Creation and Analysis of a Multimodal Dataset

Bachelor Thesis
Faculty of Science, University of Bern

submitted by

Vithusan Ramalingam

from Olten, Switzerland

Supervision:

PD Dr. Kaspar Riesen
Institute of Computer Science (INF)
University of Bern, Switzerland

Abstract

HCI has evolved to include more ways of interacting, such as hand gestures, facial expressions and speech. This thesis contributes to the field by creating and analyzing a multimodal dataset that captures these various forms of input. The dataset includes hand gesture data, facial emotions and spoken language from 30 participants, spanning over 310 minutes of recordings. The primary goal of this work is to develop a publicly available, open-source dataset for future research. Using machine learning techniques, including Random forest, Support Vector Machines, Gaussian naive Bayes, K-Nearest Neighbors, Multi-Layer Perceptron and Convolutional Neural Networks, the data is analyzed to classify gestures and emotions. Additionally, advanced speech recognition models, such as Google Speech API, Whisper AI, Citrinet and Conformer CTC, are evaluated. The secondary goal is to identify the models, which perform the best and their optimal hyperparameters. The results demonstrate the effectiveness of Convolutional Neural Networks and the practical usefulness of K-Nearest Neighbors in hand gesture classification and emotion recognition, while Citrinet and Conformer CTC emerge as the most efficient models for speech transcription.

Acknowledgements

I would like to thank my supervisor, PD Dr. Kaspar Riesen, for his guidance and support throughout my thesis, as well as for his assistance with providing the recording facilities for creating the dataset. Secondly I would also like to express my gratitude to the participants who contributed to the dataset collection, as their involvement made this work possible.

Contents

Chapter 1

Introduction

1.1 Context

Artificial Intelligence (AI) has become a key factor in advancing Human-Machine Interaction (HMI), a field that includes human-robot interaction, human-computer interaction (HCI) and robotics. HMI is used in various domains like medical research, transportation and entertainment [1].

HCI studies how users interact with computers through interfaces. The interactions evolved from traditional physical controls to graphical and natural user interfaces (NUIs). NUIs make interactions more intuitive by using natural inputs like hand movements and speech. Natural user interfaces include gesture-based controls like Microsoft Kinect and voice assistants like Amazon's Alexa and Apple's Siri [2].

This thesis aims to contribute to HCI by developing a multimodal dataset and applying machine learning to better understand human gestures, facial expressions and speech.

1.2 Positioning of the Thesis within AI and HCI

This thesis is situated within the field of HCI, with a focus on Multimodal Interaction, which integrates various forms of input, such as audio and visual channels, to create more intuitive user interfaces. Multimodal Interaction includes Audio-Based and Visual-Based inputs [3], where Speech Recognition processes spoken language, and Hand Gesture and Facial Emotion Recognition analyze visual cues like hand movements and facial expressions. The first part of this thesis is dedicated to the creation of a multimodal dataset, encompassing speech, hand gestures and facial

emotions(see Figure 1.1).

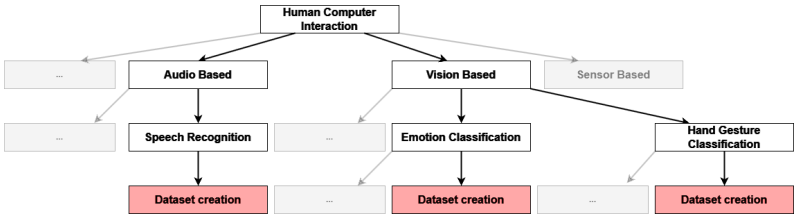


Figure 1.1: Context of the dataset creation in a top down manner.

The second part of the thesis focuses on the analysis of this dataset using machine learning and deep learning techniques (see Figure 1.2). The analysis targets Speech Recognition and Computer Vision tasks. Within Computer Vision, special attention is given to Emotion and Gesture Classification, applying different algorithms to accurately classify emotions and gestures.

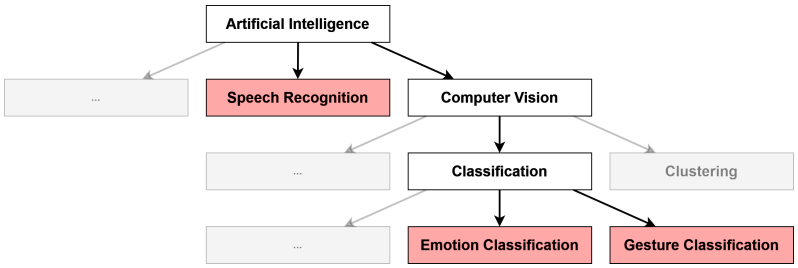


Figure 1.2: Context of the dataset analysis in top down manner.

This work involves both HCI and AI and contributes to the development of more natural and responsive human-computer interfaces by creating an open-source dataset and applying sophisticated analytical methods.

1.3 Related Datasets

Several existing datasets have significantly contributed to AI and HCI research, providing valuable resources for training and evaluating machine learning models. A selection of noteworthy datasets is provided below, along with the dataset from this thesis for comparison (a more detailed description of the dataset will be given in Section 2.1).

Hand Gesture Datasets

Table 1.1 gives an overview of several hand gesture datasets utilized in research. The American Sign Language (ASL) Finger Spelling Dataset contains images of

24 distinct static hand postures representing the English alphabet, excluding the letters J and Z, which are dynamic movements. This dataset is commonly used to train models in hand gesture recognition [4]. The dataset by Kawulok et al. includes up to 32 gesture classes from Polish Sign Language and ASL, recorded from 18 subjects under three different conditions [5]. Sébastien Marcel's dataset is used for basic hand gesture recognition tasks, featuring images of various hand gestures from multiple subjects [6].

Dataset	No. of Gestures	Subjects	No. of Images	Resolution
ASL, 2011	24	9	65'000	240x320
Kawulok et al.	32	18	574	640x490
Sébastien Marcel	6	10	382	128x128
Ramalingam	**10**	**30**	**320'000 (image frames)**	**720x1280**

Table 1.1: Summary of hand gesture datasets.

Facial Emotion Datasets

Table 1.2 lists datasets that offer a variety of facial images used for various recognition tasks. Many experiments use the Japanese Female Facial Expression (JAFFE) database, which contains images of Japanese female models performing seven distinct facial expressions [7]. Another widely used dataset is Extended Cohn-Kanade (CK+), which features a diverse set of facial expressions from 123 subjects and includes 596 sequences covering a wide range of emotions [8]. Compound Emotion (CE) includes high resolution images representing 22 categories of basic and compound emotions from a diverse group of subjects. Facial occlusions are minimized, excluding features like glasses and facial hair [7]. The Yale B Extended (Yale B+) dataset includes facial images with varying lighting and angles, making it useful for facial recognition and emotion analysis [9].

Dataset	No. of Emotions	Subjects	No. of images	Resolution
JAFFE	6	10	213	256x256
CK+	7	123	596	640x490
CE	22	230	5'060	3000x4000
Yale B+	6	28	16'128	320x248
Ramalingam	**6**	**30**	**87'000 (image frames)**	**720x1280**

Table 1.2: Summary of facial emotion datasets.

Speech Datasets

Table 1.3 provides a summary of the datasets used for speech recognition. Mozilla

Common Voice 7.0 dataset is one of the largest German speech corpora and a crowd-sourced dataset and regularly updated, with an increasing amount of data. The Hof University iisys speech corpus, initially designed for text-to-speech (TTS) systems, can also be utilized for speech transcription. Similarly, Thorsten Voice is created for TTS with 23 hours of German speech recorded by a single speaker. It features short sentences related to voice assistant prompts and is pronounced clearly [10].

Dataset	**Length (hours)**	**No. of Speakers**
Mozilla Open Voice 7.0	965	15620
Hof University iisys	326	122
Thorsten Voice	23	1
Ramalingam	**1.5**	**30**

Table 1.3: Summary of spoken language datasets in German.

1.4 Research Questions and Goals

This thesis addresses the following questions:

- *Which algorithms perform best in hand gesture classification?*
- *Which algorithms perform best in facial emotion classification?*
- *Which pretrained speech recognition models perform the best for spoken language transcription?*

Specifically, the goal is to determine the best algorithms and hyperparameters for achieving the highest classification accuracy in hand gesture and facial emotion recognition. For speech recognition, the focus is on finding models that balance accuracy with computational efficiency.

In addition, this thesis aims to develop a multimodal dataset that captures hand gestures, facial expressions and spoken language. This dataset will be made available as an open-source resource to support and advance future research in these fields.

This thesis hypothesizes that neural network architectures, such as Convolutional Neural Networks and Multi-Layer Perceptrons, will generally outperform simpler machine learning models like Gaussian Naive Bayes or Support Vector Machines in terms of classification accuracy and overall performance for hand gesture and facial emotion classification, as well as spoken language transcription. Additionally, for

speech recognition, it is anticipated that the Citrinet and Conformer CTC Large models will perform better. This is due to their advanced architectures, which are specifically designed to handle complex speech patterns and large-scale datasets effectively.

1.5 Outline

The remainder of this thesis is organized as follows. Chapter 2 provides an overview of the dataset, detailing how the various elements of the dataset, such as different facial emotions, hand gestures and speech, are selected. Chapter 3 outlines the methodology, detailing the preprocessing steps for keypoint and audio data, and describes the algorithms used for classification and transcription. Chapter 4 presents the evaluation of the algorithms and the results of their application. It includes an analysis of the models' performance in recognizing hand gestures, facial emotions and speech, as well as a discussion of their perfomance. Chapter 5 concludes the thesis by summarizing the findings and their implications. Finally, Chapter 6 explores possible avenues for future research that could benefit from further exploration and advancement.

Chapter 2

Dataset Description

The dataset for this thesis is designed to capture a diverse range of inputs, including hand gestures, facial emotions and spoken language. The sections below provide an overview of the gestures, emotions and speech samples we asked participants to perform. The detailed manual of the exact dataset can be found in Appendix A.

2.1 Dataset Overview

The dataset for this study comprises contributions from 30 individuals, predominantly students, with roughly a quarter being female. The collected video data spans a total duration of 310 minutes and 28 seconds. This dataset is segmented into three primary types of data: 182 minutes and 1 second of hand gesture data, 51 minutes and 21 seconds of facial emotions data and 77 minutes and 6 seconds of spoken language data (see Figure 2.1). These segments are essential for analyzing and improving various aspects of multimodal Human-Computer Interaction (HCI) systems.

The dataset was recorded using an Apple iMac (24", M1, 2021), equipped with an Elgato Green Screen for a consistent background and a Rode NT-USB microphone for high-quality audio capture.

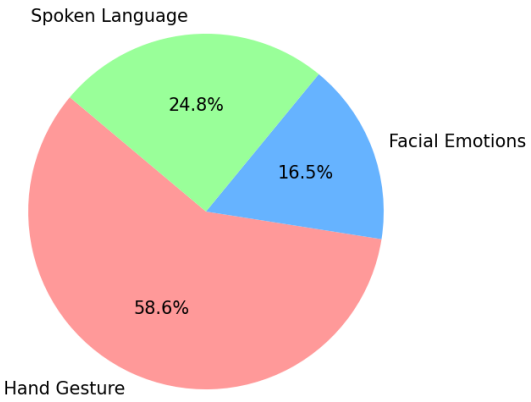


Figure 2.1: Summary of the data distribution.

2.2 Hand Gesture Data

Hand gestures are a form of body language, where the meaning is understood through the position and shape of the palm and fingers. Generally, gestures are divided into two types, static and dynamic. Static hand gestures refer to specific shapes made by the hand, while dynamic hand gestures involve a series of hand movements.

Hand gestures are a powerful tool in non-verbal communication, serving as an intuitive interface in HCI. According to recent research, hand gestures offer a natural and intuitive communication mode, eliminating the need for intermediate media in HCI. Users can interact with computers without relying on traditional input methods, using hands directly as input devices. This is particularly advantageous as it minimizes limitations for users and improves the efficiency of interactions [11].

Participants are tasked to perform ten distinct static hand gestures, including five letters from ASL and five commonly used gestures, from various angles to capture a diverse range of possible variations. By recording gestures from different perspectives and using both hands, we aim to have a more varied dataset to help train more robust models.

These hand signs are selected due to their relevance in HCI applications, aiming to explore alternatives to traditional input methods and enhance interaction possibilities. Recognition of ASL, for instance, could serve as a potential alternative to keyboard input, allowing users to interact with systems through hand gestures rather than typing.

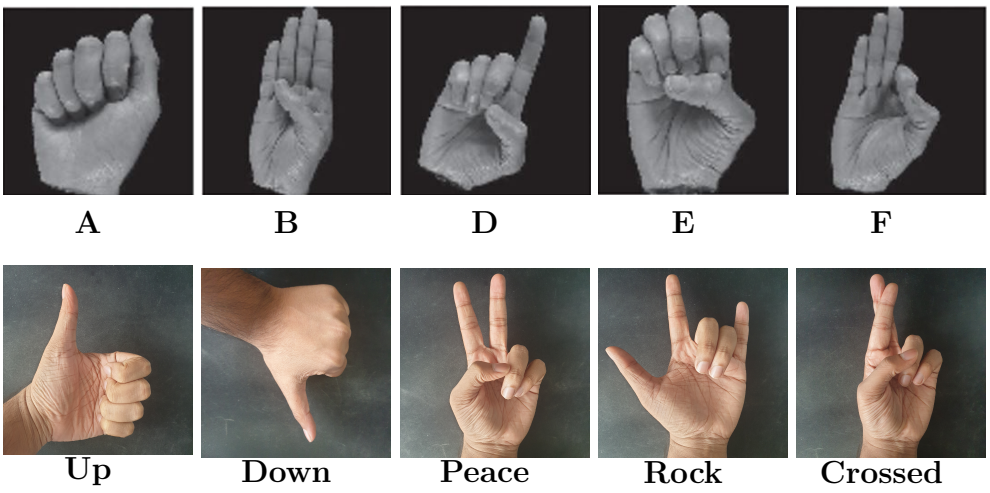


Figure 2.2: ASL [12] and more gestures used in the dataset.

2.3 Facial Emotion Data

Facial expression refers to combinations of visible movements or positions of facial muscles that convey emotions or reactions [12]. Facial expressions are important for recognizing emotions and play a major role in non-verbal communication. They are essential for daily emotional interactions and are often used automatic emotion recognition systems due to their importance [13].

Paul Ekman identifies six basic emotions in his work , which are happiness, surprise, anger, fear, disgust and sadness and they are universally recognized across different cultures. These emotions, each associated with distinct facial expressions, are fundamental to understanding human communication and play an important role in social interactions [14].

Happiness, surprise, anger and disgust, the four emotions selected for this dataset, are among Ekman's basic emotions. The Participants are requested to express these four emotions to the best of their capabilities. Recognizing these emotions allows HCI systems to adapt their responses based on users' emotional expressions, thereby enhancing interaction.

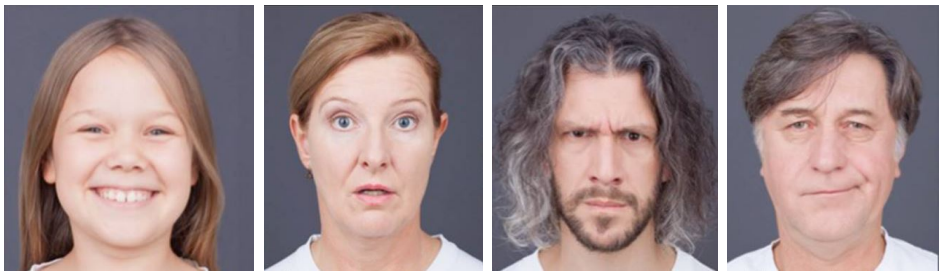


Figure 2.3: Facial emotions in dataset: Happy, Surprise, Anger and Disgusted [15].

2.4 Spoken Language Data

Speech is the main way humans communicate with each another and given its central role in human interaction, it has become a focus in HCI. Consequently Speech Recognition systems are researched and developed to allow interfaces to process spoken language and convert it into written text [16].

Participants are asked to read aloud a series of letters, words, sentences and a prosa text in German. The spoken language data includes 8 letters, consisting of a mix of vowels and consonants, along with 10 words, all chosen to capture a wide range of phonetic diversity and complexity, providing a robust dataset for testing speech recognition models.

The 11 sentences in the dataset are a mix, including sentences from Niels Schiller's work , where he outlines the phonetic variation of the German /r/ [17]. Some of the test utterances from his research are also included. Alongside sentences from the Oldenburger Satztest, a widely recognized assessment for speech intelligibility in noisy environments and additionally we also use sentences containing homophones [18].

Finally, a passage from Franz Kafka's Metamorphosis is selected for the prose text due to its complex structure and range of linguistic features, offering a rich dataset for testing and refining speech recognition algorithms.

This multimodal dataset, encompassing hand gestures, facial emotions and spoken language, is designed to provide a comprehensive resource for advancing the development of HCI technologies.

Chapter 3

Methodology

This chapter explains the methodology for analyzing the dataset. Section 3.1 details the keypoint classification pipeline, which is essential for processing and classifying gestures and facial expressions from video data (see Figure 3.11). It also covers the metrics to evaluate the classifiers . Section 3.2 introduces the speech recognition pipeline, which focuses on processing and analyzing spoken language data.

3.1 Keypoint Classification Pipeline

The keypoint classification pipeline is the most important component of our methodology, involving multiple stages from raw data processing to the final classification. These stages include frame extraction, keypoint extraction, keypoint preprocessing, dataset annotation, classification using machine learning algorithms and model evaluation.

3.1.1 Frame Extraction with OpenCV

The first step in the pipeline involves manually editing the video to remove mistakes, followed by frame extraction using OpenCV. OpenCV is an open-source computer vision library used for image and video processing. OpenCV provides the tools necessary for reading video streams, processing frames and converting the video into a sequence of images. These frames are then analyzed to extract relevant keypoints for gesture and emotion recognition tasks.

3.1.2 Mediapipe Framework

Mediapipe is an open-source framework developed by Google and is recognized for its robust capabilities in computer vision, particularly in extracting keypoints for facial and hand landmarks. This thesis utilizes Mediapipe version 0.10.9, specifically its Face Mesh and Hand Landmarker functionalities, which provide 468 and 21 landmarks, respectively, each with x, y and z coordinates.

The Hand Landmarker module uses a combination of the BlazePalm and Hand Landmark models to detect and track 21 key points on each hand, including fingertips, joints and the base of the hand. Using Convolutional Neural Networks, the module predicts 3D coordinates from 2D input images. The x and y coordinates are normalized to the range [0.0, 1.0], corresponding to the image's horizontal and vertical edges. The z-coordinate represents depth relative to the wrist, estimated based on the hand's position in the image. This normalization ensures consistency across varying hand sizes and distances. Optimized for real-time processing, the module is well-suited for applications like gesture recognition [19].

The Face Mesh module works by using the BlazeFace and Face Landmark models to detect facial landmarks in real time. It captures 468 specific points on the face, which help in identifying key features like the eyes, mouth, and nose. The modules apply Convolutional Neural Networks to extract information from 2D images and predicts 3D landmarks. The system analyzes the spatial relationships between these points to estimate the z-coordinate. This approach allows for the creation of a 3D face model using video from a single camera, even on mobile devices [20].

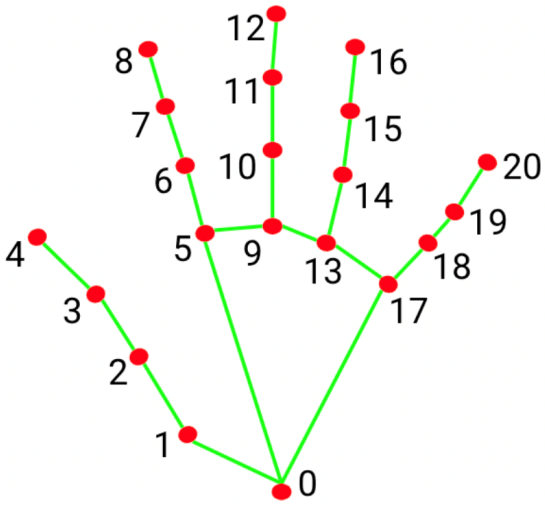


Figure 3.1: Hand Landmarker [21].

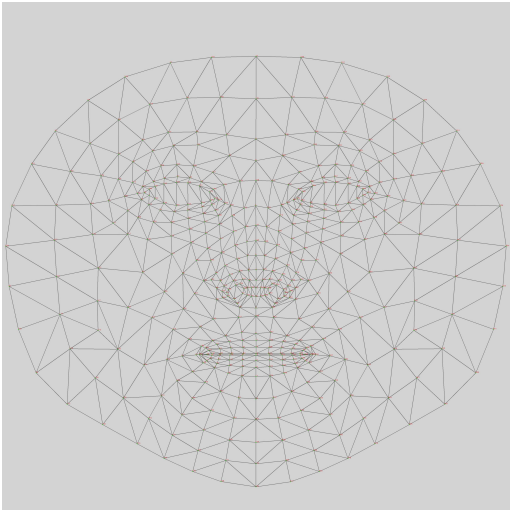


Figure 3.2: Face Mesh [22].

3.1.3 Preprocessing of Keypoints

The preprocessing, done prior to training the models, consists of a series of steps: omission of the z-coordinate, rescaling, conversion to relative coordinates, flattening and normalization [23]. Details on the implementation of the preprocessing can be found in Appendix C. The process begins by omitting the z-coordinate from the keypoints, as it could become unstable and jittery. The remaining coordinates are represented as:

$$\mathbf{X} = [[x_1, y_1], [x_2, y_2], \ldots, [x_n, y_n]] = [\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_n],$$

with $\mathbf{X}_i = [x_i, y_i]$ and n denotes the number of keypoints.

First, the coordinates are rescaled based on the video resolution $\mathbf{R} = [1280, 720]$ using element-wise multiplication:

$$\mathbf{X}'_i = \mathbf{X}_i \odot \mathbf{R} = [x_i \times 1280, y_i \times 720].$$

Next, relative coordinates are computed by subtracting the reference keypoint coordinates, $\mathbf{X}_{\text{ref}}$, from each keypoint:

$$\mathbf{X}''_i = \mathbf{X}'_i - \mathbf{X}_{\text{ref}} = [x''_i, y''_i].$$

These relative coordinates are then flattened into a single vector:

$$\mathbf{X}_{\text{flat}} = [x''_1, y''_1, x''_2, y''_2, \ldots, x''_n, y''_n].$$

Finally, max normalization is applied to scale each element relative to the maximum absolute value in the vector:

$$\mathbf{X}_{\text{norm}} = \frac{\mathbf{X}_{\text{flat}}}{\max\left(|x''_1|, |y''_1|, \ldots, |x''_n|, |y''_n|\right)}.$$

The normalization process scales each point based on the point with the largest relative coordinate in the list, ensuring that the scaling is proportional to the maximum observed value.

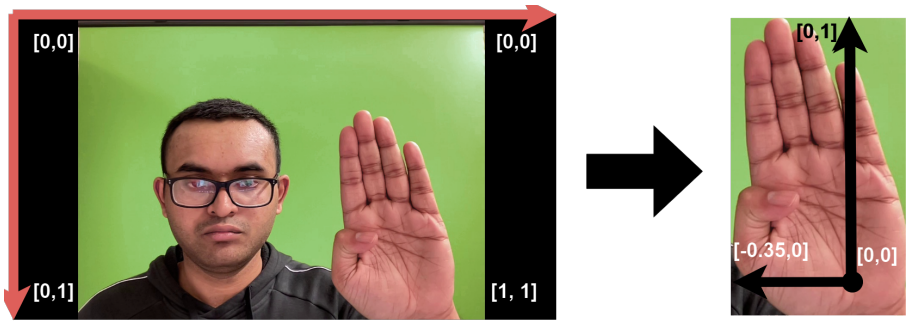


Figure 3.3: Visualization of the preprocessing.

3.1.4 Preprocessing of Face Keypoints

The preprocessing of face keypoints begins by categorizing the landmarks into four groups: mouth, left eye, right eye and the rest of the facial landmarks. Each of these groups has its own reference point (see 3.4). Together with these reference points, the same preprocessing steps explained in Chapter 3.1.3 for hand keypoints are applied to each group individually. This includes rescaling for video resolution, computing relative coordinates, flattening them into a vector and applying max normalization. Finally, the processed lists from all the groups are combined into a single list of normalized coordinates. Specific details of the implementation in Python can be found in Appendix D.

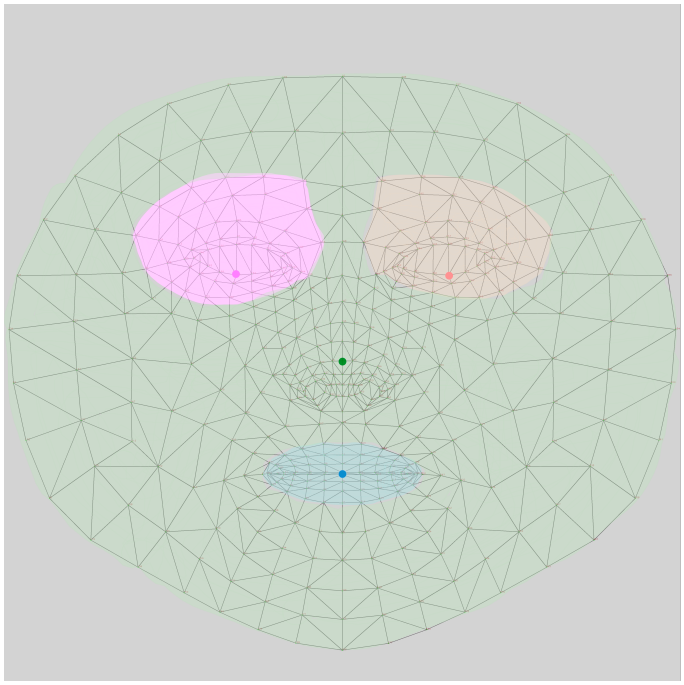


Figure 3.4: The four face regions with their respective reference points [22].

3.1.5 Dataset annotation

In this thesis, we employ supervised learning, which involves training a model on a dataset where each example is paired with an output label. This setup helps the model learn how to map inputs to outputs by recognizing patterns in the data, which it then uses to make classifications on new examples [24].

To annotate our data, we organized the different gestures and emotions into separate folders. These folders act as labels during preprocessing and make it easier to manage and categorize the data.

All keypoints and their corresponding labels are saved in Comma-Separated Values files. The implementation of the data annotation is contained in Appendix C and Appendix D.

3.1.6 Classification Using Machine Learning Algorithms

The preprocessed keypoints for the hand landmarks and face landmarks data are then fed into various machine learning algorithms for training. The algorithms tested in this study include Random Forest, Support Vector Machines, Gaussian Naive Bayes, K-Nearest Neighbors, Multi-Layer Perceptron and Convolutional Neural Networks. A brief description of these algorithms are provided below.

Random Forest

Random Forest (RF) is a supervised learning method that builds multiple decision trees during training. Each tree is created using a process called bootstrap, where a random subset of the data and features is used. The final prediction is determined by combining the outputs from all the individual trees, typically using majority voting for classification or averaging for regression tasks [25]. An exemplary schematic of the RF algorithm can be found in Figure 3.5.

Support Vector Machines

Support Vector Machines (SVMs) are supervised learning models that can be used for both classification and regression. SVMs function by computing the hyperplane that best divides data points of different classes (see example in Figure 3.6). The best hyperplane is the one that maximizes the margin, or the separation between the support vectors (the data points closest to decision boundary) and the hyperplane [26].

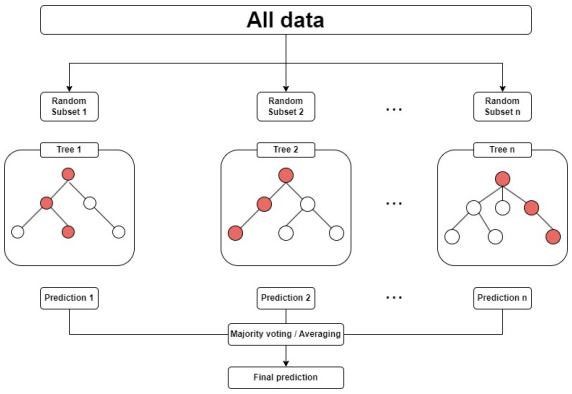


Figure 3.5: Schematic of the RF algorithm.

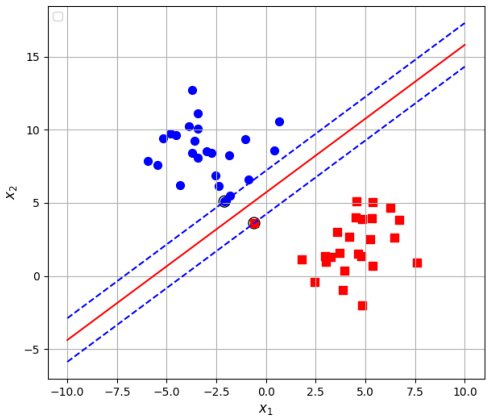


Figure 3.6: Example of SVM classification.

Gaussian Naive Bayes

Gaussian Naive Bayes (GNB) is a variant of the Naive Bayes classifier that assumes the continuous features follow a normal distribution. This is why it's referred to as Gaussian Naive Bayes. The classifier is based on the Bayes Theorem under the "naive" assumption that all features are conditionally independent given the class label [27].

K-Nearest Neighbors

Figure 3.7 depicts an example of the K-Nearest Neighbors (KNN) algorithm. K-Nearest Neighbors is a simple and intuitive algorithm used for classification and regression tasks. It works by finding the k closest data points (neighbors) in the feature space and then making predictions. For classification, it assigns the label that is most common among these nearest neighbors, while for regression tasks, it calculates the average of their values [28].

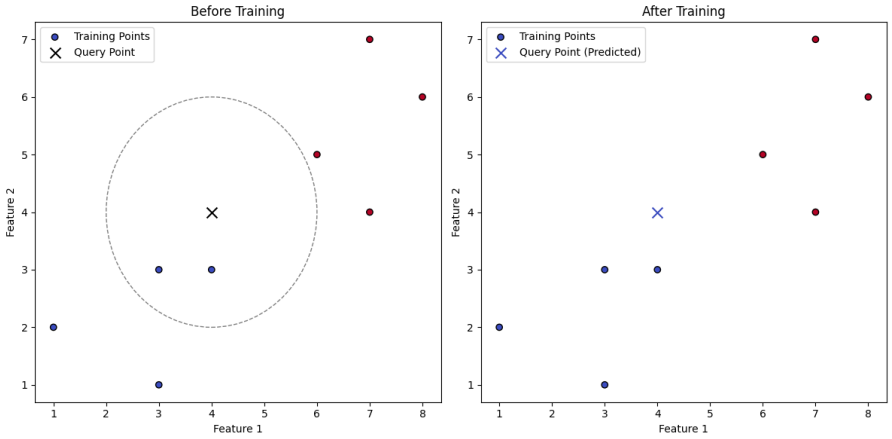


Figure 3.7: Example of KNN classification.

Multi-Layer Perceptron

A Multi-Layer Perceptron (MLP) is a feedforward neural network with an input layer, one or more hidden layers and an output layer, where each layer is fully connected to the next (see example in Figure 3.8). The network learns through an iterative process. In each iteration, it performs a forward pass to compute the output, calculates the loss to measure error and then uses a backward pass to determine gradients for each weight. Optimization algorithms update the weights to the error and this process is continued until the network reaches the desired performance [29].

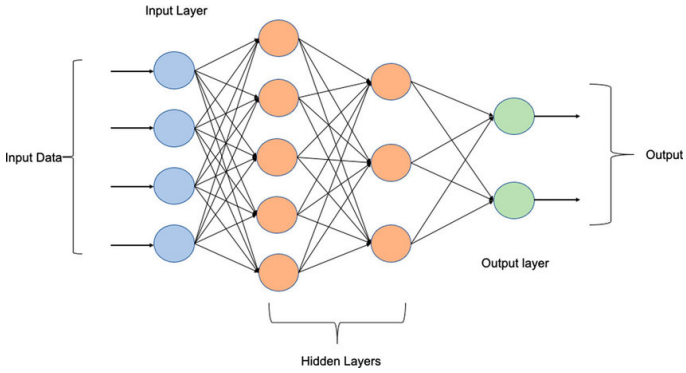


Figure 3.8: Architecture of an MLP [30].

Convolutional Neural Networks

A Convolutional Neural Network (CNN) is a type of neural network designed to identify patterns in data. Figure 3.9 provides an example of a CNN architecture. The algorithm works by applying small filters, also called kernels, that slide over the input to detect important features. By stacking several layers of these filters, the network can recognize increasingly complex patterns, making it effective for various

classification tasks. After applying the filters, the network often uses pooling layers to reduce the data size, which helps simplify the model and prevent overfitting [31].

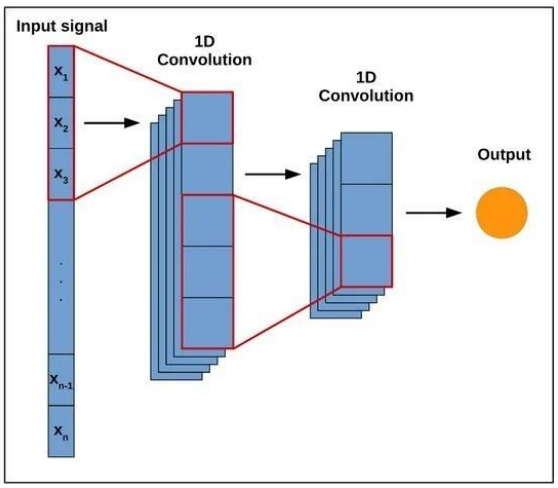


Figure 3.9: CNN with 1d Convolution [32].

3.1.7 Performance Metrics

several key metrics are employed to evaluate the performance of the machine learning models. These metrics include accuracy, precision, recall and F1-score. Before we discuss these metrics, it's essential to understand some fundamental terms.

Figure 3.10 provides an overview over the types of correct and incorrect classifications. **True Positives (TP)** refer to data points where the model correctly predicts a specific class. For a given class i, TP for that class are the data points where the actual class is i and the model also predicts class i. **True Negatives (TN)** are data points where the model correctly predicts a class other than the specific class i. **False Positives (FP)** are cases where the model incorrectly predicts a specific class i when the actual class is not i. **False Negatives (FN)** are data points where the model incorrectly predicts a class other than i class when the actual class is i.

		Actual	
		True	False
Predicted	Positive	True Positive	False Positive
	Negative	True Negative	False Negative

Figure 3.10: Confusion matrix for binary classification [33].

Accuracy (3.1) measures the proportion of correctly predicted instances among the total instances. **Precision** (3.2) is the ratio of true positive results to all

positive results predicted by the model. **Recall**, also referred to as Sensitivity (3.3), describes the fraction of actual positives correctly identified by the model. **The F1-Score** (3.4) combines precision and recall by computing the harmonic mean of precision and recall [34]. The formulas for these metrics are as follows:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{Total Predictions}} \tag{3.1}$$

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{3.2}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{3.3}$$

$$\text{F1-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{3.4}$$

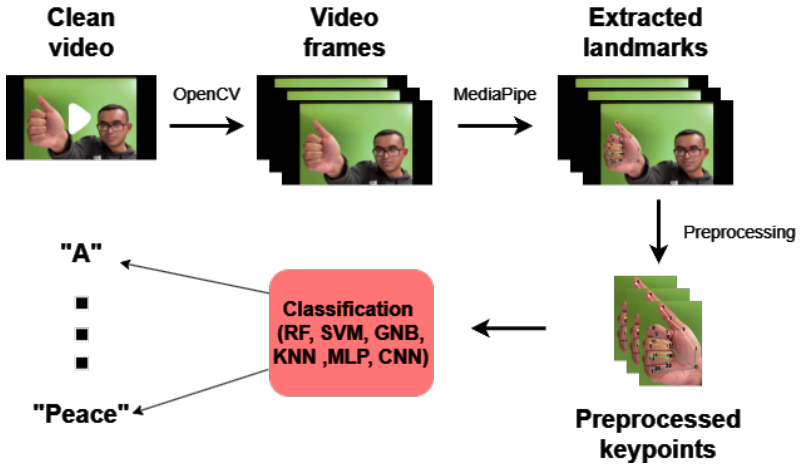


Figure 3.11: Summary of the keypoint classification pipeline.

3.2 Speech Recognition Pipeline

This part describes the methodology and algorithms employed for the speech recognition tasks in this study. The process begins with preprocessing the recorded audio,

followed by applying various speech recognition algorithms to convert the spoken content into text. This section concludes by detailing the performance metrics which measure the perfomance of these algorithms.

3.2.1 Preprocessing of Speech Data

Before applying any speech recognition algorithms, the recorded audio data undergoes a preprocessing stage. Each individual's recordings are edited to remove any bloopers or unwanted segments, ensuring that only the clean and relevant portions of the audio are utilized for analysis. In a next step the audio files are converted to mono and resampled to 16 kilohertz to be compatible with the models.

3.2.2 Speech Recognition Algorithms

In this study we employ five different pretrained speech recognition models to explore their performance and accuracy in transcribing the processed audio data.

Google Speech Recognition API

The Python Speech Recognition library is a popular library for converting speech to text. This library provides a simple interface for working with different speech recognition engines, such as Google Web Speech API, Sphinx and more. In this thesis, the Google Speech Recognition API was utilized to transcribe audio data into text [35].

Whisper AI

Whisper AI is a state of the art ASR system developed by OpenAI. Whisper AI includes multiple multilingual models designed to handle various languages and accents with high accuracy, making it particularly effective in transcribing diverse and complex speech patterns. The system is based on a transformer architecture, which allows it to model long-range dependencies in the audio data, improving transcription accuracy [36]. In this thesis, we utilized the medium and large models of Whisper AI to test the advanced capabilities for our transcription tasks.

Citrinet

Citrinet is a deep learning model from the NVIDIA NeMo toolkit designed for speech recognition. It employs a CNN architecture and subword tokenization to improve the model's ability to handle complex linguistic patterns, including rare words and names [37]. The model has been trained on the German portions of several large datasets, including Mozilla Common Voice (MCV 7.0), the Multilingual LibriSpeech

(MLS) corpus and the VoxPopuli dataset [38].

Conformer Connectionist Temporal Classification

Conformer Connectionist Temporal Classification, also part of the NVIDIA NeMo toolkit, is a powerful model that combines CNNs with Transformer architectures . Using the Connectionist Temporal Classification (CTC) loss function, this model enables flexible transcription without the need for perfectly aligned input-output pairs. By integrating attention mechanisms with convolutional features, Conformer CTC Large delivers high accuracy on a wide range of speech recognition tasks [39]. In this thesis, the Conformer CTC Large DE model is used, which is trained on German datasets such as MCV 7.0, MLS and VoxPopuli, similarly to Citrinet [40].

3.2.3 Performance Metrics for Speech Recognition

Word Error Rate (WER) and Character Error Rate (CER) are metrics for evaluating the performance of speech recognition systems. WER (3.5) measures the percentage of words in the reference transcription that are incorrectly predicted by the model, accounting for the number of substitutions, deletions and insertions required to match the predicted transcription to the reference. On the other hand, CER (3.6), operates at the character level, making it particularly useful for assessing the accuracy of transcriptions with short words or acronyms [41].

The formulas for these metrics are as follows:

$$\text{WER} = \frac{\text{S}_{\text{words}} + \text{D}_{\text{words}} + \text{I}_{\text{words}}}{\text{Number of Words}} \tag{3.5}$$

$$\text{CER} = \frac{\text{S}_{\text{characters}} + \text{D}_{\text{characters}} + \text{I}_{\text{characters}}}{\text{Number of Characters}} \tag{3.6}$$

where S = Substitutions, D = Deletions, I = Insertions.

Chapter 4

Algorithm Evaluation

This chapter provides an overview of the experimentation process, detailing the implementation of various machine learning algorithms, the identification of the best hyperparameters and the evaluation and discussion of their performance on the dataset. All Tables and Metrics can be found in Appendix c.

4.1 Experimentation Setup

The machine learning algorithms are primarily implemented using the Sci-kit learn library (version 1.3.0), with the CNN specifically implemented using PyTorch (version 2.2.0). Performance metrics are computed using Sci-kit learn.

The dataset is divided such that 6 out of the 30 individuals' data are reserved for the testset, representing approximately 20% of the data, ensuring that the testset consists of never-seen data. The remaining 80% of the data is further split into 80% for training and 20% for validation.

To fine-tune the models, we perform a gridsearch to optimize the hyperparameters. A system with an Intel i7 7700k CPU, 16 GB 2400 MHz RAM and an Nvidia GTX 1080 Ti GPU is used for hyperparameter tuning and analysis. The best hyperparameters identified for hand gesture and face emotion recognition are listed in the Table 4.1. For the analysis of the pretrained models in speech recognition, we utilize all 30 speech recordings as the testset to evaluate the performance of the models. Word Error Rate and CER are calculated using the jiwer library.

Algorithm	Hyperparameter	Explanation	Optimal Hyperparameter	
			Hand Gesture	**Facial Emotion**
RF [25]	`n_estimators`	Number of trees in the Random Forest.	65	250
	`criterion`	Splitting criteria for the nodes.	'entropy'	'gini'
	`max_depth`	Maximum depth of each tree.	30	50
	`min_samples_split`	Minimum samples to split a node.	5	2
	`min_samples_leaf`	Minimum samples required at leaf node.	1	1
	`bootstrap`	Whether sampling is with replacement.	False	False
SVM [26]	`C`	Trade-off between margin and error.	5.0	10.0
	`kernel`	Function to transform the data.	'rbf'	'rbf'
	`gamma`	Kernel coefficient for data influence.	0.9	1.5
GNB [42]	`var_smoothing`	Smoothing amount for the variance.	1e-10	1e-10
KNN [43]	`n_neighbors`	Number of neighbors for classification.	15	4
	`weights`	Weight function for predictions.	'distance'	'distance'
	`metric`	Distance metric for neighbors.	'euclidean'	'euclidean'
	`algorithm`	Algorithm to compute nearest neighbors.	'auto'	'auto'
	`leaf_size`	Leaf size for tree-based algorithms.	20	20
MLP [43]	`hidden_layer_sizes`	Sizes of hidden layers in network.	(84, 24, 12)	(512, 256, 128, 64, 32)
	`activation`	Activation function used in network.	'relu'	'relu'
	`solver`	Algorithm for optimizing weights.	'adam'	'adam'
	`max_iter`	Maximum iterations for optimization.	200	250
CNN [44]	`num_epochs`	Number of training iterations.	50	50
	`learning_rate`	Step size for weight updates.	10e-3	10e-4
	`conv1_out_channels`	Filters in first convolution layer.	32	32
	`conv2_out_channels`	Filters in second convolution layer.	64	64
	`fc1_out_features`	Neurons in first fully connected layer.	64	256
	`conv_kernel_size`	Size of convolution filters.	3	3
	`pool_kernel_size`	Size of pooling window.	2	2
	`pool_stride`	Steps taken by pooling window.	2	2
	`dropout`	Probability of neuron omission.	0.0	0.1

Table 4.1: Best Hyperparameters for Keypoint Classification.

4.2 Hand Gesture Recognition Evaluation

We evaluated all classifiers on accuracy, precision, recall, F1-score (see Subsection 3.1.7) and their training time using the testset. The results are summarized in Table 4.2.

The CNN achieves the highest accuracy of 94.6% with a longer training time of 874.12 seconds, while the MLP performs almost as well with an accuracy of 94.0% and requires a much shorter training time of 83.768 seconds.

Random Forest is slower than MLP with a training time of 97.904 seconds achieves an accuracy of 93.4%. This increase in training time could result from the complexity of the data, as more decision trees are constructed and aggregated, requiring more computing time.

Similarly, the SVM achieves an accuracy of 92.3% but is even slower than the MLP with a training time of 147.746 seconds. This longer training time reflects

the non-linear nature of the data, as the complexity of the SVM in managing non-linear decision boundaries can lead to longer training times compared to the more straightforward architecture of the MLP.

The GNB classifier has the lowest performance across all metrics, with an accuracy of 81.7%. However, GNB is computationally efficient with a very short training time of just 0.086 seconds. Similarly, KNN is the fastest to train, at just 0.022 seconds and also achieves one of the highest accuracies at 93.6%. This makes KNN a hand gesture classifier that offers an excellent trade-off between high accuracy and computational efficiency. Both GNB and KNN are exceptionally fast to train, but KNN offers the best balance between high accuracy and fast processing.

Figure 4.1 illustrates the confusion matrix and Figure 4.2 the Precision-Recall curve for the different algorithms and shows that most models generally perform well in classifying the different gestures. However, they have slight difficulty distinguishing between the "Crossed Fingers" and "Peace Sign" gestures, with some instances of "Crossed" being misclassified as "Peace Sign" as seen confusion matrices. In all models "Disgusted" has the lowest number and the curve for "Crossed" stays lower across all models. This difficulty arises from the significant overlap in the positions of the index and middle fingers, where the keypoints for these fingers and the hand position can be quite similar, making it challenging for the models to distinguish between them due to the small differences in finger arrangement.

Classifier	**Accuracy (%)**	**Precision (%)**	**Recall (%)**	**F1 (%)**	**Training Time (s)**
RF	93.4	93.5	93.5	93.3	97.904
SVM	92.3	92.3	92.2	92.1	147.746
GNB	81.7	82.0	81.7	81.5	**0.086**
KNN	93.6	93.6	93.7	93.6	0.022
MLP	94.0	94.0	94.41	94.0	83.768
CNN	**94.6**	**94.5**	**94.6**	**94.6**	874.192

Table 4.2: Summary of performance metrics for hand gesture recognition.

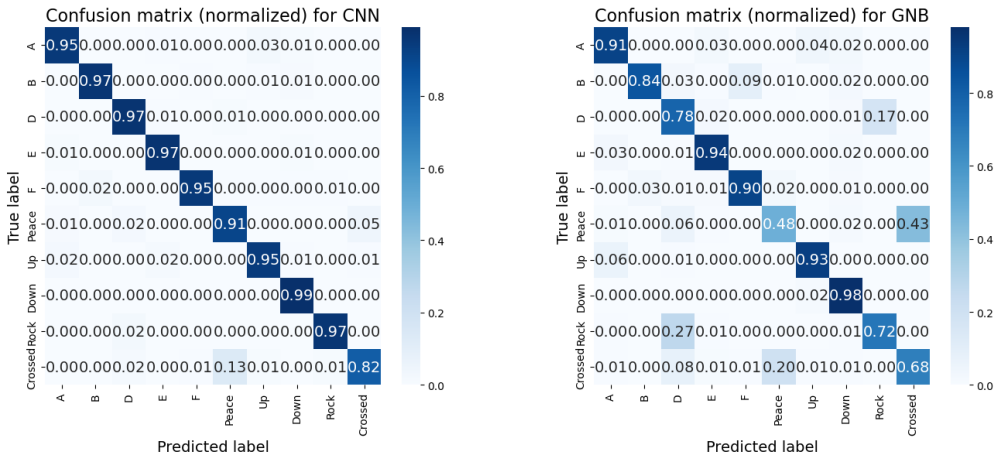


Figure 4.1: Confusion matrices for the best (left) and worst (right) performing models.

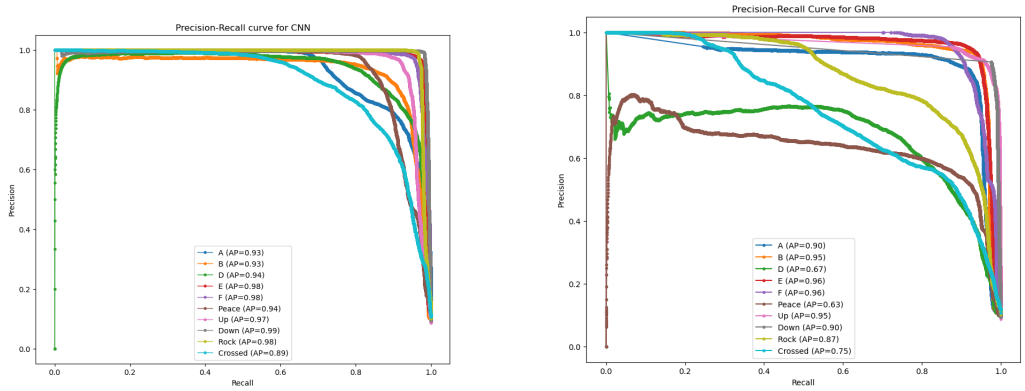


Figure 4.2: Precision-Recall curves for the best (left) and worst (right) performing models.

4.3 Emotion Recognition Evaluation

Same as with the hand gesture classification part, the facial emotion classification algorithms are evaluated based on accuracy, precision, recall, F1 score and training time.

The CNN achieves the highest accuracy at 58.7%, but it has a training time of 736.39 seconds. The RF model ranks as the second-best, with an accuracy of 57.14% and a training time of 130.573 seconds. The MLP also performs well with an accuracy of 56.30% and trains faster than the CNN and RF.

Random Forest's strong performance, which is close to that of the CNN, may be due to the relatively small dataset. With smaller datasets, RF can manage and interpret the data effectively, making it competitive with complex models like CNNs, which generally excel with larger datasets.

K-Nearest Neighbour, with an accuracy of 50.93%, has a performance that falls in the middle but it is the fastest to train at just 0.036 seconds. The SVM classifier performs on the lower end in terms of accuracy, achieving 47.93%. It also has the longest training time, taking 3107.74 seconds. Gaussian Naive Bayes shows the lowest accuracy at 47.10% but on the other hand, has the quickest training time at 0.336 seconds.

All models had lower accuracy than observed in hand gesture recognition, with an overall accuracy of about 55%. This lower accuracy is likely due to frequent confusion between emotions like "Angry" and "Disgusted," which may be caused by the varying ways individuals in the dataset express these emotions, making accurate classification more difficult. The confusion between these emotions is evident in the confusion matrices (see Figure 4.3), where both these emotions display the lowest values and also in the Precision-Recall curve (see Figure 4.3, where across all models these two curves have the lowest curves and their precision drops quickly as recall increases.

Classifier	**Accuracy (%)**	**Precision (%)**	**Recall (%)**	**F1 (%)**	**Training Time (s)**
RF	57.14	57.11	57.48	56.78	130.57
SVM	47.93	50.92	48.72	47.04	3107.74
GNB	47.10	48.14	47.32	47.06	0.336
KNN	50.93	50.44	50.88	50.55	**0.036**
MLP	56.30	56.37	56.71	56.02	86.745
CNN	**58.70**	**59.50**	**59.62**	**58.39**	736.39

Table 4.3: Summary of performance metrics for facial emotion classification.

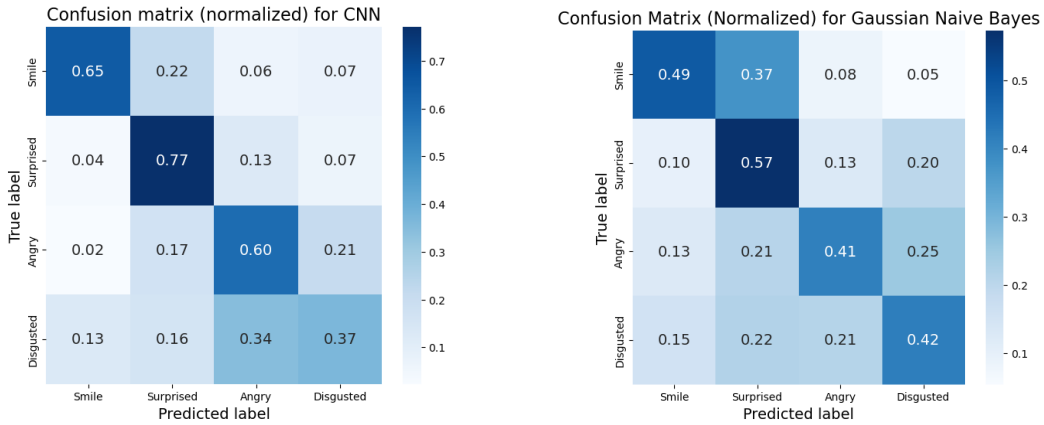


Figure 4.3: Confusion matrices for the best (left) and worst (right) performing models.

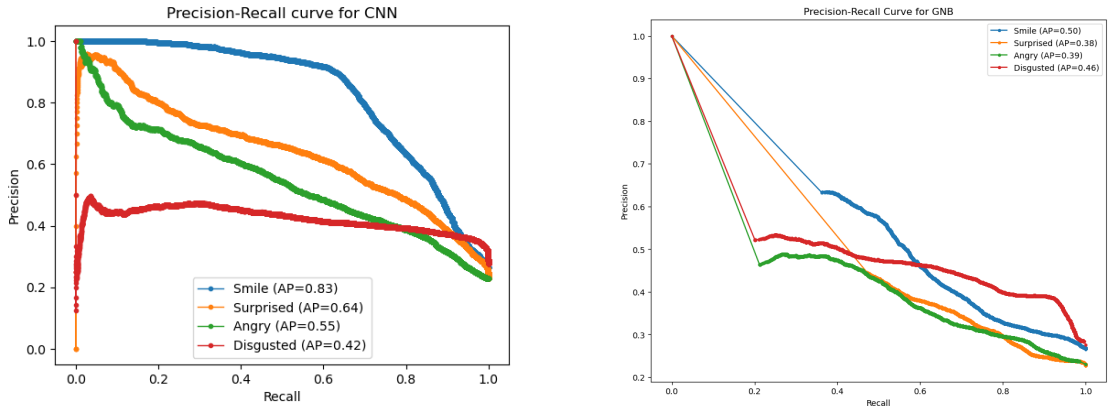


Figure 4.4: Precision-Recall curves for for the best (left) and worst (right) performing models.

4.4 Speech Recognition Evaluation

Table 4.4 shows the overall WER, CER (see Subsection 3.2.3) and average processing time for a sample from one of the 30 individuals in the dataset. Based on metrics in Table 4.4, the Citrinet model stands out with the lowest WER of 10.894% and CER of 4.366%, as well as the fastest average processing time of 0.329 seconds, making it the most efficient model tested, while the CTC Large model performs similarly in terms of WER (10.817%) and CER (3.996%), but took slightly longer to process at 1.131 seconds. The Google model had a WER of 23.740% and a CER of 10.579%, but is considerably slower, with an average processing time of 42.012 seconds. The Whisper Medium model shows intermediate performance, with a WER of 20.205% and a CER of 12.711%. Its processing time is faster than that of Google but slower than that of Citrinet, at 16.364 seconds. The Whisper Large model exhibits highest the WER and CER among the models (31.698% and 24.964%, respectively) and also the slowest processing time (50.507 seconds).

Model	**WER (%)**	**CER (%)**	**Average Processing Time (s)**
Google	23.740	10.579	42.012
Whisper Medium	20.205	12.711	16.364
Whisper Large	31.698	24.964	50.507
Citrinet	10.894	4.366	**0.329**
CTC Large	**10.817**	**3.996**	1.131

Table 4.4: Overall speech recognition metrics for our models.

Table 4.5 provides a detailed breakdown of the different types of speech data present in the dataset. The Citrinet and CTC Large models continue to demonstrate strong performance across most categories.

For the letters category, Whisper Large achieves an exceptionally low CER (0.43%),

outperforming the other models, while Whisper Medium also performs well with a CER of 3.66%. However, Google, Citrinet and CTC Large did not perform as well in letter recognition, with the CER value being higher.

At the word part, the Whisper Large model achieves the lowest WER of 9.03% and CER of 1.97%. Despite the generally low CER values across the models, indicating good accuracy at the character level, the WER values are considerably higher. This discrepancy suggests that models such as Citrinet and CTC Large, despite their robust performance with individual characters, may face greater challenges in accurately transcribing complete words.

When analyzing sentences, both Citrinet and CTC Large models deliver their best results, with sentence-level WER and CER values showing minimal error. Citrinet achieves the lowest sentence WER of 5.84%, followed closely by CTC Large at 4.67%. Their CERs in sentence transcription are also among the lowest, with Citrinet at 1.55% and CTC Large at 1.37%. the Google and Whisper Medium models have moderate performance, with Whisper Large exhibiting the poorest results among the models.

For prosa text, both Citrinet and CTC Large performs similarly well, with Citrinet achieving a WER of 8.37% and CER of 3.98% and CTC Large showing a WER of 8.47% and CER of 3.90%. These results indicate that both models are among the best in recognizing longer stretches of continuous speech, with both high accuracy and efficiency.

There are two possible reasons why the Whisper Large model performs poorly on this dataset. Since the model is multilingual, it tries to recognize multiple languages and can become confused, mistakenly detecting words from other languages. This can result in incorrect transcriptions and lead to higher WER and CER.

Whisper models excel at recognizing letters but struggle with larger speech units, potentially due to their multilingual design, which can lead to unintended language switches and lower performance metrics, especially in the Whisper Large model.

In contrast, Citrinet provides an excellent balance between accuracy and efficiency, making it a reliable option for general speech recognition tasks, while the CTC Large model performs slightly better but takes longer to process. The Google model delivers average results across all metrics, positioning it in the middle tier of performance.

Metric	**Google**	**Whisper Medium**	**Whisper Large**	**Citrinet**	**CTC large**
Letter CER (%)	47.53	3.66	**0.43**	34.19	37.85
Words WER (%)	57.10	77.67	**9.03**	36.77	29.68
Words CER (%)	8.35	13.58	**1.97**	7.73	6.10
Sentence WER (%)	18.47	24.51	31.97	**8.37**	8.47
Sentence CER (%)	12.32	18.62	25.31	3.98	**3.90**
Text WER (%)	16.24	14.52	38.60	5.84	**4.67**
Text CER (%)	6.82	4.366	9.32	1.55	**1.37**

Table 4.5: Speech recognition metrics for various types of data.

Chapter 5

Conclusions

This chapter summarizes the most important findings of the analysis and connects them to the research questions specified in Section 1.4. The goal is to find out which algorithms and models work best for classifying hand gestures and facial emotions and how different speech recognition models compare in terms of accuracy and processing speed.

The first research question asks which algorithms perform best for classifying hand gestures. The results show that all models perform well, with nearly every model achieving more than 90% accuracy. Specifically, CNN achieves the highest accuracy for classifying hand gestures. Multi-Layer Perceptron also perform well, with a slightly lower accuracy but much faster training times, making them a potential alternative in terms of accuracy. This supports our hypothesis since CNNs and MLPs are effective at identifying complex patterns in data.

Random Forest and SVM also show good results but need more training time. Simpler models like GNB and KNN are faster. K-Nearest Neighbor performed exceptionaly well and achieved high accuracy, while GNB had lower accuracy. Random Forest and SVM also show good results but need more training time. K-Nearest Neighbor provides the best trade-off tested between accuracy and training time among all the algorithms in hand gesture classification.

Even with the overall high performance, some challenges appear in hand gesture classification. For example, the models have difficulty telling the difference between gestures like "Crossed Fingers" and the "Peace Sign," which suggests that further improvements in keypoint classification could help increase accuracy.

The second research question looks at which algorithms work best for facial emotion classification. Overall, for facial emotion classification, the metrics are relatively

low, with most models achieving around 55% accuracy. This lower accuracy can be explained by the frequent confusion between emotions such as "Angry" and "Disgusted," which results from the varying ways people express these emotions in the dataset, making accurate classification more challenging.

In this task, the CNN is the best model, with RF being a close second, exhibiting both strong performance and requiring less training time compared to the CNN. The MLP also performs well, with slightly lower accuracy compared to the CNN and RF, but it is faster to train. This result partly confirms our hypothesis that CNN and MLP would perform the best.

K-Nearest Neighbors (KNN) is in the middle in terms of performance, offering moderate accuracy and fast training time. Support Vector Machine and GNB performed the poorest on this dataset, with SVM being slower to train and GNB having lower accuracy.

The third research question focuses on speech recognition. As hypothesized, Citrinet and Conformer CTC Large give the best overall performance, with a good balance between accuracy and how fast they process text. The Whisper models work well for specific tasks, such as recognizing letters, but they are not as effective for longer speech segments. This might be because these models are designed for multiple languages, which affects their performance in some cases.

Chapter 6

Future Work

One potential direction for future work is applying different encoding methods to create new features from the existing keypoint data in the hand gesture and facial emotion dataset. This could provide more useful representations and improve model performance.

Expanding the dataset through 3D face reconstruction might add a new dimension to the data. Another promising avenue involves broadening the dataset by incorporating more tasks, allowing the models to learn from a wider range of gestures and emotions. Introducing more variation in lighting conditions during data collection would also help make the models more resilient to diverse real-world environments, improving their robustness and generalization.

Furthermore, exploring dynamic hand gestures, i.e gestures involving motions, could offer valuable insights into gesture recognition. This exploration could involve analyzing how motion dynamics influence classification and developing new techniques to better classify these gestures. Evaluating more advanced classifiers, such as Long Short-Term Memory (LSTM) networks and incorporating advanced machine learning techniques, such as dictionary learning, could provide new opportunities to identify more effective approaches.

Appendix A

Guide for Data Collection

Appendix A provides detailed instructions for collecting data for our multimodal dataset. It includes guidelines for capturing facial emotions, hand gestures and spoken language.

Anleitung für Probanden

Erfassung eines multimodalen Datensets

Bitte lesen Sie die gesamte Anleitung aufmerksam durch, bevor Sie mit der Erhebung der Daten beginnen.

Die gesamte Erhebung wird etwa 30 Minuten dauern und gliedert sich in die folgenden drei Teile:

1. Im ersten Teil werden Sie verschiedene Gesichtsemotionen vor der Kamera ausdrücken.
2. Für den zweiten Teil werden Sie verschiedene Handgesten in die Kamera zeigen.
3. Abschliessend werden Sie im dritten Teil einige Texte laut ins Mikrofon vorlesen.

Weitere detaillierte Anweisungen zu jedem Teil finden Sie im entsprechenden Abschnitt der Anleitung.

Datum: 11. März 2024

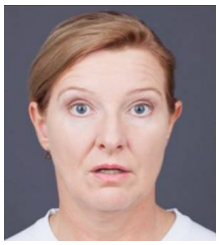
1 Erfassung der Emotionen

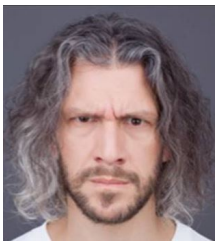
Im folgenden Experiment werden Sie verschiedene Emotionen in die Kamera ausdrücken.

Zu erfassende Emotionen

Bitte drücken Sie folgende Emotionen in die Kamera aus, ähnlich wie im Beispiel.


Lächeln


Überraschung


Ärger/Wut


Verachtung

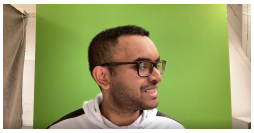
Anleitung zur Erfassung der Emotionen

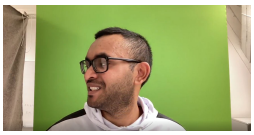
1. Richten Sie zunächst Ihr Gesicht so aus, dass es zentriert und gut sichtbar im Kamerabild ist.
2. Versuchen Sie während der Aufnahme schnelle und plötzliche Bewegungen zu vermeiden.
3. Nennen Sie zuerst die Emotion und zählen Sie dann rückwärts von 5 (zum Beispiel *"Lächeln... 5, 4, 3, 2, 1"*). **Drücken Sie dann für etwa 30 Sekunden die Emotion aus**, während Sie Ihr Gesicht in verschiedenen Winkeln zur Kamera positionieren.

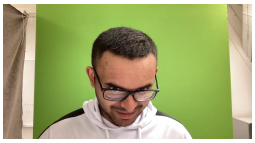
Beispiel für Emotionserfassung:

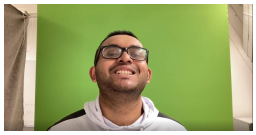
Im Folgenden finden Sie ein Beispiel für die Erfassung der Emotionen anhand von 5 Bildern für das Beispiel "Lächeln":


Frontaler Blick


Gesicht seitlich zeigen


Andere Gesichtsseite


Neigung nach unten


Neigung nach oben

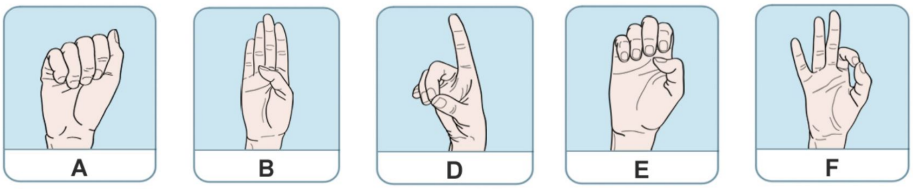
2 Erfassung der Gesten

Im folgenden Experiment werden Sie verschiedene Gesten in die Kamera zeigen.

Zu erfassende Gesten

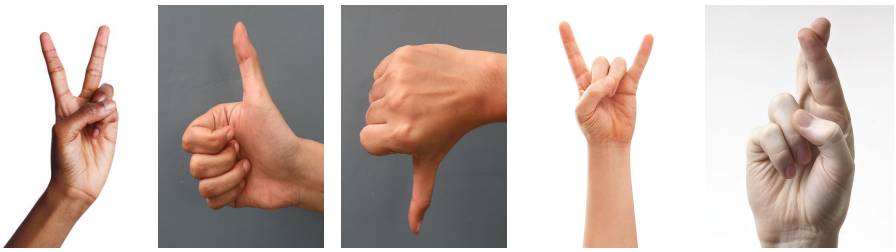
Bitte drücken Sie folgende Gesten in die Kamera aus, ähnlich wie im Beispiel.

Fingeralphabet



A B D E F

Handgesten



Peace Daumen hoch Daumen runter Metal/Rock Gekreuzte Finger

Anleitung zur Erfassung der Gesten

1. Halten Sie jede Geste gut sichtbar in die Kamera.
2. Versuchen Sie während der Aufnahme schnelle und plötzliche Bewegungen zu vermeiden.
3. Üben Sie die Geste zunächst ausserhalb des Kamerabildes und stellen sicher, dass ihre Geste dem Bild ähnelt, bevor Sie die Geste in die Kamera halten. Die Hand, die die Geste nicht ausführt, muss ausserhalb des Kamerabildes bleiben.
4. Bitte halten Sie jede Geste etwa 15 Sekunden lang in die Kamera. Variieren Sie dabei die Position und den Winkel Ihrer Hand. Danach wechseln Sie die Hand und wiederholen den Vorgang.

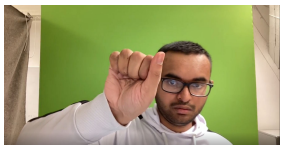
Beispiel für Gestenerkennung

Nun folgt ein Beispiel für die Erfassung von Gesten anhand einiger Bilder, die den Buchstaben "A" im Fingeralphabet darstellen:

Geste frontal halten

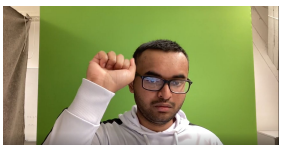
Geste näher heranführen

Geste von der Kamera entfernt halten



Variieren der Handposition

Weitere Variation der Handposition



Dritte Variation der Handposition

3 Erfassung der Sprache

Im folgenden Experiment werden Sie verschiedene Texte vorlesen. Bitte beachten Sie dabei die folgenden Schritte:

1. Bitte lesen Sie die Texte zunächst für sich durch und beginnen Sie anschliessend mit dem lauten Vorlesen.
2. Zuerst werden Sie Buchstaben vorlesen, gefolgt von Wörtern, dann Sätzen und abschließend ein Text.
3. Sprechen Sie deutlich und in normaler Lautstärke in das Mikrofon.
4. Bitte sprechen Sie die Wörter/Sätze in einem angenehmen Tempo für Sie aus. Achten Sie darauf, jedes Wort möglichst korrekt auszusprechen. Sollten Sie ein Fehler beim Vorlesen machen, ist dies kein Problem. Wiederholen Sie den Satz korrekt und fahren Sie dann fort.

Buchstaben

Lesen Sie bitte die folgenden Buchstaben vor:

A C E J Q V Y Z

Wörter

Lesen Sie bitte die folgenden Wörter vor:

Abwärts

Alchemie

Abbaugerechtigkeit

Charisma

Chirurgie

Eichhörnchen

Kläglich

Richtungsorientiert

Schallgeschwindigkeit

Zigarettenstümmel

Sätze

Lesen Sie bitte die folgenden Sätze vor:

Peter sieht zwei schöne Rosen.

Ulrich gibt achtzehn schöne Steine.

Kerstin gibt zwölf teure Messer.

Britta verleiht fünfundzwanzig weisse Dosen.

Der Bauer hat eine Menge Korn geerntet.

Der Stoff ist aus gutem Zwirn gefertigt.

Er hat den hohen Turm schon von weitem gesehen.

Die Fabrik stösst aus dem Schornstein weissen Rauch aus.

Das Boot bot viele technisch fortgeschrittene Eigenschaften.

Das Ass in einem Kartenspiel ist der Joker, während das Aas auf dem Boden ein totes Tier ist.

Obwohl die Sonne bereits untergegangen war, konnte sie immer noch den fernen Horizont erkennen.

Prosatext

Lesen Sie bitte den folgenden Text vor:

> Als Gregor Samsa eines Morgens aus unruhigen Träumen erwachte, fand er sich in seinem Bett zu einem ungeheuren Ungeziefer verwandelt. Er lag auf seinem panzerartig harten Rücken und sah, wenn er den Kopf ein wenig hob, seinen gewölbten, braunen, von bogenförmigen Versteifungen geteilten Bauch, auf dessen Höhe sich die Bettdecke, zum glänzlichen Niedergleiten bereit, kaum noch erhalten konnte.

]

This appendix contains the code for analyzing and evaluating the algorithms used in this thesis.

1 Hand Gesture Analysis

Below is the code for the hand gesture analysis.

1.1 Import Dependencies

```
# Import standard libraries
import time
import itertools
import numpy as np
import csv
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
import random

# Import machine learning models from scikit-learn
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.neural_network import MLPClassifier
from sklearn.neighbors import KNeighborsClassifier

# Import metrics and evaluation tools from scikit-learn
from sklearn.metrics import confusion_matrix, classification_report,
↪  precision_recall_curve, average_precision_score, accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import label_binarize
from sklearn.exceptions import ConvergenceWarning

# Import PyTorch libraries for deep learning
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.utils.data import DataLoader, TensorDataset
```

1.2 Load data and create validation set

```
# Loading the dataset
training = 'model/hand_keypoint_classifier/hand_keypoint_trainingset.csv'
test = 'model/hand_keypoint_classifier/hand_keypoint_testset.csv'

#Create trainingset
X_train = np.loadtxt(training, delimiter=',', dtype='float32', usecols=list(range(2,
↪  (21 * 2) + 2))) # Feature data
y_train = np.loadtxt(training, delimiter=',', dtype='int32', usecols=(1)) # Label data

# Create testset
X_test = np.loadtxt(test, delimiter=',', dtype='float32', usecols=list(range(2, (21 *
↪  2) + 2))) # Feature data
y_test = np.loadtxt(test, delimiter=',', dtype='int32', usecols=(1)) # Label data

# Split the X_train and y_train into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.2,
↪  random_state=42)
```

1.3 Load keypoint mapping

The labels are all saved in a csv file. So we load the labels to later use in the confusion matrix.

```
# Load label mapping from CSV using csv.reader
keypoint_classifier_labels = []
with open('model/hand_keypoint_classifier/hand_keypoint_classifier_label.csv',
↪  encoding='utf-8-sig') as f:
    reader = csv.reader(f)
    for row in reader:
        keypoint_classifier_labels.append(row[0])
```

1.4 Gridsearch and evaluation function

We create a custom gridsearch and evaluation function which would give us the bst parameters as well as the metrics and confusion matrix together with the Recall-Precision curve.

```
def evaluate_classifier(name, classifier, param_grid, X_train, y_train, X_val, y_val,
↪  X_test, y_test, keypoint_classifier_labels):
    """
    Performs grid search for hyperparameters, trains the best model, and evaluates it.

    Parameters:
    - name: str, name of the classifier
    - classifier: sklearn classifier, the classifier to evaluate
    - param_grid: dict, the parameter grid for grid search
    - X_train: array-like, training feature data
    - y_train: array-like, training target data
    - X_val: array-like, validation feature data
    - y_val: array-like, validation target data
    - X_test: array-like, testing feature data
    - y_test: array-like, testing target data
    - keypoint_classifier_labels: list of str, class labels for confusion matrix

    Returns:
    - None, prints results and displays plots
    """

    # Initialize variables to track the best model, parameters, and accuracy
    best_accuracy = 0
    best_parameters = {}
    best_classifier = None

    # Prepare the parameter combinations for the grid search
    keys, values = zip(*param_grid.items())
    param_combinations = [dict(zip(keys, v)) for v in itertools.product(*values)]

    # Iterate through each parameter combination
    for params in param_combinations:
        print(f"Testing parameters: {params}")

        # Set the current combination of hyperparameters
        classifier.set_params(**params)

        # Train the classifier and record the training time
        start_time = time.time()
        classifier.fit(X_train, y_train)
        end_time = time.time()
        training_time = end_time - start_time

        # Evaluate the classifier on the validation set
```

```
        accuracy_val = classifier.score(X_val, y_val)
        print(f"Validation Accuracy: {accuracy_val:.4f}")

        # If the current model is better, update the best model, accuracy, and
        ↪  parameters
        if accuracy_val > best_accuracy:
            best_accuracy = accuracy_val
            best_parameters = params
            best_classifier = classifier

    # Print the best parameters and performance information
    print("Best Parameters for", name, ":", best_parameters)
    print("Training time for", name, ":", training_time, "seconds")
    print("Accuracy on the validation set for", name, ":", best_accuracy)

    # Evaluate the best classifier on the test set
    y_pred_test = best_classifier.predict(X_test)
    test_accuracy = best_classifier.score(X_test, y_test)

    # Print the classification report for the test set
    print("Classification Report on the testing set for", name, ":")
    print(classification_report(y_test, y_pred_test))

    # Plot the confusion matrix for the test set predictions
    conf_matrix = confusion_matrix(y_test, y_pred_test, normalize='true')  # Normalize
    ↪  the confusion matrix
    df_cmx = pd.DataFrame(conf_matrix, index=keypoint_classifier_labels,
    ↪  columns=keypoint_classifier_labels)
    plt.figure(figsize=(7, 6))
    sns.heatmap(df_cmx, annot=True, fmt='.2f', square=False, cmap='Blues',
    ↪  annot_kws={"size": 14})
    plt.xlabel('Predicted label', fontsize=14)
    plt.ylabel('True label', fontsize=14)
    plt.title(f'Confusion Matrix (Normalized) for {name}', fontsize=16)
    plt.show()

    # Display a table with performance metrics
    plt.figure(figsize=(3, 3))
    plt.subplot(1, 1, 1)
    plt.axis('off')

    # Prepare the table with Accuracy, Precision, Recall, and F1-Score
    cell_text = [
        [test_accuracy],
        [classification_report(y_test, y_pred_test, output_dict=True)['macro
        ↪  avg']['precision']],
        [classification_report(y_test, y_pred_test, output_dict=True)['macro
        ↪  avg']['recall']],
        [classification_report(y_test, y_pred_test, output_dict=True)['macro
        ↪  avg']['f1-score']]
    ]

    rows = ['Accuracy', 'Precision', 'Recall', 'F1-Score']
    cols = [name]

    # Plot the performance metrics table
    plt.table(cellText=cell_text, rowLabels=rows, colLabels=cols, loc='center')
    plt.title('Performance Metrics for ' + name)
    plt.show()

    # Binarize the output labels for multiclass precision-recall curves
    y_test_bin = label_binarize(y_test, classes=range(len(keypoint_classifier_labels)))
    y_scores = classifier.predict_proba(X_test)

    # Initialize plotting for Precision-Recall curve
    plt.figure(figsize=(10, 8))
```

```
    # Calculate and plot Precision-Recall curve for each class
    for i, label in enumerate(keypoint_classifier_labels):
        precision, recall, _ = precision_recall_curve(y_test_bin[:, i], y_scores[:, i])
        average_precision = average_precision_score(y_test_bin[:, i], y_scores[:, i],
        ↪  average='macro')

        # Plot the curve for each class
        plt.plot(recall, precision, marker='.', label=f'{label}
        ↪  (AP={average_precision:.2f})', linewidth=1.5)

    # Label the Precision-Recall curve plot
    plt.xlabel('Recall')
    plt.ylabel('Precision')
    plt.title('Precision-Recall Curve for '+ name)
    plt.legend()
    plt.show()
```

1.5 Evaluation of the models

MLP, RF, GNB and KNN are evaluated using the evaluate function. SVM is tested with a slightly different loop to prevent redundant testing.

```
# Multilayer Perceptron (MLP) Grid Search
name = "MLP"
mlp = MLPClassifier(random_state=42)

param_grid = {
    'hidden_layer_sizes': [(10,), (24, 12), (50, 25, 10), (28, 28), (84, 24, 12)],
    'activation': ['relu', 'tanh'],
    'solver': ['lbfgs', 'adam'],
    'max_iter': [10, 20, 50, 100, 200, 500]
}

evaluate_classifier(name, mlp, param_grid, X_train, y_train, X_val, y_val, X_test,
↪  y_test, keypoint_classifier_labels)

# Random Forest Grid Search
name = "RF"
rf = RandomForestClassifier(random_state=42, n_jobs=-1)

param_grid = {
    'n_estimators': [50, 65, 70, 78, 100, 150, 165, 170],
    'criterion': ['entropy', 'gini'],
    'max_depth': [None, 10, 20, 30, 50],
    'min_samples_split': [2, 5, 10, 15],
    'min_samples_leaf': [1, 2, 4],
    'bootstrap': [True, False]
}

evaluate_classifier(name, rf, param_grid, X_train, y_train, X_val, y_val, X_test,
↪  y_test, keypoint_classifier_labels)

# Support Vector Machine (SVM) Grid Search
name = "Multiclass SVM"
classifier = SVC(random_state=42)

param_grid = [
    {
        'C': [0.1, 0.2, 0.3, 0.4, 0.5, 1.0, 1.5, 2.0, 5.0],
        'kernel': ['linear'],
    },
    {
```

```
        'C': [0.1, 0.2, 0.3, 0.4, 0.5, 1.0, 1.5, 2.0, 5.0, 10.0],
        'kernel': ['rbf'],
        'gamma': ['scale', 'auto', 0.1, 0.2, 0.3, 0.4, 0.5, 1.0, 1.5, 2.0]
    }
]

best_accuracy = 0
best_parameters = {}
best_classifier = None

for grid in param_grid:
    keys, values = zip(*grid.items())
    param_combinations = [dict(zip(keys, v)) for v in itertools.product(*values)]

    for params in param_combinations:
        print(f"Testing parameters: {params}")
        classifier.set_params(**params)

        start_time = time.time()

        classifier.fit(X_train, y_train)

        end_time = time.time()
        training_time = end_time - start_time

        accuracy_val = classifier.score(X_val, y_val)
        print(f"Validation Accuracy: {accuracy_val:.4f}")

        if accuracy_val > best_accuracy:
            best_accuracy = accuracy_val
            best_parameters = params
            best_classifier = classifier

print("Best Parameters for SVM:", best_parameters)
print("Training time for SVM:", training_time, "seconds")
print("Accuracy on the validation set for SVM:", best_accuracy)

y_pred_test = best_classifier.predict(X_test)
test_accuracy = best_classifier.score(X_test, y_test)
print("Classification Report on the testing set for SVM:")
print(classification_report(y_test, y_pred_test))

conf_matrix = confusion_matrix(y_test, y_pred_test, normalize='true')
df_cmx = pd.DataFrame(conf_matrix, index=keypoint_classifier_labels,
↪  columns=keypoint_classifier_labels)
plt.figure(figsize=(7, 6))
sns.heatmap(df_cmx, annot=True, fmt='.2f', square=False, cmap='Blues',
↪  annot_kws={"size": 14})
plt.xlabel('Predicted label', fontsize=14)
plt.ylabel('True label', fontsize=14)
plt.title(f'Confusion Matrix (Normalized) for {name}', fontsize=16)
plt.show()

plt.figure(figsize=(5, 3))
plt.subplot(1, 1, 1)
plt.axis('off')
cell_text = [[test_accuracy], [classification_report(y_test, y_pred_test,
↪  output_dict=True)['macro avg']['precision']],
                [classification_report(y_test, y_pred_test, output_dict=True)['macro
                ↪  avg']['recall']],
                [classification_report(y_test, y_pred_test, output_dict=True)['macro
                ↪  avg']['f1-score']]]
rows = ['Accuracy', 'Precision', 'Recall', 'F1-Score']
cols = [name]
plt.table(cellText=cell_text, rowLabels=rows, colLabels=cols, loc='center')
plt.title('Performance Metrics for ' + name)
```

```
plt.show()
# Binarize the output labels for multiclass classification
y_test_bin = label_binarize(y_test, classes=range(len(keypoint_classifier_labels)))
y_scores = classifier.predict_proba(X_test)

    # Initialize plotting
plt.figure(figsize=(10, 8))

    # Calculate and plot Precision-Recall curve for each class
for i, label in enumerate(keypoint_classifier_labels):
    precision, recall, _ = precision_recall_curve(y_test_bin[:, i], y_scores[:, i])
    average_precision = average_precision_score(y_test_bin[:, i], y_scores[:, i],
    ↪  average='macro')

    plt.plot(recall, precision, marker='.', label=f'{label}
    ↪  (AP={average_precision:.2f})', linewidth=1.5)

plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve for '+ name)
plt.legend()
plt.show()

name = "GNB"
gnb = GaussianNB()

param_grid = {
    'var_smoothing': [1e-10, 1e-09, 1e-08, 1e-07, 1e-06, 1e-05, 1e-04]
}

evaluate_classifier(name, gnb, param_grid, X_train, y_train, X_val, y_val, X_test,
↪  y_test, keypoint_classifier_labels)
```

```
# K-Nearest Neighbors (KNN) Grid Search
name = "KNN"
knn = KNeighborsClassifier()

param_grid =  {
        'n_neighbors': [3, 4, 5, 7, 9, 11, 15, 20],
        'weights': ['uniform', 'distance'],
        'metric': ['euclidean', 'manhattan'],
        'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute'],
        'leaf_size': [20, 30, 40, 50]
    }

evaluate_classifier(name, knn, param_grid, X_train, y_train, X_val, y_val, X_test,
↪  y_test, keypoint_classifier_labels)
```

1.6 CNN implementation

```
# Define the hyperparameter grid
param_grid = {
    'num_epochs': [20, 30, 50],
    'learning_rate': [0.001, 0.0001],
    'conv1_out_channels': [16, 32],
    'conv2_out_channels': [32, 64],
    'fc1_out_features': [64, 128],
    'conv_kernel_size': [3],
    'pool_kernel_size': [2],  # Fixed from previous experimentation
    'pool_stride': [2], # Fixed from previous experimentation
    'dropout': [0.0, 0.1, 0.2]
}

# Convert training data to PyTorch tensors
```

```
X_train = torch.tensor(X_train, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.long)

# Convert validation data to PyTorch tensors
X_val = torch.tensor(X_val, dtype=torch.float32)
y_val = torch.tensor(y_val, dtype=torch.long)

# Convert testing data to PyTorch tensors
X_test = torch.tensor(X_test, dtype=torch.float32)
y_test = torch.tensor(y_test, dtype=torch.long)

# Create the train, val and test datasets
train_dataset = TensorDataset(X_train, y_train)
val_dataset = TensorDataset(X_val, y_val)
test_dataset = TensorDataset(X_test, y_test)

# Create a list of all possible combinations of hyperparameters
combinations = list(itertools.product(*(param_grid[param] for param in param_grid)))

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Function to set the seed for reproducibility
def set_seed(seed):
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    np.random.seed(seed)
    random.seed(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

# Function to train and evaluate the model with a given set of hyperparameters
def train_and_evaluate(params):
    set_seed(42)
    num_epochs, learning_rate, conv1_out_channels, conv2_out_channels,
    ↪  fc1_out_features, conv_kernel_size, pool_kernel_size, pool_stride, dropout =
    ↪  params

    class EmotionCNN(nn.Module):
        def __init__(self):
            super(EmotionCNN, self).__init__()
            self.conv1 = nn.Conv1d(in_channels=1, out_channels=conv1_out_channels,
            ↪  kernel_size=conv_kernel_size, padding=conv_kernel_size//2)
            self.pool = nn.MaxPool1d(kernel_size=pool_kernel_size, stride=pool_stride,
            ↪  padding=0)
            self.conv2 = nn.Conv1d(in_channels=conv1_out_channels,
            ↪  out_channels=conv2_out_channels, kernel_size=conv_kernel_size,
            ↪  padding=conv_kernel_size//2)
            self.flatten = nn.Flatten()

            # Use a dummy input to calculate the size after conv2 and pooling layers
            with torch.no_grad():
                self._dummy_input = torch.zeros(1, 1, X_train.size(1))
                self._dummy_output_size = self._get_conv_output_size()

            self.fc1 = nn.Linear(self._dummy_output_size, fc1_out_features)
            self.fc2 = nn.Linear(fc1_out_features, 10)
            self.dropout = nn.Dropout(p=dropout)

        def _get_conv_output_size(self):
            x = self.pool(F.relu(self.conv1(self._dummy_input)))
            x = self.pool(F.relu(self.conv2(x)))
            x = self.flatten(x)
            return x.size(1)

        def forward(self, x):
```

```
            x = self.pool(F.relu(self.conv1(x)))
            x = self.pool(F.relu(self.conv2(x)))
            x = self.flatten(x)
            x = F.relu(self.fc1(x))
            x = self.dropout(x)
            x = self.fc2(x)
            return x

    # Check if a GPU is available
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    # Instantiate the model
    model = EmotionCNN().to(device)

    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)

    train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True,
    ↪  generator=torch.Generator().manual_seed(42))
    val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False,
    ↪  generator=torch.Generator().manual_seed(42))

    start_time = time.time()
    # Training loop
    for epoch in range(num_epochs):
        model.train()
        for inputs, labels in train_loader:
            inputs = inputs.unsqueeze(1).to(device)
            labels = labels.to(device)

            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
    training_time = time.time() - start_time

    # Validation loop
    model.eval()
    all_preds = []
    all_labels = []
    all_scores = []
    with torch.no_grad():
        for inputs, labels in val_loader:
            inputs = inputs.unsqueeze(1).to(device)
            labels = labels.to(device)
            outputs = model(inputs)
            _, preds = torch.max(outputs, 1)
            all_preds.extend(preds.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())
            all_scores.extend(outputs.cpu().numpy())  # Collect raw scores

    accuracy = accuracy_score(all_labels, all_preds)
    return model, accuracy, training_time, np.array(all_scores), np.array(all_labels)

# Track the best hyperparameters and accuracy
best_accuracy = 0
best_params = None
best_model = None
best_training_time = None
best_scores = None
best_labels = None

# Iterate over all combinations of hyperparameters
for params in combinations:
    model, accuracy, training_time, scores, labels = train_and_evaluate(params)
```

```
    print(f'Parameters: {params} -> Validation Accuracy: {accuracy:.4f} | Training
    ↪  Time: {training_time:.2f} seconds')
    if accuracy > best_accuracy:
        best_accuracy = accuracy
        best_params = params
        best_model = model
        best_training_time = training_time
        best_scores = scores
        best_labels = labels

print(f'Best Accuracy: {best_accuracy:.4f}')
print(f'Best Hyperparameters: {best_params}')

# Evaluate the best model on the test set
best_model.eval()
with torch.no_grad():
    test_loader = DataLoader(TensorDataset(X_test, y_test), batch_size=32,
    ↪  shuffle=False)
    all_preds = []
    all_labels = []
    all_scores = []
    for inputs, labels in test_loader:
        inputs = inputs.unsqueeze(1).to(device)
        labels = labels.to(device)
        outputs = best_model(inputs)
        _, preds = torch.max(outputs, 1)
        all_preds.extend(preds.cpu().numpy())
        all_labels.extend(labels.cpu().numpy())
        all_scores.extend(outputs.cpu().numpy())

# Print classification report
print("Classification Report on the testing set:")
print(classification_report(all_labels, all_preds))

# Plot confusion matrix
confusion_matrix_test = confusion_matrix(all_labels, all_preds, normalize='true')
# keypoint_classifier_labels = [str(i) for i in range(4)]  # Adjust labels as needed
df_cmx = pd.DataFrame(confusion_matrix_test, index=keypoint_classifier_labels,
↪  columns=keypoint_classifier_labels)
plt.figure(figsize=(7, 6))
sns.heatmap(df_cmx, annot=True, fmt='.2f', square=False, cmap='Blues',
↪  annot_kws={"size": 14})
plt.xlabel('Predicted label', fontsize=14)
plt.ylabel('True label', fontsize=14)
plt.title('Confusion matrix (normalized) for CNN', fontsize=16)
plt.show()

# Create a table for performance metrics
test_accuracy = accuracy_score(all_labels, all_preds)
report = classification_report(all_labels, all_preds, output_dict=True)
precision = report['macro avg']['precision']
recall = report['macro avg']['recall']
f1_score = report['macro avg']['f1-score']

plt.figure(figsize=(10, 3))
plt.axis('off')
cell_text = [[test_accuracy], [precision], [recall], [f1_score]]
rows = ['Accuracy', 'Precision', 'Recall', 'F1-Score']
cols = ['Best Model']
plt.table(cellText=cell_text, rowLabels=rows, colLabels=cols, loc='center')
plt.title('Performance Metrics for CNN')
plt.show()

# Plot Precision-Recall curve
y_test_bin = label_binarize(np.array(all_labels), classes=np.arange(410))
y_scores = np.array(all_scores)
```

```
precision = {}
recall = {}
average_precision = {}

plt.figure(figsize=(10, 8))

for i in range(10):
    precision[i], recall[i], _ = precision_recall_curve(y_test_bin[:, i], y_scores[:,
    ↪  i])
    average_precision[i] = average_precision_score(y_test_bin[:, i], y_scores[:, i])

for i, label in enumerate(keypoint_classifier_labels):
    precision, recall, _ = precision_recall_curve(y_test_bin[:, i], y_scores[:, i])
    average_precision = average_precision_score(y_test_bin[:, i], y_scores[:, i],
    ↪  average='macro')

    plt.plot(recall, precision, marker='.', label=f'{label}
    ↪  (AP={average_precision:.2f})', linewidth=1)

plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall curve for CNN')
plt.legend(loc='best')
plt.show()
```

2 Hand Gesture Analysis

Below is the code for the emotion recognition analysis. It is nearly the same code as for the gesture recognition part with changes regarding hyperparameters and other minor changes.

2.1 Import Dependencies

```
# Import standard libraries
import time
import itertools
import numpy as np
import csv
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
import random

# Import machine learning models from scikit-learn
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.neural_network import MLPClassifier
from sklearn.neighbors import KNeighborsClassifier

# Import metrics and evaluation tools from scikit-learn
from sklearn.metrics import confusion_matrix, classification_report,
↪  precision_recall_curve, average_precision_score, accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import label_binarize
from sklearn.exceptions import ConvergenceWarning

# Import PyTorch libraries for deep learning
import torch
import torch.nn as nn
import torch.optim as optim
```

```
import torch.nn.functional as F
from torch.utils.data import DataLoader, TensorDataset
```

2.2 Load data and create validation set

```
# Loading the dataset
training = 'D:\\BA\\face_keypoint_trainingset3c.csv'
test = 'D:\\BA\\face_keypoint_testset3c.csv'

# Create trainingsset.
X_train_full = np.loadtxt(training, delimiter=',', dtype='float32',
↪  usecols=list(range(2, (936)+2))) # Feature data (936 features)
y_train_full = np.loadtxt(training, delimiter=',', dtype='int32', usecols=(1)) # Label
↪  data

# Create testset.
X_test = np.loadtxt(test, delimiter=',', dtype='float32', usecols=list(range(2,
↪  (936)+2))) # Feature data
y_test = np.loadtxt(test, delimiter=',', dtype='int32', usecols=(1)) # Label data

# Split the training data into training and validation sets (80% train, 20% validation)
X_train, X_val, y_train, y_val = train_test_split(X_train_full, y_train_full,
↪  test_size=0.2, random_state=42)

# Load label mapping from CSV using csv.reader
keypoint_classifier_labels = []
with open('model/face_keypoint_classifier/face_keypoint_classifier_label.csv',
↪  encoding='utf-8-sig') as f:
    reader = csv.reader(f)
    for row in reader:
        keypoint_classifier_labels.append(row[0])
```

2.3 Load keypoint mapping

```
# Load label mapping from CSV using csv.reader
keypoint_classifier_labels = []
with open('model/hand_keypoint_classifier/hand_keypoint_classifier_label.csv',
↪  encoding='utf-8-sig') as f:
    reader = csv.reader(f)
    for row in reader:
        keypoint_classifier_labels.append(row[0])
```

2.4 Gridsearch and evaluation function

```
def evaluate_classifier(name, classifier, param_grid, X_train, y_train, X_val, y_val,
↪  X_test, y_test, keypoint_classifier_labels):
    """
    Performs grid search for hyperparameters, trains the best model, and evaluates it.

    Parameters:
    - name: str, name of the classifier
    - classifier: sklearn classifier, the classifier to evaluate
    - param_grid: dict, the parameter grid for grid search
    - X_train: array-like, training feature data
    - y_train: array-like, training target data
    - X_val: array-like, validation feature data
    - y_val: array-like, validation target data
    - X_test: array-like, testing feature data
    - y_test: array-like, testing target data
    - keypoint_classifier_labels: list of str, class labels for confusion matrix
```

```
    Returns:
    - None, prints results and displays plots
    """

    best_accuracy = 0
    best_parameters = {}
    best_classifier = None

    keys, values = zip(*param_grid.items())
    param_combinations = [dict(zip(keys, v)) for v in itertools.product(*values)]

    for params in param_combinations:
        print(f"Testing parameters: {params}")
        classifier.set_params(**params)

        start_time = time.time()
        classifier.fit(X_train, y_train)
        end_time = time.time()
        training_time = end_time - start_time

        accuracy_val = classifier.score(X_val, y_val)
        print(f"Validation Accuracy: {accuracy_val:.4f}")

        if accuracy_val > best_accuracy:
            best_accuracy = accuracy_val
            best_parameters = params
            best_classifier = classifier

    print("Best Parameters for", name, ":", best_parameters)
    print("Training time for", name, ":", training_time, "seconds")
    print("Accuracy on the validation set for", name, ":", best_accuracy)

    # Evaluate the best classifier on the testing set
    y_pred_test = best_classifier.predict(X_test)
    test_accuracy = best_classifier.score(X_test, y_test)
    print("Classification Report on the testing set for", name, ":")
    print(classification_report(y_test, y_pred_test))

    # Plot confusion matrix
    conf_matrix = confusion_matrix(y_test, y_pred_test, normalize='true')
    df_cmx = pd.DataFrame(conf_matrix, index=keypoint_classifier_labels,
    ↪  columns=keypoint_classifier_labels)
    plt.figure(figsize=(7, 6))
    sns.heatmap(df_cmx, annot=True, fmt='.2f', square=False, cmap='Blues',
    ↪  annot_kws={"size": 14})
    plt.xlabel('Predicted label', fontsize=14)
    plt.ylabel('True label', fontsize=14)
    plt.title(f'Confusion Matrix (Normalized) for {name}', fontsize=16)
    plt.show()

    plt.figure(figsize=(3, 3))
    plt.subplot(1, 1, 1)
    plt.axis('off')
    cell_text = [[test_accuracy], [classification_report(y_test, y_pred_test,
    ↪  output_dict=True)['macro avg']['precision']],
                 [classification_report(y_test, y_pred_test, output_dict=True)['macro
                 ↪  avg']['recall']],
                 [classification_report(y_test, y_pred_test, output_dict=True)['macro
                 ↪  avg']['f1-score']]]
    rows = ['Accuracy', 'Precision', 'Recall', 'F1-Score']
    cols = [name]
    plt.table(cellText=cell_text, rowLabels=rows, colLabels=cols, loc='center')
    plt.title('Performance Metrics for ' + name)
    plt.show()

    # Binarize the output labels for multiclass classification
```

```
    y_test_bin = label_binarize(y_test, classes=range(len(keypoint_classifier_labels)))
    y_scores = classifier.predict_proba(X_test)

    # Initialize plotting
    plt.figure(figsize=(10, 8))

    # Calculate and plot Precision-Recall curve for each class
    for i, label in enumerate(keypoint_classifier_labels):
        precision, recall, _ = precision_recall_curve(y_test_bin[:, i], y_scores[:, i])
        average_precision = average_precision_score(y_test_bin[:, i], y_scores[:, i],
        ↪  average='macro')

        plt.plot(recall, precision, marker='.', label=f'{label}
        ↪  (AP={average_precision:.2f})', linewidth=1.5)

    plt.xlabel('Recall')
    plt.ylabel('Precision')
    plt.title('Precision-Recall Curve for '+ name)
    plt.legend()
    plt.show()
```

2.5 Evaluation of the models

```
# Multilayer Perceptron (MLP) Grid Search
name = "MLP"
mlp = MLPClassifier(random_state=42)

param_grid = {
    'hidden_layer_sizes': [(512, 256, 128, 64, 32), (128, 64, 32), (32, 16)],
    'activation': ['relu', 'tanh'],
    'solver': ['lbfgs', 'adam'],
    'max_iter': [100, 250, 500, 750]
}

evaluate_classifier(name, mlp, param_grid, X_train, y_train, X_val, y_val, X_test,
↪  y_test, keypoint_classifier_labels)

# Random Forest Grid Search
name = "RF"
rf = RandomForestClassifier(random_state=42, n_jobs=-1)

param_grid = {
    'n_estimators': [50, 75, 100, 150, 200, 250],
    'criterion': ['entropy', 'gini'],
    'max_depth': [None, 10, 20, 30, 50],
    'min_samples_split': [2, 5, 10, 15],
    'min_samples_leaf': [1, 2, 4],
    'bootstrap': [True, False]
}

evaluate_classifier(name, rf, param_grid, X_train, y_train, X_val, y_val, X_test,
↪  y_test, keypoint_classifier_labels)

# Support Vector Machine (SVM) Grid Search
name = "Multiclass SVM"
classifier = SVC(random_state=42)

param_grid = [
    {
        'C': [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.5, 2.0, 5.0, 10.0],
        'kernel': ['linear'],
    },
    {
```

```
        'C': [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.5, 2.0, 5.0, 10.0],
        'kernel': ['rbf'],
        'gamma': ['scale', 'auto', 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0,
        ↪  1.5, 2.0]
    }
]

best_accuracy = 0
best_parameters = {}
best_classifier = None

for grid in param_grid:
    keys, values = zip(*grid.items())
    param_combinations = [dict(zip(keys, v)) for v in itertools.product(*values)]

    for params in param_combinations:
        print(f"Testing parameters: {params}")
        classifier.set_params(**params)

        start_time = time.time()

        classifier.fit(X_train, y_train)

        end_time = time.time()
        training_time = end_time - start_time

        accuracy_val = classifier.score(X_val, y_val)
        print(f"Validation Accuracy: {accuracy_val:.4f}")

        if accuracy_val > best_accuracy:
            best_accuracy = accuracy_val
            best_parameters = params
            best_classifier = classifier

print("Best Parameters for SVM:", best_parameters)
print("Training time for SVM:", training_time, "seconds")
print("Accuracy on the validation set for SVM:", best_accuracy)

y_pred_test = best_classifier.predict(X_test)
test_accuracy = best_classifier.score(X_test, y_test)
print("Classification Report on the testing set for SVM:")
print(classification_report(y_test, y_pred_test))

conf_matrix = confusion_matrix(y_test, y_pred_test, normalize='true')
df_cmx = pd.DataFrame(conf_matrix, index=keypoint_classifier_labels,
↪  columns=keypoint_classifier_labels)
plt.figure(figsize=(7, 6))
sns.heatmap(df_cmx, annot=True, fmt='.2f', square=False, cmap='Blues',
↪  annot_kws={"size": 14})
plt.xlabel('Predicted label', fontsize=14)
plt.ylabel('True label', fontsize=14)
plt.title(f'Confusion Matrix (Normalized) for {name}', fontsize=16)
plt.show()

plt.figure(figsize=(5, 3))
plt.subplot(1, 1, 1)
plt.axis('off')
cell_text = [[test_accuracy], [classification_report(y_test, y_pred_test,
↪  output_dict=True)['macro avg']['precision']],
                [classification_report(y_test, y_pred_test, output_dict=True)['macro
                ↪  avg']['recall']],
                [classification_report(y_test, y_pred_test, output_dict=True)['macro
                ↪  avg']['f1-score']]]
rows = ['Accuracy', 'Precision', 'Recall', 'F1-Score']
cols = [name]
plt.table(cellText=cell_text, rowLabels=rows, colLabels=cols, loc='center')
```

```
plt.title('Performance Metrics for ' + name)
plt.show()
# Binarize the output labels for multiclass classification
y_test_bin = label_binarize(y_test, classes=range(len(keypoint_classifier_labels)))
y_scores = classifier.predict_proba(X_test)

    # Initialize plotting
plt.figure(figsize=(10, 8))

    # Calculate and plot Precision-Recall curve for each class
for i, label in enumerate(keypoint_classifier_labels):
    precision, recall, _ = precision_recall_curve(y_test_bin[:, i], y_scores[:, i])
    average_precision = average_precision_score(y_test_bin[:, i], y_scores[:, i],
    ↪  average='macro')

    plt.plot(recall, precision, marker='.', label=f'{label}
    ↪  (AP={average_precision:.2f})', linewidth=1.5)

plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve for '+ name)
plt.legend()
plt.show()

name = "GNB"
gnb = GaussianNB()

param_grid = {
    'var_smoothing': [1e-10, 1e-09, 1e-08, 1e-07, 1e-06, 1e-05, 1e-04]
}

evaluate_classifier(name, gnb, param_grid, X_train, y_train, X_val, y_val, X_test,
↪  y_test, keypoint_classifier_labels)

# K-Nearest Neighbors (KNN) Grid Search
name = "KNN"
knn = KNeighborsClassifier()

param_grid =  {
        'n_neighbors': [3, 4, 5, 7, 9, 11, 15, 20],
        'weights': ['uniform', 'distance'],
        'metric': ['euclidean', 'manhattan'],
        'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute'],
        'leaf_size': [20, 30, 40, 50]
    }

evaluate_classifier(name, knn, param_grid, X_train, y_train, X_val, y_val, X_test,
↪  y_test, keypoint_classifier_labels)
```

2.6 CNN implementation

```
# Define the hyperparameter grid
param_grid = {
    'num_epochs': [20, 30, 50],
    'learning_rate': [0.001, 0.0001],
    'conv1_out_channels': [32, 64],
    'conv2_out_channels': [64, 128],
    'fc1_out_features': [128, 256],
    'conv_kernel_size': [3, 5],
    'pool_kernel_size': [2],  # Fixed from previous experimentation
    'pool_stride': [2]  # Fixed from previous experimentation
}
```

```
# Convert training data to PyTorch tensors
X_train = torch.tensor(X_train, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.long)

# Convert validation data to PyTorch tensors
X_val = torch.tensor(X_val, dtype=torch.float32)
y_val = torch.tensor(y_val, dtype=torch.long)

# Convert testing data to PyTorch tensors
X_test = torch.tensor(X_test, dtype=torch.float32)
y_test = torch.tensor(y_test, dtype=torch.long)

# Create the train, val and test datasets
train_dataset = TensorDataset(X_train, y_train)
val_dataset = TensorDataset(X_val, y_val)
test_dataset = TensorDataset(X_test, y_test)

# Create a list of all possible combinations of hyperparameters
combinations = list(itertools.product(*(param_grid[param] for param in param_grid)))

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Function to set the seed for reproducibility
def set_seed(seed):
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    np.random.seed(seed)
    random.seed(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

# Function to train and evaluate the model with a given set of hyperparameters
def train_and_evaluate(params):
    set_seed(42)
    num_epochs, learning_rate, conv1_out_channels, conv2_out_channels,
    ↪  fc1_out_features, conv_kernel_size, pool_kernel_size, pool_stride, dropout =
    ↪  params

    class EmotionCNN(nn.Module):
        def __init__(self):
            super(EmotionCNN, self).__init__()
            self.conv1 = nn.Conv1d(in_channels=1, out_channels=conv1_out_channels,
            ↪  kernel_size=conv_kernel_size, padding=conv_kernel_size//2)
            self.pool = nn.MaxPool1d(kernel_size=pool_kernel_size, stride=pool_stride,
            ↪  padding=0)
            self.conv2 = nn.Conv1d(in_channels=conv1_out_channels,
            ↪  out_channels=conv2_out_channels, kernel_size=conv_kernel_size,
            ↪  padding=conv_kernel_size//2)
            self.flatten = nn.Flatten()

            # Use a dummy input to calculate the size after conv2 and pooling layers
            with torch.no_grad():
                self._dummy_input = torch.zeros(1, 1, X_train.size(1))
                self._dummy_output_size = self._get_conv_output_size()

            self.fc1 = nn.Linear(self._dummy_output_size, fc1_out_features)
            self.fc2 = nn.Linear(fc1_out_features, 10)
            self.dropout = nn.Dropout(p=dropout)

        def _get_conv_output_size(self):
            x = self.pool(F.relu(self.conv1(self._dummy_input)))
            x = self.pool(F.relu(self.conv2(x)))
            x = self.flatten(x)
            return x.size(1)
```

```
        def forward(self, x):
            x = self.pool(F.relu(self.conv1(x)))
            x = self.pool(F.relu(self.conv2(x)))
            x = self.flatten(x)
            x = F.relu(self.fc1(x))
            x = self.dropout(x)
            x = self.fc2(x)
            return x

    # Check if a GPU is available
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    # Instantiate the model
    model = EmotionCNN().to(device)

    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)

    train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True,
    ↪  generator=torch.Generator().manual_seed(42))
    val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False,
    ↪  generator=torch.Generator().manual_seed(42))

    start_time = time.time()
    # Training loop
    for epoch in range(num_epochs):
        model.train()
        for inputs, labels in train_loader:
            inputs = inputs.unsqueeze(1).to(device)
            labels = labels.to(device)

            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
    training_time = time.time() - start_time

    # Validation loop
    model.eval()
    all_preds = []
    all_labels = []
    all_scores = []
    with torch.no_grad():
        for inputs, labels in val_loader:
            inputs = inputs.unsqueeze(1).to(device)
            labels = labels.to(device)
            outputs = model(inputs)
            _, preds = torch.max(outputs, 1)
            all_preds.extend(preds.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())
            all_scores.extend(outputs.cpu().numpy())  # Collect raw scores

    accuracy = accuracy_score(all_labels, all_preds)
    return model, accuracy, training_time, np.array(all_scores), np.array(all_labels)

# Track the best hyperparameters and accuracy
best_accuracy = 0
best_params = None
best_model = None
best_training_time = None
best_scores = None
best_labels = None

# Iterate over all combinations of hyperparameters
```

```
for params in combinations:
    model, accuracy, training_time, scores, labels = train_and_evaluate(params)
    print(f'Parameters: {params} -> Validation Accuracy: {accuracy:.4f} | Training
    ↪  Time: {training_time:.2f} seconds')
    if accuracy > best_accuracy:
        best_accuracy = accuracy
        best_params = params
        best_model = model
        best_training_time = training_time
        best_scores = scores
        best_labels = labels

print(f'Best Accuracy: {best_accuracy:.4f}')
print(f'Best Hyperparameters: {best_params}')

# Evaluate the best model on the test set
best_model.eval()
with torch.no_grad():
    test_loader = DataLoader(TensorDataset(X_test, y_test), batch_size=32,
    ↪  shuffle=False)
    all_preds = []
    all_labels = []
    all_scores = []
    for inputs, labels in test_loader:
        inputs = inputs.unsqueeze(1).to(device)
        labels = labels.to(device)
        outputs = best_model(inputs)
        _, preds = torch.max(outputs, 1)
        all_preds.extend(preds.cpu().numpy())
        all_labels.extend(labels.cpu().numpy())
        all_scores.extend(outputs.cpu().numpy())

# Print classification report
print("Classification Report on the testing set:")
print(classification_report(all_labels, all_preds))

# Plot confusion matrix
confusion_matrix_test = confusion_matrix(all_labels, all_preds, normalize='true')
# keypoint_classifier_labels = [str(i) for i in range(4)]  # Adjust labels as needed
df_cmx = pd.DataFrame(confusion_matrix_test, index=keypoint_classifier_labels,
↪  columns=keypoint_classifier_labels)
plt.figure(figsize=(7, 6))
sns.heatmap(df_cmx, annot=True, fmt='.2f', square=False, cmap='Blues',
↪  annot_kws={"size": 14})
plt.xlabel('Predicted label', fontsize=14)
plt.ylabel('True label', fontsize=14)
plt.title('Confusion matrix (normalized) for CNN', fontsize=16)
plt.show()

# Create a table for performance metrics
test_accuracy = accuracy_score(all_labels, all_preds)
report = classification_report(all_labels, all_preds, output_dict=True)
precision = report['macro avg']['precision']
recall = report['macro avg']['recall']
f1_score = report['macro avg']['f1-score']

plt.figure(figsize=(10, 3))
plt.axis('off')
cell_text = [[test_accuracy], [precision], [recall], [f1_score]]
rows = ['Accuracy', 'Precision', 'Recall', 'F1-Score']
cols = ['Best Model']
plt.table(cellText=cell_text, rowLabels=rows, colLabels=cols, loc='center')
plt.title('Performance Metrics for CNN')
plt.show()

# Plot Precision-Recall curve
```

```
y_test_bin = label_binarize(np.array(all_labels), classes=np.arange(4))
y_scores = np.array(all_scores)
precision = {}
recall = {}
average_precision = {}

plt.figure(figsize=(10, 8))
for i in range(4):
    precision[i], recall[i], _ = precision_recall_curve(y_test_bin[:, i], y_scores[:,
    ↪  i])
    average_precision[i] = average_precision_score(y_test_bin[:, i], y_scores[:, i])

for i, label in enumerate(keypoint_classifier_labels):
    precision, recall, _ = precision_recall_curve(y_test_bin[:, i], y_scores[:, i])
    average_precision = average_precision_score(y_test_bin[:, i], y_scores[:, i],
    ↪  average='macro')

    plt.plot(recall, precision, marker='.', label=f'{label}
    ↪  (AP={average_precision:.2f})', linewidth=1)

plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall curve for CNN')
plt.legend(loc='best')
plt.show()
```

3 Speech Analysis

Below is the code for speech analysis.

3.1 Import Dependencies

```
import os
import speech_recognition as sr
import csv
import string
import numpy as np
import time

# Imports the models and evaluation library
import speech_recognition as sr
from jiwer import wer, cer
import nemo.collections.asr as nemo_asr
import whisper
import torch
```

3.2 Define the models

```
def measure_time(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        duration = end_time - start_time
        return result, duration
    return wrapper

@measure_time
def speech_recognition_google(audio_file, language='de-DE'):
    recognizer = sr.Recognizer()
```

```
    with sr.AudioFile(audio_file) as source:
        audio_data = recognizer.record(source)
    try:
        transcription = recognizer.recognize_google(audio_data, language=language)
        return transcription
    except sr.UnknownValueError:
        print("Google Speech Recognition could not understand the audio")
        return ""
    except sr.RequestError as e:
        print("Could not request results from Google Speech Recognition service;
        ↪  {0}".format(e))
        return ""

@measure_time
def speech_recognition_nemo(audio_file, asr_model):
    transcription = asr_model.transcribe([audio_file])
    return transcription[0]

@measure_time
def speech_recognition_whisper(audio_file, model, model_name):
    result = model.transcribe(audio_file)
    return result["text"]

def preprocess_text(text):
    text = text.lower()
    text = text.replace("ß", "ss")
    text = text.translate(str.maketrans('', '', string.punctuation))
    return text

def save_transcription_to_csv(transcriptions, csv_filename):
    with open(csv_filename, 'a', newline='', encoding='utf-8') as csvfile:
        writer = csv.writer(csvfile)
        for transcription in transcriptions:
            writer.writerow([transcription.strip()])

def calculate_metrics(reference, hypothesis):
    wer_score = wer(reference, hypothesis)
    cer_score = cer(reference, hypothesis)
    return wer_score, cer_score

def save_metrics_to_csv(metrics, csv_filename):
    with open(csv_filename, 'a', newline='', encoding='utf-8') as csvfile:
        writer = csv.writer(csvfile)
        for key, values in metrics.items():
            for wer_score, cer_score in zip(values['wer'], values['cer']):
                writer.writerow([key, wer_score, cer_score])
```

3.3 Evalutation Loop

This code processes audio files for speech recognition with various models, including Google, NeMo CTC Large, NeMo Citrinet, and Whisper, evaluating their transcriptions against a reference text. It computes metrics (WER and CER), saves individual and overall results to CSV files, and manages GPU memory by clearing it after each model's use.

```
# Folder containing audio files
audio_folder_path = 'speech_data_mono'
```

```
reference_transcription = "a c e j q v y z abwärts alchemie abbaugerechtigkeit charisma
↪  chirurgie eichhörnchen kläglich richtungsorientiert schallgeschwindigkeit
↪  zigarettenstümmel peter sieht zwei schöne rosen ulrich gibt achtzehn schöne steine
↪  kerstin gibt zwölf teure messer brita verleiht fünfundzwanzig weisse dosen der
↪  bauer hat eine menge korn geerntet der stoff ist aus gutem zwirn gefertigt er hat
↪  den hohen turm schon von weitem gesehen die fabrik stösst aus dem schornstein
↪  weissen rauch aus das boot bot viele technisch fortgeschrittene eigenschaften das
↪  ass in einem kartenspiel ist der joker während das aas auf dem boden ein totes tier
↪  ist obwohl die sonne bereits untergegangen war konnte sie immer noch den fernen
↪  horizont erkennen als gregor samsa eines morgens aus unruhigen träumen erwachte
↪  fand er sich in seinem bett zu einem ungeheuren ungeziefer verwandelt er lag auf
↪  seinem panzerartig harten rücken und sah wenn er den kopf ein wenig hob seinen
↪  gewölbten braunen von bogenförmigen versteifungen geteilten bauch auf dessen höhe
↪  sich die bettdecke zum glänzlichen niedergleiten bereit kaum noch erhalten konnte"

# Initialize dictionaries to hold overall metrics
overall_metrics = {
    'Google': {'wer': [], 'cer': [], 'time': []},
    'CTC Large': {'wer': [], 'cer': [], 'time': []},
    'Citrinet': {'wer': [], 'cer': [], 'time': []},
    'Whisper Large': {'wer': [], 'cer': [], 'time': []},
    'Whisper Medium': {'wer': [], 'cer': [], 'time': []}
}

# Process each audio file in the folder
for audio_file in os.listdir(audio_folder_path):
    audio_file_path = os.path.join(audio_folder_path, audio_file)

    # Perform speech recognition sequentially for all algorithms except Whisper
    transcriptions = {}
    times = {}

    # Google
    transcriptions['Google'], times['Google'] =
    ↪  speech_recognition_google(audio_file_path)

    # NeMo CTC Large
    asr_model_ctc_large =
    ↪  nemo_asr.models.EncDecCTCModelBPE.from_pretrained(model_name="stt_de_conformer_ctc_large")
    transcriptions['CTC Large'], times['CTC Large'] =
    ↪  speech_recognition_nemo(audio_file_path, asr_model_ctc_large)
    del asr_model_ctc_large  # Free up memory
    torch.cuda.empty_cache()

    # NeMo Citrinet
    asr_model_citrinet =
    ↪  nemo_asr.models.EncDecCTCModelBPE.from_pretrained(model_name="stt_de_citrinet_1024")
    transcriptions['Citrinet'], times['Citrinet'] =
    ↪  speech_recognition_nemo(audio_file_path, asr_model_citrinet)
    del asr_model_citrinet  # Free up memory
    torch.cuda.empty_cache()

    # Preprocess transcriptions
    transcriptions_pp = {key: preprocess_text(value) for key, value in
    ↪  transcriptions.items()}

    # Save transcriptions to separate CSV files
    csv_filenames = {
        'Google': 'transcription_google_sr.csv',
        'CTC Large': 'transcription_ctc_large_sr.csv',
        'Citrinet': 'transcription_citrinet_sr.csv'
    }

    for key, value in transcriptions_pp.items():
        save_transcription_to_csv([value], csv_filenames[key])
        # Calculate metrics for each transcription
```

```
        wer_score, cer_score = calculate_metrics(reference_transcription, value)
        overall_metrics[key]['wer'].append(wer_score)
        overall_metrics[key]['cer'].append(cer_score)
        overall_metrics[key]['time'].append(times[key])
        print(f"{key} - {audio_file} - WER: {wer_score:.4f}, CER: {cer_score:.4f},
        ↪  Time: {times[key]:.4f} seconds")

# Now we process Whisper models separately to avoid GPU memory issues

for audio_file in os.listdir(audio_folder_path):
    audio_file_path = os.path.join(audio_folder_path, audio_file)

    # Load Whisper models and perform recognition
    whisper_model_large = whisper.load_model("large")
    transcriptions['Whisper Large'], times['Whisper Large'] =
    ↪  speech_recognition_whisper(audio_file_path, whisper_model_large, "Large")
    del whisper_model_large  # Free up memory
    torch.cuda.empty_cache()

    whisper_model_medium = whisper.load_model("medium")
    transcriptions['Whisper Medium'], times['Whisper Medium'] =
    ↪  speech_recognition_whisper(audio_file_path, whisper_model_medium, "Medium")
    del whisper_model_medium  # Free up memory
    torch.cuda.empty_cache()

    # Preprocess and save transcriptions
    transcriptions_pp = {key: preprocess_text(value) for key, value in
    ↪  transcriptions.items() if 'Whisper' in key}
    csv_filenames_whisper = {
        'Whisper Large': 'transcription_whisper_large_sr.csv',
        'Whisper Medium': 'transcription_whisper_medium_sr.csv'
    }

    for key, value in transcriptions_pp.items():
        save_transcription_to_csv([value], csv_filenames_whisper[key])
        # Calculate metrics for each transcription
        wer_score, cer_score = calculate_metrics(reference_transcription, value)
        overall_metrics[key]['wer'].append(wer_score)
        overall_metrics[key]['cer'].append(cer_score)
        overall_metrics[key]['time'].append(times[key])
        print(f"{key} - {audio_file} - WER: {wer_score:.4f}, CER: {cer_score:.4f},
        ↪  Time: {times[key]:.4f} seconds")

# Save individual metrics to CSV
save_metrics_to_csv(overall_metrics, 'metrics_individual.csv')

# Calculate and print overall metrics for each algorithm
overall_results = []
for key in overall_metrics.keys():
    overall_wer = np.mean(overall_metrics[key]['wer'])
    overall_cer = np.mean(overall_metrics[key]['cer'])
    overall_time = np.mean(overall_metrics[key]['time'])
    overall_results.append([key, overall_wer, overall_cer, overall_time])
    print(f"Overall {key} - WER: {overall_wer:.4f}, CER: {overall_cer:.4f}, Avg Time:
    ↪  {overall_time:.4f} seconds")

# Save overall metrics to CSV
with open('model-results.csv', 'w', newline='', encoding='utf-8') as csvfile:
    writer = csv.writer(csvfile)
    writer.writerow(['Model', 'WER', 'CER', 'Avg Time (seconds)'])
    writer.writerows(overall_results)
```

3.4 Further testing

This code is used for more testing. Splitting the dataset into various types of data and performing a further analysis on them.

```
# Importing dependencies
import jiwer
import csv

# Letters hypotheses
hypotheses_google_letters = [ ... ]
hypotheses_whisper_medium_letters = [ ... ]
hypotheses_whisper_large = [ ... ]
hypotheses_citrinet = [ ... , ]
hypotheses_ctc = [ ... ]

# Words hypotheses
hypotheses_google_words = [ ...]
hypotheses_whisper_mid_words = [ ... ]
hypotheses_whisper_large_words = [ ... ]
hypotheses_citrinet_words = [ ... ]
hypotheses_ctc_words = [ ... ]

# Sentence hypotheses
hypotheses_google_sentences = [ ... ]
hypotheses_whisper_mid_sentences = [ ... ]
hypotheses_whisper_large_sentences = [ ... ]
hypotheses_citrinet_sentences = [ ... ]
hypotheses_ctc_sentences = [ ... ]

# Text hypotheses
hypotheses_google_text = [ ... ]
hypotheses_whisper_mid_text = [ ... }
hypotheses_whisper_large_text = [ ... ]
hypotheses_citrinet_text = [ ... ]
hypotheses_ctc_text = [ ... ]

def compute_average_wer_cer(reference, hypothesis_lists):
    """
    Compute the average Word Error Rate (WER) and Character Error Rate (CER) for a single
    ↪  reference
    and a list of hypothesis lists.

    Parameters:
    - reference (str): The reference text.
    - hypothesis_lists (list of lists of str): List where each sublist contains
    ↪  hypotheses for the reference.

    Returns:
    - averages (list of tuples): A list of tuples, each containing the average CER and
    ↪  WER for each model.
    """
    averages = []

    # Iterate over each list of hypotheses
    for hypotheses in hypothesis_lists:
        cer_list = []  # List to store CER values for current hypothesis list
        wer_list = []  # List to store WER values for current hypothesis list

        # Compute CER and WER for each hypothesis
        for hypothesis in hypotheses:
            cer = jiwer.cer(reference, hypothesis)  # Calculate CER
            wer = jiwer.wer(reference, hypothesis)  # Calculate WER
            cer_list.append(cer)  # Append CER value to list
            wer_list.append(wer)  # Append WER value to list
```

```
        # Calculate average CER and WER for the current list of hypotheses
        average_cer = sum(cer_list) / len(cer_list) if cer_list else 0  # Compute
        ↪  average CER
        average_wer = sum(wer_list) / len(wer_list) if wer_list else 0  # Compute
        ↪  average WER

        averages.append((average_cer, average_wer))  # Append the average CER and WER as
        ↪  a tuple

    return averages

# Lists of hypotheses for different categories
hypothesis_lists_letters = [hypotheses_google_letters,
↪  hypotheses_whisper_medium_letters, hypotheses_whisper_large_letters,
↪  hypotheses_citrinet_letters, hypotheses_ctc_letters]
hypothesis_lists_words = [hypotheses_google_words, hypotheses_whisper_mid_words,
↪  hypotheses_whisper_large_words, hypotheses_citrinet_words, hypotheses_ctc_words]
hypothesis_lists_sentences = [hypotheses_google_sentences,
↪  hypotheses_whisper_mid_sentences, hypotheses_whisper_large_sentences,
↪  hypotheses_citrinet_sentences, hypotheses_ctc_sentences]
hypothesis_lists_texts = [hypotheses_google_text, hypotheses_whisper_mid_text,
↪  hypotheses_whisper_large_text, hypotheses_citrinet_text, hypotheses_ctc_text]

# Compute average CER and WER for each category of hypotheses
averages_letters = compute_average_wer_cer(reference_1, hypothesis_lists_letters)
average_words = compute_average_wer_cer(reference_2, hypothesis_lists_words)
average_sentences = compute_average_wer_cer(reference_3, hypothesis_lists_sentences)
average_text = compute_average_wer_cer(reference_4, hypothesis_lists_texts)

model_names = ['Google', 'Whisper Medium', 'Whisper Large', 'Citrinet', 'CTC Large']

# Open a CSV file for writing
with open('model_results.csv', mode='w', newline='') as file:
    writer = csv.writer(file)

    # Write the headers
    writer.writerow(['Model', 'Data Type', 'Average CER', 'Average WER'])

    # Write results for letters
    for idx, (avg_cer, avg_wer) in enumerate(averages_letters):
        print(f"{model_names[idx]} - Average Character Error Rate (CER) for characters:
        ↪  {avg_cer}")
        writer.writerow([model_names[idx], 'Letters', avg_cer, 'N/A'])

    # Write results for words
    for idx, (avg_cer, avg_wer) in enumerate(average_words):
        print(f"{model_names[idx]} - Average Word Error Rate (WER) for words:
        ↪  {avg_wer}")
        print(f"{model_names[idx]} - Average Character Error Rate (CER) for words:
        ↪  {avg_cer}")
        writer.writerow([model_names[idx], 'Words', avg_cer, avg_wer])

    # Write results for sentences
    for idx, (avg_cer, avg_wer) in enumerate(average_sentences):
        print(f"{model_names[idx]} - Average Word Error Rate (WER) for sentences:
        ↪  {avg_wer}")
        print(f"{model_names[idx]} - Average Character Error Rate (CER) for sentences:
        ↪  {avg_cer}")
        writer.writerow([model_names[idx], 'Sentences', avg_cer, avg_wer])

    # Write results for texts
    for idx, (avg_cer, avg_wer) in enumerate(average_text):
        print(f"{model_names[idx]} - Average Word Error Rate (WER) for texts:
        ↪  {avg_wer}")
        print(f"{model_names[idx]} - Average Character Error Rate (CER) for texts:
        ↪  {avg_cer}")
```

```
        writer.writerow([model_names[idx], 'Texts', avg_cer, avg_wer])

print("Results have been written to 'model_results.csv'")
```

Appendix B

Figures and Tables

In this part, all Figures and Tables from the analysis part are included.

1 Hand gesture recognition

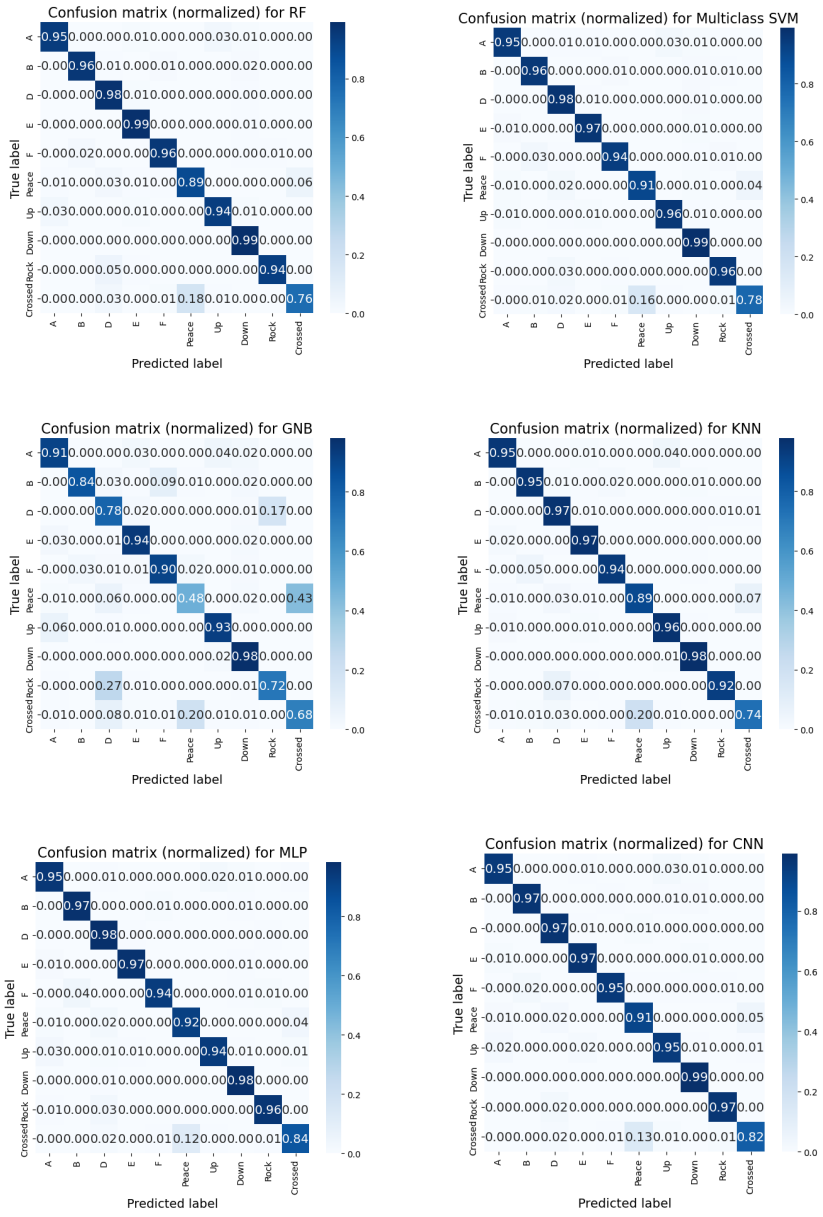


Figure 1: Confusion matrices for all tested models.

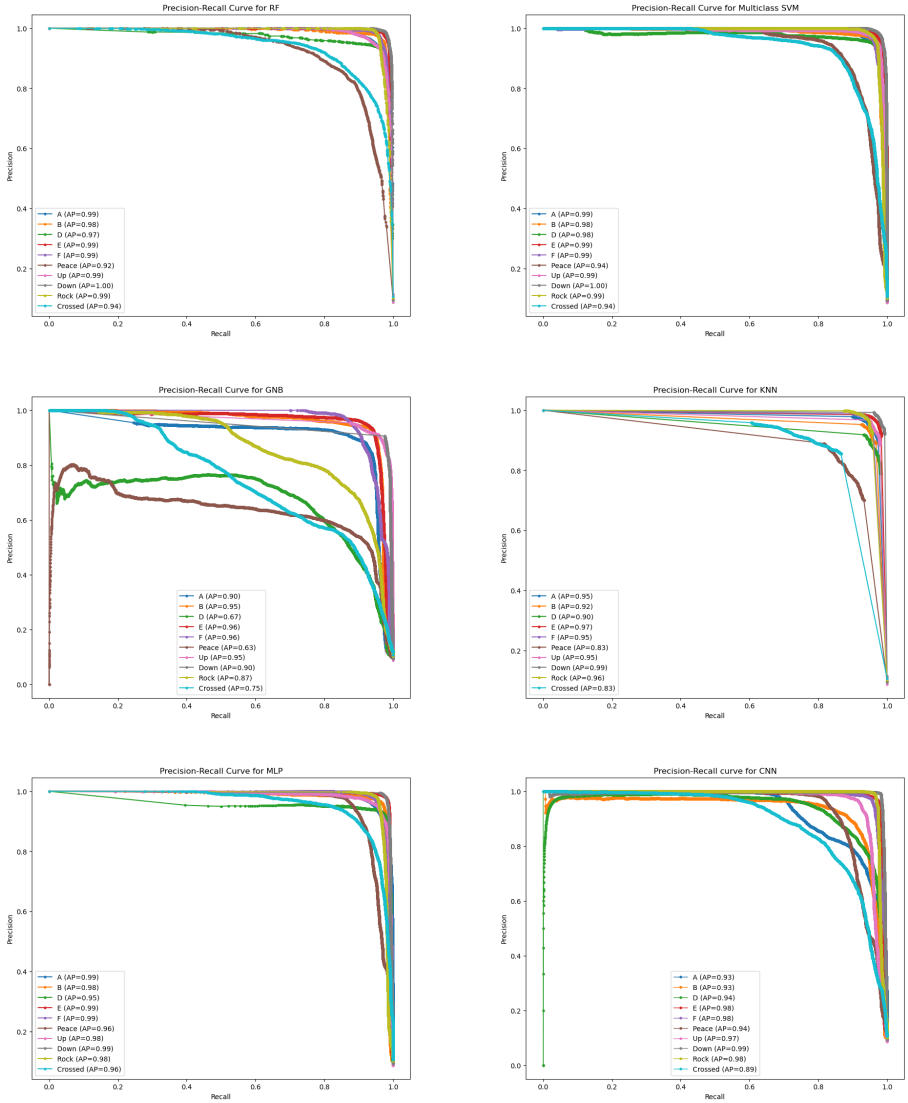


Figure 2: Precision-recall curves for all the tested models.

2 Emotion recognition

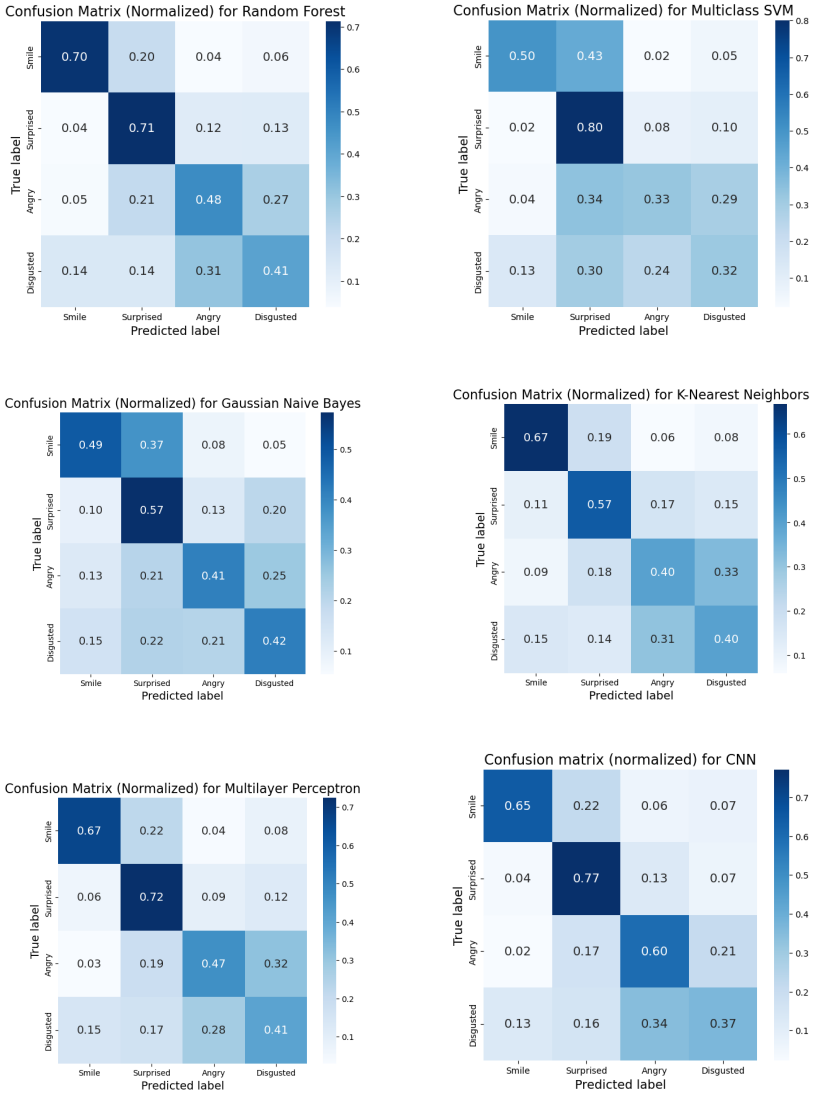


Figure 3: Confusion matrices for all tested models.

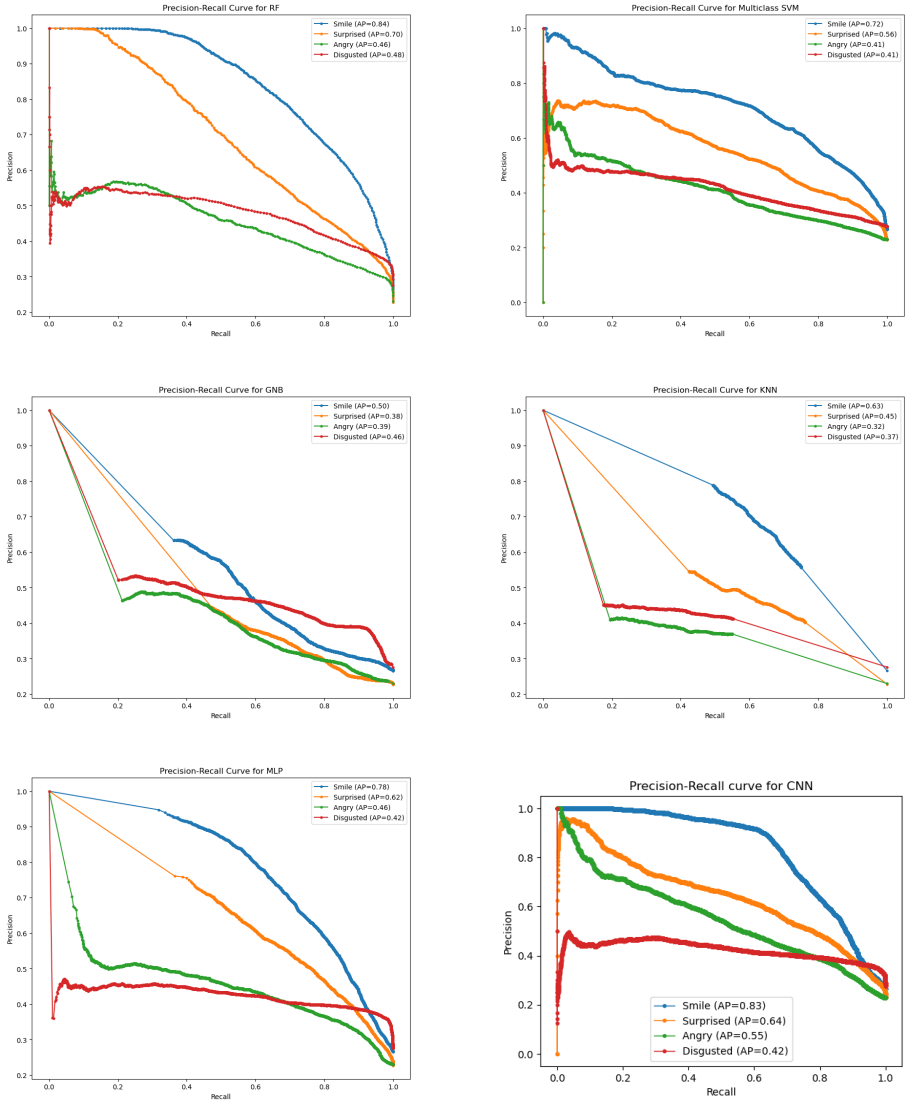


Figure 4: Precision-recall curves for all the tested models.

3 Speech recognition

Model	WER	CER	Avg Time (seconds)
Google	0.23739617539115324	0.10578661844484628	42.01206757945399
CTC Large	0.10817075526366621	0.03995800034999709	1.1310200383586269
Citrinet	0.10894340351554956	0.04366213614886543	0.32852215920725175
Whisper Large	0.3169789453351362	0.24963541970483583	50.50727157438955
Whisper Medium	0.20204751786749084	0.1271072741060491	16.363750703873173

Table 1: Overall metrics for the different speech recogntion models.

Model	Data Type	Average CER	Average WER
Google	Letters	0.47526881720430114	N/A
Whisper Medium	Letters	0.03655913978494624	N/A
Whisper Large	Letters	0.004301075268817204	N/A
Citrinet	Letters	0.34193548387096767	N/A
CTC Large	Letters	0.378494623655914	N/A
Google	Words	0.08349146110056925	0.5709677419354839
Whisper Medium	Words	0.13357843137254902	0.776666666666667
Whisper Large	Words	0.01968690702087286	0.09032258064516133
Citrinet	Words	0.07732447817836813	0.367741935483871
CTC Large	Words	0.06095825426944971	0.29677419354838713
Google	Sentences	0.12317232747340275	0.18468628146047503
Whisper Medium	Sentences	0.1861845972957084	0.24505494505494502
Whisper Large	Sentences	0.2531148660180918	0.31974477135767454
Citrinet	Sentences	0.03976787847755589	0.08365827720666429
CTC Large	Sentences	0.03902827558741537	0.08472172988302025
Google	Texts	0.0682027649769585	0.16240266963292543
Whisper Medium	Texts	0.09317134478424802	0.14516129032258068
Whisper Large	Texts	0.3281943862589024	0.3859844271412681
Citrinet	Texts	0.015500628403854213	0.058398220244716345
CTC Large	Texts	0.01365731043150398	0.04671857619577308

Table 2: Overall metrics for the different speech recogntion models.

Appendix C

Hand Keypoint Preprocessing

Appendix C contains the implementation of the preprocessing process described in Subsection 3.1.3.

```python
import os
import copy
import itertools
import numpy as np
import mediapipe as mp
import cv2
import csv

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

# Instantiate the Hands object with specific parameters
hands = mp_hands.Hands(static_image_mode=True, min_detection_confidence=0.3)


# Commenting out depending on wether training or test set is needed.
DATA_DIR = 'D:\\BA_picture_data\\hand_picture_test'
CSV_PATH = 'model/hand_keypoint_classifier/hand_keypoint_testset.csv'

# DATA_DIR = 'D:\\BA_picture_data\\hand_picture_data'
# CSV_PATH = 'model/hand_keypoint_classifier/hand_keypoint_trainingset.csv'



def calc_landmark_list(image, landmarks):
    """
    Convert landmarks to image coordinates.

    Parameters:
    - image: The input image to get dimensions.
    - landmarks: MediaPipe landmarks object.

    Returns:
    - landmark_point: List of landmark coordinates in image space.
    """
    image_width, image_height = image.shape[1], image.shape[0]
    landmark_point = []

    # Convert normalized landmarks to pixel coordinates
    for _, landmark in enumerate(landmarks.landmark):
        landmark_x = min(int(landmark.x * image_width), image_width - 1)
        landmark_y = min(int(landmark.y * image_height), image_height - 1)
        landmark_point.append([landmark_x, landmark_y])
    return landmark_point

def pre_process_landmark(landmark_list):
    """
    Pre-process landmark points: translate to origin and normalize.

    Parameters:
    - landmark_list: List of landmark coordinates in image space.

    Returns:
    - temp_landmark_list: Flattened and normalized landmark coordinates.
    """
    temp_landmark_list = copy.deepcopy(landmark_list)

    base_x, base_y = 0, 0
    for index, landmark_point in enumerate(temp_landmark_list):
        if index == 0:
            base_x, base_y = landmark_point[0], landmark_point[1]
        temp_landmark_list[index][0] = temp_landmark_list[index][0] - base_x
        temp_landmark_list[index][1] = temp_landmark_list[index][1] - base_y
```

```
    # Flatten the list of landmarks
    temp_landmark_list = list(itertools.chain.from_iterable(temp_landmark_list))

    # Normalize the landmark coordinates
    max_value = max(list(map(abs, temp_landmark_list)))

    def normalize_(n):
        return n / max_value

    temp_landmark_list = list(map(normalize_, temp_landmark_list))

    return temp_landmark_list

# Create CSV file if it does not exist
if not os.path.exists(CSV_PATH):
    with open(CSV_PATH, 'w', newline='') as f:
        writer = csv.writer(f)
        writer.writerow(['Label', *range(42)])  # Assuming 21 landmarks with X and Y
        ↪  coordinates
```

The labeling of images is determined by the folder structure. Each folder corresponds to a different label, which is used when processing the images.

```
# Process images in the specified directory
for dir_ in os.listdir(DATA_DIR):
    for img_path in os.listdir(os.path.join(DATA_DIR, dir_)):
        # Read the image
        img = cv2.imread(os.path.join(DATA_DIR, dir_, img_path))
        img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

        # Process image to find hand landmarks
        results = hands.process(img_rgb)

        if results.multi_hand_landmarks:
            for hand_landmarks in results.multi_hand_landmarks:
                # Extract landmarks as a list of points
                landmark_list = calc_landmark_list(img, hand_landmarks)
                # Pre-process landmark list
                preprocessed_landmarks = pre_process_landmark(landmark_list)

                # Save the image with landmarks drawn
                output_path = os.path.join(DATA_DIR, dir_, img_path.split('.')[0] +
                ↪  '_landmarks.jpg')
                # cv2.imwrite(output_path, white_img) # This line seems to be commented
                ↪  out

                # Append the landmarks to the CSV file
                with open(CSV_PATH, 'a', newline='') as f:
                    writer = csv.writer(f)
                    writer.writerow([img_path, dir_, *preprocessed_landmarks])

        else:
            # Print a message if no landmarks are detected
            print(f"No hand landmarks detected in image: {os.path.join(dir_,
            ↪  img_path)}")
```

Appendix D

Face Keypoint Preprocessing

Appendix D contains the implementation details of the preprocessing process from Subsection 3.1.4.

```python
import os
import copy
import itertools
import mediapipe as mp
import cv2
import csv

# Initialize MediaPipe Face Mesh and drawing utilities
mp_face_mesh = mp.solutions.face_mesh
mp_drawing = mp.solutions.drawing_utils
mp_drawing_styles = mp.solutions.drawing_styles

# Create a FaceMesh object for detecting face landmarks
face_mesh = mp_face_mesh.FaceMesh(static_image_mode=True, min_detection_confidence=0.3)

# Commenting out depending on wether training or test set is needed.
DATA_DIR = 'D:\\BA_face_picture_data\\face_picture_test'
CSV_PATH = 'D:\\BA\\face_keypoint_testset3c.csv'

# DATA_DIR = 'D:\\BA_face_picture_data\\face_picture_data'
# CSV_PATH = 'D:\\BA\\face_keypoint_trainingset3c.csv'

# Define landmark sets
mouth_landmarks = set([
    0, 11, 12, 13, 14, 15, 16, 17, 37, 72, 38, 82, 87, 86, 85, 84, 39, 73, 41, 81, 178,
    179, 180, 181, 40, 74, 42, 80, 88, 89, 90, 91, 185, 184, 183, 191, 95, 96, 77, 146,
    61, 76, 62, 267, 302, 268, 312, 317, 316, 315, 314, 269, 303, 271, 311, 402, 403,
    404, 405, 270, 304, 272, 310, 318, 319, 320, 321, 409, 408, 407, 415, 324, 325,
    307, 375, 308, 292, 306, 291
])

left_eye_landmarks = set([
    33, 246, 161, 160, 159, 158, 157, 173, 133, 155, 154, 153, 145, 144, 163, 7,
    130, 247, 30, 29, 27, 28, 56, 190, 243, 112, 26, 22, 23, 24, 110, 25,
    226, 113, 225, 224, 223, 222, 221, 189, 124, 46, 53, 52, 65, 156, 70, 63,
    105, 66, 107, 55, 193, 245, 244, 233, 232, 231, 230, 229, 228, 31, 35
])

right_eye_landmarks = set([
    136, 296, 334, 293, 300, 383, 265, 261, 448, 449, 450, 451, 452, 453, 464, 465,
    417, 285, 295, 282, 283, 276, 353, 362, 398, 384, 385, 386, 387, 388, 466, 263,
    249, 390, 373, 374, 380, 381, 382, 413, 441, 442, 443, 444, 445, 342, 446,
    463, 341, 256, 252, 253, 254, 339, 255, 359, 467, 260, 259, 257, 258, 286, 414
])

def calc_landmark_list(image, landmarks):
    """
    Converts the face mesh landmarks to image coordinates.

    Args:
        image: The image from which landmarks are extracted.
        landmarks: The landmarks detected by MediaPipe FaceMesh.

    Returns:
        A list of landmark coordinates as [x, y] in image space.
    """
    image_width, image_height = image.shape[1], image.shape[0]
    landmark_point = []

    for landmark in landmarks.landmark:
        landmark_x = min(int(landmark.x * image_width), image_width - 1)
        landmark_y = min(int(landmark.y * image_height), image_height - 1)
        landmark_point.append([landmark_x, landmark_y])
    return landmark_point
```

```python
def get_reference_points(landmark_list):
    """
    Get reference points for mouth, left eye, right eye, and nose.

    Args:
        landmark_list: List of landmark coordinates.

    Returns:
        A tuple containing reference points for mouth, left eye, right eye, and nose.
    """
    left_eye_ref = landmark_list[145]
    right_eye_ref = landmark_list[374]
    mouth_ref = landmark_list[14]
    nose_ref = landmark_list[4]

    return left_eye_ref, right_eye_ref, mouth_ref, nose_ref

def pre_process_landmark(landmark_list):
    """
    Process and normalize landmarks based on reference points.

    Args:
        landmark_list: List of landmark coordinates.

    Returns:
        A list of normalized landmark coordinates.
    """
    # Create a deep copy of the original landmark list
    temp_landmark_list = copy.deepcopy(landmark_list)

    # Get reference points for normalization
    reference_points = get_reference_points(temp_landmark_list)
    left_eye_ref, right_eye_ref, mouth_ref, nose_ref = reference_points

    # Initialize lists for different landmark groups
    mouth_coords = []
    left_eye_coords = []
    right_eye_coords = []
    other_coords = []

    # Categorize landmarks based on their type and reference points
    for index, landmark_point in enumerate(temp_landmark_list):
        x, y = landmark_point

        if index in mouth_landmarks:
            ref_x, ref_y = mouth_ref
            mouth_coords.append((x - ref_x, y - ref_y))
        elif index in left_eye_landmarks:
            ref_x, ref_y = left_eye_ref
            left_eye_coords.append((x - ref_x, y - ref_y))
        elif index in right_eye_landmarks:
            ref_x, ref_y = right_eye_ref
            right_eye_coords.append((x - ref_x, y - ref_y))
        else:
            ref_x, ref_y = nose_ref
            other_coords.append((x - ref_x, y - ref_y))

    # Flatten lists for normalization
    flattened_mouth = list(itertools.chain.from_iterable(mouth_coords))
    flattened_left_eye = list(itertools.chain.from_iterable(left_eye_coords))
    flattened_right_eye = list(itertools.chain.from_iterable(right_eye_coords))
    flattened_other = list(itertools.chain.from_iterable(other_coords))

    # Find the maximum absolute value for normalization
    max_mouth = max(list(map(abs, flattened_mouth)))
    max_left_eye = max(list(map(abs, flattened_left_eye)))
```

```
    max_right_eye = max(list(map(abs, flattened_right_eye)))
    max_other = max(list(map(abs, flattened_other)))

    # Normalize coordinates by dividing by the maximum value
    def normalize_group(coords, max_value):
        return [n / max_value for n in coords]

    normalized_mouth = normalize_group(flattened_mouth, max_mouth)
    normalized_left_eye = normalize_group(flattened_left_eye, max_left_eye)
    normalized_right_eye = normalize_group(flattened_right_eye, max_right_eye)
    normalized_other = normalize_group(flattened_other, max_other)

    # Combine all normalized landmarks into a single list
    normalized_landmarks = normalized_mouth + normalized_left_eye +
    ↪  normalized_right_eye + normalized_other

    return normalized_landmarks

# Create CSV file with headers if it does not already exist
if not os.path.exists(CSV_PATH):
    with open(CSV_PATH, 'w', newline='') as f:
        writer = csv.writer(f)
        writer.writerow(['Label', *range(936)])  # 468 landmarks with X and Y
        ↪  coordinates
```

```
# Process each image in the directory
for dir_ in os.listdir(DATA_DIR):
    for img_path in os.listdir(os.path.join(DATA_DIR, dir_)):
        # Read and preprocess the image
        img = cv2.imread(os.path.join(DATA_DIR, dir_, img_path))
        img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

        # Process the image to extract face landmarks
        results = face_mesh.process(img_rgb)

        if results.multi_face_landmarks:
            for face_landmarks in results.multi_face_landmarks:
                # Extract and preprocess landmark points
                landmark_list = calc_landmark_list(img, face_landmarks)
                preprocessed_landmarks = pre_process_landmark(landmark_list)

                # Append processed landmarks to CSV file
                with open(CSV_PATH, 'a', newline='') as f:
                    writer = csv.writer(f)
                    writer.writerow([img_path, dir_, *preprocessed_landmarks])

        else:
            # Print a message if no face landmarks are detected in the image
            print(f"No face landmarks detected in image: {os.path.join(dir_,
            ↪  img_path)}")
```

Bibliography

[1] Qiuhong Ke, Jun Liu, Mohammed Bennamoun, Senjian An, Ferdous Sohel, and Farid Boussaid. Chapter 5 - computer vision for human–machine interaction. In Marco Leo and Giovanni Maria Farinella, editors, *Computer Vision for Assistive Healthcare*, Computer Vision and Pattern Recognition, pages 127–145. Academic Press, 2018.

[2] Alejandro Jaimes and Nicu Sebe. Multimodal human-computer interaction: A survey. *Comput. Vis. Image Underst.*, 108(1-2):116–134, 2007.

[3] Maja Pantic and Léon J. M. Rothkrantz. Toward an affect-sensitive multimodal human-computer interaction. *Proc. IEEE*, 91(9):1370–1390, 2003.

[4] Nicolas Pugeault and Richard Bowden. Spelling it out: Real-time ASL fingerspelling recognition. In *IEEE International Conference on Computer Vision Workshops, ICCV 2011 Workshops, Barcelona, Spain, November 6-13, 2011*, pages 1114–1119. IEEE Computer Society, 2011.

[5] Pramod Kumar Pisharady and Martin Saerbeck. Recent methods and databases in vision-based hand gesture recognition: A review. *Comput. Vis. Image Underst.*, 141:152–165, 2015.

[6] Sébastien Marcel. Hand posture recognition in a body-face centered space. In Michael E. Atwood, editor, *CHI '99 Extended Abstracts on Human Factors in Computing Systems, CHI Extended Abstracts '99, Pittsburgh, Pennsylvania, USA, May 15-20, 1999*, pages 302–303. ACM, 1999.

[7] ByoungChul Ko. A brief review of facial emotion recognition based on visual information. *Sensors*, 18(2):401, 2018.

[8] Patrick Lucey, Jeffrey F. Cohn, Takeo Kanade, Jason M. Saragih, Zara Ambadar, and Iain A. Matthews. The extended cohn-kanade dataset (CK+): A complete dataset for action unit and emotion-specified expression. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR Workshops*

2010, San Francisco, CA, USA, 13-18 June, 2010, pages 94–101. IEEE Computer Society, 2010.

[9] Athinodoros S. Georghiades, Peter N. Belhumeur, and David J. Kriegman. From few to many: Illumination cone models for face recognition under variable lighting and pose. *IEEE Trans. Pattern Anal. Mach. Intell.*, 23(6):643–660, 2001.

[10] Johannes Wirth and René Peinl. ASR in german: A detailed error analysis. *CoRR*, abs/2204.05617, 2022.

[11] Lingchen Chen, Feng Wang, Hui Deng, and Kaifan Ji. A survey on hand gesture recognition. In *2013 International Conference on Computer Sciences and Applications*, pages 313–316, 2013.

[12] Paul Ekman and Wallace V. Friesen. *Facial Action Coding System: A Technique for the Measurement of Facial Movement*. Consulting Psychologists Press, Palo Alto, CA, 1978.

[13] Pawel Tarnowski, Marcin Kolodziej, Andrzej Majkowski, and Remigiusz J. Rak. Emotion recognition using facial expressions. In Petros Koumoutsakos, Michael Lees, Valeria V. Krzhizhanovskaya, Jack J. Dongarra, and Peter M. A. Sloot, editors, *International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland*, volume 108 of *Procedia Computer Science*, pages 1175–1184. Elsevier, 2017.

[14] Paul Ekman. An argument for basic emotions. *Cognition and Emotion*, 6(3-4):169–200, 1992.

[15] D. Eilert and B. Volke. *Mimikresonanz: Gefühle sehen. Menschen verstehen.* Reihe Fachbuch Kommunikation und Mimik. Junfermannsche Verlagsbuchhandlung, 2013.

[16] Santosh K Gaikwad, Bharti W Gawali, and Pravin Yannawar. A review on speech recognition technique. *International Journal of Computer Applications*, 10(3):16–24, 2010.

[17] Niels Schiller. *The Phonetic Variation of German /r/*, pages 261–287. G. Olms, 1999.

[18] Volker Kuehnel, Birger Kollmeier, and Kirsten Wagener. Entwicklung und evaluation eines satztests für die deutsche sprache i: Design des oldenburger satztests. *Zeitschrift für Audiologie*, 38:4–15, 09 1999.

[19] Fan Zhang, Valentin Bazarevsky, Andrey Vakunov, Andrei Tkachenka, George Sung, Chuo-Ling Chang, and Matthias Grundmann. Mediapipe hands: On-device real-time hand tracking. *CoRR*, abs/2006.10214, 2020.

[20] Yury Kartynnik, Artsiom Ablavatski, Ivan Grishchenko, and Matthias Grundmann. Real-time facial surface geometry from monocular video on mobile gpus. *CoRR*, abs/1907.06724, 2019.

[21] Google AI. Hand landmarker: Overview, 2024. Accessed: 06-08-2024.

[22] Google AI. Mediapipe face geometry canonical face model uv visualization. [Online]. Available: `https://github.com/google-ai-edge/mediapipe/blob/master/mediapipe/modules/face_geometry/data/canonical_face_model_uv_visualization.png`, 2023. Accessed: 06-08-2024.

[23] Shigeki Takahashi. hand gesture recognition using mediapipe. [Online]. Available: `https://github.com/Kazuhito00/hand-gesture-recognition-using-mediapipe`, 2020. Accessed: 28.07.2024.

[24] Iqbal H. Sarker. Machine learning: Algorithms, real-world applications and research directions. *SN Comput. Sci.*, 2(3):160, 2021.

[25] Philipp Probst, Marvin N. Wright, and Anne-Laure Boulesteix. Hyperparameters and tuning strategies for random forest. *WIREs Data Mining Knowl. Discov.*, 9(3), 2019.

[26] Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin. A practical guide to support vector classification. 2003.

[27] Harry Zhang. The optimality of naive bayes. In Valerie Barr and Zdravko Markov, editors, *Proceedings of the Seventeenth International Florida Artificial Intelligence Research Society Conference, Miami Beach, Florida, USA*, pages 562–567. AAAI Press, 2004.

[28] Uday Shankar Shanthamallu, Andreas Spanias, Cihan Tepedelenlioglu, and Mike Stanley. A brief survey of machine learning methods and their sensor and iot applications. In Nikolaos G. Bourbakis, George A. Tsihrintzis, and Maria Virvou, editors, *8th International Conference on Information, Intelligence, Systems & Applications, IISA 2017, Larnaca, Cyprus, August 27-30, 2017*, pages 1–8. IEEE, 2017.

[29] P. M. Atkinson and A. R. L. Tatnall. Introduction neural networks in remote sensing. *International Journal of Remote Sensing*, 18(4):699–709, 1997.

[30] Haitham Afan, Ahmedbahaaaldin Ibrahem Ahmed Osman, Yusuf Essam, Al-Mahfoodh Ali Najah Ahmed, Yuk Huang, Ozgur Kisi, Mohsen Sherif, Ahmed Sefelnasr, Kwok Chau, and Ahmed El-Shafie. Modeling the fluctuations of groundwater level by employing ensemble deep learning techniques. *Engineering Applications of Computational Fluid Mechanics*, 15:1420–1439, 09 2021.

[31] Yann LeCun, Yoshua Bengio, and Geoffrey E. Hinton. Deep learning. *Nat.*, 521(7553):436–444, 2015.

[32] Alex Shenfield and Martin Howarth. A novel deep learning model for the detection and identification of rolling element-bearing faults. *Sensors*, 20(18):5112, 2020.

[33] Yousef O. Sharrab, Sana'a Al-shboul, Mohammad A. Alsmirat, Alá F. Khalifeh, Zyad Dwekat, Izzat Alsmadi, and Ahmad Al-Khasawneh. Performance comparison of several deep learning-based object detection algorithms utilizing thermal images. In *Second International Conference on Intelligent Data Science Technologies and Applications, IDSTA 2021, Tartu, Estonia, November 15-17, 2021*, pages 16–22. IEEE, 2021.

[34] Mohammad Hossin and Md Nasir Sulaiman. A review on evaluation metrics for data classification evaluations. *International journal of data mining & knowledge management process*, 5(2):1, 2015.

[35] PyPI. Speechrecognition python package. [Online]. Available: `https://pypi.org/project/SpeechRecognition/`, 2024. Accessed: 13-08-2024.

[36] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 28492–28518. PMLR, 2023.

[37] Somshubra Majumdar, Jagadeesh Balam, Oleksii Hrinchuk, Vitaly Lavrukhin, Vahid Noroozi, and Boris Ginsburg. Citrinet: Closing the gap between non-autoregressive and autoregressive end-to-end models for automatic speech recognition, 2021.

[38] NVIDIA. Nvidia citrinet german model. [Online]. Available: https://catalog.ngc.nvidia.com/orgs/nvidia/teams/nemo/models/stt_de_citrinet_1024, 2024. Accessed: 14-08-2024.

[39] Anmol Gulati, James Qin, Chung-Cheng Chiu, Niki Parmar, Yu Zhang, Jiahui Yu, Wei Han, Shibo Wang, Zhengdong Zhang, Yonghui Wu, and Ruoming Pang. Conformer: Convolution-augmented transformer for speech recognition. In Helen Meng, Bo Xu, and Thomas Fang Zheng, editors, *21st Annual Conference of the International Speech Communication Association, Interspeech 2020, Virtual Event, Shanghai, China, October 25-29, 2020*, pages 5036–5040. ISCA, 2020.

[40] NVIDIA. Nvidia nemo conformer ctc large model german model. [Online]. Available: https://catalog.ngc.nvidia.com/orgs/nvidia/teams/nemo/models/stt_de_conformer_ctc_large, 2024. Accessed: 13-08-2024.

[41] Iain Mccowan, Darren Moore, John Dines, Daniel Gatica-Perez, Mike Flynn, Pierre Wellner, and Herve Bourlard. On the use of information retrieval measures for speech recognition evaluation. 2004.

[42] Scikit learn developers. Gaussiannb — scikit-learn 1.0.2 documentation. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html, 2024. Accessed: 02-09-2024.

[43] Li Yang and Abdallah Shami. On hyperparameter optimization of machine learning algorithms: Theory and practice. *Neurocomputing*, 415:295–316, 2020.

[44] Wenbo Zhu, Weichang Yeh, Jianwen Chen, Dafeng Chen, Aiyuan Li, and Yangyang Lin. Evolutionary convolutional neural networks using ABC. In *Proceedings of the 2019 11th International Conference on Machine Learning and Computing, ICMLC '19, Zhuhai, China, February 22-24, 2019*, pages 156–162. ACM, 2019.