在 kaggle_best 上是用 linear regression 進行 PM2.5 的預測

先預設幾個需要用到的參數，如 w0、w1=九個值等於 0 的 array，用來儲存 x 一次以及 x 二次前九個小時的 w 值、x0=九個值等於 1 的 array，用來儲存前九個小時的 x 值，還有 bias=20、learn=0.000000001 等等。

接著把所有豐原的資料利用 genfromtxt 讀進來，再利用 append 處理成只有 PM2.5 值的一維陣列 (crr)。

arr=np.genfromtxt("train.csv",delimiter = ',',skip_header = 1,usecols = range(3,27))

brr=arr[9::18] (從第 9 行開始，每隔 18 行取一次，也就是取了所有的 PM2.5 的資料)

接下來就是 training 的部分，利用 while 迴圈設定要 train 的次數。在這個 while 迴圈裡，把前面九個 PM2.5 的值利用 x0 陣列儲存起來，並把第十個數值儲存在 y0。

在 for 迴圈裡，range 是從 0 到最後第十個 PM2.5 的資料

x0 = crr[i+0:i+9]，y0 = crr[i+9]。crr 是一個包含所有 PM2.5 資料的一維陣列。

而在這份作業中，用到的 linear regression 的 function 是 y = bias + w0*x0 + w1*x0^2。所以 gradient descent 的微分值分別是：a0 = 2*(y0-(bias+sum(w0*x0)+sum(w1*x0**2)))*x0；a1 -= 2*(y0-(bias+sum(w0*x0)+sum(w1*x0**2)))*2*x0；b0 -=2*(y0(bias+sum(w0*x0)+sum(w1*x0**2)))。其中，a0 是 loss function 對 w0 的偏微分，a1 是 loss function 對 w1 的偏微分，b0 則是 loss function 對 bias 的偏微分。

在每讀取九個值後，就把 a0、a1、b0 的值各自累加起來。

讀取完全部的值(5760 個之後)就針對 a0、a1、b0 做修正。修正方式分別為

a0[j]=a0[j]+2*lamda1*w0[j]、a1[j]=a1[j]+2*lamda2*w1[j]、w0[j] = w0[j]-learn*a0[j]/np.sqrt(delta[j])、w1[j] = w1[j]-learn*a1[j]、bias = bias - learn*b0。其中，lamda1 跟 2 都是 100、delta 則是在 while 裡，每次 delta += a0**2。至於 w0、w1、bias 則是讓他根據 learning rate 做調整，當 a0、a1、b0 微分大於 0 時，斜率正，代表 w0、w1、bias 的值要小一點，反之亦然，所以不管微分>或是<0 都是用減號。

在做完一次 while 迴圈時，k+=1、並且 print 出 error (err = abs(y0 – y)/5750)，會發現每次 print 出來的 error 值都會往下降。下降的程度會隨 learning rate 變大而變快，但是太大的 learning rate 又會使 error 快速上升。因此要找到一個適當的值。後來用 adagrad 使 learning rate 會隨著 while 迴圈而改變。

arr=np.genfromtxt("train.csv",delimiter = ',',skip_header = 1,usecols = range(3,27))

linear regression function code

```
while k<1000 :
        err=0.0
        a0 =np.zeros(9)
```

```
b0 =0.0
for i in range(0,len(crr)-10):
        x0 = crr[i+0:i+9]
        if (i%10) == 9:
                y0 = crr[i+9]
        else :
                a0 -= 2*(y0-(bias+sum(w0*x0)+sum(w1*x0**2)))*x0
                a1 -= 2*(y0-(bias+sum(w0*x0)+sum(w1*x0**2)))*2*x0
                b0 -= 2*(y0-(bias+sum(w0*x0)+sum(w1*x0**2)))
        err += abs((y0-(bias+sum(w0*x0)+sum(w1*x0**2))))/5750
delta += a0**2
for j in range(0,9):
        a0[j]=a0[j]+2*lamda1*w0[j]
        a1[j]=a1[j]+2*lamda2*w1[j]
        w0[j] = w0[j]-learn*a0[j]/np.sqrt(delta[j])
        w1[j] = w1[j]-learn*a1[j]
        bias = bias - learn*b0
k+=1
print(k,"err",err)
```

Linear regression function by Gradient Descent.

這次 HW1 的 linear regression function 是 y = bias + w0*x0 + w1*x0^2。所以 gradient descent 的微分值分別是：a0 = 2*(y0-(bias+sum(w0*x0)+sum(w1*x0**2)))*x0；a1 -= 2*(y0-(bias+sum(w0*x0)+sum(w1*x0**2)))*2*x0；b0 -=2*(y0(bias+sum(w0*x0)+sum(w1*x0**2)))。其中，a0 是 loss function 對 w0 的偏微分，a1 是 loss function 對 w1 的偏微分，b0 則是 loss function 對 bias 的偏微分。

在每讀取九個值後，就把 a0、a1、b0 的值各自累加起來。

讀取完全部的值(5760 個之後)就針對 a0、a1、b0 做修正。修正方式分別為

a0[j]=a0[j]+2*lamda1*w0[j]、a1[j]=a1[j]+2*lamda2*w1[j]、w0[j] = w0[j]-learn*a0[j]/np.sqrt(delta[j])、w1[j] = w1[j]-learn*a1[j]、bias = bias - learn*b0。其中，lamda1 跟 2 都是 100、delta 則是在 while 裡，每次 delta += a0**2。至於 w0、w1、bias 則是讓它根據 learning rate 做調整，當 a0、a1、b0 微分大於 0 時，斜率正，代表 w0、w1、bias 的值要小一點，反之亦然，所以不管微分>或是<0 都是用減號。

目的都是希望微分值 a0、a1、b0 趨近於 0。

Discussion on regularization

Regularization 就是把 w0 裡的每個值加進去 loss function 裡考慮 (L = (y-yo)^2+lamda*sum(wi^2))。因為 loss function 要越小越好，因此我們希望 wi 的值也越小越好，因為越小的 wi 值代表預測的數據不會因為輸入的 test_x 不同而差異很大。這裡的 lamda 值越大，預測的 model function 也會越 smooth，如果有 noise 或是 variance 太大的值，這個 modle 不會有太大的影響。

Discussion on learning rate.

Learning rate 是調整 w0、w1、bias 的參數，讓微分值能更趨近於 0，這樣 loss 就會到 local minimum。太小的 learning rate 代表調整的慢，使 loss 到 local minimum 的速度就慢，但是太大的話，則有可能使它一次就跨越了 local minimum 而使 loss 增加爆掉。