Applied Forced Alignment for Language Analysis

A N D R É A L G O T S S O N

KTH Computer Science
and Communication

Bachelor of Science Thesis
Stockholm, Sweden 2010

Applied Forced Alignment for Language Analysis

A N D R É A L G O T S S O N

Bachelor's Thesis in Computer Science (15 ECTS credits)
at the School of Computer Science and Engineering
Royal Institute of Technology year 2010
Supervisor at CSC was Johan Boye
Examiner was Mads Dam

URL: www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2010/algotsson_andre_K10065.pdf

Royal Institute of Technology
School of Computer Science and Communication

KTH CSC
SE-100 44 Stockholm, Sweden

URL: www.kth.se/csc

Applied forced alignment for language analysis

Abstract

This article is aimed towards investigating the prerequisites of an application for language analysis. The idea is to use an audio recording of a human reading a written text, and by the means of *forced alignment* enabling a computer user to easily retrieve the recorded pronunciation of individual words and sentences. The functionality and design of the envisaged application is defined, the research in the field is summarized, a number of existing state-of-the-art solutions and relevant algorithms are explained and assessed in the context of the functional requirements of the application and the findings are discussed.

Tillämpad forced alignment för språkanalys

Sammanfattning

Denna artikel undersöker vad som krävs för att få till stånd ett interaktivt språkanalysprogram. Tanken är att utvinna uttalet på separata ord och meningar från en ljudinspelning av en människa som läser upp en skriven text, och direkt presentera resultatet för en datoranvändare. För att åstadkomma detta används s.k. *forced alignment* på olika detaljnivåer. Det tänkta programmets funktionalitet och design presenteras, följt av en sektion som sammanfattar forskningen inom området. Ett antal lösningar och algoritmer hämtade från den allra senaste forskningen beskrivs sedan i detalj. Allt detta är analyserat och utvärderat utifrån de funktionella kraven.

Contents

1. Introduction

This essay is a part of a bachelor degree project in computer science at *KTH, Sweden*. It examines the prerequisites for creating an application that dynamically aligns text with audio using the latest findings in the area of *forced alignment*. The application will enable a computer user to retrieve the pronunciation of *any* text string in a text. The pronunciation will be extracted from a plain audio file, containing a human-read version of the text. Such text/audio-combinations are getting increasingly common by the rising popularity of audio-books, news reports with transcripts and subtitled movies. The goal of the application is to avoid aligning parts of the text that is not of immediate interest to the user, and the alignment procedure is therefore broken up into several parts corresponding to the needed level of detail.

2. Specification of application

The envisaged application that will be the focus of this project is a system that can extract individual words from a spoken text and make it accessible to a computer user (for instance, a language learner). The system will be built upon the concept of forced alignment (see *Section 4:1*), which means that it requires two versions of each text: one human spoken version, and one written transcript. An example of a type of text that suits this application is an audiobook, where an actor artistically reads a novel word-for-word. Other types of texts such as news reports and subtitled movies naturally require more robust strategies for handling various background noises.

- The application requires spoken audio together with an exact transcript as input.
- The application should be able to map every word to its pronunciation using an effective forced alignment algorithm.
- The application should enable the user to select any excerpt from the text, and automatically retrieve the part of the audio that corresponds to the text. See *Figure 2:1* below.
- The application should be able to hierarchically divide the audio into chapters, sentences, words and letters as depicted in *Figure 2:2*.
- Ideally, the system should work in real-time with little pre-processing, only aligning text that is of immediate interest for the user.

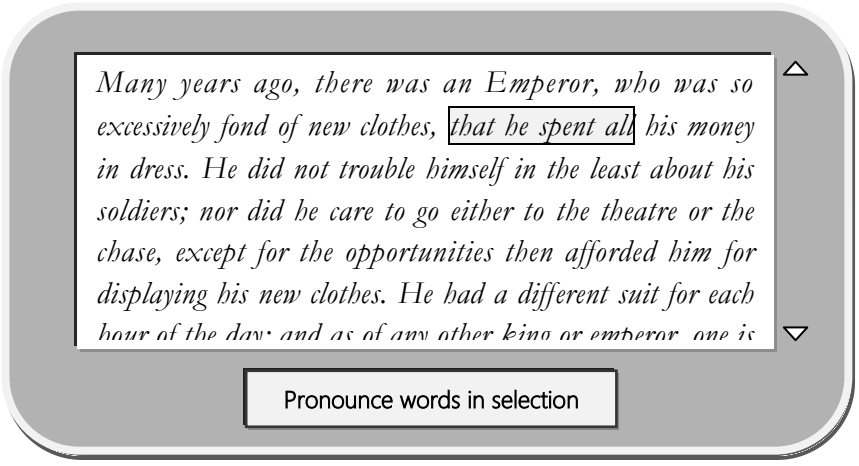
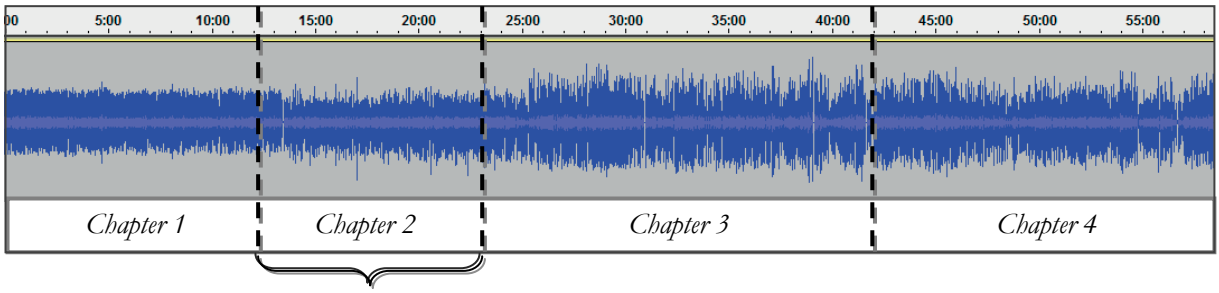
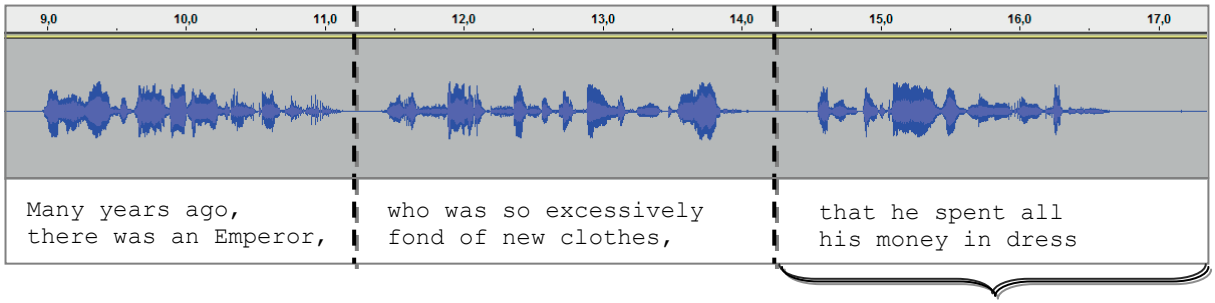


Figure 2:1 – Example application.

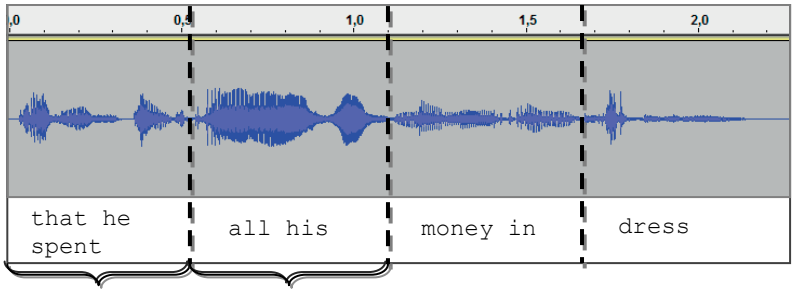
Phase 1 – Chapter-level alignment



Phase 2 – Sentence-level alignment



Phase 3 – Word-level alignment



Phase 4 – Phoneme-level alignment

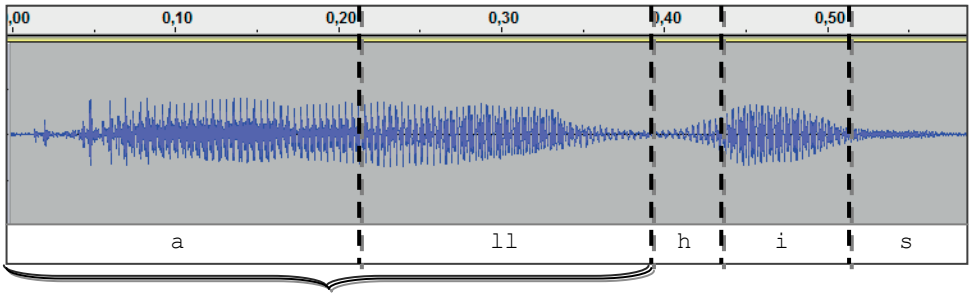


Figure 2:2 – Increasing granularity of the aligner

3. Application design

Figure 3:1 shows a use case that demonstrates the minimal interaction required by the application:

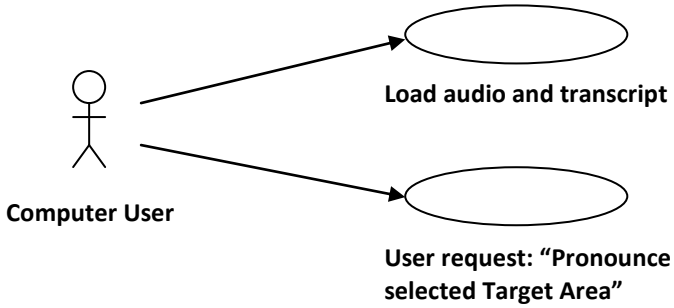


Figure 3:1 – Use Case showing user interaction with application

Figure 3:2 shows how the actual alignment can be distributed between the procedures during usage:

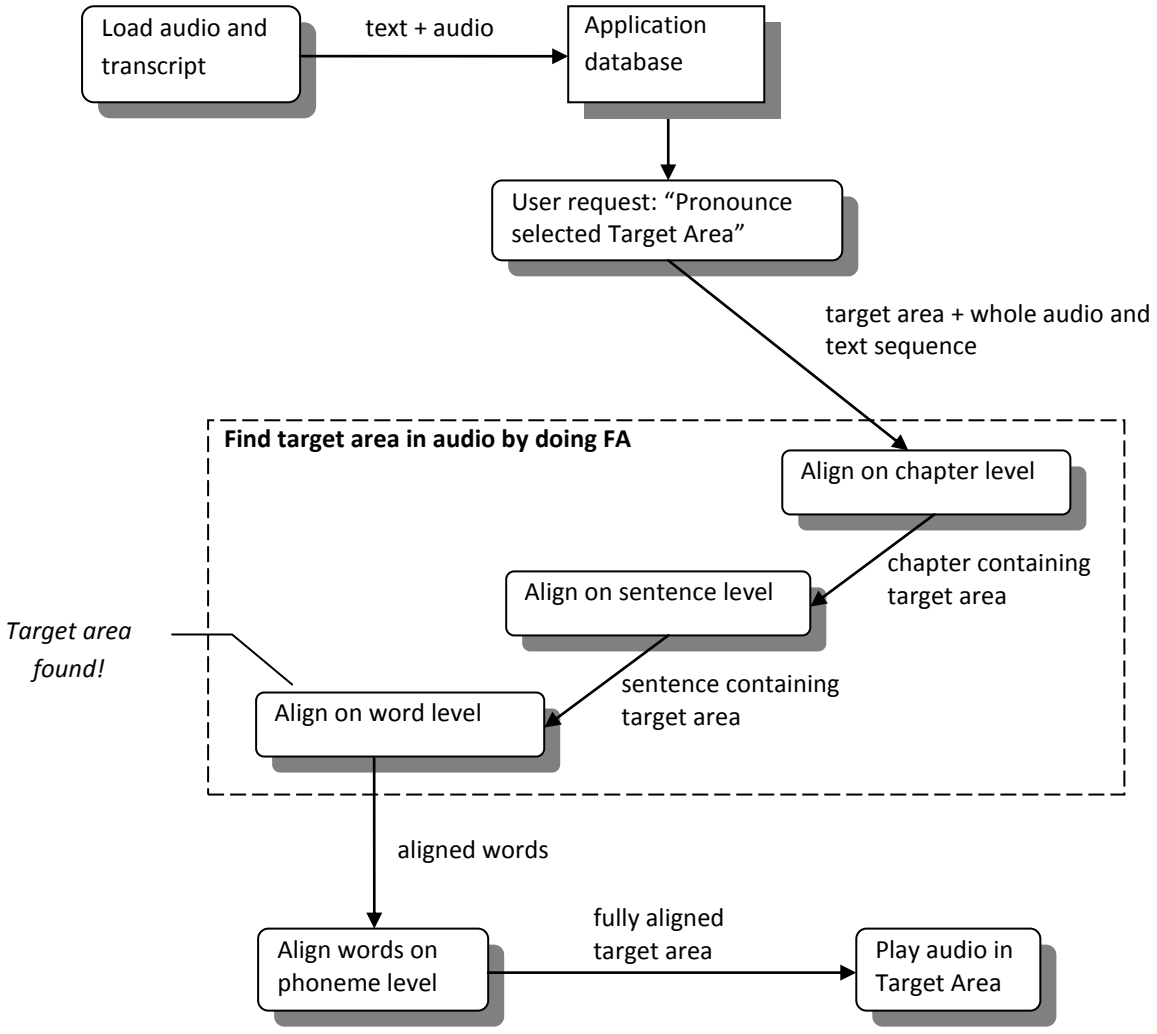


Figure 3:2 - Process model showing operations of the application

4. Background

4.1. Description and applications of Forced Alignment

Forced alignment (FA) denotes the process of aligning spoken audio to a given text, so that each written word gets connected to its corresponding pronunciation, see *Figure 4:1*.

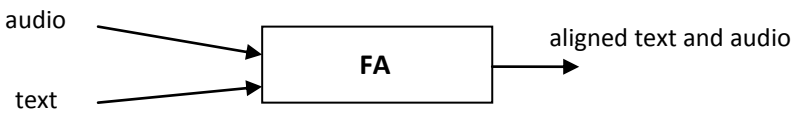


Figure 4:1 – Forced alignment

FA is used as an intermediate step in order to create comprehensible input for the training of speech synthesis engines. As opposed to the aim of this project, working with speech synthesis also requires further analysis and processing of many other parameters in great detail, such as prosody of speech and voice timbre.

The most obvious application area is when creating speech engines built upon *Unit Selection Synthesis*. The idea is to build databases with words and their pronunciation using FA, and synthesize a voice by chaining the sounds of the words together.

Some research on FA algorithms has been focused on indexing audio and its corresponding transcription, for making video information searchable through web search engines. Those algorithms are specifically designed to handle long audio segments (>1 h), and errors of about 1-2 seconds are often acceptable. [2]

The FA that this article discusses differs from other applications on the target area of alignment. Its aim is to only align requested portions of the audio; instead of aligning every single word. This may enable real-time interaction for users who want to retrieve the pronunciation of a certain sentence.

4.2. Strategies for doing Forced Alignment

As of today, there exist some different strategies for aligning text to speech. The first algorithms handled the files sequentially in great detail which gave good precision. But there were often a risk of the algorithm getting lost with no ability to recover when unexpected situations occurred. The remedy to this problem was to design recursive algorithms that tried to minimize those risks by always picking the “easiest” area to align, see [2]. The result was that difficult areas could be processed at a time when they were less likely to venture the whole process. For a demonstration of the concept, see *Figure 2:2*.

The concept of *Hidden Markov chain Models (HMM)* has been widely used as a statistical approach to FA. It is built by analyzing large sets of training data together with a number of language models. The result is a model that can compute the probability for a given word to appear at some place in the text. Forced alignment then becomes a matter of processing the audio signal, matching it to the text and finding the most probable transitions via the HMM.

HMM has also been used on phoneme level. Working at this level of detail is a delicate task because there are rarely any precise boundaries between phonemes in normal speech. Therefore, automatic phoneme alignment is always based on subjective assumptions, striving to reach consensus with a human-created alignment. [4]

Another approach has been to search for prosodic breaks and then divide the audio into speech and non-speech, creating a chain of relatively small utterances. An HMM is built from statistical data on mean word length and likelihoods of breaks occurring after each word; and the most probable transitions through that HMM represents the alignment [5]. This approach has the obvious drawback that it will have difficulties handling large deviations from standard prosody, as could be the case when a text is read in a theatrical way, or in the presence of background noises etc.

One interesting approach is to pre-process the text using a speech-synthesis engine. Then the *machine-created signal* is aligned to the original human-read audio, and the result is translated to the corresponding positions in the text, see *Figure 4:2*. Using speech-synthesis has many advantages, since there is no need for training the aligner beforehand (as is the case with any *HMM*-method), and that a good algorithm can be used on materials of any language that has a decent speech-synthesis engine. This approach is explained and assessed in [6].

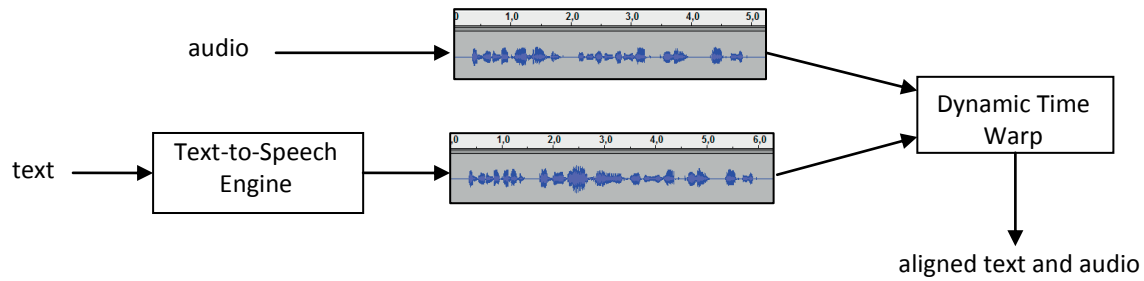


Figure 4:2 – Speech-synthesis based forced alignment

4.3. Modeling speech

The most common way of extracting data from speech recordings is by using *Mel Frequency Cepstral Coefficients, MFCC*. The signal is transformed a number of times, and divided into discrete time-windows of the signal. The information in each of these parts then gets stored in an *MFCC Feature Vector*, and statistical models are used to calculate its most likely corresponding phonemes [1].

To give a brief overview of what is done, the audio is transformed into its *cepstrum*[*], with respect to the *Mel scale* (the Mel scale models human speech in a useful way). Usually, the first 12 coefficients of the retrieved Mel Frequency Cepstrum and its first and second derivative are extracted and put into the vector. This results in a total of 36 coefficients in each MFCC Vector.

These vectors now contain enough information to reveal the underlying phoneme (or at least give a list of potential candidates).

[*] Algorithmic definition of cepstrum, FT denoting the Fourier Transform:
signal → FT → abs() → square → log → FT → abs() → square → power cepstrum

Since phonemes are seldom of equal length and equally spaced, the time-windows need to overlap. Otherwise the vector would risk missing some crucial information which just happened to end up in the adjacent window.

For instance, as in [3], the overlap can be done by employing *Hamming windows*, which include information from adjacent windows, but assign them a lower weight. See *Figure 4:3*:

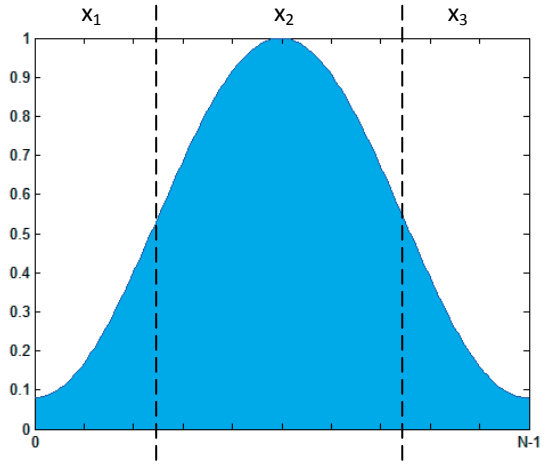


Figure 4:3 – A hamming window for time frame x_2

There are many variations as to how the MFCC vectors are composed, and how the overlapping is handled. The concepts discussed here are supposed to give a general overview of the problem.

4.4. The Viterbi algorithm

Often referred to in articles is the Viterbi algorithm. In this context, the algorithm is used to retrieve the *most probable* chain of phonemes by matching the information of the MFCC Vectors (or another structure depending on the model), to the Hidden Markov chain Model.

This is done effectively by using dynamic programming techniques.

4.5. Hidden Markov chain Models (HMM) and FA

Using HMM together with the Viterbi algorithm is the most well-known approach to FA. However, the training of the HMM is cumbersome and it has very large memory requirements and time complexity for long audio segments. On top of this, the algorithm can easily get permanently lost if the provided audio is too noisy or departs too much from the transcript. [2]

Because of these drawbacks, a lot of research has been done (see references) to find alternative approaches, which do not possess the flaws, but still give comparable (or better) results.

This article will therefore not elaborate upon the theory behind HMM, but the reader should be aware of the importance of HMM in other speech processing applications (such as Text-to-Speech engines).

4.6. Introduction to n-gram models

A practical demonstration of the usage of specialized n-gram models in the context of forced alignment is provided in *Section 5.2*.

N-grams are simply a dictionary consisting of word tuples, with *n* denoting the number of words.

One of the most extensive n-gram collections was made public by *Google* in 2006, and can be obtained from the *Linguistic Data consortium*, see [14]. *Table 4:1* is an excerpt from the Google 3-gram collection, where the number within the parentheses denotes how many times the tuples appears in their texts:

Table 4:1 – Excerpt from Google 3-gram corpus

ceramics collected by (52)
ceramics collectables fine (130)
ceramics collectible pottery (50)
ceramics collectibles cooking (45)
ceramics collectibles, (144)

One of the key aspects that makes n-grams useful in FA is the ability to convert a long sequence of phonemes into a sequence of words based upon the probability of the words showing up. Consider the phonetic transcription "ˈfrē'īs-ˈkrēm". A dictionary look up of the possible underlying words may render two alternatives: "Free ice cream" and "Free I scream", both having almost the same pronunciation. Consulting a *bigram* (an n-gram, where n=2) dictionary would probably render the result that "ice cream" and "I scream" are almost equally probable (depending on the source from which the model was trained). However, comparing the probability for "Free ice" and "Free I" should naturally result in the former being much more probable. Thus the phonetic transcription can in this example safely be translated to "Free ice cream". This demonstrates how words can be retrieved using statistics, without any further knowledge of the local context of the text and the speaker.

In order to use a bigram model on a longer sequence of words, an algorithm for adding the probability for each word pair into a total probability must be devised. This feat is elegantly handled by the Viterbi algorithm (see *Section 4.4*).

5. Construction of aligner

Before exploring the strategies for each alignment phase of the application, it is worth mentioning that one might want to pre-process the text before starting the FA. This can include classifying the text for choosing the right alignment procedure, building a concordance, sorting out passages that are too noisy [10] and compute the energy of the signal (as is used on sentence level). Determining what actions are necessary should be done after some experimentation as it is important to avoid processing areas that are not interesting for the user.

5.1. Chapter-level alignment

5.1.1. Definition of problem

Chapter-level alignment is especially applicable if the audio and the text are divided into larger chunks with significant boundaries, such as long periods of silence or the actual end of a particular audio file. Following the audiobook terminology, the word *chapter* has been chosen, but it could also refer to other divisions, such as a change of speaker etc.

One obvious example where chapter-level alignment would be particularly useful is when every part of the text starts with a common identifier, such as the word “Chapter”, as is very common in literature. See *Figure 5:1:1*:

Chapter 3	Text text text.	Text text text.
	Text text text.	***Chapter 4***
Text text text.	Text text text.	
Text text text.	Text text text.	Text text text.
Text text text.	Text text text.	Text text text.

Figure 5:1:1 – Example of significant boundaries (identifiers)

Chapter-level identifiers could be a word (as in the example above) or a numeral, together with or solely consisting of a sequence of newline characters (or something similar).

Aligning on chapter-level has the obvious advantage that divisions between chapters are relatively easy to align. It is a matter of finding a significant boundary or pause in the audio and in the text. After having done this step, the material is split up in smaller parts, and the part that contains the user-selected target area is sent for further processing. If the following procedures are based upon building dictionaries from the text (as is the case with the n-gram methods), their operations will be significantly speeded up because of the reduced size of the material.

The research topics “audio segmentation” and “speaker diarisation” may be of interest for getting a deeper understanding of the problem.

5.1.2. Proposed solutions

At the chapter-level, the aligner must work with the whole text, which could span for several hours. It is therefore of very high importance that the algorithms are as fast as possible, and do not spend time doing operations that would rather be done in a succeeding phase.

The problem could be reduced to finding long silences in the audio, and matching them to places in the text which have a break identifier. For doing this, the algorithm presented in *Section 5.2.2.1* and *Section 5.2.2.2* can be used, with a longer duration of the silences, a very low tolerance for noise and a simple string matching algorithm for finding standard identifiers.

Creating a more flexible chapter-level aligner may require further investigations of the nature of speech. This could include observing patterns among speakers, such as a tendency to raise the pitch of the voice when new topics are introduced etc. Some useful observations are discussed in [11].

5.2. Sentence-level alignment

5.2.1. Definition of problem

When doing sentence-level alignment, the application should search for “anchors” in the text and audio. These are specific parts of the text that for some reason are considered *easy* to align. Operating on sentence-level therefore means to divide the text into a set of sequences of words.

By aligning the *easiest* parts first, the algorithm will be less likely to align a segment incorrectly; leaving the troublesome parts to be taken care of by a later iteration or a more specialized algorithm (that can afford to spend more time on the task). Thus, at this level, the aligner should look for relatively long silences in the audio, analyze their environments, and find the matching positions in the transcription.

5.2.2. Proposed solutions

The very nature of speech suggests that the *easiest* parts to align are on places where the speaker is silent. This is due to the fact that finding boundaries between phonemes in a word is a difficult and subjective task as discussed in [4], whereas silent areas in audio often occur at punctuation marks, which are naturally easy to detect.

Therefore, it is a well known strategy to search for silences as possible breaking points in the first pass, which is the case in [2], [8] and [9].

When the silences have been found, the text needs to be aligned accordingly. This is a non-trivial problem that may be approached using combinations of speech recognition and n-grams, [2] [9]. The approach taken here strives to minimize the number of words processed, leaving all the details to the word-level aligner.

5.2.2.1. Finding silences in audio

An easy algorithm for detecting silences in linear time is to analyze the energy of the audio signal sample-wise and mark every segment that stays below a certain energy threshold as silent. The energy, E_s, of a signal *x(t)* is defined as:

$$E_s = \int_{-\infty}^{\infty} |x(t)|^2 \, dt \qquad (1)$$

An algorithm for doing this has been taken from the open source project *Audacity* (Audacity 2010) [12], which is a free system for processing and manipulating audio. It can be summarized as follows:

I. Decide an energy threshold that is considered as silent (this energy may be higher in the presence of background noises etc.).
II. Decide the minimum duration, *D*, of a silent part.
III. *For each sample in the audio sequence:*
    Analyze the energy of the sample and determine whether it is silent or not.
    If sample is silent:
        Increment a silence duration counter, N.
    Else
        Mark the whole previous sequence of silent samples as "silent" if N > D.
        Set N=0.

The concept is illustrated in *Figure 5:2:1*; if the minimal silence duration is set to *D=3*, only the left-most silent area will be marked as "silent":

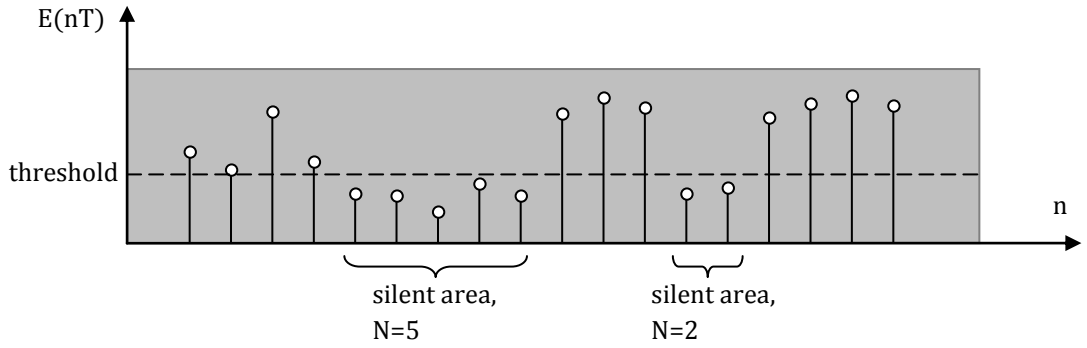


Figure 4:2:1 – The energy of an audio signal, where everything below the threshold level is regarded as silent

Finding good values for the minimum duration *D* and maximum energy threshold of silent areas requires some experimenting, as it depends on the source and quality of the audio, and possibly also on the prosody of the speaker. A high sampling rate may significantly improve the reliability of the algorithm since it can detect sudden variations better, but will naturally affect the performance negatively.

5.2.2.2. Local forced alignment with n-gram models

- *Description of algorithm*

When the silent areas have been marked, they should be aligned to the textual transcription.

Inspired by the method employed in [2], a *phonetic dictionary* is built from the transcript. This dictionary takes a string of phonemes and returns every word (that can be found in the transcript) that has the given pronunciation. In addition, bigram and trigram language models are constructed from the transcript. Doing this at a pre-processing stage may save some time, depending on the implementation and the operating environment.

When the models are ready, the following algorithm can be used:

- Go to the next silent area in the audio (elaborated upon in the previous section).
- Analyze the surrounding words by creating sequences of MFCC Vectors.
- By the use of some phoneme classification algorithm, each vector gets assigned a number of possible phonemes, thus rendering a number of unbroken strings of phonemes, see *Figure 5:2:3*.
- All words in the current area that match the phonemes are retrieved from the phonetic dictionary.

At this stage, the audio and the data could be visualized as in *Figure 5:2:2*. Since only a minimal number of words are processed, the words in the outskirts remain unknown. Note that the words *"so"* and *"sow"* may have the same pronunciation.

Figure 5:2:2 – The words surrounding the silence are retrieved

Now, the n-gram models are consulted. By using the Viterbi algorithm, the *most probable* chain of words is found. A correctly trained bigram model should be able to tell that the word pair "was so" is more probable than "was sow".[†]

When the surrounding words have been investigated, their position in the original transcript needs to be found. This can be done with a simple string finding algorithm (such as the Knuth-Morris-Pratt algorithm or the Boyer-Moore algorithm). Another approach could be to build a concordance

[†] By reducing the dictionary to only contain the words found in the transcript; the word "sow" would probably not have turned up as a possible alternative in the first place. Doing dictionary reductions may therefore prove to be particularly effective if the text is short, or if it contains a limited number of unique words. These conditions can easily be attained by dividing the text with the use of chapter-level alignment, and build the initial dictionary on chapter-level.

database of the transcript, enabling very fast look-up time. Explanations of these methods can be found in the literature.

Now the alignment problem has been reduced to finding the exact position of the silent area, which is a trivial task since the words now are familiar. The whole procedure is visualized in *Figure 5:2:3*, where the silent area is aligned to the position "30" in the transcription:

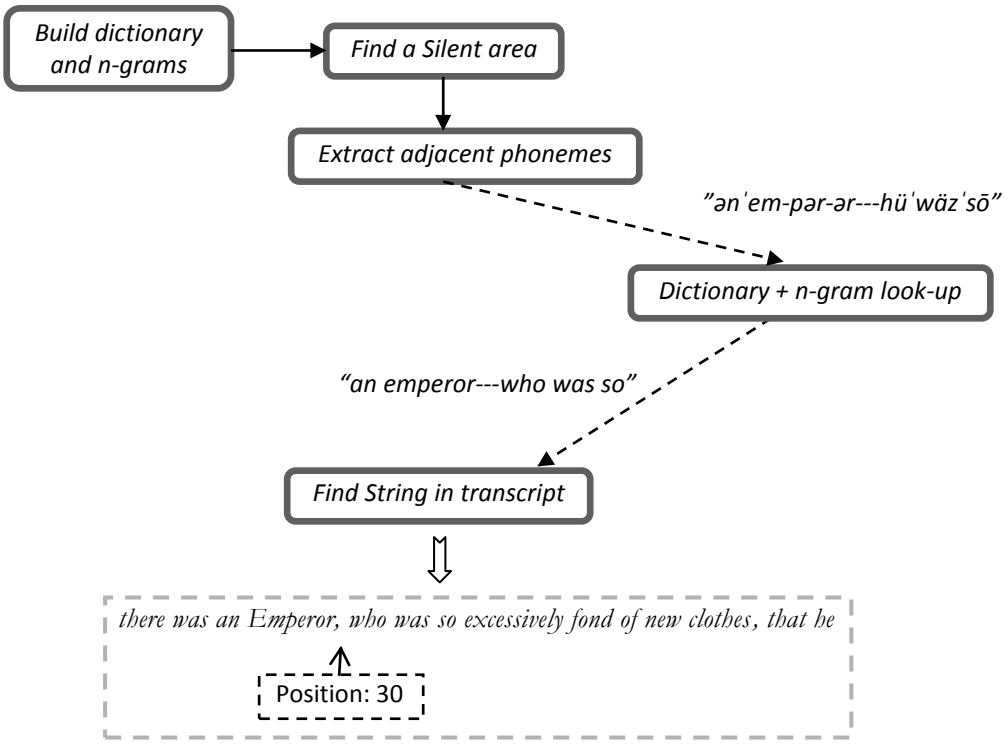


Figure 5:2:3 – The process of doing local forced alignment around a silent area

If one of the steps in the n-gram based algorithm fails (for instance, a wrong phoneme is chosen, leading to an incorrect word choice), the algorithm should be able to find the *most probable* alternative in the transcription – possibly by looking at the other surrounding words and ignoring the "difficult" word.

If the algorithm fails to find a reliable alignment, the particular place in the audio can be marked as "hard" – leaving it to a later iteration. It is important to not dwell on difficult areas for too long, as there are more powerful algorithms standing in line to fix the details.

- *Comments about sentence-level strategies*

Aligning at sentence-level is at its core a process of aligning the easiest areas in the audio as fast as possible. The process that was described in *Figure 5:2:2* and *Figure 5:2:3* focused on aligning the silent area between "emperor" and "who"; but in order to perform that, three other words were analyzed. Since the work already has been done, it could in some cases be wise to also align these auxiliary words before going on to the next area.

However, since the stated mission of this application is to only align the parts that are immediately necessary for retrieving the selected text, one should refrain from doing such operations (which naturally could slow down the overall alignment speed with some constant factor).

Therefore, an algorithm that aligns the surrounding auxiliary words should only be utilized when the aligner has reached an area that is reasonably close to the target area[‡].

At sentence level, it suffices to only get one single alignment right, and therefore it is acceptable to use light-weight language models. If the model cannot align a specific area, the area is too "hard".

When going down to word-level, more complete language models are used, together with techniques such as dynamic programming that aligns many words in one go.

5.3. Word-level alignment

5.3.1. Definition of problem

When the sentence-level alignment has been done, the text has been divided into sequences of unaligned words. The boundaries between words cannot always be expected to be found by a simple silence detection algorithm, as was one of the fundamental premises of the previous step. Therefore, a different strategy must be devised that guarantees that every alignment is the *easiest* possible.

5.3.2. Proposed solutions

5.3.2.1. Finding small silences

One might want to examine the possibility of using the silence detection algorithm that was used in the sentence alignment phase, with a smaller value of the minimum duration *D*, and possibly with a modified threshold level. Using this approach will probably require some experimenting before the results become reliable. Since all the *easy* areas were exploited in the foregoing step, one may also need to add some parameters that helps to detect "non-silent" divisions between words.

5.3.2.2. Forced alignment with n-gram models

One suggested approach is to use forced alignment with n-gram models (described in *Section 5.2.2*) recursively, with the following modifications:

- The phonetic dictionary and the n-grams are reduced to only contain the words that are part of the current interval.
- Use a large-dictionary speech-to-text engine to produce a hypothesis text of the sequence.
- Now, there are two texts, the original transcript and the hypothesis text. These are now aligned to each other using a dynamic programming algorithm.

Naturally, the hypothesis text will not match the transcript perfectly, and therefore the aligner should be able to detect areas that are likely to be correctly aligned. This is done by matching the

[‡] Recall that the target area is the specific part of the text was manually selected for retrieval, see *Figure 3:1.*

texts to each other, marking every sequence of $\geq N$ consecutive words as correct. Consider the example in *Figure 5:3:1*, where the speech recognizer has mistranslated the words "that" and "in dress". Between those words there are five correctly transcribed words. If $N \leq 5$, these words are marked as belonging to an "island of confidence". The succeeding iterations therefore focus on aligning the areas in between the islands (reducing the dictionary and possibly the value of N for each recursion).

```
      ┌─────────────────┐
 that┆he spent all his money┆in dress. He did not
     ┆                      ┆
there┆he spent all his money┆undressed. He did not
     └┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┘
```

Figure 5:3:1 - A transcription and a synthesized hypothesis text. The rectangle shows an island of confidence for N≤5.

This approach has been employed recursively on large samples of texts with good results. The whole algorithm and a presentation of the alignment results are presented in [2].

5.3.2.3. Forced alignment with factor automata

- *Introduction to the factor automaton approach*

Using factor automata has proved to be a very effective way to do FA on word level, mainly because it reduces the possible options of the aligner to a minimum, and that it can operate in linear time, determined by the size of the string that is to be aligned. The approach is thoroughly discussed in [8].

Consider the example sentence "*that he spent all his money*". A *factor* would in this context denote any substring of the sentence. For instance, the substrings: *"that he spent"*; *"spent all his money"*; *"his"*; and *"his money"* are all examples of factors.

A factor automaton can in some cases be visualized as a *directed acyclic word graph,* (*DAWG*). Since the word order remains unchanged, every substring can be constructed by traversing the graph. In the automaton in *Figure 5:3:2*, the factor "*he spent all*" could be retrieved by starting at node 1 and following the edges to node 4. The double line of the circles denotes that every node is a potential ending node:

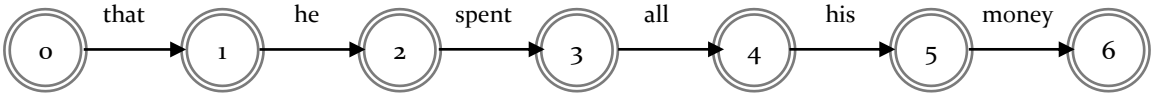


Figure 5:3:2 – Example of a simple factor automaton.

The previous figure is a bit simplified, since it does not specify any starting node. Therefore, the factor automaton is often given a starting node with a number of outgoing edges having the label "ε" (as in the Greek letter *epsilon*). This denotes an edge that can be traversed without consuming an input symbol (making the distance between the vertices "zero"). By adding a self-loop at every

vertex[§] the model can also allow for various noises and occurrences where the speaker adds a word that is not in the transcript. The factor automaton model that is used in this discussion therefore looks like *Figure 5:3:3*; where the string "*spent, umm, all his*" could be represented by the following traversal: 0→2→3→3→4→5:

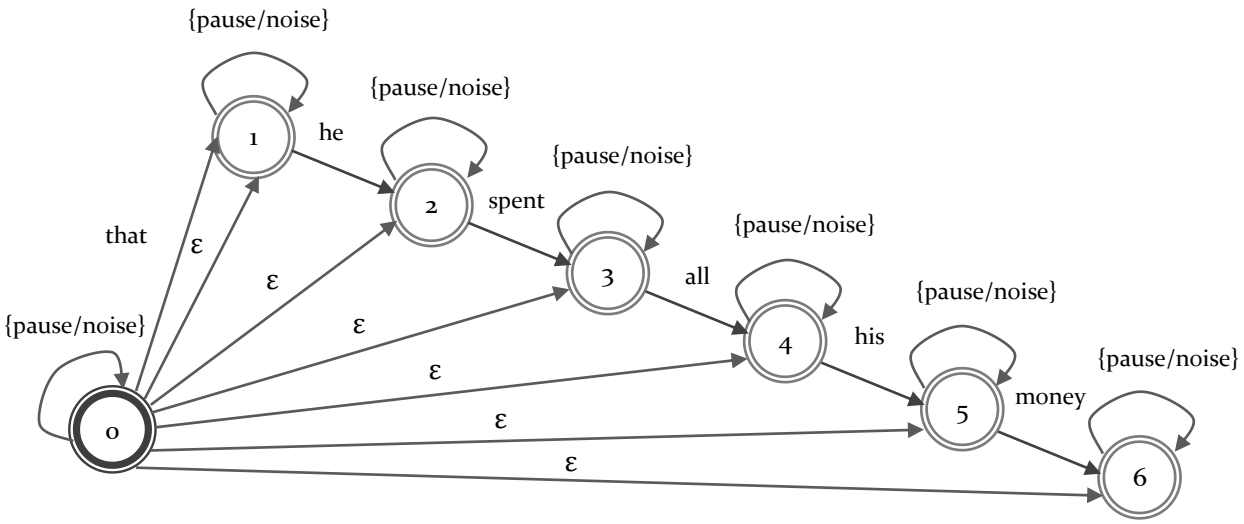


Figure 5:3:3 – A factor automaton that allows different start points and insertions.

Creating an algorithm that aligns a transcript to its audio has hereby been reduced to recognizing the input symbols (the factors) from the audio, and thereafter making the appropriate transition. Since the words are assigned to the *edges* of the graph, the vertices naturally represent the divisions between the words (i.e. the alignment points), as demonstrated in *Figure 5:3:4*:

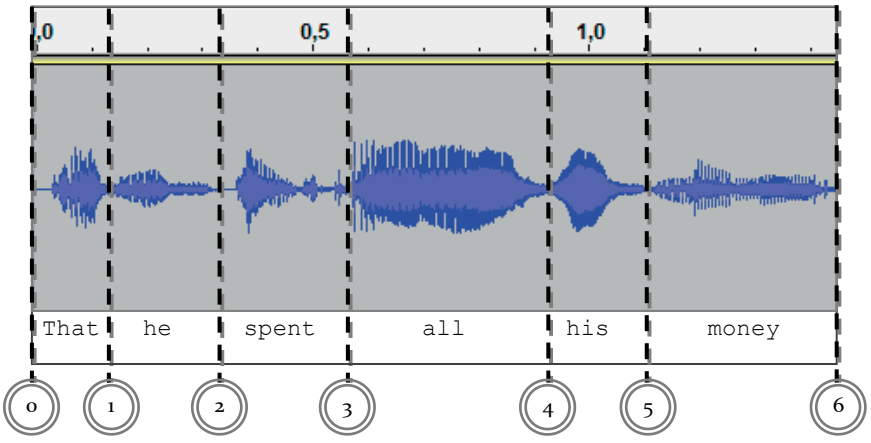


Figure 5:3:4 – Each node of the factor automaton represents a word-level alignment point

[§] This will eliminate the acyclic property of the graph!

- *Description of algorithm*

Realizing the forced alignment algorithm is now a matter of building a factor automaton (with the properties as shown in *Figure 5:3:3*) and extract words from the audio that serves as transitional inputs. An algorithm is presented below:

I. Begin at an alignment point (created on sentence-level)
II. Build a dictionary and a factor automaton from the succeeding words
III. Start at state 0
IV. Use a speech recognizer to retrieve the next word in the audio
V. Compare the synthesized word sequentially to the word of each outgoing edge
   - If match occurs, do the following:
     - Align the current state to the beginning position of the word (see *Figure 5:3:4*)
     - Follow the edge to the next state
     - Align the new state to the end position of the word
     - Go to IV
   - Otherwise, stop the alignment

Following the assumption that the transcript matches the spoken audio perfectly (no need to check for noise and pauses), and that the starting node already is aligned (which was done at sentence-level), the alignment can operate linearly in the size of the query. In order to intuitively understand this, one can observe that in states 1-*k* the algorithm only needs to make one single comparison in order to determine whether it should stop the alignment, or move on the next state – leaving a total of *k* comparisons before the process invariably stops. Making the transition from state 0 requires in the worst case *k* comparisons (either transit the natural way to state 1, or make the "effortless" transition through an epsilon edge). Hence, the alignment is done in *O(k)* time.

Adding the pause/noise-parameter does not change the complexity if the audio is "reasonably" close to the transcript – i.e. the number of noise comparisons at every state is always below a constant *c*.

- *Comments on realizing the method*

The factor automaton approach does indeed pose an interesting solution to the forced alignment problem; since it can operate with a minimal language model (i.e. models that only contain the words that are found in the current area, in the given order). If a mistake occurs, making the automaton halt, the erroneous states can be marked as "hard", and the program can move on to another area of alignment. This is the glory of combining many different approaches in one application, since the areas that made the factor automaton stumble can be fixed with phoneme-level alignment etc.

As previously mentioned, the factor automaton needs to have *at least one* aligned anchor that aligns the audio to the initial state. Otherwise, it would have to perform a brute force search for all the words in the query, without even knowing whether they actually are in the searched environment of the audio. If the words in the searched environment also resemble the word in the query, it could result in a "false positive" – rendering an incorrect alignment that may ruin the whole process.

Adding the ability to mark sections as "hard", "not completely aligned" and "confident alignment" is also important in this step, so that the application can correct these areas in later iterations.

5.4. Phoneme-level alignment

5.4.1. Definition of problem

When the application has come to this stage, the aligner is expected to be able to align every single phoneme in the given interval at a reasonable speed. More specifically, the starting position of each phoneme in the audio is sought. The preceding phases are assumed to guarantee that the length of the interval is manageable, and therefore precision is valued higher than convergence.

5.4.2. Proposed solutions

There exist a number of algorithms to do phoneme-level alignment. The most famous being the Viterbi-HMM method (see *Section 4.4* and *Section 4.5*). However, this approach has been constantly challenged by new methods, which try to achieve better and faster results. One of these is presented in this section. Descriptions of the Viterbi-HMM method can be found in the literature.

5.4.2.1. Solving maximization problem using SVM

A novel approach (as of 2010) is to use Support Vector Machines in order to align the audio at phoneme level. This method differs from many other HMM-based methods since it does not linguistically model the language. Instead it uses a number of intuitive assumptions about the nature of speech, and combines them in an elegant way, producing an alignment result that is comparable to other state-of-the-art solutions. On top of this it requires less training, provides convergence guarantees and is more efficient both in the training and the alignment phase!

This entire section is based upon the article [7], which elaborates on the theory and presents the final algorithms in pseudo-code.

The method begins as the other methods by dividing the audio into frame windows and extracting their MFCC vectors. These frame windows will be referred to as: $\overline{x} = (x_1, x_2, \dots, x_T)$.

Then a vector $\overline{e} = (e_1, e_2, \dots, e_T)$ is constructed (formally called an *event vector* in the article). This vector contains all phonemes from the transcript. Following from previous examples, the analyzed string would be “all his”, and its event vector would be: $\overline{e} = (a, ll, h, i, s)$.

The goal is to create a function: $f(\overline{x}, \overline{e}) \rightarrow \overline{y}$, where $\overline{y}$ is a vector containing the *best* starting point of each phoneme in $\overline{e}$ (that is, the desired alignment).

So how is this done? Before delving into the area of Support Vector Machines, an intuitive introduction of the approach is given.

5.4.2.2. Introduction to the maximization problem

A naïve approach for finding the best alignment could for instance be to create a distance function $d(x_i, x_j)$, which compares the time windows x_i and x_j with the following properties:

$$d(x_i, x_j) = \begin{cases} Small\ if\ x_i\ and\ x_j\ belong\ to\ the\ same\ phoneme \\ Big\ if\ x_i\ and\ x_j\ belong\ to\ different\ phonemes \end{cases} \quad (2)$$

To achieve this, $d(x_i, x_j)$ calculates the Euclidian distance between the MFCC vectors of x_i and x_j.

A one-dimensional visual representation of the MFCC vectors will be used throughout the examples, to illustrate the concept of "two time windows belonging to the same phoneme".

The upper part of *Figure 5:4:1* shows the audio as usual, and the other shows the MFCC vector representation. In this example, one would expect the following statements to hold: $d(x_1, x_2) \approx d(x_2, x_3) \ll d(x_3, x_4) \approx d(x_7, x_8)$:

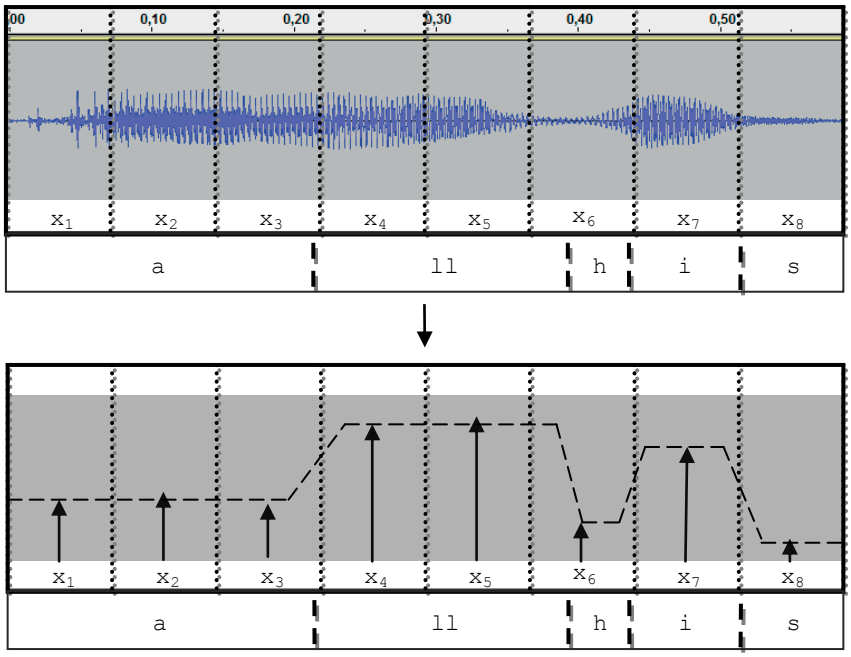


Figure 5:4:1 – A sound segment with x_t denoting MFCC-Vectors extracted from the sound frames; together with the true alignment of the phonemes. The second graph informally visualizes the MFCC vectors of the same segment.

Using the function $d(x_i, x_{i+1})$ as the so called *base alignment function* of $f(\overline{x}, \overline{e})$ would therefore reduce the alignment problem to the optimization problem of finding the set of starting points $\overline{y}$ that *maximizes* the Euclidian distance between the MFCC vectors. A non-maximal solution would put at least one of the starting points *inside* a phoneme instead of *between* two phonemes. *Figure 5:4:2* depicts a maximal solution:

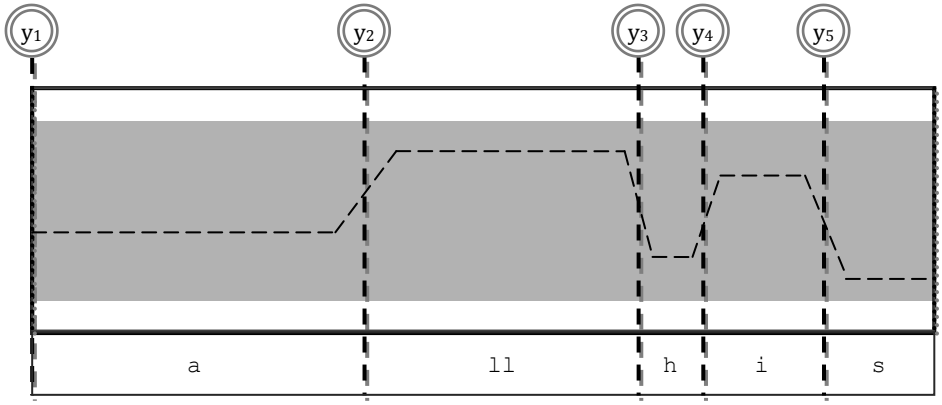


Figure 5:4:2 - Example of an optimal alignment with respect to the distance function $d(x_i, x_j)$

5.4.2.3. The seven base alignment functions

Only using the distance function as discussed in the previous section is of course not sufficient as there are many factors that could make the results non-reliable. For instance, a slight disturbance of the signal could make the algorithm detect a phoneme transition that does not exist, and it would probably have problems finding boundaries between two phonemes that happen to have similar MFCC vector representation. However, the main idea of analyzing characteristics of speech can be extended by adding more base alignment functions, where each function renders a unique alignment of the audio. Results have shown that combining seven wisely chosen base alignment functions in a clever way produces results that can be trusted.

An overview of the functionality of each base alignment function is presented in *Table 5:1:*

Table 5:1 – Summary of the base alignment functions

Function	*Alignment strategy*
$\psi_1 - \psi_4$	Evaluate alignment by comparing the euclidian distance between the MFCC vectors of two time frame windows, see *5.4.2.2. Introduction to the maximization problem.*
ψ_5	Evaluate alignment by using an external phoneme classifier, answering the question: "is it probable that this time frame contains the given phoneme?"
ψ_6	Evaluate alignment by comparing the length of the time frame to the expected length of the phoneme. The expected length of the phoneme is retrieved from statistical data.
ψ_7	Evaluate alignment by detecting changes of speed rate of the speaker. It is improbable that the speed rate of the speaker changes within a phoneme.

Section 5.4.2.4 and *Section 5.4.2.5* explain how the base alignment functions are weighted, and how they can be used in order to align phonemes.

- *Base alignment functions $\psi_1 - \psi_4$, measuring euclidian distance*

The first four base alignment functions all works under the same principle, with the difference lying in which frame pair that is analyzed. All functions rely on the Euclidian distance function $d(x_i, x_j)$ that was presented in *Section 5.4.2.2*, and are defined as:

$$\psi_1(\overline{x}, \overline{e}, y_i) = d(x_{y_i-1}, x_{y_i+1}) \qquad (3)$$

$$\psi_2(\overline{x}, \overline{e}, y_i) = d(x_{y_i-2}, x_{y_i+2}) \qquad (4)$$

$$\psi_3(\overline{x}, \overline{e}, y_i) = d(x_{y_i-3}, x_{y_i+3}) \qquad (5)$$

$$\psi_4(\overline{x}, \overline{e}, y_i) = d(x_{y_i-4}, x_{y_i+4}) \qquad (6)$$

ψ_1 compares the distance between two adjacent frames. However, this function is likely to compare frames that belong to the same phoneme, so the next function ψ_2 compares two frames that are separated by two other frames, and so on (thus giving a more general notion of the distances that are caused by the position of y_i). *Figure 5:4:3* illustrates which frames pairs that are compared:

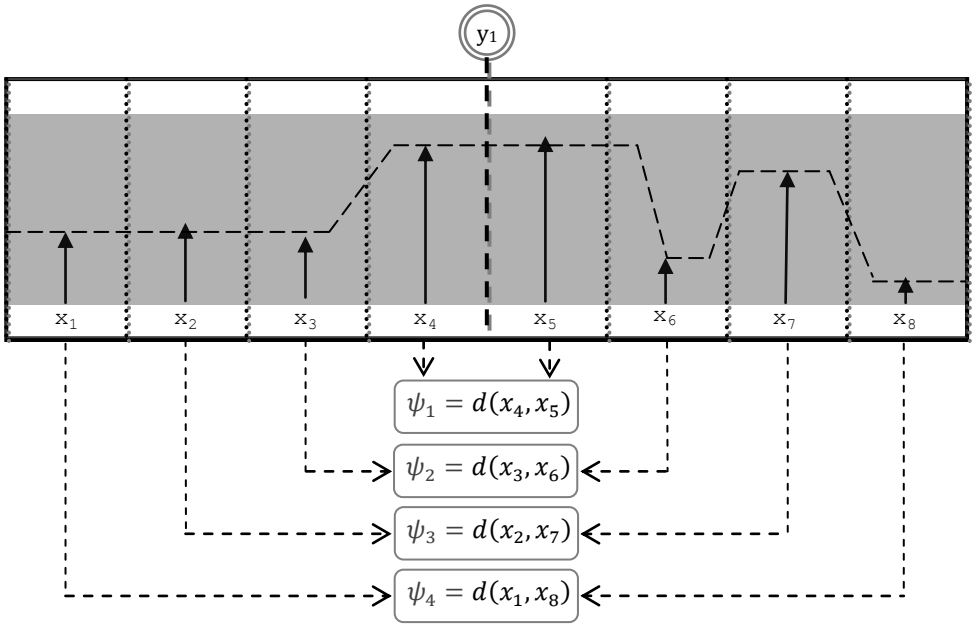


Figure 5:4:3 – Four base alignment functions

- *Base alignment function ψ_5, using phoneme classification*

The fifth base alignment function uses a phoneme classifying algorithm, see [13], which is a function, *g*, that calculates the likelihood that a given phoneme is inside a given audio segment. Recalling that e_i denotes the current phoneme, the function is formally defined as:

$$\psi_5(\overline{x}, \overline{e}, y_i, y_{i+1}) = \sum_{t=y_i}^{y_{i+1}-1} g_{e_i}(x_t) \qquad (7)$$

In words, the function takes every time frame x_t between two given alignment points, y_i and y_{i+1}. Each of these frames is processed, and the *total likelihood* that the phoneme e_i is between the alignment points is returned. *Figure 5:4:4* visualizes the functionality, with the Σ-node depicting summation:

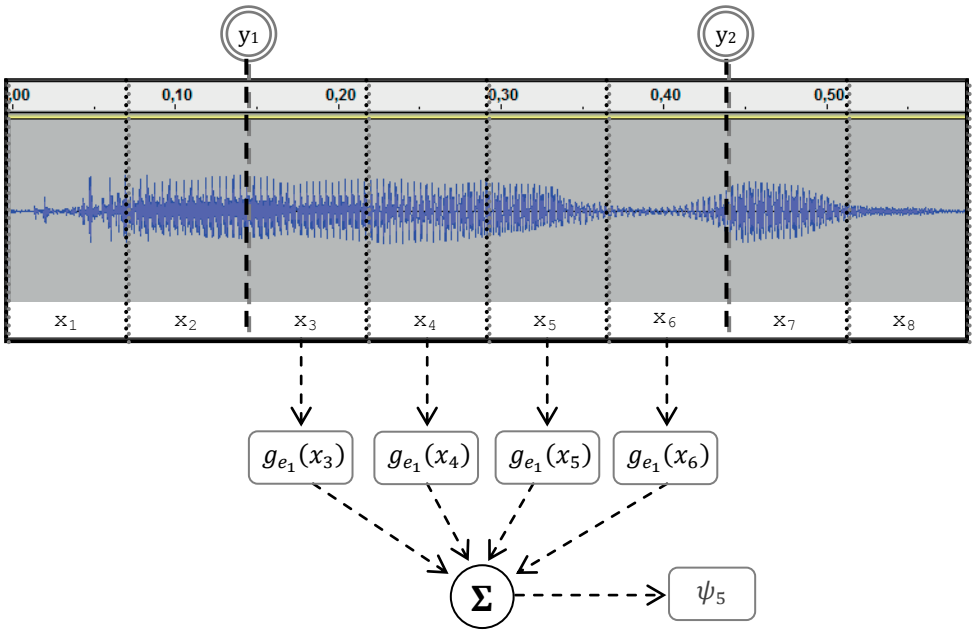


Figure 5:4:1 : Return the probability that sound frames x_3-x_6 contain the phoneme e_1.

Again, the alignment algorithm will seek to maximize the value of ψ_5, trying to bring the alignment pair y_i and y_{i+1} to encapsulate the phoneme e_i as tight as possible.

- *Base alignment function ψ_6, measuring length of utterance*

The sixth function make use of statistical data of speech, namely the *normal distribution* of *the length required to pronounce any given phoneme*, e_i: $N(\hat{\mu}_{e_i}, \hat{\sigma}_{e_i})$. The function simply calculates the distance between the two given alignment points, y_i and y_{i+1}, and returns a value of how probable it is with respect to the statistical data:

$$\psi_6(e_i, y_i, y_{i+1}) = log\ N(y_{i+1} - y_i\ ;\ \hat{\mu}_{e_i}, \hat{\sigma}_{e_i}) \qquad (8)$$

If, for instance, the mean length to pronounce a given phoneme is *t* milliseconds, this function will naturally produce a maximal value when $y_{i+1} - y_i = t$.

- *Base alignment function ψ_7, measuring speed of the speaker*

The seventh and last base alignment function calculates the speech rate, i.e. the speed of the speaker. The underlying assumption is that the speech rate varies slowly within an utterance (given that the utterance is recorded by one single person). In other words, abrupt changes in speech rate within a phoneme are unlikely, and the function will thus seek to find the places in the utterance where the changes occur, and put the alignment points there.

Using the normal distribution data of phoneme lengths $\hat{\mu}_{e_i}$, (as was introduced in base alignment function ψ_6), the relative speech rate r_i of *one phoneme* can be defined as:

$$r_i = actual\ length / average\ length = (y_{i+1} - y_i)/\hat{\mu}_{e_i} \qquad (9)$$

Now, the local change in speech rate can be defined as:

$$\psi_7(\overline{e}, y_{i-1}, y_i, y_{i+1}) = (r_i - r_{i-1})^2 \qquad (10)$$

Note that three alignment points, y_{i-1}, y_i and y_{i+1}, need to be used. *Figure 5:4:5* shows what operations are done:

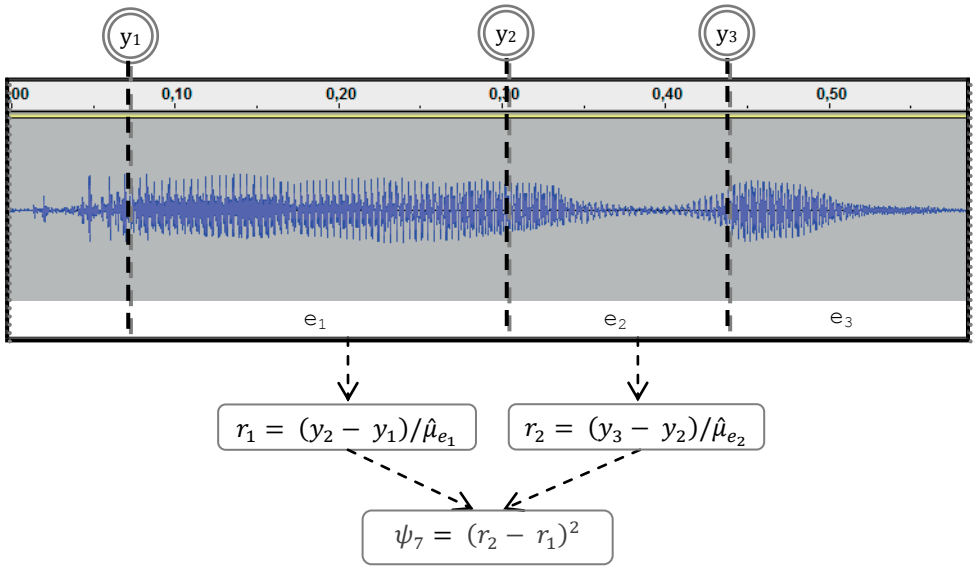


Figure 2:4:5 - Return a high value if the changes in speech rate are low <u>between</u> the alignment points.

Recall that ψ_7 calculates the local change in speech rate. Thus, maximizing this function will ideally put the divisions exactly on the *transitions between* phonemes, as those transitions are the places where such changes are most likely to occur.

- *Multi-phoneme base alignment function* ϕ

Each of the seven base alignment functions took at most three alignment points, y_{i-1}, y_i and y_{i+1}, and returned a value of how "good" the alignment was from its perspective. In order to get a value for the whole sequence (possibly containing a large number of phonemes); a multi-phoneme base alignment function ϕ is created. It takes a vector of alignment points, $\bar{y}$, and sums the base alignment value of each phoneme. *Figure 5:4:6* illustrates the composition, which is defined as follows:

$$\phi_i(\bar{x}, \bar{e}, \bar{y}) = \sum_{j=1}^{|\bar{y}|} \psi_i(\bar{x}, \bar{e}\, y_{j-1}, y_j, y_{j+1}) \qquad (11)$$

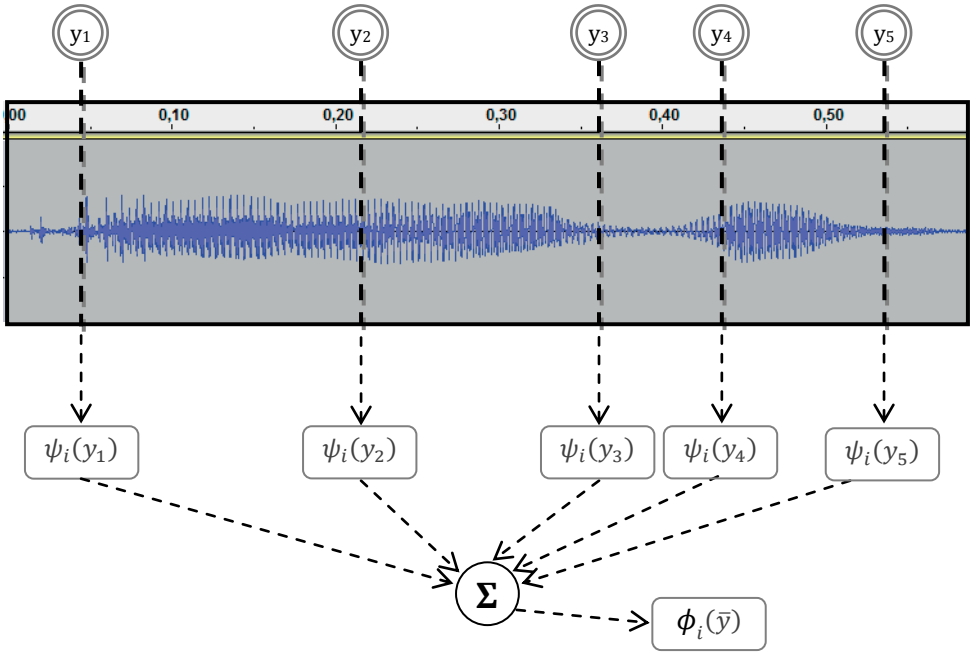


Figure 5:4:5 – Combining the alignment data for every phoneme in the sequence

Observe that the total base alignment function ϕ_i is completely dependent on ψ_i.

5.4.2.4. Weighting the base alignment functions with SVM

This section is divided in three parts in order to capture all the necessary definitions.

i. Defining the weight vector and the global alignment function

Having defined the seven base alignment functions $\phi_1 - \phi_7$, we now must find a way to create an aligner that let each function contribute to the final result in the best way possible. By grouping the base alignment functions into a vector $\boldsymbol{\phi} = (\phi_1, \phi_2, \dots, \phi_7)$; and defining a corresponding weight vector $\boldsymbol{w} = (w_1, w_2, \dots, w_7)$; a global alignment function $\boldsymbol{f}$ could be created where the values of the weight vector determine the importance of each base alignment function.

The global alignment function is presented in the following form, where *argmax* denotes a function that returns the set of values of $\bar{y}$, that maximizes the given expression:

$$f(\bar{x}, \bar{e}) = \bar{y} = \underset{\bar{y}}{argmax}\ \boldsymbol{w} \cdot \boldsymbol{\phi}(\bar{x}, \bar{e}, \bar{y}) \qquad (12)$$

ii. Creating a quality function

Let's assume a training set, $\bar{x}, \bar{e}$, has been provided, together with its *correct* alignments $\bar{y}'$. By using this correct alignment as reference, a *quality function* $\gamma(\bar{y}', \bar{y})$ can be defined. The idea is to create a quality function that can assess how well the alignment function ***f*** is performing for any given weight. A bad alignment (i.e. the alignment positions y_i do not match the phonemes) should make the quality function return a high value, and a good alignment should result in a low value. A perfect alignment, where $\bar{y} = \bar{y}'$, returns $\gamma(\bar{y}', \bar{y}) = 0$.

A quality function that lives up to these requirements has been proposed and has the following definition; In words, count all alignment points that differ from the true alignment with a value greater than ε. Divide this value with the total number of alignment points to get a number of the *average number of "incorrect alignment points".* See definition below:

$$\gamma(\bar{y}', \bar{y}) = \frac{1}{|\bar{y}|} \left|\{\mathrm{i} : |\mathrm{y_i} - \mathrm{y_i'}| > \varepsilon\}\right| \qquad (13)$$

Observe that this function regards every alignment point that is "sufficiently close" to the true alignment point as "completely correct". Generally, when doing forced alignment at phoneme level, there is no "perfect" alignment, and this is due to the fact that divisions between phonemes are subjective and ambiguous.

*iii. Using the quality function to calculate the optimal **w***

When a quality function has been defined, the weight vector **w** can be determined. Recall that every component w_i corresponds to one of the base alignment functions ϕ_i – deciding how much importance that function should have when the final alignment is calculated.

So how can the best ***w*** be found? A naïve approach could be to simply run the global alignment function ***f***, see (12), for every possible value of ***w***, and score each result with the quality function. The ***w*** that renders the best score from the quality function is the sought optimal vector.

That approach is of course practically infeasible, and this is where the theory of support vector machines comes in. It works by projecting the base alignment functions onto ***w*** and match the projections to the quality function. A sequential description is provided below:

- Create a training set, $\bar{x}$ and $\bar{e}$, with the correct alignment $\bar{y}$.
- Let $\bar{y}'$ denote every possible sequence of alignments.
- Compute the quality of every alignment: $\gamma(\bar{y}, \bar{y}')$.
- Compute $\boldsymbol{\phi}(\bar{y}) = (\phi_1(\bar{y}), \phi_2(\bar{y}), \dots, \phi_7(\bar{y}))$
- Compute $\boldsymbol{\phi}(\bar{y}') = (\phi_1(\bar{y}'), \phi_2(\bar{y}'), \dots, \phi_7(\bar{y}'))$
- Project $\boldsymbol{\phi}(\bar{y})$ and $\boldsymbol{\phi}(\bar{y}')$ onto ***w*** by using scalar product, see *Figure 5:4:7.*
- Measure the distance between the projections: $d = \boldsymbol{w} \cdot \boldsymbol{\phi}(\bar{y}) - \boldsymbol{w} \cdot \boldsymbol{\phi}(\bar{y}')$

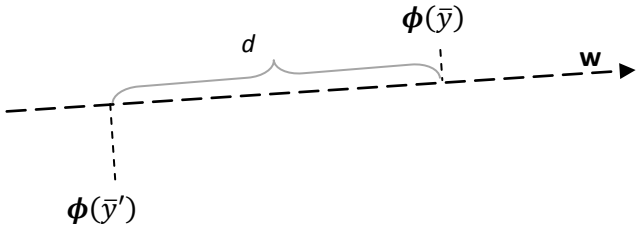


Figure 5:4:7 – Projecting base alignment functions onto the weight vector w in a 7-dimensional room

The distance between the projections can be seen as a new quality function, whose value depends on ***w***. Since a general quality function has already been defined, $\gamma(\bar{y}, \bar{y}')$, one could regard the training problem as finding the particular ***w*** that makes the distance between the projections correspond to the quality function.

What does this mean? Let's examine an example of how a bad value of ***w*** would affect the projection***.***

In *Figure 5:4:7,* the projection $\boldsymbol{w} \cdot \boldsymbol{\phi}(\bar{y}) - \boldsymbol{w} \cdot \boldsymbol{\phi}(\bar{y}')$ would yield a positive result, while *Figure 5:4:8* yields a negative result (note that $\boldsymbol{\phi}(\bar{y})$ and $\boldsymbol{\phi}(\bar{y}')$ remained unchanged).

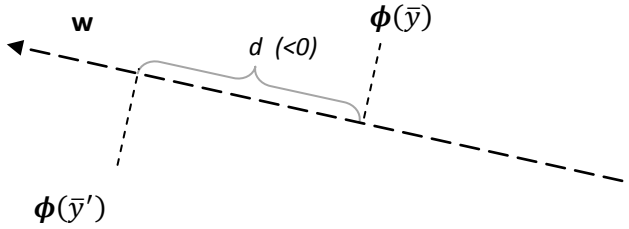


Figure 5:4:8 – A weight vector that does not match the quality function, and therefore produces bad alignment results

Recall that the quality function $\gamma(\bar{y}, \bar{y}')$ returns a low (positive) value if $\bar{y}'$ is a "good" alignment, and a high value otherwise. If the projection difference *d* is to match $\gamma(\bar{y}, \bar{y}')$, it cannot have negative values. Hence, the value of the weight vector ***w*** in *Figure 5:4:8* is invalid.

With this being said, training the phoneme-level algorithm is a matter of finding a value of the 7-dimensional vector ***w*** that makes the projected difference *d* match the general quality function as close as possible.

The difference between the base alignment functions and the general quality function is that the former can be generative, i.e. they can produce alignment on unseen material. The general quality function serves as a reference point, having information about the "true" alignment, while the base alignment functions need to be weighted in order to resemble the quality function as close as possible.

This means that one would have to consider every possible weight function ***w***, make the projection, and pick the best possible alternative. This can be formulated as a maximization problem, and an algorithm for solving this effectively is provided in [7].

5.4.2.5. Summary of method and comments

This approach to phoneme-level alignment is based upon the most general foundations of human speech, as opposed to most other methods that builds models of the actual language.

Seven base alignment functions are created, each corresponding to a prosodic feature.

The training is done by running the seven base alignment functions on a training set, and use the result to produce a weight vector that determines the importance of each function. The weight vector is created by solving an SVM optimization problem.

When the weight vector has been created, the FA problem can be formulated as a maximization problem (initially defined in (12)):

$$f(\bar{x}, \bar{e}) = \bar{y} = \underset{\bar{y}}{argmax}\ \boldsymbol{w} \cdot \boldsymbol{\phi}(\bar{x}, \bar{e}, \bar{y}) \qquad (14)$$

Now, the phoneme-level alignment problem has been reduced to solving (14). An effective algorithm for doing that in $O(|\bar{x}| \cdot |\bar{e}| \cdot L^2)$ time, where *L* denotes the maximal length of any event *e*, is presented in [7].

As previously mentioned, the size of the audio that is to be aligned can always be reduced by running sentence- and word-level alignment first, and the claimed accuracy of the method does indeed make it a very interesting last step in the bigger alignment procedure.

The general structure of the method may also make it effective for aligning other languages where speech-synthesis engines and HMMs are not available.

6. Summary

This article has examined how a user-controlled application for doing dynamic forced alignment can be constructed. The main functionality of the application is the ability to find the position of any given word in a long audio file. In order to avoid having to process the whole audio file (as is common in FA), the alignment procedure is divided into four distinct steps - each using a different strategy in order to maximize the efficiency.

In order to make an effective implementation, many design decisions and experiments with different algorithms need to be carried out. To facilitate this, some notes and suggestions of possible implementations have been given throughout the article.

This being said, there exist many interesting and promising approaches that have not been discussed. Forced alignment is an active area of research, where the goal is to find methods that give fast and trustable results on a wide variety of input, using models that only require a minimum of training.

Hopefully, this article can serve as a first step to realizing the application, and also be an introductive reference to the exciting field of forced alignment.

7. References

[1] Speech and Language Processing, 2nd edition, An introduction to Natural Language Processing, computational Linguistics, and Speech Recognition
Daniel Jurafsky and James H. Martin
Prentice Hall, ISBN: 978-0-13-187321-6, 2008

[2] A recursive algorithm for the forced alignment of very long audio segments
Pedro J. Moreno, Chris Joerg, Jean-Manuel Van Thong and Oren Glickman
Cambridge Research Laboratory, 1998

[3] Text-constrained speaker recognition using hidden Markov models
Kofi Boakye
University of California, Berkeley, 2003

[4]Speaker-independent phoneme alignment using transition-dependent states
John-Paul Hosom
Center for Spoken Language Understanding, School of Science and Engineering, Oregon Health and Science University, 2008

[5] Forced alignment for speech synthesis databases using duration and prosodic phrase breaks
Arthur R. Toth
Language Technologies Institute, Carnegie Mellon University, 2004

[6]Phonetic alignment: speech synthesis-based vs. Viterbi-based
F. Malfrère,O. Deroo, T. Dutoit, C. Ris
Faculté Polytechnique de Mons-TCTS; Babel Technologies SA, Belgium, 2002

[7] A Large Margin Algorithm for Speech-to-Phoneme and Music-to-Score Alignment
Joseph Keshet, Shai Shalev-Shwartz, Yoram Singer, and Dan Chazan
IEEE Transactions on Audio, Speech and Language processing, Vol.15, No.8, November 2007

[8] A factor automaton approach for the forced alignment of long speech recordings
Pedro J. Moreno, Christopher Alberti
Speech Research Group, Google Inc. 2009

[9] Lightly supervised and unsupervised acoustic model training
Lori Lamel, Jean-Luc Gauvain and Gilles Adda
Spoken Language Processing Group, Computer Speech and Language. France, 2002

[10] Combining speaker identification and bic for speaker diarization
X. Zhu, C. Barras, S. Meignier, and J. L. Gauvain
Proceedings, INTERSPEECH, 2005.

[11] SpeechSkimmer: A System for Interactively Skimming Recorded Speech
Barry Arons
MIT Media Lab, Massachusetts Institute of Technology, 1997

[12] Audacity (2010), Plug-in: SilenceMarker
Alex S. Brown
http://audacity.sourceforge.net. [accessed May 01, 2010].

[13] An online algorithm for hierarchical phoneme classification
Ofer Dekel, Joseph Keshet, and Yoram Singer
School of Computer Science and Engineering, The Hebrew University, Jerusalem, 2005

[14] Google n-gram corpus, Web 1T 5-gram Version 1
Thorsten Brants, Alex Franz
Linguistic Data Consortium, Philadelphia, 2006
http://www.ldc.upenn.edu/Catalog/CatalogEntry.jsp?catalogId=LDC2006T13 [accessed 02 May 2010]