Project Title - DermalScan: AI_Facial Skin Ageing Detection App

Infosys SpringBoard Virtual Internship Program

Submitted By:

Brijesh Rath to Mr. Praveen sir

INTRODUCTION

Problem Statement:

Facial aging indicators such as wrinkles, dark spots, puffy eyes, and overall skin clarity are important in understanding skin health, yet these features are difficult to assess consistently through manual observation. Variations in lighting, skin tone, facial expressions, and image quality make the visual evaluation of aging signs unreliable and time-consuming. Currently, there is no automated system that can accurately identify and categorize multiple facial aging signs from a single image in a standardized way. This creates a gap for applications in skincare analysis, cosmetic recommendations, and dermatology support, where consistent and objective assessment of facial aging is essential.

Objectives:

- Detecting and locating facial characteristics which indicate the presence of age.
- Categorise the characteristics into 4 specific categories (i.e. Wrinkled, Dark Spot, Puffy Eyes and Clear Skin) using a Convolutional Neural Network (CNN).
- Create a user-friendly web based front end by allowing users to upload images with the expected annotated outcomes and percentage predictions.
- Connect the pipeline with a backend to process uploaded images, make inferences and return annotated results.

System Requirements:

- **Python (v3.10.1):** The primary language used for all data processing and model building.
- **Pandas(2.3.3):** Used for creating DataFrames, managing the dataset, and handling CSV files (import pandas as pd).
- **Matplotlib(3.10.7):** It is the foundational plotting and visualization library for Python.
- **Seaborn(0.13.2):** It simplifies complex statistical plots (like heatmaps or count plots) into single lines of code and makes them look better automatically.
- **Scikit-learn(1.7.2):** Used for splitting the data into training and testing sets (train_test_split).
- **Numpy(2.2.6):** It is a fundamental dependency for Pandas, Scikit-Learn, and TensorFlow, and is required for numerical operations.
- **Jupyter Notebook:** Used for interactive coding and ensuring the code runs step by step

Milestone 1 of the Project

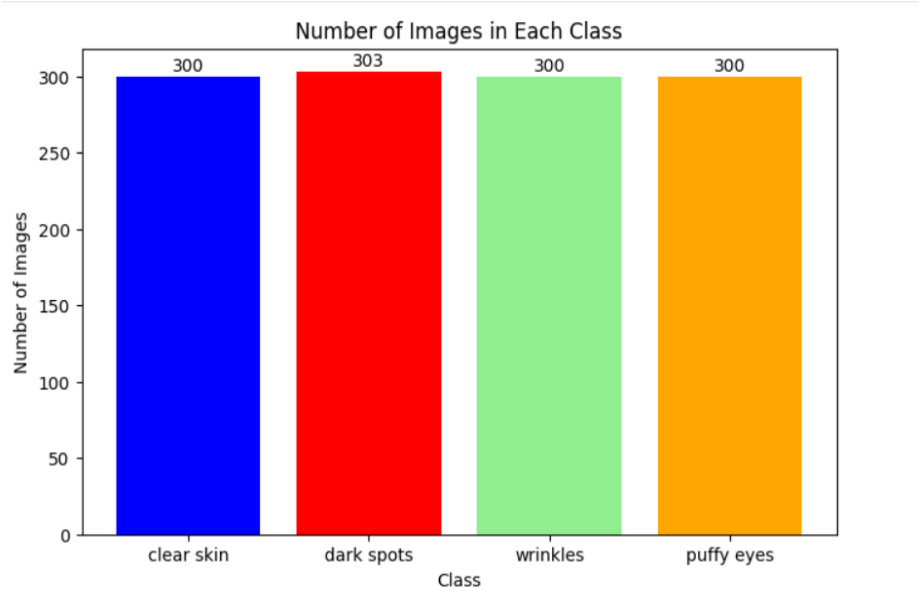
➢ Module 1: Dataset Setup and Image Labeling

- Aggregated facial images into four distinct classes: Wrinkles, Dark Spots, Puffy Eyes, and Clear Skin.
- Manually filtered out poor-quality samples and ensured equal representation across all classes to prevent model bias.
- Produced a finalized, labeled dataset directory and a class distribution plot confirming data balance.

➢ Output:

	Class	Count
0	clear skin	300
1	dark spots	303
2	wrinkles	300
3	puffy eyes	300

[Fig 1. Dataset Class Distribution Summary]



[Fig 2. Distribution of Images Across Skin Condition Categories]

➢ Module 2: Image Preprocessing and Augmentation

- In this module, all images in the dataset are prepared for ingestion by the deep learning model. Since neural networks operate on standardized input dimensions, each image is resized to 224×224 pixels. Following resizing, images are normalized to scale pixel values to a consistent numerical range, typically between 0 and 1. This normalization stabilizes gradient updates and enhances the training efficiency of the model.

```
target_size= (224, 224)
img = img.astype("float32") / 255.0  #normalize
```

- To improve the dataset's robustness and prevent overfitting, image augmentation techniques are applied. Augmentation introduces controlled variations such as horizontal flipping, rotation, and zoom transformations. These variations simulate real-world changes in facial orientation and lighting, enabling the model to learn more generalized features. Each augmentation preserves essential facial characteristics, ensuring that class-specific features like wrinkles or dark spots remain identifiable

- The dataset is split into training and validation sets to ensure the model learns from one portion of the data while being evaluated on unseen samples. This helps monitor overfitting and improves the model's ability to generalize. 75% of data is for training, 15% of data is for testing and 15% of data is for validation.

- In addition to preprocessing, the categorical labels of the dataset are converted into one-hot encoded vectors. This encoding transforms each class label into a binary vector representation suitable for multiclass classification, enabling the model to process labels mathematically during training.

- **Creation of a generator object using the method RandomFlip,RandomZoom and etc. from the tensorflow. keras.Sequential module,**

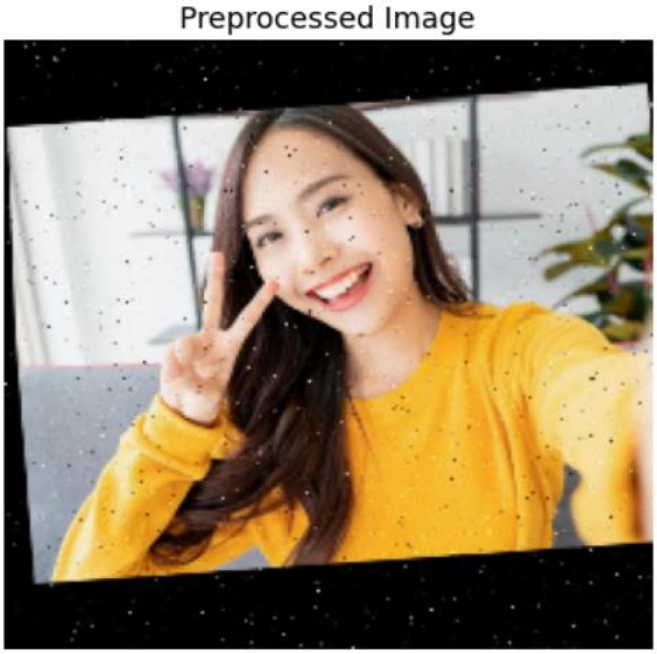
```
data_augmentation = tf.keras.Sequential([
    tf.keras.layers.RandomFlip("horizontal"),
    tf.keras.layers.RandomRotation(0.1),
    tf.keras.layers.RandomZoom(0.1),
    tf.keras.layers.RandomContrast(0.2)
])
```

- **Performing the data augmentation on the actual dataset and applying one-hot encoding on the dataset,**

```
def load_dataset(path, img_size=224, batch_size=32):
    dataset =
tf.keras.preprocessing.image_dataset_from_directory(
        path,
        image_size=(img_size, img_size),
        label_mode="categorical",  # one-hot encoding
        shuffle=True
    )
```

- This module results in a fully pre-processed, augmented, and model-ready dataset. It ensures that images maintain consistent dimensions, classes are represented with increased diversity, and the dataset becomes more resilient to variations, ultimately enhancing the performance and generalization of the final deep learning model.

➢ Preprocessed Image Preview



[Fig 3. Preprocessed Image]

➢ Augmented Image Preview

[Fig 4. Augmented Image]

- Additionally, a preview grid of 3 augmented images with the original image.

Milestone 2 of the Project

Overview of Milestone 2:

- Milestone 2 focuses on developing, training, evaluating, and deploying a deep learning–based image classification model using transfer learning. This milestone is divided into two main modules:

- Module 3: Model Training and Evaluation
- Module 4: Face Detection and Prediction Pipeline

- The objective is to train a robust CNN model using a pretrained architecture, validate its performance using appropriate metrics, and finally deploy the trained model in a real-time face detection and prediction system.

Module 3: Model Training and Evaluation

Objective:

- The goal of Module 3 is to build and train a deep learning image classification model using transfer learning, evaluate its performance on training, validation, and test datasets, and analyze model behavior using accuracy and loss curves.

Dataset Description:

- The dataset is organized into three separate directories:

- Training dataset
- Validation dataset
- Testing dataset

- Each directory contains subfolders representing different classes. The images are resized to a fixed resolution of 224 × 224 pixels, matching the input requirements of the pretrained network.

Data Preprocessing and Augmentation:

- To improve generalization and prevent overfitting, image augmentation techniques were applied to the training dataset:

  - Rescaling pixel values to the range [0, 1]
  - Random rotations
  - Width and height shifts
  - Zoom augmentation
  - Horizontal flipping
  - Brightness adjustment

- Validation and test datasets were rescaled only, without augmentation, to ensure fair evaluation.

Model Architecture:

- A MobileNetV2 pretrained model was used as the backbone for transfer learning.
- Architecture Details
  - Base Model: MobileNetV2 (ImageNet weights)
  - Top layers removed: Fully connected layers excluded
  - Custom classification head added:
    - Global Average Pooling layer
    - Fully connected Dense layer with ReLU activation
    - Dropout layers for regularization
    - Final Dense layer with Softmax activation for multi-class classification

Initial Training Phase:

- During the first training phase:
  - All layers of the pretrained base model were **frozen**
  - Only the custom classification head was trained
  - Optimizer used: **Adam**
  - Loss function: **Categorical Crossentropy**
  - Metric monitored: **Accuracy**

- Training Configuration:
  - Epochs: 20
  - Batch size: 32

```
history = model.fit(    #first TRAINING
    train_data,
    validation_data=val_data,
    epochs=20,
    verbose=1
)
```

[Fig 5. Training & Validation Accuracy – Initial Training]

Parameter-Tuning Phase:

- To further improve performance, fine-tuning was performed:
  - The last 30 layers of the base model were unfrozen
  - A lower learning rate was used to prevent large weight updates
  - This allowed the model to learn task-specific features

- Parameter-Tuning Configuration
  - Optimizer: Adam
  - Learning rate: 1e-5
  - Epochs: 20

```
model.compile(
    optimizer=Adam(learning_rate=1e-5),
    loss="categorical_crossentropy",
    metrics=["accuracy"]
)

history_fine = model.fit(
    train_data,
    validation_data=val_data,
    epochs=20,
    verbose=2
)
```

```
27/27 - 7s - 271ms/step - accuracy: 0.8998 - loss: 0.2638 - val_accuracy: 0.8729 - val_loss: 0.2993
Epoch 12/20
27/27 - 7s - 272ms/step - accuracy: 0.9057 - loss: 0.2676 - val_accuracy: 0.8729 - val_loss: 0.2923
Epoch 13/20
27/27 - 7s - 275ms/step - accuracy: 0.8974 - loss: 0.2591 - val_accuracy: 0.8840 - val_loss: 0.2890
Epoch 14/20
27/27 - 7s - 274ms/step - accuracy: 0.9010 - loss: 0.2724 - val_accuracy: 0.8840 - val_loss: 0.2893
Epoch 15/20
27/27 - 7s - 276ms/step - accuracy: 0.9117 - loss: 0.2491 - val_accuracy: 0.8785 - val_loss: 0.2864
Epoch 16/20
27/27 - 7s - 277ms/step - accuracy: 0.8675 - loss: 0.3175 - val_accuracy: 0.8785 - val_loss: 0.2828
Epoch 17/20
27/27 - 8s - 281ms/step - accuracy: 0.9117 - loss: 0.2587 - val_accuracy: 0.8785 - val_loss: 0.2859
Epoch 18/20
27/27 - 7s - 277ms/step - accuracy: 0.9093 - loss: 0.2628 - val_accuracy: 0.8895 - val_loss: 0.2851
Epoch 19/20
27/27 - 8s - 280ms/step - accuracy: 0.9141 - loss: 0.2391 - val_accuracy: 0.8840 - val_loss: 0.2845
Epoch 20/20
27/27 - 7s - 273ms/step - accuracy: 0.9189 - loss: 0.2230 - val_accuracy: 0.8840 - val_loss: 0.2847
```

```
[37]: test_loss, test_acc = model.evaluate(test_data)    #test accuracy
      print("TEST ACCURACY =", test_acc)
      print("TEST LOSS =", test_loss)

      6/6 ━━━━━━━━━━━━━━━━━━━━ 1s 149ms/step - accuracy: 0.8533 - loss: 0.3652
      TEST ACCURACY = 0.85326087474823
      TEST LOSS = 0.3652069568634033
```

```
[38]: print("PARAMETER-TUNE TRAIN ACCURACY =", history_fine.history["accuracy"][-1])
      print("PARAMETER-TUNE VALIDATION ACCURACY   =", history_fine.history["val_accuracy"][-1])

      PARAMETER-TUNE TRAIN ACCURACY = 0.9188544154167175
      PARAMETER-TUNE VALIDATION ACCURACY   = 0.8839778900146484
```

[Fig 6. Training & Validation Accuracy – Parameter-Tuning Training]

Model Evaluations:

- After training, the model was evaluated on the test dataset, which was not used during training or validation.

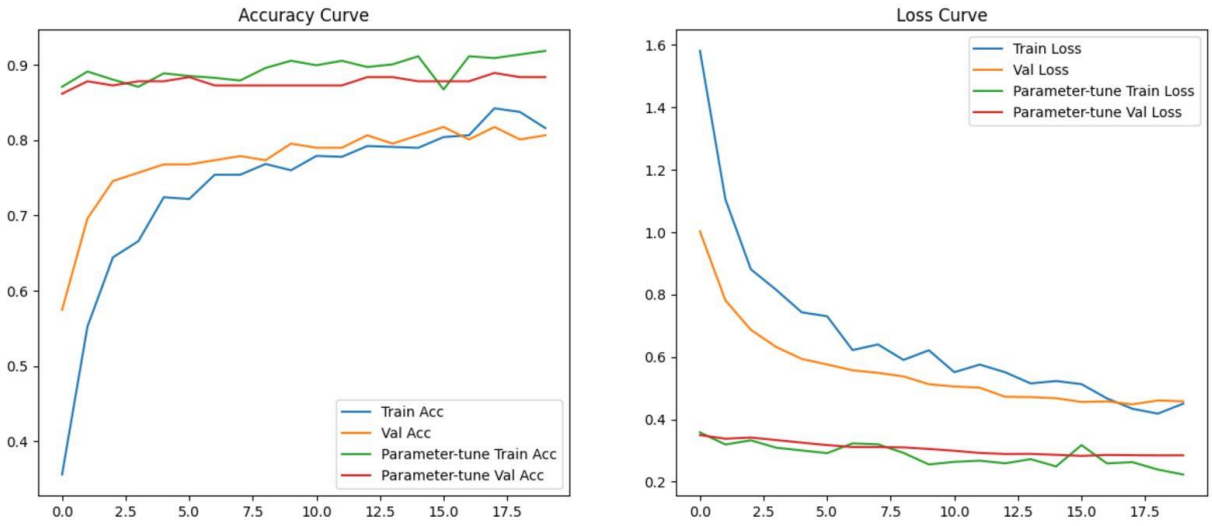
- Evaluation Metrics
  - Training Accuracy
  - Validation Accuracy
  - Testing Accuracy
  - Loss values

- Observed Performance
  - Training accuracy showed consistent improvement
  - Validation accuracy remained stable
  - Testing accuracy confirmed good generalization


```
27/27 - 7s - 271ms/step - accuracy: 0.8998 - loss: 0.2638 - val_accuracy: 0.8729 - val_loss: 0.2993
Epoch 12/20
27/27 - 7s - 272ms/step - accuracy: 0.9057 - loss: 0.2676 - val_accuracy: 0.8729 - val_loss: 0.2923
Epoch 13/20
27/27 - 7s - 275ms/step - accuracy: 0.8974 - loss: 0.2591 - val_accuracy: 0.8840 - val_loss: 0.2890
Epoch 14/20
27/27 - 7s - 274ms/step - accuracy: 0.9010 - loss: 0.2724 - val_accuracy: 0.8840 - val_loss: 0.2893
Epoch 15/20
27/27 - 7s - 276ms/step - accuracy: 0.9117 - loss: 0.2491 - val_accuracy: 0.8785 - val_loss: 0.2864
Epoch 16/20
27/27 - 7s - 277ms/step - accuracy: 0.8675 - loss: 0.3175 - val_accuracy: 0.8785 - val_loss: 0.2828
Epoch 17/20
27/27 - 8s - 281ms/step - accuracy: 0.9117 - loss: 0.2587 - val_accuracy: 0.8785 - val_loss: 0.2859
Epoch 18/20
27/27 - 7s - 277ms/step - accuracy: 0.9093 - loss: 0.2628 - val_accuracy: 0.8895 - val_loss: 0.2851
Epoch 19/20
27/27 - 8s - 280ms/step - accuracy: 0.9141 - loss: 0.2391 - val_accuracy: 0.8840 - val_loss: 0.2845
Epoch 20/20
27/27 - 7s - 273ms/step - accuracy: 0.9189 - loss: 0.2230 - val_accuracy: 0.8840 - val_loss: 0.2847
```

```
[37]: test_loss, test_acc = model.evaluate(test_data)    #test accuracy
      print("TEST ACCURACY =", test_acc)
      print("TEST LOSS =", test_loss)

      6/6 ━━━━━━━━━━━━━━━━━━━━ 1s 149ms/step - accuracy: 0.8533 - loss: 0.3652
      TEST ACCURACY = 0.85326087474823
      TEST LOSS = 0.3652069568634033
```


[Fig 7. Console output showing Test Accuracy and Test Loss]



[Fig 8. Line-Graph showing Accuracy and Loss Curve]

Model Saving:

- The trained model was saved in (.h5) format for later use in deployment.
- **Saved Files:**
  - `skin_classifier_mobilenetv2.h5`
  - `class_indices.json`

- Module 3 successfully achieved its objective of training a reliable image classification model using transfer learning. The model demonstrated stable training behaviour, good validation performance, and acceptable test accuracy, making it suitable for deployment in a real-time prediction pipeline.

Module 4: Face Detection and Prediction Pipeline

Objective:

- The goal of Module 4 is to integrate the trained model into a real-time prediction pipeline that:
  - Detects faces using OpenCV
  - Crops detected face regions
  - Applies the trained model to predict the class
  - Displays predictions as percentages in real time

Face Detection Method:

- Face detection was implemented using:
  - **OpenCV**
  - **Haar Cascade Classifier**

- The Haar Cascade model detects faces in grayscale images using pre-trained features.

```
face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades +
"haarcascade_frontalface_default.xml")
```

Image-Based Predictions Pipeline Workflow:

- The complete prediction pipeline follows these steps:

1. User uploads a facial image through the application interface
2. Load the uploaded image into the system
3. Convert the image to grayscale for face detection
4. Detect face regions using Haar Cascade classifier
5. Draw bounding box around detected face region
6. Crop the detected face area from the image
7. Resize the cropped face to the CNN model input size (224×224)
8. Normalize pixel values for model compatibility
9. Pass the processed image to the trained CNN model
10. Obtain predicted class probabilities
11. Display the predicted class label along with confidence percentage

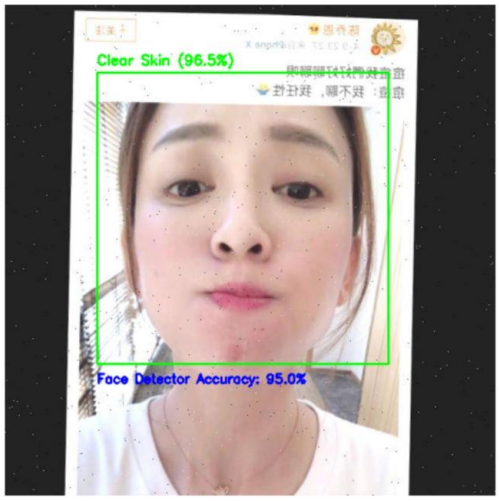
Image-Based Face Detection and Skin Condition Prediction:

- The system was tested using static facial images as input:
  - A facial image is imported directly from the local system
  - Haar Cascade is used to detect the face region in the image
  - Bounding boxes are drawn around detected face regions
  - The detected face is passed to the trained CNN model
  - Predicted skin condition class label is displayed
  - Confidence score is shown as a percentage for the predicted class

```
def run_inference(img_path, model, class_names):
    img = safe_read_image(img_path)

    faces = detect_faces(img)

    if len(faces) == 0:
        print("No face detected")
        return img

    for (x, y, w, h) in faces:
        face_roi = img[y:y+h, x:x+w]

        if face_roi.size == 0:
            continue

        class_name, confidence = predict_skin_issue(
            face_roi, model, class_names
        )

        # Draw bounding box
        cv2.rectangle(img, (x, y), (x+w, y+h), (0, 255, 0), 2)

        label = f"{class_name} ({confidence:.1f}%)"
        cv2.putText(
            img, label, (x, y-10),
            cv2.FONT_HERSHEY_SIMPLEX,
            0.6, (0, 255, 0), 2
        )

    return img
```

```
img_path = "/Users/brijesh/Desktop/AI_DermalScan/data/raw/clear
skin/clear_skin_010.jpg"

output_img = run_inference(img_path, model, class_names)
show_image(output_img)
```

```
131]: img_path = "/Users/brijesh/Desktop/AI_DermalScan/data/raw/clear skin/clear_skin_010.jpg"

output_img = run_inference(img_path, model, class_names)
show_image(output_img)
```



[Fig 9. Image-based face detection output showing detected face with bounding box, predicted class label, and confidence percentage]

Evaluation Criteria:

- **Face Detection Accuracy**
  - Cascade successfully detected frontal faces
  - Bounding boxes correctly enclosed face regions
- Haar **Prediction Accuracy**
  - Model predictions matched expected class labels
  - Confidence values reflected prediction certainty

- Module 4 successfully completed the deployment of the trained model into a real-time face detection and prediction pipeline. The system accurately detects faces and displays class predictions with confidence percentages, fulfilling all module requirements.

Milestone 3 of the Project

Overview of Milestone 3:

- Milestone 3 focuses on integrating the trained deep learning model with a user-friendly web-based frontend and a modular backend inference pipeline using Flask. The objective of this milestone is to transform the previously developed and evaluated model into an interactive web application that allows users to upload facial images, visualize predictions, bounding boxes, and obtain classification results in real time.

- This milestone bridges the gap between model development and real-world deployment by combining computer vision, deep learning inference, and web application development. The system is designed to ensure seamless input-to-output flow, minimal inference latency, accurate face localization, and clear visualization of prediction results.

Objectives of Milestone 3:

User Uploads Image (UI)
↓
Frontend (Flask + HTML)
↓
Backend Preprocessing
↓
Face Region Extraction
↓
Face Cropping & Resizing
↓
MobileNetV2 Model Inference
↓
Predictions + Probabilities
↓
Bounding Box Visualization & Results Displayed on UI

Module 5: Web UI for Image Upload and Visualization

Purpose of Module 5:

- The purpose of Module 5 is to provide an interactive and visually appealing graphical user interface that enables users to upload facial images and visualize the model's predictions in a clean and understandable format.

- This module ensures that even non-technical users can interact with the AI system without needing command-line execution or backend knowledge. The UI clearly presents original images, annotated images, predicted classes, confidence scores, age estimation, and bounding box coordinates.

Tech Stack:

- **Flask** – backend web framework
- **HTML5** – structure of the web interface
- **Python** – backend logic
- **OpenCV** – image processing and bounding box drawing
- **PIL (Pillow)** – image handling
- **NumPy** – numerical operations

Functional Workflow of Web UI:

- User opens the web application in a browser
- User uploads a facial image using the file upload button
- The uploaded image is displayed on the UI as the original image
- The image is sent to the backend inference pipeline via Flask
- Backend processes the image and extracts the face region
- Model prediction results are generated
- Bounding boxes and labels are drawn on the image
- Annotated image is displayed on the UI
- Confidence percentages and age estimation are shown
- Users can download annotated images and CSV results

User Interface Components:

- **The Flask-based UI consists of the following elements:**
  - Title and Description
  - File Upload Button
  - Analyze Skin Button
  - Original Image Preview Section
  - Annotated Image Preview Section
  - Prediction Output Section
  - Bounding Box Coordinates Display (x1, y1, x2, y2)
  - Confidence Percentage Display
  - Age Estimation Display
  - Download Annotated Image Button
  - Download CSV Button

Flask UI Script (app.py):

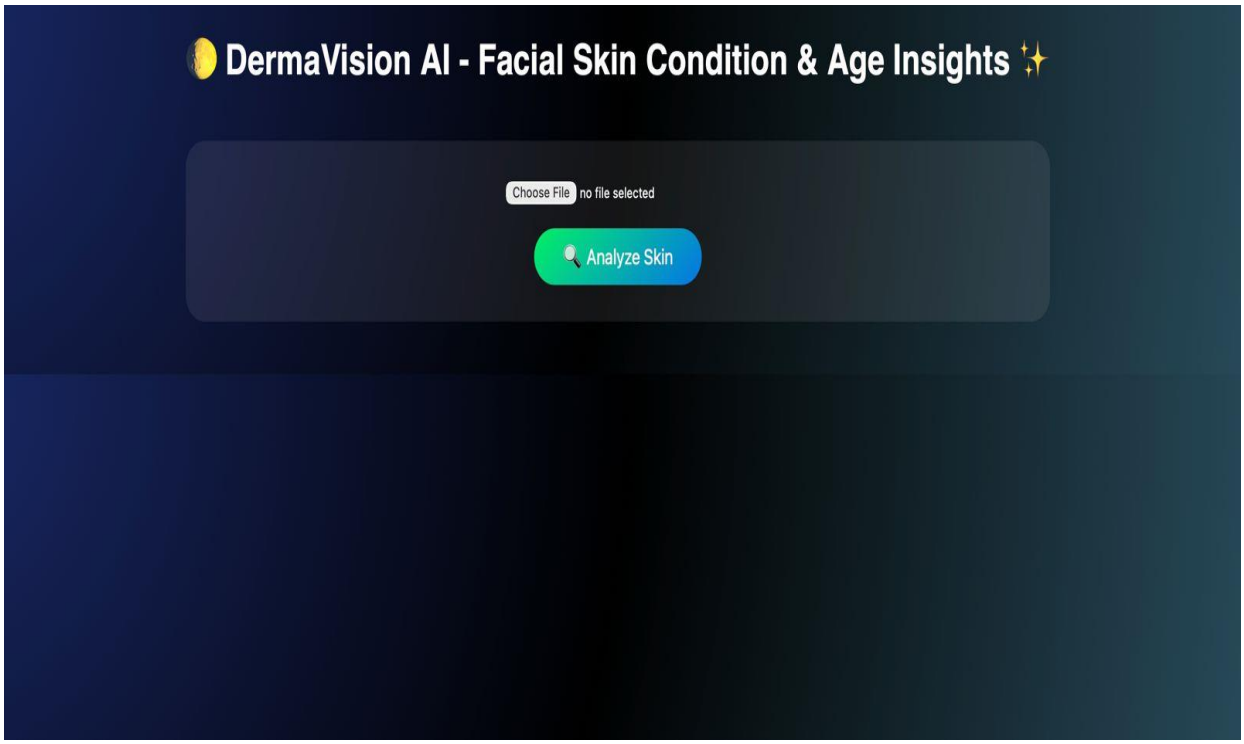
- The UI logic is implemented using Flask as a standalone Python backend combined with an HTML-based frontend. The Flask application manages user input, image visualization, and communication with the backend inference module.

- Key UI responsibilities:

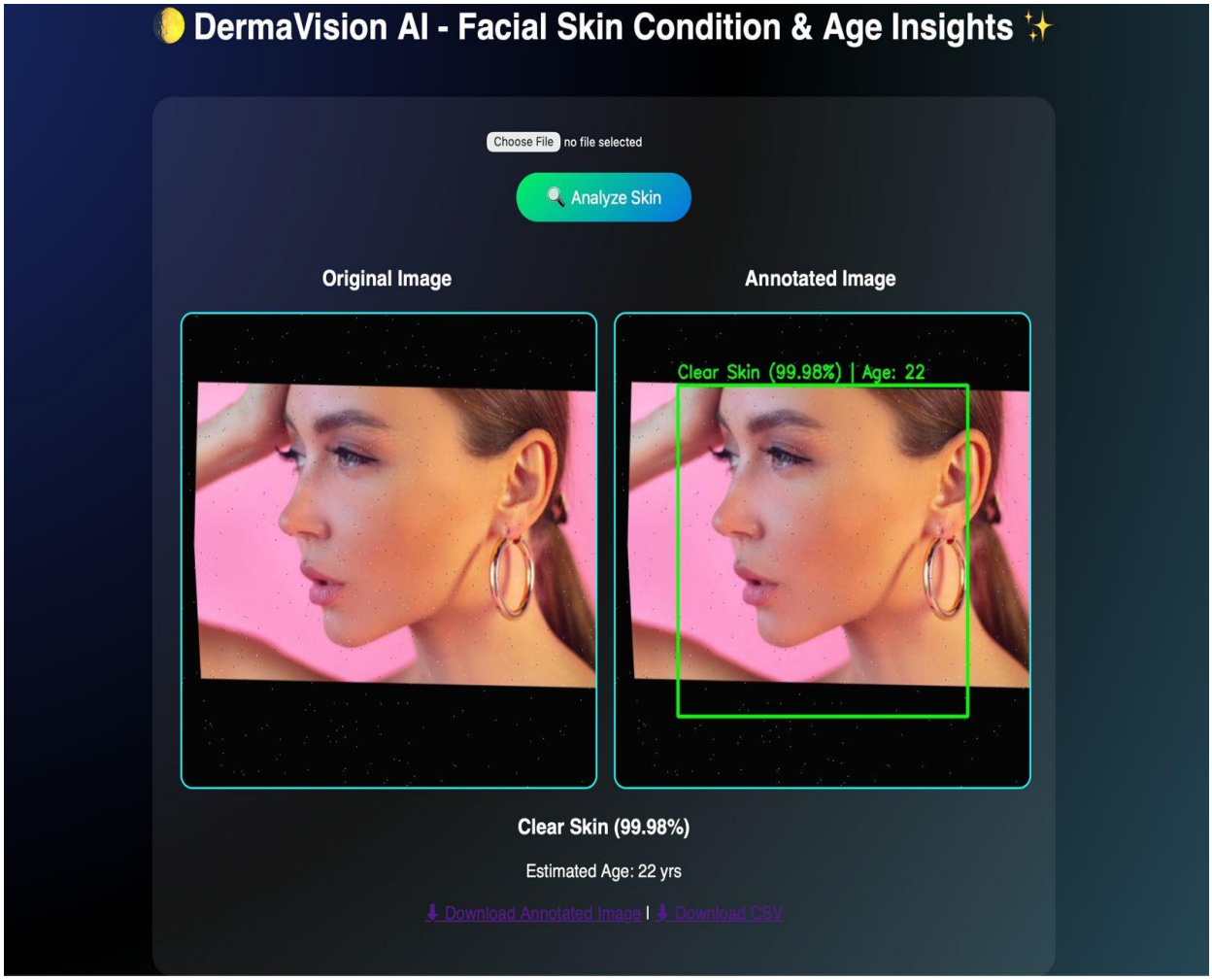
  - Accept image uploads from users
  - Display original uploaded image
  - Trigger backend inference pipeline
  - Display annotated image with bounding boxes
  - Show predicted class labels and confidence scores
  - Handle errors and invalid inputs gracefully

Output Visualization:

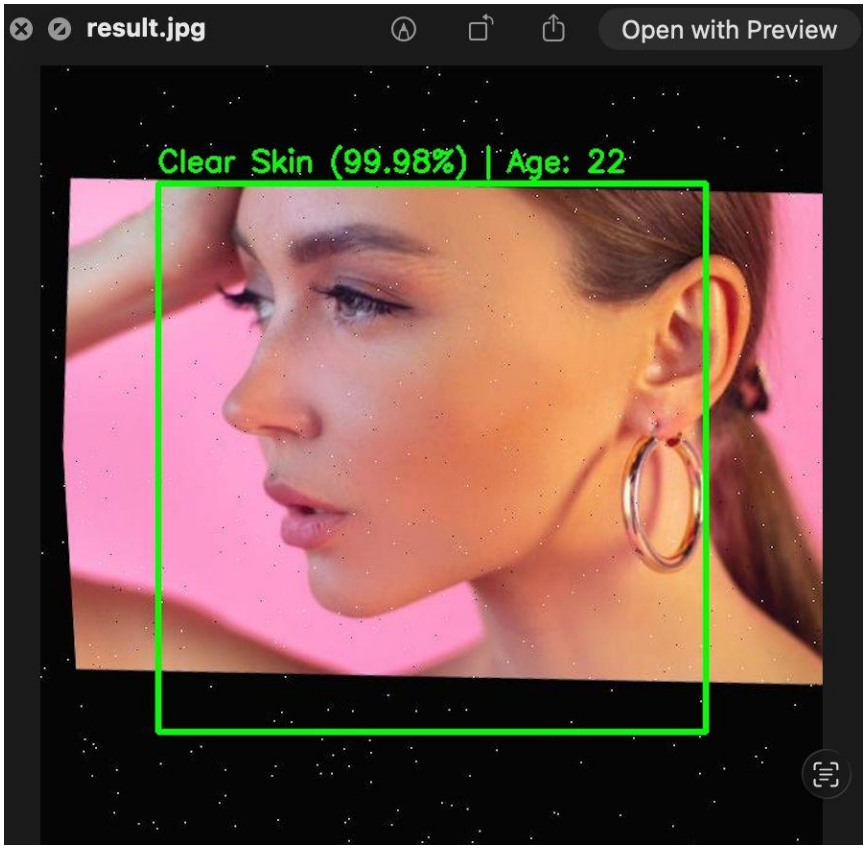
- Predictions are visualized directly on the uploaded image using bounding boxes. Each bounding box includes:
  - Predicted class label
  - Confidence score (percentage)
  - Bounding box coordinates (x1, y1, x2, y2)
- This visualization enhances interpretability and user trust in the system.



[Fig 10. Web UI]



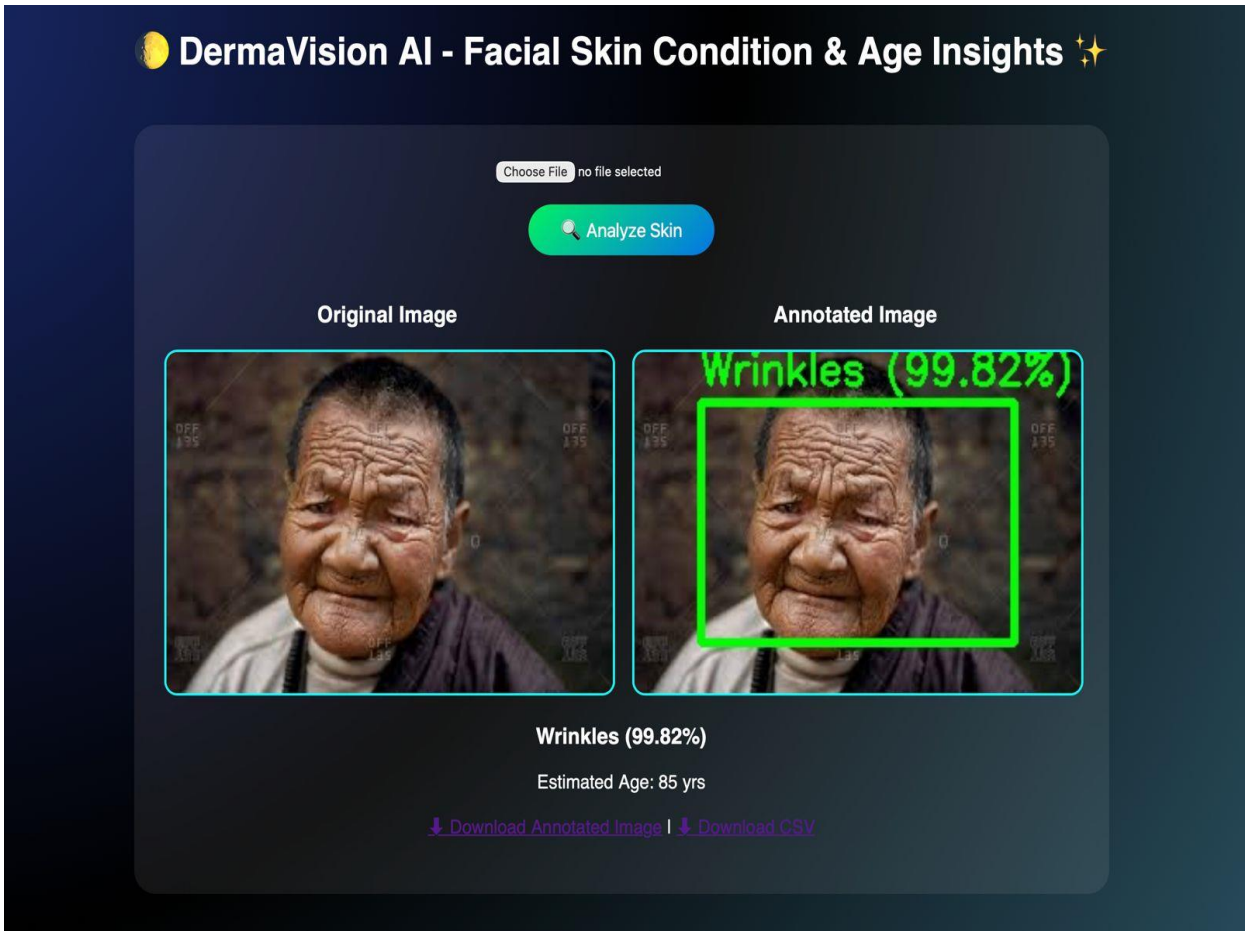
[Fig 11. Web UI with image upload and prediction visualization]



[Fig 12. Annotated image(downloaded)]

filename	box_x1	box_y1	box_x2	box_y2	class	confidence	age	age_bucket
clear_skin_140.jpg	96	96	544	544	Clear Skin	0.9998	22	22-28

[Fig 13. Prediction CSV (downloaded)]



[Fig 14. Web UI with image upload and prediction visualization]

[Fig 15. Annotated image(downloaded)]

filename	box_x1	box_y1	box_x2	box_y2	class	confidence	age	age_bucket
30.jpg	40	27	230	158	Wrinkles	0.9982	85	70-85

[Fig 16. Prediction CSV(downloaded)]

Module 6: Backend Pipeline for Model Inference:

Purpose of Module 6:

- Module 6 is responsible for handling all backend operations related to model inference. This includes loading the trained deep learning model, preprocessing input images, running predictions, generating bounding boxes, and formatting outputs for frontend consumption.

Backend Design Philosophy:

- The backend pipeline is designed to be:
  - Modular
  - Reusable
  - Scalable
  - Easy to integrate with different frontends

- The inference logic is separated from the UI code to maintain clean architecture.

Backend Workflow:

- Load trained MobileNetV2-based model
- Load class index-to-label mapping
- Receive image from Flask frontend
- Preprocess image (resize, normalize)
- Extract face region without Haar Cascade
- Run deep learning inference
- Generate prediction probabilities

- Compute confidence scores
- Draw bounding boxes
- Return formatted results to frontend

Model Loading:

- Each input image undergoes:
  - Color space conversion
  - Resizing to model input size
  - Normalization
  - Batch dimension expansion

- This ensures compatibility with the trained model.

Prediction Logic:

- The model outputs a probability vector. The backend:
  - Extracts the highest probability
  - Maps index to class label
  - Converts probability to percentage
  - Associates predictions with bounding box coordinates

Fallback logic:

- Attempt face region extraction
- If face region is detected → classify face crop
- If face region is not detected → classify full image
- Ensure prediction output is always returned to UI

Data Flow Summary:

Steps	Description
Input	User uploads image
Processing	Backend preprocesses image
Inference	Model predicts class
Output	UI displays results

Evaluation Criteria:

- As per milestone requirements:
  - Seamless input-to-output flow
  - Inference time ≤ 5 seconds per image
  - Clean visualization without UI lag

Observed Performance:

- Average inference time: < 2-3 seconds
- No UI freezing during upload or rendering
- Smooth visualization across multiple images

- Milestone 3 successfully integrates frontend and backend components into a cohesive and functional application. The system demonstrates real-time image processing, accurate model inference, and an intuitive user interface. This milestone marks a critical step toward deploying the AI model as a usable end-to-end solution.

Deliverables:

- Flask frontend application (app.py)
- Modular backend inference script
- Integrated model inference pipeline
- Clean UI with prediction visualization
- End-to-end tested system