Bachelor Thesis

Language-Agnostic Subtitle Synchronization

September 30, 2019

Kaegi

Abstract

This thesis presents a two-step process to align subtitle files to movies using voice-activity-detection and a novel alignment algorithm. All common errors related to shifts, splits and frame-rate differences are corrected by the alignment algorithm in less than 30 seconds. The process is language-agnostic and can therefore even align subtitles to movies in different languages.

Using this method on a database of more than 20 movies and 100 subtitles files resulted in an error rate of 12% for subtitle-to-movie alignments and a 0% error rate for subtitle-to-subtitle alignments. This unprecedented combination of speed and accuracy makes the algorithm suitable for realtime post-processing of subtitles obtained from unreliable online databases.

Contents

Chapter 1

Introduction

Subtitles play a central role for enjoyable video consumption. They improve comprehension for fast or quiet speech, act as an audio substitute for hearing-impaired people and facilitate watching a movie in a foreign language. Although subtitle files can be easily obtained online for almost any movie, they are rarely synchronized with a movie file from a different source. Even a time offset of a few hundred milliseconds degrades the movie experience – an offset of more than a second make the subtitles unusable.

The most common approach to circumvent the problem is using an online database that maps the exact version of the movie file to a perfectly synchronized subtitle. However, this approach requires a constantly well-maintained database and even a slight change to the video renders this approach ineffective. A different area of research is using an incorrect subtitle file in conjunction with a synchronization algorithm. Although this concept is promising, state-of-the art algorithms can only align subtitles to movies in the same language or only correct a small set of error patterns.

This thesis presents a language-agnostic two-step process for real-time subtitle synchronization based on the second approach. Independence of the of spoken language is achieved by only extracting the intervals of speech in the first step, while ignoring the actual content of the audio. In the second step, a custom synchronization algorithm is then used to align the timestamps from the subtitle file to these intervals of speech. The algorithm can handle simple constant-offset errors as well as problems due to additional breaks or cuts in the subtitle file compared to the movie. With an additional post-processing step, even differences in playback speed can be corrected. Since the synchronization algorithm handles all common error patterns, almost all resulting subtitle files get perfectly synchronized to their respective movies.

The first section discusses the common errors and requirements for good subtitle synchronizations in more detail. The second section explains all steps of the voice-activity detection and synchronization algorithm. The last section provides fine tuning of the algorithms for real-world data and analyzes the accuracy of the corrected subtitles.

Chapter 2

Subtitle Synchronization

2.1 Process overview

The entire process of aligning an *input* subtitle to a *reference* video, audio or subtitle file consists of multiple steps. First, the intervals of speech have to be identified for the input file as well as the reference file. For audio and video data, voice-activity-detection is performed to generate intervals that likely contain speech. For subtitle files this simply consists of extracting the time spans with respect to the subtitle format. Secondly, both interval sequences are post-processed to become sorted and non-overlapping, which is a requirement for the alignment algorithm. In the last step, an alignment algorithm is performed on both interval sequence which returns time offset for the input intervals. Different algorithms can correct different patterns of synchronization errors (see section 2.2).

2.2 Patterns of synchronization errors

In the (easier) subtitle-to-subtitle alignment cases, a few problems frequently lead to incorrectly synchronized subtitles:

1. **Start Timestamp**: Some versions of a movie start a few milliseconds to a few seconds earlier or later compared to another version. Using a subtitle file on a movie that is synchronized to a different version leads to an undesirable delay.

2. **Director's Cut**: Two subtitle files can be synchronized to two differently edited versions of the same movie. Usually these versions only differ by a low number scenes. This introduces multiple lags of a few seconds each at several points during the movie.

3. **Advertisement Breaks**: Some subtitles are synchronized to a movie that contains advertisement breaks. This leads to breaks from a few seconds to several minutes in the subtitle file and requires frequent readjustment of the subtitle offset when watching a movie without advertisement breaks.

4. **Frame Rate Differences**: Common frame rates are 23.976 frames per second, 24 frames per seconds and 25 frames per second. If the playback speeds of the subtitle and video differ, the user has to frequently readjust the subtitle offset during the movie, as the subtitle file gets increasingly out of sync.

Apart from differences in presentation timestamps, the number of lines in different subtitle files for the same movie can vary greatly. This complicates finding the alignment. Some differences include:

- **Content Differences**: Subtitles may or may not include translations for visual information such as street signs. Some subtitles contain interjections, while they are omitted in others. Sometimes even extra information like the name of the subtitle creator is embedded inside a subtitle.
- **Split/Unified Lines**: Long sentences might be a single subtitle line in one file, and two different subtitle lines in another file.
- **Song Lyrics**: One of the subtitle file might include lyrics for songs while the other does not. This can account for several minutes of differing content in these subtitle files. A special case of this are songs during the film intro or the credits, as the resulting subtitle files might have severely different start and end timestamps.

If the reference is a video or audio file, the voice-activity detection introduces even more errors. Since the audio landscape is very complex in movies, the detection will often wrongly classify ambient sounds as speech and vice versa. This makes simple matching algorithms unsuitable for this kind of automatic subtitle synchronization.

2.3 Subtitle Style Guide

An interesting question for subtitle synchronization is what a defines a ”good subtitle” as this is a subjective measure. Netflix provides a subtitle style guide [1] which requires following:

- a subtitle event should have a length between 5/6 seconds to 7
- the subtitle event should start within 3 frames (125ms for 24 frames per second) of the audio
- the subtitle event should end within 3 frames after the the audio or up 12 frames (500ms for 24 frames per second) if this improves readability
- a subtitle event should not cross a shot change

2.4 Voice-Activity-Detection

A voice-activity-detection (VAD) module generally consists of audio feature extraction and a classifier. The reference implementation for this thesis uses the pre-trained WebRTC VAD [2], which is heavily optimized for speed. The WebRTC VAD was chosen as it is easy-to-use and available under an open source license.

This section summarizes its approach. The WebRTC calculates the energy on several frequency bands and then uses a Gaussian mixture model (GMM) to calculate probabilities of speech and non-speech, which are then combined into a single VAD decision. Although no explicit references to literature are included in the source code, similar methods have been presented in various papers [3][4]. More complex approaches using *Recurrent Neural Networks* as classifiers have been proposed for voice-activity-detection specifically in movies [5][6]. Unfortunately no pre-trained models have been published.

Filtering audio

Any input audio in the WebRTC VAD module is firstly downsampled to 8000 samples per second. This is done as the data of interest - human speech - occurs between 300Hz and 3400Hz [7]. To satisfy the Nyquist–Shannon sampling theorem, the actual sample rate has to be at least twice as high. A VAD decision is made for every 10 milliseconds of audio data, which corresponds to only 80 audio samples.

After downsampling, the audio is filtered to yield sample data on the following sub-bands:

- 80Hz - 250Hz
- 250Hz - 500Hz
- 500Hz - 1000Hz
- 1000Hz - 2000Hz
- 2000Hz - 3000Hz
- 3000Hz - 4000Hz

This is done by two all-pass filters, which shift the phase of the signal dependent on the frequency without changing the amplitude. By adding or subtracting the result of slightly different all-pass filters, constructive or destructive interference occurs dependent on the frequency. This way the 0 - 4000Hz band is split into a 0 - 2000Hz band and a 2000Hz - 4000Hz band. In this filtering step, only half the samples are retained for each sub-band as the bandwidth of the signal was also reduced by this factor.

This splitting filter is afterwards applied again on the sub bands to split the first band on 1000Hz and the second band on 3000Hz. Similarly the resulting 0 - 1000Hz band was processed to yield the first three of the six final frequency bands. The the 80Hz - 250Hz band is obtained by high-pass filtering the 0 - 250Hz.

This approach is very efficient as both the all-pass filters and the high pass filter only have to iterate over each input sample once and each iteration only requires a few additions, multiplications and integer shifts.

Classification

After generating the sub-bands, the logarithm of the energies on each band is calculated. If $x_{i,\_}$ is the sample data on the sub-band i, the logarithm of the energy on band i is given by

$$E_i = \log \sum_j x_{i,j}^2$$

Using the logarithm of the energy instead of the energy itself correlates with human hearing, where loudness is also perceived logarithmic with respect to the energy of a sound.

The classifier assumes that the logarithmic energies of speech and nonspeech on the sub-bands obey a Gaussian distribution. Two Gaussian distributions for each bands are assigned for nonspeech and speech. The probabilities for each speech and nonspeech of a Gaussian distribution with mean μ and standard deviation σ is given by

$$p(E_i|\mu,\sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \cdot e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

The total probability of speech (or nonspeech respectively) on a band is the sum of probabilities for both Gaussian distributions. Let $p_{j,speech}$ and $p_{j,nonspeech}$ be the total speech and nonspeech probabilities on band j and sw_j a spectral weighting factor. If the formula

$$\log \prod_{j=1}^{6} (p_{j,speech}/p_{j,nonspeech})^{sw_j}$$
$$= \sum_{j=1}^{6} \log(p_{j,speech}/p_{j,nonspeech}) \cdot sw_j$$

crosses a certain threshold, the 10 millisecond audio segment is assumed to contain speech.

In the last step the decision is smoothed for consecutive audio segments, meaning a few intermediate nonspeech segments are still counted as speech segments.

Additionally to the presented formulas, the WebRTC VAD also features an adaption of the parameters of the Gaussian distributions at runtime within predetermined bounds. A detailed derivation and analysis of these adjustment formulas can be found in [3].

2.5 Algorithms

To synchronize subtitles to a reference file, a measure of the quality of an alignment - its `score` - has to be defined. This requires explaining the the fundamental building blocks of alignments which are *time units* and *spans* and *offsets*.

- A *time unit* is a length of time that is used to define one discrete "step" for the algorithms. By using smaller time units the maximum reachable accuracy of the alignment increases - as it is expressed in time units - at the cost of efficiency or increased memory requirements. All common subtitle formats use milliseconds as time units so this value was also chosen as the default in the reference implementation (but can be easily changed). One millisecond is also less than the length of one video frame, which 16.7ms for a 60Hz video, and therefore provides enough accuracy for the resulting alignment.

- A *span* can be modeled as an half-open interval with integer start and end values. Spans can be converted to actual start and end times of subtitle lines using a conversion factor. For example, given a subtitle line from a subtitle file which starts at second 1 and ends at second 3 with together with a conversion factor of 2 milliseconds per time unit, the resulting span would be $[500, 1500)$. Non-integer values have to be rounded to the nearest integer values to create a valid span. Consecutive spans that have the same start and end value do *not* overlap. For example $[0, 10)$ and $[10, 20)$ do not overlap, while $[0, 11)$ and $[10, 20)$ do overlap with one time unit. All spans $[a_1, a_2)$ are assumed to be non-empty with $a_1 < a_2$.

- An *offset* is the difference in time units between of two points of time and is used in the context of shifting spans. For example shifting the span $[500, 1500)$ with an offset of 20 results in the span $[520, 1520)$.

A few additional definitions are needed to allow precisely defining the score of an alignment.

Definition 2.5.1. Given two time spans $a = [a_1, a_2)$ and $b = [b_1, b_2)$, the functions `start`, `end`, `length`, `overlap` and `iscore` are defined as:

$$
\begin{aligned}
\texttt{start}(a) &= a_1 \\
\texttt{end}(a) &= a_2 \\
\texttt{length}(a) &= a_2 - a_1 \\
\texttt{overlap}(a, b) &= \texttt{max}(\texttt{min}(a_2, b_2) - \texttt{max}(a_1, b_1), 0) \\
\texttt{iscore}(a, b) &= \frac{\texttt{overlap}(a, b)}{\texttt{min}(\texttt{length}(a), \texttt{length}(b))}
\end{aligned}
$$

The definition of `iscore` was chosen so that the graph of $f(\sigma) = \texttt{iscore}(a, b + \sigma)$ contains 5 linear segments with a the maximum value of exactly 1 (see Figure 2.1). The offsets where the the segments start and end can be easily calculated using `start` and `end` for a and b. These segments will be used as the basis for optimizations in alignment algorithms, as only the 5 segments have to be calculated to characterize the score for all offsets σ.

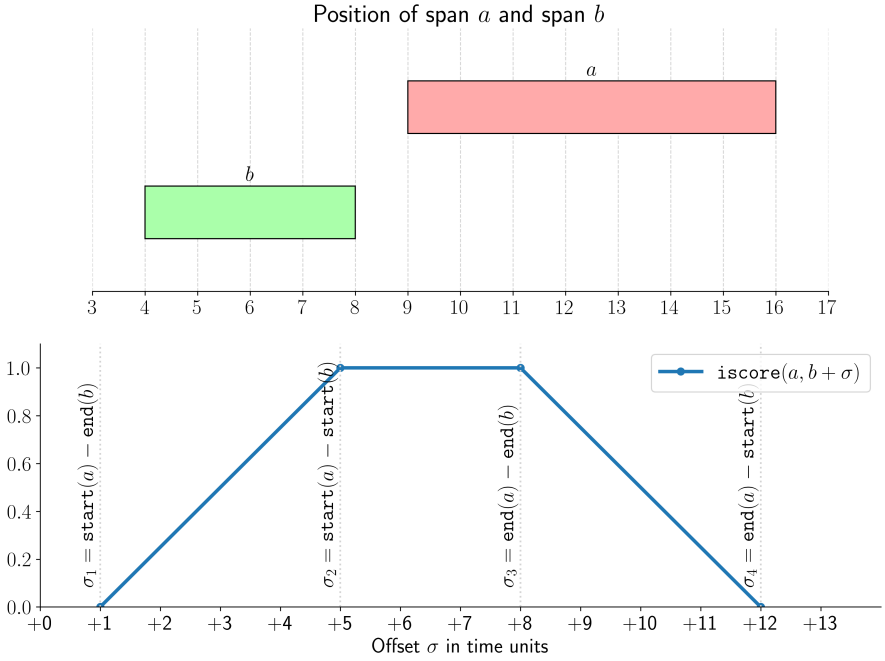

Figure 2.1: Visualizing the graph of $f(\sigma) = \texttt{iscore}(a, b + \sigma)$ for some a and b

Definition 2.5.2. Let $r = (r_1, r_2, \ldots, r_K)$ and $a = (a_1, a_2, \ldots, a_N)$ be two non-empty finite sequences of spans. r is called the *reference sequence* and a is called the *input sequence*. An *alignment* of the input sequence a to the reference sequence r is a sequence $\sigma = (\sigma_1, ..., \sigma_N)$ of offsets. The offsets σ_1 to σ_n represent the number of time units each input sequence span is shifted. $a + \sigma$ is added component-wise $(a_1 + \sigma_1, \ldots, a_n + \sigma_n)$.

The symbols r, a, K and N will be used with the given semantics throughout the thesis. σ will either be a sequence of offsets or a single offset depending on the context.

Definition 2.5.3. A sequence of spans $c = (c_1, \ldots, c_M)$ is called *valid* if $\texttt{end}(c_1) \leq \texttt{start}(c_2)$, $\texttt{end}(c_2) \leq \texttt{start}(c_3)$, ..., $\texttt{end}(c_{M-1}) \leq \texttt{start}(c_M)$ is satisfied. This is equivalent with requiring the spans in c to be sorted by their start times and to be non-overlapping.

By assuming r and a to be valid, various statements can be derived which form the basis of the alignment algorithms. No algorithm is allowed to reorder the input sequence span; $a + \sigma$, where σ is the resulting alignment, is always valid again.

Subtitle files as well as voice activity data will almost exclusively produce non-intersecting intervals. If this is not the case, a pre-processing of the subtitle files is done to yield valid sequences r and sequences a.

Definition 2.5.4. The *number of splits* of the alignment $\texttt{splits}(\sigma)$ is defined as

$$\texttt{splits}(\sigma) = \sum_{n=1}^{N-1} \begin{cases} 1 & \text{if } \sigma_n \neq \sigma_{n+1} \\ 0 & \text{if } \sigma_n = \sigma_{n+1} \end{cases}$$

By fixing the number of splits, the "degrees of freedom" can be limited. A low number of splits results in large segments of input intervals moved by the same offset. On the other hand, a large number of splits allows for many intervals to be placed individually, independently of the offset of preceding and subsequent intervals as long as they remain in the same order and are non-overlapping.

Definition 2.5.5. The similarity of reference interval to an input interval is expressed with the weighting function $w : \{1, \ldots, K\} \times \{1, \ldots, N\} \to \mathbb{R}_{\geq 0}$.

The weighting function w used by the reference implementation is

$$w(k, n) = \frac{\min(\texttt{length}(r_k), \texttt{length}(a_n))}{\max(\texttt{length}(r_k), \texttt{length}(a_n))}$$

This function yields the weight 1 for equally long spans and lower scores the more unequal the ratio between the respective lengths is. To increase or diminish this effect, an additional parameter s could be introduced with $w_s(k, n) = (\frac{\min(\texttt{length}(r_k), \texttt{length}(a_n))}{\max(\texttt{length}(r_k), \texttt{length}(a_n))})^s$.

Definition 2.5.6. The parameter p is called the *split penalty*. The *score of the alignment* σ is defined as:

$$\texttt{score}(r, a, \sigma, w, p) = \sum_{n=1}^{N} \sum_{k=1}^{K} \texttt{iscore}(r_k, a_n + \sigma_n) \cdot w(k, n) - p \cdot \texttt{splits}(\sigma)$$

Given r, a, w and p where r and a are valid, an optimal alignment σ^* is an alignment that satisfies

$$\texttt{score}(r, a, \sigma^*, w, p) = \max_{\sigma \text{ where } a+\sigma \text{ is valid}} (\texttt{score}(r, a, \sigma, w, p))$$

To illustrate how this scoring function works, assume the split penalty p is zero. The optimal alignment will then move all input intervals completely independently. Each input interval is moved to a position that maximizes the intersection length (high value of $\texttt{iscore}(r_k, a_n + \sigma_n)$) with reference intervals that have a high similarity (high value of $w(k, n)$). There might be many combinations of reference intervals and input intervals that do not intersect and therefore do not contribute to the score.

Moving all input intervals independently is often not desirable since only local information is used. For example, if reference interval sequence contains additional spans, a near input span might be incorrectly associated with one these additional reference spans.

To make use of global information and exploiting the coherence between consecutive input intervals, unnecessary splits need to be discouraged. This is why $p \cdot \texttt{splits}(\sigma)$ influences the score negatively. Every split then has to improve the first part of the score by at least a value of p, otherwise the additional split does not yield a more optimal alignment.

If the alignment does not contain any splits, it is called a *no-split alignment*. A simpler scoring function can be used in this case.

Definition 2.5.7. Given valid sequences of spans r and a, the weighting function w and a single offset σ, let $\texttt{nosplit\_score}(r, a, \sigma, w)$ be defined as

$$\begin{aligned}\texttt{nosplit\_score}(r, a, \sigma, w) :&= \texttt{score}(r, a, (\sigma)_{n=1}^{N}, w, 0) \\ &= \sum_{n=1}^{N}\sum_{k=1}^{K} \texttt{iscore}(r_k, a_n + \sigma_n) \cdot w(k, n)\end{aligned}$$

For the runtime complexity of the algorithms the total length of time units for each subtitle plays an important role. These are named T_r and T_a.

Definition 2.5.8. Let T_r and T_a be defined as

$$\begin{aligned}T_r &= \texttt{end}(r_K) - \texttt{start}(r_1) \\ T_a &= \texttt{end}(a_N) - \texttt{start}(a_1)\end{aligned}$$

The last values that have to be defined are the minimum offset σ_{min} and maximum offset σ_{max}. The smallest sensible value for an offset σ is where $a+\sigma_{min}$ is completely before r. This is the case if $\texttt{end}(a_N + \sigma_{min}) = \texttt{start}(r_1)$. On the other hand the largest useful value for σ is when $a + \sigma_{max}$ is behind r, resulting in $\texttt{start}(a_1 + \sigma_{max}) = \texttt{end}(r_K)$. This results in the following definition:

Definition 2.5.9.

$$\begin{aligned}\sigma_{min} &= \texttt{start}(r_1) - \texttt{end}(a_N) \\ \sigma_{max} &= \texttt{end}(r_K) - \texttt{start}(a_1)\end{aligned}$$

The total number of time units between σ_{max} and σ_{min} is then

$$\begin{aligned}\sigma_{max} - \sigma_{min} &= (\texttt{end}(r_K) - \texttt{start}(a_1)) - (\texttt{start}(r_1) - \texttt{end}(a_N)) \\ &= (\texttt{end}(r_K) - \texttt{start}(r_1)) + (\texttt{end}(a_N) - \texttt{start}(a_1)) \\ &= T_r + T_a\end{aligned}$$

2.5.1 Preparation of subtitle spans

Subtitle files might contain lines that overlap (i.e. characters speaking at the same time), unordered subtitle lines or degenerate lines time stamps. This can be fixed by first sorting the spans by their start time, repairing or discarding degenerate spans and then combining consecutive overlapping spans. A mapping from the original spans to the corrected spans has to be created in order transfer alignments from the corrected spans back to the original subtitle spans.

Algorithm 1 Transforming arbitrary span sequences into valid span sequences

Input: A arbitrary sequence of (degenerate) subtitle spans $x = (x_1, \ldots, x_{M_x})$
Output: A valid sequence of spans $a = (a_1, \ldots, a_N)$ where $N \leq M$ and a mapping $m : \{1, \ldots, M\} \to \{1, \ldots, N\}$ from the spans in x to their representative in a

$ys \leftarrow []$
for i from 1 to M **do** ▷ (Save original index before sorting by start time)
    $ys[i] \leftarrow (x[i], i)$
end for

$ys \leftarrow \texttt{sort}(ys, \text{by} = \lambda(y, i) : \texttt{start}(y))$ ▷ Sort lines by start time

$zs = []$
$m_{xz} = []$
$zi = 1$
for (y, xi) in ys **do** ▷ Discard zero-length lines
    $m_{xz}[xi] = zi$ ▷ Map xith entry of x to zith entry of zs

    if $\texttt{start}(y) < \texttt{end}(y)$ **then** ▷ Sort $\texttt{start}(y)$ in front of $\texttt{end}(y)$
        $zs[zi] = [\texttt{start}(y), \texttt{end}(y)]$
        $zi = zi + 1$
    else if $\texttt{end}(y) < \texttt{start}(y)$ **then**
        $zs[zi] = [\texttt{end}(y), \texttt{start}(y)]$
        $zi = zi + 1$
    end if ▷ Ignore case $\texttt{start}(y) == \texttt{end}(y)$
end for

if $j = 1$ **then return** *error* **end if** ▷ If no non-zero-length span was found, abort.

$a \leftarrow []$
$m_{za} \leftarrow []$
$n \leftarrow 1$
$zi = 1$
for z in zs **do** ▷ Combine intersecting spans
    $m_{za}[zi] = n$ ▷ Map zith entry of z to nth entry of a
    $zi \leftarrow zi + 1$

    if $a \neq []$ and $\texttt{start}(z) < \texttt{end}(a[l-1])$ **then** ▷ Do z and $a[n-1]$ overlap?
        $new\_end = \max(\texttt{end}(a[n-1]), \texttt{end}(z))$ ▷ Extend $a[n-1]$ to include z.
        $a[n-1] = [\texttt{start}(a[n-1]), new\_end]$
    else
        $a[n] = z$;
        $n = n + 1$
    end if
end for

$m_{xa} = []$
for i in 1 to M **do** ▷ Create mapping from x to a
    $m_{xa}[i] = m_{za}[m_{xz}[i]]$

45: **end for**
46:
47: **return** (a and m_{xa})

Combining the intersecting lines and fixing degenerate spans is done in linear time $O(M)$. Since most subtitles are already sorted, *Timsort* can be used with a best case performance of $O(M)$ for sorted arrays and worst case complexity of $O(M \log M)$.

The reference subtitle can be preprocessed in the same way, although the mapping function is not needed.

After transforming the input span sequence and reference span sequence into a valid ones, an alignment algorithms computes an alignment $(\sigma_1, \ldots, \sigma_N)$ for the sequence a. The alignments for the original spans x from the subtitle file can be obtained with the mapping function m as $(\sigma_{m(1)}, \ldots, \sigma_{m(M)})$.

2.5.2 Calculating `score` efficiently

Calculating of $\texttt{score}(r, a, \sigma, w)$ as given by the definition calls $\texttt{iscore}(r_k, a_n + \sigma_n) \cdot w(k, n)$ exactly $N \cdot K$ times and therefore has a run time complexity of $O(N \cdot K)$.

By exploiting the properties of a valid sequence r and $a + \sigma$ the complexity can be reduced to $O(N + K)$ without needing additional memory. The important observation is that only a small number of $\texttt{iscore}(r_k, a_n + \sigma_n)$ is non-zero, because only very few spans in r and $a + \sigma$ overlap.

Algorithm 2 Calculate the `score` efficiently

Input: r and a, σ and split penalty p where r and $a + \sigma$ are valid sequences
Output: $\texttt{score}(r, a, \sigma, w, p)$

1: $k \leftarrow 1$
2: $n \leftarrow 1$
3: $sum \leftarrow 0$
4: **while** $k \leq K$ and $n \leq N$ **do**
5: $\quad sum \leftarrow sum + \texttt{iscore}(r_k, a_n + \sigma_n) \cdot w(k, n)$
6:
7: $\quad$ // invariant 1: sum is here $score((r_1, \ldots, r_k), (a_1, \ldots, a_n), (\sigma_1, \ldots, \sigma_n), w, 0)$
8: $\quad$ // invariant 2: $\texttt{end}(r_{k-1}) \leq \texttt{end}(a_n + \sigma_n)$ if $k > 1$
9: $\quad$ // invariant 3: $\texttt{end}(a_{n-1} + \sigma_{n-1}) \leq \texttt{end}(r_k)$ if $n > 1$
10:
11: $\quad$ **if** $\texttt{end}(r_k) \leq \texttt{end}(a_n + \sigma_n)$ **then**
12: $\quad\quad k \leftarrow k + 1$
13: $\quad$ **else**
14: $\quad\quad n \leftarrow n + 1$
15: $\quad$ **end if**
16: **end while** ▷ sum is now $\texttt{score}((r_1, \ldots, r_K), (a_1, \ldots, a_N), (\sigma_1, \ldots, \sigma_N), w, 0)$
17:
18:
19: **for** n in $1, \ldots, N - 1$ **do** ▷ Count the number of splits
20: $\quad$ **if** $\sigma_n \neq \sigma_{n+1}$ **then** $sum \leftarrow sum - p$ **end if**
21: **end for**
22:
23: **return** sum

It is easy to see why the algorithm terminates. The first loop increments either k or n and the **while** condition requires $k + n \leq K + N$. The second loop terminates after exactly $N - 1$ iterations.

Theorem 2.5.1. *The algorithm is correct.*

Proof. First we prove inductively that the three invariants do indeed hold. It is easy to see that for the first iteration with $k = 1$ and $n = 1$ all invariants are satisfied.

Now assume we already know $\texttt{score}((r_1, \ldots, r_k), (a_1, \ldots, a_n), (\sigma_1, \ldots, \sigma_n), w, 0)$ for a given n and k and all invariants are satisfied.

Case 1: $\texttt{end}(r_k) \leq \texttt{end}(a_n + \sigma_n)$. After line 12 `invariant 2`

$$\texttt{end}(r_{k-1}) \leq \texttt{end}(a_n + \sigma_n) \text{ if } k > 1$$

holds as it is simply the condition on the if-branch with a shifted index `invariant 3`

$$\texttt{end}(a_{n-1} + \sigma_{n-1}) \leq \texttt{end}(r_k) \text{ if } n > 1$$

still holds, because of $\texttt{end}(r_k) \leq \texttt{end}(r_{k+1})$ the right side can only increase. $\texttt{end}(r_k) \leq \texttt{start}(r_{k+1})$ together with `invariant 3` we obtain $\texttt{end}(a_{k-1} + \sigma_{k-1}) \leq \texttt{start}(r_{k+1})$. This means that r_{k+1} can not overlap with $a_{k-1} + \sigma_{k-1}$ or any preceding spans.

As a consequence it suffices to calculate the `iscore` between r_{k+1} and $a_n + \sigma_n$ when going from $(r_1, \ldots, r_k)$ to $(r_1, \ldots, r_{k+1})$:

$$\begin{aligned}&\texttt{score}((r_1, \ldots, r_{k+1}), (a_1, \ldots, a_n), (\sigma_1, \ldots, \sigma_n), w, 0)\\ &= \texttt{score}((r_1, \ldots, r_k), (a_1, \ldots, a_n), (\sigma_1, \ldots, \sigma_n), w, 0) + \texttt{iscore}(r_{k+1}, a_n + \sigma_n)\end{aligned}$$

This means that in the next iteration of the loop, `invariant 1` is satisfied as well.

If $k = K$, the loop terminates with

$$sum = \texttt{score}((r_1, \ldots, r_K), (a_1, \ldots, a_n), (\sigma_1, \ldots, \sigma_n), w, 0)$$

No subsequent spans $a_{n+1} + \sigma_{n+1}$ to $a_N + \sigma_N$ can overlap with r_K (or preceding reference spans), because $\texttt{end}(r_K) \leq \texttt{end}(a_n + \sigma_n)$ and $a + \sigma$ is a valid sequence.

It follows for the loop termination:

$$\begin{aligned}&sum\\ &= \texttt{score}((r_1, \ldots, r_K), (a_1, \ldots, a_n), (\sigma_1, \ldots, \sigma_n), w, 0)\\ &= \texttt{score}((r_1, \ldots, r_K), (a_1, \ldots, a_N), (\sigma_1, \ldots, \sigma_N), w, 0)\end{aligned}$$

Case 2: $\texttt{end}(r_k) > \texttt{end}(a_n + \sigma_n)$. This case is analog to case 1 with reversed roles of k and n, K and N, and r and $a + \sigma$.

The second loop then subtracts the split penalty p exactly $\texttt{splits}(\sigma)$ according to the definition. *sum* then holds the value $\texttt{score}(r, a, \sigma, w, p)$.

□

The first loops does at most $K + N$ iterations and the second exactly $N - 1$ with a worst case complexity of $O(1)$ each. The total worst-case complexity is then indeed $O(K + N)$.

2.5.3 Optimal no-split alignment

Naive algorithm

The naive algorithm tries all offsets from σ_{min} to σ_{max} and applies algorithm 2 to calculate the score for each offset.

Algorithm 3 Naive no-split algorithm

Input: r, a, weighting function w and split penalty p where r and a are valid sequences
Output: offset σ and $\texttt{nosplit\_score}(r, a, \sigma, w)$

1: $score_{max} \leftarrow 0$
2: $\sigma \leftarrow \sigma_{min}$
3:
4: **for** $\sigma_{current}$ from σ_{min} to σ_{max} **do**
5: `// evaluate as score`$(r, a, (\sigma_{current})_{n=1}^{N}, w, 0)$ `using algorithm 2`
6: $score_{current} \leftarrow \texttt{nosplit\_score}(r, a, \sigma_{current}, w)$
7:
8: **if** $score_{current} > score_{max}$ **then**
9: $sigma \leftarrow sigma_{current}$
10: $score_{max} \leftarrow score_{current}$
11: **end if**
12: **end for**
13:
14: **return** $(\sigma, score_{max})$

When calculating `nosplit_score` as described in 2.5.2, the resulting runtime is $O((T_r + T_a) \cdot (K + N))$ as there are $\sigma_{max} - \sigma_{min}$ evaluations of the scoring function.

Unfortunately a dependency on T_r or T_a is not desirable for the run time complexity, as the number of time units for subtitles is very large. An alternative algorithm brings down the runtime to $O(KN \log \min(K, N))$.

Efficient algorithm

The key idea behind a more efficient algorithm is identifying segments where $\sigma_{current}$ stays constant, or rises or falls by a constant factor from each iteration to the next. It turns out that there can be at most $4KN + 1$ of such segment and most $4KN$ points where this constant factor changes. This becomes apparent when considering each summand of $\texttt{nosplit\_score}(r, a, \sigma, w) = \sum_{k=1}^{K} \sum_{n=1}^{N} \texttt{iscore}(r_k, a_n + \sigma)$ separately (see Figure 2.2).

Let $f_{k,n}(\sigma) := \texttt{iscore}(r_k, a_n + \sigma)$. As one can easily see in Figure 2.1, there are only four $\sigma_{k,n|1}$,$\sigma_{k,n|2}$,$\sigma_{k,n|3}$ and $\sigma_{k,n|4}$ where the *slope* of $f_{k,n}$ changes.

For $\texttt{length}(r_k) \geq \texttt{length}(a_n)$ these positions (with $\sigma_{k,n|1} < \sigma_{k,n|2} < \sigma_{k,n|3} < \sigma_{k,n|4}$) are given by

$$\begin{aligned} \texttt{end}(a_n) + \sigma_{k,n|1} &= \texttt{start}(r_k) \\ \texttt{start}(a_n) + \sigma_{k,n|2} &= \texttt{start}(r_k) \\ \texttt{end}(a_n) + \sigma_{k,n|3} &= \texttt{end}(r_k) \\ \texttt{start}(a_n) + \sigma_{k,n|4} &= \texttt{end}(r_k) \end{aligned}$$

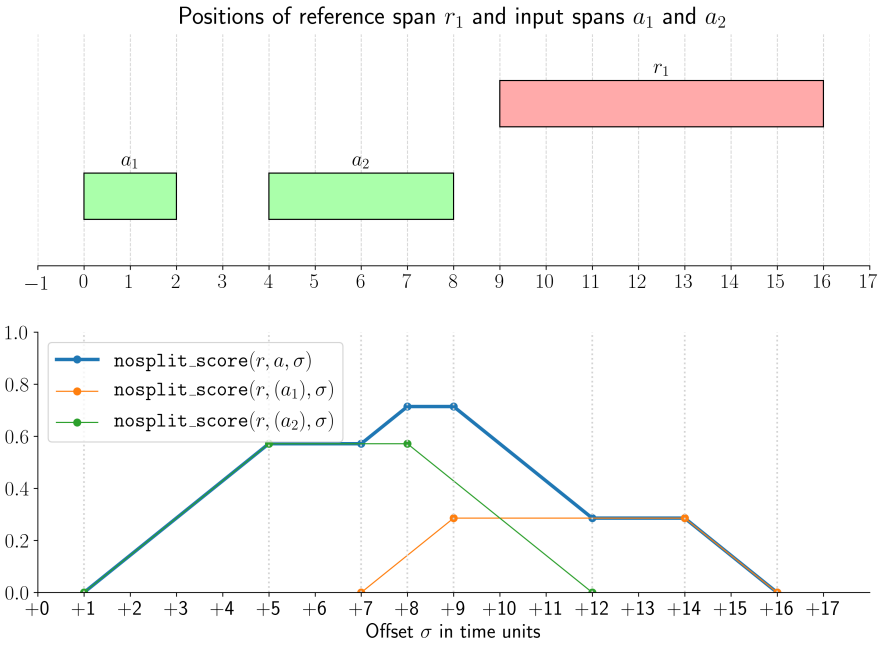

Figure 2.2: Visualization of the sum of two `nosplit_score`s. Note how in each segment the slope of $\texttt{nosplit\_score}(r, a, \sigma)$ is the sum of the slopes of $\texttt{nosplit\_score}(r, (a_1), \sigma)$ and $\texttt{nosplit\_score}(r, (a_2), \sigma)$, and slope changes for $\texttt{nosplit\_score}(r, a, \sigma)$ can only occur for σ where one of its summands has a slope change.

For $\texttt{length}(r_k) \leq \texttt{length}(a_n)$ the second and third position are switched:

$$\begin{aligned}
\texttt{end}(a_n) + \sigma_{k,n|1} &= \texttt{start}(r_k) \\
\texttt{end}(a_n) + \sigma_{k,n|2} &= \texttt{end}(r_k) \\
\texttt{start}(a_n) + \sigma_{k,n|3} &= \texttt{start}(r_k) \\
\texttt{start}(a_n) + \sigma_{k,n|4} &= \texttt{end}(r_k)
\end{aligned}$$

Interestingly, the since $\texttt{start}(a_1) \leq \texttt{end}(a_1) \leq \texttt{start}(a_2) \leq \ldots \leq \texttt{start}(a_N) \leq \texttt{end}(a_N)$ and $\texttt{start}(r_1) \leq \texttt{end}(r_1) \leq \texttt{start}(r_2) \leq \ldots \leq \texttt{start}(r_K) \leq \texttt{end}(a_K)$ we obtain not only

$$\sigma_{k,n|1} < \sigma_{k,n|2} < \sigma_{k,n|3} < \sigma_{k,n|4}$$

for all k and n, but also

$$\sigma_{1,n|1} \leq \sigma_{2,n|1} \leq \ldots \leq \sigma_{K,n|i}$$

for all n and i and even

$$\sigma_{k,1|i} \geq \sigma_{k,2|i} \geq \ldots \geq \sigma_{k,N|i}$$

for all k and i. This can later be used for further optimizations.

$f_{k,n}$ rises to and falls from value $w(k, n)$ in $\texttt{min}(\texttt{length}(r_k), \texttt{length}(a_n))$ so the slope value is

$$\delta_{k,n} = \frac{w(k, n)}{\texttt{min}(\texttt{length}(r_k, a_n))}$$

. We can now define the slope function $h_{k,n}(\sigma)$ so that $f_{r_k,a_n}(\sigma) = \int_{\sigma_{min}}^{\sigma} h_{k,n}(t)\,dt$.

$$h_{k,n}(\sigma) = \begin{cases} 0 & \sigma_{min} \leq \sigma < \sigma_{k,n|1} \\ \delta_{k,n} & \sigma_{k,n|1} \leq \sigma < \sigma_{k,n|2} \\ 0 & \sigma_{k,n|2} \leq \sigma < \sigma_{k,n|3} \\ -\delta_{k,n} & \sigma_{k,n|3} \leq \sigma < \sigma_{k,n|4} \\ 0 & \sigma_{k,n|4} \leq \sigma < \sigma_{max} \end{cases}$$

The slope function can also be represented as a series of *jumps* i.e. the change of the constant value at a specific offset σ. This representation is helpful later for adding two slope functions as their jump arrays simply have to be merged. The array of jumps for $h_{k,n}$ is $[(\sigma_{k,n|1}, \delta_{k,n}), (\sigma_{k,n|2}, -\delta_{k,n}), (\sigma_{k,n|3}, -\delta_{k,n}), (\sigma_{k,n|4}, \delta_{k,n})]$.

We now define $f(\sigma) := \texttt{nosplit\_score}(r, a, \sigma, w)$. The offset σ^* where $f(\sigma^*) = \max_\sigma(f(\sigma))$ is the best no-split alignment for a to r. f can be expressed as the sum of all $\sum_{k=1}^{K}\sum_{n=1}^{N} f_{k,n}$:

$$\begin{aligned} & f(\sigma) \\ & \quad = \texttt{nosplit\_score}(r, a, \sigma, w) \\ & \quad = \sum_{k=1}^{K}\sum_{n=1}^{N} \texttt{iscore}(r_k, a_n + \sigma) \\ & \quad = \sum_{k=1}^{K}\sum_{n=1}^{N} f_{k,n}(\sigma) \\ & \quad = \sum_{k=1}^{K}\sum_{n=1}^{N} \int_{\sigma_{min}}^{\sigma} h_{k,n}(t)\,dt \\ & \quad = \int_{\sigma_{min}}^{\sigma} \sum_{k=1}^{K}\sum_{n=1}^{N} h_{k,n}(t)\,dt \end{aligned}$$

Since all single slope functions jump exactly 4 times the sum of all slope functions can only jump at most $4KN$ times. By sorting all jumps by their offset, one can evaluate the integral piecewise between two adjacent jumps. If two adjacent jumps occur for example on σ_x and σ_y and the sum of all slope functions has the value c on $[\sigma_x, \sigma_y)$ then $\int_{\sigma_x}^{\sigma_y} c\,dt$ is simply $(\sigma_y - \sigma_x) \cdot c$. The last step is filtering f for its maximum, which can only occur on jump offsets (as f rises or falls between jumps). Combining all steps together into one algorithm yields:

Algorithm 4 Calculating the optimal no-split alignment

Input: Valid sequences r and a
Output: σ^* and $\texttt{nosplit\_score}(r, a, \sigma^*, w)$

1: $jumps = \texttt{array}(N, K, 4)$
2: **for** k from 1 to K **do** ▷ Generate jump value and jump offsets for all $h_{k,n}$
3: **for** n from 1 to N **do**
4: **if** $\texttt{length}(r_k) \geq \texttt{length}(a_n)$ **then**
5: $\delta \leftarrow w(k, n)$ / $\texttt{length}(a_n)$
6:
7: $jumps[k, n, 1] \leftarrow (\texttt{start}(r_k) - \texttt{end}(a_n), \delta)$

```
            jumps[k, n, 2] ← (start(r_k) − start(a_n), −δ)
            jumps[k, n, 3] ← (end(r_k) − end(a_n), −δ)
            jumps[k, n, 4] ← (end(r_k) − start(a_n), δ)
        else
            δ ← w(k, n) / length(r_k)

            jumps[k, n, 1] ← (start(r_k) − end(a_n), δ)
            jumps[k, n, 2] ← (end(r_k) − end(a_n), −δ)
            jumps[k, n, 3] ← (start(r_k) − start(a_n), −δ)
            jumps[k, n, 4] ← (end(r_k) − start(a_n), δ)
        end if
    end for
end for

jumps1d ← sort(jumps, by = λ(σ, δ) : σ)      ▷ Flatten 3d array to 1d array sorted by offset

σ_last = σ_min
slope = 0
f_value = 0
f_max = 0
σ* = σ_min
for (σ, δ) in jumps1d do
    f_value = f_value + slope ∗ (σ − σ_last)      ▷ integrate slope to obtain increase/decrease of f

    if f_max < f_value then
        f_max = f_value
        σ* = σ
    end if

    slope = slope + δ
    σ_last = σ
end for

return (σ*, f_max)
```

This algorithm terminates on every input. The correctness follows from the notes above.

Filling and iterating the $4KN$ entries in $jump$ and $jump1d$ array have has complexity $O(KN)$.

Sorting the $jump$ array as a simple 1D-array with merge sort or heap sort would have a worst case complexity $O(KN \log(KN))$. Since all 1D slices $jump[k, n, \_]$, $jump[k, \_, i]$ and $jump[\_, n, i]$ are already sorted individually, one can also merge either $4K$ slices of length N in $O(KN \log K)$ time or alternatively $4N$ slices of length K in $O(KN \log N)$ time. This can either be done as "merge sort" (with the already sorted sequences as leaves) or "heap sort" (maintaining a min-heap of the "current minimum" of each slice). Both sorting methods, and thus the whole algorithm, have a space complexity of $O(KN)$. By choosing the faster direction of the sort, we obtain a total worst case complexity of $O(KN \log \min(K, N))$.

Using `Vec::sort_unstable`() [8] (pattern-defeating quicksort[9]) or `Vec::sort`()[10] (stable merge-sort variant[11]) from the Rust standard library on the $jump$ array was about twice as fast on real-world data than the pre-sorted heap-sort approach and slightly faster than a custom $O(KN \log \min(K, N))$ merge-sort.

Optimal no-split alignment in $O(T_r + T_a + K\dot{N})$ time and $O(T_r + T_a)$ space

A two hour movie has around $N = K = 1200$ subtitles. The number of of possible jump positions with one millisecond per time unit is about $T_a + T_r = 2 \cdot 2\text{h} \cdot 3600\frac{\text{s}}{\text{h}} \cdot 1000\frac{\text{ms}}{\text{s}} \cdot 1\frac{\text{time unit}}{\text{ms}} = 14,400,000$ time units There are exactly $K \cdot N = 1300 \cdot 1300 \cdot 4 = 6,760,000$ jumps. This means every few offsets there will be a jump in the *jump* array. Also $T_a + T_r$ will rises linearly while $N \cdot K$ will rise quadratic with respect to the length of the movie. This makes a counting sort variant a viable alternative.

Algorithm 5 Calculating the optimal no-split alignment with counting sort

Input: Valid sequences r and a
Output: σ^* and `nosplit_score`(r, a, σ^*, w)

1: $jumps =$ `array`(init from σ_{min} to σ_{max} with 0)
2: **for** k from 1 to K **do** ▷ Generate jump value and jump offsets for all $h_{k,n}$
3: **for** n from 1 to N **do**
4: $\delta \leftarrow w(k,n)$ / min(`length`(a_n), `length`(r_k))
5: $jumps$[`start`(r_k) − `end`(a_n)] += δ
6: $jumps$[`start`(r_k) − `start`(a_n)] −= δ ▷ Order of $\sigma_{k,n|2}$ and $\sigma_{k,n|3}$ is not relevant
7: $jumps$[`end`(r_k) − `end`(a_n)] −= δ
8: $jumps$[`end`(r_k) − `start`(a_n)] += δ
9: **end for**
10: **end for**
11:
12: $slope = 0$
13: $f_{value} = 0$
14: $f_{max} = 0$
15: $\sigma^* = \sigma_{min}$
16: **for** σ from σ_{min} to σ_{max} **do**
17: $f_{value} = f_{value} + slope$ ▷ integrate slope for only one time unit
18:
19: **if** $f_{max} < f_{value}$ **then**
20: $f_{max} = f_{value}$
21: $\sigma^* = \sigma$
22: **end if**
23:
24: $slope = slope + jump[\sigma]$
25: **end for**
26:
27: **return** (σ^*, f_{max})

Inserting all jumps into the *jump* array takes $O(KN)$ iterations with $O(1)$ complexity each. Iterating all offsets takes $O(T_r + T_a)$ operations. This algorithm has therefore a runtime complexity of $O(KN + T_r + T_a)$. In practice iterating over the *jump* array takes a negligible amount of time compared to inserting the jump values, possibly because of more efficient utilization of CPU caches. It is about 3 times faster for real data than algorithm 4 (see section 2.6). This algorithm usually also takes less space than algorithm 4 since only the jump values and not the jump offsets have to be saved.

If the ratio between between $4KN$ and $T_r + T_a$ is especially low then this algorithm performs worse and requires more memory than algorithm 4. In that case it makes sense to switch back algorithm 4. If we choose counting sort if $4NK > c \cdot (T_r + T_a)$ for some constant factor c and

merge/heap-sort for $4NK \leq c \cdot (T_r + T_a)$ then the space complexity is $O(\min(NK, T_r + T_a))$.

2.5.4 Optimal split alignment

A brute-force search for the optimal alignment for the general scoring function is not practically possible. Even enumerating all possible splits has a complexity of $O(2^N)$ and is clearly not suited for a real-world data ($N = 1500$). Instead, an algorithm using dynamic programming was chosen, which reduces the complexity to $O((T_r + T_a) \cdot N)$.

In the first section the recursion will be derived, the second section describes the algorithm for the optimal split alignment and the third section explains methods to reduce runtime and memory requirements.

Recursion formulas

The idea is to create a table of size $N \cdot (T_r + T_a)$ where each $t(n, \sigma)$ represents the `score` only the sub-sequence $(a_1, \ldots, a_n)$ where the alignment's σ_n is fixed to a given σ. To efficiently compute this table, a second function $s(n, \sigma)$ is needed. This function calculates the maximum score for the first $n-1$ input sequence entries, where $a_{n-1} + \sigma_{n-1}$ has to end before the next span a_n with offset σ. This means σ_{n-1} can be at most $\texttt{start}(a_n) - \texttt{end}(a_{n-1})$ greater than σ. For brevity this "offset shift" function is called $shift(n, \sigma)$.

Definition 2.5.10. For all $1 \leq n \leq N$ and all σ the functions $shift$, t, s are defined as

$$
\begin{aligned}
shift(n, \sigma) &= \sigma + \texttt{start}(a_n) - \texttt{end}(a_{n-1}) \\
t(n, \sigma) &= \max_{\substack{(\sigma_1,\ldots,\sigma_n) \text{ where } \sigma_n = \sigma \\ \text{and } (a_1,\ldots,a_n)+(\sigma_1,\ldots,\sigma_n) \text{ is valid}}} \texttt{score}(r, (a_1, \ldots, a_n), (\sigma_1, \ldots, \sigma_n), w, p) \\
s(n, \sigma) &= \max_{\substack{(\sigma_1,\ldots,\sigma_{n-1}) \text{ where} \\ \sigma_{n-1}+\texttt{end}(a_{n-1}) \leq \sigma+\texttt{start}(a_n) \text{ and} \\ (a_1,\ldots,a_{n-1})+(\sigma_1,\ldots,\sigma_{n-1}) \text{ is valid}}} \texttt{score}(r, (a_1, \ldots, a_{n-1}), (\sigma_1, \ldots, \sigma_{n-1}), w, p) \\
&= \max_{\substack{(\sigma_1,\ldots,\sigma_{n-1}) \text{ where } \sigma_{n-1} \leq shift(n,\sigma) \\ \text{and } (a_1,\ldots,a_{n-1})+(\sigma_1,\ldots,\sigma_{n-1}) \text{ is valid}}} \texttt{score}(r, (a_1, \ldots, a_{n-1}), (\sigma_1, \ldots, \sigma_{n-1}), w, p)
\end{aligned}
$$

Lemma 2.5.1. *For all $\sigma \leq \sigma_{min}$ and $1 \leq n \leq N$*

$$t(n, \sigma) = 0$$

is true. For all $\sigma \geq \sigma_{max}$ and $1 \leq n \leq N$

$$t(n, \sigma) \leq \max_{\sigma_{min} \leq \sigma^* < \sigma_{max}} t(n, \sigma^*)$$

is satisfied.

Proof. For $\sigma \leq \sigma_{min}$ all spans are positioned before any spans in r. Therefore no overlap between sub-sequences from a and r are possible. It follows $t(n, \sigma) = 0$ for all $1 \leq n \leq N$ and $\sigma \leq \sigma_{min}$.

Let us assume $\sigma \geq \sigma_{max}$. For $t(n, \sigma) = 0$ there is σ_{min} with $t(n, \sigma) = t(n, \sigma_{min})$. So let us additionally assume $t(n, \sigma) > 0$. For offsets greater or equal than σ_{max}, the respective spans can not overlap with any spans in r. An alignment $(\sigma_1, \ldots, \sigma_{n-1}, \sigma)$ with a positive `score` from $t(n, \sigma)$ therefore has to have a split. Let the index of the last offset below σ_{max} be m so that the alignment can be written as $(\sigma_1, \ldots, \sigma_m, \sigma, \ldots, \sigma)$. The alignment $(\sigma_1, \ldots, \sigma_m, \sigma_m, \ldots, \sigma_m)$ will then have a `score` greater of equal than the `score` of of $(\sigma_1, \ldots, \sigma_m, \sigma, \ldots, \sigma)$ because there is one less split and possibly more overlaps with spans in r. Therefore $t(n, \sigma) \leq t(n, \sigma_m)$ where $\sigma_{min} < \sigma_m < \sigma_{max}$. □

Lemma 2.5.2. *For all $2 \leq n \leq N$ and all σ the recursion formula*

$$s(n,\sigma) = \begin{cases} \max \begin{cases} t(n-1, shift(n,\sigma)) \\ s(n,\sigma-1) \end{cases} & \text{if } shift(n,\sigma) < \sigma_{max} \\ s(n,\sigma-1) & \text{if } shift(n,\sigma) \geq \sigma_{max} \end{cases}$$

is satisfied.

Proof. The key observation is that going from $s(n,\sigma-1)$ to $s(n,\sigma)$ there is only one additional offset value for σ_n:

$$\begin{aligned} s(n,\sigma) &= \max_{\substack{(\sigma_1,\ldots,\sigma_{n-1}) \text{ where} \\ \sigma_{n-1} \leq shift(n,\sigma) \text{ and} \\ (a_1,\ldots,a_{n-1})+(\sigma_1,\ldots,\sigma_{n-1}) \text{ is valid}}} \texttt{score}(r,(a_1,\ldots,a_{n-1}),(\sigma_1,\ldots,\sigma_{n-1}),w,p) \\ &= \max \begin{cases} \max_{\substack{(\sigma_1,\ldots,\sigma_{n-1}) \text{ where} \\ \sigma_{n-1}+\texttt{end}(a_{n-1}) = shift(n,\sigma) \text{ and} \\ (a_1,\ldots,a_{n-1})+(\sigma_1,\ldots,\sigma_{n-1}) \text{ is valid}}} \texttt{score}(r,(a_1,\ldots,a_{n-1}),(\sigma_1,\ldots,\sigma_{n-1}),w,p) \\ \max_{\substack{(\sigma_1,\ldots,\sigma_{n-1}) \text{ where} \\ \sigma_{n-1}+\texttt{end}(a_{n-1}) \leq shift(n,\sigma)-1 \text{ and} \\ (a_1,\ldots,a_{n-1})+(\sigma_1,\ldots,\sigma_{n-1}) \text{ is valid}}} \texttt{score}(r,(a_1,\ldots,a_{n-1}),(\sigma_1,\ldots,\sigma_{n-1}),w,p) \end{cases} \\ &= \max \begin{cases} t(n-1, shift(n,\sigma)) \\ s(n,\sigma-1) \end{cases} \end{aligned}$$

The problem with this form is that $shift(n,\sigma)$ might be greater or equal σ_{max} for some $\sigma < \sigma_{max}$ which would result in an out-of-bounds access in the final algorithm. Fortunately 2.5.1 states the maximum of $t(n,\sigma)$ has already occurred for $\sigma < \sigma_{max}$. Therefore it is not necessary to compute $t(n-1, shift(n,\sigma)))$ for $shift(n,\sigma)) \geq \sigma_{max}$:

$$s(n,\sigma) = \begin{cases} \max \begin{cases} t(n-1, shift(n,\sigma)) \\ s(n,\sigma-1) \end{cases} & \text{if } shift(n,\sigma) < \sigma_{max} \\ s(n,\sigma-1) & \text{if } shift(n,\sigma) \geq \sigma_{max} \end{cases}$$

□

Lemma 2.5.3. *For all $2 \leq n \leq N$ and all σ*

$$t(n,\sigma) = \texttt{score}(r,(a_n),(\sigma),w,0) + \max \begin{cases} t(n-1,\sigma) \\ s(n,\sigma) - p \end{cases}$$

is satisfied.

Proof. For $t(n,\sigma)$ we can derive a recursion formula by separating the `score` calculation of a_n from the `score` calculation preceding sequence. For that we have to differentiate the case the

offset where the no-split bonus is added from the offsets where it is not added:

$$
\begin{aligned}
t(n,\sigma) &= \max_{\substack{(\sigma_1,\dots,\sigma_n) \text{ where } \sigma_n=\sigma \\ \text{and } (a_1,\dots,a_n)+(\sigma_1,\dots,\sigma_n) \text{ is valid}}} \texttt{score}(r,(a_1,\dots,a_n),(\sigma_1,\dots,\sigma_n),w,p) \\
&= \sum_{k=1}^{K} \texttt{iscore}(r_k,a_n) w(k,n) \\
&\quad + \max \begin{cases} \max\limits_{\substack{(\sigma_1,\dots,\sigma_{n-1}) \text{ where } \sigma_{n-1}=\sigma \\ \text{and } (a_1,\dots,a_{n-1})+(\sigma_1,\dots,\sigma_{n-1}) \\ \text{is valid}}} \texttt{score}(r,(a_1,\dots,a_{n-1}),(\sigma_1,\dots,\sigma_{n-1}),w,p) \\ \max\limits_{\substack{(\sigma_1,\dots,\sigma_{n-1}) \text{ where} \\ \texttt{end}(a_{n-1})+\sigma_{n-1} \le \texttt{start}(a_n)+\sigma \\ \text{and } (a_1,\dots,a_{n-1})+(\sigma_1,\dots,\sigma_{n-1}) \\ \text{is valid}}} \texttt{score}(r,(a_1,\dots,a_{n-1}),(\sigma_1,\dots,\sigma_{n-1}),w,p) - p \end{cases} \\
&= \texttt{score}(r,(a_n),(\sigma),w,0) + \max \begin{cases} t(n-1,\sigma) \\ s(n,\sigma) - p \end{cases}
\end{aligned}
$$

□

By combining all recursion formulas we obtain the central theorem to efficiently calculate the optimal split alignment.

Theorem 2.5.2. *For all $1 \le n \le N$ and all σ*

$$
\begin{aligned}
s(n,\sigma_{min}) &= 0 \\
t(1,\sigma) &= \texttt{score}(r,(a_1),(\sigma),w,p)
\end{aligned}
$$

is true. For all $2 \le n \le N$ the recursion formulas

$$
\begin{aligned}
s(n,\sigma) &= \begin{cases} \max \begin{cases} t(n-1,shift(n,\sigma)) \\ s(n,\sigma-1) \end{cases} & \text{if } shift(n,\sigma) < \sigma_{max} \\ s(n,\sigma-1) & \text{if } shift(n,\sigma) \ge \sigma_{max} \end{cases} \\
t(n,\sigma) &= \texttt{score}(r,(a_n),(\sigma),w,0) + \max \begin{cases} t(n-1,\sigma) \\ s(n,\sigma) - p \end{cases}
\end{aligned}
$$

are satisfied. The optimal split `score` *is given by $\max_{\sigma_{min} \le \sigma < \sigma_{max}} t(N,\sigma)$. If σ_N^* is an offset where $t(N,\sigma_N^*) = \max_{\sigma_{min} \le \sigma < \sigma_{max}} t(N,\sigma)$ then the optimal alignment σ^* is given by $(\sigma_1^*,\dots,\sigma_N^*)$ where $\sigma_{n-1}^* = t_o(n,\sigma_n^*)$ for all $2 \le n \le N$ and t_o is defined as*

$$
\begin{aligned}
s_o(n,\sigma) &= \begin{cases} shift(n,\sigma) & \text{if } shift(n,\sigma) \le \sigma_{max} \\ & \text{and } s(n,\sigma-1) < t(n-1,shift(n,\sigma)) \\ & \text{or if } \sigma = \sigma_{min} \\ s_o(n,\sigma-1) & \text{otherwise} \end{cases} \\
t_o(n,\sigma) &= \begin{cases} \sigma & \text{if } t(n-1,\sigma) \ge s(n,\sigma) - p \\ s_o(n,\sigma) & \text{otherwise} \end{cases}
\end{aligned}
$$

Proof. $s(n, \sigma_{min}) = 0$ is zero as the condition for the offsets is $\sigma_{n-1} + \texttt{end}(a_n - 1) \leq \sigma + \texttt{start}(a_n)$. This means $a_n + \sigma_{min}$ is shifted in front of all spans in r and a_{n-1} has to end before a_n. Together with the condition that $(a_1, \ldots, a_{n-1}) + (\sigma_1, \ldots, \sigma_{n-1})$ is valid, all spans in $(a_1, \ldots, a_{n-1}) + (\sigma_1, \ldots, \sigma_{n-1})$ do not overlap with any spans in r and therefore the `score` is zero. $t(1, \sigma) = \texttt{score}(r, (a_1), (\sigma), w, p)$ since no split can occur for only one input sequence span and σ_1 is equal σ.

The recursion formulas are given by 2.5.2 and 2.5.3. From 2.5.1 directly follows that the maximum $t(N, \sigma)$ for any σ occurs for a σ within $\sigma_{min} \leq \sigma < \sigma_{max}$.

The derivations of the recursion formulas contain for a given $\sigma_n = \sigma$ how σ_{n-1} has to be chosen to obtain the maximum score. For $s(n, \sigma)$ the score is the same as $s(n, \sigma - 1)$ except if $t(n-1, shift(n, \sigma))$ is larger than $s(n, \sigma - 1)$. $t(n-1, shift(n, \sigma))$ represents the `score` where $\sigma_{n-1} = shift(n, \sigma)$. If the `score` is unchanged from $\sigma - 1$ to σ, so is the offset for $s_o(n, \sigma - 1)$ and $s_o(n, \sigma)$. σ_{min} has to serve as a starting point for the recursion. Since $s(n, \sigma_{min}) = 0$, we can choose the offset $\sigma_{n-1} = shift(n, \sigma_{min})$ as the resulting score is zero (first statement of this theorem). All cases combined yield

$$s_o(n, \sigma) = \begin{cases} shift(n, \sigma) & \text{if } shift(n, \sigma) \leq \sigma_{max} \\ & \text{and } s(n, \sigma - 1) < t(n - 1, shift(n, \sigma)) \\ & \text{or if } \sigma = \sigma_{min} \\ s_o(n, \sigma - 1) & \text{otherwise} \end{cases}$$

Similarity for $t_o(n, \sigma)$ the offset depends on whether $t(n-1, \sigma)$ is greater or equal $s(n, \sigma) - p$. If $t(n-1, \sigma)$ is indeed greater or equal then the best score is achieved by not splitting the subtitle sequence, therefore $\sigma_{n-1} = \sigma$. In the other case the best offset for σ_{n-1} is calculated with $s_o(n, \sigma)$. This yields

$$t_o(n, \sigma) = \begin{cases} \sigma & \text{if } t(n-1, \sigma) \geq s(n, \sigma) - p \\ s_o(n, \sigma) & \text{otherwise} \end{cases}$$

□

Algorithm

The recursion formula can be evaluated by the dynamic programming method from the bottom up. For each *phase* n from 1 to N there are $T_r + T_a$ *sub-steps* for σ from σ_{min} up to $\sigma_{max} - 1$. The data for t in phase n is not needed after phase $n+1$ and can therefore be discarded. $s(n, \sigma)$ is only needed for $t(n, \sigma)$ and computed using the last $s(n, \sigma - 1)$. It can therefore simply be implemented as single variable.

By using algorithm 5 to calculate `nosplit_score` we obtain the following algorithm. As we will later see it is too inefficient and requires too much memory to be useful, but provides a starting point for optimization.

Algorithm 6 Computing the optimal split alignment

Input: valid span sequences r and a, the weighting w and split penalty p
Output: optimal split alignment $\sigma^* = (\sigma_1^*, \ldots, \sigma_n^*)$ and $\texttt{score}(r, a, \sigma^*, w, p)$

1: **function** SCORES(n) ▷ calculates $\texttt{nosplit\_score}(r, a_n, \sigma, w)$ for σ from σ_{min} to $\sigma_{max} - 1$
2:

```
3:      jumps ← array(from σ_min to σ_max − 1)
4:      for k from 1 to K do                          ▷ Fill jumps array like in algorithm 5
5:          δ ← w(k, n) / min(length(a_n), length(r_k))
6:          jumps[start(r_k) − end(a_n)] += δ
7:          jumps[start(r_k) − start(a_n)] −= δ       ▷ Order of σ_{k,n|2} and σ_{k,n|3} is not relevant
8:          jumps[end(r_k) − end(a_n)] −= δ
9:          jumps[end(r_k) − start(a_n)] += δ
10:     end for
11:
12:     slope ← 0
13:     score ← NewArray(σ_min to σ_max − 1)
14:     score[σ_min] = 0
15:     for σ from σ_min + 1 to σ_max − 1 do          ▷ Fill score using jumps (algorithm 5)
16:         score[σ] ← score[σ − 1] + slope           ▷ Integrate slope for one time unit
17:         slope ← slope + jumps[σ]
18:     end for
19:
20:     return score
21: end function
22:
23: t_{n−1} ← Scores(1)
24: t_o ← InitArray((2 to N) × (σ_min to σ_max − 1))
25: for n from 2 to N do                              ▷ Phase n calculates t(n, _) and s(n, _)
26:     scores ← Scores(n)
27:     s ← 0                                         ▷ Initialize running variable s with value s(n, σ_min) = 0
28:     s_o ← σ_min                                   ▷ Initialize running variable s_o with value s_o(n, σ_min) = σ_min
29:     for σ from σ_min to σ_max − 1 do
30:         // Variable s is equal to s(n, σ) and variable s_o is equal to s_o(n, σ)
31:
32:         /*
                Calculate

                    t(n, σ) = score(r, (a_n), (σ), w, 0) + max { t(n − 1, σ)
                                                               { s(n, σ) − p

33:
                    t_o(n, σ) = { σ          if t(n − 1, σ) ≥ s(n, σ) − p
                                { s_o(n, σ)  otherwise

34:         */
35:
36:         if t_{n−1}[σ] ≥ s − p then
37:             t_n[σ] ← scores[σ] + t_{n−1}[σ]
38:             t_o[n, σ] ← σ
39:         else
40:             t_n[σ] ← scores[σ] + s − p
41:             t_o[n, σ] ← s_o
42:         end if
43:
44:         /*
```

Calculate for $\sigma + 1$ the new

$$s(n,\sigma) = \begin{cases} \max \begin{cases} t(n-1, shift(n,\sigma)) \\ s(n,\sigma-1) \end{cases} & \text{if } shift(n,\sigma) < \sigma_{max} \\ s(n,\sigma-1) & \text{if } shift(n,\sigma) \geq \sigma_{max} \end{cases}$$

45:

$$s_o(n,\sigma) = \begin{cases} & \text{if } shift(n,\sigma) \leq \sigma_{max} \\ shift(n,\sigma) & \text{and } s(n,\sigma-1) < t(n-1, shift(n,\sigma)) \\ & \text{or if } \sigma = \sigma_{min} \\ s_o(n,\sigma-1) & \text{otherwise} \end{cases}$$

46: */
47: **if** $shift(n,\sigma) < \sigma_{max}$ **then**
48: **if** $s < t_{n-1}[shift(n,\sigma)]$ **then**
49: $s \leftarrow t_{n-1}[shift(n,\sigma)]$
50: $s_o \leftarrow shift(n,\sigma)$
51: **end if**
52: **end if**
53: **end for**
54: **end for**
55:
56: $\sigma^* \leftarrow$ Array(from 1 to N)
57: $\sigma^*[N] \leftarrow \sigma_{min}$
58: $score^* \leftarrow 0$ ▷ $t(N, \sigma_{min})$ is 0 (lemma 2.5.1)
59: **for** σ from σ_{min} to σ_{max} **do** ▷ Find σ_N^* where $t(N, \sigma_N^*) = \max_{\sigma_{min} \leq \sigma < \sigma_{max}} t(N, \sigma)$
60: **if** $score^* < t_{n-1}[\sigma]$ **then**
61: $score^* \leftarrow t_{n-1}[\sigma]$
62: $\sigma^*[N] \leftarrow \sigma$
63: **end if**
64: **end for**
65:
66: **for** n from N to 2 **do** ▷ Evaluate $\sigma_{n-1}^* = t_o(n, \sigma_n^*)$ for all $2 \leq n \leq N$
67: $\sigma^*[n-1] \leftarrow t_o[n, \sigma^*[n]]$
68: **end for**

This algorithm is correct as it simply evaluates theorem 2.5.2. In this form it is unfortunately not very practical.

Scores has a runtime complexity of $O(K + T_r + T_a)$ but since $K < T_r$, it can be simplified to $O(T_r + T_a)$. Scores also creates two arrays with $T_r + T_a$ each and therefore has a space complexity of $O(T_r + T_a)$. The main loop iterates $N - 1$ times where each iteration has a $O(T_r + T_a)$ time complexity. The last two loops in the algorithm have a runtime of $O(T_r + T_a)$ and $O(N)$ and are therefore negligible. The total runtime complexity is then $O(N \cdot (T_r + T_a))$. In practice synchronizing a subtitle to a full-length movie takes several minutes, which is too slow to be useful.

The memory requirements also prevent this algorithm to run in the main memory of current computers. The problem is table t_o which has $(N-1) \cdot (T_r + T_a)$ entries. For a two hour movie with one millisecond per time unit and $N = 1000$ we obtain 13.4 billion entries. If every offset is saved in 4 bytes, the table contains about 53.6GB of data.

Reducing memory footprint

The amount of needed memory can be greatly reduced by only combining offsets into segments where the "slope" stays constant. If for example for a given n and all values σ the value $t_{n-1}[\sigma]$ is always greater or equal than $s-p$ then $t_o[n,\sigma]$ is always set to σ. The whole data of $t_o[n,\_]$ can then be compressed into the single information "starts at σ_{min} and increases by 1 time unit from one value to the next". Similarly for $s_o[n,\_]$ if $s \geq t_{n-1}[shift(n,\sigma)]$ is satisfied repeatedly, then $s_o[n,\_]$ has large segments where it stays constant. If on the other hand $s < t_{n-1}[shift(n,\sigma)]$ is repeatedly satisfied, then $s_n[n,\sigma] = shift(n,\sigma)$ and $s_o[n,\_]$ has again a segment where difference between subsequent entries is exactly 1.

This means if the data is stored in segments, a new entry only has to be created if the evaluation condition switches between $t[n,\sigma]$ and $t[n,\sigma-1]$, or if the evaluation condition switch from $s[n,\sigma-1]$ to $s[n,\sigma]$. Although a theoretical limit on the number of switches is hard to establish, the amount of data stayed well below 150MB for all 118 tested subtitle alignments.

Similar to algorithm 4 and the described memory optimization, the performance can be increased by also processing the score data for $t[n,\_]$ and $s[n,\_]$ as single segments whenever the slope of the score is constant.

A segment containing score information is needed, as well as a segment type that contains score and offset information (which for example combines the data from $t(n,\sigma)$ and $t_o(n,\sigma)$).

```
struct ScoreSegment{
    start: Offset,
    end: Offset,

    score: Score,
    slope: Score
}

// a 'dual segment' contains both score and offset information
struct DualSegment{
    start: Offset,
    end: Offset,

    score: Score,
    slope: Score,

    // 'offset' is the offset when the segment starts...
    offset: Offset,
    // ...and if drag is true, the offsets have a slope of one
    drag: boolean
}

// define arrays of segments where each segment
// starts where the last segment ends
type ScoreBuffer = Vec<ScoreSegment>;
type DualBuffer = Vec<DualSegment>;
```

Instead of returning buffers, the functions' inputs and outputs are iterators and segment processing is done in an online fashion. This way much less memory has to be allocated and deallocated, and most operations can be done in the CPU registers if the compiler inlines all iterator steps into one function.

A score segment iterator is denoted by [*ScoreSegment*] and a dual segment iterator is denoted by [*DualSegment*].

A few functions which are very simple on their own are needed to elegantly implement the optimal split algorithm. Their uses becomes apparent when composed.

- [*ScoreSegment*] → `shift`(σ_{shift}) → [*ScoreSegment*]: Adds σ to `start` and `end` attribute. This results in setting the score of $out[\sigma]$ to $in[\sigma - \sigma_{shift}]$ for all σ_{min} to $\sigma_{max} - 1$.
- [*ScoreSegment*] → `fix_bounds`($\sigma_{min}, \sigma_{max}$) → [*ScoreSegment*]: Cut off and delete all segments before σ_{min} and append a zero-score segment to the end which goes to σ_{max}. It is only used after a $shift(\sigma)$ where σ is non-positive to restore the original boundaries.
- [*ScoreSegment*] → `add_score`(s) → [*ScoreSegment*]: Adds s to `score` value resulting in $out[\sigma] = in[\sigma] + s$.
- [*DualSegment*] → `add_score_iter`($in2$: [*ScoreSegment*]) → [*DualSegment*]: Add the score of another score iterator to the scores of the dual segment iterator ($out[\sigma] = in[\sigma] + in2[\sigma]$). The resulting number of segments is at most the sum of the segment counts of both input iterators together. The offsets in the dual iterator are not changed.
- [*ScoreSegment*] → `annotate_with_offset`(σ_{shift}) → [*DualSegment*]: Transform the `ScoreSegment` iterator into a `DualSegment` iterator, by setting $offset \leftarrow start + \sigma$ and $drag \leftarrow true$. This means the offset of $out[\sigma]$ is set to $\sigma + \sigma_{shift}$.
- [*DualSegment*] → `left_to_right_maximum`() → [*DualSegment*]: Go over all segments and save maximum encountered score s_{max}. If the subsequent segment drops below s_{max}, return a segment with score s_{max} and the offset where s_{max} occurred until a segment again has a higher score than s_{max}.
- `maximum`([*DualSegment*], [*DualSegment*]) → [*DualSegment*]: Two takes two input iterators and return the segments which has the higher score.
- [*ScoreSegment*] → `save_score`() → $ScoreBuffer$ This function actually executes all iterators and stores the score segments in an array.
- [*DualSegment*] → `save_dual`() → ($ScoreBuffer, OffsetBuffer$) This function actually executes all iterators and stores the segments in two arrays. It separates the score data from the offset data. It also combines subsequent segments if the end score/offset of one segment is the start score/offset of the next segment and the segments have the same slope.

Algorithm 7 Optimal split alignment with segment iterators

Input: valid span sequences r and a, the weighting w and split penalty p
Output: optimal split alignment $\sigma^* = (\sigma_1^*, \ldots, \sigma_n^*)$ and `score`(r, a, σ^*, w, p)

```
1: t_{n-1} ← GetScoreIter(r, (a_n)).save_score()
2:
3: for n in 2 to N do
4:     // t_iter has at position σ the score t(n − 1, σ) and the offset σ
5:     t_iter ← t_{n−1}.as_iter()                          ▷ t_{n−1}[σ] is t(n − 1, σ).
6:                 .annotate_with_offset(0)                ▷ Attaches offset σ to t_iter[σ]
7:
8:     // s_iter has at position σ the score t(n − 1, shift(n, σ))
9:     // and the offset shift(n, σ)
```

10: $shift\_value \leftarrow \texttt{start}(a_n) - \texttt{end}(a_{n-1})$
11: $s\_iter \leftarrow t_{n-1}.\texttt{as\_iter}()$
12: .$\texttt{shift}(-shift\_value)$ ▷ Set score of $s\_iter[\sigma]$ to $t(n-1, shift(n,\sigma))$.
13: .$\texttt{fix\_bounds}(\sigma_{min}, \sigma_{max})$ ▷ Restore boundaries as σ_{min} to $\sigma_{max} - 1$.
14: .$\texttt{annotate\_with\_offset}(shift\_value)$ ▷ Set offset of $s\_iter[\sigma]$ to $shift(n,\sigma)$.
15:
16: /*
Sweep across s_iter from left to right and only copy scores/offsets if the new score is higher than all previous scores. Otherwise keep the previous high score and its offset. This is done segment-
17: wise and has the potential to greatly reduce the number of segments if the current s_iter has many valleys. Afterwards the score of $s\_iter[\sigma]$ is $s(n,\sigma)$ and the offset of $s\_iter[\sigma]$ is $s_o(n,\sigma)$.
18: */
19: $s\_iter \leftarrow s\_iter.\texttt{left\_to\_right\_maximum}()$
20:
21: /*
Creates an score segment iterator with

$$score\_iter[\sigma] = \texttt{score}(r, (a_n), (\sigma), w, p)$$

22:
like algorithm 4. Since there only need to be merged 4 jump slices with K jumps each, this is done in $O(K)$. The resulting number of score segments is $4K+1$ as there are $4K$ slope changes.
23: */
24: $score\_iter \leftarrow$ GetScoreIter$(r, (a_n))$
25:
26:
27: /*
Calculate

$$t_n[\sigma] = \texttt{score}(r, (a_n), (\sigma), w, 0) + \max \begin{cases} t(n-1,\sigma) \\ s(n,\sigma) - p \end{cases}$$

28:
$$t_o[n][\sigma] = \begin{cases} \sigma & \texttt{if } t(n-1,\sigma) \geq s(n,\sigma) - p \\ s_o(n,\sigma) & \texttt{otherwise} \end{cases}$$

29: */
30: $(t_n, t_o[n]) \leftarrow \texttt{maximum}(t\_iter, s\_iter.\texttt{add\_score}(-p))$
31: .$\texttt{add\_score\_iter}(score\_iter)$
32: .$\texttt{save\_dual}()$
33:
34: $t_{n-1} \leftarrow t_n$
35: **end for**
36:
37: $\sigma^* \leftarrow$ InitArray(from 1 to N)
38:
39: // Get the highest score in the $ScoreBuffer$ and the σ where it occurs.

40: $(\sigma^*[N], score) \leftarrow t_{n-1}$.`get_maximum_point`()
41: **for** n in 2 to N **do** ▷ Extract the optimal alignment σ^* from t_o.
42: $\sigma^*[n-1] \leftarrow t_o[n]$.`get_offset_at`($\sigma^*[n]$)
43: **end for**
44:
45: **return** $(\sigma^*, score)$

Further improvements in memory usage can be made by discarding the `start` attribute in *ScoreBuffer*s for the segments as it is simply `end` of the next segment. The start of the first segment in the buffers is always σ_{min}.

Although this algorithm greatly reduces the memory usage, the runtime performance is not increased compared to algorithm 6. The problem is that the number of output segments of the `maximum` operation has an upper bound of the sum of the number of input segments. This means t_n might have twice as many segments as t_{n-1}. `add_score_iter` can also introduce up to $4K$ new segments. t_n will also always have at least as many segments as t_{n-1}. The observed behavior is that each iteration of the main loop takes more time than the previous iteration until each score segment has the length of one time unit. When that happens, algorithm 7 becomes slower than 6 because of the overhead of "segment" processing.

Since each iteration of the main loop takes time proportional to the number of segments of t_{n-1}, a large speedup can be achieved by reducing this number. The central idea is merging segments which have only slightly different slopes. This merging of segments only changes the implementation of `save_dual`(), which is presented in the next section.

Increasing performance by merging score segments

The merging of score segments should not introduce an error of more than ϵ, where ϵ is a constant which trades alignment accuracy for algorithm speed.

Although there exist several algorithms for *line simplification* like the Douglas-Peucker-Algorithm or Visvalingam–Whyatt-Algorithm, they require random access to all points on the line and always reuse a subset of these points (which is not required in this case). The typically also have a runtime complexity of more than $O(m)$ for m line segments. Instead we will use a custom line simplification algorithm.

The key function for a simple $O(m)$ online algorithm is the `slopes`(p, t, ϵ) function, which takes a pivot point p, a "target point" t and a maximum error ϵ as input. It returns an interval of slopes where all lines through p with a slope $s \in$ `slopes`(p, t, ϵ) have a maximum error of ϵ at position t_x.

$$\begin{aligned}&\texttt{slopes}(p, t, \epsilon) = [\\ &\quad (t_y + \epsilon - p_y)/(t_x - p_x),\\ &\quad (t_y - \epsilon - p_y)/(t_x - p_x)\\ &]\end{aligned}$$

Now the simplification can be done in three steps:

1. For the current segment with start point $start_0$ end end point end_0 choose the pivot point p as middle segment middle point $p = (start_0 + end_0)/2$. Calculate slopes $s = (p, end_0, \epsilon)$. This means all lines through p with slopes in s stay within the error ϵ compared to the current segment.

2. Pull next segment with start point *start* and end point *end* from the preceding iterator. Calculate $s' = s \cap slope(p, start, \epsilon) \cap slope(p, end, \epsilon)$. If s' is non-empty there exists a line l through pivot point p so that the error between l and the segments is less or equal ϵ. Repeat this step, until...

3. ...s' is empty, so no such line through p exists. In that case all previous segments can be merged to a single segment which goes through p and has slope in s (for example choosing the middle of the slope interval). The current segment is then used as starting point in step 1.

This approach reduces the runtime from several minutes to a few seconds without sacrificing much of the alignment quality (see section 2.6).

It is possible to choose ϵ different in each iteration of the main loop of algorithm 7. Since the number of segments is very small in the first iterations, it is not necessary to aggressively merge segments. The error ϵ_n for iteration n was chosen as $\epsilon_n = (0.2 + 0.8\frac{n}{N}) \cdot 0.05 \cdot E$ in the reference implementation for a user defined constant E. The $(0.2 + 0.8\frac{n}{N})$ leads to an almost constant time per iteration and 0.05 is a scaling factor so that useful values for E start from 1 (see section 2.6).

To analyze the upper bound on the total introduces score error, note that all operations except `save_dual` do not increase the maximum error value:

- shifting: This case is trivial. If all scores have at most an error of ϵ, after shifting all scores have an error of ϵ afterwards.
- adding $\tilde{x} = x + \delta$ and y where $|\delta_x| \leq \epsilon$: if the input value is $x + \delta$, where x is the actual value and δ the error, the result is $\tilde{x} + y = x + y + \delta$ will exactly have an error of δ. So `add_score`() and `add_score_iter`() preserve the maximum error ϵ.
- maximum of $\tilde{x} = x + \delta_x$ and $\tilde{y} = y + \delta_y$ where $|\delta_x| \leq \epsilon$ and $|\delta_y| \leq \epsilon$: Without loss of generality let's assume $x \geq y$:

$$\begin{aligned} &|\max(x + \delta_x, y + \delta_y) - \max(x, y)| \\ &= |\max(x + \delta_x, y + \delta_y) - x| \\ &= |\max(x + \delta_x - x, y + \delta_y - x)| \\ &= |\max(\delta_x, y - x + \delta_y)| \end{aligned}$$

  Since $y - x \leq 0$ we have $\delta_x \leq \epsilon$ and $y - x + \delta_y \leq \epsilon$ and therefore $\max(\delta_x, y - x + \delta_y) \leq \epsilon$. On the other hand we have $\epsilon \leq \delta_x$ which results in $\epsilon \leq \max(\delta_x, y - x + \delta_y)$.

  Both cases combined together prove $|\max(\delta_x, y - x + \delta_y)| \leq \epsilon$.

  So `left_to_right_maximum()` and `maximum`() preserve the maximum error ϵ.

The error of the optimal split alignment score is therefore at most $\sum_{n=2}^{N} \epsilon_n$. Note that for higher N the error rises, but so does the optimal split alignment score.

Normalizing the split penalty by video length

Larger N and K result in higher scores. If the same split penalty p is used for longer subtitles and movies, more splits will be introduced in total. To avoid this behavior, a *normalized* split penalty P is divided by the maximum possible score, which is $\min(K, N)$ (given all $w(k, n) \leq 1$):

$$p = 0.001 \cdot P / \min(K, N)$$

The 0.001 is a constant factor to shift useful values for P into the single digit range (see section 2.6). If $P = 1000$, it is guaranteed that the optimal split score is the same as the optimal no-split score.

2.5.5 Correcting differences in framerate

Although modern container formats can save video streams with an arbitrary framerate, due to the legacy of analog television almost all videos use one a few common framerates.

Differences in playback speed are assumed to be a result of re-encoding a video stream with a slightly different common framerate. Only a few common fractions are therefore possible, which are

- $24/23.976 = 30/29.97 = 60/59.94 = 1001/1000$
- $25/24$
- $25/23.976$

and their reciprocal values. In total there are 7 ratios, including the ratio 1. The ratio can be guessed by calculating the optimal alignment for each skew factor. For each ratio the scaled input file is aligned using the fast no-split hybrid of algorithm 4 and algorithm 5. The alignment with the highest score is assumed to have to the correctly scaling for the input file.

For the default weighting function, the `iscore` is dependent on the ratio of the input span and reference, which is changed if only the input spans are stretched or compressed. Therefore the weighting function $w(a, b) = \min(\texttt{length}(a), \texttt{length}(b))$ was chosen, so that $\texttt{iscore}(a, b) = \texttt{overlap}(a, b)$. In the test database there was at least one case where the standard weighting function lead to guessing the wrong ratio, while $\texttt{iscore}(a, b) = \texttt{overlap}(a, b)$ resulted in the right skew factor.

For 118 subtitle files downloaded from an online subtitle database by movie id, 27 subtitle files had a wrong playback speed compared to the their reference video. This method was able to guess the correct ratio in every case. For the other 91 subtitle files with an already correct framerate, 3 subtitles were guessed to have non-unit ratio between the playback speeds. Comparing the highest score to the second highest score yielded an average increase of 3%.

Given a reference subtitle instead of the VAD spans from the movie, the framerate was corrected in every case. The highest score was on average about 50% higher than the second best score. This suggests an improved voice-activity detection might increases the accuracy and avoid the 3 incorrectly classified subtitle files.

2.6 Results

A test database containing 29 full-length movies in German, English and Japanese, was created to analyze accuracy of the algorithms on real subtitle and movie data. Perfectly synchronized English subtitles were picked to act as references. Additional 118 English subtitles for the movies with arbitrary synchronizations were then aligned using the different algorithms and compared to the respective reference subtitle.

The 29 references subtitles were obtained online using the API of the popular `OpenSubtitles.org` subtitle database [12]. The API provides searching subtitles using the hash of the movie file as well as listing other subtitle files for the same movie. By querying with a movie hash, a list of supposedly perfectly synced subtitle files for the requested movie file is returned. In practice the results were very inaccurate. Of 39 movie files

- 10 hashes could not be found
- 2 hashes returned badly timed subtitle files with a no-split offset of 3 seconds and 17 seconds
- 5 hashes returned the subtitles of a *different movie*

The 2 subtitle files were manually corrected and are included in the following tests. The problematic 5 wrong subtitle files were circumvented by analyzing all subtitles in the response. Only subtitles that referenced the movie which occurred the most in the response were accepted. The highest-scoring accepted subtitle was chosen as ground-truth subtitle for the movie. In the 5 cases this approach picked the right subtitle. This statistics highlights the importance of an offline synchronization algorithm if subtitles are obtained using this method.

To objectively compare the different configurations of alignment (no-split, split, high/low values of P, ...) only the 115 of the 118 subtitles where the framerate was guessed correctly are included in the generation of the following figures. For the wrong framerates, the resulting subtitles were unusable no matter the algorithm configuration. The discussion of optimal values for E and P are therefore chosen under the assumption that the framerate is correctly guessed.

A performance figures were created on a notebook with an Intel™i7-6500U processor and a Crucial SATA SSD.

Distance metric

In the following tests the possibly unsynchronized subtitles were synchronized to their respective movie or reference subtitle. The distance between corresponding lines after synchronization was used as indicator of the quality of the synchronization.

The distance of lines was defined as

$$\min(\texttt{length}(r \backslash a), \texttt{length}(a \backslash r)))$$

where $r \backslash a$ denotes the interval of r excluding the interval a. This distance metric is defined as 0, if and only if one line contains the other. This accounts for different lengths of the same line in the two subtitles.

Finding corresponding lines

The reference subtitles and arbitrarily synchronized input subtitles might differ considerably. A map of corresponding lines has to be generated for each relevant pair of reference and input subtitles to be able to measure the quality of the alignment.

The *similarity* of subtitle lines was calculated as edit distance (obtained by the Smith-Waterman-Algorithm) divided by the text length of the longer subtitle sentence.

Afterwards an ”Edit Distance” between the subtitle files itself was generated, again using the Smith-Waterman-Algorithm. If lines with a similarity over 80% were combined the score was increased by 1. Otherwise the score was not increased, as there was no penalty for ”deletions” or ”insertions”.

This yields a 1:1 mapping, 1:n mapping or n:1 mapping of some reference lines to some input lines while respecting their order in the subtitle files. Lines that are part of a 1:n or n:1 mapping are discarded and not considered for a corresponding line pair (mainly caused by split lines in one subtitle).

Of the 1:1 mappings the reference subtitle line was compared to all other input subtitle lines and vice versa. If there was at least one line line that yielded a similarity above 40%, this pair was also discarded. This prevents false pairings for common lines like ”Okay”.

In total only 25% of lines of the 118 subtitles are part of a pairing.

Discarding small spans given by the WebRTC VAD

Figure 2.3 shows a large discrepancy in the distribution of span lengths when comparing subtitle data to the result of the WebRTC VAD. While real subtitle files rarely include lines with a length of less than one second, most of the spans from the voice activity detection are smaller than half a second.

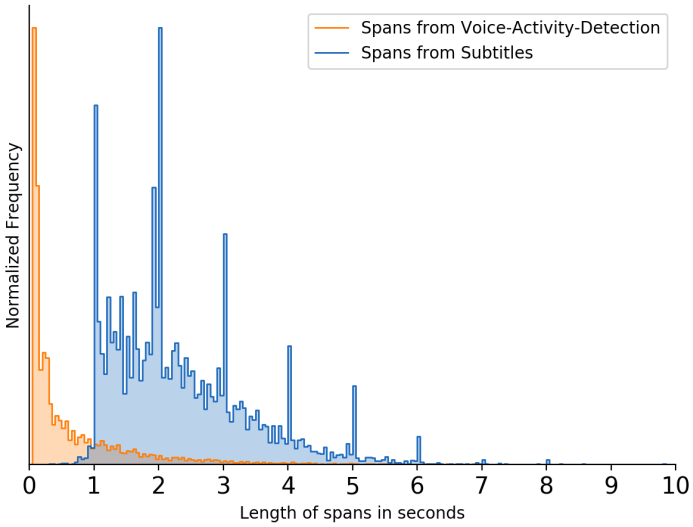

Figure 2.3: Lengths of subtitle spans and VAD spans

Inspecting the spans generated by the WebRTC VAD shows that it is very sensitive to noises like footsteps, opening/closing of doors, background music, etc. Interestingly, discarding smaller VAD spans actually *improves* the alignment accuracy (figure 2.4) as well as improving the runtime (since K is smaller). The optimum is reached when discarding spans shorter than 500ms, as 99% of all spans are within 1250ms of their corresponding reference and 90% of all spans are within 800ms of their corresponding reference.

All following statistics were therefore gathered with a minimum VAD span length of 500ms.

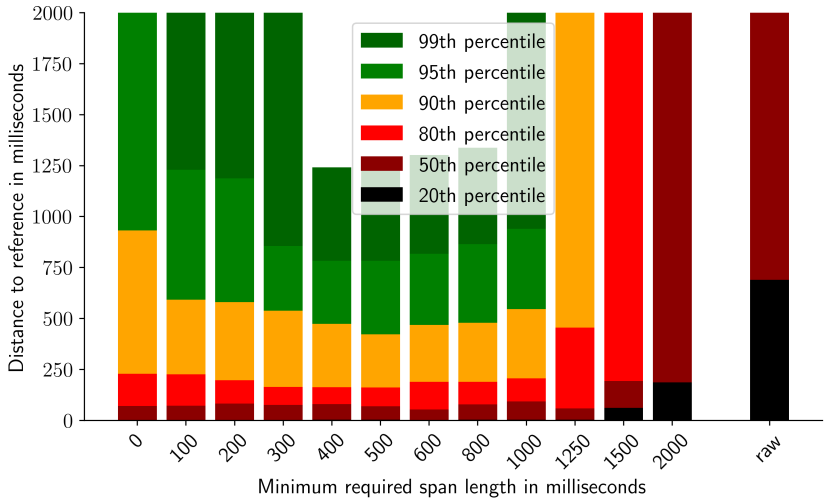

Figure 2.4: Alignment accuracy of algorithm 7 when discarding small spans from voice-activity-detection

Comparing approximation values E

Figure 2.5 shows that the approximation values E less or equal 1 do not influence the resulting alignment quality. For E between 2 and 5 the resulting alignment quality fluctuates slightly but stays within reasonable amounts. For approximation values above 5 the alignment quality degrades quickly. The approximation value generally influences alignments to audio data more than alignments to subtitle data.

Figure 2.6 shows the time required by the algorithm depending the approximation bound E. Using larger bounds clearly reduces the required runtime of the approximate optimal split algorithm. A good speed-to-quality ratio is given by values from 2 to 4.

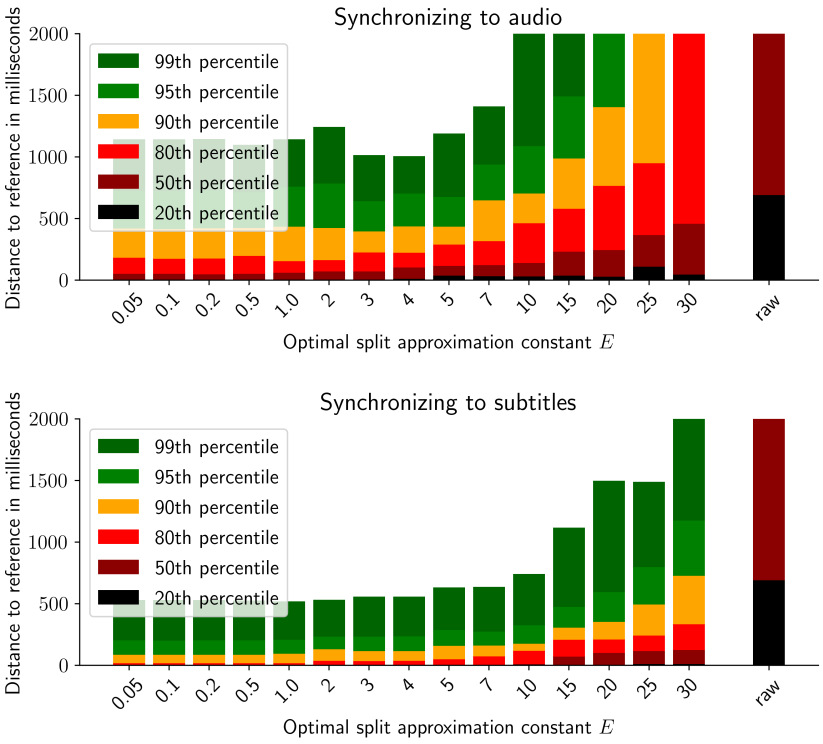

Figure 2.5: Alignment accuracy for different approximation bounds E

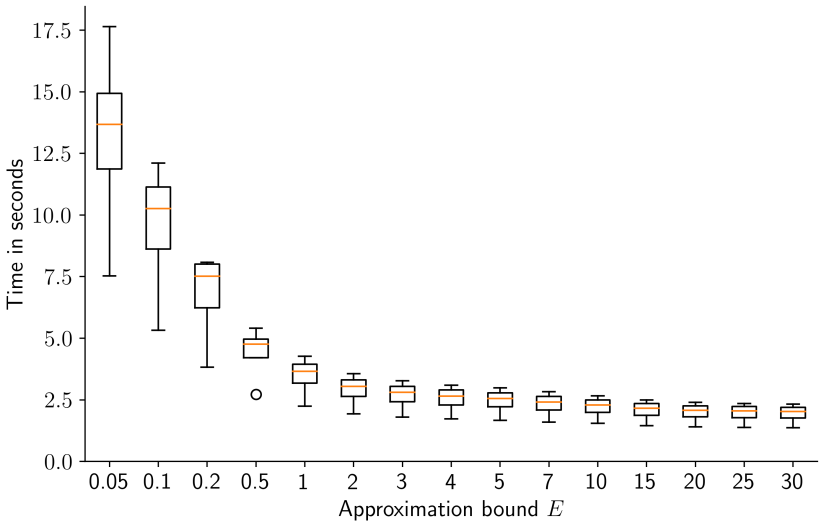

Figure 2.6: Runtime of approximate optimal split algorithm 7 with approximation bound E

Comparing split penalties P

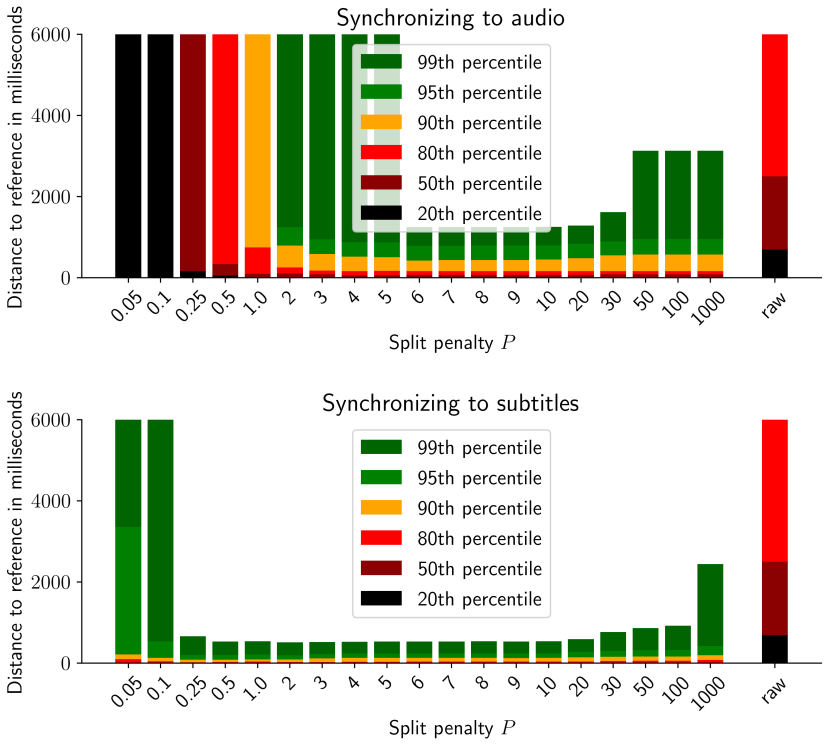

Figure 2.7: Alignment accuracy for different split penalties

Figure 2.7 shows the accuracy for different split penalties P. Values below 20 yield the best results both when aligning to audio and aligning to subtitles.

When choosing the the split penalty too low, the split algorithm will introduce splits in the wrong positions which degrades the alignment quality. A large difference for this minimum split penalty can be observed between aligning to audio and aligning to subtitles.

The accuracy decreases for audio data when P is lower than 6, and creates large errors for values lower than 1. When aligning to the reference subtitle instead, on the other hand, a lower alignment quality can not be observed until P is lowered to 0.1.

Even when allowing more splits for subtitle-to-subtitle alignments the the same offset is chosen for each split segment. This suggests that the similarity reference subtitle spans and the input spans is much higher than the similarity between the WebRTC VAD spans and the input spans.

Comparing algorithm variants

Figure 2.8 show the alignment accuracy for all combinations of the following configurations:

- using a no-split algorithm or a split algorithm (algorithm 7; $E = 2$, $P = 6$)
- correcting framerate differences (denoted by *FPS*) and not correcting framerate differences

- aligning to the reference subtitle or aligning to the WebRTC VAD spans from the audio (with a minimum length of 500ms)

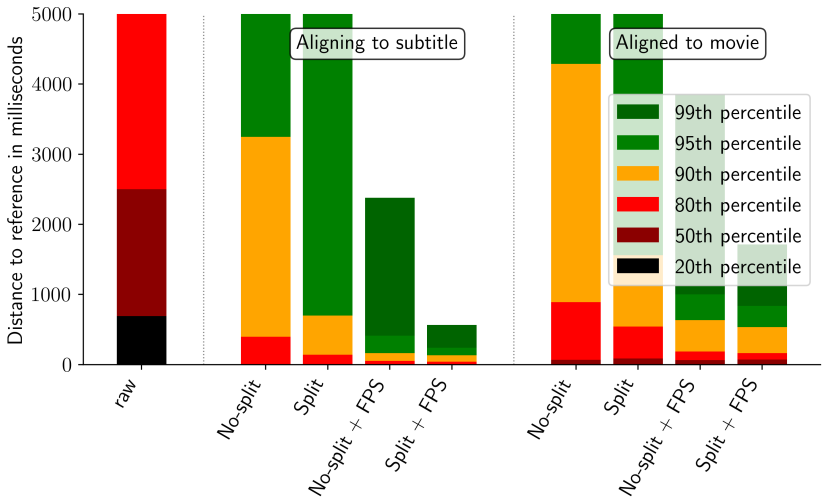

Figure 2.8: Alignment accuracy for different algorithm variants

The no-split alignment to the reference subtitle with framerate correction yields exceptional results (see figure 2.8). 99% of all lines are within 800ms of its target position and 95% of all lines are even placed within 400ms of their corresponding reference span. Switching to a no-split alignment, the accuracy degrades for 4% of the spans, while remaining about the same for the best 95% of the spans. Without performing framerate correction, the alignment quality degrades severely. That the difference between the no-split and split alignment is higher if no framerate correction occurs, suggests that the split algorithm can substitute as a crude framerate correction.

Aligning the subtitles to the audio yields very similar data to aligning to the reference subtitle. All subtitle spans are generally 300ms to 500ms farther from the reference spans compared to directly aligning to the reference subtitle.

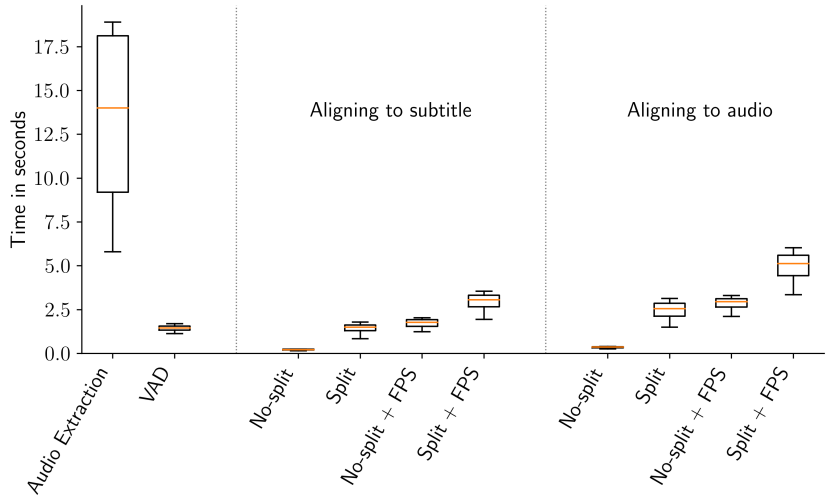

Figure 2.9: Runtime comparison of algorithm variants for full-length movies

An important criterion for the alignment algorithms is their total runtime. All alignment algorithms finish within 8 seconds (see figure 2.9).

Generating the optimal no-split alignment done in only 200 to 300 milliseconds. Generating the approximate optimal split alignment takes about 2 seconds. Correcting the framerate take additional 1.5 to 2 seconds (as this computes the no-split alignment 7 times). The spans for the audio are generated by the WebRTC VAD module in 1 to 1.5 seconds. Extracting and re-sampling the audio to 8000 samples per second using *FFmpeg* [13] generally takes longer than any alignment algorithm and is mainly limited by the speed of the hard drive. The complete process of aligning a subtitle to a 2-hour movie therefore takes less than 30 seconds in any case.

Classifying subtitles

A ”good subtitle” is defined here as

- less than 25% of lines having a distance of at most 300ms
- less than 70% of lines having a distance of at most 500ms
- less than 95% of lines having a distance of at most 1000ms
- less than 99% of lines having a distance of at most 1300ms

The more relaxed constraints to the ones given by Netflix' style guide are due to the reference file itself being slightly inaccurate. Stricter conditions result in a large percentage of subtitle files to be classified as ”bad”, even though they are not distinguishable from the reference subtitle. Figure 2.10 also includes the subtitles in the database where the framerate was guessed incorrectly.

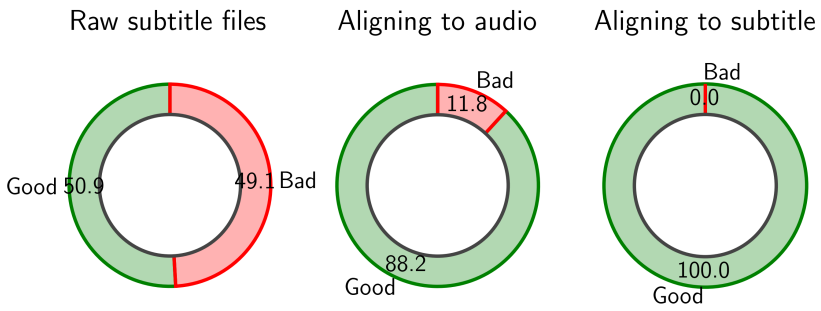

Figure 2.10: Alignment error rates using algorithm 7 ($P = 6$, $E = 2$) with framerate correction

Chapter 3

Conclusion

Synchronizing subtitles to reference subtitles using the presented algorithms shows exceptional results. All 118 subtitles in the database could be aligned perfectly.

The more complicated problem of synchronizing subtitles to the audio of a movie exhibits a higher error rate of about 12%, which is still well above the 50% error rate of using possibly unsynchronized subtitles from online databases. The largest problem with movie alignments is the low confidence when guessing differences of playback speed between the movie and the subtitle file. To solve this problem, a more accurate voice-activity-detection than the one of the WebRTC module is needed. Since the same alignment algorithm is used for audio-synchronization and subtitle-synchronization, a good voice-activity-detection module likely also results in all subtitles being perfectly aligned with the movie.

Since the complete process of aligning a subtitle file takes less than 30 seconds for full-length movies, it is suitable for real-time correction in video players.

An approach that was not discussed in this paper is exploiting a closer relationship between the voice-activity-detection and the alignment algorithms. The alignment algorithms in this thesis assume binary decisions for each segment of time. When analyzing segments of audio, the resulting probability of a segment containing speech is a real value. Using this value directly instead of its relationship to a certain threshold provides the alignment algorithm with much more fine-grained data. Unfortunately, the algorithm speed optimizations rely on the low number of switches between the speech and nonspeech segments. Using the probabilities directly therefore needs additional research.

Another way to achieve more robust subtitle correction is to include content similarity in the the weights between reference spans and input spans. For subtitles as reference data, the edit distance of the respective lines could serve as a basis. For audio data as reference data, for example syllables could be analyzed. Although this approach is not language-agnostic, it might yield superior alignment quality if a same-language synchronization is performed.

Bibliography

[1] Timed text style guide: General requirements. `https://partnerhelp.netflixstudios.com/hc/en-us/articles/215758617-Timed-Text-Style-Guide-General-Requirements`, 2019. accessed September 28, 2019.

[2] Source code of the webrtc voice-activity-detection. https://github.com/dpirch/libfvad, 2019. accessed September 23, 2019.

[3] Ying Dongwen, Junfeng Li, Qiang Fu, Y. Yan, and Jianwu Dang. Voice activity detection based on a sequential gaussian mixture model. *APSIPA ASC 2011 - Asia-Pacific Signal and Information Processing Association Annual Summit and Conference 2011*, pages 861–866, 01 2011.

[4] Z. Shen, J. Wei, W. Lu, and J. Dang. Voice activity detection based on sequential gaussian mixture model with maximum likelihood criterion. In *2016 10th International Symposium on Chinese Spoken Language Processing (ISCSLP)*, pages 1–5, Oct 2016.

[5] F. Eyben, F. Weninger, S. Squartini, and B. Schuller. Real-life voice activity detection with lstm recurrent neural networks and an application to hollywood movies. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 483–487, May 2013.

[6] Bernhard Lehner, Gerhard Widmer, and Reinhard Sonnleitner. Improving voice activity detection in movies. In *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

[7] Ronald J Baken and Robert F Orlikoff. *Clinical measurement of speech and voice*. Cengage Learning, 2000.

[8] Rust language documentation of `Vec::sort_unstable()`. `https://doc.rust-lang.org/std/vec/struct.Vec.html#method.sort_unstable`, 2019. accessed September 18, 2019.

[9] Orson Peters. Pattern-defeating quicksort. `https://github.com/orlp/pdqsort`, 2019. accessed September 18, 2019.

[10] Rust language documentation of `Vec::sort()`. `https://doc.rust-lang.org/std/vec/struct.Vec.html#method.sort`, 2019. accessed September 18, 2019.

[11] Implementation of stable merge sort in the rust standard library. `https://github.com/rust-lang/rust/commit/721609e4ae50142e631e4c9d190a6065fd3f63f7`, 2019. accessed September 18, 2019.

[12] Documentation of opensubtitles' xmlrpc api. `https://trac.opensubtitles.org/projects/opensubtitles/wiki/XMLRPC`, 2019. accessed September 28, 2019.

[13] Ffmpeg: A complete, cross-platform solution to record, convert and stream audio and video. `https://www.ffmpeg.org/`, 2019. accessed September 28, 2019.