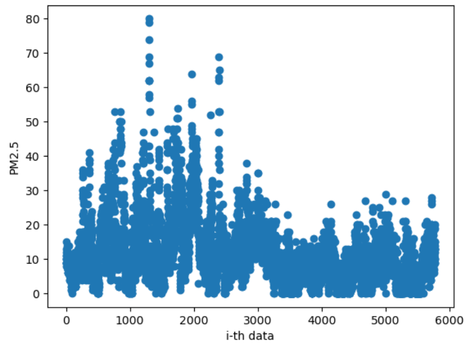
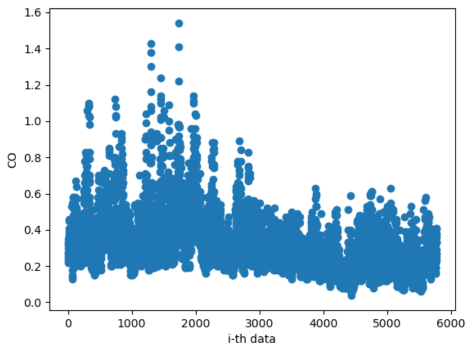
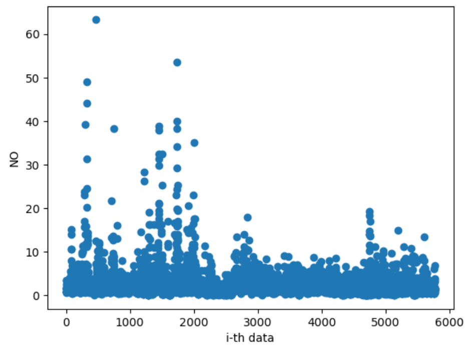
學號：R11921A16 系級： 電機碩一 姓名：何秉學

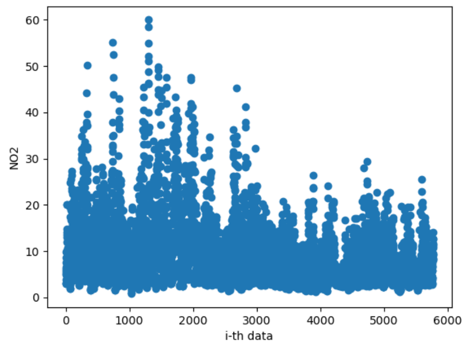
1. (1%) 解釋什麼樣的 data preprocessing 可以 improve 你的 training/testing accuracy，e.g., 你怎麼挑掉你覺得不適合的 data points。請提供數據(例如 kaggle public score RMSE)以佐證你的想法。

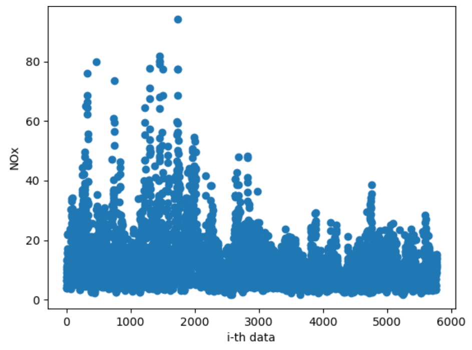
   After observing the training data visualized image, you can be aware of the relationship between the PM2.5 feature and the others.
   For instance, the CO image, NO image, NO2 image, and NOx image are much more correlated with PM2.5.



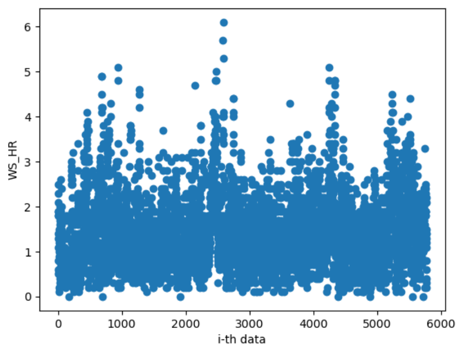


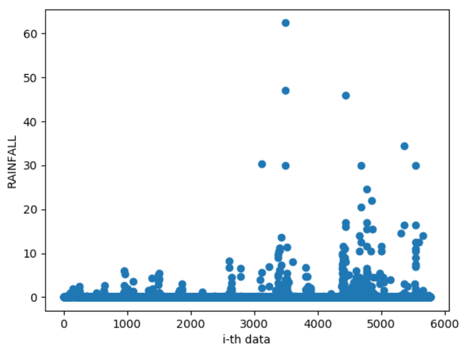


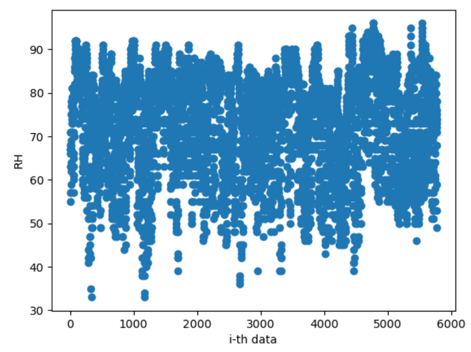


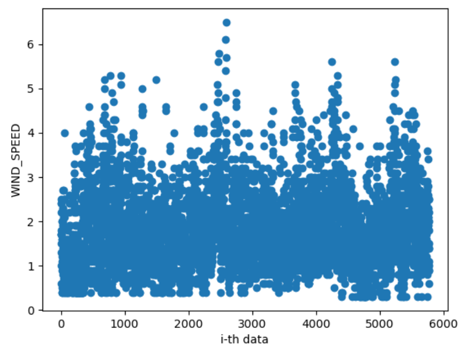


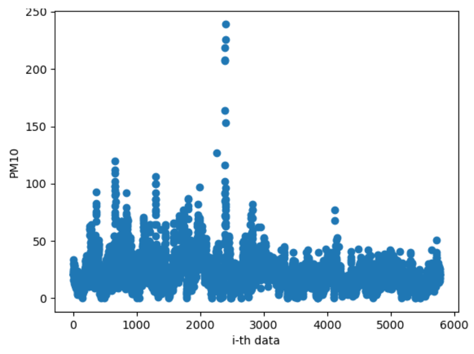
   I also choose PM10, WS_HR, RAINFALL, RH, WIND_SPEED which are also correlated to PM2.5 but not that much as above 5 features.











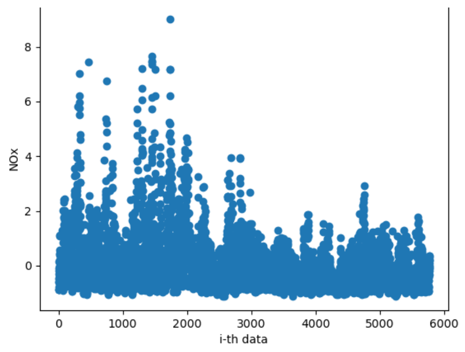
And in my experience, normalizing data can help to gather all data to a specific area that the model can converge much more rapidly. But using normalization is not like what I thought. In this case, the result is worse and also appear negative value of the PM2.5 result. According to this page, I thought maybe the normalization method is not suitable in my case.

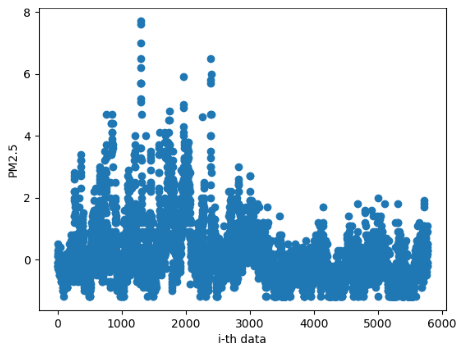
I also figured that using the stored weight and bias by my pretrained model is not the right way. I used pickle to store the dump parameters during the training and used the best one as my pretrained parameter. But it's still not that good enough.

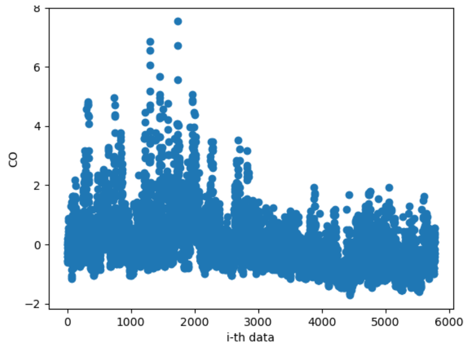
The better way in this project to enhance your accuracy is tuning your training config and select good features.

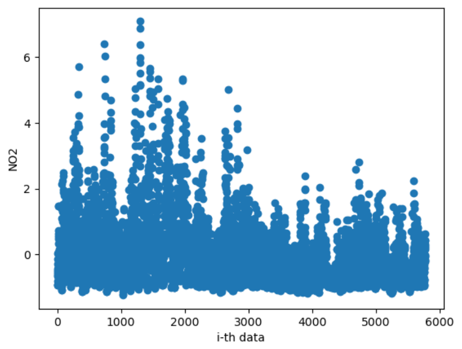
After discussing with my friend, I figured out the problem and tried to solve it successfully by fitting numpy random seed. Then, the parameter will truly fix and normalization will also work to help model converging.

I used Zscore normalization to implement in my project and can see as below









You can see the different result of using or unusing normalization with the same config.

Epoch	Regression	Learning Rate	Feats	Batch Size	Loss Function	Optimizer	ACC(Lower is Better)	Comment
1000	2nd_order	0.015	[1-4, 6-9, 13, 14]	1024	MSE	Adam	3.21961	Normalization
1000	2nd_order	0.015	[1-4, 6-9, 13, 14]	1024	MSE	Adam	3.92536	Unuse Normalization

2. (1%) 請實作 2nd-order polynomial regression model (不用考慮交互項)。

(a) 貼上 polynomial regression 版本的 Gradient descent code 內容

```
def minibatch(x, y, config):

  # Randomize the data in minibatch
  index = np.arange(x.shape[0])
  np.random.shuffle(index)
  x = x[index]
  y = y[index]

  # Initialization
  batch_size = config.batch_size
  lr = config.lr
  lam = config.lam
  epoch = config.epoch

  beta_1 = np.full(x[0].shape, 0.9).reshape(-1, 1)
  beta_2 = np.full(x[0].shape, 0.99).reshape(-1, 1)
  # Linear regression: only contains two parameters (w, b).
  w = np.full(x[0].shape, 0.1).reshape(-1, 1)
```

```
w2 = np.full(x[0].shape, 0.1).reshape(-1, 1)    # Implement 2-nd polynomial regression
bias = 0.1
m_t = np.full(x[0].shape, 0).reshape(-1, 1)
v_t = np.full(x[0].shape, 0).reshape(-1, 1)
m_t_2 = np.full(x[0].shape, 0).reshape(-1, 1)    # Implement 2-nd polynomial regression
v_t_2 = np.full(x[0].shape, 0).reshape(-1, 1)    # Implement 2-nd polynomial regression
m_t_b = 0.0
v_t_b = 0.0
t = 0
epsilon = 1e-8

# Training loop
total_loss = np.zeros(epoch)
for num in range(epoch):
    for b in range(int(x.shape[0]/batch_size)):
        t+=1
        x_batch = x[b * batch_size:(b+1) * batch_size]
        y_batch = y[b * batch_size:(b+1) * batch_size].reshape(-1,1)

        # Implement 2-nd polynomial regression
        pred = np.dot(x_batch, w) + np.dot(x_batch**2, w2) + bias

        # loss(In this project, we use MSE Loss function.)
        loss = y_batch - pred   # This loss is just a variable, that actually loss function.

        # Compute w gradient
        g_t = np.dot(x_batch.transpose(), loss) * (-2)
        m_t = beta_1 * m_t + (1-beta_1) * g_t
        v_t = beta_2 * v_t + (1-beta_2) * np.multiply(g_t, g_t)
        m_cap = m_t / (1-(beta_1**t))
        v_cap = v_t / (1 - (beta_2**t))

        # Compute w2 gradient
        g_t_2 = np.dot((x_batch**2).transpose(), loss) * (-2)
        m_t_2 = beta_1 * m_t_2 + (1-beta_1) * g_t_2
        v_t_2 = beta_2 * v_t_2 + (1-beta_2) * np.multiply(g_t_2, g_t_2)
        m_cap_2 = m_t_2 / (1-(beta_1**t))
        v_cap_2 = v_t_2 / (1 - (beta_2**t))

        # Compute bias gradient
        g_t_b = loss.sum(axis=0) * (-2)
        m_t_b = 0.9 * m_t_b + (1 - 0.9) * g_t_b
        v_t_b = 0.99 * v_t_b + (1 - 0.99) * (g_t_b * g_t_b)
        m_cap_b = m_t_b / (1 - (0.9**t))
        v_cap_b = v_t_b / (1 - (0.99**t))

        w_0 = np.copy(w)

        # Update weight & bias
        w -= ((lr * m_cap) / (np.sqrt(v_cap) + epsilon)).reshape(-1, 1)
        w2 -= ((lr * m_cap_2) / (np.sqrt(v_cap_2) + epsilon)).reshape(-1, 1)
        bias -= (lr * m_cap_b) / (math.sqrt(v_cap_b) + epsilon)

return w, bias
```

(b) 在只使用 NO 數值作為 feature 的情況下，紀錄該 model 所訓練出的 parameter 數值（w2, w1, b）以及 kaggle public score.

Weight1:
[[0.40283195][0.26396436][0.08228792][0.22204582][0.21317418][0.2013591][-0.00086484][0.63271473]]

Weight2: [[-0.0110923] [-0.00632303] [-0.00157317] [-0.00358574] [-0.00249647] [-0.00262696] [0.00265367] [-0.00908543]]
Bias: [7.09662601]

Kaggle score: 7.19883

3. (4%) Refer to math problem: https://hackmd.io/@lH2AB7kCSAS3NPw2FffsGg/Sk1n8xPWo?fbclid=IwAR0LiCps2fhIZFJT-gYP8kr7KlvLaRvS9-ftLIaPQY5DVgye1AuHM-RW3Yg