IASD Optimization Report

Matteo Barbieri

04/01/2021

Contents

Introduction

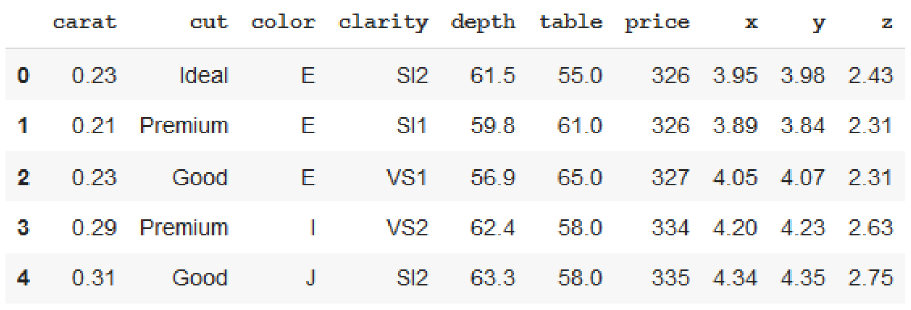
The aim of this project is to showcase different optimization methods to solve a minimization problem in the context of a Machine Learning setting.
We will start by explaining and implementing the *Stochastic Gradient Descent* algorithm and by playing around with the hyper-parameters, which include the learning rate and batch size. Later on, we may explore the different advances that have been propose throughout the years to increase the speed and accuracy for the specifics problems. Namely, we will explore a variant including *momentum* and the *Nesterov Accelerated Gradient Descent* algorithm. Finally, we will compare the results with the ones that the sklearn.linear_model package.

For the purpose stated above, we'll use in this report the "Diamonds" dataset from Kaggle - which you can download `https://www.kaggle.com/shivam2503/diamonds`. The code used to generate the following results will be available `https://github.com/17barbieri/IASD-optimization-project-2020-2021.git`.

1 Dataset and problem presentation

1.1 Dataset presentation

The *Diamonds* dataset we have at hand is a database containing the price and physical characteristics of 53920 diamonds. The layout of the data can be better seen in figure 1.



	carat	cut	color	clarity	depth	table	price	x	y	z
0	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
1	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
2	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31
3	0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63
4	0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75

Figure 1: First 5 lines of the diamonds dataset

The table contains 10 values for each diamond. The *carat* is a real variable representing the weight of the diamond in grams, the *cut* is a descrete variable that takes 5 values that represent how well the diamond has been cut. The *Clarity* is a descrete variable taking 8 values representing the clarity of the diamond, the *depth* is a real variable representing the total depth percentage $depth = \frac{z}{mean(x,y)} = \frac{2z}{x+y}$. The *table* is a real variable computed as the $\frac{width\ of\ top\ of\ diamond}{widest\ point}$ and x, y, z respectively represent the length, width and depth of the diamond.

At this point, the problem we'll use to expose the different optimization methods listed above is a *regression* problem. In other words, we'll try to predict the price of each diamond from the 9 other variables presented above. As a first move, we would like to use a linear model to carry out the prediction. As it is not only comfortable because of the properties of the function we minimize (see below), but it is also reasonable for this dataset. Indeed, figure 3 shows the distribution of the *price* and *carat* variables.

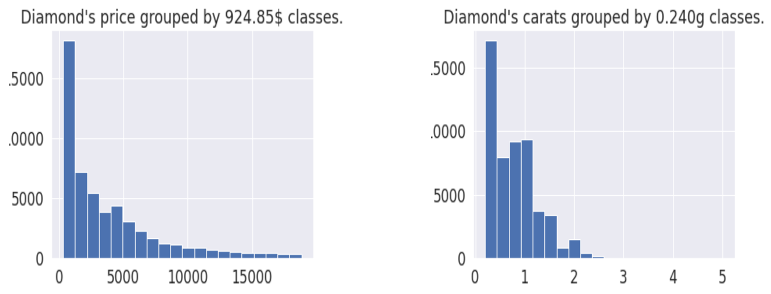
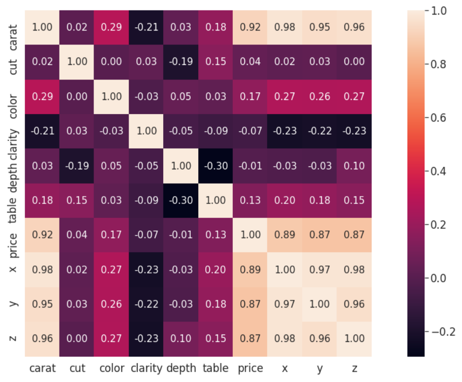


Figure 3: Comparison of the carat and price distributions in the *Diamonds* dataset.

Not only do these distributions look very similar at first sight, but the linear correlation coefficient shown in figure 4 shows a significant correlation. As a matter of fact, figure 4 shows that the variables *price, carat, x, y and z* are strongly correlated ($r^2 > 0.90$).



	carat	cut	color	clarity	depth	table	price	x	y	z
carat	1.00	0.02	0.29	-0.21	0.03	0.18	0.92	0.98	0.95	0.96
cut	0.02	1.00	0.00	0.03	-0.19	0.15	0.04	0.02	0.03	0.00
color	0.29	0.00	1.00	-0.03	0.05	0.03	0.17	0.27	0.26	0.27
clarity	-0.21	0.03	-0.03	1.00	-0.05	-0.09	-0.07	-0.23	-0.22	-0.23
depth	0.03	-0.19	0.05	-0.05	1.00	-0.30	-0.01	-0.03	-0.03	0.10
table	0.18	0.15	0.03	-0.09	-0.30	1.00	0.13	0.20	0.18	0.15
price	0.92	0.04	0.17	-0.07	-0.01	0.13	1.00	0.89	0.87	0.87
x	0.98	0.02	0.27	-0.23	-0.03	0.20	0.89	1.00	0.97	0.98
y	0.95	0.03	0.26	-0.22	-0.03	0.18	0.87	0.97	1.00	0.96
z	0.96	0.00	0.27	-0.23	0.10	0.15	0.87	0.98	0.96	1.00

Figure 4: Linear correlation matrix for the *Diamonds* dataset.

Dataset preparation Knowing this, we prepare the dataset for the Machine Learning problem. We start by checking for missing values and applying numerical values to the categorical variables. Finally, we split the dataset into a

Train, Val, Test and test splits with respectively 34 509, 8 627 and 10 784 data points.

1.2 Problem presentation

For the reasons detailed above, we're at first trying to solve a linear problem. For this purpose, we define the loss function as follows:

$$\mathcal{L}(X, w, Y) = \frac{1}{2} \cdot ||Xw - Y||^2 \tag{1}$$

$$= \frac{1}{2} \sum_{i=1}^{n} (x_i w - y_i)^2 \tag{2}$$

Where X is a matrix of shape $(n, 9)$ containing the 9 physical properties for each diamond, n the number of examples and Y a vector of shape $(n, 1)$ containing the example's label (y_i is the price diamond number i).

Note : As it is usually more comfortable to deal with a strongly convex problem (see below), we also define the ridge regularized loss function $\mathcal{L}_\lambda$ as follows:

$$\mathcal{L}(X, w, Y) = \frac{1}{2} \cdot ||Xw - Y||^2 + \frac{\lambda}{2} ||w||^2 \tag{3}$$

$$= \frac{1}{2} \sum_{i=1}^{n} (x_i w - y_i)^2 + \frac{\lambda}{2} ||w||^2 \tag{4}$$

Hence, the problem we need to solve to compute the best possible linear regression is:

$$w^* = \min_{w \in \mathbb{R}^9} \mathcal{L}(X, w, Y) \tag{5}$$

Or the ridge regularized problem :

$$w^* = \min_{w \in \mathbb{R}^9} \mathcal{L}_\lambda(X, w, Y) \tag{6}$$

2 Solving a linear regression problem

Solving a linear regression problem is one of the most basic applications of the computational optimization theory since there is an analytic solution. However, this will enable us to check that the expected convergence properties are met by comparing to the optimal solution.

2.1 Computation of the optimal solution

First, as $\mathcal{H}ess(\mathcal{L}) = X^T X \geq 0$ (*resp.* $\forall \epsilon > 0, \exists \lambda < \epsilon$ such as $\mathcal{H}ess(\mathcal{L}_\lambda) = X^T X + \lambda \mathbb{I}_n > 0$), the loss function $\mathcal{L}$ (*resp.* $\mathcal{L}_\lambda$) is convex (*resp.* strictly convex). Hence, let w^* be a minimizer of $\mathcal{L}$, then w^* exists and satisfies $\nabla \mathcal{L}(w^*) = 0$.

In our case, it appears that we can easily compute $\nabla\mathcal{L}$ and solve the above equation.

$$\nabla\mathcal{L}(X,w,Y) = X^T(Xw-Y) \tag{7}$$

$$\nabla\mathcal{L}_\lambda(X,w,Y) = X^T(Xw-Y)+\lambda w \tag{8}$$

$$\nabla\mathcal{L}(X,w,Y) = 0 \Longleftrightarrow w = (X^TX)^{-1}X^TY \tag{9}$$

$$\nabla\mathcal{L}_\lambda(X,w,Y) = 0 \Longleftrightarrow w = (X^TX+\lambda\mathbb{I}_n)^{-1}X^TY \tag{10}$$

As we have $n \gg p$, we will assume from here on that $X^TX \in GL_n(\mathbb{R})$. If this is not the case, then as $GL_n(\mathbb{R})$ is dense in $M_n(\mathbb{R})$, we can find λ as small as we want such that $X^TX+\lambda\mathbb{I}_n$ is invertible.

Finally, we have computed the exact solution to problem 5. We will use this exact solution to provide a comparison for the next computational methods.

2.2 Gradient Descent (GD)

2.2.1 Algorithm presentation

Gradient descent is a grounding method of computational optimization. The method and its improvements work very well on convex problems and even provide some comfortable convergence results.

Principle Let f be a function, and α a reasonably small real number, then the principle is to look for the minimizer of f in the direction of the steepest slope. Hence, we define x_n as follows :

$$\begin{cases} x_0 \in \mathbb{R} \\ X_{n+1} = x_n - \alpha\nabla f(x_n) \end{cases} \tag{11}$$

As the gradient is orthogonal to the function's level ligns and points in the direction of the higher values, each run consists in taking a small step in the opposite direction of the gradient.

Convergence analysis In the case of strongly convex functions (such as in equation (3) and maybe (1)), the Gradient descent method provides the following theoretical result if we use a learning rate $\alpha = \frac{1}{L}$:

$$\begin{cases} x_0 \in \mathbb{R} \\ x^* = arg\min_{x\in\mathbb{R}} f(x) \\ L = \max(eigenvalues(\mathcal{H}ess(f)) \\ \mu = \min(eigenvalues(\mathcal{H}ess(f)) \\ |f(x_t)-f(x^*)| \leq \frac{L}{2}(\frac{L-\mu}{L+\mu})^t \cdot ||x_0-x^*||^2 \end{cases} \tag{12}$$

Note that $\frac{L}{\mu}$ conditions the convergence rate of the algorithm. The close $\frac{L}{\mu}$ is to 1, the faster the descent.

Stochastic Gradient Descent (SGD) In practice, such a method may however be difficult to implement as it can put on hard trial the memory of the machines we're running the process on. Indeed, in our current case we need to store in memory the matrix X, of size (n, p) and the output w_t of size p. So the memory space is $\mathcal{O}(d+nd)$ which already limits our capacity to store the required information. Indeed, 50 000 with 9 coordinates are already too much. Hence, stochastic gradient descent (or batch gradient descent) have been developed to deal with this problem as it can be proven that by computing the gradient on a subset of the points at the time, the algorithm still converges towards the solution.

2.2.2 Practical results on SGD

To check that the aforementioned setup works, a batch-gradient-descent algorithm has been implemented to solve the regression problem for the *Diamonds* dataset. In the context of batch-gradient-descent, we redefine the loss function $\mathcal{L}_\lambda$ and its gradient $\nabla\mathcal{L}_\lambda$ as follows:

$$\begin{cases} \mathcal{L}_\lambda(X, w, Y) = \frac{1}{2n}\sum\limits_{i=0}^{n}(x_i w - y_i)^2 + \frac{\lambda}{2}||w||^2 \\ \nabla\mathcal{L}_\lambda(X, w, Y) = \frac{1}{n}\sum\limits_{i=0}^{n} x_i^T x_i w + \lambda w \end{cases} \tag{13}$$

Here, the division by n is meant to perfom the necessary averaging for SGD. In the case of SGD, n is equal to the total number of training examples, in case of batch-SGD, it is the batch size.

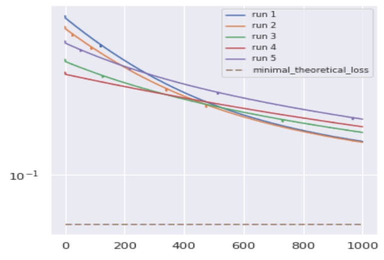
All we have left now, is to implement the SGD algorithm and run it to check the results following the pseudo-code available in algorithm (**??**).

Algorithm 1 Running SGD

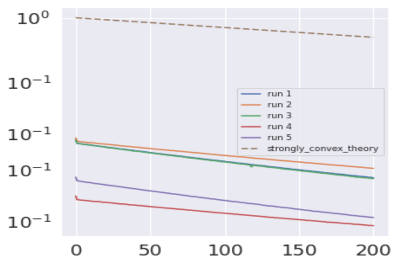
```
Require: $X_{train} \in M_{n\times p}(\mathbb{R}), Y_{train} \in M_{n\times 1}(\mathbb{R}), \lambda \in \mathbb{R}, \alpha \in \mathbb{R}, epochs \in \mathbb{N}$
  $w_0 \leftarrow random(\mathbb{R}^p)$
  for $t \leftarrow (1, 2, \ldots, epochs)$ do
    for $i \leftarrow (1, 2, \ldots, n)$ do
      $w_{t+1} = w_t - \alpha(\frac{1}{n}x_i^T(x_i w - y_i) + \frac{\lambda}{n}w)$
    end for
  end for
```

We now run the algorithm with 5 randomly picked initialization points and show the results for the loss curves in figure (6). Moreover, as the problem can be solved analytically, we also show the convergence rate of the algorithm and compare it with the theoretical convergence rate (see equation (12)) in figure (6).

As expected, we notice in figure 6.b) that the algorithm eventually has a similar convergence rate as the one presented in equation (12). However, figure 6.a) gives us a better sense of how slow this convergence rate is. Indeed, the condition number of this experiment is of about 2593, very far away from 1... Hence, we need to figure out a way to increase the convergence of the algorithm.



(a) SGD loss plot



(b) Comparison of SGD with the theoretical convergence rate (in dotted line).

Figure 6: Results for SGD with batch size = 512

We'll first try to toggle the hyper-parameters before improving the algorithm itself.

Playing with the hyper-parameters With this setup, we will play with two hyper-parameters that are the *learning rate* and the *batch size*. Let's try to model the impact of these parameters on the resolution of the problem. We start by selecting different learning rates in 3/L, 2/L, 1/L, 1/(2L), 1/(3L) before analyzing a broader range in [0.1, 1e-5]. The results are shown below in figures (8) and (10). *Note: In the following experiments, the initialization point remains unchanged to allow a better comparison accros the parameters.*

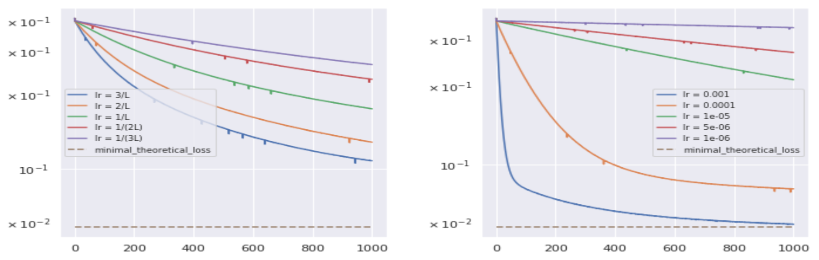


Figure 8: Loss plot of SGD for different learning rates

As we can see from this plots, the learning rate of 1/L ($\sim 1, 1e-5$) is quite suboptimal for the convergence time, and a higher value (like lr = 0.01) enables us to reach a lower loss much faster. However, we notice in figure 10.b) that the algorithm becomes unstable after around 150 epochs of training, which is probably because the learning rate becomes too large with respect to the distance $||x_t - x^*||_2$. One would probably gain from reducing the learning rate at that point to continue the learning, but this will not be explored here. **We**

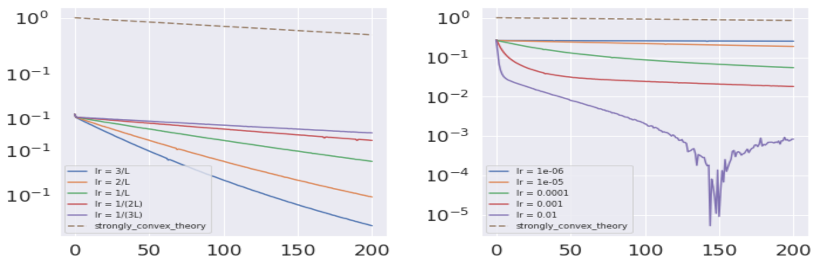


Figure 10: Convergence plot of SGD for different learning rates

will however use in the following part a *learning rate* equal to 1/L to ensure the benefit of the convergence theorem.

Now, we can look at the impact of the *batch size* by selecting it in the range of 1, 8, 16, 128, 512. Loss and convergence results are reported in figures

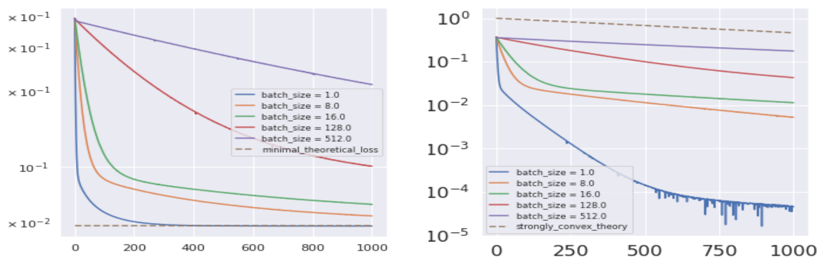


Figure 12: Convergence and loss plots of SGD for different batch sizes.

We can see in the figure (12) that the smaller the batch size is, the faster the convergence. However, we also note two other things. Firstly, the time required to compute the run increases when the batch size decreases. Secondly, the convergence rate becomes unstable around 750 epochs, which is again the sign that the learning rate is too large with respect to the distance $||x_t - x^*||_2$. **In spite of the precedent results, which show that a small batch size accelerates convergence, we will continue using a batch size of 512 to provide an easier comparison among all the presented results.**

2.2.3 Improving SGD

In this section, we'll explore different improvements that have been made throughout the years to the SGD method. We will start from the momentum-SGD and go on with Nesterov accelerated Gradient Descent.

Momentum Gradient Descent This method is inspired form the observation that Gradient Descent may often follow local slope changes or get stuck in local minima (not the case here because $\mathcal{L}_\lambda$ is strongly convex...). Hence, the sequence x_n defined in equation 11 now includes the momentum of the last iteration. The updated version can be read below:

$$\begin{cases} x_0 \in \mathbb{R} \\ m \in [0,1] \\ X_{t+1} = x_t - m(x_t - x_{t-1}) - \alpha \nabla f(x_t) \end{cases} \tag{14}$$

The results for our problem can be seen in figure (14). As we can see, adding a momentum in our current setup does not improve the convergence of the algorithm. Indeed, we have already stated that this technique is most useful in settings were the loss function is not strongly convex and could hence display local minima where the algorithm could get stuck.

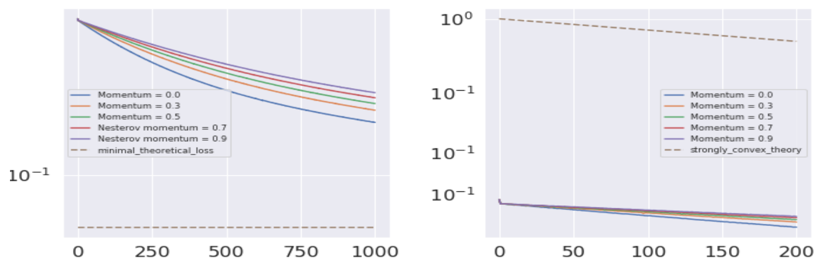


Figure 14: Convergence and loss plots of SGD for different momentums.

Nesterov Accelerated Gradient Descent We know that we will use our momentum term $m(x_t - x_{t-1}$ to move the parameters x. Computing $x_t - m(x_t - x_{t-1})$ thus gives us an approximation of the next position of the parameters, a rough idea where our parameters are going to be. We can now effectively look ahead by calculating the gradient not w.r.t. to our current parameters x but w.r.t. the approximate future position of our parameters. Hence, the update runs as follows:

$$\begin{cases} x_0 \in \mathbb{R} \\ m \in [0,1] \\ X_{t+1} = x_t - m(x_t - x_{t-1}) - \alpha \nabla f(x_t - m(x_t - xt - 1)) \end{cases} \tag{15}$$

As we can now see from figure (15), the results are now undoubtedly better, in the sense that we approach the theoretical minimal loss way faster with the nesterof Gradient Descent and a high momentum than without.

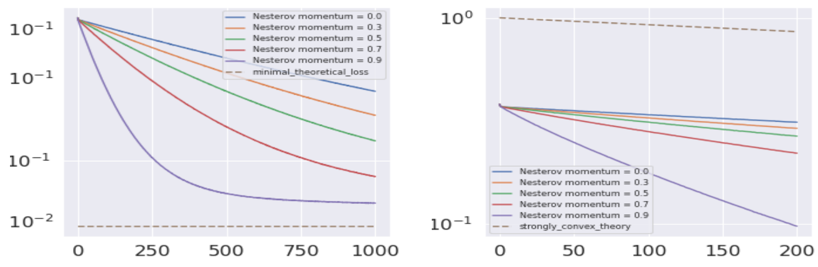


Figure 16: Convergence and loss plots of SGD for different Nesterov momentums.

3 Training and predicting

Now that we have presented in section 2 the different methods and parameters that we can choose to solve the optimization problem, it is time to evaluate the performance on the test set. For this, an early stopping process with a patience of 10 epochs has been implemented. Namely, we check the performance of the regression on a validation set that has been held out from the training and we stop whenever the validation loss doesn't improve for longer than the patience.
The parameters that we take into account are at the count of 5 and are the *learning rate*, the *batch size*, the *momentum*, the *Nesterov acceleration variant*, the *Ridge regression parameter*.

According to the projection in figure (17), the best validation score is reached for the parameters' values (lr = 1e-2, Nesterov = False, Batch size = 64, Momentum = 0, Ridge regularization parameter = 0). The validation loss is of 0.05785 and the loss on the test set is of 0.05478.

Parameter	Values
Learning rate	[1e-2, 1e-3, 1e-4, 1e-5, 1e-6]
Nesterov acceleration variant	[False, True]
Batch size	[64, 128, 512]
Momentum	[0, 0.3, 0.5, 0.7, 0.9]
Ridge regularization parameter	[0, 025, 0.05, 0.075, 0.1]

Table 1: Hyper-parameters analyzed by cross-validation

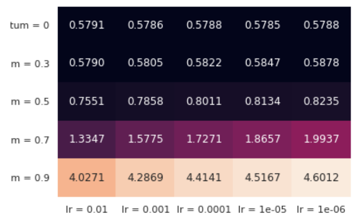


Figure 17: Results matrix (scaled $\times 10)projectedoverthecoordinatesNesterov =' False', batch\ size =' 64', andRidge\ regulrization\ parameter = 0$.

Finally, scikit-learn linear regression tool has been run as well to provide a comparison with our model. The results provided are shown in table 2.

	Test loss	Test RMSE	Test R^2
scikit-LinearRegression	0.07284	0.3817	0.9249
SGD-LinearRegression	0.05478	0.3310	0.9441

Table 2: Comparison of the scikit-learn linear regression model and the self-implemented SGD method.