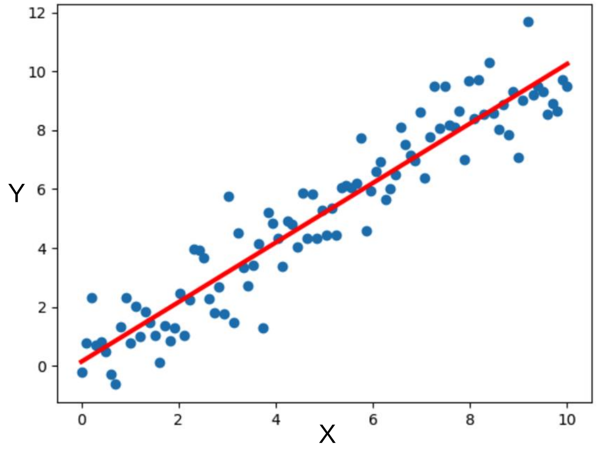
Linear Regression

Objectives

- Understand the concept of linear regression machine learning model.
- Understand the Mean Squared Error (MSE) loss function and its usage in linear regression.
- Understand the how Gradient Descent (GD) works in linear regression
- Implementation of linear regression using Python

What is linear regression?

- Linear regression is a type of supervised machine learning algorithm, which aims to determine the best-fit linear line between the independent variables (i.e., input features, X) and quantitative dependent variables (i.e., output, Y).



- It is one of the easiest, most well understood and most popular algorithms in many machine learning applications to date, such as making predictions for numeric variables, salary, sales, production yield, greenhouse gas emission, house price, to name a few.

Start from linear function

- Given 2 points (0, 1) and (100, 10), find the linear function passing through them in the form of $y = wx + b$

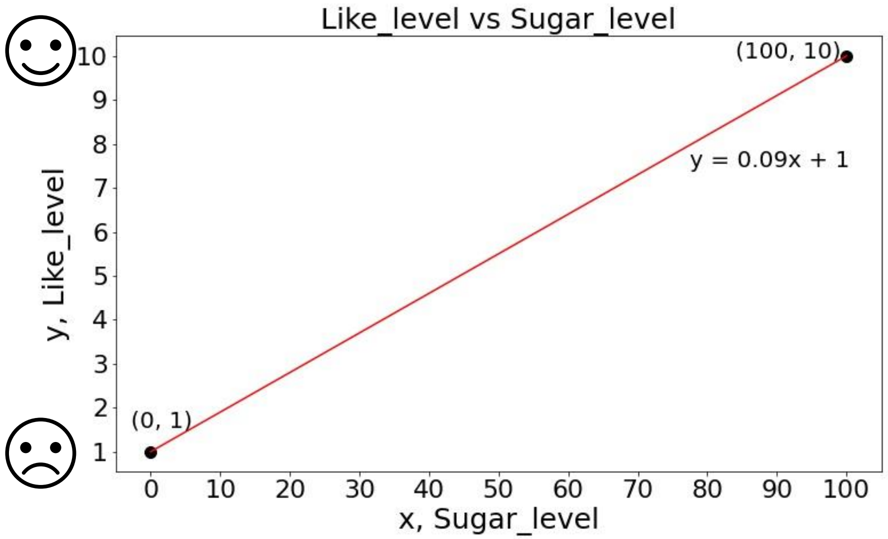
$$\begin{cases} 1 = 0w + b \\ 10 = 100w + b \end{cases}$$

$$b = 1$$

$$100w = 9$$

$$w = \frac{9}{100} = 0.09$$

$$y = 0.09x + 1$$

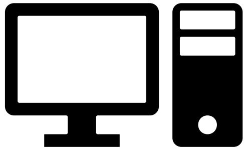


Start from linear function

- 2 points can uniquely determine a line. Now, what if there are more than 2 points, how can we find the best fit linear function?
- This can be solved by linear regression machine learning

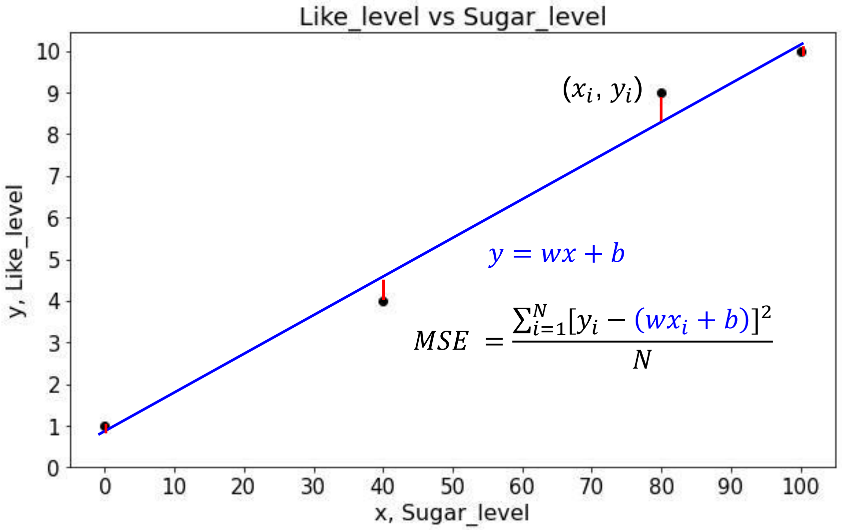
x, Sugar_level	y, Like_level
0	1
40	4
80	9
100	10

Pass to machine



machine to “learn” the x ~ y relationship and give optimal linear function

$$y = wx + b$$



The optimal/best fit line is a function that leads to minimal mean squared error (MSE) between real y values and predicted y values by the function

Loss function: mean squared error (MSE)

$$MSE = \frac{\sum_{i=1}^{N}[y_i - (wx_i + b)]^2}{N}$$

- Where y_i is the real value of Like_level for the i^{th} data point
- x_i is the value of Sugar_level for the i^{th} data point
- w, slope/gradient of a linear line, is often called weight in machine learning; that is, the weight associated with feature Sugar_level
- b, intercept with y-axis, is often called bias in machine learning
- $(wx_i + b)$ is the predicted y value for x_i by the linear model
- N is the number of training data points (in the current example, it is 4)

Loss function: mean squared error (MSE)

$$MSE = \frac{\sum_{i=1}^{N}[y_i - (wx_i + b)]^2}{N}$$

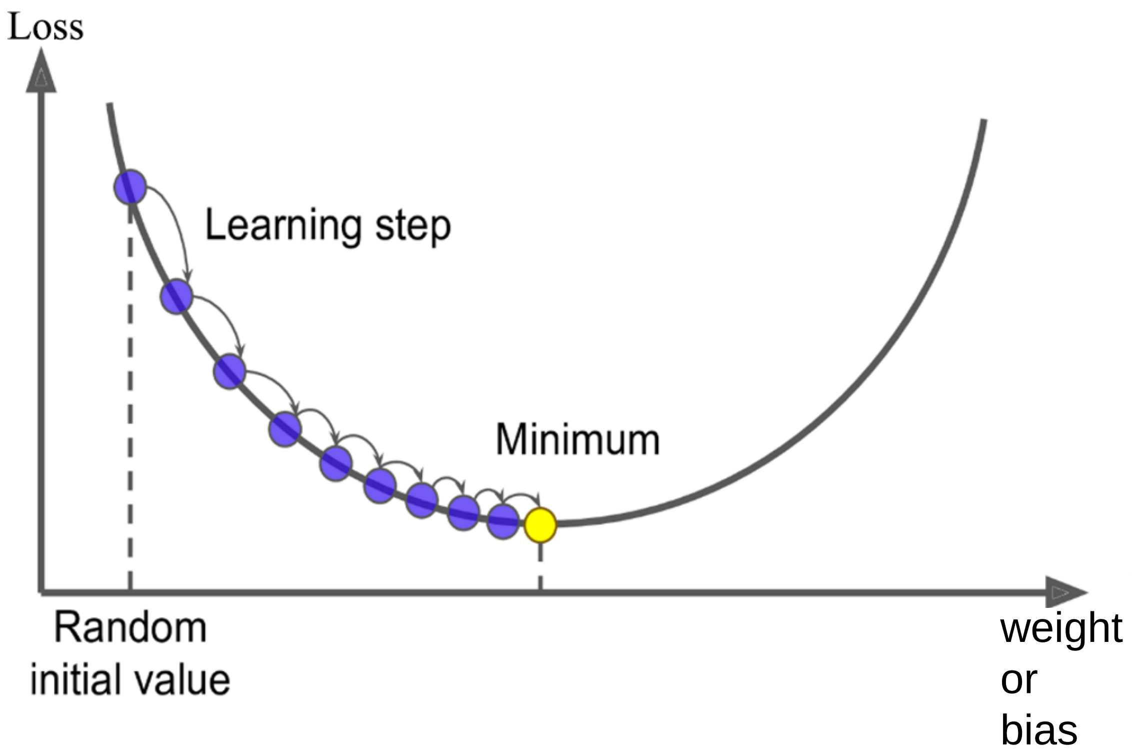
- Our objective is to find the optimal weight and bias, w and b, that minimize the loss function value, i.e., MSE here.
- If we can make the loss = zero, then the prediction is perfect
- The higher the value of the loss function, the more wrong our predictions will be.
- Now we have a loss function to optimize, and we can use an optimization algorithm to find the optimal weight and bias that lead to the minimal loss value.

Gradient descent

- A very common algorithm that we can use to solve this optimization problem is called Gradient Descent (GD).

- It is an iterative algorithm that can be used to minimize the loss function value and find the best weights and bias.

- It works by tweaking each of the weights and bias a tiny bit at a time in the direction that will reduce the loss.



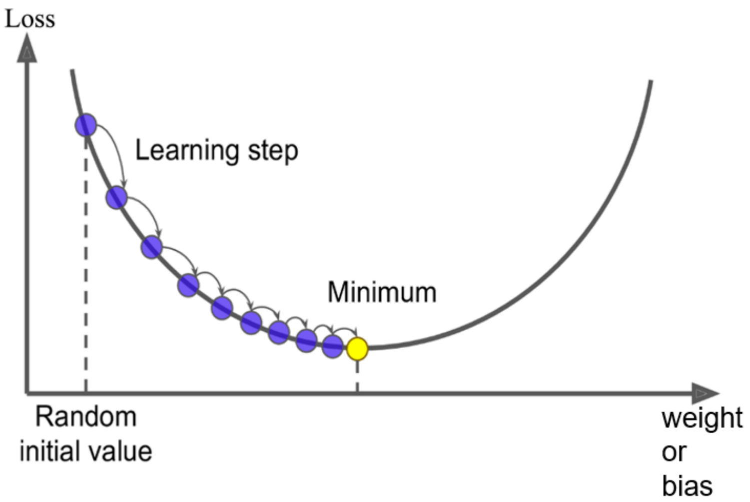
Gradient Descent

- $Loss = L = MSE = \frac{\sum_{i=1}^{N}[y_i - (wx_i + b)]^2}{N}$ let's use L to represent the loss MSE value

Since both weight and bias influence the loss function, the differentiation has to be performed w.r.t. each of them, hence the use of partial differentiation, as follows:

- $\frac{\partial L}{\partial w} = \frac{\sum_{i=1}^{N} 2[y_i - (wx_i + b)](-x_i)}{N}$
- $\frac{\partial L}{\partial b} = \frac{\sum_{i=1}^{N} 2[y_i - (wx_i + b)](-1)}{N}$
- $w_{new} = w_{old} - \frac{\partial L}{\partial w} \cdot \alpha$
- $b_{new} = b_{old} - \frac{\partial L}{\partial w} \cdot \alpha$

Calculate and tweak the weight and bias iteratively until reaching the optimum (i.e., minimal loss)



α is learning rate (e.g., 0.01), controlling how tiny the learning step size is in gradient descent.

DigiPen INSTITUTE OF TECHNOLOGY SINGAPORE

Implementation using Python

$$Loss = L = MSE = \frac{\sum_{i=1}^{N}[y_i - (wx_i + b)]^2}{N}$$

```python
import matplotlib.pyplot as plt
import numpy as np
import time

def loss_function(sugar_level, like_level, weight, bias):
    len_data = len(sugar_level)
    total_error = 0.0
    for i in range(len_data):
        total_error += (like_level[i] - (weight*sugar_level[i] + bias))**2
    return total_error / len_data
```

$$\frac{\partial L}{\partial w} = \frac{\sum_{i=1}^{N} 2[y_i - (wx_i + b)](-x_i)}{N}$$

$$\frac{\partial L}{\partial b} = \frac{\sum_{i=1}^{N} 2[y_i - (wx_i + b)](-1)}{N}$$

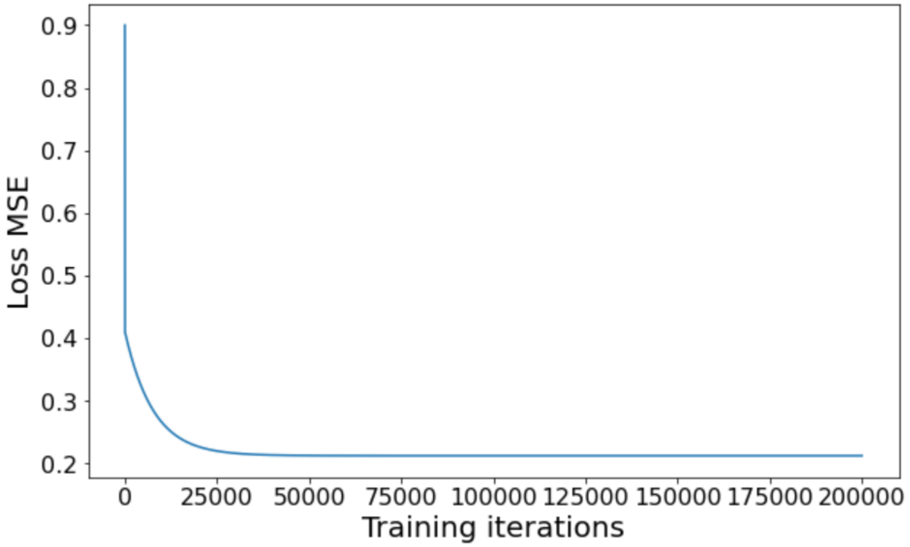
$$w_{new} = w_{old} - \frac{\partial L}{\partial w} \cdot \alpha$$

$$b_{new} = b_{old} - \frac{\partial L}{\partial w} \cdot \alpha$$

```python
def update_weights(sugar_level, like_level, weight, bias, learning_rate):
    weight_deriv = 0
    bias_deriv = 0
    len_data = len(sugar_level)

    for i in range(len_data):
        # Calculate partial derivatives
        # -2x(y - (wx + b))
        weight_deriv += -2*sugar_level[i] * (like_level[i] - (weight*sugar_level[i] + bias))

        # -2(y - (wx + b))
        bias_deriv += -2*(like_level[i] - (weight*sugar_level[i] + bias))

    # We subtract because the derivatives point in direction of steepest ascent
    weight -= (weight_deriv / len_data) * learning_rate
    bias -= (bias_deriv / len_data) * learning_rate

    return weight, bias
```

Implementation using Python

```
def train(sugar_level, like_level, weight, bias, learning_rate, iters):
    cost_history = []

    for i in range(iters):
        weight,bias = update_weights(sugar_level, like_level, weight, bias, learning_rate)

        #Calculate loss
        cost = loss_function(sugar_level, like_level, weight, bias)
        cost_history.append(cost)

        if i == 10 or i == 1000 or i == 10000 or i >= iters-4:
            print ("iter={:d} \t weight={:.4f} \t bias={:.4f} \t cost={:.4f}".format(i, weight, bias, cos
    return cost_history
```



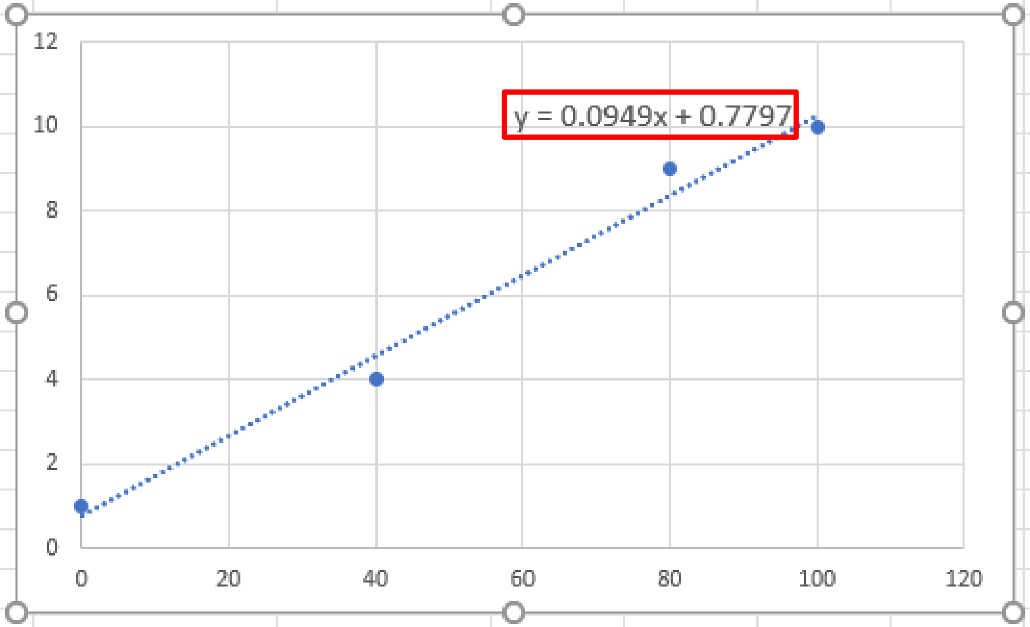

```
sugar_level = [0, 40, 80, 100]
like_level  = [1, 4, 9, 10]
initial_weight = 0
initial_bias = 0
learning_rate = 0.0001
iters = 200000

start_time = time.time()
cost_history = train(sugar_level, like_level, initial_weight, initial_bias, learning_rate, iters)
end_time = time.time()
print("Time taken to train 200000 times': " , round(end_time - start_time,3),"seconds")
```

```
iter=10          weight=0.1044   bias=0.0018    cost=0.4102
iter=1000        weight=0.1038   bias=0.0507    cost=0.3860
iter=10000       weight=0.0999   bias=0.3756    cost=0.2654
iter=199996      weight=0.0949   bias=0.7797    cost=0.2119
iter=199997      weight=0.0949   bias=0.7797    cost=0.2119
iter=199998      weight=0.0949   bias=0.7797    cost=0.2119
iter=199999      weight=0.0949   bias=0.7797    cost=0.2119
Time taken to train 200000 times':  0.65 seconds
```

In this example, the final optimal weight for Sugar_level is 0.0949 and optimal bias is found to be 0.7797
Hence, the linear regression model is $y = 0.0949x + 0.7797$

Validation 1: Quick comparison with MS Excel

x, Sugar_level	y, Like_level
0	1
40	4
80	9
100	10



- Same weight and bias are found in MS Excel linear trend line.
- This proves our theoretical understanding and implementation of linear regression in Python are correct.

INSTITUTE OF TECHNOLOGY
SINGAPORE