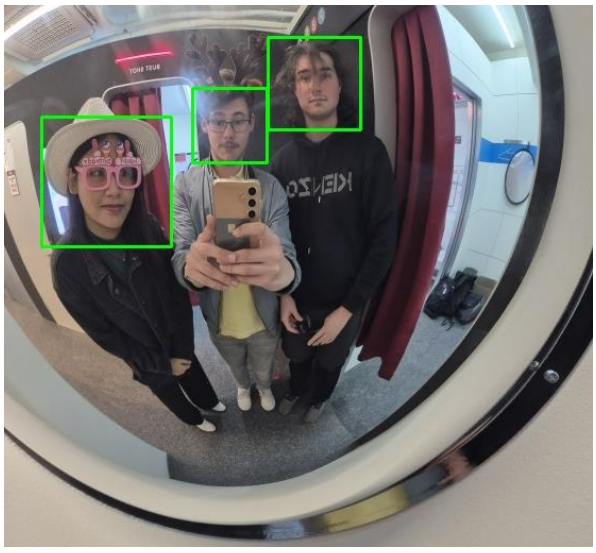
Chaw Thiri San
PORET Guillaume
LE MONIES DE SAGAZAN Cyrius

31/10/2024

Digital Image Processing:

Project #1

Face Detection using Viola Jones



Professor: Kakani Vijay

Digital Image Processing (Midterm Project)

Table des matières

I. Introduction

Face detection is a vital component in modern computer vision applications, spanning from security systems and biometric identification to interactive user interfaces and augmented reality. Our project leverages the Viola-Jones (VJ) algorithm, known for its real-time detection capability, to explore the boundaries of automated face detection in varied conditions. By combining Haar-like features and a cascade classifier, we constructed a robust face detection model aimed at recognizing facial patterns across diverse scenarios. This report presents our methodology, including dataset preparation, training, and model tuning. We explore various challenges, such as occlusion, illumination changes, and viewpoint variation, and examine the limitations of our model by analyzing minimum and maximum detectable face sizes, detection speed, and instances of false positives and false negatives. Through this, we aim to contribute insights into the adaptability of the VJ algorithm for complex, real-world applications.

Team Members

Chaw Thiri San - 12225272
PORET Guillaume - 12244696
LE MONIES DE SAGAZAN Cyrius - 12244594

Figure 1: Photo of one of our team meetings

Dataset of our best model

P image = 915 (but only 777 use, 85% positive image usage parameter)

N image = 3760

Table of Contributions

Model Development	Name
P-image	Chaw Thiri San
N-image	LE MONIES DE SAGAZAN Cyrius
Model Training and parameter tuning	PORET Guillaume
Report Preparation	
Haar Features and Cascade Classifier Training, Dataset and Parameter Problem III GitHub	PORET Guillaume
Viola-Jones (VJ) Algorithm Problem I, II	LE MONIES DE SAGAZAN Cyrius
False Detection Issue and Performance Analysis, Challenges in Object Detection, Is Our Model Perfect, Introduction and Conclusion Video editing	Chaw Thiri San

II. Background and Theoretical Framework

1. Viola-Jones (VJ) Algorithm

The Viola-Jones algorithm was developed in 2001 by computer scientists Paul Viola and Michael Jones, marking a significant breakthrough in real-time face detection. Before their work, most face detection methods were computationally expensive and too slow for practical, real-time applications.

During this project we use the OpenCV GUI software to train our model, this program uses the Viola-Jones algorithm as it is a popular method for object detection, particularly for face detection. Here's a breakdown of what the algorithm is and how it works:

The Viola-Jones algorithm is designed for real-time object detection. It's use for its speed and accuracy and is particularly effective for face detection. The algorithm comprises four key components:

Haar Features: simple rectangular features that capture the presence of gradients in specific areas of an image.

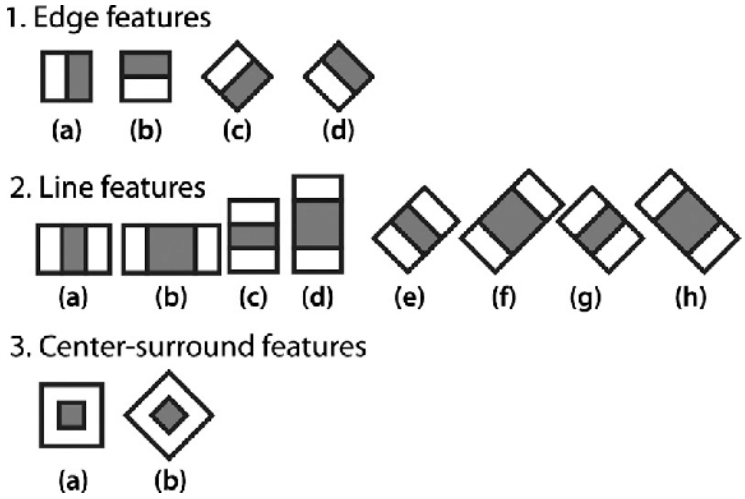


Figure 2: Image of Haar Features

Integral Image: To speed up the computation of Haar features.

AdaBoost: To select and train a series of weak classifiers.

Cascade Classifier: A series of increasingly complex classifiers arranged in a cascade that allows the algorithm to quickly discard non-face regions while performing detailed analysis only on promising regions.

This explain how Viola-Jones algorithm a powerful and efficient method for face detection is and why it is use by OpenCV.

2. Haar Features and Cascade Classifier

The Haar algorithm is a feature-based method primarily used in object detection, and it's particularly effective in face detection as part of the Viola-Jones algorithm. The algorithm operates by identifying patterns within images using Haar-like features, which are rectangular features inspired by Haar wavelets. Each feature is a simple rectangular structure that is positioned over an image, capturing differences in intensity between adjacent areas.

Haar features consist of black and white rectangles arranged to capture contrasting regions in an image. When applied to a face, for example, a Haar feature might cover the region of the eyes (darker) and compare it to the surrounding cheek area (lighter). By summing up the pixel intensities within each rectangle, the Haar algorithm calculates a difference between the light and dark regions, helping to highlight specific characteristics unique to faces, such as:

- The bridge of the nose (bright-dark-bright contrast).
- The eye sockets (dark-light-dark contrast).
- The overall shape of the face (contrast between the face and the background).

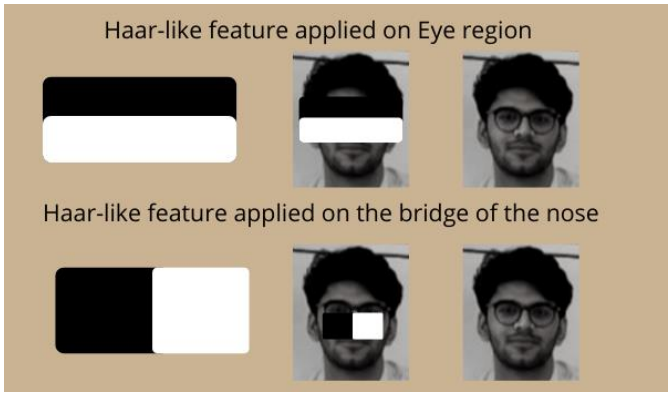


Figure 3: Example of Haar-like Features: Detecting Eye and Nose Regions in Face Detection

The cascade classifier leverages these Haar features to detect faces in a highly efficient way. Rather than processing every pixel of an image, the cascade classifier works in stages, applying Haar features progressively to filter out areas that don't resemble faces.

- Early Stages: Simple features are applied to quickly eliminate large portions of the image that are unlikely to contain a face, such as the background. This is done to improve speed by discarding negative regions as early as possible.
- Later Stages: As the algorithm moves forward, more complex and sensitive features are applied to refine the search. The regions that pass through each stage are increasingly likely to contain a face.

This multi-stage filtering approach makes the cascade classifier both fast and effective. Only the regions with the strongest likelihood of being a face reach the final stages, making the detection process efficient by focusing computation where it is most needed.

The combination of Haar features and the cascade classifier thus enables the algorithm to rapidly scan images for facial structures, making it ideal for real-time applications in face detection.

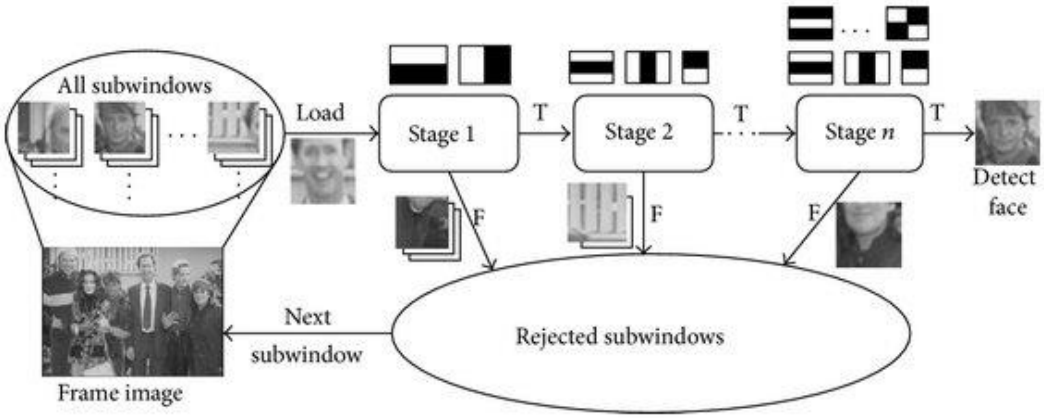


Figure 4: Multistage Approach of the Haar Cascade Model: Sequential Filtering of Subwindows for Efficient Face Detection

III. Training, Dataset, and Parameters

We divided this section into three tasks. The first two tasks focused on building a robust dataset by gathering relevant data. Chaw Thiri San worked on sourcing face images for the positive dataset (P), with the aim of including a diverse set of 5,000 face images, including our respective heads with various accessories, orientation and distance from the phone. Similarly, Cyrius handled the negative dataset (N), which included 15,000 images of objects, backgrounds, and animals to ensure the model could accurately distinguish faces from non-facial elements. In the final task, Guillaume managed the training process and fine-tuning the parameters.

Given the lengthy training times, computational demands, and multiple unsuccessful attempts, we had to make significant adjustments to our initial plan. We optimized the number of images in the final training model and modified parameters within the training application to improve efficiency and accuracy. These adjustments allowed us to balance dataset size, fine-tune application settings, and achieve our desired accuracy while managing processing constraints effectively.

- Regarding the Dataset: We cropped approximately 900 faces from our positive dataset using the training application to reduce the search area within our original images, which were 1024x768 pixels. This adjustment not only decreased the dataset size but also optimized processing by focusing on relevant regions. Additionally, we created a Python script to randomly delete a percentage of images, reducing the negative dataset from 15,000 images to 3,700. Another Python program resized the negative images to 600x450 pixels, adding padding to avoid image distortion. We also included more discriminative negative images, such as hands, animals, and various objects, to improve detection accuracy. (These image manipulation scripts can be found in the "image_database_management" directory of the repository.)

Regarding the Cascade Trainer GUI: We adjusted the positive image usage parameter to 85% due to a common issue in the application. During training, certain positive samples are sometimes removed, causing a crash when the following stage lacks sufficient positive images. This error message, “OpenCV Error: Bad argument (Can not get new positive sample… insufficient count of samples in given vec-file…)” led us to search OpenCV community forums, where we found that lowering this parameter is the best workaround.

To further improve model accuracy and reduce false detections, we increased the detection window size to 40x40 pixels. However, this adjustment also extended the model’s training time significantly. In the end, training took well over a week, with some attempts requiring more than a full day of continuous training.

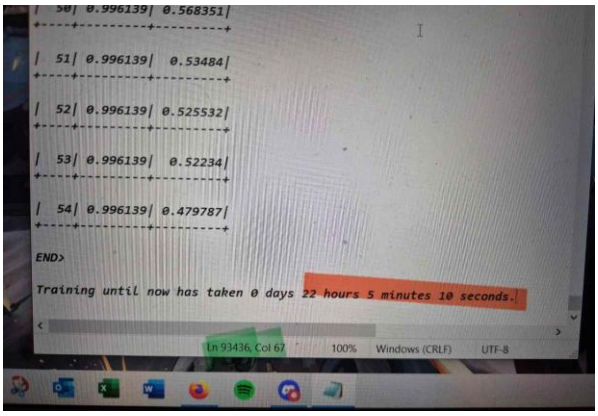


Figure 5: Training Log of Our Best Model

IV. Results and Observations

I. **Problem 1**: Minimum detectable face size.

The goal of our minimum face detection is to find the limit of our model. It works by analysing each frame of the video, detect faces, and track the smallest face detected across all frames. We set up the video and face detector using the loading function VideoCapture() of OpenCV. Then, the face detection model (cascade.xml) is loaded using the CascadeClassifier() function of OpenCV.

We initialize tracking variables to keeps track of the area of the smallest face found so far, to stores the frame number where this smallest face was detected, to holds a copy of the frame with the smallest face, and to stores the coordinates (x, y, w, h) of the smallest face, used for drawing a bounding box around it later.

Then we loop through each frame, where we detect Faces using the function detectMultiScale() of OpenCV, which returns a list of rectangles around detected faces.

We keep track the smallest face and for each of them we calculate the face sizes by calculating its area (w * h). When we identify the smallest face in this frame by comparing the areas and if this smallest face is smaller than the previously tracked smallest face, we update our variables with the new smallest area. Finaly, we store the frame number and coordinates of this face, as well as a copy of the frame.

When we have the output of the smallest detected face, we use face coordinate variable to draw a rectangle around the smallest face in the stored image. To conclude we save this frame as an image and display it as well as the area of the smallest face and the frame it was taken.

Result of the program: The largest area detects by our model using our cascade.xml file is 1600 pixels which correspond to a square of 40 pixels.

Figure 6: Image of the Smallest Face Detected by Our Model

II. **Problem 2**: Maximum detectable face size.

The biggest face detection works similarly, but with a few differences in logic to track the largest face instead of the smallest. Here's how it operates:

First, we set up the video and face detector. As with the smallest face detection, the video is loaded and the face detection model (cascade.xml) is initialized.

We use tracking variables to keeps track of the area of the largest face found so far, to stores the frame number where this largest face was detected, to keep holds a copy of the frame with the largest face and to stores the (x, y, w, h) coordinates for the largest face.

Then we loop through each frame and detect faces using the detectMultiScale() function of OpenCV.

We keep track of the largest face by calculating the face sizes and its area for each detected face. Then we find the largest face in this frame by comparing the areas. And if this largest face is larger than the previously tracked largest face, we update the variable with the new largest area. We also store the frame number and coordinates of this face, along with a copy of the frame.

After looping through all frames, we use the variable to draw a rectangle/square around the largest face. And we save this frame as an image and display it.

Result of the program: The largest area detects by our model using our cascade.xml file is 276 676 pixels which correspond to a square of 526 pixels.

And here is a picture of the biggest detectable face of our model:

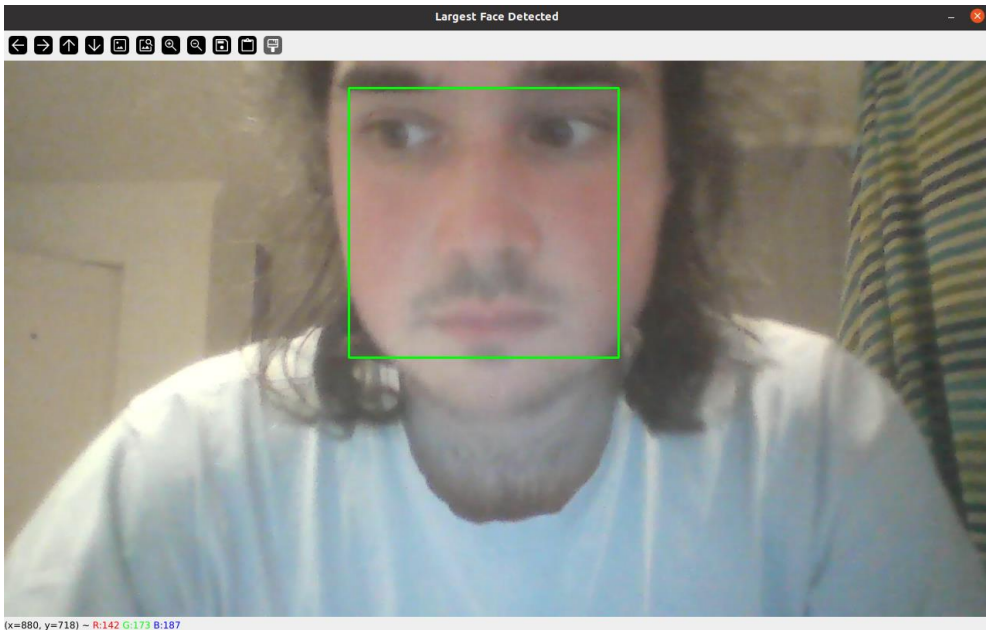


Figure 7: Image of the Largest Face Detected by Our Model

III. **Problem 3**: Average detection time (per image and per face).

To evaluate the average detection time of our face detection model, we developed a Python script using OpenCV. This script processes a series of test images, calculates the detection time for each image, and records the number of faces detected. By capturing these metrics, we can evaluate the model's detection speed per image and per face, offering insights into its responsiveness under varying conditions.

Additionally, to ensure accurate timing results, the first image (where model loading may influence timing) is excluded from the overall average calculation. The data is then analyzed to produce insights into the average processing time relative to the number of faces detected, with results visualized in a summary plot.

face_detector_time.py

```python
import cv2
import os
import time
import matplotlib.pyplot as plt
import pandas as pd

# Load the trained cascade model
cascade_path = "cascade.xml"  # The cascade file is in the same folder as the script
face_cascade = cv2.CascadeClassifier(cascade_path)

# Folder containing test images
image_folder = "image_test"  # Ensure this folder exists and contains the images
# Folder to save processed images
processed_folder = "processed_images"
os.makedirs(processed_folder, exist_ok=True)  # Create folder if it doesn't exist

# List to store data for each image
detection_data = []

# Loop through each image in the folder
for image_index, image_name in enumerate(os.listdir(image_folder), start=1):
    image_path = os.path.join(image_folder, image_name)
    image = cv2.imread(image_path)

    # Check if the image was loaded correctly
    if image is None:
        print(f"Error loading image: {image_path}")
        continue

    # Convert the image to grayscale
    gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # Start timing
    start_time = time.time()

    # Perform object detection
    detected_objects = face_cascade.detectMultiScale(
        gray_image,
        scaleFactor=1.1,
        minNeighbors=5,
        minSize=(30, 30)
    )

    # End timing
    process_time = time.time() - start_time

    # Number of faces detected
    num_faces = len(detected_objects)
    print(f"{num_faces} objects detected in {image_name} in {process_time:.4f} seconds")

    # Draw rectangles around detected objects
    for (x, y, w, h) in detected_objects:
        cv2.rectangle(image, (x, y), (x + w, y + h), (0, 255, 0), 2)
```

```python
    # Save the processed image in the new directory
    processed_image_path = os.path.join(processed_folder, image_name)
    cv2.imwrite(processed_image_path, image)

    # Add data to the table
    detection_data.append([image_index, num_faces, process_time])

# Close open windows
cv2.destroyAllWindows()

# Create a DataFrame to display the results
df = pd.DataFrame(detection_data, columns=['Image Number', 'Faces Detected', 'Processing Time (s)'])
print()
print(df)

# Exclude the first image from the statistics calculation
df_filtered = df.iloc[1:]  # Exclude the first row

# Calculate the average processing time for each number of faces detected, excluding the first image
average_times_filtered = df_filtered.groupby('Faces Detected')['Processing Time (s)'].mean()

# Convert Series to DataFrame for cleaner display
average_times_filtered_df = average_times_filtered.reset_index()
average_times_filtered_df.columns = ['Faces Detected', 'Average Processing Time (s)']
print("\nAverage processing time per number of faces detected (excluding the first image due to model loading time):")
print(average_times_filtered_df)

# Calculate the overall average detection time per image, excluding the first image
overall_average_time = df_filtered['Processing Time (s)'].mean()
print(f"\nOverall average detection time per image (excluding the first image): {overall_average_time:.4f} seconds")

# Plot the average processing time per number of faces detected
plt.figure(figsize=(10, 6))
plt.plot( *args: average_times_filtered.index, average_times_filtered.values, marker='o')
plt.xlabel("Number of Faces Detected")
plt.ylabel("Average Processing Time (s)")
plt.title("Average Processing Time vs. Number of Faces Detected")
plt.grid(True)
plt.show()
```

Figure 8: Screenshot of Python Code for Face Detection Time Calculation (Average Detection Time per Image and per Face) – Program Name: face_detector_time.py

	Image Number	Faces Detected	Processing Time (s)
0	1	0	1.523158
1	2	1	0.117893
2	3	5	0.174124
3	4	0	0.108611
4	5	2	0.142235
..	...	...	...
95	96	5	0.149982
96	97	2	0.101844
97	98	6	0.152844
98	99	3	0.077979
99	100	0	0.062533

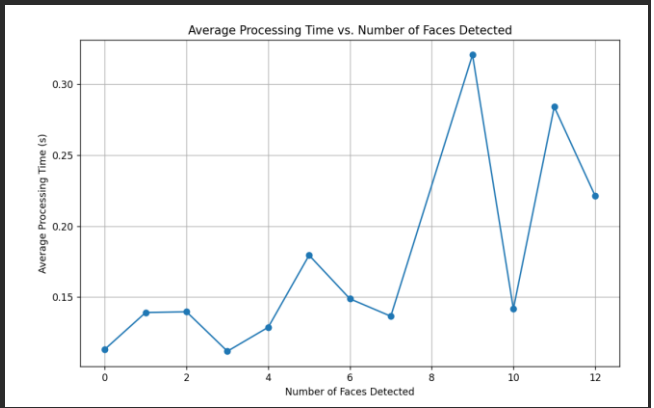
[100 rows x 3 columns]

Figure 9: Detection Time and Number of Faces Detected per Image Results

From the first table, we observe that the detection times vary per image, ranging from around 0.06 seconds to 1.52 seconds. This fluctuation can be attributed to factors like image complexity, the number of faces present, potentially the presence of non-facial elements affecting detection efficiency or for the first image the loading time of the model.

Average processing time per number of faces detected (excluding the first image due to model loading time):

	Faces Detected	Average Processing Time (s)
0	0	0.113252
1	1	0.139179
2	2	0.139683
3	3	0.111884
4	4	0.128774
5	5	0.179544
6	6	0.148812
7	7	0.136552
8	9	0.320752
9	10	0.141692
10	11	0.284114
11	12	0.221468



Overall average detection time per image (excluding the first image): 0.1386 seconds

Figure 10: Average Processing Time for Each Detected Face Count with Corresponding Graph

In the second table, the average processing time for each detected face count is calculated. When zero faces are detected, the average processing time is around 0.113 seconds. For one to fourth faces, the average processing time is quite stable, ranging between 0.11 and 0.14 seconds. However, as the number of faces increases, the processing time varies more significantly.

Notably, when eight faces are detected, the average processing time peaks at approximately 0.32 seconds. This suggests that processing time increases with the number of faces due to the additional detection effort required by the algorithm. However, this relationship is not strictly linear, as seen in some fluctuations at higher face counts.

In conclusion, excluding the initial image (to account for loading time), the model achieves an overall average detection time of approximately 0.1386 seconds per image, indicating relatively efficient performance. However, as the number of detected faces increases, particularly beyond eight faces, the processing time tends to rise, suggesting that the model's computational demand scales with face count. Additionally, variations in detection time for certain face counts indicate that image composition also influences processing efficiency, with some images requiring more computation depending on their content and complexity.

IV. False Detection Issue & Performance analysis

In order to determine the false detection issue in our model, we run though a number of testing data to classify the errors and pinpoint the root causes of them.

		Predicted	
		Positive	Negative
Actual	Positive	True Positive (TP)	False Negative (FN)
	Negative	False Positive (FP)	True Negative (TN)

Figure 11: Possible outcomes from the face detection classifier

- True Positive: Face detected as a face
- True Negative: Non-face not detected
- False Positive: Non-human face detected as human face
- False Negative: Human face not detected.

Error situations to consider

1) False Positives

False Positives occur when our model incorrectly classifies non-human faces as human. Check the following figures:

Figure 12: the constituents of the soup misclassified as faces.

Figure 13: Parts of a robot's body misclassified as human faces

2) False Negatives

False Negative occurs when human faces are not detected.

Figure 14: Author's face not being detected by the model

V. Challenges in Object Detection

Viewpoint Variation

Viewpoint variation in face detection refers to the different angles from which a face can be seen. This includes front views, side profiles, or even tilted and rotated faces.
As we can see in the following, our model can successfully detect both tilted faces and side profiles.

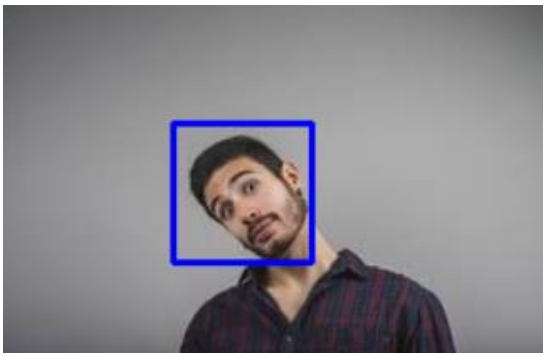


Figure 15: Tilted Face

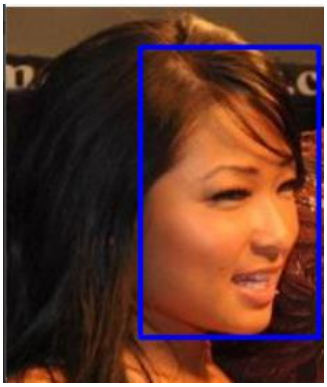


Figure 16: Side Profile

Occlusion

In face detection, occlusion occurs when parts of the face are blocked by objects like sunglasses, masks, hats, or other obstructions

Figure 17,18,19, 20: Testing the model with various occlusions

Illumination Changes

The model is tested under different lighting conditions to see its performance.

Figure 21: Face under bright sunlight

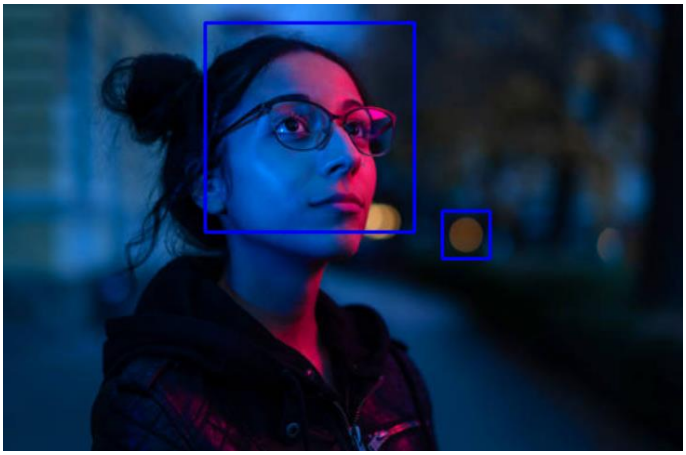


Figure 22: Face under low light setting

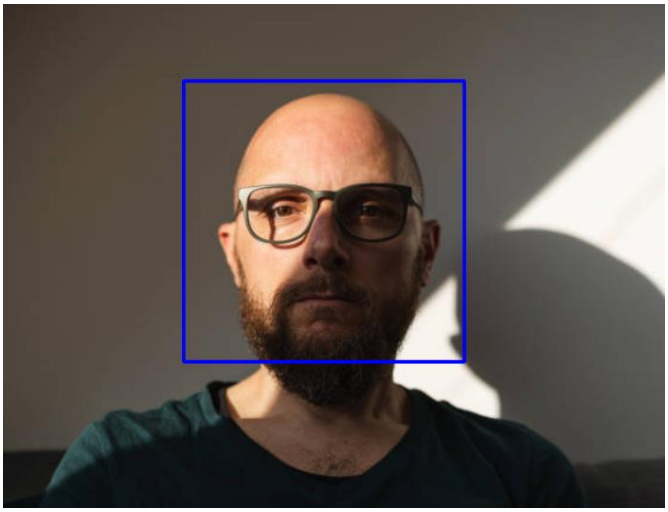
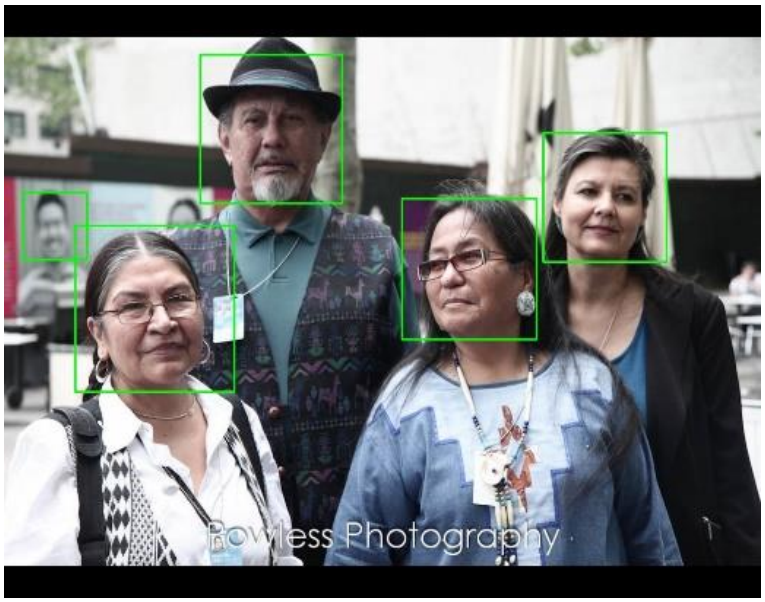


Figure 23: Sample with shadow cast over part of the face

Cluttered Backgrounds

Busy backgrounds can increase the likelihood of false positives (detecting non-face objects as faces) because patterns or textures in the background may resemble facial features. Our model performs efficiently even in the presence of multiple non-human things in the background. In figure 25, we can notice that this model even detects the face in the picture.



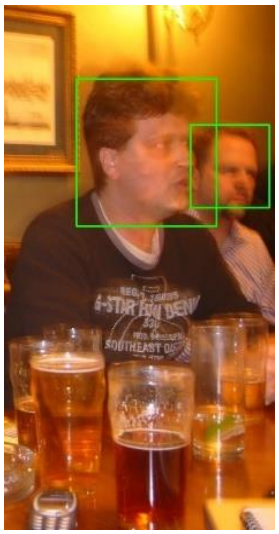
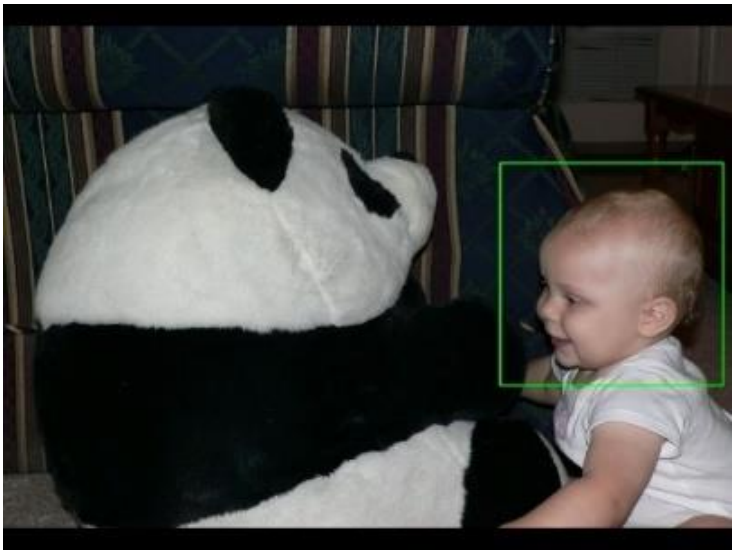


Figure 24,25: Model detecting correctly in cluttered environment

Intraclass variation

Intra-class variation in face detection includes differences in faces due to age, gender, ethnicity, facial expressions, and accessories (such as glasses or makeup).



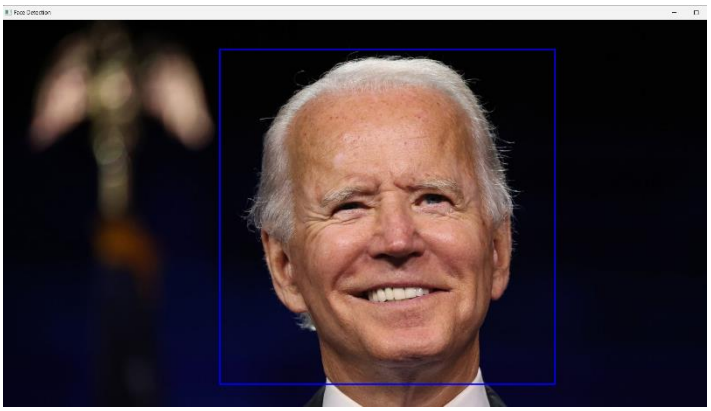


Figure 26,27: Intraclass Variation in different age groups

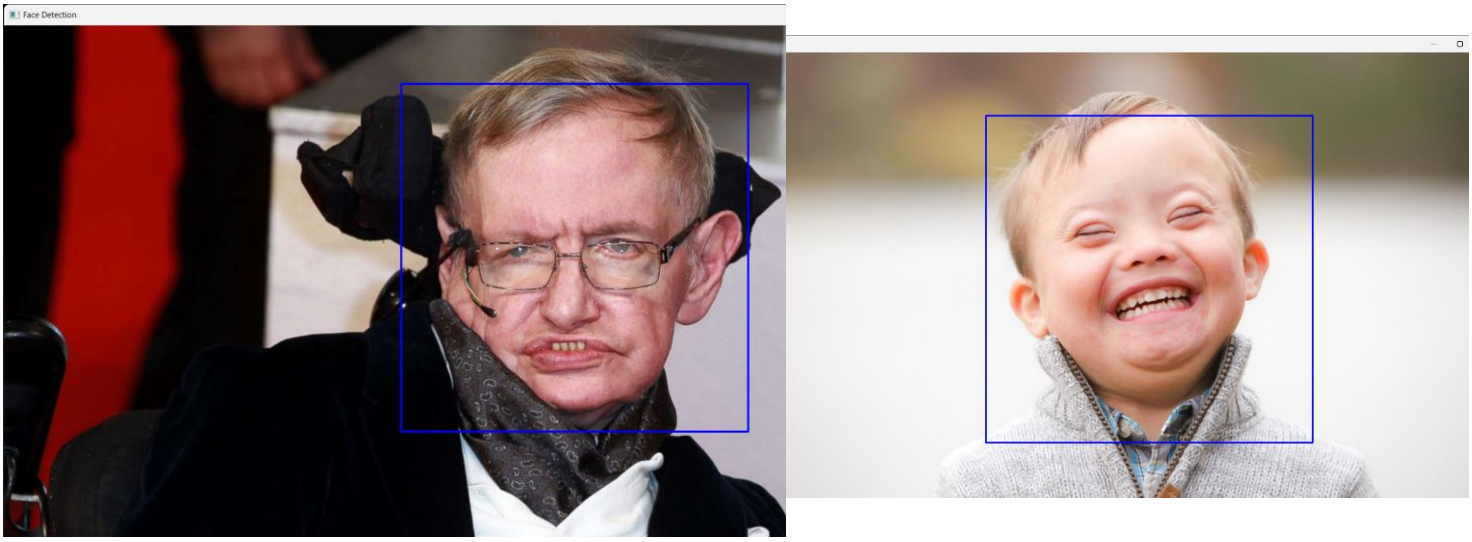


Figure 28,29: Demonstration of Intraclass Classification (People with irregular face features due to congenital or acquired health conditions)

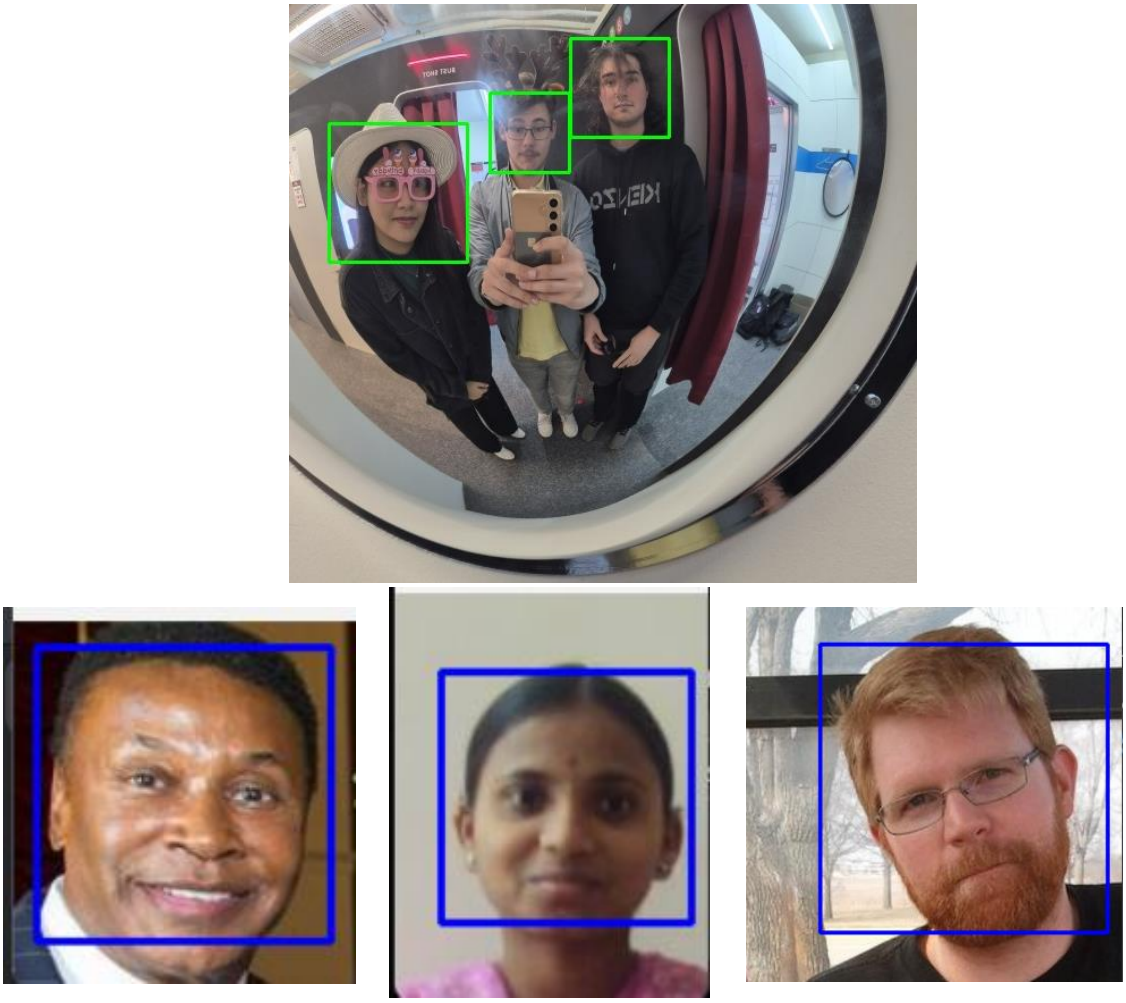


Figure 30,31,32,33: Demonstration of intraclass classification (differnet racial and ethnic groups)

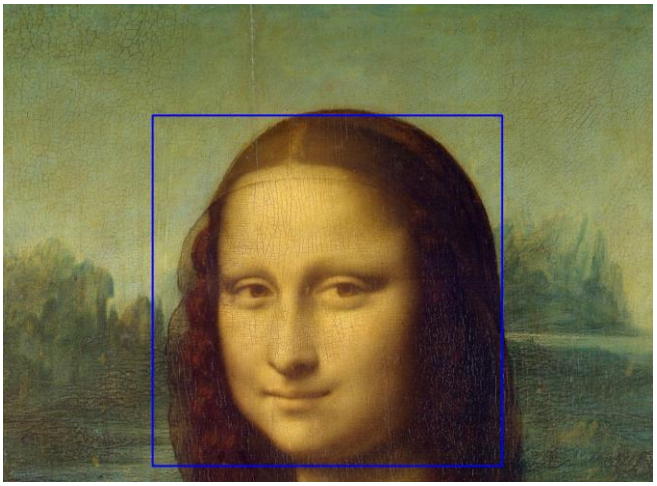


Figure 34,35,36: Demonstration of intraclass variation: Paintings(a) Random Abstract Painting from the Internet (b) The Scream by Edvard Munch (c) Monalisa by Leonardo da Vinci

We can see from figure 34 and 36 that our model can detect faces from hyper-realistic paintings and abstract pictures but will not detect the face that fails to meet the standard facial features as in figure 35.

V. Is Our Model Perfect?

Even though our model can identify faces correctly and successfully in most cases, it is still yet far away from perfection. I have listed some of the wrong detections (both false positives and false negatives below).

Samples of failed detection

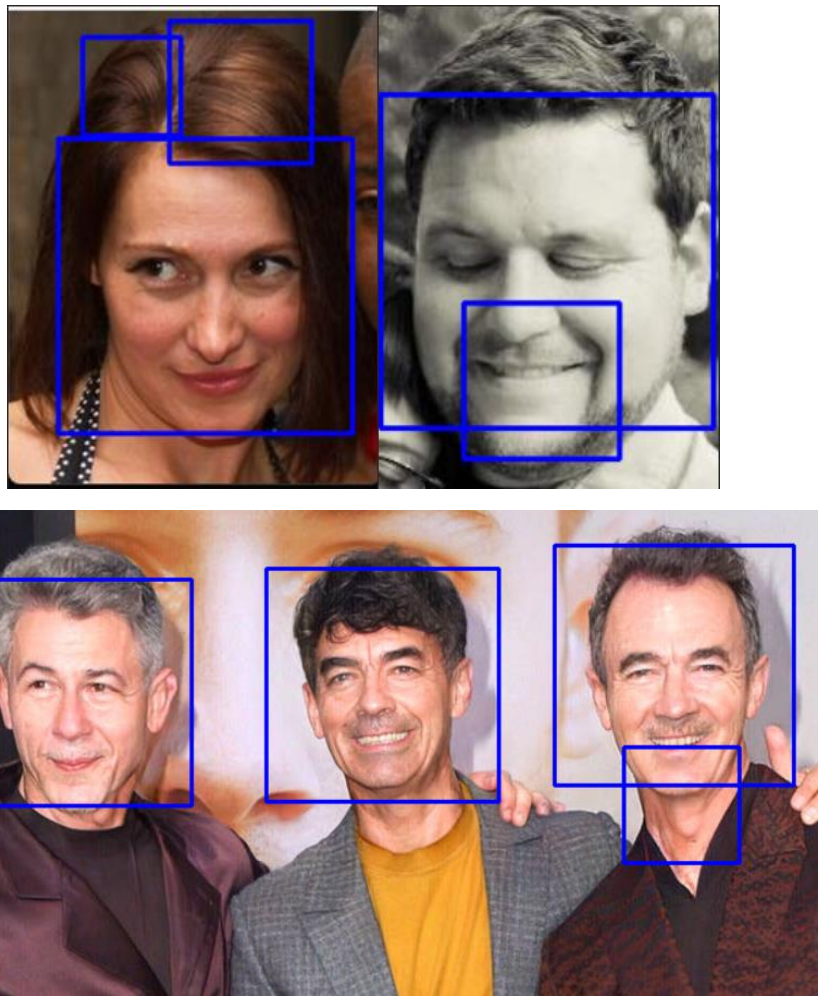


Figure 37, 38, 39: Model misinterpreting individual regions in the face as whole faces

Possible cause of such error:

a) Overfitting to features: Viola-Jones classifiers rely on Haar-like features that capture edges and regions of contrast. If the features selected during training aren't diverse or discriminative enough, they) due to might activate strongly on face parts (such as a chin or forehead similarities in texture or structure.)

b) Poor diversity in negative samples: If the negative samples in our training set are not diverse enough, the model may struggle to distinguish face parts from full faces. We have more than 3700 pictures for the negative dataset, but we did not include separate parts of the faces such as nose, chin, etc in our “n” set.

To clear up my doubt that that dataset is not diverse enough, I ran the model on some faces with uncommon features.

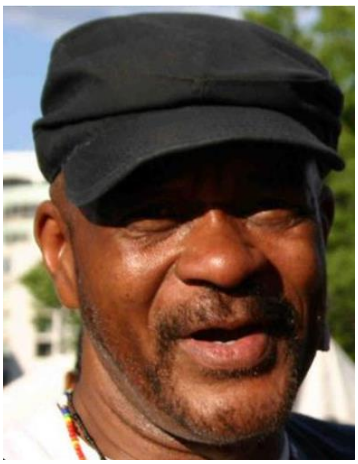


Figure 40: Shadowing of the cap covering part of the face

Figure 41: A girl with a pink wig

Figure 42: A girl wearing accessories for a special ceremony

VI. Conclusion

In conclusion, our face detection model, based on the Viola-Jones algorithm, demonstrates promising capabilities in accurately identifying faces across various settings, including changes in viewpoint, lighting, occlusions, and intraclass variations. While our model performed effectively in detecting faces in cluttered backgrounds and under diverse lighting conditions, certain limitations were evident. False positives, such as the misinterpretation of non-facial areas as faces, and false negatives, especially with unique or partially obstructed faces, highlight areas for potential improvement. We identified that the diversity of training data plays a critical role in enhancing model robustness. Additional negative samples, including partial face features, may help reduce errors. This project illustrates the strengths and limitations of traditional face detection algorithms and provides a foundation for future improvements, such as integrating deeper machine learning models that may further enhance accuracy and adaptability in complex environments.

Figure 43: Project Team: Chaw Thiri San, Cyrius, and Guillaume