A **S**egmental **C**onditional **R**andom **F**ield Speech Recognition Toolkit SCARF v1.2.1

Geoffrey Zweig and Patrick Nguyen
`gzweig@microsoft.com`

May 6, 2011

Contents

Foreword

This document covers the SCARF v1.2.1 release. The primary change from the SCARF v1.1 release is a simpler command line parsing module, which is transparent to the user. Internally, key SCARF computations are now done with doubles rather than floats, improving numerical stability. Lastly, external feature lexicalization was modified to include a non-lexicalized feature along with the lexicalized variants. This reduces sensitivity to regularization and allows similar performance with a smaller L1 norm.

SCARF v1.2.1 differs from v1.2 only in enhanced documentation: An appendix has been added describing the use of Broadcast News Lattices available through the Linguistic Data Consortium, catalog number LDC2011T06. These lattices provide a state-of-the-art baseline that can be used with SCARF.

Chapter 1

Approach

1.1 Overview

SCARF is a toolkit for using Segmental Conditional Random Fields (SCRFs) to do speech recognition. It applies a discriminative model to segment-level features which are based on acoustic detector inputs. The result is a segmentation of the data into chunks, and the assignment of a word label to each chunk. Segment based models are computationally expensive, and SCARF makes the search efficient through the use of lattice constraints. These may come from an external source such as HTK, or they may be generated with the built-in SCARF fast match.

The inspiration for SCARF comes from Maximum Entropy (ME) models, in which one may use thousands of possibly redundant features in a model to do classification. However, whereas ME models are best suited to "flat" n-way classification tasks, SCRFs are naturally suited to sequence labeling problems in which a sequence of labels (words) is assigned to an arbitrarily long input sequence. In this respect, SCRFs draw from earlier Conditional Random Field (CRF) models, which were designed for sequence labeling. SCRFs extend these models by operating at the segment level, in which multiple adjacent observations can be lumped together into a segment with a single label, and segment-level features can be extracted and used. In essence, SCRFs can be thought of as combining the sequence labeling properties of CRFs with the segment labeling properties of ME models. These properties are illustrated in Table 1.1.

The use of SCRFs in speech recognition has several potential advantages:

- As with Maximum Entropy models, they offer a convenient way to combine numerous, possibly redundant features. Unlike feature vectors as used in HMMs, we do not need to worry about keeping the features uncorrelated.
- Since the analysis is done at the segment level, long-span features such as pitch contours can be extracted and related directly to the word hypothesis for a segment.
- The models are inherently discriminative in nature. Unlike HMM models, in which discriminative training methods such as MMI, MPE and MCE are applied in a separate "add-on" process, discriminative training is built into SCRFs.

In addition to the general advantages of SCRFs mentioned above, the SCARF implementation in particular has several important points which are worth mentioning:

- N-gram language modeling has been fully incorporated, and one can easily choose whether to use a pre-trained maximum likelihood model, or to learn its parameters discriminatively, in an integrated fashion with the acoustic model parameters.
- SCARF takes as its basic input acoustic detector events (see, e.g., [1, 2]. These may be, for example, phoneme detections or syllable detections. A wide variety of features can be automatically generated from these basic inputs.
- SCARF has been designed to facilitate some of the operations that are commonly done with speech recognizers based on generative models. For example, the language model and lexicon can be changed without retraining.

	Generative Model	Discriminative Model
Framewise Analysis	HMM	CRF
Segmental Analysis	Segmental HMM	SCRF

Table 1.1: Classification of model types along two dimensions.

- SCARF supports user-defined features in the form of lattice annotations. This makes is simple to test the effect of new features without modifying any code.

1.2 Research Directions

The SCARF toolkit is intended to facilitate research into the use of SCRFs in speech recognition. There are many avenues to be explored, and we list several below.

- SCARF has the ability to work with many detector streams, and there is no requirement that they use the same signal processing or be non-redundant. Does it make sense to apply different signal processing depending on the type of units being detected?
- One way of making detectors - phone detectors, say - is to build a conventional speech recognizer and treat each phoneme as a word. In the segmental context, can we develop better detection methods?
- What segmental features can be gainfully incorporated into a system? Pitch, contour or other trajectory features? Count features such as the number of energy peaks in a segment? Template matching distances or discriminative templates?
- What are the best features to use when one is jointly training both an acoustic and language model? What is the best way of using this ability?
- Speaker adaptation in the context of SCRFs.

Chapter 2

Model

2.1 Segmental Framework

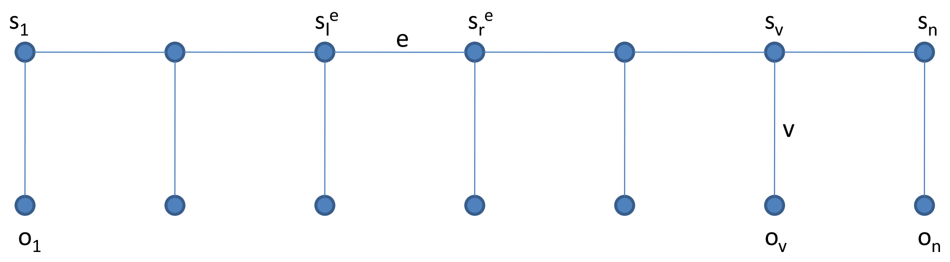

Figure 2.1: Graphical representation of a CRF.

Segmental Conditional Random Fields - also known as Semi-Markov Random Fields [3] or SCRFs - form the theoretical underpinning for SCARF. They relax the Markov assumption from the frame level to the word level, where states now correspond with a variable and automatically derived time span. To explain these, we begin with the standard Conditional Random Field model [4], as illustrated in Figure 2.1. Associated with each vertical edge v are one or more feature functions $f_k(s_v, o_v)$ relating the state variable to the associated observation. Associated with each horizontal edge e are one or more feature functions $g_d(s_l^e, s_r^e)$ defined on adjacent left and right states. (We use s_l^e and s_r^e to denote the left and right states associated with an edge e.) The set of functions (indexed by k and d) is fixed across segments. A set of trainable parameters λ_k and ρ_d are also present in the model. The conditional probability of the state sequence $\mathbf{s}$ given the observations $\mathbf{o}$ is given by

$$P(\mathbf{s}|\mathbf{o}) = \frac{\exp(\sum_{v,k} \lambda_k f_k(s_v, o_v) + \sum_{e,d} \rho_d g_d(s_l^e, s_r^e))}{\sum_{\mathbf{s}'} \exp(\sum_{v,k} \lambda_k f_k(s'_v, o_v) + \sum_{e,d} \rho_d g_d(s_l'^e, s_r'^e))}$$

In speech recognition applications, the labels of interest, words, span multiple observation vectors, and the exact labeling of each observation is unknown. Hidden CRFs [5] address this issue by summing over all labelings consistent with a known or hypothesized word sequence. However, in the recursions presented in [5], the Markov property is applied at the frame level, with the result that segmental properties are not modeled. The C-Aug model [6, 7] is also related, in applying a conditonal model at the segmental level, with a particular set of features derived from the Fisher kernel.

Here, in order to user long-span features, and to directly relate segment-level acoustic properties to the word label, we adopt the formalism of segmental CRFs. In contrast to a CRF, the structure of the model is not fixed *a priori*. Instead, with N observations, all possible state chains of length $l \leq N$ are considered, with the observations segmented into l chunks in all possible ways. Figure 2.2 illustrates this. The top part of this figure shows seven observations broken into three segments, while the bottom part shows the same observations partitioned into two segments. For a given segmentation, feature functions are defined as with standard CRFs. Because of the segmental

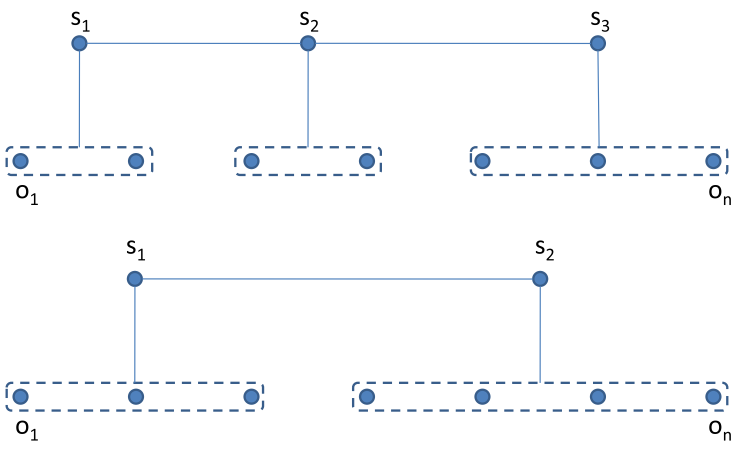

Figure 2.2: A Segmental CRF and two different segmentations.

nature of the model, transitions only occur at logical points, and it is clear what span of observations to use to model a given symbol.

Since the g functions already involve pairs of states, it is no more computationally expensive to expand the f functions to include pairs of states as well, as illustrated in Figure 2.3. This structure has the further benefit of allowing us to drop the distinction between g and f functions. To denote a block of original observations, we will use o_i^j to refer to observations i through j inclusive.

In the semi-CRF work of [3], the segmentation of the training data is known. However, in speech recognition applications, this is not the case. Therefore, in computing sequence likelihood, we must consider all segmentations consistent with the state (word) sequence $\mathbf{s}$, i.e. for which the number of segments equals the length of the state sequence. Denote by $\mathbf{q}$ a segmentation of the observation sequences, for example that of Fig. 2.3 where $|\mathbf{q}| = 3$. The segmentation induces a set of (horizontal) edges between the states, referred to below as $e \in \mathbf{q}$. One such edge is labeled e in Fig. 2.3. Further, for any given edge e, let $o(e)$ be the segment associated with the right-hand state s_r^e, as illustrated in Fig. 2.3. The segment $o(e)$ will span a block of observations from some start time to some end time, o_{st}^{et}; in Fig, 2.3, $o(e)$ is identical to the block o_3^4. (The first block of observations is handled by an implicit transition from a special start state to the first word.) With this notation, we represent all functions as $f_k(s_l^e, s_r^e, o(e))$ where $o(e)$ are the observations associated with the segment of the right-hand state of the edge. The conditional probability of a state (word) sequence $\mathbf{s}$ given an observation sequence $\mathbf{o}$ for a SCRF is then given by

$$P(\mathbf{s}|\mathbf{o}) = \frac{\sum_{\mathbf{q} \text{ s.t. } |\mathbf{q}|=|\mathbf{s}|} \exp(\sum_{e\in\mathbf{q},k} \lambda_k f_k(s_l^e, s_r^e, o(e)))}{\sum_{\mathbf{s}'} \sum_{\mathbf{q} \text{ s.t. } |\mathbf{q}|=|\mathbf{s}'|} \exp(\sum_{e\in\mathbf{q},k} \lambda_k f_k(s_l'^e, s_r'^e, o(e)))}.$$

Training is done by gradient descent using Rprop [8]. Taking the derivative of $\mathcal{L} = \log P(\mathbf{s}|\mathbf{o})$ with respect to λ_k we obtain the necessary gradient:

$$\frac{\partial \mathcal{L}}{\partial \lambda_k} = \frac{\sum_{\mathbf{q} \text{ s.t. } |\mathbf{q}|=|\mathbf{s}|} T_k(\mathbf{q}) \exp(\sum_{e\in\mathbf{q},k} \lambda_k f_k(s_l^e, s_r^e, o(e)))}{\sum_{\mathbf{q} \text{ s.t. } |\mathbf{q}|=|\mathbf{s}|} \exp(\sum_{e\in\mathbf{q},k} \lambda_k f_k(s_l^e, s_r^e, o(e)))} - \frac{\sum_{\mathbf{s}'} \sum_{\mathbf{q} \text{ s.t. } |\mathbf{q}|=|\mathbf{sl}'|} T_k'(\mathbf{q}) \exp(\sum_{e\in\mathbf{q},k} \lambda_k f_k(s_l'^e, s_r'^e, o(e)))}{\sum_{\mathbf{s}'} \sum_{\mathbf{q} \text{ s.t. } |\mathbf{q}|=|\mathbf{s}'|} \exp(\sum_{e\in\mathbf{q},k} \lambda_k f_k(s_l'^e, s_r'^e, o(e)))},$$

with

$$T_k(\mathbf{q}) = \sum_{e\in\mathbf{q}} f_k(s_l^e, s_r^e, o(e))$$

$$T_k'(\mathbf{q}) = \sum_{e\in\mathbf{q}} f_k(s_l'^e, s_r'^e, o(e)).$$

This derivative can be computed efficiently with dynamic programming and a 1st pass state space reduction, using the recursions described in [9]. In practice, we add L1 and L2 regularization terms to $\mathcal{L}$ to obtain an regularized objective function.

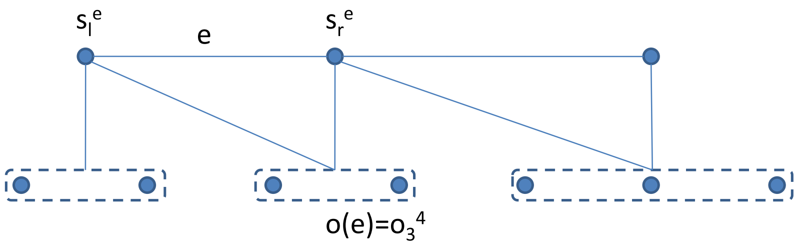

Figure 2.3: Incorporating last-state information in a SCRF.

2.2 Adapting SCRFs for Speech Recognition

In order to model continuous speech, the model structure of Figure 2.3 is given a specific meaning. While the features we use relate a word to an observation span, the state does not directly encode a word identity. Instead, the values of the state variable in this model correspond to states in a finite state representation of a n-gram language model. This is illustrated in Figure 2.4. In this figure, a fragment of a finite state language model representation is shown on the left. The states are numbered, and the words next to the states specify the linguistic state. At the right of this figure is a fragment of a CRF illustrating the word sequence "the dog nipped." The states are labeled with the index of the underlying language model state. In our search strategy [9], we extend existing hypotheses with specific words, so the word identity is always available for feature computation.

We use the language model in two ways. First, conventional smoothed ngram probabilities can be returned as transition features. A single λ is trained to weight these features, resulting in a single discriminatively trained language model weight. Secondly, indicator features can be introduced, one for each arc in the language model, which indicate when an arc is traversed in the transition from one state to another. A state transition in the CRF then results in a non-zero feature value (i.e. 1) for each arc traversed in the underlying language model structure. For example, in Figure 2.4, the arcs $(1, 2)$ and $(2, 6)$ are traversed in moving from state 1 to state 6. Each of these arcs has its own binary feature. Learning the weights on these results in a discriminatively trained language model, trained jointly with the acoustic model.

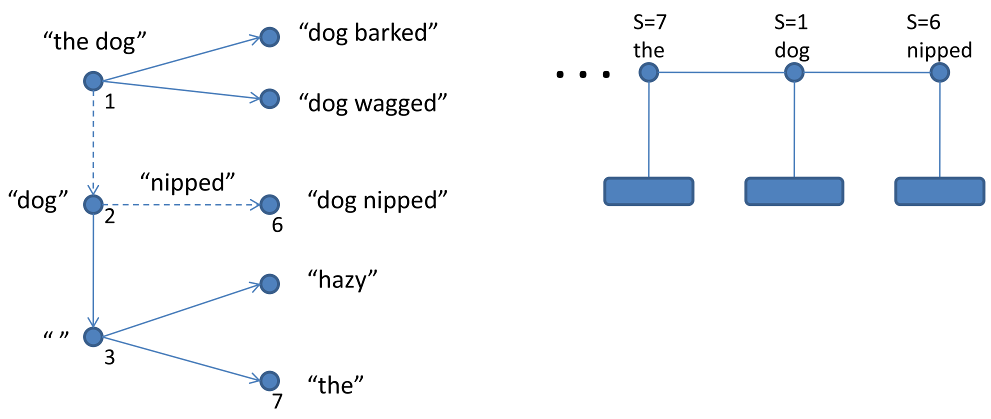

Figure 2.4: Correspondence between language model state and SCRF state. The dotted lines indicate the path taken in hypothesizing "nipped" after "the dog." A line from state 7 to state 1 has been omitted for clarity.

SCARF has also been designed to support two other common operations in speech recognition. First, it can be trained with one language model, and then at decode time a different model can be substituted. (This only applies when a single LM weight is learned.) The learned weight will be used in association with the new language model, and further, the user can manually "clamp" the weight to any desitred value. Secondly, as we will see in the next chapter, two classes of features (Expectation and Levenshtein) have been designed so that at test time a new dictionary can be used. The subword-unit weights generalize to any new words.

2.3 Related Work

In addition to the work already discussed, useful background information can be found in a number of other papers. In [10], the authors propose a speech recognition method based on the sequential estimation of state probabilities via the application of a Maximum Entropy model. This model operates at the frame level and uses gaussian ranks as features. In [11, 12], CRFs are successfully applied to the speech task, using features based on the probabilities of phonological events. These real-valued features are again computed at the frame level. Early work by Ratnaparkhi [13] in NLP provides further background on MaxEnt based approaches.

Chapter 3

Features

3.1 Nomenclature

SCARF's built-in acoustic features are defined in terms of the detection units that a word spans. Suppose we have a word with hypothesized start time st and end time et. For a specific detector unit u, the notation $u \in span(st, et)$ is used to indicate that the detection exists within the time boundaries from st to et, inclusive. $pron(w)$ is used to represent the pronunciation of word w. Handling of multiple pronunciations is specific to the feature type, and described below.

3.2 Detector Inputs

The inputs to the feature creation process consist of streams of detector events, and optionally dictionaries that specify the detection sequences that are expected for the words. Each atomic detector stream provides a sequence of detector events, which consist of a unit which is detected and a time at which the detection occurs. Each stream defines its own unique unit set, and these are not shared across streams.

A dictionary providing canonical word pronunciations can be provided for each feature stream. For example, phonetic and syllabic dictionaries could be provided. As discussed below, the existence of a dictionary enables the automatic construction of certain features that indicate (in)consistency between a sequence of detected units and those expected given a word hypothesis. These allow for generalization to words not seen in the training data.

3.3 Existence Features

Recall that a language model state s implies the identity of the last word that was decoded: $w(s)$. Existence features simply indicate whether a detector unit exists in a word's span. They are of the form:

$$f_u(s, s', o_{st}^{et}) = \delta(w(s') = u)\delta(u \in span(st, et)).$$

Dictionary pronunciations are not used in these features; however, no generalization is possible across words. Higher order existence features, defined on the existence of *ngrams* of detector units, can also be automatically constructed. Since the total number of existence features is the number of words times the number of unit ngrams, we must constrain the creation of such features in some way. Therefore, we create an existence feature in two circumstances only:

- when a word and ngram of units exists together in a dictionary
- when a word exists in a training sentence, and an ngram exists within the word's span; this correspondence is determined examining the constraint files provided for training (Sec 5.5)

3.4 Expectation Features

Expectation features represent one of three events: the correct-accept, false-reject, or false accept of an ngram of units within a word's span. In order, these are of the form:

$$f_u^{ca}(s,s',o_{st}^{et}) = \delta(u \in pron(w(s'))\delta(u \in span(st,et))$$

$$f_u^{fr}(s,s',o_{st}^{et}) = \delta(u \in pron(w(s'))\delta(u \notin span(st,et))$$
$$f_u^{fa}(s,s',o_{st}^{et}) = \delta(u \notin pron(w(s'))\delta(u \in span(st,et))$$

Expectation features are indicators of consistency between the units expected given a word ($pron(w)$), and those that are actually in the seen observation span. There is one feature of each type for each unit, and they are *independent* of word identity. Therefore these features provide important generalization ability. Even if a particular word is not seen in the training data, or if a new word is added to the dictionary, they are still well defined, and the λs previously learned can still be used. To measure higher-order levels of consistency, bigrams and trigrams of the atomic detector units can also be automatically generated.

The case where a word has multiple pronunciations requires special attention. In this case,

- A correct accept is triggered if *any* pronunciation contains an observed unit sequence.
- A false accept is triggered if *no* pronunciation contains an observed unit sequence.
- A false reject is triggered if all pronunciations contain a unit sequence, and it is not present in the detector stream.

Unit ngram features are again restricted to ngrams occurring in the training data.

3.5 Levenshtein Features

Levenshtein features are the strongest way of measuring the consistency between expected and observed detections. To construct these, we compute the edit distance between the units present in a segment and the units in the pronunciation(s) of a word. We then create the following features:

$$f_u^{match} = \text{number of times } u \text{ is matched}$$

$$f_u^{sub} = \text{ number of times } u \text{ (in pronunciation) is substituted}$$
$$f_u^{del} = \text{ number of times } u \text{ is deleted}$$
$$f_u^{ins} = \text{ number of times } u \text{ is inserted}$$

In the context of Levenshtein features, the use of expanded ngram units does not make sense and is not used. Like the expectation features, Levenshtein features provide a powerful generalization ability as they are well-defined for words that have not been seen in training.

When multiple pronunciations of a given word are present, the one with the smallest edit distance is selected for the Levenshtein features.

3.6 Language Model Features

SCARF supports two kinds of language model features: global and local; these are now described in turn.

3.6.1 Global LM Features

There are two global language model features:

1. The language model score of a word, as determined by an n-gram language model. Recall that the states in the SCRF correspond to language model history states, so the necessary information is readily available.
2. A feature that indicates whether the current word is out-of-vocabulary with respect to the language model. The language model score will be that of `<unk>` in the current context, and this additional feature provides a simple way of further training the LM.

The global language model features are especially useful because they allow for language model and vocabulary swapping after training. These weights can be learned in the context of one language model, and then used with another.

3.6.2 Local LM Features

SCARF provides local language model features in order to discriminatively train the language model itself. As mentioned in Section 2.2, when SCARF computes a language model probability, it keeps track of which arcs are traversed in an underlying finite-state representation of the language model. There is a binary feature for each arc, indicating whether it is traversed or not. Learning weights on these features results in a discriminatively trained language model. Further, the resulting SCARF model has *jointly* trained acoustic and language models.

It is important to note that when local language model features are used, the same language model should be used for training and decoding.

3.7 Baseline Features

In order to leverage the existence of high-quality baseline HMM systems, we have also added a baseline feature. This is essentially a detector stream that specifies the 1-best word output of a baseline system. The time associated with each word in this stream is its midpoint. Denote the number of baseline detections in a timespan from st to et by $C(st, et)$. In the case that there is just one, let its value be denoted by $B(st, et)$. The baseline feature is defined as:

$$f_b(s, s', o_{st}^{et}) = \begin{cases} +1 & \text{if } C(st, et) = 1 \text{ and } B(st, et) = w(s') \\ -1 & \text{otherwise.} \end{cases}$$

That is, the baseline feature is 1 when a segment spans just one baseline word, and the label of the segment matches the baseline word. It can be seen that the contribution of the baseline features to a path score will be maximized when the number of segments is equal to the number of baseline words, and the labeling of the segments is identical to the baseline labeling. Thus, as we can assign a weight approaching infinity to the baseline feature, baseline performance can be guaranteed. In practice, of course, the baseline weighting is learned and its value will depend on the relative power of the additional features.

3.8 External Features

Recall from Section 1 that the SCARF computations are guided by a set of lattice constraints. In the case that a user wishes to experiment with a new type of feature, this can easily be done by annotating the lattice links with the feature values. The precise syntax for this is defined in Section 5.5.3, however one example would be line

```
24 100 ball duration=0.123,zc=1.2
```

indicating that the word "ball" in the lattice between times 24 and 100 has duration feature score 0.123 and zero-crossing score 1.2. Externally defined features are useful for testing new features without modifying any SCARF code.

For further flexibility, external features can be *lexicalized.* In lexicalization, instead of creating just the single named feature, SCARF additionally creates a separate version of that feature for each word. For example, to learn a word insertion penalty, one might add an external feature `wip` and always set its value to 1.0. By lexicalizing this feature, SCARF will learn a distinct penalty/reward for each word. This would be the same as a discriminatively trained unigram model, except that the weights would be learned in the context of any other language model being used, and any other features. Note that the lexical features are added in addition to the original non-lexicalized variant. This allows any common value to be “factored out” thus reducing the L1 norm of the resulting model.

Chapter 4

Fast Match

To speed up computations with SCARF, the set of segmentations which is considered by the training and decoding recursions is restricted to those known a-priori to have relatively high likelihood [9]. This set of segmentations is represented by a lattice, or constraint file, which at a minimum has a set of possible words along with their start and end times. (The format of these files is defined in Section 5.5.) One way of obtaining a reasonable lattice is to train a standard HMM system such as HTK, and then use the lattices generated by a decoding with that system. As an alternative, SCARF provides a built in fast match which operates directly on the events in a detection file.

The native SCARF Fast Match is based on a vector-space model using term-frequency / inverse-document-frequency (TF-IDF) weighting. Essentially, the fast match is identical to the SCARF decoding procedure itself [9] with the following modifications:

- only one acoustic feature is used - the TF-IDF similarity between the events detected in a segment and those expected based on the hypothesized word
- *all* segmentations consisting of segments which are less than a maximum length (e.g. 20 for phonemes) are considered
- the (single) acoustic weight is hand-tuned rather than trained

The following sections are abstracted from [14], and describe the fast match in more detail.

4.1 TF-IDF Definition

A vector space model measures the similarity between two data objects as represented by vectors of terms. In our case, the data objects are segments of speech as represented by sequences of subword units, and the terms are n-grams of subword units. For example, if we use bigrams of phonemes, each segment will be represented by a vector of length k^2 where k is the number of phonemes in the dictionary. Each position j in this vector contains the TF-IDF weight of a term t_j.

This weight consists of two parts: the IDF part, which is computed with respect to some training data, and the TF part, which is a function of just the one segment which is being labeled. For speech recognition, the training data consists of one count vector C_i for each word. Position c_j in this vector contains the number of times t_j has been seen in association with word i (e.g. in forced alignments of audio data). To compute the IDF score for t_j, we count the number of words in the training data that contain t_j. Call this number d_j, and the total number of words (i.e. the vocabulary size) W. The IDF score of t_j is given by $\log(W/d_j)$. Intuitively, a term is more useful when it occurs in a small number of words. The TF part is simply the relative frequency of t_j in the segment being analyzed. If a segment has N terms and n_j of these are occurrences of t_j, the TF score is n_j/N. Altogether, the TF-IDF score of term t_j in a segment is:

$$\frac{n_j}{N} \log \frac{W}{d_j}$$

. In a "training" process, an index is created that has a TF-IDF vector for every word in the dictionary. The weights can be determined wither from term occurrences in the dictionary pronunciations themselves, or, more generally, from term occurrences in some empirically derived realizations of a word.

blake	b l ey k	11	p l ey k
blake	b l ey k	10	b l ey k
blake	b l ey k	3	l ey k
blake	b l ey k	2	t ax l ey k
blake	b l ey k	2	p ax l ey k
blake	b l ey k	1	p l ey

Figure 4.1: Realizations of the word “blake.”

Given the vector representation of two objects, in our case words, one widely used method for defining similarity is as the cosine of the angle between the vectors. If we have vectors $\mathbf{v}_a$ and $\mathbf{v}_b$, their similarity is defined as

$$S(a,b) = \frac{\mathbf{v}_a \cdot \mathbf{v}_b}{|\mathbf{v}_a||\mathbf{v}_b|}$$

This results in a quantity which varies between 0 and 1, similar to probability, and the basis for the acoustic score will be the logarithm of this quantity, analogous to log-probability. We will refer to this as the *log-TF-IDF* score. We note that TF-IDF and cosine distance are heuristic rather than statistical in nature; nevertheless, their use in information retrieval has been very successful.

4.2 Indexing Units

Typically, the fast match indexes are based on a phonemic representation of words. However, as with [15], we find that representations based on plain phonetic decoding are inferior to those based on multi-phone decoding [16]. A phonemic representation of the data may be obtained simply by breaking apart the subword representation. N-grams of units are then used as terms; bigrams or trigrams of phonemes have proved effective.

4.3 Creating an Inverted Index

In the decoding process, it will be necessary to quickly determine the set of words that are a reasonable match to the phonemes (or units generally) in a particular segment of speech. To do this, we follow a two-step process:

1. First, SCARF creates a single vector representation for every word in the dictionary.
2. Second, SCARF creates an inverted index that indicates, for a particular term, all the words which contain it.

These steps are described in turn.

The SCARF TF-IDF index is typically based on an unconstrained decoding of the training data in terms of subword units. To do this, one creates a trigram language model at the multi-phone unit level, and then uses it to decode the audio in terms of multi-phone units. This is combined with a forced alignment of the audio to the transcriptions to identify the *actual realization* of every word occurrence in the training data. For each word, one then tablulates all the ways it has been spoken, along with counts of these events. This is illustrated in Figure 4.1 for the word “blake.” This indicates that “blake,” with dictionary pronunciation “b l ey k,” has been seen eleven times as “p l ey k,” ten times in expected form, twice as “t ax l ey k,” and so forth. Note that since the index is based on actual realizations, unexpected pronunciation variability (and system error) is modeled: in this case, we will learn that “p l ey k” is a pretty good indicator of “blake.” As the index is built from more and more data, all the common variability and confusions will be represented.

The weighted set of realizations is provided as an input to SCARF, which then creates a single vector representation for each word. The terms (e.g. bi-phone pairs) for each realization are extracted, weighted with the realization count, and accumulated. Then, the term-frequencies are computed for the word, along with the IDF scores of the terms. Finally, a TF-IDF vector is created for each word.

The reverse index is straightforwardly created, with one modification for efficiency: a word w will *not* occur in the reverse index (short-list) for term t if fewer that k% (typically 10%) of the realizations of the word contain t. In the example above, “blake” would not be placed on the short-list of “t ax,” since they have been seen together just twice out of 29 occurrences. This has the effect of ignoring random garbage, and without the restriction, the reverse index becomes unmanageably large.

4.4 Extensions

SCARF provides two modifications to the log-TF-IDF score which have a beneficial effect on accuracy.

4.4.1 Length Cost

The presence of terms with very low IDF values will have little effect on the cosine distance; they are essentially invisible. However, they do represent the insertion of acoustic units into a hypothesized word, and it can be beneficial to penalize this. If we denote the number of units within a segment by n, one way of doing this is to explicitlty tabulate $P(n|w_i)$ for each word w_i. This probability can then be weighted and combined with the log-TF-IDF score. SCARF implements a simpler and equally effective method, which is to simply assume that the lengths are distributed according to a gaussian distribution. If l_i is the average number of units in a realization of word w_i, and n_s the number actually observed in a hypothesized segment s, the length score is then defined as $\alpha(l_i - n_s)^2$.

4.4.2 Exact Match Score

A linear combination of log-TF-IDF score and the length cost can be used as a surrogate for the acoustic score to produce good word lattices. The one-best accuracy is improved, however, by making the following modification. Let $AC(s,i)$ be the acoustic score associated with postulating word w_i as the label for segment s. Let $seq(s) \in pron(i)$ denote the event that the unit sequence present in the segment is an exact match to a dictionary pronunciation of w_i. We now define:

$$AC(s,i) = \begin{cases} 0 \quad \text{if } seq(s) \in pron(w_i) \\ \text{log-TF-IDF}(s,i) - \alpha(l_i - n_s)^2 \quad \text{otherwise} \end{cases}$$

In other words, an exact match to a dictionary pronunciation is allowed zero cost.

4.5 Fast Match Process

The TF-IDF acoustic score defined is used as the basis of a segmental decoder which, as mentioned above, is essentially identical to the full SCARF decoder with just two features - an acoustic score, and the standard language model score. The recursions described in [9, 14] are used to do a segmental decoding, and the likeliest paths are output as a lattice, in the basic format of 5.5.1.

Chapter 5

Inputs

In this section, we describe the inputs to a SCARF system.

5.1 Handling Multiple Utterances

When multiple utterances are to be processed, SCARF combines all the information of a given type into a single file. Having the information from several utterances in a single file makes it unnecessary to do multiple file open and close operations, which is slow on many operating systems. Files containing multiple utterances are known as "blob" files and must have a suffix beginning with ".B", for example, ".Bpd" for phoneme detection files. The part of a blob file that is relevant to an utterance will always begin with a line containing the utterance ID (an ascii string) and end with a line containing a single period (.). While utterance identifiers are supported for clarity and debugging purposes, SCARF does not insist that they match across files, and it is the user must ensure that the utterances occur in the same order across the different kinds of files. Files may be gzipped, and files ending in ".gz" are automatically uncompressed.

5.2 Special Symbols

Generally SCARF avoids the use of arbitrary conventions and requirements. However, consistent with standard practice, certain symbols are considered special. `<s>` and `</s>` are required symbols: all utterances must begin with `<s>` and end with `</s>`. This is in order to correctly apply the language model. We recommend assigning these symbols special detector units (e.g. `!sent_start` and `!sent_end`) and ensuring that these are always present at utterance beginnings and endings respectively. The symbol `~SIL` is reserved for the representation of silence. When silence is intended in the language model or in a lattice, the symbol `~SIL` must be used. Finally, `<unk>` is reserved for the internal representation of words which are out-of-vocabulary with respect to the language model. It should not be used in the vocabulary. Note that we use lower-case for this symbol.

5.3 Detector Files

A detector file provides a raw sequence of detector events, also referred to as a detector stream. For example, a phoneme detector might form the basis of a detector stream. Multiple steams are supported, for example, a "fricative detection" stream could complement a phone detection stream. Each stream defines its own unique set of detector units, and these are not shared across streams.

The format of a detector stream consists of repetitions of the following pattern:

```
utterance-identifier
# stream-name stream
(unit time)+
.
```

In the body of the file, the first column specifies the unit name. The second specifies the time at which the unit is detected. It is used to synchronize between multiple feature streams, and to provide candidate word boundaries.

A sample detector file for multi-phone units might contain:

```
s01.19.{001624F7-3548-405F-9A14-82A53A916D59}.mpd
# multiphone stream
!sent_start 1
d@eh@l@iy 1905
vn 2770
!sent_end 3060
.
s01.19.{002CEDF2-1100-4432-BB22-D6ACC35D51D7}.mpd
# multiphone stream
!sent_start 1
dtmf 240
w@ao@l@m@aa@r@t 2205
!sent_end 3580
.
```

5.4 Baseline Files

The format of baseline files is identical to that of detector streams. However, words are present rather than detector units. For example,

```
s07.02.{B2AF433A-4C8C-4E70-B1B3-901309079C5B}.base
# baseline
<s> 65
longview 405
texas 935
</s> 1585
.
s07.02.{8D88D437-AC7A-40C6-9012-6C5F49AC7C94}.base
# baseline
<s> 1070
safeway 2505
</s> 3225
.
```

5.5 Constraint Files

SCARF uses constraint files for two purposes: to represent the set of all word sequences that are potentially consistent with the audio (the "denominator constraints") and to represent the set of word sequences that are consistent with both the correct transcription of an utterance, and the audio itself (the "numerator constraints"). While it is not enforced, the numerator constraints should be present as a subset of the denominator constraints.

5.5.1 Time Constraints

The simplest type of constraint files simply provide the starting and ending time of likely words. Any word ending at time t may be followed by any other beginning at time $t + 1$.

For example,

```
s07.02.{D96B23B1-EFD9-41F2-AC48-388D8B4F73EF}.dc
1 1510 <s>
```

```
1511 2370 mexican
1511 2800 mexican
2371 2940 </s>
2801 2890 food
2891 2940 </s>
.
s07.02.{0BA4B198-3602-4559-A49A-12C2A6705F64}.dc
1 20 <s>
1 210 <s>
21 200 [dtmf]
21 210 [dtmf]
201 1600 [side_speech]
211 1600 [side_speech]
1601 1900 best
1601 1880 french
1601 1880 fresh
1881 2460 market
1901 2460 market
2461 3100 </s>
.
```

All constraint files should begin with `<s>` and end with `</s>` to ensure proper language model context. Appropriate detections (e.g. of `sent_start`) should be added to any detection files, if not already present.

5.5.2 Connectivity Constraints

It is sometimes undesirable to allow full connectivity. For example in the numerator, it might introduce undesired words sequences. Or one might wish to annotate the lattice with features that are sensitive to cross-word context. Therefore, SCARF supports a more detailed format in which each word is assigned a "from-state" and a "to-state." Connectivity is implied only when one word's "to-state" matches another's "from-state." This is a classical lattice. An example is:

```
{5686EA4B-FB42-45AD-8E09-5095C9AAFF8F}
1 150 <s> 0 1
151 3705 jamaican 1 2
151 3790 jamaican 1 3
151 3876 jamaican 1 4
3706 4564 restaurant 2 6
3791 4564 restaurant 3 6
3877 4564 restaurant 4 6
4565 5140 </s> 6 7
.
{51E589C0-AC44-4CF4-B8A4-EA7CE6169E06}
1 120 <s> 0 1
121 645 pasadena 1 2
121 710 pasadena 1 3
121 805 pasadena 1 4
121 900 pasadena 1 5
646 1280 texas 2 6
711 1280 texas 3 6
806 1280 texas 4 6
901 1280 texas 5 6
1281 2020 </s> 6 7
.
```

The initial state of a lattice *must* be labeled "0", and there can only be a single initial state. There may be more than one final state. The final states are specified implicitly; any "to" state whose associated time is the maximum time present in the utterance is considered to be a final state of the lattice. Note that acceptable paths must meet two criteria: they must start in the start state of the lattice and end in an end state of the lattice, and must end in a specified end state of the language model. In particular, an end state of the language model is one ending in the word `</s>`.

5.5.3 External Features

In addition to connectivity constraints, lattices can further be annotated with externally defined features. This is done by adding a new field that contains one or more comma-separated features. The format of the annotation field is:
(name=score[,])+

For example, we might annotate the lattices with a new acoustic model score and a duration score:

```
{5686EA4B-FB42-45AD-8E09-5095C9AAFF8F}
1 150 <s> 0 1 newam=-0.035779,dur=0.149
151 3705 jamaican 1 2 newam=-0.885920,dur=0.3554
151 3790 jamaican 1 3 newam=-0.831206,dur=0.3639
151 3876 jamaican 1 4 newam=-1.176941,dur=0.3725
3706 4564 restaurant 2 6 newam=-0.236573,dur=0.858
3791 4564 restaurant 3 6 newam=0.000000,dur=0.773
3877 4564 restaurant 4 6 newam=-0.245767,dur=0.687
4565 5140 </s> 6 7 newam=-0.014512,dur=0.575
.
{CFD094C5-596E-4335-8581-EF90D7D1EA20}
1 20 <s> 0 1 newam=-0.035779,dur=0.19
21 1570 walmart 1 3 newam=-0.985598,dur=0.1549
1571 1740 </s> 3 4 newam=-0.014512,dur=0.169
.
```

When using external formats, SCARF must be told what to expect via the command line. This is done by adding a command line argument to any others present; for example, in the case above, one would add:

```
--external_format      newam,dur
```

5.5.4 Combinations

While time constraints must always be present, connectivity constraints and feature annotations can be used independently of each other. Thus, the lattice format may be described as repetitions of the following pattern:

utterance-id
(start-time end-time word [from-state to-state]? [feature-annotation]?)+
.

Connectivity constraints can be applied to either the numerator or denominator files, and need not be applied to both. If external features are used, they must be applied consistently to the numerator and denominator constraints. Regardless of the pattern used, it should be held constant within a file.

5.6 Transcription Files

SCARF does not directly use transcription files, but they are necessary in the creation of some of the other input files. In particular, the "numerator" constraint files are typically generated by intersecting the denominator constraints with the transcription. A transcription file consists of repetitions of the following pattern:

utterance-id
transcription
.

For example,

```
s07.03.{B2AF433A-4C8C-4E70-B1B3-901309079C5B}.tr
<s> longview texas </s>
.
s07.03.{8D88D437-AC7A-40C6-9012-6C5F49AC7C94}.tr
<s> safeway </s>
.
s07.03.{BD253938-9DC3-4A81-BBBA-9A4A20E7961B}.tr
<s> olympia </s>
.
```

5.7 Dictionaries

A dictionary providing canonical word pronunciations can be provided for each feature stream. For example, phonetic and syllabic dictionaries could be provided. As discussed previously, the existence of a dictionary enables the automatic construction of certain features that indicate (in)consistency between a sequence of detected units and those expected given a word hypothesis.

The format of a dictionary is:

# stream-name dictionary
(word unit+)+

For example,

```
# multiphone dictionary
<s> !sent_start
</s> !sent_end
aerospace eh@r@ow s@p ey@s
aerostar eh@r@ow s@t@aa@r
aesthetic eh@s th eh@t@ih@k
aesthetics eh@s th eh@t@ih@k s
aetna eh@t n@ax
```

5.8 Language Models

SCARF uses language models in ARPA format, without the header information. The sample below illustrates:

```
\data\
ngram 1=19982
ngram 2=3518595
ngram 3=3153527

\1-grams:
-1.6264 <unk>   -0.3662
-1.3786 </s>    0.0000
-99.999 <s>     -2.5619
-5.3882 aaron   -0.7248
-4.8588 abandon -0.7301
```

```
-4.6225 abandoned       -0.7223
...
\2-grams:
-1.1901 <unk> <unk>     -0.1514
-1.1387 <unk> </s>      0.0000
-5.8220 <unk> amicable  0.0000
-4.3504 <unk> amid      -0.1149
-4.6090 a beneficial    -0.2693
-4.7271 a beneficiary   0.0143
-4.0227 a benefit       -0.2145
...
\3-grams:
-3.7072 a <unk> mill
-4.4368 a <unk> mind
-1.3266 a banner that
-1.5949 a banner with
-0.5643 a banner year
-1.3317 zzzz best's chief
-1.3317 zzzz best's public
\end\
```

While it is possible to set the command line parameters so that the language model scores are not used, in all cases a language model must be specified.

5.9 Lexical Count Files

A lexical count file is used to create a TF-IDF index for the fast match. The format of the count file is

(word dictionary-pronunciation count realization)+

The columns are tab-separated. There is a separate line for every way in which a word has been realized in some sample training data. Typically the file is generated by the following process:

1. Perform an unconstrained decoding of the training data in terms of mutli-phone units
2. Perform a forced alignment of the training data to the transcriptions
3. For each word occurrence in the training data,
   - find the units that are spanned
   - break the multi-phone units down into phonemes to obtain the phonetic realization of the word. The forced alignment will provide the dictionary pronunciation that was used.
   - increment the number of times the word, dictionary-pronunciation, realization has been seen together

The first column in the lexical count file is the word identity. The second is the dictionary pronunciation. The third is the number of times the actual realization indicated in the fourth column has been seen. A sample from a lexical count file is:

```
mcdonalds       m ax k d aa n ax l d z  379     m ih k d aa n ax l d z
mcdonalds       m ax k d aa n ax l d z  329     m ax k d aa n ax l d z
mcdonalds       m ax k d aa n ax l d z  105     m ae k d aa n ax l d z
mcdonalds       m ax k d aa n ax l d z  80      m ax k d aa n ax l
mcdonalds       m ax k d aa n ax l d z  11      d aa n ax l d z
mcdonalds       m ax k d aa n ax l d z  6       m ae k d ao n ax l d z
mcdonalds       m ax k d aa n ax l d z  6       m ae k d aa n ae l d z
mcdonalds       m ax k d aa n ax l d z  5       m uh k d aa n ax l d z
mechanic        m ax k ae n ih k        7       m ax k ae n ih k
mechanic        m ax k ae n ih k        5       m ih k ae n ih k
mechanic        m ax k ae n ih k        1       m ih k s eh n t ax r
mechanic        m ax k ae n ih k        1       m ih k d aa n ax l d z
mechanic        m ax k ae n ih k        1       m ax k ao r m ih k
mechanic        m ax k ae n ih k        1       ao t ow m ih k ae n ih k
mechanic        m ih k ae n ih k        140     m ih k ae n ih k
mechanic        m ih k ae n ih k        4       m ax k ae n ih k
mechanic        m ih k ae n ih k        2       m ih k ae n ih k s
mechanic        m ih k ae n ih k        1       y uw ch ax r
mechanic        m ih k ae n ih k        1       m ih k eh n z
```

In the absence of aligned training data, a lexical count file can be created from the dictionary alone, simply by repeating each dictionary pronunciation into the last column, with a count of 1.

Chapter 6

A Sample System

This chapter presents the scripts used to train and decode with a Wall Street Journal system. For intellectual property reasons, we do not include the raw data, which is derived from the Linguistic Data Consortium corpora LDC97L20 (Pronlex), LDC93S6A,B,C (WSJ0), and LDC94S13A,B,C (WSJ1). Instead, we provide samples of each of the input files in Appendix A.

In the examples below, we use the following naming convention for file types:

- ".Bnc", ".Bnc.gz", ".Bdc", and ".Bdc.gz" for numerator and denominator constraint files, possibly gzipped.
- ".Btr" for transcription files
- ".Bbase" for baseline files
- ".Bsd" for syllable detection files

6.1 Making the Constraint Files

The first step in building a system is to generate the numerator and denominator constraint files. This can either be done using the SCARF Fast Match (scripts described in Section 6.4), or with an external system. In our first experiments, the lattices used as denominator constraint files were generated using a conventional system trained with HTK. The error rate of this system on the 20k open vocabulary test set is 9.7%.

To obtain the numerator lattices, we intersect the denominator with the transcript to obtain the subgraph that corresponds to the correct words. The source program "intersect.pl" does this, using the following syntax:

```
intersect.pl denominator.Bdc transcripts.Btr number-of-files-to-process > numerator.Bnc
```

After intersection, the original denominator files must be edited to remove those for which no intersection was found. Similarly, the other input files must be edited to cover exactly the same set. The source program "remove_no_match.pl" takes a list of utterance guids for which a match was found, a file to edit, and removes non-matching data. For example, the file "s03.06.matches" might have the list of matches, and look like this:

```
01ic0201
01ic0202
01ic0203
01ic0204
01ic0206
01ic0207
01ic0208
...
```

The screening could then be done like this:

```
./remove_no_match.pl Data/s03.06.matches Data/s03.01.Bdc > Data/s03.07.Bdc
./remove_no_match.pl Data/s03.06.matches Data/s03.02.Btr > Data/s03.07.Btr
./remove_no_match.pl Data/s03.06.matches Data/s03.03.Bnc > Data/s03.07.Bnc
./remove_no_match.pl Data/s03.06.matches Data/s03.04b.Bbase > Data/s03.07.Bbase
```

The constraint files which we will subsequently use are illustrated in Section A.1.

We note that the intersection approach we use has the disadvantage of excluding a large amount of training data when the error rates are high or the utterances are long. Alternative approaches include doing a forced alignment of the transcript to the audio, and adding that path to both numerator and denominator, or computing the oracle error rate path within the denominator and using it.

6.2 Training with the Baseline Feature

The following script illustrates how a simple model can be trained. This model uses only two features - the baseline feature, and the language model. This script does twenty iterations of gradient descent, and then writes the resulting parameters to "s03.08.model". Samples of the input files are shown in Sections A.1, A.2, A.4 and A.3.

```
VNUM=s03.08

./bin/train.exe \
    --lm                 Data/dectrain.lm \
    --num_constraints    Data/s03.07.Bnc \
    --den_constraints    Data/s03.07.Bdc \
    --basefiles          Data/s03.07.Bbase \
    --num_iterations     20 \
    --printModel         true \
    --use_levenshtein_features  false \
    --model              Data/${VNUM}.model \
    > Data/${VNUM}.out
```

After training, the resulting model can be tested. Denominator constraints must be made for the test data, and then decoding can proceed as in:

```
./bin/decode.exe \
    --lm                 Data/s03.11.lm \
    --model              Data/s03.08.model \
    --den_constraints    Data/s03.09.Bdc \
    --basefiles          Data/s03.10.Bbase \
    --printModel         true \
    > Data/s03.14.out
```

6.3 Oracle Experiments

A useful way to validate a SCARF setup is to create an oracle training set - one in which a set of "perfect" detections is created. For example, using phoneme detections, one can take a path through the numerator, and replace each word by its dictionary pronunciation (the detection times must be placed within the word's boundaries). If one does the same for the test data, then after training very good recognition should result. (Note that perfect recognition is not guaranteed because of homophones and ambiguous parses.) The script below uses an oracle set of syllable detections, and does this kind of training; it is similar to the baseline training script, but with several additions:

- A line specifying the name of the detector stream that will be used: `--stream_names0 syl`. The digit at the end of "stream_names" specifies the detector stream number. When multiple detector streams are used, they should be suffixed with "1", "2", and so forth. Similar notation is used in conjunction with the "existOrder" and "expectOrder" features.

- A syllable level dictionary, "s03.20.syldict"; Section A.5 contains a sample.
- Syllable detections, "s03.19.oracle_syl.Bsd"; Section A.6 contains a sample.
- Lines specifying that existence features should be used, and that the ngram order for these features should be 1. A line explicitly setting the existence feature order to 0 is also present. When a stream file is present, both must be specified.
- "printModel" is set to false, since the resulting model is too large to be conveniently viewed.

```
VNUM=s03.21

./bin/train.exe \
    --lm                 Data/dectrain.lm \
    --num_constraints    Data/s03.07.Bnc \
    --den_constraints    Data/s03.07.Bdc \
    --basefiles          Data/s03.07.Bbase \
    --stream_names0 syl \
    --stream0            Data/s03.19.oracle_syl.Bsd \
    --existOrder0        0 \
    --expectOrder0       1 \
    --dict               Data/s03.20.syldict \
    --num_iterations     20 \
    --printModel         false \
    --use_levenshtein_features  false \
    --model              Data/${VNUM}.model \
    | gzip > Data/${VNUM}.out.gz
```

Once the model is trained, the decoding script is similarly modified, and decoding can proceed. The language model and dictionary can (and in this case do) change when going from training to decoding.

```
./bin/decode.exe \
   --lm                 Data/s03.11.lm \
   --model              Data/s03.21.it17.model \
   --dict               Data/s03.23.syldict \
   --den_constraints    Data/s03.09.Bdc \
   --basefiles          Data/s03.10.Bbase \
   --stream_names0 syl \
   --stream0            Data/s03.23b.oracle_syl.Bsd \
   --printModel         false \
   > Data/s03.24.out
```

6.4 Using the SCARF Fast Match

The SCARF Fast Match can be used to create numerator and denominator constraints, as well as an initial one-best output. First, a lexical count file is used to create the TF-IDF index. Then, the decoding process is invoked to write out the denominator constraint file. To use the fast match to create numerator constraints for training, the denominator constraints can be intersected with the transcription, to produce the set of segmentations consistent with the known word sequence. Alternatively, a single segmentation of the transcription can be obtained via a forced alignment with a standard system and used as the numerator. (It should be added to the denominator as well to ensure probabilities less than 1.0)

6.4.1 Making the Index

A sample command line to create an index is as follows:

```
./bin/fastMatch.exe \
   -dictionary Data/s05.02.lex_counts \
   -ngram-level 3 \
   -indexfrac 0.1 \
   -model-out Data/s05.06.triphone.index
```

This takes a lexical count file of the form specified in Section 5.9, and produces an index based on triphone sequences. The parameter setting "indexfrac 0.1" indicates that any term (e.g. triphone) that occurs in less than 10will not be linked to the word in the reverse index. (Unless the index is order 1, single units are not used in the reverse index either.)

6.4.2 Making the Denominator

With the index created, a detector stream can be processed to create denominator constraint files. For example,

```
OUTPRE=s05.07

./bin/fastMatch.exe \
    -model-in Data/s05.06.triphone.index \
    -lm Data/ORIGINALS/From_gzweig3_wsj/s03.11.lm \
    -unit-infile Data/s05.01.phone.Bpd \
    -lattice-outfile Data/${OUTPRE}.matches.Bdc.gz \
    -max-matches 10 \
    -dm-lmwt 0.02 \
    -fm-lmwt 0.2 \
    -maxalphas 20 \
    -usexm true \
    > Data/${OUTPRE}.onebest
```

This script uses the index created by the previous script to convert the phoneme detections present in s05.01.phone.Bpd into a denominator constraint (.Bdc) file. The other parameters present control the pruning, and are described in Section 8.3.

6.4.3 Making the Numerator

The numerator constraint files may be created in one of two basic ways:

1. intersect the denominator lattices with the transcriptions.
2. perform a forced alignment of the transcriptions with an HMM system, and add the segmentation to the denominator. Use it as the numerator as well.

At the present time, it is unclear what the best approach is. Note that the intersection approach can be complex as one may wish to allow paths which are minor variations of the transcription, for example differing only in the presence of silence, detected noise, mumbling, hesitation and so on. A script, "intersect.pl," is provided to do a basic form of intersection. It takes a denominator .Bdc file and a transcription .Btr file, intersects them, and outputs a new constraint file. A parameter specifying the maximum number of files to be processed is also provided, for example:

```
./intersect.pl Data/s07.08.matches.Bdc.gz Data/s06.02.clean_train.Btr 31354 \
    > Data/s07.10.trainlat.intersected.Bnc
```

Chapter 7

Synthetic Data

In order to provide a full example of a working setup, without distributing any third-party data, we have synthesized training and decoding data. This is provided in the subdirectory "SampleData." To avoid using a proprietary dictionary, a graphemic dictionary was constructed.

7.1 Data Generation

The purpose of this data is simply to provide examples of proper file formats and to provide a setup that can be run immediately to verify the installation. Thus, the way in which the data was generated is not critical. Nevertheless, we provide a brief description. The steps were:

1. A language model was trained on Gibbon's *Decline and Fall of the Roman Empire.*
2. Numerator files were generated:
   (a) Transcriptions were decided on by sampling from the language model
   (b) Times were assigned by assuming a random duration of between 4 and 7 time units per letter in the word.
3. Letter detections were generated. Times are distributed within the containing word. These are stored in a file with the suffix ".Bld" for "blobbed letter detections."
4. Baseline files were generated by corrupting the numerator transcriptions with a 30% substitution rate and 2% deletion rate. A substitution is a randomly generated word within edit distance 1.
5. Denominator files were generated:
   (a) consider all segmentations and add every word within edit distance 1 of the detections
   (b) add silence as an option for each segment
   (c) downsample the arcs, while preserving the property that the numerator path remains in the denominator.
6. The detection stream was corrupted with insertions, deletions and substitutions.
7. The lattices were annotated:
   - A random number between 0 and 1 was assigned to each link
   - Its sign was changed to negative if it was incorrect. This was done with 55% probability for feature 1 and 70% probability for feature 2. Links occuring in both the numerator and denominator have the same annotations.

7.2 Sample Scripts

The script "train.sh" will train a SCARF model using the input files described above. Examining the file "train.log" shows evolution of the log data-probability, L1 and L0 norms, and the weights of the globally named features. Training will produce the file "scarf.model."

The final lines of "train.log" should look like this:

```
Log data prob per utterance after iteration 19: -0.341584 for 1000 examples
L1 lambda norm after iteration 19: 1926.34
L0 lambda norm after iteration 19: 3189
L1 was computed over 392558 features
[392556 @ 1.2354] global_lm weight
[392557 @ 2.00414] baseline weight
[392554 @ 0.378126] global f1 weight
[392555 @ 1.70554] global f2 weight
parameters updated
```

Note that the f2 feature has a higher weight than f1 due to the greater correlation between positive values and link correctness. Note also that depending on your compiler, the exact numbers may be slightly different due to a different order of arithmetic operations.

The script "decode.sh" takes this model and outputs a one best sequence, including time marks. The output will begin like this:

```
SCARF Decoder v1.0
{gzweig,panguyen}@microsoft.com

initialized a raw detector: sym
Added baseline detector
Warning: no ~SIL in the LM
Adding it with logp -0.999322
LM read with 29366 history states
dictionaries read
=DECODING
utt0.dc
1 7 <s>
8 59 possessed
60 70 of
71 85 the
86 123 largest
124 150 size
151 155 </s>
.
```

Chapter 8

Commandline Parameters

SCARF is controlled by a rich set of command line parameters. Invoking SCARF without any arguments will produce a list of these along with their default values and brief descriptions. This section describes them with greater detail. Note that many of the parameters specified in training are stored in the model file that is stored after training, and thus not re-specified in decoding.

8.1 Training

8.1.1 Parameters Controlling the Input Files

- `num_constraints`: This is the file that contains the numerator constraints.
- `den_constraints`: The file that contains the denominator constraints.
- `stream`: A file containing a particular detector stream. The streams are distinguished by a digit (starting with `0`) placed at the end of stream, e.g. "–stream0 myPhoneDetectionFile.Bpd." Alternatively, a comma-separated list may be used, for example "–stream a.Bpd,b.Bsd"
- `stream_names`: The name of the stream, also suffixed with the number of the stream, e.g. "–stream_names0 phone". Used for referencing the correct dictionary to use for a stream. Alternatively, a comma-separated list can be used.
- `basefiles`: The name of the file with the baseline detections, if used.
- `dict`: The name of a file which contains a list of the dictionaries to use. A comma-separated list may be specified as well.
- `lm`: The name of the language model.

8.1.2 Parameters Controlling Training

- `num_iterations`: The number of iterations of gradient descent to do.
- `l1_reg`: The l1 norm regularization constant to use. This is scaled by the number of training examples before it is used.
- `l2_reg`: The l2 norm regularization constant to use. This is scaled by the number of training examples before it is used.
- `saveItModel`: This can be specified if you want to save the model after each iteration of gradient descent. The name may include "%d" and the iteration number will be substituted at that point. For example, "–saveItModel myModel.iteration
- `model`: The name of the model file to write after all the iterations have been completed.

- `printModel`: When set to true, all the model parameters will be output to stdout before the program terminates. The parameters are named.
- `printScores`: When true, numerator and denominator path scores are output.
- `max_alphas`: This is the pruning parameter. In the forward recursion, only this many alpha values are propagated out of a given point in time.

8.1.3 Parameters Controlling the Features

- `use_levenshtein_features`: When set to true, levenshtein features are created for each stream that has a dictionary (this is also the default). When multiple comma-separated values are separated, they are applied to the detector streams in the order specified. In this case, there must be as many values as streams. For example, suppose we have "–stream deepnet,fdlpm." The specification "–use_levenshtein_features true,false" would mean that levenshtein features are created for the deepnet stream, but not the fdlpm stream.
- `expectOrder`: This specifies what order n-gram features to use in the expectation features. This is done on a stream-by-stream basis, with the stream specified by a digit appended as a suffix. For example, "–expectOrder1 3" would use 3-gram expectation features for detector stream 1. Setting the order to "0" effectively turns off the feature for the specified stream. Note that if this parameter is specified for one stream, it must be specified for all of them.
- `existOrder`: This is analogous to the "expectOrder" specification, except for existence features.
- `use_global_lm`: When set to "true", this turns on the use of the global language model features.
- `use_full_lm`: When set to "true", this turns on the use of local language model features as well. The two language model flags can be set independently of each other.
- `external_format`: As described in Section 5.5.3, this tells SCARF what lattice annotations to expect. It is used to add external user-computed features.
- `lexternal`: This is used to lexicalize external features. When an external feature is lexicalized, a separate feature is automatically created for each word. A comma-separated list of "true" and "false" is provided. If just one argument is provided, it is applied to all extrenal features. If multiple arguments are provided, the number must be exactly equal to the number of external features, and lexicalization is set positionally. The default is no lexicalization. For example, if we have `--external_format sys1,ppm,duration` then `--lexternal true,true,false` means that sys1 and ppm features will be lexicalized and duration will have a single weight for all words.

8.1.4 System Parameters

These parameters control system related aspects of the program.

- `binstdio`: When set to true, binary data is written to stdin, stdout, and stderr. For example, in this mode the Windows OS will not convert "\n" to "\r\n".
- `crtdbgbrk`: When set to true and when running Windows, invoking the program will launch the debugger as well.
- `printver`: Prints information about the compiler which was used.
- `bufstdio`: When set to false (the default) stderr and stdout are not buffered. This can be useful for debugging as ordering is maintained.

8.1.5 Parallel Processing Parameters

As described in Section B, SCARF supports MPI based parallel processing. The following parameters are relevant to this process.

- `enable`: When set to true, this enables MPI. Note that MPI-specific binaries must be compiled and used in conjunction with this.
- `redirect_log`: The MPI default is to send the output of all programs to a single log file. When set to true, this parameter will assign each process its own log file.
- `heartbeat.monfile`: This is a monitoring file into which processes write information about their existence.
- `heartbeat.durms`: Processes update the monitoring file in intervals of heartbeat.durms.
- `numreaders`: No more than numreaders processes are allowed to read the same file at once.
- `localProcDir`: This specifies a local directory into which a process writes. "substituted.
- `discipline`: This controls how data is dispatched to the processes, either in round-robin fashion, or by block. In round-robin, with four processes, process 1 might get data parts 1, 5, and 9 while process 2 gets parts 2, 6, and 10. In block mode, process 1 would get a consecutive sequence of parts, e.g. 1 through 3.

8.2 Decoding

The parameters governing feature creation are stored in the model produced by the training process, and should not be redundantly specified in the decoding scripts. Further, most of the the parameters controlling the training process itself are irrelevant and need not be specified. Thus in decoding, only the parameters related to the input files - as described in Section 8.1.1 - and a small number of others should be specified. Of course, numerator files are excluded. In addition to the basic input files, the relevant paramters are:

- `model`: The name of the model file to use.
- `printModel`: When set to true, all the model parameters will be output to stdout when the program begins.
- `max_alphas`: This is the pruning parameter as described above.
- `asConstraint`: When set to "true" the format of the output is a constraint file as described in Section 5.5.1, with `<s>` and `</s>` omitted.
- `clamp`: This is used to clamp a feature weight at a specific value during testing, thus over-riding the learned value. The syntax is `--clamp <fvpair[,fvpair]*>` . Each fvpair is feature_name@value. The features which are available are "baseline", "global_lm", and all external features. When an external feature is lexicalized, all weights for all words will be set identically to that value. A special name, which must be set first, is ALL, sets all features to a specified value. So, for instance, –clamp ALL@0,baseline@1 will reproduce the baseline.

8.3 Fast Match

The fast match parameters are divided into those controlling the creation of the index, and those controlling the search process.

8.3.1 Index Creation

- `dictionary`: This specifies the lexical count file to create the index from.
- `ngram-level`: The terms used in the index creation are ngrams of detection events. This controls the ngram level. Typically 2 or 3 is good. If this parameter is more than 1, 1-grams are ignored.

- `indexfrac`: Any term occuring in less than this fraction of a word's realizations will not be used in the reverse index.
- `model-out`: The name of the index file being created.

8.3.2 Fast Match Search

- `model-in`: The name of the index file to use.
- `unit-infile`: The name of the detector file to use as input.
- `lattice-outfile`: The name of the contraint file to write.
- `lattice-infile`: Optionally, a lattice can be read in as well. If this is done, the lattice is annotated with the TF-IDF scores, for use as user-defined features with SCARF.
- `lm`: The language model to use.
- `fm-lmwt`: In an initial step, a small number of word candidates is identified for each segment, based on the TF-IDF and *unigram* LM scores. This parameter controls the LM weight in this step.
- `dm-lmwt`: After the candidates have been identified, partial paths are extended, and the full n-gram LM is applied. This parameter controls the LM weight in this step. (The fast match itself has "fast" and "detailed" steps.)
- `max-matches`: The maximum number of word candidates to consider for a segment (based on the unigram LM).
- `maxalphas`: This is the pruning parameter that is applied when the full ngram LM is used. Only this many distinct paths will be allowed to propagate forward at any given time.
- `max-word-len`: Segments of up to this many units will be allowed (e.g. 15 for phoneme units).
- `lengthwt`: This controls the weight on the length portion of the match score, as described in Section 4.4.1.
- `phonedelta`: When the index is created, the average length of the realizations of a word is computed. If the length of a segment differs from this expectation by more than *phonedelta*, then the word will not be considered as a possibility.
- `usexm`: If the detection events in a segment are an exact match to a dictionary pronunciation and this parameter is set to *true*, then the acoustic similarity will be forced to the maximum possible value of 1.0.
- `max-files`: Stops processing of the input after this many files have been processed. Useful for debugging.
- `del_cost`: When lattices are annotated, a word's boundaries may not cover any detection events. In this case, the acoustic score is the average length of the word times the deletion cost.
- `ins_cost`: Recall that the acoustic score is the log of the cosine distance. In the case that this is log(0), the score is the deletion cost times the average length of the word, plus the insertion cost multiplied by the number of detection events in the segment.

Appendix A

Sample Data Files

This appendix provides samples of the data files used in the scripts of Section 6. In all cases, small parts of the entire files are reproduced. Elision is indicated by "...". For constraint and detector files, only the first utterance is shown.

A.1 Constraint Files

A.1.1 Denominator File s03.07.Bdc

```
01ic0201.dc
1 14 <s>
1 35 <s>
1 97 <s>
1 98 <s>
15 97 a
15 97 i
36 97 at
36 97 if
36 97 in
98 108 the
98 109 the
98 118 bad
98 118 bet
98 118 bette
98 118 but
98 118 debt
98 118 fat
98 118 fed
98 118 net
98 118 not
98 118 that
98 118 yet
98 124 debts
98 124 fats
98 124 that's
98 125 debts
98 125 fats
98 125 that's
98 127 deaths
98 127 debts
98 127 facts
```

```
98 127 fats
98 127 that's
98 127 thus
...
559 572 won
562 572 on
562 572 own
563 572 on
564 566 i.
564 578 out
565 572 on
565 578 out
566 573 of
566 577 and
567 572 in
567 573 as
567 573 of
567 578 at
567 578 it
572 609 regular
572 613 regular
573 576 a
573 609 regular
573 613 regular
574 609 regular
574 613 regular
577 609 regular
577 613 regular
578 609 regular
578 613 regular
579 609 regular
579 613 regular
610 652 attacks
614 652 tax
653 716 returns
653 726 returns
717 726 a.
727 727 </s>
```

A.1.2 Numerator File s03.07.Bnc

```
01ic0201.nc
1 97 <s> 0 3
98 127 that's 3 7
128 137 the 7 9
138 193 arcane 9 10
194 235 federal 10 11
236 271 income 11 12
272 312 tax 12 15
313 326 that 15 20
327 339 is 20 22
340 378 triggered 22 23
379 397 by 23 25
398 400 a 25 29
```

```
398 402 a 25 30
401 435 number 29 31
401 436 number 29 32
403 435 number 30 31
403 436 number 30 32
436 441 of 31 37
437 444 of 32 36
442 482 items 37 39
445 482 items 36 39
483 492 that 39 42
493 499 are 42 43
500 555 deductible 43 45
556 572 on 45 51
573 613 regular 51 53
614 652 tax 53 54
653 726 returns 54 56
727 727 </s> 56 57
.
```

A.2 Baseline File s03.07.Bbase

This file was generated from the output of a conventional speech recognizer.

```
01ic0201.base
# baseline
<s> 49
that's 112
the 132
arcane 165
federal 214
income 253
tax 292
that 319
is 333
triggered 359
by 388
a 399
number 418
of 440
items 463
that 487
are 496
deductible 527
on 564
regular 593
tax 633
returns 689
</s> 727
.
```

A.3 Transcription File s03.07.Btr

```
01ic0201.tr
```

```
<s> that's the arcane federal income tax that is triggered by a number of items
that are deductible on regular tax returns </s>
.
```

A.4 Language Model dectrain.lm

```
\data\
ngram 1=22576
ngram 2=3518595
ngram 3=3153527

\1-grams:
-1.6264 <unk>   -0.3662
-1.3786 </s>    0.0000
-99.999 <s>     -2.5619
-5.2167 'em     -0.5132
-5.0249 'n      -1.3559
-1.6584 a       -1.7978
-5.5853 a's     -0.3193
-2.8422 a.      -0.9925
-4.2927 a.'s    -0.4701
-4.9471 a.s     -0.5647
-5.3882 aaron   -0.7248
-4.8588 abandon -0.7301
-4.6225 abandoned       -0.7223
-5.2149 abandoning      -0.6492
-5.6119 abandonment     -0.9474
...
\2-grams:
-1.1901 <unk> <unk>     -0.1514
-1.1387 <unk> </s>      0.0000
-4.6410 <unk> 'em       -0.0535
-4.3612 <unk> 'n        -0.5923
-1.5604 <unk> a -0.1418
-2.9720 <unk> a.        -0.9748
-5.3260 <unk> aaron     0.0000
-5.4415 <unk> abandon   0.0000
-4.6208 <unk> abandoned -0.0409
-6.2861 <unk> abandoning        0.0000
-5.8220 <unk> abandonment       0.0000
-6.2861 <unk> abated    0.0000
...
\3-grams:
-1.0343 <unk> <unk> <unk>
-1.0742 <unk> <unk> </s>
-1.2867 <unk> <unk> a
-2.6270 <unk> <unk> a.
-4.7246 <unk> <unk> abandoned
-4.7246 <unk> <unk> abdul
-4.7246 <unk> <unk> abe
-2.6461 <unk> <unk> about
-3.8701 <unk> <unk> above
```

```
...
-1.3317 zzzz best's accountants
-1.3317 zzzz best's chief
-1.3317 zzzz best's public

\end\
```

A.5 Dictionary File s03.20.syldict

```
# syl dictionary
<s> !sent_start
</s> !sent_end
a ey
a's ey@z
a. ey
a.'s ey@z
a.s ey@z
aaron ey@r ih@n
abandon ax@b ae@n@d ih@n
abandoned ax@b ae@n@d ih@n@d
abandoning ax@b ae@n@d ih@n ih@nx
abandonment ax@b ae@n@d ih@n@m ih@n@t
abanto ax@b ae@n@t ow
...
```

A.6 Syllable Detection File s03.22.oracle_syl.Bsd

```
01ic0201.sd
# syl stream
!sent_start 49
dh 108
ae@t@s 118
dh 131
iy 134
ao@r@k 156
ey@n 174
f 204
eh@d@r 214
ax@l 224
ih@n@k 248
ah@m 260
t 285
ae@k@s 298
dh 317
ae@t 321
ih@z 333
t@r 349
ih@g 359
er@d 369
b 385
ay 391
ey 400
n 410
```

```
ah@m@b 418
er 426
ah@v 441
ay@t 457
ax@m@z 469
dh 486
ae@t 489
ao@r 497
d 509
iy@d 518
ah@k@t 527
ax@b 536
ax@l 545
ao@n 564
r 581
eh@g@y 589
ax@l 597
er 605
t 627
ae@k@s 640
r 668
iy@t 683
er@n@z 698
!sent_end 727
.
```

A.7 Phoneme Detection File s05.01.phone.Bpd

```
440c0401.pd
# phone stream
!sent_start 1
t 2
r 3
ay 4
b 5
ax 6
l 7
l 8
ae 9
s 10
t 11
m 12
ah 13
n 14
th 15
ax 16
g 17
r 18
iy 19
d 20
ih 21
...
ow 107
```

```
z 108
d 109
s 110
ah 111
m 112
!sent_end 113
.
440c0402.pd
# phone stream
!sent_start 1
dh 2
ah 3
k 4
ax 5
...
```

A.8 Lexical Count File s05.02.lex_counts

```
</s>    !sent_end       31138   !sent_end
</s>    !sent_end       1       !sent_start m ih s t !sent_end
<s>     !sent_start     30781   !sent_start
<s>     !sent_start     97      !sent_start dh
<s>     !sent_start     70      !sent_start hh
<s>     !sent_start     57      !sent_start ih n
<s>     !sent_start     40      !sent_start t
...
accepts eh k s eh p s   15      eh k s eh p s
accepts eh k s eh p s   1       s ae p s
accepts eh k s eh p s   1       eh k s eh p t dh
accepts eh k s eh p s   1       eh k s eh p t
access  ae k s eh s     31      ae k s eh s
access  ae k s eh s     1       eh s ih s
access  ae k s eh s     1       ae k t s eh z
...
```

Appendix B

Parallel Processing with SCARF

It is possible to run SCARF on a cluster of machines to speed up the training or decoding process. We have designed SCARF in such a way that only minimal changes to the command line are required to run on a cluster.

B.1 Environment

Parallel processing is supported using the Message Passing Interface (MPI). Under the Windows OS, the Windows High Performance Computing (HPC) environment might be leveraged. Under Linux, this is done with the OpenMPI library.

The Windows HPC SDK pack can be downloaded from here:

`http://www.microsoft.com/downloads/details.aspx?familyid=3FE15731`
`-B1B6-42DE-B278-5CCD46C0863B&displaylang=en`

Windows MPI clusters need to run the HPC software. Contact your system administrator if you have questions regarding this.

Open MPI may be found here:

`http://www.open-mpi.org/`

By design, SCARF does not use code specific to a particular MPI implementation. In principle, it is possible to run SCARF against any MPI compliant implementation.

B.2 Compiling

To run SCARF on the cluster, you need to build a special executable.

Windows / Visual Studio. Under Windows, copy the contents of
`C:\\Program Files\\Microsoft HPC Pack 2008 SDK`
to "hpc2008" in the SCARF source directory. Then, select the "Release-MPI" with x64 configuration. Once built, the executables should reside in `x64\Release-MPI`.

Linux. Under Linux, executables are built with:`make MPI=1`. If executables were built previously, you need to issue `make clean` beforehand.

B.3 Invoking

Invoking SCARF in parallel follows the regular procedure for launching jobs in MPI. This depends on the environment.

Windows / HPC In the HPC environment, it is advised to schedule a job using the "job" command, as follows:

```
job submit -jobname:TRAINING -numcores:auto mpiexec train ...
```

Open MPI Under Open MPI, `mpirun` is used to launch jobs:

```
mpirun train ...
```

Additional command-line parameters While running SCARF under MPI, the following is recommended:

1. Specify `--mpi.enable true`.
2. By default, MPI merges all processes' output in a single file. This may result in garbled output. Instead, one might prefer to set `--mpi.redirect_log` to a proper filename. The filename must include '%s', followed by '%d'. They will be replaced by "out" and "err" for stdout and stderr and the process ID respectively. Therefore, `log.%d.%s` would result in process 5's stdout be saved in `log.5.out`.
3. Unless you are debugging, specify `--bufstdio true`. This will allow buffering of standard I/O, resulting in better performance.
4. Set a local process directory to save cached files if necessary. The local directory must contain a '%d' to distinguish between processes on the same machine. For instance, this may be achieved by specifying `--mpi.localProcDir /var/tmp/mpi-%d`. The directory will be automatically created if needed. SCARF does not provide a way of cleaning up local directories.
5. Large files, such as language models and model files during decoding, should be broadcasted efficiently, rather than having all processes attempting to read the same file concurrently. This is done by prefixing the file name with `mpi:bc:`, for instance: `--lm mpi:bc:trigram.lm`. Files will be copied locally. Therefore, it is imperative to set `localProcDir` properly.
6. Once a large file has been broadcasted with `mpi:bc`, it is possible to skip the broadcasting process during subsequent runs, provided that processes are allocated on the same machines, which is typical in a single-user environment. This is done by replacing `mpi:bc:` with `mpi:ln:`. The process will look for the file in its local directory.
7. Under Windows only, a "heartbeat" file can be generated. Periodically, each machine involved in running SCARF will write vital statistics (CPU usage, network and disk I/O) in selected sections of the heartbeat file. We do not provide a tool to read this file. However, as processes write, its last modification time will be updated at least every second: this can be used as a rough indication that SCARF is still running.

An example command-line for MPI. For instance, SCARF may be run in parallel using the same command-line parameters as the normal, sequential program, as follows:

```
job submit train \
     --lm               mpi:bc:trigram.lm  \
     --dict             @dict.list \
     --num_constraints  train.Bnc \
     --den_constraints  train.Bdc \
     --stream0          phone.Bstr \
     --stream_names0    phone \
     --expectOrder0     2 \
     --existOrder0      2 \
     --stream1          multiphone.Bstr \
     --stream_names1    multiphone \
     --expectOrder1     1 \
     --existOrder1      1 \
     --printModel       false \
     \
     --mpi.enable true \
     --mpi.redirect_log log/train.std%s.id%d.log \
     --mpi.localProcDir /tmp/scarf/%d \
     --bufstdio true
```

Here, for performance, we elected to specify "mpi:bc:" to transmit the language model efficiently.

Appendix C

Broadcast News Lattices

SCARF was used extensively at the Johns Hopkins / CLSP 2010 Summer Workshop, with a particular focus on Broadcast News transcription. Subsequent to this, the LDC has released a set of Broadcast News lattices derived from those used at the workshop. These lattices represent a state-of-the-art baseline system on which to build, are in SCARF format, and are available through LDC catalog number LDC2011T06. The data and lattice construction is described in detail in [17] and more briefly in [18]. The distribution includes the constraint files, baseline detector stream, and lattice annotation that indicates whether a second, independently trained, system agrees with the primary system on a word-by-word basis.

The following command illustrates the use of these lattices to train a system. The lattices reside in the subdirectory `E:/WS2010_BN_Lattices/Train/train.Bnc`, and the language model in `E:/SCARF_References/V2Train/tg.lm`. Note that the LDC distribution does not include the language model. The line modifying the path indicates the location of HPC for parallel computing on a windows workstation, and the last set of command line parameters specifies MPI related values. The scripts are numbered as indicated by the `VNUM` specification, and the output is correspondingly prefixed.

```
VNUM=v01.01
export PATH=${PATH}:/cygdrive/C/Program\ Files/Microsoft\ HPC\ Pack\ 2008\ SDK/Bin

mpiexec -n 2 E:\\usr\\Programs\\scarf\\x64\\Release-MPI\\train.exe \
            --lm                 E:/SCARF_References/V2Train/tg.lm \
            --num_constraints    E:/WS2010_BN_Lattices/Train/train.Bnc \
            --den_constraints    E:/WS2010_BN_Lattices/Train/train.Bdc \
            --basefiles          E:/WS2010_BN_Lattices/Train/train.Bbase \
            --printScores        false \
            --printModel         false \
            --use_full_lm        false \
            --external_format    confirm \
            --num_iterations     20 \
            --model              Data/${VNUM}.model \
            --mpi.enable true --mpi.redirect_log ./log/${VNUM}.log.%s.%d \
            --mpi.heartbeat.monfile train.mon --mpi.localProcDir ./log/%d \
    > ./log/${VNUM}.log
```

After training, testing can be done on the RT03 test set lattices as in the script below. This script does not involve parallelization, and so is less complex. Scoring is done with the standard NIST tools.

```
VNUM=v01.02
E:/usr/Programs/scarf/x64/Release/decode.exe \
            --lm                 E:/SCARF_References/V2Train/tg.lm \
            --den_constraints    E:/WS2010_BN_Lattices/RT03/rt03.Bdc \
            --basefiles          E:/WS2010_BN_Lattices/RT03/rt03.Bbase \
            --printModel         false \
            --model              Data/v01.01.model \
            > Data/${VNUM}.out
```

Appendix D

Known Issues

At present, there is one known issue. With Cygwin Versions 1.7.6 and 1.7.7, gzipped input files may not be properly closed on program termination. Therefore, the inputs should be uncompressed before use. This issue has not been observed with Cygwin 1.5.18, 1.5.25, or 1.7.5.

Bibliography

[1] C-H. Lee, "From knowledge-ignorant to knowledge-rich modeling: A new speech research paradigm for next generation automatic speech recognition," in *ICSLP*, 2004.

[2] I. Bromberg, Q. Fu, J. Hou, J. Li, C. Ma, B. Matthews, A. Moreno-Daniel, J. Morris, S. Siniscalchi, Y. Tsao, and Y. Wang, "Detection-Based ASR in the Automatic Speech Attribute Transcription Project," in *Interspeech*, 2007.

[3] S. Sarawagi and W. Cohen, "Semi-Markov Conditional Random Fields for Information Extraction," in *Proc. NIPS*, 2005.

[4] J. Lafferty, A. McCallum, and F. Pereira, "Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data," in *Proc. ICML*, 2001.

[5] A. Gunawardana, M. Mahajan, A. Acero, and J. C. Platt, "Hidden Conditional Random Fields for Phone Classification," in *Interspeech*, 2005.

[6] M. I. Layton and M. J. F. Gales, "Augmented statistical models for speech recognition," in *in Proc. ICASSP*, 2006.

[7] M. I. Layton, *Augmented Statistical Models for Classifying Sequence Data*, Ph.D. thesis, Cambridge University, 2006.

[8] M. Reidmiller, "Rprop - Description and Implementation Details," Tech. Rep., University of Karlsruhe, January 1994.

[9] G. Zweig and P. Nguyen, "A segmental CRF approach to large vocabulary continuous speech recognition," in *Proc. ASRU*, 2009.

[10] H-K. J. Kuo and Y. Gao, "Maximum Entropy Direct Models for Speech Recognition," *IEEE Trans. on Audio Speech and Language Processing*, vol. 14, no. 6, 2006.

[11] J. Morris and E. Fosler-Lussier, "Discriminative Phonetic Recognition with Conditional Random Fields," in *HLT-NAACL*, 2006.

[12] J. Morris and E. Fosler-Lussier, "Further Experiments with Detector Based Conditional Random Fields in Phonetic Recognition," in *ICASSP*, 2007.

[13] A. Ratnaparkhi, "A maximum entropy model for part-of-speech tagging," in *Proc. EMNLP*, 1996.

[14] G. Zweig, P. Nguyen, and A. Acero, "Continuous speech recognition with a tf-idf acoustic model," in *Proc. Interspeech*, Submitted.

[15] X. Xiao, J. Droppo, and A. Acero, "Information retrieval methods for automatic speech recognition," in *ICASSP*, 2010.

[16] G. Zweig and P. Nguyen, "Maximum Mutual Information Multiphone Units in Direct Modeling," in *Proc. Interspeech*, 2009.

[17] G. Zweig, P. Nguyen, and et al., "Speech recognition with segmental conditional random fields: Final report from the 2010 JHU summer workshop," Tech. Rep., Johns Hopkins University, 2010.

[18] G. Zweig, P. Nguyen, D. Van Compernolle, K. Demuynck, L. Atlas, P. Clark, G. Sell, M. Wang, F. Sha, H. Hermansky, D. Karakos, A. Jansen, S. Thomas, S. G.S.V.S., S. Bowman, and J. Kao, "Speech recognition with segmental conditional random fields: A summary of the JHU CLSP 2010 summer workshop," in *ICASSP*, 2011.