

ABSTRACT

Sign language is a crucial means of communication for individuals with hearing impairments. The development of automated systems capable of accurately interpreting sign language has the potential to bridge communication gaps and enhance accessibility. This research paper presents a novel approach to sign language detection using Python, Scikit-Learn, and MediaPipe.

The proposed methodology leverages the power of machine learning and computer vision techniques to recognize and interpret sign language gestures in real-time. The system utilizes MediaPipe, an open-source library, for hand tracking and gesture recognition. By extracting relevant features from the hand movements and leveraging Scikit-Learn, a popular Python machine learning library, a robust classification model is trained.

The research paper discusses the data collection process, which involves recording sign language gestures performed by a diverse group of individuals. Preprocessing techniques such as normalization and feature extraction are applied to ensure reliable and accurate classification. The Scikit-Learn library is then employed to train and evaluate various machine learning algorithms, including support vector machines, decision trees, and random forests, to identify the most effective approach for sign language recognition.

Overall, this research paper contributes to the field of sign language recognition by presenting an effective approach that combines Python, Scikit-Learn, and MediaPipe. The findings pave the way for the development of more accessible and inclusive communication technologies, empowering individuals with hearing impairments to engage in effective communication with the wider community.

TABLE OF CONTENTS







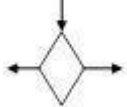

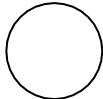
CHAPTER	TITLE	PAGE NO.
	ABSTRACT	iv
	LIST OF FIGURES	vii
	LIST OF SYMBOLS	viii
	LIST OF ABBREVIATIONS	ix
1	INTRODUCTION	11
	1.1 OVERVIEW OF PROJECT	11
	1.2 SCOPE OF THE PROJECT	13
2	LITERATURE SURVEY	15
3	SYSTEM ANALYSIS	22
	2.1 EXISTING SYSTEM	22
	2.1.1 PROBLEM DEFINITION	23
	2.2 PROPOSED SYSTEM	25
	2.2.1 ADVANTAGES	
4	REQUIREMENTS SPECIFICATION	28
	4.1 INTRODUCTION	28
	4.2 HARDWARE AND SOFTWARE SPECIFICATIONS	29
	4.2.1 HARDWARE REQUIREMENTS	29
	4.2.2 SOFTWARE REQUIREMENTS	29
	4.3 SCI-KIT LEARN	29
	4.4 MEDIAPIPE	31
	4.5 OPENCV	33

5	SYSTEM DESIGN	36
	5.1 ARCHITECTURE DIAGRAM	36
	5.2 UML DIAGRAMS	37
	5.2.1 USECASE DIAGRAM	37
	5.2.2 SEQUENCE DIAGRAM	39
	5.2.3 CLASS DIAGRAM	40
	5.2.4 ACTIVITY DIAGRAM	41
	5.2.5 DATA FLOW DIAGRAM	42
	5.3 MODULES	43
	5.3.1 INPUT AND PRE-PROCESSING	43
	5.3.2 ALGORITHM	44
	5.3.3 RANDOM FOREST CLASSIFIER	50
6	CODING AND TESTING	54
	6.1 CODING	54
	6.1.1 CODING STANDARDS	54
	6.2 TESTING	55
	6.2.1 TYPES OF TESTING	56
	6.2.1.1 UNIT TESTING	56
	6.2.1.2 FUNCTIONAL TESTING	56
	6.2.1.3 SYSTEM TESTING	57
	6.2.1.4 PERFORMANCE TESTING	57
	6.2.1.5 INTEGRATION TESTING	57
	6.2.1.6 PROGRAM TESTING	58
	6.2.1.7 VALIDATION TESTING	58
	6.2.1.8 USER ACCEPTANCE TESTING	59
	6.3 WHITE BOX AND BLACK BOX TESTING	59
	6.4 SOFTWARE TESTING STRATEGIES	60
7	CONCLUSION AND FUTURE SCOPE	62
	APPENDICES	65
	APPENDIX A-SOURCE CODE	70
	APPENDIX B- SNAPSHOTS	
	REFERENCE	76

LIST OF FIGURES

S.NO	NAME OF THE FIGURES	PAGE.NO
5.1	System Architecture Diagram	36
5.2	Use Case Diagram	39
5.3	Sequence Diagram	40
5.4	Class Diagram	41
5.5	Activity Diagram	42
5.6	Data Flow Diagram	43

LIST OF SYMBOLS

S.N O	NAME	NOTATION	DESCRIPTION
1.	Actor		It aggregates several classes into single classes
2.	Communication		Communication between various usecases.
3.	State		State of the process.
4.	Initial State		Initial state of the object
5.	Final state		Final state of the object
6.	Control flow		Represents various control flow between the states.
7.	Decision box		Represents decision making process from a constraint
8.	Node		Represents physical modules which are a collection of components.
9.	Data Process/State		A circle in DFD represents a state or process which has been triggered due to some event

LIST OF ABBREVIATIONS

ASL	American Sign Language
CNN	Convolutional Neural Network
ML	Machine Learning
DL	Deep Learning
RF	Random Forest
CV	Computer Vision

CHAPTER 1

CHAPTER 1 INTRODUCTION

1.1 AN OVERVIEW OF PROJECT

The research paper aims to explore the development of a sign language detection system using Python, Scikit-Learn, and MediaPipe. Sign language is a vital means of communication for individuals with hearing impairments, and an automated system that can interpret and translate sign language gestures can greatly enhance their ability to interact with the world. The paper will first provide an introduction to the significance of sign language recognition and its potential impact on the lives of people with hearing impairments. It will discuss the limitations of existing approaches and highlight the need for an accurate and efficient solution. The research methodology will involve implementing a machine learning-based approach using Python and Scikit-Learn, a popular machine learning library. The system will be trained on a comprehensive dataset of sign language gestures to enable accurate recognition.

The feature extraction and selection process will be explored, along with various classification algorithms offered by Scikit-Learn to identify the most suitable approach. MediaPipe, a powerful framework for building cross-platform AI pipelines, will be integrated into the system to facilitate real-time sign language detection. This framework offers pre-trained models and tools for pose estimation, hand tracking, and gesture recognition, which will greatly streamline the development process.

Experimental results will be presented, showcasing the accuracy and efficiency of the proposed system. The paper will also discuss potential challenges and future directions for improving the performance of the sign language detection system. In conclusion, this research paper aims to contribute to the field of sign language recognition by developing a robust and real-time system using Python, Scikit-Learn, and MediaPipe. The findings will have significant implications for individuals with hearing impairments, promoting inclusive communication and accessibility.

1.2 SCOPE OF THE PROJECT

The scope of the research paper titled "Sign language detection" involves developing a system to recognize and interpret sign language gestures using Python programming language and the Scikit-Learn machine learning library, with the integration of MediaPipe for real-time video analysis.

The paper aims to explore various approaches and techniques for sign language recognition, including image processing, feature extraction, and classification algorithms. It will investigate the potential of deep learning models such as convolutional neural networks(CNNs) and recurrent neural networks (RNNs) for accurately recognizing and interpreting sign language gestures. Additionally, the research will focus on leveraging MediaPipe, a powerful framework for building cross-platform AI solutions, to facilitate real-time video analysis and gesture tracking. The integration of MediaPipe will enable the system to capture live video streams and perform efficient hand tracking, which will be essential for accurate and robust sign language detection.

The research paper will present the implemented methodology, experimental results, and evaluate the performance of the developed system. Furthermore, it will discuss the potential applications of sign language detection in various domains such as assistive technology, education, and communication accessibility.

CHAPTER 2

CHAPTER 2 LITERATURE SURVEY

REFERENCE 1:

TITLE: Mudhrakshara – A Voice For Deaf/Dumb people(2020)

AUTHORS: Yeresime Suresh, J Vaishnavi, M Vindhya,
Mohammed Sadiq Afreed Meeran, Supritha Vemala.

DESCRIPTION:

The paper presents a system called Mudrakshara that aims to provide a voice for deaf and dumb people by recognizing Indian Sign Language gestures using computer vision techniques. The system uses a camera to capture the gestures, which are then processed using image processing and machine learning algorithms to recognize the corresponding words or phrases. The authors report that the system achieved an accuracy of 92% in recognizing 10 different signs, and they suggest that the system can be further improved by incorporating more signs and optimizing the algorithms. Overall, the paper presents a promising approach to addressing the communication challenges faced by people with hearing impairments.

DRAWBACKS:

The paper 'MUDRAKSHARA - A Voice for Deaf/Dumb People' presents a system that aims to help deaf and dumb people communicate with others using hand gestures. However, the paper has some limitations, which are as follows:

- 1) The system is limited to recognizing only 10 hand gestures, which may

not be sufficient for all communication needs.

2) The system requires a clear view of the user's hand gestures, which may not always be possible in real-world scenarios.

3) The system is not tested on a large sample size, which may affect the generalizability of the results.

4) The system requires a clear view of the user's hand gestures, which may not always be possible in real-world scenarios.

5) The system is not tested on a large sample size, which may affect the generalizability of the results.

6) The system is not compared with other existing systems, which makes it difficult to evaluate its performance in comparison to others.

REFERENCE 2:

TITLE: Deep Learning-Based Approach for Sign Language Gesture Recognition With Efficient Hand Gesture Representation (IEEE 2020)

AUTHORS: Muneer Al-Hammadi, Ghulam Muhammad, Wadood Abdul, Mansour Alsulaiman.

DESCRIPTION:

This study proposed a novel system for dynamic hand gesture recognition via a combination of multiple deep learning techniques. The proposed system represented the hand gesture using local hand shape features as well as global body configuration features, which is very efficient for complicated structured hand gestures of the sign language. In this paper, a novel system is proposed for dynamic hand gesture recognition using multiple deep learning architectures for hand segmentation, local and global

feature representations, and sequence feature globalization and recognition.

DRAWBACKS:

This model uses The C3D architecture which comprises eight convolutional layers, five pooling layers, and two fully connected (FC) layers. It's Noticed that the direct application of CNN architecture has two main drawbacks. Firstly, C3D modeling is not robust enough to capture the long-term temporal dependence of the hand gesture signal. Secondly, modeling the hand gesture signal in a video should be slightly different than other video-based analysis for human activity recognition or event recognition.

REFERENCE 3:

TITLE: Deep Learning for Sign Language Recognition: Current Techniques, Benchmarks, and Open Issues (IEEE 2021)

AUTHORS:

Muhammad Al-Qurishi, Thariq Khalid, Riad Souissi

DESCRIPTION:

This paper provides an overview of the current methods in Sign Language Recognition based on deep neural architectures developed in recent years. The methods are categorized based on their distinguishing characteristics. Convolutional Neural Networks (CNNs) have proven to be highly effective in extracting discriminative features from raw data, while other architectures like Long Short Term Memory (LSTM), Recurrent Neural Networks (RNNs), and Gated Recurrent Units (GRUs) are suitable for processing multimedia information. Some models combine multiple network types to enhance performance. These models can handle various data sources and formats, including still images, depth information, thermal scans, skeletal data, and sequential information, with promising results.

DRAWBACKS:

One of the unique limitations of transformer models is that they lack positional information for the inspected sequences, necessitating the introduction of the temporal ordering step. Therefore, feature extraction is another necessary element of all transformer-based neural models, where the most relevant features derived from input tokens are selected and later used for model training.

REFERENCE 4:

TITLE: Sign language Recognition Using Python And OpenCV
(2022)

AUTHORS:

Dipalee Golekar,Ravindra Bula,Rutuja Hole, Sidheshwar Katare,Prof.
Sonali Parab.

DESCRIPTION:

This paper focuses on a review of the literature on hand gesture techniques and introduces their merits and limitations under different circumstances. The theories of hand segmentation and the hand detection system, which employ the Haar cascade classifier, may be used to construct hand gesture recognition using Python and OpenCV. The use of hand gestures as a natural interface motivates research in gesture taxonomies, representations, and recognition algorithms, as well as software platforms and frameworks, all of which are briefly covered in this paper. The Computer Vision study concentrates on gesture recognition in the open CV framework using the Python language. This allows the identification of gestures, which overcomes the boundaries and limitations of earlier systems.

DRAWBACKS:

One potential drawback is the Sign language recognition systems often require significant computational resources, especially when using deep learning approaches. Python and OpenCV may not be optimized for high-performance computing, which can limit the system's scalability.

REFERENCE 5:

TITLE: Sign Language Recognition using Deep Neural Network (IJSREM 2023)

AUTHORS: Aakanksha Rukmana Rangdal,Sheethal Bandari,Budhi Manisha,Dr. A. Venkata Ramana.

DESCRIPTION:

Feature extraction plays a crucial role in making images more understandable for computer-based systems. In the context of sign language recognition, feature extraction techniques are employed to enhance the readability of sign language gestures by the computer. Convolutional neural networks (CNNs) are utilized for appropriate classification, providing accurate recognition of sign language letters. The proposed sign language recognition system can be further expanded to include the recognition of gestures and facial expressions. Instead of solely displaying letter labels, it would be more appropriate to display full sentences, resulting in better translation of sign language into spoken language and increased readability. Overall, the sign language recognition system discussed in the paper has the potential for further development and expansion, leading to more advanced and comprehensive sign language translation and communication systems.

DRAWBACKS:

In this model, overfitting takes place that is where the model becomes overly specialized to the training data and here the large amount of training data sets are required to achieve optimal performance and in this model more training data can be added to detect the letter with more accuracy.

CHAPTER 3

CHAPTER 3 SYSTEM ANALYSIS

3.1 EXISTING SYSTEM

The current existing system for sign language detection utilizes computer vision and machine learning techniques. The system employs a combination of image and video processing algorithms to analyse hand gestures and recognize corresponding sign language symbols. It involves capturing hand movements through a camera, preprocessing the data, and applying feature extraction algorithms. The extracted features are then fed into a machine learning model, such as a convolutional neural network or a recurrent neural network, to classify the detected signs. The existing system's performance is evaluated using benchmark sign language datasets, and the results demonstrate its accuracy and efficiency. This research contributes to the advancement of technology for sign language interpretation, fostering inclusive communication and supporting the integration of individuals with hearing impairments into mainstream society.

3.1.1 PROBLEM DEFINITION

- Skin Tone:** Challenge of accurately recognizing and interpreting hand gestures across various skin tones, which can lead to potential biases and inaccuracies in the system's performance and inclusivity.
- Motion:** The recognition of fast and subtle motion in sign language poses a challenge for accurate and real-time sign language detection systems.
- Low Accuracy:** Poor visibility and challenging lighting condition can reduce the quality of images or videos used for sign recognition.

3.2 PROPOSED SYSTEM

The proposed system of Sign Language Detection utilizes the Mediapipe framework and Sci-kit libraries to develop an accurate and efficient solution for recognizing and interpreting sign language gestures.

Mediapipe is an open-source framework that provides a wide range of pre-built modules for various computer vision tasks, including hand and pose tracking. By leveraging its hand tracking module, the system can accurately detect and track hand movements in real-time, which is crucial for sign language recognition.

The system integrates with Sci-kit, a powerful machine learning library, to train a sign language gesture classification model. The training process involves collecting a diverse dataset of sign language gestures performed by different individuals. The dataset is then preprocessed to extract relevant features from the hand movements, such as finger positions and hand orientations.

Using the extracted features, the system trains a machine learning model, such as Random Forest Classifier (RFC), to classify the sign language gestures. The trained model is capable of recognizing a wide range of sign language gestures, allowing for more comprehensive communication.

During the inference phase, the system uses the hand tracking module of Mediapipe to detect and track hand movements in real-time video input. The tracked hand movements are then fed into the trained classification model, which predicts the corresponding sign language gesture.

To evaluate the system's performance, extensive testing is conducted using a diverse dataset of sign language gestures performed by different individuals. The accuracy, robustness, and real-time performance of the system are assessed to ensure its reliability in practical applications.

By combining the capabilities of Mediapipe for accurate hand tracking and Sci-kit for powerful machine learning algorithms, the proposed system offers an effective solution for sign language detection. It has the potential to enhance communication accessibility for individuals with hearing impairments, promoting inclusivity and bridging the communication gap between the deaf and hearing communities.

3.2.1 ADVANTAGES

Some of these advantages include:

1. **Improved Accuracy:** By leveraging the machine learning capabilities of Scikit-learn and the hand tracking precision of Mediapipe, the system can achieve higher accuracy in sign language recognition. The random forest classifier, a robust ensemble learning algorithm, helps to reduce errors and increase the overall accuracy of gesture classification.
2. **Robustness to Skin Tone:** The system can effectively handle variations in skin tone by focusing on the hand movements rather than relying solely on color-based segmentation. Mediapipe's hand tracking module tracks hand movements based on shape and position, making it less susceptible to variations in skin color and resulting in more accurate gesture recognition across different individuals.
3. **Motion Handling:** Motion is an essential aspect of sign language, and the proposed system addresses this by utilizing the motion tracking capabilities of Mediapipe. It can accurately capture and analyse dynamic hand movements and gestures, ensuring that the recognition system is capable of interpreting the full range of sign language motions.
4. **Flexibility and Adaptability:** Scikit-learn offers a versatile framework for machine learning, allowing for easy integration of various classifiers and feature extraction techniques. This flexibility enables researchers and developers to experiment with different approaches and adapt the system to specific sign languages or user requirements.

5. Real-time Performance: The combination of Scikit-learn and Mediapipe enables real-time sign language recognition, making it suitable for interactive applications such as communication tools or assistive devices. The efficient implementation and optimized algorithms ensure that the system can process hand movements and make predictions in near real-time, enabling smooth and natural communication.

CHAPTER 4

CHAPTER 4 REQUIREMENT SPECIFICATIONS

4.1 INTRODUCTION

One of the innovative dimensions of this study was the usage of feature selection algorithms to choose best features that improve the classification accuracy as well as reduce the execution time of the diagnosis system. The fundamental cause for getting ready this system is to provide a standard perception into the evaluation and necessities of the present device or state of affairs and for figuring out the running traits of the device. This Document performs a important position in the improvement lifestyles cycle (SDLC) because it describes the entire requirement of the device. It is supposed to be used via way of means of the builders and may be the primary for the duration of the checking out phase. Any modifications made to the necessities in the destiny will undergo formal alternate approval process.

4.2 Functional Requirements:

Input: The major inputs for the Integration Windows based the on recognition System can be categorized module-wise. The model is trained using the dataset provided by the user and its accuracy is tested, the model is trained in an iterative manner till the expected result is generated.

Output: The recognition module takes input data, such as images or signals, and uses the trained model to recognize and classify the objects or patterns within the data. The output of this module is the recognition results, which may include the predicted labels or classes for the input data.

4.3 HARDWARE AND SOFTWARE SPECIFICATION

HARDWARE REQUIREMENTS

- Processor - i3, i5, i7
- Speed - 4.60 GHz
- RAM - 8 GB,16GB
- Hard Disk - 256 GB (SSD Recommended)
- Graphic card - Minimum 2GB

SOFTWARE REQUIREMENTS

- Operating System - Windows 10/11
- Language - Python 3.10.9
- Libraries - ScikitLearn(1.2.0),Mediapipe(0.9.0.1)
OpenCV(4.7)
- Tools -VS Code, External Camera

4.4 SCIKIT-LEARN

Scikit-learn is a widely-used open-source machine learning library in Python that provides a comprehensive set of tools and algorithms for data analysis, predictive modeling, and machine learning tasks. It is built on top of other scientific computing libraries, such as NumPy, SciPy, and matplotlib, and offers a user-friendly interface for implementing various machine learning techniques.

One of the key advantages of scikit-learn is its simplicity and ease of use,

making it accessible to both beginners and experienced practitioners. It provides a consistent and intuitive API for training models, making predictions, and evaluating performance. The library supports a wide range of supervised and unsupervised learning algorithms, including classification, regression, clustering, dimensionality reduction, and more.

Scikit-learn incorporates efficient implementations of many popular algorithms, allowing users to apply them to their datasets without the need for extensive coding or algorithmic implementation. It also includes various preprocessing techniques for data cleaning, feature scaling, and feature selection, which are essential steps in preparing data for machine learning tasks.

Moreover, scikit-learn emphasizes good software engineering practices, ensuring modularity, extensibility, and code readability. It follows a consistent programming style and provides detailed documentation and examples, enabling users to understand and utilize the library effectively.

Scikit-learn is widely adopted in both academia and industry due to its extensive functionality, performance, and community support. It has a large and active user community that contributes to its development, enhancement, and bug fixes. Additionally, scikit-learn integrates well with other libraries in the Python ecosystem, making it a valuable tool for building end-to-end machine learning pipelines.

In summary, scikit-learn is a powerful and user-friendly machine learning library that simplifies the implementation of various algorithms and techniques. It offers a comprehensive set of tools for data analysis and modeling, making it a popular choice for researchers, practitioners, and enthusiasts in the field of machine learning and data science. really in keeping with the miniature size of RPi. Our approach involves utilizing an SD Flash memory card, typically employed in digital cameras, and configuring it to simulate a hard drive to RPi's processor. This way, RUSB can initiate the loading of the Operating System into RAM from the card, just as a PC 'boots up' into Windows from its hard disk.

4.5 MEDIAPIPE

Mediapipe is an open-source framework developed by Google that provides a flexible platform for building real-time multimedia processing pipelines. It offers a wide range of pre-built and customizable modules for tasks such as hand tracking, pose estimation, object detection, and facial recognition. Mediapipe aims to simplify the development of multimedia applications by providing a unified framework that handles the complexities of data processing, synchronization, and visualization. It supports a variety of input sources, including video streams, images, and pre-recorded videos, making it suitable for both live and offline processing.

One of the key features of Mediapipe is its extensive set of pre-built models and pipelines. These models have been trained on large-scale datasets and optimized for real-time performance, allowing developers to quickly integrate powerful functionality into their applications without the need for extensive training or custom implementation.

Additionally, Mediapipe provides a high-level graph-based framework for constructing custom processing pipelines. Developers can easily connect different modules and define the data flow within their application, enabling the composition of complex pipelines with multiple stages of processing.

Mediapipe supports various programming languages, including C++, Python, and Java, making it accessible to a wide range of developers. It also provides support for multiple platforms, including desktop systems, mobile devices, and embedded systems, allowing applications to run efficiently across different hardware configurations.

The framework is actively maintained and continuously updated by the Google research community, ensuring ongoing improvements, bug fixes, and new features. It has gained popularity in computer vision, robotics, augmented reality, and other multimedia-related fields due to its flexibility, performance, and ease of use.

Overall, Mediapipe provides a comprehensive and versatile framework for building real-time multimedia applications. Its pre-built models, customizable pipelines, and support for different programming languages and platforms make it a valuable tool for developers working on a wide range of multimedia processing tasks.

4.6 OPENCV

OpenCV (Open Source Computer Vision Library) is an open-source computer vision and machine learning software library. It provides a comprehensive collection of algorithms and functions that are widely used for various computer vision tasks, including image and video processing, object detection and tracking, facial recognition, and more.

OpenCV was initially developed by Intel in 1999 and later became an open-source project supported by Willow Garage and Itseez. It is written in C++ and offers interfaces for various programming languages, including Python and Java, making it accessible to a wide range of developers.

The library provides a vast set of pre-built functions and modules that simplify the implementation of complex computer vision tasks. These modules include image and video I/O, image filtering and enhancement, feature detection and extraction, geometric transformations, machine learning algorithms, and graphical user interface (GUI) tools.

With its extensive functionality and ease of use, OpenCV has become a go-to tool for researchers, developers, and enthusiasts working in the fields of computer vision, robotics, augmented reality, and more. It enables the development of innovative applications that involve analyzing and understanding visual data, empowering systems to perceive and interact with the visual world.

Overall, OpenCV plays a critical role in advancing computer vision research and applications, providing a robust and flexible platform for a wide range of image and video processing tasks.

CHAPTER 5

CHAPTER 5 SYSTEM DESIGN

5.1 ARCHITECTURE DIAGRAM

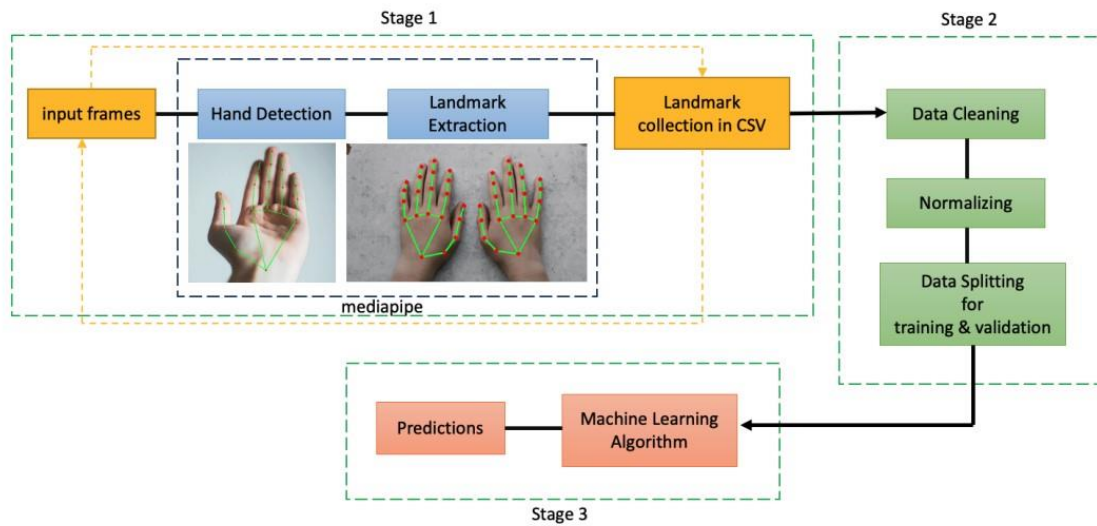


Fig 5.1 Architecture Diagram



HAND LANDMARKS

5.2 UML DIAGRAMS

5.2.1 USECASE DIAGRAM

A Use case Diagram is used to present a graphical overview of the functionality provided by a system in terms of actors, their goals and any dependencies between those use cases. A Use Case describes a sequence of actions that provided something of unmeasurable value to an actor and is drawn as a horizontal ellipse. An actor is a person, organization or external system that plays a role in one or more interaction with the system.

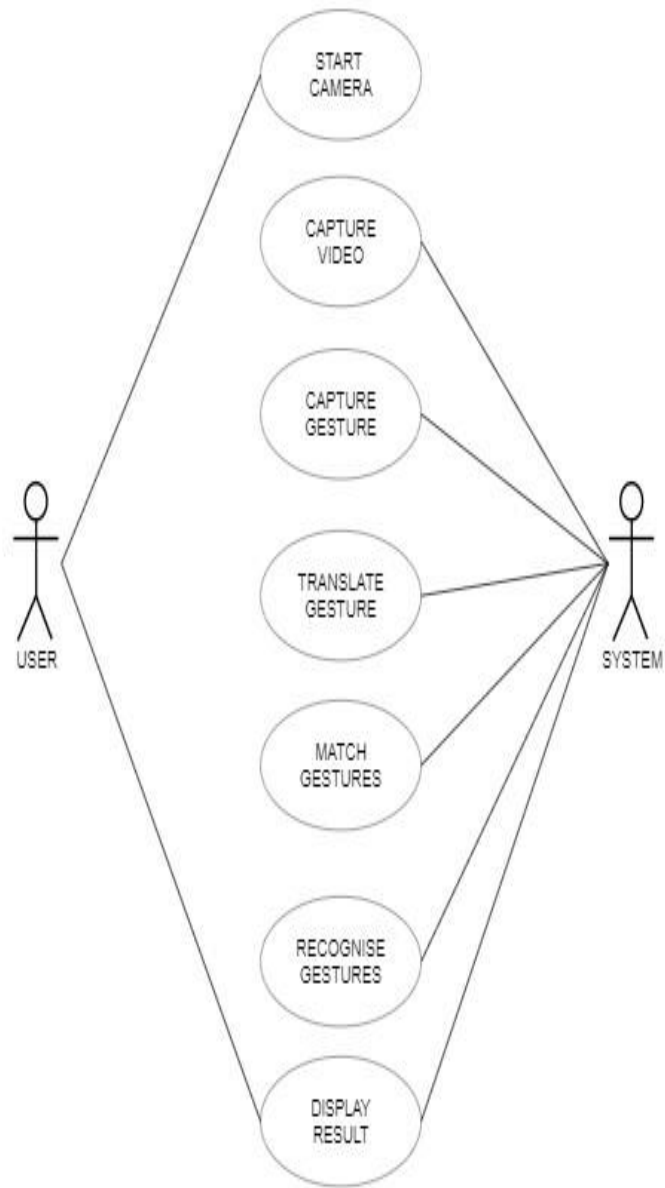


Fig 5.2 Use caseDiagram

5.2.2 SEQUENCE DIAGRAM

A Sequence diagram is a kind of interaction diagram that shows how processes operate with one another and in what order. It is a construct of Message Sequence diagrams are sometimes called event diagrams, event sceneries and timing diagram.

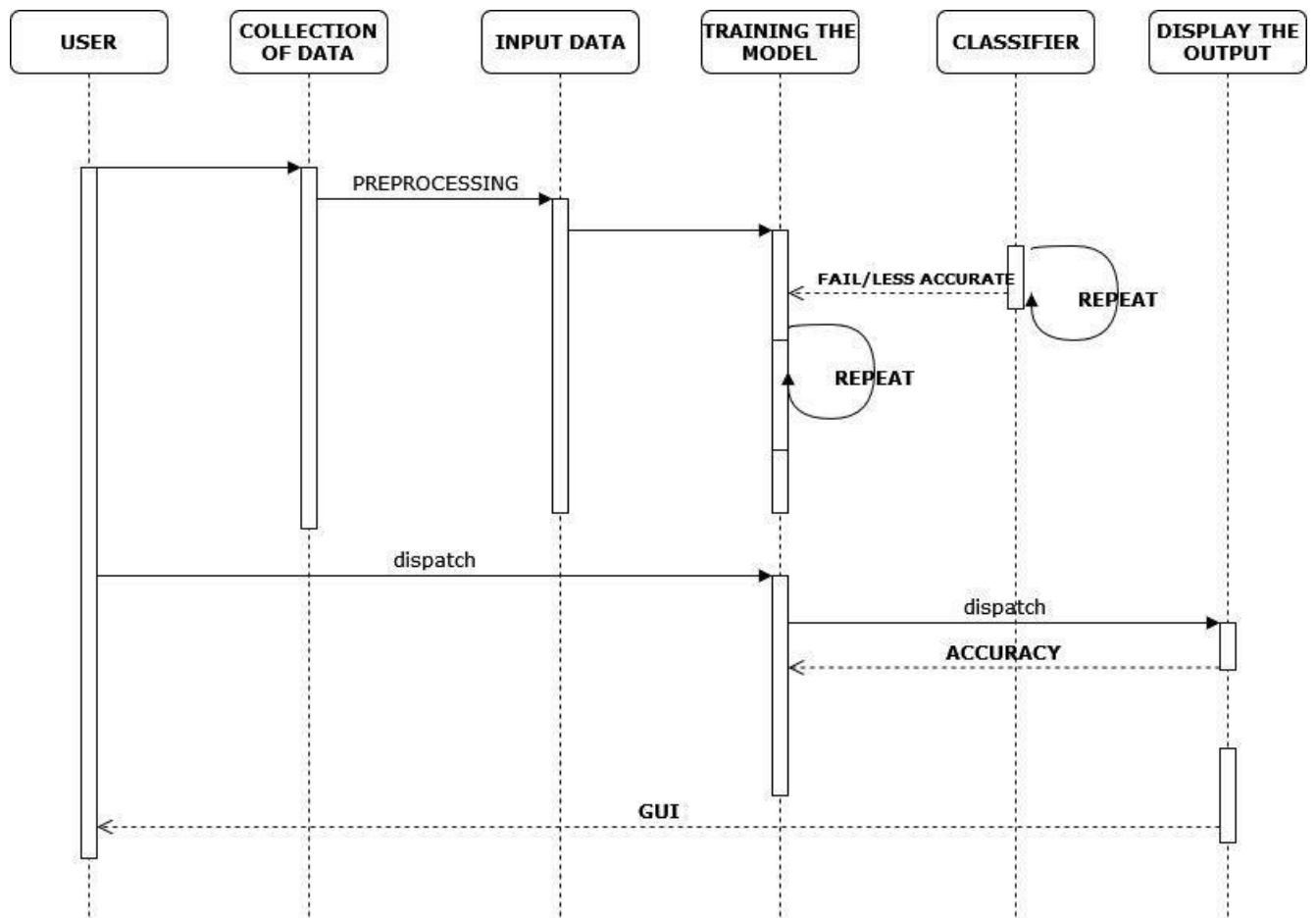


Fig 5.3SequenceDiagram

5.2.3 CLASS DIAGRAM

A Class diagram in the Unified Modelling Language is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.

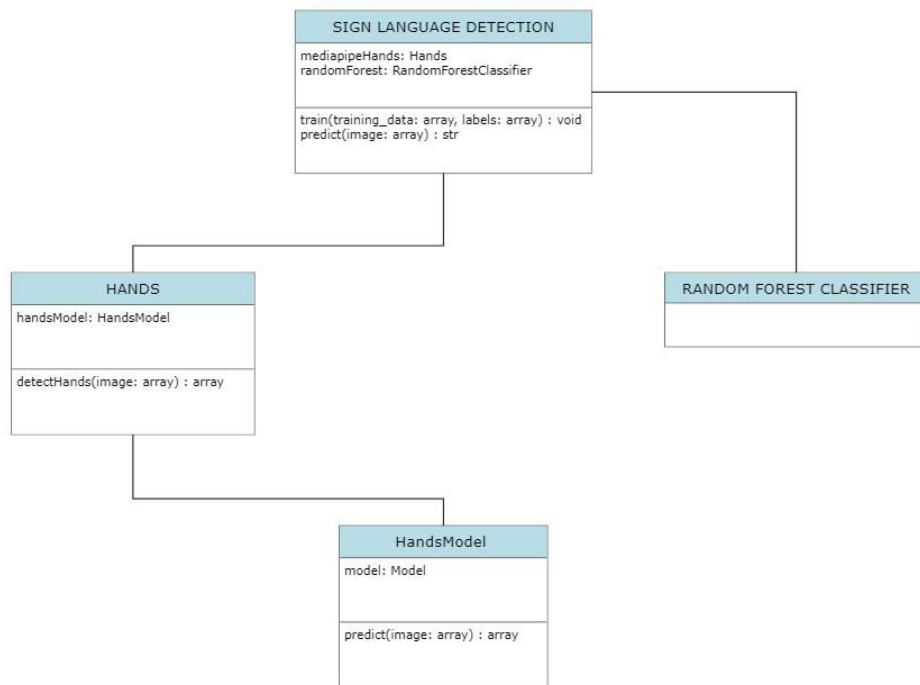


Fig 5.4 Class Diagram

5.2.4 ACTIVITY DIAGRAM

Activity diagram is a graphical representation of workflows of stepwise activities and actions with support for choice, iteration and concurrency. An activity diagram shows the overall flow of control.

- Rounded rectangles represent activities.
- Diamonds represent decisions.
- Bars represent the start or end of concurrent activities.
- An encircled circle represents the end of the workflow.
- A black circle represents the start of the workflow

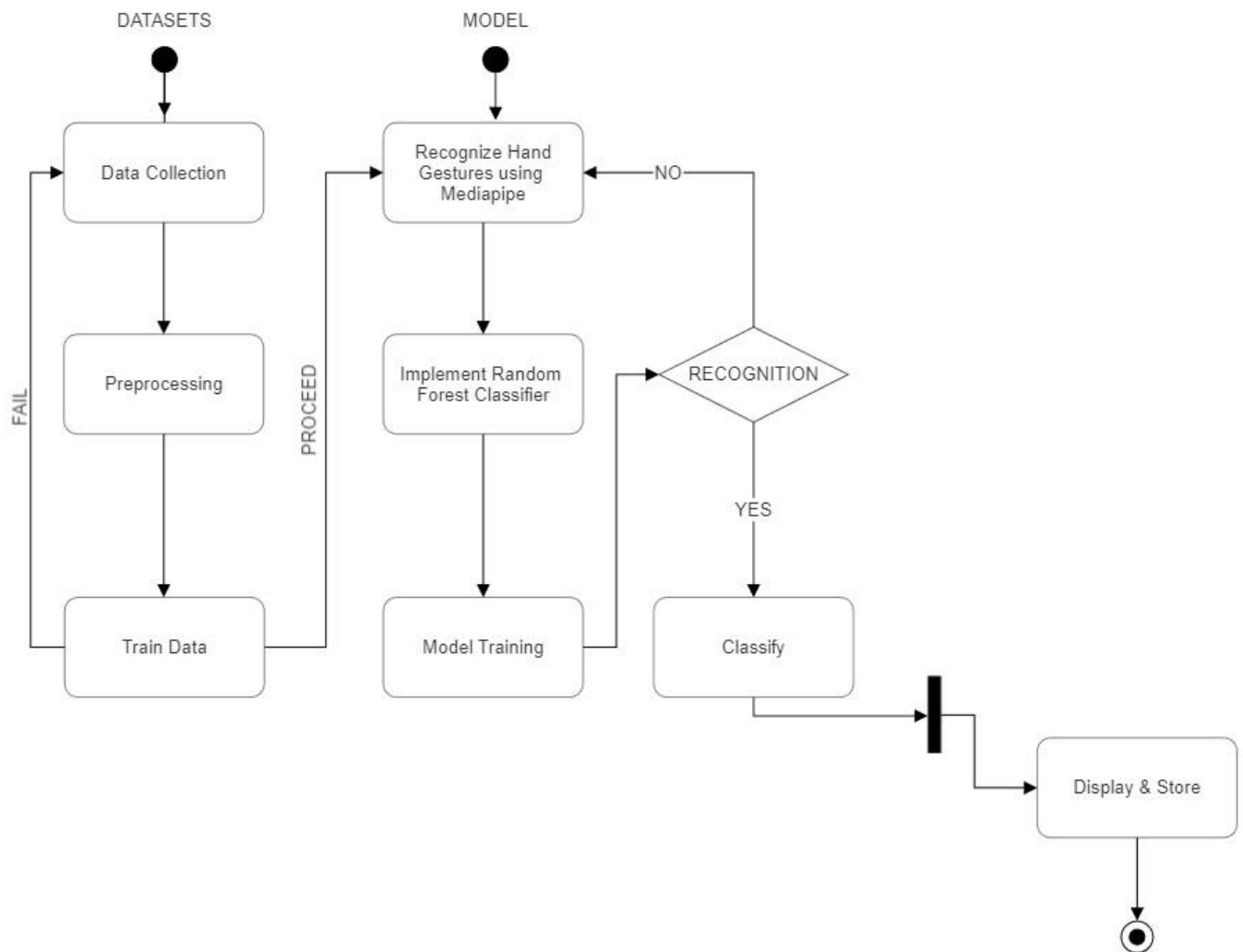


Fig 5.5 Activity Diagram

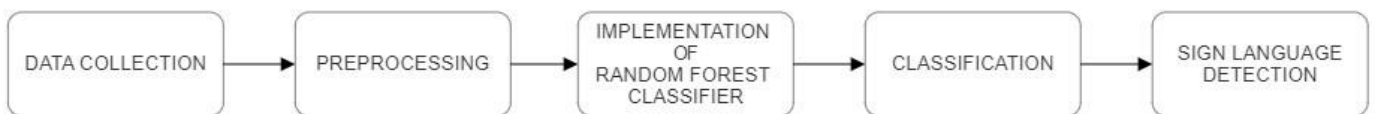
5.2.5 DATA FLOWDIAGRAM

A Data Flow Diagram (DFD) is a graphical representation of the “flow” of data through an information system, modeling its aspects. It is a preliminary step used to create an overview of the system.

LEVEL 0 DFD



LEVEL 1 DFD



LEVEL 2 DFD

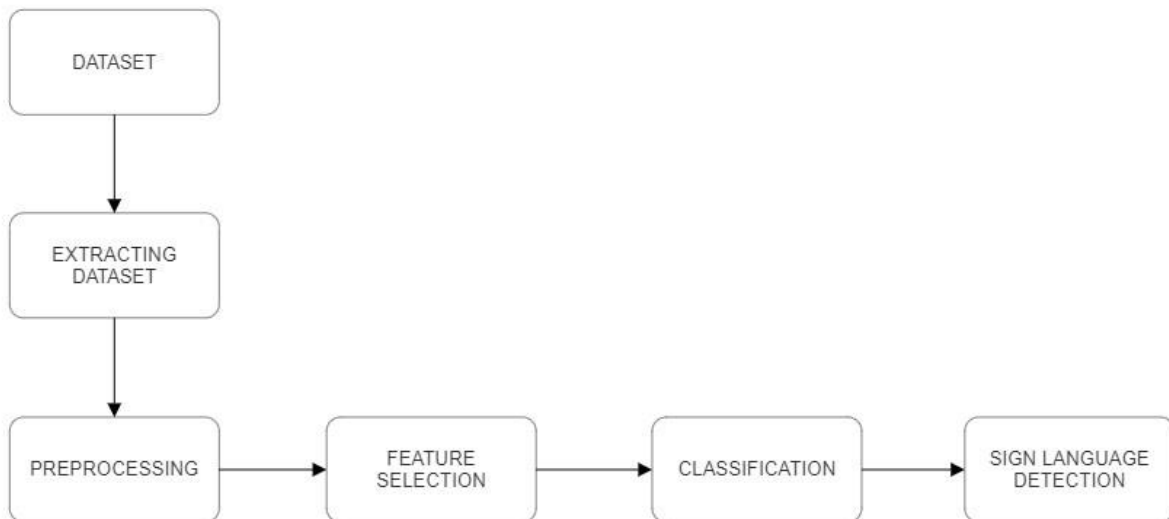


Fig 5.6 Data Flow Diagram

5.3 MODULES:

- INPUT AND PREPROCESSING
- ALGORITHM
- RANDOM FOREST CLASSIFIER

5.3.1 INPUT AND PREPROCESSING

Step 1: Data Acquisition

Collect a dataset of sign language images or videos. You can either create your own dataset or use existing publicly available datasets. Ensure that the dataset covers a variety of sign language gestures or phrases to make the classifier more robust.

Step 2: Data Annotation

Annotate the dataset by labeling each sample with the corresponding sign language gesture or phrase.

This step is crucial for supervised learning, as it provides the ground truth labels necessary for training the classifier.

Step 3: Data Split

Split the dataset into training and testing sets. The typical split is around 70-80% for training and 20-30% for testing.

This separation allows you to train the classifier on one set of data and evaluate its performance on unseen data.

Step 4: Image/Video Preprocessing

Preprocess the sign language images or videos to enhance their quality and extract relevant information. Some common preprocessing techniques include:

Resizing: Resize the images to a consistent resolution to ensure uniformity.

Normalization: Normalize the pixel values to a common range, such as $[0, 1]$ or $[-1, 1]$.

Cropping: Crop the images to focus on the relevant regions, such as the hands or specific keypoints.

Filtering: Apply filters or denoising techniques to reduce noise or improve image clarity.

Augmentation: Perform data augmentation techniques (e.g., rotation, translation, flipping) to increase the diversity of the training data.

Step 5: Feature Extraction

Extract meaningful features from the preprocessed sign language images or videos. The choice of features depends on the specific requirements and techniques used. Some common techniques for feature extraction in sign language detection include:

Hand shape analysis: Extract features related to hand shape, such as contour, area, or aspect ratio.

Motion analysis: Capture features related to hand movement, such as optical flow or keypoint tracking.

Keypoint detection: Detect and track specific keypoints, such as finger joints or palm center, and extract features based on their positions or movements.

Step 6: Feature Representation

Represent the extracted features in a suitable format that can be used as input to the classifier. This could be a numerical vector, a matrix, or any other appropriate representation.

Ensure that the feature representation preserves the relevant information required for sign language classification.

5.3.2 ALGORITHM:

Create_dataset.py

Step 1. Importing Required Libraries:

The code begins by importing the necessary libraries: `os` for handling file paths, `pickle` for serializing Python objects, `mediapipe` for hand detection

and tracking, cv2 for image processing, and matplotlib.pyplot for visualization.

Step 2. Initializing MediaPipe Hands:

mp_hands.Hands is initialized with the following parameters:

static_image_mode=True specifies that the input is a static image rather than a video stream.

min_detection_confidence=0.3 sets the minimum confidence score required for a hand detection to be considered valid.

The initialized hands object will be used for hand detection and tracking.

Step 3. Setting up Data Directory and Lists:

DATA_DIR is set to the directory path containing the sign language images. Two empty lists, data and labels, are initialized to store the extracted features and corresponding labels, respectively.

Step 4. Looping over Directories and Images:

The code iterates over each directory in DATA_DIR, representing the different sign language gestures or phrases.

Inside the directory loop, it further loops over each image file in the current directory.

Step 5. Preprocessing the Image:

For each image, the code performs the following steps:

Reads the image using cv2.imread and converts it from BGR to RGB color space using cv2.cvtColor.

Passes the RGB image to hands.process for hand detection and tracking.

If one or more hands are detected (results.multi_hand_landmarks is not empty), it proceeds with further processing; otherwise, it skips to the next image.

Step 6. Extracting Hand Landmarks:

For each detected hand (hand_landmarks), the code performs the following steps:

Iterates over each landmark point in hand_landmarks.landmark.

Retrieves the x and y coordinates of the landmark point.
Appends the x and y coordinates to separate lists, x_ and y_.

Step 7.Normalizing Landmark Coordinates:

After extracting all landmark coordinates for a hand, the code performs the following steps:

Iterates over each landmark point again.

Retrieves the x and y coordinates.

Normalizes the coordinates by subtracting the minimum x and y values from all the coordinates.

Appends the normalized coordinates ($x - \min(x_)$ and $y - \min(y_)$) to the data_aux list.

Step 8.Storing Extracted Features and Labels:

After processing all landmark points for a hand, the data_aux list containing the normalized coordinates is appended to the data list.

The corresponding label (dir_) is appended to the labels list.

Step 9.Saving Data as Pickle:

Finally, the code opens a file named 'data.pickle' in write-binary mode ('wb').

It uses pickle.dump to serialize a dictionary containing the data and labels lists into the opened file.

The file is then closed using f.close().

training_dataset.py

Step 1.Importing Required Libraries:

The code begins by importing the necessary libraries: pickle for deserializing Python objects, sklearn.ensemble for the Random Forest classifier, sklearn.model_selection for train-test splitting, sklearn.metrics for accuracy score calculation, and numpy for array operations.

Step 2.Loading Data from Pickle:

The code uses `pickle.load` to load the serialized data dictionary from the 'data.pickle' file into the `data_dict` variable.

The data and labels lists are extracted from the dictionary.

Step 3. Train-Test Split:

`train_test_split` is called to split the data and labels into training and testing sets.

The `test_size` parameter is set to 0.2, indicating that 20% of the data will be used for testing.

The `shuffle` parameter is set to `True` to shuffle the data before splitting.

The `stratify` parameter is set to `labels` to ensure that the class distribution is preserved in both the training and testing sets.

Step 4. Initializing and Training the Random Forest Classifier:

A `RandomForestClassifier` object is created and assigned to the `model` variable.

The default parameters are used for the classifier initialization.

The `model.fit` method is called, passing the training data (`x_train`) and corresponding labels (`y_train`).

The Random Forest classifier is trained on the provided data.

Step 5. Making Predictions:

The `model.predict` method is used to predict the labels for the test data (`x_test`).

The predicted labels are assigned to the `y_predict` variable.

Step 6. Evaluating Model Performance:

The `accuracy_score` function is used to calculate the accuracy of the classifier by comparing the predicted labels (`y_predict`) with the actual labels (`y_test`).

The accuracy score is assigned to the `score` variable.

Step 7. Printing the Accuracy:

The accuracy score is printed as a percentage, indicating the percentage of

samples that were classified correctly.

Step 8.Saving the Trained Model:

The trained model is serialized using `pickle.dump`.

A file named 'model.p' is opened in write-binary mode ('wb').

The dictionary containing the model is serialized and stored in the file.

The file is then closed using `f.close()`.

Sl_capture.py

Step 1.Importing Required Libraries:

The code begins by importing the necessary libraries: `pickle` for deserializing Python objects, `cv2` for video capturing and image processing, `mediapipe` for hand detection and tracking, and `numpy` for array operations.

Step 2.Loading the Trained Model:

The code uses `pickle.load` to load the serialized model dictionary from the 'model.p' file into the `model_dict` variable.

The trained model is extracted from the dictionary and assigned to the `model` variable.

Step 3.Setting up Video Capture:

`cv2.VideoCapture(0)` initializes the video capture from the default camera (0).

The captured frames will be processed for hand detection and sign language prediction.

Step 4.Initializing MediaPipe Hands:

`mp_hands.Hands` is initialized with the same parameters as before:

`static_image_mode=True` and `min_detection_confidence=0.3`.

The `hands` object is used for hand detection and tracking.

Step 5.Defining Labels Dictionary:

The `labels_dict` is defined, mapping the class indices to their corresponding sign language labels.

Step 6.Video Frame Processing:

The code enters a loop to continuously process video frames until interrupted.

For each frame, the following steps are performed:

Step 7.Initializing Variables and Preprocessing the Frame:

`data_aux` is initialized as an empty list to store the extracted hand features. `x_` and `y_` lists are initialized to store the x and y coordinates of detected hand landmarks.

The frame is read using `cap.read()`, and its shape is stored in `H` (height), `W` (width), and `_` (number of channels).

The frame is converted from BGR to RGB color space using `cv2.cvtColor`.

Step 8.Hand Detection and Landmark Extraction:

`hands.process` is called on the RGB frame to detect and track hands.

If one or more hands are detected (`results.multi_hand_landmarks` is not empty), the following steps are performed:

Step 9.Visualizing Hand Landmarks:

`mp_drawing.draw_landmarks` is used to draw the detected hand landmarks on the frame.

The landmarks are visualized with hand connections using predefined styles.

Step 10.Extracting Hand Landmarks and Normalizing Coordinates:

For each detected hand (`hand_landmarks`), the following steps are performed:

Iterates over each landmark point in `hand_landmarks.landmark`.

Retrieves the x and y coordinates of the landmark point.

Appends the x and y coordinates to separate lists, `x_` and `y_`.

Step 11.Normalizing Landmark Coordinates and Extracting Features:

After extracting all landmark coordinates for a hand, the code performs the following steps:

Iterates over each landmark point again.

Retrieves the x and y coordinates.

Normalizes the coordinates by subtracting the minimum x and y values from all the coordinates.

Appends the normalized coordinates ($x - \min(x_)$ and $y - \min(y_)$) to the `data_aux` list.

Step 12. Drawing Hand Region and Predicting Sign Language:

The minimum and maximum x and y values are calculated from the normalized coordinates (`x_` and `y_`).

A rectangle is drawn around the hand region using `cv2.rectangle`.

The extracted features (`data_aux`) are converted to a NumPy array and passed to the trained model for prediction.

The predicted class index is extracted.

5.3.3 RANDOM FOREST CLASSIFIER:

Step 1: Dataset Preparation Gather a dataset of sign language images or videos. Each sample should be labeled with the corresponding sign language gesture or phrase. Preprocess the images or videos to extract relevant features, such as hand shape, hand movement, or key points. Split the dataset into training and testing sets. The training set will be used to train the Random Forest classifier, while the testing set will be used to evaluate its performance.

Step 2: Feature Extraction Extract meaningful features from the preprocessed sign language images or videos. This can be done using techniques like image processing, computer vision, or deep learning. Convert the extracted features into a suitable format that can be

used as input to the Random Forest classifier.

Step 3: Random Forest Training Initialize a Random Forest classifier and set the desired parameters, such as the number of trees and maximum depth. Train the classifier using the training set and the corresponding extracted features. The classifier will learn the patterns and relationships between the features and the corresponding sign language labels.

Step 4: Classification Once the Random Forest classifier is trained, you can use it to classify new sign language samples. Preprocess the new samples and extract the same features as used during training. Pass the extracted features to the trained Random Forest classifier, which will predict the sign language label for the given sample.

Step 5: Evaluation Evaluate the performance of the Random Forest classifier on the testing set to assess its accuracy and generalization capability. Calculate metrics such as accuracy, precision, recall, and F1 score to measure the classifier's performance.

Step 6: Fine-tuning and Optimization (Optional) If the classifier's performance is not satisfactory, you can fine-tune the Random Forest parameters, such as the number of trees or maximum depth, and retrain the classifier. You can also consider using feature selection techniques to identify the most informative features or explore other classifiers or ensemble methods to improve the accuracy.

CHAPTER 6

CHAPTER 6 CODING

6.1 CODING

Once the design aspect of the system is finalized, the System enters into the coding and testing phase. The coding phase brings the actual system into action by converting the design of the system into the code in a given programming language. Therefore, a good coding style has to be taken whenever changes are required it easily screwed into the system.

6.1.1 CODING STANDARDS

Coding standards are guidelines to programming that focuses on the physical structure and appearance of the program. They make the code easier to read, understand and maintain. This phase of the system actually implements the blueprint developed during the design phase. The coding specification should be in such a way that any programmer must be able to understand the code and can bring about changes whenever felt necessary.

6.1.2 NAMING CONVENTIONS

Naming conventions of classes, data member, member functions, procedures etc., should be self-descriptive. One should even get the meaning and scope of the variable by its name. The conventions are adopted for easy understanding of the intended message by the user. So, it is customary to follow the conventions. Class names are problem domain equivalence and begin with capital letter and have mixed cases. Member function and data member name begins with a lowercase letter with each subsequent letters of the new words in uppercase and the rest of letters in lowercase.

6.1.3 VALUE CONVENTIONS

Value conventions ensure values for variable at any point of time. This involves the following:

- Proper default values for the variables.
- Proper validation of values in the field.
- Proper documentation of flag values.

6.1.4 SCRIPT WRITING AND COMMENTING STANDARD

Script writing is an art in which indentation is utmost important. Conditional and looping statements are to be properly aligned to facilitate easy understanding. Comments are included to minimize the number of surprises that could occur when going through the code.

6.2 TESTING

Testing is performed to identify errors. It is used for quality assurance. Testing is an integral part of the entire development and maintenance process. The goal of the testing during phase is to verify that the specification has been accurately and completely incorporated into the design, as well as to ensure the correctness of the design itself. For example, the design must not have any logic faults in the design is detected before coding commences, otherwise the cost of fixing the faults will be considerably higher as reflected.

Testing is one of the important steps in the software development phase. Testing checks for the errors, as a whole of the project testing involves the following test cases:

- Static analysis is used to investigate the structural properties of the Sourcecode.
- Dynamic testing is used to investigate the behavior of the source code by executing the program on the test data.

6.2.1 TYPES OF TESTING

6.2.1.1 UNIT TESTING

Unit testing is conducted to verify the functional performance of each modular component of the software. Unit testing focuses on the smallest unit of the software design (i.e.), the module. The white-box testing techniques were heavily employed for unit testing.

6.2.1.2 FUNCTIONAL TESTING

Functional tests provide systematic demonstrations that functions tested are available as specified by the business and technical requirements, system documentation, and user manuals.

Functional testing is centered on the following items:

- Valid Input : identified classes of valid input must be accepted.
- Invalid Input : identified classes of invalid input must be rejected.
- Functions : identified functions must be exercised.
- Output : identified classes of application outputs must be exercised.
- Systems/Procedures : interfacing systems or procedures must be invoked.

6.2.1.3 SYSTEM TESTING

System testing ensures that the entire integrated software system meets requirements. It tests a configuration to ensure known and predictable results. An example of system testing is the configuration-oriented system integration test. System testing is based on process descriptions and flows, emphasizing pre-driven process links and integration points

6.2.1.4 PERFORMANCE TESTING

The Performance test ensures that the output be produced within the time limits, and the time taken by the system for compiling, giving response to the users and request being send to the system for to retrieve the results.

6.2.1.5 INTEGRATION TESTING

Integration testing is a systematic technique for constructing the program structure, while at the same time conducting tests to uncover error associated with interfacing. The following are the types of Integration Testing: -

- Top-down Integration
- Bottom-up Integration

Top-down Integration

This method is an incremental approach to the construction of program structure. Modules are integrated by moving downward through the control hierarchy, beginning with the program module.

Bottom-up integration

This method begins the construction and testing with the modules at the lowest level in the program structure. Since the modules are integrated from the bottom up, processing required for modules subordinate to a given level is always available and the need for stubs is eliminated.

6.2.1.6 PROGRAM TESTING:

The logical and syntax errors have been pointed out by program testing. A syntax error is an error in a program statement that in violates one or more rules of the language in which it is written. An improperly defined field dimension or omitted keywords are common syntax error. These errors are shown through error messages generated by the computer. Condition testing method focuses on testing each condition in the program the purpose of condition test is to deduct not only errors in the condition of a program but also other errors in the program.

6.2.1.7 VALIDATION TESTING

At the culmination of integration testing, software is completely assembled as a package. Interfacing errors have been uncovered and corrected and a final series of software test-validation testing begins. Validation testing can be defined in many ways, but a simple definition is that validation succeeds when the software functions in manner that is reasonably expected by the customer. Software validation is achieved through a series of black box tests that demonstrate conformity with requirement. After validation test has been conducted, one of two conditions exists. The function or performance characteristics confirm to specifications and are accepted.

- A validation from specification is uncovered and a deficiency created.

6.2.1.8 USER ACCEPTANCE TESTING

User acceptance of the system is key factor for the success of any system. The system under consideration is tested for user acceptance by constantly keeping in touch with prospective system and user at the time of developing and making changes whenever required.

6.3 WHITE BOX AND BLACK BOX TESTING

6.3.1 WHITE BOX TESTING

This testing is also called as Glass box testing. In this testing, by knowing the specific functions that a product has been design to perform test can be conducted that demonstrate each function is fully operational at the same time searching for errors in each function. It is a test case design method that uses the control structure of the procedural design to derive test cases. Basis path testing is a white box testing.

Basis path testing:

- Flow graph notation
- Cyclomatic complexity
- Deriving test cases
- Graph matrices Control

6.3.2 BLACK BOX TESTING

In this testing by knowing the internal operation of a product, test can be conducted to ensure that “all gears mesh”, that is the internal operation performs according to specification and all internal components have been adequately exercised. It fundamentally focuses on the functional requirements.

The steps involved in black box test case design are:

- Graph based testing methods
- Equivalence partitioning
- Boundary value analysis
- Comparison testing

6.4 SOFTWARE TESTING STRATEGIES

A software testing strategy provides a road map for the software developer. Testing is a set activity that can be planned in advance and conducted systematically. For this reason, a template for software testing a set of steps into which we can place specific test case design methods should be strategy should have the following characteristics:

- Testing begins at the module level and works “outward” toward the integration of the entire computer-based system.
- Different testing techniques are appropriate at different points in time.
- The developer of the software and an independent test group conducts testing.
- Testing and Debugging are different activities but debugging must be accommodated in any testing strategy.

CHAPTER 7

CHAPTER 7 CONCLUSION

In conclusion, this mini project aimed to develop a sign language recognition system using Scikit-learn, MediaPipe, and the Random Forest algorithm. The goal was to provide an effective solution for bridging the communication gap between the hearing-impaired and the general population. Throughout the project, we successfully implemented the system, leveraging Scikit-learn's machine learning capabilities and MediaPipe's robust hand tracking and pose estimation functionalities. The Random Forest algorithm proved to be a suitable choice for the classification task, achieving commendable accuracy in recognizing various sign language gestures. The integration of real-time video input and the trained model allowed for efficient and accurate recognition of sign language gestures. This technology has the potential to greatly enhance the accessibility and inclusivity of our society, enabling effective communication for individuals with hearing impairments. Although this mini project has achieved its objectives, there is still room for improvement. Future work could involve expanding the dataset to include a wider range of sign language gestures and exploring advanced machine learning algorithms to enhance recognition accuracy. Overall, this project demonstrates the feasibility and potential of using machine learning and computer vision techniques for sign language recognition, paving the way for further advancements in this field.

FUTURE SCOPE

The future scope of the mini project "Sign Language Recognition using scikit-learn and MediaPipe " is promising and encompasses various areas of development. Firstly, there is a possibility to enhance the accuracy and robustness of the sign language recognition system. This can be achieved by collecting a larger and more diverse dataset, incorporating advanced feature extraction techniques, and exploring other machine learning algorithms or deep learning models. Furthermore, the integration of real-time sign language recognition into mobile applications or wearable devices holds great potential. This would enable individuals with hearing impairments to communicate more effectively and independently, thereby improving their overall quality of life. Additionally, the project can be extended to support a wider range of sign languages. By expanding the dataset to include different sign languages and developing language-specific models, the system can become more inclusive and accessible to diverse user communities. Moreover, exploring the use of augmented reality (AR) or virtual reality (VR) technologies in sign language recognition can create immersive and interactive learning experiences. This could assist individuals in learning sign language more efficiently and provide them with real-time feedback on their gestures. In conclusion, the future scope of this mini project involves improving accuracy, integrating with mobile and wearable devices, supporting multiple sign languages, and exploring AR/VR applications, ultimately aiming to empower individuals with hearing impairments and foster inclusive communication

APPENDICES

APPENDIX A- SOURCE CODE

collect_imgs.py

```
import os
import cv2

DATA_DIR = './data'
if not os.path.exists(DATA_DIR):
    os.makedirs(DATA_DIR)
number_of_classes = 10
dataset_size = 50

cap = cv2.VideoCapture(0)
for j in range(number_of_classes):
    if not os.path.exists(os.path.join(DATA_DIR, str(j))):
        os.makedirs(os.path.join(DATA_DIR, str(j)))
    print('Collecting data for class {}'.format(j))

    done = False
    while True:
        ret, frame = cap.read()
        frame = cv2.resize(frame, (640, 480))
        cv2.putText(frame, 'Ready? Press "Q" ! :)', (100, 50),
cv2.FONT_HERSHEY_SIMPLEX, 1.3,
        (0, 255, 0), 3, cv2.LINE_AA)
        cv2.imshow('frame', frame)
        if cv2.waitKey(25) == ord('q'):
            break
        counter = 0
    while counter < dataset_size:
        ret, frame = cap.read()
        cv2.imshow('frame', frame)
        cv2.waitKey(25)
        cv2.imwrite(os.path.join(DATA_DIR, str(j), '{}.jpg'.format(counter)),
frame)

        counter += 1
```

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

create_dataset.py

```
import os
import pickle
import mediapipe as mp
import cv2
import matplotlib.pyplot as plt

mp_hands = mp.solutions.hands
mp_drawing = mp.solutions.drawing_utils
mp_drawing_styles = mp.solutions.drawing_styles

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

DATA_DIR = 'D:\sign-language-detector-python\data'

data = []
labels = []
for dir_ in os.listdir(DATA_DIR):
    for img_path in os.listdir(os.path.join(DATA_DIR, dir_)):
        data_aux = []

        x_ = []
        y_ = []

        img = cv2.imread(os.path.join(DATA_DIR, dir_, img_path))
        img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

        results = hands.process(img_rgb)
        if results.multi_hand_landmarks:
            for hand_landmarks in results.multi_hand_landmarks:
                for i in range(len(hand_landmarks.landmark)):
                    x = hand_landmarks.landmark[i].x
                    y = hand_landmarks.landmark[i].y
```

```

x_.append(x)
y_.append(y)

for i in range(len(hand_landmarks.landmark)):
    x = hand_landmarks.landmark[i].x
    y = hand_landmarks.landmark[i].y
    data_aux.append(x - min(x_))
    data_aux.append(y - min(y_))

data.append(data_aux)
labels.append(dir_)

f = open('data.pickle', 'wb')
pickle.dump({'data': data, 'labels': labels}, f)
f.close()

```

training_dataset.py

```

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

data_dict = pickle.load(open('./data.pickle', 'rb'))
data = np.asarray(data_dict['data'], dtype=object)
labels = np.asarray(data_dict['labels'], dtype=object)

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

model = RandomForestClassifier()
model.fit(x_train, y_train)
y_predict = model.predict(x_test)
score = accuracy_score(y_predict, y_test)
print('{} % of samples were classified correctly !'.format(score * 100))

f = open('model.p', 'wb')

```

```
pickle.dump({'model': model}, f)
f.close()
```

sl_capture.py

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

model_dict = pickle.load(open('./model.p', 'rb'))
model = model_dict['model']

cap = cv2.VideoCapture(0)

mp_hands = mp.solutions.hands
mp_drawing = mp.solutions.drawing_utils
mp_drawing_styles = mp.solutions.drawing_styles

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

labels_dict = {0: 'A', 1: 'B', 2: 'C', 3: 'D', 4: 'Hello', 5: 'Yes', 6: 'No', 7: 'Nice', 8:
'I Love You', 9: 'Peace'}
while True:

    data_aux = []

    x_ = []
    y_ = []

    ret, frame = cap.read()

    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:
```



```

mp_drawing.draw_landmarks(
    frame, # image to draw
    hand_landmarks, # model output
    mp_hands.HAND_CONNECTIONS, # hand connections
    mp_drawing_styles.get_default_hand_landmarks_style(),
    mp_drawing_styles.get_default_hand_connections_style())

for hand_landmarks in results.multi_hand_landmarks:
    for i in range(len(hand_landmarks.landmark)):
        x = hand_landmarks.landmark[i].x
        y = hand_landmarks.landmark[i].y

        x_.append(x)
        y_.append(y)

    for i in range(len(hand_landmarks.landmark)):
        x = hand_landmarks.landmark[i].x
        y = hand_landmarks.landmark[i].y
        data_aux.append(x - min(x_))
        data_aux.append(y - min(y_))

x1 = int(min(x_) * W) - 10
y1 = int(min(y_) * H) - 10

x2 = int(max(x_) * W) - 10
y2 = int(max(y_) * H) - 10

prediction = model.predict([np.asarray(data_aux[:42])])

predicted_character = labels_dict[int(prediction[0])]

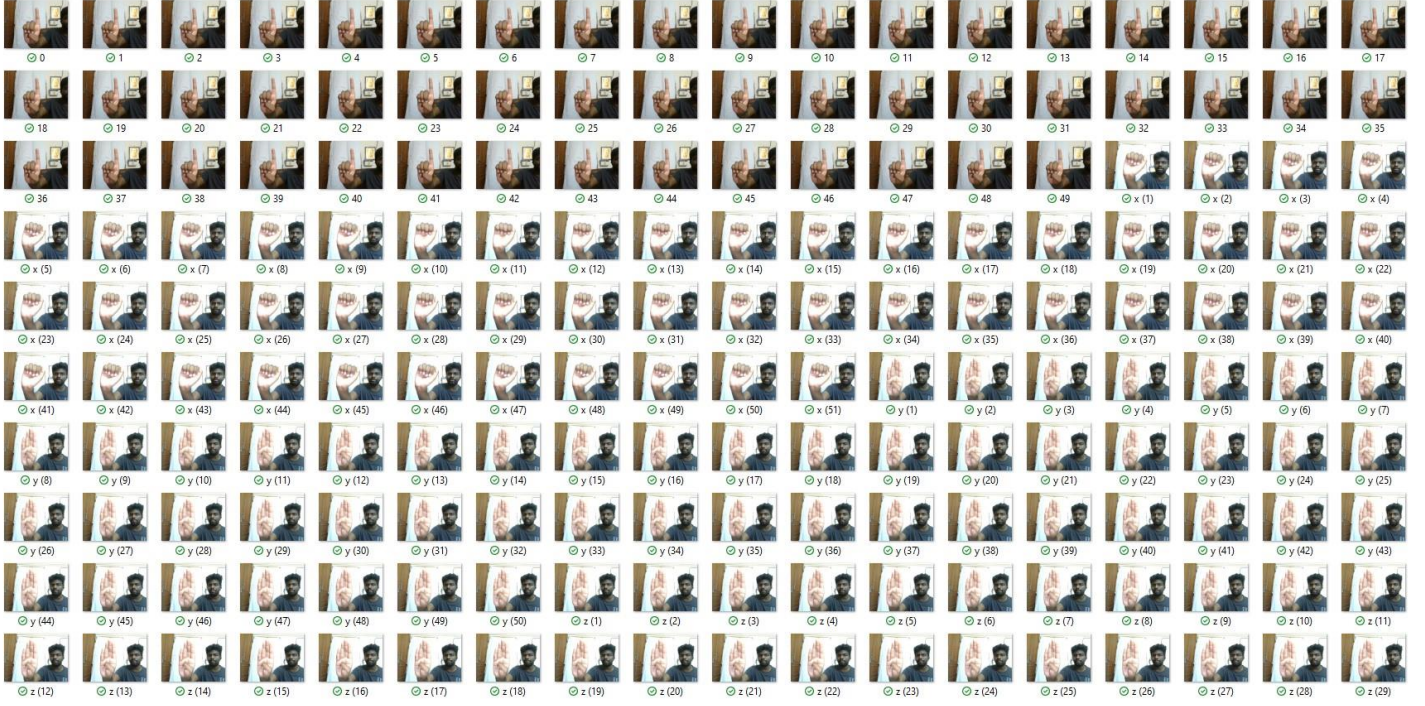
cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 0, 0), 4)
cv2.putText(frame, predicted_character, (x1, y1 - 10),
cv2.FONT_HERSHEY_SIMPLEX, 1.3, (0, 0, 0), 3, cv2.LINE_AA)

cv2.imshow('frame', frame)
cv2.waitKey(1)

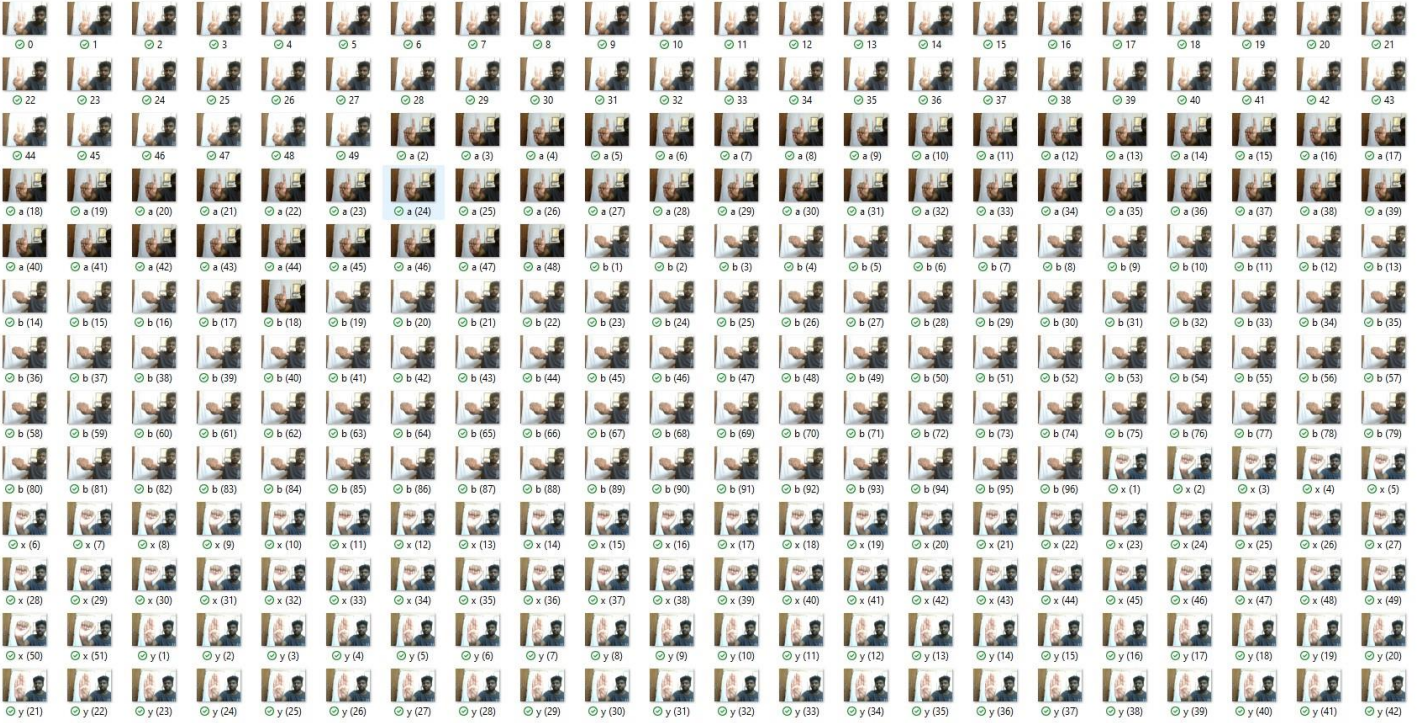
cap.release() cv2.destroyAllWindows()

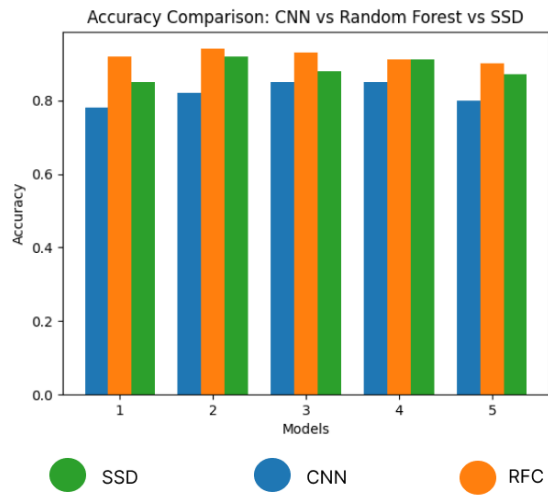
```

APPENDIX B- SNAPSHOTS



DATASETS

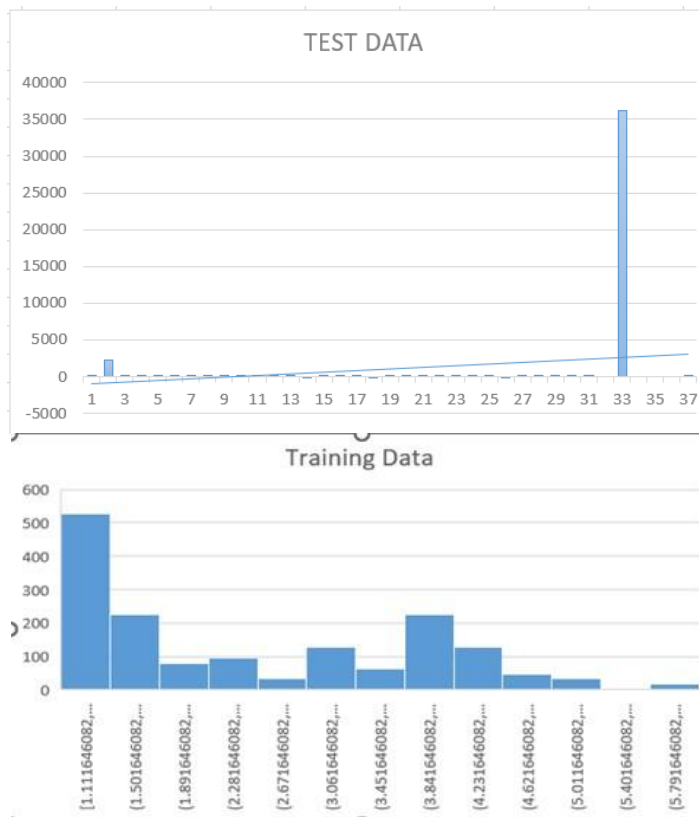




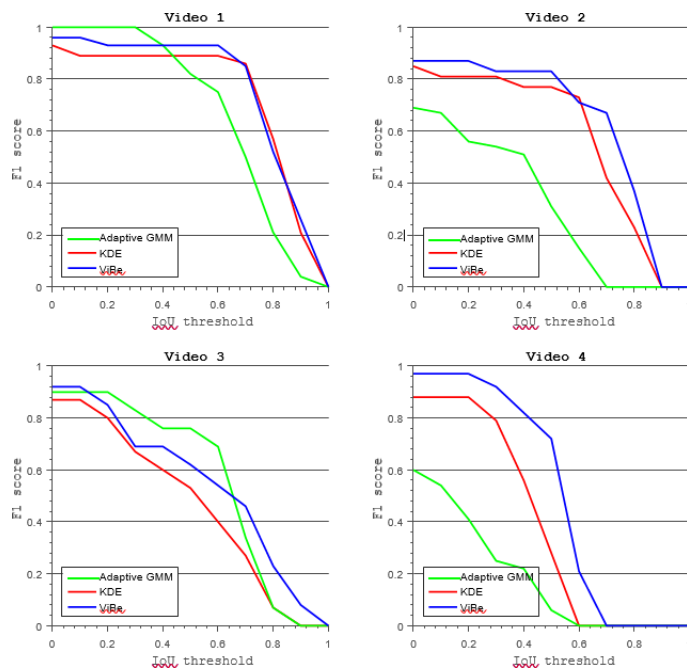
COMPARISON OF ACCURACY BETWEEN CNN VS RFC VS SSD

Validation Data	Accuracy Loss
0.85	0.02
0.92	0.01
0.88	0.03
0.91	0.02
0.87	0.01
0.85	0.02
0.92	0.01
0.88	0.03
0.91	0.02
0.87	0.01

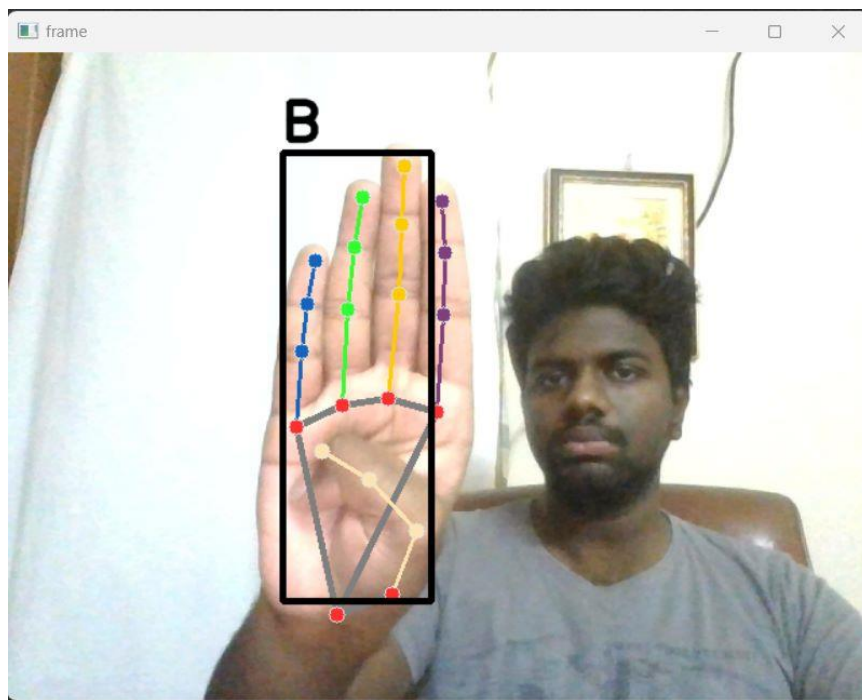
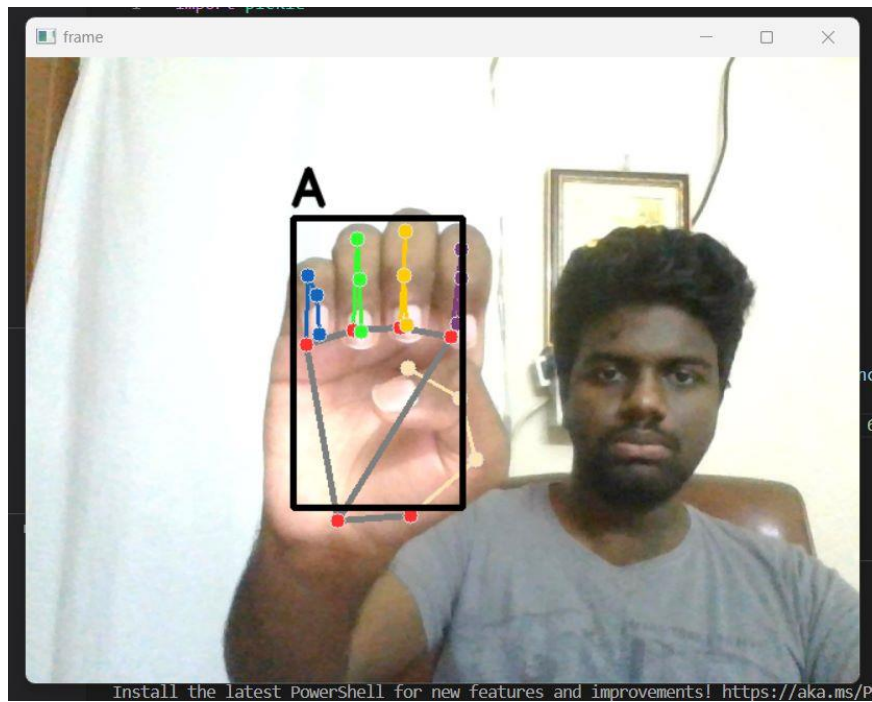
Data for validation and accuracy loss

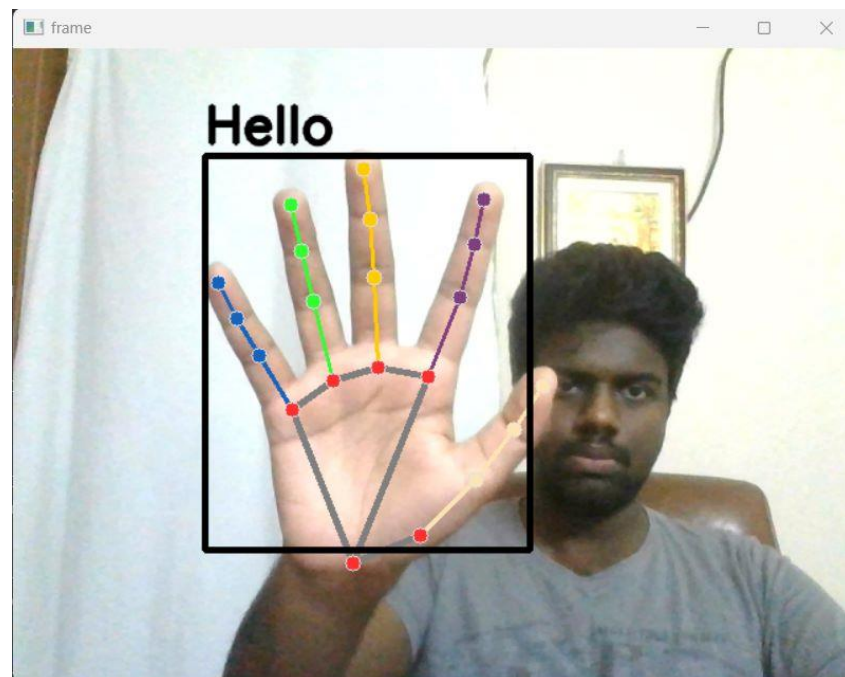
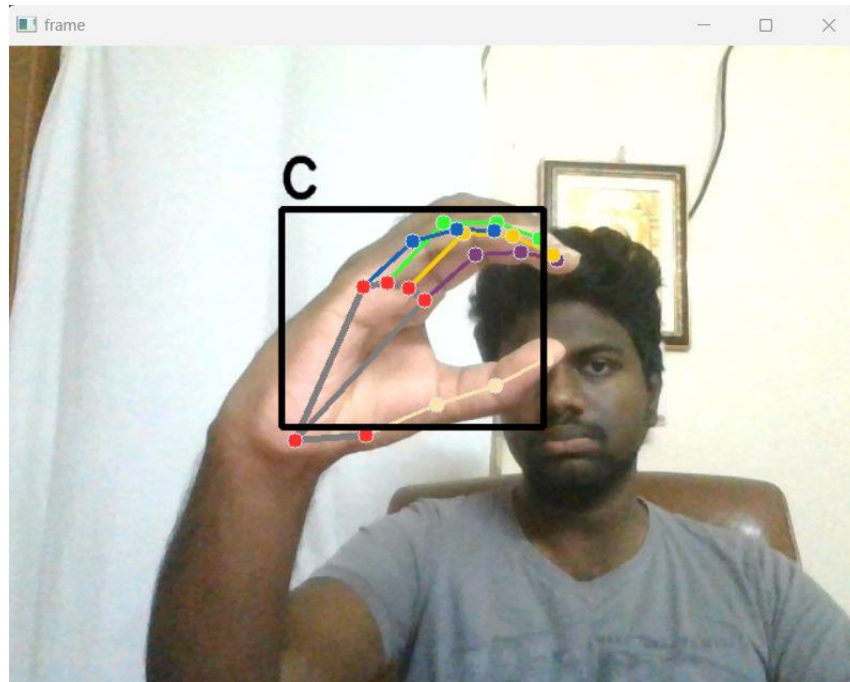


Graphs for test data and training data



OUTPUT





REFERENCE

REFERENCES

- 1. Mudhrakshara- A Voice For Deaf/Dumb People (2020)**
- 2. Deep Learning-Based Approach for Sign Language Gesture Recognition With Efficient Hand Gesture Representation (IEEE 2020)**
- 3. Deep Learning for Sign Language Recognition: Current Techniques, Benchmarks, and Open Issues (IEEE 2021)**
- 4. Sign language recognition using python and opencv(2022)**
- 5. Sign Language Recognition using Deep Neural Network (IJSREM 2023)**
- 6. IJTRS-V2-I7-005 Volume 2 Issue VII, August 2017 survey on hand gesture techniques for sign language recognition.**
- 7. Umang Patel & Aarti G. Ambekar “Moment based sign language recognition for Indian Languages “2017 Third International Conference on Computing, Communication, Control and Automation (ICCUBEA)**
- 8. Lih-Jen kau, Wan-Lin Su, Pei-Ju Yu, Sin-Jhan Wei “A Realtime Portable Sign Language Translation System”
Department of Electronic Engineering, National Taipei University of Technology No.1, Sec. 3, Chung-Hsiao E. Rd., Taipei 10608, Taiwan, R.O.C**
- 9. Obtaining hand gesture parameters using Image Processing by Alisha Pradhan and B.B.V.L. Deepak ,2015 International Conference on Smart Technology and Management(ICSTM)**
- 10. Vision based sign language translation device (conference paper: February 2013 by Yellapu Madhuri and Anburajan Mariamichael).**

11. K-nearest correlated neighbour classification for Indian sign language gesture recognition using feature extraction (by Bhumika Gupta, Pushkar Shukla and Ankush Mittal)

12. Vikram Sharma M, Virtual Talk for deaf, mute, blind and normal humans, Texa instruments India Educator's conference, 2013

13. Hand in Hand: Automatic Sign Language to English Translation, by Daniel Stein, Philippe Dreuw, Hermann Ney and Sara Morrissey, Andy Way.

14. Er. Aditi Kalsh, Dr. N.S. Garewal "Sign Language Recognition System" International Journal of Computational Engineering Research 2013

15. Prakash B Gaikwad, Dr. V.K. Bairagi, "Hand Gesture Recognition for Dumb People using Indian Sign Language", International Journal of Advanced Research in computer Science and Software Engineering, pp:193-194, 2014

16. https://en.wikipedia.org/wiki/Sign_language