Building a Unit Selection Synthesis Voice

Version 04/03/18, 02:36:09 PM

Steps

Prepare the database

- get speech data (and the accompanying text)
- annotate database using the *Aligner*
- convert to Festival format

Define synthesis modules, at least

- specify a phone set including phonetic features
- a duration module
- a lexicon

Optional

- treatment for non-standard words (for synthesizing numbers etc.)
- prosodic analysis
- F0 modeling

Preparing the database

Getting speech data

You are welcome to use any German or English data for which you have both audio data and accompanying orthographic transcriptions (as long as you don't violate any copyright restrictions). If you want to try another language, talk to me about it first.

If you don't have access to speech data, I recommend librivox.org. This is a project where volunteers read and record texts which are in the public domain in the U.S. (i.e. where the copyright has expired in the U.S.).

Everyone can download these recordings. They are like audio books, except the speakers are usually not professional speakers. Be aware, however, that there are different regulations in different countries, so the copyright might be still in place in countries outside U.S. If in doubt, check www.gutenberg.org as a starting point.) The recordings provided by librivox.org are in the public domain, too, so it would be ok to use them even for commercial purposes (but the project asks to be acknowledged in this case).

What kind of data

Usually your speech data should match the style of what you are expecting to synthesize. This is tricky since texts that are not copyrighted anymore are usually older, and that's evident in the writing style. Try to go for something not too old (except if you will synthesize this style later).

Also, we will use automatic methods to do the annotation for us. Automatic methods have trouble with unknown words. So try to pick texts that are not full of foreign names or complicated words.

We aim for several (say, 3 to 6) hours of speech, and they all need to be from the same speaker. Most recordings are around 20 minutes, so you will have to find a speaker who provided several recordings (not necessarily all for the same book.)

Prepare for automatic annotation

Our ultimate goal is to use the speech data as a database for speech synthesis. Our speech synthesis system needs to know the properties of every little snippet of speech in the database in order to determine whether that snippet would be useful for synthesizing a given utterance.

To this end we will convert our speech data to so-called utterance structures which contain most of this information. And these utterance structures can be generated by providing appropriate label files – for instance label files that identify where to find each phone, where to find each syllable, where to find each word in our speech files. We can create these labelfiles automatically, and the tool we use for the automatic annotation is called the *Aligner*.

You can find some help on the Aligner on the Phonetics website. We'll deviate a bit from the description there (we'll change some settings before running the Alignwords and Alignphones commands) but everything on preparation and trouble shooting is relevant for us.

Audio format

The Aligner wants .wav files, not .mp3.

For converting .mp3 to .wav, you can use the following set of bash commands, which convert all .mp3 files in your working directory to .wav files using the same basename, only changing the extension. If you don't know what a working directory is, and what bash commands are, ask me.

```
for file in *.mp3
do
wavname=`echo $file | sed 's/\.mp3/.wav/'`
lame --decode $file $wavname
done
```

This iterates over all files that match the pattern *.mp3. In each iteration, we will have the name of the current file stored in the variable $file. The first command in the loop takes this name, replaces the .mp3 ending by .wav, and stores the result in yet another variable called $wavname. Then, we can use the command lame --decode with the two filenames as arguments to create the .wav file.

There are more elegant ways to do this, but this seemed most transparent, and should hopefully be clear enough even for bash newcomers.

At IMS, you can check the format of your .wav files now by

```
sfinfo *.wav
```

This will give you details about the sampling rate and the number of channels. It's also convenient for checking the duration.

A propos duration – the length of the recording has a non-linear effect on processing time for alignment. When preparing this project, I found that for recordings of approx. 20 minutes the Aligner took more than half an hour, while for recordings of approx. 10 minutes, it took less than 10 minutes. So you might consider cutting recordings in halves or thirds.

The Aligner wants its input files to have a sampling rate of 16 kHz, and only one channel. The command to change this at IMS is

```
sox inputfile.wav -r 16000 -c 1 outputfile.wav
```

where you first specify the input file, then the options for the output file (rate 16000, and 1 channel in this case), then the name of the output file. Of course you don't have files of these exact names, so adapt the command above to match your file names. Also, you could do this in a loop, in the same way as above. I recommend to have easy-to-type names for your downsampled files, because we will use the a lot. Try to get this loop to work, it's a good exercise for the many loops we'll need in the process of building a synthesis voice.

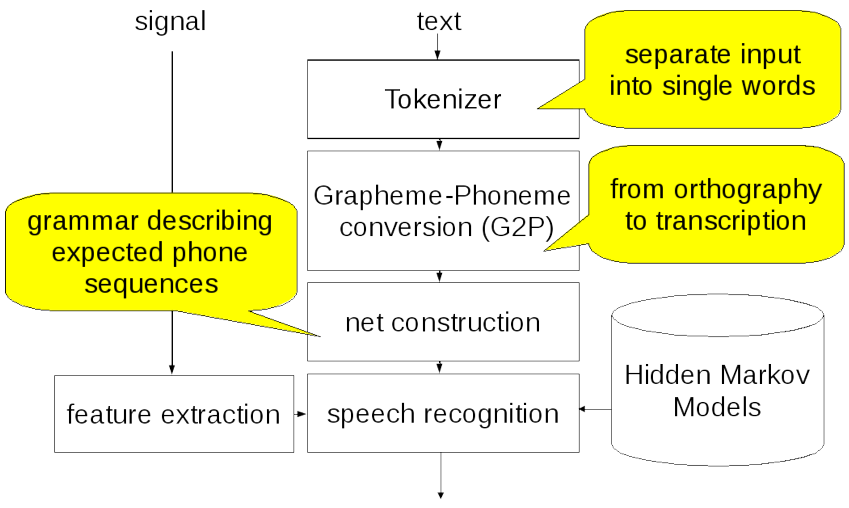
Text format

For each of your .wav files, you need a .txt file with exactly the same name (except the ending, of course) that contains exactly the text that was said in the recording. You will most likely start out with one long text file for your book, and with several audio files which each contain just a portion of the long text, and possibly introductory remarks on copyright etc. You will need to listen to your recordings, and copy their texts from the longer file to individual shorter text files. If there are additional remarks at the beginning or at the end, option one is to write their text at the beginning and end of the text. Or you will have to cut the remarks from the audio file. From an ethics point of view, it's maybe more transparent to keep the remarks in the recordings, and add the texts to the .txt files.

Concerning text format, the Aligner is even more picky than with its audio input :). Please save your .txt files in iso-8859-1 encoding (also called Latin-1 encoding). You could convert existing utf-8 files to iso-8859-1 using yet another shell command, but since you'll be editing manually anyway, it's probably easier to do this when saving the files with your editor. If you use gedit at IMS, it has a menu command "Save as" which allows you to specify the encoding.

Unfortunately some symbols that are popular in texts are not known in iso-8859-1 – for instance these: …, –, and some typographic quotes.

How the Aligner works


signal
text
Tokenizer
separate input into single words
Grapheme-Phoneme conversion (G2P)
from orthography to transcription
grammar describing expected phone sequences
net construction
Hidden Markov Models
feature extraction
speech recognition

The picture on the right-hand side shows how the Aligner works internally.

It gets a speech signal, and a text.

Basically, the Aligner works like a simple speech recognizer. It extracts the spectral properties for each time frame (see the box "feature extraction"). It will later use pre-trained Hidden Markov Models of each phoneme of the language (see the database of HMMs on the bottom right) –

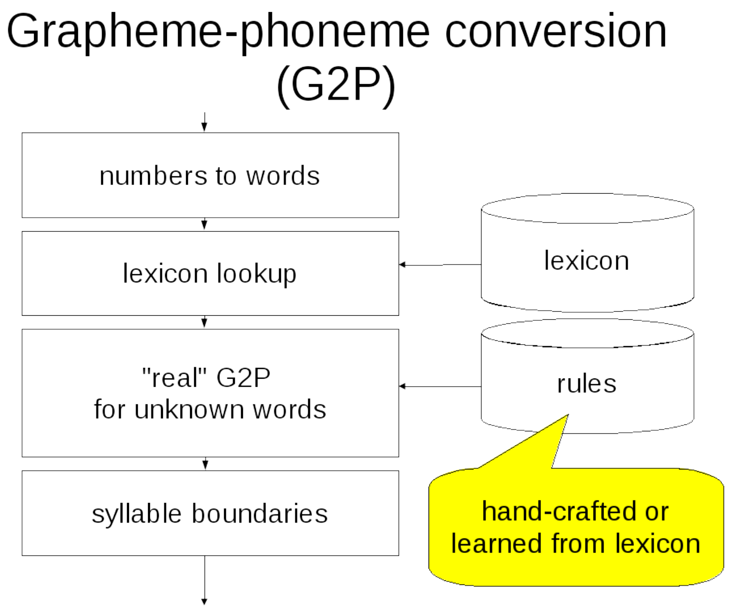
these can be thought of as describing the typical spectral properties of the phoneme.

However, in contrast to a speech recognizer, the Aligner does not attempt to guess which phonemes occurred; instead it relies on the provided text to determine which sequences of phonemes could possibly occur, given the text. To this end, the text is first tokenized. Then the Aligner determines the pronunciation of each word token (see the box G2P above). We'll have a closer look at this G2P part below. Finally, the sequence of expected phonemes, with optional pauses added between words, is represented as a net structure. The Aligner then determines the most likely path through this net given the spectral properties of the speech signal, which implicitly also yields the time points in the speech signal where the phonemes occurred. These time points are then written into the label files.

G2P in the Aligner

Below the G2P part of the Aligner is depicted schematically.

Predicting the pronunciation involves first converting numbers to text (note that this is very similar to text analysis in speech synthesis!), then if a lexicon exists, all words are looked up in the lexicon. However, since unknown words are very common, it is advisable to also have a set of rules to convert unknown words to phonemes. These can be learned from a lexicon (Festival provides tools to do so!), or written by hand.


Grapheme-phoneme conversion (G2P)
numbers to words
lexicon lookup
"real" G2P for unknown words
syllable boundaries
lexicon
rules
hand-crafted or learned from lexicon

For preparing a database for a synthesis voice, we also need syllable boundaries. So if they are not provided by the lexicon lookup and the rules for out-of-vocabulary words, these need to be inserted in a last step.

Modifying preprocessing and G2P for Automatic Alignment

The Aligner at IMS is implemented in a way that it is easy to exchange some of the components above. For preprocessing, we can use any script that takes a file as input and returns a list of tokens to standard out, separated by line breaks. For lookup, we can use any script that reads words line by line from standard in, and outputs transcriptions line by line, if they conform to the Aligner's phoneme inventory and syllable notation.

For our current purposes, we will cheat a bit and use the G2P implementation in an existing German Festival voice which is superior to the one in the Aligner itself. We will not use this voice for building our own synthesis later; but here we will use it to ensure that the annotation quality of our database is optimal. In order to make sure that the tokenization and the lookup are consistent, we will use the same existing voice for tokenization. I provide the two scripts here:

```
  /mount/studenten/synthesis/2017/bin/de.preprocess
  /mount/studenten/synthesis/2017/bin/de.lookup
```

For American English, we will also use tokenization and lookup by Festival, since the Aligner has a British English dictionary. However, here we do not need IMS-specific voices. We will use the default US English voice that comes with Festival instead.

Please note that the Aligner has been trained on British English; so the only thing we do here is to do the lookup in a US English dictionary, and to display the resulting phoneme annotations in US English, but all US English phonemes are mapped to matching British English counterparts in an intermediate step. Check whether the annotation quality is to your satisfaction before you proceed…

The two scripts are here:

```
  /mount/studenten/synthesis/2017/bin/us.preprocess
  /mount/studenten/synthesis/2017/bin/us.lookup
```

In order to use these scripts in a bash shell, first prepare the .txt file and the .wav file according to the specifications above.

Then set the following variables to make the Aligner use the modified scripts:

For German:

```
export ALANG=deu
export ORTHOGRAPHICINPUT2WORDLIST=/mount/studenten/synthesis/2017/bin/de.preprocess
export WORDLIST2PHONEMES=/mount/studenten/synthesis/2017/bin/de.lookup
```

For American English:

```
export ALANG=eng
export ORTHOGRAPHICINPUT2WORDLIST=/mount/studenten/synthesis/2017/bin/us.preprocess
export WORDLIST2PHONEMES=/mount/studenten/synthesis/2017/bin/us.lookup
export DICTFILE=/mount/studenten/synthesis/2017/bin/phones.en2us.dic
```

The last variable is to map between British English HMMs and American English phonemes.

For British English:

```
export ALANG=eng
export ORTHOGRAPHICINPUT2WORDLIST=/mount/studenten/synthesis/2017/bin/en.preprocess
export WORDLIST2PHONEMES=/mount/studenten/synthesis/2017/bin/en.lookup
export DICTFILE=/mount/studenten/synthesis/2017/bin/phones.en2en.dic
```

All scripts are tested but not tested extensively, so if you come across an error, and you are absolutely certain that your input .txt and .wav are ok, please notify me.

Other languages

If you have a language that has a very straightforward pronunciation, and is similar in phoneme inventory to either German or English, it may be manageable to build a voice without having a dictionary. I have found letter-to-sound rules for Spanish implemented in Festival, and have put them into a lookup script for Spanish.

If you are interested in building a Spanish voice, or another language that has simple G2P, have a look at

```
  /mount/studenten/synthesis/2017/bin/es.lookup
```

In the first part of this file, a phone set has to be defined. We will have to do this for most voices anyway, so this would not cause additional work. If you decide to use a new language, I'll provide more information on how to define phone sets.

In the middle part, there are so-called rewrite rules that map from graphemes or sequences of graphemes (left-hand side) to phonemes (right-hand side) like this:

```
    ( [ a ] = a )
    ( [ e ] = e )
```

These two rules simply state that the letters a and e map to the phonemes a and e in Spanish.

```
   ( [ "'" i ] = i1 )   ;; stressed vowels
```

This rule need some explanation: accented i (í) has to be mapped to 'i before applying these rules (preprocessing takes care of this in our case), and this rules states that this accented 'i (the single quote has to be quoted) is to be mapped to a stressed vowel i (the i1 is stressed i as opposed to unstressed i).

```
   ( [ q u ] = k )
```

This maps the letters qu to the phoneme k.

In this notation, the square brackets that are around the graphemes will be replaced by the phoneme specified; everything outside the brackets will be left in place. It is possible to define sets of graphemes, for instance the set AEO in the Spanish example

```
    (AEO a e o )
```

which defines a set of graphemes consisting of the three graphemes a, e, and o.

The last part of the script is for mapping Spanish phonemes to German HMM models.

If you think you can specify a phone set, LTS rules, and mappings to German or English HMMs in the way demonstrated here for Spanish, we can give it a try. Help on the format for all three things will be provided.

To test the Aligner for Spanish, do:

```
export ALANG=deu
export ORTHOGRAPHICINPUT2WORDLIST=/mount/studenten/synthesis/2017/bin/es.preprocess
export WORDLIST2PHONEMES=/mount/studenten/synthesis/2017/bin/es.lookup
export DICTFILE=/mount/studenten/synthesis/2017/bin/phones.de2es.dic
```

Running the Aligner

After setting the variables explained above to specify the scripts to be used instead of the default scripts by the Aligner, you can run the Aligner as described on the Aligner Help page at IMS, for instance, assuming your speech file is Spanish and called `spanish.wav`, by

```
  Alignwords spanish.wav
  Alignphones spanish.wav
```

This should generate several files, including `spanish.phones`, with the label times (end points) for phonemes, and `spanish.words`, with the label times (end points) for words.

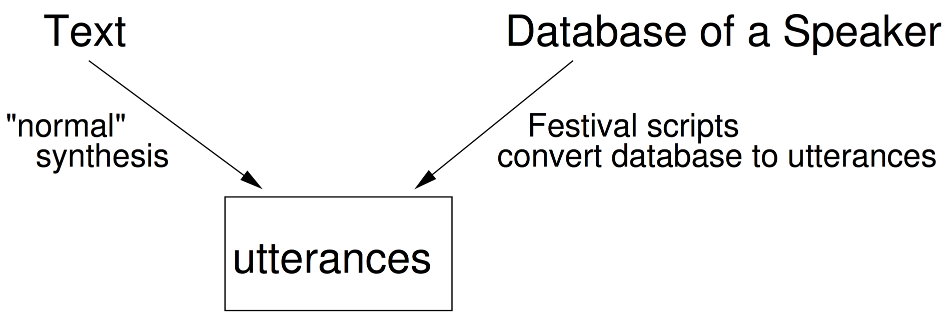
You can load them in praat using the command "Read from special tier file…" → "Read Interval tier from Xwaves…", then selecting the two tier objects, and converting to a TextGrid. This can be looked at together with the sound to check the annotations.

Converting to Festival format

Utterance structures

Utterance structures are central when synthesizing speech using the Festival speech synthesis system. They code all relevant properties of the utterance to be synthesized - in a way they are specifications of desired utterances.

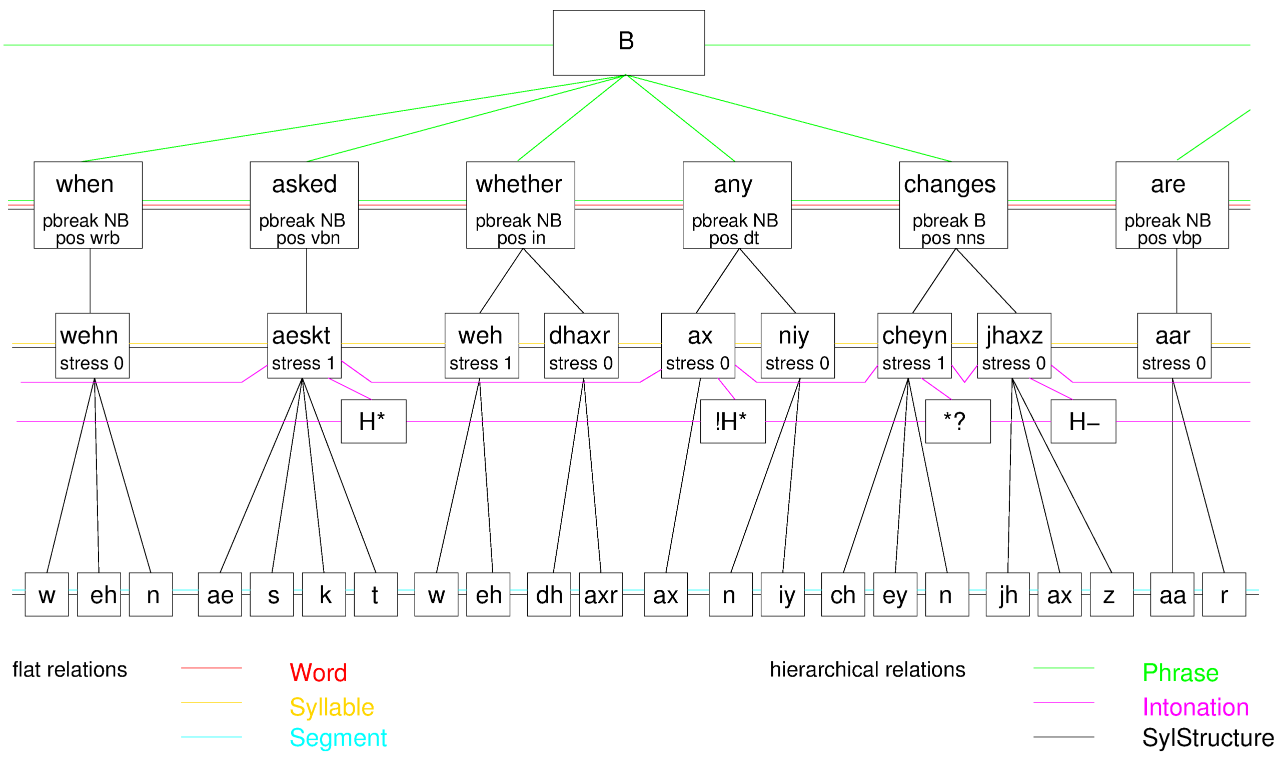
Instead of creating an utterance structure by synthesizing some text, we can create an utterance structure from a speaker's natural utterance, taking all properties from the speaker's utterance instead of predicting them from text. We then have an utterance that looks like a synthesized utterance, but which is guaranteed to be a valid, "natural" utterance, with "correct" phonetic properties.



We looked at utterance structures when you did the Festival tutorial on synthesizing speech in the interactive mode. You used functions like (utt.relationnames UTT) to inspect which relations were present in your synthesized utterances, or (utt.relation_tree UTT RELATION) to inspect which items and properties are coded in which relation.

Here's a recap. An utterance structure consists of many "items" like single words, syllables, phones, phrases, etc. Items are connected through several relations. Exactly which relations are present depends on the synthesis method, among other things. But some of them are always present, such as the Word relation connecting word items, the Syllable relation connecting syllables, the Segment relation connecting phones, and the SylStructure relation connecting items from these three relations in a hierarchy.

Here's an example of an American English utterance structure:



Some central linguistic or phonetic properties of the items are stored directly in the utterance structure in the form of features, such as part-of-speech, position of phrase breaks, and stress in the figure above. One of the most important properties, and **not shown** in the figure above, is the end time - it is stored as a feature for every single item. In our case, all these will have to be derived from the natural utterances, or more concretely, from label files for natural utterances.

(Other properties that are relevant for synthesis can be accessed dynamically by so-called feature functions, such as the duration of segments or syllables, the accent related to a word, the vowel frontness feature of a syllable's vowel, etc. These can be calculated by using linguistic and phonological knowledge which is implemented in Festival - and therefore they need not be coded in the utterance structure explicitly. Examples would be to derive durations from the end times of subsequent items, or to derive the frontness features of a vowel by accessing the phone set specification.)

Practical use of the utterance structures

In Festival, unit selection synthesis will derive the properties of the selected units from the utterance structures, and also get the start and end times for the selected units from them. Similarly, if you build a parametric system (HMM synthesis), the database from which you build the HMMs will be represented as utterances structures.

Even if you use more old-fashioned methods that do not require a speech database for concatenating units, Festival provides tools to train models for various modules using utterances structures as input. For instance, you could train a duration module using data from a database, directly in Festival, and this would require that the database is available as utterance structures. In fact, there is a recipe of how to build a duration module later in this tutorial.

Some requirements for importing utterances structures

Theoretically, we mostly need ESPS label files (which conveniently is the format the Aligner produces) for all flat relations in the utterance structure (plus some files that can be derived from the acoustics). Festival then provides a script to create utterance structures from these, linking the hierarchical relations by their time information (e.g., the script looks at the end times of items in lower relations to decide which items in a higher relation must be their parent).

However, these label files need to contain all the information we will want to use later and which cannot be calculated Festival-internally. This is the case for punctuation marks - punctuation is represented as a feature at Token level in Festival utterances. The Aligner did not differentiate between words and tokens, instead the .word label files created by the Aligner are actually what would be called tokens in Festival. So in order to have punctuation at the Token level in Festival, we would need to have that information in the .word label files, which we currently don't. Thus if we want to be able to make use of the features .punc either for unit selection, or possibly for building some component in Festival, for instance prosody prediction, we will have to integrate punctuation in the .words label files before converting to Festival utterances.

Second, the scripts for converting to utterances will create one utterance per word label file. Unfortunately, the Librivox recordings are usually quite long, and it is not possible to cut them into reasonably small pieces for Festival before running the Aligner, because then you would have to sit down and segment the .txt files accordingly.

So this leaves two "post-processing" steps to do on the alignment results: fiddle the punctuation into the .words label files, and segment the recordings along with the label files into smaller chunks. How to do this will be explained in the next section.

Post-processing the Aligner output

In order to get the punctuation marks, I have extended the tokenization scripts we used for the Aligner (i.e. the `de.preprocess` and `us.preprocess` scripts) to output punctuation in addition to the tokens (check the `de.preprocess.with.punc` and `us.preprocess.with.punc` scripts). We first use them to generate for each

`.wav` file a corresponding file with the filename extension `.preproc.result.`

```
for file in *.wav
do
basename=`echo $file | sed 's/\.wav//'`
/mount/studenten/synthesis/2017/bin/de.preprocess.with.punc $basename.txt >
$basename.preproc.result
done
```

This will loop over all `.wav` files, find out their basename by replacing the `.wav` in the filename by an empty string, and keeping this basename in a variable. It will then run the new tokenization script on the corresponding `.txt` file, and write the result to a file with extension `.preproc.result`. Note again that there are more elegant ways to write this loop, without the extra step of storing the basename in a variable, but I'm sticking to the style introduced above because it's more transparent.

We now have the list of words with punctuation marks that corresponds to all the words from the word label files generated by the Aligner, except that the Aligner has introduced pauses at some points, where our list doesn't, and that the word label file from the Aligner starts off with a line containing a # sign which in ESPS indicates the end of the (in this case empty) header.

We will now bring them together to create new ESPS word label files ending on .words.with.punc. There's loads of ways to do this; here we'll use some shell commands to get what we want (for English, please replace <P> by _SIL_):

```
# in case you use German, set your locale to Iso:
export LC_ALL=de_DE
# now generate .words.with.punc files
for file in *.preproc.result
do
basename=`echo $file | sed 's/\.preproc\.result//'`
cat $basename.words | grep --text -v "<P>" | grep --text -v "#" > \
    $basename.words.tmp
cat $basename.words | grep --text "<P>" > $basename.pauses
echo "#" > $basename.words.with.punc
paste -d ' ' $basename.words.tmp $file | cut -d' ' -f1,2,4 | cat -  \
    $basename.pauses | sort -n >> $basename.words.with.punc
rm $basename.words.tmp
rm $basename.pauses
done
```

The first iteration in the loop again gets the basename. The second eliminates pauses and the header from the Aligner word label file. The third extracts the lines of pause labels from the word label file and stores them in a file with extension .pauses (these lines are important because they contain the label times for the pauses, which we don't want to lose). We then start to create the .words.with.punc file by writing a hash sign for the end of the header into it (overwriting a possibly existing file of that name). Next we print the three columns from the word label file without pauses next to a column containing the words with punctuation marks, and then select columns 1, 2, and 4, effectively replacing the column with words only, by the column with words plus punctuation. We then use this output, append the pauses extracted earlier at the end of the output, and then sort by time - this gives us pauses and words with punctuation, with the chronological order restored. We append all this to the .words.with.punc file. We finally remove the files we don't need anymore.

Next we use a praat script I prepared that cuts long speech signals at the pauses specified in an ESPS label file. It takes the .wav file, the accompanying .words and .phonemic files, converts them to a textgrid, then loops through the grid and cuts the signal into segments at every pause that is at least 0.3 seconds long. It leaves 0.1s of silence at the beginning and at the end of every segment, and saves the resulting files together with accompanying textgrids.

```
for file in *.words.with.punc
do
iconv -f iso-8859-1 -t utf-8 $file > $file.utf8
basename=`echo $file | sed 's/\.words\.with\.punc//'`
# watch out: no newline between the following two lines,
# this should be one single command!!
praat --run /mount/studenten/synthesis/2017/bin/cut.praat $basename.wav
    $file.utf8 $basename.phonemic
rm $file.utf8
done
```

Note that since praat expects UTF-8 coded input, we first convert the .words.with.punc files to UTF-8, then run the script giving the UTF-8 coded .words.with.punc.utf8 files and the original .phonemic file (which doesn't contain any special characters and therefore needs not be converted) as arguments. This is by the way how you run praat scripts on the command line, using the --run option to praat. Note however that this only works if the praat script was implemented in a way that does not make use of any Editor commands (this is for the praat experts, if you don't know what Editor commands are, never mind). The script will create lots of files called $basename.part_XYZ, where XYZ is a number which increases for every new segment.

The only thing left to do then is to convert the textgrids for these segments back to ESPS label files, because that's the format Festival expects. **Before doing that, take time to open some of the segmented .wav files in praat, along with their .textgrids, and check whether everything looks ok (look at them together using View&Edit)**. This is a really important step, and in my experience many errors that are encountered later in the process can be anticipated if you take some time to make sure that your data have been processed with extreme diligence.

If everything looks absolutely correct and as desired, convert to ESPS like this:

```
for file in *.part*.TextGrid
do
basename=`echo $file | sed 's/\.TextGrid//'`
# no newline here
praat --run /mount/studenten/synthesis/2017/bin/grid2esps.praat $file 2 >
    $basename.phonemic
# and none here
praat --run /mount/studenten/synthesis/2017/bin/grid2esps.praat $file 1 >
    $basename.words
done
```

Set up the directory structure

In the following, we will use tools from the Festvox distribution and tools from the Edinburgh Speech Tools Library to process our database and create the unit selection voice. Festvox is a project that is meant to assist in building new synthesis voices. It can be downloaded from www.festvox.org; at IMS it is already installed at `/mount/resources/speech/festival/Festvox_2.7`. The Edinburgh Speech Tools Library contains programs to process speech signals, and it is included in the Festival speech synthesis system. Festvox has links to the website where Festival and the Speech Tools can be downloaded in case you want to install them on your computer at home; both are already installed at IMS below `/mount/resources/speech/festival/Festival_2.4`.

So we first let our Terminal know where to find Festvox and the Speech Tools:

```
export FESTVOXDIR=/mount/resources/speech/festival/Festvox_2.7/festvox
export ESTDIR=/mount/resources/speech/festival/Festival_2.4/speech_tools
```

We first create the directory structure and copy the tools and scripts needed for processing our speech data. There's a Festvox script called `setup_clunits` that can take care of this. It needs three arguments which are by convention: i) the institution ii) the language iii) the speaker's name. Let's use en for British English, us for American English, de for German, and pick a nice name for your speaker (you'll need to type this name often, so maybe not too complicated).

First create a new directory called `myvoice` for building the voice, and then run the following command inside the new directory. Adapt the arguments, this example here assumes a German speaker named Fritz:

```
$FESTVOXDIR/src/unitsel/setup_clunits ims de fritz
```

This should create lots of directories, for instance a bin directory with most relevant scripts, a wav directory for the speech files, a festival directory in which we'll keep all extracted information that is needed for selection (including the utterance structures), a festvox directory which has templates for the synthesis modules which we will need to adapt to get the new voice running, and several directories for various acoustic features (like f0, pm for pitchmarks, etc.)

Audio files

Move all short .wav files into the `wav` directory within your `myvoice` directory (not the long ones that we used for alignment!!). I recommend to also move the .phonemic files and the .words files to this directory, so everything is at the same place.

Collecting label files for flat relations

The next steps convert the label files generated by the Aligner (or rather, the small snippets we created from them) and put them into the places where Festvox expects them. **Before you run them, please read the sections regarding lexicon and phone set because you need to decide on your phone inventory before creating the phones and syllable label files.**

Let's first create the directories where Festival will look for the label files when creating the utterance structures. We need a directory called `relations` within the `myvoice/festival` directory. In this directory, Festival expects one directory for each relation, and the directory needs to have the name of the Festival relation. Inside each directory, we will have ESPS label files, with the same basename as the corresponding `.wav` files, and with the relation name as extension. So for instance the words label files will be expected in `myvoice/festival/relations/Word` because the name of the Festival relation for words is `Word`, and there extension will also be `.Word`.

```
  cd myvoice
  # create the directories
  mkdir -p festival/relations/{Segment,Syllable,Word,Target,Token,Phrase}
```

Next, let's fill the Segment directory. This is just an example, please adapt – and only after having read the sections below on phone set definition and lexicon!!!

In this example (a German one) we map the label that the Aligner gives for pauses, _p:_, to the _ symbol, because that's what the phone set will define as silence symbol. Also, for technical reasons the Aligner has introduced _ symbols before all phone labels that start with a number, so this has to be revised, too. For English data, you will definitely need different mappings. The example assumes that you have indeed moved your `.phonemic` files to the `wav` directory, as recommended above.

```
cd wav
# run Aligner script for converting .phonemic to .phones first
export ALIGNERHOME=/mount/resources/speech/aligner/Aligner
export ALANG=deu
for file in *.phonemic
do
$ALIGNERHOME/bin/$ALANG/phonemic2phones $file
done
# map _p:_ to _, and remove underscore before phonemes 2, 6 and 9
for file in *.phones
do
basename=`echo $file | sed 's/\.phones//'`
cat $file | sed 's/_p:_/_/' | sed 's/_2/2/g' | sed 's/_6/6/g'|\
   sed 's/_9/9/g' > ../festival/relations/Segment/$basename.Segment
done
```

Next we will generate the Syllable label files. Festival needs a stress attribute for marking word stress. I have prepared a script called `phonemic2syllables.withstress`, which converts `.phonemic` files to appropriate syllable label files including stress. (Whereas the Aligner's own `phonemic2syllables` script would discard stress.)

```
# Syllable
# Festival needs word stress
# this is coded in the .phonemic files
for file in *.phonemic
do
basename=`echo $file | sed 's/\.phonemic//'`
/mount/studenten/synthesis/2017/bin/phonemic2syllables.withstress \
$file ../festival/relations/Syllable/$basename.Syllable
done
```

Check the resulting `.Syllable` files to see how the word stress is coded: it is specified as a attribute-value pair, and separated by a semicolon from the word label itself. We will need the same format below for the Tokens.

Festival distinguishes between Token (which can be normal words but also unexpanded abbreviations etc.) and Word (the expanded "spelled out" forms). Punctuation is expected as a feature at Token level, but not at word level. The commands below are again an example. Note that if you work with German you have to set your `LC_ALL` variable to `de_DE` otherwise `grep` will misbehave. If you want to paste the commands below into your Terminal, you should also set the encoding in your Terminal to iso-8859-1 (in the menu of the Terminal). For English this should not be necessary. The sequence of commands below converts the utf-8 word files from praat to iso, discards pauses, and then uses a mean looking perl one-liner to isolate punctuation marks before or after the word labels and to rewrite them as features.

Neither the Token nor the Word level in Festival should contain pauses, so these have to be filtered, too.

Watch out, if you have English data, you may have to adapt the pause label in the commands below. (Also, you can probably get rid of the umlauts.)

```
# Token: split punctuation marks from words, write as separate features
# convert to iso for Festival
# don't process pauses
# for grep to work even for German umlauts, set locale to iso if German!
export LC_ALL=de_DE
for file in *.words
do
basename=`echo $file | sed 's/\.words//'`
cat $file |  iconv -f utf-8 -t iso-8859-1 | grep -v "<P>" |\
 perl -ne 'if ( m/\#/) {print; } else {chomp;
 m/^(.* 121 )(\"*)([A-Za-zäöüÄÖÜßóé0-9\-.]*[A-Za-zäöüÄÖÜßóé0-9\-])([^A-Za-zäöüÄÖÜßóé0-9\-]*)/;
 print "$1$3"; if ($2 ) { $prepunc = $2; $prepunc =~ s/"/\\"/g;
 print "; prepunc \"$prepunc\""; } if ($4) {$punc = $4;
 $punc =~ s/"/\\"/g; print "; punc \"$punc\"";}
 print "\n"}' > ../festival/relations/Token/$basename.Token
done

# Words: delete punctuation marks from words
# convert to iso for Festival
# don't process pauses
for file in *.words
do
basename=`echo $file | sed 's/\.words//'`
cat $file |  iconv -f utf-8 -t iso-8859-1 | grep -v "<P>" |\
 perl -ne 'if ( m/\#/) {print; } else {chomp;
 m/^(.* 121 )(\"*)([A-Za-zäöüÄÖÜßóé0-9\-.]*[A-Za-zäöüÄÖÜßóé0-9\-])([^A-Za-zäöüÄÖÜßóé0-9\-]*)/;
 print "$1$3"; print "\n"}' > ../festival/relations/Word/$basename.Word
done
```

Next we need label files for phrases, which we don't have. We'll approximate: we cut our files into pieces at pauses. So we are sure at the end of every .wav files, there was a pause, and thus, at the last word in every file, there should be a phrase boundary (at least there!). We exploit this and use the tail -1 command below to get just the last line of each .words label file (disregarding pauses), then take just the first two columns (label time and color), append a BB, and write this line into the Phrase label file. Don't forget to adapt if your data is different (for instance pause labels).

```
# phrases: assume last word has a phrase boundary
# so generate label files that have the label time of the last words
# but a label BB for Big Break
for file in *.words
do
basename=`echo $file | sed 's/\.words//'`
echo "#"  > ../festival/relations/Phrase/$basename.Phrase
cat $file |  grep -v "<P>" | tail -1 | cut -d' ' -f1,2 | sed 's/$/ BB/' \
>> ../festival/relations/Phrase/$basename.Phrase
done
```

Creating additional relations from the speech signal

Festival also wants label files that contain the F0 values, which it will use as F0 targets in the Target relation. We use the `get_f0` program, which is installed at IMS, to calculate raw F0 contours first. Then we use a program called `f02lab` to convert these to label files with F0 values.

```
# calculate F0 using get_f0
for f in *.wav
do
echo $f
basename=`echo $f | sed 's/\.wav//'`
get_f0 $f ../festival/relations/Target/$basename.f0
done


# Target files: convert contours to label files
cd ../festival/relations/Target
for f in *[0-9].f0
do
basename=`echo $f | sed 's/\.f0//'`
/mount/studenten/synthesis/2017/bin/f02lab $basename
# f02lab creates .f0lab file, move to correct name
mv $basename.f0lab $basename.Target
done
```

Building utterance structures

We now should have everything ready to build utterance structures. This is done from the myvoice directory, not in the Target directory where we ended up.

```
cd ../../..
/mount/studenten/synthesis/2017/bin/make_utts -tokens \
festival/relations/Segment/*.Segment
```

This should give utterance structures in festival/utts. Check them by loading them in Festival in the following way: (replace somename with an appropriate file name)

```
/mount/resources/speech/festival/Festival_2.4/festival/bin/festival
(set! utt (utt.load nil "festival/utts/somename.utt"))
(utt.relationnames utt)
(utt.relation_tree utt 'Token)
```

The command should be familiar from the tutorial on the interactive mode in Festival. The only difference is that instead of synthesizing the utterance we just load it from the file we have created.

Take the time to look at relations other than the Token relation, too, and check not only one file, but a few. It is really important that we don't overlook any systematic errors at this point.

It's likely that the pauses are not yet correctly recognized as pauses, but we'll get to that later.

Defining modules for the new voice

Defining the phone set

Before creating the utterance structures, let's have brief look at what a phone set definition looks like in Festival. Here's one for English:

```
(defPhoneSet
  radio
  ;;;  Phone Features
  (;; vowel or consonant
   (vc + -)
   ;; vowel length: short long diphthong schwa
   (vlng s l d a 0)
   ;; vowel height: high mid low
   (vheight 1 2 3 0)
   ;; vowel frontness: front mid back
   (vfront 1 2 3 0)
   ;; lip rounding
   (vrnd + - 0)
   ;; consonant type: stop fricative affricate nasal lateral approximant
   (ctype s f a n l r 0)
   ;; place of articulation: labial alveolar palatal labio-dental
   ;;                          dental velar glottal
   (cplace l a p b d v g 0)
   ;; consonant voicing
   (cvox + - 0)
   )
  ;; Phone set members
  (
   ;; Note these features were set by awb so they are wrong !!!
   (aa  +   l   3   3   -   0   0   0) ;; father
   (ae  +   s   3   1   -   0   0   0) ;; fat
   (ah  +   s   2   2   -   0   0   0) ;; but
   (ao  +   l   3   3   +   0   0   0) ;; lawn
   (aw  +   d   3   2   -   0   0   0) ;; how
   (ax  +   a   2   2   -   0   0   0) ;; about
   (axr +   a   2   2   -   r   a   +)
   (ay  +   d   3   2   -   0   0   0) ;; hide
   (b   -   0   0   0   0   s   l   +)
   (ch  -   0   0   0   0   a   p   -)
   (d   -   0   0   0   0   s   a   +)
   (dh  -   0   0   0   0   f   d   +)
   (dx  -   a   0   0   0   s   a   +) ;; ??
   (eh  +   s   2   1   -   0   0   0) ;; get
   (el  +   s   0   0   0   l   a   +)
   (em  +   s   0   0   0   n   l   +)
   (en  +   s   0   0   0   n   a   +)
   (er  +   a   2   2   -   r   0   0) ;; always followed by r (er-r == axr)
   (ey  +   d   2   1   -   0   0   0) ;; gate
   (f   -   0   0   0   0   f   b   -)
   (g   -   0   0   0   0   s   v   +)
   (hh  -   0   0   0   0   f   g   -)
   (hv  -   0   0   0   0   f   g   +)
   (ih  +   s   1   1   -   0   0   0) ;; bit
   (iy  +   l   1   1   -   0   0   0) ;; beet
   (jh  -   0   0   0   0   a   p   +)
```

```
    (k   -   0   0   0   0   s   v   -)
    (l   -   0   0   0   0   l   a   +)
    (m   -   0   0   0   0   n   l   +)
    (n   -   0   0   0   0   n   a   +)
    (nx  -   0   0   0   0   n   d   +) ;; ???
    (ng  -   0   0   0   0   n   v   +)
    (ow  +   d   2   3   +   0   0   0) ;; lone
    (oy  +   d   2   3   +   0   0   0) ;; toy
    (p   -   0   0   0   0   s   l   -)
    (r   -   0   0   0   0   r   a   +)
    (s   -   0   0   0   0   f   a   -)
    (sh  -   0   0   0   0   f   p   -)
    (t   -   0   0   0   0   s   a   -)
    (th  -   0   0   0   0   f   d   -)
    (uh  +   s   1   3   +   0   0   0) ;; full
    (uw  +   l   1   3   +   0   0   0) ;; fool
    (v   -   0   0   0   0   f   b   +)
    (w   -   0   0   0   0   r   l   +)
    (y   -   0   0   0   0   r   p   +)
    (z   -   0   0   0   0   f   a   +)
    (zh  -   0   0   0   0   f   p   +)
    (pau -   0   0   0   0   0   0   -)
    (h#  -   0   0   0   0   0   0   -)
    (brth -   0   0   0   0   0   0   -)
   )
 )

 (PhoneSet.silences '(pau h# brth))

 (provide 'radio_phones)
```

The whole thing is just one huge Scheme command (and therefore in parentheses): the command `defPhoneSet` defines a phone set, and it takes three arguments: the name for the phone set, a list of features including a specification of which values are valid for each feature, and a list of phones along with their features specifications:

```
(defPhoneSet NAME FEATURES PHONES)
```

The features, i.e. the second argument, are a list of lists. In each list, the first element is the name of the feature, and the other elements are its possible values. For instance,

```
(vc + -)
```

defines a feature vc (short for vocalic), and states that its value is either + (for vowels!) or – (for consonants!). This particular phone set was created by Alan W. Black, and he was nice enough to put comment lines (starting with semicolons) into the scheme code to explain each feature, which is useful in all cases which are not as obvious as the `vc` feature above, for instance here:

```
    ;; place of articulation: labial alveolar palatal labio-dental
    ;;                          dental velar glottal
    (cplace l a p b d v g 0)
```

So the features values for place of articulation ("`cplace` is probably short for consonant place") are short for `labial`, `alveolar`, ... etc. Note that there is a 0 included – this is intended for vowels, which don't have a place of articulation.

After the list of features, we have a list of phonemes, where each phoneme again is a list with its name as first element, and the feature values for the features defined above in the same order. Note that Alan W. Black also commented on this list of phonemes saying the values are probably wrong (because he is not a phonetician). Here's again an example taken from the definition above

```
    (iy  +   l   1   1   -   0   0   0) ;; beet
```

This defines phoneme `iy` (which is equivalent to SAMPA "i" or "i:"). The comment tells us that this is the same vowel as in the word "beet". The first feature value then is "+", and since the first feature above was `vc`, we now now that iy is a vowel (has the value "+" for vocalic). The next feature value is l, for the second feature, which was `vlng` for vowel length – so this is a long vowel. The next two values are 1, so we know that this is a high and front vowel. Next is "-", so unrounded (`vround` feature). All remaining features only make sense for consonants, and obviously Alan W. Black decided that these should be 0 for vowels.

If you want to design a new phone set, you will have to decide on a phoneme inventory. If you can get hold of an IPA table for that phoneme inventory, you will find many features implicitly in this table: the IPA table for consonants organizes phones by `cplace` in the columns, and by `ctype` in the rows; it usually has pairs of consonants in each cell where one is voiced, and one isn't. For the vowel features, look at the IPA vowel diagram to find out which vowels are high, mid, or low, and which are front, mid, or back. If there are pairs, one usually rounded, and one isn't; if there are no pairs, you will have to rely on your articulation to decide about rounding...

Once the phone set is defined, there are two commands left in the above example code: the first one defines which phones are considered pauses. The second one, the `provide` command, declares the phone set definition as a sort of "module" (called `radio_phones` in this case), and says that the code here is what is needed for this module. Other modules then can say that they rely on this module (they would have a line saying `(require 'radio_phones)`) and then Festival would know that it needs to make sure `radio_phones` has been defined.

The place to put this phoneset is the `festvox` directory in your `myvoice` directory: there should be a file called `ims_de_speakername_phoneset.scm` or similar, and if you have a language for which no phone set is contained in Festival, it is indicated in this file where your definition should go, and what name your phone set should have. If you have an American English voice then Festvox seems to generate the above file without a template for a new phoneset definition; instead it uses the `(require 'radio_phones)` command mentioned above to make sure that the radio phoneset, which is part of Festival anyway, is already loaded. The next two lines then simply state that your voice will select this phoneset.

```
  (Parameter.set 'PhoneSet 'radio)
  (PhoneSet.select 'radio)
```

You may have to remove the line which requires the radio phones, and insert your own phone set here. If you do this, make sure that you also change the two commands which select the phoneset so they select the new phoneset instead of `radio`.

In case you are dealing with a British English rather than an American English voice, you might want to use the mrpa phoneset, which is also already implemented in Festival. In this case, replace "radio" by "mrpa" in the above commands.

When you modify this file to adapt it to your voice, keep in mind that whatever you call your phones here should be consistent with two things: i) the label files you have provided above for creating the utterances: you need phone level label files where every label that occurs in your label file corresponds to a phone name in your phone set. And ii) the phone names have to be consistent with what comes from the lexicon we will use for lookup. In theory it doesn't really matter which of the three things you change (lexicon, phone set, label files) – they only have to be consistent. In practice however I'd suggest to leave the lexicon as it is because it's the biggest file.

So where do we get a lexicon?

Find a lexicon

Festival comes with two English lexicons: the CMU lexicon (American English) and the OALD lexicon (Oxford Advanced Learners' Dictionary, British English). You can find them at

/mount/resources/speech/festival/Festival_2.4/festival/lib/dicts

I've also put the BOMP lexicon for German there, that's an older dictionary that used to be open source. In its original form it is not provided any more (the institute that created it does not exist anymore). However IMS still distributes the Festival version of it, and it should be ok to use it here. I put two variants of BOMP into the bomp subdirectory at the above location, can you figure out which one you should use?

If you ever want to build a voice where you need a different lexicon – the Festvox project (see above) and the Festival documentation provide information on how to compile a new lexicon for Festival.

Check the lexicon you will need and have a look at its phone set.

The next steps are

- go back to the section on converting to Festival format, and create the `.Segment` and `.Syllable` label files in a way that the phone inventory is consistent with this lexicon
- specify your phone set following the example above, so it is consistent with this lexicon
- do the remaining steps listed above for converting to Festival format.

Configure the Lexicon

Have a look at the file `festvox/ims_LANGUAGE_SPEAKER_lexicon.scm`. **(Please replace `LANGUAGE` by your language and `SPEAKER` by your speaker name here and in the rest of this description.)** If you have specified a language other than English when setting up the directory structure for your project, then the code in this file first defines something called an addenda lexicon (for English, see further below). An addenda lexicon in Festival is an additional lexicon which you can use to add custom entries to your "real" lexicon. There are many lines commented out (the ";" marks the rest of the line as comment in Scheme), but you can see that you could for instance add the single letters of the alphabet and their pronunciation. Of course normal words that happen not to be in the lexicon could be added in the same way. The addenda should minimally contain all punctuation marks (and give them an empty pronunciation).

In the following there are a number of functions used for getting the pronunciations of unknown words, which we will not look at in detail. But further below, you will find a part that is marked as "Lexicon definition". Here, the name of the lexicon, the name of the phone set for this lexicon, the compiled lexicon file that has to be loaded are specified. At this point the function for the addenda lexicon defined above is called, i.e. at this point all the entries from above are added. If such a file exists, this will also load a second addenda lexicon from a separate file (`festvox/ims_LANGUAGE_addenda.scm`). Note that even though these commands define the lexicon, they don't yet specify that this lexicon will be used (but they do make the addenda entries available at this point).

Next, in the part marked as "Lexicon setup" a function called `ims_LANGAUGE_SPEAKER::select_lexicon` is defined. This function would, if called, finally really select the lexicon defined above. This function will later be called when selecting our newly built voice.

So what do you have to do to get your new voice to used the lexicon you chose above? The straightforward way is to adapt the Lexicon definition part above and change the name of the compiled lexicon file to the file you want. If you use the lexica I suggested above, just adapt the path to the compiled file. Originally it is specified like this:

```
   (path-append ims_LANGUAGE_SPEAKER::dir "festvox/ims_LANGUAGE_lex.out")
```

Here, `ims_LANGUAGE_SPEAKER::dir` is the directory where you have your voice, and the path-append function above will piece the path together taking the path to your voice directory and appending "`festvox/ims_LANGUAGE_lex.out`". You can replace the whole command by a string (in quotes, of course!) giving the full path to the lexicon I suggested. If you had to make any modifications to this file, you could alternatively put the outcome of that into a file `festvox/ims_LANGUAGE_lex.out` and leave the code unchanged.

If you are dealing with English, then you should find lines in your ims_LANGUAGE_SPEAKER.scm file that configure and select an English lexicon, e.g.

```
  (setup_cmu_lex)
```

and

```
  (lex_select "cmu")
```

These would use the US English lexicon; replacing oald for cmu should give you the British English OALD.

A first unit selection voice

We now have a database ready where we have utterance structures for the whole database. We will use these structures mostly for determining phonological or linguistic properties of the units in this database: When we will synthesize text, we will first calculate which units we would optimally need to synthesize the whole text. Then we need to calculate the target costs for various candidates in the database, and we do this by calculating the weighted distance between the vector of phonological and linguistic properties of the unit that we need and the vectors of various unit candidates from our database. These vectors can be derived from the utterance structures.

In addition, we will make use of acoustic features, mostly for calculating concatenation costs, and for clustering our units to make synthesis more efficient, so the next thing to do is to run some scripts that extract acoustic parameters from our speech signal files.

These scripts are again provided by Festvox. Unfortunately most of them rely on a file `etc/txt.done.data`, which we would have created if we had recorded the voices ourselves, using Festvox tools. This file is like an index file: it lists all file basenames and the corresponding text. However we never recorded our own data, so we don't have them. We will instead create a "fake" version of this file now so the Festvox scripts will work. I will call this the file list in the following.

Creating the file list

The file list is (of course, haha) in Scheme notation. Each line is supposed to look like this:

```
( chapter1.part45 "This is what was said in the file." )
```

i.e. we need the file basename without extension, and then some string. If we had used this for recording the voice, we would have needed the correct text in here, but for our purposes, the string never gets evaluated, so we might as well make our lives a tiny bit easier and just always write "DUMMY" instead of some meaningful text. Here's a loop that would do this if you are in the `myvoice` directory.

```
rm etc/txt.done.data
for f in wav/*.wav
do
basename=`echo $f | sed 's/wav\///;s/\.wav//'`
echo "( $basename \"DUMMY\" )" >> etc/txt.done.data
done
```

The `rm` command in the first line is only there to make sure that if you have run this set of commands before, the file is removed before writing to it again (otherwise we'll have repetitions in the list).

Pitchmarks

Next we need pitchmarks. Pitchmarks mark where new glottal cycles begin (and since we will always want to concatenate units in a way that the periods in the two units are synchronous, the quality of the pitchmarks is very important.)

There is a lot of documentation on how to generate pitchmarks, and what they should look like at

http://www.festvox.org/bsv/x863.html

To generate them for all wavs in your wav directory, do

```
bin/make_pm_wave wav/*.wav
```

This will create the pitchmark files in the `pm` directory. Before you actually run this command for all `.wav` files, kill the process once it has processed the first few files. You will most probably have to change parameters in the `bin/make_pm_wave` script, so it's a waste of time to run it for all your files before having found the right parameters.

Next we want to inspect the results. **Please do take some time to make sure that the pitchmarks look they way they should. The URL above describes in detail what we want, and also which parameters you have to modify in the `make_pm_wave` script.**

For inspection, we generate label files that tell us where the script above finds pitchmarks. The following command does it for all the pitchmark files in the pm directory. It writes these label files to the directory `pm_lab`. Note that we will not need the label files for synthesis later, we will need just the pitchmarks themselves (i.e. the files in the `pm` directory).

```
bin/make_pmlab_pm pm/*.pm
```

Also note that at the URL above, there is a typo in the documentation – instead of this `bin/make_pmlab_pm` command, they say `bin/make_pm_pmlab`, which is wrong (`pmlab` and `pm` interchanged).

For displaying the label files, the URL above suggests `emulabel`, but we don't have it installed, and I recommend to either use `praat` (load the `.wav` by R`ead from file…`, then the label file from the `pm_lab` directory by `Read from special tier file… Read text tier from Xwaves`, convert to `TextGrid`, select TextGrid and Sound, click `View&Edit`). Or you can use `wavesurfer`, a much less powerful, but easier to use tool for displaying speech signals and label files. You can start `wavesurfer` in the following way to directly load the `.wav` and corresponding label file:

```
 export scriptsdir=/mount/studenten/synthesis/2017/bin/
 wavesurfer -config $scriptsdir/Pitchmarks.conf wav/myfile_part4.wav
```

Once you are satisfied with the result, run the following lines to make sure that the pitchmarks are aligned with the highest peak

```
# make sure that FESTVOXDIR AND ESTDIR are set
export FESTVOXDIR=/mount/resources/speech/festival/Festvox_2.7/festvox
export ESTDIR=/mount/resources/speech/festival/Festival_2.4/speech_tools
export LD_LIBRARY_PATH=$ESTDIR/lib
# correct pitchmarks to next peak
bin/make_pm_fix pm/*.pm
```

MFCCs

The last thing left to do before we can build clusters is to generate the cepstral coefficients (MFCCs). These are used for concatenation costs and for clustering the units. This process takes a while.

```
bin/make_mcep wav/*.wav
```

Clustering the units

Before clustering, you need to make sure that your phone set is defined. Check whether the file festvox/ims_lang_speakername_phoneset.scm exists (replace lang by us, en, or de, and speakername by the name you chose for the speaker) and whether you have specified a phoneset there. See the section on phonesets above if anything is missing.

```
export FESTIVAL=$ESTDIR/../festival/bin/festival
$FESTIVAL -b festvox/build_clunits.scm '(build_clunits "etc/txt.done.data")'
```

Start this process. It will first output lines like these:

```
units "m" 1306
units "x" 303
units "a:" 776
```

Then you will most probably get many warnings like these:

```
Phone p does not have feature vrnd
Phone R does not have feature vrnd
```

or these

```
./festival/feats/_.feats: bad value e: in field n.name vector 1105
./festival/feats/_.feats: bad value Y in field pp.name vector 1105
```

These warnings come about because for clustering, Festival needs to calculate values for all features that we want to use for clustering. Which features we want to use is specified in a file `festival/clunits/all.desc`, along with their possible values. So the first two warnings above tell us that some features were not known to Festival (in this case, the phoneset didn't specify the requested feature). The other two warnings indicate that Festival was able to calculate the requested features, but the values obtained are invalid.

We should get a working voice in spite of the warnings, so for a first version, we'll just ignore them. But we will come back to this later.

Synthesis with the new voice

We are now ready to try out the new voice for the first time!! Start Festival with the file that contains the voice definition, `festvox/ims_de_antje_clunits.scm`:

```
$ESTDIR/../festival/bin/festival festvox/ims_de_antje_clunits.scm
```

Then load the voice (replace `lang` by your language, and `speaker` by your speaker name below), and synthesize an utterance:

```
(voice_ims_lang_speaker_clunits )
(set! utt (SayText "Something."))
```

Analyzing the result

There may be a number of problems with the synthesis voice in its current state. For instance, if you have a language other than English, you probably will not be able to synthesize numbers yet. Also, we didn't do anything about text preprocessing for other non-standard words. This can be addressed below, by defining modules for these purposes. In the meantime, we will have to spell everything out.

Similarly, we might have out-of-vocabulary words. We can either avoid them in the input text, or add these words to our addenda lexicon (see section on lexicon setup above).

Assuming we avoid OOV-words and non-standard words, we should be able to hear something. However, the quality may be suboptimal. This often happens when there are errors in the annotation of the database.

Here is how to check how your sentence was synthesized. Assume we have synthesized an utterance and stored it in a variable called utt (see Festival tutorial at the beginning of this class!):

```
(set! utt (SayText "Das ist eine kleine Synthese mit nur zwei Stunden
Sprache. "))
```

We can check which relations are present by

```
(utt.relationnames utt)
```

You should hopefully have the following relations:

```
(Token
 Word
 Phrase
 Syllable
 Segment
 SylStructure
 IntEvent
 Intonation
 Target
 Unit
 SourceSegments
 Wave)
```

For checking if the phonemes are as expected, I recommend checking the SylStructure relation: here you can see the words, the corresponding syllables and the phonemes in these syllables. This is a way to check if the pronunciations were correct:

```
(utt.relation_tree utt 'SylStructure)
```

Alternatively, you can just check which pronunciation your lexicon has for a specific word, to that end, do:

```
(lex.lookup "Something")
```

For checking which units were actually concatenated (and tracking down errors in the annotation that gave you wrong phonemes even though the pronunciation was predicted ok), check the Unit relation:

```
(utt.relation_tree utt 'Unit)
```

The items in this relation are the units that got selected. You should see that they are phone-sized units (and their names consist of the unit name plus an ID number). As features, you will find the fileid (the name of the file that the unit came from), and seg_start and seg_end features (giving the times of the labels in the database). There are also unit_start and unit_end features, which may have been modified from the original segment start and end times in cases where units have been joined by optimal coupling.

If there are unintelligible parts in your utterance, check the units around that part and see if the labels from the database were correct, for instance using praat to load the segmented wav files along with the textgrids you created when segmenting the long files.

How does this relate to the stuff from the lecture?

Candidates

You can find a catalogue containing all the candidates in your database in `festival/clunits/ims_LANGUAGE_SPEAKER.catalogue`. It specifies phoneme type and ID, the filename, and the start, middle, and end times of these units.

Features for clustering (and target costs)

The features for clustering (and thus implicitly for the target costs) are read from your file `festival/clunits/all.desc`. It lists the features to be used for clustering along with all possible values that these features can take.

This file is copied when you first set up the directory structure for your voice. However it used English defaults, so this should be adapted to features that make sense for your voice. For instance, it uses lots of phone set features of the current phone as well as of preceding (indicated by path names p., pp.) or of following (n., nn.) phones. These phone set features can be recognized by the ph_ prefix. You should verify that your `all.desc` file is consistent with your phone set: make sure that both the feature names and the possible values are valid. If this is consistent, there will no warnings like these left:

```
Phone p does not have feature vrnd
Phone R does not have feature vrnd
./festival/feats/_.feats: bad value e: in field n.name vector 1105
./festival/feats/_.feats: bad value Y in field pp.name vector 1105
```

Cluster trees

You can find the cluster trees in `festival/trees`. There should be one tree for each phoneme type. They are plain ascii files, you can look at them using less or cat, or any text editor. They are (of course…) in Scheme format – nodes are 3-element lists where the first element is a statement about the features you specified, the second is the subtree that has to be accessed if the statement is true, the third the subtree that is relevant if the statement is false. Clusters are at the leaves in the form of (lists of) … lists of indices of specific candidate units (same index as in the catalogue file above) along with their target costs, i.e. their distance to the centroid (at least that's what I think, the documentation is very silent about this) and the last element of this embedded list is probably the mean distance of this cluster.

You can specify some parameters for synthesis in the file `festvox/ims_LANGUAGE_SPEAKER_clunits.scm`. It defines a number of parameters that you can modify, and one of them is the `extend_selections` parameter:

```
(set! ims_LANGUAGE_SPEAKER::dt_params
 ...
        '(extend_selections 2)
...)
```

This parameter should alleviate the problem mentioned in the slides that in case of subsequent units, it might happen that one of them is not in the appropriate cluster. If the `extend_selections` parameter is set to n, then the n units which are subsequent to a candidate in the database are added to the candidate selection for subsequent units in case they are of the correct phoneme type.

Further parameters to modify are in `festvox/build_clunits.scm`. Here, you can also modify cluster sizes and the parameters for pruning. All parameters that influence the behavior in clustering are described in some detail here `www.festvox.org/bsv/c2645.html`

Optimal coupling and concatenation parameters

You can specify whether you want to use optimal coupling or not in festvox/`ims_LANGUAGE_SPEAKER_clunits.scm`. Among the `ims_LANGUAGE_SPEAKER::dt_params` parameters you also find

```
        '(optimal_coupling 1)
```

If you want to switch off optimal coupling, change the parameter to 0. You can also try to change the `join_method`, the `f0_join_weight` or the `continuity_weight` parameters.

More parameters

More parameters that influence the behavior in clustering, in candidate selection or in concatenation are described in some detail here

www.festvox.org/bsv/c2645.html

They should all be in either of the two files `festvox/ims_LANGUAGE_SPEAKER_clunits.scm` or `festvox/build_clunits.scm`.

Defining more modules

Synthesis architecture

In the file `festvox/ims_LANGUAGE_SPEAKER_clunits.scm` you can also find a specification of which modules are used in synthesis. Relatively at the start of the file, you can find the following lines, which just load the corresponding files form the `festvox` directory. Each of the files contains code which is related to the specific module.

```
(require 'ims_LANGUAGE_SPEAKER_phoneset)
(require 'ims_LANGUAGE_SPEAKER_tokenizer)
(require 'ims_LANGUAGE_SPEAKER_tagger)
(require 'ims_LANGUAGE_SPEAKER_lexicon)
(require 'ims_LANGUAGE_SPEAKER_phrasing)
(require 'ims_LANGUAGE_SPEAKER_intonation)
(require 'ims_LANGUAGE_SPEAKER_duration)
(require 'ims_LANGUAGE_SPEAKER_f0model)
(require 'ims_LANGUAGE_SPEAKER_other)
```

In order to change specific modules, you should usually edit the corresponding files.

Further down in `festvox/ims_LANGUAGE_SPEAKER_clunits.scm` you can find the definition of your new voice. This basically ensures that the functions which are defined in the above files are actually used when synthesizing with your voice. Here's what it looks like in my case:

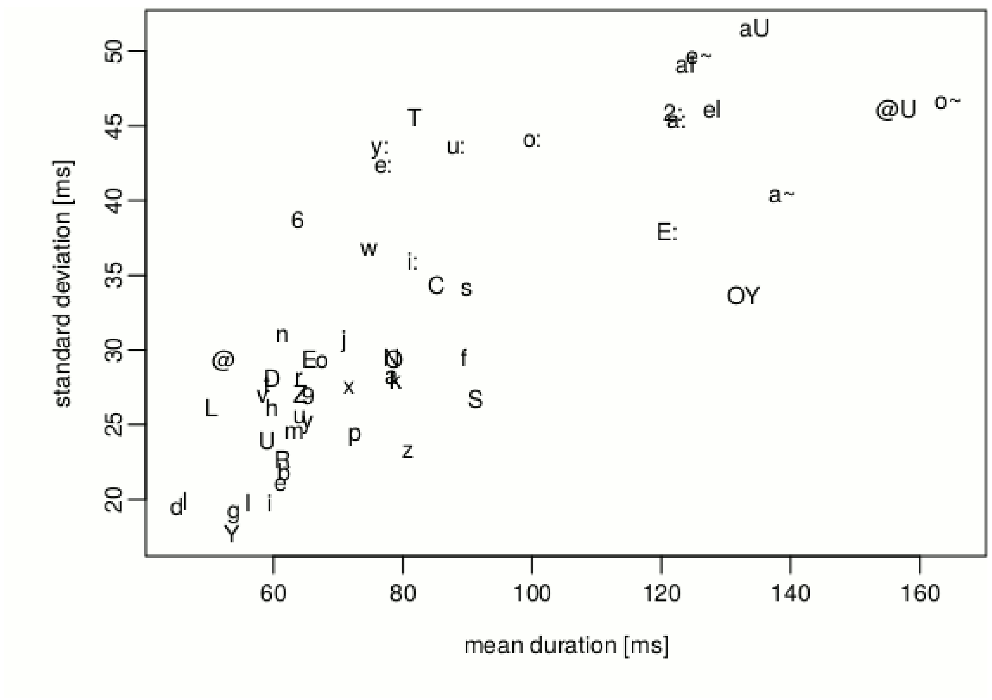
```
(define (voice_ims_de_antje_clunits)
  "(voice_ims_de_antje_clunits)
Define voice for de."
  ;; *always* required
  (voice_reset)

  ;; Select appropriate phone set
  (ims_de_antje::select_phoneset)

  ;; Select appropriate tokenization
  (ims_de_antje::select_tokenizer)

  ;; For part of speech tagging
  (ims_de_antje::select_tagger)

  (ims_de_antje::select_lexicon)
  ;; For clunits selection you probably don't want vowel reduction
  ;; the unit selection will do that
  (if (string-equal "americanenglish" (Param.get 'Language))
      (set! postlex_vowel_reduce_cart_tree nil))

  (ims_de_antje::select_phrasing)

  (ims_de_antje::select_intonation)

  (ims_de_antje::select_duration)

  (ims_de_antje::select_f0model)

  ;; Waveform synthesis model: clunits

  ;; Load in the clunits databases (or select it if its already loaded)
  (if (not ims_de_antje::clunits_prompting_stage)
      (begin
    (if (not ims_de_antje::clunits_loaded)
        (ims_de_antje::clunits_load)
        (clunits:select 'ims_de_antje))
    (set! clunits_selection_trees
          ims_de_antje::clunits_clunit_selection_trees)
    (Parameter.set 'Synth_Method 'Cluster)))

  ;; This is where you can modify power (and sampling rate) if desired
  (set! after_synth_hooks nil)
;  (set! after_synth_hooks
;      (list
;        (lambda (utt)
;          (utt.wave.rescale utt 2.1))))

  (set! current_voice_reset ims_de_antje::voice_reset)

  (set! current-voice 'ims_de_antje_clunits)
)
```

If you add functions with new names in the module-specific files, don't forget to also change the voice definition above accordingly.

Duration module

We've seen in the lecture that several context factors (such as syllable stress, position in syllable, neighboring phonemes or at least consonant classes, prosodic phrasing) affect segment durations. In order to train a very good duration model, the safest way would be to have contexts covering all possible combinations of these factors in your training data. However, the feature distributions of these follow Zipf's law, so we have a few very, very frequent ones, which we will almost certainly have in our training data automatically, there are also many, many very infrequent combinations. And while it is very unlikely for each individual infrequent combination to actually occur in text that you want to synthesize, their sheer mass makes it extremely likely that at least one of these many combinations does occur in your text. This data sparsity problem has been called a LNRE (large number of rare events) problem.

People have proposed machine learning techniques that deal relatively well with generalizing to unseen vectors (for instance so-called sums-of-products models) however, the default in Festival is to use regression trees (CARTs for short, abbreviated for classification and regression trees, in the following) for duration prediction. CARTs have been argued to be quite bad at generalizing to unseen vectors, however, the way they are used in your default voices does try to avoid the data sparsity problem by predicting duration z-scores instead of raw durations. Before I explain what that is, let's go back to a suggestion from Nick Campbell (also an important person in speech synthesis) who proposed in 1992 to predict syllable durations using what he called "elasticity" of phones. Here's a picture to explain what he meant by elasticity:



This picture shows phoneme classes by mean and standard deviation of their duration in a German 2-hour corpus of speech. You can see that some phoneme classes are quite similar in terms of mean duration, for instance, /m/ and /n/, or /OY/ and /aU/. This makes sense, they belong to the same classes. However, they have very different standard deviations: thus, while /m/ and /n/ are typically around 60-70 ms long, there is much less variation in the /m/ durations than in the /n/ durations. Similarly for /OY/ and /aU/. Campbell termed this "elasticity": /n/ phones are more "elastic" than /m/ phones. Thus, lengthening an /n/ by 30 ms is not the same as lengthening an /m/ by 30 ms.

Thus, Campbell proposes to measure lenghthening and shortening in terms of standard deviations – the idea is that lengthening an /m/ by, say, half its standard deviation should be the same as lengthening an /n/ by half its standard deviation. Campbell used this to calculated phone durations given the respective syllable's duration.

Festival by default calculates phone durations in a similar way: instead of predicting raw segment durations, which would be very phone-specific, we predict by how many standard deviations a phone should deviate from its mean duration. This simply amounts to predicting what is called a z-score instead of predicting the raw durations. Z-scores are a very common concept in statistics: Assume you have a vector X, with a mean μ and standard deviation σ, then you can apply the z-transformation to get a new vector Z (this is also called standardization).

$$Z = \frac{X - \mu}{\sigma}$$

For predicting phone durations, we calculate the z-scores for each phoneme type separately instead of once for all data, as above. So for a specific instance of a phone with duration p_i, if μ_p is the mean duration of all phones of the same type p, and σ_p ist the standard deviation of all durations of phones of this type, we calculate its z-score z_i

$$z_i = \frac{p_i - \mu_p}{\sigma_p}$$

Assuming that z-scores of all phonemes behave the same way in every context, we now need to train only one CART to predict them. Once the z-score is predicted, the raw duration can be calculated by multiplying the z-score with the standard deviation and adding the mean. Learning z-scores instead of raw durations alleviates the LNRE problem at least a bit: now we need not cover every possible context for every phoneme; instead the z-scores generalize what has been learned from a specific context and a specific phoneme to new phoneme types in the same context. To describe the idea in maybe more intuitive terms: instead of predicting raw durations, we predict a score that expresses lengthening or shortening: predicted scores which are negative indicate shortening (shorter than the mean duration of that phone), positive scores indicate lengthening.

In Festival, we use z-scores for duration prediction by default. Check your file `festvox/ims_LANGUAGE_SPEAKER_duration.scm`.

```
    (set! duration_cart_tree ims_LANGUAGE_SPEAKER::zdur_tree)
    (set! duration_ph_info ims_LANGUAGE_SPEAKER::phone_durs)
    (Parameter.set 'Duration_Method 'Tree_ZScores)
    (Parameter.set 'Duration_Stretch 1.1)
```

The first two lines state the CART that should be used for predicting z-scores, and the list from which phoneme-specific means and standard deviations are to be read so Festival can calculate raw durations given the z-score.

The third line tells Festival that it is to use the decision tree, and that this tree predicts z-scores rather than raw durations. The fourth line is a very crude way of slowing down the speech: every raw duration is finally multiplied by a factor of 1.1. You are welcome to get rid of that line, it's a bit absurd after we've just stated that lengthening by a factor should take a phoneme's "elasticity" into account...

The CART itself and the list of means and standard deviations can be found in your file `festvox/ims_LANGUAGE_SPEAKER_durdata.scm`. If you have an English voice, the tree and the list of means and standard deviations have been trained and extracted on some English data; if you have a German voice, the tree is extremely simple and standard deviations and means are faked by generating a list containing 0 for the means and 0.1 for the the standard deviations for all phones.

This is however in the default implementation not important at all since the predicted durations are not taken into account when selecting units…

You can check the predicted z-scores for an utterance by looking into the Segment relation. All segments (i.e. all phones) should have a feature `dur_factor`, and this contains the predicted z-score.

If you want to get Festival to consider the duration in selecting units, go to your file `festvox/build_clunits.scm` and change the parameter `dur_pen_weight` among the other unit selection parameters.

If you want to train a duration CART using your own data as training data right now, this is explained in the next section.

Training durations

If we want to train a model to predict duration z-scores, as described above, we need a huge table that contains one line for each phone in our training (or test) data, with these z-scores in the first column, and then all kinds of features that we want to consider for learning good durations in the other columns. Columns should be separated by a blank.

So the first thing to do is to specify the list of features you want to consider. A suggestion for a reasonable set of features (assuming feature names from the radio phoneset for English) can be found at the official URL that explains how to train such models in Festival, check it for inspiration:

http://www.festvox.org/bsv/x1902.html

Note that the very first feature listed there, `segment_duration`, is the feature we want to predict. So independent of which other features you want to consider, the first once should be `segment_duration` in any case. Also, since we want to normalize by phoneme-specific means and standard deviations, you should make sure that you have at least the name of the phone in your list (this is the case in the list above). Once you have the list, getting the huge table in the format required by Festival is pretty easy, since we already have converted all our data to Festival utterance structures, and now we can use Festival tools to extract all desired features for them and write them to a file.

Assuming you have a file with a list of features that you want to consider in the file `dur.featlist` in your voice directory, one feature per line, `segment_duration` in the first line, like the list at the above URL. Then you can use the `dumpfeats` script to extract the relevant features and write the output to such a table. Check how to use it:

```
$ESTDIR/../festival/examples/dumpfeats --help
```

So it wants to know for which relation you want to extract the features – in our case, the `Segment` relation. Output is the name of the file to which it should write the table, let's assume `dur.feats`. Since you will probably also extract phoneset features, we will have to specify Scheme files that define and select the correct phoneset. Since we have them already (`festvox/ims_LANGUAGE_SPEAKER_phones.scm` for defining it, and `festvox/ims_LANGUAGE_SPEAKER_clunits.scm` for selecting it), we can just use the `-eval` option twice and specify both files. Finally, the last argument should be the utterances from which to extract the features, so festival/utts/*.utt in our case.

If dumping the features was successful, you should now have a huge table in `dur.feats`, with the segment durations in the first column, and the name of the phones somewhere.

The next step is to replace the individual phone durations by their z-scores. This can be done using any statistics software; theoretically it could also be solved using Festival. However, that would require some Scheme programming, so I would suggest to do this outside Festival. Below I have some commands that would do it in `R`, but you're welcome to use other methods. If you want to use `R`, here's how to go about it: First, calculate the means and standard deviations:

```
# start R in the terminal
R
# read in table, save in a variable called data
data<-read.table("dur.feats",sep=" ")
# remove last column
# (which results from trailing space in feats file)
data[,ncol(data)]<-NULL
# split durations in groups based on their name
# assuming durations in first column,
# names in second column
groupedphones<-split(data$V1,data$V2)
# calculate means
means<-sapply(groupedphones,mean)
# calculate standard deviations
sdevs<-sapply(groupedphones,sd)
# display both
cbind(means,sdevs)
```

This is not urgent now and could also be done later, but you can already now copy the means and standard deviations and paste them into your file `festvox/ims_LANGUAGE_SPEAKER_durdata.scm`, since we'll need it there later. For converting to z-scores, it would be sufficient to have the values in `R`. Anyway, when you insert this in the file above, that should be at the point where the variable `ims_LANGUAGE_SPEAKER::phone_durs` is defined, and should look like this:

```
(set! ims_de_antje::phone_durs
'(( _   0.10519275 0.04229252)
( @  0.09891535 0.07404972)
( 2:  0.12076936 0.04957520)
( 2:6 0.16400001 0.06348226)
```

```
   ….
   ( Y6  0.07750002 0.02886762)
   ( z   0.09465795 0.03277169)))
```

Next we go on in `R` and calculate z-scores:

```
# first define a function for calculating
# individual z-scores using above values
getdurzscore<-function(val,name){
return ( (val-means[[name]])/sdevs[[name]])
}
# apply this function to the two first columns
data$V1<-mapply(getdurzscore,data$V1,data$V2)
# write new table
write.table(data,file="dur.zsc.feats", quote=F, row.names=F, col.names=F)
```

This should generate a table with z-scores for all our data in `dur.zsc.feats`. If we want to train on these data, the only thing missing is another desc-File. Sounds familiar? Remember that for clustering, we had to have the `festival/clunits/all.desc` file with all features and all possible values. This is because clustering was done with the same tool as the one we will use now, `wagon`. And `wagon` requires the desc-File.

Conveniently, Festival provides a tool to generate a desc-File given a data file, or at least an approximation to the desc-File… Here it is:

```
$ESTDIR/bin/make_wagon_desc <TRAINING DATA> <FEATLIST> <DESC-FILE>
```

So if you give it our `dur.zsc.feats` file as training data, your `dur.featlist` as list of features, and the name of your intended desc-File, say, `dur.desc`, you'll get a first approximation of your final file. There is only some manual work left to do: unfortunately, the tool does not know which values are continuous values, and which are categorical. So if you look into the file, you'll see that it's listed all z-scores it has encountered in your data as possible durations. That is of course not a good idea, so you need to replace this long, long list by the word `float`. If you have any other continuous features, these should also be changed to `float`.

Once you have the desc-File, you can train a simple tree by

```
$ESTDIR/bin/wagon -data dur.zsc.feats -desc dur.desc \
        -output dur.tree
```

If you want a more sophisticated one, you can first split the data in training and test data like this:

```
# write every tenth line to file for testing
cat dur.zsc.feats | \
perl -ne 'if ($i==9) {print; $i=0;} else { $i++ }' > dur.zsc.feats.test
```

```
# write everything but the tenth lines to file for training
cat dur.zsc.feats | \
perl -ne 'if ($i != 9) {print; $i++;} else { $i=0 }' > dur.zsc.feats.train
```

and then do step-wise training for the tree as described at the above URL:

```
$ESTDIR/bin/wagon -data dur.zsc.feats.train -desc dur.desc \
        -test dur.zsc.feats.test -stop 10 -stepwise \
        -output dur.10.tree
```

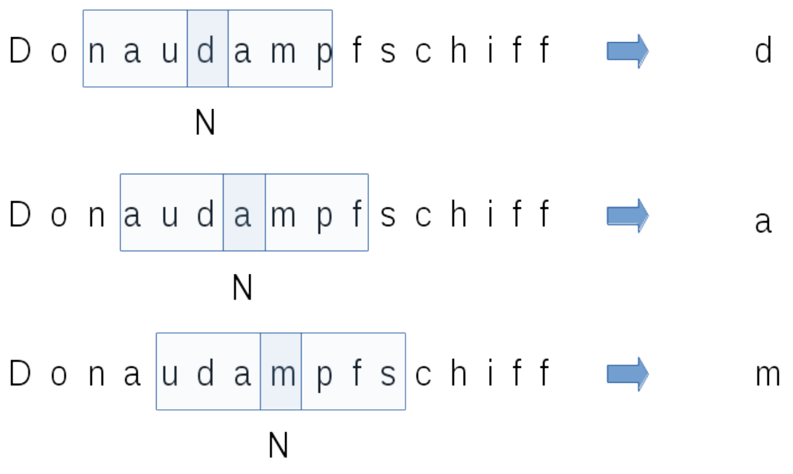
In order to use it for your voice, copy the tree from `dur.10.tree`, or from `dur.tree` above, and insert it in your `festvox/ims_LANGUAGE_SPEAKER_durdata.scm` file, replacing the tree in `ims_LANGUAGE_SPEAKER::zdur_tree`.

Letter-to-sound rules

Some information about LTS rules can be found at the festvox homepage, and the following section is based on the information there.

http://festvox.org/bsv/x1471.html

The basic idea is to predict phonemes as sketched here.


D o n a u d a m p f s c h i f f
N
d
D o n a u d a m p f s c h i f f
N
a
D o n a u d a m p f s c h i f f
N
m

We predict the pronunciation letter by letter, always looking at a window of 3 letters to the left and 3 letters to the right of the current letter, plus possibly the POS tag of the current word.

To this end we train a CART that predicts the phoneme given the window and POS context. As training data we use a lexicon which contains correct pronunciations and POS tags. This procedure requires that we already know which letter belongs to which phoneme in our lexicon. This is not always unambiguous, since there is no 1-to-1 correspondence between letters and phonemes:

```
w e e k → w i: k
t a x → t E k s
```

So before training, we "artificially" create a 1-1 correspondence in our training data by introducing epsilon place holders in the phoneme string, or by contracting several phonemes together to form a "pseudo-phoneme". (here: k-s in example 2):

```
w e e k → w i: Epsilon k
t a x → t E k-s
```

Thus in the lexicon we will for each word have as many phonemes as we have letters, and the correspondence is unambiguous. This is called an **alignment**.

Festival provides tools to create an alignment, however we first need a lexicon in the right format. You can use your lexicon to this end, but for training LTS rules, we want to get rid of syllable and stress information. We need one entry per line. Each entry is contained in brackets, and lists first the orthography in quotes, then the POS tag, and then a list of phonemes in the phone set you chose. The lexicon should thus look like this (it's a German example, so different phones and POS tags, but the format is the same):

```
("a" ADD ( a:))
("aachen" NOM ( a: x @ n))
("aal" NOM ( a: l))
("aale" NOM ( a: l @))
("aale" NOM ( a: l @))
("aalen" VRB ( a: l @ n))
("aalen" VRB ( a: l @ n))
("aalend" QPV ( a: l @ n t))
("aales" NOM ( a: l @ s))
("aalest" VRB ( a: l @ s t))
("aalet" VRB ( a: l @ t))
("aals" NOM ( a: l s))
("aalst" NAM ( a: l s t))
```

Once we have the lexicon ready, we can generate alignments in a semi-automatic way. First you have to specify which letters could possibly mapped to which phonemes. This is kept in a variable called allowables, and can be defined in a file called `allowables.scm`. An example is provided at the above URL, but without any mappings except for mapping to epsilon, so here's what allowables.scm could look like for German:

```
(require 'lts_build)
(set! allowables
      '((a a: a aU aI _epsilon_)
        (ä E: E OY _epsilon_)
        (b b p _epsilon_)
        (c k x C ts _epsilon_)
        (d d t _epsilon_)
        (e e: E E: e @ aI OY _epsilon_)
        (f f _epsilon_)
```

```
         (g g k C _epsilon_)
      ...
         (x k k-s _epsilon_)
         (y y: Y i: y _epsilon_)        (z ts _epsilon_)
         (# #)))
```

Note that I've omitted lines. Each entry in the list of allowables thus consists of first the letter in question, followed by one or several phonemes that we might want to map it to. `_epsilon_` is the place holder for getting 1-1 alignments in cases where several letters have to be mapped to one phoneme. Pseudophonemes (as in the case of the letter x above) are obtained by joining two phonemes by a dash. Note that it is recommended to always allow mapping to `_epsilon_`.

Try to be consistent when specifying the allowables. For instance in case of sequences of letters that can be mapped to diphthongs (ai mapped to eI as in raise, or oa mapped to oU as in boat), it makes sense to have a principled decision whether the first or the second letter should map to epsilon.

The easiest way to get a reasonable set of allowables is to do it iteratively. First, specify all obvious mappings, then try to align the lexicon as explained below and see which entries cause problems, and complete the mappings accordingly.

The allowables also have to contain the symbol # for word boundaries, which has to be mapped to itself. :

```
(# #)
```

We can then try to generate alignments in the following way. Change into the directory where you created your training lexicon. I'll assume it is called `lex.for.lts`.

In case of German, make sure that your Terminal is set to ISO-8859-1-encoding, and set the `LC_ALL` environment variable to `de_DE`. Create the allowables.scm into this directory. For testing the allowables, start Festival in the following way, with `allowables.scm` as an argument:

```
$ESTDIR/../festival/bin/festival allowables.scm
```

Then in the interactive mode, say

```
(cummulate-pairs "lex.for.lts")
```

As output, you'll see all (!) successfully aligned entries enumerated, plus all nun-successful alignments listed as "failed". Stop the output after a moment using <ctrl>-c and inspect the failed alignments.

```
failed: (# a b s p e n s t i g #) (# a p S p E n s t I C #)
```

Output like this indicates that the letter sequence "abspenstig" could not be mapped to the phoneme sequence a p S p E n s t I C with the current allowables. Find out which mappings were missing, and complete them in the allowables, then re-run.

Sometimes alignments fail because of errors in the lexicon, as in the following case for German:

```
failed: (# a c h t t a u s e n d s t e #) (# a x t t aU z @ n ts t @ #)
```

In this case, the mapping failed because the lexicon wrongly specifies a ts phoneme where there should only be a sequence of t and s. Such cases can be safely ignored; if no alignment is produced, they will not be in the training data, which is what we want, after all we don't want to learn errors from the training data. Similar problems can arise in case of foreign words including non-native phonemes (for instance German lexica usually contain French loan words that include French nasal vowels). Again, it can be argued that these should not serve as training data because they are not typical for German. Finally, alignments may fail in cases where sequences of letters with ambiguous mappings occur next to each other. In these cases, we will also have to accept that we loose some training data.

In any case, iterate as long as it takes to get an acceptable coverage for "representative" words. Once you are satisfied, let the command run to completion, and save the result using

```
(save-table "lex-")
```

This will generate a file called `lex-pl-tablesp.scm`. Close Festival, inspect the file, and try to interpret its content.

Then re-start Festival, using both files as arguments:

```
$ESTDIR/../festival/bin/festival allowables.scm lex-pl-tablesp.scm
```

Then align the data:

```
(aligndata "lex.for.lts" "lex.align")
```

This generates a file `lex.align`. Please inspect this one, too. It contains the letter sequence on the left, then POS tags, and then exactly as many phonemes, pseudo-phonemes, and epsilons as there were letters, which thus provides a unambiguous alignment. This is the data basis for training the CART.

In the case of LTS prediction, the features that we use as predictors are the context letters and the POS tag. So the first step is to create a features file which for every letter in the lexicon contains the relevant features. To obtain it, start Festival using the same arguments as above:

```
$ESTDIR/../festival/bin/festival allowables.scm lex-pl-tablesp.scm
```

Then run the following command to generate a file `lex.feats` for your file `lex.align`

```
(build-feat-file "lex.align" "lex.feats")
```

You may have suspected that for training the CART, we will again run `wagon`, and will need a `.desc` file, and you were correct. We can generate a first approximation using `make_wagon_desc` as above for the duration module.

So we need a list of names of the features first, and this should be:

```
phone
p1
p2
p3
p4
letter
n1
n2
n3
n4
pos
```

Write these feature names into a file called `lts.feat.names`, then use it together with the `lex.feats` from above to generate a `.desc` file (note that in case of German, your `LC_ALL` should still be set to `de_DE`!).

```
$ESTDIR/bin/make_wagon_desc lex.feats lts.feat.names lex.desc
```

Please inspect `lex.desc` – does it look correct? It should contain all features along with the values observed in the training data.

Next we separate the data into a training and a test part, so we can evaluate the performance afterwards, for instance by running the two following perl one-liners, which continuously count from 0 to 9 while reading in the original data and then either output every 10th line (first variant, for creating test data) or output all other lines (second variant, for creating training data).

```
cat lex.align |  \
perl -ne 'if ($i == 9 ) {print; $i=0;} else { $i++}' > lex.test.align
```

```
cat lex.align |  \
perl -ne 'if ($i != 9 ) {print; $i++;} else { $i=0}' > lex.train.align
```

Since we split the lexicon, we now have to repeat the feature extraction part, for each file separately.

```
$ESTDIR/../festival/bin/festival allowables.scm lex-pl-tablesp.scm
(build-feat-file "lex.test.align" "lex.test.feats")
(build-feat-file "lex.train.align" "lex.train.feats")
```

The `.desc` file can be used as is, because it already contains all potential values. The actual training is done by a shell script which iterates over all letters; analogously for testing. Two example scripts for training and testing (German) LTS rules can be found at

```
/mount/studenten/synthesis/2017/bin/test.sh
/mount/studenten/synthesis/2017/bin/train.sh
```

They are adapted from example code at the URL above. Have a look at the code, copy it and adapt it to your needs. In each iteration, a combination of `cat` and `awk` is used to extract lines from the training data which are relevant for the respective letter (i.e. all lines where that letter is in column 6). For training, the data are split further using the same perl command as above, in order to keep a small amount of test data for optimizing some parameters when building the tree (this is what the option `-stepwise` does in the following `wagon` command). In addition, the script extracts those lines from the test data which match the current letter, so the CART can be tested using `wagon_test` afterwards.

Run the script like this

```
sh train.sh
```

The trees will then end up in files called `*.tree`. (Maybe create a directory for them and adapt the script accordingly so your working directory does not get too crowded. You can have a look at the trees using `less`. There are lists of phonemes at the leaves together with their relative frequency in the training data; the last entry in each list gives the most probable phoneme in that context.

The script for testing `test.sh` is run like this

```
sh test.sh
```

and tests all trees and outputs their phoneme error rate.

However, in addition to the phoneme error rate, we are interested in the word error rate. Again, Festival provides code for calculating it. Start festival

```
$ESTDIR/../festival/bin/festival allowables.scm lex-pl-tablesp.scm
```

and combine all trees to one big tree which can be applied for all letters

```
(merge_models 'lex_lts_rules "lex_lts_rules.scm" allowables)
```

This creates the file `lex_lts_rules.scm`, which holds the complete tree.

Afterwards, terminate Festival and restart it using this file and the alignment probabilities as arguments:

```
$ESTDIR/../festival/bin/festival lex-pl-tablesp.scm lex_lts_rules.scm
(require 'lts_build)
(lts_testset "lex.align" lex_lts_rules)
```

This will output phoneme and word error rates for the test data, for instance:

```
phones 1356627 correct 1300409 (95.86)
words 134110 correct 89803 (66.96)
tree model has 26784 nodes
```

For trying out the rules, we need to make sure the lts module has been loaded, and we need to load the new file `lex_lts_rules.scm` which contains our new rules. Then you can use the Festival function `lts_predict` to apply those rules:

```
(require 'lts)
(load "lex_lts_rules.scm")
(lts_predict "testwort" lex_lts_rules)
```

Beware, this works only with lower case spelling.

For using the rules in a voice, we need a function which converts unknown words to lower case, applies the rules, then introduces stress and syllable boundaries and then gives back the result.

Then we need to ensure that the rules are used for our voice. Have a look at your lexicon definition. There should be a comment like this

```
;; If you have lts rules (trained or otherwise)
```

which indicates some commented code that would do the trick – uncomment the following lines, they downcase unknown words, apply the rules, then insert stress and syllable boundaries.

However that commented code has a bug in its very first line, it is missing parentheses around the `boundp` command. The line in its corrected version should look like this, with an opening bracket before `boundp`, and two closing ones after the `ims_LANGUAGE_lts_rules`:

```
(if (not (boundp 'ims_de_lts_rules))
```

The remaining lines of that code require that a module called `ims_LANGUAGE_lts_rules` is present:

Put the `lex_lts_rules.scm` from above in the `festvox` directory, name it `ims_LANGUAGE_lts_rules.scm. In the first line in this file, make sure that the variable that holds the tree is called ims_LANGUAGE_lts_rules, i.e., the first line should read:`

```
(set! ims_LANGUAGE_lts_rules '(
```

Also put the command

```
(provide 'ims_LANGUAGE_lts_rules)
```

at its end.

Then everything should hopefully work. If not, please ask me for help.