Wrocław University of Science and Technology

Human Face Gesture Recognition Model

Artifical Inteligence and Computer Vision Project

Yasin Aslan (276719) Software Developement

Pinar Aydin (276766) Testing and Debugging

Submission date-13.01.2025

Contents

1. Introduction

1.1 Project description

This 3ort he is analyzing facial features to detect emotions based on expressions. It identifies specific points on the face, processes their positions, and determines whether the person in the image looks happy, sad, or neutral. The output is an annotated image showing facial points and the detected emotion.

The subsequent steps are the following:

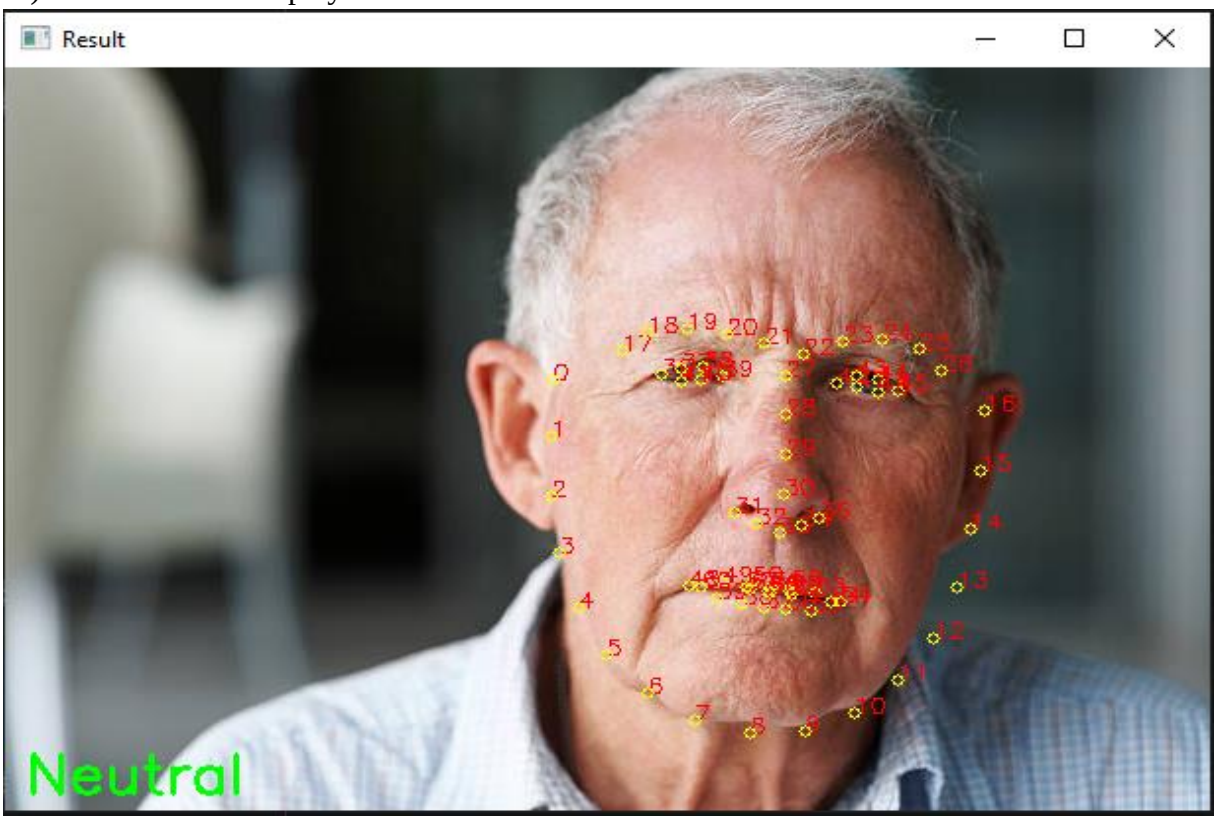
1-) We input a 3ort he of the person whose gestures we want to detect into our model.

2-) With preferred pre trained face recognition model we recgognize human face.

3-) We apply needed filters or collor changings for make our gesture recognition model's detections easier.

4-) Our Recognition model finds human gestures by looking mouth shape.

5-) And our model displays result on Picture that we use and shows it.



1.2 Case studies

The most useful resource for our project was “Face Recognition in Python: A Comprehensive Guide” by Basil CM on Medium. This website provided us with extensive information and guidance, helping us create a roadmap for our project. We found several similar projects on this platform, which were incredibly useful for building a strong foundation and portfolio for our work.

Initially, we needed to explore various filters to make human faces more recognizable, as our project aimed to work with live video capture. This focus limited our ability to find resources since most similar projects were designed for processing static images. However, as we delved deeper into image processing and computer vision, we realized that the working principles for images and live video capture were quite similar. This understanding allowed us to adapt and apply those techniques to our project.

We also relied heavily on OpenCV: OpenCV-Python Tutorials. While some explanations in the documentation were written for C++, once we understood the core concepts, it became easier to apply them to our Python-based implementation. For more complex topics, we turned to the vast information available online, which helped us clarify and deal with challenges.

Ultimately, while we applied only a small portion of what we learned, this journey significantly expanded our knowledge of computer vision and image processing.

1.3 How do we tried to solve the problem

1 **Thresholding and Filters**
We initially tried various thresholding intensity values to enhance the visibility of the lip line. After converting the video capture frame to grayscale, we applied thresholding, but it didn't give us the desired results. We then experimented with the Sobel filter, hoping it would improve the edge detection, but this approach also failed to provide the expected outcome.

2 **Using a Higher-Quality Camera**
Next, we decided to use a higher-quality camera to capture better video frames for analysis. We switched to smartphone cameras, hoping the improved data quality would help, but even with this adjustment, we still faced difficulties and couldn't achieve the desired results.

3 **Capturing Pictures from Video**
As a last resort, we attempted to capture still images from the video stream. Our plan was to apply the lip line detection process on these images, hoping that working with static frames would yield better results. Despite this change, the challenge persisted, and we didn't achieve the expected accuracy.

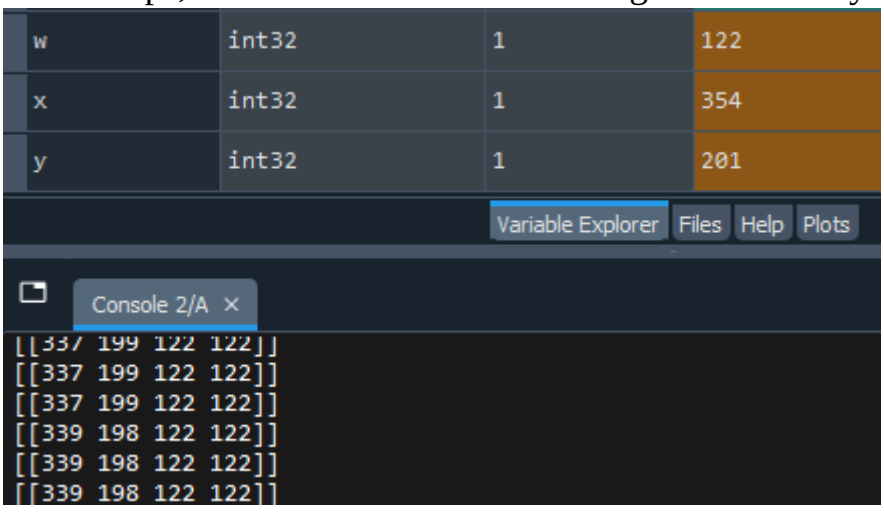
1.4 How we actually solve the problem

At the end, we solved the problem by leveraging a more accurate method for facial landmark detection using dlib's pre-trained shape predictor. This allowed us to detect the specific coordinates of facial landmarks, including key areas around the mouth, which were crucial for analyzing lip movement. By calculating the vertical positions of certain points on the mouth, such as the corners and the center of the mouth opening, we were able to assess the relative position of these points to determine emotions like neutral, sad, or happy based on the mouth's configuration. This approach provided us with a much more

2.1 First Approach

In our first approach, we frequently changed our ideas, almost like a 6ort routine. We used the HaarCascade face recognition model, which only detects human faces. With this method, we were only able to get information about the location of the face in the image, represented as a coordinate system.

We wanted to detect human emotions by analyzing the shape of the lips. We chose this approach because we believed that the lips exhibit the most significant changes during shifts in expressions. By detecting the line between the two lips, we aimed to track these changes and identify different gestures.



We were having x,y,w,h values of face.

2.2 Used tools

To find the lip line, we used the coordinate information stored in Python for the human face. The idea was that the lips always remain in the same position below the nose on every person's face. By performing quick coordinate calculations, we could approximate the location of the lips.

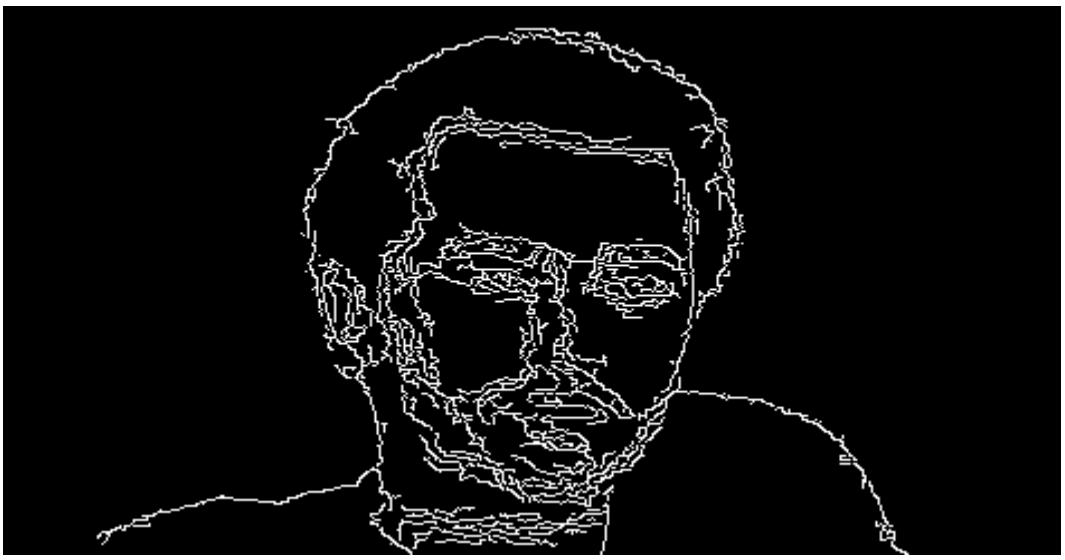
Here is the method we used for this:

```
faces = face_cascade.detectMultiScale(gray, scaleFactor=1.3, minNeighbors=5)

for (x, y, w, h) in faces:
    cv2.rectangle(thresh, (x, y), (x + w, y + h), (0, 255, 0), 2)

    lip_y_start = int(y + h * 0.65)
    lip_y_end = int(y + h * 0.9)
    lip_x_start = int(x + w * 0.2)
    lip_x_end = int(x + w * 0.8)

    cv2.rectangle(thresh, (lip_x_start, lip_y_start), (lip_x_end, lip_y_end), (0, 0, 255), 2)

    lip_region = thresh[lip_y_start:lip_y_end, lip_x_start:lip_x_end]
```

In this method, after identifying the approximate region of the lips, we applied a Region of Interest (ROI) to focus on the specific part of the face we wanted to analyze for gesture tracking. To detect the lip line, we used Canny Edge Detection, at least for a neutral face. However, after applying Canny Edge Detection to our ROI, the results were unsatisfactory. Testing in various lighting conditions revealed that this approach required precise configurations to work effectively. This led us to explore filters to make the lip line more visible.

We first applied thresholding to improve the visibility:

```
faces = face_cascade.detectMultiScale(gray, scaleFactor=1.3, minNeighbors=5)

edges = cv2.Canny(gray, 40, 120)
```



We experimented with various threshold intensity values to enhance the visibility of the lip line. Initially, we converted the video capture frames to grayscale and applied thresholding, but the results were unsatisfactory. We also tried using the Sobel Filter, another well-known method, but it did not yield the expected outcomes either.

To improve the quality of the input data, we opted for a higher-quality camera, using smartphone cameras to capture better video frames for analysis. Despite the improved data quality, this attempt also ended in failure. During this time, we explored alternative methods for lip detection and experimented with contour detection based on lip color. As part of our testing, we even used lipstick to make the lips more distinguishable. While this approach improved lip visibility to some extent, it wasn't significantly better than simply identifying the approximate lip region. It barely managed to detect the lips, providing no meaningful insights into the movement of the lip line.

As a final attempt, we decided to capture a frame from the video feed. Our plan was to apply the line detection process to this frame. So basicly when user press c button in keyboard program takes a screenshot from video capture and applied methods on that Picture. Here is the code for video capture:

```
if key == ord('c'):
    if len(faces) > 0:
        captured_frame = frame.copy()
        for (x, y, w, h) in faces:
            cv2.rectangle(captured_frame, (x, y), (x + w, y + h), (255, 0, 0), 2)


            roi = frame[y:y + h, x:x + w]
            gray_roi = cv2.cvtColor(roi, cv2.COLOR_BGR2GRAY)

            lip_mask = cv2.inRange(gray_roi, lower_intensity, upper_intensity)

            lip_mask = cv2.dilate(lip_mask, kernel, iterations=2)
            lip_mask = cv2.erode(lip_mask, kernel, iterations=1)

            contours, _ = cv2.findContours(lip_mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
            for contour in contours:
                area = cv2.contourArea(contour)
                if 500 < area < 5000:
                    cv2.drawContours(captured_frame, [contour], -1, (0, 255, 0), 2)


        cv2.imshow('Captured Photo', captured_frame)
        cv2.waitKey(5000)
        cv2.destroyWindow('Captured Photo')
    else:
        print("No faces detected to capture.")
```

The results were complete failure so we decided to apply another method:

2.3 Troubles

In the first approach, we encountered several challenges while trying to detect facial gestures. Initially, we used the HaarCascade's face recognition model, which only detected the presence of a face but didn't provide enough detail to analyze emotions. This limited us to merely identifying the location of the face within the image as coordinates, without any further insights into the facial expression.

We also faced difficulties in detecting the lip line using simple image processing techniques like Canny edge detection. Although we applied this method to the region of interest (ROI) around the mouth, the results were unsatisfactory, as the lip line remained unclear and inconsistent under different lighting conditions. We also tried adjusting the thresholding intensity values and applied Sobel filtering, but neither solution improved the accuracy of lip detection.

Furthermore, when we switched to using a higher-quality camera to capture better video frames, the improved data still didn't give us the desired results. Despite using the smartphone camera for better quality, we still struggled to

detect lip movement accurately. This led us to reconsider our approach entirely and explore alternative methods for facial expression recognition.

2.4 Second Approach

After we failed to obtain reliable data using HaarCascade's face recognition model and struggled with detecting the lipline through various filters, we decided to switch to a completely different face recognition model in hopes of obtaining more accurate and reliable data. Our goal shifted to making the model adaptable to different circumstances. Following discussions with our professor, we decided to change the focus of the project and work with static pictures rather than live video capture.

Here is a detailed explanation of each step in our final code:

```python
import cv2
import dlib
import numpy as np

PREDICTOR_PATH = "C:/faceproje/shape_predictor_68_face_landmarks.dat"
predictor = dlib.shape_predictor(PREDICTOR_PATH)
detector = dlib.get_frontal_face_detector()

class TooManyFaces(Exception):
    pass

class NoFaces(Exception):
    pass

def get_landmarks(im):
    rects = detector(im, 1)

    if len(rects) > 1:
        raise TooManyFaces
    if len(rects) == 0:
        raise NoFaces

    return np.matrix([[p.x, p.y] for p in predictor(im, rects[0]).parts()])
```

This code is designed to detect faces in an image and extract key facial landmarks using the dlib library. The primary goal of the code is to identify the locations of 68 distinct points on the face, such as the eyes, nose, mouth, and jawline, which can then be used for further facial analysis or emotion detection. The code first imports the necessary tools: a face detector to locate faces in an image, and a landmark predictor to identify the facial features after detecting the face. It uses a pre-trained model (shape_predictor_68_face_landmarks.dat) to predict the coordinates of the facial landmarks.

The Dlib's 68 Face Landmark Model is a pre-trained model that detects 68 specific points (landmarks) on a human face. These landmarks correspond to key facial features like the eyebrows, eyes, nose, mouth, and jawline. Here's how the model works:

Face Detection:
The process begins with detecting the face in an image or video frame. Dlib uses a Histogram of Oriented Gradients (HOG)-based face detector or a CNN-based detector to locate the bounding box around the face.

Landmark Localization:
Once the face is detected, the model uses a pre-trained shape predictor to locate the 68 specific points on the detected face. The model employs an Ensemble of Regression Trees (ERT) to predict the precise positions of these landmarks. These points are defined in a standard order:

Points 1-17: Define the jawline.

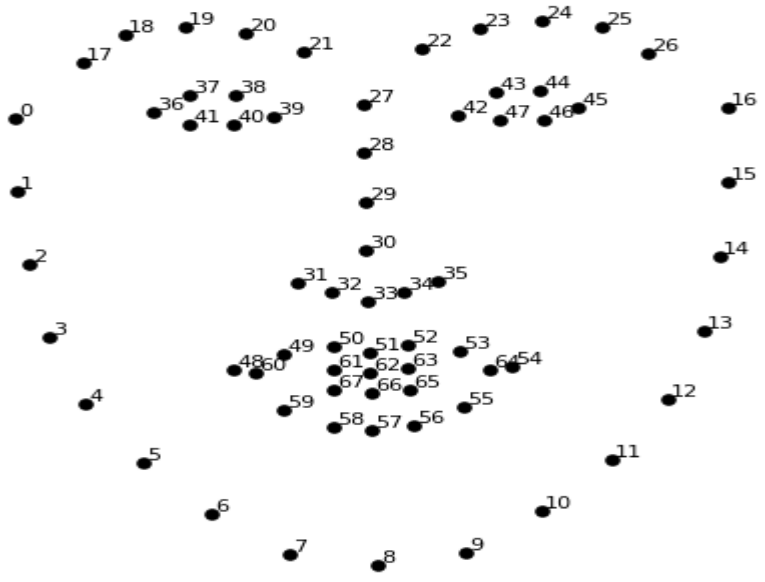
Points 18-27: Define the eyebrows.

Points 28-36: Define the nose.

Points 37-48: Define the eyes.

Points 49-68: Define the mouth.

Coordinate System:
The model returns the landmarks as 2D coordinates (x, y) in the image's pixel space. The origin (0, 0) is at the top-left corner of the image, with x increasing to the right and y increasing downward.



```python
def annotate_landmarks(im, landmarks):
    im = im.copy()
    for idx, point in enumerate(landmarks):
        pos = (point[0, 0], point[0, 1])
        cv2.putText(im, str(idx), pos,
                    fontFace=cv2.FONT_HERSHEY_SCRIPT_SIMPLEX,
                    fontScale=0.4,
                    color=(0, 0, 255))
        cv2.circle(im, pos, 3, color=(0, 255, 255))
    return im
```

The function annotate_landmarks(im, landmarks) is responsible for adding annotations to the image by marking the detected facial landmarks. It starts by creating a copy of the input image to avoid modifying the original. Then, it iterates through each landmark in the landmarks array, retrieving both the index and the coordinates of the landmark. For each landmark, the coordinates are used to determine where to place a text label showing the landmark's index on the image. Additionally, a small circle is drawn at each landmark position to visually highlight them. This annotated image is then returned for further use, such as saving or displaying.

```python
def detect_emotion(landmarks):
    tolerance = 2

    mouth_left_corner_y = landmarks[48,1]
    mouth_right_corner_y = landmarks[54,1]
    mouth_opening_center_top_y = landmarks[62,1]
    mouth_opening_center_bottom_y = landmarks[66,1]

    mouth_center_y = (mouth_left_corner_y + mouth_right_corner_y)/2
    mouth_opening_center_y = (mouth_opening_center_top_y + mouth_opening_center_bottom_y )/2

    if abs(mouth_center_y - mouth_opening_center_y) <= tolerance:
        return 'neutral'
    elif mouth_center_y > mouth_opening_center_y:
        return 'sad'
    elif mouth_center_y < mouth_opening_center_y:
        return 'happy'
    else:
        return 'unexpected result'
```

The detect_emotion function evaluates the emotion based on the vertical positions of specific landmarks around the mouth. It first calculates the y-coordinates of the left and right corners of the mouth, as well as the top and bottom of the mouth opening. These values are averaged to determine the mouth's center and the center of the mouth opening. By comparing these two values, the function determines whether the person is neutral, sad, or happy. If the difference between the centers is small (within a tolerance), the emotion is classified as neutral. If the mouth center is above the opening center, it indicates a sad expression, while if the mouth center is below, it indicates a happy

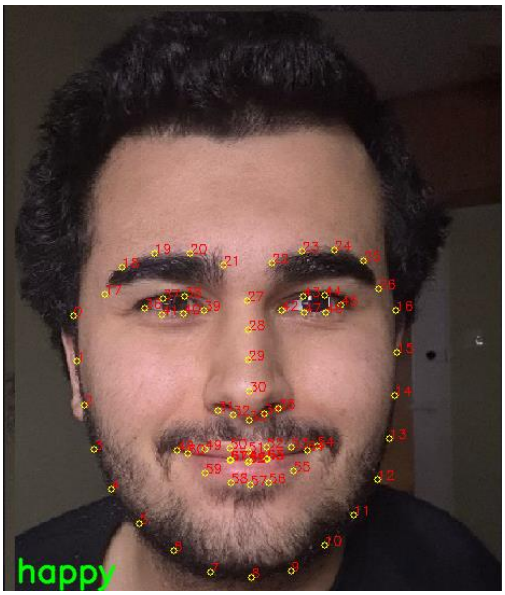
expression. The function is focused on analyzing vertical positions, as they are more relevant for detecting location of the lip corners. If you look at the mathematical operation you might notice that happy and sad emotion's mathematical conditions were mirorred this is because dlib's face recognition model's x =0 and y=0 spot is upper left corner that is why we needed to mirror it.

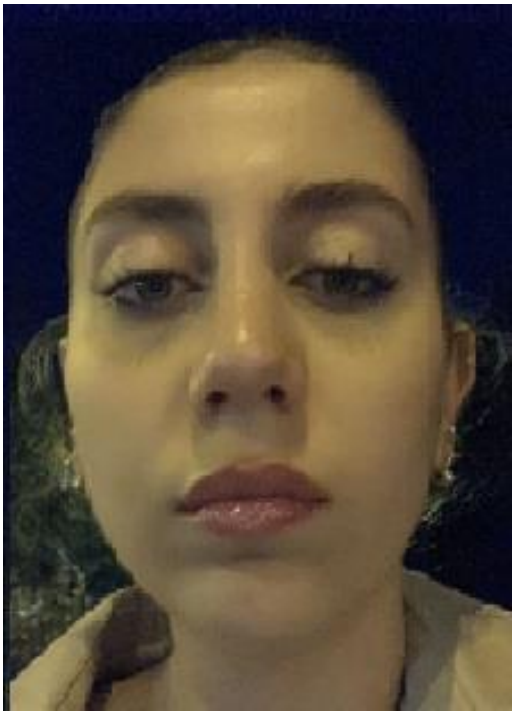
```
landmarks = get_landmarks(image)

emotion = detect_emotion(landmarks)

image_with_landmarks = annotate_landmarks(image, landmarks)


font = cv2.FONT_HERSHEY_SIMPLEX
cv2.putText(image_with_landmarks, emotion, (10, image_with_landmarks.shape[0] - 10),
            font, 1, (0, 255, 0), 2, cv2.LINE_AA)
```

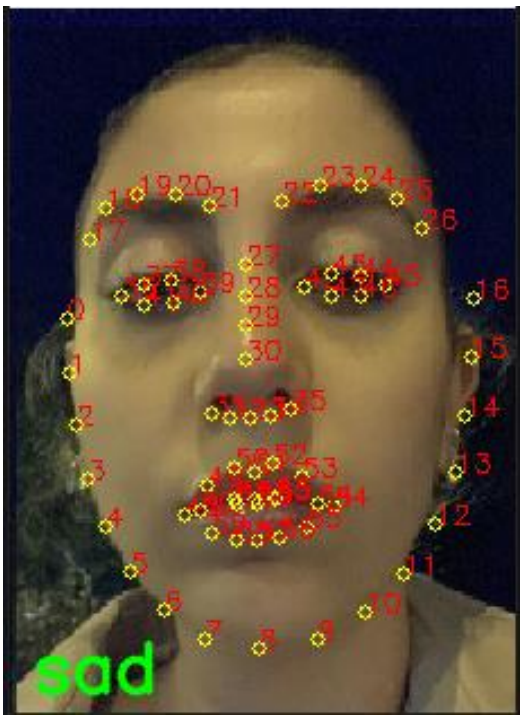
In this part of the code, the image is processed to detect facial landmarks, classify the emotion, and then annotate the image with the results.

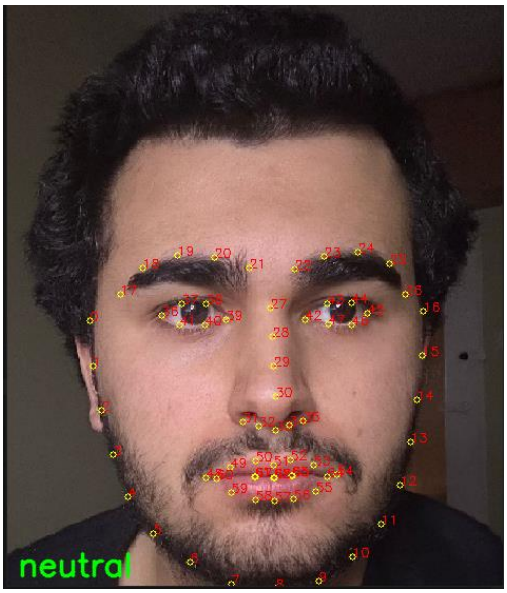
First, the image is read using cv2.imread from the specified path and stored in the image variable. Next, the function get_landmarks(image) is called, which detects the facial landmarks and stores them in the landmarks variable. These landmarks are used in the following step to detect the emotion by calling the detect_emotion(landmarks) function, which analyzes the mouth position to determine whether the emotion is neutral, sad, or happy, and stores the result in the emotion variable.

After that, the annotate_landmarks(image, landmarks) function is used to draw circles and label the landmarks on the image. Finally, the emotion is displayed on the image using cv2.putText, which writes the detected emotion (such as "neutral," "sad," or "happy") at the bottom left corner of the image. The text is drawn in green with a font size of 1, a thickness of 2, and anti-aliasing enabled (cv2.LINE_AA).


happy




sad



3.1 Initial Project vs Final Project

Objective:

The initial goal was to analyze lip movements and detect emotions in real-time using video capture.We planned to use Haar Cascade for face detection and apply image processing techniques like filters and thresholding to extract the lip line and determine the emotional state based on lip movement.

Method:

Face detection using Haar Cascade to locate faces in video frames.Use of image processing techniques like thresholding and Sobel filtering to enhance lip visibility and track lip movements.Analyze changes in lip shape or movement to determine emotional states (happy, sad, neutral).

Challenges:

Haar Cascade failed to provide accurate face detection.Filters and thresholding did not give satisfactory results in detecting the lip line, especially under varying conditions (lighting, lip color, etc.).Real-time video analysis did not provide stable or reliable data for emotion detection.

Final Project:

Objective:

After facing challenges with the initial approach, the focus shifted towards detecting emotions through static images using a more reliable face detection model.The project aimed to classify emotions based on mouth movement by analyzing facial landmarks.

Method:

Replaced Haar Cascade with a more accurate face detection model (dlib's 68-point landmark detector) for robust and consistent landmark extraction.Used static images instead of real-time video capture to simplify processing and improve data quality.The mouth region’s landmarks were analyzed to detect emotions like "neutral," "sad," or "happy" based on the vertical position of key mouth points.Annotated images with facial landmarks and emotion classification were displayed and saved.

Outcomes:

The final model successfully detected emotions (neutral, sad, or happy) based on facial landmarks, particularly the position of the mouth.The system processed static images and provided reliable results in different circumstances (lighting, camera quality, etc.).Emotions were displayed on the images with clear annotation, ensuring that the system could be used for further analysis or applications.

3.2 Overall experience

Throughout the project, we faced several challenges that significantly influenced our approach and decision-making. Initially, we aimed for a more complex solution involving real-time video analysis and dynamic emotion detection using Haar Cascade for face detection. However, we encountered limitations with Haar Cascade’s accuracy and the complexity of processing lip movements in real-time. These issues led to unreliable results, and we realized the need for a more robust solution.

After discussing our difficulties with our professor, we decided to shift our focus to analyzing static images instead of working with live video. This change allowed us to simplify the problem while ensuring more consistent results. By adopting dlib’s face detector and 68-point landmark model, we achieved much better accuracy in detecting facial features, particularly the mouth region, which is crucial for emotion classification.

The final system was able to reliably classify emotions as "neutral," "sad," or "happy" based on the analysis of mouth position, which was a significant improvement from our initial approach. This success not only validated the

effectiveness of using facial landmarks for emotion detection but also taught us valuable lessons about the importance of adapting to challenges and adjusting the project scope when needed.

3.3 Bibliography

1 – Medium Python Face Detection

Face Recognition in Python: A Comprehensive Guide | by Basil CM | Medium

2 – OpenCV Documentation

OpenCV: OpenCV-Python Tutorials

3 – Tutorial videos from YouTube

Face Landmark Detection using dlib - Python OpenCV

4 – Line detection Basics

Wykrywanie linii w pythonie z OpenCV | Metoda Houghline - GeeksforGeeks