Giovanni Piva n.: 2078341
Roberto Vicentini n.: 2055532

HOMEWORK

Optimization for Data Science

1. Introduction

In this homework, we tackle the problem of minimizing a loss function in a multi-class logistic regression problem. We have developed and implemented two optimization methods: Gradient Descent and Block Coordinate Gradient Descent with Gauss-Southwell rule. To evaluate the performance of these methods, we applied them to two different datasets: a synthetic dataset and the well-known MNIST dataset. The effectiveness of the models was assessed by comparing the final results in terms of accuracy, loss values, and computational time.

2. Problem definition

The following is the loss function we want to minimize:

$$\min \sum_{i=1}^{m} \left(-x_{bi}^{\top} a_i + \log \left(\sum_{c=1}^{k} e^{x_c^{\top} a_i} \right) \right)$$

In order to solve the problem we will define the likelihood for a single training example:

$$P(b_i | a_i, X) = \frac{\exp(x_{b_i}^{\top} a_i)}{\sum_{c=1}^{k} \exp(x_c^{\top} a_i)}$$

It uses the softmax function to convert the model's raw logits, calculated as dot products between class-specific weight *vectors* $\boldsymbol{x_c}$ and *feature vector* $\boldsymbol{a_i}$, into actual probabilities ensuring they sum up to one across all *k classes*. This probabilistic output allows for interpreting the highest logit score as the class prediction for *instance* $\boldsymbol{i}$.

$$\frac{\partial H(X)}{\partial x_{jc}} = -\sum_{i=1}^{m} a_{ij} \left[I(y_i = c) - \frac{\exp(x_c^{\top} a_i)}{\sum_{k=1}^{K} \exp(x_k^{\top} a_i)} \right]$$

This formula calculates the partial derivative of the loss function for our multi-class logistic regression. It adjusts parameters by comparing actual class labels to predicted probabilities, facilitating model optimization through Gradient Descent.

3. Jupyter Notebook Implementation

We are now diving into the actual implementation of our solution, we chose to use Jupyter Notebook.

3.1 Data Creation

To begin, we generated a 1000x1000 matrix **A** from a standard normal distribution (mean = 0, standard deviation = 1). Following that, a matrix **X** of dimensions d x k (1000 x 50) was created, also sampled from a normal distribution. Subsequently, another matrix **E** was generated from the same normal distribution. Using these matrices, the label matrix **B** was formed through the following logic:

$$\text{argmax}(A \times X + E, \text{axis} = 1)$$

At this point we have our synthetic samples labeled with a class $\boldsymbol{b_i} = 0, 1, \ldots, k$.

3.2 Functions creations

We now proceed with the creation of the functions we will use during Gradient Descent. We needed to develop the following functions:

- **lossFunction(**A, X, B**):** this function takes in input A, X and the labels matrix B but with the one hot encoded version of the label in order to represent the indicator function: $I(y_i = c)$

- **mySoftmax(**X, A**):** computes the softmax of the dot product of A and X. It first calculates the dot product, applies the exponential function to each element, and then normalizes these values so that each column sums to 1, effectively turning them into a probability distribution.

- **gradient(**samples, labels_onehot, weights**):** calculates the gradient of the loss function used in the training model. It takes three parameters: samples, labels_onehot, and weights. The function first computes mySofmax(X, A) then it calculates the gradient of the loss function by taking the dot product of the transposed samples and the difference between the one-hot encoded labels and the softmax probabilities. This result represents the direction and magnitude to adjust the weights to minimize the error in predictions.

- **accuracy(**y_true, X, A**):** calculates the model's accuracy by comparing predicted labels from the softmax outputs of weighted input data (X and A) against true labels (y_true). It identifies correct predictions, totals them, and divides by the number of samples to determine the accuracy. This measure is for evaluating and comparing model performance on both training and testing datasets.

3.3 Models

3.3.1 Gradient Descent

Our Gradient Descent implementation followed this pipeline:

1. **Parameter initialization:** the gradient descent began with setting the total iterations to a fixed number (we tried with different numbers and stayed with the one with best result overall) and a convergence threshold (EPSILON) of 0.01 to deal with minimal improvement. The process was initiated with a copy of the initial model parameters, and both the initial loss and accuracy were computed and recorded to establish baseline performance metrics.
2. **Gradient Computation:** At each iteration, the gradient of the loss function with respect to the model parameters is calculated.
3. **Parameter Update:** The parameters were updated by moving in the negative direction of the gradient, scaled by a learning rate, that is the following update rule:

$$x_{t+1} = x_t - \alpha \cdot \nabla H(x_t)$$

   In this case we used a learning rate α = 0.0001, since that resulted in one of the best values.
4. **Monitoring:** Loss and accuracy were recalculated post-update to monitor the model's performance continuously.

3.3.2 Gradient Descent Block Coordinate with GS rule

We now present the major differences between GD and GDBS with GS rule implementations:

1. **Block Selection (Gauss-Southwell Rule):**

   In order to implement the GS rule, during our Gradient Descent we hate to decide which coordinate we need to block. We tried both blocking columns and rows, but we had better results when blocking columns. The following is the formula that represents how we pick the blocks. We can see that it picks the block index i_k that has the maximum gradient norm.

$$i_k = \text{Argmax}_{j \in \{1,\ldots,b\}} \|\nabla_j f(x_k)\|$$

2. **Block Coordinate Update:**

   After selecting the block i_k with the largest gradient norm, this time we will use a different update rule:

$$x_{k+1} = x_k - \frac{1}{L}\nabla_{i_k} f(x_k)$$

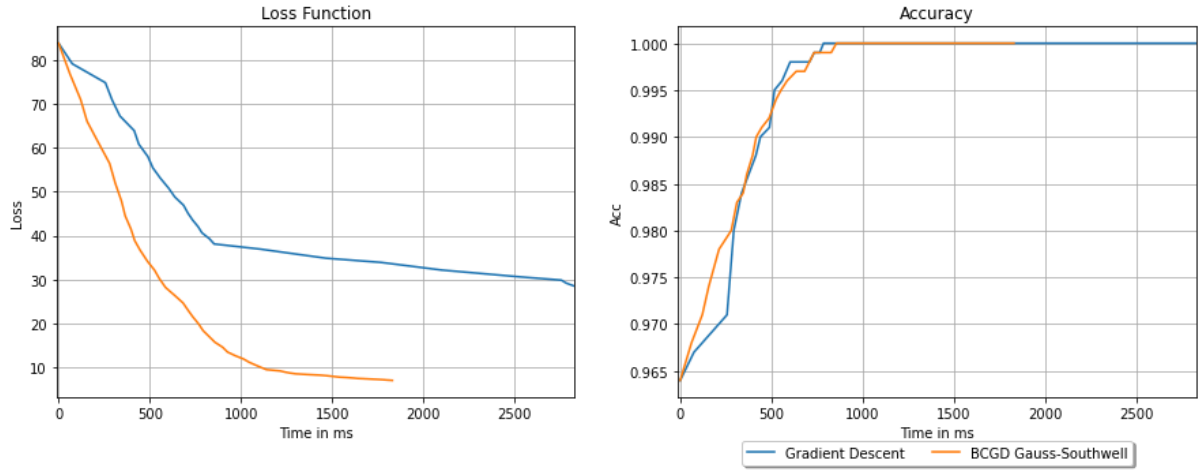
Here we encountered a challenge, since in order to obtain the correct value for L, we previously needed to compute the Hessian Matrix.
Calculating the full Hessian matrix for our optimization problem would have been too slow and computationally heavy. Instead, we used $\mathbf{L_{max}}$, the highest eigenvalue of the Hessian matrices for each class. In addition to that, only when we were dealing with the MNIST dataset, we also applied a factor of **2.70** to L, which gave us the best results. This method makes the gradient descent updates more stable and faster, improving convergence without losing accuracy.

4. Results

In this section, we present the results of applying Gradient Descent (GD) and Block Coordinate Gradient Descent (BCGD) with the Gauss-Southwell (GS) rule. We will first discuss the results from the synthetic dataset, followed by the results from the MNIST dataset.

4.1 Synthetic Dataset

For the Synthetic dataset, we had good and fast result with both GD and GDBC, the time we see in the chart is relative to **50 iterations** and as we can see after only few iteration we have good results both in terms of loss and accuracy:

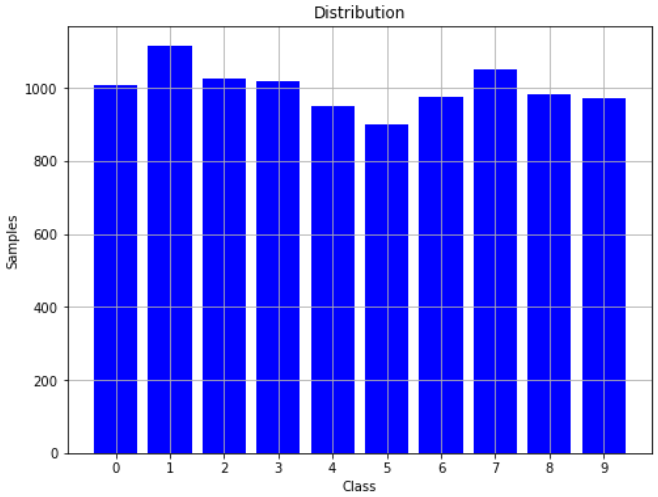


Test accuracy GD: 96.4%
Test accuracy GDBC: 96.2%

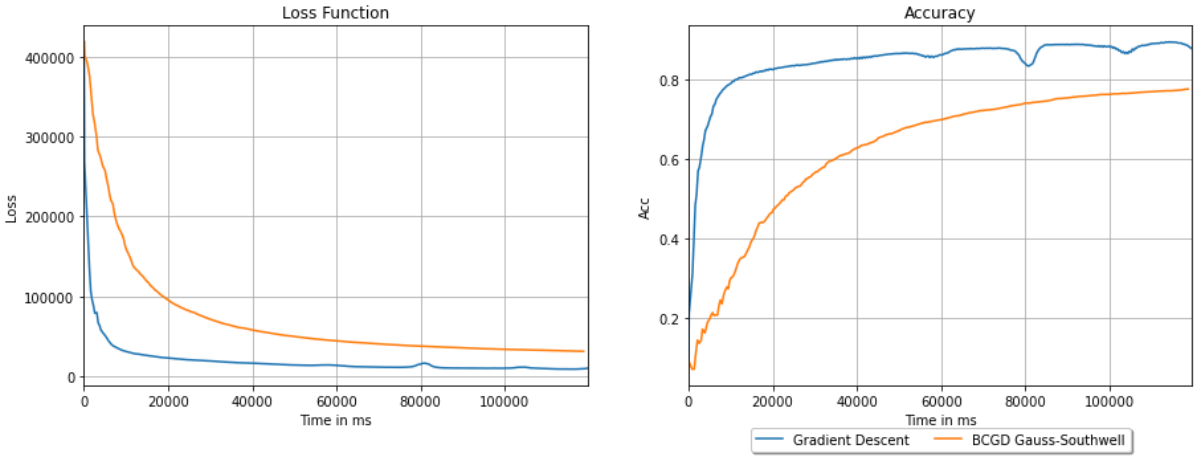
4.2 MNIST

First of all, in order to use the MNIST dataset, we had to previously transform the original dataset. We firstly took **10.000 samples** from the MNIST dataset. In this dataset every sample is a **28x28** matrix, and it represents the number associated with its label. In order to correctly process the data we need to transform every 28x28 matrix into a vector with **784** values where, every 28 values, a row from the previous matrix is represented.

We ended up with 10.000 samples with the following label distribution:



We now present the results we obtained with **300 iterations**,**:**



Test accuracy GD: 86.09%
Test accuracy GDBC: 77.85%

5. Conclusion

As observed with the synthetic dataset, we achieved excellent results immediately, with both GD (Gradient Descent) and GDBC (Gradient Descent with Block Coordinate) reaching 100% accuracy in training almost instantly (and around 96% in test). This probably leads to how simple the synthetic dataset was. One last thing we want to underline is that for this dataset, the GDBC performed better when looking at the loss function since it converges faster than the standard GD.

For the MNIST dataset, we also achieved good, even excellent results with Gradient Descent, reaching around 90% training accuracy, with the loss decreasing as expected. However, GDBC performed worse in terms of accuracy, achieving slightly below 80% accuracy in train and around 78% in test. For the loss function, this time GD was the best as we can see from the plot, it converges faster and better.