University of Erlangen-Nuernberg

Chair of Multimedia Communications and Signal Processing

Prof. Dr.-Ing. Walter Kellermann

Research Internship

Training of a word model-based offline automatic speech recognition system for a small-vocabulary speech recognition task

Ralf Gross

May 2018

Supervisor: Prof. Dr.-Ing. Walter Kellermann
M.Sc. Hendrik Barfuss

Lehrstuhl für Multimediakommunikation
und Signalverarbeitung
Prof. Dr.-Ing. W. Kellermann

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Research internship

for

Mr. Ralf Gross

Training of a word model-based offline automatic speech recognition system for a small-vocabulary speech recognition task

During the EU-FP7 Project Embodied Audition for RobotS (EARS) a real-time demonstrator for robot audition was developed, which consists of a signal enhancement stage containing a multichannel spatial filter and a single-channel post-filter, followed by a cloud-based, i.e., online, automatic speech recognition (ASR) system.

In previous work, the possibility to integrate an offline speech recognition system, primarily as backup for the online system in situations without access to an internet connection, was investigated. The respective approach used *CMUSphinx* (based on HMM-GMM models) as ASR engine and the Python *SpeechRecognition* package as interface to the demonstrator. The evaluation showed that with the available training and evaluation data, no satisfying recognition accuracy could be obtained.

The goal of this research internship is to train a word model-based ASR system for the purpose of a small-vocabulary speech recognition task. The research internship therefore involves the following tasks:

1. Training of an word model-based acoustic model for the small-vocabulary recognition task of the *CHiME* challenge
2. Verification of the trained acoustic model with evaluation data recorded by the 12-element microphone array integrated into the robot head
3. Investigation of the impact of the speech enhancement algorithms employed in the current version of the demonstrator.

The implementation is to be done using either Matlab or Python. A well-structured and well-documented code has to be handed in at the end of the research internship.

(Prof. Dr.-Ing. W. Kellermann)

Abstract

Offline automatic speech recognition (ASR) systems are required for systems without or with unreliable connection to a usually more powerful online automatic speech recognition system. During the EU-FP7 Project Embodied Audition for RobotS (EARS) a real-time demonstrator for robot audition was developed, which consists of a signal enhancement stage containing a multichannel spatial filter and a single-channel postfilter. In previous work, the offline speech recognition system CMUSphinx, based on HMM-GMM models, has been integrated into the demonstrator. With the available training and evaluation data the recognition accuracy was unsatisfactory.

Therefore, the TIDIGITS dataset, consisting of 25099 files containing only ten different words, was recorded in two different acoustic environments. Based on the recorded files, different acoustic models were trained using CMUSphinx. The evaluation shows that, for this limited set of words, the offline ASR system obtains a satisfactory recognition accuracy, even without preprocessing. Furthermore, it can be concluded that the microphones used in the demonstrator do not have a negative influence on the recognition accuracy

Contents

Chapter 1

Introduction

The usage of robots becomes increasingly convenient in private as well as in professional environments. In many cases, the interaction with humans using a speech interface is highly desirable. Therefore, the respective robots need to be able to understand and interpret words and sentences spoken by humans. A basic requirement to realise this capability for human-robot interaction is a sufficient accuracy in the recognition of words using an ASR system. As such system often work on an autonomous base regarding the power supply, systems are desirable which are useable on low-resource platforms and embedded hardware. In this work, the performance of a light-weight speech recognition engine for handheld and mobile devices is investigated using a small-vocabulary speech recognition task.

Chapter 2

Recording Setup

For evaluating the performance of the ASR system in combination with the NAO robot the small-vocabulary TIDIGITS dataset is recorded in two different acoustic environments. As the TIDIGITS dataset itself is separated into a training set and a test set, in total four datasets are recorded. Before recording, the TIDIGITS dataset, originally sampled with 20kHz, is resampled to 16kHz.

An acoustic environment with a low reverberation time of 150ms and high attenuation due to a fabric cover of the walls is used to allow for the analysis of the influence of the recording using the twelve microphones of NAO's head. This acoustic environment, the audio laboratory at the LMS (in the following just *Audio Lab*), is sound-insulated. Therefore, no background noise is contained in the recorded data. To get data usable for the evaluation of the performance of different preprocessing steps employed, a reverberant environment is selected. The choosen seminar room (in the following denoted as *Seminar Room*) has a reverberation time of 900ms, no sound-insulation and a wide window facade. It is very likely that a considerable amount of the recorded data contains additional background noise (however, no double-talk should be contained in a predominant part of the data). In the following, the resulting datasets are denoted as:

Training set recorded in Seminar Room	*Seminar Room TrainSet*
Training set recorded in Audio Lab	*Audio Lab TrainSet*
Test set recorded in Seminar Room	*Seminar Room TestSet*
Test set recroded in Audio Lab	*Audio Lab TestSet*

In both environments the recording setup consists of a loudspeaker, the NAO head and two condenser microphones for reference recordings. Additionaly, a loopback channel directly connecting an output of the soundcard to an input is used to check the data output. For the recording, each file contained in the TIDIGITS dataset is played and recorded separately, to allow for an automatic labeling of the resulting data. Before playing a file, it is normalized to a maximum value slightly below one to avoid clipping in the loopback channel. After the recording of each file, the recorded data is downsamled from 48kHz to 16kHz before saving. In total, 15 channels were recorded in parallel denoted as *Loopback* for the loopback channel, *RefMic1* for the condenser microphone close to the loudspeaker, *RefMic2* for the condenser microhphone close to NAO's head as well as the twelve channels from the microphones of NAO's head, *NAO1* to *NAO12*.

As far as practicable, the recording setup was positioned with the same distances and heights in both recording environments. Figure 2.1 and Figure 2.2 show schematic views of the recording setup in both environments. On Figure 2.4 and Figure 2.3 the recording setup in the respective environments can be seen.

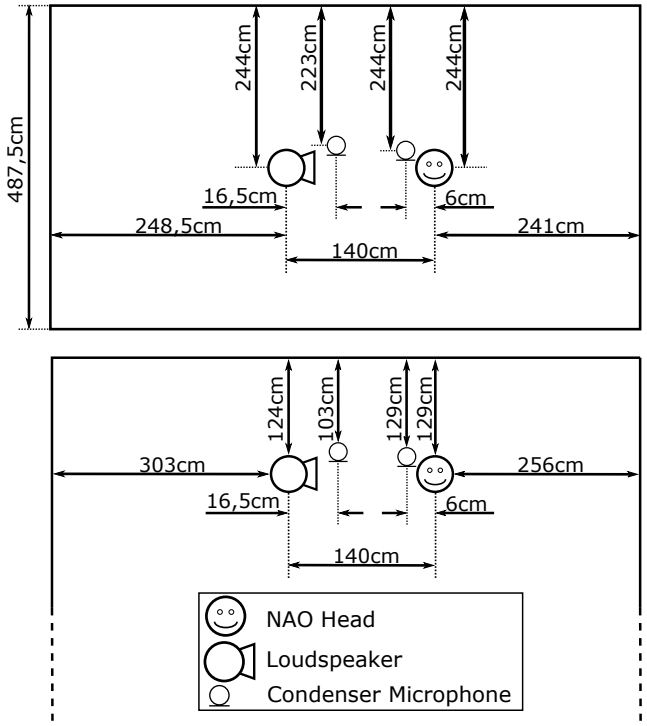

Figure 2.1: Top view of recording setup. (Top: Audio Laboratory; Bottom: Seminar Room) The relative positions of all devices to each other used are the same in both environments. In the Audio Lab the recording setup is positioned in the middle of the room.

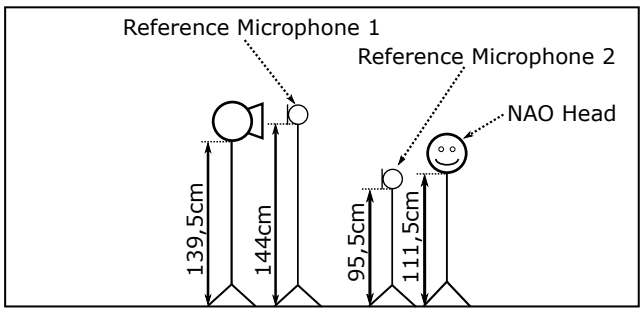

Figure 2.2: Side view of recording setup in both environments.

Figure 2.3: Recording Setup in Audio Lab.

Figure 2.4: Recording Setup in Seminar Room.

Chapter 3

Evaluation

3.1 Pocketsphinx

Pocketsphinx is an offline ASR system developed at the Carnegie Mellon University in Pittsburgh, Pennsylvania. It is designed for low-resource platforms with focus on practical applications. Support for several languages is provided. Pocketsphinx uses HMMs and GMMs as acoustic model and a N-Gram Model as language model. Acoustic models are trained using the Baum-Welch algorithm ([CMU]). A description of these concepts can be found in [YD15]. For the following evaluations, the *US English* package is used, containing an acoustic model as well as a language model.

3.2 NIST Scoring Toolkit

The NIST Speech Recognition Scoring Toolkit (SCTK) is used to evaluate the performance of pocketsphinx with different acoustic models. As the training as well as the recognition is performed using a word-dependent phonetic dictionary, the evaluation is based on the word recognition performance calculated by the SCTK. The *Percent Total Error* contained in the output file of the evaluation is calculated by

$$\textit{Percent Total Error} = \textit{Percent Substitutions} + \textit{Percent Deletions} + \textit{Percent Insertions}.$$

This value is a common metric of a speech recognition system better known as Word Error Rate (WER).

3.3 Evaluation using different acoustic models

Using the previously recorded data, several acoustic models are trained. Therefore, the phonetic dictionary as specified in appendix C.1 is used. The phonetic dictionary is designed word-dependent, resulting in multiple HMMs for the same phoneme in different words (e.g. for the phoneme *ay* two HMMs are trained, *AY_five* and *AY_nine*). This results from the fact that all HMMs in pocketsphinx need to have the same number of states, i.e. HMMs with variable length are not supported. The usage of a word-dependent phonetic dictionary is recommended for pocketsphinx as an equivalent alternative to word-based models ([CMU]). For the training of acoustic models, only the training set of the TIDIGITS dataset is utilized. In total, three different acoustic models are trained, based on the original dataset and on the datasets recorded in the seminar room and audio laboratory (denoted as *original trainset model*, *seminarroom trainset model* and *audiolab trainset model*, respectively). Furthermore, the evaluation is also performed using the acoustic model already contained in the *US english* package (denoted as *standard model*). Independent of the acoustic model, the language model provided with the *US English* package is used for all evaluations.

The decoding of the data using Pocketsphinx is performed without additional pre-processing, i.e. the data is used as it was recorded. Figure 3.1-3.6 show the resulting WERs using the different acoustic models. As a benchmark for the evaluation results, the acoustic model trained on the original training set of the TIDIGITS dataset is used to decode the original dataset itself. The resulting "best achievable" WERs are:

1.4% TIDIGITS training set

1.6% TIDIGITS test set

In contrast, decoding the original dataset using the *standard model* in combination

with the full phonetic dictionary, containing over 130000 phonetic transcriptions of english words, leads to an WER of 42.9% for the training set.

Figure 3.1 shows the resulting WERs obtained by decoding the training set recorded in the *Audio Lab*. With the *standard model*, the WERs of the data recorded with the microphones in the NAO head have an average value of 13.92% with a best score of 13.5% (NAO5, NAO12) and a worst score of 14.7% (NAO4). Using the acoustic model trained on the original dataset (*original trainset model*) the average value regarding all microphones in NAO's head decreases to a WER of 1.92%. For the data recorded with the channel NAO2 the resulting WER of 1.5% shows no relevant loss in comparison to the benchmark value of 1.4%. The worst WER in this setup is reached by the channel NAO4, with 2.6%. Using the *audiolab trainset model*, the reuslting WERs equalize for all channels to 1.9-2.0%. The usage of the *seminarroom trainset model* leads to a significant reduction of performance with an average WER of 10.3%. Noticeable is, that the channel NOA4 now obtains the best WER with 8%. Table 3.1 summarizes the results for the training set recorded in the *Audio Lab*.

Acoustic Model	Average [%]	Best [%] (Channel)	Worst [%] (Channel)
standard model	13.92	13.5 (NAO5, 12)	14.7 (NAO4)
original trainset model	1.92	1.5 (NAO2, 12)	2.6 (NAO4)
audiolab trainset model	1.98	1.9 (NAO1, 3)	2 (other)
seminarroom trainset model	10.28	8 (NAO4)	12 (NAO2)

Table 3.1: WERs on the *AudioLab TrainSet* achieved with the microphones in the NAO head

The evaluation results for the *Audio Lab TestSet*, summarized in table 3.2 are virtually identical. Surprisingly, the average WER, cosnidering the twelve microphines in NAO's head, decoding the recordings in the *Audio Lab* is not obtained by the *audiolab trainset model* but with the *original trainset model*. This is due to the fact, that the *audiolab trainset model* is trained on the recordings of the microphone NAO1, while the best WER is achieved with the microphone NAO2. With an acoustic model trained on

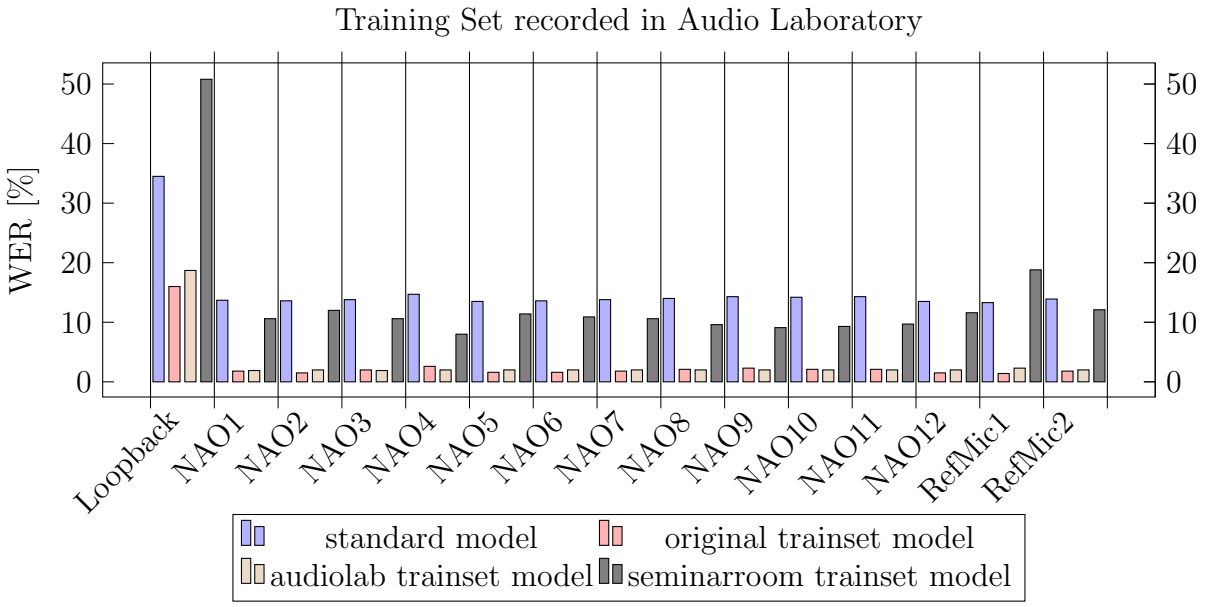

Figure 3.1: WER on the *Audio Lab TrainSet* using different acoustic models

the recordings of the microphone NAO2 the average WER on the *Audio Lab TrainSet* reduces to 1.77%. However, the best WER is still achieved with the *original trainset model*. From the resulting data it can be concluded, that the usage of the microphones in NAO's head has no relevant influence on the performance of the used ASR system. As expected, the microphone directly oriented to the loudspeaker obtains the best WER in general. Nevertheless, the WERs are very similar for all channels. The exception to this observation when using the *seminarroom trainset model* might be explained by effects of reverberation in the dataset used for the training of the acoustic model.

Acoustic Model	Average [%]	Best [%] (Channel)	Worst [%] (Channel)
standard model	13.88	13.4 (NAO5, 12)	14.7 (NAO4)
original trainset model	2.03	1.7 (NAO2, 5, 12)	2.7 (NAO4)
audiolab trainset model	1.92	1.8	2
seminarroom trainset model	10.28	8 (NAO4)	11.9 (NAO2)

Table 3.2: WERs on the *Audio Lab TestSet* achieved with the microphones in the NAO head

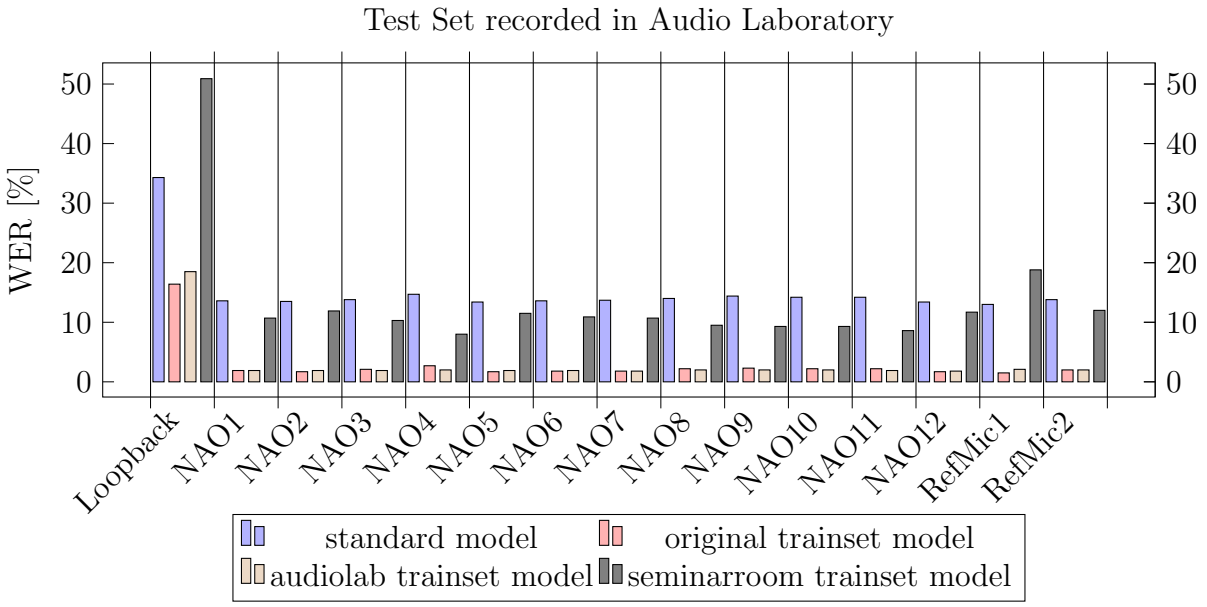

Figure 3.2: WER on the *Audio Lab TestSet* using different acoustic models

Figure 3.3 shows the resulting WERs obtained by decoding the *Seminar Room TrainSet*. The data recorded with the loopback channel reaches a WER equal to the benchmark value of 1.4% when using the *original trainset model*. With the *standard model*, the WERs of the data recorded with the microphones in the NAO head have an average value of 32.172% with a best score of 24.4% (NAO2) and a worst score of 45.2% (NAO4). Using the *original trainset model* the average value decreases to 14.26% WER. The best score in this setup is 8.6% (NAO2) and the worst score 23.6% (NAO4). Further improvements are achieved using the acoustic models trained with the recorded datasets. Table 3.3 summarizes the results for the training set recorded in the seminar room. Independent of the used acoustic model, the best score is achieved with the channel NAO2 and the worst score with the channel NAO4.

Table 3.4 summarizes the results for the *Seminar Room TestSet*, depicted in Figure 3.4. As for the data recorded in the *Audio Lab* the resulting WERs are very close to the results from the training set.

Generally, the performance of the ASR system decreases on the data recorded in the *Seminar Room* in comparison to those recorded in the *Audio Lab*. Due to reverberation

Acoustic Model	Average [%]	Best [%] (Channel)	Worst [%] (Channel)
standard model	32.17	24.4 (NAO2)	45.2 (NAO4)
original trainset model	14.26	8.6 (NAO2)	23.6 (NAO4)
audiolab trainset model	7.93	5.2 (NAO2)	13.3 (NAO4)
seminarroom trainset model	5.88	5 (NAO1, 2)	7.6 (NAO4)

Table 3.3: WERs on the *Seminar Room TrainSet* achieved with the microphones in the NAO head

Acoustic Model	Average [%]	Best [%] (Channel)	Worst [%] (Channel)
standard model	43.98	35.8 (NAO2)	54.5 (NAO4)
original trainset model	15.15	9.2 (NAO2)	23.6 (NAO4)
audiolab trainset model	9.16	6.5 (NAO2)	14.5 (NAO4)
seminarroom trainset model	7.28	6.4 (NAO2, 5)	9.4 (NAO4)

Table 3.4: WERs on the *Seminar Room TestSet* achieved with the microphones in the NAO head

effects and a likely present additional noise this behaviour was expected. Nevertheless, WERs significantly below 10% can be reached on all recorded datasets.

Table 3.5 and Figure 3.5 for the training set and Table 3.6 and Figure 3.6 provide a more detailed insight in the performance of the acoustic models trained on the recorded datasets. First it becomes clear, that the best WERs are achieved for the datasets on which the acoustic models were trained. Yet, the best achieved WER for the *Seminar Room TrainSet* is close to equal for both acoustic models. Regarding the *Audio Lab TrainSet* the *audiolab trainset model* performs much better compared to the *seminarroom trainset model*, on average as well as for the best and worst channels. Therefore, in this setup, the *audiolab trainset model* exhibits a more robust behaviour to changing environmental conditions.

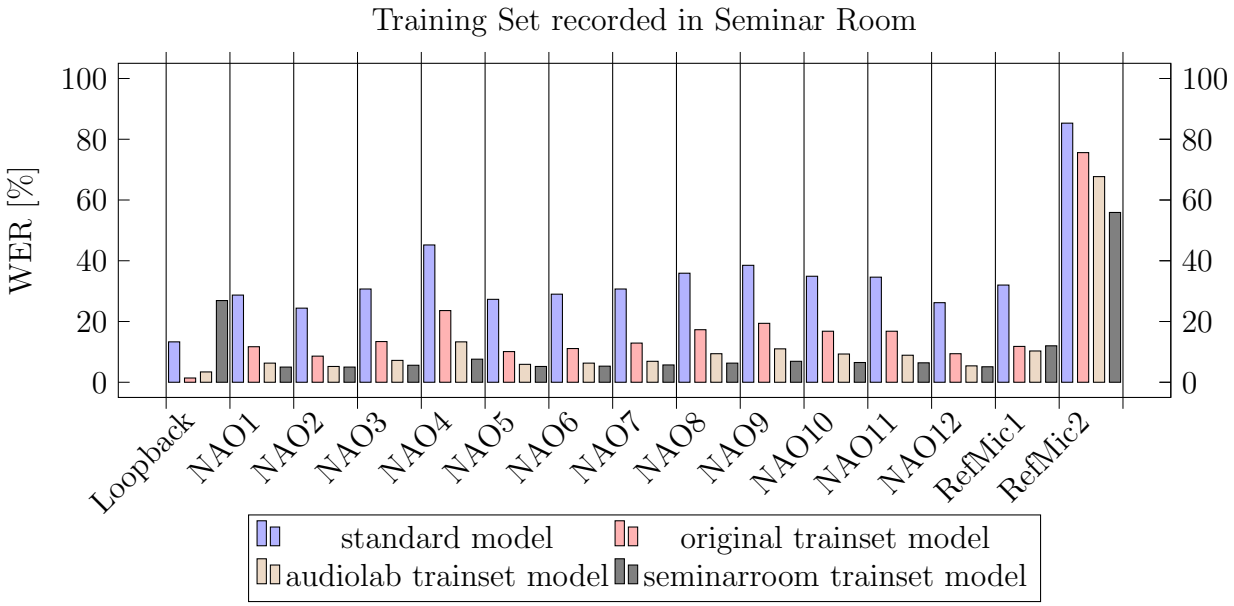

Figure 3.3: WER on the *Seminar Room TrainSet* using different acoustic models

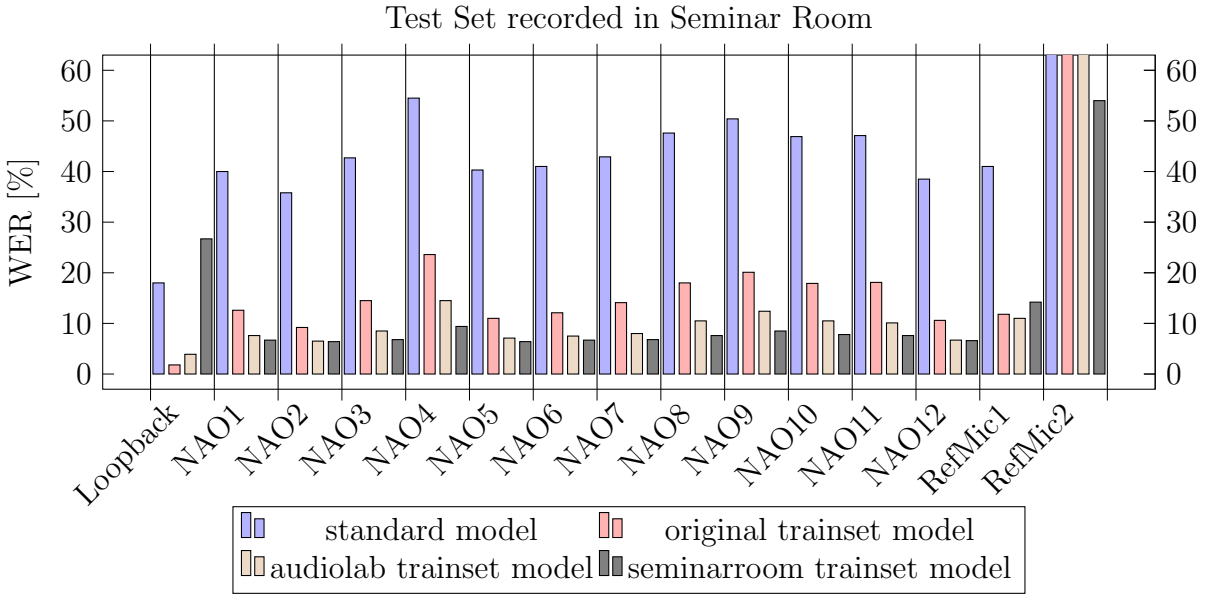

Figure 3.4: WER on the TIDIGITS test set recorded in the seminar room using different acoustic models

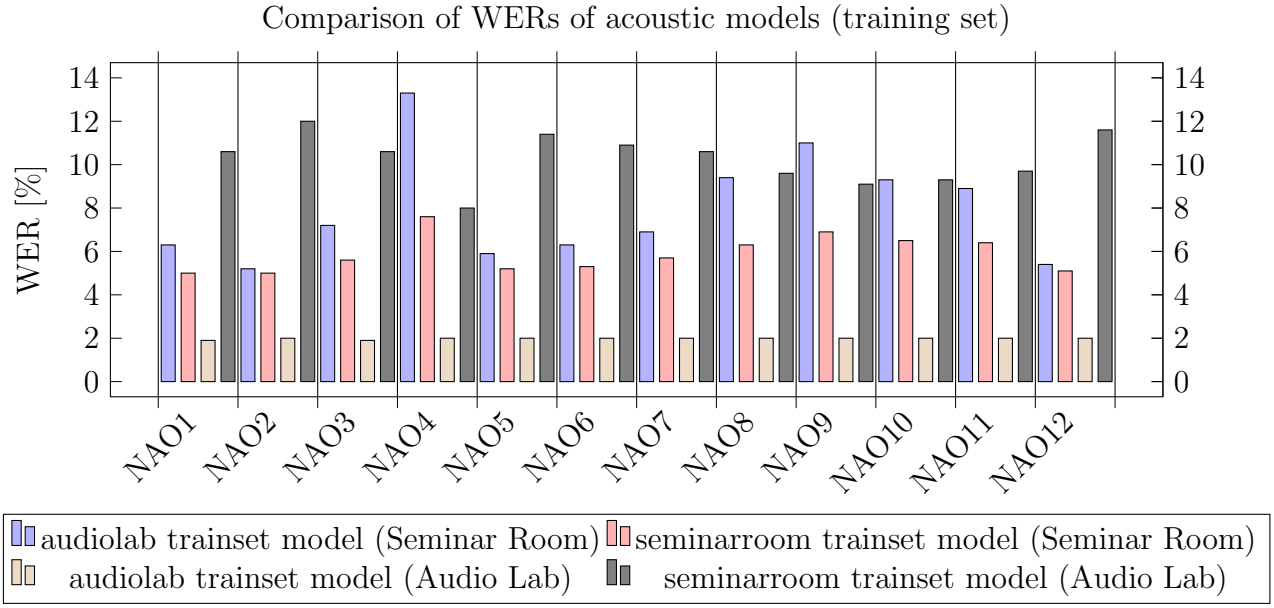

Figure 3.5: Comparison of WERs on the training set recorded in both acoustic environments using the trained acoustic models for decoding

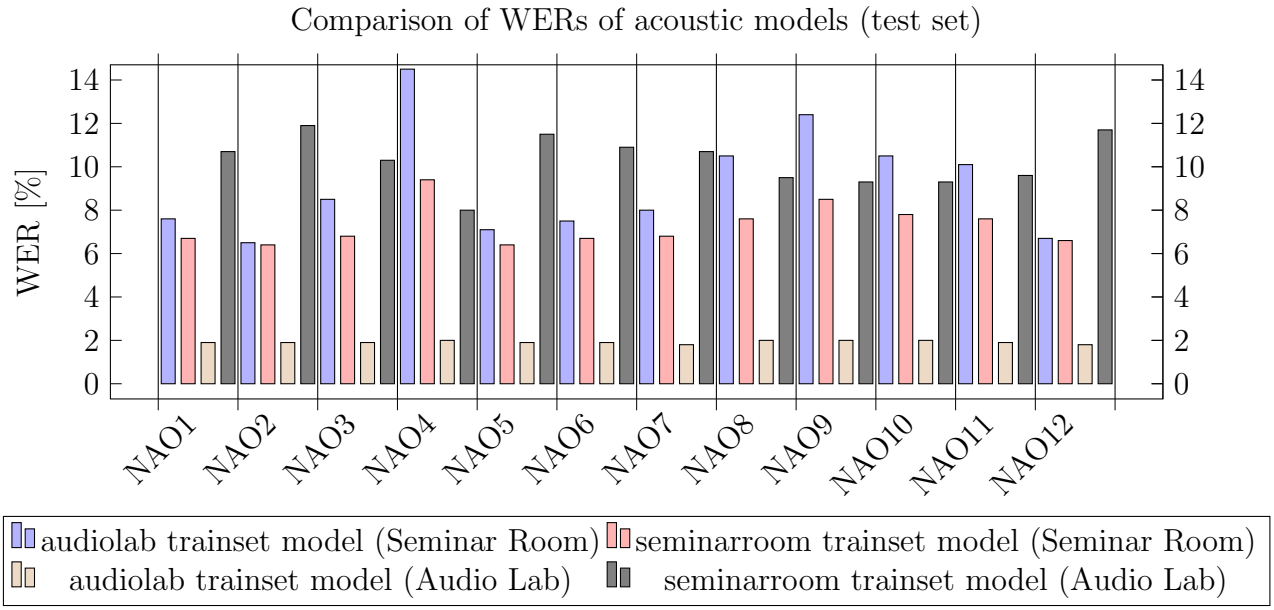

Figure 3.6: Comparison of WERs on the test set recorded in both acoustic environments using the trained acoustic models for decoding

Acoustic Model (Recording Env. TrainSet)	Average [%]	Best [%]	Worst [%]
audiolab trainset model (Seminar Room)	7.93	5.2	13.3
seminarroom trainset model (Seminar Room)	5.88	5	7.6
audiolab trainset model (Audio Lab)	1.98	1.9	2
seminarroom trainset model (Audio Lab)	10.28	8	12

Table 3.5: WERs on the recorded TIDIGITS training set achieved with the trained acoustic models

Acoustic Model (Recording Env. TestSet)	Average [%]	Best [%]	Worst [%]
audiolab trainset model (Seminar Room)	9.16	6.5	14.5
seminarroom trainset model (Seminar Room)	7.28	6.4	9.4
audiolab trainset model (Audio Lab)	1.92	1.8	2
seminarroom trainset model (Audio Lab)	10.28	8	11.9

Table 3.6: WERs on the recorded TIDIGITS test set achieved with the trained acoustic models

Chapter 4

Conclusion

In this work, the offline ASR system Pocketsphinx, based on HMMs and GMMs, has been evaluated for a small-vocabulary speech recognition task. Therefore, a word-based phonetic dictionary was used to train several word-dependent acoustic models, based on the TIDIGITS corpus. The corpus was recorded in two different acoustic environments. In comparison to the *standard model* provided by Pocketsphinx, the self-trained acoustic models resulted in a significant improvement of the WER. For the datasets recorded in the acoustic environments as well as the original dataset WERs below 10% were reached, with a lower bound of 1.4%. From the evaluation of the dataset recorded in the *Audio Lab* it could be concluded that the recording by the microphones in NAO's head has almost no influence on the resulting WER. Furthermore, also for the data recorded in the reverberant environment very low WER were reached, even without any preprocessing of the recorded data.

As the evaluations with the word-dependent phonetic dictionary yielded satisfying results on the TIDIGITS dataset, this approach might be used for a key word search in continuous speech. Such a setup allows for the detection of specific commands and is therefore interesting for the real-time demonstrator developed during the EU-FP7 Project EARS. Pocketsphinx offers the possibility to configure a list of keyphrases to be identified in continuous speech. An individual detection threshold can be assigned

to each keyphrase.

For a use as a command interface it is further desirable to detect command like words such as "Yes", "No" or "Off". The used TIDIGITS dataset solely contains digits as words. Therefore, an alternative dataset should be used for the training of an acoustic model usable for a command interface. For instance, Google's freely available *Speech Commands Dataset* includes 30 short words, of which some are usable as commands [Goo].

Appendix A

List of Abbreviations

ASR	Automatic Speech Recognition
GMM	Gaussian Mixture Model
GUI	Graphical User Interface
HMM	Hidden Markov Model
WER	Word Error Rate

Appendix B

Software

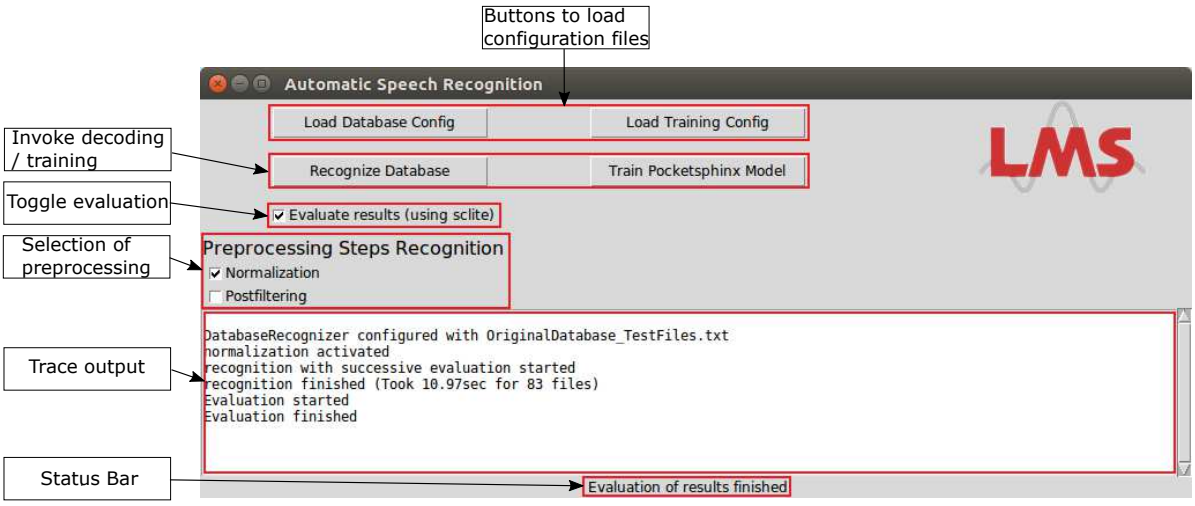

Figure B.1: Graphical User Interface of developed application

B.1 Used Libraries and Installation

Name	Version	Source
sphinxbase	5prealpha	https://github.com/cmusphinx/sphinxbase
pocketsphinx	5prealpha	https://github.com/cmusphinx/pocketsphinx
sclite	2.4.10	http://www.openslr.org/4/
sphinxtrain	5prealpha	https://github.com/cmusphinx/sphinxtrain

Table B.1: Summary of external (non-Python) software used

Module	Version
SciPy	
NumPy	
pocketsphinx	
num2words	
audioop	
wave	
PIL	
tkinter	

Table B.2: Summary of external Python modules used

B.2 Operating Instructions

The software can be used to decode audio files using an existing acoustic model or for training a new acoustic model with Pocketsphinx. For both functionalities a configuration file needs to be loaded first (using the GUI buttons "Load Database Config" or "Load Training Buttons"). The necessary content of these files can be seen in section B.2.1 and B.2.2, respectively. The recorded database has a flat structure with the filename-format *flag1_speaker_label_flag2.wav*. The procedure for extracting the necessary metadata as well as building the *.fileid* and the *.transcription*-file needed by Pocketsphinx is contained in the file *TIDIGITS_Wrapper.py*. The respective functions are passed as variables to *self.databaseModel.recognizePocketsphinxDatabase* and *self.pocketSphinxTrainer.train* within the *Controller* (ASR_Controller.py) of the program. Therefore, they can be easily exchanged by self-written functions appropriate for the repsective database.

B.2.1 Configuration file for decoding of TIDIGITS database

```
SCLITE_PATH=path to 'bin' folder of sctk
DATASET_PATH=path to dataset which chould be decoded
EVALUATION_PATH=path to folder in which evaluation results should be stored
MODEL_PATH=path to root folder of pocketsphinx model to be used for decoding
HMM=relative path to acoustic model (based on MODEL_PATH)
LM=relative path to language model + file name
DICT=relative path to phonetic dictionary + file name
```

B.2.2 Configuration file for training of acoustic model

```
MODEL_NAME=
DATASET_PATH=path to dataset which should be used for training
REFERENCE_MODEL_PATH=path to reference model used as basic setup for training
MODEL_PATH=path to store model
LM=relative path to language model + file name
DICT=relative path to phonetic dictionary + file name
```

B.3 Software Architecture

The application is designed in a classical Model-View-Controller manner. With this, a clear separation of functionality is achieved, with the single parts being responsible for:
- View: Display changes within the model and notify controller of user interactions
- Controller: Handle user interactions and send the respective commands to the model. Steer and observe the execution of commands and notify view about relevant changes in the model.
- Model: Execute code representic the business logic (i.e. training of acoustic models or recognition in this case) as requested by controller.

Beside this elements, there is a *Configuration.py*-file containing globally relevant constants.

B.4 Suggestions for Future Extension

B.4.1 Add active GUI element and bind new command

To add a new element to the GUI the file *ASR_View.py* has to be edited. The new element should be added witihin the *__init__*-method of the class *Application*. The separation of active and passive GUI elemtens should be kept.

In the following, adding a new element is explained on the example of a new *Checkbutton* for a preprocessing step. Lisiting B.1 shows the relevant code section, defining the content and shape of the *postprocessing*-section in the GUI. To add a new *Checkbutton*, the code lines declaring the *tk.IntVar*-variable and the *tk.Checkbutton*-variable can be copied. Change the variable names and edit the arguments of the declaration of the *tk.Checkbutton*-variable. The line below places the *Checkbutton* in the GUI. For each new element, the *row*- and *column*-variable need to be adapted. In the case of a new preprocessing *Checkbutton* it is sufficient to increase the *column*-value by one.

```
self.preprocessingLabel = tk.Label(self, text='Preprocessing Steps Recognition', font
    ='bold')
        self.preprocessingLabel.grid(row=3, column=0, sticky='w')

        self.normalization = tk.IntVar(value=0)
        self.normalizationCheckbox = tk.Checkbutton(self, text='Normalization',
            variable=self.normalization, height=1)
        self.normalizationCheckbox.grid(row=4, column=0, sticky='w')

        self.postfiltering = tk.IntVar(value=0)
        self.postfilteringCheckbox = tk.Checkbutton(self, text='Postfiltering',
            variable=self.postfiltering, height=1)
        self.postfilteringCheckbox.grid(row=5, column=0, sticky='w')
```

Listing B.1: Postprocessing GUI Elements

To add a functionality to the newly introduced element, a command needs to be binded to it. For the preprocessing case, the last line in Listing B.2 can be copied. Change the last two arguments to the needed values. Now, once the element is pressed

by the user, the respective command is called, and invokes the execution of the function *ecxecuteCommand* of the object assigned to the *controller*-variable (the assignment is done in *main.py* and does not need to be adapted).

```
def bindCommands(self):
        self.recognizeButton.config(command=lambda: self.controller.executeCommand('
            RecognizeDatabase'))
        self.configurationDatabaseButton.config(command=lambda: self.
            selectDatabaseConfiguration())
        self.trainingButton.config(command=lambda: self.controller.executeCommand('
            TrainPocketsphinxModel'))
        self.configurationTrainingButton.config(command=lambda: self.
            selectTrainingConfiguration())
        self.normalizationCheckbox.config(command=lambda: self.controller.
            executeCommand('ActivatePreprocessingDatabase', 'normalization', self.
            normalization.get()))
        self.postfilteringCheckbox.config(command=lambda: self.controller.
            executeCommand('ActivatePreprocessingDatabase', 'postfiltering', self.
            postfiltering.get()))
```

Listing B.2: Bind commands to GUI elements

Listing B.3 shows the called code. The *controller* sends the command to activate or deactivate the preprocessing step to *databaseModel* where it is processed further. The final adpation needs to be made in the file *Preprocess_AudioData.py*. Here, within the *__init__*-method a dictionary is declared (*self.sequencePreprocessingSteps*) assigning the names of the preprocessing steps to objects which execute the logic of the respective preprocessor. The newly introduced preprocessing step can be added here. Finally, a new class has to be implemented containing the logic of the preprocessing step. The class *NormalizationPreprocessor* can be used as a template (listing B.4).

```
    def executeCommand(self, command, key=None, value=None):
        if command == 'RecognizeDatabase':
            self.databaseModel.recognizePocketsphinxDatabase(TIDIGITS_Wrapper.
                tidigitsExtractMetadata)
        elif command == 'LoadDatabaseConfiguration':
            try:
                self.databaseModel.loadDatabaseConfigurations(self.view.
                    getDatabaseConfigurationPath())
```

```
            except ValueError as e:
                self.updateMessage(str(e.args[0]))
        elif command == 'LoadTrainingConfiguration':
            try:
                self.pocketSphinxTrainer.loadTrainingConfigurations(self.view.
                    getTrainingConfigurationPath())
            except ValueError as e:
                self.updateMessage(str(e.args[0]))
        elif command == 'TrainPocketsphinxModel':
            self.pocketSphinxTrainer.train(TIDIGITS_Wrapper.
                tidigitsGenerateFileidsFile, TIDIGITS_Wrapper.
                tidigitsGenerateTranscriptionFile)
        elif command == 'ActivatePreprocessingDatabase':
            if value is not None and key is not None:
                self.databaseModel.activatePreprocessingStep(key, value)
```

Listing B.3: executeCommand() of controller class

```
class NormalizationPreprocessor(PreprocessingStep):
    def preprocess(self, audioData):
        _dataType = audioData.dtype
        _limitValue = np.iinfo(_dataType).max
        _maxValue = abs(max(np.max(audioData), np.min(audioData), key=abs))
        if _maxValue > 0:
            audioData = np.divide(audioData, _maxValue/_limitValue)
        return audioData.astype(_dataType)
```

Listing B.4: Class NormalizationPreprocessor

Appendix C

Acoustic Model

C.1 Word Based Phonetic Dictionary

eight	EY_eight T_eight
five	F_five AY_five V_five
four	F_four OW_four R_four
nine	N_nine AY_nine N_nine_2
oh	OW_oh
one	W_one AX_one N_one
seven	S_seven EH_seven V_seven E_seven N_seven
six	S_six I_six K_six S_six_2
three	TH_three R_three II_three
two	T_two OO_two
zero	Z_zero II_zero R_zero OW_zero

C.2 Phoneset file

AX_one

AY_five

AY_nine

EH_seven

EY_eight

E_seven

F_five

F_four

II_three

II_zero

I_six

K_six

N_nine

N_nine_2

N_one

N_seven

OO_two

OW_four

OW_oh

OW_zero

R_four

R_three

R_zero

S_seven

S_six

S_six_2

TH_three

T_eight

T_two

V_five

V_seven

W_one

Z_zero

SIL

C.3 Filler dictionary

⟨s⟩ SIL

⟨sil⟩ SIL

⟨/s⟩ SIL

List of Figures

List of Tables

Bibliography

[CMU] Cmusphinx. https://cmusphinx.github.io. Accessed: 2018-05-16.

[Goo] Launching the speech commands dataset. https://ai.googleblog.com/2017/08/launching-speech-commands-dataset.html. Accessed: 2018-05-16.

[YD15] Dong Yo and Li Deng. *Automatic Speech Recognition*. Springer, 2015.