

# Machine Learning

## HW2

r05922018

黃柏智

### 1. Logistic Regression.

#### (a) function

我用的 cost function 是用 cross entropy:  $C(f(x^n), \hat{y}^n) = -[\hat{y}^n \ln f(x^n) + (1 - \hat{y}^n) \ln(1 - f(x^n))]$

而 weights 遞減的速度則為:  $w_i \leftarrow w_i - \eta \sum_n -(\hat{y}^n - f_{w,b}(x^n)) x_i^n$

#### (b) features

首先講一下全部可用的 features 有哪些，每一筆 data 共有 57 個 features，我試過只用這 57 個 features 的一次方，以及這 57 個 feature 的一次方配上二次方，一次方上傳到 kaggle 的分數，大約都落在 0.91 ~ 0.93 之間。而二次方的成績就很慘了，不論怎麼調參數，都無法突破 0.75。根據上次作業的經驗，使用高次方時，若有 overfitting，準確率通常不會下降這麼嚴重，推測是因為這次的 data 中，數值為 0 的 feature 太多，因此使用 high order 或 corelation 時，只會產生更多的 0，這很容易讓準確率嚴重下跌。

#### (c) implementation

一開始，我會先根據我想使用的 order，從原始資料中生成我要的 training data。接著進行 logistic regression。首先會 initial 一組 weight，在每一個 iteration 中，首先計算 gradient，接著根據當時的 learning rate 來改變 weights。而 learning rate 改變的機制是這樣，當連續 10 個 iterations cost 遞減速率小於 1%，learning rate 就會變 1.1 倍，當某個 iteration 的 cost 不減反增時，learning rate 便砍半。全部 iteration 結束後得出的 weight 便是 model。

#### (d) regularization

事實上我的公式除了老師投影片上列的那條之外，還加上了 regularization，就是在原本的 cost function 後面加上  $(\lambda/2m) * (w*w)$ ， $\lambda$  是 regularization rate， $m$  是 train data 總筆數， $w$  是 weight。以下是不同  $\lambda$  值跑 200 萬圈的結果：

lambda	1	5	10	15	20	25	30
accuracy	0.92000	0.91667	0.92667	0.91667	0.91333	0.91000	0.91000

縱使初始值為隨機，但是大致可以看出， $\lambda$  的影響並不會很大，而我 logistic regression 中最好的結果是在  $\lambda = 10$  時得到的。

#### (e) other discussion

此外，我還有試著做 feature scaling，用下面的公式分別對 train data & test data 做處理：

$X = (X - \text{np.mean}(X, \text{axis}=0)) / (\text{np.std}(X, \text{axis}=0))$

但不知是我實作錯誤，抑或是這份 training set 不適合用此法，加上 feature scaling 後的 performance 很差，在 kaggle 上的分數只比亂猜得到的分數好一點。

### 2. Neural Network.

我的第二個方法選擇了 Neural Network。

#### (a) Neuron

我採用的神經元，便是在 logistic regression 中使用的 sigmoid 函式。每個 neuron 所輸出的值，都是  $1.0 / (1.0 + \text{np.exp}(-z))$ ，其中  $z = wx+b$ 。

## (b) features & parameters

我使用的 feature 跟 logistic regression 一樣，是每一筆 train data 的 57 個 features，只使用一次方。而能改變的 parameters 有 layer 數、每層 layer 的 neuron 數、iteration 數、mini-batch 大小，以及 splitPercent。splitPercent 是指在給定的 training data 中，要有多少比例的 data 不參與 train 的過程。這些 data 是用來計算 training 後的 accuracy，我會用這個 accuracy 判斷一組參數的好壞，並當作是否上傳 kaggle 的依據。

## (c) implementation

根據選定的 order 生成 training data 後，拿出一小部分用來驗證，其他丟進 NN 中。首先依照 mini-batch 的大小切割資料成很多個 slice，對每一段 slice 做 back-propagation，求出 NN 中每一層 weight & bias 的 gradient，然後更新每一個 layer 的 weight & bias。每一個 iteration 結束前都會用當初選的驗證資料來做預測，所以每個 iteration 都會得出一個 accuracy。在跑完所有 iterations 後，會產生擁有最高準確率的 iteration，用該 iteration 當下的 weight & bias 當成最後輸出的 model。產生 model 後，將 testing data 丟進 NN，進行 feedforward，得到最終預測結果。

## (d) experiment result

以下列出一些實驗後得出的結論：

== feature、parameters 部分 ==

- 經過各種測試之後，同一組參數，splitPercent = 10% 時準確率通常較高。
- 用分割出來的 data 來預測 model 準確率，所得到的分數跟上傳至 kaggle 的分數高度一致，差距幾乎不會超過 1%，只有在 model 中得到的準確率超過 96% 的時候，上傳 kaggle 的誤差才有可能上升至將近 2%。
- 越深的 NN，通常要配小一點的 learning rate 才容易 train 出好結果。
- 使用超過一次方的 order 當做 input layer 的 feature，效果極差，應該是 overfitting。

== iteration 數、neuron 數部分 ==

- 同樣的參數，iteration 越多，通常會進步，以一個三層的 NN 為例，跑 1000 圈正確率 88.5%，跑 10000 圈正確率 92.25%。
- layer 越多時，iteration 也必須相對增加才能得到差不多的結果。若深度增加，iteration 不足時，結果會更差。例如三層以及四層 NN 都跑 5000 圈，後者準確率少 6%。
- 有方法能讓跑同樣的 iteration 數時，準確率提升。假設現在跑 50000 個 iterations，第一種方法是產生驗證 data 後，一直用這組 data 來做準確率預測。第二種方法是把 50000 個 iterations 分成 10 次來跑，一次只跑 5000 個 iterations，但每次跑之前會重新選定用來做準確率預測的 data。這樣能避免整個 model 一直向 train data 的某一小部份靠攏。以一個三層，每層 neuron 數分別是 57、40、2 的 NN 為例，法一準確率為 92.17%，法二的準確率提升至 96.5%，在 kaggle 上的分數也差了快 5%，差距十分驚人。
- 在 learning rate 不變的情況下，各種參數的 NN，似乎都在前 10000 圈就產生了最佳解。或許因為如此，剛剛提到的法二，效果才會如此顯著。
- input layer 有 57 個 neurons (因為有 57 個 features)，我試著讓 NN 中某一層的 neurons 數量遠超過 57 (e.g. 500)，上傳到 kaggle 的分數很差，可能是 overfitting。

== available future works ==

- 把 Adagrad 加進 NN 中，動態改變 learning rate。
- 因為這次的 training data 並不算多，如果能把預測完的 testing data 加回到 model，做 feedback，或許有進步的機會。
- 實驗結果放在：<https://www.csie.ntu.edu.tw/~r05922018/ML/hw2/>