Optimization Project - Master IASD Dauphine

Matthieu Rolland

january 2022

Note: The goal of this project is to compare and optimize **hand coded** descent methods to obtain the best and fastest results (NO EXTERNAL PYTHON LIBRARIES ALLOWED)

Contents

1 Setting and definitions

1.1 Introduction

In this project, we aim to perform optimization analysis of various descent algorithms using a Dataset found on http://archive.ics.uci.edu/ml/datasets/Superconductivty+Data (the original data is public).

Our goal is to predict the critical temperature of some superconductors with a data set composed of 81 features extracted from 21263 measure points.

To do so, we propose to model the critical temperature using a polynomial regression of degree 3.

Disclaimer: Even though the model choice is not very suited for this particular complex data set, what counts here is the comparison of various descent methods. So you should not expect incredible prediction results.

1.2 Our approach in the notebook

In the notebook, we will analyse several classic descent method and try to find the best hyper parameters for each one of them in terms of convergence rate and accuracy.

The classic descent methods we will go through includes:

- Gradient Descent
- Stochastic Gradient Descent
- Batch gradient descent
- SGD with Momentum
- Nesterov Accelerated Gradient

Then, we will try some more sophisticated descent methods as part of the optional content that should be provided for this project. The sophisticated descent methods we will go through includes:

- Adagrad
- Adam

In the end we will test the benefits of regularization and display the results of the model.

2 Presentation of the Problem

2.1 Polynomial regression Model details

A polynomial regression is a form of regression analysis in which the relationship between the independent variable x and the dependent variable y is modelled as an nth degree polynomial in x. Polynomial regression fits a nonlinear relationship. To get an intuition, in dimension 2 with a polynomial regression of degree 2, the target labels can be expressed as a combination of features and product of features (interaction features) such as:

$$\hat{y} = w_0 + w_1 z_1 + w_2 z_2 + w_3 z_1 z_2 + w_4 z_1^2 + w_5 z_2^2$$

where the w_i are the coefficient of our model, and z_i the different features of our dataset. A polynomial regression can be viewed as a special case of Linear regression where the vectors of feature containing the z_i and interactions of the z_i are stored in another vector x, where:

$$x_1 = z_1, x_2 = z_2, x_3 = z_1 z_2, x_4 = z_1^2, x_5 = z_2^2$$

therefore we can write the whole problem as a linear one

$$\hat{y} = [1, x_1, x_2, x_3, x_4, x_5]w \quad \text{or in matrix notation} \quad \hat{Y} = Xw$$

2.2 Objective function

Since we are performing a regression, we choose to optimize the half of the Mean Squared Error (MSE loss) as objective function. We also add an l_2 regularization associated with an adjustment parameter λ to restrain the set of solutions to a ball. The problem, also known as ridge regression, can be written as per the following expression:

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

with

- n is the number of samples in the dataset.

- m is the number of total features after polynomial features transformation

- $w = [w_0, \dots, w_m]^T$ the weight vector of size $\mathbb{R}^{m+1}$, where w_0 is the bias and $w_1, \dots, w_m$ are the weights.

- $Y = [y_1, y_2, \dots, y_n]^T$ the label vector of size $\mathbb{R}^n$, where every y_i is the critical temperature of sample i.

- $X = [1, x_1, x_2, \dots, x_m]^T$ of size $\mathbb{R}^{m+1}$The transformed feature matrix with all binomial feature coefficients

- $\hat{Y} = Xw = [\hat{y}_1, \hat{y}_2, \dots, \hat{y}_n]^\top$ the vector of predictions of size $\mathbb{R}^n$

This problem has the form of a finite sum $f(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} f_i(\mathbf{w})$, where

$$f_i(\mathbf{w}) = \frac{1}{2}(\mathbf{x}_i^T\mathbf{w} - y_i)^2 + \frac{\lambda}{2}\|\mathbf{w}\|^2.$$

This objective function is convex meaning a minimum exists. In addition,

- $\nabla f_i(\mathbf{w}) = (\mathbf{x}_i^T\mathbf{w} - y_i)\mathbf{x}_i + \lambda\mathbf{w}$ for every i

- $\nabla f(\mathbf{w}) = \frac{1}{n}\mathbf{X}^T(\mathbf{X}\mathbf{w} - \mathbf{y}) + \lambda\mathbf{w}$

- ∇f is $L = \frac{\|\mathbf{X}^T\mathbf{X}\|}{n} + \lambda$ is Lipschitz continuous.

2.3 Data pre processing

Knowing we want top perform a polynomial regression using the Mean squared error as criterion, we perform the following pre-processing steps:

- Get rid of outliers: A direct consequence of using the mean squared error will be that this loss will try it's best to fit the extreme data points (because the square function grows exponentially with the distance). Therefore we chose to drop any sample that differ from more than 6 standard deviations (we prefer to stay cautious and chose not less than 6 since we don't know anything on the underlying distribution).
- Make a PCA: Since our data has a lot of linear correlation in between its features, we choose to make a PCA to simplify the dimensions and be able to handle all the interactions features of a polynomial of degree 3. With only 25 PCA components, we are able to explain 99.9% of the variance in the data.
- Normalize the Data: It is good practice to scale all the data into [0;1] interval so that features with larger numerical values are not given more attention.
- Train/Test/Val Splits: In order to train, test and allows some hyperparameter tuning we proceed to split our dataset in 3 distinct sets (20% test, 20% val and 60% training).

3 Descent methods Analysis and Results

3.1 (Ordinary linear least squares closed form solution)

We still seek to minimize the MSE of our model written as:

$$L(w) = \frac{1}{2n}\|Xw - Y\|_2^2 + \frac{\lambda}{2}||w||_2^2$$

In order to solve this optimization problem, we are going to proceed by calculating the gradient of the loss with respect to the weight vector w, and then solve the equation $\nabla_w L = 0$, to find the optimal weight vector w^*, for which the minimum loss value is reached.

The expression of the gradient is:

$$\nabla_w L = \frac{1}{n}X^T(Xw - Y) + \lambda w$$

Therefore, the optimal weight vector w^* expression is:

$$\nabla_w L = 0 \quad \Leftrightarrow \quad \boxed{w^* = \left(X^T X + \lambda n I\right)^{-1} X^T Y}$$

The reason we use descent methods instead of the Ordinary least squares closed form solution, is because of the complexity of this solution (inverting a big matrix) can grow exponentially with the number of samples n, and the number of features m.

To get an intuition, let's compute the complexity of the non regularized closed form solution, by analyzing how many operations we are performing, and the complexity of each operation.

$$\boxed{w^* = \left(X^T X\right)^{-1} X^T Y}$$

In this expression, we are performing the following operations:

- X^T the transpose of a matrix is performed two times $\mathcal{O}(2n(m))$.

- $X^T X$ is a matrix multiplication where $X \in \mathbb{R}^{n,m}$ $\mathcal{O}(n(m)^2)$.

- $\left(X^T X\right)^{-1}$ is the inverse of a matrix of size $\mathbb{R}^{m,m}$ $\mathcal{O}((m)^3)$.

- $X^T Y$ is a matrix multiplication where $X \in \mathbb{R}^{n,m}$, $Y \in \mathbb{R}^n$ $\mathcal{O}(n(m))$.

- The matrix multiplication between $\left(X^T X\right)^{-1}$ of size $\mathbb{R}^{m,m}$, and $X^T Y$ of size $\mathbb{R}^m$ has a complexity of $\mathcal{O}((m)^2)$.

By suming all these complexities we have:

$$\begin{aligned} TotalComplexity &= \mathcal{O}(2nm + nm^2 + m^3 + nm + m^2) \\ &= \mathcal{O}(m^3 + m^2 + nm^2 + 3nm) \\ &\approx \mathcal{O}(m^2(n + m)) \end{aligned}$$

We can see that the complexity can easily grow with the number of samples n, and the number of features m, and if we have $m = n$, the complexity becomes $\mathcal{O}(m^3)$.

3.2 Batch Gradient Descents

Batch Gradient Descent is a famous algorithm we use to solve optimization problems, especially for minimizing loss function such as $L(w)$ when there exist no closed form solutions or when this latter is too complex to compute. The algorithm starts by initializing random weights to the vector w, and updates it in the opposite direction of the gradient $\nabla_w L$, with a quantity called the learning rate τ. This latter mainly controls the speed of convergence and the final accuracy of the solution. The BGD calculate the gradient using a batch of data ranging from 1 sample to the full data set (n samples). Many possibilities exists to pick the indices of the samples to be part of the gradient computation (i_k) but the most simple way of doing so is by random uniform sampling. With k the index of current iteration, the algo can be summarized by:

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

Where τ_k is the learning rate, and S_k the set (batch) of randomly drawn indices.

The different Batch Descent methods are essentially using the same descent algorithm but this latter can be split in distinct variant according to the batch size used:

- Vanilla Gradient descent: batch size = n (whole data set)
- Batch Gradient descent: $1 <$ batch size $< n$ (mini-batch $\simeq 1$, Big batch $\simeq n$,...)
- Stochastic Gradient descent: batch size = 1 (1 sample)

The batch size used will in fact provide completely different behaviour in terms of convergence rate and associated guarantees.

3.2.1 Vanilla Gradient Descent (VGD) and acceleration

The vanilla gradient descent is a very popular and basic descent algorithm using the whole data set every iteration. We can therefore rewrite the BGD algorithm as:

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

Speaking in terms of epochs, this variant of Batch GD makes an iteration per epoch. The perks of this variant are:

- A more direct path is taken towards the minimum (full gradient)
- Stable convergence
- Knowing our objective function is convex and that we are able to compute the Lipschitz constant of our training set, by choosing wisely the learning rate ($\tau = \frac{1}{L}$) we are able to have guarantees in terms of convergence rate, in our case $\mathcal{O}(\frac{1}{k})$ ($\mathcal{O}((.)^k)$ if μ convex)

The cons of this variant are:

- Can be very costly on big data sets with many samples
- Can converge at local minima and saddle points
- Will suffer from redundancies in the data

In order to get good result with the VGD method, a proper learning rate should be chosen in order to ensure a good convergence rate as well as a good end accuracy (see Figure 1)

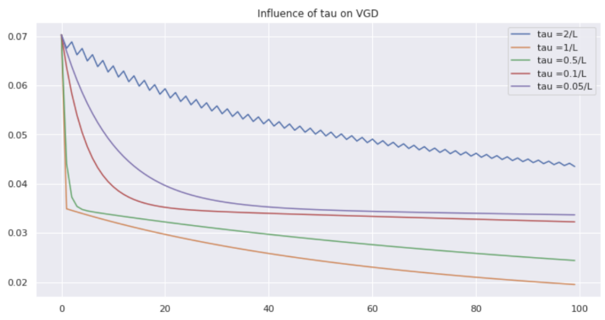

Figure 1: Influence of learning rate on VGD

From this graph we can easily see that the optimal learning rate is 1 over the lipschitz constant of the training dataset. Therefore we will keep with his value for full gradient descent methods. (see results of performances in Figure 5)

Nesterov Accelerated VGD:

An accelerated version of the VGD is by adding momentum to it in a particular way. Momentum algorithm is just like a ball following blindly the slope, since it doesn't know when to slow down. This algorithm will apply the same steps as the Momentum, however it will apply the gradient to the next approximate position of the weights, instead of the actual position like it was the case in classic Momentum algos.

$$\boxed{w^{(k+1)} = w^{(k)} - \tau \nabla_w L\left(w^{(k)} + \beta(w^{(k)} - w^{(k-1)})\right) + \beta(w^{(k)} - w^{(k-1)})}$$

knowing our objective function is convex and that we are able to compute the Lipschitz constant of our training set, by taking the learning rate $\tau = \frac{1}{L}$ we are able to have guarantees in terms of convergence rate, in our case $\mathcal{O}(\frac{1}{k^2})$ which is much better than VGD. (see results of performances in Figure 5)

3.2.2 Batch Gradient Descent (BGD)

The batch gradient descent is a very popular and basic descent algorithm using a pre fixed or variable batch size through its iterations. The perks of the batch variant are:

- in mini batch setting: Computationally efficient since batch size very close to 1 (still allows to gather more gradient info per iteration)
- Convergence is more stable than Stochastic Gradient Descent (less oscillations)
- allows parallel computing

The cons of this variant are:

- will suffer from redundancies in the data

In order to get good result with the BGD method, a proper batch size should be chosen in order to ensure a good convergence rate as well as a stable convergence (see Figure 2)

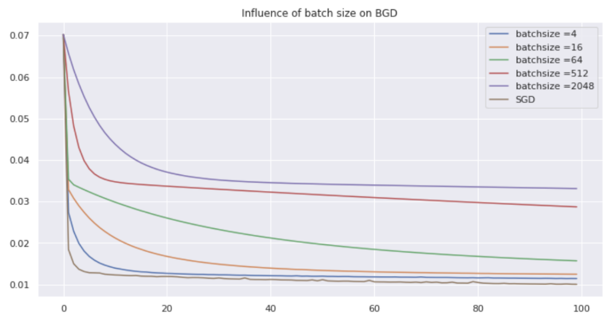

Figure 2: Influence of batch size on BGD, tau=0.01

From this graph we can easily see that the optimal mini batch of size 4 is optimal. In fact, it is very close to the convergence rate of the SGD while not suffering from its oscillations due to a finer approximation of the gradient at each step. Therefore we will keep with his value for the rest of the project. (see results of performances in Figure 5)

3.2.3 Stochastic Gradient Descent (SGD) and acceleration

The Stochastic gradient descent is one of the most popular descent algorithm using only one randomly drawn sample at every iteration. Speaking in terms of epochs, this variant makes n iterations per epoch (n times more than VGD). The perks of this variant are:

- Very computationally efficient
- The frequent updates create plenty of oscillations which can be helpful for getting out of local minimums
- Does not suffer from redundancies in the data
- can outperform vanilla gradient descent if there is enough correlation in the data

The cons of this variant are:

- Must rely on the assumption that the samples are from coming from a distribution for efficient guarantees
- Can suffer from big oscillations

In order to get good result with the SGD method, a proper learning rate should be chosen in order to ensure a good convergence rate as well as a stable convergence (see Figure 3)

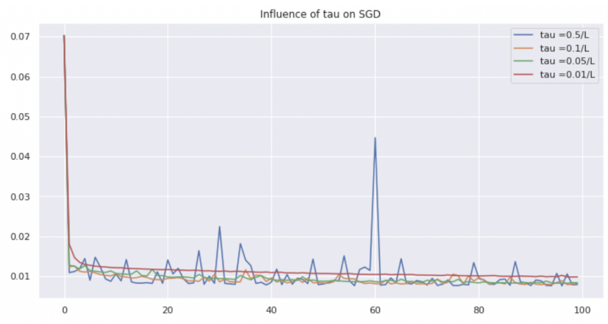

Figure 3: Influence of learning rate on SGD

From this graph we can easily see that the optimal learning rate is 0.01 over the lipschitz constant of the training dataset. Therefore we will keep with his value for the other gradient descent methods. (see results of performances in Figure 5)

SGD with momentum:

An accelerated version of the SGD is by adding momentum to it. The classic SGD algorithm has trouble navigating through ravines (areas where the surface curves much more steeply in one dimension than in another), which are common around local optima. In these scenarios, SGD oscillates across the slopes of the ravine while only making hesitant progress along the bottom towards the local optimum. To address this problem, we introduce a new term called the momentum β, and the velocity vector v that we are adding to the gradient vector, in order to make an update of the weights as follows:

$$\boxed{\begin{aligned} &v^{(k+1)} = \beta v^{(k)} + \tau \nabla_w L\left(w^{(k)}\right) \\ &w^{(k+1)} = w^{(k)} - v^{(k+1)} \end{aligned}}$$

The intuition of Momentum is a heavy ball rolling down the hill. The added momentum term acts both as a smoother and an accelerator, dampening oscillations and causing us to barrel through narrow valleys, small humps and local minima, which SGD fails to do.

In order to get good result with this method, a proper beta and learning rate should be chosen in order to ensure a good convergence rate as well as a good end accuracy and low oscillations(see Figure 4)

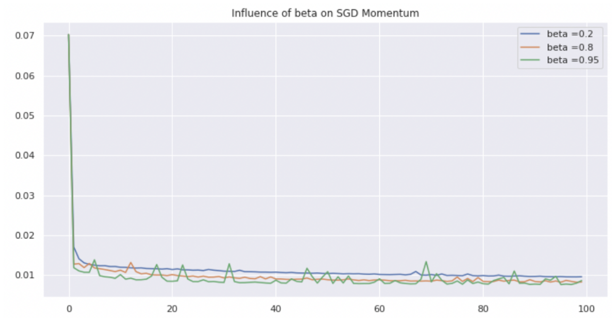

Figure 4: Influence of beta on SGDM

From this graph we can easily see that the optimal beta parameter is aroud 0.8, therefore we will keep with his value. (see results of performances in Figure 5)

3.2.4 Adaptive learning rate descent methods

In order to try the effects of a variable learning rate we chose to implement very famous methods of it.

Adagrad:

Adagrad is a descent variant that will adapt the learning rate τ all along the gradient descent process, by dividing each weight on a quantity based on the sum of the previous squared gradient up to time t. Therefore, the update will be large for infrequent data samples, and small for frequent ones, which makes it a perfectly suited algorithm for sparse data.

$$\boxed{w^{(k+1)} = w^{(k)} - \frac{\tau}{\sqrt{G_k + \varepsilon}} \odot \nabla_w L\left(w^{(k)}\right)}$$

where,

- G_k is diagonal matrix containing in each diagonal the sum of the squared gradient of parameter w_k, up until time t.

- ε is a parameter that we add to avoid division by zero

- $\odot$ is the element-wise product of matrices.

(see results of performances in Figure 5)

Adam:

Adam is also an adaptative learning rate algorithm just like Adagrad, it starts by computing the exponentially decaying average of past gradients (first moment estimation of gradient), and the exponentially decaying average of past squared gradients (second moment estimation of gradient). After that, we make a correction of the previously computed moments since they are biased towards zero, and we use the resulting corrections to update the weights.

$$\boxed{m_k = \beta_1 m_{k-1} + (1-\beta_1)\nabla_w L\left(w^{(k)}\right) \qquad v_k = \beta_2 v_{k-1} + (1-\beta_2)\nabla_w L\left(w^{(k)}\right)^2}$$

$$\boxed{\hat{m}_k = \frac{m_k}{1-\beta_1} \qquad \hat{v}_k = \frac{v_k}{1-\beta_2} \qquad w^{(k+1)} = w^{(k)} - \frac{\tau}{\sqrt{\hat{v}_k}+\varepsilon}\hat{m}_k}$$

where, m_k and v_k are respectively the first and the second moment estimation of the gradient and $\hat{m_k}$ and $\hat{v_k}$ the bias-corrected versions of it.

To get the best possible performances with this method, we chose a mini batch size of 4, β_1=0.9 and β_2=0.99. (see results of performances in Figure 5)

3.3 Global results

Now that finally tested many methods to obtain the best test accuracies on our testing data, we can clearly see that our implementation of SGDM is the one performing the best among the others on this dataset with this polynomial model of order 3. (see Figure 5)

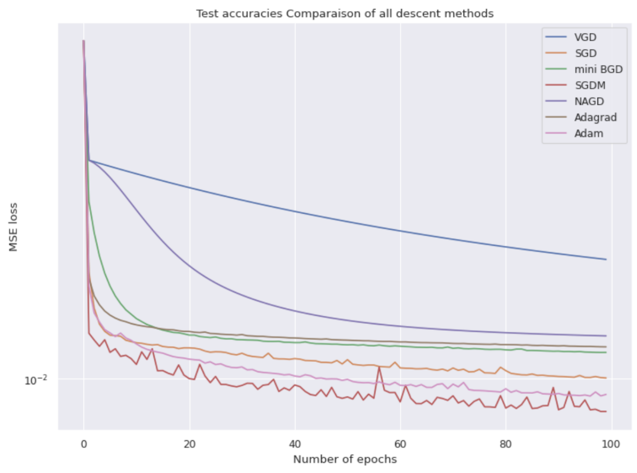

Figure 5: CV rate of all tested descent methods

The fact that the stochastic methods outperform the full gradient methods tells us that there is a lot of correlation within our data.

3.4 Improving results with regularization

A regularization is basically adding a penalty over the total value of your weights in the objective function. This penalty can take the form of different norms and has many useful effects:

- It helps fighting over-fitting by penalizing big slopes coefficients leading to high variance predictions
- It performs feature selection by minimizing the weights given to features non correlated with the labels
- in certain cases it can help convexify the objective function by restricting the feature space to a specific area where the function is convex.

In this project we made the choice to use the ℓ_2 regularized polynomial regression aka ridge regression. We now use our validation dataset to find the best λ

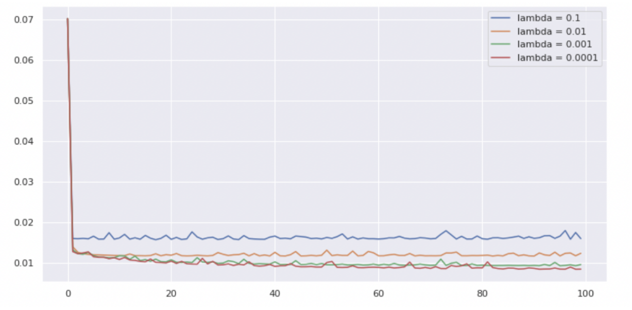

Figure 6: Influence of λ on VGD

In Figure 6 we can identify that 0.0001 is the best lambda value. This makes sense since at this phase of the fitting process we are still in clear underfit and therefore regularization will only have effect on slowing down the fitting. Nevertheless if we were to go deeper into the epochs, we would converge to a sweet spot where the best lambda would allow us to get the maximum test accuracy possible.

Finally, we were able to get a view of our best model predictions (SGDM) in Figure 7

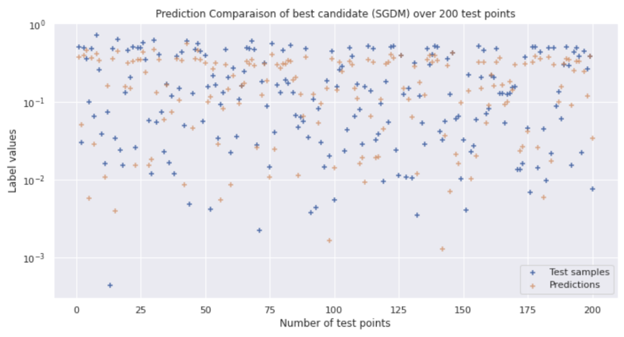

Figure 7: Prediction Comparison of best candidate (SGDM) over 200 test points

4 Conclusion

In this work we could explore the various descent methods and some optimization analysis in order to improve the global convergence rate and accuracy of optimal solutions. Our best candidate was a Stochastic gradient descent with Momentum which is a very powerful when there is enough correlation in the data. Finally to obtain even better results we would try other models rather than making a more complex polynomial regression to model the critical temperature of superconductors.