Project: Optimization for Machine Learning

Authors:
Thai Ba Tuan

Jan 10th, 2022

Contents

1 Problem and Dataset

1.1 Dataset

In this project I use the Diamond dataset it is an publish dataset in Kaggle. This dataset contains the prices and the attributes of almost 54,000 diamonds. There are 10 attributes included in the dataset including the target ie.

- Price: price of Diamond in US Dollar
- Carat: The diamond physical weight
- Cut and Color, Clarity: determining the quality of the cut, the color of gem-quality diamonds occurs in many hues and a measurement of how clear the diamond is. There features is categories
- x, y, z, depth, table: length, width, depth and depth percentage of diamond. The depth is calculate by: $\frac{2z}{x+y}$ and the table is width of top of diamond relative to widest point.

1.2 Processing Data

Based on the features of diamond, we will predict the price of this diamond, that lead to the problem of this project is "Linear Regression".

Figure 1 show us the related between each pair of features, we can see that "price" is linear with the feature x, y, z,... so we can see the "regression" is reasonable for this dataset. In deed, we can see some outliers point that can lead to the noise in this problem. So we can pre processing the data by removing the outliers data, and then Standard the dataset to make it more easier to compute

The dataset will be split in to training and testing set with ratio is 0.25.

1.3 problem

$$\text{minimize}_{\mathbf{w}\in\mathbb{R}^d} f(\mathbf{w}) := \tfrac{1}{2}\|\mathbf{X}\mathbf{w}-\mathbf{y}\|^2 \tag{1}$$

the Gradient of f is:

$$\nabla f(w) = A^\top(\mathbf{A}\mathbf{w}-\mathbf{y}) \tag{2}$$

this problem can solve directly by calculating the solution by using Moore-Penrose pseudo-inverse $w = (X^\top X)^{-1}X^\top y$.

Some information about this function is L-Lipschitz continuous and L can be calculated by

$$L = \frac{2}{\|AA^T\|_2^2} \tag{3}$$

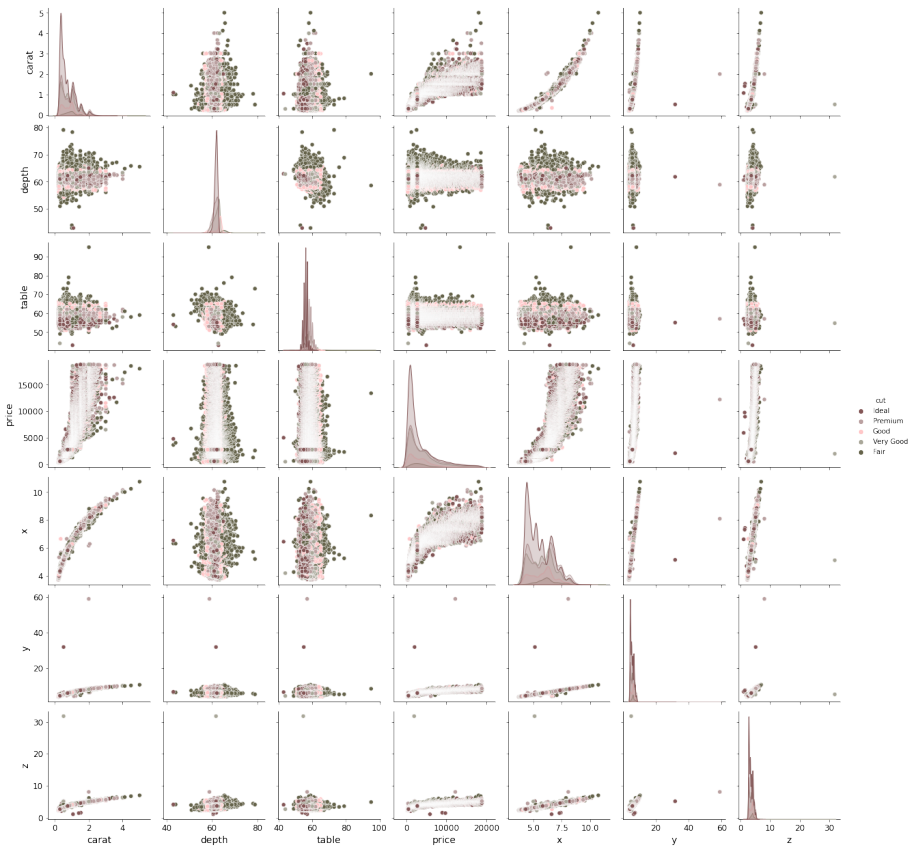


Figure 1: Pair plot data

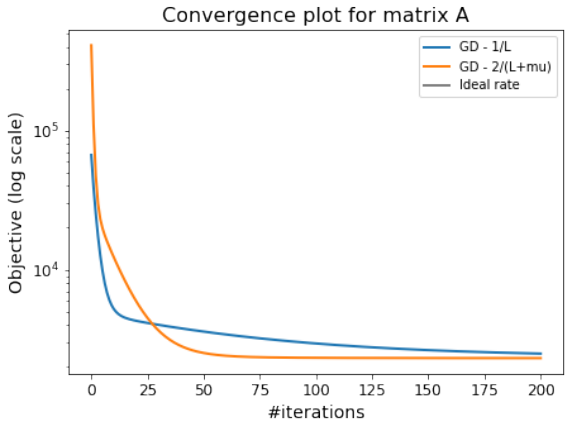


Figure 2: Convergence rate in log scale Gradient Descent

2 Mandatory Algorithms

2.1 Gradient Descent

Gradient descent is a popular method that use in computational optimization.

Let f be a function, and a reasonably small real number (step size), the algorithm iterate until convergence by using the loop:

$$\boldsymbol{w_{n+1}} \leftarrow \boldsymbol{w_n} - \alpha \nabla f(\boldsymbol{w_n})$$

the step size α is bound by $2/L$ we L is calculate by the formula above. the optimal step size is $\tau_{\text{opt}} = \frac{2}{\sigma_{\min}(C)+\sigma_{\max}(C)}$ where $\sigma_{\min}(C), \sigma_{\max}(C)$ is max and min eigen values of Hessian matrix of f $(\nabla^2 f(x) = X^T X)$

In the testing, I will test the gradient descent with difference value of step size i.e: $\frac{1}{L}$ and $\frac{2}{(L+\mu)}$ the figure below show the convergence of this method:

2.2 Batch Gradient Descent with Nesterov

Nesterov Accelerated Gradient Descent using momemtun when compute the gradient

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \alpha_k \nabla f\left(\boldsymbol{w}_k + \beta_k \left(\boldsymbol{w}_k - \boldsymbol{w}_{k-1}\right)\right) + \beta_k \left(\boldsymbol{w}_k - \boldsymbol{w}_{k-1}\right)$$

the step size α and momemtum β can constant or descent in the practice. In Nesterov, the momemtum can be caculate by the iteration below:

$$\begin{cases} \boldsymbol{w}_{k+1} = \boldsymbol{z}_k - \alpha_k \nabla f\left(\boldsymbol{z}_k\right) \\ \boldsymbol{z}_{k+1} = \boldsymbol{w}_{k+1} + \beta_{k+1}\left(\boldsymbol{w}_{k+1} - \boldsymbol{w}_k\right) \end{cases}$$

where α_k is step size parrameters and β_k is calculated by:

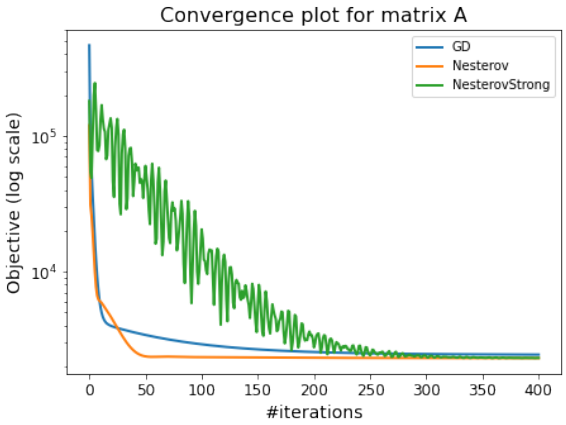


Figure 3: Convergence rate in log scale Nesterov

$$t_{k+1} = \frac{1}{2}\left(1 + \sqrt{1 + 4t_k^2}\right), t_0 = 0, \quad \beta_k = \frac{t_k - 1}{t_{k+1}}$$

β can be constant, and the optimal value for beta is $\beta = \frac{\sqrt{L}+\sqrt{\mu}}{\sqrt{L}-\sqrt{\mu}}$

In this project, I tested the Nesterov with 2 cases: use constant step size $1/L$ and β optimal and use step size 1 and β update by the function above.

Here is the result obtained, we can see that the nesterov algorithm have the optimal solution better than Gradient Descent.

2.3 Mini Batch and Stochastic gradient

Stochastic gradient descent fiding a solution for a finite sum function $f(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} f_i(\mathbf{w})$, and for linear regression problem, the function is rewrite as:

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

The algorithm will update w:

$$w_{k+1} = w_k - \tau_k \nabla f_{i_k}(w_k)$$

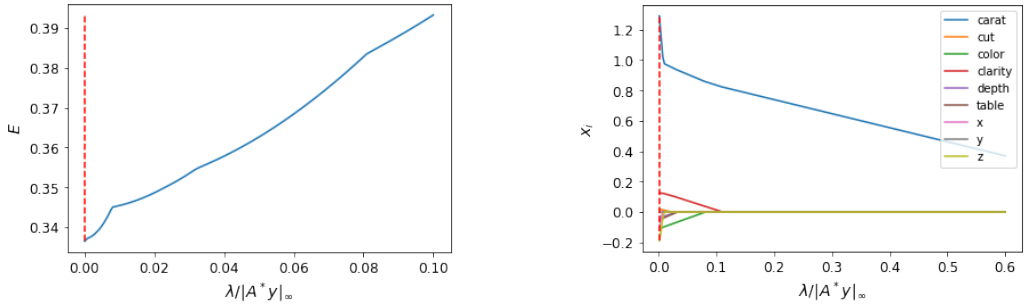
where at each iteration i_k is drawn in $\{1, \ldots, n\}$ uniformly at random.

A more general version of stochastic gradient, called batch stochastic gradient, is given by the iteration

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

Like Gradient Descent, SG use the step size $\alpha < \frac{2}{L}$

The result got in running SG with difference batch size and step size:



(a) Optimal λ for LASSO

(b) Regularization paths for LASSO

Figure 4: Testing with LASSO

2.4 Test with regularization

Regularization is obtained by introducing a penalty. It is often called ridge regression, and is defined as

$$min_w \|Aw - y^2\| + \lambda \|w\|^2$$

where $\lambda > 0$ is the regularization parameter.

Lasso regression change the norm-2 to norm-1

$$min_w \|Aw - y^2\| + \lambda \|w\|_1$$

where $\|w\|_1 = \sum_i |w_i|$

The simplest iterative algorithm to perform the minimization is the so-called iterative soft thresholding (ISTA), aka proximal gradient aka forward-backward, the function use for updating is:

$$w_{k+1} = \text{Soft}_{\tau \cdot \lambda} \left(w_k - \tau \cdot \nabla \left(f \left(w_k \right) \right) \right)$$

In the testing I choose the step size is $1/L$ and τ is $\frac{2}{\|X\|_2^2}$

We can see the solution that $\lambda = 0$ is optimal solution, it is because in this data set, we just have 9 features.

3 Optional Algorithms

3.1 Quasi-Newton Method

Quaisi-Newton use the iterate to update the solution:

$x_{k+1} = x_k + \alpha_k * p_k$ base on the method p_k is calculated difference.

- In newton method $p_k = -B_k^{-1} f(x_k)$ where $B_k = \nabla^2 f(x)$ is the Hessian matrix of $f(x)$, and $_k$ is searched by line search. In the Linear Regession Hessian matrix is: $\nabla^2 f(x) = X^T X$

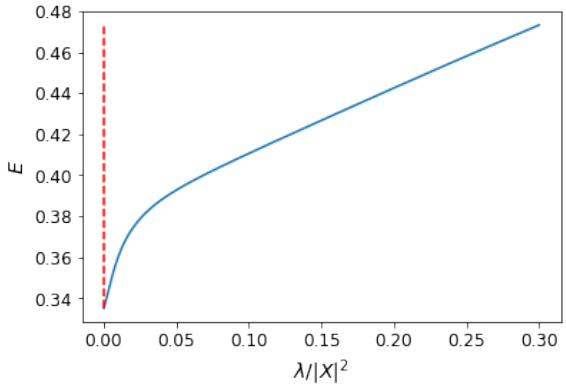


Figure 5: Optimal λ for RIGDE

- For the Broyden, Fletcher, Goldfarb, and Shanno (BFGS) method, p_k is calculated more complicate:

  Given starting point x_0, convergence tolerance $\epsilon > 0$, inverse Hessian approximation H_0; each iteration will calculate:

  $p_k = -H_k \nabla f_k$

  Set $x_{k+1} = x_k + \alpha_k p_k$ where α_k is computed from a line search procedure to satisfy the Wolfe conditions

  Define $s_k = x_{k+1} - x_k$ and $y_k = \nabla f_{k+1} - \nabla f_k$

  Compute H_{k+1} by

$$H_{k+1} = \left(I - \rho_k s_k y_k^T\right) H_k \left(I - \rho_k y_k s_k^T\right) + \rho_k s_k s_k^T,$$

  where $\rho_k = \frac{1}{y_k^T s_k}$

The source code below will show the implementation of BFGS algorithm

```
//code for BFGS

def Gradf_BFGS(w): return X_train.transpose().dot(X_train.dot(w)-y_train.reshape(n,1))
def BFGS(Lambda=0.01, iter=100):
    H_k = np.linalg.pinv(hessian())
    x_k = np.zeros((d, 1))
    all_f = []
    for i in range(iter):
        p_k = -H_k.dot(Gradf_BFGS(x_k))
        x_kp1 = x_k + Lambda*p_k
        s_k = x_kp1 - x_k
        y_k = Gradf_BFGS(x_kp1)-Gradf_BFGS(x_k)
        u = 1/y_k.T.dot(s_k)
        temp_1 = u*s_k.dot(y_k.T)
```

```
        Id_1 = np.identity(d)
        temp_2 = u*y_k.dot(s_k.T)
        Id_2 = np.identity(d)
        temp_3 = u*s_k.dot(s_k.T)
        H_k = (Id_1 - temp_1).dot(H_k).dot(Id_2 - temp_2) + temp_3
        x_k = x_kp1
        all_f.append(f(x_k))
    return x_k, all_f
```

The train loss and test loss on BFGS is:

```
MSE on testing set with BFGS method = 0.11011693465089858
fmin = 2305.373929079583
```

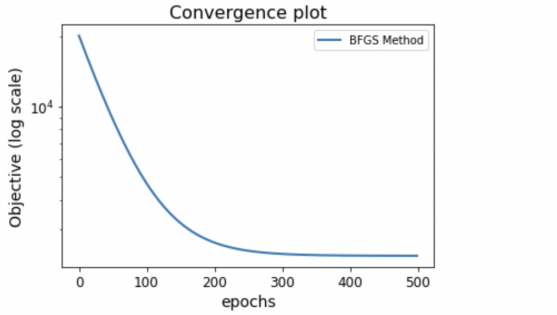


Figure 6: Training loss and test loss with BFGS method

3.2 Multilayer Perceptron

A multilayer perceptron (MLP) is a class of feedforward artificial neural network (ANN). An MLP consists of at least three layers of nodes: an input layer, a hidden layer and an output layer. Except for the input nodes, each node is a neuron that uses a nonlinear activation function.

In this project, I use the MLP with 20 nodes and the activation function is

$$\phi(x) = \frac{1}{\exp -x}$$

MLP will minimize the function

$$\min_{A,c} \mathcal{E}(A,c) \equiv \frac{1}{2}\sum_{i=1}^{n} \left(f_{A,c}\left(x_i\right) - y_i\right)^2 = \frac{1}{2}\|\varphi(XA)c - y\|^2$$

this function can solve by Gradient Descent on the variable A and c, but when tested the function is not convergence. By using the BFGS library of scipy optimize I can find the optimal value.

After finding the value of A and c, tested in the testing set with mean squared error get a better than Linear Regression

Loss on testing set with MLP = 0.028580443710929413

Loss on testing set with Optimal Solution = 0.11011693465089858