

ML2016 HW1 report

b03902071 葉奕廷

Linear regression function by Gradient Descent

在 linear regression 中我使用最直觀的 loss function 加上 regularization $L = \text{sum of } (y - \hat{y})^2 + 100 * \text{sum of } (w)^2, y = w * x$ 。所有 training data 在讀檔時我先將其存在一個18維的 list 叫 feature，舉例來說如果我想取 pm2.5 的資料的話便會去取 `feature[9][index]`，index 代表是第幾小時。還有一些常數我定義在 `constants.py` 與 `bestconstants.py`，在這邊我簡述一下這些常數的意義

`hours`:代表取要預測的時間的前幾小時當作 feature

`costsum`:在算loss function時要算幾個 $(y - \hat{y})^2$ ，因為test data 有 240 筆所以到後面都設 $240 * 10$ (*10 是為了 code 比較好看)

`loopnum`:要跑幾個 epoch，每個 epoch 中會用第1~`costsum`筆, 第2~`costsum`+1筆... 去做 gradient descent

```
while loop != loopnum:
    for cnt in range(len(feature[0])-costsum):
        w,b,g,gb,loss=adagrad(feature, w, b, g, gb, cnt)
```

`usefeature`:要使用哪些 feature，每個 feature 由其在 csv 檔中排的次序代表，如 AMB_TEMP 是 0

詳細的 code 如下

```
def adagrad(feature, w, b, g, gb, cnt):
    dw = np.zeros((len(usefeature),hours), dtype=np.float64)
    db = 0

    loss = 0 # for early stop and observation
    maxcnt = cnt + costsum

    while(cnt != maxcnt ):
        yhat = feature[9][cnt+hours] #real pm2.5
        x = np.array([feature[i][cnt:cnt+hours] for i in usefeature], dtype=np.float64) #feature of pm2.5
        y = np.sum(x * w) + b #count estimated value
        loss += (y-yhat)**2
        for i in range(len(usefeature)):
            for j in range(hours):
                dw[i,j] += 2*(y-yhat)*x[i,j] #count (dL/dw)
            db += 2*(y-yhat) #count (dL/db)
            cnt += 10

        dw = dw + 2*100*w #regularization

        #adagrad part
        g = g + (dw ** 2)
        gb = gb + (db ** 2)

        deltaw = (1/np.sqrt(g))*dw*-1
        deltab = (1/np.sqrt(gb))*db*-1
        w = w + deltaw
        b = b + deltab

    return (w , b, g, gb, loss )
```

Method Details

Feature

首先經過實驗可以簡單地發現一次將 18 種 feature * 9 個小時全部丟下去 train 結果會有點爛，所以如何取有用的 feature 成為了這次最重要的課題。首先 pm2.5 當然是最重要的 feature，接著根據行政院環保署，SO₂、NO_x、VOCs 與 NH₃ 等前驅物(維基說是硫酸鹽、硝酸鹽及銨鹽)蠻有關係的。以下是一些實驗後的數據(實驗有將 train data 分成 6,6,8 天去 cross validation 和 kaggle score，為了方便下面是用 kaggle score做比較)，learning rate 調整的部分全是用 adagrad

kagglescore	hours	usefeature
6.52	9	all
6.23	5	pm2.5
5.97	9	pm2.5
5.95	7	pm2.5
5.76	7	NO2+pm2.5
5.79	7	NO2+NOx+pm2.5+SO2
5.77	7	NO2+NOx+pm2.5

可以發現取 7 小時似乎是比 9 小時來的有用一點的 (7 小時主要是在 cross validation 時表現較好)，然後加進 NO2 可以明顯的改善只使用 pm2.5 的 score，然而再加其他 feature 並不見什麼改善，看來是出現了 overfitting。所以最後我的 linear regression 是用 7 小時，NO2 + pm2.5 下去 train。

Early stop

每次做 gradient descent 時我都會去保存其 loss function 的值，並在每個 epoch 跑完時去計算 loss function 的平均值，若是在這 epoch 的平均值並未比上一個 epoch 的平均值低便提早停止，主要是可以省時間。

Learning rate

我實作了四種方法---sgd, adagrad, adadelta, adam，然而我發現這些做法儘管在一開始的 loss 減少速度有明顯的差距，在最後收斂到的 loss 卻是差不多的。猜想這應該是因為取的 feature 種類與我選擇的 linear regression model $y=w*x$ 的 bias 造成的，所以在最後我選擇使用簡單的 adagrad 以減少運算時間。

```
#adagrad part
g = g + (dw ** 2)
gb = gb + (db ** 2)

deltaw = (1/np.sqrt(g))*dw*-1
deltab = (1/np.sqrt(gb))*db*-1
w = w + deltaw
b = b + deltab
```

Regularization

以在 kaggle 上的分數來看，加入 regularization 在 feature 取的較多時較有用。然而儘管 regularization 能發揮更大作用，取到沒用的 feature 去 train 會讓 score 變差更多，所以我在最後是決定使用較少的 feature 去 train。regularization 在 feature 少時仍是可以帶來一些改善(在 cross validation 上較顯著)，但在 kaggle score 上似乎並不會因為有了 regularization 而有了突破性的改變。

Initialization

我將 w,b 都 initialize 為 1。其他像是用 normal distribution 去 random 或是將越靠近要預測的時間的 feature 的 w 初始值設比較大都有做過嘗試，但是在最後結果上並沒有什麼大差別。

Normalization

我曾經嘗試將使用的 feature 先 normalize，但是結果卻使 score 變差了。我想是因為最重要的 feature pm2.5 在資料中原本就是值相對較大的欄位，將其值和其他應該沒那麼重要且值較小的氣體分子一起 normalize 到 mean = 1, variance = 0 反而會使 pm2.5 的影響變小，導致預測變得比較不精確，所以在最後我放棄了 normalization。

Other method

在衝 kaggle 排名的部分，我試著使用了 linear regression 的 closed form solution。

$$w = (X^T X + \lambda I)^{-1} X^T y$$

```
def closedform(feature):
    tmp = [cnt*10 for cnt in range(len(feature[0])/10 )]
    featuretmp = [[] for i in tmp]
    for i in range(len(feature[0])/10 ):
        for j in usefeature:
            featuretmp[i] += feature[j][i*10:i*10+hours]
    x = np.array(featuretmp, dtype=np.float64)
    y = np.array([ feature[9][cnt+hours] for cnt in tmp ], dtype=np.float64)

    w = (inv( (np.transpose(x)).dot(x) + 1000 ).dot(np.transpose(x))).dot(y)
    return w
```

實作:

因為是利用線代的方法直接求出答案，所以不需要擔心 gradient descent 可能發生的卡在 saddle point, local minimum 等問題，但經過實驗發現也不是因為這樣就 feature 越多越好，還是要經過一些挑選。

我發現取前七小時的NO2, NOx, O3, pm2.5, SO2 可以在 kaggle score 上達到 5.57 左右，是個蠻顯著的進步。

Kaggle score

這次作業我犯了一項很大的錯誤，就是在比較 model 好壞時儘管知道不可以還是不顧 cross validation 的結果而不信邪的去迎合了 kaggle public set 測出來的 score，所以我在最後的 private set 排名一口氣掉了 200 多名。根據我其他次 submit 的 private score 我發現應該是 public set "太好猜" 了，只需要特定兩三個 feature 就可以預測得還不錯，然而 private set 的分布與 public set 差很多且更為複雜，我所使用的參數較少的 model 便造成了 underfitting。這次的失敗讓我更體會到 cross validation 的重要還有知道了千萬千萬不要相信 kaggle 而要相信自己手上的 data。