3116 – Lab Distributed Data Analytics – Group 2

Exercise Sheet 4

Yuvaraj Prem Kumar

303384, premyu@uni-hildsheim.de

Part 1: Parallel Linear Regression

Step 1: Data pre-processing and import operation

Perhaps the most important step. Data pre-processing needs to be done properly to ensure it is ready for analysis. My target here was to place the X and Y elements into Pandas DataFrame, for easier processing.

About the Dynamic Features of VirusShare Executables" dataset: This is a sparse dataset, with 107856 number of instances ('rows') and 479 number of attributes ('columns'). The values in the files are stored as key-value pairs. The first value of a key-value pair corresponds to the categories of operations the executables (malware) performed. Such as system-calls, file openings, writing, and etc. The second value corresponds to the frequency of these operations.

Function "*ImportData*" called to pre-process and read the dataset:

i. All files are concatenated into a single file with blob library. This is a Unix type pattern matching, which is lightweight on the memory usage even though there are large number of files with large size.
ii. The merged input file is read with sklearn.load_svmlight_file library. This will output X and Y components separately.
iii. X and Y components are read as pandas dataframes. Additionally, X is initialized with "todense()" to return a matrix. Since this is a sparse matrix, the Nan values are converted to zeros with "fillna(0)". X and Y are then normalized with sklearn.preprocessing.normalize.
iv. The imported output of X dataframe: 107856 rows x 479 columns
v. The imported output of Y dataframe: 107856 rows x 1 columns

About the "KDD Cup 1998" dataset: Used for The Second International Knowledge Discovery and Data Mining Tools Competition, which was held in conjunction with KDD-98 The Fourth International Conference on Knowledge Discovery and Data Mining. The competition task is a regression problem where the goal is to estimate the return from a direct mailing in order to maximize donation profits.

Function "*ImportData*" called to pre-process and read the dataset:

i. Input file "cup98LRN.txt" is imported via pandas.read_csv(). A few parameters are passed to the read_csv() function, in order to avoid importing metadata, etc.

ii. A for loop with pandas.astype(category) is employed to convert the object columns to categorical data.
iii. In X and Y dataframes, unwanted columns are dropped, and target Y vector is set. Additionally, Y is initialized with np.expand_dims() to reshape the array along the axis=0 line; and add a new dimension.
iv. Finally, X and Y are normalized with sklearn.normalize()
v. The imported output of X dataframe: 95412 rows x 224 columns
vi. The imported output of Y dataframe: 95412 rows x 1 columns

Steps 2 to 5 are now exactly the same for both datasets, with the exact same Python code.

Step 2: Predicted values

The simple linear regression formula is given by $y = \beta_0 + \beta_1 x_1 + \cdots$

The function "*predict_yhat*" is used to calculate the predicted value, $\hat{y}$ of both the training dataset and testing dataset. As this is a numpy matrix, it's simply calculated as:

$$for\ i\ in\ range\ (features):\ \hat{y} = x[i] \times \beta[i]$$

Step 3: Stochastic Gradient Descent

The weights, β are recalculated based on learning rate (α), error and sample size. As discussed in the lab session, the sample size is set to 1000 and the learning rate, α is set to 0.01 here. Sample size is small to account for the number of processes, otherwise it would take too long to run for more processes (1 process – 1000 samples, 2 processes – 100 samples etc..). Alpha, α is learning rate i.e. how fast we want the parameters to be updated. It should not be very less as this will make the gradient descent slow and also not very big as that can make the gradient descent to overpass the minima and it will diverge instead of converging. The value was found by trial and error running, before I settled on 0.01. The function "*SGD_func*" is defined as follows:

$for\ i\ in\ range(random(data\ rows))$:
    $error = \hat{y}.rowindex[i] - y.rowindex[i]$
    $for\ j\ in\ range\ (features)$:
        $\beta[j] -= \alpha \times error \times X(matrix\ indices)$
$return\ \beta$

Step 4: Root mean squared error:

Simply calculated by $\sqrt{(\hat{y} - y)^2.mean()}$

Step 5: Parallelization strategy:

1. Within root node 0:
    - Dataset imported as detailed in Step 1
    - Imported dataset split into testing/training dataset using *sklearn. model_selection.train_test_split* into 70% training set and 30% test set. The data split is random.
    - Training data X and Y are split using *numpy.array_split* into slices. These are then sent to the worker nodes by *mpi.scatter().*
    - Initial RMSE is calculated on the testing set for verification. The starting values are zeros.
2. For each epoch:
    - Predicted value for $\hat{y}$ and error are calculated
    - following by weights coefficient, β
    - new average weights are gathered back to the root node
    - root node calculates the RMSE on testing set
    - process loops and iterates the number of epochs
3. Final error is calculated on test set. If at any point the error starts to increase (cost function) then the loop breaks and prints the final RMSE value.

The initial idea was to run an infinite while loop (as per previous Lab3). However past running around 30+ iterations, my machine overheats due to RAM usage and CPU % runs high. Hence, I set the epochs to a hard limit (12), but maintain the loop break in case it does converge earlier. Part 2 details the results of convergence.

Part 2: Performance and convergence of PSGD

The RMSE results and convergence for prediction on testing data is shown here. Training data results are not included as that is of lesser interest. Results could vary with the sample size, but as mentioned in Step 3, I have stuck to using a sample size of 1000.

Virus Share:

The program is run for 12 epochs, to see the convergence of the RMSE values. The tables below show the output for $P = \{1,2,4,6,7\}$

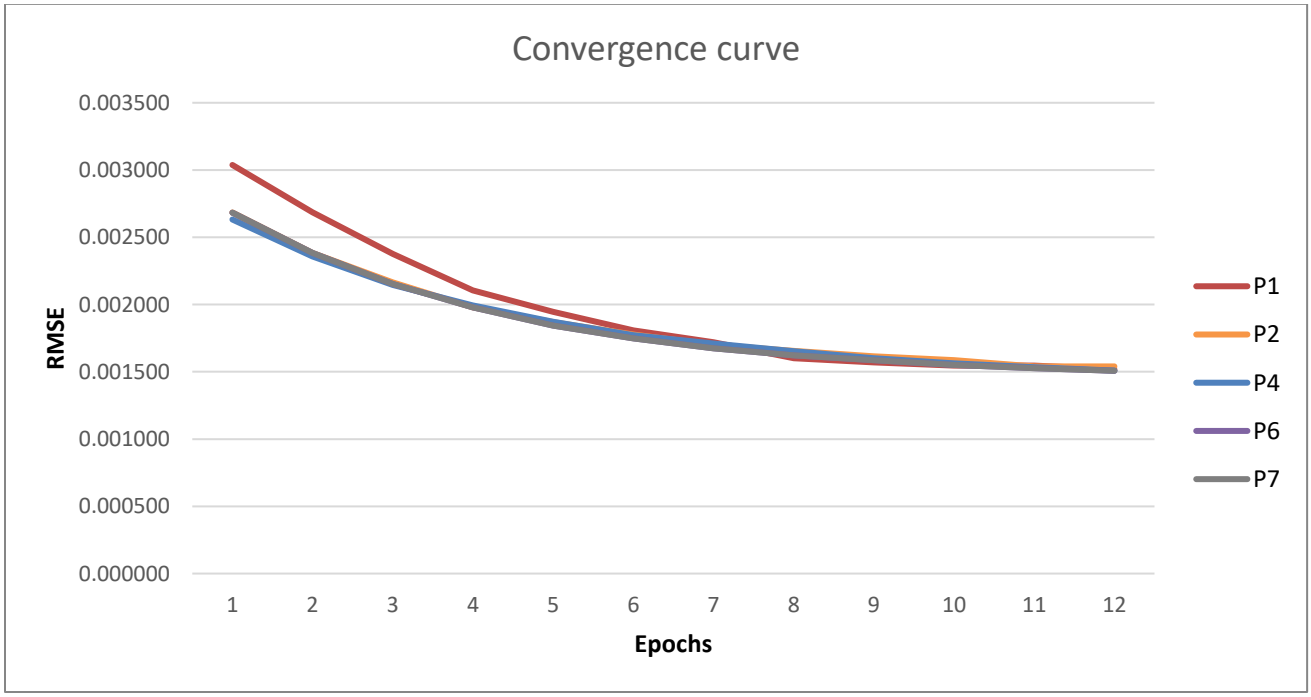
P	Epochs	RMSE value	P	Epochs	RMSE value	P	Epochs	RMSE value	P	Epochs	RMSE value	P	Epochs	RMSE value
1	1	0.64261	2	1	0.56678	4	1	0.57463	6	1	0.57214	7	1	0.57470
	2	0.63438		2	0.50829		2	0.50946		2	0.50765		2	0.51485
	3	0.62718		3	0.45955		3	0.46133		3	0.46115		3	0.46571
	4	0.61995		4	0.42411		4	0.42465		4	0.42502		4	0.42806
	5	0.61322		5	0.39436		5	0.39901		5	0.39697		5	0.39946
	6	0.60547		6	0.37330		6	0.37890		6	0.37601		6	0.37920
	7	0.59868		7	0.35677		7	0.36185		7	0.36184		7	0.36295
	8	0.59124		8	0.34443		8	0.34922		8	0.35043		8	0.35097
	9	0.58403		9	0.33636		9	0.34044		9	0.34082		9	0.34315
	10	0.57799		10	0.33119		10	0.33294		10	0.33421		10	0.33500
	11	0.57124		11	0.32782		11	0.32650		11	0.32838		11	0.32912
	12	0.56543		12	0.32363		12	0.32203		12	0.32357		12	0.32489

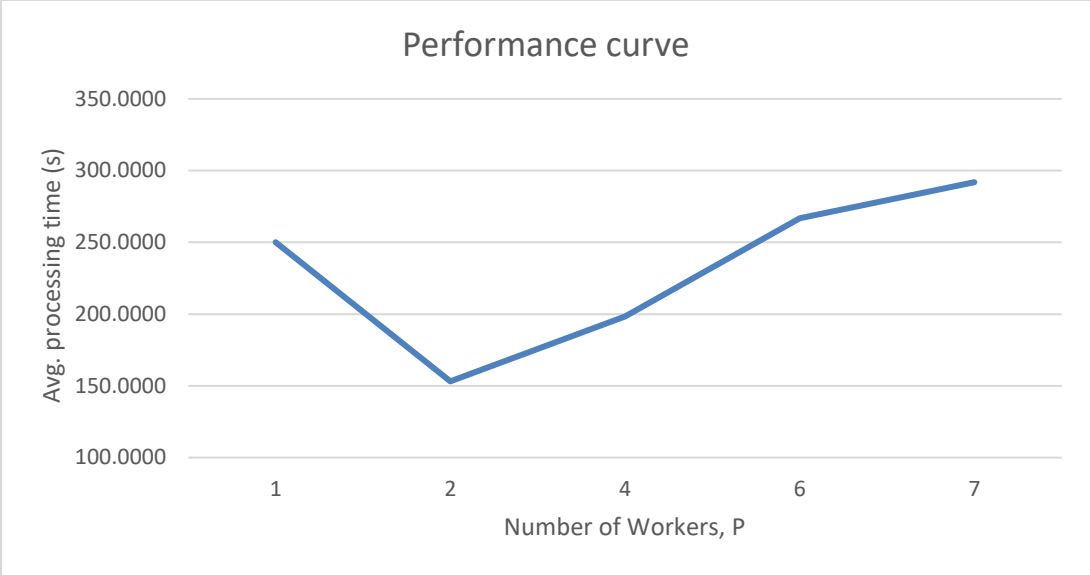
It can be seen that the RMSE values converges, and for $P = \{2,4,6,7\}$, the final RMSE value is more or less the same. When P>=2, the program outperforms by getting to the value quickly. Since the β coefficients are calculated in parallel, multiple RMSE are calculated, and the final value is actually the average of all of them. Hence the performance gain. Having multiple predictors in a linear model means that some predictors may have an influence on other predictors.

The table below shows the final average processing time for all sizes of P.

# Workers, P	Final RMSE	Avg. processing time (s)
1	0.001518	250.0052184
2	0.001540	153.1586476
4	0.001509	198.4008957
6	0.001508	266.8280029
7	0.001508	291.9363488

The average processing time drops from P1 to P4, however after that it starts to increase again. As in previous labs, this could be due to the fact that my laptop is not an actual distributed execution environment, so running in a parallel cluster should give the expected results. Adding more processes will only increase the overhead computation cost, due to the interleaving between processes. The plots of the convergence curves and learning curves against processing time better visualize the results:



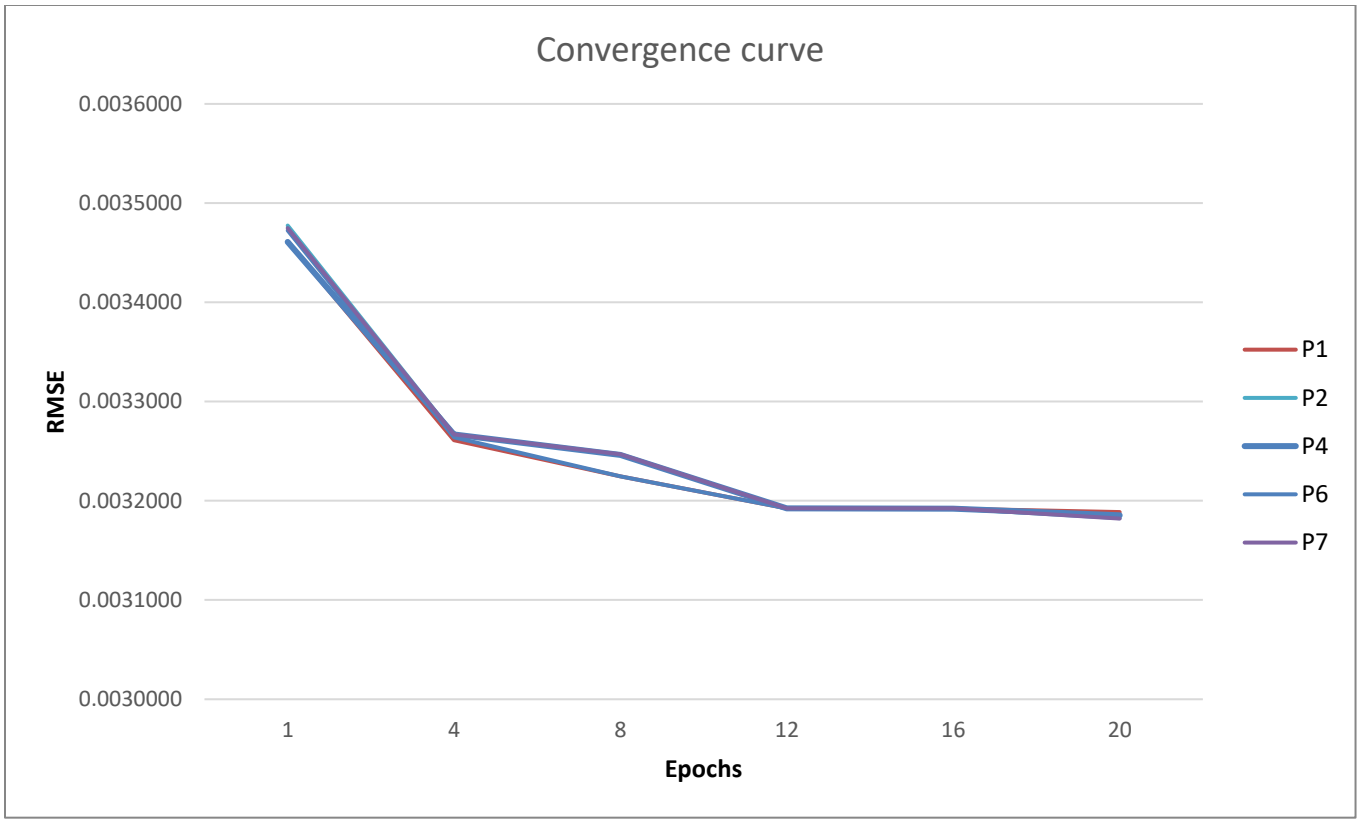


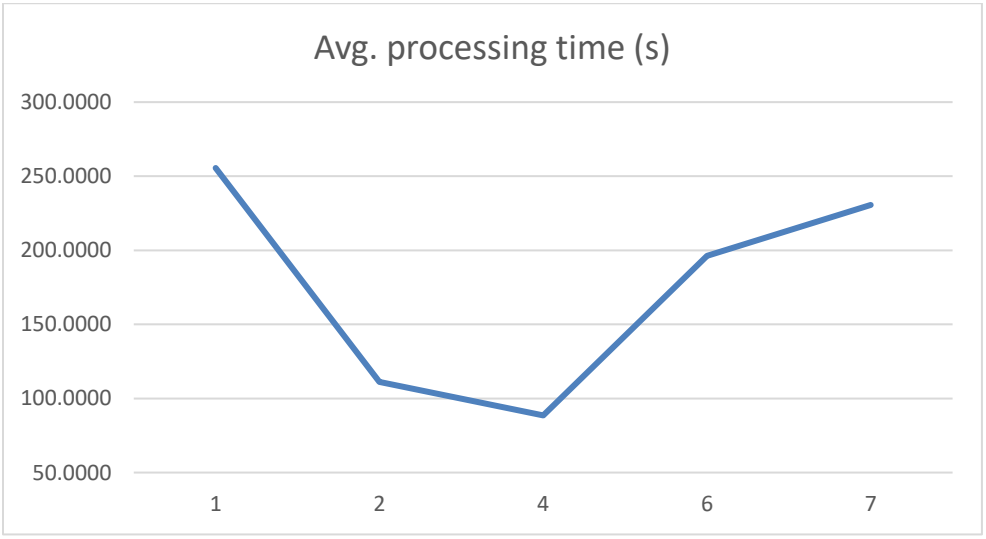
KDD Cup:

The program is run for 20 epochs, to see the convergence of the RMSE values. The tables below show the output for $P = \{1,2,4,6,7\}$. It's interesting to note that the epoch time is much shorter here. This is due to the dataset values. However, the convergence is slower here. As mentioned, due to my laptop's limitation, I was forced to intentionally limit the number of epochs. Based on the curve, the RMSE values should still continue to drop with more epochs. For the average processing time, the same phenomenon as in Virus Share dataset is observed here; with P=6 and P=7 being sub-optimal due to additional communication cost between processes.

P	Epochs	RMSE value	P	Epochs	RMSE value	P	Epochs	RMSE value	P	Epochs	RMSE value	P	Epochs	RMSE value
1	1	0.0034612	2	1	0.0034774	4	1	0.0034609	6	1	0.0034725	7	1	0.0034749
	4	0.0032612		4	0.0032667		4	0.0032667		4	0.0032639		4	0.0032669
	8	0.0032246		8	0.0032462		8	0.0032462		8	0.0032246		8	0.0032470
	12	0.0031924		12	0.0031923		12	0.0031924		12	0.0031924		12	0.0031924
	16	0.0031921		16	0.0031922		16	0.0031922		16	0.0031923		16	0.0031923
	20	0.0031887		20	0.0031825		20	0.0031854		20	0.0031854		20	0.0031821

# Workers, P	Final RMSE	Avg. processing time (s)
1	0.00220865	255.5425193
2	0.00220250	111.1311318
4	0.00214539	88.5657890
6	0.00214539	196.2733927
7	0.00215211	230.6697428


Convergence curve
0.0036000
0.0035000
0.0034000
0.0033000
0.0032000
0.0031000
0.0030000
RMSE
1
4
8
12
16
20
Epochs
P1
P2
P4
P6
P7


Avg. processing time (s)
300.0000
250.0000
200.0000
150.0000
100.0000
50.0000
1
2
4
6
7

References:

[1] Mishra, M. (2018, May 8). Understanding Linear Regression in Machine Learning. Retrieved from https://medium.com/datadriveninvestor/understanding-linear-regression-in-machine-learning-643f577eba84

[2] Python, R. (2019, April 15). Linear Regression in Python ? Real Python. Retrieved from https://realpython.com/linear-regression-in-python/

[3] Peixeiro, M. (2018, November 26). Linear Regression??'Understanding the Theory. Retrieved from https://towardsdatascience.com/linear-regression-understanding-the-theory-7e53ac2831b5

[4] Linear Regression Tutorial Using Gradient Descent for Machine Learning. (2016, September 22). Retrieved from https://machinelearningmastery.com/linear-regression-tutorial-using-gradient-descent-for-machine-learning/

[5] Understanding format of data in scikit-learn. (11, 4). Retrieved May 19, 2019, from https://stackoverflow.com/questions/24380426/understanding-format-of-data-in-scikit-learn

[6] sklearn.datasets.load_svmlight_file — scikit-learn 0.21.1 documentation. (n.d.). Retrieved May 19, 2019, from https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_svmlight_file.html

[7] How can I replace all the NaN values with Zero's in a column of a pandas dataframe. (6, 6). Retrieved from https://stackoverflow.com/questions/13295735/how-can-i-replace-all-the-nan-values-with-zeros-in-a-column-of-a-pandas-datafra

[8] Srinidhi, S. (2018, July 30). How to split your dataset to train and test datasets using SciKit Learn. Retrieved from https://medium.com/@contactsunny/how-to-split-your-dataset-to-train-and-test-datasets-using-scikit-learn-e7cf6eb5e0d

[9] Khurana, N. (2018, September 6). Linear Regression in Python from Scratch. Retrieved from https://medium.com/analytics-vidhya/linear-regression-in-python-from-scratch-24db98184276

Appendix A

Sample program output (Virus Share) for P=4

Command Prompt

```
C:\PythonProjects>mpiexec -n 4 python Lab4.py
Number of workers:  4
Importing dataset...
Initial RMSE:  0.6516395592340601
Previous RMSE 0.6516395592340601
Epoch #: 1, Current RMSE: 0.5746314850700451
Current time for Process #0: 17.55109439844273 secs.
Previous RMSE 0.5746314850700451
Epoch #: 2, Current RMSE: 0.5094617994236191
Current time for Process #0: 36.177055605965506 secs.
Previous RMSE 0.5094617994236191
Epoch #: 3, Current RMSE: 0.46133028601770293
Current time for Process #0: 55.243598444660165 secs.
Previous RMSE 0.46133028601770293
Epoch #: 4, Current RMSE: 0.4246464132370325
Current time for Process #0: 77.11713121119101 secs.
Previous RMSE 0.4246464132370325
Epoch #: 5, Current RMSE: 0.39900563822533713
Current time for Process #0: 97.7880926036014 secs.
Previous RMSE 0.39900563822533713
Epoch #: 6, Current RMSE: 0.3789040104456783
Current time for Process #0: 118.92741727656175 secs.
Previous RMSE 0.3789040104456783
Epoch #: 7, Current RMSE: 0.3618470919613995
Current time for Process #0: 138.43915884208582 secs.
Previous RMSE 0.3618470919613995
Epoch #: 8, Current RMSE: 0.3492188228596185
Current time for Process #0: 155.13557683829094 secs.
Previous RMSE 0.3492188228596185
Epoch #: 9, Current RMSE: 0.34044234221052383
Current time for Process #0: 173.062678381968 secs.
Previous RMSE 0.34044234221052383
Epoch #: 10, Current RMSE: 0.3329355309599529
Current time for Process #0: 191.1184680981787 secs.
Previous RMSE 0.3329355309599529
Epoch #: 11, Current RMSE: 0.3265003642605171
Current time for Process #0: 207.32913361189821 secs.
Previous RMSE 0.3265003642605171
Epoch #: 12, Current RMSE: 0.322032332148973
Current time for Process #0: 223.68387458293364 secs.
Total Time taken: 223.68391326732308
```

Appendix B

Example of exiting while loop after convergence achieved (cost function starts to go up). Data has been manipulated in this case to simply show to intended purpose. In the program, it doesn't really exit due to only 12 or 20 epochs being run.

```
C:\PythonProjects>mpiexec -n 2 python Lab4.py
Number of workers:  2
Importing dataset...
Initial RMSE:  0.6516533496459318
Previous RMSE 0.6516533496459318
Epoch #: 1, Current RMSE: 0.6439685754901846
Current time for Process #0: 12.623778383960598 secs.
RMSE limit reached, exiting loop...
Last epoch #: 1
Total Time taken: 12.623841530536083
Current time for Process #1: 12.47071449498435 secs.
RMSE limit reached, exiting loop...
Last epoch #: 1
Total Time taken: 12.623938241509677
```

Appendix C

My laptop overheats due to RAM and CPU limitations, forcing me to limit the number of epochs