FaceClass

Ali Alizadeh

Overall System Architecture

1. Input Layer

- Classroom videos: Supports MP4, AVI, MOV formats
- Images: Frames extracted from videos

2. Processing Layer

- Face Detection: Multiple models (YOLO, RetinaFace, MTCNN, OpenCV)
- Face Tracking: Advanced algorithms (ByteTrack, Deep OC-SORT)
- Emotion Recognition: 8 emotion categories
- Attention Analysis: Gaze direction and head pose

3. Analysis Layer

- Face Identification: ArcFace, FaceNet, VGGFace
- Attendance Management: Automatic logging and duration calculation
- Spatial Analysis: Heatmaps and movement maps
- Session Management: Multiple simultaneous sessions

4. Output Layer

- Interactive Dashboard: Web-based user interface
- Comprehensive Reports: HTML, CSV, JSON
- Analytical Visuals: Heatmaps, movement paths, seating maps

Ensemble System Architecture(Face Detection)

1. Combining Different Models

```
# src/services/enhanced_face_detection.py
self.ensemble_models = config.get('face_detection.ensemble_models',
    ['yolo', 'mediapipe', 'mtcnn', 'opencv'])
```

- **YOLOv8**: Fast and efficient

- **MediaPipe**: Optimized for face detection
- **MTCNN**: High accuracy in details
- **OpenCV Cascade**: Basic support and always available

2. Voting System

```
def _apply_ensemble_voting(self, detections: List[DetectionResult]) ->
↪  List[DetectionResult]:
    """Apply ensemble voting to combine detections from multiple models."""
    if not self.ensemble_voting or len(detections) <= 1:
        return detections

    # Group detections by spatial proximity
    detection_groups = self._group_detections_by_proximity(detections)

    # Vote on each group
    final_detections = []
    for group in detection_groups:
        if len(group) >= 2:  # At least 2 models agree
            # Select the best detection from the group
            best_detection = self._select_best_detection(group)
            final_detections.append(best_detection)
        else:
            # Single detection, keep if confident enough
            if group[0].confidence >= self.confidence_threshold:
                final_detections.append(group[0])

    # Apply NMS to remove overlapping detections
    final_detections = self._apply_enhanced_nms(final_detections)

    return final_detections
```

Ensemble Processing Steps

Step 1: Multi Preprocessing

```
def _apply_preprocessing(self, frame: np.ndarray) -> List[Tuple[float, np.ndarray]]:
    """Apply multiple preprocessing techniques to improve detection."""
    preprocessed_frames = []

    # Original frame
    preprocessed_frames.append((1.0, frame.copy()))

    # Multi-scale versions
    for scale in self.scale_factors:
        if scale != 1.0:
            h, w = frame.shape[:2]
            new_h, new_w = int(h * scale), int(w * scale)
            scaled_frame = cv2.resize(frame, (new_w, new_h),
                                      interpolation=cv2.INTER_CUBIC)
            preprocessed_frames.append((scale, scaled_frame))

    # Enhanced versions
    if self.enable_denoising:
        denoised = cv2.fastNlMeansDenoisingColored(frame, None, 3, 3, 7, 21)
        preprocessed_frames.append((1.0, denoised))
```

```
    if self.enable_contrast_enhancement:
        # CLAHE for better contrast
        lab = cv2.cvtColor(frame, cv2.COLOR_BGR2LAB)
        l, a, b = cv2.split(lab)
        clahe = cv2.createCLAHE(clipLimit=3.0, tileGridSize=(8, 8))
        l = clahe.apply(l)
        enhanced = cv2.merge([l, a, b])
        enhanced = cv2.cvtColor(enhanced, cv2.COLOR_LAB2BGR)
        preprocessed_frames.append((1.0, enhanced))

    return preprocessed_frames
```

Step 2: Simultaneous Detection with All Models

```
def detect_faces_enhanced(self, frame: np.ndarray, frame_id: int = 0) ->
↪  List[DetectionResult]:
    """Enhanced face detection with preprocessing and ensemble detection."""
    # Apply preprocessing
    preprocessed_frames = self._apply_preprocessing(frame)

    # Run ensemble detection
    all_detections = []

    for scale_factor, processed_frame in preprocessed_frames:
        # Detect with each model
        for model_name, detector in self.detectors.items():
            if detector['loaded']:
                detections = self._detect_with_model(
                    detector, processed_frame, model_name, scale_factor
                )
                all_detections.extend(detections)

        # Detect with MediaPipe
        if self.mp_face_detection:
            mediapipe_detections = self._detect_with_mediapipe(
                processed_frame, scale_factor
            )
            all_detections.extend(mediapipe_detections)

    # Apply ensemble voting and NMS
    final_detections = self._apply_ensemble_voting(all_detections)

    return final_detections
```

Step 3: Grouping Similar Detections

```
def _group_detections_by_proximity(self, detections: List[DetectionResult],
                                   iou_threshold: float = 0.3) -> List[List[DetectionResult]]:
    """Group detections that are spatially close to each other."""
    if len(detections) <= 1:
        return [detections]

    groups = []
    used = set()

    for i, det1 in enumerate(detections):
        if i in used:
            continue
```

```
            group = [det1]
            used.add(i)

            for j, det2 in enumerate(detections[i+1:], i+1):
                if j in used:
                    continue

                if self._calculate_iou(det1.bbox, det2.bbox) > iou_threshold:
                    group.append(det2)
                    used.add(j)

            groups.append(group)

        return groups
```

Step 4: Selecting the Best Detection

```
def _select_best_detection(self, group: List[DetectionResult]) -> DetectionResult:
    """Select the best detection from a group based on multiple criteria."""
    if len(group) == 1:
        return group[0]

    # Score each detection
    scored_detections = []
    for det in group:
        score = 0.0

        # Confidence score (40%)
        score += det.confidence * 0.4

        # Quality score (30%)
        if det.quality_score:
            score += det.quality_score * 0.3

        # Size score (20%) - prefer medium-sized faces
        if det.face_size:
            size_score = 1.0 - abs(det.face_size - 64) / 64.0  # Peak at 64px
            score += max(0, size_score) * 0.2

        # Model reliability score (10%)
        model_scores = {'yolo': 0.9, 'mediapipe': 0.8, 'mtcnn': 0.7, 'opencv': 0.6}
        model_score = model_scores.get(det.model.split('_')[0], 0.5)
        score += model_score * 0.1

        scored_detections.append((det, score))

    # Return detection with highest score
    return max(scored_detections, key=lambda x: x[1])[0]
```

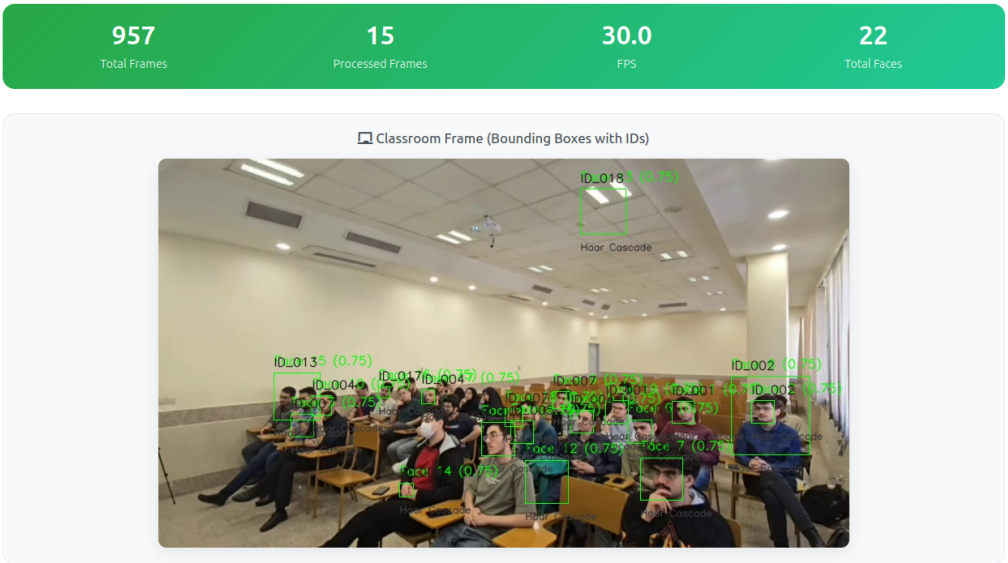

Figure 1: Face Detection

Advantages of Ensemble System

- Reduced False Negatives and higher Recall
- Reduced False Positives through voting and filtering
- Flexibility with models, preprocessing, and multi-scale detection

Conclusion

The Ensemble system in FaceClass provides:

- Higher accuracy by combining multiple models
- Better robustness through voting and filtering
- Flexibility for different scenarios
- Optimized performance with smart preprocessing

Emotion Analysis

1. Emotion Detection Models

```
_TARGET_CLASSES = [
    "angry", "disgust", "fear", "happy", "sad",
    "surprise", "neutral", "confused", "tired"
]

def analyze_emotions(face_images: List[np.ndarray]) -> List[str]:
    if _DEEPFACE_AVAILABLE:
        # Use DeepFace for precise emotion recognition
        return [_deepface_predict(img) for img in face_images]
    else:
```

```
        # Fallback to lightweight heuristic algorithm
        return [_heuristic_predict(img) for img in face_images]
```

complete code:

```
def analyze_emotions(face_images: List[np.ndarray]) -> List[str]:
    predictions: List[str] = []
    if not face_images:
        return predictions
    if _DEEPFACE_AVAILABLE:
        for img in face_images:
            predictions.append(_deepface_predict(img))
        return predictions
    # Fallback
    for img in face_images:
        predictions.append(_heuristic_predict(img))
    return predictions
```

2. Detection Algorithms

- High accuracy: Uses pre-trained models
- 9 emotion classes: Covers a wide range of states
- Smart recognition: Differentiates between similar emotions

```
def _heuristic_predict(face_img: np.ndarray) -> str:
    gray = cv2.cvtColor(face_img, cv2.COLOR_BGR2GRAY)
    mean = float(np.mean(gray))
    std = float(np.std(gray))

    # Simple rules based on brightness and contrast
    if mean > 175 and std > 45:
        return 'surprise'  # bright and high contrast
    if mean > 155:
        return 'happy'     # bright
    if mean < 55 and std < 25:
        return 'tired'     # dark and low contrast
    if 55 <= mean < 90 and std < 35:
        return 'sad'       # dark
    if 120 <= mean <= 150 and 20 <= std <= 35:
        return 'confused'  # medium range
    return 'neutral'
```

3. Attention Analysis

```
# File: src/services/attention_analysis.py
def analyze_attention(face_landmarks: Any) -> Dict:
    # Extract head pose angles
    yaw = float(face_landmarks.get('yaw', 0.0))      # left-right
    pitch = float(face_landmarks.get('pitch', 0.0))   # up-down
    roll = float(face_landmarks.get('roll', 0.0))     # rotation

    # Determine gaze direction
    if pitch >= 12.0:
        gaze_direction = 'down'
    elif pitch <= -12.0:
```

```
    gaze_direction = 'up'
elif yaw <= -10.0:
    gaze_direction = 'left'
elif yaw >= 10.0:
    gaze_direction = 'right'
else:
    gaze_direction = 'front'

# Determine attention
is_attentive = (abs(yaw) <= 25.0) and (abs(pitch) <= 20.0) and (gaze_direction ==
↪  'front')

return {
    'is_attentive': bool(is_attentive),
    'engaged': bool(is_attentive),
    'yaw': yaw,
    'pitch': pitch,
    'roll': roll,
    'gaze_direction': gaze_direction
}
```

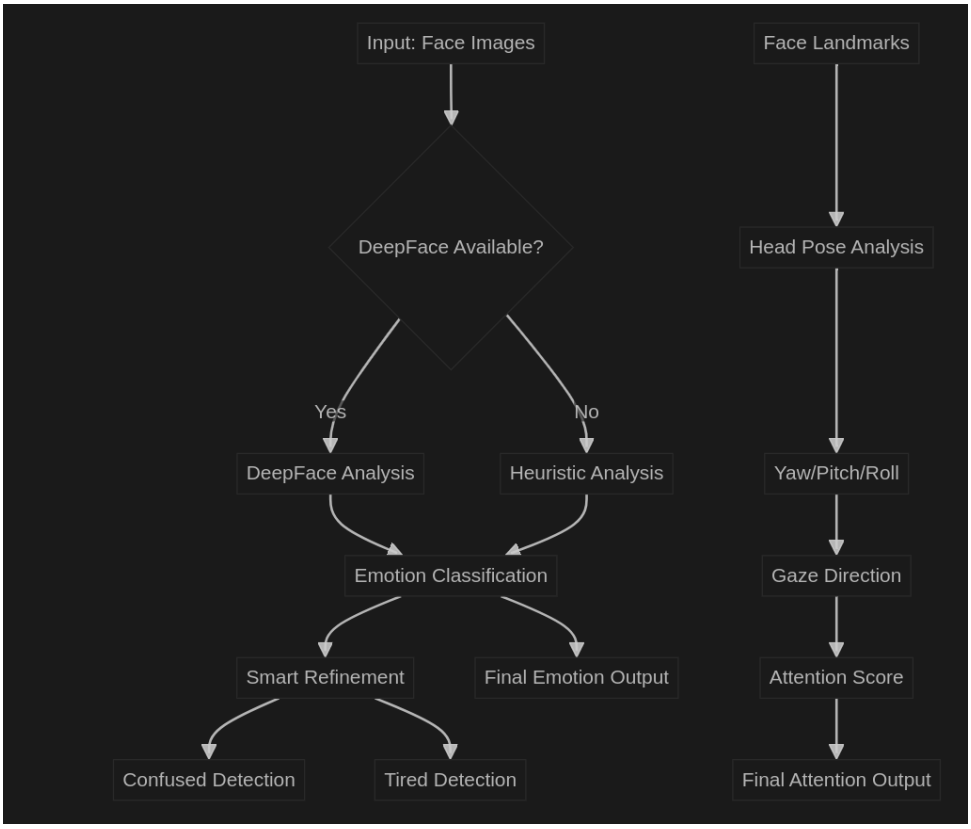

Figure 2: flow Emotion Analysis

Figure 3: Emotion/Attention Analysis

Attendance & Spatial Analysis

Video → Face Detection → Attendance Tracking → Spatial Analysis → Maps

1. Generate Heatmap

```python
def generate_heatmap(
    positions_by_student: Dict[str, List[Tuple[int, int]]],
    frame_size: Tuple[int, int],
    blur_ksize: int = 31
) -> np.ndarray:
    h, w = frame_size
    acc = np.zeros((h, w), dtype=np.float32)

    for pts in positions_by_student.values():
        for (x, y) in pts:
            if 0 <= x < w and 0 <= y < h:
                cv2.circle(acc, (x, y), 8, 1.0, thickness=-1)

    if blur_ksize > 1:
        acc = cv2.GaussianBlur(acc, (blur_ksize, blur_ksize), 0)

    acc_norm = cv2.normalize(acc, None, 0, 255, cv2.NORM_MINMAX).astype(np.uint8)
    heat = cv2.applyColorMap(acc_norm, cv2.COLORMAP_JET)
    return heat
```

2. render movement paths

```python
def render_movement_paths(
    positions_by_student: Dict[str, List[Tuple[int, int]]],
    frame_size: Tuple[int, int]
) -> np.ndarray:
    h, w = frame_size
    canvas = np.ones((h, w, 3), dtype=np.uint8) * 255

    def color_for(sid: str) -> Tuple[int, int, int]:
        seed = sum(ord(c) for c in sid)
        np.random.seed(seed % 2**32)
        return tuple(int(v) for v in np.random.randint(0, 255, size=3))

    for sid, pts in positions_by_student.items():
        if len(pts) < 2:
            continue
        color = color_for(sid)

        for i in range(1, len(pts)):
            cv2.line(canvas, pts[i-1], pts[i], color, 2)

        cv2.putText(canvas, sid, pts[-1], cv2.FONT_HERSHEY_SIMPLEX, 0.5, color, 1,
        ↪  cv2.LINE_AA)

    return canvas
```

3. render seat map

```python
def render_seat_map(
    positions_by_student: Dict[str, List[Tuple[int, int]]],
    frame_size: Tuple[int, int]
) -> np.ndarray:
```

```
h, w = frame_size
canvas = np.ones((h, w, 3), dtype=np.uint8) * 255

for sid, pts in positions_by_student.items():
    if not pts:
        continue

    xs = sorted(p[0] for p in pts)
    ys = sorted(p[1] for p in pts)
    cx = xs[len(xs)//2]
    cy = ys[len(ys)//2]

    cv2.circle(canvas, (cx, cy), 10, (0, 128, 255), thickness=-1)

    cv2.putText(canvas, sid, (cx+12, cy-12), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 0),
    ↪  1, cv2.LINE_AA)

return canvas
```

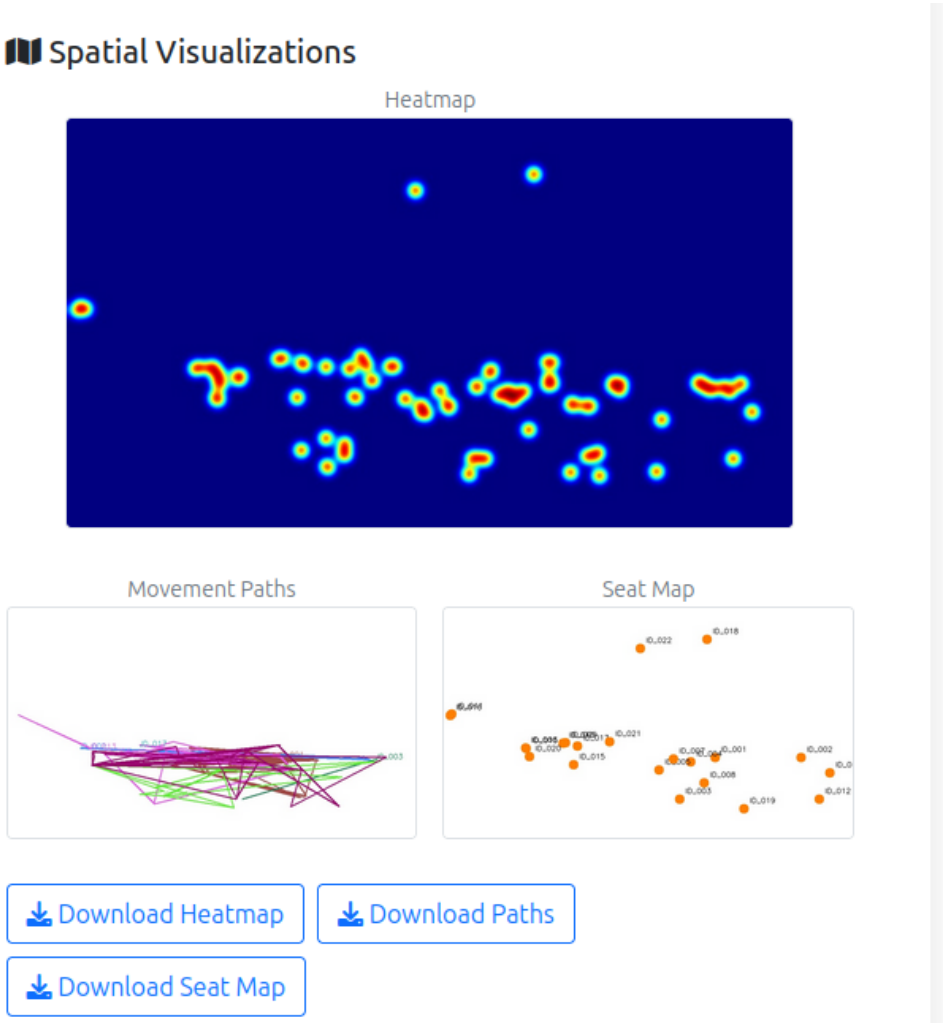

Figure 4: Attendance & Spatial Analysis