題目: **Outbrain** 廣告點擊預測

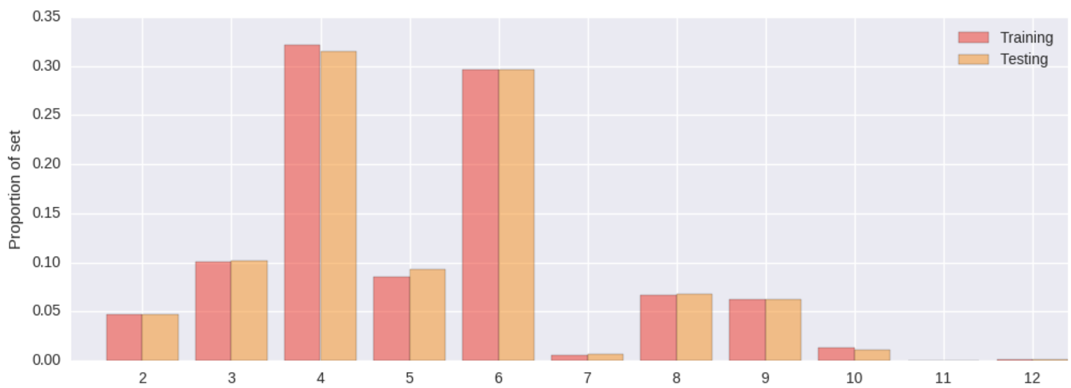
Q1. 隊伍資訊

隊名：NTU_r04945041_願便斗託神以便便不絕之效… (全名太長恕略 XD)
隊員[1]：R04922081 江宗臻、R04922129 彭馨儀、R04945041 蔡智晴、R05922038 黃郁庭
分工：江宗臻研究 FTRL 並實作之；彭馨儀研究 FM 和 libFM 如何使用；蔡智晴做 EDA 資料分析，研究 feature 怎麼抓；黃郁庭研究 FFM 和 libffm 如何使用。最後大家共同用 FFM 和 FTRL，平行嘗試不同 features 和參數的效果。

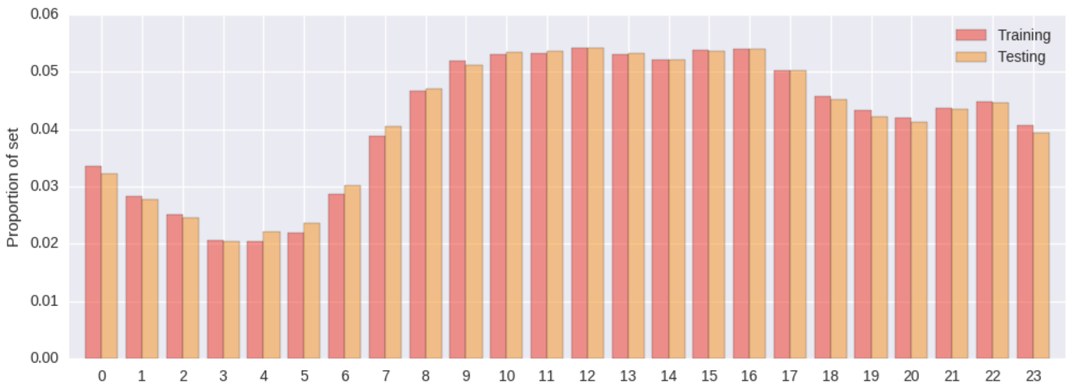
Q2. 前處理 **/** 特徵工程

抓 feature 的方法，最重要的就是用好的 validation set 驗證 feature 好壞。好的 validation 其資料分佈必須能代表 testing set。因此我們先來對 training 和 testing 資料做 EDA 分析。

我們嘗試比較 training 和 testing 的許多特徵分佈，例如 (1) 在每個 `display_id` 中包含的廣告數量，還有 (2) 一天 24 小時中資料的分佈，以及 (3) 資料在 15 天的分佈。下面三張圖表是我們實驗的結果，您可以執行 `edaplot.py` 產生它們。可以發現 testing 和 training 大致分佈都很一致 (其實還有比較過其它特徵，族繁不及備載，但也都很一致)，僅有 (3) 的差異特別懸殊！

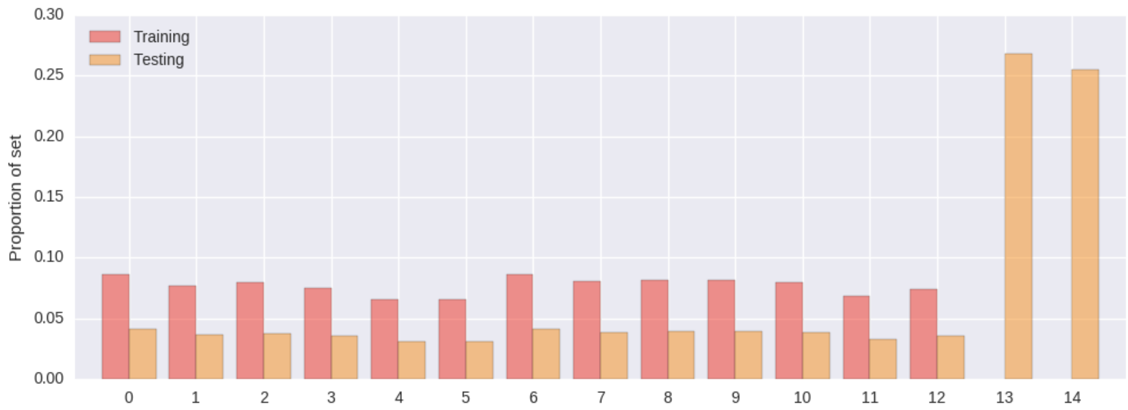


在每個 display 中的廣告數目



一天 24 小時資料分佈

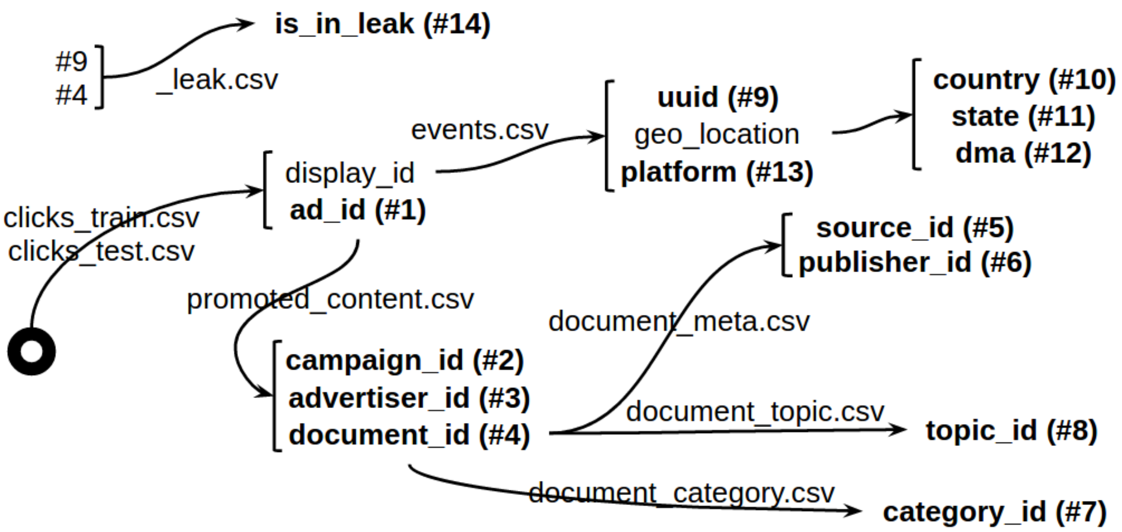
[1] 依學號 alphabetically 排序



從 2016/6/14 起算，連續 15 天的資料分佈

所有資料有連續 15 天的分佈，但我們發現 training set 只包含前面 13 天；然而 testing set 包含全部 15 天，且最後 2 天佔超過 testing 比例的 50%！因此在切 validation 時，盡可能讓時間越晚比例越多。我們的切法是把 training 最後兩天 (即第 11、12 天) 取全部，以及第 0 到 10 天各取 10% 當成 validation set (程式為 `splitvalid.py`)。

使用 validation 驗證後 (加入該特徵後若驗證分數上升，則留下) 共取了 14 個 features 如下圖。圖中線條表示這些 features 是從哪些 `.csv` 抓出來的。取 feature 的程式為 `features.py`。



值得一提的是 `_leak.csv` 這個檔案，它是我們自己生成的，欄位範例如下表。

document_id	uuid
446701	9416d88ee717a7 d8822f49e11a4e …

意義是該 document 有哪些用戶看過。該檔案生成方法，是從 `promoted_content.csv` 的 `document_id` 和 `page_view.csv` 的 `uuid` 和 `document_id` 做比對，查詢所有廣告的連結頁面 (landing page) 曾有哪些用戶來過。詳細流程請看 `leak.py`。因為 `page_view.csv` 也包含 testing set 的資料，若廣告連結頁面 user 根本沒看過，表示 user 絕對沒點擊過它！事實上，這是不該有的資訊！實務上對於未知資料，`page_view.csv` 不可能有此資訊，但在此題目中 Outbrain 不小心洩密 (leak) 給我們。

因為 feature 有許多都是 ID，把 ID 本身的值放進去當 feature 不大有意義。所以我們使用 one-hot vector 來表示。也就是若所有 feature ID 有 N 種可能，則使用一個維度為 N 的 vector，每個 ID 對應到一個位置，若出現該 ID 則設為 1。所以每筆資料進來時，14 個 features 會對應到向量的 14 個位置，故 N 維向量只有 14 個 1，其它為 0。

使用 one-hot encoding 後維度會非常大，在此題目中，依 feature 數量不同，維度大約在 10,000,000 到 50,000,000 間分佈。因此，餵給 model 做訓練時，給它 1 的位置 (index) 即可，即 spare 表示法。這種 one-hot encoding feature 除了維度大、很多 0 外，很多位置 training 的次數並不多，也就是稀疏性 (sparse)，這也是所有預測廣告點擊問題的共同性質。因此我們挑選的 models 必須善於處理這類性質的問題。

Q3. 模型 (FM + FFM + FTRL)

我們嘗試過 FFM 和 FTRL-Proximal 兩種方法，因為它們都善於處理 sparse 資料。FFM (Field-aware Factorization Machine) 是從 FM (Factorization Machine) 修改而來的，我們先花點篇幅介紹 FM 的原理，然後稍微修改即可得 FFM。

FM (Factorization Machine)[2]

令 feature 向量為 $x = (x_1, x_2, ..., x_n)^T$，傳統 linear regression 模型為[3]

$$y(x) = w_0 + \sum_{i=1}^{n} w_i x_i$$

其中 w_0 和 $w = (w_1, w_2, ..., w_n)^T$ 為模型參數，在我們問題中 n 就是 one-hot vector 維度。在此模型下，不同 feature 彼此是不相干的。為了把 feature 彼此交互關係 (interaction) 也考慮進來，可改寫成 degree-2 polynomial (Poly2) 模型：

$$y(x) = w_0 + \sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} w_{i,j} x_i x_j \qquad (1)$$

雖然多了 $w_{i,j}$ 代表任兩 feature 間的關係，但這在 sparse 資料有個大缺點。舉例來說，考慮 `ad_id=9487` 和 `uuid=5566` 兩個 feature，它們有個 $w_{i,j}$ 代表它們之間的關係。但 training 資料中很有可能這兩者沒有同時出現過 (sparse 性質)，故對應的 $w_{i,j}$ 完全沒訓練過 (通常為 0)！若 testing 時同時出現這兩個 feature，則 $w_{i,j} x_i x_j$ 的值就完全沒意義！

FM 關鍵技巧在於，把 $w_{i,j}$ 改為 $v_i^T v_j$，其中 v_i 代表特徵 x_i 的 implicit vector。更精確地說，對每個特徵 x_i，給它一組參數 $v_i = (v_{i,1}, v_{i,2}, ..., v_{i,k})^T$，共 k 個參數。所以對於全部 n 個特徵，包含原始 linear model 的 w 我們總共有 $O(nk)$ 個參數：

[2] 原始論文: **http://ppt.cc/LeJX7**
[3] 在我們的分類問題需加上 sigmoid 等 logistic function。這可輕易修改，這邊忽略之。

$$\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \ldots \\ w_n \end{bmatrix}, V = \begin{bmatrix} \mathbf{v_1^T} \\ \mathbf{v_2^T} \\ \ldots \\ \mathbf{v_n^T} \end{bmatrix} = \begin{bmatrix} v_{1,1} & v_{1,2} & \ldots & v_{1,k} \\ v_{2,1} & v_{2,2} & \ldots & v_{2,k} \\ \ldots & \ldots & \ldots & \ldots \\ v_{n,1} & v_{n,2} & \ldots & v_{n,k} \end{bmatrix}$$

為何這能解決稀疏資料問題？因為 `ad_id=9487` 和 `uuid=5566` 只要各別有出現過 (不必同時出現在同一筆資料)，那它們分別的 v_i 和 v_j 就有訓練過，故 $(v_i^T v_j)\, x_i x_j$ 中的 $v_i^T v_j$ 就不會是 0。總結 FM 數學模型如下：

$$y(x) = w_0 + \sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (v_i^T v_j)\, x_i x_j \qquad (2)$$

相較 Poly2 模型，只差在最後 interaction 項。

為何叫 factorization machine？先來看式 (1) Poly2 的 interaction 權重 $w_{i,j}$，可寫成對稱矩陣 (稱為 interaction matrix)：

$$W = \begin{bmatrix} w_{1,1} & w_{1,2} & \ldots & w_{1,k} \\ w_{2,1} & w_{2,2} & \ldots & w_{2,k} \\ \ldots & \ldots & \ldots & \ldots \\ w_{n,1} & w_{n,2} & \ldots & w_{n,k} \end{bmatrix}$$

然而在 FM 中，進一步把 $w_{i,j}$ 分解為 $v_i^T v_j$，故 interaction 矩陣為：

$$\widehat{W} = VV^T = \begin{bmatrix} \mathbf{v_1^T} \\ \mathbf{v_2^T} \\ \ldots \\ \mathbf{v_n^T} \end{bmatrix} \begin{bmatrix} \mathbf{v_1} & \mathbf{v_2} & \ldots & \mathbf{v_n} \end{bmatrix} = \begin{bmatrix} \mathbf{v_1^T v_1} & \mathbf{v_1^T v_2} & \ldots & \mathbf{v_1^T v_n} \\ \mathbf{v_2^T v_1} & \mathbf{v_2^T v_2} & \ldots & \mathbf{v_2^T v_n} \\ \ldots & \ldots & \ldots & \ldots \\ \mathbf{v_n^T v_1} & \mathbf{v_n^T v_2} & \ldots & \mathbf{v_n^T v_n} \end{bmatrix}$$

也就是把 $\widehat{W}$ 分解 (factorization) 為 VV^T。

接著分析時間複雜度，式 (2) 乍看之下複雜度為：

$$[n + (n-1)] + \{\tfrac{n(n-1)}{2}[k + (k-1) \ + \ 2] + \tfrac{n(n-1)}{2} - 1\} = O(kn^2)$$

時間多花在 $\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (v_i^T v_j)\, x_i x_j$ 項。然而經過推導可改寫成 $O(kn)$ 的式子：

$$\begin{aligned} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (v_i^T v_j)\, x_i x_j &= \tfrac{1}{2}\Big(\sum_{i=1}^{n} \sum_{j=1}^{n} (v_i^T v_j)\, x_i x_j - \sum_{i=1}^{n} (v_i^T v_j)\, x_i x_j\Big) \\ &= \tfrac{1}{2}\Big(\sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{l=1}^{k} (v_{i,l} v_{j,l})\, x_i x_j - \sum_{i=1}^{n} \sum_{l=1}^{k} v_{i,l}^2 x_i^2\Big) \\ &= \tfrac{1}{2} \sum_{l=1}^{k} \Big(\sum_{i=1}^{n} v_{i,l} x_i \sum_{j=1}^{n} v_{j,l}\, x_j - \sum_{i=1}^{n} v_{i,l}^2 x_i^2\Big) \end{aligned}$$

$$= \frac{1}{2}\sum_{f=1}^{k}[(\sum_{j=1}^{n}(v_{j,f}x_j))^2 - \sum_{j=1}^{n} v_{j,f}^2 x_j^2]$$

然後用 stochastic gradient descent (SGD) 去更新參數，梯度如下：

$$\frac{\partial \hat{y}}{\partial \theta} = \begin{cases} 1, & \text{if } \theta = w_0 \\ x_i, & \text{if } \theta = w_i \\ x_i \sum_{j=1}^{n} v_{j,f} x_j - v_{i,f} x_i^2, & \text{if } \theta = v_{i,f} \end{cases}$$

總結 FM 的三點好處：

1. 藉由考慮特徵間的關係，FM 降低了 interaction 項參數訓練不充分的問題。
2. 提升模型預測能力，藉由 interaction 項，使得沒出現過的交叉特徵也有一定權重。
3. 巧妙簡化 interaction 項，讓複雜度從 $O(kn^2)$ 降成 $O(nk)$，提升學習速度。

FFM (Field-aware Factorization Machine)[4]

在數學模型上，FFM 和 FM 只有最後 interaction 項的差異：

$$y(x) = w_0 + \sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} (v_{i,f(j)}^T v_{j,f(i)})\, x_i x_j \qquad (3)$$

也就是把 $v_i^T v_j$ 改成 $v_{i,f(j)}^T v_{j,f(i)}$，其中的 $f(i)$ 代表 x_i 的 field。我們舉例說明什麼是 field。

前面提到我們抓了些廣告特徵 (如 `ad_id`、`campaign_id`、`advertiser_id`)，可以把它們放進同一個 field；然後一些關於 user 的特徵 (如 `uuid`、`geo_location`、`platform`)，也歸類到同一個 field。例如下表是對我們 14 個 features 一種可能的分法：

field_1							field_2				field_3		
#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12	#13	#14

總共 14 個特徵，分 3 個 fields。實際上應該要如何分，並沒有明確規則，只能靠對特徵的直覺 (例如重要程度差不多或性質相似的特徵可分在同個 field)，不斷嘗試各種組合的效果。

因此對每個特徵 x_i，需要的參數不是只有一個 implicit vector v_i，而有 $v_{i,f(1)}, v_{i,f(2)}, \ldots, v_{i,f(f)}$ 共 f 個 implicit vectors，其中 f 為 field 數目。所以 FM 可說是 FFM 的特例。直觀地說，$v_{i,f(j)}$ 不僅與特徵 x_i 相關，還把特徵 x_i 和 field $f(j)$ 的關係也考慮進來。

若 one-hot vector 維度為 n，前面提到 FM 參數有 $O(nk)$ 個，但 FFM 則有 $O(fnk)$ 個。時間複雜度方面，遺憾的是式 (3) 無法像式 (2) 那樣化簡，因此複雜度就和化簡前的式 (2) 一樣為 $O(kn^2)$。儘管如此，因考慮了特徵和其它 field 間的關係，其學習力又比 FM 強 (但也容易 overfitting，使用時要小心)。

[4] 原始論文: **http://ppt.cc/XmgZB**，作者是林智仁老師的學生，在 KDD Cup 比賽中發明它的，然後一舉拿下世界冠軍，並直接入取舉辦公司 Criteo，請跪 m(_ _)m

在實作中，我們直接用現成的 LIBFFM。在林智仁教授的實驗室過去的研究顯示，當特徵能從類別的表示方式以是或否的表現方式來表示時 (也就是說，每個特徵有 s 個維度，s 表示該特徵的定義域的大小，其中只有一個位置設為 1，其餘為 0。在 LIBFFM中，只有值為 1 的位置會被存下來)，使用 FFM 會有明顯的優勢；而當特徵的表示方式為數值時，用 FFM 的效果不一定會比其它模型還來的好。

LIBFFM 使用 logistic regression，更新參數的方法為 SGD。我們所求的模型 **w** (這裡的 **w** 與式 (3) 的 **w** 意涵不同，式 (3) **v** 參數在這用 **w** 表示，為簡化說明，linear terms 與 bias 在這裡先暫不提及。) 可用下列數學式來解 (optimization problem)：

$$\min_{\boldsymbol{w}} \quad \frac{\lambda}{2}\|\boldsymbol{w}\|_2^2 + \sum_{i=1}^{m} \log(1+\exp(-y_i\, \phi_{\mathrm{FFM}}(\boldsymbol{w},\boldsymbol{x})))$$

where

$$\phi_{\mathrm{FFM}}(\boldsymbol{w},\boldsymbol{x}) = \sum_{j_1=1}^{n}\sum_{j_2=j_1+1}^{n} (\boldsymbol{w}_{j_1,f_2}\cdot \boldsymbol{w}_{j_2,f_1}) x_{j_1} x_{j_2}$$

其中 m 為 data instances 的數量；λ 表 regularization parameter；y_i 表示第 i 個 instance 的 label；$w_{j1,f2}$ 和 $w_{j2,f1}$ 分別表示長度為 k 的 implicit vector；f_1 與 f_2 分別為 j_1 與 j_2 的 fields。每回合更新 $w_{j1,f2}$ 和 $w_{j2,f1}$ 的步驟如下，首先計算 sub-gradients：

$$\boldsymbol{g}_{j_1,f_2} \equiv \nabla_{\boldsymbol{w}_{j_1,f_2}} f(\boldsymbol{w}) = \lambda\cdot \boldsymbol{w}_{j_1,f_2} + \kappa\cdot \boldsymbol{w}_{j_2,f_1} x_{j_1} x_{j_2},$$
$$\boldsymbol{g}_{j_2,f_1} \equiv \nabla_{\boldsymbol{w}_{j_2,f_1}} f(\boldsymbol{w}) = \lambda\cdot \boldsymbol{w}_{j_2,f_1} + \kappa\cdot \boldsymbol{w}_{j_1,f_2} x_{j_1} x_{j_2},$$

where

$$\kappa = \frac{\partial \log(1+\exp(-y\phi_{\mathrm{FFM}}(\boldsymbol{w},\boldsymbol{x})))}{\partial \phi_{\mathrm{FFM}}(\boldsymbol{w},\boldsymbol{x})} = \frac{-y}{1+\exp(y\phi_{\mathrm{FFM}}(\boldsymbol{w},\boldsymbol{x}))}$$

之後對 $w_{j1,f2}$ 和 $w_{j2,f1}$ 中每個 element 進行 gradient 平方的累加與更新 (共需 k 回合)：

$$(G_{j_1,f_2})_d \leftarrow (G_{j_1,f_2})_d + (g_{j_1,f_2})_d^2$$
$$(G_{j_2,f_1})_d \leftarrow (G_{j_2,f_1})_d + (g_{j_2,f_1})_d^2 ,$$

$$(w_{j_1,f_2})_d \leftarrow (w_{j_1,f_2})_d - \frac{\eta}{\sqrt{(G_{j_1,f_2})_d}}(g_{j_1,f_2})_d$$
$$(w_{j_2,f_1})_d \leftarrow (w_{j_2,f_1})_d - \frac{\eta}{\sqrt{(G_{j_2,f_1})_d}}(g_{j_2,f_1})_d$$

其中 $d = 1, ..., k$；η 為 learning rate。上述更新 $w_{j1,f2}$ 和 $w_{j2,f1}$ 的步驟可整理成下列 pseudocode：

Training FFM using SG

```
Let G ∈ R^{n×f×k} be a tensor of all ones
Run the following loop for t epochs
for i ∈ {1, ··· , m} do
    Sample a data point (y, x)
    caclulate κ
    for j1 ∈ non-zero terms in {1, ··· , n} do
        for j2 ∈ non-zero terms in {j1 + 1, ··· , n} do
            calculate sub-gradient by (5) and (6)
            for d ∈ {1, ··· , k} do
                Update the gradient sum by (7) and (8)
                Update model by (9) and (10)
```

FTRL-Proximal (Follow the (Proximally) Regularized Leader)[5]

FTRL 也是用 logistic regression，但學習參數 (或權重) 的方法並非用傳統的 gradient descent (GD)。傳統 gradient descent 如 SGD 或 online gradient descent (OGD) 好處是往 loss 的下降方向很準。然而在廣告點擊或 click-through rate (CTR) 問題上不大適合：第一，資料量大，取 batch 算梯度下降很耗資源；第二，特徵維度大，很多位置訓練次數不足，傳統 GD 方法不易學到 sparse 解；第三，SGD 收斂速度慢。FTRL-Proximal 解決了這些問題，以下解釋其數學模型。

假設第 t 筆資料 feature 向量為 x_t，且 logistic regression 的參數向量為 w_t，用 sigmoid 定義該廣告被點擊的機率：

$$p_t = \sigma(x_t \cdot w_t) = 1 / (1 + exp(-x_t \cdot w_t))$$

並以 logistic loss (LogLoss) 當 cost function：

$$l_t(w_t) = -y_t \, log(p_t) - (1 - y_t) \, log(1 - p_t)$$

可得梯度：

$$g_t = \nabla l_t(w_t) = (p_t - y_t) \, x_t$$

傳統 SGD 如此更新權重：

$$w_{t+1} = w_t - \eta_t \, g_t$$

其中 η_t 為 learning rate (如 $\eta_t = 1 / \sqrt{t}$)。此方法雖然簡單，但不易產生 sparse 解。在 FTRL-Proximal 改成：

$$w_{t+1} = argmin_w (g_{1:t} \cdot w + \tfrac{1}{2} \sum_{s=1}^{t} \eta_s \, \|w - w_s\|_2^2 + \lambda_1 \|w\|_1)$$

其中 $g_{1:t} = \sum_{s=1}^{t} g_s$。乍看之下超難解，且和 SGD 差很多，但實際上若 $\lambda_1 = 0$，則會和原本產生一模一樣的權重；若 $\lambda_1 > 0$，則能產生很好的 sparsity。此外，經過如下推導，可得簡單的 closed form 解。首先把 argmin 內的式子重寫：

[5] 原始論文: http://ppt.cc/nNcA1

$$(g_{1:t} - \sum_{s=1}^{t} \eta_s w_s) \cdot w + \frac{1}{\eta_t}\|w\|_2^2 + \lambda_1\|w\|_1 + \text{常數}$$

因此，只要紀錄 $z_{t-1} = g_{1:t-1} - \sum_{s=1}^{t-1} \eta_s w_s$，則 $z_t = z_{t-1} + g_t + (\frac{1}{\eta_t} - \frac{1}{\eta_{t-1}})w_t$，然後可得：

$$\begin{aligned} w_{t+1,i} &= 0 && \text{if } |z_{t,i}| < \lambda_1 \\ w_{t+1,i} &= -\eta_t\,(z_{t,i} - sign(z_{t,i})\,\lambda_1) && \text{otherwise.} \end{aligned}$$

下標 i 是 feature 向量的 index。可發現若取 $\lambda_1 = 0$ 則整個 update 過程和 SGD 是一樣的。另外 FTRL-Proximal 還讓不同 feature 有不同的 learning rate (per-coordinate learning rates)：

$$\eta_{t,i} = \alpha / (\beta + (\sum_{s=1}^{t} g_{s,i}^2)^{0.5})$$

最後得到如下 pseudocode：

Algorithm 1 Per-Coordinate FTRL-Proximal with L_1 and L_2 Regularization for Logistic Regression

---

Input: 參數 α、β、λ_1、λ_2

for $t = 1$ **to** T **do** # T 為資料筆數

    令此筆資料 feature 的 one-hot vector 為 x_t

    令 $I = \{i \mid x_i \neq 0\}$，即 vector 非零位置

    for $i \in I$ **do**

        if $|z_{t,i}| < \lambda_1$ **then**

            $w_{t,i} = 0$

        else

            $w_{t,i} = -(\frac{\beta + \sqrt{n_i}}{\alpha} + \lambda_2)^{-1}(z_i - sign(z_i)\,\lambda_1)$

    # 預測點擊機率 (內積取 sigmoid)

    $p_t = \sigma(x_t \cdot w)$

    # 更新參數

    令 $y_t \in \{0, 1\}$ 代表是否點擊

    for $i \in I$ **do**

        $g_i = (p_t - y_t)\,x_i$ # LogLoss 梯度

        $\eta_i = \frac{1}{\alpha}(\sqrt{n_i + g_i^2} - \sqrt{n_i})$ # 即 $\frac{1}{\eta_{t,i}} - \frac{1}{\eta_{t-1,i}}$

        $z_i \leftarrow z_i + g_i - \eta_i\, w_{t,i}$

        $n_i \leftarrow n_i + g_i^2$

該算法只有 1 個 epoch，也就是對所有資料只跑 1 次。實際上我們會 train 數十個 epochs。另外 pseudocode 還多加 L_2 regularization，原理是把權重更新方式改成：

$$w_{t+1} = argmin_w(g_{1:t} \cdot w + \frac{1}{2}\sum_{s=1}^{t} \eta_s \|w - w_s\|_2^2 + \lambda_1\|w\|_1 + \frac{\lambda_2}{2}\|w\|_2^2)$$

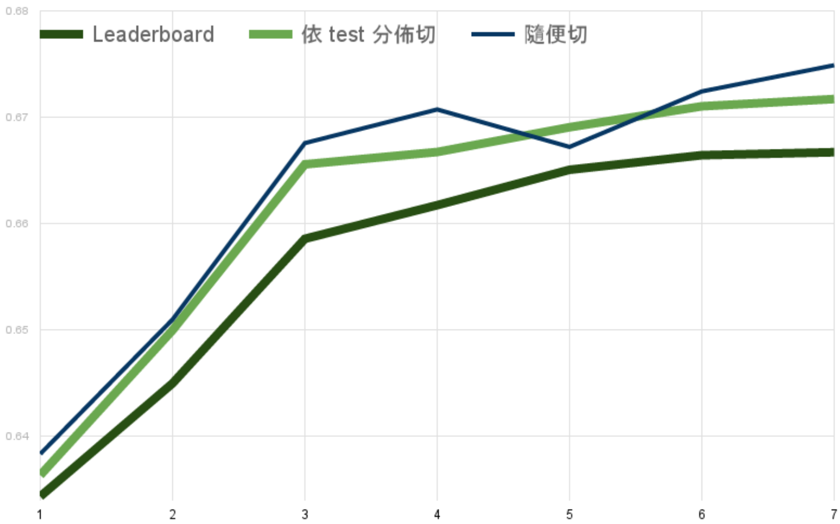
也就是多了 λ_2 那項，然後做類似推導，即可得 pseudocode 中的式子。總結 FTRL 有 α、β、λ_1 和 λ_2 四個參數。通常取 β=1 即可，其它參數則要根據 feature 不斷調整。

FTRL 沒特別 public 的 library，此競賽大家多半是拿某人在 Kaggle 分享的程式 (`ftrl.py`)。但因速度慢得一塌糊塗，我們變自己另外改良實作 (`ftrl.hpp`)。請參考 **Q4.** 後面，我們有詳述優化細節。

Q4. 實驗與討論

比較有無依照 **Testing Set** 分佈切 **Validation**

這邊來比較不同切法，在 local validation 和上傳 Kaggle 的分數，如下圖。



我們用 FTRL training，共上傳 7 次，在 public leaderboard 分數如深綠色折線。淺綠色折線是依照 testing 分佈切的 validation 分數 (程式為 `splitvalid.py`)，可以發現和 leaderboard 分數成長滿一致的，我們依此 validation 判斷分數是否進步再上傳 Kaggle，沒浪費任何一次。

為做比較，後來又隨便切一組 validation (程式為 `splitvalidrand.py`)，這次完全隨機切的，沒考慮 testing 分佈，測試分數如藍色折線。可發現的確和 leaderboard 分數成長比較不一致。

FFM 效果

FFM 進一步把 feature 歸類到 field 內，可以想成是把 feature 分類。不同分法會影響效果，下面列出我們嘗試過的各種分法，以及對應的 validation 分數。因為即使同樣 feature 和參數，FFM 每次 train 出來分數會不大一樣，我們多 train 幾次，取最高者。使用的是智仁大神的 libffm 套件。

在每次實驗表格中，第一列代表 field，第二列代表該 field 下有哪些 features。

f_1		f_2		f_3		f_4		f_5		f_6		f_7	
#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12	#13	#14

MAP@12: 0.67214

f_1				f_2				f_3					f_4
#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12	#13	#14

MAP@12: 0.67537

f_1			f_2					f_3					f_4
#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12	#13	#14

MAP@12: 0.67766

f_1								f_2					f_4
#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12	#13	#14

MAP@12: 0.67635

一些明顯可歸同一類的，如三個不同大小範圍的 location 都是位置資訊，很直觀地要分成同類。剩下比較模稜兩可的，只能不斷嘗試。像第一種分法 (隨便分) 分數就明顯比較低。但後面幾組分法就都差不多，不大能看出好壞，畢竟很有可能是誤差。

下表比較取相同 features 情況下，FTRL 和 FFM 的 validation 分數，其中有兩次 FFM 上傳 Kaggle (表格括號內分數)。可以發現 FFM 在 validation 分數高出 FTRL 不少，然而上傳 Kaggle 分數反而比 FTRL 差，可見其容易 overfitting。之後便沒再嘗試，都改用 FTRL。

Feature 數	**10**	**14**	**15**	**16**	**17**
FTRL 分數	0.66829	0.67035	0.66917	0.66951	0.66960
FFM 分數	0.67521 (0.65857)	0.67766 (0.66173)	0.67628	0.67518	0.67705

即便是同樣實驗設定，FFM 每次實驗變動都不小，和 FTRL 分數變化也不大一致，所以感覺不大適合用它來篩選 feature。但很多 top 10% 的參賽者都是用 FFM，有人甚至只取 3 個 features！我認為先用 FTRL 確定 feature 後，FFM 應該可以用同組 feature 達到 Kaggle 更好的分數 (可惜沒試出來 QAQ)。

在調整參數方面，FFM 也無法做太細的微調，因為同樣參數下 FFM 每次 train 出來的結果都有些差異。影響較大的參數是 implicit vector 的維度 k，通常越大效果越好，但執行時間和記憶體需求會成長很快。我們最多設定到 k=24。Feature 取超過 17 個記憶體就會滿到 swap，執行速度變極慢；另外因為 FFM 容易 overfitting，我們有用 validation 做 auto-stop，也就是若 validation loss 上升，則停止 training；learning rate 取夠小的值即可 (約 0.05)，讓 training epoch 數可到 10 以上 (epoch 數決定於 auto-stop)，比較能達到 loss 最小值。

FTRL 參數調整

不同 feature 對應的最佳參數是不同的，所以每次換 feature 都要重調參數。給定 feature 後，我們用 validation 來判斷參數好壞。只需 train 約 5 個 epochs 就可做 validation。實驗發現，在同樣 epoch 數下，validation 分數好的，就是比較好的參數。也就是若 train 5 個 epochs 分數比另一組參數高，那 train 50 個 epochs 分數也會比另一組高，這是 FTRL 蠻棒的性質。另外 FTRL training 過程分數都是嚴格遞增，這些性質對微調參數非常有幫助！

固定 $\beta = 1$，需調整的參數只有 α、λ_1 和 λ_2。若取的 feature 為前面所提的 14 個，最佳的參數為 $\alpha = 0.013$、$\lambda_1 = 0.2$ 和 $\lambda_2 = 0$。確定 features 和參數後，最後再改用全部 training set 做訓練，並跑 100 個 epochs 上傳 Kaggle。

各個 **Feature** 的效果

這邊紀錄取不同 feature 用 FTRL 後的 validation 的分數。每次換 feature 都會重調參數到最高分，每次都只 train 5 個 epochs。我們以最前面提到的 14 個 features 為基準，看看多一個或少一個 feature 的影響。下表是扣掉 feature 在 validation 分數的變化，四捨五入到小數第二位。

少哪個 feature	#1	#2	#3	#4	#5	#6	#14
分數變化 (%)	-0.12	-0.29	-0.27	-0.08	-0.17	-0.11	-0.93

不論少哪個，分數都會下降 (少 #7 到 #13 也是類似結果，不一一列舉)。差異最大的是 #14 的 leak feature！畢竟使用者若從沒瀏覽過該網頁，表示根本不可能點擊過廣告，這很明顯是重要的 feature。接著來看多加 feature 的效果。

多什麼 feature	`display_id`	`document_id`	`nb_ads`
分數變化 (%)	-0.20	-0.09	+0.01

注意這邊的 `document_id` 是從 `events.csv` 抓出來的，指的是目前 user 正在瀏覽的頁面，而非廣告點擊後連結到的頁面；`nb_ads` 指的是此頁面的廣告數目。原以為 feature 越多越好，畢竟不重要的 feature 頂多 weight 變 0，實驗看來並非如此。有趣的是加了 `nb_ads` 好像微幅上升，但理論上這是完全不相干的 feature 才對：因為我們要預測同一頁面中哪一個廣告會被點擊，但這些廣告 `nb_ads` 都一樣，根本不可能靠它分析差異！我們認為這只是誤差，若 train 更多 epochs 差異會更逼近 0。

優化 **FTRL** 程式碼

Kaggle 上有人提供 FTRL Python 程式碼，許多人都是用此版本。但它速度太慢，吃太多記憶體，且不同 feature 可能會 hash 到同樣的值，會對應到 one-hot vector 的同一位置。仔細分析他的 code 並發現許多效能關鍵問題後，我們決定自己實作。先來分析原始程式 (請看 `ftrl.py`)。

一開始從 `.csv` 讀入各種 feature，並以 Python dictionary 方式儲存。例如由 `ad_id` 可查詢 `campaign_id` 和 `advertiser_id`，則把 `ad_id` 當 key，把 `campaign_id` 和 `advertiser_id` 當 value 存入 dict。之後每筆資料進來時，便用 `ad_id` 去查 dict 撈出

`campaign_id` 等。這樣的 dict 會有好幾個存在記憶體中，每筆資料進來時，都要查詢這些 dict 撈出所有 feature。可參考 **Q1.** 抓 feature 的流程圖，每個箭頭的來源就是 key，箭頭指向的 feature 則存成 value。

撈出 feature 後，要轉成 one-hot vector 對應的位置。原始程式方法是，把 feature 值做 hash 得到一個整數。為防止不同 feature 有同樣 ID 會 hash 到同一個值，會做此轉換：例如 `topic_id` 是 100，則先轉成字串 ‘`topic_id`’ + ‘`_`’ + ‘`100`’ = ‘`topic_id_100`’ 然後再把此字串做 hash。這樣即便其它 feature ID 也是 100，但因為轉換後的字串不同，就 (比較) 不會 hash 到同樣的值。

然而 hash 後的整數範圍很大，從 `MIN_INT` 到 `MAX_INT`，且包含負數。設 one-hot vector 維度是 2^{20} ，hash 後的整數為 i，則這樣更新 i:

$$i \leftarrow abs(i) \ \% \ 2^{20}$$

可保證 $0 \le i < 2^{20}$，也就是在 one-hot vector 索引範圍內。但這樣的問題是 (1) 增加了不同 feature 對應到同樣 index 的機率，而且 (2) 在 one-hot vector 中可能會有很多位置完全沒被用到，因此 vector 維度必須開得很大。實驗發現每次增加維度，分數都會更高，但會非常浪費記憶體。我開到 2^{32} 包括前面所用 dict 跑此程式，32 GB 記憶體就會完全吃滿，甚至用到 swap (此時執行速度會嚴重下降)。

總結來看，每筆資料進來會做兩件事: (1) 查詢 dict 抓出各個 feature，然後 (2) 把 feature 字串用 hash 算出 index 值。分析完上述問題，我們可以做許多改善。

與其每次資料進來才查詢 dict，我們改成先一次把所有 feature 抓出來。然後把 feature 轉成 one-hot vector 對應的 index，寫進檔案中。之後讀此檔當 input，每筆資料進來就已經是我們要的 indices，可直接餵入 FTRL 中。

另外，把 feature 轉成 index 部分，我們不是用 hash。我們另外建一個 set，然後把所有可能的 feature 字串加入，再指定給每個字串唯一的 index。這樣不只完全避免 index 衝突問題，而且也不會有沒用到的 index。即便取到 20 個 features 我們也只需要 2^{26} 大小的 one-hot vector，此大小陣列多數筆電記憶體都夠；相較之下，原本用到 2^{32} 仍可能會有衝突，且 32 GB 記憶體都不夠。

FTRL 演算法，我們也自己用 C/C++ 實作 (請看 `ftrl.hpp`)，資料結構只用純 C 陣列。

這樣不只省去每個 epoch 都要查詢 dict 的時間 (只有第一次需要)，而且寫入檔案後，也不需要記憶體儲存 dict 資料結構。這部分大大減少執行時間，以及記憶體用量。以我們抓 14 個 features 來說，原本程式每個 epoch 需要超過 600 秒 (使用 `pypy` 執行)，但我們的版本只需約 50 秒 (用 `-O3` 優化)。原本記憶體光 dict 就需超過 20 GB，但我們除了第一次外，之後完全免去這些記憶體，加上 FTRL 等約只需不到 2 GB。