General purpose methodology and tooling for Text-to-Speech support in voice services for under-resourced languages

Justyna Klęczar
Vrije Universiteit Amsterdam
De Boelelaan 1105
1081 HV Amsterdam
The Netherlands
Student No. 2602204
jkleczar@gmail.com

ABSTRACT

In Africa, mobile telephony has become widespread in recent years. This trend has induced growth in development of mobile applications and services. However, due to low literacy levels as well as a huge linguistic diversity across the continent, it proves challenging to create accessible applications for a wide range of African communities. Implementing voice-based services is one possible way to bypass the problem of illiteracy, but not linguistic diversity. In this paper the Text-to-Speech Slot and Filler system, originally developed as part of the Lwazi II project, is presented. The system is aimed at under-resourced languages and supports a limited dictionary of words. Thanks to its relative simplicity of use, it allows for rapid development and creating support for new languages by non-experts. The Text-to-Speech Slot and Filler system has the potential to help existing mobile services overcome the linguistic diversity in Africa. To show this, the system was used to create TTS support for a new language, Twi, spoken in Ghana. With the help of a native speaker, a number of suitable recordings was created in the domain of weather forecast and then used to build the system. To test the quality of the TTS conversion, several new sentences in the language of Twi were synthesised and evaluated by a group of native speakers in a feedback form. Most of the syntheses were deemed easy to understand and sounded mostly natural. In some cases, a word was mispronounced due to incorrect unit selection, indicating that this part of the system may need to be revised. Finally, the system was integrated into an existing voice-based service called InfoMétéo, extending it by the language of Twi. Based on this use case, a general purpose methodology and tooling of limited domain TTS support for under-resourced languages is provided.

Keywords

TTS, Text-to-Speech, Slot and Filler, unit selection, illiteracy, voice-based services, language, linguistic diversity, accessibility, Africa

1. INTRODUCTION

In Africa, mobile telephony has become widespread in recent years. For the majority of the African community, the mobile phone is first and primary means of communication [7]. According to the GSMA study conducted in 2016, it was found that over half a billion people across the continent have been subscribed to mobile services [6]. The forecast number of subscriptions is estimated to increase to 725 million by 2020.

This development opens up many opportunities to improve African economy and information sharing. However, due to the low literacy levels among the African community, many of the regular services provided by mobile phones turn out to be inaccessible. In Sub-Saharan Africa, around 182 million adults and 48 million youths (aged 15-24) are illiterate [19]. On top of this, there is a huge diversity of languages across the continent, summing up to over 1500 in total [4]. For these two reasons, illiteracy and linguistic diversity, it proves challenging to create useful and accessible applications for such a wide range of communities.

Implementing voice-based services is one possible way to bypass the problem of illiteracy. This approach has been undertaken by researchers working at The Network Institute at Vrije Universiteit in Amsterdam. The Radio Marché system they developed is a great example of how a voice interface can make the application accessible to local farmers [14]. Its main purpose was to provide market information to users through a radio broadcast service. The market information is input by the farmers, who can advertise their produce by making a phone call to the system. The information is stored on the Web, after which it is shared on local community radios in different languages.

One especially interesting feature of Radio Marché is the TTS (Text-to-Speech) system, which allows for automatic creation of audio files from text input. It was developed by using the Slot and Filler method [14], which is based on mapping plain text to audio files and representing them as attribute-value pairs. The system has only been deployed once, providing support for the languages of Bambara and Bomu. The great advantage of the Slot and Filler TTS system is its suitability for many different languages, including those that are under-resourced. New languages can be added to the system with the help of a native speaker. It is important to note that this system is designed for a limited number of words, making it suitable for applications that focus on a very specific domain (for example: market information). Radio Marché uses this TTS system to automatically generate the radio readings from the input market information, and by doing so does not rely on actual human broadcasters for the various languages.

There are many other examples of useful voice services in Africa that could benefit from the TTS technology. Reusing it in such services could potentially help with solving the challenge of linguistic diversity in building mobile applications for African communities. The Slot and Filler system, though often overlooked in the mainstream state of the art, is definitely of interest in the field of ICT4D due to its simplicity to set up and use by non-expert users. It does not require any background in the field of linguistics, nor expert programming knowledge. With detailed documentation, it can be handled by a student with basic programming and command line skills. Another advantage of the TTS Slot and Filler system is that it is computationally inexpensive, making it an attractive candidate for Text-to-Speech conversion in ICT4D.

This paper will provide a general purpose methodology and tooling of the Slot and Filler TTS method, including a detailed guide of how the system can be built and used. This will be done for the language of Twi, based on an example use case of meteorological readings.

2. RELATED WORK

2.1 Accessibility in African Context

Accessibility of mobile applications in Africa has a somewhat different meaning than in the rest of the world. W3C describes ”accessibility” as ”making websites and applications more accessible to people with disabilities when they are using mobile phones and other devices” [5]. This meaning of the term is based on a Western perspective. In Africa, accessibility is affected by a number of factors taken for granted in W3C's description, including access to electricity, technology and the Internet, and last, but not least, a certain standard of literacy.

If we look at the technology, it is apparent that Africa is behind on the Western world. The majority of Africans use outdated mobile phones, which is an important factor that needs to be taken into consideration when creating applications for the African communities. One of the obvious solutions to this problem is the use of SMS for a broad range of services, as it is supported by all mobile phones. However, SMS is problematic in the sense that it does not cover the desired range of users, because of its heavy dependence on literacy [14]. To avoid the problem of language, symbols could be used as a more direct and universal communication tool instead. However, here too problems arise, as symbols do not always convey the intended meaning. For example, an arrow pointing to the left in some communities may be interpreted as ”go back”, while in others it means ”go forward”, depending on the local convention. The next step in tackling the problem of an accessible and usable application, is the use of voice. Voice based services bypass the problems of illiteracy and interpretation of signs all together, and allow for a direct and clear way of communicating over the phone. They are also suitable for any type of mobile phone device and do not require an Internet connection, making them the perfect candidate for building user-friendly and accessible mobile applications in Africa.

2.2 Voice Based Services

Given the high illiteracy levels in Africa, an alternative interface is needed to ensure accessibility of applications. Voice based services are a very promising option, especially since they are already familiar to the people. They have been used as solutions in several projects carried out in Ghana and Mali. This section gives a brief overview of these projects.

2.2.1 Radio Marché

Radio Marché is a voice-based service targeted at the farmers community. It is used to broadcast various produces offered by farmers on the local radio [14]. The market offerings information is first made available on the Web and then shared by local community radios, thus making this data accessible to individuals without Internet access.

The Radio Marché project was executed in Mali and its aim was to improve an already existing Market Information System (MIS), which was mostly based on sharing information via SMS and storing it in an Excel file. For many, the use of SMS was not an accessible option for sharing information. Radio Marche's new system allowed for manual processing of the information from both voice and text messages, thus supporting many more users with low literacy levels. The information collected is used for creating the market communiqués in a web interface especially made for that purpose. The audio communiqués are generated automatically by a TTS (Text-to-Speech) system, which was created solely for under-resourced languages by using the Slot and Filler method (see Section 3). The languages used in this project were Bambara and Bomu.

2.2.2 Marketing for Agricultural Products

The voice-based system for marketing of agricultural products was developed in Ghana by Dittoh et al. [13]. It was designed to be accessed through a phone call from a local number, which would be answered by an Interactive Voice Response (IVR). Support for integration of new languages was provided. In addition to the mobile voice interface, a Web interface was also created.

The work flow of the system is as follows: in order to add his products for sale to the system, a farmer makes a phone call to a local phone number, which is answered by the Voice Server. Once the farmer has provided the input, he is asked for confirmation. As soon as it is given, the system transfers the obtained data to a database, which is in turn used by a web interface providing all listings to potential buyers with Internet access.

Radio Marché and Marketing for Agricultural Products are projects that make great use of voice technologies to improve information sharing in different regions in Africa, while remaining accessible to illiterate users. Market information in Radio Marché is stored in a structured way (a spreadsheet) and covers a very specific domain, making this service suitable for the TTS Slot and Filler technology. The majority of other voice-based systems share the same unique properties: structured information and restricted vocabulary. Thus in principle, the TTS system used in Radio Marché could be relevant to them as well. In order to make the TTS Slot and Filler technology accessible to new services, a methodology and required tooling will be described in this paper.

2.3 Text-to-Speech

In a nutshell, TTS synthesis is ”the technology that converts an input text into a speech signal” [8]. This section focuses on existing solutions in the field of Text-to-Speech systems and their suitability in the context of this project. It then goes on further to explore the state of the art in TTS, describing and evaluating the relevance of various approaches.

2.3.1 *Existing TTS Systems*

This section covers two very different Text-to-Speech systems: Mary TTS and Slot and Filler TTS. Both of them are evaluated on the basis of their suitability as TTS support for under-resourced languages in voice-based services. The factors taken into account include not only the possibility of creating such support, but also ease of use and requirement for computational power of these systems.

MARY TTS

One of the most well-known Text-to-Speech systems used by researchers is the German MARY (Modular Architecture for Research on speech sYnthesis) [17]. It is an open-source, multilingual Text-to-Speech Synthesis platform written in Java and supports ten languages: German, British and American English, French, Italian, Luxembourgish, Russian, Swedish, Telugu and Turkish. The platform features several toolkits that can be used to add support for new languages as well as to build unit selection and HMM-based synthesis voices.

The toolkit for adding support for a new language contains an extensive number of modules, including:

- **Preprocessing and text normalisation** - these modules involve word tokenisation, processing abbreviations and numbers.
- **Natural language processing** - this module is responsible for linguistic analysis and annotation. The text is chunked, grapheme to phoneme[1] conversion is performed, intonation rules are defined.
- **Calculation of acoustic parameters** - this module translates the linguistic symbols into an acoustic parameter file, specifying parameters such as sound duration for each phoneme[2].
- **Speech synthesis** - this module transforms the previously created acoustic parameter file into an audio file.

By following a number of steps described in the MARY documentation [2], it seems feasible for a person with technical background to create support for a new language. Unfortunately however, the system does not seem to be well suited for under-resourced languages. In one of the first steps, the toolkit requires large amounts of textual data coming from Wikipedia for processing. This requirement makes the system impossible to use for the minority African languages, as they simply do not exist on Wikipedia.

As mentioned earlier, MARY also allows for creation of voices by using either the unit selection or the HMM-based approach. The toolkit itself has been described in detail by Pammi et al. [15] and expanded to make it more accessible to regular users, by creating a Graphical User Interface (GUI) for most of the common tasks. Unfortunately again, new voices can only be created for the languages already supported by MARY.

[1]Grapheme to phoneme - letter to sound.
[2]Phoneme - the smallest contrastive unit in the sound system of a language.

Despite the fact that MARY is a very robust and successful TTS synthesis system, it can only be applied to major languages that are present on the Web. It is therefore not a suitable technology that could be used in voice-based services such aimed at African communities as Radio Marché. Additionally, since it is a system supporting an entire language and was developed with the level of Western technology in mind, it may turn out to be too expensive in terms of computational power to use in the African context.

Slot and Filler TTS

The Slot and Filler TTS system is very different to MARY, both in terms of its popularity and intended use. It origins from the Lwazi II project carried out by Calteaux et al., which focused on creating full TTS systems for South African languages [11]. The system was later modified by Daniel van Niekerk (who also worked on Lwazi II) during the VOICES (VOice-based Community-cEntric mobile Services) project [9] to support the creation of Slot and Filler systems. VOICES is funded by the European Union and, as one of its objectives, it aims to "improve voice-based access to content and mobile ICT services through the development of a free and open source toolbox for local developers" [1].

The TTS Slot and Filler system was built with under-resourced languages in mind. Its purpose is to provide TTS synthesis for a limited dictionary of words in a specific domain, which can then be used in a variety of other applications, such as voice-based services. The system is "considerably simpler to develop than full-fledged subword-based systems" [9], allowing for rapid development and creating support for new languages. It is written in Python and Bash. It uses in-house developed TTS tools to perform the steps of text processing and speech segmentation. The whole system is based on the unit selection approach, described in more detail in the following section: *State of the Art in TTS*.

Although the TTS Slot and Filler system has not been in use for several years (the last update of the source code dates back to June 2013), it seems like the most suitable tool available for the development of TTS support for under-resourced languages in voice-based services.

2.3.2 *State of the Art in TTS*

State of the art in Text-to-Speech (TTS) synthesis has been well outlined by Barnard et al. in VOICES *Report on state of the art and development methodology* [8]. In a traditional TTS synthesis system, there are two main building blocks:

- **Linguistic processing block** - this process is responsible for extracting information from the text. The information is then represented in a symbolic sequence of phonemes, including linguistic information of the speech, such as its melody, rhythm and intensity.
- **Acoustic processing block** - this process is responsible for generating a speech signal based on the symbolic sequence produced by the Linguistic processing block.

In order to perform both of these processes, different approaches can be used. Some of these approaches are outlined in the sections below.

Linguistic Processing

Linguistic processing focuses on analysing the text. In everyday life, regular writing contains many ambiguities, such as abbreviations, as well as various symbols that need to be interpreted, such as punctuation marks. A TTS synthesis system must deal with such cases with the use of grammatical analysis, which ensures that each utterance obtains the correct pronunciation and intonation. This analysis process typically includes a number of steps, as described by Schnabel et al. [16].

In the preprocessing step, text normalisation is performed. In this stage various symbols, including numerical expressions, abbreviations and special characters, are converted into their orthographic form. Next, pronunciation of the text is determined. This is done by converting text into a sequence of phonemes with use of different technologies, including grapheme-to-phoneme rules, morphological analysis and pronunciation lexicons. The morphological analysis is responsible for carrying out the segmentation of words and defining stress positioning, making sure that the spoken words are accentuated correctly.

In the last stage of linguistic processing, contextual modification of pronunciation is performed. This process makes sure that differences of intonation are taken into account in various cases, for example a word spoken on its own versus a word inside a sentence.

Acoustic Processing

In acoustic processing, different approaches can be used for speech synthesis. The most efficient one mentioned by Barnard et al. is the HMM-Based Synthesis, based on Hidden Markov Chains. At the moment, this method is considered "the standard approach" [8]. In his book "Text-to-speech Synthesis", Paul Taylor describes the HMM approach in detail. Originally, HMM models were developed for speech recognition, however since then, they have also been used for many other tasks in the field of speech and language. HMM is a statistical, machine learning method which, in the context of speech synthesis, allows the system to "attempt to *learn* the general properties of the data" [18], rather than memorise the data itself as in the concatenative approach. There are two advantages to using HMM - firstly, in large applications, much less memory is required to store data parameters of the model than to store the entire data. Secondly, the model is flexible in the sense that it can be modified in various ways, for example converting the original voice to a different one.

Another popular approach used in speech synthesis is unit selection. It derives from a second generation system used in the 1990s, called the concatenative diphone approach [18]. The reason why the concatenative diphone system was extended, was that it was too simplistic and did not account for variations in diphones[3] correctly. The unit selection method uses a richer variety of speech, therefore it is able to capture more variation in phones and make the final speech synthesis sound more natural. The idea behind this approach is that each linguistic type consists of a number of units, which vary between each other. Then, "during synthesis, an algorithm selects one unit from the possible choices, in an attempt to find the best overall sequence of units which matches the specification" [18]. A "unit" can be a phone, syllable or a word, depending on computational resources, the intended size of the dictionary and the desired final effect of the synthesis. In the case of the TTS Slot and Filler system, a unit represents a single word, since its main purpose is to provide TTS support for voice-based services with a very specific domain.

The unit selection approach is the appropriate choice when developing a TTS synthesis system for a very specific domain. It is much simpler to implement than the HMM approach and therefore allows for rapid development, including adding support for other languages. Its second advantage is that, due to a relatively small database of words, it is computationally inexpensive to run.

[3]Diphone - an adjacent pair of phones. It is usually used to refer to a recording of the transition between two phones.

2.3.3 VOICES Recommendation

While the HMM model is advantageous in terms of memory, it is only robust in the case creating TTS support for an entire language, rather than a small subset of it. When working with a limited dictionary focusing on a very specific domain, this advantage is lost, while the complexity of the approach remains. On the other hand, the unit selection approach is simple to implement and therefore allows for rapid development and providing support for new languages with a limited dictionary quickly. The recommendation in the VOICES research is to use the unit selection speech synthesis when developing a TTS system for under-resourced languages. Following the recommendation, the TTS Slot and Filler system was developed by Daniel van Niekerk, which was incorporated into the Radio Marché project.

Having reviewed the existing TTS systems and state of the art approaches, it is clear that the TTS Slot and Filler method is the most suitable one in the context of this project. The rest of this paper focuses on a detailed description of this system, how it can be used and incorporated into an existing voice-based service. The system is then evaluated with respect to the quality of speech synthesis produced, as well as its ease of use. Based on the evaluation, recommendations are made on how the system can be improved in the future.

3. SYSTEM DESCRIPTION

This section describes the architecture and work flow of the Slot and Filler TTS System.

3.1 Slot and Filler Structure

The Slot and Filler is a well-known structure in the field of Artificial Intelligence. It was used as the primary design framework in the development of the TTS Slot and Filler system described in this paper. In this structure, a slot is an attribute value pair in its simplest form. It is often referred to as a frame, which is supposed to hold values. A filler is a value that a slot can take. It can be any value: a string, numeric, a pointer and other.

In the case of the TTS Slot and Filer system, the filler of the attribute slot is the string "example". Then, the filler of the value slot would be the corresponding audio file "example.wav". Figure 1 illustrates the usage of Slot and Filler structure in the TTS system.

3.2 System Design

The Slot and Filler TTS system designed and built by Daniel van Niekerk was implemented by using the unit selection approach with each unit representing a word, which

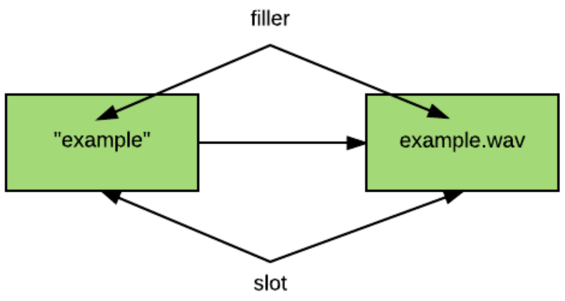


Figure 1: Slot and Filler Structure

was described in Section 2.3.2. This approach relies on creating a set of recordings for a given domain, which can then be used to select and join a subset of these recordings, in order to create a new sentence. The Slot and Filler TTS System was built by customizing already existing open-source TTS software, including normalize-audio (tool for adjusting the volume of audio to a standard level), Praat (used for speech analysis and synthesis), HTK (toolkit for building and manipulating Hidden Markov Models) and Edinburgh Speech Tools (a collection of functions for manipulating objects used in speech processing). Additionally, text-processing modules were developed for the languages of Bambara and Bomu, including basic phoneme sets, letter to sound rules and text normalisation routines [10].

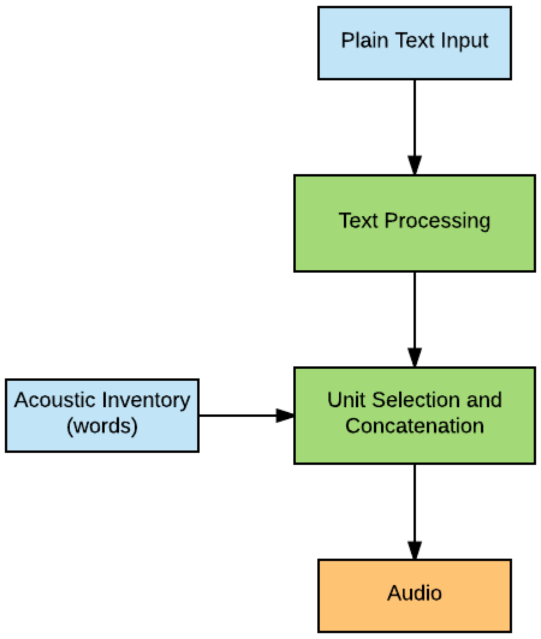


Figure 2: TTS Synthesis Process

The work flow of building the system for a new language is as follows: first, a *voice definition* is created, based on the pronunciation resources manually defined for the language of choice. These pronunciation resources include phoneme sets and letter to sound rules. Next, the pre-recorded audio files are setup for alignment. In this step, the recordings are processed by performing downsampling (reducing the sampling rate of a signal in order to reduce its size) and energy normalisation on them. Finally, the phones of the language are aligned. Text grid files and utterances are created, followed by feature extraction for the database. Once everything is in the database, the TTS has been created and is ready for testing.

Having the system setup, text to speech synthesis can be performed. As shown in Figure 2 [9], the application takes plain text as input, for example a sentence. The text is processed and divided into several units (words). Then, as each unit is selected, a corresponding audio file is chosen from the acoustic inventory. The audio files are then concatenated together, thus forming the original sentence in audio form. The output of the system is an audio file.

4. METHODOLOGY AND TOOLING

In this section, a general methodology and tooling for the Slot and Filler TTS system is provided. For a detailed user guide describing installation of dependencies, building the system for the first time and adding support for new languages, please refer to Appendix A.

4.1 System Requirements

The system runs only on case-sensitive operating systems, thus Linux is the preferred OS. MacOS may also be suitable, but only if the case-sensitive version was installed (the current default is case-insensitive). In this project, the system was built and tested on 64-bit Ubuntu 16.04.2 LTS.

4.2 Dependencies

The following dependencies are used in the TTS Slot and Filler system for building a new voice:

- Python 2.7 or higher
- Numpy
- Scipy
- SoX
- Normalize-audio
- Praat
- HTK 3.4.1
- Edinburgh Speech Tools 2.4

For details on the installation of these dependencies, please refer to Appendix A.2.

It should be noted, that these dependencies are only required for voice building, rather than the running of the system. The run time system (i.e. the system running the text-to-speech synthesis once the voice file has been created) relies only on the top three dependencies in the above list.

4.3 TTS System Build and Language Support

The TTS Slot and Filler system build requires basic knowledge of the command line in Linux based systems and basic debugging skills. A step-by-step guide for building the system for the first time is provided in Appendix A.3. The language used in the default build is Bambara, spoken in Mali.

To provide new language support, a native speaker of that language and a specific use case in a narrow domain are required. New language support can be created in tree steps:

- **Step 1.** A list of phones of the new language must be obtained from an external source or from a native speaker. This list is then split into vowels and consonants. It should be understood how each phone is supposed to be articulated, including the manner as well as place of articulation. This information is represented as a set of values, which is then used to create the voice definition of the language during the build of the new system.
- **Step 2.** Given a use case in a narrow domain, an example script should be created, containing a set of sentences covering the use case as much as possible. Where content will vary, e.g. slots for changing days of the week, numbers or weather conditions, all variations should also be included. These should be included in sentences of the script for recording in order to make sure that the prosodic features of utterances are preserved. Then, the sentences as well as "filler" words (such as days of the week or numbers) are to be recorded by one person. Doing the recordings in one session is very beneficial for the overall quality of the final speech output.
- **Step 3.** Once the recordings have been made, an accompanying text file must be created, including attribute value pairs containing the names of the audio files mapping onto the text representation of the content in that audio file.

One might ask why letter-to-sound and phone information is needed to build a Slot and Filler system based on word units. The answer is evident from the way the system has been developed - as mentioned in Section 2.3.1 under *Slot and Filler TTS*, the system originated from the Lwazi II project, where a full, diphone-based TTS system was created. The letter-to-sound and phone information were used for automatic phone alignment, which allowed for building the system from recordings automatically. The TTS Slot and Filler system works almost in the same way, with only one difference - the units are represented by words rather than by diphones.

Once the three steps described above have been completed, the system can be re-built and used for the new language. For a more detailed guide of new language support, please refer to Appendix A.4, where support for the language of Twi, spoken in Ghana, is implemented.

5. CASE STUDY: INFOMÉTÉO

In the Vrije Universiteit course called ICT4D, students learn how to develop applications for under-resourced countries in Africa. In many cases, the applications that are built by the students are voice-based services. In combination with the TTS Slot and Filler System, the ICT4D course presented a good opportunity for collaboration.

At the beginning of the ICT4D course, the Slot and Filler TTS System was presented to the students, offering it as a potential tool for their projects. A group of students working on a case study of meteorological rainfall readings was interested in the system and decided to build a voice-based application called InfoMétéo [12]. InfoMétéo is intended as a tool for farmers, allowing them to access information about the weather for the upcoming days. This service is important and potentially life-changing for local farmers in Africa, because rainfall has become more erratic and less predictable over the past few years. More than 90% of the active population in West Africa is involved in agricultural activities, making agriculture the most important source of income in the region [3].

The way the InfoMétéo system works, is as follows: the farmer calls a local number and gets a voice menu, where he can pick a language of choice, followed by a choice of region. He will then receive the weather forecast, the precipitation level, wind speed and wind direction for this region, in the chosen language. The system supports 13 administrative regions in Burkina Faso and the languages of English and French. For a more detailed view of the service control of the InfoMétéo system, please refer to Figure 3.

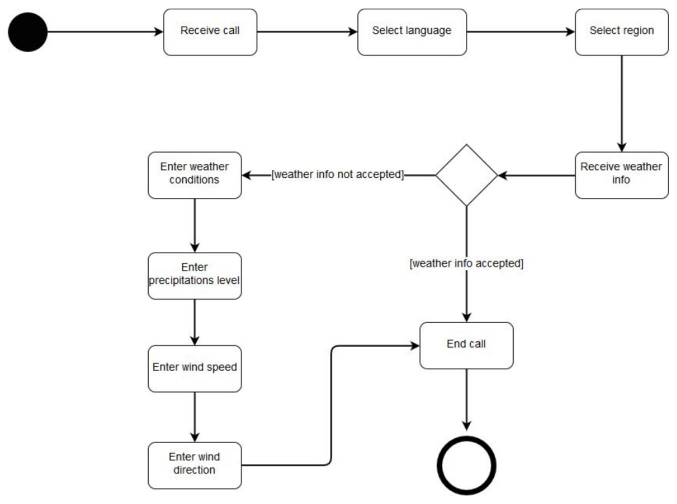


Figure 3: Service Control Flow of InfoMétéo

To extend the InfoMétéo system and prove that it is possible to provide local language support, the TTS Slot and Filler System was used to incorporate a new language, Twi, spoken in Ghana. The Twi language was chosen by the availability of a native speaker, who agreed to create suitable recordings in the domain of weather forecast. A meeting was arranged between the InfoMétéo team a mediator who was in contact with the Twi speaker in Ghana. A script with words and possible sentences was agreed upon and a total of 53 recordings were made, consisting of 16 sentences and 37 filler words (see Appendix B). They included sentences describing weather conditions and voice commands allowing a choice of language or location, as well as extra recordings for variable content such as days of the week, numbers and words describing weather. All recordings were processed by audio software Audacity in order to remove background noise.

Once the recordings were ready for use, the TTS Slot and Filler System was built on the server of the InfoMétéo team. It was integrated into the InfoMétéo application, taking text as an input and creating an audio .wav file as output. According to the InfoMétéo team, once the system was in place on the server, it was straightforward to use and integrate.

However, the installation itself proved to be difficult without the help of the system expert. That is why it was decided to create a detailed documentation of the setup, which has been included in Appendix A.

6. RESULTS

In order to measure the quality of speech synthesis of the TTS Slot and Filler system, eight new sentences in the language of Twi were synthesised and six native Twi speakers were asked for feedback on understandability and naturalness of the sentences. The eight sentences to be created were supplied in text form by the same native speaker, who created the 53 recordings for the build of the system. They can be viewed in Appendix C.

The native speakers were asked to fill out a feedback form asking three questions about each of the speech syntheses:

1. The first question asked to indicate how easy the speech synthesis is to understand by the native speaker, by rating them from 1 to 5. The numbers represent the following measures: 1 - not possible to understand, 2 - difficult to understand, 3 - acceptable, 4 - easy to understand and 5 - very easy to understand.

2. The second question asked to indicate how natural the speech synthesis sounds to the native speaker, by rating them from 1 to 5. The numbers represent the following measures: 1 - very unnatural, 2 - mostly unnatural, 3 - partly natural and partly unnatural, 4 - mostly natural and 5 - human-like natural.

3. The third question asked to write an explanation for the choices made in the first two questions, including giving examples of words or parts of the sentence that are not easy to understand or do not sound unnatural.

After the feedback forms have been filled out, they were collected and analysed. The average scores for the first two questions were calculated for each recording (see Table 1). In general, the response was positive.

Table 1: Average score of created syntheses

	How easy to understand?	**How natural?**
Synthesis 1	4.00	4.00
Synthesis 2	3.33	3.00
Synthesis 3	4	4.00
Synthesis 4	3.5	4.00
Synthesis 5	3.83	4.00
Synthesis 6	3.67	4.00
Synthesis 7	3.17	3.00
Synthesis 8	3.17	3.00

The scores for Question 1 (How easy to understand is the synthesis?) were all above 3 ("acceptable") with most of them being 3.5 and above, indicating that they were "easy to understand". Syntheses 1, 5 and 6 received the highest average scores, while syntheses 2, 7 and 8 received the lowest average scores.

The responses to Question 2 (How natural is the synthesis?) turned out to be very similar to the ones for Question 1, which was to be expected. An interesting observation is that all respondents agreed on the exact same score for each synthesis, making all averages either 3 (partly unnatural, partly natural) or 4 (mostly natural). Again, syntheses 2, 7 and 8 received the lowest scores.

Extra feedback was given for the syntheses with lower scores. For Synthesis 2, it is clear that the word "Nnɛ" at the end of the sentence is causing the problem. This is due to the fact, that a separate recording for this word was made with a very strong emphasis on ɛ. The system used this unit for the sentence synthesis, rather than one coming from a full sentence containing the word "Nnɛ", which would have sounded more natural. This could be fixed by improving the unit selection process. For example, when creating a synthesis with a word such as "Nnɛ" inside a sentence as in Synthesis 2, the unit coming from another sentence should be preferred over a unit, which was recorded on its own.

Syntheses 7 and 8 received the lowest scores. In synthesis 7, the word "na" is pronounced incorrectly. According to the respondents, it should be accentuated downwards, rather than upwards. This makes a very big difference, as the accent determines the meaning of the word, one meaning "because" and the other "and". It is not easy to determine this difference from the context of a sentence, therefore correct unit selection is difficult in this case. To fix this problem, the best thing to do would be to differentiate the spelling of the two different words in the text, in order to indicate the appropriate accent. Finally, in synthesis 8, all respondents agreed that the word "toaso" sounds incorrectly or even "funny". In the original recordings, only one sentence contains this word. It turned out that it was not recorded correctly and should be replaced by one or more new recordings with the correct pronunciation and intonation of the word "toaso".

7. EVALUATION

When compared to other existing TTS systems, the TTS Slot and Filler system appeared to be the most suitable for providing TTS support in voice-based services for under-resourced languages. Despite the fact that this software has not been in use for several years and the documentation was rather scarce, a successful attempt was made to first build the system with the default language of Bambara, and later to create support for another African language - Twi, spoken in Ghana. This process required a number of steps and several problems were encountered along the way.

Initially, an unsuccessful attempt was made to build the TTS Slot and Filler software on MacOS. After an investigation, it turned out that it was not possible due to the case-insensitive nature of the operating system. During one of the steps in the system build, a file is created for each phone of the language in a single directory. In some cases, both upper case and lower case of the same letter are used for the Bambara language in order to represent phones that do not exist in our alphabet. To give an example, the phone "ɛ" is represented by the letter "E". Since the Bambara language also contains regular "e", the software crashes on an attempt to create two separate files for "e" and "E", as these two are treated the same by MacOS.

Having learned that, another attempt was made to build the TTS Slot and Filler system on a Linux operating system, in this case Ubuntu. The problem of case sensitivity was solved, but several other issues were encountered. Most of the TTS Slot and Filler code is written in the programming language Python. Since it is a rapidly evolving language, several parts of the original code turned out to be deprecated and were not supported anymore by the obsolete libraries they relied on. These parts were updated to use new libraries and the updated version of the software was made available on Github. Lastly, a number of dependencies had to be installed for the system to work, only some of which were mentioned in the documentation.

Creating support for a new language proved much more straightforward than building the TTS Slot and Filler system for the first time. As mentioned in Section 4.3 *TTS System Build and Language Support*, this can be done three steps. Firstly, a list of phones is obtained and described how it is articulated. Secondly, a set of recordings covering a specific domain are made. Finally, a text file containing the textual version of each recording is created. Once the three steps have been completed, the system can be re-built and tested.

The whole process of building the TTS Slot and Filler system has been attempted four times in total and successfully completed three times (the one unsuccessful attempt was with MacOS). Having done it repeatedly allowed for creating a complete and detailed step-by-step documentation, which has been included in Appendix A. The fourth and final attempt was done as part of the InfoMétéo project described in Section 5: *Case Study: InfoMétéo*. The language used was Twi - one of the major languages spoken in Ghana.

The new build was successfully integrated with the InfoMétéo voice-based application, extending the system by the language of Twi in addition to the already supported English and French. Upon completion of the project, the InfoMétéo team was asked for feedback on the TTS Slot and Filler System. When asked if they would have been able to setup the system without the help of an expert for the first time, they admitted that it would be difficult and even "almost impossible". The original documentation was too scarce not detailed enough to follow by a regular student. That is why it was extended, accounting for installation of dependencies, adding some of the missing steps and including a guide for new language support. The new documentation was shown to the team, asking if they think they could set up the system again by themselves, with the help of the document. This time the answer was very positive. One of the team members - Alexandru - took the effort to try and build the system by himself and managed to do it successfully. He commented: "following the installation steps you provided proved to be very helpful". However, he also mentioned that there were "two intricacies not described in the steps", to do with dependency installation and path setting. After his feedback the documentation was updated once again. Finally, some comments were given about the integration of the TTS Slot and Filler system into the InfoMétéo application. Since text to speech conversion is based on calling only one Python file, the integration turned out to be quite straightforward. The only difficulty faced by the InfoMétéo team was referencing one file representing the corpus of words, which is required by the Python script. This problem was solved by copying the troublesome file into the InfoMétéo application directory.

8. CONCLUSIONS AND FUTURE WORK

To conclude, the TTS Slot and Filler proves to be the most appropriate method for creating Text-to-Speech synthesis for under-resourced languages. It is relatively simple to use, as it can be handled by people with standard programming skills and no background in linguistics is required. Unlike other currently available TTS systems, does not rely on large amounts of textual data in order to add a new language. It is also computationally inexpensive, which is a big advantage given the lack of technological resources in Africa. An important thing to note, however, is that the TTS Slot and Filler system is only suitable for applications with a narrow domain, using no more than a few hundred words. There are two reasons for this limitation. Firstly, the system relies on manually creating a number of recordings fully covering the intended dictionary, which can be time consuming. Secondly, too large of a dictionary might slow down the performance of the application considerably, due to the approach the system is based on. This approach is called unit selection.

Unit selection is based on on splitting all sentences available in the recordings into "units", in this case words. In order to synthesise a new sentence, one unit from all possible choices is selected for each word in the sentence. These units are then concatenated together into a single audio file. The unit selection approach is fast and effective in TTS synthesis with a relatively small dictionary, however using it for thousands of words would prove to be too heavy computationally.

There are many cases in which the TTS Slot and Filler approach can be very useful, despite its limitation to a narrow domain. The system has the potential to make information more accessible to African communities, regardless of their literacy levels or the language they speak. One possible way to do this is to provide multiple language support in the already existing voice-based services. Such services typically store information in a structured way (for example in a spreadsheet in the case of Radio Marché) and focus on one specific task, making them a perfect fit for the TTS system.

As a proof of concept, the TTS Slot and Filler system was integrated into an existing voice-based application called InfoMétéo, which was built by a group of students as part of the ICT4D course at Vrije Universiteit. InfoMétéo is intended as a tool for farmers, allowing them to access information about the weather for the upcoming days. Originally, the system supported the languages of English and French. The TTS Slot and Filler system was used in to extend it by adding a new language, Twi, spoken in Ghana. The InfoMétéo team was able to build and use the system without any major issues with the help of the documentation of the system, which was created as part of this project. According to them, any regular IT student with basic Linux skills should be able to follow the documentation and use the TTS Slot and Filler system. The integration itself also turned out to be straightforward, as it only involved one call to a single Python file.

In order to assess the quality of speech synthesis produced by the system, a group of native Twi speakers was asked for feedback on a set of eight syntheses. In general, the feedback was positive, as five syntheses out of eight were deemed "easy to understand" and sounded "mostly natural". The remaining three were commented as "acceptable" and "partly natural, partly unnatural". The less positive feedback was due to incorrect pronunciation of some words. In the first case, the word was pronounced in an inappropriate way with respect to its placement in the sentence, thus making it sound unnatural. In the second case, the mistake was made by the system in selecting the correct unit for synthesis due to the same spelling of two words with different meanings and pronunciations. In both of these cases, it may be possible to improve the unit selection algorithm of the system by adding additional rules to it, thus effectively improving the understandability and naturalness of the syntheses.

Close to the conclusion of this project, Daniel van Niekerk - the owner and maker of the TTS Slot and Filler system - was asked for clarification of some aspects of the system. After exchanging several e-mails, some interesting insights were revealed. As mentioned throughout the paper, the system relies on the unit selection approach. In the version used for the default Bambara build, as well as for the Twi build, the units represent whole words, thus making the system suitable for a limited dictionary only. However, it turns out that it is also possible to configure the system to use half-phones instead, which would allow for more flexibility in the system. According to Daniel van Niekerk though, building a half-phone or diphone based system would require more recordings, which can be difficult without a native speaker available at hand. Another interesting insight is that a new version of the TTS system is under development at the moment, including extra tools and improvements. Most of them will be used to build HMM-based voices for South African languages. This opens up a good opportunity for future work. Once ready, the new system could be tested and built for other applications to be used in Africa that might require a broader domain.

9. REFERENCES

[1] About the voices project. VOICES http://mvoices.eu/about.html. Accessed: 2017-06-10.

[2] Adding support for a new language to mary tts. Github MaryTTS https://github.com/marytts/marytts/wiki/New-Language-Support. Accessed: 2017-06-06.

[3] Groundswell International. Groundswell International http://www.groundswellinternational.org/burkina-faso/farmers-teach-farmers-in-burkina-faso/. Accessed: 2017-06-04.

[4] How Many Languages of Africa Are There? Lebogang Matshego https://www.africa.com/many-african-languages. Accessed: 2017-05-18.

[5] Mobile Accessibility. W3C https://www.w3.org/WAI/mobile. Accessed: 2017-05-22.

[6] Number of unique mobile subscribers in Africa surpasses half a billion. GSMA http://www.gsma.com/newsroom/press-release/number-of-unique-mobile-subscribers-in-africa-surpasses-half-a-billion-finds-new-gsma-study/. Accessed: 2017-05-12.

[7] N. Amanquah and M. Mzyece. Mobile Application Research and Development: The African Context. In *Proceedings of the 2Nd ACM Symposium on Computing for Development*, ACM DEV '12, pages 20:1–20:1, New York, NY, USA, 2012. ACM.

[8] E. Barnard, P. Bagshaw, M. Froumentin, A. Botha, and F. C. Pinto. Report on state of the art and development methodology. VOIce-based Community-cEntric mobile Services for social development. Deliverable no d3.1. Seventh Framework Programme, 2011.

[9] E. Barnard and F. C. Pinto. Proposal documents to standards committees. VOIce-based Community-cEntric mobile Services for social development. Deliverable no d3.5. Seventh Framework Programme, 2013.

[10] E. Barnard, D. van Niekerk, F. Pinto, and A. Bon. Language packs for local languages. VOIce-based Community-cEntric mobile Services for social development. Deliverable no d3.2. Seventh Framework Programme, 2012.

[11] K. Calteaux, F. De Wet, C. Moors, D. Van Niekerk, B. McAlister, A. S. Grover, T. Reid, M. Davel, E. Barnard, and C. Van Heerden. Lwazi ii final report: Increasing the impact of speech technologies in south africa. Technical report, CSIR, 2013.

[12] A. Custura. InfoMétéo: A System For Propagating Rainfall Information In Developing Countries, 2017.

[13] F. Dittoh, C. van Aart, and V. de Boer. Voice-based Marketing for Agricultural Products: A Case Study in Rural Northern Ghana. In *Proceedings of the Sixth International Conference on Information and Communications Technologies and Development: Notes - Volume 2*, ICTD '13, pages 21–24, New York, NY, USA, 2013. ACM.

[14] N. B. Gyan, V. de Boer, A. Bon, C. van Aart, H. Akkermans, S. Boyera, M. Froumentin, A. Grewal, and M. Allen. Voice-based Web Access in Rural Africa. In *Proceedings of the 5th Annual ACM Web Science Conference*, WebSci '13, pages 122–131, New York, NY, USA, 2013. ACM.

[15] S. C. Pammi, M. Charfuelan, and M. Schröder. Multilingual Voice Creation Toolkit for the MARY TTS Platform. In *LREC 2010*. ELRA, 5 2010.

[16] B. Schnabel and H. Roth. Automatic linguistic processing in a German text-to-speech synthesis system. In *The ESCA Workshop on Speech Synthesis*, 1991.

[17] M. Schröder and J. Trouvain. The German Text-to-Speech Synthesis System MARY: A Tool for Research, Development and Teaching. *International Journal of Speech Technology*, 6(4):365–377, 2003.

[18] P. Taylor. *Text-to-speech synthesis*. Cambridge University Press, 2009.

[19] UNESCO. Adult and Youth Literacy: National, Regional and Global Trends. UNESCO Institute for Statistics, 2013.

APPENDIX

A. TTS SLOT AND FILLER SOFTWARE DOCUMENTATION

This section provides a detailed documentation of TTS Slot and Filler documentation. It comprises of four parts: A.1 System Requirements, A.2 Dependencies, A.3 TTS Slot and Filler System Build and A.4 Adding Support for a New Language. Sections A.2 and A.3 are partly based on the documentation made by Daniel van Niekerk, which can be found on the TTS Slot and Filler Bambara Build repository on github: *https://github.com/demitasse/ttslab_bambara_build*.

A.1 System Requirements

The system must run on a case-sensitive operating system, therefore a Linux system is preferred. MacOS may also be suitable, but only if the case-sensitive version was installed (the current default is case-insensitive).

The operating system used in for the build described in this documentation is 64-bit Ubuntu 16.04.2 LTS.

A.2 Dependencies

A number of dependencies is required for the system build. They include:

- Python 2.7 or higher
- Numpy
- Scipy
- SoX
- Normalize-audio
- Praat
- HTK 3.4.1
- Edinburgh Speech Tools 2.4

This section contains a step by step guide on how each dependency should be installed.

A.2.1 Python 2.7 or higher

Python comes with the majority of Linux based systems pre-installed. Python 2.7 was used for the build described in this document, therefore using this version is recommended.

A.2.2 Numpy

Numpy is a Python library. It can be installed with Python installation tool called pip.

To get pip, run the following command from the command line:

```
$ sudo apt install python-pip
```

Once pip has been successfully installed, run the following command to install numpy:

```
$ pip install numpy
```

A.2.3 Scipy

Scipy is another Python library. To install scipy, run the following command from the command line:

```
$ pip install scipy
```

A.2.4 SoX

SoX is command line utility that can convert various formats of computer audio files to other formats, apply various effects to these sound files and play and record audio files.

To install SoX, run from the command line:

```
$ sudo apt-get install sox
```

A.2.5 Normalize-audio

Normalize-audio is a tool for adjusting the volume of audio files to a standard level. To install it, run the following command from the command line:

```
$ sudo apt-get install normalize-audio
```

Verify the installation by typing:

```
$ normalize-audio --version
```

This should result in a message similar to below:

```
normalize 0.7.7
Copyright (C) 2005 Chris Vaill
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
This copy of normalize is compiled with the following libraries:
  MAD  audiofile
```

For more information on normalize-audio, visit *http://normalize.nongnu.org*.

A.2.6 Praat

Praat is a software used for, among others, speech analysis and speech synthesis.
To install it, run the following command from the command line:

```
$ sudo apt-get install praat
```

Verify the installation by running praat:

```
$ praat
```

This command should launch the praat application.

A.2.7 HTK 3.4.1

The Hidden Markov Model Toolkit (HTK) is a portable toolkit for building and manipulating hidden Markov models.
Download the Stable Release (3.4.1) HTK source code for Linux from the following link:

http://htk.eng.cam.ac.uk/download.shtml

To do this, you will need to register with your e-mail address. Once the tarball has been downloaded, move it to the dependencies directory earlier created, untar and clean up:

```
$ mv ~/Downloads/HTK-3.4.1.tar.gz .
$ tar xvf HTK-3.4.1.tar.gz
$ rm HTK-3.4.1.tar.gz
$ cd htk
```

Before HTK can be installed, some dependencies need to be satisfied. To do this, run the following command:

```
$ sudo apt-get install libc6-dev-i386 libx11-dev:i386 libx11-dev
```

Now, configuration can be run:

```
$ ./configure
```

Before building, one file needs to be updated, as there appears to be an error in one of the `Make` files included in HTKTools. Open the `Makefile` file residing in `~/Documents/tts/dependencies/htk/HTKTools/` and on line 77 change the 8 spaces before the if statement into one tab. Save and close the file. Then, from the location `~/Documents/tts/dependencies/htk` run:

```
$ make all
$ make install
```

The second command may have to be run with sudo.

A.2.8 Edinburgh Speech Tools 2.4

The Edinburgh Speech Tools Library is a collection of C++ classes, functions and related programs for manipulating the sorts of objects used in speech processing.
Download the speech-tools-2.4 from the following link:

http://www.cstr.ed.ac.uk/downloads/festival/2.4

Once the tarball has been downloaded, move it to the `dependencies` directory earlier created, untar and clean up:

```
$ mv ~/Downloads/speech_tools-2.4-release.tar.gz .
$ tar xvf speech_tools-2.4-release.tar.gz
$ rm speech_tools-2.4-release.tar.gz
$ cd speech-tools
```

Edinburgh Speech Tools also needs a dependency to be installed. To install it, run:

```
$ sudo apt-get install libncurses5-dev libncursesw5-dev
```

Now you can configure, make and install:

```
$ ./configure
$ make
$ make install
```

Now you're done! All dependencies have been installed for the TTS Slot and Filler system.

A.3 TTS Slot and Filler System Build

This section assumes that all dependencies needed for the built have already been installed. The same directory structure as described in the Dependencies section is assumed. For the list of dependencies and their installation guide, please refer to Appenddix A.2.

A.3.1 Source code download

In this documentation, the base directory will be the `tts` directory, which was created during the installation of dependencies. Therefore, before doing anything else, first move to the `tts` directory:

```
$ cd ~/Documents/tts
```

To download the TTS software source code, git is required. If git is not installed, install it from the command line:

```
$ sudo apt-get install git
```

Git clone `ttslab`, `ttslabdev` and `tts_bambara_build` from the following repositories:

```
$ git clone https://github.com/jkleczar/ttslab.git
$ git clone https://github.com/jkleczar/ttslabdev.git
$ git clone https://github.com/jkleczar/ttslab_bambara_build
```

Now, in the `tts` directory the following directories should be pressent:

```
ttslab
ttslabdev
ttslab_bambara_build
```

A.3.2 Setting paths to the TTS software

Paths must be created for `ttslab` and `ttslabdev` directories by adding the following lines to the `.bashrc` file:

```
export TTSLAB_ROOT=~/Documents/tts/ttslab
export TTSLABDEV_ROOT=~/Documents/tts/ttslabdev
```

Note that these paths assume the directory structure used in this documentation.

Additionally, the following paths need to be added to PYTHONPATH. To do this, add the following two lines at the end of the `.bashrc` file:

```
export PYTHONPATH=$TTSLABDEV_ROOT/modules:$TTSLAB_ROOT:$PYTHONPATH
export PATH=$TTSLABDEV_ROOT/scripts:$PATH
```

Once this is done, restart the Terminal for the changes to take effect.

A.3.3 Compiling text-processing front-end

Before Bambara build can be done, first text-processing front-end must be compiled. If `ttslab` and `ttslabdev` have been setup correctly (as described above), the pronunciation resources can be compiled. This will create the voice definition of the language.

To do this, change to the `ttslab` directory in `~/Documents/tts/ttslab_bambara_build` and run `make.sh`:

```
$ cd ~/Documents/tts/ttslab_bambara_build/ttslab
$ ./make.sh
```

Once done, the directory structure should look as follows:

```
.
|-- data
|   `-- pronun
|       |-- addendum.dict
|       `-- main.rules
|-- frontend.voice.pickle
|-- frontend.wordus.voice.pickle
|-- g2p.pickle
|-- make.sh
|-- phoneset.pickle
|-- pronunaddendum.pickle
`-- pronundict.pickle
```

This has taken the Bambara language definition in `TTSLAB_ROOT/ttslab/voices/bambara_default.py`, the simple pronunciation dictionary and letter to sound rules in `BAMBARA_BUILD/ttslab/data/pronun` and compiled a voice definition `BAMBARA_BUILD/ttslab/frontend.wordus.voice.pickle`.

A.3.4 Setting paths to dependencies

The path to the HTK and Edinburgh Speech Tools binaries must be set in paths.sh, residing in
~/Documents/tts/ttslab_bambara_build. Please edit this file to set these paths.

Following the directory structure in this documentation, the paths are set as follows:

```
HTK_BIN=$HOME/Documents/tts/dependencies/htk/HTKTools
EST_BIN=$HOME/Documents/tts/dependencies/speech_tools/bin
```

A.3.5 Building the system

The directory `BAMBARA_BUILD/recordings/chunked` contains a `wavs` directory containing an audio file (with basename matching) for every entry in `utts.data` (residing in `BAMBARA_BUILD/recordings/chunked`). Audio files need to be in RIFF Wave format (any bitrate and samplerate) and `utts.data` in UTF-8 text.

In the Bambara build source code that was downloaded from github, the recordings and `utts.data` file have already been set up, so `setup_alignments.sh` can be run from the `BAMBARA_BUILD` root directory:

```
./setup_alignments.sh
```

This should create a `build` directory and process the audio files appropriately (downsampling and energy normalisation) and copy the `etc` configuration directory for this build.

Once the `build` directory is constructed, the alignment process can be done (a full path to the voice front-end file is required):

```
./do_alignments.sh $PWD/ttslab/frontend.wordus.voice.pickle
```

This should add the following subdirectories in `build`: `halign`, `textgrids` and `utts`. Next, feature extraction for the database can be done:

```
./do_us_catalogue.sh $PWD/ttslab/frontend.wordus.voice.pickle feats
```

Next, compile the database:

```
./do_us_catalogue.sh $PWD/ttslab/frontend.wordus.voice.pickle catalogue
```

This should result in the file `unitcatalogue.pickle` being created in `build` directory. This file should have a symbolic link in `BAMBARA_BUILD/ttslab/data` allowing the following to be executed in the `BAMBARA_BUILD/ttslab` directory:

```
cd ttslab
ttslab_make_voice.py wordus
```

resulting in the file `BAMBARA_BUILD/ttslab/wordus.voice.pickle`.

A.3.6 Troubleshooting

If any of the above steps fail, it is likely that one of the dependencies has not been installed correctly or that paths have not been set as described in this documentation. Double-check the paths and installations. After any possible fixes, run the following commands before attempting to build again:

```
./setup_alignments.sh clean
./do_alignments.sh clean
./do_us_catalogue.sh clean
```

Once the build has been cleaned, start the build from the beginning:

```
./setup_alignments.sh
./do_alignments.sh $PWD/ttslab/frontend.wordus.voice.pickle
./do_us_catalogue.sh $PWD/ttslab/frontend.wordus.voice.pickle feats
./do_us_catalogue.sh $PWD/ttslab/frontend.wordus.voice.pickle catalogue
```

A.3.7 Testing

The system can be tested in Python in the following way:

```
# encoding: utf-8
import ttslab

voice = ttslab.fromfile("wordus.voice.pickle")
utt = voice.synthesize(u'Banbara', "text-to-wave")
utt["waveform"].write("test.wav")
```

The first string in `voice.synthesise` can be replaced with a sentence consisting of any words residing in `BAMBARA_BUILD/recordings/chunked/utts.data` file. Running this script results in a .wav file, containing an audio recording of the sentence.

A.4 Adding Support for a New Language

To create a TTS Slot and Filler system for a new language of choice, a number of steps must be fulfilled. They include:

- **Step 1:** Creating a description of the language in terms of phone pronunciation
- **Step 2:** Creating a set of recordings covering the chosen domain of the application
- **Step 3:** Creating an accompanying text file for the recordings

This section gives a detailed overview of how each step should be executed with the example of the Twi language.

A.4.1 Step 1: Description of the Language

All supported language descriptions reside in the `ttslab` toolkit, which was cloned from github in the first step of the TTS System Build section. To see all supported languages, change to the following directory:

```
$ cd ~/Documents/tts/ttslab/ttslab/voices
$ ls
```

In the Bambara build described in the TTS System Build section, the `bambara_default.py` file was used. A similar file for the language of Twi must be created. To do this, the `bambara_default.py` file will be copied and adjusted to the language of Twi:

```
$ cp bambara_default.py twi_default.py
```

First, the name of the class needs to be changed from `BambaraPhoneset` to `TwiPhoneset`. The phoneset of the language is defined between lines 17 and 89 in the variables `self.features`, `self.phones` and `self.map`. Each one of these will have to be adjusted for the language of Twi.

The `self.features` variable is the simplest one to adjust, as it only provides some basic features of the language. For the Bambara language, it has been defined as:

```
self.features = {"name": "Bambara␣Phoneset",
                 "silence_phone": "pau",
                 "closure_phone": "pau_cl"
                 }
```

The only thing that needs to be changed in this variable for the language of Twi is the name:

```
self.features = {"name": "Twi␣Phoneset",
                 "silence_phone": "pau",
                 "closure_phone": "pau_cl"
                 }
```

The `self.phones` variable contains the description of phones. In order to create it correctly, the phoneset of the Twi language is needed. It can be either provided by a native speaker or obtained from other resources. For the purpose of this example, the vowels and consonants provided on *http://www.omniglot.com/writing/akan.htm* are used:

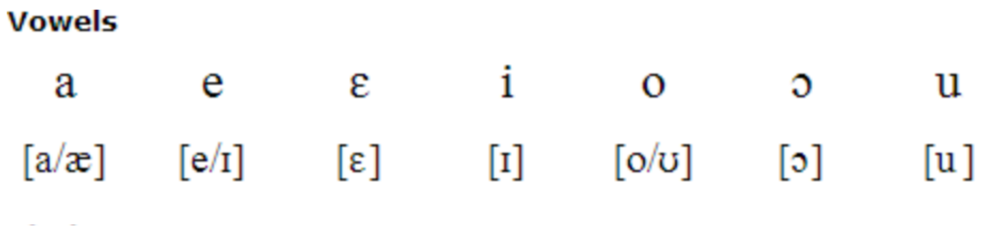
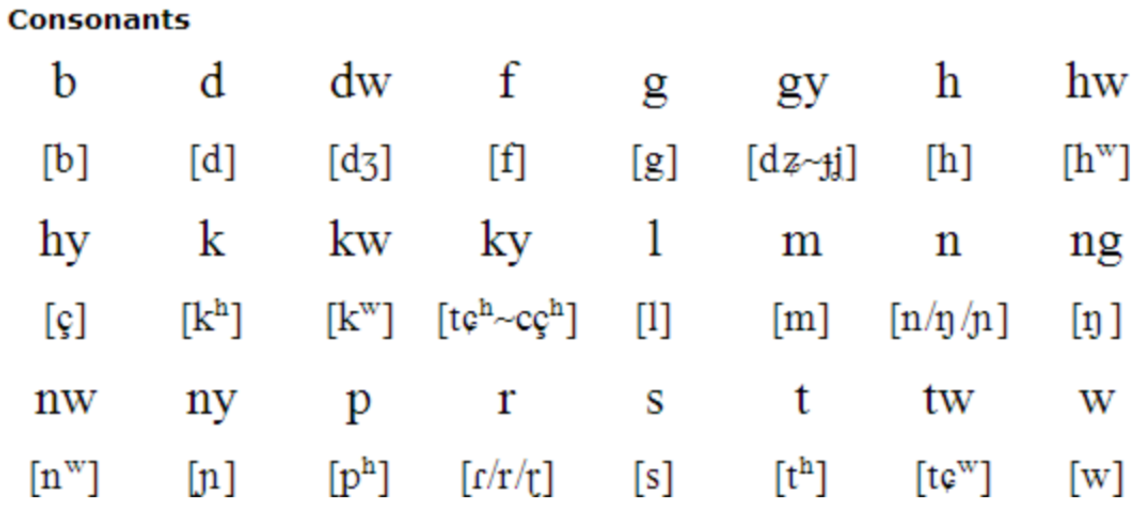

Vowels
a e ɛ i o ɔ u
[a/æ] [e/ɪ] [ɛ] [ɪ] [o/ʊ] [ɔ] [u]

Figure 4: Vowels of the Twi language


Consonants

b	d	dw	f	g	gy	h	hw
[b]	[d]	[dʒ]	[f]	[g]	[dʑ~ɟj]	[h]	[h^w]
hy	k	kw	ky	l	m	n	ng
[ç]	[k^h]	[k^w]	[tɕ^h~cç^h]	[l]	[m]	[n/ŋ/ɲ]	[ŋ]
nw	ny	p	r	s	t	tw	w
[n^w]	[ɲ]	[p^h]	[ɾ/r/ɽ]	[s]	[t^h]	[tɕ^w]	[w]

Figure 5: Consonants of the Twi language

If we compare the list of the Twi vowels to the Bambara ones, they are all the same, thus they can be left unchanged. There are, however, a number of different consonants, which need to be added. To get started, all missing Twi consonants can be added without any description values and unnecessary Bambara language consonants should be removed, leaving the rest unchanged:

```
#consonants
”b”      : set([”class_consonantal”, ”consonant”, ”manner_plosive”, ”place_bilabial”,
                  ”voiced”]),
”d”      : set([”class_consonantal”, ”consonant”, ”manner_plosive”, ”place_alveolar”,
         ”voiced”]),
”dw”     : set([]),
”f”      : set([”class_consonantal”, ”consonant”, ”manner_fricative”,
              ”place_labiodental”]),
”g”      : set([”class_consonantal”, ”consonant”, ”manner_plosive”, ”place_velar”,
              ”voiced”]),
”gy”     : set([]),
”h”      : set([”consonant”, ”manner_fricative”, ”place_glottal”]),
”hw”     : set([]),
”hy”     : set([]),
”k”      : set([”class_consonantal”, ”consonant”, ”manner_plosive”, ”place_velar”]),
”kw”     : set([]),
”ky”     : set([]),
”l”      : set([”class_sonorant”, ”class_consonantal”, ”consonant”,
              ”manner_approximant”, ”manner_lateral”, ”place_alveolar”, ”voiced”]),
”m”      : set([”class_sonorant”, ”class_syllabic”, ”class_consonantal”, ”consonant”,
              ”manner_nasal”, ”place_bilabial”, ”voiced”]),
”n”      : set([”class_sonorant”, ”class_syllabic”, ”class_consonantal”, ”consonant”,
              ”manner_nasal”, ”place_alveolar”, ”voiced”]),
”ng”     : set([]),
”nw”     : set([]),
”ny”     : set([]),
”p”      : set([”class_consonantal”, ”consonant”, ”manner_plosive”,
              ”place_bilabial”]),
”r”      : set([”class_sonorant”, ”class_consonantal”, ”consonant”, ”manner_trill”,
              ”place_alveolar”, ”voiced”]),
”s”      : set([”class_consonantal”, ”consonant”, ”manner_fricative”,
              ”place_alveolar”]),
”t”      : set([”class_consonantal”, ”consonant”, ”manner_plosive”,
              ”place_alveolar”]),
”tw”     : set([]),
”w”      : set([”class_sonorant”, ”consonant”, ”manner_approximant”, ”place_labial”,
                    "place_velar", "voiced"])
```

Each of the new phones must be described in as much detail as possible to determine how they are pronounced. This helps with the final quality of the recordings and their fluency. There are several categories with different values that can be used for this description:

1. Letter Type
   - **Vowel**
   - **Consonant**
2. Phone Class
   - **Sonorant:** in phonetics and phonology, a sonorant is a speech sound that is produced with continuous, non-turbulent airflow in the vocal tract. Examples: l, n, r
   - **Consonantal:** a consonantal speech sound is the opposite of sonorant. Examples: t, p, k.
3. Manner of pronunciation[4]
   - **Plosive:** consonants, where air is blocked at the place of articulation to accumulate pressure and it is then released in one instant. Examples: d, b, k.
   - **Fricative:** consonants, where vocal apparatus is used to partially block the airflow at the place of articulation in such a way that only some air passes through. Examples: s, sh.
   - **Approximant:** consonants where the air flows smoothly through the vocal apparatus so that very little friction is created. Examples: y, r.
   - **Lateral:** consonants, where the airflow passes to the sides (of the tongue, usually) when pronouncing them. Examples: l.
   - **Nasal:** consonants, where you let air out of your nose as you pronounce them. Examples: m, n.
   - **Trill:** consonants, where at the place of articulation, a series of repeated bursts is made. Examples: repeated r.
4. Place of pronunciation[5]
   - **Bilabial:** consonant sounds produced by using both lips together.
   - **Labial:** consonants articulated by using both the tongue and the upper lip.
   - **Labiodental:** consonants articulated by using both the lower lip and the upper front teeth.
   - **Alveolar:** consonants pronounced near the alveolar ridge which is the area lying between the upper front teeth and the palate.
   - **Velar:** consonants pronounced at the back of the palate.
   - **Glottal:** consonanced pronounced at the back of the throat.

Following the guidelines outlined above, the missing consonants can be now described in terms of letter type, phone class, manner and place of pronunciation:

```
"dw"    : set(["consonant", "class_consonantal", "manner_plosive", "place_labial"]),
"gy"    : set(["consonant", "class_consonantal", "manner_plosive", "place_labial"]),
"hw"    : set(["consonant", "class_consonantal", "manner_fricative",
                "place_glottal"]),
"hy"    : set(["consonant", "class_sonorant", "manner_fricative", "place_alveolar"]),
"kw"    : set(["consonant", "class_consonantal", "manner_plosive", "place_velar"]),
"ky"    : set(["consonant", "class_consonantal", "manner_plosive",
                "place_alveolar"]),
"ng"    : set(["consonant", "class_consonantal", "manner_nasal", "place_labial"]),
"nw"    : set(["consonant", "class_consonantal", "manner_nasal",
                "place_labiodental"]),
"ny"    : set(["consonant", "class_consonantal", "manner_nasal",
                "place_labiodental"]),
"tw"    : set(["consonant", "class_consonantal", "manner_plosive",
                "place_alveolar"]),
```

Having the consonants defined and described, the `self.map` variable can be adjusted. Here, the special characters such as ε should be mapped onto a regular character - in this case, it will be E. A mapping must be made for all phones in the phoneset. The final mapping is as follows:

```
self.map = {"pau"    : "pau",
            "pau_cl" : "pau_cl",
            "?"      : "pau_gs",
```

[4]Source: http://www.learnlanguagesonyourown.com/manners-of-articulation.html
[5]Source: http://www.learnlanguagesonyourown.com/places-of-articulation.html

```
                "a"        :  "a",
                "e"        :  "e",
                "ϵ"        :  "E",
                "i"        :  "i",
                "o"        :  "o",
                "ω"        :  "O",
                "u"        :  "u",
                "b"        :  "b",
                "d"        :  "d",
                "dw"       :  "dw",
                "f"        :  "f",
                "g"        :  "g",
                "gy"       :  "gy",
                "h"        :  "h",
                "hw"       :  "hw",
                "hy"       :  "hy",
                "j"        :  "j",
                "k"        :  "k",
                "kw"       :  "kw",
                "ky"       :  "ky",
                "l"        :  "l",
                "m"        :  "m",
                "n"        :  "n",
                "ng"       :  "ng",
                "nw"       :  "nw",
                "ny"       :  "ny",
                "p"        :  "p",
                "r"        :  "r",
                "s"        :  "s",
                "z"        :  "z",
                "t"        :  "t",
                "tw"       :  "tw",
                "w"        :  "w"
                }
```

Now that all the rules have been set, a new build can be prepared for the language of Twi. To get started, we will work with the original build of Bambara and adjust it to the language of Twi. Just as in Bambara build, git clone the source code from github and rename the old build to something else for the time being:

```
$ cd ~/Documents/tts
$ mv ttslab_bambara_build ttslab_bambara_build_old
$ git clone https://github.com/jkleczar/ttslab_bambara_build
```

Once done, rename the newly downloaded `ttslab_bambara_build` to `ttslab_twi_build`:

```
$ mv ttslab_bambara_build ttslab_twi_build
```

Since the Twi language description has already been added to `ttslab`, we can use it in the Twi build by editing the `make.sh` file residing in `~/Documents/tts/ttslab_twi_build/ttslab`. The file should be edited as follows, conforming to the language definition we have already created:

```
#!/bin/bash

ttslab_make_phoneset.py twi_default TwiPhoneset
ttslab_make_g2p.py
ttslab_make_pronundicts.py
ttslab_make_voice.py frontend
ttslab_make_voice.py wordusfrontend
ttslab_make_voice.py wordus
```

One more file needs to be adjusted. In `~/Documents/tts/ttslab_twi_build/ttslab/data/pronun` there is a file called `addendum.dict` specifying the spelling of several words in the language. At the moment the file contains words from the Bambara language. It should be changed to give several examples of the Twi language, for instance:

```
PAUSE pau
rain n s u o
no d a a b i
```

```
heavy d e n e
temperature b O b r E E
yesterday E n o r a
Monday E d w o a d a
Tuesday E b e n a d a
Wednesday w u k u a d a
Friday e f i a d a
```

Once the file has been saved, proceed to Step 2.

A.4.2 Step 2: Recordings for Chosen Domain

Before creating the recordings, a specific target application or domain should be determined. Then, an example script containing a set of sentences should be designed, ensuring that the sentences cover the domain as much as possible. In some cases, the content will vary - for example, things such as dates, numbers or other values that can be variable. These variations should also be included in the recordings.

It is preferable that the sentences of the script are recorded in the manner of one sentence per recording, in order to preserve the prosodic features of the utterances. The variable content may be recorded separately - for instance, one recording per each date or number.

If possible, all recordings should be created by one person in constant conditions, preferably all in one session. Background noise should be avoided in order to ensure clarity and quality of the final output of the system. All sentences and words should be spoken quite slowly, at a comfortable tempo. In order to increase the quality of the recordings, audio software may be used in order to reduce noise and normalize the loudness across the recordings. For the purpose of Twi recordings, Audacity software was used.

For the use case of Meteo readings in the language of Twi, a total of 53 recordings were made. They included sentences describing the weather conditions, voice commands allowing choice of language or location, as well as extra recordings for variable content such as days of the week, numbers and different weather conditions.

The TTS Slot and Filler software at the moment supports the RIFF Ware format (.wav), thus all recordings should be made or converted into that format. When the recordings have been created, they should be placed in `~/Documents/tts/ttslab_twi_build/recordings/chunked/wavs`. Make sure that all the recordings from the Bambara build are removed before placing the Twi recordings there.

A.4.3 Step 3: Text File for the Recordings

The final step in adding support for a the Twi language is creating a file containing the contents of all recordings that were just added. The file is called `utts.data` and resides in `~/Documents/tts/ttslab_twi_build/recordings/chunked/` and takes the format of one recording description per line:

```
( RecordingName "Content␣of␣the␣recording" )
```

Since at the moment this file describes all Bambara recordings, it should be adjusted according to the Twi recordings. In total, there should be 53 lines in this file, as there are 53 recordings. An important thing to note is that all spellings of the words should conform to the phoneset defined in Step 1.

Once `utts.data` file has been updated with the correct data and saved, the build for the Twi language can be created by following steps described in A.3.5 in the TTS Slot and Filler System Build section of this appendix.

B. RECORDINGS FOR THE TWI LANGUAGE

εBenada
εdwoada
εhai
εhum
εnan
εnora
εnoraakyi
εnum
εyaara
awω
awωpaa
Baako
brεε
Mea one na hunu ewiem nsakrayε aa εwω Kumasi
Mea one na hunu ewiem nsakrayε aa εwω Tamale
Daabi
Mea one na hunu εnora akyi ewiem nsakrayε aa na εwω Tamale
dene
Edu

Efiada
Hwee
ωhyew
ωhyewpaa
Nyε ne Kwan so nono. San yε no fofroω
Kwasiada
ωkyena
ωkyenaakyi
Medase
Memeneda
Mmeεnsa
Mmienu
Nkron
nnε
Nsia
Nson
nsuo
Nwωtwe
Na nsuo no ano yε den
Nsuden bεtω wω Tamale nnε
nsuo rentω wω Tamale ωkyena
ωhyew paa bεba Tamale ωkyena akyi
Sε wo pε twi aa mea one
Mea one na hunu εnora ewiem nsakrayε aa εwω Tamale")
Mea one na hunu nnε ewiem nsakrayε aa εwω Tamale
Mea one na hunu ωkyena ewiem nsakrayε aa εwω Tamale
Wukuada
Yawoada
Nsuo antω wω Tamale εnnora
εnnora nsuo tωω wω Tamale
Na εnnora bωbrεε wω Tamale yε hye
Me pa wo kyew yi numeri yi mu na toaso
εde wo bεsan akω napωso napωso
Medase

C. SPEECH SYNTHESIS FOR FEEDBACK

Synthesis 1.
Nsuo bεtω wω Tamale nnε.
There will be rain in Tamale today.
Synthesis 2.
Nsuo bεtω wω Kumasi nnε.
There will be rain in Kumasi today.
Synthesis 3.
Nsuo bεtω wω Kumasi ωkyena.
There will be rain in Kumasi tomorrow.
Synthesis 4.
Me pa wo kyew mea Mmeεnsa.
Please press 3.
Synthesis 5.
Nsuo rentω wω tamale nnε.
There will be no rain in Tamale today.
Synthesis 6.
Me bεsan akω Kumasi ωkyena akyi.
I will return to Kumasi the day after tomorrow.
Synthesis 7.
Me bεsan akω Tamale na ωhyew paa bεba Kumasi ωkyena.
I will return to Tamale because it will be very hot in Kumasi tomorrow.
Synthesis 8.
Me pa wo kyew, toaso.
Please continue.