Machine Learning Final Project Report

1. 題目：Cyber Security Attack Defender (第一題)

2. 隊伍資訊

- 隊伍名稱：NTU_r05922034_henryhenry
- 隊員與貢獻度：

名字	**學號**	**分工**
顏修溫	r05922034	25% (random forest tree model, Preprocessing)
鄭竣元	r05944034	25% (random forest tree model, experiment)
徐有慶	r05922162	25% (DNN model, stacking)
蔡承威	b01502058	25% (DNN model)

3. Preprocessing/Feature Engineering

Preprocessing 的部分，我們做了下面這幾項處理：

- <u>將 string 型別的 feature 轉為 float 型別</u>：如下圖所見，training data 裡面含有一筆一筆的連線資訊，每筆連線會有 41 個特徵、再加上最後一個欄位的 label。我們可以發現第 2 個欄位、第 3 個欄位、第 4 個欄位與最後一個 label 欄位的特徵(紅框處)是 string 型別，除此之外都是 float 型別。看到這個現象後我們就產生一個疑問，究竟有沒有辦法把 String 型別的特徵直接丟進去 model 裡頭 train 呢？經過查詢後我們的結論是，就直接用一些離散的數值把這些特徵區別開來即可，比如說把第二個欄位的 icmp 轉為數值 0、tcp 轉為 1、udp 轉為 2，而其他欄位也是以同樣的邏輯將 string type feature 轉為 float type feature。經過轉換後，每筆連線中的所有特徵都會是 float 型別，接著就能夠進行下一步處理。

　　比較需要提的是最後一個 label 欄位部分。已知攻擊名稱會有好多種，比如說 apache2, back, mailbomb, processtable…這些攻擊名稱屬於 dos 類型；ps, buffer_overflow, perl…這些攻擊名稱屬於 u2r 類型。所以這個欄位的轉換方法就會有兩種，第一種是每個攻擊名稱都分派一個數值，例如 apache2 轉為 0、back 轉為 1、mailbomb 轉為 2 等等；第二種轉換方式是依照攻擊類型來分派數值，因此 normal 轉為 0、apache2, back, mailbomb 都會轉為 1、buffer_overflow, perl 都會轉成 2 等等。我們選擇的方式是第二種。我們後來有試著去評價這兩種轉換方式孰優孰劣，實際上效果並沒有太大差異。這部分的處理程式碼寫在 preprocess.py 裡頭。

Fig 3.1

- 從原始 training data 裡取出部分資料作為新的 training data：經過計算後，我們發現原始 training data 當中每種攻擊的出現次數如下表 Table 3.1，可以發現 normal、neptune 和 smurf 這三種情況的出現次數非常之多，分別出現 87 萬、96 萬、252 萬次。所以我們從這三種攻擊裡頭，各只挑出 1 萬筆出來就好，而其他攻擊就還是全部保留。取出來後的資料集我們稱為 subtrain，subtrain 當中每種攻擊的出現次數可見表 Table 3.2。這個部分的處理寫在 make_subcorpus.py 裡頭。

  事實上我們一開始還沒有做這步處理的時候，成績一直沒甚麼長進。直到後來我們切出 subtrain 當作 training data 後，才突破 strong baseline 的，這個前處理算是我們最關鍵的一步！

攻擊名稱	次數	攻擊名稱	次數	攻擊名稱	次數	攻擊名稱	次數
guess_passwd	47	rootkit	8	satan	14309	imap	12
spy	1	pod	244	loadmodule	9	warezclient	914
ftp_write	8	portsweep	9328	buffer_overflow	25	land	19
nmap	2080	perl	1	normal	875363	neptune	964959
back	1971	ipsweep	11272	phf	4	smurf	2527107
multihop	7	teardrop	881	warezmaster	18		

Table 3.1

攻擊名稱	次數	攻擊名稱	次數	攻擊名稱	次數	攻擊名稱	次數
guess_passwd	47	rootkit	8	satan	14309	imap	12
spy	1	pod	244	loadmodule	9	warezclient	914
ftp_write	8	portsweep	9328	buffer_overflow	25	land	19
nmap	2080	perl	1	normal	**10000**	neptune	**10000**
back	1971	ipsweep	11272	phf	4	smurf	**10000**
multihop	7	teardrop	881	warezmaster	18		

Table 3.2

- 對資料做 Normalization：從 Fig 3.1 裡頭我們可以發現，部分特徵的數值 range 非常大，例如有些欄位是 0~1 之間、有些可能會到一千多。我們想說特徵之間的 range 差異這麼大，就對資料進行 Normalization，使用的工具是 sklearn.preprocessing package。使用了 Normalization 之後，成績並沒有太大的進步。

4. Model Description

我們做了兩種 Model，分別是 DNN model 和 Random Forest Tree model：

- DNN model：我們在看完這一題的資料類型之後，認為可以試試看之前 Homework 3 做過的 supervised learning，所以使用了基於 TensorFlow 的 Keras 建了一個 Deep Neural Network。該 DNN 的架構為 5 或 8 層的 Dense Layer，每一層的 Activation function 為 relu，最後加上 0.5 的 Dropout。

  之所以會有 5 或 8 層這兩個不同的架構是因為我們組員使用同一份 code 在不同的電腦上做 Training 時會出現不同的結果，若是沒有對層數或是 Dense Layer 的 neuron 數做調整的話，會出現 Train 出來的結果全部傾斜到同一個 Output 而無法透過 training 調整回來的情況。整體而言，大部分的 DNN 測試都是用下圖的架構，藉由調整 training set 來試圖達到更好的訓練效果。DNN model 寫在 model1_train.py 裡頭，只要使用 python 2 執行 model1_train.py 就會輸出 out.csv 結果檔。

```
model = Sequential()
model.add(Dense(output_dim=256, input_dim=41))
model.add(Activation("relu"))
model.add(Dropout(0.2))
model.add(Dense(neuron_size))                    # neuron_size could be 256 or 512
model.add(Activation('relu'))
model.add(Dense(neuron_size))
model.add(Activation('relu'))
model.add(Dense(neuron_size))
model.add(Activation('relu'))
model.add(Dense(neuron_size))
model.add(Activation('relu'))
model.add(Dense(neuron_size))
model.add(Activation('relu'))
model.add(Dense(neuron_size))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(nb_classes))                     # nb_classes could be 5 or 41
model.add(Activation("softmax"))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Fig 4.1 DNN model structure

- Random Forest Tree model：第二種 model 使用了 sklearn 的 RandomForestClassifier，使用方法非常直覺，這個 model 需要調整的參數就只有 estimator 的數量。經過多方嘗試，我們把 estimator 設為 200、並且使用 subtrain 當作 training data 的話，可以得到 kaggle best，分數為 0.96305。

  我們一開始是只使用 DNN model 來進行這個題目，但是我們發現 DNN model 的效果非常不穩定，training 過程有時會傾斜到某個點，然後 accuracy 就開始一直往下降，關於這點我們還沒有找出確切的原因。另一方面則是我們使用 DNN model 一直過不了 strong baseline，所以決定另尋新的解法。

然後我們就想起之前有幾位同學上台分享前幾次作業的做法，第二次作業是分類問題(分類垃圾郵件)，那幾個同學就是使用了 Random Forest Tree 得到這麼高的分數，所以我們也就決定來試試看。沒想到這個方法的效果讓我們非常驚奇，因為 Random Forest Tree model 只需要花數秒鐘的執行時間(可見 Fig 4.1 和 Fig 4.2)，比起 DNN model 快上許多(DNN model 約需要 10 分鐘)；最重要的是，Random Forest Tree model 竟然可以在這麼短的執行時間內，就達到比 DNN model 還高的成績，實在令我們非常驚喜。我們是使用了 Random Forest Tree model 後才越過 strong baseline 的。

關於 Random Forest Tree model 的 estimator 參數，我們嘗試過多種可能，estimator number 從 10、50、80、100 到 200，跑出來的最佳結果都是差不多的。但是當 estimator number 較小的時候，感覺每次跑出來的成績會有一些落差、起起伏伏的不太穩定。所以我們就決定將 estimator number 設為 200 才較為穩定。Random Forest Tree model 寫在 model2_train.py 裡頭，只要使用 python 2 執行 model2_train.py 就會輸出 out.csv 結果檔。

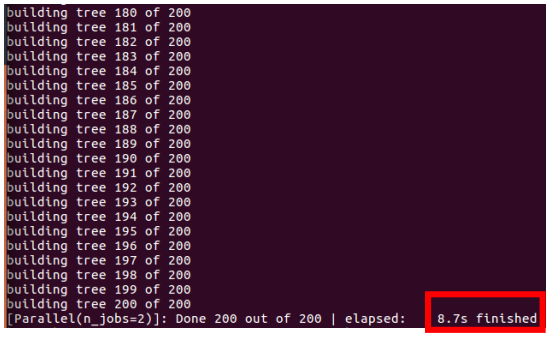


Fig 4.2 training time

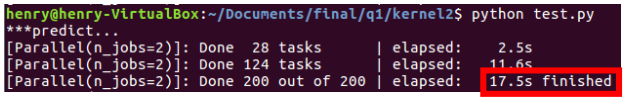


Fig 4.3 predict time

5. Experiments and Discussion

我們做了三個實驗，分別是 Split training set、Stacking、Feature vector 降維。並針對實驗結果做了一些討論：

- Split training set：為了增進 DNN 的 accuracy，我們主要針對 training set 進行調整與實驗。因為所有的 training data 總共有 440 萬左右，我們在做 training 時無法把全部的 data 一次讀到記憶體裡面，所以總共分成了三個部分，分別為 0~200 萬、200 萬~400 萬以及剩餘所有 data。將 data 切割完以後，我們嘗試了幾種不同的 training set：
  - **I.** 使用第一個 partition 的 Data
  - **II.** 全部的 Data，分三次讀進 model
  - **III.** 選取某些特定的 Data

  在一開始做這份專題的時候，為了驗證寫的 code 是否能夠有效的進行 training，我們只用了第一個 partition 的資料做 training，觀察到 loss 有明顯變小之後再放到 Kaggle 上看表現如何，結果 accuracy 落在 0.957，確認了 DNN 的做法是可行的。

於是接下來我們決定嘗試用更多的 Data 試試看，觀察是否使用更多 training set 時會有更好的 performance，所以花了很多時間在嘗試使用全部 Data 進行 training。實現的做法是依序將三個 partition 讀進 Model 做 training，重複的進行這些步驟，從中選出 loss 較小的 model 做 prediction 放到 Kaggle 上看表現如何，而 accuracy 大都落在 0.94~0.95 左右，反而 performance 無法超過只使用第一個 partition 時的表現。

為了了解這些變化的原因，我們決定針對資料做進一步的分析，以下是 training set 跟 test set 的資料分佈：

	Training Set	Test Set
Normal	19.85%	26%
DoS	79.28%	71.23%
U2R	0.0015%	0.038%
R2L	0.0234%	1.412%
Probe	0.84%	1.354%

可以發現 Normal 跟 DoS 的資料佔了絕大多數，接下來則是 R2L 跟 Probe 的資料也有一點點的比重；而 Training Set 跟 Test Set 最大的不同則是在於 R2L 跟 Probe 的比重有明顯的增加。再觀察 predict 出來的結果，有幾次的結果都是只有 0 跟 1，因此我們推測是否因為 Training Set 中的 Normal 跟 DoS 的資料過多，使得其他種 Data 無法被正確的 predict。於是我們對 Training Set 中的 Data 做了一些篩選，只取前 10 萬筆的 Normal 跟 DoS，其他種的 Data 照舊，切出了一塊較小的 Training Set 來做訓練，其資料分佈如下：

	New Training Set
Normal	29.3%
DoS	59.537%
U2R	0.01289%
R2L	0.29%
Probe	10.84%

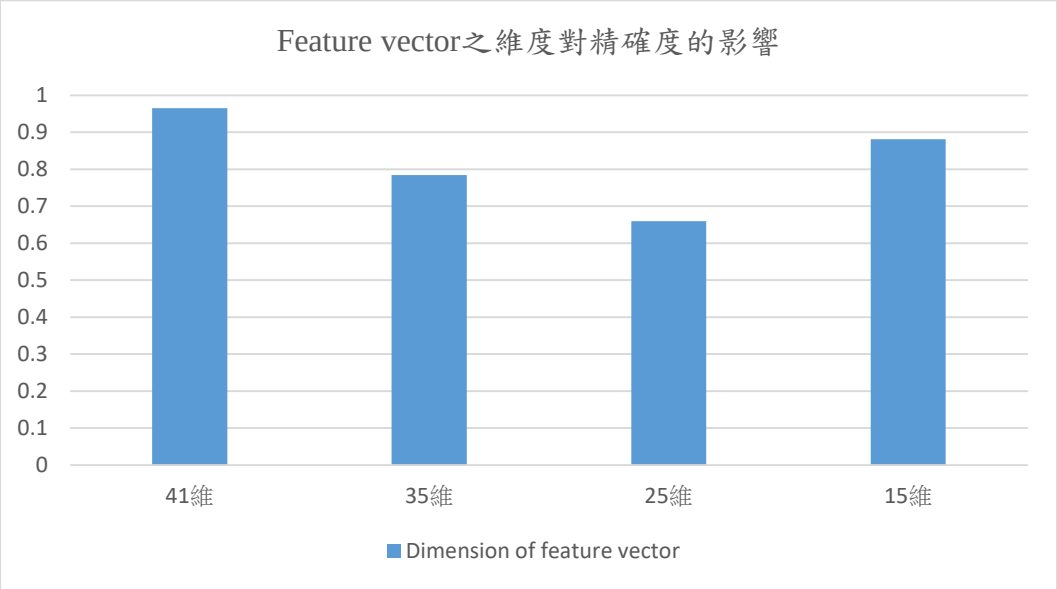
使用新的 Training Set 來做 DNN 的話，可以在 Test Set 得到 0.96 左右的正確率，十分接近 Strong Baseline 的要求。

- 使用 LSA 對 feature vector 降維：training data 裡頭的每一筆連線資訊是由 41 個 feature 所組成，這個實驗是打算使用 LSA 對 41 維的 feature vector 進行降維，降維後才丟進去 train。會有這個發想是因為在第四次作業的時候，我就是因為先使用 LSA 對 word vector 進行降維後才進行 k-means clustering，得到的正面效果非常顯著。就是因為這樣，所以我才想說如果試試看用 LSA 對每筆連線的 feature vector 進行降維，然後把降維後的

feature vector 丟進去 train 的話，效果會是如何呢？

已知原本的 feature vector 有 41 維，我們嘗試降到 35 維、25 維、15 維，出來的效果如表 Table 5.1 所示，可以發現使用 LSA 降維的效果非常地差。

我們針對這個結果進行討論，認為第四題的 word vector 是 5000 多維，這維度實在太大勢必含有許多雜訊在裏頭，所以降維前的 vector 所 train 出來的 model 效果不佳。如果先使用 LSA 找出這 5000 多維特徵之間的關聯性後才丟進去 train，那麼效果顯然會十分理想。但是 final 的第一題並不適用 LSA 降維，因為這題的 feature vector 只有 41 維，feature 數量已經算是滿少的了，如果還進行降維的話效果肯定不會比較好、甚至會大打折扣。



- Stacking：利用三個 models，random forest、decision tree 和 DNN model 去做 stacking。分成兩階段來 training，先用 71158 筆資料，分別 train 三個 models(Fig 5.1)，再取 2000 筆資料當作 validation set 及 train 完的三個 model，train 一個 logistic regression model(Fig 5.2)。最後再用兩階段 training 完畢的 model 去做 prediction。

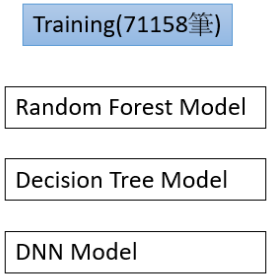


Fig 5.1

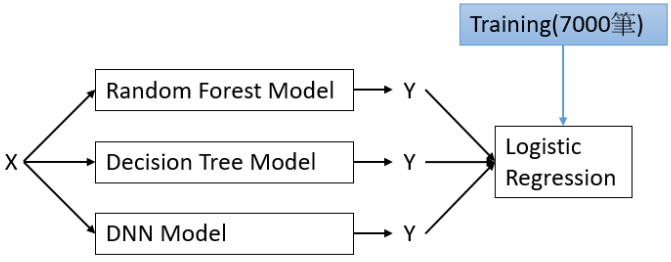
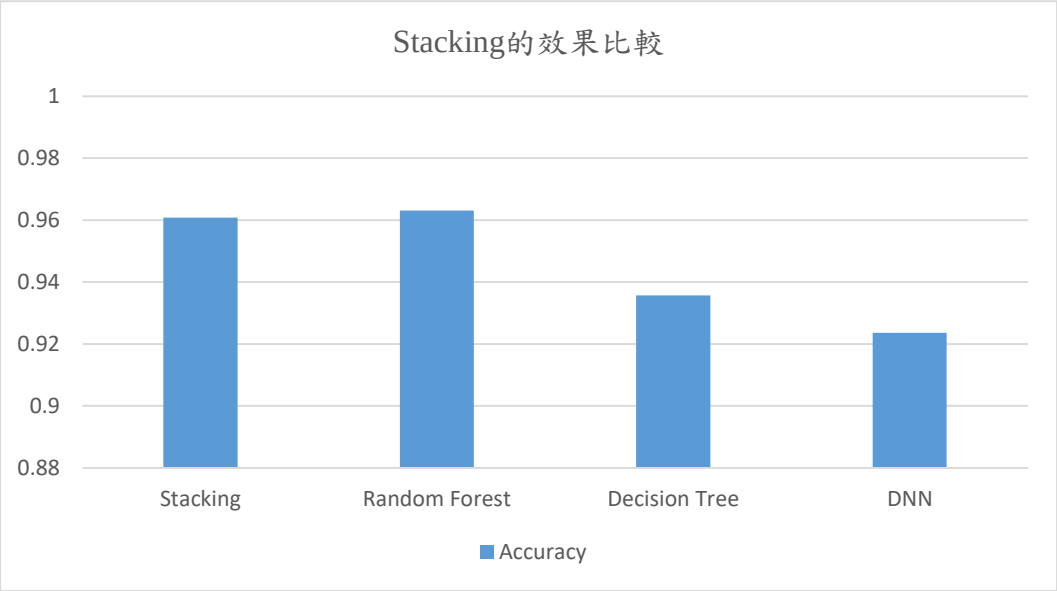


Fig 5.2

第一階段 train 完的 model 其 accuracy rate 分別為:
Random Forest: 1.0
Decision Tree: 0.9753
DNN: 0.9674
第二階段 train 完 logistic regression 後，其 accuracy rate 為 0.9926
利用此方法在 test set 上的表現，其 accuracy rate 為 0.96079

接著比較使用 stacking 後，和原來的 models 在 test set 上的表現。



由實驗結果來看，使用 stacking 後，在原本較差的 model 上，accuracy rate 可以得到提升，但在原本表現就不錯的 model 上卻會造成反效果。在此題上，若要得到整體 accuracy rate 的提升，使用 stacking 看起來得不到太好的效果，需要做的可能是對 training data 再做一些前處理。