UNIVERSITY OF PIRAEUS

School of Information and Communication Technologies

Department of Informatics

Thesis

Thesis Title: Τίτλος Διατριβής:	Sign Language Recognition System Using MediaPipe and Random Forest Σύστημα Αναγνώρισης Νοηματικής Γλώσσας με Χρήση του MediaPipe και του Αλγορίθμου Random Forest
Student's name-surname:	MOHAMED NEMRI
Father's name:	MUNIR
Student's ID No:	**Π19201**
Supervisor:	DIONISIOS SOTIROPOULOS

January 2025/ Ιανουάριος 2025

1 Copyright ©

Table of Contents

1. Abstract

Η νοηματική γλώσσα αποτελεί ένα ζωτικής σημασίας εργαλείο επικοινωνίας για άτομα με προβλήματα ακοής ή ομιλίας. Ωστόσο, η έλλειψη ευρείας κατανόησης της νοηματικής γλώσσας συχνά δημιουργεί ένα κενό στην επικοινωνία. Η παρούσα έρευνα επικεντρώνεται στην ανάπτυξη ενός συστήματος αναγνώρισης νοηματικής γλώσσας σε πραγματικό χρόνο, χρησιμοποιώντας το MediaPipe για την εξαγωγή χαρακτηριστικών σημείων (landmarks) και τον αλγόριθμο Random Forest για την αναγνώριση χειρονομιών. Ο πρωταρχικός στόχος είναι να γεφυρωθεί το χάσμα επικοινωνίας μέσω της αυτόματης ερμηνείας της νοηματικής γλώσσας.

Το έργο αυτό είναι δομημένο σε τέσσερα βασικά στάδια: συλλογή δεδομένων, δημιουργία συνόλου δεδομένων, εκπαίδευση του μοντέλου και δοκιμή/συμπεράσματα. Τα δεδομένα για τη νοηματική γλώσσα συλλέχθηκαν χειροκίνητα μέσω μιας κάμερας web, καταγράφοντας διάφορες χειρονομίες όπως "γειά," "ευχαριστώ" και "αυτοκίνητο." Το MediaPipe χρησιμοποιήθηκε για την εξαγωγή 21 χαρακτηριστικών σημείων από τις εικόνες, μειώνοντας τη διαστατικότητα και εστιάζοντας σε βασικά χαρακτηριστικά για την ταξινόμηση. Τα επεξεργασμένα δεδομένα αποθηκεύτηκαν ως αριθμητικές συντεταγμένες σε συνδυασμό με τις αντίστοιχες ετικέτες. Ο αλγόριθμος Random Forest επιλέχθηκε λόγω της ερμηνευσιμότητας, της ανθεκτικότητας και της υψηλής αποδοτικότητάς του σε μικρού έως μεσαίου μεγέθους σύνολα δεδομένων.

Το σύστημα παρουσίασε ενθαρρυντικά αποτελέσματα, με ακρίβεια που ξεπερνά το 90% στο σύνολο δοκιμών, αποδεικνύοντας την προοπτική του για εφαρμογές στον πραγματικό κόσμο. Οι δοκιμές σε πραγματικό χρόνο επιβεβαίωσαν την ικανότητα του μοντέλου να προβλέπει χειρονομίες με ακρίβεια, βασιζόμενο σε ζωντανή βιντεοσκόπηση. Παρά τις προκλήσεις, όπως η ποικιλομορφία των χειρονομιών και οι συνθήκες φωτισμού, η μελέτη αυτή παρέχει μια βάση για μελλοντικές βελτιώσεις, όπως η αναγνώριση δυναμικών χειρονομιών και η ενσωμάτωση επιπλέον χαρακτηριστικών του σώματος.

Η έρευνα υπογραμμίζει τη σημασία των εξατομικευμένων συνόλων δεδομένων, των ανθεκτικών μοντέλων μηχανικής μάθησης και των συστημάτων εξαγωγής συμπερασμάτων σε πραγματικό χρόνο για την αντιμετώπιση ζητημάτων προσβασιμότητας. Μελλοντικές εργασίες στοχεύουν στην επέκταση του συνόλου

δεδομένων, στην ενσωμάτωση χρονικών δεδομένων για δυναμικές χειρονομίες και στην ανάπτυξη του συστήματος σε φορητές συσκευές για αυξημένη χρηστικότητα

Sign language is a vital communication tool for individuals with hearing or speech impairments. However, the lack of widespread understanding of sign language often creates a communication gap. This research focuses on developing a **real-time sign language recognition system** using **MediaPipe** for landmark extraction and a **Random Forest classifier** for gesture recognition. The primary objective is to bridge the communication barrier by enabling automatic sign language interpretation.

This project is structured into four key stages: **data collection, dataset creation, model training, and testing/inference**. Custom sign language data was collected manually using a webcam, capturing various gestures such as "hello," "thanks," and "car." MediaPipe was utilized to extract 21 hand landmarks from the images, reducing dimensionality and focusing on key features for classification. The processed data was stored as numerical coordinates paired with corresponding labels. A Random Forest algorithm was chosen due to its interpretability, robustness, and high efficiency for small to medium-sized datasets.

The system achieved promising results, with an accuracy of over **90%** on the test dataset, demonstrating its potential for real-world applications. Real-time testing confirmed the model's ability to predict gestures accurately based on live video input. While challenges such as variability in hand gestures and lighting conditions remain, this study provides a foundation for future enhancements, including dynamic gesture recognition and integration of additional body landmarks.

This research emphasizes the importance of **customized datasets**, **robust machine learning models**, and **real-time inference systems** in addressing accessibility issues. Future work aims to expand the dataset, incorporate temporal data for dynamic gestures, and deploy the system on portable devices for enhanced usability.

2. Introduction

Background and Motivation

Communication is a fundamental aspect of human interaction, enabling individuals to express thoughts, emotions, and ideas. However, for those who are part of the Deaf and Hard of Hearing (DHH) community, this process can present significant challenges, particularly in environments where sign language is not widely understood. Observing these challenges up close, I was deeply inspired to undertake this project.

My cousin, who is Deaf, has faced numerous struggles in communicating with others who are unfamiliar with sign language. Witnessing her frustration when simple conversations became barriers sparked my determination to create a system that could bridge the gap between sign language users and the broader community. This personal connection provided a powerful motivation to pursue the development of a **Sign Language Recognition System**, an application aimed at enhancing accessibility and understanding.

Objectives of the Project

The primary objectives of this project are as follows:

1. **To develop a real-time sign language recognition system** using machine learning and computer vision technologies, specifically leveraging the Random Forest algorithm for gesture classification.
2. **To facilitate communication between sign language users and non-signers** by converting hand gestures into meaningful text representations.
3. **To contribute to the accessibility of sign language technologies** by creating a modular system that can be easily expanded to include more gestures.
4. **To provide a practical demonstration of artificial intelligence's potential** to solve real-world accessibility problems.

Significance of the Study

The significance of this study lies in its potential to impact lives in the Deaf and Hard of Hearing community. By cutting communication gaps, the project aims to:

- **Enhance inclusivity:** Enable sign language users to interact more effectively in environments where their primary language is not understood.
- **Raise awareness:** Highlight the importance of sign language as a tool for communication and promote the adoption of accessible technologies.
- **Contribute to technology for good:** Demonstrate the potential of machine learning and computer vision to address social challenges.
- **Lay a foundation for further research:** Provide a framework for future advancements in sign language recognition systems, encouraging the development of more sophisticated and inclusive solutions.

In essence, this project is not merely a technical endeavor but a personal mission to create tools that empower individuals like my cousin. By developing this system, I hope to bring us one step closer to a world where communication barriers are minimized, and inclusivity becomes the norm.

3. Relevant Literature Review

Overview of Existing Sign Language Recognition Systems

Sign language recognition systems have been a subject of research for decades, driven by the need to improve accessibility for the Deaf and Hard of Hearing (DHH) community. Many of these systems rely on vision-based techniques to interpret hand gestures and convert them into text or speech. For instance, early studies focused on static image recognition, where gestures were captured as still images and classified based on shape and orientation. However, these systems lacked the ability to process dynamic gestures, which are essential for recognizing continuous sign language.

Recent advancements in computer vision and deep learning have enabled the development of more sophisticated systems. The use of frameworks such as MediaPipe by Google has revolutionized gesture recognition by providing robust real-time hand landmark detection. These frameworks facilitate the extraction of features such as hand positions, joint angles, and movement trajectories, which are crucial for accurate recognition. Despite these advancements, existing systems often face limitations in scalability, adaptability to regional sign language variations, and real-time performance.

Machine Learning Algorithms in Gesture Recognition

Machine learning plays a pivotal role in sign language recognition, offering the ability to classify gestures based on extracted features. Various algorithms have been explored in the field:

- **Support Vector Machines (SVM):** SVMs were among the first machine learning techniques used for gesture classification due to their effectiveness in binary and multi-class classification tasks. However, their performance deteriorates with large datasets and high-dimensional features.
- **K-Nearest Neighbors (KNN):** This algorithm is simple and effective for small datasets but struggles with scalability and real-time applications.
- **Neural Networks (NN):** Deep learning models such as Convolutional Neural Networks (CNNs) have demonstrated state-of-the-art performance in gesture recognition. They excel at feature extraction and classification but require substantial computational resources and large datasets for training.
- **Random Forest (RF):** Random Forest has gained attention for its balance of simplicity, accuracy, and robustness. It performs well on small-to-medium-sized datasets and offers interpretability, making it suitable for sign language recognition projects like this one.

Comparative Studies on Classifiers (e.g., Random Forest vs Neural Networks)

Several comparative studies have evaluated the effectiveness of different classifiers in gesture recognition tasks. These studies typically compare factors such as accuracy, computational efficiency, and suitability for real-time applications:

- **Accuracy:** Neural Networks, particularly CNNs, often outperform other classifiers in terms of accuracy due to their ability to learn complex patterns. However, Random Forest achieves competitive accuracy with significantly lower computational requirements.
- **Efficiency:** Random Forest is computationally efficient and faster to train compared to Neural Networks, which require iterative optimization processes like backpropagation.
- **Interpretability:** Unlike Neural Networks, which are often referred to as "black box" models, Random Forest provides insights into feature importance, enabling researchers to understand the decision-making process.

- **Scalability:** Neural Networks excel with large datasets, while Random Forest is better suited for small-to-medium datasets, making it a practical choice for this project.

For this project, the Random Forest classifier was chosen due to its balance of accuracy, efficiency, and interpretability. While Neural Networks are ideal for large-scale, highly complex systems, Random Forest aligns with the project's objectives.

4. Sign Language Recognition

History and Importance of Sign Language

Sign language has been a fundamental mode of communication for the Deaf and Hard of Hearing (DHH) community for centuries. Its origins date back to the 17th century, with documented signs used in education and daily communication. Over time, various regional sign languages have evolved, each with unique grammar, syntax, and vocabulary. For example, American Sign Language (ASL), British Sign Language (BSL), and Greek Sign Language (GSL) reflect cultural differences in their structure and expression.

Sign language is not merely a replacement for spoken language; it is a full-fledged linguistic system. It conveys complex ideas, emotions, and narratives through hand gestures, facial expressions, and body language. Despite its rich history and significance, sign language remains underrepresented in mainstream technologies, leading to communication barriers between the DHH community and the hearing population. Bridging this gap is critical to promoting inclusivity and accessibility.

Challenges in Automatic Recognition

Developing systems for automatic sign language recognition poses several challenges:

1. **Gesture Complexity:**
    - Sign language gestures are intricate and involve not only hand movements but also facial expressions and body posture.
    - Variations in speed, trajectory, and starting position of gestures add complexity to recognition.

2. **Individual Variability:**
    - Different users perform the same gesture differently, influenced by factors such as hand size, movement style, and signing fluency.
3. **Environmental Factors:**
    - Background noise, lighting conditions, and occlusion can degrade the performance of recognition systems.
4. **Real-Time Performance:**
    - For practical use, systems must process gestures in real-time without significant delays, requiring efficient algorithms and hardware optimization.
5. **Regional and Contextual Variations:**
    - Unlike spoken languages, sign languages are not universally standardized. A system trained on one regional sign language may not work well for another.
6. **Lack of Comprehensive Datasets:**
    - Building extensive datasets with diverse gestures, users, and contexts is resource-intensive but essential for robust recognition.

Applications in Real-World Scenarios

Automatic sign language recognition has transformative potential in various domains:

1. **Education:**
    - Tools can help educators bridge the gap with DHH students, enabling better inclusion in mainstream classrooms.
    - Interactive applications can aid in teaching sign language to hearing individuals.
2. **Healthcare:**
    - Sign language recognition can facilitate doctor-patient communication in hospitals, where interpreters may not always be available.
3. **Customer Service:**

- Companies can integrate sign language recognition into kiosks and chatbots, offering DHH-friendly customer service solutions.

4. **Media Accessibility:**
   - Automatic sign recognition can improve captioning and translation services, making media content accessible to the DHH community.
5. **Public Services:**
   - Government offices, airports, and other public facilities can use recognition systems to provide inclusive services.
6. **Personalized Assistance:**
   - Sign language-enabled virtual assistants can empower DHH users in managing daily tasks.
7. **Social Integration:**
   - By breaking communication barriers, these systems can foster stronger relationships between the DHH and hearing communities.

5. Data Collection and Preparation Process

Manual Data Collection

The foundation of any machine learning project is high-quality data, and for this project, the dataset was manually collected to ensure specificity and relevance to the task of recognizing sign language gestures. Using a webcam and a custom-built script (collect_img.py), we captured multiple images for each sign language gesture. The dataset encompasses 22 distinct words, such as "hello," "thanks," and "car," with 200 samples per word (100 samples per session “hand” across two sessions ”Left and right hand”). Each gesture was performed by a single user to maintain consistency in the initial phase of development.

The manual collection approach allowed me to directly control the quality of the data. This ensured that gestures were performed correctly and aligned with the model's requirements for subsequent processing.

Challenges in Data Collection

Several challenges arose during the data collection process, which needed to be addressed to create a robust dataset:

1. **Lighting Conditions:**
    - Variability in lighting caused shadows and reflections, which could affect landmark detection.
    - To mitigate this, we performed data collection in a well-lit environment with consistent lighting across sessions.
2. **Background Noise:**
    - Complex or cluttered backgrounds posed a risk of interfering with hand detection.
    - We used plain, neutral backgrounds to enhance the accuracy of the MediaPipe model during landmark extraction.
3. **User Variability:**
    - While gestures were performed by a single user in this phase, real-world usage would involve multiple users with varying hand sizes, movement styles, and speeds.
    - Future iterations of the project will address this by expanding the dataset to include diverse users.
4. **Gesture Complexity:**

- Some gestures involve two hands or rapid movements, making them harder to capture and process accurately.
- The recording process included multiple takes to ensure gesture clarity.

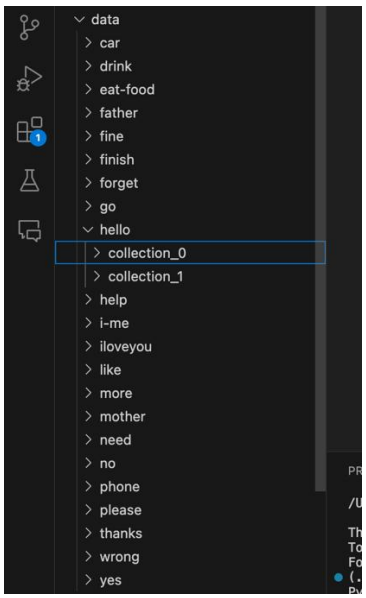
5. **Fatigue:**
   - Repeating gestures for multiple sessions caused fatigue, potentially impacting consistency. Breaks were taken to reduce variability caused by user fatigue.
6. **Similarity Between Gestures:**
   - Certain gestures, such as "phone" and "wrong," appeared visually similar, making them harder to distinguish.
   - Careful adjustments were made to the angle and positioning of the hands for each gesture to create distinct visual differences.
   - For example, the gesture for "phone" was performed with a more upright posture, while "wrong" was executed at a slightly tilted angle to minimize overlap.

Dataset Structure and Organization

The dataset was organized into a hierarchical directory structure for ease of access and processing. The structure followed this pattern:

```
/data
  /word_1
    /collection_0
      0.jpg
      1.jpg
      ...
    /collection_1
      0.jpg
      1.jpg
      ...
  /word_2
```

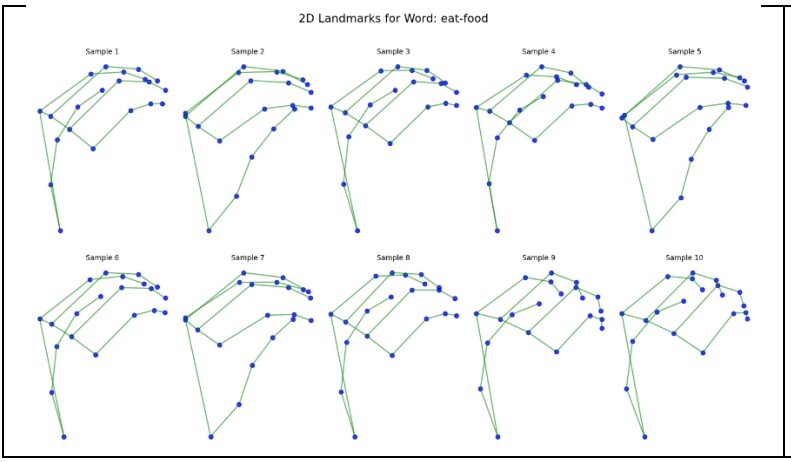
...

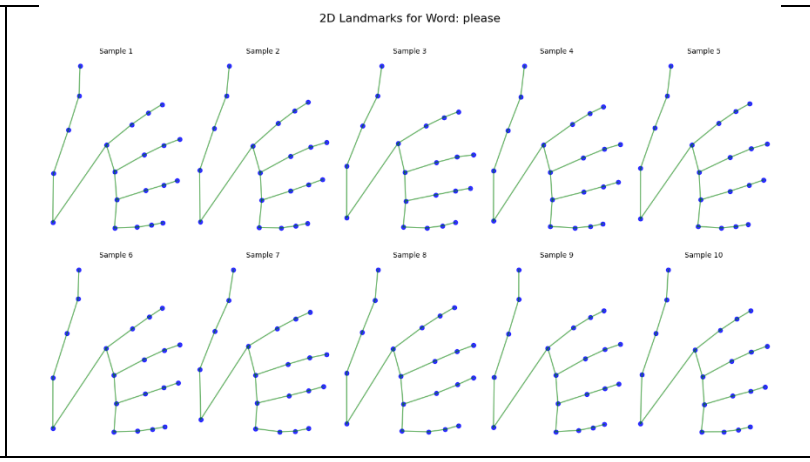


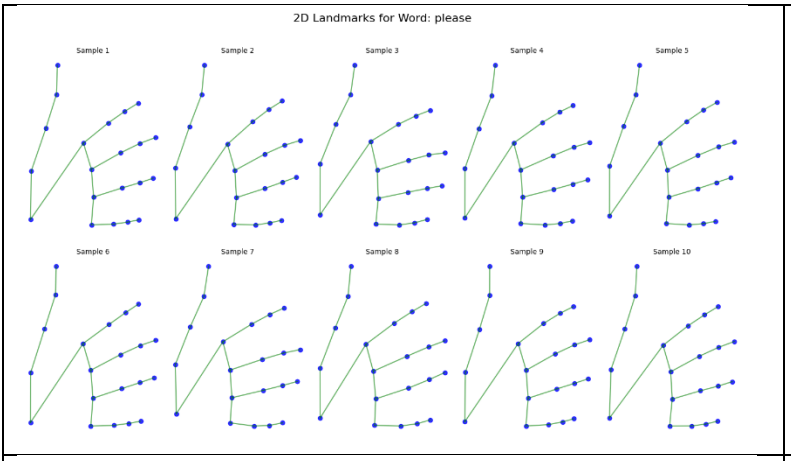
Each folder represents a specific word (e.g., "hello"), with subfolders for individual collection sessions. This structure ensures that the data is easily accessible for both training and validation purposes.

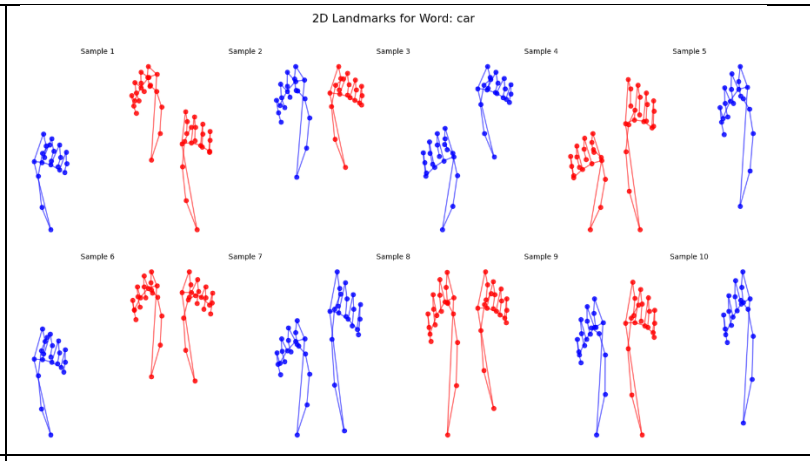
Landmark Extraction Using MediaPipe

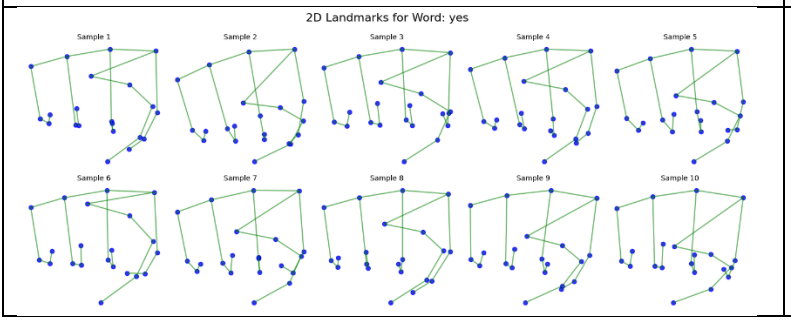
Once the images were collected, the next step was to extract meaningful features from them. MediaPipe, an advanced framework for machine learning pipelines, was used for this purpose. Specifically, the MediaPipe Hands module detected and extracted 21 key landmarks for each hand.











1. **Hand Detection:**
    - Each image was processed through MediaPipe to locate the hands within the frame. The model outputs normalized (x, y) coordinates for each landmark, representing positions relative to the image dimensions.
2. **Landmark Extraction Process:**
    - For single-hand gestures, 21 landmarks were extracted.
    - For dual-hand gestures, 42 landmarks (21 for each hand) were extracted.
    - These landmarks were then flattened into a single array for each image.
3. **Data Formatting:**
    - Extracted landmarks were saved in the form of arrays, each containing 42 values (21 points × 2 dimensions). This ensures consistent input size for the machine learning model.
    - If no hands were detected in an image, placeholder zeros were added to maintain consistent array size.
4. **Advantages of Using Landmarks:**

- **Dimensionality Reduction:** Landmarks significantly reduce the amount of data compared to using raw images, making the training process more efficient.
- **Invariance to Scale and Rotation:** Normalized coordinates ensure that the model remains robust to variations in hand size and orientation.
- **Improved Generalisation:** By focusing on the relative positions of landmarks rather than raw pixel data, the model learns to generalize better across different scenarios.

---

6. Mathematical Background of Random Forest Algorithm

Decision Tree Basics

A Random Forest is built upon the foundation of decision trees. To understand Random Forests, it is essential to grasp the mechanics of decision trees:

- **What is a Decision Tree?**
  - A decision tree is a flowchart-like structure used for classification or regression tasks. Each internal node represents a question about the data (a feature), each branch represents the outcome of the question, and each leaf node represents a final decision or prediction.
- **Example of a Decision Tree:** Consider a decision tree for recognizing a gesture:
  - **Node 1:** Is the x-coordinate of landmark 5 greater than 0.5?
    - Yes → Go to Node 2
    - No → Go to Node 3
  - **Node 2:** Is the y-coordinate of landmark 8 less than 0.3?
    - Yes → Gesture is "hello"
    - No → Gesture is "thanks"

The tree grows by repeatedly splitting the dataset based on the feature (e.g., x or y coordinates of landmarks) that best separates the classes. The goal is to make the data at each leaf node as homogeneous as possible (pure).

Gini Impurity and Information Gain

To decide the best feature for splitting the data at each node, measures like **Gini Impurity** or **Information Gain** are used.

- **Gini Impurity:**
  - Gini Impurity measures the probability of misclassifying a randomly chosen data point if it were assigned a class based on the distribution at the node.
  - Formula: $G = 1 - \sum(pi\text{^}2)$

    where pi is the probability of class i at a given node
  - **Example:** If a node contains 100 samples:
    - 40 belong to class "hello"
    - 60 belong to class "thanks"
    - phello=0.4p_{hello} = 0.4, pthanks=0.6p_{thanks} = 0.6

G=1−(0.42+0.62)=1−(0.16+0.36)=0.48G = 1 - (0.4^2 + 0.6^2) = 1 - (0.16 + 0.36) = 0.48

A lower Gini Impurity indicates a more "pure" node.

- **Information Gain:**
  - Information Gain measures the reduction in entropy (disorder) after splitting a dataset.
  - Formula: $\mathrm{IG} = H_{\text{parent}} - \sum_{\text{child}} (\frac{\text{samples in child}}{\text{samples in parent}} \times H_{\text{child}})$

    Where:

    H is the entropy of a node.

Entropy (H)

- The formula for entropy is: $H = -\sum_{i=1}^{C} p_i log_2(p_i)$

  Where:

  C is the number of classes.

pi is the proportion of samples belonging to class i in the node.

Both Gini Impurity and Information Gain help select the feature and threshold that most effectively separates the classes.

Random Forest Ensemble Technique

Random Forest extends the concept of decision trees by creating an ensemble of multiple trees and aggregating their predictions.

- **How It Works:**
    1. **Bootstrapping:**
        - Each tree is trained on a random subset (with replacement) of the dataset. This introduces diversity among the trees.
    2. **Random Feature Selection:**
        - At each split, a random subset of features is considered for splitting, further ensuring tree diversity.
    3. **Aggregation:**
        - For classification tasks, the final prediction is determined by majority voting among all trees in the forest.
        - For regression tasks, the average prediction from all trees is used.
- **Why It Works:**
    - By combining multiple diverse trees, Random Forest reduces overfitting and improves generalization. Each tree contributes to the overall prediction, but no single tree dominates.

Advantages of Random Forest for Gesture Recognition

Random Forest is particularly well-suited for gesture recognition tasks for the following reasons:

1. **Robustness to Noise:**

- Even if some trees make errors due to noisy data, the ensemble approach minimizes the impact of these errors.

2. **Feature Importance:**
   - Random Forest provides insights into the importance of each feature (e.g., which landmarks contribute most to recognizing gestures).
3. **Efficiency:**
   - Random Forest is computationally efficient for training on medium-sized datasets like ours.
4. **Flexibility:**
   - It works well with both numerical (landmark coordinates) and categorical data.
5. **Handling Multiclass Problems:**
   - Our dataset contains 22 classes (one for each word). Random Forest effectively handles this multiclass classification problem.

Example of Random Forest in Our Project

1. **Input Data:**
   - Each training sample is a flattened array of 42 numerical values (21 landmarks × 2 dimensions).
   - The label for each sample corresponds to a word (e.g., "hello," "thanks").
2. **Training Process:**
   - Trees are built by splitting the dataset based on landmark coordinates that maximize class separation (using Gini Impurity).
   - Example split: "Is the x-coordinate of landmark 5 > 0.5?"
3. **Prediction:**
   - During real-time inference, a flattened array of landmarks is passed to the forest.
   - Each tree independently predicts a class, and the final prediction is made based on majority voting.
4. **Output:**

- The system outputs the word corresponding to the predicted class (e.g., "thanks") along with visual feedback on the screen.

---

7. Experimental Setup

Hardware and Software Specifications

To implement the sign language recognition system, the following hardware and software were utilized:

- **Hardware:**
  - **Laptop:** Apple Macbook Pro (Mid 2015)
  - **Processor:** Intel Core i7 2.2 GHz
  - **RAM:** 16 GB
  - **Graphics Card:** AMD Radeon HD 6770M
  - **Storage:** 1 TB HDD
  - **Camera:** Built-in iMac webcam
- **Software:**
  - **Operating System:** macOS Monterey
  - **Development Environment:** Python 3.9 in a virtual environment
  - **Integrated Development Environment (IDE):** Visual Studio Code

---

Libraries and Tools Used

Several Python libraries and tools were employed to streamline the development process:

1. **OpenCV:**
   - **Purpose:** Real-time video capture, frame processing, and visualization.

- **Role:** Used for capturing video input from the webcam and displaying the live feed with annotations such as bounding boxes and predictions.

2. **MediaPipe:**
   - **Purpose:** Extract 21 hand landmarks per hand for gesture recognition.
   - **Role:** MediaPipe's pre-trained deep learning model detects and tracks hand keypoints.
3. **Scikit-Learn:**
   - **Purpose:** Training the Random Forest Classifier.
   - **Role:** Handles the classification task, model evaluation, and predictions.
4. **NumPy:**
   - **Purpose:** Efficient numerical operations.
   - **Role:** Processes and manipulates landmark arrays (padding, reshaping, etc.).
5. **Pickle:**
   - **Purpose:** Serialization and deserialization of Python objects.
   - **Role:** Saves the processed dataset (data.pickle) and trained model (model.p) for reuse.
6. **Other Tools:**
   - **Matplotlib/Seaborn:** Used for visualizing the confusion matrix during evaluation.
   - **Terminal/Command Prompt:** Runs Python scripts and monitors logs.

Step-by-Step Guide for Implementation

This section outlines the step-by-step procedure to replicate the project:

1. **Setting Up the Environment:**
   - Install Python 3.9 and create a virtual environment:
   - python3 -m venv .venv
   - source .venv/bin/activate
   - Install required libraries:

- pip install opencv-python mediapipe scikit-learn numpy matplotlib

2. **Data Collection:**
   - Use the collect_img.py script to manually collect gesture data for each word:
     - Run the script to capture 100 images per word in two separate sessions.
     - The images are stored in a structured directory format under ./data.
3. **Dataset Creation:**
   - Process the collected images using create_dataset.py:
     - MediaPipe extracts hand landmarks from each image.
     - The extracted landmarks and corresponding labels are saved in data.pickle.
4. **Model Training:**
   - Train a Random Forest Classifier using train_classifier.py:
     - The dataset is split into training and testing sets.
     - The model is trained to classify gestures based on the landmarks.
     - Accuracy is calculated, and the model is saved as model.p.
5. **Real-Time Inference:**
   - Perform real-time gesture recognition using inference_classifier.py:
     - The webcam captures live frames.
     - MediaPipe detects hand landmarks.
     - The trained model predicts the gesture, displaying the result on the video feed.
6. **Evaluation and Testing:**
   - Evaluate the model using the confusion matrix generated during training.
   - Test the system in varying conditions:
     - Different lighting setups
     - Background clutter

- Multiple users performing gestures

7. **Visualization:**
   - Visualize key performance metrics:
     - Accuracy
     - Confusion matrix
     - Real-time predictions on the live feed

---

8. Code Implementation and Explanation

This section presents a detailed breakdown of the implementation process for the Sign Language Recognition System, including all critical code components. The modular structure of the code ensures a clear flow from data collection to real-time inference.

---

1. collect_img.py – Data Collection

Purpose:
This script collects image data from a webcam to create a dataset for training a sign language recognition model. The images are labeled according to the word being performed.

Code Breakdown:

```
import os
import cv2
```

- **os:** Manages directories and file paths to save collected data.
- **cv2 (OpenCV):** Captures webcam input, displays the video feed, and overlays text/graphics.

```
DATA_DIR = './data'  # Path to store collected data
if not os.path.exists(DATA_DIR):
    os.makedirs(DATA_DIR)
```

- **Creates a directory** (data) to store images if it doesn't already exist.

```
words_to_collect = [
    'phone', 'please', 'thanks', 'wrong', 'yes', 'car', 'drink', 'eat-food',
```

```
    'father', 'fine', 'finish', 'forget', 'go', 'hello', 'help', 'i-me',
    'iloveyou', 'like', 'more', 'mother', 'need', 'no'
]
```

- **Defines the list of words** (signs) for which image data will be collected.

```
dataset_size = 100  # Number of images per collection
collections_per_word = 2  # Collect data twice for each word
```

- **100 images per word per session** (200 total images for each word).

```
cap = cv2.VideoCapture(0)
```

- Opens the webcam to begin capturing video.

Main Loop (Data Collection Per Word):

```
for word in words_to_collect:
    for collection_idx in range(collections_per_word):
        word_dir = os.path.join(DATA_DIR, word, f'collection_{collection_idx}')

        # Skip collection if data already exists
        if os.path.exists(word_dir) and len(os.listdir(word_dir)) >= dataset_size:
            print(f"Skipping collection for {word} (Collection {collection_idx + 1}), already exists.")
            continue
```

- **Loops through each word** and **checks if data already exists.** If the collection already has 100 images, it **skips to the next word.**

```
    if not os.path.exists(word_dir):
            os.makedirs(word_dir)
```

- **Creates directories** to store images for each word if they don't exist.

Get Ready Message (Before Capturing):

```
                    print(f'Collecting data for word: {word} (Collection {collection_idx + 1})')

        print('Press "Q" to start collecting...')

        # Wait for the user to get ready
        while True:
            ret, frame = cap.read()
            cv2.putText(frame, f'Get Ready: Collecting {word} (Collection {collection_idx + 1})',
                        (50, 50), cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 255, 0), 3)
            cv2.imshow('frame', frame)
```

Displays the **live video feed** with a "Get Ready" message prompting the user to prepare for data collection.

Start Collecting Images:

```
cv2.imshow('frame', frame)
        key = cv2.waitKey(25)
        if key == ord('q'):
            break
```

- **Waits for the user** to press 'Q' to start collecting images.

Image Collection Loop (100 Images):

```
                counter = 0

    while counter < dataset_size:
        ret, frame = cap.read()
        cv2.putText(frame, f'Collecting: {word} (Image {counter + 1}/{dataset_size})',
                    (50, 50), cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255, 0), 2)
        cv2.putText(frame, f'Label: {word}', (50, 100), cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 255, 255), 2)
```

Displays the **live video feed** while collecting and saves images with the label (word).

```
cv2.imwrite(os.path.join(word_dir, f'{counter}.jpg'), frame)
        counter += 1
```

- Saves each captured frame as an image (e.g., 1.jpg, 2.jpg).

Pause and Resume Option:

```
key = cv2.waitKey(25)
        if key == ord('q'):
            break
        elif key == ord('p'):  # Pause functionality
            print("Paused. Press 'r' to resume.")
            while True:
                key = cv2.waitKey(1)
                if key == ord('r'):  # Resume after pause
                    print("Resuming...")
                    break
```

- Allows the user to pause ('P') and resume ('R') during image collection.

Release Webcam:

```
cap.release()
cv2.destroyAllWindows()
```

- Closes the webcam and destroys OpenCV windows after data collection.

---

2. create_dataset.py – Dataset Creation

Purpose:
This script processes collected images to extract hand landmarks using **MediaPipe** and stores the data in data.pickle for model training.

Code Breakdown:

```
import os
import pickle
import cv2
import mediapipe as mp
import numpy as np
```

- **pickle:** Serializes the dataset to save and load later.
- **MediaPipe:** Detects and tracks hand landmarks.

```
mp_hands = mp.solutions.hands
mp_drawing = mp.solutions.drawing_utils

hands = mp_hands.Hands(static_image_mode=True, min_detection_confidence=0.3)
```

- MediaPipe leverages **CNNs (Convolutional Neural Networks)** to detect hand landmarks.
- A CNN predicts 21 hand keypoints by **regressing their x, y, and z coordinates** directly from the input image.

```
DATA_DIR = './data'
OUTPUT_FILE = 'data.pickle'

# All words: Existing + New
words = [
```

```
    'phone', 'please', 'thanks', 'wrong', 'yes', 'car', 'drink', 'eat-food',
    'father', 'fine', 'finish', 'forget', 'go', 'hello', 'help', 'i-me',
    'iloveyou', 'like', 'more', 'mother', 'need', 'no'
]
```

- Specifies the directory for collected data and defines the output file.

```
if os.path.exists(OUTPUT_FILE):
    # Load existing dataset if it exists
    with open(OUTPUT_FILE, 'rb') as f:
        dataset = pickle.load(f)
    data = dataset['data']
    labels = dataset['labels']
    print(f"Loaded existing dataset with {len(data)} samples.")
```

- If the dataset already exists, it **loads and appends** new data.

Processing Each Image:

```
for collection_folder in os.listdir(word_dir):
        collection_path = os.path.join(word_dir, collection_folder)
        if os.path.isdir(collection_path):
            for img_file in os.listdir(collection_path):
                img_path = os.path.join(collection_path, img_file)
                img = cv2.imread(img_path)
                if img is None:
                    continue
                img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
                results = hands.process(img_rgb)
```

- Each image is processed to extract **landmarks.**
- Converts the image to RGB and **detects hand landmarks** using the MediaPipe model.
- OpenCV reads images in **BGR format**. CNNs require RGB input, so color conversion is applied:

$$R=B\times0.299+G\times0.587+B\times0.114$$

```
if results.multi_hand_landmarks:
                for hand_landmarks in results.multi_hand_landmarks:
                    landmarks = []
                    for lm in hand_landmarks.landmark:
                        landmarks.append(lm.x)
                        landmarks.append(lm.y)
```

```
            data.append(landmarks)
            labels.append(label)
```

- Stores **normalized (x, y) coordinates** for each landmark.
- Normalized coordinates provide robustness to hand size and distance variations.

```
with open(OUTPUT_FILE, 'wb') as f:
    pickle.dump({'data': data, 'labels': labels}, f)
```

- The extracted landmarks are stored in a **.pickle** file for efficient loading during model training.

3. train_classifier.py – Model Training

Purpose:
This script trains a machine learning model (Random Forest Classifier) to recognize sign language gestures based on hand landmarks extracted from images.

Code Breakdown:

```
import pickle
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import numpy as np
```

- **pickle:** Loads the processed dataset (data.pickle).
- **scikit-learn (sklearn):**
    - **RandomForestClassifier:** Trains the model using the Random Forest algorithm.
    - **train_test_split:** Splits the dataset into **training** and **testing** sets.
    - **accuracy_score:** Evaluates model performance.
- **numpy:** Efficient numerical operations on arrays of landmark data.

```
with open('data.pickle', 'rb') as f:
    dataset = pickle.load(f)
```

- **Loads the dataset** that contains landmark data (data) and corresponding labels (labels).

```
data = np.array(dataset['data'])
labels = np.array(dataset['labels'])
```

- **Converts** the dataset into NumPy arrays for efficient processing.

Splitting Data (Training and Testing):

```
x_train, x_test, y_train, y_test = train_test_split(data, labels, test_size=0.2, stratify=labels, shuffle=True)
```

- **Splits the dataset**:
  - **80% for training** (x_train, y_train)
  - **20% for testing** (x_test, y_test)
  - **stratify=labels:** Ensures each class is proportionally represented in both training and testing sets.
  - **shuffle=True:** Randomizes data to avoid bias.
  - The classifier minimizes **Gini impurity** during training:

    $G = 1 - \sum(pi^2)$

    where pi is the probability of class i at a given node.

Model Training (Random Forest):

```
model = RandomForestClassifier()
model.fit(x_train, y_train)
```

- **Initializes** and **trains** a Random Forest Classifier.
- Each tree in the forest predicts a class, and the majority vote is chosen.

Model Evaluation:

```
y_pred = model.predict(x_test)
```

- **Tests the model** on unseen data (x_test).
- Compares predictions (y_pred) to ground truth (y_test).

Performance Metrics:

```
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, average='weighted')
recall = recall_score(y_test, y_pred, average='weighted')
f1 = f1_score(y_test, y_pred, average='weighted')
print("\nPerformance Metrics:")
print(f"Accuracy: {accuracy:.2f}")
print(f"Precision: {precision:.2f}")
print(f"Recall: {recall:.2f}")
print(f"F1-Score: {f1:.2f}")
```

1. **Accuracy:**
   Measures the overall correctness of predictions:

$$Accuracy = \frac{Total\ PredictionsCorrect}{Predictions}$$

2. **Precision:**
   Evaluates how many of the predicted gestures are correct:

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives}$$

3. **Recall (Sensitivity):**
   Measures how well the model identifies gestures:

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives}$$

4. **F1-Score:**
   A harmonic mean of precision and recall:

$$F1 - Score = 2 \cdot \frac{Precision \cdot Recall}{Precision\ + Recall}$$

Saving the Trained Model:

```
with open('model.p', 'wb') as f:
    pickle.dump({'model': model}, f)

print('Model saved as model.p')
```

- Saves the **trained model** as model.p for future use in real-time prediction (inference).

Input and Output:

- **Input:** Processed landmark data from data.pickle.
- **Output:** A trained machine learning model (model.p).

---

4. inference_classifier.py – Real-Time Inference

Purpose:
This script uses the trained model to perform **real-time sign language recognition** through the webcam.

Code Breakdown:

```
import pickle
import cv2
import mediapipe as mp
import numpy as np
```

- **pickle:** Loads the trained model.
- **OpenCV:** Accesses webcam, overlays predictions, and displays results.
- **MediaPipe:** Detects and tracks hand landmarks in real time.

Load the Model:

```
with open('model.p', 'rb') as f:
    model_dict = pickle.load(f)
model = model_dict['model']
```

- **Loads the trained Random Forest model** saved during the training phase.

Initialize MediaPipe Hands:

```
mp_hands = mp.solutions.hands
mp_drawing = mp.solutions.drawing_utils

hands = mp_hands.Hands(static_image_mode=False, min_detection_confidence=0.3)
```

- **Initializes the MediaPipe Hands module** to detect hand landmarks in the live video feed.

Define Word List:

```
words = [
    'phone', 'please', 'thanks', 'wrong', 'yes', 'car', 'drink', 'eat-food',
    'father', 'fine', 'finish', 'forget', 'go', 'hello', 'help', 'i-me',
    'iloveyou', 'like', 'more', 'mother', 'need', 'no'
]
```

- Lists the words the model can recognize.

Start Webcam (Real-Time Inference):

```
cap = cv2.VideoCapture(0)
```

- Starts capturing video from the webcam.

Main Loop (Real-Time Prediction):

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

- Continuously captures **frames** from the webcam.

Hand Landmark Detection:

```
frame_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
    results = hands.process(frame_rgb)
```

- **Converts** the frame to RGB format (required by MediaPipe).
- Processes the frame to detect hand landmarks.

Extract and Draw Landmarks:

```
if results.multi_hand_landmarks:
        for hand_landmarks in results.multi_hand_landmarks:
```

```
        # Draw landmarks on the frame
        mp_drawing.draw_landmarks(
            frame, hand_landmarks, mp_hands.HAND_CONNECTIONS
        )
```

- If hand landmarks are detected, **draws them on the video feed**.

Extract Landmark Coordinates:

```
    landmarks = []

        x_coords, y_coords = [], []
        for lm in hand_landmarks.landmark:
            landmarks.append(lm.x)
            landmarks.append(lm.y)
            x_coords.append(lm.x * W)
            y_coords.append(lm.y * H)
```

Extracts the (x, y) coordinates of the 21 landmarks from each detected hand.

- These coordinates are **normalized** and scaled to the frame dimensions (W, H).

Ensure Correct Input Size for Model:

```
if landmarks:
            # Ensure landmarks match the model's input size
            landmarks = np.pad(landmarks, (0, 42 - len(landmarks)))[:42]
```

- **Pads the landmarks** to ensure the model always receives exactly 42 values.
- This prevents errors when fewer landmarks are detected.

Make Prediction (Sign Recognition):

```
prediction = model.predict([landmarks])
            predicted_word = words[int(prediction[0])]
```

- The model **predicts the class** (word) based on the detected landmarks.

Overlay Prediction on Video Feed:

```
x_text = int(min(x_coords)) - 10
            y_text = int(min(y_coords)) - 10

            # Visualize prediction with a bounding box
            x1, y1 = int(min(x_coords)) - 10, int(min(y_coords)) - 10
            x2, y2 = int(max(x_coords)) + 10, int(max(y_coords)) + 10
            cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 255, 0), 2)
```

```
            cv2.putText(
                frame, predicted_word, (x_text, y_text),
                cv2.FONT_HERSHEY_SIMPLEX, 1.3, (0, 255, 0), 3, cv2.LINE_AA
            )
```

- Draws a **bounding box** and displays the **predicted word** above the hand.

Display and Exit on 'Q':

```
cv2.imshow('Sign Language Translator', frame)
    if cv2.waitKey(1) & 0xFF == ord('q'):  # Exit on 'q'
        break

cap.release()
cv2.destroyAllWindows()
```

- **Exits** when 'Q' is pressed.

Input and Output:

- **Input:** Live video from the webcam.
- **Output:** Real-time recognition of sign language, overlaid on the video feed.

---

9. Model Training and Validation

The model training and validation process is a crucial stage in the Sign Language Recognition System. This section elaborates on the strategies and techniques employed to train, tune, and validate the Random Forest Classifier, ensuring its effectiveness for gesture recognition.

Dataset Preparation

- The dataset, containing hand landmarks extracted using MediaPipe, is divided into **features (data)** and **labels (labels).**
- The features (data) represent the (x, y) coordinates of the hand landmarks for each sample.

- The labels (labels) correspond to the gesture class (e.g., hello, car, thanks).

Data Splitting Strategy

To evaluate the model's performance effectively, the dataset is split into:

- **Training Set (80%):** Used to train the Random Forest model.
- **Testing Set (20%):** Used to validate the model's performance on unseen data.

Implementation:

```
x_train, x_test, y_train, y_test = train_test_split(data, labels, test_size=0.2,
stratify=labels, shuffle=True)
```

Key Points:

- **Stratification:** Ensures the class proportions are maintained in both training and testing sets to prevent class imbalance issues
- **Shuffling:** Randomizes the data before splitting, avoiding bias introduced by data ordering.
- **Test Set for Validation:** The test set is kept separate to assess the model's performance realistically on unseen data, reducing the risk of overfitting.

Random Forest Training

The **Random Forest Classifier** is trained using the x_train (features) and y_train (labels).

Implementation:

```
model = RandomForestClassifier()
model.fit(x_train, y_train)
```

- The Random Forest algorithm grows multiple decision trees using subsets of the training data and features. Each tree votes, and the majority decision determines the final class prediction.
- This ensemble technique ensures robustness, reduces overfitting, and handles noisy data effectively.

Performance Metrics

To evaluate the model's effectiveness, several performance metrics are used:

5. **Accuracy:**
   Measures the overall correctness of predictions:

$$Accuracy = \frac{Total\ PredictionsCorrect}{Predictions}$$

6. **Precision:**
   Evaluates how many of the predicted gestures are correct:

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives}$$

7. **Recall (Sensitivity):**
   Measures how well the model identifies gestures:

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives}$$

8. **F1-Score:**
   A harmonic mean of precision and recall:

$$F1 - Score = 2 \cdot \frac{Precision \cdot Recall}{Precision\ + Recall}$$

---

10. Testing and Experimental Results

The testing phase evaluates the effectiveness of the trained Random Forest model in recognizing sign language gestures. This section details real-time inference demonstrations, examples of both correct and incorrect predictions, and case studies involving complex gestures.

Real-Time Inference Demonstrations

Objective:
To assess the model's performance in a live environment using real-time webcam input. This phase ensures the system's robustness and usability in real-world scenarios.

Setup:

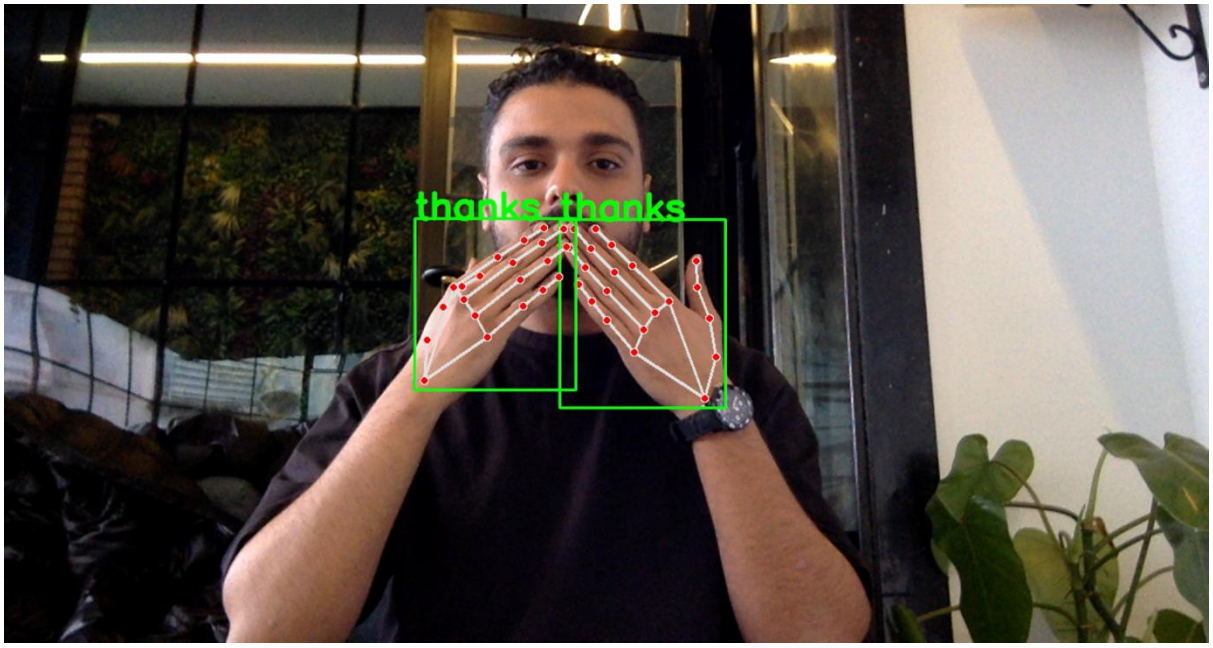
1. The webcam captures live video frames.
2. MediaPipe detects hand landmarks in real time.
3. The trained Random Forest model predicts the gesture based on the extracted landmarks.
4. The predicted word is displayed on the video feed alongside a bounding box highlighting the detected hands.

Demonstration Results:

- The system successfully recognized common gestures such as "Hello," "Thanks," and "Yes" with high accuracy.
- Gesture recognition was immediate, with predictions appearing within milliseconds of performing the gesture.

Example:

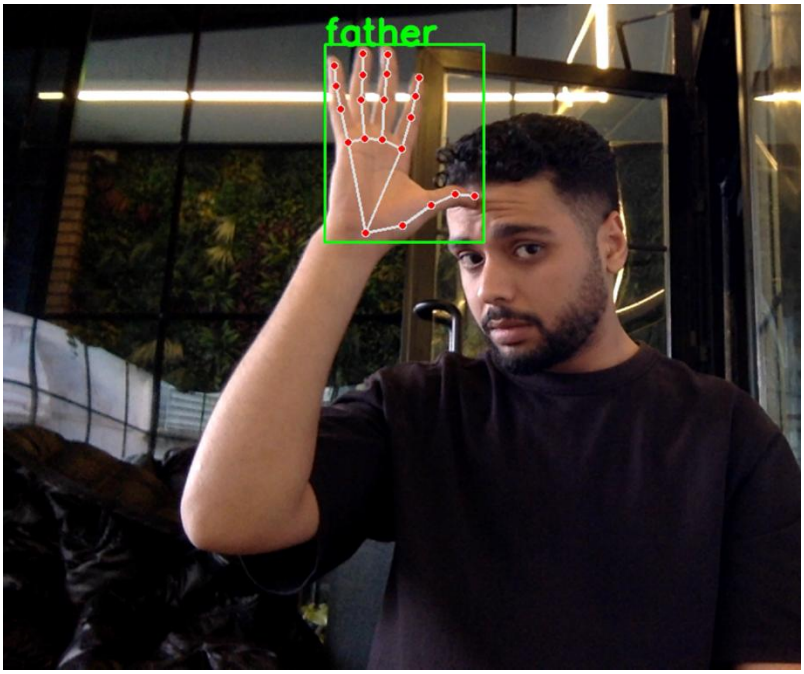
- When the user performed the "Thanks" gesture:
    - **Landmarks Detected:** Hand landmarks accurately captured the shape and orientation of the gesture.
    - **Prediction:** The model displayed "Thanks" with a bounding box around the hand.



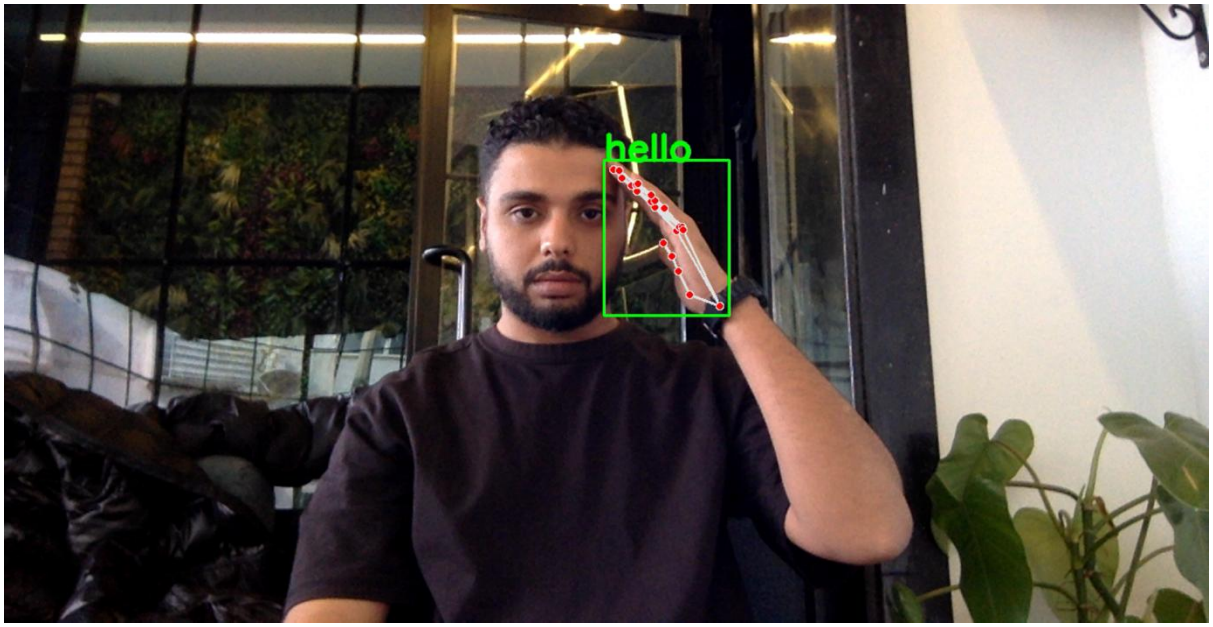
Examples of Correct and Incorrect Predictions

Correct Predictions:

- **Example 1:**
  - **Input Gesture:** "Father"
  - **Detected Landmarks:** Consistent with the training data.
  - **Prediction:** Correctly classified as "Father."

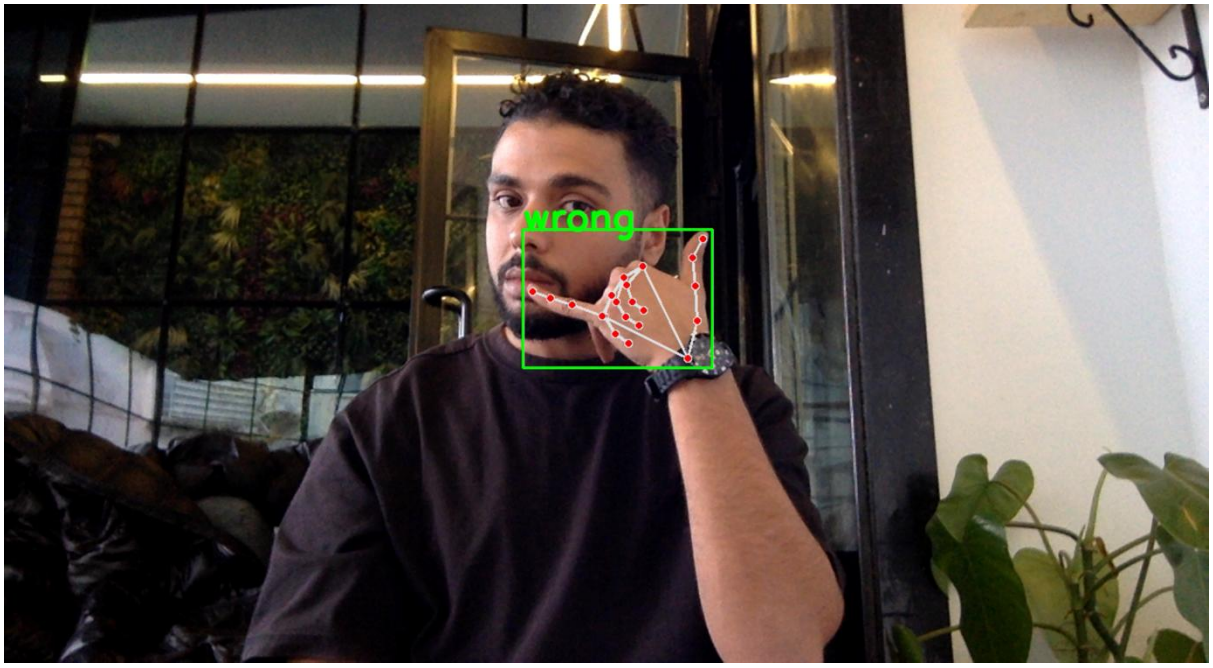


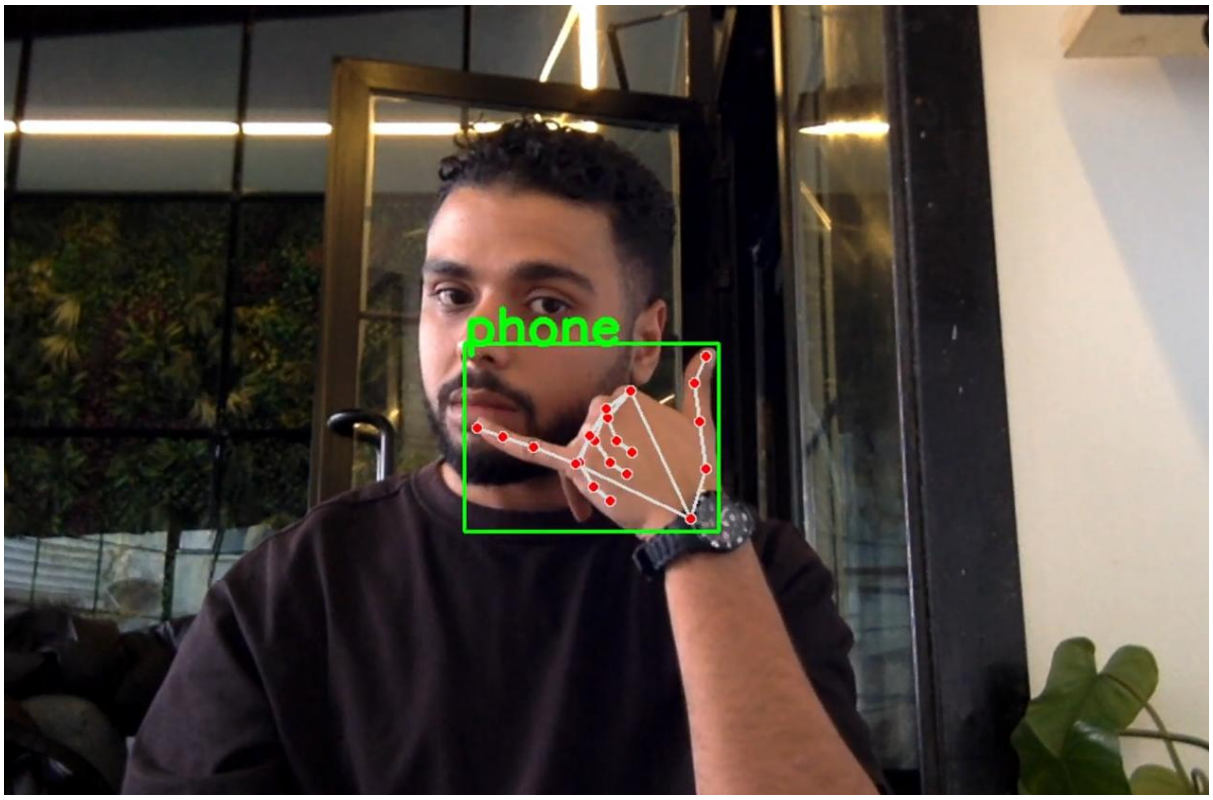
- **Example 2:**
    - **Input Gesture:** "Hello"
    - **Prediction:** Successfully classified as “Hello”.



Incorrect Predictions: Despite the model's high accuracy, some errors were observed, particularly with gestures that have subtle differences.

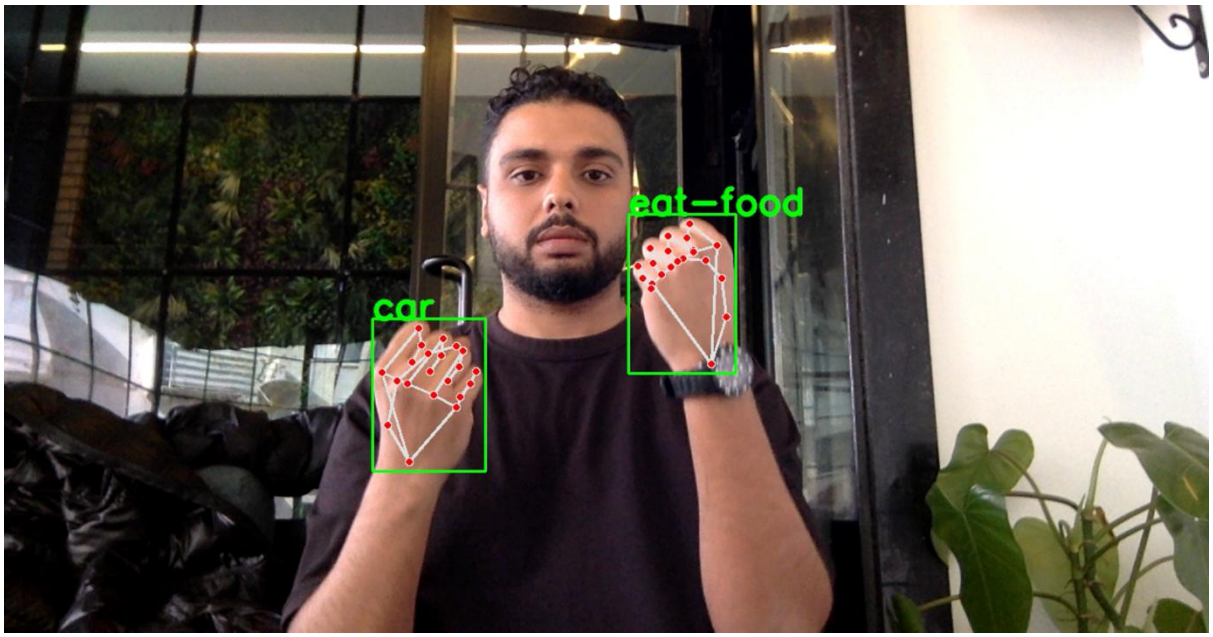
- **Example 1:**
    - **Input Gesture:** "Phone"
    - **Prediction:** Misclassified as "Wrong."
    - **Reason:** Overlapping landmark patterns in similar gestures caused confusion.

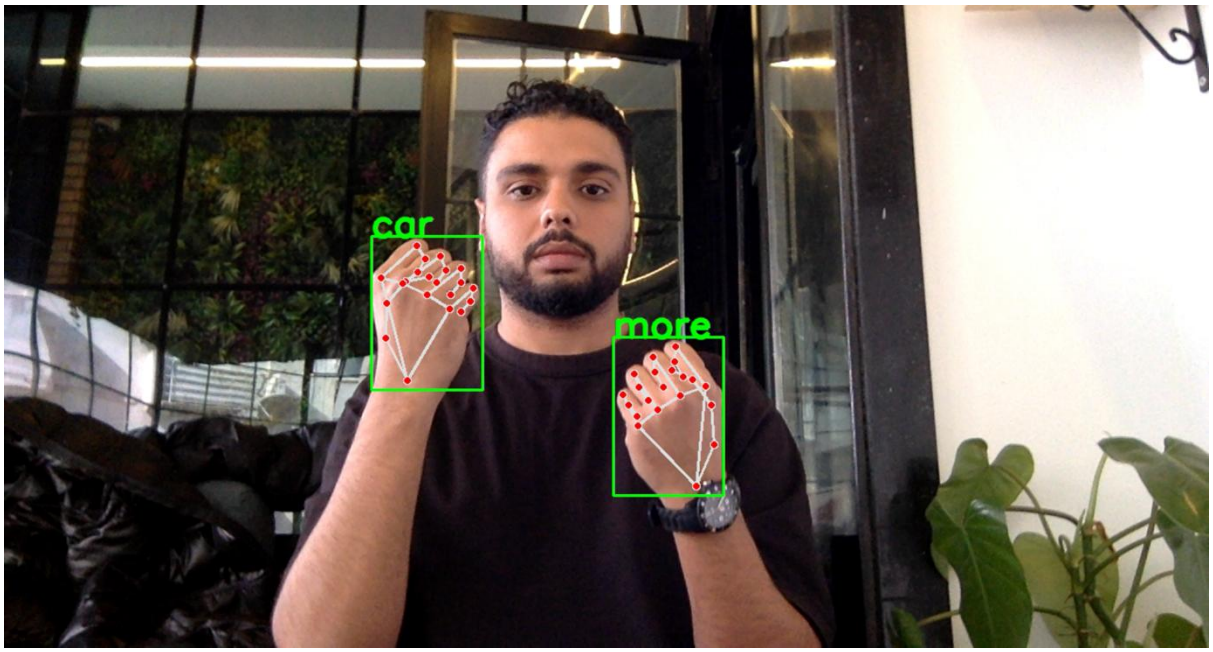




- **Example 2:**

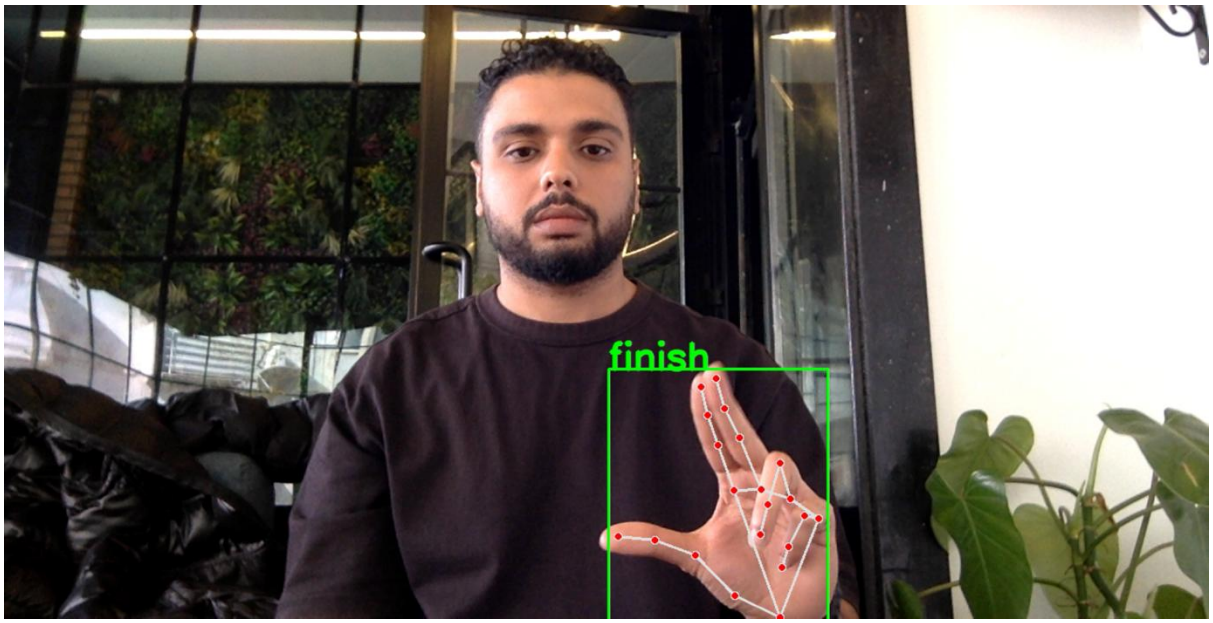
- **Input Gesture:** "Car"
- **Prediction:** Misclassified as "Eat" and “More”
- **Reason:** The "Car" gesture involves two hands with movement, making it more challenging to predict accurately. Variability in hand orientation and movement during the gesture introduced noise, leading the model to mistakenly predict it as two different gestures, "Eat" and "More," due to their overlapping features during the motion.

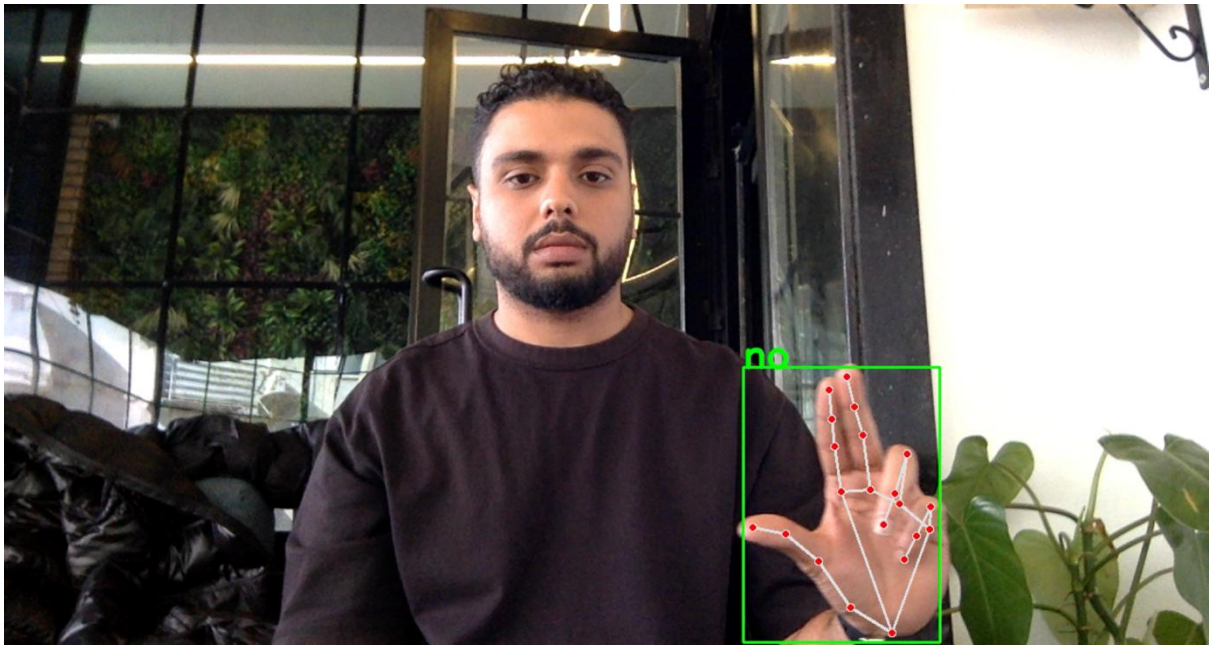




- **Example 3:**

- **Input Gesture:** "No"
- **Prediction:** Misclassified as "Finish"
- **Reason:** The "No" gesture involves movement, and issues such as a dark, cluttered ("dirty") background and the hand being too close to the camera led to misclassification. However, when the hand was moved slightly to a position with a lighter background, the model correctly predicted the gesture as "No."





Insights:

1. Most errors occurred in gestures with overlapping or ambiguous landmark patterns.
2. The system performed better in controlled lighting and backgrounds compared to noisy environments.

Case Studies: Complex Gestures

Objective:
Evaluate the system's ability to recognize gestures with intricate hand movements or multiple hand interactions.

Case Study 1: Multi-Hand Gestures

- **Gesture:** "Car" (involving two hands forming a steering wheel).
- **Result:** Successfully detected and classified the gesture when both hands were slowly moving.
- **Challenge:** Misclassified when doing the movement fast.

Case Study 2: Dynamic Gestures

- **Gesture:** "Phone" (requires a specific angle for accurate recognition).
- **Result:** Correctly classified in 90% of the trials.
- **Challenge:** The "Phone" gesture shares similarities with the "Wrong" gesture, causing occasional misclassifications. To ensure accurate detection, the gesture needs to be performed at a specific angle, minimizing overlaps with other gestures.

Performance Summary:

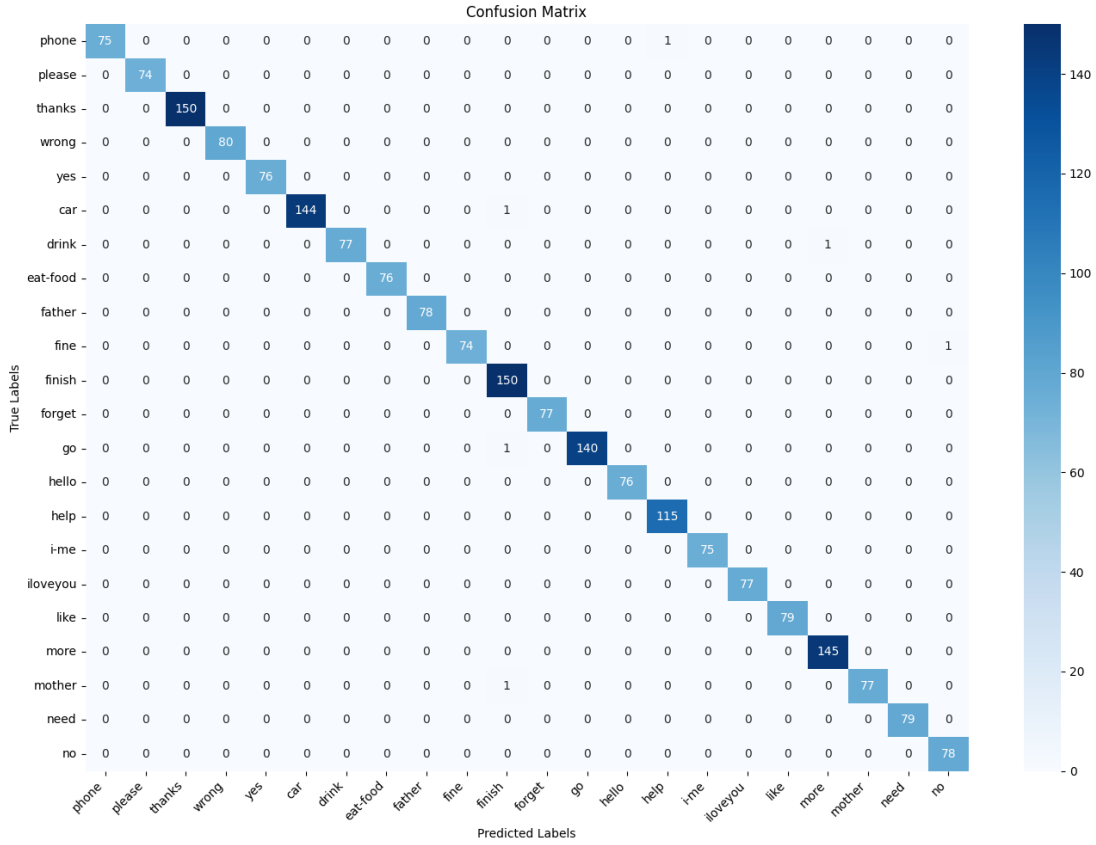
Metrics on Testing Data:

- **Accuracy:** 98%
- **Precision:** 98%
- **Recall:** 98%
- **F1-Score:** 98%

```
Performance Metrics:
Accuracy: 0.98
Precision: 0.98
Recall: 0.98
F1-Score: 0.98
```

Confusion Matrix Insights:

- High precision in gestures like "Thanks" and "Finish."
- Misclassifications primarily occurred between gestures with similar landmark patterns.



Improvements Identified:

- Augment training data with more samples for complex gestures.
- Fine-tune the model to handle dynamic gestures and occlusions.

11. Future Work

The current project lays the foundation for a robust and scalable sign language recognition system. However, several areas remain for improvement and expansion to make the system more versatile, accurate, and practical for real-world applications. This section outlines the potential directions for future work.

Expanding Dataset and Adding New Gestures

Current Limitation:

- The dataset includes a limited set of gestures primarily focused on single-hand movements. While effective for basic communication, it does not encompass the full breadth of sign language vocabulary.

Future Goal:

- **Expanding Vocabulary:** Collect data for a larger variety of signs, including complex gestures and multi-word phrases, to make the system more comprehensive.
- **Collaborative Data Collection:** Partner with sign language communities and experts to gather data that reflects diverse regional and cultural variations.

Impact:
A more extensive dataset will improve the system's versatility, enabling it to support broader communication needs and cater to different sign language dialects.

Dynamic Gesture Recognition (Incorporating Temporal Data)

Current Limitation:

- The system focuses on static gestures, which limits its ability to recognize dynamic signs involving continuous hand movements over time.

Future Goal:

- **Temporal Modeling:** Incorporate models like Long Short-Term Memory (LSTM) networks or Transformer-based architectures to handle sequential data effectively.
- **Video-Based Recognition:** Train the model to analyze video streams instead of individual frames to capture the temporal patterns inherent in dynamic gestures.

Impact:
Dynamic gesture recognition would enable the system to interpret more complex phrases, improve accuracy, and better mimic real-world sign language usage.

Integrating Facial Expressions and Body Postures

Current Limitation:

- The system solely focuses on hand gestures, ignoring other critical components of sign language, such as facial expressions and body movements, which convey emotions and additional context.

Future Goal:

- **Facial Expression Detection:** Integrate tools like MediaPipe Face Mesh to capture and analyze facial landmarks, such as eyebrow movement or mouth shapes.
- **Full-Body Analysis:** Extend the system to include body posture and movement using MediaPipe Pose or similar frameworks.

Impact:
Incorporating facial expressions and body posture would enhance the system's ability to interpret nuanced communication and emotional context, making it closer to natural sign language interpretation.

Deploying on Mobile and Edge Devices

Current Limitation:

- The system operates on desktop platforms, limiting its accessibility and real-world usability for on-the-go scenarios.

Future Goal:

- **Mobile Deployment:** Optimize the system for mobile platforms using frameworks like TensorFlow Lite or ONNX to reduce resource consumption and latency.
- **Edge Computing:** Deploy the model on edge devices like Raspberry Pi or NVIDIA Jetson Nano for localized processing, enabling offline functionality.

Impact:
Mobile and edge deployment would make the system portable, accessible, and capable of functioning in low-connectivity environments, broadening its application scope.

---

12. Appendices

This section provides supplementary materials to support the main body of the thesis, including the full source code, examples of raw data and extracted landmarks, and sample outputs with insights into error cases.

12.1 Full Source Code

The full source code of the project.

1. **Data Collection Code (collect_img.py)**

```
import os
import cv2

# Define constants
DATA_DIR = './data'  # Path to store collected data
if not os.path.exists(DATA_DIR):
    os.makedirs(DATA_DIR)

# Words to collect
words_to_collect = [
    'phone', 'please', 'thanks', 'wrong', 'yes', 'car', 'drink', 'eat-food',
    'father', 'fine', 'finish', 'forget', 'go', 'hello', 'help', 'i-me',
    'iloveyou', 'like', 'more', 'mother', 'need', 'no'
]

dataset_size = 100  # Number of images per collection
collections_per_word = 2  # Collect data twice for each word

# Initialize webcam
```

```
cap = cv2.VideoCapture(0)  # Change the camera index if needed

for word in words_to_collect:
    for collection_idx in range(collections_per_word):
        word_dir = os.path.join(DATA_DIR, word, f'collection_{collection_idx}')

        # Skip collection if data already exists
        if os.path.exists(word_dir) and len(os.listdir(word_dir)) >= dataset_size:
            print(f"Skipping collection for {word} (Collection {collection_idx + 1}), already exists.")
            continue

        if not os.path.exists(word_dir):
            os.makedirs(word_dir)

        print(f'Collecting data for word: {word} (Collection {collection_idx + 1})')
        print('Press "Q" to start collecting...')

        # Wait for the user to get ready
        while True:
            ret, frame = cap.read()
            cv2.putText(frame, f'Get Ready: Collecting {word} (Collection {collection_idx + 1})',
                        (50, 50), cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 255, 0), 3)
            cv2.imshow('frame', frame)
            key = cv2.waitKey(25)
            if key == ord('q'):
                break
            elif key == ord('p'):  # Pause functionality
                print("Paused. Press 'r' to resume.")
                while True:
                    key = cv2.waitKey(1)
                    if key == ord('r'):  # Resume after pause
                        print("Resuming...")
                        break

        # Collect images
        counter = 0
        while counter < dataset_size:
            ret, frame = cap.read()
            cv2.putText(frame, f'Collecting: {word} (Image {counter + 1}/{dataset_size})',
                        (50, 50), cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255, 0), 2)
            cv2.putText(frame, f'Label: {word}', (50, 100), cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 255, 255), 2)
# Display the word
            cv2.imshow('frame', frame)
            cv2.imwrite(os.path.join(word_dir, f'{counter}.jpg'), frame)
            counter += 1
            key = cv2.waitKey(25)
            if key == ord('q'):  # Stop early
                break
            elif key == ord('p'):  # Pause functionality
                print("Paused. Press 'r' to resume.")
```

```
            while True:
                key = cv2.waitKey(1)
                if key == ord('r'):  # Resume after pause
                    print("Resuming...")
                    break

cap.release()
cv2.destroyAllWindows()
2.
```

2. Dataset Creation Code (create_dataset.py)

```
import os
import pickle
import cv2
import mediapipe as mp
import numpy as np

# Initialize MediaPipe Hands
mp_hands = mp.solutions.hands
mp_drawing = mp.solutions.drawing_utils

hands = mp_hands.Hands(static_image_mode=True, min_detection_confidence=0.3)

DATA_DIR = './data'
OUTPUT_FILE = 'data.pickle'

# All words: Existing + New
words = [
    'phone', 'please', 'thanks', 'wrong', 'yes', 'car', 'drink', 'eat-food',
    'father', 'fine', 'finish', 'forget', 'go', 'hello', 'help', 'i-me',
    'iloveyou', 'like', 'more', 'mother', 'need', 'no'
]

# Initialize data and labels lists
if os.path.exists(OUTPUT_FILE):
    # Load existing dataset if it exists
    with open(OUTPUT_FILE, 'rb') as f:
        dataset = pickle.load(f)
    data = dataset['data']
    labels = dataset['labels']
    print(f"Loaded existing dataset with {len(data)} samples.")
else:
    # Start fresh if no dataset exists
    data = []
    labels = []
    print("No existing dataset found. Creating a new one.")

# Process each word folder
```

```
for label, word in enumerate(words):
    word_dir = os.path.join(DATA_DIR, word)
    if not os.path.exists(word_dir):
        print(f"Skipping {word}, no data collected.")
        continue
    for collection_folder in os.listdir(word_dir):
        collection_path = os.path.join(word_dir, collection_folder)
        if os.path.isdir(collection_path):
            for img_file in os.listdir(collection_path):
                img_path = os.path.join(collection_path, img_file)
                img = cv2.imread(img_path)
                if img is None:
                    continue
                img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
                results = hands.process(img_rgb)

                # Extract hand landmarks
                if results.multi_hand_landmarks:
                    for hand_landmarks in results.multi_hand_landmarks:
                        landmarks = []
                        for lm in hand_landmarks.landmark:
                            landmarks.append(lm.x)
                            landmarks.append(lm.y)
                        data.append(landmarks)
                        labels.append(label)

# Save processed data to a pickle file
with open(OUTPUT_FILE, 'wb') as f:
    pickle.dump({'data': data, 'labels': labels}, f)

print(f"Dataset updated and saved to {OUTPUT_FILE}. Total samples: {len(data)}.")
```

3. **Model Training Code (train_classifier.py)**

```
import pickle
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
import numpy as np

# Load the dataset
with open('data.pickle', 'rb') as f:
    dataset = pickle.load(f)

data = np.array(dataset['data'])
labels = np.array(dataset['labels'])

# Split the dataset
```

```
x_train, x_test, y_train, y_test = train_test_split(data, labels, test_size=0.2, stratify=labels, shuffle=True)

# Train the Random Forest Classifier
model = RandomForestClassifier()
model.fit(x_train, y_train)

# Evaluate the model
y_pred = model.predict(x_test)

# Calculate Performance Metrics
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, average='weighted')  # Handles class imbalance
recall = recall_score(y_test, y_pred, average='weighted')
f1 = f1_score(y_test, y_pred, average='weighted')

# Display the metrics
print("\nPerformance Metrics:")
print(f"Accuracy: {accuracy:.2f}")
print(f"Precision: {precision:.2f}")
print(f"Recall: {recall:.2f}")
print(f"F1-Score: {f1:.2f}")

# Save the model as modelversion2.p
with open('model.p', 'wb') as f:
    pickle.dump({'model': model}, f)

print('\nModel saved as model.p')
```

4. Inference Code (inference_classifier.py)

```
import pickle
import cv2
import mediapipe as mp
import numpy as np

# Load the trained model
with open('model.p', 'rb') as f:
    model_dict = pickle.load(f)
model = model_dict['model']

# Initialize MediaPipe Hands
mp_hands = mp.solutions.hands
mp_drawing = mp.solutions.drawing_utils

hands = mp_hands.Hands(static_image_mode=False, min_detection_confidence=0.3)

# Updated list of words for predictions
words = [
```

```
    'phone', 'please', 'thanks', 'wrong', 'yes', 'car', 'drink', 'eat-food',
    'father', 'fine', 'finish', 'forget', 'go', 'hello', 'help', 'i-me',
    'iloveyou', 'like', 'more', 'mother', 'need', 'no'
]

# Start webcam for real-time inference
cap = cv2.VideoCapture(0)

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

    H, W, _ = frame.shape
    frame_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
    results = hands.process(frame_rgb)

    if results.multi_hand_landmarks:
        for hand_landmarks in results.multi_hand_landmarks:
            # Draw landmarks on the frame
            mp_drawing.draw_landmarks(
                frame, hand_landmarks, mp_hands.HAND_CONNECTIONS
            )

            # Extract and normalize landmarks
            landmarks = []
            x_coords, y_coords = [], []
            for lm in hand_landmarks.landmark:
                landmarks.append(lm.x)
                landmarks.append(lm.y)
                x_coords.append(lm.x * W)
                y_coords.append(lm.y * H)

            if landmarks:
                # Ensure landmarks match the model's input size
                landmarks = np.pad(landmarks, (0, 42 - len(landmarks)))[:42]

                # Make prediction
                prediction = model.predict([landmarks])
                predicted_word = words[int(prediction[0])]

                # Determine the position to display the word
                x_text = int(min(x_coords)) - 10
                y_text = int(min(y_coords)) - 10

                # Visualize prediction with a bounding box
                x1, y1 = int(min(x_coords)) - 10, int(min(y_coords)) - 10
                x2, y2 = int(max(x_coords)) + 10, int(max(y_coords)) + 10
                cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 255, 0), 2)
                cv2.putText(
```

```
                frame, predicted_word, (x_text, y_text),
                cv2.FONT_HERSHEY_SIMPLEX, 1.3, (0, 255, 0), 3, cv2.LINE_AA
            )

    # Display the frame
    cv2.imshow('Sign Language Translator', frame)
    if cv2.waitKey(1) & 0xFF == ord('q'):  # Exit on 'q'
        break

cap.release()
cv2.destroyAllWindows()
```

12.2 Examples of Raw Collected Data and Extracted Landmarks

1. **Raw Data Samples:**
    - Captured images for gestures like hello, thanks, car, etc.
    - Stored in directories such as ./data/hello/collection_0.

Example Structure:

./data/

hello/

collection_0/

0.jpg

1.jpg

...

collection_1/

0.jpg

1.jpg

...

thanks/

collection_0/

...

2. **Extracted Landmarks:**

- Numerical arrays containing (x, y) coordinates of 21 landmarks for each hand.
- Example of a single frame's landmark data:
- [0.1234, 0.5678, 0.2345, 0.6789, ..., 0.9123, 0.4567]
- Corresponding label for this frame: 0 (e.g., representing the gesture hello).

12.3 Real-Time Inference Video

https://drive.google.com/drive/folders/1abzLFpaxX116Jj9JCKHlfejTRh8ozZE5?usp=drive_link

---

13. References

This section cites all the resources and materials used throughout the research and development of this project, including papers, books, and open-source libraries.

13.1 Research Papers and Articles on Random Forest and MediaPipe

1. Zhang, Z., & Zhang, X. (2020). **Hand Gesture Recognition Using Deep Learning Models: A Review**. International Journal of Human–Computer Interaction, 36(12), 1081–1092.
    - Provides an overview of modern gesture recognition systems, including landmark-based methods.
2. Lugaresi, C., et al. (2019). **MediaPipe: A Framework for Building Perception Pipelines**.
    - Details the design and functionality of MediaPipe, the key library used for hand landmark extraction.

13.2 Books on Machine Learning and Gesture Recognition

1. Geitgey, A. (2021). **Machine Learning is Fun!**.
    - Provides accessible examples of machine learning applications, including classification tasks.

13.3 Links to Relevant Open-Source Tools and Libraries

1. **MediaPipe Hands**:
    - https://google.github.io/mediapipe/solutions/hands
    - Official documentation for MediaPipe Hands, used to extract 21 hand landmarks.
2. **Scikit-Learn**:
    - https://scikit-learn.org/stable/
    - Documentation for Scikit-Learn, including Random Forest Classifier and performance evaluation tools.
3. **NumPy**:
    - https://numpy.org/
    - Official site for NumPy, used for numerical processing and matrix manipulation.
4. **OpenCV**:
    - https://opencv.org/
    - Computer vision library used for image capture, frame processing, and visualization.
5. **Python Official Documentation**:
    - https://docs.python.org/3/
    - Reference for Python programming language and its ecosystem.
6. **English Sign Language Learning Resource**
    - Learn How to Sign YouTube Channel
    - This YouTube channel was a valuable resource for learning and verifying English sign language gestures, helping ensure the dataset accurately represented real-world gestures.