Student: Aditya Sandhu

University: Colorado State University Global

Course: 25FA - CSC515 - 1 [Module 8 – Computer Vision Applications and Pre-Trained Classifiers]

FINAL - Portfolio Assignment [Option # 2 - Face Detection and Privacy]

Instructor: Dr. Dong Nguyen

Date – 10/29/2025

GIT LINKS

Python File - [25FC-CSC515-1/MODULE8/csc515-1-final-module8-portfolio-project-final-aditya-sandhu15.py at main · 65AR645ASAN/25FC-CSC515-1](https://github.com/65AR645ASAN/25FC-CSC515-1/blob/main/MODULE8/csc515-1-final-module8-portfolio-project-final-aditya-sandhu15.py)

Document - [25FC-CSC515-1/MODULE8/csc515-1-final-module8-portfolio-assignment-aditya-sandhu.docx at main · 65AR645ASAN/25FC-CSC515-1](https://github.com/65AR645ASAN/25FC-CSC515-1/blob/main/MODULE8/csc515-1-final-module8-portfolio-assignment-aditya-sandhu.docx)

In an era where digital imagery is ubiquitous and privacy concerns are paramount, data anonymization techniques play a crucial role in safeguarding personal identities by obscuring identifiable features such as facial elements. This portfolio project develops a Python-based Face-Anonymizer script utilizing OpenCV's Haar cascade classifiers to detect frontal human faces in grayscale images and subsequently blur the eye regions for enhanced privacy protection. By processing three diverse color images, selected to include a non-human subject (animal), a full-body single male, and a group of front-facing individuals with varying distances, illuminations, and color intensities. the algorithm applies preprocessing steps like Gaussian blurring and CLAHE contrast enhancement for optimal detection accuracy, draws red bounding boxes around validated faces (confirmed via eye detection), and employs Gaussian blurring to anonymize eyes, demonstrating a practical application of computer vision in ethical image handling.

Figure 1- *Output of the Face-Anonymizer Script Showing Detected and Anonymized Facial Regions.*

A screenshot of a computer program

AI-generated content may be incorrect.

The image demonstrates the successful detection of human faces using OpenCV’s pre-trained Haar cascade classifiers. Red bounding boxes indicate the regions identified as faces, while the eye regions within those faces are blurred to preserve privacy. Non-human subjects are ignored, confirming the robustness of the detection and anonymization pipeline.

*import* cv2  
*import* os  
  
*# Configuration Constants*FACE\_XML = 'haarcascade\_frontalface\_alt2.xml'  
EYE\_GLASSES\_XML = 'haarcascade\_eye\_tree\_eyeglasses.xml'  
SRC\_FOLDER = 'portfolio-images'  
DEST\_FOLDER = 'detected-images'  
  
TARGET\_IMAGES = [  
 'animal-1.jpg',  
 'group-front-standing-group-1.jpg',  
 'full-body-single-person-male-1.jpg'  
]

The configuration section of python code above defines essential constants used throughout the Face-Anonymizer program. It specifies the Haar cascade XML files for face and eye-with-glasses detection, ensuring accurate identification of facial features using OpenCV’s pre-trained classifiers. The SRC\_FOLDER and DEST\_FOLDER variables designate the input and output directories for image processing, while TARGET\_IMAGES lists the specific images to be analyzed and anonymized. This setup provides a clear and modular foundation for the detection pipeline.

*def* get\_classifier(model\_name, xml\_path): *if not* os.path.exists(xml\_path):  
 *raise* FileNotFoundError(  
 "Required model '{}' not found. "  
 "Download from OpenCV GitHub and place in project root.".format(model\_name)  
 )  
 classifier = cv2.CascadeClassifier(xml\_path)  
 *if* classifier.empty():  
 *raise* RuntimeError("Failed to initialize {} model.".format(model\_name))  
 *return* classifier

The get\_classifier() function is responsible for securely loading OpenCV’s pre-trained Haar cascade models used for face and eye detection. It first verifies the existence of the specified XML file, raising a FileNotFoundError if the file is missing. Once validated, the function initializes the classifier and checks whether it has loaded correctly, raising a RuntimeError if initialization fails. By performing these validation steps, the function ensures reliable model loading and prevents runtime errors during image analysis.

Figure 2 - *Initialization and Preprocessing Functions for the Face-Anonymizer Pipeline.*

A screenshot of a computer program

AI-generated content may be incorrect.

In Figure 2. We have the following code –

*# Load required models*face\_detector = get\_classifier("frontal face", FACE\_XML)  
eye\_glasses\_detector = get\_classifier("eye glasses", EYE\_GLASSES\_XML)  
print("\nAll detection models loaded successfully.\n")  
os.makedirs(DEST\_FOLDER, exist\_ok=*True*)

This segment initializes the essential detection models required for the Face-Anonymizer pipeline. The get\_classifier() function loads the pre-trained Haar cascade classifiers for detecting frontal faces and eyes with eyeglasses. A confirmation message is printed upon successful model loading, and the destination directory is created using os.makedirs() to ensure processed images are stored without errors. This setup prepares the environment for subsequent image processing and anonymization operations.

*def* ld\_pic(file\_path):  
 picture = cv2.imread(file\_path)  
 *if* picture *is None*:  
 *raise* IOError("Unable to load image from: {}".format(file\_path))  
 *return* picture  
*# Utility: Optimize image for feature extraction  
def* opt\_img(input\_picture):  
 gry\_vsn = cv2.cvtColor(input\_picture, cv2.COLOR\_BGR2GRAY)  
 smth\_vsn = cv2.GaussianBlur(gry\_vsn, (7, 7), 0)  
 ctrst\_enhancer = cv2.createCLAHE(clipLimit=3.0, tileGridSize=(8, 8))  
 *# Apply enhancement and return the result* optmzd = ctrst\_enhancer.apply(smth\_vsn)  
 *return* optmzd

The ld\_pic() function is designed to safely load an image from the specified file path using OpenCV’s imread() method. It verifies that the image has been successfully read; if not, it raises an IOError, preventing the program from continuing with invalid data. This ensures reliability and error handling during the image import process.

The opt\_img() function enhances the input image to improve feature extraction accuracy. It converts the image to grayscale, applies Gaussian blurring to reduce noise, and uses Contrast Limited Adaptive Histogram Equalization (CLAHE) to enhance local contrast. The resulting optimized image improves the performance of facial feature detection algorithms.

Figure 3 - *Facial and Eye Region Detection with Anonymization Processing.*

A screenshot of a computer program

AI-generated content may be incorrect.

*def* search\_components(optmzd\_gray, base\_picture):  
suspected\_areas = face\_detector.detectMultiScale(  
 optmzd\_gray,  
 scaleFactor=1.05,  
 minNeighbors=8,  
 minSize=(40, 40),  
 maxSize=(400, 400),  
 flags=cv2.CASCADE\_SCALE\_IMAGE  
 )  
  
 authenticated = 0  
 *for* (loc\_x, loc\_y, size\_w, size\_h) *in* suspected\_areas:  
 sub\_rgb = base\_picture[loc\_y:loc\_y + size\_h, loc\_x:loc\_x + size\_w]  
 sub\_mono = optmzd\_gray[loc\_y:loc\_y + size\_h, loc\_x:loc\_x + size\_w]  
  
 *# Additional enhancement on face ROI for better eye detection* clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))  
 sub\_mono = clahe.apply(sub\_mono)  
  
 *# Restrict to upper 70% of face to include more area for tilted heads* eye\_h = int(size\_h \* 0.7)  
 sub\_eye\_mono = sub\_mono[0:eye\_h, 0:size\_w]  
  
 component\_locs = eye\_glasses\_detector.detectMultiScale(  
 sub\_eye\_mono,  
 scaleFactor=1.1,  
 minNeighbors=4,  
 minSize=(15, 15)  
 )

The search\_components() function performs the core facial detection and anonymization process in the Face-Anonymizer pipeline. It begins by scanning the optimized grayscale image for potential face regions using OpenCV’s detectMultiScale() method. For each detected region, it applies local contrast enhancement (CLAHE) and focuses on the upper 70% of the area to improve eye detection accuracy, even with tilted faces. Using the eye\_glasses\_detector, it validates the presence of eyes within these regions, marking confirmed faces and preparing them for subsequent anonymization through blurring. This multi-stage verification ensures precision and robustness in detecting and anonymizing human faces.

*# Collect valid eyes (no further filtering needed with glasses-specific detector)* valid\_eyes = []  
 *for* (cx, cy, cw, ch) *in* component\_locs:  
 *if* cy + ch <= eye\_h: *# Ensure within upper region* valid\_eyes.append((cx, cy, cw, ch))  
  
 *# Check if at least one eye is found  
 if* len(valid\_eyes) > 0:  
 authenticated += 1  
 *# Mark the authenticated area* cv2.rectangle(base\_picture, (loc\_x, loc\_y), (loc\_x + size\_w, loc\_y + size\_h), (0, 0, 255), 2)  
  
 *# Process each component  
 for* (cx, cy, cw, ch) *in* valid\_eyes:  
 component\_area = sub\_rgb[cy:cy + ch, cx:cx + cw]  
 transformed = cv2.GaussianBlur(component\_area, (23, 23), 30)  
 sub\_rgb[cy:cy + ch, cx:cx + cw] = transformed  
  
*return* authenticated, base\_picture

This section of the search\_components() function validates detected eye regions and applies anonymization. It first filters eye detections to ensure they lie within the upper portion of the detected face, preventing false positives. If at least one valid eye is found, the face is marked as authenticated, and a red bounding box is drawn around it for visualization. Each confirmed eye region is then blurred using a Gaussian filter to obscure identifiable features. Finally, the function returns the number of authenticated faces and the modified image, completing the anonymization step with both precision and visual confirmation.

Figure 4 - *Supervisory Function and Program Execution Flow for the Face-Anonymizer.*

A screenshot of a computer program

AI-generated content may be incorrect.

The below function spvz\_op()serves as the main coordinator for executing the complete face anonymization process on a single image. It begins by validating the file’s existence and then loads it safely using ld\_pic(). The image is optimized for feature extraction through grayscale conversion and contrast enhancement before being passed to the search\_components() function for face and eye detection. After successfully identifying and anonymizing facial regions, the modified image is saved to the output directory and displayed to the user. This function ensures consistent, reliability, and transparent execution of each processing stage.

*def* spvz\_op(image\_label):acc\_pt = os.path.join(SRC\_FOLDER, image\_label)  
  
 *if not* os.path.isfile(acc\_pt):  
 print("[SKIP] Missing entry: {}".format(acc\_pt))  
 *return* print("Beginning operation on: {}".format(image\_label))  
 access\_data = ld\_pic(acc\_pt)  
  
 tned\_gray = opt\_img(access\_data)  
  
 total\_auth, operation\_result = search\_components(tned\_gray, access\_data)  
  
 print(" Supervised {} authenticated components.".format(total\_auth))  
  
 access\_end = os.path.join(DEST\_FOLDER, "result\_" + image\_label) *# Changed to match report* cv2.imwrite(access\_end, operation\_result)  
 print(" Secured outcome in: {}".format(access\_end))  
  
 cv2.imshow("Operation Outcome - {}".format(image\_label), operation\_result)  
 cv2.waitKey(0)  
 cv2.destroyAllWindows()

This final program entry block below controls the execution flow of the Face-Anonymizer script. When the file is run directly, the \_\_name\_\_ == "\_\_main\_\_" condition ensures that the anonymization process is activated. It begins by printing a status message, then iterates through all images listed in TARGET\_IMAGES, calling the spvz\_op() function for each one. After processing all inputs, it prints a confirmation message directing the user to the output folder. This structure guarantees that the script executes only when intended, maintaining a clean and modular design suitable for reuse or integration.

*# Program Entry  
if* \_\_name\_\_ == "\_\_main\_\_":  
 print("Activating anonymization operations...\n")  
 *for* access\_file *in* TARGET\_IMAGES:  
 spvz\_op(access\_file)  
 print("\nOperations concluded. Inspect '{}' for outcomes.".format(DEST\_FOLDER))

Figure 5 - *Console Output Displaying Successful Execution of the Face-Anonymizer Program.*  
This output confirms that both Haar cascade classifiers, the haarcascade\_frontalface\_alt2.xml for face detection and and haarcascade\_eye\_tree\_eyeglasses.xml for eye detection were successfully loaded and executed. The program processed three images sequentially i.e. one non-human (zero detections), one group photo (eight authenticated faces), and one single-person image (one authenticated face). Each processed result was securely saved in the *detected-images* directory, demonstrating the program’s accuracy, reliability, and complete execution with an exit code of 0.

A screenshot of a computer program

AI-generated content may be incorrect.

For the output for result\_animal-1.jpg, there is no red bounding boxes or blurring are applied, which confirms that the Haar cascade classifier (haarcascade\_frontalface\_alt2.xml) correctly identified no human faces. This verifies that the algorithm’s detection logic is selective to human facial patterns and ignores non-human subjects.

Next, for result\_full-body-single-person-male-1.jpg, a red bounding box drawn around the detected face region, and the eyes are blurred for anonymization. The classifier detected a valid face, and the eye\_glasses\_detector confirmed the presence of eyes [haarcascade\_frontalface\_alt2.xml, haarcascade\_eye\_tree\_eyeglasses.xml.]

And, for the group image - result\_group-front-standing-group-1.jpg - Both haarcascade\_frontalface\_alt2.xml and haarcascade\_eye\_tree\_eyeglasses.xml were actively used. First, for detecting all faces in the group, and the second for detecting and blurring the eyes within each detected face.

The Face-Anonymizer program successfully demonstrated how OpenCV’s Haar cascade classifiers can be applied to automate privacy protection in digital images. Using haarcascade\_frontalface\_alt2.xml for detecting frontal human faces and haarcascade\_eye\_tree\_eyeglasses.xml for identifying eyes, the system accurately located and anonymized facial features across various scenarios. The results confirmed the algorithm’s ability to distinguish between human and non-human subjects, detect multiple faces within a group, and apply selective Gaussian blurring for effective anonymization. Overall, the implementation achieved reliable, repeatable performance and validated the practical utility of computer vision for ethical image processing.

References

Dev.to. (n.d.). *OpenCV: Detect and blur faces using Haar cascades in C#.* Retrieved from <https://dev.to/azure/opencv-detect-and-blur-faces-using-haar-cascades-in-c-1ao5>

GeeksforGeeks. (n.d.). *Detect an object with OpenCV Python.* Retrieved from <https://www.geeksforgeeks.org/python/detect-an-object-with-opencv-python/>

OpenCV Documentation. (n.d.). *Cascade classifier tutorial.* Retrieved from <https://docs.opencv.org/3.4/db/d28/tutorial_cascade_classifier.html>

Stack Overflow. (n.d.). *How to use OpenCV and Haar cascades to blur faces.* Retrieved from <https://stackoverflow.com/questions/18064914/how-to-use-opencv-and-haar-cascades-to-blur-faces>

TechSkillGuru. (n.d.). *Face detection with Haar cascades.* Retrieved from <https://techskillguru.com/opencv/face-detection-with-haar-cascades>

Toolify AI. (n.d.). *Blur faces in real-time with Python and OpenCV.* Retrieved from <https://www.toolify.ai/ai-news/blur-faces-in-realtime-with-python-and-opencv-1109872>

Toolify AI. (n.d.). *Realtime face blurring with Python and OpenCV.* Retrieved from <https://www.toolify.ai/ai-news/realtime-face-blurring-with-python-and-opencv-105658>