VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

School of Computer Science and Engineering

(SCOPE)

MTech-Computer Science with Business Analytics

ASSIGNMENT 5

CSE4076 - Image and Video Analytics

BY:

VISHALI SHARMA (21MIA1066)

Submitted to:

Dr. Saranyaraj D

Git hub: https://github.com/Vishali2002/Task.git

Task 1: Motion Estimation and Event Detection in a Video

Objective:
Detect motion and specific events in a video using frame differencing or optical flow to estimate motion and identify events without machine learning.

Problem Statement:

Detecting motion and identifying significant events are key tasks in video processing, and they are used in a variety of applications, including security monitoring, sports analysis, video compression, and automated event detection. Manual inspection of motion patterns and event occurrence in videos is inefficient, especially for large datasets. Thus, a systematic approach is needed to automatically process video frames, detect motion, and highlight significant events.

This project aims to build an automated pipeline that:

1. Converts video frames to grayscale for efficient processing.
2. Estimates motion by analyzing histogram differences between consecutive frames.
3. Detects significant events based on the intensity of motion.
4. Visualizes the histogram data for better insight into frame-to-frame changes.

The pipeline consists of four modules:

1. **load_video.py**: Loads a video file, extracts frames, and saves them as grayscale images.
2. **motion_estimation.py**: Computes the histogram differences between consecutive grayscale frames to estimate motion intensity and saves the motion data.
3. **event_detection.py**: Detects significant events based on motion intensity by applying a threshold and marks these frames.
4. **visualize_histogram.py**: Visualizes the histograms of pixel intensity for each frame and saves these plots for analysis.

Expected output:

load_video.py:

- The module extracts individual frames from the video and converts them to grayscale.
- Each frame is saved as an image (e.g., .png format) in a specified directory.

Output:

- A folder containing grayscale frames of the video (e.g., frame_001.png, frame_002.png, etc.).

motion_estimation.py:

- This module reads the grayscale frames, computes the histogram differences between consecutive frames, and saves the motion intensity data.

Output:

- A file (motion_estimation.txt) containing motion intensity values for each consecutive frame pair.

event_detection.py:

- The module reads the motion intensity data from motion_estimation.txt, detects significant events based on a predefined motion threshold, and marks the corresponding frames.
- These event frames are highlighted and saved in a new directory.

Output:

- A folder containing annotated frames where significant events (motion exceeding the threshold) occurred, with marked moving regions (e.g., event_frame_001.png, event_frame_002.png).

visualize_histogram.py:

- This module generates and saves histograms of pixel intensity distributions for each grayscale frame.

Output:

- A folder containing histogram plots for each frame (e.g., histogram_001.png, histogram_002.png), showing the distribution of pixel intensities.

Methodology

1. Load Video:

- **Objective**: Load the video and convert frames to grayscale.
- **Steps**:
    1. Use OpenCV to load the video.
    2. Convert each frame to grayscale.
    3. Save frames sequentially in a designated directory.
- **Result**: Grayscale frames saved for further processing.

2. Motion Estimation:

- **Objective**: Detect motion by comparing histograms of consecutive frames.

- **Steps**:
    1. Load grayscale frames.
    2. Compute and compare histograms of consecutive frames.
    3. Save motion intensity data in a text file (motion_estimation.txt).
- **Result**: Motion intensity values for each frame pair.

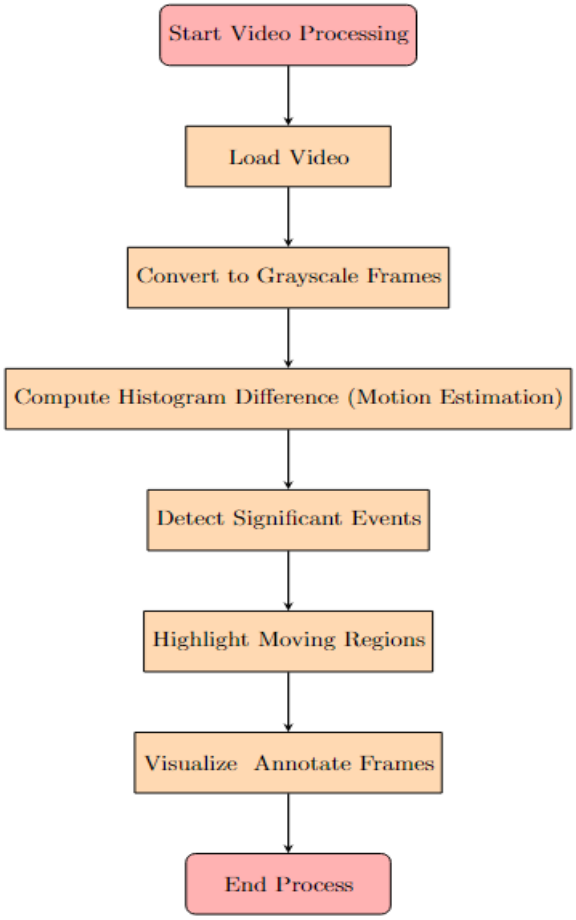
3. Event Detection:

- **Objective**: Identify significant events based on motion intensity.
- **Steps**:
    1. Read motion intensity data.
    2. Detect events where intensity exceeds a threshold.
    3. Mark event frames and highlight regions with significant motion.
- **Result**: Annotated event frames with timestamps.

4. Result:

- **Objective**: Visualize and annotate frames with detected motion.
- **Steps**:
    1. Highlight moving regions in frames.
    2. Display and save frames with event annotations.

BLOCK DIAGRAM:



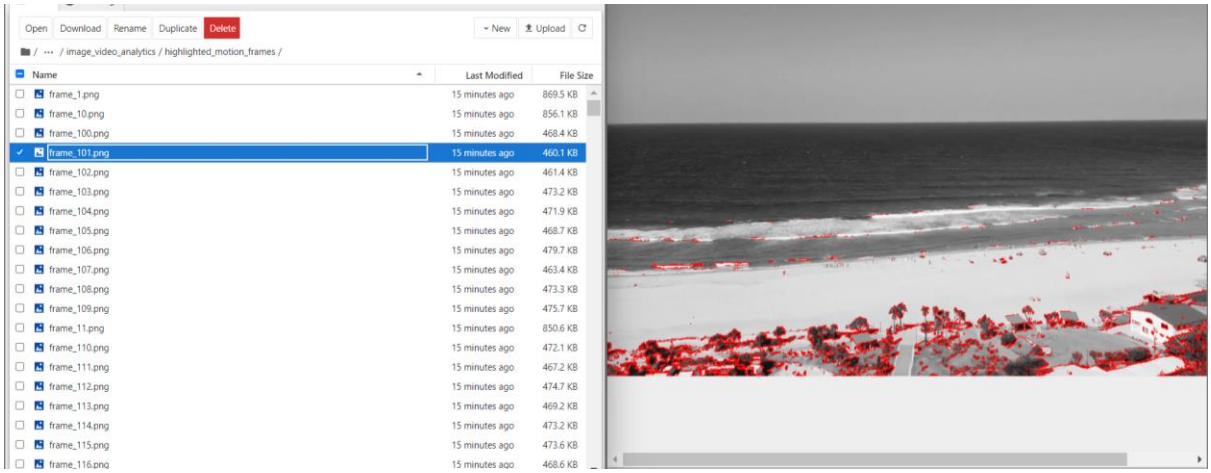
PYTHON IMPLEMENTATION:

```python
import cv2
import os

# Load the video
video_path = '33871-398473585_small.mp4'
cap = cv2.VideoCapture(video_path)

# Directory to save grayscale frames
output_dir = 'frames'
os.makedirs(output_dir, exist_ok=True)

# Frame counter
frame_num = 0

while cap.isOpened():
    ret, frame = cap.read()
    if not ret:
        break

    # Convert frame to grayscale
    gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    # Save the grayscale frame
    frame_filename = os.path.join(output_dir, f'frame_{frame_num}.png')
    cv2.imwrite(frame_filename, gray_frame)

    frame_num += 1

cap.release()
cv2.destroyAllWindows()

print(f"Total {frame_num} frames extracted and saved in {output_dir}")
```

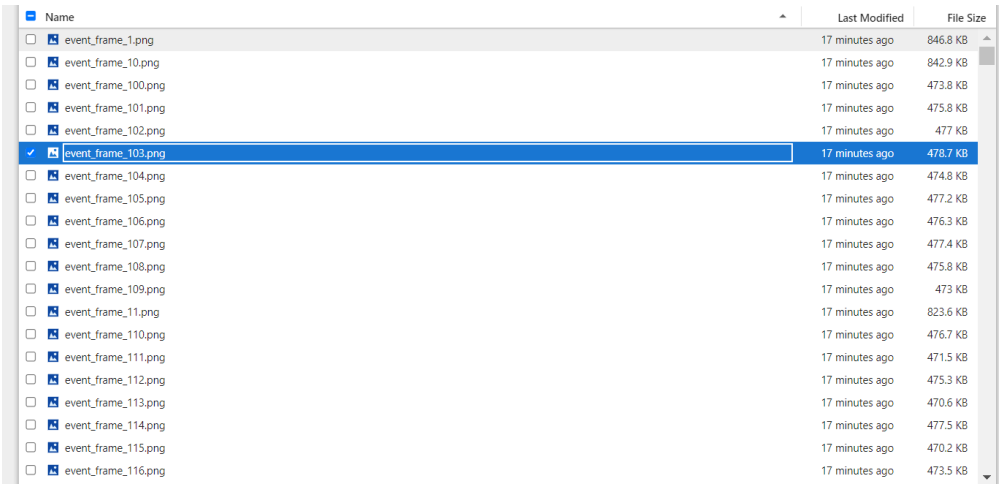
```
Total 628 frames extracted and saved in frames
```

```
    if previous_frame is None:
        previous_frame = gray_frame
        continue

    # Compute absolute difference between the current frame and the previous frame
    diff_frame = cv2.absdiff(previous_frame, gray_frame)

    # Threshold the difference to highlight moving regions
    _, thresh_frame = cv2.threshold(diff_frame, motion_threshold, 255, cv2.THRESH_BINARY)

    # Calculate the motion estimation as the sum of the pixel differences
    motion_value = np.sum(thresh_frame)
    motion_estimations.append(motion_value)

    # Convert the grayscale frame to color so we can highlight moving regions
    color_frame = cv2.cvtColor(gray_frame, cv2.COLOR_GRAY2BGR)

    # Highlight moving regions in red
    color_frame[thresh_frame > 0] = [0, 0, 255]

    # Save the highlighted frame
    highlighted_frame_path = os.path.join(highlight_dir, frame_file)
    cv2.imwrite(highlighted_frame_path, color_frame)

    # Update the previous frame for the next iteration
    previous_frame = gray_frame

# Save the motion estimations to a file
np.savetxt(motion_estimation_file, motion_estimations, fmt='%d')

print(f"Highlighted motion frames saved in {highlight_dir}")
print(f"Motion estimation values saved in {motion_estimation_file}")
```

```
Highlighted motion frames saved in highlighted_motion_frames
Motion estimation values saved in motion_estimation.txt
```




```
frame_files = sorted([f for f in os.listdir(frame_dir) if f.endswith('.png')], key=lambda x: int(x.split('_')[1].split('.')[0]))

# Event threshold (adjust based on experimentation)
event_threshold = 50000

# Directory to save event-annotated frames
event_dir = 'annotated_event_frames'
os.makedirs(event_dir, exist_ok=True)

event_count = 0
timestamps = []

# FPS (adjust based on your video)
fps = 30  # Example FPS value

# Adjust the loop to avoid out-of-range errors
for i, motion_value in enumerate(motion_estimation[:-1]):  # Stop at the second-to-last motion value
    if motion_value > event_threshold:
        event_count += 1

        # Load the frame where the event occurs
        frame_path = os.path.join(frame_dir, frame_files[i+1])  # +1 because motion is calculated for the next frame
        frame = cv2.imread(frame_path)

        if frame is not None:
            # Optionally annotate the frame here
            annotated_frame_path = os.path.join(event_dir, f'event_frame_{event_count}.png')
            cv2.imwrite(annotated_frame_path, frame)

            # Store the timestamp (frame index / fps gives the time in seconds)
            timestamp = (i+1) / fps
            timestamps.append(timestamp)

# Optionally, save the timestamps of events to a file
np.savetxt('event_timestamps.txt', timestamps)
```



```python
# Get the list of frame filenames
frame_files = sorted([f for f in os.listdir(frame_dir) if f.endswith('.png')], key=lambda x: int(x.split('_')[1].split('.')[0]))

# Create a directory to save the histograms
histogram_dir = 'histograms'
os.makedirs(histogram_dir, exist_ok=True)

for frame_file in frame_files:
    # Load the grayscale frame
    gray_frame = cv2.imread(os.path.join(frame_dir, frame_file), cv2.IMREAD_GRAYSCALE)

    # Calculate histogram
    hist = cv2.calcHist([gray_frame], [0], None, [256], [0, 256])

    # Plot and save the histogram
    plt.figure()
    plt.title(f"Histogram for {frame_file}")
    plt.xlabel("Pixel Value")
    plt.ylabel("Frequency")
    plt.plot(hist)

    hist_file = os.path.join(histogram_dir, f'hist_{frame_file}.png')
    plt.savefig(hist_file)
    plt.close()

    print(f"Histogram saved for {frame_file}")
```

```
Histogram saved for frame_0.png
Histogram saved for frame_1.png
Histogram saved for frame_2.png
Histogram saved for frame_3.png
Histogram saved for frame_4.png
Histogram saved for frame_5.png
Histogram saved for frame_6.png
Histogram saved for frame_7.png
Histogram saved for frame_8.png
```

Task 2: Estimating Sentiments of People in a Crowd – Gesture Analysis and Image Categorization

Objective:

Estimate the sentiments of individuals in a crowd using basic gesture analysis techniques, such as detecting facial expressions or hand gestures, without using machine learning models.

Problem Statement

The objective of this project is to estimate the sentiments of individuals in a crowd through basic gesture analysis techniques. This will involve detecting facial expressions and hand gestures in a set of images without utilizing machine learning models. The project aims to

develop a methodology that can identify and categorize the emotional states of individuals based on their facial features and gestures, ultimately providing an overall sentiment of the crowd.

Expected Outcomes

1. **Sentiment Analysis**:
    - Identification of individual sentiments (happy, sad, neutral) based on facial gestures.
    - Overall sentiment categorization of the crowd based on the majority sentiment detected.
2. **Visualization**:
    - Display of key facial features (eyes, mouth, eyebrows) used for gesture analysis in each processed image.
    - Summary of individual and overall sentiments, presented in a clear and interpretable format.
3. **Demonstration of Technique**:
    - A functioning system capable of processing a set of crowd images to extract emotional information without relying on machine learning models.

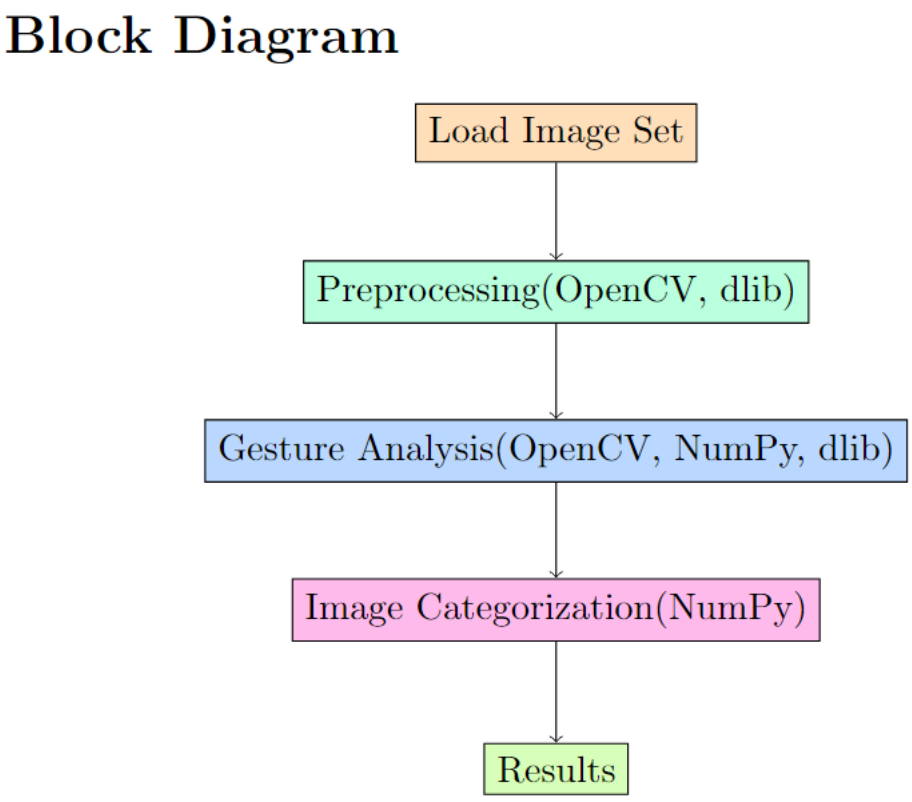
Methodology

1. **Load Image Set**:
    - Import the set of images containing individuals in a crowd.
2. **Preprocessing**:
    - Implement traditional face detection techniques, such as skin-color-based detection, to identify and isolate faces within the images.
    - Use skin color thresholding or simple contour analysis to detect hand gestures.
3. **Gesture Analysis**:
    - Perform geometric analysis to extract key facial features (positions of the eyes, mouth, eyebrows).
    - Classify emotions based on the geometry of facial expressions:
        - Happiness is indicated by upward curvature of the mouth.
        - Sadness is indicated by downward curvature of the mouth and raised eyebrows.
4. **Image Categorization**:

- Analyze individual sentiments and categorize the images based on the average sentiment of the crowd (e.g., majority happy or sad).

5. **Result Output**:
   - Present the identified sentiment for each individual alongside the overall sentiment of the crowd.
   - Display the key facial features utilized for the analysis.

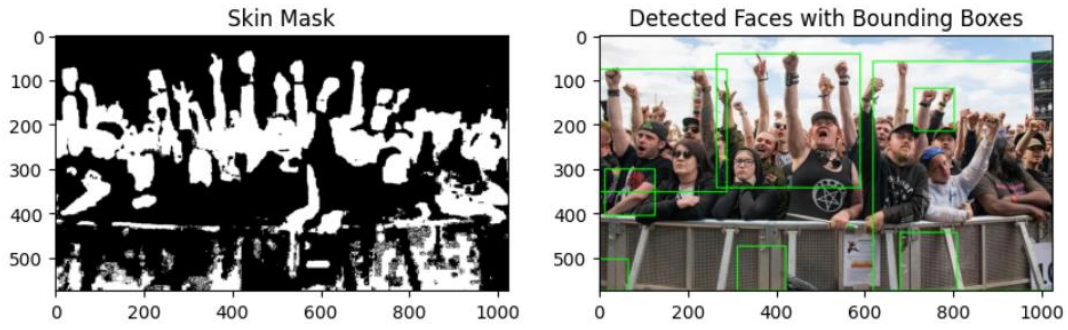
Block Diagram:



PYTHON IMPLEMENTATION:

```python
import cv2
import matplotlib.pyplot as plt

# Load the image
image_path = '/content/crowd_21mia1066.jpg'
image = cv2.imread(image_path)

# Convert the image from BGR to RGB
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

# Display the image
plt.imshow(image_rgb)
plt.title('Crowd Image')
plt.axis('off')
plt.show()
```


```python
    # Filter contours based on aspect ratio to detect potential faces (face is approximately square)
    aspect_ratio = w / float(h)
    if 0.75 < aspect_ratio < 1.25:  # Accept aspect ratios close to 1
        # Draw a rectangle around the detected face region
        cv2.rectangle(image_rgb, (x, y), (x + w, y + h), (0, 255, 0), 2)  # Green bounding box for face

# Plot the results
plt.figure(figsize=(10, 6))
plt.subplot(1, 2, 1)
plt.imshow(skin_mask, cmap='gray')
plt.title("Skin Mask")

plt.subplot(1, 2, 2)
plt.imshow(image_rgb)
plt.title("Detected Faces with Bounding Boxes")

plt.show()
```



```
            start = tuple(contour[s][0])
            end = tuple(contour[e][0])
            far = tuple(contour[f][0])

            # Calculate the angle between the fingers
            a = np.linalg.norm(np.array(start) - np.array(far))
            b = np.linalg.norm(np.array(end) - np.array(far))
            c = np.linalg.norm(np.array(start) - np.array(end))
            angle = np.arccos((a ** 2 + b ** 2 - c ** 2) / (2 * a * b))

            # If the angle is less than 90 degrees, count it as a finger
            if angle <= np.pi / 2:
                finger_count += 1
                cv2.circle(image, far, 5, [0, 255, 0], -1)

        # Classify gesture based on number of fingers detected
        gesture_text = {0: "Fist", 1: "One Finger", 2: "Two Fingers",
                        3: "Three Fingers", 4: "Open Hand"}.get(finger_count, "Unknown Gesture")

        # Display gesture text on the image
        cv2.putText(image, gesture_text, (x, y - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.8, (255, 255, 255), 2)

Display results
.t.figure(figsize=(10, 6))
.t.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
.t.title("Detected Gestures")
.t.axis('off')
.t.show()
```


```
    # Check mouth curvature
    mouth_center = np.mean(mouth_coords, axis=0)
    left_corner = mouth_coords[0]
    right_corner = mouth_coords[6]

    mouth_curvature = (mouth_center[1] - left_corner[1]) + (mouth_center[1] - right_corner[1])

    # Emotion classification
    if mouth_curvature > 0:
        return 'Happy'
    elif mouth_curvature < 0:
        return 'Sad'
    else:
        return 'Neutral'

# Apply emotion classification to detected faces
for face in faces:
    landmarks = predictor(gray, face)
    emotion = classify_emotion(landmarks)

    # Annotate image with classified emotion
    cv2.putText(image, emotion, (face.left(), face.top() - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.9, (0, 255, 0), 2)
```

```
# Convert the image back to RGB for displaying
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

# Display the image with landmarks and emotion classification
plt.imshow(image_rgb)
plt.title('Emotion Classification')
plt.axis('off')
plt.show()
```

Task 3: Gender Identification from Facial Features

Objective:
Identify individuals' genders based on facial features using traditional image processing and feature extraction techniques without using machine learning models.

Problem Statement: Gender Identification from Facial Features

The goal of this project is to develop a system for identifying the gender of individuals based on their facial features using traditional image processing and feature extraction techniques, without employing machine learning models. This involves loading a facial image dataset labeled with gender, detecting faces using Haar Cascades, and normalizing and cropping the facial regions for analysis.

The project will extract relevant facial features through geometric-based methods, such as calculating distances between key landmarks (eyes, nose, mouth, and jawline), and texture-based methods like Local Binary Patterns (LBP) and edge detection. A set of predefined rules will be formulated to classify gender based on these extracted features, leveraging characteristics such as jawline width and texture patterns.

The expected outcome is a reliable identification of gender for each image in the dataset, along with a detailed explanation of how the extracted features contributed to the classification decision. This project aims to enhance the understanding of facial feature analysis and its application in gender classification.

Expected Output

1. **Gender Classification Results:**
    - A list or table displaying the identified gender (male or female) for each image in the dataset. Each entry should be accompanied by the corresponding image filename or identifier.
2. **Visual Representation of Extracted Features:**
    - For each image, provide visualizations of the extracted facial features, including:
        - Key facial landmarks (e.g., eyes, nose, mouth, jawline) marked on the image.
        - Graphical representation of geometric measurements (e.g., distances between landmarks) used for classification.
3. **Feature Analysis:**
    - A summary of the extracted geometric and texture features for each image, including:
        - Numerical values for the distances between facial landmarks (e.g., jawline width, distance between eyes).
        - Texture descriptors obtained from methods like Local Binary Patterns (LBP) or edge detection.
4. **Classification Justification:**

- A brief explanation of how the identified features contributed to the classification decision for each image, including:
    - Specific rules applied (e.g., thresholds for jawline width or texture characteristics) that led to the final gender identification.
    - Discussion of any ambiguous cases or exceptions observed during classification.

5. **Overall Performance Metrics (if applicable):**
    - If a ground truth (true gender labels) is available, provide evaluation metrics such as accuracy, precision, recall, and F1 score to assess the performance of the gender identification system.
6. **Visual Summary:**
    - A presentation of the results in a visually appealing format, such as graphs or charts, to summarize the overall findings of the gender classification task. This could include pie charts showing the distribution of identified genders or bar charts comparing feature characteristics across genders.

Methodology for Gender Identification from Facial Features

The methodology for identifying gender based on facial features using traditional image processing techniques can be broken down into several key steps:

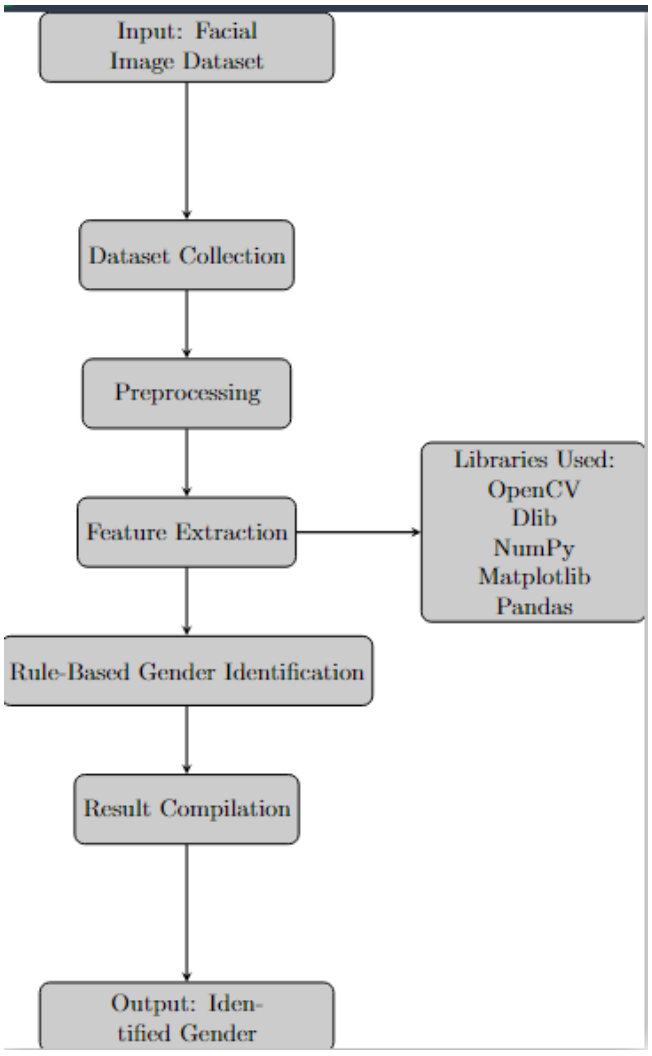
1. **Dataset Collection:**
    - **Acquire a Facial Image Dataset:** Obtain a labeled dataset containing facial images with associated gender labels (male and female). Ensure diversity in age, ethnicity, and lighting conditions to enhance the robustness of the analysis.
2. **Preprocessing:**
    - **Face Detection:**
        - Implement Haar Cascade classifiers to detect faces in the images. This step will help isolate facial regions from the rest of the image.
    - **Normalization and Cropping:**
        - Normalize detected facial images to a consistent size (e.g., 224x224 pixels) to ensure uniformity across the dataset.
        - Crop the images to focus on the facial area, removing any extraneous background elements.
3. **Feature Extraction:**
    - **Geometric-Based Feature Extraction:**

        - Identify key facial landmarks such as the eyes, nose, mouth, and jawline using methods like facial landmark detection (e.g., Dlib or OpenCV).
        - Calculate geometric features such as:
            - Distances between key landmarks (e.g., jawline width, distance between the eyes, nose length).
            - Ratios of distances to capture proportional relationships between facial features.
    - **Texture-Based Feature Extraction:**
        - Apply Local Binary Patterns (LBP) to capture local texture information from the facial images.
        - Utilize edge detection techniques (e.g., Sobel or Canny) to highlight prominent features and textures on the face.
        - Construct a feature vector combining both geometric and texture features for further analysis.
4. **Rule-Based Gender Identification:**
    - **Define Classification Rules:**
        - Establish a set of predefined rules based on extracted features. For instance:
            - Classify gender based on jawline width: a broader jawline may indicate a male, while a narrower jawline may indicate a female.
            - Use the average texture values from LBP or edge detection results to differentiate between male and female facial textures.
    - **Implement the Classification Logic:**
        - For each image, apply the rules to the extracted features to determine the gender classification. This may involve setting thresholds for specific measurements (e.g., jawline width) and comparing them against predefined criteria.
5. **Result Compilation:**
    - **Output the Identified Gender:**
        - Compile a list or table showing the identified gender for each image along with the corresponding extracted features.
    - **Visual Representation:**

- Provide visualizations, including marked images highlighting the key landmarks and graphical representations of the geometric features used in classification.

6. **Analysis and Interpretation:**
    - **Feature Analysis:**
        - Analyze the extracted features to determine the effectiveness of the classification rules. Identify which features had the most significant impact on the gender classification results.

BLOCK DIAGRAM:



PYTHON IMPLEMENTATON:

```python
import cv2
import dlib
import numpy as np

# Load the pre-trained Haar Cascade Classifier for face detection
face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade_frontalface_default.xml')

# Load the pre-trained Dlib shape predictor for facial landmarks
detector = dlib.get_frontal_face_detector()
# Update this path to the location where you saved the shape_predictor_68_face_landmarks.dat file
predictor = dlib.shape_predictor('path/to/shape_predictor_68_face_landmarks.dat')

# Function to extract geometric features
def extract_geometric_features(face_image, landmarks):
    # Get the coordinates of specific landmarks
    jawline = landmarks[0:17]  # Jawline landmarks
    left_eye = landmarks[36:42]  # Left eye landmarks
    right_eye = landmarks[42:48]  # Right eye landmarks
    nose = landmarks[30]  # Nose tip
    mouth = landmarks[48:60]  # Mouth landmarks

    # Calculate geometric features
    jawline_width = np.linalg.norm(jawline[0] - jawline[16])  # Width of jawline
    eye_distance = np.linalg.norm(left_eye.mean(axis=0) - right_eye.mean(axis=0))  # Distance between eyes
    nose_length = np.linalg.norm(nose - left_eye.mean(axis=0))  # Length of the nose

    return jawline_width, eye_distance, nose_length

# Function to classify gender based on extracted features
def classify_gender(jawline_width, eye_distance):
    if jawline_width > 15:  # Example threshold for jawline width
        return 'Male'
    elif eye_distance < 50:  # Example threshold for eye distance
```

```
        return 'Female'
    else:
        return 'Unknown'

# Function to detect faces and classify gender
def detect_and_classify_gender(image_path):
    # Load and convert the image to grayscale
    image = cv2.imread(image_path)
    gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # Detect faces in the image
    faces = face_cascade.detectMultiScale(gray_image, scaleFactor=1.1, minNeighbors=5)

    # Process each detected face
    for (x, y, w, h) in faces:
        # Extract the region of interest (face)
        face_roi = gray_image[y:y+h, x:x+w]

        # Detect landmarks using Dlib
        dlib_rect = dlib.rectangle(x, y, x+w, y+h)
        landmarks = predictor(gray_image, dlib_rect)

        # Convert landmarks to a numpy array
        landmarks_points = np.array([(p.x, p.y) for p in landmarks.parts()])

        # Extract geometric features
        jawline_width, eye_distance, nose_length = extract_geometric_features(face_roi, landmarks_points)

        # Classify gender based on extracted features
        gender = classify_gender(jawline_width, eye_distance)

        # Draw rectangle around the face and put text for gender
        cv2.rectangle(image, (x, y), (x+w, y+h), (255, 0, 0), 2)
        cv2.putText(image, gender, (x, y - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.9, (255, 0, 0), 2)
```

```
    faces = face_cascade.detectMultiScale(gray_image, scaleFactor=1.1, minNeighbors=5)

    # Process each detected face
    for (x, y, w, h) in faces:
        # Extract the region of interest (face)
        face_roi = gray_image[y:y+h, x:x+w]

        # Detect landmarks using Dlib
        dlib_rect = dlib.rectangle(x, y, x+w, y+h)
        landmarks = predictor(gray_image, dlib_rect)

        # Convert landmarks to a numpy array
        landmarks_points = np.array([(p.x, p.y) for p in landmarks.parts()])

        # Extract geometric features
        jawline_width, eye_distance, nose_length = extract_geometric_features(face_roi, landmarks_points)

        # Classify gender based on extracted features
        gender = classify_gender(jawline_width, eye_distance)

        # Draw rectangle around the face and put text for gender
        cv2.rectangle(image, (x, y), (x+w, y+h), (255, 0, 0), 2)
        cv2.putText(image, gender, (x, y - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.9, (255, 0, 0), 2)

    # Show the output image
    cv2.imshow("Gender Identification", image)
    cv2.waitKey(0)
    cv2.destroyAllWindows()

# Test the implementation
image_path = "/content/child.jpg"  # Replace with your image path
detect_and_classify_gender(image_path)
```