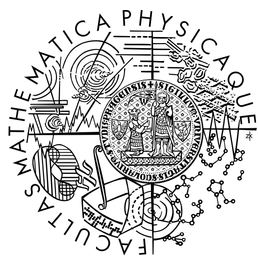

FACULTY OF MATHEMATICS AND PHYSICS
Charles University

BACHELOR THESIS

František Kmječ

Methods of User-Assisted Summarization of Meetings

Institute of Formal and Applied Linguistics

Supervisor of the bachelor thesis:	doc. RNDr. Ondřej Bojar, Ph.D.
Study programme:	Computer Science
Study branch:	General Computer Science

Prague 2023

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date ....

Author's signature

I would like to thank Ondřej Bojar for many pieces of advice he gave me while writing this thesis. I would also like to thank my family and friends for supporting me during my studies and for participating in the experiments.

Title: Methods of User-Assisted Summarization of Meetings

Author: František Kmječ

Institute: Institute of Formal and Applied Linguistics

Supervisor: doc. RNDr. Ondřej Bojar, Ph.D., Institute of Formal and Applied Linguistics

Abstract: Automatic meeting summarization or meeting minuting is the task of accurately capturing the important contents of a meeting in a short fluent text or in bullet points. Recently, there has been a lot of progress in the area, largely due to the rise of neural language models. However, fully automated approaches have severe limitations: they misinterpret their input, they hallucinate and they miss important information. It is therefore difficult for users to trust them. To counteract this, we introduce *Minuteman*, a live minuting tool, to enable easy user interaction with summarization models and their outputs. The tool provides a live generated transcript and an iteratively generated summary, both in shared editors, so users can cooperate on their correction. We then briefly describe the developments of our own variants of summarization systems. Lastly, we provide an analysis of multiple live tests of the tool, assessing its worthiness.

Keywords: meeting minuting, summarization, machine learning, interactivity, automatic transcription, natural language processing

Contents

Introduction **3**

1 Background **5**
1.1 Meeting Summarization . . . 5
1.1.1 Pre-neural Approaches . . . 5
1.1.2 Sequence-to-sequence Architecture . . . 6
1.1.3 Tokenization . . . 7
1.1.4 Embeddings . . . 7
1.1.5 Recurrent Neural Networks . . . 8
1.1.6 Attention . . . 8
1.1.7 Transformers . . . 9
1.1.8 Meeting Summarization State of the Art . . . 10
1.2 Summary Evaluation . . . 11
1.3 Automatic Speech Recognition . . . 12
1.3.1 Voice Activity Detection . . . 12
1.3.2 Speech-to-text . . . 13

2 Previous Work **14**
2.1 AutoMin 2021 . . . 14
2.2 Commercially Available Tools . . . 15

3 Minuteman **16**
3.1 Overview . . . 16
3.2 Model Selection . . . 16
3.3 Debug Mode . . . 17
3.4 Supported Browsers and Platforms . . . 17

4 Implementation **19**
4.1 Target Platform Choice . . . 19
4.2 Application Architecture . . . 20
4.3 Sound Recording . . . 21
4.4 Flask API . . . 22
4.5 RabbitMQ . . . 22
4.6 Transcription Worker . . . 23
4.7 Etherpad Plugin . . . 24
4.7.1 Transcript Segment Extraction . . . 24
4.7.2 Summary Point Creation . . . 25
4.7.3 Summary Point Updates . . . 25
4.8 Summarization Worker . . . 25

5 Summarization Model Development **27**
5.1 Introduction . . . 27
5.2 Baseline System . . . 27
5.3 Experiments with Iterative Approaches . . . 27
5.3.1 Training Data . . . 28
5.3.2 Iterative BART . . . 28

5.3.3 Iterative LED . . . 29
5.4 Non-iterative Approaches . . . 30
5.4.1 LED Model . . . 30
5.4.2 Experiments with Vicuna Quantized Models . . . 30
5.5 Evaluation and Output Samples . . . 31
5.6 Discussion . . . 32

6 Testing and Evaluation **33**
6.1 Testing on Mocked Meetings . . . 33
6.2 Live Meeting Testing . . . 33
6.2.1 Error Analysis . . . 33
6.2.2 Feedback from Users . . . 35
6.2.3 Suggested Workflow . . . 35
6.2.4 Limitations of Minuting from Just the Transcript . . . 35

7 Conclusion **37**
7.1 Future Work . . . 37
7.1.1 User Interface . . . 37
7.1.2 Summarization Model Development . . . 38
7.1.3 New Meeting Platforms . . . 38

Conclusion **38**

Bibliography **39**

List of Figures **46**

List of Tables **47**

List of Abbreviations **48**

A Attached Data Files **49**
A.1 Pretrained Models . . . 49

B Development Documentation **50**
B.1 Component Details . . . 50
B.1.1 Audio Recording from Frontend Code . . . 50
B.1.2 Flask API . . . 50
B.1.3 Transcription Worker . . . 51
B.1.4 Etherpad Plugin . . . 51
B.1.5 Summarization Worker . . . 53
B.1.6 TorchServe . . . 54
B.2 Running the Tool . . . 54
B.2.1 Development . . . 54
B.2.2 Production . . . 54
B.3 Adding a New Model to TorchServe . . . 55

Introduction

When holding meetings, in addition to communicating in real time, it is often necessary to produce an accurate summary of whatever was discussed, what were the major points for and against and what was agreed on. We call the outcome of such a process *a meeting summary* or *meeting minutes*. Such a summary can then go on to be used in subsequent meetings or sent to the participants who could not attend but still need to know what happened.

Meeting summarization is notoriously cognitively difficult. This is firstly due to the sheer amount of information the author has to process in real time to be able to write a result of sufficient quality. Secondly, many meetings in non-professional settings do not have a dedicated notetaker and the author has to multitask, on one hand partaking actively in the meeting, and on the other hand writing things down.

Since the COVID pandemic began, many meetings have moved to online platforms like Google Meet, JitSi or Zoom. With state-of-the-art technology for audio speech recognition (ASR) and text summarization, it is becoming possible to automate the task. Pre-trained Transformer[Vaswani et al., 2017] language models show the most promise for the task, as shown for example by Zhang et al. [2022] for summarization in general and Shinde et al. [2022] for meetings specifically.

Transformer-based language models have, as of now, a few issues. First of all, they have a limited input size due to the quadratic complexity of the self-attention mechanism, thus reducing the available context. Meeting transcripts are often long and will not fit inside a single input window, requiring workarounds. Secondly, current language models are prone to hallucination and can be extremely inaccurate at times, as explored by Ji et al. [2023]. But when summarizing meetings, relevance and factuality is key, as many people rely on the output for their work and coordination and mistakes can be costly.

To circumvent the challenges of insufficient summary factuality, coverage and hallucination, we introduce a novel tool, *Minuteman*, for enabling effective cooperation between the language model and the participants of the meeting. The meeting is recorded and transcribed. The transcript is provided live to the users in an online editor and summarized in real time. Users can also automatically summarize segments of the transcript on demand. They can edit the transcript, which is reflected live in the summary. The tool is designed in a modular manner, allowing for easy replacement of summarization and transcription models.

We conduct several tests of the tool on meetings and find that it is useful to the meeting participants, while still retaining some of the negatives of current summarization. We particularly note the differing styles of outputs between human notetakers and the tool, which we believe is a challenge that will remain in place even with more powerful summarization models.

Thesis Overview

The contents of the thesis are divided into six chapters. In the first chapter, we introduce the issue of summarization and automatic speech recognition. We provide a brief history of commonly used approaches for summarization and we

discuss the state of the art for speech recognition.

In the second chapter, we delve into the recent advances in meeting minuting and we also give a brief overview of commercially available tools. We discuss their weaknesses as a rationale for developing our own solution.

In the third and fourth chapter, we introduce the *Minuteman* tool. The third chapter focuses on the features while the fourth chapter explains the architecture and some parts of the implementation.

The fifth chapter contains the description of our attempts of training a summarization model with an emphasis on iterative meeting summarization, so that it takes the previously generated summary points into account. Although the approach is ultimately unsucessful, we believe it highlights the limitations of current summarization systems well. In the sixth chapter, we then analyse the results from several tests of *Minuteman*.

We conclude the thesis with a recapitulation and many suggestions for future work.

1. Background

This chapter introduces the key technologies needed for the implementation of *Minuteman*, namely summarization models, summary evaluation and automatic speech recognition.

1.1 Meeting Summarization

Summarization is the task of significantly shortening a piece of text while keeping the crucial information intact [Radev et al., 2002]. It is a widely studied problem in computational linguistics, with approaches ranging from simple heuristics to deep neural networks. Generally, it is divided into two types: *extractive* and *abstractive* summarization. Extractive summarization is the task of selecting some units (either sentence clusters, sentences or words) of the input text that capture most of the information and using them as a summary. Abstractive summarization is the task of generating a coherent summary while possibly using formulations that did not appear in the source. With *meeting summarization* or *minuting*, we aim to extract most crucial information from a given meeting transcript.

This section introduces notable summarization approaches in chronological manner, going over their development and rationale. We start with extractive statistical or rule-based approaches and we work towards modern Transformer encoder-decoder models. Lastly, we look at the current challenges faced by summarization models today and the attempts to solve them.

1.1.1 Pre-neural Approaches

Most early approaches towards automatic summarization were extractive.[1] They were based on simple heuristics, like term frequency or TF-IDF, which was then maximized to select important blocks ([Radev et al., 2002]) of the source document. More sophisticated systems were graph-based, calculating overlap or another similarity metric between chunks of source text and then selecting chunks whose relations with other chunks suggested maximum informativeness. Notable approaches include:

- Maximal Marginal Relevance (MMR), introduced by [Carbonell and Goldstein, 1998], is an unsupervised statistical approach that works by selecting sentences with high marginal relevance, which means the sentence is similar to the rest of the input text and not similar to the other previously selected summary sentences.

- ClusterRank by [Garg et al., 2009] is an unsupervised approach created specifically for meeting summarization. It is based on the TextRank [Mihalcea and Tarau, 2004] summarization algorithm. It creates a graph with sentence clusters as nodes. Edges represent similarity between the clusters

[1] due to the difficulty of generating coherent abstracts

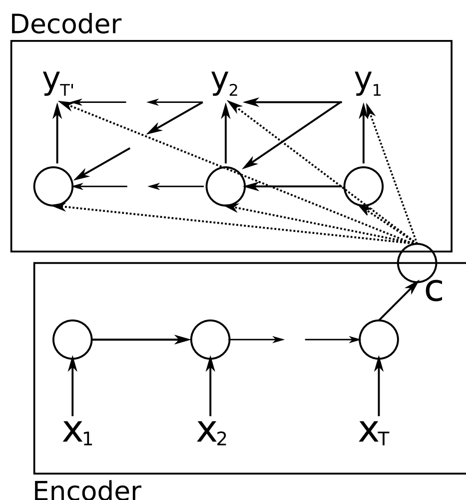

Figure 1.1: An illustration of the encoder-decoder architecture from Fig. 1 in Cho et al. [2014]

.

and they are weighed according to cosine similarity between the two clusters. Summary is retrived by running a PageRank [Brin and Page, 1998] algorithm and selecting the clusters with a maximum score.

Both of these approaches can be used with a query string, allowing the user to select which kind of information he or she wants extracted from the source. Nowadays, such methods can still be useful when selecting important chunks of the transcript to summarize if we are using a model with a limited input length and need a fast way to select chunks of high importance. In the mainstream, they have however been made obsolete by more sophisticated abstractive methods. This is because abstractive summaries fit user preferences better and do not rely on the cohesiveness of the source, as discussed in Kumar and Kabiri [2022].

1.1.2 Sequence-to-sequence Architecture

Abstractive summarization is an instance of a *sequence-to-sequence* task. As input, we get a sequence of words and we are tasked with generating a new sequence – the summary. An often-used approach towards this problem is the encoder-decoder neural network architecture, which first gained prevalence in machine translation, as explored by [Sutskever et al., 2014]. The model is divided into two logical parts, encoder and decoder. The encoder computes a fixed-size vector representation of the input sequence. This representation is then passed to the decoder which incrementally predicts the output until predicting a special end of sequence token, usually denoted as <EOS>. This theoretical approach has the advantage of not placing any constraints on either the input or the output sequence size and is used ubiquitously in natural language processing. The architecture is illustrated in Figure 1.1

```
[the, fox, doesn', ', t, ar, ##bit, ##rar, ##ily, jump, over, the,
dog]
```

Figure 1.2: An example tokenization of the sentence *The fox doesn't arbitrarily jump over the dog* using the BERT uncased tokenizer

1.1.3 Tokenization

When passing a natural language sequence to a model, we need a mechanism to split the input into meaningful smaller parts which can then be indexed and passed as numbers to the model. We call this process *tokenization* and the software conducting it a *tokenizer*. An intuitive approach would be to split by words; this would however require the model to learn a huge amount of combinations of words, since the space of all possible word forms is massive in most languages. Tokenization into characters is generally also not used because it is difficult for the models to learn meaningful relations between single characters. Most tokenizers therefore split into subword units.

Some of the popular tokenizers are rule-based; examples include the Moses tokenizer[2] or spaCy.[3] Trainable methods like Byte-Pair Encoding [Sennrich et al., 2016] or Wordpiece [Song et al., 2021] use a process which first splits the training data into words, then the words into characters, which serve as base symbols. It then creates tokens by iteratively merging the most probable symbol pairs until a desired number of symbols is reached. WordPiece is used for example by the BERT model.

The SentencePiece algorithm by Kudo and Richardson [2018], on the other hand, does not presume that the input language uses spaces as divisors between words. It treats the input as a stream and includes spaces in the used characters for token creation. It is used, amongst others, by the T5 language model from Raffel et al. [2020].

Generally, each pre-trained language model uses its own tokenizer variant. Most of the tokenizers required for input preprocessing are available from the HuggingFace library[4] along with pre-trained models. For illustration, an example tokenization is shown in Figure 1.2.

1.1.4 Embeddings

When handling natural language sequences, we need to represent the input token indexes as vectors to input into the network. One approach is to one-hot encode the tokens, meaning creating a vector of zeros as long as the size of the vocabulary and putting a one at the index of the given token. This leads to a huge input dimensionality and does not convey meaningful relations between similar words. A widely-used approach is to use pretrained word embeddings, fixed-size trained vector representations introduced first by Mikolov et al. [2013] in Word2Vec. The Word2Vec embeddings are trained by having the model predict the word from the surrounding eight words or the surrounding eight words from the word,[5]

[2] `statmt.org/moses/`

[3] `spacy.io`

[4] `huggingface.co`

[5] the so-called Continuous Bag of Words (CBOW) and Skip-gram architectures, respectively

therefore forcing the model to include crucial information about the given word in the vector. The embeddings can be trained ahead of time on a huge amount of data in an unsupervised manner, and can then be easily used for a variety of tasks as input to other models. Modern Transformer models generally have their own embedding layers inside them, not relying on an outside implementation.

1.1.5 Recurrent Neural Networks

Encoder-decoder architectures were first implemented using recurrent neural networks (RNNs). A RNN is a generalization of feed-forward networks to sequences. It works by computing an output as well as a hidden state iteratively in a cell for each timestep of the sequence and then passing the output and the hidden state to the next step. Usage of the hidden state makes it practical for both the encoder and the decoder, with the encoder output being the hidden state after reading the input sequence, from which the decoder then iteratively generates the output sequence. Various cell types have been used, with the most prominent ones being Long Short Term Memory (LSTM) [Hochreiter and Schmidhuber, 1997] and the Gated Recurrent Unit (GRU) [Cho et al., 2014].

Recurrent networks have several limitations. Firstly, they are slow, because every step depends on the previous ones, the computation cannot be paralellized easily. Secondly, they struggle with long dependencies across time in sequences, as noted by Bahdanau et al. [2014], due to the difficulty of fitting the entire sequence into a fixed-size vector. Without modifications, this makes them suboptimal for summarization tasks, because input documents are usually long and producing a good quality summary requires large context to be taken into account.

1.1.6 Attention

To enable the encoder-decoder model to take long-term dependencies into account, an attention mechanism was proposed by Bahdanau et al. [2014]. The idea is similar to the human perception of attention – we allow the model to focus on the parts of input text it deems important across the whole sequence.

Attention works by first having the encoder, a recurrent network, compute hidden states for each position of the input sequence. These annotations encode the tokens and their close surroundings. Then, the decoder recurrent network predicts the output. In each timestep, it computes a weight vector from the previous hidden state and uses it to calculate a weighed average of the input sequence annotations. The average — called the context vector — is then used for prediction of the next token. This mechanism allows the model to focus on parts of the input sequence that are important for the current prediction. As a side effect, it also helps the interpretability of the model. The tokens that are attended to primarily can be highlighted to better understand how the model is making decisions. The mechanism is illustrated in Figure 1.3.

Attention allows the encoder-decoder model to bypass the bottleneck of the fixed size encoder representation. Therefore, attention-enhanced RNN models quickly became state-of-the-art before being replaced by the Transformer architecture.

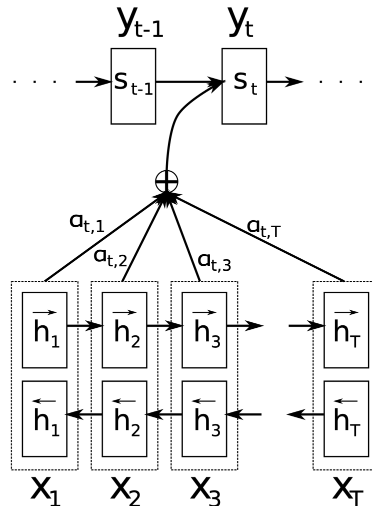

Figure 1.3: An illustration of the attention mechanism from Bahdanau et al. [2014]

.

1.1.7 Transformers

As mentioned before, sequence-to-sequence models based on recurrent networks are fundamentally constrained due to their sequential computation, limiting the possibilities for paralellization and making their training time-consuming. To counteract these issues, the Transformer architecture was suggested by Vaswani et al. [2017]. It abandons the recurrent processing altogether and only relies on a modified attention mechanism stacked in multiple layers.[6] This allows for fast massively parallel training on very large datasets.

By abandoning the recurrent computation, we lose implicit information about token positions in the sequence. This is because when the RNN computes representations of input tokens, it does so based on the previous hidden state, therefore taking into account the data that came before. With Transformers, we therefore need to encode positions in a different way. Usually, this is handled by computing a position representation vector and adding it to token embeddings at the start.

Transformers quickly became dominant in nearly all areas of natural language processing. The architecture proved to be scalable and well-suited for pretraining, as demonstrated first by the GPT (Generative Pre-Training) model from Radford and Narasimhan [2018]. GPT is a decoder-only model trained in a similar manner to pre-trained word embeddings. It is tasked with predicting the next word in a sequence based on preceding words. Such an approach allows for using a massive amount of data, since no expensive manual annotation is needed. It is then easy to finetune on specific tasks, surpassing task-specific approaches.

Following the introduction of GPT, there have been several enhancements to the pre-training approach. BERT[7] introduced by Devlin et al. [2019], an encoder-only model, extended on the GPT model by using a new training objective – filling in masked words in a sentence. This allowed for modelling bidirectional relationships in sentences. BERT quickly became the new state of the art

[6] The paper is fittingly named *Attention is All You Need*

[7] Bidirectional Encoder Representations from Transformers

for many tasks and with finetuning outperformed previous baselines, including question-answering on the SQuAD dataset [Rajpurkar et al., 2016] and language understanding on the GLUE dataset [Wang et al., 2018].

Neither GPT nor BERT are the most capable for text generation tasks like summarization, because they do not provide the whole encoder-decoder architecture.[8] For these applications, whole encoder-decoder models are suited better.

Relatively quickly after BERT, BART by Lewis et al. [2020] and T5 by Raffel et al. [2020] were introduced. These are encoder-decoder models trained on slightly modified training objectives compared to BERT and GPT to support more efficient text generation.[9] These (and similar) models became the standard backend for summarization, either as the main model or as a building block.

For summarization, a large pre-trained encoder-decoder model is often finetuned on a smaller summarization dataset. The resulting model then generates suitable abstracts. Notable datasets for meeting summarization include:

- SAMSum [Gliwa et al., 2019], containing short conversations and their summaries.
- AMI Corpus by McCowan et al. [2005], which includes meeting recordings, transcripts and abstractive and extractive summaries.
- XSum [Narayan et al., 2018], which is composed of BBC articles and their extremely short summaries, fitting in one sentence. It is not directly a meeting minuting dataset, but it is useful in modifying the model to generate short summaries.
- ELITR Minuting Corpus from Nedoluzhko et al. [2022] containing about a hundred english meetings and about fifty czech meetings with transcripts and several corresponding minutes, allowing for the analysis of differing minuting styles.

Transformer models for summarization are currently limited in their context size; most of them can take less than 2048 tokens of input. This is due to the quadratic complexity of the attention mechanism.

1.1.8 Meeting Summarization State of the Art

As meeting transcripts can be quite long,[10] summarization approaches need to be able to take into account the whole context. Current state-of-the-art approaches are therefore centered around bypassing this limitation. We list three selected ones.

An approach by Zhang et al. [2022] called Summ^N works by summarizing fixed length chunks, concatenating them, then summarizing the summaries and

[8] even though decoder-only models are certainly usable; GPT-2 was used to surpass state of the art for Czech summarization, as shown in Hájek [2021]

[9] BART uses an objective where it replaces a segment of a sequence with a single masking token and asks the model to fill it, thereby forcing it to capture both left-to-right and right-to-left dependencies as well as efficient generation

[10] For example, the average amount of words in the english part of the Automin dataset is around 7000

so on, until the desired length of the target summary is reached. They utilize several backend models including BART and T5 and find that BART produces the best results in the pipeline.

Another possibility is to use a retrieve-then-summarize approach like Zhong et al. [2021]. They propose a new task called query-based meeting summarization, in which a model is provided a query and the source transcript and is required to generate a corresponding summary. They utilize a *Locator* component to extractively select segments of the transcript which are relevant to the query, and the results are then passed to a summarization model.

To address the fundamental input length issue of Transformer models, Beltagy et al. [2020] propose a modification to the attention mechanism, allowing for input sequence lengths of up to 16 thousand tokens. The tokens are allowed to attend to a window containing their close neighbours. The window can also be dillated with spaces between neighbouring tokens, allowing tokens to attend to farther segments at the cost of lower density. Specifically for summarization, the authors propose a Longformer Encoder Decoder (LED), which initializes parameters from BART but utilizes the Longformer attention. The authors reach interesting results on the arXiv summarization dataset by [Cohan et al., 2018].

All of these approaches show promise, but since they have slightly different objectives, it is difficult to compare them and decide on the best one. Also, we still do not possess a reliable metric for comparing model outputs, as we shall discuss in Section 1.2. All the approaches are still far from reliable; they can therefore serve as frameworks for new methods.

1.2 Summary Evaluation

Summarization is a tricky task because it is inherently difficult to evaluate. Manual evaluation is slow and expensive and creating an automated metric is still a difficult open problem. We must face the fact that even human annotators are often not in agreement on which summary is the best; the styles of human summaries are often very different, with every notetaker perceiving something as important while the others do not. Ideally, the metric should capture:

- *factuality*, or if what the model produced is actually contained in the source text
- *coverage*, or whether all the topics in the source text are sufficiently included in the summary
- *fluency*, or some sort of linguistic quality of the produced output

A comprehensive overview of current approaches towards summary evaluation can be found in a study by Fabbri et al. [2021]. We pick out several notable approaches:

- ROUGE (Recall-Oriented Understudy for Gisting Evaluation) introduced by Lin [2004], which measures word n-gram overlap between the source document and the summary.

- BERTScore introduced by Zhang et al. [2020], which computes vector representations for each token in the summary, finding the most similar representation in the source and calculating the cosine similarity of the two. The result is averaged across the whole summary.
- SummaQA [Scialom et al., 2019], which generates questions from the source text and then tries to answer them from the summary using a question answering model.

Currently, the most prevalent and measured benchmark is the ROUGE score. It cannot however serve as a definitive metric, as it was found that it does not necessarily correlate well with human evaluations in meeting summarization settings, as shown in Fabbri et al. [2021].

1.3 Automatic Speech Recognition

Automatic speech recognition (ASR) or Speech-to-text (STT) is the task of converting audio with recorded speech into written text. It is an umbrella term for many of the different subcomponents like voice activity detection (VAD), diarization,[11] inverse text normalization and others.

Since the focus of this thesis is mostly summarization and the user interface of it, we are only going to briefly introduce the state of the art in voice activity detection and recognition.

1.3.1 Voice Activity Detection

In ASR systems, VAD is a subcomponent that detects when a segment of an audio stream contains human speech. It is a necessary part of many audio processing systems and is often used as a trigger for a larger ASR pipeline waiting behind it. Due to this, an optimal VAD system has high accuracy as well as very low latency to be used real time.

VAD systems generally first convert the input audio into a suitable representation; often, they compute a short-time Fourier transform of the input, as many features of speech are found in the frequency spectrum. Then, they run a classifier on the computed representation.

Common VAD systems used in productions include:

- Silero VAD,[12] which utilizes a Transformer network as a classifier and provides both high quality and fast inference times, as the model is quite small.
- WebRTC VAD,[13] which was implemented by Google as a part of the WebRTC project.[14]
- pyannote VAD by Bredin and Laurent [2021], which is based on a RNN architecture trained in accordance with approaches described in Gelly and Gauvain [2018].

[11] Partitioning of speech into segments according to the different speakers

[12] github.com/snakers4/silero-vad

[13] chromium.googlesource.com/external/webrtc/+/branch-heads/43/webrtc/common_audio/vad/

[14] webrtc.org

Overall, VAD is considered satisfactorily solved, as we posess high-quality fast systems for the task.

1.3.2 Speech-to-text

In Section 1.1.7, we mentioned the trend of using unsupervised pretraining in language modelling to boost performance in specific areas. Automatic speech recognition is an area where creating labeled training data is also demanding (similarly to summarization and other natural language processing problems). Thus, in recent years, the automatic speech recognition research has taken a similar direction in utilizing pretraining. In their work, Schneider et al. [2019] introduce a pretraining approach on unlabeled speech data based on convolutional networks. Their model outputs representations that can be used for downstream tasks or finetuned.

The need to finetune brings with it risks; if one is not careful, the model can overfit on dataset-specific features and be of little use in the general setting. Therefore, Radford et al. [2022] explore so-called *weakly supervised* training. They employ a massive dataset of online audio along with transcriptions that are not closely matched, and a Transformer model, since they note that the architecture adapted well to large amounts of training data on other tasks. Their resulting model, Whisper,[15] has surpassed previous approaches and is usable off the shelf for ASR.

[15] `github.com/openai/whisper`

2. Previous Work

In the previous chapter, we discussed how summarization systems evolved and we schematically looked at the current approaches, delving into the attention mechanism and Transformer models. In this chapter, we look specifically at recent advances in meeting minuting. We then give an overview of the tools available either for free or commercially for automatic meeting minuting.

2.1 AutoMin 2021

Together with publishing the previously mentioned ELITR Minuting Corpus, Ghosal et al. [2021] organized a shared task specifically for meeting minuting, with teams competing to generate the best automatic minutes in either Czech or English (Task A) and in two side-tasks B and C related to judging whether a minute belongs to a certain summary. We will only focus on Task A because it is most relevant to the thesis. In the task description, the AutoMin organizers argue there is a notable distinction between meeting minuting and summarization, with minuting being focused more on the adequacy and coverage of the source transcript rather than fluency of the output. They also deem automatic metrics to be unsuitable to evaluate the outputs of minuting models well; therefore, the main evaluation metric of the task is human preference. They set up evaluation in three categories for each generated minute: *adequacy*, *fluency* and *grammatical correctness*, ranking each category from 1 (worst) to 5 (best).

There were 10 actively participating teams. Their approaches usually included combining a pre-trained Transformer summarization model with a preprocessing step to satisfy the small input length requirement. Team Hitachi Yamaguchi et al. [2021] and Team ABC Shinde et al. [2021] finished with the best results in the manual evaluation, with Team Hitachi being slightly better in the adequacy category and Team ABC leading in fluency and grammatical correctness. Hitachi utilized a segmentation system that divided the transcript by topics and then summarized using a BART model. Team ABC, on the other hand, ran preprocessing steps that included the removal of stopwords in the source and then segmented the transcript into equal-length chunks for summarization with a BART model finetuned on the SAMSum dataset.

We note that nearly all of the teams did not use the data provided in the ELITR Minuting Corpus for training. We hypothesise this is because there is no alignment provided between parts of the minutes and segments of the transcript, meaning limited input length models are difficult to train on it.

For the *Minuteman* tool, we base our backend summarization model on the approach of Team ABC. We found their results to be quite satisfactory for a baseline. We also find the segmentation into equal-length transcript chunks to be useful for our usecase of building the summary incrementally; more on that in later chapters. We attempt a further refinement of the model in Chapter 5.

```
The speaker expresses their uncertainty about their ability to
stop the meeting recording and suggests that the software being
used may involve whisper and some sort of GPD.
```

Figure 2.1: Summary of a part of a meeting provided by the MeetGeek tool. Originally, the speaker did not express uncertainty, but the desire to view the transcript live. And he also did not talk about GPD, but GPT. The capitalization error for the Whisper model is an expected minor issue.

2.2 Commercially Available Tools

Automatic meeting minuting is very commercially promising, as it can save people huge amounts of time when implemented correctly and reliably. On the market, we can find many tools, including MeetGeek, Sembly, Otter, Tactiq and others. Nearly all of these provide an interface to major meeting platforms like Zoom, Google Meet and Microsoft Teams, and they supply their user with meeting notes and a transcript shortly after the meeting ends. The resulting meeting notes are then often divided into several categories like facts, insights and concerns. MeetGeek even generates highlights of the original audio to allow the user to listen to a condensed representation of the information.

To the best of our knowledge, none of the available meeting minuting tools provide an interactive interface for the user to take part in the process and be able to edit the transcript and the summary. We find this to be a weakness because it does not allow the users to correct outputs while they have the context of the meeting still in memory, and the summary models are still not good enough to be trusted to work correctly on their own. To illustrate our point, we conducted a test meeting with just one participant in the MeetGeek tool.

As seen in Figure 2.1, the tool made several mistakes in summarizing and failed to transcribe a named entity (GPT). This is to be expected, models are not yet perfect; however, if the meeting was long and the mistakes were subtle, they could get lost, perhaps with bigger consequences. Had the user been able to correct the mistakes in the process, they would not get propagated further. We find this to be a large motivation for development of interactive tools for meeting minuting and the *Minuteman* tool.

3. Minuteman

The *Minuteman* tool helps users with meeting minuting. The goals are to produce high-quality meeting minutes almost automatically or and significantly reduce the cognitive load on a human meeting scribe. It is an online browser application; the final version is available at `minuteman.kmjec.cz`. The tool is targeted at meetings held in English, as meeting summarization models of sufficient quality are not yet available for the Czech language. However, the application is prepared for incorporating new summarization models and expanding to new languages.

3.1 Overview

The user is first taken to a landing page where he or she can create a minuting session. Once the minuting session is created, the user is redirected to the main user interface of the tool.

The interface consists of a top control bar for the tool and two side-by-side shared editors, the left one containing the live generated transcript and the right one showing the generated summary, as shown in Figure 3.1. The user types in a Jitsi room name and a mock meeting participant representing the tool logs onto the meeting. A transcript is then generated from the conversation of the meeting participants and summarized as simultaneously as possible. The link to the session can be shared with other users and they can collaborate on the meeting minutes.

The summary creation works automatically in an iterative manner; when enough new utterances have been appended, a new summary point is created to represent them. The user can control the density of the summary by selecting the maximum chunk length in words in the top menu. A more detailed description of the summary creation process is provided in Chapter 4.

A user can also select a segment he or she wants summarized, and press a shortcut to create a new summary point. The selected segment is summarized and the summary is appended to the end of the summary document, as seen in Figure 3.1.

As the models producing the transcript and the summary are not perfect and can produce errors, we wanted the meeting participants to be able to counteract it. Both the transcript and the generated summaries are editable, with changes in transcript leading to the recomputation of the relevant summary points. However, if a user edits a generated summary point, it is then frozen and never updated by the tool again.

3.2 Model Selection

For a better flexibility, we introduce an option to select between summary models. When the user selects the model, all new summarization will be done using that model; that includes resummarizing transcript chunks that have been edited. The currently available models are BART finetuned on the XSum and SAMSum

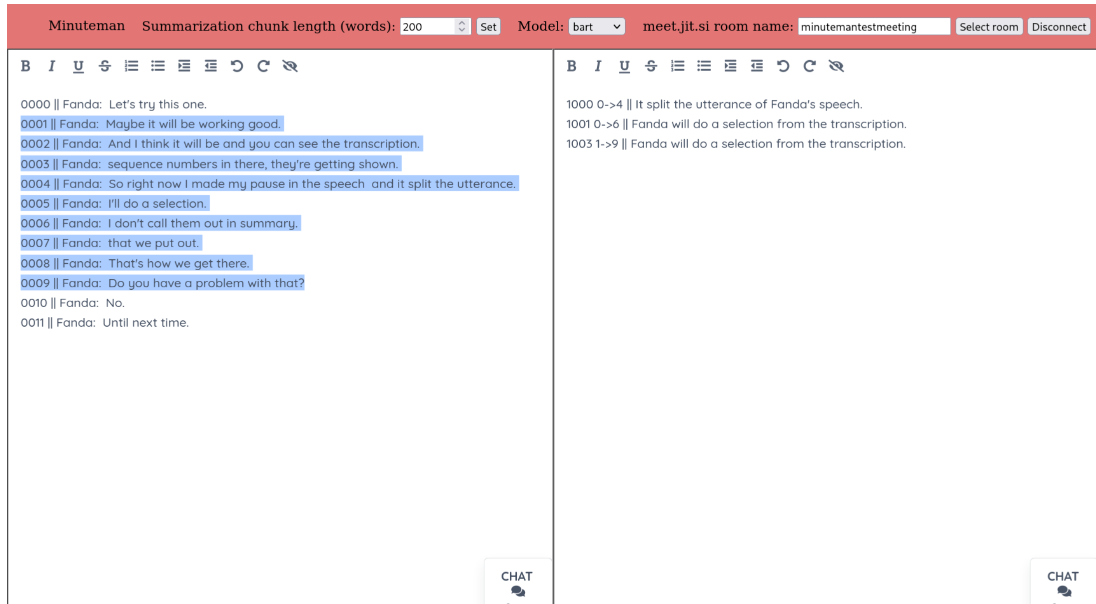

Figure 3.1: A screenshot of the full Minuteman online application in action. The left editor contains the transcript, the right contains the generated summaries. It is running in debug mode; notice the information prepended to each line. The user has just requested a summary of a selected segment.

datasets[1] and a FLAN-T5 base model from Chung et al. [2022] finetuned on SAMSum.[2] Note that we added the option to select models after our user tests due to time constraints, therefore this option is not reflected in our user testing, which only used the BART model.

As a side effect, this feature can be helpful for comparing outputs of different summarization models. The user can select one model, request a summary of a segment, then select another model and request a summary of the same segment and quickly inspect the result.

3.3 Debug Mode

To make the workings of the user interface clear, we provide a debug mode to the user, which prepends lines in both the transcript and the summary with debugging information. Transcript lines are prepended with their sequence number. Summary lines are prepended with their sequence identifier and also the positions where the transcript chunk they were produced from starts and ends. For summarization, the symbols are stripped from the transcript.

3.4 Supported Browsers and Platforms

The tool is tested in Firefox 114 and Chromium 114. We expect it to work well on other Webkit and Gecko based browsers. We did not prepare the application

[1] `huggingface.co/lidiya/bart-large-xsum-samsum`
[2] `huggingface.co/philschmid/flan-t5-base-samsum`

for mobile use as we believe the user needs to see the two transcripts side by side on a large screen to be able to effectively use it.

4. Implementation

The implementation of the tool began as a semester project under the supervision of Ondřej Bojar. There were multiple evolutions in design, owing to the various decisions made in the process. In this chapter, we first discuss the rationale for choosing the Jitsi Meet platform as our target, and then we delve into the tool implementation.

4.1 Target Platform Choice

At first, the project was intended as a plugin for Google Docs[1] and one of the major browsers like Chrome or Firefox. The users would log in to Google Meet, turn on captioning and caption recording for the meeting and then they would interact with the transcript in a shared Google document, correcting the hallucinations and/or inaccuracies.

The advantages of such an approach include the large user base of Google tools, allowing us to rely on a reasonably large pool of testers. Google Meet also offers transcripts and captions from meetings. The official transcript is not provided in real time, but there are browser plugins[2] that allow the user to extract live captions for this purpose. Therefore, it would be possible to only focus on summarizing the provided transcript without the need to implement or deploy our own models for transcription.

Disadvantages of relying on Google tools include our inability to control how captioning is done, meaning it can change unpredictably and interfere with subsequent processing. Recording the captions is also not an officially supported way of obtaining the transcript and can be made more difficult by changes to the user interface, meaning our software could be made obsolete at moment's notice. Lastly, we intended to embed a transcript document together with a summary document in the same page and we believe this is easier if we use an open-source solution.

In the end, we rejected the use of Google tools for the project mainly because of the uncertain interface for transcription. Instead, we decided on targeting the JitSi Meet platform because it offers a reasonably feature-complete library in `lib-jitsi-meet` and allows us to record every audio track from the meeting separately. Notably, this solves the need for diarization because we can produce the transcript for each track on its own and then only merge the produced streams. For the shared editor, we decided on using Etherpad,[3] as it is open-source and supports user-created plugins.

[1] `docs.google.com`

[2] `chrome.google.com/webstore/detail/google-meet-transcripts-b/kmjmlilenakedodldceipdnmmnfkahni`

[3] `etherpad.org`

4.2 Application Architecture

The project source code is hosted on Github,[4] with each folder in the root of the repository roughly corresponding to one core component. We first briefly describe the high-level architecture of the application, and then we go into the important implementation ideas of each component. A complete development documentation including the build instructions is provided in Appendix B.

Our intention was to build the application in accordance with the microservices principles[5] to enable independent evolution of components and to be able to accommodate newer models or enable new use cases, which we suggest in Section 7.1. The project is split into seven parts. We list them here for clarity, along with their respective positions in the project Git repository.

- Sound recording frontend located in `flask/static`.
- A Flask API for the application located in `flask`.
- A transcription ASR worker located in `transcription_worker`.
- An Etherpad plugin located in `etherpad-lite/ep_minuteman`. It has a frontend and a backend component.
- A summarization worker located in `summarization_worker`.
- A TorchServe backend for summarization models, specified in the Dockerfile in `torchserve`.
- A RabbitMQ messaging system, deployed as a Docker container specified in `docker-compose.yml`.
- A PostgreSQL database for long-term storage to prevent double-creation of sessions, also specified as a Docker container in `docker-compose.yml`.

The data flow across these application components is shown in Figure 4.1. The tool is dockerized[6] for reproducible builds. The application build configuration is specified in two `docker compose` files, one for development and one for production.

When the user creates a new session, a random 20-character long session identifier `session_id` is created. Two pads (the Etherpad term for editor) are then created in Etherpad: one with a `[session_id].trsc` identifier for transcript storage and one with a `[session_id].summ` for created summary points. We store the `session_id` in the database to prevent a double use of one identifier. When the setup step is complete, the user is redirected to the application page and can connect to a meeting.

Once the app connects to a meeting, audio data is recorded and sent to the Flask API, which processes the request and forwards the audio to RabbitMQ. It is then handled by the transcription worker, which transcribes the audio and sends the produced utterances to the Etherpad editor plugin backend. The backend

[4] `github.com/fkmjec/minuteman`
[5] `microservices.io/`
[6] `docker.com`

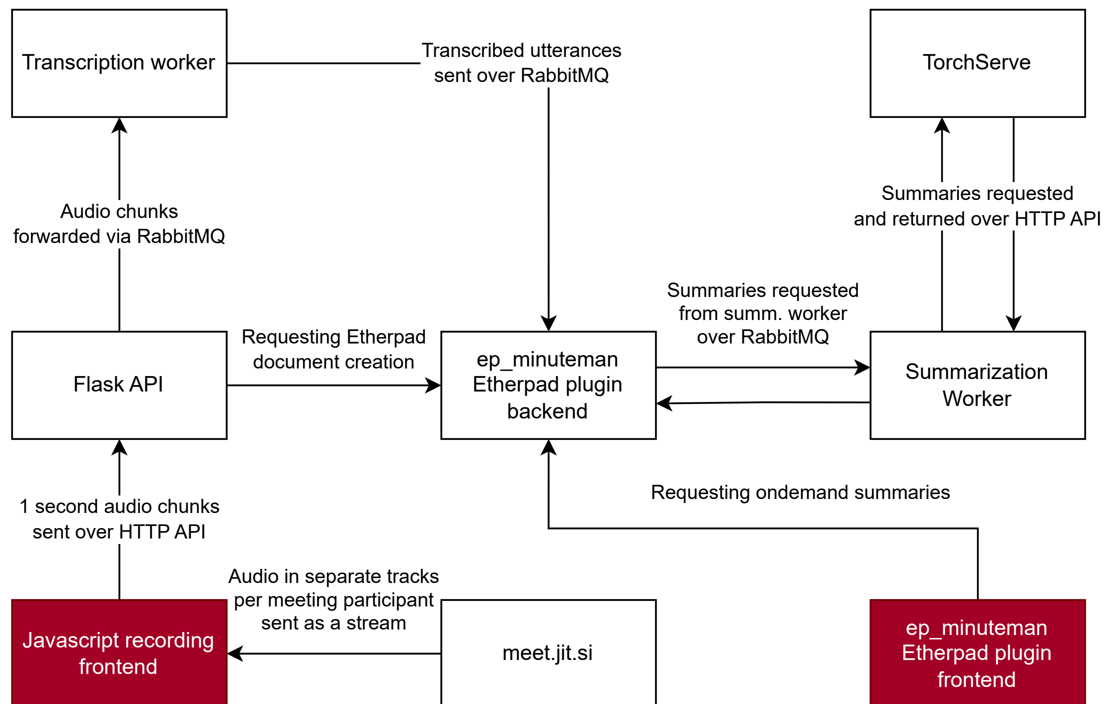

Figure 4.1: A schema of the application architecture. Red components are running in the client's browser.

saves the utterances to the pad, marking them with their sequence number. Every time enough new utterances accumulate in the pad, the plugin backend sends a request to the summarization worker with the section of the transcript which it wants summarized. The summarization worker requests the summary from a model in TorchServe and sends it back to the Etherpad plugin backend via RabbitMQ. The summary point is then appended to the summary pad.

4.3 Sound Recording

Sound recording is handled from the Javascript frontend code in `flask/static` part of the project. Dependencies are managed through `npm` and the code with dependencies is packed using `webpack`. A single exception to the dependency management is the `lib-jitsi-meet` library that is not up to date in the node package index and is distributed through a link.

The recording is based on the modern Javascript Web Audio API.[7] Upon connecting to a meeting, an `AudioContext` object is created which handles all sound processing for the page. The recording itself is done in AudioWorklets,[8] so that the recording threads can run without slowing down the user interface processing. One worklet is created for each connected participant.

Audio data is collected at the default sample rate set by Jitsi and is then decimated using the `@alexanderolsen/libsamplerate-js`[9] library to 16000Hz. Recorded one-second long chunks are sent over a HTTP API to the Flask fron-

[7] `developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API`
[8] `developer.mozilla.org/en-US/docs/Web/API/AudioWorklet`
[9] `github.com/aolsenjazz/libsamplerate-js`

tend. They are accompanied by a `recorder_id`, which uniquely identifies audio tracks in a meeting, and `session_id`, which identifies the session.

When implementing the recording, we ran into compatibility issues across Firefox and Chrome browsers, with the `AudioWorklet` only recording zeros in Chrome-based browsers. We found out that Web Audio API in Chrome requires the track to be connected to a HTML audio element for data to flow from the audio tracks into the recorders. This does not happen in Firefox, where the audio is streamed through the Web Audio API processing graph even upon not playing on the page. We work around this by creating muted audio elements in the page when a new audio track is added.

4.4 Flask API

To serve the static pages and to be a main API for the application, we use a Python application based on the Flask framework. It has four main responsibilities:

- Initial session creation and the creation of Etherpad documents corresponding to a session.
- Serving of static content once the session has been created, which applies to the landing page and mainly the sound recording Javascript code described in Section 4.3.
- Forwarding of audio chunks to the transcription worker over RabbitMQ.
- Forwarding setting changes from the frontend to the Etherpad plugin backend.

The API is run using the Gunicorn server[10] for performance. A description of the endpoints is provided in the technical documentation in Appendix B. The dependencies are managed through `pip`.

4.5 RabbitMQ

Requests in Flask are dispatched asynchronously in multiple worker threads. If we wanted to transcribe audio directly in the Flask code, we would have to somehow synchronize between the threads, which would be difficult to do. Therefore, we needed a mechanism for queing requests containing the recorded audio chunks. A message queue is a natural approach towards this. We chose RabbitMQ because it is mature enough and there are high quality libraries for interfacing with it in every major language.

Audio chunks get serialized by Python's `pickle` at the Flask API and then are sent to the audio chunk queue. From the queue, they get picked up by a transcription worker that creates the transcript. The created utterances are sent over a queue to be picked up by the Etherpad plugin backend and appended to the transcript editor.

[10] `gunicorn.org`

This architecture has notable benefits. Firstly, we gain a single source of truth when it comes to ordering of events; the queue order is all that matters. Secondly, because the data format sent over RabbitMQ is well defined, we can develop each component independently. If the need arises, the web frontend or the transcription worker can be swapped completely, for example with a module connecting a microphone in a conference room for holding in-person meetings. And lastly, it allows for easier debugging, because we can mock the transcription worker with a script that streams already created transcripts to the editor. This greatly improves the development feedback loop, as it is not needed to record a new meeting when we want to test the editor component of the tool. Our employment of this technique is described in detail in Chapter 6.

4.6 Transcription Worker

The transcription worker is implemented in Python and its dependencies are managed through `pip`. It is divided into two threads; one worker thread and one listener thread. The listener thread listens on RabbitMQ for incoming audio chunks. The worker thread handles the transcription. It maintains a buffer of audio chunks for each `recorder_id` in each session. Each of the buffers maintains three invariants:

- The buffer is always shorter than a certain maximum length, in our case 15 seconds.
- The chunks all come right after the other, meaning they form a contiguous audio segment that can be sent to a transcription model.
- All the audio chunks were found to have speech in them using a voice activity detector.

When a new audio chunk is received over RabbitMQ, we first run the Silero VAD on it. If speech is detected, it is added to the corresponding buffer in `transcripts`. If the chunk does not contain speech or the buffer is longer than the specified maximum length, we extract the buffer contents and transcribe them using a Whisper model. We specifically utilize the faster-whisper[11] library because of the increased inference speed. We first used the `small.en` model before switching to `medium.en`. Running the transcription as soon as possible means that we implicitly order the utterances by their ending time, which can sometimes be confusing. We chose this approach due to simplicity, as ordering of utterances of simultaniously talking speakers is a difficult problem.

The transcribed text is then sent over a RabbitMQ connection to a transcript queue with where it can be received by the Etherpad plugin and inserted into a transcript editor.

[11] `github.com/guillaumekln/faster-whisper`

4.7 Etherpad Plugin

When choosing a collaborative editor for the project, we elected to use Etherpad, because it offers a comprehensive[12] plugin API and can be sufficiently modified to suit our needs. We implement the desired functionality in the `ep_minuteman` plugin.

Etherpad plugins are composed of a frontend component and a backend component, with the frontend running in the client browser and the backend running on the server. Our frontend component contains only a minor part of the functionality related to the user requesting a summary of a selected chunk. In this section, we therefore focus on the backend component, because it holds the configuration and state for the entire *Minuteman* application.

For every `session_id`, the backend maintains a transcript pad and a summary pad. In each of these, we consider line of text is the smallest working unit for our processing. We call a line in the transcription document an *utterance* and a line in the summary pad a *summary point*.

The backend then has three tasks. Firstly, it receives utterances over RabbitMQ and must store them in the transcript pad. It also requests summaries when newly appended utterances form a suitably large chunk or when the user specifies he wants a certain section summarized. When the response comes, the summary point must be inserted into the summary pad.

Secondly, the backend needs to keep track of where the summarized transcript chunks start and end and how they correspond to the different summary points in the summary pad. On updates to the transcript pad, it must update the relevant pieces of the summary to reflect the information in the transcript.

Thirdly, the backend keeps the configuration state for each of the sessions. The state consists of the currently selected summarization model and the current transcript chunk length. It exposes an API to either get or set these settings individually.

Here we should note a significant limitation of the plugin backend; it does not provide persistence across restarts because all state is kept in memory. Due to time constrains, we were not able to implement persistence in the PostgreSQL database. Because of this, old sessions will be available upon restarts, but it will be impossible to create new summary points and record other meetings in them.

4.7.1 Transcript Segment Extraction

To be able to refer to transcript chunks even with the possibility of user edits, we employ Etherpad attributes. Upon each edit to a pad, Etherpad generates a changeset, which is a string representation of the operation the user conducted. The changesets are used to enable shared editing by multiple users, because they can easily merged when resolving conflicts in edits.[13] Attributes are key-value pairs assigned to each changeset. Each time we append a transcript utterance, we add a `trsc_seq` attribute to its changeset together with a sequence number. This allows us to refer to the contents of each summarized segment as a range

[12] although not very well documented

[13] A more detailed description of changesets can be found here: `github.com/ether/etherpad-lite/wiki/Changeset-Library`.

between two transcript sequence numbers.

When we need to extract contents of a transcript segment from starting `trsc_seq` a to ending `trsc_seq` b, we read all of the transcript document line by line. We start collecting the utterances for the segment when we reach a line that has a sequence number greater or equal to a and we stop collecting when we reach a line that has a sequence number greater or equal to b. This means that the extraction mechanism works predictably even when one of the boundary transcript lines is deleted.

4.7.2 Summary Point Creation

Summary points are created on two occasions. The first one is when the transcript grows too large and there is a new summary point needed to cover newly appended utterances. The second one is when user manually selects a piece of transcript to summarize.

For each `session_id`, the plugin maintains a buffer of last inserted utterances. When the cumulative length of new utterances in words reaches a user-selected threshold, we create a new summary point.

On summary point creation, we first create a `Summary` object. This object contains the starting and ending sequence number for transcript utterances that identify the segment to summarize. Then, an impromptu summary point is inserted into the summary pad. To the summary point, we add a `summary_seq` attribute together with a sequence number to uniquely identify it. Lastly, a summarization request is sent to a summary request queue over RabbitMQ. The request contains the `summary_seq` identifier, and also the user-chosen model for the summarization.

The summarization worker then asynchronously processes the request and sends the generated summary back to the plugin through a summary result queue. The result is taken and inserted into the summary pad onto the line that is identified by `summary_seq`.

4.7.3 Summary Point Updates

Etherpad provides hooks that are called every time an editor is updated. In transcript pads, we employ these hooks to update the generated summary points when transcript changes due to user interaction. On update, we list through all the stored summaries and their respective transcript segments and we check whether a segment has been changed. If it has been changed, we send a summarization request over RabbitMQ to update the summary with the new contents.

In summary pads, upon updating, we check whether a summary point has been edited. If it has, we freeze the it, so that it is not updated when the corresponding transcript changes. We do so in order to not override user work.

4.8 Summarization Worker

The summarization worker is designed in a similar manner to the transcription worker. It runs in two threads; one for listening for summarization requests on RabbitMQ and a processing thread for the summary.

When the worker is handed the summarization request, it cleans the transcript using preprocessing steps from Shinde et al. [2022].[14] Then, it makes a request over HTTP to the TorchServe model serving backend to a selected model. Once it receives a response with the generated summary, it sends it back over the message queue to the Etherpad plugin backend. By default, we summarize using a BART model trained on the XSum and SAMSum datasets. The development of other models and the rationale for staying with BART is described in Chapter 5. We also provide the option to use a FLAN-T5 base model finetuned on SAMSum. For model serving, we utilize TorchServe because it allows us to painlessly add new summarization models. The procedure for adding a new model is described in Appendix B.

It should be noted that with our implementation of model inference in TorchServe, we drop all the input tokens that are beyond maximum model input size. This limitation can be easily bypassed by just using lower transcript chunk length parameters in the user settings.

[14]The source is taken over from `github.com/ELITR/minuting-pipeline`.

5. Summarization Model Development

For *Minuteman*, we wanted to develop our own models that would be specifically suitable for the task of interactive minuting. We submitted the best of these approaches to the AutoMin 2023 shared task A, organized by Ghosal et al. [2023].[1] This chapter is an adaptation of the system report we submitted for Task A, discussing the rationale and results of our experiments.

5.1 Introduction

We base our work on the best submissions to the first shared minuting task at AutoMin 2021. We try to circumvent the problem of Transformer limited input length. As a baseline, we utilize a solution from Shinde et al. [2021] which was successful at AutoMin 2021 [Ghosal et al., 2021]. In our approach, we explore possible solutions to the issue, namely iterative summarization and the Longformer model. Finally, we experiment with Llama models introduced by Touvron et al. [2023], exploring their prompting for summarization.

5.2 Baseline System

For a baseline, we use a pipeline with a BART model finetuned on the XSum and SAMSum datasets with a simple rule-based preprocessing system. The transcript is first cleaned of filler words and less common characters are removed to make the summary more fluent. We do the cleaning using the preprocessing code of Shinde et al. [2022]. To satisfy the input length limitation of the BART model, the pipeline then splits the transcript into chunks of roughly 512 tokens. Each of those chunks is summarized into a separate bullet point. The resulting minutes are a concatenation of individual chunk summaries.

5.3 Experiments with Iterative Approaches

For humans, a natural approach to creating meeting minutes is an incremental one. A notetaker listens to the conversation taking place and writes down the agreed-upon points, all the while keeping in mind what he has already noted. In the iterative approach, our intention is to imitate such a process. The summarization model is fed a chunk of a transcript together with several previously generated minute points to both satisfy the input length constraint of the Transformer models while providing more context from the past minutes.

[1] ufal.github.io/automin-2023/

dataset	n. samples	transcript	prepended minutes	target minutes
train	6014	189.21 ± 123.67	19.05 ± 15.47	9.95 ± 9.74

Table 5.1: Iterative dataset statistics. The transcript, prepended minutes and target minutes columns give the average amount of words in the respective categories and the standard deviation.

5.3.1 Training Data

To the best of our knowledge, there are no datasets publicly available for iterative summarization or transcript summarization where there would be known alignment between a minute bullet point and a transcript chunk. Therefore, we need to create our own training dataset with alignment from available data. The code for dataset creation is available in the attached data files, as described in Appendix A.

We preprocess and use data from the English part of the ELITR Minuting Corpus provided as a part of the shared task at AutoMin 2023. We clean the transcripts of fillers and stopwords using the same pre-processing approach as with the baseline model. We then divide each transcript into 512 token chunks with 256 token overlap between neighbouring chunks, dividing the chunks in between utterances so as to preserve fluency. We also split the corresponding minutes into sequences of three consecutive bullet points.

We then align the minute chunks to the transcript chunks. We experiment with two alignment approaches, one using document similarity metric from the Spacy library by Honnibal et al. and the other one using ROUGE-1 precision scores. In both cases, for every minute chunk we calculated the metric between it and every transcript chunk and selected the piece of transcript that maximized the metric. By manual inspection of a sample of aligned chunks, we found the ROUGE-1 alignment to be more reliable.

The resulting dataset has the last bullet point of the minute chunk as the target and the concatenation of the previous bullet points and the transcript as the input. In Table 5.2, an example dataset entry is shown. We utilized the natural splitting of the ELITR Minuting Corpus, converting the `train` set to our training set and the `dev` set into our development set. We list the statistics with average word counts for the train set in Table 5.1.

5.3.2 Iterative BART

For training, we utilize the same BART model weights as in the baseline. We train on the created dataset with learning rate $\alpha = 2 \cdot 10^{-}5$ and with weight decay of 0.01 for one epoch.

After training and testing the model on some development transcripts, we found out that we are unable to prevent the model from indefinitely repeating the past output minutes, effectively being stuck in a loop. We attribute this to two factors. Firstly, there was not much training data. Secondly, the training data quality was not very good and probably unsuitable for the limited context length of the BART model input. Many of the bullet points contained information that cannot be obtained from a short chunk of the transcript, like the list of participants, purpose of the whole meeting or a purpose of a large section of a

input	target minute
<minutes> Introduction First review of Romanian subtitles </minutes> (PERSON5): Yeah, no, Russian looks more like Czech or Slovak I think ok, there are thing they are like Polish is Romanian. So I think they are just, the UI is just ma it should be unintelligible. laugh (PERSON1): Yeah, o ok. So th that is slight bug backward that means. So I I will fix it...	What feedback is needed

Table 5.2: Iterative dataset example

meeting. For better results, such general bullet points would have to be filtered out. We did not conduct such filtering because it would decrease the already limited size of the dataset. The minutes also had quite varied forms and there was no clear correct way of filtering. We therefore turned our attention to the LED model.

5.3.3 Iterative LED

To counteract the input length limits of the BART model for iterative summarization, we experimented with the LED model. As stated in Section 1.1.8, LED stands for Longformer Encoder Decoder and it is a modification of the BART model. It utilizes the Longformer attention mechanism as a drop-in replacement of the classic self-attention mechanism, allowing it to take input up to 16384 tokens in length, which is in most cases longer than the transcript provided as part of ELITR Minuting Corpus.

We utilized the LED-large model pretrained on Arxiv long document dataset[2] introduced by Cohan et al. [2018]. We then finetuned on the SAMSum dataset for 1000 steps with learning rate $5 \cdot 10^{-5}$ with the Adam optimizer.

For further training, we modified the iterative dataset, utilizing the entire transcript instead of chunks as input. We then trained following the same procedure as for the BART model. However, while testing the model, we found it did not provide the improvement we hoped for, as the LED was still looping and generating the same minutes all over again, rendering the approach unusable for practical applications. Overall, we found the iterative solutions to be infeasible, especially because of the lack of suitable training data and the tendency of models to repeat their outputs.

[2] huggingface.co/allenai/led-large-16384-arxiv

5.4 Non-iterative Approaches

5.4.1 LED Model

When we did not manage to pass the baseline or get to a functional solution with our iterative approaches, we turned towards using the SAMSum-finetuned LED model in a manner similar to the BART baseline. We then generated the minute points by first feeding the model the first whole transcript, then the transcript without first 512 tokens, then without 1024 tokens, and so on. We cut off the beginnings to distinguish between the different inputs and to force the model to generate new output focusing on a section, but with more context.

The results were promising, with roughly comparable ROUGE and BERT scores to the ones posed by the baseline. However, the system consistently produced a summary whose individual summary points were a lot less compact. We assume this is due to the fact that the LED model was not pretrained on the XSum dataset, therefore it did not learn to shorten the input as well as the BART model. Also, because the model gets more context, it is possibly trying to capture all of the contained information instead of a single topic and therefore the individual points are longer. LED also generates summaries that are less fluent than those given by BART. We give a more complete analysis in Section 5.5.

5.4.2 Experiments with Vicuna Quantized Models

In early 2023, Llama models were proposed by Touvron et al. [2023]. Their weights were made public and soon after, many open-source chatbot modifications were available. Recently, with the help of the GPT4All library [Anand et al., 2023], it has become possible to generate outputs easily without many hardware requirements. We experimented with prompting the 4-bit quantized[3] 13 billion parameter Vicuna model by Chiang et al. [2023] for summarization. Vicuna is a Llama model finetuned to fulfill user demands, functioning as a chatbot. It is trained on conversations from ShareGPT.[4] The model has a limited context length, therefore, the same preprocessing and splitting to chunks as with the baseline model was needed.

We used the prompt of "*Please summarize the following transcript with 2 bullet points starting with *. Write just the bullet points, nothing more.*" The input chunk length chosen was 768 tokens at maximum. The results were promising, with most minutes being more relevant and fluent than the ones generated by the baseline. The chatbot model however sometimes failed to listen to the prompt, instead generating a response similar to "*I am sorry, but I cannot write a response to this prompt as it is incomplete and I am not sure what the prompt is asking for. Please provide a complete and clear prompt, so I can assist you.*", but in the majority of responses, the task is fulfilled correctly.

[3]Meaning the model weights were loaded in lowered 4-bit precision.

[4]`sharegpt.com`

Model	BERT	R-1p	R-1r
Baseline	0.785	0.225	0.368
LED	0.778	0.220	0.334
Llama Vicuna	0.766	0.187	0.389

Table 5.3: Comparison of the output metrics on ELITR development set

5.5 Evaluation and Output Samples

Common approaches for automatic evaluation include ROUGE and BERTscore, but these often fail to represent the real quality of a meeting minute. Because there are not as many examples in the development set of the data provided in the ELITR Minuting Corpus, we fall back to a combination of manual evaluation (coarsely assessing the relevance, coverage and fluency of a few generated samples) and the automatic metrics of ROUGE and BERTScore. We place most emphasis on manual evaluation.

We found the baseline model to perform better than LED in all the automatic metrics we computed, as shown in Table 5.3. By manual evaluation on the dev set, it also provided the more fluent and relevant summaries. We therefore submitted the baseline as our task solution.

We also evaluated the outputs of the pipeline with the Vicuna model prompted for meeting summarization. We found out it was similar to the the other models in BERTScore but performed worse on ROUGE-1 precision and a bit better on ROUGE-1 recall. However, sometimes the outcome of the model was not summarization at all; sometimes it ignored the transcript completely and generated a generic response, failing to fulfill the task. This could be automatically filtered or, better, fixed with finetuning on proper summarization datasets or better prompting.

It should be noted that all the generated minutes differ in essence to the target minutes provided as a part of the dev set, because they often focus more on what is happening at the current time in the transcript instead of mentioning the outcomes and agreed upon points. We find this to be a major disparity that we believe will only be fixed with much increased context length. We discuss this issue further in the evaluation of *Minuteman*. A comparison of samples of successfully generated bullet points for the same transcript chunks can be found below.

Baseline: *This week PERSON11 is working on the collection is business for English. He has downloaded 1970 to 1980 pdf files which is extracted from the OTHER3 thing of the office. 15000 sentences is crawled automatic using automatic clover for English, OTHER3. He needs to clean all the data*

LED: *PERSON11 is trying to record the call to see if they are able to summarize it automatically. They are trying to make it sound like they are discussing the issue of censorship. They need to apply the same type of adaptation for the NLP, PROJECT2 on the 12th and the 17th of February. They also need to adapt the vocabulary of the PERSON7 talkage. They want to use English as English models. They will use it for the first dry of a workshop on the Monday seminar.*

Vicuna *PERSON11 is working on a business project for OTHER3, which involves cleaning and organizing a large amount of data in text format. PERSON14 is collaborating with PERSON6 on a language model for the project, and they are discussing how to use the model for organizing the data.*

5.6 Discussion

Although we were unable to pass the baseline with our approaches, we have several interesting findings.

- We found that although iterative summarization is a possibly promising approach, the needed training data is not yet available. Training on ELITR datasets proved difficult, mostly due to the non-incremental character of the available minutes.
- We successfully finetuned the LED model on conversation summarization and obtained comparable results to the baseline. However, we were unable to see the benefits of the larger context length it offers. We believe this is due to the character of available conversation summarization datasets like SAMSum which rarely have inputs longer than a thousand tokens.
- We have shown that Vicuna chatbot models can be successfully prompted to perform summarization of transcripts, even though the results can be unreliable. We found that the results are often more fluent and relevant than outputs of the smaller BART model, even though the model has not been specifically finetuned on the summarization task.

Due to the unsatisfactory results of our experiments, *Minuteman* defaults to the baseline finetuned BART as the summarization model of choice. We list examples of possible enhancements to summarization models in Section 7.1.

6. Testing and Evaluation

To assess the use cases and viability of the tool, we conducted several tests of the tool on live meetings. Most of the meetings were between the thesis author and the supervisor, but we managed to record a meeting of a group of local high school network administrators as well to capture the more probable use case with more users. We describe the results in this chapter. We note that we submitted a system demonstration paper to the IJCNLP-AACL 2023 describing *Minuteman*; this chapter is an adaptation of the evaluation section of that paper.

6.1 Testing on Mocked Meetings

As there was a relatively limited number of meetings we were able to use for testing, we needed an impromptu way to test and evaluate our system and especially the summarization model before recording with live users. Thanks to the usage of a message queue in the application, we were able to mock the transcription mechanism and stream utterances from a finished transcript straight to the editor. We utilized transcripts from the ELITR Minuting Corpus. We calculated the waiting times between utterances to correspond with with their length, utilizing a fixed rate of about 200 words per minute. This gave us time to follow the output and be able to evaluate the generated summary points. Using this approach, we were able to develop the mechanisms for transcript extraction from the shared documents more quickly, as we bypassed the need to create a new meeting and audio content every time. A script for streaming the transcript is available in the git repository in the directory `debug_workers`.

6.2 Live Meeting Testing

We conducted several tests of the tool between the author and the supervisor and also and together with a group of network administrators from a local high school, using their work meeting as a testing ground for meetings with multiple active participants. We exploited the fact that their meetings contain a lot of named entities and technical wording, allowing us to test the ASR model to the limit. Based on the results, we formed a qualitative assessment of the tool usability and possible workflows. All the participants of our experiments were briefed and consented to their recordings being used in the evaluation.

6.2.1 Error Analysis

We found that most errors were committed by the ASR model when transcribing named entities. This was expected; many of the topics discussed in the test meetings required sufficient domain knowledge or were in different languages. Examples of transcription errors are listed in Table 6.1. These could probably be largely counteracted by using a more powerful version of the `Whisper` model; while testing, we resorted to the `small.en` variant due to speed and hardware constraints. Also, many of the errors originate in the non-native English of the

Example	Error explanation
Vojta: a different DHCP server named **care** so we can try it, I've never used it.	The discussed DHCP server is called Kea, not 'care'.
Fanda: like, adapt this towards **check** summarization, like, you just, like, one thing is swapping for **check** whisper, that's easy, and one thing is just, like, uploading a new model to...	The `Whisper` model misinterpreted bad pronunciation and did not recognize the word 'Czech' in context.

Table 6.1: Examples of errors committed by the ASR model

meeting participants with imperfect pronunciation and in bad quality of the participants' microphones. Upon inspecting the transcript, we updated the model to `medium.en`, increasing the transcript quality without sacrificing much speed.

As for summarization model errors, from our experiments, we conclude that the quality of the generated output is highly dependent on the quality and coherence of the provided transcript. We divide the committed errors into three main categories. Examples are provided in the list below:

- **Overgeneralization:** "*PARTICIPANT1 and PARTICIPANT2 discuss the implementation of a text editor.*" is a true statement for our test meeting, but it does not convey any important information that would be worth writing down, since the entirety of it was devoted to improving the editor.

- **Swapping or misinterpreting the actors of an action:** "*PARTICIPANT1 wants PARTICIPANT2 to finish the machinery before the end of this month so that if she switches the cables, she can just note it down and some scripts will fix it for him.*" IN the summary it is noted that PARTICIPANT1 wants PARTICIPANT2 to do something, but this is never mentioned in the transcript.

- **Errors due to lacking context:** "*PARTICIPANT1 needs to refer to some parts of the transcript for the minutes to get summarized. PARTICIPANT2 will double check the deadline for the bachelor thesis.*" In the transcript, the checked date was supposed to be the deadline of paper submissions, not for the thesis, but it was discussed in the same context as the bachelor thesis. From a longer context window, the correct reference could be deduced by the model and the error could be avoided.

Overall, it can be stated that the generated summary is good at capturing the main topic of a transcript segment, but it very often fails in determining who is the subject of an action and what is the object; much manual fixing is needed in that regard. The generated summary also does not necessarily correspond to a predetermined meeting agenda; it can therefore be difficult for the users to manipulate the model to focus on the content that is important to them. This is however natural, as the model cannot know the agenda in advance.

We also observed that summary quality was lower in the meeting captured with the administrator group. This could be due to a number of causes including

lower microphone quality, different non-native accents, less well-arranged transcript due to more participants or the fact that the summary model was finetuned on short non-technical conversations.

6.2.2 Feedback from Users

The testers reported that they appreciated the possibility of catching up with the meeting even on getting a quick pause. They did not yet feel comfortable trusting the tool for summarizing the whole meeting, noting the differing styles of a normal summary that mostly focuses on agreed-upon conclusions stemming from a previous agenda and the generated summary. They said they would appreciate the possibility of voice commands, which is something that we have not thought about when implementing the tool.

Users also reported that sometimes, the ordering of the utterances in transcript was behaving unintuitively. This is due to the fact that the utterances are sorted by their end times. This means that when a user has a longer monologue with others nodding and reacting to the partial information stated, the reactions appear in the transcript earlier than the transcribed monologue.

6.2.3 Suggested Workflow

We believe an efficient workflow is reliant on having multiple available participants in the meeting to supervise the transcript and summary points. We found it difficult to keep track of what was happening in the transcript and in the summary in only two people, as constant activity is required of both of the participants. However, the summary was of high quality, capturing the contents of the meeting well, and if the group of two users needs to produce a summary anyway, the tool definitely helps. In a larger group of users, only several of them are usually vocally active and the rest can contribute to correcting the transcript and the generated summary. When testing with the network administrators, we found this to be practical.

6.2.4 Limitations of Minuting from Just the Transcript

From the testing, it became apparent that the meeting minuting task can be seen as two distinct subtasks. One is the process of minuting itself; generating a comprehensive summary of what happened, what conclusions were reached, what are the tasks handed out to participants and so on. The second one is the process of comprehensively capturing what happened exactly in the meeting; who said what, what were the proper opinions and so on. We call that process *meeting timelining* and it is basically a compressed transcript.

We believe the current minuting approaches tend more towards timelining. We think this is partly due to the limited context the models are provided for generating summary bullet points, but it does not explain the entire phenomenon. From our point of view, the transcript is not sufficient to create a comprehensive meeting minute on its own. Meetings themselves do not occur in a vacuum, they are usually a regular ritual of a team at work which has some history, and participants already have some perspective on the issue being discussed from

before. Therefore, there is nearly always context in play that influences what the optimal minute should be, and it is often not mentioned by the participants. To illustrate our point, imagine a human annotator who has no history with a team conducting a meeting, but he or she is just asked to produce a summary. The quality of the output would be significantly lower in comparison to someone who is well aware of the context, and the resulting minutes would be substantially less useful.

From this, we conclude that minuting systems should ideally have access to the whole history of meetings once that becomes possible, and possibly the meeting outline written before a meeting is recorded. This way, the model can formulate the minutes from the same information an informed human scribe would have.

7. Conclusion

In the thesis, we explored the topic of user-assisted meeting minuting. We shortly introduced the necessary technologies of summarization and ASR along with their evolution. We then introduced our tool, *Minuteman*, to demonstrate possible interaction of users and summarization models.

We discussed the architecture of the tool, highlighting the possibility of extension to new summarization models or other usecases. We then described our attempts at our own summarization model finetuning. Although we were ultimately unsuccessful in surpassing the baseline, we identified some of the pitfalls that need to be avoided next time and highlighted the need for more elaborate datasets. For *Minuteman*, we therefore ended up falling back to the baseline model.

In the last chapter, we report on our experiments with the tool during live meetings. We identified that while the tool is useful, the output is not yet trustworthy to be acceptable right away; the users found the summaries not to be accurate enough yet. The interactive aspects however worked well and the expected workflow for correction of errors was successful. The users also positively evaluated the possibility of catching up with the discussion after a short lapse of focus or absence.

After examining the outputs, we concluded that there is still a large distinction between the outputs of a trained and context-aware human scribe and of the summary model. We believe this is both due to the limited context length of models and inherently missing information about the real-world context and previous participant experiences in the transcript. We suggest a solution as future work.

7.1 Future Work

Because our tool was intended as a proof of concept, there are many avenues for enhancement. We therefore divide them into three subsections, one focusing on possible improvements to the user interface, one focusing on summarization model development and the last one targeted at new meeting platforms.

7.1.1 User Interface

We believe that the user experience could be improved by visualizing relations between summary bullet points and transcript segments similar to the way it is done in the ALIGNMEET tool by Polák et al. [2022]. We think this would improve the users' understanding of the capabilities of the model and enable better cooperation.

We also believe query-based summarization could be of use, in a similar manner as was suggested by Zhong et al. [2021]. The users could write queries corresponding to the meeting outline and then have the corresponding summaries filled in on the same line. We hope this could help bridge the gap between meeting timelining and meeting minuting, as described in Section 6.2.4, as user intentions could be more clearly expressed.

Lastly, we think that better structuring of generated minutes could help with the output quality; training a model that would predict shorter bullet points and increasing the output density would help in that respect.

7.1.2 Summarization Model Development

We believe the Llama models show promise for summarization and minuting; therefore, we think finetuning them on the SAMSum and XSum datasets could improve the summary results by a large margin. Bigger models could be finetuned using low-rank adaptation training as proposed by Hu et al. [2021], shown in practice on the StackLLama model by Beeching et al. [2023].

Expanding to other languages would also be helpful; there, ChatGPT API[1] could be useful for summarizing, or a finetuned mBART Liu et al. [2020] model could be used.

7.1.3 New Meeting Platforms

As Jitsi Meet is not the most popular meeting platform, expanding to Google Meet, Teams and Zoom could bring more possible users to the tool for testing. Expanding to in-person meetings would also be possible; due to the message queue in the architecture, it would be possible to just replace the Javascript frontend with a microphone and a diarization tool, for example pyannote[2] by Bredin and Laurent [2021], to distinguish when different people are speaking.

[1] `openai.com/blog/chatgpt`
[2] `github.com/pyannote/pyannote-audio`

Bibliography

Yuvanesh Anand, Zach Nussbaum, Brandon Duderstadt, Benjamin Schmidt, and Andriy Mulyar. Gpt4all: Training An Assistant-Style Chatbot With Large Scale Data Distillation From Gpt-3.5-Turbo. https://github.com/nomic-ai/gpt4all, 2023.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural Machine Translation by Jointly Learning to Align and Translate. *arXiv preprint*, September 2014. doi: 10.48550/ARXIV.1409.0473. URL arxiv.org/abs/1409.0473.

Edward Beeching, Younes Belkada, Kashif Rasul, Lewis Tunstall, Leandro von Werra, Nazneen Rajani, and Nathan Lambert. StackLLaMA: An RL Fine-tuned LLaMA Model for Stack Exchange Question and Answering, 2023. URL https://huggingface.co/blog/stackllama. Accessed: 2023-07-10.

Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint*, 2020. doi: 10.48550/arXiv.2004.05150. URL https://arxiv.org/abs/2004.05150.

Hervé Bredin and Antoine Laurent. End-to-end Speaker Segmentation for Overlap-aware Resegmentation. In *Proc. Interspeech 2021*, Brno, Czech Republic, August 2021.

Sergey Brin and Lawrence Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Computer Networks*, 30:107–117, 1998. URL http://www-db.stanford.edu/~backrub/google.html.

Jaime Carbonell and Jade Goldstein. The Use of MMR, Diversity-Based Reranking for Reordering Documents and Producing Summaries. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '98, page 335–336, New York, NY, USA, 1998. Association for Computing Machinery. ISBN 1581130155. doi: 10.1145/290941.291025. URL https://doi.org/10.1145/290941.291025.

Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%* ChatGPT Quality, March 2023. URL https://lmsys.org/blog/2023-03-30-vicuna/. Accessed: 2023-04-20.

Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar, October 2014. Association for Computational Linguistics. doi: 10.3115/v1/D14-1179. URL https://aclanthology.org/D14-1179.

Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Eric Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, Albert Webson, Shixiang Shane Gu, Zhuyun Dai, Mirac Suzgun, Xinyun Chen, Aakanksha Chowdhery, Sharan Narang, Gaurav Mishra, Adams Yu, Vincent Zhao, Yanping Huang, Andrew Dai, Hongkun Yu, Slav Petrov, Ed H. Chi, Jeff Dean, Jacob Devlin, Adam Roberts, Denny Zhou, Quoc V. Le, and Jason Wei. Scaling Instruction-Finetuned Language Models. *arXiv preprint*, 2022. doi: 10.48550/arXiv.2210.11416. URL `https://arxiv.org/abs/2210.11416`.

Arman Cohan, Franck Dernoncourt, Doo Soon Kim, Trung Bui, Seokhwan Kim, Walter Chang, and Nazli Goharian. A Discourse-Aware Attention Model for Abstractive Summarization of Long Documents. *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, 2018. doi: 10.18653/v1/n18-2097. URL `http://dx.doi.org/10.18653/v1/n18-2097`.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *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, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL `https://aclanthology.org/N19-1423`.

Alexander R. Fabbri, Wojciech Kryściński, Bryan McCann, Caiming Xiong, Richard Socher, and Dragomir Radev. SummEval: Re-evaluating Summarization Evaluation. *Transactions of the Association for Computational Linguistics*, 9:391–409, 04 2021. ISSN 2307-387X. doi: 10.1162/tacl_a_00373. URL `https://doi.org/10.1162/tacl_a_00373`.

Nikhil Garg, Benoît Favre, Korbinian Riedhammer, and Dilek Hakkani-Tür. Clusterrank: a graph based method for meeting summarization. In *INTERSPEECH 2009, 10th Annual Conference of the International Speech Communication Association, Brighton, United Kingdom, September 6-10, 2009*, pages 1499–1502. ISCA, 2009. doi: 10.21437/Interspeech.2009-456. URL `https://doi.org/10.21437/Interspeech.2009-456`.

Gregory Gelly and Jean-Luc Gauvain. Optimization of RNN-Based Speech Activity Detection. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 26(3):646–656, 2018. doi: 10.1109/TASLP.2017.2769220.

Tirthankar Ghosal, Ondřej Bojar, Muskaan Singh, and Anja Nedoluzhko. Overview of the First Shared Task on Automatic Minuting (AutoMin) at Interspeech 2021. In *Proc. First Shared Task on Automatic Minuting at Interspeech 2021*, pages 1–25, 2021. doi: 10.21437/AutoMin.2021-1.

Tirthankar Ghosal, Ondřej Bojar, Marie Hledíková, Tom Kocmi, and Anja Nedoluzhko. Overview of the Second Shared Task on Automatic Minuting

(AutoMin) at INLG 2023. In *Proceedings of the 16th International Conference on Natural Language Generation: Generation Challenges*. Association for Computational Linguistics, September 2023.

Bogdan Gliwa, Iwona Mochol, Maciej Biesek, and Aleksander Wawer. SAMSum Corpus: A Human-annotated Dialogue Dataset for Abstractive Summarization. In *Proceedings of the 2nd Workshop on New Frontiers in Summarization*, pages 70–79, Hong Kong, China, November 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-5409. URL `https://www.aclweb.org/anthology/D19-5409`.

Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Comput.*, 9(8):1735–1780, nov 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL `https://doi.org/10.1162/neco.1997.9.8.1735`.

Matthew Honnibal, Ines Montani, Sofie Van Landeghem, and Adriane Boyd. spaCy: Industrial-strength Natural Language Processing in Python. URL `spacy.io`. Accessed: 2023-04-20.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-Rank Adaptation of Large Language Models. *arXiv preprint*, 2021. doi: 10.48550/arXiv.2106.09685. URL `https://doi.org/10.48550/arXiv.2106.09685`.

Adam Hájek. Automatic Text Summarization, 2021. URL `https://is.muni.cz/th/jsw6t/`. Bachelor's thesis at Masaryk University, Faculty of Informatics, Brno.

Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. Survey of Hallucination in Natural Language Generation. *ACM Comput. Surv.*, 55(12), mar 2023. ISSN 0360-0300. doi: 10.1145/3571730. URL `https://doi.org/10.1145/3571730`.

Taku Kudo and John Richardson. SentencePiece: A Simple and Language Independent Subword Tokenizer and Detokenizer for Neural Text Processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 66–71, Brussels, Belgium, November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-2012. URL `https://aclanthology.org/D18-2012`.

Lakshmi Prasanna Kumar and Arman Kabiri. Meeting Summarization: A Survey of the State of the Art. *arXiv preprint*, 2022. doi: 10.48550/arXiv.2212.08206. URL `https://doi.org/10.48550/arXiv.2212.08206`.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880, Online, July 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.703. URL `https://aclanthology.org/2020.acl-main.703`.

Chin-Yew Lin. ROUGE: A Package for Automatic Evaluation of Summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain, July 2004. Association for Computational Linguistics. URL https://aclanthology.org/W04-1013.

Yinhan Liu, Jiatao Gu, Naman Goyal, Xian Li, Sergey Edunov, Marjan Ghazvininejad, Mike Lewis, and Luke Zettlemoyer. Multilingual Denoising Pre-training for Neural Machine Translation. *Transactions of the Association for Computational Linguistics*, 8:726–742, 2020. doi: 10.1162/tacl_a_00343. URL https://aclanthology.org/2020.tacl-1.47.

I. McCowan, J. Carletta, W. Kraaij, S. Ashby, S. Bourban, M. Flynn, M. Guillemot, T. Hain, J. Kadlec, V. Karaiskos, M. Kronenthal, G. Lathoud, M. Lincoln, A. Lisowska, W. Post, Dennis Reidsma, and P. Wellner. The AMI Meeting Corpus. In L.P.J.J. Noldus, F Grieco, L.W.S. Loijens, and P.H. Zimmerman, editors, *Proceedings of Measuring Behavior 2005, 5th International Conference on Methods and Techniques in Behavioral Research*, pages 137–140. Noldus Information Technology, August 2005. ISBN 90-74821-71-5.

Rada Mihalcea and Paul Tarau. TextRank: Bringing Order into Text. In *Proceedings of the 2004 Conference on Empirical Methods in Natural Language Processing*, pages 404–411, Barcelona, Spain, July 2004. Association for Computational Linguistics. URL https://aclanthology.org/W04-3252.

Tomas Mikolov, Kai Chen, G.s Corrado, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space. *arXiv preprint*, 2013. doi: 10.48550/arXiv.1301.3781. URL https://doi.org/10.48550/arXiv.1301.3781.

Shashi Narayan, Shay B. Cohen, and Mirella Lapata. Don't Give Me the Details, Just the Summary! Topic-Aware Convolutional Neural Networks for Extreme Summarization. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1797–1807, Brussels, Belgium, October-November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1206. URL https://aclanthology.org/D18-1206.

Anna Nedoluzhko, Muskaan Singh, Marie Hledíková, Tirthankar Ghosal, and Ondřej Bojar. ELITR Minuting Corpus: A Novel Dataset for Automatic Minuting from Multi-Party Meetings in English and Czech. In *Proceedings of the Thirteenth Language Resources and Evaluation Conference*, pages 3174–3182, Marseille, France, June 2022. European Language Resources Association. URL https://aclanthology.org/2022.lrec-1.340.

Peter Polák, Muskaan Singh, Anna Nedoluzhko, and Ondřej Bojar. ALIGNMEET: A Comprehensive Tool for Meeting Annotation, Alignment, and Evaluation. In *Proceedings of the Thirteenth Language Resources and Evaluation Conference*, pages 1771–1779, Marseille, France, June 2022. European Language Resources Association. URL https://aclanthology.org/2022.lrec-1.188.

Dragomir R. Radev, Eduard Hovy, and Kathleen McKeown. Introduction to the Special Issue on Summarization. *Computational Linguistics*, 28(4):399–408,

2002. doi: 10.1162/089120102762671927. URL https://aclanthology.org/J02-4001.

Alec Radford and Karthik Narasimhan. Improving Language Understanding by Generative Pre-Training. 2018. URL cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf. Accessed: 2023-07-10.

Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. Robust Speech Recognition via Large-Scale Weak Supervision. *arXiv preprint*, 2022. doi: 10.48550/arXiv.2212.04356. URL https://doi.org/10.48550/arXiv.2212.04356.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *J. Mach. Learn. Res.*, 21(1), jan 2020. ISSN 1532-4435.

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ Questions for Machine Comprehension of Text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2383–2392, Austin, Texas, November 2016. Association for Computational Linguistics. doi: 10.18653/v1/D16-1264. URL https://aclanthology.org/D16-1264.

Steffen Schneider, Alexei Baevski, Ronan Collobert, and Michael Auli. wav2vec: Unsupervised Pre-Training for Speech Recognition. In *Proc. Interspeech 2019*, pages 3465–3469, 2019. doi: 10.21437/Interspeech.2019-1873.

Thomas Scialom, Sylvain Lamprier, Benjamin Piwowarski, and Jacopo Staiano. Answers Unite! Unsupervised Metrics for Reinforced Summarization Models. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3246–3256, Hong Kong, China, November 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1320. URL https://aclanthology.org/D19-1320.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1162. URL https://aclanthology.org/P16-1162.

Kartik Shinde, Nidhir Bhavsar, Aakash Bhatnagar, and Tirthankar Ghosal. Team ABC @ AutoMin 2021: Generating Readable Minutes with a BART-based Automatic Minuting Approach. In *Proc. First Shared Task on Automatic Minuting at Interspeech 2021*, pages 26–33, 2021. doi: 10.21437/AutoMin.2021-2.

Kartik Shinde, Tirthankar Ghosal, Muskaan Singh, and Ondrej Bojar. Automatic Minuting: A Pipeline Method for Generating Minutes. In *Pacific Asia Conference on Language, Information and Computation (PACLIC 36), In proceedings of ACL Anthology*, 2022.

Xinying Song, Alex Salcianu, Yang Song, Dave Dopson, and Denny Zhou. Fast WordPiece Tokenization. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 2089–2103, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.160. URL `https://aclanthology.org/2021.emnlp-main.160`.

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to Sequence Learning with Neural Networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, page 3104–3112, Cambridge, MA, USA, 2014. MIT Press.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. LLaMA: Open and Efficient Foundation Language Models. *arXiv preprint*, 2023. doi: 10.48550/arXiv.2302.13971. URL `https://doi.org/10.48550/arXiv.2302.13971`.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All You Need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc. ISBN 9781510860964. URL `https://arxiv.org/abs/1706.03762`.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 353–355, Brussels, Belgium, November 2018. Association for Computational Linguistics. doi: 10.18653/v1/W18-5446. URL `https://aclanthology.org/W18-5446`.

Atsuki Yamaguchi, Gaku Morio, Hiroaki Ozaki, Ken ichi Yokote, and Kenji Nagamatsu. Team Hitachi @ AutoMin 2021: Reference-free Automatic Minuting Pipeline with Argument Structure Construction over Topic-based Summarization. In *Proc. First Shared Task on Automatic Minuting at Interspeech 2021*, pages 41–48, 2021. doi: 10.21437/AutoMin.2021-4.

Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. BERTScore: Evaluating Text Generation with BERT. *arXiv preprint*, 2020. doi: 10.48550/arXiv.1904.09675. URL `https://doi.org/10.48550/arXiv.1904.09675`.

Yusen Zhang, Ansong Ni, Ziming Mao, Chen Henry Wu, Chenguang Zhu, Budhaditya Deb, Ahmed Awadallah, Dragomir Radev, and Rui Zhang. Summ^n: A

Multi-Stage Summarization Framework for Long Input Dialogues and Documents. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1592–1604, Dublin, Ireland, May 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.acl-long.112. URL `https://aclanthology.org/2022.acl-long.112`.

Ming Zhong, Da Yin, Tao Yu, Ahmad Zaidi, Mutethia Mutuma, Rahul Jha, Ahmed Hassan Awadallah, Asli Celikyilmaz, Yang Liu, Xipeng Qiu, and Dragomir Radev. QMSum: A New Benchmark for Query-based Multi-domain Meeting Summarization. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 5905–5921, Online, June 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.naacl-main.472. URL `https://aclanthology.org/2021.naacl-main.472`.

List of Figures

1.1 An illustration of the encoder-decoder architecture from Fig. 1 in Cho et al. [2014] 6
1.2 An example tokenization of the sentence *The fox doesn't arbitrarily jump over the dog* using the BERT uncased tokenizer 7
1.3 An illustration of the attention mechanism from Bahdanau et al. [2014] 9

2.1 Summary of a part of a meeting provided by the MeetGeek tool. Originally, the speaker did not express uncertainty, but the desire to view the transcript live. And he also did not talk about GPD, but GPT. The capitalization error for the Whisper model is an expected minor issue. 15

3.1 A screenshot of the full Minuteman online application in action. The left editor contains the transcript, the right contains the generated summaries. It is running in debug mode; notice the information prepended to each line. The user has just requested a summary of a selected segment. 17

4.1 A schema of the application architecture. Red components are running in the client's browser. 21

List of Tables

5.1 Iterative dataset statistics. The transcript, prepended minutes and target minutes columns give the average amount of words in the respective categories and the standard deviation. 28
5.2 Iterative dataset example 29
5.3 Comparison of the output metrics on ELITR development set . . 31

6.1 Examples of errors committed by the ASR model 34

List of Abbreviations

ASR, STT Automatic Speech Recognition, Speech-To-Text
VAD Voice Activity Detection
RNN Recurrent Neural Network
LED Longformer Encoder Decoder
ROUGE Recall Oriented Understudy for Gisting Evaluation, a summarization metric measuring n-gram overlap between a system and a reference summary
GPT Generative Pre-Training
BERT Bidirectional Encoder Representations from Transformers

A. Attached Data Files

The attachments are divided into two folders. The `minuteman` folder includes a snapshot of the git repository with the *Minuteman* application at commit hash `ea23564`. The structure of the `minuteman` repository is listed in Chapter 4. It is also available at `github.com/fkmjec/minuteman`. The `iterative_summarization` folder includes the dataset creation code for iterative summarization described in Chapter 5.

A.1 Pretrained Models

We do not include the pretrained summary models in the attachments due to their large size. Instead, they are downloadable using the `minuteman/torch_model_dir/download_models.sh` script from `vps.kmjec.cz/minuteman/`. We provide `lidiya-bart.mar` for the BART model and `flan-t5-base-samsum.mar` for the FLAN-T5 model, as well as the `silero_vad.onnx` model.

B. Development Documentation

In this appendix, we give a more detailed technical description of the system components as well as the important interfaces. We provide a guide for deploying the tool and also for including new summarization models.

B.1 Component Details

B.1.1 Audio Recording from Frontend Code

The source is split into six files located in `flask/static/src`. The dependencies are managed through `npm` and the Javascript code is bundled together for the browser using `webpack`. The individual files are:

- `index.js`: contains the entry point of the application. It creates a MeetingRecorder object and sets up the relevant callbacks so that the recorder connects when the user types in a JitSi room name.
- `ApiInterface.js`: contains the code for interfacing with the Flask API.
- `MeetingRecorder.js`: contains the `MeetingRecorder` class. The class is responsible for handling the meeting recording including the initial connection and reacting to participants joining or leaving. Upon connecting to a meeting, it creates an `AudioContext` object which is used for audio processing. It also holds an array of `TrackRecorder` objects which record the individual meeting participants.
- `TrackRecorder.js`: holds the `TrackRecorder` class, which handles recording of a single audio track. It manages the updating of participant names and the periodic sending of 1s chunks to the Flask API.
- `VoiceRecorder.js`: contains `VoiceRecorder`, an AudioWorklet handling the recording and decimation of audio. We record in 32-bit floats and decimate to 16000Hz.
- `ConfigUtils.js`: contains functions for updating config options in the interface when they are changed by another client.

The dependencies are specified in `package.json`. Webpack configuration is located in `webpack.config.json`.

B.1.2 Flask API

As stated in Chapter 4, the Flask API is the main entry point to the application for requests from the client. We decided to add it on top of the Express API in the Etherpad plugin because we believed we needed to work with Python machine learning libraries like `transformers`.

The API endpoints largely reflect the ones found in the Express API in the Etherpad plugin, as they forward requests. They are defined in `main.py`.

- POST `/new_minuting/`: creates a new session by generating a `session_id`, creating two editors in Etherpad and requesting session object creation from the Etherpad plugin. Redirects to the new session's page.

- POST `/minuting/<session_id>/transcribe`: handles requests to transcribe individual 1s chunks. The parameters are `author` and `recorder_id` strings and `chunk`, which is the array of floats to transcribe.

- GET `/minuting/<session_id>/get_state`: returns the current state of configuration for the session by requesting it from the Etherpad plugin.

- POST `/minuting/<session_id>/set_summ_model`: sets session model by forwarding the request to the Etherpad plugin. The only parameter is `summ_model`. First checks whether the model is available in TorchServe; if it is not, it returns a response with a 400 status code.

- POST `/minuting/<session_id>/set_chunk_len`: sets session summary chunk length by forwarding the request to the Etherpad plugin. The only parameter is `chunk_len`.

B.1.3 Transcription Worker

The transcription worker uses the `pika` library to interface with RabbitMQ, from which it receives audio chunks to transcribe and to which it also sends transcribed utterances. The worker listens to incoming audio chunks on a queue with identifier `audio_chunk_queue`. Each message from that queue must be a Python dictionary serialized by `pickle` with the following fields:

- `session_id`: what session the audio chunk belongs to.

- `recorder_id`: what user the chunk belongs to.

- `chunk`: a numpy array of 32-bit floats.

- `author`: the name of the user in the Jitsi meeting.

- `timestamp`: the timestamp of the start of the chunk.

The worker keeps the audio chunks in a `Transcripts` object which maintains the invariants mentioned in Section 4.6. The Silero VAD model used for speech recognition is run using `onnxruntime`.[1]

B.1.4 Etherpad Plugin

The Etherpad plugin has the most complex structure of all the components, as it keeps nearly all of the application state and handles the logic with respect to the editors. The code is located in `etherpad-lite/ep_minuteman`.

Etherpad plugins work by registering to hooks that the editor exposes and then reacting to those hooks when they are called. The hooks are divided into frontend and backend ones. The mapping between hooks and Javascript functions is kept in

[1] `github.com/microsoft/onnxruntime`

`ep.json`. All of the functions that are registered to backend hooks are located in `index.js`, which serves as the main entry point of the application. Dependencies are managed through `npm`, with the requirements specified in `package.json`.

The other most important files of the plugin are:

- `SummaryStore.js`: contains the classes necessary to hold the state for every session. The most important ones are `TranscriptChunker`, which handles the continuously appended transcript, and `SummarySession`, which keeps the state for the session.
- `ApiUtils.js`: contains utility functions for working with the Express API that Etherpad exposes. Mostly taken over from the `ep_comments_page` plugin.
- `ChangesetUtils.js`: includes functions for creating changesets for the Etherpad editors, allowing us to replace text in the middle of an editor based on specific identifiers.
- `TranscriptUtils.js`: exposes functions used for transcript chunk extraction from the transcript editor.
- `static/index.js`: contains functions registered to frontend hooks, mostly centered around handling the on-demand summarization.

Upon startup, the plugin conducts several steps. First, it creates a `SummaryStore` object for keeping records of created transcripts and summaries. Then, it connects to RabbitMQ and starts listening on a queue with name `transcript_queue` for incoming transcript chunks, expecting JSON data in the following format:

```
{
    "utterance": string,
    "session_id": string,
    "timestamp": timestamp,
    "seq": integer
}
```

When a transcript chunk needs to be summarized, the plugin sends a summary creation request to `summary_input_queue`. The request is in JSON and has the following form:

```
{
    "model": string,
    "session_id": string,
    "summary_seq": integer,
    "text": string,
    "user_edit": bool
}
```

Model is the identifier of the summarization model we want the response from. The `summary_seq` field is used for identifying the summary point in the editor. We need it because we are computing the summary points asynchronously, so we need a way to recognize the response when it comes back. The `user_edit`

field specifies if the request is due to a user editing the transcript and is mostly a remnant of past development.

The plugin then expects a response in JSON in the following format:

```
{
    "session_id": string,
    "summary_seq": integer,
    "summary_text": string
}
```

The response summary text then replaces the correct summary point in the summary editor.

The plugin also exposes an API for interfacing with it. The API definition is in `index.js` in function `expressCreateServer`. The plugin endpoints are:

- POST `/api/createSessionObject`: creates a new session object to hold state. Called when a new session is created.
- POST `/api/createSumm`: used by the frontend for creating a summary from a selected segment on demand.
- POST `/api/setChunkLen`: serves for setting the maximum chunk length for the transcript chunks that are sent for summarization.
- POST `/api/setSummModel`: serves for setting the current summarization model.
- POST `/api/setChunkLen`: sets the maximum transcript chunk length for summarization.
- GET `/api/sessionConfig`: returns the current session configuration, including whether the session is in debug mode, what model is selected and what is the current chunk length. Periodically polled by clients to synchronize between themselves.

B.1.5 Summarization Worker

The summarization worker is very similar in design to the transcription worker. It handles preprocessing of the transcript chunks before they are sent to TorchServe for summarizing, and the forwarding of completed summary points back to the Etherpad plugin. The preprocessing includes the deletion of stopwords and various remnants of imperfect transcription. It also serializes the requests to TorchServe. This means that we lose some benefits of paralellization that TorchServe provides. The API for interfacing with the summarization worker is already specified in Appendix B.1.4, therefore we only list the source files and their various responsibilities:

- `summarization_worker.py`: contains the entrypoint and the main logic of the application along with the RabbitMQ interfacing.
- `api_interface.py`: contains the functions for requesting summaries from TorchServe.

- `transcript.py`: provides the `Transcript` class which is used for loading actors and their utterances from transcripts.
- `preprocessing_utils.py`: provides the preprocessing functions for filtering of stopwords. Used from the `Transcript` class.

B.1.6 TorchServe

TorchServe is run as a docker container. In `torchserve/Dockerfile`, we install the required `transformers` dependency to be able to run the HuggingFace models. The model directory for the container is `torch_model_dir`.

B.2 Running the Tool

For running *Minuteman*, we provide two `docker compose` files. The configuration provided in `docker-compose-dev.yml` is targeted at running the tool locally without a GPU and the models in TorchServe. The configuration in `docker-compose.yml` is meant for running in a production environment, including the summarization models, and thus requires the GPU. In the `compose` files, the configuration for the application components is specified using environment variables. Each optionally configurable variable is highlighted with a comment beginning with an `OPTION` comment, while the variables that are necessary to set before running are highlighted with the `TODO` comment.

The build process was tested using Docker version 24.0.2 and `docker compose` version 2.19.1.

We first need to generate the Etherpad API key and download the VAD model by running `init_app_files.sh`. Then, we fill out the required `TODO` fields in the respective `compose` file; specifically, it means setting the `FLASK_SECRET_KEY` in the Flask container and `ADMIN_PASSWORD` in the Etherpad container.

B.2.1 Development

For development, the only thing we need to do now is run the following commands:

```
docker compose -f docker-compose-dev.yml build
docker compose -f docker-compose-dev.yml up
```

And after downloading the necessary containers and building the dependencies, we can open `localhost:7777` for the Minuteman title page. Note that on startup, starting RabbitMQ and Etherpad usually takes a while, so there will be error messages from the containers periodically trying to connect to the queue before RabbitMQ has had enough time to start. After the queue comes online, it will work normally.

B.2.2 Production

For running in production, we need to download the summarization models. To do that, we run `download_summ_models.sh` in the `torch_model_dir` directory. We also need to set an additional option, `ETHERPAD_URL`, to the domain we have

pointed to our Etherpad editors for iframing. After that, the commands to run are the same as for the development version, except with a different compose file.

```
docker compose -f docker-compose.yml build
docker compose -f docker-compose.yml up
```

We need to note here that some modern browsers disable audio processing for non-localhost domains if the connection is not over HTTPS. Therefore, it is needed to request a certificate for your domain, perhaps from the LetsEncrypt project.[2] Otherwise, the app will not work correctly.

B.3 Adding a New Model to TorchServe

When adding a new summarization model to Minuteman, it is necessary to create a model `.mar` archive for TorchServe to accept. The most common way of doing this is to download a model from HuggingFace and then creating the archive using `torch-model-archiver`.

To be able to create the archive, it is needed to install `torch-model-archiver`. It is available from PyPi[3]. Then, it is necessary to create a handler file similar to the example handler in `torchserve/bart_model_serve.py`. It is generally only needed to alter the constant with the model name, as this downloads the correct tokenizer on startup.

After that, the selected model can be downloaded from HuggingFace using the `transformers` library and the `torchserve/model_saver.py` script. This unfortunately requires PyTorch. To bypass this, it is possible to download the necessary files from selected the HuggingFace repository manually.

To prepare the `.mar` archive containing the model, we run the following command:

```
torch-model-archiver
    --model-name [MODEL_NAME]
    --version 1.0
    --serialized-file [pytorch_model.bin]
    --extra-files "[config.json],[generation_config.json]"
    --handler "[YOUR_HANDLER_FILE].py"
```

We replace `MODEL_NAME` with the desired name of the archive and the other varibles with the paths of your downloaded model components. After running the command, you will be left with a `.mar` archive with the model. For usage, we need to move the created archive to `torch_model_dir` so that it is visible by the TorchServe docker container. In the `docker-compose.yml` file, we then specify the model and its name in the command to run torchserve like this:

```
torchserve
    --start
    --model-store torch_model_dir/
    --models [NAME]=[MODEL_NAME].mar ...‘
```

[2]`letsencrypt.org`

[3]`pypi.org/project/torch-model-archiver/`