Voicebuilding Project Seminar Report
WS 2017-18

Shrestha Ghosh (2567717)[2], Aikaterini Azoidou (2558632)[1], and Eleni Metheniti (2566390)[1]

[1]*Department of Computational Linguistics, Saarland University*
[2]*Department of Computer Science, Saarland University*

March 30, 2018

1 Introduction

Speech synthesis or text-to-speech (TTS) is the process of generating spoken waveform using written text as input; the goals of a TTS system are to generate output that is intelligible, sounds as natural as possible and is, ideally, able to read any sentence in its given language. [1]

Our goal during the project seminar was to build new voices for a pre-existing TTS system, `MaryTTS`, by using two different methods of speech synthesis: Unit Selection and Statistical speech synthesis system based on a hidden semi-Markov model (HSMM). In order to build a new voice, we had to first create a new speech corpus, and then use the according tools to create units for the Unit Selection synthesis or to synthesize the HSMM voice. In the Methodology section, we will describe our process of creating the voices from start to finish, and in the Discussion section we will present the way our voices perform and possible shortcomings and improvements.

2 Methodology

2.1 Pre-processing for corpus making

A speech corpus is a a large collection of audio recordings of spoken language, accompanied by the transcriptions of the utterances and the time in which they occurred. Since one of our tasks was to create our own speech corpus by recording one of the researchers, we first had to create the text that the speaker would read. For our speech corpus, it was more beneficial to have text aligned with the utterances as closely as possible, because it would allow us to create phonetic units in the later stages more easily, and having the speaker improvise speech and then transcribe it would be time-consuming and difficult.

One parameter that needs to be accounted for, when creating prompts, it to cover all possible phonemes in the language at least once, and for Unit Selection the diphones[1] as well. Standard American English has around 44 phonemes (therefore, 44^2 diphones) and, since our voice will be used in a non-domain specific framework (MaryTTS), we had to ensure that our speech corpus covered all of them in order to be successful in speech production.

Since our time was limited, instead of creating the prompts ourselves, we consulted a Speech Synthesis database, Festvox [2]. Festbox is a project by the Language Technologies Institute at Carnegie Mellon University which provides tools for building a voice using Unit Selection, such as databases of prompts. For our purposes, we used the CMU_ARCTIC databases which are phonetically balanced, US English single speaker databases designed for unit selection speech synthesis research. [3]

[1]Diphones are units of sound that start from the middle of one phoneme and end at the middle of the next phoneme, thus including possible co-articulation effects and allowing for easier concatenation in the place of the minimal spectral change.

The CMU_ARCTIC prompt list contains 1132 sentences derived from literary works available on the public domain by Project Gutenberg.

For the speaker to read the prompts easily during the recording session, we had to create a slide-show which would present every sentence prompt separately. We used Groovy in a Gradle wrapper to generate a list of prompts from the `cmuarctic.data` file, in the format of a Markdown file. The purpose of processing the raw prompts file was to segment each line to the prompt id and the prompt text. Then, we used the package `pandoc` [4] to convert the Markdown file to an HTML5 presentation, which presents the prompt text clearly and legibly, with the prompt id as a subtitle. We also encoded a beeping sound for each slide transition, which will come to use in the audio processing stage of the speech corpus making.

2.2 Recording and Creating the speech corpus

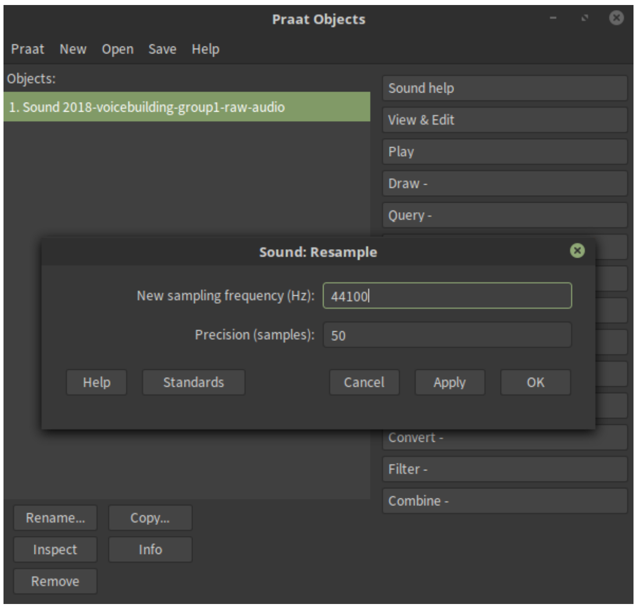
Once our prompts slide-show was ready to be used, we could start the recording process. One of the researchers (Eleni) was chosen for the task; ideally we would have selected a native English speaker, but there was none in our team. The recordings were done in a professional-grade recording studio of the Computational Linguistics department, and the studio operator was there to assist us. We used two microphones, a headset and a standing microphone, which recorded the prompts on different channels. We recorded a third channel for the beeping sounds that signaled the slide transition, and will be used in the processing stage. We also used a MIDI controller (Musical Instrument Digital Interface) that was connected to the sound system in order to mark mistakes in the recording process (slip-ups, noises etc.) and to let the speaker know that she would have to repeat the prompt. The MIDI output was recorded in a separate file. The recording went smoothly and in our time slot we managed to record 330 prompts.

After our recording session, the studio operator gave us the following files:

- `2018-voicebuilding-group1-raw-audio.flac`
- `Voicebuilding_Lena_HDMI.wav`
- `Voicebuilding_Lena_headset.wav`
- `Voicebuilding_Lena_microphone.wav`
- `Voicebuilding_Lena_Midi.mid`

The audio file in .flac format included three channels: the headset recording, the standing microphone recording and the annotations. A problem that we noticed was that the sampling rate of the .wav files was set to 44100 Hz, but the .flac file's sampling rate was set to 48000 Hz. We managed to re-sample the file by using Praat, as seen in Figure 1, and save it in order to work with it. We also used SoX [5] to down-sample the `Voicebuilding_Lena_HDMI.wav` file which contains the channel with the beeping noises to 16000 Hz, which reduced its size and its quality. The purpose was to work with a file that was smaller in size and whose quality was unimportant, as it only contained the beeping noises. Afterwards, we could combine the downsampled .wav file with the other .wav files to create a .flac file in Praat, as seen in Figure 2. The reason we are using the LongSound format of opening the files in Praat is in order to avoid memory issues in our computers, which were a problem for all team members. The LongSound format option allows the user to open a sound file without loading the entire file in the memory, but offers limited options compared to normally opening the file, e.g. the user can view the sound file's spectrogram(s) and annotate the file but they cannot resample a LongSound file.

Our next task was to create an annotations file, which would contain the starting and ending boundaries for each utterance, and the prompt text of the utterance. While selecting the .wav file with the beeping sounds, we created a new annotations file which contained the boundaries for every beeping sound, where every beeping sound was annotated as 'sounding' and everything in-between as 'silent'. The annotations file can either be saved as a TextGrid file or as a chronological TextGrid file; the former saves the annotations as numbered objects with their time boundaries and annotations,



New sampling frequency (Hz): 44100
Precision (samples): 50

Figure 1: Resampling an audio file in Praat. The user needs to then save the resampled file.

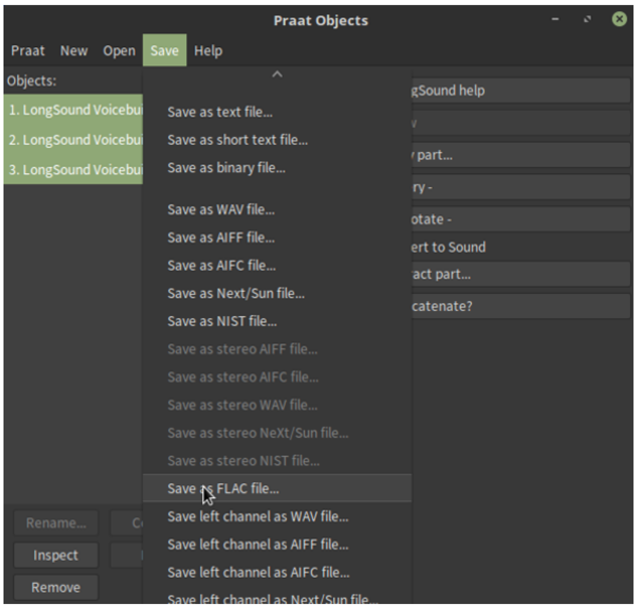

Praat Objects
Praat New Open Save Help
Objects:
1. LongSound Voicebu
2. LongSound Voicebu
3. LongSound Voicebu
Save as text file...
Save as short text file...
Save as binary file...
Save as WAV file...
Save as AIFF file...
Save as AIFC file...
Save as Next/Sun file...
Save as NIST file...
Save as stereo AIFF file...
Save as stereo AIFC file...
Save as stereo WAV file...
Save as stereo NeXt/Sun file...
Save as stereo NIST file...
Save as FLAC file...
Save left channel as WAV file...
Save left channel as AIFF file...
Save left channel as AIFC file...
Save left channel as Next/Sun file...
Rename...
Inspect
Remove

Figure 2: Saving multiple files as a .flac file with multiple channels in Praat.

while the latter stores only the boundaries and the annotations. We opted for a chronological TextGrid file, because we could all generate and work on the annotations file, and at a later point we could then merge our files without the problems that numbering would cause. Praat's annotations files can be viewed and edited with any text editor, so with a simple text editor we replaced the 'silent' annotations with the string 'speech' and the 'sounding' with an empty string.

After this process was completed, we could open the .flac file which contained the recorded speech (making sure that it was the one with the 44100 Hz frequency) along with our previously generated and edited annotations file. Viewing the files together is seen in Figure 3. However, these annotations also include boundaries for utterances that should be not taken into account (e.g. when the speaker was asked to repeat a prompt and there are two recordings of the same prompt, one incorrect and one correct). In this case, it is sufficient to remove the boundaries of an utterance we do not want to include, and leave it without any annotation, as seen in Figure 4. Also, the annotation boundaries include silences at the start and the end of every utterance and in some cases unwanted noise (e.g. the speaker repeated an utterance without a beeping sound in-between) as seen in Figure 5, so we had to adjust the boundaries manually to include only speech and no silence or other noises as seen in Figure 6. In order to split the work load more efficiently, each one of the group members worked on one third of the annotations file, with the intention to combine them.

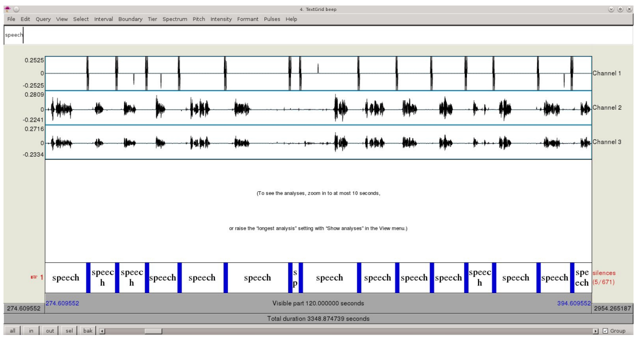


Figure 3: The audio file with three channels in Praat, and the modified annotations file.

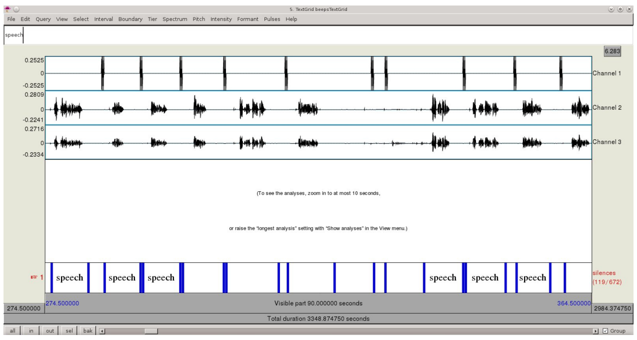


Figure 4: An example of how unwanted utterances can be omitted, by leaving the annotations empty.

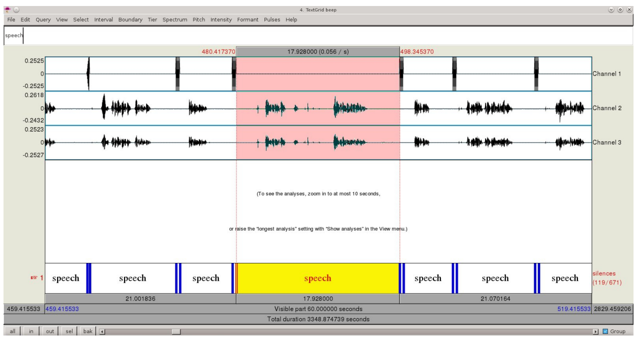


Figure 5: An utterance that includes unnecessary silences and audio.

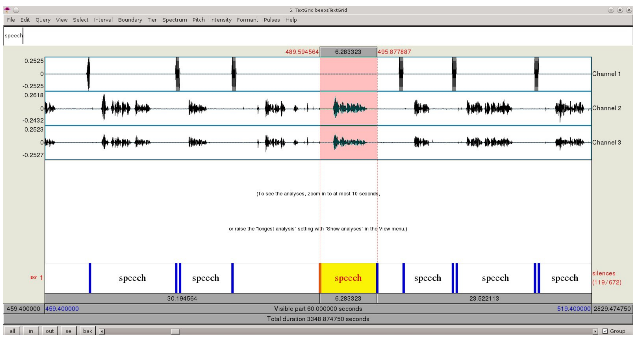


Figure 6: The same utterance, trimmed to include only the correct prompt reading.

After setting the correct boundaries, we were able to merge our three .TextGrid files in one file; we encountered a problem with a missing boundary, but we manually added it and the merging process was successful. We finally saved the .TextGrid file as a .Table file (a tab-separated values file which contains the starting and ending boundaries for every annotated segment.

2.3 Finalizing the speech corpus

Our corpus, at this point, consists of one long sound file and a file with annotated time boundaries for every prompt. It is necessary to align every utterance with the corresponding prompt text, in order to finalize our corpus and use it in the later stages of the voice generation. We wrote two scripts, using Python, and ran them via a Gradle wrapper, in order to align each prompt with the time boundaries of the corresponding utterance (`process.py`), and to split the sound file in one file per (correct) prompt utterance (`extract.py`). The full code and details on how to run it can be found in our project's Bitbucket repository.

process.py

This script takes two arguments, the annotations file and the prompts list (in our case, 'Annotations.Table' and 'cmuarctic.data.txt') and creates a metadata file which contains the prompt id, the prompt text, the starting and ending boundaries of the utterance. For example, for the first utterance, the first line of the file reads as seen in Table 1.

id	Prompt	Start Time	End Time
arctic_a0001	Author of the danger trail, Philip Steels, etc.	4.533922	7.968610

Table 1: Example data entry in the metadata .Table file

extract.py

This script takes two arguments, a metadata file with time boundaries and text and an audio file (in our case, the metadata file generated by `process.py` and the 3-channel audio flac file). For every line in the metadata file, the script creates a new text file with each prompt text, and the time boundaries are used to trim the audio file into a new audio file in .wav format. In order to achieve this in Python, we called an external command using SoX. The script outputs 330 .txt and .wav files, each identified by the prompt id, into `build/text` and `build/wav` folders respectively.

2.4 Data Packaging for Voicebuilding

2.4.1 Forced Alignment

Once our speech corpus is ready (i.e. every prompt has a respective text file and audio file), it can be used in speech synthesis. The first voice we created was based on Unit Selection, which requires a state-transition matrix with varying sizes of phonemic segments (from words to phones). In order to achieve this, we need to create forced alignments, i.e. generating time alignments between audio and text down to a phonemic level. Since we already have a corpus with aligned sentence-level speech, we can use a tool to perform forced alignments, such as the Montreal Forced Aligner. [6] The Montreal Forced Aligner uses the Kaldi Automatic Speech Recognition toolkit [7], which supports modeling of context-dependent phones of arbitrary context lengths.

We used the Montreal Forced Aligner as a Gradle plug-in with MaryTTS in the backend, in order to predict the text pronunciation. It first creates the MaryXML files by using MaryTTS; these files contain information for every utterance in various levels (from full word to phoneme) in an XML schema, such as the Grapheme-to-Phoneme alignment method used (e.g. 'lexicon'), the accents (based on the ToBI system), and the phonemes for all syllables in the utterance. The following is an excerpt of a MaryXML file for the word "hello".

```
<phrase>
<t accent="H*" g2p_method="rules" ph="' h E - l o:" pos="NE">
hello
<syllable accent="H*" ph="h E" stress="1">
<ph d="85" end="0.0852885" p="h"/>
<ph d="210" end="0.2957095" f0="(0,131) (50,143) (100,929)" p="E"/>
</syllable>
<syllable ph="l o:">
<ph d="119" end="0.41494948" f0="(0,149)" p="l"/>
<ph d="305" end="0.7201745" f0="(50,110) (100,114)" p="o:"/>
</syllable>
</t>
<boundary breakindex="5" duration="400" tone="L-%"/>
</phrase>
```

These information can help the Forced Aligner to train a monophone model, a triphone model and the speaker-based triphone model, based on the corpus and the dictionary.

2.4.2 Gradle project for final data packaging

We built a Gradle project to automate the audio processing. The task execution dependencies are set up in the following manner:

```
:packageData - top level task which creates a Zip with
 three sub-directories containing the wav, text and lab files.
+-:convertTextGridToXLab - creates the label mapping in the
   build/lab sub-directory.
  +-:runForcedAlignment - trains the monophone and triphones.
    +-:unpackMFA
      +-:prepareForcedAlignment
        +-:generateDictionaryFromMaryXml - create dict.txt in
           the build folder.
          +-:convertMaryXmlToMfaLab
            +-:convertTextToMaryXml - create MaryXml files for
               all utterances under build/maryxml sub-directory.
              +-:extract - runs the python script to generate
                individual text and audio files for utterances.
                +-:process - runs the python script to generate
                  the metadata file.
```

The Gradle task `packageData`, creates a .zip file in the `build/distributions` sub-directory of the files required by the MaryTTS voicebuilder which we shall discuss in the next section.

2.5 Day 5: Building the Unit Selection voice

The final part of the project was to build a new voice using the MaryTTS voice import tools, so we started by creating a new Gradle project called **voice-2018-voicebuilding-group1**. We used the MaryTTS voicebuilding plugin `de.dfki.mary.voicebuilding-legacy` version `5.3.2`. The Voice Import Tools tutorial gives a clear overview of how to build a new voice [8]. We ran two Gradle tasks in the following order:

```
./gradlew legacyInit
./gradlew build
```

The `legacyInit` task unpacks the packaged data created previously. The `build` runs the voice import tools. We then run the command `./gradlew assemble --dry-run` to view the tasks in order of their execution. The component descriptor .xml file and the actual voice components in a .zip file can be found under the `build/distributions` sub-directory. The .zip file contains the following files required by MaryTTS voice installer:

```
+lib
|>-voices
| >-my_voice
|    |-halfphoneFeatures_ac.mry
|    |-halfphoneUnits.mry
|    |-joinCostFeatures.mry
|    |-timeline_basenames.mry
|    |-timeline_waveforms.mry
|>voice-2018-voicebuilding-group1.jar
```

We cloned the `MaryTTS v5.2` in a separate directory and built the project using the command `./gradlew build`. Then, we copied the new voice component descriptor .xml file and .zip file from the `build/distributions` folder of the `voice-2018-voicebuilding-group1` project to the `download` sub-directory.

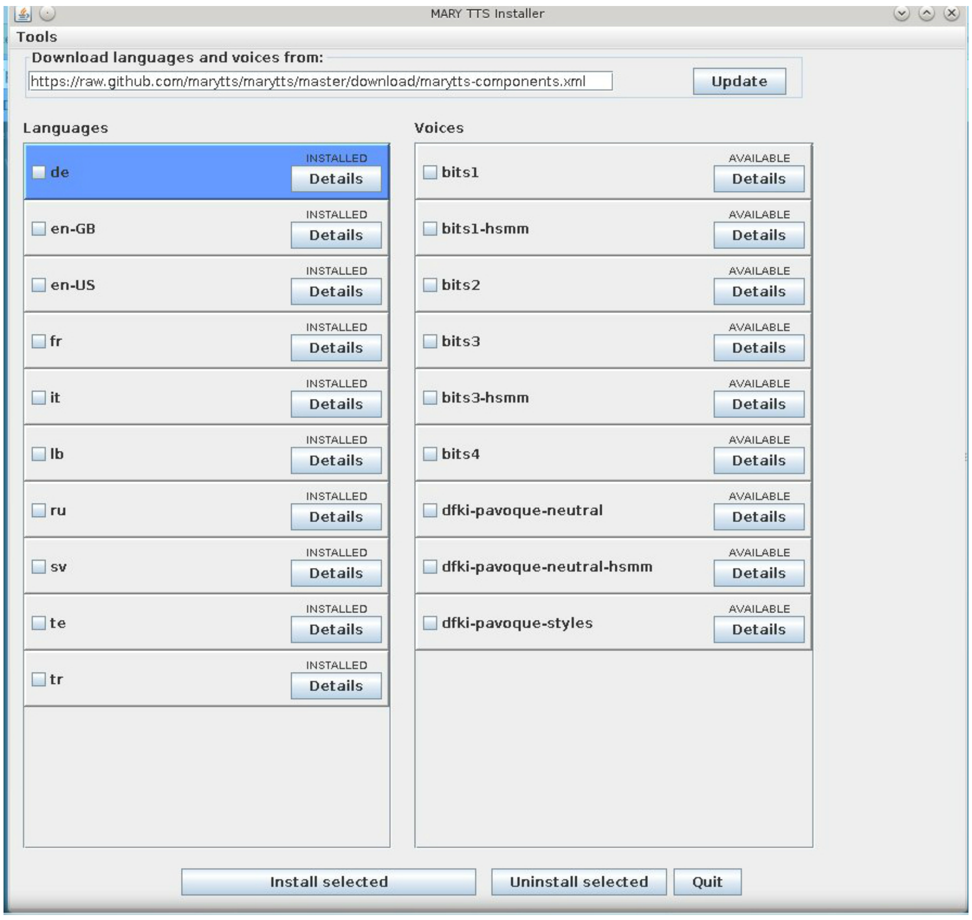
The next part was to install the voice in MaryTTS. We ran the MaryTTS component installer from the cloned MaryTTS project. The component installer is executed by the `runInstallerGui` task using `gradlew` which launches a GUI (as seen in Figure 7). New voices show up on the right panel, which we can then select and install. Once the voice is installed, we exit the GUI and ran a MaryTTS server using the command, `./gradlew run`. By navigating with a browser to the page `localhost:59125` (MaryTTS runs on port number 59125 in the local machine), we can finally select the newly installed voice from the *Voice* option. The web-browser interface allows for text input, which can be turned into speech using the selected voice and heard or downloaded, and the user can select some voice effects and view the generated MaryXML document, as seen in Figure 8.

3 HMM-Based Speech Synthesis

HMM-Based Speech Synthesis is a statistical method of modeling and generating the speech parameters of a speech unit (e.g. F0 frequency, duration, intonation) by using Hidden Markov Models based on maximum likelihood estimation. It is considered to perform better than Unit Selection, because it calculates the speech parameters for every sound, instead of looking for the unit with the optimal parameters in the corpus (which may not exist or not be as close to the target unit), and also because it preserves the Fo frequency throughout the generated utterance. Therefore, HMM voices sound more natural and with less glitches, but still fall victim to unnaturalness.

We try to generate HMM voices using the Hidden Markov Model toolkit provided by HTK [9]. The voicebuilding process is setup in a docker container. We follow the steps outlined in the slides of previous edition of the seminar [10]. The container is created from the dockerfile provided in the slides.

We faced problems while running the voiceimport tools inside the docker. The voiceimport script invokes a function for GUI and throws an exception if is not able to display. This problem occurred


MARY TTS Installer

Tools

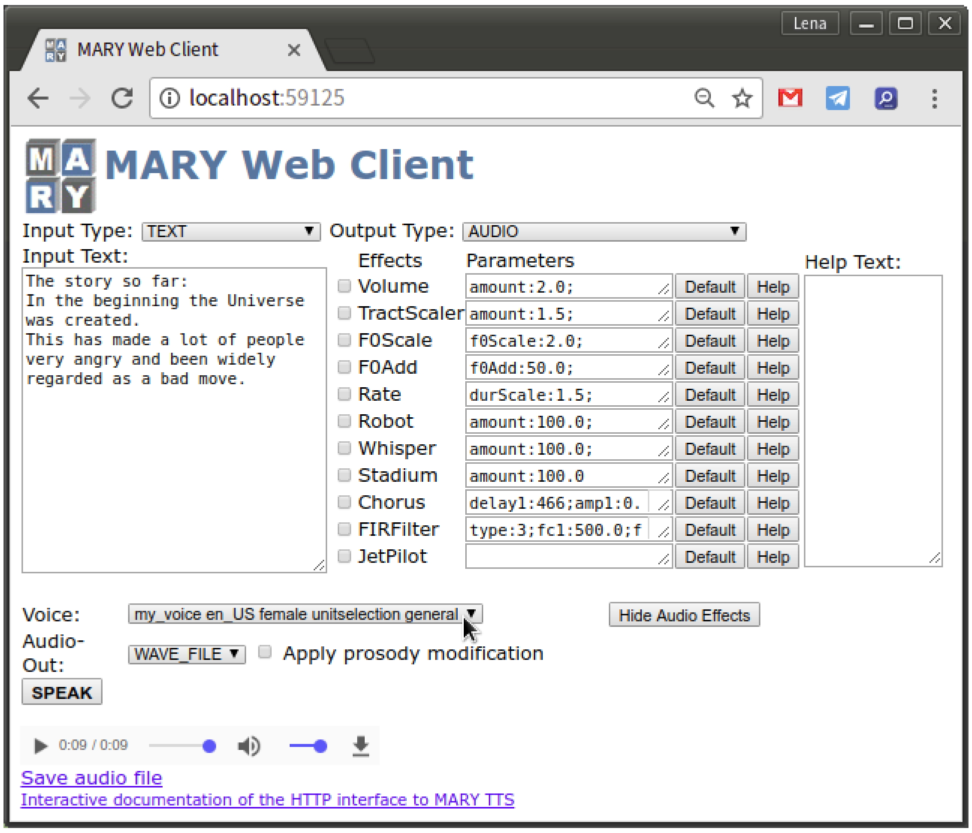
Download languages and voices from:

https://raw.github.com/marytts/marytts/master/download/marytts-components.xml

Update

Languages		Voices	
de	INSTALLED	bits1	AVAILABLE
en-GB	INSTALLED	bits1-hsmm	AVAILABLE
en-US	INSTALLED	bits2	AVAILABLE
fr	INSTALLED	bits3	AVAILABLE
it	INSTALLED	bits3-hsmm	AVAILABLE
lb	INSTALLED	bits4	AVAILABLE
ru	INSTALLED	dfki-pavoque-neutral	AVAILABLE
sv	INSTALLED	dfki-pavoque-neutral-hsmm	AVAILABLE
te	INSTALLED	dfki-pavoque-styles	AVAILABLE
tr	INSTALLED		

Figure 7: MaryTTS new voice component installer GUI


MARY Web Client

localhost:59125

MARY Web Client

Input Type: TEXT

Output Type: AUDIO

Input Text:

```
The story so far:
In the beginning the Universe
was created.
This has made a lot of people
very angry and been widely
regarded as a bad move.
```

Effects	Parameters
Volume	amount:2.0;
TractScaler	amount:1.5;
F0Scale	f0Scale:2.0;
F0Add	f0Add:50.0;
Rate	durScale:1.5;
Robot	amount:100.0;
Whisper	amount:100.0;
Stadium	amount:100.0
Chorus	delay1:466;amp1:0.
FIRFilter	type:3;fc1:500.0;f
JetPilot	

Help Text:

Voice: my_voice en_US female unitselection general

Audio-Out: WAVE_FILE

Apply prosody modification

Figure 8: The MaryTTS web interface.

because we created the docker container in the wrong folder. Ideally, the container should be built in the `build` directory of the voicebuilding project, since this directory contains all the text and wav files. Once the container is up and running, we ran the voiceimport tools which executes the following components:

- HMMVoiceFeatureSelection - is run outside the docker. The voiceimport script (located in the `marytts-builder/bin`) is executed from the voicebuilding project. We then proceed to select the project location and the task. The features are stored in the location `build/mary/hmmFeatures.txt`.
- HMMVoiceDataPreparation - sets up the environment to create HMM voice and checks if the text and wav files are available in the correct paths.
- HMMVoiceConfigure - copies the path settings depending on the installation.
- HMMVoiceMakeData - runs HTS scripts to prepare data for training.
- HMMVoiceMakeVoice - runs the HMM training to create voice.
- HMMVoiceCompiler - is the component which prepares the HMM voice for MaryTTS so that it can be installed as a voice component.

The `HMMVoiceCompiler` component is run separately after HMM training is completed. Before running this component we would need to set up a buildscript inside the Maven project created during the HMM training. The builscript is provided in the slides [10] and needs to be copied to the project folder. We used the docker copy command to copy the file in the required Maven project directory (`build/mary/voice-my_voice` inside the voicebuilding project directory).

After running the `HMMVoiceCompiler` component we would need to build the voicebuilding project. Then we would execute the following command:

```
.\gradle run
```

This would automatically load the voice and sets up the MaryTTS server on port `59125` so that we can listen to the new voice by navigating to MartTTS client on `localhost:59125`, in the same way as we have described in Section 2.5.

4 Results and Discussion

4.1 Workload

In order to complete the process of building a voice from start to finish, we had to rely on the strengths of each team member. The steps up to the Gradle scripting were followed by all team members, as it would have been more arduous (or impossible) to distribute the files amongst us. When Gradle scripting was required, Shrestha was better-versed in coding and was able to create the Gradle wrappers. An earlier version of a Python script that Eleni created was scrapped, as it would not be ideal to use with Gradle. Aikaterini and Eleni provided support to Shrestha during the coding process. The task of maintaining the repository and completing the report was also split among the team. Overall, it was possible to create at least a voice based on Unit Selection synthesis in the course of one week. We were able to set up the HMM voice component, however, by the time everything was up and running, we had reached our report deadline and could only provide limited insights on the HMM voice compared to the Unit Selection synthesis.

4.2 Voice quality

In class, we had the chance to listen to both the Unit Selection voice and the HMM-based voice, however, as previously mentioned we were not able to replicate the process of building the HMM-based voice, in order to test it. From the few minutes of testing the HMM-based voice, we concluded that the quality given our speech corpus was sub-optimal; one intuition would be that the corpus was not ideal or too small, or that the parameters would have to be tuned more carefully. In this section, we can discuss about the Unit Selection voice more in-depth.

First of all, the voice is comprehensible and the fundamental frequency is preserved for short phrases. The duration and the intonation are captured mostly correctly, with the exception of sentence endings in some cases. Indeed, our corpus included sentences with embedded clauses, such as '*Philip stood undecided, his ears strained to catch the slightest sound.*' and '*Soaked in seawater they offset the heat rays.*'. However, some phonemic units are not represented correctly; for example, the contraction "I'm" is not heard as [ʌɪm], but as [ɪm]. This could be due to the absence of the diphone [ʌɪm] in our corpus, when there are more occurrences of [ɪm] whose parameters could fit the target diphone (e.g. *'him'*, *'immaculate'*, *'impossible'*).

An unexpected problem we noticed was that the voice would, in some cases, not pronounce numbers. This is unexpected, because MaryTTS expands numerical expressions into the according words while it performs linguistic analysis on the input text. This only occurred in specific cases; for example, in the sentence 'It is still owned by a branch of the same family (the Eltz family) that lived there in the 12th century, 33 generations ago.', the number '12th' is transcribed as ['twɛlfθ] and the number '33' as ['θr̩di 'θri] – the former is pronounced correctly, the latter not. This could again be blamed to our limited corpus. The CMU_ARCTIC prompt list is carefully curated in order to include all necessary phones for Standard English, but we only recorder 330 out of the 1132 prompts. Also, we cannot ensure perfect quality of the recordings, as our speaker was not an English native speaker and not a trained voice actor. Nevertheless, the Unit Selection voice was successful in most cases and was not riddled with glitches.

5 Conclusion

Over the course of a week, we were able to build a speech corpus and two new voices for MaryTTS. Through this project we learned how to record and pre-process data to be used for voicebuilding. We learned to integrate all tasks in a single Gradle project and reuse files in the successive builds. The different components required for building and publishing a voice were explained during the course of the seminar and were implemented incrementally in our project.

The voice quality for Unit Selection is acceptable for smaller utterances, but the glitches become apparent for new or more complex sentences or unseen phenomena, as discussed in Section 4. The HMM-based voice could outperform the Unit Selection one.

References

[1] P. Taylor, *Text-to-speech synthesis.* Cambridge university press, 2009.

[2] A. Black and K. Lenzo, "Festvox: Building synthetic voices," *Language Technologies Institute, Carnegie Mellon University, PA, USA, Tech. Rep*, 2002.

[3] J. Kominek and A. W. Black, "The cmu arctic speech databases," in *Fifth ISCA Workshop on Speech Synthesis*, 2004.

[4] J. MacFarlane, "Pandoc: a universal document converter," *URL: http://pandoc. org*, 2013.

[5] "Sox - sound exchange — homepage." [Online]. Available: http://sox.sourceforge.net/

[6] M. McAuliffe, M. Socolof, S. Mihuc, M. Wagner, and M. Sonderegger, "Montreal forced aligner: trainable text-speech alignment using kaldi," in *Proceedings of interspeech*, 2017.

[7] D. Povey, A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, N. Goel, M. Hannemann, P. Motlicek, Y. Qian, P. Schwarz *et al.*, "The kaldi speech recognition toolkit," in *IEEE 2011 workshop on automatic speech recognition and understanding*, no. EPFL-CONF-192584. IEEE Signal Processing Society, 2011.

[8] “Voice import tools tutorial : How to build a new voice with voice import tools.” [Online]. Available: https://github.com/marytts/marytts/wiki/VoiceImportToolsTutorial

[9] “Htk.” [Online]. Available: http://htk.eng.cam.ac.uk/

[10] “Hts voicebuilding, devops.” [Online]. Available: http://www.coli.uni-saarland.de/ steiner/teaching/2016/winter/voicebuilding/slides/index.html