# 中華民國第 63 屆中小學科學展覽會

# 作品說明書

---

高中組　電腦與資訊學科

## 第一名

052502

## 惡意程式無所遁形—以自然語言處理模型實現惡意程式之識別

學校名稱：國立臺灣師範大學附屬高級中學

| 作者： | 指導老師： |
| --- | --- |
| 高二 陳以哲 | 李啟龍 |


關鍵詞：惡意程式識別、自然語言處理、機器學習

# 得獎感言

## 融合資安與AI：辨識惡意程式的NLP探索

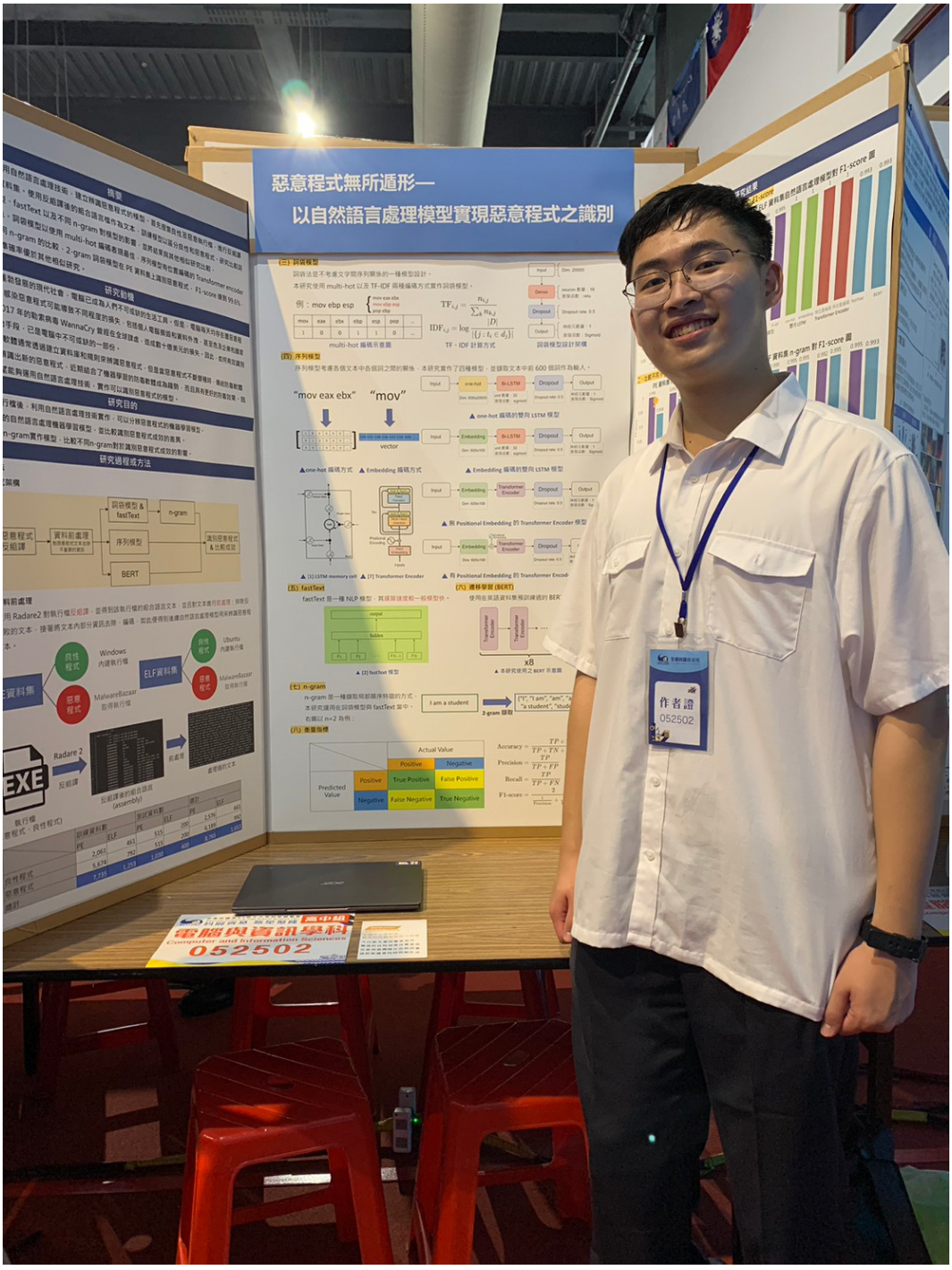
資訊安全與人工智慧都是近年來十分重要的領域。資訊安全保護我們的隱私以及使用電腦的安全，而人工智慧帶給我們前所未有的便利。大型語言模型如ChatGPT的崛起，也讓自然語言處理（NLP）這個領域受到更多的關注。

最初因為我對資訊安全和人工智慧都感興趣，就想著要如何結合兩者進行研究。看了許多文獻後，發現了這樣的方向—用人工智慧識別惡意程式。經過一段時間，我的主題聚焦在「使用自然語言處理技術，辨識惡意程式」。這個概念類似於直接丟執行檔的組合語言給語言模型，辨認是否為惡意程式。一開始我很好奇這種方式真的能成功嗎？沒想到最後模型真的能辨認惡意程式，而且成果相當不錯，未來還有繼續發展的空間。

得到這個獎項，除了感謝評審的肯定，並謝謝在我求學過程中，師大附中數理資優班、義學國中數理資優班各位老師的栽培，另外還有政大資管系蕭舜文教授的啟發與指導。

參加科展比賽的過程中，我從許多非常優秀的作品中獲得學習、精進的機會，也讓我了解自己的不足，未來要以更努力、謙虛的態度進行研究。

感謝全國科展提供一個這麼棒的舞台給學生們，讓我們可以展示成果。第63屆全國科展的結束，代表我們另一段學習旅程的開始。期許未來我們能運用在科展學習到的事物，繼續成長茁壯！




與展板合照

# 摘要

本研究旨在運用自然語言處理技術，建立辨識惡意程式的模型。首先搜集良性及惡意執行檔，進行反組譯及前處理以建立資料集。使用反組譯後的組合語言檔作為文本，訓練模型以區分良性和惡意程式。研究比較詞袋模型、序列模型、fastText 以及不同 n-gram 對模型的影響，並將結果與其他相似研究比較。

研究結果顯示。詞袋模型以使用 multi-hot 編碼表現最佳，序列模型有位置編碼的 Transformer encoder 表現最優。在不同 n-gram 的比較，2-gram 詞袋模型識別惡意程式達到 99.6%的 F1-score，且本研究的識別準確率優於其他相似研究。


# 壹、前言

隨著資訊科技日漸發達，各式計算機已成為生活及工作的一部份。然而，這些裝置時時刻刻都暴露在感染電腦病毒的風險。感染惡意程式的後果常帶來巨大的損失，小至個人電腦遭到破壞、資料遭竊取，嚴重則可能會造成機構難以承受的後果，甚至影響國家安全。如 2017 年席捲全球的勒索病毒 WannaCry，估計影響 150 個國家，造成全球數十億美元的損失。因此，如何有效辨別惡意程式的防治手段，已是電腦中不可或缺的一部份。

傳統防毒軟體透過建立資料庫以及規則，進行惡意程式的比對。但是惡意程式會不斷變種，若遇到新的惡意程式，傳統防毒軟體可能無法成功辨認出來。隨著人工智慧技術的應用越來越廣泛，近期也有防毒軟體結合 AI 來進行掃毒，其成效優於傳統防毒軟體。

近來自然語言處理技術 (Natural Language Processing, NLP) 越來越成熟， Rahali 與 Akhloufi (2021) 使用程式原始碼，結合常見的 NLP 模型 BERT，對 Android 的惡意程式進行辨認。Lu (2019) 則使用程式中的 opcode，結合 LSTM (Long Short-Term Memory, 長短期記憶) 進行惡意程式識別。在本研究中，研究者將執行檔透過逆向工程反組譯後，得到該程式的組合語言，並且做為文本，應用自然語言處理技術實作機器學習模型，進行辨識惡意程式的任務，以下為本研究之研究目的。

一、反組譯執行檔後，利用自然語言處理技術實作機器學習模型，以分辨惡意程式。

二、實作不同的自然語言處理技術機器學習模型，比較識別惡意程式成效的差異。

三、以不同的 n-gram 實作模型，比較不同 n-gram 對於識別惡意程式成效的影響。

四、與其他使用自然語言處理技術，進行惡意程式識別之研究比較。

# 貳、研究設備與器材

一、惡意程式資料庫：MalwareBazaar

MalwareBazaar 是分享惡意程式樣本的線上資料庫，該網站有 API 可供使用者查詢惡意程式，並可下載惡意程式樣本。由於該網站限制只能查詢並下載最近 1000 個上傳的惡意程式樣本，因此研究者取得的惡意程式樣本有限。

二、硬體：筆記型電腦。配備：Intel Core i5。

三、軟體

(一) Radare2

Radare2 是用於逆向工程的命令列軟體，常用於進行靜態分析。常見的靜態分析工具有 Radare2、Ghidra、IDA pro 等軟體，它們除了有將機器語言轉成組合語言的反組譯功能外，有些也同時具有反編譯的功能，讓研究人員更方便理解程式的作為。本研究使用 Radare2 分析資料集程式，並且進行反組譯。

(二) fastText (版本：0.9.2)

fastText (Joulin, Grave, Bojanowski & Mikolov, 2016) 是 Facebook 開源的函式庫，提供使用者執行文本分類以及文字轉向量等自然語言處理任務。fastText 的模型架構接近於 Word2Vec 的 CBOW，但是 fastText 採用了如階層 Softmax (Hierarchical Softmax) 函數等優化方法，使得訓練及推論時間大幅縮短。

(三) TensorFlow (版本：2.9.2)、Keras (版本：2.9.0)

TensorFlow 是用於機器學習的開源函式庫，此由 Google 提供的框架有多種模型以及工具，可用於機器學習。而 Keras 則是建立在 Tensorflow 之上的深度學習 API，提供簡易的方法，可用於建立與訓練深度學習模型。

(四) Python 3.9.7

Python 是一款易於撰寫與使用的程式語言。本研究使用 Python 作為分析腳本、前處理以及機器學習的程式語言。各使用套件的版本與用途如下。

1. Requests (版本：2.28.1)

   Requests 是用於發送 HTTP 請求的套件，本研究使用 Requests 向 MalwareBazaar (惡意程式資料庫)的 API 發送 HTTP POST 請求，查詢惡意程式的資料，並下載惡意程式的壓縮檔。

2. Pyzipper (版本：0.3.6)

   因為安全性的因素，從 MalwareBazaar 下載的惡意程式樣本都是經過 AES 加密的壓縮檔，而 Pyzipper 用來解壓縮。

3. Numpy (版本：1.23.5)

   Numpy 是 Python 的數學運算工具，本研究用於矩陣的運算。

# 參、研究過程與方法

一、研究架構

為了能夠分辨惡意程式，本研究將研究流程分為資料前處理、文本分類學習、與 n-gram 模型三個階段。圖 3-1 為研究架構與流程圖。

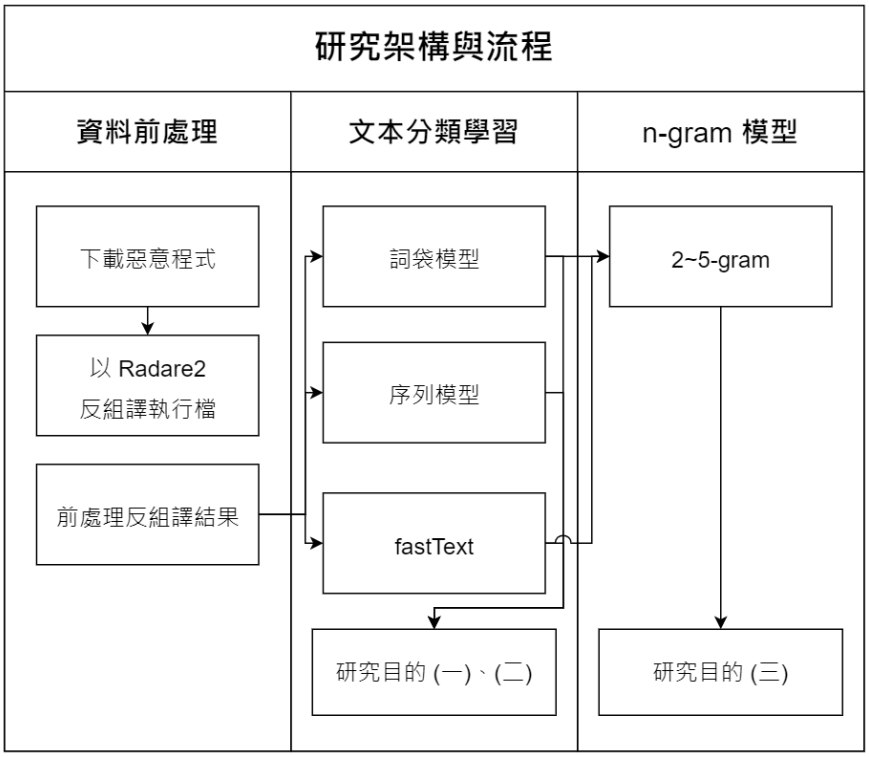



圖 3-1：研究架構與流程圖

二、資料前處理

資料前處理階段的目的是建立資料集，並且對資料集內的檔案進行前處理，此有利於後續模型的訓練。

(一) 資料蒐集

首先要建構執行檔資料集，為了執行惡意程式二元分辨的任務，資料集內需要有良性程式及惡意程式。本研究良性資料集的來源為 Windows 內建的 PE 檔，惡意資料集則是從 MalwareBazaar 下載的 PE 檔。

MalwareBazaar 只能下載指定的惡意程式，也就是說，需要有惡意程式的 sha256 雜湊值 (hash value) 才能向 MalwareBazaar 發送下載請求。所以要先向 MalwareBazaar 的 API 發送 HTTP POST 請求，詢問資料庫中 exe 檔惡意程式的資料。接著 MalwareBazaar 就會回應惡意程式的資料，如該惡意程式的雜湊值、惡意程式發現時間等。接著同樣發送 HTTP POST 請求給 MalwareBazaar 的 API，而參數要加上惡意程式的 sha256 雜湊值，這樣 MalwareBazaar 就會回傳該惡意程式的壓縮檔。

因為 MalwareBazaar 一次只能 Query 最近上傳的 1000 個惡意程式，所以本研究的做法是，慢慢累積惡意程式的數量，直到與良性程式的數量相等。圖 3-2 是本研究向 MalwareBazaar 請求下載惡意程式的示意圖。

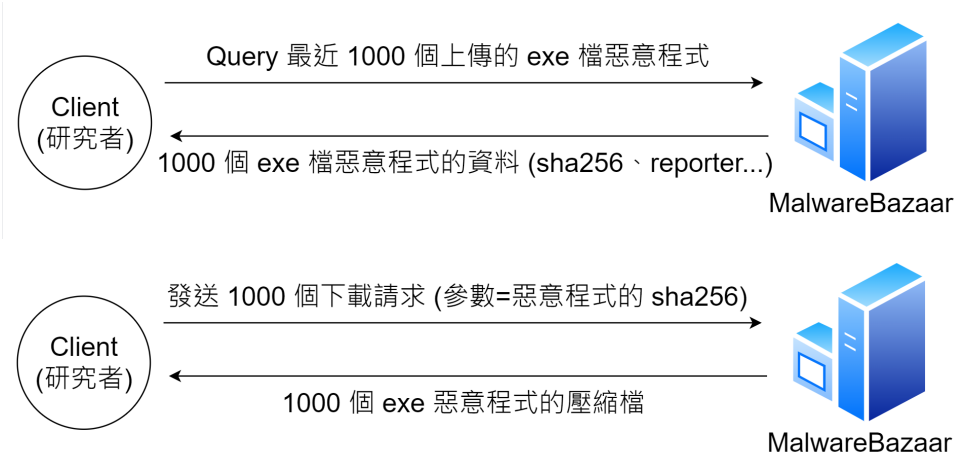



圖 3-2：向 MalwareBazaar 請求下載惡意程式示意圖

(二) 反組譯執行檔

建構執行檔資料集後，接下來要反組譯執行檔，研究者先後嘗試 Ghidra、angr 等靜態分析工具。但因為 Ghidra 使用 Java 撰寫、angr 執行速度較慢，最終選定 Radare2 作為主要分析工具。

Radare2 可用於靜態分析，並分析執行檔，反組譯執行檔內的函式。因此研究者使用 Radare2 提供的 Python API 進行反組譯，在此針對執行檔內的每個函式分別進行反組譯，並且將反組譯結果疊加在同一檔案內。

圖 3-3 是反組譯執行檔結果的文字檔案，欄位由左至右分別是：指令記憶體位置、機器語言指令、組合語言指令。

```
/ 1935: int main (int argc, char **argv, char **envp);
|           0x00402e61      55              push ebp
|           0x00402e62      8d6c2498        lea ebp, [esp - 0x68]
|           0x00402e66      81ec50010000    sub esp, 0x150
|           0x00402e6c      c78568ffffff.  mov dword [ebp - 0x98], 0x1154968f
|           0x00402e76      c745bc5d0996.  mov dword [ebp - 0x44], 0x1696095d
|           0x00402e7d      c745e0f9f5e5.  mov dword [ebp - 0x20], 0x24e5f5f9
|           0x00402e84      c745f4091458.  mov dword [ebp - 0xc], 0x4581409
|           0x00402e8b      c78570ffffff.  mov dword [ebp - 0x90], 0x47b7ea83
|           0x00402e95      c78538ffffff.  mov dword [ebp - 0xc8], 0x60a07459
|           0x00402e9f      c7453071137d.  mov dword [ebp + 0x30], 0x477d1371
|           0x00402ea6      c745a04e65ba.  mov dword [ebp - 0x60], 0x78ba654e
|           0x00402ead      c745185b8524.  mov dword [ebp + 0x18], 0x4424855b
|           0x00402eb4      c78540ffffff.  mov dword [ebp - 0xc0], 0x287a235f
|           0x00402ebe      c745e810b644.  mov dword [ebp - 0x18], 0xb44b610
|           0x00402ec5      c745a83a26c7.  mov dword [ebp - 0x58], 0x2ac7263a
|           0x00402ecc      c745201c88b9.  mov dword [ebp + 0x20], 0x30b9881c
|           0x00402ed3      c78534ffffff.  mov dword [ebp - 0xcc], 0x3159374b
|           0x00402edd      c745641540dc.  mov dword [ebp + 0x64], 0x8dc4015
|           0x00402ee4      c745cc226868.  mov dword [ebp - 0x34], 0x66686822
```

圖 3-3：反組譯執行檔結果的文字檔案

(三) 前處理反組譯結果

為了使後面步驟的文本分類學習有更好的表現，需要對反組譯後的組合語言進行處理。

有些執行檔無法被 Radare2 分析，或是分析出來的結果不符合一個正常的執行檔，研究者嘗試再以 Radare2 分析，但結果仍然一樣，其中良性程式在分析後有比較多這樣的問題。這些分析失敗的執行檔反組譯後行數太少，因此要把它們從資料集內排除。

如圖 3-4 是一個分析失敗的執行檔，它的行數有 50 行，顯然不是一個正常的執行檔應該有的大小。圖 3-5 統計執行檔反組譯後的文字檔案的行數 (大於 10000 行的不列入圖中)，圖 3-6 是圖 3-5 中小於 100 行的詳細部分，兩張圖中可以看到，大約 50 行左右執行檔數量就不聚集的過於明顯，而且實際查看剛好 50 行的反組譯結果文字檔案(如圖 3-4)，發現也的確是分析失敗的檔案。所以研究者決定排除不管是良性程式或是惡意程式，反組譯後文字檔案行數不到 50 行的分析結果。

```
15                  ;-- rip:
16                  ;-- rflags:
17      / 68: fcn.00000000 (int64_t arg3, int64_t arg5);
18      |           ; arg int64_t arg3 @ rdx
19      |           ; arg int64_t arg5 @ r8
20      |           0x00000000      f1            int1
21      |           0x00000001      ed            in eax, dx
22      |           0x00000002      96            xchg esi, eax
23      |       ,==< 0x00000003      7150          jno 0x55
24      |       |   0x00000005      413330        xor esi, dword [r8]
25      |       ,===< 0x00000008      417060        jo 0x6b
26      |       ||   0x0000000b      1196acd501b0  adc dword [rsi - 0x4ffe2a54], edx
27      |       ||   0x00000011      5e            pop rsi
28      |       ||   0x00000012      10d0          adc al, dl
29      |       ||   0x00000014      c7c40c01f470  mov esp, 0x70f4010c
30      |       ||   0x0000001a      005008        add byte [rax + 8], dl
31      |       ||   0x0000001d      dd11          fst qword [rcx]
32      |       ||   0x0000001f      97            xchg edi, eax
33      |       ||   0x00000020      6400a2487d5f.  add byte fs:[rdx - 0x59a082b8], ah
34      |       ||   0x00000027      c3            ret
35      ..
36      |       |`--> 0x00000055      aa            stosb byte [rdi], al
37      |       | ||   0x00000056      aa            stosb byte [rdi], al
38      |       | ||   0x00000057      aa            stosb byte [rdi], al
39      |       | ||   0x00000058      aa            stosb byte [rdi], al
40      |       | ||   0x00000059      aa            stosb byte [rdi], al
41      |       | ||   0x0000005a      aa            stosb byte [rdi], al
42      |       | ||   0x0000005b      aa            stosb byte [rdi], al
43      |       | ||   0x0000005c      aa            stosb byte [rdi], al
44      |       | ||   0x0000005d      aa            stosb byte [rdi], al
45      |       | ||   0x0000005e      aa            stosb byte [rdi], al
46      |       | ||   0x0000005f      a1a08f0c80ff.  movabs eax, dword [0xf8001bff800c8fa0]
47      |       | ||   0x00000068      5f            pop rdi
48      |       | ||   0x00000069      00f0          add al, dh
49      |       `---> 0x0000006b      bf00f85f00    mov edi, 0x5ff800
50      \         `-> 0x00000070      fc            cld
51
```

圖 3-4：反組譯失敗的執行檔

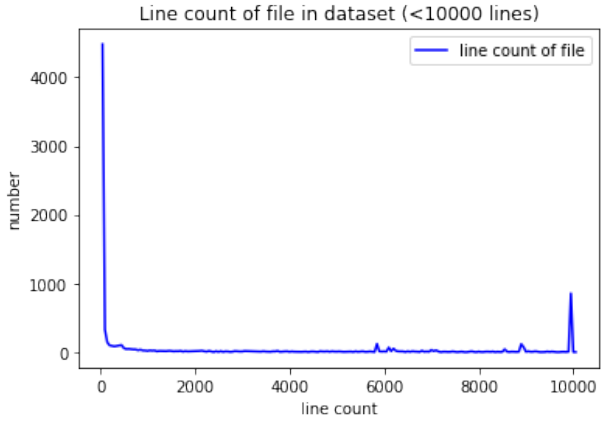



圖 3-5：小於 10000 行的反組譯結果文字檔案數量統計

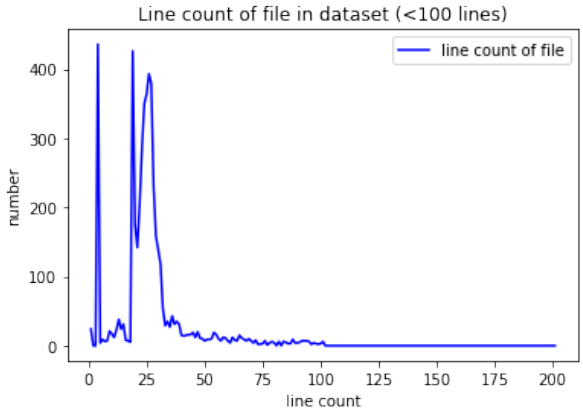
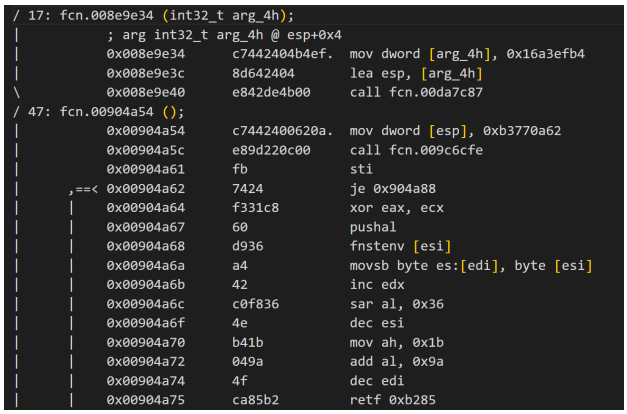



圖 3-6：小於 100 行的反組譯結果文字檔案數量統計

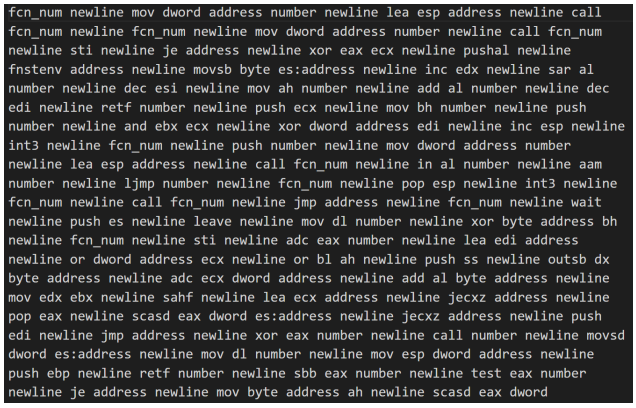
接著將註解與標點符號去除，所有的記憶體位置、常數、函式名稱、與換行編碼成單一的 token。因為註解是由反組譯器所生成，而記憶體位置也是反組譯器標記的位置，不會是真正執行時的記憶體位置。有些函式名稱 (如 fcn.008e9e34) 也是反組譯器生成的名稱，這是因為分析的目標執行檔經過 strip 處理，所以一些符號資訊會消失。因為對於分辨惡意程式與一般程式幫助不大，所以把他們處理成單一的 token。

圖 3-7 是處理前的反組譯結果文字檔案，而圖 3-8 則是處理後的反組譯結果文字檔案，可以看到換行被替換成 newline，而記憶體位置被替換成 address，這些都是被編碼後的結果。



```
/ 17: fcn.008e9e34 (int32_t arg_4h);
|           ; arg int32_t arg_4h @ esp+0x4
|           0x008e9e34      c7442404b4ef.  mov dword [arg_4h], 0x16a3efb4
|           0x008e9e3c      8d642404       lea esp, [arg_4h]
\           0x008e9e40      e842de4b00     call fcn.00da7c87
/ 47: fcn.00904a54 ();
|           0x00904a54      c7442400620a.  mov dword [esp], 0xb3770a62
|           0x00904a5c      e89d220c00     call fcn.009c6cfe
|           0x00904a61      fb             sti
|       ,==< 0x00904a62      7424           je 0x904a88
|       |   0x00904a64      f331c8         xor eax, ecx
|       |   0x00904a67      60             pushal
|       |   0x00904a68      d936           fnstenv [esi]
|       |   0x00904a6a      a4             movsb byte es:[edi], byte [esi]
|       |   0x00904a6b      42             inc edx
|       |   0x00904a6c      c0f836         sar al, 0x36
|       |   0x00904a6f      4e             dec esi
|       |   0x00904a70      b41b           mov ah, 0x1b
|       |   0x00904a72      049a           add al, 0x9a
|       |   0x00904a74      4f             dec edi
|       |   0x00904a75      ca85b2         retf 0xb285
```

圖 3-7：處理前的反組譯結果文字檔案



```
fcn_num newline mov dword address number newline lea esp address newline call
fcn_num newline fcn_num newline mov dword address number newline call fcn_num
newline sti newline je address newline xor eax ecx newline pushal newline
fnstenv address newline movsb byte es:address newline inc edx newline sar al
number newline dec esi newline mov ah number newline add al number newline dec
edi newline retf number newline push ecx newline mov bh number newline push
number newline and ebx ecx newline xor dword address edi newline inc esp newline
int3 newline fcn_num newline push number newline mov dword address number
newline lea esp address newline call fcn_num newline in al number newline aam
number newline ljmp number newline fcn_num newline pop esp newline int3 newline
fcn_num newline call fcn_num newline jmp address newline fcn_num newline wait
newline push es newline leave newline mov dl number newline xor byte address bh
newline fcn_num newline sti newline adc eax number newline lea edi address
newline or dword address ecx newline or bl ah newline push ss newline outsb dx
byte address newline adc ecx dword address newline add al byte address newline
mov edx ebx newline sahf newline lea ecx address newline jecxz address newline
pop eax newline scasd eax dword es:address newline jecxz address newline push
edi newline jmp address newline xor eax number newline call number newline movsd
dword es:address newline mov dl number newline mov esp dword address newline
push ebp newline retf number newline sbb eax number newline test eax number
newline je address newline mov byte address ah newline scasd eax dword
```

圖 3-8：處理後的反組譯結果文字檔案

表 3-1 是排除小於 50 行的分析結果後，並且經過前處理的資料集數量統計。用於訓練與測試機器學習模型，衡量模型設計的成效。

表 3-1：用於訓練與測試的資料集統計

| | 訓練資料數 | 測試資料數 | 總計 |
|---|---|---|---|
| 良性程式 | 2,061 | 515 | 2,576 |
| 惡意程式 | 5,674 | 515 | 6,189 |
| 總計 | 7,735 | 1,030 | 8,765 |

三、使用機器學習實作自然語言處理模型

(一) 程式運行環境

本研究使用 Keras 設計及訓練自然語言處理模型時，選擇在 Google Colaboratory 中運行，原因如下：

1. Google Colaboratory 是個虛擬機，不須為了環境問題而耗費額外時間成本。且它提供高效能的硬體協助運算，不須額外設備。以下是 Google Colaboratory 的硬體配置。

   CPU：Intel(R) Xeon(R) CPU @ 2.20GHz、GPU：Nvidia Tesla T4、RAM：12GiB

2. Google Colaboratory 可以直接使用 Tensorflow 和 Keras 等機器學習套件，無需額外安裝。
3. Google Colaboratory 可以和 Google 雲端硬碟互動，資料集存放在 Google 雲端硬碟就可以免除每次訓練還要上傳的時間(如圖 3-9)。

此外，在以下所有使用 keras 的自然語言處理模型中，均使用了 ModelCheckpoint 的 callback，把訓練過程中最好的模型存下來，並使用該模型進行測試。


圖 3-9：Google Colaboratory 執行視窗

另外，本研究使用 fastText 時的程式執行環境選擇在 Ubuntu 終端機執行，因為使用終端機進行 fastText 訓練較使用 Python 模組快，所以本研究選擇終端機。圖 3-10 是在 Ubuntu 終端機訓練 fastText 的畫面，在終端機訓練可以及時得到訓練數據，如 learning rate。

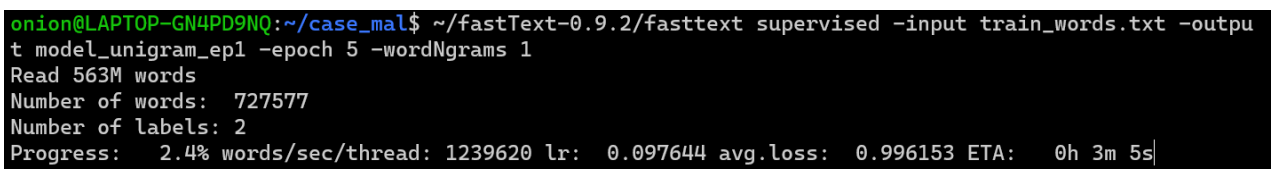



圖 3-10：在終端機訓練 fastText。

(二) 詞袋模型 (bag-of-words)

使用詞袋法的模型不考慮文本中的順序關係，將其視為一堆 token 的組合。因不考慮 token 間的順序關係，所以實作上比考慮關係的序列模型還要簡單。

雖然詞袋法不考慮順序關係，但如果要獲取局部的順序特徵，可以使用 n-gram 來得到局部的資訊。n-gram 採用滑動視窗的方式，在文本上額外一次擷取 n 個 token。例如當 n=2 時，句子「I am a student」就會被擷取成{“I”, “I am”, “am”, “am a”, “a”, “a student”, “student”}。根據不同的實作方法，有時會加上前後綴表示句子的開始及結束。

本研究根據對文本進行不同的編碼方式，實作了兩種使用詞袋法的自然語言處理模型，分別是 multi-hot 編碼及 TF-IDF 編碼。在這兩種模型中，均擷取 20000 個最常出現的 n-gram token，作為文本的編碼。

1. multi-hot 編碼

multi-hot 編碼用一個簡單的向量來呈現文本中的內容，該向量的每個元的值代表某個詞是否出現。若該詞在這個文本中出現，則向量該元的值是 1，否則為 0。

如果將這個代表文本的向量的維度設為在語料庫中詞的數量，會導致向量的維度太大，造成運算的負擔。因此，常見的作法是只計算那些最常見的詞，忽略那些在整個語料庫中出現很少次數的詞。在本研究中，限制使用 20000 個最常出現的 token。

以圖 3-11 為例，“The” 在 “The cat is black.” 中出現，因此向量上該元的值就是 1，但是 “in” 沒有在句子中出現，所以值是 0。圖 3-12 是使用 multi-hot 編碼的機器學習模型架構，在 Dense 層後添加了使用 0.5 Dropout Rate 的 Dropout 層，防止過擬合 (overfitting)。從圖中可以看到這是個非常輕量的模型，因此訓練及推論的速度不會過慢。


| The | cat | in | the | hat | is | black |
|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 0 | 1 | 1 |

圖 3-11：multi-hot 編碼示意圖

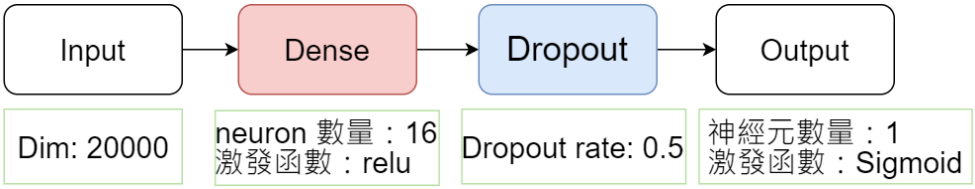



圖 3-12：使用 multi-hot 編碼的詞袋模型架構

(可訓練參數量：320,033 個)

2. TF-IDF 編碼

TF-IDF (Term Frequency-Inverse Document Frequency) 是一種統計方法 (Manning et al., 2008)，用於計算一個詞對於一份文本的重要程度，採用 TF-IDF 方法可以提取文本的特徵。

TF (Term Frequency) 的計算方式如式(1)所示，其中分子$n_{i,j}$是詞$t_i$在文件$d_j$中出現的次數，而分母$\sum_k n_{k,j}$則是文件$d_j$中所有詞的出現次數的和。一個詞在一份文件的 TF 值越高，就代表這個詞在這個文件越常出現。

$$\mathrm{TF}_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}} \qquad (1)$$

IDF (Inverse Document Frequency) 是另一種度量方法，計算一個詞$t_i$的 IDF 如式(2)所示，其中|D|是文件的總數量，$\{j: t_i \in d_j\}$是包含有詞$t_i$的文件數，將兩者相除後取對數值，即得到 IDF 值。最後，將 TF 值乘上 IDF 值，即為一個詞在那個文件的 TF-IDF 值。

$$\mathrm{IDF}_{i,j} = \log \frac{|D|}{|\{j : t_i \in d_j\}|} \qquad (2)$$

本研究採用 TF-IDF 編碼的自然語言處理模型的設計架構，與採用 multi-hot 編碼的模型相同 (如圖 3-12)。而一個前處理後的惡意程式反組譯結果，經過 TF-IDF 編碼後如圖 3-13 所示，其中第 0 元是[UNK] (代表未看過或是未編碼的 token)，第 1 元是 newline，第 2 元是 address，它們各自的 TF-IDF 值分別是 166.10194、3965.7903 以及 1766.6549。

```
tf.Tensor([ 166.10194 3965.7903  1766.6549  ...    0.
0.          0.      ], shape=(20000,), dtype=float32)
```

圖 3-13：經 TF-IDF 編碼後的向量

(三) 序列模型

在序列模型中則考慮各個詞之間的關係。本研究實作了兩種模型，在模型的設計中分別使用了雙向 LSTM 以及 Transformer encoder，並且有 one-hot 以及詞嵌入 (word embedding) 兩種編碼方式。為了控制輸入的大小，在以上模型中均只擷取文本中前 600 個詞作為輸入。

1. 雙向 LSTM

LSTM 是一種循環神經網路 (Recurrent Neural Network, RNN)，透過儲存上一個迭代的資訊來保留序列中的特徵。圖 3-14 是一個 LSTM 單元的示意圖，透過 input gate、forget gate 的機制，LSTM 可以記憶或遺忘存在 memory cell 的資訊，其儲存前面序列的資訊。

單一的 LSTM 只考慮單向的序列，而雙向 LSTM 則同時考慮正向以及逆向的序列，如此可以從逆向的序列擷取特徵。在使用循環神經網路進行自然語言處理任務時，常使用雙向的模型設計，因為一個詞在一句話中的特徵，取決於它的前後文，而本研究認為這放在前處理後的組合語言也適用，因此採用雙向的 LSTM。

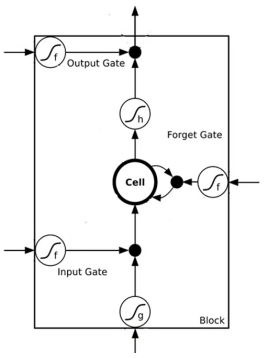



圖 3-14：LSTM 單元示意圖

(資料來源：Lee, 2017)

(1) one-hot 編碼

本研究實作了以 one-hot 編碼作為雙向 LSTM 輸入的模型，one-hot 編碼將每個詞變成一個 one-hot 向量，也就是向量中只有一個元是 1，其餘皆是 0。因為只擷取文本中前 600 個詞，而這邊將 20000 個最常出現的 token 進行編碼 (包含[UNK])。因此，一個反組譯並前處理後的惡意程式文本會變成一個 600x20000 的矩陣。

圖 3-15 是本研究採用 one-hot 編碼的雙向 LSTM 模型架構，其中在雙向 LSTM 層後也使用了 Dropout 層來防止過擬合。

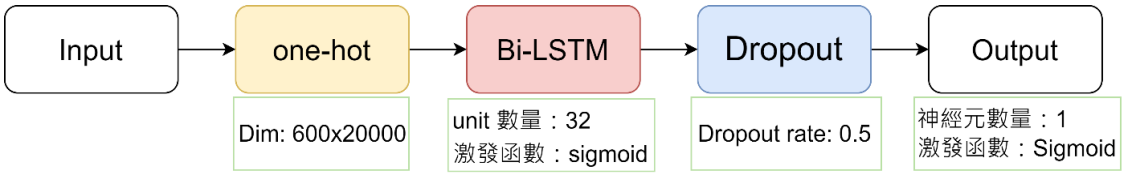



圖 3-15：使用 one-hot 編碼的雙向 LSTM 模型架構

(可訓練參數量：5,128,513 個)

(2) 詞嵌入 (Word Embedding) 編碼

詞嵌入是另一種將單詞編碼的方式，與 one-hot 編碼不同，詞嵌入會考慮上下文的關係，同個詞擁有不同的上下文則會有不同的編碼。詞嵌入是透過一個神經層被訓練出來的，在本研究中，設定每個詞被編碼成一個 100 維的詞嵌入向量，而每個文本則擷取前 600 個單詞，因此一個反組譯並前處理後的惡意程式文本會變成一個 600x100 的張量。

圖 3-16 是採用詞嵌入編碼的雙向 LSTM 模型架構。另外，對於那些長度不足 600 的惡意程式反組譯文本，在此採用遮罩 (masking) 的方式，使後面的雙向 LSTM 層不會對後面空的內容進行迭代，以獲取更好的模型表現。

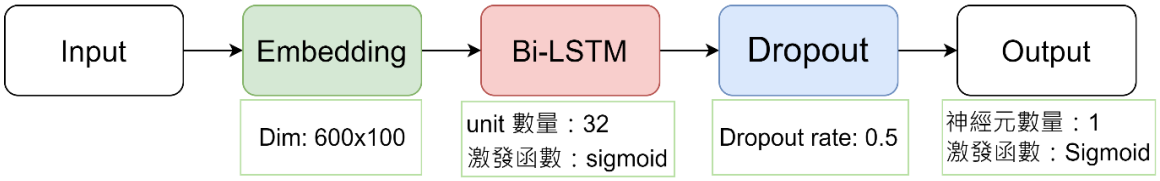



圖 3-16：使用詞嵌入編碼的雙向 LSTM 模型架構

(可訓練參數量：5,194,049 個)

2. Transformer Encoder

Transformer 是 Google 提出的 Seq2seq 模型，分為 encoder 及 decoder 兩部分。圖 3-17 是 Transformer 原論文“Attention Is All You Need” (Vaswani, *et. al.*, 2017) 中 Transformer 示意圖的 encoder 部分，其中採用了 multi-head 的 attention 機制，如此可以捕捉序列中詞之間的相對關係，並且使用殘差連接 (residual connection) 及 layer normalization，避免梯度消失以及正規化，中間還有一般的 Dense 層 (Feed Forward)，用於學習序列中更多的局部特徵。

本研究擷取 encoder 的部分，作為模型的一部份使用。圖 3-18 是使用了 Transformer encoder 的模型架構，原文即採用了詞嵌入作為最初的編碼，所以在此也在一開始使用詞嵌入。而原文也使用了位置編碼 (Positional encoding)，為輸入的向量添加位置的資訊，而本研究則實作了包含以及不包含有位置編碼的兩種模型，並比較之間的差異，其中本研究使用學習位置嵌入向量的方式進行位置編碼，也就是 positional embedding 方法。

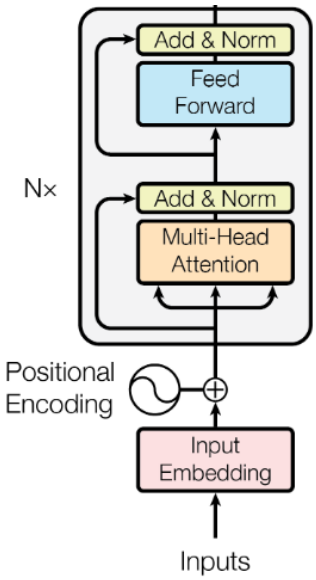



圖 3-17：Transformer 的 encoder 部分
(資料來源：Vaswani, *et. al.*, 2017: 3)

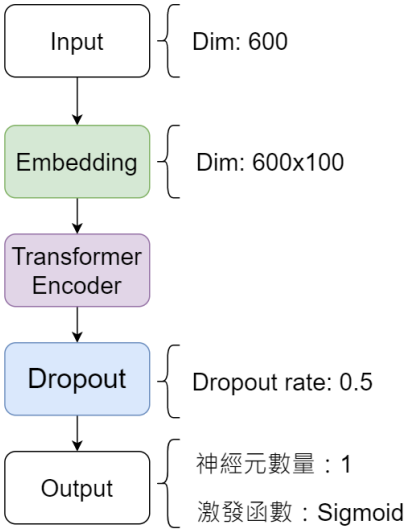



圖 3-18：使用 Transformer encoder 的模型架構
(無位置編碼，可訓練參數量：5,664,033 個
有位置編碼，可訓練參數量：5,817,633 個)

(四) fastText 分類學習模型

fastText 可以執行監督式學習文本分類任務，也可以進行非監督式的訓練文字轉向量任務。其中在訓練監督式的文本分類模型時，同時也可以產生文字轉向量的模型。fastText 最大的特點是擁有比較短的訓練時間，且它可以在多核 CPU 進行訓練。

fastText 的模型架構有三層，分別是輸入層、隱藏層、輸出層 (如圖 3-19)。此模型架構與 Word2Vec 的 CBOW 模型類似，但 CBOW 用上下文來推論被挖空的文字，而 fastText 用整個文本推論該文本的標籤。

1. 輸入層

   fastText 以整個文本的詞嵌入 (word embedding)，以及 n-gram 特徵的向量相加取平均後做為輸入，此向量可以視為代表整個文本的向量。

2. 隱藏層

   在隱藏層會把輸入層產生的單個向量，乘上權重矩陣，也就是 fastText 模型需要學習的參數。乘上這個矩陣後，會得到另一個向量，代表文本的隱藏特徵。

3. 輸出層

   輸出層會產生文本分類的機率分布，因此會將隱藏層產生的向量進行 softmax 處理。fastText 採用的是階層 softmax，此函數使用霍夫曼樹 (Huffman tree) 的方式來編碼類別，當一個類別比較常出現，則他編碼後的路徑會比較短，因此計算所需的時間也比較短。如此階層 softmax 能夠有效率的計算輸出的機率分布，這也是 fastText 的訓練及推論速度比較快的原因之一。

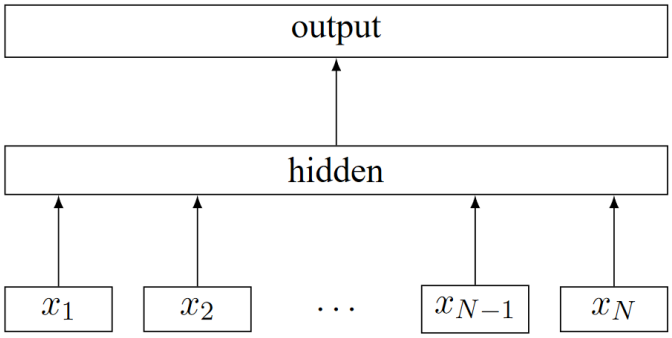



圖 3-19：fastText 模型架構

(資料來源：Joulin, *et. al*., 2016: 2)

四、以圖神經網路辨認惡意程式

研究者嘗試過使用圖神經網路 (Graph Neural Network) 進行惡意程式辨認，作法是使用執行檔的函式呼叫圖 (Call Graph) 作為圖神經網路運算的圖。該圖的邊是函式間呼叫的關係，而節點上的資訊則是該函式的組合語言，透過將組合語言前處理後，傳給 fastText 學習詞嵌入，作為節點上的向量。這樣把函式呼叫的關係作為邊，詞嵌入作為節點，期望圖神經網路可以進行惡意程式辨認。圖 4-18 是使用函式函式呼叫圖進行圖神經網路運算的示意圖，其中函式呼叫圖上的黃色條狀物表示向量。


圖 4-18：以函式呼叫圖進行圖神經網路運算示意圖

研究者實作 Graph Convolutional Network (GCN) 以及 Deep Graph Convolutional Neural Network (Zhang, 2018)，發現兩者在辨識惡意程式方面均無作用，對於良性程式或惡意程式，模型均輸出辨識結果為惡意程式，準確率等指標請見表 4-5。

圖神經網路無法識別惡意程式的原因可能是模型大小，研究者所實作的圖神經網路，可訓練參數量均小於 100,000，比本研究之自然語言處理模型還少，因此可能沒辦法學到有關惡意程式的特徵。

五、表 4-5：圖神經網路模型識別惡意程式之數據

| | Accuracy | Precision | Recall | F1-score |
|---|---|---|---|---|
| GCN | 50% | 50% | 100% | 66.6% |
| DGCNN | 50% | 50% | 100% | 66.6% |

六、衡量指標

在本研究所有實驗中，為了避免單一衡量指標所帶來的偏差，在此使用準確率 (Accuracy)、精確率 (Precision)、召回率 (Recall)、以及 F1-score。以上指標的定義來自於混淆矩陣 (Powers, 2011)，也就是二元分類問題中所有可能的情況 (見表 3-2)。準確率是所有樣本中，有多少樣本是判斷正確的。精確率是所有預測是陽性的樣本中，有多少是預測正確的。召回率是所有真實是陽性的樣本中，有多少是預測正確的。F1-score 則是綜合了精確率與召回率，所設計出來的指標。式(3)到式(6)為各衡量指標之算法，其中 TP 表示 True Positive，其他縮寫以此類推。

表 3-2：混淆矩陣 (Confusion Matrix)

| | | 實際值 Actual Value | |
|---|---|---|---|
| | | Positive | Negative |
| 預測值 Predicted Value | Positive | **T**rue **P**ositive | **F**alse **P**ositive |
| | Negative | **F**alse **N**egative | **T**rue **N**egative |

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \quad (3)$$

$$\text{Precision} = \frac{TP}{TP + FP} \quad (4)$$

$$\text{Recall} = \frac{TP}{TP + FN} \quad (5)$$

$$\text{F1-score} = \frac{2}{\frac{1}{\text{Precision}} + \frac{1}{Recall}} = \frac{2TP}{2TP + FP + FN} \quad (6)$$

# 肆、研究結果與討論

本研究進行五個實驗，實驗一到實驗三對應本研究的研究目的 (一) 與 (二)，實驗四對應實驗目的 (三)，實驗五是為了以額外的資料，對本研究訓練出的模型進行驗證，以雙重確認模型的有效性。實驗中訓練資料集有 7,735 支惡意程式，測試資料集有 1,030 支惡意程式，詳見表 3-1 (p.8)。

一、實驗一：詞袋模型用於識別惡意程式

此實驗分別以 1-gram 的 multi-hot 以及 TF-IDF 的編碼實作詞袋模型，並且比較其差別。其中 multi-hot 的詞袋模型在識別惡意程式上，準確率達到 98.6%。

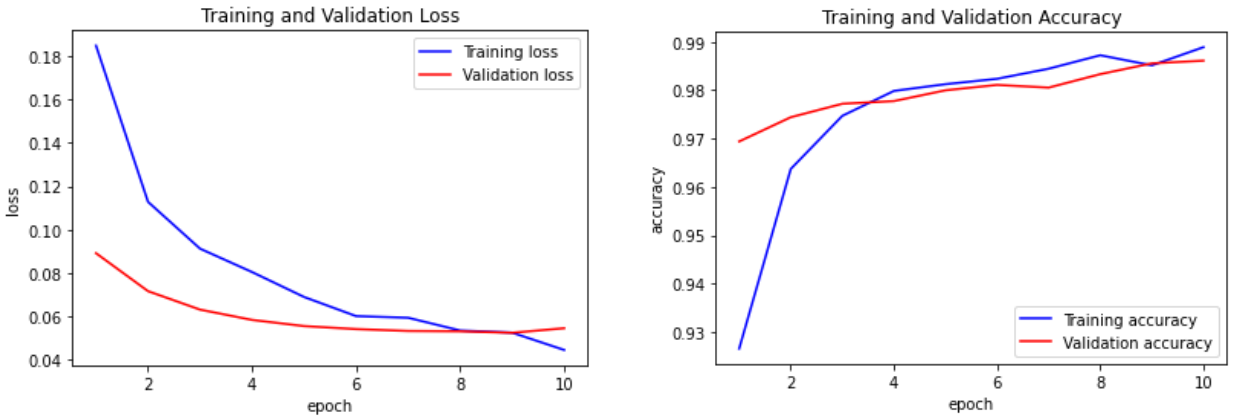



圖 4-1：multi-hot 編碼的詞袋模型訓練之損失及準確率變化圖

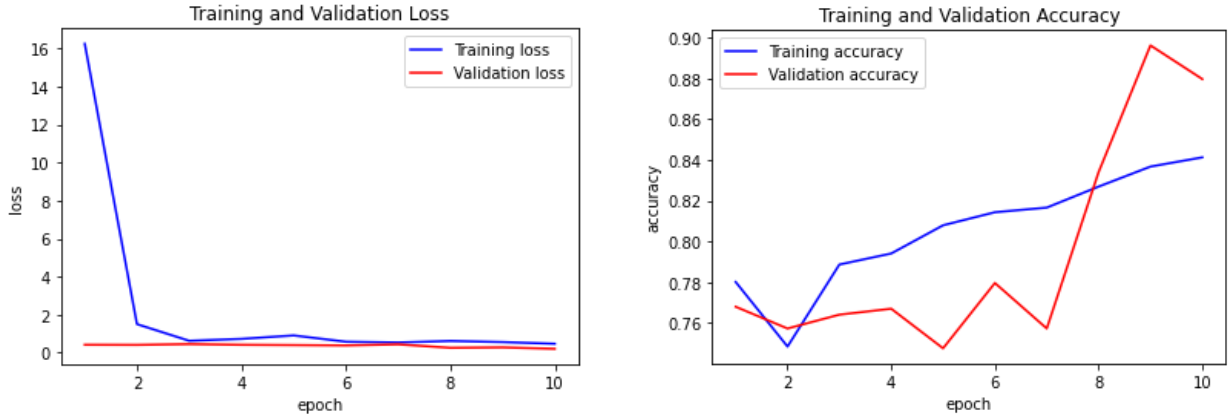



圖 4-2：TF-IDF 編碼的詞袋模型訓練之損失及準確率變化圖

表 4-1 呈現兩種編碼的詞袋模型分別在測試資料集上，識別惡意程式的準確率、精確率、召回率、以及 F1-score。

表 4-1：兩種編碼方式的詞袋模型表現比較

| | 準確率 | 精確率 | 召回率 | F1-score |
|---|---|---|---|---|
| 詞袋模型&multi-hot | **98.6%** | **98.8%** | 99.1% | **99.0%** |
| 詞袋模型&TF-IDF | 88.0% | 80.6% | **100%** | 89.3% |

其中 multi-hot 編碼的模型 F1-score 為 99.0%，TF-IDF 編碼的模型 F1-score 為 89.3%。為何以 TF-IDF 的準確率會較低的原因，研究者認為可能是資料集太小。從 MalwareBazzar 下載惡意程式本身具有限制，而本研究又對不合格的資料進行排除，因此造成資料量不足以讓 TF-IDF 編碼呈現出文本的特徵，進而導致 TF-IDF 編碼不如 multi-hot 編碼。

二、實驗二：序列模型用於識別惡意程式

此實驗分別以雙向 LSTM 及 Transformer 的 encoder 實作序列模型。雙向 LSTM 分為 one-hot 以及 embedding 兩種編碼；而 Transformer 則分為有無進行位置編碼兩種模型，均使用 embedding 作為編碼。

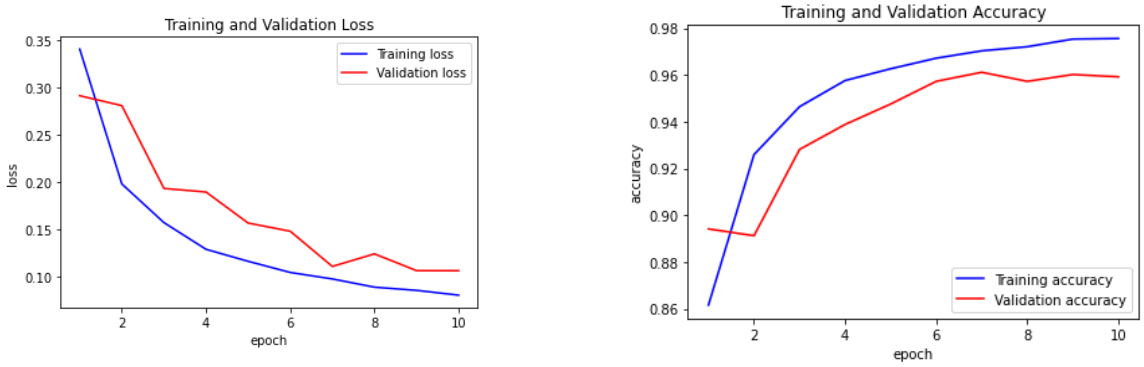



圖 4-3：one-hot 編碼的雙向 LSTM 模型訓練之損失及準確率變化圖

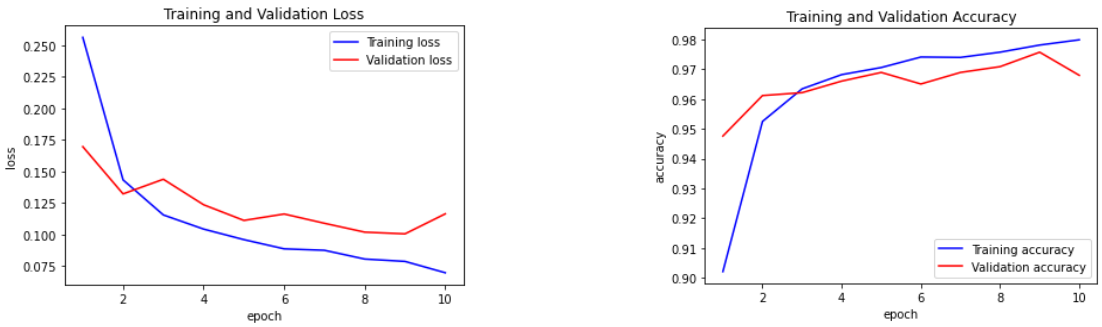



圖 4-4：embedding 編碼的雙向 LSTM 模型訓練之損失及準確率變化圖

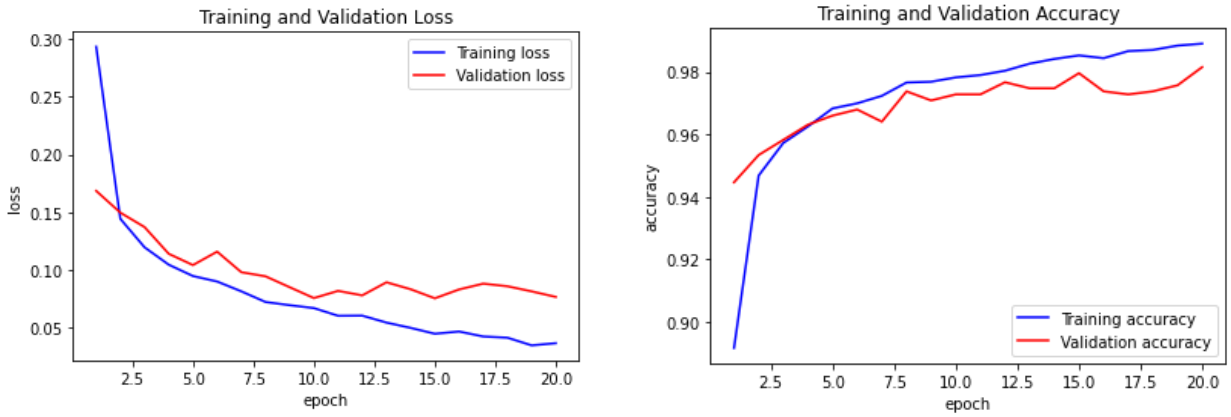



圖 4-5：無位置編碼的 Transformer 模型訓練之損失及準確率變化圖

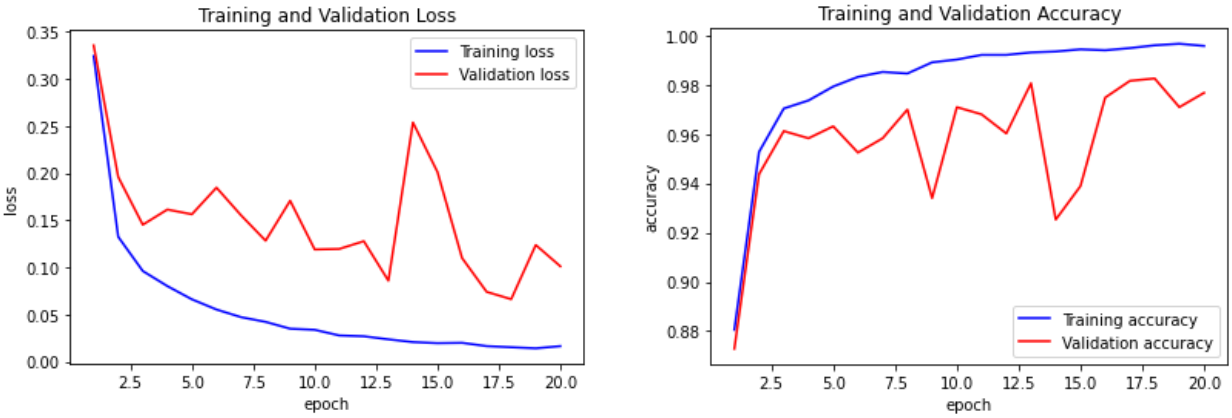



圖 4-6：有位置編碼的 Transformer 模型訓練之損失及準確率變化圖

表 4-2 呈現不同的序列模型分別在測試資料集上，識別惡意程式的準確率、精確率、召回率、以及 F1-score。其中有位置編碼的 Transformer 在識別惡意程式上，準確率達到 98.3%。

表 4-2：不同序列模型表現比較

| | 準確率 | 精確率 | 召回率 | F1-score |
|---|---|---|---|---|
| 雙向 LSTM&one-hot | 95.9% | 93.6% | 98.6% | 96.0% |
| 雙向 LSTM&embedding | 97.6% | 95.9% | **99.4%** | 97.6% |
| Transformer 無位置編碼 | 98.0% | 97.1% | 98.8% | 98.0% |
| Transformer 有位置編碼 | **98.3%** | **97.7%** | 98.8% | **98.2%** |

在序列模型中，以具有位置編碼的 Transformer encoder 模型最佳，擁有 98.2%的 F1-score，而使用 Transformer encoder 的模型比使用雙向 LSTM 的模型還要好。研究者認為可能是因為 Transformer 的架構較複雜，而且具備 attention 機制，使惡意程式具有的特徵更能被學習到。而有位置編碼的 Transformer encoder 模型，比沒有位置編碼的模型還要好，是因為位置編碼可以擷取出序列中順序的特徵，從而提升模型的表現。

三、實驗三：fastText 模型用於識別惡意程式

此實驗使用 fastText 來辨識前處理後的惡意程式反組譯文本，並比較了訓練 40 epoch 以及 50 epochs 下模型的準確率。因為 fastText 不支援在訓練途中進行驗證以及測量準確率，所以此處呈現訓練中訓練資料集損失的變化。

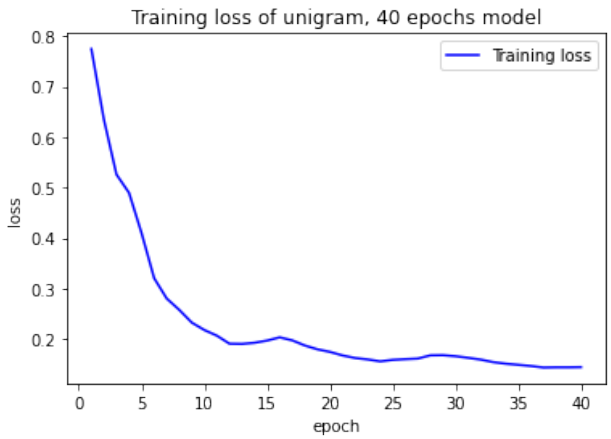



圖 4-7：fastText 訓練 40 epochs 之損失變化

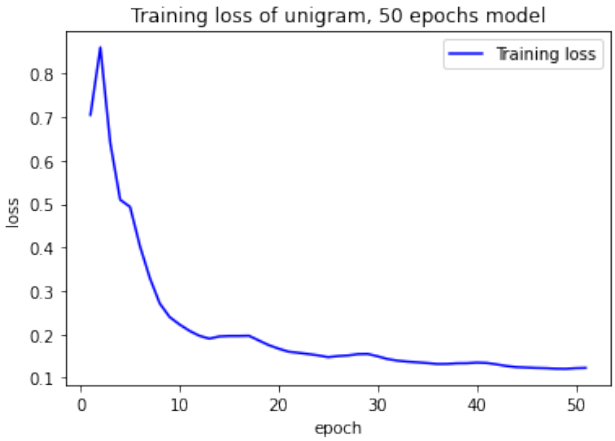



圖 4-8：fastText 訓練 50 epochs 之損失變化

表 4-3 呈現訓練不同 epoch 數的 fastText 模型分別在測試資料集上，識別惡意程式的準確率、精確率、召回率、以及 F1-score。其中訓練 50 個 epoch 的 fastText 模型準確率為 96.2%。

表 4-3：不同訓練 epoch 數之 fastText 模型表現數據

| | 準確率 | 精確率 | 召回率 | F1-score |
|---|---|---|---|---|
| fastText，40 epochs | 95.8% | 95.8% | 95.8% | 95.8% |
| fastText，50 epochs | **96.2%** | **96.2%** | **96.2%** | **96.2%** |

綜合各種指標比較詞袋模型、序列模型、以及 fastText，發現 multi-hot 編碼的詞袋模型比使用了 Transformer encoder 的模型好，其中準確率、精確率、召回率、F1-score 均為 multi-hot 編碼的詞袋模型優於使用 Transformer encoder 的模型。這可能是因為比起序列中的特徵，某個詞是否出現對於識別惡意程式更為重要。比如其他進行識別惡意程式研究常用的 API 呼叫，某特定的 API 是否出現在組合語言文本中，對於本研究識別惡意程式是重要的特徵。

四、實驗四：比較不同 n-gram 模型對模型成效影響

本實驗比較 multi-hot 的詞袋模型以及 fastText 模型在不同 n-gram 下的準確率，一共測試了 2-gram、3-gram、4-gram、5-gram 四種模型，結果如下圖所示

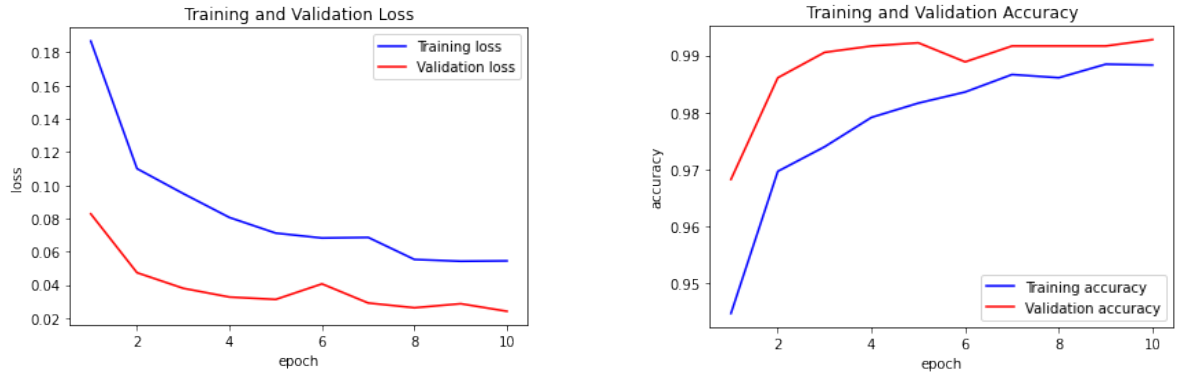



圖 4-9：2-gram 的 multi-hot 編碼詞袋模型訓練之損失及準確率變化圖

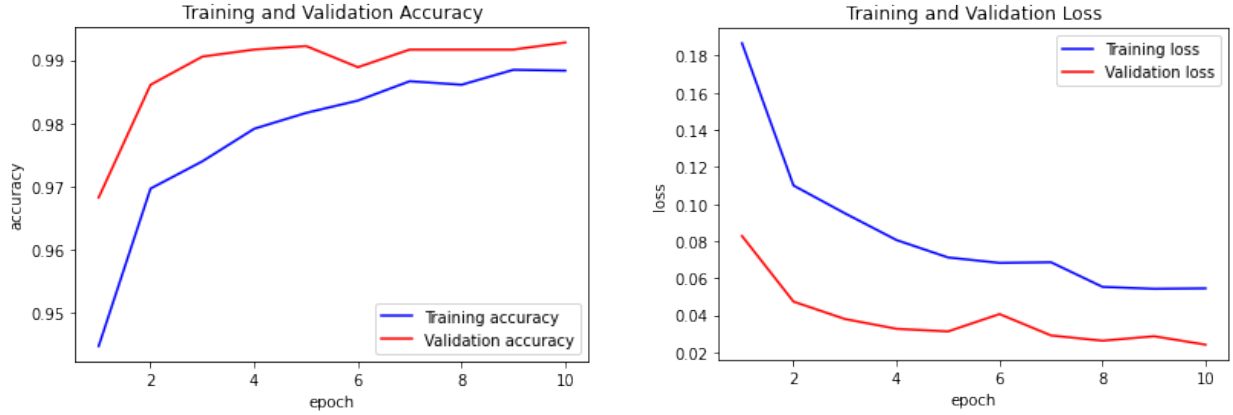



圖 4-10：3-gram 的 multi-hot 編碼詞袋模型訓練之損失及準確率變化圖

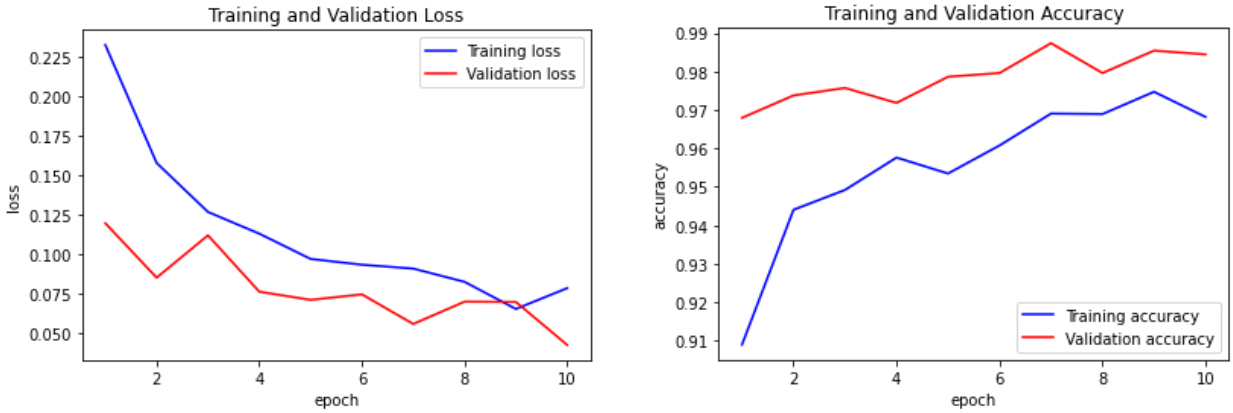



圖 4-11：4-gram 的 multi-hot 編碼詞袋模型訓練之損失及準確率變化圖

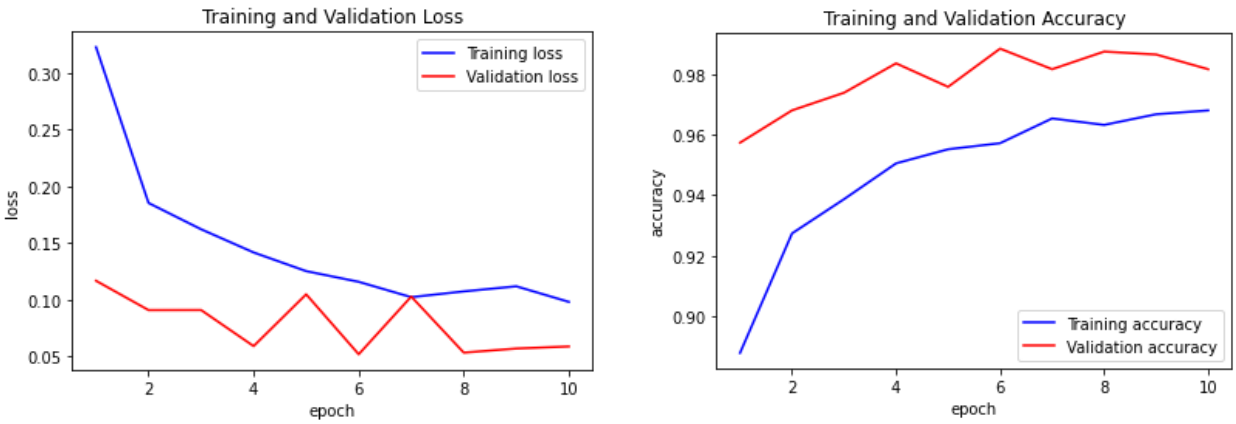



圖 4-12：5-gram 的 multi-hot 編碼詞袋模型訓練之損失及準確率變化圖

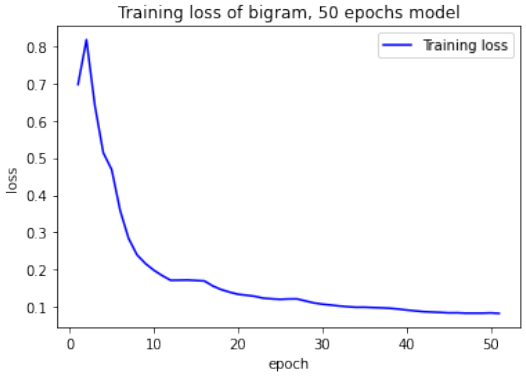



圖 4-13：fastText 2-gram 之損失變化

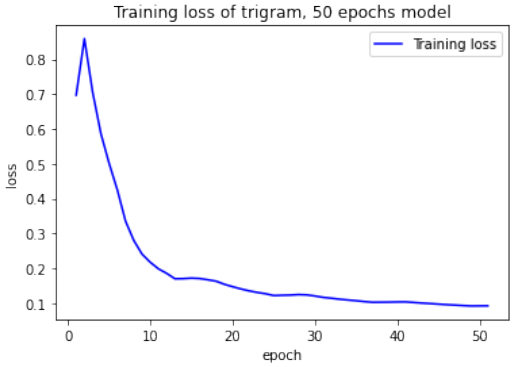



圖 4-14：fastText 3-gram 損失變化

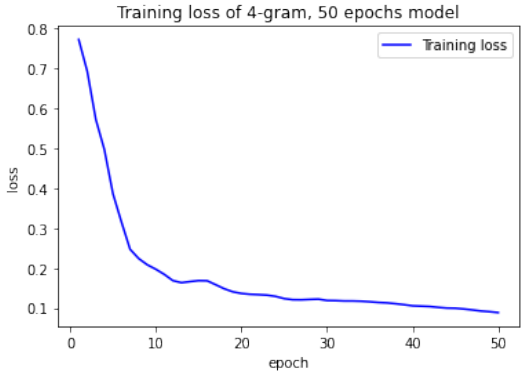



圖 4-15：fastText 4-gram 之損失變化

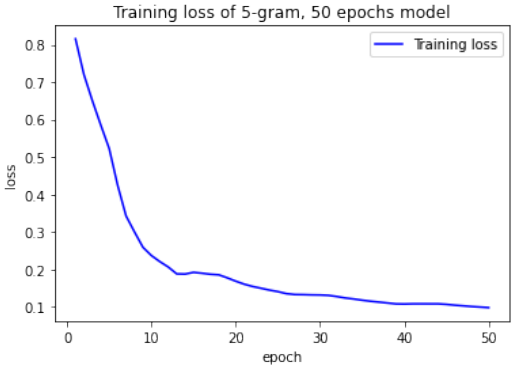



圖 4-16：fastText 5-gram 之損失變化

表 4-4 為四種 n-gram 模型在測試資料集上的準確率，。實驗中 multi-hot 詞袋模型與 fastText 模型分別在 2-gram 與 3-gram 有最高的準確率，分別為 99.4%以及 96.9%，可以有效識別惡意程式。

表 4-4：四種 n-gram 模型之準確率比較

| | 準確率 | | 精確率 | | 召回率 | | F1-score | |
|---|---|---|---|---|---|---|---|---|
| | 詞袋 | fastText | 詞袋 | fastText | 詞袋 | fastText | 詞袋 | fastText |
| 2-gram | **99.4%** | 93.4% | **99.3%** | 93.4% | **99.9%** | 93.4% | **99.6%** | 93.4% |
| 3-gram | 99.1% | **96.9%** | 99.1% | **96.9%** | 99.5% | **96.9%** | 99.3% | **96.9%** |
| 4-gram | 98.4% | 84.7% | 98.9% | 84.7% | 98.8% | 84.7% | 98.9% | 84.7% |
| 5-gram | 98.8% | 96.6% | 98.5% | 96.6% | 99.2% | 96.6% | 98.8% | 96.6% |

根據表 4-4，四種 n-gram 模型在使用 multi-hot 編碼的詞袋模型中，以 2-gram 的準確率最高，3-gram 的模型次之，4-gram 的模型較低。而 fastText 使用不同的 n-gram，則 3-gram 的準確率最高，5-gram 的模型次之，4-gram 的模型最低。研究者認為 2-gram 與 3-gram 的模型能夠優於其他 n-gram 的原因，是因為組合語言指令大多由 2 或 3 個單詞所構成 (如”push ebp、mov ebp esp”)，所以模型可以從 2-gram 或 3-gram 學習到區域的特徵。

五、實驗五：WannaCry 之辨認

本研究為了額外對自然語言處理的模型成效進行驗證，選擇使用 2017 年影響全球的勒索病毒 WannaCry 作為標的，其樣本取自 MalwareBazaar，且不存在於本研究之訓練與測試資料集。首先將 WannaCry 執行檔反組譯並且前處理後，再輸入不同的模型，觀察不同模型的輸出。圖 4-17 為 WannaCry 反組譯後並經過前處理的檔案，模型均以此文本進行預測。

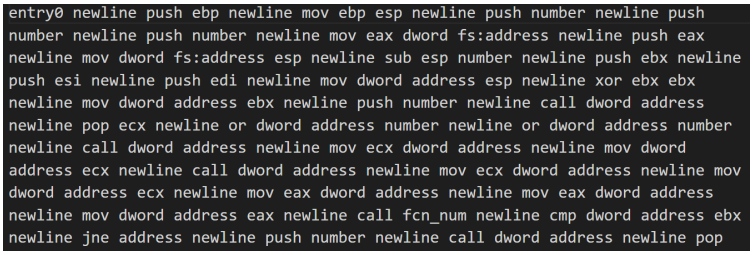



圖 4-17：WannaCry 反組譯後經過前處理的文本 (節錄)

表 4-5 為不同模型的輸出結果，其中所有模型均成功辨認，原因可能除了原本各模型的準確率本身就高，也可能是因為 WannaCry 本身是個惡意程式的特徵明顯，因此所有模型均可成功識別。

表 4-5：各模型對於 WannaCry 之辨識結果

| 模型 | 辨識結果 | 模型 | 辨識結果 |
|---|---|---|---|
| 詞袋&multi-hot | 辨識成功 | 2-gram 詞袋&multi-hot | 辨識成功 |
| 詞袋& TF-IDF | 辨識成功 | 3-gram 詞袋&multi-hot | 辨識成功 |
| 雙向 LSTM&one-hot | 辨識成功 | 4-gram 詞袋&multi-hot | 辨識成功 |
| 雙向 LSTM&embedding | 辨識成功 | 5-gram 詞袋&multi-hot | 辨識成功 |
| Transformer<br>無位置編碼 | 辨識成功 | 2-gram fastText | 辨識成功 |
| Transformer<br>有位置編碼 | 辨識成功 | 3-gram fastText | 辨識成功 |
| fastText | 辨識成功 | 4-gram fastText | 辨識成功 |
| | | 5-gram fastText | 辨識成功 |

總結研究者在本研究提出的方法，因為直接使用執行檔，並且運用 Radare2 對執行檔反組譯，取得後續自然語言處理模型進行辨識的文本。這樣的好處是直接使用執行檔，不需要任何原始碼即可進行惡意程式識別。因為一般來說，惡意程式不會公開原始碼，

因此，使用執行檔更有利於一般使用者使用本研究的模型進行惡意程式識別。而本研究之 2-gram multi-hot 編碼詞袋模型識別之 F1-score 為 99.6%，可以有效辨識惡意程式。

六、與其他研究之比較

為了瞭解本研究的模型和其他使用機器學習方法的準確率差異，在此分別與 Lu (2019) 使用 opcode 的 LSTM 模型，以及 Rahali 與 Akhloufi (2021) 使用 Android 原始碼的 BERT 模型進行比較，並列出各研究所提出之模型的最佳準確率。本研究提出之方法可以直接以執行檔進行惡意程式辨認，其準確率也比類似研究高。

表 4-6：與其他研究比較模型準確率

| 模型來源 | 所提出之最佳的模型準確率 |
|---|---|
| Lu, R. (2019). Malware detection with lstm using opcode language. | 97.87% |
| Rahali, A., & Akhloufi, M. A. (2021). MalBERT: Using transformers for cybersecurity and malicious software detection. | 97.61% |
| 本研究之 2-gram multi-hot 詞袋模型 | **99.40%** |

# 伍、結論

一、本研究所提出的自然語言處理模型，以 2-gram 的 multi-hot 編碼的詞袋模型表現最好，F1-score 為 99.6%，可有效識別惡意程式。

二、本研究實作之 1-gram 詞袋模型中，使用 multi-hot 編碼來識別惡意程式 F1-score 最高，準確率為 98.6%。

三、本研究實作之序列模型中，使用 Transformer encoder 加上位置編碼的模型 F1-score 最高，為 98.2%。fastText 模型的 F1-score 則為 96.2%。

四、比較不同 n-gram 的模型來辨認惡意程式，使用 multi-hot 編碼的詞袋模型在 2-gram 的準確率最高，fastText 則是使用 3-gram 擁有最高的準確率。

五、與其他使用自然語言處理技術，辨認惡意程式之研究比較，本研究之模型準確率較高。

六、未來展望：

(一) 擴大執行檔的資料集至其它系統或行動端，使模型的應用範圍更廣。

(二) 針對特定的 API 呼叫擷取特徵，實作更多自然語言處理模型並比較差異。

(三) 將可以有效分辨惡意程式的模型公開，架設網頁或開發防毒軟體，讓使用者可以使用本研究所研發的技術，進行惡意程式辨識。

# 陸、參考文獻資料


Hung-yi Lee (2017 年 1 月 25 日)。ML Lecture 21-1: Recurrent Neural Network (Part I)。YouTube。2023 年 2 月 26 日，取自：https://youtu.be/xCGidAeyS4M

Joulin, A., Grave, E., Bojanowski, P., & Mikolov, T. (2016). Bag of Tricks for Efficient Text Classification. *arXiv preprint arXiv:1607. 01759.*

Lu, R. (2019). Malware detection with lstm using opcode language. *arXiv preprint arXiv:1906.04593.*

Manning, C.D., Raghavan, P., & Schutze, H. (2008). Scoring, term weighting, and the vector space model. *Introduction to Information Retrieval.* p. 100.

Powers, David M. W. (2011). Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness & Correlation. *Journal of Machine Learning Technologies. 2 (1):* 37–63.

Rahali, A., & Akhloufi, M. A. (2021). MalBERT: Using transformers for cybersecurity and malicious software detection. *arXiv preprint arXiv:2103.03806.*

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems, 30.*

Zhang, M., Cui, Z., Neumann, M., & Chen, Y. (2018). An end-to-end deep learning architecture for graph classification. In *Proceedings of the AAAI conference on artificial intelligence* (Vol. 32, No. 1).

# 【評語】052502

此作品運用自然語言處理技術，建立辨識惡意程式的模型。首先搜集良性及惡意執行檔，進行反組譯及前處理以建立資料集。接著使用反組譯後的組合語言檔作為文本來訓練模型以區分良性和惡意程式。實驗中比較詞袋模型、序列模型、fastText 以及不同 n-gram 對模型效能（accuracy, precision, recall, F1 score）的影響，並將結果與其他相似研究進行比較。此作品具有實用價值且實驗探討完整。

# 作品海報

## 摘要

本研究旨在運用自然語言處理技術，建立辨識惡意程式的模型。首先搜集良性及惡意執行檔，進行反組譯及前處理以建立資料集。使用反組譯後的組合語言檔作為文本，訓練模型以區分良性和惡意程式。研究比較詞袋模型、序列模型、fastText 以及不同 n-gram 對模型的影響，並將結果與其他相似研究比較。

研究結果顯示。詞袋模型以使用 multi-hot 編碼表現最佳，序列模型有位置編碼的 Transformer encoder 表現最優。在不同 n-gram 的比較，2-gram 詞袋模型在 PE 資料集上識別惡意程式， F1-score 達到 99.6%，且本研究的識別準確率優於其他相似研究。

## 研究動機

在資訊科技蓬勃發展的現代社會，電腦已成為人們不可或缺的生活工具。但是，電腦每天均存在遭惡意程式感染的風險。感染惡意程式可能導致不同程度的損失，包括個人電腦損毀和資料外洩，甚至危及企業和國家安全。例如，2017 年的勒索病毒 WannaCry 曾經在全球肆虐，造成數十億美元的損失。因此，如何有效識別惡意程式的防治手段，已是電腦中不可或缺的一部份。

傳統的防毒軟體通常透過建立資料庫和規則來辨識惡意程式，但是當惡意程式不斷變種時，傳統防毒軟體可能無法及時辨識出新的惡意程式，近期結合了機器學習的防毒軟體成為趨勢，而且具有更好的防毒效果。因此，本研究希望能夠運用自然語言處理技術，實作可以識別惡意程式的模型。

## 研究目的

(一) 反組譯執行檔後，利用自然語言處理技術實作，可以分辨惡意程式的機器學習模型。

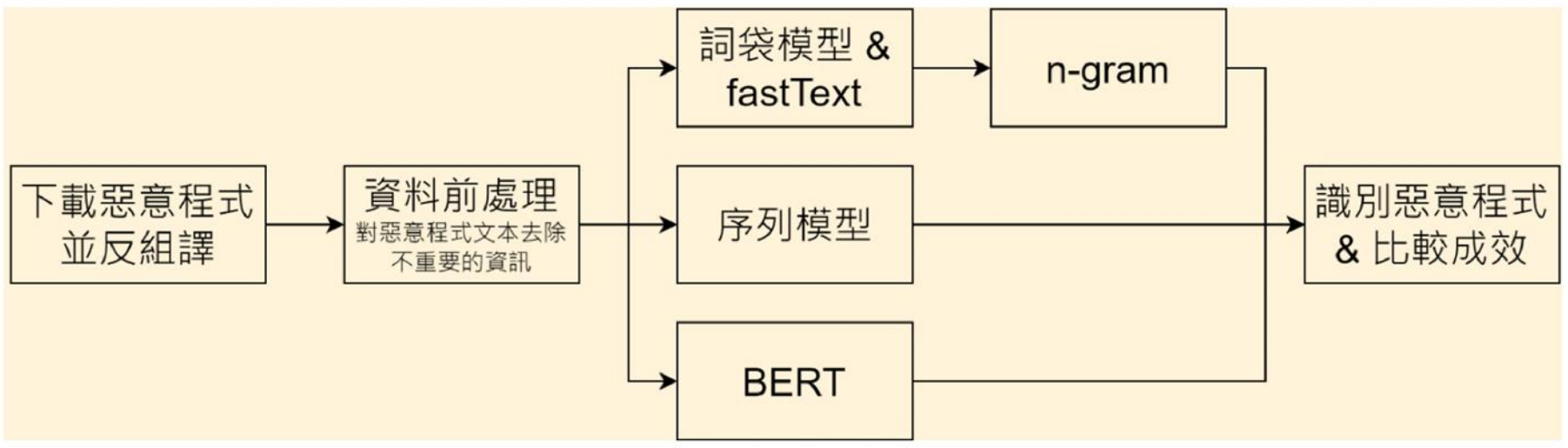
(二) 實作不同的自然語言處理機器學習模型，並比較識別惡意程式成效的差異。

(三) 以不同的n-gram實作模型，比較不同n-gram對於識別惡意程式成效的影響。

## 研究過程或方法

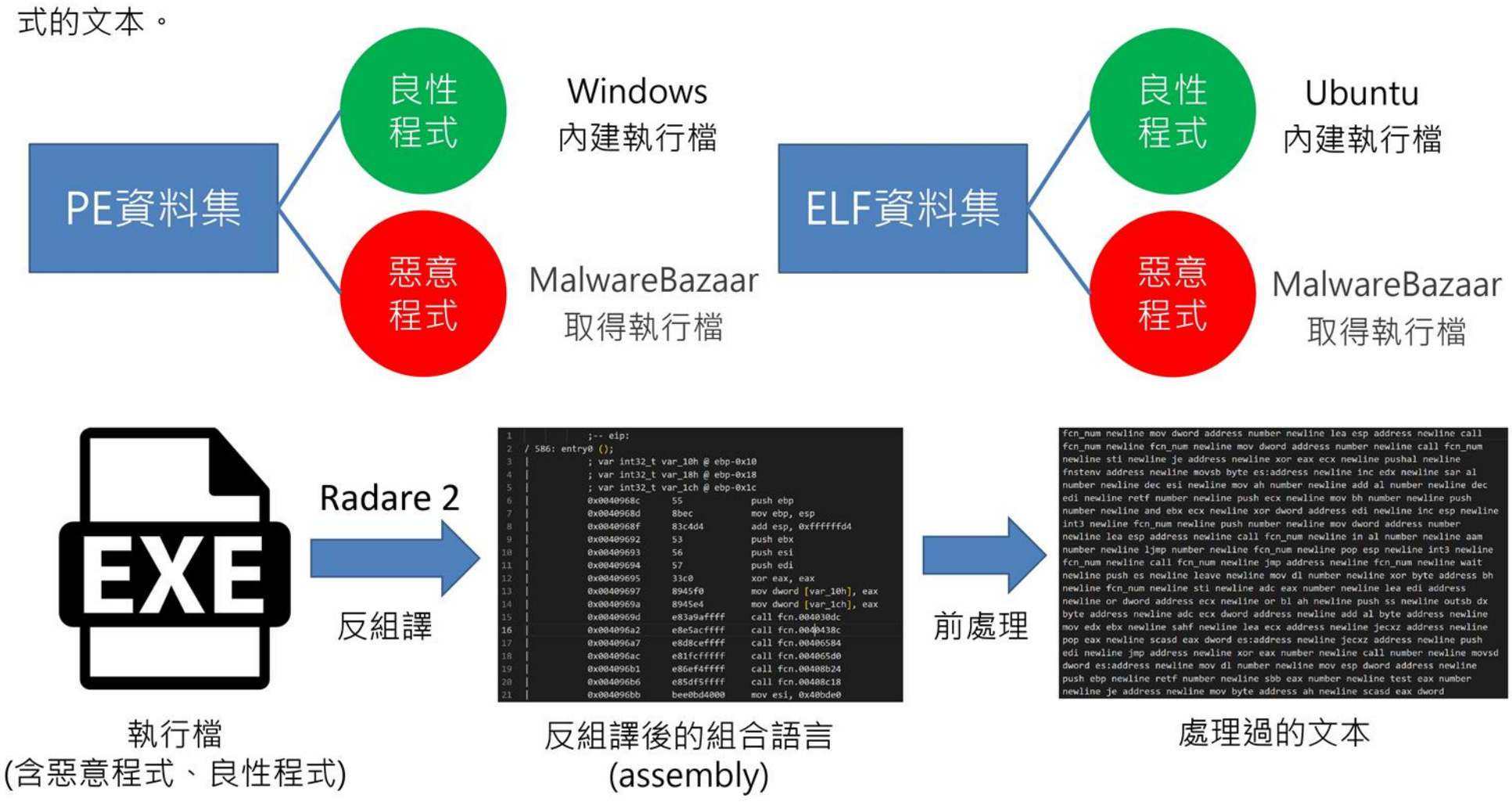
### 一、研究方法

#### (一) 研究架構




#### (二) 資料前處理

使用 Radare2 對執行檔反組譯，並得到該執行檔的組合語言文本，並且對文本進行前處理：排除反組譯失敗的文本，接著將文本內部分資訊去除、編碼，如此便得到後續自然語言處理模型用來辨識惡意程式的文本。




| | 訓練資料數 | | 測試資料數 | | 總計 | |
|---|---|---|---|---|---|---|
| | PE | ELF | PE | ELF | PE | ELF |
| 良性程式 | 2,061 | 461 | 515 | 200 | 2,576 | 661 |
| 惡意程式 | 5,674 | 792 | 515 | 200 | 6,189 | 992 |
| 總計 | 7,735 | 1,253 | 1,030 | 400 | 8,765 | 1,653 |

## (三) 詞袋模型

詞袋法是不考慮文字間序列關係的一種模型設計。

本研究使用 multi-hot 以及 TF-IDF 兩種編碼方式實作詞袋模型。

例：mov ebp esp

{ mov eax ebx
mov ebp esp
pop ebp

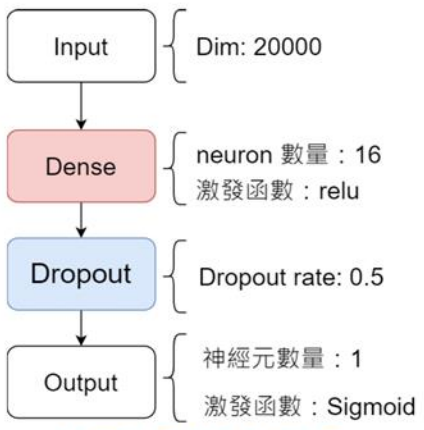
| mov | eax | ebx | ebp | esp | pop | ... |
|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 1 | 0 | ... |

multi-hot 編碼示意圖

$$\mathrm{TF}_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}}$$

$$\mathrm{IDF}_{i,j} = \log \frac{|D|}{|\{j : t_i \in d_j\}|}$$

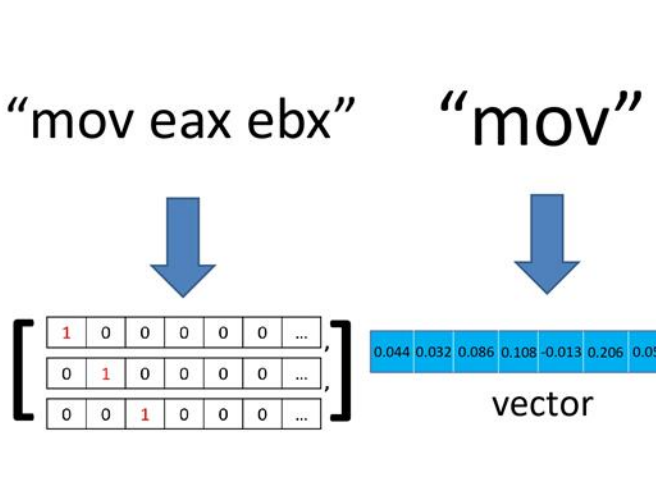
TF、IDF 計算方式

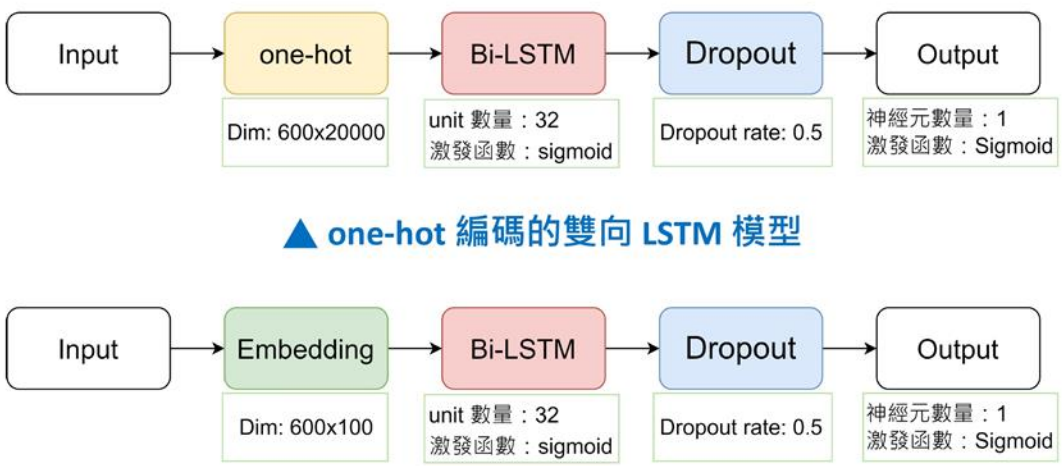



詞袋模型設計架構

## (四) 序列模型

序列模型考慮各個文本中各個詞之間的關係，本研究實作了四種模型，並擷取文本中前 600 個詞作為輸入。



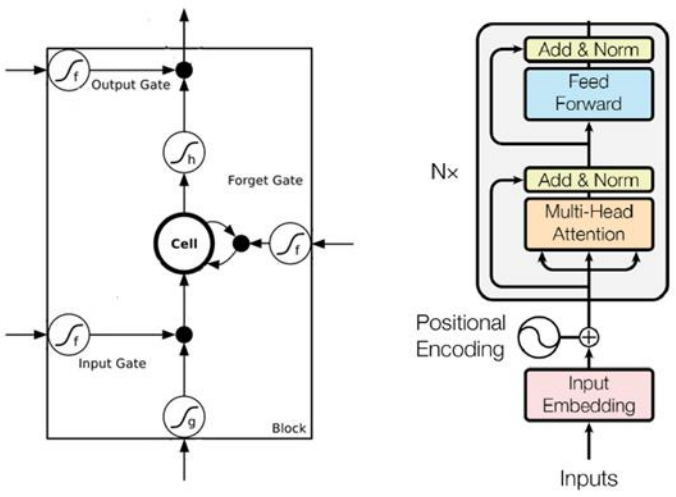




▲ one-hot 編碼的雙向 LSTM 模型

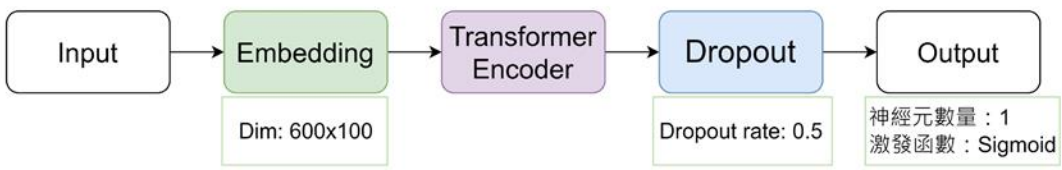



▲ Embedding 編碼的雙向 LSTM 模型

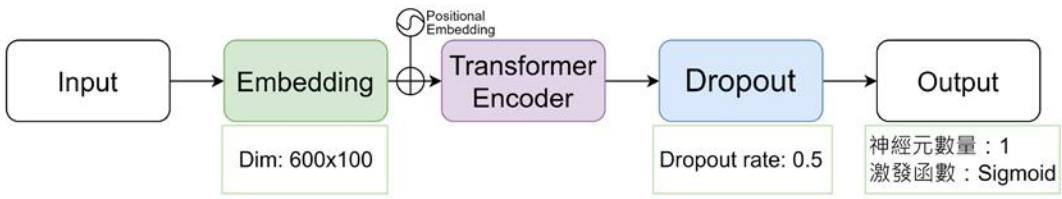
▲one-hot 編碼方式 ▲ Embedding 編碼方式




▲ [1] LSTM memory cell ▲ [7] Transformer Encoder



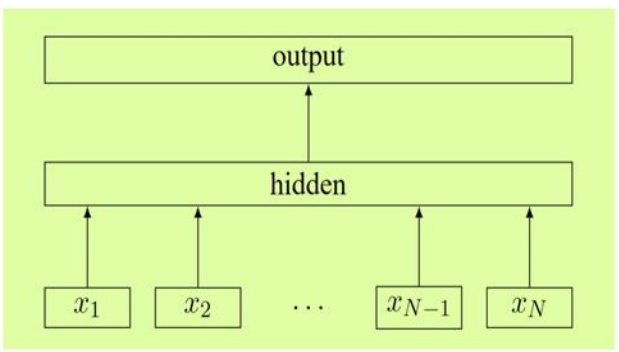

▲ 無 Positional Embedding 的 Transformer Encoder 模型




▲ 有 Positional Embedding 的 Transformer Encoder 模型

## (五) fastText

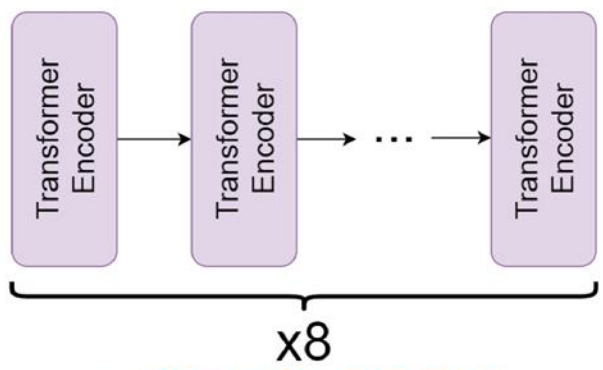
fastText 是一種 NLP 模型，其運算速度較一般模型快。




▲ [2] fastText 模型

## (六) 遷移學習 (BERT)

使用在英語資料集預訓練過的 BERT 進行微調。




▲ 本研究使用之 BERT 示意圖

## (七) n-gram

n-gram 是一種擷取局部順序特徵的方式，

本研究運用在詞袋模型與 fastText 當中。

右圖以 n=2 為例：


## (八) 衡量指標

| | | Actual Value: Positive | Actual Value: Negative |
|---|---|---|---|
| Predicted Value | Positive | True Positive | False Positive |
| | Negative | False Negative | True Negative |

$$\mathrm{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

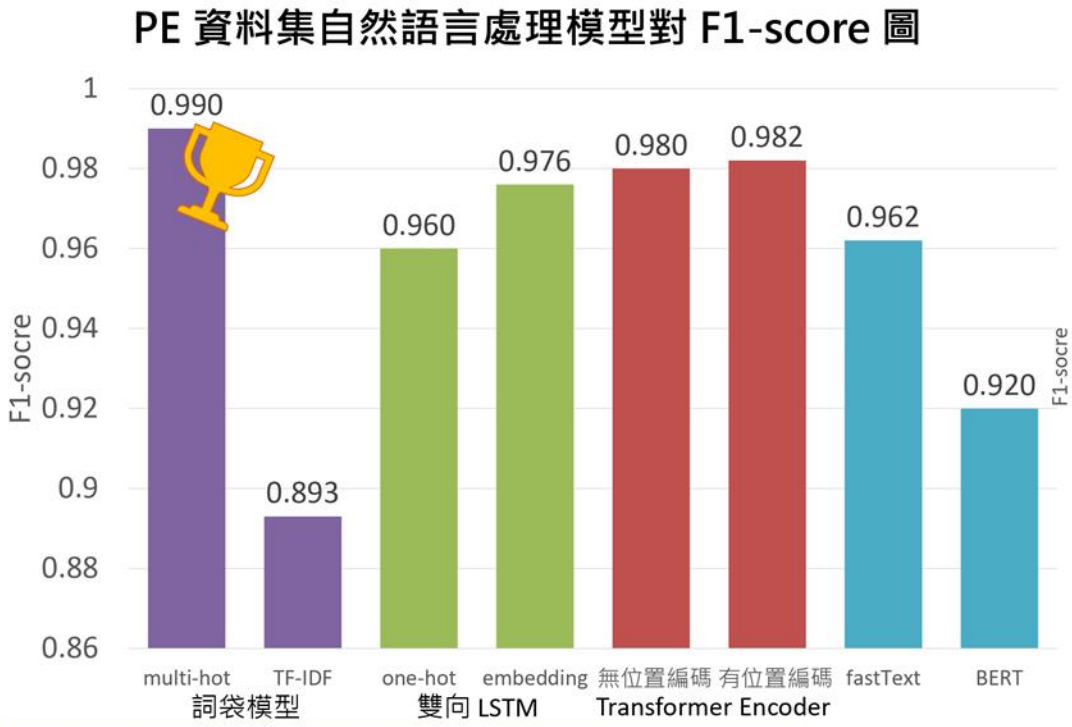
$$\mathrm{Precision} = \frac{TP}{TP + FP}$$

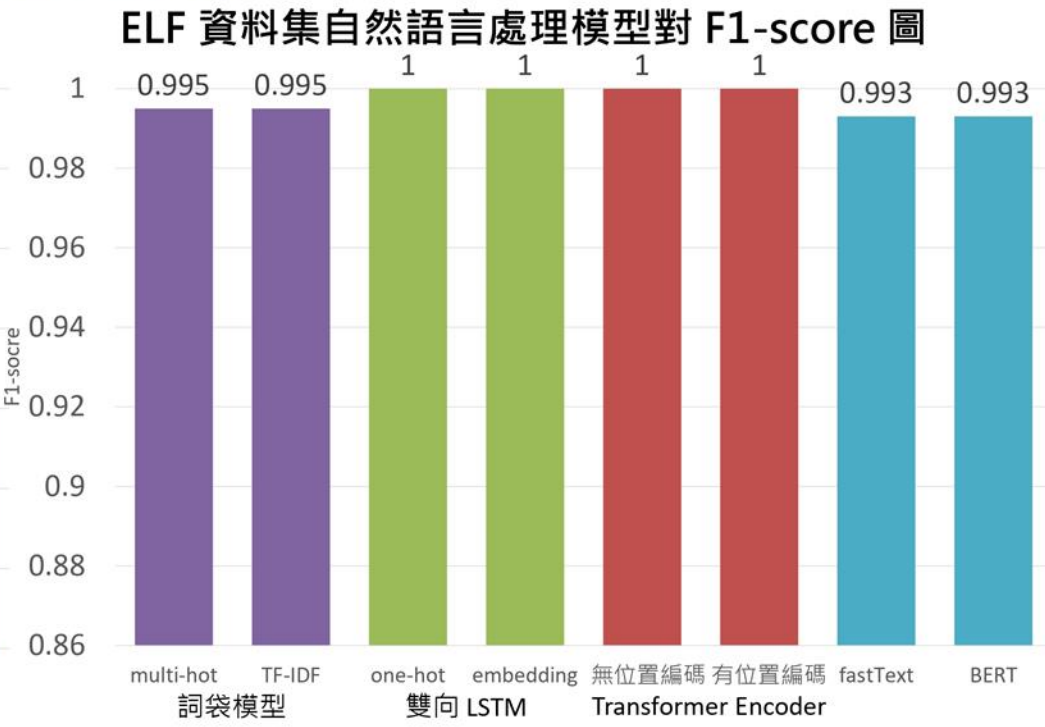
$$\mathrm{Recall} = \frac{TP}{TP + FN}$$

$$\text{F1-score} = \frac{2}{\frac{1}{\mathrm{Precision}} + \frac{1}{\mathrm{Recall}}}$$

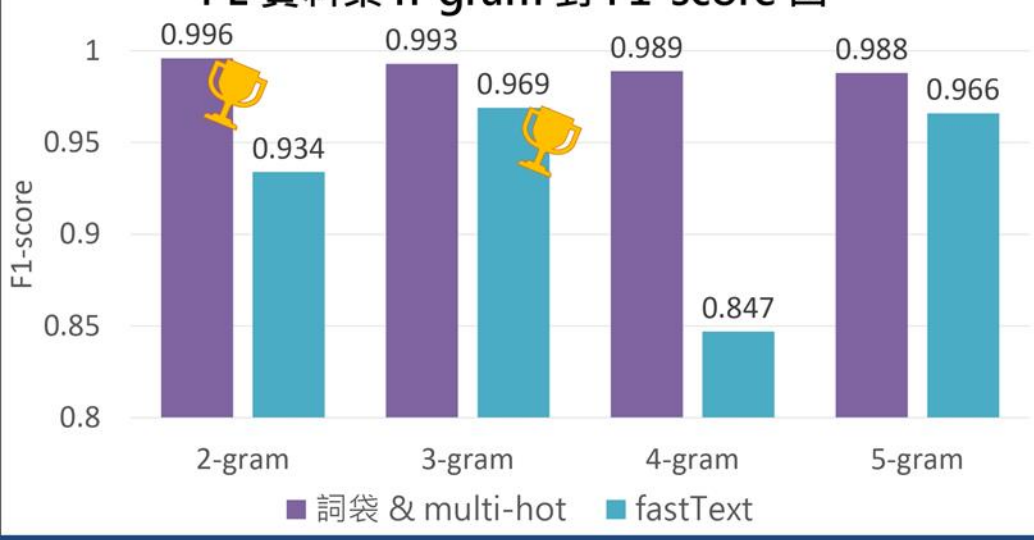
# 研究結果

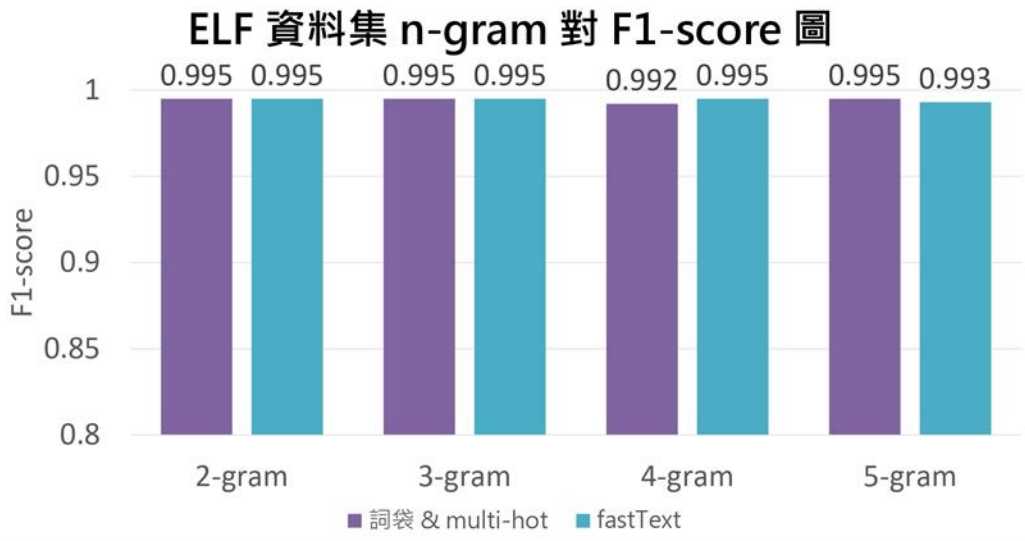
## 一、詞袋模型、序列模型、fastText、BERT 識別惡意程式之 F1-score




## 二、比較不同 n-gram 模型對模型成效影響

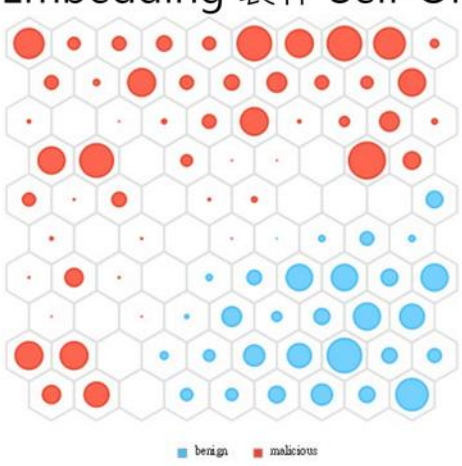



# 討論

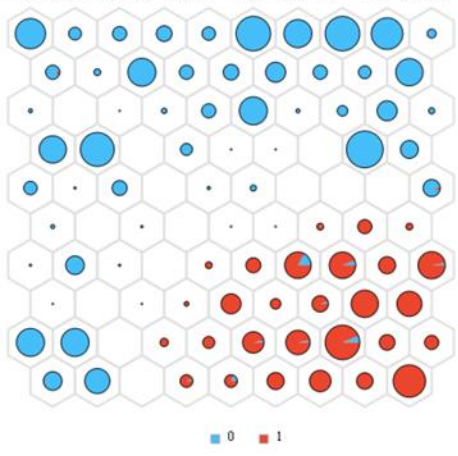
## 一、解釋模型識別惡意程式的依據：與函數呼叫相關

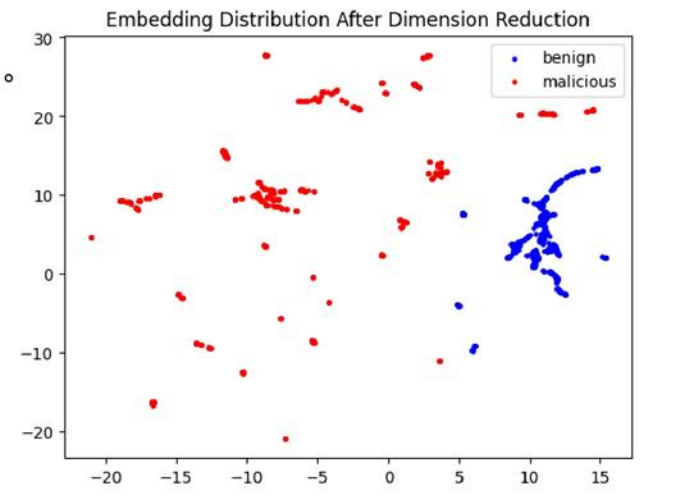
Embedding 製作 Self-Organizing Map，某些函數的分布與原始圖相似。




◀ 使用 label 上色

以 stack_chk_fail ▶ 是否被呼叫來上色。與左圖對照可發現呼叫該函數的多為良性樣本。




▲ Embedding 降維後投影到二維平面

## 二、探討不同的 n-gram 模型對於識別惡意程式成效之影響

(一) 使用multi-hot編碼的詞袋模型使用 2-gram 表現最好；fastText 模型使用 3-gram 表現最好。

(二) 推測原因：組合語言指令一般為 2 或 3 個 token 的特性使 2-gram 跟 3-gram 可以擷取出特徵。

## 三、以圖神經網路實作惡意程式識別模型

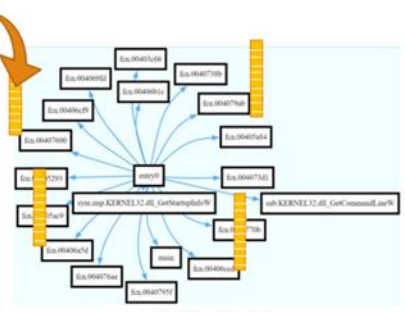
本研究額外以 Graph Convolutional Network 實作惡意程式識別模型，其結果為不作用。

可能原因是模型設計太小，使惡意程式特徵無法被學習。

| 圖神經網路 | 準確率 | F1-score |
|---|---|---|
| GCN | 50% | 66.67% |
| DGCNN | 50% | 66.67% |


該函式的組合語言


詞嵌入向量



函式呼叫圖

# 結論

一、本研究提出的模型，PE 資料集以 2-gram 的 multi-hot 編碼的詞袋模型表現最好，F1-score 為 0.996。

二、F1-score 最高的序列模型為 Transformer encoder 加上位置編碼，兩個資料集分別為 0.982 與 1.000。

三、multi-hot 編碼的詞袋模型在 2-gram 的 F1-score 最高，fastText 則是使用 3-gram 擁有最好的表現。

# 參考資料


1.Hung-yi Lee (2017年1月25日)。ML Lecture 21-1: Recurrent Neural Network (Part I)。YouTube。2023年2月26日，取自：https://youtu.be/xCGidAeyS4M

2.Joulin, A., Grave, E., Bojanowski, P., & Mikolov, T. (2016). Bag of Tricks for Efficient Text Classification. *arXiv preprint arXiv:1607. 01759.*

3.Lu, R. (2019). Malware detection with lstm using opcode language. *arXiv preprint arXiv:1906.04593.*

4.Manning, C.D., Raghavan, P., & Schutze, H. (2008). Scoring, term weighting, and the vector space model. Introduction to Information Retrieval. p. 100.

5.Powers, David M. W. (2011). Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness & Correlation. Journal of Machine Learning Technologies. 2 (1): 37–63.

6.Rahali, A., & Akhloufi, M. A. (2021). MalBERT: Using transformers for cybersecurity and malicious software detection. *arXiv preprint arXiv:2103.03806.*

7.Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems, 30.*

8.Zhang, M., Cui, Z., Neumann, M., & Chen, Y. (2018). An end-to-end deep learning architecture for graph classification. In *Proceedings of the AAAI conference on artificial intelligence* (Vol. 32, No. 1).