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 (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.

Contents

List of Figures

Introduction

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 an infinity of devices. More algorithms have been created to connect these devices to distribute.
This work implements a scenario in which there are some agents that estimate a cost function using their own information and those of the other agents; they use a *Distributed Sub-gradient Method* to update their own estimate and, in particular, they resolve a *Multinomial Logistic Regression*. After some test in *MATLAB*, it is used *MPI* implemented with *Python*.
The present work is divided into two chapters. In Chapter 1, it is introduced the theory behind the problem and it is visualized and commented the implementation code. In Chapter 2 there are the results of simulations with some considerations.

Chapter 1

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:

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

where $f_i : \mathbb{R}^n \longrightarrow \mathbb{R}$ is the cost function of agent i, known only by this agent, and $x \in \mathbb{R}$ is a 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 isn't communication delay.

Every agent i generates and maintains estimates of the optimal decision vector based on information concerning its own cost function and exchanges this estimate with its directly neighbors at discrete times $t_0, t_1, t_2, ...$ Moreover, each agent i has a vector of weights $a^i(k) \in \mathbb{R}^m$ at any time t_k; the scalar $a_i^j(k)$ is zero if the agent i doesn't directly communicate with j, else it is the weight assigned from the agent i to the information x^j obtained from

j during the time interval (t_k, t_{k+1}). The estimates are updated according to the update rule:

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

where $\alpha^i(k) > 0$ is the stepsize used by agent i and the vector $d_i(k)$ is a subgradient of agent i objective function $f_i(x)$ 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 we base 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:

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

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

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

We solve the problem by finding the solution of the following optimization problem:

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

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

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

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 neighbor j do
10:         $\theta_i(k+1) = \theta_i(k+1) + a_j^i(k)\theta^j(k)$
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 we are defining and explaining what has been implemented in Python to solve the problem. The solution has been implemented using *numpy* library which computes all vector and matrix operation such as transposition, product, division, summation. Another library used is *networkx* which creates and manages connection agent by agent.

1.2.1 create_matrix.py

The code generates random integers to create a random matrix to compute quadratic functions; *warning* warns about reading invalid values; *networkx* library is used to create the directed graph with its edges; *numpy* library computes some matrix operations.

```
from random import randint
from warnings import warn
import networkx as nx
import numpy as np
```

The *createAdjM* function create a communication graph using world_d as number of mpi agents, n_edges setted by default as 1 (if it is not given or bigger than number of agent), phi that is simply a phase inserted into graph which just shift the connection, by default is 0. 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* $e >= world\_d$ check if the sum is over the number of agents and if is true just subtracts this one
- Instead *if* $e == j$ removes the ipotetical unnecessary self loop.

```
def createAdjM(world_d, n_edges=None, phi=None):

    if n_edges is None:
        n_edges = 1

    if n_edges > world_d or n_edges == 0:
        warn("Invalid n_edges entered... Setting to 1")
        n_edges = 1

    if phi is None:
        phi = 0

    if phi > world_d - 1:
        warn("Invalid phi entered... Setting to 0")
        phi = 0

    g = nx.DiGraph()

    for i in range(0, world_d):
        g.add_node(i)

    for j in range(0, world_d):

        for k in range(0, n_edges):

            e = j + k + phi + 1
```

```python

            if e >= world_d:
                e = e - world_d

                if e == j:
                    e = e + 1

                    if e >= world_d:
                        e = e - world_d

                g.add_edge(j, e)

            else:
                g.add_edge(j, e)

    return g
```

createR function creates a random vector, to compute the quadratic function, for each agent. This is used only for quadratic optimization.

```python
def createR(d):
    r = np.zeros((d, 1))

    for i in range(0, d):
        r[i] = randint(0, 1)

    return r
```

createQ function creates a random matrix, to compute quadratic function, for each agent. This is used only for quadratic optimization.

```python
def createQ(d):
    q = np.zeros((d, d))
    eye = np.identity(d)

    for i in range(0, d):

        for j in range(0, d):
            q[i][j] = randint(0, 2)

    a = np.add(q, np.transpose(q))
    a = np.add(a, np.dot(d, eye))

    return a
```

1.2.2 functions.py

This file has the gradient and loss functions for softmax regression, quadratic and exponential. The softmax regression is the function that computes the machine learning algorithm.

```python
import numpy as np
```

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 calculate the agent i $f(x)$. This can be 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 ratio between the sum previously calculated and the exponential of the row. After that, there's the summation of all calculated elements.

```python
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
```

gradient_softmax This is the gradient of softmax equation, that has the same concept as the function mentioned before: calculate in a first time se sum of all exponential theta and then sum for each coefficient, subtract from one that coefficient and finally subtract the dataset normalized with that to the respective theta.

```python
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
```

1.2.3 working.py

In this file is implemented the main algorithm where the consensus and the minimization is calculated.

To implement the consensus and minimization these libraries are needed:

- *mpi4py* is the mpi library for python which generates parallel simulated agent that compute all calculus.
- *numpy* that implements matrix/vector computation
- *matplotlib* is the one that plots the results
- *functions and create_matrix* that are the previous file exposed
- *time* that is needed to calculate the computational time of algorithm

Then the complete iris training set is loaded, and environment variables are setted, as the number of agents, the name of $agent_i$.

```
from mpi4py import MPI
import create_matrix as cm
import numpy as np
import matplotlib.pyplot as plt
import sys
import functions as func
import time


dataset = np.loadtxt('iris_training_complete.txt', delimiter=';',
    dtype=float)

""" Define world parameter, these have been got from mpi system """
world = MPI.COMM_WORLD
agents_number = world.Get_size()
rank = world.Get_rank()
```

Here it is defined the maximum number of iterations; dimensions is an array of [number of category, number of variables].

```
""" Define variables """
MAX_ITERATIONS = 10000
category_n = 3
dimensions = [category_n , 4]
```

Then the dataset is splitted equal parts to all agents and each one gets its one and print how many row it has.

```
# Assign dataset to each agent
dataset_portion = len(dataset) / agents_number
start_dataset = rank * dataset_portion
end_dataset = (rank * dataset_portion) + dataset_portion
start_dataset = int(start_dataset)
end_dataset = int(end_dataset)
personal_dataset = dataset[start_dataset:end_dataset]

print("Agent ", rank, " got ", len(personal_dataset), " rows of 
    dataset")

world.Barrier()
sys.stdout.flush()
```

Every agent creates the same communication directed graph with number of agents and number of in connection. Then state and loss variable are created, setted to 0 of *MAX_ITERATIONS* size. In order to get wighted message, all in-neighbors 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 epsilon is reached).

```
adj = cm.createAdjM(agents_number, number_of_inn_connection)

x0 = np.ones(dimensions)

XX = np.zeros([MAX_ITERATIONS, *dimensions])
losses = np.zeros(MAX_ITERATIONS)
XX[0] = x0

num_of_neighbors = 0
for in_neighbors in adj.predecessors(rank):
    num_of_neighbors = num_of_neighbors + 1
weight = 1 / (num_of_neighbors + 1)  # 1 is for self-loop

world.Barrier()

if function_name == "quadratic":
    Q = cm.createQ(dimensions[1])
    r = cm.createR(dimensions[1])

epsilon_reached = False
buff = False

ITERATION_DONE = 0

start_time = time.time()
```

This *for* cycle calculates the consensus. For every *iters*, is calculated the new "diminishing" alpha; message(s) are sent and received, then local variable are weighted. In order to solve consensus to desiderated function, the three implemented ones are inserted into an *if* clause, and the right function is called. After that, the calculated gradient is multiplied by alpha and then new state is calculated. Then *loss function* is called and if $\|XX[tt] - XX[tt-1]\| \leq epsilon$ then buff is true and rank 0 check if all agent has reched 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

    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 values are printed to user, then all data are sent to rank 0 for centralized calculations.

```
print("Parameters of node ", rank)
print(XX[ITERATION_DONE - 3])

world.Barrier()
sys.stdout.flush()

if rank != 0:
    world.send(losses, dest=0)
    world.send(XX, dest=0)
```

After each agent has sent its own data, agent 0, collect and sum all loss function datas. Then it collects all *ITERATION_DONE* state of each agent in a vector, which will be converted in logarithmic form and plotted.

```
if rank == 0:

    # Take the losses from all the other agents and sum
    # We now have the overall loss given from the cost function
    for i in range(1, agents_number):
        agent_loss = world.recv(source=i)
        losses = np.add(losses, agent_loss)

    XX_agents = np.zeros([agents_number, *[MAX_ITERATIONS, *dimensions]])
    XX_agents[0] = XX
    for i in range(1, agents_number):
        XX_agents[i] = world.recv(source=i)

    log_losses = np.zeros(len(losses))
    for index in range(0, len(losses)-1):
        log_losses[index] = np.log(losses[index] - 8.945)

    # Plot cost function logarithmic
    plt.figure()
    plt.plot(range(0, ITERATION_DONE - 3), log_losses[0:ITERATION_DONE - 3])
    plt.title("$\sum_{i=0}^" + str(agents_number) + " f_i$")
    plt.show()

    # Plot cost function
    plt.figure()
    plt.plot(range(0, ITERATION_DONE - 3), losses[0:ITERATION_DONE - 3])
    plt.title("$\sum_{i=0}^" + str(agents_number) + " f_i$")
    plt.show()
    #plt.pause(100)
```

Here iris trainig is loaded to test if with the thetas found are capable to predict the category of iris. An iteration along all iris trainig set is launched, then for all category we implement the exponential about the product from the theta founded and current row of the set. Then data is normalized and then the arg max is taken as predicted value,;if it is not the right answer, a variable *wrong_answer* is updated.

```
if rank == 0:
    to_find = np.loadtxt('iris_training.txt', delimiter=';', dtype=float)
    # to_find = normalize_dataset(to_find)
    wrong_answers = 0
    for _set in to_find:
        _tot_exp = 0
        _tmp = np.zeros(4)
        for i in range(0, category_n):
            val = np.exp(np.dot(XX[ITERATION_DONE - 2][i], _set[0:4]) - CONSTANT_TO_SUBTRACT)
            _tmp[i] = val
            _tot_exp = _tot_exp + val
        _tmp = np.divide(_tmp, _tot_exp)
        _predicted = np.argmax(_tmp)
        # print('Predicted: ', _predicted, ', real: ', _set[4])
        if _predicted != _set[4]:
            wrong_answers = wrong_answers + 1

    print("Wrong predicted values: ", wrong_answers, "/", len(to_find))

    # Show consensus
```

```python
    # for category in range(0, category_n):
    for component in range(0, 4):
        figure = plt.figure()
        for agent in range(0, agents_number):
            label = "Agent " + str(agent)
            plt.plot(range(0, ITERATION_DONE), XX_agents[agent][0:ITERATION_DONE, 0, component], label=label)
        plt.title("Component #" + str(component) + " of each $\Theta_{0}^{i}$")
        leg = plt.legend(loc='best', ncol=2, mode="expand", shadow=True, fancybox=True)
        leg.get_frame().set_alpha(0.5)
        plt.show()
    print("Iteration done: ", ITERATION_DONE, " Agent number: ", agents_number, "\nEpsilon: ", epsilon,
          " Const  Alpha: ", alpha_coefficient, " Const Psi ", psi_coefficient,
          "\nExecution time: ", time.time() - start_time, " Wrong preditions: ", wrong_answers)

    plt.pause(20)
    input("Press [enter] to continue.")
```

Chapter 2

Chapter 2 Results of simulations

The software described in the previous chapter was used to solve a Multinomial Logistic Regression problem. Specifically, we classified the data of the Iris Dataset.

2.1 Dataset, graph description and minimization function

This dataset is composed of 150 instances, 120 used for training, the rest for tests. The instances contain 3 classes, each representing a type of Iris flower. Every instance has 4 features, sepal length, sepal width, petal length, petal width expressed in *cm*. We tried different types of graphs. These graphs are all strongly connected and the weight matrices for the nodes are doubly stochastic, as per the assumptions of convergence of the algorithm described in Chapter 1. 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 used. These last 2 functions don't guarantee useful results and/or convergence. The minimization function used in this chapter is the one described in Chapter 1.1.2, softmax.

2.2 Performance

There are some key factors that influence *the computational time*, *the numbers of iterations necessary* and *the accuracy of the results.*

2.2.1 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, the following tests will be shown on a 4 nodes setup.

2.2.2 Epsilon

This is a small constant used as stop condition. If the result of the current calculation differs less than epsilon from the previous, the algorithm is stopped.

2.2.3 Learning rate

The step-size alpha plays a big role in the speed of convergence of the algorithm. There are 3 kinds of step-size. Fixed, diminishing and adaptive (Armijo). For reasons not discussed in this paper, it's not possible to use Armijo rule in a distributed problem.

2.2.4 Fixed step-size

We will first deal with a simpler fixed step-size. With an epsilon equal to $0.001(10^{-3})$:

Value of fixed step-size	**Iteration required**	**Execution time in s**	**Wrong guesses o.30**
0.5	4601	6.8	1
0.1	2109	3.24	1
0.05	2342	3.68	0
0.01	1153	2.05	2
0.005	680	1.14	9
0.001	22	0.05	22

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 and the gradient descent most probably will overshoot the minimum and it will not converge. Through trial and error, the step-size 0.001 was identified, it allows reaching good performance and accuracy. In fact, in only 2.6 seconds, we can make predictions with no errors, using our 30 instances test dataset.

2.2.5 Diminishing step-size

The diminishing step-size implemented in the code is in this form: TODO: SISTEMARE
Again, several tests were run by tweaking the multiplying constant and the exponent.

psi_coeff	**alpha_exp coefficient**	**Iteration required**	**Execution time in s**	**Wrong guesses o. 30**
1	0.01	>20000	>30	13
1	0.1	4033	6.66	1
0.1	0.01	2130	5.19	1
0.1	0.1	2028	3.07	0
0.01	0.01	1067	1.60	2
0.01	0.1	628	1	8

The best result in term of accuracy is achived with both parameters set to 0.1. The computation time is similar to the fixed-step size. *Reducing the epsilon to 0.01, allow the program to reach consensus in 604 with parameters 0.01 and 0.1 and zero error.*

TODO: VEDERE QUALE PARTE LASCIARE

TODO: GRAFICI

Conclusions

In this work it has been resolved a Multinomial Logistic Regression problem using MPI in Python. Each agent used a Distributed Sub-gradient method to update its own estimate of optimal 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.