inf-2220-a2

Sigurd Uhre

November 2022

1 Introduction

In this assignment the student will go deeper into the stochastic gradient decent algorithm, how it works and what we can use it for. The theory behind this and the implementation made will be explained in more detail. Tests will also be run with various parameters to see how the implementation responds to this. Limitations of the implementation and further work will also be addressed as well as an assessment of the results.

2 Methods and approaches

2.1 Technical background

The data-set given in this task is a full matrix without any holes. We need to implement a certain percentage of the data-points that are missing. The missing data is referred to as *Not a number* or *NAN*. **Mean squared error** or **MSE** measures the average of the squares of the errors—that is, the average squared difference between the estimated values and the actual value. MSE is a risk function, corresponding to the expected value of the squared error loss [1]. A large MSE value indicates that the data points are dispersed widely around its central moment (mean), whereas a smaller MSE suggests the opposite [2].

A larger MSE indicates that the data points are dispersed widely around its central moment (mean), whereas a smaller MSE suggests the opposite. A smaller MSE is preferred because it indicates that your data points are dispersed closely around its central moment (mean). This means that a smaller MSE is preferable and indicates that the data-points are "more" centered around the mean, compared to if the MSE is larger which indicates the opposite. To simplify, MSE can be described as the total error in the data-set (array, table, graph) from the true error to the predicted error.

Predictions
The example matrix in figure 1 below shows a group of users (U1,U2..) and a group of items (D1, D2..). All users have data-points fore some of the items, but none of the users have data-points for all the items. The empty data-points is what *Not a number* or *NAN* -points. The general idea of *prediction* is that the missing data-points needs to be filled in such that the values would be consistent with the existing data-points in the matrix [3].

	D1	D2	D3	D4
U1	5	3	-	1
U2	4	-	-	1
U3	1	1	-	5
U4	1	-	-	4
U5	-	1	5	4

Figure 1: Example of matrix [3].

This problem indicates that it underlies a latent feature that determines how a user rates an item. By looking at different users that have somewhat the same data-points it is possible to determine to some extent the value of a missing data-point. In figure 1, *U1* and *U2* have the same data-point for D4 and "only" one value in difference for D1. If we where to predict the value of data-point U2-D2, what would it be? Mabye 3 or 2 or somewhere in between. There is to few data-points in this scenario to give an good estimate, but with more data-points the better the estimate becomes [3].

Grading descent can be explained as descending a slope until the algorithm reach the lowest point on that surface. It is an iterative algorithm it will start on a random point in a function surface and travel down step by step until it reach the lowest point. A common feature of gradient descent and stochastic gradient descent (SGD) is that one set of parameters is updated after each iteration. The aim of this is to gradually minimize errors.

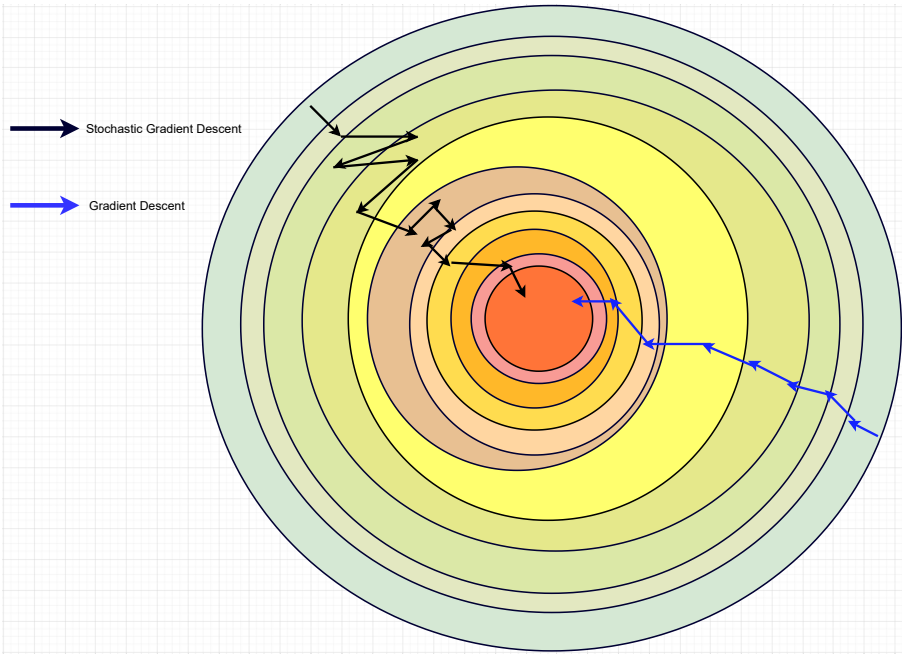

Figure 2: SGD compared to GD.

What sets the two methods apart is, among other things, that in GD all data points are run through before an update can be made to the parameters in an iteration. In SDG, only one training sample is needed before the parameters are updated.

Figure 2 above illustrates the route of the two methods at the lowest level in the graph. It shows that GD has a more direct route to the lowest level while SDG has a more messy path to the lowest level. With large data sets, GD often takes too long because it only updates the parameters after a complete scan. Here, SDG is a faster method of optimizing the processing of large data sets.

When operating on clusters, the possibility of executing calculations on the data-set can be distributed so that each cluster calculates different parts of the set without affecting the final outcome. The advantage of this is that the speed of the calculation will increase with the number of clusters assigned. This concept is called parallelism or concurrent programming, the objective is to speed up the process and share the processing load. Distributing a set of tasks to over a set of computing units is known as load balancing. Note that there is a difference in distributing the calculation crocess into clusters and arranging the dataset into clusters.

We have tested the training-set by using Kmeans-fitting presented in colloquium 10, we used to see how the dataset would be delegated to a different number of clusters. There are tree interpretation points we can search for when interpreting a kmeans graph. The average distance from the centroid (center of cluster) to the cluster edges. The maximum distance from the centroid to the "furthest away" data point in the cluster. Separation between clusters according to each other, meaning that that the clusters should not overlap or have extraneous data points in other clusters.

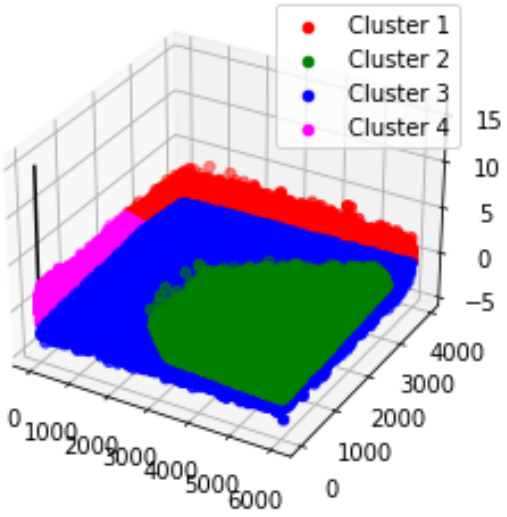

Figure 3: Train dataset with 4 clusters.

The figure above illustrates how the dataset is distributes between 4 clusters. If we look at the three interpretation methods listed above, we can see that some of them are relevant to the interpretation of this. Cluster 3 (blue), for example, may have unnecessarily high average distance to centroid, which means that the dataset is less compact and show grater variability.

2.2 Architecture

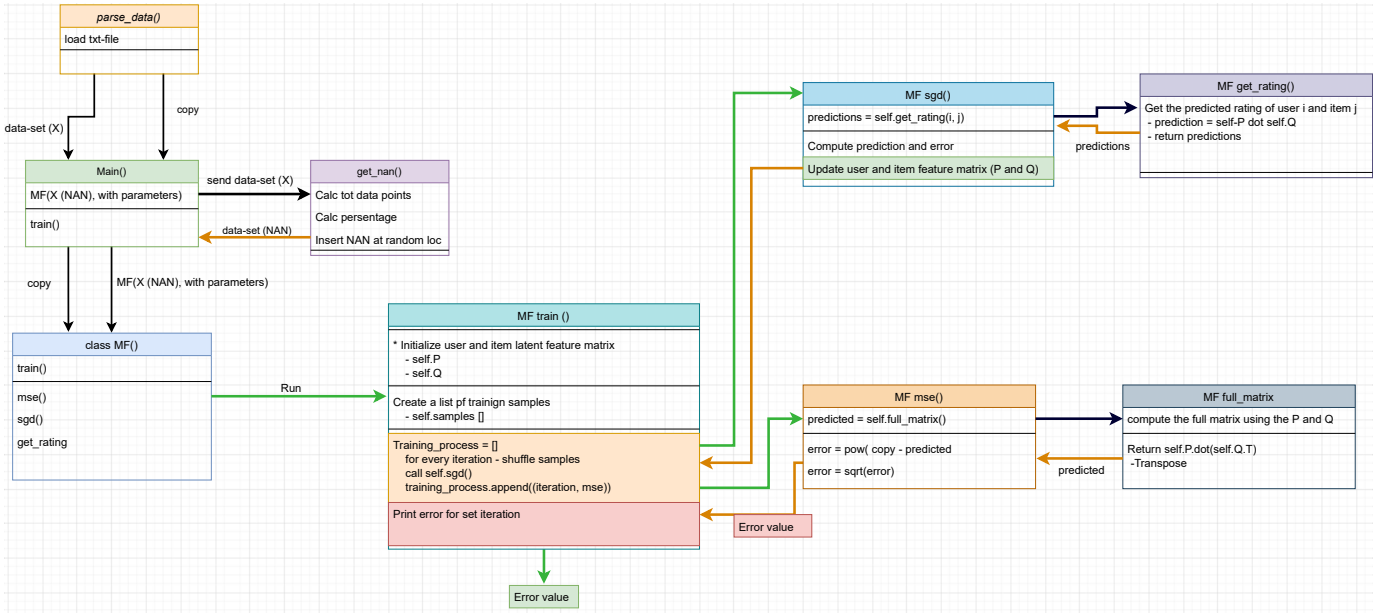

Figure 4: Architecture of implementation[3].

The implementation of the implementation is illustrated above.

2.3 Reflection on decisions

In this assignment, the method of approach is to use a stochastic gradient descent implementation (inspired by source [3]) to calculate the errors between the original matrix and the altered matrix with NAN values. The reason for this approach is that SGD have the advantage of preforming better while handling large data sets compared to gradient decent GD method. The reason for running the implementation on azure databricks is so that we can implement concurrent calculation on the matrix by delegation task to a certain amount of clusters. In this way, we can further optimize the processing of the calculations on the large data-set.

The path to the lowest error value in the graph is more messy in SGD compared to the GD method. This is a trade-off to compromise for the large data sets given. Te desired objective is to train the weights with the large data-set (train.txt) using a SGD-algorithm combined with the usage of clusters in azure databricks. We want to compare the calculated estimate with the original values to see how well our implementation calculate the missing values with different parameters.

In azure, we want to use jobs and databricks to create clusters for the processing of the calculations. If we look at figure 5 below, it shows an example of how this job distribution can be arranged between 4 different clusters.

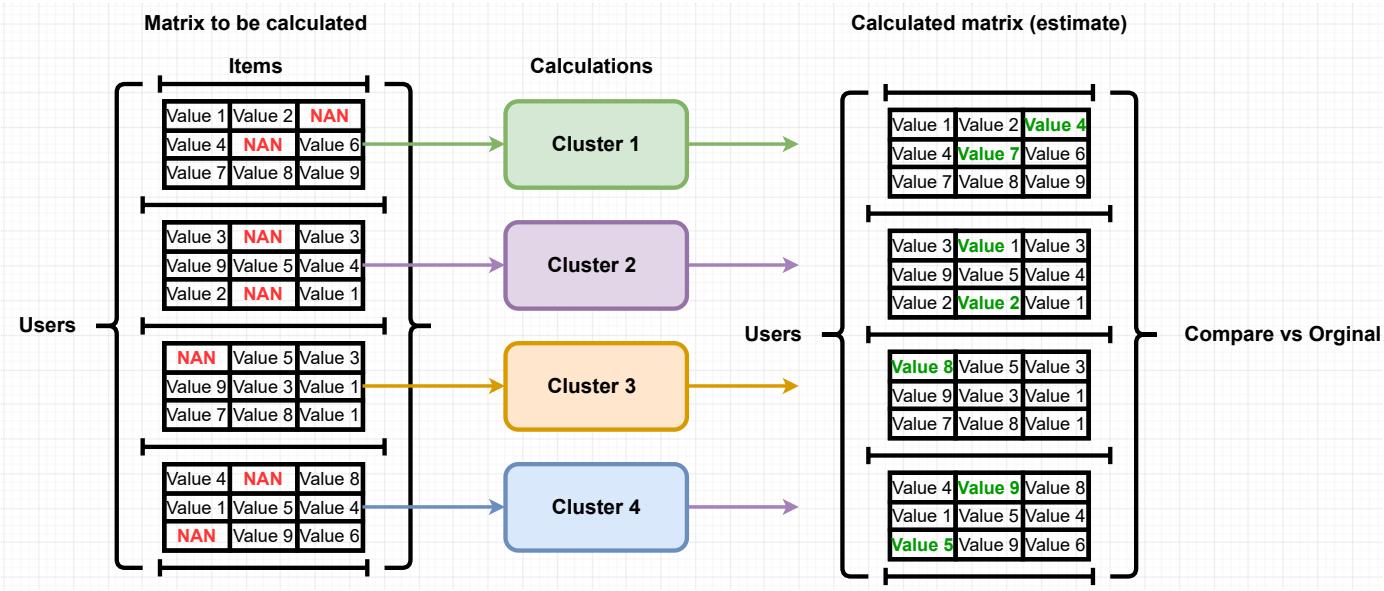

Figure 5: Architecture of potential cluster distribution.

The matrix to be calculate will have s certain percentage of NAN-values randomly allocated within the matrix. The matrix will be spitted into 4 approximately even sized matrix-es and assigned to a unique cluster each. The cluster will then perform the calculations on the given matrix-set and return in in the correct order with the calculated values. What remains now is to compare the calculated matrix with the original one and get an error value.

2.4 Implementation

The implementation in this assignment is based on and inspired from the [3] source. This section explenation is also based on how the source [3] explain the implementation.

Implementing stochastic gradient descent(SGD) based matrix completion using python.

The *parse-data* function loads an return the text file, a copy is made to so that we can compare the altered text file with the original later. The other data-set is sent to the *get-nan* function. This function inserts a certain amount of not a number (NAN) data-points into the given data-set. A simple calculation is done by calculating the total number of data points and then the percentage of the total data-points. This calculated amount is inserted into the data-set at random locations.

The data-set is now incomplete with a certain percentage of NAN data-pots. The set is sent as the array argument to the Matrix Factorization *MF()-class* along with other arguments as learning rate (), iteration number and the data-set copy. The *X.shape* is declared as number users and number items as discussed in technical background.

MSE or mean square error. The squared loss penalizes the difference between the actual outcome (copy) and the outcome estimated by choosing values for the set of parameters x,y (predicted). This error value is raised by 2 and taken square root, producing the mean square error. Different parameters will influence this value, the number of data in the set and the percentage of NAN will make a big impact on the MSE value.

In the example from figure 1 i TB, we have U of users, and D of items. The true size of the matrix that contains all the data-points values as $\mathbf{X}$ of size $U * D$. We want to discover $\mathbf{K}$ latent features. We need to find two matrix **U-users** (of size U×K) and V-items (of size D×K) such that their product approximates R: [3]

$$\mathbf{X} \approx \mathbf{U} - \mathbf{users} \times \mathbf{V} - \mathbf{items}^T = \hat{\mathbf{R}}[3]$$

In the train function, *U-users* and *V-items* is used to initialize user and item latent feature matrix. **U-users** is the association strength between the user and the data-point while **V-items** is the association strength between the item and the data-point. In the *train function* a list of training samples is created. The data-set is shuffled and each sample contain tree values, at every i and j (x,y-coordination) there is a shuffled training value (data-point). The training process use these samples to train in the range of the given iteration rate. The samples are randomly shuffled and the stochastic gradient descent(*SGD-function*) is called.

The objective is to minimize the errors to the best extend possible. Stochastic Gradient Descent calls the get rating function to get the predicted values. T is a set of tuples, every tuple is in the form (u_i, d_j, r_ij), meaning that T contains all the observed user-item data-points pairs as well as the a associated ratings. T can be described as *training data*.
SGD-function will compute a *predicted value* for the tree values in each sample trough the *get-rating function*. To get the prediction of a rating of an item d_j by u_i, we can calculate the dot product of their vectors: [3]

$$\hat{r}_{ij} = p_i^T q_j = \sum_{k=1}^{K} p_{ik} q_{kj} [3]$$

This is done and returned to the *SGD-function* as predictions, the training error e is calculating by subtraction the prediction from the training value r (data-point). The e value is used to update user an item latent feture matrix

Now, we have to find a way to obtain P and Q. A way to approach this is to generate the two matrix's with some values and then calculate how different their product is to M, and then minimizing the differences between the two matrix's by iteration. This concept is what is known as **gradient descent**, aiming at finding a local minimum of the difference [3].

The error referrs to the actual differences between the true and the estimated sets. Fore each pair, the function below can be used to calculate the difference:

$$e_{ij}^2 = (r_{ij} - \hat{r}_{ij})^2 = (r_{ij} - \sum_{k=1}^{K} p_{ik} q_{kj})^2 [3]$$

This is calculated in the *mse-function* which return the non-negative square root of the error value. The estimate can be both higher and lower than the actual data-point value, the **squared error** is therefor considered. The objective is to minimize the squared error. To do this the implementation needs to know which way (direction) to modify the values p_ik and q_kj (gradient of the current values).

This will results in the obtained gradient, the update rules can be formulated for both values p_ik and q_kj [3]. Here, the alfa is the rate of approaching the minimum, also known as the **learning rate**. This is the steps that determines the step size at each iteration while moving toward a minimum of a loss function. If the steps are to large, the function risks missing the minimum value. On the other hand, if the learning rate is to small, the function risk not reaching the minimum value.

The training process append the *mse* for every iteration end the square root error is returned for every x-amount iteration. The *mse* function will return the weights when all iterations for the train function is completed. The test function will take these argument (trained weights) and present them as the weight for the testing. The testing is done almost in the same way as the training, but without calling the *sgd()*-function for training and updating of the weights.

3 Experiments

The implementation of our solution in Azure databricks was unsuccessful. We did not find a solution to use the created cluster to partition the calculation, making it possible to calculate the large training sets and the test set within a reasonable amount of time. Since we faced this problem, we run the test set for the

implementation.

The results below show how the implementation decrease the error value for the specific NAN persentage inserted in the implementation. The test is done with the test document, the train document was to large to run without parallel programming and the creation of clusters. This was done to get results on how the implementation react to different percentage of NAN data-points since running it on the original test document was extreme time demanding without partitioning the tasks to a number of clusters in databricks.

We now want to find the ideally percentage of NAN-values in the dataset, before we run it on the test-files. Some test-runs indicate that the learning-rate need to be small for the values to not become to great. The learning is set to be *0.0000001* steps for both the train and the test functions.

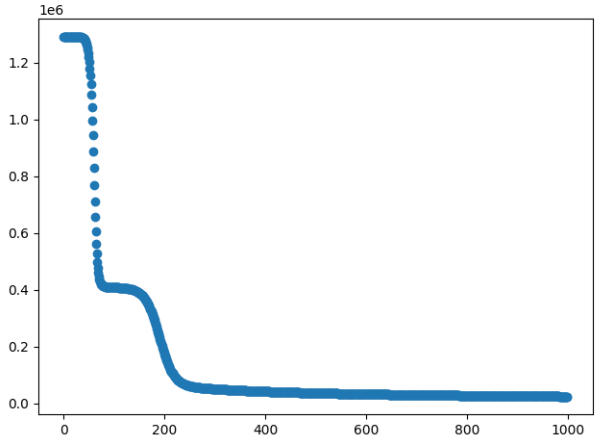

Figure 6: y = (error value) per x = (iteration) for 1 percent NAN percentage

The figure above shows how the error value decreases for the training of the weights for 1 percent NAN-data-points. The calculations stops at approx error value = 24,224, starting at error value 1,234,652 over 1000 iterations. We want to compare the calculated data with the original data.

Estimated	Orginal
[[2.07755638e+01 1.50791679e+03 8.08283597e-02]	[[1.00000e+00 1.56600e+03 5.76852e-02]
[2.51293204e+01 1.83630322e+03 9.84276110e-02]	[1.00000e+00 1.90700e+03 -3.47872e-01]
[3.07615346e+01 2.26764938e+03 1.21543203e-01]	[1.00000e+00 2.35500e+03 5.76956e-01]
...	...
[5.96263275e+03 3.41933767e+03 2.92026142e-01]	[6.04000e+03 3.47100e+03 1.12568e-01]
[5.96602013e+03 3.67354063e+03 3.05649986e-01]	[6.04000e+03 3.73500e+03 2.05528e-01]
[5.96621916e+03 3.68896215e+03 3.06476373e-01]]	[6.04000e+03 3.75100e+03 2.51798e-01]]
Total value error 24224.91483	

Figure 7: Compared estimated values with original values for 1 percent NAN at start and end of matrix

I the figure below (figure 8) we can see how the error value calculation changes for 5 percentage NAN.

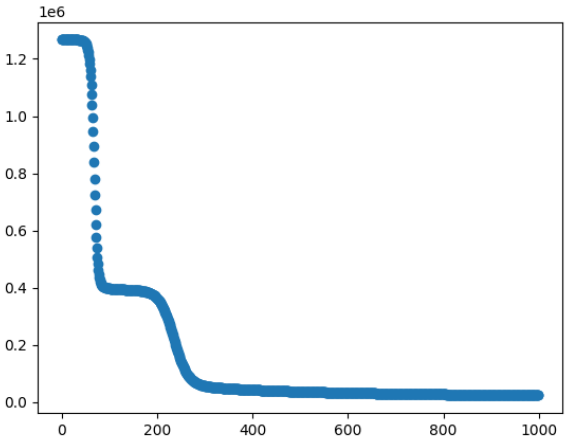

Figure 8: y = (error value) per x = (iteration) for 5 percent NAN percentage

At 5 percent NAN we get almost the exact same outcome as for 1 percent, stopping at an error value of 24,224 after 1000 iterations The same pattern reoccur for the error value when the NAN percentage is set to 10 percent. Here we have tried with 1200 iterations just to get the full extent of the flatten curve. (figure 9).

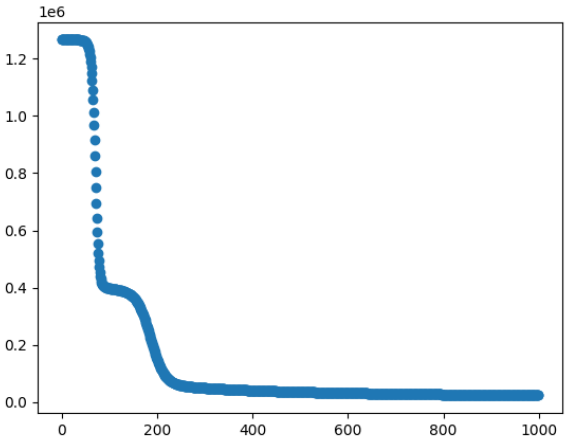

Figure 9: y = (error value) per x = (iteration) for 10 percent NAN percentage

With 1200 iterations, the error value ends up at 23,806. The course of the graph appears to be more or less the same as at 1 and 5 percent.

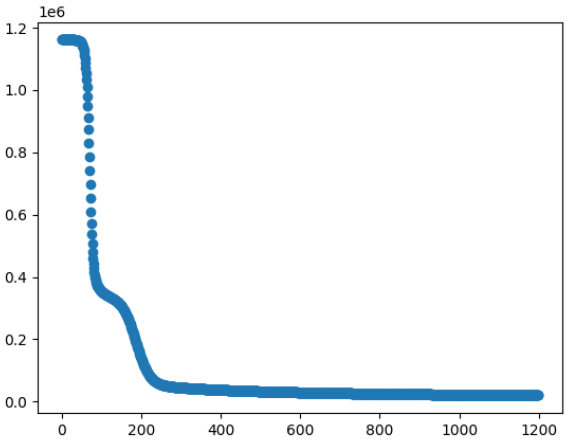

Figure 10: y = (error value) per x = (iteration) for 20 percent NAN percentage

20 percent NAN gives a minimum error value at 19,785 with 1200 iterations.

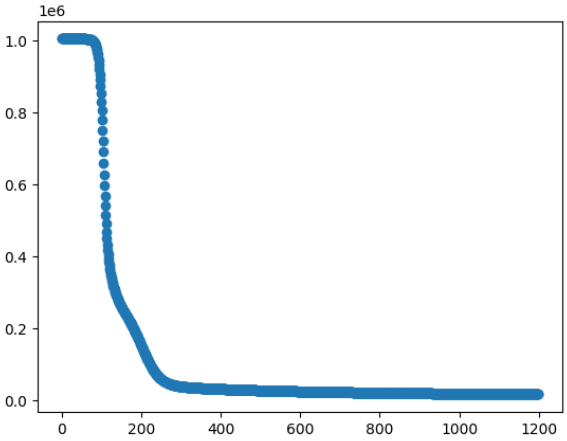

Figure 11: y = (error value) per x = (iteration) for 40 percent NAN percentage

40 percent NAN gives a average minimum error value at 17,429 with 1200 iterations.

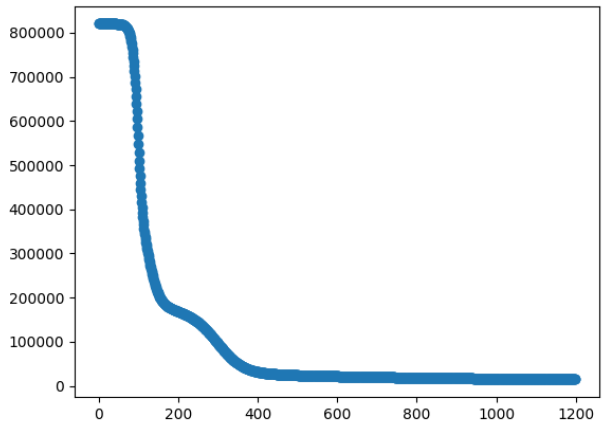

Figure 12: y = (error value) per x = (iteration) for 60 percent NAN percentage

60 percent NAN gives a minimum average value error at 14,785 after 1,200 iterations.

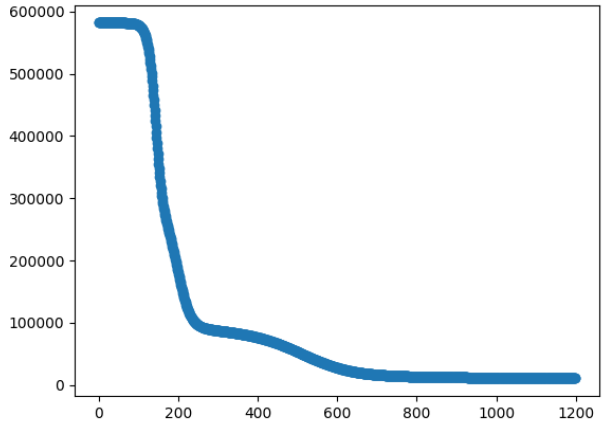

Figure 13: y = (error value) per x = (iteration) for 80 percent NAN percentage

80 percent NAN gives a minimum average value error at 10,718 after 1,200 iterations.

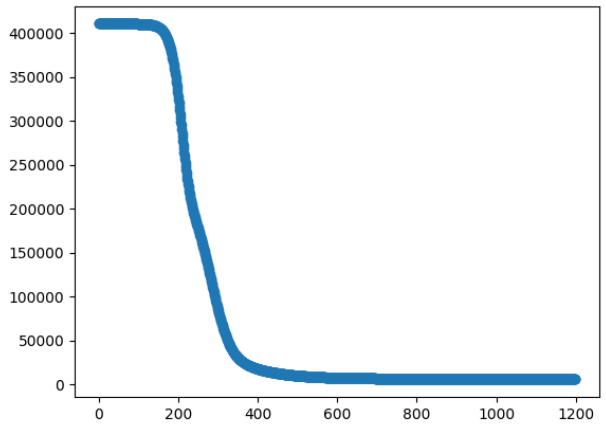

Figure 14: y = (error value) per x = (iteration) for 90 percent NAN percentage

90 percent NAN gives a minimum average value error at 5,965 after 1,200 iterations.

3.1 Drafting the experiments

We start the training on the test document with 1 percentage of NAN-values with 1K iterations. Then we increase the percentage of NAN. The objective is to use the trained weights to perform calculations on the test data-set. A common feature for all the graphs is that the value error drops significantly.

If we look at the comparison between estimated and original in figure 6, we can see that some of the estimated values (bottom half) are pretty close to the originals, while some other values (top half) are not all that close. All these errors (positive and negative) sums up to be the total value error, which is a good indication of how well the algorithm perform with the given parameters. We want the error value to be as close to zero as possible, indication a perfect prediction of the values. The starting position will also naturally affect which error value we end up with, this starting value will vary as each time we run the process.

20 percent NAN gives an error value at 19,785 with 1200 iterations while 40 percent NAN gives a minimum average value at 17,429. 60 percent NAN gives 14,785, 80 percent gives 10,718 and 90 percent gives . The trend is that the higher the percentage of NAN the lower the minimum average error value becomes.

4 Conclusion and future work

The stochastic gradient decent logic seems to be working to some extend. The training of the weights result in that the error value is reduced from over 1 million to around 23,000 for percentages 1 to 10. When the error values are compared directly, some estimates are really good while some are far off target. Since this is a large data set, each value with a significant error will contribute to the collection of a significant total error value. This means that an error value must be reflected against how large the data set is, an error value of 23,000 can be discussed as being a medium to poor value. It would have been desirable to get below 1,000 in error value on this training set.

For further work the objective in the assignment is to distribute the calculation processes between clusters using azure, this was not successfully completed due to the failure to deploy jobs in databricks. The implementation in this assignment have been tested locally, and the results show that a 90 percentage of NAN gave the best value error of 5965 for the test dataset.

5 References

1. Mean squared error [Internet]. Wikipedia. Wikimedia Foundation; 2022 [cited 2022Nov9]. Available from: https://en.wikipedia.org/wiki/Mean-squared-error

2. Gupta A. Mean squared error : Overview, examples, concepts and more: Simplilearn [Internet]. Simplilearn.com. Simplilearn; 2022 [cited 2022Nov9]. Available from: https://www.simplilearn.com/tutorials/statistics-tutorial/mean-squared-error

3. Matrix factorization: A simple tutorial and implementation in python: Albert Au Yeung [Internet]. Matrix Factorization: A Simple Tutorial and Implementation in Python — Albert Au Yeung. [cited 2022Nov10]. Available from: https://albertauyeung.github.io/2017/04/23/python-matrix-factorization.html/