Dauphine | PSL
UNIVERSITÉ PARIS

Optimization for Machine Learning Project IASD 2022-23

Adrien Golebiewski

Contents

Introduction

The project is split into 7 parts, according to the differents sections of the course.

It aims to implement differents algorithms seen in the optimization syllabus of the course program. This report follows the order of the questions asked and tries as best as possible to link the results obtained with the theoretical notions seen in class.

The implementations must be applied to a data set chosen by us. In my study, I was interested in the "Boston Housing" dataset. Never exploited yet by my part, it contains information collected by the U.S Census Service concerning housing in the area of Boston.This data set was originally a part of UCI Machine Learning Repository and it is accessible from the "scikit-learn" library.

The model and paradigm machine learning chosen for this study is **linear regression**.

In linear regression, we seek a linear model that explains the data based on minimizing a least-squares objective:

$$\text{minimize}_{\mathbf{x}\in\mathbb{R}^d} f(\mathbf{x}) := \frac{1}{2n}\|\mathbf{A}\mathbf{x}-\mathbf{y}\|^2 = \frac{1}{n}\left(\frac{1}{2}\sum_{i=1}^{n}(\mathbf{a}_i^T\mathbf{x}-y_i)^2\right)$$

with a feature matrix $\mathbf{A}\in\mathbb{R}^{n\times d}$ and a vector of labels $\mathbf{y}\in\mathbb{R}^n$.
This problem has the form of a finite sum :

$$f(\mathbf{x}) = \tfrac{1}{n}\sum_{i=1}^{n} f_i(\mathbf{x})$$

where

$$f_i(\mathbf{x}) = \frac{1}{2}(\mathbf{a}_i^T\mathbf{x}-y_i)^2.$$

1 Part 1 - Gradient Descent

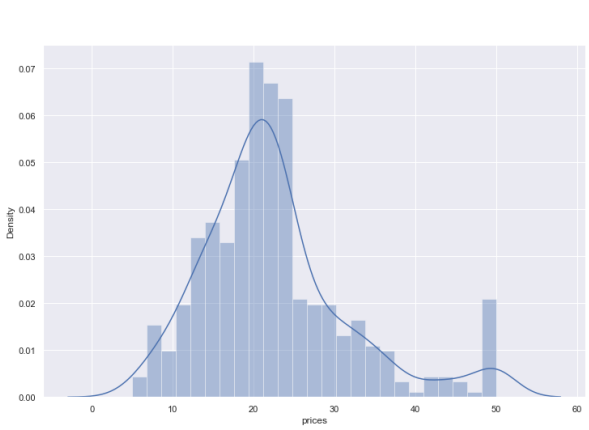
1.1 Dataset presentation

Housing prices are an important reflection of the economy, and housing price ranges are of great interest for both buyers and sellers. In this project, house prices will be predicted given explanatory variables that cover many aspects of residential houses. The goal of this project is to accurately estimate the price of the house given the features by exploiting some differents optimization algorithms.

More precisely, these features are factors such as socio-economic conditions, environmental conditions, educational facilities and some other similar factors. There are 506 observations in the data for 14 variables including the median price of house in Boston, **our target feature**. With no missing values in our data set, no feature engineering tasks were performed.

We consider our dataset $\{(\mathbf{a}_i, y_i)\}_{i=1,\dots,n}$, where $\mathbf{a}_i \in \mathbb{R}^d$ and $y_i \in \mathbb{R}$, available in the form of:
- a feature matrix $\mathbf{A} \in \mathbb{R}^{n \times d}$; with n = 506 rows and d = 13 (without the target)
- and a vector of labels $\mathbf{y} \in \mathbb{R}^n$, the price of house in Boston.

Empirically, we notice that, the average price is around 22K. There are some houses in 50's as well. We see that the values of prices are distributed normally with few outliers.



(a) Price distribution

(b) Correlation Matrix

According to the correlation matrix, every column is perfectly correlated to itself along the diagonal, as expected. We see TAX and RAD are highly correlated features. The column RM (average number of rooms per dwelling) has a correlation score of 0.7 with MEDV which is a good indication of using as predictors.

1.2 Definition of our learning problem

First, we rely on the linear regression model problem specified in the introduction. We want to study this linear regression problem and more particularly found the paramater x vector which minimize this **Mean Squared Error (MSE loss)**, our objective function :

$$\min_x \frac{1}{2n} ||Ax - Y||^2 \tag{1}$$

We go even further by adding a regularization function g to the loss, it gives the following problem:

$$\min_x \frac{1}{2n} ||Ax - Y||^2 + \lambda g(w) \tag{2}$$

More precisely, we add an l_2 regularization associated with a parameter λ to restrained the solutions to a ball. The problem can be written as per the following expression:

$$\min L(x) = \frac{1}{2n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 + \frac{\lambda}{2}||x||^2 = \frac{1}{2n}\|Ax - Y\|_2^2 + \frac{\lambda}{2}||x||^2 = \frac{1}{n}\left(\frac{1}{2}\sum_{i=1}^{n}(\mathbf{a}_i^T\mathbf{a} - y_i)^2 + \frac{\lambda}{2}\|\mathbf{x}\|^2\right)$$

The objective is to find the values of xi that *minimize* the above loss quantity, where :
- **n** the number of samples in the dataset.
- **d** the number of independant features in the dataset.
- $x = [x_0, \ldots, x_k]^T$ the weight vector of size $\mathbb{R}^{k+1}$, where x_0 is the bias and $x_1, \ldots, x_k$ are the weights.
- $Y = [y_1, y_2, \ldots, y_n]^T$, the label vector of size $\mathbb{R}^n$, where every y_i is the critical temperature of sample i.
- A of size $\mathbb{R}^{k+1}$, the transformed feature matrix with all polynomial coefficients
- $\hat{Y} = Xw = [\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_n]^\top$, the vector of predictions of size $\mathbb{R}^n$

In addition, we can define this information, taken up in the continuation of the report:
- $\nabla f_i(\mathbf{w}) = (\mathbf{x}_i^T\mathbf{w} - y_i)\mathbf{x}_i + \lambda\mathbf{w}$ for every i;
- $\nabla f(\mathbf{w}) = \frac{1}{n}\mathbf{X}^T(\mathbf{X}\mathbf{w} - \mathbf{y}) + \lambda\mathbf{w}$;
- ∇f is $L = \frac{\|\mathbf{X}^T\mathbf{X}\|}{n} + \lambda$ **Lipschitz continuous**.

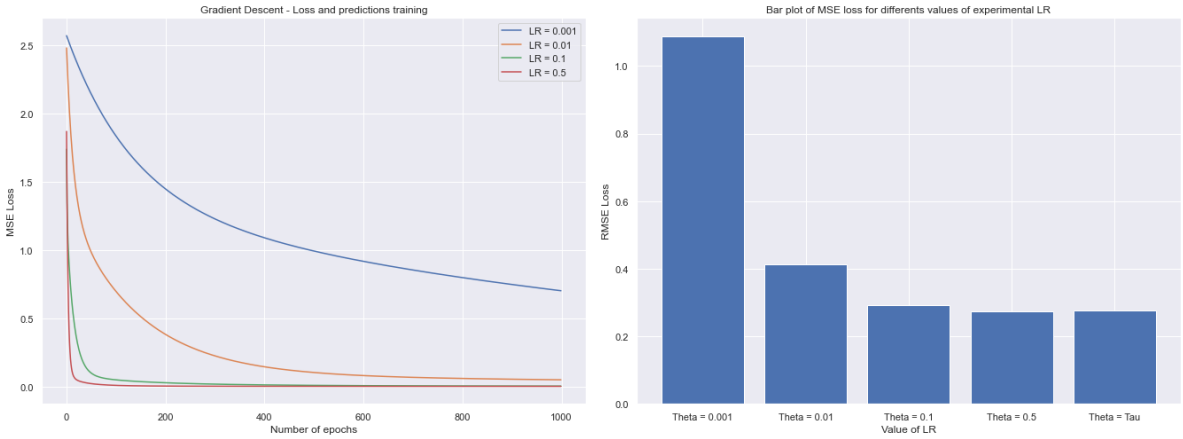
1.3 Implementation of gradient descent and numerical tests

Gradient Descent is a famous algorithm we use to solve optimization problems especially for minimizing our loss function $L(x)$. The algorithm starts by intializing random weights to the vector w, and updates it in the opposite direction of the gradient $\nabla_x L$, with a quantity called **the learning rate** τ.

τ mainly controls the speed of convergence and the final accuracy of the solution. With k the number of current iteration, the algo can be summarized by:

$$\boxed{x^{(k+1)} = x^{(k)} - \tau\nabla_x L\left(x^{(k)}\right)}$$

We implement gradient descent on our training set (beforehand, a split of the data in train set and test has been realize), and we test it with different numerical values of learning rate. We add a small fixed ridge penalty of 0.01. The algorithm stops when the gradient is close enough to 0. From this graph, we can empirically see that the optimal learning rate is lr = 0.5, followed closely by lr = tau.



(a) Influence of learning rate during the Gradient descent

(b) Bar plot of MSE loss for differents values of experimental Tau

Taking these specific learning rates, we notice that their respective gradients reach 0 faster than the others (better convergence).This impression is confirmed by the MSE values which are the lowest for these specific learning rate values.

The question we can ask ourselves now is what is the value of this "tau" which competes in terms of convergence.

1.4 Theoretical tests and comparaison

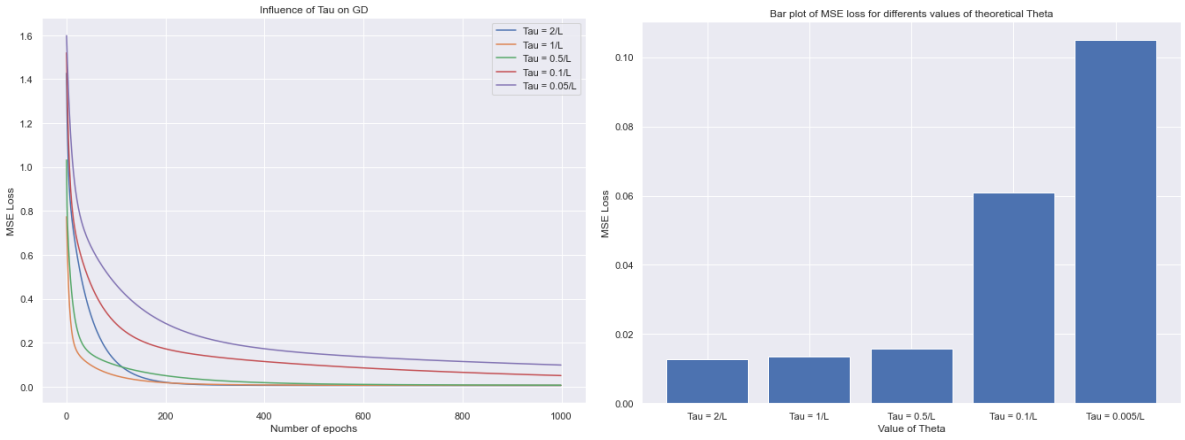
To answer the last question, we will refer to the course. In the case of our convex quadratic loss problem to minimize, the Gradient descent method provides the following theoretical result if we use a learning rate **tau** $\tau = \frac{1}{L}$ with L the lipschitz constant and f our loss function :

$$\begin{cases} x_0 \in \mathbb{R} \\ x^* = \arg\min_{x \in \mathbb{R}} f(x) \\ L = \max(\text{ eigenvalues } (\mathcal{H} \text{ ess } (f)) \\ \mu = \min(\text{ eigenvalues } (\mathcal{H} \rceil \int \int (f)) \\ |f(x_t) - f(x^*)| \leq \frac{L}{2} \left(\frac{L-\mu}{L+\mu}\right)^t \cdot \|x_0 - x^*\|^2 \end{cases}$$

According to the lectures, the $\frac{L}{\mu}$ conditions the convergence rate of the algorithm. The close $\frac{L}{\mu}$ is to 1 , the faster the descent. We have seen in the course, that if $\tau \geq \tau_{\min}$ and $\tau \leq \tau_{\max} \leq \frac{2}{L}$ (with respectively $\tau_{\min}$ and $\tau_{\max}$ the minimum and maximum singular eigenvalue of our matrix features A), the higher convergence speed is reached.

From a theoretical point of view, the learning rate tau $\tau = \frac{1}{L}$ respects this interval constraint : it is the optimal choice of learning rate. The **tau** L constant has been directly implemented in python and then called as learning rate in the different tests. **The results presented above, through the graphic prove this with a higher convergence speed**.

In a second step, to complete the experimental learning rate tests, a comparison of several **theoretical tau** τ values which contains the lipschitz constant was performed to confirm our results.



(a) Influence of the theoretical learning rate during the Gradient descent

(b) Bar plot of MSE loss for differents values of theoretical Tau

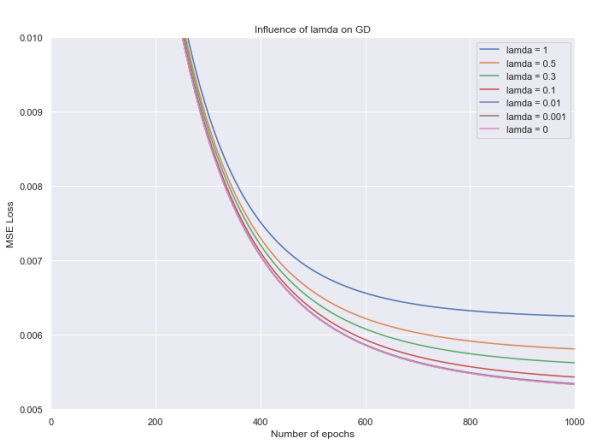
From this graph we can also empirically see that the optimal learning rate is the learning rate tau $\tau = \frac{1}{L}$ of the training dataset. It is followed closely by the learning rate $\tau = \frac{2}{L}$. This impression is also confirmed by the MSE values which are the lowest for these specific learning rate values. It is closed to what we found in the previous pratice part.

We find once again the theoretical results through our example of data set A : the optimal convergence is achieved when $\tau \geq \tau_{\min}$ and $\tau \leq \tau_{\max} \leq \frac{2}{L}$. Beyond of $\frac{2}{L}$, we speak of "explosion".

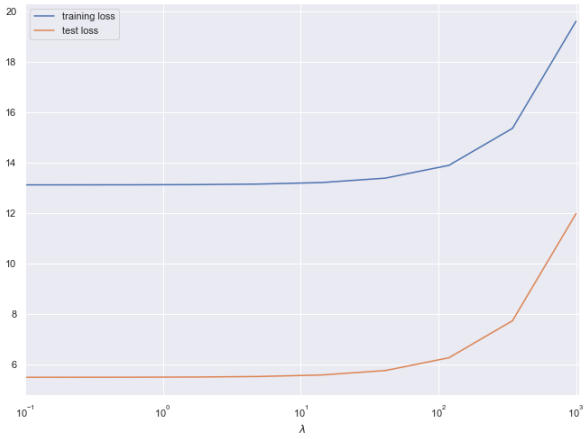
1.5 Ridge penalty test

The Ridge regression helps with strong convex problems and is characterized by a regularization term $\frac{\lambda}{2} ||x||^2$. After testing several values of our learning rate, it is interesting to evaluate the model under different values of λ while fixing our learning rate equal to tau.

Empirically, we can observe a faster convergence toward zero with a λ equal to 0.001.



(a) Impact of Ridge regularization for various λ



(b) Influence of λ to the train and test loss

This first observation was confirmed by the results obtained from a Python algorithm to return a λ value that **minimizes** our loss function. The error on the test set has a minimum (equals to 0.101) with λ ridge equals to 0.001 , by construction.

```
# find optimal Lambda
i = E_test_Ridge.argmin(axis=0)
lambda_Ridge = lbda_list[i]
w_Ridge = W_Ridge[i][:,None]
print( 'Ridge: ' + str(E_test_Ridge.min()*100) + '%' )
print('lambda = {}'.format(lambda_Ridge))
# print('w = {}'.format(w_Ridge))
```

✓ 1.1s

```
Ridge: 101.6470776597684%
lambda = 0.001
```

Figure 5: Result of an algorithm for the choice of an λ optimal

Besides, we can see that the training loss is increasing with the value of λ (right plot above), which is normal because we impose more restrictions on λ. Indeed, if we take the regularization term out of the objective function and convert it to a constraint, it is clear, it is more and more restrictive on the values λ can take, thus adding more bias and increasing the training error. The test error curve shows that Ridge regression helps to better generalize, even if the effect is small in this dataset.

In addition, it is worth remembering that the Ridge regression do not offer sparcity properties. When the value of increases, the coefficients of get closer to 0, but no zero. This is the effect of the regularization, notion which it will be reintroduced later in the report.

2 Part 2 - Automatic differentiation

The goal of this part of the project is to implement backpropagation (reverse-mode autodiff) from scratch using NumPy for a feedforward multilayer perceptron (MLP) with arbitrary number of layers.

The implementation of the different functions can be found in the notebook as well as their explication. They have been implemented based on Mathieu Blondel's slides.

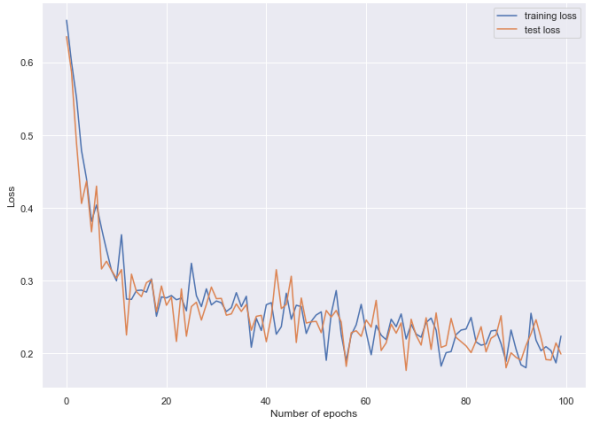
The objective in this section is to focus on the results obtained by our mlp (feedforward multilayer perceptron) from scratch using the stochastic gradient optimizer also implemented from scratch.

Results have been obtained and will be interpreted. All the functions have been applied on **our starting data set, the Housing Boston Dataset, in the continuity of part 1**.

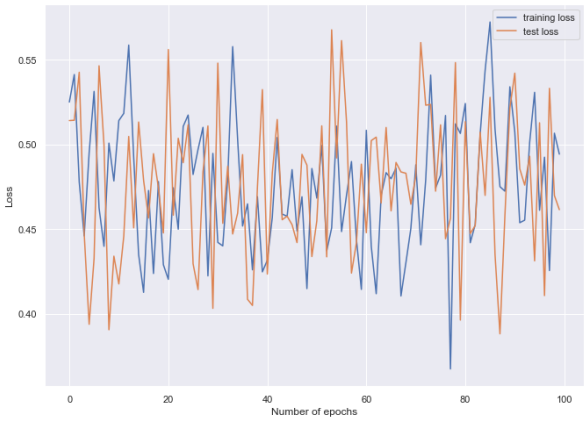
We mainly studied 2 cases:
- The application of an MLP with **two layers** (respectively composed of 2 and 3 neurons)
- The application of a **6-layer** MLP

In order to study the impact of depth (number of layers) and width (number of hidden units)

Below, the train learning and test curves of our loss, the MSE loss for the both cases.



(a) SGD - Mlp with 2 layers



(b) SGD - Mlp with 6 layers

We can observe the variability of the loss descent, a sign of a high variability **due to the random choice of the index**, characteristic to the stochastic gradient optimizer.

With a 2 layer mlp, we notice that the training loss and test loss goes towards 0 at a geometric rate after 100 iterations. Nevertheless, with a larger number of layers (equal to 4 or 6 for example), we notice that **our mlp algorithm converges extremely slowly or even not at all**. Illustrated by the plot on the right, in the case of a 6 layer mlp, the loss does not decrease progressively and remains around 0.455.

Thus,in the case of our dataset, **the number of layers shouldn't be too high**. Indeed, **the more complex the network is the higher the risk that our algorithm doesn't converge**. The risk that our model get stucks in local minima is real in our case. This interpretation can be confirmed in the assessment of the learning task. Indeed, for a 6-layer mlp, the minimal training and test error reach **0.50** (against **0.29 value** loss for the mlp with 2 layers). If the number of neurons or layer increase, the predicted results tend to be less accurate.

On the other hand, increasing the number of hidden units in an MLP can also improve its ability to learn complex representations and fit the training data more closely. However, as the number of hidden units increases, the model can become more computationally expensive to train. Our selected data set has a very small number of data and dimensions (350 data and 14 dimensions). **The nature of our dataset does not imply a consequent number of nodes per layer.**

We achieve quite good results even with **2 neurons per layer** (loss value of 0.29). It is thus obvious that for a small and not very complex data set, **it is absolutely not necessary to apply to our problem a very developed mlp architecture** (in term of number of layers than of number of hidden units)

3 Part 3 - Stochastic Gradient and its variants

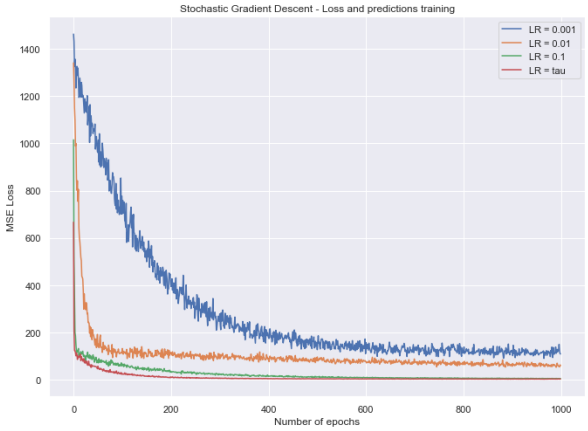
3.1 Stochastic Gradient Descent

The iteration of **stochastic gradient** (also called Stochastic Gradient Descent, or SGD) is given by:

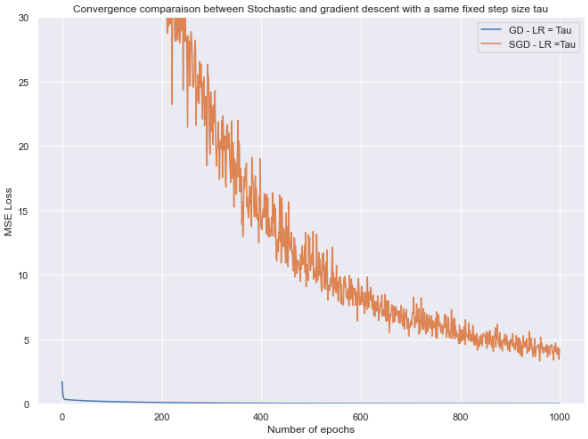
$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \nabla f_{i_k}(\mathbf{x}_k),$$

where i_k is drawn at random in $\{1, \ldots, n\}$. For the purpose of this report, i_k will be drawn uniformly at random.

An implementation of the stochastic gradient algorithm has been realized without batches, in the same setting (regression problem) than the gradient descent. Several learning rates have been tested (the learning rate $\tau = \frac{1}{L}$ remains the most relevant in terms of convergence) in a first step. Then, a comparison of this method was made with the classical gradient descent method, studied in the last part :



(a) Influence of the learning rate during the SG descent



(b) Comparaison GD vs SGD

For constant stepsizes, we observe that choosing a "large" stepsize may prevent the method from converging. On the other hand, using a very small stepsize will lead to slow convergence (as illustrated by the $\frac{0.0001}{L}$ curve). We expect this slow convergence behavior to be followed by an oscillatory phase, which we can readily see with other stepsize choices. The orange and green curves ($\frac{0.01}{L}$ and $\frac{0.1}{L}$, respectively) illustrate the trade-off between having a small stepsize, that guarantees convergence to a small neighborhood of the optimum, and having a large stepsize, that leads to faster convergence (but to a large neighborhood).

Our Xtrain dataset has a size of 354 rows. Our algorithm stops after 1000 iterations. Which is equivalent to $\frac{1000}{354} = 2.8$ epochs so 2.8 iterations of a GD. In the case of SG, 1 epoch is equivalent to n iterations (inversely, 1 iteration is equivalent to $\frac{1}{n}$ epochs).

The stochastic gradient algorithm **is not a descent method**. As such, it is not guaranteed to decrease the function value in a monotone fashion (unlike gradient descent), and it may not even converge on some instances according to the plot above. Per each single iteration, the GD seems to be more efficient than SGD but that is because it relies on n more data per iteration than sgd (only one). In this sense, our observations confirm perfectly what has been discussed during the lectures. According to the course, **The right way of comparing them is by number of epochs**, and in that sense also, sgd is more efficient, especially if the data is quite redundant. Moreover, with sgd, we are using a single data point at a time results in updates that **are n times cheaper than a full gradient step**.

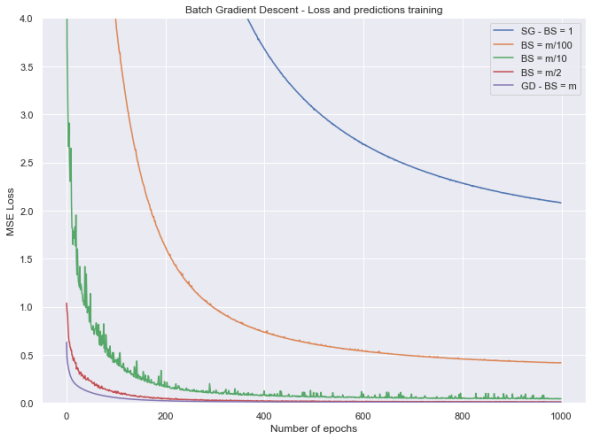
3.2 Batch Stochastic Gradient

We are now interested in a more general version of stochastic gradient, called batch stochastic gradient, is given by the iteration

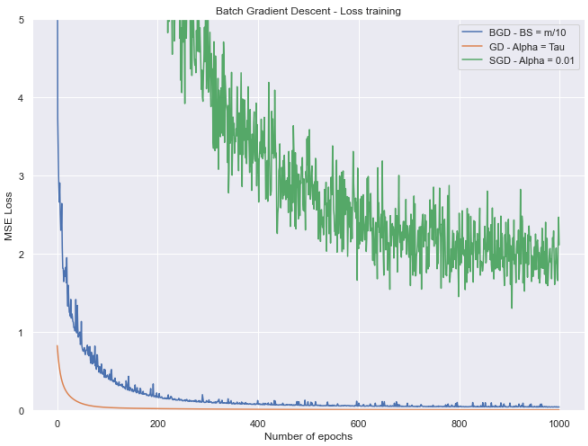
$$\mathbf{x}_{k+1} = \mathbf{x}_k - \frac{\alpha_k}{|S_k|} \sum_{i \in S_k} \nabla f_i(\mathbf{x}_k)$$

where S_k is a set of indices drawn uniformly in $\{1, \ldots, n\}$. For this report, the samples will be drawn without replacement, so that $|S_k| = n$ results in a full gradient step, while $|S_k| = 1$ corresponds to a basic stochastic gradient step. So the choice of the Batch size can be seen as a trade-off between Gradient Descent and Stochastic Gradient. In this report, we will focus on using the same batch size across all iterations.

One of the main hyperparameter of Stochastic Gradient Descent is the **BatchSize**. We have tried to run several Stochastic Gradient Descent for various sizes of mini-batch. Suggested values: $n_b \in \left\{1, \frac{n}{100}, \frac{n}{10}, \frac{n}{2}, n\right\}$ to include stochastic gradient ($n_b = 1$) and gradient descent ($n_b = n$). And finally, we compare the performance of the set of optimization strategies seen at this stage in the report.



(a) Influence of the learning rate during the BSG descent



(b) Comparaison GD vs SGD vs BSGD

It is first important to recall some elements of the course before interpreting our results : an iteration of a given batch method with batch index set S_k consists in $|S_k|$ calls to gradients ∇f_i, or, equivalently, $|S_k|$ accesses to data points. For a method with a constant batch size of n_b, an epoch corresponds to $\frac{n}{n_b}$ iterations with m = len(y) = 506 in our dataset'case.

We compare the performances of Mini Batch for **three values** of the batch size. We test those results with $|S_k| = \text{m}/100$, $|S_k| = \text{m}/10$ and $|S_k| = \text{m}/2$.

To compare each method with a consistent number of iterations, we use **the notion of epoch**. By definition, n iterations of Stochastic Gradient is equivalent to one epoch and n $|S_k|$ iterations of Batch is equivalent to one epoch. We try for 1000 epochs, which is equivalent to 1000n iterations of Stochastic Gradient and to $1000\frac{n}{|S_k|}$ iterations of Batch. According to the left plot, we obtain that Batch Stochastic Gradient has a better minimal value than Mini Batch. So using epochs as a metric, Mini Batch is better than Stochastic Gradient in our case and it always has the advantage of reducing the variance. We obtain that the curve is smoother when $|S_k| = 1$ increases, which means we have less variability.

We can see in the figure that **the higher the batch size is, the faster the convergence because of the configuration of our data set** . As we don't have too much data in our data set, we can rely on GD or choose a high batch size. On the contrary, SGD is not very useful in our case and occurs many more latency because of the many more iterations. In this sense, $|S_k| = \mathbf{m/10}$ can be a good compromise between gradient descent and stochastic gradient.

3.3 Advances variants of Stochastic Gradient

Since Stochastic Gradient Descent is sensitive to the conditioning of our problem, it has been shown that linear transformations can have a non negligeable impact. In this section, we'll explore different improvements that have been made throughout the years to the SGD method and seen in the courses.

Diagonal Scaling is one of them, and the two methods AdaGrad and RMSProp may give good results applied on our problem. This corresponds to rescaling the stochastic gradient step componentwise as follows

$$[\mathbf{x}_{k+1}]_i = [\mathbf{x}_k]_i - \frac{\alpha}{\sqrt{[\mathbf{v}_k]_i + \mu}} [\nabla f_{i_k}(\mathbf{x}_k)]_i ,$$

Where $\mu > 0$ is a regularization parameter, and $\mathbf{v}_k \in \mathbb{R}^d$ is defined recursively by $\mathbf{v}_{-1} = 0_{\mathbb{R}^d}$ and

$$\forall k \geq 0, \forall i = 1, \ldots, d, \quad [\mathbf{v}_k]_i = \begin{cases} \beta [\mathbf{v}_{k-1}]_i + (1-\beta) [\nabla f_{i_k}(\mathbf{w}_k)]_i^2 & \text{for RMSProp ,} \\ [\mathbf{v}_{k-1}]_i + [\nabla f_{i_k}(\mathbf{w}_k)]_i^2 & \text{for Adagrad.} \end{cases}$$

We used the values suggested in the lab session, and as suggested in the notebook of Clément Royer : $\mu = \frac{1}{2\sqrt{n}}, \beta = 0.8$.

We have also tested Adam optimizer method, a method which can be thought as **combining the idea of momentum (that we'll see in the last section of the report) with the diagonal scaling procedure** on which both AdaGrad and RMSProp are based. It starts by computing the exponentially decaying average of past gradients, and the exponentially decaying average of past squared gradients.An iteration of Adam falls into the generic scheme by setting :

$$m_k = \beta_1 m_{k-1} + (1-\beta_1) \nabla_w L\left(w^{(k)}\right)$$

and

$$v_k = \beta_2 v_{k-1} + (1-\beta_2) \nabla_w L\left(w^{(k)}\right)^2$$

and

$$\hat{m}_k = \frac{m_k}{1-\beta_1}$$

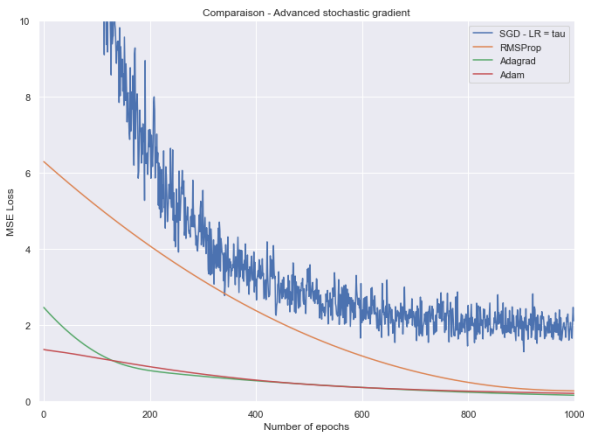
with

$$\hat{v}_k = \frac{v_k}{1-\beta_2}$$

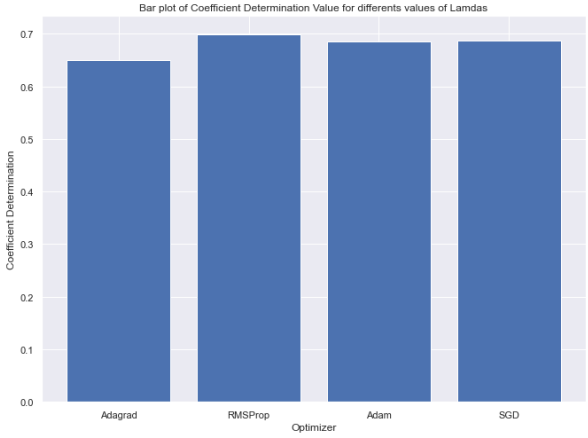
and

$$w^{(k+1)} = w^{(k)} - \frac{\tau}{\sqrt{\hat{v}_k} + \varepsilon} \hat{m}_k$$

Below, the convergence speeds of these methods have been compared :



(a) Advanced stochastic method comparaison



(b) MSE comparaison for our Xtest

From the graph, we can say that **the performance of the Diagonal Scaling and Adam methods outperforms the stochastic gradient** in our Xtrain dataset. This impression is confirmed by the MSE values which are the lowest for these methods.

This is not surprising because the resulting diagonal scaling on the coordinates of f leads to stepsizes that adapt to coordinates that can vary by orders of magnitude.On the other hand, the accumulation process at work in the definition of rk results in stepsizes that are monotonically decreasing, and that converge quickly towards 0, **faster than the classic stochastic method**.

But **it is the Adam method that outperforms all these results with an optimal convergence according to the graph**. This practical results is mostly consistent with the theorical ones discussed during the course.

The Adam method is characterized by a diagonal scaling applied to the coordinates of directions, again according to a weighted average of the coordinates from the previous iterations. This important feature **appears to be responsible for the success of Adam in practice during our training**.

4 Part 4 - Convexity and constrained optimization

In this part of the project, our objective is to implement the conditional gradient algorithm, the projected gradient and Augmented Lagrangian algorithm and to compare their behavior on a convex set. A constrained optimization template associated with its convex set is presented below :

$$\begin{aligned} &\min \frac{1}{2}\|Ax-b\|_2^2 \\ &\text{s.t. } 0 \leq x_i \leq 1 \end{aligned}$$

This problem will evolve according to the tests performed in this part of the project. It is relevant to first recall the nature of each of the methods implemented and tested on this optimization problem, and **compared the structure of their iterates**.

4.1 Projected Gradient algorithm

The idea with the projected gradient is to make step with gradient descent and project obtained point on the feasible set P :

$$\min_{x \in P} f(x) + p(x)$$

where f is a smooth convex function, while p is an extended convex function.

We first performing gradient descent on the then current solution and then projecting it back onto the constraint set. For this method we need to know the projection operator. The projection operator for a given constraint may or may not be known beforehand, it could be computationally too. In this part, we have specifically tested as a projection operator, **the projection onto the l1 ball**.

4.2 Conditionnal Gradient algorithm

The Frank-Wolfe method is **an alternative to Projected Gradient Descent** (or CG) which doesn't involve projections. The Frank-Wolfe method is also known as conditional gradient method. Instead of using a quadratic expansion as shown above for the projected GD method, it uses a **local linear expansion of f**. In the Frank-Wolfe method, we minimize the linear approximation over the constraint set, instead of projecting afterwards separately. We make step **not along the gradient descent, but along the direction that leads to feasible point** :

$$f(x_k + s_k) = f(x_k) + \langle f'(x_k), s_k \rangle \rightarrow \min_{s_k \in D}$$

The minimization is possible by approximating the target function with its gradient and solving a linear program where the gradient of the function at the current point serves as weight for the objective. We'll illustrate in the next part this algorithm using a python solver with the python module with the scipy library, which is used for solving the linear programs. In particular, we will test **the simplex method/solver** (by Laurent Condat).

4.3 Augmented Lagrangian

The idea behind the Augmented Lagrangian method is to add a penalty not to the objective function, but to the Lagrangian. It replaces a constrained optimization problem by a series of unconstrained problems and add a penalty term to the objective :

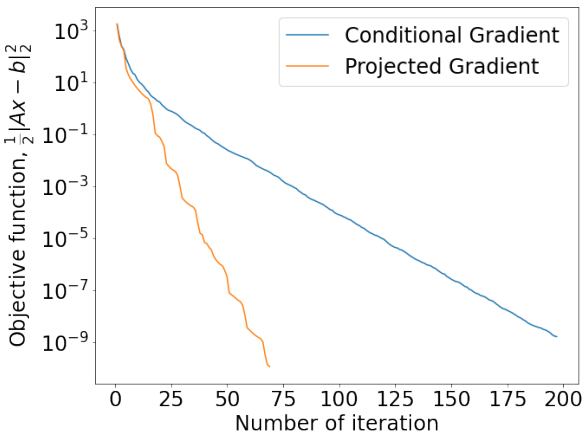
$$M(x,\mu,\lambda,\nu) = f(x) + \sum_{i=1}^{m}\left[\lambda_i g_i(x) + \frac{\mu}{2}g_i^2(x)\right] + \frac{1}{2\mu}\sum_{j=1}^{p}\left[(\nu_j + \mu h_j(x))_+^2 - \nu_j^2\right],$$

where λ are vector of dual variables corresponding to **equality** constraints, ν are vector of dual variables corresponding to **inequality** constraints - See the derivation of $M(x,\mu,\lambda,\nu)$ in lecture notes.

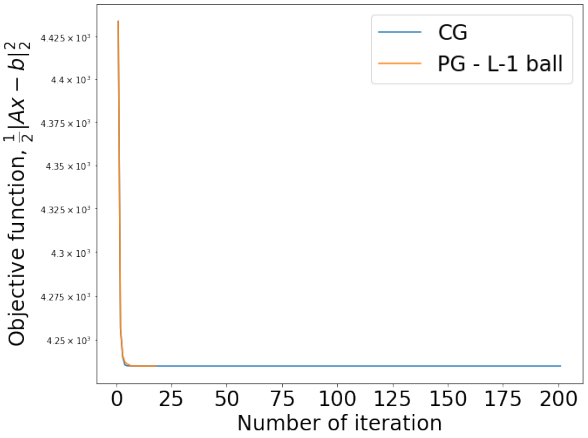
This method was implemented within the framework of this project and then tested in order to observe and understand its functioning.

4.4 Convergence comparaison, tests and interpretation

Taking up these theoretical notions, which are essential to carry out our tests, different comparisons were made in order to evaluate the different methods. The first two graphs focus on a symmetrical comparison between the PG descent and the CG descent by definition. In particular, we tested a variant of the previously presented CG method based on the backtracking search for step size which can make much better progress (from Jaggi 2013). Besides, we focus on the projection on a ball of radius l1 for projected gradient.



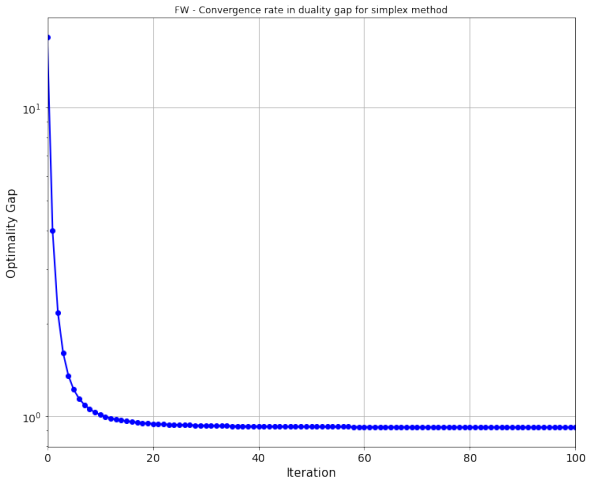
(a) Convergence speed comparaison CG vs PG



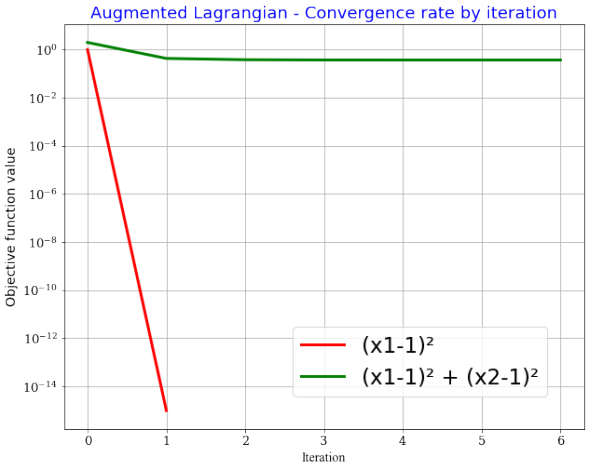
(b) Convergence comparaison CG vs PG with L1 Ball

Through the work requested, we realize that The Frank-Wolfe algorithms (or CG) does not require projections (as projected gradient descent does). However, as illustrated above, CG is not a descent estimator as the objective is not monotonically decreasing over each k. Thus, **the Frank-Wolfe converges slower than projected gradient**, even with backtracking search for step size (plot in the left). **This empirical observation was confirmed in the notebook code Python with a convergence in 196 iterations with the CG but a convergence in only 68 iterations**. Finally, It's important to point out that exist in our case a projection onto the **1-ball** and seems to propose a faster convergence than the method illustrated in the left graph. We show in class that compute projections onto the l1 can be done in O(n log n) time and O(n) memory. This would indeed guarantee a significant speed-up of the optimization process.

An optimality gap comparison with scipy's convex solver is produced graphically below which in theory is (O(1/) where is the number of Frank-Wolfe iterations. We illustrate how to minimize a function over the unit simplex : make a descent step with a guaranteed improvement relative to best approximate direction. By using the simplex method that requires little alteration to the gradient and interpolation in each iteration, we observe that **the optimization of the linearized quadratic objective function can be performed more efficiently**.



(a) Unit Simplex projection convergence



(b) Augmented Lagrangian method convergence for differents convex set

About the augmented Lagrangian, it is is capable of handling convergence to zero is faster,like we can see in the graph above on the right because not only denominator increases, but nominator also converges to zero. Due to time and resource constraints, the study of the simplex method did not go further in its analysis and interpretation.

Therefore,to conclude with all of this empiricals results combined with theoretical notions, **even though the convergence rate of Frank-Wolfe algorithm is slower than Projected Gradient** due to the search for naive descent directions in each iteration, the benefits that the algorithm brings **still outweigh the disadvantages** in our context problem.

5 Part 5 - Proximal gradient and LASSO

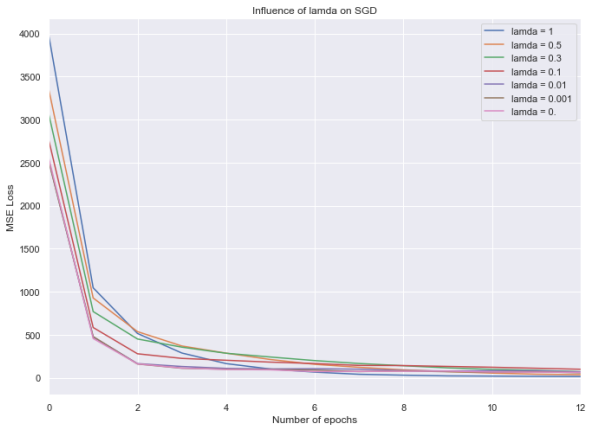
We will now in this section return to our linear regression model problem with the regularization term l2 :

$$\min L(x) = \frac{1}{2n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 + \frac{\lambda}{2}||x||^2 = \frac{1}{2n}\|Ax - Y\|_2^2 + \frac{\lambda}{2}||x||^2 = \frac{1}{n}\left(\frac{1}{2}\sum_{i=1}^{n}(\mathbf{a}_i^T\mathbf{a} - y_i)^2 + \frac{\lambda}{2}\|\mathbf{x}\|^2\right)$$

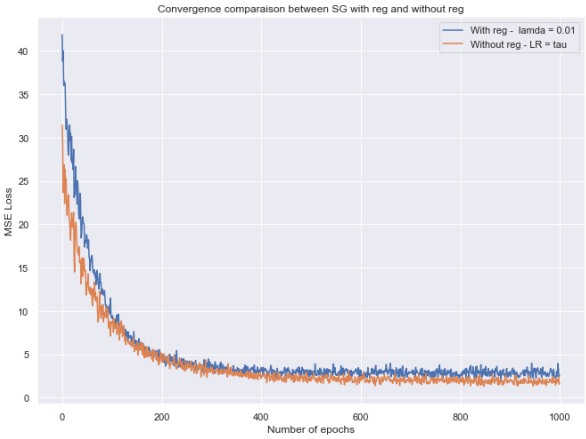
In particular, we will test and compare (with differents regularization terms) the solution of the unregularized problem to those obtained with l2 regularization for the case seen in the part 3 (sgd descent). Finally, we will add a l-1 regularization term to our objective function from Part 1 or Part 3 and solve the resulting problem. Finally, we will replace the regularization term l2 by the term l1 (Lasso problem) and we'll attempt to find a value of the regularization parameter that yields a sparse solution.

5.1 L2 regularization problem and classic problem with SGD

We first find the randomness of the stochastic gradient with this high variance visible through a non-smooth loss descent, whether for the regularized or non-regularized form (plot in the right). The second observation is about the convergence speed : our gradient converges faster to 0 in our non-regularized model after a smaller number of epochs than in the regularized model.

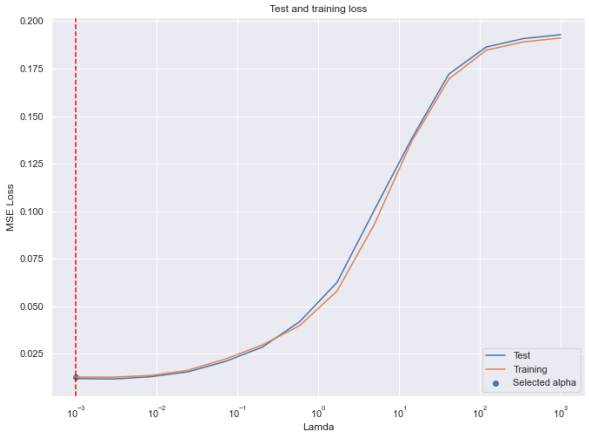


(a) Influence of λ parameter during the SG descent

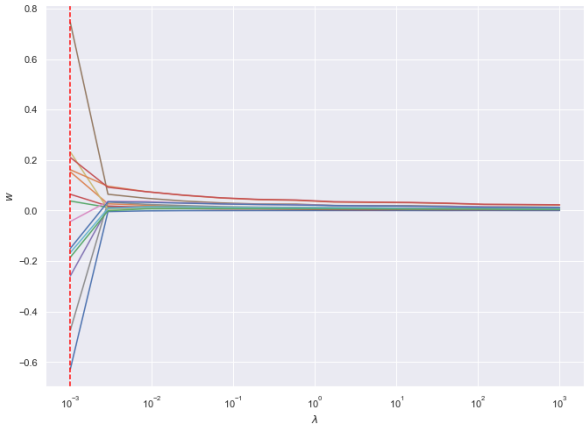


(b) Comparaison convergence with and without L2

Then by empirical observation, the left plot shows a better convergence with a regularization term l2 equal to 0.001. As explained in Part 1 of the project through, we do a grid search on λ and use the test set as a validation set, to find an optimal value for λ. The red line in the plot below left illustrates the optimal lambda value. This first observation was confirmed by the results obtained from a Python algorithm to return a λ value that **minimizes** our loss function. The error on the test set has a minimum (equals to 0.101) with λ ridge equals to 0.001 , by construction.



(a) Influence of λ to the train and test loss



(b) Regularization path with l2

The ℓ_2 norm smoothness out the solution and **leads to solutions with increasingly smaller norms**. We can observe that the solution components shrink in a rather uniform (smooth) fashion.

Besides, we can see that the Ridge regression do not offer sparcity properties. **All the coefficients of are non exactly zero**. When the value of increases, the coefficients of get closer to 0. This is the effect of the regularization. In the following graph, we see that for very low value of , the regularization has not effect whatsoever, and we obtain results similar to non regularized linear regression. For very large values of, all the coefficients reach a neighbourhood around 0. We can still see a group of parameters that have higher coefficients. Maybe we could be able to isolate them better with the **LASSO regularization**.

5.2 L1 regularization study

We solve the Lasso problem with the ISTA algorithm. The proximal operator verifies

$$prox_{lasso(\lambda)}(y) = argmin_x \frac{1}{2}||x-y||^2 + \lambda||x||_1 \tag{3}$$

We saw in class the equation of the proximal operator: it is the soft function: So it yields the

proximal operator

$$prox_{lasso(\lambda)}(y) = sign(y) * (|y| - \lambda)_+ \tag{4}$$

In this section, we will:
- check the convergence
- choose the optimal value for λ
- check the variation of the coefficients of w as a function of λ and observe the sparcity effect

Firstly, the plots below shows:

- **on the right:** the convergence of the energy as a function of the number of iteration: we see that the energy converges to a value close to 0

- **in the middle:** the log of the difference with min(J), which is at the last iteration. We get closer to -2.5, at the 300th iteration, so the difference is lower than 0.01

- **on the right:** the relative difference between two consecutive iterations. At around 300 iterations, we hit a threshold of 0.0001. We will consider that 400 iterations is enough for convergence.

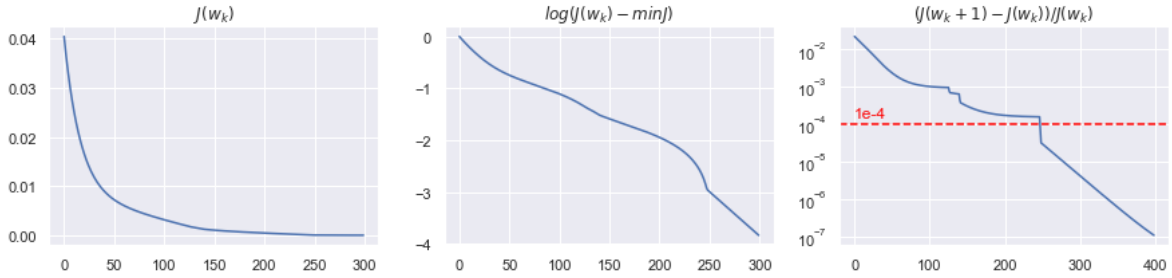
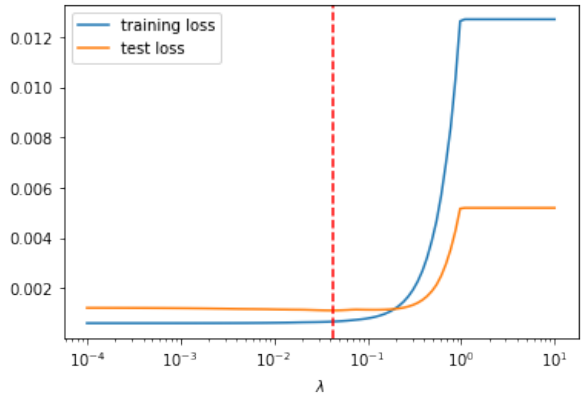


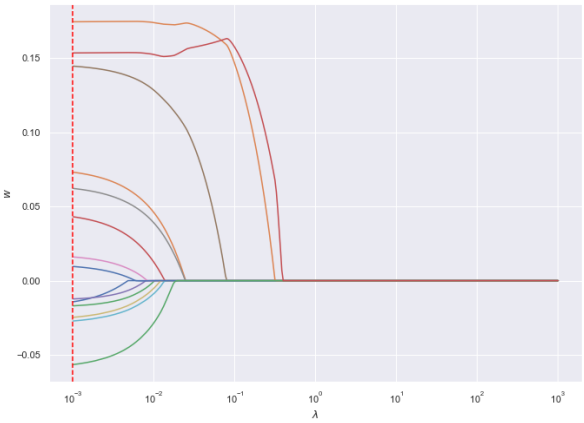
Figure 14: ISTA convergence analysis

On the other hand, we see in the following plot, in the left, the visualisation of a grid search to find an optimal value for λ using the test set as a validation set : according to the graph and python result, **the exact value is 0.001 with a minimal loss value of 0.002**.

Moreover, with $\lambda = 0.001$, the sparsity power of the Lasso regularization is real (right plot). The coefficients of w go to zero when λ is big enough. It enables to find important features and useless features. **In our case, most features are removed**. Only a few remain.



(a) Optimal λ choice for the trade-off : sparsity and fitting]



(b) Regularization path with l1

Thus, for the data-fitting and sparsity points of view, the value $\lambda = 0.001$ **appears to be a relevant** for our trade-off problem.

5.3 L1 and L2 regularization comparaison

Obtaining the value of this parameter allows us to get a better idea of the impact of an L1 and L2 regularization on our data. Indeed, through the barplot illustrated in the next page, a visual comparison allows us a better understanding of its issues.

We can see that of course the coefficient values are greater and not null for more variables with the standard regression. With Ridge regression, wee see that the coefficients are much closer to 0. LASSO non null coefficient are much rarer.**This confirms our statements**.

We can also see that the coefficient for "indus" variable is negative for Lasso regression, and positive for Ridge. Several variables have correlated effects. The selection variable procedures **might have eliminated the variables correlated with "indus" variable**.

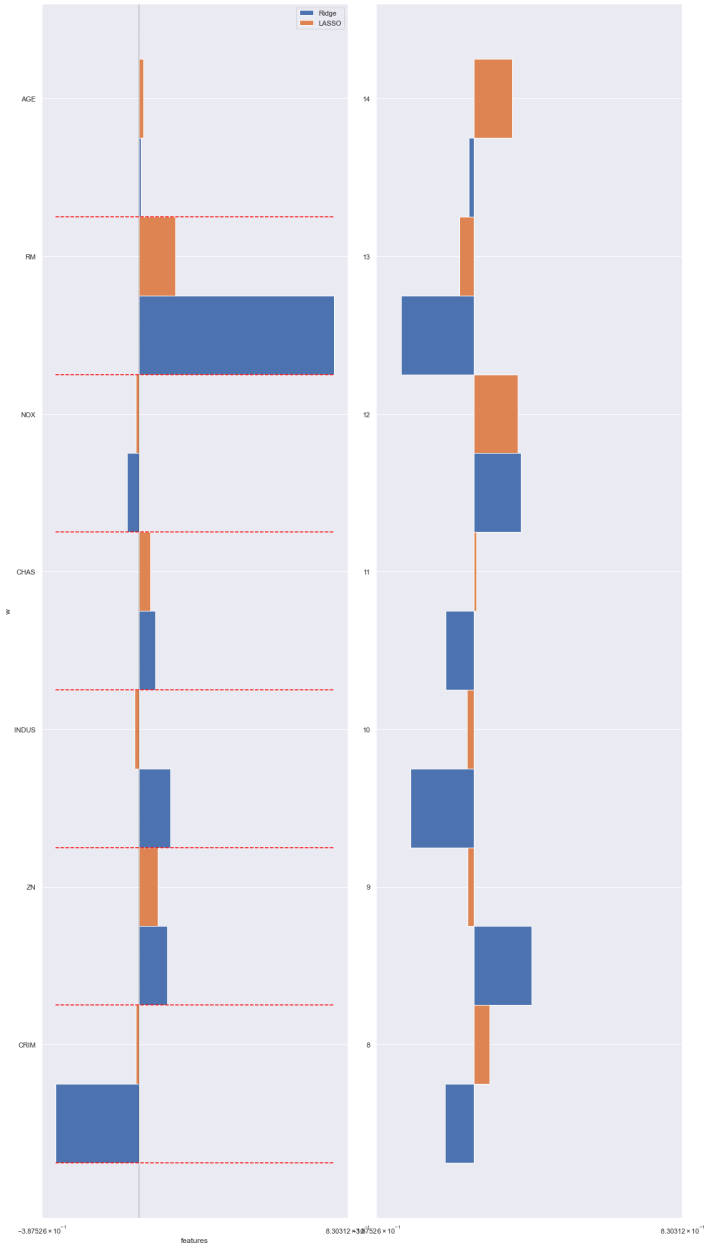


Figure 16: Sparcity level for L1 (orange) and L2 regularization (blueu) applied in our problem with λ optimal

6 Part 6 - Large-scale and distributed optimization

This part of the report is concerned with coordinate descent techniques, that are particularly suitable for large-scale problems in which coordinates (or groups thereof) can be updated in a independent fashion. This setting is increasingly common as the dimension of the problem grows.

A typical coordinate descent method starts with $\mathbf{x}_0 \in \mathbf{R}^d$. At every iteration k, the method picks a coordinate $j_k \in \{1, \ldots, d\}$, a stepsize $\alpha_k > 0$, and performs the following iteration

$$\mathbf{x}_{k+1} := \mathbf{x}_k - \alpha_k \nabla_{j_k} f(\mathbf{x}_k) \mathbf{e}_{j_k},$$

where ∇_{j_k} denotes the partial derivative with respect to the i_kth coordinate, i.e. $\nabla_{j_k} f(\mathbf{x}) = [\nabla f(\mathbf{x})]_{j_k}$ for any $\mathbf{x} \in \mathbb{R}^d$, and $\mathbf{e}_{j_k}$ is the i_kth vector of the canonical basis in $\mathbb{R}^d$.

The coordinate descent update can be rewritten as $\mathbf{x}_{k+1} := \mathbf{x}_k + c_k \mathbf{e}_{j_k}$, where

$$c_k = \text{argmin}_{c \in \mathbb{R}} \; q_k(c) := \nabla_{j_k} f(\mathbf{x}_k)(c - [\mathbf{x}_k]_{j_k}) + \frac{1}{2\alpha_k}(c - [\mathbf{x}_k]_{j_k})^2.$$

6.1 Randomized block coordinate descent

We now focus on a randomized coordinate descent approach, wherein an index j_k is drawn uniformly at random at every iteration in $\{1, \ldots, d\}$, and the new iterate is given by

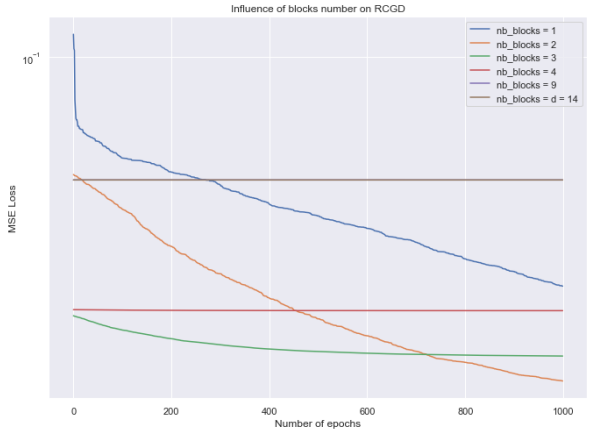
$$\mathbf{x}_{k+1} := \mathbf{x}_k - \frac{\nabla_{j_k} f(\mathbf{x}_k)}{L_{j_k}} \mathbf{e}_{j_k},$$

Specifically, we will select block with randomized choice and do a gradient descent step or exact minimization over it. We talk about now of **Randomized block coordinate descent or RBCD**. At the kth Iteration of RBCD, we have :

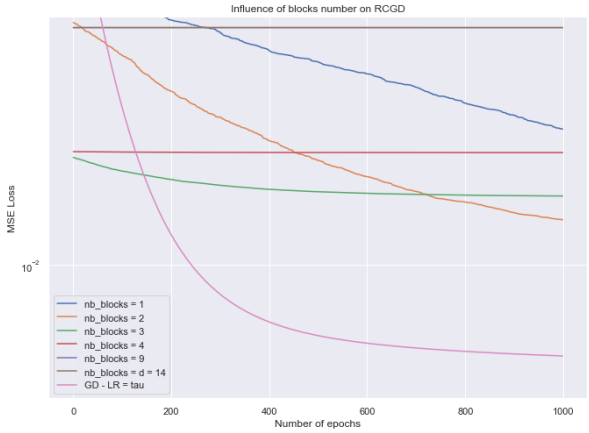
- Randomly choose $i^k \in \{1, 2, ..., n\}$.
- $$\mathbf{x}_{i^k}^{k+1} = \mathbf{x}_{i^k}^k - \frac{1}{\beta_{i^k}} \nabla_{i^k} f(\mathbf{x}^k),$$
$$\mathbf{x}_i^{k+1} = \mathbf{x}_i^k, \; \forall i \neq i^k.$$

Figure 17: RBCD Algorithm iteration

We do a gradient descent (or exact min) step over block i, then exact minimization over block n. We apply randomized block coordinate descent to our problem by considering a full batch of your dataset and for different values of number of blocks.. Then we will compare it with gradient descent method.



(a) Optimal RBCD convergence for differents block



(b) Comparaison with Gradient Descent

We notice an impact of the number of blocks involved in the gradient descent on the convergence speed of our algorithm. Indeed, **the more the number of blocks increases, the more the algorithm has difficulty in converging at a correct speed**. The algorithm does not even converge from a number of blocks equal to 4

So, we can observe that **a little block size is preferable**.

Moreover, the difference in convergence speed with the gradient descent method is striking. Indeed, we observe that my RBCD method **converges our algorithmic problem less quickly than the GD**.

6.2 Randomized block coordinate descent with stochastic gradient

To combine randomized block coordinate descent with stochastic gradient, we will need to modify the block coordinate descent algorithm to use a random subset of the coordinates in each iteration instead of optimizing all of the coordinates at once.

(a) RBCD convergence with SG

(b) Comparaison with Gradient Descent

We immediately notice that the phenomenon highlighted in the previous section is repeated and even reinforced with the stochastic gradient:

- **our problem converges only when the number of blocks is equal to 1 or 2**. It does not converge for the other cases.

- Moreover, in the case where the convergence is maintained, it is much less obvious, with a speed of convergence more reduced **because of the stochastic phenomenon creating a certain variability and a non-smooth effect of our descent**.

As in the last section, when we combine randomized block coordinate descent with stochastic gradient, the convergence **is not faster than the classical gradient descent method**. Besides, **the loss value obtained is always lower with our gradient descent optimizer.**

To sump up, in our case, the big difference is that we are not attaining a global minimum over the coordinate wi in a direction where all other points are held fixed, but rather slowly descending in the direction of a global minimum of our loss function, one point at a time. The RBCD is **a step-wise version of gradient descent but doesn't converge at any faster rate**.

The cause could come from the nature of our problem to solve. It is quite common to state that the coordinate gradient algorithm family is more efficient when applied to a LASSO problem (glasso algorithm - Friedman et al. 2007). In particular, the method outperforms existing methods if one block has much worse (but finite) smoothness parameter than the remaining ones.

7 Part 7 - Advanced topics on gradient descent

Based on the results of the previous sections, we want two things:

- **To make smaller steps in regions** of high curvature to dampen oscillations.
- **To make larger steps** and accelerate in regions of low curvature.

7.1 Heavy Ball method with Gradient Descent comparaison

One way to do both is to guide the next steps towards the previous direction. We can achieve this via a simple tweak to the gradient descent update rule :

$$x_{t+1} = x_t - \alpha \nabla f(x_t) + \underbrace{\beta(x_t - x_{t-1})}_{\text{momentum}}$$

Figure 20: Heavy Ball iteration

This idea comes from Polyak, and is also called the heavy ball method. Intuitively, a heavier ball **will bounce less and move faster through regions of low curvature** than a lighter ball due to momentum. This has two effects, related to the two goals stated:
- We penalize changes in direction to take smaller steps
- When the direction doesn't change we take bigger steps

Let's apply momentum in our setting, to our regularized model used in part 1 and compared its performance with gradient descent :

Firstly, we obtain a resulat from a Python by gridsearch algorithm which correspond to the β value that **minimizes** our loss function. This one is equal to 0.5370 in our case. The error on the test set has a minimum (equals to 0.00675) with optimal β momentum equals to 0.5370 by construction.

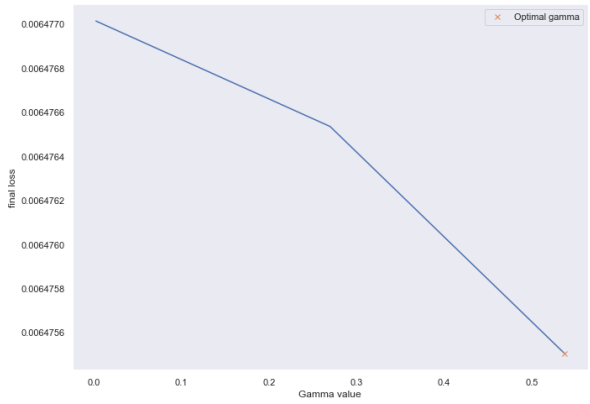
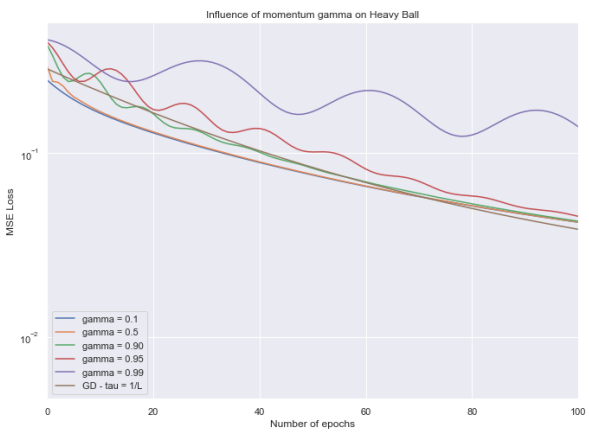
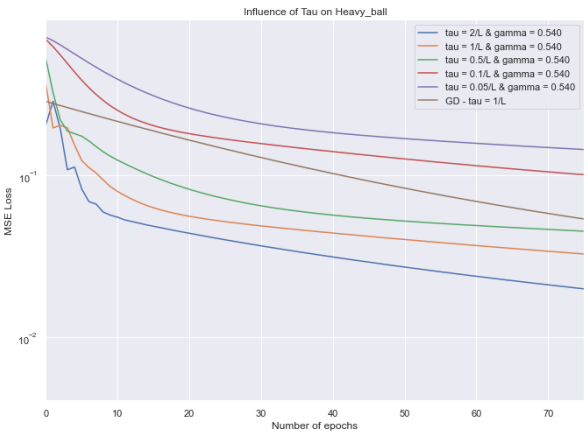


Figure 21: Optimal momentum term by gridsearch

To confirm this result, several plots illustrating the convergence of our method for several values of Beta have been realized and illustrated below :



(a) RBCD convergence with SG



(b) Comparaison with Gradient Descent

The left plot confirms the previous results with a faster convergence in the case where the momentum term **is equal to 0.5370** (orange curve). Therefore we will keep with his value for the others tests in this part. The parameter β is called **the damping factor**. β must be greater than zero, because if it is zero, we are only doing a gradient descent. It must also be less than 1, otherwise everything blows up. For higher beta values (close to 1), we can notice **more frequent and especially larger oscillations**,and therefore a tendency to decrease very slowly.

The graph on the right shows that the optimal learning rate is included in this interval $\tau \geq \tau_{\min}$ and $\tau \leq \tau_{\max} \leq \frac{2}{L}$, **as explained in the first part of the report**. Thus, **we obtain an optimal convergence**, in our case with a step size lower than 2/L (with L, the Lipschitz constant) and a beta momentum equal to 0.5370. These values ensure a good convergence rate as well as a good end accuracy and low oscillations.

The step size must generally be decreased as the momentum parameter is increased to maintain convergence. If our β is increased from 0.5 to 0.59, the learning rate **must be decreased by a factor of 10** in order to keep an acceptable convergence speed.

In term of performance on test set, the results are indisputable. Our MSE loss is much smaller (0.001 with a momentum optimizer) than the gradient descent without (0.007).

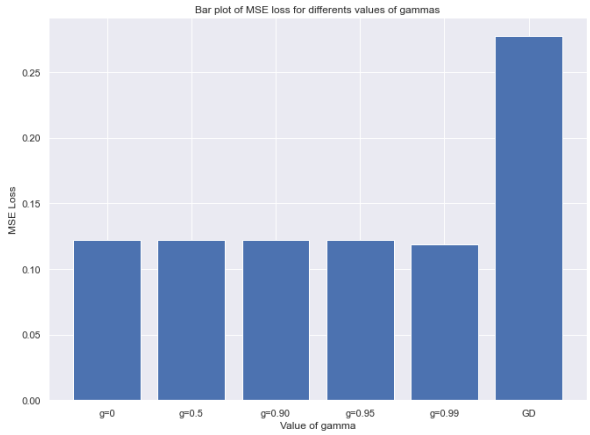


Figure 23: Barplot MSE loss for gradient descent with and without momentum

If we compare the loss value for different values of our momentum, we observe that the beta value equal to 0.95 allows a minimal loss. However, we cannot draw any conclusions from this **as the difference between the loss and the other momentum values is minimal**.

As we can now see from previous plot, **the results with momentum are undoubtedly better**, in the sense that we approach the theoretical minimal loss way faster with the Heavy Ball method and a quite high momentum than without.The better performance of our gradient descent optimizer with momentum is also confirmed from a performance point of view on our test dataset (cf barplot above). If the number of dimensions in our dataset were even larger, the difference would have been even greater and visible.

7.2 Non convex loss function study

In this last section, we propose a non-convex loss function for our initial classic regression problem. We add a non-convex penalty to our objective, and more precisely the p-norm for p belonging to [0,1].

To compute our gradient descent with this change, we have added the parameter p in our algorithm allowing us to call it to add the regularization term in the computation of our gradient.

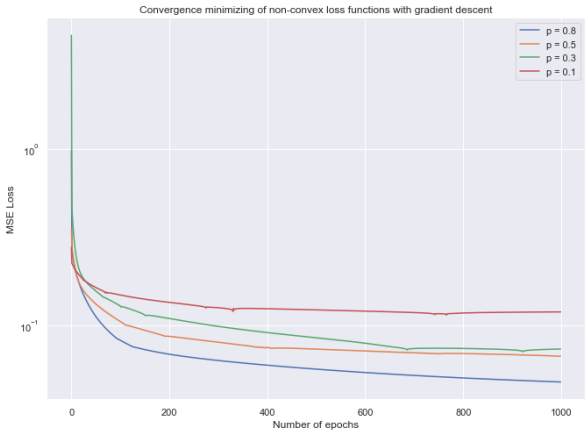
Then, we obtained the speed of convergence of our non-convex loss function for different values of p, represented in the left plot :

We notice that the higher the value of the p-norm the higher the speed of convergence in our case.

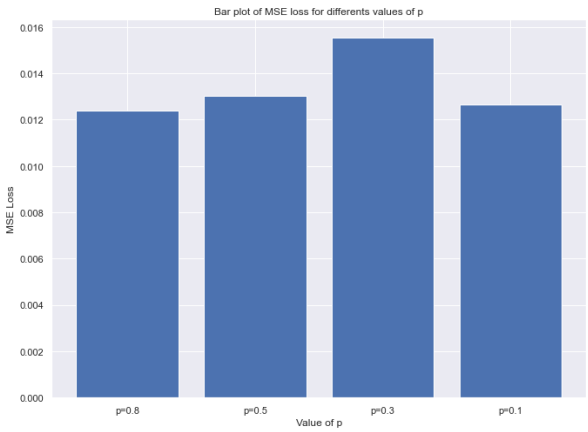
```
while i<itemax:
    y_pred = X.dot(theta)
    grad = np.dot(X.T,(y_pred-y))+lamda*(p*(np.abs(theta)**(p-1))) #add non convex reg
    theta = theta - alpha*grad
    l_loss.append((1/2*(y_pred-y)**2+lamda*np.sum(np.abs(theta)**p)).mean())
    i+=1

return theta,l_loss
```

Figure 24: Gradient descent implementation with non convex loss



(a) GD implementation with non convex loss



(b) MSE value for different p-norm

Finally, to evaluate the quality of our regression model prediction, we used the coefficient of determination.The graph on the right shows that we obtained good prediction results (of house prices) on our Xtest with more than honorable values of determination coefficient (about 0.70). This means that our regression model is rather in line with the collected data.

8 Conclusion

This project presented and compared different methods to solve differents linear regression problems thanks to the empirical risk minimization. We could explore the various descent methods and some optimization analysis. Our goal was to improve the global convergence rate and accuracy of optimal solutions from differents points of view in term of modelization problems.

Besides, we can note that most of the theorical results we studied in class can be observed in practice (and maybe the stochastic properties would be even more obvious if we had a larger Data Set). Moreover,**this project give us a first good approach of algorithms such as Gradient Descent, Stochastic Gradient, Mini Batch or ISTA, which are very useful in many types of practical problems**.

To go further in this study, and go beyond the questions, it would have been interesting to combine the stochastic gradient with a momentum term in part 3. And even to realize the contribution of the technical Nesterov momentum by comparing it with the Heavy Ball method.

It would also have been interesting to apply the **fast iterative shrinkage-thresholding algorithm (FISTA)** which preserves the computational simplicity of ISTA but with a global rate of convergence which is proven to be significantly better.

Through the "Housing Boston" data set, **I notice globally that the results obtained are in phase with the theoretical results proven in class**. The few minimal inconsistencies obtained and quoted in this report are due to the nature of the chosen data set, a low dimensional data set and to a very minimal number of data (354 data in our training data set).

References

[1] Laurent Condat. Fast Projection onto the Simplex and the l1 Ball. Mathematical Programming, Series A, 158(1):575–585, July 2016

[2] Mit.Edu - Block Coodinate Descent http://people.csail.mit.edu/joanne/WOLA18-slides/Diakonikolas-Jelena

[3] $AndersenAng - Projectionontounitl1ball - https://angms.science/doc/CVX/Proj_l1.pdf$

[4] Understanding of momentum-https://boostedml.com/2020/07/gradient-descent-and-momentum-the-heavy-balmethod.html

[5] Stat cmi.edu Coordinate Descent https://www.stat.cmu.edu/ ryantibs/convexopt-F16/lectures/coord-desc.pdf

The codes implemented on this project are based and inspired on the Jupyter Notebooks of Gabriel Peyré and of Clément Royer.