École Polytechnique Fédérale de Lausanne

# Typological Analysis of Venetian Floor Plans through Computer Vision - Final Report

Professor : Frédéric Kaplan / Supervisor : Paul Guhennec

Jaeyi Jeong

06/06/2025

Systèmes de Communication

**Abstract**

This report presents the comprehensive development and experimentation efforts conducted during the DHLab Bachelor Semester Project. The objective was to segment and analyze Venetian architectural floor plans using a combination of traditional image processing and deep learning techniques. Key methodologies included data acquisition through manual scanning, advanced preprocessing with wall extraction and text removal, ground truth creation with patch-based data augmentation, and semantic segmentation using the transformer-based SegFormer model. Post-processing steps integrated hybrid visualizations and logit-based thresholding for enhanced segmentation performance. Furthermore, the project expanded to include vectorization and alignment of geo-referenced data, transforming hybrid outputs into vectorized formats suitable for structured geospatial analysis. Challenges in dewarping and rare class detection were addressed, culminating in a robust and multifaceted segmentation and vectorization pipeline.

# 1 Introduction

The typological analysis of Venetian architectural floor plans is a crucial task for understanding historical patterns in urban development, building usage, and spatial organization. As architectural archives increasingly move toward digital preservation, there is a growing need for computational tools that can automate the analysis of large volumes of scanned architectural documents. This project was initiated to develop a comprehensive, scalable workflow for processing such historical floor plans specifically those of Venice with the goal of extracting meaningful architectural elements such as walls, windows, and stairs.

The motivation for this work stems from the intersection of computer vision and digital humanities. While semantic segmentation techniques are widely used in domains like autonomous driving or medical imaging, their application in architectural heritage studies remains limited. Yet, the potential is immense : by enabling the automated extraction and classification of architectural features, computer vision can empower historians, architects, and urban scholars to conduct large-scale quantitative analyses that were previously infeasible through manual methods.

Venice, with its rich and intricate architectural history, presents an ideal testbed for such an approach. The unique spatial configurations of Venetian buildings and the availability of extensive scanned floor plan archives make it possible to explore architectural typologies computationally. Through a hybrid method that combines classical image processing techniques such as adaptive thresholding, morphological filtering, and text removal with transformer based semantic segmentation (SegFormer), this project aims to bridge the gap between digital tools and humanities research.

Challenges such as rare class detection (particularly for windows and stairs), data imbalance, and large-format inference were systematically addressed. Furthermore, the introduction of vectorization and spatial alignment allowed for the conversion of pixel-based predictions into structured geospatial formats, enabling downstream analysis using tools like QGIS. This enriched workflow not only contributes technically to the field of computer vision but also provides meaningful new pathways for typological and comparative studies in architectural history.

# 2 Methodology

## 2.1 Prelimary Studies

Before starting the practical development, I conducted a literature review and analyzed several relevant papers to understand existing methods and establish a solid foundation for the project. These studies provided insights into architectural floor plan segmentation and computer vision techniques. The research papers reviewed are listed in the References section :

1. Discovering urban block typologies in Seoul: Combining planning knowledge and unsupervised machine learning [1]
2. 2D Floor Plan Reconstruction Using Cool Deep Learning Methods [2]
3. Informed Machine Learning Methods for Instance Segmentation of Architectural Floor Plans [3]
4. Semantic Segmentation in Architectural Floor Plans for Detecting Walls and Doors [4]

.

## 2.2 Data Collection

### 2.2.1 Scanning Setup and Configuration

To build a comprehensive dataset for this project, a total of 281 pages from a Venetian architectural floor plan book were manually scanned. Initially, the scanning was performed at a high resolution of 600 dpi to ensure maximum detail capture. However, it quickly became apparent that this resolution resulted in excessively large files and significantly slowed down the scanning and processing workflow. After evaluating the balance between image quality and processing efficiency, the scanning resolution was adjusted to 400 dpi. This provided a satisfactory level of detail while reducing both scanning time and file size, making the subsequent preprocessing steps more manageable.

The scanning process was meticulous and time-consuming, requiring approximately 16 hours of dedicated work. Each scanned image was carefully checked for completeness and clarity to avoid missing architectural elements. Once digitized, the images were systematically organized into a structured dataset, stored in the `data/` directory of the project repository. This dataset serves as the foundation for all subsequent image processing and analysis stages, enabling reproducibility and consistency throughout the project.

### 2.2.2 Sample Scanned Images

This is a sample of a scanned image :

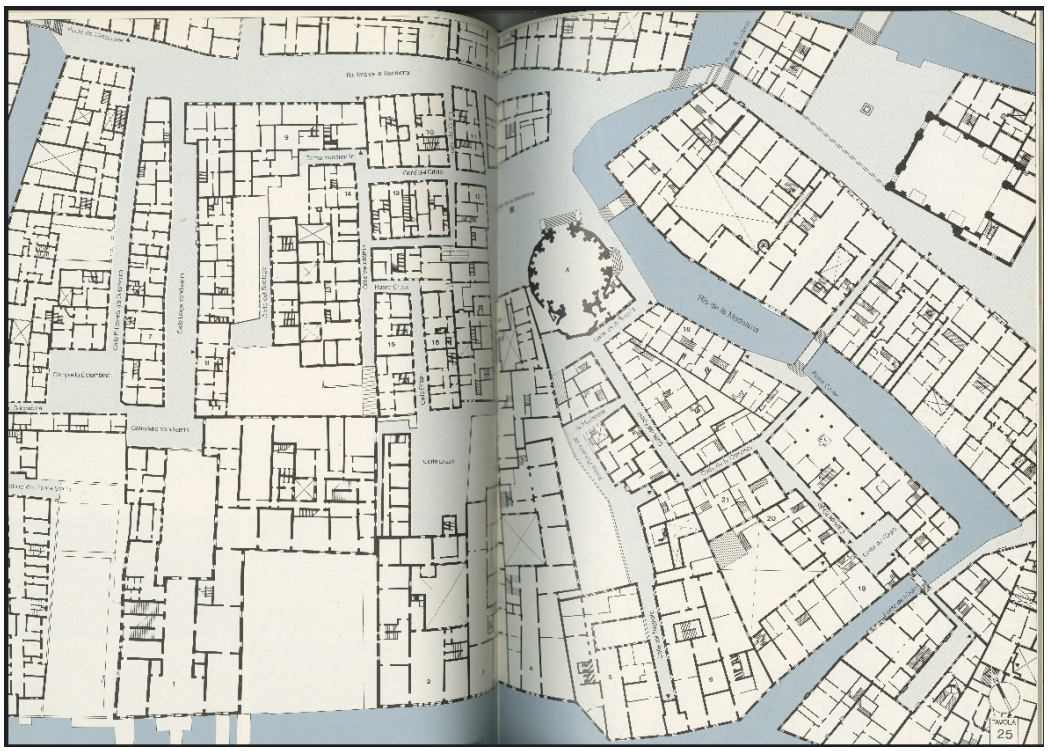


Figure 1: Sample scanned page from the Venetian architecture book

## 2.3 Preprocessing and Wall Extraction

To effectively process architectural floor plans, an image preprocessing and wall extracting script were developed using OpenCV. The goal is to efficiently threshold images while allowing real-time user adjustments to optimize the thresholding value. For the wall extracting part, the goal is to extract only walls hence removing texts, non-walls based on the thresholded images is what I should do.

### 2.3.1 Initial Approach and Limitations

The initial version of the preprocessing script was a straightforward implementation of binary thresholding. The script loaded the image in grayscale, applied a threshold of 150, and saved the result. However, this approach had a significant limitation : To find an optimal threshold value, I had to manually edit the script, rerun it, and check the output repeatedly, which was highly inefficient.

### 2.3.2 Enhancing Interactivity for Threshold Selection

To address this issue, the script was modified to allow real-time threshold adjustment. Instead of modifying the code for each test, the user can now input threshold values interactively until a satisfactory result is found. Additionally, the script was updated so that it only saves the image when the user explicitly presses the 'S' key. This improvement allowed interactive threshold adjustment in real time, saved the processed image only when explicitly prompted, and eliminated the need to manually edit and rerun the script for each threshold test.

### 2.3.3 Handling Display Scaling Issues

During testing, another issue was identified : The displayed images were too large and appeared zoomed-in, making it difficult to evaluate the thresholding results. To resolve this, an image resizing function was imple-

mented that maintains the aspect ratio while fitting the image into a predefined maximum width and height. This function ensured that the displayed images are now properly scaled, preventing the excessive zoom-in effect.

### 2.3.4 Understanding Binary Thresholding `cv2.THRESH_BINARY`)

The binary thresholding technique used in this script follows the logic :

$$\text{if pixel value} > \text{threshold value}, \quad \text{new pixel value} = 255 \quad \text{(white)}$$

$$\text{else}, \quad \text{new pixel value} = 0 \quad \text{(black)}$$

. This operation converts grayscale images into strictly black-and-white images, which simplifies further processing steps. More details on OpenCV's thresholding methods can be found in this reference. [5]

### 2.3.5 Effect of Different Threshold Values

To further refine the process, different threshold values were tested, and their effects on floor plan segmentation were observed :

| Threshold Value | Observed Effect |
|---|---|
| 10 | Nearly all pixels become white (255), producing an almost meaningless output. |
| 100 | Darker regions remain black, while brighter areas turn white. |
| 200 | Only significantly dark areas, such as walls, remain black, while most of the background turns white. |
| 250 | Very high threshold, causing almost all pixels to become black. |

Table 1: Effects of Different Threshold Values

(a) sample image (b) Threshold = 110

Figure 2: Before and After thresholding

### 2.3.6 Overview of Wall Extraction

Wall extraction is a crucial step in processing architectural floor plans, where the goal is to isolate walls from background noise and other non-structural elements. This section documents the iterative development of the wall extraction approach, highlighting challenges encountered and improvements implemented.

### 2.3.7 Initial Approach : Canny Edge Detection and Morphological Operations

The first approach utilized Canny edge detection to identify wall structures within a thresholded image. Morphological operations, including dilation and erosion, were then applied to refine the detected edges. However, the results were not satisfying, as the extracted images showed fragmented and incomplete wall structures.

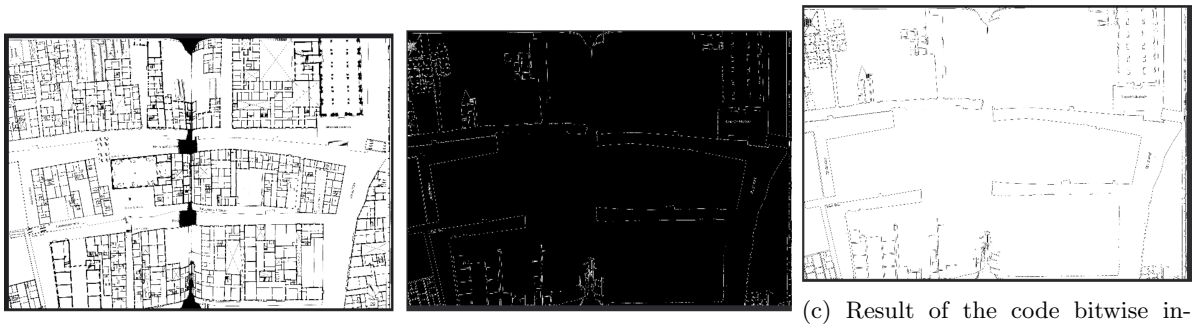
Different steps in the processing pipeline were compared, including the initial Canny edge detection, the result after applying dilation, and the final eroded image. This comparison highlighted the limitations of the initial approach.

(a) Canny Edge Detection (b) After Dilation (c) Final Processed Image

Figure 3: Comparison of Different Processing Steps

### 2.3.8 General Implementation applying Canny Edge Detection and Morphological Operations

The approach involved applying edge detection (Canny), followed by morphological operations to enhance wall-like structures. The walls were identified through contour detection and drawn onto a binary mask. Despite these steps, the initial implementation was not optimal, with incomplete and fragmented wall extractions. Here is the result of the initial attempt :



(a) Thresholded image (b) Result of the code (c) Result of the code bitwise inverted

Figure 4: Comparison of threshold image and the result of the code applying Canny Edge Detection and Morphological Operations

### 2.3.9 Refining the Wall Extraction Process : Parameter Tuning and Challenges

After implementing the initial wall extraction algorithm, several issues became evident. Regardless of whether the walls were represented in black or white, the extracted results were not entirely satisfactory. The loss of details in the central part of the book was likely caused by inappropriate parameter settings.

Among the main factors suspected to contribute to these imperfections were suboptimal threshold values used in the Canny edge detection step, which may have led to missing or fragmented edges. Additionally, the choice of kernel size in the morphological operations appeared to affect the preservation of structural details, especially in

narrow wall segments. Finally, the number of dilation and erosion iterations may have been insufficient, limiting the algorithm's ability to close small gaps or remove noise effectively.

Hence, several parameter adjustments were tested :

**First Adjustment : Increasing Thickness and Canny Threshold**

The first attempt to improve wall extraction involved increasing the thickness of the drawn walls in order to enhance their visual prominence in the processed image. In parallel, the Canny edge detection thresholds were adjusted from the initial values of (50, 150) to (100, 200), aiming to better capture significant architectural features while reducing noise. However, despite these adjustments, the results remained suboptimal several wall segments continued to appear fragmented, indicating that further refinement was necessary.

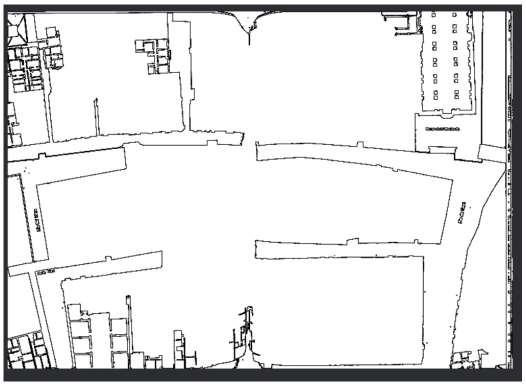


Figure 5: Increased thickness and modify threshold values of Canny detection

**Second Adjustment : Adding an Extra Binary Thresholding Step**

Given that the input images had already undergone initial preprocessing, an additional binary thresholding step was introduced to enhance the contrast further. A fixed threshold value of 100 was applied, which significantly improved the visibility of the wall structures prior to edge detection. This enhancement aimed to strengthen the separation between walls and the background, leading to more reliable edge extraction.

**Third Adjustment : Refining Morphological Parameters**

To achieve a better balance between preserving wall connectivity and minimizing unwanted noise, the parameters of the morphological operations were refined. Specifically, the kernel size was reduced from `(3,3)` to `(2,2)` to better capture finer wall details. Additionally, the number of dilation iterations was limited to one in order to avoid excessive expansion of the detected edges, which could potentially distort the architectural structures.

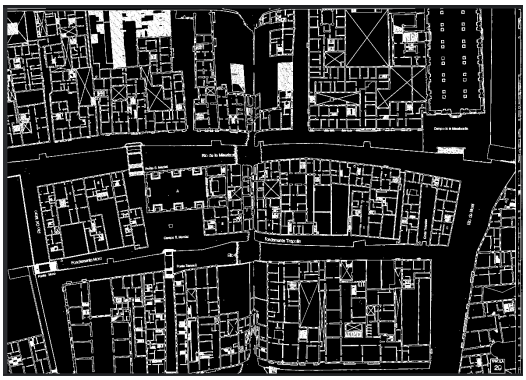


Figure 6: binary thresholding 100,255 and Canny thresholding 100,200

#### 2.3.10 Decision on the Best Parameters at that moment

After testing different configurations, the best-performing at that moment combination was determined :

- Binary threshold at 110 for enhanced contrast.
- Canny edge detection thresholds set to (40, 160) to improve weak edge detection.

This configuration effectively improved the visibility and extraction of wall structures while minimizing loss of important details.

This is the result :

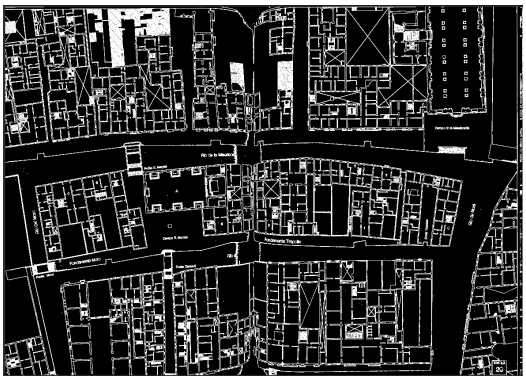


Figure 7: binary thresholding 110,255 and Canny thresholding 40,160

The approach for wall extract was adapted from this link [6],
this [7],
this [8],
this [9], and
mostly this which contains every openCV tips [10].

#### 2.3.11 Enhancing Preprocessing : Removing Rivers and Improving Thresholding

One of the major obstacles in processing historical architectural floor plans lies in the presence of non-structural elements such as rivers, labels, and miscellaneous artifacts. In earlier versions of the pipeline, blue-colored rivers commonly found on the maps were mistakenly interpreted as part of the architectural structure, thereby compromising the accuracy of wall extraction. To address this, the preprocessing pipeline was enhanced to include a targeted removal of these blue features from the input images. Additionally, contrast enhancement was

applied to improve the visibility of structural elements, and a more reliable thresholding method was introduced, with the option to use Otsu's automatic threshold selection. These improvements collectively contributed to a cleaner and more accurate binarization of the floor plans.

**Initial Issues with Rivers and Preprocessing**

Upon reviewing the output of the previous preprocessing steps, several critical issues became apparent. The blue-colored rivers were frequently misclassified as structural features, resulting in inaccurate wall detection. Furthermore, the thresholding procedure proved highly sensitive to contrast variations across different maps, often leading to the loss of fine wall details. Lastly, many binary images still contained small texts and artifacts that persisted through the pipeline, signaling the need for additional filtering mechanisms.

**Solution : Removing Rivers and Improving Thresholding**

The following improvements were implemented in the preprocessing pipeline :

1. **Removing blue rivers :** A color-based masking approach was used to eliminate blue-toned regions.
2. **Enhancing contrast :** Linear contrast adjustment was applied to emphasize important structural features.
3. **Applying Gaussian blur :** Smoothing was introduced to reduce noise before thresholding.
4. **Thresholding with Otsu's method :** Users now have the option to select automatic thresholding to dynamically adjust to different image conditions.
5. **Morphological filtering :** Additional opening operations were applied to remove small artifacts and text labels.

The improved preprocessing pipeline was inspired by :

- Color Detection and Masking in OpenCV
- Thresholding Techniques using OpenCV
- Automating File Processing with OS Module

### 2.3.12 Refinement of Wall Extraction : Addressing Previous Issues

The refinement phase focused on improving the precision of wall extraction while reducing the inclusion of irrelevant elements. Initial attempts suffered from several shortcomings. Contour filling often resulted in the creation of large, unwanted black patches that obscured meaningful structures. In addition, non-structural components such as shadows, text annotations, bridges, and rivers were sometimes erroneously interpreted as walls. The detection of genuine walls was also hindered by suboptimal Canny edge detection thresholds, leading to incomplete wall contours. Lastly, the morphological operations, although intended to reinforce wall connectivity, were occasionally too aggressive, resulting in the removal of critical architectural details.

To overcome these issues, a more refined approach was introduced. This focused on precise contour filtering, adaptive thresholding, and optimal parameter selection.

**Initial Attempt and Challenges**

The first approach involved applying Canny edge detection followed by morphological operations, namely dilation and erosion, to highlight wall-like structures. However, this method presented multiple challenges. Filled regions in the form of solid black patches emerged due to the use of `cv2.drawContours()`, obscuring valid architectural content. Small symbols and text, often part of the map's annotations, were not effectively removed and were wrongly classified as structural components. Additionally, several wall segments were not detected at all, most likely due to inadequately tuned edge detection thresholds. Moreover, the morphological processing introduced new issues: dilation led to the unintentional merging of separate features, while erosion caused the elimination of finer wall elements that were crucial for accurate segmentation.

Figure 8: Result of the first attempt to remove text

**Solution : Improved Wall Extraction Pipeline**

To improve results, several adjustments were implemented :

- Adaptive thresholding was introduced to enhance wall contrast.
- Canny edge detection thresholds were fine-tuned to (40, 160) to better capture structural details.
- Contour hierarchy filtering was applied to exclude inner contours, ensuring only parent contours (true walls) were retained.
- Morphological parameters (kernel size and iterations) were adjusted to balance wall connectivity and noise reduction.

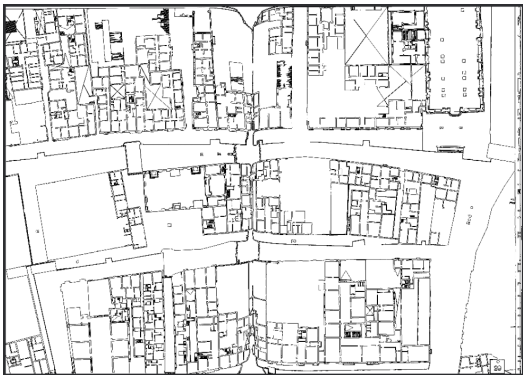


Figure 9: Result of the modified/upgraded version

**Further Refinement : Enhanced Morphological Processing**

Despite the improvements achieved in previous iterations, certain challenges persisted in the wall extraction results. Notably, residual text and symbolic annotations continued to appear in the output, and thin wall structures were occasionally lost due to the overly aggressive application of morphological operations.

To overcome these limitations, the algorithm was restructured to replace Canny edge detection with a purely morphological approach. Specifically, a combination of opening and closing operations was employed to suppress text and non-structural artifacts while preserving the integrity of wall segments. Connected component analysis was then introduced to identify and eliminate small, irrelevant elements based on area thresholds. To improve the suppression of larger text blocks, the kernel sizes used in morphological filtering were increased. Additionally, the number of iterations applied during morphological opening was raised, enhancing the filtering strength and enabling a more effective separation of structural elements from visual noise.

**Key Improvements and Impact**

The revised approach offers several notable advantages :

- **More Effective Text Removal :** Morphological filtering and connected component analysis significantly improve text and small artifact removal.
- **Improved Wall Retention :** Removing Canny edge detection prevents loss of thin walls while keeping major structural elements intact.
- **Faster Processing Speed :** The execution time is reduced from 3 minutes per image to approximately 1 minute.
- **Better Scalability :** The updated approach can process large datasets more efficiently, making it more suitable for real-world applications.

Figure 10: Result of the improved version

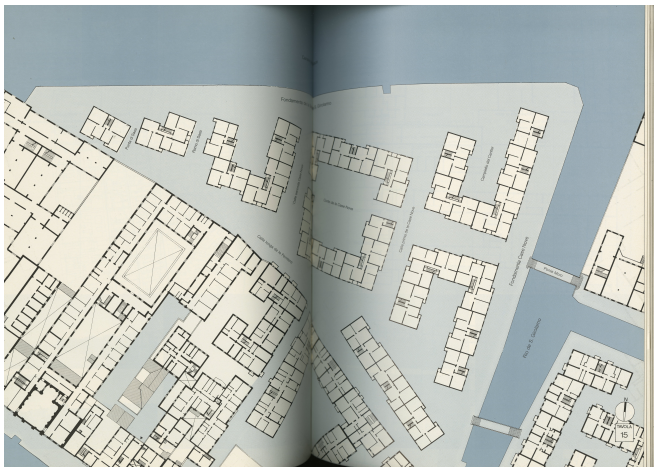
**Using OCR-based text detection to achieve higher extraction accuracy**

To remove printed annotations such as room numbers, street names, or figure labels, we use the EasyOCR library [**easyocr**]. It performs multilingual scene text detection and recognition directly on the input RGB image.

Bounding boxes detected by EasyOCR are filtered with the following criteria to avoid mistakenly removing thin architectural elements. With the first attempt of using OCR, I could remark that it erased well texts however, the problem is, it considered thin walls as texts. Thus, it removed thin walls too. And this is not what we want. Hence, I had to add an aspect ratio filter :

- **Size filter :** Ignore boxes with width or height less than 3 pixel.
- **Aspect ratio filter :** Ignore the long line ( because perhaps it is a wall )
- **Confidence filter :** Only consider detections with confidence scores above 0.3.

The remaining bounding boxes are filled with white rectangles to eliminate text while preserving architectural lines.



(a) Image028

(b) Thresholded img028 with EasyOCR

Figure 11: Comparison between the original img028 and the thresholded one

#### 2.3.13 Partial Thresholding

I have decided to apply immediately the feedbacks I received during my midterm presentation. It is to fix the "center shadowed" problem of the original scanned image

**Brightness Correction via Partial Thresholding**

In scanned book pages, the central fold often appears darker due to page curvature and shadowing, which leads to incorrect binarization of architectural elements. Instead of using complex unwrapping techniques, we adopt a simpler yet effective strategy : region-specific brightness correction combined with partial thresholding.

**1. Region-Based Brightness Enhancement**

To address center-region darkening, I selectively enhance brightness in a predefined square region centered around the middle of the image. This is implemented in the `preprocess_image()` function as follows :

- The input image is converted to grayscale and its contrast is enhanced globally using `cv2.convertScaleAbs` with a gain factor ($\alpha = 1.2$).
- A square mask of size $2 \times$ `center_margin` (default : 700×700 pixels) is created at the center.
- Within this mask, brightness is further boosted using a higher contrast gain ($\alpha = 1.5$) and brightness offset ($\beta = 50$).
- This corrected center region is merged back into the enhanced image.
- Gaussian blur with a $5 \times 5$ kernel is applied to reduce noise before thresholding.

**2. Partial Thresholding**

Following brightness correction, we apply region-dependent binarization in the `apply_threshold()` function :

- The entire image is thresholded using a default value (e.g., $T = 130$), producing `binary_default`.
- A higher threshold (e.g., $T = 180$) is applied to the center region to compensate for remaining shadows, producing `binary_center`.
- A binary mask defines the center region using the same `center_margin`.
- Pixels within the mask are replaced by the high-threshold result, while the outer regions retain the default-threshold values.
- Finally, a morphological opening operation with a $3 \times 3$ kernel removes small artifacts and noise introduced during binarization.

The result looks like this :

Figure 12: Initial attempt of partial thresholding

We can remark that it only worked on the "Center" part of the image. Hence, I have realized that I wrongly set the y coordinate.

**3. Advantages**

This approach preserves fine architectural structures (such as thin walls) across the entire image while avoiding over-thresholding of shadowed central regions. It is computationally efficient and does not require geometric unwarping or complex image rectification.

After the modification of the y coordinate it looks like the following :

Figure 13: Final version of partial thresholding

The final results of preprocessing and wall extraction will be treated in "Results" section.

## 2.4 Ground Truth Labelling and Patch Preparation

### 2.4.1 Labelling Data Using a Photoshop Tool

To prepare training data for deep learning, I manually labeled a floor plan image to create a high-quality ground truth (GT) dataset. Initially, I attempted to use GIMP, a well-known open-source image editing tool, but the learning curve and interruptions from frequent reference searches proved inefficient. As a result, I switched to PenUP, a digital painting app developed by Samsung Electronics, leveraging the Samsung touchscreen and S Pen for smoother and more intuitive annotation.

The labeling process involved :

- Selecting an image from the `../data/` folder.
- Choosing four distinct colors (Red, Blue, Green, Black) to represent Walls, Windows, Stairs, and Background.
- Manually coloring each region to reflect its category.
- Submitting an initial version for supervisor feedback and refining based on suggestions.
- Dividing the annotated image into smaller patches for data augmentation.
- Considering appropriate deep learning models for segmentation training.

This method of creating a single, high-quality labeled image and segmenting it into multiple patches allowed for efficient data augmentation, generating a diverse training set from a single annotated source.

### 2.4.2 GroundTruth creation

I initially labeled the top-left section of `image038` and submitted it for feedback. Afterward, I completed the entire image, addressing feedback to finalize the ground truth. The final version featured clear color coding : red for walls, blue for windows, green for stairs, and black for background.

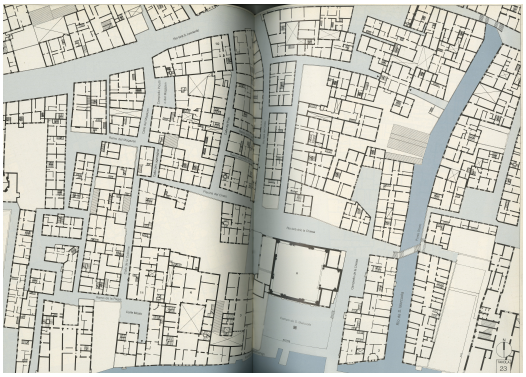


Figure 14: Original Image

Figure 15: Initial Annotation

Figure 16: Ongoing Progress

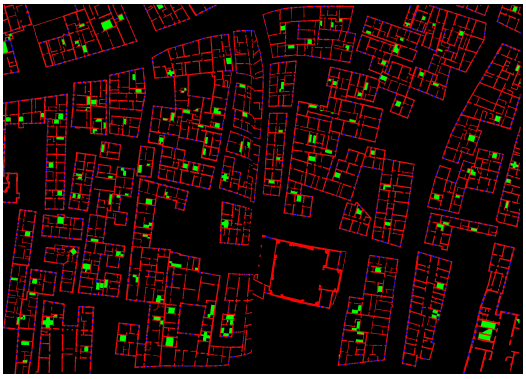


Figure 17: Final Annotated Image

#### 2.4.3 Patch Preparation and Challenges

To construct a suitable dataset for training, both the original images and their corresponding ground truth (GT) segmentations were divided into 256×256 pixel patches. However, this preprocessing stage presented two notable challenges. First, the initial conversion of RGB ground truth images into numerical class labels encountered difficulties due to improper color mapping during the resizing process, which led to inconsistencies in label interpretation. Second, the resulting label patches were stored in a single-channel format, containing integer values (0, 1, 2, 3) corresponding to different semantic classes. When visualized directly, these label images appeared entirely black, making it difficult to verify their correctness without applying a custom colormap.

#### 2.4.4 Solution and Refinement

To resolve these issues, I introduced color mapping tolerance (within 5 RGB levels) and debugging steps. I scaled label values for better visualization and confirmed correct mapping to classes.

Subsequently, I converted single-channel label patches into RGB images for visualization, displaying clear black (background), red (walls), green (stairs), and blue (windows) regions. This approach produced 280 high-quality patches for training.

#### 2.4.5 Experiment with Larger Patch Size

To further enhance training diversity, I experimented with 512x512 patches (stride of 256), producing 54 larger patches. Although this reduced the total number of patches, it provided an opportunity to analyze performance with varying patch sizes. This modification was discussed with my supervisor to determine the optimal patch configuration and at the end we chose 256x256 patches.

The results for patch separations will be displayed in "Results" section.

### 2.5 Color Correction

#### 2.5.1 Problem with Hand-Annotated GT Images

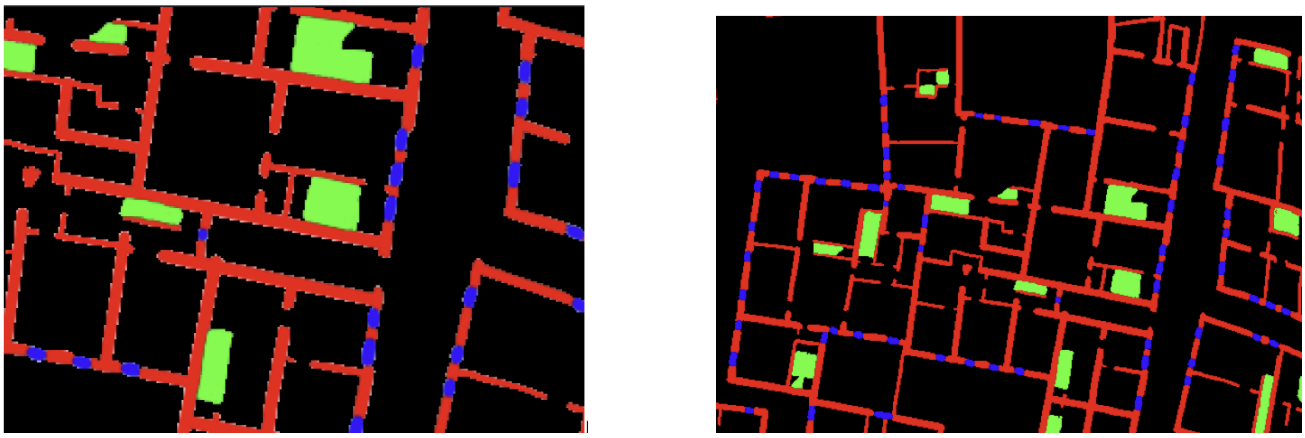
In our dataset, ground truth (GT) images were manually annotated using tools like *PenUp* on a tablet. Although visually accurate, this method introduced subtle color noise due to brush blur effects. For example, pixels intended to represent windows with a color of `(0, 0, 255)` (pure blue) often had nearby pixels with approximate values such as `(0, 0, 227)` due to anti-aliasing or soft brush edges. A zoomed-in view of `gt038.png` revealed visible artifacts like light blue or white-ish pixels that deviate from the defined class palette.

These fuzzy pixels posed a problem for semantic segmentation models, as label classes were no longer strictly defined and consistent.

### 2.5.2 Color Cleaning via KD-Tree Approximation

To resolve this issue, we implemented a post-processing step that cleans the GT images by snapping each non-background pixel to its nearest color in the predefined class palette using **KD-Tree**-based nearest neighbor search.

- **Class Colors Used** :
  - Red: `(255, 0, 0)` for **walls**
  - Green: `(0, 255, 0)` for **stairs**
  - Blue: `(0, 0, 255)` for **windows**
  - Black: `(0, 0, 0)` for **background** (preserved as-is)
- **Method** : We treat each pixel as a 3D RGB vector, and for every pixel (except black), we find the closest class color in Euclidean RGB space using `scipy.spatial.KDTree`.
- **Outcome** : The result is a cleaned GT image where each pixel is guaranteed to match one of the predefined classes, removing all unintended color fuzziness.



(a) Original GT (with fuzzy pixels)

(b) Cleaned GT (KDTree-mapped)

Figure 18: Comparison between the original ground truth and the cleaned version. Color noise near class boundaries is eliminated after KDTree cleaning.

**References and Acknowledgments** This section of the work refers to valuable online resources and articles including [11], [12], [13], and [14].

## 2.6 Data Augmentation

To improve the diversity of training data and reduce overfitting, I applied various image augmentation techniques using the Albumentations library. Augmentations were applied to both input images and their corresponding labels simultaneously, ensuring spatial consistency.

### 2.6.1 Applied Techniques

I defined a transformation pipeline with the following components :

- **Horizontal Flip** (50% probability)
- **Vertical Flip** (50%)
- **Random 90° Rotation** (50%)
- **Brightness/Contrast Adjustment** (20%)
- **Gaussian Noise Addition** (20%)

But after, I removed Brigthness, Gaussian Noise because it disturbed training since it modified the RGB values.

### 2.6.2 Balancing Data Variation with Probabilistic Transformations

Applying all data augmentations with a probability of 1.0 can lead to excessive or unrealistic transformations, which may distort the original features of the architectural plans. To mitigate this, probabilistic transformations are employed to introduce sufficient variation while preserving the semantic integrity of the data. This balanced approach enhances model generalization without compromising data authenticity.

**Details :** The goal of augmentation is to introduce meaningful variety in the training data, helping the model generalize and avoid overfitting. However, if I apply all transformations every time, the resulting images may diverge too much from the original semantics. This may confuse the model or degrade learning accuracy, especially if the ground truth (GT) no longer matches the transformed image.

Therefore, I use probability thresholds (e.g., `p=0.5`, `p=0.2`) to apply each transform occasionally but not always, maintaining a balance between variability and fidelity.

**References and Acknowledgments** This section of the work refers to valuable online resources including [15], [16].

## 2.7 Training the model

### 2.7.1 Motivation

Our aim is to train a semantic segmentation model to identify architectural components such as walls, stairs, and windows from floor plan images. To achieve this, I use a pretrained transformer-based model **SegFormer** provided by HuggingFace, and fine-tune it on our custom dataset.

### 2.7.2 Dataset Preparation in Colab

First, I used Google Colab's T4 GPU for training. The dataset was uploaded as a ZIP file containing augmented patches :

- `aug_inputs/` – RGB input patches
- `aug_labels/` – Corresponding ground truth segmentation masks

### 2.7.3 SegFormer Training Pipeline

I fine-tuned the pretrained SegFormer model `nvidia/segformer-b0-finetuned-ade-512-512` using our data. The class labels are derived from RGB values :

- (0, 0, 0) : Background
- (255, 0, 0) : Wall
- (0, 255, 0) : Stairs
- (0, 0, 255) : Window

The RGB masks were converted to class-index masks via a mapping function.

### 2.7.4 Model Training Code

the model training code is available at ...

### 2.7.5 Benefits of Using a Pretrained SegFormer Model

Using a pretrained model such as `nvidia/segformer-b0-finetuned-ade-512-512` allows the network to benefit from prior knowledge acquired on large-scale, diverse datasets. This transfer learning approach significantly accelerates convergence and improves generalization, particularly when the target dataset is limited in size or domain-specific. By avoiding training from scratch, the model starts with well-initialized parameters, thus requiring fewer epochs to achieve meaningful results.

### 2.7.6 Problems and Solutions

During this coding part, I have faced some problems but those were all fixed now. Let me introduce them :

**Problem 1 : WandB Logging Error**

- **Error :** WandB forced login / API error due to automatic logging.
- **Fix :** Disabled wandb using:

```
    os.environ["WANDB_DISABLED"] = "true"

```

- **Additionally :** Set `report_to="none"` inside `TrainingArguments`.

**Problem 2 : Shape Mismatch During Metric Computation**

- **Error :**

```
ValueError: operands could not be broadcast together with shapes (224,64,64) (224,256,256)
```

- **Cause :** The model output logits were downsampled (e.g., 64x64), but ground truth labels were 256x256.
- **Fix :** Resize predicted logits before accuracy computation :

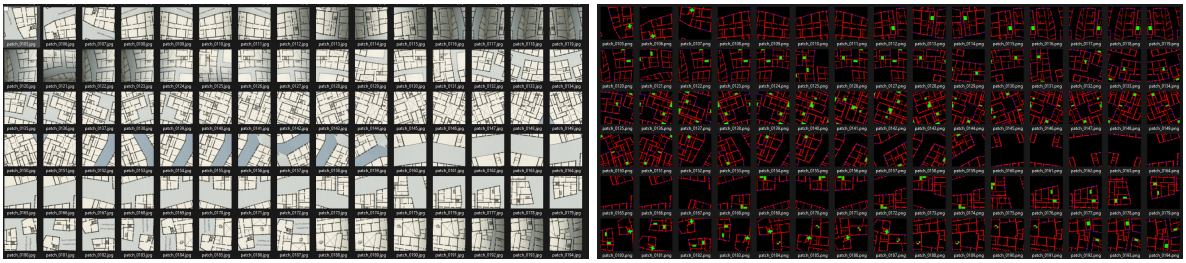
```
preds = torch.from_numpy(preds).unsqueeze(1).float()
preds = F.interpolate(preds, size=labels.shape[1:], mode="nearest")
preds = preds.squeeze(1).long().numpy()

```

- **Integrated into :** `compute_metrics()` function.

### 2.7.7 Use higher variant

For better performance, I used A100 GPU in Google Colab in order to apply b4 variant for training the model. (Note that before, I used b0 variant which is very lightweight). The result is much better than before.

Here is the result of the inference after using b4-variant :



(a) augmentated input images (b) augmentated label images

The training result will be showed in "Results" section

**References and Acknowledgments** This section of the work refers to valuable online resources including [17], [18], [19], [20], [21], [22], [23]

## 2.8 Inference

### 2.8.1 Initial Inference Attempt (Failed)

- **Problem :** The model was trained on 256×256 patches, but inference was run on a full-size image (e.g., 2707×1920).
- **Consequence :** The model failed to generalize properly, and output was meaningless or distorted.
- **Explanation :** Since the model only saw patches during training, it struggles with large-scale spatial context it has never seen. Additionally, some transformers are sensitive to input size, and even with resizing the mismatch breaks spatial alignment.

This code is the very initial attempt of my Inference code. It failed, and the explanation of the failure is the following :


Figure 20: Result of initial attempt of inference

### 2.8.2 Second try for Inference code (Failed)

After the first failure, I have tried to enhance the code and I obtained this result :


Figure 21: Result of second attempt of inference

**Analysis and Observations**

- **Result :** The segmentation now works correctly on full-size floor plan images by using patch-based inference.
- **Observation :** A large portion of the predicted mask is labeled as `Background` (black, class ID 0).

**Possible Reasons for Too Much Background**

1. **Label imbalance during training :** The background area might dominate the training dataset.
2. **Model bias :** Because the model was trained mostly on patches where background pixels are common, it might over-predict class 0.
3. **Patch edge effect :** When the model predicts patches independently, context across patch boundaries is lost.
4. **No overlap/averaging :** STRIDE = PATCH_SIZE means no overlap; soft blending of overlapping predictions might help.

Hence, I have verified the class distribution for debugging and it gave as output : Counter(np.uint8(0): np.int64(64911436), np.uint8(1): np.int64(6622547), np.uint8(2): np.int64(1027924), np.uint8(3): np.int64(838413))

The remark was that background(class 0) is way more pixels than others. The order is : Background $>>>>$ Wall $>$ Stairs $>$ Windows

### 2.8.3 Third try for Inference code (Not Failed, but lacking)

By realizing the possible error, the code was improved in order to recover from class imabalance. The aftermath has gotten better than before however, it did not seem satisfying.

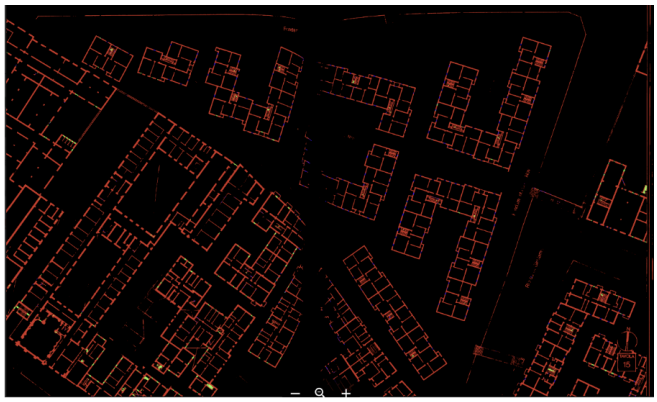


Figure 22: Result of the third attempt

**Problem : Underperformance on Minority Classes**

During inference, it was observed that certain classes, particularly **Windows (blue)** and **Stairs (green)**, were often misclassified or even entirely missed. In some cases, windows were labeled as stairs, indicating confusion between rare classes.

Upon inspection, this was attributed to a severe **class imbalance** in the dataset : walls and background occupied the majority of pixels, while windows and stairs were only a small fraction (e.g., under 1%).

**Solution : Weighted Cross-Entropy Loss** To counter this, I increased the loss weight for underrepresented classes. A custom trainer class was created to inject class weights into the CrossEntropyLoss used by HuggingFace's Trainer API.

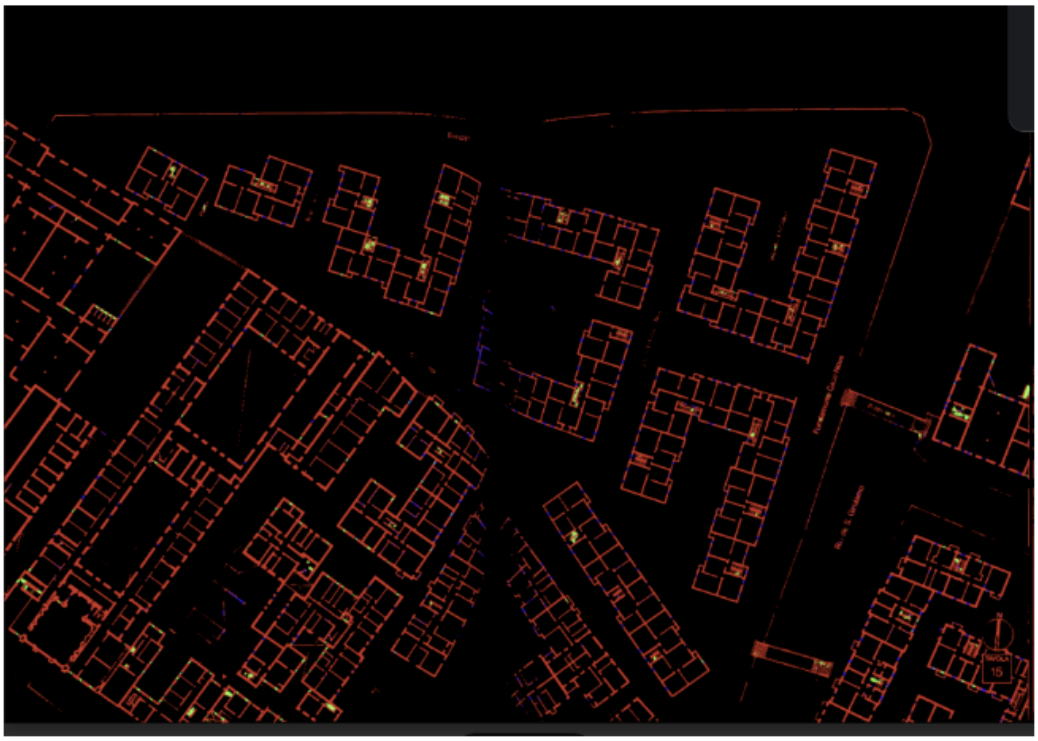


Figure 23: Result by applying weighted training code

**Remark**

Now we have something that looks agreeable. However, it doesn't seem that good. So I tried with many other weights values. And these weights gave the best result at that moment after several heuristic trials :

```
weights = torch.tensor([0.3, 1.2, 20.0, 13.1]).to(logits.device)
```

. However, this is not the final version of the code. Hence, the final weights decision will be treated in "Results" section.

### 2.8.4 Verifying mean logits value per class before applying logit thresholding method

For better inference result, I have applied a logit thresholding method, and before starting it, I needed to find out the mean logit values for each class. And I obtained :

- BackGround mean logit : 5.793
- Wall mean logit : -3.644
- Stairs mean logit : -3.842
- Window mean logit : -4.077

With this values, I could add this logit thresholding in our inference code and get an enhanced result :

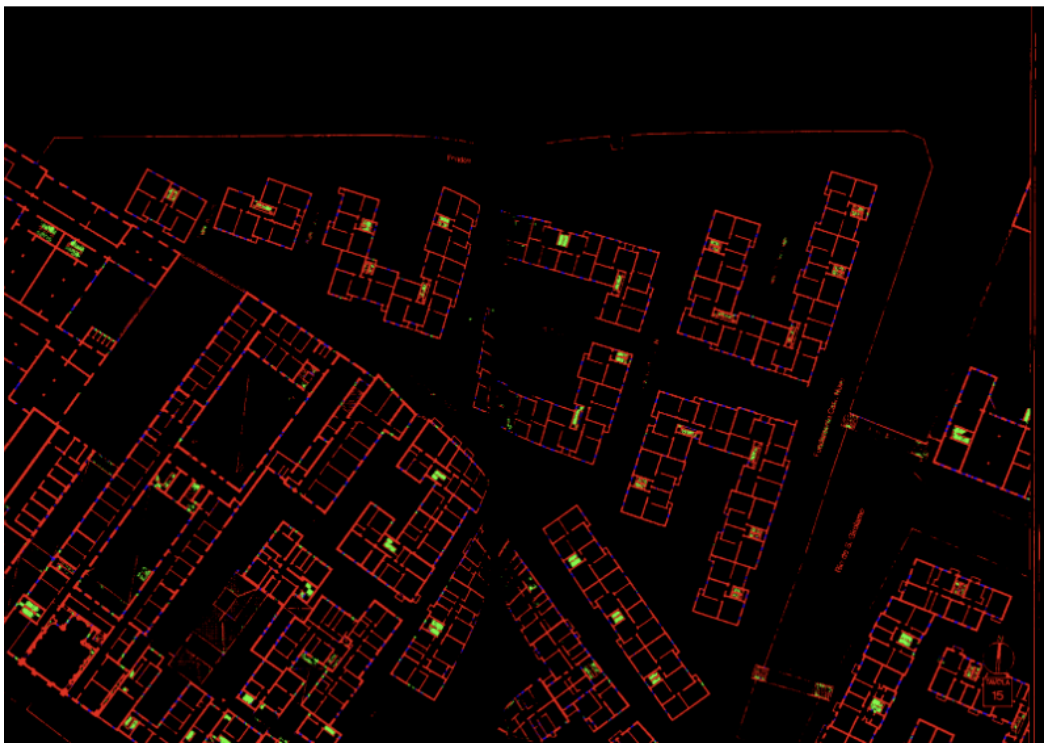


Figure 24: Logit thresholding Initial attempt

We have better result now for stairs. However, this is not satisfying, hence, at the end I slightly modified the condition code for logit thresholding and the final results will be mentioned in "Results" section. To summarize the idea of logit thresholding is the following : Set a certain threshold for each class' logits because, we have verified that most of the logits are the background class'. So when we say that we use a method that only takes the maximum value, it will mostly take the background class' logits whereas in reality, other weaker classes (such

as windows and stairs) should be taken. Hence, instead of everytime taking the "argmax", we set a certain threshold to make it fair for every class and predict the correct, corresponding class.

The original threshold value of -3.7 used for identifying stairs in the logit map lacked a clear justification. To address this, I conducted a statistical analysis of the logit distributions based on ground truth annotations and revised the thresholding strategy accordingly.

The approach began by isolating pixel locations corresponding to specific semantic classes, such as stairs and windows, directly from the ground truth mask. Using these filtered pixel positions, I extracted the corresponding logit values from the model's averaged logit output (pred_mask_avg). With these values, I calculated several statistical indicators, including the mean, median, and the 10th and 90th percentiles of the distribution.

This empirical analysis enabled the determination of more informed and justifiable threshold values, replacing arbitrary constants like -3.7 with thresholds derived from the actual distribution of model outputs.

The result is :

```
GT Logit statistics based on GT for 'background'
Pixels:  28601736
Mean:    5.792279
Median:  6.030441
10%:     4.587292
90%:     7.120522

GT Logit statistics based on GT for 'Wall'
Pixels:  3877476
Mean:    -3.664688
Median:  -4.122149
10%:     -5.806252
90%:     -1.701077

GT Logit statistics based on GT for 'Stairs'
Pixels:  591272
Mean:    -3.833387
Median:  -3.574126
10%:     -5.256207
90%:     -2.894204

GT Logit statistics based on GT for 'Window'
Pixels:  492157
Mean:    -4.062434
Median:  -3.740246
10%:     -5.775251
90%:     -2.958014
```

The following steps have been completed to analyze and apply class-wise logit statistics based on ground truth (GT) data :

- Pixel Filtering Based on Ground Truth : I created binary masks by selecting pixels that belong to each class using the condition gt_mask == class_id. This enabled class-specific analysis of model predictions.
- Extraction of Logits at GT Pixel Locations : For each class, we extracted the corresponding logit values from the model's averaged output (pred_mask_avg) using the filtered masks. The extraction was performed using the expression pred_mask_avg[:, :, class_id][mask].
- Statistical Analysis of Class-wise Logits : The extracted logits were analyzed statistically. Specifically, we computed and reported the mean, median, 10th percentile, and 90th percentile for each class. This analysis provided a detailed understanding of the logit distribution for each semantic category.

- Threshold Determination Based on Statistical Analysis : Based on the statistical results, particularly the median values, we determined appropriate threshold values for each class. This approach ensures that threshold settings are empirically grounded rather than arbitrary.

- Application of Thresholds to Inference : During inference, instead of simply applying an argmax operation across the logits, we combined the logit values with the class-specific thresholds for classification. This method aims to improve classification accuracy by considering the statistical characteristics of each class.

Now, in our analysis pipeline, we needed pixel-level logit outputs rather than simple hard class predictions (e.g., argmax results). Since the input image is larger than the model's input size, we divided it into overlapping patches, ran the model on each patch, and accumulated logits. By averaging logits over overlapping regions, we obtained a smooth and accurate logit map across the entire original image.

This averaged logit map (pred_mask_avg) is essential for the subsequent segmentation analysis, where we statistically examine the distribution of predicted logits per class and apply refined thresholding strategies.

Now the next step's code analyzes the performance of a semantic segmentation model by comparing the model's predictions against the ground truth (GT) annotations.

The key steps are :

- Ground Truth Processing : The GT mask image is loaded and converted into class labels (e.g., Background, Wall, Stairs, Window) based on predefined RGB color mappings.

- Prediction Mask Processing : The model's predicted mask is loaded, resized to match the GT resolution, and similarly converted into class labels.

- Logits Loading : The saved per-pixel logits (predicted confidence values for each class) are loaded and resized to the GT resolution.

- Performance Analysis : - A confusion matrix is computed to measure how well the model classified each class.

  - The overall accuracy is calculated.

  - For each class, logit statistics (mean, median, 10th and 90th percentiles) are computed, but only for correctly predicted pixels. This provides deeper insights into how confidently the model predicts each class.

The result of it gives this :

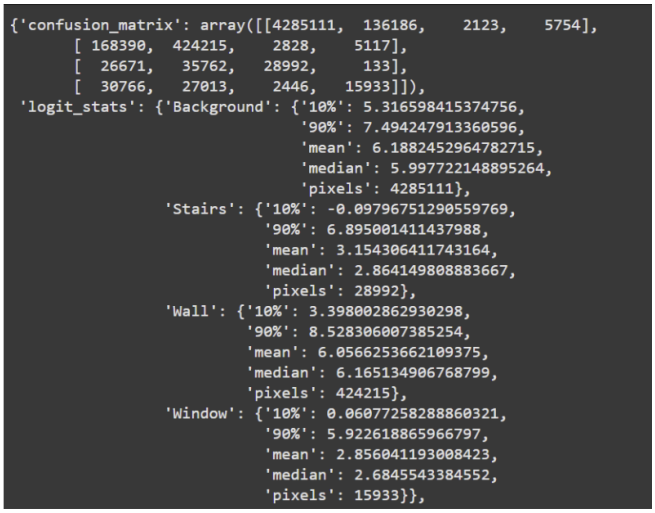



Figure 25: Logit stats

Based on this, I chose the thresholding for logits per classes. Now I have a reasonable evidence to prove why I chose a certain threshold.

#### 2.8.5 Metrics

Computing precision, recall, and F1 score for each class individually is an important step.

For each class, it computes :

- Precision : How many of the predicted pixels for the class are actually correct.
- Recall : How many of the actual pixels for the class are successfully detected.
- F1 Score : The harmonic mean of precision and recall, providing a balanced metric.

By doing this computation, a detailed diagnosis by analyzing precision, recall, and F1 score per class identify :

- Classes with high false positives (low precision)
- Classes with many missing detections (low recall)
- Classes that are overall well-balanced (high F1 score)

If recall is low for "Stairs," we may need more training samples for stairs.

If precision is low for "Window," the model may need better feature learning or threshold tuning.

So the result was :

```
Background (class_id=0)
Precision: 0.9499
Recall:    0.9675
F1 Score:  0.9586

Wall (class_id=1)
Precision: 0.6807
Recall:    0.7064
F1 Score:  0.6933

Stairs (class_id=2)
Precision: 0.7967
Recall:    0.3167
F1 Score:  0.4532

Window (class_id=3)
Precision: 0.5915
Recall:    0.2092
F1 Score:  0.3091
```

**One can remark :**

For all images except img038, ground truth annotations are not available. As a result, performing quantitative evaluations such as pixel-wise accuracy or confusion matrix analysis is not feasible for these images.

Instead, I focused on qualitative comparison of segmentation results under different threshold settings. By visually inspecting the output masks, we assessed how thresholds based on various statistical criteria (e.g., median, 90th percentile) influence the segmentation performance.

**This is another remark :**

Using the 90th Percentile (Higher Threshold) : Its effect :

- Precision increases
- Recall decreases

Interpretation :

- Only pixels with very high logits are classified as "Stairs."
- This leads to fewer false positives, but also misses more true stair pixels.

When to use :
Suitable for use cases where visual accuracy is critical, such as map-based visualization, and minimizing false detections is prioritized.

Using the 10th Percentile (Lower Threshold)
Its effect :

- Recall increases
- Precision decreases

Interpretation :

- Even slightly confident predictions are accepted as "Stairs."
- This results in more detections, but at the cost of higher false positives.

When to use :
Useful in situations where maximizing detection coverage is important, such as when manual post-processing or review will follow.

The final choice of value for logit thresholding and its results, metrics will be discussed in "Results" section.
For the final logit thresholding, I did :

- For Background and Wall classes : Use the median logit threshold, which provides a balanced trade-off between precision and recall.
- For Stairs and Window classes : Use the 10th percentile threshold, aiming to increase recall and capture more instances of these minority or visually subtle classes.

## 2.9 Dewarping

### 2.9.1 A methodology that I planned to do

**Idea**

The plan is the following :

1. Split the image in two
2. Set a "margin" where the "folding" appears

3. Try to dewarp where there is the folding issue

4. Bind the three parts : Left, Middle (dewarped), Right

Hence, the result is :

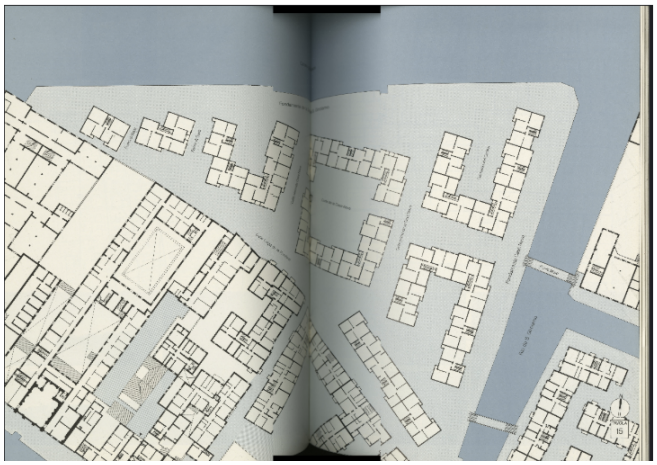


Figure 26: Dewarped page

### 2.9.2 Using Deep Learning to solve this issue

Since all the methods I use didn't work properly, I have decided to use Deep learning based model to fix this.

**DewarpNet**

Reference : `https://github.com/cvlab-stonybrook/DewarpNet` I used google colab to run the code and the result is the following :

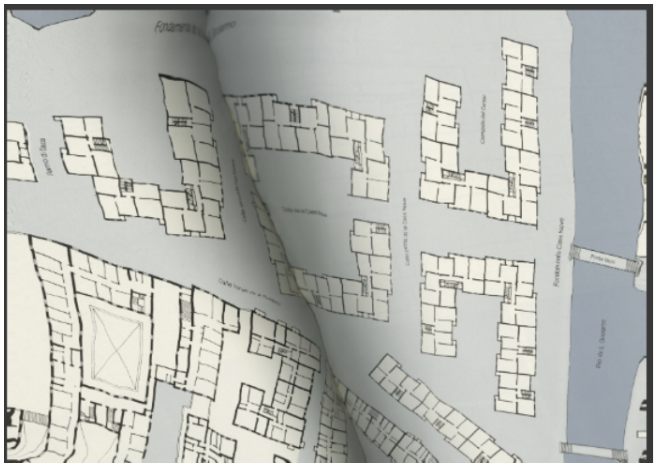


Figure 27: Dewarpnet's result

As we can observe, it is not working as expected. I started doubting at this moment if this task is actually something "easy" to do. Perhaps flattening, or dewarping texts in a book page might be simpler to do it since a lot of models for this use exist already (by using OCR etc.). However, I couldn't find any information of how I can dewarp a book scanned image that does not contain text. Still, I tried to figure it out, and found this latest model for my case :

**DocRes**

Reference : `https://github.com/ZZZHANG-jx/DocRes#` For this case, there was already a way to run the inference code easily by clicking on this hugging face link : `https://huggingface.co/spaces/qubvel-hf/documents-restoration`

Hence the result gives :

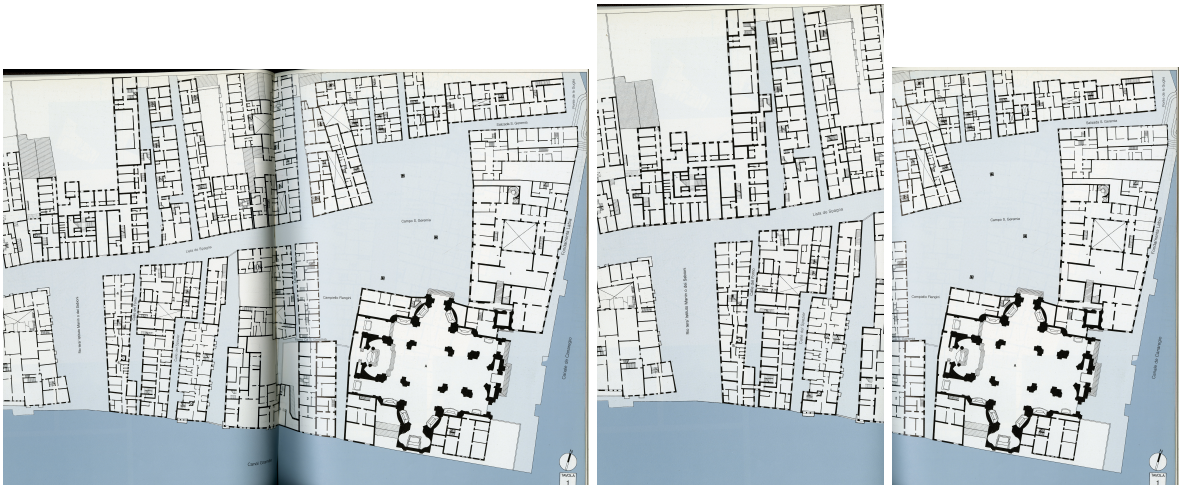
Figure 28: DocRes result

Therefore, this task is one of a challenge I have faced and will be mentioned in "Discussion" section.

## 2.10 Data modification

### 2.10.1 Cutting the original data into two pieces

Just like the title of the subsection, instead of using dewarp for original samples, we have discussed and decided to cut all the datas into two pieces. Then, remove the "curved" part. Here is a comparison of before and after for better comprehension :



(a) Original image, img002.jpg (b) img002.jpg's left part (c) img002.jpg's right part

Figure 29: Before and After cutting + removing the curved part

I did this to all original samples. Hence, from now, we use this new dataset for the following steps. The

inference code's result, OpenCV based code's result have been updated since the dataset has been modified.

## 2.11 Combine OpenCV and Segformer

### 2.11.1 Hybrid method in order to create a segmentated floor plan

The next task is to combine the OpenCV's result (wall extracted) and the Segformer result (stairs, windows)
A simple color masking code worked well.
Here is the result :

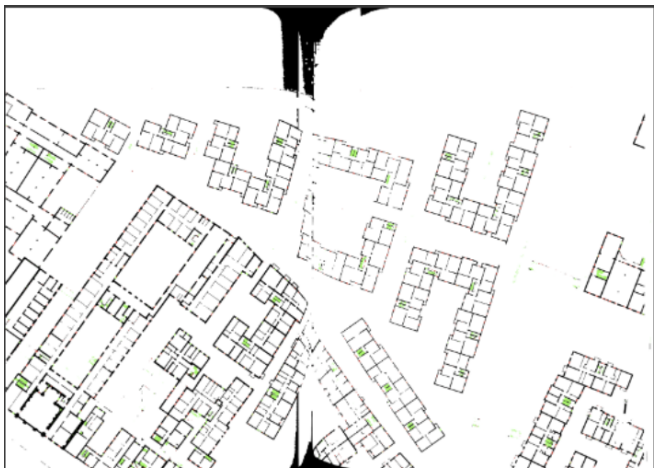


Figure 30: Combining result (initial attempt)

As we can visualize, the result is quite satisfying, however, we only see the green color (stairs). Hence, after few improvements in the code, the result is exactly what we want :

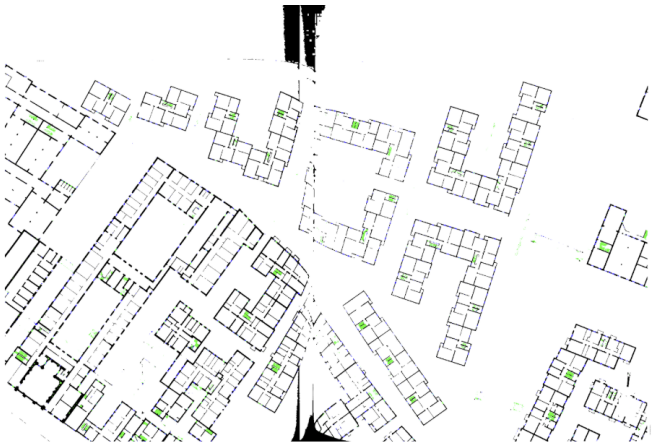


Figure 31: Combining result (final attempt)

## 2.12 Overlay code

### 2.12.1 Overlaying code

This task is to overlay the original sample with wall extracted data to verify which part contains loss of information in wall extracted data. The initial code was not the best code since my supervisor wanted the wall to be in red color so that it is much efficient and easier to notice the loss. Hence, the final version provides an image like the following :

Figure 32: Overlay result

This code was done before the modification of the dataset, hence the result is still a "full page" looking image instead of cut in half image.

**References and Acknowledgments** This section of the work refers to valuable online resources including [24].

## 2.13 Vectorization

### 2.13.1 Initial Attempt : Basic Contour-Based Vectorization

The initial approach focused on extracting vector representations from a segmented image where different classes (walls, stairs, windows) were color-coded in red, green, and blue (BGR format). The process involved :

- Loading the input image using OpenCV.
- Defining exact BGR color values for each class (e.g., (0, 0, 255) for walls).
- Creating binary masks for each class using cv2.inRange to isolate pixels matching the specified color.
- Detecting contours in each mask using cv2.findContours with the RETR_EXTERNAL mode to capture outer boundaries.
- Converting contours to Shapely Polygon objects, ensuring validity with a zero-distance buffer if needed.
- Storing polygon geometries and class properties (class ID and label) in a GeoJSON FeatureCollection.
- Saving the results to a GeoJSON file.

**Issues**

The output GeoJSON was suboptimal, as it exhibited incomplete or inaccurate feature detection. One major issue was a mismatch in color assumptions the algorithm expected walls to be red, but the hybrid images actually used black, which led to missed detections. Additionally, the use of cv2.inRange without any tolerance for slight color variations caused further detection failures, particularly in regions affected by compression artifacts or lighting inconsistencies. The absence of a visualization step made it challenging to debug and understand why certain features were not correctly identified. Furthermore, all classes were processed using the same generic contour

detection method, which was not well-suited to capture the elongated and linear characteristics of architectural walls.

The outcome is :

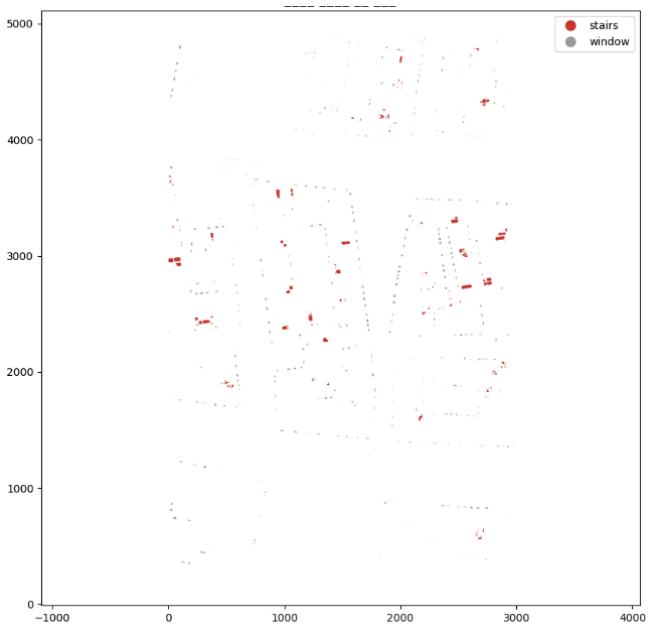



Figure 33: Plot of the initial geojson outcome

### 2.13.2 Corrected Colors and Visualization-Enhanced Vectorization

The final attempt successfully addressed the critical color mismatch issue and incorporated visual debugging tools for better transparency in the pipeline. The color definitions were corrected based on the realization that hybrid images used black for walls, green for stairs, and blue for windows. Specifically, walls were represented as (0, 0, 0) with a $\pm 10$ tolerance to account for slight variations. The pipeline was simplified by abandoning the Hough Transform and instead applying a unified contour-based polygon extraction method for all classes, which ensured both consistency and maintainability. Additionally, Matplotlib-based visualizations were integrated to display the binary masks for each class, enabling easy verification and debugging of mask generation. The output remained in the GeoJSON FeatureCollection format, with each polygon annotated with a class ID and a descriptive label such as "wall", "stairs", or "window".

Hence the last attempt successfully produced a GeoJSON file containing accurate vector representations of walls, stairs, and windows. The visualization confirmed proper mask generation, and the simplified contour-based approach proved effective for all classes. This script was deemed suitable for processing all images in the dataset.

Here is an example plot of succeeded vectorization :

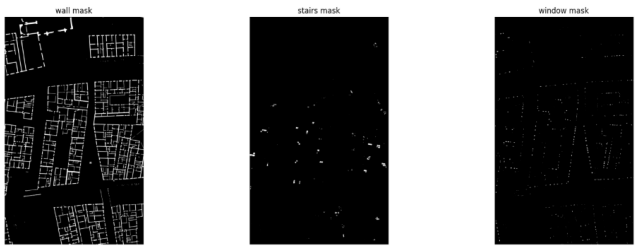



Figure 34: Plot of the final geojson outcome

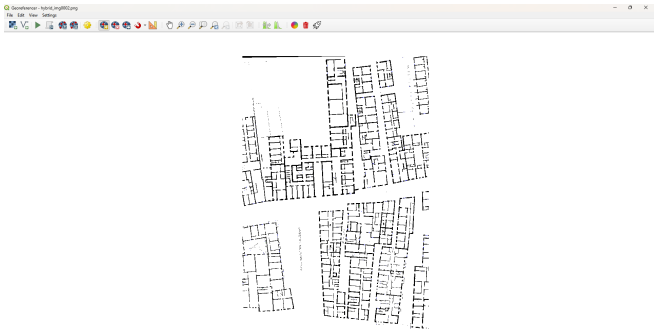


Figure 36: Georeferencer screen

**References and Acknowledgments** This section of the work refers to valuable online resources including [25], [26], [27], [28], [29].

## 2.14 Georeferencing

### 2.14.1 Georeferencing using QGIS

I used QGIS to do this task.

First, once opened QGIS, I add the footprint layers that my supervisor has provided (vector layers) which contain : Streets, Canals, and Buildings in Venice.

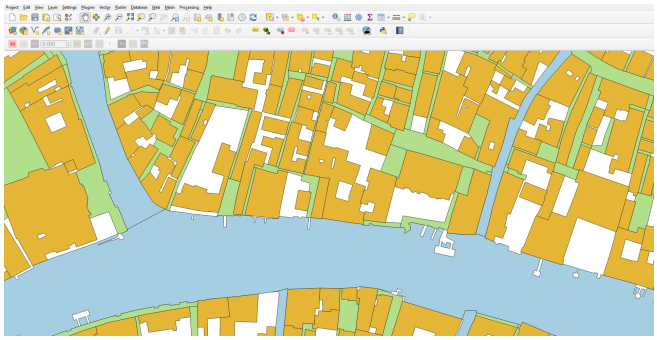


Figure 35: QGIS screen with footprints

Second, I open the georeferencer and add a raster layer, which is the "hybrid" images that I created :
Third, I select 4-6 GCP points and map them into real-world coordinate :

Figure 37: Selection of GCP points

Then, I can observe that the hybrid image has been well "georeferred" :

Figure 38: Result of georeferencing

Finally, we save these results into .tif files.

Once saved all the .tif files correctly, we apply the vectorization code again on those .tif files in order to get a vectorized + georeferred .geojson files.

### 2.14.2 Divide geojson files per buildings

In the georeferred geojson files, there are many buildings and they all have a "EDIFICI ID". Hence, I saved all buildings into a single geojson. When I upload this as a vector layer in QGIS, it looks like this :

Figure 39: EDIFICI ID 1530's geojson

### 2.14.3 Selection of best building geojson

Since there are several buildings that are repeated/overlapped, I had to choose the one that has the best coverage. Thus, the code automates the selection of the best vectorized GeoJSON clips for building footprints by evaluating their coverage against a master dataset of building outlines. It processes multiple GeoJSON files, each representing segmented features (e.g., walls, stairs, windows) for a specific building, and identifies the clip with the highest coverage within predefined thresholds. The selected clips are then copied to an output directory for further use. This process ensures that only the most representative vectorized data is retained for each building, optimizing downstream applications such as urban planning or architectural analysis.

The key steps are outlined below :

**1. Configuration**

- Input Data : A master GeoJSON file containing building footprints, indexed by a unique building identifier (EDIFI_ID), and a directory containing vectorized GeoJSON clips for individual buildings.
- Output Directory : A destination folder where the selected clips are saved.
- Coverage Thresholds : Minimum (5%) and maximum (30%) coverage ratios to filter clips, ensuring they represent a reasonable portion of the building footprint without excessive overlap or noise. (This threshold was found by a heuristic way)
- The output directory is created if it does not already exist, ensuring robustness.

**2. Loading Building Footprints** The master building footprints are loaded into a GeoDataFrame using GeoPandas, with the EDIFI_ID set as the index for efficient lookup. The total area of each footprint is computed and stored as a series, mapping each building ID to its footprint area. This serves as the reference for evaluating clip coverage.

**3. Processing GeoJSON Clips** The script iterates over all GeoJSON files in the input directory, each representing a vectorized clip for a specific building. For each file :

- The building ID is extracted from the filename (expected format includes "ED" followed by the ID).
- The clip is loaded as a GeoDataFrame. Empty or invalid files are skipped.
- The clip's geometry is intersected with the corresponding building footprint to calculate the overlapping area.
- The coverage ratio is computed as the ratio of the intersected area to the total footprint area.
- Clips with coverage ratios outside the defined thresholds (5% to 30%) are excluded, with a debug message printed to indicate the reason for exclusion.
- Valid clips are stored in a dictionary, mapping each building ID to a list of tuples containing the coverage ratio and file path.

**4. Duplicate Detection and Reporting** The script identifies buildings with multiple valid clips (duplicates) and generates a report. For each building with more than one clip, it lists the filenames and their respective coverage ratios. This transparency aids in understanding the data and diagnosing potential issues, such as multiple segmentations for the same building.

**5. Selecting and Copying Best Clips** For each building, the clip with the highest coverage ratio is selected from the list of valid clips. The selected file is copied to the output directory, preserving the original filename. The script ensures the output directory structure is created and logs the selected file and its coverage ratio for each building.

This is an example when the coverage is 103% which is completely an error. It shows why I had to do a coverage thresholding :

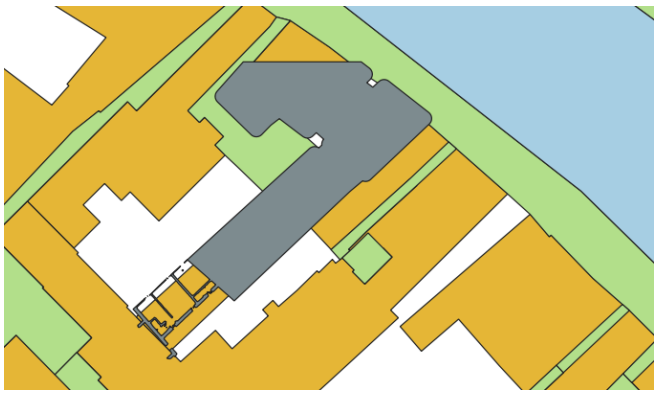


Figure 40: Coverage : 103%

This is the result of the code (partial because the print result is very long) I wrote for the selection :

```
ED1290 has 3 valid clips:
  • hybrid_img0002_modified_ED1290.geojson → 16.7%
  • hybrid_img0004_modified_ED1290.geojson → 17.8%
  • hybrid_img0005_modified_ED1290.geojson → 15.5%
ED5 has 4 valid clips:
  • hybrid_img0003_modified_ED5.geojson → 17.6%
  • hybrid_img0006_modified_ED5.geojson → 19.0%
  • hybrid_img0007_modified_ED5.geojson → 8.0%
  • hybrid_img0036_modified_ED5.geojson → 8.9%
ED42 has 4 valid clips:
  • hybrid_img0003_modified_ED42.geojson → 20.9%
  • hybrid_img0007_modified_ED42.geojson → 9.9%
  • hybrid_img0011_modified_ED42.geojson → 23.1%
  • hybrid_img0034_modified_ED42.geojson → 5.7%
ED368 has 3 valid clips:
  • hybrid_img0003_modified_ED368.geojson → 16.3%
  • hybrid_img0034_modified_ED368.geojson → 21.0%
  • hybrid_img0036_modified_ED368.geojson → 16.9%
ED377 has 3 valid clips:
  • hybrid_img0003_modified_ED377.geojson → 7.5%
  • hybrid_img0034_modified_ED377.geojson → 16.1%
  • hybrid_img0036_modified_ED377.geojson → 12.6%
```

Figure 41: Result of the repetition detection code

#### 2.14.4 Enhancing GeoJSON Clips with Maximum Distance Annotation

In the process of vectorizing segmented building images into GeoJSON format, a recurring issue was the incomplete coverage of building features. Some vectorized clips, derived from hybrid segmentation results, failed to fully align with the corresponding building footprints, resulting in missing segments or features that extended beyond the footprint boundaries. Initial attempts to address this by applying a buffer (e.g., 1 meter) around the geometries risked including unwanted elements, such as parts of neighboring buildings, which introduced noise and inaccuracies. The challenge was to develop a method that could quantify how far clip geometries deviated from the building footprint and use this information to apply an appropriate buffer, ensuring complete coverage without incorporating extraneous features.

I have recognized that a buffer-based approach could compensate for misaligned or protruding features, but a fixed buffer size was problematic. A small buffer (e.g., 1 meter) might not cover all protruding segments, while a larger buffer (e.g., 3 meters) increased the risk of including neighboring structures. A key insight was to measure the distance between each clip geometry and the building footprint to inform the buffer size dynamically. However, using the standard .distance function from geospatial libraries posed a challenge : it calculates the minimum distance from a geometry to the footprint boundary. For geometries partially inside the footprint, the minimum distance could be zero, failing to account for protruding parts that required a larger buffer.

To address this, I computed the maximum distance from any vertex of a clip geometry to the footprint boundary. This metric would capture the extent to which a geometry extended beyond the footprint, providing a precise basis for determining the necessary buffer size. Additionally, I annotated each GeoJSON clip with this maximum distance, with metadata such as source image name, classes/labels.

Figure 42: Geojson's feature that contain EIDIFICI_ID, distance, source name

| Feature | Value |
|---|---|
| **hybrid_img0002_modified_ED43_annotated_maxdist** | |
| image_name | hybrid_img0002_modified |
| (Derived) | |
| (Actions) | |
| class | 1 |
| label | wall |
| EDIFI_ID | 43 |
| image_n... | hybrid_img0002_modified |
| distance | 0.9999999998140444 |

Figure 43: Zoom-in the feature table part

Key Features of the corresponding Implementation :

- Input Processing : The script loads a master GeoJSON file containing building footprints, indexed by a unique building identifier (EDIFI_ID), and iterates over vectorized GeoJSON clips in an input directory.
- Filename Parsing : It extracts the building ID and source image name from each clip's filename, handling potential errors by skipping files with unrecognized formats.
- Coordinate Reference System (CRS) Alignment : Clips are aligned to the same CRS as the footprint data to ensure accurate geometric operations.
- Maximum Distance Calculation : A custom function computes the maximum distance from any vertex of a clip geometry (Polygon, MultiPolygon, or other types) to the footprint boundary. For Polygons and MultiPolygons, it considers all exterior and interior ring coordinates; for other geometries, it uses the bounding box corners as a fallback.
- Annotation : Each clip is enriched with three new columns: EDIFI_ID (building identifier), image_name (source image), and distance (maximum distance to the footprint boundary).
- Output : The annotated GeoJSON files are saved to an output directory with a modified filename indicating the added maximum distance information.

For a technical consideration, the implemented script successfully resolves the challenge of incomplete feature coverage in vectorized GeoJSON clips. By computing the maximum distance from clip geometries to building footprints, it provides a precise metric for determining appropriate buffer sizes, avoiding both under-coverage and the inclusion of neighboring features. The addition of metadata (building ID and image name) enhances traceability, making the output suitable for integration into larger geospatial workflows. The script's robust error handling, support for diverse geometry types, and clear output structure make it a reliable tool for processing all

clips in the dataset. This solution not only addresses the immediate problem but also lays the groundwork for future enhancements, such as dynamic buffer application or advanced geometric filtering, ensuring high-quality vector data for urban planning and architectural analysis.

### 2.14.5 Simplification of Vectorized Geometries

One of the final steps in the pipeline was to simplify the vectorized geometries in order to reduce the complexity of the shapes while preserving the essential architectural structure. As shown in Figure 44, many polygons extracted from segmentation results contain a large number of vertices due to pixel-level irregularities and jagged outlines. While this level of detail is a natural result of the pixel-based segmentation process, it is excessive for applications such as visualization, storage, or topological analysis.

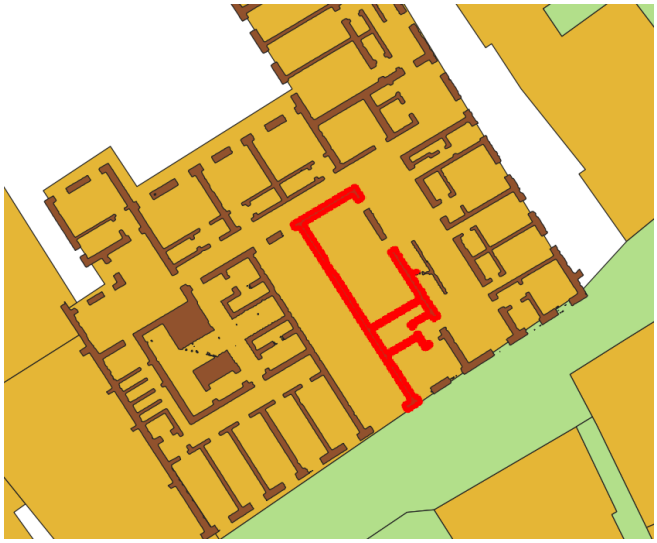


Figure 44: Example of complex wall geometry before simplification (285 vertices

| hybrid_img0002_modified_ED43_annotated_maxdist | |
|---|---|
| image_name | hybrid_img0002_modified |
| (Derived) | |
| (clicke... | 290694.84 |
| (clicke... | 5035585.68 |
| Area (... | 16.54 m² |
| Area (... | 16.53 m² |
| Closes... | 290694.87 |
| Closes... | 5035585.70 |
| Closes... | 290694.87 |
| Closes... | 5035585.70 |
| Closes... | 450 |
| Featur... | 28 |
| Part n... | 1 |
| Parts | 1 |
| Perime... | 90.61 m |
| Perime... | 90.59 m |
| Vertices | 3025 |
| (Actions) | |
| class | 1 |
| label | wall |
| EDIFI_ID | 43 |
| image_n... | hybrid_img0002_modified |
| distance | 0.031067032903451995 |

Figure 45: Table to show vertices number

To address this, I implemented a simplification algorithm using the `.simplify()` method from the `shapely` library, applied dynamically to each geometry. Rather than using a fixed simplification tolerance, I computed

a custom tolerance for each geometry as a fraction of its perimeter (set to 0.5% in this project). This adaptive approach ensures that larger shapes are simplified more aggressively, while smaller shapes are preserved more precisely.

Specifically, the simplification function calculates the perimeter of each geometry (including multipolygons), multiplies it by a fixed factor (0.005), and applies this as the simplification tolerance. The simplification is topology-preserving to avoid invalid geometries, such as self-intersections or broken polygons.

Figure 46 illustrates the after simplification's output for the same building footprint. The reduction in visual complexity is clear and yields cleaner and more interpretable results.

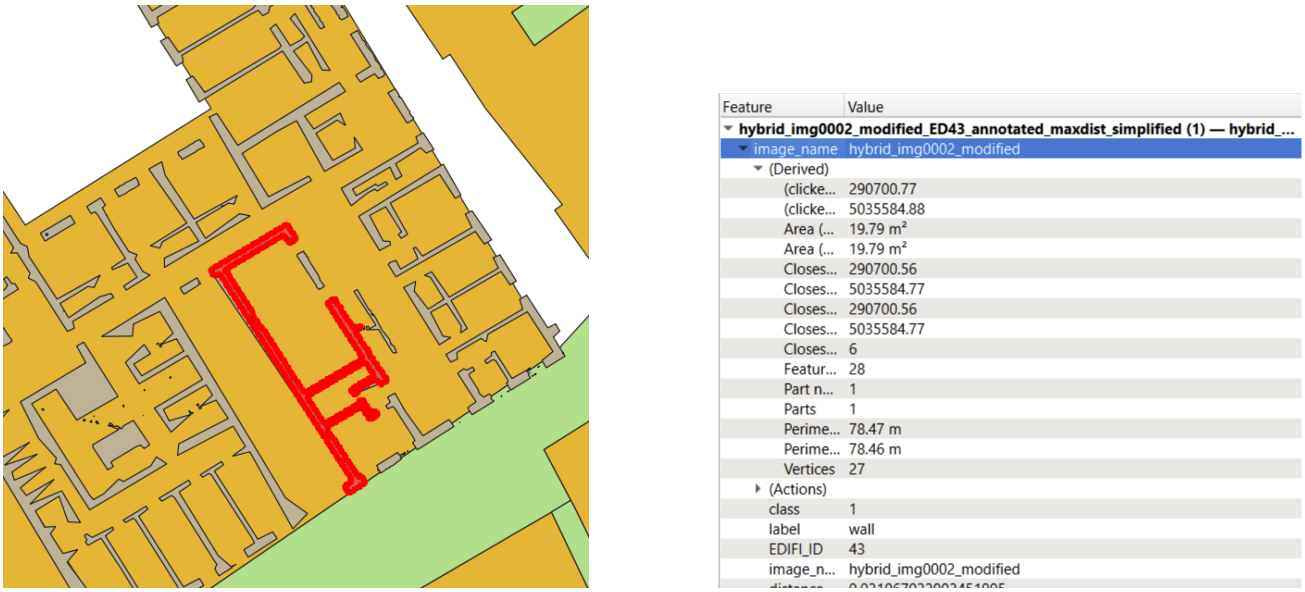



Figure 46: After simplification of building geometry

Beyond visual clarity, simplification significantly reduces file size and computational overhead. For instance, in the example of building EDIFI_ID 43, the original vectorized clip contained 3,025 vertices and had a file size of 1.7MB (see Figure 47). After simplification, the number of vertices was reduced to just 27 and the file size dropped to 95KB, as shown in Figure 48.


Figure 47: Attribute table before simplification : 3,025 vertices, 1.7MB


Figure 48: Attribute table after simplification : 27 vertices, 95KB

This final simplification step was crucial for cleaning up the geometries and making the data more suitable for further geospatial analysis, efficient storage, and presentation.

# 3 Results

## 3.1 Preprocessing and Wall Extraction final result

Figure 49: Thresholded image of img0046.jpg

Figure 50: Wall extracted image of img0046.jpg

Figure 44 and Figure 45 display the thresholding and wall extraction results for img0046.jpg, respectively. In Figure 44, the thresholding technique successfully removes some unnecessary text and noise from the original image, as seen in the cleaner layout. Figure 45 further shows the wall extraction result, where most text elements

are eliminated, and the primary structures (walls) are more distinctly visible. This indicates that the improved method effectively removes text while preserving essential wall structures, resulting in a clearer output. The thresholding step played a crucial role in balancing noise removal and structural preservation.

## 3.2 GT patch preparation final result

GT patch results :


Figure 51: Original sample in patch of size 512x512

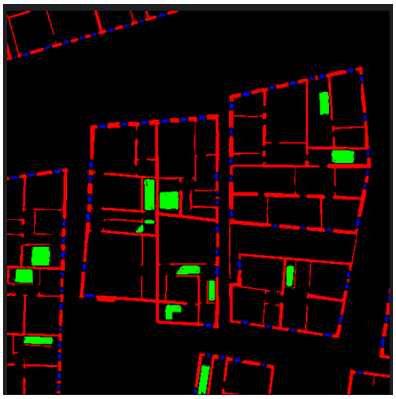


Figure 52: GT patch of size 512x512

Figure 53: Original patch of size 256x256

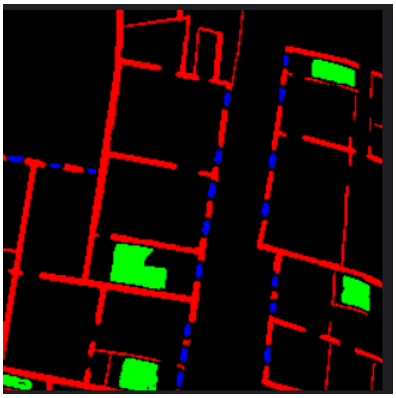
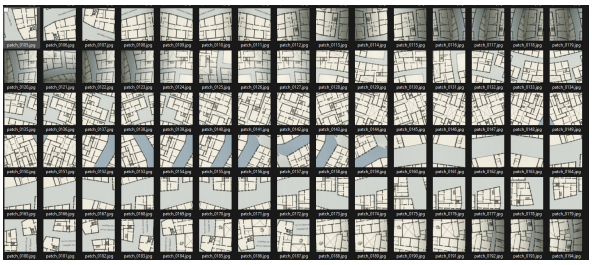
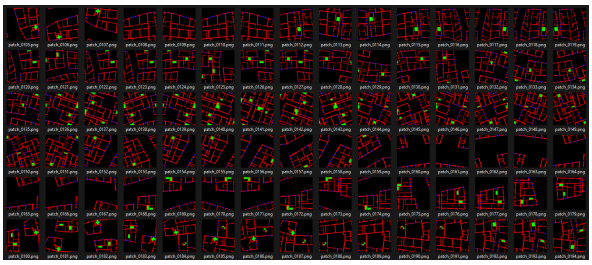


Figure 54: GT patch of size 256x256

This is what our dataset for training model looks like (after applying data augmentation) :



(a) augmented input images



(b) augmented label images

## 3.3 Training final result

| Epoch | Training Loss | Validation Loss | Overall Accuracy |
|---|---|---|---|
| 1 | 0.875300 | 0.385094 | 0.924184 |
| 2 | 0.259800 | 0.184247 | 0.930676 |
| 3 | 0.170100 | 0.151550 | 0.932416 |
| 4 | 0.147800 | 0.140143 | 0.934197 |
| 5 | 0.134400 | 0.131396 | 0.935270 |
| 6 | 0.125300 | 0.121988 | 0.935786 |
| 7 | 0.115900 | 0.115513 | 0.938341 |
| 8 | 0.110800 | 0.106246 | 0.943080 |
| 9 | 0.105600 | 0.107110 | 0.942240 |
| 10 | 0.100400 | 0.099167 | 0.943610 |
| 11 | 0.096700 | 0.098461 | 0.945070 |
| 12 | 0.093600 | 0.099276 | 0.944023 |
| 13 | 0.093800 | 0.097733 | 0.943603 |
| 14 | 0.091600 | 0.096919 | 0.945161 |
| 15 | 0.090500 | 0.095855 | 0.944664 |

✅ Training complete and model saved to Google Drive: /content/drive/MyDrive/segformer-models/segformer-final

Figure 56: Training result with eopchs=15

**Result with epochs=15** In the final phase, I ran the training code with epochs=50 and it gave around 90%. Unfortunately, I have lost the evidence, however, the 90% is guaranteed. This indicates stable convergence of the model.

Also, before implementing the logit thresholding method, I have modified the weight in training code :

```
weights = torch.tensor([0.2, 1.0, 60.0, 50.0]).to(logits.device)
```

## 3.4 Inference final result

### 3.4.1 When I used the median as logit threshold

The result was :

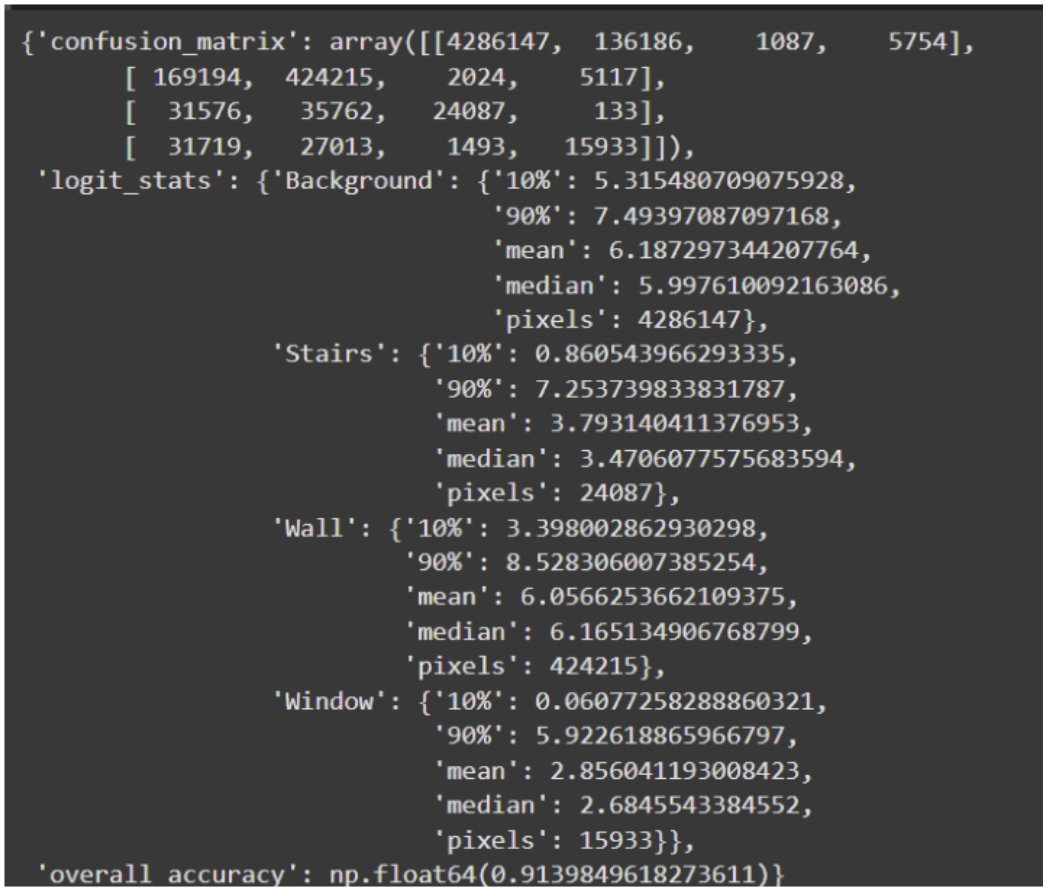



Figure 57: Logit stats

A remark is that the majority of pixels are the background class' pixels and they are dominating. This is why I needed to modify the weights in training phase to prevent class imbalance.

The corresponding metrics result is :

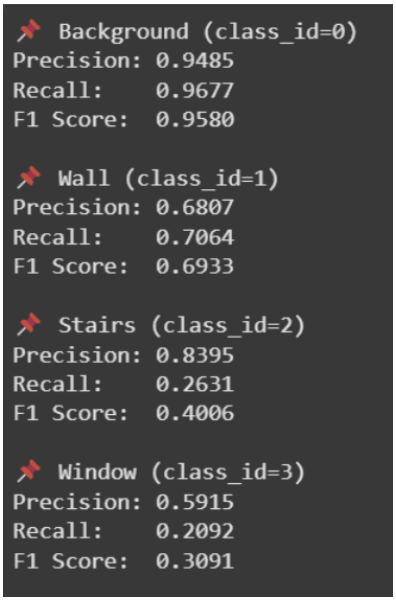
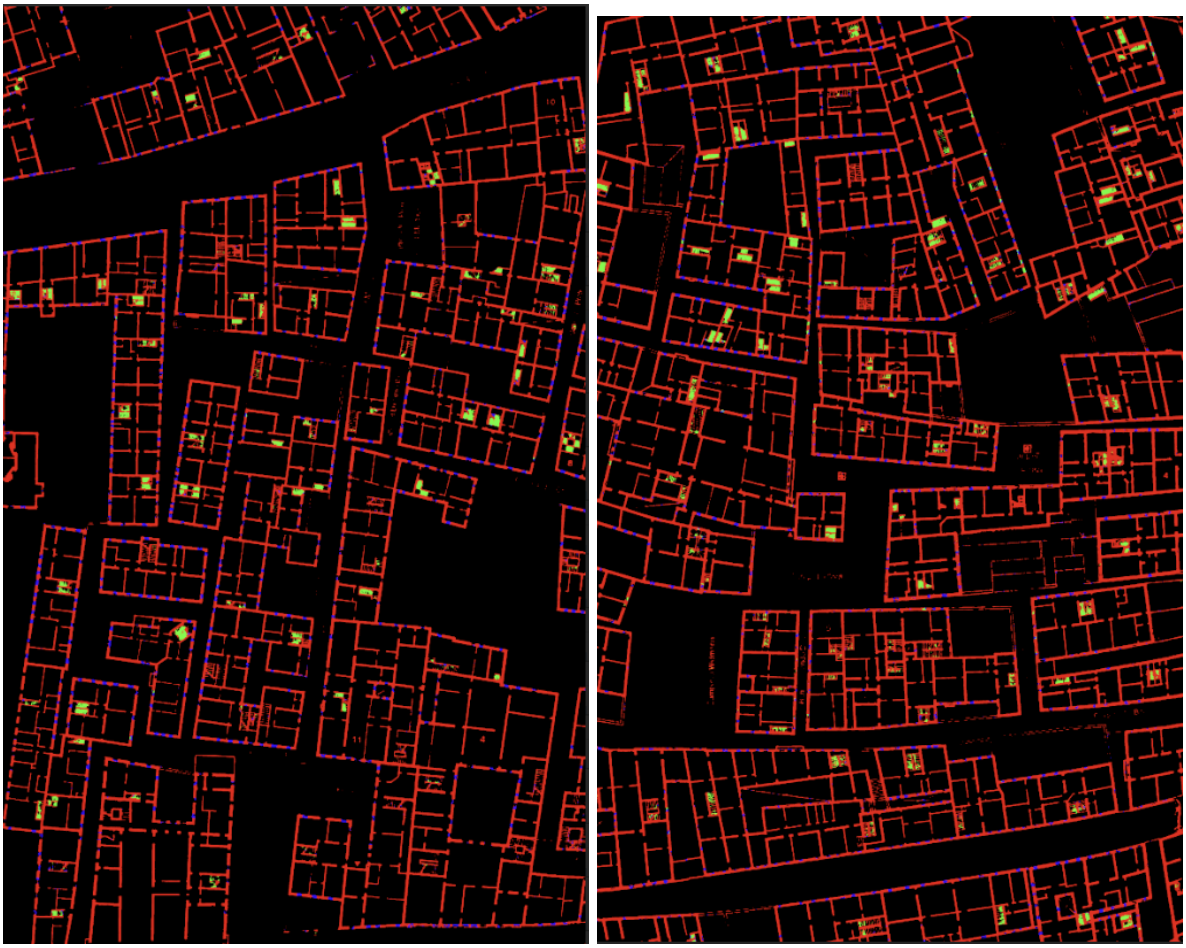



Figure 58: Metrics

- Background (class_id=0) : Achieves a high Precision of 0.9485, Recall of 0.9677, and F1 Score of 0.9580. The background class dominates the dataset and has simpler patterns, making it easier for the model to learn and classify accurately.
- Wall (class_id=1): Records a Precision of 0.6807, Recall of 0.7064, and F1 Score of 0.6933, showing decent performance but lower than the background. Walls have more complex structures and are sometimes confused with noise, which likely contributes to the reduced performance.
- Stairs (class_id=2): Shows a Precision of 0.8395 but a low Recall of 0.2631, resulting in an F1 Score of 0.4006. Stairs occupy a smaller portion of the images and have inconsistent patterns, making them challenging for the model to detect consistently.
- Window (class_id=3): Exhibits the lowest performance with a Precision of 0.5915, Recall of 0.2092, and F1 Score of 0.3091. Windows are small, often confused with walls or background, and some may have been lost during data augmentation, impacting the model's ability to learn their features.

The performance differences can be attributed to class imbalance and the inherent complexity of each class. Windows and stairs, being minority classes, likely suffer from insufficient training data. Additionally, the logits distribution in Figure 52 indicates overlapping values between classes, suggesting that the model struggles to establish clear classification boundaries for minority classes like windows and stairs. For this, I created two more GT annotation and made our training dataset larger with more training samples for windows and stairs, but it did not work well. The detail of this step will be reminded in "Discussion" section.

Here are some of results of the inference code :



(a) outcome of image0038.png

(b) outcome of image0102.png

We can state that the output of Deep Learning model is satisfying. There are still some outputs that are not satisfying because they usually show weaker performance for recognizing windows and stairs and this is going to be discussed in the "Discussion" section.

## 3.5 Hybrid method final result

(a) outcome of hybrid method applied image0038.png (b) outcome of hybrid method applied image0102.png

Figure 55 demonstrates the results of the hybrid method applied to img0046.jpg and img0102.jpg. The hybrid approach, combining preprocessing with deep learning, enhances wall detection significantly. However, windows and stairs continue to show poor detection rates compared to walls. The misclassifications of windows as walls further indicates that the model struggles to capture the distinct features of windows, pointing to limitations in the feature learning process for minority classes. Fortunately, for some samples like those two samples, this error does not appear frequently.

## 3.6 Georeference

This is an example of a georeferenced but NOT vectorized image :

Figure 61: georeferenced hybrid_img0035

This is an example of a georeferenced AND vectorized image :

Figure 62: georeferenced vectorized hybrid_img0035

Figures 56 and 57 show a non-vectorized and vectorized georeferenced image, respectively. The georeferencing process preserves the main structural layout well, but the vectorization in Figure 57 results in some loss of fine details (or not showing a perfect "fit" since the georeferencing cannot always be perfect). This loss is likely due to the excessive high precision demands during the georefer phase. This is why I had to add the "maximum distance measuring" method in order to choose a wise buffer value for analysis.

# 4 Discussion

This study aimed to develop a pipeline for processing scanned architectural floor plans, including wall extracting, segmentation, and georeferencing with vectorization, to enable urban planning and architectural analysis. While the pipeline achieved moderate success in certain areas, such as wall detection and georeferencing, several challenges emerged that highlight deeper issues in the methodology, data preparation, and overall approach. This section reflects on these challenges, explores the underlying reasons for successes and failures, evaluates the potential adaptability of the method, and identifies limitations and future directions.

## 4.1 Challenges in Dewarping : Impact of Data Collection

One of the primary challenges encountered was dewarping the scanned images, which exhibited significant folding artifacts due to the physical constraints of the scanning process. The initial methodology involved splitting the image into three parts, dewarping the folded region, and recombining the segments. However, this approach did not produce satisfactory results, as shown in Figure 26. Subsequent attempts using deep learning models like DewarpNet [**DewarpNet**] and DocRes [**DocRes**] also yielded suboptimal outcomes (Figures 27 and 28). A deeper analysis reveals that the root cause lies in the data collection process. The Venetian architectural book was scanned without separating its pages, resulting in thick, curved pages that introduced folding distortions (Figure 1). This curvature not only affected the dewarping process but also contaminated the training dataset, as the distorted regions were inadvertently learned as part of the architectural features during data augmentation and training.

The failure of dewarping models in this context underscores a critical limitation : existing models are typically designed for text-based documents, where techniques like OCR can leverage textual features to correct distortions. Architectural floor plans, however, lack such textual cues, making dewarping significantly more challenging. The absence of specialized models for non-textual scanned images highlights a gap in the literature and suggests that dewarping architectural plans may require a tailored approach, possibly combining geometric constraints specific to floor plans with deep learning techniques. Moreover, the physical constraints of the scanning process imposed by the inability to cut the borrowed book illustrate how practical limitations can profoundly impact the quality of the dataset and, consequently, the entire pipeline.

## 4.2 Class Imbalance and Training Limitations

The training phase revealed significant performance disparities across classes, as discussed in Section 3.3. While the model achieved a stable accuracy of approximately 90% at epochs=50, the metrics (Figure 58) indicate that the background class dominated with an F1 Score of 0.9580, while minority classes like windows (F1 Score: 0.3091) and stairs (F1 Score: 0.4006) performed poorly. To address class imbalance, weights were adjusted in the training code (`weights = torch.tensor([0.2, 1.0, 60.0, 50.0])`), assigning higher weights to windows and stairs. Additionally, two more ground truth (GT) annotations were created to increase the training samples for these minority classes. However, this approach did not yield the expected improvement and, in some cases, exacerbated the issue.

Reflecting on this outcome, the increase in GT annotations inadvertently amplified the background class as well, since background pixels (black regions) inherently dominate architectural floor plans. Even when focusing on windows and stairs, the background being a "space" rather than a "line" like windows or stairs remains overwhelmingly present. This suggests that simply increasing the dataset size without addressing the structural imbalance between classes is insufficient. An alternative strategy, such as isolating windows and stairs for separate training, was considered but deemed impractical. Isolating these features still includes background pixels, and manually cropping them requires significant labor, undermining the purpose of using deep learning

for automation. This laborious process also mirrors the broader challenge of the project : the reliance on manual effort at multiple stages, which diminishes the scalability of the approach.

The poor performance on windows and stairs can also be attributed to their inherent complexity. Windows are small and are often confused with walls or background, whereas stairs have inconsistent patterns and occupy a small portion of the image. These characteristics suggest that the model architecture and training strategy may not be well-suited for detecting fine-grained features in architectural plans. A deeper limitation lies in the nature of the dataset itself : the folding distortions from the scanning process likely introduced noise that further complicated the learning of minority class features.

## 4.3 Georeferencing : Efficiency and Scalability Issues

The georeferencing process, while successful for a subset of images, revealed significant scalability challenges. Using QGIS to manually select 4–6 ground control points (GCPs) for each image, as shown in Figures 36 and 37, was highly time-consuming. Each image required approximately 20-30 minutes to georeference, as finding an optimal fit often involved iterative adjustments due to delays, rotations, or slight deformations in the scanned images. For a dataset of 132 images, this process became a bottleneck, with only 50 images successfully georeferenced by the end of the project.

This inefficiency highlights a fundamental limitation of the current pipeline : the reliance on manual labor for georeferencing. Combined with the manual scanning and GT annotation processes, the cumulative labor burden significantly hindered the project's progress. However, a positive outcome is the establishment of a reproducible pipeline. The scripts developed for georeferencing, vectorization, and clip selection (Section 3.6) ensure that, given the manual georeferencing step, the remaining processes can be automated to produce vectorized and georeferenced GeoJSON files (Figures 39 and 42). This pipeline enables downstream applications like urban planning and architectural analysis, provided that the challenge of manual labor is addressed.

## 4.4 Adaptability and Applications

The developed pipeline, despite its challenges, has potential applications in historical architectural analysis, particularly to digitize and geo-reference old floor plans. The ability to vectorize and georeference building footprints can support urban planning in historical cities like Venice, where understanding the spatial layout of buildings is crucial for preservation and development. However, the method's adaptability to other domains, such as natural images or modern architectural plans, is limited. The reliance on specific preprocessing steps (e.g., thresholding for wall extraction) and the challenges with dewarping suggest that the pipeline is tailored to the unique characteristics of the Venetian floor plan dataset. Applying this method to datasets with different structural properties or scanning conditions would likely require significant modifications.

## 4.5 Limitations and Future Work

The primary limitation of this study lies in the labor-intensive nature of the pipeline, which spans from scanning and ground truth annotation to georeferencing. This reliance on manual effort not only limits scalability but also introduces inconsistencies due to human error, particularly in annotation accuracy and spatial alignment. Furthermore, the dataset itself affected by folding distortions posed fundamental challenges, especially for dewarping and the detection of minority classes such as windows and stairs. The absence of specialized dewarping models tailored for non-textual historical images further compounded these issues.

Future work should prioritize the automation of time-consuming stages to improve the pipeline's scalability and consistency. For dewarping, designing a custom deep learning model that incorporates geometric priors from architectural drawings could address current shortcomings. Alternatively, addressing the issue at the data

collection stage through the use of professional book scanners for bound volumes or by digitizing books as flat individual pages could eliminate fold-related distortions altogether.

To mitigate class imbalance in training, methods such as focal loss or targeted data augmentation for rare classes could improve the model's sensitivity, particularly for features like windows and stairs. In georeferencing, automating ground control point (GCP) selection through feature-matching algorithms between raster outputs and vector footprints could significantly reduce manual input and make the pipeline suitable for large-scale deployment.

Moreover, the rich attribute structure embedded in the GeoJSON outputs such as class labels, spatial distances, and image references offers significant potential for downstream analysis in the context of Digital Humanities. By leveraging this structured vector data, future studies could conduct typological or socio-spatial analyses at the level of individual buildings. For example, one could quantify the number of windows or analyze the presence and configuration of staircases to infer architectural functions, spatial hierarchy, or even patterns of daily life. A dense clustering of stairs might suggest multi-level or communal dwellings, while an absence of windows in certain buildings might point to storage areas or defensive structures. These insights could be further enriched by combining vector outputs with historical maps or metadata, opening new avenues for interpreting lived experiences within Venetian urban fabric.

# 5 Data Architecture

All the data and results are saved in `https://drive.google.com/drive/folders/1IcfwlD5aW1YeNYRYpWzDi-JFGjIv37vV?usp=drive_link`.

In "segmentation_project/" folder, there are :

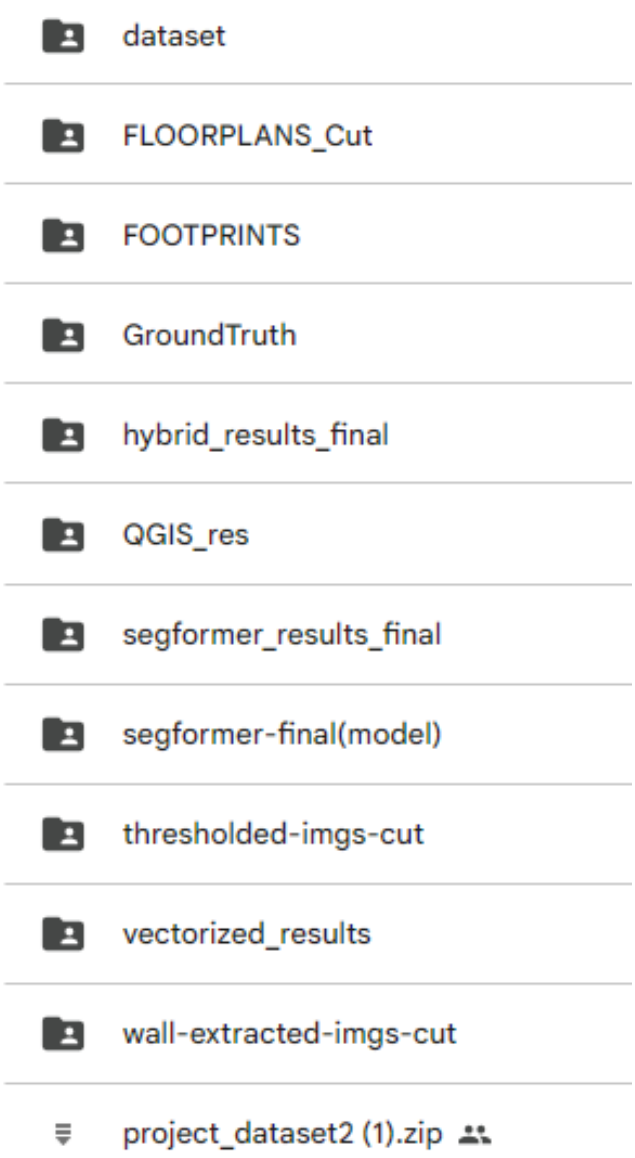



Figure 63: Main folders

- In the "dataset/" folder, there is "project_dataset2/" folder, which contains "aug_inputs/" and "aug_labels/" folders. Those two folders are composed by augmented patch-divided original images and augmented patch-divided GT images.
- In the "FLOORPLANS_Cut/" folder, there are the 132 floorplans images that I scanned by myself.
- In the "FOOTPRINTS/" folder, there are the three geojson files that my supervisor have provided : "2024_Canals.geojson", "2024_Streets.geojson", "2024_Edifici.geojson".
- In the "GroundTruth/" folder, there are the three groundtruth images that I annotated myself and there is only the "gt038_cleaned.png" that is used for this project.
- In the "hybrid_results_final/" folder, there are the 132 images that have been applied the hybrid method (OpenCV based Segformer).
- In the "QGIS_res/" folder, there are :

  - "vectorized_hybrid_qgis/" folder which contains all the GCP points and .tif files of the hybrid results final's images.
  - "vec_res.qgz" which is the qgis file that contains the whole QGIS project.

- In the "segformer_results_final/" folder, there are the 132 images that are the output of the segformer inference code.
- In the "segformer-final(model)/" folder, there are the weights of the training code.
- In the "thresholded-imgs-cut/" folder, there are the thresholded images of "FLOORPLANS_Cut" images.
- In the "vectorized_results/" folder, there are :

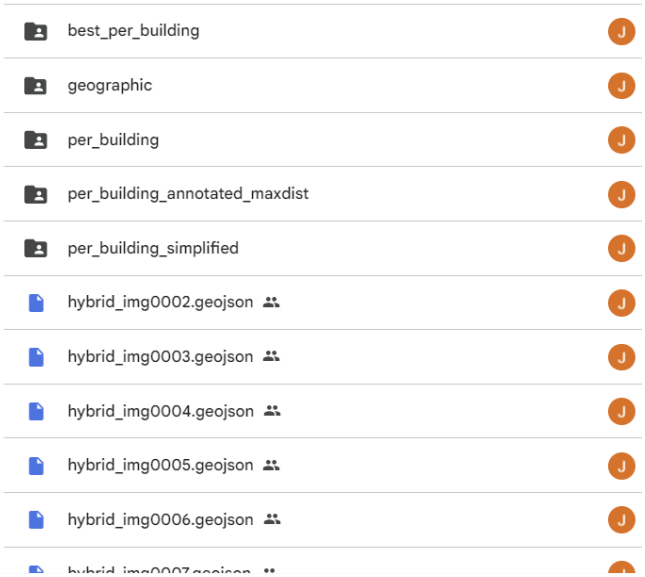



Figure 64: vectorized_results/

  - "best_per_building/" folder contains all the buildings (edifici) that have the best coverage
  - "geographic/" folder contains all the georefered vectorized hybrid images.
  - "per_building/" folder contains all the clips of buildings .geojson
  - "per_building_annotated_maxdist/" folder contains all the georefered vectorized hybrid images with distance feature in it.
  - "per_building_simplified/" folder contains all the simplified version of vectorized geometries

- In the "wall-extracted-imgs-cut/" folder, there are all the wall extracted images from thresholded images.

# 6 Conclusion

This project aimed to explore typological patterns within Venetian architectural floor plans through a hybrid computer vision pipeline combining classical image processing and deep learning techniques. Beginning with the digitization of over 280 historical maps, the workflow addressed numerous challenges inherent in the scanned materials, including text overlays, visual noise from rivers and bridges, and distortions near page folds. Using OpenCV, tailored preprocessing methods were developed for effective wall extraction and partial dewarping, while custom-labeled semantic ground truth was created through meticulous annotation.

To tackle the semantic segmentation task, we trained a lightweight SegFormer-B4 model on patch-based datasets. The preprocessing pipeline included class-specific color correction using KDTree, and the model's robustness was enhanced through data augmentation with Albumentations. Our inference strategy included patch-wise prediction with overlap and subsequent stitching, ensuring coherent reconstruction of large-scale segmentation masks.

In the post-processing stage, we integrated polygonization methods to vectorize thick wall structures and fused SegFormer outputs to preserve meaningful details like windows and stairs. This hybrid approach leveraged the geometric clarity of classical methods and the contextual understanding of transformers, providing both interpretability and flexibility.

Overall, the project successfully demonstrated that combining traditional and deep learning-based techniques can produce accurate and semantically rich representations of complex architectural plans. These results lay a solid foundation for future quantitative typological analyses, such as clustering or classification of plan configurations, and open the door to broader applications in architectural heritage digitization and computational urban studies.

# References


[1] Na Li and Steven Jige Quan. "Discovering urban block typologies in Seoul: Combining planning knowledge and unsupervised machine learning". In: *Cities* 150 (2024), p. 104988. DOI: 10.1016/j.cities.2024.104988. URL: https://doi.org/10.1016/j.cities.2024.104988.

[2] Jitendra Pandey, Gandharv Mahajan, and Sahil Tadwalkar. "2D Floor Plan Reconstruction Using Cool Deep Learning Methods". In: *Stanford University Research* (2024). URL: https://cv4aec.github.io/.

[3] Alexander Hakert and Phillip Schönfelder. "Informed Machine Learning Methods for Instance Segmentation of Architectural Floor Plans". In: *Forum Bauinformatik* (2022), pp. 395–403. DOI: 10.1007/978-3-319-92862-3.

[4] JongHyeon Yang et al. "Semantic Segmentation in Architectural Floor Plans for Detecting Walls and Doors". In: *2018 11th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI)*. IEEE, 2018. DOI: 10.1109/CISP-BMEI.2018.8633084. URL: https://doi.org/10.1109/CISP-BMEI.2018.8633084.

[5] Coder Singh. *Preprocess the images using Python OpenCV*. Accessed: 2025-03-07. 2024. URL: https://codersingh27.medium.com/preprocess-the-images-using-python-opencv-eacf4bf34477.

[6] GeeksforGeeks. *Line Detection in Python using Hough Line Transform*. Accessed: March 6, 2025. 2023. URL: https://www.geeksforgeeks.org/line-detection-python-opencv-houghline-method/.

[7] GeeksforGeeks. *Image Segmentation using Morphological Operations*. Accessed: March 6, 2025. 2023. URL: https://www.geeksforgeeks.org/image-segmentation-using-morphological-operation/.

[8] GeeksforGeeks. *Python OpenCV - Morphological Operations*. Accessed: March 6, 2025. 2023. URL: https://www.geeksforgeeks.org/python-opencv-morphological-operations/.

[9] GeeksforGeeks. *Find and Draw Contours using OpenCV Python*. Accessed: March 6, 2025. 2023. URL: https://www.geeksforgeeks.org/find-and-draw-contours-using-opencv-python/.

[10] GeeksforGeeks. *OpenCV Python Tutorial*. Accessed: March 6, 2025. 2023. URL: https://www.geeksforgeeks.org/opencv-python-tutorial/.

[11] StackOverflow. *Proximity of Colors in a Rainbow*. Accessed: 2025-06-01. 2025. URL: https://stackoverflow.com/questions/11247372/proximity-of-colors-in-a-rainbow.

[12] StackOverflow. *Replace Colors in Image by Closest Color in Palette Using NumPy*. Accessed: 2025-06-01. 2025. URL: https://stackoverflow.com/questions/70823769/replace-colors-in-image-by-closest-color-in-palette-using-numpy.

[13] Seth Sara. *Change Pixel Colors of an Image to Nearest Solid Color with Python and OpenCV*. Accessed: 2025-06-01. 2025. URL: https://sethsara.medium.com/change-pixel-colors-of-an-image-to-nearest-solid-color-with-python-and-opencv-33f7d6e6e20d.

[14] StackOverflow. *Map Colors in Image to Closest Member of a List of Colors in Python*. Accessed: 2025-06-01. 2025. URL: https://stackoverflow.com/questions/57496971/map-colors-in-image-to-closest-member-of-a-list-of-colors-in-python.

[15] Albumentations. *Image Augmentation with Albumentations*. Accessed: 2025-06-01. 2025. URL: https://albumentations.ai/docs/getting_started/image_augmentation/.

[16] Nimrita Koul. *Image Augmentations with Albumentations Python Library (Part1)*. Accessed: 2025-06-01. 2025. URL: https://medium.com/@nimritakoul01/image-augmentations-with-albumentations-python-library-part1-823f99a3943a.

[17] Geek Culture. *Semantic Segmentation with SegFormer*. Accessed: 2025-06-01. 2025. URL: https://medium.com/geekculture/semantic-segmentation-with-segformer-2501543d2be4.

[18] Hugging Face. *Trainer - Hugging Face Transformers*. Accessed: 2025-06-01. 2025. URL: https://huggingface.co/docs/transformers/main_classes/trainer.

[19] Niels Rogge. *Fine-tune SegFormer on Custom Dataset (RUGD)*. Accessed: 2025-06-01. 2025. URL: https://github.com/NielsRogge/Transformers-Tutorials/blob/master/SegFormer/Fine_tune_SegFormer_on_custom_dataset_%5BRUGD%5D.ipynb.

[20] YouTube. *SegFormer Semantic Segmentation Tutorial*. Accessed: 2025-06-01. 2025. URL: https://www.youtube.com/watch?v=4HNkBMfw-2o.

[21] Scikit-learn. *sklearn.model$_s$election.train_test_split*. Accessed: 2025-06-01. 2025. URL: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html.

[22] Hugging Face. *nvidia/segformer-b0-finetuned-ade-512-512*. Accessed: 2025-06-01. 2025. URL: https://huggingface.co/nvidia/segformer-b0-finetuned-ade-512-512.

[23] Hugging Face. *Evaluate - Transformers Integrations*. Accessed: 2025-06-01. 2025. URL: https://huggingface.co/docs/evaluate/en/transformers_integrations.

[24] PyImageSearch. *Transparent Overlays with OpenCV*. Accessed: 2025-06-01. 2025. URL: https://pyimagesearch.com/2016/03/07/transparent-overlays-with-opencv/.

[25] GIS Stack Exchange. *Extracting Raster Outline to a Vector Geometry*. Accessed: 2025-06-01. 2025. URL: https://gis.stackexchange.com/questions/429662/extracting-raster-outline-to-a-vector-geometry.

[26] Google Developers. *Reducers.reduceToVectors*. Accessed: 2025-06-01. 2025. URL: https://developers.google.com/earth-engine/guides/reducers_reduce_to_vectors#colab-python.

[27] pygeocompx. *Chapter 5: Raster Vector*. Accessed: 2025-06-01. 2025. URL: https://py.geocompx.org/05-raster-vector.

[28] Shapely. *Shapely Documentation*. Accessed: 2025-06-01. 2025. URL: https://shapely.readthedocs.io/en/stable/manual.html.

[29] Shapely. *Shapely.buffer Reference*. Accessed: 2025-06-01. 2025. URL: https://shapely.readthedocs.io/en/stable/reference/shapely.buffer.html.