MASTER'S THESIS

Swedish dialect classification using Artificial Neural Networks and Gaussian Mixture Models

DAVID LIDBERG
VIKTOR BLOMQVIST

Department of Mathematical Sciences
Division of Applied Mathematics and Statistics
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2017

Thesis for the Degree of Master of Science

Swedish dialect classification using Artificial Neural Networks and Gaussian Mixture Models

David Lidberg
Viktor Blomqvist

Department of Mathematical Sciences
Division of Applied Mathematics and Statistics
Chalmers University of Technology and University of Gothenburg
SE – 412 96 Gothenburg, Sweden
Gothenburg, August 2017

Swedish Dialect Classification using
Artificial Neural Networks and Gaussian Mixture Models

David Lidberg
Viktor Blomqvist

Supervisor: Adam Andersson, Syntronic Software Innovations
Supervisor & Examiner: Mihály Kovács, Department of Mathematical Sciences

Master's Thesis 2017
Department of Mathematical Sciences
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2017

Swedish Dialect Classification using
Artificial Neural Networks and Gaussian Mixture Models

David Lidberg
Viktor Blomqvist

Department of Mathematical Sciences
Chalmers University of Technology

Abstract

Variations due to speaker dialects are one of the main problems in automatic speech recognition. A possible solution to this issue is to have a separate classifier identify the dialect of a speaker and then load an appropriate speech recognition system. This thesis investigates classification of seven Swedish dialects based on the SweDia2000 database. Classification was done using Gaussian mixture models, which are a widely used technique in speech processing. Inspired by recent progress in deep learning techniques for speech recognition, convolutional neural networks and multi-layered perceptrons were also implemented. Data was preprocessed using both mel-frequency coefficients, and a novel feature extraction technique using path signatures. Results showed high variance in classification accuracy during cross validations even for simple models, suggesting a limitation in the amount of available data for the classification problems formulated in this project. The Gaussian mixture models reached the highest accuracy of 61.3% on test set, based on singe-word classification. Performance is greatly improved by including multiple words, achieving around 80% classification accuracy using 12 words.

Keywords: Swedish, SweDia2000, dialect classification, Gaussian mixture models, convolutional networks, artificial neural networks, deep learning, path signatures.

Acknowledgements

First and foremost we want to express our gratitude to our supervisors Adam Andersson and Mihály Kovács for continued support and feedback throughout working with this project. We give thanks to the people behind the Swedia2000 project for the data used in this thesis, in particular Anders Eriksson who let us use his maps for our illustrations. Furthermore we want to thank Syntronic for providing us with the opportunity to work with this project, as well as the department of Mathematical Sciences at Chalmers for letting us use their computational resources.

Viktor Blomqvist and David Lidberg, Gothenburg, June 2017

Contents

List of Figures

List of Tables

Chapter 1

Introduction

There are several factors that affect human speech. There are speaker-to-speaker variations — humans can often identify a person from their voice based solely on a few words. Age, gender and emotional state are other factors that have an impact on speech [2]. One of the challenges in automatic speech recognition (ASR) is to overcome such sources of variation. This thesis focuses on a particular factor, namely the dialects of speakers. Dialects can come with different pronunciations of words and phones, tone of voice and even completely different words, and can therefore have significant impact on performance of speech recognition systems [3]. Dialect classification is an approach to address this difficulty. By first determining the dialect of a speaker, an ASR system adapted for that dialect can be activated and thus improve recognition accuracy.

Previous research on dialect classification have taken various approaches. Classification of American accents based acoustic features of specific words were investigated in [4]. Phonetic context has been used to classify Arabian dialects, where the dialects are identified based the order of phonemes [5]. Both these works are based on Mel cepstral features, which are standard features in many speech processing applications [6]. More linguistical approaches involve vocabularies and lexical features [7].

This thesis investigates Swedish dialect classification, and compares methods for achieving such a classification. The study covers Gaussian mixture models (GMMs), which is a standard technique in ASR and has been used for dialect classification as well [8–10]. In fact, ASR and dialect classification, as well as speaker and gender identification, are generally implemented using similar methods. While GMMs historically have been successful in ASR applications, recently deep artificial neural networks have risen as a strong candidate to replace GMM's [11, 12]. Therefore this thesis also investigates the performance of a convolutional neural network and multi-layered perceptrons in dialect classification.

In addition to the standard features centered around the Mel scale new methods involving the signature transform is tested as feature extraction. The signature is a non-linear transformation with roots in rough paths theory [13]. By calculating the signature of a path one obtains a collection of coefficients that contains properties of that path. Signatures and the closely related log signatures have recently been successfully implemented in Chinese character recognition [14] and studied as a method for sound compression [15]. Whilst the project does not aim to extend the theory behind signatures, deriving and implementing efficient procedures for calculating the transform is a part of the project outcome.

Implementation are written in Python 3 with the exception of a shared library written in C. Scikit-learn[1] and Tensorflow[2] are the main libraries used for classifiers.

1.1 Delimitations

This thesis focuses solely on acoustic speech features, relating more or less directly to the original sound signal. This is partly motivated by the fact that prosodical and similar features are linguistically complex and require thoroughly annotated sound samples.

Furthermore the main part of classification experiments deal with dialect classification based on utterances of single words, or multiple utterances of different words. This is in contrast to spontaneous speech which is perhaps the more common variant of raw data. The incorporation of dialect identification into automatic speech recognition, possibly improving performance, is not tested.

As no examples of Swedish dialect classification with defined dialect regions have been found, seven dialects are presented and used here for the first time.

1.2 Outline

Chapter 2 introduces the data which is later used for experiments, defines dialects and outlines the type of experiments that are possible given the data and dialects. In Chapter 3 the theory behind signatures and log signatures of paths is introduced along with the necessary preliminaries. Expressions for calculating these in practice under some assumptions are also presented.

Thereafter the focus changes to machine learning with Chapter 4 that introduces typical feature extraction methods for speech data and Chapter 5 in which the classifiers used in the thesis are explained. Chapter 6 explains the details of the experiments and the results of these together with comments are presented in Chapter 7. Lastly, Chapter 8 contains conclusions and reflections on the work.

[1] http://scikit-learn.org
[2] https://www.tensorflow.org/

Chapter 2

Dialect Classification

Dialect classification, also called dialect identification, is the problem of detecting and classifying different ways of speaking the same language. A strict interpretation of dialect would be versions of a language with different phrases or words, and accent then meaning variants of pronunciation. In this thesis we use dialect to indicate variance in both word choices and pronounciation.

2.1 Speech Corpus

The data used in this thesis comes from SweDia 2000[1], a research database containing Swedish speech recordings [16]. The data consists of interviews with people at over one hundred different locations in Sweden and Swedish-speaking parts of Finland, as shown in Figure 2.1. At each location there are interviews from, on average, 12 different persons residing in the local area. The interviewed persons are all adult but divided by age and gender. Generally 3 elderly males, 3 elderly females, 3 young males and 3 young females at each location. The average age of in these demographic groups are, 66, 66, 26 and 25 years old respectively.

The database is further divided into four different categories, each one corresponding to a type of interview. The four categories are *quantity*, *prosody*, *wordlist* and *spontaneous*. The first three extract multiple utterances of predefined words. The specific words depends on the category and in some cases the recording location. Out of these three categories, only *wordlist* is used since its content varies less between different recording locations and contains a larger number of words. In contrast to the other three, the *spontaneous* interviews do not follow a script. As the name suggests each interview is actually a spontaneous dialogue between an interviewer and one (sometimes two) interviewee(s).

Most audio files in the database are accompanied by an annotation file containing a transcription of what is said and by whom. These files make it possible to divide audio from the interviewer and interviewee. This is necessary since only the interviewee can be assumed to have a dialect tied to the location where the recording took place. Since transcribing speech is very time consuming not all audio-files in the database have been annotated. The coverage is especially sparse in the spontaneous category, where transcription is very time consuming.

The sound files are in the Waveform Audio File Format (commonly referred to

[1] http://swedat.ling.gu.se/

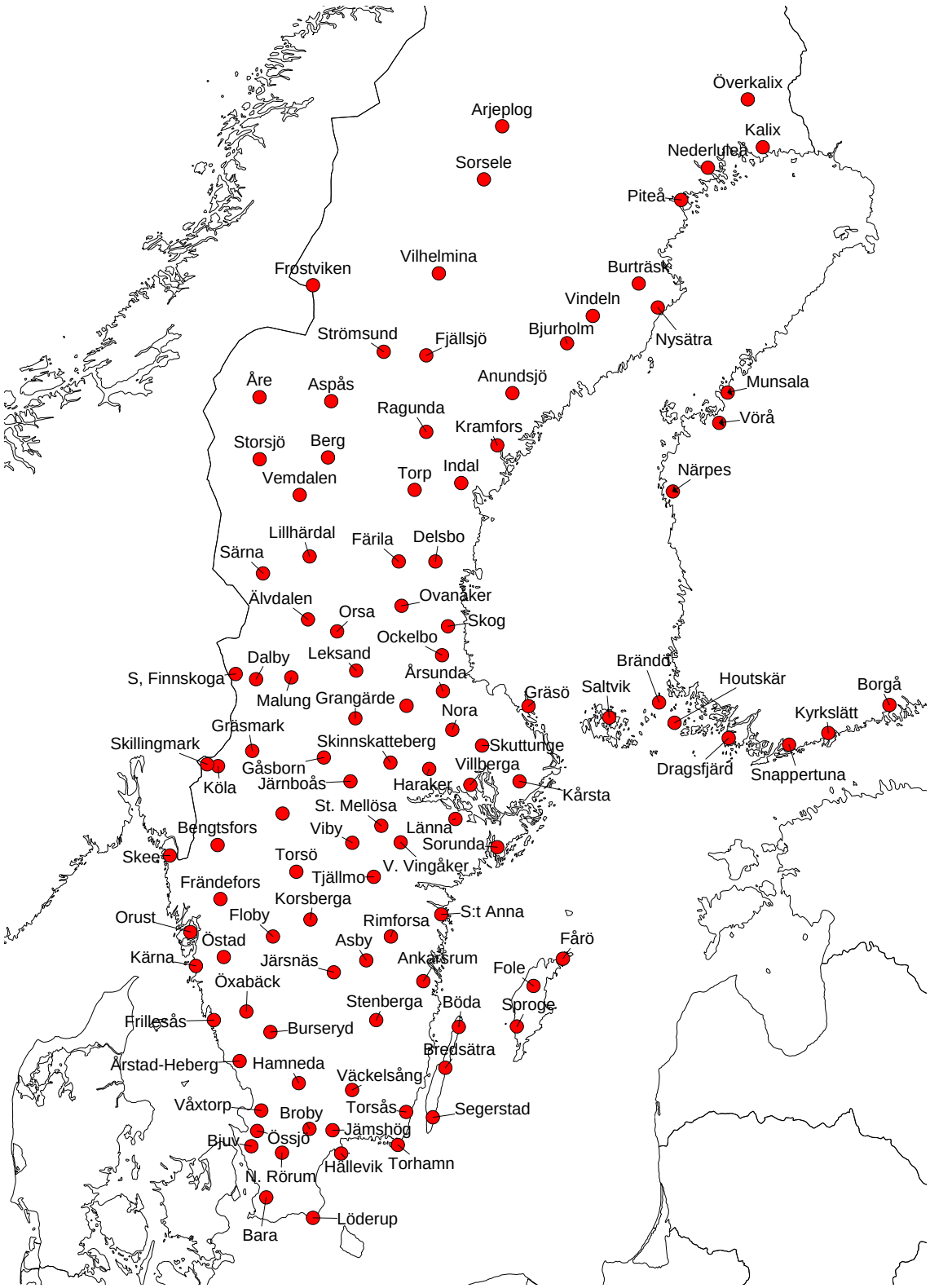

Figure 2.1: SweDia2000 recording locations. From [1]. Reproduced with permission from the copyright holder.

as `wav`) recorded as mono sound (with a few exceptions). The bit depth is 16 bits and sample rate 16kHz.

2.2 Dialect Regions

Clearly defined dialects is a precondition for performing dialect classification. One could argue that every recording location represents its own (albeit very local) dialect. Such micro-dialects are however unfeasible in practice as there would not be enough data available in each dialect.

It is therefore necessary to pool together recording locations in close geographical proximity into larger dialect regions. One such division of the Swedish language area was proposed by Wessén in [17] which divided the area into six regions of varying size. However, pooling into too large regions leads to ambiguity in where to draw the borders between dialects, as there are no clear cut lines between dialects [17].

To get around these problems we define a set of seven dialect regions, partly inspired by Wesséns division. Out of Wesséns six regions, only Gotland is too small, and it is therefore dropped from our definition. Furthermore we have chosen to split the northern part of Sweden in two, as it is geographically the largest. This also provides an opportunity for studying the variance inside what is a single region according to Wessén. We also include the Swedish province Dalarna, located in the central west part of mainland Sweden, as its own dialect region, since it is a distinct and recognizable dialect [18]. This leaves us with the seven dialects/regions: Norrbotten, Jämtland, Dalarna, Sydfinland, Mälardalen, Västkusten and Skåne.

Finally, all regions are not included in full, rather locations from internal subsets of some regions are chosen to represent the dialect, providing a margin between dialect areas and regions of comparable size. In column two of Table A.1 we see that the numbers of unique speakers in each dialect are relatively balanced.

Our seven dialect regions are shown in Figure 2.2. The exact locations included in each dialect is presented in Table 2.1. The discrepancy between the figure and table specifying dialect regions is due to unusable locations in the database. The locations Älvdalen and Orsa, inside the Dalarna region, and Bjuv in Skåne are marked on the map but contain none or very few files in the version of the database we used, and were dropped from the corresponding regions. To compensate for this, Dalarna and Skåne were expanded to balance the number of speakers across dialects, meaning they look larger on the map but they are in fact not larger in data size.

2.3 Classification

As mentioned above, only data from the *wordlist* and *spontaneous* categories are used in this thesis. The annotation coverage in each of the categories across the seven dialects is presented in Table 2.2. Clearly, annotations are not a problem in the *wordlist* category. The situation is as expected less optimistic in *spontaneous*. Based on these data limitations, two different experimental approaches to dialect classification are formulated below.

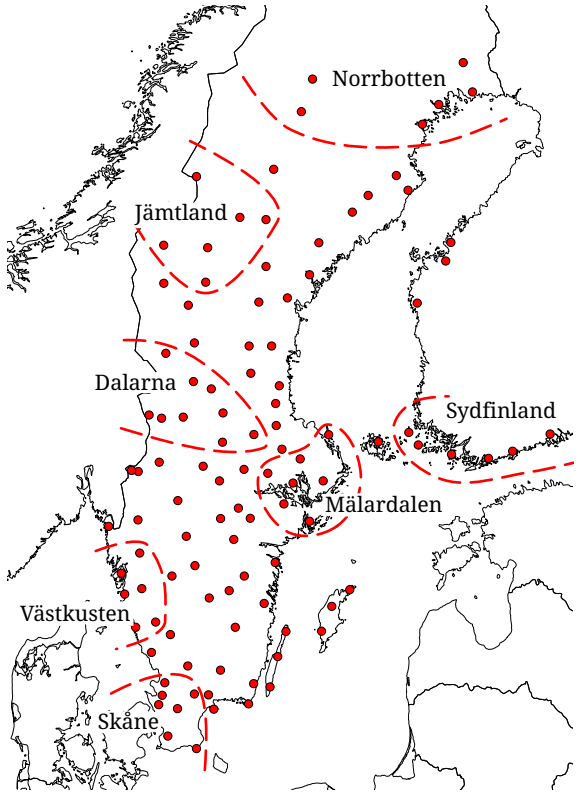

Figure 2.2: A map showing the division of SweDia recording locations into seven sub-regions, used as dialects. From [1]. Adapted with permission from the copyright holder.

Table 2.1: The SweDia2000 recording locations assigned to each dialect region.

Norrbotten	Jämtland	Sydfinland	Dalarna	Mälardalen	Västkusten	Skåne
Arjeplog	Åre	Borga	Södra Finnskoga	Skuttunge	Orust	Vaxtorp
Överkalix	Aspås	Kyrkslätt	Dalby	Villberga	Frändefors	Broby
Nederkalix	Frostviken	Snappertuna	Malung	Haraker	Kärna	Össjö
Piteå	Berg	Dragsfjärd	Leksand	Kårsta	Östad	Bara
Sorsele	Strömsund	Houtskär	Grangärde	Länna	Öxabäck	Löderup
Nederluleå	Ragunda	Brändö	Husby	Sorunda	Frillesås	Norra Rörum
			Sarna	Graso		

Table 2.2: Percentage of audio files which have a corresponding annotation file per dialect region.

Dialect	Spontaneous (%)	Wordlist (%)
Norrbotten	21.13	100.00
Jämtland	34.92	100.00
Sydfinland	29.17	97.22
Dalarna	6.58	98.86
Mälardalen	19.72	100.00
Västkusten	10.96	100.00
Skåne	37.50	100.00
Overall	22.49	99.45

2.3.1 Wordwise

Our first experiment category ignores the spontaneous data completely and focuses on utterances of single words from the *wordlist* category of interviews. With dialects defined, the frequency of words in the data can be studied. Out of all the different words uttered in *wordlist* interviews, 43 words occur frequently across all dialects, shown in Table A.2.

Four of the most frequently occurring: *dörr*, *flytta*, *kaka* and *lär*, are selected as calibration words. This is necessary since classifiers are set up and tuned based on data, but calibrating each classifier for each of the 43 words would be too time consuming. By limiting calibration to four words we hope to find hyperparameters which are suitable for capturing the complexity in single-word dialect classifications. It is assumed that a classifier setup which works well for the four calibration words will generalize without great loss of performance to the other 39 words.

It is then possible to evaluate the performance of dialect classifiers which operate on utterances of one specific word, and also find which words among the 43 are best suited for dialect classification. Lastly it is tested if combining multiple single-word classifiers into an ensemble classifier which takes a tuple of word utterances from a single speaker as input can be used to further improve performance.

2.3.2 Generalization to Spontaneous Speech

Compared to word-specific dialect classifiers, classifying spontaneous speech is arguably more practical, mostly due to the fact that a spontaneous classifier does not required predefined to perform classification.

The limited annotation coverage in the *spontaneous* category prohibits both training and evaluating classifiers on spontaneous speech. Instead we test if a classifier trained on the complete set of *wordlist* single word utterances (all 43 words), performs well when evaluated on spontaneous data. This tests if utterances of single words contain the same, or similar, dialectal markers as spontaneous speech.

Chapter 3

Signature Theory

In this chapter we introduce the concept of expressing information about a path in $\mathbb{R}^d$ by its signature. Formal power series and iterated integrals are introduced as these are necessary when defining the signature. We also present explicit expressions for transforming the signature into a log signature, and derive methods for practical signature and log signature calculations.

3.1 Preliminaries

This section provides a short overview of the mathematics used later in this chapter.

3.1.1 Tensor Product Spaces

The tensor product of two inner product spaces U and V is the space $U \otimes V$, here called W. If there is a bilinear map $\otimes$ so that

$$U \times V \to W, \quad (u, v) \mapsto u \otimes v$$

and W has an inner product $\langle \cdot, \cdot \rangle_W$ so that for $u^1, u^2 \in U$ and $v^1, v^2 \in V$

$$\langle u^1 \otimes v^1, u^2 \otimes v^2 \rangle_W = \langle u^1, u^2 \rangle_U \langle v^1, v^2 \rangle_V \tag{3.1}$$

holds, we will call W the tensor product space corresponding to a particular $\otimes$, which in this thesis will always be the outer product.

If the spaces U and V are spanned by basis vectors $\mathbf{e}_1^u, \ldots, \mathbf{e}_m^u$ and $\mathbf{e}_1^v, \ldots, \mathbf{e}_n^v$, then W is the space spanned by the basis of paired vectors $\mathbf{e}_i^u \otimes \mathbf{e}_j^v$ for $(i, j) \in \{1, \ldots, m\} \times \{1, \ldots, n\}$. Hence we have for the dimension of W that

$$\dim(W) = \dim(U)\dim(V).$$

Another consequence of this is that any element $w \in W$ can be be written as

$$w = \sum_{i,j=1}^{m,n} w_{i,j}(\mathbf{e}_i^u \otimes \mathbf{e}_j^v).$$

It follows that for elements $u \in U$ and $v \in V$ on the form

$$u = \sum_{i=1}^{m} u_i \mathbf{e}_i^u \quad \text{and} \quad v = \sum_{j=1}^{n} v_j \mathbf{e}_j^v$$

we can construct the object $u \otimes v \in W$ as

$$u \otimes v = \sum_{i,j=1}^{m,n} u_i v_j (\mathbf{e}_i^u \otimes \mathbf{e}_j^v).$$

For an n-fold tensor product of an element $v \in V$, we use the notation

$$v^{\otimes n} = \underbrace{v \otimes v \otimes \cdots \otimes v}_{n}.$$

The object $v^{\otimes n}$ is an element of $V^{\otimes n}$ which is defined analogously.

We can also join spaces U and V with direct summation, constructing a new space $W = U \oplus V$. The direct summation is similar to the Cartesian product, in that the resulting space W consists of all possible ordered pairs of elements from U and V. The direct summation however, also defines addition on W. With $u_1, u_2 \in U$ and $v_1, v_2 \in V$ the addition on W is defined as

$$(u_1, v_1) + (u_2, v_2) = (u_1 + u_2, v_1 + v_2).$$

The basis of W will be the combined set of basis elements from U and V, with slight modification. Using the notation $(\mathbf{e}_i, 0)$ or $(0, \mathbf{e}_j)$ to indicate basis vectors paired with a zero vector (from V or U respectively) together with standard notation for basis vectors in U and V, the set of vectors $(\mathbf{e}_1^u, 0), \ldots, (\mathbf{e}_m^u, 0), (0, \mathbf{e}_1^v), \ldots, (0, \mathbf{e}_n^v)$ make up a basis for W. From this we can see that $\dim(W) = \dim(U) + \dim(V)$.

3.1.2 Formal Power Series

As we shall see later on, signatures are elements in an infinite direct sum of tensor product spaces. A convenient way to express such an object is in the form of a *formal power series*. Formal power series are sums containing so called formal indeterminates, denoted by $\mathbf{e}_i$. It is no coincidence that the notation for formal indeterminates is the same as for basis vectors, since both lend structure to a mathematical objects. The main difference however is that a formal indeterminate does not represent any numerical value, but merely acts as a placeholder corresponding to its index i. Since formal indeterminates never take on any values, we do not have to worry about the convergence of a formal power series and are therefore allowed to have infinitely many non-vanishing elements in the series. For example we can use a formal power series to conveniently express a vector with an infinite number of elements as

$$\sum_{i=1}^{\infty} \lambda_i \mathbf{e}_i,$$

where λ_i are real valued coefficients.

We define our formal indeterminates to be non-commutative, meaning that $\mathbf{e}_i \mathbf{e}_j \neq \mathbf{e}_j \mathbf{e}_i$ if $i \neq j$. This allows us to use formal power series for writing an infinite matrix as

$$\sum_{i,j=1}^{\infty} \lambda_{i,j} \mathbf{e}_i \mathbf{e}_j.$$

With formal power series, we can represent tensors of arbitrary order. A series of one index of indeterminates constitutes a first order tensor, or vector. Two indices allows us to represent matrices, as above, and three indices would represent a third-order tensor, etc. Since the addition of two formal power series is practically direct summation, we can write a mathematical object which is a vector and a matrix as

$$\sum_{i=1}^{\infty} \lambda_i \mathbf{e}_i + \sum_{i,j=1}^{\infty} \lambda_{i,j} \mathbf{e}_i \mathbf{e}_j.$$

We will use addition for formal power series to represent both direct summation and standard addition. Should two formal indeterminates be identical during addition of two formal power series, standard component-wise addition is done. For differing indeterminates, addition refers to direct summation and extends the dimensionality of the object.

To represent constants in formal power series, we define the special empty indeterminate $\mathbf{e}_\varnothing$ which is implicitly placed after all terms that do not already have indeterminates. This indeterminate is special in that $\mathbf{e}_\varnothing \mathbf{e}_i = \mathbf{e}_i$, which means all operations we want to use also hold when scalars are present.

Another operation which we let formal power series inherit from the tensor product spaces is the inner product, but only between formal power series with finitely many terms, as else problems with convergence would arise. For example, the inner product between two vectors F and G with d elements, represented by the finite formal power series

$$F = \sum_{i=1}^{d} \lambda_i^{(F)} \mathbf{e}_i \quad \text{and} \quad G = \sum_{i=1}^{d} \lambda_i^{(G)} \mathbf{e}_i,$$

is

$$\langle F, G \rangle = \sum_{i=1}^{d} \lambda_i^{(F)} \lambda_i^{(G)}.$$

The inner product also extends naturally to object constructed from the direct summation and tensor products. For example, for F and G such as

$$F = \sum_{i=1}^{d} \lambda_i^{(F)} \mathbf{e}_i + \sum_{i,j=1}^{d} \lambda_{i,j}^{(F)} \mathbf{e}_i \mathbf{e}_j,$$
$$G = \sum_{i=1}^{d} \lambda_i^{(G)} \mathbf{e}_i + \sum_{i,j=1}^{d} \lambda_{i,j}^{(G)} \mathbf{e}_i \mathbf{e}_j$$

we have:

$$\langle F, G \rangle = \sum_{i=1}^{d} \lambda_i^{(F)} \lambda_i^{(G)} + \sum_{i,j=1}^{d} \lambda_{i,j}^{(F)} \lambda_{i,j}^{(G)}.$$

The inner product can also be used even when there isn't a perfect symmetry between the formal indeterminates in both arguments. We can practically view every formal power series as containing every possible indeterminate combination, but with

coefficient 0 for the ones we don't see. An example of this would be two vectors with different lengths, written as

$$F = \sum_{i=1}^{d_F} \lambda_i^{(F)} \mathbf{e}_i \quad \text{and} \quad G = \sum_{i=1}^{d_G} \lambda_i^{(G)} \mathbf{e}_i$$

where $d_F < d_G$. The inner product between these vectors is then

$$\langle F, G \rangle = \sum_{i=1}^{d_F} \lambda_i^{(F)} \lambda_i^{(G)}.$$

Lastly, we want to point that we can construct arbitrary tensors, i.e elements of the tensor product spaces, with formal power series. For example the two formal power series

$$F = \sum_{i_1=1}^{d} \dots \sum_{i_k=1}^{d} \lambda_{i_1,\dots,i_k}^{(F)} \mathbf{e}_1 \dots \mathbf{e}_k \quad \text{and} \quad G = \sum_{i_1=1}^{d} \dots \sum_{i_k=1}^{d} \lambda_{i_1,\dots,i_k}^{(G)} \mathbf{e}_1 \dots \mathbf{e}_k,$$

are kth order tensors with d dimensions. The inner product between these two formal power series, defined in (3.1), is the sum

$$\langle F, G \rangle = \sum_{i_1=1}^{d} \dots \sum_{i_k=1}^{d} \lambda_{i_1,\dots,i_k}^{(F)} \lambda_{i_1,\dots,i_k}^{(G)}.$$

This shows us that formal power series are related to inner product spaces. Therefore we want to introduce the tensor product operation to the formal power series notation. First of all we define the tensor product between two formal indeterminates $\mathbf{e}_i$ and $\mathbf{e}_j$ to be $\mathbf{e}_i \otimes \mathbf{e}_j = \mathbf{e}_i \mathbf{e}_j$, satisfying $(a\mathbf{e}_i) \otimes (b\mathbf{e}_j) = ab\mathbf{e}_i\mathbf{e}_j$ for any $a, b \in \mathbb{R}$ and having the distributive property $(\mathbf{e}_i + \mathbf{e}_j) \otimes (\mathbf{e}_k + \mathbf{e}_\ell) = \mathbf{e}_i\mathbf{e}_k + \mathbf{e}_i\mathbf{e}_\ell + \mathbf{e}_j\mathbf{e}_k + \mathbf{e}_j\mathbf{e}_\ell$. This distributive property means that a tensor product is also defined between formal power series, since these are simply sums of indeterminate terms.

For finite formal power series, this definition exactly mimics, by design, the tensor product related to inner product spaces presented in Section 3.1.1. This is the reason for having non-commutative formal indeterminates with the same behaviour as the outer product that we use as bilinear map. The tensor product for two finite formal power series F and G is then by definition

$$F \otimes G = \sum_{i=1}^{d} \lambda_i^{(F)} \mathbf{e}_i \otimes \sum_{j=1}^{d} \lambda_j^{(G)} \mathbf{e}_j = \sum_{i,j=1}^{d} \lambda_i^{(F)} \lambda_j^{(G)} \mathbf{e}_i \mathbf{e}_j.$$

For infinite formal power series the behaviour of the tensor product is very similar to the finite case. For example the tensor product between the two infinite formal power series

$$F = \sum_{i=1}^{\infty} \lambda_i^{(F)} \mathbf{e}_i \quad \text{and} \quad G = \sum_{j=1}^{\infty} \lambda_j^{(G)} \mathbf{e}_j$$

is by definition

$$F \otimes G = \sum_{i=1}^{\infty} \lambda_i^{(F)} \mathbf{e}_i \otimes \sum_{j=1}^{\infty} \lambda_j^{(G)} \mathbf{e}_j = \sum_{i,j=1}^{\infty} \lambda_i^{(F)} \lambda_j^{(G)} \mathbf{e}_i \mathbf{e}_j.$$

For infinite formal power series, however, the tensor product must differ from that of Section 3.1.1, since such series are, as noted above, not equipped with an inner product. This means that the tensor product between formal power series is a generalization, accommodating both finite and infinite series.

We will also preserve the notation of n-times repeated tensor products as $F^{\otimes n}$ for formal power series.

3.1.3 Paths and Line Integrals

A path is a continuous function $X : [a, b] \to \mathbb{R}^d$ which intuitively can be understood as a point moving through the space $\mathbb{R}^d$ over the time interval $[a, b]$. A path is said to be smooth if the first derivative $\dot{X} : [a, b] \to \mathbb{R}$ exists and is continuous on $[a, b]$. The integral of an integrable function $f : \mathbb{R}^d \to \mathbb{R}$ over a smooth path $X : [a, b] \to \mathbb{R}^d$ is defined as

$$\int_a^b f(X_t) \, \mathrm{d}X_t = \int_a^b f(X_t) \dot{X}_t \, \mathrm{d}t.$$

If X is not smooth on the whole of $[a, b]$ but rather piecewise smooth on the partitions $a = t_0 < t_1 < \ldots < t_n = b$ the line integral over X is defined as the sum of path integrals over each smooth segment

$$\int_a^b f(X_t) \, \mathrm{d}X_t = \int_{t_0}^{t_1} f(X_t) \, \mathrm{d}X_t + \int_{t_1}^{t_2} f(X_t) \, \mathrm{d}X_t + \ldots + \int_{t_{n-1}}^{t_n} f(X_t) \, \mathrm{d}X_t.$$

As $X = (X^1, \ldots, X^d)$ each X^i, which describes the change of X in direction i, is itself a path, but in $\mathbb{R}$ instead of $\mathbb{R}^d$. The line integral of the constant function $f = 1$ over one of the coordinate paths X^i in an interval with an endpoint decided by $t \in [a, b]$ is

$$I(X)_{a,t}^{(i)} = \int_a^t 1 \, \mathrm{d}X_s^i = \int_a^t \dot{X}_s^i \, \mathrm{d}s = X_t^i - X_a^i$$

which, again, is a path in $\mathbb{R}$. Since it is possible to integrate functions mapping $\mathbb{R}$ into itself over paths in $\mathbb{R}$ we can integrate $I(X)_{a,t}^{(i)}$ along any of the paths describing the change of X. If X^j is one of these paths, the function

$$I(X)_{a,t}^{(i,j)} = \int_a^t I(X)_{a,s}^{(i)} \, \mathrm{d}X_s^j = \int_a^t \int_a^s \mathrm{d}X_r^i \, \mathrm{d}X_s^j = \int_a^t \int_a^s \dot{X}_r^i \, \mathrm{d}r \dot{X}_s^j \, \mathrm{d}s$$

is yet another path in $\mathbb{R}$. This wrapping of the single dimensional paths describing the change of X leads to the concept of *iterated integrals*. The real number $I(X)_{a,b}^{(i)}$

is the 1-fold iterated integral of X along index i. The 2-fold iterated integral would be $I(X)_{a,b}^{(i,j)}$ which is along the pair of indices i, j. The general formula for a k-fold iterated integral is made over a sequence of k indices $i_1, \ldots, i_k$ and is the real number

$$I(X)_{a,b}^{(i_1,\ldots,i_k)} = \int_{a<t_k<b} I(X)_{a,t_k}^{(i_1,\ldots,i_{k-1})} \, \mathrm{d}X_{t_k}^{i_k} = \int_{a<t_k<b} \cdots \int_{a<t_1<t_2} \mathrm{d}X_{t_1}^{i_1} \ldots \mathrm{d}X_{t_k}^{i_k}.$$

3.2 Signature Definition

The signature of a path is closely related to iterated integrals, in fact it is the infinite collection of all iterated integrals along the path. A collection of real numbers which contains information about the path [19]. By using formal power series we can express this infinite set of numbers in a structured manner.

First we generalize the series of indices used in iterated integrals by introducing a lexical notation. Given a path $X = (X^1, X^2, \ldots, X^d)$ we call each index i a *letter* belonging to the *alphabet* $A = \{1, 2, \ldots, d\}$. Finite sequences of indices are called *words* and, if of finite length, are written as $w = (i_1, i_2, \ldots, i_k)$ with $|w|$ denoting the number of letters in w so that $|w| = k$. For a given alphabet there is an infinite number of possible words if no limit on word length is imposed. This set of words is denoted by W and contains the special empty word $\varnothing$ which has no letters and $|\varnothing| = 0$. With this notation we write iterated integrals as $I(X)_{a,b}^{w}$ where each letter i_j in w represents an integral along X^{i_j}. The iterated integral with respect to the empty word is equal to one by definition.

The signature of a path X is the collection of all iterated integrals corresponding to words in the infinite set W. To write this mathematical object in a meaningful way we will use formal indeterminates as placeholders for letters.

Definition 1. *The signature of a path $X : [a, b] \to \mathbb{R}^d$ is the formal power series*

$$S(X)_{a,b} = \sum_{k=0}^{\infty} \sum_{i_1,\ldots,i_k \in A} I(X)_{a,b}^{(i_1,\ldots,i_k)} \mathbf{e}_{i_1} \mathbf{e}_{i_2} \ldots \mathbf{e}_{i_k}.$$

The outer summation is over the length k of words, also called the *level* of the signature. The zeroth level contains only a single element which is equal to 1 by definition. The first level holds all iterated integrals based on one-letter words, and so for a d-dimensional path, the first level resides in $\mathbb{R}^d$. The second level is a matrix in $\mathbb{R}^d \otimes \mathbb{R}^d$, and the third is a tensor in $(\mathbb{R}^d)^{\otimes 3}$. This increase in the dimension of the space which each signature level resides in continues indefinitely. The entire signature resides in a space given by direct summation of each level-space, hence a signature is an element in $T(\mathbb{R}^d) = \bigoplus_{k=0}^{\infty} (\mathbb{R}^d)^{\otimes k}$.

Expressed as this formal power series, the operations defined in Section 3.1.2 can be applied to or between signatures[1]. An important result in the theory of signatures is that the tensor product of two signatures (corresponding to paths of equal dimensionality) will also be in $T(\mathbb{R}^d)$ and is therefore a valid signature. This is due to, and the reason behind, the zeroth level being defined as 1.

[1]Since signatures are infinite formal power series, this refers to the tensor product without an inner product.

3.3 Log Signature

While the signature is a full representation of its path, not all of its elements are independent. The log signature is a condensed representation, and contains the same information expressed with significantly fewer and independent elements [15].

This section demonstrates a method for extracting a basis representation for the log signatures, and we use the short-hand notation $\lambda^w = I(X)^w_{a,b}$ to make calculations more readable. We begin by writing the signature as the formal power series

$$S(X)_{a,b} = 1 + \sum_{k=1}^{\infty} \sum_{i_1,\dots,i_k \in A} \lambda^{(i_1,\dots,i_k)} \mathbf{e}_{i_1} \mathbf{e}_{i_2} \dots \mathbf{e}_{i_k}$$

with the term corresponding to the empty word moved out of the sum. The logarithm of a formal power series $F = a_0 + \sum_{k=1}^{\infty} a_k \mathbf{e}_k$ is defined as

$$\log F = \log a_0 + \sum_{n=1}^{\infty} \frac{(-1)^n}{n} \left(1 - \frac{F}{a_0}\right)^{\otimes n}.$$

Combining this definition with the signature gives

$$\log S(X) = \sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{n} \left(\sum_{k=1}^{\infty} \sum_{i_1,\dots,i_k \in A} \lambda^{(i_1,\dots,i_k)} \mathbf{e}_{i_1} \mathbf{e}_{i_2} \dots \mathbf{e}_{i_k} \right)^{\otimes n} \tag{3.2}$$

which has terms corresponding to every word in W except the empty word. We define W^* to be the set of all non-empty words, and let $\mathbf{e}_w$ denote the basis element for each letter in w, i.e. $\mathbf{e}_w = \mathbf{e}_{i_1} \dots \mathbf{e}_{i_k}$ if $w = (i_1, \dots, i_k)$. Using this notation we want to find a number $\hat{\lambda}_w$ for every $w \in W^*$ so that (3.2) can be rewritten as

$$\log S(X) = \sum_{w \in W^*} \hat{\lambda}_w \mathbf{e}_w. \tag{3.3}$$

For an arbitrary word w of length $|w| > 0$, all contributions to $\hat{\lambda}^w$ comes from the first $|w|$ tensor products in (3.2). Any higher n will only give rise to words longer than w. Let the concatenation of two words $u = (u_1, \dots, u_k)$ and $v = (v_1, \dots, v_\ell)$ be written as $uv = (u_1, \dots, u_k, v_1, \dots, v_\ell)$. Then we can use $w_1 \dots w_n = w$ to indicate a summation over all possible ways of splitting w into n non-empty subwords. The sought coefficient corresponding to the word w is then

$$\hat{\lambda}_w = \sum_{n=1}^{|w|} \frac{(-1)^{n+1}}{n} \sum_{w_1 \dots w_n = w} \lambda^{w_1} \dots \lambda^{w_n}. \tag{3.4}$$

This representation of the log signature has only one element less compared to the signature, namely the one corresponding to the empty word. Hence the redundancies are still present. To get rid of these we will project this expression for the log signature onto a basis, thereby using the smallest possible number of elements for expressing the same information.

It has been shown [19] that there exists numbers $\gamma_{i_1,\ldots,i_k}$ so that the log signature can be written with elements wrapped in Lie brackets as

$$\log S(X) = \sum_{k\geq 1} \sum_{i_1,\ldots,i_k \in A} \gamma_{i_1,\ldots,i_k} [\mathbf{e}_{i_1}, [\mathbf{e}_{i_2}, [\ldots, [\mathbf{e}_{i_{k-1}}, \mathbf{e}_{i_k}]\ldots]]. \tag{3.5}$$

The Lie bracket $[\cdot,\cdot]$ is a commutator for formal indeterminates defined as $[\mathbf{e}_i, \mathbf{e}_j] = \mathbf{e}_i\mathbf{e}_j - \mathbf{e}_j\mathbf{e}_i$. The fact that the log signature can be written as (3.5) means that it is an element in the so-called free Lie algebra. For an introduction to the subject of Lie algebras we refer to [20]. We note that the bracket-nested indeterminates in (3.5) does not form a basis in this algebra. For example, $[\mathbf{e}_1, \mathbf{e}_2]$ and $[\mathbf{e}_2, \mathbf{e}_1]$, which only differ in sign when expanded, both appear in the sum.

But there are known bases for the free Lie algebra, and projecting the log signature onto a basis reduces the number of terms and redundancies such as the one above. Here we apply the Lyndon basis outlined in [21]. Each element in the Lyndon basis corresponds to a Lyndon word. The set of Lyndon words $\mathcal{L}$ make up a strict subset of all non-empty words W^*. For instance all single letter words are Lyndon words, but no words consisting of a single repeated letter (more than once) are Lyndon words.

The formal indeterminates which serve as basis elements in the Lyndon basis are similar to those in (3.5) in that they are also nested in Lie brackets. In [21] the operator $\sigma(\cdot)$ is defined, which maps a Lyndon word w to nested Lie bracket expression, specifying the Lyndon basis element corresponding to w. Let the Lyndon suffix of a word w be the longest word v so that $w = uv$ with $u \neq \varnothing$ and $v \in \mathcal{L}$.

Here we define the operator $\sigma(\cdot)$ recursively. If w is a single letter word $w = (i)$ then $\sigma(w) = i$. If w is more than one letter long then it is divided into $w = uv$ where v is the Lyndon suffix of w and $\sigma(w) = [\sigma(u), \sigma(v)]$. The Lyndon basis element corresponding to a Lyndon word w is then the bracket structure given by $\sigma(w)$ but with all letters replaced by their corresponding formal indeterminates, denoted by $\mathbf{e}_{\sigma(w)}$.

For example the Lyndon word $w = (1, 2, 2)$ has the Lyndon suffix $v = (2)$ since v is a Lyndon word but $(2, 2)$ is not. Therefore w is bracked-wrapped to

$$\sigma(w) = \big[\sigma((1,2)), \sigma((2))\big] = [[1,2],2]$$

and then

$$\mathbf{e}_{\sigma(w)} = [[\mathbf{e}_1, \mathbf{e}_2], \mathbf{e}_2] = [\mathbf{e}_1\mathbf{e}_2 - \mathbf{e}_2\mathbf{e}_1, \mathbf{e}_2] = \mathbf{e}_1\mathbf{e}_2\mathbf{e}_2 + \mathbf{e}_2\mathbf{e}_2\mathbf{e}_1 - 2\mathbf{e}_2\mathbf{e}_1\mathbf{e}_2. \tag{3.6}$$

In the general case for any Lyndon word w we will have the basis element $\mathbf{e}_w$. The exact linear combination which $\mathbf{e}_w$ corresponds to can be found via the same procedure as in the example above. The word w is bracket-wrapped and the corresponding expression using formal indeterminates is expanded into a linear combination, giving us

$$\mathbf{e}_{\sigma(w)} = \sum_{v \in W_{\sigma(w)}} c_v \mathbf{e}_v, \quad w \in \mathcal{L},$$

where $c_i \in \mathbb{Z}$ and $W_{\sigma(w)}$ is a subset of the words that have the same length as w, as demonstrated in (3.6).

With the form of Lyndon basis elements known we now want to project the log signature expressed as in (3.3) onto each Lyndon basis element $\mathbf{e}_{\sigma(w)}$ to find the corresponding coefficients $\tilde{\lambda}^w$. The projection is calculated using the inner product $\langle \cdot, \cdot \rangle$. Since $W_{\sigma(w)}$ only contains words of the same length as w, it is sufficient to project only level $k = |w|$ of the log signature onto the basis. We use the notation $\log S(X)\big|_k$ to denote the kth level of the log signature. This means both arguments in the inner product are finite. The coefficient $\tilde{\lambda}^w$ is then given by

$$\tilde{\lambda}^w = \frac{\left\langle \log S(X)\big|_{|w|},\, \mathbf{e}_{\sigma(w)} \right\rangle}{|\mathbf{e}_{\sigma(w)}|^2} = \frac{\left\langle \sum\limits_{\substack{u \in W^* \\ |u|=|w|}} \hat{\lambda}^u \mathbf{e}_u,\, \sum\limits_{v \in W_{\sigma(w)}} c_v \mathbf{e}_v \right\rangle}{\left\langle \sum\limits_{v \in W_{\sigma(w)}} c_v \mathbf{e}_v,\, \sum\limits_{v \in W_{\sigma(w)}} c_v \mathbf{e}_v \right\rangle}$$

In the numerators inner product, the right hand element is only non-zero for the elements in $W_{\sigma(w)}$, thus only those terms will survive the product. Therefore the numerator is simply the linear expression which appears when expanding the commutators but with $\hat{\lambda}^w$ instead of $\mathbf{e}_w$. The denominator is even simpler since the nested basis element scalar produced with itself is the sum of squared c_v's. In conclusion

$$\tilde{\lambda}^w = \frac{\sum\limits_{v \in W_{\sigma(w)}} c_v \hat{\lambda}^v}{\sum\limits_{v \in W_{\sigma(w)}} c_v^2}, \quad w \in \mathcal{L}$$

and the log signature can be written as

$$\log S(X) = \sum_{w \in \mathcal{L}} \tilde{\lambda}^w \mathbf{e}_{\sigma(w)}. \tag{3.7}$$

3.4 Practical Calculations

Given the definition of signatures in Section 3.2, we now present two different techniques for calculating the signature and log signature of a path X under some assumptions. First of all we make an assumption regarding the paths in this thesis:

Assumption 1. *All paths are piecewise linear, and each linear segment corresponds to 1 time unit.*

This assumption is reasonable due to the fact that in this project every path is derived by interpolation from discretely sampled measurements. A path constructed from $n+1$ discrete measurements is the interpolated path $X = (X_t^1, \ldots, X_t^d)$ which maps the interval $[0, n] \to \mathbb{R}^d$.

Under these assumptions, calculating signatures for linear paths is of interest, and such signatures are easily computed. Let $X_t = a + bt$ with $a, b \in \mathbb{R}^d$ be a linear path parameterized by $t \in [p, p+1]$. It can then be easily shown that the signature element corresponding to the word $w = (w_1, \ldots, w_{|w|})$ becomes

$$I(X)^w_{p,p+1} = \frac{1}{|w|} \prod_{j=1}^{|w|} \dot{X}^{w_j} = \frac{1}{|w|} \prod_{j=1}^{|w|} b^{w_j}, \tag{3.8}$$

where $\dot{X}^{w_j}$ indicates the derivative of $X_t^{w_j}$ with regards to the time parameter t.

3.4.1 Chen's Identity

In this section we present a theorem called Chen's Identity, and use it to derive an explicit form for the signature of a path and a method of computing it. The theorem fuses together signatures from consecutive segments of a path by relating path concatenation to the tensor products of signatures, and while we will make use of Chen's Identity, the proof is beyond the scope of this thesis.

Theorem 1 (Chen's Identity)**.** *Let $a < c$. For a path $X : [a, c] \to \mathbb{R}^d$, it holds for $a < b < c$ that*

$$S(X)_{a,c} = S(X)_{a,b} \otimes S(X)_{b,c}.$$

Consider a path such as in Theorem 1. We keep the notation from the previous section and write $\lambda_{a,b}^{(i_1,\dots,i_k)} = I(X)_{a,b}^{(i_1,\dots,i_k)}$ and $\lambda_{b,c}^{(i_1,\dots,i_k)} = I(X)_{b,c}^{(i_1,\dots,i_k)}$ so that the signature $S(X)_{a,b}$ can be written on the form

$$S(X)_{a,b} = \sum_k \sum_{i_1,\dots,i_k \in A} \lambda_{a,b}^{(i_1,\dots,i_k)} \mathbf{e}_1 \mathbf{e}_2 \dots \mathbf{e}_k,$$

and analogously for $S(X)_{b,c}$. We can calculate the signature of the concatenated path using Chen's Identity, by performing a tensor multiplication between the two signatures. The tensor product is distributive and so for two signatures we have

$$(\lambda_{a,b}^{(0)}\mathbf{e}_\varnothing + \lambda_{a,b}^{(i)}\mathbf{e}_i + \lambda_{a,b}^{(i,j)}\mathbf{e}_i\mathbf{e}_j + ...) \otimes (\lambda_{b,c}^{(0)}\mathbf{e}_\varnothing + \lambda_{b,c}^{(i)}\mathbf{e}_i + \lambda_{b,c}^{(i,j)}\mathbf{e}_i\mathbf{e}_j + ...) \tag{3.9}$$

where all subscripts takes values in the alphabet $\{1 \dots, d\}$.

To calculate the signature of the concatenated path, we gather resulting terms from (3.9) according to words, keeping in mind the behaviour of the tensor product in Section 3.1.2. For word length $k = 0$, the tensor product yields only the scalar multiplication of the zeroth level elements from the two signatures, i.e.

$$I(X)_{a,c}^{(0)} = \lambda_{a,b}^{(0)}\lambda_{b,c}^{(0)}.$$

While we recall that the zeroth level of a signature is the empty word which is set to 1 by definition, we keep these in the calculations for the sake of correctness of the tensor product.

For $k = 1$ we have all words $i \in \{1, \dots, d\}$, and the concatenation gives

$$I(X)_{a,c}^{(i)} = \lambda_{a,b}^{(0)}\lambda_{b,c}^{(i)} + \lambda_{a,b}^{(i)}\lambda_{b,c}^{(0)}.$$

Continuing with increasing word size, now for $k = 2$:

$$I(X)_{a,c}^{(i,j)} = \lambda_{a,b}^{(0)}\lambda_{b,c}^{(i,j)} + \lambda_{a,b}^{(i)}\lambda_{b,c}^{(j)} + \lambda_{a,b}^{(i,j)}\lambda_{b,c}^{(0)}.$$

To illustrate the procedure we do one more step and obtain

$$I(X)_{a,c}^{(i,j,k)} = \lambda_{a,b}^{(0)}\lambda_{b,c}^{(i,j,k)} + \lambda_{a,b}^{(i)}\lambda_{b,c}^{(j,k)} + \lambda_{a,b}^{(i,j)}\lambda_{b,c}^{(k)} + \lambda_{a,b}^{(i,j,k)}\lambda_{b,c}^{(0)}.$$

Table 3.1: Word splitting of $w = (i, j, k)$ as a result of $g_2(w)$. The operator gathers all words u_1 and u_2 so that $u_1 u_2 = w$.

$\mathbf{u_1}$	$\mathbf{u_2}$
$\varnothing$	(i, j, k)
(i)	(j, k)
(i, j)	(k)
(i, j, k)	$\varnothing$

To gather terms for an arbitrary word in the new signature, we define a word splitting operator g_n, which distributes the letters of a word into all possible splits of n factors that give rise to the same word (the zero letter is "empty" and since it is a scalar it does not affect the order or changes the word). For $n = 2$, this operator distributes the word (i, j, k) as in Table 3.1. Essentially, g_n is an operator that gathers terms that correspond to the same words after performing n tensor products.

This process of concatenation can be expanded, and using Theorem 1, we can express the signature of an entire path by dividing it into n segments as

$$S(X)_{0,n} = S(X)_{0,1} \otimes S(X)_{1,2} \otimes \ldots \otimes S(X)_{n-1,n} \tag{3.10}$$

In the light of (3.10), we consider the signature element $S(X)^w_{0,n}$ for an arbitrary word w. We may write this as a concatenation of the segments $[0, n-1]$ and $[n-1, n]$. Using the operator g_2, we then write $S(X)^w_{0,n}$ as

$$I(X)^w_{0,n} = \sum_{(u_1,u_2)\in g_2(w)} \lambda^{u_1}_{0,n-1} \lambda^{u_2}_{n-1,n}.$$

We proceed with a recursive strategy and consider $\lambda_{0,n-1}$ to be the result of a concatenation of $[0, n-2]$ and $[n-2, n-1]$, and write

$$I(X)^w_{0,n} = \sum_{(u_1,u_2)\in g_2(w)} \left(\sum_{(u'_1,u'_2)\in g_2(u_1)} \lambda^{u'_1}_{0,n-2} \lambda^{u'_2}_{n-2,n-1} \right) \lambda^{u_2}_{n-1,n}. \tag{3.11}$$

By repeating this process recursively we have a method with which we can calculate signatures for entire paths by building it up from smaller segments. If we expand (3.11) it becomes

$$I(X)^w_{0,n} = \sum_{(u_1,u_2,u_3)\in g_3(w)} \lambda^{u_1}_{0,n-2} \lambda^{u_2}_{n-2,n-1} \lambda^{u_3}_{n-1,n}.$$

Repeating this procedure for all n segments of the path, the signature element for the word $w = (i_1, ..., i_k)$ can be written explicitly as

$$I(X)^w_{0,n} = \sum_{(u_1,\ldots,u_n)\in g_n(w)} \prod_{p=0}^{n-1} \lambda^{u_p}_{p,p+1}.$$

Table 3.2: An example of how the operator $s(\cdot)$ works, here applied on the word $(1, 2, 3, 4)$, i.e. $(u, v) \in s(1, 2, 3, 4)$.

j	u	v
0	$\varnothing$	(1,2,3,4)
1	(1)	(2,3,4)
2	(1,2)	(3,4)
3	(1,2,3)	(4)

We remember that $\lambda^w_{p,p+1}$ is precisely the signature element for word w on the segment $[p, p+1]$. For paths that satisfies Assumptions 1, these signature elements are readily available for according to (3.8).

3.4.2 Recursive Iterated Integral Expressions

An alternative approach for finding the real numbers making up the signature of a path is to calculate each iterated integral. The idea behind this approach is to use the assumed piecewise linearity to reduce integrals into sums and take advantage of iterated integrals recursive properties.

Since it is assumed that each piecewise linear segment corresponds to a unit increment in time we introduce the notation $\dot{X}^i_m = X^i_{m+1} - X^i_m$ for the increment of X in dimension i over the time interval $[m, m+1]$.

Expanding the iterated integrals corresponding to the first four levels (not counting the zeroth) in the signature of X reveals a distinct pattern

$$
\begin{aligned}
I(X)^{(i)}_{0,n} &= X^i_n \\
I(X)^{(i,j)}_{0,n} &= \sum_{m=0}^{n-1} \left[I(X)^{(i)}_{0,m} \dot{X}^j_m + \frac{\dot{X}^i_m \dot{X}^j_m}{2} \right] \\
I(X)^{(i,j,k)}_{0,n} &= \sum_{m=0}^{n-1} \left[I(X)^{(i,j)}_{0,m} \dot{X}^k_m + I(X)^{(i)}_m \frac{\dot{X}^j_m \dot{X}^k_m}{2!} + \frac{\dot{X}^i_m \dot{X}^j_m \dot{X}^k_m}{3!} \right] \\
I(X)^{(i,j,k,l)}_{0,n} &= \sum_{m=0}^{n-1} \left[I(X)^{(i,j,k)}_{0,m} \dot{X}^l_m + I(X)^{(i,j)}_{0,m} \frac{\dot{X}^k_m \dot{X}^l_m}{2!} \right. \\
&\qquad \left. + I(X)^{(i)}_m \frac{\dot{X}^j_m \dot{X}^k_m \dot{X}^l_m}{3!} + \frac{\dot{X}^i_m \dot{X}^j_m \dot{X}^k_m \dot{X}^l_m}{4!} \right].
\end{aligned}
$$

To generalize this trend and write an expression for arbitrary iterated integrals we define the operator $s(.)$ for splitting words w into two parts. If $(u, v) \in s(w)$ then u are the first j letters of w and v are the letters after the j-th, with $j = 0, \ldots, |w| - 1$. When $j = 0$ the sub-word u is an empty word which corresponds to a signature term equal to 1 by definition. Table 3.2 shows an example of how this split operator works, note that s is similar but not equivalent to the operator g_2 defined in the section above.

Under the assumed interpolation of n discrete measurements the signature of X consists of iterated integrals over the interval $[0, n]$. However any proof of a general expression for iterated integrals has to deal with the real valued times inside each linear segment, therefore the following theorem is formulated to hold for arbitrary times t.

Theorem 2. *Let $X : [0, \infty) \to \mathbb{R}^d$ be a path satisfying Assumption 1. Furthermore let $w = (i_1, \ldots, i_k)$ be an arbitrary word with letters in the alphabet $A = \{1, \ldots, d\}$. Then for any $t \in [0, \infty)$ with $n = \lfloor t \rfloor$ the iterated integral $I(X)^w_{0,t}$ is given by*

$$I(X)^w_{0,t} = \sum_{m=0}^{n-1} \left(\sum_{(u,v)\in s(w)} I(X)^u_{0,m} \frac{1}{|v|!} \prod_{i\in v} \dot{X}^i_m \right) + \sum_{(u,v)\in s(w)} I(X)^u_{0,n} \frac{(t-n)^{|v|}}{|v|!} \prod_{i\in v} \dot{X}^i_n. \tag{3.12}$$

Proof. First we note that if $t = n$ in (3.12) the second sum vanishes and

$$I(X)^w_{0,n} = \sum_{m=0}^{n-1} \left(\sum_{(u,v)\in s(w)} I(X)^u_{0,m} \frac{1}{|v|!} \prod_{i\in v} \dot{X}^i_m \right)$$

hence (3.12) can be written in a slightly shorter but equivalent form

$$I(X)^w_{0,t} = I(X)^w_{0,n} + \sum_{(u,v)\in s(w)} I(X)^u_{0,n} \frac{(t-n)^{|v|}}{|v|!} \prod_{i\in v} \dot{X}^i_n. \tag{3.13}$$

Note that $I(X)^w_{0,n}$ does not explicitly depend on t.

To prove that these equations hold for any word w, induction over the word length is used. The base case is a word of length 1, i.e. some single index i. For a single letter word the split operator s will only produce one pair of sub-words where $u = \varnothing$ and $v = (i)$. Starting at the definition of an iterated integral and using the piecewise linearity of X^i_t we get that

$$I(X)^{(i)}_{0,t} = \int_0^t 1 \, \mathrm{d}X^i_t = \int_0^t \dot{X}^i_t \, \mathrm{d}t = \sum_{m=0}^{n-1} \int_m^{m+1} \dot{X}^i_t \, \mathrm{d}t + \int_n^t \dot{X}^i_t \, \mathrm{d}t.$$

Since $\dot{X}^i_t$ is constant on each interval between two whole numbers it can be moved outside the integrals and

$$I(X)^i_{0,t} = \sum_{m=0}^{n-1} \dot{X}^i_m \int_m^{m+1} \mathrm{d}t + \dot{X}^i_n \int_n^t \mathrm{d}t = \sum_{m=0}^{n-1} \dot{X}^i_m + X^i_n (t-n)$$

which shows that (3.12) holds when $w = (i)$.

For the induction step first assume that (3.13) (which is equivalent to (3.12)) holds for $I(X)^w_{0,t}$ where w is a word with k letters. Add an extra letter $j \in A$ to this word to get $\widetilde{w} = (w, j)$ which has $k + 1$ letters. We now want to show that (3.12)

is fulfilled for $\widetilde{w}$ by using the inductive assumption. The definition of the iterated integral corresponding to $\widetilde{w}$ can be expanded into

$$\begin{aligned} I(X)_{0,t}^{\widetilde{w}} &= \int_0^t I(X)_{0,t}^{w} \,\mathrm{d}X_t^j = \int_0^t I(X)_{0,t}^{w} \dot{X}_t^j \,\mathrm{d}t \\ &= \sum_{m=0}^{n-1} \dot{X}_m^j \int_m^{m+1} I(X)_{0,t}^{w} \,\mathrm{d}t + \dot{X}_n^j \int_n^t I(X)_{0,t}^{w} \,\mathrm{d}t = (\mathrm{I}) + (\mathrm{II}). \end{aligned}$$

The terms (I) and (II) will be expanded independently. Starting with (I) we replace $I(X)_{0,t}^{w}$ according to the induction assumption which leads to

$$\begin{aligned} &\sum_{m=0}^{n-1} \dot{X}_m^j \int_m^{m+1} I(X)_{0,t}^{w} \,\mathrm{d}t \\ &\quad = \sum_{m=0}^{n-1} \dot{X}_m^j \int_m^{m+1} \left(I(X)_{0,m}^{w} + \sum_{(u,v)\in s(w)} I(X)_{0,m}^{u} \frac{(t-m)^{|v|}}{|v|!} \prod_{i\in v} \dot{X}_m^i \right) \mathrm{d}t \\ &\quad = \sum_{m=0}^{n-1} \dot{X}_m^j \left(I(X)_{0,m}^{w} \int_m^{m+1} \mathrm{d}t + \sum_{(u,v)\in s(w)} I(X)_{0,m}^{u} \int_m^{m+1} \frac{(t-m)^{|v|}}{|v|!} \,\mathrm{d}t \prod_{i\in v} \dot{X}_m^i \right). \end{aligned}$$

The integrals in this expression are computed by a change of variable

$$\int_m^{m+1} \frac{(t-m)^{|v|}}{|v|!} \,\mathrm{d}t = \int_0^1 \frac{s^{|v|}}{|v|!} \,\mathrm{d}s = \frac{1}{(|v|+1)!}.$$

Thus

$$\begin{aligned} &\sum_{m=0}^{n-1} \dot{X}_m^j \int_m^{m+1} I(X)_{0,t}^{w} \,\mathrm{d}t \\ &\quad = \sum_{m=0}^{n-1} \left(I(X)_{0,m}^{w} \dot{X}_m^j + \sum_{(u,v)\in s(w)} I(X)_{0,m}^{u} \frac{1}{(|v|+1)!} \left(\prod_{i\in v} \dot{X}_m^i \right) \dot{X}_m^j \right) \qquad (3.14) \\ &\quad = \sum_{m=0}^{n-1} \left(\sum_{(u,v)\in s(\widetilde{w})} I(X)_{0,m}^{u} \frac{1}{|v|!} \prod_{i\in v} \dot{X}_m^i \right) \end{aligned}$$

where in the final step the term outside the sum over $s(w)$ and the factor $\dot{X}_p^j$ is absorbed when instead summing over $s(\widetilde{w})$.

Similarly, the term denoted by (II) can be manipulated into

$$\begin{aligned}
&\dot{X}_n^j \int_n^t I(X)_{0,t}^w \,\mathrm{d}t \\
&\quad = \dot{X}_n^j \int_n^t \left(I(X)_{0,n}^w + \sum_{(u,v)\in s(w)} I(X)_{0,n}^u \frac{(t-n)^{|v|}}{|v|!} \prod_{i\in v} \dot{X}_n^i \right) \mathrm{d}t \\
&\quad = \dot{X}_n^j I(X)_{0,n}^w \int_n^t \mathrm{d}t + \dot{X}_n^j \sum_{(u,v)\in s(w)} I(X)_{0,n}^u \int_n^t \frac{(t-n)^{|v|}}{|v|!} \,\mathrm{d}t \prod_{i\in v} \dot{X}_n^i
\end{aligned}$$

where the integrals are computed as before, yielding

$$\int_n^t \frac{(t-c)^{|v|}}{|v|!} \,\mathrm{d}t = \int_0^{t-n} \frac{s^{|v|}}{|v|!} \,\mathrm{d}s = \frac{(t-n)^{|v|+1}}{(|v|+1)!}.$$

Resolving the integrals and once again absorbing extra terms into a sum over $s(\widetilde{w})$ gives

$$\begin{aligned}
&\dot{X}_n^j \int_n^t I(X)_{0,t}^w \,\mathrm{d}t \\
&\quad = I(X)_{0,n}^w (t-n)\dot{X}_n^j + \sum_{(u,v)\in s(w)} I(X)_{0,n}^u \frac{(t-n)^{|v|+1}}{(|v|+1)!} \left(\prod_{i\in v} \dot{X}_n^i \right) \dot{X}_n^j \\
&\quad = \sum_{(u,v)\in s(\widetilde{w})} I(X)_{0,n}^u \frac{(t-n)^{|v|}}{|v|!} \prod_{i\in v} \dot{X}_n^i.
\end{aligned}$$

This, when combined with (3.14), means that

$$I(X)_{0,t}^{\widetilde{w}} = \sum_{m=0}^{n-1} \left(\sum_{(u,v)\in s(\widetilde{w})} I(X)_{0,m}^u \frac{1}{|v|!} \prod_{i\in v} \dot{X}_m^i \right) + \sum_{(u,v)\in s(\widetilde{w})} I(X)_{0,n}^u \frac{(t-n)^{|v|}}{|v|!} \prod_{i\in v} \dot{X}_n^i,$$

recognised as the formula we want to prove, which means that the induction step holds and the proof is complete. □

The previous theorem shows that it is possible to calculate any iterated integral of X by starting with single letter words and iteratively calculating more and more complex iterated integrals. The sum over m indicates that is it not sufficient to only know the lower order iterated integrals on the entire interval $[0, n]$, one has to know all $n-1$ partial iterated integrals on the intervals $[0, m]$ for $m = 0, \ldots, n-1$. Since in practical applications, the signature up to a real valued time t is not of any interest, we formulate a corollary which only concerns iterated integrals over whole number intervals.

Corollary 3. *Let $X : [0, n+1] \to \mathbb{R}^d$ be a path satisfying Assumption 1. Then for an arbitrary word $w = (i_1, \ldots, i_k)$ with letters in $A = \{1, \ldots, d\}$ the iterated integral $I(X)^w_{0,n+1}$ is given by*

$$I(X)^w_{0,n+1} = I(X)^w_{0,n} + \sum_{(u,v)\in s(w)} I(X)^u_{0,n} \frac{1}{|v|!} \prod_{i\in v} \dot{X}^i_n.$$

Proof. Follows directly from (3.13) since $t = n$. □

With this corollary it becomes evident that a program which calculates signature elements can do so by looping over a finite set of words and over the segments of the piecewise linear path.

3.4.3 Log Signatures

The log signature of a path can be calculated from a signature first by (3.4) and then (3.7). The only details missing in this computation is how to retrieve the set of Lyndon words $\mathcal{L}$. While the set of Lyndon words is infinite, practical applications must put a limit to the number of levels in a log signature. Such a truncated set of Lyndon words up to a certain word length k can be calculated with Duval's algorithm [22, 23]. The exact procedure is described in Algorithm 1.

Notation : Let w_{-1} denote the last letter of the word w
Input : $n \in \mathbb{Z}^+$ – maximum Lyndon word length
$d \in \mathbb{Z}^+$ – alphabet size
Output : L_n – A set of integer tuples corresponding to all Lyndon words of up to length n based on the alphabet $\{1, \ldots, d\}$

```
L_n = ∅
w ← (0)
while |w| > 0 do
    w_{-1} ← (w_{-1} + 1)  mod d          // Increment last digit
    L_n ← L_n ∪ {w}                       // Now w is a Lyndon word

    w ← www...                            // Repeat w
    w ← (w_1, ..., w_n)                   // Trim to length n

    while w_{-1} = d and |w| > 0 do
        w ← (w_1, w_2, ..., w_{|w|-1})    // Remove trailing d's
    end
end
```

Algorithm 1: A procedure for generating a finite list of Lyndon words on the alphabet $\{1, 2, \ldots, d\}$, based on Duval's algorithm [22].

Chapter 4

Feature Extraction

In many applications of signal processing and data analysis, raw data is often difficult to make use of. For example sound data is sampled thousands of times per second. This means that a raw data sample is extremely high-dimensional. Furthermore, noise can distort the signal and translation issues occur since samples are not necessarily aligned in time. Data preprocessing is necessary in order to avoid these issues and extract usable information. The goal of preprocessing is to extract a set of so called features from a raw sample, expressing properties of the sample.

When dealing with sound as raw data, there are several options for feature extraction. In speech recognition [6] as well as language [4,24] and dialect classification [25], the most prominent technique involves Mel-frequency cepstral coefficients (MFCCs) which are based on the idea of mimicking human hearing.

In this thesis we will work with MFCCs and a closely related feature representation called Mel-frequency spectral coefficients (MFSCs). Furthermore, we also investigate the possibility of using signatures, both as a stand-alone feature and as a data enhancing compliment to other features.

4.1 Short-Time Fourier Transform

Given an audio signal, we are interested in the frequencies contained in the sample and how these change over time. The short-time Fourier transform (STFT) computes the frequency spectrum of a small time interval window, moving through the signal. This produces a spectrogram that provides us with an overview of how the frequencies evolve throughout the speech sample.

The window size is set to 25 ms, with the assumption that human speech is stationary on a short time interval [26]. Before calculating the Fourier transform of a window frame, however, the frame should be smoothed using some window function to avoid artifacts arising from the edges of the window. A common window function is the Hamming window, which is also the function of choice in this thesis. It is the weighted cosine with coefficients given by

$$w_n = 0.54 - 0.46 \cos\left(\frac{2\pi n}{L-1}\right), \quad 0 \leq n < L$$

with L being the window size. This function ensures that the signal vanishes towards the edges of the window. To make up for the fact that this causes loss of information near the edges, the window frames are made to overlap by 15 ms. After application

of the window function, we calculate the frequency spectrum $F(x)$ by discrete Fourier transform. Letting the signal in a window be $x = (x_0, \ldots, x_{L-1})$:

$$F(x)_k = \sum_{n=0}^{L-1} w_n x_n e^{-i2\pi kn/L}, \quad k = 0, \ldots, 2K-2$$

where w_n are the components of the Hamming window and $2K-2$ is the number of frequency bins used in the transform. Here we choose $2K-2 = 512$. All frequencies above the Nyquist frequency are discarded, which for a 16kHz sample rate is 8 kHz. This leaves us with $K = 257$ actual frequency bins. The transformation is done using fast Fourier transform, and the result of the STFT is a spectrogram where each column contains the frequency spectrum of its respective time frame.

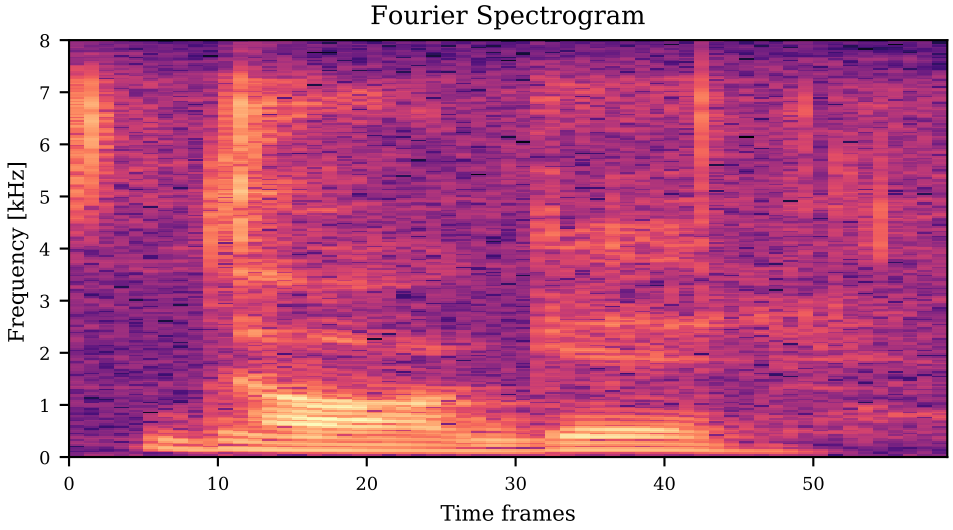

Figure 4.1: Fourier spectrogram. The figure shows logarithm of the pixel values for illustrativ purposes.

4.2 Mel-Frequency Spectral Coefficients

This section presents Mel-Frequency Spectral Coefficients, or MFSCs. The first step is to calculate a frequency spectrogram by transforming the raw signal through STFT. In the second step, we use a Mel scale filter bank. The Mel-scale is a non-linear frequency scale that mimics human hearing [27]. The human ear has finer frequency resolution at lower frequencies and the Mel scale aims to be linear in terms of human perception, rather than in actual frequency. The Mel-frequency relates to the standard frequencies according to the relation

$$m = 2595 \log_{10}\left(1 + \frac{f}{700}\right). \tag{4.1}$$

The Mel-scale filter bank consists of a set of overlapping triangular filters, with centers equidistantly spaced in the Mel-scale of (4.1). From a discrete and uniform

frequency range ($0 = f_0 < \ldots < f_K = 8000$), we compute the corresponding discrete Mel-frequency range using (4.1) and then place the filters uniformly on this range as seen in Figure 4.2, where 15 filters were used. The Mel-scale ensures that the filters are more densely distributed at lower frequencies, and that they grow farther apart and wider at higher frequencies.

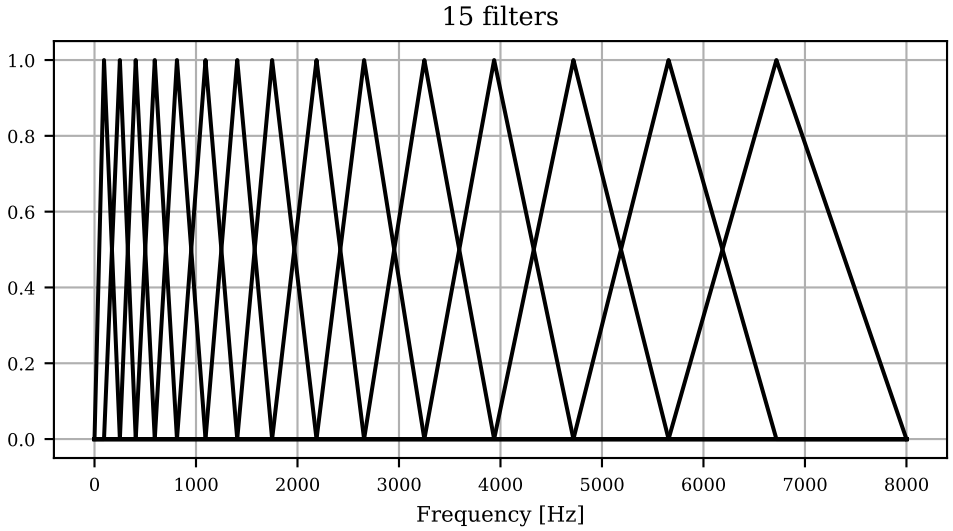

Figure 4.2: Mel-scale filter bank with 15 filters.

Given a speech sample $x \in \mathbb{R}^N$, we begin by computing the corresponding spectrogram, as done in Figure 4.1. Before applying a Mel filter bank however, each column in the STFT spectrogram is transformed into a power spectrum

$$P_k = \frac{1}{L}|F(x)_k|^2, \quad k = 0, \ldots, K$$

with K frequency bins. The result is a power spectrogram $S_{\mathrm{P}} \in \mathbb{R}^{K \times T}$ with K rows representing frequency bins and T columns, corresponding to the time frames.

The power spectrogram is then passed through a filter bank such as the one in Figure 4.2. Each filter is a function $b : \mathbb{R} \to [0, 1]$, and a filter bank B with F filters is a matrix the rows of which will contain the set of uniformly discretized triangular filters. For each row i we have $B_i = (b^{(i)}(f_0), \ldots, b^{(i)}(f_K)$. Thus, B is an object in $\mathbb{R}^{F \times K}$. Applying this filter bank to the power spectrogram produces a new spectrogram $S_{\mathrm{Mel}} \in \mathbb{R}^{F \times T}$:

$$S_{\mathrm{Mel}} = BS_{\mathrm{P}}.$$

After transforming S_{Mel} to decibels, this yields the MFSCs, which are actually Mel-frequency spectrograms. Figure 4.3 shows an example of a Mel spectrogram for a person saying the Swedish word *dagar*.

The number of filters in the filter bank determines the number of rows in the spectrogram. Figure 4.3 used $F = 40$ filters. Common values range from 20–40 filters.

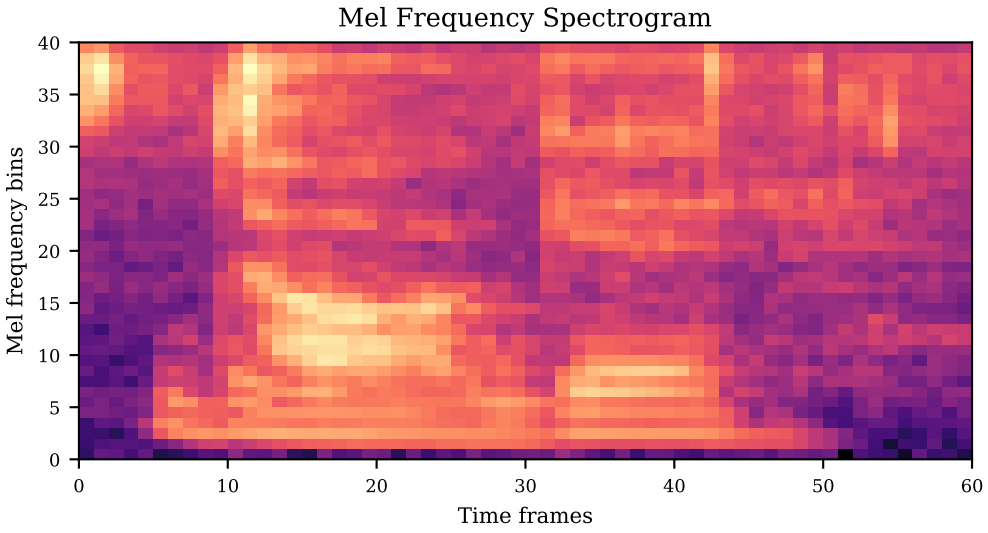

Figure 4.3: Mel spectrogram of the word *dagar* from a dalarna dialect

4.3 Mel-Frequency Cepstral Coefficients

In a spectrogram each column is a frequency spectrum, or Mel spectrum, for a specific time frame. This gives an overview of how the frequencies change throughout the sample. However, the frequency bins in the MFSCs are correlated, something that is worsened by the fact that the filters in the filter bank are overlapping. This poses a problem for classification algorithms that uses covariance matrices, such as Gaussian Mixture Models. Therefore, MFCCs proceeds one step further, in which the spectrogram is transformed using a discrete cosine transform (DCT), defined as

$$c_k = \sum_{n=0}^{N-1} X_n \cos\left(\frac{\pi k}{N}\left(n + \frac{1}{2}\right)\right), \quad k = 0, \ldots, N-1$$

along each column, which decorrelates the rows.

This also has the benefit of allowing us to discard the higher-order terms of the DCT, without losing valuable information. These terms represent rapid, high-frequency changes in the Mel spectrum that carries little information about the sound. The lower-order terms instead correspond to the overall spectral shape, or spectral envelope. Thus, the DCT step can be thought of as a frequency analysis of the frequencies, known as cepstral analysis. Keeping only lower order terms, provides a smoother spectrum and a more dense representation. The result is a set of MFCC feature vectors, as illustrated in Figure 4.4 where each column is a MFCC vector.

Not all classification techniques utilizes covariance matrices, however. Artificial neural networks can handle correlated samples, and convolutional networks are even based on the idea of local correlations within samples and so would benefit from the correlation. Therefore we will make use of both MFSCs and MFCC.

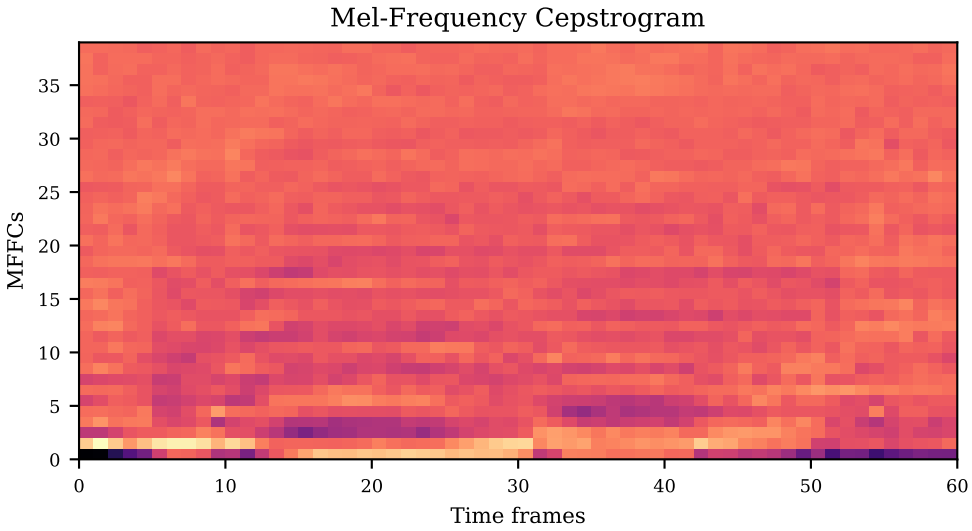

Figure 4.4: Spectrogram that has been decorrelated using a discrete cosine transform, hence a cepstrogram. The columns in the figure are MFCC feature vectors. Commonly only coefficients 0-12 are used, which means the other rows in the figure are discarded.

4.3.1 Shifted Delta Cepstra

While an MFCC sample does capture how a sound signal changes over time, we below present classifiers which see each column in such a matrix as its own individual sample. Such classifiers are limited to finding properties of the signal which fits inside single time frames. To get around this restricted temporal context we will use a method for expanding MFCC feature vectors with additional temporal information.

Temporal context is achieved by delta vectors [4], which express the change in features across neighbouring values. Let $(x_1, x_2, \ldots, x_T)$ be the column vectors in a Mel-frequency cepstrogram with N cepstral coefficients. By letting $x_t = x_1$ if $t < 1$ and $x_t = x_T$ if $t > T$ we indicate that the cepstrogram is extended at the left and right edges by repeating the first and last column respectively.

The classical definition for the delta features at time t with parameter n is $\Delta x_t = x_{t+n} - x_{t-n}$ which only utilizes the edges of the interval $[t-n, t+n]$ [28, 29]. While this is a well-tested method we have opted for the alternative approach where change around x_t is approximated using all neighbouring values [30]. The alternative definition of delta feature vectors is

$$\Delta x_t = \frac{\sum_{k=-n}^{n} k x_{t+k}}{\sum_{k=-n}^{n} k^2} \quad \text{for} \quad t = 1, \ldots, T. \tag{4.2}$$

This approach is perhaps most easily understood via linear regression. For each time $t = 1, \ldots, T$ we define a linear model

$$x_{t+k} = \alpha + \beta k + e_k \quad \text{for} \quad k = -n, \ldots, n$$

where α is intercept, β is slope and e_k are residuals. The aim is to approximate β and use that approximation as our delta feature vector for time t. Note that while all

terms in the linear model are vectors, the regression is done for each component of the vectors, independently. Using least squares to find our approximations produce the slope estimate

$$\frac{\sum_{k=-n}^{n}(k-\bar{k})(x_{t+k}-\bar{x})}{\sum_{k=-n}^{n}(k-\bar{k})^2}$$

where the bar notation indicates an average over $k = -n, \ldots, n$. Since $\bar{k} = 0$ this reduces to (4.2).

Now it is possible add temporal information to each MFCC feature vector x_t by concatenating it with its corresponding delta vector Δx_t. This would result in T feature vectors of length $2N$. This can be taken a step further by applying the above procedure to Δx_t instead of x_t, resulting in delta-delta vectors and combined feature vectors of length $3N$.

Here however, Shifted Delta Cepstra (SDC) [28, 29] which is a generalization of the delta and delta-delta methodology are used. In SDC multiple delta features from shifted time positions are concatenated, providing temporal context. SDC features are based on the delta vectors, meaning that we have both x_t and Δx_t for $t = 1, \ldots, T$ calculated with parameters N and n respectively. The parameter P denotes the difference in time index between each shifted delta vector and k denotes how many delta vectors will be picked out and concatenated for each time-frame. Finally the SDC vector at time t is the concatenation

$$\left[\Delta_t^T, \Delta_{t+P}^T, \Delta_{t+2P}^T, \ldots, \Delta_{t+(k-1)P}^T\right]^T.$$

Combining MFCC and SDC vectors by yet another concatenation results in the final MFCC + SDC feature vectors, each containing $(k+1)N$ elements.

4.4 Log Signatures as Acoustic Features

When incorporating signatures into dialect classification we have focused on log signatures, since it has non-redundant coefficients and is smaller in size, which will aid classification methods. To calculate log signatures it is necessary that the input data can be seen as a path. A sound signal is sampled over time, and is therefore inherently a path. Using amplitude as a single dimensional path however is not an option, since the log signature of a path in $\mathbb{R}$ consists of just one element (due to properties of the Lyndon basis).

Instead we have focused on the interpretation of MFCC samples as paths. MFCC samples are sequences of feature vectors with length N, corresponding to how many cepstral coefficients are kept. By interpreting each cepstral coefficient as a dimension, the sequence of feature vectors become a path in an N-dimensional space, from which log signature can be computed. In practice truncated log signatures are used, which contain elements up to some level ℓ.

Log signatures calculated from un-normalized MFCC feature vectors are numerically unstable, producing extremely large elements in higher levels. To remedy this, before the log signature is calculated for a path, all path elements are normalized to zero mean and standard deviation one.

To simplify classification on log signatures we step away from the formal power series notation

$$\log S(X) = \sum_{w \in \mathcal{L}} \tilde{\lambda}^w \mathbf{e}_{\sigma(w)}$$

and instead express the log signature as the vector

$$\log S(X) = [\tilde{\lambda}_{w_1}, \tilde{\lambda}_{w_2}, \ldots]$$

for a fixed ordering of Lyndon words $\{w_1, w_2, \ldots\} \subset \mathcal{L}$ of length up to ℓ. In both notations each $\tilde{\lambda}_w$ is a feature which will vary depending on the sample to log signature is calculated for.

4.5 Normalization

The last step when preparing data for classification is normalization, specifically featurewise normalization, which is applied to extracted features. The desired outcome of featurewise normalization is a set of feature samples where the average and standard deviation of every feature over the entire set is equal to zero and one respectively.

If the feature samples are vectors where each index corresponds to some feature, as is the case with our log signature features, normalization is very simple. If all feature vectors are gathered as columns in a matrix, the normalized feature vectors would be the original vectors minus the row means and divided by the column standard deviations.

For the spectro- and cepstro-scopic features however, there are multiple measurements for each feature per sample, since each row corresponds to a spectral or cepstral feature. Therefore the statistics are the average and standard deviation of each row across *all* spectrograms or cepstrograms. The actual normalization is similar to the vectorized case, each column in the sample matrices are shifted and skewed individually with the computed statistics.

Chapter 5

Classifiers

The classification problems formulated in this thesis are examples of what is called supervised classification which is a general procedure were classifiers are constructed (also called trained) from examples. The training is based on a set of samples $x_1, x_2, \ldots, x_N \in \mathcal{X}$ (also called the training set) and their corresponding labels $Y_1, Y_2, \ldots, Y_N \in \{1, \ldots, D\}$. The specifics of the input space $\mathcal{X}$ depends on which type of features are used but the labels always indicate which class each sample belongs to (or comes from).

A classifier is in this context just a function or method for performing the mapping

$$\mathcal{X} \to \{1, \ldots, D\}$$

as accurately as possible. This accuracy is both in terms of the examples available through the test data (samples and labels), and ideally, accurate in terms of data collected in the future.

We will see that classifiers are often defined in terms of parameters, of which there are two important categories. Hyperparameters are specified before training and state, for example, how large the classifier is or how the training should be controlled. The other kind are simply parameters, sometimes called trainable parameters for clarity, and as this name suggest these are derived during training such that the intended mapping to predictions is accurate. A classifier or classification method can produce very different results depending on its hyperparameters. It is therefore a fundamental part of supervised classification to find suitable hyperparameters.

5.1 Gaussian Mixture Models

Gaussian Mixture Models (GMMs) are one of the standard approaches to identification tasks on speech data [31]. Their suitability arises from their capability to adapt to arbitrarily shaped density functions. As the name suggests the mixture components in GMMs are normal distributions. The multivariate normal distribution in $\mathbb{R}^d$ with mean vector $\mu \in \mathbb{R}^d$ and covariance matrix $\Sigma \in \mathbb{R}^{d \times d}$ has the density function

$$f(x;\, \mu, \Sigma) = \frac{1}{(2\pi)^{k/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right), \quad x \in \mathbb{R}^d,$$

where $|\Sigma|$ denotes the determinant of Σ.

A GMM is built up from a weighted sum of M Gaussian density functions, often called components of the mixture model. Besides a mean vector and covariance matrix, each component is associated with a positive weight parameter w. The sum of all weight parameters is equal to one which means that the mixture model itself is a valid probability distribution on $\mathbb{R}^d$. For brevity we introduce $\theta = \{w_i, \mu_i, \Sigma_i\}_{i=1}^{M}$ as the complete set of parameters in a single GMM. The density function of a GMM with parameters θ is then given by

$$G(x \mid \theta) = \sum_{i=1}^{M} w_i \, f(x;\, \mu_i, \Sigma_i), \quad x \in \mathbb{R}^d.$$

The most demanding operation in GMMs is the inverse of the covariance matrix which is present in every mixture components density function. To decrease computational load we assume all covariance matrices to be non-zero only on their diagonals. While this is a pruning of which densities the GMM can mimic, the setback can be compensated by adding more mixtures [32, p. 75].

5.1.1 Training

Since a GMM is entirely described by its parameters, fitting a GMM to data is a matter of finding θ such that the density of the GMM resembles the unknown distribution of which the data is presumed to be sampled from. Here this is achieved by maximum likelihood estimation, i.e. finding parameters that maximize the likelihood of the GMM given a set of training vectors $X = \{x_1, \ldots, x_N\} \subset \mathbb{R}^d$. The likelihood function is

$$L(\theta \mid X) = \prod_{n=1}^{N} G(x_n \mid \theta).$$

To this end the Expectation–Maximization (EM) algorithm is utilized. The algorithm is an iterative procedure applicable to a wide array of statistical models. In each step model parameters are updated so that the model likelihood increases monotonically. In the case of GMMs there are fairly simple explicit expression for how to update θ. Exactly how many iterations are needed for convergence depends on the initial parameters. While initialization of parameters can be done randomly, the implementation used for GMMs in this thesis uses k-means clustering [33] (calculated with Elkan's algorithm [34]).

A two-dimensional example of what a GMM can look like post training is presented in Figure 5.1. The figure also indicates that the enforced diagonality of covariance matrices is not an issue when a suitably large number of components is chosen. The overall shape of the data cluster is captured by the log-density function, which is the intended outcome.

5.1.1.1 *k*-means clustering

The goal with using k-means clustering for parameter initialization is to assign every vector in X to a mixture component. Then w_i, μ_i and Σ_i can be initialized based on the vectors assigned to the mixture with index i.

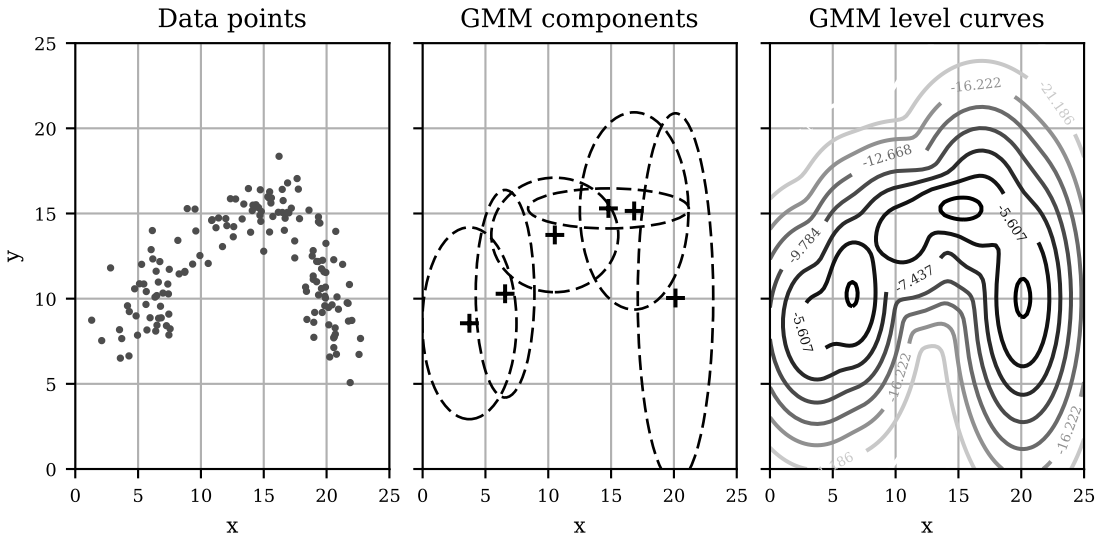

Figure 5.1: An example of how the distribution of cluster of data points in $\mathbb{R}^2$ can be approximated with a GMM constructed with the EM algorithm.

In k-means clustering X is divided into M non overlapping subsets, called clusters, $\mathcal{C} = \{C_1, C_2, \ldots, C_M\}$ so that the sum of variances from each cluster is minimized. This can be expressed as the optimization of finding $\mathcal{C}$ that minimizes the quantity

$$\sum_{i=1}^{M} \sum_{x \in C_i} \|x - m_i\|^2.$$

Here $\|.\|$ is the euclidean norm and m_i is the mean of points in cluster i.

After optimal clusters have been found the initial parameters for the mixture component with index i are calculated as

$$w_i^{(0)} = \frac{1}{N} \sum_{x \in C_i} 1, \quad \mu_i^{(0)} = m_i \quad \text{and} \quad (\Sigma_i)_{j,j}^{(0)} = \frac{1}{N} \sum_{x \in C_i} x^2 - m_i^2 \quad \text{for} \quad j = 1, 2, \ldots, d.$$

5.1.1.2 EM algorithm

With initial parameters $\theta^{(0)} = \{w_i^{(0)}, \mu_i^{(0)}, \Sigma_i^{(0)}\}_{i=1}^{M}$ in place the EM algorithm is used to refine the parameters [32]. Iteration k of the algorithm begins by calculating the intermediary probabilities

$$P_{i,n}^{(k)} = f(x_n; \mu_i^{(k)}, \Sigma_i^{(k)}), \quad \text{for} \quad i = 1, \ldots, M \quad \text{and} \quad n = 1, \ldots, N$$

which represent how probable it is that sample x_n belongs to mixture i under the model parameters $\theta^{(k)}$. Then the next set of parameters $\theta^{(k+1)}$ are computed according to the update rules

$$w_i^{(k+1)} = \frac{1}{N} \sum_{n=1}^{N} P_{i,n}^{(k)}, \quad \mu_i^{(k+1)} = \frac{\sum_{n=1}^{N} x_n P_{i,n}^{(k)}}{\sum_{n=1}^{N} P_{i,n}^{(k)}}$$

and

$$\left(\Sigma_i^{(k+1)}\right)_{jj} = \frac{\sum_{n=1}^{N} x_n^2 P_{i,n}^{(k)}}{\sum_{n=1}^{N} P_{i,n}^{(k)}} - (\mu_i^{(k+1)})^2 \quad \text{for} \quad j = 1, 2, \ldots, d.$$

Stopping conditions are implementation specific. In the package used here the algorithm stops when the gain from updating parameters falls below a specified tolerance.

5.1.2 Classification

When using GMMs for supervised classification each class is represented by its own GMM. For D distinct classes this will results in D different sets of parameters $\theta_1, \ldots, \theta_D$ which are found when training using data from only the corresponding class. Then each GMM mimics the distribution of that specific category and prediction is then a question of seeing which GMM agrees the most with the sample. Suppose a sample is a collection of N feature vectors on the form $X = \{x_1, \ldots, x_T\} \subset \mathbb{R}^d$. We define the average log likelihood for the GMM θ as

$$\ell(X \mid \theta) = \frac{1}{T} \sum_{t=1}^{T} \log G(x_t \mid \theta)$$

which is a more robust measurement of how well the sample fits to the GMM than the regular likelihood, especially when the number of vectors per sample varies. The predicted class of the sample X is then the GMM with the highest average log likelihood, meaning

$$\hat{Y} = \underset{i=1,\ldots,D}{\arg\max}\ \ell(X \mid \theta_i).$$

5.2 Artificial Neural Networks

A deep neural network consists of multiple layers of nodes, often referred to as neurons. Similarly to the neurons of the brain, these neurons can be activated and produce a response upon receiving an input signal, modeled by some activation function.

Layers of neurons are linked together so that a signal can propagate forward through the network. Each connecting link between neurons in two layers is associated with a weight and a bias. These weights and biases make up the parameters θ of the network. The purpose of an artificial neural network is to construct a model of neuron layers, linked through various connection designs and operations, and adjust its parameters so that the network approximates the true distribution $p(y|x)$. Thus, neural networks are layered models, and can be expressed as a composition[1] of functions, mapping input x to output $\hat{y}$ as

$$\hat{y} = (F^{(L)} \circ F^{(L-1)} \circ \ldots \circ F^{(1)})(x),$$

where the $F^{(1)}, \ldots, F^{(L)}$ are functions representing the neuron layers, which can be associated with different designs and operations in order to build the desired type of network.

[1]The composition of two functions $g(x), f(x)$ is written as $(g \circ f)(x) = g(f(x))$

5.2.1 Network Building Blocks

The layered structure of feedforward neural networks means that as long as each layer produces output in a space which is the input space of the next layer, different layers can be combined in different orderings. This section lists the layers and operations that make up the networks used in this thesis.

5.2.1.1 Fully Connected Layer

The perhaps most basic type of layer is a fully connected, or dense, layer, in which every neuron is linked to every single neuron in the previous layer. Hence, two layers ℓ and $\ell + 1$ with n and m neurons respectively, are connected through a weight matrix $W \in \mathbb{R}^{n \times m}$ and bias vector $b \in \mathbb{R}^m$. In layer ℓ, the neural network computes the neuron activations $h^{(\ell+1)} \in \mathbb{R}^m$ of the next layer through a function $H^{(\ell+1)} : \mathbb{R}^n \to \mathbb{R}^m$ such that

$$h^{(\ell+1)} = H^{(\ell+1)}(h^{(\ell)}) = G\left(W^{(\ell)}h^{(\ell)} + b^{(\ell)}\right). \tag{5.1}$$

G is a function that applies the so-called activation function $g : \mathbb{R} \to \mathbb{R}$ component-wise, so that $G(x) = [g(x_1), \ldots, g(x_d)]^T$.

5.2.1.2 Activation Functions

The activation function is a vital component of artificial neural networks. If g is chosen to be a non-linear function, this allows the network to learn non-linear properties of the data. Common alternatives for g include sigmoid functions and tanh, as well as rectified linear units (ReLU) which are the activation function of choice in this thesis. A ReLU is defined as

$$g(x) = \max(0, x). \tag{5.2}$$

This function is non-linear, and have readily available derivatives which are used when training networks. It is worth noting that (5.2) is technically not differentiable in $x = 0$, however in practice this is not an issue since x is unlikely to be exactly zero, and should it occur it is easily dealt with by defining the gradient to zero at $x = 0$.

5.2.1.3 Output Layer

As already stated, ReLUs are used for activations in this thesis. In the last layer of a network, however, a different activation function is applied. As mentioned above, the final layer corresponds to the class associations. In order to help us make sense of the output, we apply a softmax function, which allows us to interpret the output of the network as probabilities. The true distribution $y = (y_1, \ldots, y_D)$ will then be a *one-hot* representation, meaning that $y_d = 1$ for a sample that belongs to class d, and all other elements in y will be zero, since a sample can only belong to one class. If $\hat{o}$ is the output of the network without any activation function, the softmax function generates the final output $\hat{y}$, corresponding to the probabilities of a sample

x belonging to class d through

$$\hat{y}_d = \frac{e^{\hat{o}_d}}{\sum_j^D e^{\hat{o}_j}}, \tag{5.3}$$

so that $\hat{y_d} = \mathrm{P}(Y = d|x)$ where Y denotes the random variable representing class labels. The artificial neural network thus models the distribution $p(y|x, \theta)$, where the parameters θ are the weights and biases of the network.

5.2.1.4 Convolutional Layers

The layers in the network architecture in Figure 5.2 are called dense or fully connected layers, since each neuron is connected to all neurons in the previous layer. Hence, the number of parameters in deep fully connected network architectures grows extremely fast as the complexity increases. Convolutional layers are a design that exploits local correlations in the input by convolving small filters with the input. In each convolutional layer a set of trainable filters, each with its own weights and biases, are swept across the entire input looking for patterns. While the filters are randomly initialized, they will be trained to search for patterns that are useful for classification. Such a setup greatly reduces the number of parameters in the model since the filters have limited support and are small compared to the input, and the convolution allows the filters to detect patterns anywhere in input, regardless of position.

During the convolution, the input $x \in \mathbb{R}^d$ is extended with zeroes around the edges, in order to maintain the dimensions, so that $x_i = 0$ for $i < 1$ and $i > d$. This procedure also means that we reduce a bias towards the center of the input, since without this extension of zeros, elements near the edges of x would be included in fewer filters. In the one-dimensional case for a convolution between an input vector $x \in \mathbb{R}^d$ and a filter $f \in \mathbb{R}^M$, the convolution operation can be written as

$$(f * x)_n = \sum_{m=1}^{M} f_m x_{n-m+\mu}, \quad n = 1, \ldots, d$$

for filter size M and where μ is an offset parameter that takes care of the padding so $\mu = \frac{M+1}{2}$ for M odd and $\mu = \frac{M}{2}$ for M even. Due to the padding procedure where x is extended with zeros, we can maintain the dimension after the convolution so that $f * x \in \mathbb{R}^d$.

The convolution can be generalized to any dimension. For $x \in \mathbb{R}^{d_1 \times \ldots \times d_N}$, a convolution with a filter $f \in \mathbb{R}^{M_1 \times \ldots \times M_N}$ is defined as

$$(f * x)_{n_1,\ldots,n_N} = \sum_{m_1=1}^{M_1} \ldots \sum_{m_N=1}^{M_N} f_{m_1,\ldots,m_N} x_{n_1-m_1+\mu_1,\ldots,n_N-m_N+\mu_N}.$$

with $n_1, \ldots, n_N$ running from 1 to $d_1, \ldots, d_N$, respectively.

A convolutional layer can and usually does contain multiple filters, that can be taught to search for different patterns in the input. For a 2D-image input, a layer might for instance contain one filter that has learned to look for horizontal edges, another for vertical edges and a third could be searching for curved patterns. Each

filter in the layer thus give rise to individual outputs, called feature maps, and as these filters sweep across the image they will produce a high activation when looking on a subsection of the image that matches their specific patterns. After the filter convolution, biases are added and finally an activation function is applied to produce the output of the convolutional layer.

For a two-dimensional input image $x \in \mathbb{R}^{d\times d}$, a first convolutional layer with $N^{(1)}$ number of M-by-M filters will have a kernel $W \in \mathbb{R}^{M\times M\times N^{(1)}}$, which is a tensor that stores the filters of the layer. The layer will also have biases $b \in \mathbb{R}^{N^{(1)}\times d\times d}$. For filter n, this layer produces feature map $h_n \in \mathbb{R}^{d\times d}$, according to

$$h_n = G\left(W_n * x + b_n\right), \tag{5.4}$$

with $W_n \in \mathbb{R}^{M\times M}$ being the n:th slice of the kernel that represents filter n and b_n the bias of that filter. It is common to have a single bias term per filter, hence all components in $b_n \in \mathbb{R}^{d\times d}$ will be identical.

The result of the first convolutional layer will be $h \in \mathbb{R}^{d\times d\times N^{(1)}}$. Hence, for subsequent convolutional layers, the input will have another dimension determined by the number of filters in the previous layer (the input layer could also have additional dimensions, as is the case for RGB-images, for which the input is $x \in \mathbb{R}^{3\times d\times d}$). A general convolutional layer with $N^{(\ell+1)}$ filters can thus be described by a function $K_{N^{(\ell+1)}} : \mathbb{R}^{d\times d\times N^{(\ell)}} \to \mathbb{R}^{d\times d\times N^{(\ell+1)}}$ that applies (5.4) for all filters in the kernel $W^{(\ell+1)} \in \mathbb{R}^{N^{(\ell)}\times M\times M\times N^{(\ell+1)}}$:

$$h^{(\ell+1)} = K_{N^{(\ell+1)}}(h^{(\ell)}) = \begin{bmatrix} G(W_1 * h^{(\ell)} + b_1) \\ G(W_2 * h^{(\ell)} + b_2) \\ \vdots \\ G(W_{N^{(\ell+1)}} * h^{(\ell)} + b_{N^{(\ell+1)}}) \end{bmatrix}$$

5.2.1.5 Max-Pooling

Max pooling is a downsampling technique that is often applied after a convolutional layer. Feature maps are downsampled by only keeping the maximum value from regions of the input. This type of non-linear downsampling is called max-pooling. While in general max-pooling can be done using k-by-k sized patces, a common setup in image applications is non-overlapping two-by-two max-pooling, which reduces the image size by a factor of four by only keeping the max value of each 2-by-2 patch. This reduction in data size is one of the reasons for max-pooling, as it makes computations easier for subsequent layers. Another reason is that this provides a local translation invariance, since it will not matter where in the pooling patch the filter has been maximally activated. In turn, this can help the network handle local noise and variations. Two-by-two max-pooling can be written as the function $P : \mathbb{R}^{d\times d} \to \mathbb{R}^{d/2\times d/2}$, with $d \in \mathbb{N}$ being even:

$$h^{(\ell+1)}_{i,j} = P(h^{(\ell)})_{i,j} = \max\left\{h^{(\ell)}_{2i,2j}, h^{(\ell)}_{2i,2j+1}, h^{(\ell)}_{2i+1,2j}, h^{(\ell)}_{2i+1,2j+1}\right\}.$$

5.2.2 Network Architectures

Using the building blocks of the previous sections, we construct two well-known types of network architectures. The first is a more basic design and it uses only fully connected layers. The other type is the convolutional network design, which have proven very successful in wide-spread applications.

5.2.2.1 Multi-layer Perceptrons

By stacking fully connected layers such as in Section 5.2.1.1, we map some input feature vector x from the first layer, through intermediate hidden layers, to an output $\hat{y}$. Such a network architecture is referred to as a multi-layer perceptron (MLP). A MLP network of L layers can thus be written as a series of function compositions

$$\hat{y} = (H^{(L)} \circ H^{(L-1)} \circ \ldots \circ H^{(1)})(x).$$

Figure 5.2 illustrates a MLP using two hidden layers and two output classes. The network maps an input vector $x \in \mathbb{R}^4$, through two hidden layers $h^{(1)}, h^{(2)} \in \mathbb{R}^5$ to a final output vector $\hat{y} \in \mathbb{R}^2$, where each hidden layer performs the operation described by (5.1).

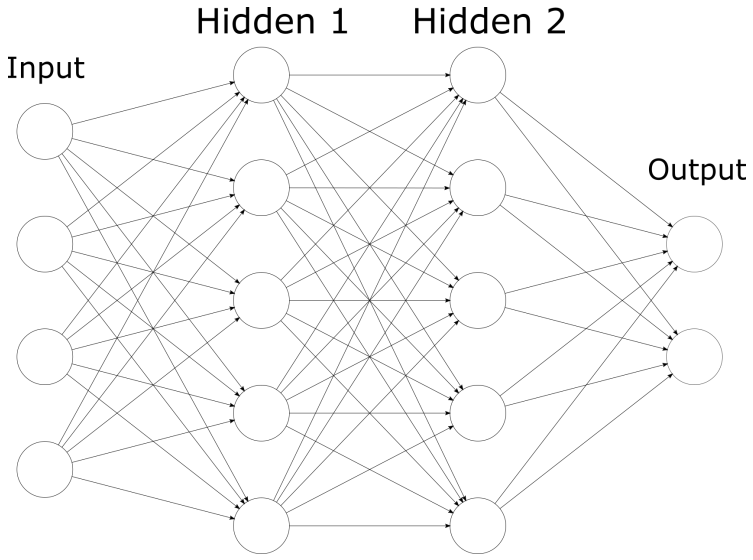

Figure 5.2: Artificial neural network with four-dimensional input and two hidden layers, mapping to two output nodes. The arrows between the neurons show the structure of the connections.

5.2.2.2 Convolutional Neural Networks

A convolutional neural network (CNN) are another type of network architecture which, logically, utilizes the convolutional layers described in Section 5.2.1.4. However, CNNs aren't composed solely of convolutional layers. As mentioned, the activations of the filters in a convolutional layer are called "feature maps". These feature maps tells us if and where in the input the filters found their patterns. In this sense,

convolutional layers act like feature extractors, and therefore convolutional layers are usually followed by one or more fully connected layers, which perform classification on the features and feature locations, as found by the convolutional layers. An example of a CNN architecture is shown in Figure 5.3. Using the building blocks of operations described in this chapter, the network in Figure 5.3 can be written as

$$\hat{y} = (H \circ H \circ P \circ K_{32} \circ P \circ K_{32})(x),$$

with a flattening of the tensor after the second max-pooling layer. This is necessary whenever a fully connected layer follows a convolutional layer, since the output of the latter is a set of 2D-feature maps while the fully connected layer disregards such structure and simply takes a flattened feature vector as input.

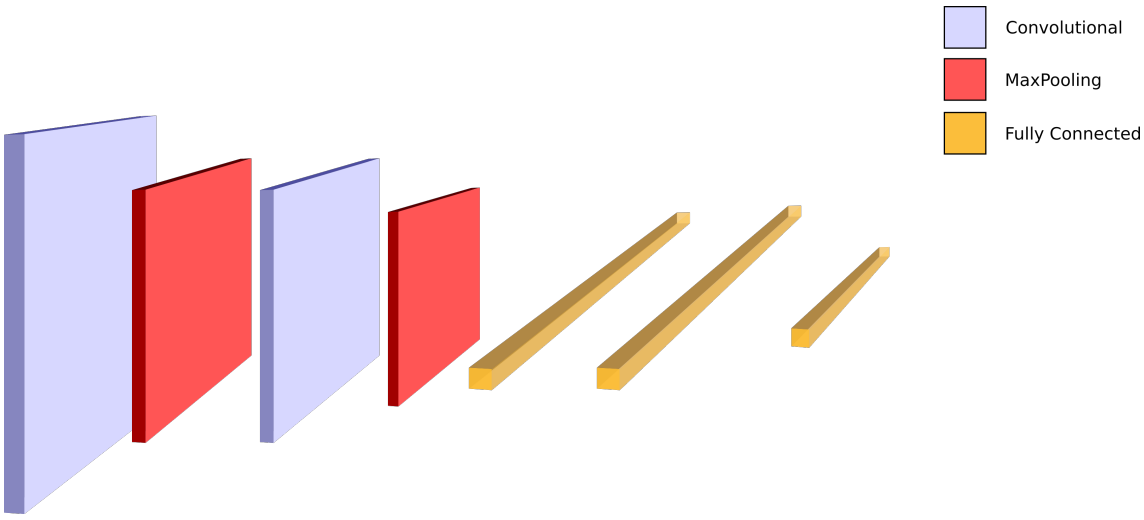

Figure 5.3: Example of a convolutional neural network architecture with two convolutional layers with max-pooling layers between them. The final two layers of the network are fully connected layers.

5.2.2.3 Modular Neural Networks

The building blocks of artificial neural networks allows for some freedom in terms of network architectures. A plethora of options exists; multi-layered perceptrons, convolutional networks and recurrent networks are perhaps the most common types. The choice of network type depends on application and the nature of the data. However, in some cases, multiple types of input data could be available that does not necessarily fit into a specific kind of network. A system looking to identify people could make use of a modular network where a convolutional network extracts visual features from images and another network processes audio such as voice samples. A modular neural network architecture could then fuse these modules together as in Figure 5.4, for instance by simply concatenating their output into a single vector and then proceed with the concatenated vector for classification.

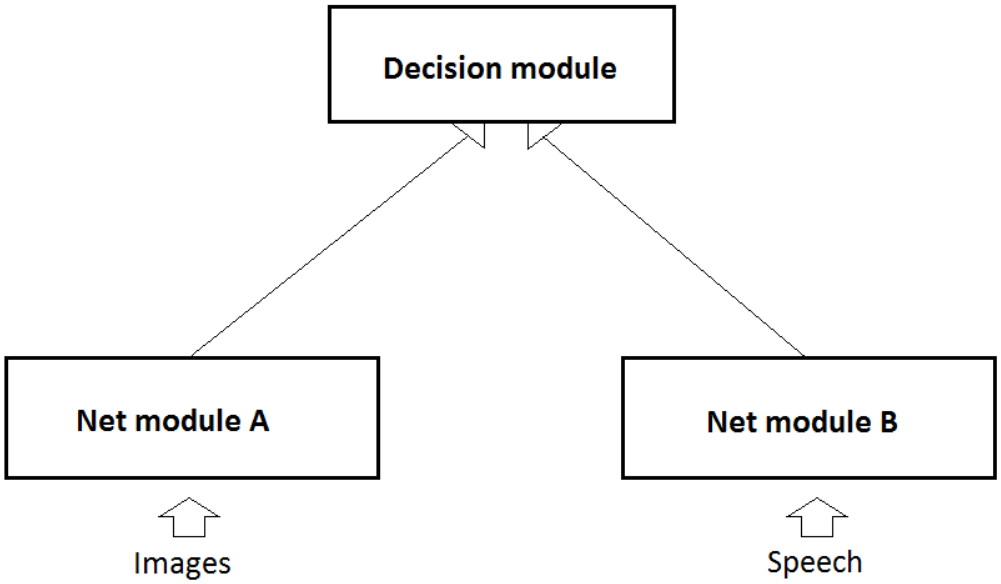

Figure 5.4: Modular network setup for a case when two different types of input are available. Module A and B process different inputs, and their outputs are concatenated and fed to a decision module that performs classification.

5.2.3 Training

Once a network has been designed, we want to modify the weights and biases of network in order to learn a good approximation of the true distribution $p(y|x)$. This process is known as training.

Letting $\mathbf{y}$ and $\hat{\mathbf{y}}$ be the true and predicted label distributions for all samples $x_1, x_2, \ldots, x_N$ in our the data set that we will use to train our model, we begin by defining our cost function as the cross entropy error function

$$C(\theta; \mathbf{y}, \hat{\mathbf{y}}) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{d} y_d^{(i)} \log(\hat{y}_d^{(i)}), \tag{5.5}$$

where i denotes a sum over all N samples in the so-called training data set. The cross entropy has several properties that makes it an attractive choice as cost function. Since $y_d^{(i)}$ and $\hat{y}_d^{(i)}$ are probabilities, C is always positive, and will give a high cost if $\hat{y}_d^{(i)}$ is small when $y_d^{(i)} = 1$. Furthermore, this cost function will give zero cost only if $\hat{\mathbf{y}}$ exactly equals $\mathbf{y}$. Even if the network were to predict the correct label for a sample, a cost would be present unless the prediction was done with 100% confidence, i.e. $\hat{y}_d^{(i)} = 1$. This means that the cross entropy cost function can distinguish between a network that predicts the correct labels with high margin and one that just barely made the right guess. The logarithm in (5.5) always defined, since the network will never hypothesize that a class has zero probability thanks to the softmax function (5.3) in the output layer.

We are now looking to adjust the weights, filters and biases that constitute θ in order to minimize the cost function for our training set. The gradients $\nabla_\theta C$ can be

be found through the so-called backpropagation algorithm, in which we find gradients by chaining partial derivatives of the cost function w.r.t all trainable parameters θ in the network, such as $\frac{\partial C}{\partial W^{(\ell)}}$ and $\frac{\partial C}{\partial b^{(\ell)}}$. The cost can then be minimized using some gradient descent method. Unfortunately, computing the full gradient is in practice very expensive. The cost function (5.5) contains all samples in our training data set, so finding the full gradient would require us to feed our entire data set through the network, which is infeasible in many applications due to the size of data sets. The solution is the stochastic gradient descent method, where the cost function is approximated by the cost $\widehat{C}$ of a small batch of randomly chosen samples with labels $\mathbf{y}_b, \hat{\mathbf{y}}_b$:

$$\widehat{C}(\theta; \mathbf{y}_b, \hat{\mathbf{y}}_b) = -\frac{1}{B} \sum_{i=1}^{B} \sum_{d} y_d^{(i)} \log(\hat{y}_d^{(i)}), \tag{5.6}$$

where B is the size of the batches, chosen so that $B << N$. This circumvents the need for repeatedly going through a large training set at every step, and also provides stochasticity that helps the optimization escape local optima. During training, batches are continuously fed to the network. A full sweep of data set, i.e. the entire data set has been passed to training, is referred to as an epoch. During training, it is common to perform several epochs in order to reach high performance.

For training, we use the Adaptive Moment Estimation (Adam) algorithm [35], which is a stochastic gradient descent technique. The name stems from the fact that it utilizes an adaptive learning rate based on past gradients from previous iterations. Given the gradient $g_t = \nabla_\theta \widehat{C}(\theta_{t-1}; \mathbf{y}, \hat{\mathbf{y}})$ at iteration t, the Adam algorithm calculates first and second moments m_t and v_t as

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t, \quad t \in \mathbb{N} \tag{5.7}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2, \quad t \in \mathbb{N} \tag{5.8}$$

where β_1 and β_2 are hyperparameters of the algorithm. Adam then updates the parameters θ with some learning rate α, according to the update rule

$$\theta_t = \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}, \quad t \in \mathbb{N} \tag{5.9}$$

The number ϵ is a small constant included for numerical stability. The moments $\hat{m}_t$ and $\hat{v}_t$ are bias-corrected versions of m_t and v_t, for details see [35].

The update (5.9) is a combination of the current gradient and previous updates. The moments counteract oscillations in the path of convergence by providing an inertia that drags the updates in the directions of gradients from earlier time steps, which leads to faster convergence.

5.2.3.1 Regularization

One of the major issues with artificial neural networks is their proneness to overfit to the training data, which gives poor generalization abilities. This is explained in more detail below but in short a network which performs *too* well on the training data might not be the best network overall. To combat this, regularization techniques are applied to networks, or components in networks.

One such technique is L^2-regularization, which can be applied to some or all parameters (weights and biases) in a network. The regularized parameter's L^2-penalty is added to the cost function to discourage the network from growing too large parameters. If some subset of the parameters in a network $\theta_r \subset \theta$ are L^2-regularized this means that the cost function (5.6) is changed to

$$\widehat{C}(\theta; \mathbf{y}_b, \hat{\mathbf{y}}_b) = -\frac{1}{B}\sum_{i=1}^{B}\sum_{d} y_d^{(i)} \log(\hat{y}_d^{(i)}) + \alpha||\text{vec}(\theta_r)||^2,$$

where θ_r, which contains all L^2-regularized weights matrices and filters, is flattened to a vector its norm added as a penalty term. The influence of the penalty is controlled through the parameter α. This is to keep individual filters or weights from dominating the output of the network, which counteracts overfitting.

Dropout [36] is another form of regularization where neurons in a network are randomly *dropped* during training. Neuron dropping is controlled by the drop rate $\text{P}(d) \in (0, 1)$ which can be specific for each layer or even each neuron. The dropping of neurons occur at the start of each batch, and the output of all dropped neurons are then fixed at zero. After parameters have been updated all neurons are brought back online, then a new set of randomly selected neurons are dropped for the next batch. This forces the network to train on randomly selected sub-networks of itself at each iteration, which again keeps individual features or weights from dominating the decision process. During evaluations of the network, however, the network should use all its capacity and so all neurons are made available once training is complete.

Chapter 6

Experiments

In this chapter we present and explain the implementations for experiments outlined in Section 2.3. There are three main categories of experiments: hyperparameter calibrations for single word classifiers, construction of multi-word classifiers and lastly testing the performance of a classifier trained with single word utterances on spontaneous speech. Before the details of each experiment are given, some prerequisites are presented.

6.1 Generalization

When performing classification with trainable models, one of the most important goals is to construct a model which can accurately classify data it has not seen before. This is the concept of generalization. To test generalization it is necessary to refrain from training classifiers on all available data and instead set aside some data for evaluation, emulating the concept of unseen data. Similarly, any pre-processing steps which involve aggregated information about the data, such as normalization, is only calculated from training data, but applied to all data.

Furthermore, when the performance of multiple versions of a classifier (differing in hyperparameters) are compared, the winner's accuracy will have a selection bias. Essentially the classifier which performs the best on the held out data, might do so because of some intrinsic property of that specific evaluation set.

Thus it is necessary to further split the data, setting aside yet another set which the winning model can be evaluated on, producing a fair performance estimate. In total, data is split into three sets: training which is used when training classifiers, validation used for comparing their performance, and testing which accurately measures the performance of the model with the best validation performance. All splits are made randomly. Here a typical 70% for training and 15% for validation and testing each is used if nothing else is specified.

In all different experiments of this thesis, there are multiple samples coming from the same speaker. For example in single word experiments each subject leaves on average five utterances of each word. Therefore it is necessary that all splitting of the data is made based on speakers and not samples, meaning that all samples from a speaker must be in the same subset (training, validation or testing). If a certain speaker has samples present in both training and evaluation sets, the properties of their speech is not strictly unseen and result will be skewed.

6.1.1 K-Fold Cross-Validation

One problem when splitting data in this fashion is that the outcome, i.e. the test accuracy, is dependent on which data happened to be chosen as test data. If a particularly difficult or simple sample ends up in the validation or test set, it might skew the results.

K-fold cross-validation is a method for mitigating this variance dependent on splits. Instead of just splitting data once, the data is randomly divided into k subsets. One at a time, the subsets take turns being the evaluation set, with the other $k-1$ subsets making up the training set. This means every sample will be in the evaluation set exactly once, and the average of the validation performance for all k folds is a more robust performance measure. This procedure is most commonly used after a testing set has been set aside and the remaining data is to be divided into training and validation. But one can also go a step further and cross-validate the test set *and* the validation set, as a sort of nested k-fold cross-validation.

6.2 Metrics

To measure performance of trained classifiers we employ the accuracy score as metric. A trained classifier can map a set of samples $x_1, \ldots, x_N$ to either predicted labels $\hat{Y}_1, \ldots, \hat{Y}_N$ or predicted class densities $\hat{y}_1, \ldots, \hat{y}_N$. If a classifier produces densities we pick the most probable class as the prediction by

$$\hat{Y}_i = \underset{d=1,\ldots,D}{\arg\max}\ y_{i,d} \quad \text{for} \quad i = 1, \ldots, N.$$

The accuracy score of the classifier on the dataset is then the percentage of correctly predicted labels

$$\frac{100}{N} \sum_{i=1}^{N} \mathbb{I}_{Y_i = \hat{Y}_i}.$$

When predicting class labels, every error will, from the point of view of a class d, be interpreted in one of two possible ways. Either a sample that is of class d is predicted to be of some other class, called a false negative, or a sample which is not in class d is mistakenly predicted as such, called a false positive. To quantify these errors we introduce the metrics recall and precision, which we express using the indicator function $\mathbb{I}$.

The *recall* of a classifier with regards to class d is the proportion of samples in category d that are also classified as d, given by

$$\frac{\sum_{i=1}^{N} \mathbb{I}_{Y_i = \hat{Y}_i = d}}{\sum_{i=1}^{N} \mathbb{I}_{Y_i = d}}.$$

As such recall puts a number on how well the classifier covers all samples from the d category.

Precision on the other hand quantifies the false positives of the classifier with respect to some class d. It is defined as

$$\frac{\sum_{i=1}^{N} \mathbb{I}_{Y_i = \hat{Y}_i = d}}{\sum_{i=1}^{N} \mathbb{I}_{\hat{Y}_i = d}}$$

which is the proportion of samples predicted as class d that are truly belonging to class d. A high precision would indicate that samples assigned the label d most likely actually belongs to that class.

6.2.1 Confusion Matrix

The confusion matrix is a way of visually illustrating the precision and recall for a classifier regarding all classes at the same time. It also shows exactly which classes are mistaken for each other by the classifier.

In a classification problem with D classes the confusion matrix is a $D \times D$ matrix where rows indicate the true label of samples, and columns indicate predicted labels. Each element $C_{i,j}$ in a confusion matrix is how many samples from class i were predicted to be from class j. On this form a perfect classifier would only be non-zero on the diagonal.

We define the normalized confusion matrix as the confusion matrix but normalized such that each row sums to 1. The normalization means that if an element on the diagonal is equal to one, then all samples from that class are predicted to be in that class, indicating perfect recall. However samples from other classes could be mistakenly predicted as belonging to this class as well, meaning precision is not necessarily perfect in this case.

6.3 Single-word Calibration

As stated above the calibration of each classifier to single word dialect classification is done by focusing on one of the calibration words *dörr*, *flytta*, *kaka* and *lär* at a time. For each word all utterances available in the *wordlist* data is extracted with information about speaker id. and dialect. All utterances shorter than one frame (25ms) were discarded since they cannot be used to construct features. On the other end all samples longer than 2 seconds were discarded since they were found to often contain irrelevant additional sound bedsides the sought after calibration word utterance. For each of the calibration words there are around 1400 samples.

To mitigate the variance in test accuracy 5 fold cross-validation is employed, creating five non overlapping test sets. The remaining data in each fold is then split once more into five, where each fifth takes turns being the validation set. The remaining four subsets in the inner splits are training data. Each hyperparameter combination of a classifier is fit and evaluated on the five training and validation data pairs. From this an average accuracy on the validation sets is obtained for each hyperparameter combination, and the best one indicates the winner. This whole procedure is then repeated for all five test folds, producing five (possibly different) winning setups.

For each classifier, an overall winner is picked manually based on either most frequent wins or consistently high ranking among all five folds. The average performance of this calibrated version of each classifier on the test folds can then be used to compare the classifiers. Some of the classifiers do not need the validation set during training, therefore their calibrated winners are trained on the remaining data in each test-fold, i.e. no splitting to training and validation sets.

6.4 Multi-word Classification

To construct multi-word ensemble classifiers we reuse the optimal setup of each classification method as per the calibration described above. The procedure is the same for all classifiers: for each of the 43 *wordlist* words an independent calibrated classifier are trained on samples of only that word, which is extracted in the same fashion as for the calibration words. Then an ensemble classifier which uses the predicted class probabilities from multiple single word classifiers is constructed. Which single word classifiers to include is decided by forward selection.

The major difference between a single- and a multi-word classifier is that the latter takes an n-tuple of input samples, one for each of the n internal single word classifiers it consists of. Our implementation of a multi-word classifier predicts labels by sending each sample in the input tuple to the corresponding internal single word classifier. This produces n different class distributions that are merged by summation and the predicted label for the input tuple is then the argmax of the sum. This procedure is straightforward for neural networks, which use a softmax activation to produce output in $[0, 1]^D$. When GMMs are used in multi-word classifiers they are redefined to instead output the average log likelihood of each internal distribution given the sample, which can be summed in the same manner as a density.

6.4.1 Splitting

While the single word classifiers used as building blocks in the multi-word classifier are trained independently (and therefore can be done in parallel), evaluation on testing and validation data must be done with n-tuples of input samples. Exactly which words will be included is needs to be set beforehand, since it is necessary to be able to evaluate the classifier using all 43 words. For this to be possible the validation and test sets must be constructed so that for every speaker, at least one sample for each of the 43 words is present.

Therefore the validation and test sets are constructed around speakers rather than samples, so that each item in either set consists of utterances for all 43 words from a single speaker. In Table A.1 we see that there are not enough speakers in the data uttering all 43 words to enable folded cross-validation, for example in the Jämtland dialect only 33% of speakers have uttered all 43 words at least once. But there are enough speakers uttering all words for a static split with 15% validation and 15% test data. In fact there are a large enough subset of speakers uttering all words *at least twice*, meaning we can construct evaluation sets for the multi-word classifiers as follows.

For each dialect, approximately 30% of the number of speakers are selected from the speakers with at least two samples per word. These are divided evenly into test and validation speakers. This selection and division of speakers is performed exactly once and every multi-word simulation loads the same division. This stops any variance in results from including different people in validation/testing between simulations.

From this, test and validation sets are constructed so that for each speaker, two series of all 43 words are randomly sampled without replacement from their available

utterances. The limiting of each speaker only supplying two utterances per word is to keep a balance between dialects and speakers. This sampling of utterances, together with the stochasticity in classifier training, introduces randomness between simulations and some variance is expected between runs for multi-word classifiers.

Lastly the training sets are specific to each word, and therefore is unaffected if utterances for all 43 words aren't available for some speakers. There are 43 sets of training data, each one containing all utterances of a word except those coming from speakers selected for testing or validation. These training sets are what is used to initialize and train all 43 single word classifiers. Then comes the question of how to best combine them.

6.4.2 Forward Selection

To select an optimal subset of words for dialect classification we implement forward selection. For this we define two sets of words: W^+ which is initially empty, which holds the words added to the multi-classifiers and W^- which holds the words not yet added to the classifier, initially containing all 43 words. We define $A(W)$ to be the accuracy score of multi-word classifier using the set of words W. Forward selection is the iterative procedure where in each step the word $w \in W^-$ maximizing

$$A(W^+ \cup \{w\})$$

is added to W^+ and removed from W^-. This will grow the set of words one word at a time, and by saving the validation accuracy for the chosen set of words at each iteration as a_i where i goes from 1 to 43, we study how adding words affects performance.

To mitigate the mentioned randomness between runs this procedure is repeated three times for each classifier used as basis in multi-word classification. This means, that three times over, data is initialized, single word classifiers are trained, combined through forward selection, and the best model is evaluated on the test set.

6.5 Spontaneous Speech Classification

As we saw in Table 2.2, some dialects do not have sufficient annotation coverage for testing the performance on spontaneous speech. Therefore we restrict ourself to only three dialects: *Jämtland*, *Sydfinland* and *Skåne*. In each of these dialects around 30% of speakers have an annotated spontaneous interview, which means they can be used in the test and validation sets for this experiment. These speakers are split randomly into testing and training. For each such speaker, all annotated sound samples are extracted and trimmed if necessary to a maximal length of 10 seconds. For the remaining 70% of speakers, all utterances of the 43 *wordlist* words are extracted and used in the training set.

Of course, this is an easier problem than a 7-way classification, and earlier results will not be comparable. To get a fair reference the calibrated classifiers are trained and evaluated for each calibration word, using the same split of speakers as above on the 3-way dialect classification.

Chapter 7

Results

7.1 Hyperparameter Calibration

Here we present which classifiers are used with which features for dialect classification. We also identify which tuneable hyperparameters each classifier and feature combination has.

7.1.1 Gaussian Mixture Models

As a dialect classifier, GMMs are combined with MFCC + SDC features. This combination is henceforth referred to in short as GMM. As MFCC feature vectors are coefficients of an orthogonal (cosine) basis, the assumption of diagonal covariance matrices made when defining GMMs is plausible. Of course the vectors might not be perfectly decorrelated, depending on properties of the underlying data.

The varying length of samples is not an issue for a GMM classifier. In training all samples in each dialect are combined when constructing that dialect model. And in prediction the average log-likelihood is insensitive to varying length.

The hyperparameters specifying this classifier are the number of components M in each dialect mixture model, and the quadruple (N, n, P, k) which states how MFCC and SDC vectors are generated, as explained in Section 4.3.1. A filterbank with 20 filters were used when generating MFCC features. Table 7.1 shows which values of hyperparameters were tested, all possible combinations of these were trained and evaluation according to the nested folding procedure outlined above. The table also shows which hyperparameter setup performed on average the best. Most notably the smallest number of mixture components performed the best, whereas in other dialect identification studies 256 (and even up to 2048 when a lot of data is available) mixture components have been used [8]. Most likely this is due to utterances of single words makes up less complex distributions than spontaneous speech and therefore not as many mixture components are needed. As indicated larger GMMs are also more common in situations where more data is available for training them.

7.1.2 Multilayer Perceptron

As a classifier for log signature feature vectors (based on MFCC paths) we have chosen a Multilayer Perceptron. The maximum log signature level is fixed to $\ell = 3$ and path dimension is fixed to $N = 10$. This results in vectorized log signatures that

Table 7.1: Tested hyperparameters when calibrating the Gaussian Mixture Model classifier for single word dialect classification.

Parameter	Values
M	**64**, 128, 256
N	8, **10**, 12
n	1, **3**, 5
P	**6**, 10
k	1, **2**, 3

are elements of $\mathbb{R}^{385}$. During training the MLP uses 32 samples per mini-batch and all weights and biases are L^2-regularized with the hyperparameter α.

In hyperoptimzation, network architecture, α and the number of training epochs are varied. Networks were designed so that the first hidden layer had $64, 128, 256$ and 512 neurons. All consecutive hidden layers then has half the amount of neurons as the previous one. Following this pattern networks two, three and four layers deep were constructed. Each network architecture was combined with all combinations of $\alpha = 0.0001, 0.001, 0.01$ and $N_{\text{epochs}} = 100, 500, 1000$ when evaluated according to the nested cross-validation scheme.

The results of this hyper-optimization were more inconclusive, with many different network architectures performing the best on different folds, although compared to the other classifiers they all performed poorly. The network with two hidden layers of size 512 and 256 was chosen together with $\alpha = 0.001$ and $\mathrm{N}_{\text{epochs}} = 500$ to play the role of calibrated multilayer perceptron.

7.1.3 Convolutional Neural Network

Convolutional neural networks are an extension of MLPs, and as such they have even more hyperparameters. Combined with longer training times, thorough parameter sweeps are not feasible. Instead, a number of setups were chosen for testing.

The number of convolutional layers was varied from a single layer to three consecutive layers, followed by two or three fully connected layers. After each convolutional layer, a 2-by-2 max-pooling layer was implemented. Filter sizes of 5-by-5, 10-by-5 and 5-by-10 were tested, and the number of filters in the convolutional layers were either 64, 32 or 16. Initial learning rates of 0.0001 and 0.001 were tested. All networks were designed with L^2-regularization on the fully connected layers, and a drop rate of 0.5 on all fully connected layers during training. All setups used ReLU activation functions for all hidden layers.

Performances were similar for all tested designs.The network with two convolutional layers with 32 and 16 filters respectively, followed by two fully connected layers with 512 and 7 neurons, gave the highest accuracy.

Table 7.2: Average test set classification accuracy over 5-fold cross-validation on the four calibration words for hyperparameter-calibrated classifiers.

Classifier	Features	Calibration word performance (avg. %)			
		dörr	flytta	kaka	lär
GMM	MFCC + SDC	61.3	60.3	61.4	55.7
MLP	Log signature	42.5	41.7	46.0	39.3
CNN	MFSC	55.4	56.3	58.7	49.7
Modular	MFSC + Log Sig.	56.7	54.4	59.6	52.6

7.1.4 Modular Network

The modular network was not optimized for any parameters, partially due to time restrictions but also the even larger number of parameters available to choose from. Instead, the structure was chosen from the winning convolutional neural network along with a MLP structure of two hidden layers with 256 and 7 neurons each.

Since the modular network contains more parameters than the winning convolutional network, dropout regulation was used not only on the fully connected layers but also on the last convolutional layer.

7.2 Performance on Calibration Words

We now want to compare the performance of our calibrated classifier setups. It was found that performance of single word classification varied strongly between runs, depending on how data was split, which is the motivation for employing folded cross-validation. For example the test accuracy of the calibrated GMM varied between 52.2% and 60.0% on the five folds.

The average accuracies over folds for calibrated classifiers are shown in Table 7.2. The Gaussian mixture models gave better performance than any of the networks, outperforming the convolutional neural network with 3–g6%. While the higher accuracy of the GMM is consistent for all four words, the CNN and the modular net are quite close. A reason for this could be that GMMs are well known and fine tuned as a method, whereas CNNs are newer in this application and the optimal structure of layers might not have been found. Furthermore, the CNN could also suffer more than the GMM from the relatively small amount of available data.

The multilayer perceptron, however, lagged behind the GMM on average by 17.3% over the four words. This large discrepancy among otherwise similar performances could indicate that the log signatures passed to the MLP are not suitable for one-word dialect classification. A convolutional network based on log signatures is unlikely to perform better than the MLP since the log signatures do not contain local correlations, which CNNs is designed to capture.

Figures 7.1 and 7.2 show normalized confusion matrices from classification methods based on utterances of the word *kaka* with a GMM and CNN respectively (numerical versions of these matrices can be found in Appendix A). The confusion

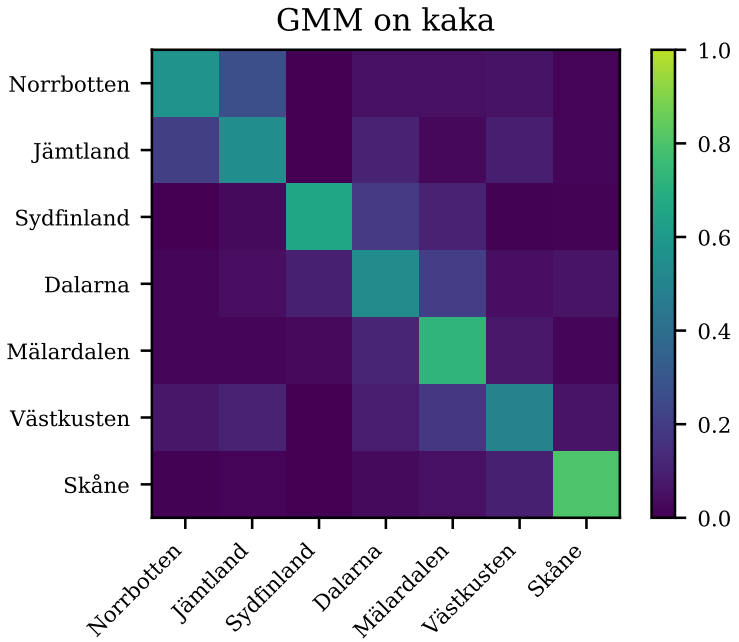

Figure 7.1: Normalized confusion matrix for a Gaussian mixture model on the word *kaka*.

matrices are basically in agreement between the two classifiers. Both classifiers reach highest precision and recall for *Skåne*, suggesting that this is the most distinguishable dialect. Also, *Norrbotten* and *Jämtland* have high confusion between them, which is in agreement with the dialect division done by Wessén as discussed in Section 2.2, where these two regions are considered to share the same dialect.

7.3 Multi-word Performance

In Table 7.2, the word *lär* yields significantly lower performance than the other three. This indicates that some words are more suitable than others for use in dialect classification, which is the motivation behind multi-word classifiers.

Each of the four classifiers were implemented as root for an ensemble classifier over words. As mentioned the experiment was repeated three times for each classifier and moderate variance in results was observed. The average results are shown in Figure 7.3. It appears that using multiple words during classification significantly improves classification performance.

Most surprisingly, the multilayer perceptron combined with log signatures, which had the worst performance on calibration words, catches up with the other ensembles in terms of validation accuracy. This behavior is similar to that of [4], where an accuracy of 76.92% was reported for seven North American dialects (TIMIT) using five words, with the best one-word classifier reaching 41.35% accuracy.

Table 7.3 shows the average number of words picked by forward selection and the average validation and test accuracy for the picked multi-word classifiers. The results suggests that the MLP classifier using log signature features benefit to a higher degree from the information in multiple words. While the GMM performed

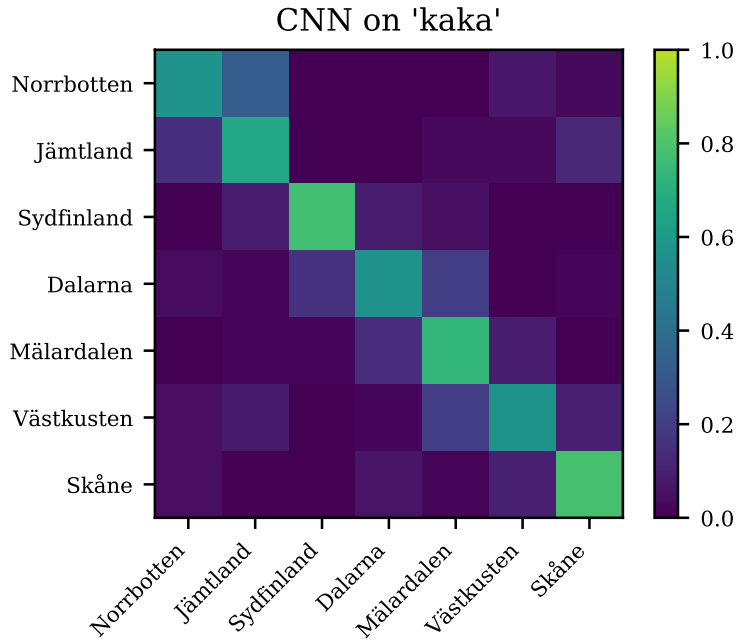

Figure 7.2: Normalized confusion matrix for a convolutional neural network on the word *kaka*.

best on the test set, the difference in accuracy compared to the other classifiers is barely significant. The GMM also required the fewest number of words, but again only with a small margin except for the MLP.

All four classifiers benefit from the ensemble setup with multiple words, reaching the best performance using around 15 words, after which accuracy actually starts dropping. This indicates that adding some words merely confuse the classification. While some words have higher individual accuracy than others, the forward selection doesn't simply choose the best individual words, which means that it is not necessarily the words with the lowest individual accuracy on their own that causes this degradation. One run chose *sitt* as the fifth word, while leaving *gata* at 34:th place, despite the words having an individual accuracy of 42% and 61% respectively. Hence, word combinations seem to matter for the classifiers. This suggests that different words can carry complimentary information that is valuable for dialect classification.

Table A.3 shows the order in which the forward-selections chose each word, averaged over three runs for each classifier. The fact that three of the calibration words are in the top 9 most chosen words indicates that results can be skewed. Because the calibration words were used to tune the classifiers, they are unfairly favored. A more robust method for a multi-word classifier would not just train a classifier for each word, but also tune it individually.

A normalized confusion matrix for a multiword classifier using GMMs can be seen in figure Figure 7.4 (again, a numerical equivalent can be found in Appendix A). Interestingly enough, *Skåne* is no longer the easiest dialect to classify, and the confusion between *Norrbotten* and *Jämtland* is much less pronounced. A notable trend here would be that many persons are incorrecly predicted to come from *Dalarna*.

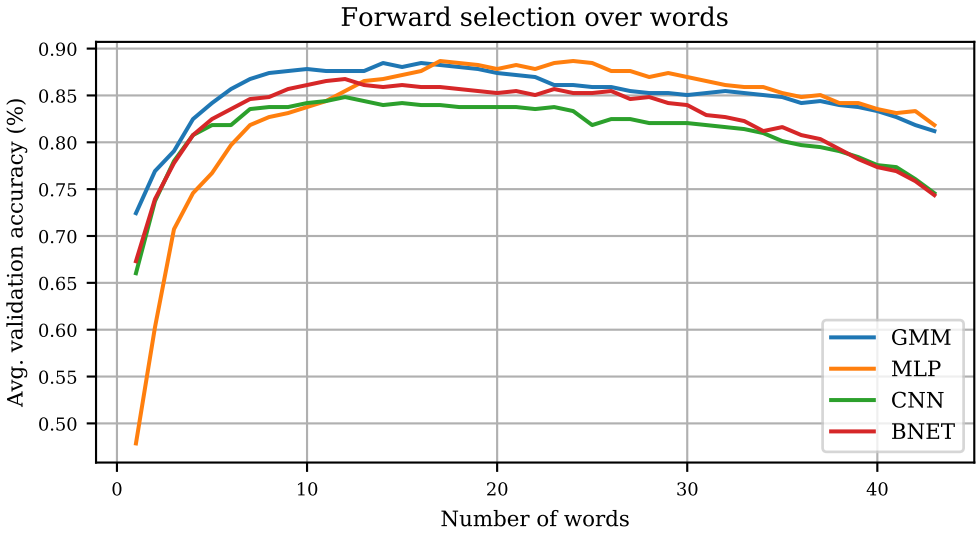

Figure 7.3: Average validation set accuracy over three independent training runs for ensemble classifiers combining multiple single-word classifiers via forward selection.

Table 7.3: Average classification accuracy and number of words selected over three random initializations of multiword data.

Classifier	Average forward select results		
	Num. of words	Val. acc. (%)	Test acc. (%)
GMM	12.0	88.6	80.1
MLP	19.3	89.1	79.3
CNN	14.3	84.9	78.0
Modular	13.7	87.4	79.7

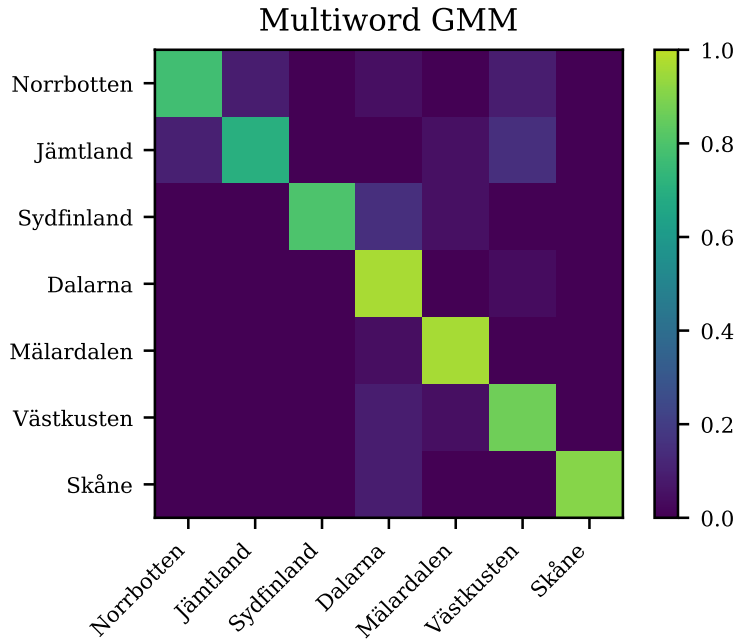

Figure 7.4: Normalized confusion matrix for a multi-word GMM using a forward selected set of words.

7.4 Other Signature-based Implementations

During the course of this project, several signature-based implementations were investigated, however, initial results were not promising. In [14], signatures were calculated on moving windows along a path. Inspired by this, we implemented a short-time signature transform in combination with convolutional networks, analogously to the short-time Fourier transform in Section 4.1, but classification was very poor.

Signatures were also computed from MFSCs, rather than MFCCs as in the results above, but yet again no increased performance was observed. Trials were run using both GMMs and MLPs. Signatures were investigated as a feature to provide temporal information for GMMs by replacing shifted delta-cepstra with log signatures. Paths derived from lead-lag transforms [19] were attempted, but generally led to very low classification accuracy.

In [15] the path is divided into sub-paths, motivated by the idea that multiple signatures with few levels capture more information than one single signature with many levels. This was implemented in combination with MFCC features and convolutional networks but did not surpass the performance of a single large signature.

7.5 Spontaneous Speech Classification

For the simplified problem of discerning the dialects *Jämtland*, *Sydfinland* and *Skåne*, the calibrated classifiers gained much higher accuracies. A calibrated GMM reached classification accuracies of 87.6% on *dörr*, 75.2% on *flytta*, 76.1% on *kaka* and 67.5% on *lär*. This is a significant gain in accuracy compared to the classification with

secen dialects, which is to be expected. All of these classifiers had test accuracies that were around 10% higher than their validation accuracies. This indicates that this specific split caused the test set to be easier, and highlights the importance of folded cross validation, which unfortunately is impossible with the current data.

When training a GMM on all 43 words, it is necessary to increase the number of components. Rudimentary hyperparameter optimization showed that 1024 mixture components were suitable for this task, achieving a validation accuracy of 59.0%. The test accuracy for this GMM was 69.0%. While this is not on par with the single word classifiers, it shows that there is at least some overlap in dialectal features between utterances of words and spontaneous speech.

7.6 Effect of Speaker Identity

As discussed in Section 6.1, the experiments in this thesis was conducted by dividing samples in the data into training, validation and test sets based on speaker identity. However, the relatively low number of unique speakers leads to high variance in the experiments, as accuracy can be heavily influenced by a difficult speaker being present in the test or validation set. This suggests that the number of speakers is important to achieve decent classification.

In fact, speaker identity appear to be a very significant factor. During this investigation, a single CNN was trained and evaluated on all 43 *wordlist* words. Classification on data splits that allows for samples from a particular speaker being present in training, validation and test sets reached over 90% accuracy for seven dialects. While repeated utterances of the same word by the same speaker are very similar, as shown in the two upper panels in Figure 7.5, and therefore easily identified if such utterances are found in both training and test sets, such cases does not explain the entire gain in accuracy. This was illustrated by an experiment in which data again was split so that we allowed the same speakers to be present in all three sets, but now restricted utterances of particular words to be present only in one set. This experiment reached close to 80% for seven dialects, which suggests that the models more easily picks up on speaker identity itself, even when it is exposed to words that it has never encountered before.

In Figure 7.5, it is perhaps possible that the pronunciations of the vowel *a*, visible as a tilted V-shape in the spectrogram, are similar for the samples from Speaker 1 (all but the bottom right pane). Such local features could feasibly explain why speaker identity appears to be a dominating source of variance in our data set.

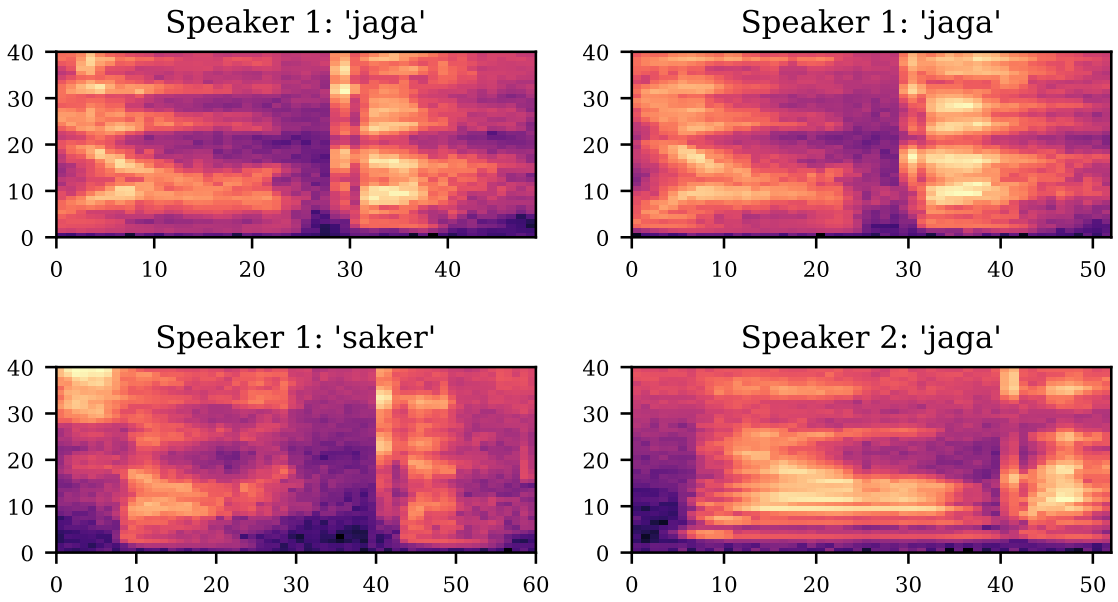

Figure 7.5: Four word utterances. The two upper figures are from the same speaker uttering the word *jaga*. The bottom left is also from this speaker, uttering *saker*. Bottom right is again *jaga* but from a different speaker.

Chapter 8

Conclusions

The results of this thesis suggest that classification of Swedish dialects based on acoustic features is possible and can be done with similar performance as in other languages. As the SweDia2000 project continues to annotate their database, it is not unfeasible that future studies can also test this on entirely spontaneous data. Furthermore, state of the art dialect classifiers use combinations of the different features such as the lexical or prosodical features, whereas here only acoustic ones were used. Such a pooling of information about samples will very likely push performance even further.

In this thesis we have only looked at dialect classification itself. It would be of interest to evaluate the intended boost in speech recognition performance in Swedish that would come from tailoring automatic speech recognition methods for each dialect. Using such a system it would also be possible which dialects (and which formulations of them) are distinct enough

While the established Gaussian mixture models and convolutional neural networks were superior for single-word classification, the relatively simple feedforward network applied to the log signature of Mel-frequency cepstrums caught up in performance as a multi-word classifier. This is both remarkable and suspicious, and may indicate that a larger data set should be used to verify the result. Many different methods for incorporating signatures or log signatures into speech features were tested without success, but the search has by no means been exhaustive. It is certainly possible that the optimal way of utilizing signatures is yet to be discovered.

Across the board, the experiments in this thesis has been formulated with the available data in mind. In the end, many experiments were on the edge of what is acceptable in terms of data size. It would therefore be of interest to expand the dialect regions, most likely resulting in more robust experiments. This would at the same time decrease the margin between them, and depending on the outcome of classification on such regions, the idea of smooth transitions between dialects can be tested.

Appendix A

Tables

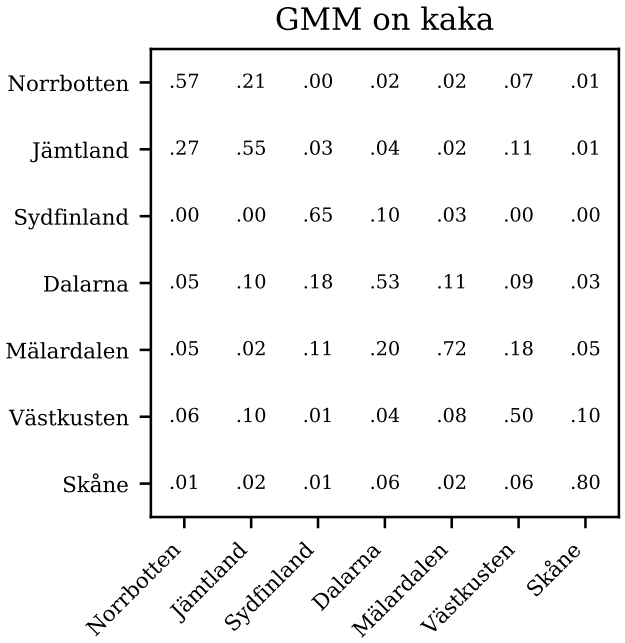

Figure A.1: Normalized confusion matrix for the Gaussian mixture model on the word *kaka*.

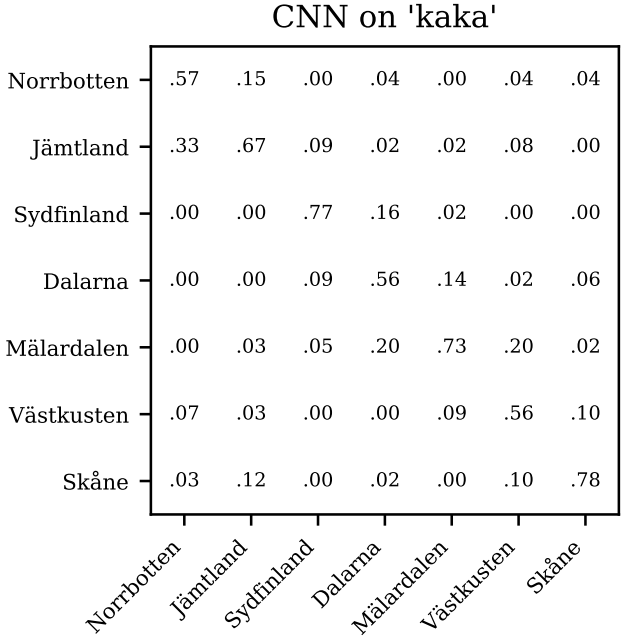

Figure A.2: Normalized confusion matrix for the Convolutional network on the word *kaka*.

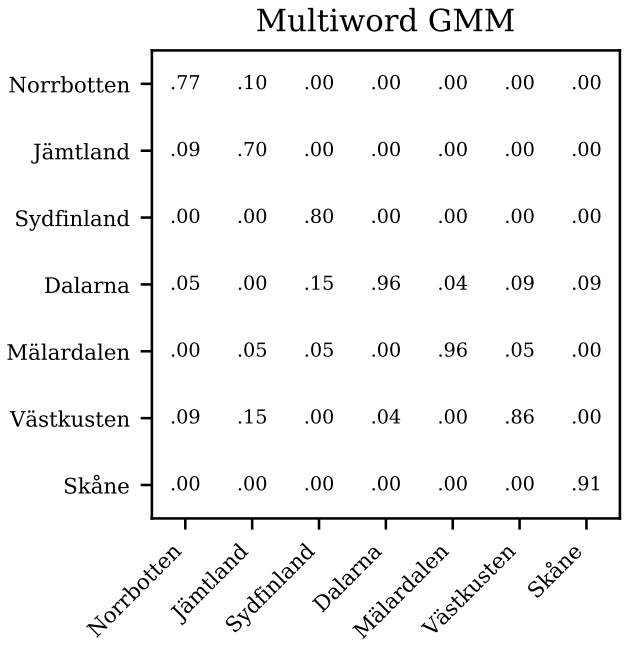

Figure A.3: Normalized confusion matrix for a multi-word GMM using a forward selected set of words.

Table A.1: The number of unique speakers in each dialect region as well as information about how many speakers in each dialect provides at least n utterances for every *wordlist* word.

Dialect	Unique speakers	$n = 1$	$n = 2$	$n = 3$	$n = 4$	$n = 5$
Norrbotten	79	30 (38.0%)	24 (30.4%)	20 (25.3%)	16 (20.3%)	4 (5.1%)
Jämtland	72	24 (33.3%)	24 (33.3%)	22 (30.6%)	17 (23.6%)	5 (6.9%)
Sydfinland	70	33 (47.1%)	30 (42.9%)	28 (40.0%)	2 (2.9%)	0 (0.0%)
Dalarna	87	43 (49.4%)	41 (47.1%)	32 (36.8%)	6 (6.9%)	0 (0.0%)
Mälardalen	82	71 (86.6%)	71 (86.6%)	61 (74.4%)	6 (7.3%)	0 (0.0%)
Västkusten	75	61 (81.3%)	60 (80.0%)	58 (77.3%)	28 (37.3%)	2 (2.7%)
Skåne	75	62 (82.7%)	61 (81.3%)	60 (80.0%)	28 (37.3%)	2 (2.7%)

Table A.2: Number of utterance available per word in each dialect in the *wordlist* category. Only words occurring frequently across all seven dialects are included.

	Västk.	Sydfin.	Jämtland	Skåne	Norrb.	Dalarna	Mälard.
lott	741	282	373	698	292	336	328
lär	387	302	456	383	411	429	411
dagar	384	271	384	403	353	363	357
kaka	335	266	414	347	360	358	357
dörr	343	290	382	330	333	365	354
flytta	375	283	365	339	340	348	346
panna	333	292	374	317	376	346	355
saker	347	262	387	332	360	351	352
blöt	350	257	380	354	341	361	343
käke	347	274	385	327	356	361	333
fara	345	264	384	317	348	377	347
nät	338	293	353	332	349	358	359
leta	353	277	369	316	346	369	350
särk	318	301	402	331	342	347	332
lat	335	284	366	341	338	358	349
sitt	346	268	364	332	355	342	363
söt	356	277	367	346	349	344	327
nagel	350	257	363	333	362	348	352
lass	337	279	367	320	347	345	359
själen	353	239	367	334	346	353	360
mage	334	256	357	346	347	364	346
hage	358	248	362	317	356	363	341
dör	351	267	384	327	305	356	355
gata	345	256	364	343	347	347	342
sot	348	279	368	341	295	362	349
disk	337	274	370	325	343	348	345
tala	328	252	373	324	365	349	348
lös	332	276	382	332	342	342	332
vana	341	262	373	323	348	346	344
lus	329	290	368	333	333	348	335
jaga	327	263	375	326	338	352	351
rasa	352	255	352	331	343	357	334
bada	329	259	367	323	352	353	336
lett	355	219	377	347	349	333	338
lång	357	265	352	317	306	355	363
laga	345	249	355	319	344	361	341
dis	336	276	383	339	260	358	336
såll	329	250	372	330	290	333	337
sytt	328	242	393	326	288	305	344
ludd	331	256	360	350	308	278	337
dalar	355	249	235	328	327	362	343
tysk	334	254	364	329	227	347	341
typ	337	282	306	322	305	293	341
Total:	19482	15239	18939	18792	17204	17914	17332

Table A.3: Average ranking of words in forward selection, sorted from best to worst overall ranking. Blue rows indicate words used for calibration.

Word	GMM	MLP	CNN	BNET	Overall
saker	1	15	2	3	5
själen	9	2	2	11	6
leta	15	5	17	12	12
kaka	2	12	14	26	14
flytta	11	19	9	15	14
blöt	17	12	9	19	15
disk	22	11	15	14	16
dörr	10	17	24	14	17
särk	7	9	28	23	17
typ	15	26	18	8	17
dör	28	2	23	15	17
vana	21	27	13	10	18
rasa	13	24	16	21	19
lat	9	22	35	11	19
lång	25	15	27	11	19
lass	12	30	7	30	20
lett	31	8	21	20	20
hage	30	16	23	11	20
dalar	7	29	26	18	20
tala	30	6	23	23	21
sot	16	15	26	25	21
laga	33	20	17	14	21
nagel	11	27	13	36	22
gata	14	35	18	22	22
lös	24	26	20	21	23
dis	23	20	22	27	23
ludd	36	23	21	17	24
fara	30	18	23	28	25
lott	13	30	23	35	25
panna	35	25	27	18	26
käke	29	16	29	33	27
jaga	22	42	17	26	27
bada	20	33	32	23	27
nät	22	31	21	34	27
dagar	23	29	31	27	27
mage	33	27	33	25	30
sitt	32	32	27	28	30
sytt	35	18	35	32	30
lus	26	40	27	33	31
söt	42	35	23	29	32
lär	43	33	37	17	32
såll	38	28	34	40	35
tysk	30	34	38	39	35

Bibliography

1. A. Eriksson, *Swedia recording locations*, http://www.ling.gu.se/~anders/SWEDIA/kartor/Map.pdf, 2016, Online, accessed 2017-05-08.

2. H. Petkar, *A review of challenges in automatic speech recognition*, International Journal of Computer Applications **151** (2016), no. 3, 23–26, DOI: 10.1016/j.specom.2007.02.006.

3. F. Biadsy, *Automatic dialect and accent recognition and its application to speech recognition*, Ph.D. thesis, Columbia University, 2011, URL: http://www.cs.columbia.edu/speech/ThesisFiles/fadi_biadsy.pdf.

4. M. Rizwan, B. O. Odelowo, and D. V. Anderson, *Word based dialect classification using extreme learning machines*, Neural Networks (IJCNN), 2016 International Joint Conference on, IEEE, 2016, pp. 2625–2629, DOI: 10.1109/IJCNN.2016.7727528.

5. N. F. Chen, W. Shen, J. P. Campbell, and P. A. Torres-Carrasquillo, *Informative dialect recognition using context-dependent pronunciation modeling*, Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on, IEEE, 2011, pp. 4396–4399, DOI: 10.1109/ICASSP.2011.5947328.

6. S. A. Majeed, H. Husain, S. A. Samad, and T. F. Idbeaa, *Mel frequency cepstral coefficients (mfcc) feature extraction enhancement in the application of speech recognition: A comparison study*, Journal of Theoretical and Applied Information Technology **79** (2015), no. 1, 38.

7. A. Faria, *Accent classification for speech recognition*, vol. 3869, pp. 285–293, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, DOI: 10.1007/11677482_25.

8. P. A. Torres-Carrasquillo, T. P. Gleason, and D. A. Reynolds, *Dialect identification using gaussian mixture models*, ODYSSEY04-The Speaker and Language Recognition Workshop, 2004, pp. 297–300, URL: http://www.isca-speech.org/archive_open/archive_papers/odyssey_04/ody4_297.pdf.

9. M.-L. Gu and B. Zhang, *Chinese dialect identification using sc-gmm*, Advanced Materials Research, vol. 433-440, 2012, pp. 3292–3296, DOI: 10.4028/www.scientific.net/AMR.433-440.3292.

10. S. Deshpande, S. Chikkerur, and V. Govindaraju, *Accent classification in speech*, Fourth IEEE Workshop on Automatic Identification Advanced Technologies (AutoID'05), vol. 2005, IEEE, 2005, pp. 139–143, DOI: 10.1109/AUTOID.2005.10.

11. J. Pan, C. Liu, Z. Wang, Y. Hu, and H. Jiang, *Investigation of deep neural networks (dnn) for large vocabulary continuous speech recognition: Why dnn surpasses gmms in acoustic modeling*, Chinese Spoken Language Processing (ISCSLP), 2012 8th International Symposium on, IEEE, 2012, pp. 301–305.

12. G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury, *Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups*, IEEE Signal Processing Magazine **29** (2012), no. 6, 82–97.

13. T. Lyons, *Differential equations driven by rough signals*, Revista Matemática Iberoamericana **14** (1998), no. 2, 215–310 (English), URL: http://intlpress.com/site/pub/files/_fulltext/journals/mrl/1994/0001/0004/MRL-1994-0001-0004-a005.pdf.

14. B. Graham, *Sparse arrays of signatures for online character recognition*, arXiv preprint (2013), arXiv:1308.0371.

15. T. J. Lyons and N. Sidorova, *Sound compression: a rough path approach*, Proceedings of the 4th international symposium on Information and communication technologies, Trinity College Dublin, 2005, pp. 223–228, URL: http://www.ucl.ac.uk/~ucahnsi/Papers/lyons_sidorova_capetown.pdf.

16. O. Engstrand, R. Bannert, G. Bruce, C.-C. Elert, and A. Eriksson, *Phonetics and phonology of swedish dialects around the year 2000: a research plan*, FONETIK 98, PHONUM **4** (1997), 97–100, URL: https://pdfs.semanticscholar.org/df3e/075333e55b95a749bdee9026eb09df5ba93d.pdf.

17. E. Wessén, *Våra folkmål*, 4 ed., Fritze, 1969 (Swedish).

18. F. Schaeffler, *Phonological quantity in swedish dialects : Typological aspects, phonetic variation and diachronic change*, Ph.D. thesis, Umeå University, Philosophy and Linguistics, 2005, p. 142, URL: http://umu.diva-portal.org/smash/get/diva2:143871/FULLTEXT01.pdf.

19. I. Chevyrev and A. Kormilitzin, *A Primer on the Signature Method in Machine Learning*, ArXiv e-prints (2016), arXiv:1603.03788.

20. P. Lalonde and A. Ram, *Standard lyndon bases of lie algebras and enveloping algebras*, Transactions of the American Mathematical Society **347** (1995), no. 5, 1821–1830, DOI: 10.1090/S0002-9947-1995-1273505-4.

21. J. Reizenstei, *Calculation of iterated-integral signatures and log signatures*, 2016, URL: http://www2.warwick.ac.uk/fac/cross_fac/complexity/people/students/dtc/students2013/reizenstein/logsignatures.pdf.

22. J. Berstel and M. Pocchiola, *Average cost of duval's algorithm for generating lyndon words*, Theoretical Computer Science **132** (1994), no. 1, 415–425, DOI: 10.1016/0304-3975(94)00013-1.

23. J. P. Duval, *Factorizing words over an ordered alphabet*, Journal of Algorithms **4** (1983), no. 4, 363 – 381, DOI: 10.1016/0196-6774(83)90017-2.

24. L. M. Arslan and J. H. L. Hansen, *Language accent classification in american english*, Speech Communication **18** (1996), no. 4, 353–367.

25. V. Arora, P. Sood, and K. U. Keshari, *A stacked sparse autoencoder based architecture for punjabi and english spoken language classification using mfcc features*, 2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom), Bharati Vidyapeeth, 2016, pp. 269–272.

26. K. Paliwal and K. Wojcicki, *Effect of analysis window duration on speech intelligibility*, IEEE Signal Processing Letters **15** (2008), 785–788, URL: https://research-repository.griffith.edu.au/bitstream/handle/10072/23589/53526_1.pdf.

27. J. Volkmann, S. S. Stevens, and E. B. Newman, *A scale for the measurement of the psychological magnitude pitch*, The Journal of the Acoustical Society of America **8** (1937), no. 3, 208–208.

28. M. A. Kohler and M. Kennedy, *Language identification using shifted delta cepstra*, Circuits and Systems, 2002. MWSCAS-2002. The 2002 45th Midwest Symposium on, vol. 3, IEEE, 2002, pp. III–69.

29. P. A. Torres-Carrasquillo, E. Singer, M. A. Kohler, R. J. Greene, D. A. Reynolds, and J. R. Deller Jr, *Approaches to language identification using gaussian mixture models and shifted delta cepstral features*, Interspeech, 2002, URL: https://pdfs.semanticscholar.org/4c0d/706b7fd1ab191c733e83ab0843e0d5d88f30.pdf.

30. D. R. González and J. Calvo de Lara, *Speaker verification with shifted delta cepstral features: its pseudo-prosodic behaviour*, Proceedings on I Iberian SLTech (2009), 85–88, URL: http://m.isca-speech.org/archive_open/sltech_2009/papers/isl9_085.pdf.

31. Y. Lei and J. H. L. Hansen, *Dialect classification via text-independent training and testing for arabic, spanish, and chinese*, IEEE Transactions on Audio, Speech, and Language Processing **19** (2011), no. 1, 85–96, DOI: 10.1109/TASL.2010.2045184.

32. D. A. Reynolds and R. C. Rose, *Robust text-independent speaker identification using gaussian mixture speaker models*, IEEE Transactions on Speech and Audio Processing **3** (1995), no. 1, 72–83, URL: http://www.cs.toronto.edu/~frank/csc401/readings/ReynoldsRose.pdf.

33. J. Wu, *Advances in k-means clustering: A data mining thinking*, 1 ed., Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, DOI: 10.1007/978-3-642-29807-3.

34. C. Elkan, *Using the triangle inequality to accelerate k-means*, ICML, vol. 3, 2003, pp. 147–153, URL: http://www.aaai.org/Papers/ICML/2003/ICML03-022.pdf.

35. D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, arXiv preprint (2014), arXiv:1412.6980.

36. N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, *Dropout: a simple way to prevent neural networks from overfitting*, Journal of Machine Learning Research **15** (2014), no. 1, 1929–1958, URL: https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf.