Lab Course: Distributed Data Analytics
Exercise Sheet 4
Group 2 - Monday

Submitted by: Sruthy Annie Santhosh, 312213

Topic:Distributed Machine Learning (Supervised)

- My System

  Hardware and Software SetUp:

Processor	M1 chip
Operating System	MacOS Monterey
Number of Cores	8
RAM	8 GB
Python version	3.8.8

  The code editor used in this sheet : VSCode and Jupyter Notebook (graph)
  The output images attached are screenshots from the terminal.

- Exercise 1: Parallel Linear Regression

  In this exercise we have to implement Linear Regression using Parallel Stochastic Gradient Descent learning method. I have used Collective communication here i.e, Scatter, Bcast and Gather methods are used for parallelism.

  Parallel stochastic gradient descent(PSGD) is a learning algorithm in a distributed setting. The master worker splits the data and sends each split to different workers. The workers then find the beta values for the given dataset using stochastic gradient descent and then send them back to the master. The

master then finds the average of all the beta values to get the new beta. Since I used collective communication, the master worker also participates in the work (finding local beta).

Stochastic Gradient Descent:

The Linear Regression model and the loss function used:

- Linear Regression model is given as $\hat{y}^n = \sum_{m=1}^{M} \beta_m x_m^n$
- Least square loss function is given as $l(x, y) = \sum_{n=1}^{N} (y^n - \hat{y}^n)^2$

The objective is to minimize the loss, hence we find the gradient: (here θ = **β**)

$$\nabla L(\boldsymbol{\theta}) = \frac{\partial}{\partial \boldsymbol{\theta}} \frac{1}{N} \left\| \mathbf{X}\boldsymbol{\theta} - \mathbf{y} \right\|_2^2 = \frac{1}{N} (-\mathbf{X})^{\mathrm{T}} 2(\mathbf{y} - \mathbf{X}\boldsymbol{\theta})$$

The new theta values are found based on the learning rate(η) and the gradient.

Gradient descent algorithm

1. $\boldsymbol{\theta}_0$ = randomInitialization()
2. for $i = 0,\ldots,i_{max}$:
3. $\quad \boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \eta \nabla L(\boldsymbol{\theta}_i)$
4. $\quad$ if $L(\boldsymbol{\theta}_i) - L(\boldsymbol{\theta}_{i+1}) < \epsilon$:
5. $\quad\quad$ return $\boldsymbol{\theta}_{i+1}$
6. raise Exception("Not converged in i_{max} iterations")

Convergence depends on the learning rate chosen. If it is too small, convergence can take many epochs, if it's too large, overshooting can occur and divergence happens.
Here I have tried for various learning rates and have finally fixed for : 0.0001

For Stochastic Gradient Descent, mini batches of data are taken and beta is found. This constitutes one epoch. Here I am taking each row in the dataset and updating beta accordingly. One epoch is over when beta is updated for all data points.

Here I have fixed the total number of epochs as 200.
The RMSE score is used for analysing the performance of the model

In this exercise sheet, two datasets are given.

FIRST DATASET

First I have taken: KDD Cup 1998 Data Data Set
https://archive.ics.uci.edu/ml/datasets/KDD+Cup+1998+ Data

This dataset contains 95412 rows and 481 columns. I have taken 'TARGET_D' as the target column.

1. First the data is read and pre-processed. All categorical values are converted to their numerical codes. Nan values are removed and data is shuffled. Then the target column is removed and placed in a different data frame.
2. Min-max normalization is performed on the data and then columns containing only nan values and redundant target columns are removed.

```
#Function for preprocessing the data
def preprocessing(df_kdd):
    #replacing categorical values
    for col in df_kdd.columns:
            df_kdd[col] = df_kdd[col].astype('category')
            df_kdd[col] = df_kdd[col].cat.codes

    df_kdd = df_kdd.dropna()
    #Taking random sample
    df_kdd = df_kdd.sample(frac = 1, random_state=311)
    #Taking target y
    df_kdd_x = df_kdd.drop(["TARGET_D"], axis=1)
    df_kdd_y = df_kdd[["TARGET_D"]]

    #Applying min-max normalization on x and y
    df_kdd_x=(df_kdd_x - df_kdd_x.min())/(df_kdd_x.max() - df_kdd_x.min())
    df_kdd_y = (df_kdd_y - df_kdd_y.min()) / (df_kdd_y.max() - df_kdd_y.min())


    print(df_kdd["TARGET_D"].value_counts())
    print(df_kdd["TARGET_D"].unique())
    '''''
    nan_values = df_kdd_x.isna()
    nan_columns = nan_values.any()

    columns_with_nan = df_kdd_x.columns[nan_columns].tolist()
    columns_with_nan
    '''''
    #Dropping unwanted target column, nan column
    df_kdd_x = df_kdd_x.drop(columns=['TARGET_B','CONTROLN','RFA_2R'], axis=1)
```

3. The data has to be split in 70: 30 ratio for training and testing. Training is done on the train data set. Then we compute the rmse scores for both train and test sets.

```
#Function to split data into 70% for training and 30 % for testing
def train_test_split(df_kdd_x,df_kdd_y):
    train_df_x = df_kdd_x.iloc[:int(95412*0.7) , :]#70% for training, 30% for testing
    test_df_x = df_kdd_x.drop(train_df_x.index)
    train_df_y = df_kdd_y.iloc[:int(95412*0.7) , :]#70% for training, 30% for testing
    test_df_y = df_kdd_y.drop(train_df_y.index)

    #Converting data frames to arrays
    y_train_df = np.array(train_df_y) # puttingthe target feature to y array
    x_train_df = np.array(train_df_x)

    y_test_df = np.array(test_df_y)
    x_test_df = np.array(test_df_x)


    print("Shape of x",x_train_df.shape)
    print("Shape of y",y_train_df.shape)

    return x_train_df, y_train_df, x_test_df, y_test_df
```

4. The train data is split into different parts according to the workers available using array_splits.
5. The above 4 steps are done by the **master** worker.

```
#master worker
if (rank==0):
    #Reading data set
    df_kdd = pd.read_csv('kdd.txt',low_memory=False)
    #Pre processing data
    df_kdd_x,df_kdd_y = preprocessing(df_kdd)
    #Splitting data for training and testing
    x_train_df, y_train_df, x_test_df, y_test_df = train_test_split(df_kdd_x,df_kdd_y)

    #Splitting data for parallelism
    x_train_split = np.array_split(x_train_df,size)
    y_train_split = np.array_split(y_train_df, size)

#other workers
else:
    x_train_split = None
    y_train_split = None
```

6. Initialize the beta values to zeros with the correct dimensions(one beta for one column).
7. Then the data splits are send to all the workers using comm.scatter() and the beta is broadcasted to all by comm.bcast().

```
#scatter the divided train dataset across workers
x_train_split = comm.scatter(x_train_split,root=0)
y_train_split = comm.scatter(y_train_split,root=0)

#initializing beta
beta = np.zeros((x_train_split.shape[1],1))
print("shape of beta-", beta.shape)
```

8. PSGD is implemented until convergence criteria is met. If the difference between the current and previous train rmse value is very less (0.000001),then iteration stops. But sometimes this may not converge to optimal solution as it **depends on the learning rate** taken. Hence I have also taken the maximum iterations as 200.

```
#While convergence criteria is not met
while(not converged and counter<max):
    #Broadcasting the beta to all workers
    beta = comm.bcast(beta,root=0)

    #master worker
    if(rank == 0):
        #Finding y train and y test values
        #Calculating the rmse values for y predicted and true
        y_predicted_train = np.dot(x_train_df,beta)
        rmse_train.append(rmse(y_predicted_train,y_train_df))

        y_predicted_test = np.dot(x_test_df,beta)
        rmse_test.append(rmse(y_predicted_test,y_test_df))

    #Calling the function to do stochastic gradient descent for each split
    local_beta = PSGD(x_train_split,y_train_split,beta,lr)
    comm.barrier()

    #collecting the betas for each split
    local_betas = comm.gather(local_beta,root = 0)
```

9. The rmse train and test scores are also calculated by finding the corresponding target prediction values using the updated beta for each iteration.

```
#Function to calculate the rmse value
def rmse(y_predicted, y):
    error = y - y_predicted
    rmse = np.sqrt(np.mean(error ** 2))
    return rmse
```

10. Time is also calculated for the whole program. This program was run for different number of workers { 1, 2, 4, 6, 7 }.

Parallel stochastic Gradient Descent :

- Here each worker gets a data split , while the beta values are available to all
- For each data point in the data split, we find a new beta as follows:
    - Y predicted value is found by dot operation.
    - Gradient is found according to the equation above.
    - New beta values are found using the learning rate and gradient.
    - The old beta values are replaced with the new ones

- After each data point in each split is traversed, all the final betas of different splits are gathered together using comm.gather.
- The master worker now finds the average of all the local betas gathered , which will be the new global beta value. This value is then broadcasted to all the workers.
- The master works calculates the predicted y values using the new betas for the train and test set and then finds the root mean square error between the predicted value and the true value. These error scores are appended to array
- The master also checks the convergence criteria. If satisfied then program stops, else counter is updated and again iteration continues with new betas which gets broadcasted.

```python
#Function to perform stochastic gradient descent
def PSGD(x,y,beta,lr):
    #for each data point
    for i in range(x.shape[0]):
        #Finding y
        y_predicted = np.dot(x[i],beta)

        b = x[i]
        b=b[:,np.newaxis]
        diff = y[i] - y_predicted
        diff = diff[:,np.newaxis]
        #Finding gradient
        grad = -2 * np.dot(b,diff)
        #finding new beta
        new_beta = beta - lr * grad
        beta = new_beta

    return beta
```

```python
#master worker
if(rank == 0):
    print("local betas, ", len(local_betas))
    #Finding beta calculation
    beta = np.mean(local_betas, axis=0)

    if(counter>1):
        #storing rmse values for checking
        current_rmse = rmse_train[counter]
        prev_rmse = rmse_train[counter-1]

        #convergence criteria check
        if(prev_rmse - current_rmse < 0.000001):
            converged = True
            print("------Converged------")
            print("Iteration- ", counter)
            print("\n Time taken for convergence for {} workers-{}".format(s
            break

        else:
            print("------Continue------")
            print("Iteration - ", counter)

    #maximum iterations
    if(counter == max-1):
            #Can occur if learning rate are not chosen right
            print("Not converged in {} iterations hence stopping".format(max
            print("\n Time taken for {} workers-{}".format(size,round(MPI.Wt
            break
    counter+=1
```

MPI framework:
The reading, preprocessing, and splitting of data is done by the Master worker. Then the data splits are sent using scatter method and betas are sent using broadcast.

Then each worker handles its given data. The master also handles it. They find the local betas for their given data split.

Then the local betas are gathered back and then the master finds the global beta, which is again broadcasted if convergence is not met. The master also checks the convergence criteria and the rmse values.

Thus the main part of the algorithm which takes the most time, which is calculating the new beta for each data point by finding the gradient , is handled in a parallel manner as we distribute the data among workers. Thus all workers are participating in the actual work.

I have observed that the Train rmse values always decrease with the increase in number of iterations. The test rmse values also exhibit a decreasing trend, but sometimes show slight increases. This maybe because of overfitting of the model or lack of regularisation
Also I have observed that the time taken depends on the learning rate. The time taken decreases as the number of workers increases.

Second Dataset

The second dataset given is the Dynamic Features of VirusShare Executables Data Set https://archive.ics.uci.edu/ml/datasets/Dynamic+Features+of+VirusShare+Executables

This dataset is a folder containing multiple files having the data in the form of svmlight / libsvm format. This format is a text-based format, with one sample per line. It does not store zero valued features hence is suitable for sparse dataset. The first element of each line can be used to store a target variable to predict.

Each line contains features and value in the form feature:value. First I wrote all the data from all the files to a single file. Then each line was read and the first element was taken and put into a dictionary under the key value ‘TARGET’.

The rest of the elements in the line were split based on ':'. Those on the left became the key value in the dictionary and those on the right became the value.

This was done for all lines. The final dictionary was converted to a data frame. It is of shape (107856, 265).

```
    return beta
#Function to read the file
def read_file(path):

    #Write all data in one file
    data=os.listdir(path)
    #opem file to write
    with open('all.txt','w') as s:
        for f in data:
            f=open('virus_dataset/'+f)
            s.write(f.read())
    # Read to list
    with open("all.txt", mode="r") as fp:
        svmformat = fp.readlines()

    # For each line we save the key:values to a dict
    df_list = []

    for line in svmformat:
        test_dict = dict()
        #First value is the target
        line_split = line.split(' ')
        test_dict["TARGET"] = line_split[0]

        #among the rest of elements in the line
        for elt in line_split[1:]:
            elt = elt.rstrip()  # Remove 'n'
            #Left of : is feature name and Right is value
            elt_split = elt.split(':')

            if(len(elt_split)==2):
                col, value = elt_split[0], elt_split[1]
                #store the features and values in dictionary
                test_dict[col] = value
        #append all the dictionaries for each line
        df_list.append(test_dict)

    #convert dictionary to data frame
    df_virus = pd.DataFrame(df_list)
    return df_virus
```

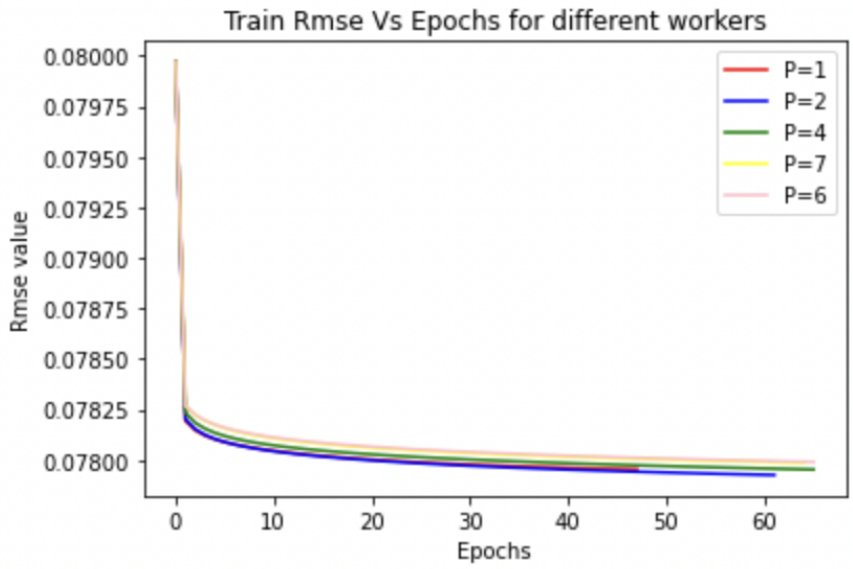
The rest of the functions are the same as was used in the previous dataset. All the hyper parameters taken are also same. I observed that the model was not converging when retaining the same criteria. But convergence was occurring, when I changed the criteria to the difference of current and previous rmse values must be less than 0.00001 or by slightly increasing the learning rate.

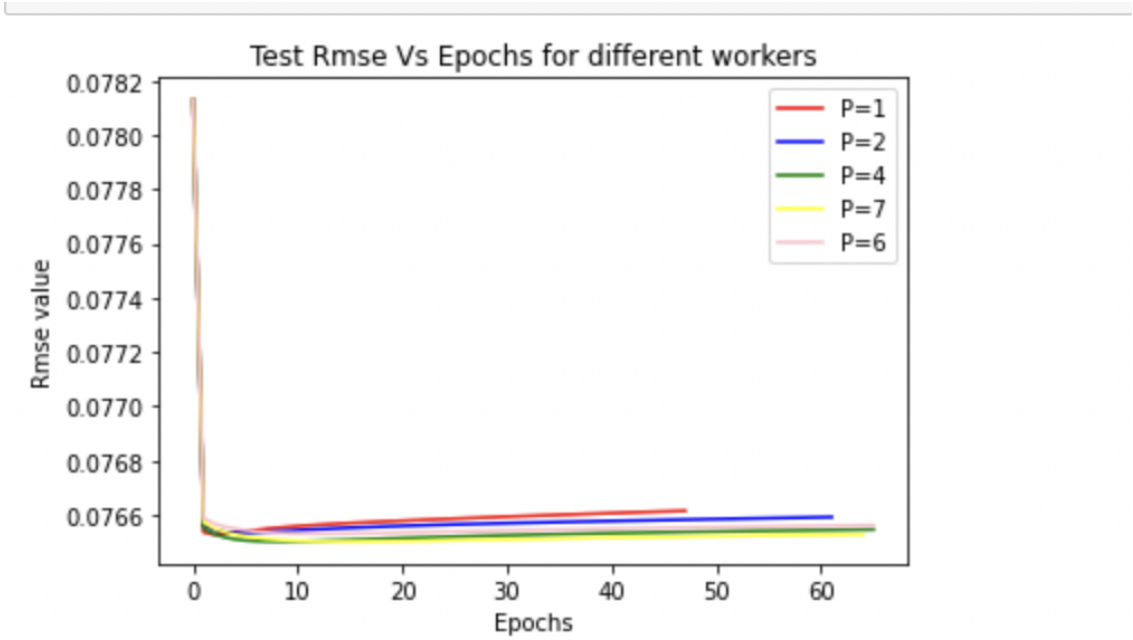
- Exercise 2: Performance and convergence of PSGD

  Convergence behaviour

  Checking the convergence behaviour of the model learned through PSGD and comparing it to a sequential version. Here we plot the convergence curve (Train/Test score versus the number of epochs) for P = {1, 2, 4, 6, 7}.

  For KDD Dataset:





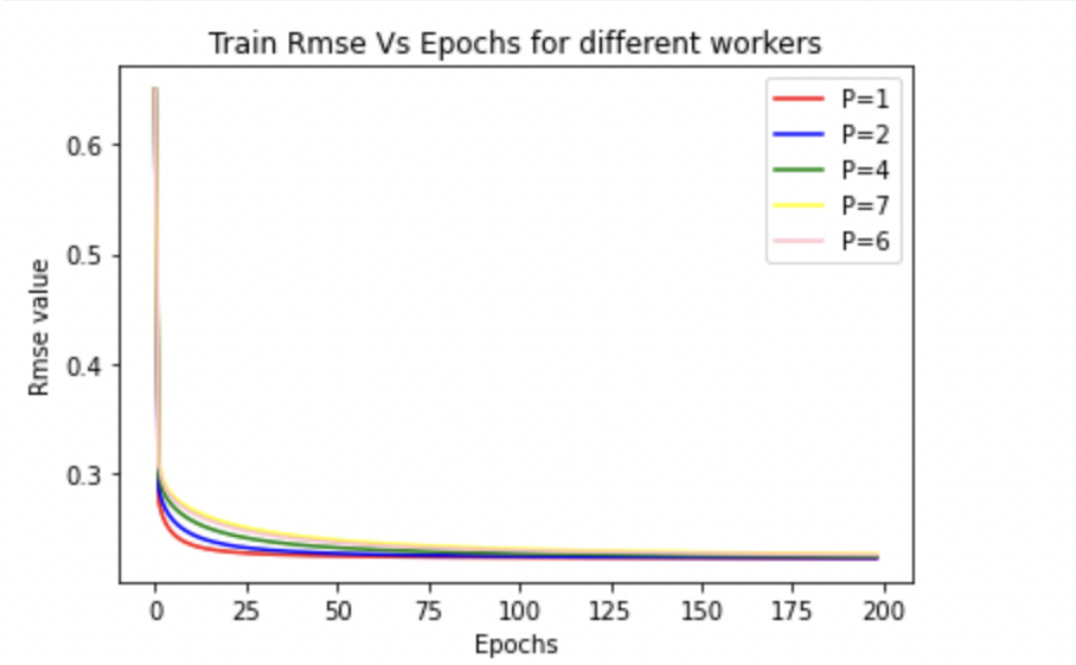
Observations:

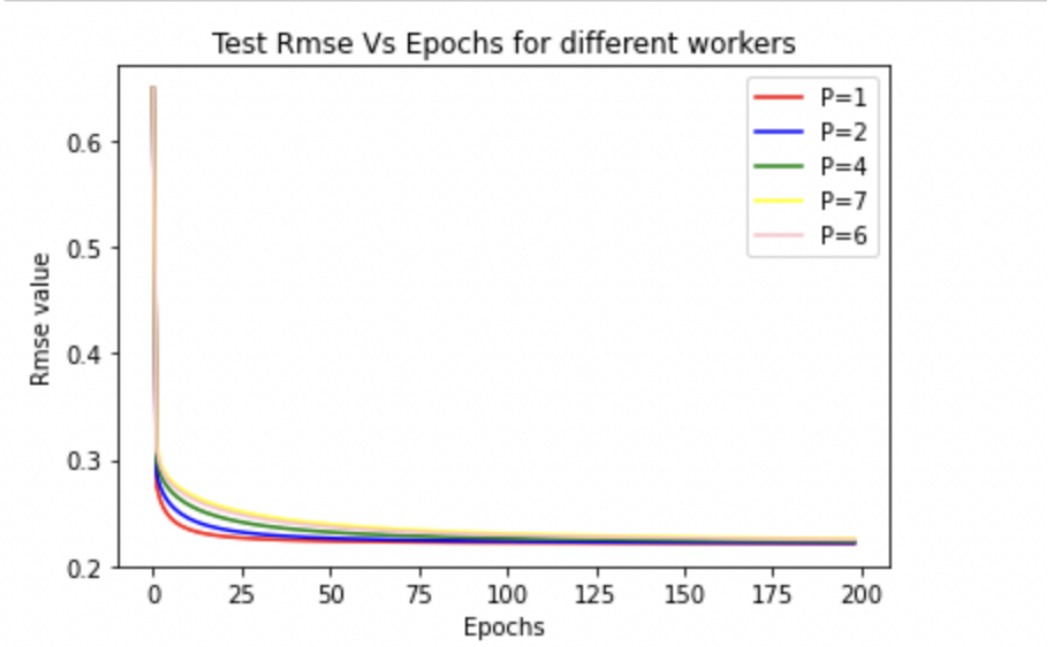
We can see that in all cases, convergence occurs in less than 100 epochs.

Train Rmse values decrease with increase in epochs. The steepest decrease occurs in the beginning. Then its more gradual. The range of rmse values are also very less. We can see that the rmse values are slightly more when the number of processes increase.But this is very very slight, this can be due to the data splits and averaging. Also more workers require more iterations to converge. But these values keep changing each time I run. Hence there may not be a significant relation between the number of workers and the convergence or learning curve.

For Test Rmse graph, the rmse values show an overall decreasing trend with the increase in epochs. But here, there is slight increase in values also. This can be due to overfitting of model or lack of regularisation. Here also the range of rmse values is very small. Also there is no particular effect on rmse values on changing the number of workers.

Virus Dataset:





Observations:

We see that for virus dataset also, both train rmse and test rmse values decrease as we increase the epochs. Here the rmse values range between 0.2 - 0.7. We

see that they converge near 200 epochs. There is a steep decrease in the beginning and then the decrease is gradual. We can also observe that the rmse values are slightly higher for a higher number of workers. This maybe due to the averaging over more number of local betas(not sure). Also for virus dataset we see that the test rmse values are also less, implies the model is not overfitted.

Overall both datasets show similar trend.

I have tried the program for different numbers of workers. The following table shows the time taken in seconds for the same.

Total time taken for running the whole program in seconds:

Number of Workers	Time in seconds for KDD dataset	Time in seconds for Virus dataset
P=1	72.8693	223.1402
P=2	54.2405	136.0188
P=4	48.0322	120.6206
P=6	48.5409	120.0451
P=7	47.7746	118.0429

Observations:

We see that the time decreases as number of workers increase. There is significant decrease when we increase the workers from 1 to 2. After that the decrease is less.

The time taken by KDD dataset is lesser than that taken by Virus dataset. This can be due to the learning rate, convergence criteria and the size of the dataset.

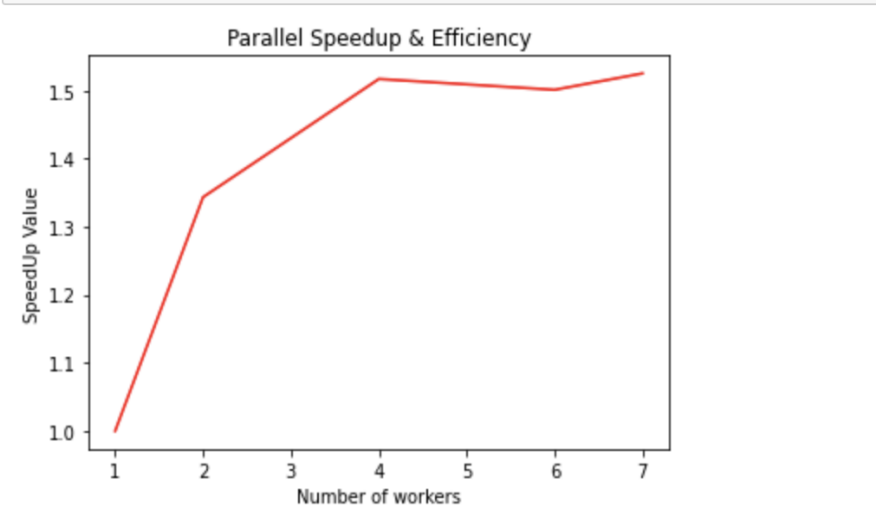
Speed Up Graph

The speedup gained from applying n workers, Speedup(n), is the ratio of the one-worker execution time to the n-worker parallel execution time:

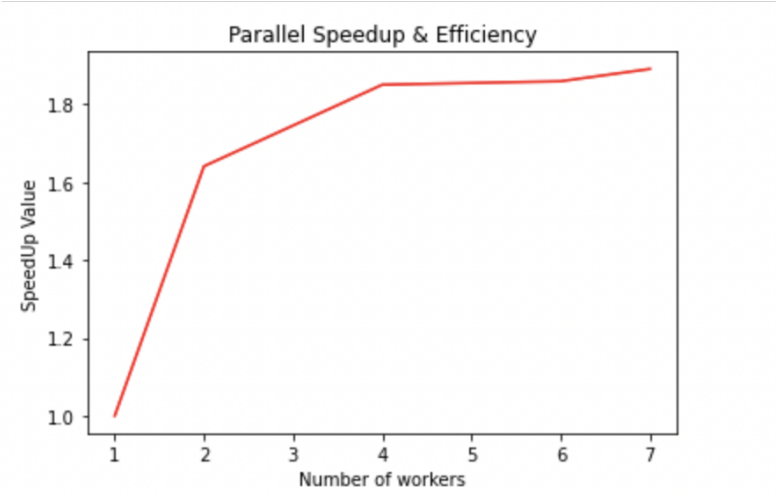
$$Speedup(n) = T(1)/T(n).$$

So Speedup(n) should be a number greater than 1.0, and the greater it is, the better. Normally the Speedup(n) is less than n as there will be some parts of code we cannot run parallely(like calculating the global centroid), but in some cases, it becomes greater than n which is called super linear speed up.

Speed up graph for KDD dataset



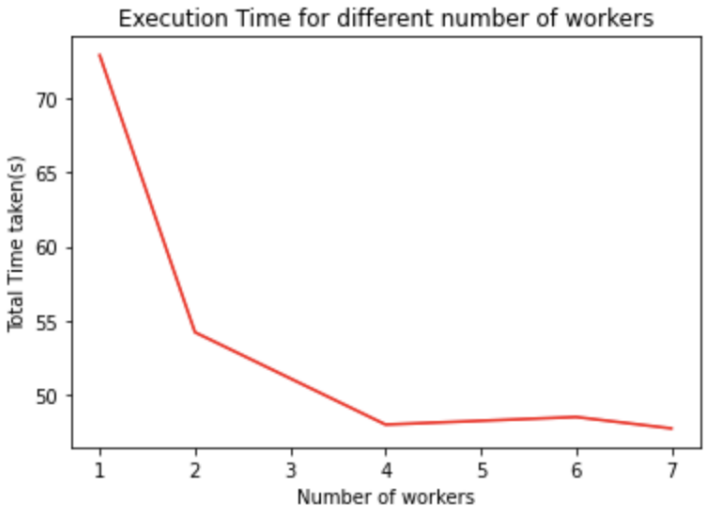
Speed Up graph for Virus dataset



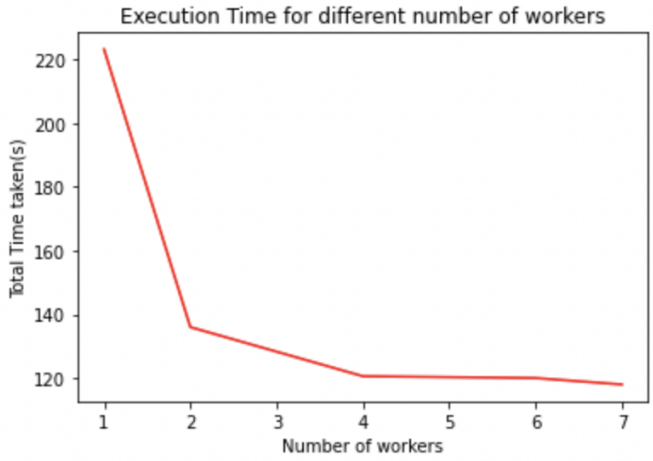
We can see that the speed increase the most from 1 to 2 workers. Then the rate of increase is lesser and it decreases slightly from 4 to 6 workers for KDD dataset but for virus dataset there is slight increase. After that, the speedup value again increases. It shows a sublinear trend overall.

The observations are as expected. There is no super linear increase as all parts of the code cannot be parallelised.

The below graphs plots the execution time with the number of workers
KDD Dataset:



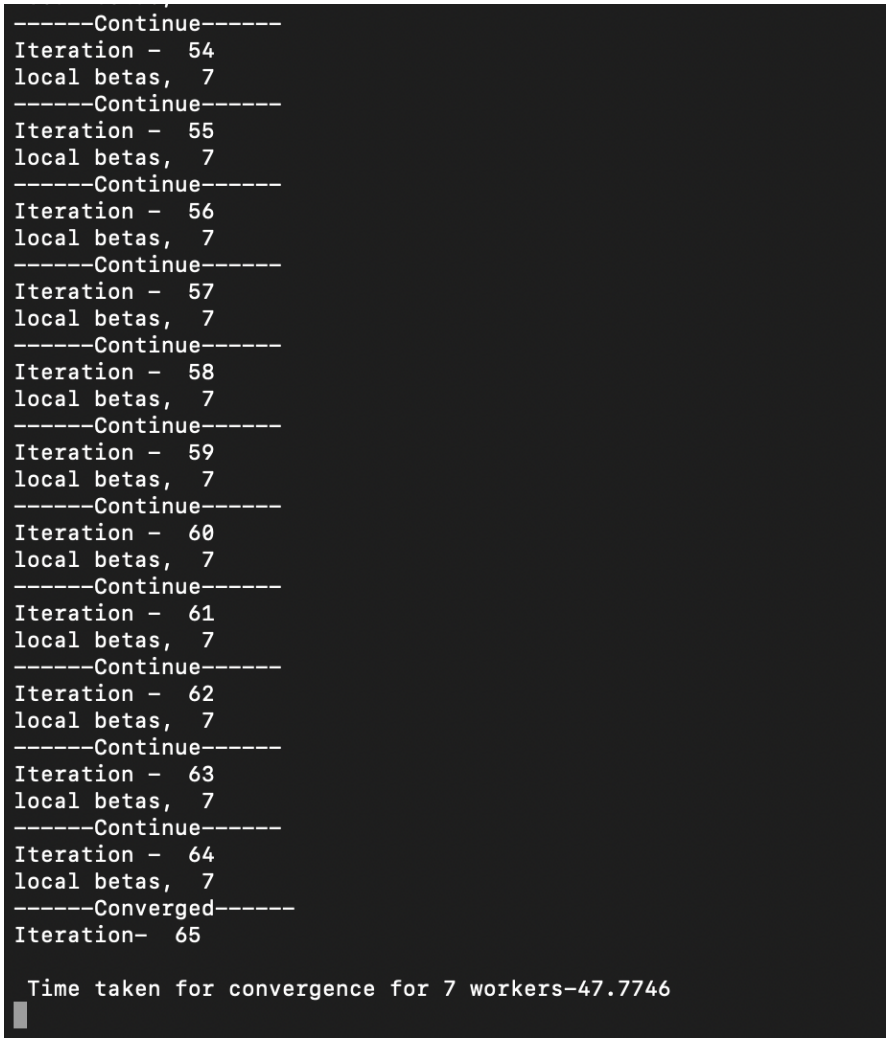
Virus Dataset:



Here we can see that the execution time decreases as the number of workers increase as expected for both datasets.

Sample Outputs for KDD dataset:

```
(/opt/anaconda3) sruthysanthosh@Sruthys-MacBook-Air Sem2 % mpiexec -n 7 python Lab5_312213.py
0     90569
14      941
22      591
30      577
7       503
      ...
6         1
24        1
67        1
68        1
8         1
Name: TARGET_D, Length: 71, dtype: int64
[ 0 17  7 35 30 14 12 22 16 66 23 10 70 19 21  9 31 13 33 49 40 36 25 32
 53 38 18 47 39 34 27 44  4 43 42 37  5 59 57 29  2 45 48 46 15  6  1 61
 63 41 26 64 52 55 20 68 60 67 24 51 56 50 65 69 54 28 11  3 58 62  8]
Shape of x (66788, 477)
Shape of y (66788, 1)
shape of beta- (477, 1)
local betas,  7
local betas,  7
local betas,  7
------Continue------
Iteration -  2
local betas,  7
------Continue------
```



Rmse values for 7 workers for KDD:

kdd_results_7

	Train RMSE	Test RMSE
0	0.07996978753665050	0.0781296334795994
1	0.07826919780581920	0.07657318775555290
2	0.07822317889522670	0.07655051417926740
3	0.07819436980990690	0.07653777349726630
4	0.07817359641537310	0.07652867401473140
5	0.07815743282069520	0.07652172809932600
6	0.07814427368550560	0.076516368925579
7	0.07813322926081	0.07651225844790040
8	0.07812374860435110	0.07650914319449800
9	0.07811546545003400	0.07650682052600780
10	0.07810812441787910	0.07650512743137510
11	0.07810154115662200	0.07650393363327550
12	0.07809557889520600	0.07650313569655000
13	0.07809013381069900	0.07650265182083840
14	0.07808512550897850	0.07650241740171810
15	0.07808049063435650	0.07650238138053910
16	0.07807617846965670	0.07650250331817750
17	0.07807214784031370	0.07650275108690680
18	0.07806836489350760	0.07650309906721700

19	0.07806480147678080	0.07650352674617620
20	0.07806143393511800	0.0765040176299881
21	0.07805824220522020	0.076504558400111
22	0.078055209124261	0.07650513825726880
23	0.07805231989575640	0.07650574841012820
24	0.07804956167212320	0.0765063816753738
25	0.07804692322500760	0.07650703216369040
26	0.07804439468240580	0.07650769503215330
27	0.07804196731715880	0.07650836628811380
28	0.07803963337534890	0.07650904263315570
29	0.07803738593596980	0.07650972133835390
30	0.07803521879531180	0.07651040014408250
31	0.07803312637103330	0.07651107717915840
32	0.07803110362202560	0.07651175089527950
33	0.07802914598103770	0.07651242001360990
34	0.07802724929767920	0.07651308348105990
35	0.07802540978991530	0.07651374043433160
36	0.07802362400255670	0.07651439017021640
37	0.07802188877154210	0.07651503212094640
38	0.07802020119304860	0.07651566583364780
39	0.07801855859664690	0.07651629095314050
40	0.0780169585218633	0.07651690720747820
41	0.07801539869762850	0.07651751439574240
42	0.07801387702418340	0.07651811237769860
43	0.0780123915570866	0.07651870106499830
44	0.07801094049303060	0.07651928041366760
45	0.0780095221572176	0.0765198504176743
46	0.07800813499209110	0.0765204111033992
47	0.07800677754724640	0.07652096252487240
48	0.07800544847037340	0.07652150475965580
49	0.07800414649910630	0.07652203790527670
50	0.078002870453671	0.07652256207613030
51	0.07800161923023940	0.07652307740078600
52	0.07800039179490910	0.07652358401963860
53	0.07799918717824140	0.07652408208285950
54	0.07799800447029540	0.07652457174860560
55	0.07799684281610720	0.07652505318145350
56	0.07799570141156740	0.07652552655102850
57	0.07799457949965710	0.0765259920308052
58	0.077993476367006	0.07652644979705740
59	0.07799239134074210	0.0765269000279389
60	0.07799132378560340	0.07652734290268080
61	0.07799027310128880	0.07652777860088930
62	0.07798923872002410	0.07652820730193400
63	0.07798822010432440	0.07652862918441580
64	0.07798721674493480	0.07652904442570480

The virus dataset

```
0.833333333333 24:2 27:1 67:4 78:1 84:3 93:1 100:2 107:165 1
0.363636363636 0:1 15:5 19:190 20:4 22:1 24:79 34:67 36:10 4
0.767857142857 5:6 8:1 15:2 19:9 20:3 24:35 34:41 36:3 49:2
0.803571428571 5:6 8:1 15:2 19:9 20:3 24:35 34:41 36:3 49:2
0.789473684211 5:6 8:1 15:2 19:9 20:3 24:35 34:41 36:3 49:2
0.789473684211 5:6 8:1 15:2 19:9 20:3 24:35 34:41 36:3 49:2
0.785714285714 5:6 8:1 15:2 19:9 20:3 24:35 34:41 36:3 49:2
0.796296296296 0:1 5:37 8:1 15:2 19:80 20:31 22:14 24:129 34
0.789473684211 5:6 8:1 15:2 19:9 20:3 24:35 34:41 36:3 49:2
0.789473684211 5:6 8:1 15:2 19:9 20:3 24:35 34:41 36:3 49:2
0.789473684211 5:6 8:1 15:2 19:9 20:3 24:35 34:41 36:3 49:2
0.811320754717 0:1 5:33 8:1 15:2 19:80 20:27 22:14 24:129 34
0.803571428571 5:6 8:1 15:2 19:9 20:3 24:35 34:41 36:3 49:2
```

Output samples for virus dataset:

```
local betas,  7
------Continue------
Iteration -  196
local betas,  7
------Continue------
Iteration -  197
local betas,  7
------Continue------
Iteration -  198
local betas,  7
------Continue------
Iteration -  199
Not converged in 200 iterations hence stopping

 Time taken for 7 workers-118.0429
```

virus_results_4

	Train RMSE	Test RMSE
0	0.650129215469078	0.6500540200262390
1	0.2972839534076610	0.29924578462909600
2	0.2847541577109390	0.28640104217075000
3	0.2780427866442360	0.2794625166611000
4	0.27319601572762600	0.27446784395934800
5	0.26930742618144500	0.27047713222849300
6	0.26604138488844700	0.26713428454661400
7	0.26322899002552800	0.26425943739721100
8	0.2607671216840170	0.26174367605395700
9	0.2585856064497700	0.25951369377274
10	0.25663368318994800	0.2575169712337560
11	0.25487321263108900	0.25571434118122700
12	0.25327476142670700	0.2540757454663950
13	0.2518151343880110	0.25257759632907100
14	0.25047573033209000	0.2512010390838600
15	0.24924140534869400	0.24993076222577000
16	0.24809966535205200	0.24875415828286800

References:

- https://www.ismll.uni-hildesheim.de/lehre/bd-16s/exercises/bd-02-lec.pdf
- https://github.com/rebordao/kdd98cup/blob/master/lib/analyser.py
- https://stackoverflow.com/questions/26414913/normalize-columns-of-pandas-data-frame
- https://stackoverflow.com/questions/33480985/convert-numpy-vector-to-2d-array-matrix
- https://www.activestate.com/resources/quick-reads/how-to-slice-a-dataframe-in-pandas/

- https://stackoverflow.com/questions/33480985/convert-numpy-vector-to-2d-array-matrix
- https://www.adamsmith.haus/python/answers/how-to-check-which-columns-in-a-pandas-dataframe-has-nan-values-in-python
- https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_svmlight_file.html
- https://python.tutorialink.com/how-to-load-svmlight-format-files-in-compressed-form-to-pandas/
- https://www.geeksforgeeks.org/ml-stochastic-gradient-descent-sgd/