ML2016 HW1

R04942070 吳冠融

1. Linear regression function by Gradient Descent.

This is my core linear regression code (using momentum).

```
while True:
    # one training epoch

    # reset temp variables
    loss = 0
    b_grad = 0
    w_grad = np.zeros((x_row_num, x_col_num))

    # calculate loss and gradient over all training data
    for month in range(12):
        for hour in range(470):
            # extract x and y from training data
            # train_array_months is a list of 12 arrays
            # x will be an ?x9 numpy array. '?' depends on how we process training data
            x = train_array_months_processed[month][:, hour:hour+9]
            y_data = train_array_months[month][9, hour+9]

            # calculate y from x, w, b
            y_pred= einsum('ij,ij', x, w) + b

            # update loss
            diff = y_data - y_pred
            loss += diff * diff

            # update gradient of b and w
            mult = 2 * diff * (-1)
            b_grad += mult
            w_grad += mult * x  # 2 * diff * (-1) * x

    # calculate rms error
    loss = sqrt(loss/(12*470))

    # save model
    if total_epoch % 100 == 0:
        model_file_info = "epo" + str(total_epoch) + "_los" + str(loss)[:6]
        print(model_file_info)
        np.savez(MODEL_FOLDER + model_file_info, b=b, w=w, vb=vb, vw=vw, lr=lr, total_epoch=total_epoch)

        # stop training in some cases
        if loss < 6:
            break

    # update parameters
    vb = lr * b_grad + gamma * vb
    vw = lr * w_grad + gamma * vw
    b = b - vb
    w = w - vw
    total_epoch += 1

print('end of training')
```

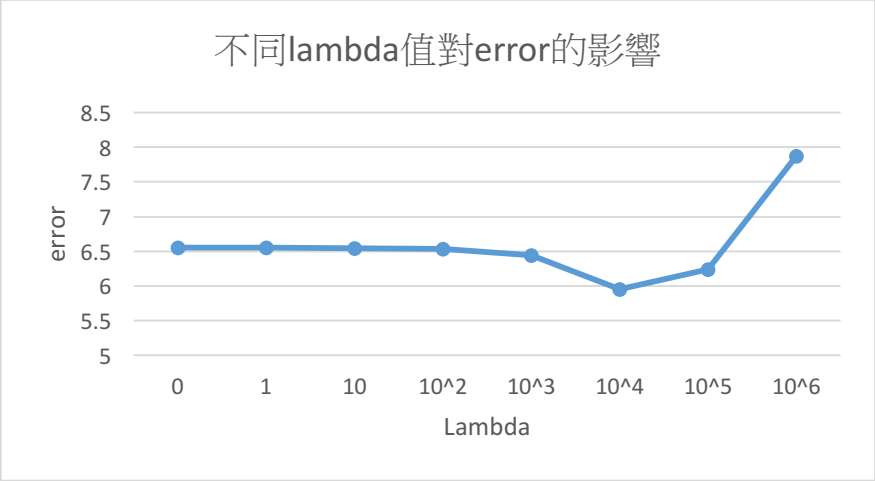
2. Describe your method.

- Linear Regression
  - 在本次作業中，我只有嘗試 Linear Regression 一個方法，沒有嘗試其他方法。但也嘗試了很多種最佳化的策略，以下一一介紹。
- Pre-process training data
  - 我嘗試過四種方式來構成 x：
    1. 只使用 PM2.5 的資料，成為 1*9 的陣列。
    2. 使用全部 data，成為 18 * 9 的矩陣。
    3. 從 18 列中抽出與 PM2.5 較相關的 11 列，成為 11 * 9 的矩陣。（我判斷相關性的方法是把每一列跟 PM2.5 那列算相關係數（Pearson r）的絕對值，再取出前 11 高的。）
    4. 承 2，再將 PM2.5 取平方形成新的列，成為 12 * 9 的矩陣。
  - 結果：
    - 用第 1 種方法就可以超過 baseline 了
    - 用第 2 種方法可以得到不錯的結果，但第 3、4 種方法更快
    - 我的最佳結果出自第 4 種方法。
- Gradient Descent Optimization
  - 我嘗試過以下幾種方式來更新 w, b
    1. 單純的根據 learning_rate * gradient
    2. 使用 momentum = 0.9
    3. 使用 Adagrad
  - 結果：
    - 只要設對 learning rate，用 1 就能得出不錯的結果。
    - 用 2 明顯比 1 快。
    - 根據我的測試結果，Adagrad 收斂的速度跟 momentum 差不多，但因為我 Adagrad 比較晚寫出來，所以我的最佳結果出自花比較多時間跑的 momentum。
- 結論
  - 我的最佳結果使用的是 Linear Regression，資料用與 PM2.5 最相關的 11 列（含 PM2.5 那一列）以及 PM2.5 的平方，更新參數的方法用 momentum，train 了 190 萬個 iterations。
  - 最佳的分數是 public score = 5.68834, private score = 6.81155，排名 37/348。

3. Discussion on regularization.

我另外寫了一個程式（hw_linear-regression_part-data_try-regularization）來測試 regularization 的效果。使用相同的初始參數。使用的 data 是前面所述的抽出 11 列的版本。使用的 learning rate 是 4 * 10^(-9)
測試的 lambda 的值有 0, 1, 10, …, 10^6，共 8 個。
我使用不同的 Lambda 值跑 1000 個 iterations 後得到以下結果

lambda	0	1	10	10^2	10^3	10^4	10^5	10^6
error	6.5504	6.5502	6.5459	6.5354	6.4380	5.9578	6.2418	7.8717

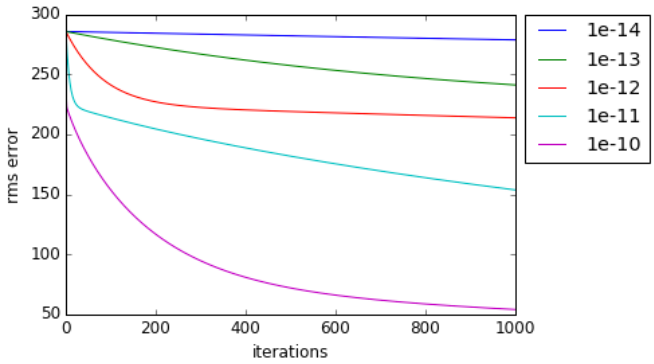


可見在相同的 iteration 次數之下，加上 regularization 以及適當的 lambda（10^4）後，可以加快收斂速度，得到更低的 error。大於或小於那個適當的 lambda 值，都會使 error 增加。原因是 Regularization 藉由在 Loss function 裡加上 $+\lambda\sum(w_i)^2$ 這一項，強迫壓低 w，當壓低的程度適中的時候，可以減少對 training data 的 overfit，但壓低太多（lambda 太大）會使得 w 過小，無法正確走到最佳的 w。

4. Discussion on learning rate.

我另外寫了一個程式（hw_linear-regression_all-data_try-lr.ipynb）來測試適合的 Learning Rate，x 使用 18*9 的 matrix，沒有做額外處理。
測試的 Learning Rate 有 1, 10^{-1}, …, 10^{-14}，共 15 個。
測試結果是：

- 當 Learning Rate >= 10^{-9}，每更新一次 w, b，error 就會不斷擴大，無法收斂。這部分沒有畫在下圖中
- 當 Learning Rate <= 10^{-10}，error 可以收斂且 10^{-10} 的收斂速度最快，且 Learning Rate 越小，收斂的越慢，如下圖



可見對於原始資料而言，最適合的起始 Learning Rate 在 10^{-10} 附近。
值得注意的是，Learning Rate 只要低於會發散的臨界值（10^{-9}），就會直接到接近最佳的 Learning Rate（10^{-10} 左右）。