Università degli Studi di Padova

MSc in Data Science

Optimizing multi-class logistic regression with gradient-based methods

Optimization for Data Science

Matteo Gorni Silvestrini 2104372
Tomas Guarini 2125594
Daniele Virzì 2121353
Martino Zaratin 2125047

Academic Year 2023/2024

May 21st 2024

Contents

1 The multi-class logistic regression problem

1.1 Introduction

Gradient-based methods are often considered the best choice when dealing with optimization problems in Data Science. In this homework we will analyze, implement and discuss different gradient-based optimization algorithms applied to a model for a multi-class classification task, i.e. the multi-class Logistic Regression.
First we will introduce the mathematical formulation of this problem, explaining the basic assumptions underlying this model. We will derive some useful properties, which are going to be helpful in the implementation phase. Then we will implement and discuss three gradient-based algorithms: Gradient Descent, Block Coordinate Gradient Descent with Gauss-Southwell rule, Block Coordinate Gradient Descent with random rule. We will evaluate their performance in two different scenarios: on the one hand on a synthetic randomly generated dataset, on the other on a real dataset. In the latter case we will perform a supervised learning task, testing the accuracy of our model on part of the dataset available.
Finally we will confront and discuss all the properties of the above given algorithms.
All the programming part is available in a *Python notebook* which follows the same structure as this report.

1.2 Mathematical formulation

In this section we will try to show how the given optimization problem is the direct consequence of some basic mathematical assumptions, as well as some natural implementation choices, we have to make when modelling the multi-class logistic regression task.
Let's consider the classification task among k different classes. We represent these classes with integers ranging from 1 to k. Suppose then that we are given m independent and identically distributed samples. Each one of these training sample has d features and a known label, indicating the class it belongs to. For each sample i, for $i = 1, ..., m$, we denote with $a_i \in \mathbb{R}^d$ the vector collecting the features of i and we indicate with $b_i \in \{1, \dots, k\}$ the label identifying the class of i.
We collect all the parameters of our model in a matrix $X \in \mathbb{R}^{d \times k}$ and indicate its columns as X_j, for $j = 1, \dots, k$:

$$X = \begin{pmatrix} X_1 & \dots & X_k \end{pmatrix}$$

The first assumption regarding the multi-class logistic regression model is the choice of the probability distribution related to the training set. Given a label $j \in \{1, \dots, k\}$ and $a_i \in \mathbb{R}^d$, the probability of a_i of belonging to class j (i.e. $b_i = j$) is:

$$p(\{b_i = j\}|a_i, X) = \frac{\exp(a_i^T X_j)}{\sum_{c=1}^{k} \exp(a_i^T X_c)}$$

The reasoning behind this is that we consider k different linear channels $X_1 . \dots, X_k$ and what we are doing is basically passing the feature vector through some linear function involving the parameters. Then, for the output to be a probability function, we scale it in the interval $[0, 1]$ with a non-linear mapping (specifically, it is the j-th component of the *softmax* function).

Given this probability distribution, we optimize the multi-class LR by minimizing a loss (cost) function. The chosen loss function is the negative log-likelihood.
First we compute the likelihood $\mathcal{L}(X)$ and log-likelihood $l(X)$:

$$\mathcal{L}(X; a_1, \dots, a_m, b_i) = \prod_{i=1}^{m} p(b_i; X, a_i) = \prod_{i=1}^{m} \left(\frac{\exp(a_i^T X_{b_i})}{\sum_{c=1}^{k} \exp(a_i^T X_c)} \right)$$

$$\begin{aligned} l(X) =& \log \mathcal{L}(X; a_1, \dots, a_m, b_i) = \sum_{i=1}^{m} \log \left(\frac{\exp(a_i^T X_{b_i})}{\sum_{c=1}^{k} \exp(a_i^T X_c)} \right) \\ =& \sum_{i=1}^{m} \left(a_i^T X_{b_i} - \log \left(\sum_{c=1}^{k} \exp(a_i^T X_c) \right) \right) \end{aligned}$$

From here we derive the expression for the cost function $f(X)$:

$$f(X) = - \sum_{i=1}^{m} \left(a_i^T X_{b_i} - \log \left(\sum_{c=1}^{k} \exp(a_i^T X_c) \right) \right)$$

Hence we obtain the formulation of our optimization problem:

$$\min_{X \in \mathbb{R}^{d \times k}} \sum_{i=1}^{m} \left(-a_i^T X_{b_i} + \log \left(\sum_{c=1}^{k} \exp(a_i^T X_c) \right) \right)$$

1.3 Computing the gradient

Since any of the algorithms we are dealing with in this homework are gradient-based methods, it is fundamental to compute the analytical expression of the gradient of the loss function. Notice that our loss function $f\colon \mathbb{R}^{d\times k} \to \mathbb{R}$ is a matrix function, i.e. it takes in input a $d \times k$ matrix, the matrix of parameters, and returns a real number. Hence the gradient of the loss ∇f is well-defined in the following spaces:

$$\begin{aligned} \nabla f\colon \mathbb{R}^{d\times k} &\to \mathbb{R}^{d\times k} \\ X &\mapsto \nabla f(X) \end{aligned}$$

Here we compute the partial derivative of f with respect to X_{jc}, i.e. the entry in the j-th row, c-th column of $\nabla f(X)$:

$$\begin{aligned} \frac{\partial f(X)}{\partial X_{jc}} &= -\frac{\partial}{\partial X_{jc}} \sum_{i=1}^{m} \left(a_i^T X_{b_i} - \log\left(\sum_{c=1}^{k} \exp(a_i^T X_c) \right) \right) = \\ &= -\sum_{i=1}^{m} \left(\frac{\partial (a_i^T X_{b_i})}{\partial X_{jc}} - \frac{\partial}{\partial X_{jc}} \log\left(\sum_{c=1}^{k} \exp(a_i^T X_c) \right) \right) = \\ &= -\sum_{i=1}^{m} \left(a_{ij} \mathbb{I}_{\{b_i=c\}} - a_{ij} \frac{\exp(a_i^T X_c)}{\sum_{c'=1}^{k} \exp(a_i^T X_{c'})} \right) = \\ &= -\sum_{i=1}^{m} a_{ij} \left(\mathbb{I}_{\{b_i=c\}} - \frac{\exp(a_i^T X_c)}{\sum_{c'=1}^{k} \exp(a_i^T X_{c'})} \right) \end{aligned}$$

where $\mathbb{I}_{\{b_i=c\}}$ represents the indicator function.

2 Function vectorization

2.1 Matrix representation of variables and features

Gradient-based methods are usually iterative algorithms which require to compute the gradient of the loss function one time at each iteration. For most of the problems in the real world, the loss function is a multivariate function, so the gradient ends up being an multi-dimensional vector or even a matrix, like in our case. If then the dimensions of the problem are big, or even huge, like in the case of big datasets with lots of features, then computing the gradient one entry at a time for each iteration is infeasible.
To speed up the computation, one possible solution is to *vectorize* the functions involved in the problem. This means writing these functions as composition of simple vector operations, such as matrix-vector operations, scalar products or point-wise products. This way, if there are similar operations that are to be carried out on different entries of some input, they are executed in parallel via basic linear algebra operations. As a consequence, we can avoid implementing these algorithms with many *for loops*, which are in general computationally demanding, and exploit parallel computing of vectorized math, which is usually implemented in an efficient way in most programming languages.

In order to rewrite our relevant functions as vectorized functions, we introduce the following matrices, which are going to be exploited later.
First we collect the data features in a matrix $A \in \mathbb{R}^{m\times d}$, where each row corresponds to the features $a_i \in \mathbb{R}^d$ of a single sample i of the dataset:

$$A = \begin{pmatrix} a_{11} & \dots & a_{1d} \\ \vdots & & \vdots \\ a_{m1} & \dots & a_{md} \end{pmatrix} = \begin{pmatrix} a_1^T \\ \vdots \\ a_m^T \end{pmatrix} \in \mathbb{R}^{m\times d}$$

We already introduced the matrix of parameters $X \in \mathbb{R}^{d\times k}$, obtained by stacking horizontally k vectors of parameters $X_1, \dots, X_k \in \mathbb{R}^d$:

$$X = \begin{pmatrix} x_{11} & \dots & x_{1d} \\ \vdots & & \vdots \\ x_{m1} & \dots & x_{md} \end{pmatrix} = \begin{pmatrix} X_1 & \dots & X_k \end{pmatrix} \in \mathbb{R}^{m\times d}$$

Finally we define a matrix for the class labels. To do so, we consider the *one-hot encoding* of the labels $b_i \in \{1, \dots, k\}$, for $i = 1, \dots, m$. By one-hot encoding we basically mean the operator h from the set of labels $\{1, \dots, k\}$

to the space $\mathbb{R}^k$, which maps a label j to the corresponding basis vector $e_{b_j} \in \mathbb{R}^k$:

$$\begin{aligned} h \colon \{1, \dots, k\} &\to \mathbb{R}^k \\ j &\mapsto e_{b_j} \end{aligned}$$

We compose a matrix $H \in \mathbb{R}^{m \times k}$ by stacking vertically the vectors $h(b_i)$ for $i = 1, \dots, m$, i.e. for all the samples in the dataset:

$$H = \begin{pmatrix} h_{11} & \dots & h_{1k} \\ \vdots & & \vdots \\ h_{m1} & \dots & h_{mk} \end{pmatrix} = \begin{pmatrix} h(b_1)^T \\ \vdots \\ h(b_m)^T \end{pmatrix} \in \mathbb{R}^{m \times k}$$

Notice that, in general, the entry h_{ij} is 1 if the sample i is in the class j, 0 otherwise. So each of the columns of H, let us call them $H_1, \dots, H_k$, contains information about which of all the samples belong to each of the k classes.

2.2 Convexity

The first property we are discussing is convexity of the loss function. This simple property has great implications for the convergence of our algorithms. Indeed we will show that the loss function is convex. Notice that we can rewrite $f(X)$ in the following way:

$$\begin{aligned} f(X) &= \sum_{i=1}^{m} \left(-a_i^T X_{b_i} + \log \left(\sum_{c=1}^{k} \exp(a_i^T X_c) \right) \right) = \\ &= \sum_{i=1}^{m} \left(\log \Big(\exp(-a_i^T X_{b_i}) \Big) + \log \left(\sum_{c=1}^{k} \exp(a_i^T X_c) \right) \right) = \\ &= \sum_{i=1}^{m} \left(\log \left(\sum_{c=1}^{k} \exp(a_i^T (X_c - X_{b_i})) \right) \right) = \\ &= \sum_{i=1}^{m} \left(\log \left(\sum_{c=1}^{k} \exp(a_i^T X (h(c) - h(b_i)) \right) \right) \end{aligned} \tag{1}$$

Now, the function $g(X) = a_i^T X(h(c) - h(b_i))$ is linear, so it is convex. Moreover, it is known that the function

$$z(x) = \log \Big(\sum_{i=1}^{n} \exp(x_i) \Big)$$

is convex (we use Hölder inequality). Then, the expression in (1) is convex too. As a consequence, $f(X)$ is a sum of convex functions and indeed convex itself.

2.3 Vectorizing the loss function

Recall that the loss function we are willing to minimize with our optimization procedure is

$$f(X) = \sum_{i=1}^{m} \left(-a_i^T X_{b_i} + \log \left(\sum_{c=1}^{k} \exp(a_i^T X_c) \right) \right)$$

For simplicity of exposition, from now on the functions exp and log are meant to be element-wise. So whenever one of these two is applied to a vector or a matrix, what we are considering is the result of applying the very same uni-variate function to all the entries of the vector/matrix.

First let's consider the following matrix product:

$$AX = \begin{pmatrix} a_1^T \\ \vdots \\ a_m^T \end{pmatrix} \begin{pmatrix} X_1 & \dots & X_k \end{pmatrix} = \begin{pmatrix} a_1^T X_1 & \dots & a_1^T X_k \\ \vdots & & \vdots \\ a_m^T X_1 & \dots & a_m^T X_k \end{pmatrix}$$

Notice that the arguments of the exponential function in the sum

$$\sum_{c=1}^{k} \exp(a_i^T X_c)$$

are collected in the i-th row of AX. Therefore we can write

$$\sum_{i=1}^{m} \log \left(\sum_{c=1}^{k} \exp(a_i^T X_c) \right) = \sum_{i=1}^{m} \log \Big(\exp(a_i^T X) \cdot \mathbf{1}_k \Big) = \mathbf{1}_m^T \Big[\log \Big(\exp(AX) \cdot \mathbf{1}_k \Big) \Big] \tag{2}$$

where $\mathbf{1}_m \in \mathbb{R}^m$ stands for the m-dimensional vector with 1's as entries. We make use of these vectors to get rid of sums over indices.

Then consider the following product:

$$AXH^T = \begin{pmatrix} a_1^T X_1 & \dots & a_1^T X_k \\ \vdots & & \vdots \\ a_m^T X_1 & \dots & a_m^T X_k \end{pmatrix} \begin{pmatrix} e_{b_1} & \dots & e_{b_m} \end{pmatrix} = \begin{pmatrix} a_1^T X_{b_1} & \dots & a_1^T X_{b_k} \\ \vdots & & \vdots \\ a_m^T X_{b_1} & \dots & a_m^T X_{b_k} \end{pmatrix}$$

Notice that all the entries in the diagonal of the matrix above are the factors that contribute to the sum

$$\sum_{i=1}^{m} -a_i^T X_{b_i}$$

with opposite sign. Therefore

$$\sum_{i=1}^{m} -a_i^T X_{b_i} = \mathbf{1}_m^T \Big[- \operatorname{diag}(AX) \Big] \tag{3}$$

Equations (2) and (3) let us formulate the vectorized expression of the loss function:

$$f(X) = \mathbf{1}_m^T \Big[- \operatorname{diag}(AX) + \log \Big(\exp(AX) \cdot \mathbf{1}_k \Big) \Big]$$

2.4 Vectorizing the gradient

In the previous section we computed an analytical expression for the gradient of the loss function with respect to a single entry of X:

$$\frac{\partial f(X)}{\partial X_{jc}} = - \sum_{i=1}^{m} a_{ij} \left(\mathbb{I}_{\{b_i = c\}} - \frac{\exp(a_i^T X_c)}{\sum_{c'=1}^{k} \exp(a_i^T X_{c'})} \right) \tag{4}$$

Our goal now is to find a mathematical formulation for $\frac{\partial f(X)}{\partial X}$.
Notice that we have already analyzed the sum in the denominators in the previous paragraph (refer to (2)). Indeed

$$\sum_{c'=1}^{k} \exp(a_i^T X_{c'}) = \exp(a_i^T X) \cdot \mathbf{1}_k$$

Now we try to get rid of the sum over the indices i's stacking vertically the components of the sum in (4). We get the following vectors:

$$\begin{pmatrix} a_{1j} \\ \vdots \\ a_{mj} \end{pmatrix} = A_j \in \mathbb{R}^m$$

$$\begin{pmatrix} \exp(a_1^T X_c) \\ \vdots \\ \exp(a_m^T X_c) \end{pmatrix} = \exp(AX_c) \in \mathbb{R}^m$$

$$\begin{pmatrix} \exp(a_1^T X) \cdot \mathbf{1}_k \\ \vdots \\ \exp(a_m^T X) \cdot \mathbf{1}_k \end{pmatrix} = \exp(AX) \cdot \mathbf{1}_k \in \mathbb{R}^m$$

Notice also that

$$\begin{pmatrix} \mathbb{I}_{\{b_1=c\}} \\ \vdots \\ \mathbb{I}_{\{b_m=c\}} \end{pmatrix} = H_c \in \mathbb{R}^m$$

because of the way we defined the label matrix H. This leads us to a vectorized form for $\frac{\partial f(X)}{\partial X_{jc}}$:

$$\frac{\partial f(X)}{\partial X_{jc}} = -A_j^T \Big(H_c - \exp(AX_c) \odot \frac{1}{\exp(AX) \cdot \mathbf{1}_k} \Big)$$

where $\odot$ stands for the Hadamard product (or point-wise product). We can now compute the partial derivative with respect to a colums of parameters X_c, i.e. $\frac{\partial f(X)}{\partial X_c}$. We simply stack vertically the rows A_j^T, for $j = 1, \dots, d$:

$$\frac{\partial f(X)}{\partial X_c} = -A^T \Big(H_c - \exp(AX_c) \odot \frac{1}{\exp(AX) \cdot \mathbf{1}_k} \Big)$$

Finally, we collect side by side all the derivatives with respect to $X_1, \dots, X_k$. Now, in order for the Hadamard product to broadcast point-wise operations, in theory both factors should have the same dimensions. So, for consistency reasons, we introduce an $m \times k$ matrix C, which simply contains k copies of the vector $\frac{1}{\exp(AX) \cdot \mathbf{1}_k}$:

$$C(X) = \begin{pmatrix} \frac{1}{\exp(AX) \cdot \mathbf{1}_k} & \cdots & \frac{1}{\exp(AX) \cdot \mathbf{1}_k} \end{pmatrix} \in \mathbb{R}^{m \times k}$$

This operation is just for theoretical reasons, indeed it won't be implemented in the code. In the end we end up with the desired formula:

$$\frac{\partial f(X)}{\partial X} = -A^T \Big(H - \exp(AX) \odot C(X) \Big) \in \mathbb{R}^{d \times k}$$

2.5 LCG gradient and Lipschitz constant

Now that we have a vectorized expression for the gradient, we will try to show a very useful property of our problem, i.e. that the gradient $\frac{\partial f(X)}{\partial X}$ is indeed a Lipschitz continuous function. To show this, we will derive the value of the Lipschitz constant, which then will be exploited multiple times in the code part.

First we introduce the *softmax function* $\sigma \colon \mathbb{R}^k \to \mathbb{R}^k$, which will come in handy later. Formally

$$\sigma(v) = \sigma \begin{pmatrix} v_1 \\ \vdots \\ v_k \end{pmatrix} = \frac{1}{\sum_{c=1}^{k} \exp(v_c)} \begin{pmatrix} \exp(v_1) \\ \vdots \\ \exp(v_k) \end{pmatrix}$$

As we can see in [1], the softmax function is a Lipschitz function with Lipschitz constant $L = 1$. In other words,

$$||\sigma(x) - \sigma(y)||_2 \leq ||x - y||_2 \quad \forall x, y \in \mathbb{R}^k \tag{5}$$

where we chose the *2-norm* on $\mathbb{R}^k$ to define a metric space. Notice that inequality (5) is equivalent to the following:

$$\sum_{i=1}^{k} \Big(\sigma_i(x) - \sigma_i(y)\Big)^2 \leq \sum_{i=1}^{k} \Big(x_i - y_i\Big)^2 \tag{6}$$

Then, since the gradient of the loss function is a matrix function, it is also necessary to define norms for matrices. Two are the norms we will use. Given $A \in \mathbb{R}^{m \times n}$,

- The **2-norm** of A is
$$||A||_2 = \sup_{\substack{x \in \mathbb{R}^n \\ x \neq 0}} \frac{||Ax||_2}{||x||_2}$$

- The **Frobenius norm** of A is
$$||A||_F = \left(\sum_{i=1}^{n} \sum_{j=i}^{m} |a_{ij}|^2 \right)^{\frac{1}{2}}$$

One can show that, for any matrix A, $||A||_2 \leq ||A||_F$.

It is easy to see that the gradient is continuous, because one can see it as a composition of continuous functions. To show the Lipschitz property, we directly compute the Lipschitz constant. It goes as follows:

$$\begin{aligned}
\left\| \frac{\partial f(X)}{\partial X} - \frac{\partial f(Y)}{\partial Y} \right\|_2 &= \left\| A^T \Big(\exp(AX) \odot C(X) - \exp(AY) \odot C(Y) \Big) \right\|_2 \leq \\
&\leq \left\| A^T \right\|_2 \left\| \exp(AX) \odot C(X) - \exp(AY) \odot C(Y) \right\|_2 \leq \\
&\leq \left\| A^T \right\|_2 \left\| \exp(AX) \odot C(X) - \exp(AY) \odot C(Y) \right\|_F = \\
&= \left\| A^T \right\|_2 \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{k} \Big[\sigma_j(X^T a_i) - \sigma_j(Y^T a_i) \Big]^2} \overset{(*)}{\leq} \\
&\leq \left\| A^T \right\|_2 \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{k} \Big(X_j^T a_i - Y_j^T a_i \Big)^2} = \\
&= \left\| A^T \right\|_2 \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{k} \Big[(X_j^T - Y_j^T) a_i \Big]^2} = \\
&= \left\| A^T \right\|_2 \sqrt{\sum_{i=1}^{m} \left\| (X - Y)^T a_i \right\|_2^2} \leq \\
&\leq \left\| A^T \right\|_2 \sqrt{\left\| X - Y \right\|_2^2 \sum_{i=1}^{m} \left\| a_i \right\|_2^2} = \\
&\leq \left\| A^T \right\|_2 \left\| X - Y \right\|_2 \sqrt{\sum_{i=1}^{m} \left\| a_i \right\|_2^2} = \\
&\leq \left\| A \right\|_2 \left\| A \right\|_F \left\| X - Y \right\|_2 \qquad (7)
\end{aligned}$$

where we used multiple times the fact that $\|Ax\|_2 \leq \|A\|_2 \|x\|_2$ and that $\left\|A^T\right\|_2 = \|A\|_2$ for any matrix A and vector x. In particular, inequality $(*)$ holds because of inequality (6). In this step we use the Lipschitz property of the softmax function.
This chain of inequalities implies that, for the multi-class logistic regression loss function,

$$L = \|A\|_2 \, \|A\|_F$$

Remark 1. *Actually, what we have just shown is that*

$$L \leq \|A\|_2 \, \|A\|_F$$

i.e. we have found an upper bound on the Lipschitz constant. However, there are two reasons, a theorical one and a practical one, that lead us to say that very likely this upper bound is tight:

1. *In the chain of inequalities developed in (7), there is no inequality that is actually slack. By this we mean that, in theory, there should exist a matrix A and matrices X and Y such that all the inequalities above hold with "=" sign. This is due to the way we defined matrix norms $\|\cdot\|_2$ and $\|\cdot\|_F$ and to the fact that the softmax function is Lipschitz with exact constant 1.*

2. *The Lipschitz constant is useful to implement a fixed learning rate in the gradient-based methods. We know that, for a convex objective function, the optimal stepsize is $1/L$. During the implementation phase of the stepsize, we performed a sort of "grid search" over a discrete set of values and we discovered that $1/\|A\|_2 \|A\|_F$ is actually the best one for this hyperparameter, in the sense that it is the best trade-off between performance and convergence.*

3 Datasets

- **Randomly generated dataset**
  In order to create the dataset, the guidelines given in the homework delivery were followed. These instructions are provided below:

  1. *Randomly generate a matrix $A \in \mathbb{R}^{1000\times1000}$ with entries from a $N(0,1)$ distribution.*
     The matrix A refers, as usual, to the matrix containing the features of the data. This means that $m = 1000$ and $d = 1000$. In other words, we are considering a set of 1000 samples with 1000 features each. The features are collected row-wise, i.e. each row of the matrix is a vector of features referring to the same sample.
  2. *Generate $b_i \in \{1, \dots, 50\}$ by computing $AX + E$, with $X \in \mathbb{R}^{d\times k}$ and $E \in \mathbb{R}^{m\times k}$ sampled from a normal distribution as well. To assign class label $b_i = j$ to a sample i, choose the j-th entry of the i-th row of $AX + E$.*
     First of all, $k = 50$. Then, we try to justify the above expression. Recall that

     $$AX = \begin{pmatrix} a_1^T X_1 & \dots & a_1^T X_{50} \\ \vdots & & \vdots \\ a_m^T X_1 & \dots & a_m^T X_{50} \end{pmatrix}$$

     If we look at the i-th row of AX, we can see that it represents the sample a_i "filtered" through the k linear channels we adopted as parameters. In other words, the entry $(AX)_{ij} = a_i^T X_j$ represents the *non-normalized probability* of sample a_i to belong to class j. Therefore it makes sense to define class labels according to the index of the entry with maximum argument, because this conveys the idea of classifying samples according to highest likelihood, once both the samples and the parameters have been fixed.

- **Real world dataset**
  We chose to work with the Optical Recognition of Handwritten Digits dataset, which we found on the UC Irvine Machine Learning Repository.
  The dataset consists of 5620 instances, 64 features and a multi-class label ranging 0-9. We downloaded the data, standardized it using a standard scaler and then performed a train-test split. In this case the variables defined above become: $m = 5620$, $d = 64$ and $k = 10$

4 Optimization via gradient methods

4.1 Hyperparameter selection

As a first step, we talk about the hyperparameters within our algorithms and the values we chose for them:

- Learning rate α: we set $\alpha = \frac{1}{L}$ where L is the Lipschitz constant found in Section 2.5. This fixed learning rate allowed us to have a good trade-off between performance and convergence;

- Maximum number of iterations num_{it}: this value sets how many times our algorithms will iterate at most before stopping, we chose 2000 for all algorithms and for both dataset;

- Degree of tolerance ϵ: since we are working with a finite precision machine this value is usually set as stopping criterion by choosing a very small number that is still bigger than zero, in our case we set $\epsilon = 10^{-6}$.

- Starting point X_0: for the toy dataset we used the Xavier initialization (see [2]), whereas for the real dataset we implemented the He initialization (as explained in [3]). That is essentially to achieve better convergence and preventing exploding or vanishing gradient issues.

- Number of blocks b: we set the blocks as columns of the matrix X, i.e. each block contains only one linear channel. For the randomly generated dataset there are 50 classes, so $k = 50$. Whereas for the real dataset the number of classes is 10, so in this case $k = 10$.

4.2 Algorithm outline

The first algorithm we implement is the classic gradient descent algorithm:

Algorithm 1: Gradient descent

```
  Input: $x_1 \in \mathbb{R}^{d \times k}$
1 for $i = 1, \dots, num_{it}$ do
2     if the norm of the gradient of the objective function is less than $\epsilon$ then
3         STOP;
4     Set $x_{i+1} = x_i - \alpha \nabla f(x_i)$;
```

The pros of this algorithm are the strong mathematical bases and the easy implementation, on the other hand we have to compute the whole gradient and this could lead to a high computational cost.

The second algorithm is the Randomized BCGD:

Algorithm 2: Randomized BCGD method

```
  Input: $x_1 \in \mathbb{R}^{d \times k}$
1 for $i = 1, \dots, num_{it}$ do
2     if the norm of the gradient of the objective function is less than $\epsilon$ then
3         STOP;
4     Randomly pick $j_i \in \{1, \dots, k\}$;
5     Set $x_{i+1} = x_i - \alpha U_{j_i} \nabla_{j_i} f(x_i)$;
```

The randomized BCGD algorithm is less robust than the classic gradient method, but computing partial derivatives instead of the whole gradient makes it much cheaper and less memory demanding. The cons of this algorithm are the dependence on the block sampling and the related cost.

Lastly, we introduce BCGD with Gauss-Southwell rule:

Algorithm 3: Gauss-Southwell BCGD method

```
   Input: $x_1 \in \mathbb{R}^{d\times k}$
1  for $i = 1, \dots, num_{it}$ do
2      if the norm of the gradient of the objective function is less than $\epsilon$ then
3          STOP;
4      Pick a block $j_i \in \{1, \dots, k\}$ such that $j_i = \arg\max_{l\in\{1,\dots,k\}} \|\nabla_l f(x_i)\|$;
5      Set $x_{i+1} = x_i - \alpha U_{j_i} \nabla_{j_i} f(x_i)$;
```

The Gauss-Southwell BCGD method has a good convergence rate when proper conditions are met, but it needs the computation of the whole gradient and searching for the best index could be expensive in huge scale problems.

5 Results

5.1 Randomly generated dataset

Our algorithms appear to have similar convergence rates as it was expected to be; the graphic below illustrates how the norm of the gradient decreases at each iteration. It can be seen that the Gradient Descent is smoother than the BCGD methods due to its strong mathematically bases.

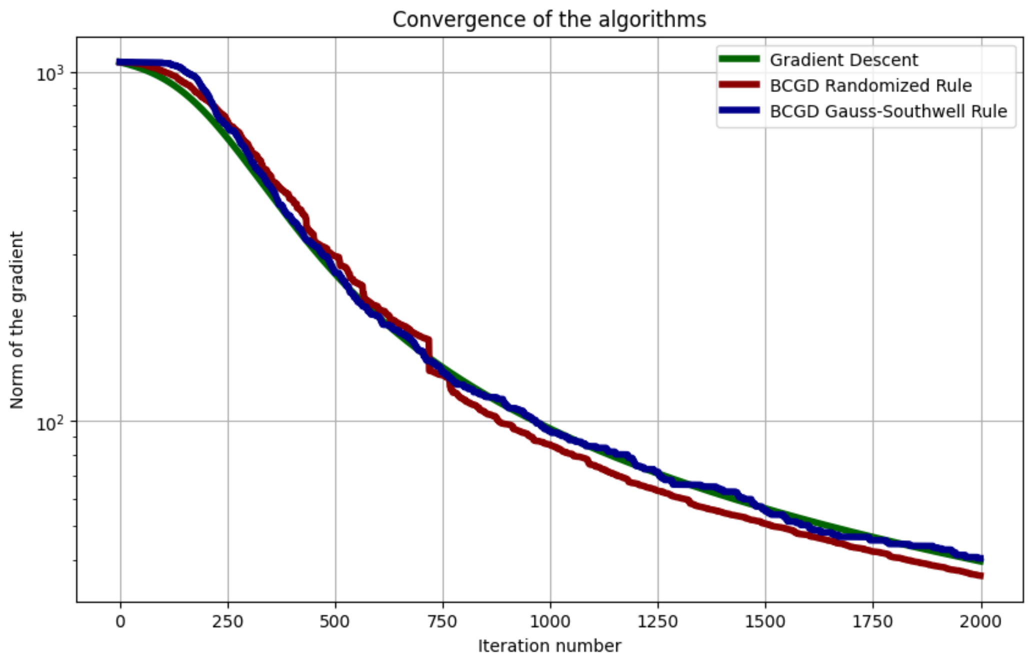

Figure 1: Norm vs Iteration number

Figure 2 highlights a key difference between the two BCGD algorithms and Gradient Descent: BCGD-based methods showcase a much faster decrease of the loss function over time compared to classic gradient descent. This is possible thanks to the reduced number of computations required during each iteration of the BCGD algorithms, thus allowing the calculator to perform many more iterations per second.

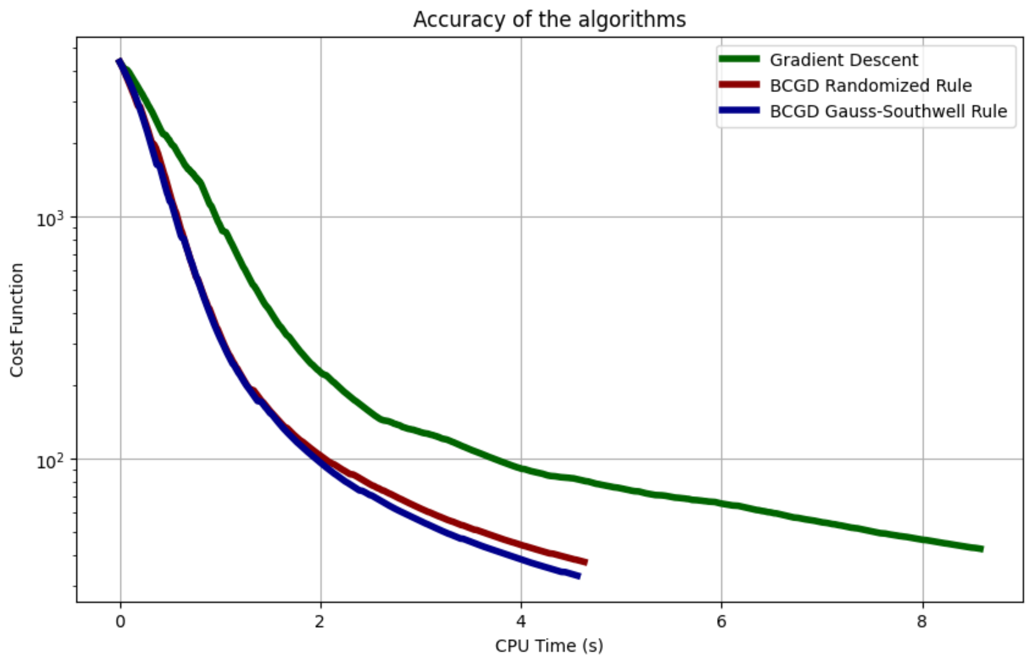

Figure 2: Cost function vs CPU time [s]

	CPU time [s]
Gradient Descent	8.57
Randomized BCGD	4.63
Gauss-Southwell BCGD	4.56

Table 1: CPU running time

As we can see from the table above the Gauss-Southwell BCGD method has the best performance in terms of CPU running time.

5.2 Real Dataset

The algorithms have a similar convergence rate, showcasing a similar behaviour compared to the Toy Dataset plot that was presented above.

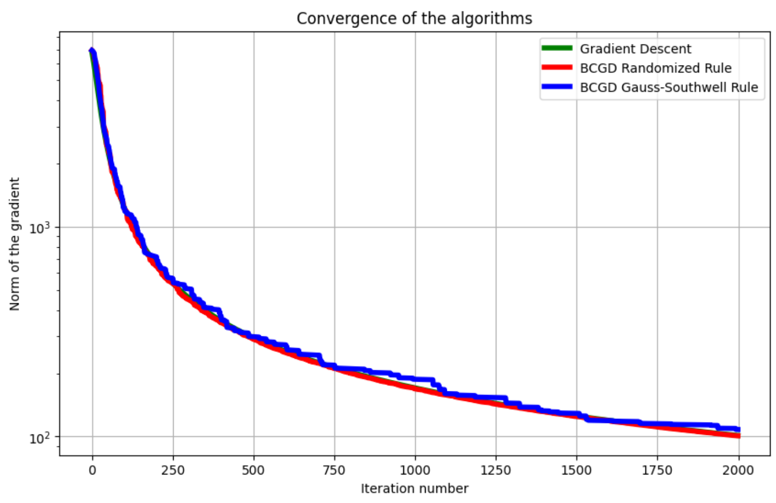

Figure 3: Norm vs Iteration number

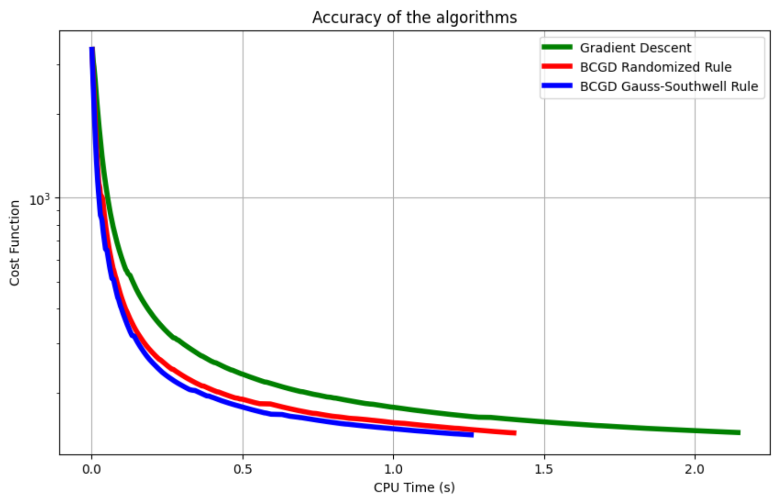

Figure 4: Cost function vs CPU time [s]

	CPU time [s]
Gradient Descent	2.14
Randomized BCGD	1.40
Gauss-Southwell BCGD	1.25

Table 2: CPU running time

As we can see in the Figure 5 and in Table 3 Gradient Descent performs a bit better accuracy-wise, such behaviour is expected given the mathematical robustness of the algorithm.

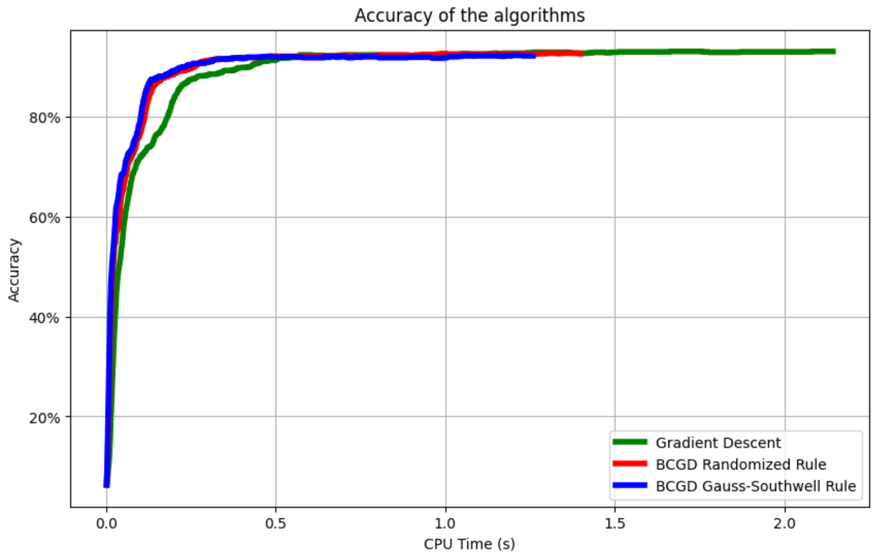

Figure 5: Accuracy vs CPU time [s]

	Accuracy
Gradient Descent	93.1 %
Randomized BCGD	92.6 %
Gauss-Southwell BCGD	92.2 %

Table 3: Accuracy on the test set

In conclusion we saw that all the algorithms are valuable and perform well on both datasets. The difference in terms CPU running time and accuracy are not so relevant in this case, but they may be more significant in huge scale problems.

References

[1] Bolin Gao and Lacra Pavel. On the properties of the softmax function with application in game theory and reinforcement learning. *arXiv preprint arXiv:1704.00805*, 2017.

[2] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.