An Architecture for Scalable, Universal Speech Recognition

David Huggins Daines

CMU-LTI-10-019

Language Technologies Institute
School of Computer Science
Carnegie Mellon University
5000 Forbes Ave., Pittsburgh, PA 15213
www.lti.cs.cmu.edu

Thesis Committee:

Alexander I. Rudnicky, chair
Bhiksha Raj
Noah A. Smith
Thomas Schaaf, M*Modal, Inc.

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
In Language and Information Technologies

Keywords: speech recognition, parallel computing, embedded systems

Abstract

This thesis describes MultiSphinx, a concurrent architecture for scalable, low-latency automatic speech recognition. We first consider the problem of constructing a universal "core" speech recognizer on top of which domain and task specific adaptation layers can be constructed. We then show that when this problem is restricted to that of expanding the search space from a "core" vocabulary to a superset of this vocabulary across multiple passes of search, it allows us to effectively "factor" a recognizer into components of roughly equal complexity. We present simple but effective algorithms for constructing the reduced vocabulary and associated statistical language model from an existing system. Finally, we describe the MultiSphinx decoder architecture, which allows multiple passes of recognition to operate concurrently and incrementally, either in multiple threads in the same process, or across multiple processes on separate machines, and which allows the best possible partial results, including confidence scores, to be obtained at any time during the recognition process.

Acknowledgments

This thesis would not be possible without the support of all the friends, family, faculty, and colleagues who steered, encouraged, and supported me all the way. First and foremost, I am deeply grateful to my advisor, Dr. Alexander I. Rudnicky, for his support through the MLT and PhD programs here at CMU, and for all of his advice on research, writing, and presentation. He has always been willing to give me the time and resources to explore unconventional ideas, and to get me to explain these ideas in ways that make sense to myself and others.

Obviously I would not be here at all without my parents, who I cannot thank enough for all their love and support and for the opportunities they've provided me. In particular, my father is in a way directly responsible for this thesis. After all, he gave me my first computer (a Timex-Sinclair ZX81) and introduced me to the concept of Fourier series at a ridiculously young age. This is what happens when you tell your kids that it's all made out of sine waves... Somewhat later in life, he sent me a copy of Anthony Burgess' rambling and occasionally fanciful treatise on linguistics, *A Mouthful of Air*, which inspired me to hang up my messenger bag and sign up for LIN1101 at the University of Ottawa. Somewhere in the phonetics portion of the course, I thought to my self, "wait, this is all made out of sine waves too, isn't it?" The rest, as they say, is history. Along the way, however, I was inspired greatly by my professors at the Department of Linguistics, particularly Drs. John T. Jensen, Ian MacKay, and Shana Poplack, the latter of whom I credit with introducing me to the concept of thinking about language in probabilistic terms.

For bringing me to Pittsburgh and CMU in the first place, I am deeply indebted to Kevin Lenzo and Dr. Alan W. Black. Witnessing speech technology research in action during my years at Cepstral was the prime motivator for me to undertake this PhD.

Last and most of all, thank you, MJ, for all your sacrifice and support through this whole process. It's a testament to your strength and compassion that we've made it through a PhD together and lived to tell about it. This dissertation is as much yours as it is mine.

Glossary

ASR Automatic Speech Recognition, the conversion of a speech signal to a symbolic representation by computational means. 8, 14, 16, 20, 22, 64, 90

Baum-Welch An algorithm, named after its inventors, for maximum likelihood estimation of the parameters of an HMM from unlabeled input data. 19, 33

beam In a heuristic search algorithm, the beam is the range of scores outside of which current states or paths will not be extended. This is usually done by applying a factor less than 1.0 to the best score of all paths reaching a given point and discarding all paths whose scores are less than the resulting score. 38, 39, 44, 45, 51, 69

CFG Context-Free Grammar, a description of a formal language consisting of rules which expand a single non-terminal symbol to any sequence of terminal and non-terminal symbols. Frequently used to describe natural language in higher-level understanding tasks. x, 53

computational complexity The number of basic computational operations which must be performed in order to achieve a given task, expressed as a function of the input to that task. 9, 47, 61, 105

cross-entropy If entropy is a notion of the *inherent* uncertainty contained in a model, cross-entropy is a notion of the "functional" uncertainty of the model when applied to data drawn from another, perhaps unknown, distribution. Namely, the cross-entropy from a model $Q(x)$ to a distribution $P(x)$ is the expectation of the information in an event or symbol x drawn from $P(x)$ but evaluated according to $Q(x)$, or $\sum_x P(x) \log \frac{1}{Q(x)}$. vii, 12, 33, 50, 52, 71, 113

decoding Translation of a message into codewords. In the case of speech, refers to the translation from a representation of a speech signal into a sequence of words or linguistic units. Usually, but not always, the input representation is the acoustic signal (for example, we can also speak of decoding a word lattice). Often used synonymously with search. viii, x, 9, 10, 14, 16, 19, 28, 31–33, 39, 40, 46, 54, 57, 58, 60, 61, 64, 65, 67–69, 71, 75, 77, 79–81, 85, 88, 90, 92, 93, 98, 113

divergence An information theoretic measure of the difference between probability distributions. The most common divergence measure is the Kullback-Leibler divergence, also known as relative entropy or information gain, which is best described as the difference between the cross-entropy from one distribution to another and the entropy of the first distribution. In keeping with the description of information as a reduction in uncertainty, divergence is characterized as the *increase* in uncertainty that

results from having the "wrong" model for some observed data. For this reason it is used casually in this thesis to refer to the "wrongness" of a model or representation (e.g. a lattice or a language model). 33, 50, 51, 54, 60

domain A distinct subset of a human language with characteristic vocabulary, syntax, or acoustic properties. For example, the conversational telephone speech domain of English refers to the style of speech used in casual telephone conversations. 8, 19, 31, 47, 50–57, 59, 61, 64, 69, 70, 82, 122

EDGE Enhanced Data rates for GSM Evolution, an extension of the Global Standard for Mobile (GSM) wireless networking standard, allowing higher bitrates (though not up to broadband speeds). 2

entropy A corrolary of Shannon Information, entropy is a mathematical notion of the inherent uncertainty contained in a probability distribution, or, by extension in a probabilistic model of some observed data. The entropy of a distribution $P(x)$ is simply the expected or average information in an event or symbol x drawn from $P(x)$, or $\sum_x P(x) \log \frac{1}{P(x)}$.. vii, 11, 13, 113, 117, 122

EV-DO Evolution-Data Optimized, a wireless broadband networking standard used on mobile phones and other mobile devices, primarily in the USA and Japan. 2

FSG Finite-State Grammar, a grammar describing a regular language, or one which can be recognized by an automaton with a finite set of states and no memory. Nearly all language models or grammars used in speech recognition are finite-state. In speech recognition it is usually assumed that the arcs in these grammars are weighted. 57, 58

FST Finite-State Transducer, a mathematical formalism for representing a regular relation between two alphabets of symbols by way of a finite-state machine whose arcs have both input and output symbols. 58, 59, 101

full word In the MULTISPHINX system, we use this term to refer to one word in a class represented by a pseudoword. For example, "hipster" is a full word with "hip" as its pseudoword in the example lexicon used in Figure 5.9. 86, 87

HMM Hidden Markov Model. A statistical model used to model a process which evolves over time, where the exact state of the process is unknown, or "hidden". vii, x, 14–20, 26–28, 33, 39–41, 43, 45, 87, 91, 93, 114, 117

HSPA High-Speed Packet Access, an extension of the UMTS (Universal Mobile Telecommunications System) allowing higher upload and download speeds (up to 14 Mbps down and 5.8 Mbps up) over existing UMTS networks. 2

hypothesis A single sequence of words or linguistic units considered by a speech recognizer as the result of decoding an input utterance. May also refer to one particular component of such a sequence, as in a "word hypothesis". ix, 9, 10, 14, 16, 20, 26, 27, 29–33, 39, 40, 42, 90, 94–96, 98, 103, 113

information A mathematical notion of the reduction in uncertainty incurred by observing an event, or equivalently reading or receiving a symbol in data encoding or transmission. The Shannon Information of an event or symbol x is equal to $\log \frac{1}{P(x)}$ where $P(x)$ is the probability of observing an event x according to a probability mass function defined over a set of possible outcomes (either events or symbols in an alphabet). vii, viii, 11–13, 33, 93, 94

latency The amount of time between two stages in processing of an event. In this thesis, most commonly refers to the time between the reception and recognition of a segment of audio by a speech recognizer. 9, 33, 41, 82, 90, 108–113, 115, 123, 124

lattice A directed acyclic graph representation of the set of hypotheses generated by a speech recognizer, where both word identities and timing information are represented . vii, viii, 12, 29–33, 37, 40–43, 57–60, 64, 71, 75, 77, 79–82, 90–92, 94, 96, 98, 101, 103, 105–108, 113–116

LDA Linear Discriminant Analysis, a dimensionality reduction technique which uses the eigenvectors of the ratio of in-class to between-class covariance matrices to project data onto a lower-dimensional space. 26

lookahead Many optimizations to speech recognition rely on information from a few moments in the future. In continuous recognition, this is accomplished by extracting the desired information and delaying the part of recognition which requires it by the necessary amount of time, which is known as the lookahead window. 68, 82, 90, 91, 94

LPC Linear Predictive Coding, the encoding of a discrete 1-dimensional signal by using a linear combination of the preceding samples to predict the following sample. 23–25

LVCSR Large Vocabulary Continuous Speech Recognition. Automatic recognition of fluent, connected speech, where the vocabulary is large or unknown. 8, 9

MAP Maximum A-Posteriori. Refers to a type of optimization where the objective function is the *a posteriori* probability of an event (or symbol, or sequence). This is the probability of that event, conditioned on a previous belief (or model) about the probability of that event (the *a priori* probability distribution) and one or more observations of that event. A fundamental concept in Bayesian inference. Also refers to the application of MAP optimization to the parameters of an acoustic model for speaker adaptation. 9, 10, 20, 32, 51

MFCC Mel-Frequency Cepstral Coefficients, the coefficients of the cepstrum of the short-term spectrum, downsampled and weighted according to the mel scale, a frequency scale thought to represent the sensitivity of the human ear. 25

MLLR Maximum Likelihood Linear Regression. An algorithm for computing a linear transformation which, when applied to the means of one or more Gaussian Mixture Models, maximizes the likelihood of a set of data according to thos models. Used for rapid speaker adaptation. 51

PCA Principal Components Analysis, a dimensionality reduction technique which uses the eigenvectors of the covariance matrix to project data onto a lower-dimensional space. 25, 26

pseudoword In the MULTISPHINX system, a pseudoword is an entry in the lexicon which represents a class of words to be disambiguated in a future pass of search. For instance, "hip" is the pseudoword for "hippie", "hipster", "hips", "hypocrite", and "hypocrites" in the example lexicon used in Figure 5.9. viii, 75, 77, 79, 87, 88

real-time factor Often abbreviated as xRT, the measure typically used to report the performance of a speech recognizer. This is calculated as the ratio between the amount of time required to decode an utterance and the length of the utterance. For example, a realtime factor of 0.2 xRT means that each second of audio requires 0.2 seconds to decode.. 65, 67

rescoring Also known as *reranking*, rescoring is the re-evaluation of a set of top-scoring hypotheses using a separate, usually more complex model. Models for which inference is costly, such as probabilistic CFGs, are frequently used for rescoring only. 40, 41, 43, 57, 61, 64, 69, 72, 77, 82, 85

search In automatic speech speech recognition, the process of searching the set of linguistic or symbolic representations of an utterance for one (or more) which are considered the most probable by the statistical models used by the recognizer. Often used synonymously with decoding. vii, x, 9, 10, 16, 26–33, 36, 37, 39–45, 68, 81, 90–96, 98

search graph The recursive network traversed by search in speech recognition. Often refers specifically to the internal representation of the lexicon and language model, where the arcs represent phonemes or words. 29, 30, 39, 51, 65, 90, 93

search space The set of possible transcriptions for an utterance which are considered by the speech recognizer during recognition. 8–10, 22, 29, 30, 33, 38, 50, 69, 83

semi-continuous An HMM-based acoustic model, whose output density functions are Gaussian Mixture Models which share a single set of Gaussian distributions. 38, 54, 93, 94

task A specific instantiation of a general technology, for example the use of automatic speech recognition (general technology) for medical transcription (task). 8, 44, 47, 50, 64, 77, 78, 80, 86, 122

tied When multiple HMM states in an acoustic model share a set of parameters, these states are said to correspond to a single "tied" state. This is usually done for states that, despite having different symbolic representations, are acoustically similar. This allows for more compact and efficient acoustic models and better use of sparse training data. 38, 47, 51, 54

utterance The longest segment of speech operated on at one time by an automatic speech recognizer. Typically corresponds to an uninterrupted phrase, sentence, or paragraph spoken by a single speaker. viii, x, 8, 40, 41, 43, 50–53, 55, 65, 69, 71, 92, 93, 107

VQ Vector Quantization, the representation a continuous vector space with a discrete set of prototype vectors or codewords. 15

WFST Weighted Finite-State Transducer, a finite-state transducer whose arcs have weights. In automatic speech recognition, these can be used to represent the probabilistic models which make up a recognizer, and can be composed to produce a transducer directly mapping input speech units to output words with the appropriate probabilities or scores. 30, 31, 68

word error rate Often abbreviated as WER, the measure typically used to report the accuracy of a speech recognizer. This is calculated as the minimum total number of errors (insertions, deletions, and substitutions) in the output of the recognizer compared to a reference text, divided by the total number of words in the reference. It is possible for the error rate to exceed 100%, if there are a large number of insertions.. 64, 65, 67, 70, 74, 77, 79, 82, 85

Contents

List of Figures

List of Tables

Chapter 1

Introduction

After several decades of refinement, automatic speech recognition technology has advanced to the point where it is sufficiently accurate and flexible for a number of highly practical applications. In particular, it has become popular in mobile and hands-free situations, where it provides the ability to more safely and reliably access information systems and create textual input. These accomplishments can be attributed partly to advances in algorithms and statistical modeling for speech recognition, but also in large part due to the massive advances in CPU power, storage, and network bandwidth that have occurred over the same time period, and most crucially the widespread availability of wide-area wireless IP networks (e.g. EV-DO, EDGE, HSPA, etc.)

Without exception, the most advanced mobile speech recognition systems rely on a data connection to function properly, and limit their on-device processing to audio capture and encoding. This architecture is well suited to tasks such as Internet search and messaging which presume the existence of a data connection, and devices such as smartphones which rely on Internet access for a large part of their functionality. However, there is a large class of applications which cannot depend on a data connection, or where security or privacy concerns make it undesirable to entrust voice input to a remote server. In the consumer realm this class includes in-car voice control, speech translation, music and contact search, and potentially other types of on-device search.

It is clear that the use cases as well as the functional requirements of server-based and on-device speech recognition are quite different. However, these use cases are exclusive neither in time nor space, and it is no longer uncommon to encounter both types of speech recognition operating on the same device. One may, for example, use the on-device speech recognition on the Apple iPhone for dialing and other basic control tasks, while using Google's server-based voice search application on the same phone for Internet and local search and Nuance's server-based dictation application for composing e-mail and text messages.

Finally, mobile device hardware has advanced to the point where it is capable of handling medium-vocabulary speech recognition tasks in real time. This raises the possibility of providing much lower latency for speech recognition by processing audio data immediately as it becomes available. However, for the best recognition accuracy, and for inherently network-based tasks such as Internet search, it continues to make sense to perform recognition remotely, allowing much more substantial computational resources to be devoted to the problem.

What is crucial to note here is that the current paradigm enforces a choice of either server-based or on-device recognition for any given task, which leads either to an insufficiently flexible speech recognition engine, or to a wasteful duplication of effort, as in the above example of three separate engines in use by the same device. It is this problem, namely the inflexible and inefficient allocation of computing resources in mobile speech recognition systems, which is addressed by this dissertation. However, it turns out that the strategies devised for this problem are applicable to other problems of speech coding and recognition which require serialization, transformation, and expansion of the hypothesis space. This thesis presents a unified architecture for solving these types of problems.

1.1 Thesis Statement

The task of automatic speech recognition can be conceived of as a process of transduction from a low-information representation of the utterance to a high-information representation. By low and high information here, we refer to the information *density* or *rate* of a representation; for example, an audio file is a low-information representation because it takes a large number of samples to represent a few words, and

because each sample gives us little to no information about the content of the message. By contrast, a compressed text file is a high-information representation, because the same few words can be represented with just a few bits, and because each bit gives crucial information about the content. At any level of representation, recognition errors result from a divergence between the system's probabilistic model of that level of representation and the true distribution of the target language. This divergence tends to increase as we move to higher-information representations, as errors at a lower level of representation propagate to higher levels, leading to cascading failure.

Fortunately, the inverse of this statement is also true in that divergence between reality and models tends to decrease at lower levels of representation. This implies that errors at higher levels are recoverable to some extent if the corresponding low-information representation is retained or can be reconstructed. This is particularly true if the cause of the higher level errors is known *a priori*, for example a known vocabulary or domain mismatch. In this case we are not so much correcting errors as we are expanding or transforming the search space.

The ability to expand or transform the search space allows for a flexible and scalable architecture for speech recognition, which is described in this dissertation. Specifically, this thesis describes the mechanisms of expansion and transformation and their application to domain-adaptive and large-vocabulary speech recognition. In addition, it describes the construction of a "core" recognizer which leads to an optimal expansion for a given level of complexity. It provides an architecture and implementation of this technique which allows for processing load to be shared between a mobile client and remote server, or between multiple threads on the same computer. Finally, this architecture makes it possible to treat speech recognition as an *anytime algorithm* [Zilberstein, 1996], where the results of recognition at a given timepoint reflect the best available accuracy, and this level of accuracy improves over time.

1.2 Contributions

In [Ringger & Allen, 1996], an error-correction model based on statistical machine translation techniques and the noisy channel model was used to compensate for domain mismatch between the training and test sets. The motivation for this work was to allow a speech recognition engine to be used as a "black box" in a variety of spoken dialog systems, that is, without requiring customized language and acoustic models for every domain. In this thesis, we revisit this idea, extending it to use acoustic and phonetic information in the error modeling and correction process.

First, we consider the problem of bootstrapping a new speech application, where domain-specific training data is limited or non-existent. In this case, we use the information in the baseline acoustic and language models to predict the errors which result from applying these models to the domain language. This information can be used to construct an external error correction model as in the previous work mentioned above, or to adapt the language model. We accomplish this primarily by constructing a mapping between the baseline vocabulary and the domain specific vocabulary. This can be done at various levels of representation. For example - at a high level we can simply split and join compound words based on their orthographic representation. At the next level, we map between domain words and sequences of baseline words which are phonetic matches (homophones) or near-matches. At the lowest level, we find sequences of baseline words whose representation in the acoustic model is minimally divergent from that of domain words. There are advantages and disadvantages to each of these strategies, and we explore techniques for combining them.

Second, we consider the more specific case of mapping from a subset of a vocabulary to the full vocabulary. In this case it becomes obvious that the choice of the subset is very important to the performance of the system as a whole. This supports one of the important ideas of this thesis, which is that knowing the structure of approximations made to a model can facilitate recovery from the errors introduced by said approximations. In Chapter 7 we consider the implications of this for other types of model approximations and other natural language processing tasks.

Finally, we describe the implementation of these techniques in a concurrent, distributed architecture for speech recognition in the form of the MULTISPHINX system. This is an extension of the existing POCKETSPHINX system which factors the recognition process into a queue of independent recognition tasks, each of which operates inside a fixed or elastic lookahead window, and which communicate via a hierarchical, time-based, garbage-collected lattice structure. We demonstrate results in terms of speed, accuracy, and memory efficiency for this system.

1.3 Thesis Organization

This thesis describes the development and implementation of the MULTISPHINX system, a flexible and scalable speech recognizer which provides for rapid domain adaptation and distributed processing between a mobile device and remote server.

Chapter 2 surveys the field of automatic speech recognition, with a particular emphasis on concepts and previous experiments relevant to the thesis statement. We describe in detail the problem of heuristic search in speech recognition, grounding this in relevant aspects of information theory and automata theory.

In Chapter 3 we describe the genesis and evolution of the POCKETSPHINX recognition system which constitues the baseline system for this thesis. This chapter introduces the specific terminology used in this system and the details of the architecture on which the following work is based. In addition, the CPU and memory performance of this system are described, both on standard benchmark tasks as well as the specific data sets used in this thesis.

Chapter 4 defines the idea of a "domain" and describes the motivation behind domain-adaptive speech recognition. This is specifically situated in the problem of developing language resources for mixed-initiative spoken dialog systems and conversational user interfaces. We describe methods for bootstrapping these resources from minimal amounts of domain knowledge and give results with varying amounts of domain-specific training data.

Chapter 5 moves from domain adaptation to the related problem of vocabulary expansion for multi-pass systems. We consider both the problem of expanding the search space of a recognizer to allow a larger vocabulary in rescoring or re-recognition, as well as the question of designing the core vocabulary to optimize the results of such a system. We describe a novel, tunable method of integrated vocabulary pruning and expansion which allows the computational load to be more evenly balanced between recognition passes.

In Chapter 6, we describe the implementation of MULTISPHINX, which integrates domain and vocabulary adaptation into a concurrent, distributed architecture for on-line speech recognition. We detail the means by which multiple passes of speech recognition can be run concurrently in separate threads or over a network.

Finally, Chapter 7 summarizes the work contained in this thesis and describes several unsolved problems for future research. As mentioned above, we also specifically consider the extension of techniques discussed in this thesis to other natural language processing and understanding tasks.

Throughout this document, rather than using pseudocode to describe algorithms, I have adopted the convention of describing them in the Python programming language. By contrast, where low-level implementation details, such as the layout and composition of binary data structures, are presented, snippets of the C language implementation have been used instead.

Chapter 2

Automatic Speech Recognition

Automatic speech recognition, hereafter referred to as ASR, is the identification of linguistic content (words, phrases, or concepts) in audio data without human intervention. In the most general case, this can be considered an AI-complete problem [Shahaf & Amir, 2007], or one whose solution implies a complete solution to all problems of artificial intelligence. That is, to recognize speech in all situations and conditions requires a full knowledge of the ontological context of the utterance in question. This is because speech does not exist in isolation but as part of a pattern of human communication which refers to things in the real world.

For this reason, ASR is nearly always treated in the context of a particular application. It is also common to talk of the task or domain in which an ASR system operates. While a broad range of applications have been constructed or proposed based on this technology, ASR systems can generally be grouped into four main tasks, which are presented here in roughly increasing order of difficulty:

- *Voice control* - Use of voice commands, typically from a single user and directed at the system, to control a computer or some application running on it.
- *Dictation* - Conversion of speech, typically from a single user and directed at the system, to text, with control components to allow for editing and error correction.
- *Dialog* - Speech-based interaction between a computer and a human, whose purpose is to achieve some external goal.
- *Transcription* - Conversion of speech, typically from multiple users and not directed at the system, to text. This is frequently used to provide input to other natural language processing tasks, such as information retrieval and machine translation.

For these tasks, there are certain natural affinities to manners of speaking, classes of devices used for input, and acoustic environments, which give rise to specialized, task-specific systems. We discuss the problem of domain and task specificity further in Chapter 4.

2.1 Isolated and Continuous Speech Recognition

ASR is generally thought of as being a problem of pattern classification. Namely, we are attempting to classify an observation (or utterance) O as belonging to a class S, where S is a symbolic or linguistic representation of the observation. In the simplest case, there exists a finite vocabulary V of isolated words or phrases, and the recognition task consists simply of finding the word $S \in V$ which matches the utterance. This type of recognition is frequently implemented without the use of probabilistic models, for example using *dynamic time warping* [Itakura, 1975].

In a more general case, which is the one with which this proposal is concerned, the classification S is a sequence of words, and the observation consists of connected, natural speech. While the words in S are still drawn from a finite vocabulary, the number of possible classes or sequences is now extremely large. We refer to the set of possible word sequences as the search space and denote it with the symbol $\mathbb{S}$.

This task, *large-vocabulary continuous speech recognition*, known hereafter as LVCSR, is considerably more difficult, for several reasons:

- There are no clear boundaries between words in the input.

- The pronunciation of words can vary considerably due to context.
- Since the search space is extremely large, is no longer possible to simply test each possible classification in turn and pick the best.

Since we cannot exhaustively evaluate the entire search space, all LVCSR systems rely on heuristic search algorithms. The effect of the various heuristics is that a much smaller effective search space, denoted $\mathbb{S}'$, is actually considered by the recognizer. The speed and accuracy of recognition depend heavily on the effects of the heuristic used to determine $\mathbb{S}'$. Since $\mathbb{S}'$ is that part of $\mathbb{S}$ which is, in fact, exhaustively evaluated by the recognizer, the computational complexity of the recognizer is a non-decreasing function of the size of $\mathbb{S}'$. However, as the size of $\mathbb{S}'$ decreases, the probability that the true classification S lies outside $\mathbb{S}'$ increases, and therefore the error rate of the recognizer tends to increase. A primary goal of this thesis is to develop heuristics for decreasing the size of $\mathbb{S}'$ in such a way that the missing elements can be reconstructed as needed by subsequent phases of search. This can also be viewed as modeling and correcting the errors incurred by these heuristics.

As well, practical systems require that recognition be done faster than real time, or in other words, that it should take less than one second of clock time to recognize one second of input. This implies that that results of recognition should be available either concurrently with the input or as soon as possible after a chunk of input (a sentence, for example) is complete. We discuss further this problem of minimizing latency, and our approach to it, in Section 6.5, but at this point is is sufficient to note that this imposes on our models the requirement that they be sufficiently simple as to be computable in real-time, and, if possible, that they be time-synchronous, such that recognition can be done progressively as new input is available.

2.2 Noisy Channel Paradigm

Virtually all successful implementations of LVCSR are based on the information-theoretic concept of the *noisy channel paradigm*. Conceptually, we treat the speaker as an *information source*, which emits a sentence S according to a probability distribution $P(S)$. We assume that S has been sent through a noisy channel, which has "corrupted" the original message, producing a speech utterance O, according to a probability distribution $P(O|S)$. Speech recognition, therefore, is viewed as a process of decoding, or recovering the original message.

To decode, we search for the hypothesis $\hat{S}$ which minimizes the probability of error.[1] This leads us to the *maximum a posteriori* (MAP) hypothesis, which is, as its name indicates, the hypothesis with the maximum posterior probability $P(\hat{S}|O)$.

$$S_{MAP} = \arg\max_{\hat{S} \in \mathbb{S}} P(\hat{S}|O) \tag{2.1}$$

By Bayes' Rule, this can be expressed in terms of the probability distributions over the source and the channel:

[1] In fact, we are implicitly searching for $\hat{S}$ which minimizes the expectation of a *loss function*, which in this case is simply the *zero-one* loss function $\delta(S, \hat{S})$.

$$S_{MAP} = \arg\max_{\hat{S}} \frac{P(O|\hat{S})P(\hat{S})}{P(O)} \tag{2.2}$$

$$= \arg\max_{\hat{S}} P(O|\hat{S})P(\hat{S}) \tag{2.3}$$

$$= \arg\max_{\hat{S}} \log P(O|\hat{S}) + \log P(\hat{S}) \tag{2.4}$$

Being able to factor the search problem in this way considerably simplifies the task of speech recognition. This is partly due to the fact that the speech signal is continuous while the text is discrete, making it difficult to model the distribution $P(S|O)$. In Section 2.5, we will briefly review the standard techniques for modeling $P(O|S)$ and $P(S)$.

2.3 Speech Recognition as Data Compression

An alternative view of the speech recognition problem is that instead of a decoding task, it constitutes a *lossy data compression* or *source coding* task. Here, we view speech as simply a highly redundant encoding of the original message. The goal of speech recognition in this view is to find a compact representation of the speech which jointly minimizes the *entropy rate* of the output, and the *distortion* between the original message and the output, as measured by some distortion function.

If we choose a sequence of words as the representation, and the negative acoustic log-likelihood of the hypothesis as the distortion function, then this is actually very similar to the standard MAP decoding technique described in Equation 2.4:

$$S = \arg\min_{\hat{S}} H(\hat{S}) + D_{KL}(O|\hat{S}) \tag{2.5}$$

$$= \arg\min_{\hat{S}} -E[\log P(\hat{S})] - \log P(O|\hat{S}) \tag{2.6}$$

$$= \arg\max_{\hat{S}} \log P(O|\hat{S}) + E[\log P(\hat{S})] \tag{2.7}$$

However, one reason to take this alternate view of speech recognition is that it allows us to consider the output of a speech recognizer as something other than simply a sequence of words. Namely, we view the output of a speech recognizer as a *hypothesis space*, which consists of a single hypothesis in the limit. The hypothesis space is a representation of the *uncertainty* remaining in the output of the speech recognizer. In multiple-pass decoding algorithms, described further in 2.8, the hypothesis space from one pass of recognition becomes the effective search space $\mathbb{S}'$ for a subsequent pass. This is a very useful heuristic for search, and allows the use of more exact decoding algorithms and models than would otherwise be possible. The problem with this approach is that it can result in the propagation of errors if the first-pass heuristic is too restrictive, or if the models used for first-pass recognition result in the hypothesis space being impoverished in some way.

2.4 Probability and Information Theory

Information theory [Cover & Thomas, 2006], invented in the 1940s by Claude Shannon, is a way of answering two fundamental questions regarding communication systems:

- What is minimum amount of information required in order to represent a message? In other words, what is the limit of *data compression*)?
- What is the maximum amount of information that can be sent through a channel? In other words, what is the limit of *data transmission*?

To answer these questions, we first need to give definitions of some of the terms involved, which will continue to be used throughout this thesis. Recall from Section 2.2 that there exists a *source*, or a sender, which emits an utterance, which is a random variable S drawn according to some probability distribution $P(S)$. A diagram of the communication process is shown in Figure 2.1.

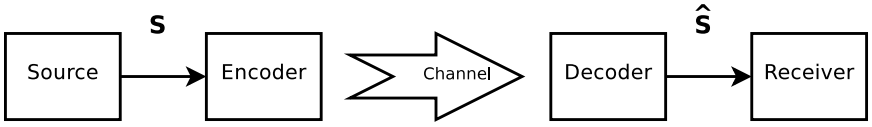

Figure 2.1: Data transmission in the Noisy Channel Paradigm

We define information as a *reduction in uncertainty* on the part of the receiver. The minimal amount of information required to represent a message is exactly the amount of uncertainty contained in that message. The answer to question (1), therefore, can be defined with reference to a particular random variable S as the expected amount of uncertainty which exists in S.

2.4.1 Entropy

Since S has a probability distribution $P(S)$, we can easily quantify the uncertainty associated with some value of S as being the inverse of the probability of that value. Intuitively, we believe that the receiver is more surprised by low-probability values of S. Because in real-world situations, information is additive, we use the logarithm of the inverse. The expectation of this quantity is a measure of the uncertainty of S known as the *entropy* of S. For a discrete random variable S, it is defined as:

$$H(S) = E_{P(S)}\left[\log \frac{1}{P(S)}\right] = -\sum_{s \in S} P(s) \log P(s) \tag{2.8}$$

In practice, we may have no way to know the actual probability distribution $P(S)$, and must instead approximate it with a statistical model $Q(S)$. Therefore, we are also unable to calculate the true entropy of S, since it requires knowledge of this distribution. However, given a sufficient number of samples of S, we can compute the *cross-entropy* $CH(Q;S)$ between the model and the *empirical distribution* of S, which we denote as $\tilde{P}(S)$. This is the expectation of the estimated information contained in S according

to $Q(S)$, taken under the empirical distribution $\tilde{P}(S)$. It can be shown, first, that $\tilde{P}(S)$ converges to $P(S)$, and second, that $CH(Q;S) \geq H(S)$, with equality if and only if $\tilde{P}(S) = Q(S)$.

$$CH(Q;S) = -\frac{1}{N}\sum_{i=1}^{N} \log Q(S_i) = E_{\tilde{P}(S)}\left[\log \frac{1}{Q(S)}\right] \tag{2.9}$$

In speech recognition, it is common to speak of the *perplexity* of a model on a data set, which is simply the exponential of the cross-entropy between that data set and a particular model. For N-Gram models and word lattices, the perplexity has a useful interpretation, namely the average branching factor of the lattice or the intersection of the input with the model.

$$pplx(Q;S) = \exp -\frac{1}{N}\sum_{i=1}^{N} \log Q(S_i) \tag{2.10}$$

2.4.2 Channel Capacity

To answer question (2), namely, what is the limit of data transmission, we define the channel as a random function which maps an input code X to an output code Y according to some probability distribution $P(Y|X)$. Note that the problem of *channel coding* is usually treated separately from the problem of *source coding*, or in other words, the theoretical limit of data transmission is not affected by any compression or coding which may have previously been applied to the data. We can define the *capacity* of the channel as the maximum amount of information which can be transmitted through the channel with an arbitrarily low probability of error.

This quantity is defined, again, in terms of a reduction in uncertainty. What we first wish to quantify is the amount of uncertainty that exists in Y when we know X, which informally is a measure of how much confusion between different values of X is created by the channel. This is known as the *average conditional entropy* $H(Y|X)$ and is defined, for discrete X and Y, as:

$$H(X|Y) = \sum_{y\in\mathcal{Y}} P(y)H(X|y) \tag{2.11}$$

$$= \sum_{y\in\mathcal{Y}} -P(y)\sum_{x\in X} P(x|y)\log P(x|y) \tag{2.12}$$

$$= \sum_{x,y\in\mathcal{X},\mathcal{Y}} -P(x,y)\log P(x|y) \tag{2.13}$$

$$= E_{P(X,Y)} \log \frac{1}{P(X|Y)} \tag{2.14}$$

Now, having calculated the amount of uncertainty in the input to the channel X, namely $H(X)$, and the amount of uncertainty we have about the original message X after seeing the channel output Y, namely

$H(X|Y)$, we can calculate a very useful and important quantity known as the *average mutual information* between X and Y. This is defined quite simply as:

$$I(X;Y) = H(X) - H(X|Y) \tag{2.15}$$

Continuing to speak in terms of uncertainty, $I(X;Y)$ is the *reduction in uncertainty* about X that we experience upon receiving the channel output Y. Clearly, if the channel is perfect and $Y = X$, then observing Y removes all uncertainty about X and thus $I(X;Y) = I(X;X) = H(X)$.

Alternately, we can also say that $I(X;Y)$ is how much information Y gives us about X. It is this definition that leads to the *channel coding theorem*, which states that the theoretical capacity of the channel can be computed as the maximum mutual information $I(X;Y)$, under all possible choices of the distribution $P(X)$:

$$C = \max_{P(X)} I(X;Y) \tag{2.16}$$

The reason for the maximization is that in channel coding problems, the true distribution $P(X)$ is not known, whereas the distribution $P(Y|X)$ is a fixed property of the channel.

In the noisy-channel approach to automatic speech recognition, despite the block diagram shown in Figure 2.1, we do not make a clear separation between the coding of the original message and the effects of the channel. Instead, we construct *acoustic models* [Brown, 1987], which are probabilistic models of $P(O|S)$, that is, of the acoustic realization of the linguistic message, including any channel noise. The concept of mutual information is used extensively in acoustic modeling in the framework of *maximum mutual information estimation* [Bahl et al., 1986], where the goal is to find the acoustic model which maximizes the mutual information between the source text and the acoustic observation. This is, more or less, equivalent to maximizing the channel capacity, although the maximization is carried out over the parameters of the channel model rather than the source model. Acoustic modeling is discussed in more detail in Section 2.5.1.

2.4.3 Rate-Distortion Theory

Entropy defines a lower bound on *lossless* compression, that is, it defines the minimum length of a code with the same informational content as the original message. Likewise, channel capacity defines the maximum length of a code that can be sent over a channel with an arbitrarily small probability of error. However, it is possible to achieve higher rates of compression, or conversely, higher data rates, by accepting some amount of *distortion* in the reconstructed message. According to the *rate-distortion theorem*, for a stationary ergodic source and a distortion function D, there exists a function $R(D)$ which establishes the lowest achievable entropy rate for any given distortion. Conversely, there also exists a function $D(R)$, which establishes the lowest achievable distortion for a given entropy rate.

The theory of rate-distortion allows us to establish a lower bound on data compression given a *distortion measure* and some acceptable level of distortion.[2]

[2]Equivalently, it allows us to establish a lower bound on distortion for a given rate of compression, and this is how it is most frequently expressed in data compression literature.

2.5 Statistical Modeling for ASR

The basic equation for decoding in automatic speech recognition, as shown in Equation 2.17, results in the hypothesis S_{MAP} with the highest *a posteriori* probability, or alternatively, the lowest probability of error.

$$S_{MAP} = \arg\max_{\hat{S}} P(O|\hat{S})P(\hat{S}) \tag{2.17}$$

However, this hypothesis is optimal under the assumption that we know the probability distributions $P(O|S)$ and $P(S)$, which are, respectively, the likelihood of acoustic realizations given a word sequence, and the prior probability of a word sequence. In practice, these distributions are not known, and thus we must rely on *statistical estimation* to build *models* of them from observed linguistic data. These models are known, respectively, as the *acoustic model* and the *language model*.

2.5.1 Acoustic Modeling

Virtually all contemporary ASR systems use *Hidden Markov Models* (hereafter referred to as HMMs) as a statistical model for the speech generation process. HMMs are a mathematical formalism for pattern recognition, first described in [Baum & Petrie, 1966], which have proven to be extremely useful in modeling speech, both for recognition [Rabiner, 1989] and for synthesis [Masuko et al., 1996]. As with all generative models of speech production (e.g. the source-filter model), an HMM is a highly simplified and abstract version of a very complicated process. However, in practice, it captures enough of the structure of speech to be useful. Estimation and inference on HMMs is also computationally efficient.

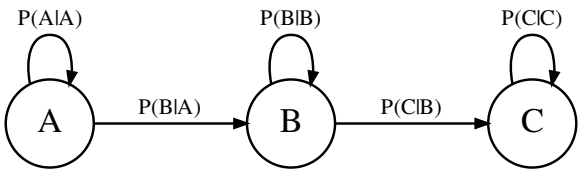

Figure 2.2: First-order Markov Chain with three states A, B, and C

HMMs were originally, and perhaps more precisely, referred to as *probabilistic functions of Markov chains*. We say that a sequence of discrete random variables $X_1, X_2, X_3, ..., X_N$ forms a *first-order Markov chain* if the distribution of X_k for any k is conditionally independent of all other variables in the sequence given the preceding element X_{k-1}. The values which the random variables X_k take on are known as the *states* of the Markov chain. A first-order Markov chain can be described compactly as a vector of initial probabilities $P(x_i)$ and a matrix of transition probabilities $P(x_i|x_j)$. Higher-order Markov chains are also possible, in which case the conditional independence assumption is made given the previous M symbols. In this case, the matrix of transition probabilities contains N^{M+1} parameters.

Markov chains are useful in modeling the distribution of discrete sequences where successive symbols are not statistically independent. For non-trivial examples such as human language, it is generally the case that each symbol depends on more than simply the identity of the previous symbol, but in practice, the benefits of this simplifying assumption of independence in terms of computational and statistical efficiency outweigh the possibility of modeling errors.

The use of Markov chains for modeling the speech signal is motivated by the fact that speech is *quasi-stationary* in nature. That is, the acoustic properties of the signal are stable over short time periods. We can therefore view these stable periods as states in a Markov process. However, there are two major problems in attempting to model speech using a simple Markov chain. The first and most obvious is that the speech signal is continuous rather than discrete. The second is that the acoustic realization of these states is itself highly variable, even within the same utterance by the same speaker.

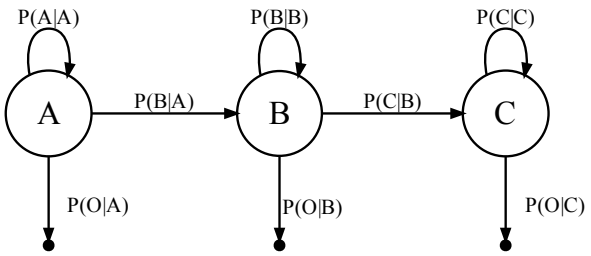

Figure 2.3: First-order Hidden Markov Model with three hidden states A, B, and C and corresponding output distributions

The first problem can be solved using *vector quantization* (VQ), which maps the continuous units of acoustic space to discrete symbols. However, in order to minimize quantization error, it is necessary to use many more VQ codewords than there are states in the underlying Markov model. In addition, since vector quantization is a form of unsupervised learning, the individual codewords are generally not *identifiable*, that is, they do not correspond directly to any prior classification of the acoustic space, such as the set of model states.

Therefore, we formally treat the *observation* of a state as a random variable whose distribution is conditional on the identity of the state. In order to simplify the resulting model, we make a further independence assumption, known as the *output independence assumption*, which states that the distribution of an observation is conditionally independent of all other observations in the sequence given the identity of the underlying state. A graphical representation of a three-state HMM is shown in Figure 2.3.

Although the independence assumptions inherent in HMMs can be viewed as flaws in the model, their main effect is that HMM based acoustic models tend to exaggerate the dynamic range of the conditional probability $P(O|S)$. This is particularly acute when continuous density functions are used to model the observation of states, as is commonly done for automatic speech recognition. Because of this, the results of evaluating an acoustic model are generally thought of as *scores* rather than *probabilities* as such, and a

variety of empirically tuned heuristics are used to regularize them.

There are in fact several ways in which the acoustic model contributes to modeling error. The first and most obvious is a mismatch between the acoustic conditions in which the model was trained, and those in which it is used for recognition. The second arises from inadequate amounts of training data, such that the parameters of the output density functions for the states of the model are not robustly estimated. To mitigate this problem, essentially all speech recognition systems employ some form of *parameter tying*, where equivalence classes are constructed among the various acoustic units in the system, and these classes share some or all HMM parameters.

HMM Algorithms

Traditionally, HMMs are discussed through the "three problems" of *evaluation*, *decoding*, and *estimation* (or training). The first two problems are special cases of inference on the model, while the third is the learning task. For all of these problems there are corresponding dynamic programming algorithms which have quadratic time complexity in the number of states.

The problem of search in large-vocabulary continuous speech recognition is dealt with in greater detail in Section 2.7. Typically, ASR systems use separate HMMs to model individual words or sub-word units, and then employ a two-level search strategy to generate sentence-level hypotheses. For the purposes of this section, the most important problems related to HMMs are:

1. How can we evaluate a set of HMMs relative to each other for a given acoustic observation?

2. How can we choose HMM parameters to give the best fit to a given acoustic observation?

The first problem corresponds to the evaluation problem, though in practice it is treated as a decoding problem, for reasons that will soon become clear. The evaluation problem consists of calculating the probability of a particular observation O given an HMM λ. Although the state sequence corresponding to the observation is not known, it is possible to calculate the joint probability of the observation and the state sequence for any given state sequence. This is defined simply in terms of the parameters of the HMM:

$$P(O, S|\lambda) = \prod_{i=1}^{N} P(s_i|s_{i-1})P(o_i|s_i) \tag{2.18}$$

Therefore, the probability of the observation can be calculated by marginalizing this joint probability distribution over all possible state sequences:

$$P(O|\lambda) = \sum_{\forall S} P(O, S|\lambda) \tag{2.19}$$

The number of state sequences is quite large. For an HMM of K states and an observation of N points, there are a maximum of K^N possible state sequences. However, the Markov assumptions in the form of the model lead to a convenient recursive definition of $P(O|\lambda)$ in terms of partial observation sequences:

$$P(O_1^j|\lambda) = \sum_{i=1}^{K} P(O_1^j, s_i|\lambda) \tag{2.20}$$

$$P(O_1^j, s_i|\lambda) = \sum_{allS:S_j=s_i} P(O_1^j, S|\lambda) \tag{2.21}$$

$$= P(o_j|s_i) \sum_{k=1}^{K} P(s_i|s_k) P(O_1^{j-1}, s_k|\lambda) \tag{2.22}$$

Therefore, in order to compute $P(O|\lambda) = P(O_1^N|\lambda)$, it suffices to calculate all of the terms $P(O_1^j, s_i|\lambda)$, of which there are NK, and each of which depends only on the model parameters and the K values $P(O_1^{j-1}, s_k)$ computed at the previous timepoint. This is known as the *forward algorithm*, though in fact it is simply a particular instance of the generic single-source shortest-path algorithm over directed graphs.

If the summation in the forward algorithm is replaced with maximization, we obtain the *Viterbi algorithm*, which calculates the joint probability of the observation and the *single best* state sequence. It is also possible to store the maximum argument along with the probability, which gives an optimal *alignment* of states to observations. Such an alignment is useful because, as mentioned previously, the exact state sequence corresponding to any observation sequence is not known. This creates particular problems in attempting to estimate the parameters of the HMM. If the state sequences of the training data were known, it would be trivial to estimate the observation and transition probabilities using maximum likelihood estimation:

$$P_{ML}(o_j|s_i) = \frac{C(o_j, s_i)}{C(s_i)} \tag{2.23}$$

$$P_{ML}(s_j|s_i) = \frac{C(s_i, s_j)}{C(s_i)} \tag{2.24}$$

The ability to find the most probable alignment given the current parameters suggests an iterative method for estimating the HMM parameters:

1. Start with some reasonable initial values of the parameters.
2. Find the most probable alignments for the training data.
3. Estimate new parameters using Equations 2.23 and 2.24.
4. Go to step 2, unless the parameters have converged.

In fact, this is a commonly used method for training HMM parameters, known as the *segmental k-means* algorithm. It has the advantage of being quite fast. What it lacks is a guarantee that it will converge to anything useful, or in fact that it will converge at all.

The problem of training HMM parameters can be solved more formally as an instance of *expectation-maximization* [Dempster et al., 1977]. In fact, the Baum-Welch algorithm [Baum & Petrie, 1966] for training HMMs predates the general Expectation-Maximization algorithm. This is a method for estimating parameters of a model using maximum likelihood where some of the data are missing. In this case, the missing

data is the state sequence, or the alignment of state identities to timepoints. First, we define the *full-data log-likelihood function* for the model:

$$\ell(O, S|\lambda) = \sum_{i=1}^{N} \log P(s_i|s_{i-1}) + \sum_{i=1}^{N} \log P(o_i|s_i) \tag{2.25}$$

While this is a very nice-looking function, it is not possible to maximize it, since the state identities s_i are not known. Instead, we create an *auxiliary function*, which is simply the *expectation* of this log-likelihood under the distribution over the missing data (i.e. the state sequence) defined by the current model parameters:

$$Q(\lambda, \bar{\lambda}) = E_{P(S|O,\lambda)}\ell(O, S|\bar{\lambda}) = \sum_{allS} P(S|O, \lambda)\ell(O, S|\bar{\lambda}) \tag{2.26}$$

It can be proven that maximization of this function (which is a fully-defined function of the observed data and the current model parameters) also maximizes the complete-data likelihood function. By taking the gradient of $Q(\lambda, \bar{\lambda})$, adding Lagrange multipliers to ensure that the sum-to-one constraints are satisfied, and equating this to zero, we obtain the following re-estimation formulae for the transition and observation probabilities:

$$\hat{P}(s_i|s_j) = \frac{\sum_{t=2}^{T} P(S_t = s_i, S_{t-1} = s_j|O, \lambda)}{\sum_{t=1}^{T} P(S_t = s_j|O, \lambda)} \tag{2.27}$$

$$\hat{P}(o_k|s_i) = \frac{\sum_{t:O_t=o_k} P(S_t = s_i|O, \lambda)}{\sum_{t=1}^{T} P(S_t = s_i|O, \lambda)} \tag{2.28}$$

In fact, these are simply the *expectations* of the maximum-likelihood update equations in Equations 2.23 and 2.24, taken under the posterior distribution over state alignments. The probabilities $P(S_t = s_j|O, \lambda)$ and $P(S_t = s_i, S_{t-1} = s_j|O, \lambda)$ can be computed efficiently using the forward algorithm and the related *backward algorithm* [Rabiner, 1989].

Continuous-Density HMMs

Because of the inherent error associated with vector quantization, it is common to use a continuous probability distribution to model the observation probabilities of an HMM. In practice, this is nearly always a mixture of Gaussian distributions with diagonal covariance matrices (though in some cases, a mixture of Laplacian distributions may be used for computational efficiency):

$$\ell(O, S|\lambda) = \sum_{i=1}^{N} \log P(s_i|s_{i-1}) + \sum_{i=1}^{N} \log \sum_{k=1}^{K} w_{ik} \frac{1}{\sqrt{(2\pi)^D|\Sigma_{ik}|}} \exp \sum_{d=1}^{D} \frac{-(o_d - \mu_{ikd})^2}{2\sigma_{ikd}^2} \tag{2.29}$$

For a continuous-density HMM, the set of parameters consists of the transition probabilities $P(s_i|s_j)$, the Gaussian means and variances, and the set of mixture weights for each state's output distribution.

State and Mixture Tying

In large-vocabulary continuous speech recognition, it is common practice to use *context-dependent subword units* as the basic units of recognition. These are typically phoneme-like units, consisting of a single phoneme in a particular phonetic context. Most frequently, the context is defined by the identities of the phonemes immediately preceding and following, in order to model coarticulatory effects. This unit is known as a *triphone*.

In training, a separate HMM is created for each such unit, and the resulting HMMs are concatenated according to a *pronunciation model* to form the models for each input sentence. The standard Baum-Welch re-estimation algorithms are then applied, with the parameters of the subword models being updated according to the results of evaluating the concatenated sentence HMM. Likewise, for recognition, word models are created by concatenating subword models according to a pronunciation model, and these word models are then composed into a decoding network, as described in Section 2.7.

Using subword models allows the system to recognize words which were not present in the training set. It also allows parts of words with the same acoustic properties to share the same acoustic models, leading to more efficient use of the training data. However, when context-dependent models are used in conjunction with continuous output distributions, the number of such models, and hence the number of Gaussian parameters, becomes extremely large. This results in intractably large model files as well as data sparsity problems in training. For this reason, it is necessary to share sets of parameters between multiple models, which is known as *parameter tying*.

The two most common forms of parameter tying are *state tying* and *mixture tying*. In the former, the state output distributions of all triphones are mapped to a set of acoustic clusters, usually known as *senones* [Hwang, 1993]. Each senone consists of a complete mixture distribution with its own set of Gaussian parameters. Due to the requirement that senones map to a unique base phone, it is often the case that these clusters may correspond to unequal amounts of training data, and therefore it is common to use variable numbers of Gaussians per senone.

Even using state tying, it is still possible to encounter data sparsity problems, or alternately, the number of parameters may still be too large to allow efficient computation of state output densities. In this case, tying can also be applied at the mixture level. In the simplest scheme, known as *semi-continuous* models [Huang, 1989], all senones share a single *codebook* of Gaussian density parameters, and only the mixture weights are specific to each senone. Other configurations are possible, such as *phonetically-tied mixtures*, where each base phoneme has a separate codebook, or *genones*, where mixture components are tied based on bottom-up clustering over states [Digalakis et al., 1996].

2.5.2 Language Modeling

When the input to speech recognition is continuous speech, rather than isolated words, it becomes useful to have a *language model* as well as an acoustic model. While the acoustic model is the component of a speech recognizer which determines how closely a part of a spoken utterance matches a word or sequence of phones, the language model is the component which determines how likely a word or sentence is to have been spoken in the first place. The most obvious reason why this is necessary is that many words or sequences of words sound very similar, and it is not possible to decide among them without some *prior* knowledge of which ones are admissible or likely for a given language or domain. Mathematically, the language model is a statistical model of the probability distribution $P(S)$ over word sequences. As with

HMMs in acoustic modeling, virtually all modern ASR systems use history-based or *N-Gram* models for language modeling.

Another justification for the language model arises when we view recognition in terms of *Bayesian Decision Theory*. In this framework, speech recognition is the process of selecting between a (potentially infinite) set of alternate transcriptions for a sentence. Equivalently, we can think of this as a *classification* of the acoustic observation, where each class contains all the realizations of a particular sentence.

We will denote the set of transcriptions as $\mathbb{S}$, and an acoustic observation as O. We wish to select the transcription S which is optimal according to some *decision rule*, which we can represent as a function $R(S, O)$ mapping a hypothesis to a real value. Equivalently, we can view this as minimizing a *loss function* $L(S, O)$.

$$S = \arg\max_{S \in \mathbb{S}} R(S, O) \tag{2.30}$$

$$S = \arg\min_{S \in \mathbb{S}} L(S, O) \tag{2.31}$$

Obviously, we would like to select a decision rule which minimizes the probability of selecting the wrong transcription. The optimal rule in this case, which corresponds to the *Bayes Optimal Classifier*, is the MAP, or *maximum a posteriori* decision rule, which corresponds to the function R_{MAP}, which is simply the *posterior probability* of a hypothesis. Therefore, to find the transcription with the minimum probability of error, we find the hypothesis S_{MAP} which maximizes the posterior probability:

$$R_{MAP} = P(S|O) \tag{2.32}$$

$$S_{MAP} = \arg\max_{S \in \mathbb{S}} P(S|O) \tag{2.33}$$

The difficulty in actually finding S_{MAP} is twofold: we need to be able to search the set $\mathbb{S}$ in some efficient manner, and for each element S of this set, we need to be able to calculate $P(S|O)$. This implies the existence of a conditional probability distribution $P_{\mathbb{S}}(S|O)$ over the set of sentences $\mathbb{S}$. Modeling this distribution directly is intractable. However, we can factor it using Bayes' Rule into three separate distributions:

$$P_{\mathbb{S}}(S|O) = \frac{P_{\mathbb{O}}(O|S)P_{\mathbb{S}}(S)}{P_{\mathbb{O}}(O)} \tag{2.34}$$

And, more importantly, since the observation O is fixed during the process of recognition, we can eliminate the troublesome term $P_{\mathbb{O}}(O)$ from the denominator, resulting in a decision rule based on the *likelihood* of an observation and the *prior* probability of a sentence:

$$S_{MAP} = \arg\max_{S \in \mathbb{S}} P(O|S)P(S) \tag{2.35}$$

If a sentence is represented as a sequence of underlying HMM states, then the distribution $P_{\mathbb{O}}(O|S)$ can be directly modeled using an HMM, where the "model" variable in the HMM equations λ is equal to S. The

likelihood $P(O|S)$ can then be computed directly as the sum of the final α variables, or approximated using the final Viterbi path score.

This leaves us the task of estimating the distribution $P_{\mathbb{S}}(S)$. The most common method of doing this is to use a history-based or *N-Gram* model. First, we define a sentence S as a sequence of N words, denoted w_i, and we use the notation w_i^j to represent the sequence of words from index i to j:

$$S = (w_1, w_2, ..., w_N) = w_1^N$$

The probability of the sentence is taken to be the joint probability of the sequence as a whole, which is intractable to model directly due to the extremely large number of possible sequences. However, the definition of conditional probability allows us to factor the joint probability of a sequence into a product of conditional probabilities:

$$P(A, B) = P(A|B)P(B) \tag{2.36}$$

$$P(A, B|C) = P(A|B, C)P(B|C) \tag{2.37}$$

$$P(A, B, C) = P(A, B|C)P(C) = P(A|B, C)P(B|C)P(C) \tag{2.38}$$

Using this chain rule, the probability of a sentence can be factored into the product of the conditional probability of each word given the sequence of all preceding words, which we refer to as the *history*, denoted as h_i:

$$\begin{aligned} P(S) &= P(w_1, w_2, ..., w_N) \\ &= P(w_N, w_{n-1}, ..., w_1) \\ &= P(w_N|w_1^{n-1})P(w_{n-1}|w_1^{n-2})...P(w_1) \\ &= P(w_N|h_N)P(w_{n-1}|h_{n-1})...P(w_1) \end{aligned} \tag{2.39}$$

To model these conditional probabilities efficiently, we use the observation that the mutual information between a word and any predecessor word tends to decay with distance. Therefore, we can consider all histories ending with the same m words to be part of an equivalence class. This is equivalent to making an assumption of conditional independence of the kind used in other Markov chains (and, as we saw earlier, in Hidden Markov Models). Since the word and history taken together form a sequence of N words, we refer to a history-based model as an *N-Gram* model, though in fact, it is simply an $N - 1$ order Markov chain over word sequences.

Because the number of words in a natural language is still extremely large, we encounter a number of unique problems in estimating the conditional probabilities $P(w|h)$ which make up an N-Gram model, and these are potential sources of modeling error. The most serious problem is that for any corpus of text, the majority of the words in the vocabulary may only occur a few times, and the overwhelming majority of N-Grams, that is, word sequences of length N, will never be observed. If maximum likelihood estimation is used, the variance of the resulting probability estimates will be quite large, and the model will also assign zero probability to many plausible word sequences. A wide variety of *smoothing* techniques have been proposed to mitigate this problem. The most well-known and widely used ones are *Katz smoothing* [Katz, 1987] and *Modified Kneser-Ney smoothing* [Chen & Goodman, 1998].

As the size of the basic vocabulary grows, N-Gram models encounter two issues with consequences for speech recognition. The first is that the number of possible N-Grams grows polynomially in the size of the vocabulary. As the number of N-Grams increases, so does the number of parameters to be trained, as well as the storage and memory space needed to store them. The second is that a larger vocabulary tends to increase the *perplexity* of the model, which has an adverse effect on the speed and accuracy of the recognizer, since it increases the size of the search space. For these reasons it may be preferable to limit the size of the vocabulary in earlier stages of recognition.

2.6 Acoustic Feature Calculation

Until this point we have been purposefully vague about exactly what the "observations" used in speech recognition consist of. This is because the exact form of these observations is largely irrelevant to the main problem of recognition - we simply assume that they are continuous in nature and can be effectively modeled with multivariate Gaussian Mixture Models. For this reason, it is possible to adapt a speech recognition system such as SPHINX to other language and sequence recognition tasks based on noisy-channel models, such as handwriting recognition or optical character recognition.

The purpose of acoustic feature calculation is to transform the speech signal into a numerical representation appropriate for pattern recognition. The input signal is nearly always represented in a sampled and quantized time-domain form, using pulse code modulation or some variant thereof. This representation is unsuitable for automatic speech recognition for several reasons. The most important is that the human perceptual system distinguishes between classes of speech sounds primarily through coarse distinctions in frequency domain, and therefore a spectral representation is more useful. Also, the raw speech signal is non-stationary and therefore cannot be meaningfully analyzed as a whole. Therefore, it must be segmented into smaller units.

The approach used by most current systems to deal with the non-stationarity of speech is to segment the audio into evenly spaced frames. The frame spacing used is usually 10 milliseconds, which is short enough to capture the rapid transitions present in some speech sounds. The choice of 10 millisecond shifts is arbitrary and represents a compromise between achieving sufficient time-domain resolution and avoiding redundant computation. Variable frame rates have been tried with some success, particularly in noisy conditions [Zhu & Alwan, 2000]. The length of each frame is chosen so as to give sufficient resolution in frequency domain, and is typically about 25 milliseconds.

After segmenting the audio, the individual frames are analyzed to produce a suitable feature vector for classification. The goal of this analysis is to produce features which contain only the information relevant for classifying speech sounds, and which are robust to variations in the recording channel, environmental noise, and speaker variability.

The most popular feature parametrizations are based on the *source-filter model* of speech. The variety of speech sounds present in natural speech is produced by the through the complex interaction of vibration of the glottis, turbulence in the mouth, and resonances generated by reshaping the vocal tract through the motion of various articulators. In the source-filter model, we simplify this drastically into two components. The *source* represents either the vibration of the glottis for voiced sounds, or a non-specific noise source for unvoiced sounds. The *filter* represents the frequency response of the vocal tract. A schematic of this model is shown in Figure 2.4.

Given this model, feature extraction for ASR typically attempts to model the filter, or the coarse spectral

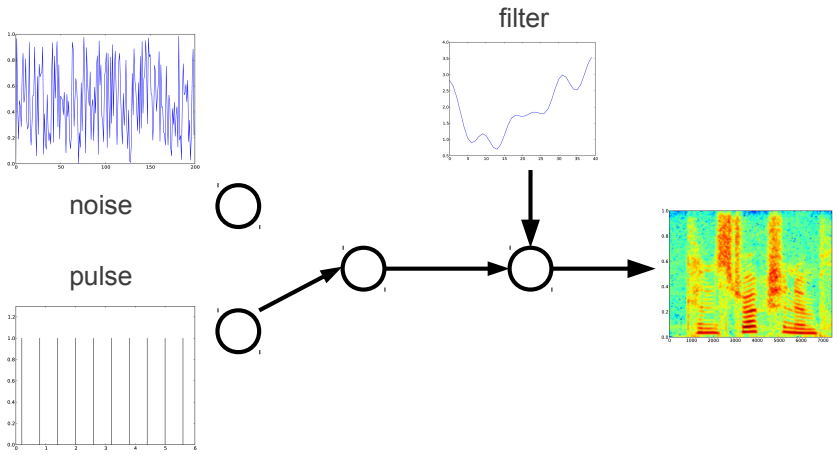

Figure 2.4: Source-Filter Model of speech production, with a noise-pulse switch as the source

characteristics of the speech, while removing the effects of the source. The two most popular ways of doing this are *linear prediction* and *cepstral analysis*.

2.6.1 Linear Prediction

Linear predictive analysis (referred to as LPC from here on) is based on the idea that each speech sample can be predicted as a linear combination of the preceding samples. Specifically, for a discrete-time signal $x[t]$, we can describe each sample as a sum of the error signal $e[t]$ and a weighted sum of the N preceding samples multiplied by the *linear prediction coefficients* a_i.

$$x[t] = e[t] + \hat{x}[t] \tag{2.40}$$

$$= e[t] + \sum_{i=1}^{N} a_i x[t-i] \tag{2.41}$$

The goal of LPC analysis is to find the values of a_i which minimize the mean squared error of the predicted signal $\hat{x}[t]$ compared to the original signal $x[t]$:

$$\hat{a} = \arg\min_{a} \sum_{t} (x[t] - \hat{x}[t])^2 \tag{2.42}$$

By taking the derivative and equating it to zero, we obtain the *Yule-Walker equations*, a set of linear equations which can be solved to find the optimal values of a. Although it is possible to solve these equations using Gaussian elimination, they have a particular form which enables more efficient methods. Specifically, the left-hand side of the equations is a matrix consisting of values of the *discrete-time autocorrelation*

function. Depending on the particular definition of this function, the matrix may be either symmetric or *Toeplitz*. In the latter case, there is a particularly efficient algorithm for solving for a known as the *Levinson-Durbin* recursion.

LPC analysis is useful for speech recognition, because the error signal $e[t]$ and the prediction coefficients a correspond directly to the source and filter in the source-filter model of speech. In LPC, the filter has the particular form of an all-pole model, though there are other ways of modeling the filter, as discussed in Section 2.6.2. In current systems, LPC is typically used as part of the PLP [Hermansky, 1990] and RASTA [Hermansky & Morgan, 1994] algorithms.

2.6.2 Cepstral Analysis

Another commonly used technique for feature extraction, and the one used by the system described in this thesis, is *cepstral* analysis. This form of analysis is based on the principle of *homomorphic filtering*. Starting from the source-filter model, we assume that the speech symbol results from the convolution of the glottal or noise source $e[t]$ with the filter $h[t]$:

$$x[t] = e[t] * h[t] = \sum_{i=-\inf}^{\inf} e[t]h[t-i] \tag{2.43}$$

As in LPC analysis, the filter here is assumed to be a linear time-invariant system, though we make no assumptions about the particular form of this system. Taking the z-transform of this converts the convolution to a multiplication:

$$X(z) = E(z)H(z) \tag{2.44}$$

Now, by applying the logarithm operation to both sides, the source and filter become linearly separable in both the frequency and time domain:

$$\log X(z) = \log E(z) + \log H(z) \tag{2.45}$$

$$\tilde{x}[t] = \tilde{e}[t] + \tilde{h}[t] \tag{2.46}$$

The output of the inverse transform $\tilde{x}[t]$ is known as the *cepstrum*, a play on words from the term *spectrum*. The "time" indices in the cepstrum are likewise referred to as *quefrencies*. The cepstrum has several interesting properties. The one which is most relevant to the feature extraction problem is that components of the signal which are periodic in time domain, and therefore discrete impulses in frequency domain, are also represented as impulses in cepstral domain, at time indices which correspond to their period in time domain. By contrast, the "filter", or the coarse spectrum, is represented by the low-quefrency indices in the cepstrum. Therefore, source-filter separation can be achieved by discarding the high-quefrency indices of the cepstrum. This is also known as *cepstral smoothing*.

Another useful property of the cepstrum for speech recognition is that all convolutional effects in time domain, such as the convolution of source and filter, become linear in the cepstral domain. This gives

cepstral coefficients robustness to channel variability. Assuming the channel can be modeled as a linear system, its effect on the signal in time domain can be expressed as a convolution of the "pure" signal with a channel-specific "noise" signal. Therefore, to remove the effect of the channel, we can simply subtract out the noise cepstrum. This noise cepstrum can be estimated as the long-term average of the cepstrum, a technique known as *cepstral mean subtraction*. If the long-term cepstrum is not available or does not produce a reliable estimate, local methods can be used, such as *codeword-dependent cepstral normalization* [Acero, 1990].

In current systems, cepstral analysis is usually done as part of the computation of *mel-frequency cepstral coefficients* (MFCC), which are the cepstrum computed over a weighted and non-linearly downsampled version of the short-term spectrum.

2.6.3 Discriminative Feature Transforms

While LPC and cepstral features are optimal with respect to the source-filter model, this model is only loosely related to the problem of distinguishing between classes of speech sounds. It has therefore become popular to apply linear transformations to these features in order to decorrelate the individual components of the feature vector and to maximize the separability of acoustic classes in vector space. These transforms are generally *dimensionality reducing*, in that they consist of basis vectors which are ordered in some way such that the lower order ones can be omitted, resulting in a more compact output feature vector. This also allows the use of high (sometimes very high) dimensionality input features, incorporating a variety of information sources.

For example, in order to better capture the time-varying properties of the speech signal, the LPC or cepstral vector is nearly always augmented with *dynamic features*, which are time derivatives of the feature stream. The most commonly used are simple delta and delta-delta features, which are calculated given a sequence of feature vectors $..., \vec{x}_{t-2}, \vec{x}_{t-1}, \vec{x}_t, \vec{x}_{t+1}, \vec{x}_{t+2}, ...$ around time point t as:

$$\Delta\vec{x}_t = \vec{x}_{t+2} - x_{t-2} \tag{2.47}$$

$$\Delta\Delta\vec{x}_t = \Delta\vec{x}_{t+2} - \Delta\vec{x}_{t-2} \tag{2.48}$$

These are then concatenated into a single feature vector with the initial feature vector for time t, and the resulting vectors are used as the input to training and recognition. This is, however, a simplistic model of time dependencies in the feature stream, and therefore given the ability to compute a dimensionality reducing transform over a long input vector, we can instead simply concatenate a large window of frames into a single supervector and compute a transform which extracts the most "important" subspaces of this input feature space. Also, since the discrete cosine transform used to compute MFCCs is itself a linear transform with a basis consisting of cosines, dimensionality reducing transforms are frequently computed on the mel-spectral coefficients rather than on the MFCCs. This has the advantage of potentially finding better bases than cosines, but the disadvantage of requiring more computation due to the typically greater dimensionality of the mel spectrum.

The simplest such transform is the Karhunen-Loève Transform, also known as Principal Components Analysis (PCA). The transform produced by PCA rotates the feature space such that the output dimensions are uncorrelated and ordered by variance. These basis vectors are simply the eigenvectors of the covariance matrix of the input, ordered by their eigenvalues. PCA is commonly used in visualization applications to

provide a two or three-dimensional projection of higher-dimensional data. While PCA is a dimensionality reducing transformation it is not discriminative.

Linear Discriminant Analysis (LDA), also known as Fisher's Linear Discriminant, attempts to find a transformation which maximizes not the variance of each dimension but instead the ratio between the *between-class* and *within-class* covariances. In other words, it attempts to project them onto a subspace where the classes are maximally separated. For this reason it is frequently used to produce features for a variety of machine learning and pattern recognition tasks. As with PCA, the transformation can be estimated simply through eigenvector decomposition.

Heteroscedastic Linear Discriminant Analysis (HLDA) is a generalization of LDA, which relaxes the assumption that all classes share the same covariance matrix (the *within-class* covariance in LDA is simply the global covariance matrix of the data). Unlike LDA and PCA, there is no closed-form derivation of the HLDA transformation, which makes it more complex and computationally intensive to calculate. MLLT [Gopinath, 1998] is another linear transformation which was designed to compensate for the error introduced by using multivariate Gaussians with diagonal covariance matrices to model speech sounds, as discussed in Section 2.5.1. While in fact this is simply a variety of HLDA, empirically it has been shown to be useful as a postprocessing step after applying an LDA transform.

2.7 Search Algorithms for LVCSR

For a small-vocabulary isolated word recognition system, where each word is modeled using a single HMM, a word hypothesis can be found by simply evaluating the input data against all word models and picking the model with the highest probability. However, this approach becomes very slow as the size of the vocabulary increases, and is completely intractable for connected-word recognition. For this reason, heuristics are necessary to make automatic speech recognition achievable in practical, real-time systems, and therefore, search errors are inevitable in the absence of perfect heuristics. Although, as previously mentioned, the focus of this thesis proposal is on modeling errors, the mechanics of large-vocabulary search, and in particular the technique of multi-pass search are essential background for the proposed work.

2.7.1 Viterbi Beam Search

A common solution to the problem of search complexity is to evaluate all models time-synchronously, retaining only the highest scoring paths at each time point. Unlikely hypotheses are discarded early on, such that for most of the observation data, only a small subset of the model and state space needs to be explicitly evaluated. Since the Viterbi algorithm is concerned only with finding the probability of the most likely state sequence, the resulting score remains exact as long as the most likely state sequence was not eliminated from the search.

This algorithm is known as *beam search* [Haeb-Umbach & Ney, 1994], and is implemented by keeping a list of "active" states for each timepoint (for first-order HMMs, in fact, only the present and previous timepoint need to be stored). After the probability of the best path entering each state has been calculated, a "beam", or ratio, is multiplied by the highest probability to obtain a threshold value. All states whose best path probability falls below the threshold are removed from the active list for the current frame, and no transitions out of them are considered when evaluating the next frame.

2.7.2 Two-Level Viterbi Search

In large-vocabulary continuous speech recognition, as mentioned in Section 2.1, more advanced algorithms are required, due to the fact that the space of possible sentences is extremely large. Since it is clearly impossible to model every possible sentence using a separate HMM, we must use a *two-level* search strategy, in which sentence hypotheses are constructed dynamically based on the results of word-level recognition. This entails a modified version of the Viterbi algorithm, where, instead of finding the best state sequence, we attempt to find the best *word* sequence. Therefore, transitions between states inside a word and transitions between words are treated separately.

This simple two-level strategy is known as *flat lexicon search*. For each word in the vocabulary, an HMM is constructed, possibly by concatenating a sequence of subword models. Transitions are then created between the final state of each word HMM and the initial state of all other word HMMs. When calculating the path score for the initial state of a word, the language model score is calculated based on the previous word, and the identity of the best previous word is recorded along with the current time point in a *backpointer table*. For word-internal transitions, instead of recording the previous state in the best path ending in a given state, a pointer to this backpointer entry is "propagated" through the states of the word along with the corresponding path score. For each state of each word, the algorithm stores a pair consisting of the best path score ending in that state, and the backpointer entry corresponding to that path score. This is also known as the *token-passing algorithm*, where the "token" is a structure which here contains a score and backpointer entry [Young et al., 1989].

When a higher-order language model is used, it may be the case that the best state sequence entering a given word does not correspond to the best word sequence. Assuming a trigram model (i.e. a second-order Markov model), this is the case when the language model score given the previous two words of the best path is sufficiently lower than that for some other two-word history. Consider the case in Figure 2.5. Here, the best path entering word C involves exiting word A with path score x. Likewise, the best path entering word D involves exiting word C with path score z, which incorporates word A and path score x, since these were the optimal candidates open entering C, and all lower-scoring paths were discarded.

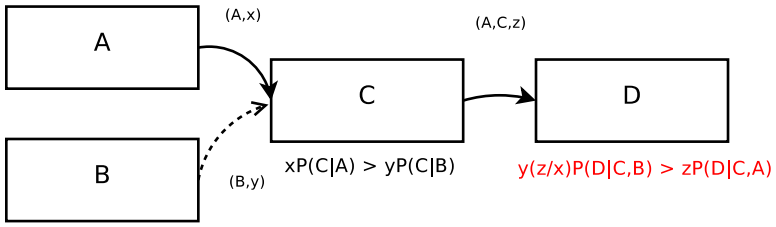

Figure 2.5: Search error: The best state sequence is not the best word sequence

However, in this example, it is actually the case that, had we instead entered word C from B with path score y, the trigram language model score $P(D|C,B)$, when combined with the alternative path score that would have resulted from entering C from B with y, would be greater than the score for the best word sequence (A,C,D) found by the search algorithm. Unfortunately, since the path exiting word B with path score y was discarded on entry to C, there is no way for the search algorithm to discover this optimal word sequence. In order to fully incorporate a trigram language model, it is necessary to propagate all the alternative two-word histories and associated path scores, as shown in Figure 2.6.

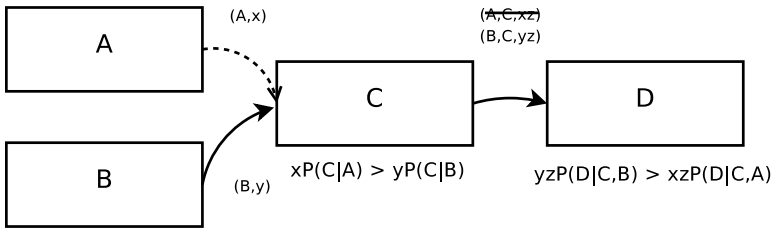

Figure 2.6: Full N-Gram histories retained to avoid search errors

In practice, since this involves an order of magnitude more storage space in the backpointer table, an approximate trigram search is frequently employed, where, as in Figure 2.5, only the current best word sequence is considered when calculating language model scores.

2.7.3 Lexicon Tree Search

With large vocabularies, the flat lexicon search algorithm can be quite slow, because the number of states in the decoding network becomes extremely large. In addition to beam pruning, which temporarily removes words from consideration when the best path score falls below a given threshold, it is also possible to compress the decoding network so that fewer states need to be evaluated at any given timepoint. A commonly used method of compressing the decoding network is to use a *tree-structured lexicon*. This is based on the fact that many words share common phonetic prefixes. Therefore, in a recognizer based on subword units, it is possible to group all words beginning with the same phone into a prefix tree, where the leaf nodes correspond to unique words, as shown in Figure 2.7.

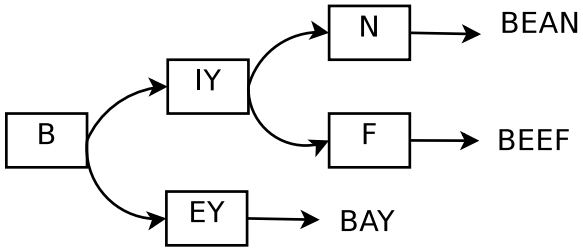

Figure 2.7: Tree-structured Lexicon for BEAN, BEEF, and BAY

In a lexicon tree decoder, transitions between phones must also be treated separately from normal HMM transitions, with propagation of tokens through the lexicon tree. One specific problem with lexicon tree decoding arises from the fact that the identity of a hypothesized word is not known until a leaf node in the lexicon tree is reached. If a single, static lexicon tree is used, it is not possible to apply language model scores until the final phone of a word has been entered.

This results a similar problem to the one encountered with trigram scoring in flat lexicon search, except that it also occurs for bigram language models. As shown in Figure 2.8, only the predecessor word with the highest path score is propagated to the point where the language model score is applied. However it is possible, and in fact quite likely, that another predecessor word would have been preferred had the language model score been available at word entry.

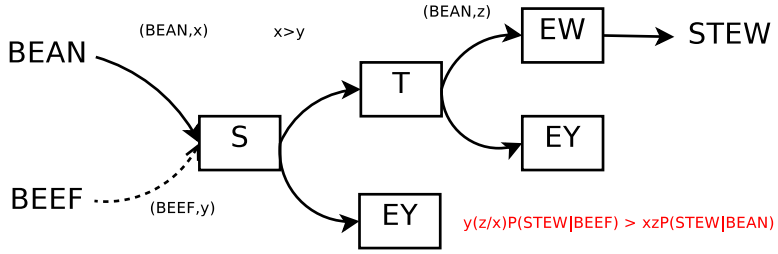

Figure 2.8: Search Error: A single lexicon tree does not retain N-Gram history

This problem is typically solved by making a copy of the lexicon tree for each unique trigram history, as in the *word-conditioned lexicon tree search* algorithm [Kanthak et al., 2000]. Alternately, in a token-passing decoder, multiple active hypotheses (in the form of tokens) can be associated with any given state in the search graph. An approximate solution is employed in the SPHINX-III decoder (since Sphinx 3.3), which uses a fixed number of lexicon trees and "rotates" them on every N word transitions. This reduces the probability of search errors of the sort shown in Figure 2.8 without the overhead of dynamically creating lexicon tree copies. The SPHINX-II decoder and its descendants instead use a multi-pass strategy, described in more detail in the next section.

2.8 Multiple Pass Search

The output of speech recognition systems typically consist not only of a single hypothesized word sequence but also a *word lattice* which is an encoding of a very large number of alternative sentence hypotheses [Ney & Aubert, 1994]. This word lattice is an approximate, finite representation of the space of sentences $\mathbb{S}$ which has been searched by the decoder. It is also frequently the case that speech recognition systems use a *multi-pass* search strategy [Schwartz et al., 1996], which implicitly involves a multi-stage reduction in the search space. For example, in the PocketSphinx system [Huggins-Daines et al., 2006] used as the baseline in our research, a three-pass search strategy inherited from the earlier SPHINX-II system [Huang et al., 1993, Ravishankar, 1996] is used.

In the PocketSphinx decoder, an approximate first-pass search, using a static lexicon tree, is first used to generate a short-list of words at each frame. As mentioned in Section 2.7.3, this search strategy suffers from widespread search errors. Therefore, the second pass uses a flat lexicon search, but uses the short-list generated in the first pass to restrict the set of words to be searched in each frame to a manageable number. However, as described in Section 2.7.2, this search algorithm uses an approximate trigram scoring technique, where only the best 2-word history is considered when applying language model scores at word transitions. To compensate for this, the third pass then performs an A* search over the resulting word lattice, allowing all trigram histories to be considered.

This organization of multiple passes is designed such that each successive pass is more exhaustive than the previous one, but also searches a more restrictive space of hypotheses. In this way, the known deficiencies of the earlier passes, such as the search error problem in the case of a static lexicon tree, and the approximate trigram problem in the case of a simple flat lexicon search, are corrected by a subsequent pass of search.

Another approach to multiple pass search, exemplified by the AT&T LVCSR-2000 system [Ljolje et al., 2000], is to use a cascade of progressively "stronger" models for each stage of recognition. This approach allows very complex models, such as 6-gram language models, to be used without resulting in a combinatorial explosion of search states. It is this strategy of moving from weak to strong models which is the basis of the research proposed here. The problem with this approach, which we propose to solve, is that the computational load is very unevenly balanced between the different passes. In particular, the first pass in the AT&T system is approximately 60 times slower than subsequent passes, and produces extremely large output lattices. This occurs both because the first pass uses a weak language model, and because it is specifically tuned to minimize the *lattice word error rate*. That is, the search space has been expanded so as to increase the probability that the correct hypothesis is contained within it, even if its probability according to the weak model is low.

The requirement of very dense first-pass word lattices is driven in part by the need to accurately model the uncertainty in recognition. In this case, uncertainty is modeled *internally* by expanding the bounds of the space of hypotheses considered by the decoder. In contrast, we propose to model uncertainty *externally*, by capturing the relation between the reduced and expanded hypothesis space. We focus on one method of "weakening" the model, namely, reducing the size of the vocabulary, which poses problems for the standard model of multi-pass search. Drawing on prior work [Ringger & Allen, 1996, Geutner et al., 1998, Palmer, 2001] which has explored the problem of modeling lexical uncertainty, we show preliminary results which suggest that this approach can be effective in allowing accurate multi-pass recognition with a time-constrained first pass.

2.9 Finite State Transducer Search

A growing tendency in automatic speech recognition work over the last decade has been the explicit use of weighted finite state transducers to represent the search graph, and the use of generic transducer operations to statically optimize these transducers. The original work on this subject was done at AT&T in the late 1990s [Mohri et al., 2000], but the technology has been adopted by a wide variety of research sites and companies. The current state of the art is implemented in the OpenFST toolkit developed at NYU and Google [Allauzen et al., 2007], which forms the basis of the Google speech recognizer and is used in a number of research systems.

The basis of WFST based speech recognition is the observation that the acoustic, language, and lexical models used in speech recognition can all be efficiently expressed in the form of weighted finite-state transducers. Through the operation of *weighted composition* it is possible to link them together into a single transducer mapping acoustic model state sequences to word sequences. The benefit of doing this is that other generic finite-state operations, namely *determinization* and *weighted minimization*, can be used to produce an efficient version of this transducer. These optimizations are able to work on the recognizer as a whole, rather than being confined to one particular component of it, as is the case with standard heuristics such as tree-structured lexicons and language-model lookahead.

The drawback of building the recognizer as a static network is that the size of this network can be unmanageably large, and the process of determinizing and minimizing it is extremely expensive, requiring days or weeks of CPU time. For this reason practical WFST recognizers typically use dynamic composition algorithms, where only some parts of the recognizer are pre-composed and minimized [Allauzen et al., 2009]. In practice, these algorithms are similar to the heuristics used in standard recognizers to handle cross-word

phonetic context and track language model histories, but their generic nature allows them to be more cleanly implemented and more flexible.

Past comparisons of WFST and lexicon tree search [Kanthak & Ney, 2002] have shown the WFST based recognizers to be considerably faster than lexicon tree based recognizers, at the cost of increased memory usage. However, more recent work has focused on "backporting" some of the benefits of WFST recognition to standard dynamic network decoders, resulting in a much smaller performance gap [Soltau & Saon, 2009].

One major drawback of WFST recognizers is that, because the decoding network or parts of it must be generated and optimized off-line in a time-consuming process, it is much more difficult to add new words at runtime, or to adjust language model weights or pronunciations. This is the main reason WFST search techniques are not used in this thesis, since this work is targeted towards highly dynamic systems.

2.10 Word Lattice Generation

In Section 2.8 we discussed the concept of a *word lattice* as the output of intermediate passes of recognition without describing exactly what a word lattice is or how they are generated.

The word lattice is a compact representation of the set of hypotheses generated by a speech recognizer with respect to an input utterance. In this thesis, we distinguish between the *word lattice* which contains timing information and distinguishes between different *segmentations* of the same word sequence, and the *word graph* which represents only the unique word sequences hypothesized by the recognizer. It is trivial to generate a word graph from a word lattice by discarding timing information and applying the *finite-state determinization* algorithm, which is the approach we use in this thesis, although more specific algorithms are available.

The word lattice is a weighted, directed acyclic graph whose nodes represent time points and whose arcs represent recognized instances of a word, with a weight corresponding to this word instance's contribution to the path score of all paths which traverse it. In some systems, SPHINX among them, each node represents a single word, in which case the word identities are typically recorded on the nodes rather than the arcs. The two representations are equivalent and conversion between the two is a straightforward operation.

Another distinction is between *unigram lattices* and *N-gram lattices*. In the former case, the word arcs do not have unique N-gram histories, and thus their weights do not include language model scores. In the latter case, each word arc represents not just a recognized instance of a word, but an instance of that word in a unique N-gram context. The N-gram context here is also referred to as the *language model state*. In the subsequent chapter we will discuss the issue of tracking language model state in more detail, however, it is sufficient to note here that decoders which do not track multiple language model states, such as POCKETSPHINX, are only capable of outputting unigram lattices. In a *word graph*, the lack of N-gram context is not a serious problem, because it can be added to the graph by weighted finite-state composition with the language model. We employ this technique in the domain adaptation and vocabulary expansion experiments detailed in Chapters 4 and 5.

There are two well-established ways of generating word lattices; the more accurate, but more expensive way is to treat word lattice generation as a generalized case of N-best search. Here, each word hypothesis retains pointers to multiple predecessor words, either by retaining the N best predecessors, or by retaining all of those whose scores fall within a beam. The lattice is then constructed by directly by using these back-pointers as the arcs. The drawback of this method is the increased overhead of storing multiple predecessors in the backpointer table.

The approximate form of lattice generation, used by SPHINX, is to reconstruct "imaginary" arcs based on adjacency of word hypotheses. In this case, only the best scoring predecessor is stored with each word hypothesis, which implies that the output of search is actually a tree rather than a directed acyclic graph. Each word may have multiple successors but only one predecessor. If the lattice is constructed in the same manner as above by simply using the backpointers as arcs, it will quickly converge to a single trunk once unreachable arcs are removed. While this fact will become useful to us for entirely different reasons, as described in Section 6.3.3, it results in a lattice which is not particularly useful for rescoring, generating N-best lists, or calculating word posterior probabilities, since it contains only a small number of hypotheses which only diverge towards the end of the utterance.

Therefore, wherever two word hypotheses are adjacent in time, that is, the end frame of the first immediately precedes the start frame of the second, we create an "imaginary" arc linking them even if no such arc was present in the backpointer table. For obvious reasons it is not possible to reuse the language model scores from the history table in building a lattice in this manner, since they will usually refer to completely different N-gram contexts from the ones these new arcs represent. Since these arcs also involve cross-word phonetic contexts that are different from the one recorded in the backpointer table, it is necessary to retain all the final acoustic scores for each possible right context in the backpointer. We can then reconstruct the appropriate acoustic score by looking up the first phone of the second word in this table.

2.10.1 Word Posterior Probability Calculation

In the preceding section, word posterior probability calculation was mentioned as one important use of word lattices. Although decoding in speech recognition is based on the MAP criterion, it is the *a posteriori* probability of the sentence which is being maximized rather than individual words. In addition, this probability is not explicitly calculated since the normalization term is irrelevant to the $\arg\max$ operation.

Word posterior probabilities are useful as a basis for *confidence estimation*. They are also used in discriminative training, specifically for the *maximum mutual information* criterion. The combined acoustic and language model scores for an individual word hypothesis are not equivalent to the posterior probability; instead, they constitute the joint probability of a word and a segment of audio conditioned on the best word sequence and all audio preceding this hypothesis. The posterior probability that we are interested in is the probability of this word hypothesis conditional on the utterance as a whole. In particular, it forms a probability distribution with all the other word hypotheses which are active at a given point in time.

This property of word posterior probabilities inspired a popular technique for computing them known as *consensus decoding*. The consensus algorithm performs a multiple alignment of the word lattice, transforming it into a sequence of *equivalence classes*, which are words (or epsilons) that are considered to be aligned with each other. Each equivalence class forms a probability distribution which is equivalent to the posterior probability distribution over its constituent words. The hypothesis formed by taking the maximum *a posteriori* element in each equivalence class is known as the *consensus hypothesis*, and typically contains fewer errors than the MAP hypothesis obtained from the decoder. This is because maximizing the posterior probability in each alignment is equivalent to explicitly maximizing the word error rate, as opposed to maximizing the sentence error rate, which is the effect of using the MAP hypothesis.

An alternative way of computing word posterior probabilities is to calculate them directly over arcs in the word lattice. This technique has been used both to compute posterior probabilities for confidence estimation [Wessel et al., 2001] and for maximum mutual information training [Povey, 2003]. The algorithm

for computing lattice-based posteriors is based on a factorization of the marginal probability of a word into *forward* and *backward* probabilities, just as in the Baum-Welch algorithm for training HMMs. The product of the forward and backward probabilities gives the joint probability of a word at a given time point and the audio of the utterance as a whole. By normalizing this using the posterior probability of the entire utterance, we obtain the posterior probability of just that given word hypothesis. The normalizer is easily calculated as the final forward probability or the initial backward probability, just as in Baum-Welch.

A full description of the word posterior probability algorithms used in POCKETSPHINX and MULTISPHINX can be found in Chapters 3 and 6. In particular, MULTISPHINX, as an anytime system, has the capability of computing posterior probabilities based on partial results. The accuracy of these probability estimates is discussed at greater length in Chapter 6.

2.11 Summary

In this chapter we have introduced the current approach to the problem of automatic speech recognition, which is also the one used by the recognition systems described in this thesis. Where possible we have noted where our system stands in respect to the state of the art, a comparison which will be continued in Chapter 3.

The problem of search, along with the concept of information, are fundamental to the perspective of automatic speech recognition presented in this thesis. Our goal is to manage the size of the search space, as well as its lattice representation, so as to preserve as much information as possible during recognition, while minimizing the latency of the decoding algorithm. We are concerned not only with the information contained in a lattice, but also the related concepts of cross-entropy and divergence. Namely, the goal is a lattice that is not only compact but also representative of the input *at some level of representation*. As we will see in future chapters, the notion of a level of representation is central to the strategies used in this thesis to achieve this goal.

Chapter 3

The PocketSphinx System

This chapter describes the POCKETSPHINX speech recognition system, which is the basis for the work proposed in this thesis. POCKETSPHINX draws on a long history of speech recognition research at CMU. It incorporates source code and algorithms from the Sphinx-II [Huang et al., 1993] and Sphinx-III [Placeway et al., 1997] systems, along with a number of more recent innovations in efficient ASR. The POCKETSPHINX system was first described in [Huggins-Daines et al., 2006], with updated descriptions in [Huggins-Daines & Rudnicky, 2008] and [Huggins-Daines & Rudnicky, 2009]. This chapter corresponds to the development version as of October 2010, which is essentially the same as version 0.7.

3.1 Design Goals

The initial POCKETSPHINX project [Huggins-Daines et al., 2006] was intended simply to be a version of SPHINX-II implemented with fixed-point math for use on mobile devices of the time which lacked floating-point hardware. However, in order to achieve reasonable performance it became clear that other architectural changes were necessary. Therefore, POCKETSPHINX has five main design goals, which reflect its orientation towards resource-constrained and mobile devices. This section will cover these in some detail as a preface to a detailed description of the system and its architecture. These goals are:

- Portability
- Simplicity of implementation
- Small code and data footprint
- Thread safety
- Memory efficiency

POCKETSPHINX is intended to be portable to a wide range of systems and devices. It is written in the ANSI standard C programming language, with limited use of platform-specific functionality. Currently supported platforms are GNU/Linux, *BSD, Mac OS X, and other Unix-like systems, Android, uClinux, Windows, and Windows CE. A port to SymbianOS (Series 60) is in progress. The use of the standard C programming language is in fact fundamental to all design goals. In keeping with modern best practices in C programming, object-oriented techniques such as abstract types and struct encapsulation are used throughout the code base to ensure thread safety and interface stability.

3.2 Decoder Architecture

The decoder architecture of POCKETSPHINX is largely inherited from Sphinx-II. A high-level overview of the decoder is shown in Figure 3.1. POCKETSPHINX uses a three-pass search strategy, consisting of the following stages:

- FWDTREE - Viterbi search using a static lexicon tree
- FWDFLAT - Viterbi search using a flat lexicon

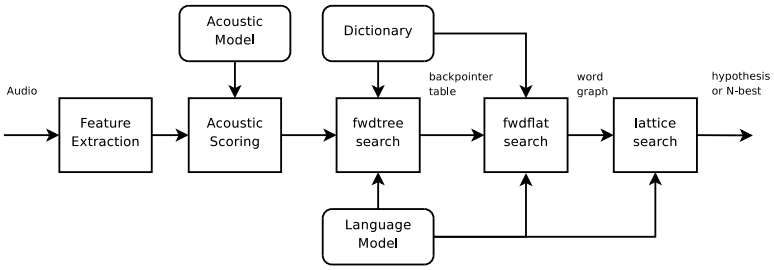

Figure 3.1: POCKETSPHINX Decoding Architecture

- BESTPATH - Word graph search

The decoder itself is separated into three main modules: acoustic modeling (which includes front-end processing and output density evaluation), forward search, and word graph search. These are represented by the objects `acmod_t`, `ngram_search_t`, and `ps_lattice_t`, respectively. The flow of data through the acoustic modeling component is shown in Figure 3.2.

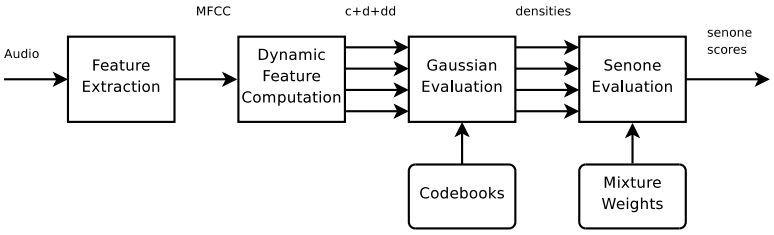

Figure 3.2: POCKETSPHINX Acoustic Modeling (`acmod_t`)

The general architecture of the two forward search passes is shown in Figure 3.3. The forward lattice (also known as the *backpointer table*) from lexicon tree search is used to generate a frame-by-frame list of *expansion words*, which forms the dynamic vocabulary searched by the flat lexicon search.

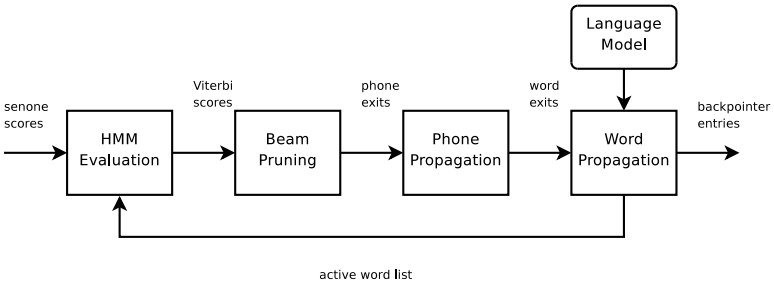

Figure 3.3: POCKETSPHINX Forward Search (`ngram_search_t`)

3.3 Acoustic Modeling

POCKETSPHINX is specifically optimized for *semi-continuous* acoustic models [Huang, 1989], also known as *tied-mixture* models, though it also contains support for standard *fully-continuous* models, also known as *tied-state* models. The use of semi-continuous models is very important to the performance of the system, because the structure of these models is such that it allows for output density computation which is efficient both in terms of the number of operations required and the patterns of memory access.

Since all Gaussian densities are shared between all tied states, the so-called *codebook* of densities need only be evaluated once per frame. More importantly, it has been observed that the density values for this codebook are highly "peaked" - that is, a few densities account for the majority of the mixture density for any given observation. Therefore, only the top N densities (typically for $N \leq 4$) are used in computing the observation density for a feature vector:

$$p(x|\lambda_i) = \sum_{k \in topN} w_{ik} N(x; \vec{\mu}_{ik}, \vec{\sigma}^2_{ik}) \tag{3.1}$$

The indices of these top N densities tend not to change from frame to frame. For this reason, they can be used for an efficient form of Gaussian selection. Specifically, the computation of densities is split into two parts: first, the densities of the previous frame's top N Gaussians are computed. The smallest of them is then used as a threshold for partial computation of all the other densities - that is, if during computation the density of any other Gaussian becomes smaller than that of the currently Nth best Gaussian, computation is terminated, since said Gaussian will not be used in mixture density computation. As new Gaussians are computed, they are inserted into the top N table, with the effect that the partial computation becomes more efficient as it progresses through the set of Gaussians. Even though in a semi-continuous model, Gaussian computation is not particularly costly, this partial computation is quite effective in reducing the computation load of acoustic model evaluation, with no cost in accuracy.

Another key feature of the semi-continuous acoustic modeling done in POCKETSPHINX is the division of the acoustic feature vector into multiple *streams*, each of which is modeled independently with its own codebook of Gaussians. The acoustic score for a given state is then computed as the product of the posterior probabilities of that state given each stream's mixture model. Aside from the fact that this provides more trainable parameters to offset the extremely small number of Gaussians in a semi-continuous model, this makes computation of the individual stream probabilities more efficient since their dynamic range is much smaller, allowing them to be represented in 8 bits or less. Since all computation is done in log space, the *log-addition* operation consumes a large portion of the CPU time used by acoustic model evaluation. The ability to quantize individual Gaussian densities to less than 8 bits enables the use of a very small lookup table for computing the log-addition function.

A recent innovation has been the ability to vary N according to the range of densities in a given frame. This is accomplished by counting the number of densities which fall inside a beam for each feature stream, taking the maximum of this, and using only this many densities in computing the mixture scores. Because of the large number of log-additions involved in computing mixture densities, reducing the number of inputs that are used makes a significant impact on CPU usage, although if the beam is too narrow it also leads to increased error and a more complex search space.

3.4 Decoding

Decoding in POCKETSPHINX uses a multi-pass approach, where each pass serves to overcome the approximations made by the previous pass. In that sense this thesis can be thought of as a straightforward extension of this idea to more sophisticated approximations and ways of overcoming them in subsequent passes. As mentioned in 3.2, the three passes of search in POCKETSPHINX are known as `fwdtree`, `fwdflat`, and `bestpath`, and are described in detail in this section.

3.4.1 Approximate Tree-based Search

The `fwdtree` search is based on a tree-structured lexicon. It is approximate in the sense that it uses only a unigram tree, as described in Section 2.7.3, leading to an inability to fully utilize N-gram language models. In addition to this approximation, several other approximations are made which reduce its memory and CPU consumption. For the most part these have been inherited from SPHINX-II and are thus detailed in [Ravishankar, 1996], but they will be reviewed here for clarity in the sections to come.

The `fwdtree` search splits the search graph into the *lexicon tree*, which is a static tree of phone HMMs that represents the multiple-phone words in the lexicon, and the *word HMM array*, which is a dynamically changing array of individual phone HMMs that represents single-phone words and the *final phones* of the multi-phone words in the lexicon tree. These final phones are not included in the lexicon tree, which is why single-phone words are treated separately. In addition, they are allocated on demand, as words reach their final phones. The lexicon tree itself is properly described as a forest, since it has distinct root nodes for each initial phone.

The final phones of words are always special in a lexicon tree based system since they represent the earliest point at which word identities are known. In addition, since the POCKETSPHINX system uses triphones, the final phone of a word is in fact a set of triphones corresponding to all the possible initial phones of the next word, which is the main reason why final phones' HMM structures are allocated on demand. The `fwdtree` search uses an auxiliary (and slightly misnamed) *homophone set* table which links together all words which share the same non-final phones. This gives the set of words to be considered as *candidates* for generating new final phone HMMs.

Before generating or entering the final phone HMMs in the word HMM array, a separate *candidate list* is generated containing all of the words whose final phone is about to be entered. These candidates are then sorted by start frame and word ID. Since a given hypothesis word nearly always produces multiple word exits until its final phone is pruned away by the beam, it is possible to cache the language model and path score for that word. The sorting of candidates allows this to be done through a separate table of "delta scores" which include the language model score for the best path entering this word (allowing a limited form of N-gram modeling) combined with the score of this best path up to the initial frame of this word. The efficiency gains from this caching are detailed in [Ravishankar, 1996].

The `fwdtree` search uses a large number of tunable *beam widths* in order to speed computation. In particular, a separate beam is applied to the final phone of words from all other HMMs. This is feasible because the language model score has been applied to the path score in the final phone, making the scores more discriminative and allowing for a narrower beam. This is also important because of the large number of separate right-context HMMs which are allocated for the last phone in order to model cross-word triphone contexts.

3.4.2 Flat-lexicon Search

The second pass `fwdflat` search is a rescoring pass over the lattice generated by `fwdtree`. However, it does not rescore the lattice directly. Instead it uses a *shortlisting* strategy, vwhere the active vocabulary for search is determined by the set of words active in the first pass within a sliding window of frames. The overall lexicon for the search is determined by the complete set of words hypothesized in the first-pass lattice, and is generally a factor of 10 smaller than the lexicon as a whole. For this reason, the decoding graph is allocated dynamically for each utterance using only these words, a strategy which is less than optimal for mobile devices with their limited memory bandwidth.

The "flat lexicon" is also generally known as a *word loop*. Specifically, for each word in the lexicon, a chain of phone HMMs, known as a *word HMM*, is constructed, modeling the word as a whole. There are then implicit links from the non-emitting end states of each word HMM to the entry states of all other word HMMs in the loop. The advantage of this construction over a lexicon tree is that the identity of words is known throughout a word HMM. Therefore, bigram probabilities (but not trigram probabilities) can be calculated exactly upon entering the first state of a word. Higher order N-gram probabilities are calculated based on the best path ending in the current word, a strategy sometimes known as *poor man's trigram* [Soltau et al., 2001]. In practice, the difference in accuracy between decoding a bigram network using poor man's trigram language modeling, and decoding an exact trigram network, is fairly small. In addition, the size of an exact decoding network grows exponentially with the order of the N-gram model used, such that in practice it is not feasible to allocate such a network statically.

The shortlisting strategy works by first constructing a table for each frame of audio of all words starting in that frame. These are usually then pruned to only include those words which persist for more than a certain number of frames. This construction is necessary because the output of the first pass is represented as a backpointer table which records only the end frame and predecessor node for each word hypothesis. In Chapter 6 we describe an algorithm for constructing this table incrementally in parallel with the first pass of recognition, which is used in the MULTISPHINX decoder which is the subject of this thesis.

When shortlisting is active, whenever a word exit is reached, instead of simply entering all word HMMs in the loop, the table of word entries is consulted to determine which words were entered in the vicinity of the current frame in the first pass of recognition, and only these words are actually entered. The primary effect of this is to reduce the amount of HMM evaluation that need be done, since fewer words will be active in any given frame. A secondary benefit is that only the output distributions for the states of active words will need to be calculated. This is important because the `fwdflat` pass does not reuse the acoustic scores from the `fwdtree` pass. The main reason for this is that the memory footprint of these scores is quite large. In addition, the set of active states can be disjoint between the two passes.

In MULTISPHINX the two passes operate concurrently within a bounded number of frames of each other, and thus it becomes more feasible to reuse acoustic scores, since only this window of active frames need be maintained in memory.

3.4.3 Lattice Generation

The third pass of POCKETSPHINX is a lattice rescoring pass over a word lattice generated from the backpointer table output by `fwdflat` search. In fact, all three passes of search in POCKETSPHINX are optional, so this can just as well be computed over the output of `fwdtree` search instead, from a lattice file stored on disk, or not done at all. This ability to generate hypotheses from any pass of search is fundamental to

the idea of *anytime recognition* which is the goal of the MULTISPHINX system and one of the goals of this thesis. In practice, in POCKETSPHINX, partial results are only obtainable from the `fwdtree` pass, with the second and third passes operating in batch over the entire utterance. In MULTISPHINX this is updated such that, in addition to the passes operating incrementally in parallel, it is possible to obtain a partial result from any pass, or a combination of them. When combined with the vocabulary scaling and optimization techniques discussed in Chapter 5, which allow us to split the decoder into multiple discrete passes, each of which improves on the previous, this allows a coarse-grained continuum of latency versus accuracy to be achieved.

The lattices in POCKETSPHINX are generated in the "approximate" manner described in Section 2.10.1, where "imaginary" arcs are posited between word hypotheses that are adjacent in time. One consequence of this is that these lattices have a tendency to be "front-loaded". That is, due to the lack of language model context, the initial parts of the backpointer table tend to be full of short word hypotheses which do not produce viable successor paths. However, since they do emit word exits, they are recorded in the backpointer table, and are thus linked to all adjacent successor words in the lattice generation process, whether or not the path produced by such a transition would have survived in the original search. In particular, this makes lattice visualization difficult and may lead to inaccurate calculation of word posterior probabilities since these bogus arcs all consume small amounts of probability mass. In the discriminative training process used in SPHINX, posterior probability based pruning is applied to the lattice which does tend to remove many of these extraneous arcs, and also reduces the computational load of discriminative training while improving the accuracy of the resulting models [Qin & Rudnicky, 2010].

One feature of SPHINX lattices is that nodes represent unique words, or more specifically unique (word, start frame) pairs. As mentioned before this allows the words to be recorded on the nodes rather than the arcs. This representation is equivalent to one where words are recorded on arcs but is more memory efficient. However, the primary reason why it is done this way in SPHINX is related to the computation of acoustic scores for arcs. Recall from Section 2.5.1 that the smallest sub-word unit used in building word models is not an independent phone, but rather a *triphone*, which represents a single phone in a particular left and right context. At the end of a word, *right context expansion* occurs, where the word HMM is effectively split into separate branches for each right context which results in a different senone sequence for the final triphone. Therefore, if more than one of these branches is exited in the same frame, this results in multiple instances of the same word with different path scores. Although, as per the general Viterbi update rule, we propagate only the path with the best score, determining the best score upon entering a successor word requires that we match the first phone of that word with the corresponding right context for the previous word.

It is for this reason that we cannot collapse multiple words into a single node in the lattice - the acoustic score for a given arc is conditional on the identity of the destination word for that arc. In addition, having word identities on nodes allows an optimization of lattice rescoring described in the next section. As will be discussed in Chapter 6, however, in order to properly calculate word posterior probabilities it is necessary to add language model information to the lattice, at which point the node identities become *language model states* rather than words.

Lattice Rescoring

The lattice rescoring is based on a modified single source shortest path algorithm [Dijkstra, 1959], where path scores are stored for arcs rather than nodes. Since nodes in Sphinx lattices represent unique words, each

```
1  import collections
2  from lattice import baseword, LOGZERO
3
4  def bestpath_edges(dag, lm):
5      start = dag.start_node()
6      end = dag.end_node()
7      for e in start.exits:
8          e.lscr, e.lback = lm.score(baseword(e.dest.sym),
9                                     baseword(e.src.sym))
10         e.pscr = e.ascr + e.lscr
11     bestend = None
12     bestescr = LOGZERO
13     for e in dag.traverse_edges_topo():
14         for f in e.dest.exits:
15             lscr, lback = lm.score(baseword(f.dest.sym),
16                                    baseword(e.dest.sym),
17                                    baseword(e.src.sym))
18             pscr = e.pscr + f.ascr + lscr
19             if pscr > f.pscr:
20                 f.pscr = pscr
21                 f.lscr = lscr
22                 f.lback = lback
23                 f.prev = e
24                 if f.dest == end and f.pscr > bestescr:
25                     bestend = f
26                     bestescr = f.pscr
27     return bestend
```

Listing 3.1: The `bestpath_edges` algorithm

arc can be said to represent a unique bigram history. Therefore, exact trigram computation can be done in the extension step of the algorithm - the path score for an arc is the acoustic score for that arc plus the language model score of the two words at each end of that arc given the first word of the best predecessor arc. To obtain the full path score, the acoustic score for the final word is stored separately in lattice generation and added in after search.

This algorithm, known as `bestpath_edges`, is described in Python code in Listing 3.1. Because the number of arcs can be extremely large, the search is based on a topological traversal of the nodes, the algorithm for which is shown in Listing 3.2. The important thing to note in this algorithm is that the language model component of the path score for an arc f which follows another arc e, as calculated on lines 15 through 17, is actually the language model score for the word following f, in the context e, f. Therefore, the best path through the lattice is obtained by finding the arc entering the final node with the highest path score. This has the side effect that the language model score for f is unrelated to the actual word hypothesis

```
def traverse_edges_topo(self):
    for w in self.nodes():
        w.fan = 0
    for x in self.edges():
        x.dest.fan += 1
    start = self.start_node()
    end = self.end_node()
    Q = deque(start.exits)
    while Q:
        e = Q.popleft()
        yield e
        e.dest.fan -= 1
        if e.dest.fan == 0:
            if e.dest == end:
                break
            Q.extend(e.dest.exits)
```

Listing 3.2: The topological edge traversal algorithm

represented by f. This problem is solved in MULTISPHINX by the lattice expansion algorithm detailed in Section 6.3.2.

Posterior Probability Computation

Lattice rescoring and posterior probability computation are integrated in POCKETSPHINX. This is done because the forward traversal of the lattice required for bestpath search is the same as that required for calculating the forward probability, and little extra computational cost is incurred by always calculating both. Due to the lack of unique N-Gram histories for nodes in the lattice, posterior probabilities are calculated only using the bigram portions of an N-gram language model even in the case of a trigram model. With the exception of this, the basic algorithm is similar to that described in [Kemp & Schaaf, 1997, Wessel et al., 2001], and also extremely similar to the forward-backward algorithm used for estimating posterior probabilities of HMM states and arcs, as discussed in Section 2.5.1. For each arc, *forward* and *backward* probabilities are computed, which represent, respectively, the joint probability of all paths ending in that arc, and the conditional probability of all paths which follow it. Assuming the conditional independence of the observation sequence, the product of the forward band backward probabilities is the joint probability of a word instance and the utterance as a whole, which need simply be normalized to obtain the posterior probability of that word instance, as shown in Equation 3.5.

$$\alpha_t(w) = P(o_1^t, w) = \sum_{v_1^N : v_N = w} P(v_1^N, o_1^t) \tag{3.2}$$

$$\beta_t(w) = P(o_{t+1}^N | w) = \sum_{v_1^N : v_1 = w} P(v_2^N, o_1^t | w) \tag{3.3}$$

$$\alpha_t(w)\beta_t(w) = P(w, o_1^N) \tag{3.4}$$

$$P(w|o_1^N) = \frac{\alpha_t(w)\beta_t(w)}{P(o_1^N)} \tag{3.5}$$

3.5 Benchmarks

This section gives performance and accuracy results for POCKETSPHINX on several standard recognition tasks, as well as on several CMU-specific dialog system tasks. Since performance and accuracy are highly dependent on the tuning of search beams and other decoder parameters, we do not claim these to be the best achievable results with the decoder. Tuning a speech recognizer involves a wide variety of parameters, which tend to have contradictory effects on the speed and accuracy of the decoder. It is therefore difficult even to formulate an objective function for tuning, let alone one which can be optimized numerically.

In all cases, testing was done using the original waveforms rather than precomputed feature files as input. This allows the profiling charts in Figure 3.4 to show the true impact of feature extraction on performance, which, in fact, is quite minimal. Because of the orientation of POCKETSPHINX towards mobile platforms, many of which are incapable of recording wide-band audio, all training and test data was downsampled to 8kHz, which should be kept in mind when comparing error rates with other systems.

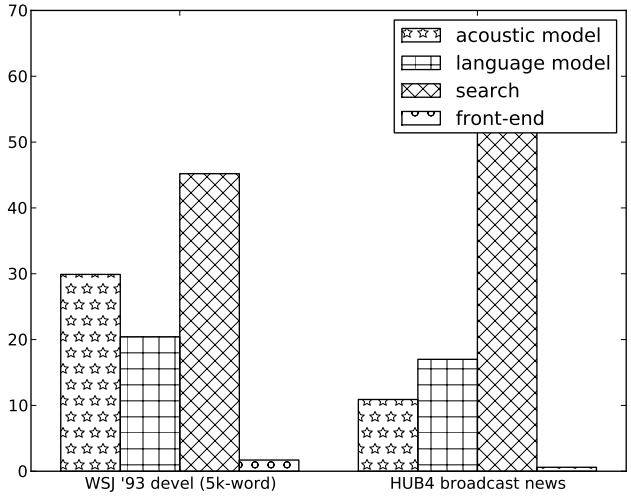

Figure 3.4: Profiling Data, PocketSphinx (desktop)

The tuning algorithm we have employed on the development set in these benchmarks is a crude stepwise search, where a pre-determined list of parameters is adjusted one at a time, carrying forward the "best" value of all previously adjusted parameter when moving to the next. Since it is usually possible to make the decoder arbitrarily fast by disregarding accuracy, we use the word error rate as the primary objective for tuning. However, certain parameters such as the last-phone beams in POCKETSPHINX can be adjusted so as to show only modest gains in accuracy combined with considerable slowdowns. Therefore, an option is provided to set a target error rate; once the error rate falls below this threshold for some parameter, values which result in lower error rates will not be considered unless they also improve runtime performance. The effect of the target error rate is shown in Table 3.1 for the WSJ5k development set. Standard deviations were generated over a randomized 10-way split of the test data. The result of tuning with a target error rate is the strictest set of parameters which result in the desired accuracy. Since the speed of a recognizer can degrade significantly in the face of noisy or out-of-domain data, this is a desirable outcome if consistent runtime performance is desired.

Target WER	WER	xRT	Parameters
7.5	7.57 ± 1.41	0.058 ± 0.001	`-beam 1e-60 -wbeam 1e-35 -fwdflatbeam 1e-65 -lpbeam 1e-35 -lponlybeam 1e-32 -maxhmmpf 1500 -maxwpf 5`
8.5	8.41 ± 1.84	0.041 ± 0.000	`-beam 1e-50 -wbeam 1e-30 -fwdflatbeam 1e-50 -lpbeam 1e-30 -lponlybeam 1e-24 -maxwpf 3 -maxhmmpf 500`
10	9.85 ± 2.16	0.039 ± 0.000	`-beam 1e-50 -wbeam 1e-25 -fwdflatbeam 1e-45 -lpbeam 1e-25 -lponlybeam 1e-24 -maxwpf 3 -maxhmmpf 500`

Table 3.1: Effect of Target Error Rate on Pruning, devel5k

The parameter space has many local maxima, and therefore the order in which parameters are varied has a significant effect on the result, as shown in Table 3.2. In particular, we found that the general beam widths should not be optimized first, because the negative effects of a narrow beam only become apparent in conjunction with other tuning parameters considered much later, such as the absolute pruning of HMMs per frame (it is for this reason that `-maxhmmpf` is not present in Table 3.1, because considering it last resulted in a catastrophic rise in error). Some previous work has gone into a principled solution to these problems [Colthurst et al., 2007, Bulyko, 2010].

Finally, the tuning process generates a large number of (x, y) points which define the relationship be-

Target WER	WER	xRT	Parameters
10	9.85 ± 2.16	0.039 ± 0.000	`-beam 1e-50 -wbeam 1e-25 -fwdflatbeam 1e-45 -lpbeam 1e-25 -lponlybeam 1e-24 -maxwpf 3 -maxhmmpf 500`
10	9.97 ± 1.78	0.038 ± 0.000	`-maxwpf 11 -lponlybeam 1e-28 -lpbeam 1e-30 -fwdflatbeam 1e-50 -wbeam 1e-20 -beam 1e-30`

Table 3.2: Effect of Parameter Order on Pruning, devel5k

tween error and performance. For this reason it is often more informative to report a curve fit to these points, such as a *time-accuracy* plot where error rate is on one axis and realtime factor is on the other, in place of a table or chart of "optimal" values. An example of such a plot can be seen in Figure 5.2.

3.5.1 Platforms

For testing we used two platforms representing typical embedded and desktop platforms. For embedded and mobile testing, we used a BeagleBoard, with a 500MHz ARM Cortex-A8 core. For desktop testing we used a 3GHz Intel Core 2 Quad. The systems were kept otherwise idle, and 10-way cross-validation was used to generate error bars for performance and accuracy results. On the multi-core system we used all available processors to run separate decoding jobs in parallel.

On the Intel platform the code was compiled with GCC 4.4.3 using the 64-bit instruction set and optimization flags `-O3 -ffast-math -march=core2 -mno-ieee-fp -msse3`. On ARM, the code was compiled with GCC 4.4.3 using the ARMv7 instruction set, including VFP and NEON floating point, using the optimization flags `-O3 -ffast-math -march=armv7-a -mtune=cortex-a8 -mfpu=neon -mfloat-abi=softfp`. Test data and models were stored on an NFS filesystem mounted from the workstation platform over a local 100Mbps Ethernet link.

For all tests, the development version of POCKETSPHINX (as of 2010-11-29) was used. There has been steady progress in the performance of POCKETSPHINX since its inception in 2006, as shown in Table 3.3.

Version	Dataset	Runtime Factor (ARM)
0.5.1	devel5k	1.23 ± 0.06
0.6.1	devel5k	1.14 ± 0.05
20091129	devel5k	1.10 ± 0.05

Table 3.3: 10% Performance Improvement in POCKETSPHINX since 2006 using the same parameters and models

3.5.2 Evaluation Data

A variety of evaluation sets have been used to evaluate the baseline decoder and the system developed in this thesis, representing different domains and levels of computational complexity.

The least difficult data set used was the CSR-0 Wall Street Journal data, specifically the "November '93" development and evaluation sets, hereafter referred to as the "WSJ0" task. This data set is divided into "5000-word" and "20,000-word" tasks, where a specific vocabulary of the given size has been provided which covers the respective test set. For the 5000-word task there is a further division in to "open" and "closed" vocabulary sets, and for both there are "verbalized" and "non-verbalized punctuation" variants, where the former contains explicitly spoken punctuation. This last feature stems from the fact that these training and test data were designed for constructing dictation systems. Because of this, the audio consists of carefully read speech, recorded in an acoustically clean environment judged to be similar to a quiet office.

For the 5000-word closed-vocabulary WSJ0 task, we used the 5000-word vocabulary trigram model supplied with the evaluation materials. For the 20,000-word task we used a 64,000-word vocabulary trigram model trained on the WSJ0 language model training data. The acoustic model used in these tests is the standard one shipped with POCKETSPHINX 0.5, trained from the complete WSJ0 and WSJ1 close-talking microphone sets (approximately 180 hours), using 5000 tied states and four independent feature streams with a tied codebook of 256 Gaussians for each.

For a more complex task using real-world speech data we used the 1996 HUB4 Broadcast News development and test sets. Only the F0 condition (planned speech) was used in these benchmarks. As in the WSJ task the acoustic models were trained with speech downsampled to 8kHz. The training set consisted of the 1996 and 1997 HUB4 training sets along with the WSJ0 and WSJ1 acoustic model data, for a total of 332 hours of speech.

Because the concept of anytime decoding is specifically oriented towards the needs of spoken dialog systems, we also evaluated the system on two dialog system corpora. The first is the NIST evaluation set from the DARPA Communicator project, while the second consists of data from the Let's Go system constructed at CMU. In both cases we used a combination of telephone bandwidth acoustic model data collected from a variety of sites as part of the DARPA Communicator project, as well as data collected from the Let's Go system since 2003. The training set for the Let's Go task contains both Communicator and Let's Go data for a total of 65 hours of speech, while the Communicator models are trained only on Communicator data, including extra corpora not used in Let's Go, for a total of 52 hours of speech. Baseline results after tuning for these corpora are summarized in Table 3.4.

Corpus	Domain	Word Error Rate	Real-Time Factor
Let's Go	Telephone dialog	33.2 ± 5.4	0.07 ± 0.009
Communicator	Telephone dialog	26.1 ± 5.0	0.07 ± 0.001
HUB4 devel	Broadcast news	24.2 ± 3.3	0.51 ± 0.050
WSJ0 (20k devel)	Read news	14.5 ± 1.7	0.38 ± 0.083
WSJ0 (5k devel)	Read news	7.24 ± 1.5	0.06 ± 0.001

Table 3.4: Baseline Speed and Accuracy Resuls (desktop), POCKETSPHINX 0.7

3.6 Summary

Through the POCKETSPHINX system, we have provided an example of a compact large-vocabulary speech recognition system and discussed the particular considerations involved in tailoring such a system for mobile applications. Since POCKETSPHINX is ultimately based on SPHINX-II, which is over 20 years old, it is important to note the shortcomings of the design and where it differs from state of the art systems. However, the goal of this thesis is not to rectify these well-known shortcomings, such as the lack of full N-Gram scoring in search noted in Section 3.4.1 and illustrated earlier in Figure 2.5, since their solutions are well studied and documented elsewhere. Neither is it the goal of this thesis to optimize the performance of particular components of the system, such as the poor performance of the language modeling code noted in Figure 3.4.

Instead, POCKETSPHINX provides a well understood and documented platform (at the source code level) used to evaluate new models, and which will be extended with new algorithms. The interaction between the multiple passes of search, in particular, is a major focus of Chapters 5 and 6.

Chapter 4

Domain Adaptive Speech Recognition

The stated goal of this thesis is the construction of an architecture for speech recognition which consists of a low-latency but also potentially low-accuracy "core" recognizer, on which partial recognizers are layered to achieve greater accuracy. While ultimately the goal is to construct this core recognizer specifically for this task, a first approximation to this is the problem, first discussed in [Ringger & Allen, 1996], of correcting the output of a recognizer built for one domain so as to be useful for a different domain. This can be thought of two ways - either as an *adaptation* problem, where the goal is to transform the source domain models into ones appropriate for the target domain, or as a *translation* problem, where the goal is to post-process the output to eliminate vocabulary mismatch and reduce the word error rate. In this chapter, we analyze both approaches in advance of the integrated approach to scalable speech recognition described in Chapters 5 and 6. This chapter largely describes unpublished work, although it has a similar goal to the implicitly supervised language model adaptation described in [Huggins-Daines & Rudnicky, 2007].

4.1 Domain Dependency

It is common in speech recognition, as well as in other kinds of language technology, to speak of the domain on which a speech system operates. Although this term generally means the real-world knowledge possessed by users of a system, it is also used in a more restricted sense to refer to the language over which the speech recognizer is defined. We frequently refer to, for example, the broadcast news domain, the flight information domain, or the robot control domain. As such, a domain encompasses both the vocabulary and grammar related to a particular task, but also more generally the style of speaking employed by participants, the acoustic conditions associated with the task, and the background knowledge that gives rise to said vocabulary.

Formally, we can define a domain simply as a discrete probability distribution $P_{\mathcal{D}}$ over the entire universe $\mathcal{L}$ of possible utterances. In-domain data, defined as a set of utterances $S_{\mathcal{D}}$, is any subset of the power set $2^{\mathcal{L}}$ of $\mathcal{L}$, whose empirical distribution $\tilde{P}$ is minimally divergent from $P_{\mathcal{D}}$:

$$S_{\mathcal{D}} \stackrel{\text{def}}{=} S \subseteq 2^{\mathcal{L}} : D(P_{\mathcal{D}}||\tilde{P}(S)) < \epsilon \tag{4.1}$$

Language models for automatic speech recognition are typically evaluated using the *test set perplexity*, which is the exponential of the cross-entropy from the empirical distribution of a sample of in-domain data to the language model. This quantity is directly related to the divergence between the in-domain data and the model:

$$pplx(S_{\mathcal{D}}||P_M) = \exp - \sum_{s \in S_{\mathcal{D}}} \tilde{P}_{S_{\mathcal{D}}}(s) \log P_M(s) \tag{4.2}$$

$$= \exp CH(\tilde{P}_{S_{\mathcal{D}}}||P_M) \tag{4.3}$$

$$= \exp D(\tilde{P}_{S_{\mathcal{D}}}||P_M) + H(\tilde{P}_{S_{\mathcal{D}}}) \tag{4.4}$$

Because the perplexity can also be interpreted as the average branching factor of the language model [Rosenfeld, 2000], a low perplexity results in a more compact search space in the speech recognizer. This

leads to improved runtime performance, or the ability to use a wider beam for pruning the runtime search graph, which may result in improved accuracy.

Aside from the well-known benefits of matched acoustic condition training for environmental robustness, we can describe the effects of in-domain data at the word level, the phonetic level, and the subphonetic level. At the word level, we wish to minimize the divergence $D(\tilde{P}_{S_\mathcal{D}}||P_M)$ as shown in Equation 4.4, which is non-negative and zero when $P_M = \tilde{P}_{S_\mathcal{D}}$. At the phonetic level, we need correct and domain-appropriate pronunciations for the lexical items in the language model. At the subphonetic level, the presence of characteristic phone sequences in the domain vocabulary and grammar leads to the corresponding context-dependent phones being better represented in the mapping of phones to states - this reduces the divergence between the output distributions of the tied states and the acoustic units which they model. For example, if the triphone A(B,C) is acoustically distinct from A(B,D), but it is not observed in the training set, the two may end up sharing output distributions in the resulting model. This results in a suboptimal acoustic model in the case where A(B,C) occurs frequently in the recognizer's input.

4.2 Domain Adaptation

Given the preceding characterization of the domain and the benefits of in-domain data for training the recognizer, how can we improve the performance of a recognizer on in-domain data in the absence of a suitable corpus of training data? Collecting appropriate data can be quite expensive and time-consuming. As well, in many practical systems, we face a "chicken-and-egg" problem, where the most desirable in-domain data is that which comes from users of the system, yet in the absence of a working system, it is not possible to collect this data.

At the subphonetic level, it is possible to "adapt" a model trained from multiple domains or a mismatched domain by using the available in-domain data to build the decision trees (or other structures) used for state tying, and retraining the tied-state model with the resulting trees. This technique has also been successfully applied to language-adaptive acoustic modeling [Schultz & Waibel, 2000]. However, this requires access to the original training data, and can be time-consuming since it potentially requires multiple iterations of training on said data. If sufficient in-domain data is available from a variety of speakers, speaker adaptation techniques such as MLLR and MAP can also be somewhat effective.

At the phonetic level it is simply necessary to include correct pronunciations for domain-specific vocabulary in the recognizer's dictionary. In general, these pronunciations can be guessed using a text-to-speech system with grapheme-to-phoneme rules for the desired phoneset and appropriate non-standard word processing [Sproat et al., 1999].

At the word level, the basic problem is to include domain-specific vocabulary in the language model. Since all automatic speech recognition systems use a closed vocabulary, they effectively assign zero probability to all utterances containing out-of-vocabulary words, leading, strictly speaking, to infinite perplexity. Empirically speaking, it has been estimated that each percent increase in the frequency of out-of-vocabulary words leads to a 1.2% absolute increase in the word error rate [Rosenfeld, 1995]. This may be a particular concern for systems with specialized vocabularies with idiosyncratic pronunciations, or with a large number of domain-specific compound words. For example, the Let's Go bus information system contains a large number of bus route numbers and stop names which are treated as single lexical items by the parser and back-end.

Our previous work has shown that simply achieving "vocabulary closure", by interpolating between a language model trained on a small amount of in-domain text and a background language model, is a highly effective way to improve performance on a new domain [Huggins-Daines & Rudnicky, 2007].

4.3 Language Modeling of Unknown Words

This gives rise to the first problem with which this chapter is concerned, namely, how to integrate these new words into the language model *with an appropriate probability distribution over them*. In the aforementioned work, a smoothed trigram model was trained from the domain-specific text and interpolated with the background language model with a low, empirically-determined weight. In addition, the component language models of the background model were re-interpolated with weights estimated on the domain-specific text. However, the use case which particularly interests us is the construction of initial language resources for mixed-initiative dialog systems [Bohus et al., 2007], where the background acoustic and language model are "out of the box" and potentially quite different from the target domain, and where little or no transcribed in-domain data is available.

Following the example of [Ringger & Allen, 1996] we can view this as a machine translation problem. However, rather than translating the output of the speech recognizer from one vocabulary to another, we are translating the language model from one vocabulary to another. More importantly, we do not have a body of transcribed in-domain data with which to train a translation model. In machine translation, the problem of data sparsity can be addressed by performing translation at a higher level of linguistic representation, such as by using a combination of hand-written and automatically learned transfer rules which operate on syntactically analyzed input [Lavie et al., 2003].

We propose to deal with data sparsity by translating at a *lower* level of representation. First, we note that it has been shown that humans are capable of recognizing nonsense syllables in isolation with high accuracy [Allen, 1994]. Furthermore, phonological regularity and similarity to existing words improves repetition of nonsense words [Glosser et al., 1998]. The specific hypothesis tested in this chapter is that, given that the domain language is derived from a human language (English, in this case), we can estimate the unigram distribution over words in the domain vocabulary by mapping them to similar words or word sequences in a generic background model for that same language. In effect, we attempt to predict the errors which result from using the generic language model on in-domain inputs and use this information to estimate the probabilities of the misrecognized in-domain words to which they correspond.

4.4 Experimental Data and Baselines

Our experiments used the Let's Go telephone bus information system [Raux et al., 2006]. We used two disjoint development and test sets consisting of recorded calls from users of the system. The first set of data consists of 1254 utterances collected between 2005-04-25 and 2005-04-28, a total of 32 minutes of audio. The second set of data contains 1591 utterances collected between 2010-02-06 and 2010-04-17, a total of 51 minutes of audio. Despite the elapsed time between the collection of the two data sets, they are quite similar in vocabulary and content - the cross-entropy from the second set to a bigram language model trained on the first is approximately 4 bits.

For all experiments, we used an acoustic model trained on the training data for the DARPA Communicator telephone travel reservation system [Levin et al., 2000]. There is very little vocabulary overlap between this training set and the target domain; however, they have similar acoustic conditions and speaker characteristics. The background language model is the standard US English trigram language model included in the POCKETSPHINX [Huggins-Daines et al., 2006] recognition system, version 0.6.1, which is based on a 5000-word vocabulary and is trained on the 1996-1998 HUB4 broadcast news corpus [MacIntyre, 1998]. The best performance of this baseline language model on the development set is a word error rate of 117%, with an out of vocabulary rate of 13.66%. Note that the word error rate is greater than 100% because it includes inserted words. The test set perplexity is not meaningful due to the large number of out of vocabulary words.

Given the context-free grammar used in the language understanding component of the dialog system, an alternative way of constructing a domain language model is to sample a large number of sentences from this grammar and build a smoothed trigram language model in the normal fashion. Using a sample of 100000 sentences results in a word error rate of 84%. One peculiarity of the Let's Go test data is that it contains a significant number of out-of-vocabulary words and out-of-domain utterances; even using the domain vocabulary, the OOV rate is relatively high at 2.23%. Finally, an important baseline is a simple unigram language model with a uniform distribution over the in-domain vocabulary, with which we obtain a best error rate of 77%, compared to a "cheating" unigram language model estimated from the testing set, which gives an error rate of 62% (perplexity 40). It is not surprising that the uniform grammar results in a lower error rate than the one generated from the domain CFG - the lack of explicit weights on rules leads to an implicit weighting based on a uniform distribution over rule productions rather than terminal nodes. Baseline results are summarized in Table 4.1. The "cheating" language model trained from the test set gives us a rough lower bound of the accuracy that can be obtained, while the uniform language model is the baseline on which we wish to improve.

dataset	2005		2010	
LM	WER	OOV	WER	OOV
HUB4 (3g)	117%	13.66%	92%	12.33%
CFG (3g)	84%	2.23%	81%	3.1%
Uniform (1g)	77%	2.23%	79%	3.1%
Cheating (1g)	62%	0	68%	7.21%

Table 4.1: Baseline Results for Let's Go vocabulary

Given that the error rate for the "cheating" unigram model is still astronomically high, it is fair to ask if there are other factors which make it difficult to achieve reasonable performance on this particular data set. One property of this data is that it includes a large number of unrecognizable utterances. However, ignoring them does not change the relative error rates of the various language models. More importantly, the data was collected in inconsistent real-world acoustic conditions, over a wide variety of fixed-line and cellular phones, and includes a large proportion of non-native speakers. Therefore we may consider it to be intrinsically difficult; the best result obtainable with in-domain acoustic and language model training data is around 45% WER.

4.5 Vocabulary Transfer

The POCKETSPHINX 0.6 recognizer contains an API for adding words to the language model, and therefore our first experiment simply used this facility to add the words from the domain vocabulary to the background model, creating an augmented language model and associated dictionary. The implementation of this API is quite naïve in that it simply adds the specified words to the unigram distribution in the language model with a score equivalent to some multiple of the original uniform weight (i.e. $\frac{\alpha}{V}$ where V is the number of words in the language model when it was originally loaded and α is the multiplier). No renormalization is done on the language model. This approach, predictably, improves over the background model, since the OOV rate is much reduced at 1.43%. However, aside from the now-defective probability model, the language model scores are very poorly estimated. The best error rate of 99% is achieved with $\alpha = 100$.

The poor performance of the previously mentioned methods of creating domain specific language models in comparison with a uniform distribution over the vocabulary can be attributed to interference from the background language model and to poor estimation of the probabilities of in-domain words and N-grams. The background language model, obviously, cannot recognize a large number of in-domain words. Its error rate is above 100% because it substitutes somewhat longer phonetically appropriate sequences of shorter words, leading to a high insertion rate. The vocabulary-closure model described previously is able to recognize domain-specific words, but it frequently misrecognizes words which are present in both the background and domain language model (such as STREET, PARK, DOWNTOWN, and BUS), since these are given a different (and lower) weight than words exclusively present in the domain vocabulary. Finally, the uniform language model performs better on street, bus, and stop names, but worse on general English words such as YES, NO, and HELLO.

4.5.1 Acoustic Transfer

In Section 4.3 we propose to predict the errors that result from using the generic language model on in-domain inputs and use this information to estimate probabilities for a domain-specific language model. We first attempted to do this directly using the output of the decoder on the domain vocabulary, an approach which has been used with some success for learning pronunciations of out-of-vocabulary words [Fetter, 1998]. However, we find ourselves in exactly the opposite situation; we have pronunciations but no speech data.

We resolve this problem by constructing synthetic inputs for the recognition engine from the domain dictionary. After converting the pronunciation of each word to a tied state sequence, we approximate the acoustic scores for each state using the KL divergence between this state's mixture weight distribution and all other states:

$$-\log P(o_s|q,\lambda) \approx D(W_q(\cdot)||W_s(\cdot)) \tag{4.5}$$

In the case of semi-continuous models, this is actually a very good estimate of the acoustic score, since all states share the same codebook of Gaussians. This can be verified by creating and decoding synthetic data corresponding to the reference transcripts for an evaluation set. On the November '93 Wall Street Journal development set, we obtain an error rate of 2.37% using the reference transcripts and 14.79% using the first-pass decoding hypothesis (which has an error rate of 11.24%).

Since the short synthetic utterances created by the vocabulary transfer contain no duration information, we removed self-loops from the transition matrices of the acoustic model. Without this, the decoder produced many empty hypothesis strings. To construct the language model, we then evaluated the resulting hypothesis strings with the background language model and used the resulting probabilities to construct a unigram language model. While the resulting model outperformed the baseline and vocabulary-closure models, it was unable to outperform the uniform language model, which indicates that it poorly estimates the unigram probabilities.

To address this problem, we first noted that, since we used the marginal probabilities of recognized word sequences as the unigram probabilities, we are implicitly penalizing long word sequences. These long word sequences are generally correlated with domain words that are very poorly matched by the background language model. However, since they presumably reflect the most likely errors which would be made by the speech recognizer, this is not sufficient reason to penalize them. Therefore, we instead use the geometric mean of the background word probabilities. In addition, we add a "squashing factor" α to further reduce the dynamic range of the language model probabilities:

$$\log P_{adapted}(w_D) = \frac{\alpha}{N} \log P_{generic}(w_L^{(1)}..w_L^{(N)}) \tag{4.6}$$

	WER (2005)	WER (2008)
Vocabulary Closure (3g)	99%	88%
Sum of Background LM Probs (1g)	86%	86%
Mean of Background LM Probs $\alpha = 1$ (1g)	82%	84%
Mean of Background LM Probs $\alpha = 0.1$ (1g)	77%	79%
Uniform (1g)	77%	79%

Table 4.2: Results for new language model built with acoustic vocabulary transfer

Best results for the "vocabulary closure" technique, the summation of background probabilities, and the weighted mean of background probabilities with $\alpha = 1$ and $\alpha = 0.1$ are shown in Table 4.2. Note that only by applying a heavy "squashing factor" to the language model are we able to achieve comparable results to the uniform model.

4.5.2 Phonetic Transfer

We next attempted to perform vocabulary transfer at the phonetic level. Here, we can simply create a finite-state automaton for each word in the target domain dictionary that recognizes all the alternate pronunciations (specified as phoneme sequences) of that word. We then compose this with an inverse transducer built from the background pronunciation dictionary. After projection and determinization this results in an automaton which recognizes the set of word sequences in the background dictionary which contain the same phonemes as the in-domain word, or the null set (in the case where no sequence of background words can be found to match the target word). We can then intersect this with the background language model to find the most likely sequence according to this model.

In the case where no match is found, we introduce an error model between the target domain automaton and the background dictionary, which expands each phone to a set of confusable phones with appropriate error weights. To construct this transducer we built a phoneme confusion matrix by running a phoneme loop decoder on the HUB4 training set, using dynamic programming to align the recognized phone sequences to the reference transcripts, and computing substitution, insertion, and deletion probabilities using maximum likelihood. This process was then iterated, recomputing the alignment using the estimated probabilities.

We encountered a similar issue with long word sequences as in Section 4.5.1, and used the same approach for estimating unigram probabilities using the geometric mean and squashing factor. Note that in both phonetic and acoustic transfer, we discard the "acoustic scores" (in this case the phone confusion matrix costs). We found that in the phonetic transfer case these were not helpful. Results of phonetic transfer experiments are shown in Table 4.3.

	WER 2005	WER 2008
Closure (3g)	99%	88%
Phonetic Mean (1g)	81%	79%
Phonetic Mean $\alpha = 0.2$ (1g)	75%	77%
Uniform (1g)	77%	79%

Table 4.3: Results of new language model built with phonetic vocabulary transfer

4.6 Error Modeling

In Section 4.3 we spoke of performing vocabulary transfer on the language model rather than the output of the recognizer. The rationale for this is rooted in the data sparsity problem we discussed earlier with respect to machine translation. The original statistical machine translation systems were based on the noisy channel paradigm, which separates the problem of modeling the probability of a translation pair into a translation model, which predicts the probability of an "underlying" word or phrase in the source language given a word or phrase in the target language, and a language model, which predicts the probability of a word sequence in the target language. Though more recent systems [Matusov et al., 2005] have largely abandoned the source-filter paradigm in favor of direct translation models using linear combination of a variety of input features, the principle of dividing up the modeling problem has been maintained.

The practical benefit of the noisy channel model is that the distribution over sentences given the translation model tends to be very noisy and uninformative, assigning non-negligible probabilities to a large number of incorrect and meaningless sentences. The language model can be estimated from a much larger amount of (target language only) data, and thus plays an extremely important role in guiding the decoder towards plausible outputs.

In the domain bootstrapping case, however, we do not have access to a large amount of target domain data, in fact, quite the opposite. Intuitively, it stands to reason that adding an additional source of error by using a domain-neutral acoustic model in the initial decode, then applying a translation model for which we also have insufficient data, can not improve on the performance of the language model adaptation technique already discussed. However, we do construct such a translation model in the course of both the acoustic and phonetic transfer algorithms. Therefore, it is not difficult to test this hypothesis, which we do in

Section 4.6.1.

If, on the other hand, we assume that a domain-specific language model exists, but for some reason cannot be used in the first pass of recognition, the error modeling technique becomes considerably more practical. In Section 4.7 we describe the use of such a model to resolve the mismatch between vocabularies in a lattice rescoring framework. This technique is particularly relevant because the situation where the target language model is not available in the first pass is equivalent to the separation of the recognition task into a "core" recognizer and peripheral recognizers which is the primary goal of this thesis.

4.6.1 Low-Knowledge Lattice Expansion

As mentioned above, in Sections 4.5.1 and 4.5.2 we construct a word transfer model which can also be thought of as a translation model. In the case of language model adaptation, we simply use this model to obtain one or more word strings in the background vocabulary which can be used to hypothesize language model scores for words which are only present in the target vocabulary. However, if we believe that this model accurately represents the errors made by the background language model when it encounters words in the target vocabulary, then it stands to reason that we can use this model to do *post hoc* correction of said errors.

In principle this is simply a different way of achieving the same result as we did by modifying the language model. Recall that in the abstract view of speech recognition as a series of cascaded composition operations on transducers [Mohri et al., 2000], we represent the output as $I \circ H \circ C \circ L \circ G$. We can assume that the domain words are present in the dictionary L, because if they do not appear in G they will simply be removed by the composition operation. Therefore, when adapting the language model we are effectively adding an extra transduction and projection to G, the language model (or grammar) transducer, corresponding exactly to the translation model described above, which we name E. In fact, this is exactly how the language model expansion is implemented in Section 4.5.2.

Because weighted finite-state composition is commutative, there is no difference between $I \circ H \circ C \circ L \circ (G \circ E)$ and $(I \circ H \circ C \circ L \circ G) \circ E$, though of course in practice the order of composition may have significant effects on runtime performance.

Hybrid FSG decoding

The difficulty in testing this with the POCKETSPHINX system is that we do not have a proper static network decoder which can be used with arbitrary finite-state networks such as the ones created by such a process. Therefore, we take a hybrid approach, where we use a unigram language model to generate an initial word graph, which is then pruned using arc posterior probabilities and modified using the error model. The resulting transducer is projected onto its output labels, resulting in a word-level weighted finite state grammar which is used in a second pass of decoding with the finite-state grammar decoder. In the case where a language model for the target domain is available, this grammar is expanded to give unique N-gram histories for each state. As shown in Table 4.4, this gives approximately equivalent performance to the baseline multi-pass POCKETSPHINX decoder on the Communicator and Let's Go datasets. In these experiments the output of the first pass was used directly to generate the FSG for hybrid decoding, which explains why the three-pass error rates are somewhat lower.

Data set	First-pass WER	Three-pass WER	Hybrid FSG WER
Let's Go (2005)	61.54%	53.37%	52.77%
Let's Go (2010)	54.34%	48.78%	51.76%
NIST Communicator	29.79%	24.19%	26.89%

Table 4.4: Results of hybrid FSG decoding compared to multi-pass N-Gram decoding

Training Data Versus Vocabulary Transfer

The results in Table 4.4 were obtained using trigram language models trained on the acoustic model training set for both Let's Go and Communicator. As in Section 4.4 and unlike in the deployed systems, no word classes were used in the language model. The training sets in both cases are fairly small, consisting of approximately 23000 sentences for Let's Go and 50000 for Communicator, where each sentence is typically one to five words. The vocabulary size was 1455 word types for Let's Go and 1998 word types for Communicator. It is clear that even with this relatively limited amount of data we can easily surpass the performance obtained by the vocabulary transfer techniques discussed in Section 4.5. In order to explore the amount of data required to outperform vocabulary transfer, we trained trigram language models for Let's Go with the 1455 word vocabulary and ten randomly selected subsets of a given size of the training data. Unseen words in the training set were assigned a uniform probability equal to the probability of singleton words. It turns out that with only 100 sentences of training data, we are able to outperform the best vocabulary transfer model. This is entirely due to the presence of explicit trigram estimates in the model, as shown by the comparison with unigram models, whose error rate falls much more slowly with increased training data, if at all.

Vocabulary Transfer Expansion

As mentioned earlier, the error model FST output from the vocabulary transfer algorithm can easily be used in a hybrid FSG decoding experiment. Since the vocabulary transfer process creates a mapping from target vocabulary words to background vocabulary word sequences, we must first invert this transducer. Since it creates only a mapping to single target vocabulary words we then perform Kleene closure on it. This transducer can then simply be inserted into the hybrid FSG experimental setup in place of the language model transducer that was previously used to expand word graph states.

In this experiment we tuned the weight given to the word posterior probabilities obtained from the original (background vocabulary) when constructing the FSG for second-pass decoding. In conjunction with this it is also necessary to tune the pruning threshold applied to these FSGs in order to retain approximately the same lattice density. There are no results reported for varying the pruning threshold with low posterior weights (less than 1.0). This is because, due to the large number of zero phonetic costs, the density of these lattices rapidly grows out of control without posterior pruning. In any case, the error rates are sufficiently high at low posterior weights that less pruning is unlikely to improve them.

The results of this experiment are not directly comparable to those in Table 4.3 because the phonetic confusion scores have been retained in the error model, which we have already shown to be unhelpful when estimating language model scores. In addition, the smoothed background language model scores were used to estimate the target language model scores. This can be achieved in the hybrid FSG framework by

discarding the phonetic confusion scores from the error model and left-composing it with the background language model.

There are a lot of "junk" words that show up repeatedly in the target lattices - basically these are short but not high-frequency words (many of them are partial words) which match very common phone strings. We have no probability distribution over target words and therefore no good way to score these down as they should be. Misspelled words in the LM training set also appear with great frequency in the target lattice since they are mapped to a large set of background words.

Although on the one hand this simply points to the need to clean up the language model, it is better understood as a consequence of the distribution over target words implied by the error model. Consider the words HOMEWOOD and HOMEWOO, the latter of which is a misspelling of the former, but neither of which is found in the background lexicon. These words expand to the background lists shown in Table 4.5.

HOMEWOO	HOMEWOOD
hole we're	home wood
whom we're	home would
home were	
home way	
home why	
home we	
home wow	
home would	
home wood	
home we're	

Table 4.5: Similar words can have vastly different numbers of phonetic expansions

If we discard the phonetic match costs, all of the alternative background word sequences on the left side of the table are considered valid for HOMEWOO, while only the two (actually correct) sequences "home wood" and "home would" are considered valid for HOMEWOOD. The error FST generated from these words is shown in Figure 4.1. We see that not only is HOMEWOO considered equally valid for the same word sequences as HOMEWOOD, but it also covers a number of other word sequences. The effect of this is that HOMEWOO becomes at least as likely to occur in the target lattice as HOMEWOOD. Therefore the task of discriminating them in the target domain falls completely on the acoustic model.

In addition, when the background language model scores are added to the error model as in the experiment described above, HOMEWOO ends up with a slightly higher probability (shown here as a lower weight). This is because the minimum weight of all sequences to which HOMEWOO transduces is lower than that of the two sequences "home wood" and "home would". Were we to use the log semiring and marginalize over all these sequences when composing the error transducer with the background lattice, this disparity in scores would be somewhat worse.

Again, although cleaning up the target lexicon and language model alleviate this problem somewhat, this is a general problem with the error modeling technique. Namely, the more expansions in the background language model for a given target lexicon word, the higher the likelihood assigned to it by the error model. In some cases, such as typos and partial words (e.g. HOMEWOO), this is exactly the opposite of the desired result. In other cases, it is less clear. However, the general effect is to assign greater likelihoods to words

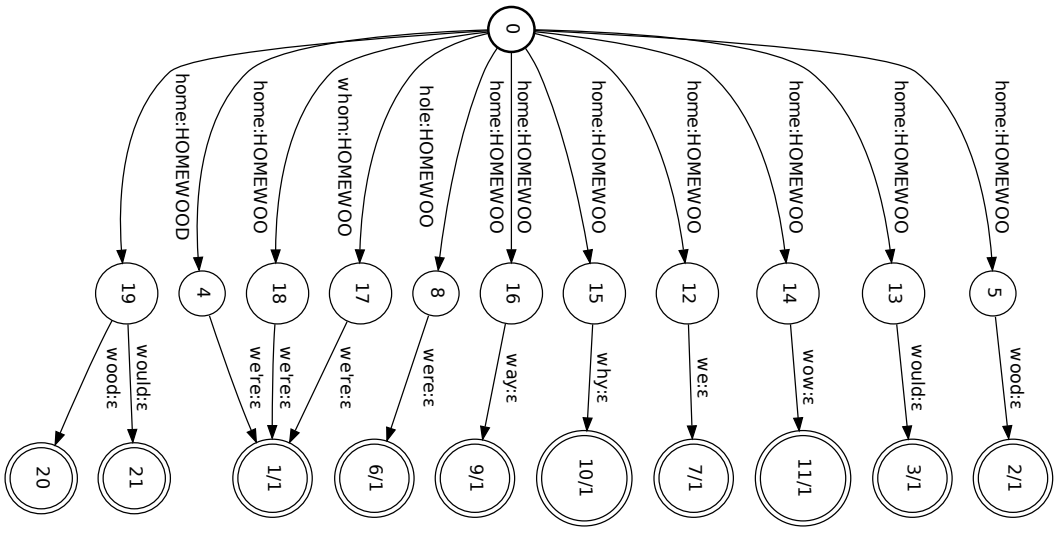

Figure 4.1: An error model for similar words will favor the wrong word

which have a greater divergence from the background lexicon, and if we take the background lexicon to be somewhat representative of the general English language, this is clearly an undesirable result.

How can we explain the gap in accuracy between the the experiments in Section 4.5.2 and these? The answer is simply that, despite the conceptual similarity noted above, the lattice expansion technique described here do not accomplish the same thing as the language model generation algorithms detailed earlier. With the adapted language models, it is considerably easier to deal with the incorrect estimation of target lexicon word probabilities. For one thing, the option to smooth with a uniform model exists, which is difficult to accomplish within the framework of finite state transducer composition laid out here.

More importantly, when using an adapted language model, there is no divergence at the lexical level in the first pass of decoding, resulting in a much more compact lattice. This is important because, as noted previously, it is necessary to prune the lattices fairly heavily when generating the second pass grammars. If the expansion model has little or no discriminative power, pruning the resulting lattice will inevitably remove correct hypotheses. Furthermore, the greater lexical divergence in the first pass of decoding also results in a considerably higher computational cost, which defeats any potential gains that might accrue from separating the decoder into core and domain-specific components.

For this reason, although it is possible to conceive of a more principled way of constructing the error model, we consider this approach to be fundamentally contrary to the stated goals of this thesis. The first pass of recognition cannot be treated as a "black box" if we wish it to run efficiently.

4.7 Summary

As mentioned earlier, if a language model for the target domain is available, the task is considerably simplified. This scenario is also somewhat less useful, since it is equally feasible, if not more so, to simply use this language model for the first pass of decoding. Nonetheless, in a system such as Let's Go, where multiple domain or state specific language models are used, there can be considerable overhead in both storage and computation involved with switching language models on the fly, or running multiple language models in parallel. It is for this reason that we have proposed the concept of a "core" recognizer with multiple lightweight vocabulary and domain specific recognizers layered on top of it. In this architecture, it is, of course, reasonable to assume that these lightweight decoders have complete domain or state specific models available to them.

The essential problem, then, becomes how to provide suitable input to these lightweight decoders such that they are not required to perform a full re-recognition of the original input. It is helpful to think about this in terms of the types of computation that would be involved in doing such a full re-recognition, and the ways in which they can be reduced or avoided given access to the output of a first pass of recognition using a generic decoder. In addition, we must consider the amount of memory, storage, and network bandwidth which will be consumed by the intermediate representations output by this first pass. For example, while we can avoid recomputing the acoustic scores by storing and reusing them, the size and bitrate of the resulting array is considerably larger than the original audio, which renders this approach infeasible in distributed configurations of the sort discussed in Chapter 6.

We can easily extend the decoding procedure developed in the previous section to include a full target language model, but as noted in the previous section, this fails to address the problem of computational complexity in the initial pass. In addition, it is not clear how the goal of anytime recognition can be achieved in this manner, since the composition and rescoring operations, while they can be done incrementally, are not time-synchronous in nature. In the next chapter, we describe a technique which is a natural outgrowth of the existing shortlisting strategy for multi-pass search, described in Chapter 3, which achieves the desired results without substantial run-time overhead. In Chapter 6 we describe a pipelined, multithreaded implementation of this technique which satisfies the anytime property, as well as a design for a distributed implementation.

Chapter 5

Vocabulary Expansion

In this chapter, we discuss the problem of expanding on the results of a compact recognizer to recognize a larger vocabulary. This is similar to the domain adaptation problem, since it relies on mapping from one vocabulary to another. However, in this case, we assume that a well-estimated language model for the full vocabulary exists. Furthermore, we assume that, given appropriate acoustic scores, we can achieve equivalent performance with a decoder using the full language model by reconstituting the missing vocabulary words in the output of the reduced language model decoder and rescoring.

Just as with the domain adaptation problem, we view this as a matter of translation. In this case, we are directly translating the word lattice, rather than attempting to adapt the language model. As shown in Section 4.6, this simplifies the problem considerably, since the language model can be relied upon to exclude wildly implausible outputs.

In Section 5.1, we demonstrate the effect of varying the vocabulary size on transcription and dictation tasks. We revisit some classic experiments by Rosenfeld [Rosenfeld, 1995] on the relationship between out-of-vocabulary rate and word error rate. In addition, we describe in greater detail the less well-studied relationship between vocabulary size, perplexity, and decoder performance.

5.1 Vocabulary Scaling

Automatic speech recognition systems are closed vocabulary systems by definition, in that they are incapable of recognizing words which are not present in the language or lexical model. As we have seen in Chapter 4, integrating previously unseen words into these models such that they can be recognized correctly is a non-trivial problem. This is primarily because ASR depends on a statistical model of word observation, which exhibits high variance in its estimates of low-frequency events. Unseen words are, by definition, low-frequency events, although given some prior knowledge of word equivalence classes, it is possible to achieve acceptable estimates of their distribution [Gallwitz et al., 1996, Kemp et al., 1996].

Nonetheless, the decoding algorithms for ASR, as detailed in Chapter 2, make the assumption that the vocabulary is known *a priori*. The computational complexity of adding new words to the decoding graph ranges from cheap and relatively simple, in the case of a flat lexicon decoder, to complex, in the case of a pre-compiled static network decoder based on finite-state transducers [Schalkwyk et al., 2003].

For these reasons it is important to choose an appropriate vocabulary for a recognition system. In the case of a well-circumscribed domain such as in most dialog systems, it is simply necessary to include all of the words which are necessary for the higher level components of the system to function. However, for the predominant mobile applications of search and dictation, as well as in other large-vocabulary tasks such as broadcast news recognition, the task of constructing a vocabulary is considerably more difficult. The reason for this is that the effective vocabulary of these domains is extremely large, continually changing and expanding, and is primarily composed of low-frequency types for which accurate density estimation is extremely difficult. Furthermore, the sources of vocabulary information, namely search queries and published text, can be extremely noisy. Finally, accurate pronunciation information is frequently unavailable, particularly in the case of foreign names and loanwords for which no standard pronunciation is attested.

The observation that large vocabularies change over time, and that occurrences of a given word tend to be "bursty" inspired the *cache language model* technique [Jelinek et al., 1991], where the base language model is interpolated with a dynamic "cache" based on a small, sliding window of recently recognized text. Subsequent work has focused on *language model adaptation* techniques, where a non-negligible amount

of text, either transcribed or drawn from the output of a previous pass of recognition, is used to directly adjust the parameters of the language model [Bacchiani & Roark, 2003]. In the unsupervised case, multiple iterations of adaptation and regeneration of these transcripts can be performed. More recently, *topic models* have been widely investigated as a way to reduce the amount of adaptation data needed and make adaptation more robust in the unsupervised case [Tam, 2009].

In all of the above-mentioned techniques, the vocabulary remains closed; only the probabilities of known words and N-grams are adjusted over time. The continuous expansion of computational capacity, combined with advances in search graph optimization and distributed language modeling, have made it possible for very large vocabularies (in the hundreds of thousands of word types) to be employed for hosted recognition tasks. It has been shown that the out of vocabulary rate has a linear effect on the word error rate, but due to the Zipfian distribution of word tokens, adding word types to an already large vocabulary has only a sublinear effect on the out of vocabulary rate [Rosenfeld, 1995]. Therefore, if word error rate is the only evaluation metric, there is no benefit to expanding the vocabulary past some (indeterminate, but large) threshold. For the standard HUB4 broadcast news evaluation, we see clearly that increasing the vocabulary beyond the top 10,000 word types has a minimal effect on word error rate, but a strong (nearly linear) effect on decoding performance, as shown in Figure 5.1. (Note that "xRT" in this and all future graphs refers to the real-time factor, which is the ratio between the amount of time required to decode an utterance and the length of the utterance). It is for this reason that this thesis focuses on optimizing the vocabulary size, particularly in the first pass of decoding which consumes the largest amount of computational resources.

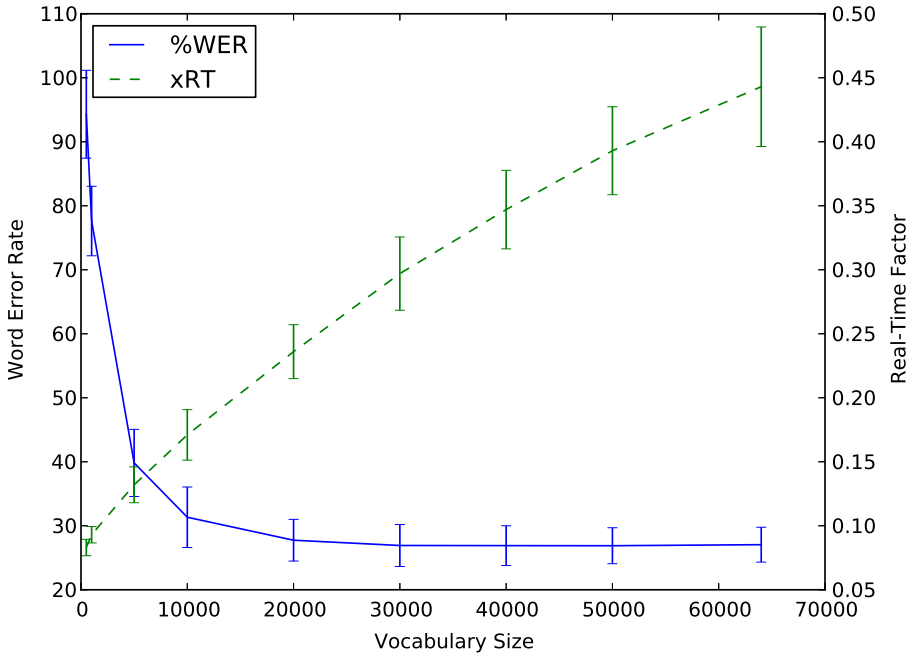

Figure 5.1: Increasing the vocabulary size affects speed more than accuracy

In the previous section, we ignored the role of the language model parameters when considering the

effects of scaling the vocabulary. This is because removing the least frequent words from the language model has little effect on the probability distribution of the remaining words, and little effect on the performance of the decoder. We can see this clearly by re-running the experiment from Figure 5.1 using the largest (64000-word) language model and instead varying the dictionary. This has the effect of removing the low-frequency words without re-estimating the probabilities. A comparison between the time-error curves of the reestimated (as in Figure 5.1) and non-reestimated models is shown in Figure 5.2. It is clear that for this evaluation set, there is little or nothing to be gained from correctly reestimating the language model probabilities from training data, which is a time-consuming process.

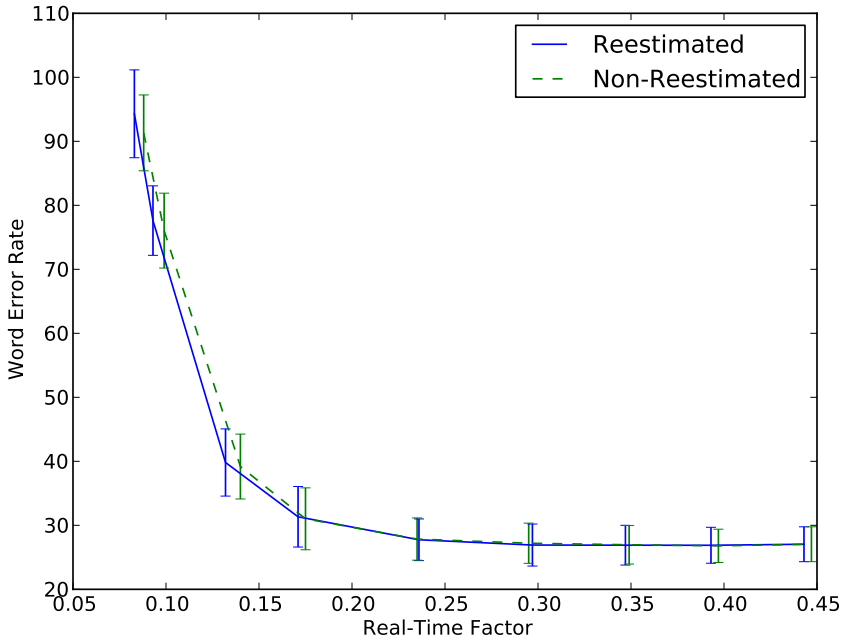

Figure 5.2: Re-estimation of language model probabilities after vocabulary reduction is not necessary

In [Brants et al., 2007], it was shown that for very large data sets, a defective probability model known as "Stupid Backoff" is nearly as effective as Kneser-Ney smoothing for machine translation. This is partly because the system in question is based on a linear combination of input features, which are not required to be probability distributions. Nonetheless, in speech recognition, language model scores, though they start out as properly normalized multinomial distributions, are modified by the language model weight and word insertion penalty such that they too no longer sum to one.

All of this indicates that there is some flexibility with respect to the parameters of the language model. However, when performing more sophisticated operations on the vocabulary which may affect higher-frequency words, the question of how to adjust the language model probabilities becomes fairly important. In particular, it may be more important for the *run-time performance* of the decoder than for the accuracy. While low-frequency words and their probability estimates may not strongly affect the accuracy of the decoder, they seem to be quite relevant to its runtime performance. We can see a hint of this effect in Figure 5.2,

where the performance of the non-reestimated model is consistently degraded at higher error rates.

It may be that this is simply an artifact of an incorrectly tuned language model weight. To test this, we varied the first and second-pass language weights for the 5000-word condition in both the reestimated and non-reestimated experiments. Because of the shortlisting strategy used by the POCKETSPHINX decoder, the first pass language model weight has a much smaller effect on accuracy, but a strong effect on performance - in general, it is best to choose the lowest first-pass language weight that gives acceptable performance. The second-pass language weight, by contrast, affects both performance and accuracy. In Figure 5.3, we plot word error rate versus real-time factor (larger is slower) for the non-reestimated and reestimated cases. We see clearly here that the non-reestimated model is nearly always slower at a given level of accuracy.

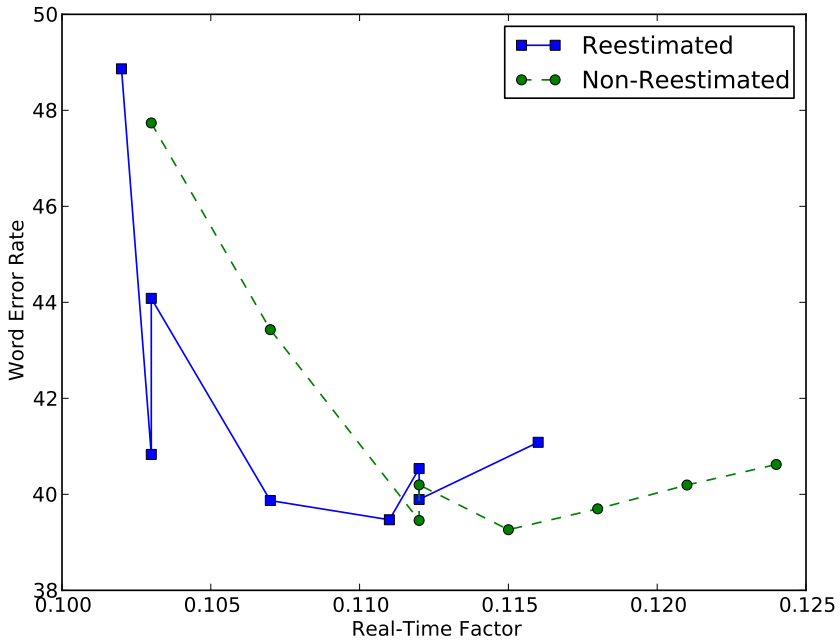

Figure 5.3: Re-estimation of language model weights after vocabulary reduction improves performance across different second-pass language model weights

By contrast, as shown in Figure 5.4, altering the first-pass language weight makes no significant difference in accuracy, but as expected, the reestimated model is consistently faster.

Another reason why re-estimating the language model parameters affects performance relates to the usage pattern of the language model in the decoder. First, as background, it is common when evaluating language models without performing speech recognition to report, in addition to the perplexity, the "N-Gram hit rates," which are the percentage of words for which an explicit N-Gram of a given order was found in the model. In general, we wish to increase the hit rate of higher-order N-Grams. In fact, this is related to the perplexity through the basic principle of information theory which states that *conditioning reduces entropy*.

However, evaluation of the language model on a single stream of text is completely different from how the language model is used during decoding. In the course of our research into efficient implementation

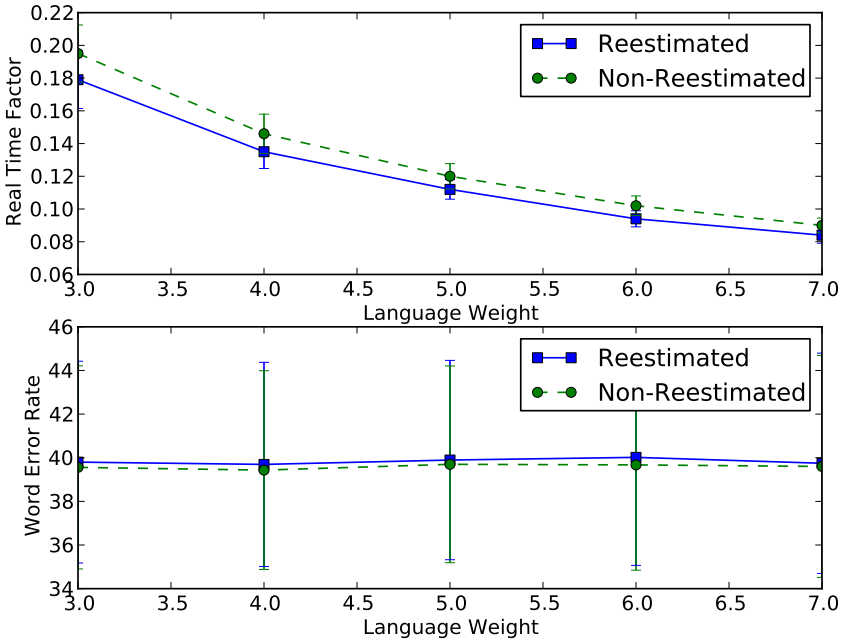

Figure 5.4: Re-estimation of language model weights after vocabulary reduction has no impact on accuracy but also consistently improves first-pass performance

of backoff language models, we measured hit rates for bigrams and trigrams in the decoder, which show a marked contrast with the hit rates reported when computing the test set perplexity from text, as shown in Figure 5.5 for the first 100 sentences of the HUB4 development set. This suggests that backoff to the unigram distribution is by far the most common mode of probability estimation at decoding time. This is in fact not surprising at all, since the decoder must evaluate all possible transitions for a given word exit, the majority of which result in unobserved N-Grams, where the percentage of unobserved (and thus backed-off) N-Grams increases with N. This problem can of course be mitigated somewhat with effective phonetic lookahead information. As well, a major advantage of WFST search, as described in Section 2.9 is that by "pushing" the language model scores towards the beginning of the static decoding graph at compilation time, it allows these unproductive word sequences to be eliminated from consideration earlier.

This has implications both for the implementation of backoff language models (it is important to make backoff operations as fast as possible) as well as for the estimation of backoff language model parameters. In particular, it implies that the accuracy of the *backoff weights* is particularly important for runtime performance, though it may have little if any impact on accuracy. To demonstrate this, we applied a constant factor to the backoff weights of the full 64,000-word trigram model. The effect on accuracy is shown in Figure 5.6.

Figure 5.6 shows two important findings with respect to the backoff weights. First, they can be heavily over-estimated (in this case, up to a factor of 10^2) without sacrificing any accuracy - the dashed line shows the accuracy using the unscaled, correctly estimated weights. Second, the penalty in accuracy for

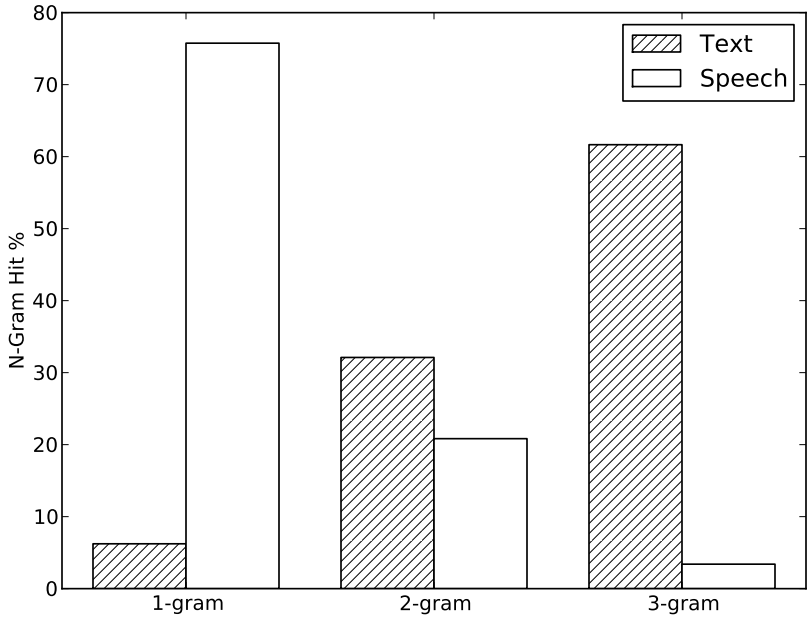

Figure 5.5: N-Gram hit rates are vastly different in decoding versus language model evaluation

under-estimating them is much greater than for over-estimating them - even a factor of 10^{-1} causes an approximately 5% relative increase in the word error rate.

The penalty for over-estimating the backoff weights is in the performance of the decoder. The explanation for this is actually quite simple - increasing the backoff weights has the effect of increasing the probability of low-frequency words, which makes them less likely to be pruned by the decoder at a given word transition beam width. This, in turn, increases the size of the search space. The lack of any corresponding increase in accuracy is due to the simple fact that no extra information has been obtained about these words.

5.2 Vocabulary Optimization and Expansion

Vocabulary expansion, here, refers to an analogous task to the domain adaptation task discussed in Chapter 4. Given a background language model and the output obtained by decoding an utterance with it, we wish to adapt the vocabulary of that output to a target vocabulary, so as to provide input for a subsequent pass of decoding or rescoring. In this case, however, we assume the existence of a suitably trained language model over the target vocabulary. More importantly, though, we have control over the background vocabulary. Therefore it is not necessary or prudent to consider the problem of vocabulary *expansion* in isolation from the problem of vocabulary *optimization*.

In Section 5.1 we hinted at how word error rate might not be the best metric for deciding what to put in

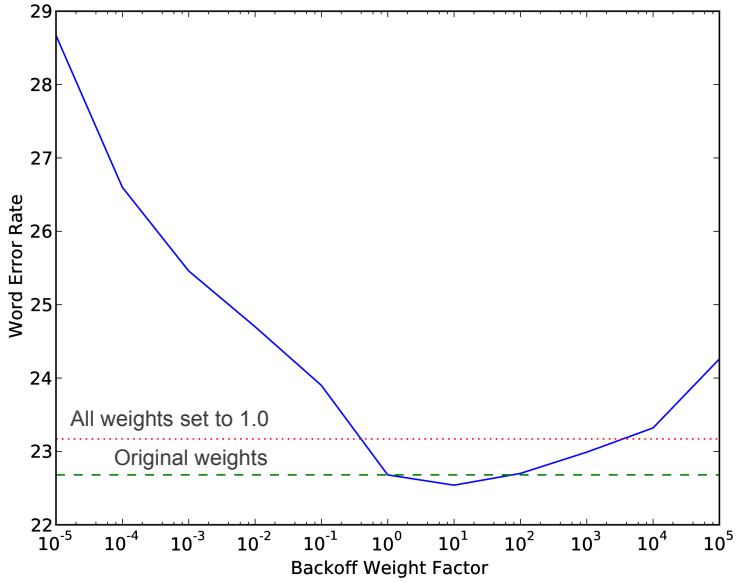

Figure 5.6: Over-estimating backoff weights, or even setting them to 1.0, has a minimal effect on accuracy

the vocabulary. We also said that increasing the size of the vocabulary past a certain point doesn't help word error rate. The question that we didn't address is whether using a simple threshold to determine the contents of the vocabulary, for example, taking the top 10000 word types, as shown in Figure 5.1, is the best strategy for determining the contents of the vocabulary.

It's trivially true that if you have a closed vocabulary recognizer, no plans to expand the vocabulary in subsequent passes, and out of vocabulary rate is the metric you wish to optimize, that a thresholding strategy is the optimal one. This, of course, depends on there being no major divergence between the distribution of word types in the training set and the data on which you evaluate the recognizer. As pointed out in [Rosenfeld, 1995], all other things being equal, this does not always translate into a reduced word error rate, due to the increase in perplexity which results from using a larger vocabulary with the same language model training data. The usual solution to this is fairly uninteresting: add more data.

The techniques described in this thesis raise a more interesting question, which is, as one might guess, what if the first two conditions are not satisfied? If we plan to expand the vocabulary later, is there a better criterion for selecting the first-pass vocabulary? Also, if there is a domain mismatch, as in Chapter 4, is it possible to construct a "neutral" vocabulary which produces output more suitable for vocabulary transfer and domain adaptation?

5.2.1 Compound Word Substitution

The easiest way to illustrate the concept of joint vocabulary optimization and expansion is with the case of compound words [Berton et al., 1996]. The vocabulary of the Let's Go system consists of a large number of these words. They have been given their own lexical items primarily because at the language understanding level they constitute individual tokens, therefore it was judged to be preferable to recognize them as single units if possible. This also allows more language model context to be taken into account when scoring them.

Consider the case where the word to be recognized is the now defunct Pittsburgh bus route 86B:

```
86B     EY T IY S IH K S B IY
```

This could equally be expressed as a sequence of three words:

```
EIGHTY    EY T IY
SIX    S IH K S
B.    B IY
```

There are several advantages to expanding these compound words to their constituents in the language model. First, the test set perplexity is considerably reduced. This can be explained intuitively by the fact that there are no longer a large number of bus routes occurring in the same context. It is also a simple consequence of there being more tokens and fewer types in the text, thus decreasing the denominator in the cross-entropy calculation. On the other hand, the language model score for bus names decreases due to the effects of smoothing and word insertion penalties.

Additionally, the lattice density, even when decoding utterances consisting only of the bus route tokens in question, is reduced. There is an obvious explanation for this as well, namely, that the component words (EIGHTY, SIX, B. in this case) remain in the language model even when a compound word is added, and thus occur frequently in the resulting lattices as competing word hypotheses. Not only does this slow down the decoder, it also creates problems for confidence estimation and discriminative training since the probability mass is spread out among multiple equally plausible (from the acoustic point of view, at least) alternatives. This suggests a reasonable strategy for finding words that should be eliminated from the vocabulary - if in decoding synthetic data built from a word you obtain too many equally valid hypotheses, the word is confusable. We return to this strategy later in this chapter.

In fact, this textual substitution is effective with the original compound-word based language model as well. The results of substitution on the output of the compound and non-compound language models are compared with the accuracy of the original compound language model in Figure 5.7.

Interestingly enough, performing textual substitution on the output of the compound LM also improves accuracy. This shows the problem with competing compound and non-compound hypotheses. However, the non-compound language model is not only more accurate but also significantly faster (14% relative in these experiments).

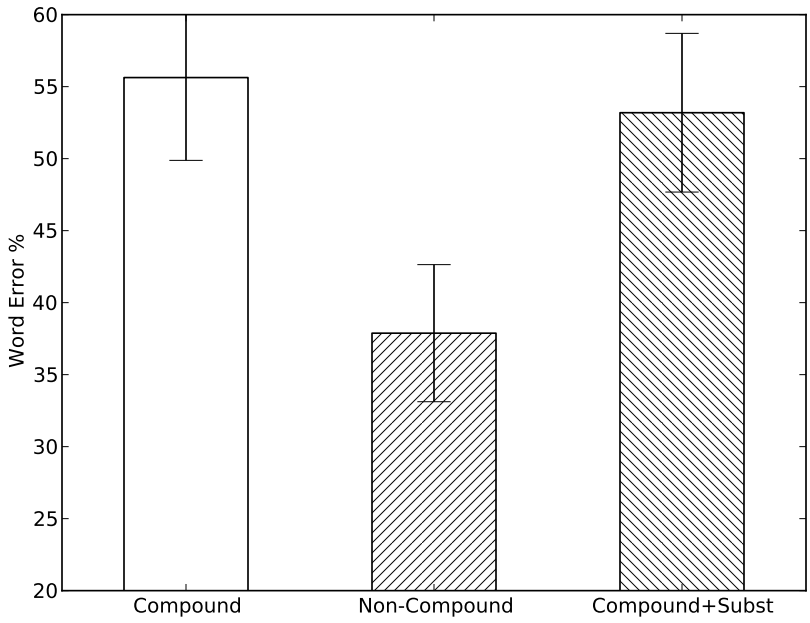

Figure 5.7: A language model without compound words is considerably more accurate on Let's Go, even after reconstructing them via textual substitution

5.2.2 Homophone Expansion

In the case describe previously, we are just blindly doing textual substitution on the output of the decoder, in cases where there is a direct mapping between a single sequence of 1-best hypothesis words and a single compound word. In addition, the "full" language model here is distinctly inferior to the reduced one, therefore we don't actually want to do any kind of rescoring or re-recognition using it.

Nonetheless we should be able to progressively relax these constraints to cover other cases of vocabulary mapping and expansion. Let us continue from the non-compound language model and proceed to eliminate more vocabulary words.

The simplest kind of vocabulary expansion is suggested by the existing POCKETSPHINX architecture. Recall that the second pass of recognition is a word-loop decoder which uses a dynamic vocabulary generated from a shortlist of unweighted word arcs. It is therefore trivial to perform vocabulary expansion on these word arcs by simply substituting a word from the output of the first pass of recognition with set of words which are considered likely to be recognized in the same location.

As with the substitution of compound words, we can think of this as a strategy of removing *redundancy* from the first pass language model. Alternately we can view it as deferring the work of disambiguation to the second pass. In either case, the ability to shift the computational load from the first to the second pass is very useful for parallel implementations such as the one discussed in Chapter 6. The disambiguation function of the language model is often illustrated through the example of homophones; introductory classes on automatic speech recognition typically make the claim, whether explicitly or not, that a major function

of the language model is to select the proper word for a given phone sequence based on context.

Homophony

Natural languages, particularly ones with restricted phonemic inventories, exhibit a large degree of *homophony*. For the purpose of automatic speech to text conversion, we define this as the phenomenon of multiple orthographic forms corresponding to a single phoneme sequence. Human listeners and transcribers are generally able to disambiguate these ambiguities - even in extreme cases such as the Mandarin word /yì/, for which the CC-CEDICT[CC-CEDICT, 2009] online dictionary lists 82 different orthographic forms (as of October 2009). For automatic speech recognition, however, the syntactic, semantic, and ontological information which humans draw upon to disambiguate homophones is not available. Therefore, even assuming a perfect acoustic model, homophony seems to constitute a fundamental problem of speech recognition.

Tables 5.1 and 5.2 show some examples of homophones in North American English, as described using the CMU variant of the ARPABET, and Standard Mandarin, as described using Hanyu Pinyin romanization. The frequencies included here were obtained by exact phrase Google searches on 2009-10-16, with the language restricted to English for Table 5.1, and Traditional and Simplified Chinese for Table 5.2 (counts are shown for both character sets). They are included to show that many pairs of homophonous words are equally common - that is, homophones are not limited to rare variants or technical terms. Note that here we have included stress in the definition of homophones for English, and tone for Mandarin, while these features are not always used in speech recognition.

Phones	Word	Freq	Word	Freq
S IY1 Z	seas	45.7m	seize	21.7m
M EY1 Z	maize	12m	maze	20.2m
S OW1 L	soul	243m	sole	111m
F EH1 R IY0	ferry	41.9m	fairy	87.1m

Table 5.1: Examples of homophones in English with roughly the same frequency

Pinyin	Hanzi	Gloss	Freq	Hanzi	Gloss	Freq
wúxiàn	無限	*unlimited*	15.1m	無線	*wireless*	15m
	无限		38.6m	无线		32.2m
jībiān	畸變	*distortion*	31.8k	機變	*improvisation*	50.5k
	畸变		757k	机变		411k
pīsà	披薩	*pizza*	1.8m	批薩	*pizza*	19k
	披萨		694k	批萨		314k

Table 5.2: Examples of homophones in Mandarin with roughly the same frequency

In continuous speech, the degree of homophony can increase dramatically due to the effects of coarticulation and neutralization - for example, in English, unstressed vowels are reduced, while in Mandarin, some tone distinctions are neutralized word-internally. Furthermore, distinctions in the official pronunciation may not exist in all dialects, such as the distinction between /AA/ and /AO/ in North American English, which is

lost for western and northern dialect speakers, and the distinction between alveolar and retroflex fricatives and affricates in Mandarin, which is lost for southern dialect speakers. For speaker-independent speech recognition, this can also be considered a form of homophony.

Finally, a last form of homophony, which is particularly widespread with respect to proper names in English, is the existence of multiple competing orthographic forms for the same word. As seen in Table 5.2, due to the phenomenon of *variant characters*, Mandarin is not immune to this problem. In particular, users of simplified characters are fairly confused about the appropriate way to write "pizza".

The good news is that homophony is much *less* of a problem than one might imagine based on introductory courses and lectures in automatic speech recognition. Looking at the incorrectly recognized words in the HUB4 1996 English Broadcast News development set and the incorrectly recognized characters in the RT04 Mandarin Broadcast News test set, we find that in both cases the number and frequency of confusion pairs which consist of exact homophones, where a confusion pair shares at least one pair of identical pronunciations, is very small. In the English data, the most frequent such pair is 'two/to' (both pronounced /T UW/), with 11 occurrences (out of 21,453 hypothesized words). In total there are 80 misrecognized tokens (47 types) which consist of exact homophones, contributing 0.35% to the total word error rate of 43.0%.

For the Mandarin test set, we ran both word and character alignments using the NIST `sclite` tool and a dictionary of 101,491 Mandarin words. In terms of word errors, there are 30 misrecognized tokens and 22 types which are exact homophones, out of 8,764 hypothesized words. Not surprisingly, the most frequent pair is 地/的 (both pronounced /de/ or /dì/[1]) with 7 occurrences. Homophones contribute 0.32% of the total word error rate, which in this test was 44.8%. The number for characters is higher, as expected - 173 misrecognized tokens and 134 types were exact homophones, out of 14736 hypothesized characters. Exact homophones contributed 0.98% to the total character error rate of 39.9%. If we ignore tone, the number of homophone errors rises to 73 word tokens (55 types) and 410 character tokens (324 types).

This result is counter-intuitive in two respects. First, we would expect the number of homophone errors in recognition to reflect the prevalence of homophones in the dictionary, but this is not the case - in fact, homophone errors in recognition are quite rare. Second, the prevalence of homophone errors is roughly the same between Mandarin and English, despite the widespread belief that homophony is rampant in spoken Mandarin. However, it is not difficult to explain. A majority of the homophones in the English dictionary are orthographic variants, usually of proper names, which are typically collapsed to a single canonical form in the language model. The remaining homophones tend to be syntactically, semantically, and distributionally distinct, which we have verified by manually annotating a random sample of the dictionary.

This is reflected by the observation that the most common homophone pair in the Mandarin test set, when tone is ignored, is bàodào/bàodǎo or 报道/报 导, both of which mean "report" (the first is a noun, the second is a verb). Of the other toneless homophone errors, none occurs more than once, and the majority consist of similar pairs of semantically related words. This seems to indicate that, at the word level, tone in Mandarin functions similarly to stress in English, in that it is not obligatory for disambiguating words in context, and that in some cases it may be advantageous to omit it from the dictionary, such as when there is insufficient acoustic model training data to obtain full coverage of all phoneme/tone combinations.

By an examination of the facts and effects of homophony, we conclude that N-Gram language models and trigram acoustic models, as imperfect as they may be, are effective at disambiguating homophones and even near-homophones (such as in Mandarin when tones are omitted). We proceed to examine how we can

[1] This alternate pronunciation is somewhat rare in the case of 的.

use this fact to optimize the first pass of recognition and scale the vocabulary.

Homophony in the Decoder

However, the ability to disambiguate homophones through the language model requires extra work from the decoder. We stated previously that in a tree-structured decoding network, the identity of a hypothesized word is not known until the last phone, which means that in order to calculate language model scores exactly, it is necessary either to track multiple language model states simultaneously in the decoding network, or to copy the lexicon tree such that each copy possesses a unique N-Gram history. For homophones this problem is exacerbated since the identity of the word remains ambiguous even after the last phone has been recognized. This increases the number of entries in the lattice and the number of language model states which must be propagated to successor words in decoding.

In the static lexicon tree strategy used by the first pass POCKETSPHINX decoder, the optimal language score is not available in any case since language model states are not propagated through the lexicon tree. The effect of this is that only one homophone will ever be considered when calculating successive words' language model scores, and it may be the "wrong" one. This leads to an obvious conclusion which is that, since treating homophones as separate words in the first pass of recognition is at best not useful, we can remove them from the first pass language model and use vocabulary expansion to recover the alternatives for the second pass, which is better equipped to disambiguate them.

Language Model Pruning

What is meant exactly by "removing" homophones from the first pass language model? The most obvious answer to this question is to re-train the language model by collapsing homophones in the input text. However, recall from Section 5.1 that there is a considerable amount of flexibility to modify a backoff language model without adversely affecting the accuracy of the decoder. In fact, in the case where we wish to *merge* words rather than adding or removing them, we can easily update the N-Gram probability estimates without the need to store unnormalized counts, as was suggested previously. The algorithm for merging homophone sets is described in Listing 5.1.

This algorithm relies on a trie representation of the N-Gram model, where each individual N-Gram is represented by a head word and a list of successor N-Grams. In this data structure, each N-Gram node contains the conditional probability of its head word given the words of each node on the path from the root, along with the backoff weight to be applied to any successor N-Grams not explicitly represented in the trie.

The `merge_classes` algorithm takes as input a set of equivalence classes (which in this instance are sets of homophones). It traverses the N-Gram trie in preorder, ensuring that any successors of a given node will have been touched before it. For each node n with history h in this traversal, any equivalence classes in the successors of h, n are merged. This is done by creating a new pseudoword p (which can also simply be one word out of the equivalence class C) whose conditional probability $P(p|h, n) = \sum_{c \in C} P(c|h, n)$ is the total probability of all class words in the successor distribution of h, n. The successor distribution of p is constructed by linearly interpolating the successor distributions of $c \in C$.

Clearly, there is a small problem with this algorithm, in that the successors of $c \in C$ are not recursively merged; instead, the successors of h, n, p consist of those of one arbitrary input successor c. Therefore, some higher-order N-Grams will be lost in the merging process. In practice, this is only possible for 3-grams and

```
def add_weighted_successors(dest, src, weight):
    if src.no_successors(): return
    lw = math.log(weight)
    for ss in src.successors():
        if dest.has_successor(ss.word):
            ds = dest.get_successor(ss.word)
            ds.log_prob = logadd(ds.log_prob, lw + ss.log_prob)
        else:
            ss.log_prob += lw
            dest.add_successor_ngram(ss)

def merge_classes(ng, classes):
    if ng.no_successors(): return
    for succ in ng.successors():
        merge_classes(succ, classes)
    for pseudo, words in classes.iteritems():
        word_succ = []
        succ_prob = 0
        for w in words:
            if ng.has_successor(w):
                succ = ng.get_successor(w)
                p = math.exp(succ.log_prob)
                word_succ.append((succ, p))
                succ_prob += p
        if len(word_succ) == 0: continue
        elif len(word_succ) == 1:
            word = word_succ[0][0].word
            pseudo_succ = word_succ[0][0]
            pseudo_succ.word = pseudo
            ng.del_successor(word)
            ng.add_successor_ngram(pseudo_succ)
        else:
            pseudo_succ = ng.add_successor(pseudo)
            pseudo_succ.log_prob = math.log(succ_prob)
            for succ, p in word_succ:
                weight = p / succ_prob
                add_weighted_successors(pseudo_succ,
                                        succ, weight)
                ng.del_successor(succ.word)
```

Listing 5.1: The homophone merging algorithm

higher, and the N-Grams in question are still represented by backoff 2-grams, and therefore the effects of this on a trigram model are minimal. After merging, the backoff weights for the model are recomputed based on the Katz backoff equation.

What does this imply for speech recognition performance and accuracy? Since, as discussed earlier, when all else is equal, decreased perplexity leads to decreased computational complexity, we should expect that merging homophones in particular will improve the speed of the decoder. One can think of this as resulting from decreased confusability in the language model coupled with no increased acoustic confusability (since the set of phonetic word forms has not changed).

To evaluate this, we first created a merged language model and applied the same class merging to the test corpus, then evaluated the perplexity, as described above. In order to evaluate the speech recognition performance, we performed two tests. In the first, we simply measured the word error rate against the class-merged test corpus transcripts. In the second, we applied the inverse class transformation to the decoding transcripts, generating a lattice of alternate hypotheses, which we then expanded with the full language model and rescored using the N-Gram expansion algorithm described in Section 6.3.2. Since, in theory, we are doing one-to-one merging of exact homophones, the acoustic scores are irrelevant to the rescoring process. However, due to the presence of multiple pronunciation variants in the input dictionary, we must take care not to merge words whose pronunciations are dramatically different, such as, for instance, "AM" (as in NINE AM) and "AM" (as in I AM). Possible solutions to this problem are investigated below. For the purposes of this experiment, we simply ensure that the pronunciation in the dictionary for a pseudoword is consistent with that of its equivalence class.

The perplexity, pseudoword error, and rescoring results for merging homophones in several test corpora are shown in Table 5.3. The results here are decidedly mixed - in the dialog corpora, merging homophones has little or no effect on accuracy or perplexity, while in the WSJ dictation corpora, it incurs a small increase in perplexity and a significant increase in error rate. In order to verify that this increase in error was not due to the trigram problem noted above, we tested WSJ with the bigram as well as trigram model and obtained a similar result. Finally, however, on the most complex of these tasks, the 64,000-word broadcast news transcription task, the merged model performs as well as the unmerged one after rescoring. Runtime performance is not shown here, but on the 3GHz Core 2 Quad machine used in testing, the merged model also runs 13% faster (0.35 xRT versus 0.4 xRT).

Corpus	Perplexity		WER		
	full	merged	full	merged	rescored
Communicator	50.7	16.1	23.89%	24.36%	22.03%
Let's Go 2005	14.5	14.5	53.86%	55.77%	53%
Let's Go 2010	16.9	16.8	50%	50.12%	50.15%
WSJ 5k devel	62.4	64.8	7.29%	8.91%	8.98%
WSJ 5k devel (bigram)	110.3	112.9	9.88%	11.36%	11.73%
WSJ 20k devel	62.4	64.8	14.31%	19.80%	20.13%
HUB4 64k devel (F0)	237.2	241.8	27.38%	30.54%	27.75%

Table 5.3: Results of the homophone merging algorithm on several test sets

```
def greedy_reassign(classes, order):
    classes.collapse_alternates()
    ordered_classes = classes.items()
    ordered_classes.sort(cmp=order)
    classmap = dict()
    for pseudo, classwords in ordered_classes:
        if pseudo in classmap:
            classmap[pseudo] |= classwords
            del classes[pseudo]
            continue
        bogus = set()
        for w in classwords:
            if w in classmap:
                bogus.add(w)
            else:
                classmap[w] = classwords
        classwords -= bogus
        if len(homos) < 2:
            del classes[pseudo]
```

Listing 5.2: Greedy reassignment algorithm for overlapping classes

Finding Homophones

The algorithm for efficiently finding homophones is relatively trivial and as such is not described here, as it is subsumed by the suffix merging algorithm described in the next section. However, there is a slight complication in that many words in the language model may have multiple pronunciations, resulting in equivalence classes that are not disjoint. There are two main ways we can address this problem. The first is to enforce a single pronunciation per word in the dictionary, an approach which, if done intelligently, does not significantly degrade accuracy and can even increase it on some tasks [Schultz & Yu, 2003]. However, optimal use of a single pronunciation dictionary requires not only that the dictionary be optimized with respect to speech data, but that the acoustic model also be retrained with the updated dictionary [Hain, 2005]. This requires both a considerable amount of computation as well as the original acoustic model training data, which may not be easily available.

The second approach is to attempt to find the "correct" class for a word which, due to having multiple pronunciations, is present in multiple classes, and remove it from all other classes. This is a similar problem to determining the appropriate form for a single pronunciation dictionary. We approach this as a post-processing step over the homophone map created by the finding algorithm. As a first approximation, we can simply greedily assign words to the "best" existing class according to some ordering over classes. This algorithm is shown in Listing 5.2.[2]

The `greedy_reassign` algorithm works by first collapsing alternate pronunciations within each

[2]Note that |= and −= are the Python operators for set union and subtraction, respectively.

class. Next, an ordering is created over the classes. We keep track of the "best" class for each word in the associative array `classmap`. For each class, if this class' pseudoword is already a member of some "best" class, we merge its classwords with that class and delete it from the set of classes. Otherwise, for each class word, we remove it if it is already a member of some "best" class, and otherwise, since this class comes before any other classes containing this word in the ordering, we set this class as the "best" class for this word. Finally, any classes containing a single item or no items are removed.

To evaluate this versus a single pronunciation dictionary we used a 64,000 word language model trained on the WSJ0+1 language model training set, since a reliable single pronunciation dictionary was available for it. As above we measured the test set perplexity and the word error rate on transcripts with the given homophone mapping applied to them. Two orderings were considered - in one, we assign words from overlapping classes to the largest class, while in the other, we assign them to the smallest class. Since the number of overlapping words is relatively small, we do not expect a large difference in perplexity or error rate between these. A third option is to assign words to classes based on phonetic similarity.

The results show that assigning duplicate words to the smaller class is the clear winner in terms of accuracy. This is because, rather than consolidating words which may have divergent pronunciations into a few larger and larger classes, it redistributes these words among classes, thus implicitly lessening the within-class confusability.

5.2.3 Error-Driven Expansion

The name of the `merge_classes` algorithm presented in Section 5.2.2 should indicate that it is useful for more than simply merging homophones. In fact, any disjoint set of equivalence classes can be merged by the algorithm. Given our stated goal of reducing the computational load of the first pass of decoding while allowing the resulting errors to be easily recovered from in future passes, we can consider strategies for optimizing the vocabulary to satisfy these criteria.

Without the possibility of error correction, the frequency cutoff strategy employed for the experiments in Section 5.1 is optimal for in-domain input data. However, it is mostly useless in the case where vocabulary expansion will be performed. This is because not only does it not define any relation between the removed words and those still in the lexicon, but the removed words themselves, being low-frequency words, are less likely to be semantically or phonetically related either to each other or to those words which remain. For this reason, our initial attempts to define expansion classes over a vocabulary reduced in this fashion were largely unsuccessful.

In an initial "cheating" experiment, we simply used correction pairs collected by aligning word lattices generated using only the high-frequency words to the reference transcripts. For the compact language model we used the top 5000 words of the WSJ language model training set, while the expanded language model used the top 20,000 words. Some example mappings found by this method are shown in Table 5.4. It seems that they fall into five main categories, as shown here. The first category consists of homophones or near-homophones, which are generally also semantically related, as in the pair `PRODUCTION / PRODUCTIVE`. The second consists of unrelated words which are phonetically close. Typically these share a common phonetic subsequence, as in `EXCEPTION / ACCEPTS`, or a common metrical structure, as in `ARABIA / CARIBBEAN`. The third category is seemingly unrelated words. The fourth category is splits, nearly all of which are well-motivated phonetically. Finally, function words, which are typically short, highly confusable acoustically, and highly probable according to the language model, are typically align to

a variety of other words.

Incorrect	Correct
WORKERS	WORKERS'
PRODUCTION	PRODUCTIVE
ACCEPTABLE	UNACCEPTABLE
FALLING	FOLEY'S
EXCEPTION	ACCEPTS
ARABIA	CARIBBEAN
OPPOSITE	PARADOXICAL
DOWNTURN	GILBERT
STRESSED RATINGS	FRUSTRATING
WILL TRADED	INFILTRATED
BASED TRUST	DISTRUST
A	ENGAGE, CONDITIONER, ACQUIRER, REBUILD, TO, HER, ORDER, STABLE

Table 5.4: Examples of confusion pairs found when aligning output of the 5000-word decoder with that of the 20000-word decoder

Immediately we can see that these are not useful equivalence classes for expansion. And, indeed, when the confusion pairs found using the lattice alignments were used to augment the shortlists when decoding using the full 20k vocabulary, they completely failed to generalize beyond the set used to train them, as shown in Table 5.5. Another problem, not shown here, is that expansion using this model massively increases the size of the first-pass lattice. This is because the expansion pairs in the "model" include spurious mappings from function words to a large number of confusion words. This can easily be solved by excluding a small set of stopwords from the expansion process. This results in lattices which are roughly the same size as the unexpanded ones, and in the cheating experiment detailed above, has a minimal effect on the resulting error rate (12.39% versus 12.24%).

First Pass	WER (devel20k)	WER (test20k)
20k bigram	12.18%	12.08%
5k bigram	24.20%	24.96%
5k expanded	12.24%	24.85%

Table 5.5: Second pass results for error model driven vocabulary expansion show that it fails to generalize

In an attempt to produce a more general model for the 5k to 20k task, we implemented a simple correction model using phonetic edit distance. The edit distance is the minimum number of insertions, deletions, and substitutions needed to transform one string of phones into another. For each *hypothesis word* in the 5k vocabulary, we generated a ranked list of *candidate words* for correction from the non-overlapping portion of the 20k vocabulary. The edit distance for each candidate was normalized by the number of phones in the hypothesis word, and those candidates under a threshold were retained for use in expansion. The results of two-pass decoding using this expansion model are shown in Table 5.6.

For this experiment, an identical, fixed cost was assigned to all substitutions, as well as to insertions

Threshold	WER (devel20k)	WER (test20k)
0.0	24.20%	24.96%
0.1	24.01%	24.92%
0.2	22.27%	23.29%
0.3	20.37%	21.31%
0.4	18.15%	18.83%

Table 5.6: Adding phonetic edit distance to the error model improves generalization up to a point

and deletions. This creates the problem that, particularly for short words, a large number of candidate words have an equal ranking. As well, the degree of acoustic similarity between substitution pairs is not considered in the model, and thus the ranking of candidates is imprecise. The effect of both of these problems is that the candidate lists must be made longer in order to achieve the desired error correction effect, leading to very large expanded lattices. One solution to this is to use costs estimated from a phoneme confusion matrix, as in [Pusateri & Thong, 2001].

The results in Table 5.6 show that lattice expansion using phonetic distance measures is effective in reducing the second-pass error rate. However, it does not allow us to achieve comparable results to single-pass decoding using the full vocabulary and detailed acoustic model. Using a threshold of 0.4 to expand the first-pass lattice, the error rate is still roughly 50% higher than it would be had we run a first-pass search using the full vocabulary.

Hypothesis	Candidates
WORKERS	WORKER'S, WORKERS'
PRODUCTION	PRODUCTIONS, DEDUCTION, PERFECTION
ACCEPTABLE	ACCESSIBLE, SUSCEPTIBLE, UNACCEPTABLE, EXCEPTIONAL
FALLING	(none)
EXCEPTION	EXCEPTIONS, INCEPTION, DECEPTION, EXCEPTIONAL, RECEPTION
ARABIA	ARABIA'S, ARABIAN
OPPOSITE	(none)
DOWNTURN	DOWNTURNS

Table 5.7: Examples of confusion sets generated by the phonetic edit distance based error model generation algorithm

While some of this effect can be attributed to the shortcomings of the correction model noted above, it may also be the case that there are broad classes of errors which cannot be corrected by this model. In fact, in comparing the candidate lists generated by this model with the taxonomy of confusions shown in Table 5.4, it seems that only the first class, namely near-homophones, are captured with any regularity. Candidate lists for some of the same words, using a distance threshold of 0.3, are shown in Table 5.7. The second class of errors from Table 5.4, namely words with similar phonetic structure, may be more easily captured by using an acoustically sensitive distance measure.

Aside from near-homophones, the class of confusions consisting of word splits in the hypothesis is

clearly phonetically motivated. Although we made no effort to address splits in the "cheating" experiment, splits were captured by the fact that all of the split hypothesis words were ultimately aligned to the correct word. Therefore, in the expanded lattice, they were all present in the short list throughout the duration of the original word and were thus available to the recognizer. As well, from the "cheating" results, it appears that many of the words which participate in splits are exact subsequences of the longer word for which they are confused.

From the more recent work detailed in Section 5.2.1, we know that it is effective to simply decompose these compound or pseudo-compound words and later recombine them, either in a second stage of recognition, through textual substitution, or by lattice rescoring. We also know, from Section 5.2.2, that homophones and near-homophones can be collapsed and expanded without a major loss of accuracy. Therefore, it seems that a better strategy for automatically finding useful expansion classes would be by specifically including only these types of approximations which are known to be recoverable. Since, unlike in the domain adaptation problem discussed in Chapter 4, we assume the ability to choose the source vocabulary, it follows that the reduced vocabulary should also be constructed using these effective approximations. In a way, this creates an analogous situation to the "cheating" experiment without actually "cheating" - if these reversible approximations are not tied to any particular data set, they should generalize to any data set within the domain of the target language model and vocabulary. While we may not be able to reduce the size of the base vocabulary as much as in the error-driven experiments above, this is irrelevant as long as the reduced vocabulary improves first-pass performance.

5.2.4 Prefix Expansion

If removing low-frequency words is an optimal strategy for minimizing the increase in word error rate in a single pass system, what is an optimal strategy for maximizing the decrease in computational complexity? If the goal is to minimize latency, simply removing long words will achieve this goal, since as we discuss in the next chapter, the amount of lookahead required to incrementally generate a final hypothesis is proportional to the longest word in the vocabulary.[3] This is also not an unreasonable strategy from the point of view of first-pass accuracy, since word length is inversely correlated with frequency.

Seen in this light, the compound splitting discussed in Section 5.2.1 is simply a special case of removing long words from the vocabulary, where the decision on which words to truncate is constrained to manually annotated compound words. In all cases, when we remove words from the lexicon we are, either implicitly or explicitly, creating equivalence classes, as noted in the previous section. In order to avoid the requirement of domain knowledge to perform compound splitting, we would like to determine splittable compounds automatically. In addition, we have observed that short suffixes are frequently misrecognized in English; as such, a word and its inflected versions form a natural equivalence class. Finally, previous work on out-of-vocabulary word detection [Schaaf, 2001] has shown that the false positive rate can be reduced by modeling the prefixes of out-of-vocabulary words using regular phones, as if they were normal words.

Since earlier we noted that homophones are good candidates for merging because they *are not* confusable, why does it make sense to merge inflected words which *are* confusable? The key is that in the first case, homophones are *acoustically* confusable, but not *linguistically* confusable, while in the second case,

[3] In WFST-based systems, *label pushing* allows word hypotheses to be emitted before reaching the final state. However, this requires a post-processing step to assign the correct segmentation and acoustic score to these words.

```
def mark_distance(trie):
    for node in trie.postorder_traverse():
        if node.children:
            node.distance = max(x.distance for x in node.children)
            if node.words:
                node.distance += 1
        else:
            node.distance = 1

def propagate_classes(trie, level):
    classes = dict()
    for node in trie.postorder_traverse():
        node.equiv = list(node.words)
        if node.distance <= level:
            for x in node.children:
                node.equiv.extend(x.equiv)
                del x.equiv
        if node.distance == level \
                and node.words \
                and len(node.equiv) > 1:
            classes[node.words[0]] = node.equiv
    return classes

def build_prefix_map(indict, level=1):
    trie = PhoneTrie(indict)
    mark_distance(trie)
    return propagate_classes(trie, level)
```

Listing 5.3: The prefix class building algorithm

inflected words are linguistically confusable but not acoustically confusable. In both cases, information exists in our models to disambiguate the words, we are simply choosing to defer its application until the search space has become more tractable. In the case of word classes which are both acoustically and linguistically confusable, such as we might obtain by removing words randomly or based on frequency alone, we have no such information.

This leads us to a simple algorithm which can be parametrized to accomplish all of the merging tasks previously described. This algorithm is based on creating a hierarchy of equivalence classes of common prefixes. Given a pronunciation dictionary represented as a trie, it has linear time complexity. This algorithm, `build_prefix_map`, is shown in Listing 5.3.

The functioning of the prefix class building algorithm can be illustrated on a miniature vocabulary consisting of words beginning with the phone sequence `/HH IH P/`. A trie representing this vocabulary is shown in Figure 5.8 after the initial `mark_distance` pass. Each final node is simply marked with the

maximum number of output nodes between it and a word-final node, inclusive.

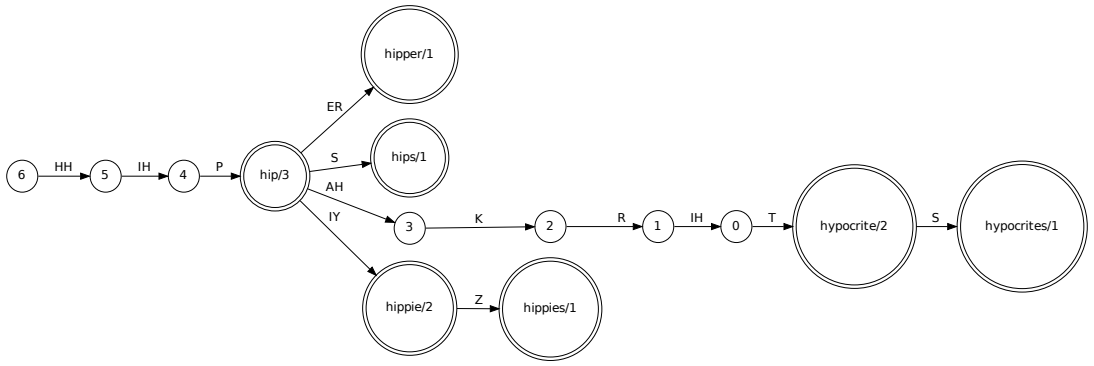

Figure 5.8: Dictionary trie after the distance marking phase of the prefix class building algorithm

Equivalence classes are then built by propagating equivalence classes up through the trie, as shown in Figure 5.9. We have left the lower-order classes in place to illustrate the function of the distance parameter. Leaf nodes, which have distance 1, contain homophone sets, while nodes of distance 2 contain primarily sets of words distinguished by a single affix, and nodes of distance 3 and above contain more diverse and longer suffixes. While this parametrization is crude, the equivalence classes can be generated in linear time.

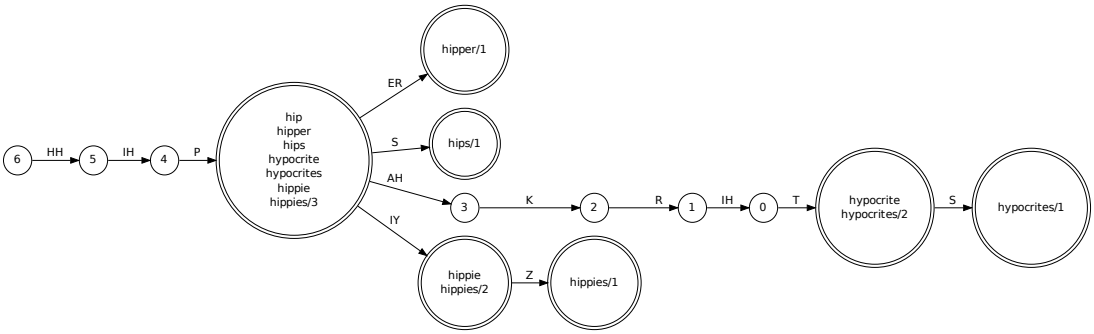

Figure 5.9: Dictionary trie after the class propagation phase of the prefix class building algorithm

As with the homophone merging algorithm discussed earlier (and, by definition, any merging of equivalence classes in a language model), this algorithm is guaranteed to reduce the perplexity of the language model. The question becomes whether this reduced perplexity is offset by two other important factors in the performance of the recognizer. The first of these is the acoustic diversity of the classes, or more precisely the acoustic divergence of the members of an equivalence class from the canonical pronunciation with which it is entered in the dictionary. The second is the cost of disambiguating the classes in a subsequent pass of recognition, since as soon as we begin to admit phonetic diversity inside equivalence classes, we face the problem of requiring acoustic information to disambiguate them.

Predictably, increasing the level of prefix merging leads to a much smaller vocabulary but also a higher word error rate, even when measured over that vocabulary. Results for the Communicator data set are shown in Table 5.8. In the MULTISPHINX system described in the next chapter, however, the use of a shortlisting second pass of decoding rather than language model rescoring means that this is not a serious problem, at least when all passes of recognition are running on the same local machine.

level	xRT	vocabulary	perplexity	error	rescoring error
0	0.16	2001	16.1	23.89%	N/A
1	0.16	1366	16.2	32.90%	29.71%
2	0.14	1070	16.1	40.94%	39.70%

Table 5.8: Higher levels of prefix merging increase word error rate when only language-model based rescoring is used to reconstruct the original vocabulary result

5.2.5 Residual Word Expansion

Even with the latitude afforded us by the shortlisting strategy, it is abundantly obvious that an equivalence class with pronunciation `/HH IH P/` containing both `hip` and `hippopotamus` will create problems for the decoder. The success of the compound splitting experiments in Section 5.2.1 relied heavily on the fact that in the split-word case, the component words were all represented in the dictionary as well as in the language model - therefore no acoustic divergence was incurred by the absence of compounds. If we wish to replicate this condition, we must either confine the splitting of long suffixes to those which are existing words in their own right (in which case the algorithm becomes simply a way of automatically detecting compound words), or we must find a way to add the "residual" suffixes to the language model and dictionary.

As noted in Section 5.2.3, many of these residuals correspond to existing words in the full vocabulary. Therefore, one strategy for using residual information is simply to use the prefix merging algorithm as a compound word finder, merging only words whose residual suffixes are all existing words. Unfortunately, when using the class merging algorithm from Section 5.2.2, this leads to much poorer accuracy, because the probability estimates for these existing words in the language model are now wildly incorrect, being that they are missing the probability mass that had been previously allocated to compounds containing them. As a first approximation to correctly estimating them, we can simply expand them in the language model training set, as was done with compound words previously. This allows us both to achieve better first-pass recognition accuracy when compared with the original transcripts, and to achieve a greater gain in accuracy when rescoring. For example, using prefix expansion level 3, we can achieve a rescoring error rate of 29.15% versus 39.70% without residual words.

5.2.6 Residual versus Fragment Expansion

In effect, rebuilding the language model after splitting words into stems and residuals using the prefix class building algorithm is a particular case of *hybrid language modeling*, where the language model is built on a combination of words and sub-word units. The typical application of hybrid language modeling is out-of-vocabulary word detection and transcription. In the former case, a recent successful approach has been to

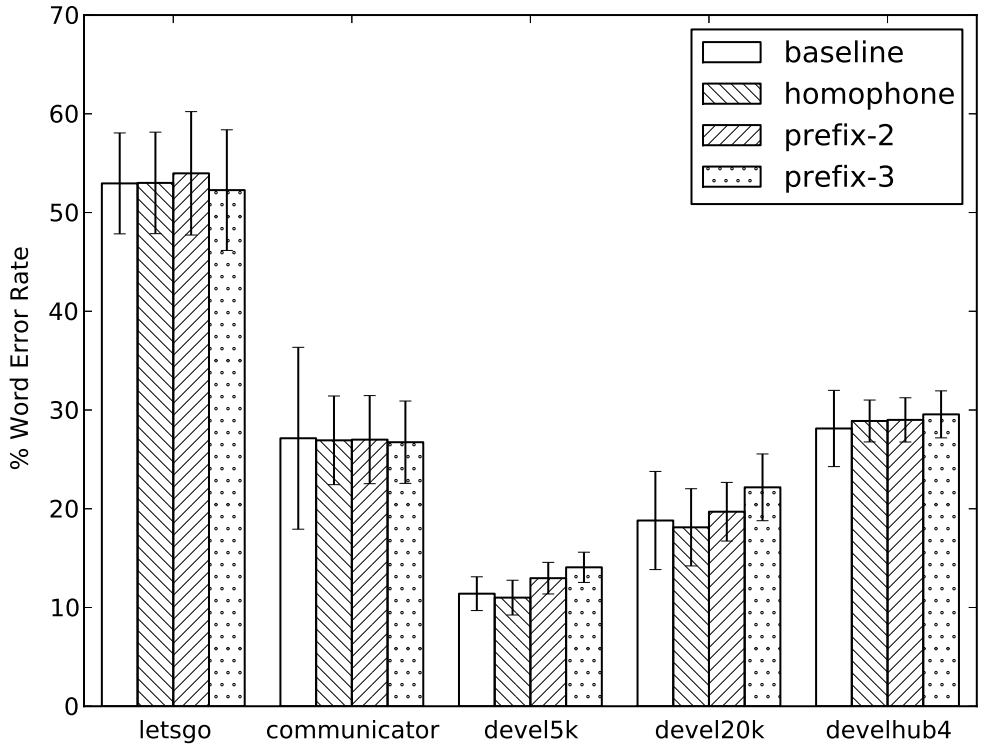

Figure 5.10: When shortlisting and second pass search are used, even higher levels of prefix merging do not significantly affect accuracy

use the marginal posterior probability of sub-words as an estimate of the probability of an out-of-vocabulary event [Rastrow et al., 2009]. In the latter case, joint grapheme-phoneme subword units have been proposed as a means to recover the orthography of out-of-vocabulary words [Galescu, 2003]. In future work it may be interesting to compare the performance of fragments built with the prefix algorithm on this task to those built using language model pruning.

5.2.7 Suffix Tree Lexicon

As mentioned above, explicitly including residuals in the language model presents various complications when it comes to estimating their language model probabilities. Therefore, a simpler alternative is to retain the phonetic structure of the full vocabulary in the lexicon tree, but restrict the set of emitted words to those in the reduced vocabulary. This is based on the observation that higher levels of stemming lead to large numbers of insertion errors as well as denser lattices, as the decoder struggles to find word hypotheses for the "dangling" suffixes removed by stemming. The assumption here is that the increased memory use and computation incurred by recognizing the full words is less than that incurred by attempting to recognize the

residual words using a divergent vocabulary and language model.

Note that given the output of the prefix merging algorithm, we can make several useful assumptions about the reduced and full lexicons:

1. Both pseudowords and full words are present in the dictionary and therefore share the same space of word IDs.
2. Only one pseudoword can occupy a given lexicon tree node.
3. Every pseudoword is a (non-strict) prefix of a full word.
4. Every pseudoword is also a full word.

Recall from Section 3.4.1 that the POCKETSPHINX lexicon tree is split into two parts, a static tree of word-initial and word-medial phones and a dynamic array of word-final phones. The assumption that every pseudoword is also a full word is crucial to a simple implementation of this, given the two-part structure of the lexicon tree. In particular this means that every pseudoword will be allocated a right context array, and there is no need to add a mechanism for "premature exit" from the lexicon tree. As a corollary, this means that no modification to the structure of the lexicon tree is necessary in order to distinguish between pseudowords and full words; we need only change the construction of the dynamic final-phone list and the handling of word transitions.

The transition from the static lexicon tree to the dynamic word-final phone list is managed by a list of "homophone sets" in the decoder, which are not actually homophone sets but rather sets of words with the same static phone sequence (i.e. which differ only in the final phone, if at all). The first thing that we must do, therefore, is to collapse all actual homophones in these sets. This subsumes the level-1 merging (homophone merging) and also ensures that duplicate suffixes are not searched. Then, we need to somehow mark those full words which do not correspond to any pseudowords - these words will then not be entered into the backpointer table in the word transition phase of the decoding algorithm.

In practice, this is complicated slightly by the fact that POCKETSPHINX, quite sensibly, adds the language model score to the path score when entering the final phone of a word. However, for the full words, there is no language model score, since they are not in the (reduced-vocabulary) language model. Therefore, we map them back to their corresponding pseudowords and use the resulting language model scores. Finally, rather than marking the full words, we simply check at the point where word exits are recorded if the word in question is present in the reduced vocabulary, ignoring it if so. A variation on this strategy is to perform word transitions for all full vocabulary words, simply entering the corresponding pseudoword in the backpointer table. The effect of this is to extend the window in which this pseudoword is considered to be active for shortlisting.

This approach was tested on the MULTISPHINX platform detailed further in the next chapter. First, we found that, when running without a reduced vocabulary, the number of HMMs evaluated in the first pass increased. This is as expected since there are more HMMs in the suffix lexcon tree. Unfortunately, HMM evaluation is also responsible for the bulk of the CPU time spent on search, and we therefore saw no improvement in CPU usage for the first pass. When running with a high level of prefix merging (such as at level 3), we would expect the differences to be more pronounced. In this case we again see that more HMMs are being evaluated per frame in the first pass. We do, however, see a consistent reduction in the number of HMMs evaluated in the second pass as well as the amount of time spent there.

Unfortunately, this is the very opposite of the goal of vocabulary optimization and expansion, namely to move the computational load from the first to the second pass. The problem here is, as noted previously, inherent to the way we have implemented this, by choosng to retain the phonetic structure of the full lexicon.

5.3 Summary

There is an optimal vocabulary size in terms of both performance and accuracy for any given task, which is often much smaller than the language model vocabulary. In a backoff N-Gram language model, it is possible to disable portions of the vocabulary without re-estimating the parameters of the model; this has little effect on accuracy, but a significant effect on runtime performance. Pursuing this further, it appears that the accuracy of the *backoff weights* in the model is crucial for performance. This is backed up by the observation that, in decoding, the majority of language model queries result in backoff.

Having established the value of a reduced vocabulary, we turn to the question of how to optimize the vocabulary of the recognizer. It appears that removing *redundancy* from the vocabulary improves performance, and, furthermore, if this is done such that the ambiguities created can be resolved by the language or acoustic model, it has no effect on total system accuracy. This achieves one of the key goals of this thesis, namely, the ability to shift complexity from the initial pass of recognition to a subsequent pass. In addition, it allows us to neatly sidestep the issue of confidence estimation and rejection for an asymmetric two-pass system, as detailed in [Levit et al., 2009, Sainath, 2009]. It also appears that beyond simply removing redundancy in an easily recoverable way, it is possible to "damage" the first pass further in order to balance the computational load between first and second passes while maintaining system accuracy. This can be easily done by collapsing common prefixes into "pseudowords", which are expanded to their corresponding word classes when constructing the vocabulary for second-pass search. This is of particular interest in the case of a multithreaded decoder such as the one to be described in Chapter 6.

Finally, the problem with prefix merging is that it introduces noise into the search space when the discarded suffixes go unrecognized. While it is possible to address this problem by including these "residual words" in the language model, there can be an extremely large number of them, and estimating their language model probabilities is problematic. A simpler solution is to retain the phonetic forms of all words in the full dictionary in the search, while only propagating pseudowords at the sentence level (using the reduced language model). However, this not only negates the performance advantage of using the reduced vocabulary but also leads to further confusability at the second pass. As such, the issue of residual words has not been satisfactorily solved. It is likely that a head-tail model where the residuals are modeled with general phonetic models, as in [Schaaf, 2001], rather than with full context-dependent triphones, would produce better results.

Chapter 6

MultiSphinx Architecture

One problem with the POCKETSPHINX decoder is that, though it has a relatively efficient integrated multi-pass search, this component is monolithic in nature. That is, the separate passes are not independent objects but are implemented as methods in a single "N-Gram Search" class. In particular, they share the same backpointer table and parts of the search graph. Information is shared between them by the second and third passes reading the backpointer table in its entirety, processing it into word lookahead tables or lattices, and then overwriting it.

The advantage of this architecture is that the first and second passes are able to share a considerable amount of code and memory. However, it leads to significant latency, particularly on low-memory-bandwidth architectures. Since reduced latency is one of the arguments we give for performing ASR tasks on the device, this is a serious problem. In this chapter, we describe the architecture of the MULTISPHINX decoder, which allows multiple passes of recognition to run concurrently in a producer-consumer relationship, and provides a standardized *arc buffer* mechanism for communication between passes.

This chapter describes unpublished work, although the analysis of acoustic scores in Section 6.5 is similar to that previously presented in [Huggins-Daines & Rudnicky, 2008, Huggins-Daines & Rudnicky, 2009].

6.1 Architectural Concepts

We begin by introducing the major underlying concepts of the MULTISPHINX architecture, then moving on to specific implementation details. The primary architectural goal of this system is the minimization of latency, or more precisely, the maximization of accuracy for any given amount of latency. To that end, MULTISPHINX departs dramatically from the batch-oriented architecture of previous SPHINX decoders; everything is done incrementally, and in parallel where possible. Not only does this allow for decreased latency but as well, as we shall see in the section on incremental posterior probability calculation, it also gives us empirical results about the amount of lookahead which is necessary for inference in a dynamic system, a result which may be of interest in other domains of natural language processing and understanding.

6.1.1 Anytime Recognition

A high-level diagram of the MULTISPHINX architecture is shown in Figure 6.1. One of the most important features of this architecture is that it allows for *anytime recognition*. Specifically, note that partial hypotheses are available from any of the concurrently running decoders. Since these decoders operate in a pipelined fashion, each improving on the output of the one to its left, the lowest latency can be achieved by querying the initial decoder, while the highest accuracy can be achieved by querying the final decoder, at the cost of a noticeable lag between the user's speech and the last recognition result.

Furthermore, when the earlier passes of decoding are queried, only the parts of the result not available from later, more accurate passes are included in the returned hypothesis. A *hypothesis splicing* technique is used to combine the results of multiple passes, in order to yield the most accurate result possible with the minimum latency possible. More details and results related to hypothesis splicing are available in Section 6.4.

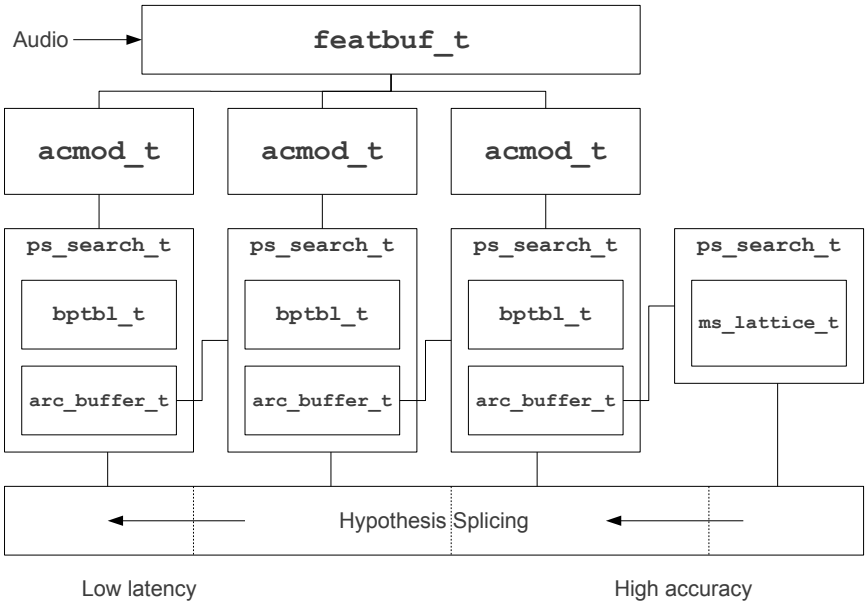

Figure 6.1: Multisphinx architecture and data flow

6.1.2 Elastic Lookahead

Since the information used by the second pass to constrain search is restricted to a sliding window of word entries, it is not necessary for it to read the entire backpointer table at once; we need only allow a sufficient delay between the first and second passes to ensure that the complete set of words beginning in the sliding window has been recognized. Likewise, the lattice construction done by the third pass requires only that all instances of a word starting in a given frame have exited before that (word,frame) pair can be entered as a node in the lattice.

Determining the exact lookahead window is straightforward. It is sufficient to track the oldest backpointer ID which can still generate a word exit. This requires scanning the set of active HMM states, which can be done concurrently with the construction of the set of active senones at the beginning of each frame of search. A side benefit of tracking the start of the active region in the backpointer table is that unreachable entries before it will never be reachable from any future entries, and can thus be garbage collected. In practice we split the backpointer table into "retired" and "active" tables. As the active region advances, newly inactive entries are garbage collected and the remaining ones transferred to the "retired" table. Since no entries can be added to the "retired" table except by the garbage collection process, and the start and endpoints of nodes in it are fully known, we are able to use a fixed size buffer for it. The final pass incrementally transforms it into a word graph, discarding backpointer entries as it goes.

In developing the architecture described in this chapter, we first developed the split backpointer table described in this section. We then refactored the code to make the three N-Gram search passes into stan-

dalone search modules, each with their own backpointer table. Since the number of backpointers generated by the second pass search is an order of magnitude smaller than that generated by the first pass, this actually reduces total memory consumption once garbage collection is applied to the tables.

However, these tables are still subject to reallocation, either for long utterances, or for dense regions of the word lattice. Clearly, the final lattice must be allowed to grow to the full size of the utterance.

6.2 Implementation: Objects and Threads

Although implemented in ANSI C, MULTISPHINX uses an object-oriented architecture, albeit one with a very limited degree of inheritance. There are three distinct components to the interface presented by the MULTISPHINX library:

- The public API, contained in `pocketsphinx.h`, which is largely unchanged from POCKETSPHINX.
- The utility API, contained in the `sphinxbase` include directory, which subsumes the SPHINXBASE component which was previously distributed separately from POCKETSPHINX.
- The low-level API, contained in the `multisphinx` include directory, which gives access to internal components in order to allow the construction of recognition pipelines and manipulation of these components from scripting languages.

It is this low-level API, and both its public and private interfaces, which is described in this section. We pay particular attention to the threading and synchronization model used. A number of objects act as conduits between producer and consumer threads, and as such, their methods are labeled as being for the exclusive use of one or the other.

6.2.1 Feature Buffer

The *feature buffer*, implemented by `featbuf_t`, is particular to a single acoustic input stream. In the POCKETSPHINX API it belongs to the `ps_decoder_t` object and runs in the application thread. The purpose of the feature buffer is to perform acoustic feature computation from audio and buffer these features for the chain of search modules. Frames of acoustic features are reference counted, and the memory occupied by them is freed when all search modules have released them. In the threading model, the feature buffer has a single *producer*, which provides input in the form of audio data or precomputed features, and multiple *consumers*, which wait for, process, and release frames of features. The actual synchronization services for the frame array are provided by an opaque type, `sync_array_t`. Currently the implementation uses a growable array, but the use of an opaque type will allow this to be implemented using a circular buffer in the future, if the copying of memory becomes a performance issue.

The other function of the feature buffer is to synchronize the beginning and end of utterances. This is accomplished by way of two semaphores, `release` and `start`, which are manipulated by four methods: `featbuf_producer_start_utt()`, `featbuf_consumer_start_utt()`, `featbuf_producer_end_utt()`, and `featbuf_consumer_end_utt()`. Each decoding thread registers itself with the feature buffer by retaining a reference to it, then enters a main loop of the form

```
1 def decoding_thread_main(search):
2     while acmod_consumer_start_utt(search.acmod) != DONE:
3         arc_buffer_producer_start_utt(search.output_arcs)
4         while acmod_consumer_wait(search.acmod) != EOU:
5             acmod_score(acmod)
6             decode(search)
7             arc_buffer_producer_sweep(search.output_arcs)
8         arc_buffer_producer_end_utt(search.output_arcs)
9         acmod_consumer_end_utt(search.acmod)
```

Listing 6.1: Decoding thread main loop

shown in Listing 6.1. The main thread calls `featbuf_producer_start_utt()` to signal the start of an utterance, which raises the `start` semaphore by the number of decoding threads. Each decoder thread lowers the `start` semaphore and begins to process input data. The main thread calls feature buffer methods which process audio data and queue features. At the end of the utterance, it calls `featbuf_producer_end_utt()`, which lowers the `release` semaphore by the number of decoding threads. Each decoding thread calls `featbuf_consumer_end_utt()` when finished, raising the semaphore each time. This causes the main thread to block until all decoding threads are complete, returning all semaphores to their initial state.

The main loop is slightly different for search modules which take arcs or a combination of arcs and acoustic features as input, though the synchronization mechanism for arc buffers is exactly the same. The `arc_buffer_producer_sweep()` method will be discussed in greater detail in Section 6.2.4.

6.2.2 Acoustic Models

The *acoustic model* object encapsulates the acoustic scoring components of the recognizer. It consumes frames from the feature buffer and outputs acoustic scores for an active set of senones. Each search module has its own acoustic model object, which acts as an adaptor for the feature buffer. Currently, the acoustic model scoring and search run in the same thread, largely because the acoustic score array is quite large. The acoustic model parameters themselves can be shared between multiple search modules. However, it is also possible to use different acoustic model parameters for different passes - this allows for the maximum flexibility in balancing the load between passes, as a fast, approximate model can be used to offset the greater search complexity of the initial pass of search.

Some varieties of acoustic model, semi-continuous models in particular, can benefit from sharing intermediate results between multiple acoustic model instances with the same underlying parameter set. In general, the acoustic scores themselves are not shared, because the set of active HMM states for a given frame can vary greatly between different search modules, and thus to share acoustic scores would require that all state output distributions be computed at every frame. However, for semi-continuous models, the Gaussian densities themselves, being shared between all HMM states, are independent of the current configuration of the search graph. In addition, since only the top N codewords are used to compute mixture densities, the amount of information that must be stored per frame is extremely small. Therefore, the feature

buffer provides a *side channel* through which this or any other frame-based information can be forwarded from one search thread to the next. With semi-continuous models, the amount of computation which can be saved by sharing this information is comparatively small, although in a system with many threads, it can be significant.

6.2.3 Backpointer Tables

Because of the right context expansion problem described in Section 3.4.3, one of the important functions of the backpointer table is to record and calculate exit scores for word hypotheses in a given context.

In the implementation of MULTISPHINX it was noticed that the differences between right context acoustic scores are quite small. In addition, since the best path score is the one stored in the backpointer table entry, it is guaranteed that the scores of any other right contexts are worse than this score. Therefore, we can encode them as deltas using an unsigned 8-bit value.

6.2.4 Arc Buffers

One complication is that, while the output of the first and second passes of recognition consists of *backpointers* which correspond to word *exits* at a given time, both the second and third passes instead require *word arcs* as input, that is, their input consists of (word, starting-frame, score) tuples. In fact, this is a general requirement for any kind of lookahead or rescoring algorithm. Therefore, MULTISPHINX provides a standard mechanism for transforming the output of a backpointer table to word arcs. This is very similar to the "arc generation" phase of lattice construction in POCKETSPHINX. In principle, the final lattice generation sink simply links these arcs into a lattice, although the reality is somewhat more complicated, as discussed in Section 6.3.2.

While, as mentioned in section 6.1.2, it is possible to divide the backpointer table into an "active" component and a "retired" component, which can be reordered and garbage collected, the requirement that all outgoing arcs from a particular starting frame be known at once imposes an additional partition on the backpointer table. Specifically, although the set of backpointers *exiting* at a certain frame can be obtained from the search graph, in order to fix the set of words *entering* at a certain frame it is necessary to also track all backpointers which are referenced from the active component of the backpointer table. Therefore we maintain an index of the oldest start frame for any active backpointer.

The component which transforms backpointers into arcs and forwards them to the next pass of recognition is known as the *arc buffer*. It is a buffer, because in order to generate arcs from backpointers, it is necessary to retain their predecessors in order to determine the appropriate start frames and the context-dependent acoustic scores. The arc structures, shown in Listing 6.2, contain only the minimum amount of information needed by the consumer. In the case of `fwdflat` search, the arc scores are not required, and thus the basic arc structure contains only information about which word was recognized at which time-points. Although the language model scores are approximate, as noted in Section 3.4.2, for reasons that will be described in more detail in Section 6.3.2, it is not possible to store only the segmental acoustic score for each arc, and therefore the expanded arc structure is accompanied by an auxiliary array which contains information about the distinct right context scores with which a given word exited. This is indexed by the `rc_idx` field, while the set of right context IDs is stored in a bit vector. The size of this bitvector is not

```
1  typedef struct arc_s {
2      int32 wid;      /**< Word ID. */
3      int16 src;      /**< Source frame index. */
4      int16 dest;     /**< Destination frame index. */
5  } arc_t;
6
7  typedef struct sarc_s {
8      arc_t arc;           /**< Base arc structure. */
9      int32 score;         /**< Best path score. */
10     int16 lscr;          /**< LM component of score. */
11     uint16 rc_idx;       /**< Index into right context table. */
12     bitvec_t rc_bits[0]; /**< Set of active right contexts. */
13 } sarc_t;
```

Listing 6.2: Arc structures

specified at compile time because it depends on the number of context-independent phones present in the acoustic model.

The original implementation of arc buffers retained the idea of the backpointer tables as being buffer queues. However, because the backpointer table is not only used as a sink for word hypotheses, but is also queried at runtime by the search algorithms in order to generate language model scores, this resulted in a fairly complicated regime of mutual exclusion locks in order to safely share the backpointer table between the first and second passes of search.

The arc buffer, by contrast, is a genuine queue object - the first pass only adds new arcs to its end, while the second pass only removes arcs from its start. That is to say, the two passes act purely as a *producer* and *consumer*. The synchronization primitives used are essentially the same as those used by the feature buffer, although they are simplified by the fact that arc buffers are restricted to a single producer and a single consumer. We have continued to implement the arc buffer using a growable array, however, so it is still necessary to add locks around operations which might move memory.

6.3 Implementation: Algorithms

In this section we give details of some of the specific algorithms mentioned in the previous section. The majority of this section is concerned with the mechanisms of incremental search which are used to produce accurate partial results and confidence measures for anytime recognition.

6.3.1 Arc Buffer Sweep

Because entries in the arc buffer are indexed by start frame, and because the set of entries for a given start frame is closed, it is possible to construct the arc buffer incrementally using a radix sort algorithm. From the point of view of the arc buffer producer, this entire process is encapsulated in a single function,

`arc_buffer_producer_sweep()`. Each arc buffer is attached to a backpointer table, and the producer functions of the arc buffer are expected to run in the same thread as all backpointer table operations. As such, the arc buffer has privileged access to the internals of the backpointer table, and the search module code need only call the sweep function at some point in each frame. This is preferably done after garbage collection of the backpointer table, because the sweep operation takes as input newly inaccessible entries from the retired table, that is, entries which are no longer backreferenced by any active word exits. The arc buffer sweep algorithm is shown in Listing 6.3.

Internally, the arc buffer sweep involves three operations: `extend`, `add_bps`, and `commit`. First, the array of start frames is extended up to the first frame in which active backpointer table entries still exist. These newly allocated frames are marked as "incomplete", because arcs beginning in them have not yet been sorted by start frame. In the `add_bps` operation, the incoming entries are appended to the end of the arc buffer and scanned to count the number of entries for each start frame. If scores are being retained, the language model score component is calculated and the right context array is compressed and copied to the internal right context table. Finally, in the `commit` operation, indices in the arc buffer for the new frames are generated from the entry counts, and the arcs are permuted to match these. The permutation involves copying the array counters as well as the incoming arcs, but also has linear time complexity.

6.3.2 Incremental Word Lattice Generation and Expansion

The concept of a *word lattice* was described previously in Section 2.10, along with an overview of the conventional algorithms for generating word lattices. Not mentioned there is that lattice generation is nearly always treated as a post-processing step. When they are used in a live system, lattices are primarily generated as intermediate results for a multi-pass system or as a compact representation of N-best lists to be processed by a natural language understanding system. In all cases the lattice is generated at some point after the 1-best hypothesis has been recognized.

From the point of an anytime system this is not particularly useful, since any gains in accuracy or coverage that might be obtained from the lattice are only available at the very end of recognition. In MULTISPHINX we already perform parallel multi-pass recognition through the arc buffer mechanism. However, it is also useful to be able to obtain a proper lattice, both for rescoring and confidence estimation. Therefore we have extended MULTISPHINX to support the generation and rescoring of N-Gram lattices in an incremental fashion. Not only does this allow us to obtain rescoring results and confidence scores at any time, but it also allows the use of 4-gram or higher order language models for rescoring.

It is important to understand that this is not only generation but also *expansion* of lattices. Because of the nature of the first and second passes of search in MULTISPHINX it is not possible to generate an accurate N-Gram lattice from the search output. In addition, the arc buffer mechanism purposely excludes language model information in order to save memory. Recall from Section 2.10 that we define an N-Gram lattice as one where every instance of a word in the lattice has a unique N-1-Gram history. For reasons of efficiency, it is useful to assign word instances to arcs in such a lattice, which implies that each node corresponds to a given N-1-Gram history.

```
def extend(arc_buffer, next_sf):
    arc_buffer.sf_idx.resize(next_sf)

def add_bps(arc_buffer, bptbl):
    for ent in bptbl[arc_buffer.next_idx, bptbl.retired_idx]:
        if ent.start_frame < arc_buffer.active_sf:
            continue
        if ent.start_frame >= len(arc_buffer.sf_idx):
            arc_buffer.next_idx = bptbl_idx(bptbl, ent)
            break
        arc = Arc(ent.wid, ent.start_frame, ent.end_frame)
        arc_buffer.arcs.append(arc)
        calculate_scores(arc, ent)
        arc_buffer.sf_idx[ent.start_frame] += 1

def commit(arc_buffer):
    prev_count = arc_buffer.sf_idx[arc_buffer.active_sf]
    arc_buffer.sf_idx[arc_buffer.active_sf] = arc_buffer.active_arc
    for i in xrange(arc_buffer.active_sf + 1, len(arc_buffer.sf_idx)):
        tmp = arc_buffer[i]
        arc_buffer[i] = arc_buffer[i-1] + prev_count
        prev_count = tmp
    frame_pos = arc_buffer.sf_idx[arc_buffer.active_sf:]
    incoming_arcs = arc_buffer.arcs[arc_buffer.active_arc:]
    for arc in incoming_arcs:
        arc_buffer.arcs[frame_pos[arc.start_frame]] = arc
        frame_pos[arc.start_frame] += 1
    arc_buffer.active_sf = len(arc_buffer.sf_idx)
    arc_buffer.active_arc = len(arc_buffer.arcs)
```

Listing 6.3: Arc buffer sweep algorithm

PocketSphinx Lattice Generation

Our initial implementation of N-Gram lattices was simply an expansion algorithm which takes a conventional SPHINX unigram lattice as input and outputs an N-Gram lattice. In order to understand this we should first understand how POCKETSPHINX generates its unigram lattices. A description of the POCKETSPHINX lattice generation algorithm is shown in Listing 6.4.

Significant qualities of this algorithm are, first, that its time complexity is $O(N^2B)$ where N is the number of unique word/start frame pairs and B is the number of backpointer table entries. This assumes that appending nodes and adding links occurs in amortized constant time, which in this implementation it does.

Second, the lattice is generated in reverse, starting from the final node. This is done in order to mark reachable nodes so that unreachable nodes can be pruned. Because the backpointer table is actually a tree, which only stores a single predecessor for any word hypothesis, all nodes are reachable by definition from the start node. This also means that if we were to simply take the intersection of the nodes directly coaccessible from the final node and those accessible from the beginning node, we would obtain a single path. Therefore the definition of node adjacency is relaxed in this backward pass of lattice construction to include any nodes for which some arc exists ending in the frame before a given node under consideration. This is shown in lines 29 and 30 of Listing 6.4.

This is the reason why `bptbl.compute_ascr()` exists. When word exits are recorded in the backpointer table, all instances of a given word exiting in the same frame are collapsed into a single backpointer entry, with only the best scoring path retained in the backtrace. However, the path scores for distinct right contexts must be retained. The first reason for this is that when performing word transitions it is necessary to re-enter the roots of the lexicon tree with the appropriate path scores for the corresponding initial phones. In addition, certain parts of the `fwdtree` search algorithm require the ability to calculate the segment score for incomplete word hypotheses, which requires the ability to look up these initial path scores. In order to do so it is necessary to reference the backpointer entries pointed to by nodes in the decoding graph, which is the primary reason we maintain an unpruned "active" component of the backpointer table in MULTISPHINX.

Finally, of course, when calculating the acoustic score for adjacency links in the lattice which do not explicitly exist in the backpointer table, it is necessary to know the corresponding exit score in a predecessor arc corresponding to the initial phone of a node. This is done in two parts by POCKETSPHINX. The first, `compute_seg_score`, shown in Listing 6.6, computes the best segmental score of all right contexts for the preceding backpointer entry, while `exit_score`, shown in Listing 6.5, computes the specific right context score corresponding to the initial phone of the target node. Note that in MULTISPHINX the right context scores are already stored as deltas, so we need only look up the appropriate delta score, if any.

Although we have been considering these adjacency links to be "artificial" in that they are not explicitly represented in the backpointer table, if there exists a right context score in a predecessor corresponding to the initial phone of a word, then it is in fact true that this path was at least partially searched in the decoder, since the lexical tree root for this word would have been entered from that particular word-final triphone. What if no such right context score exists? We have two alternatives - either we can create artificial links with some acoustic score, or we can ignore them. In the latter case we run the risk of propagating search errors from the previous pass, but in the former case we are introducing modeling errors, since the true acoustic score for these links is not known.

```
def build_lattice(bptbl, lm):
    dag = Dag()
    for i, bp in enumerate(bptbl):
        if not bp.valid: continue
        node = None
        for n in dag.nodes:
            if (n.word, n.start_frame) == (bp.word, bp.start_frame):
                node = n
                node.last_bpidx = i
                break
        if node == None:
            node = Node(bp.word, bp.start_frame)
            node.first_bpidx = i
            node.reachable = False
            nodes.append(node)
    dag.nodes.reverse()
    dag.start = dag.end = None
    for n in dag.nodes:
        if n.word == start_word and n.start_frame == 0: dag.start = n
        if n.word == end_word and n.end_frame == last_frame: dag.end = n
    dag.end.reachable = True
    for i, n in enumerate(dag.nodes):
        if not n.reachable: continue
        for m in dag.nodes[i+1:]:
            bp = None
            for j in xrange(m.first_bpidx, m.last_bpidx + 1):
                if bptbl[j].word == m.word \
                        and bptbl[j].end_frame == n.start_frame - 1:
                    bp = bptbl[j]
                    break
            if bp == None: continue
            ascr = bptbl.compute_ascr(bp, n.word)
            if ascr == WORST_SCORE: continue
            m.add_link(n, ascr, bp.end_frame)
            m.reachable = True
    return dag
```

Listing 6.4: POCKETSPHINX lattice generation algorithm

```
def exit_score(bp, dic, phone):
    if dic.n_phones(bp.word) < 2:
        return bp.score
    else:
        rcidx = dic.right_context_index(dic.last_phone(bp),
                                        dic.penultimate_phone(bp),
                                        phone)
        return bp.rcscore[rcidx]
```

Listing 6.5: Computation of word exit scores

```
def compute_seg_score(bp, lm, dic):
    if bp.prev == None:
        bp.ascr = bp.score
        bp.lscr = 0
    else:
        start_score = exit_score(bp.prev, dic,
                                 dic.first_phone(bp.word))
        bp.lscr = lm.score(bp.word, bp.prev.word, bp.prev.prev_word)
        bp.ascr = bp.score - start_score - bp.lscr
    return bp.ascr, bp.lscr
```

Listing 6.6: Computation of segmental acoustic scores

Static N-Gram Expansion

In previous experiments, as described in Section 4.6.1, the OpenFST toolkit [Allauzen et al., 2007] was used to perform N-Gram expansion on lattices. In that case the reason for doing so was to apply a domain-specific language model to a transformed lattice for rescoring. Not noted in that chapter was that this procedure was very time-consuming. It was this observation coupled with the desire to avoid a dependency on the FST toolkit which led us to implement a SPHINX specific algorithm for N-Gram expansion of lattices. In addition to expansion, the algorithm also removes redundant edges and nodes, which is akin to determinization of finite state automata.

Since the SPHINX lattices mark word identities on the source nodes rather than the arcs, we start by "pushing" the word identities from the nodes to the arcs. In addition, we create a new final node and link the existing final node to it with an arc for the language model's end word, since in POCKETSPHINX this word is defined in the language model and also has a phonetic realization. The previous final node can then be treated the same as any other node by the expansion algorithm.

Top-level code for the algorithm is shown in Listing 6.7. It examines each node in topological order, duplicating it as needed in order to create unique word histories for its outgoing arcs. When a node is duplicated, the entry arc for which the new node was created is repointed at the new node, while all exit arcs from the original node are merged into the set of arcs exiting the new node. Once all nodes have been expanded, any nodes which have been rendered unreachable by the arc merging procedure are removed.

The N-Gram history which is used as the node's identity in the expanded lattice is referred to as a *language model state*. For each incoming arc to the node under consideration, we find the longest explicit N-Gram in the language model corresponding to the N-Gram formed by concatenating the previous node's language model state to the word identity of the current node (remember that the input nodes are SPHINX lattice nodes and therefore they represent a unique word. If the resulting language model state is less than N-1 for the language model, then the backoff weight for the history part of the requested N-Gram is added to the incoming arc's language model score. The search for the next language model state and associated language model score and backoff weight is done in the `next_lmstate` function shown in Listing 6.8. Note that the language model score is applied to the *outgoing* arc, while the backoff weight is applied to the *incoming* arc.

Since this algorithm avoids generating backoff states unless they are necessary to provide a language model score for a particular arc, it is not susceptible to the problem where the backoff probability for an N-Gram is greater than its explicit probability [Allauzen et al., 2003]. Although the backoff arcs in the resulting lattice are followed unconditionally in rescoring, they will never be generated in the first place if an explicit N-Gram exists.

Incremental Lattice Construction

Given that our goal is to perform not only lattice expansion but also lattice generation on the fly, what is the optimal way of combining lattice generation algorithm of Listing 6.4 and the lattice expansion algorithm of Listing 6.7, and how can this be done incrementally using the information provided by the arc buffer? The difficulty of constructing lattices on the fly has been known for quite some time [Sagerer et al., 1996]. However, it turns out that the arc buffer sweep and pruning addresses a number of the problems, such as avoiding dead-ends, which have plagued this task.

```
def ngram_expand(dag, lm, lw=1.0, wip=1.0):
    logwip = math.log(wip)
    end = next_final_node(dag)
    expand_nodes = {}
    for node in dag.traverse_topo():
        push_words_to_arcs(node)
        backoff_entries = []
        duplicate_entries = set()
        for x in node.entries:
            if x.src.sym == None: continue
            lscr, bowt, lmstate = next_lmstate(x, node, lm, lw, logwip)
            if lmstate:
                if (lmstate, node.frame) not in expand_nodes:
                    new_node = dag.Node(lmstate, node.frame)
                    expand_nodes[lmstate, node.frame] = new_node
                    dag.nodelist.append(new_node)
                else:
                    new_node = expand_nodes[lmstate, node.frame]
                merge_exits(new_node, node, lscr)
                merge_entry(new_node, x, bowt, duplicate_entries)
            else:
                x.lscr += bowt
                backoff_entries.append(x)
        for x in duplicate_entries:
            x.src.exits.remove(x)
        node.entries = backoff_entries
        if node == dag.start_node() or node.entries:
            set_ug_lmscores(node, lm, lw, logwip)
            node.sym = ("&epsilon;",)
        else:
            node.sym = None
    dag.end = end
    dag.nodelist.append(end)
    dag.remove_unreachable()
```

Listing 6.7: Static N-Gram expansion algorithm

```
def next_lmstate(x, node, lm, lw, logwip):
    if x.src.sym == ("&epsilon;",):
        lmstate = (x.sym,)
    else:
        lmstate = (x.sym,) + x.src.sym
    bowt = 0
    while lmstate:
        try:
            ng = lm.ngram(node.sym, *lmstate)
            break
        except:
            try: bowt = lm.ngram(*lmstate).log_bowt * lw
            except: bowt = 0
            lmstate = lmstate[:-1]
    if len(lmstate) == 0:
        ng = lm.ngram(node.sym)
    lscr = ng.log_prob * lw + logwip
    return lscr, bowt, lmstate
```

Listing 6.8: Static N-Gram expansion algorithm

The lattice expansion algorithm, like the finite-state composition and determinization algorithms it resembles, is a local algorithm - that is, for each node under consideration, the only information required to perform expansion is the set of incoming and outgoing arcs and the language model states of the predecessor nodes. It is useful for all nodes in a given frame to be processed at the same time, since this would allow the `expand_nodes` array to be indexed purely by language model state, but this is not required.

This suggests that if we can create an incremental version of the lattice construction algorithm, lattice expansion can be done concurrently, provided some way of closing the set of outgoing arcs for each node. The reason it is important to close the set of outgoing arcs is that in the existing lattice expansion algorithm, this whole set of arcs is copied or rather merged to the new nodes created for the destination language model states of incoming arcs. The alternative is to postpone attaching outgoing arcs to specific nodes until all nodes for a given start frame have been generated.

The arc buffer mechanism discussed earlier performs a task similar to this, in that it provides a closed set of outgoing arcs for each frame of input. The arc buffer can be thought of as a pure unigram lattice, with implicit nodes at each frame in which a word hypothesis begins. Therefore to construct a Sphinx-style lattice from the arc buffer, it suffices to make these nodes explicit and calculate segmental acoustic scores for their outgoing arcs with reference to the incoming arcs' right context exit scores. This algorithm is shown in Listing 6.9. Note that since a node represents a unique word/frame pair, all outgoing arcs from a node share the same initial phone, and therefore they also share the same best initial path score. Since the exact set of right contexts which will be represented in the lattice for a given arc is not known until all nodes in that arc's destination frame have been created, we create only incoming links in the lattice for each new node. Once iteration over the arc buffer is complete, we can then create links to an end node at the final frame of

a lattice. Alternately, we can pick a final node and create incoming links to it.

Incremental Lattice Construction and Expansion

The problem with the algorithm in Listing 6.9 is that it does not satisfy the requirement that all outgoing arcs for a node be known at the time when that node is created. While it is possible to generate speculative lattice arcs for every distinct right context found in the arc buffer, this rapidly leads to an extremely large number of unconnected arcs in the lattice which require garbage collection.

Note, however, that the node generation performed by this lattice construction algorithm is quite similar to the node copying and arc merging done by the lattice expansion algorithm. In addition, since the original nodes in the input lattice are frequently discarded, performing lattice construction and expansion in sequence incurs a large amount of useless computation and memory consumption, as a large number of nodes are created only to be pruned and discarded.

The adjustment to the incremental lattice construction algorithm to do N-Gram expansion is conceptually very simple. Whereas previously we created new nodes for each unique word/frame pair, we now create new nodes for each language model state/frame pair. Thus the major change is simply how we determine the identity of the nodes to be created for a given arc in the arc buffer.

While it would seem that this complicates the computation of acoustic scores, since we can no longer rely on having a unique word identity for all outgoing arcs from a node, in fact it simply shifts the computational complexity to a different location. Whereas, when each node had a unique word identity, the exit score for an arc was determined by its destination node, we now have multiple successor words for any arc and therefore the exit score for an arc is simply the best right context score among those. If we continue to work in the same framework of creating incoming links only, we can simply update these best scores after finding or creating a matching node for a given arc in the arc buffer (as on lines 12-13 of `build_arcbuf_\gls{lattice}()`. If we had gone through a two-phase process of constructing a word-node lattice and expanding it, we would have performed this maximization in the latter phase, as in the `merge_entry()` function of the static N-Gram expansion algorithm in Listing 6.7.

The algorithm for incremental construction of N-Gram lattices is shown in Listing 6.10. The outer loop of this algorithm is largely the same as the previous one - for each arc we receive from the arc buffer, we search for an appropriate node and create one if it does not exist, adding incoming links to it in the process. There are two differences; first, in order to find or create the node, it is necessary to construct the appropriate language model state. In fact, there will likely be multiple nodes, one for each possible language model state preceding the arc. In order to construct these language model states it is necessary to know the set of predecessor arcs; however, as in the previous algorithm, we have not yet created the incoming links to the new node.

This is made somewhat more complicated by the fact that the start nodes for arcs in the arc buffer are not easily found, since nodes are no longer identified by their outgoing word. Therefore, we must maintain an auxiliary array of node/arc pairs for each ending frame. After an arc has been processed, these pairs are entered for every node created.

```
def build_arcbuf_lattice(arcbuf, dic):
    dag = Dag()
    def build_arc_node(word, frame):
        node = dag.Node(word, frame)
        dag.add_node(node)
        if frame == 0:
            node.score = 0
            return node
        for incoming in arcbuf.ending_at(node.frame):
            if incoming.word == "</s>": continue
            src = dag.find_node(incoming.word, incoming.frame)
            rc = dic.first_phone(dic.wordid(word))
            rcdelta = incoming.rcdelta[rc]
            if rcdelta == NO_RC: continue
            incoming_score = incoming.pscr - rcdelta
            ascr = incoming_score - incoming.lscr - src.score
            link = dag.Link(src, node, ascr)
            src.exits.append(link)
            node.entries.append(link)
            node.score = max(node.score, incoming_score)
        return node
    for outgoing in arcbuf:
        if dag.find_node(outgoing.word,frame): continue
        if outgoing.word == "<s>" and outgoing.sf != 0: continue
        build_arc_node(outgoing.word, frame)
    end_pscr = WORST_SCORE
    for incoming in arcbuf.ending_at(arcbuf.n_frames):
        if incoming.word != "</s>": continue
        if incoming.pscr > end_pscr:
            src = dag.find_node(incoming.word, incoming.sf)
            dag.end = src
            dag.final_ascr = incoming.pscr - incoming.lscr - src.score
    dag.start = dag.find_node("<s>", 0)
    dag.remove_unreachable()
    dag.n_frames = arcbuf.n_frames
    return dag
```

Listing 6.9: Arc buffer lattice generation algorithm

```
def build_expanded_arcbuf_lattice(arcbuf, dic):
    dag = Dag()
    incoming_arcs = defaultdict(list)
    def build_lmstate_node(src, incoming, lmstate, lscr):
        node = dag.find_node(lmstate, incoming.endframe)
        if node == None: node = dag.Node(lmstate, incoming.endframe)
        link = dag.Link(src, node, incoming.score - lscr
                        - incoming.entry_score)
        link.sym = incoming.sym
        link.lscr = lscr
    for outgoing in arcbuf:
        for src, incoming in incoming_arcs[outgoing.frame]:
            rcdelta = incoming.rcdelta(dic.first_phone(outgoing.word))
            incoming_score = incoming.score - rcdelta
            outgoing.entry_score = max(outgoing.entry_score, incoming_score)
            lmstate, lscr = next_lmstate(src.sym, incoming.word,
                                         outgoing.word)
            node = build_lmstate_node(src, incoming, lmstate, lscr)
            incoming_arcs[outgoing.endframe].append((node, outgoing))
```

Listing 6.10: Incremental N-Gram expansion algorithm

Comparison to Finite-State Composition and Determinization

As mentioned earlier, the lattice expansion algorithm, particularly in its incremental arc buffer version, is quite similar to the weighted finite-state composition algorithm described in [Mohri et al., 2000]. This is made clearer if we consider the arc buffer to be a finite state automaton whose nodes are identified by frame indices. The use of frame/lmstate pairs to identify nodes in the output lattice is not accidental - it is simply the Cartesian product of the arc buffer's nodes and the nodes of the language model, which correspond exactly to language model states.

The composition algorithm works by matching arcs between two finite state automata and lazily creating the product set of nodes as needed in order to provide destination nodes for the matched arcs. This is essentially the same as the parts of the incremental lattice construction algorithm which generate language model and backoff states. As alluded to earlier, generating backoff states requires some extensions to the basic WFST composition algorithm, either in the form of "failure arcs", or by a preprocessing step which detects and removes incorrect backoff paths [Allauzen et al., 2003].

6.3.3 Partial Posterior Probability Calculation

Confidence measures for speech recognition are widely used in dialog systems and other speech understanding tasks. As well, they are useful for unsupervised language and acoustic model adaptation, where they can be used to exclude, or weight down, those parts of the transcript which are potentially erroneous [Pitz et al., 2000].

While the partial transcript and segmentation are quite useful already for certain kinds of higher-level processing, we have not yet mentioned the ability to calculate posterior probabilities. This is partly because the algorithm for calculating word posterior probabilities, which are the basis of most confidence measures, requires a forward-backward pass over the entire word lattice. Recall from Chapter 2 that the posterior probability of a word is defined as the probability of that word given the utterance as a whole, and specifically involves a summation over all paths entering and leaving the word in question, as well as a normalization constant which is calculated as the marginal probability of the input speech, obtained by summation over all paths considered by the recognizer.

The forward variable $\alpha_t(w)$ can be exactly computed for intermediate results, and in fact is computed incrementally as part of the lattice expansion process. Therefore the challenge in providing posterior probability estimates for partial recognition results is estimating $\beta_t(w)$ without having seen the entire utterance. In fact, this is quite simple as long as we are willing to accept that the confidence values for the most recent part of recognition are approximate. To calculate partial betas, we modify the recursion which sets $\beta_T(endw) = 1.0$ to include, instead of arcs entering the final node, all word arcs terminating at the current frame. As detailed in Listing 6.9, this information is available as soon as we finish processing the incoming arcs from the arc buffer for a given frame. This has the advantage that there is no separate final beta computation. However, since it is necessary to propagate the $\beta_t(w)$ variable backwards to the beginning of the utterance, we only perform this calculation if posterior probabilities are requested.

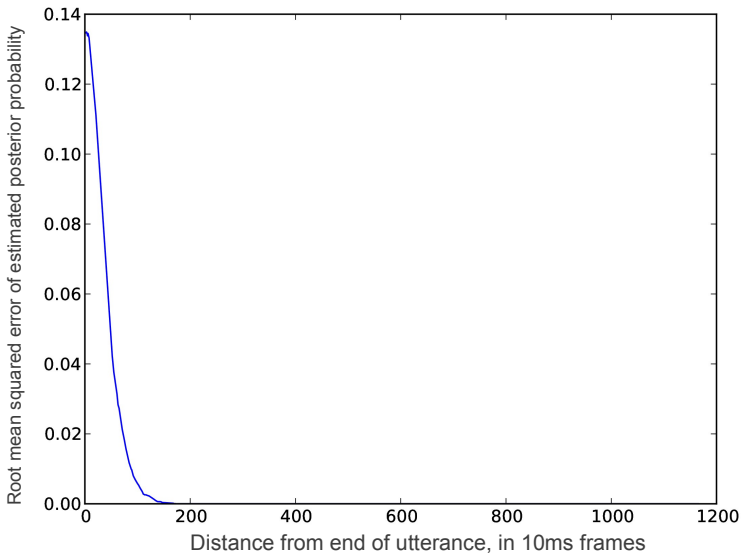

Figure 6.2: RMS error of posterior probabilities based on partial backward variable plotted by lookahead window size

In order to measure the accuracy of the resulting estimates, we computed partial posterior probability estimates repeatedly over each utterance in a test set; for every frame f_i in which there were word exits in

the lattice we recomputed $\beta_t(w)$ for all arcs in the lattice using these word endings as the set of final nodes. Then, after recomputing $P(w,t|o)$ for all preceding arcs, we calculated the RMS error of the resulting probability estimates when compared to probabilities calculated over the full lattice, as a function of the number of frames from f_i. This function, shown in Figure 6.2, shows that the partial posterior probabilities converge towards the full-utterance estimates within a reasonably small window. As with the elastic window used to generate lattices in the first place, discussed in Section 6.1.2, this window's size cannot be calculated *a priori*, but it is correlated with the length of words in the vocabulary.

6.4 Hypothesis Splicing

The architecture of MULTISPHINX is specifically designed to allow multiple decoders to co-operate in a producer-consumer relationship, such that the later passes of recognition are not required to wait until the entire utterance has been processed. This allows the system to take advantage of multiple processors, particularly if the computational load is well balanced between passes. However, the principle of anytime recognition, and the desire to minimize latency, require that the system be able to produce recognition results at any point in time, and furthermore that complete results be available as soon as possible after the end of the utterance. For this reason, we use *hypothesis splicing* to produce recognition results which use as much of the results from later passes (which are assumed to be more accurate) as possible with the least latency possible, where latency is defined as the time between the reception of the audio for a given word and its successful recognition.

6.4.1 Splicing Algorithm

The hypothesis splicing algorithm works exactly as its name implies - we "splice" together the hypotheses from multiple passes of search. This is done by looking for a point of consensus between the passes as close to the end of the utterance as possible. Working within the existing POCKETSPHINX and MULTISPHINX architecture we currently splice only two passes of search. For more than two passes a multiple alignment algorithm such as the one described in [Mangu, 1999] can be employed.

Finding the consensus point is done by aligning the two time-stamped hypotheses U, W using dynamic programming, where U is the shorter, second-pass hypothesis. These hypotheses consist of sequences u_1^N, w_1^N of words and end times r_1^N, t_1^N. The segmental cost function for alignment is defined in terms of both the word identities u_i, w_j and their time stamps r_i, t_j:

$$cost(u_i, r_j, w_i, t_j) = 1 - \delta(u_i, w_j) + |r_j - t_i| \tag{6.1}$$

The convergence point is then defined as the alignment u_i, w_j closest to the end of the hypothesis where the minimum segmental cost is reached. Splicing then consists of simply concatenating u_1^{i-1} with w_j^N.

6.4.2 Latency in Concurrent Recognition

A useful metaphor for hypothesis splicing, and for anytime recognition in general, is that of a horse race. At any particular time during the race, the horses are arranged in a particular order, which provides incomplete

information about the final outcome of the race. As horses cross the finish line, the uncertainty in the outcome diminshes over time, until all have finished and the rankings are fixed. Likewise, in anytime recognition, the various passes of search have consumed differing amounts of the input, and taken as an ensemble, their decoding output constitutes an incomplete picture of the search hypothesis.

Simplifying this to two passes of recognition, the first pass has a more complete result set, or in our metaphor, it knows the ranking of more horses, but its results are less certain and subject to change. By contrast, the second pass has a smaller result set, perhaps only tracking the part of the track closest to the finish line, but its rankings are largely immutable and more predictive of the final outcome. The hypothesis splicing algorithm, then, takes the results from these two passes and attempts to reconcile the immutable, confident results with the mutable, uncertain ones in order to provide a complete picture of the hypothesis (or, in the metaphor, the probable ranking of the horses).

How much distance is there between the two passes in a simple MultiSphinx setup with the `fwdtree` and `fwdflat` recognizers running in parallel? To measure the latency, we instrumented the decoder to record timestamped partial results for each pass. The partial results were generated by tracking the best path at each frame of recognition and emitting a partial result upon a change in the identity of the final word in the path. We simulated real-time audio input by rate-limiting the processing of audio data such that each block of data was passed to the recognizer at the corresponding timestamp. Latency was measured by aligning a time-aligned transcript with the timestamped partial results. The latency for a word in a partial result is the time differential between the end of this word in the time-aligned transcript and the timestamp of the first partial result containing it in the proper sequence. Note that there is no penalty for words which are incorrectly recognized; they simply do not figure in the latency calculation. For this reason this metric is best suited for planned or read speech tasks. Mean and standard deviations for latency on several data sets, running on a 2.4GHz MacBook Pro, are shown for the `fwdtree` (first) and `fwdflat` (second) passes of recognition in Figure 6.3.

These results on latency tell us where the benefit in hypothesis splicing lies - namely, within the 1-2 seconds between the recognition of a chunk of audio by the `fwdtree` and the `fwdflat` passes. In particular, it allows us to obtain a more accurate overall hypothesis immediately after the end of an utterance while waiting for the `fwdflat` pass to complete.

One peculiarity of this definition of latency is that, as is evident from Figure 6.3, it can be negative. This is both because the timestamps in the aligned transcripts are automatically generated and thus may not reflect human perceptions of the end of words, and because it is possible for a word to be recognized before it is completely spoken. This is particularly true for single-phone words and for words which are unambiguous without their final phone.

6.4.3 Accuracy in Spliced Recognition

What gain in accuracy do we obtain through hypothesis splicing? In the algorithm described above, we assume that the second pass hypothesis U is more accurate, and the algorithm is limited to finding the appropriate place to switch over to the less accurate first pass hypothesis. This seems to imply that:

1. The accuracy of the spliced hypothesis will be somewhere between that of the first and second pass hypotheses.

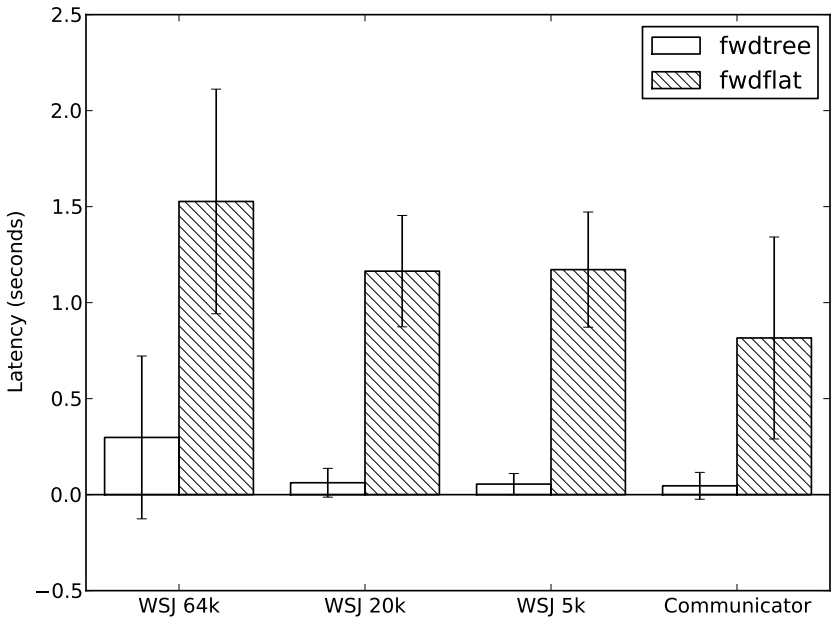

Figure 6.3: The MULTISPHINX architecture allows partial results to be obtained from the second pass of recognition with latency in the 1-2 second range

2. The accuracy of the spliced hypothesis will depend on both the accuracy of the individual passes as well as the latency between them.

3. If the first pass of recognition has produced a final result, the overall accuracy will depend on how much time has passed since this result was generated (note that this is essentially the definition of an anytime system).

4. The end of the spliced hypothesis will be less accurate.

This can be verified using the same data generated to evaluate the latency of the system above, namely time-stamped partial recognition transcripts from the individual passes. The results of splicing hypotheses immediately after the end of the first pass are compared with the full first and second pass accuracy figures in Figure 6.4.

How do the vocabulary optimization and expansion techniques described in Chapter 5 affect the outcome of hypothesis splicing? Clearly one side effect of vocabulary expansion is that the word match component $\delta(u_i.w_j)$ of the cost function in Equation 6.1 becomes less reliable. In addition, obviously, as we saw in Section 5.2.4, the unexpanded first-pass result will have a considerably higher error rate. On the other hand, the rebalancing of load between passes achieved by vocabulary optimization should improve the latency of the system. Also, in particular, the prefix merging algorithm reduces the average length of words in the first pass lexicon, which in turn reduces the size of the lookahead window.

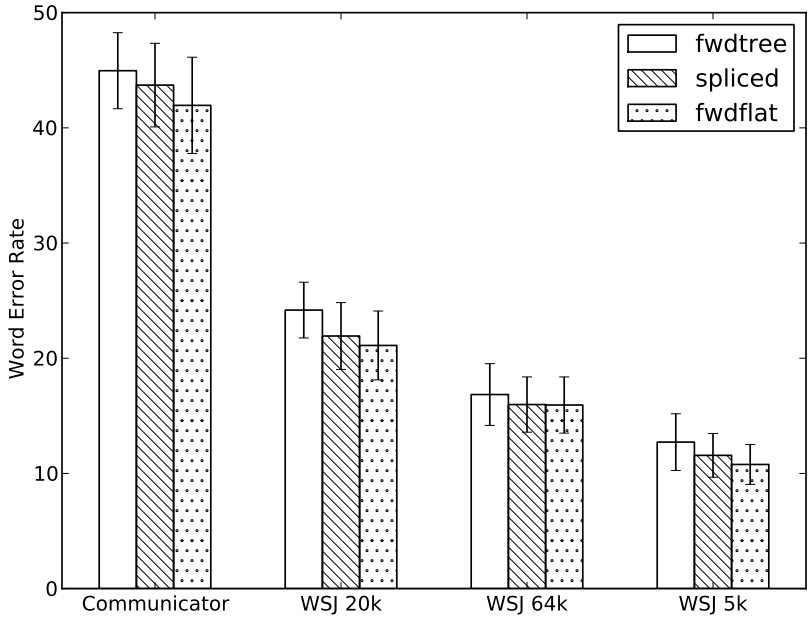

Figure 6.4: Hypothesis splicing immediately after first-pass completion significantly improves accuracy with very little extra latency

6.5 Remote Arc Buffers

Current state of the art, server-based mobile speech recognition systems, while they can achieve impressive accuracy by leveraging the power of enormous distributed language models and Internet search, are fundamentally constrained by network latency. Among other issues, this prevents them from providing real-time feedback on recognition. In addition, the minimum recognition latency, defined here as *time-to-text* by analogy with the *time-to-speak* typically used to evaluate text-to-speech systems, is fundamentally constrained by the speed of light. So, for instance, assuming a direct fiber-optic cable with no switches, in which light travels at $2 \cdot 10^8$ m/s, the theoretical minimum round trip latency between Pittsburgh, Pennsylvania and Palo Alto, California is approximately 36 milliseconds, not including the transmission time of the speech data in question. Given real-world networks which incur routing and packet switching delays, and including transmission time, which is constrained by the bandwidth of the network link, the actual lower band is closer to 80 milliseconds, and real-world latencies, particularly on wireless networks, are typically hundreds of milliseconds.

The first question this raises, then, is what kind of useful recognition we can achieve on the device in those hundreds of milliseconds. The second question is, can the results of this useful recognition be used to accelerate more detailed subsequent passes of recognition, which we have discussed in Chapter 5. In the previous section, we discuss the possibility of running these passes of recognition concurrently in order to exploit multiple CPU cores on a local system. We now consider the final question, namely, whether it is

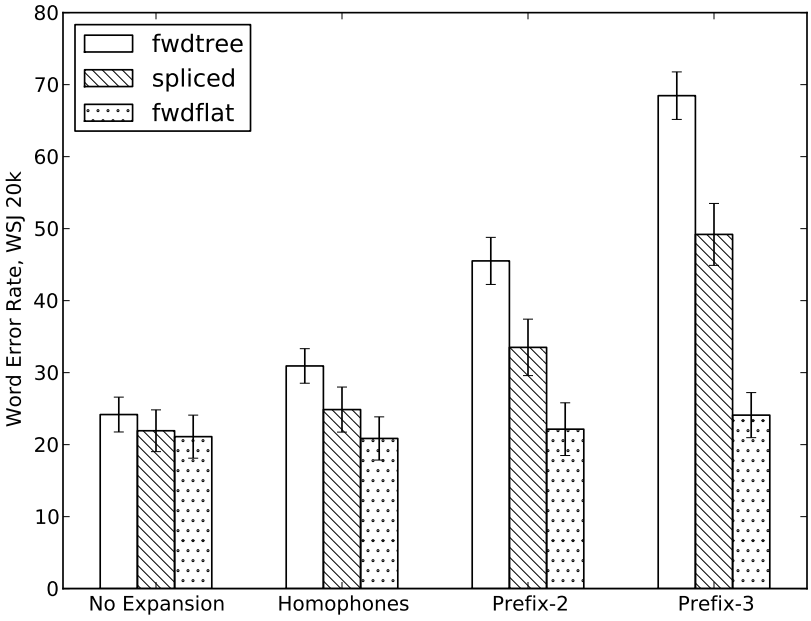

Figure 6.5: Hypothesis splicing alone does not fully recover from the errors introduced by vocabulary optimization

possible to use the results of domain-adaptive on-device recognition to accelerate server-based recognition, and in doing so, reduce the total latency of the system.

This implies the construction of a distributed system for speech recognition, where different components of the system reside on physically separate hosts connected by a network of some sort. Thankfully, the architecture described in this chapter so far is equally applicable to distributed recognition. This section describes the design for a distributed version of MULTISPHINX based on a *remote arc buffer* architecture for distributed speech recognition. There is currently no working implementation of this system; rather, this is a case study of the considerations for building such a system, and as such should be considered as future work.

6.5.1 Remote Arc Buffer Design

The remote arc buffer presents the same producer and consumer interfaces as the regular arc buffer. Producers register a backpointer table with the remote arc buffer at initialization time and add arcs to it using `arc_buffer_producer_sweep()`, while consumers wait on the arc buffer with `arc_buffer_consumer_wait()` and iterate over arcs using the same arc structures and iterator functions. The difference, obviously, is that the remote arc buffer is split into two separate objects, one for the producer and one for the consumer.

Internally, the arc and frame tables are split between the producer and consumer objects. The producer stores the active (incomplete) arcs, while the consumer stores the committed arcs. The producer sorts the

arcs by start frame, serializes them, and forwards them to the consumer over the network.

Due to the approximate and real-time nature of speech recognition, the robust nature of the Sphinx short-listing and lattice generation algorithms, and the fact that arcs in the arc buffer describe their own start and end points, there is no requirement that arcs arrive in exact order, or that all arcs arrive. As such, the remote arc buffer protocol can be datagram-oriented rather than stream-oriented, and should ideally be implemented on standard media streaming and control protocols such as RTP and SIP or RTSP, such as are commonly used in Voice over IP applications.

Although minimizing latency is the primary concern here (and another reason why a datagram protocol is preferred), it is also desirable to minimize the bitrate of the serialized arc stream. One reason in particular for this is that, as should be evident from the previous sections, many MultiSphinx configurations will require that the original audio or features be transmitted along with the arc information. The transmission time for a frame, as such, can be non-negligible.

6.5.2 Compression of Arc Buffer Data

Given that we do not enforce reliable transmission of arcs, it is not possible to use an adaptive, universal compression algorithm such as LZ77 [Ziv & Lempel, 1977] or one of its many variants (DEFLATE, gzip, etcetera). However, the remote arc buffer algorithm assumes a considerable amount of shared information between the client and server, in particular, an estimate of the probability distribution over words in the form of the language model. In addition, we assume that each packet corresponds to a unique start frame (though there may be many packets per frame) which allows us to encode many internal values as deltas.

Despite this, there is the possibility that the distribution of words, durations, and scores is not well modeled by this external information. As shown in Figure 5.5, the access patterns for N-Grams in the decoder are quite different from those we see when evaluating text input. It stands to reason, then, that the frequency distribution of word hypotheses in the arc buffer may be quite different from that in the language model. On the other hand, while the access patterns for N-Grams reflect words which were *evaluated*, the arc buffer contains words which were actually recognized. Indeed, when we compare the word frequencies in the arc buffers for the November '93 WSJ development set with those in the CSR-I language model training corpus, the rank-frequency curves, shown in Figure 6.6, are quite similar. More importantly, when we compare the frequencies of the word types in the development and training sets, as shown in Figure 6.7, we see that despite some noise, they are also similar.

More usefully, we can simply calculate the cross-entropy from the unigram distribution in language model training set to that of words in the arc buffers, as shown in Equation 6.2. For the standard WSJ `bcb05cnp` language model, the cross-entropy from the unigram distribution to the distribution of words in the arc buffers for the development set is 8.11 bits, which is the lower bound on the average number of bits needed to represent a word given the model. This is extremely close to the entropy of the empirical distribution of words in the arc buffers, which is 7.96 bits, and thus we can conclude that, at least for this test set, the language model used in decoding is an effective model for compressing the words in the arc buffer.

$$CH(LM; AB) = -\sum_{w \in AB} \frac{c_{AB}(w)}{N_{AB}} \log P_{LM}(w) \tag{6.2}$$

However, quite aside from the overall distribution of words in an arc buffer, we have attested that for

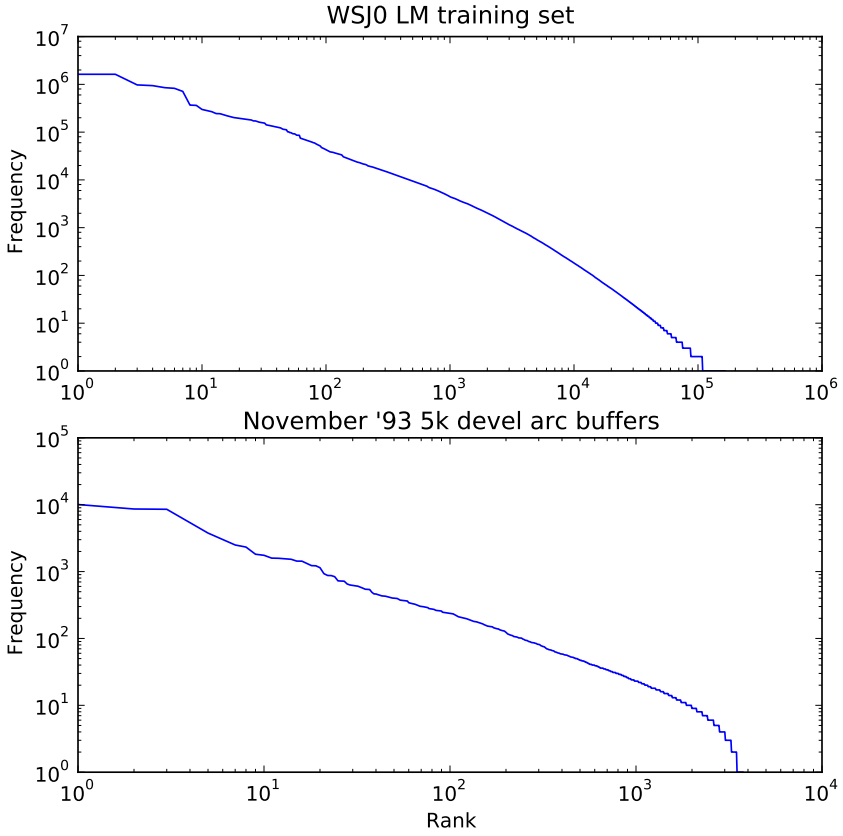

Figure 6.6: Rank-frequency curves of vocabulary words in text versus arc buffers

any given start frame, there are nearly always many outgoing arcs for the same word. This fact is used in the POCKETSPHINX lattice generation algorithm, as shown in Listing 6.4, to avoid explicitly creating links for all exits of a node until the adjacent nodes have been created. In that algorithm, references to the backpointer table are used to recover the acoustic scores for these implicit arcs. Since we do not have access to the backpointer table, we must store each outgoing arc and its scores explicitly in the arc buffer. Nonetheless, this fact can be used to effectively compress the arc buffer by grouping together arcs with the same word identity, and by encoding their respective scores as unsigned deltas from the shortest arc.

As mentioned earlier, the durations of arcs are also predictable to some extent given the shared information between the client and server. First, since the duration of a word can never be less than the number of non-skipped HMM states in that word, we can immediately subtract it from the duration. Second, the distributions of duration appear to fall into two broad classes, where we see a huge spike at three frames (the minimum duration allowed by the topology) accompanied by a second, wide distribution with a peak somewhat later. When the minimum distribution is subtracted from the duration of each word, however, this becomes a unimodal geometric distribution.

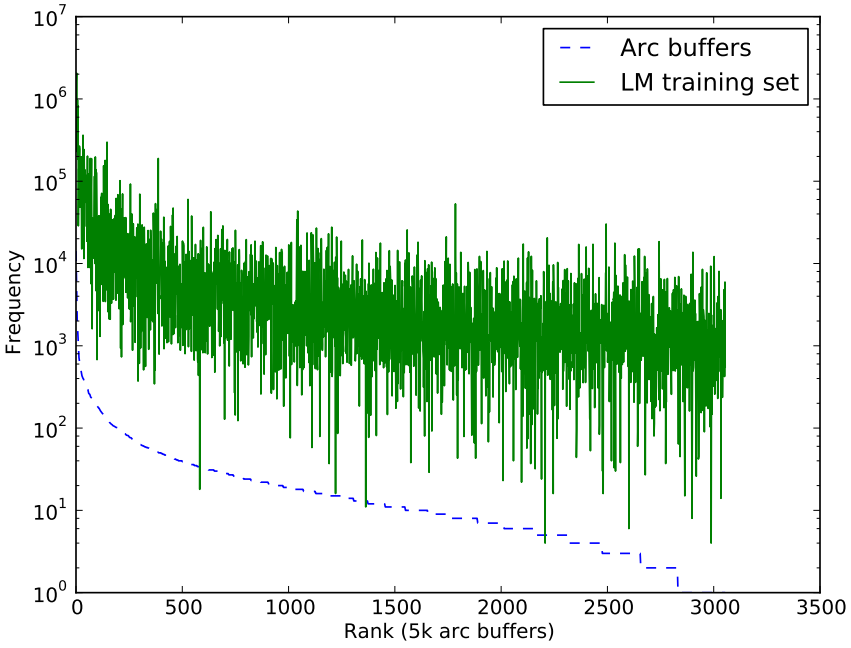

Figure 6.7: Word types in Nov '93 5k test set are distributed similarly to arc buffers

Outgoing arcs for a particular frame are grouped by word ID. A set of arcs with the same word ID is roughly equivalent to a link in a Sphinx lattice. We refer to it here as an "arc bundle". The word ID is compressed using Huffman coding over the unigram distribution in the language model, as described above. Since we do not have an *a priori* estimate of the distribution of durations, and since the protocol is stateless and unreliable, we must either use a universal code for them, or transmit (and periodically retransmit, if an adaptive code is used) the codebook. This is because the client and server are not guaranteed to have seen the same sequence of arcs at any given point in transmission. If the codebook is sent from client to server, it must be sent reliably, and it must be received before all data using the code, which can potentially incur considerable latency.

For this reason we use a universal code for the durations, specifically the exp-Golomb code[1]. The durations for an arc bundle are divided into the initial duration, which is computed as the difference between the current frame (shared by all bundles) and the first exit frame of this bundle. The minimum duration of the word in question is subtracted from this. The distributions of initial and delta durations for the November '93 development set are shown in Figure 6.8, along with the maximum likelihood estimates for the geometric distribution for each of them. For any geometric distribution there exists an optimal Golomb code [Witten et al., 1999] (though not necessarily the exp-Golomb code), and as such they are a good choice for duration information.

Specifically, the exponential Golomb code with $k = 0$ is chosen here for its simplicity, although it is

[1]Technically, we use the exp-Golomb code for the initial durations and the Elias-Gamma code for the deltas.

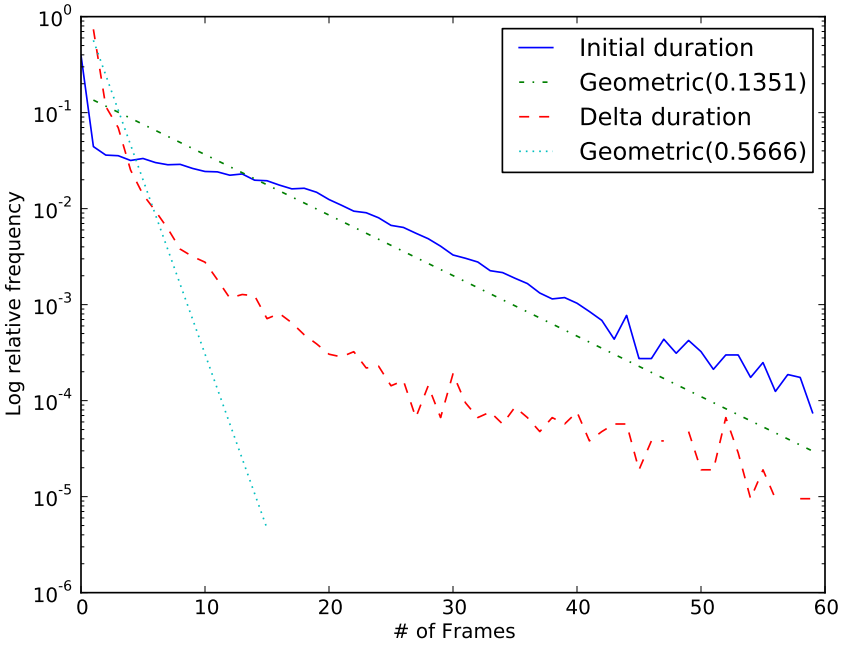

Figure 6.8: Initial durations of arcs have a distinct distribution from delta durations

not theoretically optimal. The codeword for a given integer is separated into a unary-coded exponent and a truncated binary-coded mantissa. To find the codeword for the initial duration, we add one to the value to be encoded, take the floor of its logarithm in base 2, and emit that number of zeros, followed by the binary representation of the value plus one starting with the high non-zero bit. Since the minimum delta duration is one, we encode them in nearly the same way, but without adding one to obtain the mantissa. This is illustrated in Table 6.1, with the exponent and mantissa separated by a binary point to illustrate the structure of the code.

Initial duration	Delta duration	Binary codeword
0	1	1
1	2	0.10
2	3	0.11
3	4	00.100
42	43	00000.101011
69	70	000000.1000110

Table 6.1: Exponential Golomb codewords

In practice, although the exponential Golomb code is able to represent any non-negative integer, extremely long durations and deltas are undesirable for practical reasons, since as we saw in Section 6.3.2, it is necessary to know the arcs in the arc buffer ending at a given frame in order to generate lattice nodes

starting in that frame. This means that we cannot release memory associated with an arc until its end frame has been encountered. In addition, there is a physical limit to the length of words which can be produced by the human vocal apparatus, therefore extremely long words in the arc buffer cannot correspond to relevant words in the recognition result, and may in fact be an indication of problems with the acoustic model. In practice these words are nearly all silences, which allows us to avoid them by using a "silence cutting" technique in the first stage search originally developed to constrain the size of the backpointer table.

The acoustic scores present a similar situation to the durations, where the distribution is not known *a priori*, and an adaptive coding scheme is not possible due to the statelessness of the protocol. Since, in theory, path scores for a given word HMM are non-increasing over subsequent frames, we can use a very similar delta coding technique to that used for durations. This is complicated somewhat by the fact that a given word HMM has multiple right contexts, which means that the path score may actually increase between the best instances of a given word in adjacent frames. Therefore, we need to store the best path score for an arc bundle separately from the initial arc's score. In addition, it is necessary to encode the right context deltas efficiently.

The distribution of delta scores is not nearly as compressible as that of durations. After excluding bundles with a single arc and words with a single right context score, the distributions over path score deltas and right context deltas are shown in Figure 6.9. In both cases a delta of zero is roughly a hundred times more likely than any other delta score. The non-zero right context deltas have a Poisson-like distribution, while the non-zero path score deltas are more closely approximated by a two-sided geometric distribution, as shown in Figure 6.10. In both cases, the path score deltas in particular, the distribution is relatively uniform; the entropy of the empirical distribution is 10.94 bits for non-zero path score deltas and 8.8 bits for non-zero right context score deltas. With zeros included the entropy is 9.13 and 8.47 bits, respectively. This indicates that simply encoding zeros separately can achieve a considerable savings.

6.5.3 Network Protocol Considerations

Since, unfortunately, arcs are bursty - that is, we typically go many frames with no output arcs, followed by a large number of arcs in the same frame, the arcs for a given frame must be divided into multiple packets. The payload size should kept to 512 bytes to avoid fragmentation when sent over links with small MTUs.

In addition, there is the question of dealing with arcs which arrive out of sequence, or which fail to arrive. As to the problem of missing arcs, ironically, the transmission of intermediate results is most robust where those results are the least certain, namely in regions of high lattice density. One potential solution to this problem is simply to add redundancy to the protocol by retransmitting previous arcs during the low-density regions of the utterance. In both the shortlisting and incremental lattice generation algorithms described previously, this has no ill effect should these arcs have already been processed.

6.6 Summary

In Chapter 5 we primarily addressed the issue of *load balancing* within a multi-pass recognizer, by making recoverable approximations to the vocabulary of earlier passes. This chapter, by contrast, is primarily concerned with the issue of *latency* and the implementation of a speech recognizer as an anytime system.

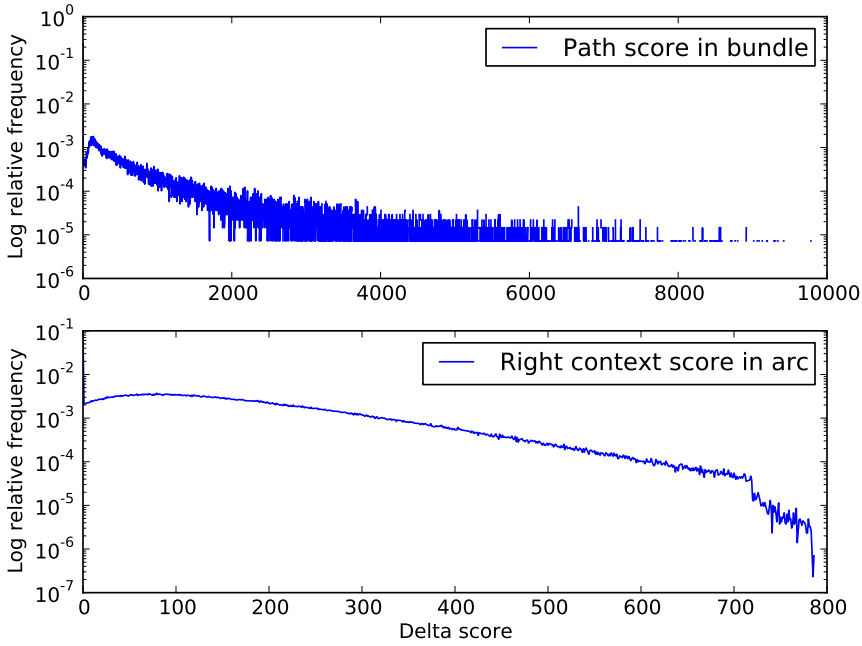

Figure 6.9: Distribution of arc score deltas, which are overwhelmingly zero

We have presented a novel architecture for concurrent multi-pass speech recognition, along with a detailed description of the algorithms and data structures which enable it.

The use of multiple decoders in sequence, where the earlier stages restrict the search space so as to allow more complex models to be used, is well established in speech recognition [Nguyen et al., 1993, Ljolje et al., 2000, Levit et al., 2009]. More recently this idea has been generalized to other domains of machine learning, pattern recognition, and natural language processing in the form of "coarse to fine inference" algorithms [Petrov, 2009]. Multi-threaded speech recognition has also been investigated to some extent [Phillips & Rogers, 1999, Ishikawa et al., 2006].

When viewed against the background of these related works, the distinguishing characteristic of the architecture presented in this chapter is its incremental, anytime nature. Specifically, the search components in MULTISPHINX are themselves stand-alone decoders whose output is usable by higher-level systems. Furthermore, through *hypothesis splicing*, these systems are able to obtain incremental results which reflect the best currently available information with the lowest possible latency. This not only addresses a long-standing limitation of the POCKETSPHINX recognizer, but demonstrates the viability of the anytime principle for speech recognition.

With respect to another long-standing limitation of POCKETSPHINX, namely the generation of word lattices, we find that the *arc buffer* mechanism developed for communication between concurrent passes has the desirable side effect of greatly reducing dynamic memory consumption for long utterances, and also facilitates the incremental construction of lattices with explicit N-Gram history. Finally, we return to the information-theoretic foundations discussed in 2.4 with an investigation of issues involved in moving from

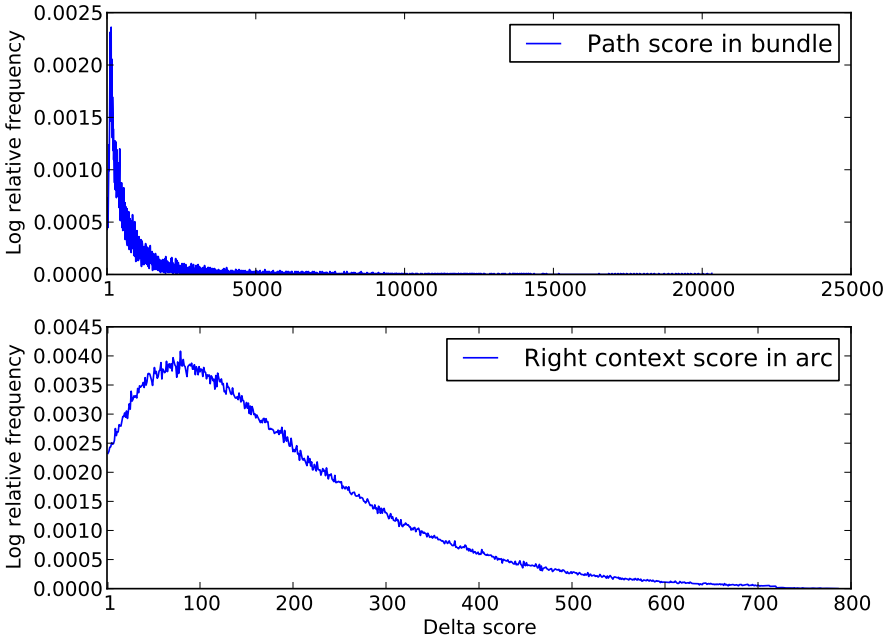

Figure 6.10: Distribution of non-zero arc score deltas

the current multi-threaded paradigm to a distributed one, where the compact representation of intermediate results is of paramount importance.

Chapter 7

Summary

The first major contribution of this thesis is the use of the vocabulary as a variable for tuning the performance of a speech recognizer, specifically with the aim of balancing computational load betwen two passes of recognition. We began with a very uneven breakdown of the task in the POCKETSPHINX recognizer which was the starting point for this thesis work, as shown in Figure 7.1. The net effect of the techniques presented in Chapter 5, when implemented and tested inside the MULTISPHINX sytem, are shown in Firgure 7.2.

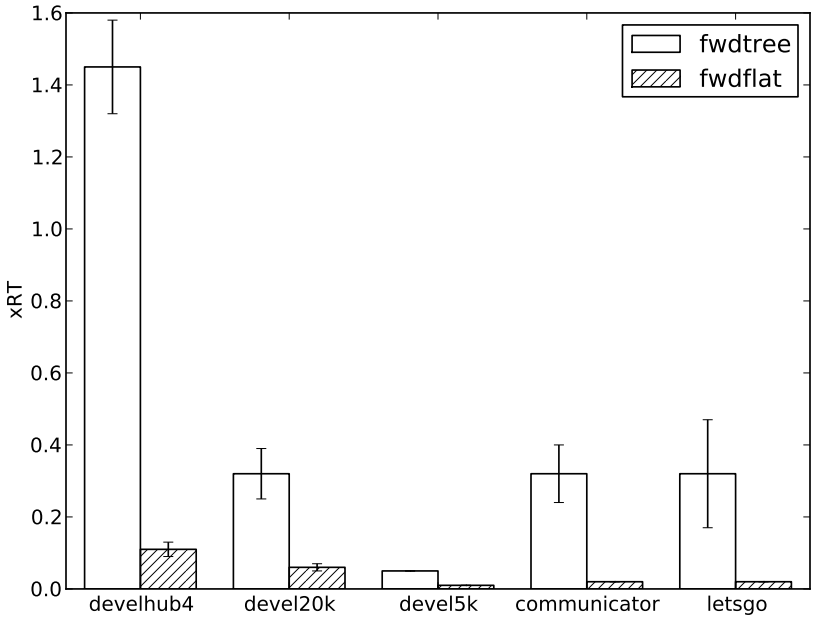

Figure 7.1: The baseline distribution of load between search passes in POCKETSPHINX is highly unequal

In Section 1.2 we described the notion of a "black box" recognizer on which domain and task models can be layered as post-processing stages. Subsequent chapters explored a diverse set of techniques which could be employed to achieve this goal.

The most important observation we can draw from these experiences is that such a system is best approached from the top down. That is, if the goal is to create a compact set of models for a system which includes a expansion or error correction component, it is still much more effective to begin with a large model (be it a language model or a vocabulary) and remove the most redundant elements first. This is the premise behind existing techniques such as entropy-based language model pruning [Stolcke, 1998]. The contribution of this thesis here is, first, to extend this principle to the vocabulary itself, and second, to extend the definition of redundancy to include not merely elements of the model which can be merged without immediately affecting accuracy, but also those which can be merged without a general loss of *information*. We distinguish between those ambiguities which can and cannot be resolved by application of a higher-order model in subsequent processing. In addition, we introduce the concept of *residuals*, which are effectively placeholders added to a model in order to prevent approximation at one level of representation from incurring additional confusability at a different level.

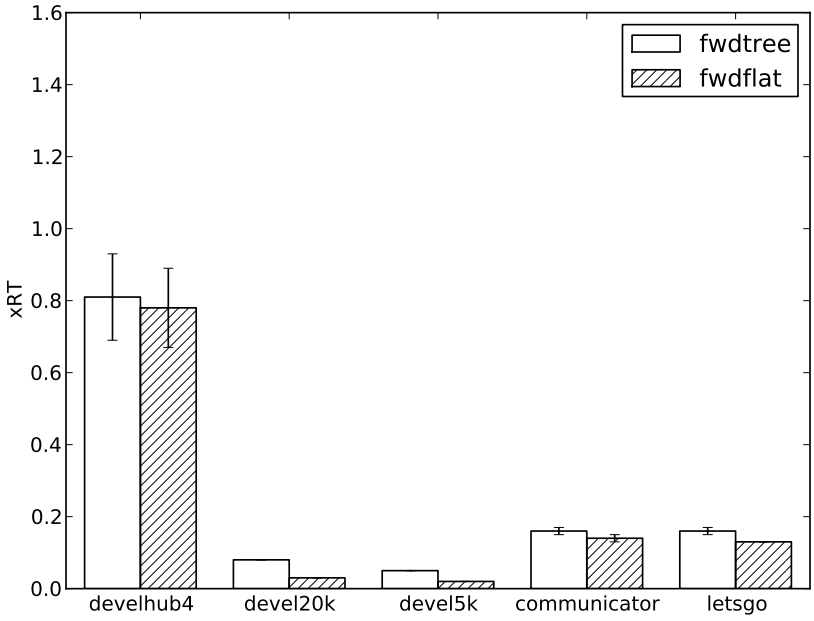

Figure 7.2: After implementing prefix merging and expansion in MULTISPHINX, the distribution of load between search passes is considerably more equal, at no cost to accuracy

We then demonstrate how these concepts can be integrated into a concurrent architecture in order to achieve a form of load balancing. We show that in some cases removing redundancy from the initial pass of recognition in a multi-pass system can not only shift the computational burden to subsequent passes but also improve the performance of the system as a whole. When we extend this architecture to a distributed one, paying particular attention to compact representation of intermediate results, we find that the pruning techniques have the additional benefit of reducing latency and bandwidth.

All of this leads back to a concept introduced in passing in Section 2.3, which is that automatic speech recognition can be thought of as a kind of lossy data compression. In the distributed version of the system, this is quite literally the case, as we leverage the known probability distributions over various aspects of the speech signal to compress the intermediate results. The techniques described in this thesis rely on an implicit and often informal notion of the "confusability" of the current representation of speech. However, in information theoretic terms, confusability is synonymous with uncompressibility, which is merely another way of saying the *cross-entropy* between the model used for compression (which may be implicit, in the case of universal codes) and the data, which in turn is synonymous with *perplexity*. The contribution of this thesis is, first, an examination of the relationship between perplexity of the language model and confusability in the acoustic model, and second, an implementation of techniques to reduce the former without increasing the latter.

A larger problem, which remains to be addressed in future work, is how to apply these optimizations to the recognizer as a whole. In this thesis, both due to the nature of the SPHINX platform and the need for

memory efficiency, we have constructed algorithms which operate independently on the lexicon, language model, and acoustic model. This is despite the fact that the major contribution described above relies on a conception of the effects on the whole system incurred by approximations in one component or another. Reformulating the algorithms discussed in this thesis as operations on a composed finite-state transducer representation of the system holds the promise of not only achieving greater efficiencies, but also providing a more formal and abstract characterization of these techniques. This was touched on briefly in Section 4.6.1, in the context of an ineffective error modeling technique. However, given the findings in Chapters 5 and 6, there is a clear need for a way to quantify overall confusability of the system, and to better understand the interaction between the lexicon and the language model. Converting the proposed algorithms to operate on the composition of the lexicon and the language model may further this understanding.

Finally, the question remains as to the relevance of this work for the broader field of natural language processing and understanding. We have deliberately tried, in this thesis, to avoid speculation about linkages between the architecture proposed here and higher level knowledge sources. However, the ultimate inspiration for this work is the need to reduce latency in real-time speech communication between humans and computers. It is our expectation that the availability of an anytime system, not to mention an architecture upon which pipelined, concurrent decoders of various descriptions can be built, will encourage further research into real-time and anytime natural language understanding systems. From a scientific, rather than an engineering perspective, the empirical characterization of the limits of online and incremental processing found in Chapter 6 is relevant to the nature of spoken language. In particular, we are anxious to understand the limitations of syntactic and semantic analysis when an attempt is made to implement them under anytime conditions. At a lower level, the question of the utility and salience of whole-sentence versus history-based language models is not one that we take a strong position on in this thesis, and the architecture described here is not fundamentally oriented towards one or the other; we do, however, make the assumption that useful information is available at a subsentential and incremental level.

The questions posed here for future study are somewhat removed from the low-level nature of much of this dissertation, yet they are central to its stated goals. As such the work described here is merely a foundation on which many things are yet to be built; by beginning to think differently about the lowest level of the architecture of machine speech understanding and interaction, we hope to propagate these ideas of incremental processing, error recovery, and anytime inference to the higher levels on which our future work will ultimately take place.

Bibliography

[Acero, 1990] Acero, A. (1990). *Acoustical and Environmental Robustness in Automatic Speech Recognition*. PhD thesis, Carnegie Mellon University.

[Allauzen et al., 2003] Allauzen, C., Mohri, M., & Roark, B. (2003). Generalized algorithms for constructing statistical language models. In *Proceedings of ACL 2003*.

[Allauzen et al., 2009] Allauzen, C., Riley, M., & Schalkwyk, J. (2009). A generalized composition algorithm for weighted finite-state transducers. In *Proceedings of Interspeech 2009*.

[Allauzen et al., 2007] Allauzen, C., Riley, M., Schalkwyk, J., Skut, W., & Mohri, M. (2007). OpenFst: A general and efficient weighted finite-state transducer library. In *Proceedings of the Twelfth International Conference on Implementation and Application of Automata (CIAA 2007)*.

[Allen, 1994] Allen, J. B. (1994). How do humans process and recognize speech? *IEEE Transactions on Speech and Audio Processing*, 2(4), 567–577.

[Bacchiani & Roark, 2003] Bacchiani, M. & Roark, B. (2003). Unsupervised language model adaptation. In *Proceedings of ICASSP 2003*.

[Bahl et al., 1986] Bahl, L. R., Brown, P. F., de Souza, P. V., & Mercer, R. L. (1986). Maximum mutual information estimation of Hidden Markov Model parameters for speech recognition. In *Proceedings of ICASSP '86*.

[Baum & Petrie, 1966] Baum, L. E. & Petrie, T. (1966). Statistical inference for probabilistic functions of finite state Markov chains. *Annals of Manthematical Statistics*, 37, 1554–1563.

[Berton et al., 1996] Berton, A., Fetter, P., & Regel-Brietzmann, P. (1996). Compound words in large-vocabulary German speech recognition systems. In *Proceedings of ICSLP '96*.

[Bohus et al., 2007] Bohus, D., Grau, S., Huggins-Daines, D., Keri, V., Krishna, G., Kumar, R., Raux, A., & Tomko, S. (2007). Conquest - an open-source dialog system for conferences. In *Proceedings of HLT-NAACL 2007* Rochester, NY, USA.

[Brants et al., 2007] Brants, T., Popat, A. C., Xu, P., Och, F. J., & Dean, J. (2007). Large language models in machine translation. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)* (pp. 858–867).

[Brown, 1987] Brown, P. F. (1987). *The Acoustic-Modeling Problem in Automatic Speech Recognition*. PhD thesis, Carnegie Mellon University.

[Bulyko, 2010] Bulyko, I. (2010). Speech recognizer optimization under speed constraints. In *Proceedings of Interspeech*.

[CC-CEDICT, 2009] CC-CEDICT (2009). CC-CEDICT Online Chinese-English Dictionary. http://usa.mdbg.net/chindict/chindict.php?page=cc-cedict.

[Chen & Goodman, 1998] Chen, S. F. & Goodman, J. (1998). *An Empirical Study of Smoothing Techniques for Language Modeling*. Technical Report TR-10-98, Harvard University.

[Colthurst et al., 2007] Colthurst, T., Arvizo, T., Kao, C.-L., Kimball, O., Lowe, S., Miller, D. R. H., & Sciver, J. V. (2007). Parameter tuning for fast speech recognition. In *Proceedings of Interspeech*.

[Cover & Thomas, 2006] Cover, T. & Thomas, J. (2006). *Elements of Information Theory*. Wiley-Interscience, 2nd edition.

[Dempster et al., 1977] Dempster, A. P., Laird, N., & Rubin, D. (1977). Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society*, 39(B).

[Digalakis et al., 1996] Digalakis, V., Monaco, P., & Murveit, H. (1996). Genones: Generalized mixture tying in continuous Hidden Markov Model based speech recognizers. *IEEE Transactions on Speech and Audio Processing*, 4(4), 281–289.

[Dijkstra, 1959] Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1, 269–271.

[Fetter, 1998] Fetter, P. (1998). *Detection and Transcription of OOV Words*. Technical report, Daimler-Benz AG.

[Galescu, 2003] Galescu, L. (2003). Recognition of out-of-vocabulary words with sub-lexical language models. In *Proceedings of Eurospeech*.

[Gallwitz et al., 1996] Gallwitz, F., Noth, E., & Niemann, H. (1996). A category based approach for recognition of out-of-vocabulary words. In *Proceedings of ICSLP '96* (pp. 228–231).

[Geutner et al., 1998] Geutner, P., Finke, M., & Waibel, A. (1998). Phonetic-distance-based hypothesis driven lexical adaptation for transcribing multilingual broadcast news. In *Proceedings of ICSLP '98*.

[Glosser et al., 1998] Glosser, G., Friedman, R. B., Kohn, S. E., Sands, L., & Grugan, P. (1998). Cognitive mechanisms for processing nonwords: Evidence from alzheimer's disease. *Brain and Language*, 63, 32–49.

[Gopinath, 1998] Gopinath, R. (1998). Maximum likelihood modeling with gaussian distributions for classification. In *Proceedings of ICASSP '98*, volume 2 (pp. 661–664vol.2).

[Haeb-Umbach & Ney, 1994] Haeb-Umbach, R. & Ney, H. (1994). Improvements in beam search for 10 000-word continuous-speech recognition. *IEEE Transactions on Speech and Audio Processing*, 2(2), 353–356.

[Hain, 2005] Hain, T. (2005). Implicit modelling of pronunciation variation in automatic speech recognition. *Speech Communication*, 46(2), 171–188.

[Hermansky, 1990] Hermansky, H. (1990). Perceptual linear predictive (PLP) analysis of speech. *Journal of the Acoustical Society of America*, 87(4), 1738–1752.

[Hermansky & Morgan, 1994] Hermansky, H. & Morgan, N. (1994). RASTA processing of speech. *IEEE Transactions on Speech and Audio Processing*, 2(4), 578–589.

[Huang et al., 1993] Huang, X., Alleva, F., Hon, H.-W., Hwang, M.-Y., & Rosenfeld, R. (1993). The SPHINX-II speech recognition system: an overview. *Computer Speech and Language*, 7(2), 137–148.

[Huang, 1989] Huang, X. D. (1989). *Semi-continuous Hidden Markov Models for Speech Recognition*. PhD thesis, University of Edinburgh.

[Huggins-Daines et al., 2006] Huggins-Daines, D., Kumar, M., Chan, A., Black, A., Ravishankar, M., & Rudnicky, A. (2006). PocketSphinx: A free, real-time continuous speech recognition system for hand-held devices. In *Proceedings of ICASSP 2006*.

[Huggins-Daines & Rudnicky, 2007] Huggins-Daines, D. & Rudnicky, A. I. (2007). Implicitly supervised language model adaptation for meeting transcription. In *Proceedings of HLT-NAACL 2007* Rochester, NY, USA.

[Huggins-Daines & Rudnicky, 2008] Huggins-Daines, D. & Rudnicky, A. I. (2008). Mixture pruning and roughening for scalable acoustic models. In *Proceedings of ACL-08 Workshop on Mobile Language Processing* Columbus, OH, USA.

[Huggins-Daines & Rudnicky, 2009] Huggins-Daines, D. & Rudnicky, A. I. (2009). Combining mixture weight pruning and quantization for small-footprint speech recognition. In *Proceedings of ICASSP 2009* Taipei, Taiwan.

[Hwang, 1993] Hwang, M.-Y. (1993). *Subphonetic Acoustic Modeling for Speaker-Independent Continuous Speech Recognition*. PhD thesis, Carnegie Mellon University.

[Ishikawa et al., 2006] Ishikawa, S., Yamabana, K., Isotani, R., & Okamura, A. (2006). Parallel LVCSR algorithm for cellphone-oriented multicore processors. In *Proceedings of ICASSP*.

[Itakura, 1975] Itakura, F. (1975). Minimum prediction residual principle applied to speech recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 23(1), 67–72.

[Jelinek et al., 1991] Jelinek, F., Merialdo, B., Roukos, S., & Strauss, M. (1991). A dynamic language model for speech recognition. In *Proceedings of HLT '91* (pp. 293–295). Morristown, NJ, USA.

[Kanthak & Ney, 2002] Kanthak, S. & Ney, H. (2002). Context-dependent acoustic modeling using graphemes for large vocabulary speech recognition. In *Proceedings of ICASSP 2002*.

[Kanthak et al., 2000] Kanthak, S., Sixtus, A., Morlau, S., Schlüter, R., & Ney, H. (2000). Fast search for large vocabulary speech recognition. In *ISCA ITRW Automatic Speech Recognition 2000*.

[Katz, 1987] Katz, S. M. (1987). Estimation of probabilities from sparse data for the language model component of a speech recognizer. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 35(3), 400–401.

[Kemp et al., 1996] Kemp, T., Jusek, A., Systems, I., Ag, L., & Informatik, A. (1996). Modelling unknown words in spontaneous speech. In *Proceedings of ICASSP '96* (pp. 530–533).

[Kemp & Schaaf, 1997] Kemp, T. & Schaaf, T. (1997). Estimating confidence using word lattices. In *Proceedings of Eurospeech '97*.

[Lavie et al., 2003] Lavie, A., Vogel, S., Levin, L., Peterson, E., Probst, K., Font-Llitjós, A., Reynolds, R., Carbonell, J., & Cohen, R. (2003). Experiments with Hindi-to-English Transfer-based MT under a Miserly Data Scenario. *ACM Transactions on Asian Language Information Processing*, 2(2).

[Levin et al., 2000] Levin, E., Narayanan, S., Pieraccini, R., Biatov, K., Bocchieri, E., Fabbrizio, G. D., Eckert, W., Lee, S., Pokrovsky, A., Rahim, M., Ruscitti, P., & Walker, M. (2000). The AT&T-DARPA Communicator mixed-initiative spoken dialog system. In *Proceedings of ICSLP*.

[Levit et al., 2009] Levit, M., Chang, S., & Buntschuh, B. (2009). Garbage modeling with decoys for a sequential recognition scenario. In *Proceedings of ASRU 2009*.

[Ljolje et al., 2000] Ljolje, A., Riley, M. D., Hindle, D. M., & Sproat, R. W. (2000). The AT&T LVCSR-2000 system. In *Proceedings of NIST 2000 Speech Transcription Workshop*.

[MacIntyre, 1998] MacIntyre, R. (1998). 1996 CSR HUB4 Language Model. Linguistic Data Consortium.

[Mangu, 1999] Mangu, L. (1999). *Finding Consensus in Speech Recognition*. PhD thesis, Johns Hopkins University.

[Masuko et al., 1996] Masuko, T., Tokuda, K., Kobayashi, T., & Imai, S. (1996). Speech synthesis using HMMs with dynamic features. In *Proceedings of ICASSP '96*, volume 1 (pp. 389–392).

[Matusov et al., 2005] Matusov, E., Schlüter, R., & Ney, H. (2005). Phrase-based translation of speech recognizer word lattices using loglinear model combination. In *Proceedings of ASRU 2005*.

[Mohri et al., 2000] Mohri, M., Pereira, F., & Riley, M. (2000). Weighted finite state transducers in speech recognition. In *Proceedings of ISCA ITRW Automatic Speech Recognition 2000*.

[Ney & Aubert, 1994] Ney, H. & Aubert, X. (1994). A word graph algorithm for large vocabulary, continuous speech recognition. In *Proceedings of ICSLP '94*.

[Nguyen et al., 1993] Nguyen, L., Schwartz, R., Kubala, F., & Placeway, P. (1993). Search algorithms for software-only real-time recognition with very large vocabularies. In *Proceedings of HLT '93*.

[Palmer, 2001] Palmer, D. D. (2001). *Modeling Uncertainty for Information Extraction From Speech Data*. PhD thesis, University of Washington.

[Petrov, 2009] Petrov, S. (2009). *Coarse-to-Fine Natural Language Processing*. PhD thesis, University of California at Berkeley.

[Phillips & Rogers, 1999] Phillips, S. & Rogers, A. (1999). Parallel speech recognition. *International Journal of Parallel Programming*, 27(4), 257–288.

[Pitz et al., 2000] Pitz, M., Wessel, F., & Ney, H. (2000). Improved MLLR speaker adaptation using confidence measures for conversational speech recognition. In *Proceedings of ICSLP 2000*.

[Placeway et al., 1997] Placeway, P., Chen, S., Eskenazi, M., Jain, U., Parikh, V., Raj, B., Ravishankar, M., Rosenfeld, R., Seymore, K., Siegler, M., Stern, R., & Thayer, E. (1997). The 1996 hub-4 sphinx-3 system. In *Proceedings of the 1997 DARPA Speech Recognition Workshop*.

[Povey, 2003] Povey, D. (2003). *Discriminative Training for Large Vocabulary Speech Recognition*. PhD thesis, Cambridge University.

[Pusateri & Thong, 2001] Pusateri, E. & Thong, J. M. V. (2001). N-best list generation using word and phoneme recognition fusion. In *Proceedings of Eurospeech 2001*.

[Qin & Rudnicky, 2010] Qin, L. & Rudnicky, A. I. (2010). The effect of lattice pruning on mmie training. In *Proceedings of ICASSP 2010* (pp. 4898–4901).

[Rabiner, 1989] Rabiner, L. R. (1989). A tutorial on Hidden Markov Models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2), 257–286.

[Rastrow et al., 2009] Rastrow, A., Sethy, A., & Ramabhadran, B. (2009). A new method for OOV detection using hybrid word/fragment system. *Proceedings of ICASSP 2009*, (pp. 3953–3956).

[Raux et al., 2006] Raux, A., Bohus, D., Langner, B., Black, A. W., & Eskenazi, M. (2006). Doing Research on a Deployed Spoken Dialogue System: One Year of Let's Go! Experience. In *Proceedings of Interspeech 2006*.

[Ravishankar, 1996] Ravishankar, M. K. (1996). *Efficient Algorithms for Speech Recognition*. PhD thesis, Carnegie Mellon University.

[Ringger & Allen, 1996] Ringger, E. K. & Allen, J. F. (1996). Error correction via a post-processor for continuous speech recognition. In *Proceedings of ICASSP '96*.

[Rosenfeld, 1995] Rosenfeld, R. (1995). Optimizing lexical and N-Gram coverage via judicious use of linguistic data. In *Proceedings of Eurospeech '95*.

[Rosenfeld, 2000] Rosenfeld, R. (2000). Two decades of statistical language modeling: Where do we go from here? *Proceedings of the IEEE*, 88(8).

[Sagerer et al., 1996] Sagerer, G., Rauenstrauch, H., Fink, G. A., Hildebrandt, B., Jusek, A., & Kummert, F. (1996). Incremental generation of word graphs. In *Proceedings of ICSLP '96*.

[Sainath, 2009] Sainath, T. N. (2009). Island-driven search using broad phonetic classes. In *Proceedings of ASRU 2009*.

[Schaaf, 2001] Schaaf, T. (2001). Detection of oov words using generalized word models and a semantic class language model. In *Proceedings of Eurospeech 2001*.

[Schalkwyk et al., 2003] Schalkwyk, J., Hetherington, L., & Story, E. (2003). Speech recognition with dynamic grammars using finite-state transducers. In *Proceedings of Eurospeech 2003* (pp. 1969–1972).

[Schultz & Waibel, 2000] Schultz, T. & Waibel, A. (2000). Polyphone decision tree specialization for language adaptation. In *Proceedings of ICASSP 2000*.

[Schultz & Yu, 2003] Schultz, T. & Yu, H. (2003). Enhanced tree clustering with single pronunciation dictionary for conversational speech recognition. In *Proceedings of EuroSpeech 2003*.

[Schwartz et al., 1996] Schwartz, R., Nguyen, L., & Makhoul, J. (1996). Multiple-pass search. In C.-H. Lee, F. K. Soong, & K. K. Paliwal (Eds.), *Automatic Speech and Speaker Recognition: Advanced Topics* chapter 18, (pp. 429–456). Springer.

[Shahaf & Amir, 2007] Shahaf, D. & Amir, E. (2007). Towards a theory of AI completeness. In *Proceedings of Commonsense 2007, 8th International Symposium on Logical Formalizations of Commonsense Reasoning*.

[Soltau et al., 2001] Soltau, H., Metze, F., Fügen, C., & Waibel, A. (2001). A one-pass decoder based on polymorphic linguistic context assignment. In *Proceedings of ASRU 2001* (pp. 214–217).

[Soltau & Saon, 2009] Soltau, H. & Saon, G. (2009). Dynamic network decoding revisited. In *Proceedings of ASRU 2009*.

[Sproat et al., 1999] Sproat, R., Black, A., Chen, S., Kumar, S., Ostendorf, M., & Richards, C. (1999). *Normalization of Non-Standard Words: WS '99 Final Report*. Technical report, Hopkins University.

[Stolcke, 1998] Stolcke, A. (1998). Entropy-based pruning of backoff language models. In *Proceedings of DARPA Broadcast News Transcription and Understanding Workshop* (pp. 270–274).

[Tam, 2009] Tam, Y.-C. (2009). *Rapid Unsupervised Topic Adaptation - a Latent Semantic Approach*. PhD thesis, Carnegie Mellon University.

[Wessel et al., 2001] Wessel, F., Schlüter, R., Macherey, K., & Ney, H. (2001). Confidence measures for large vocabulary continuous speech recognition. *IEEE Transactions on Speech and Audio Processing*, 9(3), 288–298.

[Witten et al., 1999] Witten, I. H., Moffat, A., & Bell, T. C. (1999). *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann.

[Young et al., 1989] Young, S. J., Russel, N. H., & Thornton, J. H. S. (1989). *Token Passing: A Simple Conceptual Model for Connected Speech Reccognition Systems*. Technical Report CUED/F-INFENG/TR38, Cambridge University.

[Zhu & Alwan, 2000] Zhu, Q. & Alwan, A. (2000). On the use of variable frame rate analysis in speech recognition. In *Acoustics, Speech, and Signal Processing, 2000. ICASSP '00. Proceedings. 2000 IEEE International Conference on*, volume 3 (pp. 1783–1786).

[Zilberstein, 1996] Zilberstein, S. (1996). Using anytime algorithms in intelligent systems. *AI Magazine*, (pp. 73–83).

[Ziv & Lempel, 1977] Ziv, J. & Lempel, A. (1977). A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3), 337–343.