Intro. to NLP - HW1 Report

> 學號：0812253
>
> 姓名：陳彥廷
>
> Kaggle username：`yentingchen_\lol/`

報告需求

```
1. Describing your methods in detail.(50%)
2. Is there any difference between your expectations and the results? Why?(20%)
3. What difficulties did you encounter in this assignment? How did you solve it?(30%)
```

目錄

演算法解釋

概要

首先利用 POS tag + dependency parsing 的結果來找出所有可能的 verb，再根據找到的 verb 去找每一種 verb 對應到的主詞以及受詞。最後因為題目要求的主詞、動詞與受詞沒有要求要在同一句裡面，因此將主詞、動詞與受詞的結果分別儲存成三個獨立的 list。

最後對於每一筆 test case，只要 test case 的主詞、受詞與動詞「包含於任何一個輸出結果」，就 輸出 1，都不包含的話就輸出 0。舉個例子，`willing to kill` 包含於一筆輸出結果 `is willing to kill`，如此便會輸出 1，反過來會輸出 0。

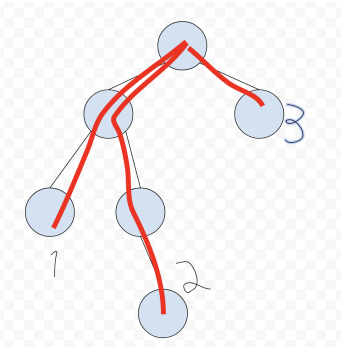
詳細解析如何找出動詞

首先，對於每一個 POS tag 是 `VERB` 的單字，會基於 dependency parsing 的 tree 結構實作一個 DFS。這個 DFS 的需求比較特殊，並不會 traverse 整個 tree，而是只會允許從特定的 dependency relation（比方說是 `xcomp`）向下查找。

另外，查找完成之後，我們儲存的資料結構會是一個 2D list of index，如下：

```
[[0, 1, 2, ...], [3, 4, ...] ...]
```

每一個 list 單元儲存的是一組「從 root 到最遠的 leaf 的 path」，請見下方示意圖：



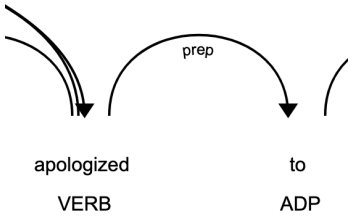
接著以下開始講解程式實作。首先 `verb_longest_path_list` 負責存儲上方的這種 2D list of index。

`verb_relation` 則是用於記錄可以向下查找的 relation，這些是透過觀察資料集得來的，因為動詞可能包含時態、被動態、動詞片語等等等...，下方有列出一些觀察。

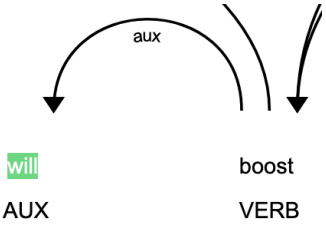
其中 `nsubj` 用於一些動詞會包含一點點主詞的特殊子句。

```
# Start from src line 147
verb_longest_path_list = []
verb_relation = ['aux', 'xcomp', 'prep', 'prt', 'advmod', 'neg', 'auxpass', 'nsubj', 'attr']
```

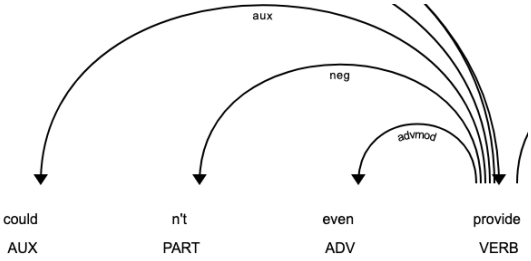
- 一種動詞片語



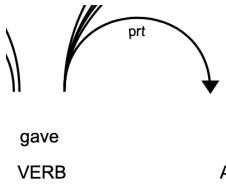
- will 等等的時態



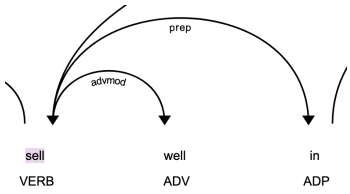
- 否定的 relation `neg`



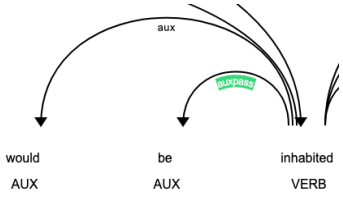
- 另外一種動詞片語



- 這裡的 `sell well in` 是一組動詞片語，中間的 ADV 可以無限延長



- 前面有 would be 之類的東西，所以 `auxpass` 也是考慮的 relation



定義完成之後，撰寫 recursive function。branch 用來記錄從 root 出發到最遠的 leaf 的 path，並且 `is_edge` 用來判斷是否到最遠的 leaf，如果到了才會把這隻 branch 記錄在剛剛定義的 global variable。

```
# Start from src line 150
def get_verb_index(token, branch, is_edge):
    branch.append(token.i)

    for child in token.children:
        if child.dep_ in verb_relation:
            is_edge = False
            get_verb_index(child, branch.copy(), True)

    if is_edge:
        global verb_longest_path_list
        branch.sort()
        verb_longest_path_list.append(branch)

    return
```

接下來，定義下方函數回傳 global `verb_longest_path_list` 的結果，並且將其重設為空陣列。

```
# Start from src line 165
def copy_and_reset():
    global verb_longest_path_list
    if len(verb_longest_path_list) == 0:
        return []

    l = verb_longest_path_list.copy()
    verb_longest_path_list = []
    return l
```

在取得 predict 結果的主函數中，對於 POS tag 是 `VERB` 或是特殊情況的 `AUX` 我們都納入找動詞的考量。

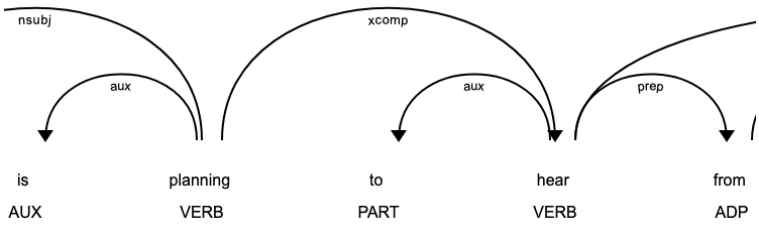
```
# Start from src line 199
def get_predict(doc):
    # 這邊定義三個到時候會輸出的所有可能 V S O
    verbs = []
    subjects = []
    objects = []

    for token in doc:
        children_deps = list(token.dep_ for token in token.children)

        # 對於 VERB 以及特殊情況的 AUX 都進行查找
        if (token.pos_ == 'VERB') or (token.pos_ == 'AUX' and 'relcl' == token.dep_) or (token.pos_ == 'AUX' and 'attr' in
 children_deps):
            # find verb
            get_verb_index(token, [], True)
            verb_2d_arr = copy_and_reset()

            for verb_list in verb_2d_arr:
                verbs.append(concat(doc, verb_list))
```

但是只考慮這種 longest path 組成的動詞會忽略掉一些情況，觀察下方這個例子：



在這個情況下 `is planning to hear from` 是一組合格的動詞，但是基於上面的 longest path 算法的話很顯然無法同時考量到 `is planning` 跟 `planning to hear from` ，這兩組會被記錄在不同的 case。

所以我構建了 flatten 函數，目標是不重複地將剛剛的 longest path 的所有 index 攤平成一維陣列，並且串起來包含進去考量的 case。

```
# 這個函數用來攤平剛剛的 2d array
def flatten_arr(arr_2d):
    f = []
    for l in arr_2d:
        for idx in l:
            if idx not in f:
                f.append(idx)
    f.sort()
    return f

# 這個函數用來把攤平的結果串成 string
def concat(doc, list_of_index):
    return ' '.join([doc[x].text for x in list_of_index])
```

在主函數中，將這筆操作的結果添加進去 verbs 陣列。

```
verb_1d_arr = flatten_arr(verb_2d_arr.copy())
verbs.append(concat(doc, verb_1d_arr))
```

詳細解析如何找出主詞

每一次找尋主詞都是基於一個動詞開始找，因此在剛剛的主函數內 `get_predict` 找完動詞之後會接續開始找主詞。

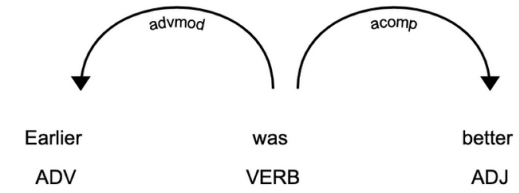
找主詞的方法比較單純，基本上是透過判斷 `dep_` ，並且確定符合之後就把「之下的整個 subtree」加入進來。以下將會分析主詞會出現的幾種 `dep_` 情況。

1. 首先是像是 `nsubj` 之類的 dep 都會出現主詞，因此這裡利用 `in` 來做字串判斷。and 後面的部分是判斷是不是屬於現在討論的動詞。

   ```
   if 'subj' in token.dep_ and token.head.i == verb_token.i:
       subtree = list(token.subtree)
       start = subtree[0].i
       end = subtree[-1].i + 1
       subjects.append(doc[start:end].text)
   ```

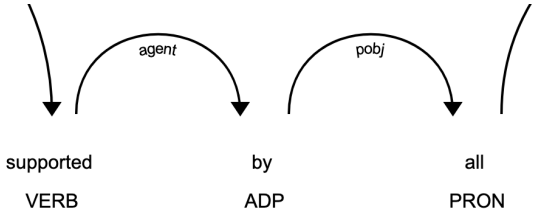
2. verb 給予的 `advmod` 也會是主詞

- ADVMOD(was, Earlier)



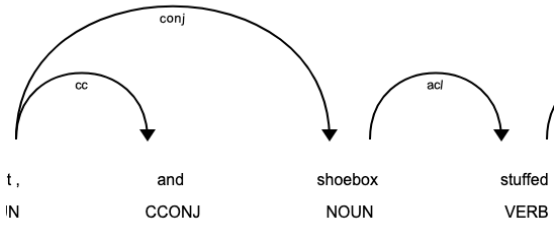

```
if 'advmod' == token.dep_ and token.head.i == verb_token.i:
    subtree = list(token.subtree)
    start = subtree[0].i
    end = subtree[-1].i + 1
    subjects.append(doc[start:end].text)
```

3. 被動態的動詞，主詞的位置跟關係會像下圖一樣，因此也要加上特殊判斷



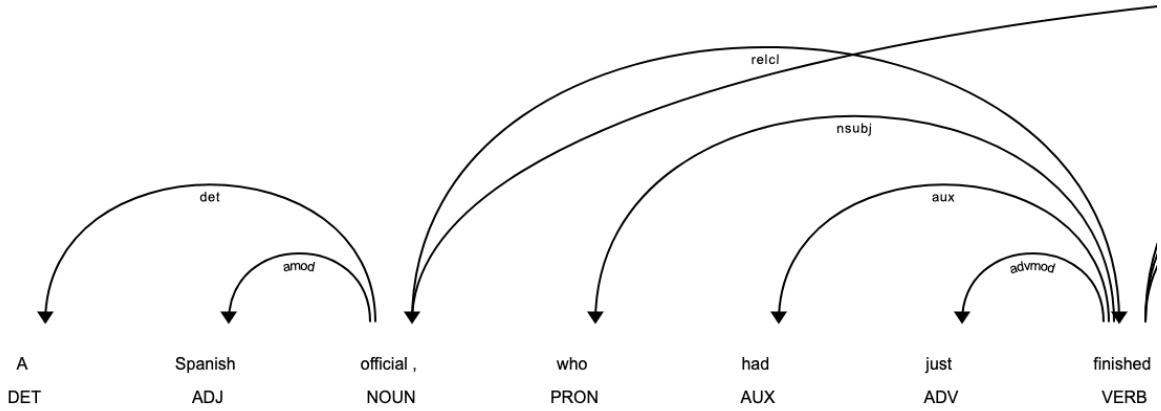

```
if 'pobj' == token.dep_ and token.head.pos_ == 'ADP' and token.head.head.i == verb_token.i and 'agent' == token.head.de
p_:
    subtree = list(token.subtree)
    start = subtree[0].i
    end = subtree[-1].i + 1
    subjects.append(doc[start:end].text)
```

4. 連接詞後的子句要特殊考慮，因為連接詞斷句的關係有時候 dependency 會變得很特殊。下面範例中的主詞是 shoebox 很明顯就必須將 `acl` 納入判斷。




```
if 'acl' in verb_token.dep_:
    subtree = list(verb_token.head.subtree)
    start = subtree[0].i
    end = subtree[-1].i + 1
    subjects.append(doc[start:end].text)
```

5. 關係子句的主詞可能會指定到關西代名詞上，但其實我們需要的是真正的主詞。如下，我們需要 `A Spanish Offcial` 而不是 `who` 。




```
if 'relcl' in verb_token.dep_:
    subtree = list(verb_token.head.subtree)
    start = subtree[0].i
    end = subtree[-1].i + 1
    subjects.append(doc[start:end].text)
```

以上是對於找主詞函數 `get_subject` 內部的說明。在主要函數內 `get_predict` 只要簡單地將結果添加進去 subject list 就可以了。

```
for s in get_subject(doc, token):
    subjects.append(s)
```

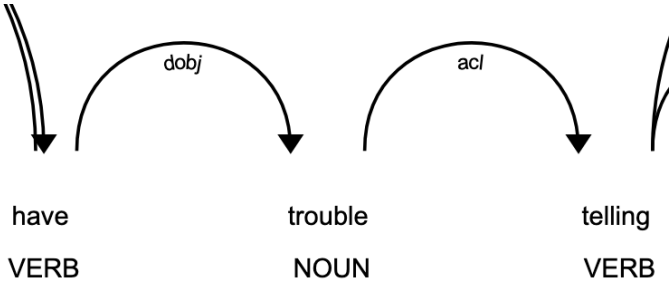
詳細解析如何找出受詞

受詞判斷跟主詞很類似，尋找特別的 dep_ 、處理特殊情況之後把整個符合的 subtree 都添加進去。以下一樣針對各種特殊處理的情況進行說明。

1. 和主詞的判斷很類似，一班情況只要利用 `obj in dep_` 判斷就可以找出的大部份的受詞。

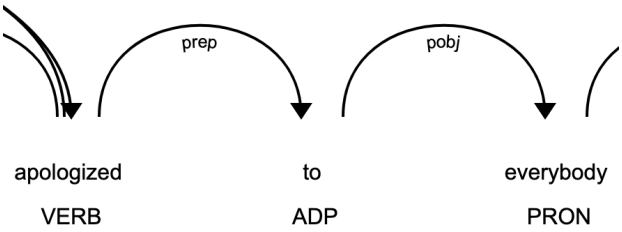
```
if 'obj' in token.dep_ and token.head.i == verb_token.i:
            # special case : "have trouble telling" -> trouble is the object
            is_not_object = False
            for child in token.children:
                if 'acl' in child.dep_ and child.pos_ == 'VERB':
                    is_not_object = True
                    break
            if is_not_object:
                continue
            subtree = list(token.subtree)
            start = subtree[0].i
            end = subtree[-1].i + 1
            objects.append(doc[start:end].text)
```

這裡有一個特殊判斷是處理下方截圖的特殊情況：



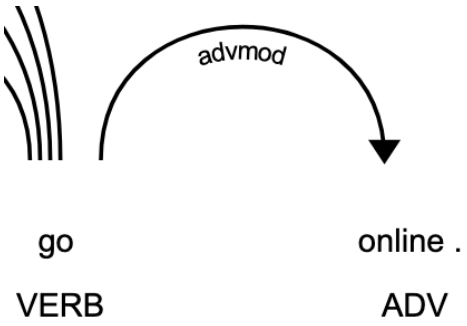
這裡的 `trouble` 並不是一種受詞，而是動詞片語的一部分。

2. 動詞片語會造成雖然有 `pobj` 這種 dep 但是因為 head 不是 VERB 所以無法納入受詞的情形。因此這邊針對 ADP tag 進行特殊判斷。



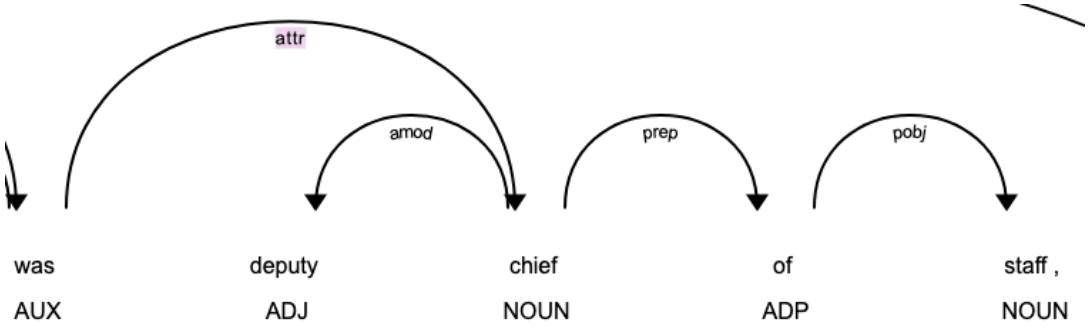

```
if 'pobj' == token.dep_ and token.head.pos_ == 'ADP' and token.head.head.i == verb_token.i and 'prep' == token.head.dep_:
    subtree = list(token.subtree)
    start = subtree[0].i
    end = subtree[-1].i + 1
    objects.append(doc[start:end].text)
```

3. 下面是副詞也有可能是一種受詞的特殊情況，這種用法非常少見因此也做特殊判斷。




```
if token.pos_ == 'ADV' and 'advmod' == token.dep_ and token.head.i == verb_token.i:
    subtree = list(token.subtree)
    start = subtree[0].i
    end = subtree[-1].i + 1
    objects.append(doc[start:end].text)
```

4. 下面這個特殊的例子，受詞是 `deputy chief of staff` ，但是 chief 的 dep 卻是 `attr` 。這邊也利用特殊判斷處理。




```
if 'attr' in token.dep_ and token.head.i == verb_token.i:
    for child in token.head.children:
        if 'nsubj' in child.dep_:
            subtree = list(token.subtree)
            start = subtree[0].i
            end = subtree[-1].i + 1
            objects.append(doc[start:end].text)
```

以上是對於找主詞函數 `get_oject` 內部的說明。在主要函數內 `get_predict` 只要簡單地將結果添加進去 object list 就可以了。

```
for o in get_object(doc, token):
    objects.append(o)
```

期望與結果的差異

這次作業主要是透過 `example_and_answer.csv` 在本地做測試玩得結果在切換成正式的 `dataset.csv` 做上傳。

一開始的做法只是單純地利用 tag 來判斷，並且透過 dep 來對於每一種特殊情況做判斷，有曾經將本地的 accuracy 壓到 96%。但是上傳結果卻只有 60%，很明顯是 overfit。

之後發現了其實透過 tree traversal 的算法可以考量更一般性的情況，也証實了一開始只利用 POS tag 跟 dep 的做法並沒有發揮 dependency parsing 的正確邏輯。

最後又發現其實 verb, subject, object 彼此之間是有關連的，往這個方向 + tree traversal 做出了本地只有 90% accuracy 的 model，但是在理論上我自己是覺得比較合理並且比較符合觀念的。

上傳了解答之後也得到了更好的成績，讓我理解到觀念遠大於片面實作的中要性。

因為中間的實驗過程比較繁複無法紀錄，這裡提供簡單紀錄 wrong case index 的實驗過程截圖。這是透過觀察判斷錯誤的 case 來持續改善 Model 的過程。

Local Acc: 0.62 Online Acc: 0.55

Local wrong:

怪外的名單：

❓ 3 13 22

[3, 4, 6, 11, 12, 16, 18, 24, 26, 28, 29, 30, 32, 35, 36, 42, 46, 48, 49]

[3, 6, 11, 12, 16, 18, 24, 26, 28, 29, 30, 32, 35, 36, 42, 46, 48, 49]

[3, 11, 12, 16, 18, 24, 28, 29, 30, 32, 35, 36, 41, 42, 46, 48, 49]

[3, 12, 16, 18, 24, 28, 29, 30, 32, 35, 36, 41, 42, 46, 48, 49]

[3, 13, 18, 24, 28, 29, 30, 32, 35, 36, 41, 42, 46, 49]

[3, 13, 22, 26, 28, 29, 30, 32, 35, 36, 41, 42, 49]

[3, 13, 14, 20, 22, 41, 42, 49]

[3, 13, 14, 20, 22, 41, 46, 49]

[3, 13, 14, 20, 22, 41, 43, 49]

[3, 13, 14, 22, 35, 36, 41, 43, 49]

[3, 13, 14, 20, 22, 41, 43, 49]

[3, 4, 12, 22, 32, 35, 36, 42, 43, 44, 49]

[3, 12, 22, 32, 35, 36, 42, 43, 49]

[3, 22, 26, 28, 29, 30, 42, 49]

[22, 42, 49]

[13, 14, 22, 41, 49]

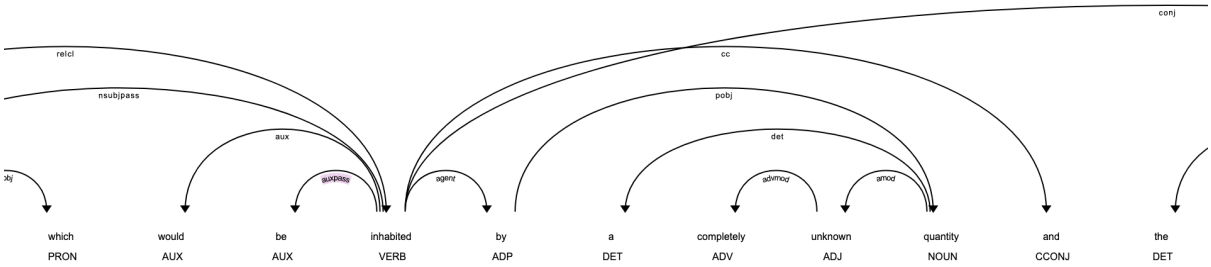
遇到的困難與解決方式

這次的作業需要大量的觀察資料集，因此資料的可視化我覺得是非常重要的部分。大部分遇到的困難都是透過自己寫的可視化函數工具來找到解決方法的。其實了解觀念之後遇到的困難真的很少，所以這邊我簡單 demo 一下 wrong case 的呈現方式。

大部分跑完單次測試如果有錯誤的情況會如下呈現：

```
======
Accuracy:  0.9
Correct:  45
Wrong Indexes: [13, 14, 22, 41, 49]
======
Row index:  13
Output:  1
Expected Verb: " 's adding "
Expected Subject: " Dell Computer "
Expected Object: " 25 million shares "
Predicted patches:
Verb:  ['Computer said', 'Computer said', 'it adding', "'s adding", 'adding to', 'adding for', "it 's adding to for"]
Subject:  ['Dell Computer', 'it']
Object:  ['25 million shares', 'its stock repurchase program', 'a total of 125 million shares , almost a third of its 383 million outstanding as of last quarter']
======
Row index:  14
Output:  1
Expected Verb: " was "
Expected Subject: " Sabri "
Expected Object: " a first step "
Predicted patches:
Verb:  ['he left', 'he left', 'Sabri said', 'Sabri said', 'he hoped', 'he hoped', 'announcement was', 'was step toward', 'announcement was step toward']
Subject:  ['he', 'Sabri', 'he', "Iraq 's announcement"]
Object:  ['the UN building', "a first step toward the `` lifting of this brutal regime of sanctions '' against his country"]
======
Row index:  22
Output:  1
Expected Verb: " has increased "
Expected Subject: " the department "
Expected Object: " the number of cattle "
Predicted patches:
Verb:  ['admitting', 'admitting', 'Without increased', 'department increased', 'has increased', 'rapidly increased', 'Without department has rapidly increased', 'it tests'
, 'tests annually', 'it tests annually']
Subject:  ['the department', 'rapidly', 'the number of cattle it tests annually', 'it', 'annually']
Object:  ['any error', 'rapidly', 'the number of cattle it tests annually', 'annually']
======
Row index:  41
Output:  1
Expected Verb: " huddled "
Expected Subject: " Euromoney 's executives "
Expected Object: " managing editor of Institutional Investor 's Americas edition "
Predicted patches:
Verb:  ['executives spent', 'spent huddled in', 'executives spent huddled in', 'huddled in', 'huddled in', 'they named', 'they named', 'to replace', 'to replace']
Subject:  ["Euromoney 's executives", "Robert Teitelman , managing editor of Institutional Investor 's Americas edition , whom they named editor to replace Cudaback", 'the
y']
Object:  ["negotiations with Robert Teitelman , managing editor of Institutional Investor 's Americas edition , whom they named editor to replace Cudaback", 'whom', 'Cudab
ack']
======
Row index:  49
Output:  0
Expected Verb: " could n't even provide lift for "
```

針對每一筆錯誤的 index 再利用 spacy 提供的 display 工具觀察 dependency 關係。



以上是我解決大部分問題的方式。

🙏 感謝助教花時間批改與閱讀 🙏