Gradient Methods for Multi-class Logistic Regression

Alessandro Pala - 2107800

Tanner Aaron Graves - 2073559

Alisa Snezskaia - 2107497

Anna Glado -

May 2024

1 Introduction

In this report, we analyze the relative performance of various gradient methods at optimizing a multi-class logistic regression model. Specifically, we are interested in comparing full gradient descent with block coordinate-based methods (BCGD) which optimize one coordinate, or column in parameter matrix X, at a time.

Multi-class logistic classification is an extension of traditional logistic regression. The task goes from predicting a Bernoulli random variable to predicting one label to classify each example from a set of k mutually exclusive labels. We encode the target $b_i \in \mathbb{R}^k$ where k is the number of output classes to be a one-hot array, corresponding to the correct classification of example $a_i \in \mathbb{R}^d$. It assigns these labels by learning the conditional probability distribution that the correct label is b_i given example a_i and learned parameter matrix $X \in \mathbb{R}^{d \times k}$ where d is the number of features in each example. The output distribution is computed with the softmax function in this context:

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

Multi-class logistic regression is a simple yet effective method with ubiquitous use. Its training does, however, require learning the parameter matrix X, which can be done using various methods like maximum likelihood or gradient descent-based methods. This corresponds to solving the following optimization problem:

$$\min_{x \in \mathbb{R}^{d \times k}} \sum_{i=1}^{m} [-x_{b_i}^T a_i + \log(\sum_{c=1}^{k} \exp(x_c^T a_i))]$$

which is the cross-entropy of the predicted class distribution and the target label b_i. Of importance to the application of gradient methods is this problem's convexity. This is nice for many reasons, but most importantly we can guarantee that a local minimum found by GD methods will be the global minimum.

2 Algorithms

Here we provide an overview of the methods studied. We implemented the various methods in Python using either PyTorch or NumPy. The three implementations are a full Gradient Descent and two Block-Coordinate Gradient Descent methods, one with the Gauss-Southwell rule and one with the randomized rule.

2.1 Gradient Descent

Algorithm 1 Full Gradient Descent

Initialize $X_1 \in \mathbb{R}^{d\times k}$
for $k = 1, \ldots$ **do**
  if $X \in X^*$ **then STOP** (Optimality conditions)
    Set $X_{k+1} = x_k - \alpha_k \nabla$
  end if
end for

2.2 Block Coordinate Gradient Descent

BCGD methods are specialized in optimizing high-dimensional problems where calculation of the gradient can be split up into blocks which are easier to compute. These methods exploit the convex nature of the problem to optimize one coordinate at a time and approach the global optimum.

Algorithm 2 BCGD

Initialize $X_1 \in \mathbb{R}^{d\times k}$
for $k = 1, \ldots$ **do**
  if $X \in X^*$ **then STOP** (Optimality conditions)
    Select $i_k \in$ block according to some **RULE**
    Set $x_{k+1} = x_k - \alpha_k [\nabla_x f(x_k)]_{i_k}$
  end if
end for

Above $\alpha_k \geq 0$ is a step size for the kth iteration. We later analyze the effect of different methods for calculating this. The choice of **RULE** will distinguish the Gauss-Southwell and Randomized variants of BCGD. The Gauss-Southwell (GS) rule is a greedy selection method that focuses on the block with the largest gradient component at each iteration. This method involves computing the

gradient of the objective function for all blocks and selecting the one with the maximum gradient norm. By updating the block that is expected to provide the greatest improvement in the objective function, the GS rule can lead to faster convergence, especially in problems where gradients vary significantly across blocks. However, the computational cost is higher due to the need to evaluate the full gradient at each step, making this approach more complex and potentially less efficient for high-dimensional problems.

For the GS rule we compute

$$i_k = \arg\max_j ||\nabla_j f(x_k)||$$

Taking i_k uniform random samples to be the block index gives the randomized rule. This method simplifies the implementation and reduces computational cost, as it only requires computing the gradient of the randomly selected block rather than the full gradient.

3 Step Sizes

3.1 Fixed Step Size and the Lipschitz Constant

3.2 Block Step Size

3.3 Exact Step Size

4 Synthetic Dataset

To initially validate the convergence of our algorithms we generate a high-dimensional dataset for the multi-class classification problem. This consists of our data matrix $A \in \mathbb{R}^{1000\times 1000}$ and target classes b. The rows of A correspond to individual examples in the dataset and the columns their numeric features. Entries for A were drawn i.i.d. from a $N(0,1)$ distribution. Here $b \in \mathbb{Z}_{50}$ represents the vector of target classes corresponding to their respective rows in A. We define our starting parameter matrix $X \in \mathbb{R}^{1000\times 50}$. We can then define an error matrix with standard normal entries to inject noise into our calculation of the target classes

$$b = AX + E$$

and further process b to be the argmax of each row. This dataset has the feature of having an expensive gradient calculation due to the high number of features and output classes. This creates the potential for BCGD methods to be advantaged as ones like the randomized rule may compute a smaller block of the gradient, though this also depends on the generation seed for the distribution.

4.1 Base Performance

All the presented results were run at a 6×10^{-6} learning rate.[1] As for the synthetic dataset, we find that at fixed step size the full gradient outperforms both block-coordinate methods, with the Randomized BCGD performing the worst. In absolute terms, however, all three methods converge to a high accuracy with an efficient CPU time.

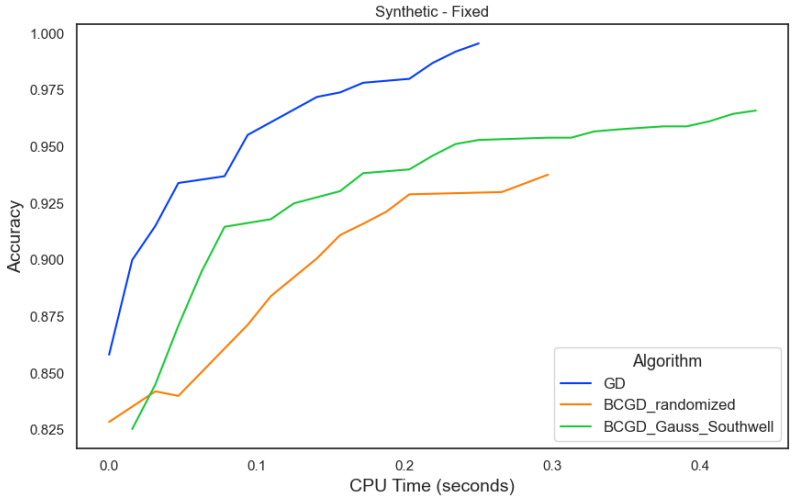

Figure 1: Synthetic Dataset - Fixed Step Size

4.2 Step Size Analysis

We also find that the performances of the algorithms remain the same for the other implemented step sizes, with minimal differences solely related to the seed of the distribution and the learning rate. – Why? –

[1] Testing was done with many learning rate values with all trends remaining almost identical; the chosen value is simply the one that produces the plots that best represent them.

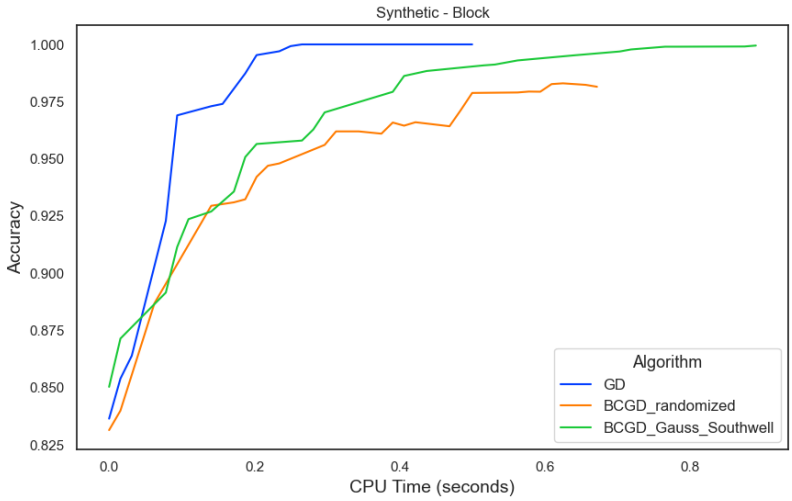

Figure 2: Synthetic Dataset - Block Step Size

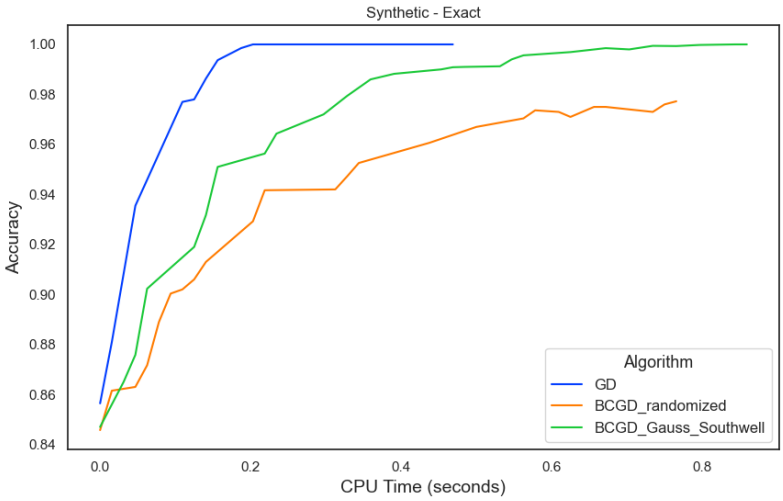

Figure 3: Synthetic Dataset - Exact Step Size

All step sizes increase indefinitely during descent at roughly the same rate, and all resemble step functions. – Is this expected? What does it mean? –

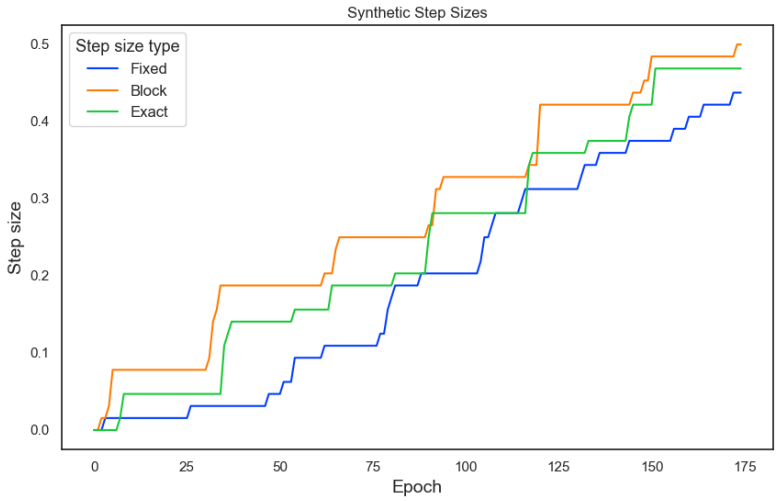

Figure 4: Synthetic Dataset - Various Step Sizes

5 Real-world Non-Sparse Dataset

5.1 Base Performance

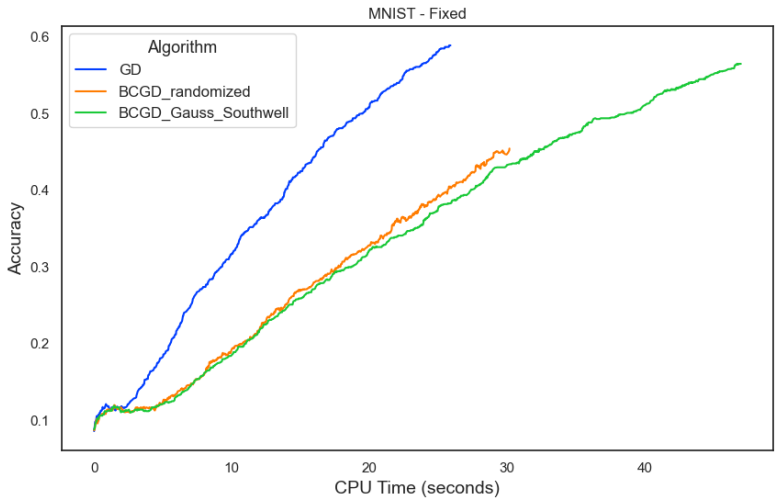

Figure 5: MNIST Dataset - Fixed Step Size

5.2 Step Size Analysis

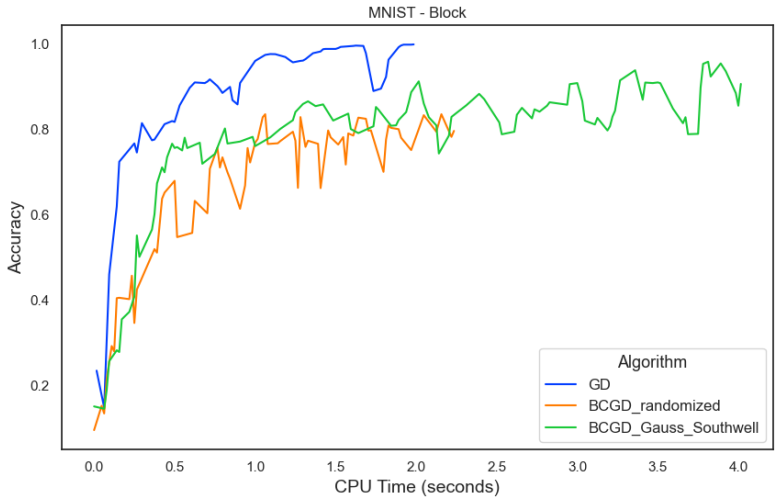

Figure 6: MNIST Dataset - Block Step Size

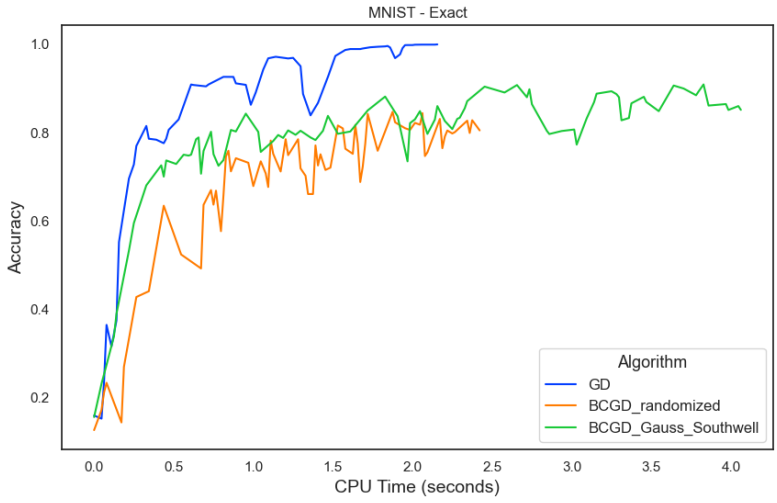

Figure 7: MNIST Dataset - Exact Step Size

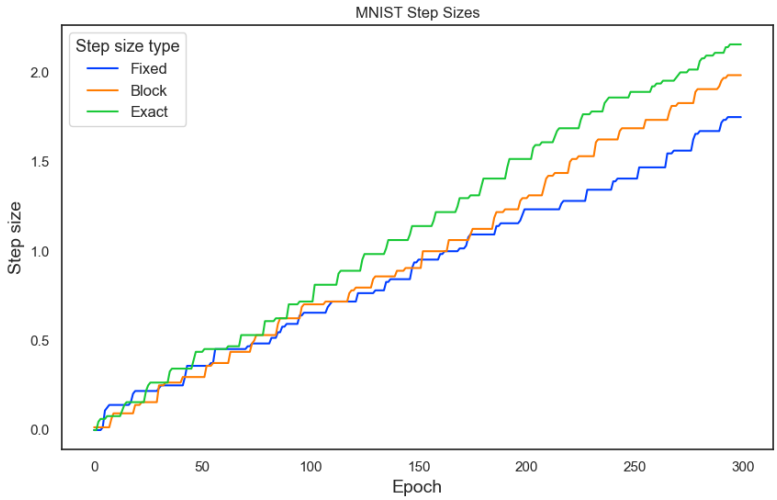

Figure 8: MNIST Dataset - Various Step Sizes

6 Real-world Sparse Dataset

We selected the MNIST dataset for our first real-world, publicly available dataset.

6.1 Base Performance

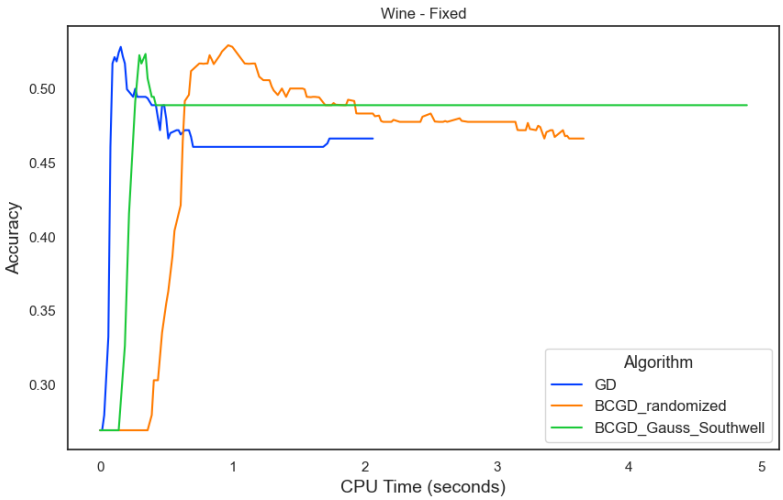

Figure 9: Real-world Sparse Dataset - Fixed Step Size

6.2 Step Size Analysis

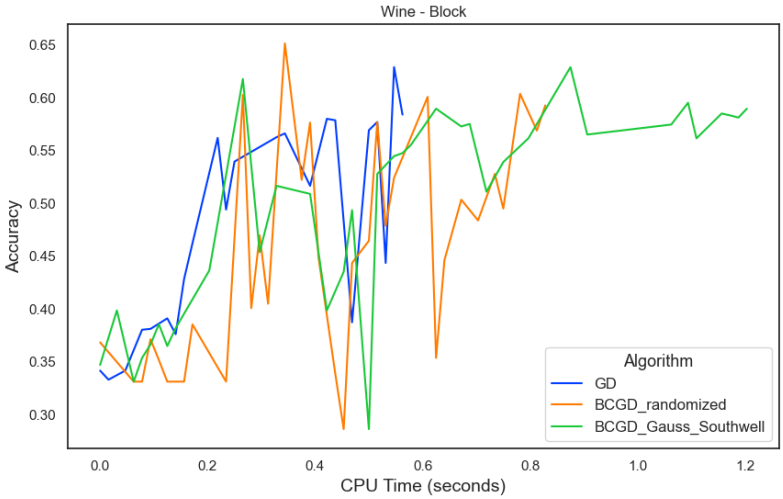

Figure 10: Real-world Sparse Dataset - Block Step Size

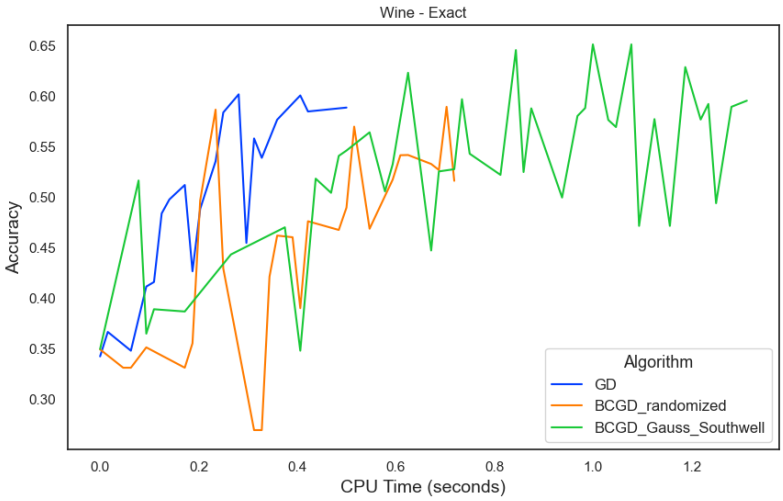

Figure 11: Real-world Sparse Dataset - Exact Step Size

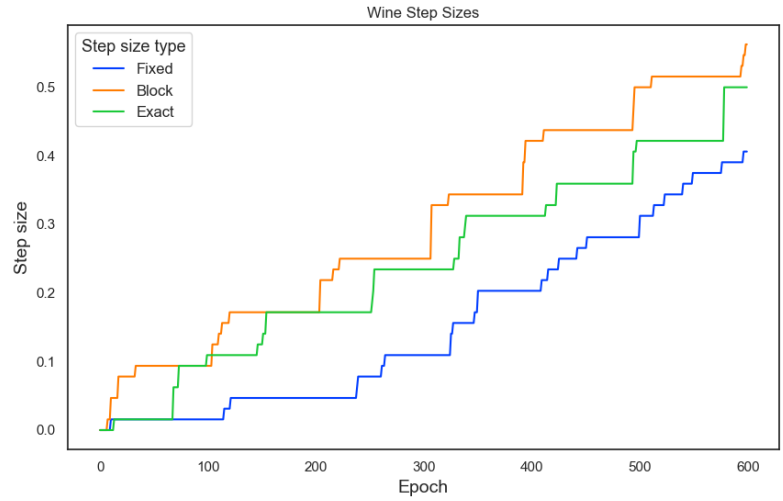

Figure 12: Real-world Sparse Dataset - Various Step Sizes