas as pd
from tqdm import tqdm

rows = []
for file in tqdm(os.listdir('dataset')):
    for line in open(f"dataset/{file}", 'r'):
        features = line.split()
        target = features[0]
        row = {feature.split(':')[0]: feature.split(':')[1] for feature in features[1:]}
        row['target'] = target
        rows.append(row)

df = pd.DataFrame(rows)
new_columns = {int(col) for col in df.columns if col != 'target'}
new_columns = list(map(str, new_columns)) + ['target']
df = df.reindex(new_columns, axis=1)
# df.to_csv('virus_dataset.csv', index=False)
```

✓ 6.2s

```
100%|██████████| 45/45 [00:02<00:00, 19.70it/s]
```

```python
#print(df.shape)
#print(df.describe(include='all'))
# print(df.DESCRIPTIVE_FEATURES)
```

✓ 0.2s

```python
#print(df)
```

✓ 0.3s

```python
df = df.fillna(0)
```

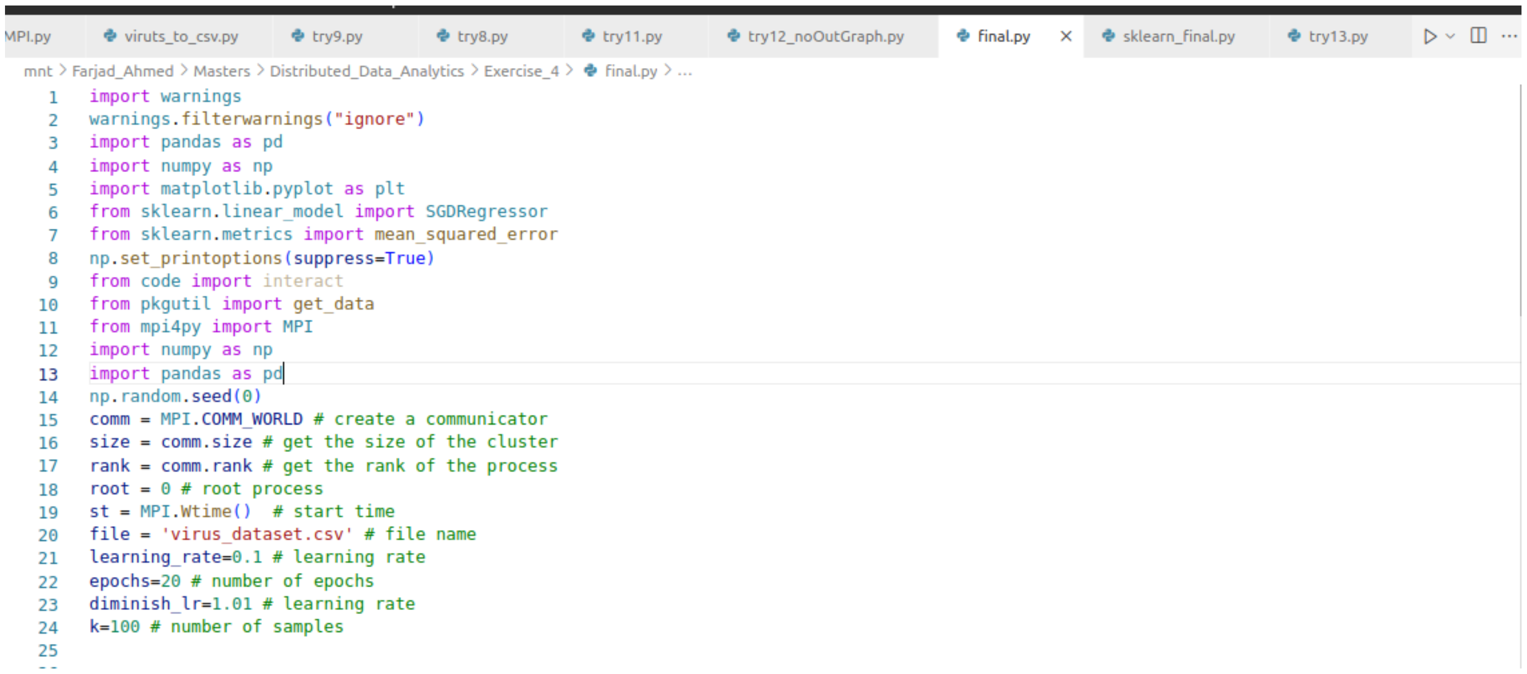
✓ 3.4s

```python
df.to_csv('virus_dataset.csv', index=False)
```

✓ 7.9s

Reading Data

1. Required libraries are imported and MPI communication is enabled. Size, rank are read and root it set. Moreover, time is noted and hyperparameters are set.



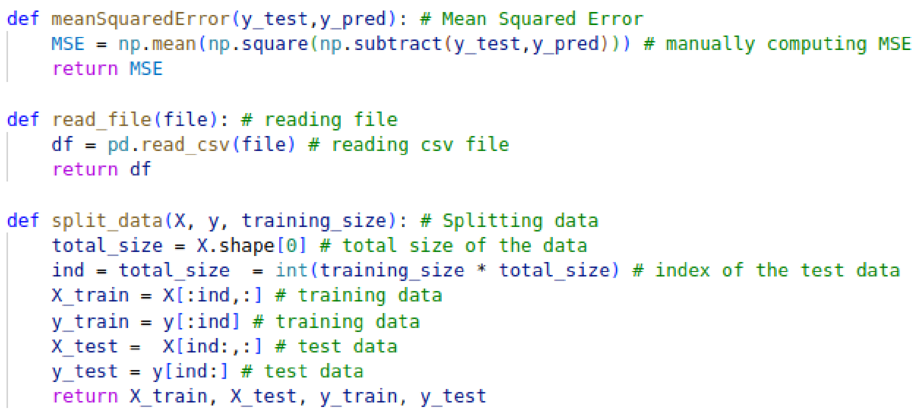
MPI.py | viruts_to_csv.py | try9.py | try8.py | try11.py | try12_noOutGraph.py | final.py | sklearn_final.py | try13.py

mnt > Farjad_Ahmed > Masters > Distributed_Data_Analytics > Exercise_4 > final.py > ...

```python
import warnings
warnings.filterwarnings("ignore")
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import SGDRegressor
from sklearn.metrics import mean_squared_error
np.set_printoptions(suppress=True)
from code import interact
from pkgutil import get_data
from mpi4py import MPI
import numpy as np
import pandas as pd
np.random.seed(0)
comm = MPI.COMM_WORLD # create a communicator
size = comm.size # get the size of the cluster
rank = comm.rank # get the rank of the process
root = 0 # root process
st = MPI.Wtime()  # start time
file = 'virus_dataset.csv' # file name
learning_rate=0.1 # learning rate
epochs=20 # number of epochs
diminish_lr=1.01 # learning rate
k=100 # number of samples
```

It is to be noted that for this whole report the hyperparameters are set as,
epochs : 20
learning rate : 0.1
diminish_lr : 1.01
k = 100

2. Required functions are defined after the initializations.
3. meanSquaredError is a custom implementation to calculate the mean square error for the predicted and the actual target values.



```python
def meanSquaredError(y_test,y_pred): # Mean Squared Error
    MSE = np.mean(np.square(np.subtract(y_test,y_pred))) # manually computing MSE
    return MSE

def read_file(file): # reading file
    df = pd.read_csv(file) # reading csv file
    return df

def split_data(X, y, training_size): # Splitting data
    total_size = X.shape[0] # total size of the data
    ind = total_size  = int(training_size * total_size) # index of the test data
    X_train = X[:ind,:] # training data
    y_train = y[:ind] # training data
    X_test =  X[ind:,:] # test data
    y_test = y[ind:] # test data
    return X_train, X_test, y_train, y_test
```

4. read_file() takes the file name as the argument and reads it. It returns a dataframe.
5. split_data is a custom implementation to split data in training and test sets. It works by taking the training_set size, it calculates the indices from where to split the data and slices the obtained X and y arrays as per the training size.
6. Next, get_data() is defined to take the dataframe returned by the read_file() function and extract the features and target values. The x_test and x_train sets are also standardized by subtracting their mean from the original matrix and then dividing it by the length. This is combined together in a train_data dataframe and the test sets x_test and y_test and also the train sets x_train and y_train, the train sets are returned specifically to compare with the sklearn performance that was being called in the root rank. For the algorithm the train_data packages this whole information together and is later on used in the SGD algorithm as we will see.

```
def get_data(): # reading file
    data = read_file(file) # reading csv file
    data = data.sample(frac=1) # shuffling the data
    data = data.to_numpy() # converting data to numpy array
    # np.random.shuffle(data) # shuffling the data
    Y = data[:, -1] # target variable
    X = data[:,:(data.shape[1]-1)] # features
    x_train,x_test,y_train,y_test=split_data(X,Y, 0.8) # splitting data
    # print("X Shape: ",X.shape) # printing shape of X
    # print("Y Shape: ",Y.shape) # printing shape of Y
    # print("X_Train Shape: ",x_train.shape) # printing shape of X_train
    # print("X_Test Shape: ",x_test.shape) # printing shape of X_test
    # print("Y_Train Shape: ",y_train.shape) # printing shape of Y_train
    # print("Y_Test Shape: ",y_test.shape) # printing shape of Y_test
    # Standardizing data
    x_train = (x_train - np.mean(x_train))/len(x_train) # standardizing X_train
    x_test = (x_test - np.mean(x_test))/len(x_test) # standardizing X_test
    train_data=pd.DataFrame(x_train) # converting X_train to dataframe
    train_data['target'] = y_train # adding target variable to dataframe
    x_test=np.array(x_test) # converting X_test to numpy array
    y_test=np.array(y_test) # converting Y_test to numpy array
    return x_train,x_test,y_train,y_test, train_data
```

7. Sklearn is packaged in a function called sklearn_result to keep track of the benchmark result to tune the algorithm. It takes in the train and test data, initializes and fits the sgdregressor, to later predict and calculate the mean squared error result.

```
def sklearn_result(x_train, y_train, y_test, x_test): # sklearn result
    # SkLearn SGD classifier
    clf_ = SGDRegressor(learning_rate= 'adaptive' , alpha=1, max_iter=500, shuffle=False) # creating SGD classifier
    clf_.fit(x_train, y_train) # fitting the classifier
    y_pred_sksgd=clf_.predict(x_test) # predicting the test data
    print('Mean Squared Error from SKlearn :',meanSquaredError(y_test, y_pred_sksgd)) # printing MSE
```

8. Further, another function is defined to initialize the values of the intercept and gradient of the regression equation, called initialze_terms(). It initializes values to zeros, since it is linear regression, it is possible to initialize these values to zero, it would have been an issue if it were a neural network. I tried various initializations such as zero, random ints and random floats. Random ints delayed the convergence while the other two did not make noticeable impact hence I moved forward with zero initializations.

```
def initialize_terms(shape): # initializing terms
    w=np.zeros(shape=(1,shape-1)) # initializing w
    # w = np.random.uniform(low = 0.0001, high = 0.005, size=(1,shape-1)) # initializing w
    b = 0 # initializing b
    return w, b
```

9. In the SGD algorithm, the gradients are to be calculated for the randomly picked sample and weights are to be updated accordingly, hence a function called update_terms() is defined to do just that. It requires a number of required arguments, particularly, learning_rate, k, x, y, w, b, w_gradient, b_gradient. The for loop runs over all the k samples picked randomly in the SGD algorithm and for each the prediction is calculated, gradients are determined for that prediction and updated. Since there is a batch size, the gradients are updated by normalizing the gradient result towards the end of the loop. Once the loop completes and values are updated, w and b are returned which are the gradients and the intercept respectively.

```
def update_terms(learning_rate, k, x, y, w, b, w_gradient, b_gradient): # updating terms
    for i in range(k): # looping over k samples
            prediction=np.dot(w,x[i])+b # prediction
            w_gradient=w_gradient+(-2)*x[i]*(y[i]-(prediction)) # updating w_gradient
            b_gradient=b_gradient+(-2)*(y[i]-(prediction)) # updating b_gradient
    w=w-learning_rate*(w_gradient/k) # updating w
    b=b-learning_rate*(b_gradient/k) # updating b
    return w, b
```

10. A function get_result() is defined that simply calls the meanSquaredError function defined earlier, to calculate the final result once weights from each of the ranks are aggregated and mean is calculated.

```
    return w, b

def get_result(y_test, y_pred): # getting result
    print('Mean Squared Error :', meanSquaredError(y_test, y_pred)) # printing MSE
```

11. Now, the main SGD function is defined. This function takes a series of arguments particularly, train_data, x_test, y_test, learning_rate, epochs, k, diminish_lr. First, required lists are intialized, local_mse to track mse values for each epoch, the timeList captures time for each mse value calculated at each epoch. Next, traindata is converted into a dataframe for simplicity of using the pandas sampling function later in the loop. Next, w,b are initialized using the initialize_term() function as defined above. In the for loop that runs as much as the epochs, first the data temp captures the k samples from the train_data, features and target variables are extracted as numpy arrays. W_gradient and b_gradient are initialized along with w and b. Next, the learning rate is factored at each epoch to improve the convergence, it is reached the global minima and the step size will become smaller to avoid divergence, this is done by dividing it by the diminish_lr variable at each epoch. Next, a prediction at each epoch is made and the mse is calculated and stored in the local_mse while the time is stored in timeList. At the end of this loop, the final calculated w and b are returned along with mse and time list for plotting purposes.

```
def SGD(train_data, x_test, y_test, learning_rate, epochs, k, diminish_lr): # SGD
    local_mse = [] # local mse
    timeList = [] # time list
    train_data = pd.DataFrame(train_data) # converting train_data to dataframe
    w, b = initialize_terms(train_data.shape[1]) # initializing terms
    for _ in range(epochs): # looping over epochs
        temp = train_data.sample(k).to_numpy() # sampling k samples
        y = temp[:, -1] # target variable
        x = temp[:,:(temp.shape[1]-1)] # features
        w_gradient=np.zeros(shape=(1,train_data.shape[1]-1)) # initializing w_gradient
        b_gradient=0 # initializing b_gradient
        w, b = update_terms(learning_rate, k, x, y, w, b, w_gradient, b_gradient) # updating terms
        learning_rate=learning_rate/diminish_lr # diminishing learning rate
        y_pred_at_epochs = predict(x_test,w,b)  # predicting the test data
        local_mse.append(mean_squared_error(y_pred_at_epochs, y_test)) # appending local mse
        timeList.append(MPI.Wtime()) # appending time
    return w,b, local_mse, timeList # returning w,b, local_mse, timeList
```

12. A predict function is defined to predict the values from the calculated parameters. It requires the test data and the w and b parameters. It then goes over each record in the test set and generates a prediction through y = wX + b, and stores the prediction in an array and returns it.

```
def predict(x,w,b): # predicting
    y_pred=[] # initializing y_pred
    for i in range(len(x)): # looping over x
        y_pred.append(np.asscalar(np.dot(w,x[i])+b)) # appending y_pred
    return np.array(y_pred) # returning y_pred
```

13. Now that we have defined all the functions, we can look at the main process. Root rank, calls the get_data() function to obtain data, converts the train_data into a numpy array in order to enable it to be scattered among ranks by splitting it. Splits the data and stores it in sdata variable. Next, the test set is also split and stored into sx_test and sy_test, it is not a necessary step, it could be taken whole and broadcasted but due to its size and my system's performance I decided to use a small test set for each rank to compute on. I tried the full test set as well and it did not make any significant difference in the convergence of the algorithm hence I decided to move forward with the smaller size of test set.

```
if rank == root: # root process
    x_train,x_test,y_train,y_test, train_data = get_data() # getting data
    # sklearn_result(x_train, y_train, y_test, x_test) # sklearn result
    train_data = train_data.to_numpy() # converting train_data to numpy array
    sdata = np.array_split(train_data, size) # splitting data
    sx_test = np.array_split(x_test, size) # splitting x_test
    sy_test = np.array_split(y_test, size) # splitting y_test
    net_time = []
else:
    x_train = None
    x_test = None
    y_train = None
    y_test = None
    train_data = None
    sdata = None
    sx_test = None
    sy_test = None
    net_time = None
```

14. Once data is split as required, the tain_data splits are scattered and captured as sdata at each rank. Correspondingly the test sets are also captured as sx_test and sy_test. SGD function is called and w,b, mseList and timeList is obtained by each rank. Times and MSE are gathered to be plotted that will be discussed later. Important thing here is that the wand b are reduced and the root receives them as w1 and b1.

```
sdata = comm.scatter(sdata, root = root) # scattering data
sx_test = comm.scatter(sx_test, root = root)  # scattering x_test
sy_test = comm.scatter(sy_test, root = root) # scattering y_test
# sx_test = comm.bcast(x_test, root = root)
# sy_test = comm.bcast(y_test, root = root)
w,b, mseList, timeList = SGD(sdata,sx_test,sy_test,learning_rate=learning_rate,epochs=epochs,diminish_lr=diminish_lr,k=k) # SGD
times = comm.gather(timeList, root = root) # gathering timeList
mseGather = comm.gather(mseList, root = root) # gathering mseList
w1 = comm.reduce(w, root = root) # reducing w
b1 = comm.reduce(b, root = root) # reducing b
```

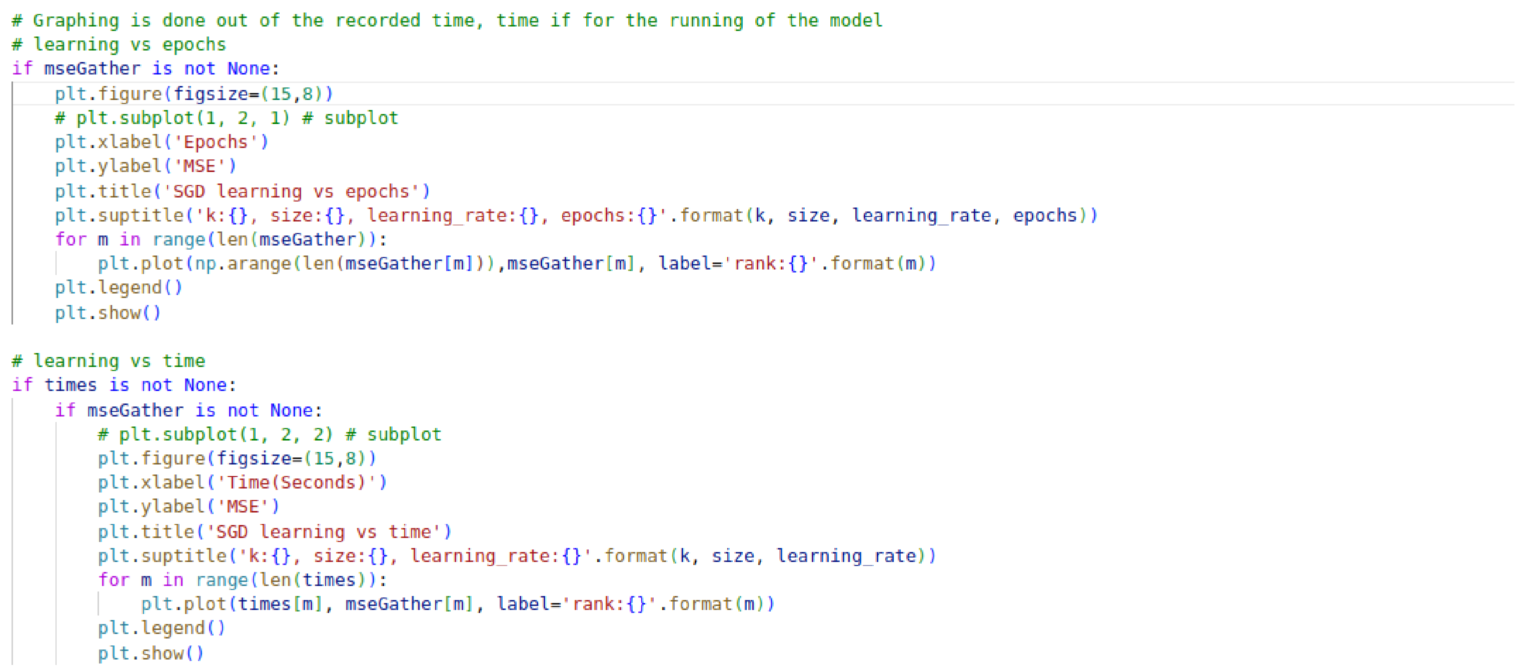
15. Root receives w1 and b1, takes a mean over the arrays and calls the predict function. Once the prediction list is obtained, it is sent to the get_result() function and it prints the MSE as the output which is the final result of this algorithm for the calculations.

```
if w1 is not None:
    if b1 is not None:
        w1 = w1/size # dividing w by size
        b1 = b1/size # dividing b by size
        y_pred = predict(x_test,w1,b1) # predicting
        get_result(y_test, y_pred) # getting result
```

16. This part shows the time calculation for this algorithm. Time for each rank is calculated and then reduced, the total time from all ranks is obtained through reduce, this is averaged by the size and average time per each rank is obtained that is printed.

```
et = MPI.Wtime() # ending time
ttime = et-st # calculating time taken
# print('Time is {} for rank {}: '.format(rank, ttime)) # printing time
totaltime = comm.reduce(ttime, root = root) # reducing time
if totaltime is not None:
    print('Avg Total time is {}'.format(totaltime/size)) # printing avg time
```

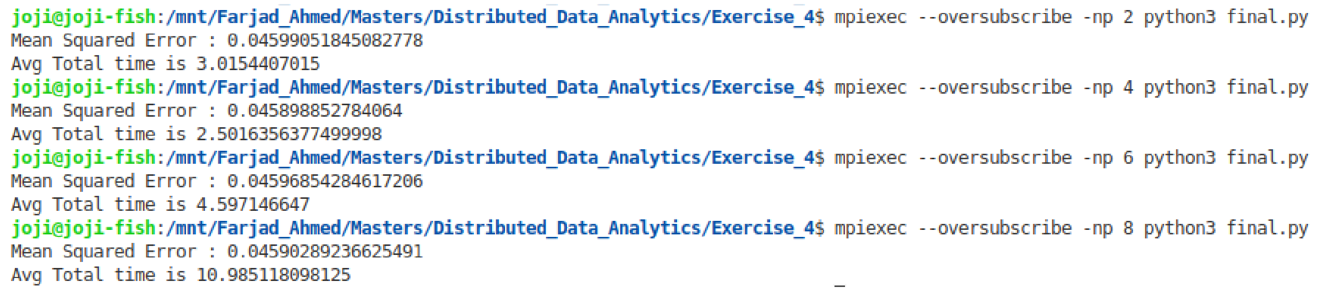
17. Two types of graphs are generated in this algorithm through the code, one is the MSE vs epochs, this is done by taking the timeList from the SGD algorithm that was updated at epoch and the mseList that was also updated similarly. The mse is plotted once with the number of epochs, for each rank on a single graph. Next, it is plotted with the time values for each epoch and the graphs shown earlier are generated by these snippets of code.



```python
# Graphing is done out of the recorded time, time if for the running of the model
# learning vs epochs
if mseGather is not None:
    plt.figure(figsize=(15,8))
    # plt.subplot(1, 2, 1) # subplot
    plt.xlabel('Epochs')
    plt.ylabel('MSE')
    plt.title('SGD learning vs epochs')
    plt.suptitle('k:{}, size:{}, learning_rate:{}, epochs:{}'.format(k, size, learning_rate, epochs))
    for m in range(len(mseGather)):
        plt.plot(np.arange(len(mseGather[m])),mseGather[m], label='rank:{}'.format(m))
    plt.legend()
    plt.show()

# learning vs time
if times is not None:
    if mseGather is not None:
        # plt.subplot(1, 2, 2) # subplot
        plt.figure(figsize=(15,8))
        plt.xlabel('Time(Seconds)')
        plt.ylabel('MSE')
        plt.title('SGD learning vs time')
        plt.suptitle('k:{}, size:{}, learning_rate:{}'.format(k, size, learning_rate))
        for m in range(len(times)):
            plt.plot(times[m], mseGather[m], label='rank:{}'.format(m))
        plt.legend()
        plt.show()
```

Question 1: Parallel Linear Regression (10 points)
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 thought 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 thought 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}.



```
joji@joji-fish:/mnt/Farjad_Ahmed/Masters/Distributed_Data_Analytics/Exercise_4$ mpiexec --oversubscribe -np 2 python3 final.py
Mean Squared Error : 0.04599051845082778
Avg Total time is 3.0154407015
joji@joji-fish:/mnt/Farjad_Ahmed/Masters/Distributed_Data_Analytics/Exercise_4$ mpiexec --oversubscribe -np 4 python3 final.py
Mean Squared Error : 0.045898852784064
Avg Total time is 2.5016356377499998
joji@joji-fish:/mnt/Farjad_Ahmed/Masters/Distributed_Data_Analytics/Exercise_4$ mpiexec --oversubscribe -np 6 python3 final.py
Mean Squared Error : 0.04596854284617206
Avg Total time is 4.597146647
joji@joji-fish:/mnt/Farjad_Ahmed/Masters/Distributed_Data_Analytics/Exercise_4$ mpiexec --oversubscribe -np 8 python3 final.py
Mean Squared Error : 0.04590289236625491
Avg Total time is 10.985118098125
```

Question 2: Performance and Convergence of PSGD (10 points)

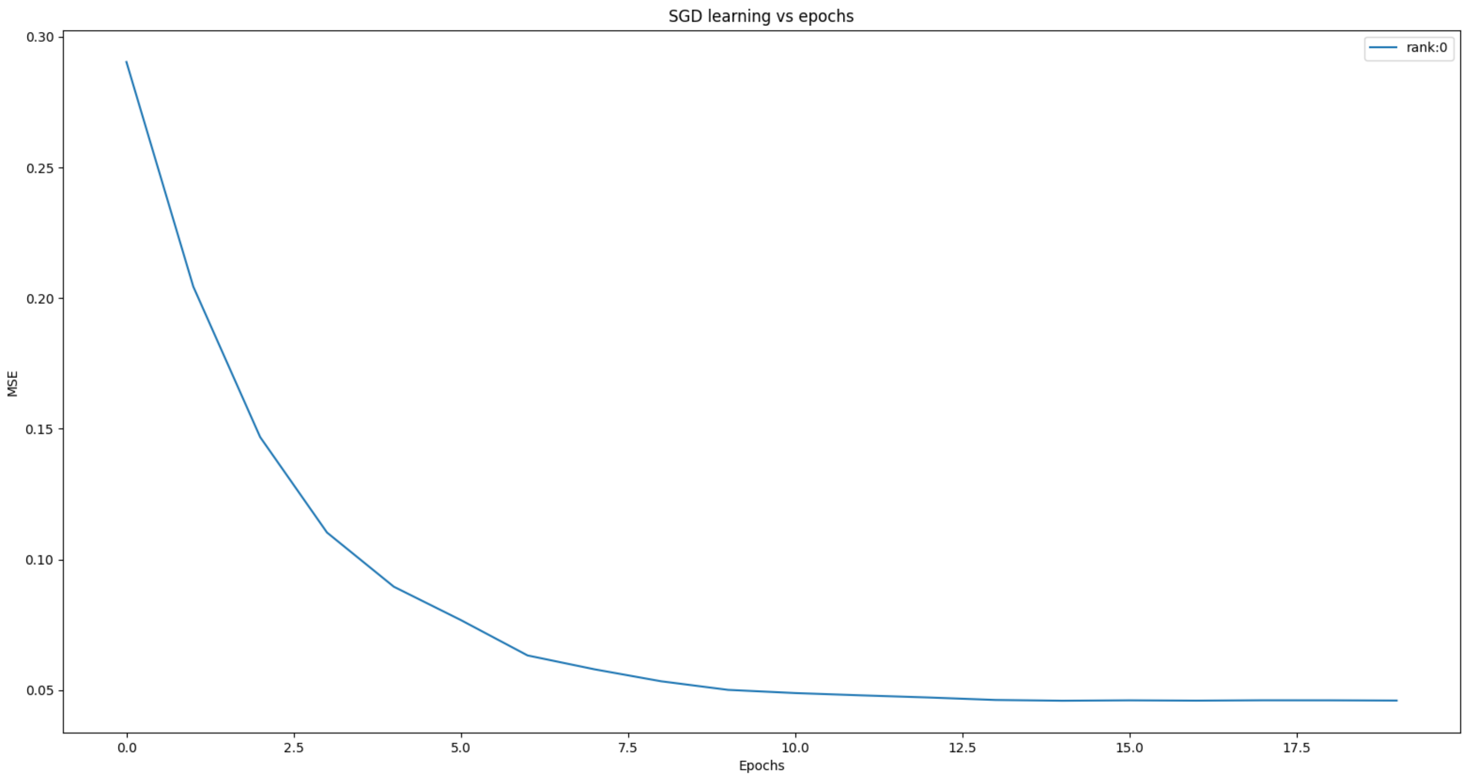
The second task is to do some performance analysis and convergence tests.

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 versions you can use sklearn).

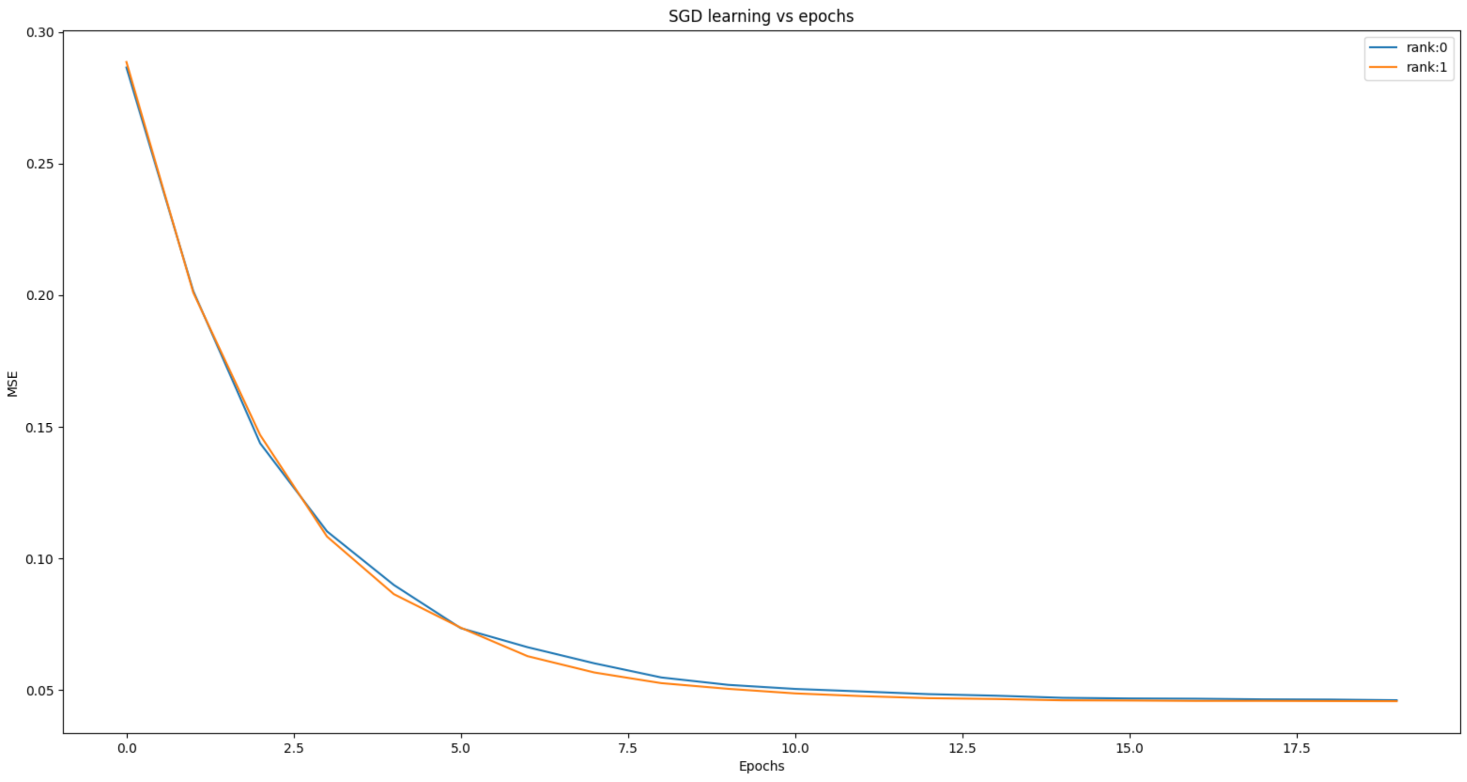
For benchmark and sequential results sklearn was used. The preprocessing of the data remains the same, just the model used was from sklearn. The result obtained was a mean squared error of **0.04591156625856042** from sklearn.
For my algorithm, P = 1 was run to obtain the convergence result, **0.04602546660362024** which agrees with the result obtained from Sklearn, in fact better than it by 0.001 due to fixed parameters in sklearn whereas I did a lot more experimentation with my algorithm to improve the performance.

Curves for MSE vs Epochs for P = {1, 2, 4, 6, 7}

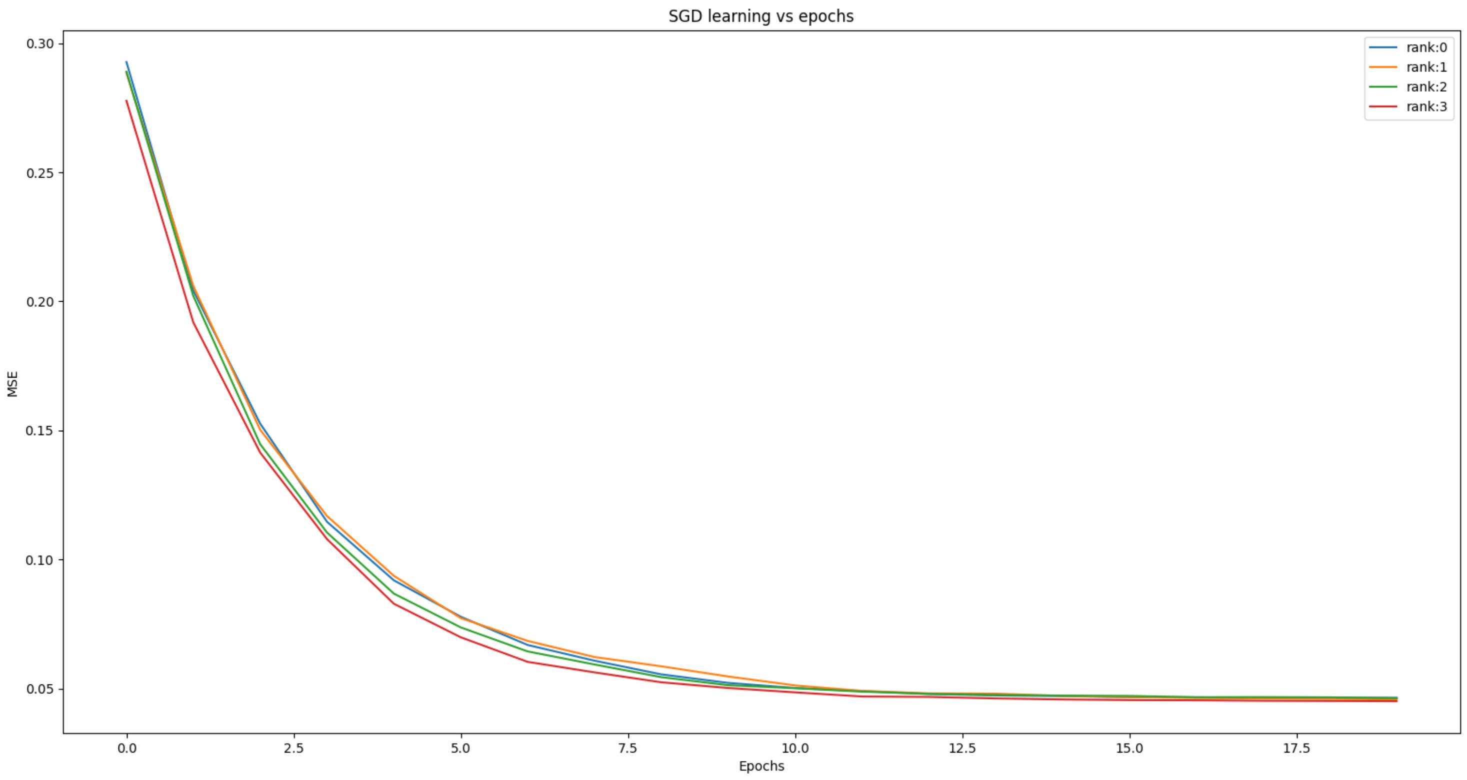
k:100, size:1, learning_rate:0.1, epochs:20



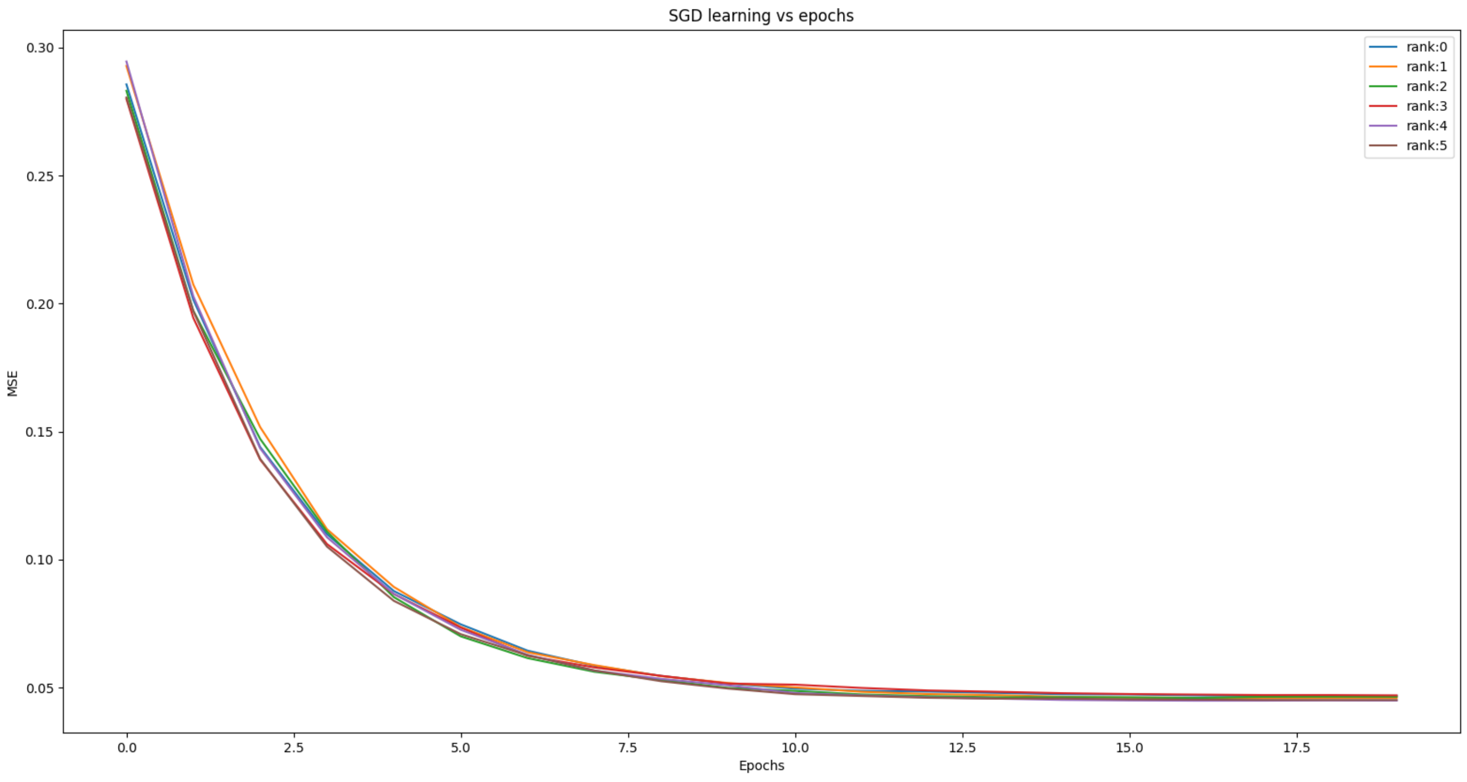
k:100, size:2, learning_rate:0.1, epochs:20

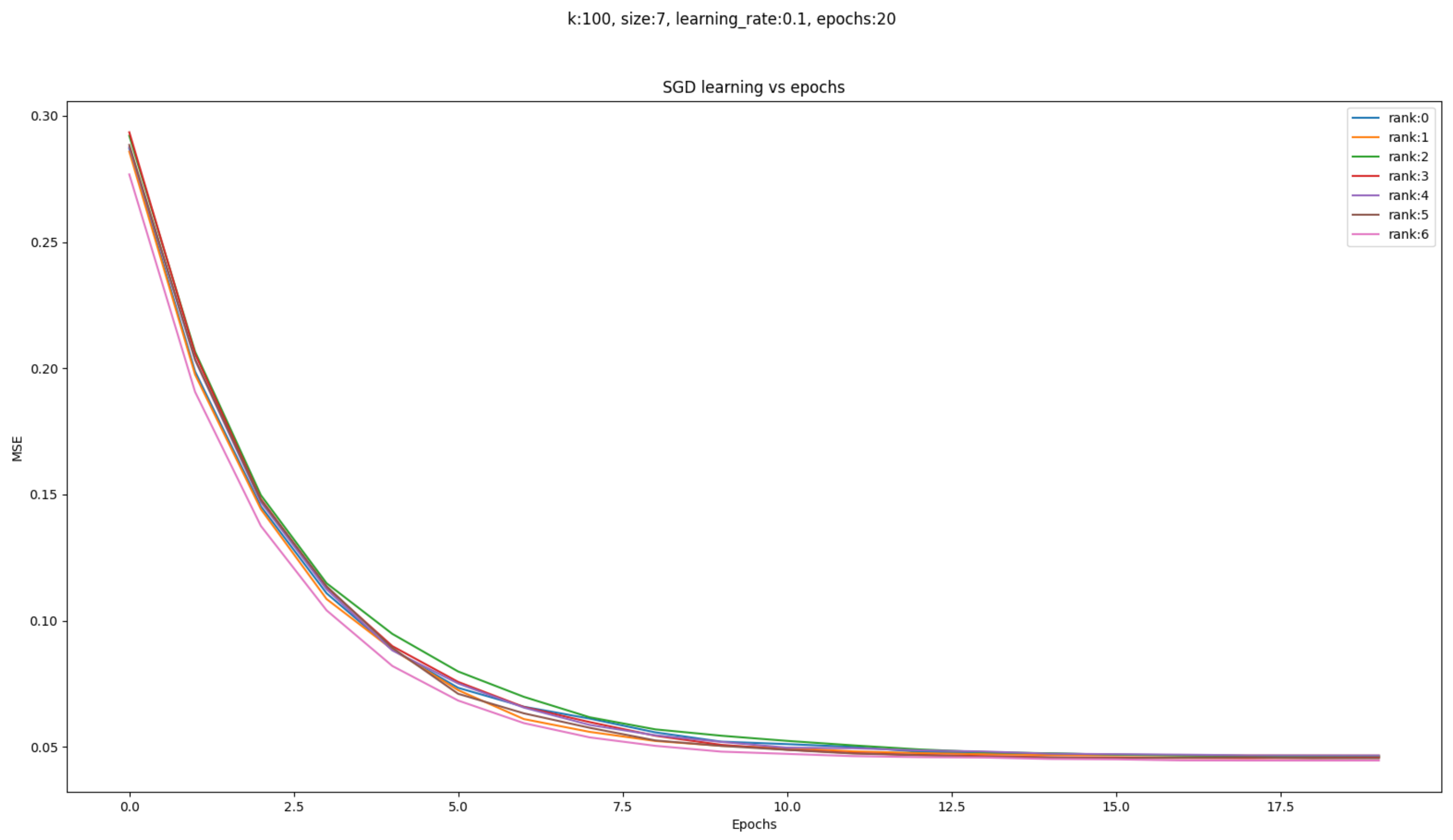


k:100, size:4, learning_rate:0.1, epochs:20



k:100, size:6, learning_rate:0.1, epochs:20

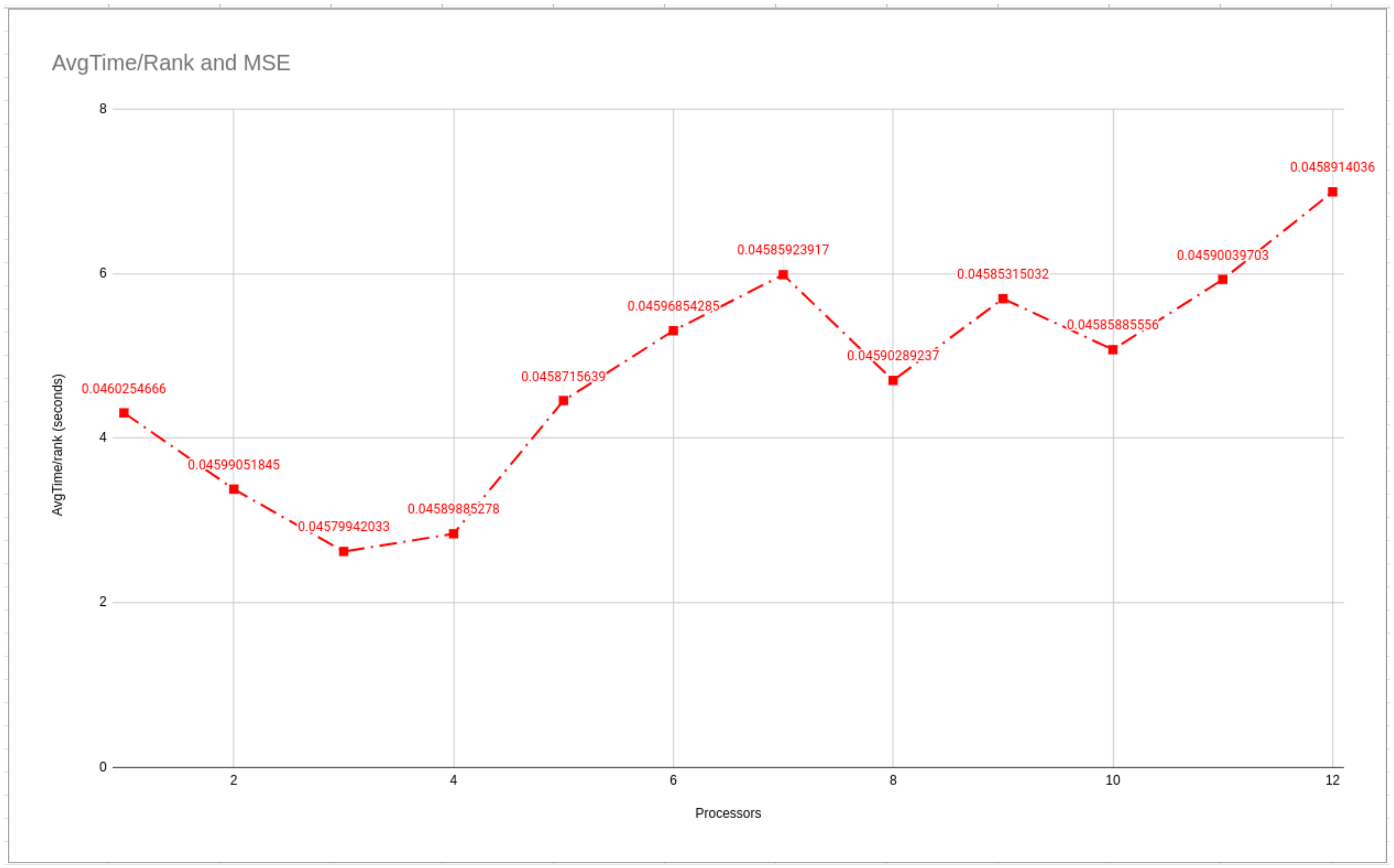




2. Second, you have to do a performance analysis by plotting learning curve (Train/Test scores) verses time. Time your program for P = {1, 2, 4, 6, 7}.

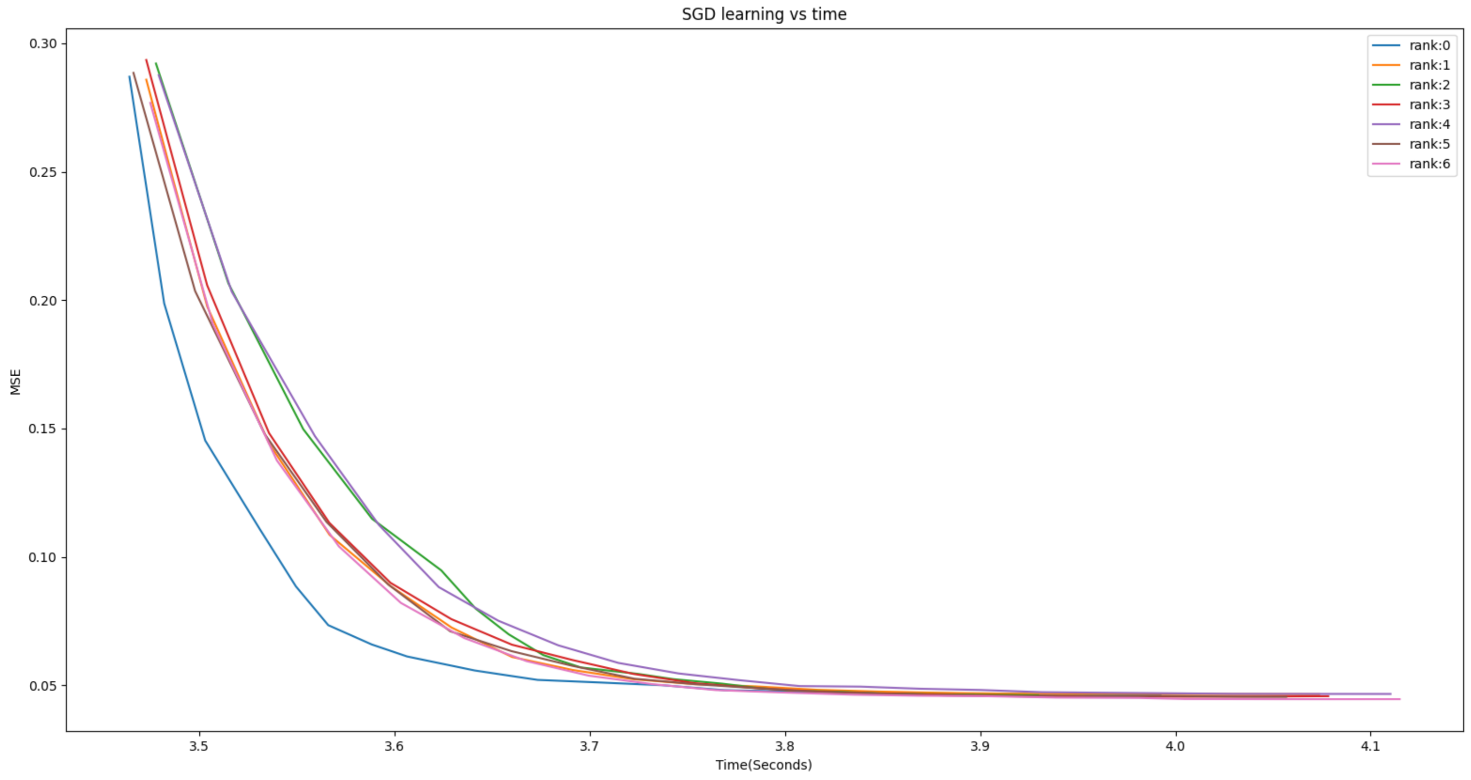
The train/test scores are shown below with the training times on the vertical axis and number of processors as the x-axis.
The interesting thing about this performance is that from 1 to 4 the performance improves as the process becomes more and more distributed, however after exceeding the number of processors from 4 the performance deteriorates. I have reviewed the code and the functioning, from what I understand is that since my system has 4 cores, so for parallel processes up to 4 cores the speed increases, however when I oversubscribe the processors there is suspected serialization or multiple threads within the same processor due to which the time increases.

Processors	AvgTime/Rank	MSE
1	4.308355828	0.0460254666
2	3.380647484	0.04599051845
3	2.622736641	0.04579942033
4	2.83869646	0.04589885278
5	4.458035051	0.0458715639
6	5.30784563	0.04596854285
7	5.988681451	0.04585923917
8	4.702970819	0.04590289237
9	5.697409738	0.04585315032
10	5.07701828	0.04585885556
11	5.93137755	0.04590039703
12	6.99612632	0.0458914036



Learning of Processors vs Time Each Processor takes. This is done directly for 7 processes as that captures data collectively for all processors in a single graph.

k:100, size:7, learning_rate:0.1


SGD learning vs time
MSE
Time(Seconds)
rank:0
rank:1
rank:2
rank:3
rank:4
rank:5
rank:6