Project No: A1071-241


# STREAMLINING TIMETABLE & STUDENT DATA MANAGEMENT

**Submitted by: Sean Young Song Jie**

**Supervisor: A/P Ling Keck Voon**

School of Electrical & Electronic Engineering

A final year project report presented to Nanyang Technological University
in partial fulfilment of the requirements for the
Degree of Bachelor of Engineering

2025

# Table of Contents

## Abstract

This project presents an end-to-end system for converting university teaching timetable PDFs into structured calendar files, addressing a common inefficiency faced by academic staff. At Nanyang Technological University (NTU), faculty teaching schedules are distributed as multi-week PDF tables, which are not directly compatible with digital calendars like Google Calendar or Microsoft Outlook. Manual entry is time-consuming and error-prone, especially when changes occur prior to semester start.

To automate this process, the system combines deep learning-based layout detection with optical character recognition (OCR), spatial grid mapping, semantic parsing, and calendar file generation. A custom-trained YOLOv8 model is used to detect timetable structures such as time slots, week columns, and course cells. Extracted text is clustered and interpreted to recover course information including week ranges, time windows, locations, and optional holiday overrides. A web-based interface built using Streamlit allows users to review and edit extracted entries before exporting them as .ics files compatible with major calendar platforms.

The system achieves an extraction accuracy of approximately 80–85% on NTU EEE timetables and supports correction through an intuitive user interface. By streamlining this conversion workflow, the project offers a practical and extensible tool that reduces manual workload and enables timely calendar synchronization for academic staff.

**Acronyms**

| | |
|---|---|
| NTU | Nanyang Technological University |
| PDF | Portable Document Format |
| EEE | Electrical and Electronic Engineering |
| FYP | Final Year Project |
| OCR | Optical Character Recognition |
| DBSCAN | Density-Based Spatial Clustering of Applications with Noise |
| JSON | JavaScript Object Notation |
| YOLOv5/v8 | You Only Look Once version 5/8 |
| ICS | Internet Calendaring and Scheduling |
| R-CNN | Region-based Convolutional Neural Network |
| ICDAR | International Conference on Document Analysis and Recognition |
| DPI | Dots per inch |
| PIL | Python Image Library |
| ASCII | American Standard Code for Information Interchange |

# List of Figures

# List of Tables

# Chapter 1 Introduction

Timetable distribution and management are essential in academic institutions. At universities like Nanyang Technological University (NTU), teaching schedules are typically provided to faculty members in the form of Portable Document Format (PDF) documents. These PDFs often contain multi-week timetables structured in a tabular format. However, this format is not directly compatible with calendar applications such as Google Calendar or Microsoft Outlook, which require event-based inputs, usually via .ics files.

Several third-party tools such as DocHub, pdfFiller, and Aspose.Total offer basic PDF-to-calendar conversion features [1][2][3]. These tools, however, are not designed for institutional timetables and lack support for layout-specific interpretation, stacked course entries, or academic-specific annotations such as holidays. As such, they are not suitable for parsing structured timetable PDFs used by NTU's School of Electrical and Electronic Engineering (EEE).

## 1.1 Motivations

Faculty members at NTU, including course instructors and project supervisors, receive their teaching and consultation timetables in PDF format, typically structured as multi-week tables. While these documents are official and complete, detailing course codes, time slots, venues, and weekly recurrence, they cannot be imported directly into calendar applications such as Google Calendar or Microsoft Outlook. As a result, professors often need to either refer back to these static documents repeatedly or manually recreate their schedules in personal calendar platforms.

This workflow is tedious, repetitive, and inefficient. The information is already known, yet faculty members are burdened with transcribing it into digital calendars just to manage their day-to-day schedules. This project was directly motivated by a pain point raised by my Final Year Project (FYP) supervisor, who expressed frustration with the manual effort required to input teaching schedules every semester.

While this conversion task may seem one-off, changes to the timetable can still occur before the semester begins. These revisions may arise from departmental reshuffling or last-minute adjustments, such as taking over another professor's class, shifting teaching groups, or rescheduling due to conflicts. In such cases, the entire manual conversion must be repeated with the updated version of the timetable. Automating this process not only simplifies the initial entry but also allows users to re-export corrected .ics files efficiently when such updates occur.

### 1.2 Objectives and Scope

The objective of this project is to develop a system that automates the conversion of NTU EEE teaching timetables that are originally distributed as PDF documents, into .ics calendar files that can be directly imported into widely used calendar applications such as Google Calendar, Microsoft Outlook, and Apple Calendar. This system aims to address a recurring inefficiency faced by academic staff, who must manually transfer their semester schedules into digital calendars due to the incompatibility between PDF formats and calendar platforms.

To achieve this goal, the system integrates layout detection, optical character recognition (OCR), and semantic post-processing to extract structured timetable information from PDF-based teaching load documents. It identifies course codes, group identifiers, venues, day and time slots, and week ranges, and converts them into a structured event-based format suitable for calendar integration. Additional logic is included to handle academic-specific patterns such as stacked course blocks and annotated holiday entries like Deepavali.

The scope of the project is limited to teaching load timetables issued by the School of EEE at NTU. It specifically targets PDFs with consistent, structured layouts intended for machine reading. Timetables from other faculties or with irregular formats are not supported in this version. The system supports weekly recurrence as defined by explicit week numbers listed in the timetable but does not handle complex recurrence rules such

as alternating weeks or conditional scheduling logic if there is such case. The focus remains on delivering a reliable, user-controllable pipeline for extraction, correction, and calendar export tailored to EEE faculty needs.

### 1.3 Work Done

This project involved the development of an end-to-end system to extract structured academic timetable information from PDF documents and convert it into .ics calendar files. Development followed an iterative process, beginning with exploratory methods, followed by a working extraction pipeline, and culminating in the integration of deep learning-based layout detection and a user interface.

During the initial phase of the project, multiple extraction strategies were explored. These included line-by-line text extraction using pdfplumber, followed by parsing with regular expressions as shown in Fig. 1.1 and Fig. 1.2. A structured approach based on identifying weekday columns and scanning associated rows was also tested with results shown in Fig. 1.3. Another attempt combined OpenCV region detection using colour cues from the timetable legend with OCR via Tesseract. You can compare the colours extracted with the legend in Fig. 1.4 and Fig. 1.5 and you can tell it was in random order which made it practically unusable. While these methods offered partial results, none were sufficiently robust or generalisable.

```
Page 0 Table Extracted:
  Monday_Page0          None_Page0          None_Page0  \
0         Week                   1                   2
1  Time \\nDate  08 Aug 22\n12 Aug 22  15 Aug 22\n19 Aug 22
2   0830-\n0900
3   0900-\n0930
4   0930-\n1000

          None_Page0          None_Page0          None_Page0  \
0                  3                   4                   5
1  22 Aug 22\n26 Aug 22  29 Aug 22\n02 Sep 22  05 Sep 22\n09 Sep
22
2
3
4

          None_Page0          None_Page0          None_Page0  \
0                  6                   7
1  12 Sep 22\n16 Sep 22  19 Sep 22\n23 Sep 22  26 Sep 22\n30 Sep 22
2
3
4

  None_Page0 None_Page0          None_Page0          None_Page0  \
0    Recess                          8                   9
1      None       None  03 Oct 22\n07 Oct 22  10 Oct 22\n14 Oct 22
2      None       None
```

Figure 1.1: Sample of using pdfplumber

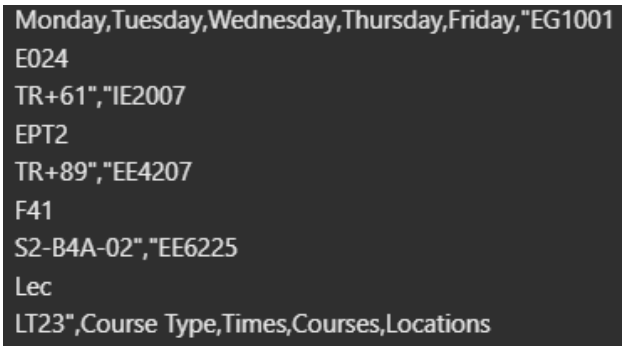



Figure 1.2: Sample of using regex

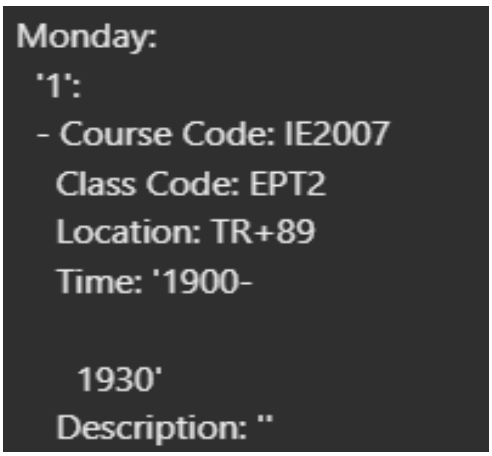



Figure 1.3: Sample of using structured approach with pdfplumber

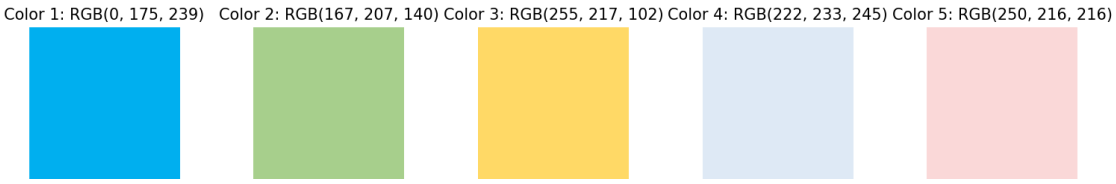



Figure 1.4: Colours extracted from OCR

| Course Type | Background Color |
|---|---|
| Lecture | |
| Tutorial | |
| Lab | |
| Part-time | |
| Design | |

Figure 1.5: Legends in timetable PDF

Next, the project explored layout detection models using a custom-labelled dataset created with LabelImg. Five distinct region types (day, timeSlots, weeks, date, courseCells) were annotated for training object detectors as shown in Fig. 1.6. YOLOv5 was first tested, and the results is shown in Fig. 1.7 that it produced unreliable results, with low-confidence detections and bounding box inconsistencies. CascadeTabNet was considered, but its heavy dependencies made it unsuitable for a lightweight application. DeepDeSRT, though frequently cited, lacked any available public implementation.

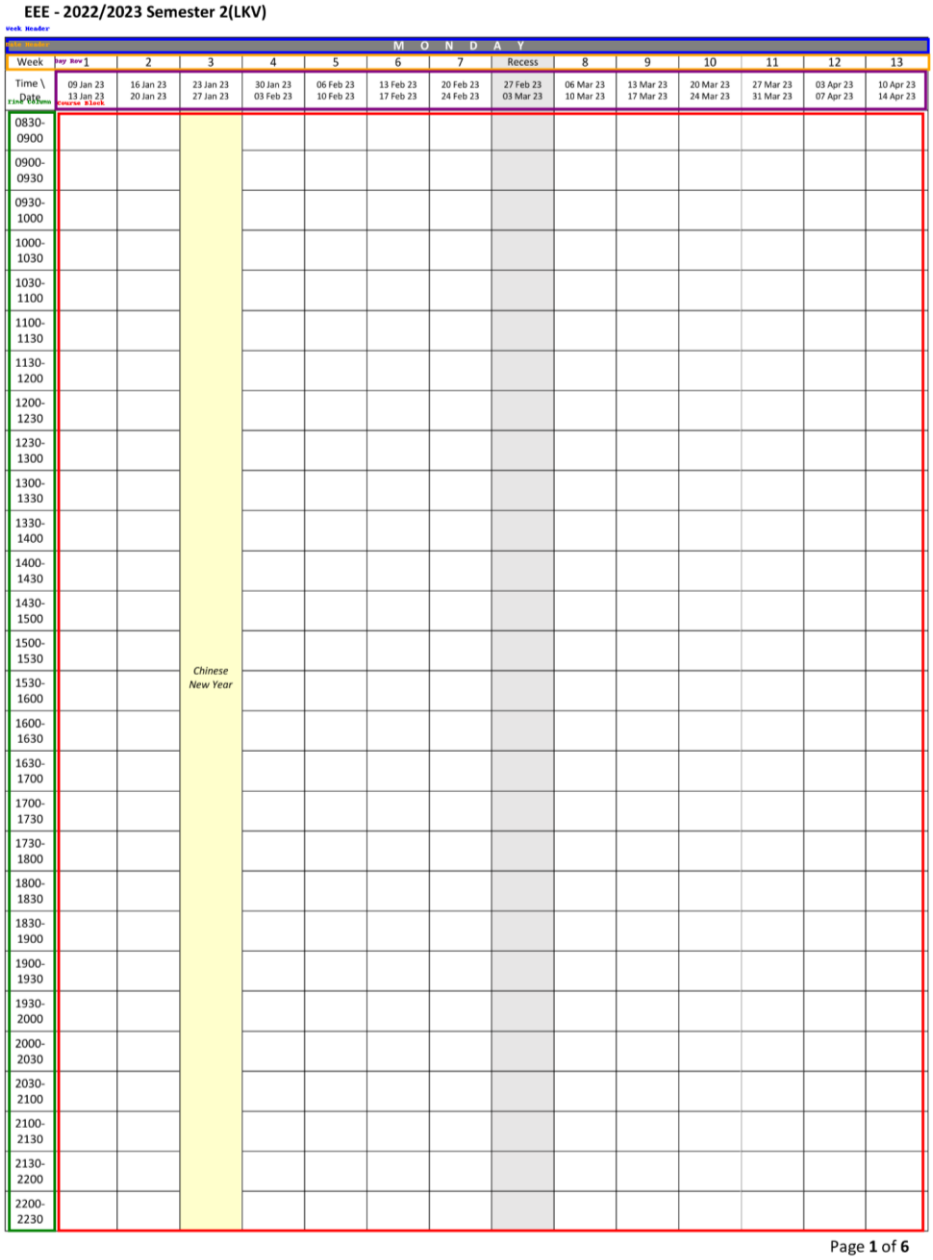



Figure 1.6: Five distinct regions annotated with LabelImg

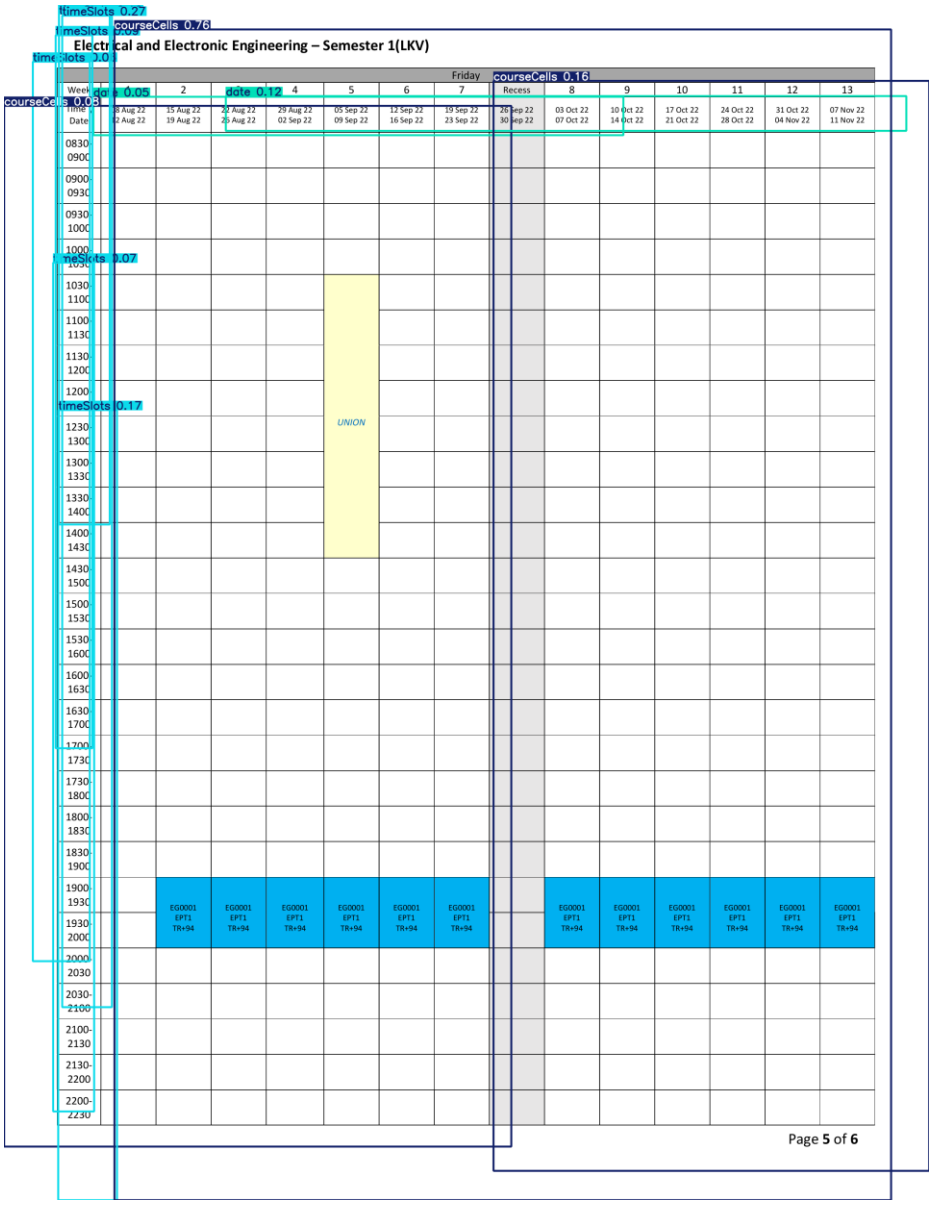



Figure 1.7: YOLOv5 results

With model-based detection on hold, focus shifted to implementing the core extraction pipeline. The previously annotated region definitions were used as a reference layout. OCR was performed using Tesseract within each region, followed by DBSCAN-based text clustering to isolate course blocks. Post-processing logic was developed to extract structured fields including course code, group, location, week range, and time. Fuzzy

matching was used for holiday detection, and date logic was added to align week numbers with actual calendar dates. Fig. 1.8 shows that each course block is formatted into JavaScript Object Notation (JSON) at the end of the pipeline. This pipeline became the foundation of the system and worked reliably across test timetables when the region layout matched expectations.

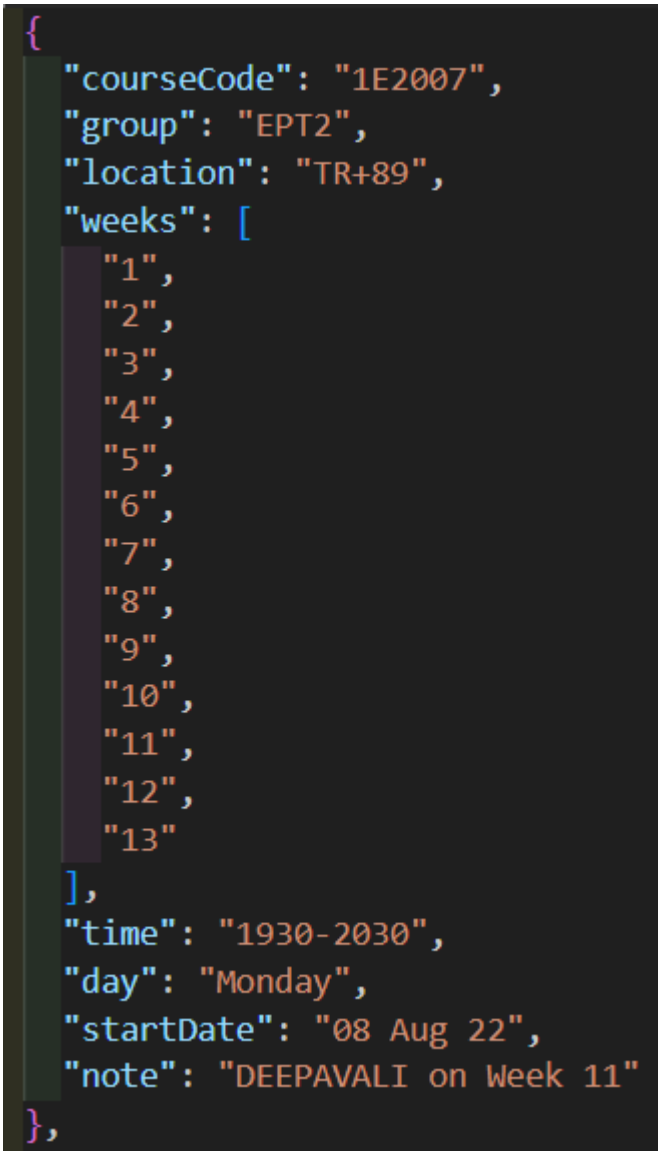



Figure 1.8: Pipeline output in JSON format

YOLOv8 was later introduced to automate region detection. Initial training on five

region classes using the yolov8n model produced overlapping and noisy bounding boxes. Improvements were made by reducing the number of classes to three (timeSlots, weeks, courseCells), switching to the yolov8s model, and increasing training epochs to 200. The results are shown in Fig. 1.9. Additional post-processing logic was added to refine the bounding box outputs as shown in Fig. 1.10. Once the model reached acceptable accuracy, it was integrated into the pipeline in place of static region references. The logic was updated to support the new three-class layout, resulting in an extraction accuracy of approximately 80–85%.

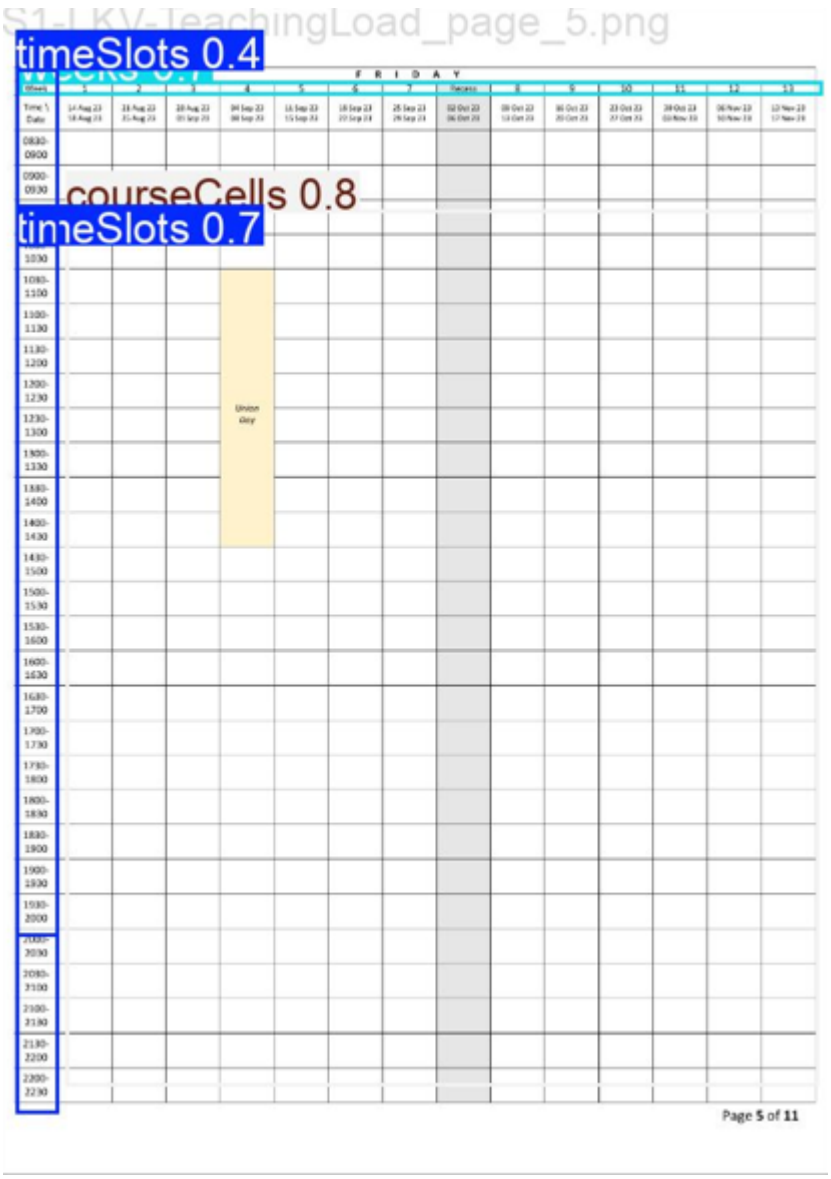
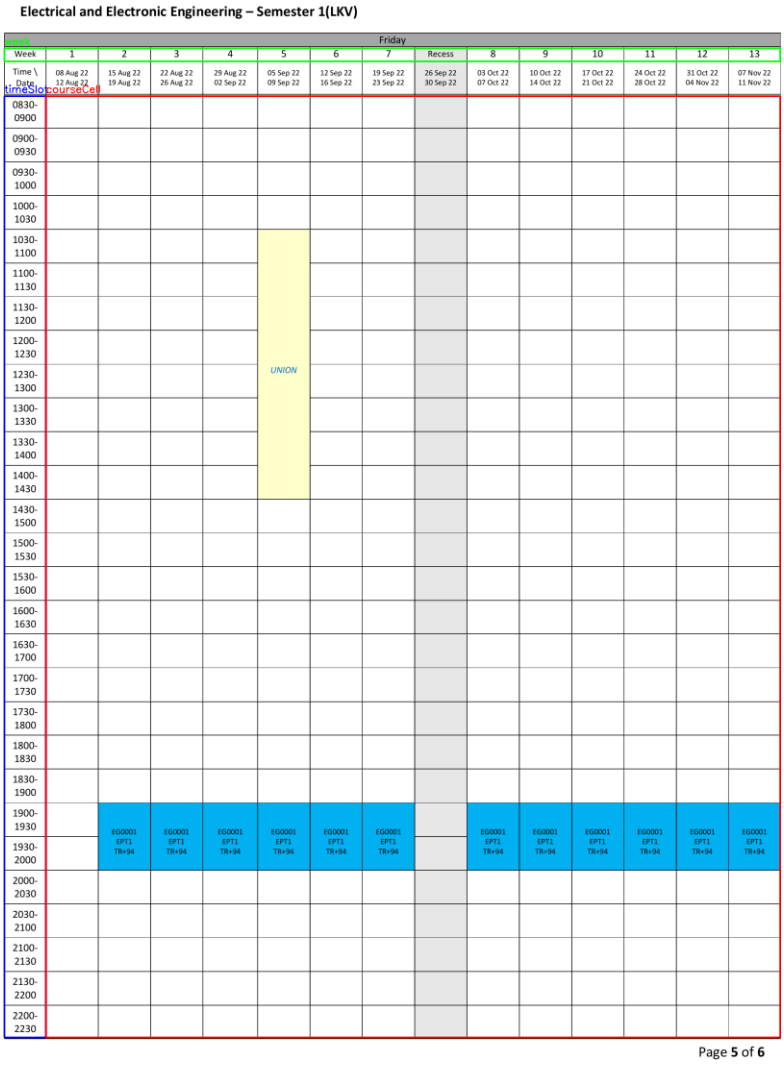



Figure 1.9: YOLOv8 results



**Electrical and Electronic Engineering – Semester 1(LKV)**

| | Friday | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Week | 1 | 2 | 3 | 4 | 5 | 6 | 7 | Recess | 8 | 9 | 10 | 11 | 12 | 13 |
| Time \ Date | 08 Aug 22<br>12 Aug 22 | 15 Aug 22<br>19 Aug 22 | 22 Aug 22<br>26 Aug 22 | 29 Aug 22<br>02 Sep 22 | 05 Sep 22<br>09 Sep 22 | 12 Sep 22<br>16 Sep 22 | 19 Sep 22<br>23 Sep 22 | 26 Sep 22<br>30 Sep 22 | 03 Oct 22<br>07 Oct 22 | 10 Oct 22<br>14 Oct 22 | 17 Oct 22<br>21 Oct 22 | 24 Oct 22<br>28 Oct 22 | 31 Oct 22<br>04 Nov 22 | 07 Nov 22<br>11 Nov 22 |
| 0830-0900 | | | | | | | | | | | | | | |
| 0900-0930 | | | | | | | | | | | | | | |
| 0930-1000 | | | | | | | | | | | | | | |
| 1000-1030 | | | | | | | | | | | | | | |
| 1030-1100 | | | | | UNION | | | | | | | | | |
| 1100-1130 | | | | | | | | | | | | | | |
| 1130-1200 | | | | | | | | | | | | | | |
| 1200-1230 | | | | | | | | | | | | | | |
| 1230-1300 | | | | | | | | | | | | | | |
| 1300-1330 | | | | | | | | | | | | | | |
| 1330-1400 | | | | | | | | | | | | | | |
| 1400-1430 | | | | | | | | | | | | | | |
| 1430-1500 | | | | | | | | | | | | | | |
| 1500-1530 | | | | | | | | | | | | | | |
| 1530-1600 | | | | | | | | | | | | | | |
| 1600-1630 | | | | | | | | | | | | | | |
| 1630-1700 | | | | | | | | | | | | | | |
| 1700-1730 | | | | | | | | | | | | | | |
| 1730-1800 | | | | | | | | | | | | | | |
| 1800-1830 | | | | | | | | | | | | | | |
| 1830-1900 | | | | | | | | | | | | | | |
| 1900-1930 | | EG0001 EPT1 TR+94 | EG0001 EPT1 TR+94 | EG0001 EPT1 TR+94 | EG0001 EPT1 TR+94 | EG0001 EPT1 TR+94 | EG0001 EPT1 TR+94 | | EG0001 EPT1 TR+94 | EG0001 EPT1 TR+94 | EG0001 EPT1 TR+94 | EG0001 EPT1 TR+94 | EG0001 EPT1 TR+94 | EG0001 EPT1 TR+94 |
| 1930-2000 | | | | | | | | | | | | | | |
| 2000-2030 | | | | | | | | | | | | | | |
| 2030-2100 | | | | | | | | | | | | | | |
| 2100-2130 | | | | | | | | | | | | | | |
| 2130-2200 | | | | | | | | | | | | | | |
| 2200-2230 | | | | | | | | | | | | | | |

Page **5** of **6**

Figure 1.10: After post-processing logic

A Streamlit-based interface was developed to enable users to upload timetables, visualise extracted data, make manual corrections, and export results in .csv format. This interface ensures usability and flexibility, particularly in scenarios where timetable changes or errors need to be reviewed before calendar integration. The user interface is shown in Fig. 1.11.


Figure 1.11: Streamlit User Interface

### 1.4 Organisation

The rest of this report is structured as follows. Chapter 2 presents a literature survey of related works and tools for timetable parsing, layout detection, OCR, and calendar export. Chapter 3 outlines the overall system architecture and describes the extraction pipeline, including OCR, layout reconstruction, grid mapping, semantic parsing of course data, and JSON formatting. Chapter 4 details the YOLOv8 layout detection model, covering its class definitions, training process, and post-processing for bounding box refinement. Chapter 5 introduces the Streamlit-based user interface, which allows users to review and edit extracted timetable entries and explains how .ics calendar export functionality was implemented. Finally, Chapter 6 concludes the report with a summary of project outcomes and potential directions for future work.

# Chapter 2 Literature Review

## 2.1 Existing Tools for Timetable-to-Calendar Conversion

Several tools aim to convert PDF documents into calendar formats, yet they often fall short when handling complex academic timetables. For instance, DocHub, pdfFiller, and Aspose.Total offer features to convert PDFs to ICS files, facilitating calendar integration. However, these tools primarily cater to straightforward documents and lack the sophistication to interpret intricate timetable structures with overlapping sessions or varying formats [1] – [3].

In response to these limitations, an independent developer created PDFtoCal, a tool designed to convert PDF schedules into calendar events. Despite its targeted approach, PDFtoCal struggles with unique timetable formats like those used by NTU EEE, which feature stacked cells and non-standard layouts, leading to inaccurate data extraction [4].

Moreover, user experiences shared on platforms like Reddit highlight the challenges faced when attempting to import PDF-based schedules into calendar applications. One user noted the inefficiency of manually converting PDFs to CSVs for calendar import, emphasizing the need for a more streamlined solution [5].

These examples underscore the limitations of existing tools in handling the nuanced requirements of academic timetable conversions.

## 2.2 Table and Layout Detection in PDFs

Accurate detection of tables and layouts within PDFs is crucial for extracting structured information. Several deep learning-based models were considered for this task.

YOLOv5 was initially tested using a custom-labelled dataset annotated via LabelImg, covering five region types including time, weeks, and course cells. However, YOLOv5 produced unreliable results, with low-confidence detections and bounding box

inconsistencies, making it unsuitable for consistent layout parsing.

CascadeTabNet, introduced by Prasad et al., employs a Cascade Mask R-CNN architecture to detect tables and recognise their structures in image-based documents. It has demonstrated high accuracy on datasets like ICDAR 2013 and TableBank [6]. Despite its capabilities, CascadeTabNet comes with heavy dependencies that made it impractical to integrate into a lightweight, deployable application for this project.

DeepDeSRT, developed by Schreiber et al., utilizes deep learning for table detection and structure recognition. It has achieved high detection accuracy on standard datasets [7]. However, DeepDeSRT lacks an available public implementation, which prevented practical experimentation and integration within the project timeline.

These findings led to a decision to focus on YOLOv8, which provided more stable bounding box outputs and supported lightweight integration into the application pipeline. YOLOv8 introduces architectural enhancements over YOLOv5, such as an anchor-free detection head and a refined Concatenate-to-Fuse (C2f) neck, contributing to improved accuracy and speed [8].

### 2.3 OCR in Semi-Structured Documents

Timetable PDFs issued by universities like NTU often contain structured data embedded within grid-like layouts but lack machine-readable metadata. Extracting this information relies heavily on OCR, especially in segments like course blocks and headers where font styles, colour overlays, and alignment vary.

Tesseract OCR was selected for this project due to its ease of integration and open-source nature. Despite this, several challenges arose: Text in course cells often contains line breaks and vertical stacking (e.g., course code, group, venue), fonts may be coloured or shaded due to timetable legend cues, some entries contain split tokens or misshaped glyphs that require post-cleaning.

To address these, preprocessing techniques such as HSV masking and adaptive thresholding were employed. These helps neutralize colour interference (e.g., cyan highlighting) and sharpen text edges, improving OCR fidelity. Following extraction, lines are grouped using DBSCAN clustering on the vertical y-coordinates to separate stacked entries, a technique that has shown success in previous layout parsing efforts [9].

OCR errors are further handled with fuzzy matching logic, especially when identifying known holiday labels like "Deepavali" or "Union Day", which may be spelled inconsistently due to OCR noise.

### 2.4 Temporal and Semantic Post-Processing

Raw OCR and layout data alone are insufficient to produce calendar-ready outputs. Academic timetables use implicit rules like weekly recurrence, stacked class blocks, and holiday overrides. The system therefore implements a semantic layer that enriches and validates extracted content.

Each course cell is assumed to follow a 3-line vertical structure (Course Code, Group, Location). The system identifies holidays by checking if the first OCR line matches known public holidays using fuzzy string similarity. If a match is detected, the block is interpreted as a 4-line entry, where the first line is stored as a note (e.g., “Deepavali on Week 13”), and the remaining lines are parsed as a regular course entry.

To reconstruct event durations, bounding box coordinates (y1, y2) are matched against the time slot grid. Overlapping rows are used to compute a start–end time window (e.g., "1000–1130"). Week numbers are inferred by comparing the horizontal span of the course cell with week columns, while base dates are extracted from the header region below the “Week” label using OCR and fuzzy date parsing [10].

This process enables the output to support .ics event recurrence, structured per course,

time, and week.

### 2.5 Related Work in Academic Scheduling and Structured Timetable Extraction

Most existing academic scheduling platforms, such as UniTime and Celcat, are built to facilitate the generation of course schedules, where event times and venues are algorithmically assigned based on constraints like room availability and staff allocation. However, these systems do not address the inverse problem: interpreting and extracting structured data from pre-existing timetable documents. In many institutions, including NTU, finalised teaching timetables are distributed as static PDF documents. Although digital, these files are visually formatted for human readability rather than machine interpretability, and they lack embedded metadata necessary for calendar integration.

Academic timetable PDFs often feature stacked course entries, vertically arranged text, merged cells, and annotations for holidays or recess weeks, making direct parsing challenging. Unlike domains such as invoice processing or scientific article layout analysis—where large, annotated datasets and benchmark models exist—academic timetable parsing remains underexplored. While research in document layout analysis has advanced significantly, particularly through projects like PubLayNet, these efforts have focused on form-like or publication-based documents [11]. Timetables differ due to their grid-based nature, temporal semantics, and domain-specific structure.

The system developed in this project addresses this gap by combining layout detection using YOLOv8 with OCR and clustering-based line interpretation, followed by a semantic post-processing layer designed to interpret NTU EEE’s specific formatting conventions as shown in Fig. 2.1. This end-to-end approach enables the extraction of calendar-ready structured data from complex timetable layouts, representing a novel contribution to the field of educational document automation.

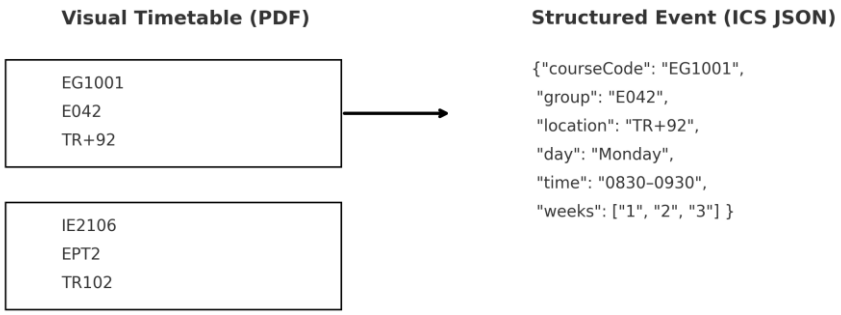



Figure 2.1: visual vs. semantic structure

# Chapter 3 System Design

## 3.1 Overview of Architecture

The timetable-to-calendar conversion system is implemented as a modular pipeline that processes teaching timetable PDFs from NTU's School of EEE and converts them into structured .ics files compatible with Google Calendar, Outlook, and other calendar platforms. Fig. 3.1 shows the entire pipeline of the system.

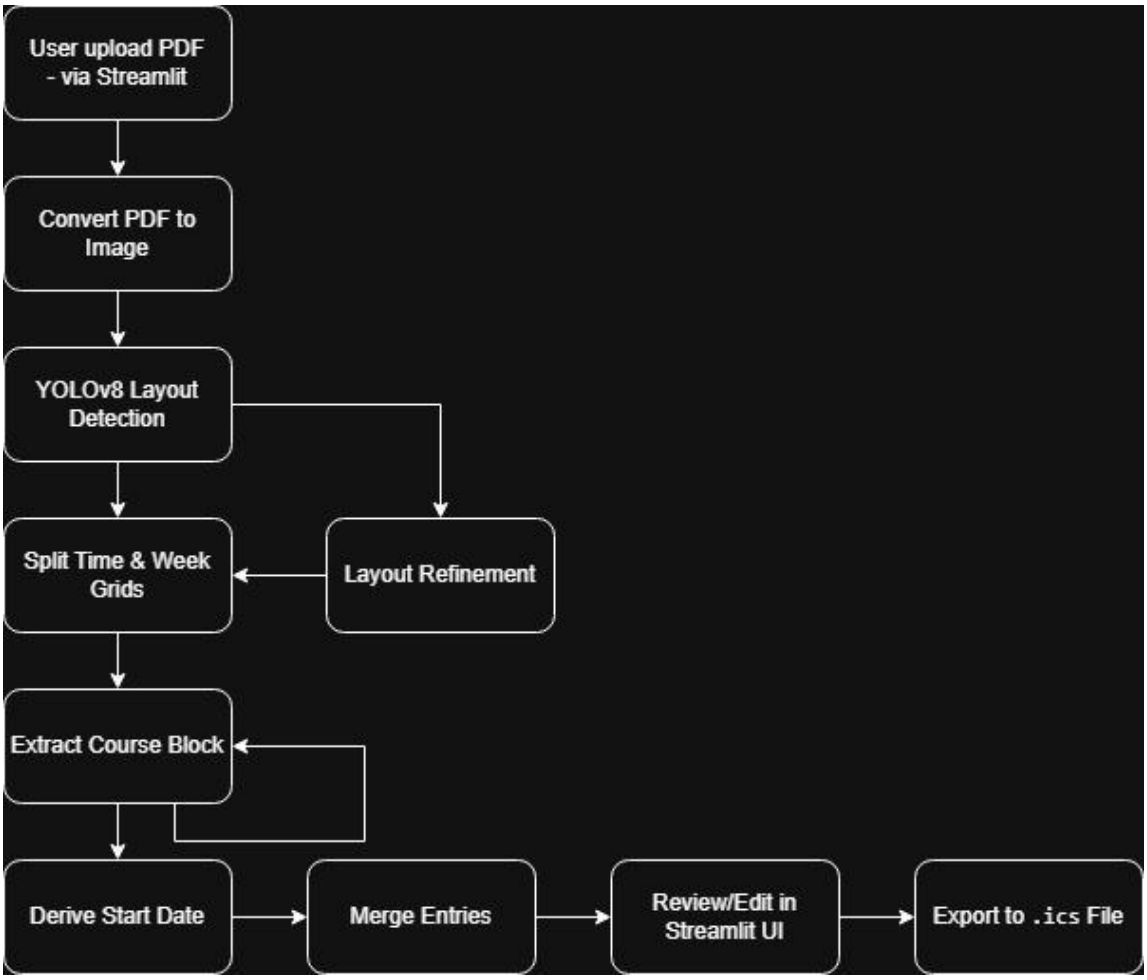



Figure 3.1: System Architecture - Timetable to Calendar conversion pipeline

The system comprises three major components:

**Layout Detection** detects structural regions of the timetable (e.g., time slots, week

columns, course blocks) using a custom-trained YOLOv8 object detector.

**Text Extraction and Clustering** applies OCR to extract raw text within each detected region and clusters vertically stacked lines to reconstruct course information.

**Semantic Parsing and Calendar Formatting** maps course data to weeks and time ranges, detects holidays, formats into JSON, and generates calendar events.

The pipeline is designed for high adaptability across multiple semester PDFs with consistent layouts. It supports user intervention via a Streamlit interface, allowing course block editing and manual validation before final export.

**Technology Stack**

**YOLOv8**: Region detection (layout structure)
**Tesseract OCR**: Text extraction
**DBSCAN**: Clustering vertically stacked text
**RapidFuzz**: Fuzzy matching for holiday detection
**Streamlit**: User interface for review/export
**Python**: Core logic, image processing, and calendar formatting

### 3.2 Input Format and Constraints

The system is tailored to process NTU EEE's official teaching load timetable PDFs, which follow a consistent tabular structure designed for human readability.

**Expected input Format**

File type: PDF
Layout:

1. Each page = one weekday (Monday–Friday)
2. 15 columns: "Week", 1–7, "Recess", 8–13
3. 28 rows: 30-minute slots from 0830–2230

4. Course cells are vertically stacked (up to 3–4 lines per cell)
5. Some weeks may include holiday notes like “Deepavali”

**Processing Constraints**

PDFs must be:

1. A4 size, processed at 300 DPI
2. Unrotated and properly aligned
3. Coloured with distinguishable highlights (legend-based)

Timetables outside this structure (e.g., different faculties, scanned images, inconsistent column counts) are considered unsupported in the current version.

### 3.3 Layout Detection Using YOLOv8

Accurate detection of structural regions in timetable PDFs is crucial for correctly interpreting course entries. This project employs a custom-trained YOLOv8 model to detect three types of layout regions:

**Class 0**: Time slot region

**Class 1**: Week header region

**Class 2**: Course cells (which may contain stacked text)

These regions form the foundational structure upon which all downstream logic (OCR, mapping, parsing) depends.

#### 3.3.1 Model Design and Training

The YOLOv8 layout detector was trained on manually annotated timetable images using LabelImg, following the YOLO format. The training data included: 23 annotated timetable pages, Classes: TimeSlot, Week, CourseBlock, Resolution: 2480 × 3508 (A4 @ 300 DPI), Model: yolov8s (switched from yolov5 and yolov8n due to accuracy issues), Training: 200 epochs, augmented data, fixed-size images.

Annotations were converted into YOLO format with class index and normalised coordinates, shown in Fig. 3.2.

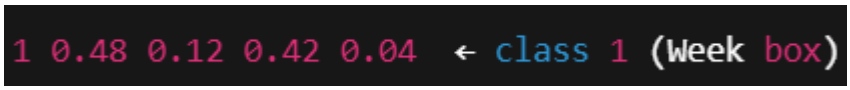



Figure 3.2: YOLO coordinates

The model outputs bounding boxes with class predictions and confidence scores and stores in memory, which are then post-processed before use.

### 3.3.2 Inference and Integration

During runtime, the uploaded PDF is first converted into an PIL image. The function run_yolo_detection(image) in layout_detector.py: Converts the image to RGB NumPy array, feeds it to the loaded YOLO model, extracts the coordinates and class of each predicted box, converts results into a DataFrame shown in Fig. 3.3.

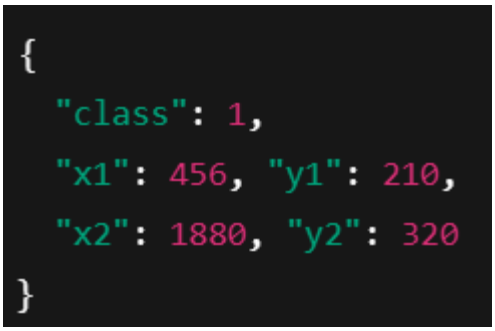



Figure 3.3: YOLO coordinates converted into DataFrame

### 3.3.3 Fallback and Refinement Logic

Although YOLOv8 performs well on clean layouts, inconsistencies may arise due to OCR shifts, header alignment issues, boxes cut off due to colour contrast.

To handle this, the pipeline includes a refinement module: refine_yolo_boxes_with_fallback(boxes_df, ocr_df).

This function uses OCR text to adjust or re-estimate bounding boxes. It anchors week

box using fuzzy-matched text like “Week”, “13”, “EEK”, etc. and estimates time column width proportionally if YOLO fails. It also clamps boxes using known header text (e.g., 0830–0900, 2200–2230) and ensures downstream modules always receive consistent, usable region definitions.

### 3.3.4 Layout Region Roles

Table 3.1. shows the structured output that ensures clean spatial slicing during OCR and helps align blocks precisely to the timetable’s week and time grid.

| Class | Region | Purpose |
|---|---|---|
| 0 | Time Slot Box | Split into 28 vertical 30-min rows |
| 1 | Week Header Box | Split into 15 columns (Week 1–13 + Recess) |
| 2 | Course Cell Box | Intersected with grid to isolate content |

Table 3.1: Layout Regions

## 3.4 Text Extraction and Clustering

Once the structural regions are detected via YOLOv8, the system proceeds to extract textual content from each region using OCR. Given the semi-structured nature of timetable PDFs with stacked text, coloured highlights, and varied alignment. A robust and clean OCR pipeline is critical.

### 3.4.1 OCR Extraction

The OCR is performed using Tesseract via pytesseract.image_to_data, which provides word-level bounding boxes, confidence scores, and text content. The function extract_ocr_df(image) in extract_timetable.py performs full-page OCR. It then converts OCR output into a structured Pandas DataFrame with columns as shown in Fig. 3.4.

```
["text", "conf", "x1", "y1", "x2", "y2", "xc", "yc"]
```

Figure 3.4: OCR output as Pandas DataFrame

All text is cleaned via clean_text() in ocr_utils.py, which normalises symbols (e.g. £ → E, – → -), removes non-ASCII artifacts and uppercases everything for uniformity.
This standardisation greatly improves clustering and fuzzy matching reliability.

### 3.4.2 Preprocessing for OCR Accuracy

To enhance OCR performance on coloured timetable blocks, a cyan colour mask is applied using HSV thresholding to neutralise background highlights. Adaptive thresholding is used to binarize the image and sharpen text edges. OCR is then run on the cleaned image to avoid misreads caused by faded or shaded cells.

These preprocessing steps are encapsulated in extract_ocr_from_block(img_pil, offset_x, offset_y) which accounts for position offsets to maintain accurate bounding box alignment.

### 3.4.3 Clustering with DBSCAN

Fig. 3.5 shows course blocks within timetable cells are usually stacked vertically in 3–4 lines.

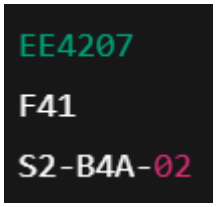



Figure 3.5: Course block stacked vertically within cells

To reconstruct this structure, the OCR words inside each course cell are grouped using DBSCAN based on the vertical centre (yc). This separates lines like Line 1: Course code, Line 2: Group, Line 3: Location. Within each vertical group, words are sorted left to right to rebuild logical text lines. If 4 lines are detected and the first one matches a known holiday (via fuzzy matching), it is handled as a holiday override block.

This logic is key to distinguishing between normal entries and special notes such as Fig. 3.6.


Figure 3.6: Course block that clash with holiday

Fig. 3.7 is an example output after clustering and Fig. 3.8 if a holiday is detected.

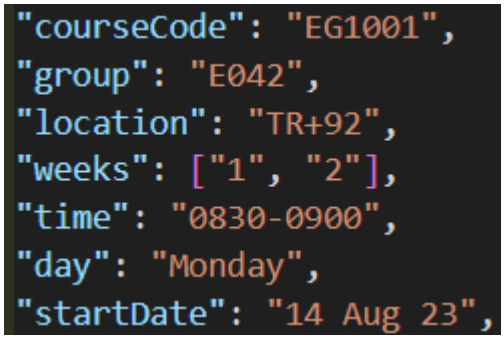



Figure 3.7: Sample output after clustering


Figure 3.8: Sample output with holiday after clustering

### 3.5 Grid Mapping and Time Alignment

After detecting layout regions and extracting text, the system must anchor course blocks to their corresponding weeks and time slots. This is achieved through a spatial mapping process that aligns bounding box coordinates with a precomputed timetable grid.

#### 3.5.1 Grid Mapping and Time Alignment

Using the refined week header bounding box (class 1), the system splits it horizontally into 15 equal-width columns with labels: “Week”, “1”–“7”, “Recess”, “8”–“13”. This is done by get_weeks(week_box) in extract_timetable.py

Fig. 3.9 shows have course block is mapped to the week column. These columns are later used to check which weeks each course block overlaps with. A small padding shrink ratio is applied to avoid misclassification from border noise.

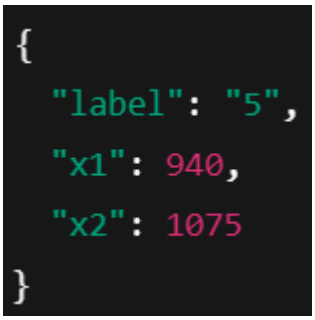



Figure 3.9: Sample week column

### 3.5.2 Time Row Generation

Similarly, the class 0 bounding box (TimeSlot) is split vertically into 28 rows, each representing a 30-minute interval from 0830–0900 to 2200–2230. It is implemented in get_time_rows(time_box) and Fig. 3.10 shows a sample output.

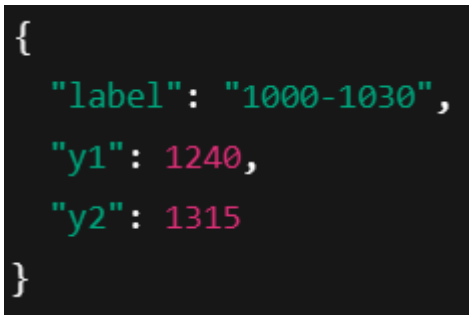



Figure 3.10: Sample time slot row

Each row can later be tested for overlap with a course cell's vertical position to determine its start and end time.

### 3.5.3 Mapping Course Cells to Grid

For each course block bounding box, the system checks for horizontal overlap with week columns and assigns week numbers. It also checks for vertical overlap with time rows then determines start and end time

This is handled in extract_courses(...), using bounding box coordinates of each block.

For example, a course block spans from x=940 to 1075 which matches with week “5”. The same block spans vertically from y=1240 to 1380 which covers two rows: 1000–1030 and 1030–1100. The final time would be “1000–1100”.

The matching uses pixel-based overlap, ensuring precision even with irregularly sized course blocks.

### 3.5.4 Base Date Parsing for Recurrence

To map week numbers to calendar dates (for .ics export), the system will use OCR to detect the line below the “Week” header. Extracts date ranges using fuzzy date parsing (extract_week_date_ranges) and aligns week “1” with its startDate, and computes future dates using Python datetime.

If date parsing fails (due to noise or font issues), the system falls back to UNKNOWN, which can later be edited manually in the Streamlit UI.

### 3.5.5 Summary of Grid Logic

Table 3.2 displays how the mapping ensures that the final JSON output includes precise week sets, day names, and time windows for each course.

| **Element** | **Count** | **Purpose** |
|---|---|---|
| Week Columns | 15 | Assign course to correct weeks |
| Time Rows | 28 | Assign start and end time |
| Grid Mapping | N/A | Converts pixel space to calendar logic |

Table 3.2: Table for grid logic

### 3.6 Semantic Parsing and Calendar Structuring

Once each course block is matched to specific weeks and time slots through spatial mapping, the final stage of the pipeline applies semantic parsing to produce clean, structured entries in JSON format. These entries are suitable for .ics event generation and human-readable review.

#### 3.6.1 Standard Course Block Structure

By default, each course cell is assumed to follow a **3-line vertical format**:

1. Course Code (e.g. EG1001)
2. Group (e.g. E042)
3. Location (e.g. TR+92)

This assumption is based on NTU's consistent formatting of timetable entries and is validated after clustering.

After the course block has been clustered and matched to weeks and time rows, the system constructs a dictionary which is shown in Fig. 3.11.


```
{
  "courseCode": "EG1001",
  "group": "E042",
  "location": "TR+92",
  "weeks": ["5", "6", "7"],
  "day": "Monday",
  "time": "1000-1130",
  "startDate": "14 Aug 23",
  "note": ""
}
```


Figure 3.11: JSON output for standard course block

This format is later used to generate .ics events and to populate editable fields in the Streamlit interface.

### 3.6.2 Holiday Detection

Some timetable entries begin with a holiday label (e.g. Deepavali, National Day) followed by a normal 3-line course block. These special cases are handled by matching the first and/or second OCR line against a fuzzy list of KNOWN_HOLIDAYS (using RapidFuzz). If matched, the block is treated as a 4/5-line entry, where line 1/2 is stored as note and lines 2/3–4 is parsed as a normal course as shown in Fig. 3.12.


Figure 3.12: JSON with holiday

This enables clear visibility of class overrides due to public holidays in both the UI and exported calendars.

### 3.6.3 Grid Mapping and Time Alignment

The function derive_time_range(y1, y2) determines the time label (e.g. “0900–1030”) by comparing the course cell’s vertical span to the 28 known time rows. Week numbers are inferred by measuring which week column(s) the course cell horizontally overlaps with. Multiple overlapping columns mean the class runs across several weeks. If the bounding box fully spans "Recess", it is included in the weeks list but sorted as (7.5) for consistency.

### 3.6.4 Base Date Assignment

Each course block includes a startDate, representing the start of Week 1 on the corresponding weekday. This is inferred from header OCR in the week columns.

If parsing fails, or if multiple blocks exist on the same day/week with known start dates, a fallback mechanism attempts to infer the missing date based on calendar position.

**Merging Logic**

After all blocks are parsed, entries are merged if they satisfy the following conditions:

- Same day
- Same or overlapping time
- Matching courseCode, group, and location

Merged blocks combine their weeks and preserve any notes, ensuring compact calendar entries.

### 3.6.5 Final Output Format

Fig. 3.13 shows how the output format is passed to the ICS generator and Streamlit UI for export or review.


```
{
  "courseCode": "EE4207",
  "group": "F42",
  "location": "S2-B4A-02",
  "weeks": ["1", "2", "3", "4", "5"],
  "day": "Wednesday",
  "time": "1400-1600",
  "startDate": "10 Jan 23",
  "note": ""
}
```


Figure 3.13: Final output format

### 3.7 Error Handling and Fallback Logic

Academic timetable PDFs, while generally structured, are not always cleanly formatted or consistently annotated. This section describes the system's built-in fallbacks and recovery mechanisms, which ensure robust extraction even in the presence of layout inconsistencies, OCR errors, or missing information.

**Missing or Inaccurate Bounding Boxes**

The YOLOv8 layout detector may occasionally fail to detect key regions such as the week column or time slot box due to overlapping elements (e.g. box overlaps "Week" and "1"), low confidence detections from noisy headers or colour schemes that reduce contrast.

To resolve this, the system uses OCR as a fallback anchor and call this function refine_yolo_boxes_with_fallback(...)**.** It looks for text like "WEEK", "13", "0830", "2230" in the OCR output and decide whether if it should expand or recalculates the bounding boxes for class 0 and class 1, applies proportional clamping if bounding box width is off (e.g. time_x2 = week_x1 + width/15). It ensures all downstream functions have valid spatial inputs even if YOLO is partially wrong

#### 3.7.1 OCR Errors and Noisy Text

OCR output may contain broken glyphs (e.g. E042 misread as £042), split text across multiple lines and symbols misinterpreted due to legend colouring.

To handle this, we use preprocessing to includes HSV masking of cyan/pastel highlights before OCR, post-cleaning (clean_text) to removes stray characters and normalizes text and token de-duplication (dedup()) to removes repeated or mis-segmented entries.
This reduces downstream parsing failures and boosts fuzzy matching accuracy.

#### 3.7.2 Holiday Detection Robustness

Fig 3.14 displays holiday entries that are matched against a list of known keywords

(KNOWN_HOLIDAYS).

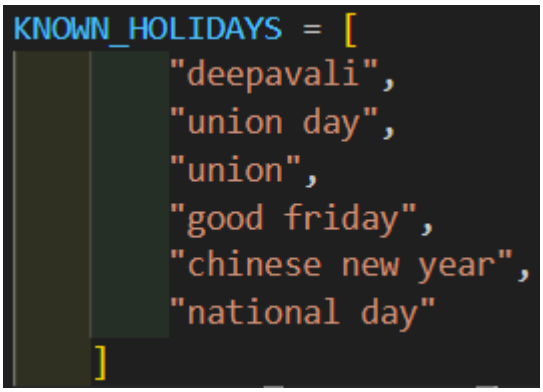



Figure 3.14: Known holidays to match with RapidFuzz

To account for OCR noise, the system uses RapidFuzz with token-sort fuzzy matching that accepts matches above a certain threshold (e.g., 80%) and converts fuzzy hits to standardised note to “Deepavali on Week X” format.

This allows successful holiday detection even when spellings vary (e.g. Deepavli, Chinesc New Year).

### 3.7.3 Date Parsing Failures

Week headers include start and end dates like in the Fig. 3.15.


Figure 3.15: Week headers

When OCR fails to read these lines correctly (e.g. due to formatting or noise), the system attempts fuzzy parsing using dateutil.parser.parse(...) and if parsing still fails, startDate is temporarily set to "UNKNOWN". During Streamlit editing, users can manually input or copy inferred dates from other blocks

### 3.7.4 Redundancy and Merge Safeguards

Merged course entries may produce incorrect overlaps if group names vary slightly (E42 vs E042), or time slots differ slightly due to bounding box rounding.

To handle this, the merge logic includes Fuzzy token matching for courseCode, group and location, time overlap check via minute-level comparison and only merges if it is >85% match on all semantic keys.
This prevents incorrect collapsing of unrelated courses.

## 3.8 Summary of Failover Behaviours

Table 3.3 shows the summary of chapter 3.

| **Issue** | **Fallback Strategy** |
|---|---|
| YOLO box missing | OCR-based coordinate recovery |
| Time/week header partial | Approximate using neighbors + default ratios |
| Broken OCR lines | Clustering + deduplication + token cleanup |
| Holiday misspelling | Fuzzy match with known holidays |
| Date parse failure | Soft fallback to "UNKNOWN"; editable in UI |
| Merge mismatch | Time + token-based match thresholding |

Table 3.3: Summary of failover behaviours

# Chapter 4 YOLOv8 Layout Detection and Training

## 4.1 Motivation for Using YOLOv8

Accurately detecting layout regions is essential for any timetable extraction system. Early attempts using static bounding box templates or colour-based OpenCV heuristics proved unreliable due to shifts in PDF formatting across semesters, variations in alignment.

Thus, a deep learning-based object detector was adopted to locate structural regions dynamically. YOLOv8 was selected for its lightweight architecture and speed, out-of-the-box support for custom training and superior bounding box accuracy compared to YOLOv5 and YOLOv8n.

## 4.2 Dataset Preparation and Annotation

To train the model, a custom dataset of NTU EEE timetable images was manually annotated using LabelImg. Each image was processed at 300 DPI, ensuring high-resolution inputs that match A4 paper size (2480 × 3508 pixels).

The following three classes were annotated is described in Table 4.1.

| Class ID | Region Name | Description |
|---|---|---|
| 0 | TimeSlot | Leftmost column indicating 30-minute rows |
| 1 | WeekHeader | Horizontal row listing Week 1–13 + Recess |
| 2 | CourseBlock | Cells containing stacked course information |

Table 4.1: Description of annotated classes

Each annotated bounding box was saved in YOLO format shown in Fig. 4.1.

```
<class_id> <x_center> <y_center> <width> <height>  ← normalized
```

Figure 4.1: YOLO format of bounding box

The final dataset consisted of 20 training images, 3-way balanced annotation and a test set drawn from unseen semester PDFs.

### 4.3 Model Architecture and Training Process

Initial experiments with yolov5s and yolov8n were dropped due to lower recall. The finalised base model is yolov8s (Ultralytics). Training setup consists of 200 epochs, Adam optimizer, Image size: 640 × 640 and mixed augmentation (scaling, flipping).

Model path in the project shown in Fig. 4.2.

```
model = YOLO("./yolov8/runs/detect/train_3_class/weights/best.pt")
```

Figure 4.2: YOLO model path

Training artifacts were stored in the path shown in Fig. 4.3.

```
/yolov8/runs/detect/train_3_class/
```

Figure 4.3: YOLO training artifacts path

The evaluation metrics includes precision/recall per class, mAP@0.5 and mAP@0.5:0.95 and Visual confirmation on unseen PDFs.

### 4.4 Inference Pipeline

At runtime, the function run_yolo_detection(image) is used to convert PIL image to NumPy array, pass it into the trained model, then extract the results: boxes.xyxy, boxes.cls. Lastly, it formats as a DataFrame with the structure shown in Fig. 4.4.

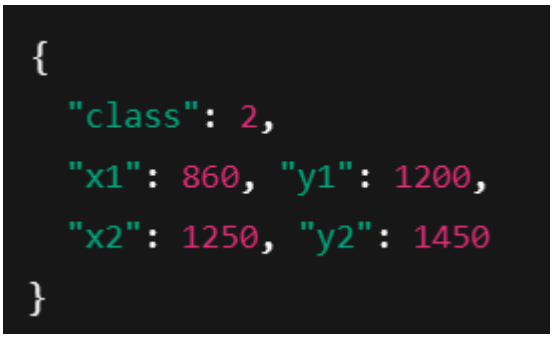



Figure 4.4: Bounding box dataframe structure

This is then passed into downstream logic for OCR and semantic parsing.

### 4.5 Post-Processing and Refinement

Although the model performs well, edge cases still arise. To handle these, this function is created**:** refine_yolo_boxes_with_fallback(...) which includes text-anchored adjustments: if “WEEK” is not aligned, expands x1 and if “13” is cropped, expands x2.

The heuristic estimation checks if time column is missing then recalculate width using 1/15 rule and if time top row (0830–0900) or bottom row (2200–2230) is missing then it uses OCR anchors to infer y1/y2.

This hybrid of learned detection plus rule-based fixing ensures resilience across varying layouts.

# Chapter 5 User Interface and Calendar Export

## 5.1 Motivation and Design

While the automated extraction pipeline performs well, real-world timetables can contain unexpected formatting issues, OCR noise, or schedule changes. To accommodate this, a user-facing interface was developed using Streamlit which is a lightweight Python framework for building web apps.

The interface serves two core purposes:

1. **Review and Correction**

   Allows users to view and manually edit extracted course blocks before exporting.

2. **Calendar Integration**

   Enables users to convert structured course data into .ics files compatible with major calendar platforms.

## 5.2 Application Workflow

Upon visiting the app, users are presented with a simple and guided experience:

1. **Upload Timetable PDF**

   The user uploads a teaching load PDF from NTU EEE. The app handles file conversion and triggers extraction.

2. **Extraction & Preview**

   On clicking “Extract Timetable”, the PDF is processed through the pipeline:

   - PDF → Image
   - Layout detection (YOLOv8)
   - OCR & clustering
   - Grid alignment
   - Semantic structuring

3. **Review Interface**

   Each course block is displayed in an expandable form where the user can:

   - Edit the course code, group, location

- Select weeks (multi-select)
- Adjust day of week and time range
- Modify or assign a start date
- Add or verify holiday notes

4. **Validation**

   Fields marked with * are required. The app checks for missing values and warns users before allowing export.

5. **Calendar Export**

   Once all entries are verified, clicking “Convert to ICS” triggers event generation. Users can then download the .ics file for import into Google Calendar, Outlook, or Apple Calendar.

### 5.3 Key Features

Manual Editing where each extracted entry is editable. This is crucial for fixing OCR mistakes (e.g., misread course codes), handling missed entries or rare formatting edge cases, adding inferred start dates when not auto detected.

Add/Delete Course Entries where Users can add new blocks manually via “➕ Add New Course” and delete any auto-extracted blocks that are incorrect or duplicated.

Time & Week Selection where Weeks are multi-select dropdown with support for 1–13 + Recess, time is a structured start/end picker using hour-minute granularity and start date uses Calendar picker with formatting hint (e.g., 14/08/2023).

ICS Generation converts the JSON results into .ics file by using the generate_ics_from_courses() function from ui_helpers.py. Events include Title: CourseCode (Group), Location, Recurrence: Weekly, with specific week numbers, Start/end time and optional note (e.g., Deepavali on Week 13).

### 5.4 Interface Snapshot

The following Fig. 5.1 to Fig. 5.4 shows how the application looks like.


Figure 5.1: Landing page


Figure 5.2: Timetable PDF selected

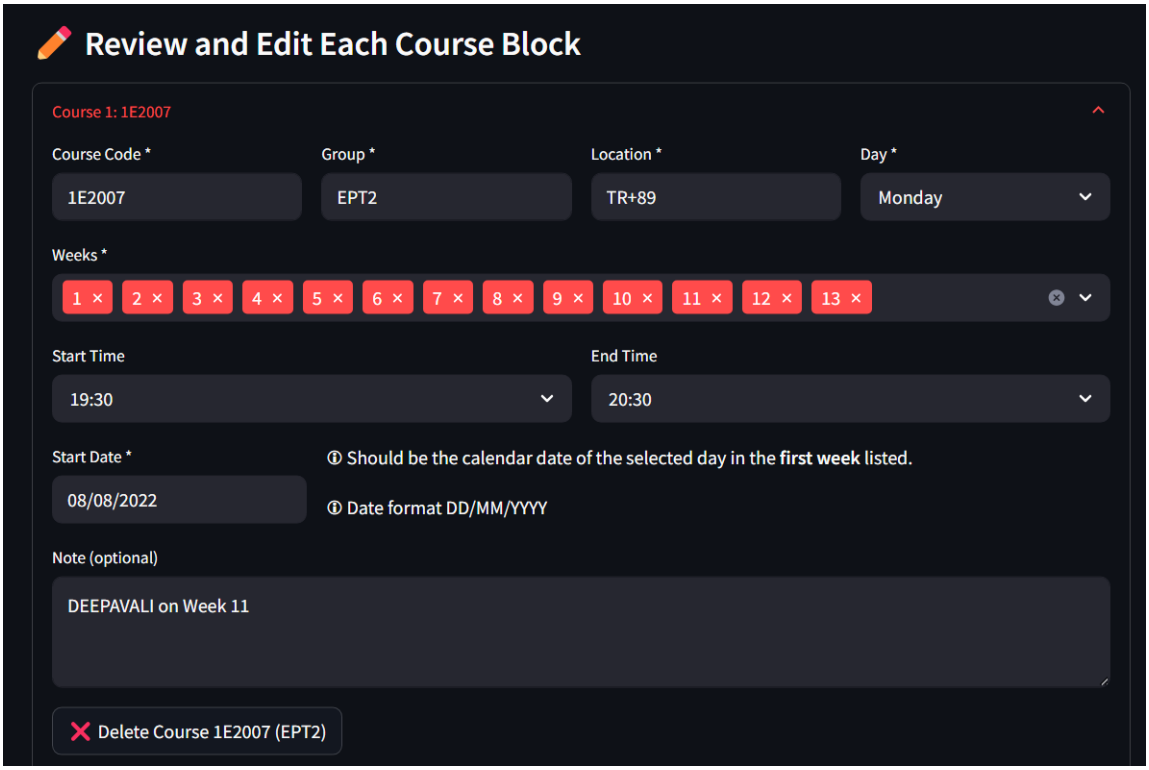



Figure 5.3: Timetable extracted and displayed

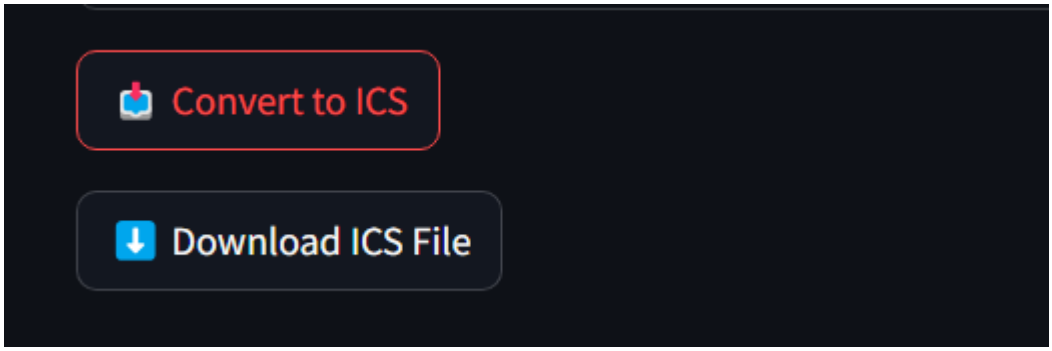



Figure 5.4: “Convert to ICS” is clicked and Download is ready

# Chapter 6 Conclusion and Future Work

## 6.1 Conclusion

This project successfully delivers a complete end-to-end system that automates the conversion of structured university timetables into calendar-compatible formats. By integrating deep learning, OCR, clustering algorithms, and a user-facing interface, the system addresses a persistent inefficiency faced by academic staff, mainly the manual re-entry of teaching schedules into personal calendar tools.

Key accomplishments include:

- **Custom YOLOv8-based layout detection** trained on NTU EEE timetable formats
- **OCR and clustering pipeline** for reconstructing vertically stacked course entries
- **Robust mapping** of bounding boxes to weekly time grids with fallback logic
- **Holiday and note detection** via fuzzy text matching
- **Streamlit interface** for interactive review, editing, and .ics export

With an average extraction accuracy of **80–85%**, and editable UI features for resolving edge cases, the system provides both automation and user control in a domain where precision matters.

This tool not only improves the quality of life for NTU faculty but also demonstrates how layout-aware machine learning techniques can streamline document-to-calendar workflows.

## 6.2 Recommendation in Future Work

While the current system is functional and effective within the scope of NTU EEE timetable PDFs, several enhancements can improve robustness, scalability, and user experience:

**Generalize Beyond EEE**

- Train on a broader range of faculty timetable formats
- Add layout adaptability using self-supervised table detection (e.g., Table Transformer)

**Smarter Recurrence Logic**

- Handle alternating week patterns (e.g., "Week 1, 3, 5 only")
- Detect and group repeated events into a single recurring calendar entry

**Multi-language OCR Support**

- Some public holidays may be listed in non-English terms
- Add language detection or multilingual OCR fallback

**Batch Processing**

- Enable upload of multiple timetables at once (for department coordinators)
- Export merged .ics calendars with color-coded layers per course/group

**Calendar Sync API Integration**

- Provide direct push to Google Calendar via OAuth
- Allow users to sync updates without manual .ics downloads

**Accuracy Dashboard**

- Visual feedback on confidence scores or extraction errors
- Let users highlight/correct problematic entries to retrain future models

## Reflection on Learning Outcome Attainment

During the course of this FYP, I developed both practical and conceptual skills across multiple engineering competencies. This project demanded not only technical implementation but also careful structuring of problem domains and iterative system design.

Firstly, I strengthened my ability in the design and development of solutions. The project evolved through several phases such as from initial attempts using static heuristics to the integration of YOLOv8 for layout detection and OCR for content parsing. Designing the end-to-end pipeline required modular thinking, fallback strategies, and the development of custom logic for edge cases such as stacked course entries and public holiday overrides.

Secondly, this project deepened my exposure to modern tool usage. I gained hands-on experience with deep learning frameworks such as Ultralytics YOLO, optical character recognition using Tesseract, and rapid deployment with Streamlit. Integrating these tools into a cohesive pipeline taught me not only how to use them, but how to make them work together in a user-centric, production-grade setting.

Finally, the project honed my problem analysis skills. Extracting structured data from unstructured PDF layouts is inherently ambiguous and error prone. I had to continuously observe failure modes such as OCR misreads, layout inconsistencies, and ambiguous week-date mappings and adapt the system to address them. This analytical feedback loop was crucial to achieving a usable and robust solution.

## References

[1] DocHub, "Convert PDF to ICS," [Online]. Available: https://www.dochub.com/en/functionalities/convert-pdf-to-ics

[2] pdfFiller, "Convert PDF to ICS Online," [Online]. Available: https://www.pdffiller.com/en/functionality/convert-pdf-to-ics-online.htm

[3] Aspose.Total, "Convert PDF to ICS," [Online]. Available: https://products.aspose.com/total/python-net/conversion/pdf-to-ics

[4] PDFtoCal, "Convert PDF Schedules to Google Calendar Events," [Online]. Available: https://www.pdftocal.com/

[5] Reddit, "Easiest way to convert PDF to something readable for Google Calendar?," [Online]. Available: https://www.reddit.com/r/techsupport/comments/12wu29x/easiest_way_to_convert_pdf_to_something_readable/

[6] D. Prasad et al., "CascadeTabNet: An approach for end to end table detection and structure recognition from image-based documents," in Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. Workshops, 2020, pp. 572–573.

[7] S. Schreiber et al., "DeepDeSRT: Deep learning for detection and structure recognition of tables in document images," in Proc. 14th IAPR Int. Conf. Document Anal. Recognit., 2017, pp. 1162–1167.

[8] Ultralytics, "YOLOv8 vs YOLOv5: A Detailed Comparison," [Online]. Available: https://docs.ultralytics.com/compare/yolov8-vs-yolov5/

[9] A. W. Harley et al., "Evaluation of Deep Learning Methods for Document Image Classification and Retrieval," in Proc. IAPR Int. Conf. Document Analysis and Recognition (ICDAR), 2015, pp. 991–995.

[10] H. Zhao and K. Lu, "Calendar Extraction and Visualization from Historical Tables in PDF Files," in Proc. 2020 Int. Conf. on Intelligent Human Systems Integration (IHSI), pp. 654–660.

[11] V. Zhong, C. Sun, and K. Srivastava, "PubLayNet: Largest Dataset for Document Layout Analysis," in Proc. Int. Conf. on Document Analysis and Recognition (ICDAR), 2019, pp. 1015–1022.

# Appendix

A. constants.py

```
DAYS = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday"]
WEEKS = ["1", "2", "3", "4", "5", "6", "7", "Recess", "8", "9", "10", "11", "12", "13"]
KNOWN_HOLIDAYS = [
        "deepavali",
        "union day",
        "union",
        "good friday",
        "chinese new year",
        "national day"
    ]
IMAGE_SIZE = (2480, 3508)
```

B. ocr_utils.py

```
import re

def clean_text(s: str) -> str:
    """
    Cleans OCR text by removing special characters and normalizing symbols.
    """
    s = s.replace("£", "E").replace("—", "-").replace("–", "-").strip()
    s = re.sub(r"[^\x20-\x7E]", "", s)   # Remove non-ASCII
    return s.upper()
```

C. ui_helpers.py

```
# scripts/utils/ui_helpers.py

from datetime import datetime, timedelta, date
import streamlit as st
from ics import Calendar, Event
from ics.grammar.parse import ContentLine
import pytz

from datetime import datetime, timedelta
from scripts.utils.constants import DAYS, WEEKS

def red_alert(text):
    st.markdown(f'<div style="color: red; font-weight: bold;">⚠️ {text}</div>', unsafe_allow_html=True)

def get_week_index(w):
    return WEEKS.index(w) if w in WEEKS else -1

def extract_semester_year(courses):
    for c in courses:
        raw = str(c.get("startDate", "")).strip()
        try:
            dt = datetime.strptime(raw, "%d %b %y")
            return dt.year
        except:
            continue
```

```
    return datetime.today().year

def get_inferred_start_date(course, all_courses):
    """
    Infer course startDate based on other block with:
    - Same day, any week → adjust by week index diff
    - Same week, different day → adjust by weekday diff
    Skips if both week and day are different.
    """
    def day_index(day_name):
        try:
            return DAYS.index(day_name)
        except:
            return -1

    raw = str(course.get("startDate", "")).strip().upper()
    if raw and raw != "UNKNOWN":
        try:
            datetime.strptime(raw, "%d %b %y")
            return None, None   # already valid
        except:
            pass

    try:
        course_weeks = [w for w in course.get("weeks", []) if w in WEEKS]
        course_week = course_weeks[0] if course_weeks else None
    except:
        course_week = None

    course_day = course.get("day")
```

```
for other in all_courses:
    if other["id"] == course["id"]:
        continue

    other_day = other.get("day")
    other_weeks = [w for w in other.get("weeks", []) if w in WEEKS]
    other_week = other_weeks[0] if other_weeks else None

    try:
        known_raw = str(other.get("startDate", "")).strip()
        known_date = datetime.strptime(known_raw, "%d %b %y").date()
    except:
        continue

    # Same week, different day
    if course_week == other_week and course_day != other_day:
        d_diff = day_index(course_day) - day_index(other_day)
        return known_date + timedelta(days=d_diff), other

    # Same day, different week
    if course_day == other_day and course_week != other_week:
        w_diff = get_week_index(course_week) - get_week_index(other_week)
        return known_date + timedelta(weeks=w_diff), other

    # Exact match
    if course_day == other_day and course_week == other_week:
        return known_date, other

return None, None   # Nothing found
```

```
def render_date_input(c1, course, all_courses):
    """
    Display a date_input with safe update logic:
    - Automatically assigns inferred dates
    - Shows red alerts for missing or inferred values
    - Only updates if the user actually changes the date
    """
    from scripts.utils.ui_helpers import get_inferred_start_date, extract_semester_year, red_alert
    from datetime import datetime, date

    raw = str(course.get("startDate", "")).strip()
    parsed_date = None
    inferred_note = ""
    inferred_flag = False

    # Try to parse the course startDate
    if raw and raw.upper() != "UNKNOWN":
        try:
            parsed_date = datetime.strptime(raw, "%d %b %y").date()
        except:
            parsed_date = None
    elif isinstance(course.get("startDate"), date):
        parsed_date = course["startDate"]

    # If not valid, try to infer
    if not parsed_date:
        inferred_date, inferred_from = get_inferred_start_date(course, all_courses)
```

```
            if inferred_date:
                parsed_date = inferred_date
                course["startDate"] = inferred_date  # ✅ Save it directly
                inferred_flag = True
                inferred_note = f" (inferred from {inferred_from.get('courseCode', 'another block')})"
            else:
                course["startDate"] = "UNKNOWN"

        # Label setup
        label = "Start Date *"
        if course["startDate"] == "UNKNOWN":
            label += " (⚠️ not selected)"
        elif inferred_flag:
            label += inferred_note

        # Fallback year for dummy UI value
        fallback_year = parsed_date.year if parsed_date else extract_semester_year(all_courses)
        shown_date = parsed_date or date(fallback_year, 1, 1)

        # Render the widget
        picked_date = c1.date_input(
            label,
            value=shown_date,
            key=f"sd_{course['id']}",
            format="DD/MM/YYYY"
        )

        # Save if the user manually changed it
```

```
    if parsed_date is None or picked_date != parsed_date:
        course["startDate"] = picked_date

    # Show red alert if missing or inferred
    if course["startDate"] == "UNKNOWN":
        red_alert("❗ Start date is missing. Please select a valid date before exporting.")
    elif inferred_flag:
        red_alert(f"⚠️ This date was inferred from {inferred_from.get('courseCode', 'another course')}. Please confirm it is correct.")

    return course




def render_time_inputs(st, course):
    """
    Render time inputs for start and end. Validates order and limits to 08:30–22:30.
    Returns (updated_course, error_message).
    """
    # Allowed range
    MIN_TIME = datetime.strptime("08:30", "%H:%M").time()
    MAX_TIME = datetime.strptime("22:30", "%H:%M").time()

    # Parse or default
    try:
        start_raw, end_raw = course["time"].split("-")
        start_dt = datetime.strptime(start_raw, "%H%M").time()
        end_dt = datetime.strptime(end_raw, "%H%M").time()
```

```
    except:
        start_dt = MIN_TIME
        end_dt = datetime.strptime("09:00", "%H:%M").time()

    # Show inputs
    time_cols = st.columns([1, 1])
        start_time = time_cols[0].time_input("Start Time", value=start_dt, step=timedelta(minutes=30), key=f"start_{course['id']}")
        end_time = time_cols[1].time_input("End Time", value=end_dt, step=timedelta(minutes=30), key=f"end_{course['id']}")

    # Validate inputs
    if start_time >= end_time:
        return course, " ⚠️ End time must be after start time."
    if start_time < MIN_TIME or end_time > MAX_TIME:
        return course, " ⚠️ Time must be between 08:30 and 22:30."

    # Store formatted string
    course["time"] = f"{start_time.strftime('%H%M')}-{end_time.strftime('%H%M')}"
    return course, None


def split_weeks_into_blocks(weeks):
    blocks = []
    current = []
    prev_index = None

    for w in weeks:
        if w == "Recess":
            if current:
```

```
                blocks.append(current)
                current = []
            prev_index = None
            continue

        idx = get_week_index(w)
        if prev_index is not None and idx != prev_index + 1:
            blocks.append(current)
            current = []
        current.append(w)
        prev_index = idx

    if current:
        blocks.append(current)

    return blocks

def generate_ics_from_courses(courses):
    cal = Calendar()
    errors = []
    sg = pytz.timezone("Asia/Singapore")
    all_events = []

    for idx, entry in enumerate(courses):
        try:
            raw_date = entry.get("startDate")
            if isinstance(raw_date, str):
                if raw_date.upper() == "UNKNOWN":
                    raise ValueError("Missing startDate")
                start_date = datetime.strptime(raw_date, "%d %b %y").date()
```

```
        elif isinstance(raw_date, date):
            start_date = raw_date
        else:
            raise ValueError("Invalid startDate type")

        start_str, end_str = entry["time"].split("-")
        start_h, start_m = int(start_str[:2]), int(start_str[2:])
        end_h, end_m = int(end_str[:2]), int(end_str[2:])

        course_weeks = entry["weeks"]
        all_blocks = split_weeks_into_blocks(course_weeks)

        course_first_week = course_weeks[0]
        course_first_week_index = get_week_index(course_first_week)

        for block in all_blocks:
            block_first_week = block[0]
            block_week_index = get_week_index(block_first_week)
            week_diff = block_week_index - course_first_week_index

            # ✅ Adjust only by week_diff to preserve original weekday
            session_date = start_date + timedelta(weeks=week_diff)

            dt_start = sg.localize(datetime(session_date.year, session_date.month, session_date.day, start_h, start_m))
            dt_end = sg.localize(datetime(session_date.year, session_date.month, session_date.day, end_h, end_m))

            e = Event()
            e.name = f"{entry['courseCode']} ({entry['group']})"
```

```
                e.location = entry["location"]
                e.begin = dt_start
                e.end = dt_end
                e.description = f"Weeks: {', '.join(block)}"
                if entry.get("note"):
                    e.description += f"\nNote: {entry['note']}"

                # RRULE: recurrence per week block
                e.rrule = {"FREQ": "WEEKLY", "COUNT": len(block)}
                                        e.extra.append(ContentLine(name="RRULE",
value=f"FREQ=WEEKLY;COUNT={len(block)}"))

                all_events.append(e)

        except Exception as e:
            errors.append(f"Error in Course {idx + 1}: {e}")

    # Sort by time and course
    all_events.sort(key=lambda e: (e.begin.datetime, e.name))
    for e in all_events:
        cal.events.add(e)

    return cal, errors
```

D. extract_timetable.py

```
import re
import pandas as pd
from datetime import datetime, timedelta
```

```
from pdf2image import convert_from_path
from pytesseract import image_to_data, Output
from sklearn.cluster import DBSCAN
import numpy as np
from rapidfuzz import process, fuzz
import cv2
from scripts.utils.ocr_utils import clean_text
from scripts.utils.constants import DAYS, WEEKS, KNOWN_HOLIDAYS
from scripts.layout_detector import get_refined_layout_boxes
from dateutil import parser

def dedup(text):
    tokens = text.split()
    clean_tokens = []
    seen = set()
    for token in tokens:
        key = re.sub(r"[^A-Z0-9+]", "", token)
        if key and key not in seen:
            clean_tokens.append(token)
            seen.add(key)
    return " ".join(clean_tokens)

def week_sort_key(w):
    return 7.5 if w == "Recess" else int(w)

def time_overlap(t1, t2):
    def to_minutes(t): return int(t[:2]) * 60 + int(t[2:])
    s1, e1 = t1.split('-')
    s2, e2 = t2.split('-')
    return not (to_minutes(e1) <= to_minutes(s2) or to_minutes(e2) <= to_minutes(s1))
```

```
def is_month_like(token, threshold=80):
    token = token.lower()
    months = [m.lower() for m in ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',
                                  'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']]
    return any(fuzz.partial_ratio(token, m) >= threshold for m in months)


def extract_week_date_ranges(image, weeks_box):
    week_columns = get_weeks(weeks_box)
    box_height = weeks_box["y2"] - weeks_box["y1"]
    y1 = weeks_box["y2"]
    y2 = y1 + box_height * 2   # Go below to get both lines

    week_to_date_pair = {}
    for i, wk in enumerate(week_columns[1:]):   # skip "Week"
        x1, x2 = int(wk["x1"]), int(wk["x2"])
        crop = image.crop((x1 - 5, int(y1), x2 + 5, int(y2)))

                lines = [line.strip() for line in image_to_data(crop,
output_type=Output.DICT)["text"] if line.strip()]

        if len(lines) >= 6:
            try:
                start_str = " ".join(lines[0:3])
                end_str   = " ".join(lines[3:6])
                start = datetime.strptime(start_str, "%d %b %y")
                end = datetime.strptime(end_str, "%d %b %y")

                week_to_date_pair[wk["label"]] = (start, end)
```

```
            except:
                week_to_date_pair[wk["label"]] = ("UNKNOWN", "UNKNOWN")
        else:
            week_to_date_pair[wk["label"]] = ("UNKNOWN", "UNKNOWN")

    return week_to_date_pair

def extract_base_start_date_from_weeks(image, week_boxes):
    if week_boxes.empty:
        # print("❌ No week box (class 1) found.")
        return None

    weeks = get_weeks(week_boxes)

    # Use Week 1 column
    week1 = weeks[1]   # Index 1 = Week 1
    x1, x2 = week1["x1"], week1["x2"]
    y2 = week_boxes["y1"]
    y1 = y2 - 60   # go upwards above week label

    # Crop image
    cropped = image.crop((x1, y1, x2, y2))
    img = np.array(cropped.convert("RGB"))

    # HSV masking & preprocessing
    hsv = cv2.cvtColor(img, cv2.COLOR_RGB2HSV)
    cyan_mask = cv2.inRange(hsv, (70, 20, 100), (110, 255, 255))
    img[cyan_mask > 0] = [255, 255, 255]
    gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
    norm = cv2.normalize(gray, None, 0, 255, cv2.NORM_MINMAX)
```

```
    thresh = cv2.adaptiveThreshold(norm, 255, cv2.ADAPTIVE_THRESH_MEAN_C,
                                   cv2.THRESH_BINARY, 15, 10)

    config = r'--psm 6'
    ocr_data = image_to_data(thresh, output_type=Output.DICT, config=config)
    lines = [t.strip() for t in ocr_data["text"] if t.strip()]
    text = " ".join(lines)

    # Try fuzzy date parsing
    try:
        parsed = parser.parse(text, fuzzy=True, dayfirst=True)
        return parsed
    except:
        return "UNKNOWN"



def detect_holiday_from_ocr(lines, threshold=80):
    """Try to match joined OCR lines to a known holiday using fuzzy matching."""
    joined = " ".join(lines).lower().strip()
        match, score, _ = process.extractOne(joined, KNOWN_HOLIDAYS,
scorer=fuzz.token_sort_ratio)
    if score >= threshold:
        return match.upper()   # standardized clean name
    return None

def get_weeks(week_box, shrink_ratio=0.1):
    week_labels = ["Week"] + WEEKS
    col_width = (week_box["x2"] - week_box["x1"]) / 15

    boxes = []
```

```
    for i in range(15):
        x1 = week_box["x1"] + i * col_width
        x2 = week_box["x1"] + (i + 1) * col_width
        pad = (x2 - x1) * shrink_ratio / 2
        boxes.append({
            "label": week_labels[i],
            "x1": x1 + pad,
            "x2": x2 - pad,
            "index": i
        })
    return boxes

def get_time_rows(time_box):
    start = datetime.strptime("0830", "%H%M")
    row_height = (time_box["y2"] - time_box["y1"]) / 28
    return [{
        "label": f"{(start + timedelta(minutes=30 * i)).strftime('%H%M')}-{(start + timedelta(minutes=30 * (i + 1))).strftime('%H%M')}",
        "y1": time_box["y1"] + i * row_height,
        "y2": time_box["y1"] + (i + 1) * row_height
    } for i in range(28)]

def extract_ocr_df(image):
    ocr = image_to_data(image, output_type=Output.DICT)
    df = pd.DataFrame({
        "text": pd.Series(ocr["text"]).str.strip(),
        "conf": ocr["conf"],
        "x1": ocr["left"],
        "y1": ocr["top"],
        "width": ocr["width"],
```

```
        "height": ocr["height"]
    })
    df = df[(df["text"] != "") & (df["conf"] != "-1")].copy()
    df["x2"] = df["x1"] + df["width"]
    df["y2"] = df["y1"] + df["height"]
    df["xc"] = (df["x1"] + df["x2"]) / 2
    df["yc"] = (df["y1"] + df["y2"]) / 2
    return df

def extract_ocr_from_block(img_pil, offset_x=0, offset_y=0):
    img = np.array(img_pil.convert("RGB"))
    hsv = cv2.cvtColor(img, cv2.COLOR_RGB2HSV)
    cyan_mask = cv2.inRange(hsv, (70, 20, 100), (110, 255, 255))
    img[cyan_mask > 0] = [255, 255, 255]
    gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
    norm = cv2.normalize(gray, None, 0, 255, cv2.NORM_MINMAX)
    thresh = cv2.adaptiveThreshold(norm, 255, cv2.ADAPTIVE_THRESH_MEAN_C,
                                   cv2.THRESH_BINARY, 15, 10)
    config = r'--oem 3 --psm 6'
    data = image_to_data(thresh, output_type=Output.DICT, config=config)
    df = pd.DataFrame({
        "text": pd.Series(data["text"]).str.strip(),
        "conf": data["conf"],
        "x1": data["left"],
        "y1": data["top"],
        "width": data["width"],
        "height": data["height"]
    })
    df = df[(df["text"] != "") & (df["conf"] != "-1")].copy()
    df["x1"] += offset_x
```

```
    df["x2"] = df["x1"] + df["width"]
    df["y1"] += offset_y
    df["y2"] = df["y1"] + df["height"]
    df["xc"] = (df["x1"] + df["x2"]) / 2
    df["yc"] = (df["y1"] + df["y2"]) / 2
    return df

def extract_courses(image, course_df, weeks, time_rows, day, week_to_date_pair):
    entries = []
    for _, blk in course_df.iterrows():
        bx1, bx2, by1, by2 = blk["x1"], blk["x2"], blk["y1"], blk["y2"]
        for wk in weeks:
            if wk["index"] == 0 or bx2 < wk["x1"] or bx1 > wk["x2"]:
                continue

            margin = 10   # pixels
            sx1 = max(bx1, wk["x1"] - margin)
            sx2 = min(bx2, wk["x2"] + margin)
            cropped = image.crop((sx1, by1, sx2, by2))
            ocr_inside = extract_ocr_from_block(cropped, offset_x=sx1, offset_y=by1)
            if ocr_inside.empty:
                continue

            y_coords = ocr_inside["yc"].values.reshape(-1, 1)
            labels = DBSCAN(eps=20, min_samples=1).fit(y_coords).labels_
            ocr_inside["line_group"] = labels
            grouped = list(ocr_inside.groupby("line_group", sort=False))
            line_map = [" ".join(group.sort_values("x1")["text"].values) for _, group in
grouped]
```

```
                index_to_group_id = {i: group_id for i, (group_id, _) in
enumerate(grouped)}

            used_lines = set()
            i = 0
            while i < len(line_map):
                if i in used_lines:
                    i += 1
                    continue

                # === Step 1: Detect holiday block ===
                note_lines = []
                note_index = None

                while i < len(line_map):
                    current = clean_text(line_map[i])
                    next_line = clean_text(line_map[i + 1]) if i + 1 < len(line_map) else
""

                    maybe_combo = detect_holiday_from_ocr([current, next_line])
                    if maybe_combo:
                        if maybe_combo not in note_lines:
                            note_lines.append(maybe_combo)
                        note_index = i + 2
                        i += 2
                        continue

                    maybe_single = detect_holiday_from_ocr([current])
                    if maybe_single:
                        if maybe_single not in note_lines:
```

```
            note_lines.append(maybe_single)
        note_index = i + 1
        i += 1
        continue

    break

# === Step 2: Look for next 3 non-holiday lines ===
while i <= len(line_map) - 3:
    block = [clean_text(line_map[j]) for j in range(i, i + 3)]
    if all(line.lower() not in KNOWN_HOLIDAYS for line in block):
        break
    i += 1
else:
    break

course_lines = [clean_text(line_map[j]) for j in range(i, i + 3)]

# ✅ Only attach note if this block comes right after holiday
note = ""
if note_lines and i == note_index:
    unique_lines = list(dict.fromkeys(note_lines))
    note = f"{' '.join(unique_lines).strip()} on Week {wk['label']}"

def is_location(t):
    t = t.upper()
    return (
        t.startswith(("TR", "LT", "LKC", "S")) or
        "+" in t or "-" in t or
        (any(c.isdigit() for c in t) and any(c.isalpha() for c in t) and
```

```
len(t) >= 5)
                    )

                def is_course_code(t):
                    t = t.upper()
                    return (
                        len(t) >= 5 and
                        any(c.isdigit() for c in t) and
                        any(c.isalpha() for c in t) and
                        not is_location(t)
                    )

                  courseCode = next((x for x in course_lines if is_course_code(x)), course_lines[0])
                   location = next((x for x in course_lines if is_location(x) and x != courseCode), course_lines[2])
                group = next((x for x in course_lines if x not in [courseCode, location]), course_lines[1])

                group_ids = [index_to_group_id.get(j, -1) for j in range(i, i+3)]
                y_groups = ocr_inside[ocr_inside["line_group"].isin(group_ids)]
                y1_lines = y_groups["y1"].min()
                y2_lines = y_groups["y2"].max()

                matched_times = [r["label"] for r in time_rows if not (r["y2"] < y1_lines or r["y1"] > y2_lines)]
                if not matched_times:
                    i += 1
                    continue
```

```
                                                                time_range =
f"{matched_times[0].split('-')[0]}-{matched_times[-1].split('-')[1]}"

                    start, end = week_to_date_pair.get(wk["label"], ("UNKNOWN",
"UNKNOWN"))
                day_offset = DAYS.index(day)

                if isinstance(start, datetime):
                                                        start_date = (start +
timedelta(days=day_offset)).strftime("%d %b %y")
                else:
                    start_date = "UNKNOWN"

                entries.append({
                    "courseCode": courseCode,
                    "group": group,
                    "location": location,
                    "weeks": [wk["label"]],
                    "time": time_range,
                    "day": day,
                    "startDate": start_date,
                    "note": note
                })

                used_lines.update({i, i+1, i+2})
                i += 3

    return entries

def merge_entries(entries):
```

```
def score_course_code(code):
    tokens = re.findall(r"[A-Z0-9]+", code)
    return (len(tokens), len(code))


merged = []
used = [False] * len(entries)
for i, e1 in enumerate(entries):
    if used[i]:
        continue
    group = [e1]
    used[i] = True
    for j in range(i + 1, len(entries)):
        e2 = entries[j]
        if used[j]:
            continue
        if e1["day"] == e2["day"] and time_overlap(e1["time"], e2["time"]):
            group.append(e2)
            used[j] = True
    course_codes = [e["courseCode"] for e in group]
    best_code = dedup(clean_text(max(course_codes, key=score_course_code)))
    get_common = lambda field: max(set(field), key=field.count)
    group_field = dedup(clean_text(get_common([e["group"] for e in group])))
    location_field = dedup(clean_text(get_common([e["location"] for e in group])))
    weeks = sorted(set(w for e in group for w in e["weeks"]), key=week_sort_key)
    notes = "; ".join(sorted(set(e["note"] for e in group if e["note"])))
    merged.append({
        "courseCode": best_code,
        "group": group_field,
        "location": location_field,
        "weeks": weeks,
```

```
            "time": e1["time"],
            "day": e1["day"],
            "startDate": e1["startDate"],
            "note": notes
        })
    return merged

# === MAIN PIPELINE ===
def extract_timetable(pdf_path: str) -> list[dict]:
    all_output = []

    for idx in range(5):
        day = DAYS[idx]

            image = convert_from_path(pdf_path, dpi=300, first_page=idx+1, last_page=idx+1)[0]

        ocr_df = extract_ocr_df(image)
        refined = get_refined_layout_boxes(image, ocr_df)

        time_box = refined[0]
        week_box = refined[1]
        course_df = pd.DataFrame([refined[2]])

        weeks = get_weeks(week_box)
        time_rows = get_time_rows(time_box)

        week_to_date_pair = extract_week_date_ranges(image, week_box)

          day_entries = extract_courses(image, course_df, weeks, time_rows, day,
```

```
week_to_date_pair)
            merged_day = merge_entries(day_entries)
            all_output.extend(merged_day)

    return all_output
```

E. layout_detector.py

```
# layout_detector.py

import pandas as pd
import numpy as np
from PIL import Image
from rapidfuzz import fuzz
from scripts.utils.ocr_utils import clean_text
from ultralytics import YOLO

model = YOLO("./yolov8/runs/detect/train_3_class/weights/best.pt")  # Update if path changes

def run_yolo_detection(image: Image.Image):
    img_array = np.array(image.convert("RGB"))
    results = model(img_array)
    boxes = []
    for box in results[0].boxes:
        cls = int(box.cls[0])
        x1, y1, x2, y2 = box.xyxy[0].tolist()
        boxes.append({
            "class": cls,
            "x1": x1, "x2": x2,
            "y1": y1, "y2": y2
```

```
            })
    return pd.DataFrame(boxes)

def refine_yolo_boxes_with_fallback(box_df, ocr_df, ocr_line_gap=10):
    refined = {}

    # Group OCR lines
    ocr_df_sorted = ocr_df.sort_values("y1")
    grouped_lines = []
    current_line = []
    last_y = None
    for _, row in ocr_df_sorted.iterrows():
        if last_y is None or abs(row["y1"] - last_y) <= ocr_line_gap:
            current_line.append(row)
        else:
            grouped_lines.append(current_line)
            current_line = [row]
        last_y = row["y1"]
    if current_line:
        grouped_lines.append(current_line)

    def find_line_pair(start_str, end_str):
        for i in range(len(grouped_lines) - 1):
            line1 = " ".join(clean_text(w["text"]) for w in grouped_lines[i])
            line2 = " ".join(clean_text(w["text"]) for w in grouped_lines[i + 1])
            if start_str in line1 and end_str in line2:
                y1 = min(w["y1"] for w in grouped_lines[i])
                y2 = max(w["y2"] for w in grouped_lines[i + 1])
                return y1, y2
        return None, None
```

```
    # === Week (class 1) ===
    week_boxes = box_df[box_df["class"] == 1]
    if week_boxes.empty:
        return refined
    wk = week_boxes.iloc[0].copy()
    for _, row in ocr_df.iterrows():
        txt = clean_text(row["text"])
        if any(fuzz.partial_ratio(txt, key) > 80 for key in ["WEEK", "VEEK", "EEK", "EKS"]):
            if row["x1"] < wk["x1"]:
                wk["x1"] = row["x1"] - 30
        if "13" in txt and row["x2"] > wk["x2"]:
            wk["x2"] = row["x2"] + 60
    refined[1] = wk

    # === TimeSlot (class 0) ===
    time_col_width = (wk["x2"] - wk["x1"]) * 0.08 # Get proportional width for fallback
    time_boxes = box_df[box_df["class"] == 0]
    ts = time_boxes.iloc[0].copy() if not time_boxes.empty else {}
    if "x1" in ts and "x2" in ts:
        x_shift = wk["x1"] - ts["x1"]
        ts["x1"] += x_shift
        ts["x2"] += x_shift
    else:
        ts["x1"] = wk["x1"]
        ts["x2"] = ts["x1"] + time_col_width

    # 👇 NEW: use "Week" label OCR to clamp x2
    first_week_col_width = (wk["x2"] - wk["x1"]) / 15
```

```
    ts["x2"] = ts["x1"] + first_week_col_width - 30

    y1_pair = find_line_pair("0830", "0900")
    y2_pair = find_line_pair("2200", "2230")
    if y1_pair[0] is not None:
        ts["y1"] = y1_pair[0] - 15
    if y2_pair[1] is not None:
        ts["y2"] = y2_pair[1] + 15
    refined[0] = ts

    # === CourseCell (class 2) ===
    course_boxes = box_df[box_df["class"] == 2]
    cc = course_boxes.iloc[0].copy() if not course_boxes.empty else {}
    cc["x1"] = ts["x2"] + 1
    cc["x2"] = wk["x2"]
    cc["y1"] = ts["y1"]
    cc["y2"] = ts["y2"]
    refined[2] = cc

    return refined

def get_refined_layout_boxes(image: Image.Image, ocr_df: pd.DataFrame) -> dict:
    """
    Given a PIL image, returns refined bounding boxes for layout classes.
    Output: { class_id: {x1, y1, x2, y2}, ... }
    """

    box_df = run_yolo_detection(image)
    return refine_yolo_boxes_with_fallback(box_df, ocr_df)
```