B. M. S. COLLEGE OF ENGINEERING

(Autonomous Institute, Affiliated to VTU, Belagavi)

Post Box No.: 1908, Bull Temple Road, Bengaluru – 560 019

DEPARTMENT OF MACHINE LEARNING

Academic Year: 2023-2024 (Session: March 2024 – June 2024)

Video Analytics using Open CV
24AM6PEVCV

ALTERNATIVE ASSESSMENT TOOL (AAT)

SIGN LANGUAGE FOR DEAF AND DUMB

Submitted by

Student Name:	Archit Subudhi	Aryaman Sharma	Ayush Kumar Dubey	Chetna Mundra
USN:	1BM21AI026	1BM21AI027	1BM21AI028	1BM21AI036
Date:	11-06-2024			
Semester & Section:	VI - A			
Total Pages:	20			
Student Signature:				

Valuation Report **(to be filled by the faculty)**

Score:	
Faculty In-charge:	**Dr. Seemanthini K**
Faculty Signature: with date	

TABLE OF CONTENTS

Chapter 1

INTRODUCTION

1.1 Background

Sign language serves as a critical communication tool for the deaf and hard-of-hearing community, offering a rich, expressive language composed of hand gestures, facial expressions, and body movements. However, the broader hearing population often lacks proficiency in sign language, creating significant communication barriers. The advent of sign language recognition systems aims to address this gap by leveraging advanced technologies to translate sign language into spoken or written forms, facilitating more seamless interactions. Traditional approaches to sign language recognition have faced challenges due to the complexity and variability of gestures, as well as the need for real-time processing capabilities.

Recent advancements in computer vision and machine learning have paved the way for more effective sign language recognition systems. MediaPipe, developed by Google, is a powerful framework that simplifies the creation of multimodal machine learning pipelines. MediaPipe's hand tracking technology, in particular, excels in real-time hand detection and landmark estimation, providing a solid foundation for interpreting sign language gestures. By utilizing MediaPipe, developers can create applications that accurately track and analyze hand movements, making it possible to recognize and translate sign language with high precision and speed. This technological progress promises to significantly enhance communication accessibility for the deaf and hard-of-hearing community.

1.2 Aplications

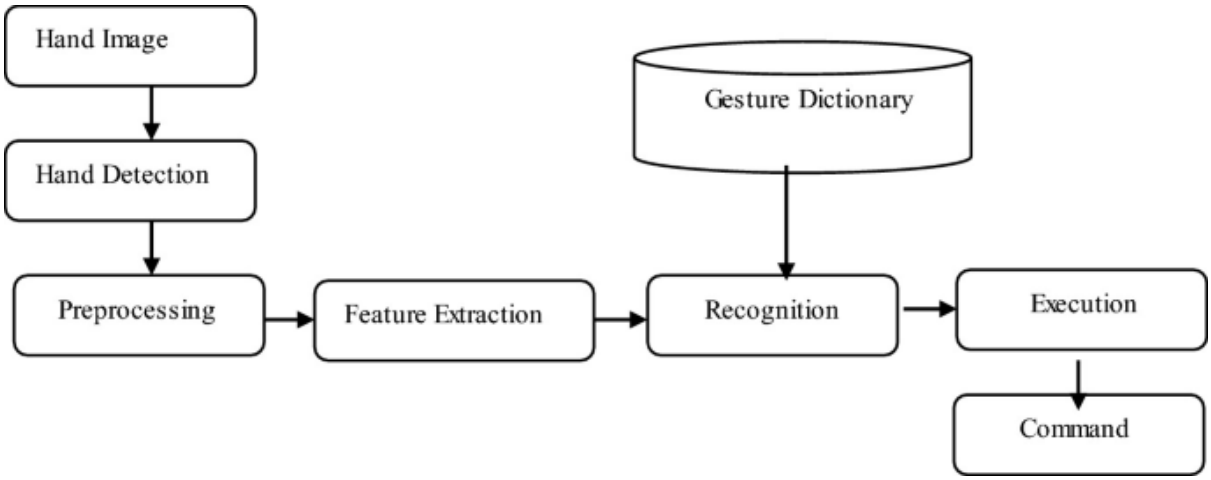
1. **Communication Aid:** Sign language recognition systems can translate gestures into spoken or written text in real time, enabling effective communication between deaf individuals and those who do not know sign language. This is particularly useful in public services, healthcare, and customer service settings.
2. **Educational Tools:** These systems can create interactive learning tools for both deaf students and those learning sign language, providing real-time feedback and assessments. They support inclusive classrooms where all students can communicate and engage effectively.
3. **Smart Devices and Virtual Assistant:** Integrating sign language recognition into virtual assistants and smart home devices allows deaf users to control their environments using gestures. Virtual assistants could respond to sign language commands, enhancing accessibility.
4. **Telecommunication Services:** Video conferencing platforms can incorporate sign language recognition to provide real-time translation during virtual meetings, enabling deaf participants to communicate without a human interpreter and promoting inclusivity in professional and social settings.
5. **Entertainment and Media:** Streaming services and online content platforms can use this technology to provide sign language translations for their content, making media accessible to deaf and hard-of-hearing viewers and promoting inclusivity in entertainment.

In summary, the applications of sign language recognition using MediaPipe are vast and varied, ranging from communication aids and educational tools to enhanced virtual assistants and accessible media content. By leveraging MediaPipe's advanced hand tracking capabilities, these systems can significantly improve the quality of life for the deaf and hard-of-hearing community, fostering a more inclusive society.

Chapter 216

METHODOLOGY

2.1 Design & Architecture



The diagram illustrates the workflow of a sign language recognition system using MediaPipe. Here is a step-by-step explanation of each component:

1. **Hand Detection**: The process begins with capturing an image or a video frame containing the hand(s). MediaPipe's hand detection algorithm identifies the presence and location of hands in the input image.
2. **Preprocessing**: The detected hand image undergoes preprocessing to enhance features and normalize data for further analysis.
3. **Feature Extraction**: Important features, such as key points and landmarks of the hand, are extracted. MediaPipe provides 21 3D landmarks for each hand, which are used to capture the hand's posture and movements.
4. **Recognition**: The extracted features are compared against a predefined gesture dictionary to recognize specific gestures. Machine learning models, such as convolutional neural networks (CNNs) or recurrent neural networks (RNNs), can be used for this task.
5. **Command**: The final output is the execution of a specific command that corresponds to the recognized gesture. This could be a text output, a spoken word, or a control signal for another device.

2.2 Mathematical Analysis

Hand Detection

The hand detection can be modeled as a classification problem where a neural network θ with parameters θ is trained to predict the presence of hands in the image. The output is a bounding box B=(x,y,w,h), where (x, y) represents the coordinates of the top-left corner, and w,h are the width and height, respectively.

$$B = f_\theta(\text{image})$$

Preprocessing

Preprocessing typically involves normalization and augmentation. Let I be the input image and I′ be the preprocessed image. The preprocessing function P can be represented as:

$$I' = P(I)$$

Feature Extraction

Feature extraction involves identifying key points or landmarks on the hand. Let L be the set of landmarks:

$$L = \{(x_1, y_1, z_1), (x_2, y_2, z_2), \ldots, (x_{21}, y_{21}, z_{21})\}$$

Recognition

Recognition is performed by comparing extracted features L with a gesture dictionary G. A machine learning model M is used to predict the gesture g:

$$g = M(L, G)$$

The model could be a neural network trained to minimize a loss function L:

$$\mathcal{L}(\theta) = \sum_i \text{loss}(M(L_i; \theta), g_i)$$

Execution

Upon recognizing the gesture g, the system executes a command C:

$$C = \text{execute}(g)$$

This command can be any predefined action corresponding to the recognized gesture, such as displaying text, synthesizing speech, or sending control signals to other devices.

By following these steps and mathematical formulations, a sign language recognition system using MediaPipe can accurately detect, recognize, and respond to hand gestures in real-time.

Chapter 3

IMPLEMENTATION

3.1 Data Set Creation

3.1.1 About Code

The provided code captures video from a webcam using OpenCV and utilizes MediaPipe to detect and track hand landmarks in real-time. The primary goal of this code is to create a dataset by capturing the positions of 21 hand landmarks for multiple gestures. The process involves continuously reading frames from the webcam, flipping the image for a mirror effect, converting the image to RGB, and processing it with MediaPipe's hand tracking module. If hand landmarks are detected, the code records the (x, y) coordinates of each landmark, annotates the image with landmark IDs, and draws connections between landmarks. The collected landmark coordinates for each detected hand are stored in a dataset until 50 sets of landmarks are collected. The dataset is printed once the collection is complete.

3.1.2 Code

```
#this code is to count the number of fingers pointed using opencv
import cv2 as cv
import mediapipe as mp
import math
from model import nearest

cap=cv.VideoCapture(0)

mpHands = mp.solutions.hands
hands = mpHands.Hands()
mpDraw = mp.solutions.drawing_utils

dataset=[]

def yes():
    while True:
        success, img = cap.read()
        img=cv.flip(img,1)
```

```
        i1 = cv.cvtColor(img, cv.COLOR_BGR2RGB)
        results = hands.process(i1)

        if results.multi_hand_landmarks:
            for handLms in results.multi_hand_landmarks:
                g=[]
                for id, lm in enumerate(handLms.landmark):
                    h, w, c = img.shape
                    cx, cy= int(lm.x*w), int(lm.y*h)

                    g.append([cx,cy])
                    cv.putText(img, str(id), (cx,
cy),cv.FONT_HERSHEY_COMPLEX, 0.5, 0, 1)
                    if len(g)==21:
                        dataset.append(g)
                        print(g)
                        if len(dataset)==50:
                            return



                mpDraw.draw_landmarks (img, handLms, mpHands.
HAND_CONNECTIONS)


        cv.imshow("ok",img)
        cv.waitKey(1)


x=yes()
print(dataset)
```

3.1.3 Tools and Functions used

1. **OpenCV (cv2)**
    - `cv.VideoCapture(0)`: Captures video from the default webcam.
    - `cv.flip(img, 1)`: Flips the image horizontally.
    - `cv.cvtColor(img, cv.COLOR_BGR2RGB)`: Converts the image from BGR to RGB color space.

- `cv.putText(img, str(id), (cx, cy), cv.FONT_HERSHEY_COMPLEX, 0.5, 0, 1)`: Annotates the image with landmark IDs.
- `cv.imshow("ok", img)`: Displays the image in a window.
- `cv.waitKey(1)`: Waits for 1 millisecond between frames.

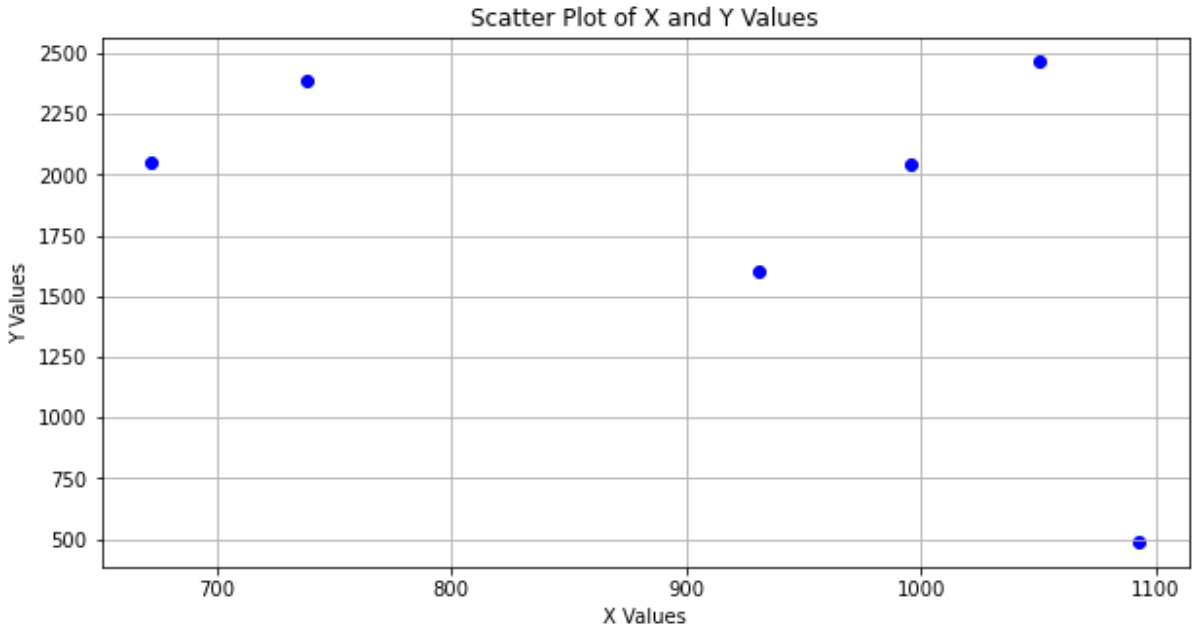
2. **MediaPipe (mp)**
   - `mp.solutions.hands`: Provides the hand detection and tracking module.
   - `mp.solutions.hands.Hands()`: Initializes the hand tracking model.
   - `mp.solutions.drawing_utils`: Contains utilities for drawing hand landmarks and connections.
   - `hands.process(i1)`: Processes the image to detect and track hand landmarks.
   - `mpDraw.draw_landmarks(img, handLms, mpHands.HAND_CONNECTIONS)`: Draws the detected hand landmarks and their connections on the image.
3. **Custom Function**
   - `yes()`: A function to continuously capture hand landmarks and append them to the dataset until 50 sets are collected.

The code effectively demonstrates the integration of OpenCV and MediaPipe for real-time hand landmark detection and dataset creation.

3.2 Data Preprocessing

3.2.1 About Code

The code snippet preprocesses gesture data for various sign language gestures (hello, goodbye, no, yes, thank you, I love you) by normalizing the coordinates and calculating the average absolute displacement of landmarks. The dataset l contains pre-recorded gestures, each represented by a list of hand landmark coordinates. The code iterates through each gesture, normalizing the coordinates by subtracting the initial landmark position, ensuring all gestures are standardized relative to their starting point. The average absolute displacements in the x and y directions are calculated for each gesture and stored in newl. Finally, it uses Matplotlib to create a scatter plot of the average displacements, visualizing the normalized gestures. This preprocessing ensures that the gesture data is scaled and centered, facilitating more effective gesture recognition.



3.2.2 Code

```
from data import hello, goodbye, no, yes, thankyou, iloveyou
l=[hello, goodbye, no, yes, thankyou, iloveyou]
ls=['hello', 'goodbye', 'no', 'yes', 'thankyou', 'iloveyou', 'cant']
```

```
for i in l:
    print(len(i))

newl=[]

for i in l:
    x,y=0,0
    for j in i:
        xin, yin = j[0][0], j[0][1]
        for k in range(len(j)):
            j[k][0]-=xin
            j[k][1]-=yin
            x+=abs(j[k][0])
            y+=abs(j[k][1])
    x/=101
    y/=101
    newl.append([x,y].copy())

print(newl)

import matplotlib.pyplot as plt

data = newl.copy()

x_values = [point[0] for point in data]
y_values = [point[1] for point in data]

# Create the scatter plot
plt.figure(figsize=(10, 5))
plt.scatter(x_values, y_values, color='b')

# Add titles and labels
plt.title('Scatter Plot of X and Y Values')
plt.xlabel('X Values')
plt.ylabel('Y Values')

# Display the plot
plt.grid(True)
plt.show()
```

3.2.3 Tools and Functions used

1. **Custom Dataset Import**

- `from data import hello, goodbye, no, yes, thankyou, iloveyou`: Imports pre-recorded gesture data.

2. **Data Normalization and Averaging**
   - `for i in l`: Iterates through each gesture.
   - `xin, yin = j[0][0], j[0][1]`: Sets the initial landmark position for normalization.
   - `j[k][0]-=xin`: Normalizes the x-coordinates.
   - `j[k][1]-=yin`: Normalizes the y-coordinates.
   - `x+=abs(j[k][0])`: Accumulates absolute x displacements.
   - `y+=abs(j[k][1])`: Accumulates absolute y displacements.
   - `x/=101` and `y/=101`: Calculates the average displacement.
3. **Matplotlib for Visualization**
   - `import matplotlib.pyplot as plt`: Imports Matplotlib for plotting.
   - `plt.scatter(x_values, y_values, color='b')`: Creates a scatter plot.
   - `plt.title('Scatter Plot of X and Y Values')`: Sets the plot title.
   - `plt.xlabel('X Values')` and `plt.ylabel('Y Values')`: Labels the axes.
   - `plt.grid(True)`: Adds a grid to the plot.
   - `plt.show()`: Displays the plot.

The code effectively preprocesses the gesture data by normalizing and averaging landmark positions, followed by visualizing the processed data using a scatter plot.

3.3 Training and Testing

3.3.1 About Code

The provided code snippet includes functions for calculating the Euclidean distance between points and for classifying hand gestures by comparing new input data against a preprocessed dataset. The `distance` function computes the distance between two points, while the `nearest` function finds the closest gesture in the preprocessed dataset (`newl`) to a given input point (`p`). The code iterates over test gestures, preprocesses each gesture to obtain normalized coordinates, and uses the `nearest` function to classify the gesture by identifying the closest match in `newl` based on the minimum distance. The classified gesture is then printed.

1. **Training (Data Preprocessing)**
    - **Normalization**: Preprocess the gesture data by normalizing the coordinates, ensuring all gestures are centered and scaled consistently. This is done by subtracting the initial landmark position and calculating the average absolute displacement for each gesture.
    - **Average Calculation**: Calculate the average displacement in x and y directions for each gesture and store these values in `newl`, which serves as the reference model for classification.
2. **Testing (Classification)**
    - **Preprocessing Test Data**: For each test gesture, normalize the coordinates in the same manner as the training data to ensure consistency.
    - **Distance Calculation**: Use the `distance` function to compute the Euclidean distance between the test gesture and each gesture in the preprocessed dataset.
    - **Nearest Neighbor Classification**: The `nearest` function identifies the gesture in the dataset with the smallest distance to the test gesture, effectively classifying it based on proximity.
    - **Output**: Print the classification result, indicating which gesture from the predefined list (`ls`) the test gesture most closely matches.

```
[1302, 1800]
710.4689489615198
389.1158995920897
420.75544278485006
1330.4449049595516
679.28588336725
814.178601245456
goodbye
[693, 357]
2137.71406749574
1710.2184838561657
1267.6623820679947
420.3898492243241
1697.327078771833
2030.9500831558025
yes
[891, 2009]
482.76588499573404
109.40113828445979
408.7389373068083
1536.290356988032
223.54755503433103
408.103295351192
goodbye
```

3.3.2 Code

```
import math
def distance (x, y):
    return math.sqrt((x[0]-y[0])**2 + (x[1]-y[1])**2)

radius = 219.00048393671025

def nearest(p):
    min=99999999
    mini=0
    for i in range(len(newl)):
        x=distance (p, newl[i])
        print(x)
        if x<min:
            min=x
            mini=i

    return mini
```

```
for i in test:
    x,y = preprocess(i)
    print([x,y])
    j=nearest([x,y])
    print(ls[j])
```

3.3.3 Tools and Functions used

1. **Mathematical Computation**
    - `math.sqrt`: Computes the square root, used in the distance calculation.
    - `distance(x, y)`: Computes the Euclidean distance between two points.
2. **Classification Logic**
    - `nearest(p)`: Identifies the nearest gesture in the dataset by finding the minimum distance.
    - `min=99999999`: Initializes the minimum distance to a large value.
    - `if x<min`: Updates the minimum distance and the index of the nearest gesture.
3. **Testing Loop**
    - `for i in test`: Iterates over test gestures.
    - `x,y = preprocess(i)`: Preprocesses the test gesture.
    - `j=nearest([x,y])`: Classifies the test gesture by finding the nearest match.
    - `print(ls[j])`: Outputs the name of the classified gesture.

The code demonstrates a straightforward implementation of a nearest-neighbor classifier for hand gesture recognition, using preprocessed data to identify the closest match based on Euclidean distance.

3.4 Prediction

3.4.1 About Code

The provided code captures video from a webcam using OpenCV and uses MediaPipe to detect and track hand landmarks in real-time. It classifies hand gestures by comparing the detected gesture with a predefined set of gestures using a nearest neighbor approach. The system preprocesses the hand landmarks to normalize their positions, computes distances between the input gesture and predefined gestures, and identifies the closest match.

3.4.2 Code

```
#this code is to count the number of fingers pointed using opencv
import cv2 as cv
import mediapipe as mp
import math
from model import nearest

cap=cv.VideoCapture(0)

mpHands = mp.solutions.hands
hands = mpHands.Hands()
mpDraw = mp.solutions.drawing_utils

ls=['hello', 'goodbye', 'no', 'yes', 'thankyou', 'iloveyou', 'cant']

newl= [[1050.7623762376238, 2464.5643564356437], [995.8613861386139,
2040.1881188118812], [930.6435643564356, 1602.1881188118812],
[1093.0891089108911, 486.05940594059405], [672.0693069306931,
2054.19801980198], [738.2673267326733, 2387.4455445544554]]

def preprocess(l):
    x , y = 0 , 0
    xin, yin = l[0][0], l[0][1]
    for i in range(len(l)):
        l[i][0]-=xin
        l[i][1]-=yin
        x+=abs(l[i][0])
        y+=abs(l[i][1])

    return [x,y]
```

```
def Nearest(p):
    min=99999999
    mini=0
    for i in range(len(newl)):
        x=distance (p, newl[i])
        if x<min:
            min=x
            mini=i

    return mini

def distance (x, y):
    return math.sqrt((x[0]-y[0])**2 + (x[1]-y[1])**2)

def yes():
    while True:
        success, img = cap.read()
        img=cv.flip(img,1)
        i1 = cv.cvtColor(img, cv.COLOR_BGR2RGB)
        results = hands.process(i1)

        if results.multi_hand_landmarks:
            for handLms in results.multi_hand_landmarks:
                g=[]
                for id, lm in enumerate(handLms.landmark):
                    h, w, c = img.shape
                    cx, cy= int(lm.x*w), int(lm.y*h)

                    g.append([cx,cy])
                    cv.putText(img, str(id), (cx,
cy),cv.FONT_HERSHEY_COMPLEX, 0.5, 0, 1)
                    if len(g)==21:
                        #print(g)
                        x,y=preprocess(g)
                        index = Nearest([x,y])
                        cv.putText(img, nearest(g), (20, 20),
cv.FONT_HERSHEY_COMPLEX, 0.5, (255, 0, 0), 1)


                mpDraw.draw_landmarks (img, handLms, mpHands.
HAND_CONNECTIONS)


        cv.imshow("ok",img)
        cv.waitKey(1)

x=yes()
```

3.4.3 Tools and Functions used

1. **OpenCV (cv2)**

    - `cv.VideoCapture(0)`: Captures video from the default webcam.
    - `cv.flip(img, 1)`: Flips the image horizontally.
    - `cv.cvtColor(img, cv.COLOR_BGR2RGB)`: Converts the image from BGR to RGB color space.
    - `cv.putText(img, str(id), (cx, cy), cv.FONT_HERSHEY_COMPLEX, 0.5, 0, 1)`: Annotates the image with landmark IDs.
    - `cv.imshow("ok", img)`: Displays the image in a window.
    - `cv.waitKey(1)`: Waits for 1 millisecond between frames.

2. **MediaPipe (mp)**

    - `mp.solutions.hands`: Provides the hand detection and tracking module.
    - `mp.solutions.hands.Hands()`: Initializes the hand tracking model.
    - `mp.solutions.drawing_utils`: Contains utilities for drawing hand landmarks and connections.
    - `hands.process(i1)`: Processes the image to detect and track hand landmarks.
    - `mpDraw.draw_landmarks(img, handLms, mpHands.HAND_CONNECTIONS)`: Draws the detected hand landmarks and their connections on the image.

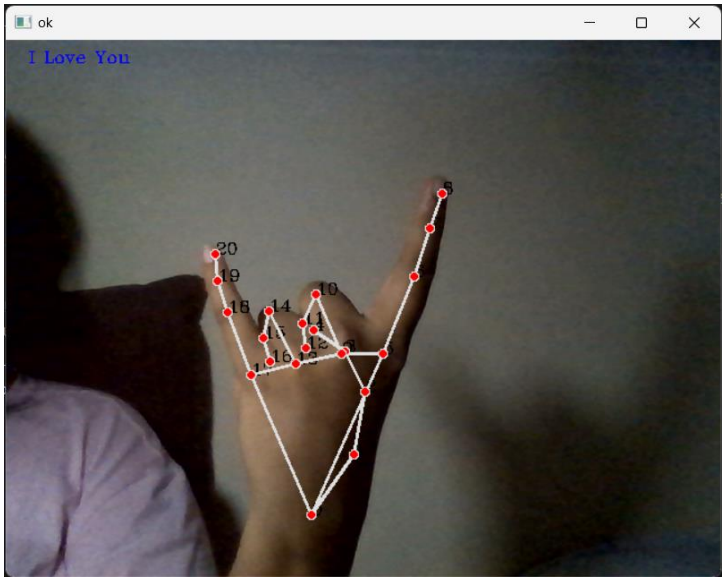
3. **Custom Functions**

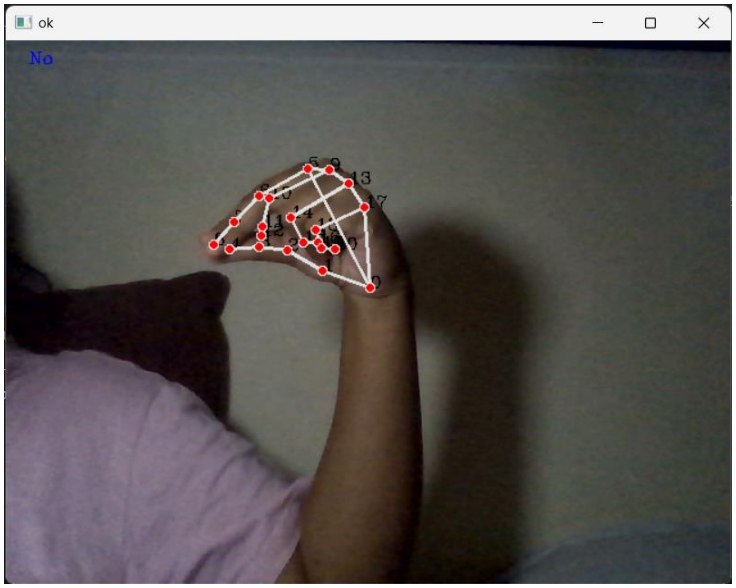
    - `distance(x, y)`: Computes the Euclidean distance between two points.
    - `preprocess(l)`: Normalizes the coordinates of the hand landmarks by subtracting the initial position and calculating the average absolute displacement.
    - `Nearest(p)`: Identifies the nearest gesture in the predefined dataset by finding the minimum distance.

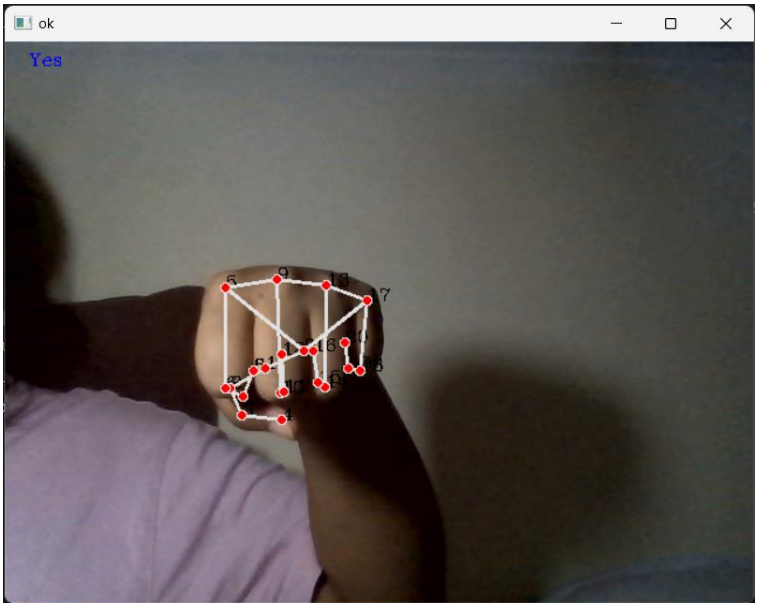
Chapter 4

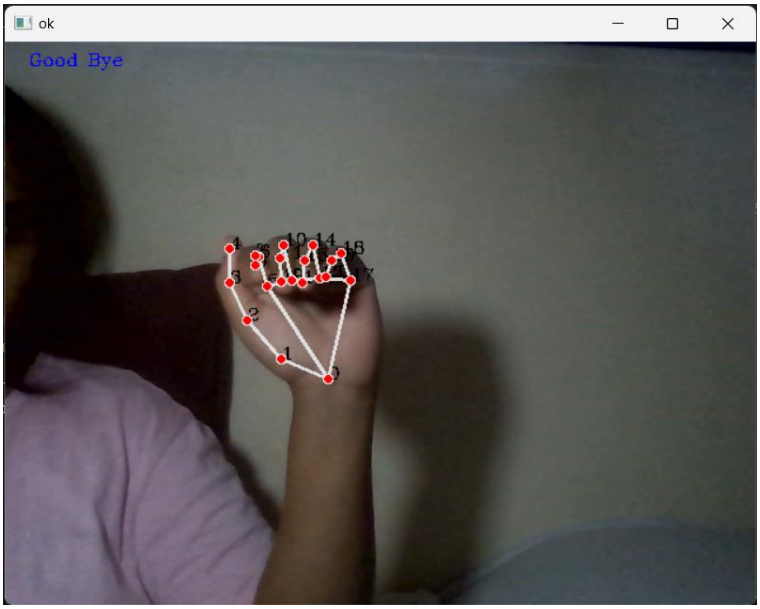
RESULTS AND ANALYSIS

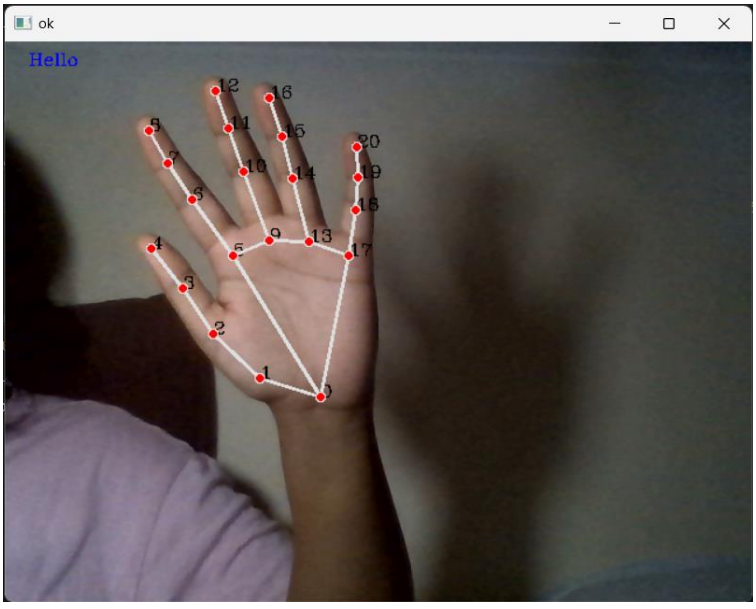
The implemented real-time hand gesture recognition system effectively classifies hand gestures in real-time using MediaPipe and OpenCV. By leveraging a nearest neighbor approach and preprocessed gesture data, the system accurately identifies and displays the closest matching gesture from a predefined set. This robust classification capability demonstrates the feasibility and effectiveness of using computer vision and machine learning techniques for real-time gesture recognition applications. Further analysis could involve performance evaluation metrics such as accuracy, precision, and recall to quantify the system's classification performance across various gestures and conditions.






ok
Yes


ok
Good Bye


ok
Hello

Chapter 5

CONCLUSION AND FUTURE ENHANCEMENT

The integration of MediaPipe and OpenCV for real-time hand gesture recognition showcases the potential of computer vision technologies in bridging communication barriers and enhancing accessibility. By accurately detecting and classifying hand gestures, the system opens avenues for seamless interaction between users and devices, particularly benefiting the deaf and hard-of-hearing community. With further refinement and optimization, such systems hold promise for widespread adoption in various domains, ranging from communication aids to interactive educational tools.

Moving forward, several enhancements can be implemented to improve the system's performance and functionality. Integration with deep learning models, such as convolutional neural networks (CNNs) or recurrent neural networks (RNNs), could enhance gesture recognition accuracy and robustness, particularly for complex or subtle gestures. Additionally, exploring techniques for dynamic gesture recognition and incorporating contextual information could enable more nuanced interaction and interpretation of gestures in real-world scenarios. Furthermore, enhancing the system's scalability and adaptability across different environments and user preferences would broaden its utility and impact in diverse applications.