Milestone 3 of the Project

<u>Module 5: Web UI for Image Upload and Visualization</u>

Objective

The objective of Module 5 is to design and implement a responsive, user-friendly web interface that allows users to upload facial images, visualize detected faces with annotations, and view prediction results clearly without UI lag. User should also be able to download the annotated image and the predictions as a csv file.

Technologies Used

- **HTML5**: Page structure and layout
- **CSS3**: Styling, responsiveness, and UI enhancements
- **JavaScript (Vanilla JS)**: Client-side logic and asynchronous communication
- **Flask (Frontend Rendering)**: Serving HTML templates and static assets

Frontend Architecture Overview

The frontend consists of a single-page interface rendered by Flask, supported by JavaScript for asynchronous operations. The UI dynamically interacts with the backend inference pipeline through a REST API.

<u>Tasks Implemented</u>

1. Frontend Development

A minimal UI was designed using HTML and CSS to ensure clarity and ease of use. The interface is divided into clearly defined sections:

- Image upload section
- Uploaded image preview
- Annotated image visualization
- Analysis summary section
- Prediction table with CSV download option

The layout ensures a logical flow of interaction from image upload to result analysis.

2. Image Upload and Preview

- Implemented an image upload input that supports common image formats such as JPG, PNG and JPEG.
- The uploaded image is displayed instantly on the UI without a page reload.
- Handles both single-face and multi-face image uploads seamlessly.
- Utilizes asynchronous requests (Fetch API) to avoid UI blocking.

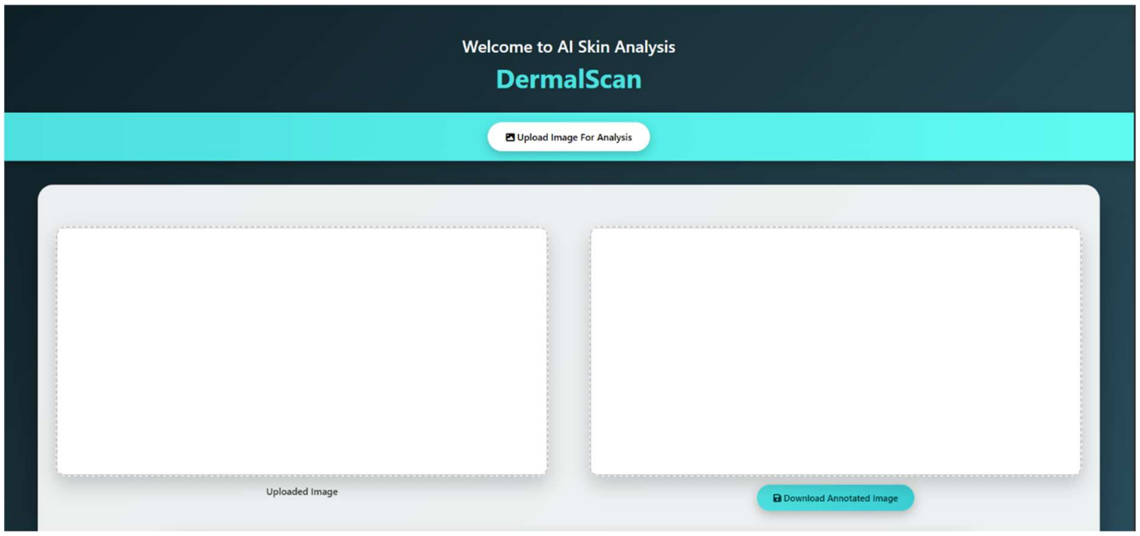
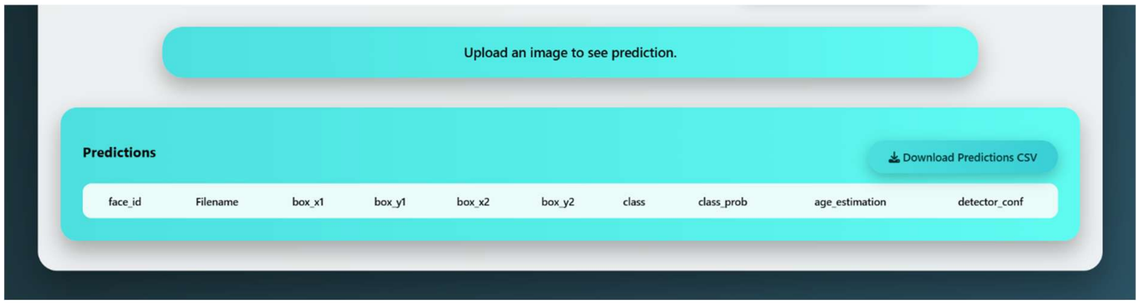


Figure 7: User-Interface of DermalScan AI Skin Analysis - 1



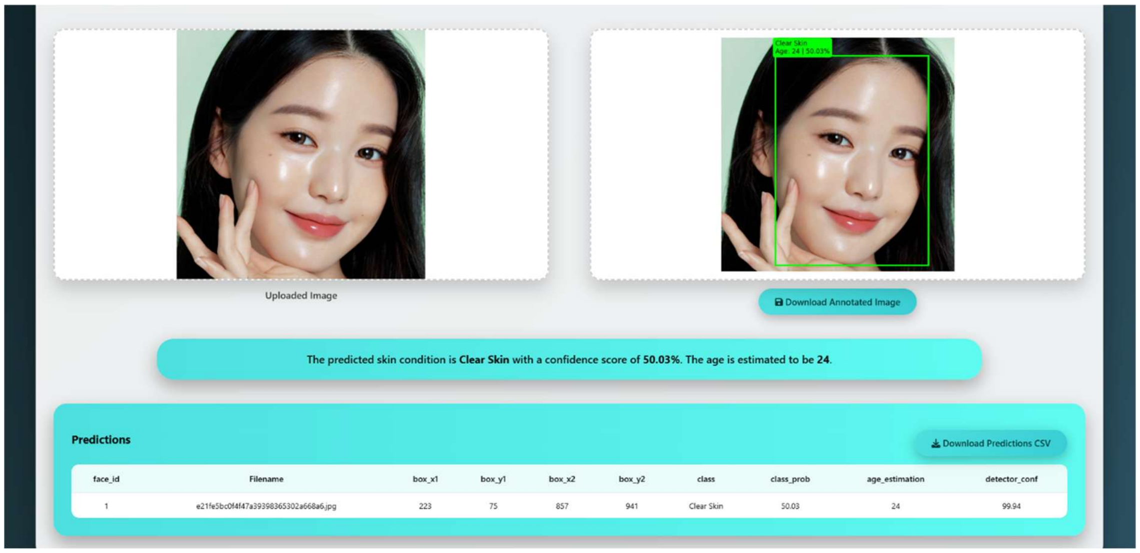
face_id	Filename	box_x1	box_y1	box_x2	box_y2	class	class_prob	age_estimation	detector_conf

Figure 8: User-Interface of DermalScan AI Skin Analysis - 2

3. Visualization and Annotation Display

- The annotated image returned from the backend is displayed dynamically.
- Bounding boxes are drawn around detected faces.
- Each face annotation includes:
    - Skin condition label (Predicted Class)
    - Estimated age
    - Confidence score (percentage)

This ensures visual clarity and easy interpretation of results.



The predicted skin condition is **Clear Skin** with a confidence score of **50.03%**. The age is estimated to be **24**.

Predictions

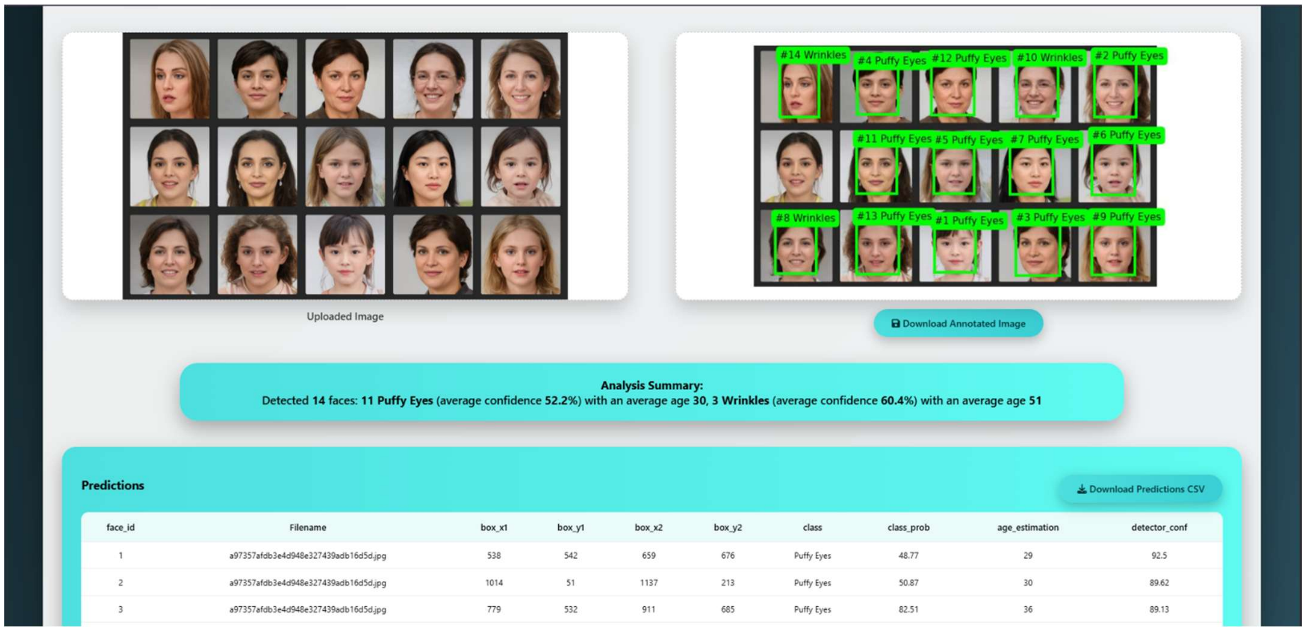
face_id	Filename	box_x1	box_y1	box_x2	box_y2	class	class_prob	age_estimation	detector_conf
1	e21fe5bc0f4f47a39398365302a668a6.jpg	223	75	857	941	Clear Skin	50.03	24	99.94

Figure 9: Visualization of annotated Image and Results

4. Multi-Face Handling in UI

The UI automatically adjusts based on the number of detected faces:

- Single face images
    - Full detailed annotation
    - Detailed prediction text with skin condition, confidence score and the estimated age.
- Multiple face images
    - Compact annotations using Face ID + class name
    - Reduces text overlap and visual congestion
    - Maintains clean and professional visualization



Analysis Summary:
Detected **14** faces: **11 Puffy Eyes** (average confidence **52.2%**) with an average age **30**, **3 Wrinkles** (average confidence **60.4%**) with an average age **51**

Predictions

face_id	Filename	box_x1	box_y1	box_x2	box_y2	class	class_prob	age_estimation	detector_conf
1	a97357afdb3e4d948e327439adb16d5d.jpg	538	542	659	676	Puffy Eyes	48.77	29	92.5
2	a97357afdb3e4d948e327439adb16d5d.jpg	1014	51	1137	213	Puffy Eyes	50.87	30	89.62
3	a97357afdb3e4d948e327439adb16d5d.jpg	779	532	911	685	Puffy Eyes	82.51	36	89.13

Figure 10: Multi-Face Image Output and Results

Multi-face Handling Logic for Result Text

```
# Summary Response
 total_faces = len(results)
 if total_faces == 1:
  r = results[0]
  response_text = (
  f"The predicted skin condition is <b>{r['class']}</b> "
  f"with a confidence score of <b>{r['class_prob']}%</b>. "
  f"The age is estimated to be <b>{r['age_estimation']}</b>."
  )
multi_face = False
 else:
  # For multiple faces
  summary = defaultdict(list)
  for r in results:
   summary[r["class"]].append((r["class_prob"], r["age_estimation"]))
   summary_parts = []
   for cls, values in summary.items():
    probs = [v[0] for v in values]
```

```
        ages = [v[1] for v in values]
        summary_parts.append(f"<b>{len(values)}</b> <b>{cls}</b> "
                    f"(average confidence <b>{sum(probs)/len(probs):.1f}%</b>) "
                    f"with an average age <b>{sum(ages)//len(ages)}</b>")
        response_text = (f"<b>Analysis Summary:</b><br>"
                  f"Detected <b>{total_faces}</b> faces: " +
                  ", ".join(summary_parts))
    multi_face = True
```

Module 6: Backend Pipeline for Model Inference

Objective

The objective of Module 6 is to build a robust backend pipeline capable of performing:

- Face detection
- Skin condition classification
- Age estimation
- Annotation rendering
- Seamless communication with the frontend

All results must be returned within an acceptable response time.

Technologies Used

- **Python:** Main language for backend logic
- **Flask:** Backend framework
- **TensorFlow / Keras**: Deep learning model inference
- **OpenCV (DNN Module)**: Face detection
- **Matplotlib:** Annotation rendering
- **NumPy:** Numerical operations
- **CSV & Flask Session Storage**: Prediction logging and export

Backend Architecture Overview

The backend pipeline follows a structured flow:

1. Input Image Handling
2. Face Detection using DNN
3. Preprocessing for Model Inference
4. Skin Condition Classification & Age Estimation
5. Post-processing & Annotation Rendering
6. Response Packaging for Frontend

Key Backend Components

1. DNN-Based Face Detection

A Deep Neural Network (DNN) face detector is a CNN-based model trained on large-scale datasets to detect faces accurately under varying conditions.

In this project we have used:

- OpenCV’s DNN module
- SSD (Single Shot Detector) architecture
- ResNet-10 backbone

The detector predicts the:

- Face presence
- Bounding box coordinates
- Detection confidence score

```
# DNN Face Detector

PROTO = "deploy.prototxt"

WEIGHTS = "res10_300x300_ssd_iter_140000.caffemodel"

face_net = cv2.dnn.readNetFromCaffe(PROTO, WEIGHTS)
```

Why we have used DNN over traditional Haar Cascade for face detection?

Traditional Haar Cascade	DNN Face Detector
Sensitive to lighting	**Robust to lighting**
Misses faces easily	**High recall**
Poor multi-face support	**Excellent multi-face support**
Outdated	**Industry standard**

Table 4: Comparison between Haar Cascade and DNN face detector

2. Blob Creation and DNN Inference

The input image is converted into a blob, which:

- Normalizes pixel values
- Resizes the image
- Converts it into a tensor suitable for DNN inference

This ensures consistent input and reliable detection performance.

```
# Face Detection
  blob = cv2.dnn.blobFromImage(
    cv2.resize(image, (300, 300)),
    1.0,
    (300, 300),
    (104.0, 177.0, 123.0)
  )
  face_net.setInput(blob)
  detections = face_net.forward()
```

3. Multi-Face Detection with Non-Maximum Suppression (NMS)

Why **Non-Maximum Suppression** is Required?

DNN detectors often output multiple overlapping bounding boxes for the same face. Without filtering, this causes duplicate detections.

Non-Maximum Suppression (NMS):

NMS removes overlapping boxes by:

- Retaining the bounding box with the highest confidence
- Suppressing overlapping boxes beyond a defined threshold

This allows:

- Accurate multi-face detection
- Elimination of redundant detections
- Clean visualization and correct face count

```
# Non-Maximum Suppression (NMS)

indices = cv2.dnn.NMSBoxes(boxes, confidences, 0.5, 0.4)
```

4. Face Preprocessing and Model Inference

Each detected face undergoes:

- Cropping using bounding box
- Resizing to 224×224
- Pixel normalization
- Expansion into batch format

The processed face is passed into a **trained MobileNet-based CNN model** that predicts skin conditions.

6. Annotation Rendering

Matplotlib is used to:

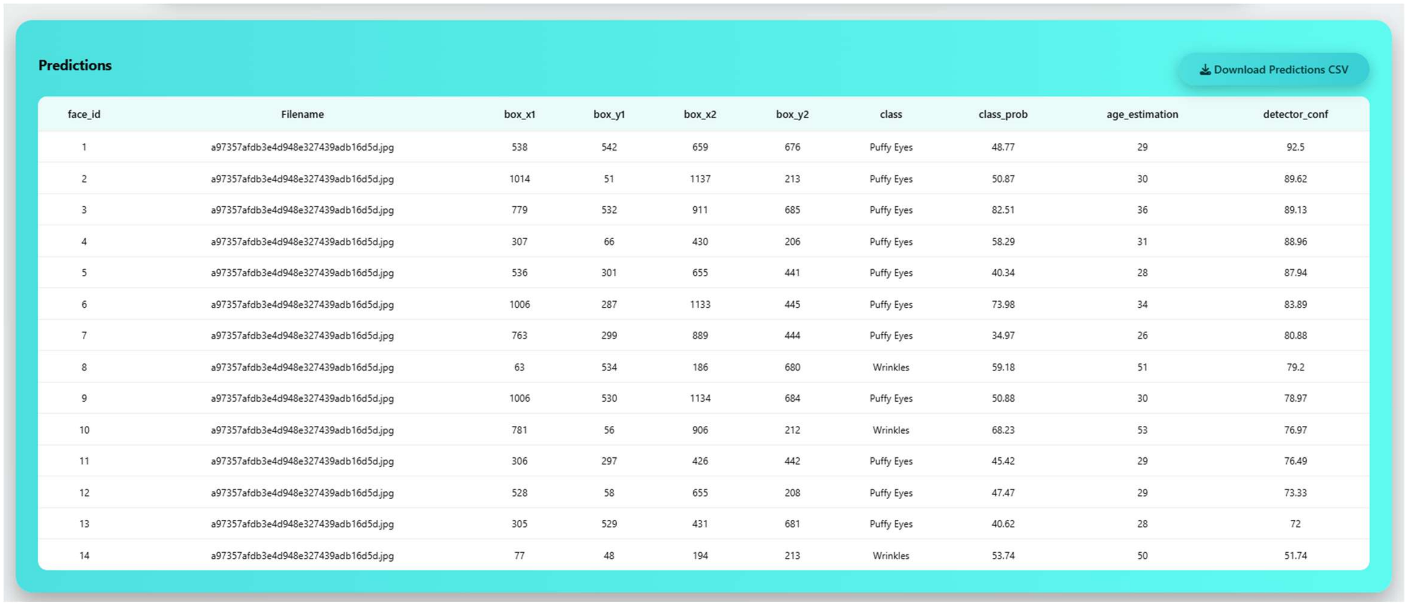
- Draw bounding boxes
- Overlay class labels, confidence, and age
- Adjust annotation detail dynamically for congested images

The annotated image is saved and returned to the frontend.

7. Session-Based Prediction Logging

- Each detected face is assigned a Face ID
- Face IDs persist throughout the session
- Predictions are logged with:
    - Face ID
    - Bounding box coordinates
    - Predicted class
    - Confidence score
    - Estimated age

This data is displayed in a table and exportable as a CSV file.



Predictions

face_id	Filename	box_x1	box_y1	box_x2	box_y2	class	class_prob	age_estimation	detector_conf
1	a97357afdb3e4d948e327439adb16d5d.jpg	538	542	659	676	Puffy Eyes	48.77	29	92.5
2	a97357afdb3e4d948e327439adb16d5d.jpg	1014	51	1137	213	Puffy Eyes	50.87	30	89.62
3	a97357afdb3e4d948e327439adb16d5d.jpg	779	532	911	685	Puffy Eyes	82.51	36	89.13
4	a97357afdb3e4d948e327439adb16d5d.jpg	307	66	430	206	Puffy Eyes	58.29	31	88.96
5	a97357afdb3e4d948e327439adb16d5d.jpg	536	301	655	441	Puffy Eyes	40.34	28	87.94
6	a97357afdb3e4d948e327439adb16d5d.jpg	1006	287	1133	445	Puffy Eyes	73.98	34	83.89
7	a97357afdb3e4d948e327439adb16d5d.jpg	763	299	889	444	Puffy Eyes	34.97	26	80.88
8	a97357afdb3e4d948e327439adb16d5d.jpg	63	534	186	680	Wrinkles	59.18	51	79.2
9	a97357afdb3e4d948e327439adb16d5d.jpg	1006	530	1134	684	Puffy Eyes	50.88	30	78.97
10	a97357afdb3e4d948e327439adb16d5d.jpg	781	56	906	212	Wrinkles	68.23	53	76.97
11	a97357afdb3e4d948e327439adb16d5d.jpg	306	297	426	442	Puffy Eyes	45.42	29	76.49
12	a97357afdb3e4d948e327439adb16d5d.jpg	528	58	655	208	Puffy Eyes	47.47	29	73.33
13	a97357afdb3e4d948e327439adb16d5d.jpg	305	529	431	681	Puffy Eyes	40.62	28	72
14	a97357afdb3e4d948e327439adb16d5d.jpg	77	48	194	213	Wrinkles	53.74	50	51.74

Figure 11: Predictions Table for One Session

8. Summary Generation Logic

- Single-face images display detailed individual results.
- Multi-face images generate an aggregated summary:
    - Face count per class
    - Average confidence
    - Average age per condition

This avoids UI clutter while providing meaningful insights.

Frontend–Backend API Integration

API Type Used: Custom REST API built using Flask

The backend returns structured JSON containing:

- Annotated image URL
- Per-face prediction data
- Dynamic summary text
- CSV-ready table data

The frontend consumes this data asynchronously for seamless updates.