Enhancing Speech Recognition and Accessibility in Panopto Videos: A Comparative Analysis of OPENAI's Whisper and Traditional Automatic Speech Recognition

Master Of Science
Audio Engineering

Sabian Hibbs

2023

Page Break

Abstract –

This paper tests neural network-based transcription metrics including 'Word Error Rate, Character Error Rate, Sentence Error Rate, Word Accuracy, F1-Score and Overall Score'. Panopto's platform specific automatic speech recognition Deepspeech 0.9.2 will be tested and compared to Open AI Whisper, as well as a fine-tuned variant of the models used by Open AI called LMT2, a purpose-built model that has been further developed on datasets of speech. An internal test of 20,000 audio files will be carried out for all models and variants, with the best preforming variants of the LMT2 model being set as the foundation for an online test, where participants will be exposed to an audio file of speech originally from Panopto's platform, where the participants will select one of 3 transcriptions that is best correlated to the audio provided. Comparing the results of the models internal and online test with statistical analysis techniques.

Acknowledgements –

Emily Bayliss – Providing support throughout the project, especially when it came to the writeup.

Open AI – Providing the models that were pretrained and ready to use, fully opensource.

Huggingface Website – Providing all documentation for training and datasets.

Mozilla Common Voice – Providing the datasets used for tuning and testing.

Senior Lecturers at the University of Derby – Provided a platform suitable for this paper.

Monster Energy Drink – Providing needed caffeine in times of need.

Nvidia Developments – Providing support for hardware specific queries.

The Kitchen Kettle – Always there to warm me up and give me a boost in the morning.

Table of Contents – **Page**

List of Figures and Tables –

List of Abbreviations

ASR	Automatic Speech Recognition
HMM	Hidden Markov Model
NN	Neural Network
WER	Word Error Rate
SER	Sentence Error Rate
CER	Character Error Rate
LMT2	Whisper Model Fine Tuned Variant (Language Model Transform 2)
WSPR	Whisper Model Master

1.0 Introduction

With the advancements of machine learning platforms with respect to speech, the underpinning question if such technologies can be adapted and shaped to preform a purpose better than what is currently available. In the case of this paper, the rationale comes from an international student at the University of Derby campus that was studying as an international student, with a basic understanding of the English language. During the global pandemic of Covid-19, many students worked from home and relied heavily on the services provided by the website 'Panopto' for media sharing of the lectures and seminars. The Panopto video platform automatically transcribes the speech in the videos using a technology called Automatic Speech Recognition, a lightly accurate speech to text technology that has been used by the platform for over 5 years Panopto. (2022). The lack of accuracy in this technology has caused situations where non-native international English speakers are given either conflicting transcriptions of the lectures or seminars, or transcriptions that do not conform to any context to what was said in the recordings. The use of Open AI's Whisper neural network model can be used to improve on these factors. The following paper outlines such issues and provides a detailed analysis of both tuning a model for the use on the Panopto platform, as well as the implications of the Panopto automatic speech recognition system on a test set of data. Concluding with the overall results of the findings.

2.0 | 2.1 Literature Review - History of Automatic Speech Recognition (ASR)

The history of Automatic Speech Recognition (ASR) can be traced back to the advent of key technologies throughout the late 1800's and 1900's, where the inception and creation of Alexander Graham Bell's telephone on the 7th of March 1876, sparked a new era of verbal communication. This key technology was able to transmit speech from one person to another via transmission through conductive cable spanning many miles. The first communication of the telephone by Alexander Graham Bell, was made on the 10th of March 1876 to his assistant Thomas Watson, from the Bells Laboratory in Boston Massachusetts to Thomas Watsons workshop in the same building. Where Alexander Graham Bell's first communication was "Mr. Watson, come here, I want to see you." Thomas Watson recalled that he was able to hear Mr. Bell through the device clearly. And so, this became the origin of speech telecommunications. Bell labs then went on to show the world the telephone in 1876 at the Centennial Exposition in Philadelphia. Science and Technology (2020)

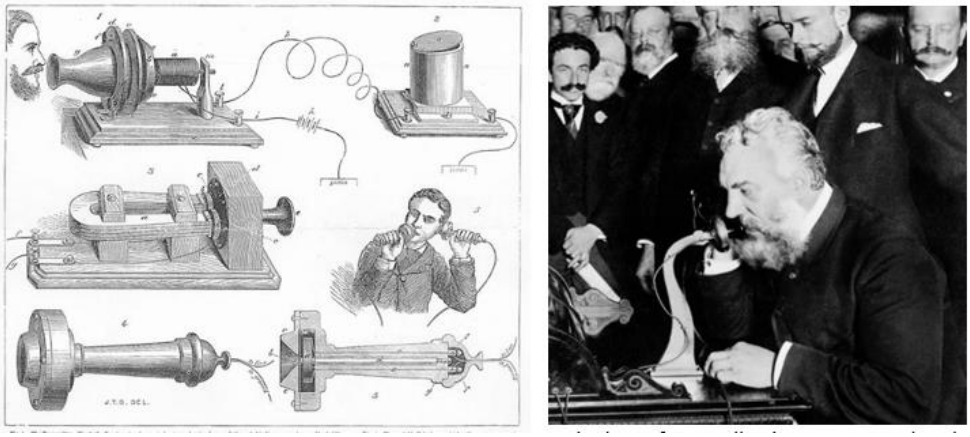

Figure 1 – (Left) Technical Drawings of the telephone from Bell Labs. Science and Technology (2020) (Right) Alexander Graham Bell speaking through the telephone. History.com Editors (2009).

The telephone itself in simple terms worked by utilising acoustic coupling. The term states that the sound energy from one person's speech, is transferred to a medium that induces electrical resistance so that it be sent through a transmission line to the recipient, where the signal itself would then be transferred back to acoustical sound, allowing the recipient to hear what was said from the other side. This technology was possible with the invention of the carbon microphone. The carbon microphone worked by having sound pressure vibrate a diaphragm connected mechanically to a thin layer of carbon powder, the vibration of the diaphragm would change the resistive properties of the carbon as electricity was passed through it, this change in resistance was used to capture the speech of whoever was speaking into it. This breakthrough in telecommunications sparked countless technologies that have helped communication both domestically and internationally, not only through the power of sending speech to other areas of the world, but also through the medium of machines, and how they interact with us. The following points made within this section will outline the history of speech to text, and its technological advancements from basic machines understanding voltages from speech to complex machines, that can mimic the intricacies of the human brain. History.com Editors (2009)

2.1.1 IBM Shoebox Speech Recognition 1961

The IBM Shoebox was first introduced by William C. Dersch in 1961, the name 'Shoebox' came from the machine itself being roughly the same size as a general shoebox at the time. IBM (2003). The device was classified as a speech recognition machine, that was able to understand up to 16 spoken words, that included numbers 0 to 9, along with various arithmetic operations such as addition and subtraction. Ibm.com (2012). The machine was then connected to an adding machine, a machine that housed a mechanical calculator and a typewriter device inside, that was able to do calculations while printing onto paper, this device was like the advent of store cash registers, where the inputs from a clerk would be calculated and displayed onto paper for the customer to see. To use the device, the user would speak into a carbon microphone using the array of pre-built commands such as 0 to 9 and basic arithmetic, the speech itself would be converted into an electrical signal impulse and sent

through to a measuring circuit. The circuit itself would measure and classify the signal impulse coming from the carbon microphone according to their frequency and amplitude.

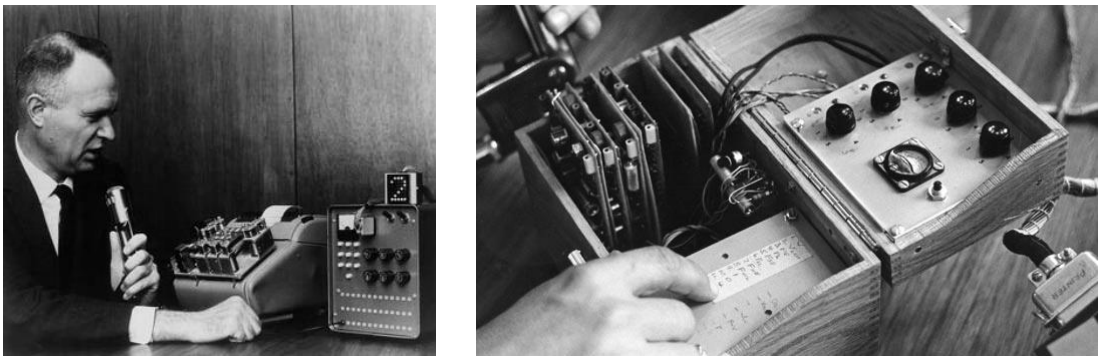

Figure 2 – (Left) William C. Dersch presenting the 'Shoebox' IBM (2003). (Right) Closer look at the 'Shoebox' and its internal workings Ibm.com. (2012).

This was done by having an array of 16 specific filters that would replicate the nuances of the frequency and amplitude of the input signal impulse, the filter with the highest voltage would then be sent to some internal physical relay logic, where it would send specific commands to the adding machine, based on the input itself. Ibm.com (2012). The machine itself was not capable of classifying more than the pre-determined voice commands, so any speech outside of the range of the device would not work as intended. This technology was defined as pattern matching, a simple yet effective way to classify data by measuring the input to an array of preexisting data. Put in simple terms, the device worked in a similar way to a common children's toy, where you place shapes into precut holes, the device works in a similar way, where you are comparing the shape of the object to the holes, only specific shapes can fit and therefore pass through the hole, but like most thing this comes with its own challenges, in a similar way you can sometimes place such shapes in holes that were not intended as the tolerance of the holes on the toy are low, you can end up with a situation where some shapes are wrongly placed through the wrong holes. The device itself had such limitations, where even the slightest background noise was able to change the desired output of the machine.

Despite its obvious limitations, the IBM Shoebox was a breakthrough in this field of electronics and its bridge to mechanical uses. The Shoebox proved that a machine could be capable of understanding the differences between specific words of speech and not only register the variances, but also act upon them. This research from IBM and William C. Dersch in 1961 inspired a new age of machines that could understand not only human voice but the nature of the human brain itself.

2.1.2 Dynamic Time Warping (DTW) – 1970

The Dynamic Time Warping algorithm was created by Taras Klymovych Vinsiuk a soviet researcher, in his publication "Speech Discrimination by Dynamic Programming" published under the Kibernetika, volume 4 1968, being Taras first publication. This algorithm was primarily used to determine the similarities of signals, similar to the use of the 'IBM Shoebox', where the principle of the technology was to use a comparison between input data, and reference data, although the Dynamic Time Warping algorithm itself can both align two temporal signals in the time domain and determine the similarities between the signals by comparing the 'cost' of aligning the two signals together. The term 'cost' is a phrase that outlines one of the algorithms outputs, determined by the last data point in its dataset. The algorithm can be used to classify the similarities between two sets of data or signal. In simple terms the algorithm uses a non-normalised float value to determine the similarity of the data you are trying to test. It gets this float value from calculating the 'cost-index' from a table of data, derived from two datasets. This algorithm can be used for speech recognition accuracy by allowing two signals to be aligned with respect to the levels of the input signal, to its reference signal. Simmilar to how the filter banks worked in the 'IBM Shoebox', the ability to compare an input signal to a reference signal, allows the algorithm to classify an alignment 'cost' this is the non-normalised float value mentioned in the previous statement. The algorithm itself can be simplified into its fundamental blocks shown below.

1. *Input:*
   Series $\mu = \{\mu_1, \mu_2, \ldots, \mu_{T_\mu}\}$
   Series $\upsilon = \{\upsilon_1, \upsilon_2, \ldots, \upsilon_{T_\upsilon}\}$
2. *Base Conditions:*
   $g(0,0) = 0$
   $g(1,1) = d(\mu_1, \upsilon_1) \cdot \omega_D$
   $g(i,0) = \infty \; for \; 1 \leq i \leq T_\mu$
   $g(0,j) = \infty \; for \; 1 \leq i \leq T_\upsilon$
3. *Recursive Relation:*
   $$g(i,j) = min \begin{cases} g(i,j-1) + d(\mu_i, \upsilon_j) \cdot \omega_V \\ g(i-1,j-1) + d(\mu_i, \upsilon_j) \cdot \omega_D \\ g(i-1,j) + d(\mu_i, \upsilon_j) \cdot \omega_H \end{cases}$$
   for $1 \leq i \leq T_\mu$ *and* $1 \leq j \leq T_\upsilon$
4. *Output:*
   DTW-Distance $D(\mu, \upsilon) = k(\omega) \cdot g(T_\mu, T_\upsilon)$

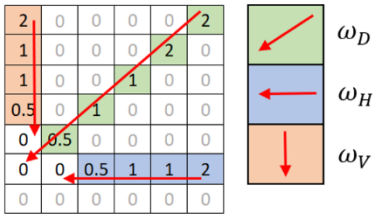

Figure 3 – DTW Algorithm Mathematical Expression

This algorithm can be reclassified and simplified in the following algorithm. Signal 1 = $x_{1:N}$ (Input signal you want to align.) Signal 2 = $y_{1:N}$ (Reference signal you want x to be aligned to.) Cost Matrix = $\boldsymbol{D} \in \mathbb{R}^{N+1 \times M+1}$ ($\mathbb{R}$ = Real Numbers for $x_{1:N}$ and $y_{1:N}$). Where $\boldsymbol{D}$ is the cost matrix, and the dimensionality of $\boldsymbol{D}$ is derived from the set of real numbers in the cost matrix of $\mathbb{R}^{N+1 \times M+1}$. Initialisation of matrix and primary inputs. $for \; i = 1 \; to \; N : \; D_{i,0} = \infty$, $for \; j = 1 \; to \; M : \; D_{0,j} = \infty$ Where $D_{0,0} = 0$ This denotes that all $D_{i,0}$ and $D_{0,j}$ prepopulate from 0 to the N set to infinity. Now the cost matrix has been created and initialised with the correct inputs, the process of calculating the data that goes inside the matrix. for i is referenced at each $i{:}N$ point, so the values of *Signal 1* will be represented at each $i{:}N$ point from *0*. This carries over to *Signal 2*, where its $j{:}N$ points are taken from the signals data starting from *0*. $x_{1:N} = i_N \; and \; y_{1:N} = j_N$

Calculate the cost matrix data:

$$for \; i = 1 \; to \; N:$$
$$for \; j = 1 \; to \; M:$$
$$D_{i,j} = d(x_i, y_j) + min \begin{cases} D_i - 1, j - 1 \; (match) \\ D_i - 1, j \quad (insertion) \\ D_{i,j} - 1 \quad (deletion) \end{cases} \tag{1}$$
$$d(x_i, y_j) = |x_i - y_j|$$

Get Alignment: Traceback from $D_{N,M}$ to $D_{0,0}$, x_1 = *[0, 2, 0, 1, 0, 0] and* y_1 = *[0, 0, 0.5, 2, 0, 1, 0]* So $x_{1:N} = i_{1:6} \; and \; y_{1:N} = j_{1:7}$, Alignment Cost (float) is the value at the $D_{N,M}$ datapoint.

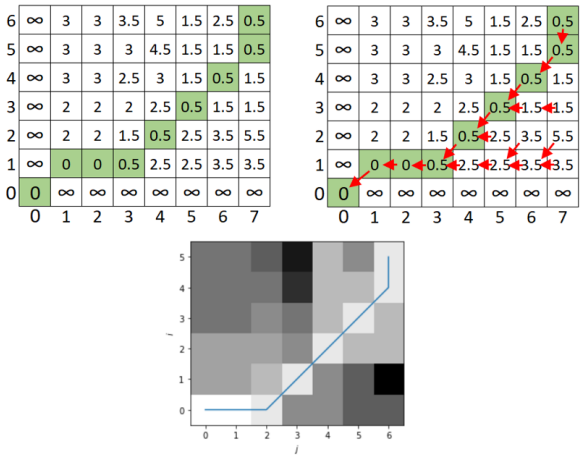

Figure 4 – Cost Matrix filled out with x and y data points.

Using the example shown in Figure 4, the $x_1 and\ y_1$ datapoints have been calculated for each column and row of the matrix. To calculate the time alignment, the calculation starting from $D_{N,M}$ must be made. Starting at this datapoint, a traceback must be made to the min value to the opposing blocks leading down to $D_{0,0}$, this will give a clear line that can be traced from $D_{N,M}$ to $D_{0,0}$. When the traceback moves through the cost matrix, the data at the point of the traceback will be the alignment needed for that data So $x_{1:N} = i_{1:6}\ and\ y_{1:N} = j_{1:7}$. This means that with the example in Figure 4, the cost index alignment is 0.5 at $D_{N,M}$.

Euclidean distance can be used to measure the distance between any two points in n-space and can be derived from the following equation.

$$d(\boldsymbol{p}, \boldsymbol{q}) = \sqrt{\sum_{i=0}^{n} (q_i - p_i)^2} \tag{2}$$

$\boldsymbol{p}, \boldsymbol{q}$ = Two points in Euclidean n-space.

q_i, p_i = Euclidean vectors, starting from the origin of the space (Initial Point).

n = n-space.

The term 'n-space' is derived from the mathematical principles of geometric space with n dimensions.

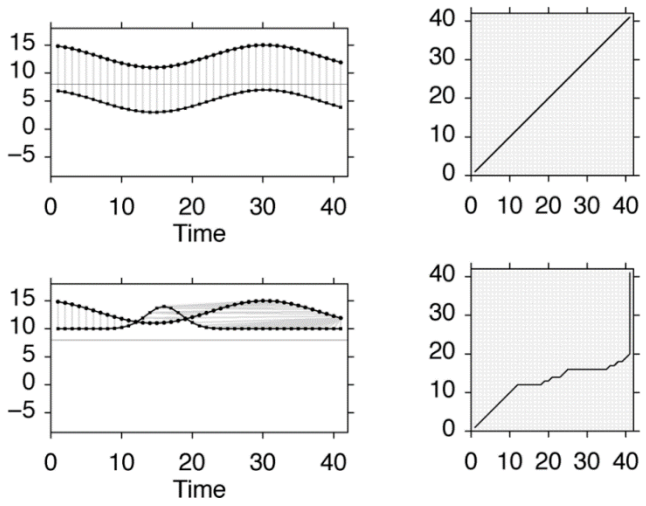

Figure 5 – Euclidean Distance of two signals and corresponding cost matrix (Top Left and Right) Non-Euclidean distanced signals with corresponding cost matrix (Bottom Left and Right) Furlanello, C (2006).

Although not specifically noted in the algorithm of the Dynamic Time Warping algorithm, the principles in place that allow the DTW algorithm to measure the distance between data points, and correlate the output of the cost matrix, can also use the same mathematical principles of the Euclidean Distance equation to give a metric to the amount of alignment needed for two points. As the principles of the DTW find the closest alignment needed for two datasets. Because the DTW method provides a single float value for the cost index alignment, derived from the cost matrix at the $D_{N,M}$ datapoint, the alignment float value can be used to determine how similar two datasets are to each other. As the process of aligning data that is similar decreases the values of the cost matrix overall, meaning that you now have a comparator algorithm. Because the dataset can be used in many forms, the introduction of large multi-dimensional data can be used for each of the algorithm's inputs. If the algorithms reference data was an array of words, and the input was also an array of words, the cost index alignment float value would dictate what data in the array was similar to the reference. This form of comparison could then be used to match specific words in a reference matrix.

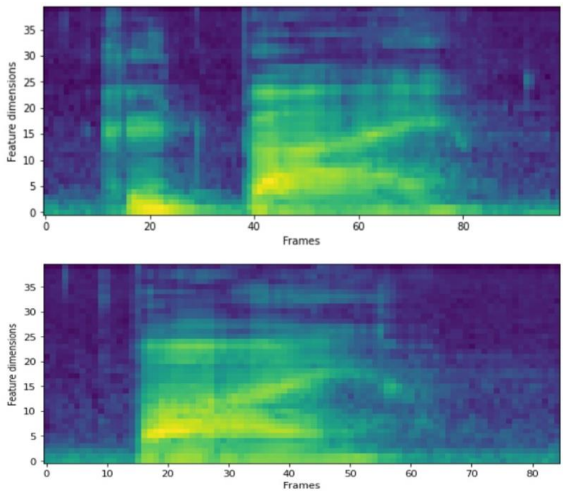

Figure 6 – Input signal "Hello" speech (Top) Reference signal "Hello" speech (Bottom) .Self (2023)

The example shown in Figure 6 shows two different variances of the word "Hello", where each audio recording has been shown in the form of a spectrogram. A spectrogram is a form of image that allows for time and frequency of a signal to be displayed at the same time, where frequency is determined by the height of the points in the image from 0, commonly found on the y axis, and the time of the image shown in the x axis, and finally the signals amplitude is shown as the colours each point within the image. Now that the algorithm has both sets of data, the input signal and the reference signal, the process of calculating the alignment can be done. Using the same process as mentioned in Example 6, we can create a cost index from attributing the input signal as $x_{1:N}$ and the reference signal as $y_{1:N}$.

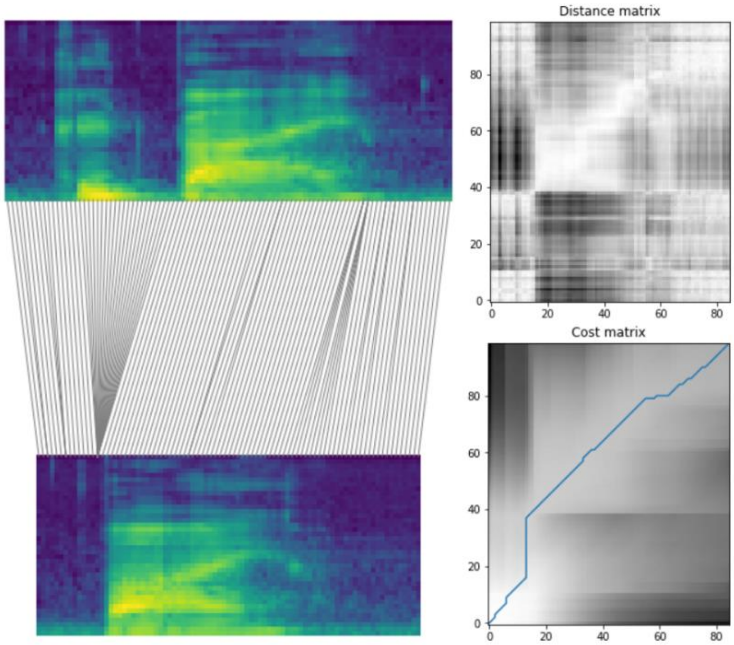

Figure 7 – Input and Reference signals aligned with the DTW Algorithm. Time alignment shown in physical form (Left) Distance Matrix (Top Right) Cost Matrix (Bottom Right) .Self (2023)

Once the cost index has been created, and the value derived from the $D_{N,M}$ datapoint has been taken, the float value can now be compared to other values to see if the alignment is similar, meaning that there is a match in similarity between two signals. The value with the lowest alignment cost at the datapoint $D_{N,M}$ will be the reference signal that is most similar. This technology can then be used to analyse input signals of speech and correlate them with a database of thousands of reference words, thus creating a form of speech recognition. With the algorithm being very simple in nature, the ability to process datasets with limited hardware capabilities.

2.1.3 Hidden Markov Model (HMM) 1980

A Hidden Markov Model (HMM) is a statistical model that is used to describe the probability of a sequence. The sequence itself may have datapoints that are observed, and others that are hidden. The term observable datapoints can be derived from any state or data that can be measured, a simplified example of this can be the transition of weather from one state to another, where at any point in time one weather state may change from rain to no rain. Hidden datapoints are described as events that are not directly observable, but the overall HMM assumes these datapoints are needed to create the observable datapoints. Keeping to the simplified example of the weather that changes from one state to another, the assumption of hidden data is responsible for the observed changes, through either air pressure changes or wind changes, something that in the classification of the example, are datapoints that are not visible. Przemyslaw Dymarski (2011). This means that each model is defined by state probability, transition probability, emission probability and initial probabilities, so in order to define the HMM, the following five elements have to be defined.

1. The N states of the Model, defined by
$$S = \{S_1, S_2, \dots, S_N\}$$
2. The M observation symbols per state $V = \{v_1, v_2, \dots, v_M\}$. If the observations are continuous then M is infinite.
3. The state transition probability distribution $A = \{a_{ij}\}$, where a_{ij} is the probability that the state at time $t+1$ is S_j, is given when the state at time t is S_i. The structure of this stochastic matrix defines the connection structure of the model. If a coefficient a_{ij} is zero, it will remain zero even through the training process, so there will never be a transition from state S_i to $S_j \cdot\ a_{ij} = p\{q_{t+1} = j | q_t = i\}, 1 \le i, j \le N$. Where q_t denotes the current state. The transition probabilities should satisfy the normal stochastic constraints, $a_{ij} \ge 0,\ 1 \le i, j \le N$ and $\sum_{j=0}^{N} a_{ij} = 1,\ 1 \le i \le N$.
4. The Observation symbol probability distribution in each state, $B = \{b_j(k)\}$ where $b_j(k)$ is the probability that symbol v_k is emitted in state S_j.
$$b_j(k) = p\{o_t = v_k | q_t = j\}, \qquad 1 \le j \le N, \qquad 1 \ge k \ge M$$
Where v_k denotes the k^{th} observation symbol in the alphabet, and o_t the current parameter vector. The following stochastic constraints must be satisfied:
$b_j(k) \ge 0,\ 1 \le j \le N,\ 1 \le k \le M$ and $\sum_{k=1}^{M} b_j(k) = 1,\ 1 \le j \le N$
If the observations are continuous, then we will have to use a continuous probability density function, instead of a set of discrete probabilities. In this case we specify the parameters of the probability density function. Usually, the probability density is approximated by a weighted sum of M Gaussian distributions N,
$$b_j(o_t) = \sum_{m=1}^{M} c_{jm} N(\mu_{jm}, \Sigma_{jm}, o_t)$$
Where $c_{jm} =$ weighting coefficients, $\mu_{jm} = mean\ vectors$, and $\Sigma_{jm} = covariance\ metrices$. c_{jm} should also satisfy the stochastic assumptions $c_{jm} \ge 0$, $1 \le j \le N,\ 1 \le m \le M$ and $\sum_{m=1}^{M} c_{jm} = 1,\ 1 \le j \le N$.
5. The Hidden Markov Model is the initial state distribution $\pi = \{\pi_i\}$, where π_i is the probability that the model is in state S_i at the time $t = 0$ with:
$$\pi_i = p\{q_1 = i\}\ and\ 1 \le i \le N$$

Figure 8 – Mathematical definition of the Hidden Markov Model. Przemyslaw Dymarski (2011).

The Hidden Markov Model (HMM) was first introduced by Leonard E. Baum, a mathematician and statistician who worked with IBM in his early career at IBM Thomas J. Watson Research Centre. In the 1972 paper "Statistical Inference for Probabilistic Functions of Finite State Markov Chains" Baum first theorised the use of 'Markov Chains' to be used for statistical purposes, theoretically outlining the use of Hidden Markov Models for statistical analysis. In simple terms the HMM is used to statistically predict the next set of data based on the previous data. An example of how this works is with the analogy of the weather changing previously mentioned, if you were able to take the hidden data at any one point in time, you could create a statistical presumption upon what the next set of data would be. nipunbatra (2018).

To do this in theory, the introduction of a Markov Chain needs to be created, outlining the parameters needed to change observation states. The Markov Chain assumes that all observations x_{t+1} at the future point in time denoted as $t+1$ is dependent on the observation x_t at the time t.Put simply, after the model has been given the present observation x_t, the future points $t+1$ are independent of the x_t observations.

Where $P(x_{t+1}|x_1, x_2, \dots, x_t) = P(x_{t+1}|x_t)$.

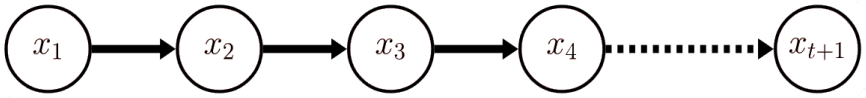

Figure 9 – Markov Chain Example. nipunbatra (2018).

To calculate the joint probability of the sequence the rules of independence need to be considered, with the sequence of Figure 9 – Markov Chain Example. The mathematical notation for the probability is: $P(x_1, x_2, \dots, x_{t+1}) = P(x_1)P(x_2|x_1)P(x_3|x_2) \dots P(x_t|x_{t+1})P(x_{t+1}|x_t)$.

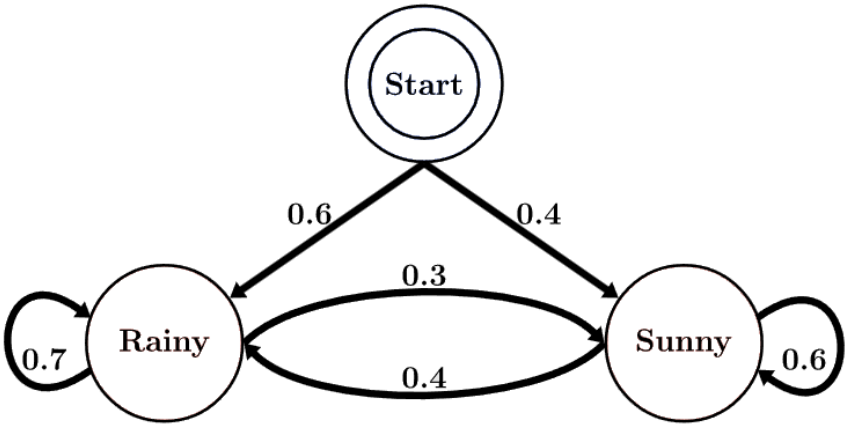

Figure 10 – Markov Chain for Weather. nipunbatra (2018).

Observation x_t at time t can take discrete values or K states. Factorisation of Figure 10 – Markov Chain for Weather. Generalised Markov Chain.

$$P(x_1, x_2, \dots, x_T) = P(x_1) \prod_{t=2}^{T} P(x_t|x_{t-1}) \tag{3}$$

Markov chains assumption that the conditional probability $P(x_t|x_{t-1})$ does not change with time. The transitional matrix (A): stores the probability of transition between one state i to another state defined as j. Transitional matrix represented as $K \times K$ matrix, where the entry A_{ij} is given by $A_{ij} = P(x_t = j \mid x_{t-1} = i)$ where $i, j \in \{1, 2, \dots, K\}$. Prior probability (π) starting from one available stare is denoted as $\pi_i = P(x_1 = i)$ where $i \in \{1, 2, \dots, K\}$.

Now that the model's architecture is defined, the Hidden Markov Model can now be used in speech recognition. Because the model uses the current state to predict the future state, we can use the input of speech and decode the speech itself into its subsequent phonemes. Phonemes are around 10-20 milliseconds of speech that define the characteristics of a spoken word. For example, if you have the spoken word "Hello" the phonemes for this word would be 'h, eh, l, ow', adapting the 4 phonemes together we can determine the word "hello". The way the HMM defines these phonemes can be shown in Figure 11.

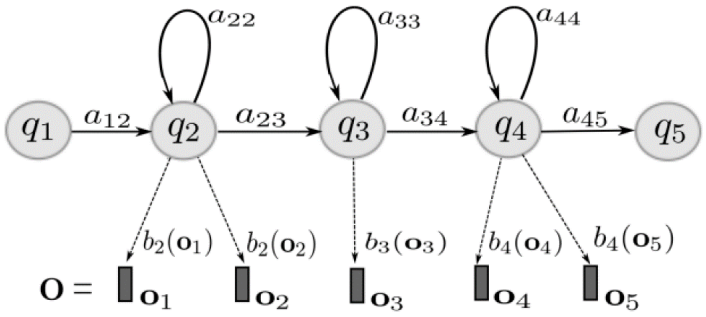

Figure 11 – Process chain for HMM and speech recognition. Swietojanski, Pawel. (2016).

In the example shown in Figure 11, we can observe the HMM chain of the word "Cup" where each of the HMM chains noted as $\{q_1, q_2, \dots, q_5\}$ represents the phonemes generated from the observations $\{b_2(o_1), b_2(o_2), \dots, b_4(o_5)\}$ so $q_2 = /c/$, $q_2 = /uh/$,$q_2 = /p/$. The acoustic properties of the input signal speech are represented by its acoustic frames $\{o_1, o_2, \dots, o_5\}$. An acoustic frame is a segment of the input speech that has features that corrospond to speech, although not outlined in this section, the use of feature extraction is used to determine where and when the important data is within the input speech.

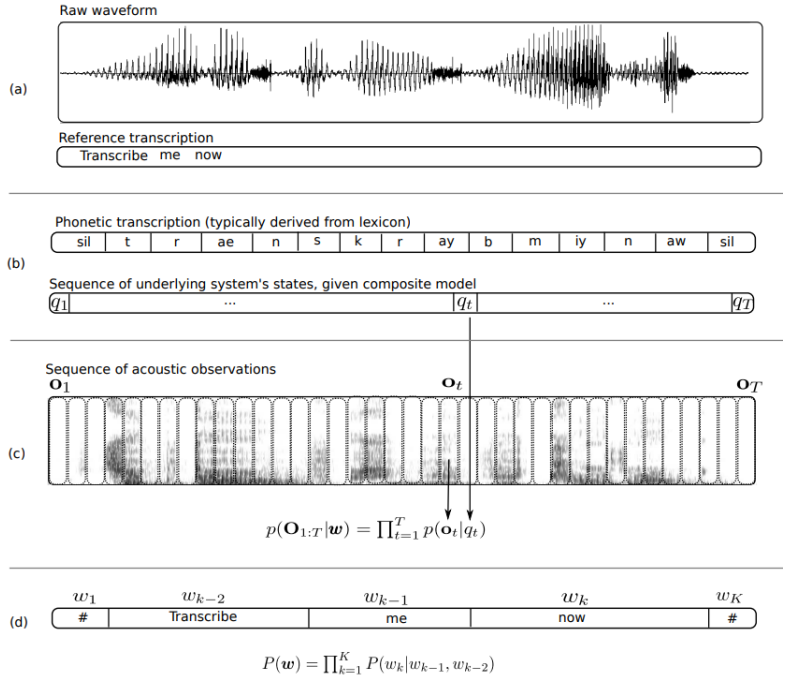

Figure 12 – Frame Extraction for speech recognition. Swietojanski, Pawel. (2016).

In Figure 12 we can see the full process of the Hidden Markov Model and the frame extraction of the original input. Where (a) shows an example of a raw waveform with the internal data being speech composed of the following sentence "Transcribe me now". Furthermore (b) and (c) are both linked to the HMM, where (b) is the layer defined in Figure 12, as $\{q_1, q_2, \dots, q_n\}$, and (c) is notated as $\{o_1, o_2, \dots, o_5\}$.

The equation notating this behaviour is defined as:

$$\mathrm{p}(\mathrm{O}_{1:T}|\mathrm{w}) = \prod_{\mathrm{t=1}}^{\mathrm{T}} \mathrm{p}(o_t|q_t) \tag{4}$$

And (d) is the finalised output of the process and is defined by the following equation:

$$P(w) = \prod_{k=1}^{K} P(w_k|w_{k-1}, w_{k-2}) \tag{5}$$

2.1.4 Neural Networks (NN) 1990

Put simply, a neural network is a method that does very simple calculations, although the calculations may seem simple, the overall complexity of the 'system' referred to the internal workings of the neural network, can be a magnitude of between a simple one layer three node model, to models that have billions of nodes and tens of thousands of layers. Furthermore, these systems are used to create a vector graph that encompasses the input data together as much as possible, this is an extremely simplified description of the workings of neural networks, but the classification itself does not change, no matter what level of detail. So how do they work? Neural Networks work by taking in data, processing the data, and outputting results.

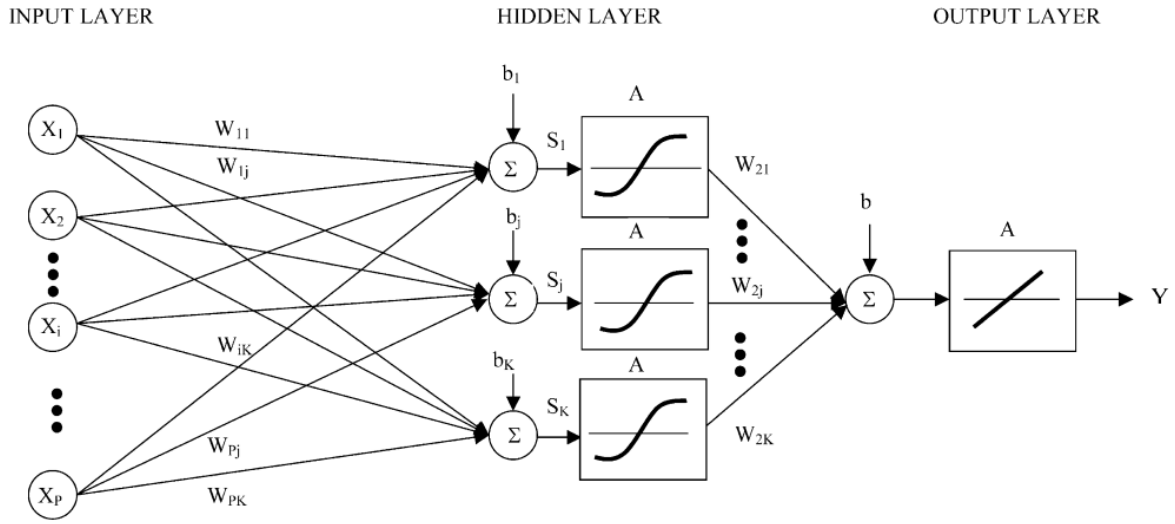

Figure 13 – Basic Neural Network. Vuckovic (2015)

Inputs = $\{X_1, X_2, \dots, X_p\}$ Parameter weights = $\{W_{11}, W_{1j}, \dots, W_{PK}\}, \{W_{21}, W_{2j}, \dots, W_{2K}\}$, Hidden layer nodes = $\{b_1, b_j, \dots, b_k\}$ sigmoid function $\{S_1, S_j, \dots, S_k : A\} = \sigma(x) = \frac{1}{1+e^{-x}}$ Output Node = b.

For example, you need to build a neural network to understand letters of the alphabet, firstly you would need to construct the network, for simplicity the area that the letters are written is composed of a $[32, 32]$ array of pixels, this works out to be 1024 pixels in total. So, for the purpose of this example, we will define the input nodes of the network and have 1024 input nodes, where each node is represented as a pixel. Every node in the network including the hidden layers, input nodes and output nodes can only store a float value between 0 and 1. With this in mind, we can define the next properties of the network with its hidden layers. There is no mathematical basis or outline for the number of layers and nodes needed for the hidden layers of a system, as this is an area that is mostly changed and optimised for the purpose of what the network is built for, in this example the network will host 2 hidden layers of 512 nodes. Finally, the theoretical network will need an output stage of 10 nodes, one for every number of the alphabet, these are $[0, 1, 2, . . ,9]$.

Now the theoretical network has been constructed, the connection to the nodes needs to be classified and defined, as these connections have data that define its use within the network itself. Keeping to the example of the theoretical network we have node X_1 connected to b_j via the link W_{1j}. The link connecting these two nodes together has 2 pieces of data attributed to it, one being the overall weight of the connection, this is defined as a float number which can be a negative number or a positive number, secondly the last piece of data is called the *'bias'* and this is also a number similar to the last mentioned. This means that W_{1j} could contain the following data $[-2.32\ 'weight', 0.65\ 'bias']$.

When an input is sent through this theoretical network, we will find that the network will try and sum all the weights and biases from all connections at $\{b_1, b_j, \dots, b_k\}$, and because the node connected to these weights and biases can only hold a value between 0 and 1, the system will not work correctly. To stop this from happening the network implies a function to the summation of all the weights and biases, this function can be a common sigmoid function denoted as $\sigma(x) = \frac{1}{1+e^{-x}}$ in *Figure 14*. Where $r^x = [0, 1]$.

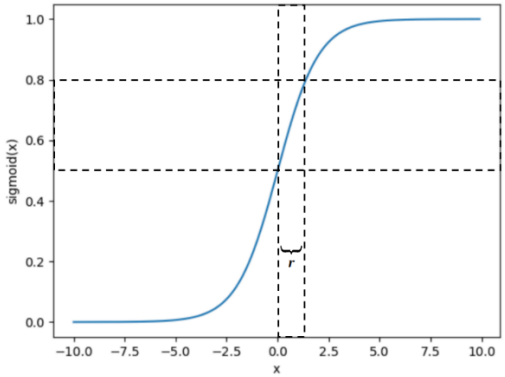

Figure 14 – Basic Neural Network where r is the area within the curve 0 to 1. Vuckovic (2015)

The purpose of this function is to combine all the parameters values together so that the output to the node will be a value between the 0 and 1 limitations. This method allows for a uniform approximation of the level of parameters at any given node denoted as A in *Figure 13*.

The summation of these values can be shown by the following:

$$\sigma(W_{11}b_1 + W_{1j}b_j + \cdots + W_{PK}b_k - bias) \tag{6}$$

Where $bias = \{h_1, h_2, .., h_n\}$

Although this equation only represents one single node and its links to previous nodes, the implementation of the $-bias$ allows the network to determine at what point will the summation nodes $\{b_1, b_j, \dots, b_k\}$ be activated. To visualise the entire system together with respect to weights, bias and summation nodes, the following expression can be used.

$$\sigma\left(\begin{bmatrix} W_{11} & W_{21} & \cdots & W_{n1} \\ \vdots & \vdots & \ddots & \vdots \\ W_n & W_n & & W_{nK} \end{bmatrix}\begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix} + \begin{bmatrix} h_1 \\ \vdots \\ h_n \end{bmatrix}\right) \tag{7}$$

$$b_{Sn}^{n} = \sigma(W b_n + h)$$

Where: Inputs = $\{X_1, X_2, \dots, X_p\}$ Parameter weights = $\{W_{11}, W_{1j}, \dots, W_{PK}\}, \{W_{21}, W_{2j}, \dots, W_{2K}\}$, Hidden nodes = $\{b_1, b_j, \dots, b_k\}$ Sigmoid function $\{S_1, S_j, \dots, S_k : A\} = \sigma(x) = \frac{1}{1+e^{-x}}$ Output Node = b.

Now that the foundation of the neural network has been created, the implementation of a speech recognition method needs to be added, this is done by allowing the input stage of the network to be an array of phonemes in succession. In 2.1.3 the method of splitting speech into frames and processing them is introduced. A simplistic way of understanding how this would work with the neural network, is by presuming a dataset that hosts all possible word phonemes of any given language. In this example the assumption is made that the input data is English. Now the neural network has a system in place that can understand the phonemes as inputs, the addition of training data can increase the accuracy of the output of the network.

How does the neural network understand what is correct or incorrect? It does this by adding a process called 'Back-Propagation'. Back-Propagation is a way of giving the neural network the ability to change its own weights and parameters in line with the output of the network, the process starts at the output of the network and works back through the network all the way to the input, through all the hidden layers and its subsequent nodes. This

very complex process can be simplified by implying the use of a cost function, this function is a way of ranking if the output of the neural network is the same as the training data, this data will already be referenced with an input and required output. The cost function will rank the output of the network with a float value between 0 and 1. If the cost function is lower than the desired amount needed to determine if the output is in fact the same as the input reference, then the cost function will change the weights of the nodes starting from the output. The cost function will change the weights connecting to the nodes by a specific amount determined by a metric called 'step-size' a self-explanatory variable that is referenced when creating the network, that determines the amount of change requested at a minimum for any given value change of the weights.

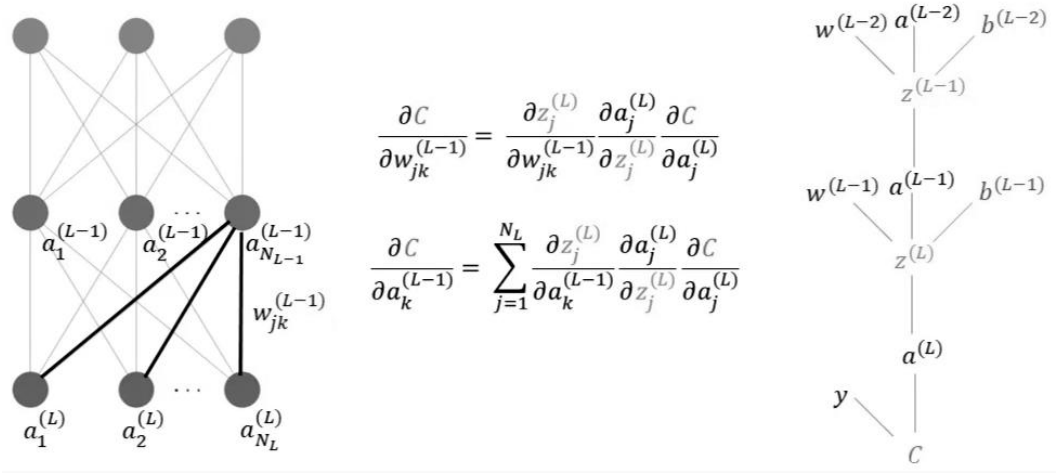

Figure 15 – Simplified mathematics for back propagation. Silva, S.D. (2020).

Where: $a_{N_L}^{(L):(L-N)}$ denotes the node, and its place in the array. $w_{jk}^{(L-1)}$ is the connection to node j up to the number of nodes at layer (L): $(L-N)$

$$\sigma\left(\begin{bmatrix} w_{a_1^1} & w_{a_2^1} & \cdots & w_{a_{N_L}^1} \\ \vdots & \vdots & \ddots & \vdots \\ w_{a_1^n} & w_{a_2^n} & & w_{a_{N_N}^1} \end{bmatrix}\begin{bmatrix} a_1^1 \\ \vdots \\ a_{N_L}^{(L):(L-N)} \end{bmatrix} + \begin{bmatrix} \frac{\delta C}{\delta w_{jk}^{(L-1)}} = \frac{\delta z_j^{(L)}}{\delta w_{jk}^{(L-1)}} \frac{\delta a_j^{(L)}}{\delta z_j^{(L)}} \frac{\delta C}{\delta a_j^{(L)}} \\ \vdots \\ \frac{\delta C}{\delta a_k^{(L-1)}} = \sum_{j=1}^{N_L} \frac{\delta z_j^{(L)}}{\delta a_k^{(L-1)}} \frac{\delta a_j^{(L)}}{\delta z_j^{(L)}} \frac{\delta C}{\delta a_j^{(L)}} \end{bmatrix}\right) \quad (8)$$

Although the final building blocks of the neural network have been created, there has been an assumption that the networks connection weights $w_{jk}^{(L)}$ have been pre initialised. Although most neural networks assume this presumption to its architecture, the initialisation of the weights is important to the training of the network's nodes. Using the function mentioned in Figure 15, initialising the networks weights is done using a modified version of the back propagation algorithm. As a common ruling, the weights of the network are initially set to a random number derived from the following expression, where each weight and bias is set at random. Xavior Glorot (2010) uses a stepped approach to this task by first initialising the weights from gaussian or uniform distribution, then scaling the weights proportional to the number of inputs to the layer. Firstly, the expression to scale the weights proportional to the number of inputs of the layer can be shown as the following.

$$W^{(l)} \coloneqq W^{(l)} \cdot \sqrt{\frac{1}{m^{(l-1)}}} \quad (9)$$

An example of this would look something like this:

$$W_{i,j}^{(l)} \sim N(\mu = 0, \sigma^2 = 0.01) \quad (10)$$

Gaussian weight distribution: $N(\mu = 0)$ Features of the layer = m

3.0 Methodology & Research Design

The following research was conducted in conjunction to Radford (2022) and Silva, S.D. (2020). The overarching design of the research is to create a test environment that is capable of fine tuning a pre-trained neural network with a dataset unfamiliar to the network itself. Radford (2022) outlines training a neural network for the purpose of transcribing and translating speech to text within the boundaries of the languages within the training data, in the case of Radford (2022) the network named 'Whisper' is able to create transcriptions and translations to English, for up to 52 languages. The nature of the research carried out by Radford (2022) allows for the network to be retrained in a process called tuning, with all of the documentation and processes used to create and train the original model, being set to 'MIT Licence - Collister, L. (2020). Defines this licence as the following "Users of software using an MIT License are permitted to use, copy, modify, merge publish, distribute, sublicense and sell copies of the software." This licence allows for all the networks weights and parameters to be accessible by the public domain, allowing the use of these models in fine tuning. Silva. S.D. (2020) defines a basic framework for the use of traditional training data and its effects on neural networks with respect to training with data it knows and does not know. A combination of method in the research of the two papers, allows a greater understanding of the overarching principles and procedures in the research carried out. The research design of the paper was constructed on the understanding and foundational knowledge of Radford (2022) and Silva, S.D. (2020).

3.1 Datasets –

The datasets were provided by Mozilla and their '*Common Voice´* corpus. The common voice datasets are community driven speech datasets that can offer short and longform speech and corresponding transcriptions. The datasets are collected sequentially, meaning that dataset 13.0 will host the data provided in corpus (11.0 and 6.0). The datasets provide all audio in '*.mp3'* format, and all relevant transcriptions in *'.tsv'* formats. Files provided outside of the audio dataset include the following (Validated, Invalidated, Training and Other). The Validated dataset is a confirmed dataset listing of all files and corresponding transcriptions. Invalidated files correspond to files that have no transcription linked to the audio files listed in the file. Training is a mix of both data (validated and invalidated). And Other corresponds to the various unrecognised audio files that were not validated correctly when submitted to the online corpus at the time of the dataset.

The following table shows the datasets used for both training and testing of the language models.

Table 1 – Datasets and corresponding information. Self (2023)

Dataset	Audio Files	Validated	Invalidated	Training	Other
Common Voice 13	1,573,386	1,013,968	264,713	1,013,968	278,333
Common Voice 11	1,508,536	948,736	252,600	948,736	290,846
Common Voice 5.1	793,876	435,947	166,816	435,947	175,084

3.1.1 Model Selection/Development & Justification –

There were three independent speech recognition models used within this paper.

Table 2 – Model Information. Self (2023)

Model Name	Publisher	Model Definition	Parameters	Sub Models Used
WHISPER	OpenAI	Transcribe + Translate	Small – 241,734,912 Base – 72,593,920 Tiny – 37,760,640	Small Base Tiny
Deepspeech	Mozilla	Transcribe	800,000,000	Deepspeech 0.9.3
Tuned LMT2	Self	Transcribe + Translate	Small – 241,734,912 Base – 72,593,920 Tiny – 37,760,640	Small x 10 CP Base x 10 CP Tiny x 10 CP

The automatic speech recognition models used in the development of this paper are '*Whisper*' and 'Deepspeech'. These two models utilise neural networks to achieve their proposed functions. Deepspeech 0.9.3 was created by the Mozilla foundation in 2017, where it is currently updated and maintained through the collaboration website Github. Mozilla (2019). Whisper was created by OpenAI in 2022. Radford (2022). Where it too is updated and maintained on Github. Github (2022). Lastly, the LMT2 model was created by utilising the opensource nature of Whispers dataset and models, where the selected models shown in Table 2 are (Small, Base, Tiny). These models were 'fine-tuned' to a new dataset prior to testing. The term 'fine-tuned' refers to the process of re-training the existing model on new or improved data that is has never processed before. A common ASR technology that Panopto utilises for its video platform automatic speech transcription is 'Deepspeech', thus the use of this model was needed to conduct the conjecture of the paper. All the models used in the testing of the conjecture were based on a number of programming languages, most notably the 'Python' language and the 'Rust' Language. Python is a general-purpose coding language that is dynamically typed and interpreted, thus the code written is not compiled into machine code before deployment, this means that the language is inherently slower with performance. Python Software Foundation (2019). The Rust programming language is contrasting to python due to its memory safety standards and performance benefits of the language being a compiled language. Github (2022). Meaning that the code that is deployed will be converted to machine code for better processing and performance. The models proposed in Table 1 were developed for the sole purpose of both transcribing speech audio to text, and in some cases such as Whisper and the subsequent LMT2 models, will also host an array of functions that are able to both transcribe and translate to the English language. Radford (2022).

3.1.2 Model Training –

The prerequisites that were needed for this paper varied between libraries and drivers. The primary libraries include 'transformers', 'datasets', 'huggingface' and 'torch'. The use of all these libraries depended on a software suite 'CUDA', this package was used to allow 'torch' the capability of using the graphics processor to increase the speed of the overall system. The graphics processor and details can be found in the table 3 below.

Table 3 – GPU Model Information. Self (2023)

Producer	Name	Model	Variant	Memory	CUDA Cores	Tensor Cores
Nvidia	RTX	3090	FE	24GB	10,496	328

The datasets from common voice (common voice 13, common voice 11, common voice 5.1) were downloaded and extracted to the following specifications shown in table 4 below.

Table 4 – Dataset File Size. Self (2023)

Dataset	Total Audio Files	Audio Delta	Download Size MB	Extracted Size MB
Common Voice 13	1,573,386	64,850	76,390	89,570
Common Voice 11	1,508,536	714,660	74,270	82,480
Common Voice 5.1	793,876	-	50,060	68,050

The training code created for the use of training the pre-existing model (Whisper – Tiny, Base, Small) was constructed in a way that both included the install and management of all libraries and utilities needed for the process, and the management of the systems files. The code itself iterated through a simple process. Firstly, check if the location of the environment has enough disk space to process the dataset and train the model, then continue to resource management and utility package downloads, this specific area of the code made sure that any packages linked to the function of the code was preinstalled before processing of the datasets stated. Secondly, the code would connect to 'huggingface' via the 'requests' library in python, so that the datasets used to train the models can be downloaded, extracted, and serialised into subsequent folders linked to (validated, invalidated, training and other). Then the code would begin by processing the audio.

One of the key libraries that was used in the next stages of the code was the use of a library called 'WAV2VEC' from the publishers 'Facebook Researchers'. research.facebook.com. (2020). This software package was able to

encode each audio file and extract the 'key features' of the audio and store them in a vector library, for later use in training the model. The output of this stage is considered the most processor intense as the use of GPU (Graphics Processing Unit) acceleration is not suited for this processing application. The full process of the code can be shown in figure 16 below.

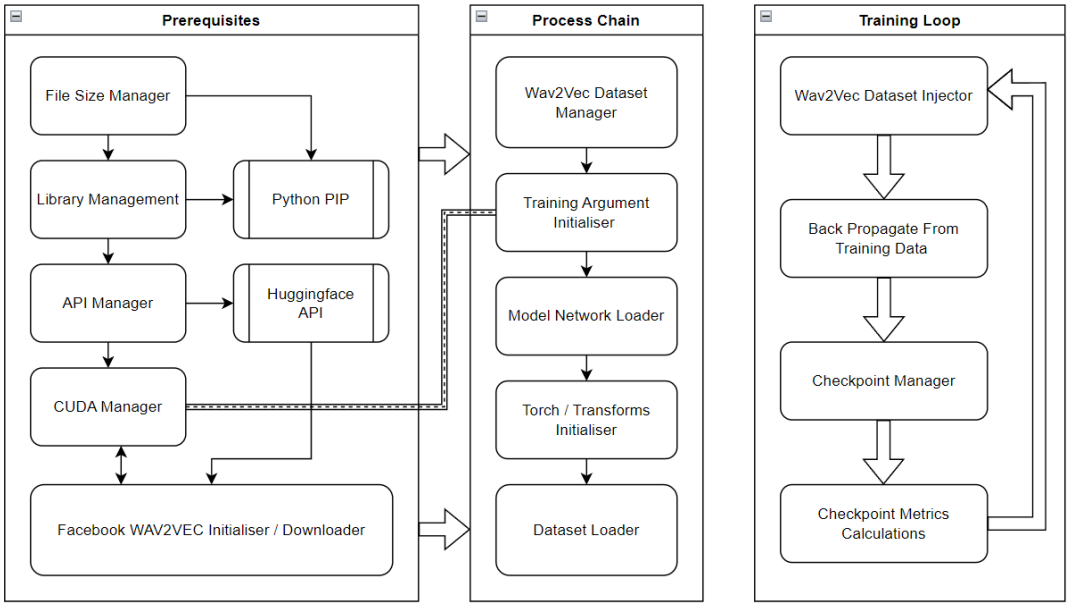

Figure 16 – Process Chain of Tuning Code. Self (2023)

3.1.3 Model Testing –

The use of key metrics that calculate the accuracy of the data is key to classifying the model's intrinsic accuracy when transcribing speech to text. The metrics used to determine the data accuracy and subsequent data about the metrics can be found in Table 5.

Table 5 – Key Metrics Identity and Calculations. Self (2023)

Key Metric	Values	Identity Description Calculations
WOR – Word Error Rate	0-100%	The total error of *'UTF-8'* characters between spaces in an array matched with a reference array. Calculated by comparing the array per space to the reference. $WER = \frac{Total\ Letters}{Correct\ Letters}$ (11)
SER – Sentence Error Rate	0-100%	The total error of *'UTF-8'* characters of the entire array in respect to the reference array. Calculated by comparing the total words of the array with the output of the array. $SER = \frac{Total\ Words}{Correct\ Words}$ (12)
CER – Character Error Rate	0-100%	The total error of *'UTF-8'* characters with respect to the reference characters of the array. By comparing each character to the position of that character in the array to the reference array. $SER = \frac{Total\ Characters}{Total\ Correct\ Character}$ (13)

Table 6 – Key Metrics Code Calculations. Self (2023)

Metric	Code

WOR	`def calculate_wer(reference, hypothesis):`
	`  # Calculate the Word Error Rate`
	`  wer = edit_distance(reference.split(), hypothesis.split()) / len(reference.split())`
	`  return wer`

SER	`def calculate_sentence_error_rate(reference, hypothesis):`
	`  # Calculate Sentence Error Rate`
	`  ref_set = set(reference.split())`
	`  hyp_set = set(hypothesis.split())`
	`  ser = edit_distance(ref_set(),hyp_set()) / len(reference.split())`
	`  return ser`

CER	`def calculate_cer(reference, hypothesis):`
	`  # Calculate the Character Error Rate`
	`  cer = edit_distance(reference, hypothesis) / len(reference)`
	`  return cer`

The code that is listed in Table 6 outlines the use of a reference and a hypothesis, where the reference is the validated dataset, and the hypothesis is the results returned from the model's output. The use of a reference dataset to match future data to, is one of the most important classifications of data used when training a model, especially with a model that uses non-standardised objects from a pool of community contributors. Mozilla (2019) outlines this issue within its terms and conditions with regards to the common voice datasets. The issue that was faced in respect to this was the nonstandard input of the reference data provided from the 'validated' dataset of the common voice corpus. The reference dataset text reference itself cannot be verified by both internal and external examination, only the spoken input into the dataset can be matched. This causes a situation where special characters can be injected into the dataset causing discrepancy within the model's metric calculations and subsequent training and tuning of said models. Radford (2020) outlines a 'text standardisation' section that provides a brief reshaping of the reference data in relation to the common voice corpus used to train the initial model. Although the accuracy of the standardisation outlined in the paper suggests a robust approach to the issues of the non-standard text issue, the issue of the English language having both 'English' and 'American' classified spelling is outlined. This presumes that all input to the common voice corpus is set to 'American' classified spelling, which through testing shown in table 7 outlines the differences stated in both the reference input and the classified output of the 'Whisper – Large' model.

Table 7 – Non regular outputs from Whisper-Large (Language) . Self (2023)

Model Name	Validated File	Validated Transcript	Whisper Output	Metrics
Whisper Large	common_voice 20836413.mp3 Common-Voice Corpus 13.0	The intention was to help to **standardise** prices among locations.	The intention was to help to **standardize** prices among locations.	WER = 10.00% CER = 1.56%

The output from the Whisper model matches the context and spelling of all other words in the reference data. But changes the 's' to a 'z' to conform to the classified American spelling. This standardisation causes a situation where although the transcript is almost 100% accurate, the input of said data causes the WER and CER errors to increase from the expected 0%. The architecture of the process chain of the tuning code outlines the use of 'checkpoints', these are another word for the term snapshots, where the environment and current progress of the dataset and models are saved into its own version. To put simply, the code makes a duplicate of itself at the point where a checkpoint is being rendered, allowing the user to test the model and dataset at that point in time with reference to what state the models neural network was set to. This type of historic callback allows for better technical analysis of the data and the models, due to the stepped approach to the history of the model and being able to test at what points the model may or may have not failed or had any loss in performance. Radford (2020) outlines in the paper the use of checkpoints and how these can be used to both troubleshoot and optimise the current dataset or model, as understanding the current state towards what has happened in the chain of events, similar to how HMM's can create situations where the present and previous inputs can be used to show a probability for future events.

3.1.4 Model Evaluation Metrics –

With all the checkpoints and datasets collected for testing the total amount of independent models that needed to be tested was 34 separate individual models with various differences in performance and neural network weights. The breakdown of these models can be shown in the table below.

Table 8 – All Model Details. Self (2023)

Model Name	Deepspeech	Whisper	Whisper	Whisper						
Model Variant	9.3.0	Small	Base	Tiny						
Model CP	N/A	N/A	N/A	N/A						
Number	1	2	3	4						
Model Name	LMT2	LMT2	LMT2	LMT2	LMT2	LMT2	LMT2	LMT2	LMT2	LMT2
Model Variant	Small	Small	Small	Small	Small	Small	Small	Small	Small	Small
Model CP	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Number	5	6	7	8	9	10	11	12	13	14
Model Name	LMT2	LMT2	LMT2	LMT2	LMT2	LMT2	LMT2	LMT2	LMT2	LMT2
Model Variant	Base	Base	Base	Base	Base	Base	Base	Base	Base	Base
Model CP	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Number	15	16	17	18	19	20	21	22	23	24
Model Name	LMT2	LMT2	LMT2	LMT2	LMT2	LMT2	LMT2	LMT2	LMT2	LMT2
Model Variant	Tiny	Tiny	Tiny	Tiny	Tiny	Tiny	Tiny	Tiny	Tiny	Tiny
Model CP	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Number	25	26	27	28	29	30	31	32	33	34

The models themselves all need to be standardised for the process of calculating the metrics. This includes isolating each checkpoint in its own sub directory for later use with an automated tool. The tool in question is an automatic script that runs through the sub directories of all the models, and both runs the model on a set corpus of speech data defined in the 'other' classification of audio within the common voice corpus used. And finally, once all transcriptions have taken place, the system will run a subroutine that calculates all the metrics for all models and checkpoints sequentially, saving each result in the sub directories of all relevant models. The structure and process of the script and architecture of the test environment can be shown with a set of simple block diagrams shown below. The reason the automated script was created was to limit error that could be caused by logging information throughout the 34 various models, as the tuning stage of the process already allocates the apposed sub directories in the data structure of the test environment. This simplifies the process to train and calculate the performance of the models in a batch scenario.

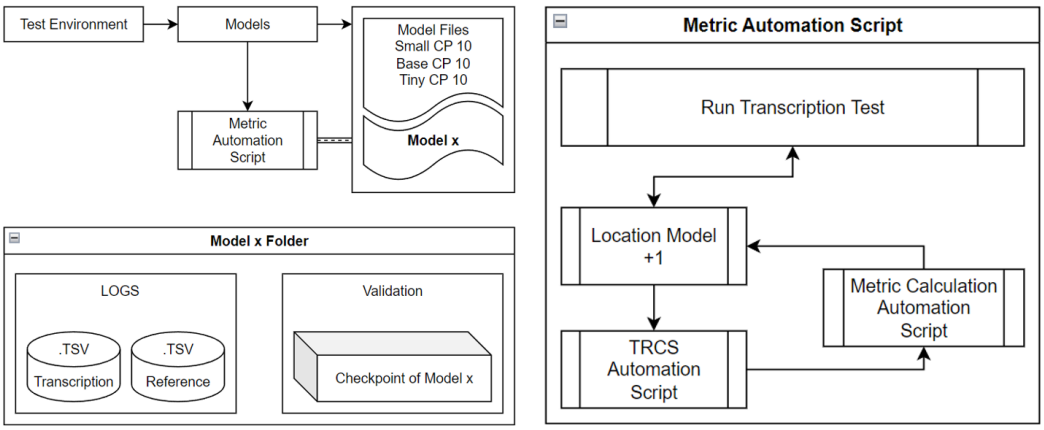

Figure 17 – Architecture of the Test Environment (left) Architecture of the Test Loop (Right). Self (2023)

From the standard format of the test environment, the subdirectory to this folder is the 'Models' folder, and the specified automation script. Within the model's folder are the 34 separate sub folders for each model used in the tests. Furthermore, the notation of 'x' can be derived from the model number within the sub directory in question, meaning that within every sub directory there will be a further 2 sub directories, classified as 'Logs' and 'Validation'. The Logs directory houses both the reference transcription as a '.tsv' file, as well as the subsequent transcription file of the same data type. Although the transcription file is not located in the directory

before the automation of the test starts, as this file is created when testing the performance output of the models in question. The validation folder houses all the necessary files needed to construct the model's neural networks and checkpoints.

3.2 Model Testing & Experiments –

During the initial process of calculating the metrics that outline the model's performance, there were a number of key metrics that were not used but are projected in the results from the automated metric calculation script mentioned in 3.1.5. These were 'F1-Score' and 'Word Accuracy'. These metrics were not used in the final conclusions regarding the specific model's performance due to the accuracy of the calculations, the reason for this is that in simple terms, the F1-Score itself uses complex float values that were being consistently rounded up due to the nature of transitioning integer values to float values. Thus, the with primary testing the results determined from this score would fluctuate within a margin of plus or minus 5 percent, far beyond any reasonable representation of accuracy. This representation of the data caused discrepancies throughout the initial testing period of the project and was subsequently removed from the list of active metrics calculated for this paper. Secondly the 'Word Accuracy' was determined by calculating the average error across all states of the metrics, including the F1-Score metric, this too created data that was out of suspected margin of error that was set to zero.

Table 9 – F1 Score and Word Accuracy Code. Self (2023)

Metric	Code

F1-SCORE	`def calculate_f1_score(reference, hypothesis):`
	`  # Calculate F1-Score`
	`  if not reference or not hypothesis:`
	`  return 0.0`
	`  ref_set = set(reference.split())`
	`  hyp_set = set(hypothesis.split())`
	`  f1 = f_measure(ref_set, hyp_set)`
	`  return f1`

Word Accuracy	`def calculate_word_accuracy(reference, hypothesis):`
	`  # Calculate Word Accuracy`
	`  num_correct_words = sum(1 for ref_word, hyp_word in zip(reference.split(), hypothesis.split()) if ref_word == hyp_word)`
	`  word_accuracy = num_correct_words / len(reference.split())`
	`  return word_accuracy`

In computer science the term f-score is determined by the accuracy of a neural network, it does this by considering all measurements done and comparing it to a completed validated dataset, something that should be predetermined outside of the experiment or model itself. In this specification the term f-score is calculated by taking all correct points and dividing this number with the correct number of points in the dataset. Although this may seem like an ideal metric to calculate the model's accuracy in regard to the model's sustained outputs, the classification of the metric conforms too closely with the metrics already in use *'WER, SER, CER'.* Where in the case of determining the accuracy using the f-score would indicate a similar metric to the character error rate metric, where each point in the array would be compared to a validated version of the original dataset. This would mean the metric would oppose a similar result to the character error rate metric.

3.2.1 Tuning Models –

The models that were tuned stated in table 8 outline the final results of the models with respect to the tuning and training process, but this result was formed from countless experiments that was tuned specifically for the task of tuning the original Whisper models. The *'LMT2'* models were all originally trained on one of many specific datasets used to train the original model. Common-voice dataset 5.1 was used in the original process of training the Whisper model's before being tuned further in this paper. With the common voice dataset being a corpus that has data added to the corpus on every iteration, the dataset 'common-voice 11' will have the same dataset as the 'common-voice 5.1' dataset but with the added data at the iteration 11. Put simply, the model cannot just have the latest dataset for further tuning of the model, this is because the very principle of adding data to the tuning process forces the model to try and re-train the data it has already been trained on. This can cause issues

with the model being too harshly tuned with data it is familiar with, a process called ‘Over Fitting’. A term that outlines a model’s ability to readjust its own neural weights and biases in line with data it has already seen, creating a situation where the model’s accuracy for other ‘unknown’ data will result in the model losing performance. Ying, X. (2019) states that overfitting can be described as the following, “Overfitting is a fundamental issue in supervised machine learning which prevents us from perfectly generalizing the models to well fit observed data on training data, as well as unseen data on testing set. Because of the presence of noise, the limited size of training set, and the complexity of classifiers, overfitting happens.” With the following considered, the proposed dataset that would be used to further tune the models would need to have the original dataset removed from the training data. The process of this was done with an automated python script that would remove any references from one dataset to another, removing all datapoints, leaving only the data that was not referenced. The new dataset would then be structured into a format where it could easily be accessed by the tuning processes of the automated python scripts.

Secondly, the initial experiments had an array of various neural network training parameters to both improve effectiveness of the training data, and the ability to speed the training process up with the limited hardware capacity used in the experiment. The performance of the model tuning was a key metric to consider due to both the time constraints and the storage constraints of the experimental phase of the paper, as the model needed to be able to be trained efficiently and fast, while still maintaining the level of precision needed to tune the model effectively. The performance specifications of the hardware in table 3 outlined the singular processing unit used to tune the Whisper models, while still being effective in the process of tuning the models, the limited processing power available, created a situation where steps that could lead to better results could have been used. One of these key metrics is mentioned in 2.1.4, where the key parameters used to train a model and its weights is the step-size. The greater the step size the faster the model can process the training data, but the performance of the model would decrease to a significant amount. While decreasing the step-size would benefit the model’s accuracy to a point before overfitting takes place but would also increase the processing time to tune the model exponentially with the smaller step-sizes used. The code specification for these parameters can be found in the ‘LMT2-Tuning-(Model)’ script, where the following code can be found that outlines these parameters. The parameter that indicates step size is ‘learning_rate’ where this would determine the number of steps for every change in weights through the back propagation stage of the training process expressed in 2.1.4 and outlined in Figure 15. The training arguments used in this code can be linked to many aspects of the training of the model, where ‘max_steps’ indicates the amount of steps needed to complete the training of the model, in conjunction to this parameter, the parameter ‘warmup_steps’ is also an additional parameter to the ‘max_steps’ parameter, as this allows the model to initialise the models weights and biases with the specified number given in the specific argument. These parameters were all altered in many variations of the training code, to allow the hardware the ability to be at an optimised balance between accuracy and speed of the training.

Table 10 – Training Code. Self (2023)

Parameters	Code

Training Arguments	`training_args = Seq2SeqTrainingArguments(`
	`output_dir="./LMT2-Small-10",  # change to a repo name of your choice`
	`per_device_train_batch_size=10,`
	`gradient_accumulation_steps=1,  # increase by 2x for 2x decrease in batch size`
	`learning_rate=1e-6,`
	`warmup_steps=500,`
	`max_steps=10000,`
	`predict_with_generate=True,`
	`save_steps=1000,`
	`eval_steps=1000,`
	`logging_steps=100,`
	`metric_for_best_model="wer",`
	`greater_is_better=False`
	`)`

3.2.2 Data Collection Process –

During the process of training the model the tuning script would print a detailed summary of the current state of the model that is being tuned, and once the specified steps have been completed referenced in Table 10, the script will run an internal test to determine the overall performance of the checkpoint at the exact point in the models training process. The overall performance of the current checkpoint will be shown in a log file saved into the checkpoint directory under the 'Validation' folder, referenced to Figure 17. The details that are logged show the current process of the tuning, and its effects on the model's parameters, shown in the table below is the output of the log from 0 percent to 10 percent in the tuning process. As the model itself has had its metrics for tuning defined by the arguments mentioned in Table 10, the point at which the model creates a checkpoint and subsequent internal test is set to every 1000 steps, thus the evaluation of the checkpoint will be set for every 1000 steps to a max step size of 10,000 meaning that the following log will only show the progress and evaluation of the model tuning from 0 to 10%. The log can show a number of important metrics that can support the conclusions set by optimising the models tuning process outlined in 3.2.1, where the script that runs the processing of the data and automates the tuning process. Loss – This metric is a normalised float value determined by the percentage change of the model at the current point in the tuning process.

For example, if the tuning process log outlined that the model had no loss, this would indicate that the injection of the training data has not changed any of the model's parameters or weights, thus meaning that the model is not being tuned correctly, while having a loss in the context of the log would indicate that the training data is being processed by the model correctly. The overall performance of the loss metric should always show a decreasing value, this would indicate a sustained increase in performance of the model, although as mentioned by Ying (2019) a linear unchanged loss level could indicate model overfitting.

Table 11 – Training Code Output Log. Self (2023)

Steps to 10%	Code Output
	{--}
Training 1%	{'loss': 3.9085, 'learning_rate': 1.92e-07, 'epoch': 0.0 }
Training 2%	{'loss': 2.7621, 'learning_rate': 3.92e-07, 'epoch': 0.0 }
Training 3%	{'loss': 1.4779, 'learning_rate': 5.919999999999999e-07, 'epoch': 0.0 }
Training 4%	{'loss': 0.9408, 'learning_rate': 7.92e-07, 'epoch': 0.0 }
Training 5%	{'loss': 0.8258, 'learning_rate': 9.92e-07, 'epoch': 0.0 }
Training 6%	{'loss': 0.7299, 'learning_rate': 9.898947368421051e-07, 'epoch': 0.01}
Training 7%	{'loss': 0.6606, 'learning_rate': 9.793684210526315e-07, 'epoch': 0.01}
Training 8%	{'loss': 0.6071, 'learning_rate': 9.68842105263158e-07, 'epoch': 0.01}
Training 9%	{'loss': 0.5281, 'learning_rate': 9.583157894736841e-07, 'epoch': 0.01}
Training 10%	{'loss': 0.4687, 'learning_rate': 9.477894736842104e-07, 'epoch': 0.01}
	{--}
-Evaluation-	{'eval_loss': 0.540 # Model Weight Change }
-Evaluation-	{'eval_wer': 24.288 }
-Evaluation-	{'eval_runtime': 5190.101 # Seconds }
-Evaluation-	{'eval_samples_per_second': 3.154 }
-Evaluation-	{'eval_steps_per_second': 0.394 }
-Evaluation-	{'epoch': 0.01 }
	{--}

The evaluation stage of the log shows the internal metrics being calculated for the checkpoint of the model at that point in time, where the code will take 1000 validated samples and test the performance of the model using the metrics outlined in 3.1.4. The log will also print other valuable metrics on performance such how long the evaluation has taken in seconds and how many samples per second the model was able to process during the evaluation process.

3.2.3 Data Collection Instruments and Procedures –

With 34 models that all have specific metrics that need to be compared, the appointment of a singular diagram or graph to express the complexities of the results would be too complex to understand at point value. The models are split into their fundamental versions i.e. (Small, Base, Tiny) with respect to the key metrics that were calculated (WER, SER, CER) with the key metrics being compared per checkpoint to the other model's metrics at any one time within the checkpoint list. The overarching software that was used to display said results was mixed between 'Microsoft Excel', and Pythons internal mapping libraries 'matplotlib'. These software packages were used for both accuracy and ease of understanding basic values in the results, with more basic designs relegated to the excel diagrams and more complex standardised grids and figures expressed in the python package mentioned above. The 'matplotlib' package was used specifically due to its use of technology and data structures similar to the use of 'Matlab' a common software package for data science.

3.2.4 Software Defined Internal Testing –

Internal testing comprised of a metric calculation script that was able to process all the models sequentially and output results based on the performance of the model when injected with 20,000 validated and standardised audio files. The 20,000 audio files will be known as '20K'. The 20K files were specifically used from the dataset of 'Common Voice 13.0' as the original models from Whisper i.e. (Small, Base, Tiny) before fine tuning into checkpoints, would not have been trained on the files used to test, thus no bias on overfitted audio files that the model has already been trained on will be considered. This step was taken into consideration by the steps used when Radford (2022) expresses the need to test the models' capabilities with data that is unknown to the models' parameters. This objective created a test environment which all models could be classified and tested without any internal bias or data limitations. The process of creating the dataset used for the testing phase can be seen in 3.3.1 Data Standardisation.

3.2.5 Online Testing –

Online testing was conducted though the website 'golisten' an online platform that is able to create online audio-based tests and save the results subjectively and accurately. This platform was used as an alternative to the common 'WebMUSHRA' format due to its simplicity and ease of use for both the creator of the test, and its participants. The test itself was streamlined to only test the following models (Deepspeech, Whisper Small, Whisper Tuned LMT2 Small). The reasoning for using the largest models in the dataset, both with the master model and its fine-tuned model being the classified 'Small' model was a decision that would allow a more accurate representation of the abilities of the models. Put simply, the model with the best accuracy overall will be matched with the best model with the best accuracy for the test, as measuring 34 variants of the same test scenario would deter participants from answering correctly or answering anything at all due to the high information ingest needed for every test. The online test comprised of 10 questions with real world recordings of lectures from between (December 2021 to December 2023), with the recording locations being at 'Derby University, United Kingdom' within the discipline of 'MSc Audio Engineering'. These recordings were used due to the limited corpus available from Panopto. Each audio file was a segment of a lecture that was already published for students to recite, solidifying a test scenario explicitly similar in nature to what would be expected for the use of the models. Throughout the 10 questions, the participants were to listen to an audio file from a lecture mentioned above, and determine a multiple choice question on what transcription was able to replicate the audio with the most accuracy, The multiple choice question was split into the results from each of the key models stated (Deepspeech, Whisper Small, Whisper Tuned LMT2 Small), with each question having the order of the questions transcriptions set at random, eliminating any common bias on transcript positioning for each question. This would continue throughout the 10 allocated questions, with each question answered the participant would be given a section to express any reasoning or concern with the previous question, though this was not mandatory. Prior to the test mentioned above, the use of a pilot test was carried out as a prerequisite to determine any common problems or issues that may change the accuracy of the results proposed in the primary testing. The pilot test comprised of 3 audio files similar to the primary test in nature of audio context, with each of the transcriptions originally being a random selection of the 34 models' variants used for the paper. With the pilot theoretically setting each transcription model at random, this caused a large discrepancy within the transcripts as intrinsically the larger more robust models of any variant would outperform other less accurate

models. With this information the use of the most accurate and robust model was used in the primary testing to remove the discrepancy between each test.

3.2.6 Testing Limitations –

Test limitations are a key aspect of any data driven experiment, and thus should be considered when experiments are proposed and used to collect data. In this paper the online test was carried out by the participants by subjecting them to multiple audio recordings, then asking to select various transcriptions of the audio recordings in sequence. Information about the online testing can be found in 3.2.5. Because the way humans interpret audio stimulus varies between each person, the presumption that all data collected within the dataset will be accurate to some degree is false. The reasoning for this is that the online test itself although realistic to the modern application of the conjecture of the study, does not account for the bias in the generalised understanding of the data that is presented to the participants. An example of this is the use of native English speakers and comparing their ability to understand speech with varying accuracy to non-native English speakers. This classification of bias is defined by the online test limitation on the data collected, due to the small pool of participants used in the online test, the outlining bias will appear to be more visible within the result data. Classifying the proposed bias as noise in an experiments result, the greater datapoints you have within the set, the lower the noise impacts the overall averages of the test data. Comparing this aspect of bias with the offline test, although more accurate in the details of the data being tested, implied its own bias. The reasoning behind this assumption is that the offline test consisted of generating metrics based on the similarities of binary outputs, with the classification of these metric points being either yes or no for any datapoint being tested. The limitation of this classification is defined by the ability to understand the context of the data. As mentioned above, the implications of the online test outline the need to define a greater dataset, but this does not include creating a scenario that the participants need to contextually understand what is being said in the test recordings. This implies that the balance between participants and the lack of context is the main limitations to the overarching test carried out by the study. Put simply, the online test lacks the ability to de-noise the bias created by the diverse context given by the human nature when it comes to understanding of audio presented to them. Whereas the online test cannot understand the context of the test, thus presented with a situation where contextually a result is correct, the offline test will define such result as false.

3.3 Data Standardisation –

Standardisation is the process of removing unwanted data from a dataset or method, the idea being that the removal of such data will improve the accuracy of the results collected. In the case of the datasets processed by the tuned neural network, the use of a data standardisation procedure allows for the results of any test using standardised dataset to be classified as robust and accurate. Because the dataset used in the study 'common-voice' is community driven, the overarching accuracy of the dataset can be put into question.

3.3.1 Special Characters –

During the initial stages of training the 'Whisper' model to the new fine-tuned 'Whisper – LMT2' variant, the process of the training script outputting the training results in the form of the evaluation metrics outlined in 3.3.2 – Table 11 showed lower metrics than expected from the original Whisper metrics defined in Radford (2022). An investigation into the process of the evaluation script showed that the training scrip was using the defined classification validation dataset with the corresponding '.TSV' file that outlined the file names and the corresponding transcription, this would then be used to match the results of the checkpoint of the training process with key metrics like WER. The validated '.TSV' file and subsequent internal data was split into columns 'Path, Sentence' where the sentence section would be parsed with the data output of the model. Due to the community driven aspect of the dataset, some of the inputs from the community dataset implied special characters within the dataset that could not be created by the model during the evaluation stage, causing a situation where the model output would be defined as wrong, due to the reference data having special characters within the dataset. Radford (2022) outlines the use of a text standardisation procedure that defines all special characters from the output of the model, but not the inject to the model during the training of the model. Radford (2022) also presumes that all English spelling is converted to 'American Spelling' defined by the results shown in 3.1.3 – Table 7.

Table 12 – Invalid Characters found in Common Voice 5.1 – Validated.tsv File. Self (2023)

Set	Path .mp3	Sentence	Invalid Character
5.1	22215682	"It was also known as the ""Sunflower""	“”
5.1	543325	"He wasn't an alchemist!"	“”!
5.1	17406427	Why did you tell me?	?
5.1	17552437	On a scale from one (for not at all) and ten (for very much), how much pain do you feel?	(),?
5.1	20274470	"Karina Smirnoff of ""Dancing With The Stars"" hosted the following month."	“”.
5.1	1726832	Does this mean he will go to jail?	?
5.1	590018	The boy was also saddened; his friend was in pursuit of his destiny.	;.
5.1	18698093	"""Key: P–games played, W–games won, D–games drawn; L–games lost, %–win percentage"""	;“”-,:%
5.1	18698122	"Golino is the niece of ""L'Espresso"" journalist Enzo Golino; her brother is a musician."	“”';

The table above shows an example of invalid characters that are present in the validated file within the dataset, where in some cases the special character has no reason to be used in the classified sentence. This is the validated file that is parsed when evaluating the metrics when fine tuning the neural network. With this non-standard dataset the need to remove all reference files with invalid characters needs to be done before the use of any training data is used in the fine tuning process, to do this the use of a python script that finds and removes the items within the ‘validated.tsv’ file with respect to the audio file itself too being deleted from the dataset, resulting in a dataset that has files that do not host any invalid characters within the validation file or the audio files. The reason for the use of audio files being deleted in this case is to prevent the model being given any classification of audio that does not have a corresponding validated transcription for training.

Table 13 – Text Standardisation Code. Self (2023)

Code

```
with open(tsv_file1, 'r', encoding='utf-8') as file:
    reader = csv.DictReader(file, delimiter='\t')
    for row in tqdm(reader, desc="Processing rows", unit="row"):
        path = row['path']
        sentence = row['sentence']
        if (
            any(char in sentence for char in "',!-;£$%^&*()|~@→‘\\?'\"")
            or sentence in unique_sentences  # Check if the sentence has already been encountered
        ):
```

The code shown in Table 13 shows the array of special characters that the script will iterate over the validated dataset with, depending on the specific characters shown in the array ‘sentence’ referring to the sentence section of the ‘validated.tsv’ file.

3.3.2 Longform & Shortform Transcripts –

In conjunction with the work presented in 3.3.2, the script that would remove special characters from the specified dataset would also determine the overall length of the sentence within the dataset. The reason for this is based on primary test results of varying transcript lengths changing the results of the metrics involved when evaluating the model’s performance. Initial results indicated that the model was substantially accurate with shortform sentences with 3 to 5 words, this was indicated with a WER substantially higher than the overall metrics calculated when testing the model’s performance on a greater dataset including files that are more than

5 words long. With the context to the application of the model, where the model may be subjected to both long and shortform tasks, the use of an average was needed to accurately determine the overall performance of the model. With primary parameter for sentence length is defined as a sentence that is no longer than 30 characters or less than 15 characters in length, including any removal of special characters shown in Table 14.

Table 14 – Text Standardisation Code with respect to sentence length. Self (2023)

Code

```
with open(tsv_file1, 'r', encoding='utf-8') as file:
    reader = csv.DictReader(file, delimiter='\t')
    for row in tqdm(reader, desc="Processing rows", unit="row"):
        path = row['path']
        sentence = row['sentence']
        if (
            any(char in sentence for char in "',!-;£$%^&*()|~@→‘\\?'\"")
            or len(sentence) < 15  # Check if the sentence is less than 5 characters long
            or len(sentence) > 30 # Check if the sentence is more than 15 characters long
            or sentence.count('.') > 2  # Check if the sentence has more than 2 full stops
            or sum(1 for c in sentence if c.isupper() and c.isalpha()) > 1
            or sentence in unique_sentences  # Check if the sentence has already been encountered
        ):
            file_path = os.path.join(folder_path, path)
            if os.path.isfile(file_path):
                os.remove(file_path)
        else:
            paths_to_keep.add(path)
            unique_sentences.add(sentence)
```

3.3.3 TSV Standardisation –

As mentioned in 3.3.2 and 3.3.3, the removal of special characteristics within the dataset was used to create a test environment that is as robust as possible without introducing bias. Once this process has been completed the overall volume of test data is decreased substantially, allowing for further refinement to increase the accuracy and unbiased accuracy of the models when testing metrics. The ‘validated.tsv’ file used in the examples above is split into a corpus of 20,000 audio files at random based on a random number generator within the boundaries of the number of audio files present at the time the process is run. Returning 20,000 audio files that have been randomly selected to be the test dataset to evaluate all 34 variants of the models used in the study outlined in table 8. This standardisation also included creating a new format for the testing processes to operate from, where only the information that is needed for the test is given to the models. The reason this is important is due to the format of the ‘validated.tsv’ file before it is standardised, as the original format includes the following information that is not needed for the purpose of testing a models performance metrics, (up_votes, down_votes, age, gender, accent, locale, and segment). These cumulus and corresponding data points are removed in the process of the ‘validated.tsv’ standardisation.

3.4 Software & Data Limitations –

The original models that were trained in the paper Radford (2022) outlined a test environment that comprised of a dataset that housed over 680,000 hours of speech data. This data was then sent into a purpose-built neural network to train it on both transcription and translation. The hardware needed or used to process the original models is not disclosed by Radford (2022) but with key parameters that were used when initiating training were limited by the hardware used in the fine-tuning process of the ‘Whisper – LMT2’ model the calculations come to an approximation of a test environment being over 1 million times more capable in both hardware specific performance and memory bandwidth limitations. This poses an issue when determining the hardware used to fine tune the model and its ability to accurately create its own changes within the models’ structure with the limited bandwidth available with the hardware mentioned in table 3. This can create a situation where if you create a test environment that exceeds the current hardware limitations when fine tuning the original model, the test environment will fail due to hardware limitations, but the flipside to this scenario is that the parameters mentioned in table 10, do not allow the injection of the training data enough bandwidth to create any meaningful

change within the model as its training. Put simply, there is a hardware limit that is needed to accurately change a model at any size. Limiting hardware capability can cause errors.

Hardware limitations also include the use of adequate storage for the datasets, as well as the overall hardware needed to run the training. With the prerequisites of the training process completed the original dataset can be converted from its original size of 89GB referencing common voice corpus 13, to a size of 1250 Gigabyte with all files converted to a format used to train the specified model. With this taken into consideration the minimum storage needed to train the original model was 39 Terabyte compressed and over 40 Petabytes uncompressed.

4.0 Results –

Calculating the overall results of the metrics for the fine-tuned LMT2 Whisper models was carried out under the guidance of Ohri, a (2029) with statistical analysis of data driven by the use of logical reasoning mathematics outlined in the table below. The results that were used to calculate the overall score were (WER, CER, SER, WA, F1) with the output of the calculations being the 'Overall Score' calculated by taking the summation of the values and multiplying them by the 'F1 Score'. Because the F1 Score is defined by metrics calculated from the (WER, CER, SER, WA) metrics, the overarching multiplication of a value less than 1 is necessary to calculate an overall score.

Table 15 – Full Metric results for the Fine Tuned LMT2 Whisper models (Small, Base and Tiny). Including overall score. Self (2023)

Tiny Checkpoint	WER	CER	SER	WA	F1	Overall Score
1	50.7	65.98	23.8	66.54	0.7037	187.304198
2	42.8	45.79	22.39	67.37	0.7089	158.738593
3	38.9	37.74	21.97	69.62	0.7325	149.60665
4	37.77	36.66	21.46	70.56	0.7422	148.259632
5	36.74	37.07	20.86	70.65	0.7418	147.07817
6	37.08	38.08	21	71.73	0.7511	150.036403
7	32.29	30.68	20.63	72.95	0.7628	139.24626
8	30.28	25.7	20.43	73.54	0.7685	132.92549
9	34.14	29.31	20.36	73.48	0.7682	140.257336
10	33.67	29.77	20.42	73.37	0.7673	140.156801

Base Checkpoint	WER	CER	SER	WA	F1	Overall Score
1	21.25	15.04	16.91	80.27	0.8309	119.896343
2	20.45	14.13	16.55	81.27	0.8334	118.860418
3	19.79	13.43	16.7	80.6	0.833	117.0598
4	18.01	11.34	16.48	80.89	0.8352	113.389328
5	17.78	11.23	16.3	81.09	0.837	113.18233
6	17.76	11.23	16.28	81.15	0.8372	113.20878
7	19.13	13.18	16.21	81.22	0.8379	116.574238
8	19.03	13.16	16.12	81.34	0.8388	116.537992
9	19.02	13.15	16.05	81.37	0.8395	116.530115
10	19.07	13.16	16.12	81.34	0.8388	116.577992

Small Checkpoint	WER	CER	SER	WA	F1	Overall Score
1	15.11	10.8	12.25	85.7	0.877	113.3189
2	15.72	10.34	12.48	85.63	0.8742	113.397746
3	18.42	13.18	13.51	84.79	0.8644	118.402476
4	20.61	15.6	13.84	84.36	0.8607	122.658652
5	18.63	13.9	13.33	84.9	0.8663	119.40887
6	17.27	11.51	12.82	85.49	0.8718	116.130182
7	18.25	13.12	13.09	85.2	0.8689	118.49028
8	19.35	15.67	13.23	85.09	0.8677	122.082593
9	20.57	15.46	13.42	84.93	0.8658	122.982394
10	20.23	15.41	13.37	84.98	0.8663	122.628174

$$\sum_{i=1}^{n}(WER_i + CER_i + SER_i + WA_i \cdot F1_i) \tag{11}$$

$WER_i = Word\ Error\ Rate\ for\ the\ i^{th}\ Checkpoint$
$CER_i = Character\ Error\ Rate\ for\ the\ i^{th}\ Checkpoint$
$SER_i = Sentence\ Error\ Rate\ for\ the\ i^{th}\ Checkpoint$
$WA_i = \ Word\ Accuracy\ for\ the\ i^{th}\ Checkpoint$
$F1_i = F1\ Score\ for\ the\ i^{th}\ Checkpoint$
$Overall\ Score_i = \ WER_i + CER_i + SER_i + WA_i \cdot F1_i$

Because the array of numbers outlined in table 15 is complex, the introduction of graphical images creates a better image of what is happening within the metric results. Shown below is a series of figures outlining the performance of the 'Overall Score' based on the upper and lower limits of the score range.

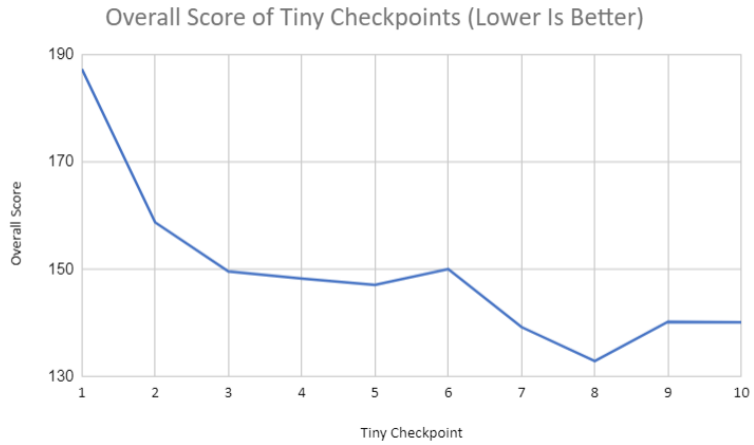

Figure 18 – Overall Score performance of the Tiny LMT2 Fine Tuned Model. Lowest Score being set at a value of 132.92549 at checkpoint 8. Self (2023)

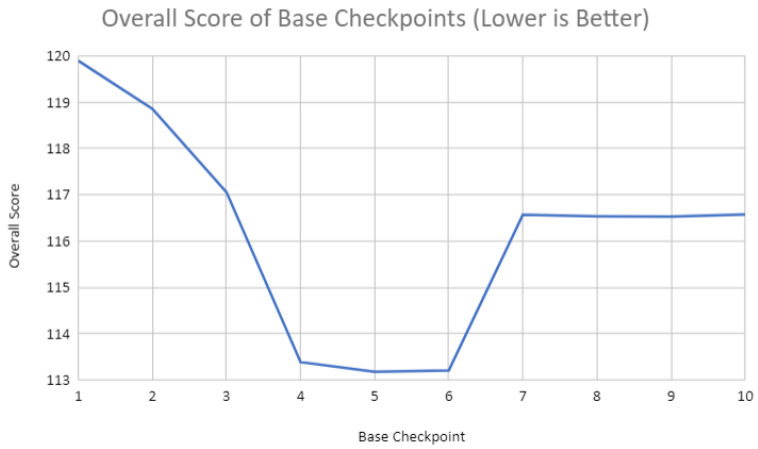

Figure 19 – Overall Score performance of the Base LMT2 Fine Tuned Model. Lowest Score being set at a value of 113.18233 at checkpoint 5. Self (2023)

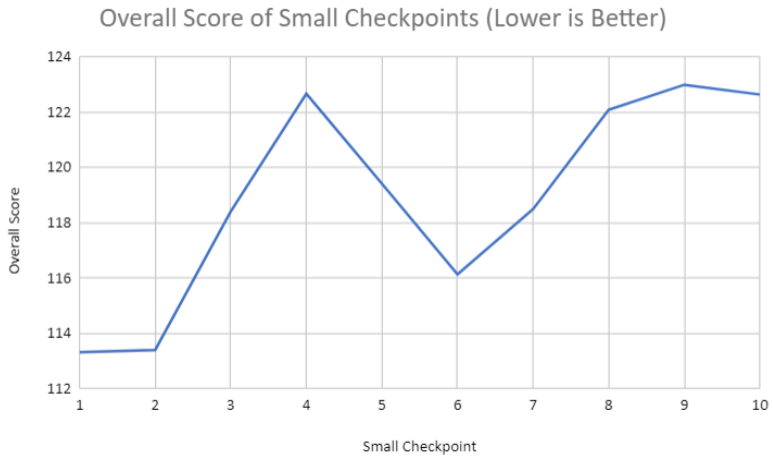

Figure 20 – Overall Score performance of the Small LMT2 Fine Tuned Model. Lowest Score being set at a value of 113.3189 at checkpoint 1. Self (2023)

In the models presented the overall lowest score comes from the 'Base' model at checkpoint 5, closely followed by checkpoint 1 of the 'Small' model at checkpoint 1. In addition to the fine-tuned LMT2 models the use of the original Whisper models classified as the Master models, as well as the original automatic speech recognition model used by Panopto is displayed in the following tables and figures. The master models include the following (Small, Base and Tiny) with the additional data given from the Deepspeech ASR model 0.9.3.

Table 16 – Full Metric results for the Whisper Master models (Small, Base and Tiny). Including overall score. Self (2023)

Whisper Master	WER	CER	SER	WA	F1	Overall Score
Tiny	63.28	73.99	24.15	64.97	0.6904	206.275288
Base	19.12	12.52	16.57	80.15	0.8343	115.079145
Small	15.31	11.1	13.64	87.02	0.8878	117.306356

Table 17 – Full Metric results for the Deepspeech 0.9.3 ASR model, Including overall score. Self (2023)

Deep Speech	WER	CER	SER	WA	F1	Overall Score
0.9.3	66.31	23.08	62.21	34.28	0.3779	164.554412

Table 18 – Overall Score results for Master Models. Self (2023)

Master Models	Overall Score
WSPR Tiny	206.275288
WSPR Base	115.079145
WSPR Small	117.306356
DS 0.9.3	164.554412

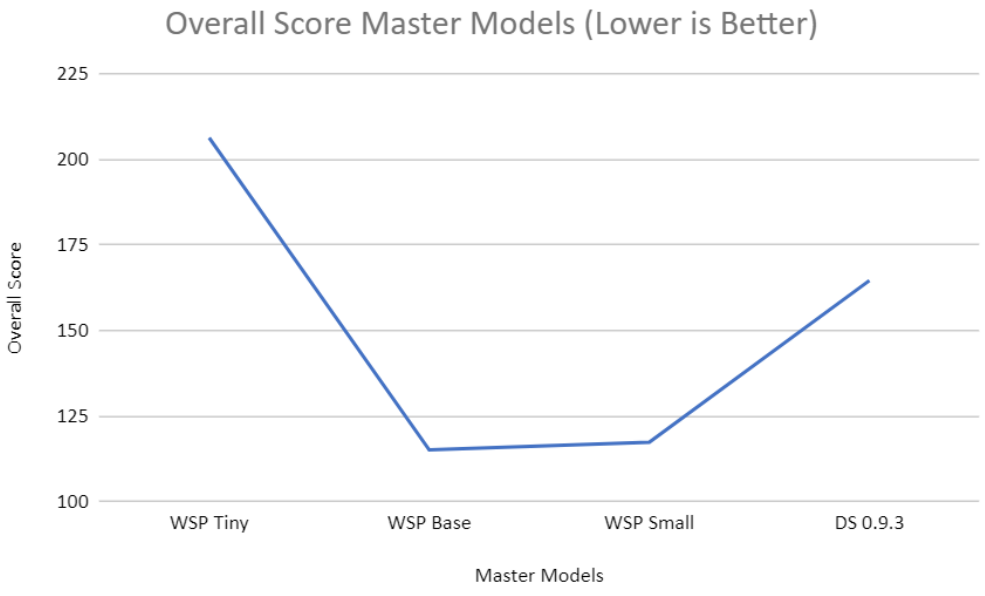

Figure 21 – Overall Score performance of the Master Models. Lowest Score being set at a value of 115.079145. Self (2023)

Outlining the best preforming LMT2 with its counterpart before and after the training procedure has taken place the following calculations can be carried out, in addition to a T-Test to define the mean difference calculated. Ohri, a (2029) defines that a set group of results with the purpose of defining if the datasets are statistically different can be expressed by preforming a T-Test. The test itself simply compares the mean of two separate groups to determine if they are statistically different from each other.

$$
\begin{aligned}
&Whisper\ Master\ Base\ Overall\ Score = 115.079145 \\
&LMT2\ Base\ Overall\ Score = 113.18233 \\
&\frac{|V_1 - V_2|}{\left[\frac{(V_1 + V_2)}{2}\right]} \times 100 = \frac{|113.18233 - 115.079145|}{\left[\frac{(113.18233 + 115.079145)}{2}\right]} \times 100 = \frac{|-1.896815|}{\left[\frac{228.261475}{2}\right]} \times 100 \\
&= \frac{1.896815}{114.1307375} \times 100 = 0.0166197 \times 100 \\
&Value\ difference: 1.66197\%
\end{aligned}
\tag{12}
$$

This result implies that the overall score difference between the two models is approximately 1.66197%. To define the statistical variance between the value defined by both models the use of a t-test is used in the following tables and equations.

Table 19 – Model results comparing Whisper Base Master and the fine-tuned LMT2 Base Model. Self (2023)

Models	WER	CER	SER	WA	F1	Overall Score
Base CP 5	17.78	11.23	16.3	81.09	0.837	113.18233
WSPR Base	19.12	12.52	16.57	80.15	0.8343	115.079145

Table 20 – Statistical T-Test Independent Means. Self (2023)

Model	Array [1, 2]	Diff (X - M)	Sq. Diff (X - M)^2	M:	SS:
LMT2 CP 5	17.78	-22.29	496.84	40.07	10460.88
	11.23	-28.84	831.74		
	16.3	-23.77	565.01		
	81.09	41.02	1682.65		
	0.837	-39.23	1539.22		
	113.18233	73.11	5345.43		
WSPR Base	19.12	-21.59	466.22	40.71	10519.9
	12.52	-28.19	794.8		
	16.57	-24.14	582.85		
	80.15	39.44	1555.34		
	0.8343	-39.88	1590.25		
	115.079145	74.37	5530.44		

$$
\begin{aligned}
&LMT2\ Results: N_1 = 6 \mid M_1 = 40.07 \mid SS_1 = 10460.88 \\
&WSPR\ Results: N_2 = 6 \mid M_2 = 40.71 \mid SS_2 = 10519.9 \\
&Degrees\ of\ freedome: df_1 = N_1 - 1 = 5 \mid df_2 = N_2 - 1 = 5 \\
&Varience\ of\ arrays: s_1^2 = \frac{SS_1}{N_1 - 1} = \frac{10460.88}{5} = 2092.18 \mid s_2^2 = \frac{SS_2}{N_2 - 1} = \frac{10519.9}{5} = 2103.98 \\
&Pool\ varience: s_p^2 = \left(\frac{df_1}{df_1 + df_2}\right) \times s_1^2 + \left(\frac{df_2}{df_1 + df_2}\right) \times s_2^2 \\
&s_p^2 = \left(\frac{5}{10}\right) 2092.18 + \left(\frac{5}{10}\right) 2103.98 = 2098.08 \\
&Varience\ of\ means\ for\ arrays: s_{M_1}^2 = \frac{s_p^2}{N_1} = \frac{2098.08}{6} = 349.68 \mid s_{M_2}^2 = \frac{s_p^2}{N_2} = \frac{2098.08}{6} = 349.68 \\
&T - value\ Calculation: t = \frac{M_1 - M_2}{\sqrt{s_{M_1}^2 + s_{M_2}^2}} =: t = \frac{-0.64}{\sqrt{699.36}} \\
&t = 0.02429 \mid p - value = 0.49055 \mid p < .01
\end{aligned}
\tag{13}
$$

The online testing phase of the study expressed in section 3.2.5 can be expressed in the following tables and graphs. Note the initial data displayed will be the underlying data from the test itself. The table below shows the array of transcriptions and Answers to such transcriptions in reference to the test questions.

Table 21 – Test Outline showing the model outputs and the corresponding selections of results. Self (2023)

Source File	Audio File	Model	Test Question Selection	Question
2023-07-26 18-21-27	2023-07-26 18-21-27_Audio	ASR	C	1
		WHISPER	A	
		LMT2	B	
2023-07-26 18-39-09	2023-07-26 18-39-09_Audio	ASR	B	2
		WHISPER	A	
		LMT2	C	
2023-07-26 18-55-19	2023-07-26 18-55-19_Audio	ASR	A	3
		WHISPER	C	
		LMT2	B	
2023-07-26 18-47-58	2023-07-26 18-47-58_Audio	ASR	C	4
		WHISPER	B	
		LMT2	A	
2023-07-26 18-52-08	2023-07-26 18-52-08_Audio	ASR	B	5
		WHISPER	C	
		LMT2	A	
2023-07-26 18-33-18	2023-07-26 18-33-18_Audio	ASR	A	6
		WHISPER	C	
		LMT2	B	
2023-07-26 18-19-03	2023-07-26 18-19-03_Audio	ASR	A	7
		WHISPER	C	
		LMT2	B	
2023-07-26 18-26-14	2023-07-26 18-26-14_Audio	ASR	C	8
		WHISPER	B	
		LMT2	A	
2023-07-26 19-02-52	2023-07-26 19-02-52_Audio	ASR	C	9
		WHISPER	A	
		LMT2	B	
2023-07-26 19-01-22	2023-07-26 19-01-22_Audio	ASR	B	10
		WHISPER	A	
		LMT2	C	

The test results themselves were separated into groups that were classified as participants that were non-native English speakers and those who were native English speakers. The reason for this divide is to ensure that any bias that may be a result of a result being classified by the participants native ability to understand the question, was not to impact the overall mean average calculated. The table below indicates the results of the test with respect to the LMT2 model and Whisper Model.

Table 22 – Test Outline of model outputs and the corresponding selections of results T-Test. Self (2023)

Model	Results Array	Diff (X - M)	Sq. Diff (X - M)^2
LMT2	[5,2,5,8,4,5,8,7,3,6,5,7,2,2,6,4,5,3,8,8,8,4]	M: 5.23	SS: 91.86
WSPR	[5,8,4,2,6,5,2,3,7,4,5,3,8,8,4,4,5,7,2,2,2,6]	M: 4.64	SS: 91.09

$$
\begin{aligned}
&LMT2\ Results: N_1 = 22 \mid M_1 = 5.23 \mid SS_1 = 91.86 \\
&WSPR\ Results: N_2 = 22 \mid M_2 = 4.64 \mid SS_2 = 91.09 \\
&Degrees\ of\ freedome: df_1 = N_1 - 1 \mid df_2 = N_2 - 1 \\
&Varience\ of\ arrays: s_1^2 = \frac{SS_1}{N_1 - 1} \mid s_2^2 = \frac{SS_2}{N_2 - 1} \\
&Pool\ varience: s_p^2 = \left(\frac{df_1}{df_1 + df_2}\right) \times s_1^2 + \left(\frac{df_2}{df_1 + df_2}\right) \times s_2^2 \\
&Varience\ of\ means\ for\ arrays: s_{M_1}^2 = \frac{s_p^2}{N_1} \mid s_{M_2}^2 = \frac{s_p^2}{N_2} \\
&T - value\ Calculation: t = \frac{M_1 - M_2}{\sqrt{s_{M_1}^2 + s_{M_2}^2}} \\
&t = 0.93901 \mid p - value = 0.176547 \mid p < .01
\end{aligned}
\tag{14}
$$

Lastly with the assumption that the model LMT2 Base is compared with the Deepspeech model 0.9.3 with regards to the research conjecture, that being the model accuracy is comparable to the ASR technology used within the Panopto transcription.

Table 23 – LMT2 Base CP 5 Metric Comparison with Deepspeech 0.9.3 ASR model. Self (2023)

Model	Function	WER	CER	SER	WA	F1	Overall Score
Base CP 5		17.78	11.23	16.3	81.09	0.837	113.18233
DS 0.9.3		66.31	23.08	62.21	34.28	0.3779	164.554412
	A-B Diff	48.53	11.85	45.91	46.81	0.4591	-
	Difference %	115.434	69.076	116.953	81.148	75.578	91.6378 %

Using a basic form of arithmetic, the [A-B] differences between each of the metrics from both the LMT2 Base CP 5 model and the Deepspeech 0.9.3 models, can be calculated. Using the mean average difference between all resulting metric differences, the overall performance difference between the two models can be statistically classified as the LMT2 Base CP 5 model being 91.6378 percent more accurate across all collected metrics than the Deepspeech model 0.9.3 ASR model. Finally, each of the key metrics can be visualised in comparison to the model and the checkpoints in the following figures that outline the checkpoints for each of the models trained as well as where each of the metrics compare with the other models within the set.

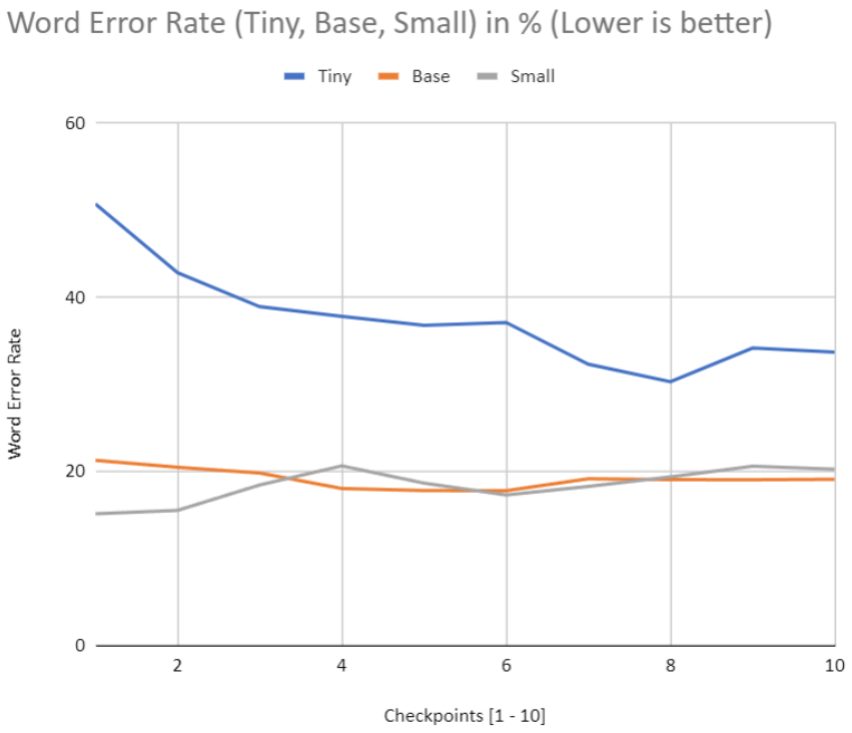

Figure 22 – Word Error Rate (WER) comparison in reference to the LMT2 models and subsequent checkpoints. Self (2023)

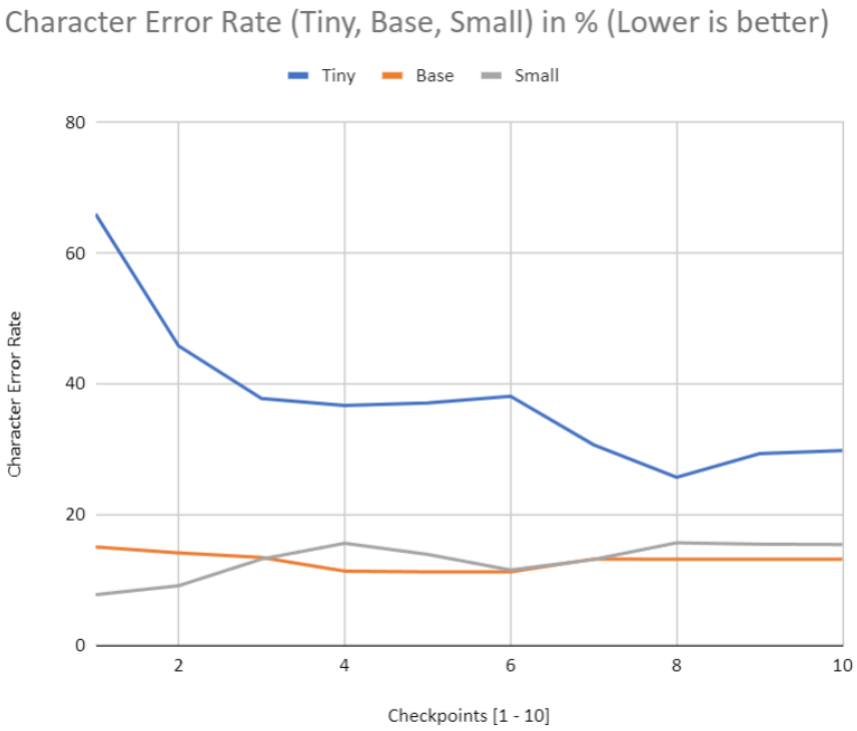

Figure 23 – Character Error Rate (CER) comparison in reference to the LMT2 models and subsequent checkpoints. Self (2023)

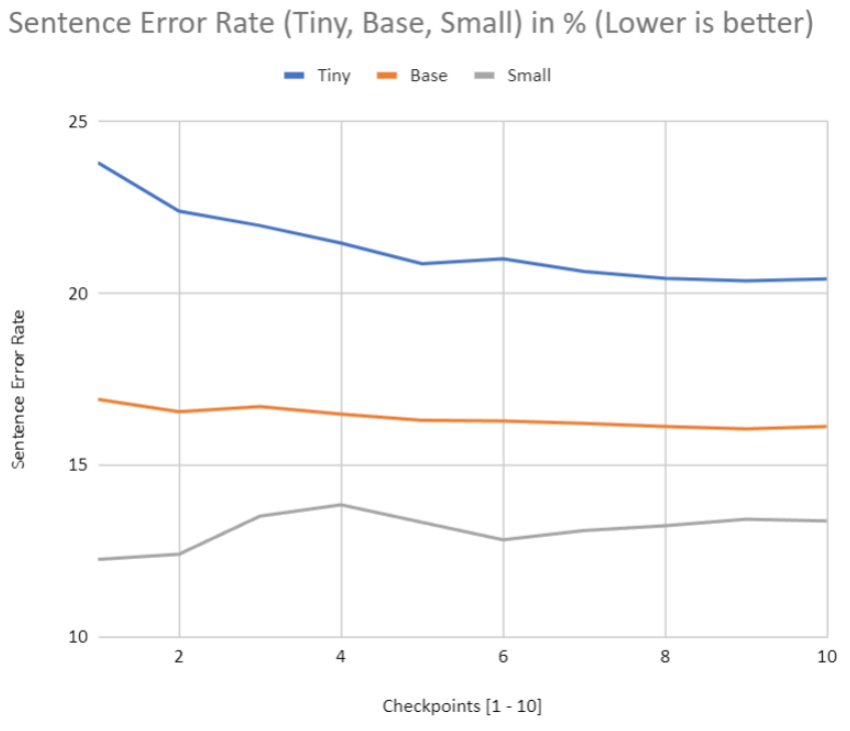

Figure 24 – Sentence Error Rate (SER) comparison in reference to the LMT2 models and subsequent checkpoints. Self (2023)

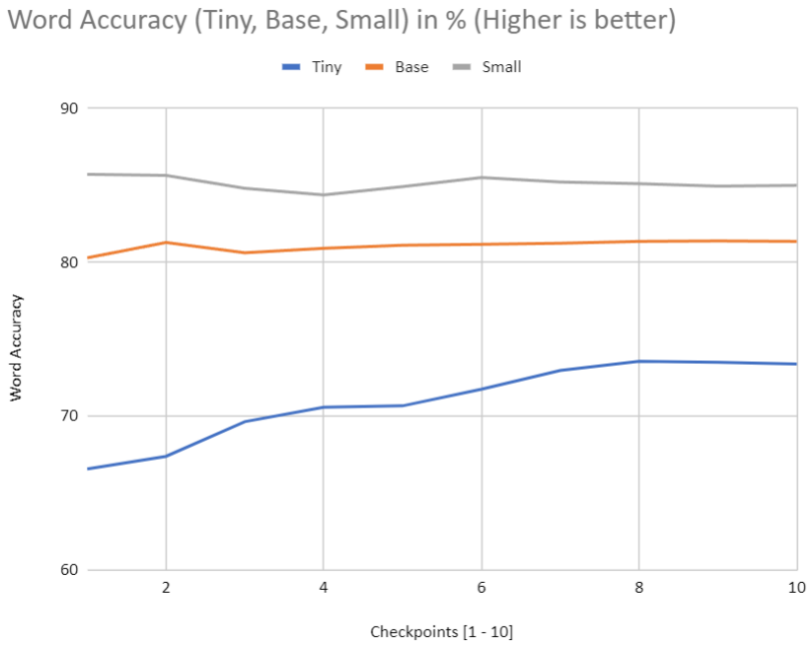

Figure 25 – Word Accuracy (WA) comparison in reference to the LMT2 models and subsequent checkpoints. Self (2023)

5.0 Discussion –

Although the results of the tests show a somewhat fragmented result, the underlying points can be made, both from a statistical standpoint and a subjective standpoint within the confines of the online test results. During the initial stages of the research the prospective that the larger model defined by its parameters being 'Whisper Small' would preform at the highest accuracy. This initial correlation does not show to be true within the results shown in table 15, as the most accurate model in reference to the 'Overall Score' is defined by the 'LMT2 Base Checkpoint 5' model. Statistical analysis using a T-Test at significance <.01 defined by the findings by Ohri, A (2019) show no significant deviations between results or all items in the datasets shown and expressed in table 19 and equation 13. What can be drawn from these results is that with respect to the 'Whisper Master Tiny, Base' the process of fine tuning the models 'LMT2 – Tiny, Base' show a positive correlation that the tuning of further data improves the overall accuracy across the metrics tested. Data to support this can be seen in the following tables, 15, 16, 17 and 19.

Online testing does not show any statistical deviation at <.01 between the use of the 'LMT2' model and the 'Master Whisper' model, although these results show a contrasting argument with respect to the ASR technology used for Panopto transcriptions. The use of 'Deep Speech 0.9.3' within the online tests showed no data representing any links to transcriptions, meaning this model was not selected in any of the responses from the participants. With a statistical analysis of 'Deep Speech 0.9.3' and its offline performance with respect to 'LMT2 – Base Checkpoint 5' the overall accuracy of 'LMT2 - Base Checkpoint 5' is 91.6% higher than that of 'Deep Speech 0.9.3' shown in table 23. A reasonable assumption that the 'Whisper Master Small' was somewhat outperformed by the 'LMT2 – Base' model can be roughly attributed to the hardware limitations of the training process outlined in 3.4.

Where the assumption that the tuning parameters limited by hardware were sufficient enough to change the data structure of the smaller models, while the larger model 'Whisper Master Small' was too large in comparison to the parameters that were used in the tuning process, creating a situation where the model was not being trained correctly. This assumption can be expressed by observing table 15 where the 'Overall Score' of each checkpoint of the 'LMT2 – Small' model, as from checkpoint 1 to 10, the score increases, indicating the model is becoming less accurate with each training checkpoint. This can also be seen by the individual metrics outlined in table 15 and shown in Figure 20 as the model throughout the training process indicates a positive curve with each increasing checkpoint the model's overall performance is decreasing and the overall accuracy of the model is deteriorating with each step.

With the overall conjecture of the research being a comparison of Deepspeech and its accuracy compared to both the tuned LMT2 models and the original Whisper models, the outline of hardware capabilities needs to be considered. With the current size of the Deepspeech 0.9.3 model having approximately 800 million parameters shown in table 2, the use of hardware needs to be considered when processing any meaningful data. Statistically a model parameter is a 32-bit number at 4 bytes of memory usage, multiplying this number with the total parameters the result comes to approximately 3,200,000,000 bytes, or 3,200 MB in overall memory usage. This number is only considering the model itself, not the underlying models periphery software.

Comparing this result to the use of the LMT2 model with its 72,593,920 parameters, the overall memory usage for this model is approximately 290.37 MB. Meaning that the LMT2 model is 166.72% smaller than the Deepspeech model, equating to the LMT2 model being 11 times smaller than the Deepspeech model. This result indicates that the overall usage of the LMT2 model would not only be more beneficial from a hardware standpoint but with the LMT2 model being over 3.72 times more accurate then Deepspeech with respect to WER, 2 times more accurate with respect to CER, 3.8 times more accurate with respect to SER and finally, 3.36 times more accurate with respect to WA.

6.0 Conclusion –

Overall, the models created by 'Open AI' with reference to (Small, Base, Tiny) can be further trained effectively with new unrecognised data to improve overall accuracy of the models. The fine-tuned models as well as the master models all outperformed the automatic speech recognition model used by 'Panopto' in every metric. Statistically, the LMT2 model variants are not significantly deviated from the original master models with the same name at <.01 using a T-test. Online testing indicated that none of the participants tested were inclined to select transcriptions made with 'Deepspeech ASR (Panopto)' with respect to either 'LMT2' or 'Whisper Master' models, as nobody from the test pool selected 'Deepspeech' transcriptions but, testing on a greater pool size may change results. The relationship between a neural-network parameter count, and the overall accuracy of such numbered parameters, could indicate a relationship that defines what parameter count would indicate a higher performance network with respect to the metrics calculated.

Figures 22 and 23 indicate a strong visual correlation between the performances of the 'Small' and 'Base' models, although each of these models have a parameter delta of 169,140,992. This delta indicates that the 'Small' model is 2.33 times more parameter dense than the 'Base' model, although the overall performance delta between the two models is 1.92%. Although the statistical analysis showed no variance of the 'Whisper' models in comparison to the 'LMT2' models, the overall performance from the data provided shows an increase in accuracy across the key metrics calculated.

Secondly, as mentioned in the discussion, the results outlined in Table 22 show a brief indication that the performance of the Deep Speech ASR model used within the test was not desired at all, with no participants on the online test outlining any value of accuracy to the aforementioned model, with no transcripts in the test relating to this model being chosen when compared to the Whisper model variants. Further information and data size is needed to conclude such findings, but the probability that the information gathered indicates any bias towards ASR is zero.

Lastly, the performance of the models from a hardware standpoint can only be calculated with respect to the overall parameters used within the models themselves, with raw parameter values being shown in Table 2, the large delta between the whisper models and the Deepspeech models indicates a memory resource benefit, when compared with the other models in the table.

6.1 Future Works –

The overall accuracy of the models during the training phase is intrinsically limited to the hardware capabilities of the hardware capable of tuning and injecting data to the models. Further work with respect to the larger models provided by 'Open AI – Whisper' being 'Medium, Large and Large-v2' could indicate higher accuracy across the metrics. Secondly, collecting more data for the use of training would give the model more infrastructure change to encourage better reasoning. Lastly, the use of this software in the workplace of Panopto or any other model-based transcription platform could indicate more accuracy with respect to the metrics calculated.

Lastly, the inclusion of other datapoints with the input reference data being tested could increase the overall accuracy of the models results, especially when secondary languages and accents are added into the datapoints. A model that is able to fit the gaps in data between these points will be able to determine if such variables effect the overall accuracy from the perspective of the reader.

REFERENCES –

1. *Panopto (2022). Email Exchange. [online] Outlook.com. Available at: N/A [Accessed 4 Jun. 2022].*
2. *IBM (2003). IBM Archives: IBM Shoebox. [online] Ibm.com. Available at: https://www.ibm.com/ibm/history/exhibits/specialprod1/specialprod1_7.html.*
3. *Ibm.com. (2012). IBM100 - Pioneering Speech Recognition. [online] Available at: https://www.ibm.com/ibm/history/ibm100/us/en/icons/speechreco/transform/.*
4. *Science and Technology (2020). Alexander Graham Bell's box telephone. [online] National Museums Scotland. Available at: https://www.nms.ac.uk/explore-our-collections/stories/science-and-technology/alexander-graham-bell/.*
5. *History.com Editors (2009). Alexander Graham Bell. [online] HISTORY. Available at: https://www.history.com/topics/inventions/alexander-graham-bell.*
6. *Furlanello, C., Merler, S. and Jurman, G. (2006). Combining feature selection and DTW for time-varying functional genomics. IEEE Transactions on Signal Processing, 54(6), pp.2436–2443. doi:https://doi.org/10.1109/tsp.2006.873715.*
7. *Przemyslaw Dymarski (2011). Hidden Markov Models, Theory and Applications. doi:https://doi.org/10.5772/601.*
8. *Baum, L.E. and Petrie, T. (1966). Statistical Inference for Probabilistic Functions of Finite State Markov Chains. The Annals of Mathematical Statistics, [online] 37(6), pp.1554–1563. Available at: https://www.jstor.org/stable/2238772 [Accessed 14 Oct. 2022].*
9. *nipunbatra.github.io. (2018). Exploring Hidden Markov Models. [online] Available at: https://nipunbatra.github.io/hmm/ [Accessed 6 Aug. 2023].*
10. *Swietojanski, Pawel. (2016). Learning Representations for Speech Recognition using Artificial Neural Networks.*
11. *Cinelli, L., Chaves, G. and Lima, M. (2018). Vessel Classification through Convolutional Neural Networks using Passive Sonar Spectrogram Images. Anais de XXXVI Simpósio Brasileiro de Telecomunicações e Processamento de Sinais. doi:https://doi.org/10.14209/sbrt.2018.340.*
12. *www.astroml.org. (n.d.). Neural Network Diagram — astroML 0.4 documentation. [online] Available at: https://www.astroml.org/book_figures/chapter9/fig_neural_network.html [Accessed 7 Aug. 2023].*
13. *Vuckovic, Aleksandra & Radivojevic, Vlada & Chen, Andrew & Popović, Dejan. (2015). EEG Drowsiness 2002 MedEngPhy.*
14. *Silva, S.D. (2020). The Maths behind Back Propagation. [online] Medium. Available at: https://towardsdatascience.com/the-maths-behind-back-propagation-cf6714736abf [Accessed 9 Aug. 2023].*

15. *Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. [online] proceedings.mlr.press. Available at: http://proceedings.mlr.press/v9/glorot10a.*
16. mozilla (2019). *mozilla/DeepSpeech*. [online] GitHub. Available at: https://github.com/mozilla/DeepSpeech.
17. Radford, A., Kim, J.W., Xu, T., Brockman, G., McLeavey, C. and Sutskever, I. (2022). Robust Speech Recognition via Large-Scale Weak Supervision. *arXiv:2212.04356 [cs, eess]*. [online] Available at: https://arxiv.org/abs/2212.04356.
18. GitHub. (2022). *Whisper*. [online] Available at: https://github.com/openai/whisper.
19. Python Software Foundation (2019). *What is Python? Executive Summary*. [online] Python.org. Available at: https://www.python.org/doc/essays/blurb/.
20. Ying, X. (2019). An Overview of Overfitting and its Solutions. Journal of Physics: Conference Series, 1168(2), p.022022. doi:https://doi.org/10.1088/1742-6596/1168/2/022022.
21. Collister, L. (2020). *Guides: Open Licenses: Creative Commons and other options for sharing your work: MIT License*. [online] pitt.libguides.com. Available at: https://pitt.libguides.com/openlicensing/MIT#:~:text=Like%20the%20Apache%202.0%2C%20and.
22. Ohri, A 2019, SAS for R Users : A Book for Data Scientists, John Wiley & Sons, Incorporated, Newark. Available from: ProQuest Ebook Central. [24 August 2023].
23. Socscistatistics.com. (2019). T-Test Calculator for 2 Independent Means. [online] Available at: https://www.socscistatistics.com/tests/studentttest/.
24. research.facebook.com. (2020). *Log in or sign up to view*. [online] Available at: https://research.facebook.com/publications/wav2vec-2-0-a-framework-for-self-supervised-learning-of-speech-representations/ [Accessed 29 Aug. 2023].

APENDICIES –

Following links provided below will show the corpus of code used in the paper.

1. https://github.com/TtesseractT/LMT2-Tuning-Whisper/tree/main
2. https://github.com/TtesseractT/LM-S2T

Page Break for Code

Next Page

LMT2 – Tiny Training Code Master

```
'''
Model Trainer from HF Fine tune Whisper tiny Model

Author: Sabian Hibbs
University of Derby
United Kingdom, England

Licence MIT
'''
# Open the log file in append mode
log_file = open("cmd_log_tiny-10.txt", "a")

import os
os.environ["HF_DATASETS_CACHE"] = "E:\huggingface"
os.environ["TRANSFORMERS_CACHE"] = "E:\huggingface"

# Loading checkpoints from hugging face
from huggingface_hub import notebook_login
from datasets import load_dataset, DatasetDict
from transformers import WhisperFeatureExtractor
from transformers import WhisperTokenizer
from transformers import WhisperProcessor
from transformers import WhisperForConditionalGeneration
from transformers import Seq2SeqTrainingArguments
from transformers import Seq2SeqTrainer
from datasets import Audio
from dataclasses import dataclass
from typing import Any, Dict, List, Union

import torch
import evaluate
import subprocess

# Huggingface pylance token
# Make sure you run cmd first and use

#subprocess.run(['huggingface-cli', 'login'])

# huggingface-cli login
# login to hugging face and get a token
# paste the token below
# accept and sign the agreement for 13_0 dataset

if __name__ == '__main__':

    notebook_login()

    common_voice = DatasetDict()
    common_voice["train"] = load_dataset("mozilla-foundation/common_voice_13_0", "en",
split="train+validation")
    common_voice["test"] = load_dataset("mozilla-foundation/common_voice_13_0", "en", split="test")
    # Print for debug
    #print(common_voice)

    common_voice = common_voice.remove_columns(["accent", "age", "client_id", "down_votes",
"gender", "locale", "path", "segment", "up_votes"])
    # Print for debug
    #print(common_voice)

    # --------------------------
    # Feature Extraction Process - tiny
    # --------------------------

    feature_extractor = WhisperFeatureExtractor.from_pretrained("openai/whisper-tiny")

    # --------------------------
    # Load Whisper Tokenizer - tiny
    # --------------------------
    tokenizer = WhisperTokenizer.from_pretrained("openai/whisper-tiny", language="en",
task="transcribe")
```

```
    # -------------------------
    # Combine WhisperProcessor - tiny
    # -------------------------

    processor = WhisperProcessor.from_pretrained("openai/whisper-tiny", language="en",
task="transcribe")

    # -------------------------
    # Prepare Data
    # -------------------------

    print(common_voice["train"][0])
    common_voice = common_voice.cast_column("audio", Audio(sampling_rate=16000))

    # Re-loading the first audio sample in the Common Voice dataset will resample
    print(common_voice["train"][0])

    def prepare_dataset(batch, feature_extractor, tokenizer):
        # load and resample audio data from 48 to 16kHz
        audio = batch["audio"]

        # compute log-Mel input features from input audio array
        batch["input_features"] = feature_extractor(audio["array"],
sampling_rate=audio["sampling_rate"]).input_features[0]

        # encode target text to label ids
        batch["labels"] = tokenizer(batch["sentence"]).input_ids
        return batch


    common_voice = common_voice.map(prepare_dataset, fn_kwargs={'feature_extractor':
feature_extractor, 'tokenizer': tokenizer}, remove_columns=common_voice.column_names["train"],
num_proc=31)

    # -------------------------
    # Define Data Collector
    # -------------------------

    @dataclass
    class DataCollatorSpeechSeq2SeqWithPadding:
        processor: Any

        def __call__(self, features: List[Dict[str, Union[List[int], torch.Tensor]]]) -> Dict[str,
torch.Tensor]:
            # split inputs and labels since they have to be of different lengths and need different
padding methods
            # first treat the audio inputs by simply returning torch tensors
            input_features = [{"input_features": feature["input_features"]} for feature in features]
            batch = self.processor.feature_extractor.pad(input_features, return_tensors="pt")

            # get the tokenized label sequences
            label_features = [{"input_ids": feature["labels"]} for feature in features]
            # pad the labels to max length
            labels_batch = self.processor.tokenizer.pad(label_features, return_tensors="pt")

            # replace padding with -100 to ignore loss correctly
            labels = labels_batch["input_ids"].masked_fill(labels_batch.attention_mask.ne(1), -100)




            # if bos token is appended in previous tokenization step,
            # cut bos token here as it's append later anyways
            if (labels[:, 0] == self.processor.tokenizer.bos_token_id).all().cpu().item():
                labels = labels[:, 1:]

            batch["labels"] = labels

            return batch

    data_collator = DataCollatorSpeechSeq2SeqWithPadding(processor=processor)
```

```
# --------------------------
   # Evaluation Metrics
   # --------------------------

   metric = evaluate.load("wer")

   def compute_metrics(pred):
       pred_ids = pred.predictions
       label_ids = pred.label_ids

       # replace -100 with the pad_token_id
       label_ids[label_ids == -100] = tokenizer.pad_token_id

       # we do not want to group tokens when computing the metrics
       pred_str = tokenizer.batch_decode(pred_ids, skip_special_tokens=True)
       label_str = tokenizer.batch_decode(label_ids, skip_special_tokens=True)

       wer = 100 * metric.compute(predictions=pred_str, references=label_str)

       return {"wer": wer}

   # --------------------------
   # Load Pretrained Checkpoints
   # --------------------------

   model = WhisperForConditionalGeneration.from_pretrained("openai/whisper-tiny")
   model.config.forced_decoder_ids = None
   model.config.suppress_tokens = []

   # --------------------------
   # Define the Training Config
   # --------------------------

   training_args = Seq2SeqTrainingArguments(
       output_dir="./LMT2-Tiny-10",  # change to a repo name of your choice
       per_device_train_batch_size=32,
       gradient_accumulation_steps=1,  # increase by 2x for every 2x decrease in batch size
       learning_rate=1e-5,
       warmup_steps=500,
       max_steps=10000,
       gradient_checkpointing=True,
       fp16=True,
       evaluation_strategy="steps",
       per_device_eval_batch_size=8,
       predict_with_generate=True,
       generation_max_length=225,
       save_steps=1000,
       eval_steps=1000,
       logging_steps=100,
       report_to=["tensorboard"],
       load_best_model_at_end=True,
       metric_for_best_model="wer",
       greater_is_better=False,
       push_to_hub=True,
   )









   trainer = Seq2SeqTrainer(
       args=training_args,
       model=model,
       train_dataset=common_voice["train"],
       eval_dataset=common_voice["test"],
       data_collator=data_collator,
       compute_metrics=compute_metrics,
       tokenizer=processor.feature_extractor,
   )
```

```
    processor.save_pretrained(training_args.output_dir)

    '''
    -------------------------------------------
    ##### Model Training Whisper tiny Model #####
    -------------------------------------------
    '''

    trainer.train()

    kwargs = {
        "dataset_tags": "asr, speech-to-text, lmt2, tesseract3d, whisper-tiny, mozilla, common-
voice, en",
        "dataset": "Common Voice 13.0",
        "dataset": "Common Voice 13.0",
        "dataset_args": "config: en, split: test",
        "language": "en",
        "model_name": "Tesseract3D/LMT2-tiny",
        "finetuned_from": "openai/whisper-tiny",
        "tasks": "automatic-speech-recognition",
        "tags": "hf-asr-leaderboard",
    }

    trainer.push_to_hub(**kwargs)

    print("TRAINING COMPLETE!")
        # Close the log file
log_file.close()
```

Metric Calculation Script

```
'''
Author: Sabian Hibbs
University of Derby
United Kingdom, England

Licence MIT
'''

import csv
import random
from nltk.metrics.distance import edit_distance
from nltk.metrics import f_measure
import os
from tqdm import tqdm

def calculate_wer(reference, hypothesis):
    # Calculate the Word Error Rate
    wer = edit_distance(reference.split(), hypothesis.split()) / len(reference.split())
    return wer

def calculate_cer(reference, hypothesis):
    # Calculate the Character Error Rate
    cer = edit_distance(reference, hypothesis) / len(reference)
    return cer

def calculate_word_accuracy(reference, hypothesis):
    # Calculate Word Accuracy
    num_correct_words = sum(1 for ref_word, hyp_word in zip(reference.split(), hypothesis.split())
if ref_word == hyp_word)
    word_accuracy = num_correct_words / len(reference.split())
    return word_accuracy

def calculate_sentence_error_rate(reference, hypothesis):
    # Calculate Sentence Error Rate
    ref_set = set(reference.split())
    hyp_set = set(hypothesis.split())
    f1 = f_measure(ref_set, hyp_set)
    if f1 is not None:
        ser = 1 - f1  # Sentence Error Rate is 1 minus the F1 score
    else:
        ser = None
    return ser

def calculate_f1_score(reference, hypothesis):
    # Calculate F1-Score
    if not reference or not hypothesis:
        return 0.0

    ref_set = set(reference.split())
    hyp_set = set(hypothesis.split())
    f1 = f_measure(ref_set, hyp_set)
    return f1

def compare_transcriptions(validated_file, speech_recognition_file, num_samples=4000, num_tests=10):
    # Read and parse TSV files
    validated_data = {}
    speech_recognition_data = {}

    with open(validated_file, 'r', encoding='utf-8') as file:
        reader = csv.reader(file, delimiter='\t')
        next(reader)  # Skip the header row
        for row in reader:
            validated_data[row[0]] = row[1]  # Map file name to sentence

    with open(speech_recognition_file, 'r', encoding='utf-8') as file:
        reader = csv.reader(file, delimiter='\t')
        next(reader)  # Skip the header row
        for row in reader:
            speech_recognition_data[row[0]] = row[1]  # Map file name to sentence
```

```
    total_results = []
    with tqdm(total=num_samples*num_tests, desc="Running Tests") as pbar:
        for _ in range(num_tests):
            results = []
            selected_references = random.sample(list(validated_data.keys()), num_samples)
            for reference in selected_references:
                validated_sentence = validated_data.get(reference, '')
                hypothesis = speech_recognition_data.get(reference, '')

                # Calculate metrics
                wer = calculate_wer(validated_sentence, hypothesis)
                cer = calculate_cer(validated_sentence, hypothesis)
                word_accuracy = calculate_word_accuracy(validated_sentence, hypothesis)
                ser = calculate_sentence_error_rate(validated_sentence, hypothesis)
                f1_score = calculate_f1_score(validated_sentence, hypothesis)

                # Append the metric values to results only if hypothesis is not None
                if hypothesis is not None:
                    results.append([wer, cer, ser, word_accuracy, f1_score])
                pbar.update(1)

            total_results.extend(results)

    # Calculate averages
    metric_sums = [sum([value if value is not None else 0 for value in metric_values]) for
metric_values in zip(*total_results)]
    metric_counts = [len([value for value in metric_values if value is not None]) for metric_values
in zip(*total_results)]
    avg_results = [metric_sum / metric_count if metric_sum is not None and metric_count != 0 else
None for metric_sum, metric_count in zip(metric_sums, metric_counts)]

    return avg_results


# Usage
script_directory = os.path.dirname(os.path.abspath(__file__))
validated_tsv_file = os.path.join(script_directory, 'New-Val-Ref-LONGFORM-20k.tsv')
speech_recognition_tsv_file = os.path.join(script_directory, 'transcriptions.tsv')

# Run the test 10 times and calculate averages
num_tests = 1
num_samples = 5000
avg_results = compare_transcriptions(validated_tsv_file, speech_recognition_tsv_file,
num_samples=num_samples, num_tests=num_tests)

# Save averages to a file in the same directory as the script
results_file = os.path.join(script_directory, 'results.tsv')

with open(results_file, 'w', newline='') as file:
    writer = csv.writer(file, delimiter='\t')
    writer.writerow(['Metric', 'Average'])
    writer.writerow(['Word Error Rate (WER)', f'{avg_results[0] * 100:.2f}%'])
    writer.writerow(['Character Error Rate (CER)', f'{avg_results[1] * 100:.2f}%'])
    writer.writerow(['Sentence Error Rate (SER)', f'{avg_results[2] * 100:.2f}%'])
    writer.writerow(['Word Accuracy', f'{avg_results[3] * 100:.2f}%'])
    writer.writerow(['F1-Score', f'{avg_results[4]:.4f}'])

print("Results saved successfully!")
```

LMT2 Model Testing Script

```
'''
Author: Sabian Hibbs
University of Derby
United Kingdom, England

Licence MIT
'''

from transformers import pipeline
import torch
import argparse
import librosa as lr
import warnings
import os
import csv
from tqdm import tqdm

pipe = pipeline("automatic-speech-recognition", model='./Validation', device=torch.device("cuda:0"
if torch.cuda.is_available() else "cpu"))
tokenizer = pipe.tokenizer

def transcribe(audio):
    out = pipe(audio)
    return out["text"]

def LoadAudio(file_path):
    x, sr = lr.load(file_path, sr=16000)
    return x

def split_audio(audio, duration):
    sample_rate = 16000
    audio_duration = len(audio) / sample_rate
    num_segments = int(audio_duration / duration)
    segments = []
    for i in range(num_segments):
        start = int(i * duration * sample_rate)
        end = int((i + 1) * duration * sample_rate)
        segment = audio[start:end]
        segments.append(segment)

    # Check if there is remaining audio that doesn't fit into a full segment
    remaining_audio = audio[num_segments * duration * sample_rate:]
    if len(remaining_audio) > 0:
        segments.append(remaining_audio)

    return segments

warnings.filterwarnings("ignore")
folder_path = 'E:/TestEnvironment/cv-corpus-12.0-delta-2022-12-07-en/en/AudioFiles'
output_folder = './Logs'
segment_duration = 29  # Segment duration in seconds

# Create the output folder if it doesn't exist
os.makedirs(output_folder, exist_ok=True)

# List all files in the input folder
files = os.listdir(folder_path)

# Prepare the output TSV file
tsv_file = os.path.join(output_folder, 'transcriptions.tsv')
with open(tsv_file, 'w', encoding='utf-8', newline='') as f:
    writer = csv.writer(f, delimiter='\t')
    writer.writerow(['File_Name', 'TS_Data'])  # Write the headers

    # Process each file and write to the TSV file
    for file in tqdm(files, desc="Transcribing files", unit="file"):
        # Construct the full path of the audio file
        audio_path = os.path.join(folder_path, file)
```

```
        # Load the audio
        audio = LoadAudio(audio_path)

        # Calculate the actual duration of the audio in seconds
        audio_duration = len(audio) / 16000

        # Split the audio into segments
        audio_segments = split_audio(audio, segment_duration)

        # Transcribe each audio segment
        transcripts = []
        for i, segment in enumerate(audio_segments):
            segment_transcript = transcribe(segment)
            transcripts.append(segment_transcript)

        # Merge the transcripts from all segments
        full_transcript = " ".join(transcripts)

        # Write the file name and transcription to the TSV file
        writer.writerow([file, full_transcript])

print("Transcription completed. TSV file created.")
```

Model Testing and Automation Script 1 od 2

```
'''
Author: Sabian Hibbs
University of Derby
United Kingdom, England

Licence MIT
'''

import os
import torch
import subprocess

if torch.cuda.is_available():
    device = torch.device("cuda")
    print("CUDA device name:", torch.cuda.get_device_name(0))  # Prints the name of the first GPU
    print("CUDA device count:", torch.cuda.device_count())  # Prints the number of available GPUs
else:
    print("CUDA is not available.")

models_directory = "./Models"
initial_directory = os.getcwd()  # Store the initial working directory

# Get a list of subdirectories inside the Models directory
subdirectories = next(os.walk(models_directory))[1]

# Iterate through each subdirectory
for subdirectory in subdirectories:
    subdirectory_path = os.path.join(models_directory, subdirectory)

    print(f"Entering subdirectory: {subdirectory}")

    # Change to the subdirectory
    os.chdir(subdirectory_path)

    # Run the TEST.py script
    os.system("python TRSC.py")

    print(f"Finished running TRSC.py in subdirectory: {subdirectory}")

    # Return to the initial working directory (Models)
    os.chdir(initial_directory)

print("All subdirectories processed.")
print("Processing Metrics")

models_folder = "./Models"
os.chdir(models_folder)

for root, dirs, files in os.walk(".", topdown=True):
    for name in dirs:
        subfolder_path = os.path.join(root, name)

        logs_folder = os.path.join(subfolder_path, "Logs")
        if not os.path.isdir(logs_folder):
            continue

        script_path = os.path.join(logs_folder, "Metric-calc.py")
        subprocess.run(["python", script_path], shell=True)
```

Model Testing and Automation Script 2 of 2

```
'''
Author: Sabian Hibbs
University of Derby
United Kingdom, England

Licence MIT
'''

import os
import deepspeech
import scipy.io.wavfile as wav

from tqdm import tqdm

def read_wave_file(audio_file):
    # Read the WAV file using scipy.io.wavfile
    sample_rate, audio_data = wav.read(audio_file)

    # Convert audio data to 16-bit PCM format
    if audio_data.dtype != "int16":
        audio_data = (audio_data * 32767).astype("int16")

    return audio_data, sample_rate

def transcribe_audio(audio_file_path, model_path, scorer_path, sample_rate):
    # Read audio data from the file
    audio_data, _ = read_wave_file(audio_file_path)

    # Create a DeepSpeech model
    model = deepspeech.Model(model_path)

    # Perform the speech-to-text transcription
    text = model.stt(audio_data)

    return text

def main():
    audio_folder = "E:/TestEnvironment/GenericASR_Testing/TestAudio"  # Replace with the path to
your audio folder
    model_path = "E:/TestEnvironment/GenericASR_Testing/deepspeech-0.9.3-models.pbmm"  # Replace
with the path to the downloaded model
    scorer_path = "E:/TestEnvironment/GenericASR_Testing/deepspeech-0.9.3-models.scorer"  # Replace
with the path to the downloaded scorer

    # Get a list of audio files in the folder
    audio_files = [os.path.join(audio_folder, file) for file in os.listdir(audio_folder) if
file.endswith(".wav")]

    # Initialize the TQDM progress bar
    pbar = tqdm(total=len(audio_files))

    # Transcribe each audio file and save the transcriptions to the .tsv file
    with open("ASR_Out.tsv", "w", encoding="utf-8") as tsv_file:
        tsv_file.write("path\tsentence\n")
        for audio_file_path in audio_files:
            file_name = os.path.basename(audio_file_path)
            sample_rate = wav.read(audio_file_path)[0]
            transcript = transcribe_audio(audio_file_path, model_path, scorer_path, sample_rate)
            tsv_file.write(f"{file_name}\t{transcript}\n")
            pbar.update(1)

    pbar.close()

if __name__ == "__main__":
    main()

# E:/TestEnvironment/GenericASR_Testing/TestAudio
```

Dataset Standardisation Script 1 of 3

```
'''
Author: Sabian Hibbs
University of Derby
United Kingdom, England

Licence MIT
'''

import os
import csv
from tqdm import tqdm

tsv_file = 'validated.tsv'  # Replace with the path to your TSV file
tsv_file1 = 'new-val.tsv'  # Replace with the path to your new TSV file
folder_path = 'clips'  # Replace with the path to the folder containing the files

# Step 1: Read the TSV file and extract the 'path' column
paths_to_keep = set()
with open(tsv_file, 'r', encoding='utf-8') as file:
    reader = csv.DictReader(file, delimiter='\t')
    for row in reader:
        path = row['path']
        paths_to_keep.add(path)

# Step 2: Remove files in the folder that are not referenced in the TSV file
removed_files = []
for filename in tqdm(os.listdir(folder_path), desc='Removing Files'):
    file_path = os.path.join(folder_path, filename)
    if os.path.isfile(file_path) and filename not in paths_to_keep:
        os.remove(file_path)
        removed_files.append(filename)

# Step 3: Edit the TSV file to keep only the 'path' and 'sentence' columns
tsv_output_file = 'new-val.tsv'  # Replace with the desired output file name
with open(tsv_file, 'r', encoding='utf-8') as input_file, open(tsv_output_file, 'w', newline='',
encoding='utf-8') as output_file:
    reader = csv.reader(input_file, delimiter='\t')
    writer = csv.writer(output_file, delimiter='\t')
    header = next(reader)
    path_index = header.index('path')
    sentence_index = header.index('sentence')
    writer.writerow(['path', 'sentence'])
    for row in reader:
        path = row[path_index]
        sentence = row[sentence_index]
        writer.writerow([path, sentence])

print("File parsing and filtering completed.")
#print("Removed files:", removed_files)

# Step 1: Read the TSV file and extract the 'path' column
paths_in_tsv = set()
with open(tsv_file1, 'r', encoding='utf-8') as file:
    reader = csv.DictReader(file, delimiter='\t')
    for row in reader:
        path = row['path']
        paths_in_tsv.add(path)

# Step 2: Get the list of files in the folder
files_in_folder = os.listdir(folder_path)

# Step 3: Check for files in the folder that are not referenced in the TSV file
unreferenced_files = [filename for filename in files_in_folder if filename not in paths_in_tsv]

# Step 4: Print the list of unreferenced files
if unreferenced_files:
    print("Unreferenced files found in the folder:")
    for file in unreferenced_files:
        print(file)
```

```
else:
    print("No unreferenced files found in the folder.")

'''
# Step 1: Read the TSV file and extract the 'path' column
paths_to_keep = set()
with open(tsv_file1, 'r', encoding='utf-8') as file:
    reader = csv.DictReader(file, delimiter='\t')
    for row in reader:
        path = row['path']
        sentence = row['sentence']
        if any(char in sentence for char in "',‘\\?'\""):
            file_path = os.path.join(folder_path, path)
            if os.path.isfile(file_path):
                os.remove(file_path)
        else:
            paths_to_keep.add(path)

# Step 2: Edit the TSV file to keep only the 'path' and 'sentence' columns and remove files with
invalid sentences
tsv_output_file = 'new-validated.tsv'  # Replace with the desired output file name
with open(tsv_file1, 'r', encoding='utf-8') as input_file, open(tsv_output_file, 'w', newline='',
encoding='utf-8') as output_file:
    reader = csv.reader(input_file, delimiter='\t')
    writer = csv.writer(output_file, delimiter='\t')
    header = next(reader)
    path_index = header.index('path')
    sentence_index = header.index('sentence')
    writer.writerow(['path', 'sentence'])
    for row in reader:
        path = row[path_index]
        sentence = row[sentence_index]
        if path in paths_to_keep:
            writer.writerow([path, sentence])

print("File parsing and filtering completed.")
'''
```

Dataset Standardisation Script 2 of 3

```
'''
Author: Sabian Hibbs
University of Derby
United Kingdom, England

Licence MIT
'''

import os
import csv
from tqdm import tqdm

tsv_file = 'validated.tsv'  # Replace with the path to your TSV file
tsv_file1 = 'new-val.tsv'  # Replace with the path to your new TSV file
folder_path = 'clips'  # Replace with the path to the folder containing the files

# Step 1: Read the TSV file and extract the 'path' column
paths_to_keep = set()
unique_sentences = set()  # Track unique sentences
with open(tsv_file1, 'r', encoding='utf-8') as file:
    reader = csv.DictReader(file, delimiter='\t')
    for row in tqdm(reader, desc="Processing rows", unit="row"):
        path = row['path']
        sentence = row['sentence']
        if (
            any(char in sentence for char in "',!-;£$%^&*()|~@→‘\\?'\"")
            or len(sentence) < 5  # Check if the sentence is less than 5 characters long
            or len(sentence) > 15 # Check if the sentence is more than 15 characters long
            or sentence.count('.') > 2  # Check if the sentence has more than 2 full stops
            or sum(1 for c in sentence if c.isupper() and c.isalpha()) > 1  # Check if the sentence
has more than 1 capital letter
            or sentence in unique_sentences  # Check if the sentence has already been encountered
        ):
            file_path = os.path.join(folder_path, path)
            if os.path.isfile(file_path):
                os.remove(file_path)
        else:
            paths_to_keep.add(path)
            unique_sentences.add(sentence)

# Step 2: Edit the TSV file to keep only the 'path' and 'sentence' columns and remove files with
invalid sentences
tsv_output_file = 'New-Val-Ref-SHORTFORM.tsv'  # Replace with the desired output file name
with open(tsv_file1, 'r', encoding='utf-8') as input_file, open(tsv_output_file, 'w', newline='',
encoding='utf-8') as output_file:
    reader = csv.reader(input_file, delimiter='\t')
    writer = csv.writer(output_file, delimiter='\t')
    header = next(reader)
    path_index = header.index('path')
    sentence_index = header.index('sentence')
    writer.writerow(['path', 'sentence'])
    for row in tqdm(reader, desc="Writing rows", unit="row"):
        path = row[path_index]
        sentence = row[sentence_index]
        if path in paths_to_keep:
            writer.writerow([path, sentence])

print("File parsing and filtering completed.")
```

Dataset Standardisation Script 3 of 3

```
'''
Author: Sabian Hibbs
University of Derby
United Kingdom, England

Licence MIT
'''

import random
import csv
import tqdm

tsv_file = 'new-val-ref-15-30.tsv'  # Replace with the path to your TSV file
output_file = 'new-val-ref-15-30-5k.tsv'  # Replace with the desired output file name

# Step 1: Read the TSV file and extract all the lines
lines = []
with open(tsv_file, 'r', encoding='utf-8') as file:
    reader = csv.reader(file, delimiter='\t')
    header = next(reader)
    lines = list(reader)

# Step 2: Randomly select 5,000 lines with tqdm progress monitoring
selected_lines = random.sample(lines, 5000)

# Step 3: Save the selected lines in the new TSV file with tqdm progress monitoring
with open(output_file, 'w', newline='', encoding='utf-8') as file:
    writer = csv.writer(file, delimiter='\t')
    writer.writerow(header)
    for line in tqdm.tqdm(selected_lines, desc="Saving lines", unit="line"):
        writer.writerow(line)

print("Random selection and saving completed.")
```

Huggingface Dataset Downloader (External Downloader)

```
'''
Author: Sabian Hibbs
University of Derby
United Kingdom, England

Licence MIT
'''

import os
from datasets import load_dataset

# Get the directory of the current script
script_dir = os.path.dirname(os.path.realpath(__file__))

# Specify the name of the dataset
dataset_name = 'mozilla-foundation/common_voice_13_0'

# Replace 'your-token' with the token you got from HuggingFace's website
dataset = load_dataset(dataset_name, 'en', data_dir=script_dir, use_auth_token='HuggingFace_Token')
```

END OF PAPER