Linear Regression from Scratch

Bhanu Prasanna Koppolu

Table of contents

Note

Full implementation available at GitHub - ML from Scratch

Linear Regression

Closed Form Solution

The Linear Regression closed form solution is:

$$\theta = (X^T X)^{-1} X^T y$$

Where θ is the optimal parameters.

We need to add another column of 1s for the **Intercept (bias)**. Without the Intercept, the fitted line is forced to pass through the origin. So we form a design matrix $\bar{X} = [1, X]$ and use θ to compute the closed form solution.

```
import numpy as np

X_bar = np.column_stack((np.ones(X.shape[0]), X))

l_h_t = np.linalg.inv(np.transpose(X_bar) @ (X_bar))
r_h_t = X_bar.T @ y
theta = l_h_t @ r_h_t
```

Gradient Descent

It took me a long time to perfectly understand gradient descent. But in short it is just the **chain rule** underneath.

The algorithm doing all this is **Backpropagation** calculating gradients in this process allowing the gradient descent.

So, for linear regression, the equation looks like this:

$$y = mx + b$$

Where if we push it to matrix notation then:

$$y = X \cdot \theta + B$$

or

$$y = \bar{X} \cdot \theta$$

where the intercept (bias) is included in $\bar{X}$. $\bar{X}$ is the design matrix.

So, we are trying to find the θ and B (intercept or bias) where the X gets as close as to Y, when we plug in X and the optimized θ and B.

It is all partial derivatives underneath, that is more terms (or multiple variables not just one like in ordinary differential equations or normal differential equations) to derivate with.

Loss Function

We have a loss function:

$$\ell(y_i, \hat{y}_i) = \frac{1}{2}(\hat{y}_i - y_i)^2$$

where $\hat{y}$ is the prediction $\hat{y} = \bar{X} \cdot \theta$, and y is the true value.

So, our goal is to get $\hat{y}$ as close as possible to y by minimizing the loss.

Cost Function

The cost J is:

$$J = \frac{1}{m} \sum_{i=0}^{m} \ell(y_i, \hat{y}_i)$$

Sometimes it is also $\frac{1}{2m}$ rather than just $\frac{1}{m}$ for expanded loss. Where m is the number of data samples.

Gradient Computation

Then we do delta (referred to as d throughout for brevity - concise notation):

$$\frac{\partial \ell}{\partial w} = \frac{\partial \ell}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w} = (\hat{y} - y) \cdot x$$

$$\frac{\partial \ell}{\partial b} = \frac{\partial \ell}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial b} = (\hat{y} - y)$$

$$\frac{\partial J}{\partial w} = \frac{1}{m} \sum \frac{\partial \ell}{\partial w}$$

$$\frac{\partial J}{\partial b} = \frac{1}{m} \sum \frac{\partial \ell}{\partial b}$$

This is the **chain rule** if you clearly observe it, so this applies for more complex architectures like FFN with multiple layers, which goes on. But the final goal is to minimize the loss.

But as all these are matrix operations it is much more easier for us to do, Lord NumPy.

At final after solving these, we get:

$$\frac{\partial J}{\partial w} = \frac{1}{m} X^T (\hat{y} - y)$$

$$\frac{\partial J}{\partial b} = \frac{1}{m} \sum (\hat{y}_i - y_i)$$

Parameter Update

So, we just found the gradients but haven't yet updated the w and b to get closer, we need another term η which is the learning rate that we have to use:

$$w = w - \eta \cdot \frac{\partial J}{\partial w}$$

$$b = b - \eta \cdot \frac{\partial J}{\partial b}$$

This happens over a lot of iterations which allows us to minimize the loss as much as possible.

```python
def linear_regression(X, y, learning_rate=0.01, iterations=10):
    m = X.shape[0]  # Number of samples

    w = np.random.rand(X.shape[1])
    bias = np.random.rand(1)

    cost_history = []

    for i in range(iterations):
        y_hat = X @ w + bias
        dl_dw = X.T @ (y_hat - y)
        dl_db = (y_hat - y)

        dj_dw = (1/m) * dl_dw
        dj_db = (1/m) * np.sum(dl_db)

        w = w - learning_rate * dj_dw
        bias = bias - learning_rate * dj_db

        cost = (1/(2*m)) * np.sum((y_hat - y)**2)
        cost_history.append(cost)

    return w, bias, cost_history
```