



Linear Regression -- My first Machine Learning Model

Jackson Hao

Table of Contents

- [Linear Regression -- My first Machine Learning Model](#)
 - [Table of Contents](#)
 - [Introduction](#)
 - [Section 1: Math Farmula of Linear Regression](#)
 - [Math Formula of Linear Regression](#)
 - [How to proceed Linear Regression](#)
 - [Least Squares Method](#)
 - [Gradient descent](#)
 - [Section 2: How to implement Linear Regression via Python & Pytorch](#)
 - [Try to express Linear Regression via linear algebra](#)
 - [Implement Linear Regression via Python & Pytorch](#)
 - [Conclusion](#)
-

Introduction

Linear Regression is one of the simplest in machine learning. My first learning model is from here. It is used to predict the relationship between a dependent variable and one or more independent variables by fitting a linear equation to observed data.

In this document, I learned from math formulas, then using `Python` and `Pytorch` framework to implement a simple Linear Regression model. Hope you would like it 😊

Section 1: Math Formula of Linear Regression

Math Formula of Linear Regression

The mathematical formula for a simple linear regression model with one independent variable is:

$$\hat{y} = w * x + b$$

Where:

- y is predicted output (dependent variable)
- x is input feature (independent variable)
- w is the weight (slope of the line)
- b is the bias (y-intercept)

How to proceed Linear Regression

Least Squares Method

Linear regression aims to find the best w and b to make \hat{y} as close as possible to the actual output y .

This is typically done by minimizing a loss function, such as Mean Squared Error (MSE):

$$Loss = \frac{1}{2m} \sum_{i=1}^m (y_i - \hat{y}_i)^2$$

Where:

- $Loss$ is the loss
- n is the number of data points

Minimizing the Loss, we can get a equation to solve for w and b :

$$w = \frac{n(\sum xy) - (\sum x)(\sum y)}{n(\sum x^2) - (\sum x)^2}, \quad b = \frac{\sum y - w(\sum x)}{n}$$

More math explanations please refer high school mathematics books 😊

Gradient descent

Gradient is described as the rate of change of a function. Take the single variable function $y = 2x^2 + 1$ as an example.

$$\nabla y = \frac{d}{dx}y = 4x$$

- Where ∇y is the gradient of function y to variable x .

There $y = f(x)$ is a single variable function, so the gradient is the derivative of y to x .

For multivariable functions, such as $z = f(x, y)$, the gradient is a vector that contains the partial derivatives of the function with respect to each variable:

$$\nabla z = \left(\frac{\partial z}{\partial x}, \frac{\partial z}{\partial y} \right)$$

- Where ∇z is the gradient of function z to variables x and y .

For a two-dimensional plane, if I want to find the fastest way to go up the hill, I can follow the direction of the gradient. Conversely, if I want to find the fastest way to go down the hill, I can follow the opposite direction of the gradient.

By acknowledging this property, we can choose MSE (the above-mentioned Loss function) as our objective function, and use gradient descent to minimize the Loss by updating the parameters w and b iteratively:

Each point's predicted error is:

$$y_i - \hat{y}_i = y_i - (w * x_i + b)$$

Use MSE as Loss function:

$$Loss = J(w, b) = \frac{1}{2m} \sum_{i=1}^m (y_i - (w * x_i + b))^2$$

There, MSE is used to describe the degree between predicted value and actual value.

Then, find the partial derivatives of Loss function with respect to w and b :

$$\frac{\partial J}{\partial w} = -\frac{1}{m} \sum_{i=1}^m x_i (y_i - (w * x_i + b))$$

$$\frac{\partial J}{\partial b} = -\frac{1}{m} \sum_{i=1}^m (y_i - (w * x_i + b))$$

Finally, update the parameters w and b using the gradients and a learning rate α , and use the descent direction to minimize the Loss:

$$w \leftarrow w - \alpha * \frac{\partial J}{\partial w}$$

$$b \leftarrow b - \alpha * \frac{\partial J}{\partial b}$$

By repeating this process iteratively, we can find the optimal values of w and b that minimize the Loss function, thus fitting the linear regression model to the data.

Section 2: How to implement Linear Regression via Python & Pytorch

Try to express Linear Regression via linear algebra

Written in matrix form, each sample's error can be expressed as:

$$\mathbf{e} = \mathbf{y} - \hat{\mathbf{y}} = \mathbf{X}\mathbf{w} - \mathbf{y}$$

Where:

- $\mathbf{X} \in \mathbb{R}^{n \times m}$ is the input feature matrix, with n samples and m features.
- $\mathbf{w} \in \mathbb{R}^{m \times 1}$ is the weight vector.
- $\mathbf{y} \in \mathbb{R}^{n \times 1}$ is the actual output vector.

So the MSE Loss function can be expressed as:

$$Loss = J(\mathbf{w}) = \frac{1}{2m} \mathbf{e}^T \mathbf{e} = \frac{1}{2m} (\mathbf{X}\mathbf{w} - \mathbf{y})^T (\mathbf{X}\mathbf{w} - \mathbf{y})$$

Take a example, in cartesian coordinate system, we have 4 data points: (1,2), (2,3), (3,5), (4,7). So our input feature matrix will have 4 samples and 1+1 features (the extra 1 is constant 1 for bias term). The actual output vector will have 4 samples.

$$\mathbf{X} = \begin{bmatrix} 1 & 1 \\ 2 & 1 \\ 3 & 1 \\ 4 & 1 \end{bmatrix}$$

If our weight and bias are:

$$\mathbf{w} = \begin{bmatrix} b \\ w \end{bmatrix} = \begin{bmatrix} 0.5 \\ 1.5 \end{bmatrix}$$

Then the predicted output vector will be:

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} = \begin{bmatrix} 1 & 1 \\ 2 & 1 \\ 3 & 1 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} 0.5 \\ 1.5 \end{bmatrix} = \begin{bmatrix} 2.0 \\ 2.5 \\ 3.0 \\ 3.5 \end{bmatrix}$$

Symbol	Meaning	Dimension	Example Value
\mathbf{X}	Input feature matrix	$n \times m$	$\begin{bmatrix} 1 & 1 \\ 2 & 1 \\ 3 & 1 \\ 4 & 1 \end{bmatrix}$
\mathbf{w}	Weight vector	$m \times 1$	$\begin{bmatrix} 0.5 \\ 1.5 \end{bmatrix}$
$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w}$	Predicted output vector	$n \times 1$	$\begin{bmatrix} 2.0 \\ 2.5 \\ 3.0 \\ 3.5 \end{bmatrix}$

Gradient descent update rules in matrix form:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha * \nabla J(\mathbf{w})$$

Implement Linear Regression via Python & Pytorch

```
import torch

# Sample data
X = torch.tensor([[1.0], [2.0], [3.0]])
y = torch.tensor([[2.0], [3.0], [4.0]])

# Initialize parameters
w = torch.randn(1, requires_grad=True) # weight
b = torch.randn(1, requires_grad=True) # bias

# Hyperparameters
lr = 0.1
epochs = 100

for epoch in range(epochs):
    # frontward Pass:
    y_hat = X * w + b

    # Loss calculation: Mean Squared Error
    loss = ((y_hat - y) ** 2).mean() / 2

    # backward Pass
    loss.backward()

    # upload parameters
    with torch.no_grad():
        w -= lr * w.grad
        b -= lr * b.grad

    # clear gradients, or they will accumulate
    w.grad.zero_()
    b.grad.zero_()

    if epoch % 10 == 0:
        print(f'Epoch {epoch}, loss={loss.item()}, w={w.item():.3f}, b={b.item():.3f}')
```

In this code, our input is expressed as a 2D tensor `X` with shape (n, m) , where `n` is the number of

samples and `m` is the number of features. The weight `w` is also a 2D tensor with shape $(m, 1)$. The predicted output `y_hat` is calculated using matrix multiplication, and the loss is computed using Mean Squared Error (MSE). The gradients are calculated using PyTorch's automatic differentiation, and the parameters are updated using gradient descent.

Conclusion

Using gradient descent to optimize the parameters of a linear regression model is a fundamental technique in machine learning. By iteratively updating the weights and bias based on the gradients of the loss function, we can effectively minimize the error between predicted and actual values. This approach not only provides a clear understanding of how linear regression works but also lays the groundwork for more complex models and optimization techniques in the field of machine learning.

Written by Jackson Hao, November 7th 2025 Friday, Happy coding! 😊