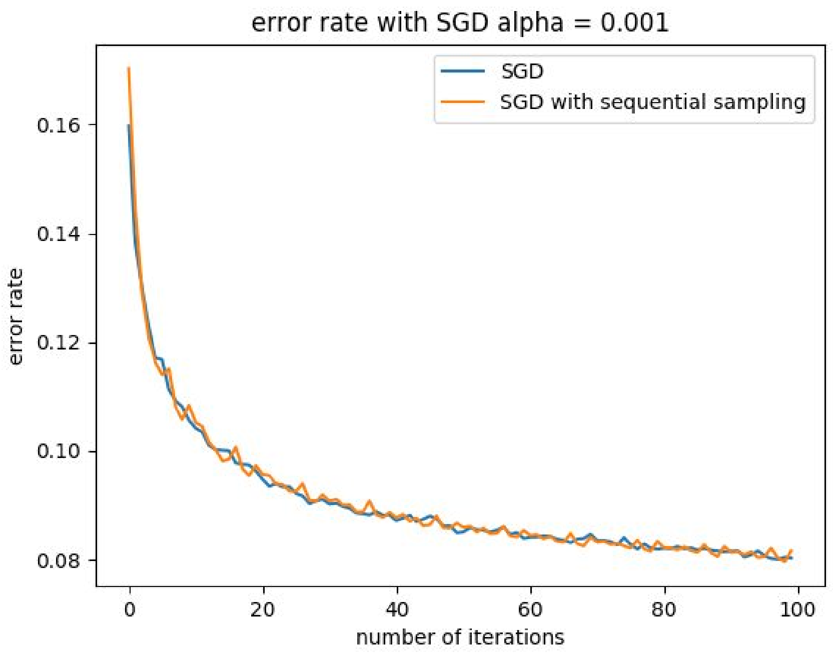
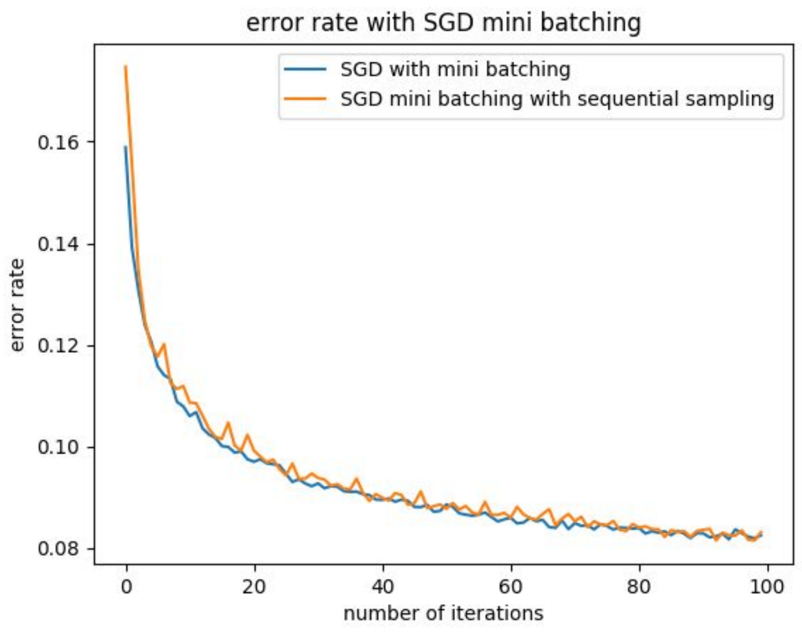
Scaling with SGD

Part 1 Implementation

Below is the result of SGD on the training data set. As we can see, the error rate decreases as we make updates on the weight vector. Each point represents 6000 update iterations.



`

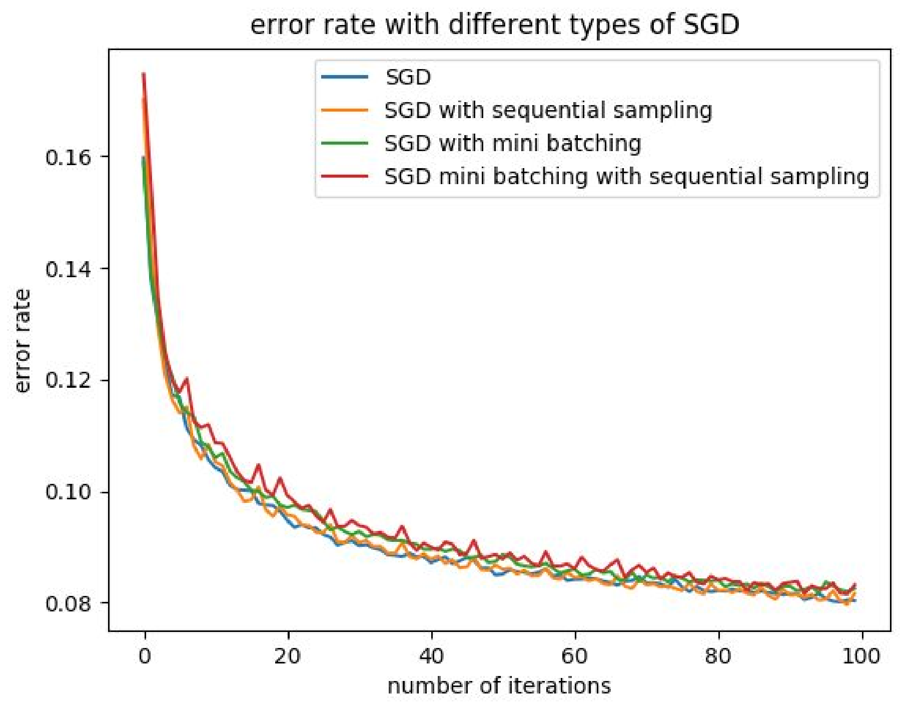


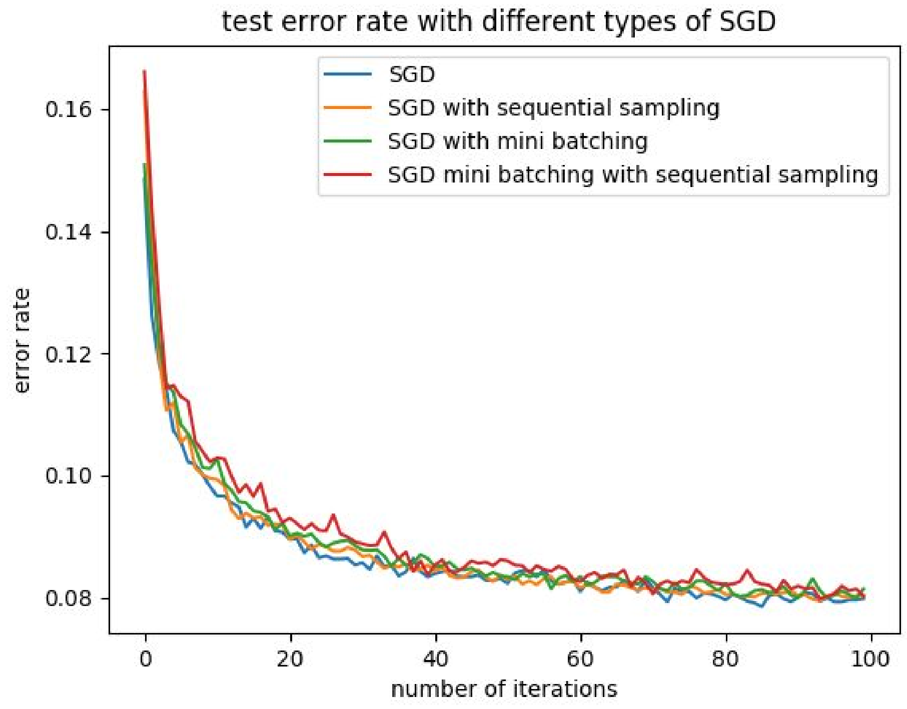
5.

Error rate	SGD	SGD with sequential sampling	mini batching	mini batching, sequential sampling
Training	0.0803	0.0816	0.0825	0.0831
Testing	0.0804	0.0804	0.0811	0.0803

As we can see from the table, SGD with mini batching reached nearly the same result as normal SGD. However, the SGD with mini batching took less computing time.

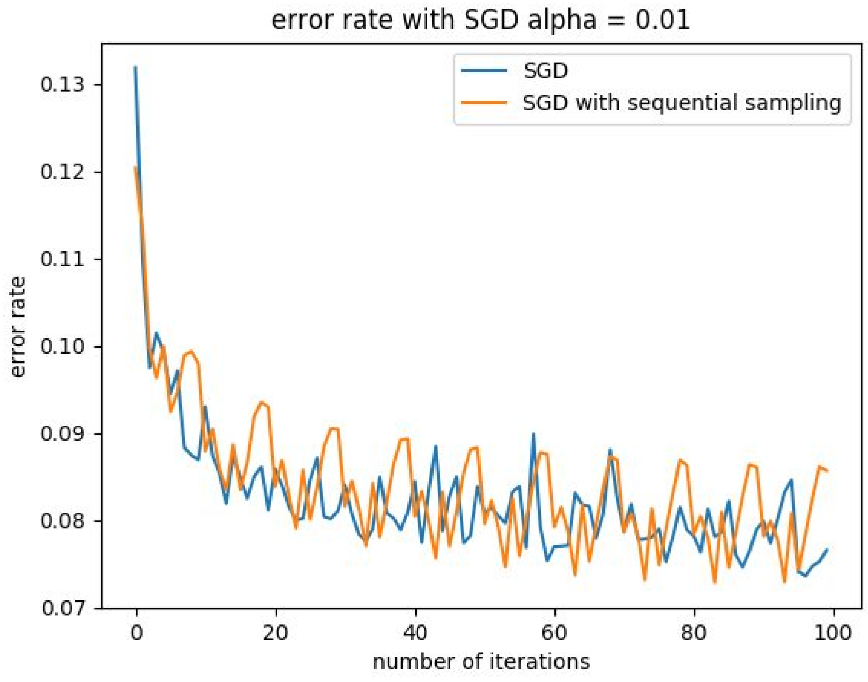
6. The two plots below shows the relationship between the error rate and the number of epochs. The first plot shows the error rate on the training data and the second plot shows the error rate on the testing data. As we can see, algorithm 1 and 2 reaches about the same error rate. However SGD with sequential sampling took less time to compute. Algorithm 3 and 4 were able to reach similar error rate. However, it is slightly higher than the error rate from algorithm 1 and 2. Algorithm 3 took less time to run compared to algorithm 2, and algorithm 4 took even less time compared to algorithm 3. In conclusion, algorithm 3 and 4 took less time, but algorithm 1 and 3 outputs better results.



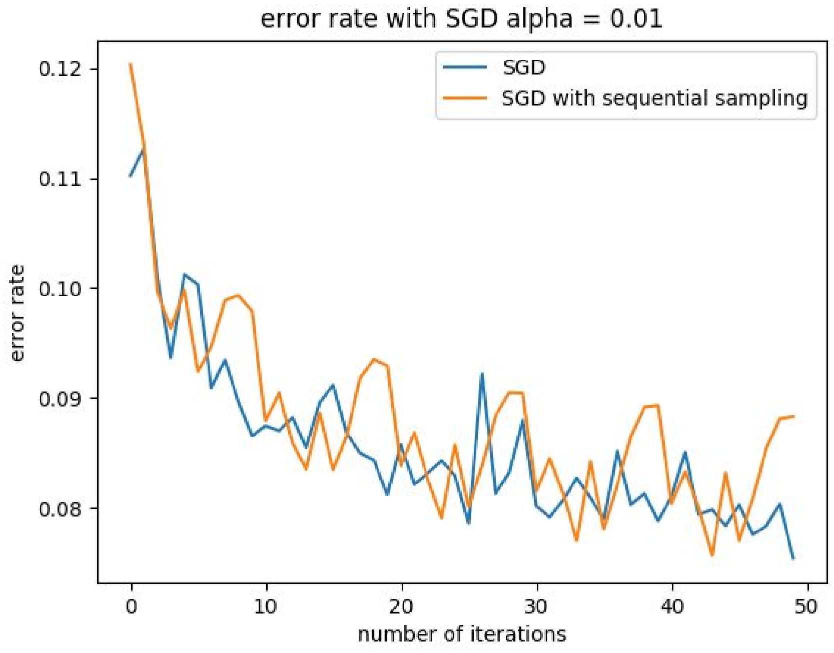


Part 2 Exploration or SGD

1. With a step size of 0.01, SGD (algorithm 1) outputs a training error 0.075 and a testing error 0.079 which is lower than the previous training error of 0.0803 and testing error 0.0804. Below is a graph of SGD error rate with alpha = 0.01 and 10 epochs.



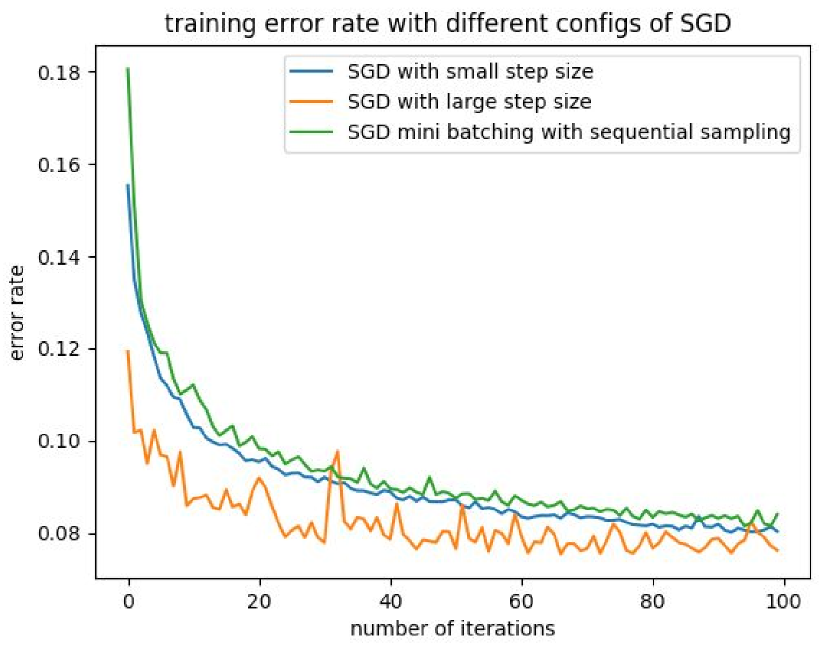
2. For SGD (algorithm 1), we can achieve a lower error rate with a step size of 0.01 after 10 epochs. This change of step size also resulted in a lower testing error. However, if we keep adjusting the step size to achieve an lowest training error, the final testing error is expected to increase due to overfitting of the training data.
3. With a step size of 0.01, SGD (algorithm 1) is able to generate a lower training error in 5 epochs than could be achieved in 10 epochs with step size given in part 1. The training error with 5 epochs and step size 0f 0.01 is 0.07543 while the error with 10 epochs and step size of 0.001 is 0.0803. Below is a graph of SGD error with alpha = 0.01 and 5 epochs.

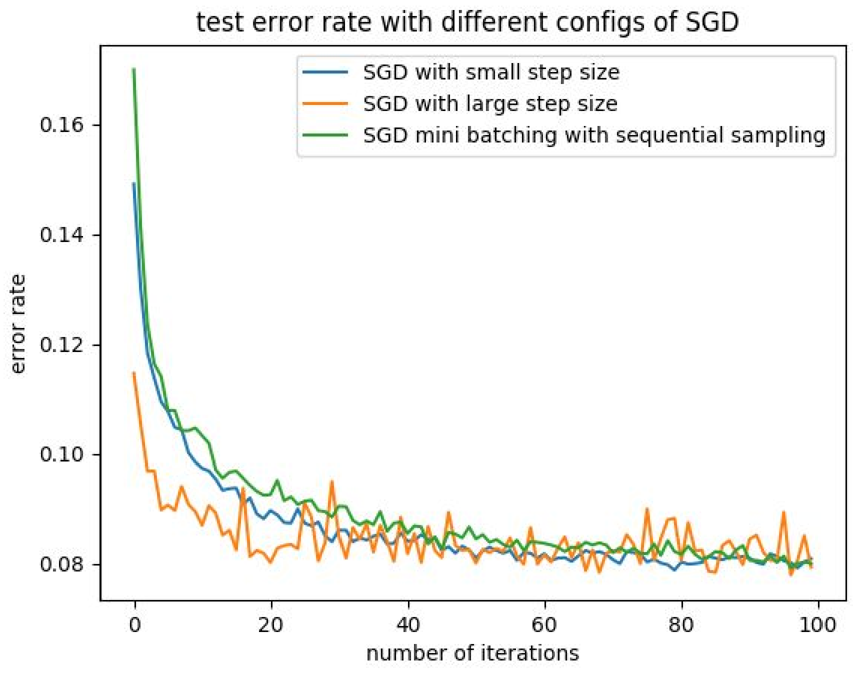


4. The error rate of various batch size and step size with 5 epochs are listed below:

Step size-> Batch size	0.01	0.05	0.1	0.5	1
8	0.0864	0.0882	0.1137	0.1313	0.1446
16	0.087	0.0835	0.0839	0.1153	0.1517
32	0.0939	0.0828	0.0824	0.1014	0.107
64	0.1007	0.0841	0.0806	0.0868	0.088
128	0.1138	0.0904	0.0855	0.0822	0.0882
256	0.1269	0.0997	0.0908	0.0847	0.0894
512	0.1516	0.1118	0.1013	0.085	0.0843

The error with batch size 60 and 10 epochs is 0.804, which is better than all of the error rates listed above. However, looking at the table, we can see that as the number of step size and batch size getting smaller and smaller, the training result is generally getting better. Considering the significant difference in terms of training epochs, we can make the assumption that given the same number of epochs, a model with smaller batch size and step size is highly possible to beat the original model. However, as will be discussed below, the cost of having small batch size and step size is the increase in the runtime of training process, which is an important trade off to consider when choosing parameters.
5.





Part 3 System Evaluation

The runtime of four algorithms are listed above:

Methods	Sequential sample	Time elapsed (seconds)
Stochastic Gradient	no	74.1862
Stochastic Gradient	yes	56.8365
minibatch_SGD	no	28.3648
minibatch_SGD	yes	14.5895

Each time period represented in the table above is the average result across 5 total runs of the algorithm. All the parameters of these four methods implemented to test the runtime are based on what provided in Part 1.

Runtime regarding minibatch:

It is quite obvious from the table above that minibatched_SGD is considerably faster than standard Stochastic Gradient Descent in terms of runtime provided that same number of total gradient samples are used during the training process, just like what happened in this experiment. One of the reasons could be that in order to use the same number of total samples, standard gradient descent would need significantly more number of iterations which is related to the batch size of minibatch method, where each iteration requires an update to the weight array(vector) **W**, whose computational cost is related to the size of **W**. Say the batch size is 60, as it was in the experiment, the standard SGD would need to iterate, update weights W, in other words, 60 times the total time minibatched_SGD need to conduct, which is a big difference.

Runtime regarding sequential sampling:

On the other hand, given the same basic model (SGD or minibatched_SGD), methods with sequential sampling are also significantly faster than those without this feature. One of the reasons could be, as stated in the introduction page of this assignment, the cache issue. Suppose our dataset is stored in memory in the order of their index. Since the space of Cache and register is limited compared to datasize, each time we need a certain sample or period of sample, the Cache will fetch a small range of data, usually several KB, and let the register in CPU grab the piece of data it needs from Cache. When the next time register need another piece of data, it will first search in the Cache for its target, if the piece of data happen to be in the Cache, which is called a “hit”, the register can have this piece of data “right away”, on the other hand, though, if the Cache does not contain that piece of data, which is called a “miss”, the Cache would have to grab another range of data from the memory, which takes a LONG time compared to the speed of Cache to register.

Back to the scenario in our experiment, when we request certain sample in one iteration of the algorithm, the Cache will likely to load the sample we need and several of its following samples. If the sample we need after that happens to be the next sample of current sample, which is the case in a sequential sampling process, we can create a “hit” in the Cache, thus fetch the next sample immediately. Whereas if we randomly sample data from the dataset, it quite likely that the Cache would need to load a new range of data every time we request a new sample so as to reach randomness, which means we will “miss” almost every time, and that would be a lot slower compared to the speed of sequential sampling.