1003243509 Agia 1
March 13, 2019

ROB313 Intro to Learning from Data - Assignment 3

Objectives

The objective of this assignment is to acquire first hand experience training various types of models, such as a Linear Model and a Logistic Regression Model, using gradient descent. Multiple forms of gradient descent including full-batch and mini-batch gradient descent will be implemented for training at a range of learning rates. The performance of full-batch and mini-batch will be compared by computing predictions on the test set after a set number of training iterations. Furthermore, the full-batch loss or full-batch log-likelihood will be computed on the training set after every training iteration for each gradient descent method at a range of learning rates. These values will be plotted as to display the decay in prediction error with increasing training iterations.

Code Structure and Strategies

The code structure is designed to be modular and easily understandable. Each question has a corresponding code section in the script and in the main block, which is boldly outlined with comments. To run the code for each question, the only modifications needed to be made to the file is to set the Booleans corresponding to each question to True in the main block. The results of each question are printed neatly on the terminal when run, and relevant plots will be generated and saved in the current directory. The remaining code is structured as follows:

Q1: **linear_regression(**data**)**, **gd(**data, rmse, model**)**, **generate_lossplots(**data, losses, model**)**.

In Question 1, the **linear_regression()** function is used to determine the optimal RMSE loss on the pumadyn32nm dataset. This optimal RMSE loss is passed in **gd()**, which is a function responsible for carry-out GD or SGD (depending on model parameter)s to train a linear model on the same dataset at several learning rates. One-thousand iterations are computed for each learning rate. **gd()** also computes the test RMSE at each training iteration to determine when the model is able to make predictions within 1% of the optimal test RMSE. Additionally, **gd()** will track the loss on the training set at each iteration. The function **generate_lossplots()** will simply plot the computed losses over increasing iterations.

Q2: **log_reg_GD(**data, model**)**, **log_reg_plots(**losses), **utils()**

In Question 2, a logistics regression model is trained using GD or SGD (based on ‘model’ parameter) for a set number of training iterations. The negative log-likelihood is computed on the training set at every iteration. The log-likelihood values are plotted over increasing training iterations with the **log_reg_plots()** function. Finally, there are several utility functions that are used to compute a single inference (sigmoid function) and to compute the log-likelihood.

Commenting and code organization were used to make the code easy to read and to reduce the time spent searching for critical/error prone sections. Utility methods for frequently used functions (e.g. root mean squared error) were written. Effective code-loop structure was implemented to ensure the re-use of common data structures in the inner loops.

Q1 – Training a Linear Model with Gradient Descent and Stochastic Gradient Descent

A linear model was implemented to make predictions on the pumadyn32nm dataset which contains a multi-dimensional feature space. Initially, a standard SVD implementation was made to determine the model's optimal minimizer, $\boldsymbol{\omega_{opt}}$, on the training set. Predictions were then made on the test set, and the test RMSE was computed as: $e_{opt,RMSE} = 0.870653108623$.

The optimal test RMSE, $e_{opt,RMSE}$, is passed into the function responsible for training a new linear model with gradient descent. The $e_{opt,RMSE}$ parameter is simply used to compare and identify at what exact iteration in the GD training process does the model's test predictions come within 1% of $e_{opt,RMSE}$ (i.e. used as a convergence metric).

Gradient descent and stochastic gradient descent were used to train a linear model with the initial weights set to zero at a range learning rates. For each learning method (e.g. Full-batch GD and Mini-batch SGD), the model was trained for 1000 iterations. For each learning rate, several key learning metrics were tracked:

- Full-batch loss at every training iteration
- Training time to either 1000 iterations, or convergence time to within 1% of $e_{opt,RMSE}$
- Test RMSE values

For full-batch and mini-batch GD, the preferred learning rate was identified as the rate that first reached within the 1% of $e_{opt,RMSE}$ before 1000 iterations. If no learning rate met this threshold, the preferred learning rate was chosen as the rate resulting in the lowest test RMSE after 1000 training iterations. The results for GD and SGD are shown in *Table 1* and *Table* 2, respectively.

Table 1 – Results of Training Linear Model with Gradient Descent [1000 Iterations, Full-Batch]

Learning Rate	**Convergence Time [s]**	**Total Iterations to $e_{opt,RMSE}$**
0.0001	9.532856941223145	1000
0.001	4.116743087768555	424
0.01	0.40128517150878906	42
0.1	0.03853487968444824	4
Preferred Rate		**0.1**
Test RMSE		**0.866812797319**

Table 2 – Results of Training Linear Model with Stochastic Gradient Descent [1000 Iterations, Mini-Batch 1]

Learning Rate	**Convergence Time [s]**	**Total Iterations to $e_{opt,RMSE}$**
0.0001	0.05889010429382324	1000
0.001	0.02471923828125	420
0.01	0.058133840560913086	1000
Preferred Rate		**0.001**
Test RMSE		**0.877175378267**

The preferred rate for full-batch GD is $\eta = 0.1$, which converges to within the 1% of $e_{opt,RMSE}$ in only 4 training iterations. For mini-batch SGD, we see that $\eta = 0.001$ yields the best results, converging after only 420 iterations. In both cases, the predictions on the test set yielded test

RMSE's comparable to $e_{opt,RMSE}$ (e.g. lower in the case of full-batch). These results also show that stochastic gradient descent is significantly faster in terms of computation time. It is shown that the time it takes to preform 1000 iterations of mini-batch SGD is almost equivalent to the time needed to run 4 iterations of full-batch GD. *Figure 1* and *Figure 2* illustrate the full-batch loss on the training set plotted against increasing iteration number.

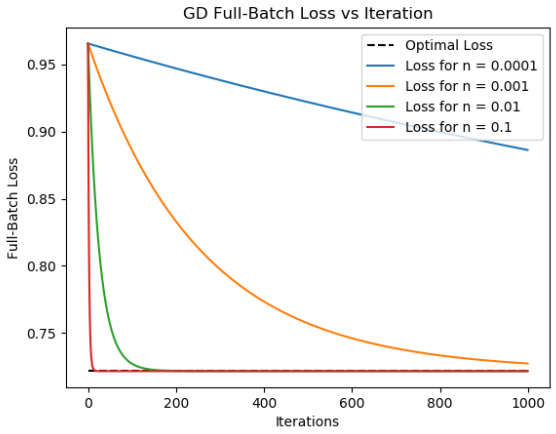

Figure 1 – Full-Batch GD Loss

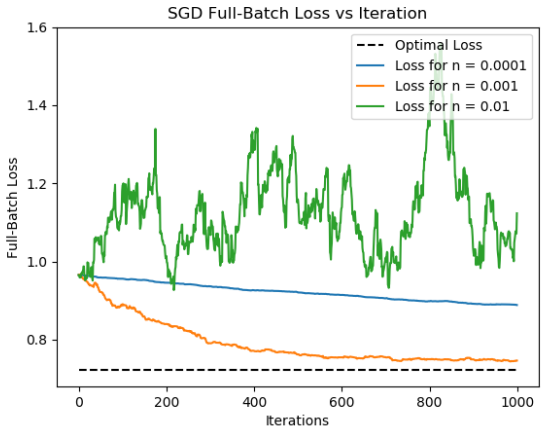

Figure 2 – Mini-Batch SGD Loss

Here we can see the convergence trends of each technique. In both plots, the full-batch loss tends towards to exact optimum value of $L_{opt} = 0.72161047$**,** which is represented by the black dashed line. In *Figure 1*, an optimal loss of $L_{GD} = L_{opt}$ occurred after 1000 iterations with learning rate $\eta = 0.1$ (red line). In *Figure 2*, an optimal loss of $L_{SGD} = 0.73591041$ occurred with a learning rate of $\eta = 0.001$ (orange line). As expected, these results are in agreement with the preferred learning rates specified in *Tables 1-2*.

Overall, we see that full-batch gradient descent training is able to perform well with larger learning rates as the is gradient computed and averaged using all data points. In the case of mini-batch stochastic gradient descent, it is noticeable that model tends to become unstable when training with larger weights. This may be a result of the randomly selecting a datapoint to compute the gradient with, in which case an outlier datapoint may skew the gradient and lead to undesired training. For this reason, learning rates above 0.01 were not considered for SGD for this specific dataset and batch size.

Q2 – Training a Logistics Regression Model with Full-batch and Mini-batch Gradient Descent

As in Question 1, a logistics regression model using the Bernoulli likelihood for binary classification was trained using full-batch gradient descent and mini-batch stochastic gradient descent. However, instead of attempting to minimize the loss on the training set, training this model required maximizing the log-likelihood derived from the Bernoulli formula. Inferences of the trained model were made using a sigmoid function, which provided a conditional probability of a datapoint being labeled as class 1.

For both full-batch and mini-batch gradient descent, the full-batch negative log-likelihood was computed over the training set at each training iteration for 5000 total iterations at a range of learning rates. These values are displayed below in *Figure 3* and *Figure 4*.

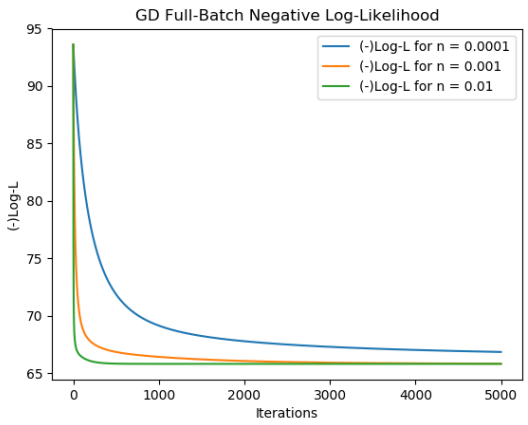

Figure 3 – Full-Batch GD (-) Log-Likelihood

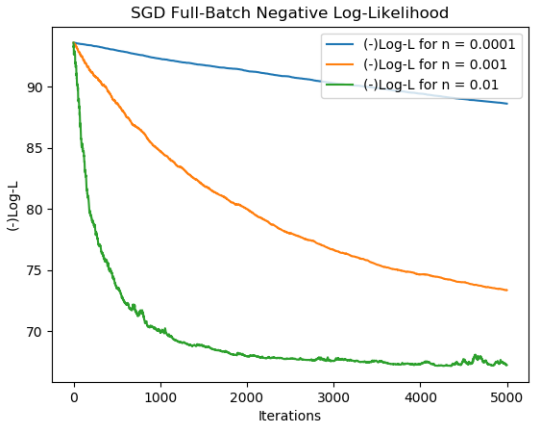

Figure 4 – Mini-Batch SGD Log-Likelihood

After 5000 iterations of training, both models were used to make predictions on the test set. The test accuracy ratio and the test log-likelihood were computed and are provided in *Table 3*.

Table 3 – Test Accuracy Ratio and Test Log-Likelihood for GD and SGD

Training Method	**Test Accuracy Ratio**	**(-) Test Log-Likelihood**
Full-Batch GD	0.733333333333	7.11428961
Mini-Batch (1) SGD	0.733333333333	9.86146089

The computation time was once again tracked to compare the efficiency of both training methods. As expected, the mini-batch SGD took on average 0.068s to complete the 5000 training iterations, while the GD took 3.59s on average. Ultimately, the predictions of both models yielded similar accuracy. This suggests that the accuracy-time trade-off for using full-batch GD may not be worth it in this particular case.

Q 2.1) Why is the value of the log-likelihood when your model predicts class 1 with 100% certainty, but the correct label is 0?

The second term in the log-likelihood expression would encounter a $\log\big(1-\sigma(z)\big)=\log(0)$ which is undefined. As we know, the logarithmic function tends to negative infinity as $x \to 0^{+}$. Intuitively, the model will only predict $\sigma(z)=1=\frac{1}{1+e^{-z}}$ if $z \to infinity$. However, z is simply a sum of products of our features and weights. Because we know our feature values are finite, z only tends to infinity if our weights tend to infinity, which means we are overfitting our model. This can be solved by introducing a regularization parameter to our log-likelihood function to ensure the weights of our model do not grow out of proportion. Overall, this is not reasonable behaviour. It implies that our model is overfitting to make incorrect classifications with high degrees of certainty, which is both unreasonable and undesireable.

Q 2.2) Why might the test log-likelihood be a preferable performance metric?

The test accuracy ratio simply computes the number of correct classifications of the model predictions. However, since our model uses a sigmoid function to make classifications on a range of [0, 1], we may want to leverage exactly where on this range did an inference for a specific data point fall. Test log-likelihood allows us to leverage the conditional probability output from our sigmoid function by taking into account “how correct” or “how incorrect” a prediction is. This makes test log-likelihood a more precise indicator when measuring the accuracy of our model.