Linear Regression

December 2021

1 Machine Learning Lab#4

1.1 Aim: Implement Linear Regression Algorithm on the given dataset

2 Introduction

Linear Regression is a supervised machine learning algorithm where the predicted output is continuous. In this lab we will implement the Linear Regression algorithm in Python using Pytorch library. PyTorch is an open source machine learning library based on Torch library which is used for scientific computation. Pytorch tutorials for practice and documentation is available in the following link https://pytorch.org/docs/stable/index.html.

2.0.1 Univariate Linear Regression

Simple linear regression uses traditional slope-intercept form, where θ_0 and θ_1 are the parameters our algorithm will try to "learn" to produce the most accurate predictions. x represents our input data and y represents our prediction.

$$y = \theta_0 + \theta_1 x$$

2.0.2 Multiple Variable Linear Regression

A more complex, multi-variable linear equation might look like this, where θ represents the coefficients, our model will try to learn.

$$y = \theta^T X = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + ... + \theta_n x_n$$

The variables x_1,x_2,x_3...x_n represent the independent variables in our dataset. θ_0 represents intercept to y-axis or the bias term. For crop yeild prediction of apples these independent variables might include temperature, rainfall and humidity.

$$y = \theta_0 + \theta_1 temperature + \theta_2 rainfall + \theta_3 humidity$$

2.0.3 Cost Function

Cost function is used to compare the actual target value with model predictions. MSE(Mean Square Error) function is used in Linear Regression to calculate the loss.

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

where, $\hat{y^i}$ represents the model predicted value for i^{th} training sample. m represents the number of training samples. y^i represents the actual target value for i^{th} training sample.

2.0.4 Gradient Descent Algorithm

In linear regression the main goal of the model is to find the best fit line for a given dataset. The model targets to minimize the cost function.

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

Gradient descent is an iterative optimization algorithm to find the minimum of a function.

Repeat until convergence

{

$$\theta_j := \theta_j - \alpha \frac{\partial J(\theta_0, \theta_1)}{\partial \theta_j}$$

}

where $J(\theta_0, \theta_1)$ is the cost function and $\frac{\partial J(\theta_0, \theta_1)}{\partial \theta_j}$ is the update or rate of change.

Linear Regression utilizes this algorithm to minimize the Cost function by finding the optimal value of parameters θ. Gradient descent starts with a random value of θ and keep updating the parameter values until convergence. On each iteration, we apply the "update rule" (the := symbol means replace theta with the value computed on the right):
For Univariate Linear regression

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (\hat{y^i} - y^i)$$

$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} (\hat{y^i} - y^i) x^i$$

where α is a parameter called the learning rate. It represents the step size. Selecting the right learning rate is critical. If the learning rate is too large, one can overshoot the minimum and diverge. For Multivariate Linear regression:
Repeat until convergence

{

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (\hat{y^i} - y^i) x_j^i$$

}

simultaneously update θ_j for j = *0,1,2,...n*.

In this lab we'll create a model that predicts crop yeilds for apples (target variable) by looking at the average temperature, rainfall and humidity (input variables or features) in different regions.

3 Implementation

[ ]:
```
# Import Numpy & PyTorch
import numpy as np
import torch
```

A tensor is a number, vector, matrix or any n-dimensional array.

3.1 Problem Statement

We'll create a model that predicts crop yeilds for apples (*target variable*) by looking at the average temperature, rainfall and humidity (*input variables or features*) in different regions.

Here's the training data:

Temp	Rain	Humidity	Prediction
73	67	43	56
91	88	64	81
87	134	58	119
102	43	37	22
69	96	70	103

In a **linear regression** model, each target variable is estimated to be a weighted sum of the input variables, offset by some constant, known as a bias :

```
yeild_apple  = w11 * temp + w12 * rainfall + w13 * humidity + b1
```

It means that the yield of apples is a linear or planar function of the temperature, rainfall & humidity.

Our objective: Find a suitable set of *weights* and *biases* using the training data, to make accurate predictions.

Training Data

The training data can be represented using 2 matrices (inputs and targets), each with one row per observation and one column for variable.

[2]:
```
# Input (temp, rainfall, humidity)
inputs = np.array([[73, 67, 43],
                   [91, 88, 64],
                   [87, 134, 58],
                   [102, 43, 37],
                   [69, 96, 70]], dtype='float32')
```

[3]:
```
# Target (apples)
targets = np.array([[56],
                    [81],
                    [119],
                    [22],
                    [103]], dtype='float32')
```

Before we build a model, we need to convert inputs and targets to PyTorch tensors.

[4]:
```
# Convert inputs and targets to tensors
```

3.2 Linear Regression Model (from scratch)

The *weights* and *biases* can also be represented as matrices, initialized with random values. The first row of `w` and the first element of `b` are use to predict the first target variable i.e. yield for apples, and similarly the second for oranges.

[5]:
```
# Weights and biases
```

The *model* is simply a function that performs a matrix multiplication of the input `x` and the weights `w` (transposed) and adds the bias `b` (replicated for each observation).

$$X \qquad \times \qquad W^T \qquad + \qquad b$$

$$\begin{bmatrix} 73 & 67 \\ 43 & \\ 91 & 88 \\ 64 & \\ \vdots & \vdots \\ \vdots & \\ 69 & 96 \\ 70 & \end{bmatrix} \times \begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{bmatrix} + \begin{bmatrix} b_1 & b_2 \\ b_1 & b_2 \\ \vdots & \vdots \\ b_1 & b_2 \end{bmatrix}$$

[6]:
```
# Define the model
```

The matrix obtained by passing the input data to the model is a set of predictions for the target variables.

[7]:
```
# Generate predictions
```

[8]:
```
# Compare with targets
```

Because we've started with random weights and biases, the model does not perform a good job of predicting the target varaibles.

3.3 Loss Function

We can compare the predictions with the actual targets, using the following method: * Calculate the difference between the two matrices (`preds` and `targets`). * Square all elements of the difference matrix to remove negative values. * Calculate the average of the elements in the resulting matrix.

The result is a single number, known as the **mean squared error** (MSE).

[9]:
```
# MSE loss
```

[10]:
```
# Compute loss
```

The resulting number is called the **loss**, because it indicates how bad the model is at predicting the target variables. Lower the loss, better the model.

3.4 Compute Gradients

With PyTorch, we can automatically compute the gradient or derivative of the `loss` w.r.t. to the weights and biases, because they have `requires_grad` set to `True`.

More on autograd: https://pytorch.org/docs/stable/autograd.html#module-torch.autograd

[11]:
```
# Compute gradients
```

The gradients are stored in the `.grad` property of the respective tensors.

[12]:
```
# Gradients for weights
```

[13]:
```
# Gradients for bias
```

A key insight from calculus is that the gradient indicates the rate of change of the loss, or the slope of the loss function w.r.t. the weights and biases.

- If a gradient element is **postive**,
  - **increasing** the element's value slightly will **increase** the loss.
  - **decreasing** the element's value slightly will **decrease** the loss.
- If a gradient element is **negative**,
  - **increasing** the element's value slightly will **decrease** the loss.
  - **decreasing** the element's value slightly will **increase** the loss.

The increase or decrease is proportional to the value of the gradient.

Finally, we'll reset the gradients to zero before moving forward, because PyTorch accumulates gradients.

[13]:
```

```

3.5 Adjust weights and biases using gradient descent

We'll reduce the loss and improve our model using the gradient descent algorithm, which has the following steps:

1. Generate predictions
2. Calculate the loss
3. Compute gradients w.r.t the weights and biases
4. Adjust the weights by subtracting a small quantity proportional to the gradient
5. Reset the gradients to zero

[14]:
```
# Generate predictions
```

[15]:
```
# Calculate the loss
```

[16]:
```
# Compute gradients
```

[17]:
```
# Adjust weights & reset gradients
```

[18]:
```
#print(w)
```

With the new weights and biases, the model should have a lower loss.

[19]:
```
# Calculate loss
```

3.6 Train for multiple epochs

To reduce the loss further, we repeat the process of adjusting the weights and biases using the gradients multiple times. Each iteration is called an epoch.

[20]:
```
# Train for 100 epochs
```

[21]:
```
# Calculate loss
```

[22]:
```
# Print predictions
```

[23]:
```
# Print targets
```