A comparison of Wav2Vec 2.0 and humans in handling frequency shifted speech: A Qualitative Analysis

Nilansha Dargan
Student number: 13130366

BA Thesis Linguistics
University of Amsterdam
Supervisor: Prof. Dr. P.P.G. (Paul) Boersma
June 2023

Table of Contents

Abstract

With the current highs of AI, the exploration of speech recognition has become a big part of the field of Natural Language Processing, with attention also given to exploring the quantitative differences between human and AI speech model performance. Building on the foundation of Adolfi et al. (2023) and the time domain perturbed speech, the current study aimed to look into how the AI language model, particularly Wav2Vec2.0, qualitatively performs with the manipulated speech in the frequency domain as compared to humans and if there is a convergence in their performance curve with different manipulation conditions, while some of Adolfi's et al. manipulations were also replicated. It was found that the qualitative performance curve of wav2vec2.0 for each manipulation is consistent with that of humans. However, it should be noted that the study utilised only ten audio samples, which represented a limitation. Therefore, a future recommendation is made to conduct a comparative analysis between human and AI models using a larger sample size and a standardised experimental setup.

1. Introduction

Speech, the evolutionary trait that is uniquely present in humans, serves as an intricate and remarkable tool for communication. The multiplex system present for speech processing and production is what allows humans to master it, even with the distinct variety it appears with in the world (Pagel, 2017). What is fascinating to researchers about speech in humans is not only its idiosyncratic nature but also the robustness with which it is produced and comprehended. *Speech perception* is the skill with which language is comprehended and, thus, is fundamental for language development. The use of multiple mental processes, like memory, analytical reasoning, attention to detail, et cetera, altogether makes it possible to accomplish the complexities, like ambiguity and variety in meaning by change of context, with which speech emerges and thus, achieve speech perception. With all this mental power that goes into this process, speech processing, therefore, serves as a window into various facets of human intelligence and cognitive abilities. (Heald & Nusbaum,2014).

With the recent progression that Artificial Intelligence, which is "the simulation of human intelligence process performed by a machine or particularly computer systems" (Burns, 2023), has experienced, this attempt to replicate human intelligence and the processes it can achieve has also made its stride to recreate the mental process of language processing. This niche field of AI is called Natural Language Processing (NLP). Speech perception in this area has gotten its own name, Automatic Speech Recognition (ASR). This process is achieved with the help of Deep Neural Networks (DNNs), which are created through the stacking of neural networks and the presence of a hidden layer, to some extent, an attempt to recreate the neural network in the human brain (Aouichaoui, A. R. et al., 2021). This age of ASR DDNs has also seen its own transformations. There have been many carefully engineered architectures used in Automatic Speech Recognition, such as convolutional (O'Shea & Nash, 2015), recurrent (Amodei et al., 2015; Hannun et al., 2014), and Transformer (Baevski, Zhou, Mohamed, & Auli, 2020; Schneider, Baevski, Collobert, & Auli, 2019).

As this field sees its enhancements, the robustness of AI with speech recognition is also checked compared to Human speech comprehension. Recent studies have also moved away from the quantitative-centric research and started to qualitatively examine trends of AI ASR (Like Long's (1990), U-shaped learning of a second language has been a trend that is observed in human speech) and their similarities to the trends, for the same conditions, seen in human speech comprehension. The following section will highlight some studies that help understand this quantitative-to-qualitative transition.

1.1. Background

Although former research had established that humans consistently outperform ASR models in speech recognition, with consideration to the recent developments in the field of NLP. Spille and Meyer (2018) conducted an investigation of the difference in the quantitative performance of deep neural networks (DNNs) automatic speech recognition systems and standard human speech recognition with regard to simple and complex speech signals; in order to investigate the gap between ASR and human speech recognition and suggest further improvements in the DNNs. They made use of simple one-channel audio with stationary speech and a multi-talker babble noise. And also, in contrast, complex audios had what the authors called 'multi-channel scenes', which included diffused sounds and moving talkers, in order to reach their aim.

The study found that, overall, human speech recognition did perform better than ASR at all times. It was observed that in simple audio, ASR systems had reached the level of human speech recognition; the same cannot be said for complex audio recognition. Furthermore, ASR always performs worse in this aspect.

Furthermore, the study by Spille and Meyer (2018) emphasised the significance of incorporating multi-channel scenes and diffused sounds into the evaluation of ASR systems. By simulating realistic audio environments with moving talkers and complex acoustic backgrounds, the researchers provided insights into the limitations of current ASR models. These findings indicate the need to enhance ASR algorithms to handle better real-world scenarios where multiple sources of speech and environmental noise coexist. By addressing the challenges posed by such complex audio, ASR systems can strive towards achieving uniformity with human speech recognition across an extensive range of conditions. The study by Spille and Meyer serves as a valuable reference for future research endeavours aimed at refining ASR technology and narrowing the gap between machine and human speech perception.

Another study looked deeper into the aspect of the new advancement of AI. Millet & Dunbar (2021) focused on exploring if self-supervised models, without the presence of any form of their own representation spaces, are similar to humans when initially learning a language. The focus was to investigate if self-supervised models also show perception bias, the specialisation of perceptual differences encountered in the native language that weakens the ability to discriminate the perceptual differences present in other languages, and not present in the native language. In the experiment, ASR machines were trained in one language, and their output was observed with the stimulus of another language. It was studied if their output is similar to what you see in human speech recognition output. They used perceptual spaces of French and English human speakers as a comparison.

It was found that, unlike humans who show a clear perceptual bias or perceptual magnetic effect (Kuhl, 1990), the self-supervised models were able to learn the representation spaces of the new sounds in the new language. They were good at capturing detailed perceptual patterns and presenting language-neutral characteristics. Models were more effective in capturing broader, language-related effects rather than how people perceive sound and show magnetism toward familiar ones. These findings strongly indicated that the advantages of self-supervised speech models do not just learn specific features of audio but rather process general features present in the audio available to them.

The previous studies have provided valuable insights into the quantitative and qualitative evaluation of automatic speech recognition (ASR) models using natural speech. However, it is essential to note that humans also exhibit high recognition performance with perturbed speech, as extensively studied in the field of human speech comprehension. Building upon this understanding, the subsequent investigation by Adolfi et al. (2023) explores a range of signal perturbations, laying the conceptual groundwork for the current study. By examining the effects of these perturbations on speech recognition, this research aims to establish a solid foundation for its ensuing exploration.

1.2. Introduction: Adolfi et al. (2023)

Adolfi et al. (2023) recognised that there is variation in performance between natural and artificial speech recognition when analysing the qualitative results of the same task. But also, that, narrowing the variety of tasks can lead to resonance in the type of results seen in these two recognition systems. The study focused on qualitative conversion, which explores the trend in performance, rather than a quantitative one, which measures how well the system performs. This recognition called for a closer inspection of the qualitative convergence, which was investigated by examining the robustness of state-of-the-art neural networks in speech recognition when encountering different Spectro-temporal granularities.

1.2.1. Methodology: Adolfi et al. (2023)

Adolfi et al. conducted a comprehensive investigation involving eight different manipulations applied to audio signals. These manipulations can be described as follows:

Firstly, there is the technique of shuffling, which involves the disruption of audio samples given a specific time window which leads to loss of coherence, in the temporal domain, at a larger scale. Secondly, reversing involves the local reversal of the signal, resulting in frame-wise reversed speech over time. The third manipulation, known as masking, entails adding background noise to the original signal, effectively introducing a layer of interference.

Silencing, the fourth manipulation, involves inserting periods of silence in places where speech signals were originally present. Chimaera, the fifth technique, combines the slow amplitude modulation of one sound with the rapid carriers of another sound. This process results in the creation of chimeric audio, involving the factorisation of envelopes and fine structuring of signals.

The sixth manipulation, known as mosaic, involves altering the Spectro-temporal resolution of speech signals by manipulating the coarse-graining of time-frequency bins. This systematic alteration leads to mosaicized sounds. Time warping, the seventh technique, involves modifying the speed of speech without affecting its pitch. This is achieved through temporal compression or stretching in the time-frequency domain.

Finally, Repackaging, this technique redistributes the original chunks of samples in the temporal domain while maintaining the pitch of the speech signal. Additionally, a period of silence is concatenated with the repackaged signal.

In summary, Adolfi’s investigation encompasses a diverse range of manipulations applied to audio signals, including shuffling, reversing, masking, silencing, chimaera synthesis, mosaicization, time warping, and repackaging. These manipulations serve to explore the several ways in which audio signals can be transformed and manipulated, contributing to a deeper understanding of signal processing and its impact on speech perception.

The stimuli of each of these manipulations were concatenated together and presented to the systems in the form of waveforms or spectrograms, depending on the type of input the model takes. The word "rate error" was used as the evaluation metric for the performance of the neural network, which was calculated by dividing the sum of substitutions, deletions, and insertions by the sum of substitutions, deletions, and correctly recognised words. The study looked into four speech recognition models, Deepspeech (LSTM), Wav2vec2.0 (1DConv-TF), Fairseq-s2t (2DConv-TF), Silero (Sep-2DConv).

1.2.2. Results: Adolfi et al. (2023)

Adolfi et al. found that with specific manipulations, which are of a less destructive nature, such as reversal, shuffling, time warping, chimerizing, and mosacizing performance patterns observed were similar, to some extent, to those seen in human studies for the same perturbations. On the contrary, manipulations of a more complex nature or the ones that were more destructive, such as masking and silencing, showed a disparity in the type of performance for humans and the models used except for with the wav2vec2.0 model, where a similar pattern, to humans, for these signal destructing manipulations was seen.

One peculiar and informative outcome was for the repacking manipulation experiment, where all models ultimately failed to capture the human study performance results. This finding highlights a systematic failure of models to recover perceptual performance in certain situations or conditions in which humans almost always excel. This experiment brings to attention the significant difference between ASR and human comprehension in the perceptual domain.

Finally, the research points away from the nominal quantitative differences between ASR and human beings to focus on the qualitative drawbacks of further achieving human-like perception. Adolfi et al. suggests that the models need more flexibility in task performance by effectively utilising alternative spectral and temporal scales. Merely introducing different training approaches or increasing model capacity would not address these qualitative differences. Instead, substantial architectural modifications would be necessary to overcome these limitations. The identified qualitative differences provide insights into potential architectural constraints and improvements, indicating areas for further model development and comparison.

The most significant observation is the phenomenon of repackaging, where all models consistently fail to capture human behaviour. This effect opens up alternative avenues for theorising, computational cognitive modelling, and improving engineering solutions. By exploring the reasons behind this failure, researchers can gain a deeper understanding of the architectural modifications required to bridge the gap between model performance and human perception.

In summary, the study reveals that specific perturbations elicit similar performance patterns among models and align with human perception, while other perturbations highlight disparities. The repackaging experiment explicitly demonstrates a systematic failure of all models to capture human performance. The qualitative differences observed suggest that substantial architectural modifications are necessary to address these shortcomings. This research provides valuable insights for further model development and offers alternative directions for theorising and computational cognitive modelling.

1.3. Gap In the Last Studies

Adolfi et al. offered a linguistic lens to the study of the performance of AI. Although it includes different types of perturbed speech; most of the perturbations consisted of manipulation of the time domain, leaving the frequency domain untouched. Therefore, leaving an unfulfilled gap of lack of frequency domain change exploration, which has been, previously, investigated in humans.

1.4. Present Study

The present study is an adaptation of Adolfi et al.'s paper and tries to bridge the gap left by Adolfi et al.'s research, exploring frequency domain changes. Although the study builds on Adolfi et al.'s conceptual foundation, including some of the audio manipulations included in the paper, it is not a direct replication due to not having completely similar methodologies.

The specific research question was: How does the qualitative performance of the wav2vec2.0 model compared to that of humans when recognising frequency-shifted speech?

2. Methodology

2.1. Model: Wav2Vec2.0

This study uses the hugging face model Wav2Vec2.0, which is also used in the Adolfi et al. study, the paper that gives the conceptual foundation of the current study. The model Wav2Vec2.0, which changes raw waveforms into a floating array of vectors, was first proposed in the paper "wav2vec 2.0: A Framework for Self-Supervised Learning of Speech Representations" by Alexei Baevski, Henry Zhou, Abdelrahman Mohamed, and Michael Auli (Baeyski et al., 2020). This section will elaborate on the architecture, training and working of the model and specify how it was used in the current study.

2.1.1. Architecture

Wav2Vec2.0 is a self-supervised model with a transformer architecture that makes use of the Self-attention mechanism for its working.

2.1.2. What are Transformers?

Transformer in the context of machine learning and artificial intelligence refers to a type of model architecture used in natural language processing tasks. This architecture was introduced by Vaswani et al. in a paper titled "Attention is All You Need" in 2017. Transformers are based on the concept of "attention mechanisms", where the model learns to weigh the importance of different inputs in the sequence rather than processing them independently or in a fixed order like in previous recurrent neural networks (RNNs) and convolutional neural networks (CNNs). This approach allows transformers to handle long-range dependencies between words in a text more effectively. They are particularly powerful in tasks like translation, summarisation, and generation of text.

2.1.3. What is Self-Supervised?

Self-supervised learning is a type of machine learning where the data provides the supervision itself, meaning that labels are generated from the input data instead of relying on explicitly provided labels. It is a form of unsupervised learning but with a twist. The model is given a pretext task, such as predicting the next word in a sentence (in the case of language models like GPT-3) or predicting a masked part of an image (in the case of some computer vision tasks). This process allows the model to learn to use representations of the data. Once trained, the model can be fine-tuned for specific tasks using a smaller amount of labelled data.

Unsupervised learning and self-supervision share similarities in their approach, but they differ in terms of the tasks they address. Unsupervised learning is primarily utilised for clustering tasks, where the absence of data labels poses a challenge. Conversely, self-supervision focuses on training algorithms typically employed in supervised learning, such as classification and regression, without providing explicit labels.

2.1.4. What is Self-Attention?

Self-attention, also known as intra-attention, is a mechanism used in Transformer models that allows each element in a sequence (e.g., each word in a sentence) to look at every other element to gather contextual information.

Self-attention calculates the relevance score between all pairs of input elements. Then, it uses these scores to weigh the impact of different elements when generating a new representation for a given element. This calculation enables the model to consider the context of each word in the sentence.

More formally, self-attention works by applying three matrices (learned during training) to the input vector of each word: the Query, Key, and Value matrices. It then computes the dot product of the Query and Key vectors of each word, applies a SoftMax function to get the weights (attention scores), and multiplies these weights with the Value vectors to get the output.

The power of the self-attention mechanism comes from its ability to focus on distinct parts of the input sequence when encoding a specific word, allowing for the modelling of complex dependencies.

An illustrative example that sheds light on this concept is the task of sentence translation, specifically when faced with a sentence like "The animal could not cross the road because it was slippery." In this case, the pronoun 'it' can be interpreted as referring to either the animal or the road. The self-attention mechanism plays a vital role in enabling a model to discern the accurate attribution of 'it' by leveraging contextual information. By employing self-attention, the model can effectively capture the dependencies and relationships between words, thereby making informed decisions regarding the intended referent of 'it'.

2.2. Training

The basic Wav2Vec2.0 model is trained on 960 hours (about one and a half months) of unlabeled data from the *LibriSpeech* Dataset.

2.3. Wav2Vec2CTCTokenizer

The Wav2Vec2CTCTokenizer is the fine-tuned version of the Wav2Vec for automatic speech recognition of English. This Tokenizer helps map the sequence of the wav2vec2 output to its corresponding transcription. (Platen, 2021)

2.4. Methodology

The research framework employed in this study builds upon an open-source codebase provided by the authors of the previous paper, allowing for a robust and reliable foundation. Extending the existing code from Adolfi et al., several selective steps were meticulously replicated and further modified to enhance functionality and address specific research objectives. In alignment with the preceding work, this paper primarily focuses on the qualitative evaluation of the model's performance. However, it uniquely centres on the recreation of repackaged, masked and silencing speech signals. By employing advanced techniques and adaptations from existing methods, these recreated signals were used as input for the Wav2Vec2.0 model, facilitating a thorough qualitative assessment of its performance. The significance of qualitative evaluation lies in its ability to offer valuable insights into the model's behaviour and its suitability for real-world scenarios. Consequently, this research contributes to the existing body of knowledge by improving upon the open-source code and providing a comprehensive qualitative analysis of the Wav2Vec2.0 model's performance in the context of the frequency domain-related manipulated speech signals.

2.4.1. Data Manipulation

This section presents a comprehensive overview of the data manipulation techniques employed in our study. Each technique is explained in detail, highlighting its underlying principles and the corresponding code implementation available in the Appendix. These techniques enable us to investigate the effects of various audio manipulations on speech signals, providing valuable insights into the characteristics and behaviours of the modified data.

Repackaging

Repackaging is a versatile technique that combines several audio manipulations, including time warping, multiple-timescale modification, and insertion of silence. The goal of repackaging is to modify the audio signal while preserving its underlying pitch. To achieve the said manipulation, a window of the signal is compressed by a factor of two, effectively altering its temporal characteristics. Subsequently, a specific length of silence is added to the signal, creating gaps or pauses within the audio. The analysis of this manipulation focuses on the ratio between the duration of the modified audio and the inserted silence length. This ratio, ranging from 0.5 to 2.0, allows for investigating the impact of varying degrees of compression and silence on speech perception and intelligibility. The code snippet for implementing the repackaging technique can be found in the Appendix under the 'compress_insert_mix' function.

Masking

Masking is a type of speech interruption technique that introduces interferences in the speech by adding noise masks to the signal. This manipulation aims to create glimpses of the original input signal amidst the added noise. By applying the 'mask' function from Adolfi et al.'s code, a portion of the audio signal is corrupted by the addition of noise. The length of the masked portion can be adjusted to explore various levels of speech interruption. This technique enables the study of how noise interference affects speech perception and recognition. The corresponding code snippet for implementing the masking technique is available in the Appendix.

Silencing

Silencing, similar to masking, is another type of speech interruption technique used to disturb a portion of the audio signal. However, instead of adding noise, silence is inserted into specific sections of the signal, resulting in glimpses of the original input. By utilising the same 'mask' function, silence is introduced to create interruptions in the speech signal. The length of the window portion affected by silence can be varied to examine the influence of silence interruptions on speech intelligibility. The code snippet for implementing the silencing technique can be found in the Appendix.

Shifting Frequency Up

Frequency manipulation involves changing the frequency content of a signal. To achieve this, we use an algorithm called Fast Fourier Transform (FFT), which operates in both the time domain and frequency domain.

In the time domain, the signal is represented as a waveform. The FFT algorithm isolates the frequency information that is found in a waveform, separating the different frequency components. By applying the inverse of this process, we can convert the isolated frequency back into its original form.

In the code, the SciPy library's FFT function is used, precisely the RFFT method. Usually, when a waveform is passed through the FFT function, the input is a NumPy array consisting of complex and real values. However, the RFFT method takes in as input an array of only real values, which facilitates frequency shifting.

To shift the frequency, we specify the desired frequency shift value. The shift is then calculated in bins, which determine how the frequency components will be replaced. This is calculated using the following formula:

(1)

$$Frequency\ shift\ in\ bins = \frac{shift\ frequency(Hz) * length\ of\ array\ transformed\ audio}{sampling\ rate\ of\ the\ audio(Hz)}$$

Finally, the frequency shift is performed on the transformed audio, and the RFFT output is inverted to obtain the modified waveform.

Shifting Frequency Down

Shifting the frequency down is done on the same principle and steps as mentioned in the section above. The only difference when shifting frequency down is the shift frequency is specified as a negative value. So, if you specify the shift frequency as a thousand, in frequency shifted up as:

(2)

$$shift\ frequency = 1000\ \text{Hz}$$

It would be specified for shifting frequency down by a thousand as follows:

(3)

$$shift\ frequency = -1000\ \text{Hz}$$

2.4.2. Analysis

In evaluating the performance of various manipulations, each manipulated metric was distinct for the manipulation. However, the models' overall performance was determined using a standard metric called Word Error Rate (WER). The WER metric takes into account different types of measures, including deletions (D), substitutions (S), correct words (C), and insertions (I). These error types are calculated and then combined in a formula to find out the overall performance of the Wav2Vec2.0 model.

The formula for calculating the WER is as follows:

(4)

$$WER = \frac{S + D + I}{S + D + C}$$

A higher WER score indicates a more significant number of errors and thus indicates the poorer performance of the Wav2Vec2.0 model. To evaluate the performance of the model, it was tested on the first ten audio samples from the test set of *LibriSpeech*, as described by Panayotov et al. (2015). It is worth noting that although the Wav2Vec2.0 model was trained on 960 hours of the *LibriSpeech* dataset, it had not been exposed to the specific test set used in this evaluation. This ensures the integrity of the testing process and avoids any potential bias or overfitting issues.

By employing the WER metric and testing the model on unseen data, researchers can effectively assess the performance of the Wav2Vec2.0 model and compare it to other models or variations that underwent different manipulations. The goal is to identify the model with the lowest WER score, indicating superior performance in accurately transcribing speech and minimising errors.

3. Results

This section presents an overview of the manipulation types investigated in this study, focusing on their performance evaluation through Word Error Rate (WER). Furthermore, the qualitative outcomes of relevant human studies are discussed to facilitate a comparative analysis of the model's qualitative performance. It is necessary to mention that the model performs poorly in a quantitative way, but this study focuses only on the qualitative manner.

3.1. Repackaged

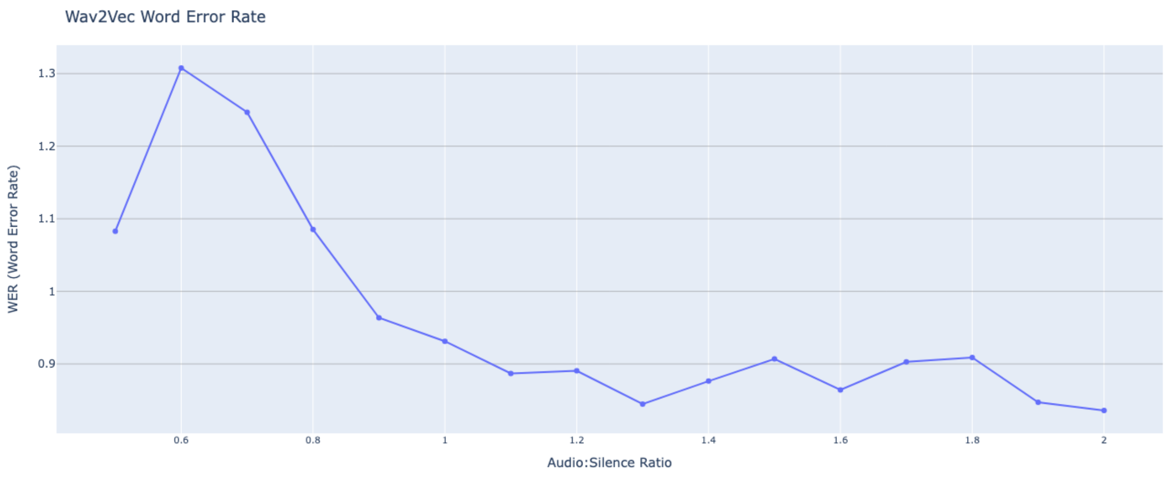


Figure 1. x Audio: Silence ratio vs WER

Repackaging manipulation refers to the repositioning of speech information in time. Previous studies conducted on both humans and machines (Fu et al., 2001) have observed a significant decline in performance once the speech signal is compressed. However, it has been found that this compression of the speech signal can be improved by strategically adding appropriate amounts of silence. Multiple studies (Bosker & Ghitza, 2018; Penn et al., 2018; Ghitza, 2012, 2014; Ghitza & Greenberg, 2009) have demonstrated that the inclusion of the right amount of silence in compressed speech leads to a recovery in performance, resulting in a U-shaped performance curve.

In the current study, a similar trend is observed, where a soft U-shaped curve is evident in *Figure 1*. Towards the end of the study, while the error rate increases in the human participants for the last Audio: Silence ratio, a subsequent decline in the error rate is observed. This suggests that carefully managing the balance between speech and silence can have a positive impact on performance.

3.2. Masked

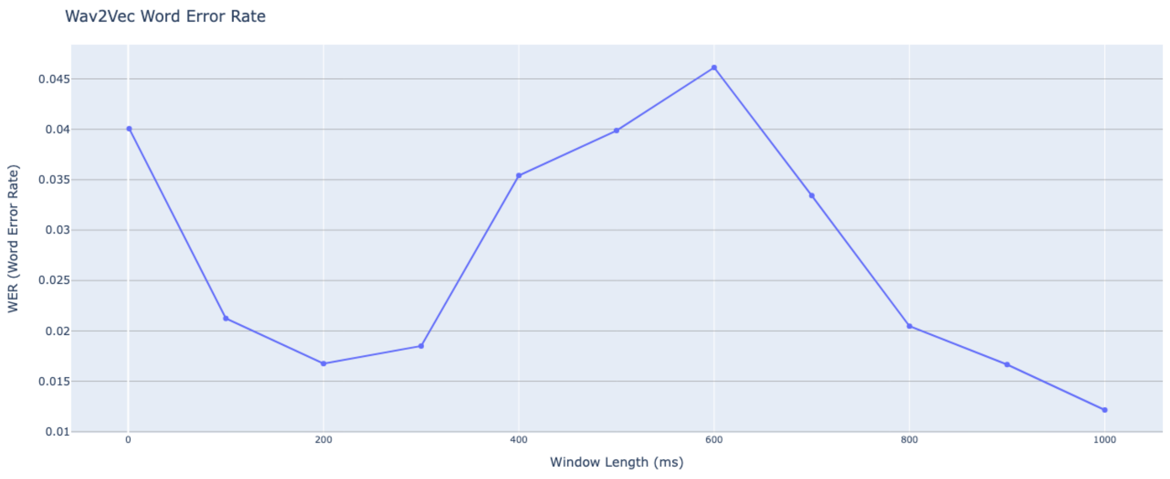


Figure 2. Window length(ms) vs WER

Masking refers to the process of introducing masking noise to corrupt speech information present in a signal. Previous studies conducted on humans (Miller & Licklider, 1950) have demonstrated that the error rate shows a notable improvement as the window length increases. Interestingly, the worst performance is consistently observed at the mid-length window values.

In the current study, Figure 2 displays a strikingly similar pattern to the findings in human studies. Performance shows a significant enhancement at longer window lengths, while the worst performance is consistently observed at the mid-length window values. This observation further supports the notion that the manipulation of window length affects speech perception, aligning with previous research findings.

3.3. Silencing

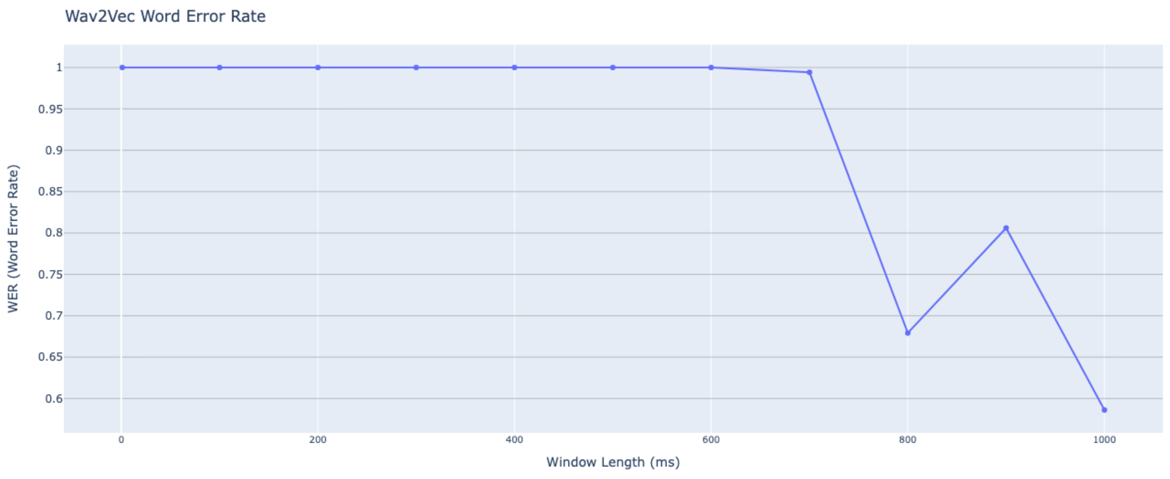


Figure 3. Window length (ms) vs WER

Silencing refers to the deletion of speech information from a speech signal. Previous human studies conducted on silence speech interruption have revealed a performance curve that closely resembles the findings from the masking study (Miller & Licklider, 1950). However, in contrast to the mid-length window values showing the worst performance in the masking study, the performance curve observed in the present study, as depicted in Figure 3, follows a similar pattern to the human study. In this case, the performance is initially poor and gradually improves over time.

The results obtained in Figure 3 reinforce the similarity between the effects of silencing on speech perception in humans and the findings from previous research. Despite the variation in the specific window length dynamics, both studies demonstrate that manipulating silence interruptions can have a significant impact on performance.

3.4. Shifting Frequency Up

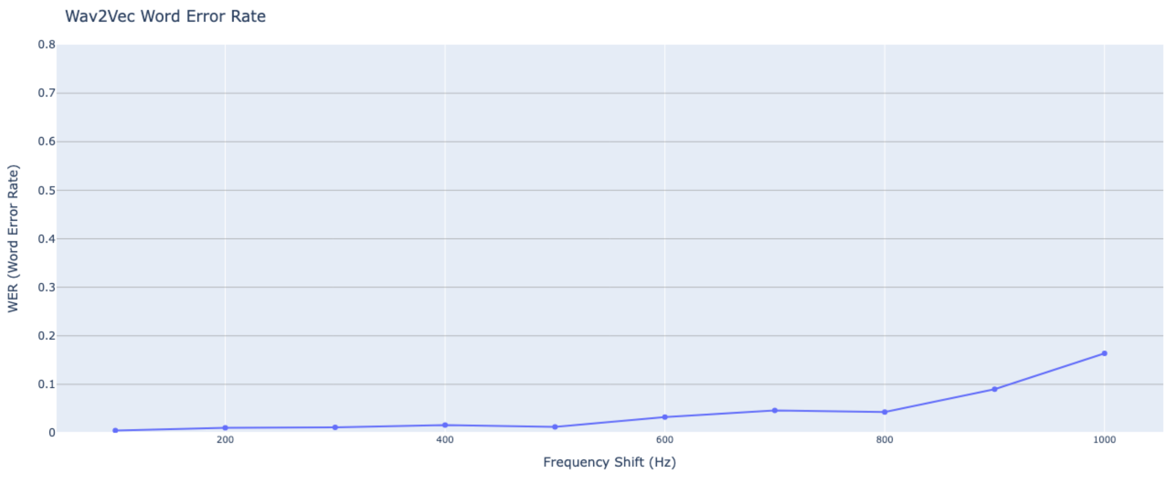


Figure 4. Frequency shift (Hz) vs WER

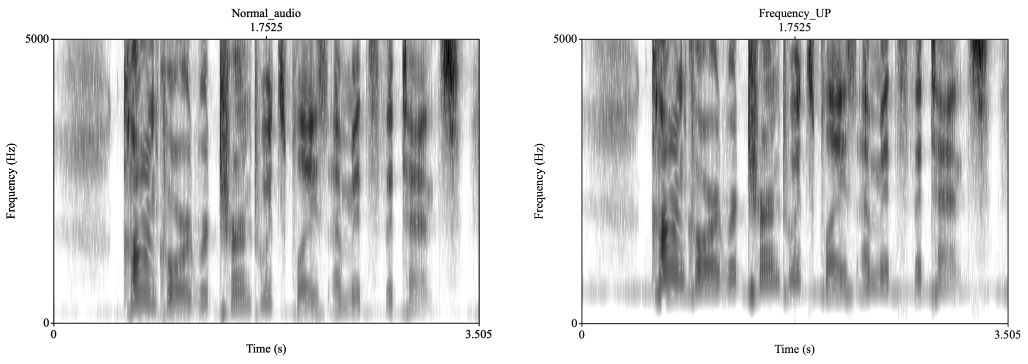


Figure 5. The spectrogram of the original audio and the Spectrogram of the audio shifted up by 1000 Hz, respectively.

When the frequency is shifted upward, previous human studies have shown that this specific type of alteration typically leads to a decrease in performance as extreme values are reached in cases where the worst performance can be observed (Shannon et al., 1995). A slightly similar trend can be observed in Figure 4, which depicts the performance of Wav2Vec2.0 across different shift values. The performance curve remains relatively stable with increasing shift values, except for the extreme value of 1000 Hz, where a sharp decline in performance is evident.

The findings depicted in Figure 4 align closely with the observations from previous human studies, reinforcing the notion that frequency shifts within a specific range do not significantly impact speech perception. However, when the shift value reaches extreme levels, it disrupts the normal perception of speech, leading to a noticeable decrease in performance.

3.5. Shifting Frequency Down

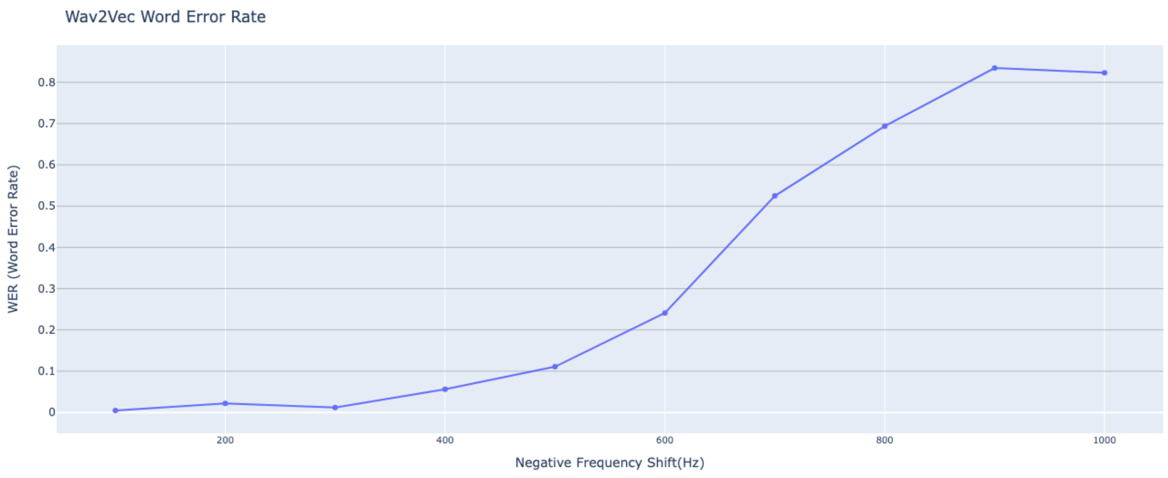


Figure 6. Negative Frequency shift (Hz) vs WER

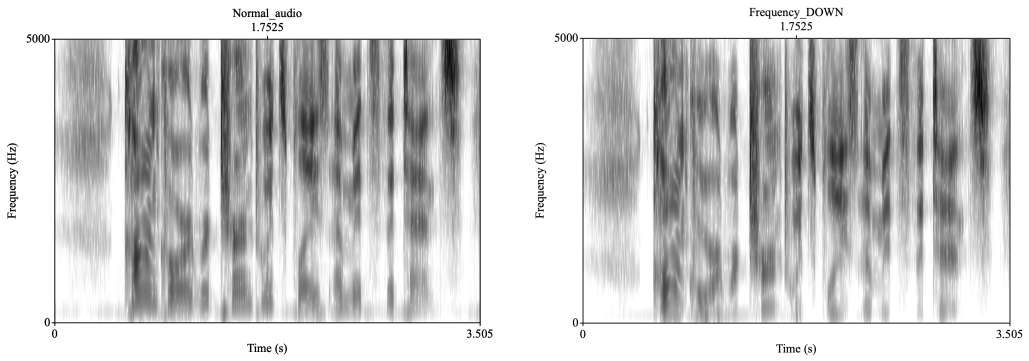


Figure 7. Spectrogram of normal audio and the Spectrogram of audio shifted down by 1000 Hz, respectively.

The study on humans (Shannon et al., 1995) does not specify any differences between performance curves of frequency being shifted upwards or downwards but rather points to a gradual decrease in performance as frequency-related information is varied. Thus, a similar performance curve to frequency-shifted up is assumed. With this assumption, the results depicted in Figure 6 are consistent with the observations made in previous research (Shannon et al., 1995). It is also observed that for this type of frequency change, higher error rates and worse performance are noticeable, even in middle values. This observation shows that downward frequency shifts within a certain range start to show a substantial impact on speech perception. And reaches the worst decline in performance at an extreme value. The performance curve starts worsening after reaching the middle value of 500 Hz., ultimately reaching the worst performance at the extreme value of 1000 Hz.

Unlike shifting the frequency upward, the current observation shows that when the frequency is shifted downward, a gradually worsening performance curve is found.

4. Discussion

In this section, we will discuss the differences between the current findings and those of previous studies, highlighting variations in methodology and suggesting potential areas for improvement. We will also address the mistakes made by Adolfi et al. in their paper and further expand on the implications of the observed performance trends. This section also conveys the answer to the research question.

Regarding two of the three manipulations adopted from the Adolfi et al. paper, namely masking and silencing, both of which involve speech interruption, the findings of the current study align with those of the previous study. However, an interesting observation can be made about the third manipulation, repackaging. In contrast to the Adolfi et al. paper, where the repackaging curve did not converge with the human performance curve, the current study observed a partial convergence. One key difference between the two studies is that while Adolfi et al. tested the entire test set of *LibriSpeech* and averaged the results, the current study averaged only the first ten audios in the dataset. Although this can be seen as a drawback, it allowed for a closer examination of the individual audio and highlighted the importance of stimulus type. It was observed that some audios exhibited a complete match with the performance curve seen in human studies, providing valuable insights into the effects of repackaging and choice of stimulus.

This brings attention to a limitation of the Adolfi et al. paper, where the same stimulus as the human studies was not employed, potentially impacting the generalizability of their findings, particularly when considering the effects of word frequency on speech recognition. Addressing this discrepancy in future studies can lead to a more comprehensive understanding of the factors influencing speech perception. That is, creating a study design where humans and AI models are tested on the same stimulus in the same condition.

Furthermore, it was noted that Adolfi et al.'s paper contains contradictions regarding the performance of Neural Networks. While they claim that the word error rate (WER) increases as silence is added, their Audio: Silence ratio actually increased in the graph shown in the paper, indicating a decrease in silence. These inconsistencies highlight the importance of thorough analysis and accurate reporting in research publications.

Moving on to the manipulations exclusively explored in the current study, namely frequency shifting (both upward and downward), it was observed that the performance of the model converged with that seen in human studies. This result indicates the model's ability to handle frequency shifts in a manner consistent with human speech perception. On the other hand, it was interesting to notice that when the frequency is shifted downward, wav2vec2.0 showed worse performance as compared to when the frequency is shifted upward. This finding becomes clearer by observing the frequencies that are lost in the two manipulations. When the frequency is shifted up, only the high frequencies that are less critical to speech perception are lost. On the other hand, when the frequency is shifted down, lower frequencies, that are fundamental to speech perception, like f0, are lost, which makes recognition of speech harder.

Overall, the qualitative performance of Wav2vec2.0 is remarkably high, displaying its potential for practical applications in the field of Natural Language Processing (NLP). Although the performance curves do not perfectly align with human studies, displaying some differences in specific values while still following a similar trend, these variations provide valuable insights that can be further investigated and improved upon. Future research can build upon these findings to refine the understanding and application of speech manipulations, ultimately enhancing the performance of speech recognition systems in various NLP tasks. Such advancements can contribute to the development of more robust and accurate speech technologies, benefiting a wide range of applications and users.

5. Conclusion

The present study builds upon previous research that has explored the convergence of the qualitative performance of neural network automatic speech recognition systems with that of human speech recognition. This convergence suggests a resemblance in the functioning of AI and human intelligence. In this study, the performance of Wav2Vec2.0 was examined in relation to perturbed or frequency-shifted speech signals, and it was found that the model performs similarly to humans in a qualitative sense. The performance curves for all five different manipulations showed a high degree of alignment.

However, it should be noted that one of the replicated manipulations, namely repackaging, exhibited a performance curve that did not align with previous research. This discrepancy emphasises the drawback of using a smaller number of audio samples compared to the foundational study, Adolfi et al. (2023). Additionally, the choice of stimulus is crucial, and the fact that the previous study did not utilise the same stimulus as human studies could be a contributing factor to the differing performance curves. Hence, it is suggested that future research should consider conducting studies involving both humans and AI models using the same stimulus for a more comprehensive analysis.

In conclusion, the current study offers valuable insights into the similarities and differences between human speech recognition and artificial speech recognition. It highlights the importance of qualitative research in understanding the behaviour of language models, emphasising that quantitative research alone may not provide a complete picture. The findings contribute to advancing our understanding of the capabilities and limitations of AI models in speech recognition tasks, laying the groundwork for further improvements and applications in the field.

6. References

Amodei, D., et al., (2016). Deep speech 2: End-to-end speech recognition in English and Mandarin. In *International Conference on Machine Learning*, 173–182. The Proceedings of Machine Learning Research.

Aouichaoui, A. R., Al, R., Abildskov, J., & Sin, G. (2021). Comparison of Group-Contribution and Machine Learning-based Property Prediction Models with Uncertainty Quantification. In *Elsevier eBooks*, 755–760. https://doi.org/10.1016/b978-0-323-88506-5.50118-2

Baevski, A., Zhou, Y., Mohamed, A., & Auli, M. (2020). wav2vec 2.0: A framework for self-supervised learning of speech representations. *Advances in Neural Information Processing Systems*, 33, p. 12449 3–12460

Bosker, H. R., & Ghitza, O. (2018). Entrained theta oscillations guide perception of subsequent speech: Behavioural evidence from rate normalisation. *Language, Cognition and Neuroscience*,1–13. http://dx.doi.org/10.1080/23273798.2018. 1439179.

Burns, E., Laskowski, N., & Tucci, L. (2023). Artificial intelligence (AI). *Enterprise AI.* https://www.techtarget.com/searchenterpriseai/definition/AI-Artificial-Intelligence.

Fu, Q.-J., Galvin, J. J., & Wang, X. (2001). Recognition of time-distorted sen- tences by normal-hearing and cochlear-implant listeners. *The Journal of the Acoustical Society of America, 109(1)*, 379–384. http://dx.doi.org/10.1121/1. 1327578.

Ghitza, O. (2012). On the role of theta-driven syllabic parsing in decoding speech: Intelligibility of Speech with a Manipulated Modulation Spectrum. Frontiers in Psychology, 3, Article 238. https://doi.org/10.3389/fpsyg.2012.00238.

Ghitza, O. (2014). Behavioral evidence for the role of cortical θ oscillations in determining auditory channel capacity for speech. Frontiers in Psy, Article 652. https://doi.org/10.3389/fpsyg.2014.00652.

Ghitza, O., & Greenberg, S. M. (2009). On the possible role of brain rhythms in speech perception: Intelligibility of time-compressed speech with periodic and aperiodic insertions of silence. *Phonetica*, *66*(1–2), 113–126. https://doi.org/10.1159/000208934.

Hannun et al., (2014). Deep Speech: Scaling up end-to-end speech recognition. https://doi.org/10.48550/arXiv.1412.5567

Heald, S. L. M., & Nusbaum, H. C. (2014). Speech perception as an active cognitive process. *Frontiers in Systems Neuroscience*, *8*, Article 35. https://doi.org/10.3389/fnsys.2014.00035

Kuhl, P.K. Human adults and human infants show a “perceptual magnet effect” for the prototypes of speech categories, monkeys do not. *Perception & Psychophysics* 50, 93–107 (1991). https://doi.org/10.3758/BF03212211

Long, M. H. (1990). The Least a Second Language Acquisition Theory Needs to Explain. *TESOL Quarterly*, *24*(4), 649 – 666. https://doi.org/10.2307/3587113

Miller, G. A., & Licklider, J. C. R. (1950). The Intelligibility of Interrupted Speech. *Journal of the Acoustical Society of America*, *22*(2), 167–173. https://doi.org/10.1121/1.1906584

O'Shea, K., & Nash, R. R. (2015). An Introduction to Convolutional Neural Networks. https://doi.org/10.48550/arxiv.1511.08458

Penn, L. R., Ayasse, N. D., Wingfield, A., & Ghitza, O. (2018). The possible role of brain rhythms in perceiving fast speech: Evidence from adult ageing. *The Journal of the Acoustical Society of America*, *144(4)*, 2088–2094. https://doi.org/10.1121/1.5054905.

Pagel, M. (2017). Q&A: What is human language, when did it evolve and why should we care? *BMC Biology*, *15(1*). https://doi.org/10.1186/s12915-017-0405-3.

Schneider, S., Baevski, A., Collobert, R., & Auli, M. (2019). *wav2vec: Unsupervised Pre-Training for Speech Recognition*. https://doi.org/10.21437/interspeech.2019-1873.

Shannon, R. V., Zeng, F. G., Kamath, V., Wygonski, J., & Ekelid, M. (1995). Speech recognition with primarily temporal cues. *Science*, *270*(5234), 303–304. DOI: 10.1126/science.270.5234.303

Spille, C., & Meyer, B. (2017). Listening in the Dips: Comparing Relevant Features for Speech Recognition in Humans and Machines. *INTERSPEECH 2017.* 20–24 https://doi.org/10.21437/interspeech.2017-1168.

Vaswani, A., et al., (2017). Attention is all you need. *Advances in Neural Information Processing Systems*, 30, 5998–6008. https://doi.org/10.48550/arXiv.1706.03762

Panayotov, V., Chen, G., Povey, D., & Khudanpur, S. (2015). Librispeech: An ASR corpus based on public domain audio books. *IEEE international conference on acoustics, speech and signal processing (ICASSP).* 5206–5210. http://dx.doi.org/10.1109/ICASSP.2015.7178964.

7. Appendix

The code is attached below in pdf form.

A comparison of Wav2Vec 2.0 and humans in handling frequency shifted speech: A Qualitative Analysis

This Colab notebook presents an analysis of speech manipulations using the Wav2Vec2.0 model and evaluates its performance using evaluation metrics. The research is conducted as part of a bachelor thesis, aiming to investigate the robustness of the model in the presence of various speech manipulations. The notebook demonstrates the tokenization of audio signals and the application of manipulations such as masking and shifting. It feeds the manipulated signals to the model for transcription prediction and calculates evaluation metrics like Word Error Rate (WER) by comparing the predicted transcriptions with a reference. The obtained transcriptions, WER values, and relevant metrics provide insights into the model's accuracy and performance in different speech manipulation scenarios. This work contributes to the understanding of the Wav2Vec2.0 model's applicability in real-world speech processing tasks and serves as a valuable resource for researchers and practitioners in the field of automatic speech recognition.

Nilansha Dargan 13130366

Installing and Importing

```
#Installing datasets & wget
!pip install -q datasets wget
```

```
         ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 486.2/486.2 kB 8.4 MB/s eta 0:00:00
      Preparing metadata (setup.py) ... done
         ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 110.5/110.5 kB 9.0 MB/s eta 0:00:00
         ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 212.5/212.5 kB 18.6 MB/s eta 0:00:00
         ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 134.3/134.3 kB 12.7 MB/s eta 0:00:00
         ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.0/1.0 MB 34.3 MB/s eta 0:00:00
         ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 236.8/236.8 kB 17.8 MB/s eta 0:00:00
         ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 114.5/114.5 kB 6.4 MB/s eta 0:00:00
         ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 268.8/268.8 kB 18.0 MB/s eta 0:00:00
         ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 149.6/149.6 kB 10.0 MB/s eta 0:00:00
      Building wheel for wget (setup.py) ... done
```

```
#Installing Transformers
!pip install -q transformers
```

```
         ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 7.2/7.2 MB 44.6 MB/s eta 0:00:00
         ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 7.8/7.8 MB 32.7 MB/s eta 0:00:00
         ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.3/1.3 MB 61.7 MB/s eta 0:00:00
```

```
#Importing important libraries and Adolfi(2023) available code
import wget
import os
if not os.path.exists('pycochleagram'):
  !git clone https://github.com/mcdermottLab/pycochleagram
  os.chdir('pycochleagram')
  !python setup.py install
if not os.path.exists('manipulations.py'):
  wget.download('https://gitfront.io/r/fedeadolfi/b8d002dbffa6392bbe8d1793e2ea5d66a8e209ac/asr-vs-humans/raw/manipulations.py'
if not os.path.exists('analyses.py'):
  wget.download('https://gitfront.io/r/fedeadolfi/b8d002dbffa6392bbe8d1793e2ea5d66a8e209ac/asr-vs-humans/raw/analyses.py')
```

```
    copying build/lib/pycochleagram/erbfilter.py -> build/bdist.linux-x86_64/egg/pycochleagram
    copying build/lib/pycochleagram/demo.py -> build/bdist.linux-x86_64/egg/pycochleagram
    copying build/lib/pycochleagram/utils.py -> build/bdist.linux-x86_64/egg/pycochleagram
    copying build/lib/pycochleagram/__init__.py -> build/bdist.linux-x86_64/egg/pycochleagram
    byte-compiling build/bdist.linux-x86_64/egg/pycochleagram/subband.py to subband.cpython-310.pyc
    byte-compiling build/bdist.linux-x86_64/egg/pycochleagram/cochleagram.py to cochleagram.cpython-310.pyc
    byte-compiling build/bdist.linux-x86_64/egg/pycochleagram/erbfilter.py to erbfilter.cpython-310.pyc
    byte-compiling build/bdist.linux-x86_64/egg/pycochleagram/demo.py to demo.cpython-310.pyc
    byte-compiling build/bdist.linux-x86_64/egg/pycochleagram/utils.py to utils.cpython-310.pyc
    byte-compiling build/bdist.linux-x86_64/egg/pycochleagram/__init__.py to __init__.cpython-310.pyc
    creating build/bdist.linux-x86_64/egg/EGG-INFO
    copying pycochleagram.egg-info/PKG-INFO -> build/bdist.linux-x86_64/egg/EGG-INFO
    copying pycochleagram.egg-info/SOURCES.txt -> build/bdist.linux-x86_64/egg/EGG-INFO
    copying pycochleagram.egg-info/dependency_links.txt -> build/bdist.linux-x86_64/egg/EGG-INFO
    copying pycochleagram.egg-info/not-zip-safe -> build/bdist.linux-x86_64/egg/EGG-INFO
    copying pycochleagram.egg-info/requires.txt -> build/bdist.linux-x86_64/egg/EGG-INFO
    copying pycochleagram.egg-info/top_level.txt -> build/bdist.linux-x86_64/egg/EGG-INFO
    creating dist
    creating 'dist/pycochleagram-0.1-py3.10.egg' and adding 'build/bdist.linux-x86_64/egg' to it
    removing 'build/bdist.linux-x86_64/egg' (and everything under it)
    Processing pycochleagram-0.1-py3.10.egg
    creating /usr/local/lib/python3.10/dist-packages/pycochleagram-0.1-py3.10.egg
    Extracting pycochleagram-0.1-py3.10.egg to /usr/local/lib/python3.10/dist-packages
    Adding pycochleagram 0.1 to easy-install.pth file

    Installed /usr/local/lib/python3.10/dist-packages/pycochleagram-0.1-py3.10.egg
    Processing dependencies for pycochleagram==0.1
    Finished processing dependencies for pycochleagram==0.1
```

```
from datasets import load_dataset, get_dataset_config_names, get_dataset_split_names
import IPython.display as ipd
from IPython.display import Audio
from manipulations import *
from analyses import *


import soundfile as sf
import librosa
import torch
from transformers import Wav2Vec2ForCTC, Wav2Vec2Tokenizer
```

Loading LibriSpeech test data

```
ls_test = load_dataset("librispeech_asr", "clean", split="test", streaming=True)
```

```
    Downloading builder script:    11.5k/? [00:00<00:00, 280kB/s]

    Downloading metadata:    10.1k/? [00:00<00:00, 174kB/s]

    Downloading readme:    10.2k/? [00:00<00:00, 191kB/s]
```

```
N_samples = 100
ls_test_subset = list(ls_test.take(N_samples))
```

```
def get_signal(ls_item):
  return ls_item['audio']['array']

def get_sr(ls_item):
  return ls_item['audio']['sampling_rate']
```

```
# choose audio sample
sample = ls_test_subset[0]
print(type(sample))
signal = get_signal(sample)
sr = get_sr(sample)
sr_ms = sr / 1000
print(type(signal))
print(sr)
```

```
    <class 'dict'>
    <class 'numpy.ndarray'>
    16000
```

```
Audio(data=signal, rate=sr)
```

0:00 / 0:03

Using the model: Wav2Vec2.0

```
# Create an instance of the Wav2Vec2Tokenizer class and load the tokenizer from the "facebook/wav2vec2-base-960h" pretrained m
tokenizer = Wav2Vec2Tokenizer.from_pretrained("facebook/wav2vec2-base-960h")

# Create an instance of the Wav2Vec2ForCTC class and load the model from the "facebook/wav2vec2-base-960h" pretrained model
model = Wav2Vec2ForCTC.from_pretrained("facebook/wav2vec2-base-960h")
```

```
    Downloading (…)olve/main/vocab.json: 100%                                    291/291 [00:00<00:00, 9.35kB/s]

    Downloading (…)okenizer_config.json: 100%                                    163/163 [00:00<00:00, 7.43kB/s]

    Downloading (…)cial_tokens_map.json: 100%                                    85.0/85.0 [00:00<00:00, 2.31kB/s]

    Downloading (…)lve/main/config.json:    1.60k/? [00:00<00:00, 49.5kB/s]
    The tokenizer class you load from this checkpoint is not the same type as the class this function is called from. It may
    The tokenizer class you load from this checkpoint is 'Wav2Vec2CTCTokenizer'.
    The class this function is called from is 'Wav2Vec2Tokenizer'.
    /usr/local/lib/python3.10/dist-packages/transformers/models/wav2vec2/tokenization_wav2vec2.py:792: FutureWarning: The cla
      warnings.warn(
    Downloading model.safetensors: 100%                                  378M/378M [00:04<00:00, 54.3MB/s]
    Some weights of Wav2Vec2ForCTC were not initialized from the model checkpoint at facebook/wav2vec2-base-960h and are newl
    You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
```

```
# Tokenize the audio signal using the tokenizer and convert it to PyTorch tensors
input_values = tokenizer(signal, return_tensors="pt").input_values

# Pass the input values through the Wav2Vec2 model to get the logits
logits = model(input_values).logits

# Find the predicted token ids by taking the argmax along the last dimension of the logits
predicted_ids = torch.argmax(logits, dim=-1)

# Decode the predicted token ids into text using the tokenizer and extract the first (and only) sequence
text = tokenizer.batch_decode(predicted_ids)[0]
```

```
#Original audio as input
text
```

```
    'CONCORD RETURNED TO ITS PLACE AMIDST THE TENTS'
```

Repackaging

Redefined timewarp function

```
#@title Redefined timewarp function
def my_timewarp(signal, stretch_factor):
  hop_len = 512
  n_fft = 1024
  power = None  # if None, the complex spectrogram is returned
  _spectrogram = torchaudio.transforms.Spectrogram(
      n_fft=n_fft,
      win_length=None,
      hop_length=hop_len,
      power=power,
      center=True,
      pad_mode="reflect",
      )
  _timestretch = torchaudio.transforms.TimeStretch(
      hop_length=hop_len, n_freq=513, fixed_rate=stretch_factor  # > 1.0 is compression
      )

  _magnitude = lambda arr: torch.abs(arr)
  ###
  _griffinlim = torchaudio.transforms.GriffinLim(
      n_iter=32,
      n_fft=n_fft,
      win_length=None,
      hop_length=hop_len,
      power=1.0,
      )
  return _griffinlim(_magnitude(_timestretch(_spectrogram((torch.tensor(signal)))))).numpy()


def _compute_segmentation(signal, win_len):
    # Get the remainder of the signal that will be missed by windowing
    num_remainder = signal.shape[-1] % win_len
    signal_remainder = np.array(signal[signal.shape[-1] - num_remainder:])[np.newaxis, :]
    # Get sliding windows of the signal, keep only adjacent non-overlapping windows
    chunks = np.array(
        np.lib.stride_tricks.sliding_window_view(signal, win_len, axis=0)[0::win_len, :]
        )  # shape=(num_chunks, window_shape)
    return chunks, signal_remainder

def _compute_insert(chunks, len_silence):
    # Assumes shape of `chunks` is (num_chunks, len_chunks)
    # returns chunks with silence. Shape = (num_chunks, len_chunks + len_silence)
    return np.concatenate([
        np.append(arr, np.zeros(len_silence))[np.newaxis, :]
        for arr in chunks
        ], axis=0)

def _assemble_sequence(chunks, remainder):
    return np.concatenate([arr for arr in chunks] + [remainder[0, :]], axis=0)

def compress_insert(signal, len_silence, stretch_factor=3.0, win_len=640):
    signal_compressed = my_timewarp(signal, stretch_factor)
    chunks, remainder = _compute_segmentation(signal_compressed, win_len)
    chunks_transf = _compute_insert(chunks, len_silence)
    signal_transf = _assemble_sequence(chunks_transf, remainder)
    return signal_transf

def compress_insert_mix(signal, stretch_factor=3.0, win_len=640, snr=1.0, len_silence=640, src=None):
    return  mix(
        compress_insert(signal=signal, len_silence=len_silence,
                        stretch_factor=stretch_factor, win_len=win_len),
        snr=snr,
        src=src
        )


#Changing the number of samples in which silence is added
len_silence_list = [1280, 1067, 914, 800, 711, 640, 582, 533, 492, 457, 427, 400, 376, 356, 337, 320 ]
#Creating list of manipulated audios
cimed_list = []
#Manipulating audios with different silence length
for length in len_silence_list:
  cimed = compress_insert_mix(signal,
                              stretch_factor= 2.0,
                              win_len=640,
                              snr=1.0,
                              len_silence= length,
                              src=None)
  print(length)
  ipd.display(Audio(data=cimed, rate=sr))
  cimed_list.append(cimed)
```

```
    1280

        0:00 / 0:05

    1067

        0:00 / 0:04

    914

        0:00 / 0:04

    800

        0:00 / 0:03

    711

        0:00 / 0:03

    640

        0:00 / 0:03

    582

        0:00 / 0:03

    533
```

```
#Feeding manipulations to Wav2Vec2.0
text2_list = []
for s in cimed_list:
  input_values = tokenizer(s,return_tensors="pt").input_values
  logits = model(input_values).logits
  predicted_ids = torch.argmax(logits,dim=-1)
  text_2 = tokenizer.batch_decode(predicted_ids)[0]
  text2_list.append(text_2)
  print(text_2)
```

```
    OEROR T TE ETE T T TEIT TOTEIIT  TTEMCATT
    E ER  IITEST ITST TEGIST  THETTEMEGAEST
    ORDWARD TI T TI TE MINUTE THE TE
    OETU ST AMIDSTT THE TETT
    AR TURNS AMIDST THE DACKERSS
    R TI  AIS THE TT
    S MIDST THE TAT
    REAT
    RTURAMIT THE DEAT

    E


    RETURAM
    RETURNEMITHE AT
```

Analysis

```
        0:00 / 0:02
```

```
!pip install jiwer
```

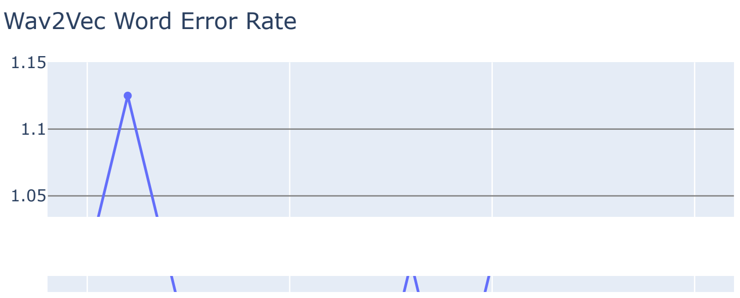
```
    Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
    Collecting jiwer
      Downloading jiwer-3.0.2-py3-none-any.whl (21 kB)
    Requirement already satisfied: click<9.0.0,>=8.1.3 in /usr/local/lib/python3.10/dist-packages (from jiwer) (8.1.3)
    Collecting rapidfuzz==2.13.7 (from jiwer)
      Downloading rapidfuzz-2.13.7-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (2.2 MB)
         ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.2/2.2 MB 24.4 MB/s eta 0:00:00
    Installing collected packages: rapidfuzz, jiwer
    Successfully installed jiwer-3.0.2 rapidfuzz-2.13.7
```

```
#Calculating Word Eroor Rate for each manipulation
from jiwer import wer

reference = text
error_list = []
for hypothesis in text2_list:
  error = wer(reference, hypothesis)
  print(error)
  error_list.append(error)
  print(error_list)
```

```
1.0
[1.0]
1.125
[1.0, 1.125]
1.0
[1.0, 1.125, 1.0]
0.875
[1.0, 1.125, 1.0, 0.875]
0.875
[1.0, 1.125, 1.0, 0.875, 0.875]
0.75
[1.0, 1.125, 1.0, 0.875, 0.875, 0.75]
0.875
[1.0, 1.125, 1.0, 0.875, 0.875, 0.75, 0.875]
0.875
[1.0, 1.125, 1.0, 0.875, 0.875, 0.75, 0.875, 0.875]
1.0
[1.0, 1.125, 1.0, 0.875, 0.875, 0.75, 0.875, 0.875, 1.0]
0.875
[1.0, 1.125, 1.0, 0.875, 0.875, 0.75, 0.875, 0.875, 1.0, 0.875]
1.0
[1.0, 1.125, 1.0, 0.875, 0.875, 0.75, 0.875, 0.875, 1.0, 0.875, 1.0]
1.0
[1.0, 1.125, 1.0, 0.875, 0.875, 0.75, 0.875, 0.875, 1.0, 0.875, 1.0, 1.0]
1.0
[1.0, 1.125, 1.0, 0.875, 0.875, 0.75, 0.875, 0.875, 1.0, 0.875, 1.0, 1.0, 1.0]
1.0
[1.0, 1.125, 1.0, 0.875, 0.875, 0.75, 0.875, 0.875, 1.0, 0.875, 1.0, 1.0, 1.0, 1.0]
1.0
[1.0, 1.125, 1.0, 0.875, 0.875, 0.75, 0.875, 0.875, 1.0, 0.875, 1.0, 1.0, 1.0, 1.0, 1.0]
1.0
[1.0, 1.125, 1.0, 0.875, 0.875, 0.75, 0.875, 0.875, 1.0, 0.875, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
```

Visualisation

```
import plotly.express as px

def create_graph(x_values, y_values):
    # Create the plot using Plotly Express
    fig = px.line(x=x_values, y=y_values, markers=True)

    # Customize the plot
    fig.update_layout(
        xaxis_title="Audio:Silence Ratio",
        yaxis_title="WER (Word Error Rate)",
        title="Wav2Vec Word Error Rate",
        legend_title="",
        showlegend=True,
        xaxis=dict(tickfont=dict(size=10)),
        yaxis=dict(showgrid=True, gridcolor='gray', gridwidth=0.5),
        margin=dict(l=50, r=50, t=50, b=50),
    )

    # Show the plot
    fig.show()

# Example data
x_data = [0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2]
y_data = error_list

# Call the function to create the graph
create_graph(x_data, y_data)
```



```
# Define a list of window lengths(number of samples) for masking
winlen_list = [1, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000]

# Create an empty list to store the masked signals
masked_list = []

# Iterate over each window length in the winlen_list
for length_2 in winlen_list:

    # Mask the original signal using the specified parameters
    masked = mask(signal,
                  win_len=length_2,
                  mask_fraction=0.5,
                  mask="noise",  # "silence" or "noise"
                  snr=0.75,
                  fade_len=0)

    # Display the masked audio signal using IPython's display function
    ipd.display(Audio(data=masked, rate=sr))

    # Append the masked signal to the masked_list
    masked_list.append(masked)
```

```
0:00 / 0:03

0:00 / 0:03

0:00 / 0:03

0:00 / 0:03

0:00 / 0:03

0:00 / 0:03

0:00 / 0:03

0:00 / 0:03

0:00 / 0:03

0:00 / 0:03

0:00 / 0:03
```

```
# Create an empty list to store the resulting text after feeding masked signals to Wav2Vec2.0
text3_list = []

# Iterate over each masked signal in the masked_list
for m in masked_list:

    # Tokenize the masked signal using the tokenizer and convert it to PyTorch tensors
    input_values = tokenizer(m, return_tensors="pt").input_values

    # Pass the input values through the Wav2Vec2.0 model to get the logits
    logits = model(input_values).logits
    # Find the predicted token ids by taking the argmax along the last dimension of the logits
    predicted_ids = torch.argmax(logits, dim=-1)

    # Decode the predicted token ids into text using the tokenizer and extract the first (and only) sequence
    text_3 = tokenizer.batch_decode(predicted_ids)[0]

    # Append the resulting text to the text3_list
    text3_list.append(text_3)

    # Print the resulting text
    print(text_3)
```

```
 CONCORD RETURNED TO ITS PLACE AMIDST THE TENTS
 CONCORD RETURNED TO ITS PLACE AMIDST THE TENTS
 CONCORD RETURNED TO ITS PLACE AMIDST THE TENTS
 CONCORD RETURNED TO ITS PLACE AMIDST THE TENTS
 CONCORD RETURNED TO ITS PLACE AMIDST THE TENTS
 CONCORD RETURNED TO ITS PLACE AMIDST THE TENTS
 CONCORD RETURNED TO ITS PLACE AMIDST THE TENTS
 CONCORD RETURNED TO ITS PLACE AMIDST THE TENTS
 CONCORD RETURNED TO ITS PLACE AMIDST THE TENTS
 CONCORD RETURNED TO ITS PLACE AMIDST THE TENTS
 CONCORD RETURNED TO ITS PLACE AMIDST THE TENTS
```

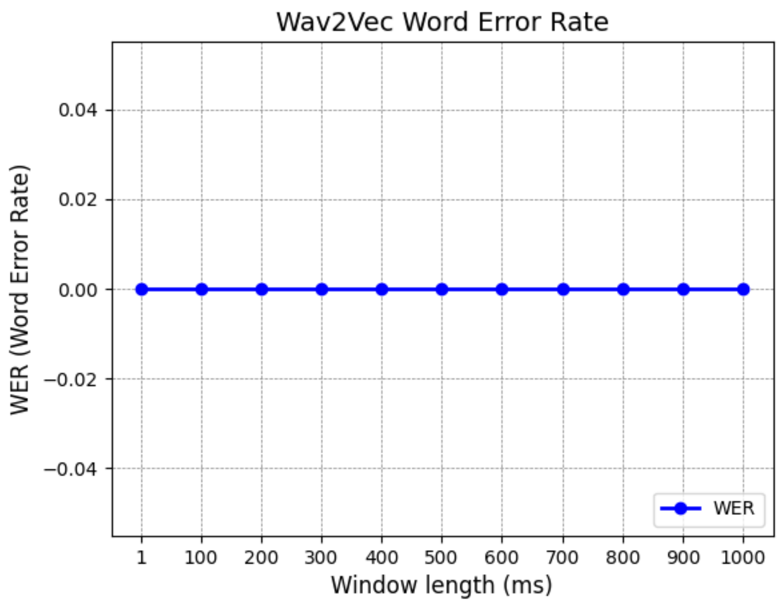
analysing

```
# Import the word error rate (WER) calculation function from the jiwer library
from jiwer import wer

# Set the reference text for comparison
reference = text

# Create an empty list to store the WER for each manipulation
error_list_2 = []

# Iterate over each hypothesis text in text3_list
for hypothesis_2 in text3_list:

    # Calculate the word error rate (WER) between the reference and the hypothesis text
    error_2 = wer(reference, hypothesis_2)

    # Print the calculated WER
    print(error_2)

    # Append the WER to the error_list_2
    error_list_2.append(error_2)

    # Print the current contents of the error_list_2
    print(error_list_2)
```

```
 0.0
 [0.0]
 0.0
 [0.0, 0.0]
 0.0
 [0.0, 0.0, 0.0]
 0.0
 [0.0, 0.0, 0.0, 0.0]
 0.0
 [0.0, 0.0, 0.0, 0.0, 0.0]
 0.0
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
 0.0
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
 0.0
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
 0.0
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
 0.0
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
 0.0
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

Visualising (make changes)

```
import matplotlib.pyplot as plt

def create_graph(x_values, y_values):
    # Customize the plot
    plt.plot(x_values, y_values, marker='o', linestyle='-', color='b', linewidth=2)
    plt.xlabel("Window length (ms)", fontsize=12)
    plt.ylabel('WER (Word Error Rate)', fontsize=12)
    plt.title("Wav2Vec Word Error Rate", fontsize=14)
    plt.grid(True)

    # Customize the x-axis tick values and labels
    plt.xticks(x_values, fontsize=10)

    # Add a background grid
    plt.grid(color='gray', linestyle='--', linewidth=0.5)

    # Add a legend
    plt.legend(['WER'], loc='lower right')




    # Adjust the plot margins
    plt.margins(0.05)

    # Show the plot
    plt.show()

# Example data
x_data = [1, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000]
y_data = error_list_2

# Call the function to create the graph
create_graph(x_data, y_data)
```



Silencing

```
# Define a list of window lengths(number of samples) for silencing
win_len_list = [1, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000]

# Create an empty list to store the silenced signals
silenced_list = []

# Iterate over each window length in the win_len_list
for length_3 in win_len_list:

    # Silence the original signal using the specified parameters
    silenced = mask(signal,
                    win_len=length_3,
                    mask_fraction=0.5,
                    mask="silence",  # "silence" or "noise"
                    snr=0.75,
                    fade_len=0)

    # Display the silenced audio signal using IPython's display function
    ipd.display(Audio(data=silenced, rate=sr))

    # Append the silenced signal to the silenced_list
    silenced_list.append(silenced)
```

```
/usr/local/lib/python3.10/dist-packages/IPython/lib/display.py:174: RuntimeWarning:

invalid value encountered in true_divide

     0:00 / 0:03

     0:00 / 0:03

     0:00 / 0:03

     0:00 / 0:03

     0:00 / 0:03

     0:00 / 0:03

     0:00 / 0:03

     0:00 / 0:03

     0:00 / 0:03

     0:00 / 0:03

     0:00 / 0:03
```

```
# Create an empty list to store the resulting text after feeding silenced signals to Wav2Vec2.0
text4_list = []

# Iterate over each silenced signal in the silenced_list
for silence in silenced_list:

    # Tokenize the silenced signal using the tokenizer and convert it to PyTorch tensors
    input_values = tokenizer(silence, return_tensors="pt").input_values

    # Pass the input values through the Wav2Vec2.0 model to get the logits
    logits = model(input_values).logits

    # Find the predicted token ids by taking the argmax along the last dimension of the logits
    predicted_ids = torch.argmax(logits, dim=-1)

    # Decode the predicted token ids into text using the tokenizer and extract the first (and only) sequence
    text_4 = tokenizer.batch_decode(predicted_ids)[0]

    # Append the resulting text to the text4_list
    text4_list.append(text_4)

    # Print the resulting text
    print(text_4)
```

```
     AYMATHEPEND
     COR RETURN LAAMIDS THE TENT
```

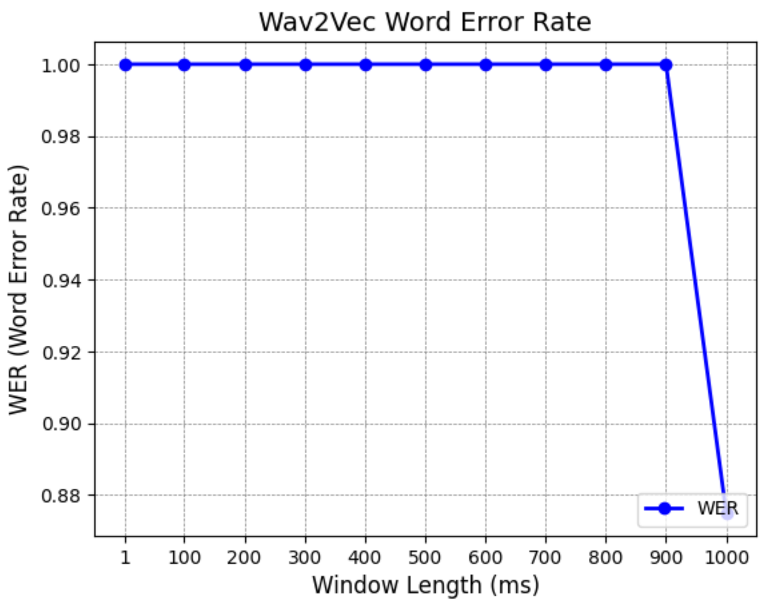
Analysis

```
# Set the reference text for comparison
reference = text

# Create an empty list to store the WER for each manipulation
error_list_3 = []

# Iterate over each hypothesis text in text4_list
for hypothesis_3 in text4_list:

    # Calculate the word error rate (WER) between the reference and the hypothesis text
    error_3 = wer(reference, hypothesis_3)

    # Print the calculated WER
    print(error_3)

    # Append the WER to the error_list_3
    error_list_3.append(error_3)

    # Print the current contents of the error_list_3
    print(error_list_3)
```

```
1.0
[1.0]
1.0
[1.0, 1.0]
1.0
[1.0, 1.0, 1.0]
1.0
[1.0, 1.0, 1.0, 1.0]
1.0
[1.0, 1.0, 1.0, 1.0, 1.0]
1.0
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
1.0
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
1.0
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
1.0
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
1.0
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
0.875
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.875]
```

Visualising

```
import matplotlib.pyplot as plt

def create_graph(x_values, y_values):
    # Customize the plot
    plt.plot(x_values, y_values, marker='o', linestyle='-', color='b', linewidth=2)
    plt.xlabel("Window Length (ms)", fontsize=12)
    plt.ylabel('WER (Word Error Rate)', fontsize=12)
    plt.title("Wav2Vec Word Error Rate", fontsize=14)
    plt.grid(True)

    # Customize the x-axis tick values and labels
    plt.xticks(x_values, fontsize=10)

    # Add a background grid
    plt.grid(color='gray', linestyle='--', linewidth=0.5)
    # Add a legend
    plt.legend(['WER'], loc='lower right')



    # Adjust the plot margins
    plt.margins(0.05)

    # Show the plot
    plt.show()

# Example data
x_data = [1, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000]
y_data = error_list_3

# Call the function to create the graph
create_graph(x_data, y_data)
```



My Manipulation (frequency shift)

```
#Shifting to higher frequency
import numpy as np
from scipy.io import wavfile
from scipy.fft import rfft, irfft


# Normalize the audio data
signal = signal / np.max(np.abs(signal))

# Compute the FFT
transformed_audio = rfft(signal)

# Perform the frequency shift
shift_frequency_list =[100, 200, 300, 400, 500, 600, 700, 800, 900, 1000]  # Frequency shift amount (Hz)


# Shift the frequencies
shift_list = []
for shift_frequency in shift_frequency_list:
  shift_indices = np.round(shift_frequency * len(transformed_audio) / sr).astype(int)
  transformed_audio_shifted = np.roll(transformed_audio, shift_indices)
  shift_list.append(transformed_audio_shifted)

# Apply the inverse FFT
shift_data_list = []
for transformed_audio in shift_list:
  shifted_audio_data = irfft(transformed_audio)
  shift_data_list. append(shifted_audio_data)
  ipd.display(Audio(data=shifted_audio_data, rate=sr))
```

0:00 / 0:03

0:00 / 0:03

0:00 / 0:03

0:00 / 0:03

0:00 / 0:03

0:00 / 0:03

0:00 / 0:03

0:00 / 0:03

0:00 / 0:03

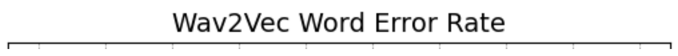
```
#Feeding manipulations to Wav2Vec2.0
# Create an empty list to store the resulting text after feeding shifted data to Wav2Vec2.0
text5_list = []

# Iterate over each data in shift_data_list
for data in shift_data_list:

    # Tokenize the shifted data using the tokenizer and convert it to PyTorch tensors
    input_values = tokenizer(data, return_tensors="pt").input_values

    # Pass the input values through the Wav2Vec2.0 model to get the logits
    logits = model(input_values).logits

    # Find the predicted token ids by taking the argmax along the last dimension of the logits
    predicted_ids = torch.argmax(logits, dim=-1)

    # Decode the predicted token ids into text using the tokenizer and extract the first (and only) sequence
    text_5 = tokenizer.batch_decode(predicted_ids)[0]

    # Append the resulting text to the text5_list
    text5_list.append(text_5)

    # Print the resulting text
    print(text_5)
```

```
     CONCORD RETURNED TO ITS PLACE AMIDST THE TENTS
     CONCORD RETURNED TO ITS PLACE AMIDST THE TENTS
     CONCORD RETURNED TO ITS PLACE AMIDST THE TENTS
     CONCORD RETURNED TO ITS PLACE AMIDST THE TENTS
     CONCORD RETURNED TO ITS PLACE AMIDST THE TENTS
     CONCORD RETURNED TO ITS PLACE AMIDST THE TENTS
     CONCARD RETURNED TO ITS PLACE AMIDST THE TENTS
     CONCARD RETURNED TO ITS PLACE AMIDST THE TENTS
     CONCARN RETURNED TO ITS PLACE AMIDST THE TENTS
     KANCAR RETURNED TO ITS PLACE AMIDST THE TENTS
```

```
# Set the reference text for comparison
reference = text

# Create an empty list to store the WER for each manipulation
error_list_4 = []

# Iterate over each hypothesis text in text5_list
for hypothesis_4 in text5_list:

    # Calculate the word error rate (WER) between the reference and the hypothesis text
    error_4 = wer(reference, hypothesis_4)

    # Print the calculated WER
    print(error_4)

    # Append the WER to the error_list_4
    error_list_4.append(error_4)
    # Print the current contents of the error_list_4
    print(error_list_4)
```

```
0.0
[0.0]
0.0
[0.0, 0.0]
0.0
[0.0, 0.0, 0.0]
0.0
[0.0, 0.0, 0.0, 0.0]
0.0
[0.0, 0.0, 0.0, 0.0, 0.0]
0.0
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
0.125
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.125]
0.125
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.125, 0.125]
0.125
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.125, 0.125, 0.125]
0.125
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.125, 0.125, 0.125, 0.125]
```

```
import matplotlib.pyplot as plt

def create_graph(x_values, y_values):
    # Customize the plot
    plt.plot(x_values, y_values, marker='o', linestyle='-', color='b', linewidth=2)
    plt.xlabel("Frequency Shift (Hz)", fontsize=12)
    plt.ylabel('WER (Word Error Rate)', fontsize=12)
    plt.title("Wav2Vec Word Error Rate", fontsize=14)
    plt.grid(True)

    # Customize the x-axis tick values and labels
    plt.xticks(x_values, fontsize=10)

    # Add a background grid
    plt.grid(color='gray', linestyle='--', linewidth=0.5)

    # Add a legend
    plt.legend(['WER'], loc='lower right')




    # Adjust the plot margins
    plt.margins(0.05)

    # Show the plot
    plt.show()

# Example data
x_data = [100, 200, 300, 400, 500, 600, 700, 800, 900, 1000]
y_data = error_list_4

# Call the function to create the graph
create_graph(x_data, y_data)
```



```
import numpy as np
from scipy.io import wavfile
from scipy.fft import rfft, irfft

# Normalize the audio data
signal = signal / np.max(np.abs(signal))

# Perform the frequency shift
shift_frequency = [-100, -200, -300, -400, -500, -600, -700, -800, -900, -1000]  # Frequency shift amount (Hz)

# Compute the FFT
transformed_audio = rfft(signal)

# Shift the frequencies
shifted_list = []
for shifted_frequency in shift_frequency:
  shift_indices2 = np.round(shifted_frequency * len(transformed_audio) / sr).astype(int)
  transformed_audio_shifted2 = np.roll(transformed_audio, shift_indices2)
  shifted_list.append(transformed_audio_shifted2)

# Apply the inverse FFT
shifted_data_list =[]
for transformed_audio2 in shifted_list:
  shifted_audio_data2 = irfft(transformed_audio2)
  shifted_data_list.append(shifted_audio_data2)
  ipd.display(Audio(data=shifted_audio_data2, rate=sr))
```

```
0:00 / 0:03

0:00 / 0:03

0:00 / 0:03

0:00 / 0:03

0:00 / 0:03

0:00 / 0:03

0:00 / 0:03

0:00 / 0:03

0:00 / 0:03

0:00 / 0:03
```

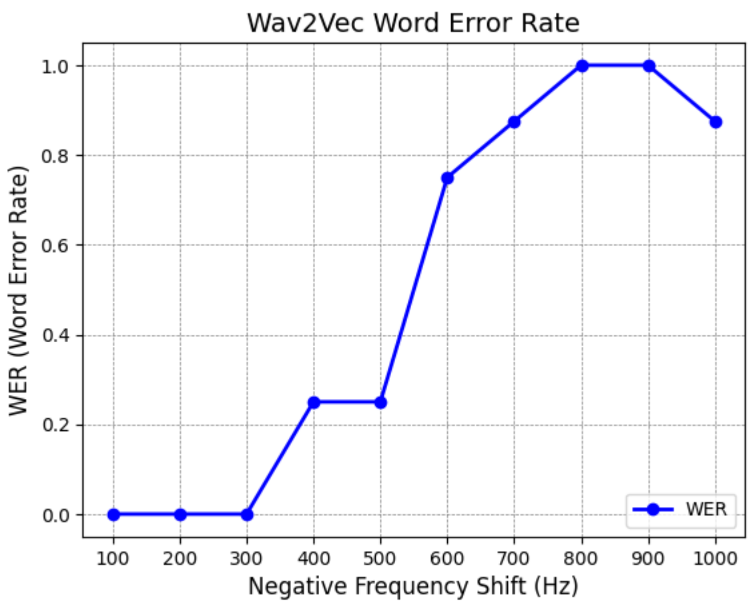
```
# Create an empty list to store the resulting text after feeding shifted data to Wav2Vec2.0
text6_list = []

# Iterate over each data2 in shifted_data_list
for data2 in shifted_data_list:

    # Tokenize the shifted data2 using the tokenizer and convert it to PyTorch tensors
    input_values = tokenizer(data2, return_tensors="pt").input_values

    # Pass the input values through the Wav2Vec2.0 model to get the logits
    logits = model(input_values).logits

    # Find the predicted token ids by taking the argmax along the last dimension of the logits
    predicted_ids = torch.argmax(logits, dim=-1)

    # Decode the predicted token ids into text using the tokenizer and extract the first (and only) sequence
    text_6 = tokenizer.batch_decode(predicted_ids)[0]

    # Append the resulting text to the text6_list
    text6_list.append(text_6)
    # Print the resulting text
    print(text_6)
```

```
     CONCORD RETURNED TO ITS PLACE AMIDST THE TENTS
     CONCORD RETURNED TO ITS PLACE AMIDST THE TENTS
     CONCORD RETURNED TO ITS PLACE AMIDST THE TENTS
     CONICORD RETURNED TO ITS PLACE AMITS THE TENTS
     CONCON RETURNED TO ITS PLACE AMITS THE TENTS
     CORRINCLLY ON WHAT TURN TO ITS PLACE IN AMIDST THE TURNS
     CARECALY O MATURN T WHICH PASON AMIDST THE TUDES
     CARENTLY ON A TRUNK WHICH CANS NEITS THE TUBES
     CERECLY O A TRONTWITCH GAINST EM ITS THE TURNS
     CAR CLE OF A TROMPAT GAINST AMIDST THE TANTS
```

```
# Set the reference text for comparison
reference = text

# Create an empty list to store the WER for each manipulation
error_list_5 = []

# Iterate over each hypothesis text in text6_list
for hypothesis_5 in text6_list:

    # Calculate the word error rate (WER) between the reference and the hypothesis text
    error_5 = wer(reference, hypothesis_5)

    # Print the calculated WER
    print(error_5)

    # Append the WER to the error_list_5
    error_list_5.append(error_5)

    # Print the current contents of the error_list_5
    print(error_list_5)
```

```
     0.0
     [0.0]
     0.0
     [0.0, 0.0]
     0.0
     [0.0, 0.0, 0.0]
     0.25
     [0.0, 0.0, 0.0, 0.25]
     0.25
     [0.0, 0.0, 0.0, 0.25, 0.25]
     0.75
     [0.0, 0.0, 0.0, 0.25, 0.25, 0.75]
     0.875
     [0.0, 0.0, 0.0, 0.25, 0.25, 0.75, 0.875]
     1.0
     [0.0, 0.0, 0.0, 0.25, 0.25, 0.75, 0.875, 1.0]
     1.0
     [0.0, 0.0, 0.0, 0.25, 0.25, 0.75, 0.875, 1.0, 1.0]
     0.875
     [0.0, 0.0, 0.0, 0.25, 0.25, 0.75, 0.875, 1.0, 1.0, 0.875]
```

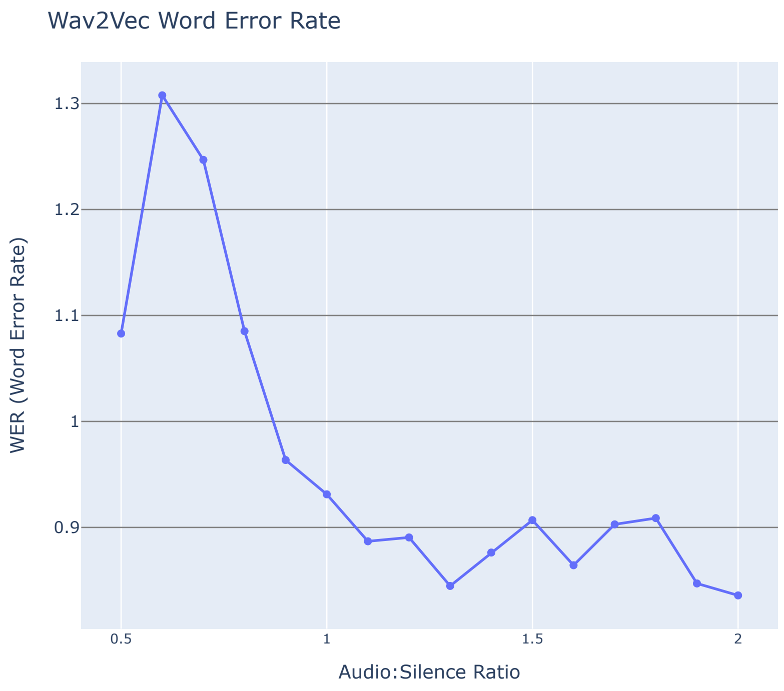
```
import matplotlib.pyplot as plt

def create_graph(x_values, y_values):
    # Customize the plot
    plt.plot(x_values, y_values, marker='o', linestyle='-', color='b', linewidth=2)
    plt.xlabel("Negative Frequency Shift (Hz)", fontsize=12)
    plt.ylabel('WER (Word Error Rate)', fontsize=12)
    plt.title("Wav2Vec Word Error Rate", fontsize=14)
    plt.grid(True)

    # Customize the x-axis tick values and labels
    plt.xticks(x_values, fontsize=10)

    # Add a background grid
    plt.grid(color='gray', linestyle='--', linewidth=0.5)

    # Add a legend
    plt.legend(['WER'], loc='lower right')




    # Adjust the plot margins
    plt.margins(0.05)
    # Show the plot
    plt.show()

# Example data
x_data = [100, 200, 300, 400, 500, 600, 700, 800, 900, 1000]
y_data = error_list_5

# Call the function to create the graph
create_graph(x_data, y_data)
```



Final Repackaging

```
#Summing list repacking of 10 audios

import numpy as np

# Declaring initial list of list
List_rp = np.array([[1.0, 1.125, 1.0, 0.875, 0.875, 0.75, 0.875, 0.875, 0.875, 0.875, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0],
[0.9767441860465116, 1.0465116279069768, 1.0, 1.0, 0.9767441860465116, 1.0, 0.8837209302325582, 0.8837209302325582, 0.97674418
[1.0, 1.0, 1.2727272727272727, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0],
[1.0, 1.0, 1.0, 0.9841269841269841, 0.9682539682539683, 0.9682539682539683, 0.9523809523809523, 0.9365079365079365, 0.92063492
[1.2903225806451613, 1.032258064516129, 0.967741935483871, 1.0, 0.9354838709677419, 0.967741935483871, 0.967741935483871, 0.93
[1.5454545454545454, 1.2727272727272727, 1.3636363636363635, 1.1818181818181819, 1.0303030303030303, 0.9696969696969697, 0.727
[1.0588235294117647, 2.411764705882353, 2.0588235294117645, 1.3529411764705883, 0.9411764705882353, 0.8235294117647058, 0.5882
[1.0, 1.0, 0.8888888888888888, 1.0, 1.0, 1.0, 1.0, 0.8888888888888888, 0.6666666666666666, 0.8888888888888888, 0.666666666666666
[1.0, 2.272727272727273, 2.0, 1.0, 0.9090909090909091, 1.0, 1.0, 1.0, 0.8181818181818182, 0.6363636363636364, 0.81818181818181
[0.9583333333333334, 0.9166666666666666, 0.9166666666666666, 1.4583333333333333, 1.0, 0.8333333333333334, 0.875, 0.83333333333

# Using numpy sum
res_rp = np.sum(List_rp, 0)

# printing result
print("final list - ", str(res_rp))

     final list -  [10.82967817 13.07765561 12.46848466 10.85221968  9.63605244  9.31255562
       8.86935184  8.90551963  8.44773561  8.76324605  9.06871431  8.64305329
       9.02906952  9.08798065  8.4714072   8.35866845]


# Define a list of floating-point numbers
myList_rp = [10.82967817, 13.07765561, 12.46848466, 10.85221968, 9.63605244, 9.31255562, 8.86935184, 8.90551963, 8.44773561, 8

# Define an integer value
myInt = 10

# Create a new list by dividing each element of myList_rp by myInt
newList_rp = [x / myInt for x in myList_rp]

# Print the new list
print(newList_rp)


     [1.0829678169999999, 1.307765561, 1.2468484659999999, 1.085221968, 0.963605244, 0.9312555619999999, 0.886935184, 0.89055]


#Creating the final visualisation
import plotly.express as px
```

```
def create_graph(x_values, y_values):
    # Create the plot using Plotly Express
    fig = px.line(x=x_values, y=y_values, markers=True)

    # Customize the plot
    fig.update_layout(
        xaxis_title="Audio:Silence Ratio",
        yaxis_title="WER (Word Error Rate)",
        title="Wav2Vec Word Error Rate",
        legend_title="",
        showlegend=True,
        xaxis=dict(tickfont=dict(size=10)),
        yaxis=dict(showgrid=True, gridcolor='gray', gridwidth=0.5),
        margin=dict(l=50, r=50, t=50, b=50),
    )

    # Show the plot
    fig.show()

# Example data
x_data = [0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2]
y_data = [1.0829678169999999, 1.307765561, 1.2468484659999999, 1.085221968, 0.963605244, 0.9312555619999999, 0.886935184, 0.89

# Call the function to create the graph
create_graph(x_data, y_data)
```



Final Masking

```
List_mask = np.array([[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
[0.09302325581395349, 0.0, 0.046511627906976744, 0.046511627906976744, 0.046511627906976744, 0.09302325581395349, 0.0930232558
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
[0.14285714285714285, 0.047619047619047616, 0.07936507936507936, 0.047619047619047616, 0.14285714285714285, 0.1428571428571428
[0.03225806451612903, 0.03225806451612903, 0.0, 0.0, 0.03225806451612903, 0.0, 0.03225806451612903, 0.03225806451612903, 0.032
[0.0, 0.0, 0.0, 0.0, 0.0, 0.030303030303030304, 0.06060606060606061, 0.0, 0.0, 0.0, 0.0],
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
[0.09090909090909091, 0.09090909090909091, 0.0, 0.09090909090909091, 0.09090909090909091, 0.09090909090909091, 0.0909090909090
[0.041666666666666664, 0.041666666666666664, 0.041666666666666664, 0.0, 0.041666666666666664, 0.041666666666666664, 0.04166666

# Using numpy sum
res_mask = np.sum(List_mask, 0)

# printing result
print("final list - ", str(res_mask))
```

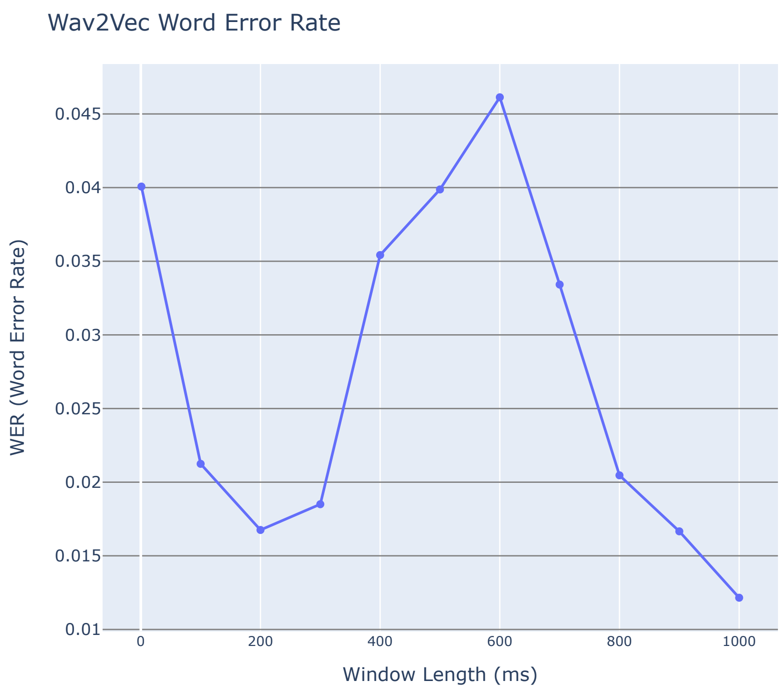
```
final list -  [0.40071422 0.21245287 0.16754337 0.18503977 0.35420259 0.39875919
 0.46132028 0.33416836 0.2046464  0.16662499 0.12154378]
```

```
# Define a list of floating-point numbers
myList_mask = [0.40071422, 0.21245287, 0.16754337, 0.18503977, 0.35420259, 0.39875919, 0.46132028, 0.33416836, 0.2046464, 0.16
```

```
# Define an integer value
myInt = 10

# Create a new list by dividing each element of myList_mask by myInt
newList_mask = [x / myInt for x in myList_mask]

# Print the new list
print(newList_mask)
```

```
    [0.040071422, 0.021245286999999998, 0.016754337, 0.018503976999999998, 0.035420258999999996, 0.039875918999999996, 0.0461
```

```
#Final Visualisation for masking
import plotly.express as px

def create_graph(x_values, y_values):
    # Create the plot using Plotly Express
    fig = px.line(x=x_values, y=y_values, markers=True)

    # Customize the plot
    fig.update_layout(
        xaxis_title="Window Length (ms)",
        yaxis_title="WER (Word Error Rate)",
        title="Wav2Vec Word Error Rate",
        legend_title="",
        showlegend=True,
        xaxis=dict(tickfont=dict(size=10)),
        yaxis=dict(showgrid=True, gridcolor='gray', gridwidth=0.5),
        margin=dict(l=50, r=50, t=50, b=50),
    )

    # Show the plot
    fig.show()

# Example data
x_data = [1, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000]
y_data = [0.040071422, 0.021245286999999998, 0.016754337, 0.018503976999999998, 0.035420258999999996, 0.039875918999999996, 0.

# Call the function to create the graph
create_graph(x_data, y_data)
```



Final Silencing

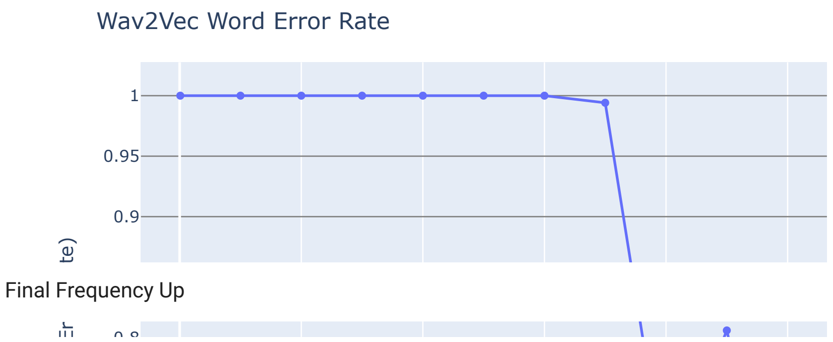
```
#Summing list repacking of 10 audios

import numpy as np

# Declaring initial list of list
List_sil = np.array([[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.875],
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.7674418604651163, 0.8837209302325582, 0.6511627906976745],
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.7272727272727273, 0.9090909090909091, 0.5454545454545454],
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.7936507936507936, 0.8571428571428571, 0.5555555555555556],
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.9032258064516129, 0.8387096774193549, 0.7096774193548387],
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.3939393939393939, 0.6060606060606061, 0.36363636363636365],
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.9411764705882353, 0.23529411764705882, 0.5294117647058824, 0.29411764705882354],
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.4444444444444444, 1.0, 0.5555555555555556],
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.8181818181818182, 0.7272727272727273, 0.7272727272727273],
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.7083333333333334, 0.7083333333333334, 0.5833333333333334]])

# Using numpy sum
res_sil = np.sum(List_sil, 0)

# printing result
print("final list - ", str(res_sil))
```

```
    final list -  [10.         10.         10.         10.         10.         10.
     10.          9.94117647  6.7917843   8.05974281  5.86076594]
```

```
# Define a list of floating-point numbers
myList_sil = [10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 9.94117647, 6.7917843, 8.05974281, 5.86076594]

# Define an integer value
myInt = 10

# Create a new list by dividing each element of myList_sil by myInt
newList_sil = [x / myInt for x in myList_sil]

# Print the new list
print(newList_sil)
```

```
    [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.994117647, 0.67917843, 0.8059742809999999, 0.586076594]
```

```
#Final Visualisation for silencing
import plotly.express as px

def create_graph(x_values, y_values):
    # Create the plot using Plotly Express
    fig = px.line(x=x_values, y=y_values, markers=True)

    # Customize the plot
    fig.update_layout(
        xaxis_title="Window Length (ms)",
        yaxis_title="WER (Word Error Rate)",
        title="Wav2Vec Word Error Rate",
        legend_title="",
        showlegend=True,
        xaxis=dict(tickfont=dict(size=10)),
        yaxis=dict(showgrid=True, gridcolor='gray', gridwidth=0.5),
        margin=dict(l=50, r=50, t=50, b=50),
    )

    # Show the plot
    fig.show()

# Example data
x_data = [1, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000]
y_data = [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.994117647, 0.67917843, 0.8059742809999999, 0.586076594]


# Call the function to create the graph
create_graph(x_data, y_data)
```



Final Frequency Up

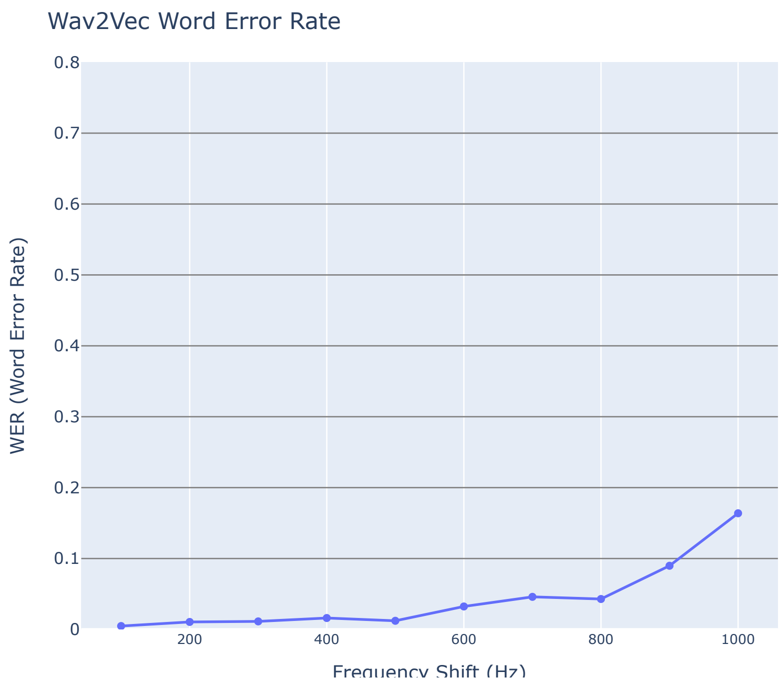
```
List_fsu = np.array([[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.125, 0.125, 0.125, 0.125],
[0.0, 0.0, 0.023255813953488372, 0.023255813953488372, 0.0, 0.0, 0.023255813953488372, 0.023255813953488372, 0.069767441860465
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.09090909090909091, 0.09090909090909091],
[0.047619047619047616, 0.06349206349206349, 0.015873015873015872, 0.06349206349206349, 0.047619047619047616, 0.047619047619047
[0.0, 0.0, 0.03225806451612903, 0.03225806451612903, 0.03225806451612903, 0.03225806451612903, 0.06451612903225806, 0.06451612
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.030303030303030304, 0.0, 0.09090909090909091, 0.09090909090909091],
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.17647058823529413, 0.5294117647058824],
[0.0, 0.0, 0.0, 0.0, 0.0, 0.1111111111111111, 0.1111111111111111, 0.1111111111111111, 0.1111111111111111, 0.1111111111111111],
[0.0, 0.0, 0.0, 0.0, 0.0, 0.09090909090909091, 0.0, 0.0, 0.0, 0.36363636363636365],
[0.0, 0.041666666666666664, 0.041666666666666664, 0.041666666666666664, 0.041666666666666664, 0.041666666666666664, 0.04166666

# Using numpy sum
res_fsu = np.sum(List_fsu, 0)

# printing result
print("final list - ", str(res_fsu))
```

```
final list -  [0.04761905 0.10515873 0.11305356 0.16067261 0.12154378 0.32356398
 0.45934481 0.42904178 0.89784628 1.63819167]
```

```
# Define a list of floating-point numbers
myList_fsu = [0.04761905, 0.10515873, 0.11305356, 0.16067261, 0.12154378, 0.32356398, 0.45934481, 0.42904178, 0.89784628, 1.63

# Define an integer value
myInt = 10

# Create a new list by dividing each element of myList_sil by myInt
newList_fsu = [x / myInt for x in myList_fsu]

# Print the new list
print(newList_fsu)
```

```
[0.004761905, 0.010515873, 0.011305355999999999, 0.016067261, 0.012154378, 0.032356397999999995, 0.045934481, 0.04290417
```

```
# Final Visualsiation Frequency Up

import plotly.express as px

def create_graph(x_values, y_values):
    ## Create the plot using Plotly Express
    fig = px.line(x=x_values, y=y_values, markers=True)

    ## Customize the plot
    fig.update_layout(
        xaxis_title="Frequency Shift (Hz)",
        yaxis_title="WER (Word Error Rate)",
        title="Wav2Vec Word Error Rate",
        legend_title="",
        showlegend=True,
        xaxis=dict(tickfont=dict(size=10)),
        yaxis=dict(showgrid=True, gridcolor='gray', gridwidth=0.5, range=[0, 0.8]),  # Set y-axis range to [0, 0.8]
        margin=dict(l=50, r=50, t=50, b=50),
    )

    ## Show the plot
    fig.show()

## Example data
x_data = [100, 200, 300, 400, 500, 600, 700, 800, 900, 1000]
y_data = [0.004761905, 0.010515873, 0.011305355999999999, 0.016067261, 0.012154378, 0.032356397999999995, 0.045934481, 0.04290

## Call the function to create the graph
create_graph(x_data, y_data)
```



Final Frequency down

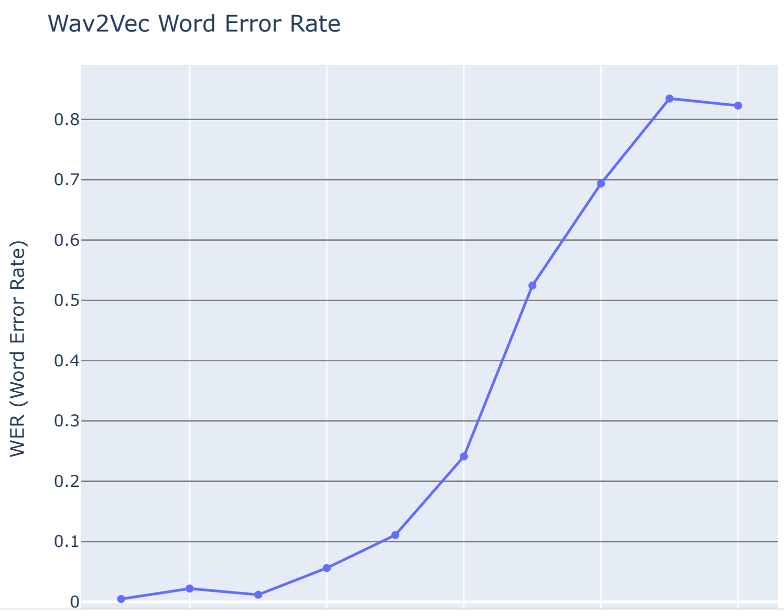
```
List_fsd = np.array([[0.0, 0.0, 0.0, 0.25, 0.25, 0.75, 0.875, 1.0, 1.0, 0.875],
[0.0, 0.0, 0.0, 0.023255813953488372, 0.046511627906976744, 0.046511627906976744, 0.13953488372093023, 0.5813953488372093, 0.8
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.8181818181818182, 1.2727272727272727, 1.4545454545454546, 1.1818181818181819],
[0.047619047619047616, 0.06349206349206349, 0.015873015873015872, 0.06349206349206349, 0.047619047619047616, 0.047619047619047
[0.0, 0.03225806451612903, 0.0, 0.0, 0.03225806451612903, 0.25806451612903225, 0.7096774193548387, 0.8709677419354839, 0.96774
[0.0, 0.0, 0.06060606060606061, 0.030303030303030304, 0.030303030303030304, 0.09090909090909091, 0.18181818181818182, 0.272727
[0.0, 0.0, 0.0, 0.0, 0.0, 0.11764705882352941, 0.29411764705882354, 0.5294117647058824, 0.7058823529411765, 0.8235294117647058
[0.0, 0.0, 0.0, 0.1111111111111111, 0.4444444444444444, 0.4444444444444444, 0.7777777777777778, 0.7777777777777778, 0.88888888
[0.0, 0.0, 0.0, 0.0, 0.09090909090909091, 0.36363636363636365, 0.6363636363636364, 0.8181818181818182, 0.9090909090909091, 0.9
[0.0, 0.125, 0.041666666666666664, 0.08333333333333333, 0.16666666666666666, 0.2916666666666667, 0.75, 0.75, 0.875, 0.75]])

# Using numpy sum
res_fsd = np.sum(List_fsd, 0)

# printing result
print("final list - ", str(res_fsd))
```

```
final list -  [0.04761905 0.22075013 0.11814574 0.56149535 1.10871197 2.41049882
 5.24596343 6.93668106 8.34670476 8.22911742]
```

```
# Define a list of floating-point numbers
myList_fsd = [0.04761905, 0.22075013, 0.11814574, 0.56149535, 1.10871197, 2.41049882, 5.24596343, 6.93668106, 8.34670476, 8.22

# Define an integer value
myInt = 10

# Create a new list by dividing each element of myList_sil by myInt
newList_fsd = [x / myInt for x in myList_fsd]

# Print the new list
print(newList_fsd)
```

```
[0.004761905, 0.022075012999999997, 0.011814574, 0.056149534999999993, 0.110871197, 0.241049882, 0.524596343, 0.69366810
```

```
# Final Visualsation Frequency Down

import plotly.express as px

def create_graph(x_values, y_values):
    # Create the plot using Plotly Express
    fig = px.line(x=x_values, y=y_values, markers=True)

    # Customize the plot
    fig.update_layout(
        xaxis_title="Negative Frequency Shift(Hz)",
        yaxis_title="WER (Word Error Rate)",
        title="Wav2Vec Word Error Rate",
        legend_title="",
        showlegend=True,
        xaxis=dict(tickfont=dict(size=10)),
        yaxis=dict(showgrid=True, gridcolor='gray', gridwidth=0.5),
        margin=dict(l=50, r=50, t=50, b=50),
    )

    # Show the plot
    fig.show()

# Example data
x_data = [100, 200, 300, 400, 500, 600, 700, 800, 900, 1000]
y_data = [0.004761905, 0.022075012999999997, 0.011814574, 0.056149534999999993, 0.110871197, 0.241049882, 0.524596343, 0.69366


# Call the function to create the graph
create_graph(x_data, y_data)
```



✓ 0s completed at 23:42