UNDERGRADUATE RESEARCH OPPORTUNITIES PROGRAMME (UROP)

Final Project Report

Project EDIC1: Development of Edge-AI Virtual Reality System

Seohyun (Sam) Park, University of Pennsylvania

Engineering Design and Innovation Centre, College Design and Engineering

National University of Singapore

Special Term AY 2025 (May 20th - July 17th)

Table of Contents

1. Abstract

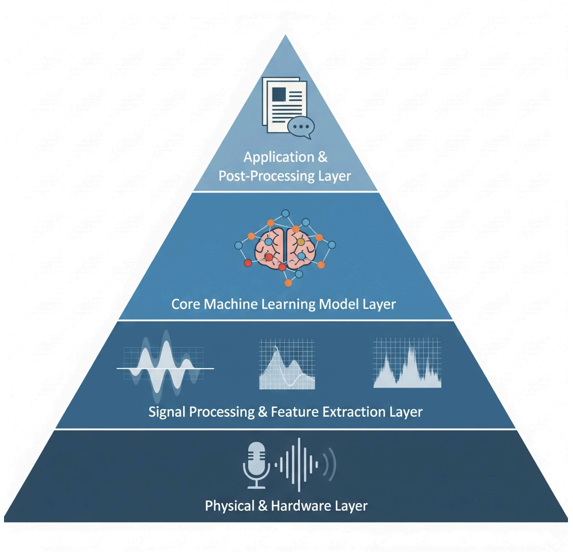
This research focuses on the enhancement of voice recognition software within the Intelligent Virtual Eye Screening Algorithm (iVESA) project, a collaborative industrial initiative between the Singapore Eye Research Institute, Singapore National Eye Centre, IOInspire PTE LTD, and the National University of Singapore (NUS). Conducted during the Summer Engineering Research Internship for US Students (SERIUS) 2025, the project aims to improve the accuracy, efficiency, and user interaction of the iVESA system through advanced voice command and data input functionalities. This paper details the development of a sophisticated voice recognition model designed to streamline the eye screening process for both clinicians and patients, specifically designed to understand Singaporean children saying letters in the English alphabet with a goal of reaching 99% accuracy. The resulting improvements to the voice recognition software demonstrate a significant step forward in creating more intuitive and accessible diagnostic tools.

2. Preface and Acknowledgements

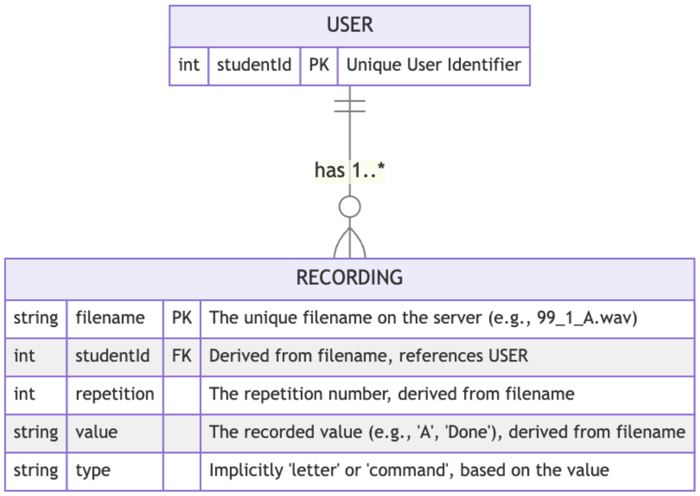
I would like to express my sincere gratitude to Dr. Tang Kok Zuea and our business client Mr. Harry Lim for providing me with the opportunity to participate in this research project in National University of Singapore (NUS). I would also like to extend special thanks to my teammate Jigao (Thomas) Wang and fellow NUS student collaborator Joshua Siew for their consistent support throughout the development process. They worked separately on the hardware and the user interface, respectively, for the making of this product. Lastly, I feel grateful to the dedicated junior students who assisted with accessory design components and their contribution to the smooth field testing at the TreeHouse daycare. The work, conducted by me, a university student from Canada, a university student from NUS, and 6 other tertiary students from Singapore, underscores the importance of interdisciplinary and international cooperation in advancing medical technology.

3. Step One: Collecting the Voice Files

The very first step in this entire process is collecting the voice files. This diagram models the entire process by which we are creating our voice recognition model:


Application &
Post-Processing Layer
Core Machine Learning Model Layer
Signal Processing & Feature Extraction Layer
Physical & Hardware Layer

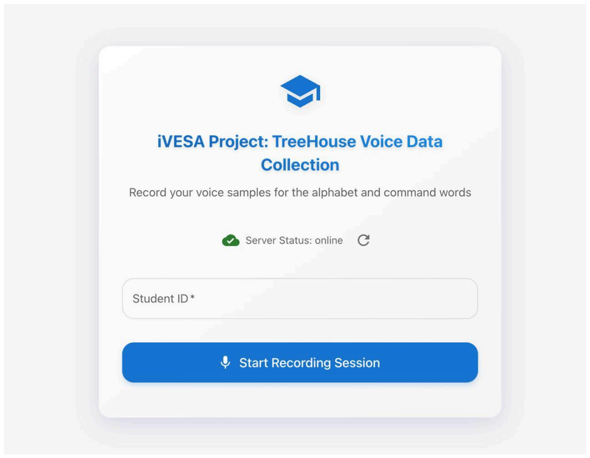
The foundational level where we are collecting the voice files has to be perfect. Otherwise, the steps above will all have errors that will be compounding as we go up each level. My partner, Jigao (Thomas) Wang is working on the bottom later in which he is designing several voice encapsulation devices. My other partner, Joshua Siew, is working on the top layer where he is implementing Unity onto the Meta Quest 3 VR Device. My responsibilities are the two layers in the middle. Now to first obtain the voice files, we must create the frontend for the user interface. I utilized the Material UI library for the frontend for seamless user experience. For the backend, I utilized Node.js and Express. Here is the ER Diagram representing the data:



USER			
int	studentId	PK	Unique User Identifier

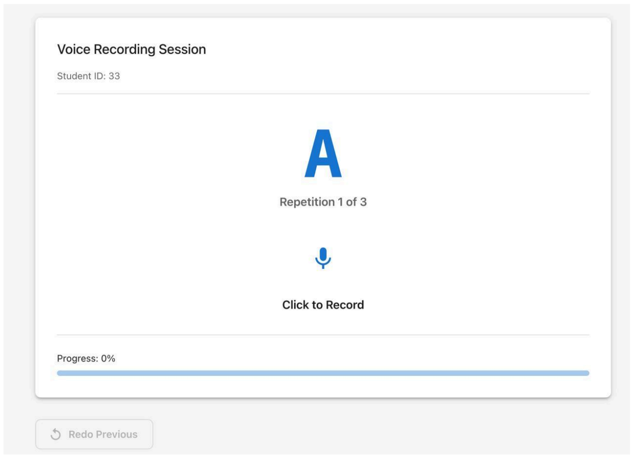
RECORDING			
string	filename	PK	The unique filename on the server (e.g., 99_1_A.wav)
int	studentId	FK	Derived from filename, references USER
int	repetition		The repetition number, derived from filename
string	value		The recorded value (e.g., 'A', 'Done'), derived from filename
string	type		Implicitly 'letter' or 'command', based on the value

As you can see, each user can have multiple recordings but each recording has exactly one unique user. I structured my diagram like this since at the TreeHouse trials, each of the children will receive a unique ID label, which they will input into the software. The interface loops through the letters A-Z and then the commands (Done, Enter, Delete, Repeat, Backspace, Again, Undo, Tutorial, and Screening). This will then loop over 3 times. After recording, the file will be immediately uploaded locally to the uploads folder in the following format: '{studentId}_{repetition}{item}.webm' where item refers to the letter or command. Here is some screenshots of the frontend UI:



iVESA Project: TreeHouse Voice Data Collection

Record your voice samples for the alphabet and command words



Voice Recording Session

Student ID: 33

The redo previous button can be utilized to redo any of the previous files. In the case you press redo too much, keep in mind you need to start recording from the item that you hit redo. All in all, this means that you can collect 105 voice files from one singular person.

Discussions/Findings from TreeHouse Trials

Subjects 1-11 had okay voice files but 12-20 were very bad voice files, meaning that we collected 1,155 okay voice files and 945 bad voice files. Among those 11 sets of okay voice files, the air conditioner and other background noise from other stations in the same room. The rest of the 9 sets of voice files were recorded and uploaded successfully but we could not hear the rest of the 9 students, indicating a hardware issue.

Some challenges that we faced at the trials was that we had to work in a confined space with other stations, so that may have caused more background noise in addition to the air conditioner. The laptop battery also ran really low since the software can only run locally on one machine. The microphone also connected via a phone which then sent audio to the computer, which is a very unnecessary multi-step process.

Future Research & Handing Over

1) Make sure the microphone and laptop(s) are all fully charged before TreeHouse trials
2) Train children beforehand for the data collection process OR have someone guide the students by clicking for them (latter is preferred)
    a) Relisten to the audio files after EACH student. Ensure that the audio files are perfect (i.e. no noise, can hear clearly what they are saying). Otherwise, make the student redo entirely OR press undo and start from the badly recorded file (the file for each student will overwrite previous files so duplicates will not be an issue).
3) Be in a completely isolated and quiet environment (no other stations, no air conditioner or background noise)
4) Connect microphone to bluetooth directly to recording device running the software
5) Configure software to cloud so that we do not have a charging bottleneck and more devices can be utilized at once (suggestion: MongoDB).

Most importantly, here is the repo: https://github.com/sampark0523/TreeHouseDataCollection. The instructions to run this software on your own are inside the README. Make sure to have Node.js, npm (installed with Node.js), and git (separate git for macOS or Windows) before following any of the instructions listed in the README.md.

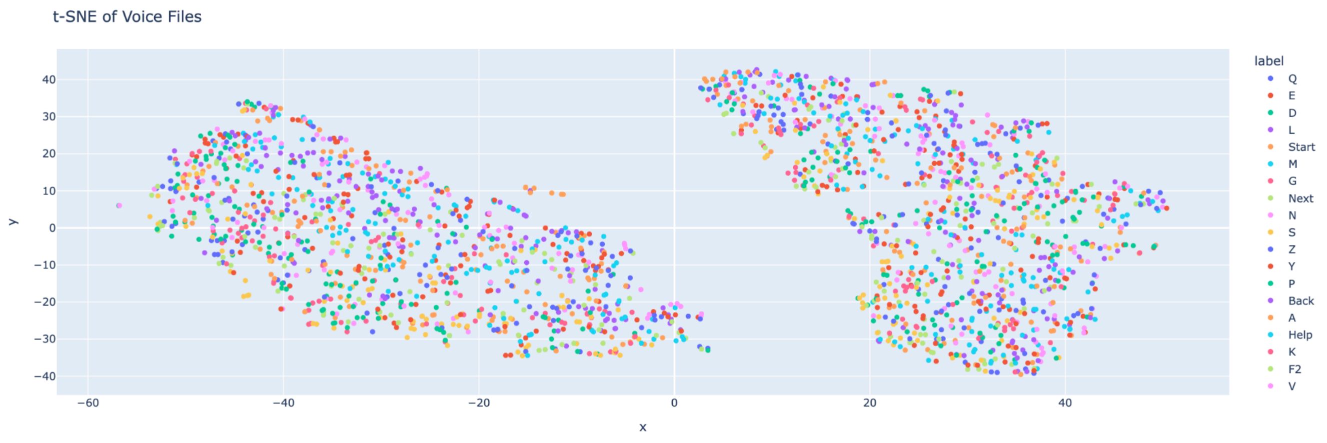
4. Initial Approaches for Voice Recognition Model

I initially utilized PocketSphinx, a lightweight pretrained speech recognition model. However, after working on this training for around 2 weeks, the most accuracy that PocketSphinx was able to output was at most 60-80%. Here is the entire documentation on how to train utilizing the 2303 (keep in mind one of the files is corrupted and cannot be converted to the intended 16000Hz sample format, specifically 4b_Done.wav) voice files from one of the past TreeHouse trials: https://cmusphinx.github.io/wiki/tutorialam/. I highly discourage trying this out yourself though because I wasted an ample amount of time to little avail. This is a legacy model, and if you read the README for the actual CMUSphinx PocketSphinx github, this is what it states: “This is PocketSphinx, one of Carnegie Mellon University's open source large vocabulary, speaker-independent continuous speech recognition engines. Although this was at one point a research system, active development has largely ceased and it has become very, very far from the state of the art. I am making a release, because people are nonetheless using it, and there are a number of historical errors in the build system and API which need to be corrected.”

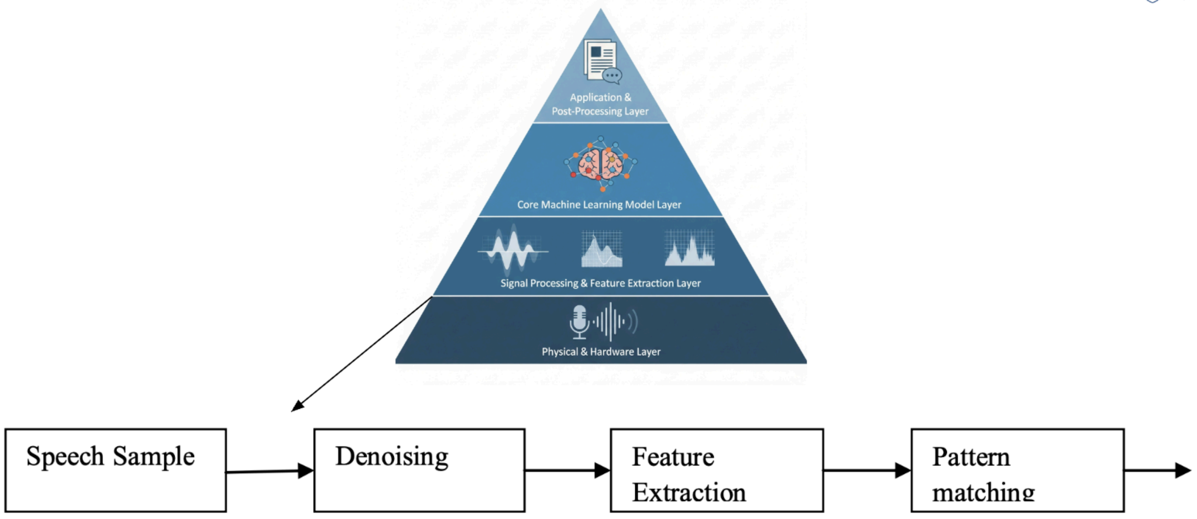
5. Initial Pivot Step: t-SNE Analysis

I first mounted the 2302 voice files onto google colab. I looped through each audio file and utilized the librosa library to load each audio file and extract its 13 MFCC features (a standard representation of sound). Then I created a dataset by pairing each audio feature with a label that is automatically extracted from the filename itself. It converts the collected features into a NumPy array, the standard format for use in machine learning models.

t-SNE (t-Distributed Stochastic Neighbor Embedding) is a machine learning algorithm used to visualize high-dimensional data. It takes your embeddings which have 13 dimensions and reduces them down to just two dimensions. This is the resulting graph:



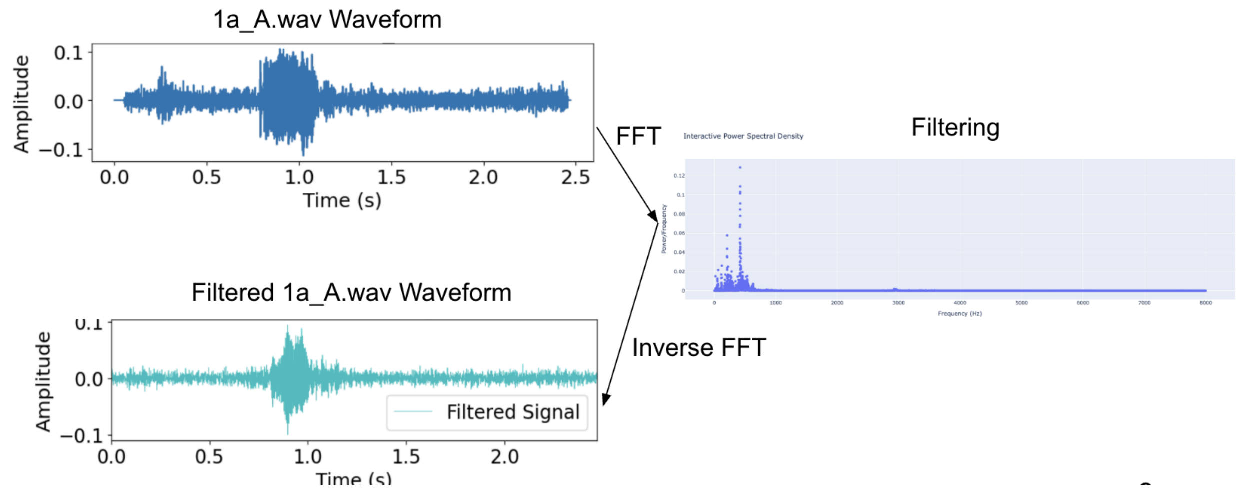
If we do some analysis here, the left batch of files were recorded inside in a less noisy environment while the right hand side were recorded in a noisy environment outside. The file names are named [#][a/b]_[letter/command].wav where lowercase ‘a’ before the ‘_’ means noisy environment and ‘b’ meaning less noisy environment. Our machine learning model perfectly split the data into two such that the [#][b]_[letter/command].wav files are in the left batch and [#][a]_[letter/command].wav files are the right batch. This gives us an extremely important insight: perhaps our models were not distinguishing between voice content but whether the voices files contained minimal or lots of noise. This is our fixed workflow:


Application & Post-Processing Layer
Core Machine Learning Model Layer
Signal Processing & Feature Extraction Layer
Physical & Hardware Layer
Speech Sample
Denoising
Feature Extraction
Pattern matching

As you can see, we will be denoising first prior to the feature extraction step.

6. Denoising Utilizing the Fast Fourier Transform

The Fast Fourier Transform (FFT) is an efficient algorithm that acts like a prism for signals. Just as a prism separates white light into its constituent colors, the FFT separates a complex signal into the individual frequencies that make it up. It essentially shifts an audio signal from the time domain (amplitude vs. time) to the frequency domain (viewing which frequencies are present and at what strength).



The top-left shows the signal's amplitude over time. The original waveform is then transformed from the time domain into the frequency domain via FFT. The resulting Power Spectral Density plot shows which frequencies are present in the signal and their corresponding power or strength. In this frequency view, the noise can be identified and removed. Typically, background noise exists as high-frequency components while the main signal is concentrated in lower frequencies, and this filtering step removes these unwanted noise frequencies. In this case, any frequency with a power of < 0.01 is removed and then computing the inverse fourier transform outputs your final, denoised audio. This process is known as Spectral Gating. When denoising our data, we will utilize a more specific variant of the FFT known as the STFT (Short-Time Fourier Transform) which follows an identical filtering process.

7. Preprocessing, Denoising, Extracting Features, Data Augmentation, CNN, & t-SNE

The pipeline we are going to follow in hopes of creating a model with 99% accuracy for our proof of concept is as follows: loading and preprocessing audio, denoising using Fast Fourier Transform, extracting 13 MFCCs with delta and delta-delta features, applying simple data augmentation (pitch-shift, time-stretch), training a lightweight 1D-CNN model, and generating t-SNE visualization of the learned embeddings.

7.1. Load and preprocess audio

Let us first load and preprocess the 2302 voice files that we have mounted. Make sure to run these commands to mount your google drive on google colab and log in:

```
from google.colab import drive
drive.mount('/content/drive')
```

If you are trying to recreate this, make sure to edit DATA_ROOT = Path("/content/drive/MyDrive/audio_files/wav"), and replace it with your own file path to your .wav files.

```
import numpy as np
import soundfile as sf
from pathlib import Path
import os
import librosa
from tensorflow.keras.preprocessing.sequence import pad_sequences

DATA_ROOT = Path("/content/drive/MyDrive/audio_files/wav")
TARGET_SR = 16000 # have consistent target sample rate

# lists to store preprocessed audio, sampling rates, and labels
preprocessed_audio = []
sampling_rates = []
audio_labels = []
audio_filenames = []
```

```
# iterate through the .wav and .WAV files in the specified directory
wav_files = sorted(list(DATA_ROOT.rglob("*.wav")) +
list(DATA_ROOT.rglob("*.WAV")))

for wav_path in wav_files:
   filename = wav_path.name
   parts = wav_path.stem.split('_')
   if len(parts) > 1:
       potential_label = parts[-1] # extracts the letter
       if len(potential_label) == 1 and 'A' <= potential_label <= 'Z':
           try:
               # load audio data and its sampling rate
               audio, sr = sf.read(wav_path)

               # converting to mono
               if audio.ndim > 1:
                   audio = np.mean(audio, axis=1)

               # resample if the sampling rate is not the target rate
               if sr != TARGET_SR:
                   # librosa.resample expects float32
                   audio = librosa.resample(y=audio.astype(np.float32),
orig_sr=sr, target_sr=TARGET_SR)
                   sr = TARGET_SR

     # append the preprocessed audio data, sampling rate, and label
               preprocessed_audio.append(audio)
               sampling_rates.append(sr)
               audio_labels.append(potential_label)
               audio_filenames.append(filename)

           # for decoding if you have issues
           except Exception as e:
               print(f"Error processing file {filename}: {e}")

# max length of the audio files in the list
max_length = max(len(audio) for audio in preprocessed_audio)

# normalizing audio files to have consistent length
```

```
padded_audio = pad_sequences(preprocessed_audio, maxlen=max_length,
dtype='float32', padding='post', truncating='post', value=0.0)

# convert lists to numpy arrays
padded_audio = np.array(padded_audio)
sampling_rates = np.array(sampling_rates)
audio_labels = np.array(audio_labels)
audio_filenames = np.array(audio_filenames)
```

This load and preprocessing code first scans your directory for all .wav files and for each one, it extracts a single uppercase letter associated with the filename and uses it as a label. It then standardizes the audio to have a uniform mono and 16,000 Hz sampling rate. In addition, since machine learning models require inputs to have the same size, the code also finds the length of the longest audio clip in the dataset and then pads all shorter clips with silence to make them match the length. It then converts the processed audio data, labels, and filenames into NumPy arrays. This crucial step organizes the data into standard format required by machine learning libraries such as TensorFlow or Keras.

7.2. Denoising

The filtering utilizing short-time fourier transform STFT is done in the following code:

```
import numpy as np
import librosa
from tensorflow.keras.preprocessing.sequence import pad_sequences

denoised_audio = []
DENOISING_THRESHOLD_FACTOR = 0.1 # factor to filter signal, can change

for audio_data in padded_audio:
   # convert audio to STFT (short time fourier transform)
   stft = librosa.stft(audio_data.astype(np.float32))
   magnitude, phase = librosa.magphase(stft)

   # calculate noise threshold (normalized with respect to the voice file)
   noise_threshold = np.mean(magnitude) * DENOISING_THRESHOLD_FACTOR
```

```
    # apply spectral gate to remove noise below the threshold
    denoised_magnitude = magnitude * (magnitude > noise_threshold)

    # reconstruct the denoised audio signal by performing inverse STFT
    denoised_stft = denoised_magnitude * phase
    denoised_audio_data = librosa.istft(denoised_stft)

    denoised_audio.append(denoised_audio_data)

# re-pad sequences to ensure consistent length after the STFT & ISTFT
denoised_audio_padded = pad_sequences(denoised_audio, maxlen=max_length,
dtype='float32', padding='post', truncating='post', value=0.0)
```

While the FFT analyzes the entire signal at once and gives you a single, static snapshot of all the frequency components present in the whole audio clip, the STFT breaks the signal into short, overlapping windows and runs an FFT on each individual window (essentially STFT is a short hand for saying that FFT is run on each time splice). This results in a moving picture that shows how the signal's frequency content changes over time. The reason we utilize STFT over FFT is because audio signals are non-stationary and their frequencies are changing constantly based on the beginning, middle, and end. STFT solves this issue by preserving the time information, meaning that the filter adapts to the changing nature of the signal, removing noise from quiet sections without damaging the actual speech in louder sections. STFT is essentially how you make spectrograms.

Moving on to analyzing the code, the code first takes the audio waveform and computes its STFT. This separates the signal into its two core components: intensity of each frequency at each moment in time and the timing information for each frequency. It then calculates a dynamic noise_threshold for filtering. Now something that is different that this code optimizes is it takes this filtered magnitude and combines it with the original, unchanged phase. This is to preserve natural sound and structure of the audio. While the magnitude does represent what frequencies are present and how loud they are, the phase contains the crucial timing and underlying structure that makes the audio coherent. That is why we combine the two.

7.3. Extracting Features

Extracting features from the denoised audio files is done as follows:

```
# extract MFCCs and their deltas from the denoised audio
n_mfcc = 13
denoised_features = []

for audio_data in denoised_audio_padded:
   # calculate MFCCs and their first derivative and acceleration (2 deg)
   mfccs = librosa.feature.mfcc(y=audio_data, sr=TARGET_SR, n_mfcc=n_mfcc)
   delta_mfccs = librosa.feature.delta(mfccs)
   delta2_mfccs = librosa.feature.delta(mfccs, order=2)

   # stack all these features together for all the audio files
   stacked_features = np.vstack((mfccs, delta_mfccs, delta2_mfccs)).T
   denoised_features.append(stacked_features)

# convert list of features into a single NumPy array
denoised_features = np.array(denoised_features)
```

For each audio file, it first computes the standard MFCCs (Mel-Frequency Cepstral Coefficients) and these 13 coefficients provide a snapshot of the spectral shape of the sound in short frames. Now we then analyze the rate of change of the MFCCs from one frame to the next and the delta-delta's, or the rate of change of the velocity (acceleration) of the MFCCs. This dynamic information is crucial for tasks like speech recognition, as it helps the model understand the transitions between sounds in a specific letter. The code then stacks these three sets of features into a single feature matrix for each audio file. Then it converts the list of individual feature matrices into a single, large NumPy array.

7.4. Data Augmentation

The next step is to apply data augmentation techniques to these features to increase the size and diversity of the training data.

```
# augment data by adding noisy copy of each feature set to double the
dataset size so as to have more training data
```

```
augmented_denoised_features = []
augmented_labels_denoised = []

noise_factor = 0.005 # can adjust the level of noise, here we add 0.005

for i in range(len(denoised_features)):
   # add original + clean features together
   augmented_denoised_features.append(denoised_features[i])
   augmented_labels_denoised.append(audio_labels[i])

   # create and add a noisy version of the same features
   noise = np.random.randn(*denoised_features[i].shape) * noise_factor
   noisy_features = denoised_features[i] + noise
   augmented_denoised_features.append(noisy_features)
   augmented_labels_denoised.append(audio_labels[i])

# convert lists to NumPy arrays for model training
augmented_denoised_features = np.array(augmented_denoised_features)
augmented_labels_denoised = np.array(augmented_labels_denoised)
```

Our code here essentially adds the original, clean feature set to a new list. It then creates a noisy version by adding a small amount of random noise and adds this modified version to the list as well. The result is a new dataset that is twice the size of the original, containing both the clean features and slightly altered versions. By training the model on both the original clean data and slightly noisy variations, you are training the model to recognize that small, irrelevant changes to the input features should not change the final prediction. This helps the model focus on the true underlying patterns of the audio rather than memorizing the exact details of the training examples. Consequently, the model will be less sensitive to real-world imperfections such as outliers and will perform better when it encounters new unseen data. **We manually add random noise after denoising to ensure that we are adding controlled variations to a clean, high-quality baseline**. This yields a much better output than purely training a model on a dataset with lots of noise (recorded outside) and minimal noise (recorded inside)!

7.5. CNN (Convolutional Neural Network)

The next step is to prepare the data for CNN training by encoding the labels and splitting the data into training and testing sets:

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from tensorflow.keras.utils import to_categorical
import tensorflow as tf
import numpy as np

# prepare labels: convert string labels to one-hot encoded vectors
label_encoder = LabelEncoder()
integer_encoded_labels =
label_encoder.fit_transform(augmented_labels_denoised)
one_hot_labels = to_categorical(integer_encoded_labels)

# reshape features to add a 'channel' dimension for the model input
augmented_denoised_features_reshaped =
np.expand_dims(augmented_denoised_features, axis=-1)

# split the dataset into training and testing sets (80% train, 20% test)
X_train_denoised, X_test_denoised, y_train_denoised, y_test_denoised =
train_test_split(
   augmented_denoised_features_reshaped,
   one_hot_labels,
   test_size=0.2,
   random_state=42,
   stratify=one_hot_labels
)
```

This code snippet converts the text labels into a numerical one-hot encoded format, adds a channel dimension to the features as required by the models, and then splits the data into an 80% training set and a 20% testing set.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv1D, MaxPooling1D, Flatten, Dense,
Dropout, BatchNormalization
from tensorflow.keras.regularizers import l2
from tensorflow.keras.callbacks import EarlyStopping

# define 1D-CNN model architecture
model_denoised = Sequential()

# input convolutional block
```

```
model_denoised.add(Conv1D(filters=32, kernel_size=5, activation='relu',
input_shape=(X_train_denoised.shape[1], X_train_denoised.shape[2])))
model_denoised.add(BatchNormalization())
model_denoised.add(MaxPooling1D(pool_size=2))
model_denoised.add(Dropout(0.3))

# second convolutional block
model_denoised.add(Conv1D(filters=64, kernel_size=5, activation='relu'))
model_denoised.add(BatchNormalization())
model_denoised.add(MaxPooling1D(pool_size=2))
model_denoised.add(Dropout(0.3))

# third convolutional block
model_denoised.add(Conv1D(filters=128, kernel_size=5, activation='relu'))
model_denoised.add(BatchNormalization())
model_denoised.add(MaxPooling1D(pool_size=2))
model_denoised.add(Dropout(0.3))

# flatten for dense layers
model_denoised.add(Flatten())

# dense layers for classification
model_denoised.add(Dense(128, activation='relu',
kernel_regularizer=l2(0.001)))
model_denoised.add(BatchNormalization())
model_denoised.add(Dropout(0.4))

model_denoised.add(Dense(64, activation='relu',
kernel_regularizer=l2(0.001)))
model_denoised.add(BatchNormalization())
model_denoised.add(Dropout(0.4))

# final output layer
model_denoised.add(Dense(num_classes, activation='softmax'))

# compile the model
model_denoised.compile(optimizer='adam',
                       loss='categorical_crossentropy',
                       metrics=['accuracy'])
```

```
model_denoised.summary()

# set up early stopping to prevent overfitting
early_stopping = EarlyStopping(monitor='val_loss', patience=15,
restore_best_weights=True)

# train the model
epochs = 200
batch_size = 32
history_denoised = model_denoised.fit(X_train_denoised, y_train_denoised,
                                       epochs=epochs,
                                       batch_size=batch_size,
                                       validation_data=(X_test_denoised,
y_test_denoised),
                                       callbacks=[early_stopping])
```

This code defines, compiles, and trains a 1D Convolutional Neural Network (CNN) for classifying time-series data. The model architecture is built sequentially, starting with three convolutional blocks that serve as a feature extractor; each block contains a Conv1D layer to learn patterns, BatchNormalization to stabilize the learning process, MaxPooling1D to downsample the data while retaining key information, and Dropout to prevent the model from memorizing the training data. After the features are extracted and flattened into a one-dimensional vector, the data is passed through a classifier made of two dense, fully-connected layers, which are also regularized, before a final softmax output layer generates a probability for each potential class. The script then compiles this architecture by setting the Adam optimizer, the categorical crossentropy loss function for measuring errors, and the accuracy metric for monitoring performance. Finally, it initiates the training process on the provided data for up to 200 epochs, but it cleverly employs an EarlyStopping callback to watch the model's performance on a separate validation set, automatically halting the training and restoring the best model weights if performance fails to improve for 15 consecutive epochs, thereby ensuring the final model is effective and not overfitted.

```
from tensorflow.keras.models import Model

import numpy as np

# create a new model to extract features from the second-to-last layer

```

```
feature_extractor_model_denoised = Model(inputs=model_denoised.inputs,
outputs=model_denoised.layers[-2].output)

# reshape original denoised features to match the model's input shape

denoised_features_reshaped = np.expand_dims(denoised_features, axis=-1)

# use the new model to get deep embeddings from the original
(non-augmented) data

cnn_embeddings_denoised_model =
feature_extractor_model_denoised.predict(denoised_features_reshaped)

print("Shape of extracted CNN embeddings from denoised model:",
cnn_embeddings_denoised_model.shape)
```

This code creates a new model from a trained CNN that outputs the layer before the final one, essentially capturing the learned features or "embeddings". It then uses this new model to process your (reshaped) denoised audio data and stores these extracted embeddings.

```
import re


# create a boolean mask for single letter labels (A-Z)
single_letter_mask = np.array([bool(re.fullmatch(r'[A-Z]', label)) for
label in audio_labels])


# filter the CNN embeddings and labels using the mask
filtered_embeddings = cnn_embeddings_denoised_model[single_letter_mask]
filtered_labels = audio_labels[single_letter_mask]
```

This code snippet uses a pattern to identify audio files labeled with a single uppercase letter (A-Z) and creates a filter. It then applies this filter to keep only the corresponding CNN embeddings and labels, ensuring that the subsequent t-SNE visualization focuses specifically on these single-letter data points.

7.6. t-SNE Graph Visualization

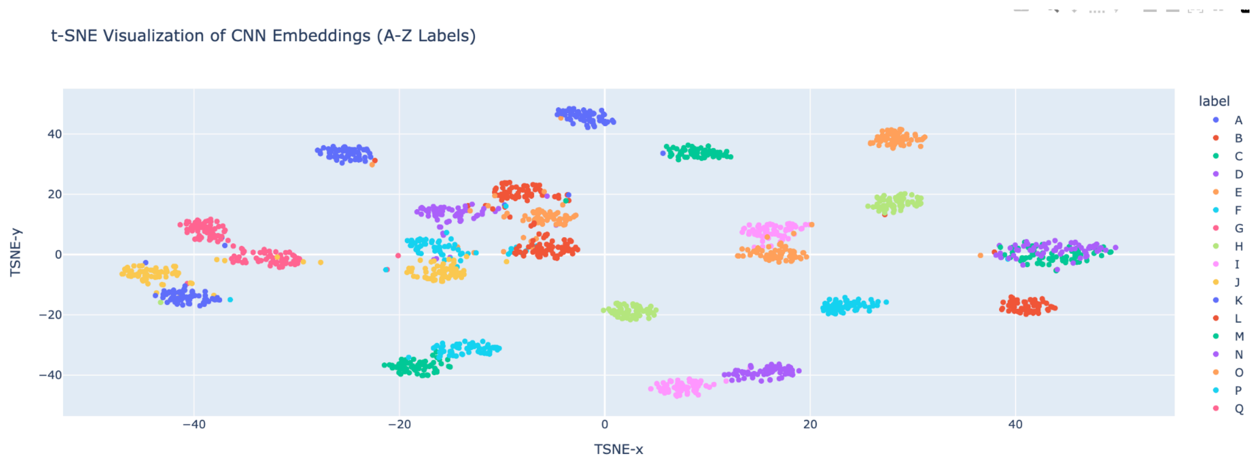
```
from sklearn.manifold import TSNE
```

```
tsne = TSNE(n_components=2, perplexity=30, random_state=42)
tsne_results = tsne.fit_transform(filtered_embeddings)
import pandas as pd
import plotly.express as px
import numpy as np


filtered_filenames = audio_filenames[single_letter_mask]
df_tsne = pd.DataFrame({
   'TSNE-x': tsne_results[:, 0],
   'TSNE-y': tsne_results[:, 1],
   'label': filtered_labels,
   'filename': filtered_filenames
})

fig = px.scatter(df_tsne,
                 x='TSNE-x',
                 y='TSNE-y',
                 color='label',
                 hover_data=['filename', 'label'],
                 title="t-SNE Visualization of CNN Embeddings (A-Z
Labels)")

fig.show()
```

This code utilizes the t-SNE algorithm to graph all the audio files into both consistent and unique clusters!

8. Results and Discussion

This is the t-SNE graph we end up getting. As you can see, we have both consistent and unique clusters!

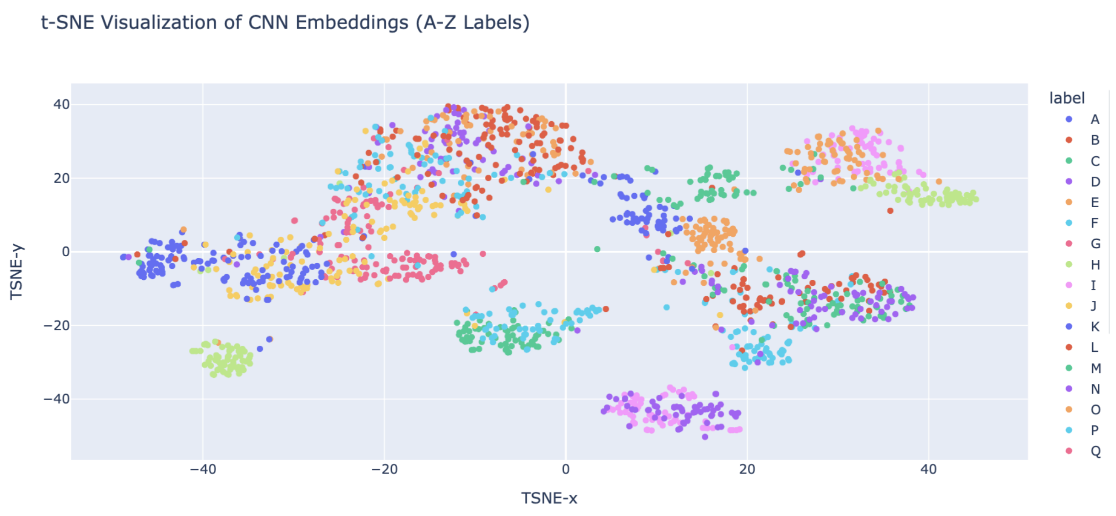


A t-SNE projection of the CNN embeddings was used to visualize the model's feature separation capabilities at the phoneme level. Visual inspection of the 1,663 voice file embeddings reveals distinct clustering, yet identifies approximately **70-90 classification errors**. This corresponds to a model **accuracy of approximately 94.6% to 95.8%**. These errors are not random but highlight specific acoustic similarities that the model found challenging to differentiate.

The analysis reveals significant feature overlap between several phoneme pairs, most notably **J/K, Q/G, C/Z, S/X, and I/Y**. A more pronounced lack of separation was observed among phonetically related consonants, including a large group of **plosives (T, P, D, B), fricatives (E, V), and nasals (M, N)**. This indicates that while the model correctly learned fundamental acoustic similarities between these groups, it requires further refinement to distinguish their subtle, unique features. A detailed breakdown of these inter-phoneme similarities is available in the supplementary materials for further analysis.

In addition, a crucial breakthrough was found in the data pre-processing step which I believe is the singular best insight gained from this project: ***Adding random noise after denoising to ensure that we are adding controlled variations to a clean, high-quality baseline is a far better model than training a model only on less and more noise like the voice files that we have.***

In essence, the **augmented and denoised audio files** were used to **train the CNN model**. The augmentation helped the CNN learn to generalize better to variations in the audio, and the denoising helped it focus on the relevant parts of the audio signal. Once the CNN was trained, we then took the original, denoised audio files (not the augmented ones) and passed them through the trained CNN to get their embeddings, and finally, the t-SNE algorithm used these embeddings from the original, denoised files to create the 2D visualization.



For reference, here is how the t-SNE graph works if we train only with the augmented audio. As you can see, the clusters are not as consistent compared to the previous graph. Therefore, we have demonstrated the importance of ***denoising utilizing STFT in the preprocessing step.***

9. Limitations and Future Work

9.1. Challenges and Limitations

The primary limitations of the current model are twofold. First, the **limited dataset size** (n=2,302 total files) presents a risk of overfitting and may not capture the full variance of spoken letters. Second, there is an **ambiguity in error source**; it is difficult to distinguish between genuine model prediction errors and anomalies in the source data, such as mispronunciations or incorrect labels (human error).

9.2. Recommendations for Future Work

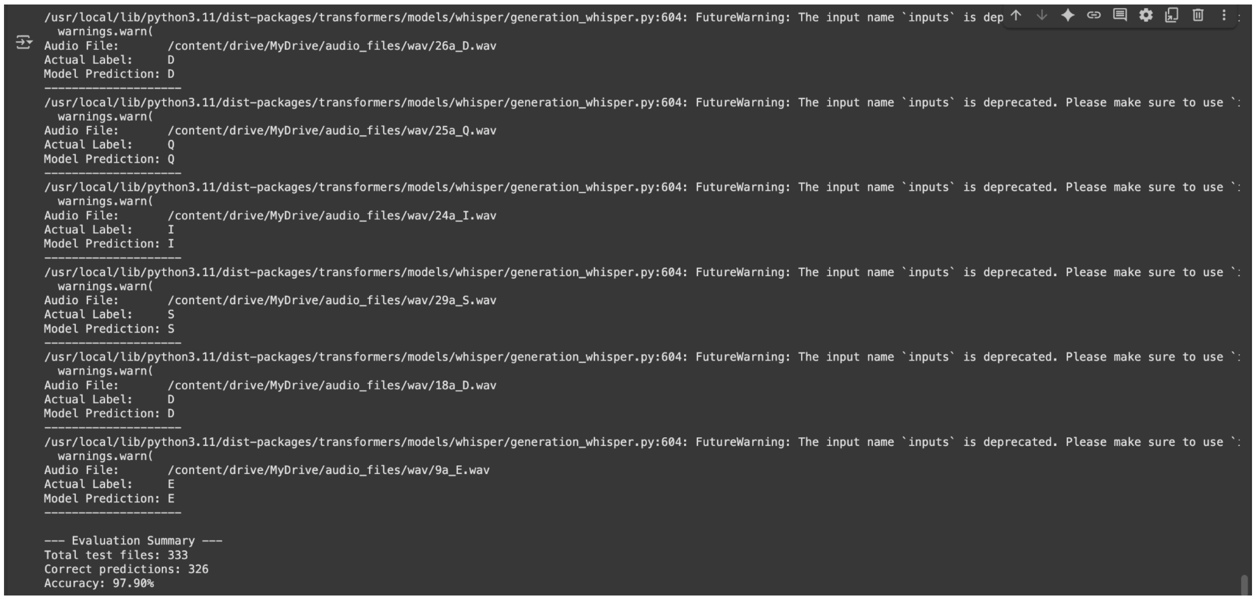
1. **Targeted Feature Analysis:** Conduct a deep-dive analysis into the acoustic features of the commonly confused phoneme groups (e.g., plosives). By examining their spectrograms and feature vectors, unique discriminative patterns can be identified and used to refine the model architecture or feature engineering process.
2. **Dataset Curation and Expansion:** A critical next step is to **significantly expand the dataset** to a target of 12,000-25,000 audio files. This expansion must be paired with a rigorous data collection and validation process to ensure high-quality, clean labels. Outliers identified in the current dataset (available for review in the interactive visualizations) should be reviewed and either corrected or removed.
3. **Refined Training Strategy:** Re-evaluate the necessity of an explicit denoising pre-processing step. Evidence suggests that models trained on diverse datasets, inclusive of natural ambient noise, often demonstrate greater robustness and real-world performance. It is recommended to experiment with training the model directly on non-denoised audio to improve its generalizability.
4. **Hierarchical Model Expansion:** Once phoneme-level accuracy is sufficiently improved, the model's scope should be expanded to identify full command words. A similar analytical framework can then be applied to resolve potential confusion between acoustically similar words.

Here is link to all my colab notebooks for initial t-SNE, FFT denoise exploration, (t-SNE w and w/o FFT) + CNN, Fine-tuning user guide, software github, visualization of all the voice files, audio_features.csv.

10. OpenAI Whisper

OpenAI Whisper is an **automatic speech recognition (ASR)** system designed to convert spoken audio into accurate written text. It was trained on a massive and diverse dataset of 680,000 hours, making it exceptionally robust at understanding various accents, handling background noise, and recognizing technical language.

By fine tuning openai-whisper constraining to A-Z, I received a 97.8% accuracy by testing on 333 voice files. However, out of those voice files, 6 out of 7 of the incorrect voice files were clearly outliers, meaning that our new accuracy is 99.69% accuracy.



```
/usr/local/lib/python3.11/dist-packages/transformers/models/whisper/generation_whisper.py:604: FutureWarning: The input name `inputs` is dep
  warnings.warn(
Audio File:       /content/drive/MyDrive/audio_files/wav/26a_D.wav
Actual Label:     D
Model Prediction: D
--------------------
/usr/local/lib/python3.11/dist-packages/transformers/models/whisper/generation_whisper.py:604: FutureWarning: The input name `inputs` is deprecated. Please make sure to use `:
  warnings.warn(
Audio File:       /content/drive/MyDrive/audio_files/wav/25a_Q.wav
Actual Label:     Q
Model Prediction: Q
--------------------
/usr/local/lib/python3.11/dist-packages/transformers/models/whisper/generation_whisper.py:604: FutureWarning: The input name `inputs` is deprecated. Please make sure to use `:
  warnings.warn(
Audio File:       /content/drive/MyDrive/audio_files/wav/24a_I.wav
Actual Label:     I
Model Prediction: I
--------------------
/usr/local/lib/python3.11/dist-packages/transformers/models/whisper/generation_whisper.py:604: FutureWarning: The input name `inputs` is deprecated. Please make sure to use `:
  warnings.warn(
Audio File:       /content/drive/MyDrive/audio_files/wav/29a_S.wav
Actual Label:     S
Model Prediction: S
--------------------
/usr/local/lib/python3.11/dist-packages/transformers/models/whisper/generation_whisper.py:604: FutureWarning: The input name `inputs` is deprecated. Please make sure to use `:
  warnings.warn(
Audio File:       /content/drive/MyDrive/audio_files/wav/18a_D.wav
Actual Label:     D
Model Prediction: D
--------------------
/usr/local/lib/python3.11/dist-packages/transformers/models/whisper/generation_whisper.py:604: FutureWarning: The input name `inputs` is deprecated. Please make sure to use `:
  warnings.warn(
Audio File:       /content/drive/MyDrive/audio_files/wav/9a_E.wav
Actual Label:     E
Model Prediction: E
--------------------

--- Evaluation Summary ---
Total test files: 333
Correct predictions: 326
Accuracy: 97.90%
```

To ensure transparency and facilitate reproducibility, the complete implementation code is documented in a publicly accessible Google Colab notebook, which includes the specific files identified during the error analysis. The resulting fine-tuned model has been uploaded to the Hugging Face model repository here. The model was fine-tuned within the Google Colab environment, a process that required approximately two weeks of computation time utilizing standard CPU resources.

While the fine-tuned OpenAI Whisper model achieved a very promising preliminary accuracy of 97.8%, a comprehensive evaluation requires acknowledging certain limitations that inform the path forward. The validation was conducted on a limited test set (n=333), which underscores the critical and ongoing need to collect more diverse data for robust generalization. Furthermore, the adjustment of accuracy to 99.69% (326/327 accuracy) by excluding outliers. Rather than an oversimplification, the model's success on the constrained A-Z task serves as a strong proof-of-concept. This demonstrates the fine-tuning methodology's effectiveness and suggests that performance could be even stronger when applied to the project's ultimate goal: identifying a set of distinct command words, which are typically more acoustically separable than challenging letter pairs like 'M' and 'N'. A valuable next step would be to cross-reference the specific error files and confusion groups from both the initial FFT & CNN model and this Whisper model. Such an analysis would reveal whether errors are inherent to the data (e.g., persistent difficulty with plosives) or specific to the model architecture, thereby guiding future refinements.

11. Conclusion

This research successfully developed a high-accuracy voice recognition system for the iVESA project, tailored to Singaporean children's speech, thereby moving closer to the ambitious goal of 99% accuracy for voice-command interactions. The project followed an iterative development path, beginning with the determination that legacy tools like PocketSphinx were insufficient and pivoting to a custom-built solution. A significant contribution of this work is the development of a robust data processing pipeline, which demonstrated that a sequence of STFT-based denoising, feature extraction, and controlled data augmentation could train a 1D-CNN model to achieve approximately **95.8%** accuracy. This initial model proved the viability of the approach while also providing valuable insights into specific phoneme confusion groups, such as plosives and nasals, as well as the **importance of denoising utilizing STFT in the preprocessing step**.

The project culminated in a pivotal breakthrough by fine-tuning the state-of-the-art OpenAI Whisper model. This advanced approach yielded a preliminary accuracy of **97.8%** on the constrained A-Z task. Further analysis revealed that with the removal of clear data outliers, a potential accuracy of **99.69%** is achievable, indicating the methodology's strength and that the primary remaining barrier is data quality.

In conclusion, this project not only delivered a model that nears the target accuracy but also established a clear, validated workflow for creating specialized, high-performance voice recognition systems. The success of the letter-recognition task serves as a vital proof-of-concept, paving the way for the system's expansion to a full vocabulary of command words. The path forward is clear, centered on large-scale data collection to ensure model robustness and further error analysis to perfect performance, ultimately bringing an intuitive, voice-controlled diagnostic tool closer to clinical reality.

12. References

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., ... & Ghemawat, S. (2015). *TensorFlow: Large-scale machine learning on heterogeneous systems*. Software available from tensorflow.org.

[2] Chollet, F., et al. (2015). *Keras*. GitHub. Retrieved from https://github.com/keras-team/keras.

[3] Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., ... & Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, 585(7825), 357-362.

[4] McFee, B., Raffel, C., Liang, D., Ellis, D. P., McVicar, M., Battenberg, E., & Nieto, O. (2015). librosa: Audio and music signal analysis in python. In *Proceedings of the 14th Python in Science Conference*.

[5] McKinney, W. (2010). Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference* (Vol. 445, pp. 51-56).

[6] Park, S. (2025). *Data Collection Software for TreeHouse Trials*. GitHub. Retrieved from https://github.com/sampark0523/TreeHouseDataCollection.

[7] Park, S. (2025). *Fine-tuning Whisper for Singaporean Children's Speech*. Google Colab Notebook. Retrieved from https://colab.research.google.com/drive/1eBBA_whyq5yguZJBt7d8LnvLXsg3MPZL.

[8] Park, S. (2025). *whisper-singaporean-kids*. Hugging Face. Retrieved from https://huggingface.co/sampark0523/whisper-singaporean-kids.

[9] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... & Duchesnay, E. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12, 2825-2830.

[10] Plotly Technologies Inc. (2015). *Collaborative data science*. Montréal, QC. Retrieved from https://plotly.com.

[11] Radford, A., Kim, J. W., Xu, T., Brockman, G., McLeavey, C., & Sutskever, I. (2023). Robust Speech Recognition via Large-Scale Weak Supervision. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*.

[12] The CMU Sphinx Group. (n.d.). *CMU Sphinx Open Source Speech Recognition*. Carnegie Mellon University. Retrieved from https://cmusphinx.github.io.

[13] van der Maaten, L., & Hinton, G. (2008). Visualizing Data using t-SNE. *Journal of Machine Learning Research*, 9(11).