Multi-class Logistic Regression

Alberto Calabrese, Greta d'Amore Grelli, Marlon Helbing,
Eleonora Mesaglio

21 May 2024

Let us consider m samples described by d features and divided in k classes. We want to solve the multi-class classification problem of the form:

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

where $a_i \in \mathbb{R}^d$ are the features of the i-th sample, $x_c \in \mathbb{R}^d$ is the column vector of the matrix of parameters $X \in \mathbb{R}^{d\times k}$ relating to class c and $b_i \in \{1, \dots, k\}$ is the label associated to the i-th sample, given by the following probability:

$$P(b_i|X, a_i) = \frac{\exp(x_{b_i}^T a_i)}{\sum_{c=1}^{k} \exp(x_c^T a_i)}. \tag{2}$$

In this report we will describe how we implemented Gradient Descent and Block Coordinate Gradient Descent - both with randomized rule and Gauss-Southwell rule - to solve the given problem (1) and we will discuss the results obtained testing it on a dataset of our choice.

1 Generating the dataset

The first and second tasks of the assignment require to randomly generate the matrices $A \in \mathbb{R}^{m\times d}$ of the features, $X \in \mathbb{R}^{d\times k}$ of the parameters and $E \in \mathbb{R}^{m\times k}$, with entries from normal distribution. So, we define the matrices A, X and E as shown in the following code:

```
    A = np.random.normal(0, 1, size = (m, d))
    X = np.random.normal(0, 1, size = (d, k))
    E = np.random.normal(0, 1, size = (m, k))
```

Now, the labels b_i of the i-th sample are chosen by considering i-th row of the following matrix

$$B = AX + E \tag{3}$$

and taking as b_i the column index with maximum value. We generate those labels by creating the vector $labels \in \mathbb{R}^m$ as

```
    labels = np.argmax(B, axis=1)
```

that in position i has the label b_i of the i-th sample.

Once completed these two tasks, the dataset is generated.

2 Cost function and gradient calculations

In order to construct an efficient algorithm, it is useful to re-write the functions we need to implement (in our case the cost function f and the respective gradient) in a more compact way.

Let us start with the cost function f:

$$\begin{aligned} f(x) &= \sum_{i=1}^{m} \left[-x_{b_i}^T a_i + \log\left(\sum_{c=1}^{k} \exp(x_c^T a_i) \right) \right] \\ &= \sum_{i=1}^{m} \left(f_i^1 + f_i^2 \right) \\ &= \left(f^1 + f^2 \right)^T \mathbf{1}_m, \end{aligned}$$

where $\mathbf{1}_m \in \mathbb{R}^m$ is a vector only containing 1's and $f^1, f^2 \in \mathbb{R}^m$ have in position i, respectively, f_i^1 and f_i^2.

To obtain the first term, f^1, we have to consider the column of the matrix X corresponding to the class b_i of the i-th sample, x_{b_i} and the vector of features a_i, that is the i-th column of the matrix A^T. We can use the matrices A, X and a matrix that contains the information about the classes of each sample to extract the elements that we want. Indeed, defining the indicator vectors $I_{b_i} \in \mathbb{R}^k$ as

$$(I_{b_i})_j = \begin{cases} 1 & \text{if } b_i = j \\ 0 & \text{otherwise} \end{cases} \qquad \text{for } j = 1, \ldots, k$$

and so the indicator matrix I_b as

$$I_b = \left[I_{b_1}, \ldots, I_{b_m} \right], \tag{4}$$

we get a matrix in $\mathbb{R}^{k \times m}$ that encodes for each sample i the relative class b_i. At this point, it is easy to see that the elements we need are on the diagonal of the matrix $-(AX)^T \cdot I_b$. Therefore, we are able to re-write the vector f^1 making use of the fact that

$$f_i^1 = -\left[(AX)^T \cdot I_b \right]_{ii}.$$

From now on, we will use the following notation for f^1:

$$f^1 = -\text{diag}\left((AX)^T \cdot I_b \right). \tag{5}$$

Now, let us focus on the second term f^2. Taking into consideration the same matrices A, X as before and using a vector $\mathbf{1}_k \in \mathbb{R}^k$ to sum over the rows of the matrix $\exp(AX)$, we can simply write

$$f^2 = \log(\exp(AX) \cdot \mathbf{1}_k). \tag{6}$$

All in all, our cost function has the following form:

$$f = \left(-\text{diag}\left((AX)^T \cdot I_b\right) + \log\left(\exp(AX) \cdot \mathbf{1}_k\right)\right)^T \cdot \mathbf{1}_m. \tag{7}$$

In code,

```
term_1 = -1 * (np.diag((A @ X) @ I_b))
term_2 = np.exp(A @ X) @ np.ones((k,1))
f = (term_1 + (np.log(term_2)).flatten()) @ np.ones((m,1))
```

Let us proceed to discuss about the gradient of the cost function f. The partial derivative with respect to x_{jc}, in which $j \in \{1, \dots, d\}$ indicates the feature and $c \in \{1 \dots, k\}$ indicates the class, has the form

$$\frac{\partial f}{\partial x_{jc}}(x) = -\sum_{i=1}^{m} a_{ij} \left(I(b_i = c) - \frac{\exp(x_c^T a_i)}{\sum_{c'=1}^{k} \exp(x_{c'}^T a_i)} \right), \tag{8}$$

where $I(b_i = c)$ assumes the value 1 if the class of the sample b_i is equal to c and 0 everywhere else. From this formula we can obtain the full gradient with respect to X, that is the matrix $G \in \mathbb{R}^{d \times k}$ with element (8) in position G_{jc}. In particular, thanks to the indicator matrix I_b we previously defined, (4), and some mathematical calculations the matrix G can be written as follows:

$$G = A^T \left(I_b^T - \frac{\exp(AX)}{\exp(AX) \cdot \mathbf{1}_k} \right), \tag{9}$$

that we implemented in code as

```
def full_gradient(X,A,labels):
    return (-1 * A.T) @ (I - ((np.exp(A @ X)) / (np.exp(A @ X)
           @ np.ones((k,1)))))
```

2.1 Lipschitz constant

We can prove that f has Lipschitz continuous gradient with constant

$$L = ||A||_2 \, ||A||_F \,, \tag{10}$$

where $||\cdot||_2$ is the Euclidean norm and $||\cdot||_F$ is the Frobenius norm, defined as

$$||A||_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{k} |a_{ij}|^2}$$

for a matrix $A \in \mathbb{R}^{m \times k}$.

Let us introduce the following notation for the softmax function

$$s(x) = \frac{\exp(x)}{\exp(x) \cdot \mathbf{1}_k}.$$

Indeed:

$$\begin{aligned}||G(X)-G(Y)||_2 &= \left|\left|A^T\left(I_b^T - s(AX)\right) - A^T\left(I_b^T - s(AY)\right)\right|\right|_2 \\ &= \left|\left|A^T\left(s(AX)-s(AY)\right)\right|\right|_2 .\end{aligned}$$

Thanks to some basic properties of the Euclidean and Frobenius norm, in (11), and using the fact that $s(x)$ is 1-Lipschitz, in (12), we can obtain the bound:

$$\begin{aligned}||G(X)-G(Y)||_2 &\le ||A||_2\,||s(AX)-s(AY)||_F && (11)\\ &= ||A||_2\sqrt{\sum_{i=1}^{m}\sum_{j=1}^{k}\left[s_j(X^Ta_i)-s_j(Y^Ta_i)\right]^2} \\ &\le ||A||_2\sqrt{\sum_{i=1}^{m}\sum_{j=1}^{k}\left[X^Ta_i-Y^Ta_i\right]^2} && (12)\\ &\le ||A||_2\,||X-Y||_2\sqrt{\sum_{i=1}^{m}||a_i||_2^2} \\ &= ||A||_2\,||A||_F\,||X-Y||_2 .\end{aligned}$$

Hence, $L = ||A||_2\,||A||_F$. This constant will be useful during the implementation of our algorithms to find the best learning rate.

```
    def lipschitz(A):
        L = np.linalg.norm(A,2) * np.linalg.norm(A,'fro')
        return L
```

This said, we can start working with Gradient Descent optimization methods.

3 Gradient Descent - GD

Gradient Descent algorithm has the following structure:

```
1. Choose a point $x_1 \in \mathbb{R}^n$
2. For $h = 1, \dots$
3.      If $x_h$ satisfies some specific condition, then stop                (13)
4.      Set $x_{h+1} = x_h - \alpha_h \nabla g(x_h)$ with $\alpha_h > 0$ stepsize
5. End for
```

For a differentiable convex function $g : \mathbb{R}^n \to \mathbb{R}$ and an appropriate choice of stopping criterion in step 3, learning rate $\alpha_h > 0$ in step 4, this method converges to a good approximation of the optimal solution of the minimization problem

$$\min_{x\in\mathbb{R}^n} g(x). \tag{14}$$

Notice that with GD we update x_h using the full gradient.

Our cost function f is indeed convex, since it is sum and composition of convex functions, and it also is trivially differentiable. Thus, we can apply the scheme in (13) to f to find a good approximation of the solution of our problem (1). In fact, we introduce a parameter $EPSILON$ (in our code that is assigned a value of $1 \cdot 10^{-6}$) to set the stopping condition in line 3 to be

```
    norm = np.linalg.norm(grad)
    if norm < EPSILON:
        break
```

In other words, if the norm of the gradient of our function is close enough to zero then we stop our search. Furthermore, we choose a fixed stepsize

```
    LR = 1/lipschitz(A)
```

where $lipschitz(A)$ is the Lipschitz constant previously defined, and a maximum number of iterations

```
    ITERATIONS = 3000
```

The updating rule has the following form:

```
    X = X - LR * grad
```

where $grad = full\_gradient(X, A, labels)$. Moreover, in our gradient descent we also keep track of our values x_h, to calculate the cost function $f(x_h)$, the norm of the gradient, to plot our results, and the time, to check the speed of our algorithm, per each iteration.

In order to facilitate more straightforward comparisons of the results, we have set a consistent stopping criterion, learning rate, and maximum iteration count for the entire assignment.

4 Block Coordinate Gradient Descent - BCGD

Block Coordinate Gradient Descent is a variant of classic Gradient Descent where, instead of working with the full gradient, we limit our updates to a selection of blocks of coordinates.

Suppose that we have to optimize the problem (14). Partitioning the variables in b blocks such that the sum of the cardinalities of the blocks is equal to n, the general algorithm scheme is the following:

```
1. Choose a point $x_1 \in \mathbb{R}^n$
2. For $h = 1, \dots$
3.     If $x_h$ satisfies some specific condition, then stop
4.     Set $y_0 = x_h$, pick blocks $S \subseteq \{1, \dots, b\}$ and set $l = |S|$
5.     For $j = 1, \dots, l$ select $j_i \in S$ and set                      (15)
           $y_i = y_{i-1} - \alpha_i U_{j_i} \nabla_{j_i} g(y_{i-1})$
       with $\alpha_i > 0$ calculated using a suitable line search
6.     Set $x_{h+1} = y_l$
7. End for
```

where $\{1,\dots,b\}$ is our choice of blocks of coordinates, S is a subset of these, U_{j_i} is the matrix created out of the column vectors of the identity matrix with indexes in j_i and $\nabla_{j_i} g(x) = U_{j_i}^T \nabla g(x)$ is the j_i-th vector of partial derivatives. S can be selected according to different rules, for example in this assignment we will be employing random sampling and Gauss-Southwell rules. In both of these cases we have $|S| = 1$.

Applying this procedure to our problem (1), we decided to pick as blocks the column vectors of our matrix X, so $x_{j'c}$ for $j' \in \{1,\dots,d\}$. In other words, we selected as a block the features for a fixed class. In this way we have 50 gradient blocks of dimension 3000×1, and each of them is, on average, updated multiple times during our training process. With such a choice of blocks, at each iteration h our parameters' updates focus on the columns of the gradient matrix G indexed with elements of S. In order to expedite access, we define the function $partial\_gradient(X, A, labels, c)$.

```
    def partial_gradient(X,A,labels,c):
        return (-1 * A.T) @ (I[:,c] - (np.exp(A @ X[:, c]) / ((np.
                exp(A @ X) @ np.ones((k,1))).flatten())))
```

At this stage, we proceed to integrate BCGD Randomized and BCGD Gauss-Southwell into our script. As in the case of classical Gradient Descent, in our implementation we want to keep track of x_h, of the norm of the gradient and of the time per each iteration.

4.1 BCGD Randomized

This version of BCGD algorithm (15) randomly chooses one block $i \in \{1,\dots,b\}$ to update per each iteration h.

We generate the list of random classes selections beforehand, $random\_choices$, for better efficiency, and we indicate as $curr\_c$ the class choice at current iteration. Moreover, we recall that $grad = full\_gradient(X, A, labels)$. With this notation, the updating rule of BCGD Randomized follows the steps

```
    curr_grad = grad[:, curr_c]
    grad[:, curr_c] = partial_gradient(X,A,labels,curr_c)
    X = X - LR * grad
```

4.2 BCGD Gauss-Southwell

Compared to the randomized version, per each iteration h BCGD Gauss-Southwell specifically selects the block i_h such that

$$i_h = \text{argmax}_{j \in \{1,\dots b\}} ||\nabla_j g(x_h)||_2 \qquad (16)$$

This method is designed to make the most significant improvement in reducing the function's value at each step. The block with the largest gradient norm is chosen because moving along this direction is likely to result in the largest decrease of the cost function g. We put this selection strategy into practice:

```
    # we recalculate the gradient of the block gradient we updated
    grad[:, max_norm_class_index] = partial_gradient(X,A,labels,c =
     max_norm_class_index)
    # we recalculate the norm of the block gradient we updated
    norms[max_norm_class_index] = np.linalg.norm(grad[:,
    max_norm_class_index])
    # we get the index of the column with the largest norm in the
    current iteration
    max_norm_class_index = np.argmax(norms)
    # we select the column with the largest norm
    max_norm_partial_grad = grad[:, max_norm_class_index]
    # and finally, we update
    X = X - LR * grad
```

5 Results

Finally, we shall try our algorithms on some datasets and comment on their performances.

5.1 Generated dataset

Let us consider the dataset we generated in section 1 and apply the three algorithms previously described. Each method minimizes the cost function and approximates a global minimum successfully, as illustrated in Figure 1. An essential observation of the graph shows us that BCGD algorithms exhibit a consistent pattern in terms of performance and graphical representation, characterized by a similar descent trajectory. In contrast, standard GD requires approximately double the time to optimize the cost function, as evidenced by a less steep descent curve. Especially, we notice that BCGD Gauss-Southwell outperforms the other algorithms in terms of optimization speed and efficiency.

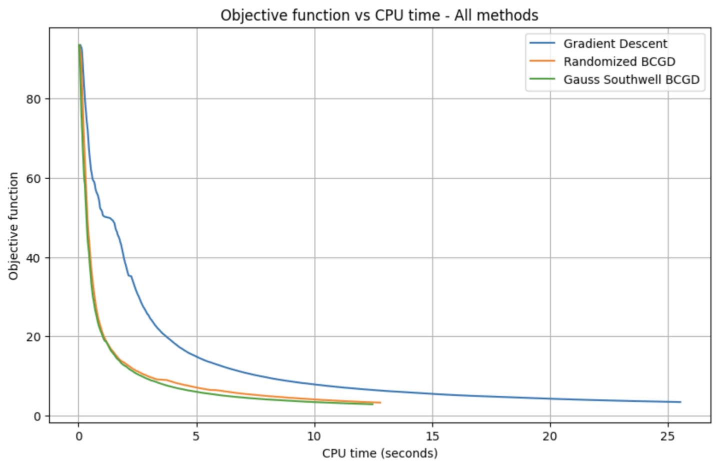

Figure 1: GD and BCGD performances on the generated dataset.

5.2 Real dataset

At this moment, we move forward to put our algorithms to the test on a real-world dataset, [1], which consists of 5620 handwritten digits. Each sample represents a digit in the range of 0 to 9, and it's described by 64 features, each taking on integer values that span from 0 to 16. The reasoning for choosing this dataset is based on several factors. Firstly, it has a substantial number of samples, which allows for a robust train/test division. Secondly, the features are uniformly structured, leading to a more efficient training process. Lastly, the dataset is divided in 10 different classes, with which we can show a reliable representation of the performance of our multi-class logistic regression algorithms. To ensure a more resilient training process, we standardize the features by removing the mean and scaling to unit variance. In the end, we use 80% of our samples for training and 20% to check how accurate the labels were predicted. More information on this dataset can be found here.

The observed behavior aligns with the one obtained on the generated dataset, as visible in Figure 2. Yet again, GD algorithm requires approximately twice the time to optimize the cost function compared to BCGD algorithms, which consistently outperform GD. However, we are now able to see the primacy of BCGD Gauss-Southwell algorithm more clearly. Additionally, GD exhibits a steeper descent, indicating a more rapid optimization process.

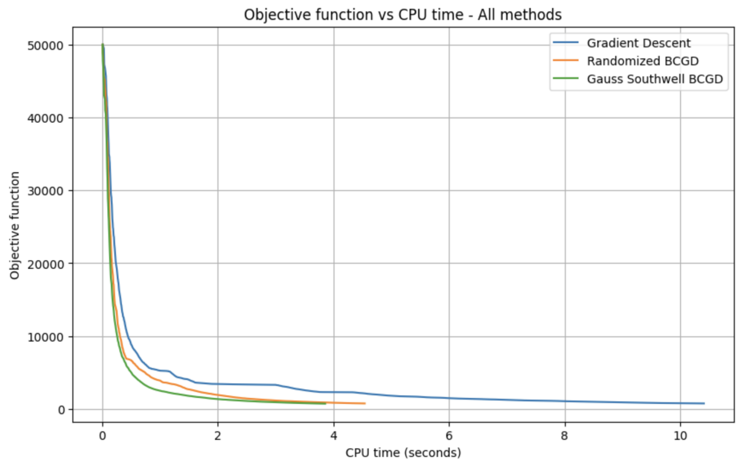

Figure 2: GD and BCGD performances on the real dataset.

5.2.1 Accuracy

Moreover, we compute the accuracy on the test split of the real dataset. We derive our predictions by estimating the probabilities for each sample to be in one of the k classes and then we pick the index with the largest probability. Given that these probabilities are provided by equation (2), we can employ the softmax function to determine them, obtaining

```
    matrix = (data @ feature_matrix) + E
    softmax_results = softmax(matrix)
    labels_predicted = np.argmax(softmax_results , axis=1)
```

Once we know the predicted labels, we can define the accuracy as presented in the ensuing code lines

```
    def accuracy(labels_predicted , labels):
        same_values = (labels_predicted == labels.flatten())
        num_same_values = np.sum(same_values)
        accuracy = (num_same_values / labels.shape[0]) * 100
        return accuracy
```

The accuracy on the test set for the real data, with respect to each optimization method, is:

Classic GD: 92.88256227758008%

BCGD Randomized: 92.97153024911033%

BCGD Gauss-Southwell: 93.14946619217082%

In light of the fact that all algorithms are able to approximate the minimum effectively, each technique results in a high accuracy score. As expected, BCGD Gauss-Southwell performs best, followed by BCGD randomized and GD.

References

[1] Alpaydin E. and Kaynak C., *Optical Recognition of Handwritten Digits*, UCI Machine Learning Repository, 1998.