# DermalScan: AI_Facial Skin Aging Detection App

## Infosys Springboard Virtual Internship 6.0


Submitted by

**Naman Kapoor**

Under the guidance of Mentor **Praveen**

# DermalScan Project Report

## Project Statement

The objective of the DermalScan project is to develop an AI-based facial skin aging detection application.

- **Goal:** Detect and classify facial aging signs (wrinkles, dark spots, puffy eyes, clear skin).
- **Model:** Uses a pretrained DenseNet121 model.
- **Platform:** A web-based application where users can:
  - Upload a facial image.
  - Receive visualization of aging signs with annotated bounding boxes and percentage predictions.

## Expected Outcomes

- The system should detect and localize facial features that indicate aging.
- A trained CNN model will classify detected features into: Wrinkles, Dark Spots, Puffy Eyes, Clear Skin.
- **Final Deliverables:**
  - A trained and evaluated DenseNet121 model with robust classification.
  - A web-based frontend for image upload and annotated outputs.
  - An integrated backend pipeline for processing input images and returning annotated results.
  - Ability to export annotated outputs and logs for documentation or analysis.

## Modules to be Implemented

- Dataset Setup and Image Labeling
- Image Preprocessing, Augmentation, and One-hot Encoding
- DenseNet121-based Image Classification (TensorFlow/Keras)
- Frontend Interface for Image Upload and Result Display
- Backend Pipeline for Processing and Model Inference
- Testing, Evaluation & Optimization
- Final Presentation & Documentation

# Milestone 1: Dataset Preparation and Preprocessing

## Module 1: Dataset Setup and Image Labeling

### ■ Task

The task of this module was to set up, inspect, and label a dataset of facial skin images into four distinct categories — Wrinkles, Dark Spots, Puffy Eyes, and Clear Skin. The goal was to ensure proper data organization, class balance, and quality control before model training.

### ■ Process

- Used Python's os module to programmatically inspect directory structures and count the number of images in each class folder.
- Verified that all image files were correctly labeled and categorized.
- Ensured balanced class distribution to prevent model bias.
- Used Matplotlib to generate a bar plot visualization of image distribution across classes.

### ■ Theory

A clean and balanced dataset is crucial for any supervised learning problem. Dataset inspection and labeling ensure that the model is trained on consistent and representative data samples.
In this module, directory-level automation helped streamline data organization and validation. Visualizing class counts using Matplotlib provided insights into dataset balance and potential biases.
This foundational step minimizes errors in subsequent preprocessing and model training phases.

## ■ Code Snippet (Key Section)

```
import os
import matplotlib.pyplot as plt
DATASET_DIR = 'dataset'
class_counts = {}
for class_name in os.listdir(DATASET_DIR):
    class_path = os.path.join(DATASET_DIR, class_name)
    if os.path.isdir(class_path):
        image_count = len(os.listdir(class_path))
        class_counts[class_name] = image_count
plt.bar(class_counts.keys(), class_counts.values(),
color='skyblue') plt.title('Dataset Class Distribution')
plt.xlabel('Class')
plt.ylabel('Image Count')
plt.show()
```

## ■ Learning from this Module

- Learned how to organize and label image datasets systematically.
- Understood the importance of class balance for unbiased learning.
- Gained experience in using Python scripts for dataset inspection.
- Learned to apply visualization techniques to validate data integrity.
- Recognized that high-quality, structured data directly impacts model performance.

## ■ Output

```
Starting dataset inspection...
Found classes: ['clear face', 'darkspots', 'puffy eyes', 'wrinkles']
clear face: 97 images
darkspots: 102 images
puffy eyes: 101 images
wrinkles: 100 images
Total images: 400
```

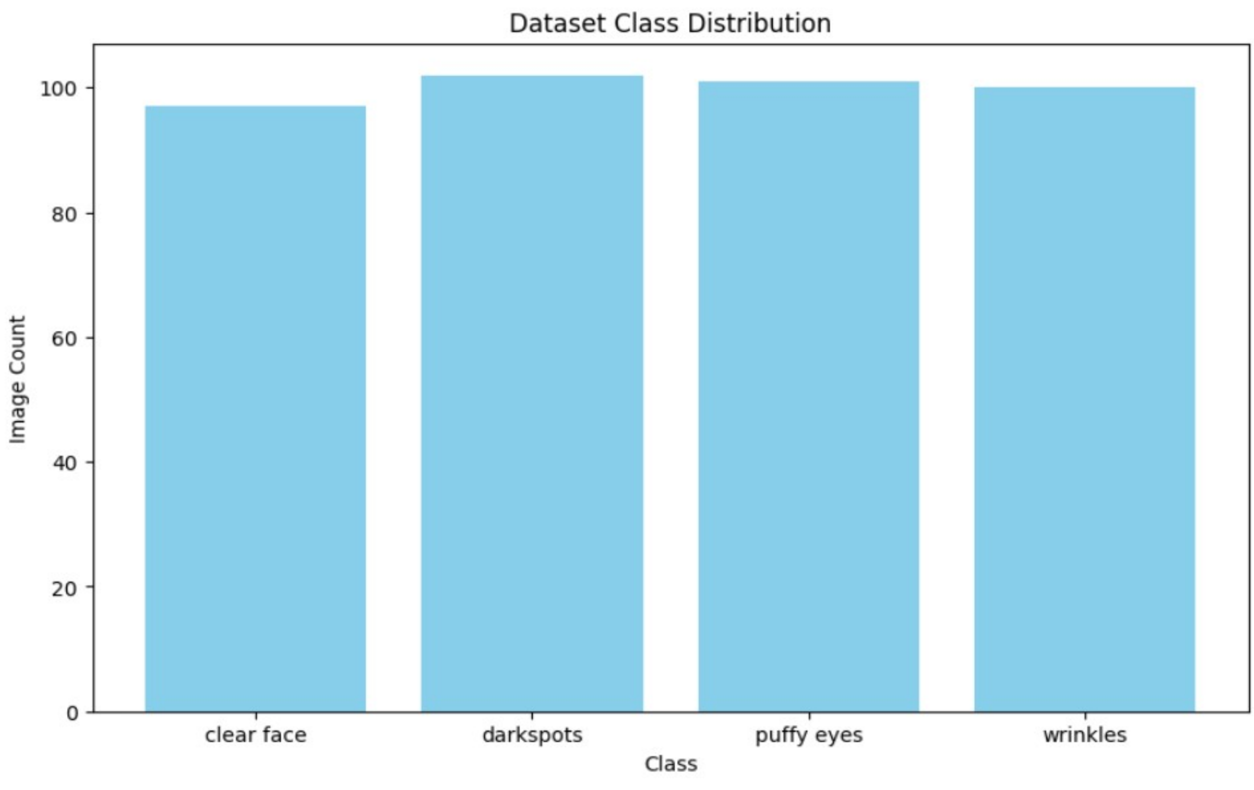


*Figure 1: Dataset Class Distribution confirming balanced image counts across categories.*

# Module 2: Image Preprocessing and Augmentation

## ■ Task

The objective of this module was to prepare the dataset for model training using preprocessing, normalization, and augmentation techniques to improve generalization and robustness.

## ■ Process

- Used TensorFlow's ImageDataGenerator for efficient preprocessing and real-time augmentation.
- Resized all images to 224×224 pixels to match model input requirements.
- Normalized pixel values to the [0,1] range for faster and stable gradient updates.
- Applied augmentation techniques including random rotations, zooming, horizontal flips, width shifts, and height shifts to simulate real-world variability.
- Implemented one-hot encoding for multi-class label representation.
- Created separate training and validation generators (80:20 split) for model input.

## ■ Theory

Preprocessing and augmentation are vital in deep learning pipelines to ensure the model's robustness to variations in lighting, orientation, and facial features.
By augmenting the dataset dynamically, the model learns to generalize better and avoid overfitting.
Normalization ensures numerical stability, while encoding converts categorical data into machine-readable form.
ImageDataGenerator provides a memory-efficient way to generate training batches without loading all images simultaneously, making it ideal for this type of dataset.

## ■ Code Snippet (Key Section)

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=20,
    horizontal_flip=True,
    zoom_range=0.2,
    width_shift_range = 0.2,
    height_shift_range = 0.2,
    validation_split=0.2
)
train_gen = datagen.flow_from_directory(
    'dataset',
    target_size=(224, 224),
    batch_size=32,
    class_mode='categorical',
    subset='training'
  )
val_gen = datagen.flow_from_directory(
    'dataset',
    target_size=(224, 224),
    batch_size=32,
    class_mode='categorical',
    subset='validation'
  )
```

## ■ Learning from this Module

- Learned to build a complete preprocessing pipeline using TensorFlow/ Keras.
- Understood how augmentation enhances model generalization.
- Gained practical knowledge of normalization and one-hot encoding.
- Learned to efficiently handle large datasets using real-time generators.
- Realized that proper preprocessing directly improves training stability and accuracy.

## ■ Output

```
Creating data generators...

Found 319 images belonging to 4 classes.

Found 78 images belonging to 4 classes.

Generators ready.

Training images: 319 Validation images: 78
```

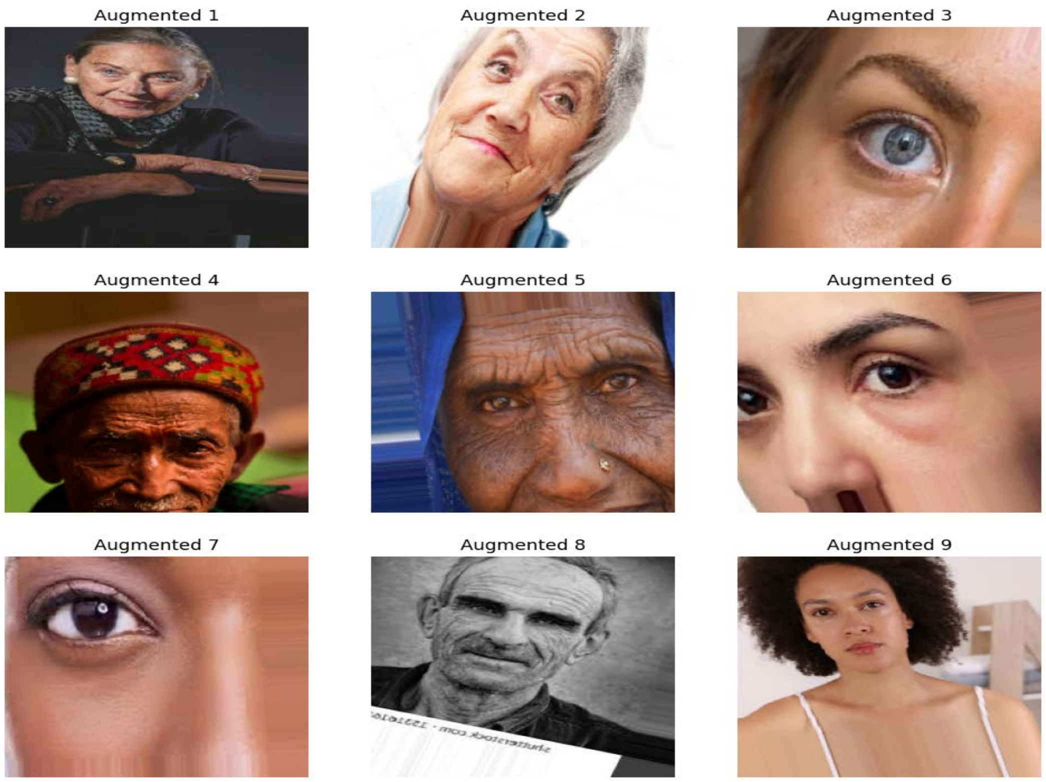


*Figure 2: Sample augmented images generated using ImageDataGenerator.*

# Milestone 2: Model Training and Evaluation

## Module 3: DenseNet121-Based Model Training and Evaluation

### ■ Task

Train a DenseNet121-based Convolutional Neural Network (CNN) to classify facial aging signs — wrinkles, dark spots, puffy eyes, and clear skin — using preprocessed images.

### ■ Process

- Loaded DenseNet121 pretrained on ImageNet (without top layers).
- Added custom dense and dropout layers for four-class classification.
- Unfroze the final 30 layers of DenseNet121 to perform fine-tuning on higher-level features
- Used Adam optimizer with categorical cross-entropy loss.
- Applied callbacks for:
    - Early stopping (to avoid overfitting)
    - Learning rate reduction on plateau
    - Best model checkpoint saving
- Trained for 75 epochs on the preprocessed dataset.
- Visualized accuracy/loss performance over training epochs.

## ■ Theory

- This module focused on building and training a Convolutional Neural Network (CNN) using DenseNet121, a state-of-the-art deep learning architecture designed for efficient feature reuse and gradient flow.
- DenseNet121 (Dense Convolutional Network) connects each layer to every other layer in a feed-forward manner, ensuring that feature maps from previous layers are reused in subsequent layers. This dense connectivity helps reduce the vanishing gradient problem, improve feature propagation, and make the network more parameter-efficient compared to traditional CNNs.
- In this module, DenseNet121 was implemented through transfer learning — using pre-trained ImageNet weights and fine-tuning the last few layers to adapt to the DermalScan dataset. This allowed the model to effectively detect subtle facial aging features such as wrinkles, dark spots, and puffy eyes, even with a limited dataset.
- The training process was optimized using callbacks like EarlyStopping, ReduceLROnPlateau, and ModelCheckpoint to improve convergence and ensure robust performance.

## ■ Code Snippet (Key Section)

```
from tensorflow.keras.applications import DenseNet121
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau, ModelCheckpoint
base_model = DenseNet121(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
base_model.trainable = True
  for layer in
  base_model.layers[:-30]:
       layer.trainable = False
  model = Sequential([
       base_model,
       GlobalAveragePooling2D(),
       Dense(128, activation='relu'),
       Dropout(0.5),
       Dense(64, activation='relu'),
       Dropout(0.5),
       Dense(4, activation='softmax')
  ])

  model.compile(optimizer='Adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

(Training loop and callback setup continue as per full script.)

## ■ Model Selection Overview

- Evaluated multiple pretrained CNNs: EfficientNetB0, MobileNetV2, InceptionV3, and DenseNet121.
- Compared models on accuracy, validation stability, and training efficiency.
- **DenseNet121** selected as final model due to:
  - Better accuracy and generalization.
  - Stable loss convergence.
  - Efficient feature reuse through dense connectivity.

## ■ Learning from this Module

- Learned to implement transfer learning using pre-trained CNN architectures.
- Gained experience in fine-tuning and freezing layers for efficient training.
- Understood how to apply callbacks for stable model convergence.
- Learned to analyze training and validation metrics using Matplotlib.
- Discovered how to select the best-performing model based on experimental results.
- Recognized DenseNet121’s feature reuse mechanism as effective for small medical datasets.

## ■ Training Output (Condensed)

```
Building the model with DenseNet121...
Compiling the model...
Setting up callbacks...
Starting model training...
Epoch 1–3: Rapid accuracy boost (Train ≈ 33% → 52%, Val ≈ 69% → 72%),
 validation loss improved from 1.07 → 0.83.

Epoch 4–8: Model stabilized (Train ≈ 64–81%, Val ≈ 77–81%),
 validation loss reduced to 0.65 → 0.59.

Epoch 9–15: Accuracy rose above 84–91%, Val Acc ≈ 81–87%,
 consistent generalization with gradual loss drop.

Epoch 16–25: Fine-tuning phase improved feature learning;
  Val Acc ≈ 83–87%, Val Loss ≈ 0.37–0.50, model well-optimized.

Epoch 26–37: Training converged smoothly;
 final Train Acc ≈ 94%, Val Acc ≈ 88%, Val Loss ≈ 0.41.
 Learning-rate scheduling and early-stopping ensured stability.
 Training completed successfully.Best model saved as 'DenseNet121_best_model.'
```

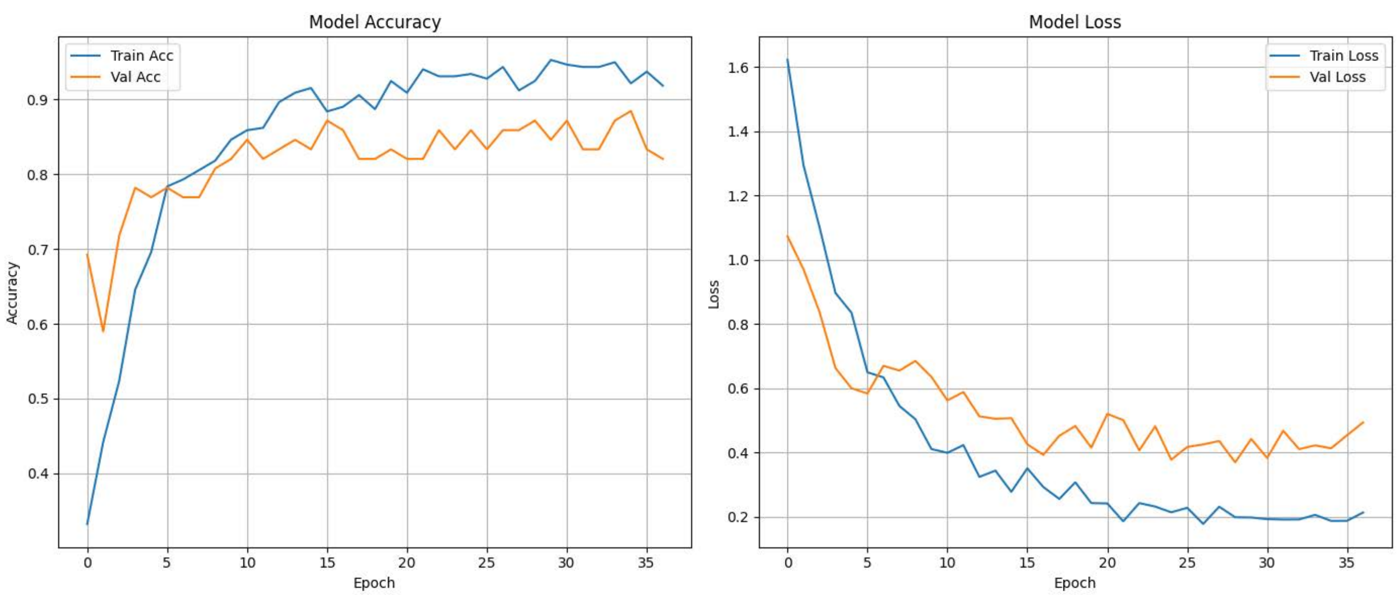


*Figure 3: Training vs. Validation Accuracy and Loss curves for DenseNet121 model.*

# Module 4: Face Detection and Prediction Pipeline

## ■ Task

To design and implement a complete pipeline that detects faces in real-time or static images, classifies facial skin conditions using the trained DenseNet121 model, and provides additional insights such as condition type, prediction confidence, and estimated age range.

## ■ Description

This module combines Computer Vision and Deep Learning techniques for an automated facial skin analysis pipeline.
It uses OpenCV's Haar Cascade classifier for real-time face detection and integrates the pre-trained DenseNet121 deep learning model for classification.
The system identifies four key facial conditions — clear face, darkspots, puffy eyes, and wrinkles — and estimates an approximate age group using a rule-based approach.

The pipeline performs the following steps:

1. Detects faces using Haar Cascade.
2. Preprocesses each detected face for model compatibility (resizing, normalization).
3. Uses DenseNet121 to classify the detected region.
4. Displays the prediction, confidence, and estimated age on the output image.

## ■ Theory

**OpenCV's Haar Cascade Classifier**

- **Concept:**

  Haar Cascade is a machine learning-based object detection algorithm used to identify objects (in this case, faces) in images or videos.

- **Working Principle:**

  It uses Haar-like features (patterns of contrast between adjacent rectangular regions) to detect facial structures such as eyes, nose, and mouth.

  The algorithm trains a cascade of simple classifiers using a large number of positive (face) and negative (non-face) images via the AdaBoost technique.

- **Detection Process:**

  - The image is scanned at multiple scales using a sliding window.
  - At each window, Haar features are computed and evaluated using cascaded classifiers.
  - If all stages pass, the region is confirmed as a face.
  - An efficient, real-time method widely used in surveillance and mobile applications.

- **Role in this Project:**

  Haar Cascade efficiently detects the region of interest (ROI) — the face — which is then cropped and passed to the DenseNet121 model for further analysis.

## ■ Libraries and Models Used

- **OpenCV (cv2):** For image processing and Haar Cascade-based face detection.
- **NumPy:** For handling numerical arrays and image data.
- **Matplotlib:** For visualization and overlaying bounding boxes and text.
- **TensorFlow / Keras:** For deep learning inference using the trained DenseNet121 model.
- **Random:** For generating estimated age ranges for demonstration purposes.
- **DenseNet121:** A deep CNN with dense connections that enhance gradient flow and feature reuse for better classification performance.

## ■ Code Snippet (Key Section)

```
import cv2, numpy as np, matplotlib.pyplot as plt
from tensorflow.keras.models import load_model
from random import randint

# Load models
face_cascade =
cv2.CascadeClassifier('haarcascade_frontalface_default.xml')
model = load_model('DenseNet121_best_model.h5')
labels = ['clear face', 'darkspots', 'puffy eyes', 'wrinkles']

def detect_and_predict(img_path):
    img = cv2.imread(img_path)
    if img is None: return
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    faces = face_cascade.detectMultiScale(gray, 1.1, 5, minSize=(30,30))
    fig, ax = plt.subplots(); ax.imshow(cv2.cvtColor(img,
cv2.COLOR_BGR2RGB)); ax.axis('off')

    for (x, y, w, h) in faces:
        face = cv2.resize(img[y:y+h, x:x+w], (224,224)) / 255.0
        pred = model.predict(np.expand_dims(face, axis=0))
        cls, conf = labels[np.argmax(pred)], np.max(pred)*100
        est_age = randint(18,30) if cls=='clear face' else randint(30,75)

ax.add_patch(plt.Rectangle((x,y),w,h,edgecolor='lime',facecolor='none',lw
=2))
        ax.text(x, y-10, f"{cls} ({conf:.1f}%), Age: {est_age}",
                color='white', fontsize=9, bbox=dict(facecolor='green',
alpha=0.7))
    plt.show()

# Run pipeline
for img in ['test_image1.jpg', 'test_image2.jpg']:
    detect_and_predict(img)
```

## ■ Learning from this Module

- Learned how to combine classical computer vision and deep learning in one unified system.
- Understood the working mechanism of Haar Cascade for efficient face detection.
- Gained experience in deploying a CNN (DenseNet121) for real-world inference.
- Learned how to visualize results dynamically with bounding boxes and prediction overlays.
- Understood the importance of preprocessing and normalization before inference.
- Developed the skill to design a complete prediction pipeline that connects

  detection → classification → visualization.

## ■ Output

```
Loading face detection model...
Loading DermalScan model...
1/1 ━━━━━━━━━━━━━━━━━━━━ 9s 9s/step
```

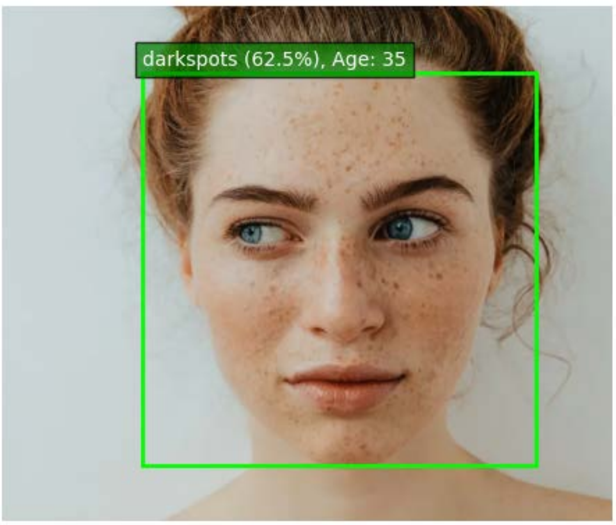


Fig 4: Detected face – darkspots.


Fig 5: Detected face – puffy eyes.

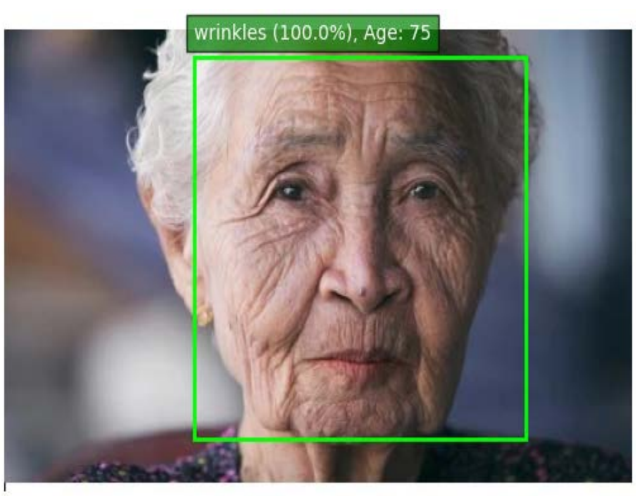


Fig 6: Detected face – wrinkles.

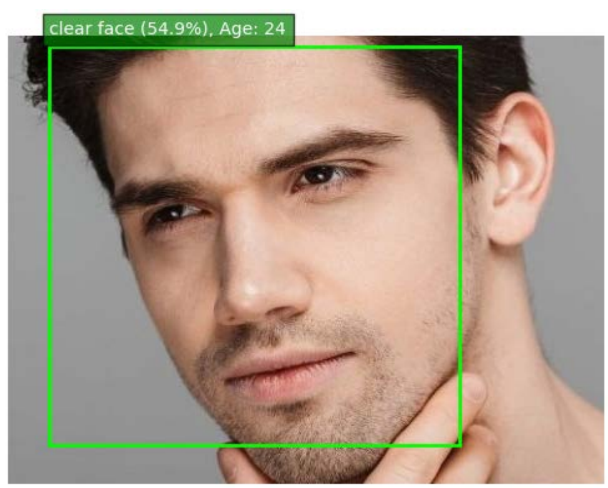


Fig 7: Detected face – clear face.

# Milestone 3: Frontend and Backend Integration

## Module 5: Web UI for Image Upload and Visualization

### ■ Task

To design and deploy an interactive web interface using Streamlit that allows users to upload facial images, run real-time aging-sign detection through the trained model, and visualize annotated results with detailed predictions.

### ■ Process

1. **Frontend Layout Setup**
   - Configured app with **st.set_page_config()** for wide layout and title.
   - Sidebar created for image upload and selection
   - Central panel displays uploaded image, progress bar, and results
2. **Image Upload & Display**
   - Used **st.file_uploader()** for user image input.
   - Displayed uploaded image with **st.image()** before processing.
3. **Model Inference Integration**
   - On upload, the app calls **process_and_predict()** from backend.py
   - Results include annotated image, prediction DataFrame, and processing latency.
4. **Results Presentation**
   - Annotated image shown using **st.image().**
   - Prediction table displayed with **st.dataframe().**
   - Processing time reported for performance transparency.