UNIVERSITÀ DEL SALENTO

Facoltà di Ingegneria

Corso di Laurea Magistrale in Computer Engineering

Advanced Control Techniques Project

Multinomial Logistic Regression for a Supervised Learning problem

Professor: **Giuseppe Notarstefano**

Students: **Alfarano Gianluca,**
Basile Davide,
Capoccia Leonardo,
Larini Ludovico

Academic year 2017/2018

Abstract

In this report, it will be shown how to solve a Multinomial Logistic Regression problem (also known as *Softmax Regression*) using a distributed method. The sub-gradient method has been used to distribute calculations among a configurable number of agents. A portion of the dataset is given to each agent and used to minimize a cost function related to the portion of the dataset in its possession. A Python3 program has been written supported by several libraries. The most important library is MPI4py which provides implementation of the MPI specifics. MPI is a communication protocol between nodes that executes the same program in parallel. We compared the results obtained through this program with MATLAB script which executes all operation with a centralized architecture (meaning that all operations are executed by a single agent).
Several graphs are plotted in order to see the graph of communicating agents, how the consensus is reached and how the cost diminishes to a minimum.

Contents

List of Figures

Introduction

Motivations

In the past there was a single *Mainframe* that executed all digital computations. Years after, with the creation of the Personal Computer, more people could execute the same operations in private. Today's *Microcontrollers* allow to make smart all kind of devices. More algorithms have been created to connect these devices to distribute computations and tasks.
In general, this approach is useful because it allows cooperation between agents, to reach a common goal. In this case, the ability to split the workload in to several agents guarantees faster execution times. The Multinomial Logistic Regression generalizes the logistic regression for a multiclass problems, with more than two possible discrete outcomes, where each observation has several features.

Contributions

This work implements a scenario in which there are some agents that estimate a cost function using their own information and those of other agents; they use a *Distributed Sub-gradient Method* to update their own estimates and, in particular, they resolve a *Multinomial Logistic Regression* problem.

The first step was to create a MATLAB script that solves a quadratic optimization problem using a gradient method. In this script it has been *simulated* a distributed approach in order to test the effectiveness of the method saw in Distributed Subgradient Methods for Multi-Agent Optimization [1]. MATLAB was only used in the prototype phase, because it doesn't allow real parallel computations. After the confirmation that the approach was working, it started the implementation of the algorithm in Python3 using the Message Passing Interface.
The quadratic form used was

$$x^T Q x + r^T x \tag{I}$$

.
This form was used only to test the convergence of the method.

After having established that, considering that the Supervised Learning problem known as Multinomial Logistic Regression gives as output a probability value for every considered class, it was necessary to implement the softmax function.

$$g_i\left(\theta\right) := \sum_{k=1}^{K} 1\{y^{(i)} = e_k\} \log\left(\frac{exp(\theta^{(k)\top} x^{(i)})}{\sum_{k=1}^{K} exp(\theta^{(j)\top} x)}\right) \tag{II}$$

[3].
The formula saw in 1.7 was minimized in a centralized setup using MATLAB, and the result was compared to the one of the Python distributed program. It was seen that the algorithm was converging, several tests were executed to find the optimal values for the step-size.
The present work is divided into two chapters. In Chapter 1, it is introduced the theory behind the problem, it is shown and commented the code. In Chapter 2 there are the results of simulations with some considerations.

Chapter 1

Problem and its implementation

1.1 Theory of the problem

1.1.1 Distributed Subgradient Methods for Multi-Agent Optimization

In this problem there are m agents that cooperatively minimize a common additive cost. The general optimization problem is:

$$min \quad \sum_{i=1}^{m} f_i\left(x\right) \qquad subject\ to \quad x \in \mathbb{R}^n, \tag{1.1}$$

where $f_i : \mathbb{R}^n \longrightarrow \mathbb{R}$ is the cost function of agent i, known only by the agent i, and $x \in \mathbb{R}^{\ltimes}$ is the decision vector. It is assumed that:

- the cost function is convex;
- the agents are distributed over a time-varying topology;
- the graph (V, E_∞) is connected, where E_∞ is the set of edges (j, i) representing agent pairs communicating directly an indefinite number of times;
- there is no communication delay.

Every agent i generates and maintains an estimate of the optimal decision vector based on information concerning its own cost function and exchanges this estimate with its neighbours at discrete times $t_0, t_1, t_2, ...$ A weights matrix is associated to the communication graph, and its elements are defined as follows:

$$\begin{cases} > 0, & \text{if } (i, j) \in E(k) \\ 0, & \text{otherwise} \end{cases}$$

Each agent j that receives an information by a neighbour agent i weights this information by multiplying it by $a_i^j(k) \in \mathbb{R}$. Then the estimate of the optimal solution is updated according to:

$$x^i(k+1) = \sum_{j=1}^{m} a_j^i(k)\, x^i(k) - a^i(k)\, d_i(k) \tag{1.2}$$

where $\alpha^i(k) > 0$ is the step-size used by agent i and the vector $d_i(k)$ is a sub-gradient of objective function $f_i(x)$ calculated at $x = x^i(k)$. [1]

1.1.2 Multinomial Logistic Regression

The problem to be solved is a Supervised Learning problem called Multinomial Logistic Regression, also known as Softmax Regression, and it generalizes the more common Logistic Regression. The difference between them is that in the former there are several classes to be considered, in the latter, there are only two classes (or equivalently a binary class).
The problem to be solved is to find a set of coefficients based on a given dataset to predict the belonging class for an unseen set of features, while minimizing a cost function. The dataset is composed of N labelled examples $\{(x^{(1)}, y^{(1)}), ..., (x^{(N)}, y^{(N)})\}$. Each $x^{(i)} \in R^{d_x}$ for $i = 1, ..., N$ is composed of some features which represent the value upon which is based the estimation of the belonging class, while $y^{(i)} \in R^{d_y}$ is the belonging class for the *i-th* example, and can be a values in $\{1, ..., K\}$.

Given a single training example $(x^{(i)}, y^{(i)})$, the definition of the cost function is:

$$f_i(\omega) := \left|\left| h_\omega\left(x^{(i)}\right) - y^{(i)} \right|\right|^2 \tag{1.3}$$

where the $\omega \in R^{d_x}$ are the weights of the hypothesis function h_ω. The overall cost function can be defined as:

$$f(\omega) := \sum_{i=1}^{N} f_i(\omega) \tag{1.4}$$

The problem is solved by finding the solution of the following optimization problem:

$$\omega^* := \arg\min_{\omega} f(\omega) \tag{1.5}$$

In the Multinomial Logistic Regression, a common choice for the hypothesis function is the following:

$$h_\theta = \frac{1}{\sum_{j=1}^{K} exp\left(\theta^{(j)\top} x\right)} \begin{bmatrix} exp\left(\theta^{(1)\top} x\right) \\ \vdots \\ exp\left(\theta^{(K)\top} x\right) \end{bmatrix} \tag{1.6}$$

where the weights $\omega = \theta = (\theta^{(1)}, ..., \theta^{(K)}) \in R^{d_x}$.
Using this function, the solution of the problem is given by finding:

$$\theta^* = \arg\min_{\theta} -\sum_{i=1}^{N} g_i(\theta) \tag{1.7}$$

with

$$g_i(\theta) := \sum_{k=1}^{K} 1\{y^{(i)} = e_k\} \log\left(\frac{exp(\theta^{(k)\top} x^{(i)})}{\sum_{k=1}^{K} exp(\theta^{(j)\top} x)}\right) \tag{1.8}$$

where $\mathbf{1\{\cdot\}}$ being the *indicator function.* [2]

1.1.3 Pseudocode

Algorithm 1

```
1: Stop Rules:
2: $||\theta_{k+1} - \theta_k|| \leq \varepsilon$    $\varepsilon$ fixed
3: Number of maximum iterations reached
4: Start:
5: Fix initial conditions for each node $\theta_i(0) = [0 \quad ... \quad 0]^T$
6: Define the Adjacency Matrix, Weights Matrix, $\alpha^i = \alpha$ constant for each
   iteration
7: while No stop rule is true, each node $i$ does: do
8:     calculate $\nabla f_i$
9:     for each neighbour j do
10:        $\theta_i(k+1) = \theta_i(k+1) + a^i_j(k)\theta^j(k)$
11:    end for
12:    $\theta_i(k+1) = \theta_i(k+1) - \alpha\nabla f_i$
13: end while
14: Result:
15: Each node $i$ should converge to $\theta^*$
16: The minimum of function is $\sum_{i=1}^{m} f_i(x^*)$
```

1.2 Code Implementation

Here is defined and explained the Python implementation that solves the problem. The solution has been implemented using: *numpy* library which computes all vector and matrix operations such as transposition, product, division, summation; *networkx* which creates and manages adjacency matrices for all agent; *matplotlib* that plots the results with data from the test and training phases. At first, the code checks if all the parameters are valid, then it starts to create the environment, the adjacency matrix and it initializes the local variables where the data will be saved. The weights depend on how many in-neighbours each agent has. The number of in-neighbours for each agent can be passed as a paramether at launch, by specifying it with -k .

The adjacency matrix is created by the *createAdjM* function. It makes a communication graph using world_d as number of mpi agents, n_edges (setted by default as 1), phi that is simply a phase inserted into graph which just shift the connection, by default is 0 (TODO: la parte di phi non mi convince). The first for cycle sets the number of agents that the system has, then edges are computed as follows:

- The sum j + k + phi + 1 means the position of edge in the graph where item j is the current row, k is the count of agent that the agent j will send a message, phi is the phase and just shift the agent that j will send a message and the "+1"
- The instruction $if\ e >= world\_d$ checks if the sum is over the number of agents and if is true just subtracts this one
- Instead $if\ e == j$ removes the hypothetical unnecessary self loop.

Loss softmax function,*loss_softmax* gets as parameter the current state(*all_theta*), the cardinality of iris set (*category_count*), the agent dataset (*personal_dataset*) and a *CONSTANT_TO_SUBTRACT*, a constant to prevent overflow.
The algorithm wants to calculate the $\sum_{i=0}^{number\_of\_agents} f_i(x)$ and this function calculates the $f(x)$ of the agent i . This is done as follows: for every row in the personal dataset, the algorithm computes the denominator that is the sum over all exponential. Then it computes, category by category, row by row, the logarithm of the ratio between the sum previously calculated and the exponential of the row. After that, there's the summation of all calculated elements.

```
def loss_softmax(all_theta , category_count , personal_dataset ,
    CONSTANT_TO_SUBTRACT):
    the_sum = 0

    for index in range(0, len(personal_dataset)):
        denominator = 0

        for theta in all_theta:
            denominator = denominator + np.exp(np.dot(theta ,
                personal_dataset[index][0:4]) - CONSTANT_TO_SUBTRACT)

        for category in range(0, category_count):

            if category == personal_dataset[index][4]:
                _exp = np.exp(np.dot(all_theta[category],
                    personal_dataset[index][:4]) -
                    CONSTANT_TO_SUBTRACT)
                _log = np.log(np.divide(_exp, denominator))
                the_sum = the_sum - _log

    return the_sum
```

In the function *gradient_softmax* there is implemented the gradient of softmax equation. It is calculated similarly as the function mentioned before: it calculates at first the sum of all exponential theta and then it sums for each coefficient. It subtracts from one that coefficient and finally subtract the dataset normalized with that to the respective theta. TODO l'ultima frase non la capisco.

```
def gradient_softmax(all_theta, category_count):

    thetas = np.zeros(dimensions)

    for index in range(0, len(personal_dataset)):
        denominator = 0

        for theta in all_theta:
            denominator = denominator + np.exp(np.dot(theta,
                personal_dataset[index][:4]) - CONSTANT_TO_SUBTRACT)

        for category in range(0, category_count):
            coeff = 0

            if category == personal_dataset[index][4]:
                coeff = 1

            _exp = np.exp(np.dot(all_theta[category], personal_dataset
                [index][:4]) - CONSTANT_TO_SUBTRACT)
            coeff = coeff - np.divide(_exp, denominator)
            thetas[category] = thetas[category] - ((1/len(
                personal_dataset)) * np.multiply(personal_dataset[
                index][:4], coeff))

    return thetas
```

1.2.1 working.py

In this file it is implemented the main algorithm where the loss function is minimized and the consensus is reached.

The complete iris training set is loaded, and some environment variables are setted like the number of agents. Then the dataset is splitted in equal parts to all agents and each one gets its own piece and it prints how many row it has.

Every agent creates the same communication directed graph with number of agents and number of in connection. Then state and loss variables are created, setted to 0 of *MAX_ITERATIONS* size. In order to get weight the messages, all in-neighbours are found and the variable *weight* is setted as the average. If the function name inserted is ”quadratic” Q and r variables are created randomly. *epsilon_reached* and *buff* are epsilon checker variable that say to agent when they have to exit from the loop (if an accuracy better then epsilon is reached).

This *for* cycle calculates the consensus. For every *iters*, is calculated the new ”diminishing” alpha; message(s) are sent and received, then local variables are weighted. An *if* clause allows to choose how to solve the consensus, with one of the implemented functions. After that, the calculated gradient is multiplied by alpha and then the new state is calculated. Then *lossfunction* is called and if $\|XX[tt] - XX[tt-1]\| \leq epsilon$ then buff is true and rank 0 check if all agent has reached the epsilon condition; if true it sends a broadcast message with a *true* value to say to all that the cycle is done.

```
for tt in range(1, MAX_ITERATIONS - 1):

    if alpha_type == "diminishing":
        alpha = psi_coefficient * (1 / tt) ** alpha_coefficient
    else:
        alpha = alpha_coefficient

    # Update with my previous state
    u_i = np.multiply(XX[tt - 1], weight)

    # Send the state to neighbors
    for node in adj.successors(rank):
        world.send(XX[tt - 1], dest=node)

    # Update with state of all nodes before me
    for node in adj.predecessors(rank):
        u_i = u_i + world.recv(source=node) * weight

    # Go in the opposite direction with respect to the gradient
    gradient = 0

    if function_name == "softmax":
        gradient = func.gradient_softmax(XX[tt - 1], category_n,
            dimensions, personal_dataset, CONSTANT_TO_SUBTRACT)

    elif function_name == "quadratic":
        gradient = func.gradient_quadratic(XX[tt - 1], category_n,
            dimensions, personal_dataset, Q, r)

    elif function_name == "exponential":
        gradient = func.gradient_exponential(XX[tt - 1], category_n,
            dimensions, personal_dataset, CONSTANT_TO_SUBTRACT)

     #print(gradient)

    grad = np.multiply(alpha, gradient)

    for i in range(0, dimensions[0]):
        u_i[i] = np.subtract(u_i[i], grad[i])

    # Store  my new state
    XX[tt] = u_i

```

```python
    if function_name == "softmax":
        losses[tt] = func.loss_softmax(XX[tt], category_n,
            personal_dataset, CONSTANT_TO_SUBTRACT)

    elif function_name == "quadratic":
        losses[tt] = func.loss_quadratic(XX[tt], category_n,
            dimensions, personal_dataset, Q, r)

    elif function_name == "exponential":
        losses[tt] = func.loss_exponential(XX[tt - 1], category_n,
            dimensions, personal_dataset, CONSTANT_TO_SUBTRACT)

    # Checking epsilon reached condition
    if np.linalg.norm(np.subtract(XX[tt], XX[tt - 1])) < epsilon:
        buff = True

    # Rank 0 get all epsilon and check if all reached it
    buffer = world.gather(buff, root=0)

    # If true it set epsilon reached
    if rank == 0:
        if False not in buffer:
            epsilon_reached = True

    # Send epsilon reached to all agents
    epsilon_reached = world.bcast(epsilon_reached, root=0)

    # Check if all agent have reached epsilon condition and then exit
        from loop
    if epsilon_reached:
        if rank == 0:
            print("Exiting at iteration ", tt, "/", MAX_ITERATIONS, "
                Condition on epsilon reached")
            sys.stdout.flush()

        break

    if tt in range(0, MAX_ITERATIONS, 100):
        if rank == 0:
            print("Iteration ", tt, "/", MAX_ITERATIONS)
            sys.stdout.flush()
    ITERATION_DONE = tt
```

When consensus is reached, the theta values are printed to user, then all data are sent to rank 0 for centralized calculations and plot result.

Chapter 2

Results of simulations

To verify the correctness of the software, a couple of simulations were run, divided into two parts:

- Minimization of the Softmax function to prove the convergence to the optimal value
- Application of the minimization algorithm to find the best coefficients to do the best predictions on the test dataset

2.1 Minimization of the Softmax function

The simulation has been carried out using a random directed graph made up of 17 agents, as shown in figure 2.1.

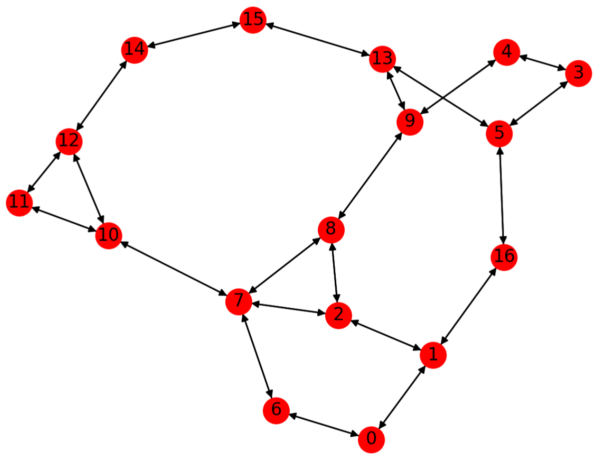


Figure 2.1: Communication graph used for the simulation.

For completing this simulation, a diminishing step-size has been used, with the following formulation:

$$\alpha^k = 0.01\left(\frac{1}{k}\right)^{0.4} \tag{2.1}$$

with k indicating the current iteration.
The problem to be solved is the one formulated in 1.7. Let's suppose to indicate the whole dataset as S. Each agent i can access to a limited set s_i of examples contained in S. To each agent is given a set s_i, chosen as:

$$S = \bigcup_{i=1}^{17} s_i \tag{2.2}$$

and

$$\bigcap_{i=1}^{17} s_i = \emptyset \tag{2.3}$$

In the simulation, the training set S is composed by 120 examples, split into sets s_i of 6 or 7 examples, chosen in order to equation 2.2 and 2.3 hold.
The iterations run are 15000, sufficient enough to see the function approaching the convergence, but not sufficient enough to reach the minimum value of the function, calculated with MATLAB to be 8.945. The figure 2.2 shows in a logarithmic scale how the algorithm tends to minimize the cost value at each iteration.

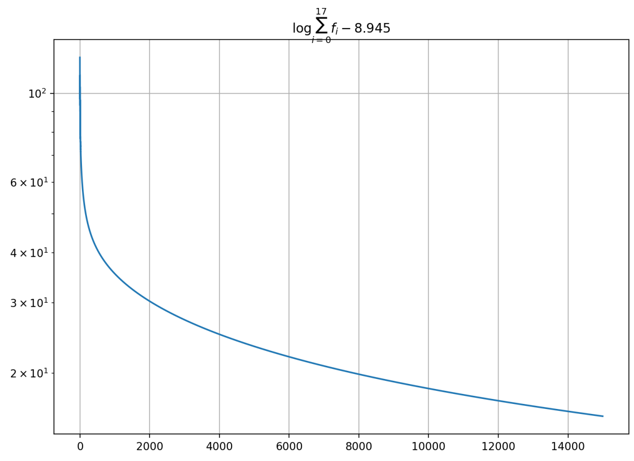


Figure 2.2: Value of the normalized cost function in a logarithmic scale over all the iterations.

The value of the cost function is shown in figure 2.3 in a linear scale too, so it can be seen how the algorithm approaches the minimum value Θ^* in order to have the minimum of the problem 1.7.

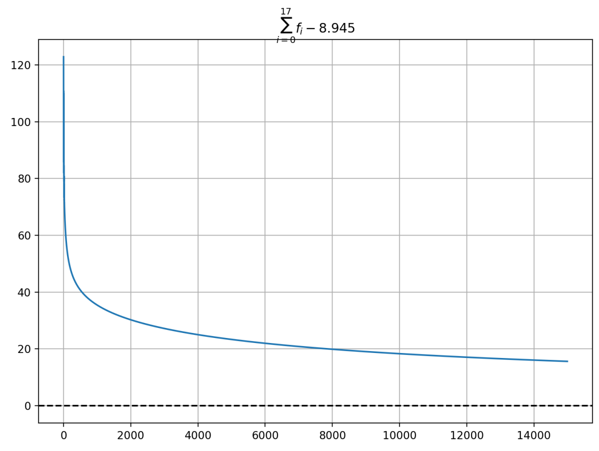
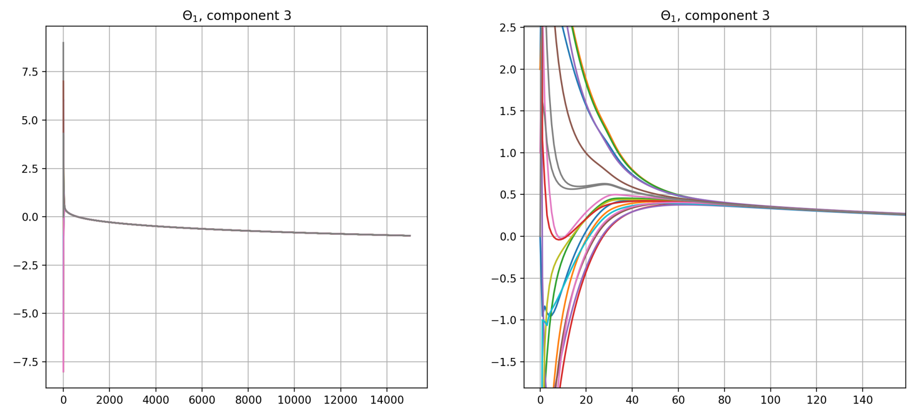


Figure 2.3: Value of the normalized cost function in a linear scale.

As can be seen in figure 2.4, all the agents try to reach the consensus for each component of the local minimum of the problem.

To better see the consensus, in this simulation the initial condition have been chosen to be random integer number between -10 and $+10$. With this choice is more visible the trend of the agents that try to reach consensus.



(a) Overall trend over all the iterations (b) Detail for the first 140 iterations

Figure 2.4: Consensus on the third component of Θ_1

2.2 Dataset, graph description and minimization function

This dataset is composed of 150 instances, 120 of them used for training while the remaining 30 used for testing purpose. The instances contain 3 classes, each representing a type of Iris flower. Every instance has 4 features: sepal and petal length, sepal and petal width, expressed in *cm*. We tried different types of graphs, all of them chosen to be strongly connected. The results discussed in this chapter, if not differently noted, refer to cyclic graphs with a variable number of nodes. The program can minimize all kinds of loss functions. As shown in Chapter 1.2, the quadratic and exponential functions can also be optimized, even if they don't guarantee useful results to apply to our problem. The minimization function used in this chapter is the one described in Chapter 1.1.2.

2.3 Performance

There are some key factors that influence *the computational time*, *the numbers of iterations necessary* and *the accuracy of the results*. They are, but not limited to, number of nodes, type of graph, step-size type and value, etc.

2.3.1 Number of nodes

The Python program, thanks to the MPI platform, is capable of running on an arbitrary number of nodes. It was tested on as little as 2 nodes to as many as 60 nodes, which means that every node was processing the data of 2 instances (120 instances divided into 60 nodes). The best performances are obtained when the number of nodes corresponds to the number of physical cores of the machine where it runs. When the number of nodes exceeds greatly the number of physical cores, the resources are oversubscribed. In this case, the performances degrade notably as the nodes compete for cache and memory and the processors' schedulers are put in a difficult situation. On a 4-core test machine, a computation with 5000 iterations and 30 nodes is done in 5 minutes. The same machine can do the same number of iterations, but with 60 nodes, in 15 minutes. Therefore, to show the scalability of the software, the following tests will be shown on a 4 nodes set-up and a 17 nodes setup (the same of figure 2.1) .

2.3.2 Epsilon

This is a small constant used as stop condition. If the norm of the result of the current calculation differs less than ϵ (chosen arbitrary small) from the previous, the algorithm is stopped. This is used as convergence stop condition. In fact, given the nature of the method used to minimize the cost function, the converge to the optimal value can be reached in a huge amount of time. To prevent this from happening, it has been used this condition.

2.3.3 Learning rate

The step-size plays a big role in the speed of convergence of the algorithm. There are 3 kinds of step-size:

1. fixed step-size: it has always the same value, which means $\alpha^{k+1} = \alpha^k$
2. diminishing step-size: chosen such that $\alpha^{k+1} < \alpha^k$
3. adaptive step-sizes: chosen with respect to Armijo or (limited) minimization rule.

For reasons not discussed in this paper, it's not possible to use the last type of rules in a distributed problem, so the first two types have been used in these simulations.

2.3.4 Initial condition

The agents start from an initial condition x_0, which represents a guess of the optimal value. The program was tested with random initial conditions for all agents and the consensus was still reached. To guarantee the repeatability of the experiments, the initial condition in the tests is always set to a matrix full of ones.

2.4 Four Nodes Setup

2.4.1 Fixed step-size

We will first deal with a simple fixed step-size such as $\alpha^k = \alpha$, with $\epsilon = 10^{-3}$ equal to:

α	**Iteration required**	**Execution time in s**	**Wrong guesses**
0.5	overflow occurred	-	-
0.1	>10000	>15	2
0.05	>10000	>15	1
0.01	1968	2.8	1
0.005	1558	2.3	1
0.001	2114	3.1	0
0.0005	1619	2.3	2
0.0001	527	0.7	14

These results show a general truth about the step-size. If it is too little, the learning process proceeds in a very slow way and it requires a huge amount of iterations. If the learning rate is too high, the gradient descent will most probably overshoot the minimum and it will not converge.
Through trial and error, the step-size 0.001 was chosen. It allows to reach good performances and the best accuracy: only 3.1 seconds are required to obtain 0 errors over the test dataset.

2.4.2 Diminishing step-size

The diminishing step-size implemented is chosen to be:

$$\alpha = \psi \left(\frac{1}{k}\right)^{\gamma} \tag{2.4}$$

Again, several tests were run by tweaking the multiplying constant and the exponent.

ψ	γ	**Iteration required**	**Execution time in s**	**Wrong guesses**
1	0.01	overflow	-	-
1	0.1	overflow	-	-
0.1	0.01	>10000	>15	1
0.1	0.5	349	0.5	0
0.1	0.1	>10000	>16	1
0.01	0.01	1852	2.7	1
0.01	0.1	1259	1.8	1

The best result is obtained in only 349 iterations, with $\psi = 0.1$ and $\gamma = 0.5$. This is done in roughly half of a second, obtaining no error. The accuracy is worst then the previous result, but this is obtained in a fraction of the time needed to obtain 0 error with a fixed step-size. A difference of 2 or 3 seconds may not seem important using the Iris Dataset, with a small amount of calculations. In a set-up where a bigger number of calculations and a bigger dataset are involved, one may appreciate the advantages of this faster approach.

2.5 Seventeen Nodes Setup

The following results are the ones for the set-up shown in fig. 2.1.
The computation times are greatly larger than the ones shown earlier. This is because the tests involving 17 agents are carried out on a quad-core machine, which means that MPI oversubscribe the resources and the CPU scheduler are put at hard work.
The optimal step-size is different from the ideal step-size of the 4 nodes

set-up. Only through several tests and experience is possible to find a good step-size to reduce the errors in the best way.

ψ	γ	**Iteration required**	**Execution time in s**	**Wrong guesses**
0.1	0.1	>10000	>440	2
0.5	0.1	>1000	>440	2
0.9	0.1	>10000	>440	2
0.01	0.1	1894	83	2
0.1	0.5	698	30	2
0.5	0.5	1017	44	1
0.9	0.5	2409	107	0
0.01	0.5	318	13	14
0.1	0.01	>10000	>440	1
0.5	0.01	>10000	>440	2
0.9	0.01	>10000	>440	2
0.01	0.01	3717	164	2

TODO: AGGIUNGERE GRAFICI

Conclusions

In this work it has been resolved a Multinomial Logistic Regression problem using MPI in Python. Each agent use a Distributed Sub-gradient method to update its own estimate of the optimal solution. At the end of the iterations, all the agents converge to the same solution.

Bibliography

[1] A. Nedic and A. Ozdaglar. Distributed subgradient methods for multi-agent optimization. *IEEE Transactions on Automatic Control*, 2009.

[2] Stanford. Softmax regression. Available at `http://ufldl.stanford.edu/tutorial/supervised/SoftmaxRegression/`.

[3] Wikipedia. Multinomial logistic regression. Available at `https://en.wikipedia.org/wiki/Multinomial_logistic_regression`.