# Automatic Subtitle Generation

Project Report

2016 Speech Technology Project BSc./MSc.

Universität Hamburg

# Contents

# Chapter 1

# Goals and Motivation

In this year's Speech Technology project, our task was to write software that automatically generates subtitles for German lecture videos by using transcripts. We were, however, free to further specify this task and decide for ourselves what kind of subtitles we wanted to generate. As a target audience we chose both hearing impaired students that might want to watch lectures online and students that just wanted to read a transcript without listening to the lecture. What both these groups have in common when it comes to subtitles is, that the subtitles should be as comfortable to read as possible. Our criteria to work towards good readability on the screen were the removal of excess fillers like "äh" or "ähm", trying to fix more complicated mistakes in speech, denormalizing numbers, adding common abbreviations and punctuation and then creating chunks of text that are split syntactically well and have a good length for reading. Other aspects that weren't as important consequently were the accurate reproduction of the speakers words and alignment to the video.
The starting point from where we've been working are transcripts of lectures that have been created manually and look like automatic speech recognition output. That means we can assume everything that has been said is written down correctly, including unfinished words and repetitions. Punctuation and capitalization depending on punctuation are not included, other capitalization is. This is of course a bit of a special case, so the software might have to be modified slightly to work well on automatic speech recognition output.
Last year's project also worked on automated subtitling, albeit in English, so we reviewed and reused some of their results, particularly in our software structure. It's a pipeline of different modules that will be explained in the order they're traversed in the development part of this report. We conducted a study concerning the quality of our pipeline output, which will be discussed in the evaluation section.

# Chapter 2

# Previous Work

Most notably this years project relies on the work done in last years project. It was also focused on subtitles for videos, although with different input. The project focused on aligning a given transcript to a given video and segmenting the transcript into chunks which are appropriate for display as subtitles. They used transcripts written by humans, and isolated the alignment problem.

In contrast, this project uses speech recognition output instead of human written transcripts, which means that for generating subtitles, the alignment of the text to the video is not a problem, because speech recognition output already has timings attached to the words. However, the output of speech recognition is not the same as a traditional transcript; it is missing punctuation, capitalization of words and various other properties a human does on the fly.

The previous project established various technical basics which were reused. Transforming the input to video subtitles is done in multiple steps, therefore the software uses a pipeline structure. Last years project decided to use python to write the pipeline, because this makes it easy to also integrate other programs written in different languages into the pipeline, such as CMU Sphinx, MaryTTS or a Syntaxparser (See 3.7). Other basic software infrastructure was also reused, such as reading input files, writing output files and handling configuration files and command line switches, as these parts of a program are always necessary and mostly independent of what the software actually does.

# Chapter 3

# Development

## Software- and Datastructure

This years project is build upon the software which was created by the last Speech Technology project. The basic software structure is a pipeline of different modules which all interact with a central data structure. The software takes a config file as input, in which the user can specify which modules he wants to run. When the software is executed the config gets read and it runs all chosen modules one after another. Additional information which might be needed for individual modules is also passed on from the config file.
Since modules can't interact directly with each other a central data structure is needed to pass along the pipeline. This data structure is an instance of the *Data* class, which is structured in the following way:

```
{
    "words": [],
    "sentences": [],
    "chunks":[]
}
```

The *Data* class consists of three lists of Objects:

1. **words**:
   A *Word* object represents one individual word of the transcript. Every *Word* object has attributes like a string of the word itself, an index which shows the position of the word in the transcript and additional information that is added by the different modules.

```
{
    "original": "ja",
    "index": 0,
    "syntax_relation": "S",
```

```
    ...
}
```

2. **sentences**:
   A *Sentence* object represents one sentence of the transcript. All *Sentences* have the index of the first and last word, marking the beginning and end of the sentence, as attributes.

```
{
    "start_word": 0,
    "end_word": 34
}
```

3. **chunks**:
   A *Chunk* object represents one subtitle. All Chunks have the index of their start- and end-words as attributes. They also hold start- and end-timings for the subtitle if the input-transcript provided timing information.

```
{
    "start_word": 0,
    "end_word": 11,
    "start_time": 960,
    "end_time": 7420
}
```

During runtime each module reads from this *Data* object and modifies it. After a module has finished its job the *Data* object is updated and then passed along to the next module in the pipeline. The end-product of the pipeline is a list of *Chunk* objects that can be output as an `.srt` subtitle file.
The following sections will cover the individual modules of the pipeline.

## Input

The input module is always the beginning of the pipeline. It has the purpose of initialising the *Data* object with the data from the input-transcript.
The module splits the text from the transcript into single-word tokens and adds them with their index to the *Word*-list in the *Data* object.
If the input-transcript was already a subtitle file (`.eaf` or `.srt`), the module also takes the preexisting subtitles and adds them to the *Chunk*-list in the *Data* obejct. These Chunks will be overwritten later by the *chunkify* module (3.9, but the *fake_timings* module (3.3) uses the timing information that is provided by them.

## Fake Timings

The *fake_timings* module is responsible for creating timings for individual words. It could also be called "infer word timings".

Input files which are also used as complimentary subtitle files for videos, like `.eaf`, contain only partial information about the timing of each word in the subtitle, because there is only timing information about chunks of words. Because a subtitle is usually displayed for the entire time the text in the subtitle is said, this duration can be taken as the time it takes to say the words in the subtitle. One subtitle which is displayed for a few seconds is called a chunk.

Subtitle formats contain information how long the subtitle is displayed, which corresponds to the time it takes to say the words in the subtitle. If the chunking is reorganized later in the pipeline, it is necessary to have the timings for each individual word, so the timeframe in which a subtitle should be displayed can be calculated. The timing information from the input chunks can be used to estimate a time frame for each individual word in the chunk.

The most basic estimate would be to give each word in a chunk the same amount of time, but this is obviously not correct, as there are shorter and longer words. The module relies on `sayit.py`, a small script by Hal Daumé [4] that estimates how long it will take to say a word, based on various features extracted from the word. There are various features used, for example number of characters, number of vowels, number of vowel-consonant switches, if the words starts with a vowel or with a consonant or character counts of specific unicode characters. Which feature is used with which coefficient is decided via a regression problem. As input the author used 50k to 100k of the most frequent german words and synthesized them with MaryTTS [1] to determine how long it takes to say these words.

For each word in a chunk the duration is estimated with `sayit.py` and then multiplied by a factor to make the durations of all words in a chunk add up to the duration that was given by the input file for the complete chunk. A chunk is usually displayed for two to four seconds. Because this is a small time frame already, if it is combined with the output of `sayit.py`, the resulting time frames for each word are fairly close to the actual time the speaker takes to say each word. Additionally the tolerance for duration errors is quite high, because it does not matter if the subtitle switches a few milliseconds too late or too early. Overall the approach is a very effective solution to finding word timings considering the given requirements.

## Denormalize

The denormalization module converts numbers written in text to numbers consisting of digits, as in converting "twenty" to 20. It is common to use digits instead of words to represent numbers above twelve, also this conversion is intended to make the subtitles easier and faster to read, especially for longer numbers like years.

The way it works is inspired by the way normalization of numbers - converting digits to words - works in MaryTTS [1]. The biggest parts of a number are converted first, and then the rest gets denormalized again. For example in "neunzehn*hundert*vierundachtzig" the biggest part of the number is indicated by the "hundert" in the middle of the number. The part before it ("neunzehn") is then used as a multiplier for "hundert" (100) and the rest after it is recursively denormalized and added. This means after the first step the value is 1900 ($19 \times 100$) and "vierundachtzig" is still to be denormalized. There are multiple ways of writing the same number. The variant described above is common when the number refers to a year, but for lengths, weights and other metrics the same number would usually be written as "ein*tausend*neunhundertvierundachtzig" or just "*tausend*neunhundertvierundachtzig". The same mechanism can be used, only in this case the biggest part of the number is marked by the "tausend", so in the first step the value would be 1000 plus the denormalization of "neunhundertvierundachtzig".

To avoid converting false positives like "ein" or "sieb", which are number prefixes but not real numbers on their own, it is checked if the word contains at least one of a selection of essential substrings and after that it is matched with a regex to ensure no non-number parts are in the word.

For every next bigger part of a number, a new case has to be programmed manually. The implementation in this project works for numbers where the biggest part is "tausend", which means it currently works up to 999.999.

To improve the module further, multi token numbers could be implemented. Currently only one token is looked at and then it is decided if this token is a number or not. In actual speech recognition output a year like 1984 would look more like "neun zehn hundert vier und achtzig", so multiple tokens have to be looked at to get the complete number. Support for decimal numbers could also be implemented, though these are relatively rarely found in transcripts.

## Repairs

The task in the Repairs module is to detect instances where there are mistakes in the speech data and take actions to correct them. Examples of such mistakes can be as simple as the utterance of filler words like "ahm" or "äh" in the transcribed speech data. It can get as complex as interrupting a sentence midway to rephrase

that sentence completely.

It is important to note that the input is the data that has been through only one module, the Denormalize module and therefore there are no additional information for the data that are processed. The following subsections each describe the different stages in this module.

### Defining fillers and articles

The algorithms in the Repairs module depend mainly on the fillers. Consequently, a list of possible fillers is defined for use in the next stages. Most of the algorithms perform three general steps. Firstly, the filler is detected. Secondly the algorithm determines whether a repair can be applied. And lastly, the repair is applied and the filler is removed.

However, using only fillers would limit the number of algorithms that could be used on the data and therefore reduce the number of repairs and quality of the output data. This is where articles are important because more algorithms can be used given this new information. And as it was the case for the fillers, they are also provided to the algorithms as a list.

The two lists, fillers and articles, which are essential for the Repairs module are handcrafted and could be modified/extended in future work.

### Removal of excess fillers

The first algorithm is tasked with the removal of excess fillers. These fillers are not important to the other algorithms but will increase processing time and complexity. Therefore it makes sense to remove the excess fillers for efficiency in the very first procedure itself. The other leading reason to remove them is because the filler acts only as a marker to indicate the occurrence of a possible repair.

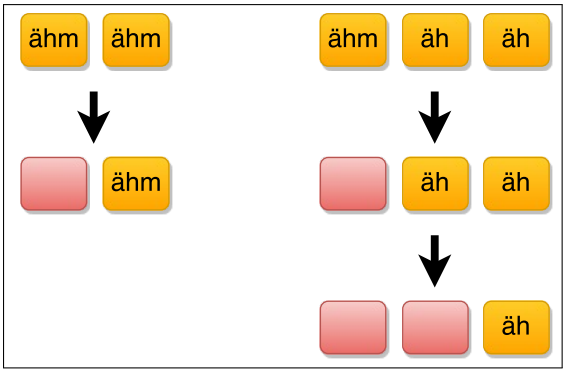


**Figure 3.1:** Removing Excess Fillers

### Repairing of articles

This algorithm caters for the correction of the speech data where any two consecutive non-deleted words are articles. This makes sense in case where the speaker utters the same article twice or corrects the article in the second occurrence. However, this problem cannot be solved by only looking at words at they do not contain any information about their use in the sentence. Additional information could be generated by POS-Tagging which is done later in the pipeline. Doing repairs before POS-Tagging improves the result of the tagging which is why the modules were ordered in this way. It might be interesting to investigate how tagging and repairs benefit each other in future work.

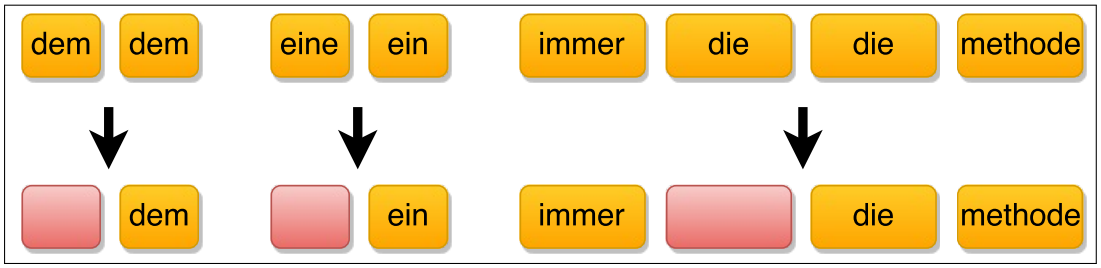


**Figure 3.2:** Repairing Articles

### Repairs where a filler is between identical words

In this stage, this specific algorithm detects cases where there are two similar words that are separated by a filler. Usually, this scenario occurs as a hesitation in speech and one of those reoccurring words and the filler must be removed. However, since the word surrounding the filler does not have to be specifically a non-article, it also filters out cases where there are two similar articles with a filler in between. This might lead to the same problem concerning the articles as discussed in 3.5.3.

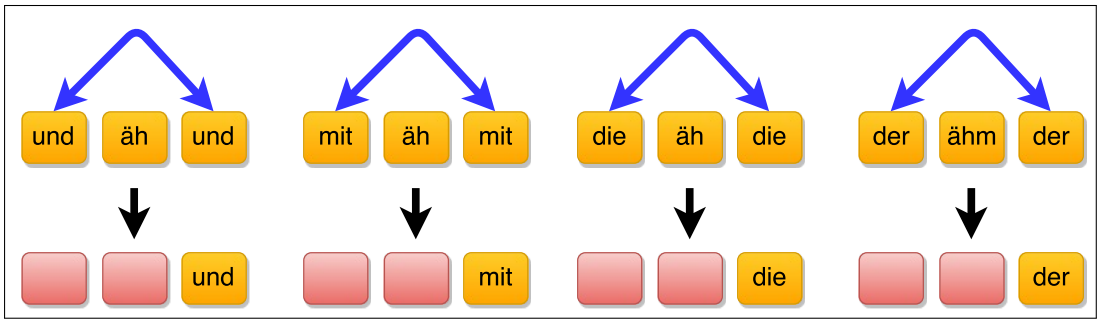


**Figure 3.3:** Repairing cases of filler between identical words

### Repairs where a filler is between articles

This part of the repair module deals with cases where there are two articles that are separated by a filler. Note that there is no possibility of having two similar articles separated by a filler since that would have been dealt with in the previous step. As with the previous stage, this scenario occurs when there is a hesitation in speech and the first article is redundant and the filler must be removed. Again, this might also, in some rare cases, lead to the same problem concerning the articles as discussed in 3.5.3.

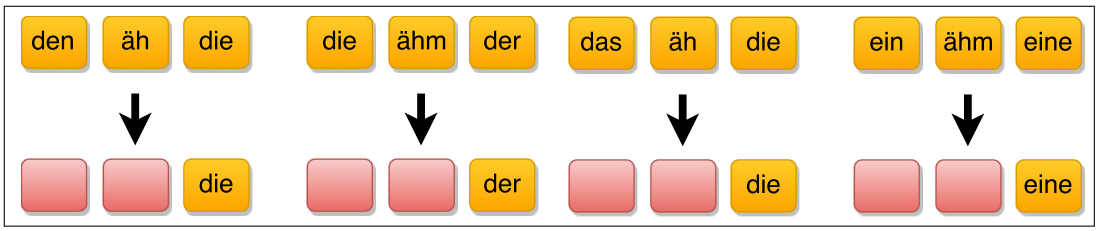


**Figure 3.4:** Repairing cases of filler between articles

### Repairs where a filler is in between 4 words

The last repair algorithm treats cases where there is a filler between four words. However, there is no simple or direct procedure to correct such a scenario. Some simple heuristics are used to find some cases that can be repaired. Using the format [Word1] [Word2] [Filler] [Word3] [Word4] for the data, there are three cases that are considered.

1. **[Word1] = [Word3] and [Word2] = [Word4]**
   This leads to the format [Word1] [Word2] [Filler] [Word1] [Word2] and therefore the first occurrences of [Word1] [Word2] along with the filler can be removed.

2. **[Word1] = [Word3] = "und"**
   This leads to the format [Word1] [Word2] [Filler] [Word1] [Word4] and therefore only the filler can be removed since there is a conjunction here.

3. **[Word1] = [Word3] and [Word2], [Word4]** $\in$ **Articles**
   This leads to the format [Word1] [Word2] [Filler] [Word1] [Word4] and therefore the first occurrence of [Word1], [Word2] and the filler can be removed since it is being repaired after the filler.

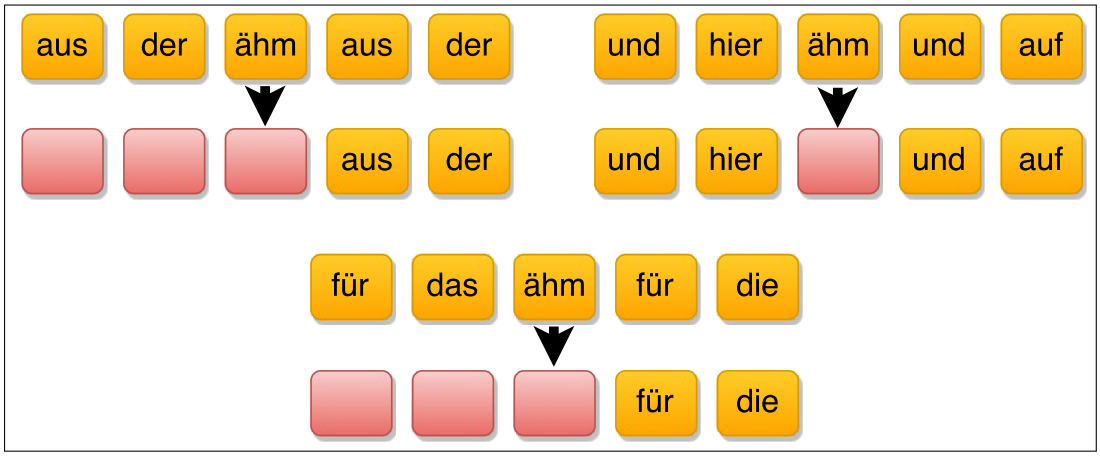


**Figure 3.5:** Repairing cases of filler between 4 words

### Tagging remaining fillers

In this last part of the Repairs module, all the repair algorithms have been applied on the data but there are still some fillers remaining. The task is to remove them from the subtitles. But in doing so, it is important to acknowledge that these fillers have not been removed when the repair algorithms have been used. The reason is that no possible repairs could be made using those algorithms. The optimal decision is to tag those remaining fillers and then they can either be removed completely for subtitles or shown as they can be helpful to explain speech hesitations in the video. As a possible future work, in-depth repairs using those tagged fillers could be made at a later stage after POS-tagging is done.

## Punctuation

For chunking the subtitles, we decided to continue using syntax information and punctuation like in last year's project. That, however, means generating punctuation which turned out to be a rather difficult task. We wanted to work only on the transcripts, not using audio information, and eventually selected two tools to work with: the CMU Sphinx post-processing framework and the punctuator 2. The data they both have been trained on were German Wikipedia articles without special characters and tagged in places with punctuation symbols. Additionally, Sphinx was trained to capitalize text. We trained both of them to insert only periods and commas, that means question or exclamation marks as well as other punctuation symbols won't be found. The training size and data sets were different for them both, but comparing the two of them on the same texts yielded very clear results as to which one worked better. Before discussion this, though, the tools themselves will be examined in greater detail.

### Sphinx Postprocessing Framework

The Sphinx Postprocessing Framework [7] is a tool that restores punctuation and capitalization for lower-case text without punctuation like the output of automatic speech recognizers. To achieve that it scores word sequences based on a language model, discards low scoring sequences and provides the best scoring sequence of the same length as the input text as a result. The language model used for the scoring has to be trained on a text with correct capitalization where periods and commas have been marked (<*PERIOD*> instead of . and <*COMMA*> instead of ,).

For this project language models were trained with SRILM [8] on parts of the German Wikipedia. Models were trained with different amounts of training data and evaluated on Wikipedia text which was not part of the training data. The results can be found in table 3.1.

**Table 3.1:** Performance of Sphinx PPF with different language models

| Wikipedia text | | Precision | Recall | F-Score |
|---|---|---|---|---|
| 1 MB | Period | **0.56** | 0.42 | **0.48** |
| | Comma | **0.13** | **0.22** | **0.16** |
| 10 MB | Period | 0.4 | **0.51** | 0.44 |
| | Comma | 0.06 | 0.10 | 0.08 |
| 100 MB | Period | 0.51 | 0.32 | 0.39 |
| | Comma | 0.07 | 0.14 | 0.09 |

Strangely enough increasing the training data did not improve the results. Because of the low scores for comma recovery we built new language models for periods only with the 1 and 10 megabyte training data. These language models performed

slightly worse (0.51/0.42/0.46 for 1 MB and 0.40/0.49/0.44 for 10 MB)[1] which suggests that comma recovery, poor as it is, still benefits the recovery of periods. In some cases the postprocessing framework puts a comma after every word in a sentence which is why commas restored by the sphinx postprocessing framework should be removed in the displayed subtitle. Furthermore period recovery is the more important task in punctuation recovery, because sentence boundaries are needed for part-of-speech tagging and subtitle chunking.

The results reported by the developers who trained and tested on texts from the Gutenberg project are similar for period recovery (0.57/0.55/0.56)[1] and much better for the recovery of commas (0.63/0.44/0.52)[1] which might be related to the difference between English and German rules for comma placement.

### punctatuor2.py

The punctuator 2 [9] a python tool that uses the deep learning library theano to train neural networks to insert punctuation into text. It can work with plain text, the mode we used in this project, or pause-annotated text. The network architecture it uses is called Bidirectional Long Short Term Memory recurrent network (BLSTM/RNN) which has been shown to work very well for speech processing. Its precessor, the punctuator worked similarly with LSTM recurrent networks and outperformed N-gram models on text and speech recognition output [10].
There are, of course, reasons why this kind of network structure is particularly well suited for speech processing in general and punctuation recovery in particular. First, the fact that the net is recurrent, that is that is that neurons in the neural net have conncetions to neurons that are in the same or even previous layers, means that previous inputs can influence the proccessing of following signals. As there are no bounds on these connections inside the neural net, recurrent nets aren't limited to taking in account only the preceding $N-1$ words in the same way other net architectures and $N$-gram models are [6]. This is ideal for speech processing as dependencies can span the whole sentence and therefore might not be processed properly otherwise. The LSTM architecture only reinforces this ability to draw from the input history as it helps the net "remember" long term dependencies better. It adds LSTM layers to a network that contain recurrently connected neurons with three different gates: input, output and forget. The forget gate enables the neuron to reset itself and forget the maybe no longer needed bit of input history, the other two gates multiply the input and output to the neuron. These layers save potentionally important dependencies and use them to influence new inputs traveling through the net via their recurrent connections. Using these LSTM nets bidirectionally means using two networks and one input sequence that is given to the first network in original order and backwards to the second net. They're

[1] precision/recall/f-score

connected to the same output layer, forming a BLSTM net. The advantage of using this structure is that now the network doesn't only have information about preceding input but also about the input still to follow [5]. Therefore, BLSTM networks are very good at taking dependencies over greater distances in account, which is important for recovering punctuation and other speech processing tasks.
There is no official data on how well the punctuator 2 does in comparison to its predecessor, but there is an online demo version (http://bark.phon.ioc.ee/punctuator) that worked remarkably well when we tried it. Combined with the net architecture and better documentation, it seemed like a good idea to use the punctuator 2. An additional argument was the training data size used for the online demo which was just 350MB. It seemed as though the relatively small amount of data needed to train a fairly well working model (about 50MB according to the creator) would make the training relatively fast as well.
To train a model, first the data has to be cleaned. In this case, it was Wikipedia articles that were stripped of all special characters and punctuation except for periods and commas. Then they were lower cased and commas and periods were replaced by ",COMMA" and ".PERIOD" tags, respectively. The data then has to be processed into pickle files before starting the training of a model. The output doesn't have periods and commas, but the same tags as the input. They're then replaced in the punctuation module that also restores the original punctuation. As the punctuator 2 doesn't capitalize text, we also added capitalization after periods in the module. The rest of the transcripts is already capitalized, so the only additional capitalization that is needed would be the one after a period. This way, the capitalization is consistent with the punctuation and in places with correctly set periods it's also correct.
We chose the suggested standard configuration of a hidden layer size of 256 neurons and a learning rate of 0.02. Training, however, wasn't as fast as we hoped and so out model is a bit smaller. Part of that was that the 25MB model we are using currently took about 11 days to train whereas Tilk told us to expect less than 20 hours for 50MB (he's using a high performance graphics processor, so that's most likely what makes the difference), the other was that the 25MB model worked well already. After training the model, we tested it on a piece of text from the Wikipedia that wasn't used as training data. This file was about 1MB in size, without empty lines and the like just like the training data, so the amount of characters should be around one million. The model performance was measured in a recall (the percentage of expected punctuation that was found), precision (the percentage of the found punctuation that was correct) and F-Score value (computed from the former: $F = 2 \cdot \frac{recall \cdot precision}{recall + precision}$) for periods and commas each. It could probably still be improved by more data, but with an F-Score of 0.71 for periods and 0.47 for commas, the punctuator model works relatively well on Wikipedia articles. Furthermore, the accuracy of this model is relatively high (0.81 precision on periods and 0.69

on commas), so that wrong periods that might irritate viewers as well as the later modules don't appear very often. In general, wrong periods and commas are still near the correct positions and are spaced out evenly along the text.
On lecture transcripts, the model makes different mistakes, but works similarly well as on Wikipedia article. An automated analysis isn't possible because of the lack of pre-existent punctuation, so we did a manual analysis of three shorter text fragments (about 2 minutes each) from three different speakers. As speakers tend to make long sentences and sometimes don't finish them properly, there have unsurprisingly been much more commas than in the Wikipedia texts. Sometimes commas are also placed where a periods might be more appropriate although the comma isn't necessarily wrong. This leads to very long sentences overall and misplaced periods at the end of such sentences. Still there were no comma or period sequences and even misplaced punctuation was at least close to where it should have been. With an F-Score of 0.57 for periods and 0.62 for commas and the subjective impression that the punctuation is helpful, the punctuator 2 recovers periods and commas well enough to give a good basis for the following syntax analysis and the chunking.

### Comparison

The syntax module 3.7 and the chunkify module 3.9 both rely on (correct) punctuation. To decide which punctuation tool should be used in the pipeline we tested the *Sphinx Postprocessing Framework* and the *punctuator2.py* on 1mb of wikipedia text, which was not part of the training data. As table 3.2 shows, punctuator2.py greatly outperformed the Sphinx Postprocessing Framework and consequently was used in the pipeline.

**Table 3.2:** Comparison of Sphinx PPF and punctuator2.py

| | | Precision | Recall | F-Score |
|---|---|---|---|---|
| Period | Sphinx PPF | 0.31 | 0.40 | 0.35 |
| | punctuator2.py | **0.81** | **0.64** | **0.71** |
| Comma | Sphinx PPF | 0.12 | 0.16 | 0.14 |
| | punctuator2.py | **0.69** | **0.35** | **0.47** |

## Syntax

The syntax module was taken over from last years project and didn't get modified much. The module is a python wrapper for the Turboparser, which is a dependency parser written in C++. It inputs every sentence from the transcript to the Turboparser, which does denpendency parsing and Part-of-Speech tagging on it.

The results of the parser get added to the *Word* objects in *Data*.

```
{
    "original": "Dank",
    "index": 2,
    "syntax_relation": "S",
    "pos_tag": "NN",
    "syntax_parent_local": 0
}
```

The syntax relations and Part-of-Speech tags are prerequisite for the chunkify module, which uses this information to make informed decisions about how to split the transcript into subtitles.

## Abbreviate

The abbreviate module is similar to the denormalization module in its idea. Like the denormalization module converts written out numbers into digits, the abbreviate module converts common expressions into their common abbreviations. An Example:

> "was natürlich in der Natur der Sache liegt näml. in der Natur der Rechtsfragen"

Here "nämlich" is abbreviate as "näml.", common abbreviations include "zum Beispiel" as "z.B.", "bezüglich" as "bzgl." or "und so weiter" as "usw.". The idea is that the subtitle becomes faster to read because "usw." is faster to read than "und so weiter" but conveys the same meaning. However, with less popular abbreviations to cognitive load of decoding the abbreviation can be quite high, which leads to an overall slower reading speed.

The implementation relies on a simple dictionary mapping complete words and multiple word expressions to their abbreviations. A dictionary mapping abbreviations to their complete expressions is included in MaryTTS and used for denormalization.

The selection of expressions which get abbreviated includes only common terms, and no abbreviations that are common only in a specific field like law studies, or medicine, as these would be difficult to understand for an outsider. For the implementation this meant that large parts of the MaryTTS dictionary were removed, as the dictionary was written to include all abbreviations which could possibly be encountered in a text, to guarantee that every abbreviation will be denormalized.

## Chunkify

The next step is determining which segments of subtitles ("chunks") to display at a given time. This means that the text generated by previous steps has to be split at certain points, and that it has to be decided what makes a splitting position or a chunk “proper” or “improper”. By chunkifying a couple of video snippets manually, we found that the quality of a chunk mainly depends on two criteria: The total length of the chunk and the position of the two splits a chunk consists of in the respective sentence’s syntactic structure. For each of these criteria, we agreed on a set of rules based on our experience with the snippets we worked on.

### Chunking rules

#### Penalty system

The quality of a split is represented by an integer called “penalty score” where a low penalty score represents a split that should be preferred over a split with a higher penalty score. For every possible split, the module computes two separate penalties, one for the syntax and one for the length. The two scores are added, and the sums are compared.

#### Optimal length of subtitles

During the discussion about the manually produced chunks, the thoughs about the maximal length of subtitles were rather different. The highest number of characters mentioned was 110, the lowest 60. For creating the java program’s basic structure, we agreed to work with a maximum length of 85 characters and the old, *syntaxbased* module has a preset maximum length of 68, both of which led to convincing results. In both approaches towards subtitling, the maximum length can be configured, but for the evaluation of the modules, we continued working with 68 (*syntaxbased*) and 85 (*java*) characters.
As for the minimum length of a chunk, the two modules differ due to the syntaxbased module working with two-lined subtitles. The preset minimum length in the syntaxbased module is 34, while the most useable results of the java module were generated with a minimum length of 22 characters. Similarly to the maximum length, both of these values can be configured.

#### Syntactical position of splits

The very first step of both chunking modules is splitting the text at every full stop. While the java module requires this step due to its internal structure, the syntaxbased module’s main motivation are performance issues.
After the text is split into sentences, the java module takes in one step further and

splits at every comma[2]. The syntaxbased output tends to produce splits after commas as well, but since the length rules are more restrictive than they are in the java module, it makes no sense to split at these spots forcibly.

We found that in german, it is almost always harmful to split directly before a verb or a noun - unless there is a comma (or a full stop) in front of it, most of those are required for the previous word to make sense. Thus, we decided to prefer splitting before pronomina, articles, conjunctions, prepositions and appositions. Among these, we created preferences by testing different settings and counting cases where they produced strange or improper splits on the basis of rather long texts (mostly written speech) and then choosing the setting with the most convincing results. This resulted in us agreeing on the following rules:

- 0 penalty points for splitting before a preposition
- 3 penalty points for splitting before an enumerating word (erstens, zweitens, einerseits, andererseits)
- 6 penalty points for splitting before a preposition or an apposition (except for "sich", a special case as this word frequently appears directly after one of the above)
- 9 penalty points for splitting before a pronomen, an article or "sich"
- 12 penalty points in any other case.

## Chunking modules

Since the chunkify modules of last year's project did not work when we started building our project and was additionally set up to work with english texts, we decided to program an additional module using java. When working on the modules, we quickly decided to throw two of the existing ones out and keep only the module *chunkify_syntaxbased* as well as *chunkify_java*.

### chunkify_syntaxbased

Out of the three modules provided by last year's project, the chunkify_syntaxbased module delivered the best results according to the report. Even though it was necessary to rewrite parts of the module in order to make it work on german texts (and in order to make it work at all in the first place), we decided to keep it and throw the other two modules, *chunkify_greedy* and *chunkify_naiv* out.

The first step taken by the module is splitting the text into sentences. Afterwards,

[2]The commas generated by the punctuation module were unreliable and therefore ignored.

every possible way of splitting a sentence is simulated and rated after the set of rules. The computed values are then compared, the lowest (best) one is chosen and used. Since the total amount of possible splittings grows exponentially with the number of words in a sentence (every gap between two words can either be a split or not, giving a total of 2 to the power of n possibilities), the performance decreases drastically on very long sentences. Since it is possible that abnormally long sentences are generated by the punctuation module, we added a cap so that long sentences are first shortened by performing a single split in the way described under chunkify_java and then processed as described above. An implementation of the Dijkstra algorithm finds optimal splits for a sentence based on the penalties.

**chunkify_java**

The java module works a little different. Similarly to its Python counterpart, it starts of splitting the text at full stops. Afterwards, however, it looks at each sentence individually and checks whether the sentence's length is within the predefined boundaries. Every sentence that is not will be split into two parts by computing the specific penalty rule of every possible split and then choosing the lowest (if two or more scores are equal, the most central one is chosen). After finishing this step for every sentence that is considered "too long", a script iterates over the chunks produced this way and generates a new list of "too long" chunks. The splitting process is then repeated recursively until every sentence is shorter than the defined maximum length.
In the next step, the program will try to make extremely short sentences (such as one-word-sentences that are generated by the punctuator module at some points) more conveniently readable. This happens by iterating over all chunks that are shorter than the minimum length and deciding whether the previous chunk is the better choice to be glued to it or whether the next one is: The better chunk is chosen by calculating which one of the resulting chunks (previous and this or this and next) has a length closer to the median between minimum and maximum length. If the resulting chunk is longer than the maximum length, it is once again split in the "optimal" way.
This will result in undoing the glueing in some cases, but since the function did no harm and was even helpful in some cases (such as lists of two or more subsequent one-word-sentences), we decided to keep it.

## Output

The output module is the last module of the pipeline. It generates the output files, which are a subtitle file (`.srt`) and a text file that contain the generated subtitles. The output module was taken over from last years project and not modified much.

The module goes through all *Chunk* objects in *Data* and writes them to an output file. The chosen subtitle file format for this software is `.srt`, which stands for "*SubRip text file format*".

```
1 1
2 00:00:09,560 --> 00:00:13,096
3 wenn man das hoert diese- diese
4 einzelnen Worte dann klingt es sehr
```

The `.srt` format is a very basic subtitle file format. Every subtitle has to have an index (line 1) followed by the duration of the subtitle (line 2) and lastly the contents of the subtitle (lines 3 - 4).

# Chapter 4

# Evaluation

## Framework

In order to evaluate our pipeline and get an understanding of how well our automatically generated subtitles are performing we set up an A/B-Test online based on the beaqlejs-Framework [2]. Beaqlejs originally is designed to create listening tests (its name is an acronym for "Browser based Evaluation of Audio Quality and comparative Listening Environment"), which was very useful in our case as we could realize a randomized Test-Order trivially utilizing the config.js-file. With a few modifications using html, javascript and php we were able to use it for videos and thus to evaluate the quality of our different approaches on chunking and error repairs. We used social media and word of mouth to get people from different backgrounds to help us evaluate our approaches.
With html5, video output on websites can be done purely in html and doesn't rely on Adobe Flash any more, however there are still some cross-browser issues when dealing with these new standards. Initially we were using just one video file for both A and B, as it decreases the load time of the website and it is possible to embed different subtitles with additional files. With `.srt` files being created by the pipeline, we just had to convert these to `.vtt` (or WebVTT, an acronym for Web Video Text Tracks) which is the W3C standard for displaying timed text in html5. When using Google Chrome though, there were some bugs using this technique, the subtitles would be shown multiple times if one were to switch between A and B videos time and again. Later on, we decided that due to these bugs we had to "burn" the subtitles into the video files (see 4.2).
The results collected by the website were submitted to our server using php and saved as `.txt`-files with the votings of the users for us to evaluate.

## Utilities

For A-B-Testing, whole subtitled lectures are impractical. Instead smaller snippets need to be used, which lead to a few requirements:

- A snippet should be 25 to 35 seconds in duration, the snippet should neither start nor end in the middle of a subtitle chunk. This should also be checked if two different chunkings are evaluated against each other, in which case the start and end of the snippets have to align in both chunking versions.
- If the snippet is found, the proper subsection of the video needs to be cut out and put into a seperate file. The same applies to the subtitle files for the A and B subtitles.
- later on it became clear that separate subtitle files and only one video file save disk space, but are not as reliable as two different video files where the subtitle is already edited into each frame of the video; so "burning in" the subtitles into the videos became an additional requirement.

To generate snippets with the given requirements a `snippetfinder.py` script was written, and later on a `subburn.sh` script, to account for two different video files with different subtitles already edited into the video file. The snippetfinder takes two output directories of the main pipeline, and a third directory where snippets will be generated, and then generates all the snippets it can find. The subburn script can then be run on the directory containing the snippets. After that, hand selected snippets can be taken for evaluation, as the number of generated proposed snippets is quite large.

## Study

We conducted a study with 14 participants to evaluate the repair module (3.5) and the two different chunking approaches (3.9). For both test sets we used nine snippets from three different lecturers. We swapped A and B for half of the trials and randomized the ordering to prevent order effects which can occur in a repeated measures design [3].

### Quantitative Results

Figure 4.1 shows that our participants seem to like B better than A in both tests, which means they prefer repairs and two-line subtitles created by the module *chunkify_syntaxbased*. This observation was confirmed by performing a chi-squared test. The null hypothesis that there is no difference between A and B can be rejected at the 0.01 level for both repairs ($\chi^2 = 38.707, p = 4.921 \cdot 10^{-10}$) and chunking

($\chi^2 = 18.256, p = 1.931 \cdot 10^{-5}$). Missing data points can be ignored since they do not change anything ($p = 4.109 \cdot 10^{-09}$ and $p = 0.00018$).

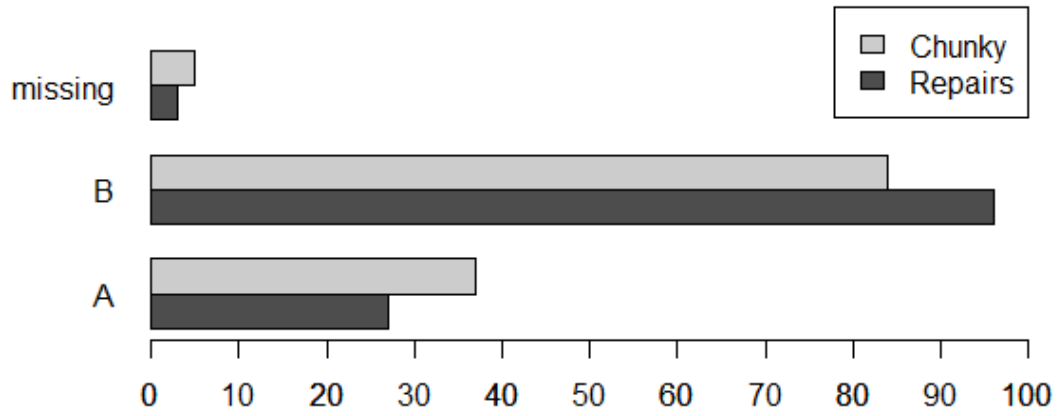


**Figure 4.1:** Results

## Qualitative Results

Many participants gave additional feedback in the comment boxes. The general opinion was, that fillers do not contain useful information and should be removed. One participant suggested using "..." for fillers which is easy to recognize as *"not important"*. Another comment was that for some pairs it was difficult to find differences which could be solved by doing a side-by-side comparison. Most participants commented that they like two-line subtitles better because of the shorter line width which led to shorter movements for the eyes. There was a case though where two-line subtitles obstructed the information presented on the slides, this should be avoided. Also typos and missing punctuation were criticized, spell-checking might be a good idea. In one case the chucks were split between "semi" and "professionelle" which was perceived as a bad split. This is mainly because the word "semiprofessionelle" is written as two words, which is a problem related to compound words which are common in the German language. Since a speech recognizer can produce both results this problem should be taken into account.

# Chapter 5

# Conclusion

In this report we presented the steps we took to implement software that automatically generates subtitles. Our goal was to focus on the readability of subtitles for people who are not able to or do not want to listen to a lecture. The main criteria for good subtitles we identified were *repairs of speech and fillers, chunking based on syntax and length, reading speed (denormalization)* and *punctuation*.

The pipeline we built consists of nine modules and uses json files as the central data structure to store information. The input module prepares the given transcript for the pipeline. Fake timings approximates the duration of words which is later used for chunking and aligning subtitles. Denormalization and abbreviations are similar but were split into two modules because other modules rely on different parts of their output. The repairs module removes fillers and repairs word repetitions which are unwanted in subtitles but often occur in speech because speakers tend to restructure their sentences while talking. The punctuation module tries to restore punctuation which is not part of the output of automatic speech recognizers but is needed for syntax parsing and chunking. We tested the Sphinx Postprocessing Framework and punctuator2.py and found that punctuator2.py performs significantly better. The syntax module is a turboparser wrapper which was taken over from last years project and is needed for chunking. The chunkify module splits the transcript based on punctuation, syntax, minimal and maximal acceptable subtitle length as well as the number of lines a subtitle is allowed to have. The output module saves the results of this process as an .srt subtitle file.

With respect to the goals we set in the beginning of the project, we evaluated the value of repairs and compared the two chunking approaches. The study was conducted online with beaqlejs which we modified so it was able to play videos. We performed a repeated measure A-B-test with randomized ordering and found that repairs significantly improve the quality of the generated subtitles ($p < 0.01$). The comparison of the two chunking approaches showed, that two-line subtitles were perceived as easier to read and thus rated significantly better ($p < 0.01$).

In addition to to the quantitative evaluation we allowed the participants to leave comments. From those we learned that people were irritated by the fillers and liked shorter eye movements which implies that short one-line or two-line subtitles are better. Participants also often criticized typos and missing punctuation.

With these results we are able to say that our ideas about good subtitles which we collected at the beginning of the project and the goals which we derived from those ideas were reasonable. The size of subtitles and repairs which we identified as most important for readability are critical. Then again we underestimated punctuation and spelling so these are two aspects that could be worked on. Furthermore in the evaluation process we discovered the problem of dealing with German compound words which, if handled properly, could improve the chunking of subtitles.

# Bibliography


[1] URL: http://mary.dfki.de/.

[2] URL: https://github.com/HSU-ANT/beaqlejs.

[3] Paul C Bates Cozby et al. *Methods in behavioral research*. 150.72 C6. 2012.

[4] Hal Daumé. *sayit.py*. URL: http://www.umiacs.umd.edu/~hal/sayit.py.

[5] Alex Graves and Jürgen Schmidhuber. "Framewise phoneme classification with bidirectional {LSTM} and other neural network architectures". In: *Neural Networks* 18.5–6 (2005). {IJCNN} 2005, pp. 602–610. ISSN: 0893-6080. DOI: http://dx.doi.org/10.1016/j.neunet.2005.06.042. URL: http://www.sciencedirect.com/science/article/pii/S0893608005001206.

[6] Tomas Mikolov et al. "Recurrent neural network based language model." In: *Interspeech*. Vol. 2. 2010, p. 3.

[7] *Postprocessing Framework*. URL: http://cmusphinx.sourceforge.net/wiki/postpframework.

[8] Andreas Stolcke et al. "SRILM-an extensible language modeling toolkit." In: *Interspeech*. Vol. 2002. 2002, p. 2002.

[9] Ottokar Tilk. *Github - ottokart/punctuator2.py*. URL: https://github.com/ottokart/punctuator2.

[10] Ottokar Tilk and Tanel Alumäe. "LSTM for Punctuation Restoration in Speech Transcripts". In: *Interspeech 2015*. Dresden, Germany, 2015.

# Appendix A

# Who wrote what

- Goals and Motivation - Theresa
- Previous Work - Felix
- Software- and Datastructure - Michael
- Input - Michael
- Fake Timings - Felix
- Denormalize - Felix
- Repairs - Khooshal
- Punctuation - Theresa
- Sphinx Postprocessing Framework - Kolja
- punctuator2.py - Theresa
- Comparison - Kolja
- Syntax - Michael
- Abbreviate - Felix
- Chunkify - Antonia, Tore
- Output - Michael
- Evaluation
- Framework - Jasper
- Utilities - Felix
- Study - Kolja
- Conclusion - Kolja

Additionally, Kolja, Michael and Felix went through the whole document to proof-read and fix formatting or spelling errors.