Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Vetle Haflan

Noise Robustness in Small-Vocabulary Speech Recognition

Master's thesis in Electronics
Supervisor: Torbjørn Karl Svendsen
March 2019

Norwegian University of Science and Technology

Vetle Haflan

Noise Robustness in Small-Vocabulary Speech Recognition

Master's thesis in Electronics
Supervisor: Torbjørn Karl Svendsen
March 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Sammendrag

Denne masteroppgaven omhandler små-vokabular talegjenkjenning, og mer spesifikt støyrobusthet i systemer designet for dette formål. Tradisjonelle og moderne gjenkjenningssystemer har blitt trent på relativt store mengder norsk taledata og deres ytelse har blitt evaluert ved hjelp av mindre mengder støyete taledata. De moderne nevronett-baserte systemene viste seg å ikke være trenbare uten betydelig med beregningsressurser, men det tradisjonelle system ble brukt med suksess. Evaluering av det tradisjonelle systemet indikerte at dets ytelse er brukbar for veldig ren taledata, men at den fort minker for støyete taledata hvor signal-støy-forholdet er mindre enn 30 dB.

Abstract

This thesis deals with the task of small-vocabulary speech recognition, and more specifically noise robustness in systems designed for this task. Traditional and modern speech recognition systems have been trained on a relatively large amount of Norwegian speech data and had their performance evaluated on small sets of noisy speech data. The modern, neural network based systems proved infeasible to train without significant computational resources, while the traditional system was successfully employed. Evaluation of the traditional system indicated that its performance is sufficient for recognition of very clean speech data, but quickly deteriorates for data corrupted by noise with a signal-to-noise ratio below 30 dB.

Preface

This thesis is the final project of a master's degree program in Electronics at NTNU, Trondheim. The thesis work was carried out between September 2018 and March 2019.

The thesis problem was defined in part by the company *LuxSave AS*, which wanted a to implement a voice controlled interface for their website. Although the main interest of LuxSave was to get a working voice control system, the thesis problem is a more academic one, and LuxSave has not taken part in the supervision of the project. I found it a bit challenging to balance the academic and practical interests, and I realized relatively early during my work that LuxSave might be more interested in just using a commercial system like Google Cloud Speech instead of trying to implement their own.

The intended audience for this thesis is students of electronic engineering with a specialization in signal processing. I therefore assume that the reader has a basic theoretical understanding of signal processing, but that they might not be familiar with the basic theory of speech recognition. A brief explanation of speech recognition basics is therefore included in addition to more specific theory. Both of these are included in order to put the experimental work in context.

I would like to thank my supervisors, Abdolreza Sabzi Shahrebabaki and Ali Shariq Imran for lots of help with my work, and Torbjørn Svendsen for providing a sensibly organized version of the database I used for system training. I also want to thank Negar Olfati for helping during supervision despite her not being a supervisor officially. Last, I want to thank all the people who helped me collect the speech data I needed for testing / evaluations.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Objective . . . 1
1.3 Thesis Overview . . . 2

2 Theory **3**
2.1 SR System Overview . . . 3
2.1.1 Feature extraction . . . 3
2.1.2 Speech Units . . . 5
2.1.3 Language model . . . 6
2.1.4 Acoustic model and Decoder . . . 6
2.2 Noise . . . 7
2.3 Traditional Speech Recognition . . . 8
2.3.1 Hidden Markov Models . . . 8
2.3.2 Gaussian Mixture Models . . . 8
2.3.3 Environmental Robustness . . . 9
2.4 Neural Networks . . . 10
2.4.1 Basic Theory . . . 10
2.4.2 Recurrent Neural Networks . . . 12
2.4.3 Connectionist Temporal Classification . . . 12

3 Experiments **17**
3.1 Method . . . 17
3.1.1 Tools . . . 17
3.1.2 Choice of Techniques . . . 17
3.1.3 Training databases . . . 17
3.1.4 Testing database . . . 18
3.1.5 Noise Simulation . . . 19
3.2 GMM/HMM Implementation . . . 19
3.2.1 Data preparation . . . 20
3.2.2 Cross-validation on `luxdb` . . . 20
3.2.3 Benchmark models . . . 20
3.2.4 Improving Robustness . . . 21
3.3 GMM/HMM Evaluation . . . 22
3.3.1 Language Model . . . 22
3.4 CTC-trained RNN . . . 24
3.4.1 Implementation . . . 24
3.4.2 Evaluation . . . 25
3.5 Google Cloud Speech Evaluation . . . 26
3.5.1 Models . . . 26
3.5.2 Phrase hints . . . 26

4 Results **29**
4.1 Measurements of Performance . . . 29
4.2 GMM/HMM . . . 30
4.2.1 Cross-validation . . . 30
4.2.2 Benchmark models . . . 30
4.2.3 Adaptations . . . 31
4.2.4 Multistyle . . . 31
4.2.5 Language Model WERs . . . 33
4.2.6 Out-of-grammar classifications . . . 33
4.3 Google Cloud Speech . . . 36

5 Discussion **37**
5.1 GMM / HMM . . . 37
5.1.1 Benchmarks . . . 37
5.1.2 Adaptation . . . 37
5.1.3 Multistyle Training . . . 37
5.1.4 Improvement Potential . . . 37
5.1.5 Sphinx4 vs. Pocketsphinx . . . 38
5.2 Google Cloud Speech . . . 38
5.3 Production Considerations . . . 38
5.4 Limitations . . . 38
5.4.1 Testing Database . . . 40
5.4.2 Evaluator Problems . . . 40
5.4.3 Other . . . 40
5.5 Future work . . . 40
5.5.1 General Evaluation of Sphinx . . . 40
5.5.2 Recurrent Neural Networks . . . 41
5.5.3 Denoising . . . 41
5.6 Resource Intensity . . . 41

6 Conclusion **43**

Bibliography **45**

A Detailed Results **49**
A.1 Complete Tables of Results . . . 49
A.2 Confusion Matrices . . . 50
A.3 F_1 scores . . . 53

B Data Preparation Scripts **57**
B.1 Audio Processing . . . 57

C Evaluation Code **59**
C.1 Java Evaluator Source Code . . . 59
C.2 Shell Scripts . . . 64

D Setup Scripts **67**
D.1 CMU Sphinxtrain Setup . . . 67
D.2 WarpCTC Conda Setup . . . 68

Chapter 1

Introduction

Automatic speech recognition has seen significant performance improvements in the recent years. With the emergence of more powerful computers capable of training deep neural networks, machine learning in general is moving towards networks trained on vast amounts of training data rather than carefully engineered systems that work with smaller amounts. Although deep neural networks seem very promising for *general-purpose* speech recognition, there might be cases where such architectures are overkill. One case like this is the field of domain specific, *small-vocabulary* speech recognition, which is the topic of this thesis.

1.1 Motivation

This thesis is written for *LuxSave AS*[1], a company that works with smart control of street lights. LuxSave runs a web interface that lets technicians control light poles by sending commands to controllers connected to the light poles. The website currently uses a standard touch interface that looks like the one seen in figure 1.1. This is not always optimal, as a common use case for the system is by technicians who get their hands dirty. In these cases, a *voice controlled* system could be preferable, and that is the overall motivation for this thesis. Such a voice controlled system would ideally have to satisfy several criteria. It should be able to handle multiple languages, accents and dialects, as LuxSave is not limited to a single country nor region. Light poles are often placed in areas with much traffic, which means that noise will be a significant issue that the system has to deal with. Although the thesis uses the LuxSave case as a basis, it tries to stay general when possible, writing with startups and early-stage companies in mind. As startup systems are often rapidly evolving, there is a high probability of new commands emerging after the initial vocabulary specification, and the company might not have the resources to train a domain-specific system. As such, yet another criteria for the system is that it should be flexible enough to expand the grammar without resource intensive redesign.

1.2 Objective

To limit the scope of the thesis, the focus here will be on noise robustness in a generalizable system like this. The official goal of the thesis is the following:

> Evaluate the noise robustness of different classification systems and find what methods of speech classification give the most favorable trade-offs between noise robustness and computational resource intensity.

To limit the scope further, the thesis will not touch on *wake-up-word detection*, which is another speech recognition task [18]. Instead, it is simply assumed that the system uses push-to-talk to record speech.

[1] http://luxsave.com

Figure 1.1: The LuxSave web interface showing the `Temporary control` menu.

1.3 Thesis Overview

Chapter 2 explains fundamental speech recognition theory that is needed to understand the chapters that follow. This entails a short description of the components normally found in a speech recognition system, what they are and why they are needed, and a bit more detailed theory about acoustic models, which is the component of focus in this thesis. Chapter 3 describes the choice of methodology and tools, which systems to evaluate, and how system evaluations were set up and executed. Chapter 4 contains the results of evaluations and some notes on how to interpret them, while Chapter 5 discusses them in more detail.

Chapter 2

Theory

This chapter contains some of the basic speech recognition theory that is needed to understand the systems that will be experimented on in the thesis. It will start in section 2.1 with a high level explanation of components that are common between most speech recognition systems. Section 2.3 then tries to explain the theory behind techniques used in traditional speech recognition techniques. At last, section 2.4 explains a more modern alternative.

2.1 SR System Overview

The common goal of all speech recognition (SR) systems is to interpret the intended meaning, $\mathbf{y}$ of of an input audio signal $\mathbf{x}$. How a system achieves this can vary, but most traditional systems look similar to the one in figure 2.1. The components and their purpose are as follows [14].

Feature extraction extracts useful feature data from the raw audio data.

Acoustic model models a statistical relationship between observed data and speech *units* (section 2.1.2) without any knowledge of the syntax or semantics of the language spoken.

Language model describes which combinations of speech units are allowed for the given language, i.e. which words are possible, and which words are likely to co-occur.

Pronunciation model is used for connecting the acoustic model and the language model.

Decoder Performs the inference of the most likely transcription based on the information available.

2.1.1 Feature extraction

A sound wave can be viewed as continuous one-dimensional signal or, when digitized, a single vector of values. Such a vector is hereby referred to as *raw audio*. Although some modern techniques are able to work on a raw speech audio directly [33], traditional speech recognition systems need to perform some sort of pre-processing of the audio signal. The reason is that raw audio contains too much data for the acoustic model to learn from. *Feature extraction*, one of the most important parts of pre-processing, leads to a more compact representation. The goal of feature extraction is to reduce the total amount of data while retaining the properties that are most useful for the recognition task at hand. There are multiple methods of extracting features from speech, some of which are illustrated in figure 2.3. These will be discussed here.

Short-time stationarity

Speech can be considered a *short-time stationary* process, meaning its statistical properties do not change over a certain time. This assumption is important for speech processing systems that rely on statistical modeling of the acoustic properties. By assuming that audio is short-time stationary, a speech signal can be modeled as a sequence of such stationary processes.

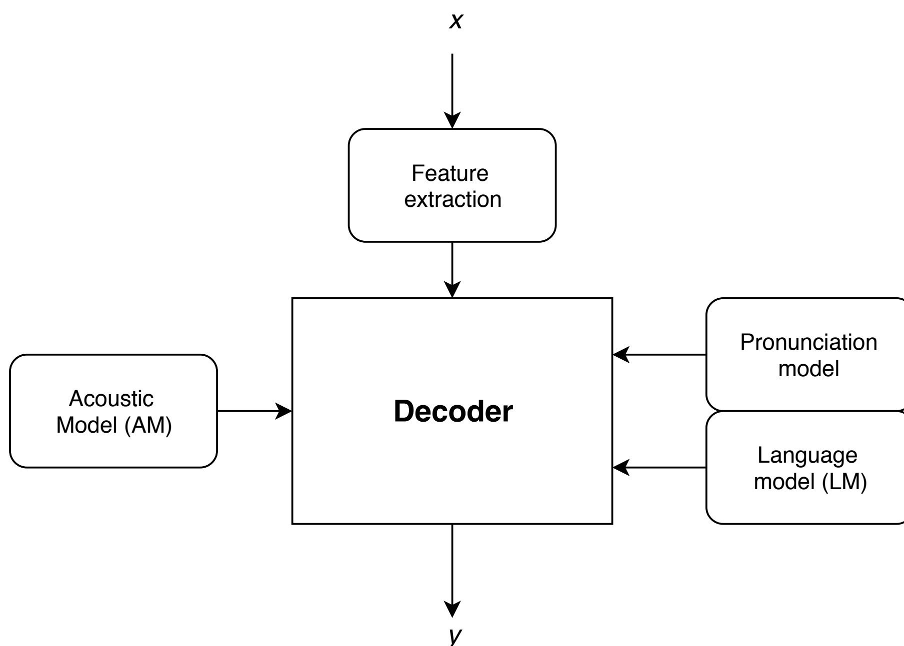


Figure 2.1: A standard traditional speech recognition system.

Short-Time Fourier Transform

The simplest form of pre-processing is a simple transformation from the time domain to the frequency domain, i.e. a Fourier transform. Short-time Fourier Transform, or *STFT* is, the application of the discrete Fourier transform (DFT) to short, often overlapping segments of audio. This produces a frequency spectrum for each segment, here called $\mathbf{o}_n$. Concatenation of all resulting frequency spectra is what makes up a *spectrogram*. Figure 2.2 illustrates the STFT with the use of a *window*. This is often done in order to avoid unwanted artifacts resulting from the Fourier transform of a finite signal. It is common to use a window length of of 20-30 ms with an overlap of about 10 ms, as this has been found to work well. A more thorough explanation of windowing can be found in [16].

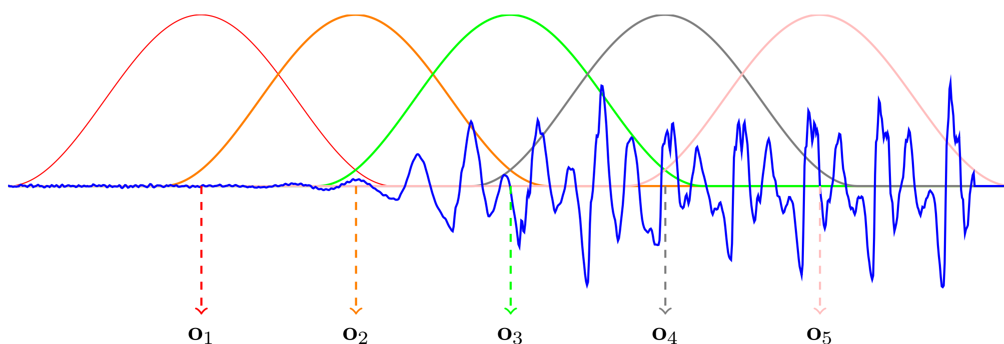


Figure 2.2: The STFT process, using a window function on overlapping audio segments to produce a sequence of frequency spectra.

Mel-Frequency Cepstral Coefficients

While frequency domain representations are often more useful than raw audio for recognition tasks, many systems have trouble using these too. Speech spectra still contain a lot of information that the system might not be able to take advantage of. This is especially true for traditional methods of

speech recognition, which are generally not that good with high dimensions of data. For systems like that, an even more compact feature representation is desired. One such representation is *Mel-frequency cepstral coefficients*, or MFCC.

MFCC is based on the human auditory system, and is probably the most commonly used feature type. In addition to DFT, the process of calculating the MFCCs from an audio segment involves filtering the segment with a bank of triangular filters based on the *Mel scale* [29], and performing a discrete cosine transform (DCT) on the resulting filterbank energies. DCT helps decorrelate the feature values, which in turn makes the feature representation more compact. It has been empirically found that the 13 first cepstral coefficients are often sufficient for speech recognition. To further improve accuracy, first- and second-order *delta coefficients* are sometimes used in addition to plain cepstral coefficients. Delta coefficients are useful because they capture temporal changes in the spectra, something that is important in speech recognition [14].

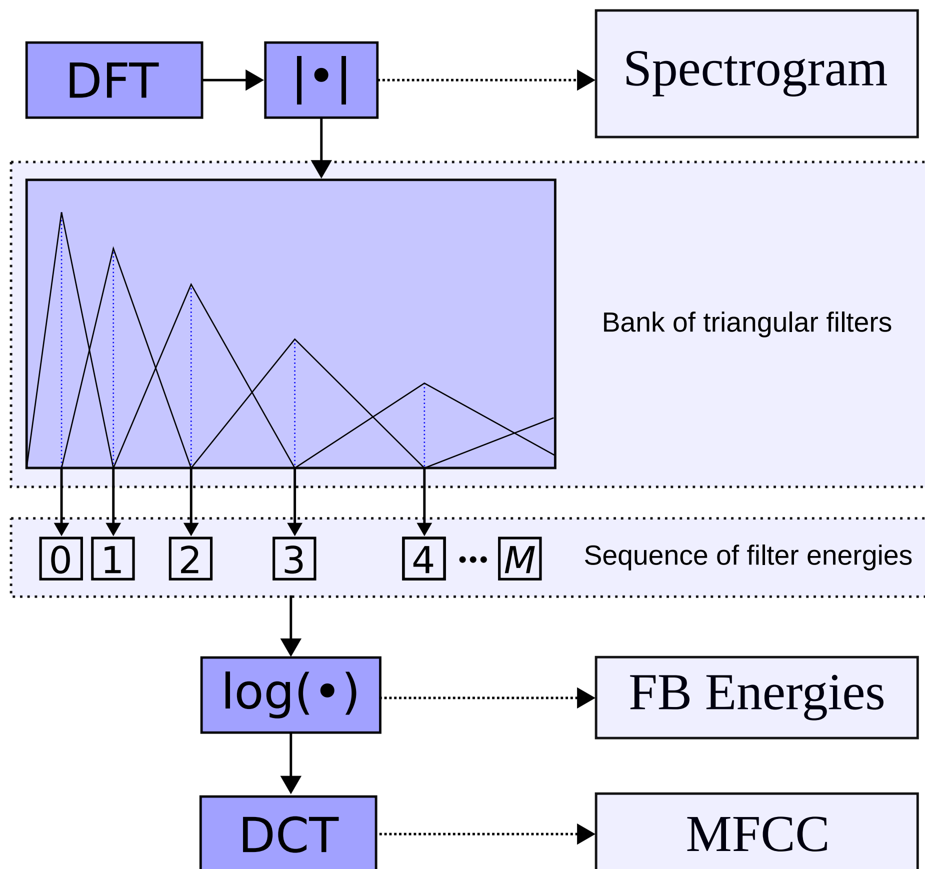


Figure 2.3: Overview over some common speech feature representations [15, modified]

2.1.2 Speech Units

There are multiple ways of modeling speech. A choice that must be made when designing a SR system is what should be the smallest unit of speech. The most common units are *whole words* and *phonemes*. Different units have different properties, and the best choice of units depends on what properties are desired most in the task the system will be used for. The three most important are accuracy, trainability and generalizability. *Trainability* is mostly decided by how many classes there are to train compared to how much data is available for each class. *Generalizability* means whether the system can be used for cases it has not been trained for, e.g. words that were not included in the training database.

Whole words

Traditionally, whole-word models have been the go-to choice for small-vocabulary recognition. Systems that are trained to recognize a small set of utterances do not need generalizable models, because they will be specialized to recognize only those utterances. By training a system on whole words, it will learn the phonetic co-articulation within each word, something that improves recognition performance. Whole-word models are therefore accurate if enough samples are available for each word to be trained [14]. A consequence of this is that a lot of task-specific data must be acquired in order to train an accurate model. If the vocabulary is to be changed or extended, even more new data must be collected. In other words, whole-word models are not generalizable.

Phonemes

A *phoneme* is thought of as the minimal acoustic unit that can be used as a building block for realizing the pronunciation of bigger units, like words. For general-purpose speech recognition, phoneme models are the most common choice because of their generalizability. When training a model for the task of general dictation, it is infeasible to gather samples of all possible words and utterances. Gathering data samples of smaller units, like phonemes, is much easier, because all languages have fewer phonemes than words. With a phoneme model, the SR system can be trained on utterances that may not match the ones it will be used for. The acoustic model will then learn the characteristics of each phoneme, and with help from the language model (section 2.1.3) be able to decode words that were not in the training dataset.

A weakness of this kind of model is that it assumes independency between consecutive phonemes. This is not the case in realistic speech. Smooth transitions between phonemes cause their realization to depend on their context, which leads to many phoneme models having poor accuracy. A compromise between whole-word models and such context-independent phoneme models are *context-dependent* phoneme models. These use *allophones* - different context-dependent realizations of the same phoneme, and therefore offers a trade-off between trainability and accuracy. Trainability will be worse because there are more allophones to train, but accuracy will be higher because of more accurate speech unit models [14, 17].

2.1.3 Language model

Language models (LMs) assign probabilities to sequences of words, i.e. they contain information about which words are likely to co-occur. One of the simplest forms of LMs is the n-gram. N-grams give the probability of a word occurring as a function of the $n-1$ previous words, for instance for $n = 3$ (a *tri*-gram), the probability of the word *there* given the previous words *are* and *you* is $P(\text{"there"} \mid \text{"are"}, \text{"you"})$. Using a large database of text in the language desired, the n-gram probabilities can be calculated simply by observing frequencies of different word sequences. More about this can be found in [17].

Pronunciation Model

A pronunciation model, also referred to as a phonetic dictionary or *lexicon*, is needed because the *graphemes* that build a word, i.e. the letters and characters, seldom describe the exact pronunciation of that word. By describing every valid word in terms of phonemes instead, the model to be trained will find and learn patterns much more easily. One example is the word *recipe*. While the correct phonetic transcription is /'rɛsəpi/, a system that has never seen the word and does not have a pronunciation model might expect it to be pronounced /ˌriˈsaɪp/, sounding more similar to the word *recite* [17].

2.1.4 Acoustic model and Decoder

The remaining components are arguably the most important parts of the SR system. The acoustic model, as mentioned, builds a statistical relationship between acoustic features and speech units. The decoder uses the information available from all the other components to infer the most likely transcription of the input signal [14]. While the other components discussed above will not change

much during the following experiments, different acoustic models and decoders will be evaluated. The next chapters will therefore be dedicated to some of the techniques that can be used for such modeling. The theory will be discussed in the context of the three basic problems of interest usually addressed when talking about hidden Markov models (chapter 2.3). The problems are roughly as follows [14].

Evaluation - What is the probability $P(\mathbf{X} \mid \mathbf{\Phi})$ that the model $\mathbf{\Phi}$ generates the observed output sequence $\mathbf{X}$?

Decoding - What character sequence $\mathbf{S}$ is most likely to produce the observed output $\mathbf{X}$?

Learning - How can the model parameters be adjusted to maximize the joint probability $\prod_{\boldsymbol{X}} P\left(\boldsymbol{X} \mid \mathbf{\Phi}\right)$?

2.2 Noise

To represent and measure the level of noise, it is common to use a *signal to noise ratio*, or *SNR*. This is usually given in decibel format:

$$\mathrm{SNR}_{\mathrm{dB}} = 10 \log \frac{P_{\mathrm{sig}}}{P_{\mathrm{noise}}}, \tag{2.1}$$

where P_{sig} and P_{noise} are signal power and noise power respectively. These powers can be estimated by calculating the empirical variance of the signals, which for an arbitrary signal x is given by.

$$P_x = \sigma_x^2 = \frac{1}{N} \sum_{n=0}^{N-1} x^2[n]. \tag{2.2}$$

This assumes that the signal, $x[n]$, has an expectation value of $\mu_x = 0$, which usually holds for audio signals.

2.3 Traditional Speech Recognition

This chapter tries to explain the theory behind traditional SR systems briefly. As *traditional* is an ambiguous term, it is defined here as a system that uses *hidden Markov models* and *Gaussian mixture models* for acoustic modeling and decoding.

2.3.1 Hidden Markov Models

Hidden Markov Models (HMMs) have for a long time been the go-to technique for modeling sequences in speech recognition. This subsection tries to give a short overview of how they are used. To explain HMMs, a short explanation of *Markov chains* is necessary first. Markov chains model sequences of *observable states*. A *first-order* Markov chain, which is the most commonly used, is a chain where the probability of a given state only depends on the previous state. This is the *Markov assumption*, and it allows a simple form of temporal modeling. First-order Markov chains are described by a set of state *transition probabilities*, represented by a matrix $\mathbf{A} = \{a_{ij}\}$ where element a_{ij} is the probability of transitioning to state j from state i. In addition to $\mathbf{A}$, a vector $\boldsymbol{\pi}$ of *initial probabilities* is needed to represent the probability of the first state in the chain. A Markov chain is completely defined by the parameters $\boldsymbol{\Phi} = (\mathbf{A}, \boldsymbol{\pi})$.

While Markov chains are useful when the states of interest are directly observable, this is not the case in speech recognition. The chain of states in SR is usually a sequence of speech units, but the observable data is a sequence of speech feature vectors, like those explained in subsection 2.1.1. This is where *hidden* Markov models (HMMs) become necessary. HMMs are Markov models with hidden states, where each state is said to emit an observable output from a certain probabilistic function. In addition to the Markov assumption, HMMs also assume *output independence*, i.e. the probability (distribution) of an observable output depends *only* on the state that emits it, not on previous states [14].

HMMs can be used to model phonemes or words directly, and the number of models and hidden states per model is highly dependent on the kind of model used. Figure 2.4 attempts to illustrate the use of HMMs for phoneme-level modeling. In models like this, one HMM is used to model each *phoneme* that is available, for instance using three states for each phoneme. Three-states lets the HMM model the start, middle and end of the phoneme. Full words can then be constructed by concatenating phoneme-level HMMs [14], as demonstrated in the figure.

In speech recognition, the observed data is, as mentioned, a sequence of speech feature vectors, i.e. a matrix $\mathbf{O}$. *Decoding*, as mentioned in chapter 2, is in this case to process of inferring the state sequence $\mathbf{S}$ that is most likely to produce $\mathbf{O}$ given the model. In other words, the goal is to find

$$\hat{\mathbf{S}} = \arg\max_{\mathbf{S}} P\left(\mathbf{X} \mid \mathbf{S}\right)$$

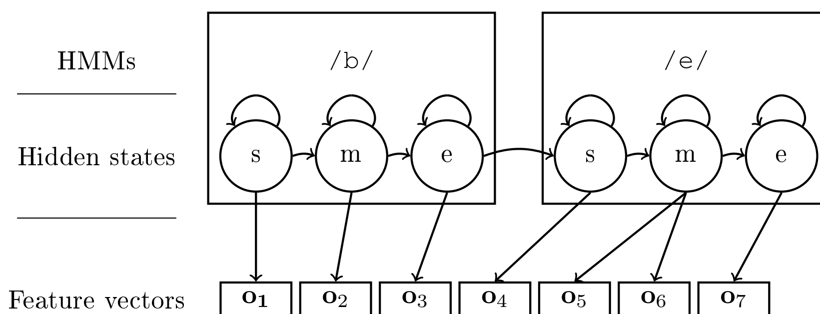


Figure 2.4: HMM emitting the word *be*

2.3.2 Gaussian Mixture Models

As mentioned, each state in an HMM emits an observable output from a certain probabilistic function. For this function it is common to use a *multivariate Gaussian mixture*. A Gaussian

mixture distribution can make any one-dimensional probability distribution as a weighted sum of individual Gaussians, like illustrated in figure 2.5. *Multivariate* distributions generalize this to multiple dimensions. This ability to model any multi-dimensional distribution is the reason why multivariate Gaussian mixture models (GMMs) are used to model speech features statistically [14].

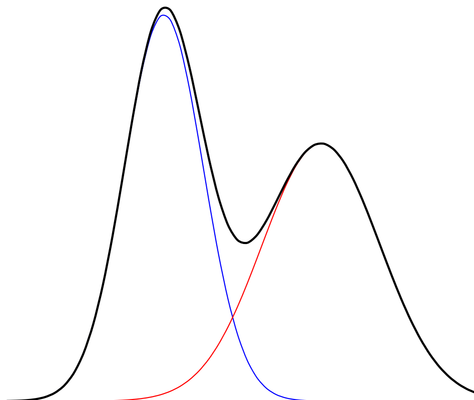


Figure 2.5: A distribution made by a mixture of two Gaussian distributions.

2.3.3 Environmental Robustness

There are multiple ways of improving the robustness of HMMs using data from the noisy environments. One of them is to simply retrain the model using speech from the relevant environment. There are several ways to do this too. The method that is probably best fit for training speaker-independent systems is *multistyle* training, which means training HMMs on data from different acoustic environments. This has been shown to reduce error rate by more than a factor of two [19].

To avoid retraining the entire model, post-training *adaptation* methods can be used, like *MAP* or *MLLR*, which will be explained briefly below. Both of these methods can be used for adapting to both speaker and environment. While MAP (Maximum A Posteriori) generally can improve the performance more than MLLR, it requires a significant amount of adaptation data. When data is limited, MLLR (Maximum Likelihood Linear Regression) is a better option [14, 5]. More details about the mathematics and algorithms used for adaptation can be found in [36] and [14].

MLLR Adaptation

MLLR works by computing a set of transformations that can be used to reduce the mismatch between the initial model and the adaptation data. More specifically, the *expectation maximization* technique, a commonly used technique for unsupervised machine learning, can be used to calculate transformation matrices. These transform the GMM means and variances so that they are better adapted to the speech data of interest, or in this case, the environment.

With a limited amount of data, a *global* adaptation transform can be generated. This transform will then be applied to every Gaussian component in the model set. With more adaptation data available, more transformations can be produced, and applied to different groups of Gaussian components, for a finer adaptation [36].

MAP Adaptation

The Maximum A Posteriori adaptation method, in contrast to MLLR, creates a completely new model based on the original model and the adaptation data. In MAP, *every* mean component in the system is updated, and not just classes of Gaussians like in MLLR. This is the reason why more data is required [36].

2.4 Neural Networks

Neural networks have in recent years shown promising results in several machine learning problems, including speech recognition. They are therefore naturally interesting for the problem of noise robust small-vocabulary systems too. This section will explain the basics of neural networks (NNs) and their advantages over traditional methods.

2.4.1 Basic Theory

The basic building block of a neural network is the (artificial) *neuron*, which is illustrated in figure 2.6(a). This can be viewed as a non-linear function that generates one output value from one or more input values. Generally, the output depends on a weighted sum of the inputs, a single bias value, and an *activation function*. The purpose of the activation function is to keep the output from "blowing up" by restricting its value to a certain range, like $y \in [0.0, 1.0]$.

While the functionality of a single neuron is relatively simple, the power of artificial neurons become apparent when they are connected in a network, hence the name *neural network*. A neural network consists of multiple layers of neurons, including one input layer, one output layer and *at least* one hidden layer between them, as illustrated in 2.6(b).

In practice, the output of neural networks is calculated as matrix multiplications, using the *forward computation* algorithm [37].

2.4.1.1 Hybrid models

DNNs can be used for ASR in multiple ways. One of the most successful uses so far is in combination with HMMs. In these so-called DNN-HMM hybrid systems, DNNs are used in place of GMMs for frame-wise acoustic classification. In other words, HMMs are still used to model sequences, but the posterior probability of HMM states given acoustic observations are estimated by DNNs instead of GMMs [37].

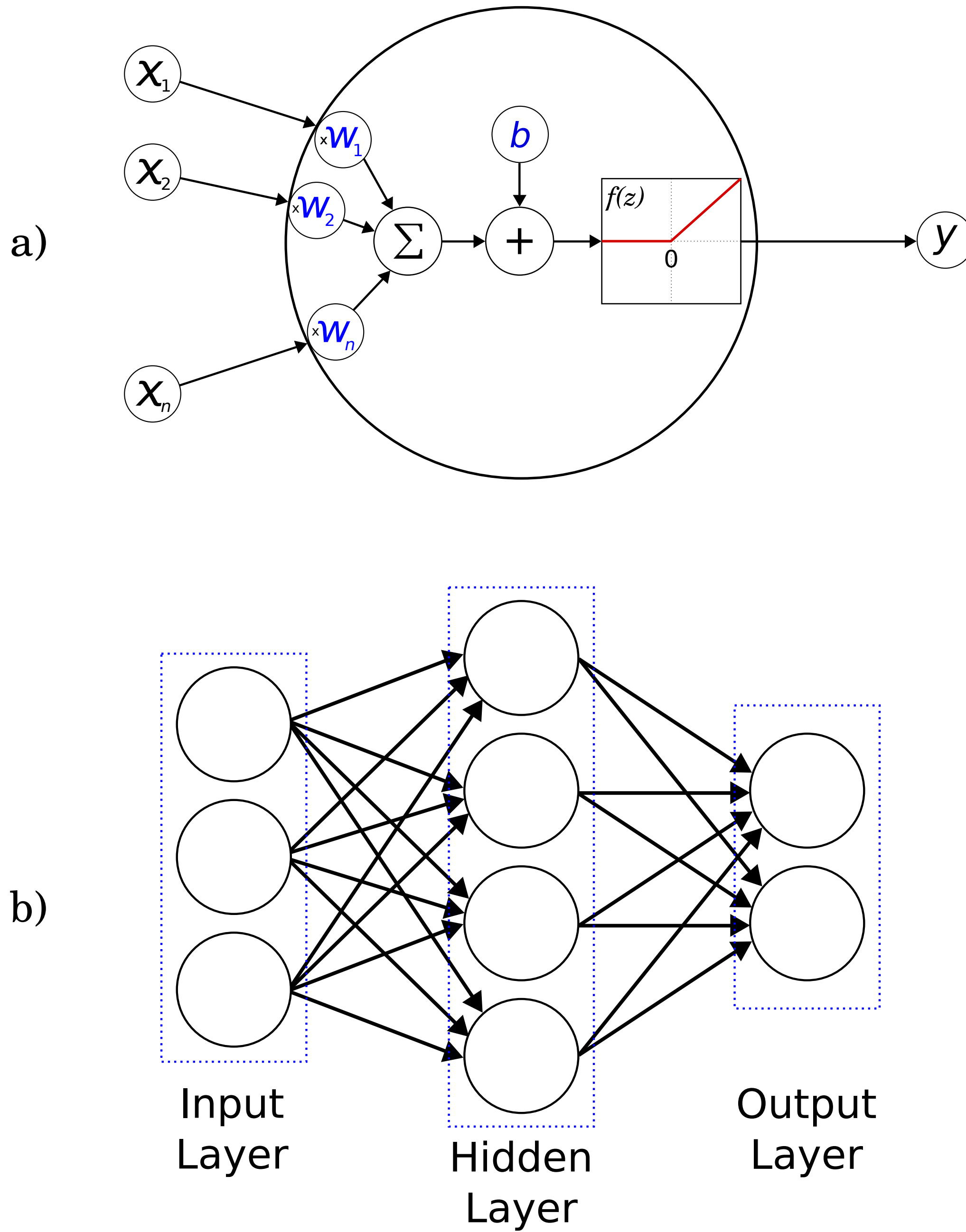


Figure 2.6: **(a)** The structure of an artificial neuron. **(b)** A "vanilla" neural network [15].

2.4.2 Recurrent Neural Networks

The "vanilla" deep neural network illustrated in figure 2.6 is a feed-forward network, in other words a network with no recurring connections. Although such a network can work well for classification of stationary data, it is not the best choice for sequential data. For that, *recurrent neural networks* (**RNNs**) is a better option.

RNNs are neural networks with recurrent connections, illustrated as cycles in the graph in figure 2.7. What this means is that the output of the neuron with a cycle depends on its previous output values in addition to an external input, like stated mathematically in equation 2.3. Here, $\mathbf{c}_t$ is the output value of the given neuron at time t, x_t is the external input, and $\boldsymbol{\theta}$ is the parameters, like weights and biases.

$$c_t = f\left(c_{t-1}, x_t; \boldsymbol{\theta}\right) \tag{2.3}$$

As feed-forward NNs, RNN outputs are also computed by a form of forward propagation. Training of RNNs consists of such a forward-propagation followed by a special form of back-propagation called *back-propagation through time* (BPTT). The details of forward computation and BPTT are not explained here, but can be found in *Goodfellow et al. 2016* [9].

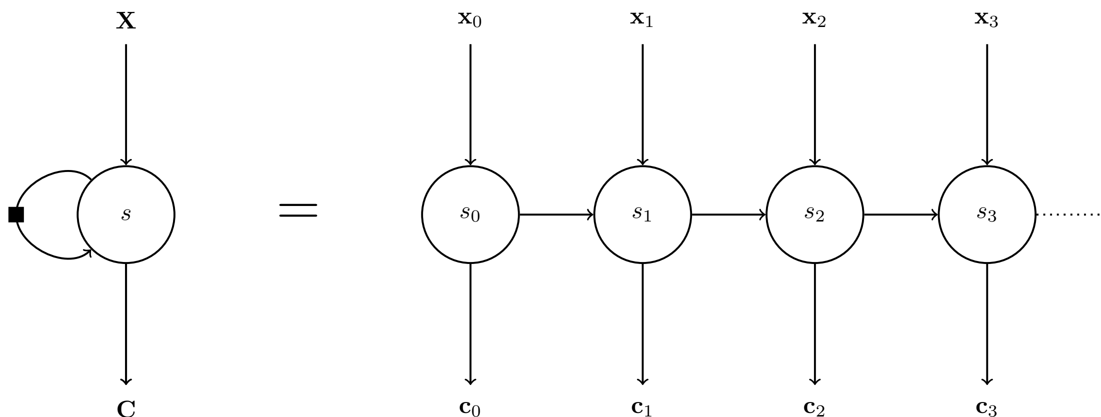


Figure 2.7: Recurrent neuron. The black square on the left represents a delay of one time unit. The right side of the figure shows the neuron unfolded in time.

Long-Term Dependencies

While RNNs work well at modeling sequences with short-term dependencies, they have more problems with long-term dependencies. This is because gradients propagated through time will tend to either explode or, more often, vanish. The reason is that weights given to long-term interactions are exponentially smaller compared to short-term interactions. The vanishing gradient problem has multiple possible solutions, including *leaky units*, and *gated RNNs* like *long short-term memory* (LSTMs) and *gated recurrent units* (GRUs). [9].

2.4.3 Connectionist Temporal Classification

Vanilla NNs generally have to work with fixed length inputs and outputs. RNNs are a bit more flexible. They can work with sequences of any length, as long as each element in the sequence has the same dimensions. A challenge that still remains with the use of RNNs in speech recognition, however, is that the length of the output sequence is equal to the length of the input sequence. In speech recognition, the output sequence (transcription) is in the majority of cases *shorter* than the input sequence (feature matrix).

Connectionist Temporal Classification, or *CTC*, is an algorithm used to train deep neural networks for labeling of unsegmented data [13, 12], using a loss function with some useful properties that will be explained below. One of the advantages of this type of training is its ability to work with sentence-level transcriptions *without* depending on HMMs like hybrid models do. Another

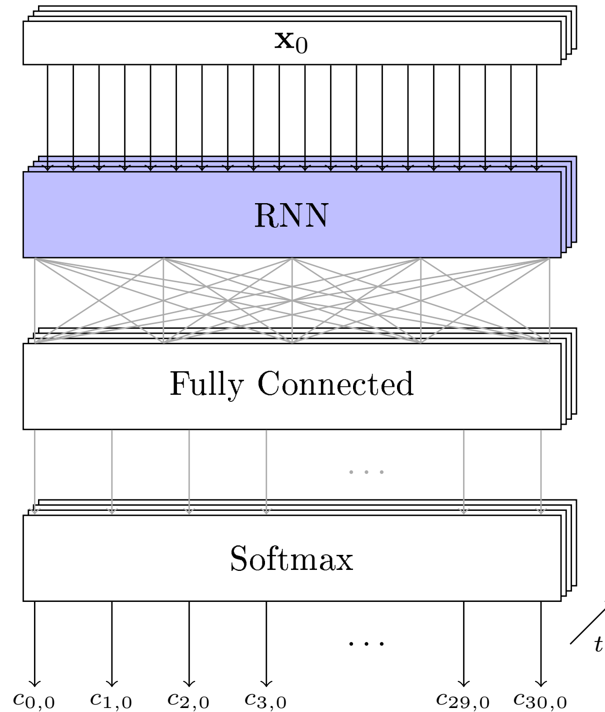


Figure 2.8: The output of an RNN made for training with CTC is a softmax probability distribution $c_{l,t}$ over the characters in the alphabet L.

advantage is the ability to work with graphemes directly, without a pronunciation lexicon. This requires large amounts of data, however, so for smaller databases it is still a good idea to use a lexicon and train on phonemes instead.

To explain CTC, the output of a basic RNN as the one illustrated in figure 2.8 is used as basis. A fully connected layer and a *softmax* function both follow the RNN in order to create a probability distribution over the valid output labels, i.e. the *alphabet* L. Such a probability distribution is then produced for each time slice input to the network, as the t-dimension illustrates. In the context of speech recognition, this input will be a single feature vector, like MFCC, or the Fourier transform of a single audio frame.

A resulting sequence of softmax distributions is illustrated in figure 2.9. In this example, the alphabet L of graphemes recognized by the network is defined as the lower-case Norwegian alphabet, in addition to *space* and the *blank* token. The blank token is explained in the context of CTC below. The probability of a certain character sequence, or *path*, $\mathbf{s} = [s_0, s_1, ..., s_{T-1}]$ is then given by the product of probabilities,

$$P(\mathbf{s} \mid \mathbf{X}) = \prod_{t=0}^{T} c_{s_t,t}. \tag{2.4}$$

For example, the probability of the correct path being $\mathbf{s} = \text{"a} - \text{b} - \text{bbb} - \text{aa"}$ is then given by equation 2.4 for the sequence $s = \{2, 0, 3, 0, 3, 3, 3, 0, 2, 2\}$, i.e. the product of the circled elements in figure 2.9.

2.4.3.1 Mapping

To solve the problem of the output sequence being of equal length to the input sequence, CTC defines a mapping $\beta(\mathbf{s}) \rightarrow \mathbf{y}$. The mapping simply removes repeat tokens and blank tokens. The blank token is a token introduced by CTC in order to deal with silence within words and the need to "force" repeated characters, like double consonants, which would otherwise be impossible to model. Using the same example sequence as above:

$$\beta(\text{"a} - \text{b} - \text{bbb} - \text{a"}) = \text{"abba"}.$$

-	$c_{0,0}$	$c_{0,1}$	$c_{0,2}$	$c_{0,3}$	$c_{0,4}$	$c_{0,5}$	$c_{0,6}$	$c_{0,7}$	$c_{0,8}$	$c_{0,9}$
–	$c_{1,0}$	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$	$c_{1,4}$	$c_{1,5}$	$c_{1,6}$	$c_{1,7}$	$c_{1,8}$	$c_{1,9}$
a	$c_{2,0}$	$c_{2,1}$	$c_{2,2}$	$c_{2,3}$	$c_{2,4}$	$c_{2,5}$	$c_{2,6}$	$c_{2,7}$	$c_{2,8}$	$c_{2,9}$
b	$c_{3,0}$	$c_{3,1}$	$c_{3,2}$	$c_{3,3}$	$c_{3,4}$	$c_{3,5}$	$c_{3,6}$	$c_{3,7}$	$c_{3,8}$	$c_{3,9}$
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
ø	$c_{29,0}$	$c_{29,1}$	$c_{29,2}$	$c_{29,3}$	$c_{29,4}$	$c_{29,5}$	$c_{29,6}$	$c_{29,7}$	$c_{29,8}$	$c_{29,9}$
å	$c_{30,0}$	$c_{30,1}$	$c_{30,2}$	$c_{30,3}$	$c_{30,4}$	$c_{30,5}$	$c_{30,6}$	$c_{30,7}$	$c_{30,8}$	$c_{30,9}$
	0	1	2	3	4	5	6	7	8	9

$\longrightarrow$
t

Figure 2.9: Softmax matrix output of an RNN. The first character, $c_{0,t}$, marks the *blank* character introduced by CTC. The circled elements constitute a sequence $\mathbf{s}$ with $\beta(\mathbf{s}) =$ "abba".

Since $\beta(\mathbf{s})$ is a many-to-one mapping, the actual probability of a transcription $\mathbf{y}$ being the right transcription is given by the sum of probabilities for all sequences $\mathbf{s}$ mapping to it:

$$P(\mathbf{y} \mid \mathbf{X}) = \sum_{\mathbf{s}:\beta(\mathbf{s})=\mathbf{y}} P(\mathbf{s} \mid \mathbf{X}), \tag{2.5}$$

for instance

$$\begin{aligned} P(\text{"abba"} \mid \mathbf{X}) =& P(\text{"a} - \text{b} - \text{bbb} - \text{aa"} \mid \mathbf{X}) \\ &+ P(\text{"aa} - -\text{b} - \text{baa} - \text{"} \mid \mathbf{X}) \\ &+ P(\text{"} - -\text{a} - -\text{bba} - -\text{"} \mid \mathbf{X}) \\ &+ \cdots \end{aligned}$$

2.4.3.2 Training

As seen in equation 2.5, the CTC objective is not defined for an individual time slice, which is the case for standard NNs, but rather for the entire input-output pair $(\mathbf{X}, \mathbf{y})$. During training, the model parameters are adjusted to minimize the negative log-likelihood for all input-output pairs in the training set $\mathcal{D}$:

$$\sum_{(\mathbf{X},\mathbf{y})\in\mathcal{D}} -\log P(\mathbf{y} \mid \mathbf{X}). \tag{2.6}$$

Calculation of the conditional probabilities $P(\mathbf{y} \mid \mathbf{X})$ can be done with a dynamic programming algorithm similar to the forward-backward algorithm used for HMMs [12]. An explanation of this algorithm will not be included here, but can be found in the 2017 online article by Hannun [13].

SortaGrad

Amodei et. al. 2016 found that CTC training was occasionally unstable, especially in the early stages. To counter this, they experimented with a *curriculum learning method* they call *SortaGrad*. A curriculum learning method simply means introducing gradually more difficult training examples

instead of choosing data randomly [3]. In the case of SortaGrad this means that training data is sorted by duration for the first training epoch, then chosen randomly for the remaining epochs. The results of the experiments were successful, showing that SortaGrad helped improve training stability [1].

2.4.3.3 Inference / Decoding

The most likely transcription, $\mathbf{y}^*$ given input $\mathbf{X}$ is

$$\mathbf{y}^* = \arg\max_{\mathbf{y}} P(\mathbf{y} \mid \mathbf{X}),$$

In practice, this is of course way too computationally expensive, as the number of possible transcriptions increases exponentially with the length of the input. Inference of the most likely transcription can therefore only be approximated. This can be done in two ways,

1. *Best path decoding*, and
2. *Prefix (beam) search decoding* [12].

Best path decoding simply takes the most probable path and applies the β-mapping to it:

$$\mathbf{y}^* = \beta\left(\arg\max_{\mathbf{s}} P(\mathbf{s} \mid \mathbf{X})\right)$$

This is computationally easy, but has a significant drawback. It does not take into account that multiple output sequences $\mathbf{s}$ from the RNN can map to the same transcription $\mathbf{y}$. Sometimes sequences that map to the same $\mathbf{y}$ each have smaller individual probabilities than a $\mathbf{s}$ mapping to a different $\mathbf{y}$, but their sum is greater.

Prefix beam search decoding is a modified version of the beam search algorithm that takes the many-to-one mapping into account. Prefix beam decoding, like normal beam search, offers a choice between better accuracy and lower computational demand. This is determined by the beam width, which in turn determines the number of possible alignments checked [13].

Chapter 3

Experiments

This chapter deals with the methodology used, the choice of software tools, and describes how the experiments were set up and executed. Section 3.1 describes the general approach, what kind of systems to implement, and the choice of tools and databases used to do so. The remaining sections are dedicated to model implementation and evaluation descriptions.

3.1 Method

3.1.1 Tools

There already exists a vast selection of toolkits for making speech recognition systems based on different techniques, including those discussed in chapters 2.3 and 2.4. Although the goal of this thesis is to find which classifier is best suited for a speech command system that should be used in noisy environments, the focus is also on the eventual implementation of such a system in a production environment. Many of the open source toolkits are available for both Linux and Darwin operating systems. This is preferable because some of the systems will potentially be used both on Darwin and Linux servers.

3.1.2 Choice of Techniques

Some properties should be common for all the systems to be evaluated here. These properties are based on the criteria mentioned in the introduction, one of the most important being *expandability*. In practice this means that the system has to be trained for *general-purpose* speech recognition, although the task is really small-vocabulary recognition. With this in mind, whole-word models, as discussed in section 2.1.2, become infeasible to train. All the systems experimented on here will therefore be trained on sub-word units. Another commonality between the systems is that they will all be trained and tested on Norwegian data, as this is the most probable use of the potential final implementation of the system.

As there exists an uncountable number of different speech recognition systems and system configurations, it is impossible to perform experiments on all of them. The number of systems discussed in this thesis has been limited by multiple factors, including computational complexity, database size, and most importantly time constraints.

3.1.3 Training databases

When training a general-purpose system, a large amount of speech data is required. The National Library of Norway offers multiple speech databases through their service *Språkbanken* [24], some of which are already available on servers on NTNU. The chosen databases are presented below.

NAFTA

NB Tale, also called *NAFTA*, is a Norwegian acoustic phonetic speech database [21]. In other words, it contains speech audio that has been manually annotated on a sub-word (phoneme) level, and

is therefore relatively small. Although the phoneme-level annotations can be used for supervised learning, all the trainings performed here only use sentence-level transcriptions. NAFTA consists of recordings from 260 native speakers of Norwegian and 120 speakers with Norwegian as a foreign language. Two out of three parts are manuscript-read speech, while the third is spontaneous speech.

NST Acoustic database

NST acoustic speech database for Norwegian [22], or just *NST* for short, is one of the largest databases available. The database has multiple parts, including 16 kHz, 22 kHz and 44 kHz audio. Only the 16 kHz part has been used for experiments here. According to the documents published with the database, this should consist of a training part with 900 speakers reading 312 lines and a testing part consisting of 80 speakers reading 987 lines. The publicly available database, however, seems unstructured and uncontrolled, and the first experiments using this database revealed that some audio samples are erroneous and others completely missing. The NST database used for the successful experiments here is therefore a restructured and controlled version that is available on the NTNU servers.

NST lexical database

Section 2.1.3 explained the need for a pronunciation model, also referred to as a phonetic dictionary or lexicon. One such lexicon, and probably the most relevant, is the *NST Lexical database for Norwegian Bokmål* [23]. Also this database was used in a modified version (made for HTK) that is available on NTNU servers.

3.1.4 Testing database

Grammar specification

In order to evaluate the noise robustness of the systems on a small vocabulary, a provisional grammar was specified based on a limited subset of the temporal control commands seen in figure 1.1. The specified grammar, hereby referred to as `luxgrammar`, is illustrated as a grammar network in figure 3.1. This yields a total of 10 valid sentences, which should be feasible to collect enough testing data for.

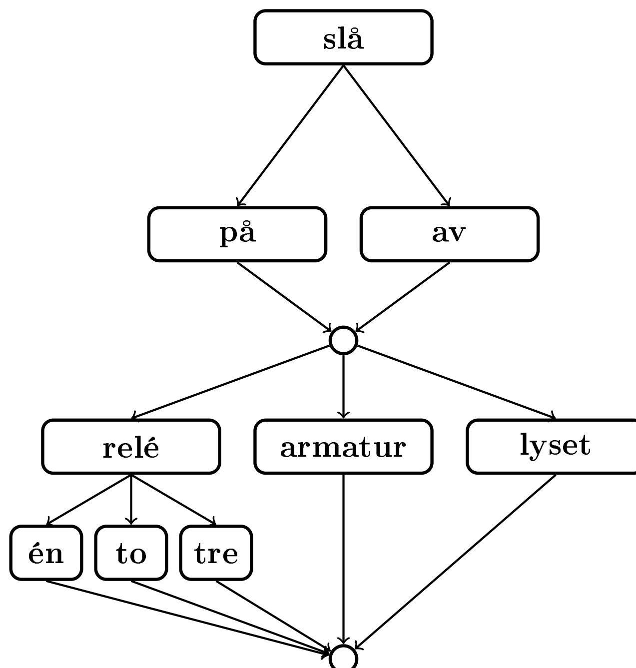


Figure 3.1: Grammar network for a small vocabulary.

Data collection

In order to collect a decent amount of data for testing, a simple website was set up with functionality for recording and uploading to the host server, based on open source code.[1]

On the website, detailed instructions were given for how to perform the recordings. Several design decisions were made for lowering the threshold for people to participate, some of which have drawbacks. One worth mentioning is the choice of recording all the 10 sentences in a single take. A potential consequence of this is that participants might emphasize words differently from what they would do if read individually. Another consequence is the need for post-processing in the form of splitting each recording into 10 sentences, although this has a positive effect too, as it forces manual control of all the samples. Yet another choice that was made in order to lower the participation threshold was to avoid the need to fill in any forms, making the process completely anonymous. Because of this, no personal information was gathered on the speakers. From listening to the recordings, it sounds like 24 speakers participated in total. As speakers were simply encouraged to do as many recordings as possible, the number of recordings per speaker varies between one and eight.

The collected database will be referred to as `luxdb` for the rest of the thesis.

3.1.5 Noise Simulation

As with all of the experiments, there are multiple ways of making noisy recordings. The ideal way would be to perform relevant data collection in noisy environments directly, but this would require significant time and resources compared to simply simulating noise digitally. The choice, therefore, fell on the latter alternative. Noise was recorded along the road using a smart phone with and without an external (headset) microphone.

Using equations 2.1 and 2.2, the noise scaling factor can be calculated as

$$a = \sqrt{\frac{P_{\text{sig}}}{10^{\text{SNR}_{\text{dB}}/10} P_{\text{noise}}}}, \tag{3.1}$$

where

$$P_{\text{sig}} = \frac{1}{N} \sum_{n=0}^{N-1} x_{\text{sig}}^2[n], \quad \text{and} \quad P_{\text{noise}} = \frac{1}{N} \sum_{n=0}^{N-1} x_{\text{noise}}^2[n].$$

This was used to make a Python script that automatically adds noise with a given level relative to each audio clip. All the audio files in a given *noise* directory are read, along with all speech audio files to be corrupted by noise. The script iterates through the list of speech files, and for each file picks a new segment of noise (with length equal to that of the speech signal) from one of the noise files. When reaching the end of a noise sample, the scripts jumps to the next noise sample and starts using segments from that. Because of the limited amount of noise samples, they are used in a cyclic fashion, i.e. after reaching the end of the last noise sample, the script jumps back to the first and starts over. The chosen noise segment is scaled by a factor found using equation 3.1 before being added to the speech sample. To be consistent with the limitations of toolkits used, audio is restricted to a 16 bit resolution and 16 kHz sample rate. In cases where the sum of the noise signal value and speech signal value exceeds the maximum value for 16 bit integers, the signal is simply clipped, as it will most probably do in a real-word scenario. The entire noise simulation script can be found in appendix B.1.

3.2 GMM/HMM Implementation

The technique given the most focus here, because of the limited amount of data and computational resources, is the traditional GMM/HMM. Implementation of such a model from scratch is a hard task, so the choice naturally fell on using open source speech recognition tools, as there are plenty to choose from. Perhaps the most recognized toolkit is *the Hidden Markov Model Toolkit* (HTK)

[1]source available at `https://github.com/addpipe/simple-recorderjs-demo`

[31]. This is however primarily intended for speech recognition research, and has a restrictive commercial license [30].

As the system in question should not be hard to port to a commercial production system, the choice therefore fell on another set of tools called CMUSphinx, or just *Sphinx*. In addition to a license that allows for commercial distribution [7, *About*], CMUSphinx has multiple recognizer implementations. One of these, *Sphinx4*, is written in Java, which is ideal for a company like Luxsave, which currently has a Java dominated code base. Yet another benefit of the Sphinx toolkit, is that it includes tools for acoustic model adaptation using the techniques described in chapter 2.3. Lastly, Sphinx has been found to give good results out of the box in large-scale evaluations [8].

While the implementation of a recognizer using Sphinx4 was straight-forward, significant problems arose when attempting to train an acoustical model for use with it. As mentioned earlier, the intention of the experiments is to evaluate the performance of a system trained on Norwegian speech data. There are multiple pre-trained Sphinx models available, but none for the Norwegian language. In order to train acoustic models for use with Sphinx recognizers, another tool called *Sphinxtrain* had to be used, and using this proved more difficult than first expected.

Sphinxtrain depends on several other software distributions, including two other CMUSphinx tools: *Sphinxbase* and *Pocketsphinx*. These dependencies eventually led to multiple issues, as some of the released versions contains bugs and incompatibilities with other dependencies. Debugging was challenging because it is hard to tell software errors and database errors apart without a thorough understanding of the system. The details of the debugging will be left out, as they are not directly relevant to the experiments. In the end, it turned out that installing all the packages from their newest source code solved most of the problems, despite the tutorials recommending the use of the (binary) released versions. After the setup of Sphinxtrain was successful, it was replicated in a *Docker* container to ensure reproducibility [4]. A Docker container with the successful setup can be made using the Dockerfile and scripts supplied in section D.1.

3.2.1 Data preparation

With a working Sphinxtrain setup, all that is needed for training is to get the training resources on the right format. Sphinx takes training data on a specific format that differs from the formats found in available databases. The formats also differ between the different Norwegian databases. For this reason, unique data preparation scripts had to be made for each database to be used. The scripts are not included in this thesis, because the potential for reuse is small.

With the right resources on the right format, a single script included in Sphinxtrain can be run to automatically do everything needed to make and train an acoustic model. This includes (MFCC) feature extraction, model initialization and training, and in the end decoding to give an impression of the model performance.

3.2.2 Cross-validation on `luxdb`

Just to confirm the hypothesis that `luxdb` is too small for training a good acoustic model, the database was used for a quick cross-validation. As seen in section 4.2, the hypothesis seems to hold, and so more general models are needed. The following subsections deal with these models.

3.2.3 Benchmark models

To begin with, models were trained on clean data from the Norwegian databases. Sphinxtrain seems to have carefully chosen default model parameter values, so these were left untouched for all the trainings performed. The parameters are as follows.

- *Number of filters*: 25
- *Lowest filter frequency*: 130 kHz
- *Highest filter frequency*: 6800 kHz
- *Transform type*: DCT
- *Feature type*: MFCC including Δ and $\Delta\Delta$ (`1s_c_d_dd`)
- *Vector length*: 13

- *Liftering sin-curve length*: 22
- *Cepstral Mean Normalization type*: batch
- *BW min iterations*: 1
- *BW max iterations*: 10
- *Tied states (senones)*: 200

Because of time constraints, the only factor changed between the different models is the database used for training. This was done to see if more data improves model performance as much as one would expect. The three benchmark models trained have been named `nafta_12`, `nst_small`, and `nst_full`. The model trained on the least amount of data, `nafta_12`, was trained on a limited number of samples from parts one and two of NAFTA. This was done because NAFTA was the only database available without errors at the time of the first experiments. As mentioned, the publicly available NST database is unorganized and erroneous, and it was therefore problematic to use, which is why a `nst_full` was trained on the newly restructured version. `nst_small` was trained on a subset that was copied from the NTNU speech database server while the restructuring was taking place. The given subset consisted of about 6.6 GB of data, while the full NST on the server has 55 GB, so going by the size alone, the subset consists of 11-12% of the full set.

For all of the experiments, the modified version the phonetic dictionary from NST was used as a pronunciation lexicon. For the experiments using the NST database, this worked well. Most of the data in NST 16 kHz database consists of words found in the lexicon, although there are still some samples that had to be excluded because they contained out-of-dictionary words. The missing words could of course have been added to the dictionary, but this was considered unnecessary. After running a script that automatically removed all samples containing out-of-dictionary words, there were still 352010 samples left in the `nst_full` case, and 39495 in `nst_small`. These numbers were deemed sufficient. For the NAFTA setup, the NST lexicon did not work that well. Out of 7539 original samples, only 2662 were left after the removal of samples containing non-dictionary words. A model was trained on these samples nonetheless, just to see if it would be comparable to the NST based models.

3.2.4 Improving Robustness

In order to improve recognition accuracy, the techniques explained in section 2.3.3 can be employed on the models trained on large databases to adapt them for the noisy environments. Sphinxtrain makes this process easy, as it implements programs for performing both MLLR and MAP adaptations. This can be done in several ways, two of which were attempted. The chosen methods are explained in more detail in the following subsections.

MLLR Adaptation on `luxdb`

As MLLR is most often the best choice when adaptation data is limited, this was naturally the method of choice for adapting models to the collected testing data directly. For this adaptation, 20% of the `luxdb` database (130 audio samples) was randomly picked for use as adaptation data. A random $\mathrm{SNR}_{\mathrm{dB}} \in \{3, 6, 12, 15, 20, 30, 40\}$ was chosen for each of the 130 audio clips in order to simulate different environments. The noisy audio clips were not generated specifically for the adaptation. Instead, the already "noisified" clips from section 3.1.5 were used, and the randomly chosen $\mathrm{SNR}_{\mathrm{dB}}$ value was simply used to tell the Sphinx tools which directory to find the samples in.

MAP Adaptation on NST

For MAP adaptation, the amount of data in `luxdb` is probably not sufficient. Instead, the data from NST itself was used to adapt to the environmental noise. This was done by running a modified version of the noise simulation script (section 3.1.5) on the `nst_small` subset of NST. The modified noise script was made to pick SNR value randomly (from the usual set of SNRs) for each speech sample and scale the noise accordingly. That way a new database of speech samples with SNRs from 3 dB to 40 dB was produced, with the same amount of data as `nst_small`. This

data was used for the MAP adaptation, which means that the new model should learn to handle noises better.

Multistyle Training

After evaluating the performance of the adapted models, a new model was trained from scratch using noisy data. Multistyle training, explained in section 2.3.3, ideally consists of a training database that contains data recorded in multiple environments. As mentioned earlier, the collection of such a database requires more resources than most small companies have available. To make things simple, partly because of restricted time, the "multistyle" training here consists of the exact same data as was used for the MAP adaptation explained above, in addition to the remaining (*clean*) parts of NST. More specifically, the training data consisted of 352010 speech samples, 39696 of which were corrupted by various levels of noise.

3.3 GMM/HMM Evaluation

To evaluate the performance of models trained by Sphinxtrain, a Java application using Sphinx4 was used. Sphinxtrain itself could be used for evaluation, as it does include a decoding script that uses the Pocketsphinx decoder and calculates a word error rate from the results. Sphinx4 was chosen, however, to achieve a setup as close to the potential production system as possible. The evaluation program was made to take two arguments, the first being the name of the model to evaluate, and the second being the path to speech audio files. A third, optional, `--nogrammar` argument was later implemented for running evaluations using a language model instead of a grammar, as explained in more detail below.

Although Sphinx4 supports the use of both a language model and a *grammar*, documentation on the use of grammars seems scarce. Fortunately the configuration proved relatively straightforward to figure out with the help of an auto-completing Java IDE. To set up the system for use with a grammar, the following lines were needed:

```
configuration.setUseGrammar(true);
configuration.setGrammarName("luxgrammar");
configuration.setGrammarPath("/path/to/grammar_directory/");
```

With this configuration, Sphinx4 will then search for a file named `luxsave.gram` in the grammar directory. The `.gram` file is on the JSpeech Grammar Format (JSGF), a format designed specifically for use in speech recognition systems [34]. As evident by the documentation, this format is quite sophisticated, and should be able to support advanced grammars. The `luxgrammar` from figure 3.1 however, is simple enough to be implemented in its entirety using the `.gram` file listed in listing 3.1, at least for evaluation purposes.

Listing 3.1: luxgrammar.gram

```
#JSGF V1.0;

grammar luxgrammar;
public <command> = slå (på | av) (relé (én | to | tre) | armatur | lyset);
```

When using a grammar, the Sphinx decoder returns either one of the utterances allowed by the grammar, or *unknown* (`<unk>`) if it was unable to align the audio with any of the valid utterances. In case of valid utterances there might also be silence (`<sil>`) in between the words, or before or after them. To use the grammar for classification, these are simply removed using a regular expression before the string is compared to the valid classes. Again, this might not be the "right" way to do it in a production system, but it is sufficient for the model evaluation to be performed here.

3.3.1 Language Model

Although language model evaluations are more relevant for general-purpose speech recognition than they are here, they are included for comparison. Recognition performance should be better

both for the grammar case and language model case when a better acoustic model is used, but nevertheless it is interesting to see if the system behaves as expected.

For the creation of a language model, another Sphinx-related toolkit called *CMUCLMTK* was used. The procedure of building a language model with this toolkit is documented in the Sphinx tutorial [6]. To create a language model that is usable for evaluations on the `luxdb`, the text used for gathering statistics consisted of all the transcriptions from NST (351042 utterances), in addition to all the transcriptions of the `luxdb`. As the LM is based on trigrams made from the text supplied, the 650 sentences from the testing database should make the LM sufficiently able to recognize the testing samples.

3.4 CTC-trained RNN

This subsection presents a modern alternative to the GMM/HMM recognition system. The experiments discussed in this subsection uses an RNN with gated recurrent units, as explained in subsection 2.4.2. This RNN is trained using CTC (subsection 2.4.3) loss function. Because of the limited results from the experiments, this will be discussed more briefly than the GMM/HMM experiments, even though significant time was spent on it.

3.4.1 Implementation

As stated by *Hannun 2017* [13], the implementation of CTC is difficult. Luckily there are multiple open-source implementations of it already. One of these is `warp-ctc` [28] by Baidu Research, which is licensed under the Apache License 2.0. This was chosen for the following experiments, as it seems like one of the most efficient implementations judging by their self-reported benchmarks. It was also easy to set up for training, following Baidu's own example in their `ba-dls-deepspeech` repository [27]. This implements a "CTC-compatible" model in Python 2, using Keras with Theano as back end, and an external Python package simply called `ctc` that implements Theano bindings for `warpctc`. Although `warp-ctc` implements both a CPU and a GPU version, bindings are only implemented for the CPU in the package used by `ba-dls-deepspeech`. An attempt was made at simply modifying the source to use the GPU version instead, but this didn't work as intended, as it made `warpctc` return a loss of zero for each call to it, for reasons unknown. No further attempts at making it work were done, because that would require more advanced and time consuming CUDA [20] programming. It was also later found that this is an unresolved issue (at the time of writing) with warpctc.

Network Architecture

The network used is based on the *DeepSpeech* architecture discussed by *Amodei et al. 2016* [1], and consists of the following layers.

1 Convolutional layer is used for handling temporal dependencies in the input spectrogram.

3 GRU layers are included for handling long-term dependencies. This has been chosen over LSTM because Amodei et al. found through experiments on smaller datasets that GRUs were faster to train and less likely to diverge.

1 Fully connected layer follows the GRU layers to produce a single output activation for each valid label.

Softmax completes the network by making a probability distribution over the defined alphabet, as explained in section 2.4.3. Softmax is included in the CTC algorithm implementation, and is therefore not needed in the Python/Keras model definition.

Training Procedure

Data preparation for CTC training consisted of writing yet another transcription conversion scripts, in order to get the transcriptions on the right format. A test run of the setup was done on a laptop with an Intel Core i7-740QM processor. The example worked out of the box for the *LibriSpeech* [25], but the hardware setup unsurprisingly proved too inefficient for training any model of a reasonable size. After making sure the training scripts worked, they were therefore copied to a more powerful computation server run by the signal processing department of NTNU, hereby referred to by its name, `sirkus`. On `sirkus`, the experiments were set up in an *Anaconda* environment [2] using the following software packages.

- `Python 2.7.15`
- `Keras 1.1.2`
- `Theano 0.8.2`

- numpy 1.15.4
- cmake 3.12.2
- make 4.2.1

The last two were only needed for building warpctc from its source code. A shell script for the entire setup can be found in Appendix D.2. The first attempt at training a network on NAFTA resulted in a NaN loss after two epochs, despite the use of SortaGrad and batch normalization. Decreasing the network learning rate from $2 \cdot 10^{-4}$ to $2 \cdot 10^{-8}$ made the training last longer, but it still did not converge, and the loss stayed high.

3.4.2 Evaluation

Although the CTC based training of RNN did not seem to work out well, some attempts were made at recognition using the network. The ba-dls-deepspeech code base implements best-path decoding only. Fortunately, Hannun [13] has published a code snippet[2] that implements prefix beam decoding.

Both best-path and prefix beam decoding were tested, but as expected they were unsuccessful at producing any meaningful interpretations of the output from the poorly trained network. Eventually, the experiments on CTC had to be considered a lost cause.

[2] https://gist.github.com/awni/56369a90d03953e370f3964c826ed4b0

3.5 Google Cloud Speech Evaluation

As a commercial alternative to the self-trained, self-hosted systems, the choice fell on Google's *Cloud Speech-to-Text* system. This is one of the most recognized cloud speech system, and it has a straightforward API that makes evaluation on it easy to perform.

The Cloud Speech-to-Text API is called by a Google *Cloud Client Library*, a software library used to construct requests to Google Cloud services. These libraries are available for multiple programming languages, including Python and Java. Requests are constructed as JSON[3] objects and sent to a Google server. The server then decodes the given speech audio file and returns a JSON object containing a list of possible transcriptions along with a value between 0 and 1 indicating the system's confidence in the associated transcription [10].

3.5.1 Models

Cloud Speech-to-Text offers four different speech recognition models at the time of writing. For these experiments, the `command_and_search` model was used, as that is made for short queries such as voice commands or voice search [10]. Details about the models used could not be found beyond the fact that they use neural networks. It seems, not surprisingly, like the models are meant to be treated as a black box by its users. The evaluations and discussion about this will therefore be quite superficial, and more interesting from a business perspective than a scientific perspective. It is included nonetheless, as the motivation for this thesis is in part commercial use. It is therefore interesting to compare the results of an open-source solution to one that is commercially available.

3.5.2 Phrase hints

In addition to offering general-purpose speech recognition, the speech API lets the user supply *speechContext* information in the form of a list of *phrase hints*. Phrase hints are given to the recognizer to tell it that said phrases have a higher probability of occurring [10]. This is naturally useful in the small-vocabulary, domain specific speech recognition case.

While phrase hints can improve recognition, it is not synonymous with a grammar, like the one used in Sphinx. Google still uses its general recognition system, seemingly a system that includes a language model, only with an increased probability of recognizing the given phrases. After some trial and error to see what the system seemed to make use of, the final phrases supplied as hints, were the ones listed in listing 3.2. As seen, the API can sometimes respond with different versions of the sentences, like "2" instead of "to". To evaluate whether a transcription is correct, these variations are simply corrected for using string replacements in the evaluation script.

[3] `http://json.org/`

Listing 3.2: Phrase hints given to the Google SR system

```
slå av
slå på
armatur
relé én
rele en
relé to
rele to
relé tre
rele tre
relé
rele
1
2
3
lyset
```

Evaluations of the Google system, hereby referred to as the `gspeech` model, were carried out on *all* the 650 samples in `luxdb`, as none of them were used for adaptation. Otherwise, the data was equal to the one used for Sphinx, i.e. consisting of speech corrupted by noise with $\text{SNR}_{\text{dB}} \in \{3, 6, 10, 15, 20, 30, 40, 60\}$.

Chapter 4

Results

This chapter starts by discussing speech recognition performance metrics in section 4.1. The sections that follow present the results from the evaluations explained in chapter 3.

4.1 Measurements of Performance

Word error rate (WER) is a commonly used measure of recognition performance in general-purpose ASR systems. It is defined by equation 4.1, and depends on the following three types of word recognition errors. *Substitution* means that an incorrect word was substituted for the correct word. *Deletion* error means a correct word being omitted in the recognized sentence. Lastly, an *insertion* error means that an extra word was added to the recognized sentence [14]. In equation 4.1, S, D and I are the number of substitutions, deletions and insertions, respectively. N is the total number of words in the reference sentence, i.e. the true transcription.

$$WER = \frac{S + D + I}{N} \tag{4.1}$$

WER can be calculated using the *Levenshtein distance*, or *edit distance* between two utterances [32]. For calculations presented in this chapter, an open source code Python implementation of the edit distance algorithm[1] was modified to return the *total* number of substitution, insertion, and deletion errors for all reference-hypothesis pairs in a model evaluation.

As mentioned, WER is a common performance metric when evaluating general-purpose ASR systems. The systems of interest here, however, are not general-purpose, and so WER is not that useful, except for the evaluations that used an LM instead of the grammar. There are a very limited number of valid sentences, at least in the provisional grammar. Each sentence represents a unique command, and the consequence of mixing them can be unfortunate. A performance metric that for this reason makes more sense to use here, is simply the *accuracy* of which the system is able to classify sentences correctly as one of the 10 valid commands (or as out-of-grammar). Another reason to not use WER for the grammar evaluations is that misclassified samples that are *not* classified as `<unk>` are guaranteed to be somewhat similar to the right utterance, which means that the WER will probably be low no matter how many misclassifications are done. Because some noisy samples had to be removed from the `luxdb`, the distribution of data is not uniform between the classes. For cases like this, it is often useful to calculate the F_1 score instead of just the accuracy of the different classifications [26].

Another option for presenting the results of a classification task is a *confusion matrix* (CM), which is probably the most unambiguous representations possible. CMs are in the format $\mathbf{A}_{t,p}$, where t is the true class and p is the model hypothesis, or *predicted* class. This makes it possible to see which classes the system often confuses. As there is a total of 171 confusion matrices from the experiments performed, only a select few will be included here. The CMs included differ from usual CMs in that they include an extra `<unk>` class for unclassifiable utterances. This is only included among the *predicted* classes, because no out-of-vocabulary samples were present in the `luxdb` to test Sphinx' rejection accuracy.

[1] Available at `https://github.com/zszyellow/WER-in-python`

4.2 GMM/HMM

As explained in section 3.3, evaluations were performed using both the luxgrammar and the language model. This section presents the most important findings from both kinds of evaluations. A more complete set of results can be found in the tables in appendix A. The results are represented here as plots of accuracy as function of SNR, as this makes it easier to compare them. Clean speech is included in the plots as $\text{SNR}_{\text{dB}} = \infty$, for further simplicity. The F_1 score of each evaluation was also calculated, but as they turned out almost indistinguishable from the accuracy values, they are only included in the appendix (A.3).

4.2.1 Cross-validation

The simple 5-fold cross-validation on context-independent phone based models trained on the luxdb gave an average utterance classification accuracy of **58.31%**. This is the accuracy of *clean* speech recognition, so based on this result it is clear that training a general model on a bigger Norwegian database is desirable.

4.2.2 Benchmark models

Figure 4.1 shows the performance of the three baseline models, and table 4.1 shows a confusion matrix for nst_full. As mentioned, there are too many confusion matrices to include all of them. Even for a single model there is a unique matrix for each of the 9 each SNRs tested. Table 4.1 therefore shows a *combined* confusion matrix for all SNRs used for evaluation of nst_full, in other words the sum of the individual matrices (which can be found in appendix A.2). The individual CMs reveal that most of the confusion stems from the model not being able to classify utterances with high levels of noise, as seen by the high number of **unk** misclassifications. For lower SNRs, the model correctly classifies most utterances, but there are some obvious points of confusion here too, which are underlined.

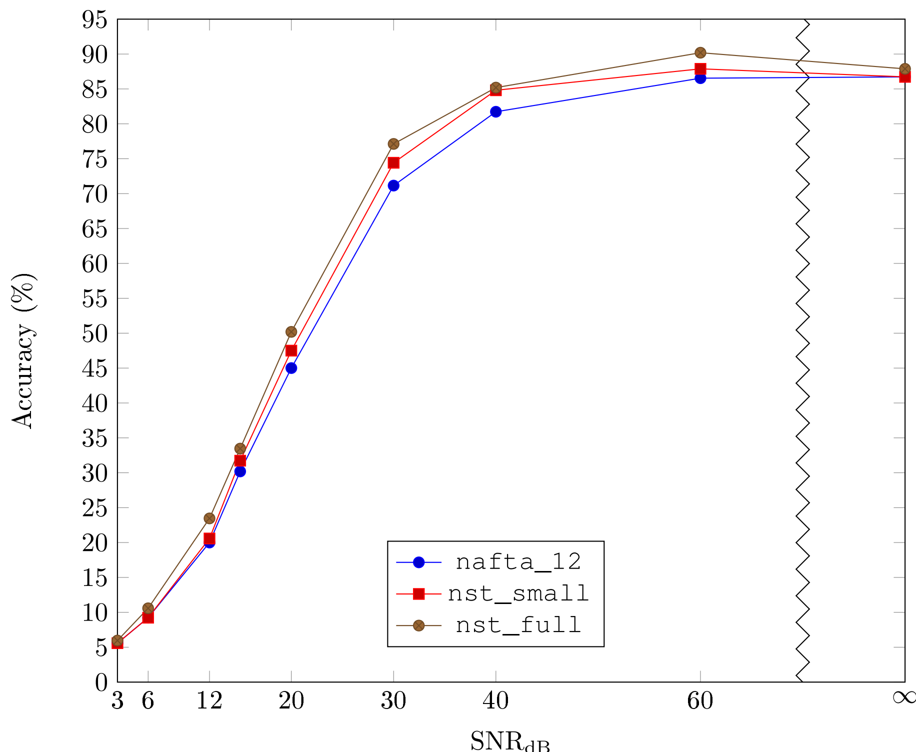


Figure 4.1: Classification accuracies for the three benchmark models

Table 4.1: Total confusion matrix for all evaluations on `nst_full`

T \P	**AA**	**AL**	**AR2**	**AR3**	**AR1**	**PA**	**PL**	**PR2**	**PR3**	**PR1**	**unk**
AA	294	0	0	0	3	31	0	0	0	0	185
AL	0	322	0	0	0	0	8	0	0	0	138
AR2	0	0	215	4	23	0	0	17	0	8	174
AR3	0	0	0	246	39	0	0	0	9	0	192
AR1	0	4	0	5	276	0	0	0	1	10	163
PA	34	0	0	0	0	217	0	0	0	0	208
PL	0	6	0	0	0	0	211	0	0	0	170
PR2	0	0	8	0	3	0	0	199	1	11	255
PR3	0	0	0	17	6	0	0	0	179	47	237
PR1	0	0	0	2	19	0	0	0	0	254	229

4.2.3 Adaptations

Figure 4.2 plots accuracies of the benchmark models after they have been MLLR adapted on the same noisy data. The plot in figure 4.3 shows and compares the effect of MAP adaptation and combined MAP+MLLR adaptation on `nst_full`.

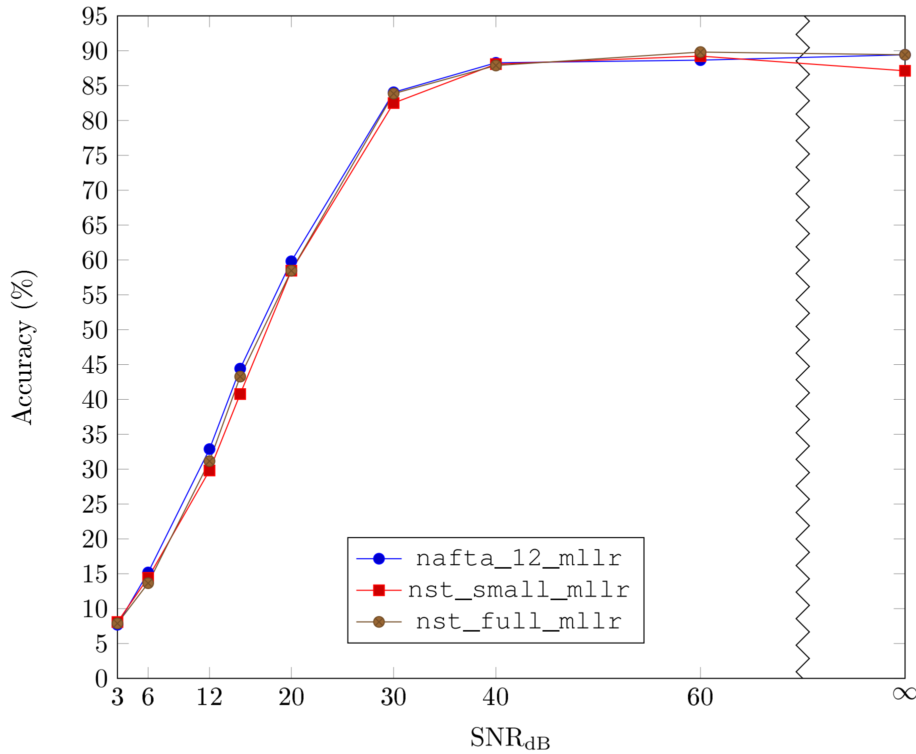


Figure 4.2: Classification accuracies after MLLR adaptation

4.2.4 Multistyle

Figure 4.4 plots the results of the multistyle model evaluations compared to the most successful adapted model from above. The 0% accuracy for 3 and 12 dB are not errors in the plot, but errors that occurred in the actual evaluation. By interpolating, however, it seems like the accuracy would be about 20-25% for 12 dB if no error had occurred.

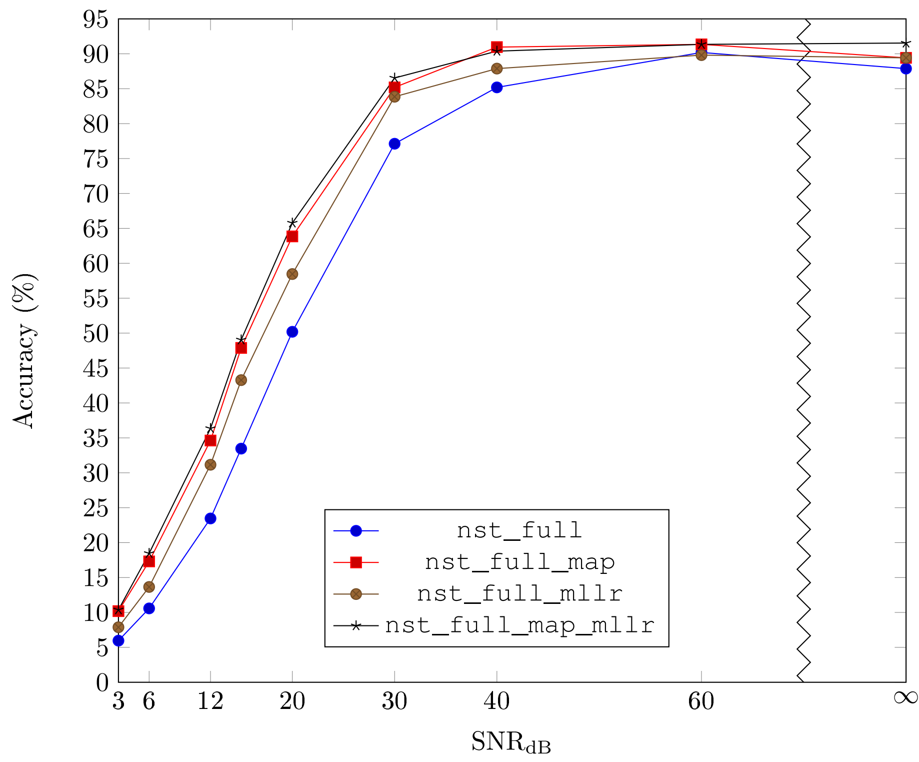


Figure 4.3: Accuracies for all models trained on the full NST, including benchmarks and adaptations.

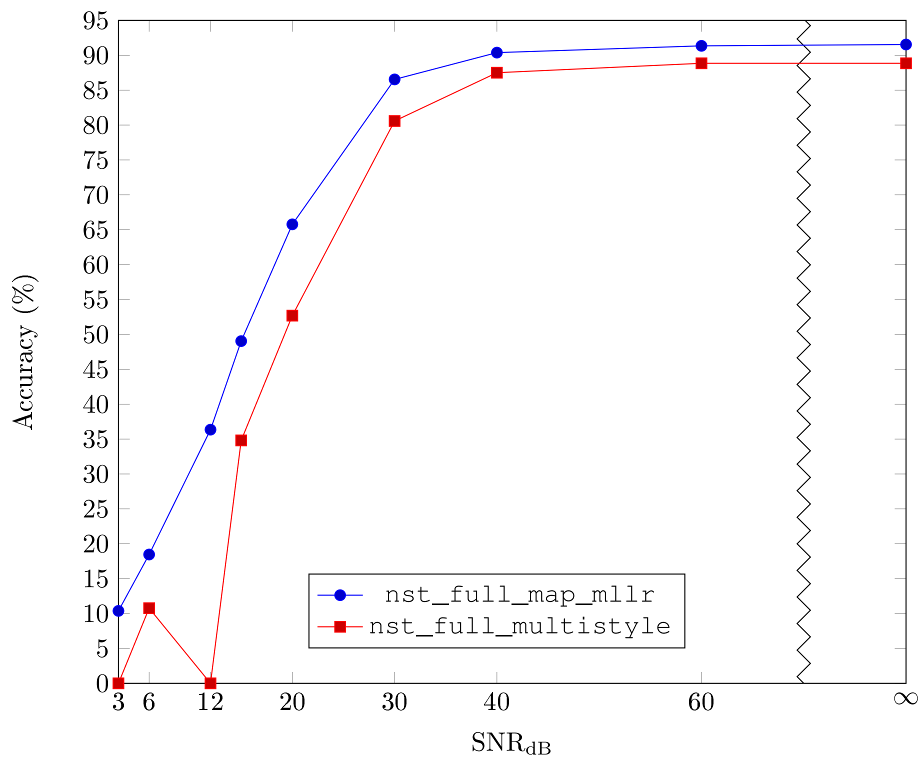


Figure 4.4: Accuracies for multistyle training compared to the best model from adaptations.

4.2.5 Language Model WERs

For the language model evaluations, WER makes more sense as a metric. The following figures plot the LM WER equivalents of the plots in the previous sections.

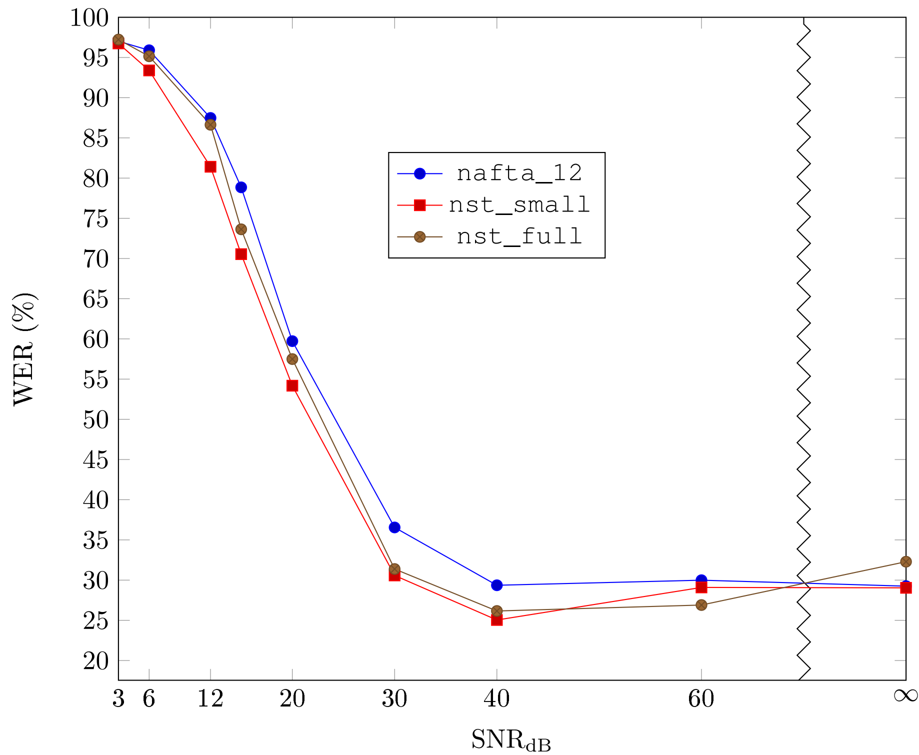


Figure 4.5: WERs for the three benchmark models

4.2.6 Out-of-grammar classifications

To get an impression of how far off the out-of-grammar (unk) classifications seen in the CM above are, recognition using an LM was performed on some of the samples that Sphinx couldn't classify while using the grammar. Table 4.2 shows the results.

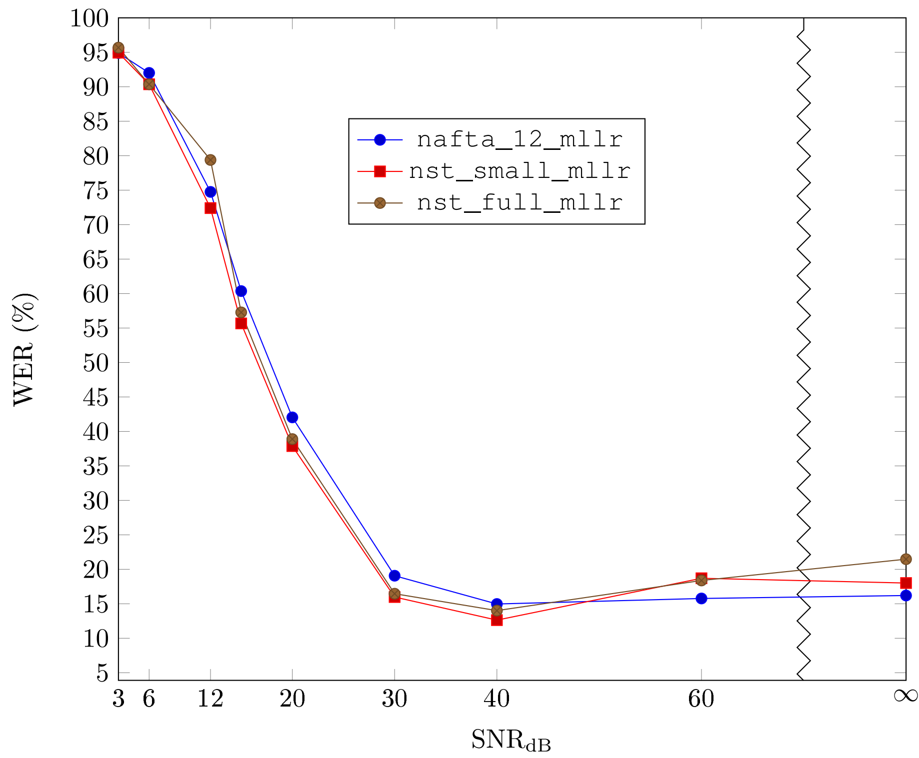


Figure 4.6: WERs for the three models after MLLR

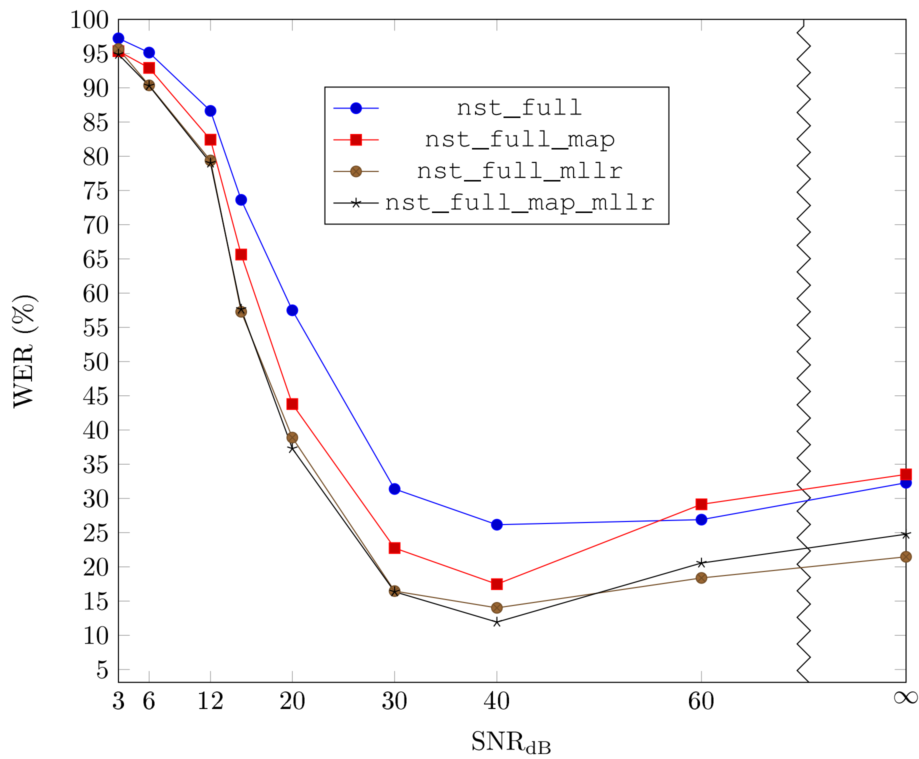


Figure 4.7: WERs for all models trained on the full NST, including benchmarks and adaptations.

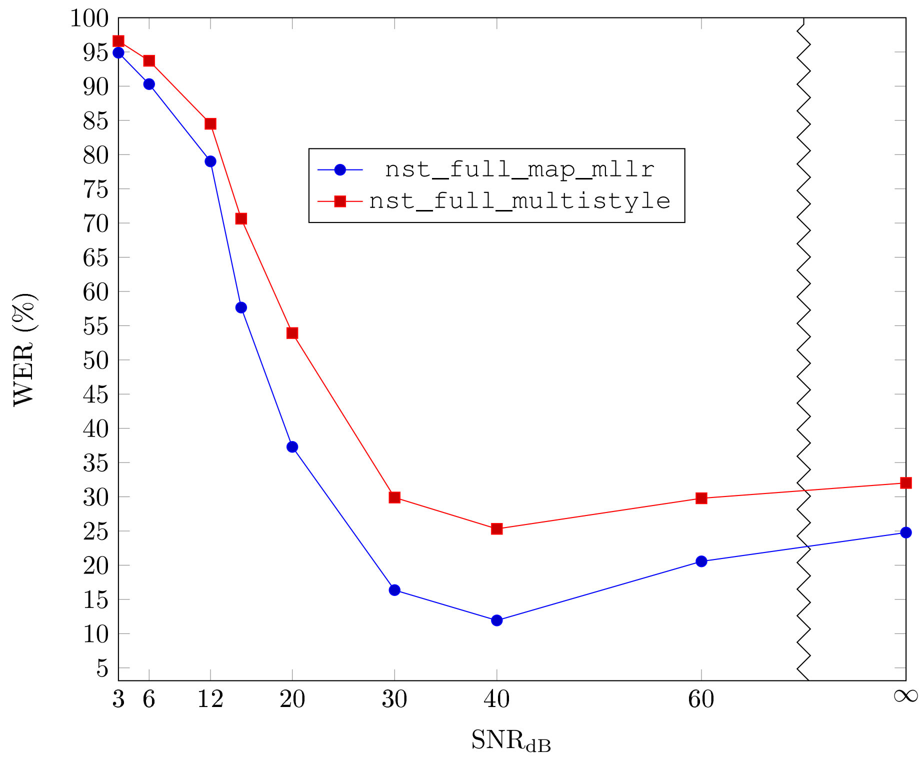
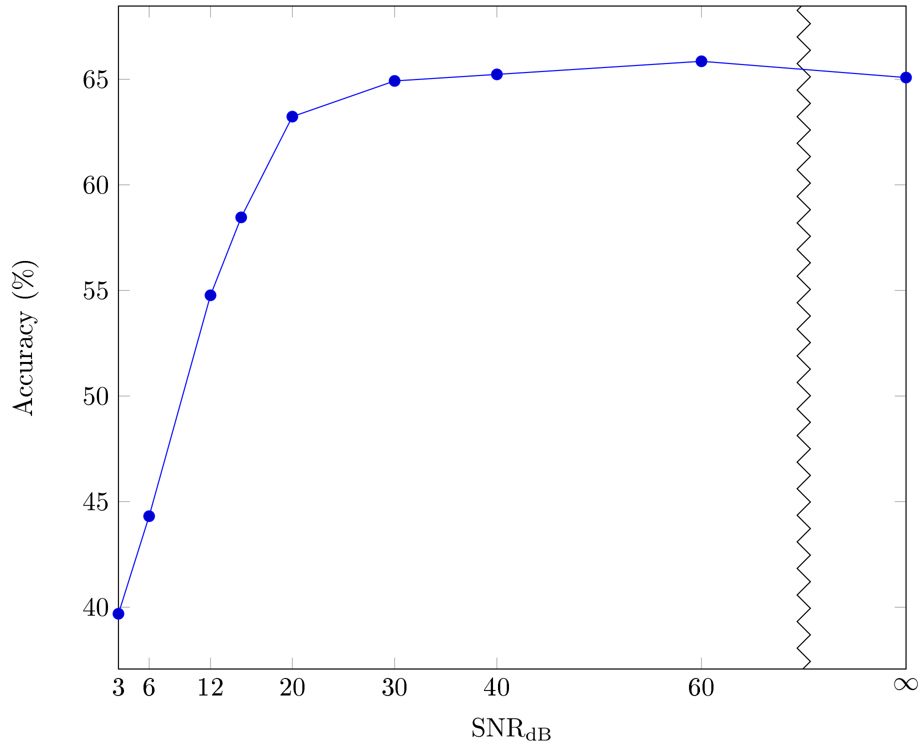


Figure 4.8: WERs for multistyle training compared to the best model from adaptations.

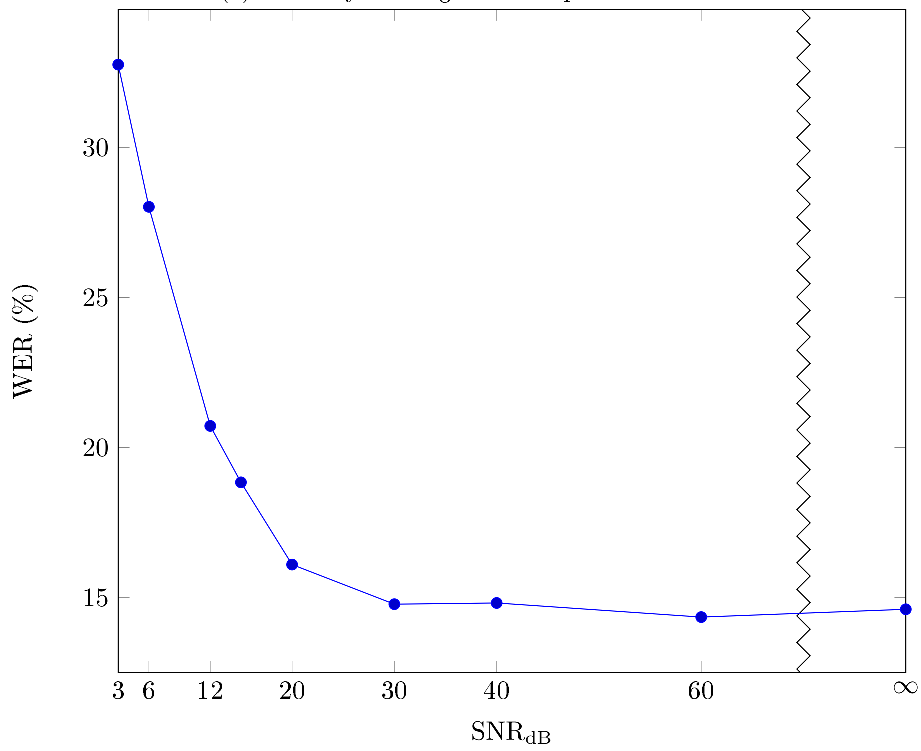
Table 4.2: True text and predicted text when using a language model to transcribe samples that couldn't be transcribed when using a grammar.

True	**Predicted**
slå av relé to	av relé to
slå av armatur	også her driver
slå av relé én	lokalmiljøet
slå av relé tre	slå av relé én drøye
slå av relé tre	slå av relé én drøye
slå på relé én	slå på relé én en
slå på relé tre	det forelå den
slå av armatur	slå opp armatur
slå av armatur	alternativer
slå av relé tre	slå opp elevtallet
slå på relé én	slå på relé én
slå av lyset	aviser
slå av relé én	slå av relé tre den
slå på relé tre	slå på relé én tredje
slå av relé tre	av relé tre
slå på relé to	logg på relé to
slå av relé tre	lag ruller tre

4.3 Google Cloud Speech



(a) Accuracy of Google Cloud Speech-to-Text



(b) WER of Google Cloud Speech-to-Text

Figure 4.9: Google Speech performance measures.

Chapter 5

Discussion

5.1 GMM / HMM

5.1.1 Benchmarks

The first results in figure 4.1 seem promising for clean speech signals. As expected, the performance decreases significantly with increased noise levels. Something worth noticing is the surprisingly little difference between the benchmark models trained on different databases. `nst_full` barely performs better than the two others, despite being trained on almost 10 times more data. For the language model evaluations shown in figure 4.5, the difference is also small, but even more surprisingly, `nst_small` seems to perform best out of the three models. Common for all the models is that none of them perform well in the presence of noise. An utterance classification accuracy of around 5% and a WER of over 95% for an SNR of 3 decibels means that the model is not better than a model that randomly assigns each utterance to one of the 11 classes (including `unk`).

5.1.2 Adaptation

The results of MLLR, as seen in figure 4.2, shows that this simple adaptation method improves the performance of all models noticeably. It also makes the model performance even more similar between the three models. This indicates that the adaptation to noisy data has a greater impact than adding more training data, at least in this case. This might be due to the fact that the databases used have such small variations in environment and speaking style that adding more data from the same database doesn't really bring any new knowledge to the model.

MAP adaptation, as evident by figure 4.3, improves performance further, just as the theory will suggest. Keeping in mind that the MAP adaptation performed here did not include any data directly from `luxdb`, it is safe to say that the model was adapted to the environmental noise, and not the testing data. MLLR adaptation of the already MAP adapted model did not seem to improve performance significantly, which suggests that `nst_full` cannot be improved by further adaptation using the existing noise recordings. Although adaptations did improve model performance, the resulting models can still not be said to be noise robust.

5.1.3 Multistyle Training

The limited multistyle experiment plotted in figure 4.4 shows that this attempt was not really successful, and that it is of more use to train on clean data and adapt to noisy data later.

5.1.4 Improvement Potential

As is evident by the confusion matrix in table 4.1, a significant portion of the clips could not be classified by the Sphinx4 decoder. Even though all audio samples used for evaluation contains utterances found in the grammar, some of them are classified as unknown, which means that Sphinx thinks they're out-of-grammar. There are a couple of ways this could be improved upon

in practice. One could perform additional analysis on all utterances that are classified as `<unk>`, for instance using a different form of decoding constraint. Looking at the results from language model evaluations in table 4.2, several transcriptions are close to being correct, and some actually are. One way of improving accuracy could therefore be to use a decoder with a language model to further analyze all utterances that are rejected by the grammar decoder.

Other points of confusion include the words "av" and "på" and "én" and "tre". One possible remedy for the "på" / "av" confusion, although not speech recognition related, is to use information about the specific system that the ASR system is used for. In the case of the `luxgrammar`, the probability of an utterance being `slå på lyset` (turn on the light) should be zero when the lights are already on. If the system classifies the utterance as `slå på lyset` when the lights are on, the correct transcription is most probably `slå av lyset` and vice versa. This will work for `slå på / av armatur` too, but not necessarily when the system is confusing `relé én` with `relé tre` (relay one and relay three), as those are different relays that might not depend on each other. This problem must be addressed through the improvement of the SR system itself, eventually implementing some kind of special case model for solving this type of confusion.

5.1.5 Sphinx4 vs. Pocketsphinx

After training the acoustic model with Sphinxtrain, the WER is automatically evaluated by a script that uses the Pocketsphinx decoder. Although the results from this evaluation were not saved, the few that were observed seemed to be better than what was achieved with Sphinx4. This is consistent with the findings by Gaida et al. [8], which in short are that Pocketsphinx performs better than Sphinx4. Future work could be done on evaluating the model performance using Pocketsphinx instead of Sphinx4.

5.2 Google Cloud Speech

The plots in figure 4.9 show that `gspeech` performs good in general. It is, however, less accurate than the GMM/HMM models for cleaner speech, as seen in the comparison with the best NST-trained model in figure 5.1. This is expected, as `gspeech`, in contrast to the Sphinx system, does not use a grammar constraint. From figure 5.2, it seems like the constraint is the most crucial feature for Sphinx to perform better than the Google system for lower levels of noise. For higher levels of noise, `gspeech` is clearly superior, even to Sphinx using a grammar. In other words, this is the most noise robust system out of the two.

Google proving superior is not a big surprise. Their systems consist of neural networks trained on huge amounts of data, probably a proper multistyle training consisting of multiple dialects, noise levels and all other thinkable factors.

5.3 Production Considerations

Sphinx adds complexity to the system. Although Sphinx4 is Java based and fits well into the pipeline, its integration means another potential point of failure. For a startup with speech recognition as anything other than first priority, this might not be worth the effort. Delegating all the speech processing to Google or another provider therefore seems to be a better alternative in some ways. This, however, raises other concerns, most notably about pricing and privacy, which the company has to take into account.

5.4 Limitations

In addition to the narrow focus on environmental noise, which does not take into account microphone or reverberations, nor variations in accents and dialects, the experiments have some other specific limitations worth addressing. This section is dedicated to those limitations.

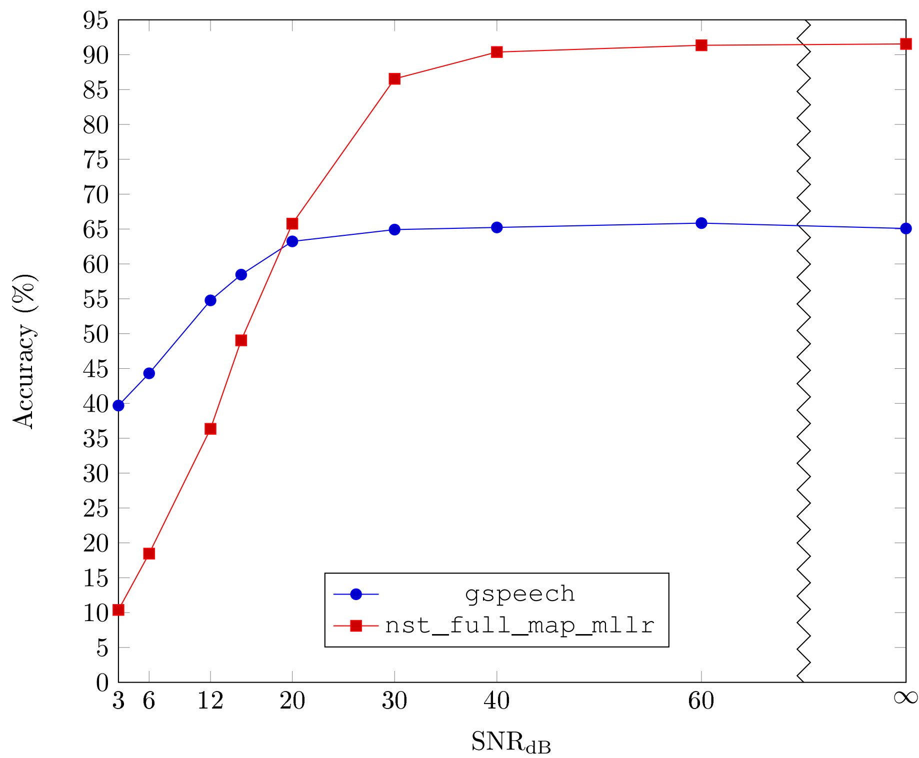


Figure 5.1: The most successful Sphinx model compared to GSpeech

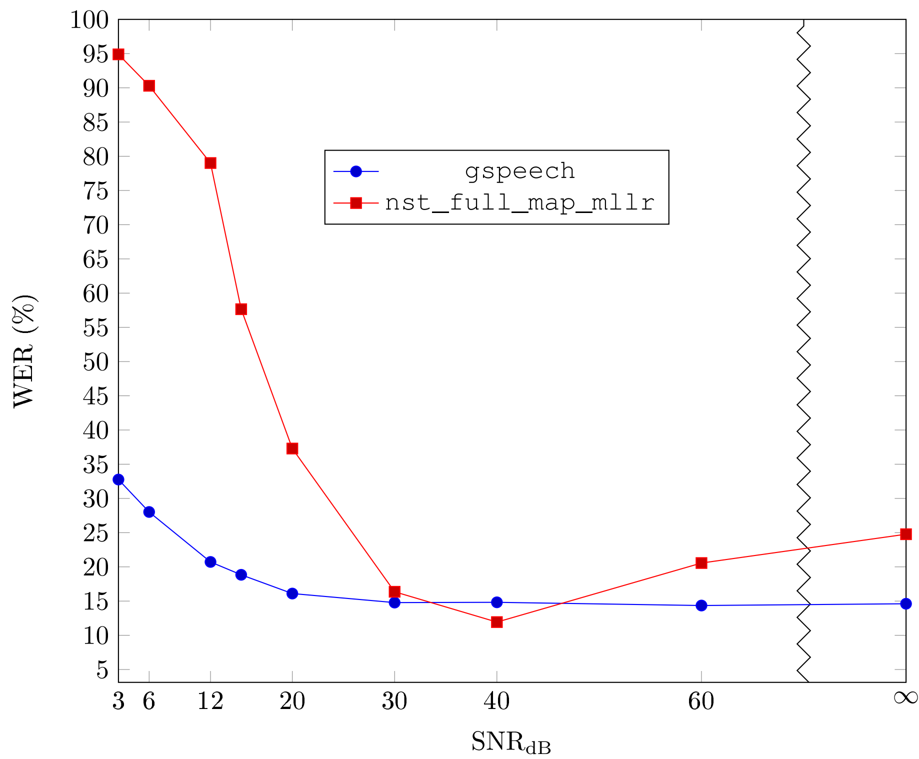


Figure 5.2: The most successful Sphinx model used with a language model, compared to GSpeech.

5.4.1 Testing Database

There are several problems with the database collected and used for model evaluations, some of which are mentioned in section 3.1.4. The "self-service" recording interface makes it hard to control the environment that recordings are performed in. Although the most obviously corrupted recordings were removed from the database, there is probably a significant amount of background noise present in some of the remaining samples too. Despite the web interface, there were also relatively few participants. Many of the speakers have gone through the recording process many times, so there is enough data to get an impression of the performance of the models evaluated, but a higher amount of speakers and samples in general would be preferable.

Another factor in the recording process worth mentioning is that it doesn't account for the *Lombard effect*. This effect, in short, means that speakers tend to subconsciously raise their voice levels when they experience background noise. This is very often accompanied by a change in the spectral contents of the speech signal too [38], something that might be significant for speech recognition.

Some of the problems mentioned could be solved by performing the data collection in a more controlled way. An example could be to record the speakers inside an an-echoic chamber while they listen to environmental noise on headphones. This would eliminate environmental noise in the clean speech recording and induce the Lombard effect.

The acquisition of noise data also had its shortcomings. All noise data was recorded using one of two microphones, and the amount of data is limited. Most of the speech recordings were performed with different microphones than those that were used for noise recordings. This means that the corruption is not that realistic, as it doesn't take into account the channel. The small amount of noise also lead to reuse of the samples, both in adaptation and testing. This could have had a significant impact on the evaluations of the adapted models, because it is possible that they adapted to the specific noise clips and not to the noise in general. For similar evaluations in the future, this could be handled by dividing noise data into training and testing parts.

5.4.2 Evaluator Problems

As seen by the results of multistyle training, the Sphinx4 based `Evaluator` ran into problems with some audio files. In these cases the Sphinx decoder threw a `Too many UTTERANCE_START` exception. In addition to the 3 and 12 dB cases for multistyle, this seemed to happen for *all* audio samples with an SNR of 9 or 10, which is why those are not included in the evaluations. The reason for the errors could not be identified, because not enough time was available for thorough debugging. Despite these problems, which would be important to fix in a final product, the evaluator has been sufficient for researching how the given models perform.

5.4.3 Other

MLLR adaptation was only attempted on noisy samples from the `luxdb`. It would make sense to see if an adaptation to clean data is just as effective, or if noisy adaptation data is essential for performance improvements.

5.5 Future work

5.5.1 General Evaluation of Sphinx

It would be interesting to evaluate the noise robustness of the pre-trained English model that is included with Sphinx decoders. Assuming that the model is well trained, this could reveal whether Sphinx is able to compete with Google Speech when it comes to noise robust speech recognition. This could be done completely without collection any new speech data, for instance using the open speech data-set *Speech Commands* [35].

5.5.2 Recurrent Neural Networks

Limited time and computational resources lead to CTC being an infeasible technique for the scope of this thesis. In the future, more work could be put into training a neural network using CTC. Using one or multiple GPUs instead of a single CPU for training, and using all the speech data available in NST and other Norwegian databases, better results will probably be achieved. For a properly trained network, it would also be interesting to implement a form of *constrained decoding*, as explained in Graves' 2012 book [11]. In the context of small-vocabulary recognition, a grammar like the one used in the Sphinx experiments would be a natural constraint. It could be interesting to see how different kinds of neural networks, including hybrid models, perform with such a grammar in general, and how robust they would be to noise compared to GMM/HMMs.

5.5.3 Denoising

Although the main goal of this thesis has been to find a noise robust `classifier`, the experiments have focused on evaluating the performance and robustness of different `acoustic models` based on GMM/HMMs. As it is hard to evaluate the performance of a system by looking at a single component in isolation, future work may look at other methods of improving robustness in a system, like pre-processing in the form of `denoising`, e.g. using auto-encoders.

5.6 Resource Intensity

The thesis introduction states that the objective is to find which methods of speech classification has the most favorable trade-off between noise robustness and computational resource intensity. However, only a single kind of self-implemented system was successfully evaluated here. Considering the fact that the self-implemented systems based on neural networks were not able to recognize a single word, it wouldn't matter how light on resources it is. It is obvious that using a third-party service is the choice that is least demanding of computational resources, so this was not interesting to evaluate either. Resource intensity, therefore, has not been given much attention in this thesis, and will have to be a topic for future work.

Chapter 6

Conclusion

RNN based and traditional GMM/HMM based speech recognition systems have been trained on big Norwegian speech databases. While the RNN based models proved too demanding to train properly, the traditional models were trained on clean speech data and later adapted to speech data corrupted by street noise. The utterance classification accuracies and word error rates of models have been evaluated for noise simulated at different signal-to-noise ratios, in order to see how robust the models are to such noise. For comparison, a commercially available speech recognition system was also evaluated.

The self-trained models, used in conjunction with a predefined grammar, seemed to perform well for clean speech, but quickly deteriorated with increasing levels of noise. The points of confusion for clean speech were easy to spot in a confusion matrix, and could probably be fixed with some further engineering of the system. The confusions for noisy speech, however, are too many to fix on a case-by-case basis, and indicates that the system is probably not sufficiently robust for a real-life scenario. It is hard to say whether the GMM/HMM systems give a good trade-off between noise robustness and resource intensity, because no other systems were possible to evaluate for comparison, except the commercial one. This system without a doubt proved more stable and noise robust than any of the self-trained models.

To find the most favorable classifier for noise robust small-vocabulary recognition, which is the academic focus of this thesis, more research is needed. The choice of system between the self-trained and commercial one, which is more interesting from the business perspective, comes down to practical considerations regarding pricing and privacy of the commercial system, and the resources available for further developing a self-made system.

Bibliography

[1] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. In *International Conference on Machine Learning*, pages 173–182, 2016.

[2] Anaconda. Anaconda Software Distribution. `https://anaconda.com`. [Website, accessed 2019-2-5].

[3] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48. ACM, 2009.

[4] Carl Boettiger. An introduction to docker for reproducible research. *ACM SIGOPS Operating Systems Review*, 49(1):71–79, 2015.

[5] CMU Sphinx. Adapting the default acoustic model. `https://cmusphinx.github.io/wiki/tutorialadapt/`. [Website, accessed 2019-3-12].

[6] CMU Sphinx. Building a language model. `https://cmusphinx.github.io/wiki/tutoriallm/`. [Website, accessed 2019-3-13].

[7] CMU Sphinx. CMU Sphinx Documentation. `https://cmusphinx.github.io/wiki/`. [Website, accessed 2019-2-5].

[8] Christian Gaida, Patrick Lange, Rico Petrick, Patrick Proba, Ahmed Malatawy, and David Suendermann-Oeft. Comparing open-source speech recognition toolkits. *Tech. Rep., DHBW Stuttgart*, 2014.

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[10] Google. Google Cloud-to-Text basics. `https://cloud.google.com/speech-to-text/docs/basics`. [Website, accessed 2019-2-2].

[11] Alex Graves. Supervised sequence labelling. In *Supervised sequence labelling with recurrent neural networks*, pages 5–13. Springer, 2012.

[12] Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd international conference on Machine learning*, pages 369–376. ACM, 2006.

[13] Awni Hannun. Sequence modeling with ctc. *Distill*, 2017. `https://distill.pub/2017/ctc`.

[14] Xuedong Huang, Alex Acero, and Hsiao-Wuen Hon. *Spoken language processing: A guide to theory, algorithm, and system development*, volume 1.

[15] Ali Shariq Imran, Vetle Haflan, Abdolreza Sabzi Shahrebabaki, Negar Olfati, and Torbjørn Karl Svendsen. Evaluating feature map representations in 2d-cnn for speaker identification. 2019.

[16] National Instruments. Understanding ffts and windowing. http://download.ni.com/evaluation/pxi/Understanding%20FFTs%20and%20Windowing.pdf. [Link accessed 2019-2-3].

[17] Dan Jurafsky and James H Martin. Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition, 2009.

[18] Veton Këpuska. Wake-up-word speech recognition. In *Speech Technologies*. InTech, 2011.

[19] R Lippmann, Edward Martin, and D Paul. Multi-style training for robust isolated-word speech recognition. In *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP'87.*, volume 12, pages 705–708. IEEE, 1987.

[20] Nvidia. CUDA Toolkit. https://developer.nvidia.com/cuda-toolkit. [Website, accessed 2019-2-5].

[21] National Library of Norway. Nb tale - a basic acoustic phonetic speech database for norwegian. https://www.nb.no/sprakbanken/show?serial=oai%3Anb.no%3Asbr-31&lang=en. [Website, accessed 2019-2-8].

[22] National Library of Norway. Nst acoustic speech database for norwegian. https://www.nb.no/sprakbanken/show?serial=oai%3Anb.no%3Asbr-13&lang=en. [Website, accessed 2019-2-8].

[23] National Library of Norway. Nst lexical database for norwegian bokmål. https://www.nb.no/sprakbanken/show?serial=oai%3Anb.no%3Asbr-23&lang=en. [Website, accessed 2019-2-16].

[24] National Library of Norway. Språkbanken - a language technology resource collection for norwegian. https://www.nb.no/en/forskning/sprakbanken/. [Website, accessed 2019-2-8].

[25] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. Librispeech: an asr corpus based on public domain audio books. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pages 5206–5210. IEEE, 2015.

[26] Sebastian Raschka. Machine learning faq. https://sebastianraschka.com/faq/docs/multiclass-metric.html. [Website, accessed 2019-3-17].

[27] Baidu Research. ba-dls-deepspeech. https://github.com/baidu-research/ba-dls-deepspeech, 2017. [Website, accessed 2019-2-10].

[28] Baidu Research. Warp-ctc. https://github.com/baidu-research/warp-ctc, 2018. [Website, accessed 2019-2-10].

[29] Md Sahidullah and Goutam Saha. Design, analysis and experimental evaluation of block based transformation in mfcc computation for speaker recognition. *Speech Communication*, 54(4):543–565, 2012.

[30] The HTK Team. HTK FAQ. http://htk.eng.cam.ac.uk/docs/faq.shtml. [Website, accessed 2019-2-5].

[31] The HTK Team. HTK Speech Recognition. http://htk.eng.cam.ac.uk/. [Website, accessed 2019-2-5].

[32] Martin Thoma. Word error rate calculation. https://martin-thoma.com/word-error-rate-calculation/, 2013. [Website, accessed 2019-3-8].

[33] Aäron Van Den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *CoRR abs/1609.03499*, 2016.

[34] W3C. JSpeech Grammar Format. `https://www.w3.org/TR/jsgf/`. [Website, accessed 2019-3-13].

[35] Pete Warden. Speech commands: A dataset for limited-vocabulary speech recognition. *arXiv preprint arXiv:1804.03209*, 2018.

[36] Steve Young, Gunnar Evermann, Mark Gales, Thomas Hain, Dan Kershaw, Xunying Liu, Gareth Moore, Julian Odell, Dave Ollason, Dan Povey, et al. *The HTK book*, volume 3. Cambridge Univ. Press, 2002.

[37] Dong Yu and Li Deng. *Automatic Speech Recognition - A Deep Learning Approach*. Springer, 2015.

[38] Sue Anne Zollinger and Henrik Brumm. The lombard effect. *Current Biology*, 21(16):R614–R615, 2011.

Appendix A

Detailed Results

A.1 Complete Tables of Results

Table A.1 lists the resulting accuracies from all evaluations performed, both with and without using the `luxgrammar`. Table A.2 lists resulting WERs from evaluations performed using language models only.

Table A.1: Classification accuracy (%) of all models evaluated for different levels of noise.

Model \SNR$_{dB}$	3	6	12	15	20	30	40	60	∞
lm_nafta_12_mllr	2.88	4.42	13.08	20.96	36.92	67.31	74.81	73.65	73.08
lm_nst_full_multistyle	1.73	2.69	7.12	14.23	29.23	57.5	62.5	57.31	53.85
nst_small_mllr	8.08	14.42	29.81	40.77	58.46	82.5	88.08	89.23	87.12
nst_full_map_mllr	10.38	18.46	36.35	49.04	65.77	86.54	90.38	91.35	91.54
lm_nst_full_map	2.69	4.42	10.0	18.85	35.96	66.73	74.42	63.08	57.69
nst_full_multistyle	0.0	10.77	0.0	34.81	52.69	80.58	87.5	88.85	88.85
nst_full_mllr	7.88	13.65	31.15	43.27	58.46	83.85	87.88	89.81	89.42
lm_nst_small_mllr	3.27	5.58	15.77	25.77	42.5	72.69	77.31	70.77	69.81
nafta_12	5.58	9.23	20.0	30.19	45.0	71.15	81.73	86.54	86.73
nst_full	5.96	10.58	23.46	33.46	50.19	77.12	85.19	90.19	87.88
nafta_12_mllr	7.69	15.19	32.88	44.42	59.81	84.04	88.27	88.65	89.42
lm_nst_small	1.73	2.88	8.85	14.04	29.04	56.35	64.04	60.77	58.65
nst_small	5.58	9.23	20.58	31.73	47.5	74.42	84.81	87.88	86.73
lm_nst_full	1.54	2.31	5.77	12.31	26.73	54.62	61.35	59.23	54.62
nst_small_multistyle	7.88	14.42	30.58	41.15	55.38	78.65	80.96	80.58	78.27
lm_nst_full_mllr	2.12	5.19	11.54	23.46	39.81	70.19	75.38	68.85	64.42
lm_nst_full_map_mllr	2.88	6.54	12.5	24.81	41.92	73.27	80.38	70.96	66.73
nst_full_map	10.19	17.31	34.62	47.88	63.85	85.19	90.96	91.35	89.42

Table A.2: WERs for models evaluated using a language model instead of grammar

Model \SNR$_{dB}$	3	6	12	15	20	30	40	60	∞
nst_full_multistyle	96.59	93.71	84.5	70.64	53.92	29.89	25.31	29.78	32.02
nafta_12_mllr	94.89	92.01	74.75	60.36	42.04	19.07	14.97	15.77	16.2
nst_full_mllr	95.68	90.36	79.38	57.27	38.89	16.46	14.01	18.38	21.47
nst_small	96.75	93.39	81.41	70.54	54.18	30.58	25.04	29.09	29.04
nst_full	97.23	95.15	86.63	73.63	57.49	31.38	26.16	26.9	32.29
nst_full_map	95.36	92.91	82.42	65.64	43.79	22.75	17.47	29.14	33.51
nst_full_map_mllr	94.89	90.3	79.01	57.65	37.29	16.36	11.93	20.56	24.77
nst_small_mllr	94.94	90.36	72.4	55.67	37.88	15.98	12.63	18.7	18.01

A.2 Confusion Matrices

Tables A.3 to A.11 show confusion matrices for all evaluations performed on `nst_full`. 162 more matrices were produced, but are not included, as they don't provide much information beyond what is already seen here.

Table A.3: nst_full_3_db

T \P	AA	AL	AR2	AR3	AR1	PA	PL	PR2	PR3	PR1	unk
AA	3	0	0	0	1	2	0	0	0	0	51
AL	0	9	0	0	0	0	0	0	0	0	43
AR2	0	0	2	0	1	0	0	0	0	1	45
AR3	0	0	0	3	2	0	0	0	0	0	49
AR1	0	0	0	2	4	0	0	0	0	2	43
PA	0	0	0	0	0	2	0	0	0	0	49
PL	0	0	0	0	0	0	3	0	0	0	40
PR2	0	0	0	0	0	0	0	2	0	1	50
PR3	0	0	0	1	0	0	0	0	0	2	51
PR1	0	0	0	1	1	0	0	0	0	3	51

Table A.4: nst_full_6_db

T \P	AA	AL	AR2	AR3	AR1	PA	PL	PR2	PR3	PR1	unk
AA	7	0	0	0	1	3	0	0	0	0	46
AL	0	15	0	0	0	0	1	0	0	0	36
AR2	0	0	3	1	4	0	0	1	0	1	39
AR3	0	0	0	4	2	0	0	0	0	0	48
AR1	0	0	0	1	11	0	0	0	1	0	38
PA	0	0	0	0	0	4	0	0	0	0	47
PL	0	1	0	0	0	0	3	0	0	0	39
PR2	0	0	0	0	0	0	0	1	0	1	51
PR3	0	0	0	2	1	0	0	0	1	1	49
PR1	0	0	0	0	2	0	0	0	0	6	48

Table A.5: nst_full_12_db

T \P	AA	AL	AR2	AR3	AR1	PA	PL	PR2	PR3	PR1	unk
AA	22	0	0	0	1	3	0	0	0	0	31
AL	0	28	0	0	0	0	0	0	0	0	24
AR2	0	0	7	2	4	0	0	2	0	1	33
AR3	0	0	0	16	4	0	0	0	2	0	32
AR1	0	0	0	0	17	0	0	0	0	3	31
PA	3	0	0	0	0	7	0	0	0	0	41
PL	0	1	0	0	0	0	8	0	0	0	34
PR2	0	0	0	0	1	0	0	2	0	1	49
PR3	0	0	0	2	0	0	0	0	4	1	47
PR1	0	0	0	0	2	0	0	0	0	11	43

Table A.6: nst_full_15_db

T \P	AA	AL	AR2	AR3	AR1	PA	PL	PR2	PR3	PR1	unk
AA	27	0	0	0	0	4	0	0	0	0	26
AL	0	34	0	0	0	0	1	0	0	0	17
AR2	0	0	14	1	4	0	0	2	0	3	25
AR3	0	0	0	21	4	0	0	0	2	0	27
AR1	0	1	0	1	28	0	0	0	0	1	20
PA	6	0	0	0	0	11	0	0	0	0	34
PL	0	1	0	0	0	0	12	0	0	0	30
PR2	0	0	0	0	1	0	0	6	0	0	46
PR3	0	0	0	3	1	0	0	0	8	3	39
PR1	0	0	0	1	4	0	0	0	0	13	38

Table A.7: nst_full_20_db

T \P	AA	AL	AR2	AR3	AR1	PA	PL	PR2	PR3	PR1	unk
AA	38	0	0	0	0	5	0	0	0	0	14
AL	0	40	0	0	0	0	2	0	0	0	10
AR2	0	0	23	0	4	0	0	2	0	1	19
AR3	0	0	0	31	5	0	0	0	1	0	17
AR1	0	2	0	1	32	0	0	0	0	2	14
PA	3	0	0	0	0	23	0	0	0	0	25
PL	0	1	0	0	0	0	25	0	0	0	17
PR2	0	0	1	0	1	0	0	13	0	3	35
PR3	0	0	0	1	1	0	0	0	13	5	34
PR1	0	0	0	0	2	0	0	0	0	23	31

Table A.8: nst_full_30_db

T \P	AA	AL	AR2	AR3	AR1	PA	PL	PR2	PR3	PR1	unk
AA	46	0	0	0	0	5	0	0	0	0	6
AL	0	45	0	0	0	0	2	0	0	0	5
AR2	0	0	37	0	2	0	0	2	0	0	8
AR3	0	0	0	42	1	0	0	0	1	0	10
AR1	0	0	0	0	40	0	0	0	0	1	10
PA	3	0	0	0	0	39	0	0	0	0	9
PL	0	0	0	0	0	0	40	0	0	0	3
PR2	0	0	2	0	0	0	0	33	0	4	14
PR3	0	0	0	2	0	0	0	0	34	6	12
PR1	0	0	0	0	0	0	0	0	0	45	11

Table A.9: nst_full_40_db

T \P	AA	AL	AR2	AR3	AR1	PA	PL	PR2	PR3	PR1	unk
AA	49	0	0	0	0	4	0	0	0	0	4
AL	0	48	0	0	0	0	2	0	0	0	2
AR2	0	0	41	0	1	0	0	4	0	0	3
AR3	0	0	0	41	6	0	0	0	2	0	5
AR1	0	1	0	0	45	0	0	0	0	1	4
PA	3	0	0	0	0	46	0	0	0	0	2
PL	0	0	0	0	0	0	41	0	0	0	2
PR2	0	0	1	0	0	0	0	48	0	1	3
PR3	0	0	0	3	0	0	0	0	35	12	4
PR1	0	0	0	0	2	0	0	0	0	49	5

Table A.10: nst_full_60_db

T \P	AA	AL	AR2	AR3	AR1	PA	PL	PR2	PR3	PR1	unk
AA	52	0	0	0	0	2	0	0	0	0	3
AL	0	52	0	0	0	0	0	0	0	0	0
AR2	0	0	44	0	2	0	0	2	0	0	1
AR3	0	0	0	45	6	0	0	0	1	0	2
AR1	0	0	0	0	49	0	0	0	0	0	2
PA	8	0	0	0	0	43	0	0	0	0	0
PL	0	0	0	0	0	0	41	0	0	0	2
PR2	0	0	2	0	0	0	0	48	1	0	2
PR3	0	0	0	1	1	0	0	0	43	9	0
PR1	0	0	0	0	3	0	0	0	0	52	1

Table A.11: nst_full_clean

T \P	AA	AL	AR2	AR3	AR1	PA	PL	PR2	PR3	PR1	unk
AA	50	0	0	0	0	3	0	0	0	0	4
AL	0	51	0	0	0	0	0	0	0	0	1
AR2	0	0	44	0	1	0	0	2	0	1	1
AR3	0	0	0	43	9	0	0	0	0	0	2
AR1	0	0	0	0	50	0	0	0	0	0	1
PA	8	0	0	0	0	42	0	0	0	0	1
PL	0	2	0	0	0	0	38	0	0	0	3
PR2	0	0	2	0	0	0	0	46	0	0	5
PR3	0	0	0	2	2	0	0	0	41	8	1
PR1	0	0	0	0	3	0	0	0	0	52	1

A.3 F_1 scores

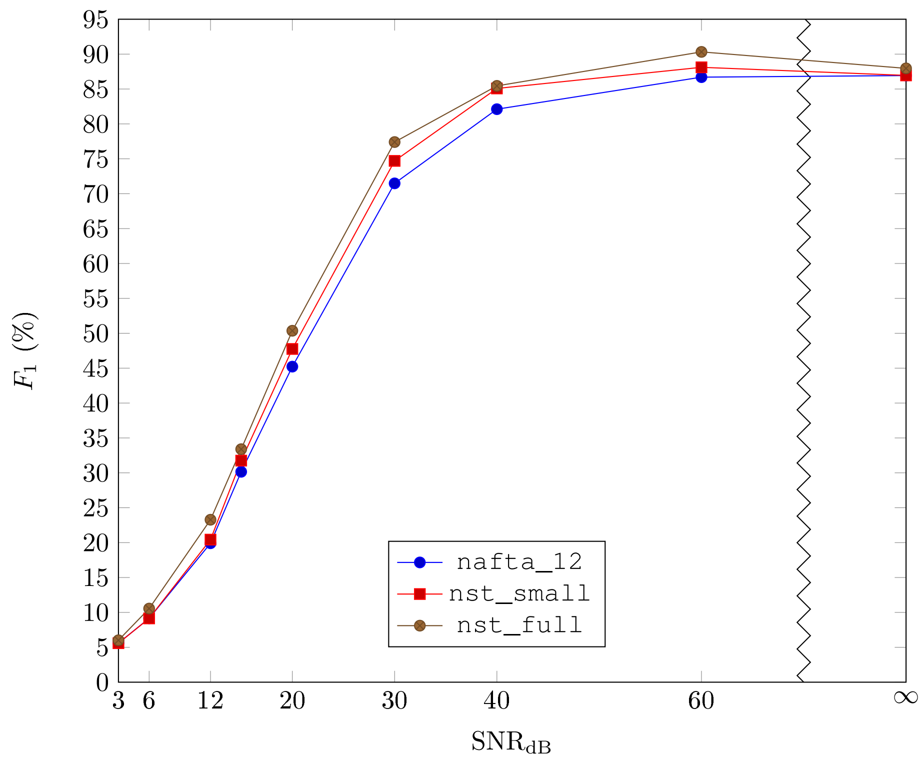


Figure A.1: F_1 scores for the three benchmark models

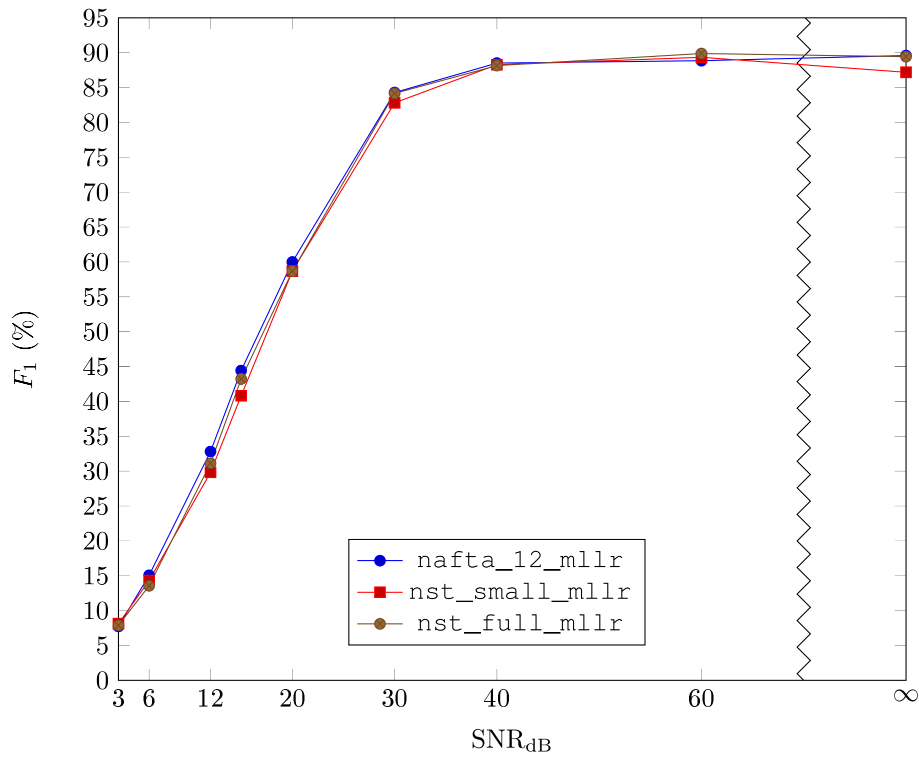


Figure A.2: F_1 scores for the three benchmark models after MLLR

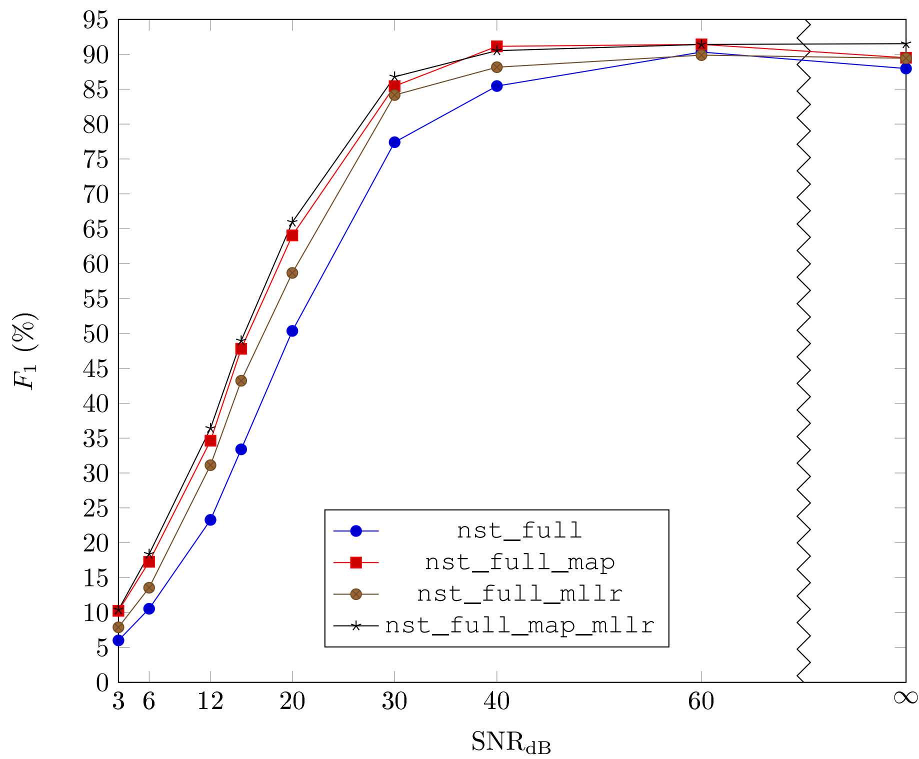


Figure A.3: F_1 scores for all models trained on the full NST, including benchmarks and adaptations.

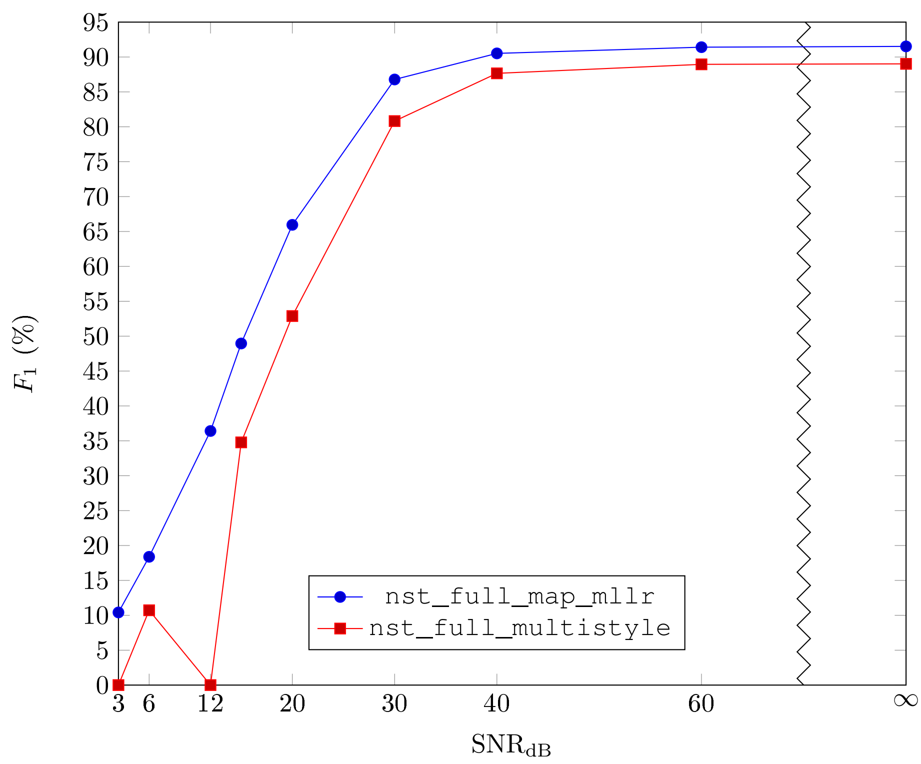


Figure A.4: F_1 scores for multistyle training compared to the best model from adaptations.

Appendix B

Data Preparation Scripts

B.1 Audio Processing

The script in listing B.1 was used to mix all speech audio in one directory with noise from another directory using a certain SNR.

Listing B.1: noisify.py

```
#!/usr/bin/python3

import soundfile
import argparse
import os
from math import sqrt, log10
import numpy as np

INT16_MAX = 2**15-1
INT16_MIN = -2**15

def open_audio(audiofile):
    with soundfile.SoundFile(audiofile) as sf:
        audio = sf.read(dtype="int16")
        sr = sf.samplerate
        assert sr == 16000
    return audio

def get_amp_factor(speech_power, noise_power, snr):
    return sqrt(speech_power/(snr*noise_power))

def get_sig_power(signal):
    sigsum = 0
    mu = sum(signal)/len(signal)
    for x in signal:
        sigsum += (x - mu)**2
    return sigsum/len(signal)

def noisify_sf(speech_sig, noise_sig, snr):
    speech_power = get_sig_power(speech_sig)
    noise_power = get_sig_power(noise_sig)
    a = get_amp_factor(speech_power, noise_power, snr)
    noise_sig = np.array(a*noise_sig, dtype="int32")
    scaled_noise_power = get_sig_power(noise_sig)
    mixed_signal = np.clip(noise_sig+speech_sig, INT16_MIN, INT16_MAX)
    # Clipping the signal, then converting back to int16
    return np.array(mixed_signal, dtype="int16")

def main(speechdir, noisedir, outdir, snr):
    speech_filenames = [wavfile for wavfile in os.listdir(speechdir) if ".wav" in
        wavfile ]
    noise_filenames  = os.listdir(noisedir)
    noise_sigs = []
    for noise_fn in noise_filenames:
```

```
            noise_sigs.append(open_audio(os.path.join(noisedir, noise_fn)))
    noiseindex = 0
    nc = 0  # noise clip (sample) number
    num_sfs = len(speech_filenames)
    num_noiseclips = len(noise_sigs)
    for i, speech_fn in enumerate(speech_filenames):
        print("Noisifying file {}/{} using noise clip {}/{} ({})".format(
                i+1, num_sfs, nc+1, num_noiseclips, noise_filenames[nc]), end='\r')
        speech = open_audio(os.path.join(speechdir, speech_fn))
        while noiseindex + len(speech) >= len(noise_sigs[nc]):
            nc = (nc + 1) % num_noiseclips
            noiseindex = 0
        noisy_speech = noisify_sf(speech, noise_sigs[nc][noiseindex:noiseindex+len(
            speech)], snr)
        noiseindex += len(speech)
        soundfile.write(os.path.join(outdir, speech_fn), noisy_speech, 16000)


if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument("speechdir", type=str, help="Directory of speech audio files
        ")
    parser.add_argument("noisedir", type=str, help="Directory of noise")
    parser.add_argument("snrdb", type=int, help="SNR in dB Power")
    args = parser.parse_args()
    snr = 10**(args.snrdb/10)

    speechdir = args.speechdir
    if speechdir[-1] == '/':
        speechdir = speechdir[:-1]
    outdir = speechdir + "_" + str(args.snrdb) + "_dbsnr"
    os.makedirs(outdir, exist_ok=True)
    print("Adding noise. '{}' -> '{}'".format(speechdir, outdir))
    main(args.speechdir, args.noisedir, outdir, snr)
```

Appendix C

Evaluation Code

C.1 Java Evaluator Source Code

Listing C.1: Evaluator.java

```
package com.luxsave.eval;

import edu.cmu.sphinx.api.Configuration;
import edu.cmu.sphinx.api.StreamSpeechRecognizer;

import java.io.*;
import java.nio.file.Files;
import java.util.*;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Evaluator {

    private static ArrayList<String> listDir(String dirPath){
        ArrayList<String> wavFiles = new ArrayList<>();
        try {
            Files.list(new File(dirPath).toPath()).forEach(path -> {
                wavFiles.add(path.toString());
            });
        } catch (IOException e){
            System.out.println("Couldn't open audio directory.");
        }
        return wavFiles;
    }

    private static Map<String, String> getFileIdTranscriptionPairs(String filename,
        String wavRoot){
        Pattern transRE = Pattern.compile("<s> *(.*) *</s>");
        Pattern fidRE = Pattern.compile("[(](.*)[)]");
        try {
            BufferedReader reader = new BufferedReader(new FileReader(filename));
            String line;
            HashMap<String, String> fidTransPair = new HashMap<>();
            while ((line = reader.readLine()) != null){
                Matcher transM = transRE.matcher(line);
                Matcher fidM = fidRE.matcher(line);
                if (transM.find() && fidM.find()) {
                    String newTrans = transM.group(1);
                    String newFileId = wavRoot + fidM.group(1) + ".wav";
                    fidTransPair.put(newFileId, newTrans);
                }
            }
            return fidTransPair;
        } catch (Exception e) {
            System.out.println(e);
            return null;
```

```
        }
    }

    private static void saveFile(String filename, String text){
        try {
            PrintWriter fout = new PrintWriter(filename);
            fout.println(text);
            fout.close();
        } catch (FileNotFoundException e){
            System.out.println("Error while saving to file");
            System.out.println(e.toString());
        }

    }

    public static void main(String[] args) {

        String modelPath;
        String wavPath;
        Map<String, String> fileIdTranscriptionMap;
        if (args.length < 2){
            System.out.println("Path to model and wav root dir not given. Exiting..."
                );
            return;
        }
        modelPath = args[0];
        if (!modelPath.endsWith("/")) {
            modelPath += "/";
        }
        wavPath = args[1];
        if (!wavPath.endsWith("/")) {
            wavPath += "/";
        }
        String transFile = modelPath + "luxdb_test.transcription";
        fileIdTranscriptionMap = getFileIdTranscriptionPairs(transFile, wavPath);
        boolean noUseGrammar = false;
        if (args.length == 3 && args[2].contentEquals("--nogrammar")){
            noUseGrammar = true;
        }
        File f = new File(modelPath + "mllr_matrix");
        /* Use MLLR matrix if it exists */
        boolean useMLLR = false;
        if(f.exists() && !f.isDirectory()){
            useMLLR = true;
        }
        if (fileIdTranscriptionMap == null){
            System.out.println("Error in transcription file. Exiting.");
            return;
        }
        Classifier classifier = new Classifier(fileIdTranscriptionMap);
        Configuration configuration = new Configuration();
        configuration.setAcousticModelPath("file:" + modelPath);
        if (noUseGrammar){
            configuration.setLanguageModelPath("file:" + modelPath + "nst.lm.bin");
            configuration.setDictionaryPath("file:" + modelPath + "nst.dic");
        } else {
            configuration.setUseGrammar(true);
            configuration.setGrammarName("luxgrammar");
            configuration.setGrammarPath("file:" + modelPath);
            configuration.setDictionaryPath("file:" + modelPath + "dictionary");
        }
        StreamSpeechRecognizer recognizer;
        try {
            recognizer = new StreamSpeechRecognizer(configuration);
            if (useMLLR) {
                recognizer.loadTransform(modelPath + "mllr_matrix", 1);
            }
        } catch (Exception e) {
            System.out.println("Unable to initialize speech recognizer");
            e.printStackTrace();
```

```
            return;
        }

        List<String> wavFiles = new ArrayList<>(fileIdTranscriptionMap.keySet());
        try {
            for (String wav : wavFiles){
                String detectedWord;
                try {
                    detectedWord = (new SpeechProcessor(recognizer, wav)).
                        recognizeOnce();
                } catch (Error e){
                    System.out.println("Error recognizing utterance. Setting to <unk
                        >." + e.toString());
                    detectedWord = "<unk>";
                }
                classifier.classify(wav, fileIdTranscriptionMap.get(wav),
                    detectedWord);
            }
        } catch (Exception e) {
            System.out.println("OTHER EXCEPTION: " + e.toString());
            e.printStackTrace();
        }
        /* Generate results file with confusion matrix */
        String resultsString = classifier.getConfusionMatrixCSV();
        resultsString += "\nACC: " + classifier.getAccuracy() + "\n";
        System.out.print(resultsString);
        saveFile(modelPath + "results.csv", resultsString);

        /* Generate complete transcription / classification JSON file */
        ArrayList<String> classPairs = classifier.allClassificationPairs();
        StringBuilder classPairsOutput = new StringBuilder();
        for (String line : classPairs){
            classPairsOutput.append(line).append("\n");
        }
        saveFile(modelPath + "classifications.json", classPairsOutput.toString());
    }
}
```

Listing C.2: SpeechProcessor.java

```
package com.luxsave.eval;

import edu.cmu.sphinx.api.SpeechResult;
import edu.cmu.sphinx.api.StreamSpeechRecognizer;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.util.ArrayList;
import java.util.List;

public class SpeechProcessor {
    private String wavFile;
    private StreamSpeechRecognizer recognizer;

    public SpeechProcessor(StreamSpeechRecognizer recognizer, String wavFileToRead) {
        this.wavFile = wavFileToRead;
        this.recognizer = recognizer;
    }

    public String recognizeOnce(){
        try {
            InputStream stream = new FileInputStream(new File(this.wavFile));
            recognizer.startRecognition(stream);
            SpeechResult result;
            StringBuilder recognizedStringBuilder = new StringBuilder();
            while ((result = recognizer.getResult()) != null) {
                recognizedStringBuilder.append(result.getHypothesis());
```

```
                recognizedStringBuilder.append(" ");
            }
            recognizer.stopRecognition();
            return recognizedStringBuilder.toString();
        } catch (IOException e) {
            System.out.println(e.toString());
            return null;
        }
    }
}
```

Listing C.3: Classifier.java

```
package com.luxsave.eval;

import java.util.*;

public class Classifier {
    private ArrayList<String> stringClasses;
    private Map<String, String> fileIdTrueClass = new HashMap<>();
    private Map<String, String> fileIdPredictedClass = new HashMap<>();
    private Map<String, Integer> sClassToIntMap = new HashMap<>();
    private Map<Integer, String> intToSCLassMap = new HashMap<>();
    private Map<String, String> classificationExceptions = new HashMap<>();
    private int[][] confusionMatrix;

    public Classifier(Map<String, String> fidTranscriptionPair) {
        this.stringClasses = uniqueClassificationStrings(new ArrayList<>(
            fidTranscriptionPair.values()));
        Collections.sort(this.stringClasses);
        this.stringClasses.add("<unk>");
        int numClasses = this.stringClasses.size();
        this.confusionMatrix = new int[numClasses][numClasses];
        for (int i = 0; i < numClasses; i++){
            this.sClassToIntMap.put(stringClasses.get(i), i);
            this.intToSCLassMap.put(i, stringClasses.get(i));
            for (int j = 0; j < numClasses; j++){
                this.confusionMatrix[i][j] = 0;
            }
        }
    }

    private static String sanitizeHypothesis(String hypothesis){
        if (hypothesis.contains("<unk>")){
            return "<unk>";
        }
        String[] transcription = hypothesis.replace("<sil>", "")
                .split(" ");
        StringBuilder transcriptionBuilder = new StringBuilder();
        for (String word : transcription){
            if (!word.isEmpty()){
                transcriptionBuilder.append(word).append(" ");
            }
        }
        return transcriptionBuilder.toString();
    }

    public void classify(String fileId, String trueClass, String hypothesis){
        String predictedClass = sanitizeHypothesis(hypothesis);
        if (!this.sClassToIntMap.containsKey(predictedClass)){
            classificationExceptions.put(fileId, predictedClass);
            predictedClass = "<unk>";
        }
        int r = this.sClassToIntMap.get(trueClass);
        int c = this.sClassToIntMap.get(predictedClass);
        this.fileIdTrueClass.put(fileId, trueClass);
        this.fileIdPredictedClass.put(fileId, predictedClass);
        this.confusionMatrix[r][c]++;
    }
```

```
    public String getConfusionMatrixCSV(){
        String topRow = "class,AA,AL,ARTO,ARTRE,AREN,PA,PL,PRTO,PRTRE,PREN,unk";
        StringBuilder csvMatrix = new StringBuilder(topRow);
        for (int r = 0; r < stringClasses.size(); r++){
            csvMatrix.append("\n").append(intToSCLassMap.get(r));
            for (int c = 0; c < stringClasses.size(); c++){
                csvMatrix.append(",").append(Integer.toString(confusionMatrix[r][c]))
                    ;
            }
        }
        return csvMatrix.toString();
    }


    private static ArrayList<String> uniqueClassificationStrings(ArrayList<String>
        allClassificationStrings){
        ArrayList<String> unique = new ArrayList<>();
        for (String sClass : allClassificationStrings){
            if (!unique.contains(sClass)){
                unique.add((sClass));
            }
        }
        return unique;
    }

    public double getAccuracy() {
        int numTotalTranscriptions = this.fileIdTrueClass.size();
        int numCorrectPredictions = 0;
        for (int i = 0; i < this.stringClasses.size(); i++) {
            numCorrectPredictions += this.confusionMatrix[i][i];
        }
        System.out.println("Correctly classified: " + numCorrectPredictions + "/" +
            numTotalTranscriptions);
        return (double) 100 * numCorrectPredictions / numTotalTranscriptions;
    }

    public ArrayList<String> allClassificationPairs(){
        ArrayList<String> classificationPairs = new ArrayList<>();
        for (Map.Entry<String, String> fidTrue: fileIdTrueClass.entrySet()) {
            String fid = fidTrue.getKey();
            String trueClass = fidTrue.getValue();
            String predictedClass = fileIdPredictedClass.get(fid);
            if (this.classificationExceptions.containsKey(fid)){
                /* Do not duplicate non-grammar transcriptions */
                continue;
            }
            if (!predictedClass.equals(trueClass)){
                classificationPairs.add("{ \"key\":\"" + fid + "\", \"true\":\"" +
                    trueClass + "\", \"predicted\":\"" + predictedClass + "\", \"info
                    \":\"misclassified\" }");
            } else {
                classificationPairs.add("{ \"key\":\"" + fid + "\", \"true\":\"" +
                    trueClass + "\", \"predicted\":\"" + predictedClass + "\", \"info
                    \":\"correct\" }");
            }
        }
        for (Map.Entry<String, String> fidErroneous : classificationExceptions.
            entrySet()){
            String fid = fidErroneous.getKey();
            String trueClass = this.fileIdTrueClass.get(fid);
            String predictedClass = fidErroneous.getValue();
            classificationPairs.add("{ \"key\":\"" + fid + "\", \"true\":\"" +
                trueClass + "\", \"predicted\":\"" + predictedClass + "\", \"info
                \":\"out_of_grammar\" }");
        }
        return classificationPairs;
    }
}
```

C.2 Shell Scripts

This section contains scripts used for the final model evaluations. Script C.4 was used to perform the final evaluations using the Sphinx4 evaluator with the `luxgrammar`. In case noise clips with the desired SNR does not exist, it runs script B.1 to create them before evaluation. After evaluations, the result files are sorted in results directories for later analysis.

Listing C.4: noise_eval.sh

```
#!/bin/bash

MODELNAME="$1"

NOISIFIER="noisify.py" # assumes directory of noisify.py being added to PATH
EVALUATOR="java -jar ./evaluator/target/eval-1.0-SNAPSHOT-jar-with-dependencies.jar"

DIR_CLEANSPEECH="../2_data/luxdb/mono16"
DIR_NOISES="../2_data/luxdb/noise16"
DIR_EVALMODEL="./$MODELNAME"

SNRs=(3 6 12 15 20 30 40 60)
# Includes 60 for control. Results should be similar to clean speech.

# First get the clean speech benchmark
$EVALUATOR $DIR_EVALMODEL $DIR_CLEANSPEECH
mv "$DIR_EVALMODEL/results.csv" "./all_results/${MODELNAME}_clean.csv"
mv "$DIR_EVALMODEL/classifications.json" "./all_classifications/${MODELNAME}_clean.
    json"

for snr in ${SNRs[*]}; do
    # Noisify with the right SNR if not already done
    DIR_NOISYSPEECH="${DIR_CLEANSPEECH}_${snr}_dbsnr"
    if [ "$2" = "clean" ]; then rm -r $DIR_NOISYSPEECH; fi
    if [ ! -d "$DIR_NOISYSPEECH" ]; then
        $NOISIFIER $DIR_CLEANSPEECH $DIR_NOISES $snr
    fi
    $EVALUATOR $DIR_EVALMODEL $DIR_NOISYSPEECH
    mv "$DIR_EVALMODEL/results.csv" "./all_results/${MODELNAME}_${snr}_db.csv"
    mv "$DIR_EVALMODEL/classifications.json" \
       "./all_classifications/${MODELNAME}_${snr}_db.json"
done
```

Script C.5 is similar to C.4 , but uses a language model instead of the grammar.

Listing C.5: lm_eval.sh

```
#!/bin/bash

MODELNAME="$1"

NOISIFIER="noisify.py"
EVALUATOR="java -jar ./evaluator/target/eval-1.0-SNAPSHOT-jar-with-dependencies.jar"

DIR_CLEANSPEECH="../2_data/luxdb/mono16"
DIR_NOISES="../2_data/luxdb/noise16"
DIR_EVALMODEL="./$MODELNAME"

SNRs=(3 6 12 15 20 30 40 60)

# First get the 'clean' speech benchmark
$EVALUATOR $DIR_EVALMODEL $DIR_CLEANSPEECH --nogrammar
mv "$DIR_EVALMODEL/results.csv" "./all_results/lm_${MODELNAME}_clean.csv"
mv "$DIR_EVALMODEL/classifications.json" "./all_classifications/lm_${MODELNAME}_clean
    .json"


for snr in ${SNRs[*]}; do
    # Noisify with the right SNR if not already done
    DIR_NOISYSPEECH="${DIR_CLEANSPEECH}_${snr}_dbsnr"
    if [ "$2" = "clean" ]; then rm -r $DIR_NOISYSPEECH; fi
    if [ ! -d "$DIR_NOISYSPEECH" ]; then
        $NOISIFIER $DIR_CLEANSPEECH $DIR_NOISES $snr
    fi
    $EVALUATOR $DIR_EVALMODEL $DIR_NOISYSPEECH --nogrammar
    mv "$DIR_EVALMODEL/results.csv" "./all_results/lm_${MODELNAME}_${snr}_db.csv"
    mv "$DIR_EVALMODEL/classifications.json" \
       "./all_classifications/lm_${MODELNAME}_${snr}_db.json"
done
```

Appendix D

Setup Scripts

D.1 CMU Sphinxtrain Setup

The Dockerfile in listing D.1 should describe everything necessary to install Sphinxtrain. Running `docker build -t cmusphinx .` in the directory of this file will build a Docker image with a working installation that can be run as a Docker container.

Listing D.1: Sphinxtrain `Dockerfile`

```
FROM debian
RUN apt-get -yqq update
RUN apt-get -yqq install autoconf make wget gcc bison file perl python python-dev
    swig libtool git python3
WORKDIR /opt/

# Install sphinxbase
RUN git clone https://github.com/cmusphinx/sphinxbase.git; \
    cd ./sphinxbase; \
    git checkout 8d1bf98;\
    ./autogen.sh; ./configure; make clean all; make check; make install; \
    cd /opt/

# Install pocketsphinx
RUN git clone https://github.com/cmusphinx/pocketsphinx.git; \
    cd pocketsphinx; \
    git checkout 600fe3e; \
    ./autogen.sh; ./configure; make clean all; make check; make install; \
    cd /opt/

# Install sphinxtrain
RUN git clone https://github.com/cmusphinx/sphinxtrain.git; \
    cd ./sphinxtrain; \
    git checkout eb8bfba; \
    ./autogen.sh; ./configure; make clean all; make check; make install

# Install CMUCLMTK for making a language model, and sphinxbase-utils for making
    binary LM
RUN wget https://sourceforge.net/projects/cmusphinx/files/cmuclmtk/0.7/cmuclmtk-0.7.
    tar.gz; \
    tar -xzf cmuclmtk-0.7.tar.gz; cd ./cmuclmtk-0.7; \
    ./configure; make check; make install; \
    apt-get -yqq install sphinxbase-utils; \
    cd /opt/

# Set all necessary environment variables
RUN echo "export LD_LIBRARY_PATH=/usr/local/lib" >> /root/.bashrc; \
    echo "export PKG_CONFIG_PATH=/usr/local/lib/pkgconfig" >> /root/.bashrc; \
    echo "export PATH=/usr/local/bin:$PATH" >> /root/.bashrc; \
    echo "alias train=/root/train.sh" >> /root/.bashrc

WORKDIR /root/
```

D.2 WarpCTC Conda Setup

With `Anaconda` installed on a Linux or Darwin machine, the script seen in listing D.2 can be run to achieve a setup similar to the one used for the CTC experiments in section 3.4. The only step that should be remaining after this setup is to configure `Keras` to use `Theano` as back-end, which is usually done by adding `{"back-end": "theano"}` to `~/.keras/keras.json`. This script should also be available at `https://gitlab.com/snippets/1797979`, but is included here in case it disappears in the future.

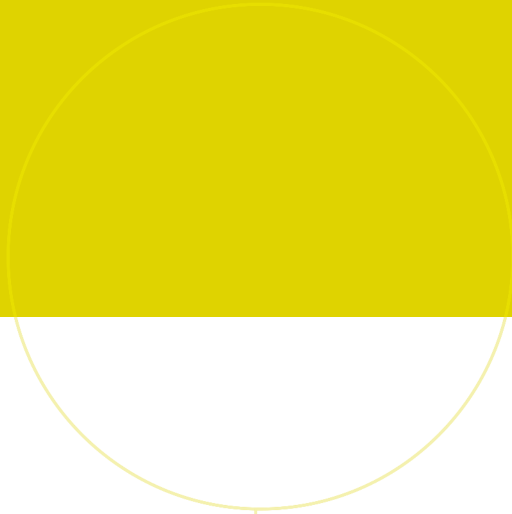
Listing D.2: ctc_setup.sh template

```
#!/bin/bash

CONDAENV=warpctc

conda create -n $CONDAENV python=2.7
conda install -n $CONDAENV cmake make

source activate $CONDAENV
pip install 'keras==1.1.2' 'theano==0.8.2' lasagne 'scipy==0.18.1' \
    'numpy==1.15.4' soundfile futures
git clone https://github.com/sherjilozair/ctc.git theano-warp-ctc/
mkdir theano-warp-ctc/build
cd theano-warp-ctc/build
cmake ..
make
cd ../python
python2 setup.py install
```



NTNU
Norwegian University of
Science and Technology