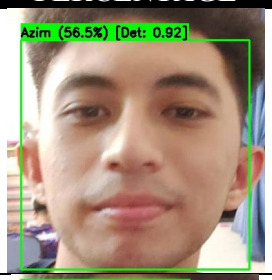
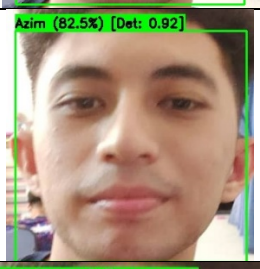
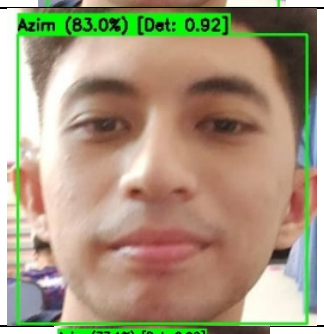
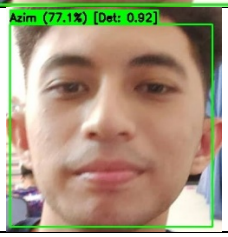
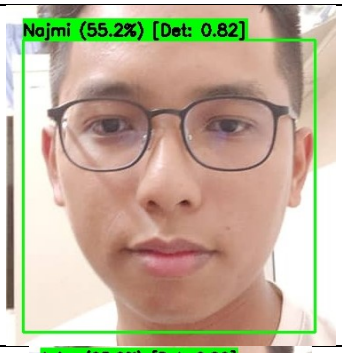
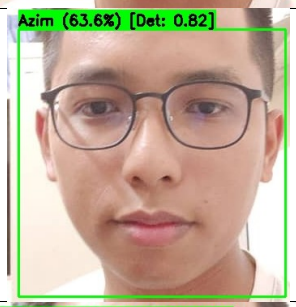
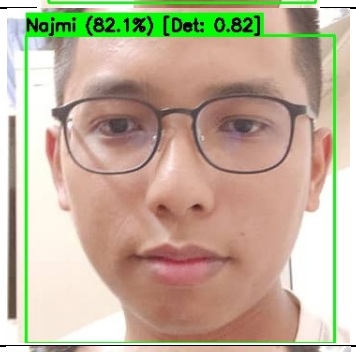
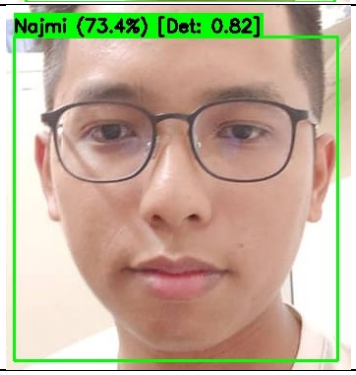
FACES	MODELS & DETECTOR	PERCENTAGE
 *Figure 1 - Azim*	VGG-Face (Model) & Retinaface (Backend)	
	OpenFace & Retinaface	
	Facenet & Retinaface	
	Facenet512 & Retinaface	

<table>
<tr>
<td rowspan="4"> 
Figure 2 - Najmi</td>
<td>VGG-Face & Retinaface</td>
<td> 
</td>
</tr>
<tr>
<td>OpenFace & Retinaface</td>
<td> 
</td>
</tr>
<tr>
<td>Facenet & Retinaface</td>
<td> 
</td>
</tr>
<tr>
<td>Facenet512 & Retinaface</td>
<td> 
</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</table>

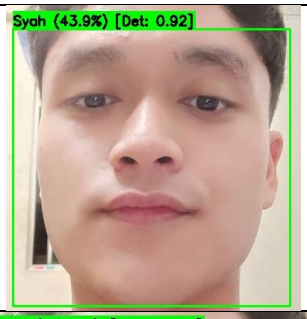
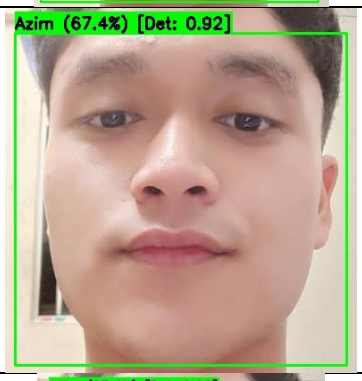
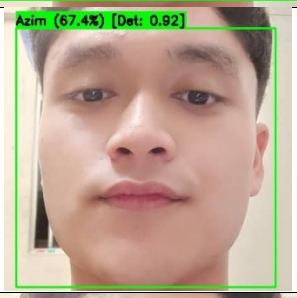
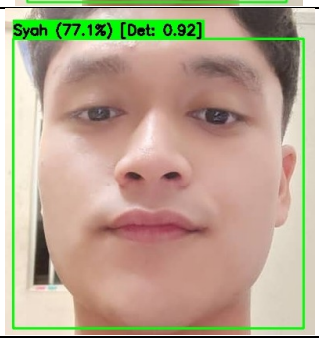
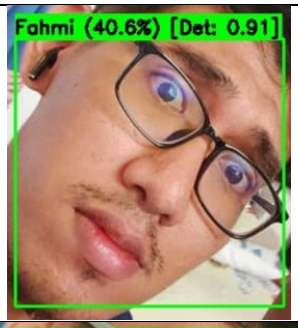
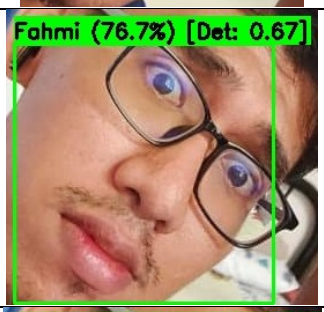
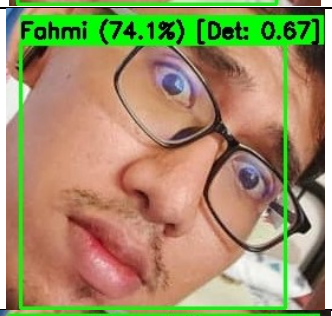
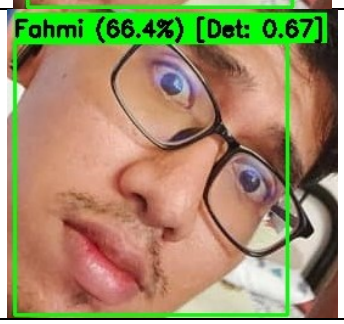
<table>
<tr>
<td rowspan="4"> 
Figure 3 - Syah</td>
<td>VGG-Face & Retinaface</td>
<td> 
</td>
</tr>
<tr>
<td>OpenFace & Retinaface</td>
<td> 
</td>
</tr>
<tr>
<td>Facenet & Retinaface</td>
<td> 
</td>
</tr>
<tr>
<td>Facenet512 & Retinaface</td>
<td> 
</td>
</tr>
</table>

 *Figure 4 - Fahmi*	VGG-Face & Retinaface	
	OpenFace & Retinaface	
	Facenet & Retinaface	
	Facenet512 & Retinaface	

ABOUT CODE

Explanation

1. `model_name='VGG-Face'`:
    - This specifies which deep learning model to use for creating face embeddings (numerical representations of facial features)
    - VGG-Face is a convolutional neural network model specifically trained for face recognition
    - It converts face images into 2622-dimensional feature vectors that capture facial characteristics
    - Other possible models in DeepFace include 'Facenet', 'OpenFace', 'DeepFace', 'DeepID', 'ArcFace', but this code uses VGG-Face
2. `detector_backend='opencv'`:
    - This specifies which face detection method to use to locate faces in images
    - 'opencv' refers to OpenCV's cascade classifier for face detection
    - DeepFace supports several other detector backends including:
        - 'mtcnn': Multi-task Cascaded Convolutional Networks
        - 'ssd': Single Shot Detector
        - 'dlib': Dlib's HOG + Linear SVM face detector
        - 'retinaface': RetinaFace detector
        - 'mediapipe': MediaPipe face detector

Interestingly, while this code specifies 'opencv' as the detector_backend in DeepFace, it actually uses MediaPipe for the main face detection through the `detect_faces` method. The OpenCV detector is only used during the embedding creation process.

The choice of these parameters affects:

- Speed of detection/recognition
- Accuracy of results
- Resource usage (memory and CPU/GPU)
- Minimum face size that can be detected
- Tolerance to different face angles and lighting conditions

Actual Code

```
import cv2
import os
import numpy as np
from deepface import DeepFace
import mediapipe as mp
import tkinter as tk
from tkinter import ttk, filedialog, messagebox
from PIL import Image, ImageTk


class FaceRecognitionGUI:
    def __init__(self, root):
        self.root = root
        self.root.title("Face Recognition System")
        self.root.geometry("1200x800")
```

```
        # Make the root window responsive
        self.root.columnconfigure(0, weight=1)
        self.root.rowconfigure(0, weight=1)

        # Initialize face recognition system
        self.recognition_system = FaceRecognitionSystem()

        # Create GUI elements
        self.create_gui()

    def create_gui(self):
        # Create main frame
        main_frame = ttk.Frame(self.root, padding="10")
        main_frame.grid(row=0, column=0, sticky="nsew")

        # Configure main frame grid weights
        main_frame.columnconfigure(1, weight=3)  # Right frame takes more
space
        main_frame.columnconfigure(0, weight=1)  # Left frame takes less
space
        main_frame.rowconfigure(0, weight=1)

        # Create left frame for controls
        left_frame = ttk.Frame(main_frame, padding="5")
        left_frame.grid(row=0, column=0, sticky="nsew")
        left_frame.columnconfigure(0, weight=1)

        # Database section
        ttk.Label(left_frame, text="Database Management", font=('Arial',
12, 'bold')).grid(row=0, column=0, pady=5,

sticky="w")

        self.database_path = tk.StringVar(value="./employee_database")
        ttk.Label(left_frame, text="Database Path:").grid(row=1, column=0,
pady=2, sticky="w")
        ttk.Entry(left_frame, textvariable=self.database_path).grid(row=2,
column=0, pady=2, sticky="ew")
        ttk.Button(left_frame, text="Browse Database",
command=self.browse_database).grid(row=3, column=0, pady=5,

sticky="ew")
        ttk.Button(left_frame, text="Load Database",
command=self.load_database).grid(row=4, column=0, pady=5,

sticky="ew")

        # Input image section
        ttk.Separator(left_frame, orient='horizontal').grid(row=5,
column=0, pady=10, sticky='ew')
        ttk.Label(left_frame, text="Image Processing", font=('Arial', 12,
'bold')).grid(row=6, column=0, pady=5,

sticky="w")

        ttk.Button(left_frame, text="Select Image",
command=self.browse_image).grid(row=7, column=0, pady=5,

sticky="ew")
        ttk.Button(left_frame, text="Process Image",
```

```
command=self.process_image).grid(row=8, column=0, pady=5,

sticky="ew")
        ttk.Button(left_frame, text="Save Result",
command=self.save_result).grid(row=9, column=0, pady=5, sticky="ew")

        # Status section
        ttk.Separator(left_frame, orient='horizontal').grid(row=10,
column=0, pady=10, sticky='ew')
        ttk.Label(left_frame, text="Status", font=('Arial', 12,
'bold')).grid(row=11, column=0, pady=5, sticky="w")

        # Make status text expand with window
        self.status_text = tk.Text(left_frame, wrap=tk.WORD)
        self.status_text.grid(row=12, column=0, pady=5, sticky="nsew")
        left_frame.rowconfigure(12, weight=1)

        # Create right frame for image display
        right_frame = ttk.Frame(main_frame, padding="5")
        right_frame.grid(row=0, column=1, sticky="nsew")

        # Configure right frame grid weights
        right_frame.columnconfigure(0, weight=1)
        right_frame.rowconfigure(1, weight=1)
        right_frame.rowconfigure(3, weight=1)

        # Image display areas
        ttk.Label(right_frame, text="Input Image", font=('Arial', 12,
'bold')).grid(row=0, column=0, pady=5)

        # Create frames to contain the image labels
        self.input_image_frame = ttk.Frame(right_frame)
        self.input_image_frame.grid(row=1, column=0, sticky="nsew")
        self.input_image_frame.columnconfigure(0, weight=1)
        self.input_image_frame.rowconfigure(0, weight=1)

        self.input_image_label = ttk.Label(self.input_image_frame)
        self.input_image_label.grid(row=0, column=0)

        ttk.Label(right_frame, text="Processed Image", font=('Arial', 12,
'bold')).grid(row=2, column=0, pady=5)

        self.output_image_frame = ttk.Frame(right_frame)
        self.output_image_frame.grid(row=3, column=0, sticky="nsew")
        self.output_image_frame.columnconfigure(0, weight=1)
        self.output_image_frame.rowconfigure(0, weight=1)

        self.output_image_label = ttk.Label(self.output_image_frame)
        self.output_image_label.grid(row=0, column=0)

        # Initialize variables
        self.input_image_path = None
        self.processed_image = None
        self.photo_images = []  # Keep reference to prevent garbage
collection
        self.original_input_image = None  # Store original input image
        self.original_processed_image = None  # Store original processed
image

        # Bind resize event
        self.root.bind("<Configure>", self.on_window_resize)
```

```
    def on_window_resize(self, event):
        # Only handle window resize events, not other widget configure
events
        if event.widget == self.root:
            # Update images if they exist
            if self.input_image_path and hasattr(self,
'original_input_image'):
                self.display_image(self.input_image_path,
self.input_image_label, True)
            if hasattr(self, 'original_processed_image') and
self.original_processed_image is not None:
                self.display_processed_image(True)

    def update_status(self, message):
        self.status_text.insert(tk.END, f"{message}\n")
        self.status_text.see(tk.END)
        self.root.update_idletasks()

    def browse_database(self):
        folder_path = filedialog.askdirectory()
        if folder_path:
            self.database_path.set(folder_path)
            self.update_status(f"Database path set to: {folder_path}")

    def load_database(self):
        try:
            self.recognition_system =
FaceRecognitionSystem(self.database_path.get())
            self.update_status("Database loaded successfully")
        except Exception as e:
            messagebox.showerror("Error", f"Failed to load database:
{str(e)}")
            self.update_status(f"Error loading database: {str(e)}")

    def browse_image(self):
        file_path = filedialog.askopenfilename(
            filetypes=[("Image files", "*.jpg *.jpeg *.png *.bmp *.gif
*.tiff")]
        )
        if file_path:
            self.input_image_path = file_path
            self.update_status(f"Selected image: {file_path}")
            self.display_image(file_path, self.input_image_label)

    def process_image(self):
        if not self.input_image_path:
            messagebox.showwarning("Warning", "Please select an input image
first")
            return

        try:
            self.update_status("Processing image...")
            self.processed_image =
self.recognition_system.process_image(self.input_image_path)

            if self.processed_image is not None:
                self.display_processed_image()
                self.update_status("Image processed successfully")
            else:
                self.update_status("Failed to process image")
```

```
        except Exception as e:
            messagebox.showerror("Error", f"Failed to process image:
{str(e)}")
            self.update_status(f"Error processing image: {str(e)}")

    def save_result(self):
        if self.processed_image is None:
            messagebox.showwarning("Warning", "No processed image to save")
            return

        file_path = filedialog.asksaveasfilename(
            defaultextension=".jpg",
            filetypes=[("JPEG files", "*.jpg"), ("All files", "*.*")]
        )

        if file_path:
            cv2.imwrite(file_path, self.processed_image)
            self.update_status(f"Result saved to: {file_path}")

    def get_display_size(self):
        # Calculate the maximum size for images based on window size
        right_frame_width = self.root.winfo_width() * 0.7  # 70% of window
width
        right_frame_height = (self.root.winfo_height() * 0.45)  # 45% of
window height for each image
        return right_frame_width, right_frame_height

    def display_image(self, image_path, label, resize=False):
        try:
            if not resize:
                # Load the original image first time
                self.original_input_image = Image.open(image_path)

            # Get current display size
            max_width, max_height = self.get_display_size()

            # Calculate scaling factor while maintaining aspect ratio
            image = self.original_input_image.copy()
            scale = min(max_width / image.width, max_height / image.height)
            new_size = (int(image.width * scale), int(image.height *
scale))

            # Resize image
            image = image.resize(new_size, Image.Resampling.LANCZOS)
            photo = ImageTk.PhotoImage(image)

            # Update label
            label.configure(image=photo)
            self.photo_images.append(photo)  # Keep a reference

        except Exception as e:
            messagebox.showerror("Error", f"Failed to display image:
{str(e)}")

    def display_processed_image(self, resize=False):
        try:
            if not resize:
                # Store the original processed image first time
                self.original_processed_image =
Image.fromarray(cv2.cvtColor(self.processed_image, cv2.COLOR_BGR2RGB))
```

```
            # Get current display size
            max_width, max_height = self.get_display_size()

            # Calculate scaling factor while maintaining aspect ratio
            image = self.original_processed_image.copy()
            scale = min(max_width / image.width, max_height / image.height)
            new_size = (int(image.width * scale), int(image.height *
scale))

            # Resize image
            image = image.resize(new_size, Image.Resampling.LANCZOS)
            photo = ImageTk.PhotoImage(image)

            # Update label
            self.output_image_label.configure(image=photo)
            self.photo_images.append(photo)  # Keep a reference

        except Exception as e:
            messagebox.showerror("Error", f"Failed to display processed
image: {str(e)}")


class FaceRecognitionSystem:
    def __init__(self, database_path='./employee_database'):
        self.database_path = database_path
        self.known_embeddings = {}
        self.recognition_threshold = 0.3

        # Initialize MediaPipe Face Detection
        self.mp_face_detection = mp.solutions.face_detection
        self.mp_drawing = mp.solutions.drawing_utils
        self.face_detection = self.mp_face_detection.FaceDetection(
            model_selection=1,
            min_detection_confidence=0.5
        )

        print("Initializing face embeddings...")
        self._load_known_faces()

    def _load_known_faces(self):
        """Pre-load and cache face embeddings for known faces"""
        for person_folder in os.listdir(self.database_path):
            folder_path = os.path.join(self.database_path, person_folder)
            if os.path.isdir(folder_path):
                try:
                    image_files = [f for f in os.listdir(folder_path)
                                   if f.lower().endswith(('.png', '.jpg',
'.jpeg'))]
                    if image_files:
                        image_path = os.path.join(folder_path,
image_files[0])
                        embedding = DeepFace.represent(
                            img_path=image_path,
                            model_name='Facenet512',
                            enforce_detection=False,
                            detector_backend='retinaface'
                        )
                        if embedding and len(embedding) > 0:
                            self.known_embeddings[person_folder] =
embedding[0]['embedding']
```

```
                            print(f"Loaded embedding for {person_folder}")
                except Exception as e:
                    print(f"Error loading embedding for {person_folder}:
{e}")

    def _get_face_embedding(self, image_path):
        """Get embedding for face in image"""
        try:
            embedding = DeepFace.represent(
                img_path=image_path,
                model_name='Facenet512',
                enforce_detection=False,
                detector_backend='retinaface'
            )
            if embedding and len(embedding) > 0:
                return embedding[0]['embedding']
            return None
        except Exception as e:
            print(f"Error getting embedding: {e}")
            return None

    def _compare_embeddings(self, embedding1, embedding2):
        """Compare two face embeddings using cosine similarity"""
        try:
            if embedding1 is None or embedding2 is None:
                return 0

            vec1 = np.array(embedding1).flatten()
            vec2 = np.array(embedding2).flatten()

            similarity = np.dot(vec1, vec2) / (np.linalg.norm(vec1) *
np.linalg.norm(vec2))
            return similarity
        except Exception as e:
            return 0

    def detect_faces(self, frame):
        """Detect faces in the frame using MediaPipe"""
        frame_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
        results = self.face_detection.process(frame_rgb)

        faces = []
        if results.detections:
            frame_height, frame_width, _ = frame.shape
            for detection in results.detections:
                bbox = detection.location_data.relative_bounding_box
                x = int(bbox.xmin * frame_width)
                y = int(bbox.ymin * frame_height)
                w = int(bbox.width * frame_width)
                h = int(bbox.height * frame_height)

                x = max(0, x)
                y = max(0, y)
                w = min(w, frame_width - x)
                h = min(h, frame_height - y)

                faces.append((x, y, w, h, detection.score[0]))

        return faces

    def recognize_face(self, image_path):
```

```
        """Recognize face in the image"""
        frame_embedding = self._get_face_embedding(image_path)
        if frame_embedding is None:
            return []

        recognized_faces = []
        for name, known_embedding in self.known_embeddings.items():
            similarity = self._compare_embeddings(frame_embedding,
known_embedding)
            if similarity > self.recognition_threshold:
                confidence = similarity * 100
                recognized_faces.append((name, confidence))

        recognized_faces.sort(key=lambda x: x[1], reverse=True)
        return recognized_faces

    def draw_face_box(self, frame, x, y, w, h, name, confidence,
detection_score):
        """Draw bounding box and labels on the face"""
        cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 0), 2)

        label = f"{name} ({confidence:.1f}%) [Det: {detection_score:.2f}]"

        font = cv2.FONT_HERSHEY_SIMPLEX
        font_scale = 0.6
        thickness = 2
        text_size = cv2.getTextSize(label, font, font_scale, thickness)[0]

        cv2.rectangle(frame,
                      (x, y - text_size[1] - 10),
                      (x + text_size[0], y),
                      (0, 255, 0),
                      cv2.FILLED)

        cv2.putText(frame,
                    label,
                    (x, y - 5),
                    font,
                    font_scale,
                    (0, 0, 0),
                    thickness)

    def process_image(self, input_image_path):
        """Process a single image and perform face recognition"""
        frame = cv2.imread(input_image_path)
        if frame is None:
            raise Exception(f"Could not read image {input_image_path}")

        faces = self.detect_faces(frame)
        if not faces:
            print("No faces detected in the image")
            return frame

        recognized_faces = self.recognize_face(input_image_path)

        if recognized_faces:
            name, confidence = recognized_faces[0]
            for (x, y, w, h, detection_score) in faces:
                self.draw_face_box(frame, x, y, w, h, name, confidence,
detection_score)
        else:
```

```
            for (x, y, w, h, detection_score) in faces:
                cv2.rectangle(frame, (x, y), (x + w, y + h), (255, 0, 0),
2)
                cv2.putText(frame, f"Unknown [Det: {detection_score:.2f}]",
                            (x, y - 5), cv2.FONT_HERSHEY_SIMPLEX, 0.6,
(255, 0, 0), 3)

        return frame


def main():
    root = tk.Tk()
    app = FaceRecognitionGUI(root)
    root.mainloop()


if __name__ == "__main__":
    main()
```