

影像表格文字化的實作與學習

作者：薛詠謙

目錄

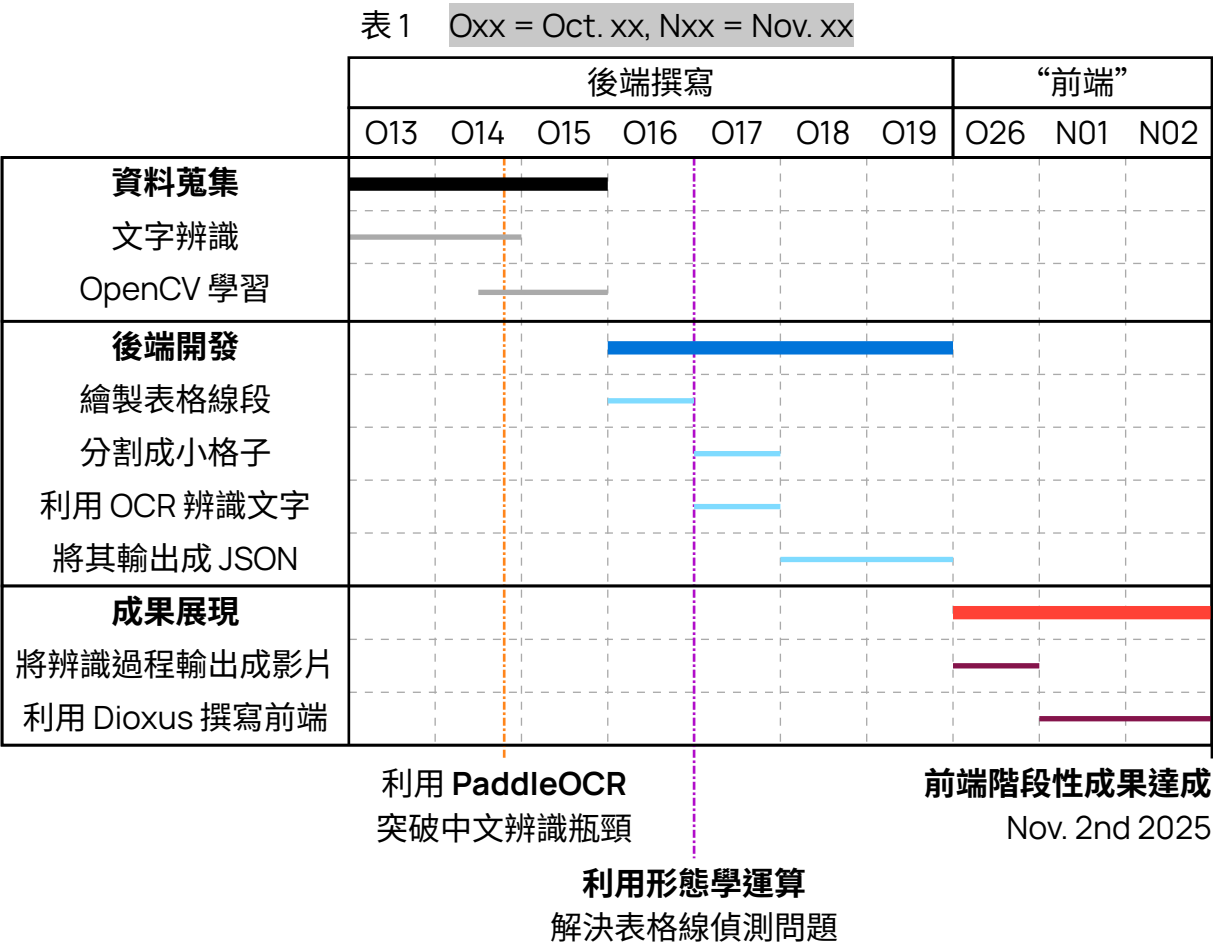
壹、動機	2
貳、時間線(甘特圖)	2
參、資料蒐集	3
一、影像預處理	3
(一) 二值化	3
二、文字辨識	3
(一) Tesseract	3
(二) PaddleOCR	4
(三) 小結	4
肆、操作方法	4
一、外圍邊線辨識與分割	4
(一) 消除文字	5
(二) 辨識邊線	5
(三) 分割表格區段	6
程式架構演進：為什麼需要區域切割？	6
二、表格內部邊線辨識	7
(一) 前製處理	7
撰寫歷史	7
(二) 偵測橫線	8
(三) 偵測直線	9
(四) 統一線寬	9
關於矩陣轉置策略：	9
關於為何採用 <code>keep_last_line</code> ：	9
(五) 相交點計算	10
布林運算的好用之處	10
三、單元格分割	11
(一) 原始演算法	11
原始演算法簡介	11
(二) 最終演算法：	12
文字辨識：	12
四、資料結構化	13
伍、我的收穫與反思	13
參考文獻	13

壹、動機

在尋找合適校系的過程中，我發現 [University TW](#) 所提供的分數搜尋功能非常實用。不過，該網站的資料更新速度並不一定與官方同步，而 **大學甄選入學委員會** 所公布的錄取分數資料又多以 **圖片表格** 的形式呈現，難以直接整理或分析。

因此，我產生了想要將影像表格轉換成可數位化利用文字資料的想法，因此，我開始學習 **OCR**（光學文字辨識）技術，並嘗試實現影像中文字資料的自動化擷取與轉換。

貳、時間線(甘特圖)¹



專案現況與規劃：

- 截至 11 月 2 日，我已完成：
- 後端辨識引擎 (100%)
 - 表格切割與資料結構化 (100%)
 - [辨識過程視覺化影片](#)
 - 前端介面開發中 (85%、剩下與大考中心結合的部分)

¹平日每天約莫花兩小時、週末每天約莫花六小時在開發與資料蒐集上。

開發策略：

我採用了 Proof-of-Concept 優先的開發方式：先快速實作核心功能 驗證技術可行性，再回頭處理 technical debt 和程式品質優化。這種迭代開發策略讓我能最短時間內，專注於最關鍵的技術挑戰——表格辨識與切割的演算法實作。

叁、資料蒐集

在有了這個想法之後，大約花了週末左右做資料蒐集，尋找是否有前人所做之成果。在閱讀了 [1] 與 [2] 之後，我將影像表格分析流程簡略分為以下幾個步驟：

1. **影像前處理**：對原始影像進行去噪、二值化或對比度等調整。
2. **表格分割**：偵測並標註表格的行列結構，將整體表格拆分為單元格。
3. **文字辨識**：對每個單元格進行 OCR 辨識，取得對應文字內容。
4. **轉換輸出格式**：將最終文字結果整理成結構化資料，如 `csv` 或 `json`。

一、影像預處理

(一) 二值化

`threshold(thresh: f64)` 函式可將灰階影像轉換為純黑與純白的二值化影像。然而，由於中文字筆劃粗細不一，閾值 (`thresh`) 的設定若不恰當，可能會降低文字辨識的準確度。

檢定標準	檢定標準	檢定標準
圖 1 閾值=100	圖 2 閾值=150	圖 3 閾值=200

因此，雖然在此範例中使用 圖 2（閾值 = 150）的二值化處理結果尚可接受，但鑒於本專案涉及不同文字樣式，最終我決定在文字辨識的預處理階段**不採用二值化**。

二、文字辨識

(一) Tesseract

將中文文字影像直接輸入 Tesseract 進行辨識時，即使經過灰階化等前處理，辨識效果仍不盡理想。推測原因可能在於大考中心提供的檔案**解析度**過低^[3]，導致模型無法正確辨識文字。

```
tesseract figure-1.png -l chi_tra
```

無法辨識出文字

不過，Tesseract 在數字的辨識上表現十分精準，能穩定且無誤地擷取出校系代碼欄位的資料。

```
tesseract figure-2.png -c  
tessedit_char_whitelist=0123456789
```

成功辨識出 011012

(二) PaddleOCR

PaddleOCR 是由百度開發的文字辨識工具，相較於 Tesseract，在中文文字的擷取上具有顯著的提升。

在大多數情況下，PaddleOCR 能夠成功辨識中文文字，其特色包括：

- 對中英文混合及標點符號具有良好容錯性，大部分情況下都能正確辨識，例如在圖 6 成功辨識出「(英文+數學 A)25」。
- 少數情況下可能漏字，例如在圖 4 中僅辨識出「(華語文教學組)」。
- 純數字辨識有時也會漏字，如在圖 5 中僅辨識出「1012」。

中國文學系乙組(華語文教學組) 圖 4	011012 圖 5	(英文+數學A)25 圖 6
------------------------	---------------	-------------------

(三) 小結

綜上所述，將 tesseract 用於單元格的純數字辨識，而將 PaddleOCR 作為主要文字辨識工具，似乎為最佳組合。

肆、 操作方法

根據圖 7 所示，來源影像包含多個表格區段，必須先將主要區塊切出，才有辦法進行更細緻的分割。因此，需要先辨識各區域的**外圍邊線**並進行分割，再辨識表格內的**分割線**，將表格切分成單元格，最後進行**文字辨識**，並將結果整理成結構化的 JSON 格式。

圖 7

處理流程可簡述如下：

外圍邊線辨識 → 區域分割 → 表格邊線辨識 → 單元格分割 → 文字辨識 → 將資料結構化

一、 外圍邊線辨識與分割

根據 [1]，可使用 `find_contours_with_hierarchy` 函數進行直線邊緣偵測。然而，此方法同時會偵測到文字筆劃中的直線，導致難以準確判定外框邊界。

(一) 消除文字

為消除文字干擾，可採用膨脹與腐蝕（morphological closing）處理。

```
let kernel =  
    get_structuring_element(imgproc::MORPH_RECT, Size::new(20, 5),  
    Point::new(-1, -1))?;  
  
let mut closed = Mat::default();  
  
morphology_ex(  
    &thresh,  
    &mut closed,  
    imgproc::MORPH_CLOSE,  
    &kernel,  
    Point::new(-1, -1),  
    1,  
    BORDER_CONSTANT,  
    Scalar::default(),  
)?;
```

程式 1 其中 `thresh` 為經過二值化的影像

由 程式 1 和轉換過後的 圖 8 可見，以長 20、寬 5 的矩形結構元素（kernel）進行運算，能有效的腐蝕掉文字，不過同時也腐蝕掉了表格。為保留表格結構，可將處理後的影像與原始影像進行 `bitwise_or` 運算，以刪除文字，如 圖 9 所見。



圖 8 經過腐蝕過後的圖像

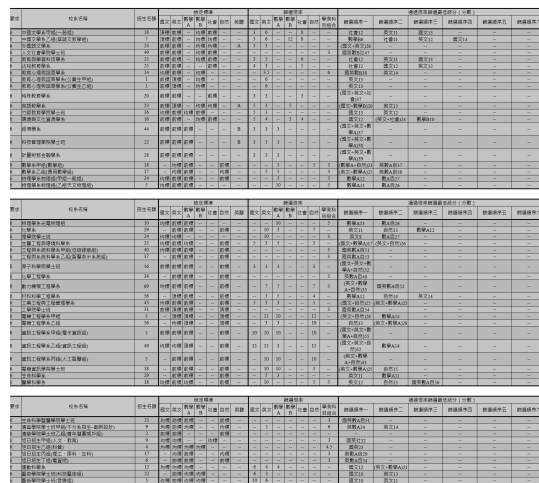


圖 9 經 bitwise_or 合併的圖像

(二) 辨識邊線

如程式 2 所示，將影像經由 `threshold` 函數進行二值化處理後，再執行 `bitwise_not` 運算，即可得到如 圖 10 之結果。

```
let mut thresh = Mat::default();
threshold(&output, &mut thresh, 254.0, 255.0,
THRESH_BINARY)?;
bitwise_not(&thresh.clone(), &mut thresh, &no_array())?;
```

程式 2 其中 `output` 為經過 `bitwise_or` 處理過後的影像

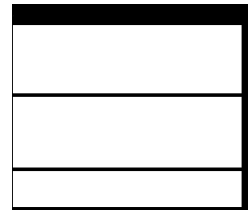


圖 10

(三) 分割表格區段

```
let mut contours = Vector::<Vector<Point>>::new();
let mut hierarchy = Vector::<Vec4i>::new();
find_contours_with_hierarchy(
    &thresh,
    &mut contours,
    &mut hierarchy,
    imgproc::RETR_EXTERNAL,
    imgproc::CHAIN_APPROX_SIMPLE,
    Point::new(0, 0),
)?;
let mut blocks: Vec<Rect> = contours
    .iter()
    .map(|contour| {
        let rect = imgproc::bounding_rect(&contour).unwrap();
        let area = (rect.width * rect.height) as f64;
        (rect, area)
    })
    .collect();

// sort by y-values first
blocks.sort_by(|a, b| a.y.cmp(&b.y).then_with(|| a.x.cmp(&b.x)));

let blocks_mat = blocks
    .into_iter()
    .map(|rect| Mat::roi(img, rect).map(|x| x.clone_pointee()))
    .collect::<Result<Vec<_>, opencv::Error>>()?;
```

程式 3 其中 `thresh` 為經過 `bitwise_not` 處理過後的影像

最後如 程式 3 所見，利用 `find_contours_with_hierarchy` 與 `Mat::roi` 即可將原影像分割成三個子區塊。

程式架構演進：為什麼需要區域切割？

專案初期，我是直接將完整影像送入 小節 二 處理。但發現了一個問題：來源影像包含多個表格區段，而區塊之間的 garbage data（標題、註解文字等）會干擾 小節 四 的邏輯判斷。

透過長久的 debug，發現問題不在邊緣辨識，而是缺少預處理步驟。因此回頭開發區域切割功能。

二、表格內部邊線辨識

[4] 指出可透過形態學運算進行表格辨識；而 [1] 與 [2] 則是採用 Canny 邊緣檢測函數，並結合 HoughLinesP 霍夫直線變換進行偵測。經實測比較後發現，利用形態學運算所需調整的參數較少，前處理需求也更為簡化。

由於本專案處理的影像並非真實拍攝結果，而是不含掃描噪點的數位圖片，因此形態學運算在此應用情境下更為合適。

（一）前製處理

```
let mut gray_inv = Mat::default();
bitwise_not(&gray, &mut gray_inv, &no_array());
let mut binary = Mat::default();
threshold(&gray_inv, &mut binary, 90.0, 255.0, THRESH_BINARY)?;
```

程式 4 gray 為將圖像灰白化後的圖片

因為我們要偵測的是邊線，所以將圖像負片化能使邊線轉換成白色，較易偵測。

```
let dilate_kernel = get_structuring_element(MORPH_RECT, Size::new(3, 1),
Point::new(-1, -1))?;
let mut binary_dilated = Mat::default();
dilate(
    &binary,
    &mut binary_dilated,
    &dilate_kernel,
    Point::new(-1, -1),
    1,
    BORDER_CONSTANT,
    morphology_default_border_value()?,
)?;
```

程式 5

再先進行簡單的膨脹運算，將文字變得更粗，如 程式 5 可見。

撰寫歷史

原本的流程直接進入橫線偵測，沒有 dilate 這個步驟，但由於辨識時發現有部分區域偵測出的邊緣有小瑕疵，導致後來切割單元格的時會裁到邊線，影響文字辨識。

經過繁複 debug 過後，在二值化後增加簡單的膨脹運算，能讓邊線更完整，提高辨識穩定性。

(二) 偵測橫線

```
let horizontal_kernel =  
    get_structuring_element(MORPH_RECT, Size::new(kernel_width, 1),  
    Point::new(-1, -1))?;  
  
let mut detected = Mat::default();  
  
morphology_ex(  
    &binary,  
    &mut detected,  
    MORPH_OPEN,  
    &horizontal_kernel,  
    Point::new(-1, -1),  
    1,  
    BORDER_CONSTANT,  
    morphology_default_border_value()?,  
)?;
```

程式 6 `binary` 圖片是經過 程式 4 和 程式 5 處理過後的圖像

透過使用極扁的 `kernel` 並施行開運算 (MORPH_OPEN)，可將非橫向排列的區域轉換為背景。因此，調整 程式 6 中的 `kernel_width` 會影響處理結果。

經測試，圖 11 與 圖 12 仍可觀察到未完全腐蝕的文字，最終決定將 `kernel_width` 設為 60，以確保文字完全被去除。

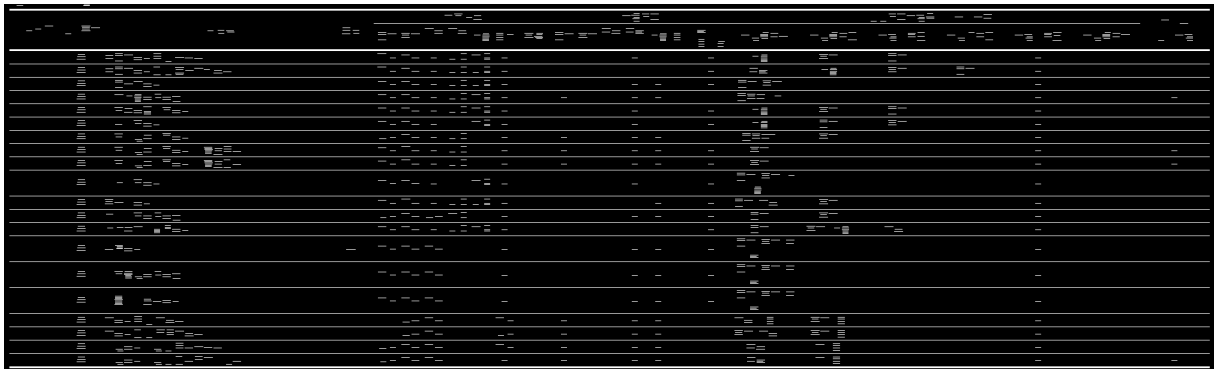


圖 11 `kernel_width = 10`



圖 12 `kernel_width = 20`

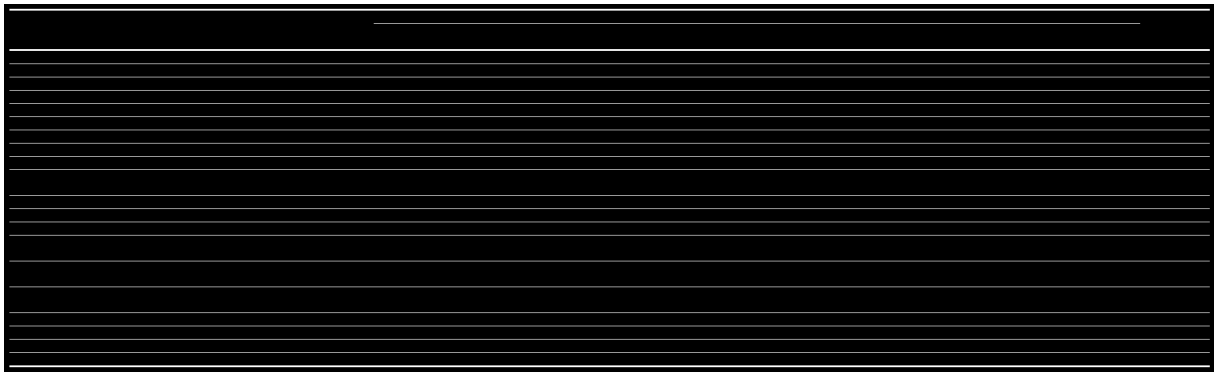


圖 13 `kernel_width = 40`

(三) 偵測直線

同樣的邏輯可以套用在直線上，而 `kernel_height` 這次設定為 40，因為有較多更短的直線。

```
let vertical_kernel =  
  get_structuring_element(MORPH_RECT, Size::new(1, kernel_height),  
    Point::new(-1, -1));
```

程式 7

(四) 統一線寬

由於 `.data_bytes_mut()` 必須取得一塊連續記憶體，因此原本在 OpenCV 中不是連續記憶體的列(column)，無法直接使用該方法進行操作。

為了統一處理線寬，可利用矩陣轉置，將原本的列(column)轉換為行(row)，使其變成連續的記憶體區塊，即可直接傳入 `keep_middle_line`。

關於矩陣轉置策略：

最初我使用 `.at(x,y)` 逐元素將列(column)轉換成 `Vec` 進行處理。這個方法可以運作，但有兩個問題：

1. 只能線性執行，無法平行化
2. 程式碼不直觀

而在重構時想到：既然列不是連續的，那把整個矩陣轉置不就行了，把原本的列變成行（連續記憶體），就能可直接操作。

關於為何採用 `keep_last_line`：

目前採用保留最後一行的方案，但這並非一定是最佳解答。在部分測試案例中，保留第一行或中間那行的效果更好。由於本專案重點在於驗證整體技術可行性，我選擇使用較穩定的「保留最後一行」策略。

```
// transpose to make columns continous
transpose(&h_lines.clone(), &mut h_lines)?;
for y in 0..h_lines.rows() {
    keep_last_line(h_lines.row_mut(y)?.data_bytes_mut()?);
}
transpose(&h_lines.clone(), &mut h_lines)?;

for y in 0..v_lines.rows() {
    keep_last_line(v_lines.row_mut(y)?.data_bytes_mut()?);
}

fn keep_last_line(line: &mut [u8]) {
    let line_with_index =
line.iter().copied().enumerate().collect::<Vec<_>>();

    // if consectuive elements are the all 255, group them together
    let consecutive_runs = line_with_index
        .chunk_by(|(_, lhs), (_, rhs)| *lhs == 255 && *rhs == 255)
        // prevents boundary condition of [255, 0]
        .filter(|x| !x.iter().rev().any(|x| x.1 == 0));
    line.fill(0);
    for stride in consecutive_runs {
        line[stride[stride.len() - 1].0] = 255;
    }
}
```

程式 8

(五) 相交點計算

根據 程式 9，可利用 `bitwise_and` 計算線條的交點。然而，交點結果儲存在 `Mat` 影像中，不易直接運用。因此，使用 `find_non_zero` 將非零像素（即交點）轉換為 `Vec<Point>`，以便後續處理。

布林運算的好用之處

最初用雙層 for 迴圈逐步檢查像素交點，這種 imperative 的寫法雖然簡單但巢狀層級過深，回頭閱讀時會增加認知複雜度。重構時想到：既然有 `bitwise_and`，何不直接計算交點？然後再利用 OpenCV 所提供的 `iter()` 函數，即可在無 nesting 的方式，撰寫出效果相同，但更直觀、更好理解的程式碼。

這次重構利用了「Never Nester」原則²：透過提前返回、Declarative Coding 等概念和 API 的善用，將多層巢狀迴圈轉換為清晰的 pipeline，降低閱讀程式碼時的認知負荷。

²[Why You Shouldn't Nest Your Code](#) by CodeAesthetic

```

let mut intersection_mat = Mat::default();
bitwise_and(&v_lines, &h_lines, &mut intersection_mat, &no_array());?;

let mut points_mat = Mat::default();
find_non_zero(intersections_mat, &mut points_mat)?;

let mut intersections = points_mat
    .iter::<Point>()
    .into_iter()
    .flatten()
    .map(|e| (e.1.x, e.1.y))
    .collect::<Vec<_>>();

```

程式 9

三、單元格分割

大部分文獻所提供的分割方法都是存在表格為「整齊」的前提下撰寫的，而如此的切割方法(程式 10)假如遇到如圖 14 或圖 15 的情況就會無法將完整的單元格切割出來。

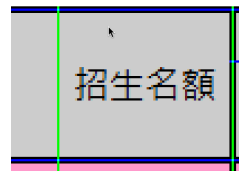


圖 14

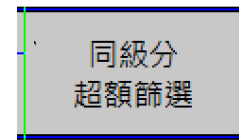


圖 15

(一) 原始演算法

程式 10 對左邊有三個交點的單元格(圖 15)有效，但對右邊有三個交點的單元格(圖 14)卻會失效。

```

intersections.sort_unstable();

// 使得每個窗口都必須是同一列(column)
let iter = intersections.windows(2).filter(|w| w[0].0 == w[1].0);

for window in iter {
    let [top_left, bottom_left] = window else {
        unreachable!()
    };

    let bottom_right = (bottom_left.0 + 1..max_x)
        .find(|e| table.contains(&(*e, bottom_left.1)))
        .map(|x| (x, bottom_left.1));

    let cell = Mat::roi(gray, Rect::from_points(top_left, bottom_right))?;
}

```

原始演算法簡介
採用逐列掃描（滑動窗口，每次取 2 個元素）：取左上和左下兩點，從左下向右搜尋右下角。

程式 10 Naive implementation

解決這個問題最「標準」的方法是判別目前的格子到底是在圖 15 還是圖 14 的情況，但我有想到一個能避免複雜判斷的方式。

(二) 最終演算法：

1. 利用滑動窗口(程式 10)得知左上、左下兩點
2. 向右搜尋右上、右下兩點 (程式 11)
3. 形成兩個候選矩形並比較面積 (程式 12)
4. 取較大者作為單元格邊界

```
let top_right = (top_left.0 + 1..max_x)
    .find(|e| table.contains(&(*e, top_left.1)))
    .map(|x| (x, top_left.1));

let bottom_right = (bottom_left.0 + 1..max_x)
    .find(|e| table.contains(&(*e, bottom_left.1)))
    .map(|x| (x, bottom_left.1));
```

程式 11

```
fn get_biggest_possible_bounding_box(
    [top_left, top_right, bottom_left, bottom_right]: [Point_<i32>; 4],
) -> Rect_<i32> {
    let rect1 = Rect::from_points(
        (bottom_left.x, bottom_left.y).into(),
        (top_right.x, top_right.y).into(),
    );

    let rect2 = Rect::from_points(
        (top_left.x, top_left.y).into(),
        (bottom_right.x, bottom_right.y).into(),
    );

    if rect1 > rect2 {
        rect1
    } else {
        rect2
    }
}
```

程式 12

在**視覺化影片**中，可以看到在招生名額(0:09)、檢定標準(0:12)等單元格都正確判斷。

文字辨識：

在經 `Mat::roi` 獲得單元格，輸出成圖片檔給 PaddleOCR 做辨識後即可獲取到中文文字，至於何時會使用到 Tesseract 則會在 小節 四 時詳細描述。

四、 資料結構化

伍、 我的收穫與反思

參考文獻

- [1] 蔡桓銘, 「用 Tesseract 結合 LSTM 模型實作手填表格辨識」, 2021. doi: [10.6846/TKU.2021.00596](https://doi.org/10.6846/TKU.2021.00596).
- [2] Johnny Chang, 「使用 python 萃取掃描文件中的表格(一)切豆腐篇」. [線上]. 載於: [https://between2058.medium.com/使用 python 萃取掃描文件中的表格——切豆腐篇-d5b65b7ec320](https://between2058.medium.com/使用-python-萃取掃描文件中的表格-一切豆腐篇-d5b65b7ec320)
- [3] Willus Dotkom, 「Optimal image resolution (dpi/ppi) for Tesseract 4.0.0 and eng.traineddata?」. [線上]. 載於: https://groups.google.com/g/tesseract-ocr/c/Wdh_JJwnw94/m/24JHDYQbBQAJ?pli=1
- [4] S. Mandal, S. P. Chowdhury, 及 A. K. Das & Bhabatosh Chanda, 「A simple and effective table detection system from document images」, 2006 年 3 月 24 日. doi: [10.1007/s10032-005-0006-5](https://doi.org/10.1007/s10032-005-0006-5).