Student Name: Mounir Messaoudi
Student ID: 22204483

Optimization Machine Learning
Project

Dauphine | PSL UNIVERSITÉ PARIS

Part 1 - Gabriel Peyré (gradient descent)

Question 1 : Load an original dataset, comment on this dataset (what are the features, the dimensions of the problem, how does the correlation matrix looks like)

This is my dataset from Kaggle, about red wine quality, taking the parameters of the wine as features : fixed acidity, volatile acidity, citric acid, residual sugar, chlorides, free sulfur dioxide, total sulfur dioxide, density, pH, sulphates and alcohol. (all float)
We are working on $n = 1359$ samples with features $x_i \in \mathrm{R}^p$ in dimension $p = 11$. The goal is to predict the quality value $y_i \in \mathbb{R}^{\mathrm{l}}$.
Here is the correlation matrix :

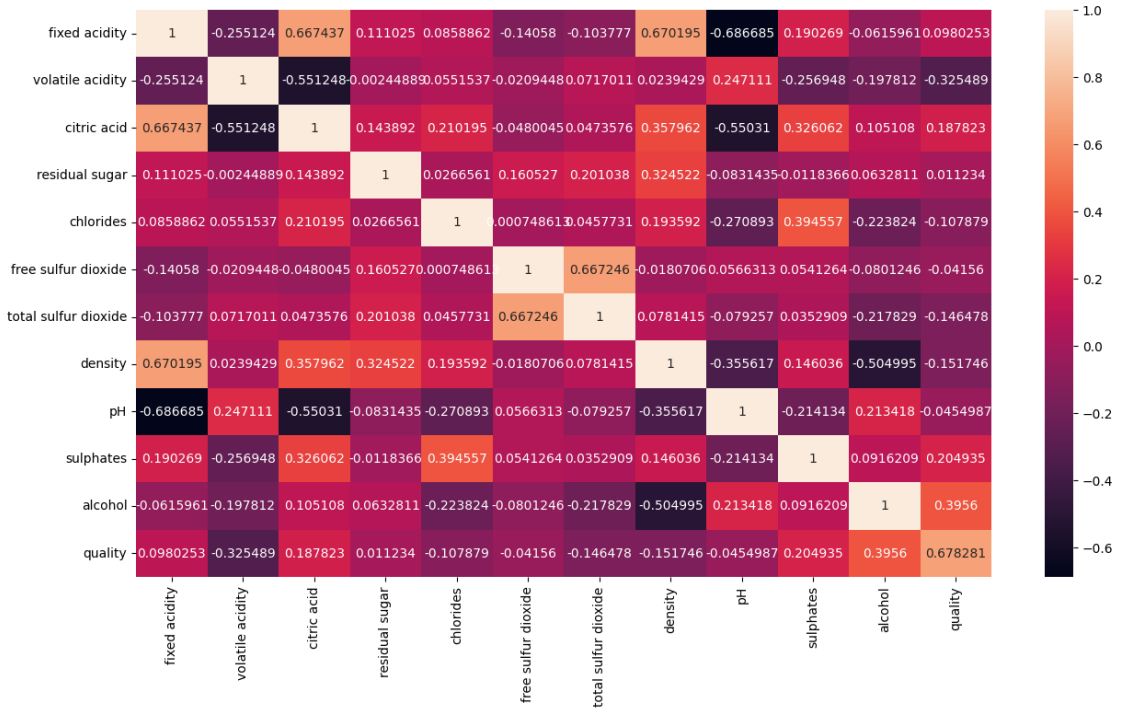


Figure 1: correlation matrix

About the correlation matrix 1, we can do the following observation :

- "fixed acid" is positive correlated (> 0.6) with "citric acid" and "density", but negative correlated with "pH"($-0.6 >$). Poor correlated with the rest.

- "volatile acidity" is negative correlated with citric acid ($-0.5 >$). Poor correlated with the rest.
- "citric acid" is negative correlated with pH ($-0.5 >$). Poor correlated with the rest.
- "residual sugar" and "chlorides" are poor correlated.
- "free sulfur dioxide" is positive correlated with "total sulfur dioxide" (> 0.6).
- "density" is negative correlated with "alcohol" (around 0.5)
- "alcohol" is positive correlated with "quality" (around 0.5)

Finally, the more interesting, what is correlated with quality :

- alcohol have the highest correlation (+ 0.47), that's a non-negligible influence over the quality.
- Then we have "volatile acidity) with a negative correlation (-0.39), that's a moderate influence, but still interesting to look.
- Finally, "sulphates" and "citric acid" have low influence (around 0.23), but are still interested in sensible product as wine (I guess).

Question 2 : Implement gradient descent for regression (l2 loss) with a small ridge penalty. Display the convergence rate on the training loss for several fixed step sizes.

We are working on the following function (to minimize):

$$\min_x f(x) = \frac{1}{2n}||Ax - y||^2 + \frac{1}{2}\lambda||x||^2$$

where $\lambda > 0$ is the regularization parameter, with the following gradient of f :

$$\nabla f(x) = \frac{1}{n}A^\top(Ax - y) + \lambda||x||$$

I implemented a gradient descent for regression (l2 loss) with a ridge penalty $\lambda = 0.01$ as follow :

$$x_{k+1} = x_k - \tau_k \nabla f(x_k)$$

Here is the result for different $\tau_k = [0.17, 0.34, 0.51, 0.66, 0.68]$ (choosing especially for the next question)

Looking this graph 2:

- 0.17, 0.34, 0.51, 0.66 are converging, but 0.17 seems to be really slow (could be interesting to check how it is converging over many steps). We have to check the difference to the optimal value, to see the best between 0.51 and 0.66.
- 0.69 is diverging.

I tried the model on test-set to check the model mean square error and I got the same result :

- For tau = 0.17 the error on the test set is 0.10274363978706
- For tau = 0.34 the error on the test set is 0.10132009330945999

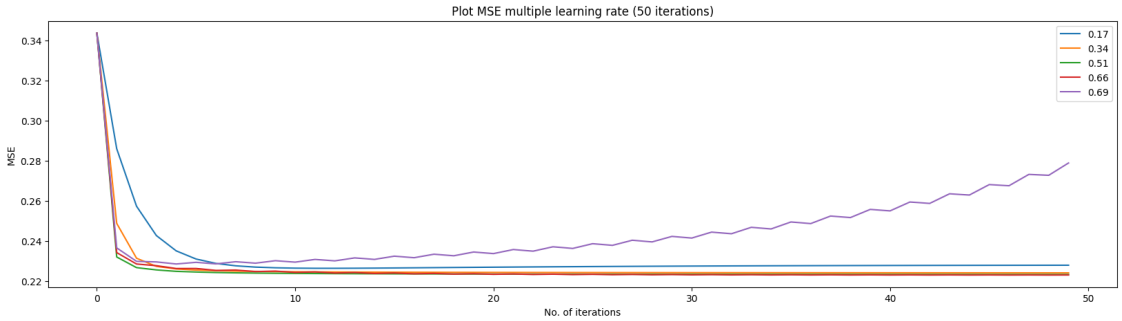


Figure 2: Gradient Descent for different step size

- For tau = 0.51 the error on the test set is 0.10129650107981125
- For tau = 0.66 the error on the test set is 0.1012455650016346
- For tau = 0.69 the error on the test set is 0.12076275986159984

The best performing model is for tau = 0.66 with an error on test set of 0.1012455650016346

question 3 : What is the optimal step choice ? How does this compare with the theory ?

The theory said that the optimal step size is lower than :

$$\tau \leq \tau_{\max} = \frac{2}{{AA^\top}_{op}}$$

where $\cdot_{op}$ is the maximum singular eigenvalue.
Besides, the optimal step size to minimize a quadratic function $< Cw, w >$ is

$$\tau_{\text{opt}} = \frac{2}{\sigma_{\min(C)} + \sigma_{\max(C)}}$$

where $\sigma_{\min(C)}, \sigma_{\max(C)}$ are respectively the minimum and maximum singular eigenvalue and $C = AA^T$. By calculation, $\tau_{\max} = 0.6801491488961905$ and $\tau_{\text{opt}} = 0.6675449160241176$ that's why 0.69 is diverging and 0.66 is converging so well on 2. To enforce the fact that 0.66 is the best solution, I plotted the log of the distance to the optimal value : $log(MSE(f(x)) - MSE(f^*(x))$

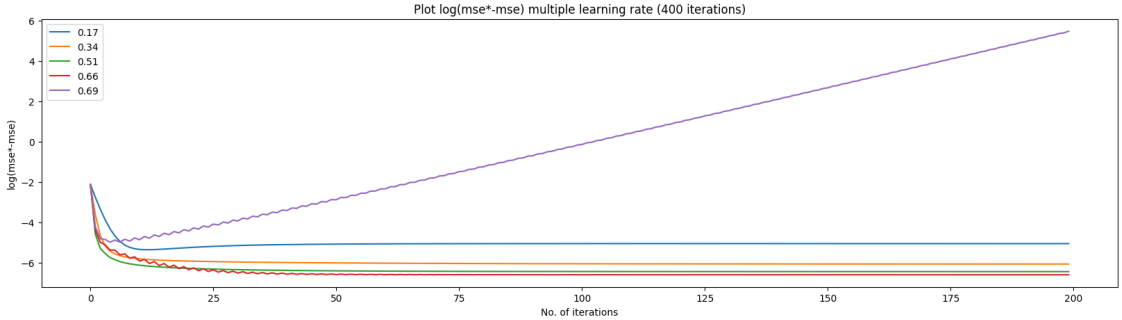


Figure 3: Distance to the optimal value for different step size

Here 2 we can check that 0.66 is closer to the optimal value than 0.5, as it is lower. As the result and the theory give the same stepsize, we can conclude that the theory and the result are coherent in my case.

Question 4: Show the regression performance on the test set as the ridge penalty changes.

I applied the regression performance on the test sets as the ridge penalty was changing : In

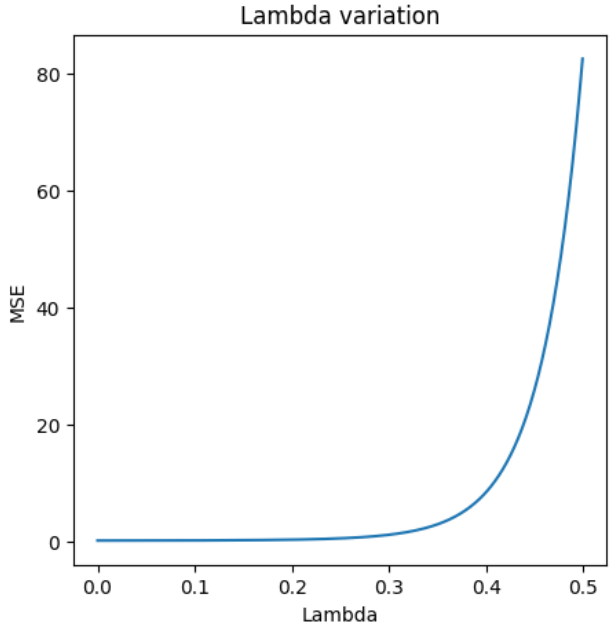


Figure 4: Distance to the optimal value for different step size

our case, this result show that the ridge penalty is not really working, the best is to set the ridge parameter to 0, so to use an unreguralized model.

Part 2 - Mathieu Blondel (automatic differentiation)

voir le notebook

Part 3 - Clément Royer (SGD)

This is my dataset from Kaggle, about red wine quality. As in part 1, we will perform a regression. The function we want to optimise is :

$$f(x) = \|Ax - y\|^2 = \sum_{i=1}^{n} f_i(x)$$

Where each f_i is a function only of the i-th datapoint and $f_i(x) = (A_i.x - y_i)^2$, thus the gradient with the i-th datapoint is $\nabla f_i(x) = (A_i.x - y_i).A_i^\top$
Note that the problem with no regularisation term is exactly the same as the problem above when

setting $\lambda = 0$: In this project, conversely to project 1, we will use the unregularised problem.
The iteration of stochastic gradient (also called Stochastic Gradient Descent, or SGD) is given by:

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

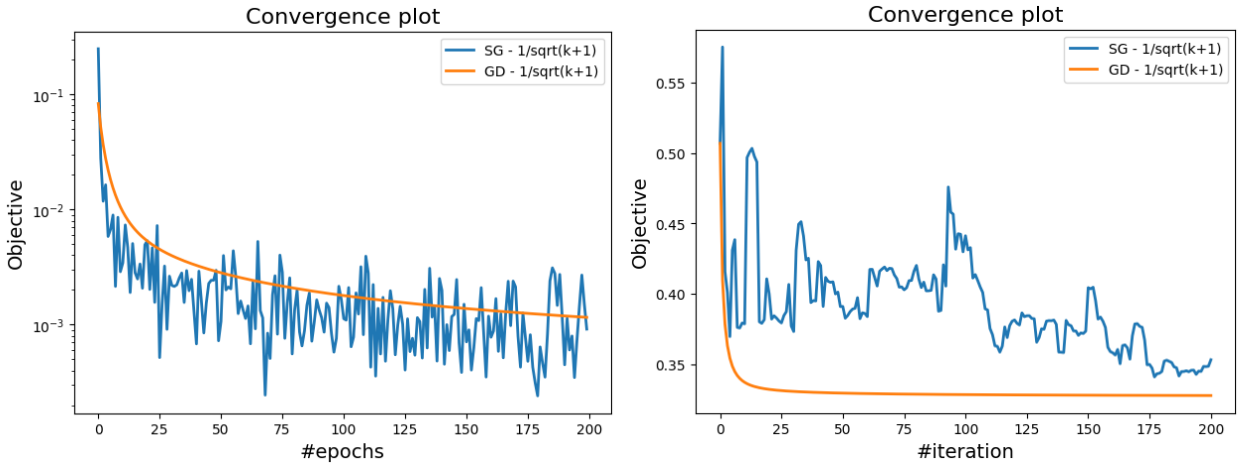
where i_k is drawn at random in $\{1, \ldots, n\}$.
We can define a more general version of SG, called batch stochastic gradient (BSG), given by the iteration

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

where S_k is a set of indices drawn uniformly in $\{1, \ldots, n\}$. 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. In this question, we will focus on using the same batch size across all iterations.

Question 1: Implement stochastic gradient and compare its performance with that of your gradient descent implementation from Part 1 on the same problem and dataset. What happens when both methods are run using the same stepsize? Do your observations confirm what has been discussed during the lecture?

We must compare the performances of both algorithm with two points of view: in terms of epochs (i.e. the number of times we went through the complete number of datapoints) and the number of iteration of the algorithm.



(a) Convergence plot in terms of epoch

(b) Convergence plot in terms of iteration

Figure 5: Attack raidus effect on the accuracy, with and without adversarial training

The blue curve illustrates a commonly observed behavior of SG, characterized by rapid progress during the first iterations, followed by an "oscillating phase". Besides, as we saw in course, we can see :

- in terms of epochs : As we access to the full dataset for both, we can see that stochastic gradient is better but keep oscilate around a "solution".

- in terms of iteration : As stochastic gradient access 1 point per iteration, the convergence is not efficient as the gradient descent.(but SG have a lower cost in terms of access to the data point)

Question 2: Find a value for the batch size that realizes a good compromise between gradient descent and stochastic gradient.

In order to see the best performance, I chose to plot per iteration, to see the real impact of batch on the result.

As the course, we can see the oscillation proportional to the batch size. Besides, the convergence seems closer to the gradient descent. Looking the graph above :

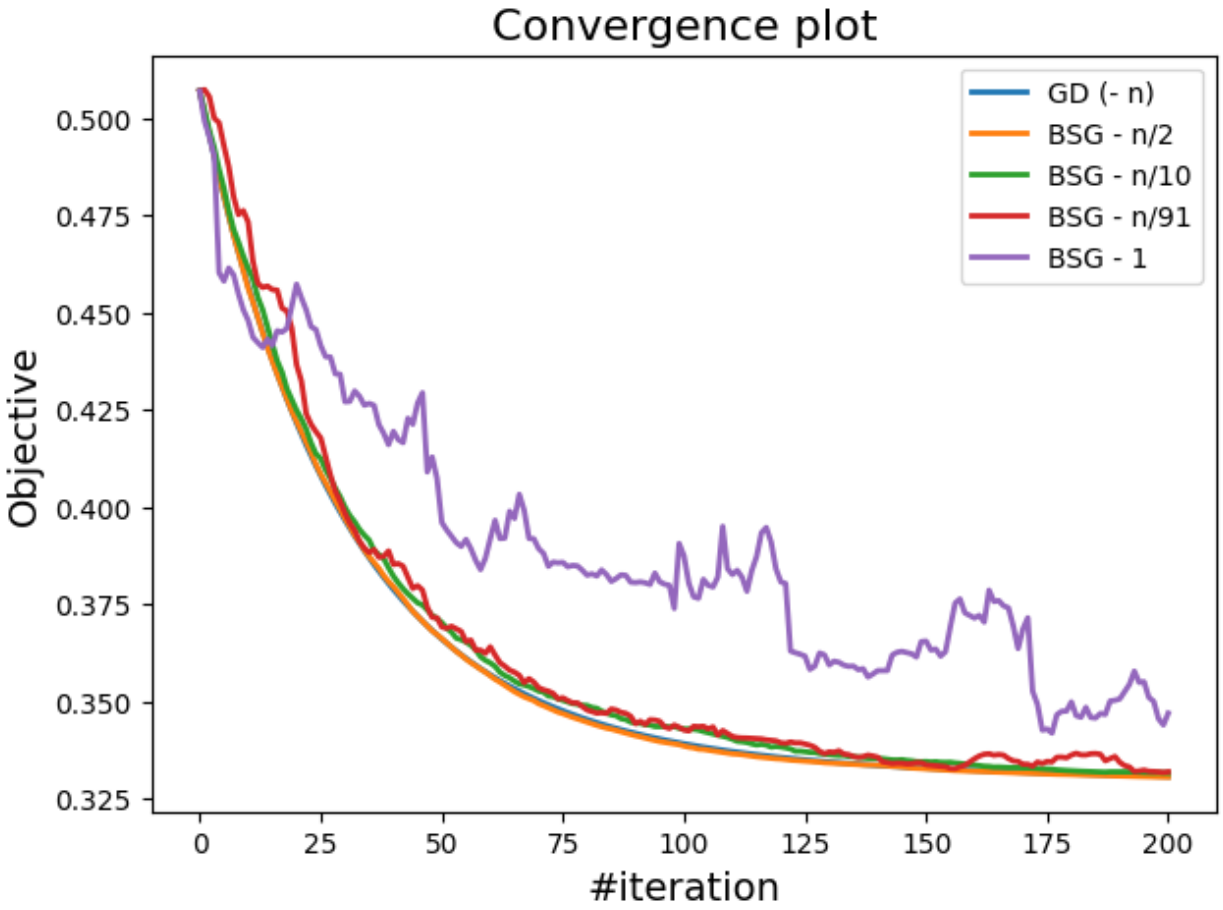


Figure 6: Convergence plot for different batch size

- n/2 perform well, but is still cotly in terms of data access. In compare with the lower batch, it is worth to have a lower batch as the result seems close.
- n/10 perform good too, and is really cheaper than GD for similar result on our dataset.
- n/91 as n/10 perform well, but with a little bit more oscilation. We can stop the study here, n/91 seems to be the maximum compromise, it could be a good batch size.

Finally, if we want the lower cost as possible, with similar result, n/91 is the best compromise. But if we want to have a lower cost with a convergence similar to gradient descent as possible, we go for n/10 or even n/2. It will depend on the priority to choose. As an averaging comprise between this 2 point of view, we can stand on n/10.

Question 3: Compare your stochastic gradient method with one of the advanced variants seen in class on your selected problem. Discuss your results, and what interpretation you can draw from them.

Personally, I was fascinating by the gradient with momentum technic, as it have this smart way to take in consideration the past vector for updating. So I chose to implement it, and compare it behavior to the classic stochastic gradient. As we can see on the above graph, SG and momentum

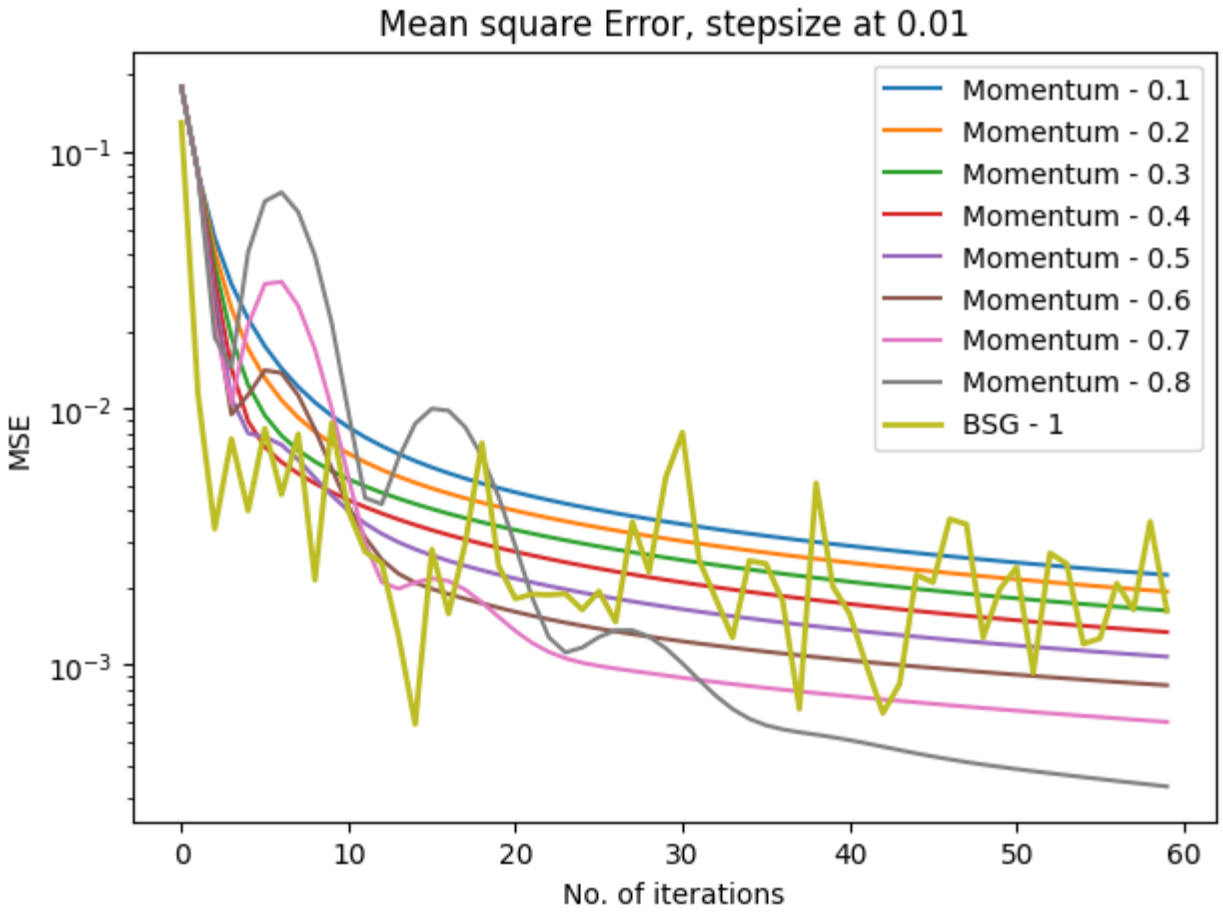


Figure 7: Convergence plot for SGD and Momentum

have close performance. But with the good momentum parameter, we can have better result, and the oscillation seems to disappear earlier on momentum. On our case, with a momentum parameter equal to 0.8 or 0.7 we have better result than the SGD.

Part 4 - Vincent Duval (Convexity and constrained optimization)

For this part, I generate my dataset by the convex function (to get an easiest case of study):

$$f(x) = \frac{1}{1 + \exp\left(-(x^2 + y^2)\right)}$$

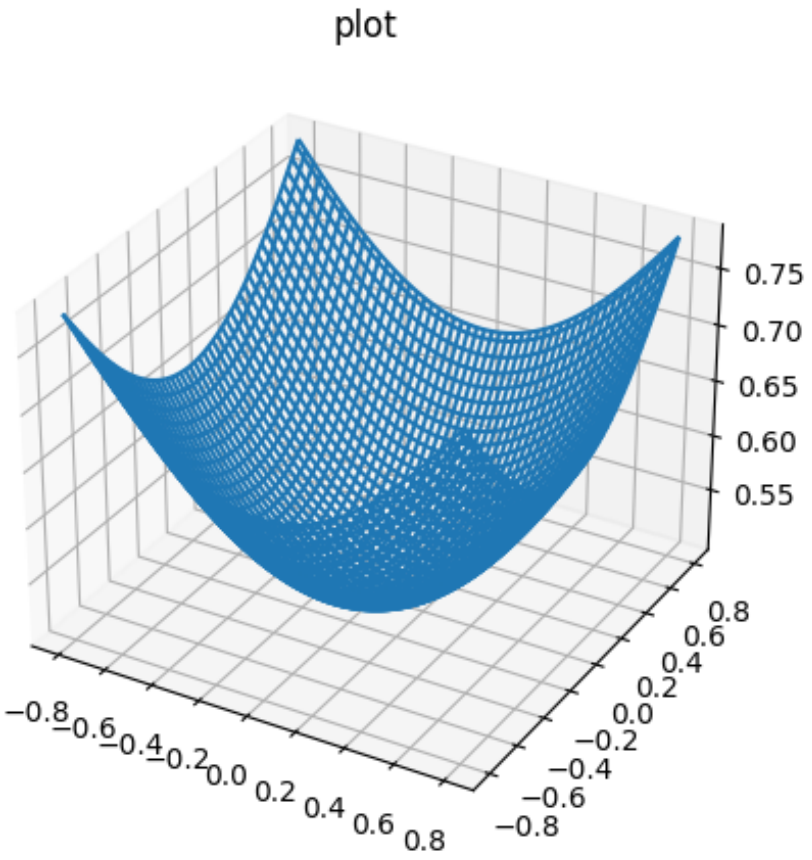


Figure 8: Convex function

Question 1: Implement the conditional gradient algorithm and the projected gradient algorithm. Compare their behavior (speed of convergence, structure of the iterates...) on a convex set for which you can implement both methods: the l-2 or l-infinity ball

Projected gradient algorithm

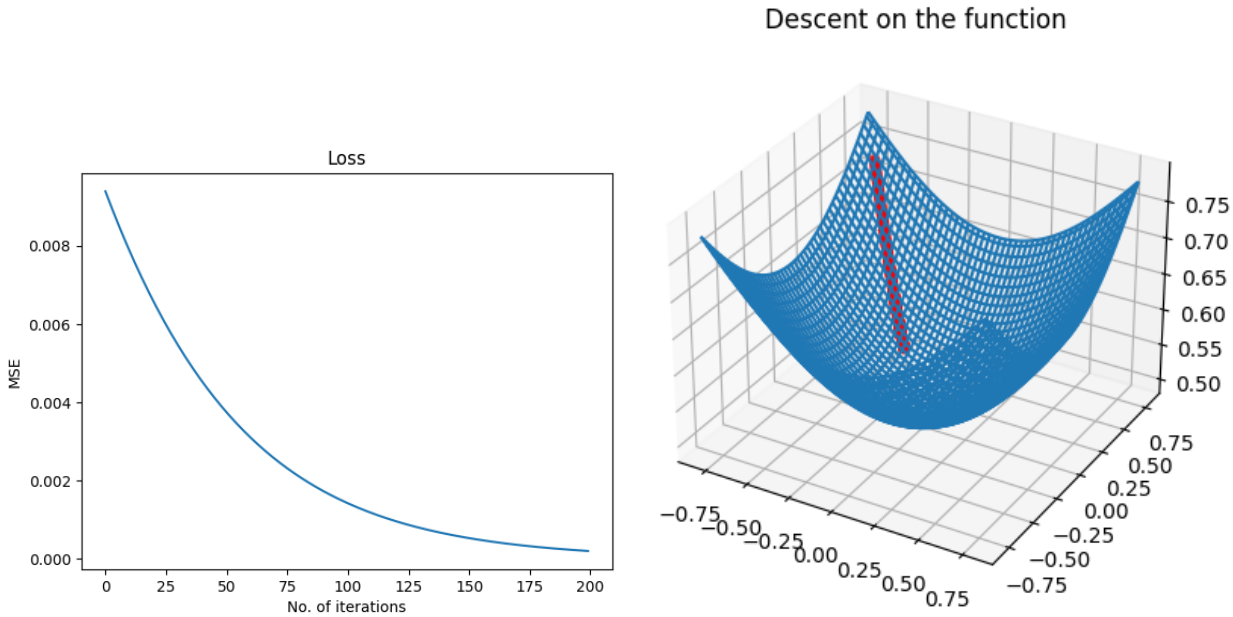
The projected gradient algorithm is an adaptation of gradient descent, using a projection onto the feasible set (A). Indeed, the iteration of projected gradient algorithm is given by:

$$\mathbf{x}_{k+1} \leftarrow Proj_A(\mathbf{x}_k - \alpha_k \nabla f(\mathbf{x}_k),$$

As I am using a closed and convex set here, the projection is well-defined and is an optimization problem itself. Using l-2 (Euclidian ball), the projection is given by :

$$Proj_A = \frac{R}{max(||x||_2, R)}x$$

After implementation on my problem, I got the following result :



(a) Loss plot

(b) Descent on the function

Figure 9: Projected gradient descent

Conditional gradient algorithm

For conditional gradient algorithm, we are minimizing the linearization, (instead of f over A as before).

$$s_{k+1} \in argmin_{s \in A}(f(x_k) + \nabla < f(x_k), s - x_k >)$$

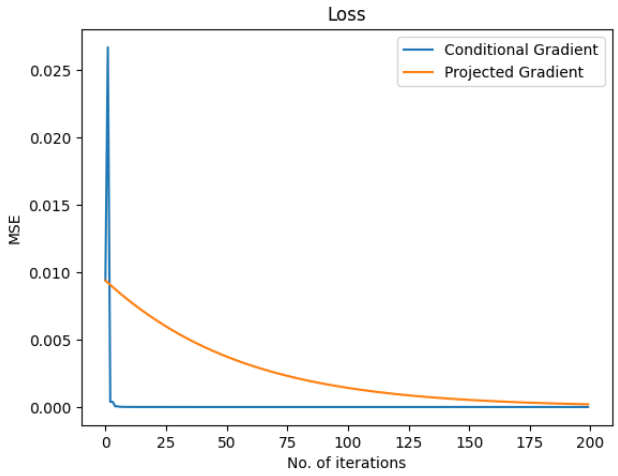
, instead of f over A as before. The iteration of conditional gradient is given by:

$$\mathbf{x}_{k+1} \leftarrow \theta_k s_{k+1} + (1 - \theta_k)x_k$$

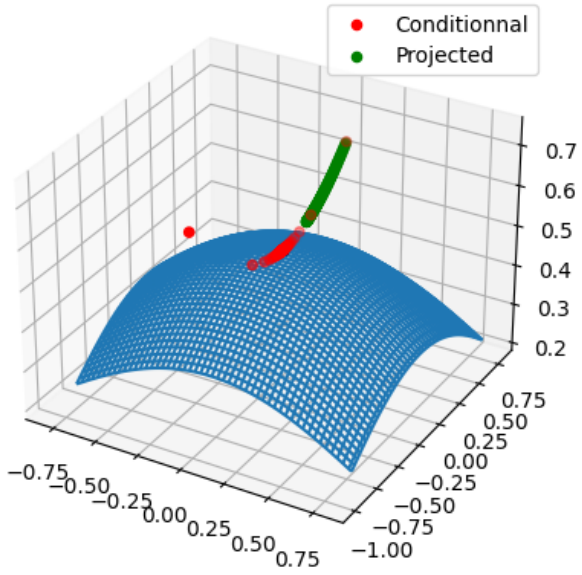
where $\theta_k = \frac{2}{k+2}$ (I choose prescripted stepsize).
In order to solve the linearization part, I used the l_{inf}-unit ball as the optimisation was easier on the extrem value :

$$\forall j \in [1; p], s_j = -(sign(\nabla f_j))$$

Here is the comparison between Conditional and Projected gradient descent optimisation: 10b
We can see 2 differents behaviors. Projected is going slow but surely to the optimal point and conditional take few step with a sort of oscilation around the optimal point. At the end, they are converging to the same point.



(a) Loss plot



(b) Descent in 3D with the inversed function (for visibility)

Figure 10: Projected vs Conditional

Question 2: the simplex (see the reference to Laurent Condat's paper in the notes, you may choose any method described therein), or the l1-ball, or some other exotic convex set of your choice (see the paper by Jaggi in the notes for some ideas)

I choose to implement the following method from the Laurent Condat's paper (page 3) :

Algorithm 1 (using sorting) [10]

1. Sort $\mathbf{y}$ into $\mathbf{u}$: $u_1 \geq \cdots \geq u_N$.
2. Set $K := \max_{1 \leq k \leq N} \{k \mid (\sum_{r=1}^{k} u_r - a)/k < u_k\}$.
3. Set $\tau := (\sum_{k=1}^{K} u_k - a)/K$.
4. For $n = 1, \ldots, N$, set $x_n := \max\{y_n - \tau, 0\}$.

Figure 11: Fast Projection onto the Simplex

After implementation in the PGD, I got this result 12. We can see that the convergence is similar than before, but there is a little pic at the biginning this time. (I cannot do the comparison with Coordinate because of probleme with implementation, on the replacment by the simplex projection.)

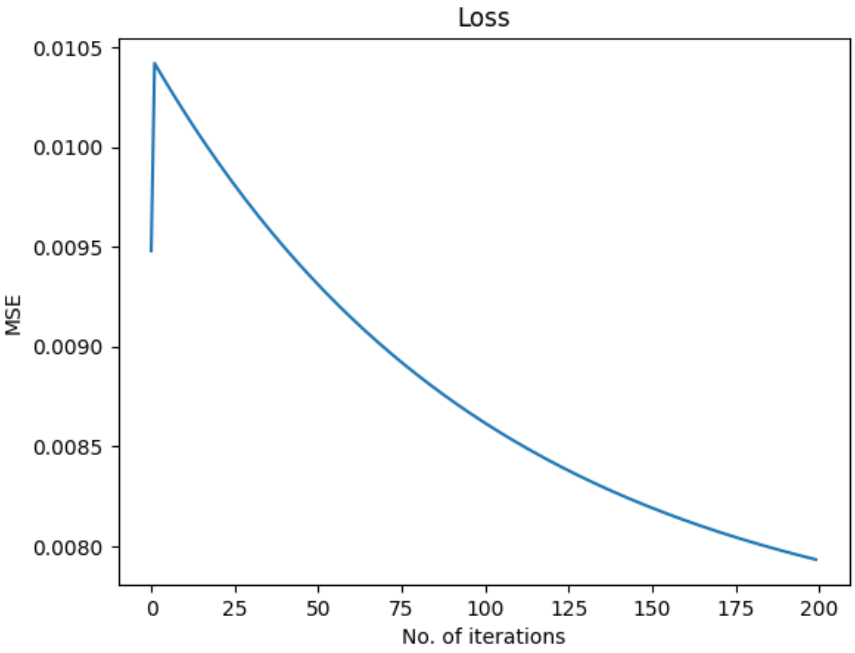


Figure 12: Fast Projection onto the Simplex

Part 5 - Clément Royer (Proximal gradient and LASSO)

Question 1: Add an l2 regularization term to your objective function from Part 1 or Part 3. Compare the solution of the unregularized problem to those obtained while solving the problem with a) a small value for the regularization parameter and b) a large value for the regularization parameter

This is my dataset from Kaggle, about red wine quality. As in part 1, we will perform a regression, but this time with a first form of regularization commonly called ridge regression problem. The problem con be write as follow:

$$\text{minimize}_{\mathbf{x}\in\mathbb{R}^d} f_{\ell_2}(\mathbf{x}) = \frac{1}{2n}\|\mathbf{A}\mathbf{x}-\mathbf{y}\|^2 + \frac{\lambda}{2}\|x\|^2,$$

The function f_{ℓ_2} is continuously differentiable, and its gradient is given by:

$$\nabla f_{\ell_2}(\mathbf{x}) = \frac{\mathbf{A}^T\mathbf{A}+\lambda n\mathbf{I}}{n}\mathbf{x} - \mathbf{A}^T\mathbf{y}.$$

To answer to the question, I run the gradient descent on the ridge regression, with different value of ridge parameter (λ), from 0 to 1 (for 0 we have an unregularized problem) :

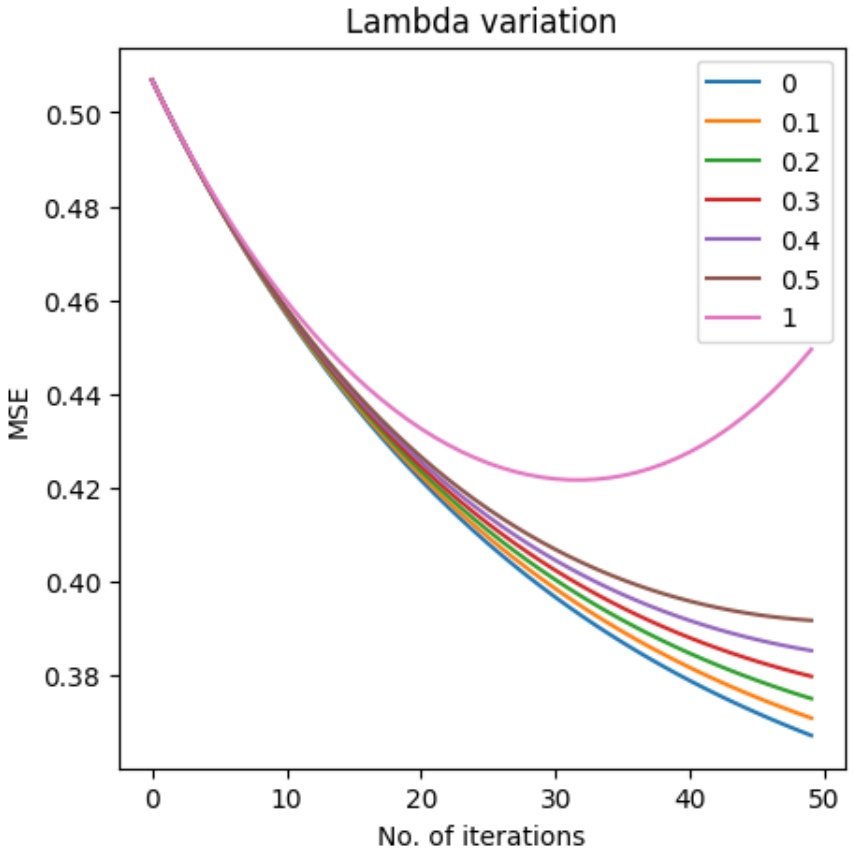


Figure 13: GD with different ridge parameter

On our problem, the results 13 shows us that we have a convergence for a small value of λ, but the best performing model is without the regularization, when $\lambda = 0$. For a big value, the algorithm

is even diverging. So finally, the best choice here is to solve the regression problem without the regularized term.

Question 2 : Add a l-1 regularization term to your objective function from Part 1 or Part 3 and solve the resulting problem. Can you find a value of the regularization parameter that yields a sparse solution? Does it provide a good value for the data-fitting term?

We now consider ℓ_1 regularization applied to the regression problem:

$$\text{minimize}_{\mathbf{x}\in\mathbb{R}^d} f_{\ell_1}(\mathbf{x}) = \frac{1}{2n}\|\mathbf{A}\mathbf{x}-\mathbf{y}\|^2 + \lambda\|\mathbf{x}\|_1,$$

l_1 penalty term make the objective function not differentiable, to deal with optimization we have to use the subgradients. The next iterate $\mathbf{x}_{k+1}$ is defined componentwise as follows:

$$\forall i = 1,\dots,d, \quad [\mathbf{x}_{k+1}]_i = \begin{cases} [\mathbf{g}_k]_i + \lambda\alpha_k & \text{if}[\mathbf{g}_k]_i < -\lambda\alpha_k \mathbf{g}_k > g_k \\ {}_i - \lambda\alpha_k & \text{if}[\mathbf{g}_k]_i > \lambda\alpha_k \\ 0 & \text{otherwise.} \end{cases}$$

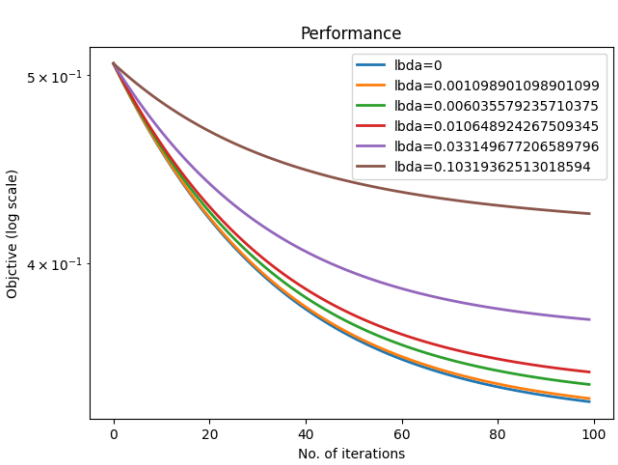
I tried different value of lambda define as follow $\lambda = \frac{1}{n^{\lambda'}}$, with $\lambda' = [1, 3/4, 2/3, 1/2, 1/3]$. Here 14a is the result with those value of λ. As we can see on the graph 14b14c, the l_1 regularization leads to sparser solution, moreover, as we increase the regularization parameter, the number of zero components increase (and only the largest features remain). Finally the sparser solution is given for the largest value of lambda, here is a summary of the results about sparcity (over 11 features): 14c

- Nonzero coefficients with lbda=0: 11
- Nonzero coefficients with lbda=0.001098901098901099: 11
- Nonzero coefficients with lbda=0.006035579235710375: 11
- Nonzero coefficients with lbda=0.010648924267509345: 11
- Nonzero coefficients with lbda=0.033149677206589796: 10
- Nonzero coefficients with lbda=0.10319362513018594: 7

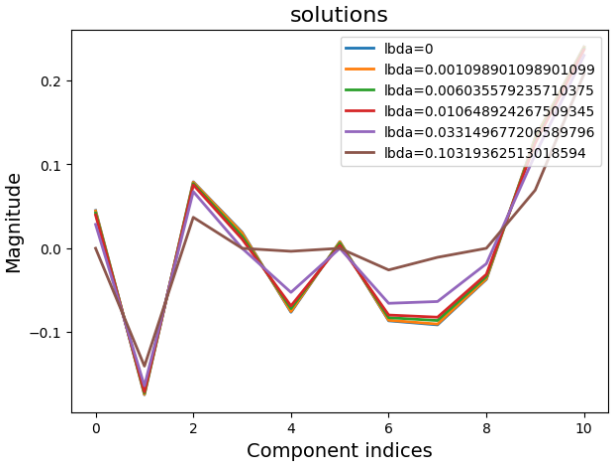
Finally, we can choose the larger value lbda=0.10319362513018594, as it lead to the sparser solution. However this solution doesn't provide the fastest convergence in our case as we can see in 14a.
For recall, the data fitting term is just the part without regularization, equivalent to $\lambda = 0$:

$$f(\mathbf{x}) = \frac{1}{2n}\|\mathbf{A}\mathbf{x}-\mathbf{y}\|^2$$

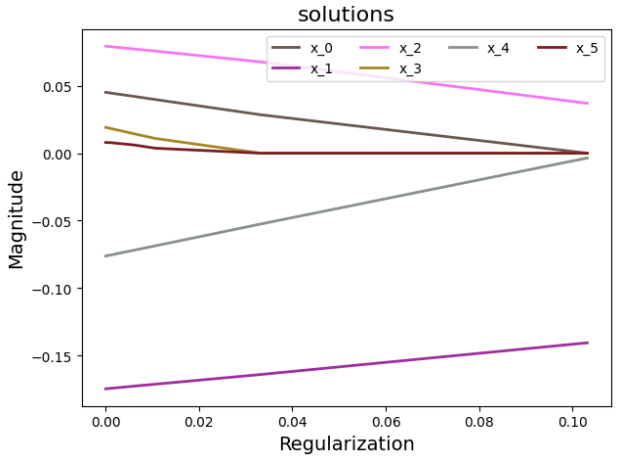
About the data-fitting term 14d, as the lambda is larger we have a datafitting term "less efficient" in term of optimisation than the others. But that make sense actually, because the data-fitting term is diluated by the reguralized term, that's why the data-fitting term is lower for unreguralized curve ($\lambda = 0$).



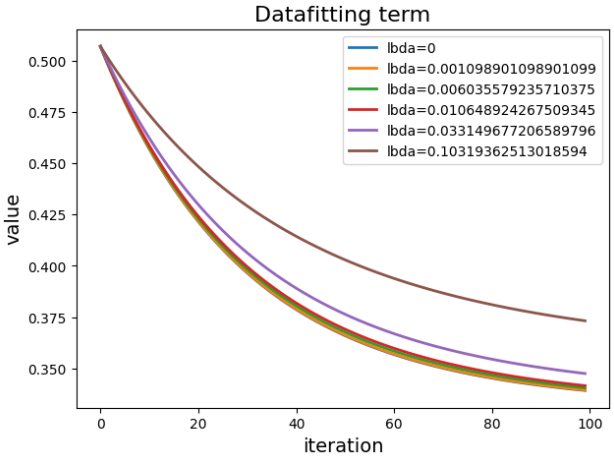
(a) Loss plot



(b) Vector solution magnitude for each feature



(c) Magnitude of the solution



(d) Data$_{f}ittingterm$

Figure 14: l-1 regularization

Part 6 - Clément Royer (Large-scale and distributed optimization)

This is my dataset from Kaggle, about red wine quality. As in part 1, we will perform a regression, but this time with block coordinate gradient descent.

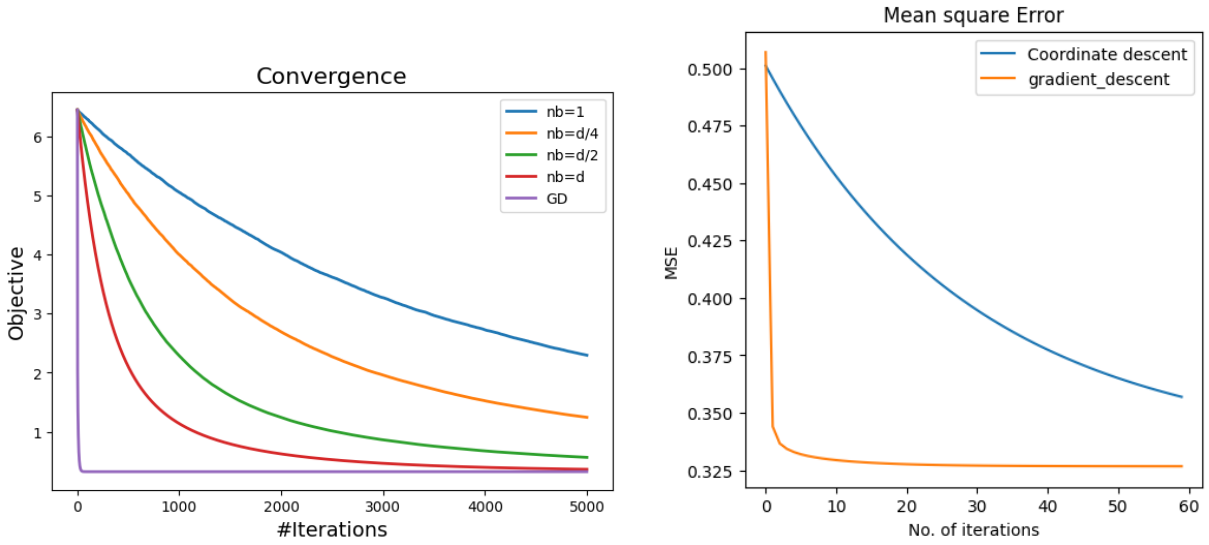
Question 1: Apply randomized block coordinate descent to your problem by considering a full batch of your dataset, and compare it with gradient descent. Is the use of coordinate descent beneficial on your problem?

We will apply a block-coordinate descent method to this problem. At every iteration, a block of coordinates $\mathcal{B}_k \subset \{1, \dots, d = 11\}$ is drawn (uniformly at random), and the new iterate is computed by solving:

$$\mathbf{x}_{j_{k+1}} = x_{j_k} - \frac{1}{L_{j_k}} * \nabla f_{j_k}$$

where L_{j_k} is a Lipschitz constant with respect to the j_kth coordinate of the batch. Here is a plot of the convergence to the solution for different block values :

$$[d = 11(fullbatch), d/2 = 6, d/4 = 3, 1]$$



(a) Coordinate GD and GD in terms of iteration

(b) Full batch coordinate and GD

Figure 15: Block Coordinate gradient descent

On my problem, the use of the coordinate descent is not really beneficial. For a full batch, the solution finally converges to the GD solution, but it is still slower and for a very large number of iteration.

Question 2: Combine randomized block coordinate descent with stochastic gradient (i.e. the method from Part 3). Do you observe a benefit from using coordinates together with stochastic gradient?

When we are combining coordinate (with full batch) and a stochastic gradient descent, the convergence seems faster. I plotted the result here :

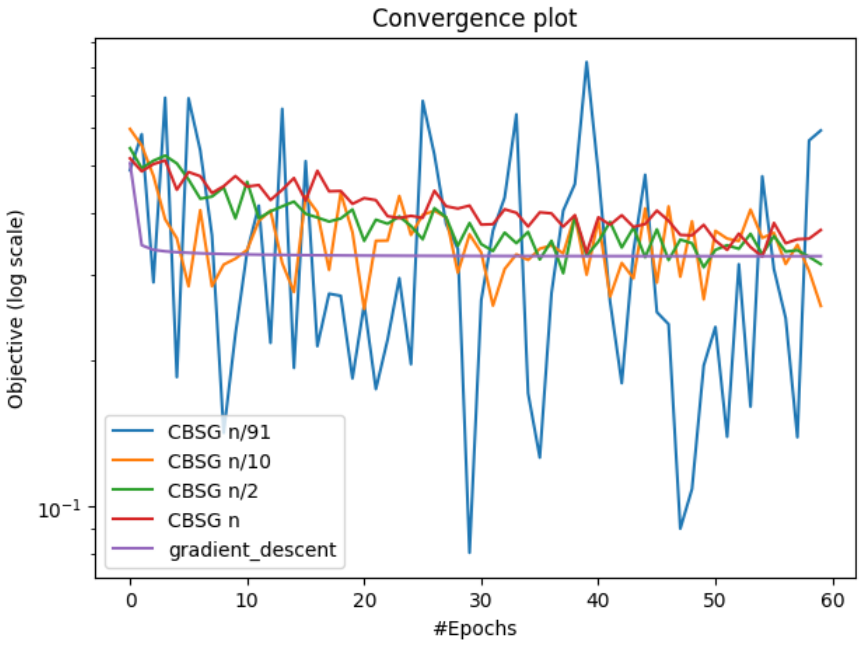


Figure 16: Stochastic Coordinate GD and GD

Naturally, the fact of using SGD brings oscillation in the results. But SGD is beneficial for the convergence, as it seems really faster than before. It allows us to do a compromise for example on the data access to get a lower cost with similar result. So we can conclude that we can observe a benefit from using coordinate together with stochastic gradient, because the convergence is better with lower cost.

Part 7 - Irène Waldspurger (advanced topics on gradient descent)

This is my dataset from Kaggle, about red wine quality. As in part 1, we will perform a regression (l2 loss), but this time with momentum.

Question 1: For the objective of Question 2, Part 1, implement Heavy Ball. Try several momentum parameters and stepsizes, and find the best ones.

The iteration of Heavy-Ball (also called Momentum) is given by:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \nabla f(\mathbf{x}_k) + \beta(x_k - x_{k-1})$$

We will tune the parameter β (momentum parameter) and α (step size), to find the best model.

Quick summary about the optimal step size from Part 1

The theory said that the optimal step size is lower than :

$$\tau \leq \tau_{\max} = \frac{2}{{AA^\top}_{op}}$$

where $\cdot_{op}$ is the maximum singular eigenvalue.
Besides, the optimal step size to minimize a quadratic function $< Cw, w >$ is

$$\tau_{\text{opt}} = \frac{2}{\sigma_{\min(C)} + \sigma_{\max(C)}}$$

where $\sigma_{\min(C)}, \sigma_{\max(C)}$ are respectively the minimum and maximum singular eigenvalue and $C = AA^T$. By calculation, $\tau_{\max} = 0.6801491488961905$ and $\tau_{\text{opt}} = 0.6675449160241176$ that's why 0.69 is diverging and 0.66 is converging so well on 2. To enforce the fact that 0.66 is the best solution, I plotted the log of the distance to the optimal value : $log(MSE(f(x)) - MSE(f^*(x))$

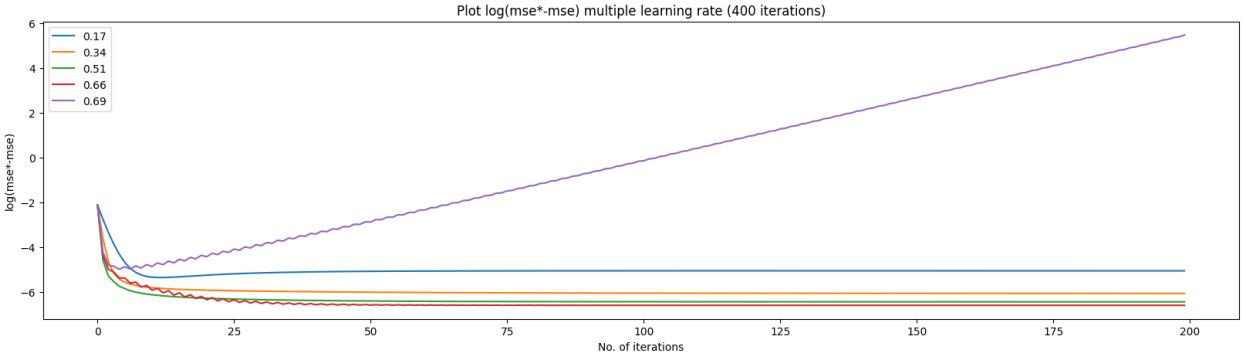
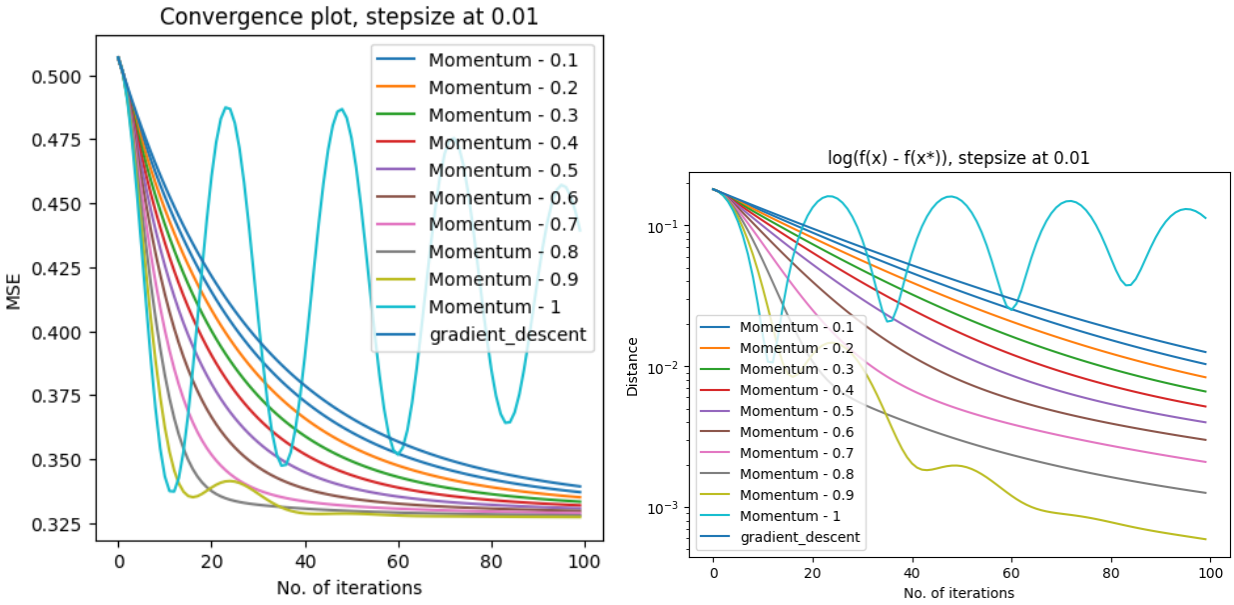


Figure 17: Distance to the optimal value for different step size

Here 2 we can check that 0.66 is closer to the optimal value than 0.5, as it is lower. We can conclude that the theory and the result are coherent here.

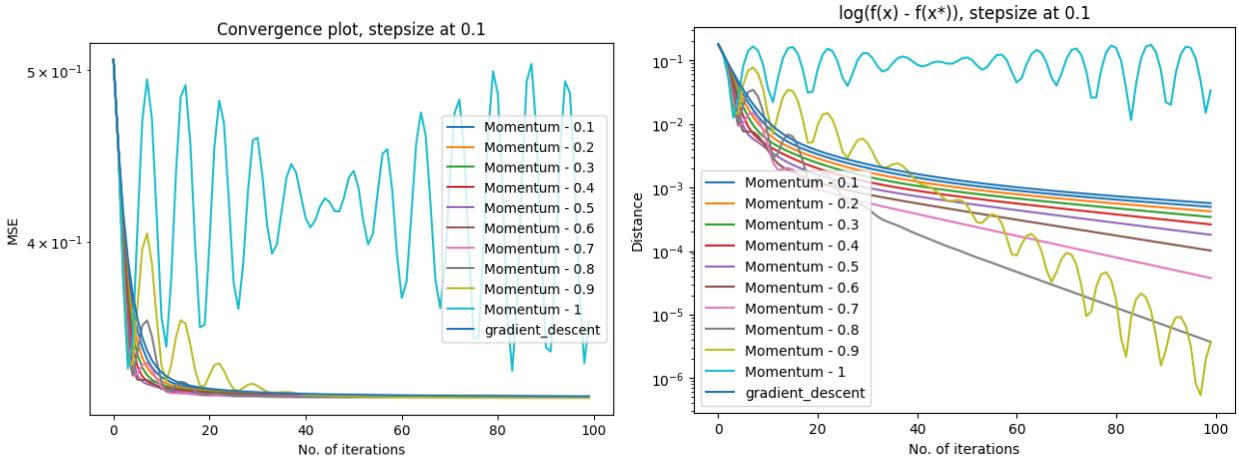
Now for Momentum

Let's see the momentum behavior for a fix step size. We will try some value commonly tried in my previous part [0.01, 0.1, 0.66] (0.66 was the best value in theory):



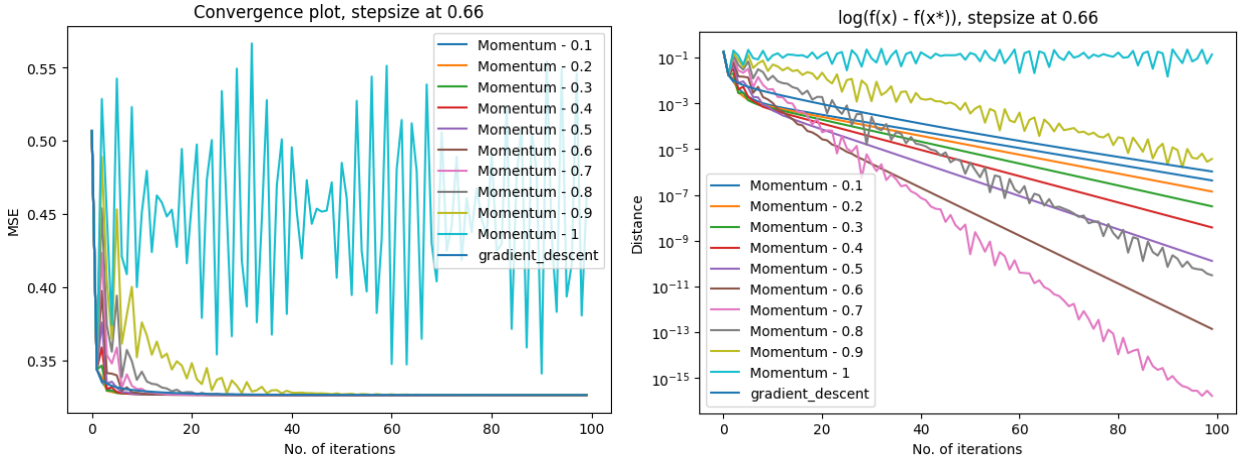
(a) Loss (b) Log of distance to optimal value

Figure 18: Momentum with stepsize at 0.01 and multiple β



(a) Loss (b) Log of distance to optimal value

Figure 19: Momentum with stepsize at 0.1 and multiple β



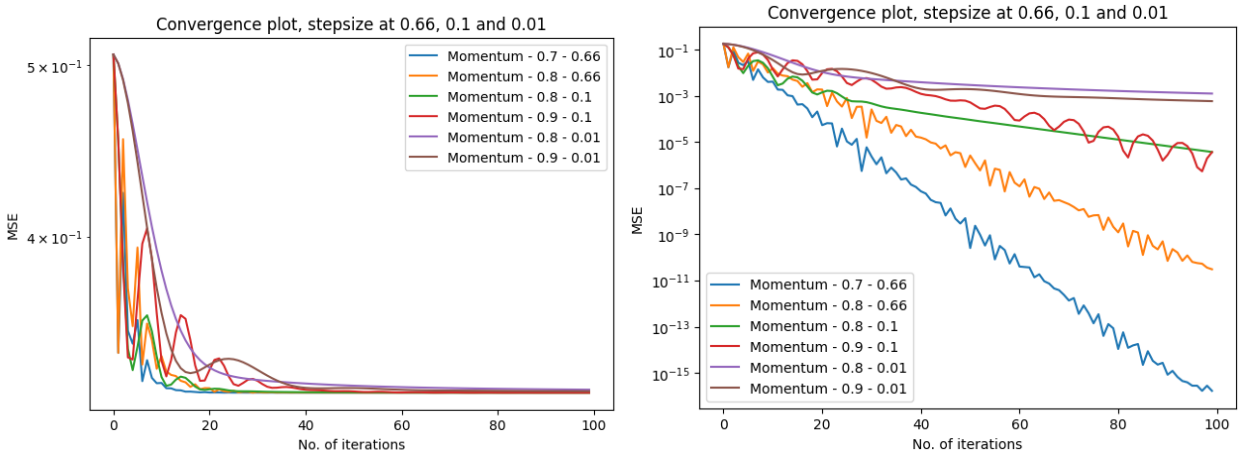
(a) Loss (b) Log of distance to optimal value

Figure 20: Momentum with stepsize at 0.66 (optimal) and multiple β

For each duo of graph, we have first the loss and then the log of the distance to the optimal value, to see which one is closer at the end and how they are taking the lead on eachother.

- for $\alpha = 0.01$ 18b : It seems that for a fix step size of 0.01, a value of $\beta = [0.8, 0.9]$ lead to the fastest convergence, and for the value 1 we are going to diverge.
- for $\alpha = 0.1$ 19b : Looking the second graph, the best result in terms of convergence to the optimal value is 0.9 (with a lot of oscillation) and 0.8 (without oscillation)
- for $\alpha = 0.66$ 20b (optimal step size in theory), looking the second graph, the best result in terms of convergence to the optimal value is 0.7 (with a lot of oscillation) and 0.6 (without the oscillation).

From this result we can make the assumption that the step size have an influence on the momentum parameter. When α go higher, β go lower. Now, that we have a set of potential optimal parameter, let's find the best overall by plotting them and compare the convergence performance :



(a) Loss

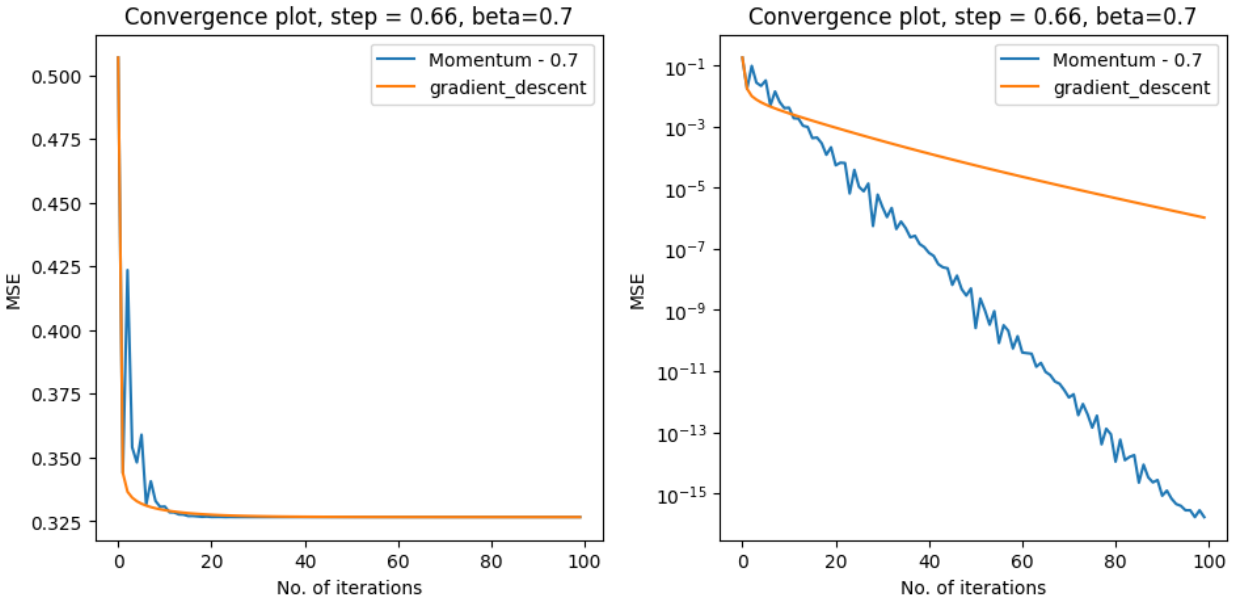
(b) Log of distance to optimal value

Figure 21: Momentum with the potential optimal parameters

There is some curve more smooth, with a slow convergence and others with a lot of oscilation but the congence is really faster. Finally, as an answer to the question, the best converging curve is the blue one, with the following parameters $\alpha = 0.66$ and $\beta = 0.7$.

Question 2: With these parameters, how does Heavy Ball compare with gradient descent?

Let's compare gradient descent with a step size of 0.66 and the momentum with the same step size and momentum parameter at 0.7.



(a) Loss

(b) Log of distance to optimal value

Figure 22: GD and Momentum with optimal parameters

We can see that at the beginning, GD is better performing and since iteration 15, the momentum is going closer to the optimal solution (with some oscillation). So for our problem, the momentum is better performing than simple gradient descent.

Question 3: Propose a non-convex loss function for your regression or classification problem (for instance, you can add a non-convex penalty to your convex objective, like the p-norm for $0 < p < 1$, or you can try prediction with a multilayer perceptron, as here). Try minimizing the loss with gradient descent, and check that you reach at least an approximate first-order approximate critical point. Do you get good prediction results?

I chose to implement a non-convex loss function for my regression (still the same problem) by adding a non-convex penalty term using the 0.5-norm. In order to check that I was reaching a first order optimal point, I put the following criteria : $norm(gradient)/norm(A) < 1e-5$ and got the following result :

- for lambda = 0.001098901098901099 at iteration 1515, first order optimal
- for lambda = 0.006035579235710375 at iteration 1503, first order optimal
- for lambda = 0.010648924267509345 at iteration 1622, first order optimal
- for lambda = 0.033149677206589796 at iteration 1174, first order optimal

About the convergence :

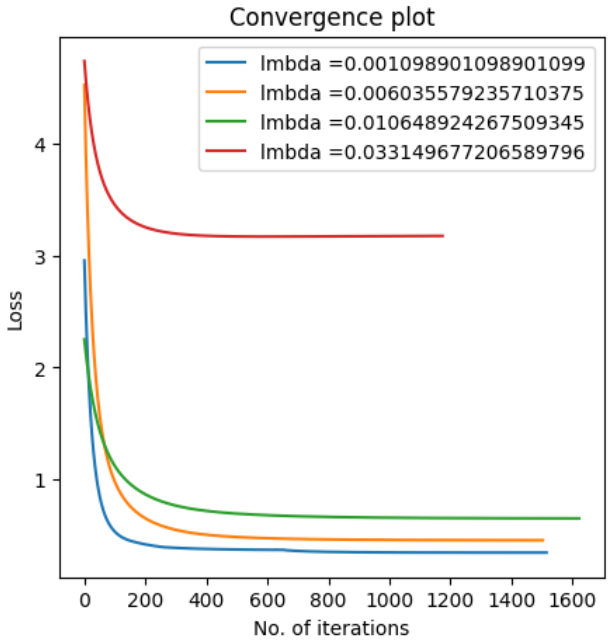


Figure 23: GD with non-convex loss