Optimization for Data Science Homework: Report on Multiclass Logistic Regression Problem

Karim Eugenio Hamdar(**2092041**), Davide Christian Mancosu Bustos(**2089208**), Luca Tusini(**2092227**)

Introduction

This report explores the implementation and resolution of a multiclass logistic regression problem utilizing three distinct optimization techniques: Gradient descent, Block Coordinate Gradient Descent (BCGD) with a randomized rule, and BCGD with the Gauss-Southwell rule. The study involves generating synthetic data, training the models, and assessing their performance on a publicly available dataset. The primary objective is to evaluate both the accuracy of the models and the computational efficiency of each optimization method: the goal is to find the matrix $X \in R^{dxk}$ that minimizes the negative log-likelihood of the given data. For our work we decided to use python and the following are the libraries that we used:

```
# libraries
import xlrd
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
import time
```

Problem Description

The optimization problem is formally defined as:

$$\mathrm{f(x)} = \min_{x \in R^{dxk}} \sum_{i=1}^{m} \left[-x_{b_i}^T a_i + \log\left(\sum_{c=1}^{k} e^{x_c^T a_i} \right) \right]$$

where:

- m is the number of training samples,
- d is the dimensionality of the feature vector a_i,
- k is the number of classes,
- $a_i \in R^d$ is the feature vector for the *i-th* training sample,
- $b_i \in \{1,2,\dots,k\}$ is the class label for the *i-th* training sample,
- x_c is the *c-th* column of the matrix X.

The likelihood of a single training example *i* with features a_i and label b_i is given by:

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

```python
# Function to compute softmax row-wise of an input matrix
def softmax(matrix):
    return np.array([np.exp(row) / sum(np.exp(row)) for row in matrix]).reshape(matrix.shape)
```

The task involves minimizing the negative log-likelihood over m independent and identically distributed training samples, expressed as:

$$f(x) = \sum_{i=1}^{m} \left[-x_{b_i}^T a_i + \log\left(\sum_{c=1}^{k} e^{x_c^T a_i} \right) \right]$$

```python
# Function to compute negative log-likelihood
def loss(A, B, X):
    # NOTICE: B must be in one-hot encoding for
    loss = (np.trace(A @ X @ B.T) + np.sum(np.log(np.sum(np.exp(- np.dot(A, X)), axis=1))))
    return np.round(loss,2)
```

In the experimental section, we will:

1. Generate a synthetic dataset by creating a 1000x1000 matrix with entries drawn from a normal distribution *N(0,1)*;
2. Simulate class labels by computing $AX + E$, where *X* and *E* are matrices sampled from a normal distribution and assign the class label based on the maximum index in each row.
3. Apply the three optimization algorithms to solve the problem.
4. Validate our model on a publicly available dataset, analysing the accuracy and CPU time for each method.

Optimization Techniques

1. **Gradient Descent (GD)**:
   We implemented standard gradient descent to minimize the negative log-likelihood function. In this method, we iteratively update the entire matrix $X \times X$ by taking steps proportional to the negative gradient of the objective function. This approach ensures that all parameters are updated simultaneously, leading to potentially faster convergence. However, it requires the computation of the full gradient at each iteration, which can be computationally expensive for large datasets. The gradient is as follow:

$$\frac{\partial f(x)}{\partial X_{jc}} = -\sum_{i=1}^{m} [I(b_i = c) - \frac{e^{x_c^T a_i}}{\sum_{c'=1}^{k} e^{x_{c'}^T a_i}}]$$

2. **Block Coordinate Gradient Descent (BCGD) with Randomized Rule:**

   In the BCGD with the randomized rule, we update the matrix $X \times X$ one block at a time, where each block corresponds to a row of $X \times X$. At each iteration, we randomly select one block and perform gradient descent on that block. This method can be implemented in two ways:

- **Full Gradient Calculation**: Calculate the full gradient of the objective function and then update only the selected row. This approach ensures that the update is based on accurate gradient information but can be computationally intensive as it requires computing the full gradient.

```
# Function to compute the gradient
def gradient(A, B, X):
    # NOTICE: B must be in one-hot encoding for
    P = softmax(- np.dot(A, X))
    gd = (A.T @ (B - P))
    return gd
```

- **Column-wise Selection**: Select a column and update the corresponding parameter without computing the full gradient. This approach reduces computational cost but updates fewer parameters at a time, potentially slowing down convergence.

```
# Function to compute the gradient of just one column of the parameters
def gradient_col(A, B, X, index):
    # NOTICE: B must be in one-hot encoding for
    P = softmax(- np.dot(A, X))
    gd = (A.T @ (B[:,index] - P[:,index]))
    return gd
```

3. **BCGD with Gauss-Southwell Rule**:

   The BCGD with the Gauss-Southwell rule aims to improve convergence by selecting the most promising block for updates. At each iteration, we select the block (a row of $X \times X$) with the largest gradient norm, indicating the direction with the steepest descent. By focusing on the block with the highest potential for reducing the objective function, this method can achieve faster convergence compared to random selection. However, it requires computing the gradient norm for all blocks at each iteration, which can be computationally expensive.

Implementation

The implementation involves writing Python code to:

- Generate the synthetic data. For the generation of the data we used the numpy.random.randn() function:

```
# Constants
m, d, k = 1000, 1000, 50
# randomly generate a 1000x1000 matrix with entries from a N(0,1) distribution
A = np.random.randn(m, d)
# Generate random matrices
X = np.random.randn(m, k)

# Generate random matrices
W = np.random.randn(d, k)
E = np.random.randn(m, k)

# Compute AX + E
AX_plus_E = np.dot(A, X) + E

# Find max index in each row
B = np.argmax(AX_plus_E, axis=1) # <-- range from 0 to 49

print("Class labels shape:", B.shape)
print("Class labels:", B[:10])

[]Class labels shape: (1000,)
[]Class labels: [ 1  6 32 10 26 45 27  4 23 37]
```

- Implement each optimization technique (we provide the gradient descent implementation as an example below).

```
def gradient_descent(A, B, alpha, max_iter, threshold):

    B_onehot = one_hot_encode(B)

    # different ways to initialize the parameters
    X = np.zeros((A.shape[1], B_onehot.shape[1]))
    # X = np.random.rand(A.shape[1], B_onehot.shape[1])
    # X = np.random.normal(1, 1, (A.shape[1], B_onehot.shape[1]))

    step = 0
    step_lst = []
    loss_lst = []
    acc_lst = []

    init_time = time.time()
    ticks_gd = [] # CPU time

    # print(f"Initial accuracy: {accuracy(A,X,B)}%\n")

    while step < max_iter:
        step += 1
        step_lst.append(step)

        X -= alpha * gradient(A, B_onehot, X)

        loss_lst.append(loss(A, B_onehot, X))

        acc_lst.append(accuracy(A, X, B))

        print(f"Step: {step} ------------ Loss: {loss_lst[-1]} ------------ Accuracy: {acc_lst[-1]}%")
        ticks_gd.append(time.time() - init_time)

        # Stopping condition over accuracy
        if acc_lst[-1] >= threshold:
            print(f"\nStopping Condition reached at iteration {step}: accuracy = {acc_lst[-1]}%")
            break

    df = pd.DataFrame({
        'step': step_lst,
        'loss': loss_lst,
        'accuracy': acc_lst,
        'CPU': ticks_gd
    })

    return df, X
```

we decided to use the one-hot encode method and apply it to our label vector. Briefly one-hot encoding is a method to represent categorical variables as binary vectors. Each category is transformed into a binary vector, where only one bit is '1' (indicating the presence of that category) and the rest are '0'. This technique is essential for machine learning algorithms to process categorical data, enabling them to understand and analyse categorical variables in a numerical format:

```
# Convert a vector of random numbers into a one-hot encoding version.
def one_hot_encode(vector):
    unique_values = np.unique(vector)
    num_unique = len(unique_values)
    num_elements = len(vector)

    one_hot = np.zeros((num_elements, num_unique))

    for i, value in enumerate(vector):
        index = np.where(unique_values == value)[0][0]
        one_hot[i, index] = 1

    return one_hot
```

- Implement each optimization technique by training the model using a dedicated class. This approach ensures systematic organization and clear separation of concerns, allowing for efficient experimentation and comparison of results across different optimization methods:

```
class Multiclass:

    def fit(self, A, B, type, alpha, max_iter, threshold):
        if type=="GD":
            self.loss_steps, self.X = gradient_descent(A, B, alpha, max_iter, threshold)

        elif type=="BCGD_R":
            self.loss_steps, self.X = R_bcgd(A, B, alpha, max_iter, threshold)

        elif type=="BCGD_GS":
            self.loss_steps, self.X = GS_bcgd(A, B, alpha, max_iter, threshold)

    def plot(self, w):
        if w == "Loss":
            return self.loss_steps.plot(
                x='step',
                y='loss',
                xlabel='Step',
                ylabel='Loss',
                color='orange',
                title='Iterations VS Loss')

        elif w == "Accuracy":
            return self.loss_steps.plot(
                x='step',
                y='accuracy',
                xlabel='Step',
                ylabel='Accuracy',
                color='orange',
                title='Iterations VS Accuracy')

        elif w == "CPU":
            return self.loss_steps.plot(
                x='CPU',
                y='accuracy',
                xlabel='CPU time',
                ylabel='Accuracy',
                title='CPU time VS Accuracy',
                color='orange',
                legend=False)

    def backup(self):
      return self.loss_steps

    def predict(self, H):
        Z = - H @ self.X
        P = softmax(Z)
        return np.argmax(P, axis=1)
```

- Evaluate the performance on a publicly available dataset plotting the loss and accuracy and analysing the results obtained in the test set (see below).

Public Dataset

Regarding the second part of the homework, we have decided to test our models on multiple public datasets, all focused on multiclass classification. Here, and in the code file, we report only the two datasets that yield the best results in terms of time taken and accuracy.

Dry Bean Dataset

(link to the dataset: https://archive.ics.uci.edu/dataset/602/dry+bean+dataset)

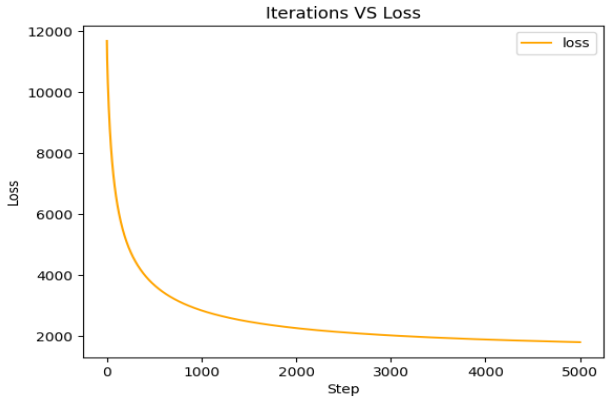
The first dataset comprises 13,611 observations, each with 17 features related to dry beans, which need to be classified into one of seven classes. After an initial Exploratory Data Analysis (EDA) phase, where we checked for potential issues such as missing values, imbalanced classes, and non-numerical features, we proceeded to train the models. To avoid interruptions and disconnections on Google Colab, we decided to use only 9,000 out of the 13,611 observations. We then randomly split the data into training and test sets using scikit-learn's Python methods, adhering to the common practice of allocating 30% of the data for the test set.

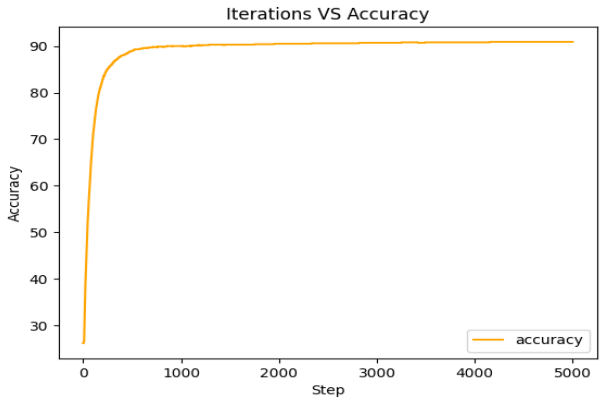
We initially applied classic gradient descent, followed by BCGD with a randomized rule and GS scheme. The primary challenge we faced across all scenarios was the significant variability in results depending on the chosen alpha value. As anticipated, a high alpha value induced strong oscillations in

accuracy measures, effectively halting the learning process. Conversely, a low alpha value resulted in a slow training process, necessitating more iterations to achieve satisfactory models. We opted for a fixed step size of 0.0001 and a maximum of 5000 iterations.

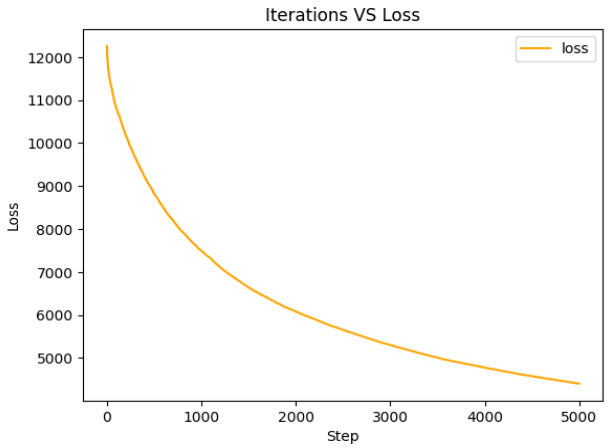
We now present the results obtained from the models and provide our commentary on them.

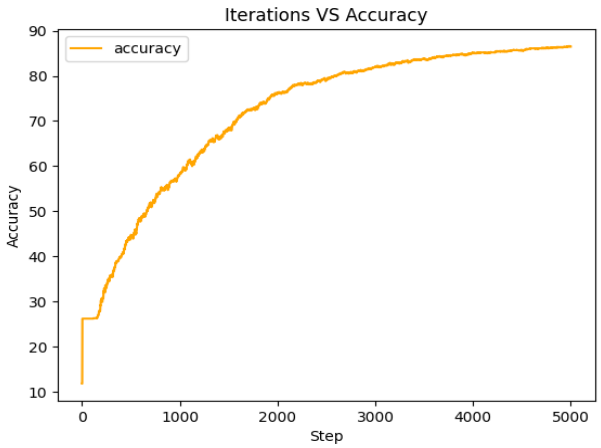
Classic Gradient Descent:



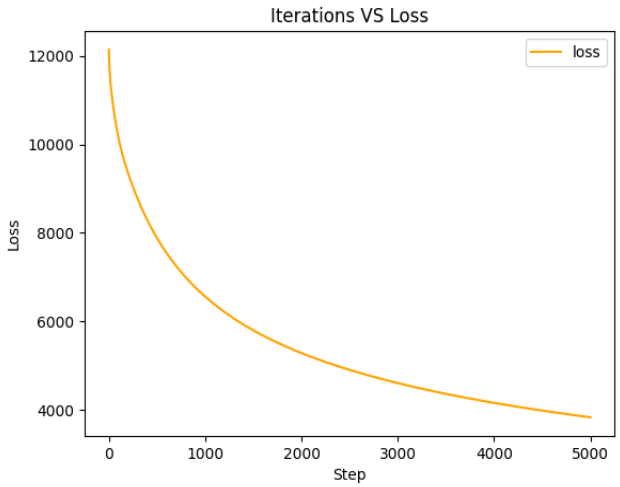


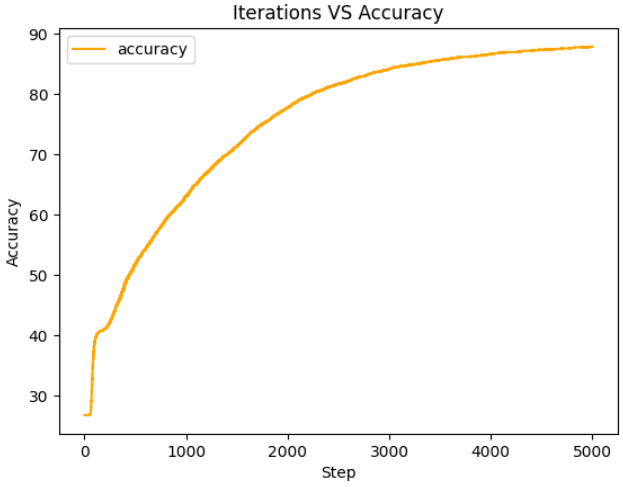
BCGD with Randomized rule:



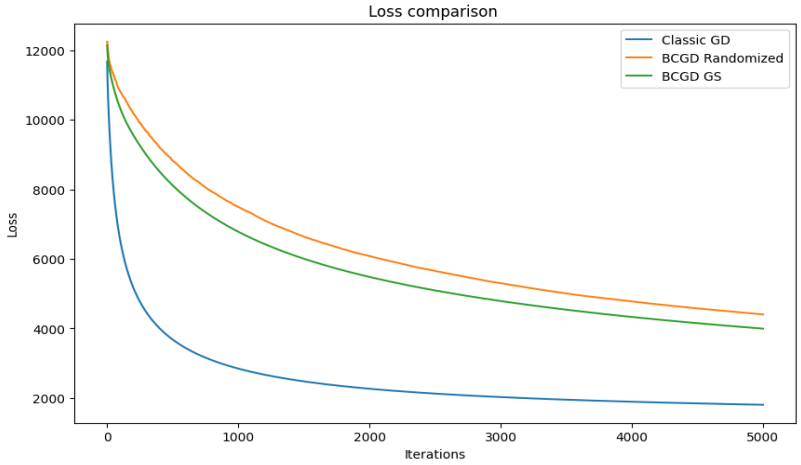


BCGD with GS rule:





Comments on the results:



This final plot demonstrates the distinct behaviours of the three models. There are no specific anomalies to note; however, it is immediately apparent that the blue line, representing the classic gradient descent method, is much steeper than the other two lines, which are identical. This difference can be attributed to both theoretical and practical reasons.

Firstly, it's important to note that BCGD methods are designed to avoid computing the entire gradient, which is very costly for ill-conditioned problems, thereby reducing computational expense. Given that our scenario involves significantly fewer features than observations, we expected the gradient descent method to converge faster than the others.

A more suitable choice in this scenario might be a Stochastic Gradient Descent (SGD) method, considering the large number of observations in the original dataset (also true for the other dataset we tested).

From a practical perspective, it is also important to note that we did not apply any line search. Instead, we tested different alpha values through grid search and selected the one that best fit our needs.

Regarding accuracy, we observed that all three methods achieved around 90% accuracy on the test set. The best method was gradient descent (92%), followed by the Gauss-Seidel scheme (90%), and lastly the randomized method (89%). Note that we set the maximum number of iterations to 5000, which is an indicative value, but increasing this threshold could potentially lead to further improvements in accuracy.

The CPU time for the methods was not significantly different; all methods took approximately 24 minutes to train, except for the randomized method, which took considerably less time due to the partial computation of the gradient. For this last method, it is important to note that we set a seed to ensure the experiments were reproducible. With the chosen seed, the Randomized BCGD performed better than the GS scheme. This may be due to the way we selected the seed. We tried different seeds, and it appears that most led to similar outcomes, with a few exceptions.

Finally, we tested the speed of convergence for these methods based on a stopping condition determined by accuracy. We allowed the models to run until the accuracy on the training set reached a predetermined threshold to evaluate which method performed the best. The fixed threshold was 90%, so we trained the models until they achieved 90% accuracy on the training set.

Here are the results obtained:

Gradient Descent (GD)		**BCGD with Randomized Rule**		**BCGD with Gauss-Southwell Rule**	
CPU Time	Acc. On Test Set	CPU Time	Acc. On Test Set	CPU Time	Acc. On Test Se
200.88 sec.	88.9%	1495.99 sec	86.5%	1852.98 sec	86.6%

As expected, the classic gradient descent method outperforms the other two models. It takes only 201 seconds (3 minutes and a half) to reach the stopping conditions and also delivers the best performance on the test set. In contrast, the randomized method takes about 24 minutes, and the GS method takes around 30 minutes.

Ultimately, all models perform similarly on the test set, showing no significant differences. We could likely achieve better results on the test set by allowing the models more time to tune their parameters, as we did in the previous section.

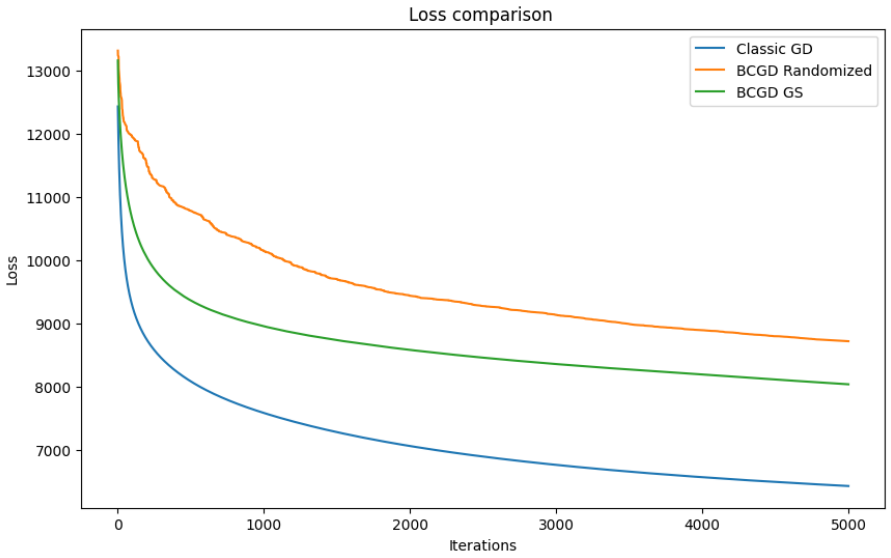
Microbes Dataset

(link to the dataset: https://www.kaggle.com/datasets/sayansh001/microbes-dataset)

The second and final dataset consists of 30,527 observations, each with 26 features related to microorganisms, which need to be classified into one of 5 microbe classes. After a brief preliminary EDA phase, we trained the models using only 10,400 observations, this choice is due to the fact than in this dataset we have much more variables and so the time required is much more. These were randomly split into training and test sets, following the standard practice.

As before, we first applied the classic gradient descent method, followed by the two versions of the BCGD method.

For convenience this time, we will only report the final plot, which includes a comparison of the three models and the test accuracy obtained.



Again the first thing that comes immediately to mind is the steep of the blu line related to the gradient descent that outperform the other methods. Another notable fact is that in this scenario the scale of the loss is much bigger: before GD model reach 2000 as loss function while now all models return a quite high value of the loss (around 9000). This may be due to several factors such as the preprocessing phase in which we deleted some variables and apply min max scalex for example, but also because the dataset may be much more complex.

Finally let's notice how in this scenario the BCGD with GS scheme, take also more time than the others models, performs better than the Randomized, maybe 'cause of the structure of the problem.

Coming to the accuracy measures, in 5000 iterations, models all return quite bad accuracy over the test set but as said before likely letting the models train for more time may lead to better measures.

Here the results obtained:

Gradient Descent (GD)		**BCGD with Randomized Rule**		**BCGD with Gauss-Southwell Rule**	
CPU Time	Acc. On Test Set	CPU Time	Acc. On Test Set	CPU Time	Acc. On Test Se
2570.6 sec.	69.04%	2250.14 sec	55.29%	3039.19 sec	57.84%

Conclusion

In conclusion, this report highlights the implementation and comparative analysis of three optimization techniques: Gradient Descent (GD), Block Coordinate Gradient Descent (BCGD) with a randomized rule, and BCGD with the Gauss-Southwell rule applied to a multiclass logistic regression

problem. Our findings demonstrate that while Gradient Descent achieved the fastest convergence and highest accuracy on both synthetic and real-world datasets, the BCGD methods offered significant computational savings, particularly the randomized version. The Gauss-Southwell rule, although computationally intensive, also showed competitive performance.

The choice of optimization method should be guided by the specific requirements of the problem, including the trade-off between computational efficiency and accuracy. We tested the optimization algorithms with different real datasets, showing how the performance changes significantly based on them. These differences could exist for several reasons: the presence of autocorrelation in the data, outliers, the necessity of more pre-processing, and also the implicit structures of the dataset variables. This study underscores the importance of algorithm selection and parameter tuning in optimizing multiclass classification tasks.