Mansoor Nabawi, #309498

May 27, 2022

Strategy (same for both datasets):

Master node, rank==0 is
- loading the datasets
- Normalizing if needed,
- Splitting data into train and test,
- We'll concatenate y_train and x_train in one array so it is easier to send to other workers,

A fixed seed number is considered to provide the same random number every time.
We make random numbers for weights in a specific range and size depending on the dataset columns, e.g range = 0,5 or size = (1,261).

Learning rate and a fixed number of epochs and lists for the losses are considered.

- Depending on the number of epochs we choose, we loop and check if it is converged or not.
    - In each epoch, when we are in the master process, the data is splitted by the number of workers we have .
    - Each data part is sent via scatter,
    - But the weights is sent via broadcast as it is the same for all the workers.
    - In each worker the part of the data received is shuffled.
        - For each instance/row of the data received in each worker.
            - SGD is calculated.
            - Weights are updated.
        - After all the instance weights are updated.
        - Master node gathers all the weights, predicted y and true y.
- Master node finds the mean of the weights, calculates the RMSE and append it to our list for train and it uses new mean weights to predict new values for test data and computes the RMSE of the true test target and predicted ones to append it to out test loss list.
- If the test oss of each instance is less than a specific value then it is converged, it means it can't improve more.
    - If it is then broadcast the converge value so the program stops.
- And finally we save the losses and show the final time.

Virus dataset.

The STRATEGY is the same as above.

For this dataset, we define load_virus() functions which goes into the dataset directory and list all of them into a list called files.

We go through each file and append them in another list, later we concat them to make a bigger dataframe.

The first value of each input is our target y, all the others, there is a key and a value pointing to a feature and its values exist here, we make a sparse matrix and go through each file and split them the first value is the column name and the second one is the value of that column.

We make ones as bias and add them to x and we have our final datasets.

```
def load_virus():
    #freeing up the memory
    gc.collect()
    path = 'dataset/dataset/'
    files = os.listdir(path)
    li = []
    for filename in files:
        df = pd.read_csv(path+filename, header=None)
        li.append(df)

    dat_file = pd.concat(li, axis=0, ignore_index=True)

    y = np.zeros((107856,1), float)
    x = np.zeros((107856,479))
    for i in dat_file.index:
        doc = dat_file.iloc[i][0].strip().split()
        y[i] = float(doc[0])

        for element in doc[1:]:
            k, v = element.split(":")
            x[i,int(k)] = int(v)
    bias = np.ones((107856,1))
    x = np.append(bias, x, axis=1)

    return x,y
```

The rest of the functions are as below.

- train_test_split() makes data into two splits by chosen fraction.
- row_normalizer(), normalize each row.
- predic(), predict new y based on x and weights received.
- sgd(), calculates the main part of our program, stochastic gradient descent.
- calculate_RMSE(), calculate RMSE between prediction and true values.

```
#splitting dataset
def train_test_split(data ,frac=0.7):
    data = pd.DataFrame(data)
    #train_test split
    train = data.sample(frac=frac, random_state=10)
    test = data.drop(train.index)

    return train.to_numpy(), test.to_numpy()

#normalzining based on rows
def row_normalizer(x: np.ndarray):
    return x/np.linalg.norm(x, ord=2, axis=1, keepdims=True)

#prediciton
def predict(x, betas):
    return np.matmul(x, betas.T)

#SGD function
def sgd(x, weight, y, y_hat, lr):

    #Computing Derivatives using Chain Rule
    dv_loss = x * (-2*(y - y_hat) )
    #Computing new weights
    weight = weight - lr * dv_loss
    return weight

#calculate_RMSE
def calculate_RMSE(true_y, pred_y):
    return np.sqrt(mt.mean_squared_error(true_y, pred_y))

```

Starting from the master node.

```
if rank == 0:
    print('Loading data...', flush=True)
    x,y = load_virus()
    print(f"X shape: {x.shape}")
    x = normalize_rows(x)
    print('Splitting into train and test sets...', flush=True)
    x_train, x_test = train_test_split(x ,frac=0.7)
    y_train, y_test = train_test_split(y ,frac=0.7)

    data = np.concatenate((y_train.reshape(-1,1), x_train), axis=1)


#Initializing weights

w = np.random.randint(0,2,size=((1,480)))

#learning rate
lr = 0.00001

#Setting the number of epochs
N = 100
train_loss = []
test_loss = []

converged = False #For checking convergence
epoch = 0

if rank == 0:
    #Starting time
    t0 = MPI.Wtime()
```

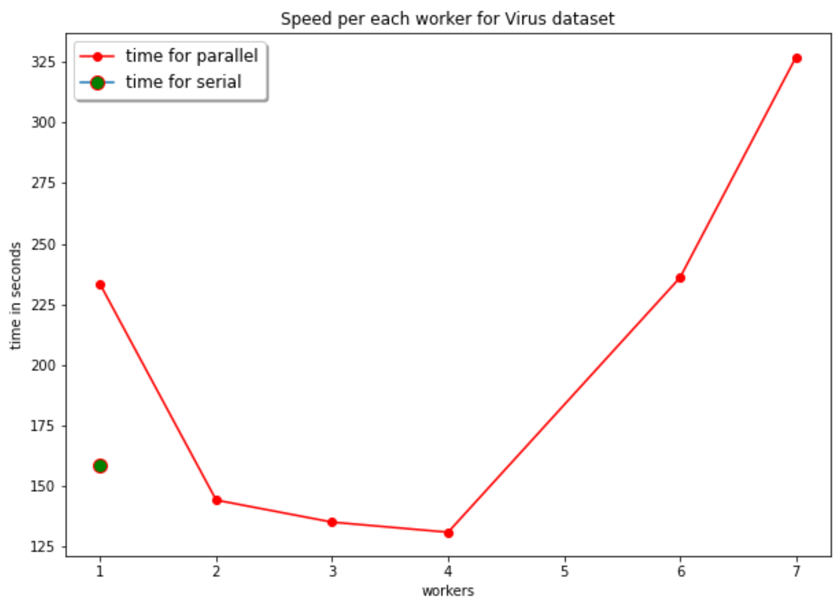
```python
#Training for N Epochs
while (not converged) and (epoch < N):

    if rank == 0:
        #print onlt every 10 steps
        if epoch%10==0:
            print("Epoch : ", epoch, flush=True)

        #splitting
        data_part = np.array_split(data,size)
    else:
        data_part = None
    #Sending data using scatter and weights via bcast
    recvd_dt = comm.scatter(data_part,root=0)
    weights = comm.bcast(w,root=0)
    row_, col_ = np.shape(recvd_dt)
    y_hat = np.zeros((row_,1))

    #Shuffling the received dataset at each worker
    np.take(recvd_dt,np.random.permutation(recvd_dt.shape[0]),axis=0,out=recvd_dt)
    #Epoch
    for i in range(0, row_):
        X_part = recvd_dt[i,1:]
        y_part = recvd_dt[i,0]
        #Prediction of instance i, forward pass
        y_hat[i,0] = predict(X_part, weights.T)

        #Computing new weights
        weights = sgd(X_part, weights, y_part, y_hat[i,0], lr)

    #Gathering weights at Worker 0
    w_r = comm.gather(weights, root=0)
    #Gathering predicted output (yy) and original data (true_dt) at Worker 0
    y_hat_recvd = comm.gather(y_hat, root=0)
    true_dt = comm.gather(recvd_dt[:,0], root=0)

    if rank == 0 and converged == False:
        #Worker 0 computing mean of the weights
        w = np.mean(w_r, axis=0)

        pred_y = np.vstack(y_hat_recvd)
        true_y = np.hstack(true_dt)

        #Worker 0 computing calculate_RMSE
        calculate_RMSE_train = calculate_RMSE(true_y, pred_y)
        train_loss.append(calculate_RMSE_train)

        #test prediction
        pred_y_test = predict(x_test, w.T)
        #Computing RME Testing
        calculate_RMSE_test = calculate_RMSE(y_test, pred_y_test)
        test_loss.append(calculate_RMSE_test)

        #Checking for convergence in RMSE test
        if epoch > 0 and abs((test_loss[epoch-1] - test_loss[epoch])) < 10**-6:
            print(test_loss[epoch-1]- test_loss[epoch], flush=True)
            converged = True
    #the converged flag is sent back for stopping epochs
    converged = comm.bcast(converged,root=0)
    epoch += 1

if rank == 0:
    #Ending time
    t1 = MPI.Wtime()
    df = pd.DataFrame(list(zip(train_loss, test_loss)),columns =['train_loss', 'test_loss'])
    df.to_csv(f"loss_virus{size}.csv", index=False)
    #t1 = MPI.Wtime()
    time_final = t1 - t0
    print("With P = {} Workers, the process took {} seconds" .format(size, time_final), flush=True)
```

Here is the screenshot of running with 4 workers.

```
!mpirun -n 4 python virus.py
```

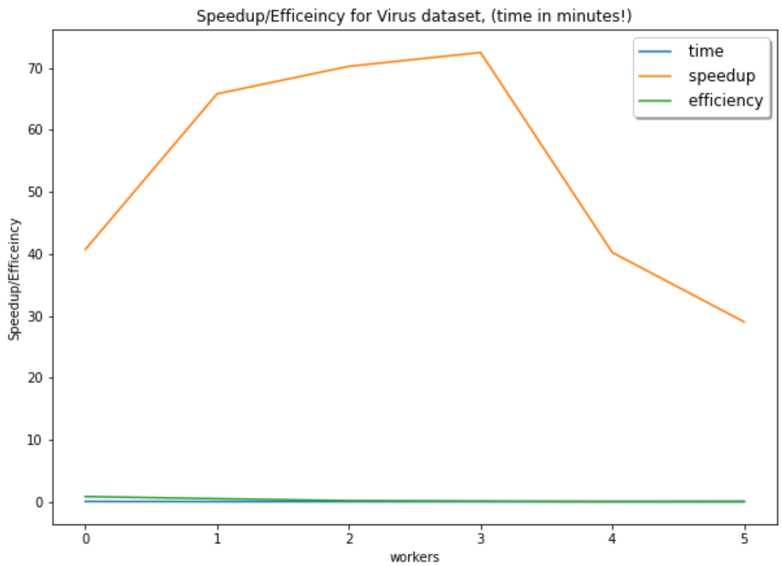
```
Loading data...
X shape: (107856, 480)
Splitting into train and test sets...
Epoch :  0
Epoch :  10
Epoch :  20
Epoch :  30
Epoch :  40
Epoch :  50
Epoch :  60
Epoch :  70
Epoch :  80
Epoch :  90
With P = 4 Workers, the process took 130.96906805038452 seconds
```

Let’s look at the time elapsed using different workers.



Looking at the graph we see that the sequential code is faster than parallel code using only one worker. Using 2 workers is definitely faster than one and so until 4th workers.
Using 6th and more is not fast anymore as my machine has only 4 cores.

Let’s look at the speedup/efficiency graph.



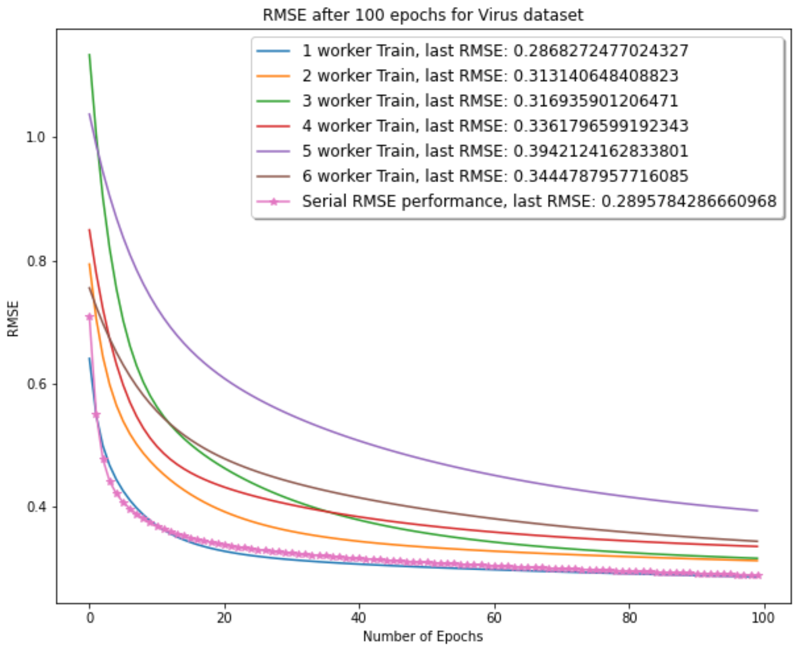
Looking at the above graph we see that we have a huge speed up until using 3 workers and then the speed goes down. The efficiency overall goes down. But more details are observed in this table.

```
: 1 time_virus
```

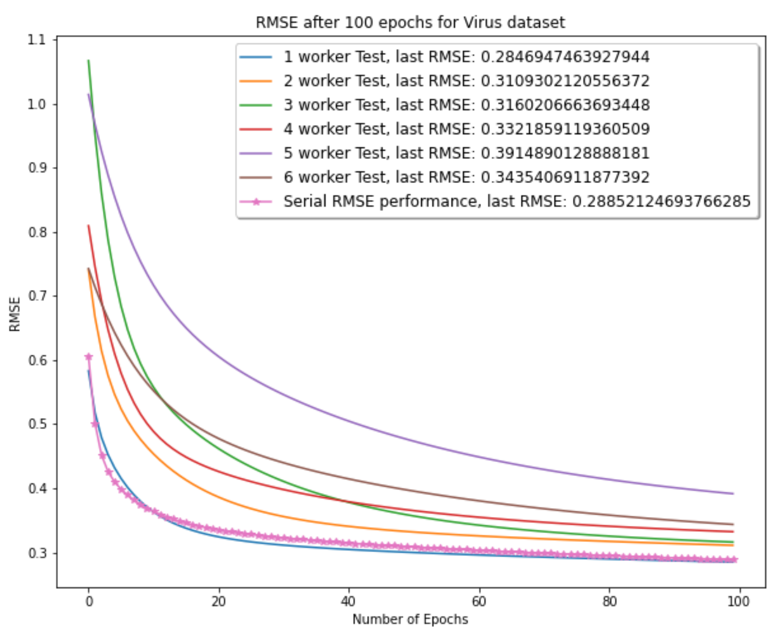
:

	workers	time	speedup	efficiency
0	1	0.064786	40.718604	0.848725
1	2	0.040067	65.840266	0.504409
2	3	0.037536	70.278991	0.186993
3	4	0.036380	72.511816	0.113861
4	6	0.065559	40.238802	0.027713
5	7	0.090825	29.044867	0.019868

RMSE Performance.
Looking at the graph the RMSE is going down and we have a good accuracy.
But still sequential performance and using one worker is giving us better accuracy than more workers.



Test RMSE Observation:



Even in the test set we see better performance in serial code and using one core, also 3 workers give us a good RMSE score.

- Using one worker it didn't converge but we have a good RMSE score.

- 2 workers did not converge, RMSE is worse than using one worker.
- 3 workers did not converge, RMSE is worse than using 1/2 workers.
- 4 workers did not converge, RMSE is worse than before.
- 6 workers did not converge, RMSE is worse than before.
- 7 workers did not converge, RMSE is worse than before.

KDD dataset.

For this data set to save time I first concat the three datasets for learning and validation and validation target in one data frame.
Replaced " " with 0 and applied one hot encoding for features which are categorical plus the columns that have less than 5 unique values, they are converted to categorical features and one hot encoding applied.
Next I checked the Pearson correlation and those which had positive correlation were chosen.
Finally we have a data set with shape (191779,261) bias added.

—---
First we define some functions:

- load_data_kdd() to load csv file we already cleaned and preprocessed, it adds bias to it and create x and y matrix.
- train_test_split() makes data into two splits by chosen fraction.
- row_normalizer(), normalize each row.
- predic(), predict new y based on x and weights received.
- sgd(), calculates the main part of our program, stochastic gradient descent.
- calculate_RMSE(), calculate RMSE between prediction and true values.

```python
#reading dataset
def load_data_kdd(data = "kdd_final_filtered.csv"):
    #freeing up the memory
    gc.collect()
    #reading the saved dataset
    kdd = pd.read_csv(data, low_memory=False)
    #target
    y = kdd["TARGET_D"].to_numpy()
    x = kdd.drop(columns=["TARGET_D"], axis=1)

    bias = pd.DataFrame(np.ones(x.shape[0]).reshape(-1,1), columns = ["bias"])
    x = pd.concat([bias, x], axis=1).to_numpy()

    return x, y

#splitting dataset
def train_test_split(data ,frac=0.7):
    data = pd.DataFrame(data)
    #train_test split
    train = data.sample(frac=frac, random_state=10)
    test = data.drop(train.index)

    return train.to_numpy(), test.to_numpy()

#normalzining based on rows
def row_normalizer(x: np.ndarray):
    return x/np.linalg.norm(x, ord=2, axis=1, keepdims=True)

#prediciton
def predict(x, betas):
    return np.matmul(x, betas.T)

#SGD function
def sgd(x, weight, y, y_hat, lr):

    #Computing Derivatives using Chain Rule
    dv_loss = x * (-2*(y - y_hat) )
    #Computing new weights
    weight = weight - lr * dv_loss

    return weight
#calculate_RMSE
def calculate_RMSE(true_y, pred_y):
    return np.sqrt(mt.mean_squared_error(true_y, pred_y))

```

The rest of the code is the same as explained in the STRATEGY.

```python
if rank == 0:
    print('Loading data...', flush=True)
    x,y = load_data_kdd()
    print(f"X shape: {x.shape}")
    #print("Normalizing")

    x = row_normalizer(x)

    print('Splitting into train and test sets...', flush=True)
    x_train, x_test = train_test_split(x ,frac=0.7)
    y_train, y_test = train_test_split(y ,frac=0.7)

    data = np.concatenate((y_train.reshape(-1,1), x_train), axis=1)


#fixing seed
np.random.seed(2021)
#random weights
w = np.random.randint(0,5,size=((1,261)))

#learning rate
lr = 0.00001

#Setting the number of epochs
N = 100
train_loss = []
test_loss = []

converged = False #For checking convergence
epoch = 0

if rank == 0:
    #Starting time
    t0 = MPI.Wtime()
```

```python
while (not converged) and (epoch < N):

    if rank == 0:
        #print onlt every 10 steps
        if epoch%10==0:
            print("Epoch : ", epoch, flush=True)

        #splitting
        data_part = np.array_split(data,size)
    else:
        data_part = None
    #Sending data using scatter and weights via bcast
    recvd_dt = comm.scatter(data_part,root=0)
    weights = comm.bcast(w,root=0)
    row_, col_ = np.shape(recvd_dt)
    y_hat = np.zeros((row_,1))

    #Shuffling the received dataset at each worker
    np.take(recvd_dt,np.random.permutation(recvd_dt.shape[0]),axis=0,out=recvd_dt)
    #Epoch
    for i in range(0, row_):
        X_part = recvd_dt[i,1:]
        y_part = recvd_dt[i,0]
        #Prediction of instance i, forward pass
        y_hat[i,0] = predict(X_part, weights.T)

        #Computing new weights
        weights = sgd(X_part, weights, y_part, y_hat[i,0], lr)

    #Gathering weights at Worker 0
    w_r = comm.gather(weights, root=0)
    #Gathering predicted output (yy) and original data (true_dt) at Worker 0
    y_hat_recvd = comm.gather(y_hat, root=0)
    true_dt = comm.gather(recvd_dt[:,0], root=0)
```

```
    if rank == 0 and converged == False:
        #Worker 0 computing mean of the weights
        w = np.mean(w_r, axis=0)

        pred_y = np.vstack(y_hat_recvd)
        true_y = np.hstack(true_dt)

        #Worker 0 computing calculate_RMSE
        calculate_RMSE_train = calculate_RMSE(true_y, pred_y)
        train_loss.append(calculate_RMSE_train)

        #test prediction
        pred_y_test = predict(x_test, w.T)
        #Computing RME Testing
        calculate_RMSE_test = calculate_RMSE(y_test, pred_y_test)
        test_loss.append(calculate_RMSE_test)

        #Checking for convergence in RMSE test
        if epoch > 0 and abs((test_loss[epoch-1] - test_loss[epoch])) < 10**-6:
            print(test_loss[epoch-1]- test_loss[epoch], flush=True)
            converged = True
    #the converged flag is sent back for stopping epochs
    converged = comm.bcast(converged,root=0)
    epoch += 1

if rank == 0:
    #End time
    t1 = MPI.Wtime()
    df = pd.DataFrame(list(zip(train_loss, test_loss)),columns =['train_loss', 'test_loss'])
    df.to_csv(f"loss{size}.csv", index=False)
    #t1 = MPI.Wtime()
    time_final = t1 - t0
    print("With P = {} Workers, the process took {} seconds" .format(size, time_final), flush=True)
```

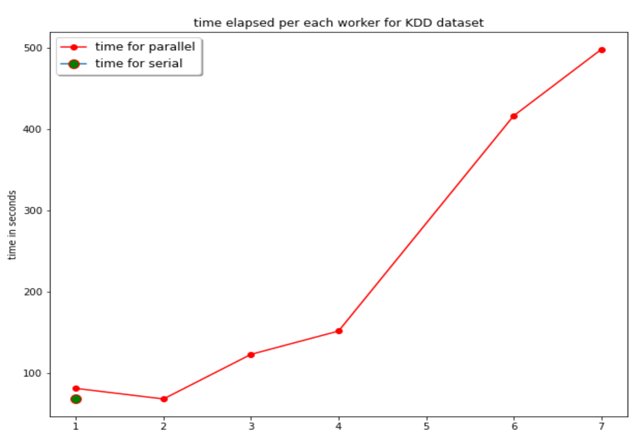
Running the code.

```
!mpiexec -n 2 python kdd_.py
```

```
Loading data...
X shape: (191779, 261)
Splitting into train and test sets...
Epoch :  0
Epoch :  10
Epoch :  20
8.017752826106062e-07
With P = 2 Workers, the process took 68.61729502677917 seconds
```

–

Looking at the time elapsed for each time with different workers.



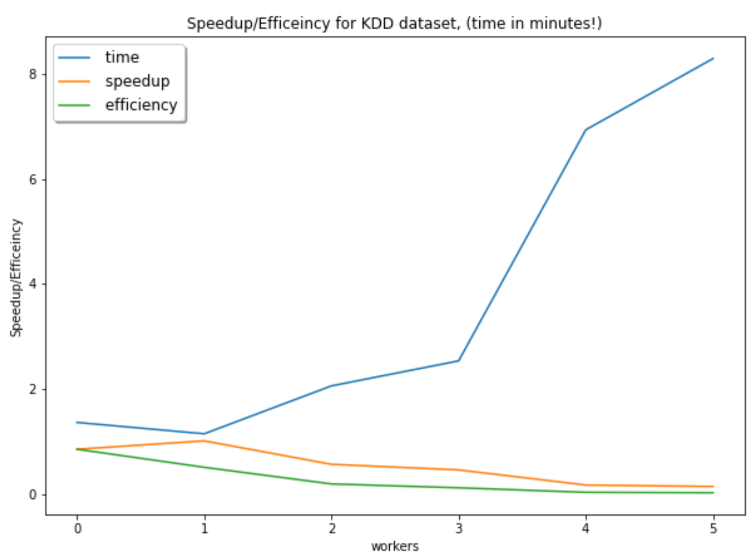
- Looking at this graph we see that using 2 workers gave us the quickest run time, as my machine has only 4 cores it is understandable why it takes longer time for more workers.
- Below is the total time, speedup and efficiency for this program.

	workers	time	speedup	efficiency
0	1	1.359333	0.848725	0.848725
1	2	1.143617	1.008817	0.504409
2	3	2.056583	0.560979	0.186993
3	4	2.533133	0.455444	0.113861
4	6	6.938417	0.166277	0.027713
5	7	8.295667	0.139073	0.019868

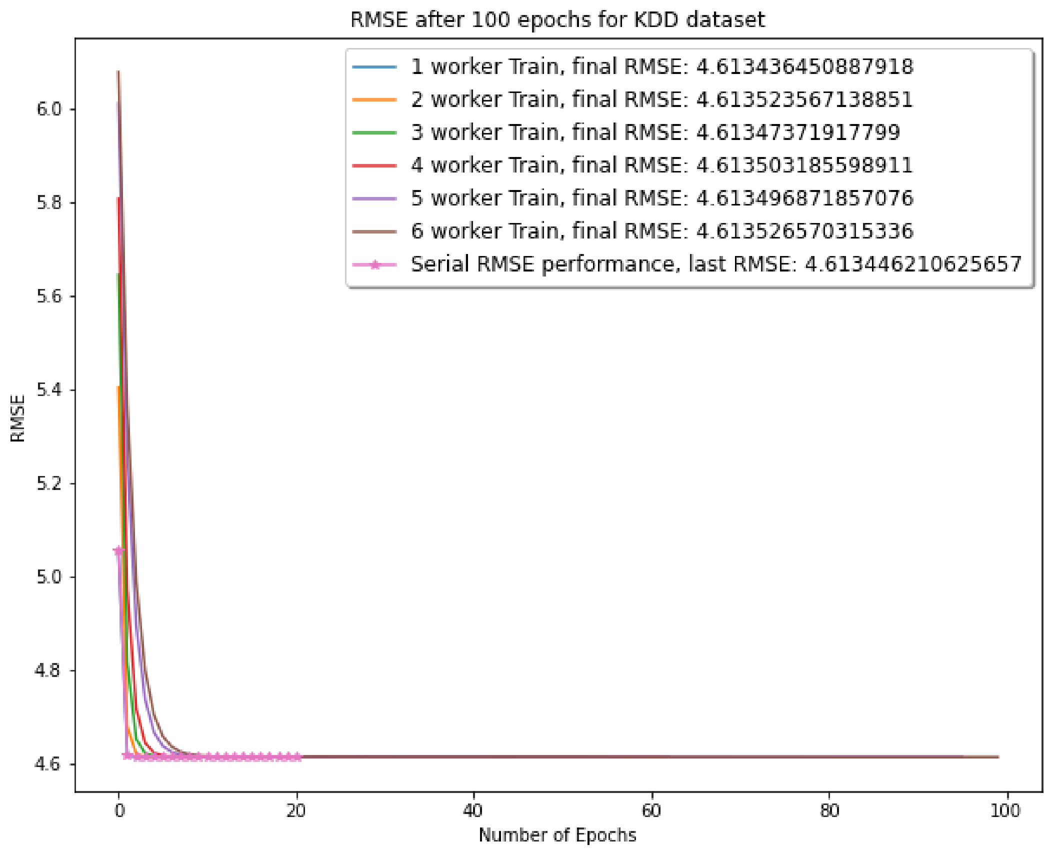
-
- Below is the graph for speedup and efficiency , time is divided by 60 seconds so we can see more details.
- The efficiency overall goes down with the number of processes increasing! I don’t know why this happens, the time for the serial program is 69.222 here how I calculated the speedup and efficiency.
- The speedup increase and later it decreases.

```
time_["speedup"] = (69.222)/time_.time
time_["efficiency"] = time_["speedup"]/time_.workers
```

-



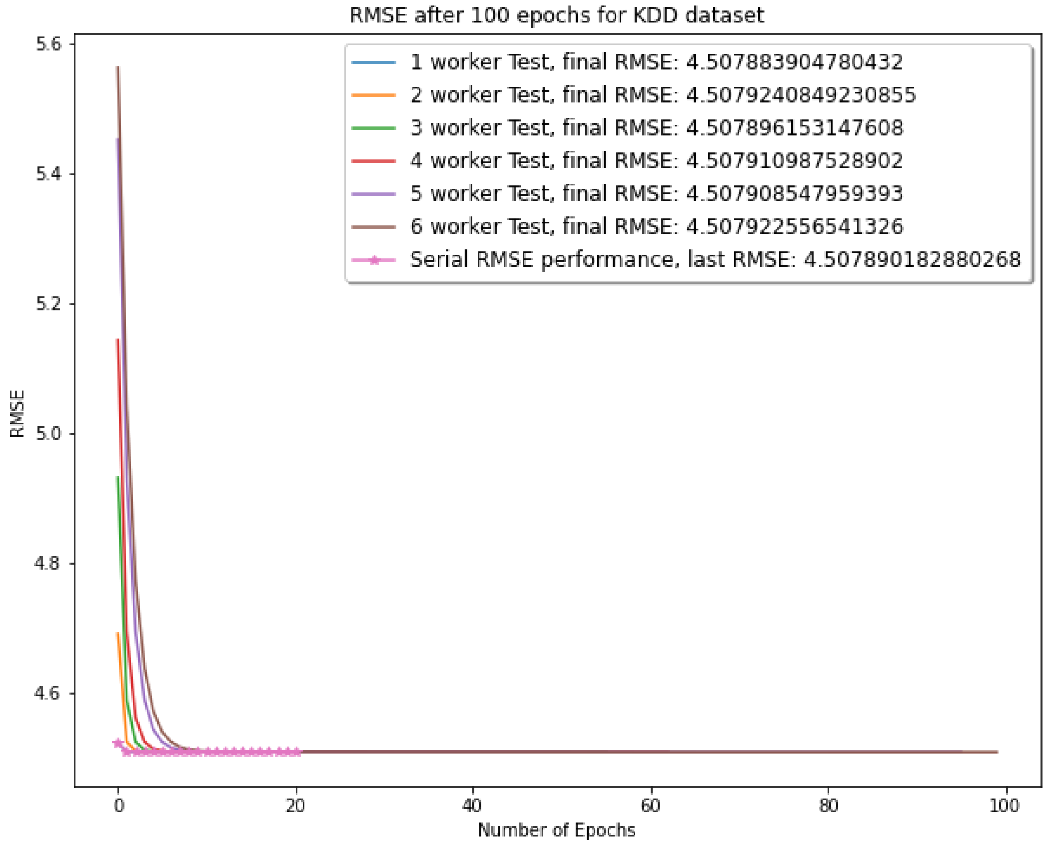
Looking at RMSE result for train and test sets.



- Using one worker it converged at the 23rd epoch.
- Using 2 workers it converged at the 30th epoch.
- Using 3 workers it converged at the 56th epoch
- Using 4 workers it converged at the 63rd epoch.
- Using 6 workers it converged at the 96th epoch
- Using 7 workers it did not converge.

Overall the RMSE is going lower but i could not get a lower RMSE even training for longer epochs and different learning rates.

Below is the testing RMSE values for KDD dataset. The behavior is as the same as train set.



RMSE after 100 epochs for KDD dataset
1 worker Test, final RMSE: 4.507883904780432
2 worker Test, final RMSE: 4.5079240849230855
3 worker Test, final RMSE: 4.507896153147608
4 worker Test, final RMSE: 4.507910987528902
5 worker Test, final RMSE: 4.507908547959393
6 worker Test, final RMSE: 4.507922556541326
Serial RMSE performance, last RMSE: 4.507890182880268
RMSE
Number of Epochs