Gesture-Recognition Design Documentation

Table of Contents

Introduction

Welcome to the documentation for the Gesture Recognition Library for C++. This document serves as a comprehensive guide to understanding and implementing the capabilities of the software system. The library is designed to facilitate gesture recognition in both images and live video streams, with a primary focus on MacOS compatibility. To learn more about running or using the library please refer to the manual and tutorial.

Motivation

Gesture recognition libraries in C++ are scarce and not many are beginner-friendly. One option for those who want to build a gesture recognition project in C++ is to implement their own application that tracks and identifies hand gestures, which is computation-heavy and can become mathematically complex. Another option is to leverage existing libraries but the C++ gesture recognition community is small compared to Python's gesture recognition community and thus not many resources are available. The libraries that do exist are often difficult to set up and poorly documented.

The goal of this project is to simplify gesture recognition tasks for programmers who want to integrate this feature into a bigger project. We define gesture recognition tasks as simply providing an image and returning a prediction of what the hand gesture in the image is. We have simplified the process to only a few lines of code that a programmer using this library will have to add to their project to implement gesture recognition. We want to make this library more widely accessible to the wider C++ community, especially those who are new to C++ and the field of gesture recognition.

System Overview

The Gesture Recognition Library is a C++ software solution developed to simplify the process of recognizing gestures in images and live video feeds. As of December 2023, the library has undergone testing and optimization for MacOS, ensuring seamless integration and performance on this platform.

Key Features

1. Efficient gesture recognition in images and live video streams.
2. Compatibility with MacOS, offering a tailored experience for users of this operating system.
3. Integration with Google's Mediapipe repository, configured for MacOS, to obtain hand landmarks.

Dependencies

1. **Google's Mediapipe Repository**
    a. This library is designed to work alongside a modified version of Google's Mediapipe repository, configured as a default for MacOS. Specifically, it functions as a server to capture hand landmarks, represented as a list of 21 3D coordinates (x, y, z).
2. **Homebrew**
    a. Homebrew is utilized as a package manager for MacOS. It simplifies the installation and management of various software packages and dependencies.
3. **OpenCV and OpenCV@3**
    a. OpenCV is an essential computer vision library, employed for image processing tasks within the Gesture Recognition Library. Both OpenCV and OpenCV@3 need to be installed separately.
4. **FFmpeg**
    a. FFmpeg is utilized for audio and video processing functionalities. Its inclusion ensures compatibility with a wide range of multimedia formats.
5. **Numpy**
    a. Numpy is a fundamental library for numerical operations in Python. It is employed to enhance array manipulation and mathematical operations in the Gesture Recognition Library.
6. **Jsoncpp**
    a. Jsoncpp is a C++ library used for parsing and manipulating JSON data. Its integration enables effective handling of JSON-formatted data within the library.
7. **Pkg-config**
    a. Pkg-config is utilized to retrieve information about installed libraries, aiding in the compilation and linking process of the Gesture Recognition Library.
8. **XCode**
    a. XCode is an integrated development environment for MacOS and needed by Mediapipe in order to run on MacOS devices.

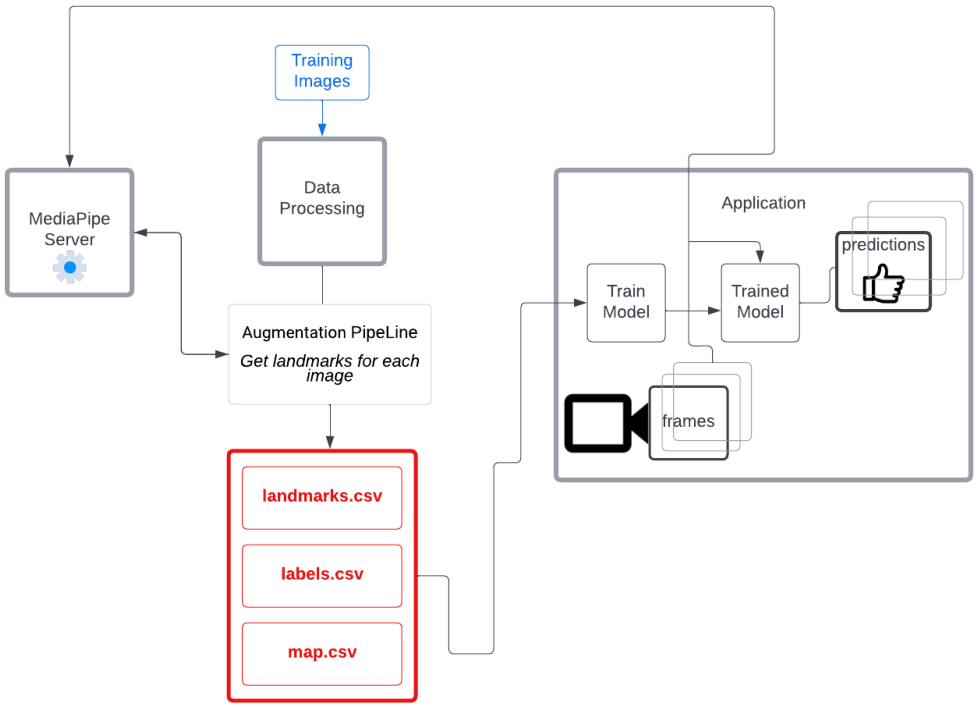
Inspiration and Background

The development of this project draws inspiration from the Python Gesture Recognition Library (GRLib), and its algorithmic approach is documented in this GRLib Github repository.

Use Cases

To showcase the versatility of the Gesture Recognition Library, we have implemented examples such as American Sign Language (ASL) Alphabet Recognition and PowerPoint gesture control. These demonstrations serve as practical applications of the library's functionality.

Data Flow



The workflow begins by configuring and initiating the **MediaPipe server**, a pivotal component responsible for obtaining hand landmarks—(x,y,z) coordinates of the human hand that are detected and tracked in an image or video frame. This step is essential for both model training and subsequent predictions. Moving to the **data processing** phase, the training data undergoes preparation using our specialized application. This involves extracting landmarks from the server and associating them with labels for each training image, resulting in a uniquely structured representation of the data. With the preprocessed data in hand (saved in csv files), **the model is trained** to recognize intricate gesture patterns. Once trained, the model becomes a potent tool for making predictions on various inputs, including images, live videos, or recorded videos. During the **prediction phase**, each image frame is sent to the server to obtain landmarks, and these landmarks are subsequently fed to the model for making predictions.

Concepts

1. Data Augmentation and Pre-processing

1.1. What is Data Augmentation and Why?

Data augmentation is a technique used to enhance dataset diversity, aiding in classification tasks by applying transformations to the original data. In our case, extracting hand landmarks for each image using MediaPipe introduces challenges like odd hand positions or poor lighting. Data augmentation tackles these issues by creating diverse image variations. This helps the MediaPipe server obtain landmarks for more images and improves the model's robustness.

1.2. Augmentation Strategy and Structure

To address the challenges in processing hand landmarks, we adopted an augmentation strategy inspired by the Griblib library. We structured our augmentation process using a pipeline with stages, where each stage represents a specific image augmentation. Each stage can process an image by adjusting its rotation and brightness. The augmented images are sequentially sent to the Mediapipe server until a valid set of landmarks is received for one of the augmentations. The rationale behind this structure is to create a systematic approach to image augmentation, allowing for the exploration of multiple variations of the input until a valid set of landmarks is obtained.

1.3. Loading Images From Folder

The `loadImgsFromFolder` function systematically loads images by efficiently gathering file paths and labels from a specified folder. Utilizing the C++ filesystem library, it navigates through subfolders, assigning unique float labels to classes for compatibility with classifiers like K-Nearest Neighbors (KNN). Instead of returning a vector of images along with their numeric and string labels, the function accumulates file paths and their corresponding numeric and string labels. This strategic choice enhances memory efficiency by postponing the loading and processing of image data until subsequent stages in the program.

1.4. Saving and Loading Processed Data

The reason we save and load processed data is to reduce the time spent preparing it each time we train, test, or run our model. Our model involves several preprocessing steps like image rotation, brightness adjustment, and extracting landmarks from each image. This part of the process can be time-consuming, particularly when dealing with a large volume of data. The part involves saving and loading 3 types of files elaborated in the next section.

1.4.1. Files

`landmarks.csv:` This file is dedicated to storing the landmarks extracted from each image. Our aim is to preserve and retrieve these landmarks in the same format as initially saved. Each

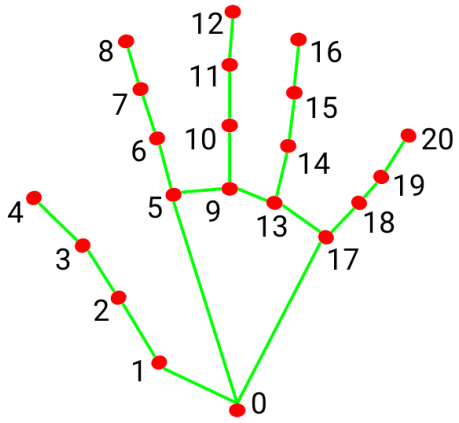
image is associated with 21 landmarks. We've chosen to record these landmarks in a CSV format, with each landmark separated by commas and each image's landmarks separated by a newline character. As a result, each line in the CSV file represents 21 landmarks from one image.

`labels.csv`: It's also crucial to save the labels corresponding to each image. These labels include terms like “call”, “dislike”, “fist”, etc. We convert these labels into numerical values for easier processing and certain classifiers purposes. Each line in the landmarks file correlates with a line in this label file, meaning the first line in the landmarks file corresponds to the first line in the labels file, which contains a simple numerical representation of the label.

`map.csv`: This file serves as a straightforward directory, mapping each image numerical label to its original string label.

2. Hand Landmarks

Hand landmarks play an important role in gesture recognition because they represent specific points on a person’s hand, providing a rich set of information about its pose and configuration. By tracking these landmarks, it becomes possible to interpret gestures and movements, enabling applications ranging from sign language recognition to virtual reality interactions.



0. WRIST
1. THUMB_CMC
2. THUMB_MCP
3. THUMB_IP
4. THUMB_TIP
5. INDEX_FINGER_MCP
6. INDEX_FINGER_PIP
7. INDEX_FINGER_DIP
8. INDEX_FINGER_TIP
9. MIDDLE_FINGER_MCP
10. MIDDLE_FINGER_PIP
11. MIDDLE_FINGER_DIP
12. MIDDLE_FINGER_TIP
13. RING_FINGER_MCP
14. RING_FINGER_PIP
15. RING_FINGER_DIP
16. RING_FINGER_TIP
17. PINKY_MCP
18. PINKY_PIP
19. PINKY_DIP
20. PINKY_TIP

Figure 1: Hand Landmarks definition (Source: Mediapipe)

We leverage Google’s Mediapipe library to obtain these landmarks, because:

1. Rich Information Source
    a. Hand landmarks capture the spatial coordinates (x, y, z) of key points on the hand, such as fingertips, knuckles, and palm center. This information is rich and provides a detailed representation of hand poses and movements.
2. Gesture Variability
    a. Different gestures involve specific arrangements of hand landmarks. By analyzing the relative positions and movements of these points, a system can distinguish between various gestures, making it versatile for recognizing a wide range of interactions.
3. Mediapipe Hand Tracking Capabilities

a. Google's Mediapipe library offers robust hand tracking capabilities, making it a convenient choice for obtaining hand landmarks. The library employs machine learning models trained on large datasets, enabling accurate and real-time hand tracking in diverse conditions.

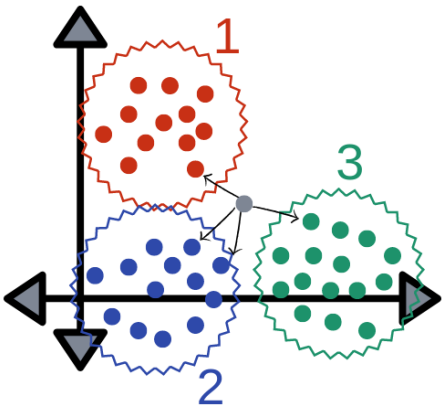
4. Ease of Integration
    a. Integrating with Mediapipe streamlines the process of obtaining hand landmarks. The library abstracts away many of the complexities associated with hand tracking and provides a convenient API, reducing the development effort required to implement this functionality.

In this library, we made Mediapipe as a server and the application as a client. Examples of using Mediapipe as a server could be found in the ASL Gesture Recognition and Powerpoint Gesture Control applications as discussed in the tutorial documentation.

3. Classifiers

Classifiers are essential for any data driven application and our Gesture Recognition is not an exception. Choosing the right classifier for your data and for your application is important and should be done with thought. In our library there are 3 models that are available for use: KNN, SVM, and Decision Trees. We will provide the overview of the following models, what are their strengths and weaknesses, and how to make a choice for your application.

3.1 KNN



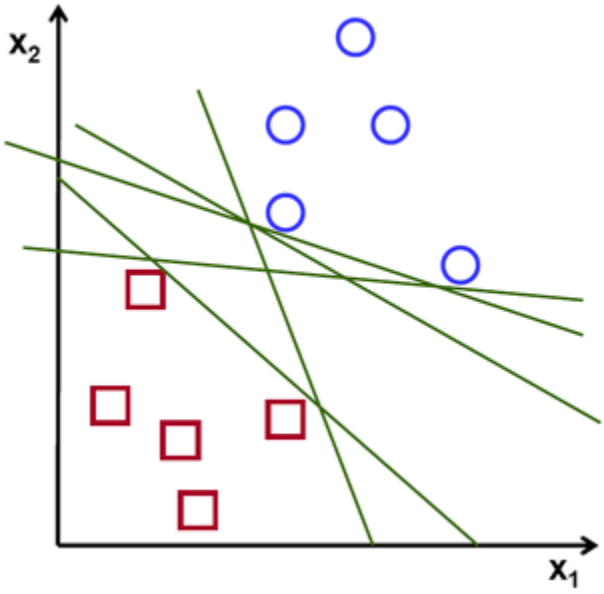
KNN or K-Nearest Neighbour is one of the simplest classification algorithms, its simple design makes it easy to implement and understand. After the data has been separated into train_data and test_data (an essential step for all classification algorithms), KNN model uses the train_data points to make predictions about a new introduced point by computing the distance between the new data point and its closest neighbors. The metrics for finding the closest neighbors can be Euclidean distance, Manhattan distance, Minkowski Distance, or Hamming Distance. For classification, a class label is assigned based on the majority vote of k-neighbors.

KNN is a non-parametric classifier and a lazy algorithm. There is almost zero time for data preparation as the “learning” involves only storing the data and all the computation occurs when prediction is made. That makes the KNN model adaptive to changes in new data as the model is

not fixed during training. However, the computational complexity and memory usage of the model makes it unsuitable for large datasets.

Another important note about the algorithm is that the choice of K-Value can heavily influence the performance of KNN. Lower values of k can have high variance and low bias, while the high values of k may lead to higher bias and lower variance. If your data is noisy, consider choosing higher values for k. Defining K is a balancing act that requires several test runs; however remember to always choose K as an odd number.

3.2 SVM



A Support Vector Machine (SVM) is a discriminative classifier defined by a separating hyperplane. It can solve linear and nonlinear problems and perform well in high dimensional spaces with little data.

The optimal hyperplane separating the classes will have the smallest maximum margin(the distance between the closest examples of trained data to the separating hyperplane). The hyperplane is computed by

$f(x) = \beta 0 + \beta^T x,$

Where β is known as the *weight vector* and B0 as the bias.

While the original SVM was designed to separate two classes, the current openCV implementation allows to classify the data into several classes. There are several important parameters that need to be considered when SVM is called:

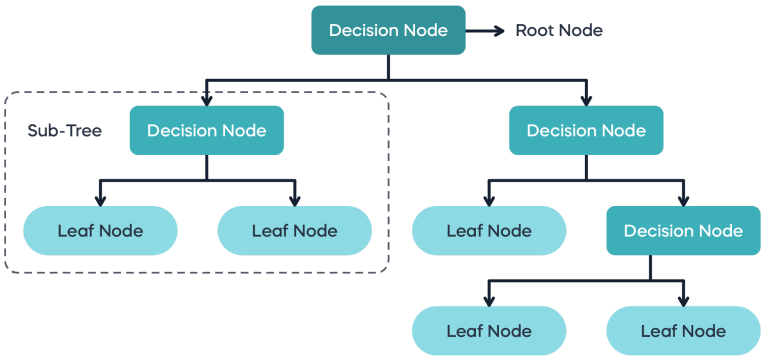
- Type of SVM, defines whether SVM is a vector classification / distribution estimation / vector regression.
- Type of Kernel,defines what kernel will be used in the model. Linear kernel is done in the original feature space, no mapping is done. It is the fastest option, but not always optimal. Consider using Radial basis function/ Sigmoid Kernel/ Exponential Chi kernel/ or Histogram intersection Kernel for better results.
- Gamma parameter defines how much weight will be given to outliers.

To learn more about SVM parameters follow OpenCV documentation. If you do not know which parameters are optional you can call svm.trainAuto() that will run several different parameters

underhood and find the best choice for your application. trainAuto() takes a long time to run so consider doing it once and saving the model, or finding the best parameters yourself.

SVM is memory efficient and robust to overfitting. It is also a convex optimization problem and performs well with limited labeled data. However, it is also computationally intensive and can have limited performance on overlapping classes.

3.3 Decision Trees



Decision Trees is another example of a simple but efficient machine learning algorithm. Many people prefer this classifier due to its clear interpretability and no assumption of linearity.

Moreover, Decision Trees can handle both numerical and categorical data. The root node contains all_data and each Leaf Node is a class, multiple leaves may have the same level. This is a greedy algorithm and the splits happen when the local optimum is found. Hence Decision Tree is not guaranteed to provide the optimal solution. Decision Trees can also struggle to capture relationships that are similar to XOR gate. However, they are fast and can be good for large datasets.

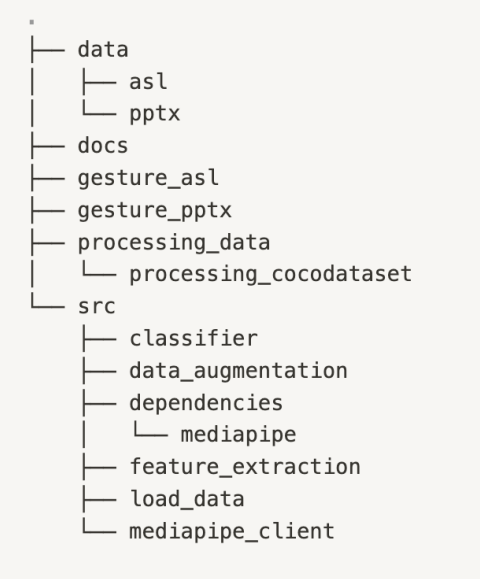
3.4 Choosing your classifier

When choosing between KNN, SVM, and DTree a user might consider the following criterias:

- Is Data Linear?
- Are labels numerical or categorical?
- Is the dataset large and will be memory consuming?
- Are you working in high dimensional space?
- Is there a lot of noise in your dataset?
- Does your application prioritize optimal solutions or speed?

Depending on your answers you might try the different models with different parameters to find the best fit for your Gesture Recognition application.

Architecture Decisions



Our library relies on two separate repositories to operate. You will need both repositories to have the library working:

1. Mediapipe: Accompanying library originally provided by Google that we have specifically reconfigured to operate on MacOS and altered so that the library could function as a server to extract hand landmarks from images. In the directory architecture above, we created a new directory called dependencies under the src directory. We then cloned Mediapipe locally within the Gesture Recognition src/dependencies directory. Cloning it within this directory is not necessary and the library could still function with the Mediapipe repository cloned elsewhere but we recommend cloning Mediapipe in here.
2. Gesture Recognition: Our main library contains functions to preprocess training images, classifiers for prediction of gestures, and client interface to communicate with the Mediapipe server.

Mediapipe

How to work with Mediapipe?

We have simplified the whole repository so you only need to run the required commands to run the relevant parts of the library as a server. See part 2 of the manual for more information.

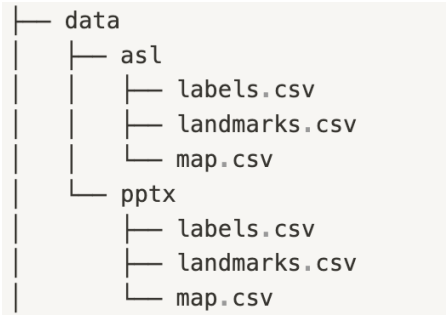
Why as a Server?

A key feature of our library is leveraging Google's existing Mediapipe library to extract hand landmarks from images. Mediapipe is configured around a bazel environment making it challenging to use it as an external library. Mediapipe also has many features that go beyond the scope of our Gesture Recognition library and we are only concerned with one file that extracts hand landmarks from images. Therefore, a workaround that we have where we can simplify Mediapipe to what we need is to set up the specific file of interest as a server that will receive images from the client and return extracted landmarks back. We decided on a server as it allows for live communication with our client applications (e.g., ASL and Powerpoint applications) and allows for live video processing and extraction of hand landmarks.

Gesture Recognition

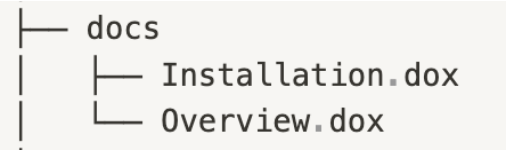
Here, we will discuss each folder that is in the Gesture-Recognition repository and explain their purposes as well as the design choices that we made.

Data



This is a designated directory where users could choose to store the output of the processing script (processing_data/processing.cpp), which are the **labels.csv, landmarks.csv,** and **map.csv** files. These files are then used when running the client applications (e.g., the ASL and the Powerpoint applications) that requires training the desired classifier first. Since the example applications take in a directory to where these files are stored, it's not necessary for the user to use this designated directory and could choose to provide a different path as well.

Docs



This is where all the documentation can be found. In this directory, you will find a tutorial that walks you through the main features of the library and how to use them. You will also find a manual that's written as an easy look up for information regarding installation, usage, and troubleshooting methods associated with the library. Last, you will find this design documentation.

Gesture_asl

```
├── gesture_asl
│   ├── CMakeLists.txt
│   ├── asl_application.cpp
│   └── asl_dataset.txt
```

Gesture ASL is where you can find the code to run our example ASL application. The ASL application allows the user to train a classifier of their choice using their processed training data and launches the camera that will allow the users to sign an ASL alphabet and have the prediction printed out at the bottom right of the camera screen.

This directory is meant to show an example of how you could write your own application using the library. Here, you will find the following files:

1. Asl_dataset.txt: This file contains a link to where you can download the ASL dataset that we will be using to train the classifier in this application
2. Asl_application.cpp: This is the file that will run the ASL application
3. CMakeLists.txt: We have provided this file so that users wanting to try run the application could easily build and run the application with all the dependencies handled

Asl_application.cpp

To build this application, we first designed it so that it takes in a command line argument of where the processed image data is found, and these are the **labels.csv, landmarks.csv,** and **map.csv** files. It's given as a command line argument as the user is free to use any directory that they want instead of the designated data directory. If the user uses the designated data directory, then they could simply provide the relative path *../../data/asl* as a command line argument.

We want the ASL and Powerpoint application to be a guide on how to implement our library, including how to use the different classifiers in your project. We have also included code for how to use the 3 different classifiers provided by our library that the user can just uncomment and try. The application uses SVM as a default as that is the model that yielded the highest accuracy.

Below is the code snippet in the ASL application of how to obtain the different classifiers:

```
// // Train the classifier

/* KNN */
// std::cout << "Training KNN...\n";
// auto [knn,knn_accuracy] = KNN_build(all_images_landmarks_from_csv, all_labels_from_csv, 3);
// std::cout << "KNN Accuracy: " << knn_accuracy << std::endl;

/* SVM */
std::cout << "Training SVM...\n";
auto [svm,svm_accuracy] = SVM_build(all_images_landmarks_from_csv, all_labels_from_csv, stringLabelMap.size());
std::cout << "SVM Accuracy: " << svm_accuracy << std::endl;

/* Decision Tree */
// std::cout << "Training Decision Tree...\n";
// auto [dtree,dtree_accuracy] = DTree_build(all_images_landmarks_from_csv, all_labels_from_csv, stringLabelMap.size());
// std::cout << "DTree Accuracy: " << svm_accuracy << std::endl;
```

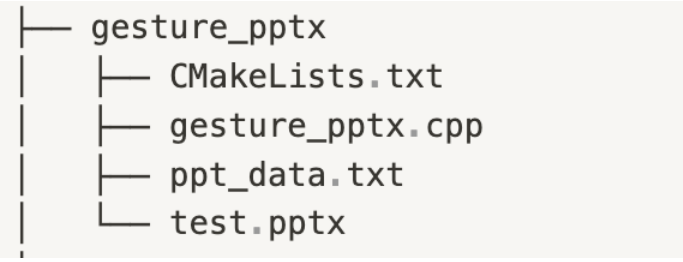
Below is the code snippet from the ASL application that uses the classifiers to make a prediction and saves that prediction in the **result** variable:

```
// knn->findNearest(input_cvMat, 3, result);
svm->predict(input_cvMat,result);
// dtree->predict(input_cvMat,result);
```

The rest of the application are written in a straightforward manner that demonstrates the general algorithm that the user will follow to implement our library which is:
Provide an image → Get landmarks from Mediapipe → Predict using classifier → Do action

You see this implemented by first connecting to the server at the beginning of the application and in a while loop capturing frames from the live camera feed. These frames are then fed into the connected server to extract landmarks. For every landmark identified within an image, we then predict the gesture of that landmark with a classifier. The ASL application is only able to predict on one hand per image frame at a time and will therefore only predict on the last landmark identified by Mediapipe. Once a prediction is made, the predicted alphabet based on the gesture is then displayed on the camera screen for the user to see.

Gesture_pptx



Gesture_pptx is where you can find everything you need to run the Powerpoint Gesture Control application. Similar to the ASL application, the Powerpoint application is also written so it's customizable and the user can choose to train a classifier of their choice. The user will then be able to sign the gesture trained on to control the Powerpoint by entering slideshow mode, going to the next slide, going to the previous slide, and ending slideshow mode.

This directory is meant to show an example of how you could write your own application using the library. Here, you will find the following files:

1. ppt_data.txt: This file contains a link to where you can download the gesture dataset that we will be using to train the classifier in this application
2. test.pptx: This is an example Powerpoint presentation that we have provided for the user to test with
3. gesture_pptx.cpp: This is the file that will run the Powerpoint Gesture Control application
4. CMakeLists.txt: We have provided this file so that users wanting to try run the application could easily build and run the application with all the dependencies handled

gesture_pptx.cpp

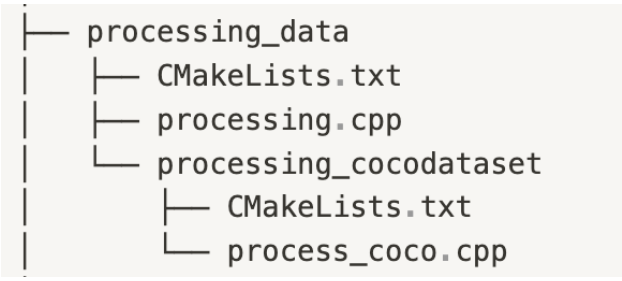
To build this application, we first designed it so that it takes 2 command line arguments:

1. Path to where the processed image data is found, and these are the **labels.csv, landmarks.csv,** and **map.csv** files. It's given as a command line argument as the user is free to use any directory that they want instead of the designated data directory. If the user uses the designated data directory, then they could simply provide the relative path *../../data/pptx* as a command line argument.
2. Path to the Powerpoint presentation that the user wants to run the gesture tool on. This path must be an absolute path as it uses an Applescript to execute the command and it only works on absolute paths.

Similar to the ASL Application, there are classifier codes in the main function of the gesture_pptx.cpp file that can be commented and uncommented so that the user can experiment with the classifier they want.

A design choice that was made when identifying gestures is that often the classifier would make consecutive different predictions on the same gesture as it is unsure what the gesture being signed is. To handle this scenario, we have made it so that the application will only respond with the corresponding action associated with the gesture if it is able to predict the same gesture for 2 consecutive frames of a live video feed. For this reason, the user will need to hold that gesture before the powerpoint presentation will respond. This also prevents the Powerpoint tool from being too reactive to accidental gestures and only responding to gestures that are intentional.

Processing_data



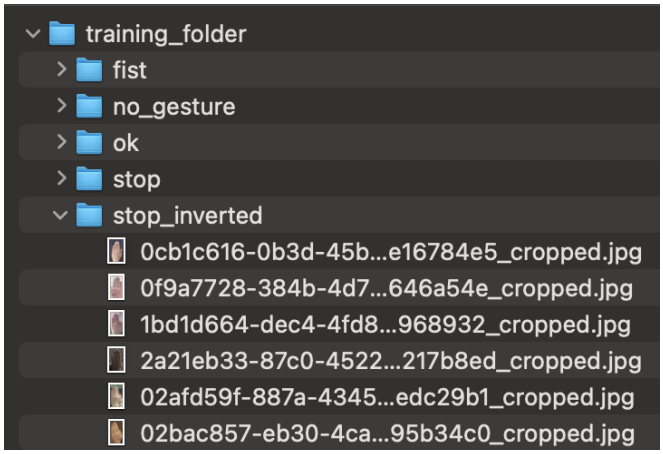
The processing data directory includes the scripts that will allow users to preprocess their training images by extracting landmarks and matching them to the appropriate label through Mediapipe. The reason why preprocessing images have a separate script is because the process of preprocessing an image could take a while especially when training on large datasets. For this reason, having a separate script means that the user could preprocess their images and extract landmarks once and could train on different classifiers using the same preprocessed data.

In this processing_data directory, there are three items:

1. processing.cpp: Processing script that will take in the training images and extract hand landmarks by running the images through Mediapipe
2. CMakeLists.txt: Written to build the processing.cpp file so that users can just build and execute the processing script without worrying about dependencies
3. processing_cocodataset: Directory that contains a specific script that will properly parse coco datasets into the format that our processing.cpp script will take

Processing.cpp

This processing script will need to take in a directory of training images that are organized into the following format:



The reason why we ask the users to follow this format is so that it is clearly labeled which class the training images belong to and provides an easy folder hierarchy that is easy to work with. Moreover, it encourages the users to look at their training images and make sure that they are clean. The processing script will then run the images through the Mediapipe server to extract landmarks and save these landmarks in a csv file along with the corresponding numeric labels as well as a key that converts the numeric labels back to the class labels as strings. The purpose of this is to clean the training images into a format that is ready for training the classifiers. It is also so that the training images can be preprocessed once and could then be used for training in multiple different contexts, which is advantageous given that the preprocessing step is a lengthy process especially for a big training dataset.

Processing COCO dataset

In this directory, contains a script called process_coco.cpp. The point of creating a script to process the COCO dataset, is that we want to have one hand in an image. By having both images and the JSON file corresponding to the image data (label, bounding boxes, and etc.). We are able to crop the image based on the bounding boxes and have one hand in an image, instead of two. Since we are sending these dataset to Mediapipe to get landmarks, having one hand in an image is recommended for Mediapipe to accurately get the landmarks of the correct gesture.

To be able to work with this script it is necessary to have the dataset in COCO format. Once the user has their dataset ready composed of JSON file and images (.jpg/.jpeg), they can run the script to extract one hand from an image. This script is built and run using cmake and takes two arguments when running the script, the path to the dataset and another path for the output folder.

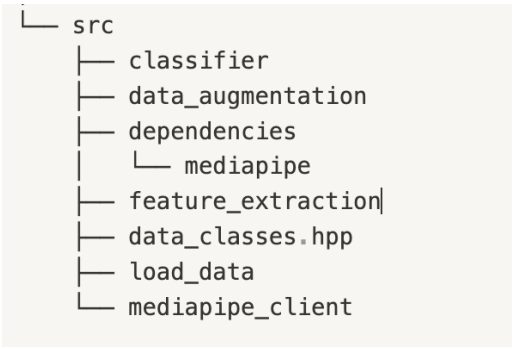
The script will then extract the necessary files from the given dataset path, within the script it first gets the JSON file and gets the necessary information like labels, bounding boxes, image name etc. After obtaining this information, it then gets the image based on the corresponding

image name and crop the image based on the calculations of bounding boxes and save it under the folder based on its respective label.
The image is cropped based on the bounding box with this formulation,

```
cv::Rect roi(x * image.cols, y * image.rows, width * image.cols,
height * image.rows)
```

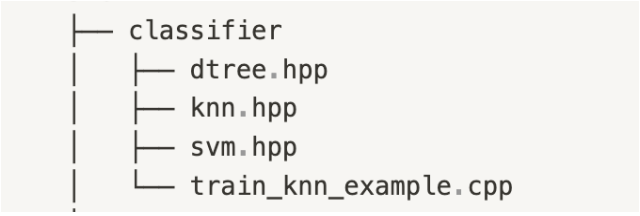
- cv::Rect : is a class in OpenCV that represents the rectangular region
- x, y, width, and height : will be the values of the bounding box
- Image.cols and image.rows : are the width and height of the image, respectively.

Src



The src directory is where all the files and the functions of the library can be found. The files are organized by the different functionalities of the library. Note: the dependencies directory is not included in the Gesture-Recognition repository but is a directory we encourage the user to create and clone the Mediapipe repository in.

Classifiers



The goal of the classifiers functions is to make a model with as little effort from a user as possible. For that reason we create functions KNN_build, SVM_build, and DTree_build, that

take care with different formats and functions in opencv::ml and return the built model and its accuracy.
All three models are following similar algorithm:

- Divide all_data into ***train_data*** and ***test_data***. By taking care of the test validation inside the build model we are ensuring that our users will have a verified model to use and do not have to worry about testing the model themselves.
- Convert Hand_Landmarks vector to **cvMat::CV_32F** type and the labels vector to **cvMat**::**CV_32S** type. Using a different library can be confusing and requires reading documentation and understanding the type parameters openCV is using. Hence, we are taking care of converting the Hand_Landmarks data from application into the cvMat types under the hood of our classifier functions.
- Train the model by calling cv::stat_model::create() and cv::stat_model::train(). Using other verified libraries saves time for error handlings and finding exceptions. Hence when available we will prefer to use machine learning functions instead of implementing SVM training in our code which will be less optimized.
- Test the model with test_data and compute the accuracy. We are computing the accuracy when comparing the percentage of correct predictions of the test_data. Our classifier functions return a type (model,accuracy). We decided that returning accuracy makes it easy for a user to immediately check if the model is good enough for the application or if s/he needs to train a different model.
- The functions also return a smart pointer instead of the model itself as it will be memory inefficient to pass the whole model.

In conclusion, due to the classifier functions the user does not have to know anything about splitting data into train/test or how to find the accuracy of the model. The simplified process will require just passing the Hand_Landmarks together with labels and getting back the model with its accuracy.

Data Augmentation

```
├── data_augmentation
│   ├── image_augmentation.hpp
```

- `increaseBrightness` Function
    - cv::Mat increaseBrightness(const cv::Mat& image, double value = 30)
    - Description: Increases the brightness of the input image.
    - Parameters:
        - `image`: Input image.
        - `value`: Degree of brightness adjustment (default: 30).
    - Return Type: `cv::Mat`

    - Why: Modifies the brightness of the image in HSV color space, ensuring that values do not exceed the valid range.
- `rotate` Function
    - cv::Mat rotate(const cv::Mat& image, double degrees = 0)
    - Description: Rotates the input image by the specified degrees.
    - Parameters:
        - `image`: Input image.
        - `degrees`: Degree of rotation (default: 0).
    - Return Type: `cv::Mat`
    - Why: Applies an affine transformation to rotate the image, preserving lines and parallelism.

Data_classes

```
├── data_classes.hpp
```

In order to help to organize the data structure while working with Landmarks we decided to create helper "abstraction" classes Landmark and Hand_Landmark.

Landmark

Stores {x,y,z} coordinates of one landmark of an image. Can be constructed from passed values of x,y,z separately or from a vector<float> of length 3. When no parameters are passed, creates a default Landmark {0,0,0}.
We decided to create the Landmark class instead of just using vector<float> for the purpose of design abstraction and ease of understanding the code for a user. Besides the constructor the class also has two "conversion" functions and a print function.

- toMatRow()

Converts a landmark to a cv::Mat (CV_32F). The function takes care of preparing the Landmark for the classifier's functions and returns a cv::Mat object.

- toVector()

Converts a Landmark to a vector. We provide this function in case a user would like to have Landmark as a vector class and use std::vector functions on the Landmark object.

- print()

Prints the coordinates of the Landmark.

Hand_Landmark

A class that represents the mediapipe's return for one hand - one Hand_Landmark consists of 21 Landmarks. Again, we are using the abstraction of classes to ease the use of the dataset so a user

wouldn't have to deal with vector<vector<float>> and can instead call Hand_Landmark. The abstraction becomes even more useful when dealing with several Hand_Landmark and creating a vector<Hand_Landmark>.

The constructor takes in a vector<Landmark> and constructs a Hand_Landmark object, otherwise it creates a vector of default Landmark option {0,0,0}. Besides the constructor the class also has a toMatRow() function and print().

- toMatRow()

Converts Hand_Landmark to cv::Mat (1,63,CV_32F) in order to prepare for the classifiers' functions. The function also calls under the hood Landmark's function of toMatRow().

- print()

Print all the coordinates of 21 Landmarks of Hand_Landmark.

Dependencies

```
├── dependencies
│   └── mediapipe
```

As mentioned previously, this dependencies directory is not included in the Gesture Recognition but a directory we recommend the user to create and clone the Mediapipe repository in. The reason why we recommend this is to preserve the logic that Mediapipe is meant to be a dependency of the Gesture Recognition repository.

Feature Extraction

```
├── feature_extraction
│   ├── pipeline.hpp
│   ├── stage.hpp
```

Pipeline Class

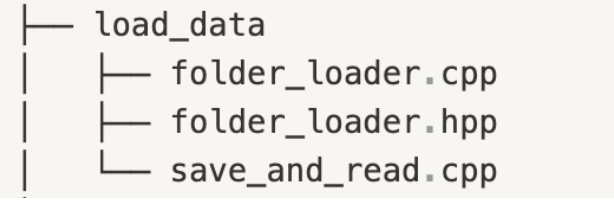
- Type: `class Pipeline`
- Description: Manages a sequence of image augmentation stages.
- Functions:
    - `Pipeline(int num_hands)`: Constructor, initializes the pipeline with the specified number of hands.
        - Type: Constructor
        - Parameters:
            - `num_hands`: Number of hands (1).
        - Why: Ensures the correct number of hands is provided for the pipeline.
    - `void add_stage(double brightness, double rotation)`: Adds a new stage to the pipeline with specified brightness and rotation.
        - Type: Member Function

        - Parameters:
            - `brightness`: Degree of brightness adjustment.
            - `rotation`: Degree of image rotation.
        - Why: Facilitates the addition of customizable augmentation stages to the pipeline.
    - `const std::vector<Stage>& getStages() const`: Retrieves the vector of stages in the pipeline.
        - Type: Member Function
        - Return Type: `const std::vector<Stage>&`
        - Why: Allows external access to the stages for further usage.

Stage Class

- Type: `class Stage`
- Description: Represents an individual image augmentation stage.
- Functions:
    - `Stage(int initial_index, double brightness, double rotation)`: Constructor, initializes the stage with an initial index, brightness, and rotation.
        - Type: Constructor
        - Parameters:
            - `initial_index`: Index of the stage.
            - `brightness`: Degree of brightness adjustment.
            - `rotation`: Degree of image rotation.
        - Why: Sets up the stage with specified parameters.
    - `cv::Mat process(const cv::Mat& image) const`: Processes the input image through brightness adjustment and rotation (used function written in data_augmention/image_augmentation.cpp)
        - Type: Member Function
        - Parameters:
            - `image`: Input image to be processed.
        - Return Type: `cv::Mat`
        - Why: Applies the defined augmentation operations to the input image and returns the processed image.

Load Data



folder_loader.hpp & folder_loader.cpp

- `loadImgsFromFolder` function
    - Description: Facilitates the systematic loading of image data for machine learning tasks by collecting file paths and associated labels from a specified folder.
    - Parameters:
        - `folderPath`: Path to the folder containing class-subfolder structures.
    - Return Type: `std::vector<ImageData>`
    - Why: Enables efficient loading of image data information, optimizing memory usage. Assigns numeric labels to classes for compatibility with machine learning classifiers. Incorporates error handling to ensure the validity of the folder path.

    - `ImageData` Struct
    - The function `loadImgsFromFolder` populates a vector of `ImageData` structures for each image. Each structure contains:
        - `filePath`: The path to the image file.
        - `classStr`: A string representation of the class, extracted from the subfolder name.
        - `label`: The float numeric label associated with the class.

`save_and_read.cpp`

We handle three types of files for loading and saving: `landmarks.csv, labels.csv, and map.csv.` Each file has two associated functions: one for saving processed data and another for reading it back before training or using the model. This results in a total of six functions, all located in this file.

- Landmarks:
    - `void saveToCSV(const std::vector<Hand_Landmarks>& landmarksVector, const std::string& filename)`
      This function uses a nested loop to iterate through the `landmarksVector`, saving each of the 21 landmarks of an object, separated by commas. A semicolon (;) is used between the x, y, z vector points of an object, and a newline character (\n) between different objects in the vector.
    - `std::vector<Hand_Landmarks> readFromCSV(const std::string& filename)`
      This function initializes a `Hand_Landmarks` vector to hold all landmarks from

the file. It uses three nested while loops to read an entire line, then each part up to the semicolon, and each point up to the comma. After reading 3 points, they are saved in a landmark vector, and every 21 landmarks are added to the `Hand_Landmarks` vector.

- Labels:
  - `void saveLabelsToCSV(const std::vector<float>& labels, const std::string& filename)`
    This function iterates through the labels vector, saving each element (number) in a new line in the file.
  - `std::vector<float> readLabelsFromCSV(const std::string& filename)`
    This function initializes a float vector to store all label representations from the file. It reads one line at a time in a while loop, converts the string to a float, and stores it in the float vector, returning this vector.
- Map:
  - `void saveMapToCSV(const std::unordered_map<float, std::string>& stringLabelMap, const std::string& filename)`
    This function iterates through the `stringLabelMap`, saving both elements of each map entry, separated by a comma, and using a newline character between entries.
  - `std::unordered_map<float,std::string> readMapFromCSV(const std::string& filename)`
    This function creates an `unordered_map`, filling it based on the contents of the file. It reads the file line by line, with each line containing two elements separated by a comma - the first being the key and the second the value.

Mediapipe Client

```
└── mediapipe_client
    ├── mediapipe_client.cpp
    └── mediapipe_client.hpp
```

The Mediapipe Client repository is meant to be the interface between the client application that will implement our Gesture Recognition library and the Mediapipe server. The purpose of this aspect of the library is to simplify all server communications so that the client application could just call the necessary functions that will then be handled here. The header file contains all of the function declarations and dependencies that will then be implemented in the .cpp file.

There are 3 functions that can be found in mediapipe_client:

1. connectToServer(): Takes in the port number and ip address of the server, returns a clientSocket. The purpose of this function is to simplify the process of establishing a

connection with the server and the user can simply just call the function and obtain the clientSocket.

2. parseLandmarks(): A helper function for getLandmarksFromServer() function that will take in the raw landmarks from the server that is received as a long string and parse them into Hand_Landmarks structs. This makes working with landmarks easily subscriptable and simpler to work with.
3. getLandmarksFromServer(): Takes in the clientSocket returned by connectToServer(), an input image, and a vector of landmarks that will then be filled with all the landmarks found in the input image. This simplifies the landmark extraction process for the user as they can simply call this function to obtain landmarks from an input image.

Measurables

ASL Alphabet Recognition

- Accuracy of testing images
    - KNN: ~92%
    - SVM: ~98%
    - Decision Tree: ~86%
- Training time
    - KNN: 2.1 seconds
    - SVM: 2.45 minutes
    - Decision Tree: 2.8 seconds
- Responsiveness
    - ~100ms to process each image frame

Powerpoint Gesture Control

- Accuracy of testing images
    - KNN: ~88%
    - SVM: ~97%
    - Decision Tree: ~86%
- Training time
    - KNN: ~7.29 ms
    - SVM: ~67.61ms
    - Decision Tree: ~2.67ms
- Responsiveness
    - ~308ms to process each image frame
    - ~621ms for action to be executed after gesture is signed on camera

Future Work

The current version of the project lays the foundation for a robust and functional system. However, there are several areas where additional work and improvements can be considered in future iterations. The following are potential directions for future development:

1. **Gesture Recognition Expansion** : Enhance the gesture recognition library to support a broader range of gestures beyond the ASL alphabet. Explore additional hand movements (dynamic movements) and poses to increase the versatility of the system.
2. **Cross-Platform Compatibility** : Investigate and implement compatibility with operating systems other than MacOS. Ensure the gesture recognition library and dependencies work seamlessly on Windows and Linux environments.
3. **Machine Learning Integration** : Explore the integration of machine learning techniques to enhance gesture recognition accuracy. Train the system on a diverse dataset to improve recognition performance, especially in challenging scenarios.
4. **Two Hands Extension** : Enhance the gesture recognition library to properly handle multiple hands shown in the screen, and identify which hand it should detect.

Acknowledgements

We would like to express our gratitude to the following individuals and organizations for their contributions to this project:

- The creators and maintainers of Mediapipe for providing a critical component in our system.
- The creators of GRLib for the inspiration of this project.
- Kaggle community for the HaGRID and ASL Alphabet dataset used in our project.
- Professor Bjarne Stroustrup from Columbia University for his valuable insights on the project design using C++.

We also acknowledge the support of the COMS 4995: Design Using C++ teaching assistants, Ulas Alakent and Srishti Srivastava, for their feedback and engagement.

Team Members

The design and development of Gesture Recognition Library in C++ were made possible through the collaborative efforts of the following team members:

- Elifia Muthia
  - Barnard College
  - em3308@barnard.edu
- Elvina Wibisono
  - Barnard College
  - ew2709@barnard.edu
- Yanna Botvinnik
  - Barnard College
  - yb2462@barnard.edu
- Maitar Asher
  - Columbia University
  - ma4265@columbia.edu
- Noam Zaid
  - Columbia University
  - nmz2117@columbia.edu