人工智慧導論 HW4 報告

資訊 113 F74094017 李昆翰

一、了解資料：

在本次作業中，我觀察訓練的資料的方式共有以下幾個步驟：

- 首先，我直接用 .head 和 .info 這兩個 pandas DataFrame 資料特性，來先了解 dataset 的資料內容。

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

train = pd.read_csv("new_data/train.csv")

# Display the first few rows of the dataset
print(train.head())
# Get information about the dataset
print(train.info())
```

圖一、資料匯入和結構顯示程式碼

先來看 train.head 的結果：

```
   profile pic  nums/length username  fullname words  nums/length fullname  \
0            1                  0.27               0                   0.0
1            1                  0.00               2                   0.0
2            1                  0.10               2                   0.0
3            1                  0.00               1                   0.0
4            1                  0.00               2                   0.0

   name==username  description length  external URL  private  #posts  \
0               0                  53             0        0      32
1               0                  44             0        0     286
2               0                   0             0        1      13
3               0                  82             0        0     679
4               0                   0             0        1       6

   #followers  #follows  fake
0        1000       955     0
1        2740       533     0
2         159        98     0
3         414       651     0
4         151       126     0
```

圖二、train.head 的結果

從以上結果可以大概估略每一個 column 的資料型態和名稱。

然後就是 train.info 的結果：

```
Data columns (total 12 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
...
 11  fake                  576 non-null    int64
dtypes: float64(2), int64(10)
memory usage: 54.1 KB
None
```

圖三、train.info 的結果

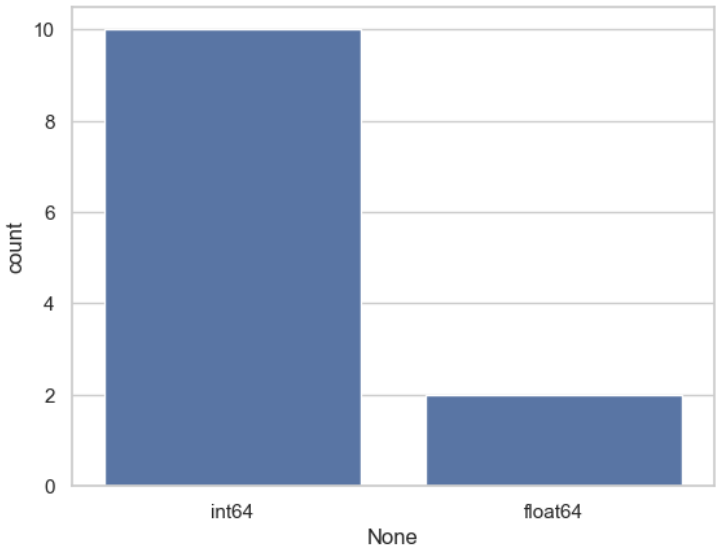
從 train.info 的結果中，可以知道 column 的數量、資料型態占比、以及此 dataset 所用的空間。

若要將 dataset 裡各個資料的資料型態分布視覺化，可以用以下的程式碼實現。

```
# Plot the distribution of data types
sns.set(style="whitegrid")
sns.countplot(x=train.dtypes)
plt.show()
```

圖四、資料型態分布視覺化程式碼

得到的結果圖表為：



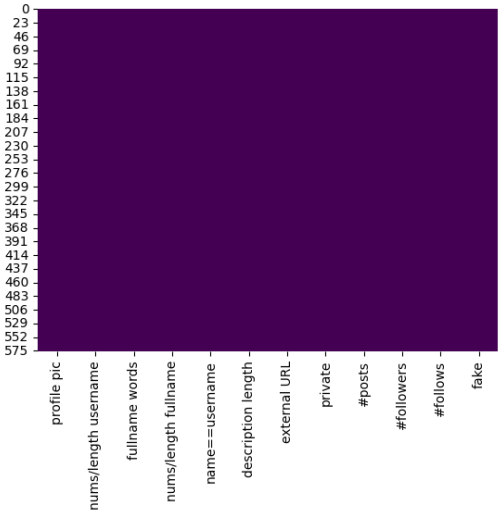
圖五、資料型態分布圖

- 接下來，使用 seaborn 的 heatmap 顯示 dataset 有沒有空的值。

```
# Plot missing values using Seaborn
sns.heatmap(train.isnull(), cbar=False, cmap='viridis')
plt.show()
```

圖六、檢查 missing data 的程式碼

以下為結果圖：



圖七、real data heap map

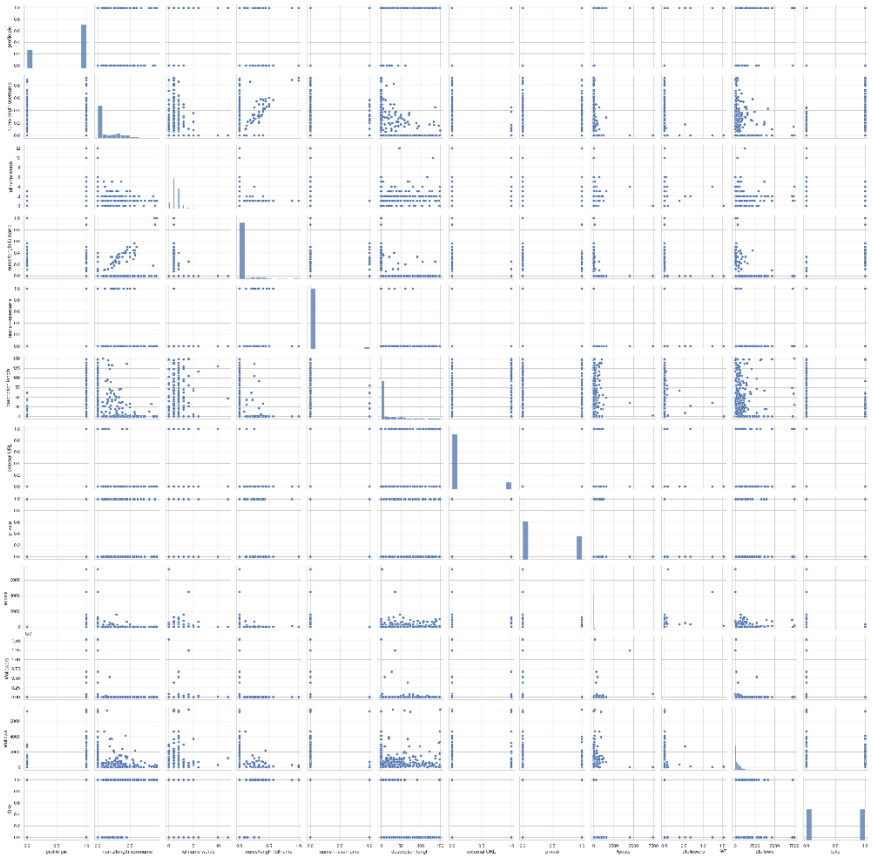
此圖顯示出此 dataset 沒有 missing value，因此不用做相關的前處裡。

- 再來，使用 seaborn 的 pairplot 配合 pandas 的 select_type 來製造多圖表的結構，顯示此 dataset 若有 numeric features 的情況下，它們之間的相關性為何。

```
# Visualize numeric features
sns.pairplot(train.select_dtypes(include='number'))
plt.show()
```

圖八、顯示 numeric feature 的程式碼

得到的結果圖為以下的圖表：



圖九、numeric feature pairplot

圖表 1（由於圖太大的關係，所以有另存一個版本在作業資料夾 figure 中）

在這組 12*12 的圖組中，除了顯示出每個 numeric feature 的資料分佈外，也側面的表示了各個資料之間的相關性。例如說 nums/length username 的資料和 nums/length fulllname 的資料呈正相關。

- 當然，除了檢查數值資料，也要檢查有無 Categorical Feature 的存在。這裡我用 seaborn 的 set 以及 countplot 去完成。

```
# Visualize categorical features
sns.set(style="whitegrid")
for column in train.select_dtypes(include='object').columns:
    plt.figure(figsize=(12, 6))
    sns.countplot(x=column, data=train)
    plt.show()
```

圖十、顯示 categorical feature 的程式碼

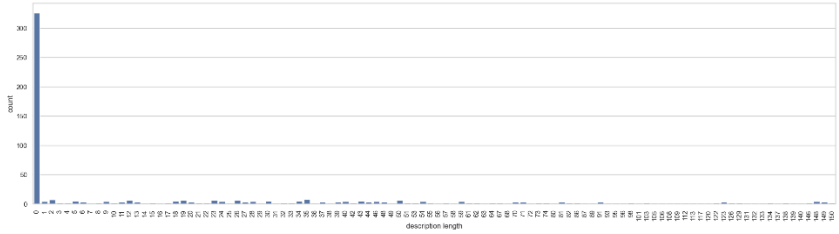
在這支程式中，沒有對應的結果圖生出來，顯示說這組 dataset 無 Categorical Feature，不須考慮相關的前處理方式。

- 最後，我一樣使用了 seaborn 的 set 以及 countplot 去印出所有 column 的資料分布。

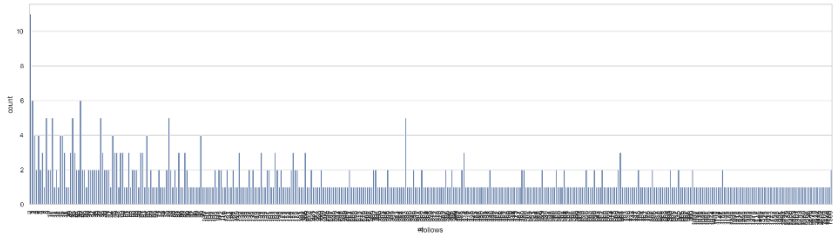
```python
# Visualize bias in the data
sns.set(style="whitegrid")
for column in train.columns:
    plt.figure(figsize=(24, 6))
    sns.countplot(x=column, data=train)
    plt.xticks(rotation=90)
    plt.show()
```

圖十一、視覺化資料集中各行資料的 bias 的程式碼

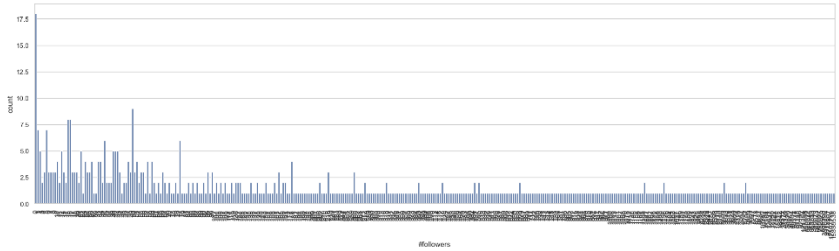
結果為下圖組：



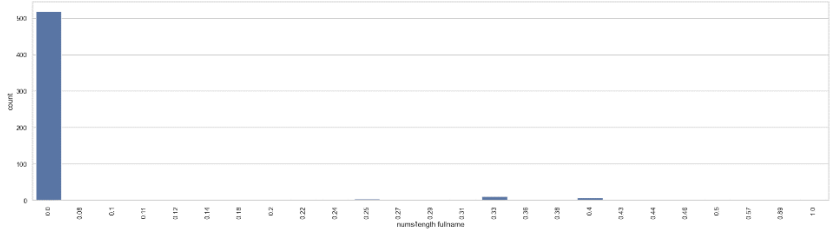
圖十二、description length



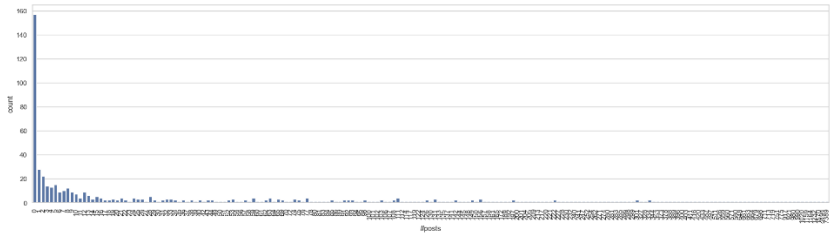
圖十三、#follows



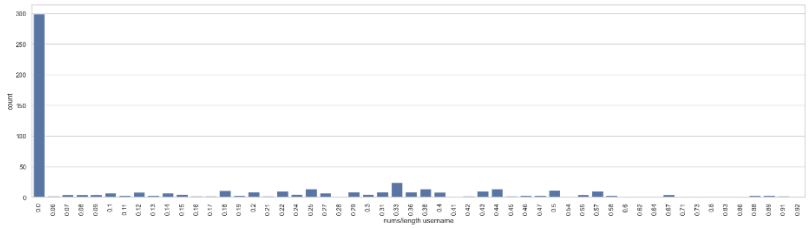
圖十四、#followers



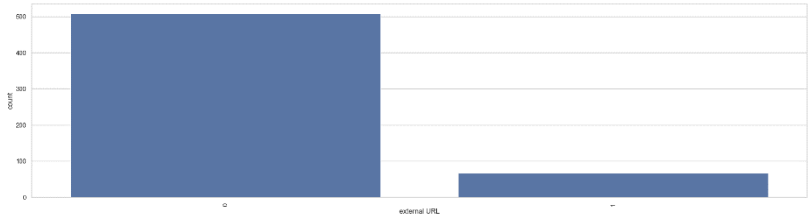
圖十五、nums/length fullname



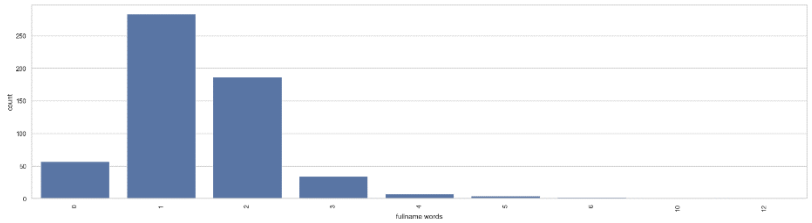
圖十六、#posts



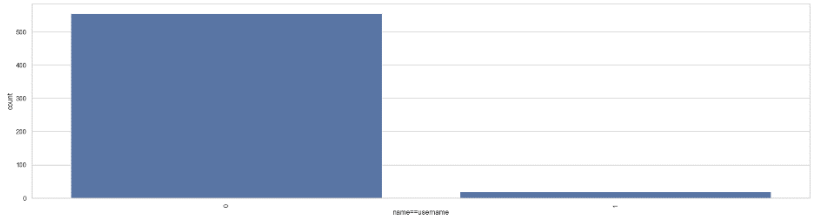
圖十七、nums/length username



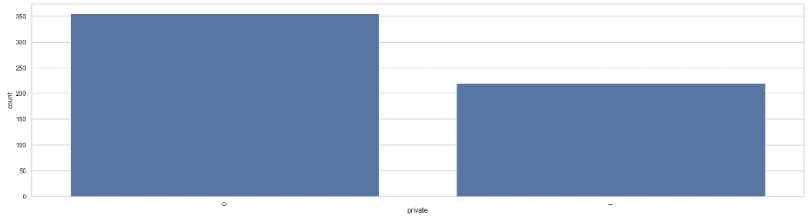
圖十八、external URL



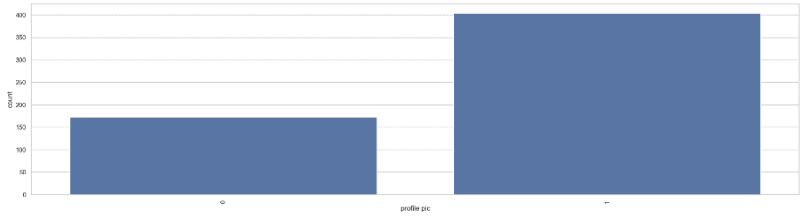
圖十九、fullname words



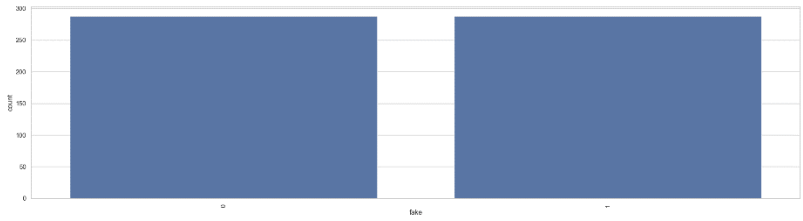
圖二十、name==username



圖二十一、private



圖二十二、profile pic



圖二十三、fake

從結果圖組中可以發現到，description length、#follow、#followers、#post、nums/length username、以及 nums/length fullname，都是呈現類似連續性的分佈。而其他的資料則是二元性的分布，也就是離散型的資料分佈。

除此之外，也可以發現到，被設定為 class 的 column – `fake` 的資料分佈是呈現 5:5 的樣態。

二、**前處理**：

根據我們上面的資料分析，我認為這個 dataset 的問題是部分資料過於連續，可能會增加連接 root 的 branches 的數量。因此我只做將 dataset 的連續資料變成離散資料的前處理。

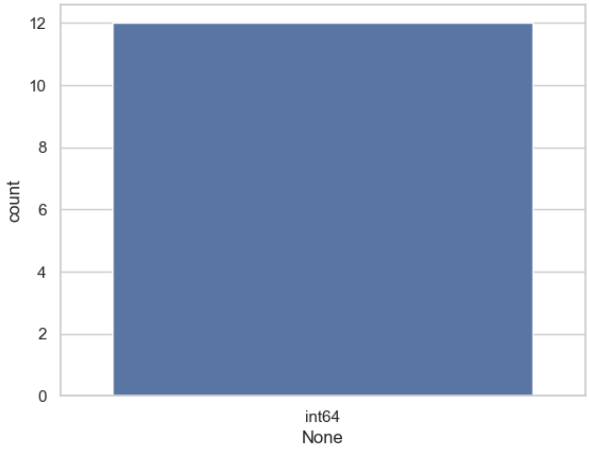
我進行離散化的前處理工作主要是用 pandas 的 cut()，將 cut() 的變數 bins 設定為整數，以進行 equal-weight binning。實作程式為以下：

```
# Equal-width binning
num_bins = [3, 10, 4, 2, 2, 2]
discrete_column = ['description length', '#follows', '#followers', '#posts', 'nums/length username', 'nums/length fullname']
i = 0
for c in discrete_column:
    train[c] = pd.cut(train[c], bins=num_bins[i], labels=False)
    test[c] = pd.cut(test[c], bins=num_bins[i], labels=False)
    i += 1
```

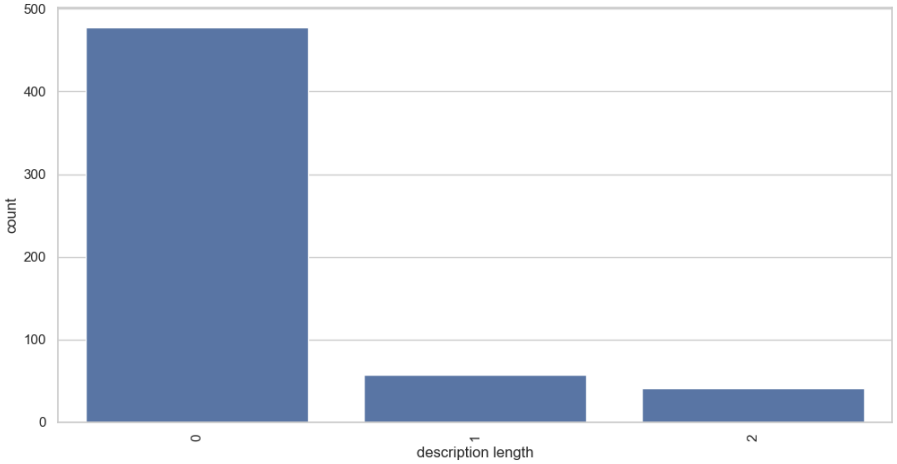
圖二十四、離散化連續資料

在實作程式中，我將 description length、#follow、#followers、#post、nums/length username、以及 nums/length fullname 的資料全部使用 equal-weight binning 的方式離散化，分配用的 num_bins 是在經過幾次調整後的結果。

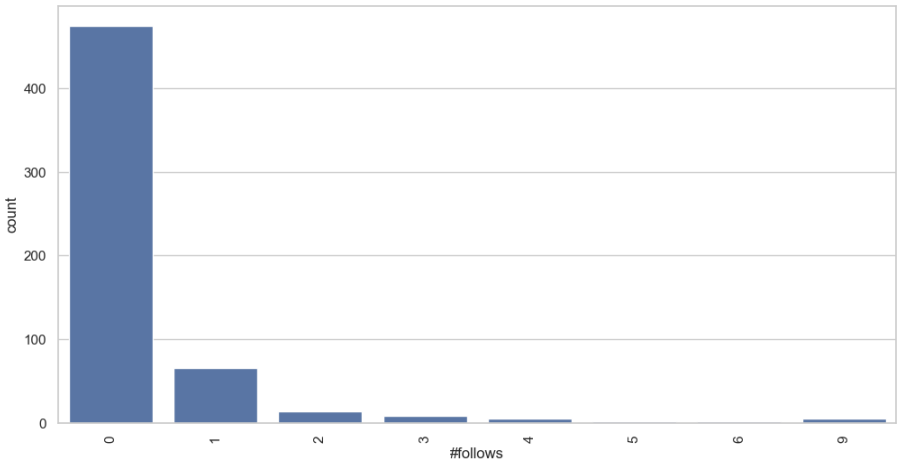
在經過前處理後，經過離散化的資料分布，以及整個 train 的 data type 將變為以下的結果：



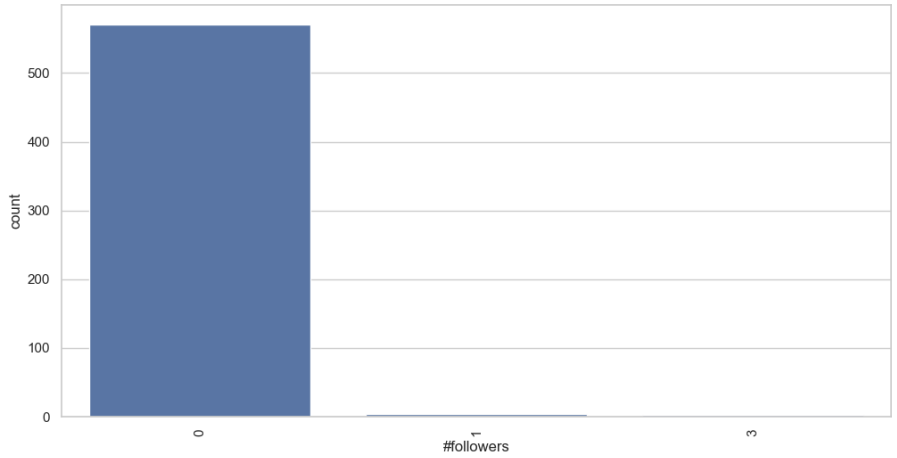
圖二十五、train 的 data type 分佈



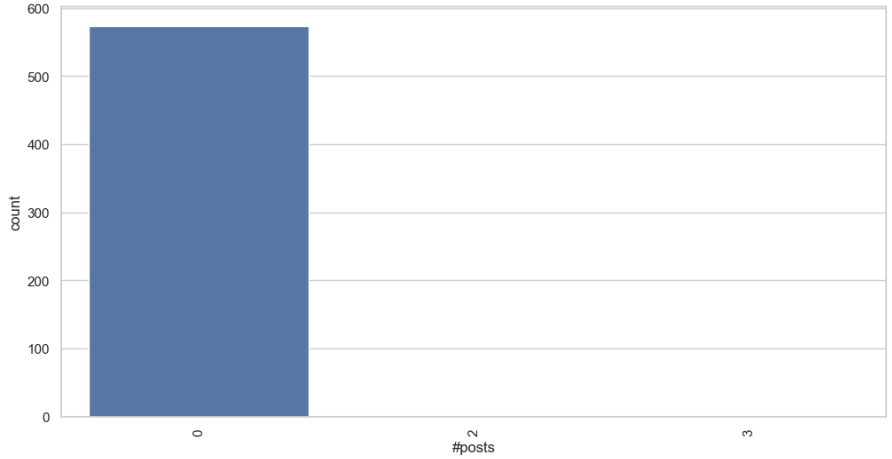
圖二十六、dexcription length



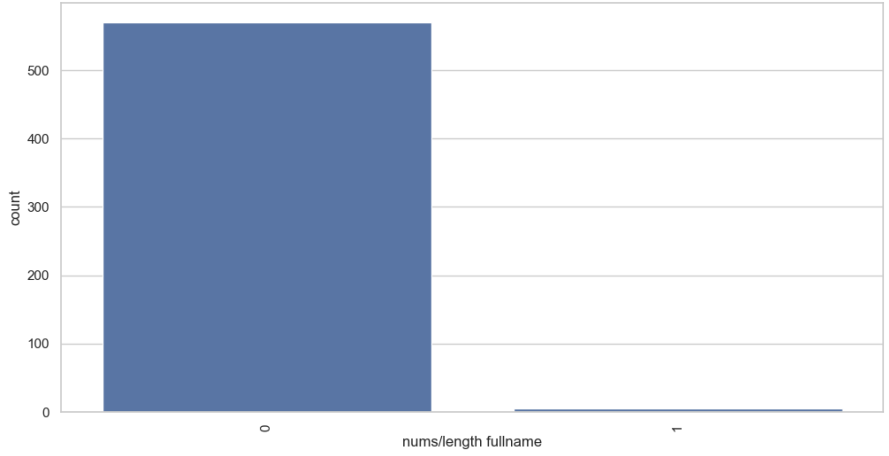
圖二十七、#follows



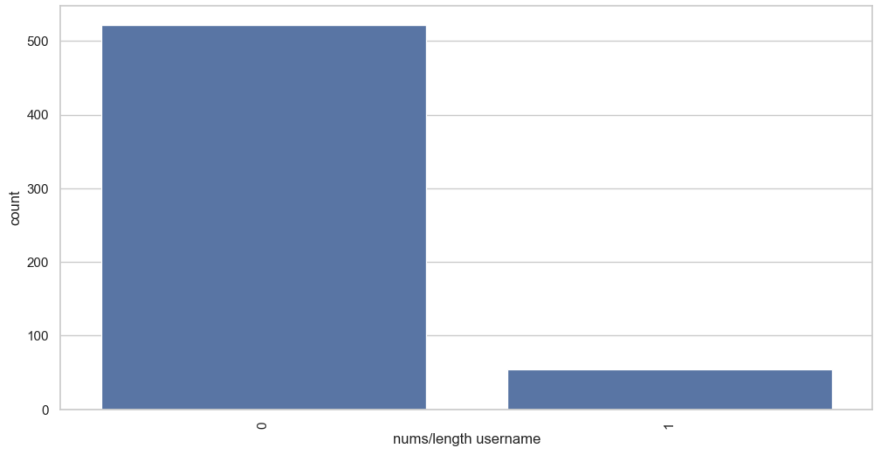
圖二十八、#followers



圖二十九、#posts



圖三十、nums/length fullname



圖三十一、nums/length username

從結果中可以發現，這些原本都有些連續的資料都變得相當的離散，方便決策數更容易的判斷何時該做 split，也不會在 root 中生成太多 branches 出來。

三、建立模型：

我的決策樹模型的程式結構共分成以下三個 Class：

- ***class TreeLeaf:***：

```
class TreeLeaf:
    def __init__(self, value):
        self.value = value
```

圖三十二、class Treeleaf.__init__()

主要是作為決策樹的枝葉的葉片，只儲存 fake 參數。

- ***class DecisionFork:***：

```
class DecisionFork:
    def __init__(self, attribute, attribute_name, default_value):
        self.attribute = attribute
        self.attribute_name = attribute_name
        self.default_value = default_value
        self.branches = {}

    def add(self, value, subtree):
        self.branches[value] = subtree
```

圖三十三、class DecisionFork()

在這個 class 中，主要是存決策樹中每一個決策點的參數，包含要做決策的 attribute 以及它的名字 attribute_name（這裡是以數字

編號做為它的名子）、此決策點儲存的 fake 參數 default_value、和此決策點所連結的枝葉 branches。

除此之外，它也有一個 add(self, value, subtree) 函式，用來處理作業講義中的 psuedocode 的 subtree 加入主樹幹 tree 的部分。

- ***class DecisionTree:***：

最後的這個 class，就是我們的主角－決策樹本樹。首先，在創造決策樹物件時，它會先做以下的初始化：

```
class DecisionTree:
    def __init__(self, min_samples_split=0, max_depth=999):
        self.min_samples_split = min_samples_split
        self.max_depth = max_depth

        self.target = '11' # the class (`fake`)
        self.attr_name = list(train.columns) # the actual name of the attributes
        self.root = None # the decision tree
```

圖三十四、class DecisionTree.__init__()

這些初始化的目的在於指定此樹的 classification column name、定義字串型態的 atrributes name list、以及 tree 的 root。當然，=最小 atrribute 分割和最大樹深的限制的定義也是在這裡完成，但這是後續優化部分要做的，現在先不做相關的定義和比較。

而為了能夠將訓練資料輸入進這棵樹，我在這個 class 中定義了一個 fit 函式，程式碼為以下：

```
# do the simple initialization of the given splited data for fitting the model
def fit(self, X, y):
    # initialize the example and sample for decision tree
    # convert X_train and y_train to DataFrames
    X_df = pd.DataFrame(X)
    y_df = pd.DataFrame(y, columns=['11']) # define colemn `11` for the class of original dataset
    # concatenate X_df and y_df
    example = pd.concat([X_df, y_df], axis=1).to_dict('records') # change it to dictionary
    attribute = list(range(len(example[0])))

    # build a decision tree and save the tree to the `self.root`
    self.root = self.decision_tree_learning(example, attribute)
```

圖三十五、class class DecisionTree.fit()

內容就是將輸入進來的 X（作業中指定為 feature column 的資料）和 y（作業中指定為 class column 的資料）變成 pandas DataFrame，並將它們用 pandas.concat() 去做合併，再轉成 dictionary 的資料型態（此 class 裡面大部份的函式用 dictionary 資料型態會比較方便）。以及，將合併後的 example 的 colemn 數目用 range 做成 list，代入進 attribute 中，代表每一行的 column 名字。

其中，輸入進來的 X 和 y 是用以下的方式對原本的 train 資料集做切割：

```
# Split the Data into feature and class
X = train.drop("fake", axis=1).values # features
y = train["fake"].values # class

X_sample = test.drop("fake", axis=1).values # features
y_sample = test["fake"].values # class

from sklearn.model_selection import train_test_split

# Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42) # 8:2 segmentation
```

圖三十六、training data X, y and test data X_sample, y_sample generation

Train.csv 和 test.csv 的 X 和 y 的分開方式主要是用 pandas 的 drop，配合 numpy 的 value() 將其轉成 ndarray 的形式。至於 train:test = 8:2 切割的方式用的是 sklearn 的 train_test_split，配合 test_size = 0.2 和 random state = 42 的設定，完成 train 資料的切割。

在資料經過 fit() 函式的匯入後，fit() 函式的最後一行會啟動 decision tree 的創造，也就是驅動函式 decision_tree_learning()。decision_tree_learning() 的運作構造為下圖所示：

```
# generated from the psuedocode at `Figure 18.5`
def decision_tree_learning(self, examples, attributes, parent_examples=(), depth=0):
    if len(examples) == 0:
        return self.plurality_value(parent_examples)
    if self.all_same_class(examples):
        return TreeLeaf(examples[0][self.target]) # the classification
    if len(attributes) == 0 or len(attributes) < self.min_samples_split or depth >= self.max_depth:
        return self.plurality_value(examples)
    A = self.choose_attribute(attributes, examples)
    tree = DecisionFork(A, self.attr_name[int(A)], self.plurality_value(examples))
    for (v_k, exs) in self.split_by(A, examples):
        subtree = self.decision_tree_learning(exs, self.remove_all(A, attributes), examples, depth+1)
        tree.add(v_k, subtree)
    return tree
```

圖三十七、class DecisionTree.decision_tree_learning()

其實 decision_tree_learning() 基本上就是藉由作業的 psuedocode 所寫成的。前三個 if 判斷式做例外的處裡，判斷沒事後就找尋現存的最佳分割 attribute，並以該 attribute 創造樹或子樹。最後用遞迴的方式遍歷被選中的 attribute split 的 example 所產生的 item，去生成現在這棵樹的子樹，並在遞迴完後加入現在的樹的分支中。

有稍微不一樣的是第三個 if 判斷式的後兩個條件，它們是當剩下的 attribute 小於指定的最高限度切割次數 min_sample_split 或者是樹深超過指定的深度，就會直接回傳 plurality_value()。這兩個判斷式後續的定義和觀察留待下一大段做使用和分析。

在 decision_tree_learning() 中，我對於 plurality_value() 的定義為

以下：

```python
def plurality_value(self, examples):
    # Return the most common target value in the examples, breaking ties randomly
    # note that examples is a list of dictionaries, and self.target is the target attribute
    target_values = [example[self.target] for example in examples]
    # Manually count occurrences
    counts = {}
    for value in target_values:
        counts[value] = counts.get(value, 0) + 1
    # Find the most common value
    most_common = sorted(counts.items(), key=lambda x: x[1], reverse=True)

    # If there is a tie, shuffle the tied values and select one randomly
    if len(most_common) > 1 and most_common[0][1] == most_common[1][1]:
        # tied_values -> list
        tied_values = [value for value, count in most_common if count == most_common[0][1]]

        # shuffle the tied_value and return the first one of the shuffled data
        random.shuffle(tied_values)
        return TreeLeaf(tied_values[0])
    else:
        return TreeLeaf(most_common[0][0])
```

圖三十八、class DecisionTree.plurality_value()

函式 plurality_value() 在做的任務是找到一個在該輸入的 examples 中最多的共同 target value。白話說，其實就是看這個 example 中哪個 `fake` 參數（也就是 1 或 0）比較多，從而回傳那個數值，也就是前五行程式在做的事。如果都一樣多的話，就使用我下方 if 條件式中的隨機方式將那些 `fake` 參數位置做隨機，並回傳隨機完後的第一個元素。

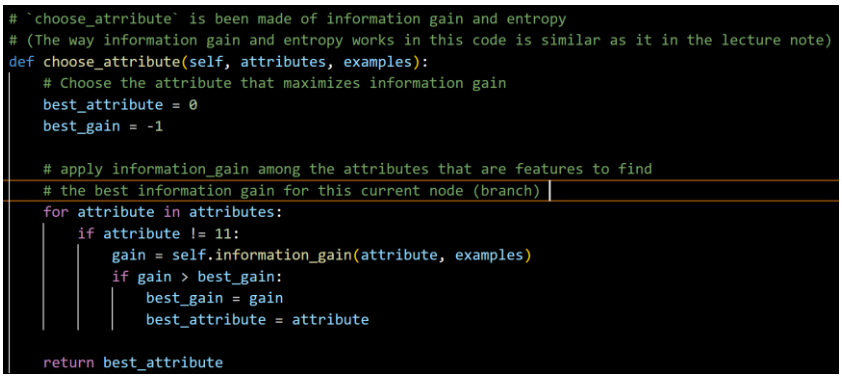
除此之外，decision_tree_learning() 裡的 all_same_class() 製作方式為以下的程式碼截圖：

```python
def all_same_class(self, examples):
    # Check if all examples have the same target class
    return len(set([example[self.target] for example in examples])) == 1
```

圖三十九、class DecisionTree.all_same_class()

主要就是回傳目前的 example 的 `fake` class 是否都是一樣的，以給 decision_tree_learning() 判斷需不需要直接回傳樹葉給上一次的函式呼叫。

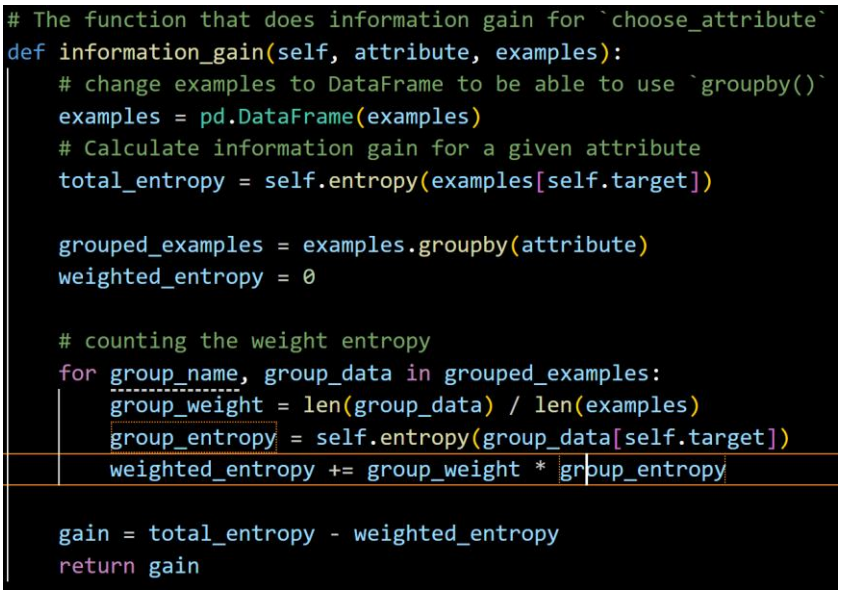
在經歷了一些例外處理後，decision_tree_learning() 會需要做當前最適合用來 split 的 attribute 選取，也就是 choose_attribute() 函式的使用目的。它的構成方式為以下：



```
# `choose_atrribute` is been made of information gain and entropy
# (The way information gain and entropy works in this code is similar as it in the lecture note)
def choose_attribute(self, attributes, examples):
    # Choose the attribute that maximizes information gain
    best_attribute = 0
    best_gain = -1

    # apply information_gain among the attributes that are features to find
    # the best information gain for this current node (branch)
    for attribute in attributes:
        if attribute != 11:
            gain = self.information_gain(attribute, examples)
            if gain > best_gain:
                best_gain = gain
                best_attribute = attribute

    return best_attribute
```

圖四十、class DecisionTree.choose_attribute()

從上面的程式碼截圖可以發現，choose_attribute() 的 attribute 選擇方式靠的是 information gain。我將做該事的函式 information_gain() 定義為以下程式碼：



```
# The function that does information gain for `choose_attribute`
def information_gain(self, attribute, examples):
    # change examples to DataFrame to be able to use `groupby()`
    examples = pd.DataFrame(examples)
    # Calculate information gain for a given attribute
    total_entropy = self.entropy(examples[self.target])

    grouped_examples = examples.groupby(attribute)
    weighted_entropy = 0

    # counting the weight entropy
    for group_name, group_data in grouped_examples:
        group_weight = len(group_data) / len(examples)
        group_entropy = self.entropy(group_data[self.target])
        weighted_entropy += group_weight * group_entropy

    gain = total_entropy - weighted_entropy
    return gain
```

圖四十一、class DecisionTree.information_gain()

在這個函式中，我一開始先把 examples 變回 pandas DataFrame 的資料型態，以方便使用 pandas 的 groupby() 功能，將資料以 attribute 為索引值去做整理並計算。在 information gain 的做法中，通常都是用 entropy 去代表 information 的 value。因此，我也定義了用來算 entropy 的函式 entropy()，其定義為以下程式碼：

```python
# The function that count entropy of the labels for the `information_gain`
def entropy(self, labels):
    # Calculate entropy for a set of labels
    total_samples = len(labels)
    if total_samples == 0:
        return 0

    unique_classes, counts = np.unique(labels, return_counts=True)
    probabilities = counts / total_samples
    entropy_value = -np.sum(probabilities * np.log2(probabilities))

    return entropy_value
```

圖四十二、class DecisionTree.entropy()

這裡的 entropy 主要是參考了上課講義的等式來做成，藉由對於指定 attribute 的 labels 計算，定義出一個該 attribute 在某個場合下的 entropy 值。其中有用到 np.unique() 去輸出 labels 中的 unique classes。

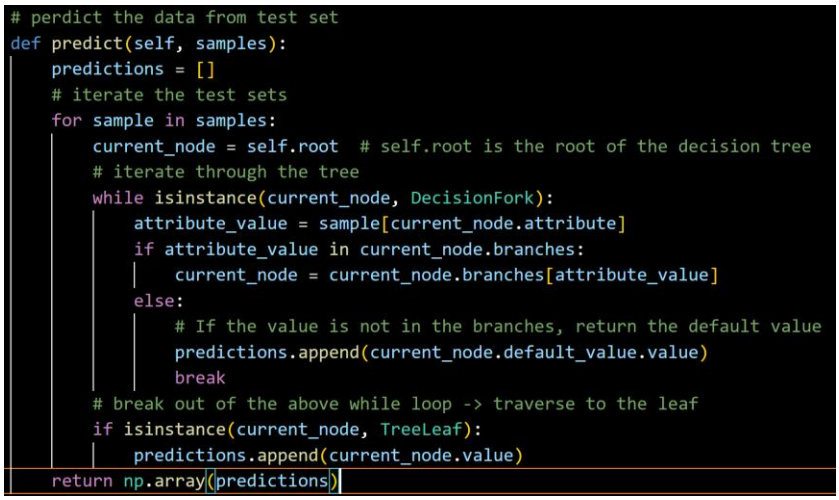
最後的話，decision_tree_learning() 要遍歷每一個被 attribute split 分割的 v_k 和 exs 去遞迴生成此樹的子樹。用來做切割的 split_by() 函式被定義為如下程式碼：

```python
# Split examples into groups based on the values of the specified attribute
def split_by(self, attribute, examples):
    grouped_examples = {} # dictionary type
    for example in examples:
        value = example[attribute]
        if value not in grouped_examples:
            grouped_examples[value] = []
        grouped_examples[value].append(example)
    return grouped_examples.items()
```

圖四十三、class DecisionTree.split_by()

這段程式主要就是將目前遞迴到的 examples 對用 choose_attribute() 選出來的 attribute 做基於它的分群。其中儲存被分群的 example 的 group_example 用的是 dictionary 資料型態。

在樹生成完成後，我們會需要一個函式掌管 test set 的輸入，並回傳此決策樹的預測結果。為了達成這個目的，我在這個 class 定義了一個函式 predict()，結構為以下：

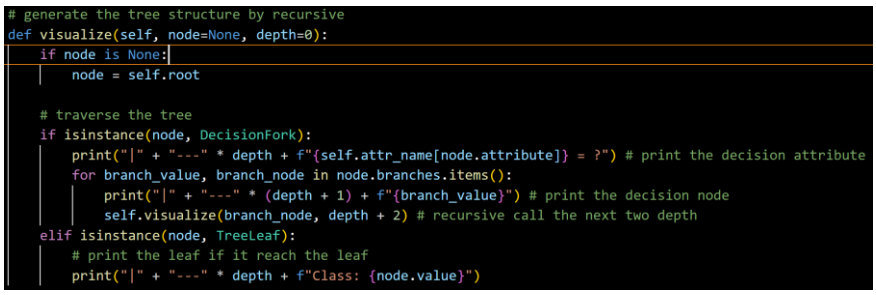


```
# perdict the data from test set
def predict(self, samples):
    predictions = []
    # iterate the test sets
    for sample in samples:
        current_node = self.root  # self.root is the root of the decision tree
        # iterate through the tree
        while isinstance(current_node, DecisionFork):
            attribute_value = sample[current_node.attribute]
            if attribute_value in current_node.branches:
                current_node = current_node.branches[attribute_value]
            else:
                # If the value is not in the branches, return the default value
                predictions.append(current_node.default_value.value)
                break
        # break out of the above while loop -> traverse to the leaf
        if isinstance(current_node, TreeLeaf):
            predictions.append(current_node.value)
    return np.array(predictions)
```

圖四十三、class DecisionTree.predict()

函式 predict() 要做的事就相對簡單一些，就是取出 test set 的資料並拿去遍歷所有的樹的節點，遇到無法行動的節點或到達樹葉就分別回傳該節點的 default 預測結果或該樹葉存放的預測結果。最後回傳一個存放每個 test case 結果的 ndarray。

除此之外，我的樹的視覺化函式也是定義在這個 class 中，構造為以下：



```
# generate the tree structure by recursive
def visualize(self, node=None, depth=0):
    if node is None:
        node = self.root

    # traverse the tree
    if isinstance(node, DecisionFork):
        print("|" + "---" * depth + f"{self.attr_name[node.attribute]} = ?") # print the decision attribute
        for branch_value, branch_node in node.branches.items():
            print("|" + "---" * (depth + 1) + f"{branch_value}") # print the decision node
            self.visualize(branch_node, depth + 2) # recursive call the next two depth
    elif isinstance(node, TreeLeaf):
        # print the leaf if it reach the leaf
        print("|" + "---" * depth + f"Class: {node.value}")
```

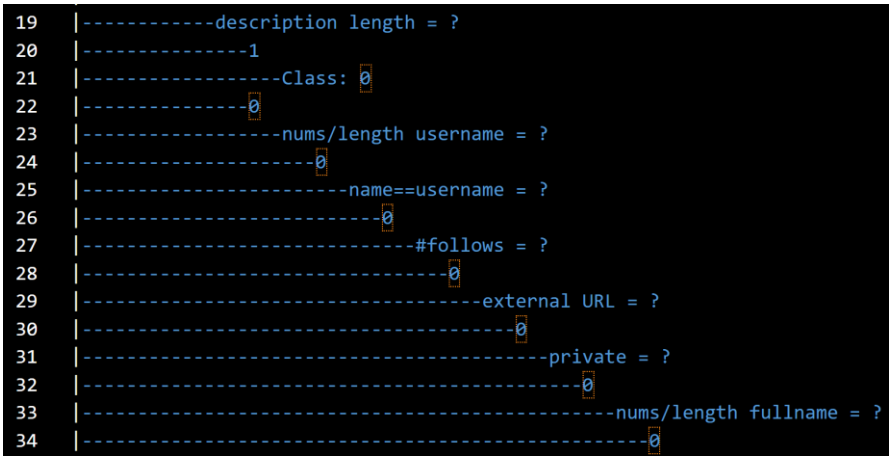
圖四十三、class DecisionTree.visualize()

這個函式做的事就是從上到下，做類似 depth-first search 的方式將整棵樹遍歷並印出。實際印出的結果圖長什麼樣子，可以看下一個標題的內容或作業資料夾中的附錄。

四、**結果及優化**：

1. 結果：

首先，先展示一下樹沒經過優化的結構。以下為其中一段結構為：

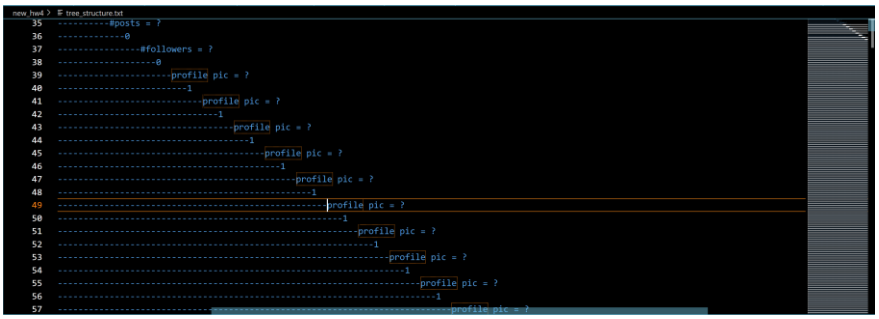


圖四十四、未優化的樹結構截取（一）

（一小段截圖，完整版可以到 figure 資料夾中觀看）

基本上來說，配合上一個標題的圖四十三的話，一筆資料在做預測時會到每一個節點做停留和辨認，確認自己的特徵值是否吻合或相近，再繼續往下一個 branch 邁進。不過，如果在經過資料前處理後的某些資料特徵值無法對上節點的特徵值的話，就會直接停留並回傳該節點的 default 預測值；反之，到了樹葉，就會回傳該樹葉的預測值

但是，其實這棵樹如果有再往下看的話，他還有 overfitting 的問題。以下的一段結構可證明此事：



圖四十五、未優化的樹結構截取（二）

（一小段截圖，完整版可以到 figure 資料夾中觀看）

從這裡可以很明顯的看出他的深度和 split 有很多，且有不少是無意義的行為。

接下來，我要展示我的樹在沒經過優化時的精確度測試。

在這個區塊中，我一共會做兩種測試。第一種是針對原本 train.csv 所分割出來的 test set – 特徵值 X_test 和 class 值 y_test – 去做此 train.csv 的精確度測試（關於資料的生成可前往圖三十六）。測試程式碼為以下：

```
# build the decision tree object
dt = DecisionTree()

# fit the training data to the tree
dt.fit(X_train, y_train)

# perdict the value from the test case
y_pred = dt.predict(X_test)
print(y_pred)

# calculate accuracy
correct_predictions = np.sum(y_test == y_pred)
total_samples = len(y_test)
acc = correct_predictions / total_samples
print(f"Accuracy: {acc:.3f}")
```

圖四十六、X_test 和 y_test 的測試程式

得到的結果為以下：

```
[0 1 1 1 0 1 0 0 0 0 0 1 1 0 0 1 0 0 1 0 1 0 1 0 1 0 1 1 0 0 0 1 1 1 0 0 0
 0 0 0 1 0 1 0 0 0 0 0 0 1 1 1 0 0 0 1 0 1 0 0 0 0 1 0 1 1 0 0 0 1 1 1 0 0
 0 1 1 0 0 0 1 1 0 0 0 1 1 0 1 0 0 0 1 1 1 0 0 1 0 0 0 1 0 0 1 1 0 1 0 1 1
 1 0 0 0 0]
Accuracy: 0.793
```

圖四十七、X_test 和 y_test 的測試結果

得到了 0.793 的精確度

第二種，我們將以 test.csv 的資料去做精確度的分析。測試程式碼如下：

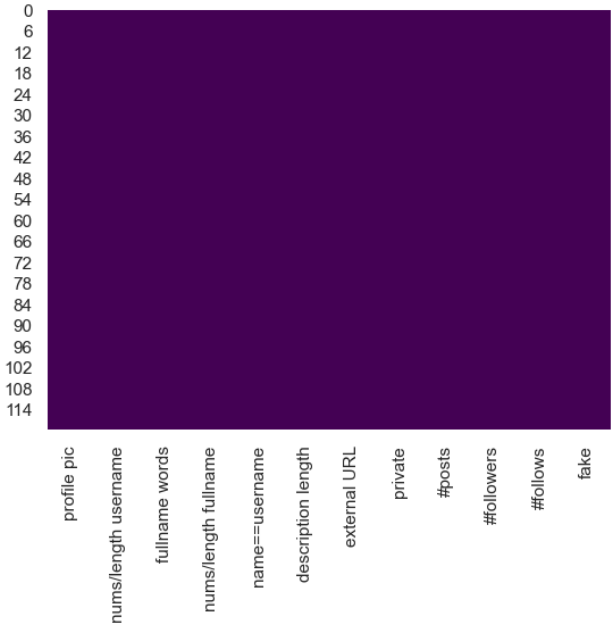
```
# perdict the value from the test case
y_pred = dt.predict(X_sample)
print(y_pred)

# calculate accuracy (by `sklearn.metrics`)
correct_predictions = np.sum(y_sample == y_pred)
total_samples = len(y_sample)
acc = correct_predictions / total_samples
print(f"Accuracy: {acc:.3f}")
```

圖四十八、針對 test.csv 資料的預測精確度測試程式

其中，我想針對 test.csv 的資料做簡單的分析。這裡我只針對原始資料

缺值、column 資料分布做分析，因為它和 train.csv 都是差不多的資料，不需用做已知無效的分析。並且，它的資料前處理的方式和 train.csv 是一樣的，所以預處理前後的區別也可以參考 train.csv 的結果。

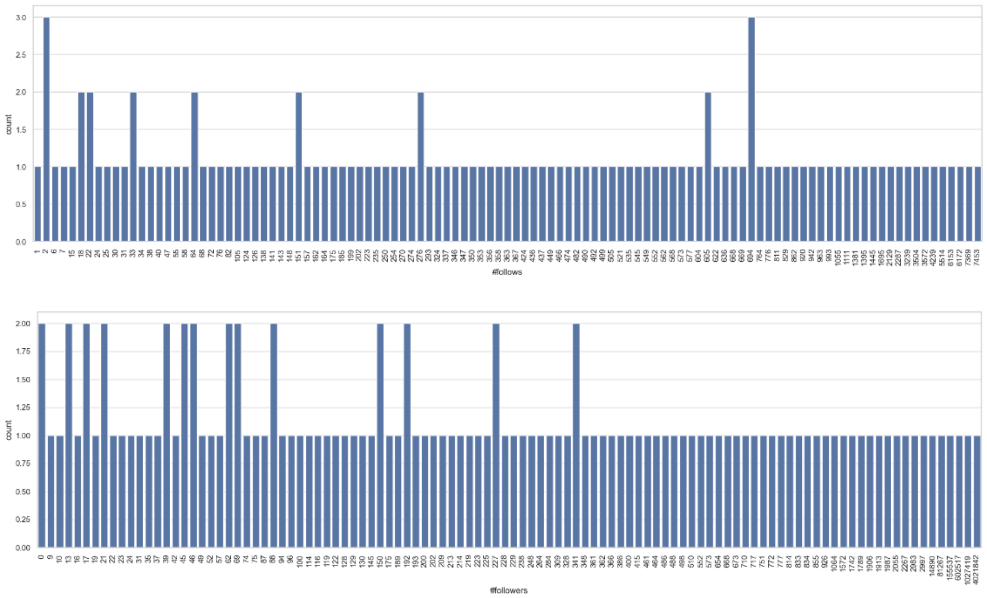
我們先看資料缺值的部份，使用的是 seaborn.heatmap。

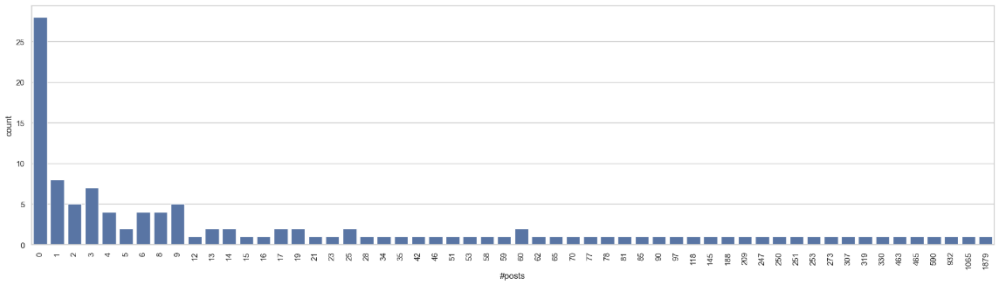


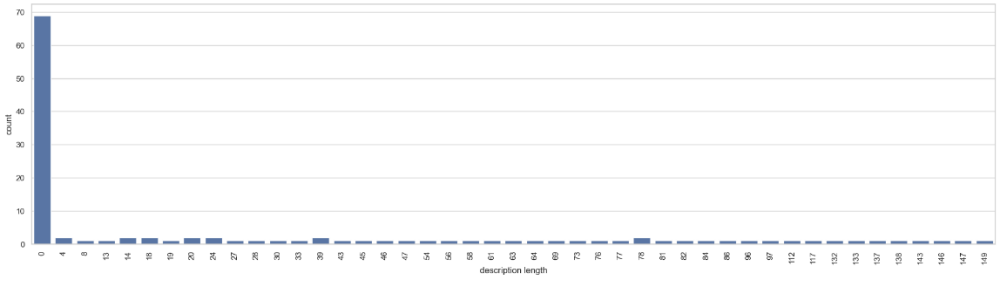
圖四十九、real data heap map in test.csv

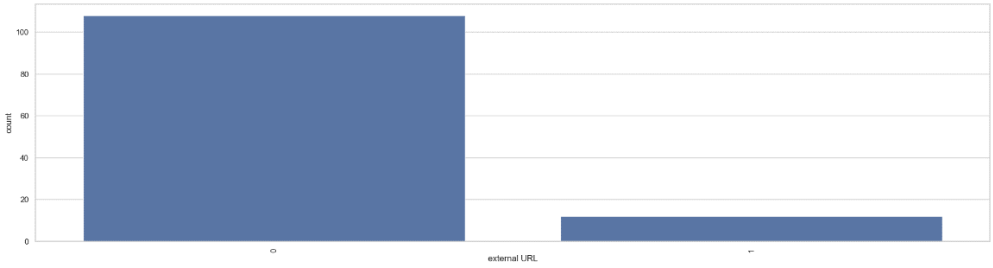
從上圖中可以發現到，test.csv 並沒有資料缺值

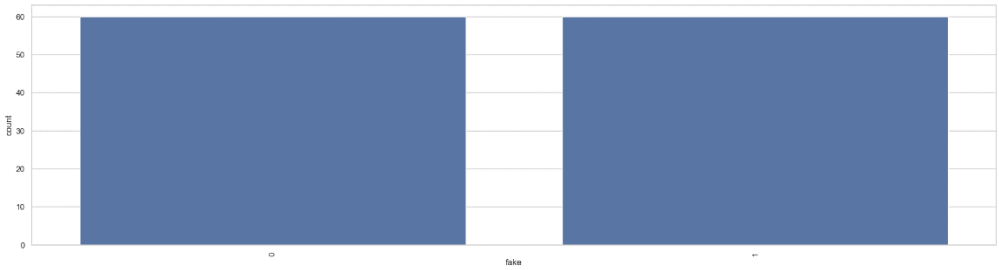
那我們接下來看到 column 資料分布的部份。

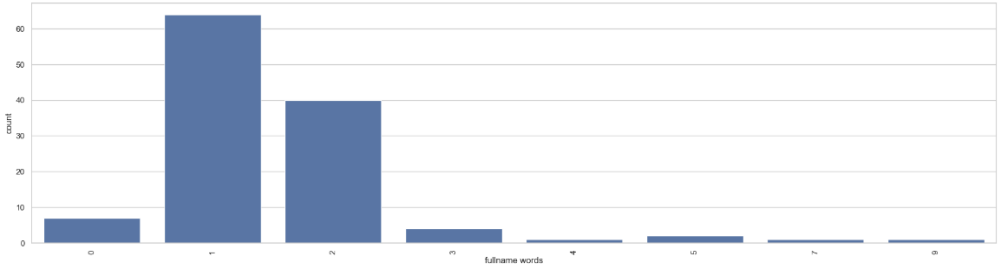


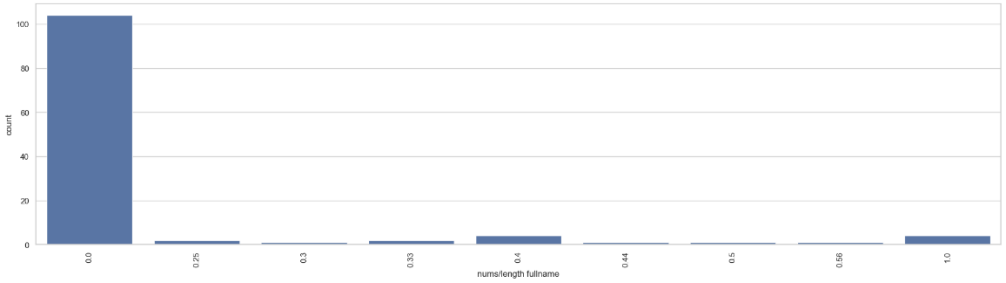

count
#posts

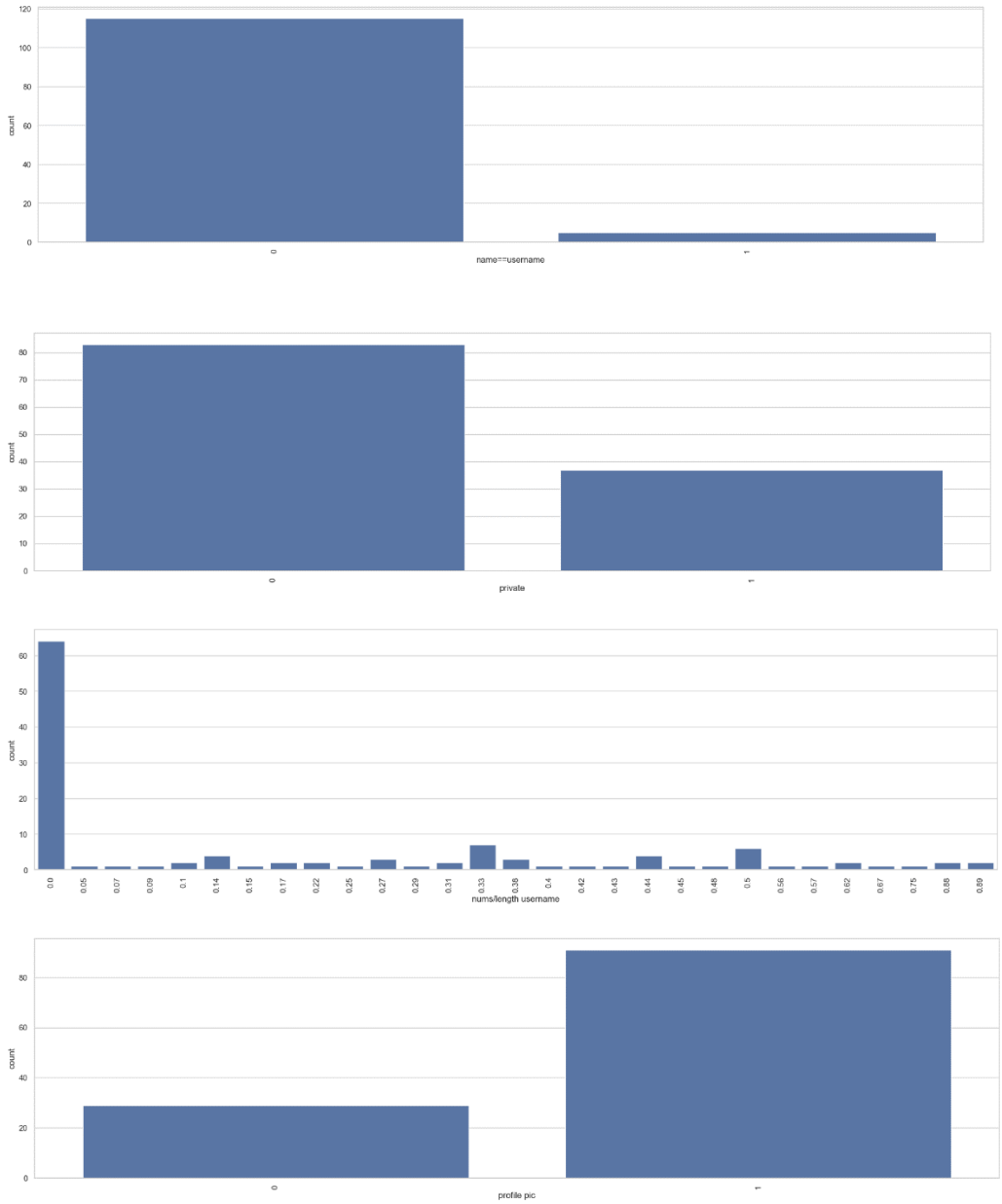

count
description length


count
external URL


count
fake


count
fullname words


count
nums/length fullname



圖五十、test.csv 的 columns 資料分佈

從上面的圖組我們可以看到，description length、#follow、#followers、#post、nums/length username、以及 nums/length fullname，一樣都是呈現類似連續性的分佈。而其他的資料也都是二元性的分布，換句話說就是離散型的資料分佈。

並且，test.csv 中被設定為 class 的 column – `fake` 的資料分佈，和 train.csv 一樣，是呈現 5:5 的樣態。

我們回來到第二種精確度分析的部份。圖四十八的程式碼運作後，可以得到以下的結果：

```
[1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 1 1 0 0 1 0 0 0 0 0 0 0 0
 0 0 1 1 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 0 1 1 1 0 0 1 1 0 1 0 1 0 1 1 0 0 0 1 0 1 1 0 0 1 1 1 1 0 1 1 0 0
 0 1 1 0 0 0 1 1 0]
Accuracy: 0.733
```

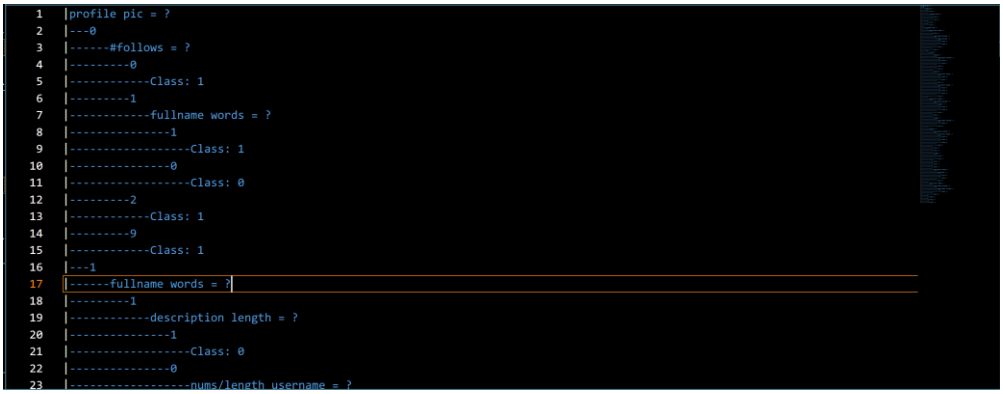
圖四十八、針對 test.csv 資料的預測精確度測試結果

可以發現精確度在 0.733，和第一種的分析相近。

2. 優化：

在優化的部分，為了能夠使我的 decision tree 有更好的效率，我在建造新的決策樹物件時，定義了上一段沒定義的 min_samples_split 和 max_depth，分別用來 attribute 切割的數量和限制樹的深度。

我這裡只做了 min_samples_split=3、max_depth=5 的優化測試，可以得到以下的結構圖的一部分：



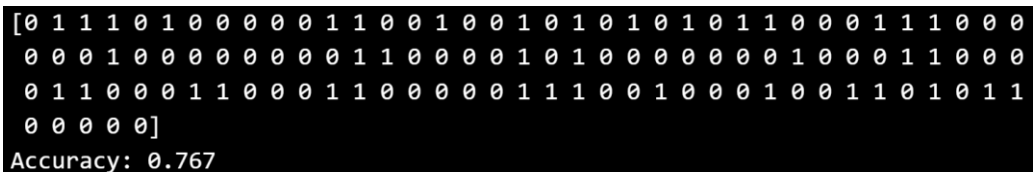
圖四十九、優化後的決策樹結構

（一小段截圖，完整版可以到 figure 資料夾中觀看）

從圖中的樹的結構以及右上方有點微弱的預覽，可以發現到，樹的結構比起沒優化的版本縮小了許多，變得更加容易被人所觀察，也同時可以使得資料預測的速度增快，但也有可能會丟失一些些判斷訊息。

至於資料預測的精確度的部份，我和未優話的區塊一樣做兩種精確度分析。

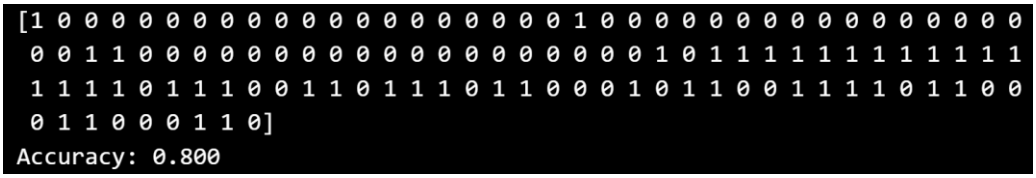
我們先來看第一種分析，也就是針對原本 train.csv 所分割出來的 test set – 特徵值 X_test 和 class 值 y_test – train.csv 的精確度測試。測試程式就如同圖四十六一樣，我就不再重複貼上截圖。藉由相同的測試程式，可以得到以下的結果：



圖四十九、X_test 和 y_test 的測試結果（優化樹）

結果為 0.767 的精確度結果，比起原本的 0.793 還要沒那麼精確。

但是，當我們使用優化過的樹，做第二種精確度分析，也就是將 test.csv 的資料去做精確度的測試的話，反而得到了以下的結果：



圖四十九、針對 test.csv 資料的預測精確度測試結果（優化樹）

結果竟然是 0.800，大於未優化的預測精確度。

從這裡我們就可以發現，決策樹並不是說結構越大就越好，反而要小心有沒有 overfitting 的存在，使得樹的結構過於臃腫；而且從未優化和優化的結果比對也可以發現，其實優化後（剪枝後）的樹的預測精準度其實沒比優化前來的弱，反而會強一些。