IDIAP COMMUNICATION REPORT

INTEGRATING LARGE LANGUAGE MODELS AND ASR SYSTEMS USING CONFIDENCE MEASURES AND PROMPTING

Maryam Naderi[a]

Idiap-Com-02-2024

JULY 2024

[a]Idiap Research Institute

Centre du Parc, Centre du Parc, Rue Marconi 19, CH - 1920 Martigny
T +41 27 721 77 11 F +41 27 721 77 12 info@idiap.ch www.idiap.ch

Integrating large language models and ASR systems using confidence measures and prompting

Author	: Maryam Naderi
Student number	: 22 - 695 - 068
Project supervisor	: Dr. Mathew Magimai Doss
Project co-supervisor	: Dr. Enno Hermann
Company supervisor	: Alexandre Nanchen

Acknowledgements

First of all, I would like to thank my supervisor, Dr. Mathew Magimai Doss, for giving me the opportunity to do my project in his research group. I am very grateful to him for his constant support, encouragement, and wise counsels during my master studies. I would like to express my gratitude to Dr. Enno Hermann for his invaluable assistance, patiently answering my questions, and helping to debug my codes. I would like to thank Mr. Alexandre Nanchen for his insightful exchanges and the technical support he provided. Additionally, I am grateful to Mr. Oliver Bornet for his interesting discussions and technical support with the API key and local LLMs. I also wish to thank Dr. Sevada Hovsepyan for his interesting discussions on understanding human speech comprehension.
Finally, I want to thank all my friends and family for their unwavering support and encouragement.

Martigny, July 31, 2024 M. N.

Abstract

As large language models (LLMs) grow in parameter size and capabilities, such as interaction through prompting, they open up new ways of interfacing with automatic speech recognition (ASR) systems beyond rescoring n-best lists. This work investigates post-hoc correction of ASR transcripts with LLMs. To avoid introducing errors into likely accurate transcripts, we propose a variety of confidence-based filtering methods: sentence-level confidence, lowest-word confidence, and correction of specific low-confidence words.

In the first method, sentence-level confidence, we utilize LLM to correct transcription errors if the sentence confidence score of the transcription falls below a certain threshold. For the lowest-word confidence method, we submit transcriptions for correction by LLM only if the lowest-word confidence in the sentence is below a certain threshold. Finally, in the last filtering method, we instruct the LLM to correct only certain words with confidence scores lower than a specific threshold.

We determine the optimal threshold for each approach using the Librispeech `dev-clean` and `dev-other` datasets and evaluate them using the Librispeech `test-clean` and `test-other` datasets. Additionally, we examine the impact of various prompts, language models (LMs), and Whisper models on the quality of corrected ASR transcriptions. Our findings suggest that these methods can enhance the performance of less competitive ASR systems. Furthermore, we conduct an error analysis and explore the detailed modifications made by the LLM and the corresponding parts of speech in the sentence, evaluating whether each modification corrects existing errors or introduces new errors into the transcription.

Contents

List of Figures

List of Tables

1 Introduction

Speech perception is a complex process that relies not only on acoustic information but also on environmental information, visual cues, context, and other factors. Consequently, speech perception in the brain is organized in a hierarchical and highly parallel processing network, where information on different time scales, about different linguistic units and from different modalities is analyzed to decipher the semantic content of speech (Rauschecker and Scott, 2009). Due to the reliance on these contextual cues during speech perception, humans can be considered "noisy listeners": to successfully understand the message, humans do not need to recognize every part of the speech they hear. Our predictive brain can replace the missing information based on the available contextual information (Miller and Isard, 1963; Shannon et al., 1995; Gwilliams et al., 2023; Sohoglu et al., 2012).

ASR systems operate in a similar way. An acoustic model first processes the speech signal and identifies linguistic units, such as phonemes. Then, a LM, which encodes prior knowledge about the likelihood of different word sequences, helps to find the most likely transcription given the potentially noisy information from the acoustic model.

The number of parameters in LMs and their performance on numerous benchmarks even without task-specific fine-tuning has increased so much in recent years that we commonly refer to them as LLMs. Especially instruction-tuned LLMs offer new possibilities for downstream applications through their prompting mechanism (Ouyang et al., 2022). LLMs can also work directly with very long input contexts, obviating the need to specifically adapt LMs to recently observed sequences (Krause et al., 2018).

Motivated by these developments, we investigate whether combining an ASR system with a additional LLM can address ASR errors.

The primary challenge is the trade-off between leveraging LLMs to correct errors in low-accuracy transcripts while minimizing the risk of introducing new errors in more accurate ones. To reduce the chance of the LLM introducing new errors into the transcript, we propose leveraging LLMs based on the ASR confidence scores of the original transcripts. Figure 1.1

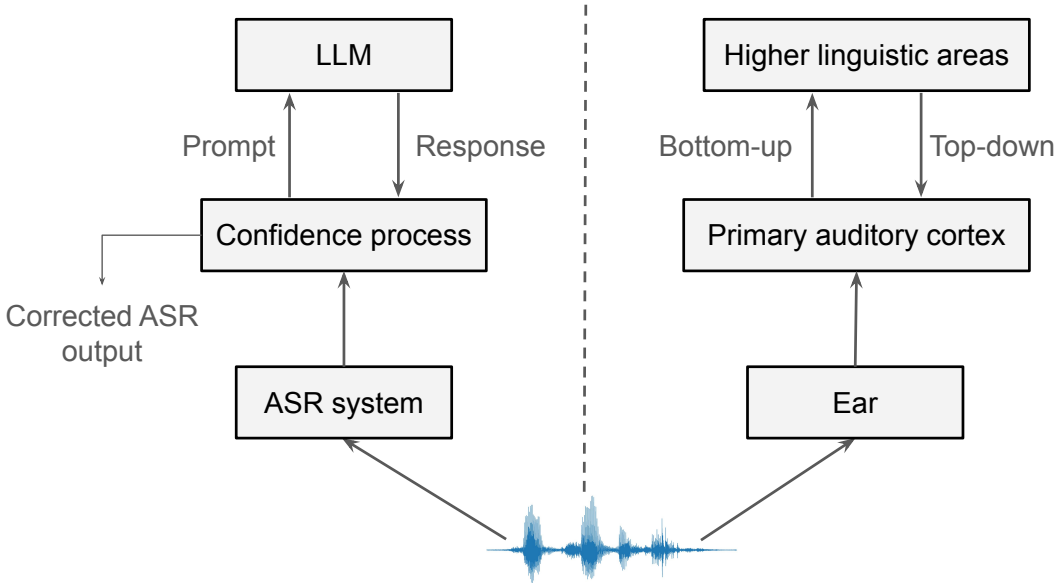

Figure 1.1: Proposed approach (left) and speech processing in the brain (right).

illustrates the proposed approach with an analogy to speech perception discussed before.

1.1 Contributions

The main contributions of this thesis are:

- We propose three filtering methods that rely on the ASR confidence scores. For the first one, we let the LLM correct only transcripts where the average confidence score of all tokens in the transcript falls below a given threshold. For the second one, we leverage LLM to correct only the transcripts where the lowest word confidence falls below a given threshold. For the third method, we prompt the LLM to correct only specific low-confidence words in the transcript, leaving the high-confidence words untouched. We name these methods sentence-level confidence, lowest-word confidence, and correction of specific words, respectively. We use English Librispeech datasets to evaluate our model.

- We study the impact of LLM, ASR model size, and various prompts on the effectiveness of LLM corrections.

- To gain deeper insights into the LLM's behavior, we perform an error analysis. We begin by presenting concrete examples where LLM has corrected errors in the transcription and other examples where LLM performed poorly. We then analyze modifications made by the LLMs that resulted in new errors or corrected existing errors in the transcriptions and examine their corresponding parts of speech in the sentence.

A part of the work presented in this thesis has already been accepted in the form of a peer-reviewed paper titled "Towards interfacing large language models with ASR systems using confidence measures and prompting" at the Interspeech 2024 Conference.

1.2 Structure

The structure of the thesis is as follows: In Chapters 2 and 3, we give an overview of LLMs and ASR. In Chapter 4, we review previous works on applying LLMs to ASR. Chapter 5 details our approach and experimental setup and in Chapter 6 we present our results and analysis. Finally, we conclude this thesis in Chapter 7.

2 Overview of Language Models

A LM is a statistical model that assigns a probability, denoted as $p(w_1 w_2 \dots w_n)$, to a sequence of words, represented as $w_1 w_2 \dots w_n$. In simpler terms, it estimates the likelihood of a given sequence occurring within a language. For instance, the phrase:

```
I am a student in AI
```

should have a higher probability compared to a phrase made up of the same words but with a random arrangement like:

```
student a AI in am I
```

LMs typically possess a set of parameters that must be learned from a training dataset. This learning process, often referred to as training, enables the model to learn the structure and patterns within the language. Once trained, the LM can be utilized for making predictions, a procedure known as inference.

Assessing the performance of a LM requires evaluation using a separate, non-overlapping test dataset. The choice of evaluation metrics depends on the specific task the LM is designed to solve. For example, in classification tasks like sentiment analysis, accuracy—representing the ratio of correctly predicted instances to the total—is commonly used. One of the most important metrics for evaluating a LM in natural processing which is independent of the task is Perplexity. Perplexity is defined as:

$$\text{PPL} = p(w_1 w_2 \dots w_N)^{-\frac{1}{N}} \tag{2.1}$$

where N denotes the length of the sequence in the test set. Lower perplexity values indicate that the LM assigns higher probabilities to sentences within the test dataset, indicative of a better performance.

In the following, we provide a brief overview of basic topics in language modelling. We

begin with tokenization followed by the simplest LM, known as N-gram model, and then proceed with embedding and more advanced models such as recurrent neural networks and transformers.

2.1 Tokenization

The first step in the language modeling process is tokenization. Tokenization is the process of converting text into individual tokens, which could be words, subwords, or characters. The collection of these tokens is known as the vocabulary. Tokenization can be done simply by splitting the text based on whitespace or punctuation. Some special tokens such as `SOS`, `EOS`, and `UNK` are also added to the vocabulary to mark the start of sentence, end of sentence, and unknown words.

Some of the more advanced tokenization techniques that are used in LMs such as GPT and BERT include Byte-Pair Encoding (BPE) and wordpiece, which I will briefly explain in the following subsections.

2.1.1 Byte-Pair Encoding

BPE tokenization is a data-driven approach that learns token representations from the training dataset. During the training phase, BPE initializes a vocabulary comprising all unique characters present in the dataset. It then identifies the pair of most frequently occurring characters and merges them in the dataset, adding them to the vocabulary. This process is repeated until k merges are created, where k is a hyperparameter of the model. Once the BPE is trained, we use the vocabulary to segment the test dataset in the same order that the tokens have been merged (Sennrich et al., 2016).

2.1.2 WordPiece

WordPiece tokenizaion works similarly to BPE, namely it starts with a vocabulary set comprising unique characters in the training dataset and merges them to build larger units. However, instead of merging the two characters that occur most frequently in each iteration, it merges the two characters that maximizes the likelihood of the training dataset if the merge is done. For instance, characters c and h are merged if the ratio $\frac{p(ch)}{p(c)p(h)}$ is highest at a given iteration (Wu et al., 2016).

In the rest of this thesis we use the terms word and token interchangeably.

2.2 N-gram

The N-gram model (Shannon, 1948) relies on the assumption that the probability of observing a word at a particular position in a sequence only depends on the previous $N-1$ words and is independent of the words beyond that. This assumption, known as the Markov assumption, simplifies the modeling process by reducing the complexity of the language model while still capturing some of the contextual information within the text.

Mathematically, this implies that the probability of the word-sequence:

$$p(w_1 w_2 \dots w_n) = p(w_n|w_1 w_2 \dots w_{n-1}) p(w_{n-1}|w_1 w_2 \dots w_{n-2}) \dots p(w_2|w_1) p(w_1) \tag{2.2}$$

$$= \prod_{i=1}^{n} p(w_i|w_1 w_2 \dots w_{i-1}) \tag{2.3}$$

can be approximated as:

$$p(w_1 w_2 \dots w_n) \approx \prod_{i=1}^{n} p(w_i|w_{i-N+1} \dots w_{i-1}) \tag{2.4}$$

In the case of $N = 2$, bigram, the probability of each occurring word only depends on immediate previous word.

$$p(w_1 w_2 \dots w_n) \approx \prod_{i=1}^{n} p(w_i|w_{i-1}) \tag{2.5}$$

2.2.1 Training

The parameters of the bigram model are all the probabilities in Equation 2.5. These are estimated by the normalized counting of the occurrences of these words in the training set:

$$p(w_i|w_{i-1}) = \frac{Count(w_{i-1} w_i) + \alpha}{\sum_{w_j} \left(Count(w_{i-1} w_j) + \alpha \right)} \tag{2.6}$$

This is known as maximum likelihood estimation (MLE). The α-term, referred to as smoothing, is added to prevent the issue of zero probabilities for unseen words. It ensures that even if certain word combinations are absent from the training data, they still receive a non-zero probability during inference and evaluation.

2.2.2 Inference

Having trained the bigram model, we can utilize the model to perform text generation. For this purpose, we sample from the probability distribution provided by the model. We can do this in the following ways:

Greedy search

Starting from the SOS, we can sample the word with maximum likelihood at each step:

$$
\begin{aligned}
w_1 &= \arg\max_w p(w|SOS) \\
w_2 &= \arg\max_w p(w|w_1) \\
&\vdots \\
w_i &= \arg\max_w p(w|w_{i-1})
\end{aligned}
$$

We continue sampling next word until we get EOS. This sampling approach is known as greedy search (Sutskever et al., 2014).

Beam search

Instead of sampling the most likely word at each step, an alternative and better approach is to sample words in a way that maximizes the probability of the entire sentence. However, exploring all possibilities to obtain the most likely sentence is computationally intensive. Beam search addresses this issue by maintaining a beam of the most likely word sequences. This way, we expand multiple possible sequences and keep only the top-k sequences based on their cumulative probabilities. This allows for exploring a wider range of possibilities and often leads to more diverse and higher-quality outputs compared to greedy search (Sutskever et al., 2014). In the case of $k = 1$, beam search reduces to the greedy search.

2.3 Embedding

The bigram model lacks long-term dependency between words, as it only considers the current word when predicting the next word. While increasing the value of N in the N-gram model (e.g., tri-gram, four-gram) can mitigate this issue, it also results in these models becoming computationally more intensive, making them infeasible in practice. In the following sections, I explore more modern LMs such as RNNs and Transformers. Unlike the N-gram model, these LMs do not directly work with words but rather work with numerical vector representations of words known as embedding.

Embedding is the process of encoding words into vectors in a multidimensional vector space, where related words tend to be closer to each other. One basic embedding method is the one-hot representation, in which each word is represented by a vector of length $|V|$ (the number of words in the training set), where only one element is set to 1 (indicating the index of the word), and all other elements are set to 0. However, one-hot embeddings have three main limitations: sparsity, high dimensionality, and the inability to capture semantic similarities between words (all words in this representation are equidistant).

In this part, we briefly explain some of the embedding techniques that aim to address these issues.

2.3.1 Word2Vec

Proposed by Mikolov et al. (2013), Word2Vec overcomes the three challenges of one-hot representation by learning short, dense, and semantically meaningful representations for words that take into account the context. Word2Vec usually refers to skip-gram and continuous bag of words (CBOW) models.

Skip-gram

In this approach, a simple model is trained to predict the context words given a target word. The weights in the model serve as embedding vectors. For any target word, context words are defined as words that occur within a short window near the target word. The combinations of (t,c), where t is the target word and c is the context word, form positive examples. For each positive example, we generate some negative examples by randomly selecting words from the vocabulary that do not occur with the target word. We begin by randomly initializing two weight matrices (one for words as target and the other for words as context) with shape $(|V| \times d)$, where d is the embedding length and a hyperparameter of the model. Then we train a simple logistic regression model that maximizes/minimizes the similarity between the target word and the context words in positive/negative examples, respectively, in the training dataset. Similarity is defined as the dot product of the corresponding rows in the weight matrices. Once the model is trained, the sum of the corresponding rows of the weight matrices serve as the embedding vector for each token in the vocabulary.

CBOW

This approach is similar to skip-gram but instead of predicting the context words from the target, it is trained to predict the target word based on the context.

2.3.2 FastText

While successfully capturing some semantic similarity between words, Word2Vec fails to address the embedding of unknown words, even though these unknown words have been seen in the training dataset in different forms. For example, they may differ in prefixes, suffixes, or minor spelling errors. FastText addresses this issue by splitting each word into subwords using n-grams (Bojanowski et al., 2017). For instance, with 4-grams, the word "university" is split into the following subwords:

```
<uni univ nive iver vers ersi rsit sity ity>
```

Here, < and > mark the start and end of the word, respectively. Having split the words into subwords, FastText trains a simple model like skip-gram to learn embeddings for these subwords. The embedding of a word is then computed as the sum of the embeddings for its constituent subwords.

2.3.3 Contextual embedding

The embedding methods explained so far learn a fixed vector for each word, regardless of the context in which the word occurs. This is known as static embedding. For example, the word `bank` in the following contexts get the same embedding,

```
the bank of the river
deposit money into the bank
```

even though the meaning is different. Contextual embedding, on the other hand, takes into account the surrounding context and assigns different embeddings to the same word in different contexts. One can learn context-dependent embeddings by utilizing LMs such as RNNs like ELMo (Sarzynska-Wawer et al., 2021) or transformers such as BERT (Devlin et al., 2019) and GPT (Radford et al., 2018).

2.4 RNN

As previously mentioned, the N-gram model fails to consider context when predicting the next word. A Recurrent Neural Network (RNN) is a better alternative that takes into account the sequential nature of text data (Rumelhart et al., 1986; Cho et al., 2014). An RNN can be considered as a cyclic feed-forward neural network that is recursively applied to sequential data. By introducing a hidden state containing context information, at each time step t the RNN takes as input the current word's embedding vector x_t and the output from the previous time step, denoted by h_{t-1}, and outputs the updated hidden state h_t according to the formula:

$$h_t = \tanh(W_x x_t + W_h h_{t-1} + b) \tag{2.7}$$

Where W_x, W_h, and b are the parameters of the RNN (weights and biases) learned during training. The hidden state is a vector of some length that contains the information about the context as it gets updated in each time step. The length of the hidden state is a hyperparameter of the model and its initial value, h_0, is a vector of zeros.

The exact architecture of RNNs and training procedure (e.g., loss function) in RNNs depend on the specific task they aim to solve. RNNs can address a wide range of NLP tasks, categorized into many-to-many, many-to-one, and one-to-many. Many-to-many tasks include text generation, named entity recognition, and translation, etc., where both the input and output are sequences. Many-to-one tasks, such as sentiment analysis, have a sequence as input and a

single label or fixed-size vector as output. One-to-many tasks, like image captioning, take a fixed-size vector as input and produce a sequence as output. Various RNN architectures that can tackle these tasks are illustrated in Figures 2.1, 2.2, 2.3, and 2.4.

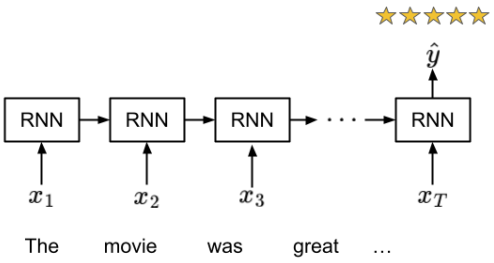

Figure 2.1: Many-to-one architecture, e.g., sentiment analysis

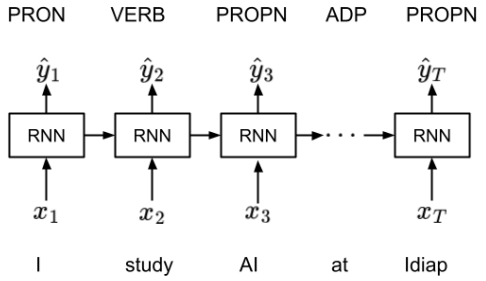

Figure 2.2: Many-to-many architecture, e.g., part-of-speech tagging

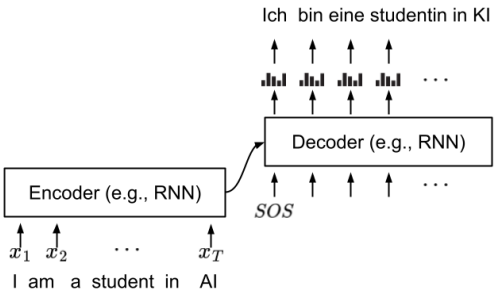

Figure 2.3: Encoder-decoder architecture, e.g., translation

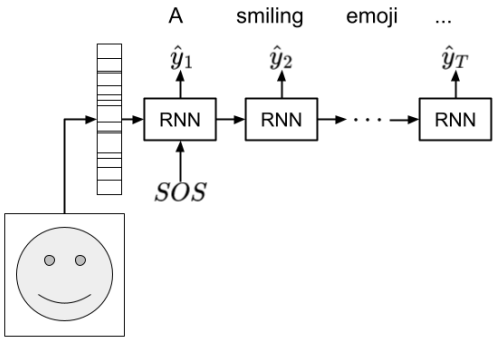

Figure 2.4: One-to-many architecture, e.g., image captioning

During inference, we provide the trained RNN with a `SOS` token and then iteratively generate text based on the previously generated tokens using one of the search algorithms greedy, beam search, etc. In figure 2.5 we show an example of inference in RNNs for text generation.

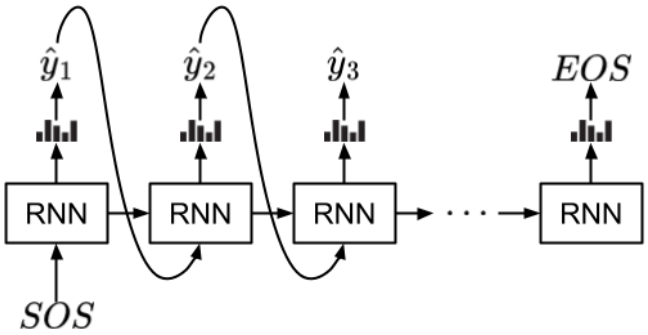

Figure 2.5: Inference in RNNs

Although RNNs can successfully handle sequential data, they suffer from two main issues:

1) being slow and non-parallelizable due to their recursiveness, and 2) inability to capture long-term dependency in longer sequences because of vanishing gradients problem. Two other variants of RNNs, namely Gated Recurrent Unitss (GRUs) (Cho et al., 2014) and Long Short-Term Memorys (LSTMs) (Hochreiter and Schmidhuber, 1997), have been proposed to tackle the second issue. Although they improved upon the original RNNs, these issues still remain.

In the next section, we will briefly explain an alternative LM, namely transformers, that have addressed both of these issues.

2.5 Transformer

Proposed by Vaswani et al. (2017) in the influential article "Attention is all you need", transformer address the challenges posed by RNNs and their variants LSTMs and GRUs by proposing the attention mechanism. The attention mechanism creates a rich context-aware representation for the words of a given sequence. In this representation, the dependencies between words are captured, regardless of how distant the words are in the sequence. In addition, the computation of this contextual representation for different words are independent and therefore can be performed in parallel.

Assuming that x_i is the original word embedding for the ith word in a sequence, a context-aware representation for this word can be calculated in the simplest case as:

$$x_i \rightarrow a_i \propto \sum_j s_{ij} x_j \tag{2.8}$$

Where s_{ij}s are the reweighing scores. A natural choice for these scores can simply be the dot product between the embedding vectors $s_{ij} = x_i \cdot x_j$

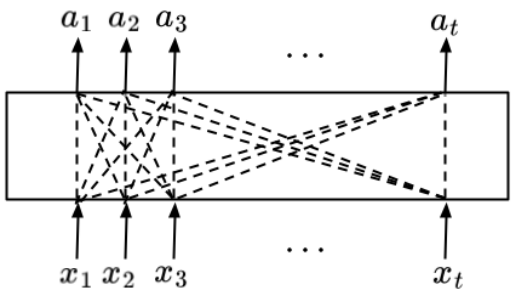

Figure 2.6: Visual representation of attention

But to get a more flexible representation that learns various aspects of the language, we add some parameters to the model, namely three matrices W_K, W_Q, and W_V, all with shapes $d \times d$

with d being the embedding dimension. These matrices are initialized randomly and become updated and learned during the training. Equation 2.8 is re-written as:

$$a_i = \sum_j \text{softmax}\left(\frac{q_i \cdot k_j}{\sqrt{d}}\right) v_j \tag{2.9}$$

or in matrix form:

$$\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_N \end{bmatrix} = \text{softmax}\left(\frac{1}{\sqrt{d}} \begin{bmatrix} q_1 \cdot k1 & q_1 \cdot k_2 & \dots & q_1 \cdot k_N \\ q_2 \cdot k1 & q_2 \cdot k_2 & \dots & q_2 \cdot k_N \\ \vdots & & \ddots & \vdots \\ q_N \cdot k_1 & q_N \cdot k_2 & \dots & q_N \cdot k_N \end{bmatrix} \right) \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_N \end{bmatrix} \tag{2.10}$$

where $k_j = x_j W_K$, $q_i = x_i W_Q$, and $v_j = x_j W_V$ are usually called keys, queries, and values and the term $\frac{1}{\sqrt{d}}$ is added to avoid the large values in the softmax function. We can even define multiple keys, queries, and values matrices which is known as multihead attention.

In some situations, such as next prediction, that we need to prevent the flow of information to the future, we use a mask matrix in the argument of the softmax function, known as masked attention. This is demonstrated below for a sequence of length 4:

$$\begin{bmatrix} 0 & -\infty & -\infty & -\infty \\ 0 & 0 & -\infty & -\infty \\ 0 & 0 & 0 & -\infty \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.11}$$

The attention mechanism is a component of a larger architecture known as the encoder. In this block, the original embeddings are combined with the output of the attention mechanism and then normalized. Normalization facilitates the learning process during training. The output then passes through a fully-connected neural network and is added to itself with another normalization. The final output of the encoder block maintains the same dimensionality as the input, so for each input token of dimension d, there will be an output vector of dimension d. The transformer architecture is depicted in Figure 2.7. Originally proposed for machine translation tasks, this transformer comprises a series of six stacked encoder blocks connected to a series of six stacked decoder blocks. The decoder block is similar to the encoder block, except that it has two attention mechanisms. The first one is a masked attention focused on the translation process, while the second one is a standard attention mechanism that attends to both the source and the translation.

Two primary LMs derived from transformers, which serve as foundational architectures for many LLMs, include GPT and BERT. Below, I provide a brief overview of these LMs.

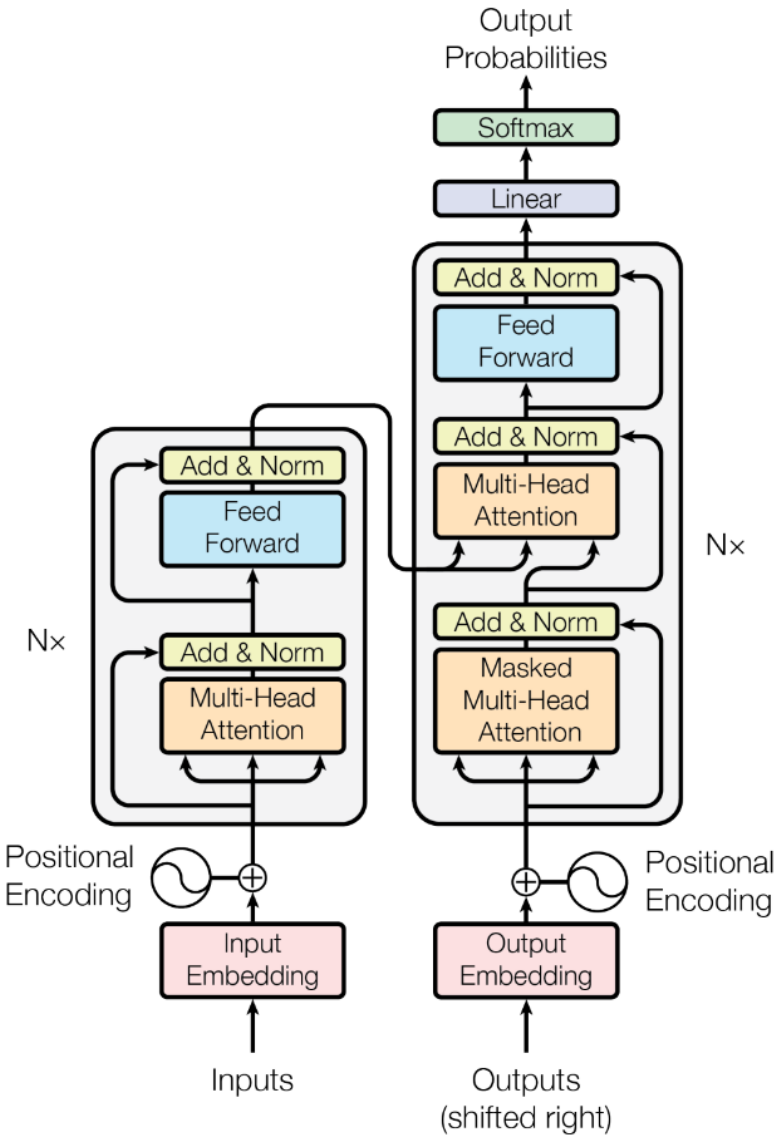

Figure 2.7: Transformer architecture (image from Ref. Vaswani et al. (2017))

2.5.1 GPT

The Generative Pre-trained Transformer (GPT), proposed by Radford et al. (2018), is based on the transformer decoder. Utilizeing a masked attention mechanism, meaning that words only attend left-to-right, it models a unidirectional language model. The training of these LMs involve two phases: unsupervised pre-training and supervised fine-tuning. In the pre-training phase, the GPT model is trained to predict the next word given a sequence of words. In the fine-tuning phase, it is further refined and trained on various tasks such as text classification, question answering, text entailment, etc. GPT-3, for example, comprises 96 stacked decoder layers with a total of 175 billion trainable parameters.

2.5.2 BERT

Bidirectional Encoder Representations from Transformers (BERT), proposed by Devlin et al. (2019), is based on the transformer encoder. Unlike the GPT model, BERT, as the name suggests, is a bidirectional language model, meaning that each word attends to both left and right. Similar to GPT, the training of BERT also includes pre-training and fine-tuning. In the pre-training phase, the model is trained to predict 1) masked words in a given sequence of words and 2) whether or not two consecutive sentences in the input are related. For the first task, to predict masked words in the sequence, 20 percent of the words in the input sequence are randomly replaced either with a special `<MASK>` token or a random word from the vocabulary, and the model predicts these masked words. For the second task, a special token `SEP` is defined to separate two sentences within the input and the model predicts whether the two sentences in the input seperated by `SEP` are related.

In the fine-tuning phase, the model is further modified and trained to learn downstream tasks such as classification, question-answering, etc. BERT (large version) contains 24 stacked layers of transformer encoder blocks and comprises about 340 million trainable parameters.

2.6 Large Language Models

Due to the parallelizability of transformers and the availability of highly optimized deep learning libraries with GPU support, such as TensorFlow (Abadi et al., 2015) and PyTorch (Paszke et al., 2019), it is nowadays feasible to train transformers of very large sizes on vast amounts of text data. These transformers are typically designed to process long contexts ranging from 1024 to 4096 input tokens and contain hundreds of millions to hundreds of billions of trainable parameters. They are trained on extensive text corpora, including datasets like Wikipedia, the entire internet archive, and numerous books. These LMs are pre-trained in an unsupervised manner, for example using tasks like next-word prediction. This approach has proven effective, as it has been shown that many NLP tasks can be framed as language modeling or next-word prediction tasks (Jurafsky, 2000).[1]

However, since LMs are trained primarily to generate text (by predicting either the next or masked tokens), they might not always follow user instructions accurately when prompted. Additionally, they are prone to generating untruthful or toxic text. To address this, Ouyang et al. (2022) proposed instruction-tuning in LLMs. In this approach, human labelers provide desired outputs for a set of diverse prompts across various NLP tasks, such as text generation, question answering, summarization, etc., and the outputs from human labelers, together with the corresponding prompts, are used for further supervised fine-tuning of the model based on the given instructions in the prompts.

Additionally, they proposed using reinforcement learning from human feedback to align LLMs

[1] For example, a sentiment detection task can be cast into next-word prediction task by inputting the LM with 'the sentiment of the sentence "the movie was great!" is' and the LM, most likely, predicts "positive" as next words.

with user instructions. In this approach, human labelers rank multiple LLM outputs for a given prompt, and a reward model is trained to predict the human-preferred output.

By undergoing the instruction-tuning, the model learns to generate less toxic, more truthful, and more human-preferred outputs. Ouyang et al. (2022) showed that outputs from an instruction-tuned GPT model with 1.3B parameters are preferred in human evaluations over the outputs of a 175B GPT model without instruction tuning.

3 Overview of Automatic Speech Recognition

ASR refers to the task of transcribing speech. It has several important applications, such as real-time captioning and voice assistants.

Transcribing speech can vary in difficulty. While transcribing clearly read speech or short, isolated phrases may be relatively straightforward, the more challenging scenarios involve transcribing people's conversations in real-time, often with multiple speakers having different dialects and accents, and amid background noise. Although they are not perfect, state-of-the-art ASR systems have made significant progress in recent years. These systems leverage advanced neural network architectures and large datasets to achieve high accuracy rates.

In this chapter, I provide an overview of ASR and some underlying models. The chapter begins with a discussion of feature extraction, detailing how acoustic features are extracted from audio speeches. Next, I describe two main ASR approaches: conventional and end-to-end methods. Within the conventional approach, I describe hidden markov models (HMMs) while within the end-to-end approach, I explain Connectionist Temporal Classification (CTC), Recurrent Neural Network Transducer (RNN-T), and attention-based models. Finally, I conclude this chapter with an explanation of how the accuracy of ASR transcripts is evaluated.

3.1 Feature extraction

In the first step in ASR, speech needs to be converted into numerical features that can be processed further with models. Sound consists of continuous changes in air pressure, which are detected and converted to electrical signals by microphones. In the case of speech, the sound is generated by the human vocal system and contains meaningful information or language. An analog-to-digital converter (ADC) converts this electrical signal into a discrete, digitized signal in both time and amplitude through processes called sampling and quantization.

3.1.1 Sampling and Quantization

An ADC converts the electrical signal into a discrete signal with digitized amplitudes through sampling and quantization. Sampling refers to measuring the amplitude at regular time intervals. The number of measurements taken per second is called the sampling rate. To capture a signal with a frequency f, the required sampling rate is at least $2f$, known as the Nyquist limit. The higher the sampling rate, the better the resolution of the signal. The human auditory system usually perceives sound frequencies below 8000 Hz. Therefore, a sampling frequency of 16 kHz is considered sufficient for ASR applications.

Quantization refers to mapping continuous amplitude values into a finite number of discrete levels. The bit depth, which is the number of bits used to store each amplitude measurement, determines the number of these levels. In ASR applications, a bit depth of 16 is typically used. This allows the amplitude to be quantized into 65,536 levels (from -32,768 to 32,767), with each amplitude value rounded to the nearest integer within this range.

3.1.2 Frequency Domain

The discrete audio signal from the previous section contains values of amplitude at various sampled times. However, the audio signal can also be represented in the frequency domain using the Discrete Fourier Transform (DFT). By representing the audio signal in the frequency domain, the frequencies that make up the audio and their strengths can be extracted.

3.1.3 Mel-Spectrogram

To extract features from each short segment of audio corresponding to a part of a phoneme, the frequency spectrum is calculated by applying the DFT to the signal using a sliding window of 25 ms with a stride of 10 ms. This produces a spectrogram, which shows the evolution of frequencies and their amplitudes over time.

As humans are more sensitive to changes in lower frequencies than higher ones, the spectrogram is often adjusted by taking the logarithm of the frequencies to mimic human hearing perception, resulting in a mel-spectrogram (Davis and Mermelstein, 1980). Similarly, the logarithm of the amplitudes is taken to account for human sensitivity to changes in lower amplitudes, resulting in a log-mel-spectrogram. Log-mel-spectrograms are widely used as inputs in ASR models, such as the Whisper model (Radford et al., 2023).

3.1.4 Mel-Frequency Cepstral Coefficients

Another widely used feature in ASR models, such as HMMs, is Mel-Frequency Cepstral Coefficients (MFCCs). These features are derived from log-mel-spectrograms by further processing, which includes calculating cepstral coefficients as well as their delta (first-order differences)

and delta-delta (second-order differences, representing acceleration) coefficients. This processing typically results in feature vectors of size 39 for each window, effectively capturing the essential characteristics of the speech while reducing data dimensionality (Davis and Mermelstein, 1980).

3.2 ASR approaches: Conventional Vs. End-to-end

An ASR system takes as input a sequence of acoustic features corresponding to spoken speech, for example, MFCC features denoted by $F = (f_1, f_2, \ldots, f_T)$, and outputs the text, represented by letters, words, or BPE tokens denoted by $W = (w_1, w_2, \ldots, w_n)$. Therefore, the task is to find the mapping:

$$p(w_1, w_2, \ldots, w_n | f_1, f_2, \ldots, f_T)$$

In the past, this task was approached by splitting the model into three separate parts: the acoustic model, lexicon, and language model:

$$p(W|F) \propto \sum_L P(F|L)\, p(L|W)\, p(W)$$

where $p(F|L)$ represents the acoustic model, solved using a HMM with a Gaussian Mixture Model (HMM-GMM), $p(L|W)$ denotes the lexicon, and $P(W)$ is the language model. The acoustic model takes in the acoustic features as input and outputs the phonemes that constitute the words. This strategy is reasonable due to the vast number of words, for example 20k to 50k words in English, contrasted with the comparatively smaller set of phonemes. The lexicon maps these phonemes to form words. Finally, the language model is used to assign probabilities to sequences of words, ensuring that the resulting transcriptions are grammatically and contextually coherent. However, with the advent of deep learning and modern architectures like attention mechanisms, we can now more efficiently map the acoustic features directly to the spoken text. This approach is known as end-to-end.

In the following sections, I give a brief overview of the conventional HMM-based[1] ASR and end-to-end approaches in ASR: attention-based, CTC, and RNN-T.

3.2.1 Hidden Markov Model

The acoustic model in conventional ASR approach typically employs HMM. In this section, we introduce HMMs and briefly describe their usage in ASR as acoustic model.

[1] Since in this thesis, exclusively an end-to end ASR model is employed, the HMM section can be skipped and the readers can proceed to the subsequent sections.

A HMM is a statistical model to describe sequences of observable events that are believed to depend on underlying, unobservable states. HMM is Widely employed in various fields including speech recognition, bioinformatics, and natural language processing. HMM comprises a set of hidden states where each state is not directly observable. These states emit observations, $X = x_1, x_2, \ldots, x_T$, with each observation corresponding to a particular measurable outcome.

A first-order HMM makes two key assumptions:

- The probability of being in a particular state at the immediate future state depends only on the current state, expressed as:

$$P(s_{t+1}|s_1, s_2, \ldots, s_t) = P(s_{t+1}|s_t) \tag{3.1}$$

- The probability of a particular observation at the current time depends only on the current state, i.e.:

$$P(x_i|s_1, \ldots, s_T, x_1, \ldots, x_T) = p(x_i|s_i) \tag{3.2}$$

With these assumptions, the probability of a given sequence of states and observations can be written as:

$$p(x_1, \ldots, x_T, s_1, \ldots, s_T) = p(x_1, \ldots, x_T|s_1, \ldots, s_T)p(s_1, \ldots, s_T) \tag{3.3}$$

$$= \prod_{i=1}^{T} p(s_i|s_{i-1}) \prod_{i=1}^{T} p(x_i|s_i) \tag{3.4}$$

The parameters of an HMM include the initial, transition, and emission probabilities:

- **Initial probabilities** HMM begins from one of the hidden states, with the initial state probability vector represented as $\pi_i = p(s_1 = i) = p(s_1 = i|s_0)$. The sum of all initial probabilities should be 1.
- **Transition probabilities** Modeling the transition between hidden states is achieved using a state transition probability matrix, $A = [a_{ij}]$, where a_{ij} represents the probability of transitioning from state i to state j, $p(s_{t+1} = j|s_t = i)$. The sum of probabilities of all transitions from a particular state must equal 1.
- **Emission probabilities** Furthermore, HMM models the emission of observations from each state using an emission probability matrix, $B = [b_j(k)]$, where $b_j(k)$ denotes the probability of emitting observation o_k from state j, $b_j(k) = P(o_t = k|s_t = j)$.

In the context of ASR, the states correspond to distinct phonemes, usually 3 states for each phoneme, and the observations correspond to acoustic features. A graphical representation of HMM is shown in Fig. 3.1.

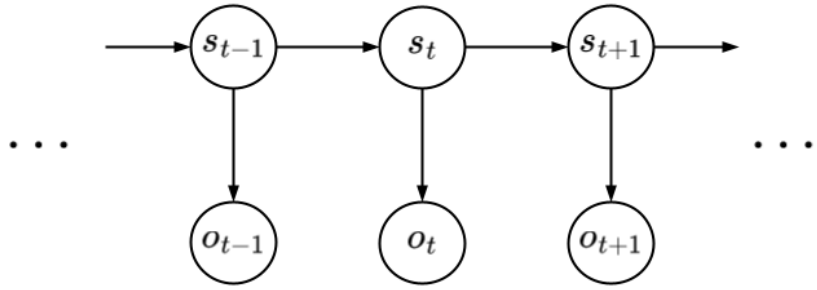

Figure 3.1: Graphical representation of an HMM

The problem that HMM aims to solve in the context of ASR is to find the sequence of most likely hidden states, or phonemes, based on a given speech, or a sequence of acoustic features, i.e.:

$$\underset{s_1,\ldots,s_T}{\arg\max}\, p(s_1,\ldots,s_T|o_1,\ldots,o_T) \tag{3.5}$$

Assuming that the parameters of the HMM are known, for an HMM with N hidden states and a sequence of T observations, there are N^T possible alignments of hidden states. Therefore, finding the most likely alignment of phonemes in a brute-force comparison takes a time complexity of N^T, which makes it infeasible in practice. Fortunately, we can use the Viterbi algorithm, which is based on dynamic programming, to solve this in N^2T time complexity.

Viterbi Algorithm

Viterbi is a dynamic programming approach that finds the most likely sequence of hidden states for a given sequence of observations (Forney, 1973). It defines values $v_t(j)$ as the highest probability of any path that ends in state j at time t based on the first t observations.

$$v_t(j) = \underset{s_1,\ldots,s_{t-1}}{\arg\max}\, p(s_1,\ldots,s_{t-1},x_1,\ldots,x_t,s_t=j) \tag{3.6}$$

$$= \max_{i}^{N} v_{t-1}(i)\,a_{ij}\,b_j(o_t) \tag{3.7}$$

By recursively computing $v_t(j)$, the Viterbi algorithm finds the most probable sequence of hidden states. The algorithm for Viterbi is depicted in Algorithm 1.

Algorithm 1 Viterbi Algorithm

Initialize for each state i: $v_1(i) \leftarrow \pi_i \cdot b_i(o_1)$ and $p_1(i) \leftarrow 0$
for each time step $t = 2$ to T **do**
  for each state j **do**
    $v_t(j) \leftarrow \max_i \left(v_{t-1}(i) \cdot a_{ij} \cdot b_j(o_t)\right)$
    $p_t(j) \leftarrow \arg\max_i \left(v_{t-1}(i) \cdot a_{ij} \cdot b_j(o_t)\right)$
  end for
end for
Best last state $\leftarrow \arg\max_i v_T(i)$
Backtracking:
$S[T] \leftarrow$ Best last state
for each time step $t = T-1$ down to 1 **do**
  $S[t] \leftarrow p_t(S[t+1])$
end for
Return S

Forward-Backward Algorithm

In practice, the parameters of the HMM are often not known and need to be estimated. This is typically done using the forward-backward or Baum-Welch algorithm (Baum et al., 1972). The Baum-Welch algorithm is a version of Expectation-Maximization (EM) algorithm where we start with an initial estimate of the model parameters and iteratively update them. To formulate this problem, we define two key variables: forward and backward probabilities.

- **Forward probability**, denoted by $\alpha_t(j)$, is the probability of being in state j at time t and observing the sequence of observations $o_1, \ldots, o_t$. It is defined as $\alpha_t(j) = p(o_1, \ldots, o_t, s_t = j)$. The forward probability at time t can be computed recursively using the forward probabilities from the previous time step $t-1$, allowing for dynamic programming:

$$\alpha_t(j) = \sum_i \alpha_{t-1}(i) a_{ij} b_j(o_t) \tag{3.8}$$

- **Backward probability**, denoted by $\beta_t(i)$, is the probability of observing the sequence of observations $o_{t+1}, \ldots, o_T$ given that the system is in state i at time t. It is defined as $\beta_t(i) = p(o_{t+1}, \ldots, o_T | s_t = i)$. Similar to the forward probability, the backward probability can be computed recursively using dynamic programming:

$$\beta_t(i) = \sum_j a_{ij} b_j(o_{t+1}) \beta_{t+1}(j) \tag{3.9}$$

Using the forward and backward probabilities, we can estimate the transition and emission probabilities as follows:

$$\hat{a}_{ij} = \frac{\sum_t \xi_t(i,j)}{\sum_t \sum_k \xi_t(i,k)} \tag{3.10}$$

$$\hat{b_j}(v_k) = \frac{\sum_{t \text{ s.t. } o_t = v_k} \gamma_t(j)}{\sum_t \gamma_t(j)} \tag{3.11}$$

where $\xi_t(i,j)$ and $\gamma_t(j)$ are:

$$\xi_t(i,j) = p(s_t = i, s_{t+1} = j | o_1, \ldots, o_T) = \frac{\alpha_t(i) a_{ij} b_j(o_{t+1}) \beta_{t+1}(j)}{\sum_j \alpha_t(j) \beta_t(j)} \tag{3.12}$$

$$\gamma_t(j) = \frac{p(s_t = j, o_1, \ldots, o_T)}{p(o_1, \ldots, o_T)} = \frac{\alpha_t(j) \beta_t(j)}{\sum_j \alpha_t(j) \beta_t(j)} \tag{3.13}$$

In the context of ASR, the observations are typically 39-dimensional MFCCs. Therefore, a GMM is often used for the emission probabilities, i.e.:

$$b_j(o_t) = \sum_{m=1}^{M} w_{jm} \mathcal{N}(o_t; \mu_{jm}, \Sigma_{jm}) \tag{3.14}$$

With this modification, the updating formula for $\mu_j m$, $\Sigma_j m$, and w_{jm} will be:

$$\hat{\mu}_{jm} = \frac{\sum_t \xi_{tm}(j) \cdot o_t}{\sum_t \sum_m \xi_{tm(j)}} \tag{3.15}$$

$$\hat{w}_{jm} = \frac{\sum_t \xi_{tm(j)}}{\sum_t \sum_k \xi_{tk}(j)} \tag{3.16}$$

$$\hat{\Sigma}_{jm} = \frac{\xi_{tm}(j)(o_t - \mu_{jm})(o_t - \mu_{jm})^T}{\sum_t \sum_k \xi_{tm}(j)} \tag{3.17}$$

where $\xi_{tm}(j) = \sum_i \xi_{tm}(i,j)$.

The modified EM algorithm for training an HMM with GMM emission probabilities involves iterating between updating the HMM parameters (using the forward-backward algorithm) and updating the GMM parameters for each state.

The modified EM algorithm is shown below:

This iterative process continues until the parameters converge, yielding an HMM that best fits the training data. Once the HMM is trained, the Viterbi algorithm can be used to find the

Algorithm 2 EM Algorithm for HMM with GMM Emission Probabilities

1: Initialize HMM parameters: initial probabilities π_i, transition probabilities a_{ij}, and GMM parameters for emission probabilities $b_j(o_t)$.
2: **repeat**
3: **E-step:** Compute forward probabilities $\alpha_t(j)$ and backward probabilities $\beta_t(i)$ for all states and time steps.
4: Compute $\xi_t(i,j)$ and $\gamma_t(j)$ using $\alpha_t(j)$ and $\beta_t(i)$.
5: **M-step:** Update transition probabilities.
6: Update GMM parameters for emission probabilities.
7: **until** convergence

most likely sequence of phonemes. These phonemes can then be mapped to words using a lexicon. Finally, a LM is used to improve the accuracy of the recognized words. The LM assigns probabilities to sequences of words, ensuring that the resulting transcription is grammatically and contextually coherent.

3.2.2 CTC

One of the most significant challenges in ASR is that the sequence of audio features is often much longer than the number of output words, and determining which audio features correspond to specific words – a problem known as alignment – poses a major obstacle. Proposed by (Graves et al., 2006), CTC is an end-to-end approach used in ASR that addresses the alignment problem. In this approach, a deep learning model, typically an RNN or transformer encoder, processes the input sequence of acoustic features (or subsampled acoustic features) and outputs a probability distribution over all possible characters, including a special character called blank, denoted by ϵ, for each time step in the input sequence. Formally, the model outputs:

$$p_t(a_t|X) \tag{3.18}$$

where a_t represents the character at time step t, and X represents the input sequence of acoustic features.

Each possible sequence of output characters is called an alignment. The inclusion of the blank character ϵ allows the model to handle repeated characters and variable silencing in different pronunciations effectively.

To generate the final transcription from an alignment, a collapsing function is applied. This function first merges consecutive duplicate characters that are not separated by a blank and then removes all blank characters from the merged sequence. The resulting sequence should represent the model's predicted transcription.

The visual representation of the CTC network is depicted in Figure 3.2.

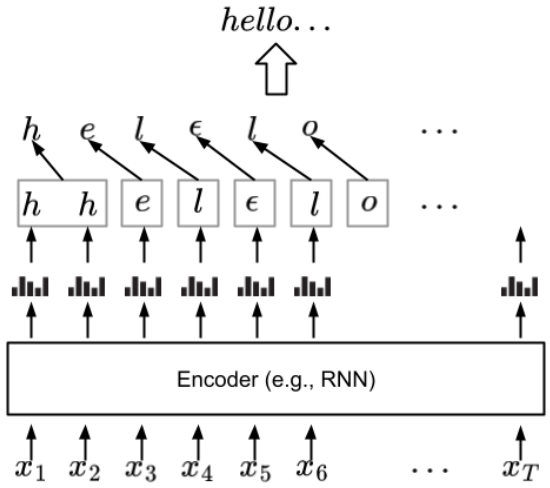

Figure 3.2: Visual representation of CTC model.

To better understand how CTC works, it's common to visualize the process using a two dimensional grid of nodes. The horizontal axis of the grid represents the time step and the vertical axis represents the characters , also called states, delimited by a blank with blanks in the start and end. Each node in the grid at position (t, s) represents a possible state (character or blank) at each time step and its value is the per time-step probabilities, $p_t(a_t = s|X)$, denoted as $\hat{y}_t^s$.

In order to ensure a valid alignment, we need to make a few constraint about the transition within nodes:

- First node at the first time step can be a ϵ or the first character of the word and last node at the final time-step should be the final ϵ or the last character of the word.
- The nodes follow a sequence which travels monotonically from top left to bottom right. This can be rewritten in terms of transition rules for each state s_i at time t as:
  - Transition to the same state $(t+1, s_i)$ (stay in the same state).
  - Transition to the next state $(t+1, s_{i+1})$ (move to the next character).
  - If s_i is not a ϵ, transition to the second next state $t+1, s_{i+2})$ (handle blanks).

In Figure 3.3 I showed some of the valid alignments for the word "hello" and summing 8 input time steps.

These black, blue, and pink paths correspond to the following alignments:

```
h e e l ϵ l l o
```

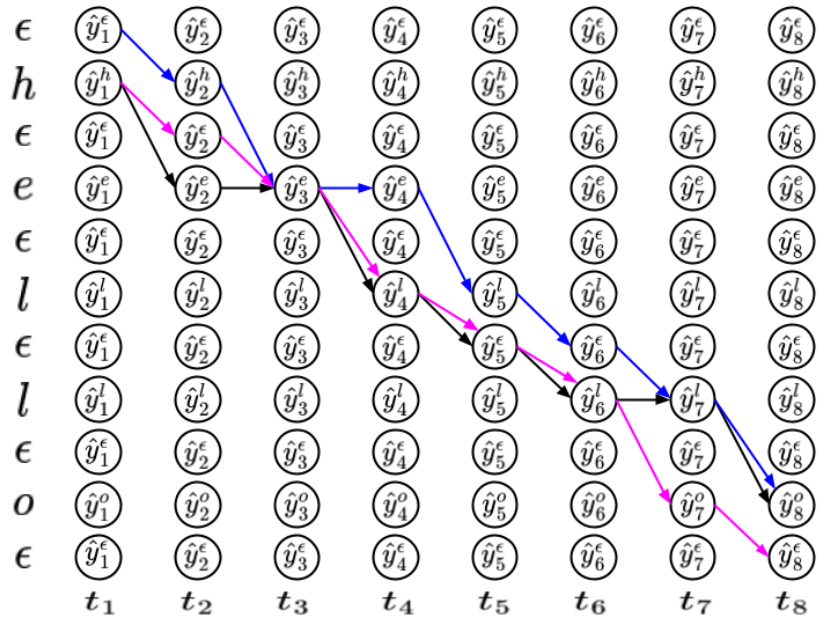

Figure 3.3: Some valid alignments for the word "hello" in 8 time steps.

```
ϵ h e e l ϵ l o
h ϵ e l ϵ l o ϵ
```

All these alignments result in the word "hello" after the collapsing function is applied on.

In contrast, some invalid alignments are:

```
h e l e ϵ l l o → helelo
ϵ h e l o ϵ o o → heloo
h h e e l l ϵ o → helo
```

Training

During training, the model should learn to output valid alignments that result in the correct transcription after the collapsing process is applied. To achieve this, we minimize the following negative log-likelihood marginalized over all possible valid alignments for the entire training dataset:

$$\text{loss} = -\sum_{(X,Y)\in D_{tr}} loss(X,Y) \tag{3.19}$$

Here, D_{tr} represent the training dataset and $loss(X,Y)$ is the loss for a given (X,Y) pair in D_{tr}:

$$loss(X,Y) = -\sum_{A \in A_{X,Y}} \sum_{t} \log p_t(a_t|X) \tag{3.20}$$

Here, $A = \{a_1, a_2, \ldots, a_T\}$ refers to any alignment, and $A_{X,Y}$ refers to all valid alignments for the given (X, Y) pair. The marginalization over all valid alignments ensures that all possible valid alignments are taken into account in the training process.

A naive approach to considering all valid alignments in the summation is to consider all alignments and perform the collapsing process, treating them as valid if they give the correct output. However, this is computationally expensive. For the word "hello" with 8 time steps, the computational cost would be $5^8 = 390,625$, where 5 refers to the characters `h`, `e`, `l`, `o`, and blank, and 8 refers to the number of time steps.

A better alternative is to use dynamic programming (DP). In DP, a complex problem is broken into more manageable subproblems. This process is iterated for each subproblem, solving them individually and storing their solutions for later use in addressing larger parent problems. This iterative process continues until the main problem is solved.

In our case, we use DP to compute the probability of reaching a state s at time t, denoted by $\alpha_t(s)$, from the probabilities of the previous time step $t-1$. Additionally, we compute the probability of obtaining the remaining sequence starting from state s at time t, denoted by $\beta_t(s)$. We compute these variables with the forward-backward algorithms shown in Algorithms 3 and 4.

Algorithm 3 CTC Forward Algorithm

Initialize: $\alpha_1(1) = \hat{y}_1^{S(1)}$, $\alpha_1(2) = \hat{y}_1^{S(2)}$, and $\alpha_1(r) = 0, r > 2$
for $t = 2$ to T **do**
  $\alpha_t(1) = \alpha_{t-1}(1)\hat{y}_t^{S(1)}$
  for $s = 2$ to S **do**
    if $s = \epsilon$ **or** $s = s-2$ **then**
      $\alpha_t(s) = (\alpha_{t-1}(s) + \alpha_{t-1}(s-1))\hat{y}_t^s$
    else
      $\alpha_t(s) = (\alpha_{t-1}(s) + \alpha_{t-1}(s-1) + \alpha_{t-2}(s-2))\hat{y}_t^s$
    end if
  end for
end for

Having computed these two variables, the probability of all paths that pass through state s at time t will be proportional to:

$$\alpha_t(s)\beta_t(s) \tag{3.21}$$

Equation 3.20, which contains a summation over all possible alignments, can therefore be

Algorithm 4 CTC Backward Algorithm

Initialize: $\beta_T(S) = 1$, $\beta_T(S-1) = 1$, and $\beta_T(r) = 0, r < S-1$
for $t = T-1$ to 1 **do**
  $\beta_t(S) = \beta_{t+1}(S)\hat{y}_{t+1}^{S(S)}$
  for $s = S$ to 1 **do**
    if $s = \epsilon$ **or** $s = s-2$ **then**
      $\beta_t(s) = \beta_{t+1}(s)\hat{y}_t^s + \beta_{t+1}(s+1)\hat{y}_t^{s+1}$
    else
      $\beta_t(s) = \beta_{t+1}(s)\hat{y}_t^s + \beta_{t+1}(s+1)\hat{y}_t^{s+1} + \beta_{t+2}(s+2)\hat{y}_t^{s+2}$
    end if
  end for
end for

rewritten in terms of these $\alpha_t(s)$ and $\beta_t(s)$ as:

$$loss(X, Y) \propto -\sum_t \sum_{s=1}^{S} \alpha_t(s)\beta_t(s)\log \hat{y}_t^s \tag{3.22}$$

Where the constant of proportionality is $\sum_r \alpha_t(r)\beta_t(r)$. This loss function is now a differentiable function of the RNN's output, $\hat{y}_t^s$, and back-propagation through the network can be performed to find the parameters of the model.

Inference

Once the model is trained, a simple greedy search can be performed for inference. In the greedy search, the most likely character is selected at each time step. The final transcription is obtained by applying the collapsing function, namely merging the consecutive repeated characters followed by removing blanks. While this works well in many cases, the greedy search is a suboptimal algorithm. Instead, a better alternative is to consider the sum of all alignments that result in the same output. This can be achieved by performing a beam search.

3.2.3 RNN-T

CTC assumes conditional independence of predicted characters from previously predicted characters, which reduces its accuracy compared to attention-based models. RNN-T was proposed by Graves (2012); Graves et al. (2013) to address this limitation of the CTC model. The architecture of RNN-T is shown in Figure 3.4. RNN-T comprises a CTC acoustic model, a predictor, and a joiner. In RNN-T, the CTC encoder processes the acoustic audio features and generates a hidden state for each time step. The predictor network serves as a language model, encoding the previous non-blank characters and outputting a hidden state. The joiner takes as input the two hidden states from the CTC encoder and the predictor and predicts the current character. Consequently, Equation 3.18 changes to:

$$p_t(a_t|h_t, \hat{y}_{<u_t}) \tag{3.23}$$

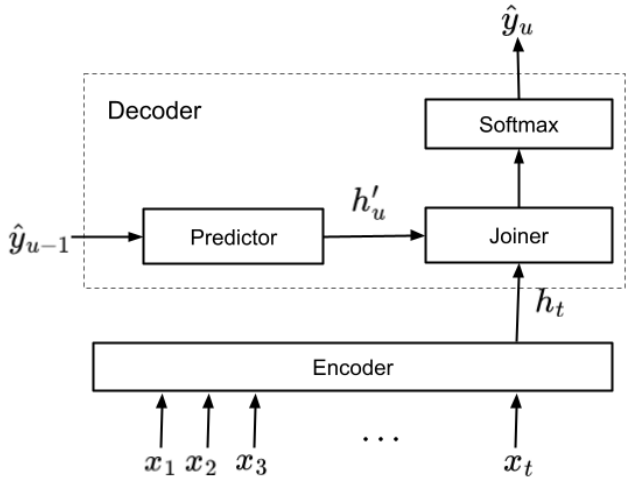

Figure 3.4: RNN-T model

3.2.4 Attention-based

Inspired by a similar task in NLP, specifically machine translation, where the input and output lengths differ, one can employ an encoder-decoder network to address the alignment issue.

Given the large input audio sequence, it is subsampled before processing:

$$x_1, x_2, \ldots, x_{T'} = \text{subsample}(f_1, f_2, \ldots, f_T) \tag{3.24}$$

Subsampling can be achieved, for example, by concatenating every few feature vectors into a larger vector or by utilizing a convolutional neural network. This process results in a shorter sequence of features ($T' < T$), facilitating faster computation.

The encoder takes the subsampled sequence as input and produces a context, c, containing information about the entire audio:

$$c = \text{encoder}(x_1, x_2, \ldots, x_{T'}) \tag{3.25}$$

The decoder utilizes the context and previously predicted words to generate the next word:

$$\hat{w}_t = \text{decoder}(\hat{w}_1, \hat{w}_2, \ldots, \hat{w}_{t-1}, c) \tag{3.26}$$

The visual representation of this encoder-decoder network is shown in Figure 3.5.

The underlying network in both the encoder and the decoder can be any RNN or transformer encoder/decoder blocks.

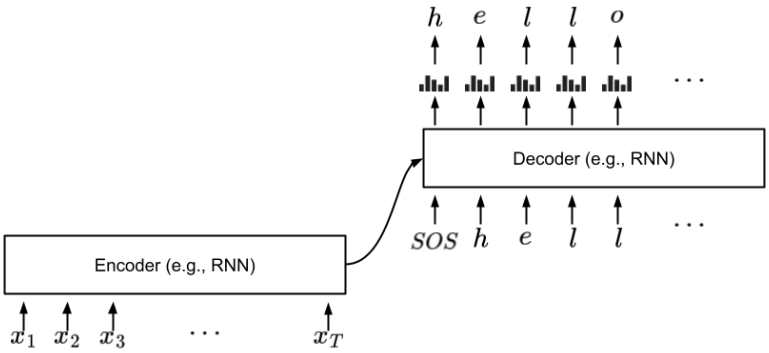

Figure 3.5: Visual representation of an encoder-decoder network for end-to-end ASR.

During training, negative log-likelihood loss is minimized on the training set to determine the model parameters:

$$\text{loss} = -\sum_t \log p(\hat{w}_1, \hat{w}_2, \ldots, \hat{w}_{t-1}, X) = -\sum_t \log \hat{w}_t \tag{3.27}$$

During inference, words are sampled from the model using one of several methods, such as greedy decoding, beam search, or other sampling techniques. In practice, beam search is often used, generating a list of the most probable possibilities (n-best list), which is then rescored using a pretrained large language model to select the best option.

This approach is known as an attention-based approach and was proposed in (Chorowski et al., 2014) and (Chan et al., 2016).

Proposed by Radford et al. (2023), whisper is a well-known attention-based model in automatic speech recognition (ASR). Whisper employs a transformer architecture, including an encoder and decoder, and is trained on 680,000 hours of audio to perform multiple tasks such as translation and transcription. These audio samples come in English and 96 other languages, with 125,000 hours of translations to English.

The log-magnitude mel spectrograms are calculated for every 25 ms window of the audio, overlapped by 10 ms. Then, two convolutional neural networks with a filter size of 3 and GELU activation are applied to them. Independent of the task type, the encoder takes the output of the convolutional neural networks and creates high-dimensional embedding representations, which are then fed to the decoder. As the Whisper model aims to perform multiple tasks, including transcription, translation, language identification, and more, on the same audio at once, this is formalized in the format of the decoder's input. The decoder's input contains a start of transcript token (`SOT`) that marks the beginning of the prediction. Transcript text

preceding the current audio segment can be added to the decoder's context with some probability, but it doesn't contribute to the final loss. The decoder then predicts the language of the input speech. If there is no speech in the audio, the model predicts a `nospeech` token. The next token specifies the task, which can be either transcription or translation. The decoder then predicts all the BPE tokens followed by an end of transcript token for the input audio. Timestamps for each token can also be added to perform alignment task. The visual representation of the Whisper model is depicted in Figure 3.6.

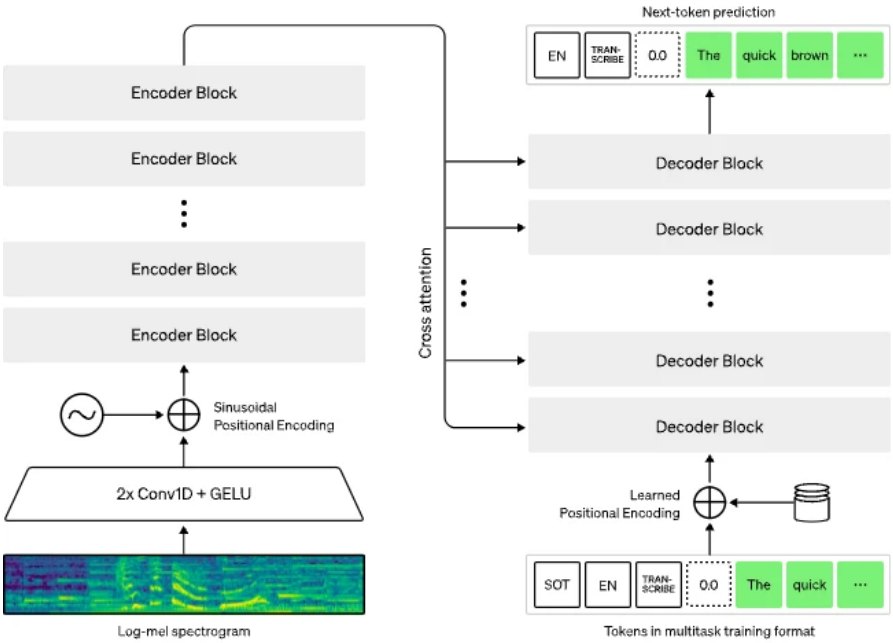

Figure 3.6: Whisper architecture (image from Ref. (Radford et al., 2023))

3.3 Evaluation

In this section, I describe how ASR transcriptions are evaluated. We typically evaluate the performance of ASR transcriptions using the following metrics:

- **Sentence error rate (SER)** measures whether the entire sentence is correct. SER compares the ASR transcription with the reference (the true transcription) and assign an error of 0 if they are identical, otherwise 100%. This measure is very strict and doesn't take into account the words that are transcribed correctly. For example, the SER for the following reference and transcription will be 100% even if a few words are transcribed correctly:
  - **Reference** I like to study in Idiap.

  - **Transcription** I like to see in the app.

- **Word error rate (WER)** is a better alternative and is proportional to the number of words that need to be edited to match the transcription with the reference. It is defined as:

$$WER = \frac{\texttt{\#edits}}{\texttt{\#words in reference}} \tag{3.28}$$

  Edits include insertion, deletion, and substitution of words. In the following example, it is straightforward to identify the edited words, namely substitutions and deletions shown in red and green respectively. The WER is 50% in this case.

  - **Reference:** I like to study in Idiap.
  - **Transcription:** I like to see in the app.

  However, it's not always straightforward to find optimal edits. Therefore, we use edit distance or Levenshtein distance (Levenshtein et al., 1966) to find the minimum number of edits required to convert one sequence of words to the other. Unlike SER, WER is less restrictive and takes correct words in the transcription into account. WER is a suitable measure for languages such as English, where words are the key building blocks.

- **Character error rate (CER)** is another measure of error in the transcription, quantifying how different the transcription is from the reference. CER is computed similarly to WER but with using characters. CER is usually used in languages such as Chinese and Japanese, in which characters are the main building blocks of the language.

In this thesis, we mainly use WER and CER to evaluate the performance of LLM correcting ASR transcriptions.

4 Related Works

With the advent of LLMs, a wide range of works have investigated how these could improve ASR performance. Generative instruction-tuned LLMs in particular offer new possibilities of combining ASR systems and LMs via prompting.

In hybrid speech recognition (Bourlard and Morgan, 1994), the decoder returns a list or lattice of hypotheses by combining probabilities from the acoustic model and a basic n-gram language model. These hypotheses can then be rescored with a more powerful neural LM (Deoras et al., 2011) or LLM (Udagawa et al., 2022). Recent neural end-to-end ASR approaches directly learn an LM, but can also be combined with a separately trained one by shallow fusion or other methods (Toshniwal et al., 2018).

In addition to traditional integration of LMs, one can focus specifically on identifying and correcting errors in the ASR output (Errattahi et al., 2018). Previous works have framed post-hoc ASR error correction as a spelling correction or a machine translation problem (Guo et al., 2019; Hrinchuk et al., 2020). The spelling correction model presented in the Ref. (Guo et al., 2019) is based on an encoder-decoder architecture using recurrent neural networks. This model is designed to correct errors in transcriptions produced by automatic speech recognition (ASR) systems by using a text-only dataset. The approach begins with the use of a highly competitive text-to-speech (TTS) system to convert a text corpus into audio speeches. These audio files are then processed by the ASR system, which converts them back into text. Each text output from the ASR system's n-best list serves as an input to the spelling correction network, while the original text serves as the target output. During training, the model learns to correct errors in the ASR transcriptions. Their approach showed 18.6% improvement on the Librispeech dataset. Ref. (Hrinchuk et al., 2020) used a similar approach to correct errors, but in contrast to (Guo et al., 2019), they employed a transformer model to translate the ASR output into grammatically and semantically correct text. The transformer model is trained on pairs of ASR outputs and their corresponding correct transcriptions. This training helps the correction model learn to map erroneous transcriptions to their correct forms.

In traditional n-best rescoring, only the best hypothesis is selected, although another one

could also be partially or fully correct. Chen et al. (Chen et al., 2023) therefore instruct LLMs to generate a new hypothesis based on all n-best options. They found that zero-shot prompting did not yield improvements on two datasets but adapting pre-trained LLMs with few-shot prompting, i.e. providing some example ASR outputs and corresponding corrections, and fine-tuning on a larger set of examples did.

Min and Wang (2023) explored the integration of LLMs in ASR systems to improve transcription accuracy. Their results show that directly applying the in-context learning capabilities of the LLMs for improving ASR transcriptions presents a significant challenge, and often leads to a higher WER. However, other works (Ma et al., 2023; Yang et al., 2023; Radhakrishnan et al., 2023) that explored the ability of LLMs to select, rescore and correct n-best list or ASR transcripts showed that zero and few-shot in-context learning can yield performance gains that are comparable to rescoring by domain-tuned LMs and can even achieve error rates below the n-best oracle level. Pu and Nguyen (2024) proposed to correct errors in the ASR output in two stages: 1) using a LM in the first stage to re-score an n-best list of ASR hypotheses and 2) using LLM through prompting in the second stage to correct errors in the less confident results from the first stage.

Other works (Li et al., 2023; He and Garner, 2023) have also applied LLMs to spoken language understanding (SLU) tasks, where the focus lies on identifying the correct intent from ASR transcripts – which may contain some errors – rather than correcting errors. This is crucial for applications like voice assistants. Ref (Li et al., 2023) investigates how well ChatGPT performs in SLU tasks: spoken document question answering, sentiment analysis, semantic similarity classification, natural language inference, and recognizing textual entailment. Their results show that ChatGPT is robust to some ASR errors in SLU tasks and can also correct some ASR errors.

Our work, on the contrary, focuses on giving more insights on how to effectively use LLMs to improve ASR performance. In a few-shot, in-context learning scenario, we evaluate the ability of LLMs to correct ASR transcripts. Similar to Pu and Nguyen (2024), we propose filtering ASR outputs based on confidence scores to prevent the LLM from introducing errors into transcripts that are likely already correct. We further analyze the errors introduced and the corrections made by LLMs. By doing so, our work seeks to shed light on the strengths and limitations of LLMs, when applied to ASR.

5 Experimental setup

In this section, we present our confidence-based filtering methods, which determine when to utilize LLMs to correct ASR transcriptions. A visual representation of our approach is shown in Figure 5.1. As depicted in the figure, the final transcription either undergoes LLM correction or is directly sent to the user, depending on the confidence scores of the transcript or its constituent tokens/words.

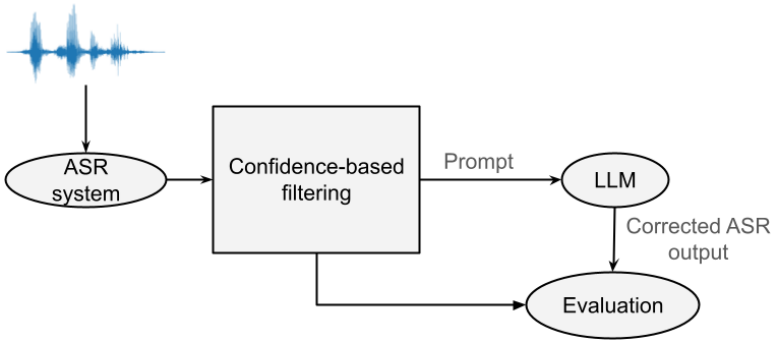

Figure 5.1: Visual representation of confidence-based filtering approach.

We begin by describing the ASR model used in our experiment, including an example of ASR output. Next, we describe the LLM used in our experiment. Finally, we elaborate on the proposed confidence-based filtering approach and the dataset used to evaluate it.

5.1 ASR system

We obtain initial ASR transcriptions from Whisper (Radford et al., 2023), described in the previous chapter. We run Whisper via the `whisper-timestamped` Python package (Louradour, 2023), which supports extracting token-level confidence scores. We compare the following models: *Tiny*, *Medium*, *Large V3*. Details of the specific Whisper models used in our experiments are given in Table 5.1. While English-only variants exist for the smaller models, we always use the multilingual one. Below we show the shortened version of tiny Whisper's output for a short audio segment from Librispeech `dev-clean` dataset. The output includes the start time (0.32 seconds) and end time (2.58 seconds) of the recognized speech segment. The transcribed text for this segment is "Their fingers see her me like fire". Additionally, the output provides a list of token IDs, [50364,6710,7350,536,720,385,411,2610,13,50500], representing the BPE tokens generated by the model. The temperature parameter used during decoding is 0.0, indicating deterministic decoding, and the overall confidence score for the transcription segment is 0.651, reflecting the model's certainty about its output. The output also includes a list of word-level details, with each word ("Their", "fingers", "see", "her", "me", "like", "fire") accompanied by its respective start and end times (0.32 to 0.62 seconds, 0.62 to 1.12 seconds, and 1.12 to 1.42 seconds, 1.42 to 1.52 seconds, 1.52 to 1.7 seconds, 1.7 to 2.06 seconds, 2.06 to 2.58 seconds) and individual confidence scores (0.488, 0.979, 0.508, 0.254, 0.851, 0.96, 0.99). These words are built by merging the PBE tokens. Finally, the detected language of the audio segment is specified as English ("en").

Table 5.1: Overview of Whisper ASR models.

Model	Tiny	Medium	Large-v3
Parameters	39M	769M	1550M

```
{
    "start": 0.32, "end":2.58, "temperature": 0.0, "confidence": 0.651,
    "text": "Their fingers see her me like fire.",
    "tokens": [50364,6710,7350,536,720,385,411,2610,13,50500],
    "words": [
        {
          "text": "Their",
          "start": 0.32, "end": 0.62,
          "confidence": 0.488
        },
        {
          "text": "fingers",
          "start": 0.62, "end": 1.12,
          "confidence": 0.979
        },
        {
          "text": "see",
          "start": 1.12, "end": 1.42,
          "confidence": 0.508
        },
        {
          "text": "her",
          "start": 1.42, "end": 1.52,
          "confidence": 0.254
        },
        {
          "text": "me",
          "start": 1.52, "end": 1.7,
          "confidence": 0.851
        },
        {
          "text": "like",
          "start": 1.7, "end": 2.06,
          "confidence": 0.96
        },
        {
          "text": "fire.",
          "start": 2.06, "end": 2.58,
          "confidence": 0.99
        }
    ],
    "language": "en"
}
```


Listing 5.1: Shortened output of the tiny Whisper model for a segment of Librispeech `dev-clean` audio speeches.

5.2 Large language model

In Chapter 2, we described LLM with two particular examples, namely BERT and GPT. We use the following OpenAI ChatGPT models as LLMs for error correction in the ASR transcriptions: `gpt-3.5-turbo-1106`, `gpt-3.5-turbo-0125`, and `gpt-4-0125-preview`. The GPT-3.5 turbo models have a context window of 16,385 tokens and can understand and generate natural language while GPT-4, with its 128,000 token context window, is a large multimodal model (handling both text and image inputs with text outputs) and tackles more complex problems more accurately than its predecessors (Achiam et al., 2023). We call ChatGPT via its Python API and do not modify any default parameters. In addition, we explore the open-source local LLM Llama 3, which has a context window of 8,912 tokens.

5.3 Confidence-based filtering

Passing every ASR output to the LLM for correction risks introducing errors into correctly transcribed sentences. To mitigate this, we compare different filtering methods based on confidence scores returned by the Whisper ASR model.

Whisper internally uses tokens that are obtained with byte-pair encoding (Sennrich et al., 2016). Each output token is associated with a confidence score based on its probability [1]. The `whisper-timestamped` Python package (Louradour, 2023) computes word-level confidence scores by averaging the confidence scores of all tokens that form a word. Similarly, sentence-level confidence scores are computed by averaging the confidence scores of all tokens in the sentence, excluding punctuation tokens in both cases.

To decide which ASR outputs should be passed to the LLM for correction, we filter based on either the sentence-level confidence score or the lowest word-level confidence score within the sentence. We refer to these two methods as *sentence-level* and *lowest-word* confidence filtering. Sentences with a confidence score above a chosen threshold retain their original ASR outputs.

As a third option, we prompt the LLM to correct only *specific words* in the ASR transcription that fall below a certain confidence threshold. If no words within the transcription fall below the confidence threshold, the original ASR transcription is retained.

For example, considering the Whisper ASR transcription presented in Listing 5.1 and a threshold of 0.7, in the sentence-confidence approach, this transcription is passed to the LLM for correction because the confidence score of the transcription (0.651) is lower than the threshold. For the same threshold in the lowest-word confidence method, it is also passed to the LLM for correction since the lowest confidence score of the words ("her", 0.254) is lower than the

[1] Confidence score for each token is the softmax of the logits, with logits being the raw scores for each possible output token produced in the Whisper model. The result of the softmax function is the probability distribution over all possible tokens.

threshold. In the third filtering method, correction of specific words, we prompt the LLM to correct only among the words below this threshold, namely ("Their", 0.488), ("see",0.508), and ("her",0.254) in the transcription.

The reference transcription for this case is:

```
"THEIR FINGERS SEAR ME LIKE FIRE"
```

In Section 6.4, we will observe that the LLM has successfully corrected all the errors in this ASR transcription, meaning that the words "see her" that sound phonetically similar to "sear" is identified and corrected by the LLM.

By implementing these filtering methods, we aim to reduce the risk of introducing errors into already accurate transcriptions while ensuring that the most uncertain parts of the transcription are corrected by the LLM. We will compare the differences in WER for each filtering method and threshold to determine the most effective approach.

5.4 Dataset

We evaluate our proposed approach on the English LibriSpeech corpus (Panayotov et al., 2015), which consists of audiobook recordings. This corpus is widely used in the ASR community for benchmarking due to its high-quality and well-annotated recordings. For our experiments, we use the `dev-clean` and `dev-other` subsets for initial experiments and hyperparameter tuning. These subsets are relatively balanced in terms of speaker demographics and recording conditions, making them suitable for fine-tuning models and evaluating initial performance.

After tuning, we report final results on the `test-clean` and `test-other` evaluation sets. Each of these subsets contains approximately 2500 to 3000 utterances. The `clean` subsets feature speech from speakers with clear and distinct pronunciation, whereas the `other` subsets include speech from speakers with more challenging accents and less distinct pronunciation. This variation leads to higher WERs for the `other` portions, providing a robust test of our approach's effectiveness under more difficult conditions.

It is important to note that while these LibriSpeech subsets are not part of the Whisper training data, we cannot exclude the possibility that ChatGPT was trained on them due to the proprietary nature of the model. This consideration is relevant for understanding the potential limitations and biases in the evaluation of our approach.

6 Results

In this section, we present our results in terms of WER and CER on identifying a suitable prompt, comparing different ASR models, and filtering based on confidence scores. We also discuss and analyze the types of errors made by the LLM.

6.1 Prompt selection

We first describe our process of selecting a suitable prompt and analyze which elements of the prompt are important for ASR performance. LLMs perform best when the prompt contains a clear description of the task. For this reason, we provided information about the task, the format of the input and the expected output, and provided two examples in the prompt[1].

In the prompt, we clearly explain the task of correcting ASR errors to the LLM. We further describe the format of the input and expected response and instruct the LLM not to provide any explanatory or additional text besides the corrected transcription. We then provide one or two example input-output pairs for a few-shot learning scenario (Chen et al., 2023).

We show the base prompt for our experiments with a basic description of the task in Table 6.1. In other prompts, we explicitly instruct the LLM to make grammar corrections and to make changes that closely match the input transcription acoustically or phonetically. Results for these different prompts in Table 6.2 show that in particular providing more than one example and instructing to make phonetically plausible corrections improve the ASR performance. For all following experiments, we therefore use prompt 4.

6.2 Influence of ASR performance

We study the influence of ASR performance on the LLM corrections by comparing Whisper models of different strength. As shown previously (Udagawa et al., 2022), less competitive ASR

[1] For simplicity, we pass the ASR transcripts to the LLM in json format.

Table 6.1: LLM prompts used in this work. For certain experiments, the red and/or blue parts are added. Italic text shows examples provided after the system prompt, with the intended response in bold.

System:
You are a helpful assistant that corrects ASR errors.
You will be presented with an ASR transcription in json format
with key: text and your task is to correct any errors in it.
If you come across errors in ASR transcription, make corrections that
closely match the original transcription acoustically or phonetically
If you encounter grammatical errors, provide a corrected version
adhering to proper grammar.
Provide the most probable corrected transcription in string format.
Do not change the case, for example, lower case or upper case,
in the transcription.
Do not output any additional text that is not the corrected transcription.
Do not write any explanatory text that is not the corrected transcription.

User:
`text`: *Why not allow your silver tuff to luxuriate in a natural manner?*

Assistant
why not allow your silver tufts to luxuriate in a natural manner?

User:
`text`: *Meanwhile, how fair did it with the flowers?*

Assistant:
Meanwhile, how fared did it with the flowers?

User:
`text`: *ASR transcription*

Table 6.2: WER (%) on LibriSpeech `dev-clean` of the original Whisper `tiny` output and corrections with `gpt-3.5-turbo-1106` for different prompts.

Prompt	**WER**
Original ASR output	8.51
1: Base prompt (with one example)	7.49
2: Base prompt (with two examples)	6.76
3: 2 + do correct grammar mistakes	6.90
4: 2 + ensure corrections are phonetically similar	**6.65**
5: 2 + 3 + 4	6.79

models — Whisper *Tiny* and *Medium* in our case — leave more room for improvement.

We summarize the results in terms of WER and CER for the original ASR and the LLM-corrected transcripts (relative improvement in parentheses) on both development sets in Table 6.3. While we observe improvements in WER with LLM correction in most cases, the relative improvements in `dev-other` are smaller compared to the ones in `dev-clean`. This suggests that correcting errors in more difficult speech data also presents a greater challenge for the LLM model.

Table 6.3: WER and CER of the original ASR output and LLM corrections with `gpt-3.5-turbo-1106` for different Whisper models (relative change in parentheses).

Whisper model	**WER**			**CER**		
	ASR	+LLM (rel. (%))		ASR	+LLM (rel. (%))	
dev-clean						
Tiny	8.51	6.65	(-21.9)	3.49	3.08	(-11.7)
Medium	4.12	3.50	(-15.0)	1.79	1.42	(-20.7)
Large V3	3.11	3.34	(+7.4)	1.16	1.21	(+4.3)
dev-other						
Tiny	17.03	14.87	(-12.7)	8.16	7.71	(-5.5)
Medium	6.54	6.19	(-5.4)	2.96	2.90	(-2.0)
Large V3	4.62	4.59	(-0.6)	1.83	1.89	(+3.3)

6.3 Confidence-based filtering

6.3.1 Sentence and Lowest-word confidence

In Figure 6.1 (left) we show the WERs for various *sentence-level* confidence thresholds for all Whisper models on the LibriSpeech `dev-clean` subset. Transcriptions with a confidence score higher than the threshold are not passed to the LLM for correction. The Figure shows that the optimal value for the threshold is 0.95 for *Tiny* and *Medium* models while the *Large*

model is not sensitive to the threshold.

Figure 6.1 (right) shows the effect of varying the *lowest-word* confidence thresholds. A value of 0.7 provides a good trade-off of stable ASR performance and reducing the number of utterances that needs to be corrected by the LLM. We observed similar patterns for both methods on `dev-other` (Figure 6.2).

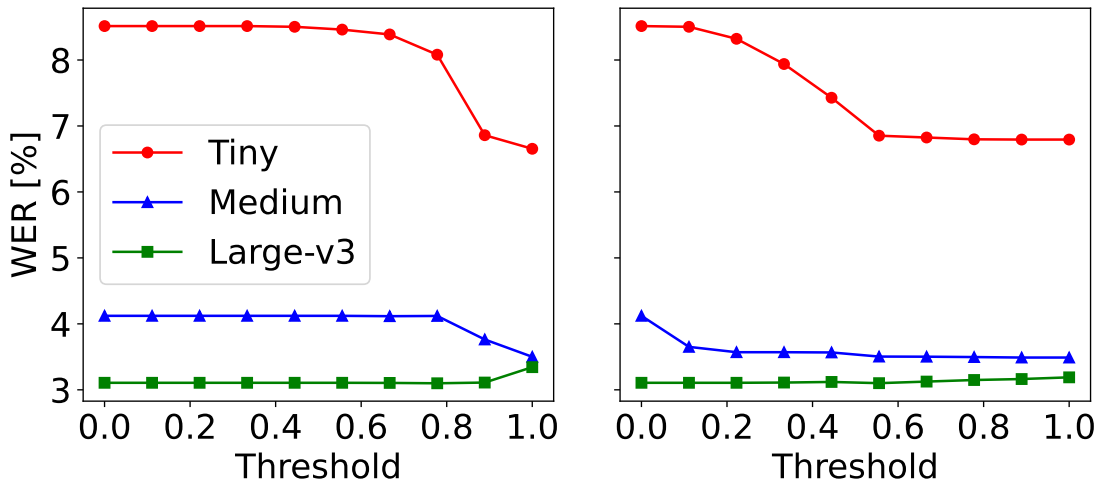

Figure 6.1: WER for various sentence-level (left) and lowest-word (right) confidence thresholds for Tiny, Medium, and Large V3 Whisper models applied on `dev-clean` dataset with `gpt-3.5-turbo-1106`.

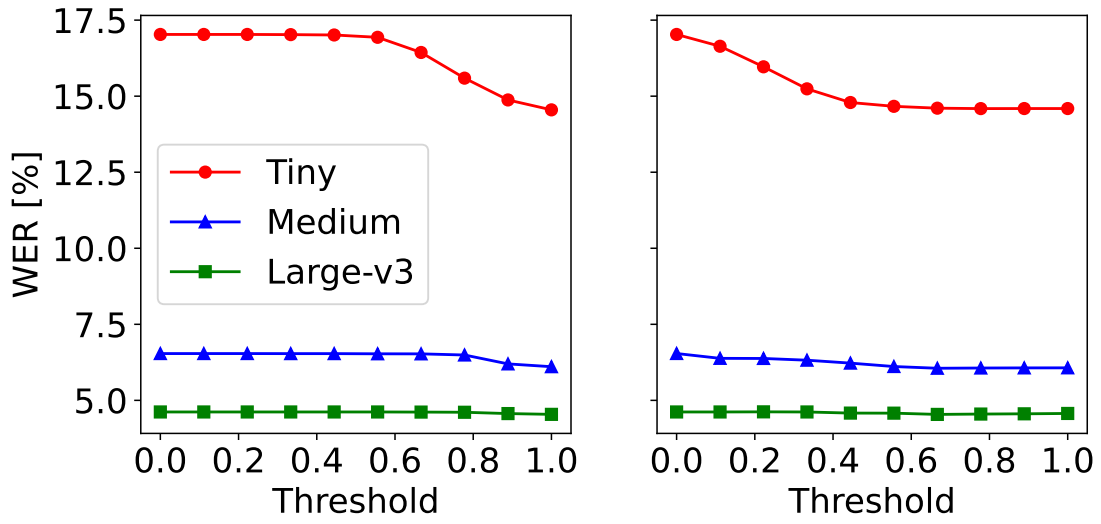

Figure 6.2: WER for various sentence-level (left) and lowest-word (right) confidence thresholds for Tiny, Medium, and Large V3 Whisper models applied on `dev-other` dataset with `gpt-3.5-turbo-0125`.

Table 6.4: LLM prompt used in correction of specific words experiment. The rest of the system message is similar to Table 6.1 and is not shown here.

System:

⋮

You will be presented with an ASR transcription and
a list of words in the transcription with low confidence scores.
The input will be formatted as json with keys: text and low_confidence_words.
Your task is to correct any errors in the transcription.
Make sure you correct only words from within the low_confidence_words list.

⋮

User:
`text`: *Why not allow your silver tuff to luxuriate in a natural manner?*
`low_confidence_words`: *[tuff]*

Assistant
why not allow your silver tufts to luxuriate in a natural manner?

User:
`text`: *Meanwhile, how fair did it with the flowers?*
`low_confidence_words`: *[fared]*

Assistant:
Meanwhile, how fared did it with the flowers?

User:
`text`:*ASR transcription*
`low_confidence_words`: *list of words*

6.3.2 Correction of specific words

In this experiment, we pass both the ASR transcriptions and a list of words with confidence scores below a predefined threshold to the LLM. A threshold of 1 essentially grants ChatGPT the freedom to correct any word, similar to previous experiments. We further adjusted our prompt, specifically instructing to only address potential errors within the provided list of low-confidence words. The adjusted prompt is shown in Table 6.4:

Figure 6.3 presents the WER results for various confidence thresholds.

As the figure demonstrates, thresholds close to 0 results in a WER near the original WER (without ChatGPT correction) but a threshold of 1 gives a higher value for WER than the previous experiments where there was no low-confidence word list restriction. The WER reaches its lowest value of 7.55 at a threshold of 0.5, not matching the performance of the

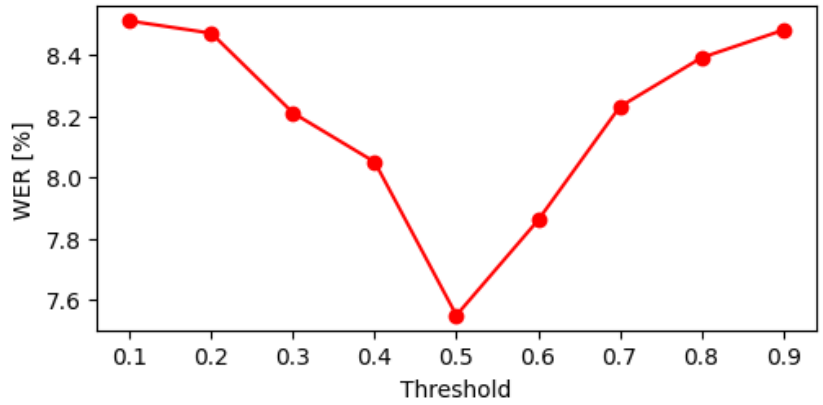

Figure 6.3: WER for various thresholds for specific low-confidence words with Tiny Whisper model applied on `dev-clean` dataset with `gpt-3.5-turbo-1106`.

Table 6.5: WER on the LibriSpeech test sets of the original ASR output and LLM corrections with `gpt-3.5-turbo-0125`/`gpt-4-0125-preview` for different Whisper models, comparing lowest-word and sentence-level confidence. For each case, we also show what percentage of utterances was passed to the LLM after confidence-thresholding.

Whisper	**test-clean**				**test-other**			
model	ASR	GPT-3.5	GPT-4	% corrected	ASR	GPT-3.5	GPT-4	% corrected
Lowest-word confidence (threshold: 0.7)								
Tiny	8.13	6.55	5.65	86.6%	17.45	15.49	**13.65**	94.0%
Medium	4.27	**3.42**	3.54	64.3%	8.20	6.67	6.97	72.8%
Large V3	**2.78**	2.86	3.21	53.0%	**4.82**	4.91	4.93	60.4%
Sentence-level confidence (threshold: 0.95)								
Tiny	8.13	6.56	**5.63**	94.5%	17.45	15.51	13.67	98.0%
Medium	4.27	3.71	3.56	67.1%	8.20	6.77	**6.62**	79.8%
Large V3	**2.78**	2.83	3.13	48.4%	**4.82**	4.93	4.94	60.8%

sentence-level and lowest-word confidence approaches.

6.3.3 Test set performance

Finally, we also present results for our selected best prompt and confidence thresholds on the LibriSpeech `test-clean` and `test-other` evaluation sets in Table 6.5. The best result for each dataset and Whisper model is highlighted in bold. Our findings indicate that, despite its higher number of parameters, GPT-4 only outperforms GPT-3.5[2] for Whisper *Tiny*, but does not result in additional improvements in WER for the transcriptions of the *Medium* and *Large* models.

[2]Here we used `gpt-3.5-turbo-0125` which we found to perform similar to `gpt-3.5-turbo-1106` on the development sets, but is faster and more robust to API errors.

6.4 Error analysis

In this section, we showcase examples of Whisper *Tiny* outputs on the development sets in which the LLM has corrected errors in the transcriptions, has failed to correct or even introduced new errors into the transcription.

(1) *REF: their fingers *** sear me like fire*
ASR: their fingers see her me like fire
LLM: their fingers *** sear me like fire

(2) *REF: damn your impertinence sir burst out burgess*
ASR: dam your impertinent sur burst out burges
LLM: damn your impertinent sir burst out burgess

(3) *REF: *** *** fedosya 's face made her anxious*
ASR: the dose used to face nature *** anxious
LLM: the dose used to face nature *** anxiously

Example 1 and 2 show cases where the LLM has corrected all or most of the errors within the ASR transcriptions of Whisper *Tiny*. Here, `REF`, `ASR`, and `LLM` denotes reference, ASR, and LLM-corrected transcriptions respectively. Example 3 is a typical case of where the LLM struggles to correct the transcript because it already contains too many errors and, for example, reconstructing proper nouns without acoustic context is challenging. Furthermore, Table 6.6 breaks down for how many utterances the LLM improved, worsened, or did not change the ASR performance.

Table 6.6: Percentage of utterances where LLM improved, worsened, and didn't change WER of Whisper Tiny outputs.

Dataset	Improved	Worsened	No Change
`dev-clean`	26.38	4.85	68.78
`dev-other`	29.96	6.11	63.93

We also note that LLM corrections can sometimes decrease WER while increasing CER. This occurs because any number of character changes within a word only affects the WER by one unit ($\frac{1}{N}$ with N being the number of words in the reference transcription). However, the same changes can have a greater impact on CER.

(4) *REF: pour mayonnaise over all chill and serve*
ASR: parme a nays overall chill and serve
LLM: parmesan *** over all chill and serve

Example 4 demonstrates this effect. The LLM reduces WER from 57.14% to 28.57% in this example, while CER increases from 25.00% to 27.50%.

Additionally, Table 6.7 presents the detailed modifications performed by each language model (LM) and prompt without filtering using the `dev-clean` dataset. These modifications, including the number of insertions, deletions, and substitutions required to align the transcription with the reference, are depicted relative to the original transcription. For instance, the table demonstrates that the corrected transcription by `gpt-4-0125-preview` with prompt 2 for the tiny Whisper model requires a reduced number of insertions, deletions, and substitutions compared to the original transcriptions to match the reference.

Similarly, Table 6.8 illustrates a similar analysis using the `dev-other` dataset.

Table 6.7: Metrics of `dev-clean` Librispeech dataset for various Whisper models (Tiny, Medium, and Large-v3) and experiments (prompt, LLM-model, etc.) without confidence-based filtering.

Whisper model	Prompt/LLM	Metrics			
		Substitutions	Insertion	Deletions	WER
Tiny	Original	3372	716	544	8.51
	Prompt-2 (gpt-3.5-turbo-1106)	-721	-294	58	6.76
	prompt-2 (gpt-4-0125-preview)	-1048	-351	3	5.95
	Prompt-4 (gpt-3.5-turbo-1106)	-782	-305	75	6.65
Medium	Original	1450	396	396	4.12
	Prompt-4 (gpt-3.5-turbo-0125)	-71	-279	17	3.51
	Prompt-4 (gpt-3.5-turbo-1106)	-69	-283	14	3.50
	Prompt-4 (gpt-4-0125-preview)	-70	-245	48	3.63
	Prompt-2 (gpt-3.5-turbo-1106)	-22	-288	15	3.58
	Prompt-2 (gpt-4-0125-preview)	48	-191	71	3.99
Large-v3	Original	1137	60	493	3.11
	Prompt-4 (gpt-3.5-turbo-0125)	-1	11	-5	3.12
	Prompt-4 (gpt-3.5-turbo-1106)	8	21	100	3.34
	Prompt-4 (gpt-4-1106-preview)	9	94	50	3.76
	Prompt-2 (gpt-3.5-turbo-1106)	62	15	16	3.28

Table 6.8: Metrics of `dev-other` Librispeech dataset for various Whisper models (Tiny, Medium, and Large-v3) and experiments (prompt, LLM-model, etc.) without confidence-based filtering.

Whisper model	Prompt/LLM	Metrics			
		Substitutions	Insertions	Deletions	WER
Tiny	Original	6537	1325	824	17.03
	Prompt-4 (gpt-3.5-turbo-1106)	-922	-256	68	14.87
	Prompt-4 (gpt-3.5-turbo-0125)	-1112	-277	45	14.41
	Prompt-4 (gpt-4-0125-preview)	-1684	-185	-22	13.34
	Prompt-2 (gpt-3.5-turbo-0125)	-881	-254	83	14.98
Medium	Original	2450	384	498	6.54
	Prompt-4 (gpt-3.5-turbo-1106)	-137	-55	16	6.19
	Prompt-4 (gpt-3.5-turbo-0125)	-160	-73	4	6.09
	Prompt-4 (gpt-4-0125-preview)	-309	-86	-3	5.76
	Prompt-2 (gpt-3.5-turbo-0125)	-101	-62	24	6.27
Large-v3	Original	1771	162	420	4.62
	Prompt-4 (gpt-3.5-turbo-0125)	-53	14	14	4.57
	Prompt-4 (gpt-3.5-turbo-1106)	-45	18	17	4.60
	Prompt-4 (gpt-4-0125-preview)	-125	39	6	4.46
	Prompt-2 (gpt-3.5-turbo-1106)	-2	28	8	4.69

6.4.1 Parts of speech analysis

In the previous sections, we used WER to assess the overall impact of LLM on transcript quality. Now, we delve deeper into how LLM corrects transcripts and analyze these modifications in a more detailed manner.

An ASR transcript can either be entirely correct or contain errors. An LLM is expected to leave correctly transcribed words untouched while identifying errors and correcting them through deletions, substitutions, or insertions. We categorize LLM modifications into four distinct groups:

- **Improved**:LLM corrects an error in the transcript. This may involve deleting or substituting an erroneous word or adding a correct word.
- **IntroducedError**: LLM introduces a new error in the transcript. This could involve deleting/adding a correct/incorrect word, or substituting a correct word with an incorrect one.
- **LeftCorrect**: LLM leaves an incorrectly transcribed word unchanged.
- **LeftIncorrect**: LLM leaves an incorrectly transcribed word unchanged.

For example 1 shown previously, these modifications are:

REF:	their	fingers	***	sear	me	like	fire
ASR:	their	fingers	see	her	me	like	fire
LLM:	their	fingers	***	sear	me	like	fire
operation	-	-	D	S	-	-	-
modification-type	Left-Correct	Left-Correct	-	Improved	Left-Correct	Left-Correct	Left-Correct

The example above demonstrates that the words "see" and "her" in the transcript were deleted and substituted, respectively, improving the transcript. Other words in the transcript remain correct (unchanged).

Additionally, we analyzed the POS of words for each type of modification (Improved, IntroducedError, etc.). We used the SpaCy package for calculating parts of speech (Honnibal et al., 2020). The results for the `dev-other` dataset transcribed by the Tiny model are shown in Table 6.9. The table shows modification types for each POS in %, where the sum of all modification types for each POS equals 100%. For instance, nouns within all transcripts, 5.47% were corrected by the LLM, 0.93% introduced errors, 76.89% remained correct, and 16.71% remained incorrect.

The table reveals that the majority of words in the ASR transcripts, across all POSs, are already correct, with only a small fraction being incorrect (with notable exception of proper nouns, where only about 49% are accurate). This can be observed from large contributions in the LeftCorrect column.

The table also shows that the LLM has mainly left most errors unchanged and was unable to identify and correct them. This can be observed by comparing the columns Improved and LeftIncorrect. However, for the modifications it did make, it consistently corrected errors more often than it introduced them across all POSs (Improved column is larger than IntroducedError column for all POSs). Specifically for proper nouns, where there are many more errors in the transcripts, the LLM corrected only 6.5% of errors while introducing errors in 0.5%, leaving the remainder 44.12% unchanged.

Similar to Table 6.9, Table 6.10 shows LLM modifications categorized by POS. In contrast, it shows the proportion of each POS for each modification type (in %), where the sum across all POSs is 100%. The Percentage column indicates the proportion of each POS within all transcripts. For instance, it can be observed from the table that nouns, pronouns, and verbs are the most prevalent POSs in the transcripts, collectively comprising approximately 46% (18% nouns, 14% pronouns, and 14% verbs) of POSs in the transcripts.

It is noticeable that nouns which constitute 18% of all POSs in the transcripts, they account for 30% of all the improvements. Similarly, verbs, which constitute 14% of all POSs, receive about 17% of all the improvements. In contrast, despite pronouns constituting 14% of all POSs in the transcripts, only approximately 7% of them are improved.

Table 6.9: LLM modifications by POS in the `dev-other` dataset transcribed by the Tiny Whisper model, presented as percentages. The totals across modification types (Improved, IntroducedError, LeftCorrect, and LeftIncorrect) sum up to 100%.

POS	Improved	IntroducedError	LeftCorrect	LeftIncorrect
adjective	3.36	1.04	83.77	11.82
adposition	2.77	0.86	88.00	8.37
adverb	2.00	0.85	87.94	9.20
auxiliary	1.92	0.56	87.74	9.78
coordinating conjunction	2.05	0.18	88.80	8.97
determiner	2.62	0.68	86.18	10.51
interjection	0.44	0.00	76.75	22.81
noun	5.47	0.93	76.89	16.71
numeral	4.99	0.88	67.45	26.69
particle	2.04	0.61	90.02	7.33
pronoun	1.56	0.51	89.90	8.04
proper noun	6.50	0.54	48.84	44.12
subordinating conjunction	1.43	0.75	90.16	7.67
verb	3.98	0.96	81.24	13.82

Table 6.10: Same as 6.9 but the totals across POSs sum up to 100%. Percentage column shows the total percentage of each POS within all transcripts.

POS	Improved	IntroducedError	LeftCorrect	LeftIncorrect	Percentage
adjective	6.97	9.09	6.62	6.40	6.62
adposition	9.74	12.63	11.78	7.67	11.21
adverb	3.67	6.57	6.13	4.39	5.84
auxiliary	4.15	5.05	7.21	5.50	6.89
coordinating conjunction	2.76	1.01	4.56	3.15	4.30
determiner	7.87	8.59	9.85	8.23	9.58
interjection	0.06	0.00	0.40	0.82	0.44
noun	31.37	22.47	16.81	25.01	18.32
numeral	1.02	0.76	0.53	1.43	0.65
particle	1.80	2.27	3.04	1.69	2.83
pronoun	6.85	9.34	15.06	9.22	14.04
proper noun	5.05	1.77	1.44	8.94	2.48
subordinating conjunction	1.26	2.78	3.04	1.77	2.83
verb	17.43	17.68	13.54	15.78	13.97

6.5 Local LLMs

We have shown that ChatGPT as the LLM can correct ASR transcriptions. However, due to noticeable costs and transparency issues, we are interested in exploring more sustainable alternatives, such as open-source LLMs. In this section, we explore the open-source Meta Llama LLM in the correction of ASR transcriptions. Llama is a family of LLMs based on a transformer architecture, with models available in 8 billion (8B) and 70 billion (70B) parameters.

It is pretrained on over 15 trillion tokens from publicly available sources and fine-tuned on over 10 million human-annotated examples for tasks such as text generation and code generation. Llama is also instruction-tuned using supervised fine-tuning (SFT) and reinforcement learning with human feedback (RLHF) (Touvron et al., 2023).

We employed Llama 3 to correct ASR transcriptions from the `dev-other` dataset, which were originally transcribed by the Tiny Whisper model. The results for WER and CER, presented in Table 6.11, indicate that the corrected transcriptions exhibit lower accuracy compared to both the original transcriptions and those corrected by ChatGPT.

Table 6.11: WER and CER of the original ASR output and LLM corrections with Llama for Tiny Whisper model and `dev-other` dataset (relative change in parentheses).

Whisper model	**WER**			**CER**		
	ASR	+Llama	(rel. (%))	ASR	+Llama	(rel. (%))
Tiny	17.03	20.1	(+18.0)	8.16	11.33	(+38.84)

This suggests that currently, Llama introduces more errors than it corrects. Additionally, Llama's modifications to the ASR transcription include 7274 substitutions, 1882 insertions, and 1077 deletions. Compared to the GPT models in Table 6.8, Llama performs more changes (substitutions, insertions, and deletions) which introduces more errors into the transcription rather than correcting the existing ones.

We used an identical prompt for the Llama model as we did for ChatGPT. However, Llama may require a different prompt than ChatGPT. In our future work, we plan to explore Llama further with various prompts. Additionally, we will investigate other open-source LLMs across various datasets, confidence-based filtering methods, and ASR models.

6.6 Discussion

In this chapter, we have shown that the proposed confidence-based filtering methods can improve low-accuracy ASR transcriptions of the Librispeech dataset transcribed by the less competitive Whisper ASR models: Tiny and Medium. However, there are several challenges and limitations involved:

- **Dataset Limitation**: We only evaluated our method on the Librispeech dataset, which consists of audiobook recordings. This dataset is less challenging than real-life conversational datasets. Therefore, it remains a question how well our method performs on more challenging datasets such as AMI meeting corpus. Testing on these diverse datasets is necessary to confirm the generalizability of our approach.
- **ASR Model Limitation**: Our study focused only on transcriptions generated by the Whisper ASR model. Studying the effect of the ASR model in our approach is important

to ensure that the proposed methods are not model-specific and can be effectively applied across different ASR systems. This would confirm the general effectiveness and robustness of our approach.

- **LLM Limitation**: In this study, we used ChatGPT as the LLM to correct ASR transcriptions via an API. However, due to high costs, transparency issues, and potential unavailability, ChatGPT may not be practical for real-time applications. To explore more sustainable options, we tested Meta Llama that can be run locally on `dev-other` dataset transcribed by the whisper Tiny model. The results showed that the accuracy of corrected transcripts significantly degraded compared to ChatGPT. This indicates that local LLMs need substantial improvements before they can effectively be used to correct ASR transcripts.

7 Conclusions

In this work, we investigated LLMs for ASR error correction. We studied the impact of various prompts on the WER of the corrected ASR transcriptions. Our results indicate that prompts explicitly asking the LLM to ensure phonetic similarity in corrections reduce the WER the most, while prompts focusing on correcting grammar mistakes while still reducing WER achieve the least reduction in WER.

Viewing ASR systems as noisy listeners, inspired by human speech perception, we proposed three methods for filtering ASR outputs based on confidence measures. These methods allow the LLM to focus only on less accurate transcripts. In the first method, sentence-level confidence, we passed transcriptions to the LLM for correction only if the sentence confidence was below a certain threshold. In the second method, the lowest-word confidence, we used the LLM to correct transcriptions where the lowest confidence score among the words was below a certain threshold. In the last method, correction of specific words, we asked the LLM to correct only words with confidence scores below a specific threshold. We carefully calculated these thresholds using the training data. Our results confirm that these filtering methods, particularly the lowest-word confidence and sentence confidence, significantly boost ASR performance for less competitive ASR models because these models have more room for improvement.

We plan to investigate additional confidence estimation methods and other ASR systems than Whisper in future work. LLM outputs could also be rescored again with the acoustic model to validate them. We will further consider other long-form datasets where utterances are not evaluated one-by-one and LLMs are expected to provide more benefits because of their long context windows. Finally, studies also need to be conducted on other languages, where LLMs might not perform as well as on English.

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., an d Derek Murray, S. M., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Achiam, J. et al. (2023). GPT-4 Technical Report. https://api.semanticscholar.org/CorpusID:257532815.

Baum, L. E. et al. (1972). An inequality and associated maximization technique in statistical estimation for probabilistic functions of markov processes. *Inequalities*, 3(1):1–8.

Bojanowski, P., Grave, E., Joulin, A., and Mikolov, T. (2017). Enriching word vectors with subword information. *Transactions of the association for computational linguistics*, 5:135–146.

Bourlard, H. and Morgan, N. (1994). *Connectionist Speech Recognition: A Hybrid Approach*. Kluwer Academic Publishers.

Chan, W., Jaitly, N., Le, Q., and Vinyals, O. (2016). Listen, attend and spell: A neural network for large vocabulary conversational speech recognition. In *2016 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pages 4960–4964. IEEE.

Chen, C., Hu, Y., Yang, C.-H. H., Siniscalchi, S. M., Chen, P.-Y., and Chng, E.-S. (2023). HyPoradise: An Open Baseline for Generative Speech Recognition with Large Language Models. In *Proc. NeurIPS*.

Cho, K., van Merrienboer, B., Bahdanau, D., and Bengio, Y. (2014). On the properties of neural machine translation: Encoder–decoder approaches. In *SSST@EMNLP*.

Chorowski, J., Bahdanau, D., Cho, K., and Bengio, Y. (2014). End-to-end continuous speech recognition using attention-based recurrent nn: First results. *arXiv preprint arXiv:1412.1602*.

Davis, S. and Mermelstein, P. (1980). Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. *IEEE transactions on acoustics, speech, and signal processing*, 28(4):357–366.

Deoras, A., Mikolov, T., and Church, K. (2011). A Fast Re-scoring Strategy to Capture Long-Distance Dependencies. In *Proc. EMNLP*, pages 1116–1127.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. In Burstein, J., Doran, C., and Solorio, T., editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Errattahi, R., El Hannani, A., and Ouahmane, H. (2018). Automatic Speech Recognition Errors Detection and Correction: A Review. *Procedia Computer Science*, 128:32–37.

Forney, G. D. (1973). The viterbi algorithm. *Proceedings of the IEEE*, 61(3):268–278.

Graves, A. (2012). Sequence transduction with recurrent neural networks. *arXiv preprint arXiv:1211.3711*.

Graves, A., Fernández, S., Gomez, F., and Schmidhuber, J. (2006). Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd international conference on Machine learning*, pages 369–376.

Graves, A., Mohamed, A.-r., and Hinton, G. (2013). Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. Ieee.

Guo, J., Sainath, T. N., and Weiss, R. J. (2019). A spelling correction model for end-to-end speech recognition. In *Proc. ICASSP*, pages 5651–5655.

Gwilliams, L., Marantz, A., Poeppel, D., and King, J.-R. (2023). Top-down information shapes lexical processing when listening to continuous speech. *Language, Cognition and Neuroscience*, 0(0):1–14. Publisher: Routledge _eprint: https://doi.org/10.1080/23273798.2023.2171072.

He, M. and Garner, P. N. (2023). Can ChatGPT Detect Intent? Evaluating Large Language Models for Spoken Language Understanding. In *Proc. Interspeech*, pages 1109–1113.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

Honnibal, M., Montani, I., Van Landeghem, S., and Boyd, A. (2020). spaCy: Industrial-strength Natural Language Processing in Python.

Hrinchuk, O., Popova, M., and Ginsburg, B. (2020). Correction of Automatic Speech Recognition with Transformer Sequence-To-Sequence Model. In *Proc. ICASSP*, pages 7074–7078.

Jurafsky, D. (2000). *Speech & language processing*. Pearson Education India.

Krause, B., Kahembwe, E., Murray, I., and Renals, S. (2018). Dynamic evaluation of neural sequence models. In *Proc. ICML*, pages 2766–2775.

Levenshtein, V. I. et al. (1966). Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710. Soviet Union.

Li, G., Chen, L., and Yu, K. (2023). How ChatGPT is Robust for Spoken Language Understanding? In *Proc. Interspeech*, pages 2163–2167.

Louradour, J. (2023). whisper-timestamped. https://github.com/linto-ai/whisper-timestamped.

Ma, R., Qian, M., Manakul, P., Gales, M. J. F., and Knill, K. (2023). Can Generative Large Language Models Perform ASR Error Correction? *ArXiv*, abs/2307.04172.

Mikolov, T., Chen, K., Corrado, G. S., and Dean, J. (2013). Efficient estimation of word representations in vector space. In *International Conference on Learning Representations*.

Miller, G. A. and Isard, S. (1963). Some perceptual consequences of linguistic rules. *Journal of Verbal Learning and Verbal Behavior*, 2(3):217–228.

Min, Z. and Wang, J. (2023). Exploring the Integration of Large Language Models into Automatic Speech Recognition Systems: An Empirical Study. In *Proc. International Conference on Neural Information Processing (ICONIP)*, pages 69–84.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C. L., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P., Leike, J., and Lowe, R. (2022). Training language models to follow instructions with human feedback. In *Proc. NeurIPS*.

Panayotov, V., Chen, G., Povey, D., and Khudanpur, S. (2015). Librispeech: an ASR corpus based on public domain audio books. In *Proc. ICASSP*, pages 5206–5210.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. (2019). Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32.

Pu, J. and Nguyen, T.-S. (2024). Multi-stage Large Language Model Correction for Speech Recognition. In *Submitted to ICASSP*. https://arxiv.org/abs/2310.11532.

Radford, A., Kim, J. W., Xu, T., Brockman, G., McLeavey, C., and Sutskever, I. (2023). Robust speech recognition via large-scale weak supervision. In *Proc. ICML*, pages 28492–28518.

Radford, A., Narasimhan, K., Salimans, T., Sutskever, I., et al. (2018). Improving language understanding by generative pre-training.

Radhakrishnan, S., Yang, C.-H., Khan, S., Kumar, R., Kiani, N., Gomez-Cabrero, D., and Tegnér, J. (2023). Whispering LLaMA: A cross-modal generative error correction framework for speech recognition. In *Proc. EMNLP*, pages 10007–10016.

Rauschecker, J. P. and Scott, S. K. (2009). Maps and streams in the auditory cortex: Nonhuman primates illuminate human speech processing. *Nature Neuroscience*, 12(6):718–724.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning internal representations by error propagation, parallel distributed processing, explorations in the microstructure of cognition, ed. de rumelhart and j. mcclelland. vol. 1. 1986. *Biometrika*, 71:599–607.

Sarzynska-Wawer, J., Wawer, A., Pawlak, A., Szymanowska, J., Stefaniak, I., Jarkiewicz, M., and Okruszek, L. (2021). Detecting formal thought disorder by deep contextualized word representations. *Psychiatry Research*, 304:114135.

Sennrich, R., Haddow, B., and Birch, A. (2016). Neural Machine Translation of Rare Words with Subword Units. In *Proc. ACL*, pages 1715–1725. Association for Computational Linguistics.

Shannon, C. E. (1948). A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423.

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

Sohoglu, E., Peelle, J. E., Carlyon, R. P., and Davis, M. H. (2012). Predictive Top-Down Integration of Prior Knowledge during Speech Perception. *Journal of Neuroscience*, 32(25):8443–8453.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27.

Toshniwal, S., Kannan, A., Chiu, C.-C., Wu, Y., Sainath, T. N., and Livescu, K. (2018). A Comparison of Techniques for Language Model Integration in Encoder-Decoder Speech Recognition. In *Proc. SLT*, pages 369–375.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. (2023). Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Udagawa, T., Suzuki, M., Kurata, G., Itoh, N., and Saon, G. (2022). Effect and analysis of large-scale language model rescoring on competitive ASR systems. In *Proc. Interspeech*, pages 3919–3923.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, 30.

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., et al. (2016). Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*.

Yang, C.-H. H., Gu, Y., Liu, Y.-C., Ghosh, S., Bulyko, I., and Stolcke, A. (2023). Generative Speech Recognition Error Correction With Large Language Models and Task-Activating Prompting. In *Proc. ASRU*, pages 1–8.

Glossary

ASR automatic speech recognition

BERT Bidirectional Encoder Representations from Transformers

BPE Byte-Pair Encoding

CER character error rate

CTC Connectionist Temporal Classification

GPT Generative Pre-trained Transformer

GRU Gated Recurrent Units

HMM hidden markov model

LLM large language model

LM language model

LSTM Long Short-Term Memory

POS part of speech

RNN Recurrent Neural Network

RNN-T Recurrent Neural Network Transducer

WER word error rate