Optimization For Data Science

Exploring Gradient Descent and BCGD Approaches for Multi-Class Logistic Regression

Department of Mathematics

Università degli Studi di Padova

Fairouz Baz Radwan, ID:2071913
Sofia Pope Trogu, ID:2072117

Supervisor: Prof. Francesco Rinaldi

May 21st, 2024

Table of Contents

1 Introduction

Amidst the abundance of available data, parameter tuning, feature selection, and classification pose notable challenges for modern algorithms. This report shifts the focus from the classic binary classification problem to the domain of multi-class logistic regression. Our objective is to address a multi-class logistic regression problem utilizing a negative log-likelihood loss function, applied specifically to a synthetic dataset randomly generated in matrix form based on a normal distribution.

Moving away from binary classification, we tackle the complexities of multi-class logistic regression. In multi-class logistic regression, the algorithm models the probability that each instance belongs to each class using the logistic function (also known as the sigmoid function). The probabilities for each class are then normalized to ensure they sum up to 1. The class with the highest probability is then predicted as the output class. The objective of multi-class logistic regression is to find the best set of parameters (coefficients) that maximizes the likelihood of the observed data given the model. This is typically done by minimizing a loss function, such as the cross-entropy loss, which measures the difference between the predicted probabilities and the actual class labels. The parameters are adjusted iteratively using optimization techniques until convergence is reached, optimizing the model's ability to accurately classify instances into their respective classes.

We employ a series of unconstrained analytical algorithms tailored for multi-class logistic regression. These algorithms include gradient descent and two variants of block coordinate gradient descent (BCGD). The aim is to optimize the minimization of a defined negative log-likelihood loss function across multiple classes.

Our analysis progresses through experimentation with both synthetic and real datasets. We begin by testing our methodologies on synthetic data to refine algorithms in a controlled environment. Subsequently, we transition to real datasets to assess the practical applicability and robustness of our techniques in real multi-class settings.

In summary, our investigation revolves exclusively around the challenges and solutions associated with multi-class logistic regression. Through the application of semi-supervised learning and experimentation with synthetic and real datasets, we aim to develop effective methodologies for addressing multi-class classification problems.

2 Definitions and Notations

2.1 Loss Function

In our multiclass logistic regression problem, we aim to predict the probabilities of different classes for a given input. To achieve this, we use the softmax function, which converts the raw output scores of the model into probabilities that sum to one. The softmax function for class b_i for training sample i given an input a_i is defined as:

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

where:

- x_{b_i} is the weight vector for class b_i (column vector in $\mathbf{X} \in \mathbb{R}^{d \times k}$)
- a_i is the input feature vector for sample i (the transpose of i_{th} row in matrix $\mathbf{A} \in \mathbb{R}^{m \times d}$)
- k is the number of classes.

The likelihood of the model parameters X (the set of all weight vectors) given the data is the product of the probabilities assigned to the true classes of all training examples. For a dataset with m examples, the likelihood $L(X)$ is:

$$L(X) = \prod_{i=1}^{m} P(b_i \mid a_i, X)$$

To simplify the optimization process, we work with the log-likelihood:

$$\ell(X) = \sum_{i=1}^{m} \log P(b_i \mid a_i, X)$$

In order to optimize our predictions, we minimize the negative log likelihood (NLL) as our loss function. By doing that, we aim to maximize the likelihood of our model generating the correct labels for the given input data. The negative log-likelihood $-\ell(X)$ for our multiclass logistic regression model is:

$$-\ell(X) = -\sum_{i=1}^{N} \log \left(\frac{e^{x_{b_i}^T a_i}}{\sum_{c=1}^{k} e^{x_c^T a_i}} \right) = \sum_{i=1}^{m} \left[-x_{b_i}^T a_i + \log \left(\sum_{c=1}^{k} \exp\left(x_c^T a_i\right) \right) \right] \tag{1}$$

Therefore, the optimization problem can be expressed as:

$$\min_{X} -\ell(X) \tag{2}$$

2.2 Gradient

To implement the gradient descent scheme for minimizing the negative log likelihood, we first need to compute the partial derivative with respect to each parameter:

$$\begin{aligned}\frac{\partial}{\partial x_{js}} \sum_{i=1}^{m} \left[-x_{b_i}^T a_i + \log \left(\sum_{c=1}^{k} \exp\left(x_c^T a_i\right) \right) \right] \\ = \sum_{i=1}^{m} \left(-a_{ij}\mathbb{I}(b_i = s) + \frac{a_{ij}\exp(x_s^T a_i)}{\sum_{c=1}^{k} \exp(x_c^T a_i)} \right) \\ = \sum_{i=1}^{m} -a_{ij} \left(\mathbb{I}(b_i = s) + \frac{\exp(x_s^T a_i)}{\sum_{c=1}^{k} \exp(x_c^T a_i)} \right)\end{aligned}$$

where $\mathbb{I}(b_i = s)$ is an indicator function that is 1 if the true class b_i of example i is s, and 0 otherwise.

In order to be computationally efficient, we computed the loss and gradient using matrix form. Let $\mathbf{Y}_{\text{oh}}$ be the one-hot encoded matrix of dimensions $m \times k$, where m is the number of samples and k is the number of classes. Each row of $\mathbf{Y}_{\text{oh}}$ represents the one-hot encoded vector for a sample, indicating the true class.

We define the matrix multiplication $\mathbf{AX}$ as follows:

$$\mathbf{AX} = \begin{bmatrix} \sum_{j=1}^{d} a_{1j}x_{j1} & \sum_{j=1}^{d} a_{1j}x_{j2} & \cdots & \sum_{j=1}^{d} a_{1j}x_{jk} \\ \sum_{j=1}^{d} a_{2j}x_{j1} & \sum_{j=1}^{d} a_{2j}x_{j2} & \cdots & \sum_{j=1}^{d} a_{2j}x_{jk} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{j=1}^{d} a_{mj}x_{j1} & \sum_{j=1}^{d} a_{mj}x_{j2} & \cdots & \sum_{j=1}^{d} a_{mj}x_{jk} \end{bmatrix}$$

where $\mathbf{A}$ is the matrix of features (of dimensions $m \times d$), and $\mathbf{X}$ is the matrix of weights (of dimensions $d \times k$).
The softmax function is then applied to the result of this matrix multiplication to obtain the predicted probabilities for each class:

$$\mathbf{P} = \text{softmax}(\mathbf{AX})$$

where the softmax function is applied row-wise to the matrix $\mathbf{AX}$.
Using the fact that $x_{bi} = \mathbf{X}Y_{\text{oh,i}}^T$, the negative log-likelihood (or cross-entropy loss) can be expressed in matrix form as:

$$-\ell(\mathbf{X}) = -tr(\mathbf{AXY}_{\text{oh}}^T) + \sum_{i=1}^{m} \log \sum_{c=1}^{k} \exp((\mathbf{AX})_{ic}) \tag{3}$$

where tr$(\cdot)$ denotes the trace of a matrix.
The gradient of the negative log-likelihood with respect to the weight matrix $\mathbf{X}$ is given by:

$$\frac{\partial}{\partial \mathbf{X}} \left(-\ell(\mathbf{X})\right) = \mathbf{A}^T(\mathbf{P} - \mathbf{Y}_{\text{oh}}) \tag{4}$$

2.3 Performance Metrics

We assess and compare the performance of the algorithms across the datasets by generating plots of their loss values, considering both the number of iterations and the computational time elapsed. While we provide accuracy plots to measure the performance of these algorithms, it's important to note that we haven't partitioned the dataset into typical train/validation/test splits, as is common in traditional machine learning scenarios. Our primary focus here is not to evaluate the algorithms' ability to generalize on unseen data, instead our main interest lies in comparing these algorithms in terms of their convergence behavior. We aim to analyze and compare how quickly and efficiently each algorithm converges to the solution, rather than their predictive performance on new data, and as such the overfitting scenario is not much of an issue in our case

3 Datasets

3.1 Pre-processing

In real-world problems, data is abundant and varied across multiple dimensions. However, visualizing labeled data poses a challenge. To address this, we aim to adapt algorithms to synthetic data. Raw data can be messy and have a wide range, so we preprocess it by scaling using the min-max scaling:

$$z = \frac{x - \min(x)}{\max(x) - \min(x)}$$

This enhances convergence speed and reduces errors.

3.2 Synthetic Dataset

In order to develop the gradient algorithms for our unconstrained problem, we tested them on a synthetic dataset. We first generated our feature matrix, A by randomly generating a 1000x1000 matrix with entries drawing from a normal distribution with mean zero and standard deviation one. Next, we needed to create our labels for each sample or row from A in a matrix, B. To arrive at the final label for each sample, we performed the following steps: generate a weight matrix, X with dimensions features (d=1000) x classes (k=50) sampled from normal distribution, N(0,1), generate a noise matrix, E with dimensions samples (m=1000) x classes (k=50) also sampled from the same normal distribution, perform matrix multiplication of the previously generated A and X matrices plus the E matrix (AX + E), and finally, assign the final label to the index with the maximum value for each sample and save it in vector Y.

3.3 Real Dataset

For a real dataset, we used the wine quality dataset from UCI, a popular benchmark in the realm of machine learning and classification tasks. This dataset comprises various chemical properties associated with wine samples, offering insights into their composition and quality. The total number of samples is 4,898. The 11 features encompass a range of attributes including fixed acidity, volatile acidity, citric acid content, residual sugar levels, chlorides, free sulfur dioxide, total sulfur dioxide, density, pH values, sulphates, and alcohol content.

Each wine sample in the dataset is assigned a quality rating, represented as an integer on a scale from 0 to 10, where 0 indicates the lowest quality and 10 denotes the highest. The dataset is substantial, often comprising thousands of samples, each characterized by its unique combination of chemical features and corresponding quality rating. Moreover, we note that in the final dataset, we only observe 7 out of these 11 classes (i.e, with non-zero frequency)

Derived from either real-world observations or simulated data based on expert evaluations and chemical knowledge, the wine quality dataset serves multifaceted purposes in machine learning research. Primarily, it facilitates the exploration of intricate relationships between diverse chemical attributes and the perceived quality of wines. Moreover, it serves as a testbed for evaluating the efficacy of classification algorithms in predicting wine quality based on multivariate input data.
(source: https://archive.ics.uci.edu/dataset/186/wine+quality)

4 Algorithms

4.1 Gradient Descent

Gradient descent is an optimization algorithm used to minimize the cost function in various machine learning algorithms. It is an iterative method that moves towards the minimum of the cost function by taking steps proportional to the negative of the gradient (or approximate gradient) of the function at the current point. One starts with an initialization of the parameters, assigned initially to small values or zero. In our work, these parameters are stored in the X matrix that assigns weights to each feature and class combination. In each iteration, the algorithm calculates the full gradient of the cost function with respect to each parameter. This gradient is represented as a matrix of partial derivatives of the cost function with respect to the parameters. We express the formulation of this gradient in Section 2.2 of this report. After computing the gradient, the algorithm performs a parameter update in the direction of the negative gradient with a certain step size. This step size is also known as the learning rate, a hyperparameter that determines the size of the steps we take to update the weights. The previous steps are repeated until convergence is reached or other stopping conditions are met. The general formula for gradient descent is as follows:

$$x^{(k+1)} = x^k - \alpha_k \nabla f(x^k)$$

In our project, we ran gradient descent for 500 iterations, used a standard learning rate of 0.001, and saved the losses, accuracies and cpu times every ten iterations. This protocol was applied equivalently between the synthetic and real dataset.

4.2 Block Coordinate Gradient Descent

Block coordinate gradient descent (BCGD) is an optimization algorithm that aims to minimize a function by updating subsets of variables (blocks) in each iteration.

4.2.1 BCGD Randomized

Randomized BCGD is a variant of BCGD where the order of updating the blocks is random. In our study, we consider a uniform distribution to randomly generate our blocks with a fixed probability 1/u. We defined a block as a *column* of the X matrix, which corresponds to a unique class across all features. Therefore, we perform a "full-update" in the BCGD method every number of class updates. If we had considered the *row* as a block, this "full-update" would happen every number of feature updates. In most cases, the features vastly outnumber the number of classes, so this is a significant computational consideration.

The general update equation for the i_k-th block coordinate can be written as:

$$x_{i_k}^{(k+1)} = x_{i_k}^{(k)} - \alpha_k \nabla_{i_k} f(\mathbf{x}^{(k)})$$

In our project code, we set the same parameters as gradient descent algorithm (max iterations = 500 and learning rate = 0.001). For each iteration, we iterate through all the the blocks or classes to align with a full gradient descent algorithm. In the nested loop, we randomly select a block to update from index 0 to the number of classes. To calculate the gradient with respect to that block, we use a modified soft-max determination where the numerator only considers the j-th column of the X matrix. Similarly, we only consider the j-th column of the one-hot encoded Y matrix for each sample in the full gradient calculation. The previously calculated gradient is then used to update the weight parameters of just the j-th block in the X matrix. As in the gradient descent set-up, we measured the loss, accuracies, and cpu-times every ten iterations.

4.2.2 BCGD Gauss-Southwell

Unlike the previous algorithm, which randomly selects the blocks at each iteration, the BCGD Gauss-Southwell algorithm chooses the indices based on a deterministic strategy known as the Gauss-Southwell rule. In our code, we rely on choosing the maximum absolute value of the norm of the gradient column array as our index for a block in each iteration. And so, it is important to us to compute the updated full gradient array at each of the iterations in a closed-form way instead of running the full gradient. To do that, we use our loss function and the gradient derived with respect to each block to compute the update of the gradient descent.

In opting to define the column as a block instead of the row , we simplify the closed-form update of the gradient necessitated by the GS update rule. Considering a feature row as a block implies that we update a feature for each class (every BCGD block update). However, this leads to the fact that in the gradient soft-max term, specifically in the denominator, every term in the summation is updated by a different value (which corresponds to dividing each term in the summation by a different constant). This makes the update of the gradient very complicated as finding a closed-form solution for the residual that we have to add to the old gradient requires handling these different denominators.

For the implementation of this algorithm, we used our standard loop parameters (max iterations = 500 and learning rate = 0.001). Prior to entering the loop, we cached the soft max terms of the full initial gradient. This allows us to perform our closed-form updates of the gradient within the loop. For each class update, we calculate the gradient using our cached variables, use an argmax to determine the block index with the largest gradient, use the block's gradients values to update the j-th column of the X matrix, re-calculate the j-th column of the numerator of the softmax using the previously updated X matrix, and finally update the denominator of the softmax by adding and subtracting the new numerator and old numerator, respectively. As in the two previous algorithms, we generate and monitor the losses, cpu-times, and accuracies every ten iterations.

5 Results

5.1 Results: Synthetic Dataset

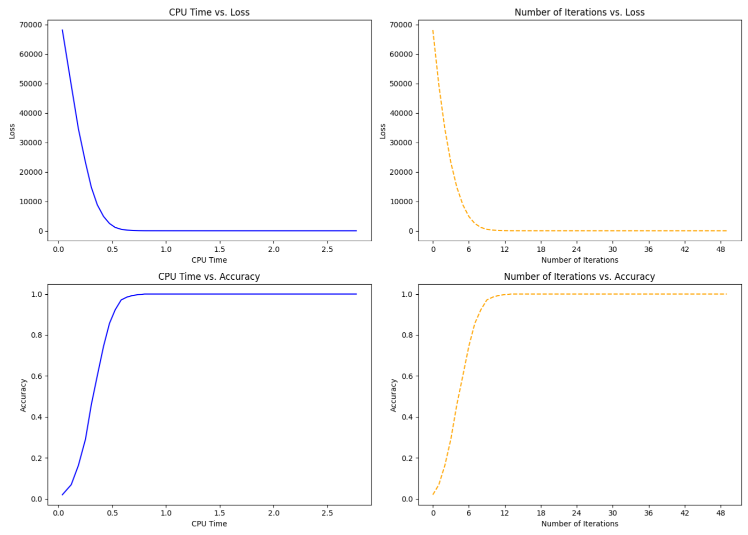

Figure 1: Synthetic dataset: Loss and Accuracy metrics of Gradient Descent

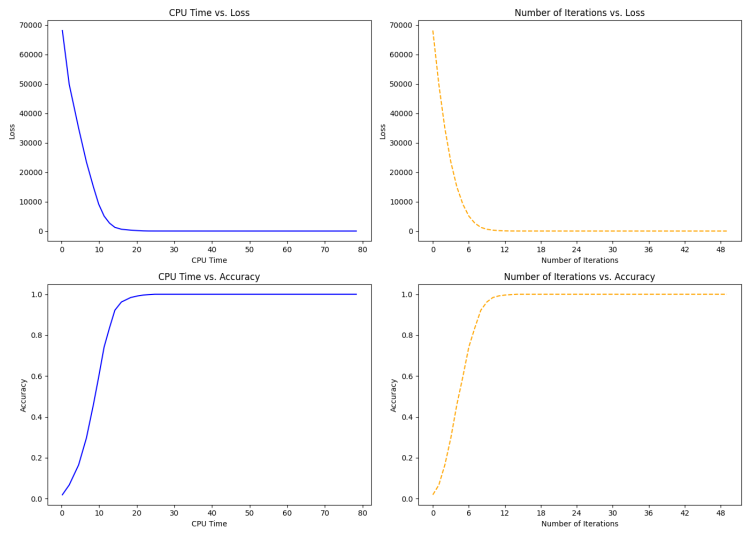

Figure 2: Synthetic dataset: Loss and Accuracy metrics of BCGD Randomized

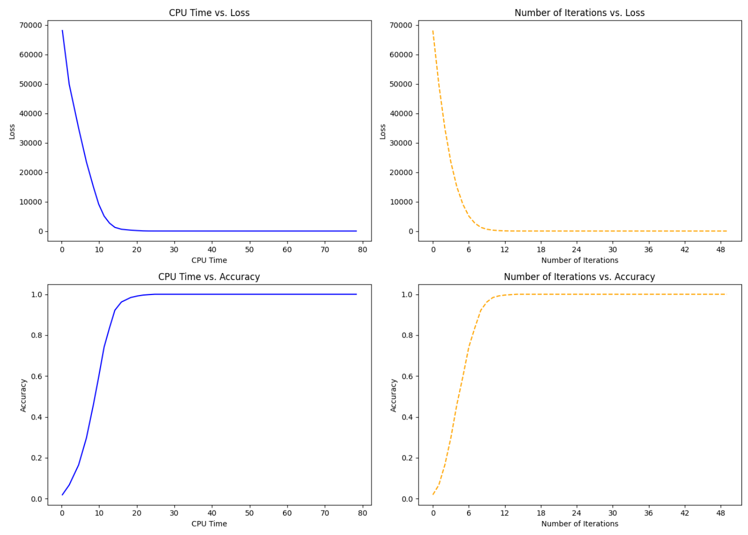

Figure 3: Synthetic dataset: Loss and Accuracy metrics of BCGD Gauss-Southwell

5.2 Results: Real Dataset

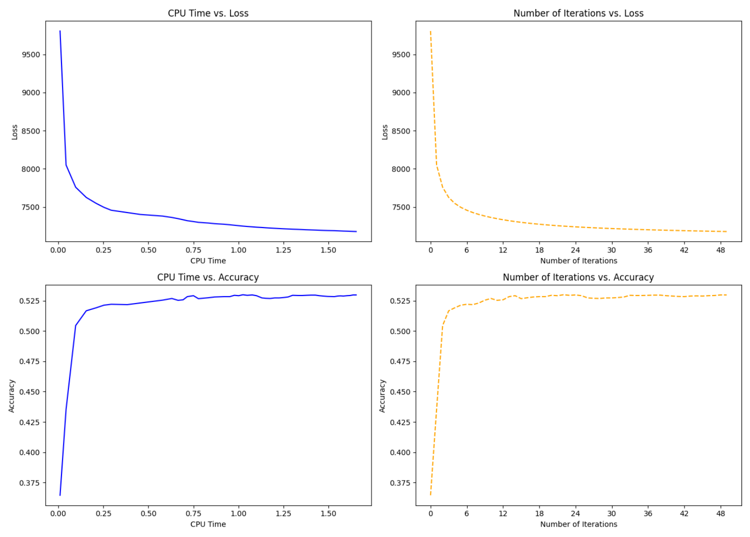

Figure 4: Real dataset: Loss and Accuracy metrics of Gradient Descent

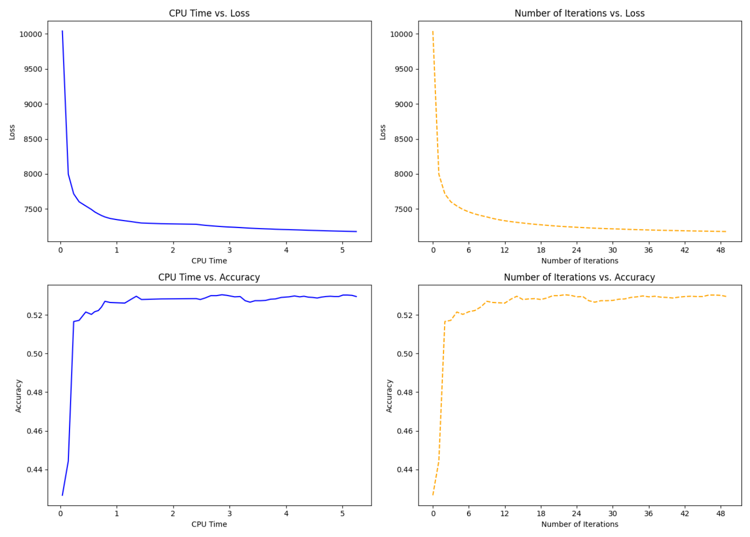

Figure 5: Real dataset: Loss and Accuracy metrics of BCGD Randomized

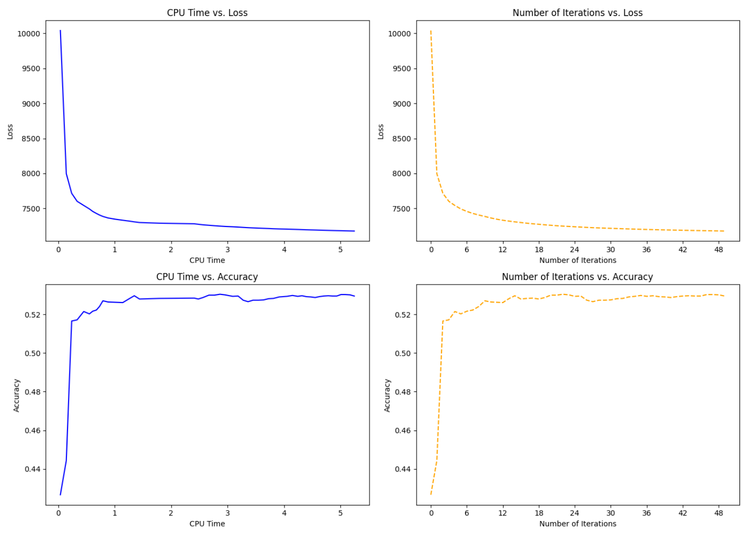

Figure 6: Real dataset: Loss and Accuracy metrics of BCGD Gauss-Southwell

5.3 Results: Compare All Models

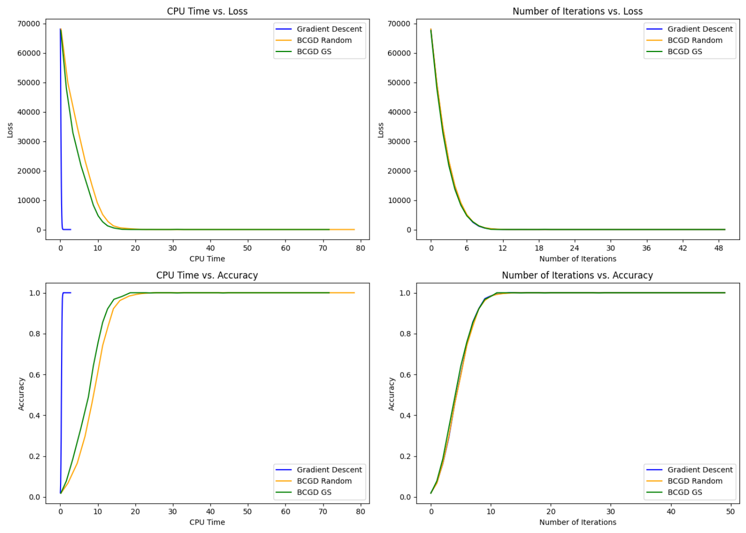

Figure 7: Synthetic dataset: Comparison of all optimization models

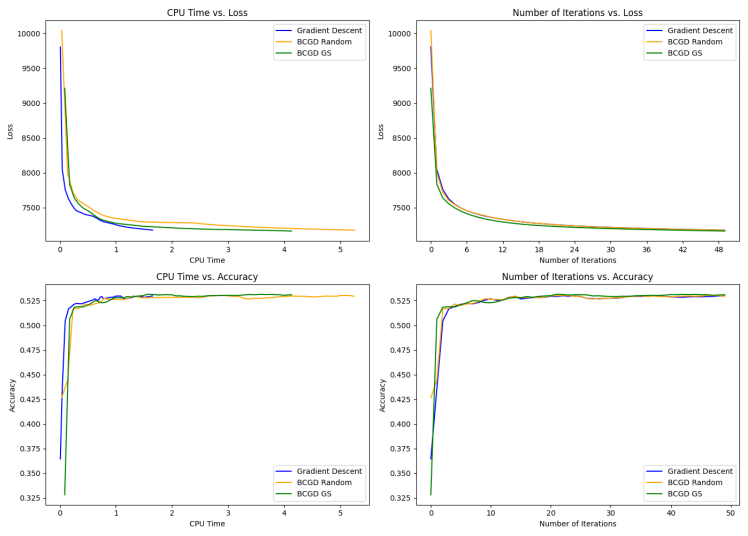

Figure 8: Real dataset: Comparison of all optimization models

6 Discussion

We initially implemented our optimization algorithms on synthetic data that we have randomly generated, and we take note of the good performance of said algorithms expressed in our loss and accuracy metrics. Moreover, we can further prove the efficacy of our algorithms when applied to real-life data. In the case of the real data, an accuracy of 100% is unrealistic, yet the accuracy above 50% is better than random guessing, given that we have 7 present classes (out of 11 possible classes) in total which implies that the models are in fact learning.

After visualizing our results, we observe that the algorithms' performance are very similar with respect to the number of iterations for the synthetic data, while the BCGD Gauss-Southwell slightly outperforms the other two algorithms in the real dataset. However, the general Gradient Descent algorithm performs the best overall in terms of CPU time amongst all the tested algorithms in both synthetic and wine quality datasets, demonstrating the fastest convergence of the loss function and accuracies per unit time.

These convergence rate results partially contradict our intuition that the BCGD algorithms, particularly GS and the block updates would outperform the naive gradient descent approach in terms of cpu time. We can however attribute the observed computational efficiency of the gradient descent algorithm to the Python package, numpy's highly efficient vectorization which outperforms the improvement carried out by the block update.

Additionally, in the case of the synthetic dataset, where the BCGD GS performs only slightly better than BCGD Random in terms of cpu time, the number of blocks is very small, thus it's either cheaper to just use brute force over all the blocks then updating the full gradient, or the problem is very well-conditioned (created A using gaussian noise). Therefore, a sweep over random blocks is almost as good as the greedy policy by GS, and does not justify the additional cost for the gradient update. The idea behind GS is to avoid doing steps in the direction of gradients close to 0, because it's unnecessary computation, thus if all the block gradients are pretty much the same magnitude, then random is close to GS.

In conclusion, our work can be expanded by considering alternative values for our hyperparameters such as the learning rate. Additionally, we can implement other variations of the algorithms we have tested, such as the BCGD Cyclic, Heavy Ball or the Nesterov Accelerated Gradient Descent.