COMP 551

Group 25: Mini Project 2 Write-Up

Wassim Wazzi

Cesar Arnouk

Andrew Geday

Abstract

This Mini Project consisted of two parts. In the first part, we were given a diabetes dataset (split into test, validation and training) that required no pre-processing. First, we had to choose a learning rate for the logistic regression that fits our model the best, and we found that for α=0.0002 and max_iters=40,000 were the best parameters. Adding mini batch SGD and momentum helped with the convergence speed of our model, and also improved the validation accuracy.
The second part consisted of text classification, where the main challenge was mapping raw text to features. We implemented a base model which achieved an accuracy of 0.716. We also tried to go above and beyond in hyperparameter tuning and achieved an accuracy of 0.743.

Introduction

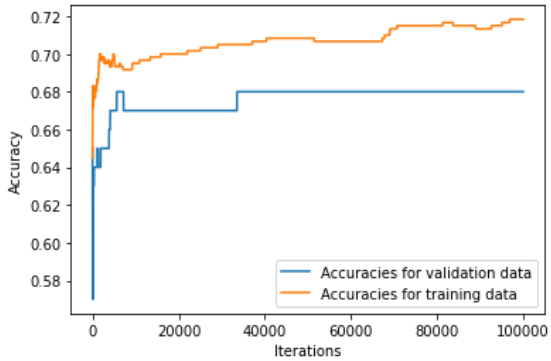
The first task was to run the given implementation of the logistic regression, which served as a baseline for the following parts of the project. From this implementation, the first objective was to find an appropriate learning rate and number of iterations for which the model converges to a solution. We began by testing the model for different values of the learning rate, in order to visualize for which value it was converging (Table 1, Appendix). Once we got our model to fully converge (the norm of the gradient was sufficiently small), we tested the model for training and validation accuracies, taking the previously found learning rate as our parameter. This allowed us to observe the number of iterations at which the model was converging, which occurs when the validation accuracy stabilizes and stops increasing.
Afterwards, we had to add mini-batch stochastic gradient descent to our initial model, and test it for different batch sizes (8, 16, 32, 64, 128, 256, 600). Similarly, we tested the model with training and validation accuracies for each batch size to observe the point of convergence (Table 2, Appendix). At first, our algorithm was predicting at each iteration inside the chosen batch , but we realized that it was running too slowly. We then decided to add a new parameter (step, set to 1000 by default) in order to get a prediction every 1000 iterations. This improved the run-time complexity of the model without jeopardizing the visual representation.
The last improvement to the model was to add momentum to the gradient descent implementation. For this part, we used two different approaches: first using the basic formula we saw in class , then using a more improved one. When we first ran the algorithm using the regular formula, we noticed that the results were not changing as expected, and got nearly the same ones as for the mini-batch model (Table 3, Appendix). Some research and documentation were conducted that led us to find an alternate popular formula for the gradient descent algorithm using momentum. Once we tested our model with this newer version, we noticed improved results (Table 4, Appendix). For both approaches, however, the analysis was similar: we tested the model for training and validation accuracies using multiple values of beta (the momentum parameter), and compared visually the convergence speed and its respective accuracy.
Once all our previous models were correctly implemented and gave us the results we expected, one last experiment had to be conducted: we ran our gradient descent with momentum algorithm on the smallest and largest batch sizes, and compared the results to the previous algorithm (gradient descent with momentum on the full batch). The results were visualized similarly as before (training vs validation accuracy for different values of beta for each batch size: 8 (the smallest batch) and 256 (the largest)).

For the second part, using the resources given in the assignment description, namely the sklearn's text data tutorial (https://scikit-learn.org/stable/tutorial/text_analytics/working_with_text_data.html). The idea is to

convert raw text to features that can be used for classification. The way this was done is by using sklearn's feature_extraction library. The library provides a function for counting the frequency of each word in the text, namely the CountVectorizer() function. We also use something called "Term Frequency times Inverse Document Frequency" through the TfidfTransformer() function. This would achieve downscaling on the weights for words that occur in many documents in the corpus and are therefore less informative than those that occur only in a smaller portion of the corpus. Once we process our raw text data using those 2 functions mentioned above, we get 152673 features for each row of data. Hence, we run sklearn's LogisticRegression model on our data and we achieve an accuracy of 0.716.
This was a basic version of our model. After this was done, we moved on to try and implement more preprocessing in order to achieve higher accuracies for our model by using sklearn's GridSearchCV model which performs a non-exhaustive grid search on the hyper-parameters of the model in order to tune them as much as possible. Hence, we get back tuned hyperparameters and plug them into our model to get an improved accuracy of 0.743.

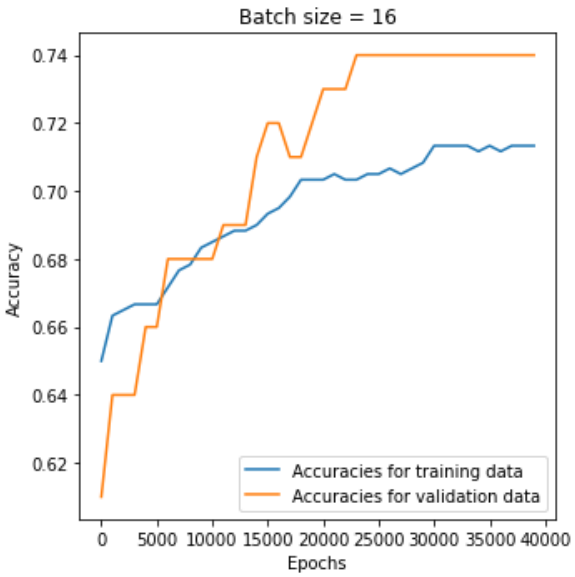
Results

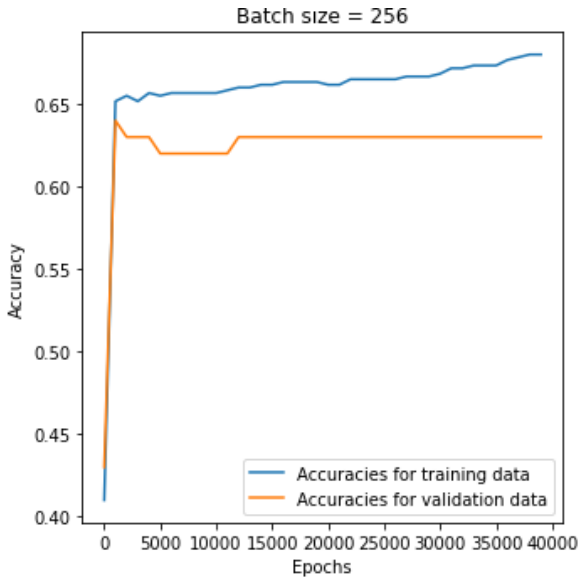
- The first thing we noticed, was that using large learning rate values, caused the gradient to oscillate back and forth, and we never reached a local minimum, however, with smaller learning rate values, we saw that the norm immediately went down to values close to 0, and stayed there, without oscillating, and for a value of 0.0002, we had the best performance on the validation data. To determine convergence, we still had to look at the test vs training accuracy, and we noticed that for α=0.0002, the model converges to an optimal solution after 40,000 iterations, and then starts to overfit to the training data (improves accuracy on validation data only); this can be visualised in the plot below.



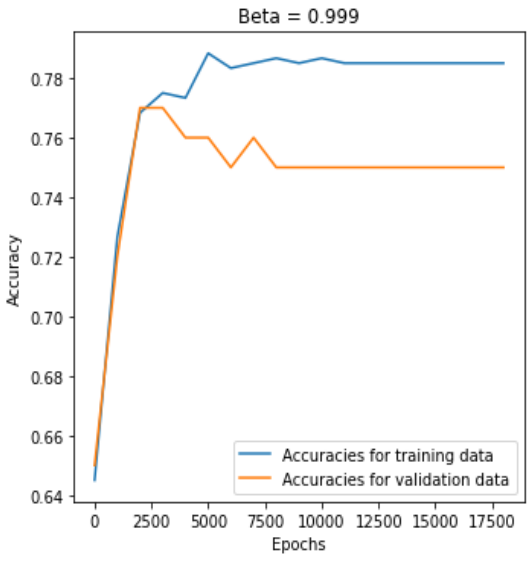
- Using the optimal learning rate and max iters values, we ran stochastic mini batch gradient descent with different batch sizes. We saw that for smaller batches, we had higher accuracies, but slower convergence (the algorithm did not converge for small batch sizes after 40,000 epochs). At first, we were surprised by these results, but after discussing and thinking about it, we realised that since we are using the number of epochs (40,000epochs) as a termination condition, no matter the batch size, we will loop through all the data points 40,000 times, unless the algorithm fully converges earlier. Given that fact, it is not surprising that smaller batch sizes give higher accuracy, since we compute the

gradient many more times (600/batch_size number of times more) than the fully batched baseline. The highest validation accuracy of 74% was achieved for batch=16, compared to 68% on the fully batched implementation. And, the fastest convergence was at around 500 epochs for batch=256, compared to 7500 for batch=600.
The plots below are for the highest validation accuracy and fastest convergence, the result for each batch size can be seen in the notebook.

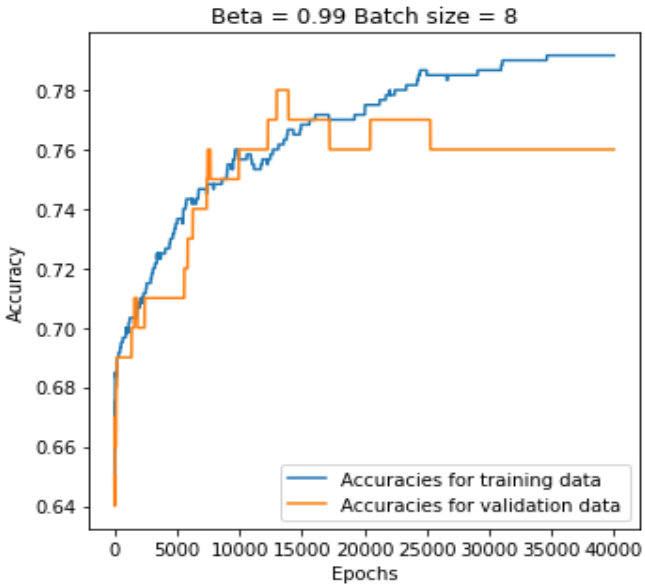




- When adding momentum to our gradient descent implementation, using the formula taught in class, we were not able to visualize any difference in the convergence speed or the validation accuracy for all beta values less than or equal to .99, at beta=0.999, the model converged very quickly before it started overfitting, and for beta=0.9999, the model did not converge (Plots here).
  To get improved results, we used a different formula for momentum (Gradient Descent Implementation). We were able to clearly see that as beta increased, the validation accuracy at the point of convergence also increased, and the algorithm converged to an optimal solution faster.



- Finally, we added momentum to the SGD implementation (momentum formula from class). We noticed that using batch_size=8, generally gave better results (see Figure 3).. With other values, the model either did not converge, overfitted, or didn't perform as well. Here is the best validation accuracy we got (78%).



For part 2, as stated in the introduction, the basic version of our model resulted in an accuracy 0.716. This model consisted only of using sklearn's LogisticRegression model on our preprocessed data (which was preprocessed using sklearn's CountVectorizer() and TfidfTransformer() functions). We then went on to improve our model by tuning the hyperparameters using sklearn's GridSearchCV model and used the resulting parameters in sklearn's LogisticRegression to get an improved accuracy of 0.743.

Discussion and Conclusion

- The first discussion point that came up was whether we had to use the number of iterations or the epochs when we were analyzing our mini-batch model. We noticed that the use of epochs was more adequate because it represented the number of times the algorithm went across all the data inside a batch, whereas the number of iterations would count the number of times the gradient was computed, which was kind of irrelevant in this case. In addition, this allowed for a more fair comparison across different batch sizes.
However, this seemed to defeat the purpose of the mini-batch algorithm, since it is meant to be used for very big datasets, where we cannot iterate through all data points, so in reality, we would not loop over the dataset the same amount of times for different batch sizes. Since our dataset was relatively small, we were able to do the comparison in terms of epochs.

- Another important challenge we encountered was whether our implementation of the gradient descent with momentum was correct, since the results we first observed were very close to the ones we had for our original model. Once we figured out the implementation was valid, we looked for a way to improve it. We found an alternative formula to compute the moving average which increased the accuracy of our results.

- We noticed that sometimes, we were receiving higher accuracy on the validation data than the test data, this means that using the validation set alone is not optimal, and it would have been better to use cross-validation by combining the test and validation data, to reduce the random improvements on seen on the validation set, however, this would be too time consuming, and computationally expensive and slow

- For part 2, the tutorial for text data provided in the assignment description was complete enough to cover almost all aspects of text classification using logistic regression. The accuracy of 0.716 achieved after solely following a quick tutorial is impressive, and we tried to take a step further to improve the model and achieve an accuracy of 0.743.

- We discussed that in order to improve the model further, we should add even more preprocessing on our raw data such as cross-validation or word embedding.

Statement of Contribution

Cesar - Mainly worked on implementation of stochastic mini-batch in part1, and implementing the basic model in part 2. Worked on Write-up

Wassim - Implemented the methods in the Logistic Regression class for gradient descent, and tried to go further than the basic version in part 2. Worked on Write-up

Andrew - Focused on plotting and visualising the results, also added the momentum implementation. Worked on Write-up

Appendix

	validation accuracy	training accuracy	learning rate	max iters	gradient norm
6	0.68	0.718333	0.00020	100000.0	0.032481
7	0.68	0.708333	0.00010	100000.0	0.034308
8	0.67	0.701667	0.00005	100000.0	0.035262
5	0.63	0.665000	0.00030	100000.0	26.548212
0	0.52	0.550000	0.05000	100000.0	66.584983
2	0.52	0.551667	0.00500	100000.0	66.765607
4	0.52	0.573333	0.00050	100000.0	56.951204
1	0.50	0.538333	0.01000	100000.0	66.311542
3	0.50	0.551667	0.00100	100000.0	65.000156

Table 1: Gradient norm, validation and training accuracies for different values of the learning rate

	validation_accuracy	training_accuracy	gradient_norm	iterations	batch_size
1	0.74	0.713333	8.269434	1480000.0	16.0
3	0.70	0.681667	16.036704	360000.0	64.0
4	0.68	0.721667	6.584158	160000.0	128.0
6	0.68	0.706667	0.034686	40000.0	600.0
0	0.67	0.690000	57.390731	3000000.0	8.0
2	0.67	0.760000	9.938512	720000.0	32.0
5	0.63	0.680000	27.629272	80000.0	256.0

Table 2: Gradient norm, training and validation accuracies for different batch sizes

	validation_accuracy	training_accuracy	gradient_norm	iterations	batch_size	beta
0	0.68	0.706667	0.034686	40000.0	600.0	0.0000
1	0.68	0.706667	0.034686	40000.0	600.0	0.5000
2	0.68	0.706667	0.034687	40000.0	600.0	0.9000
3	0.68	0.706667	0.034687	40000.0	600.0	0.9250
4	0.68	0.706667	0.034687	40000.0	600.0	0.9500
5	0.68	0.706667	0.034688	40000.0	600.0	0.9750
6	0.68	0.706667	0.034690	40000.0	600.0	0.9900
7	0.68	0.706667	0.034723	40000.0	600.0	0.9990
8	0.68	0.693333	1.262373	40000.0	600.0	0.9999

Table 3: Training and validation accuracies for different values of beta (first formula)

	validation_accuracy	training_accuracy	gradient_norm	iterations	batch_size	beta
4	0.76	0.768333	0.016002	40000.0	600.0	0.950
5	0.75	0.775000	0.008417	40000.0	600.0	0.975
6	0.75	0.781667	0.001962	40000.0	600.0	0.990
3	0.74	0.755000	0.020606	40000.0	600.0	0.925
2	0.73	0.751667	0.023583	40000.0	600.0	0.900
0	0.68	0.706667	0.034686	40000.0	600.0	0.000
1	0.68	0.715000	0.033199	40000.0	600.0	0.500

Table 4: Training and validation accuracies for different values of beta (second formula)

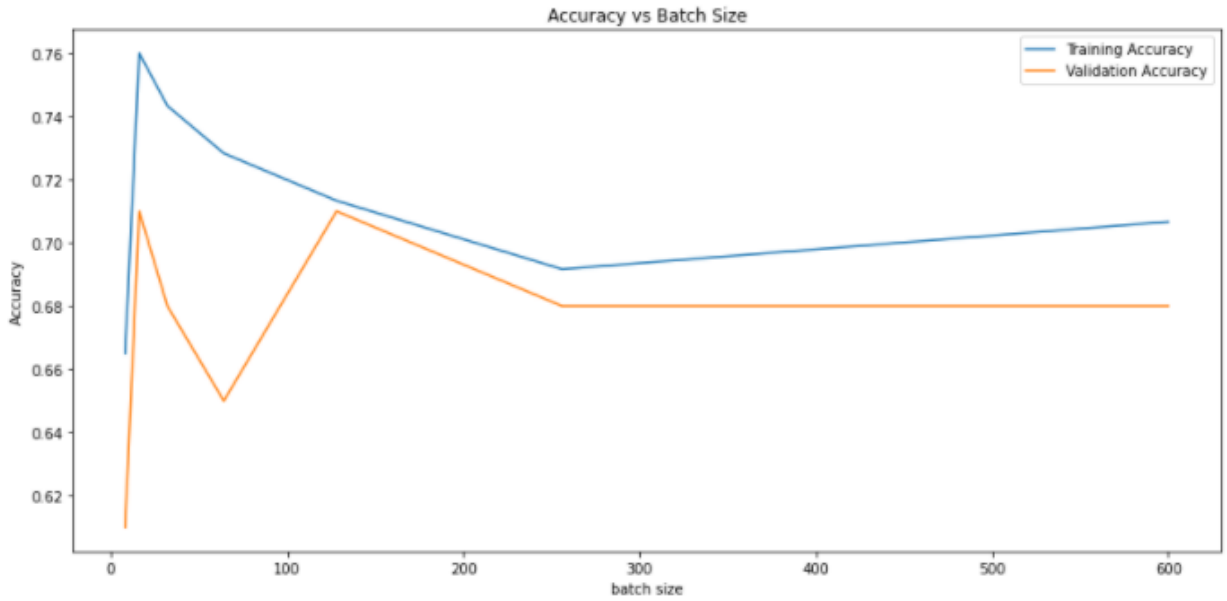


Figure 1: Training and validation accuracies for different batch sizes

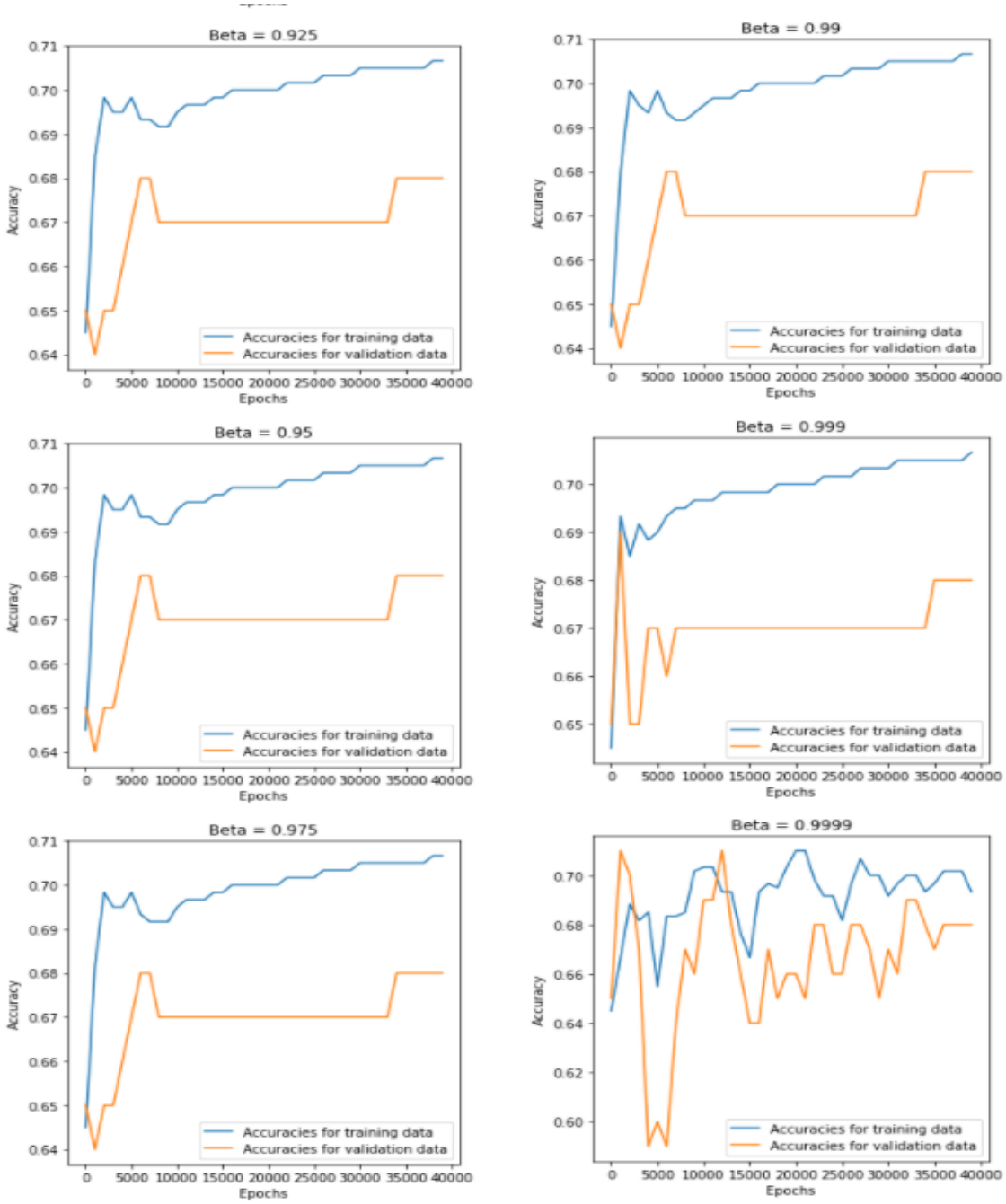


Figure 2: Training and validation accuracies for each beta value, using momentum formula seen in class

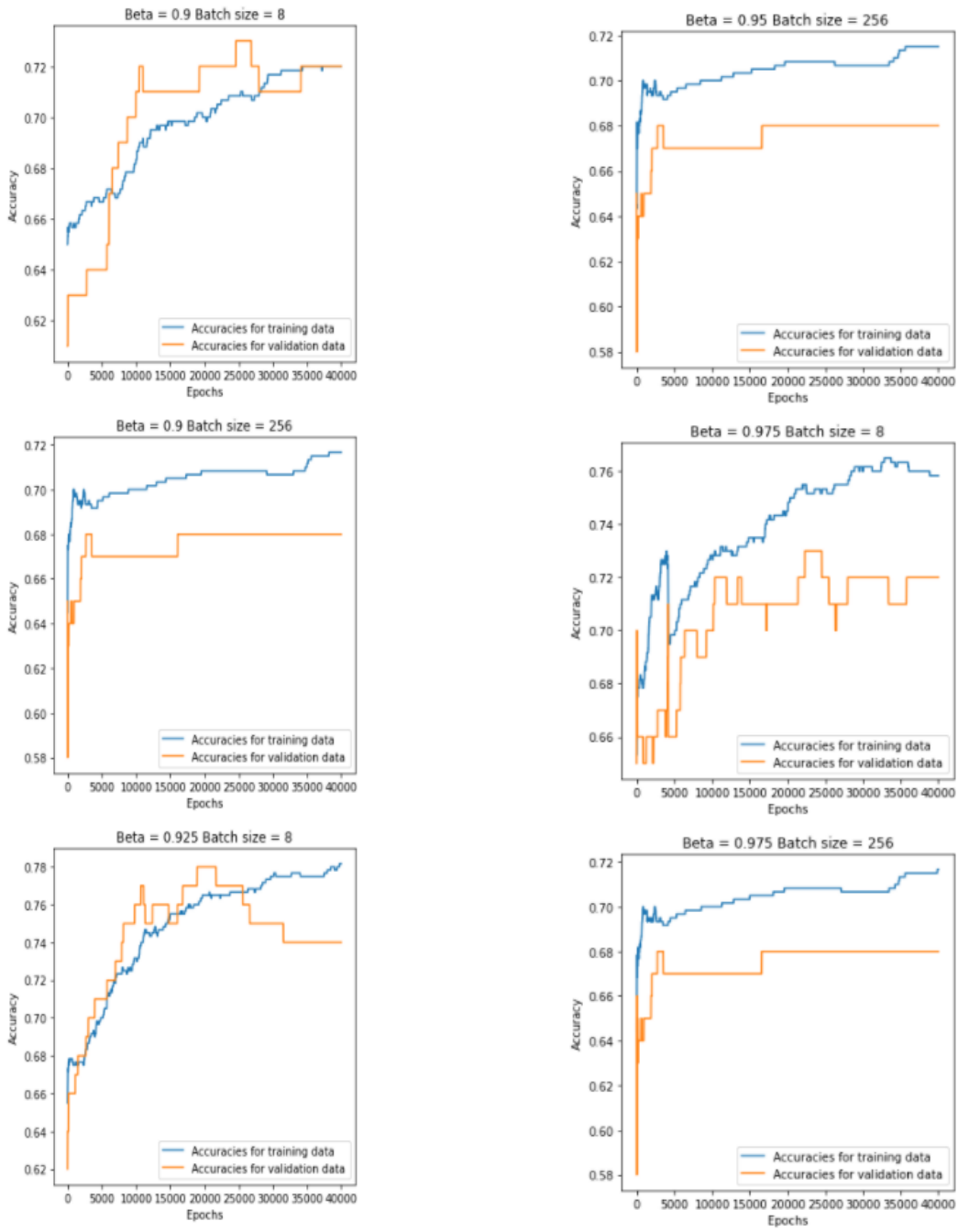


Figure 3: Some results of momentum SGD implementation.

N.B: Some plots and tables were not added from the notebook to the appendix for the sake of simplicity..

References

Gradient Descent in Python: Implementation and Theory (Mehreen Saaed)