ML2016 HW2 Report

R05921040 謝友恆

- Objective

  由抽取過之 feature 使用 logistic regression 進行 spam classification。

- Data Preprocessing

  為了使每一維度的資料更新速率接近，先將各維度資料 normalize。

- Logistic Regression

  Logistic Regression 中要最小化的誤差值定義是:

  $$\mathrm{E(w)} = \Sigma - (\mathrm{y^n} \times \ln \sigma(\mathrm{x^n} \cdot \mathrm{w}) + (1 - y^n) \times \ln(1 - \sigma(x^n \cdot w)))$$

  經過微分得到 gradient 值為:

  $$\nabla \mathrm{E(w)} = \Sigma(y^n - \sigma(x^n \cdot w)) \times x^n$$

  Python Code 如下:

```
def error(x, w, b, y):
    return (-(y*np.log(logistic(x.dot(w)+b)+eta) + (1-y)*np.log(1-logistic(x.dot(w)+b)+eta))).mean()

def gradDes(x, w, b, y):
    return (-(y-logistic(x.dot(w)+b)).transpose()*x.transpose()).mean(axis = 1)

def gradDesb(x, w, b, y):
    return (-(y-logistic(x.dot(w)+b))).mean()
```

  另外使用 Adagrad 加速更新速度，整體 code 如下:

```
#random columns for stochastic gradient descent
r = np.random.randint(4001,size=200)
#calculate gradient descent for w and b
gdw = gradDes(train_data[r,:], w, b, y[r])
gdb = gradDesb(train_data[r,:], w, b, y[r])
#adagrad for w and b
gdwSum = gdwSum + np.square(gdw)
gdbSum = gdbSum + np.square(gdb)
#update parameter value
w = w - alpha * gdw / np.sqrt(gdwSum+eta).astype(float)
b = b - alpha * gdb / np.sqrt(gdbSum+eta).astype(float)
```

- Other method

  - Neural Network

  一開始選用了一層 hidden layer 的 NN，其誤差項的定義跟 logistic 版本一樣，而 gradient 項∇L(w)則透過 backpropagation 修正為:

  $$\delta_j = \begin{cases} (o_j - y_j) & if\ \ j\ \ is\ \ at\ \ output\ \ layer \\ (\Sigma_L \delta_l w_{jl}) o_j (1 - o_j) & if\ \ j\ \ is\ \ hidden\ \ layer \end{cases}$$

  $$\nabla \mathrm{E}(\mathrm{w_{ij}}) = \mathrm{o_i} \times \delta_\mathrm{j}$$

  其中$\mathrm{L} = \mathrm{the\ \ set\ \ of\ \ all\ \ neuron\ \ receives\ \ input\ \ from\ \ j}$

  Python 的 NN code 如下:

```
#calculate gradient descent for w
r = np.random.randint(dataLen, size = dataNum)
o_input = logistic(train_data[r,:].dot(w_input[i].T))
o = logistic(np.column_stack((o_input, np.ones(dataNum))).dot(w[i]))
#delta of the output layer
delta_o = (o-y[r])
#back propagation of output layer
gdw = (np.column_stack((o_input, np.ones(dataNum))).T * delta_o.T).mean(axis = 1)
gdwSum = gdwSum + np.square(gdw)
#back propagation of hidden layer
gdwi = ((train_data[r,:].T).dot(delta_o[:,None].dot(w[i][:nNum,None].T) * o_input * ( 1 - o_input)))/dataNum
gdwi = gdwi.T
gdwiSum = gdwiSum + np.square(gdwi)
#update weight value
w[i] = w[i] - alpha * gdw / np.sqrt(gdwSum +eta)
w_input[i] = w_input[i] - alpha * gdwi / np.sqrt(gdwiSum + eta)
```

然而用 NN 之後並不是很能抓到 learning rate、neuron number、iteration number 的設定，試過用 cross validation 來找最適當的範圍，但在 public set 上表現也不是很理想。因此便改用下個方法。

- Neuron Network with Boosting

Boosting 是將 sample 分割成不同部分，每部分各自的得到一個 model，再透過這些 model 的 voting 來決定最終的結果，在這次的實作由於有用 stochastic 的方式同時 sample 數量也不多的情況下，便沒有切割 sample。而對於每個不同的 model 我們隨機給予他使用的 neuron 數量(2 到 25 之間)，以及 iteration number(12000~18000)還有 learning rate(1-2)。由於希望將 training time 控制在十分鐘以內，設定 model 數量為 25，而使用這個方式在 public 與 private set 上都能有相對好一點的效果。

- Discussion
  - Logistic Regression vs NN

Logistic Regression 其實就是只有 output layer 的 NN，因此就表達模型的能力而言 NN 應該是超越 Logistic Regression 的。因此 NN 可以在 training error 上大幅度超越 Logistic Regression，NN 甚至可以將 cross entropy 壓到 0.01 以下，但這樣意義並不大，因為這次的模型相對簡單，這樣做反而會 overfit。因此使用 NN 最大的問題是如何找到適當的停止點，就此而言使用 cross validation 應該是不錯的方法。

  - Normalization

這次使用了兩種 normalization 的方法，一種是 L2 norm，讓整個 feature vector 的平方和等於一，另一種是減去平均，並除以標準差，兩者相比較的結果後者 normalize 的方式相對好一些，推測是因為 L2 norm 會受最大最小值影響，另一種方式並不會。

  - Efficiency

由於 model 較為簡單，logistic regression 執行的速度較 NN 快了十倍左右，而 NN 執行速度則正比於使用的 neuron 數量。但整體而言執行時間皆不會太久，logistic 可以在十幾秒內完成十萬個 iteration，而 NN 單一 model 需要時間約為 15 秒左右(15000 iteration)。

- Boosting

  會想使用 boosting 主要原因是若是用單一的 nn，則出來結果跟 neuron 數量，iteration 數量，learning rate 有直接關係，同時很難區分是否不小心 overfit。而慢慢調參數的作法過於耗時，因此以多次取平均的方式希望找到較好的結果。而在實際實驗中的確如此。但 Boosting 並不能超越選用 model(NN)的極限，只是讓結果較不受實驗過程中的雜訊影響。

討論對象: b01901001, b01901169