UNIVERSITY OF
CAMBRIDGE
Department of Engineering

Monoids: efficient segmental features for speech recognition

R. C. van Dalen
rcv25@cam.ac.uk

M. J. F. Gales
mjfg@eng.cam.ac.uk

Technical Report CUED/F-INFENG/TR.687

August 2013

Abstract

Recently, there has been interest in speech recognition with log-linear models that use features for whole segments, for example, words. The segmentation is often taken from a conventional speech recogniser. However, this limits the performance of moving to a new model. An alternative is to find the optimal segmentation. This requires acoustic features for all possible segments, which a recently proposed method extracts efficiently. It shares computation between features for segments with the same start time. This is useful when all segments are considered, but feature extraction still takes quadratic time in the length of the utterance. A more realistic strategy for decoding would prune the hypothesis space. This report therefore proposes a new, more flexible class of features. When features for all segments are required, extracting them has the same time complexity, but when only a limited number of segments are considered, they allow more re-use of computation. A specific subclass of features of interest derives from the total weight of a hidden Markov model (HMM) or a similar finite-state model. This report shows how to compute scores efficiently for such a finite-state model with weights in any *semiring*.

1 Introduction

State-of-the-art speech recognisers are usually based on hidden Markov models (HMMs). They model a hidden symbol sequence with a Markov process, with the observations independent given that sequence. These assumptions yield efficient algorithms, but limit the power of the model.

Recently, there has been interest in discriminative log-linear models that can deal with a wide range of features extracted from spans longer than one frame and of variable length (e.g., Layton 2006; Zweig and Nguyen 2009; Gales and Flego 2010). When using longer-span features to recognise continuous speech, segmentations into, e.g., words must be found explicitly. Existing approaches for structured models often derive a segmentation from a conventional speech recogniser. However, these segmentations are not optimal for the structured model, and may limit the performance gains from moving to a more powerful model. It is therefore desirable for the decoding process to find the optimal combination of word sequence and segmentation.

Ragni and Gales (2012); Van Dalen *et al.* (2013) have proposed an efficient method for extracting features for one word (or other unit of speech) from every possible contiguous segment of audio. The specific features are in *generative score-spaces*. These contain log-likelihoods for word HMMs (Ragni and Gales 2012) and their derivatives (van Dalen *et al.* 2013). The number of possible segments is quadratic in the length of the audio. By re-using part of the computation, the time needed to extract the features for all segments also becomes quadratic. If features for all segments are needed, feature extraction therefore takes average constant time.

However, a realistic speech recogniser will not need features for all these segments, because it will approximate its hypothesis space. One approximation scheme is to take hypotheses from an existing decoder and to compute features only for segments around ones contained in these hypotheses. It is therefore an interesting question what types of features can be efficiently computed for segments that can overlap in different ways. Of particular interest are segments that start and end around two times given by the arc in a lattice.

This report will discuss log-linear models to show where word-level features come in (section 2). It will then re-analyse the method in van Dalen *et al.* (2013) to highlight what makes it efficient. Computing the feature for a word can be phrased in terms of the primitive that pure functional programming languages use for iteration: the higher-order function "fold" (section 3). The method assumes a schedule that progressively computes features for all segments that start at the same time. This paper will then introduce a related form of feature. Used with the same schedule, it is as time-efficient, but can also be used within different types of schedules (section 4). These features are in a *monoid*. This makes it possible to combine two features for any two consecutive segments into one value for the segment encompassing both.

2 Log-linear models

Discriminative models (Gales *et al.* 2012) are probabilistic models that can operate on a wide range of features derived from the same segment of audio. Unlike a generative model, a discriminative model for speech recognition directly yields the posterior probability of the word sequence $\boldsymbol{w}$ given the observation sequence $\mathbf{O}$. Here, each of

the elements w_i of $\boldsymbol{w}$ is equal to one element v_j from the vocabulary $\boldsymbol{v}$. To enable compact discriminative models to be trained, the input sequence must be segmented into, e.g., words. Let $\mathbf{s} = \{s_i\}_{i=1}^{|\boldsymbol{w}|}$ denote a segmentation. This paper will use a log-linear model:

$$P(\boldsymbol{w}, \mathbf{s}|\mathbf{O}; \boldsymbol{\alpha}) \triangleq \frac{1}{Z(\mathbf{O}, \boldsymbol{\alpha})} \exp\Big(\boldsymbol{\alpha}^T \boldsymbol{\phi}(\mathbf{O}, \boldsymbol{w}, \mathbf{s})\Big). \tag{1}$$

Here, $Z(\mathbf{O}, \boldsymbol{\alpha})$ is the normalisation constant. $\boldsymbol{\phi}(\mathbf{O}, \boldsymbol{w}, \mathbf{s})$ is the score function that returns a score vector characterising the whole observation sequence. $\boldsymbol{\alpha}$ is the parameter vector.

For simplicity, this work will assume there is no language model. The distribution then factorises over the segments of the audio, i.e. the score function is a sum of scores for each segment:

$$\boldsymbol{\phi}(\mathbf{O}, \boldsymbol{w}, \mathbf{s}) \triangleq \sum_i \boldsymbol{\phi}(\mathbf{O}_{s_i}, w_i), \tag{2}$$

where $\mathbf{O}_{s_i}$ indicates the observations in segment s_i.

For decoding, it is in theory possible to marginalise out the segmentation. However, this is infeasible, so instead the segmentation and word sequence that maximise the posterior in (1) will be found:

$$\begin{aligned} \arg\max_{\boldsymbol{w},\mathbf{s}} P(\boldsymbol{w}, \mathbf{s}|\mathbf{O}; \boldsymbol{\alpha}) &= \arg\max_{\boldsymbol{w},\mathbf{s}} \frac{1}{Z(\mathbf{O}, \boldsymbol{\alpha})} \exp\Big(\boldsymbol{\alpha}^T \boldsymbol{\phi}(\mathbf{O}, \boldsymbol{w}, \mathbf{s})\Big) \\ &= \arg\max_{\boldsymbol{w},\mathbf{s}} \Big(\boldsymbol{\alpha}^T \boldsymbol{\phi}(\mathbf{O}, \boldsymbol{w}, \mathbf{s})\Big) = \arg\max_{\boldsymbol{w},\mathbf{s}} \sum_i \boldsymbol{\alpha}^T \boldsymbol{\phi}(\mathbf{O}_{s_i}, w_i). \end{aligned} \tag{3}$$

There are two aspects to performing this optimisation. First, the score $\boldsymbol{\phi}(\mathbf{O}_{s_i}, w_i)$ must be extracted for all possible words and segments. Second, the best combination of word sequence and segmentation must be found. Assuming the language model constant, the latter task takes $\Theta\big(T^2\big)$ time Ragni and Gales (2012). The former task, of extracting $\Theta\big(|\boldsymbol{v}| \cdot T^2\big)$ scores, forms the bottleneck in performance.

This report will consider the score for one word (or sub-word unit). To highlight the aspects that influence the efficiency of computing these scores, ϕ will be decomposed into two parts. First, a segmental *feature* is computed, and then it is converted into a *score*. The features are representations of segments of observations that allow them to be re-used; the scores are (often trivially) derived from the features. This report therefore focuses on the structure of the features.

First, section 3 will re-analyse the method in Ragni and Gales (2012); van Dalen *et al.* (2013) as computing a function of the form

$$\boldsymbol{\phi}(\mathbf{O}_{\tau:t}, w_i) = h\big(g(g(\dots g(g(\lambda, o_\tau), o_{\tau+1}) \dots, o_{t-1}), o_t)\big). \tag{4a}$$

Here, h is the word score function, and g a function that extends a feature by one observation o. To compute the feature for a segment, g is called recursively $t-\tau$ times. λ, the feature for the empty segment, is the base case for the recursion. (This will be formulated in terms of higher-order function "fold".) This recursion limits re-use of computation to segments that start at the same time τ.

In contrast, section 4 will introduce a less general but more generally applicable type of feature. The computation is of the form

$$\boldsymbol{\phi}(\mathbf{O}_{\tau:t}, w_i) = h\big(f(o_\tau) \odot f(o_{\tau+1}) \odot \ldots \odot f(o_t)\big), \tag{4b}$$

where f extracts a feature for one observation, and the operation $\odot$ combines two of these features. $\odot$ is required to be associative over the features, i.e., the result must be the same whichever order the features are combined in. This freedom allows many different pruning schemes.

3 Uni-directional segmental features

Ragni and Gales (2012); van Dalen *et al.* (2013) proposed a method for computing segmental features in average constant time if features for all segments are needed. This section will discuss this method at a high level. It will use a form that highlights its limitations and allows section 4 to relate this to a more general form. Section 3.1 will explain how this description maps to the specific instantiation in Ragni and Gales (2012); van Dalen *et al.* (2013).

Assume that the observations $o_t \in \mathcal{O}$ (for example, vectors with Mel-frequency cepstral coefficients) for an utterance of length T are available as $[o_1 \ldots o_T]$. Segmental features are to be extracted from each possible segment of consecutive observations. Two functions will act on this: first a function g that extends a feature by one observation, and a function h that computes a score given such a feature.

The features are in a feature space $\mathcal{F}$. The feature for the empty segment, which contains 0 observations, is defined as $\lambda \in \mathcal{F}$. Features can be extended in one direction. This is performed by the function

$$g : \mathcal{F} \times \mathcal{O} \to \mathcal{F}, \tag{5}$$

which takes a feature for a segment, and the next observation, and returns the feature for the segment extended with the observation.

Thus, the feature extracted from the segment containing just observation o_1 extends the feature for the empty segment with

$$f_1 \triangleq g(\lambda, o_1)\,. \tag{6a}$$

To compute the feature for observations $[o_1, o_2]$, the function is applied on this result again:

$$f_{1:2} \triangleq g(g(\lambda, o_1), o_2)\,. \tag{6b}$$

As an example, assume an observation sequence consisting of four elements $\mathbf{O} = [o_1, o_2, o_3, o_4]$. The feature for the whole sequence is computed with four consecutive applications of the function g:

$$f_{1:4} \triangleq g(g(g(g(\lambda, o_1), o_2), o_3), o_4) = g(g(f_{1:2}, o_3), o_4)\,. \tag{6c}$$

The interesting aspect that is clear from (6c) is that computation can be shared between segments. The feature for sub-segments, like $f_{1:2}$ in (6c), has already been computed in (6b).

To exploit this when features for all segments are required, it is useful to write them in terms of higher-order functions "fold" and "scan" (see section A.1). The feature for the whole sequence can be written as

$$f_{1:4} \triangleq \text{fold}(g, \lambda, [o_1, o_2, o_3, o_4]) \,. \tag{7}$$

The number of applications of the function g is equal to the length of the sequence.

It is possible to avoid duplication of computation by computing features for all segments starting at the same time at once. This can be expressed with the higher-order functional primitive "scan". It performs the same computation as "fold" but returns all intermediate results as a sequence:

$$\text{scan}(g, \lambda, [o_1, o_2, o_3, o_4]) = [\lambda, f_1, f_{1:2}, f_{1:3}, f_{1:4}] \,. \tag{8}$$

The number of applications of g is, just like in (7), equal to the number of elements of the sequence. This means that the average number of applications of g to compute a feature vector for each segment starting at a given time is constant.

To compute features for all segments, the recursion is started at each possible start time:

$$\text{scan}(g, \lambda, [o_1, o_2, o_3, o_4]) = [\lambda, f_1, f_{1:2}, f_{1:3}, f_{1:4}] \,; \tag{9a}$$
$$\text{scan}(g, \lambda, [o_2, o_3, o_4]) = [\lambda, f_2, f_{2:3}, f_{2:4}] \,; \tag{9b}$$
$$\text{scan}(g, \lambda, [o_3, o_4]) = [\lambda, f_3, f_{3:4}] \,; \tag{9c}$$
$$\text{scan}(g, \lambda, [o_4]) = [\lambda, f_4] \,; \tag{9d}$$
$$\text{scan}(g, \lambda, []) = [\lambda] \,. \tag{9e}$$

This computes the features for all $\Theta(T^2)$ segments in $\Theta(T^2)$ time, which is average constant time.

The word score function,

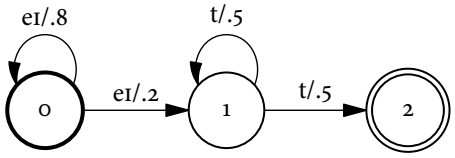
$$h : \mathcal{F} \to \mathcal{S}, \tag{10}$$

takes a feature and returns a score in $\mathcal{S}$ that can be integrated in, for example, a log-linear model. This function is applied separately to each feature in (9).

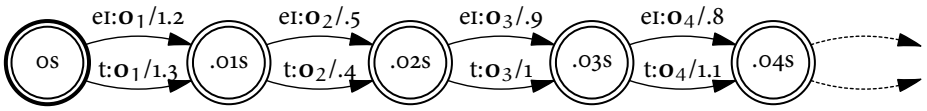
3.1 Instantiation: generative score-spaces

The original instantiation of the algorithm to extract segmental features (Ragni and Gales 2012; van Dalen *et al.* 2013) computes features in a generative score-space. These contain log-likelihoods of the data with respect to the parameters of a generative model (an HMM, in this case) and their derivatives.

First consider the likelihood of an HMM. The following represents HMMs as weighted finite state automata (WFSAs). A more traditional representation is possible, but requires additional matrix multiplications. The FSA representation also draws out the symmetry of the model, which will become important in section 4.1. Composing the weighted finite state automata in figure 1 on the facing page, one producing a weighted symbol sequence, and another converting symbols into observations, produces the trellis in figure 2 on the next page. In this composed automaton, time proceeds from left to right; HMM states are laid out vertically. The weights in the trellis are products of



(a) An automaton that generates sub-word symbols.



(b) An automaton that converts sub-word symbols into observations, weighted by their output weights.

Figure 1 *The two automata that are composed to yield the automaton in figure 2.*

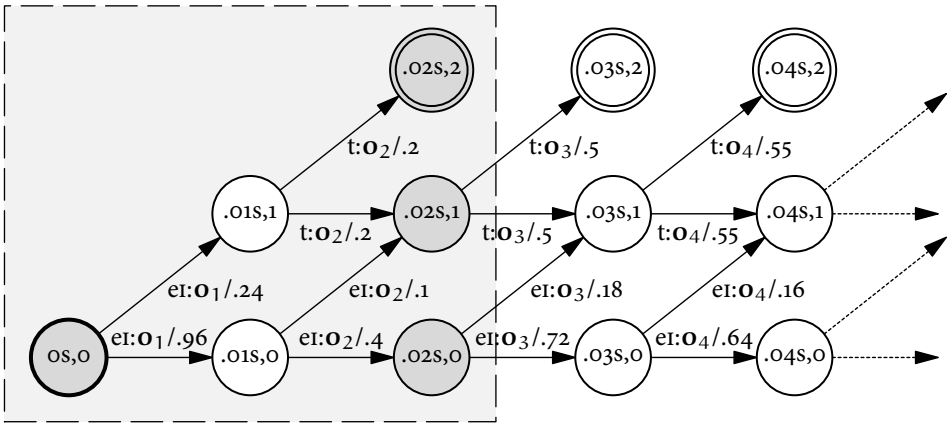


Figure 2 *A trellis expressed as a weighted finite state automaton. The feature vector* $\mathbf{f}_{1:2}$ *contains the total weights from the state "0s,0" to each state at 0,02 s (highlighted).*

transition weights in figure 1a and the output weights in figure 1b. The likelihood for a sequence of observations is the sum over all paths from the highlighted start state "0s,0" to the final state (with double line) corresponding to the last observation in the segment. There is one start state, and many final states, because the start time of the segment is fixed, while the end time is flexible. (In section 4.1, both start and end times will be flexible.) To compute the likelihoods efficiently, the original method applies the forward algorithm. This algorithm takes a vector $\mathbf{f}$ of "forward weights" for all possible states of a finite-state automaton at time $t-1$.

The initial feature $\boldsymbol{\lambda}$, for the empty segment, contains 1 for the start state and 0 for all other states. For the three-state HMM in the example in figure 2,

$$\lambda \triangleq \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}. \tag{11a}$$

Applying the function g on current weights $\mathbf{f}$ and new observation o_t is implemented

as one step of the forward algorithm. One step of the forward algorithm takes the forward weights for time $t-1$ (say, 0.01 s) and computes the weights for time t (say, 0.02 s). The value of o_t is necessary to compute the weights on the transitions for the next step in the automaton. The first evaluation of g generates the forward weights at 0.01 s, and the second evaluation the weights at 0.02 s:

$$\mathbf{f}_1 = g(\lambda, o_1) = \begin{bmatrix} 0.96 \\ 0.24 \\ 0 \end{bmatrix}; \qquad \mathbf{f}_{1:2} = g(\mathbf{f}_1, o_2) = \begin{bmatrix} 0.384 \\ 0.144 \\ .048 \end{bmatrix}. \tag{11b}$$

The second result, $\mathbf{f}_{1:2}$, is indicated in the figure. This feature vector contains the sum of the weights up to each of the highlighted states at time 0.02 s.

The elements of $\mathbf{f}$ can be scalars, as in this example. In that case, the standard forward algorithm for computing the likelihood of a segment is used. This is exploited by Ragni and Gales (2012) to efficiently produce likelihoods for all segments of audio.

In van Dalen *et al.* (2013), the elements of $\mathbf{f}$ are in the expectation semiring. By running the forward algorithm using generalised $+$ and $\times$ operations, not only the likelihoods but also their derivatives are produced.

The word score function h takes a vector $\mathbf{f}$ and extracts the element that indicates the weight in the final state (in the example, the third element of the vector). It then converts it into a score, which is then used in the log-linear model.

4 Monoid features

The previous section has discussed a general class of segmental features which can be efficiently computed if all features for all possible segments are required. Though feature extraction is then feasible for small-vocabulary recognisers, for larger systems additional approximations will be necessary. This could be done by producing a set of hypotheses with a faster recogniser and rescore them using segmental features. In that scenario, segmental features are required for segments with start and end times around those from the hypothesis set. The feature extraction process in section 3 allows flexibility only around the end time of segments, not around start times.[1]

This section will therefore propose a related class of features that are more flexible. Section 4.1 will show how scores in likelihood score-spaces can be computed in this framework. Section 4.2 will generalise this to HMM-like automata with weights in any semiring.

The requirement is made that features for two subsequent segments can be combined into the feature for the joint segment. The process of computing word scores from segments of observations is therefore split up into three functions that act on them consecutively.

First, the function $f(o_t)$ converts the observation into a segmental feature for the segment $[o_t]$. Second, the function $\odot$ combines two of these features into a feature representing the combined segment. Third, the function h computes a score for a word (or phone) given a segmental feature f.

[1] If the backward algorithm is used instead of the forward algorithm, the start time is flexible, but the end time fixed.

The first step is to convert a sequence of observations into a sequence of features. This works by applying the function

$$f : \mathcal{O} \to \mathcal{F}. \tag{12}$$

Applying f to each observation can be formulated with the higher-order function "map" (see appendix A):

$$\text{map}(f,[o_1, o_2, o_3, o_4]) = [f_1, f_2, f_3, f_4]\,, \tag{13}$$

where f_t is a shorthand for $f(o_t)$. f_t is a feature for the segment $[o_t]$, of length 1.

The second, and most interesting, step is to combine two features for consecutive segments with the function

$$\odot : \mathcal{F} \times \mathcal{F} \to \mathcal{F}. \tag{14}$$

The feature for the segment $[o_1, o_2, o_3, o_4]$ can then be computed as

$$f_1 \odot f_2 \odot f_3 \odot f_4. \tag{15}$$

To speed up the computation, it is useful for features for shared sub-segments to be reusable. The features in section 3 are only shared between features for segments with the same start time. For more flexible re-use to be possible, it is useful to allow the computation in (15) to be performed in any order. To see why the freedom of evaluation order is important, consider an example. If a segment is hypothesised to start at o_1 or o_2 and end at o_3 or o_4, and $f_{2:3}$ has been computed, the necessary features can be derived from it efficiently:

$$f_{1:3} = f_1 \odot f_{2:3}; \tag{16a}$$
$$f_{2:4} = f_{2:3} \odot f_4; \tag{16b}$$
$$f_{1:4} = (f_1 \odot f_{2:3}) \odot f_4 = f_{1:3} \odot f_4. \tag{16c}$$

Here, $f_{2:3}$ can be cached and re-used in (16a) and (16b), and $f_{1:3}$ in (16c).

To ensure that the segmental feature is the same whatever order the sub-segments are combined in, $\odot$ must be associative. That is, for any $f, f', f'' \in \mathcal{F}$,

$$(f \odot f') \odot f'' = f \odot (f' \odot f''). \tag{17}$$

This implies that the set $\mathcal{F}$ of features is a *monoid* with $\odot$ as its operation.

In general, a segmental feature can be computed from the features for different sub-segments. One possible order of evaluation goes through the observations consecutively. This is the same order as the one in section 3. It can be written

$$f_{1:4} = \text{fold1}(\odot, [f_1, f_2, f_3, f_4])\,. \tag{18}$$

In section 3, the same order of evaluation was produced with the "fold" function (which requires an extra argument, the initial state). "fold1" computes a "skewed reduction". Since the $\odot$ operation is associative, the "fold1" function can be generalised to a function defined as computing the same value as "fold1", but with the evaluations of $\odot$ in any order. In this report, this function is called "reduce":

$$f_{1:4} = \text{reduce}(\odot, [f_1, f_2, f_3, f_4])\,. \tag{19}$$

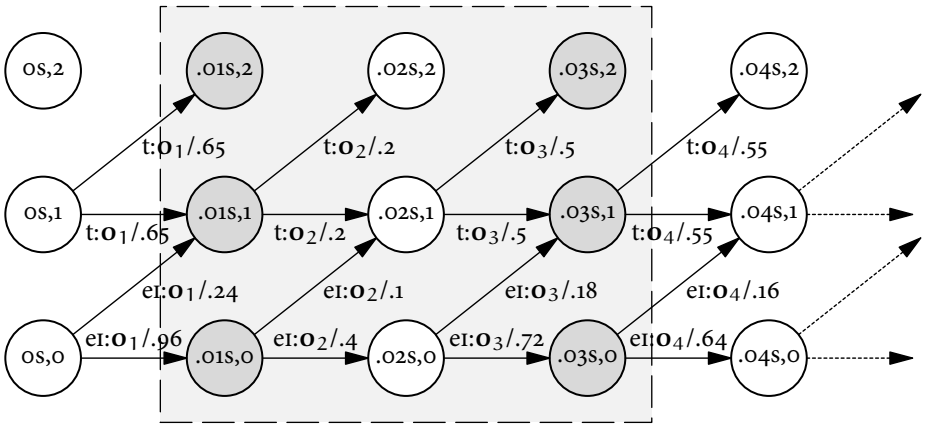


Figure 3 *A trellis like in figure 2, but without a specific start state. The feature* $\mathsf{f}_{2:3}$ *contains total weights between each pair of states at 0.01 s and at 0.03 s (highlighted).*

If features for all segments are required, then expressions analogous to (9) can be used:

$$\mathrm{scan1}(\odot,[\mathsf{f}_1,\mathsf{f}_2,\mathsf{f}_3,\mathsf{f}_4]) = [\mathsf{f}_1,\mathsf{f}_{1:2},\mathsf{f}_{1:3},\mathsf{f}_{1:4}]\,; \tag{20a}$$

$$\mathrm{scan1}(\odot,[\mathsf{f}_2,\mathsf{f}_3,\mathsf{f}_4]) = [\mathsf{f}_2,\mathsf{f}_{2:3},\mathsf{f}_{2:4}]\,; \tag{20b}$$

$$\mathrm{scan1}(\odot,[\mathsf{f}_3,\mathsf{f}_4]) = [\mathsf{f}_3,\mathsf{f}_{3:4}]\,; \tag{20c}$$

$$\mathrm{scan1}(\odot,[\mathsf{f}_4]) = [\mathsf{f}_4]\,. \tag{20d}$$

This still computes the feature values for all $\Theta(\mathsf{T}^2)$ segments in $\Theta(\mathsf{T}^2)$ time, which is constant time on average.

The third step is, as in section 3.1, to apply the word score function:

$$\mathsf{h} : \mathcal{F} \to \mathcal{S}. \tag{21}$$

This function takes a feature and returns a score in $\mathcal{S}$ that can be integrated in, for example, a log-linear model. This function is applied separately to each feature in (20).

4.1 Instance: likelihood score-spaces

Section 3.1 has described a method introduced by Ragni and Gales (2012); van Dalen *et al.* (2013) for computing features in "generative score-spaces" derived from HMMs. This section will sketch how this can be extended to monoid features, and how the observations and the functions f, $\odot$, and h are then defined.

At first, assume that log-likelihood score-spaces are used. This means that the word scores are given by the log-likelihoods of HMMs. The following discussion will again represent HMMs as weighted finite state automata (WFSAs). This draws out the symmetry of the model, which is important since the features must be extensible in two directions. The rows and columns of the matrix correspond to all states in a word (or sub-word) HMM.

Figure 3 contains a trellis similar to the one in figure 2 on page 5. However, this one is not specific to a start time. Each feature f is a square matrix indicating the total

weight going from the state indicated by the row index at one time to the state indicated by the column index at one other time. The entries of matrix $f(o_t)$ correspond to the weights on the transitions between two consecutive times. For example, the feature $f_2 = f(o_2)$ is derived from the weights between 0.01 s and 0.02 s:

$$f(o_2) = \begin{bmatrix} 0.4 & 0.1 & 0 \\ 0 & 0.2 & 0.2 \\ 0 & 0 & 0 \end{bmatrix}. \tag{22a}$$

The way to read this matrix is

		To state 0	1	2
From state	0	0.4	0.1	0
	1	0	0.2	0.2
	2	0	0	0

. (22b)

In this particular example, the feature matrix is for one observation. In general, this matrix gives the weights for combinations of start and end states for a specific segment, starting at the start time and at the end time. The trellis diagram in figure 3 illustrates $f_{2:3}$, which contains weights for each pair of states at 0.01 s and states at 0.03 s. To derive the total weight between each of these pairs from f_2 (in (22a)) and f_3, the connecting states, at 0.02 s, must be summed out. Expressed in terms of the elements of the matrices,

$$(f_2 \odot f_3)_{ij} = \sum_k f_{2,ik} \cdot f_{3,kj}. \tag{23a}$$

In general, the monoid operation on two features is therefore a matrix multiplication:

$$f \odot f' \triangleq f \cdot f'. \tag{23b}$$

Matrix multiplication is associative, so this adheres to the requirement for being a monoid.

To compute the word score for any feature f, the element of the matrix in (22a) representing the start and end state, in the example at position $(0, 2)$, must be selected. In general, word w will have a vector of WFSA start weights $\boldsymbol{\lambda}$ and a vector of end weights $\boldsymbol{\rho}$. For the WFSA in figure 1a on page 5, they are

$$\boldsymbol{\lambda} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}; \qquad \boldsymbol{\rho} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}. \tag{24}$$

The word score can then be computed as

$$h(f) \triangleq \boldsymbol{\lambda}^\mathsf{T} \cdot f \cdot \boldsymbol{\rho}. \tag{25}$$

4.2 Features from finite-state models

So far, the entries of the matrices have been assumed to be just likelihoods. However, it is possible to accumulate other types of features. One example was shown in section 3.1,

where the score required not only likelihoods, but also their derivatives (van Dalen *et al.* 2013). Accumulating these worked with the forward algorithm, but the weights on the finite state automata were in the expectation semiring, instead of being just scalars. The expectation semiring, as does every other semiring, replaces normal addition and multiplication by operations $\oplus$ and $\otimes$.

The monoid features discussed so far have been a matrix with scalar entries derived from an HMM. Now the entries can be generalised to be in the expectation semiring. The monoid operation between two features then is a generalised matrix multiplication, where the entries of the matrix are not just scalars as in (23a), but in a semiring:

$$(f \odot f')_{ij} = \bigoplus_k f_{ik} \otimes f'_{kj}. \tag{26}$$

The function $f(\cdot)$, which computes a feature from one observation, produces a matrix with entries in the expectation semiring. For generative score-spaces, the first element of each of the values in the semiring is the same likelihood, and the second element the partial derivatives with respect to the parameters of the generative model. However, other types of entries are also possible.

It is well-known, and shown in Appendix B, that square matrices form a monoid under matrix multiplication iff its elements are in a semiring. Therefore, any finite-state model with the same shape as an HMM — one state model generating symbols, and one linear transducer from symbols to the observations — with weights in any semiring can be used to extract features in a monoid.

This opens up the possibility for many types of segmental features. There is no requirement for the semiring to contain a likelihood or anything similar. Of particular interest are features based on neural networks. Features that, just the likelihood of HMMs, are additive over paths of the finite state automaton can then be extracted in average constant time if they are computed for all possible segments of audio. Additionally, monoid features for consecutive segments can be combined straightforwardly, which allows flexibility when pruning is used during decoding.

5 Conclusion

This report has discussed a general class of segmental features for speech recognition that are efficient to compute. The new features are as efficient as features used in Ragni and Gales (2012); van Dalen *et al.* (2013) when features for all possible segments are required. However, they are more flexible in re-using features for sub-segments. This is done by requiring that the features are in a monoid. Because monoids are associative, features for two consecutive segments can be combined to form a feature for the union of the segments. This will allow more flexibility in integrating this type of feature in speech recognisers that perform pruning to obtain good performance. A type of features that is of particular interest for future work can be derived from HMM-like finite-state automata with weights in any semiring.

A Higher-order functions

Higher-order functions are functions that take other functions as parameters. In calculus, higher-order functions are therefore known as "functionals". They are important primitives for functional programming languages (see, e.g., Bird and Wadler 1988). The following discussion will use generally-used names for the primitives.

An important primitive is "map", which applies a function f to each elements of a sequence $e = (e_1, e_2, \ldots)$ separately:

$$\mathrm{map}(f, (e_1, e_2, \ldots)) \triangleq (f(e_1), f(e_2), \ldots). \tag{27}$$

A.1 Fold

Another primitive is the basis for iteration in functional programming. It is often called "fold". It applies a function recursively:

$$\mathrm{fold}(f, s, [e_1, e_2, \ldots]) \triangleq f(\ldots f(f(s, e_1), e_2), \ldots). \tag{28}$$

In general, the two parameters of f can have different types. For example, if $f(s, e) \triangleq s + 1$, then $\mathrm{fold}(f, 0, [\ldots])$ will count the number of elements of the list. Whatever the list's element type, the first parameter to f will always be an integer. The number of applications of f is equal to the length of the sequence.

A related function is "scan", which essentially applies function f in the same way as fold but returns intermediate results in the form of a list:

$$\mathrm{scan}(f, s, [e_1, e_2, \ldots]) \triangleq [s, f(s, e_1), f(f(s, e_1), e_2), \ldots]. \tag{29}$$

Since each element in the resulting list is the result of applying function f to the previously computed element and the next element in the source list, the number of applications of f is equal to the length of the sequence.

A.2 Reductions on monoids

A reduction is a function that collapses a list (or a more general structure) to a value of the same type as the list elements (e.g., Hinze and Paterson 2006). Folds are more general than reductions, since the resulting type of a fold is not necessarily the same type as the list elements. Thus, a specific type of reduction, a *skewed reduction*, can be computed with the function "fold1":

$$\mathrm{fold1}(f, [e_1, e_2, e_3, \ldots]) \triangleq \mathrm{fold}(f, e_1, [e_2, e_3, \ldots]) = f(\ldots f(f(e_1, e_2), e_3), \ldots). \tag{30}$$

For example, if the function "max(x, y)" returns the greatest value of its parameters, then $\mathrm{fold1}(\max, [\ldots])$ returns the greatest value of the list. In this case, it does not make a difference in which order the values of the list are combined. This is the case if the elements is a monoid, so that the operation on elements is associative:

$$f(f(e_1, e_2), e_3) = f(e_1, f(e_2, e_3)). \tag{31}$$

This essentially implies that the elements are in a monoid.

This work denotes with "reduce" a reduction on a list with elements in a monoid. A straightforward implementation is to use "fold1". Alternatively, it can exploit the associativity, in (31), by re-ordering the function applications. For example, the reduction can be computed in two halves:

$$\begin{aligned}\mathrm{reduce}(f, (e_1, e_2, e_3, e_4)) &\triangleq \mathrm{fold1}(f, [e_1, e_2, e_3, e_4]) \\ &= f(f(f(e_1, e_2), e_3), e_4) \\ &= f(f(e_1, e_2), f(e_3, e_4)). \end{aligned} \tag{32}$$

When reductions of overlapping segments of a list are required, this freedom to reorder can often be exploited more easily than using folds.

B Matrix elements in a semiring

Section 4.2 generalises the type of elements in matrices that are used as features from scalars. It is therefore important to check what requirements on the elements of the matrices are to ensure that the features are in a monoid. Substituting (26) into (31), the requirement of associativity, for three matrices f, f', f'',

$$((f \odot f') \odot f'')_{ij} = (f \odot (f' \odot f''))_{ij}. \tag{33a}$$

Expanding the left- and right-hand sides separately,

$$((f \odot f') \odot f'')_{ij} = \bigoplus_l (f \odot f')_{il} \otimes f''_{lj} = \bigoplus_l \left(\bigoplus_k f_{ik} \otimes f'_{kl} \right) \otimes f''_{lj}; \tag{33b}$$

$$(f \odot (f' \odot f''))_{ij} = \bigoplus_k f_{ik} \otimes (f' \odot f'')_{kj} = \bigoplus_k f_{ik} \otimes \left(\bigoplus_l f'_{kl} \otimes f''_{lj} \right). \tag{33c}$$

These two expressions are equal if for the elements of f, f', and f'', $\oplus$ distributes over $\otimes$ (for (34a) and (34d)), $\oplus$ is commutative (for (34b)), and $\otimes$ is associative (for (34c)):

$$\bigoplus_k f_{ik} \otimes \left(\bigoplus_l f'_{kl} \otimes f''_{lj} \right) = \bigoplus_k \bigoplus_l f_{ik} \otimes \left(f'_{kl} \otimes f''_{lj} \right) \tag{34a}$$

$$= \bigoplus_l \bigoplus_k f_{ik} \otimes \left(f'_{kl} \otimes f''_{lj} \right) \tag{34b}$$

$$= \bigoplus_l \bigoplus_k \left(f_{ik} \otimes f'_{kl} \right) \otimes f''_{lj} \tag{34c}$$

$$= \bigoplus_l \left(\bigoplus_k f_{ik} \otimes f'_{kl} \right) \otimes f''_{lj}. \tag{34d}$$

Monoids are normally taken to have an identity element as well. The identity for multiplication can be defined as a matrix with as diagonal entries the multiplicative identity ("1"), and as the other entries the additive identity ("0"), which must annihilate elements under multiplication.

These requirements on the elements are the ones required for semirings. Square matrices with elements in a semiring are therefore in a monoid under generalised matrix multiplication.

Bibliography

Richard Bird and Philip Wadler (1988). *An introduction to functional programming*. Prentice Hall, Hertfordshire, UK.

M. J. F. Gales and F. Flego (2010). "Discriminative classifiers with adaptive kernels for noise robust speech recognition." *Computer Speech and Language* 24 (4), pp. 648–662.

M. J. F. Gales, S. Watanabe, and E. Fosler-Lussier (2012). "Structured Discriminative Models For Speech Recognition: An Overview." *IEEE Signal Processing Magazine* 29 (6), pp. 70–81.

Ralf Hinze and Ross Paterson (2006). "Finger trees: a simple general-purpose data structure." *Journal of Functional Programming* 16 (2), pp. 197–217.

Martin Layton (2006). *Augmented Statistical Models for Classifying Sequence Data*. Ph.D. thesis, Cambridge University.

A. Ragni and M. J. F. Gales (2012). "Inference Algorithms for Generative Score-Spaces." In *Proceedings of ICASSP*. pp. 4149–4152.

R. C. van Dalen, A. Ragni, and M. J. F. Gales (2013). "Efficient Decoding with Generative Score-Spaces Using the Expectation Semiring." In *Proceedings of ICASSP*.

Geoffrey Zweig and Patrick Nguyen (2009). "A Segmental CRF Approach to Large Vocabulary Continuous Speech Recognition." In *Proceedings of ASRU*.