# ML2016 HW1

R04942070 吳冠融

## 1. Linear regression function by Gradient Descent.

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

1. **while** True:
2. # one training epoch
4. # reset temp variables
5. loss = 0
6. b\_grad = 0
7. w\_grad = np.zeros((x\_row\_num, x\_col\_num))
9. # calculate loss and gradient over all training data
10. **for** month **in** range(12):
11. **for** hour **in** range(470):
12. # extract x and y from training data
13. # train\_array\_months is a list of 12 arrays
14. # x will be an ?x9 numpy array. '?' depends on how we process training data
15. x = train\_array\_months\_processed[month][:, hour:hour+9]
16. y\_data = train\_array\_months[month][9, hour+9]
18. # calculate y from x, w, b
19. y\_pred= einsum('ij,ij', x, w) + b
21. # update loss
22. diff = y\_data - y\_pred
23. loss += diff \* diff
25. # update gradient of b and w
26. mult = 2 \* diff \* (-1)
27. b\_grad += mult
28. w\_grad += mult \* x  # 2 \* diff \* (-1) \* x
30. # calculate rms error
31. loss = sqrt(loss/(12\*470))
33. # save model
34. **if** total\_epoch % 100 == 0:
35. model\_file\_info = "epo" + str(total\_epoch) + "\_los" + str(loss)[:6]
36. **print**(model\_file\_info)
37. np.savez(MODEL\_FOLDER + model\_file\_info, b=b, w=w, vb=vb, vw=vw, lr=lr, total\_epoch=total\_epoch)
39. # stop training in some cases
40. **if** loss < 6:
41. **break**
43. # update parameters
44. vb = lr \* b\_grad + gamma \* vb
45. vw = lr \* w\_grad + gamma \* vw
46. b = b - vb
47. w = w - vw
48. total\_epoch += 1
50. **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。

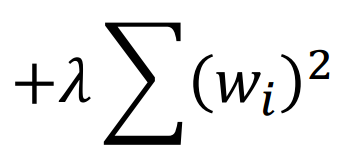
## 3. Discussion on regularization.

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, …, 106，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）。