Phonetic study and text mining of Spanish for English to Spanish translation system

A Thesis Presented by

Jorge Gilabert Hernando

in Partial Fulfillment of the Requirements for the Degree of

Enginyer de Telecomunicacions

at the

Universitat Politècnica de Catalunya

Thesis supervisors

Prof. Shri Narayanan

Dr. Panayiotis Georgiou

Signal Analysis and Interpretation Laboratory

Escola Tècnica Superior d'Enginyers de Telecomunicacions de Barcelona

Thesis Title: Phonetic study and text mining of Spanish for English to Spanish translation system.

Author: Jorge Gilabert Hernando

Supervisors: Proff. Shri Narayanan and Dr. Panayiotis Georgiou

Contents

List of Figures

1. Introduction

Today, building a new two-way speech-to-speech translation system is a task of many years' effort. In this thesis there were three main goals: Do research for generating a robust speech recognizer, build a text-to-text machine translator and put it together using a speech synthesizer to create the two-way speech-to-speech translation system.

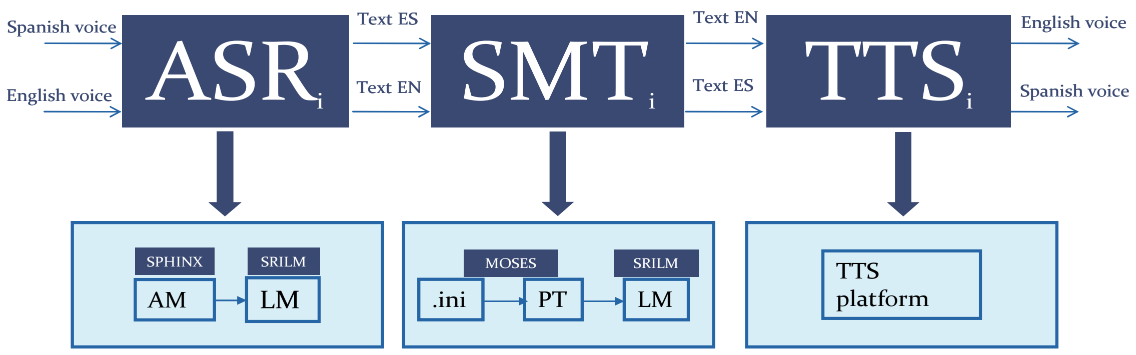
In order to achieve these goals in a short time, a baseline work has been established:

First of all, the CMU's (Carnegie Mellon University) Sphinx speech recognition system is freely available and currently is one of the most robust speech recognizers in English. Moreover, it is adaptable to Spanish without making many changes to the system.

Secondly, the Moses statistical machine translator is also freely available, and it gives acceptable results with any two languages as long as the parallel text is well prepared, relevant and focused in one concrete environment.

Finally, the USC (University of Southern California), where this thesis was developed, developed a two way speech-to-speech translation system. This project was called SpeechLinks and it worked in Farsi and in English. In the project described in this document, the SpeechLinks system is going to be adapted to work in Spanish and in English.

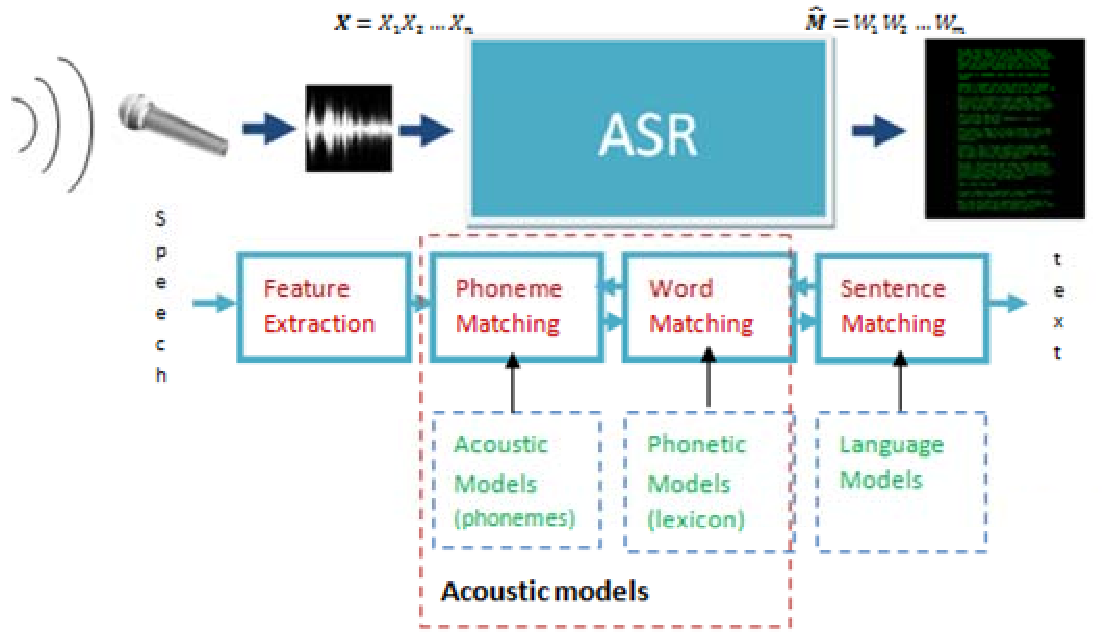
This document explains how all the blocks described in figure 1.1 work, and how they have been adapted to work together in creating a two-way speech-to-speech translation system.



1.1 Two-way Speech-to-Speech translation system

Automatic Speech Recognition

ASR (Automatic Speech Recognition) is a system that converts spoken words to written output. However, a number of well-known factors determine accuracy: those most noticeable are variations in context, in speaker and in environment.



0.1 Automatic Speech Recognizer

The challenge of an ASR (Automatic Speech Recognition) program [2] is for a given acoustic observation $X = X_1X_2...X_n$ to discover the corresponding word sequence $\hat{M} = W_1W_2...W_m$ that has the Maximum posterior probability $P(M/X)$.

$$\hat{M} = \arg\max_w P(M/X) = \arg\max_w \frac{P(X/M)P(M)}{P(X)}$$

0.2 Estimation of a word sequence

Since the maximization of figure 2.2 is carried out with the observation **X** fixed, the above maximization is equivalent to:

$$\hat{M} = \arg\max_w P(X/M)P(M)$$

$$P(X/M) = \sum_{i=1}^{L} P(X/P_i)P(P_i/M)$$

0.3 Acoustic Model equations

The practical objective is to build accurate **acoustic models**, $P(X/M)$, and **language models**, $P(M)$, that allow us to recognize the spoken language. For large-vocabulary speech recognition, since there are a large number of words, we need to decompose a word into a subword sequence. Consequently, $P(X/M)$ is closely related to phonetic modeling $P(P_l/M)$ and to the acoustic models of the phonemes $P(X/P_l)$ (figure 2.3). $P(X/M)$ should take into account speaker variations, pronunciation variations, environmental variations, and context-dependent phonetic coarticulation variations. Furthermore, any acoustic or language model will not meet the needs of real applications by itself; therefore, it is fundamental to dynamically adapt both language and acoustic models to maximize $P(M/X)$ while using spoken language systems. The decoding process of finding the best word sequence M to match the input speech signal X in speech recognition systems is more than a simple pattern recognition problem, since in continuous speech recognition there are an infinite number of word patterns to search [2].

Acoustic Models

Acoustic models include the representation of knowledge about acoustics, phonetics, microphone and environmental variability, gender and dialect differences among speakers.

An acoustic model is created by taking audio recordings of speech, and their text transcriptions, and using software to create statistical representations of the sounds that make up each word.

The audio recordings of speech can be encoded at different sampling rates (for example, 8kHz, 16kHz, 22.5kHz, 32kHz, 44.1kHz, 48kHz or 96kHz), and different bits per sample (for example, 8-bits, 16-bits or 32-bits). Speech recognition engines work best if the acoustic model they use was trained with speech audio which was recorded at the same sampling rate/bits per sample as the speech being recognized.

The lexicon used in the language model is also important: it includes the phonetic rules of the language for which such speech is being recognized, and it helps to convert recognized phonemes into words.

Language Models

Language model refers to a system’s knowledge of what constitutes a possible word, what words are likely to co-occur, and in what sequence.

For the language model, two things are fundamental: the grammar and the parsing algorithm. The grammar is a formal specification of the permissible structures for the language. The parsing technique is the method of analyzing the sentence to see if its structure is compliant with the grammar. With the advent of bodies of text (corpora) that have had their structures hand-annotated, it is now possible to generalize the formal grammar to include accurate probabilities. Furthermore, the probabilistic relationship among a sequence of words can be directly derived and modeled from the corpora with the so-called stochastic language models such as n-gram, avoiding the need to create broad-coverage formal grammars.

An n-gram model is based in the hypothesis that the probability of having a word in a sentence does not depend more on all the words of the sentence than the n previous words.

$$P(M)=\prod_{k=1}^{K} P(W_k/W_{k-1},W_{k-2},...,W_1) \cong \prod_{k=1}^{K} P(W_k/W_{k-1},W_{k-2},...,W_{k-n})$$

0.4 Language Model Equations

One example with trigram could be:

P(All that glitters is not gold) = P(All|-,-)P(that|All,-)P(glitters|that,All)x xP(is|glitters,that)P(not|is,glitters)P(gold|not,is)

1.1 Tools and resources used to build the ASR

The SRI Language Modeling Toolkit (SRILM)

Statistical language modeling is the science (and often art) of building models that estimate the prior probabilities of word strings [3].

The Spanish LM (Language Models) have been built with the SRI Language Modeling Toolkit (SRILM), which is a collection of C++ libraries, executable programs and helper scripts designed to allow both production of and experimentation with statistical language models for speech recognition. As pointed out in [3], compared to existing LM (Language Models) tools, SRILM offers a programming interface and an extendible set of LM classes, several non-standard LM types, and a more comprehensive functionality that goes beyond language modeling to include tagging, N-best rescoring, and other applications.

Since SRILM can create a LM (Language Model) either by reading counts from a file or by scanning text, the first thing to do is to clean some training text with useful sentences, then generate the n-gram counts and estimate n-gram language models with the program ngram-count.

A standard LM would be created by:

ngram-count –text TRAINDATA –order N[1] –lm LM

It is important to clean the text before using ngram-count because SRILM by itself performs no text conditioning and treats everything between white spaces as a word.

Once the n-gram language models were generated, and in order to make them compatibles with Sphinx3, which was the decoder that I used for the Spanish speech recognition, two scripts included in SRILM were used: add-dummy-bows and sort-lm.

- add-dummy-bows: this script adds the ‘missing’ back-off weights (in fact, when these weights equal to 0, ngram-count does not print them)
- sort-lm: this program sorts n-grams in lexical order

[1] There are other parameters that can be changed; the default of this program creates a trigram with Good-Turing discounting and Katz back-off for smoothing.

And the last step to create a LM compatible with Sphinx3 is to create the binary with Sphinx3_lm_convert (included in the Sphinx3 package).

The CMU Sphinx Group Open Source Speech Recognition Engines

CMU Sphinx, also known as just Sphinx, is a group of speech recognition decoders (Sphinx 2-4) and an acoustic model trainer (SphinxTrain) developed at Carnegie Mellon University.

The speech recognition decoder

Sphinx, developed by Kai-Fu Lee [4], was the first continuous-speech, speaker-independent recognition system to make use of Hidden Markov acoustic Models and an n-gram statistical language model. Sphinx was significant in that it demonstrated the feasibility of continuous-speech, speaker-independent large-vocabulary recognition.

Sphinx3[2]

Previous versions of this speech recognition system used a semi-continuous representation for acoustic modeling (for example, a single set of Gaussians is used for all models, with individual models represented as a weight vector over these Gaussians). Sphinx3 adopted the prevalent continuous HMM (Hidden Markov Models) representation and has been used primarily for high-accuracy, non-real-time recognition. Recent developments (in algorithms and in hardware) have made Sphinx3 "near" real-time. Sphinx3 is under active development and, in conjunction with SphinxTrain, provides access to a number of modern modeling techniques, such as LDA/MLLT, MLLR and VTLN, which improve recognition accuracy.

Sclite

Sclite is a tool for scoring and evaluating the output of an ASR (Automatic Speech Recognition) system. Sclite belongs to the NIST SCTK Scoring Toolkit. The program compares the hypothesized output of the ASR to the reference text. After comparing reference to hypothesis, statistics are collected during the scoring process and a variety of reports can be produced to summarize the performance of the recognition system [5].

For this project three different kinds of reports have been done to analyze the performance of the ASR.

[2] Appendix III: Sphinx tutorial

The first one has a summary of the substitutions, insertions and deletions that the ASR has done during the recognition:

- Substitution: an incorrect word was substituted for the correct word.
- Deletion: a correct word was omitted in the recognized sentence.
- Insertion: an extra word was added in the recognized sentence.

The second one has a table with the mean, variance and sum of the Word Error Rate and the Sentence Error Rate.

$$WER = \frac{substitutions + deletions + insertions}{number\_of\_words\_in\_the\_correct\_sentence} \bullet 100\%$$

0.5 Word Error Rate

And the last one was a report with all the hypothesis sentences and reference sentences and with the errors highlighted.

The complete manual of how to use SCLITE performed by the University of Berkeley is in Appendix V.

Corpus LDC2006S37 from the Linguistic Data Consortium

West Point Heroico Spanish Speech is a database of digital recordings of spoken Spanish. It was designed and collected by staff and faculty of the Department of Foreign Languages (DFL) and the Center for Technology Enhanced Language Learning (CTELL) to develop acoustic models for speech recognition systems. Additionally, parts of this corpus were designed to model question/answer dialogues for use in domain-specific speech-to-speech translation systems. The corpus consists of two subcorpora, one collected in September 2001 at El Herico Colegio Militar (HEROICO), the Mexican Military Academy in Mexico City, and the other at USMA (United States Military Academy) at different times since 1997. The USMA subcorpus includes data from non-native speakers and data collected through a throat microphone [6].

The data from this corpus was collected using several different microphones and a sampling rate of 22,050 Hz with a pcm format. A total of 111,515 words are recorded in this corpus with their equivalent transcripts. The HEROICO data was recorded from free-response answers to 143 questions and from read speech of 724 distinct sentences.

The USMA data was obtained from native and non-native participants who each recorded 205 read sentences.

1.2 Spanish Language Models

The aim of this section is to explain how to build a Language Model in Spanish, specifying the text sources that have been used, giving a summary of the results and making some extractions from the final Language Model.

Since ASR (Automatic Speech Recognition) is designed to work in a medical domain, the main source for this LM (Language Model) was the Medline encyclopedia [7]. From this encyclopedia, 3,734 articles, for a total of 1,433,506 words, were used. To get all this text from the online encyclopedia a Perl script was utilized[3]. This script opened every article from the encyclopedia's website, then made a copy to a text file and cleaned it from html code so that only the text was left. Once all the articles were in different text files in a folder, they were merged into one unified article, cleaned of the symbols that were not necessary for this purpose (punctuation, etc.) and mapped in the same way as the acoustic models.

Although it is a medical domain ASR (Automatic Speech Recognition), it is also an ASR that has to work in a conversational environment, and since the Medline encyclopedia only has descriptive articles some conversational text had to be added to make the Language Model work fluently in the environment where it would be used. In order to get the conversational text the same method as employed with the Medline encyclopedia was used, but with web pages that are used to teach conversational Spanish [8], [9] & [10] containing many common conversational sentences. The sentences from the transcripts in the LDC2006S37 were also used; these sentences are simulations of conversations in Spanish. The total number of words added with this system is 114,644; with the words from Medline encyclopedia this is a total of 1,548,150 words.

The next step is to give as an input the text cleaned and unified in one file to the SRILM that generates as an output the Language Model with all the probabilities and decomposition in n-grams of the input words. For this Language Model, only 1-grams, 2-grams and 3-grams were needed, since it was for use in an ASR that needed to work real-time.

[3] Appendix II: Script Manuals. Parallel text

```
\1-grams:

-1.783162       A               -3.991723
-4.630289       ABAJO           -0.961434
-4.204341       ABASTECIDOS     -0.9515075
-3.676076       ABIERTAS        -1.387051
 :
 :
\2-grams:
-1.917061       A AFEITAR       0
-3.221762       A AFIRMAR       0
-2.963434       A ALGUIEN       0
-1.840673       A ATAHUALPA     -1.271719
 :
 :
\3-grams:

-0.60206        A ATAHUALPA QUE
-0.60206        A ATAHUALPA SE
-0.01579427     A BUENOS AIRES
-0.49485        A CABO FRANCISCO
```

0.6 Example of Language Model

Figure 2.7 shows a summary of the n-grams from the definitive Spanish Language model.

n-grams	Words
1-grams	30,464
2-grams	282,878
3-grams	143,193

0.7 Summary of n-grams in the Spanish LM

30k 1-grams may seem large for a Language Model but as it is mentioned before, this LM is made for a medical environment and in this kind of environment there are many technical words; therefore, the encyclopedia used many medical terms for every disease, and all those technical names were necessary for the objective of this work. Furthermore, Spanish has many verb conjugations, and each tense of each person is a new word that computes in the language model. (For example, the first person singular of any verb run is different from the second person or the third and also different from the plural, so there are more or less six combinations for each tense, and each verb has 18 different tenses).

1.3 Spanish Acoustic Models V1

Lexicon V1

The lexicon development process consists of defining a phonetic set and generating the word pronunciations list (dictionary) for training acoustic and language models[4].

Label	Phoneme	Letters	Example	Phonetic Transcription
A	**/a/**	A	**A**MO	**A** M O
B	**/B/**	B, V, W	**B**IEN	**B** I E N
CH	**/č/**	CH, X	**CH**INO	**CH** I N O
D	**/D/**	D	**D**E**D**O	**D** E **D** O
E	**/e/**	E	P**E**RA	P **E** R A
F	**/f/**	F	**F**OCA	**F** O K A
G	**/G/**	G, W, H	**H**UESO	**G** U E S O
I	**/i/**	I	L**I**SO	L **I** S O
J	**/x/**	G, J	**J**AMÓN	**J** A M O N
K	**/k/**	C, K, Q	**Q**UESO	**K** E S O
L	**/l/**	L	**L**AGO	**L** A G O
M	**/m/**	M	**M**A**M**Á	**M** A **M** A
N	**/n/**	N	E**N**	E **N**
NY	**/ñ/**	Ñ	NI**Ñ**O	N I **NY** O
O	**/o/**	O	**O**J**O**	**O** J **O**
P	**/p/**	P	**P**AVO	**P** A B O
R	**/r/**	R	CA**R**O	K A **R** O
RR	**/R/**	R	**R**ATÓN	**RR** A T O N

These words are mapped as "NINYO"

[4] Appendix II: Script Manuals. Dictionaries

S	/s/	S, (C, Z)[5]	CA**S**A	K A **S** A
T	/t/	T	**T**OMA	**T** O M A
U	/u/	U, W	P**U**NTO	P **U** N T O
Y	/ʎ/	LL, Y	RA**Y**O	RR A **Y** O
Z	/θ/	C, Z	CO**C**ER	C O **Z** E R

0.8 Lexicon V1

The approach for modeling Spanish phonetic sounds in the CMU Sphinx3 speech recognition system consisted of an adapted version from the phonetic set introduced by Antonio Quilis in “Fonética Acústica De La Lengua Española” [11] & [12], which resulted in the 23 phonemes listed in the figure 2.8. The adaptation consisted of discovering which phonemes were too similar for the human ear and merging them and adding letters in order to adapt to the Mexican sounds. (In Mexican Spanish the main phonetic difference is that the sound /θ/ is merged with the sound /s/).

The vocabulary size is approximately 30,000 words, which is based on a word list created from the Language Model text explained previously. The automatic generation of pronunciations was performed using a simple list of rules. The rules determine the mapping of clusters of letters into phonemes.

In the list of rules there are 5 kinds of rules:

- Letters that are treated differently because of their position in the word (for example, when the combination G+E was found it was changed to J+E, because if it was found, then G+U+E had to be changed to G+E); in this group are almost all the exceptions found in Spanish.
- Three letter groups inside words which are transcribed by two phonemes or by three phonemes, so that the combination of all those letters is needed to determine them (for example, H+U+E it has to be converted to the phonemes G+U+E or Q+U+I converted to K+I).
- Two letter combinations that have a specific phonetic transcription (for example, C+A is transcribed by K+A).

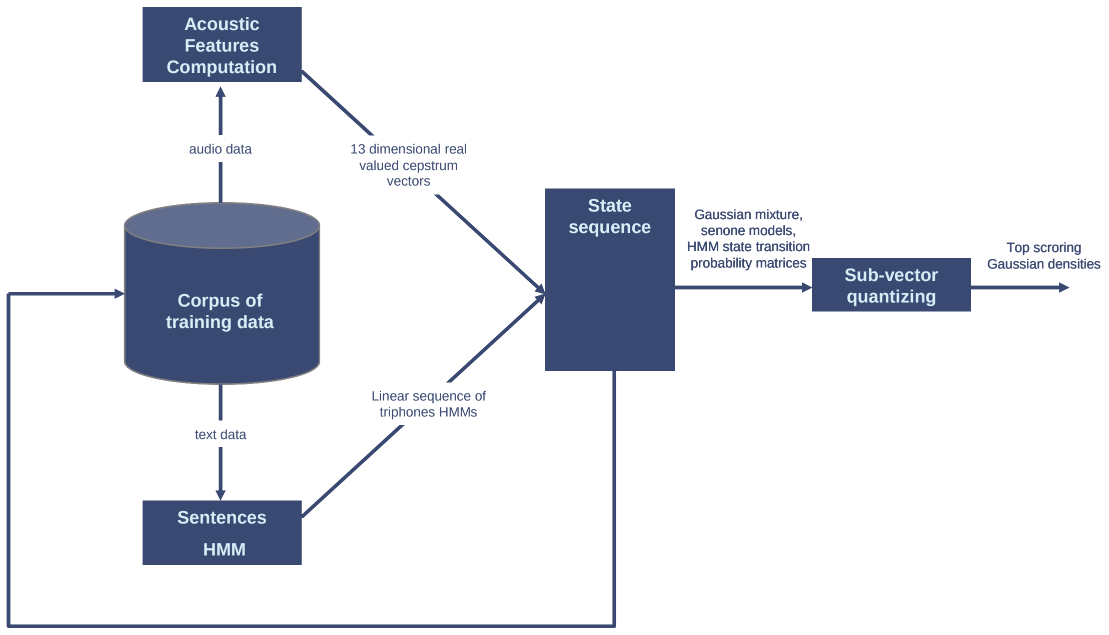
[5] Mexican Spanish

- One letter group: this group contains those phonemes that do not have the same phonetic transcription as the original letter (for example, the letter V is translated into the new phonetic language as B).
- Finally, there is a special group for the Mexican differences and foreign language inheritance letters. In this group, there is a duplication of the word that has the special pronunciation and two transcriptions were made of the differences (for example, the word CENAR is Castilian Spanish and is transcribed in phonemes as Z E N A RR and in Mexican Spanish as S E N A RR).

Acoustic Models V1

For training acoustic models it is necessary to have a set of feature files computed from the audio training data, one each for every recording in the training corpus. Each recording is transformed into a sequence of feature vectors consisting of the Mel-Frequency Cepstral Coefficients (MFCC). The training of acoustic models is based on utterances without noise.

The training process (see figure 2.9) consists of the following steps: Obtain a corpus of training data and for each utterance, convert the audio data to a stream of feature vectors; convert the text into a sequence of linear triphones HMM (Hidden Markov Models) using the pronunciation lexicon; and find the best state sequence or state alignment through the sentence HMM (Hidden Markov Models) for the corresponding feature vector sequence. For each senone, gather all the frames in the training corpus that mapped to that senone in the above step and build a suitable statistical model for the corresponding collection of feature vectors. The circularity in this training process is resolved using the iterative Baum-Welch or forward-backward training algorithm. Due to the fact that continuous density acoustic models are computationally expensive, a model is built by sub-vector quantizing the acoustic model densities.


Acoustic Features Computation
audio data
13 dimensional real valued cepstrum vectors
Corpus of training data
text data
Sentences HMM
Linear sequence of triphones HMMs
State sequence
Gaussian mixture, senone models, HMM state transition probability matrices
Sub-vector quantizing
Top scroring Gaussian densities

0.9 Acoustic Models layout

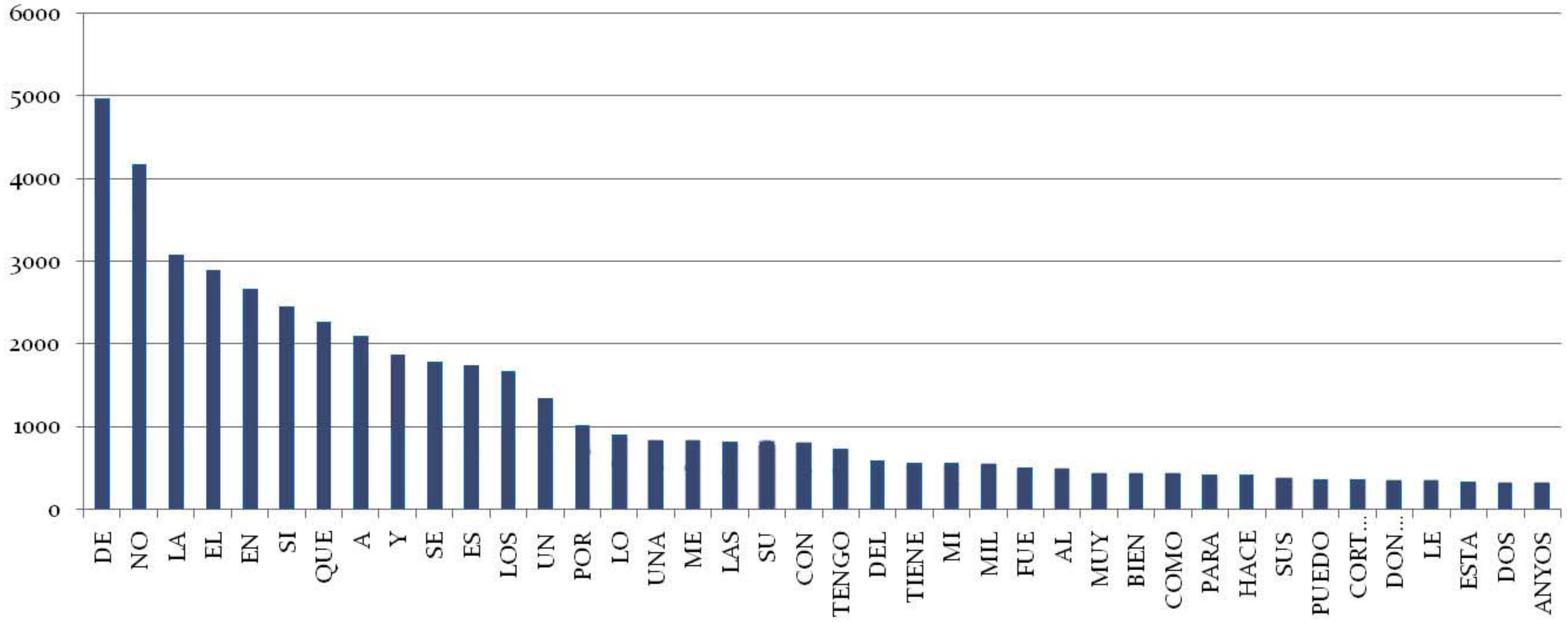
Training corpus statistics V1

The corpus used for the training has been cleaned and mapped in the same way as in the dictionary (for example, changing the letter “ñ” by “NY” or the accented vowels by the same vowels but without the accent). To train the acoustic models a sample of 111,515 words of the LDC2006S37 was used. Figure 2.11 shows that the 20 more common words (with a representation of 34% of the corpus) are words of less than 3 letters and cover only 13 phonemes (which symbolize 62% of the 21 phonemes described in this version).

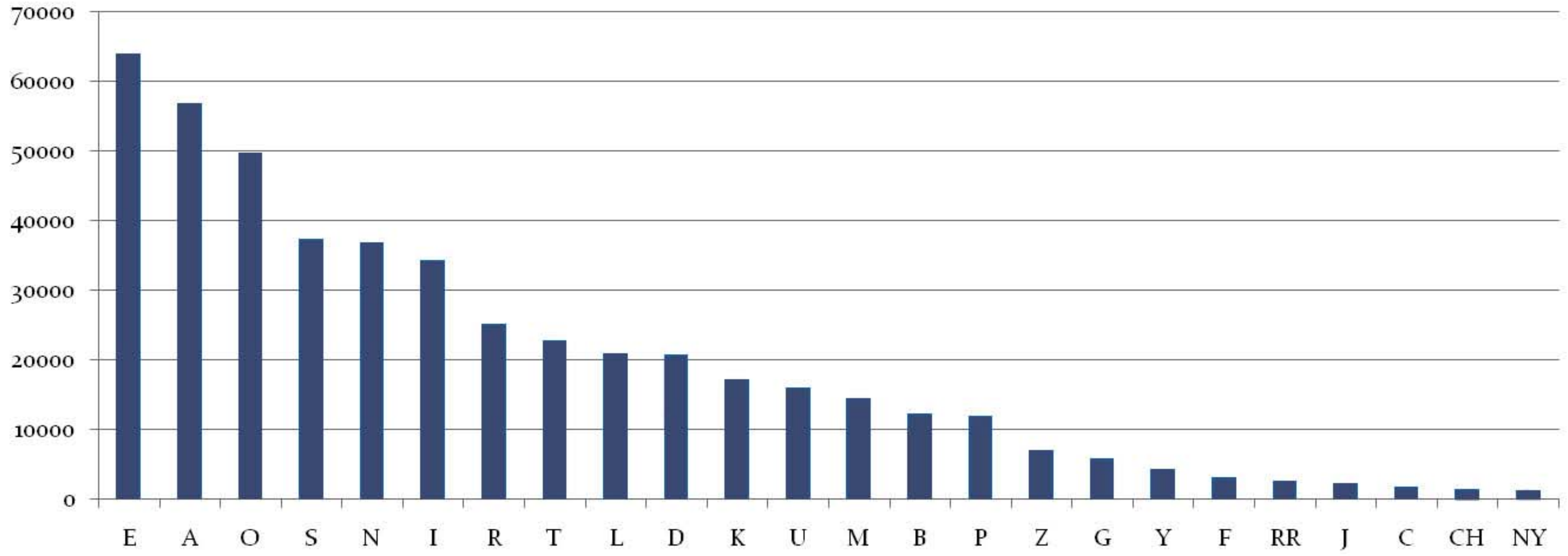
These 13 phonemes represent slightly more than 80% of the corpus (figure 2.10). Since these are also the most used phonemes in Spanish, it is advisable that those phonemes are well estimated and trained. However, it would also be desirable that some phonemes that are not as common were well trained because if they do not have enough data to train them, the errors might be concentrated in those phonemes.

A	N
D	O
E	P
I	RR
K	S
L	U
M	

0.10 Most common phonemes with the V1 mapping in LDC2006S37



0.11 Word histogram of corpus V1



0.12 Phonemes histogram of corpus V1

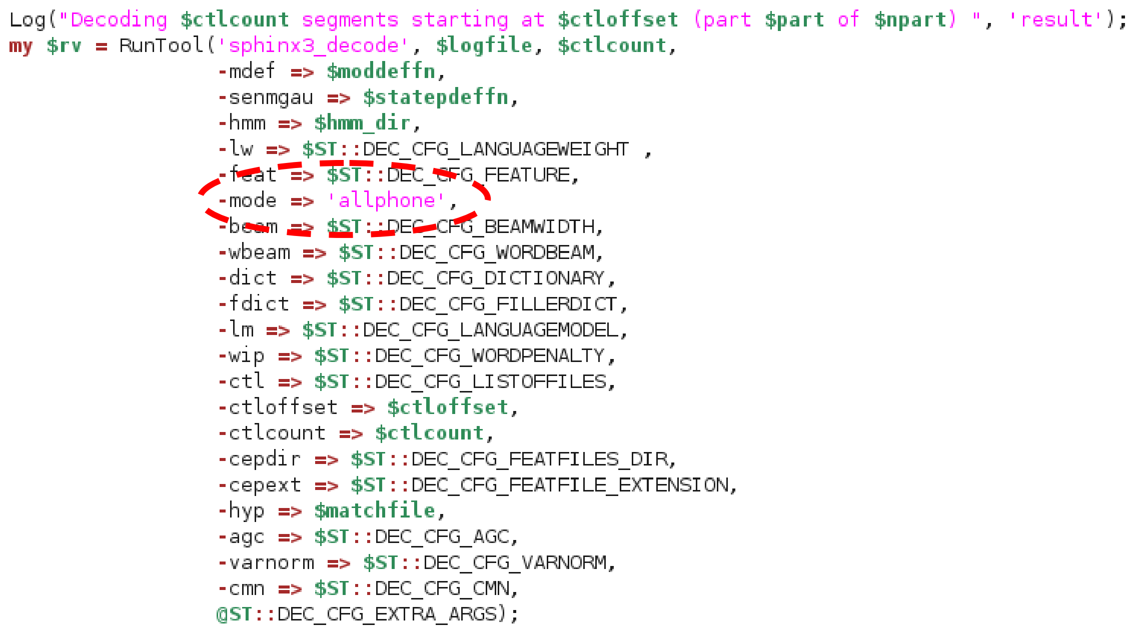
Results V1

After performing the training with SphinxTrain, some test data is prepared to make an evaluation of the system.

A sample of 6,110 words, which have not been used for the training data, is prepared with the same mapping used in the dictionary.

Once the test transcripts are the same format as the output of the ASR, the audio that belongs to the test transcripts is decoded with Sphinx3 and then with Sclite, and the word recognition performance is evaluated (figure 2.14).

Since the ASR made is phoneme based, it is also desirable to know the phoneme accuracy. There is an option in the script s3decode.pl from the Sphinx3 decoder (figure 2.13) to give the output as a sequence of phonemes, and with the force align tool provided by SphinxTrain (combined with the script create_phone_transcript_forced.pl[6]), a phoneme transcript is made to evaluate the phoneme accuracy (figure 2.14).



```
Log("Decoding $ctlcount segments starting at $ctloffset (part $part of $npart) ", 'result');
my $rv = RunTool('sphinx3_decode', $logfile, $ctlcount,
                 -mdef => $moddeffn,
                 -senmgau => $statepdeffn,
                 -hmm => $hmm_dir,
                 -lw => $ST::DEC_CFG_LANGUAGEWEIGHT ,
                 -feat => $ST::DEC_CFG_FEATURE,
                 -mode => 'allphone',
                 -beam => $ST::DEC_CFG_BEAMWIDTH,
                 -wbeam => $ST::DEC_CFG_WORDBEAM,
                 -dict => $ST::DEC_CFG_DICTIONARY,
                 -fdict => $ST::DEC_CFG_FILLERDICT,
                 -lm => $ST::DEC_CFG_LANGUAGEMODEL,
                 -wip => $ST::DEC_CFG_WORDPENALTY,
                 -ctl => $ST::DEC_CFG_LISTOFFILES,
                 -ctloffset => $ctloffset,
                 -ctlcount => $ctlcount,
                 -cepdir => $ST::DEC_CFG_FEATFILES_DIR,
                 -cepext => $ST::DEC_CFG_FEATFILE_EXTENSION,
                 -hyp => $matchfile,
                 -agc => $ST::DEC_CFG_AGC,
                 -varnorm => $ST::DEC_CFG_VARNORM,
                 -cmn => $ST::DEC_CFG_CMN,
                 @ST::DEC_CFG_EXTRA_ARGS);
```

0.13 script s3decode.pl

[6] Appendix II: Script Manuals. Dictionaries

Recognition Performance	Words	Phonemes
Number	6,110	26,833
Substitutions	807 (13.5%)	2,920 (11.2%)
Deletions	633 (10.6%)	4,558 (17.6%)
Insertions	119 (2%)	1,610 (6%)
Accuracy	**4,551 (74%)**	**18,483 (64.8%)**

0.14 Results summary V1

1.4 Spanish Acoustic Models V2.1

Lexicon V2.1

Once the first version of the Spanish ASR (Automatic Speech Recognition) system is tested, it is observed that there are many decoding errors coming from the vowels and the phonemes S and Z. Since almost all the sentences that are recorded in the corpus LDC2006S37 are spoken in Mexican Spanish, the phoneme Z (a Castilian phoneme) makes no sense, so for this new version it is deleted.

Another important change in this version is that there are five new phonemes [13] added to symbolize the Spanish accents[7].

Label	Phoneme	Letters	Example	Phonetic Transcription
A	**/a/**	A	**A**MO	**A** M O
AA	**/a”/**	Á	MÁS	M **AA** S
B	**/B/**	B, V, W	**B**IEN	**B** I E N
CH	**/č/**	CH, X	**CH**INO	**CH** I N O
D	**/D/**	D	**D**E**D**O	**D** E **D** O
E	**/e/**	E	P**E**RA	P **E** R A
EA	**/e”/**	É	CAFÉ	K A F **EA**
F	**/f/**	F	**F**OCA	**F** O K A
G	**/G/**	G, W, H	**H**UESO	**G** U E S O
I	**/i/**	I	L**I**SO	L **I** S O
IA	**/i”/**	Í	TENÍA	T E N **IA** A
J	**/x/**	G, J	**J**AMÓN	**J** A M OA N
K	**/k/**	C, K, Q	QUESO	**K** E S O
L	**/l/**	L	LAGO	**L** A G O

[7] Appendix II: Script Manuals. Dictionaries

M	**/m/**	M	**M**A**M**Á	**M** A **M** AA
N	**/n/**	N	E**N**	E **N**
NY	**/ñ/**	Ñ	NIÑO	N I **NY** O
O	**/o/**	O	**OJO**	**O** J **O**
OA	**/o''/**	Ó	CAMI**Ó**N	K A M I **OA** N
P	**/p/**	P	**P**AVO	**P** A B O
R	**/r/**	R	CA**R**O	K A **R** O
RR	**/R/**	R	**R**ATÓN	**RR** A T OA N
S	**/s/**	S, (C, Z)*	CA**S**A	K A **S** A
T	**/t/**	T	**T**OMA	**T** O M A
U	**/u/**	U, W	P**U**NTO	P **U** N T O
UA	**/u''/**	Ú	CANC**Ú**N	K A N K **UA** N
Y	**/ʎ/**	LL, Y	RA**Y**O	RR A **Y** O

These words are mapped as "NINYO"

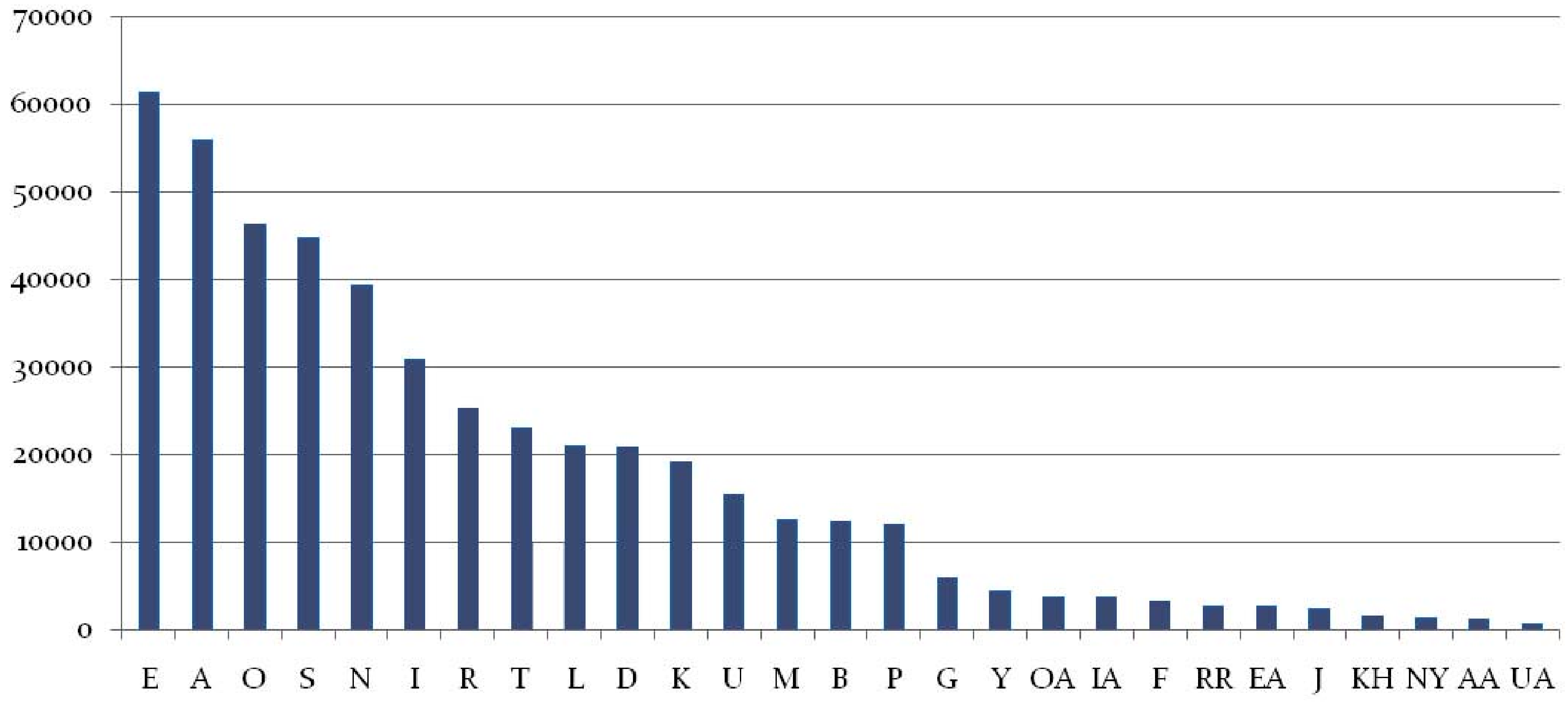
0.15 Lexicon V2.1

Acoustic Models V2.1

The acoustic models of this version are trained in the same way as the version before, but now we use the new dictionary and the new phone list with the five new phonemes and without the phoneme Z.

Corpus statistics V2.1

Figure 2.16 shows that the vowel phonemes (without accent) have less representation (the graphic becomes more flat): the reduction of the vowels without accent in the corpus is approximately 6%. It also shows that the representation of the phoneme S is augmented by approximately 21%. Furthermore, the tail of the graphic shows that the accentuated vowels are not well-represented in the corpus (two of the less represented phonemes, AA and UA, belong to this new accentuated phonemes group).



0.16 Phonemes histogram of corpus V2.1

Results V2.1

This version is evaluated following the same steps as for the first version: preparing the word and phoneme transcripts, decoding with Sphinx3, and evaluating with Sclite.

Recognition Performance	Words	Phonemes
Number	6,107	26,945
Substitutions	864 (14.4%)	2,874 (11.0%)
Deletions	719 (11.9%)	4,644 (17.8%)
Insertions	116 (1.9%)	872 (3.5%)
Accuracy	**4,413 (71.7%)**	**19,362 (67.6%)**

0.17 Results summary V2.1

Figure 2.17 shows that the word accuracy declines slightly but the phoneme accuracy improves almost 3%. With this result, the next version tries to merge the results in Word Error Rate of Version 1 and the Phoneme Error Rate of Version 2.1.

1.5 Spanish Acoustic Models V2.2

Lexicon V2.2

In order to improve the Word Error Rate of the previous section of the experiment, a new mapping for the words with accents is made. The combination “vowel + WW” is used in the dictionary and in the transcripts to let SphinxTrain differentiate between the words with accent and the words without[8].

Label	Phoneme	Letters	Example	Phonetic Transcription
A	**/a/**	A	**A**MO	**A** M O
AA	**/a’’/**	Á	MÁS	M **AA** S
B	**/B/**	B, V, W	**B**IEN	**B** I E N
CH	**/č/**	CH, X	**CH**INO	**CH** I N O
D	**/D/**	D	**D**E**D**O	**D** E **D** O
E	**/e/**	E	P**E**RA	P **E** R A
EA	**/e’’/**	É	CAFÉ	K A F **EA**
F	**/f/**	F	**F**OCA	**F** O K A
G	**/G/**	G, W, H	**H**UESO	**G** U E S O
I	**/i/**	I	L**I**SO	L **I** S O
IA	**/i’’/**	Í	TENÍA	T E N **IA** A
J	**/x/**	G, J	**J**AMÓN	**J** A M OA N
K	**/k/**	C, K, Q	QUESO	**K** E S O
L	**/l/**	L	LAGO	**L** A G O
M	**/m/**	M	**M**A**M**Á	**M** A **M** AA
N	**/n/**	N	E**N**	E **N**
NY	**/ñ/**	Ñ	NIÑO	N I **NY** O

These words are mapped as “**MAWWS**”

These words are mapped as “CAF**EWW**”

These words are mapped as “**TENIWWA**”

These words are mapped as “NINYO”

[8] Appendix II: Script Manuals. Dictionaries

O	**/o/**	O	**OJO**	**O** J **O**
OA	**/o''/**	Ó	CAMI**O**N	K A M I **OA** N
P	**/p/**	P	**P**AVO	**P** A B O
R	**/r/**	R	CA**R**O	K A **R** O
RR	**/R/**	R	**R**ATÓN	**RR** A T OA N
S	**/s/**	S, (C, Z)*	CA**S**A	K A **S** A
T	**/t/**	T	**T**OMA	**T** O M A
U	**/u/**	U, W	P**U**NTO	P **U** N T O
UA	**/u''/**	Ú	CANC**Ú**N	**K** A N K **U** N
Y	**/ʎ/**	LL, Y	RA**Y**O	RR A **Y** O

These words are mapped as "CAMI**OWW**N"

These words are mapped as "CANC**UWW**N"

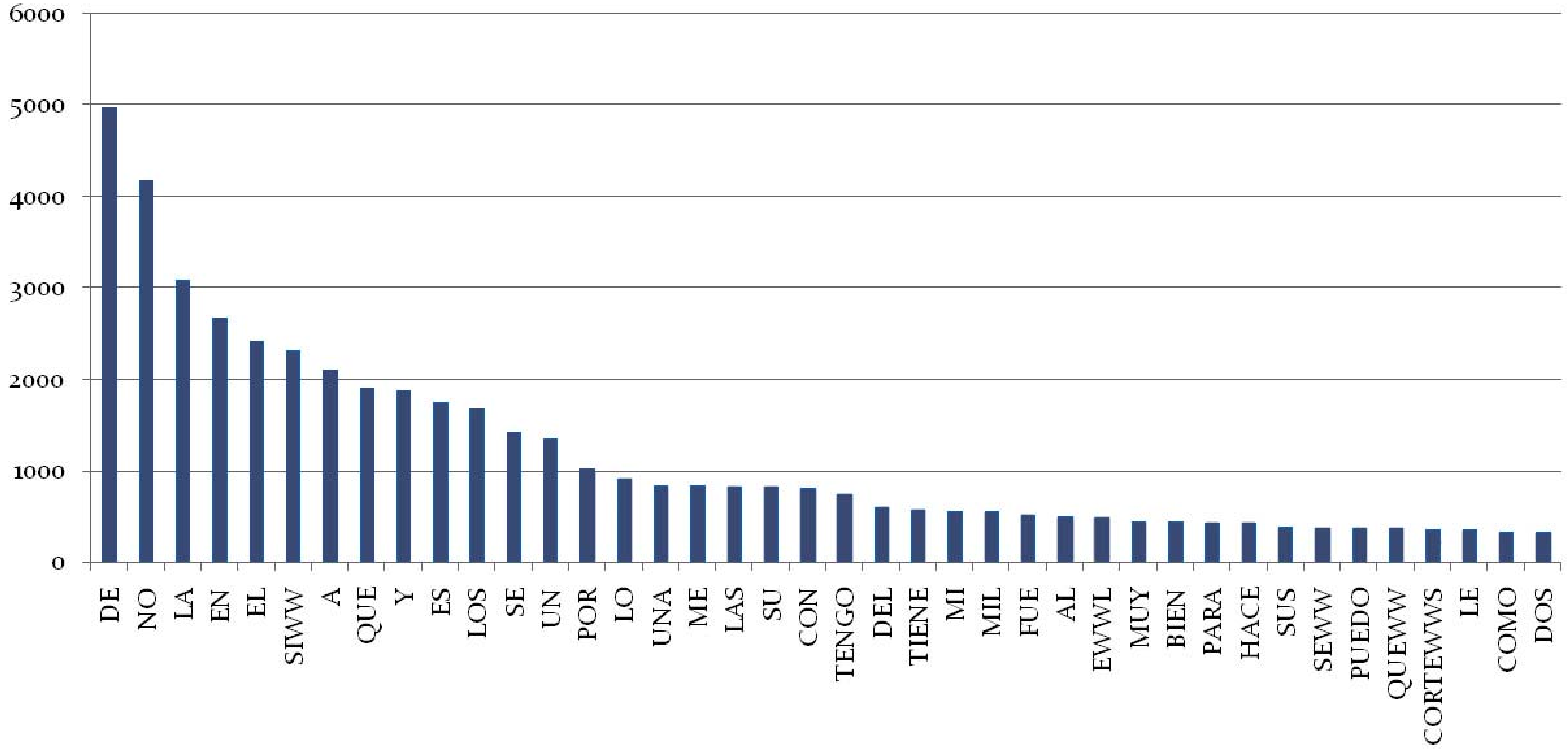
0.18 Lexicon V2.2

Acoustic Models V2.2

The acoustic model V2.2 is also trained with SphinxTrain but with the new mapping applied to the transcripts, the new dictionary and using the same phone list as in the version V2.1.

Corpus statistics V2.2

As mentioned before, the word mapping in the version 2.2 changes and this causes the histogram of the word transcripts of the corpus LDC2006S37 to become more flat (figure 2.18). Moreover, in the 40 most used words of the transcripts there are three pairs of words (EL - EWWL, SI – SIWW and QUE – QUEWW) that were previously written in the same way and are now differentiated.



0.19 Word histogram of corpus V2.2

Results V2.2

To evaluate this version, the same phone transcription as in version 2.1 is used, but it is necessary to make a new word transcription to make it match with the mapping of the output, or to make an unmapped output of the decoder. The solution was to simply change the transcription.

Recognition Performance	Words	Phonemes
Number	6,060	26,995
Substitutions	750 (12.5%)	3,010 (11.5%)
Deletions	588 (9.8%)	4,443 (17%)
Insertions	75 (1.3%)	873 (3.5%)
Accuracy	**4,653 (76.4%)**	**19,455 (67.8%)**

0.20 Results summary V2.2

Figure 2.20 shows that this version of the ASR not only improves in Word Error Rate but also improves slightly (0.2%) in Phoneme Error Rate. This is due to the new mapping that has been made for the accented words.

1.6 Spanish Acoustic Models V3

Lexicon V3

This version was an experiment to see if using the phoneme describing an accentuated word to also describe the vowel of the stressed syllabi [14] would be effective, since the accents just add an extra stress on those vowels[9].

Label	Phoneme	Letters	Example	Phonetic Transcription
A	**/a/**	A	**A**MO	**A** M O
AA	**/a''/, /a'/**	Á, A	M**Á**S, CASA	M **AA** S, K **AA** S A
B	**/B/**	B, V, W	**B**IEN	**B** I E N
CH	**/č/**	CH, X	**CH**INO	**CH** I N O
D	**/D/**	D	**D**E**D**O	**D** E **D** O
E	**/e/**	E	P**E**RA	P **E** R A
EA	**/e''/, /e'/**	É, E	CAF**É**, DEDO	C A F **EA**, D **EA** D O
F	**/f/**	F	**F**OCA	**F** O K A
G	**/G/**	G, W, H	**H**UESO	**G** U E S O
I	**/i/**	I	L**I**SO	L **I** S O
IA	**/i''/, /i'/**	Í, I	TEN**Í**A, IBA	T E N **IA** A, **IA** B A
J	**/x/**	G, J	**J**AMÓN	**J** A M O N
K	**/k/**	C, K, Q	QUESO	**K** E S O
L	**/l/**	L	**L**AGO	**L** A G O
M	**/m/**	M	**M**A**M**Á	**M** A **M** AA
N	**/n/**	N	E**N**	E **N**

These words are mapped as "MA**WWS**"

These words are mapped as "CAFE**WW**"

These words are mapped as "TEN**IWWA**"

[9] Appendix II: Script Manuals. Dictionaries

NY	**/ñ/**	Ñ	NI**Ñ**O	N I **NY** O
O	**/o/**	O	**OJO**	**O** J **O**
OA	**/o''/, /o'/**	Ó, O	CAMI**Ó**N, ROTO	K A M I **OA** N, RR **OA** T O
P	**/p/**	P	**P**AVO	**P** A B O
R	**/r/**	R	CA**R**O	K A **R** O
RR	**/R/**	R	**R**ATÓN	**RR** A T O N
S	**/s/**	S, (C, Z)*	CA**S**A	K A **S** A
T	**/t/**	T	**T**OMA	**T** O M A
U	**/u/**	U, W	P**U**NTO	P **U** N T O
UA	**/u''/, /u'/**	Ú, U	CANC**Ú**N, MUNDO	K A N K **UA** N, M **UA** N D O
Y	**/ʎ/**	LL, Y	RA**Y**O	RR A **Y** O

These words are mapped as "CAMI**OWW**N"

These words are mapped as "CANC**UWW**N"

0.21 Lexicon V3

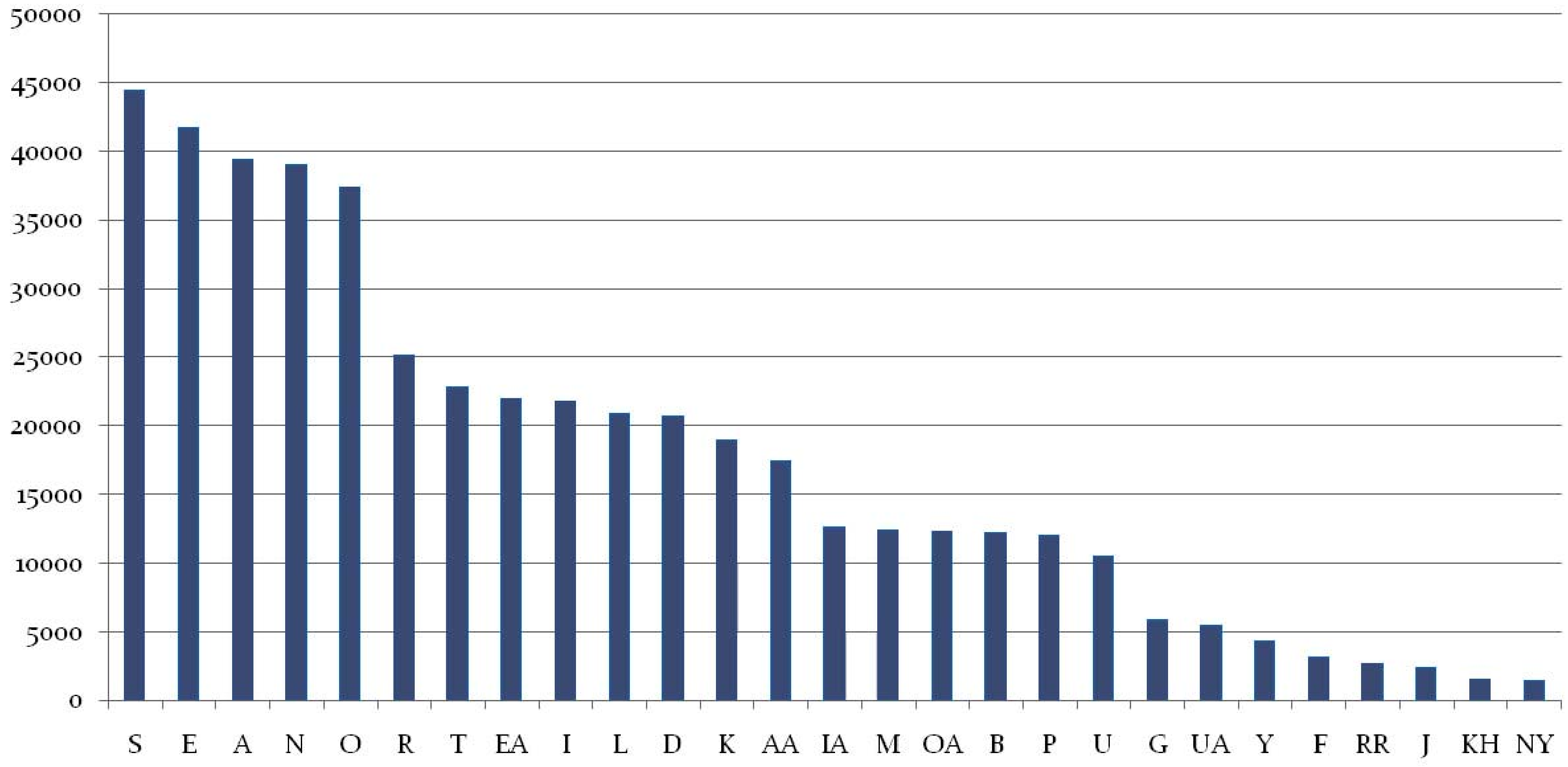
Acoustic Models V3

To train this new model, the same corpus and phoneme lists as in the previous models were used; the only changes were made in the dictionary, which now has new rules to create the phoneme transcription.

Corpus statistics V3

The distribution of phonemes in the corpus with this new version of the dictionary is much flatter (figure 2.22); the stressed vowels now have much more relevance; and the phoneme S becomes the most used in the corpus, even more than any vowel (in all the other versions was the fourth most used).

The phoneme histogram becoming flatter means that the phonemes used for training are more distributed: there are more examples of each phoneme. However, if the new distribution is not necessary or it is too difficult to differentiate between the phonemes with stress and the ones without, the system will be less efficient.



0.22 Phonemes histogram of corpus V3

Results V3

Version 3 has been evaluated in exactly the same way as version V2.2, since the outputs of the system were in the same format. The test has been done using 6,095 words that were not used for the training data.

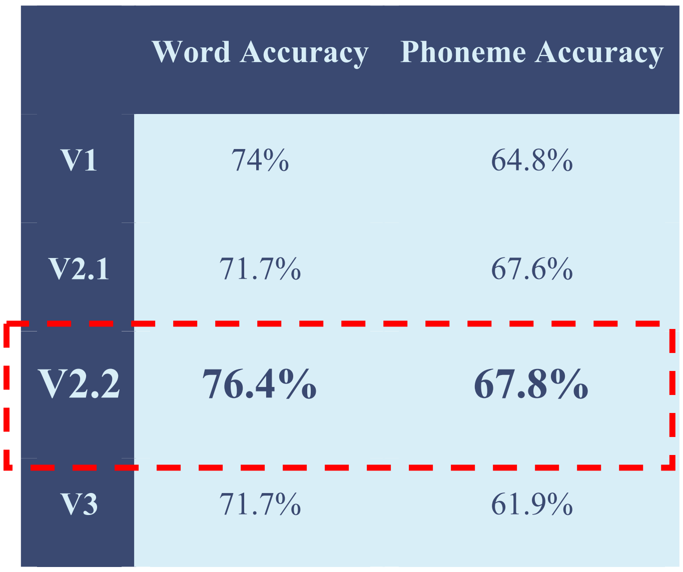
Recognition Performance	Words	Phonemes
Number	6,095	27,011
Substitutions	850 (14.2%)	4,228 (16.2%)
Deletions	740 (12.4%)	4,697 (18%)
Insertions	104 (1.7%)	987 (3.7%)
Accuracy	**4,401 (71.7%)**	**17,099 (61.9%)**

0.23 Results summary

Figure 2.23 shows that both the Word Error Rate and the Phoneme Error Rate have been increased. These increased error rates come from the difficulty in differentiating between the vowels without stress, with stress and accented: in creating a state between the accented and not accented, the difference between them becomes less and therefore harder to differentiate, and this leads to increased error rates.

1.7 Final decision of Acoustic Model

Three versions of Acoustic models have been detailed: the first was the simplest, using the main phonemes of Spanish from Spain and Mexico without taking into account the stresses that occur in the different Spanish words. The second version took into account the stressed vowels, but only in the words with extra stress (accented), and eliminated the phonemes that only occur in Spain's Spanish. The third version made use of phonemes to describe the accented vowels in all the words, giving the situation inside the word of the stressed syllabi.



	Word Accuracy	Phoneme Accuracy
V1	74%	64.8%
V2.1	71.7%	67.6%
V2.2	**76.4%**	**67.8%**
V3	71.7%	61.9%

0.24 Results summary

The accents in Spanish have information; they can change the meaning of a word, and since this ASR (Automatic Speech Recognition) is designed to make a speech-to-speech translator, the meaning of each word is needed in order to make a proper translation. For example, the word "Cómo" is translated as "How" and the word "Como" is translated as "I eat."

It would appear that the best version to use for the Spanish to English translator is Version 2.2, not only for its better accuracy in words and phonemes (figure 2.24) but also because it makes a mapping of the accents and gives an output in which the words with accents can be found.

2. Statistical Machine Translator

Statistical machine translation (SMT) is a machine translator model in which translations are generated on the basis of statistical models whose parameters are derived from the analysis of bilingual text corpora. The statistical approach contrasts with the rule-based approaches to machine translation as well as with example-based machine translation.

The first theory of statistical machine translation, including the idea of applying Claude Shannon’s information theory, was introduced by Warren Weaver in 1949 [15],. Statistical machine translation was re-introduced in the late 1980s by researchers at IBM’s Thomas J. Watson Research Center [16] with the Candide project. IBM's original approach maps individual words to words and allows for deletion and insertion of words.

More recently, various researchers have demonstrated better translation quality with the use of phrase translation. Phrase-based Machine Translators can be traced back to Franz Josef Och's alignment template model [17], which can be re-framed as a phrase translation system.

Daniel Marcu introduced a joint-probability model for phrase translation [18]. At this time, however, most competitive statistical machine translation systems use phrase translation.

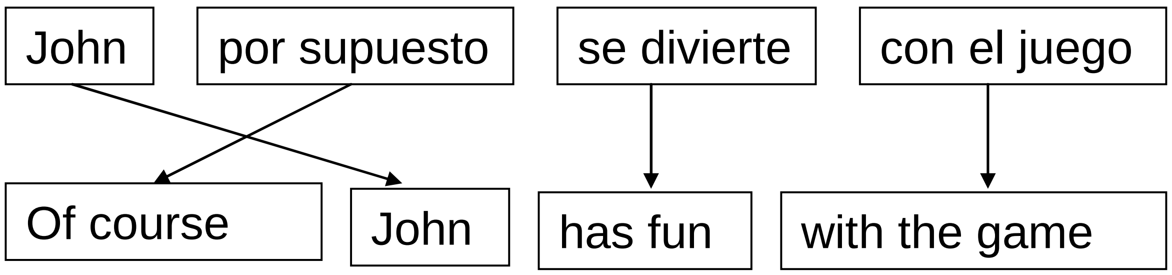
Of course, there are other ways to do machine translation. Most commercial systems use transfer rules and a rich translation lexicon. Until recently, machine translation research has been focused on knowledge-based systems that use an interlingua representation as an intermediate step between input and output.

There are also other ways to do statistical machine translation. There has been some effort toward building syntax-based models that either use real syntax trees generated by syntactic parsers or tree transfer methods motivated by syntactic reordering patterns.

2.1 Moses

Model

Figure 3.1 illustrates the process of phrase-based translation. The input is segmented into a number of sentences. Each phrase is translated into an English phrase, and English phrases in the output may be reordered.



2.1 Process of phrase-based translation

The phrase translation model is based on the noisy channel model [19]. With the Bayes rule, the translation probability for translating a foreign sentence **f** into English **e** can be formulated as:

$$\arg\max_e p(e/f) = \arg\max_e p(f/e)p(e)$$

2.2 Translation probabilities (I)

According to the model used in Moses, the best English output sentence **e** given a foreign input sentence **f** is:

$$e_{best} = \arg\max_e p(e/f) = \arg\max_e p(f/e)p_{lm}(e)\omega^{lenght(e)}$$

2.3 Translation probabilities (II)

Where $p(f/e)$ is split into:

$$p(\underline{f}_1^I / \underline{e}_1^I) = \Phi_{i=1}^I \varphi(\underline{f}_i / \underline{e}_i) d(start_i, end_i)$$

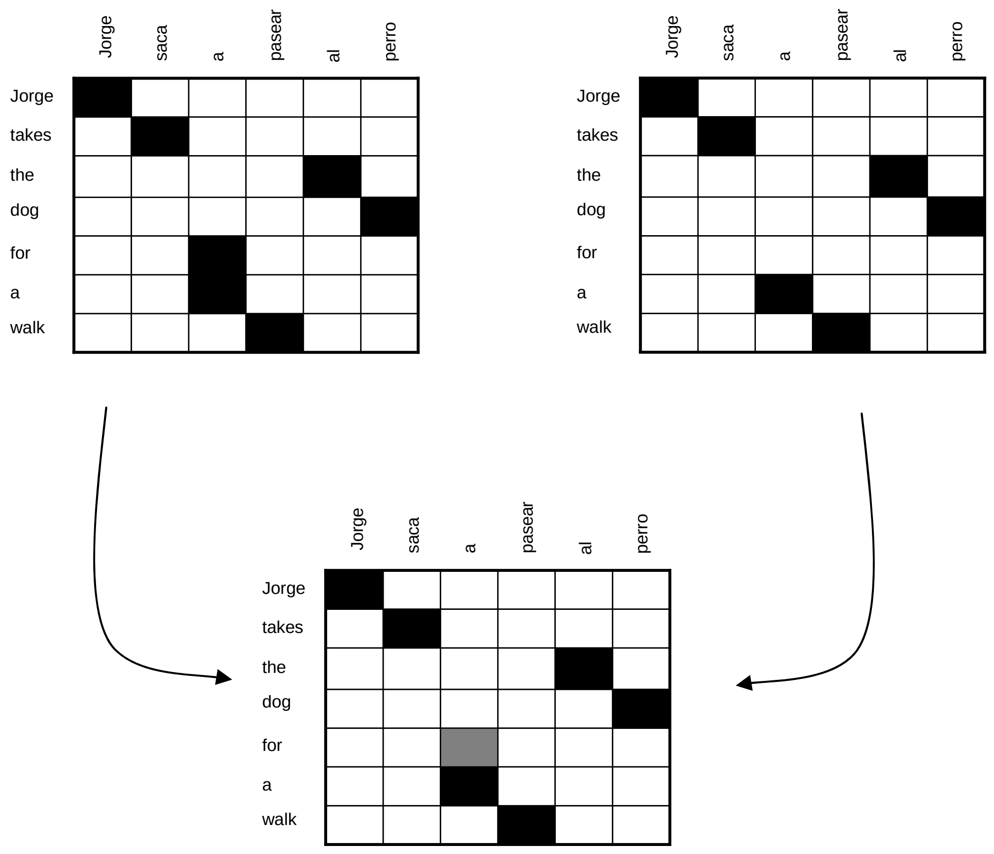
2.4 Translation probabilities (III)

Word Alignment

Most recently published methods on extracting a phrase translation table which maps foreign phrases to English phrases from a parallel corpus start with a word alignment.

At this point, the most common tool to establish a word alignment is the toolkit GIZA++. This toolkit is an implementation of the original IBM Models that started statistical machine translation research. However, these models have some serious drawbacks. Most importantly, they only allow at most one English word to be aligned with each foreign word. To resolve this, some transformations are applied.

First, the parallel corpus is aligned bidirectionally, for example, Spanish to English and English to Spanish. This generates two word alignments that have to be reconciled. Intersecting the two alignments, we get a high-precision alignment of high-confidence alignment points. And taking the union of the two alignments, we get a high-recall alignment with additional alignment points. See figure 3.5 [19] for an illustration.



2.5 Phrase translation tables

Decoder

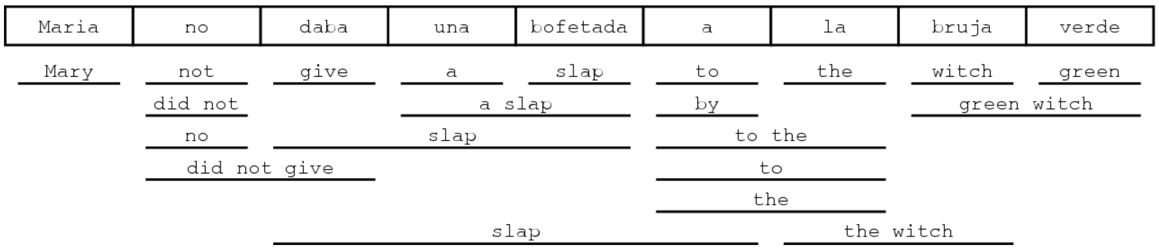
The decoder was originally developed for the phrase model proposed by Marcu and Wong [18].

The decoder implements a beam search and is roughly similar to work by Tillmann [20] and Och [21]. In fact, by reframing Och's alignment template model as a phrase translation model, the decoder is also suitable for his model, as well as other recently proposed phrase models.

The concepts of **translation options** (pruning, beam search and future probability estimates) and **n-best list generation** are defined below.

Translation options

Given an input string of words, a number of phrase translations could be applied. Such applicable phrase translation is a translation option. This is illustrated in figure 3.6 [19], where a number of phrase translations for the Spanish input sentence “María no daba uma bofetada a la bruja verde” are given.



2.6 Example of phrase translation

These translation options are collected before any decoding takes place. This allows a quicker lookup than consulting the whole phrase translation table during decoding. The translation options are stored with the following information:

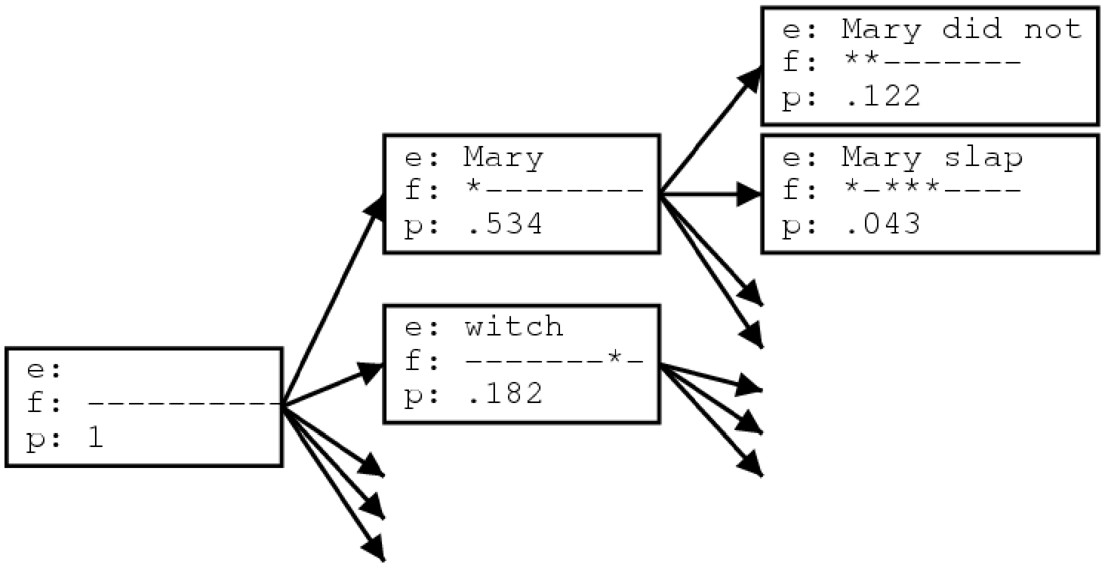
- first foreign word covered
- last foreign word covered
- English phrase translation
- phrase translation probability.

Note that only the translation options that can be applied to a given input text are necessary for decoding. Since the entire phrase translation table may be too large to fit into memory, the Moses decoder restricts itself to these translation options in order to overcome such computational concerns.

Core Algorithm

Moses' phrase-based decoder employs a beam search algorithm, similar to the one used by Jelinek [22] for speech recognition. The English output sentence is generated left to right in form of hypotheses.

This process is illustrated in figure 3.7 [19]. The search begins in an initial state where no foreign input words are translated and no English output words have been generated. New states are created by extending the English output with a phrasal translation that covers some of the foreign input words not yet translated.



2.7 Beam search

The current probability of the new state is the probability of the original state multiplied by the translation, distortion and language model costs of the added phrasal translation.

Each search hypothesis is represented by:

- a back link to the best previous state
- the foreign words covered so far
- the last two English words generated

- the end of the last foreign phrase covered
- the last added English phrase
- the probability so far
- an estimate of the future probability
- final states in the search are hypotheses that cover all foreign words. Among these, the hypothesis with the highest probability is selected as best translation.

This algorithm can be used for exhaustively searching through all possible translations.

Recombining hypothesis

Recombining hypothesis is a risk-free way to reduce the search space. Two hypotheses can be recombined if they agree in:

- the foreign words covered so far
- the last two English words generated
- the end of the last foreign phrase covered.

If there are two paths that lead to two hypotheses that agree in these properties, the decoder keeps only the hypothesis with higher probability, for example, the one with the highest probability so far. The other hypothesis cannot be part of the path to the best translation, and it can be safely discarded.

Beam Search

While the recombination of hypotheses as described above reduces the size of the search space, this is insufficient for all but the shortest sentences. Now we must estimate how many hypotheses are generated during an exhaustive search. Considering the possible values for the properties of unique hypotheses, an upper bound can be estimated for the number of states by N ~ 2nf |Ve|2, where nf is the number of foreign words, and |Ve| the size of the English vocabulary. In practice, the number of possible English words for the last two words generated is much smaller than |Ve|2. The main concern is the exponential explosion from the 2nf possible configurations of foreign words covered by a hypothesis.

In the Moses beam search the hypotheses that cover the same number of foreign words are compared and the inferior hypotheses are pruned out. If there is a three-word foreign phrase that easily translates into a common English phrase, this may carry a much higher probability than translating three words separately into uncommon English words. The search will prefer to start the sentence with the easy part and discount alternatives too early.

So the Moses measure for pruning out hypotheses in our beam search not only includes the probability so far, but also an estimate of the future probability. This future probability estimation should favour hypotheses that already covered difficult parts of the sentence and have only easy parts left and discount hypotheses that covered the easy parts first.

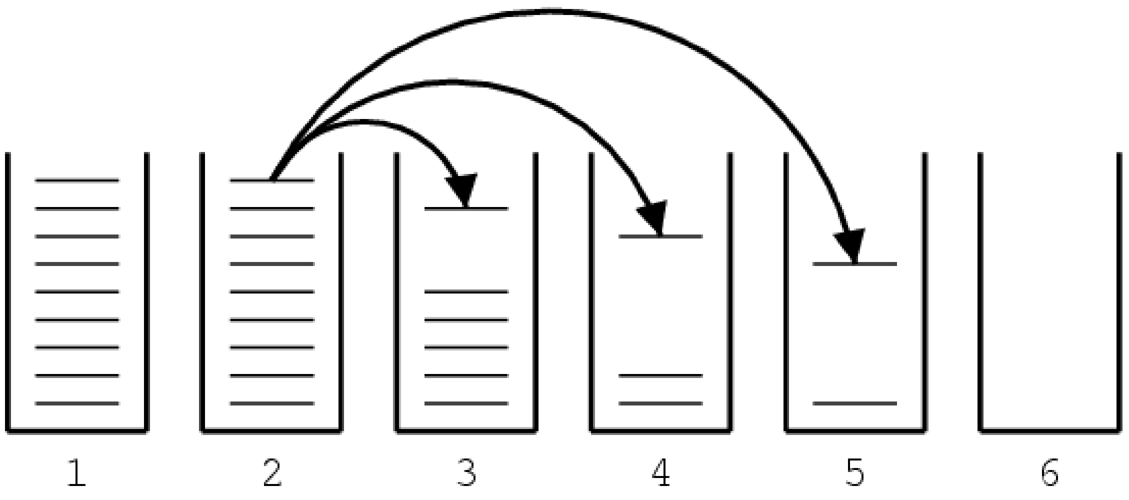
Given the probability so far and the future probability estimation, we can prune out hypotheses that fall outside the beam. The beam size can be defined by threshold and histogram pruning. A relative threshold cuts out a hypothesis with a probability less than a factor α of the best hypotheses (for example, $\alpha = 0.001$). Histogram pruning keeps a certain number n of hypotheses (for example, $n = 100$).

The figure 3.8 [18], [19] gives pseudo-code for the algorithm used for the beam search. For each number of foreign words covered, a hypothesis stack is created. The initial hypothesis is placed in the stack for hypotheses with no foreign words covered. Starting with this hypothesis, new hypotheses are generated by committing to phrasal translations that covered previously unused foreign words. Each derived hypothesis is placed in a stack based on the number of foreign words it covers.

```
initialize hypothesisStack[0 .. nf];
create initial hypothesis hyp_init;
add to stack hypothesisStack[0];
for i=0 to nf-1:
  for each hyp in hypothesisStack[i]:
    for each new_hyp that can be derived from hyp:
      nf[new_hyp] = number of foreign words covered by new_hyp;
      add new_hyp to hypothesisStack[nf[new_hyp]];
      prune hypothesisStack[nf[new_hyp]];
find best hypothesis best_hyp in hypothesisStack[nf];
output best path that leads to best_hyp;
```

2.8 Beam search pseudo-code algorithm

The Moses beam search proceeds through these hypothesis stacks, going through each hypothesis in the stack, deriving new hypotheses for this hypothesis and placing them into the appropriate stack (see figure 3.9 [19] for an illustration). After a new hypothesis is placed into a stack, the stack may have to be pruned by threshold or histogram pruning if it has become too large. In the end, the best hypothesis of the ones that cover all foreign words is the final state of the best translation. We can read off the English words of the translation by following the back links in each hypothesis.



2.9 Hypothesis stacks

Future Probability Estimation

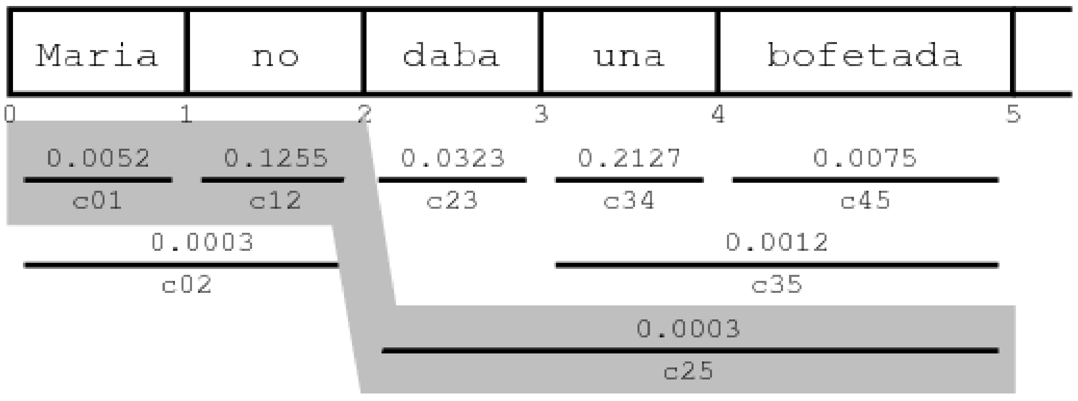
While it is possible to calculate the cheapest possible probability cost for each hypothesis, this is computationally so expensive that it would defeat the purpose of the beam search.

The future probability is tied to the foreign words that are not yet translated. In the framework of the phrase-based model, not only may single words be translated individually, but also consecutive sequences of words as a phrase.

Each such translation operation carries a translation probability, language model costs, and a distortion cost. For the future cost estimation, only the translation and the language model costs are considered. The language model probability is usually calculated by a trigram language model. However, the preceding English words for a translation operation are not known. Therefore, the decoder approximates this probability by computing the language model score for the generated English words alone. That means that if only one English word is generated, it takes its unigram probability. If two words are generated, the decoder takes the unigram probability of the first word and the bigram probability of the second word, and so on.

For a sequence of foreign words, multiple overlapping translation options exist. The way to translate the sequence of foreign words with the highest probability includes the translation options with the highest probability. The cost for a path through translation options is approximated by the product of the cost for each option.

To illustrate this concept, refer to figure 3.10 [19]. The translation options cover different consecutive foreign words and carry an estimated cost c_{ij}. The cost of the shaded path through the sequence of translation options is $c_{01}c_{12}c_{25}$ = 1.9578 * 10-7.



2.10 Path cost estimation

The path with the highest probability for a sequence of foreign words can be quickly computed with dynamic programming. Also note that if the foreign words not covered so far are two (or more) disconnected sequences of foreign words, the combined cost is simply the product of the costs for each contiguous sequence. Since there are only $\frac{n(n+1)}{2}$ contiguous sequences for n words, the future probability estimates for these sequences can be easily precomputed and cached for each input sentence. Looking up the future probabilities for a hypothesis can then be done very quickly by table lookup. This has considerable speed advantages over computing future cost on the fly.

N-Best Lists Generation

Usually, the decoder is expected to give us the best translation for a given input according to the model. But for some applications, we might also be interested in the second best translation, third best translation, and so on.

A common method in speech recognition, which has also emerged in machine translation, is to first use a machine translation system such as the Moses decoder as a base model to generate a set of candidate translations for each input sentence. Then, additional features are used to rescore these translations.

An n-best list is one way to represent multiple candidate translations. Such a set of possible translations can also be represented by word graphs [23] or forest structures over parse trees [24]. These alternative data structures allow for a more compact representation of a much larger set of candidates. However, it is much harder to detect and score global properties over such data structures.

Additional Arcs in the Search Graph

Recall the process of state expansions. The generated hypotheses and the expansions that link them form a graph. Paths branch out when there are multiple translation options for a hypothesis, from which multiple new hypotheses can be derived. Paths join when hypotheses are recombined.

Usually, when the decoder recombines hypotheses, it simply discards the worst hypothesis, since it cannot possibly be part of the best path through the search graph (in other words, part of the best translation).

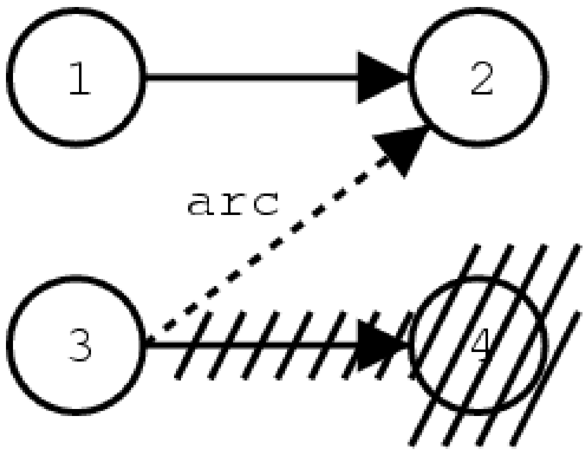
But since now the second best translation is also needed, information about that hypothesis cannot simply be discarded. If it is discarded, the search graph would only contain one path for each hypothesis in the last hypothesis stack.

If the information on the multiple ways to reach a hypothesis is stored, the number of possible paths also multiplies along the path when the decoder traverses backward through the graph.

In order to keep the information about merging paths, a record of such merges is kept, containing:

- identifier of the previous hypothesis
- identifier of the lower-cost hypothesis
- cost from the previous to higher-cost hypothesis.

Figure 3.11 [19] gives an example for the generation of such an arc: in this case, hypotheses 2 and 4 are equivalent in respect to the heuristic search, as detailed above. Hence, hypothesis 4 is deleted. But since we want keep the information about the path leading from hypothesis 3 to 2, it is stored a record of this arc. The arc also contains the cost added from hypothesis 3 to 4. Note that the probability from hypothesis 1 to hypothesis 2 does not have to be stored, since it can be recomputed from the hypothesis data structures.



2.11 Arcs in Search Graph

Mining the Search Graph for an n-Best List

The graph of the hypothesis space can also be viewed as a probabilistic finite-state automaton. The hypotheses are states, and the records of back-links and the additionally stored arcs are state transitions. The added probability scores when expanding a hypothesis are the costs of the state transitions.

Finding the n-best path in such a probabilistic finite state automaton is a well-studied problem. In this implementation, the decoder stores the information about hypotheses, hypothesis transitions, and additional arcs in a file that can be processed by the finite state toolkit Carmel, which is used to mine the n-best lists. This toolkit uses the k_ shortest paths algorithm by Eppstein [25].

2.2 Parallel text mining

To create a Statistical Machine Translator, it is necessary to get parallel text in the languages in which you want to make the SMT work. The speech-to-speech translator performed is English to Spanish and Spanish to English in a medical domain, and the parallel text has been taken from a medical encyclopedia, a dictionary with medical terms, the transcripts of the European Parliament (Europarl) and from some books of conversational Spanish for English speakers.

It has been used for a total of 551,789 lines of parallel text (almost 10,000,000 words per language), including all the medical terms and the conversational sentences. From the transcripts of the European Parliament, only a small sample has been used.

To get the text from the Medline encyclopedia [7], two scripts have been used[10]. Those scripts were opening the different articles of the webpage, and copying them in a txt file of each language. Once all the articles were copied in txt files, the articles that were not translated sentences by sentence were deleted: to find these articles and delete them, another script was used which counts the number of sentences in English and the number of sentences in Spanish, and if the number of sentences is the same the script assumes that the articles are translated sentence by sentence. Before creating this script we found out that in the encyclopedia the articles that were not translated sentence by sentence had a different number of sentences (to check this we made a random confirmation that this was the case in 10% of the articles, and 100% of the random test cases followed this pattern).

The texts from the Europarl, since it was text prepared to work with Moses to create a SMT (Statistical Machine Translator), were already aligned.

And to get the text from Spanish learning e-books [8], [9], [10] another script was used[11]: this script operated similarly to the one used in Medline, by obtaining the text from the web pages and making a txt file with the sentences in each language.

With all this parallel text, the SMT (Statistical Machine Translator) can be made with Moses.

[10] Appendix II: Script Manuals. Parallel text

2.3 Statistical Machine Translator models

To perform a two-way speech-to-speech translator, two Statistical Machine Translators are needed: one that translates from English to Spanish and another that translates from Spanish to English.

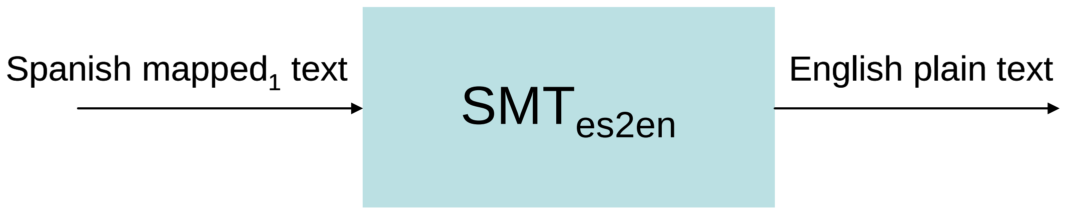
Spanish to English

The input of the Spanish to English SMT model comes from the Spanish ASR, and, as explained in the Spanish ASR V2.2, the output of the ASR is mapped in a specific way. So to build the SMT, a mapped Language Model and a mapped Phrase Table have been used.

The Language Model is a 5-gram built with SRILM, with corresponding labels:

- Accents changed for the combination of the vowel with "WW"
- And the letter "Ñ" changed to "NY."

The Phrase Tables have been built with the Moses training script[11], giving the parallel text and the language model as an input. The Spanish part of the parallel text has the same labels as in the Language Model, and the English part was written in plain English since the Text to Speech system reads plain English without any special mapping.



2.12 Layout Spanish to English SMT

[11] Appendix IV: Moses manual

English to Spanish

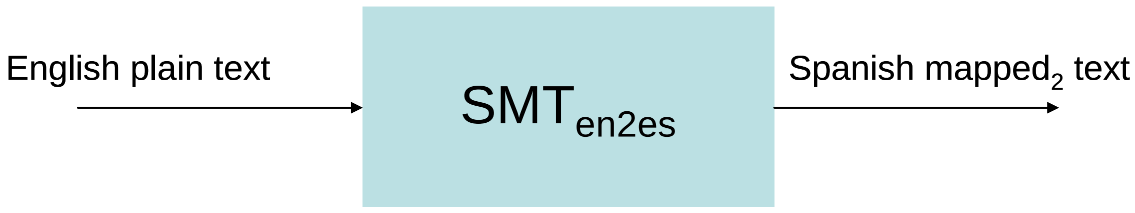
The input of the English to Spanish SMT has to be English written straightforward. However, the output (Spanish) has to be translated in a way that the Text to Speech system understands it.

The Language Model is also a 5-gram made with SRILM, but now with plain English words without any mapping.

Nevertheless, the Phrase Tables, which are also created with the same Moses script as in the Spanish to English model, have different labels in the Spanish side; these labels are defined by the Festival TTS (Text To Speech):

- The accents are configured as "vowel." For example, the accented vowel "**Á**" is mapped as an "**A.**"
- And the letter "**Ñ**" is tagged as "**NY.**"

Once the Phrase Tables and the Language Models are created, the Moses decoder is used to make text to text translations. This decoder will be used in the whole system to make the speech-to-speech translations.



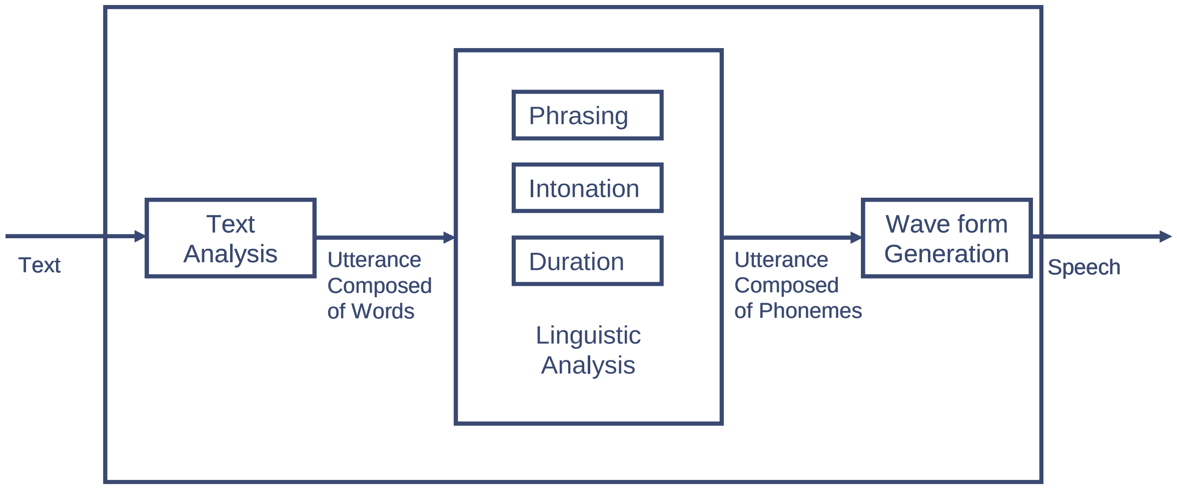
2.13 Layout English to Spanish SMT

3. Text To Speech

Speech synthesis is the artificial production of human speech. A computer system used for this purpose is called a speech synthesizer, and can be implemented in software or hardware. A TTS (Text to Speech) system converts normal language text into speech [26].

Synthesized speech can be created by concatenating pieces of recorded speech that are stored in a database. Systems differ in the size of the stored speech units; a system that stores phones or diphones provides the largest output range, but may lack clarity. For specific usage domains, the storage of entire words or sentences allows for high-quality output. Alternatively, a synthesizer can incorporate a model of the vocal tract and other human voice characteristics to create a completely "synthetic" voice output [27].

The quality of a speech synthesizer is judged by its similarity to the human voice and by its ability to be understood. An intelligible text-to-speech program allows people with visual impairments or reading disabilities to listen to written works on a home computer. Since the early 1980s, many computer operating systems have included speech synthesizers.



3.1 Text To Speech synthesizer layout

A text-to-speech system is composed of two parts, a front-end and a back-end. The front-end has two major tasks. First, it converts raw text containing symbols like numbers and abbreviations into the equivalent of written-out words. This process is often called text normalization, pre-processing, or tokenization. The front-end then assigns phonetic transcriptions to each word and divides and marks the text into prosodic units, like phrases, clauses and sentences. The process of assigning phonetic transcriptions to words is called text-to-phoneme or grapheme-to-phoneme conversion [28]. Phonetic transcriptions and prosody information together make up the symbolic linguistic representation that is output by the front-end. The back-end—often referred to as the synthesizer—then converts the symbolic linguistic representation into sound.

3.1 Text To Speech tools

Spanish Text To Speech

Festvox lpcu, the Spanish package of Festival, was used for the Spanish Text To Speech system.

Festival is an open source speech synthesis system for multi-purpose language. It was originally developed by the Research Center of Language Technologies at the University of Edinburgh, although Carnegie Mellon University and other schools have made substantial contributions to the project.

The project, which is programmed in C++, includes the complete documentation to develop speech synthesis and is ideal for development and research of speech synthesis techniques.

The festival project is multilingual (currently supports English--British and American--and Castilian), although English is the most advanced. Furthermore, some groups have developed tools that allow other languages in the project.

The tools and documentation for the utilization of new voices in the system are available in the FestVox project from the CMU (Carnegie Mellon University).

FestVox project

The goal of FestVox is to make the construction of new synthetic voices more systematic and better documented, making it possible for anyone to build new voices.

Project Festvox is a toolkit for building synthetic voices for the Festival's Text To Speech synthesizer. This includes a step-by-step tutorial with examples.

English Text To Speech

On the other hand, for the English Text To Speech conversion, a Cepstral system was used.

In contrast with previous technologies, which are either very large systems or offer lower quality due to outdated technology, Cepstral's TTS engines and voices can be deployed on mobile devices or in multiple instances on server platforms, making it the easiest to use and most versatile product available today.

Cepstral has created new techniques for general-purpose voices and "domain voices" which allow the spoken output to be tailored to an application. This is combined in a single software application, resulting in extremely versatile, high-quality voices.

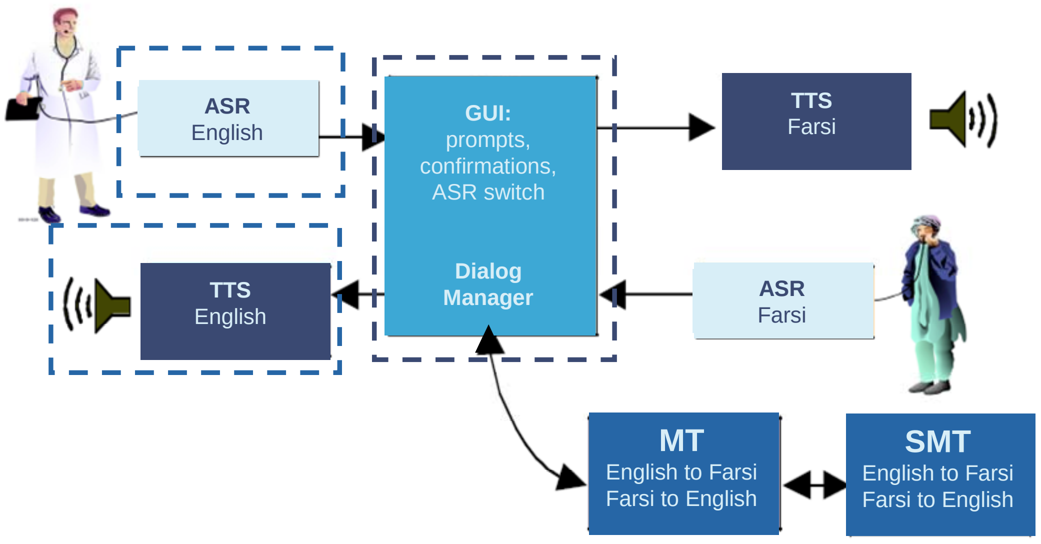
4. Putting everything together

The final two-way speech-to-speech translation system is based in USC's (University of Southern California) SpeechLinks solution [29]. The SpeechLinks translation system was developed in the SAIL (Signal Analysis and Interpretation Laboratory) laboratory and translates from Farsi (the most widely spoken Persian language) to English and vice versa.

SpeechLinks receives an input speech, then the speech is converted into text by the ASR (Automatic Speech Recognizer), then the text is translated by the MT (Machine Translator) and synthesized with the target's language TTS.

The English ASR system works on a vocabulary of over 22,000 words, and it gives high-quality results functioning in real time. This unit uses models of human speech trained by example recordings and statistical knowledge.

The core of the SpeechLinks system is the DM (Dialog Manager), which redirects the messages to enable the internal communication of the system. It gets the text from the ASR system, displays it in the GUI (Graphical User Interface) and sends it to the MT, and then it receives the translated text from the MT and sends it to the TTS synthesizer, which gives an output.



4.1 Layout of SpeechLinks [1]

Some changes have to be applied to the system to make it work with Spanish instead of Farsi:

First, replace the Farsi ASR with the Spanish ASR V2.2 developed in this project.

Second, use Spanish to English two-way Machine Translator.

Third, instead of the Farsi Text To Speech synthesizers, put the Spanish TTS developed by Festival.

And last, make some modifications over the GUI to write the screen messages with Latin alphabet and also make some changes in the way that the Dialog Manger tags the messages.

If all those changes are made and the English ASR and TTS units are left in the program, the result is a two-way speech-to-speech translator (in this case working on a medical domain).

Once the two-way speech-to-speech translator is done, these steps must be followed in order for it to function:

1. Run the Spanish and English ASR systems.

2. Run the English to Spanish and the Spanish to English MT.

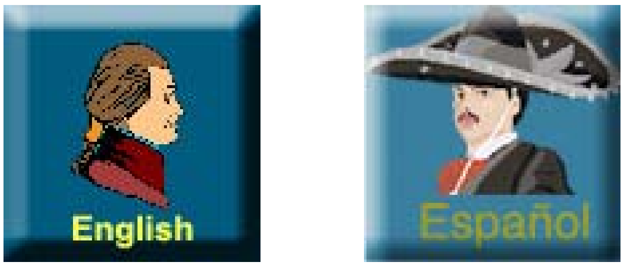
3. Run the system.

4.2 Speech-to-Speech system starts

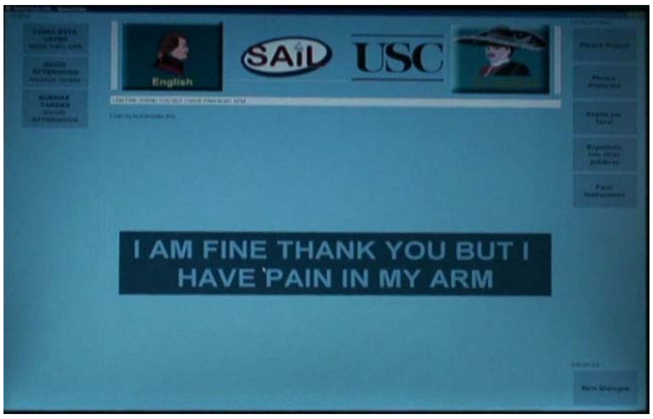
Once the system is fully loaded:

To translate from English to Spanish, click on the English (figure 5.3) button, speak over the microphone and click again on the English button, then if the hypothesis that the system gives in the middle of the screen is correct, click over it and the system will automatically translate it.

To translate from Spanish to English, follow the same process, clicking on the Spanish button rather than the English one.



4.3 English and Spanish buttons



I AM FINE THANK YOU BUT I HAVE PAIN IN MY ARM

4.4 Speech-to-speech translation system running

5. Conclusions

During this project a two-way speech-to-speech translator system that translated English and Farsi was converted to one that works in Spanish and English.

To perform this adaptation some new units have been designed (Spanish Automatic Speech Recognizer, Spanish-English and English-Spanish Machine Translator and a Spanish Text To Speech synthesizer have been used). Even all these units work fairly well, they cannot be considered totally robust applications.

In order to make them more robust, the ASR should be trained with a bigger corpus, and it is recommendable to use a corpus more based in a medical domain. Moreover, the Machine Translator also should be trained with more parallel text provided from medical conversations. To text could be obtained from medical TV shows' subtitles (like *House*, *Grey's Anatomy*, etc.).

With these two new trainings, the new system would be notably more robust.

6. References

[1] Ettelaie, E., Gandhe, S., Georgiou, P., Knight, K., Marcu, D., Narayanan, S. et al. (2005). *Transonics: A Practical Speech-to-Speech Translator for English-Farsi Medical Dialogues.* University of Southern California, Los Angeles, CA.

[2] Huang, X., Acreo, A. & Hon H.W. (2001). *Spoken Language Processing: A guide to theory, algorithm and system development.* Prentice Hall PTR, New Jersey.

[3] Stolcke, A. (2002). *SRILM – An extensible language modeling toolkit*. Speech Technology and Research Laboratory SRI International, Menlo Park, CA.

[4] Lee, K.F., Hon, H.W. & Reddy, R. (1990). *An overview of the SPHINX speech recognition system*. Morgan Kaufmann Publishers Inc. San Francisco, CA.

[5] International Computer Science Institute (n.d.). Sclite manual. http://www.icsi.berkeley.edu/Speech/docs/sctk-1.2/sclite.htm

[6] Linguistic Data Consortium (2006). West Point Heroico Spanish Speech. http://www.ldc.upenn.edu/Catalog/CatalogEntry.jsp?catalogId=LDC2006S37

[7] National Library of Medicine (n.d.). MedlinePlus Health Information from the National Library of Medicine. http://medlineplus.gov/

[8] Flash Cards Exchange (n.d.). Conversational and Medical Spanish. http://www.flashcardexchange.com/tag/Spanish

[9] 1, 2, 3 TeachMe (n.d.). Conversational Spanish. http://www.123teachme.com/learn_spanish/conversational_spanish

[10] Wikilearning: comunidades libres para aprender (n.d.). *Libro de frases Español.* http://www.wikilearning.com/monografia/libro_de_frases_espanol_english_ingles/2818

[11] Navarro Tomás, T. (1918). *Manual de la pronunciación española.* Consejo Superior de Investigaciones científicas. Spain.

[12] Quilis, A. (1987). *Fonetica Acustica De La Lengua Española*. Editorial Gredos, S.A., Spain.

[13] Enriquez, E., Casado, C. & Santos, A. (1989). *La percepción del acento en español.* Lingüística española actual. Spain.

[14] Navarro Tomás, T. (1944). *Manual de la entonación española.* Hispanic Institute, NY. Guadarrama, Spain.

[15] Weaver, W. (1955). “Translation” (1949). In: Machine Translation of Languages, MIT Press, Cambridge, MA.

[16] Brown, P., Della Pietra, S., Della Pietra, V., & Mercer, R. (1993). “The mathematics of statistical machine translation: parameter estimation.” Computational Linguistics.

[17] Och, F.J. & Ney, H. (2000). *Improved Statistical Alignment Models.* In Proceedings of the 38th Annual Meeting of the Association for Computational Linguistics

[18] Marcu, D. & Wong, W. (2002). *A Phrase-Based, Joint Probability Model for Statistical Machine Translation.* In Proceedings of the 2002 Conference on Empirical Methods in Natural Language Processing (EMNLP).

[19] Koehn, P. (n.d). *Background on Statistical Machine Translator*. http://www.statmt.org/moses/?n=Moses.Background.

[20] Tillmann, C (2001). *Word Re-ordering and Dynamic Programming based Search Algorithm for statistical Machine Translation*. PhD Thesis, RWTH Aachen, Germany.

[21] Och, F.J. (2002). *Statistical Machine Translation: From Single-Word Models to Alignment Templates*. Ph.D. thesis, Lehrstuhl für Informatik 6, Computer Science Department, RWTH Aachen University, Germany.

[22] Jelinek, F. (1997). *Statistical methods for speech recognition.* MIT Press Cambridge, MA, USA

[23] Ueffing, N., Och, F.J., Ney, H. (2002). *Generation of Word Graphs in Statistical Machine Translation.* In Proc. of the Conf. on Empirical Methods for Natural Language Processing (EMNLP). Philadelphia, PA.

[24] Langkilde-Geary I. (2002). *An Empirical Verification of Coverage and Correctness for a General-Purpose Sentence Generator*. In Proceedings of INLG2002, New York, NY.

[25] Eppstein, D. (1997). *Finding the k Shortest Paths*. University of California Irvine, California.

[26] Allen, J., Hunnicutt, M.S. & Klatt, D. (1987). From *Text to Speech: The MITalk system*. Cambridge University Press. Cambridge, MA.

[27] Rubin, P., Baer, T., & Mermelstein, P. (1981). *An articulatory synthesizer for perceptual research*. Journal of the Acoustical Society of America. Melville, NY.

[28] Van Santen, J.P.H., Sproat, R., Olive, J., & Hirschberg, J. (1997). *Progress in Speech Synthesis.* Springer-Verlag New York, Inc., New York, NY.

[29] Signal Analysis and Interpretation Laboratory, USC (n.d.). SpeechLinks Solution. http://sail.usc.edu/transonics/s2s.php#speechlinks_solutions.

[30] Department of Electrical & Computer Engineering, CMU (2008). Learning to use the CMU SPHINX Automatic Speech Recognition system. www.speech.cs.cmu.edu/sphinx/tutorial.html

[31] Statistical Machine Translation at the University of Edinburgh (n.d.) Moses Installation and Training Run-Through. www.statmt.org/moses_steps.html

Appendix I: Files outline

- Perl scripts

a. Dictionaries

   i. add_tonic_syllabi.pl
   ii. create_spanish_dictV1.pl
   iii. create_spanish_dictV2_1.pl
   iv. create_spanish_dictV2_2.pl
   v. create_spanish_dictV3.pl
   vi. make_phoneList.pl
   vii. sort_and_num_dic_entries.pl

b. Format text

   i. create_phone_transcript.pl
   ii. create_phone_transcript_aligned.pl
   iii. make_plaintext.pl
   iv. new_fileids_transcription_and_plaintext.pl
   v. renaming.pl
   vi. statistics_of_text.pl
   vii. transcripts_and_fileids_heroico_answ.pl
   viii. transcripts_and_fileids_heroico_record.pl
   ix. transcripts_and_fileids_usma.pl

c. parallel text

   i. compare_lines.pl
   ii. get_med_dic.pl

        iii. get_medline.pl

        iv. get_teachme.pl

        v. medline_parallel_eng.pl

        vi. medline_parallel_spa.pl

- Acoustic Models

  a. Spanish_V1

  b. Spanish_V2_1

  c. Spanish_V2_2

  d. Spanish_V3

  e. feat

  f. wav

- Statistical Machine Translator

  a. english2spanish

  b. spanish2english

- Text to Speech

  a. Spanish

- Definitive Models

  a. asr

  b. opt

  c. tts

Appendix II: Script manuals

Dictionaries

create_spanish_dictV*.pl

SYNOPSIS

perl **create_spanish_dictV*.pl** [OPTION1] [FILE] [OPTION2] [FILE]

DESCRIPTION

Creates a phonetic dictionary from a given regular, non-formatted text

OPTION1:

-ft /home/usr/folder/ with this option the input comes from a folder with different text files (it will read all the text files from the folder)

-t /home/usr/folder/textfile.txt the input data comes from the given text file

-help prints the help option in the standard output

OPTION2:

(default) prints the dictionary in the standard output

-ad /home/usr/folder/dictionary.dic the script actualizes the existing dictionary in dictionary.dic, adding the new words to it.

-nd /home/usr/folder/dictionary.dic to create a new dictionary in dictionary.dic

add_tonic_syllabi.pl

SYNOPSIS

perl **add_tonic_syllabi.pl** [FILE]

DESCRIPTION

This script complements the version 3 of the dictionaries adding the tonic phoneme in the accented words, the input file has to be an existing dictionary created with the version 3 and the output dictionary is given through the standard output.

make_phoneList.pl

SYNOPSIS

perl **make_phoneList.pl** [FILE]

DESCRIPTION

From a given dictionary this script creates a phoneme list, and it is printed in the standard output.

sort_and_num_dic_entries.pl

SYNOPSIS

perl **sort_and_num_dic_entries.pl** [FILE]

DESCRIPTION

This script gets a phonetic dictionary as an input and it gives as an output an alphabetically sorted and with the repetition number (words with more than one pronunciation).

Format text

create_phone_transcript.pl

SYNOPSIS

perl **create_phone_transcripts.pl** [FILE]

DESCRIPTION

This script converts a regular transcript to a phonetic transcript, it uses the dictionary V1 and the Mexican pronunciation.

create_phone_transcript_aligned.pl

SYNOPSIS

perl **create_phone_transcript_aligned.pl** [FILE1] [FILE2] [FILE3]

DESCRIPTION

From a given aligned transcript, and a dictionary it converts the transcript in a phonetic transcript with the correct pronunciations. The input file1 has to be the aligned transcript, the file2 is for the dictionary and the file 3 has to be a list of the ids from the transcript.

make_plaintext.pl

SYNOPSIS

perl **make_plaintext.pl** [FILE1]

DESCRIPTION

This script gets a transcript as an input, and gives a text without the IDs (only with the language text) as a output.

new_fileids_transcription_and_plaintext.pl

SYNOPSIS

perl **new_fileids_transcription_and_plaintext.p**l [FILE1] [DIR_OUT] [FILE2] [DATABASE]

DESCRIPTION

This script get from all the txt files that are in a folder tree, reads them, and then gives 3 files as an output in the given directory. The 3 output files are: a transcript with the needed tags and IDs, a list with all the directions of the IDs, and a file with only the language text.

File1 has to be user name

DIR_OUT is the directory where you want the outputs

FILE2 is for the root name of the outputs

Database is to put the root directory of the tree where you want to look for txt files.

renaming.perl

SYNOPSIS

perl **renaming.perl** [FILE1]

DESCRIPTION

This script takes a regular transcript and changes the regular IDs for numerical ones; it is useful to evaluate an acoustic model.

statistics_of_text.pl

SYNOPSIS

perl **statistics_of_text.pl** [FILE1]

DESCRIPTION

This script counts the words of a text and gives as an output (by the standard output) the repetitions of each word.

transcripts_and_fileids_*.pl

SYNOPSIS

perl **transcripts_and_fileids_*.pl** [FILE1] [DATABASE]

DESCRIPTION

This script gets the named corpus, and transcripts and creates the transcript with the format needed for SphinxTrain, and then an IDs list and all the transcripts with just plain text.

Parallel text

get_medline.pl

SYNOPSIS

perl **get_medline.pl** [FOLDER1] [FOLDER2] [FILE]

DESCRIPTION

This script gets the text from all the articles in the medline encyclopedia, making a file from each of them in the corresponding folder. Folder1 is for the English articles, Folder2 for the Spanish and the file is for an error report.

Medline_parallel_*.pl

SYNOPSIS

perl **Medline_parallel_*.pl** [FOLDER] [FILE1] [FILE2]

DESCRIPTION

This script gets from the folder all the English or Spanish (depending on with script are you using) articles gotten from the Medline encyclopedia and cleans them from the html programming text, the definitive output files are found in the same folder as file2. File1 is just an state between the clean text and the one with all the programming text.

compare_lines.pl

SYNOPSIS

perl **compare_lines.pl** [FOLDER1] [FOLDER2] [FOLDER3]

DESCRIPTION

This script counts the number of lines on each article of the folder1 and compares them with the number of lines of the same article of the folder2. If the number of lines matches it creates two folders inside the folder3, one with the matching Spanish articles and another one with the English articles. If they do not match, then it makes two folders with the articles that did not match.

get_med_dic.pl

SYNOPSIS

perl **get_med_dic.pl** [FILE1] [FILE2] [FILE3]

DESCRIPTION

This script gets all the words from the medical dictionary that can be found in http://users.ugent.be/ and writes in file2 the English word and in file3 the Spanish translation; file1 is a report of errors. This script maintains the parallelism between both languages.

get_teachme.pl

SYNOPSIS

perl **get_teachme.pl** [FILE1] [FILE2] [FILE3]

DESCRIPTION

This script gets all the conversational sentences that can be found in www.123teachme.com and writes in file2 the English sentence and in file3 the Spanish translation; file1 is a report of errors. This script maintains the parallelism between both languages.

Appendix III: Sphinx tutorial

This tutorial has been performed at the Carnegie Mellon University [30]

LEARNING TO USE THE CMU SPHINX AUTOMATIC SPEECH RECOGNITION SYSTEM

Introduction

In this tutorial, you will learn to handle a complete state-of-the-art HMM-based speech recognition system. The system you will use is the SPHINX system, designed at Carnegie Mellon University. SPHINX is one of the best and most versatile recognition systems in the world today.

An HMM-based system, like all other speech recognition systems, functions by first learning the characteristics (or parameters) of a set of sound units, and then using what it has learned about the units to find the most probable sequence of sound units for a given speech signal. The process of learning about the sound units is called training. The process of using the knowledge acquired to deduce the most probable sequence of units in a given signal is called decoding, or simply recognition.

Accordingly, you will need those components of the SPHINX system that you can use for training and for recognition. In other words, you will need the SPHINX trainer and a SPHINX decoder.

You will be given instructions on how to download, compile, and run the components needed to build a complete speech recognition system. Namely, you will be given instructions on how to use SphinxTrain and you will have to choose one of PocketSphinx, SPHINX-2, SPHINX-3, SPHINX-3 Flat, or SPHINX-4. Please check a short description for capabilities of each of these, or the CMUSphinx project page for more details. This tutorial does not instruct you on how to build a language model, but you can check the CMU SLM Toolkit page for an excellent manual.

At the end of this tutorial, you will be in a position to train and use this system for your own recognition tasks. More importantly, through your exposure to this system, you will have learned about several important issues involved in using a real HMM-based ASR system.

Important note for members of the Sphinx group: This tutorial now supports the PBS queue. The internal, csh-based Robust tutorial is still available, though its use is discouraged.

Components provided for training

The SPHINX trainer consists of a set of programs, each responsible for a well defined task, and a set of scripts that organizes the order in which the programs are called. You have to compile the code in your favorite platform.

The trainer learns the parameters of the models of the sound units using a set of sample speech signals. This is called a training database. A choice of training databases will also be provided to you. The trainer also needs to be told which sound units you want it to learn the parameters of, and at least the sequence in which they occur in every speech signal in your training database. This information is provided to the trainer through a file called the transcript file, in which the sequence of words and non-speech sounds are written exactly as they occurred in a speech signal, followed by a tag which can be used to associate this sequence with the corresponding speech signal. The trainer then looks into a dictionary which maps every word to a sequence of sound units, to derive the sequence of sound units associated with each signal. Thus, in addition to the speech signals, you will also be given a set of transcripts for the database (in a single file) and two dictionaries, one in which legitimate words in the language are mapped sequences of sound units (or sub-word units), and another in which non-speech sounds are mapped to corresponding non-speech or speech-like sound units. We will refer to the former as the language dictionary and the latter as the filler dictionary.

In summary, the components provided to you for training will be:

1. The trainer source code
2. The acoustic signals
3. The corresponding transcript file
4. A language dictionary
5. A filler dictionary

Components provided for decoding

The decoder also consists of a set of programs, which have been compiled to give a single executable that will perform the recognition task, given the right inputs. The inputs that need to be given are: the trained acoustic models, a model index file, a language model, a language dictionary, a filler dictionary, and the set of acoustic signals that need to be recognized. The data to be recognized are commonly referred to as test data.

In summary, the components provided to you for decoding will be:

1. The decoder source code
2. The language dictionary
3. The filler dictionary
4. The language model
5. The test data

In addition to these components, you will need the acoustic models that you have trained for recognition. You will have to provide these to the decoder. While you train the acoustic models, the trainer will generate appropriately named model-index files. A model-index file simply contains numerical identifiers for each state of each HMM, which are used by the trainer and the decoder to access the correct sets of parameters for those HMM states. With any given set of acoustic models, the corresponding model-index file must be used for decoding. If you would like to know more about the structure of the model-index file, you will find a description following the link Creating the CI model definition file.

Setting up your system

You will have to download and build several components to set up the complete systems. Provided you have all the necessary software, you will have to download the data package, the trainer, and one of the SPHINX decoders. The following instructions detail the steps.

Required software before you start

You will need Perl to run the provided scripts, and a C compiler to compile the source code. Additionally, if you choose to use SPHINX-4, you will need the Java (TM) Runtime Environment and, if you need to compile code, the Java platform compiler. You may also want to measure the word error rate using a word alignment program, such as NIST's sclite.

Perl

You will need Perl to use the scripts provided. Linux usually comes with some version of Perl. If you do not have Perl installed, please check the Perl site, where you can download it for free.

C Compiler

SphinxTrain, PocketSphinx, SPHINX-2, and SPHINX-3 use GNU autoconf to find out basic information about your system, and should compile on most Unix and Unix-like systems, and certainly on Linux. The code compiles using GNU's make and GNU's C compiler (gcc), available in all Linux distributions, and available for free for most platforms.

We also provide files supporting compilation using Microsoft's Visual C++, i.e., the solution (.sln) and project (.vcproj) files needed to compile code in native Windows format.

Java Platform

The Java platform is not necessary for SphinxTrain, PocketSphinx, SPHINX-2, or SPHINX-3. However, SPHINX-4 was written in the Java Programming Language. You can download the binaries directly, which you can use in any platform if you have the Java Runtime Environment (JRE). If you want to compile the SPHINX-4 code, you will also need the Java JDK. Both the JRE and the JDK are available at the Java Technology site.

In addition to the Java compiler, you will also need ant to compile. ant is similar to make and is available from apache.org.

Word Alignment

You will need a word alignment program if you want to measure the accuracy of a decoder. A commonly used one, available from the National Institute of Standards and Technology (NIST), is sclite, provided as part of their scoring packages. You will find their scoring packages in the NIST tools page. The software is available for those in the speech group at ~robust/archive/third_party_packages/NIST_scoring_tools/sctk/linux/bin/sclite.

Internally, at CMU, you may also want to use the align program, which does the same job as the NIST program, but does not have some of the features. You can find it in the robust home directory at ~robust/archive/third_party_packages/align/linux/align.

Setting up the data

The Sphinx Group makes it available two audio databases that can be used with this tutorial. Each has its peculiarities, and are provided just as a convenience. The data provided are not sufficient to build a high performance speech recognition system. They are only provided with the goal of helping you learn how to use the system.

The databases are provided at the Databases page. Choose either AN4 or RM1. AN4 includes the audio, but it is a very small database. You can choose it if you want to include the creation of feature files in your experiments. RM1 is a little larger, thus resulting in a system with slightly better performance. Audio is not provided, since it is licensed material. We provide the feature files used directly by the trainer and decoders. For more information about RM1, please check with the LDC.

The steps involved:

1. Create a directory for the system, and move to that directory:

```
mkdir tutorial
cd tutorial
```

2. Download the audio tarball AN4 , by clicking on the link and choosing "Save" when the dialog window appears. Save it to the same tutorial directory you just created. For those not familiar with the term, a tarball in our context is a file with extension .tar.gz. Extract the contents as follows.

3. In Windows, using the Windows Explorer, go to the tutorial directory, right-click the audio tarball, and choose "Extract to here" in the WinZip menu.

4. In Linux/Unix:

```
gunzip -c an4_sphere.tar.gz | tar xf -
```

By the time you finish this, you will have a **`tutorial`** directory with the following contents

```
tutorial
    an4
    an4_sphere.tar.gz
```

Setting up the trainer

Code retrieval

SphinxTrain can be retrieved using subversion (svn) or by downloading a tarball. svn makes it easier to update the code as new changes are added to the repository, but requires you to install svn. The tarball is more readily available.

You can find more information about svn at the SVN Home.

Using svn

```
svn co
https://cmusphinx.svn.sourceforge.net:/svnroot/cmusphinx/trunk/Sphi
nxTrain
```

- Using the tarball, download the SphinxTrain tarball by clicking on the link and choosing "Save" when the dialog window appears. Save it to the same **tutorial** directory. Extract the contents as follows.
    - In Windows, using the Windows Explorer, go to the **tutorial** directory, right-click the SphinxTrain tarball, and choose "Extract to here" in the WinZip menu.
    - In Linux/Unix:

```
gunzip -c SphinxTrain.nightly.tar.gz | tar xf -
```

Further details about download options are available in the cmusphinx.org page, under the header *Download instructions*

By the time you finish this, you will have a **tutorial** directory with the following contents

```
tutorial
     an4
     an4_sphere.tar.gz
     SphinxTrain
     SphinxTrain.nightly.tar.gz
```

Compilation

In Linux/Unix:

```
cd SphinxTrain
configure
make
```

Tutorial Setup

After compiling the code, you will have to setup the tutorial by copying all relevant executables and scripts to the same area as the data. Assuming your current working directory is **tutorial**, you will need to do the following.

```
cd SphinxTrain
# If you installed AN4
perl scripts_pl/setup_tutorial.pl an4
# If you installed RM1
perl scripts_pl/setup_tutorial.pl rm1
```

Setting up the decoder

SPHINX-3

Code retrieval

SPHINX-3 can be retrieved using subversion (svn) or by downloading a tarball. svn makes it easier to update the code as new changes are added to the repository, but requires you to install svn. The tarball is more readily available. SPHINX-3 is also available as a release from SourceForge.net. Since the release is a tarball, we will not provide separate instructions for installation of the release.

You can find more information about svn at the SVN Home.

- Using svn

```
svn co
https://cmusphinx.svn.sourceforge.net:/svnroot/cmusphinx/trunk/sphi
nxbase
svn co
https://cmusphinx.svn.sourceforge.net:/svnroot/cmusphinx/trunk/sphi
nx3
```

- Using the tarball, download the sphinx3 tarball and sphinxbase by clicking on the link and choosing "Save" when the dialog window appears. Save them to the same `tutorial` directory. Extract the contents as follows.

    -
    - In Linux/Unix:

```
gunzip -c sphinxbase.nightly.tar.gz | tar xf -
gunzip -c sphinx3.nightly.tar.gz | tar xf -
```

Further details about download options are available in the cmusphinx.org page, under the header *Download instructions*

By the time you finish this, you will have a **tutorial** directory with the following contents

```
tutorial
an4
an4_sphere.tar.gz
SphinxTrain
SphinxTrain.nightly.tar.gz
sphinx3
sphinx3.nightly.tar.gz
sphinxbase
sphinxbase.nightly.tar.gz
```

Compilation

In Linux/Unix:

```
# Compile sphinxbase
cd sphinxbase
# If you used svn, you will need to run autogen.sh, commented out
# here. If you downloaded the tarball, you do not need to run it.
#
# ./autogen.sh
./configure
make

# Compile SPHINX-3
cd sphinx3
# If you used svn, you will need to run autogen.sh, commented out
# here. If you downloaded the tarball, you do not need to run it.
#
# ./autogen.sh
configure --prefix=`pwd`/build --with-
sphinxbase=`pwd`/../sphinxbase
make
make install
```

Tutorial Setup

After compiling the code, you will have to setup the tutorial by copying all relevant executables and scripts to the same area as the data. Assuming your current working directory is **tutorial**, you will need to do the following.

```
cd sphinx3
# If you installed AN4
perl scripts/setup_tutorial.pl an4
```

How to perform a preliminary training run

Go to the directory where you installed the data. If you have been following the instructions so far, in linux, it should be as easy as:

```
# If you are using AN4
cd ../an4
```

The scripts should work "out of the box", unless you are training models for PocketSphinx or SPHINX-2. In this case, you have to edit the file **etc/sphinx_train.cfg**, uncommenting the line defining the variable **$CFG_HMM_TYPE** so that it looks like the box below.

```
#$CFG_HMM_TYPE = '.cont.'; # Sphinx III
```

On Linux machines, you can set up the scripts to take advantage of multiple CPUs. To do this, edit **etc/sphinx_train.cfg**, change the line defining the variable **$CFG_NPART** to match the number of CPUs in your system, and edit the line defining **$CFG_QUEUE_TYPE** to the following:

```
# Queue::POSIX for multiple CPUs on a local machine
# Queue::PBS to use a PBS/TORQUE queue
$CFG_QUEUE_TYPE = "Queue::POSIX";
```

If you have a grid of computers running the TORQUE or PBS batch system, you can schedule training jobs to be run on the grid by defining **$CFG_NPART** as noted above and editing **$CFG_QUEUE_TYPE** like the following:

```
# Queue::POSIX for multiple CPUs on a local machine
# Queue::PBS to use a PBS/TORQUE queue
$CFG_QUEUE_TYPE = "Queue::PBS";
```

The system does not directly work with acoustic signals. The signals are first transformed into a sequence of feature vectors, which are used in place of the actual acoustic signals. To perform this transformation (or parameterization) from within the directory **an4**, type the following command on the command line. If you are using Windows instead of linux, please replace the / character with \. Notice that if you downloaded **rm1** instead, the files are already provided in cepstra format, so you do not need, and in fact, cannot, follow this step.

```
perl scripts_pl/make_feats.pl  -ctl etc/an4_train.fileids
```

This script will compute, for each training utterance, a sequence of 13-dimensional vectors (feature vectors) consisting of the Mel-frequency cepstral coefficients (MFCCs). Note that the list of wave files contains a list with the full paths to the audio files. Since the data are all located in the same directory as you are working, the paths are relative, not absolute. You may have to change this, as well as the**an4_test.fileids** file, if the location of data is different. This step takes approximately 10 minutes to complete on a fast machine, but time may vary. As it is running, you might want to continuing reading. The MFCCs will be placed automatically in a directory called **./feat**. Note that the type of features vectors you compute from the speech signals for training and recognition, outside of this tutorial, is not restricted to MFCCs. You could use any reasonable parameterization technique instead, and compute features other than MFCCs. SPHINX-3 and SPHINX-4 can use features of any type or dimensionality. In this tutorial, however, you will use MFCCs for two reasons: a) they are currently known to result in the best recognition performance in HMM-based systems under most acoustic conditions, and b) this tutorial is not intended to cover the signal processing aspects of speech parameterization and only aims for a standard usable platform in this respect. Now you can begin to train the system.

In the scripts directory (**./scripts_pl**), there are several directories numbered sequentially from **00*** through **99***. Each directory either has a directory named **slave*.pl** or it has a single file with extension **.pl**. Sequentially go through the directories and execute either the the **slave*.pl** or the single **.pl** file, as below. As usual, if you are using Windows instead of linux, you have to replace the /character with \.

```
perl scripts_pl/00.verify/verify_all.pl
perl scripts_pl/10.vector_quantize/slave.VQ.pl
perl scripts_pl/20.ci_hmm/slave_convg.pl
perl scripts_pl/30.cd_hmm_untied/slave_convg.pl
perl scripts_pl/40.buildtrees/slave.treebuilder.pl
perl scripts_pl/45.prunetree/slave-state-tying.pl
perl scripts_pl/50.cd_hmm_tied/slave_convg.pl
perl scripts_pl/90.deleted_interpolation/deleted_interpolation.pl
perl scripts_pl/99.make_s2_models/make_s2_models.pl
```

Alternatively, you can simply run the RunAll.pl script provided.

```
perl scripts_pl/RunAll.pl
```

From here on, we will refer to the script that you have to run in each directory as simply **slave*.pl**. In directories where no such a file exists, please understand it as the single **.pl** file present in that directory.

The scripts will launch jobs on your machine, and the jobs will take a few minutes each to run through. Before you run any script, note the directory contents of your current directory. After you run each**slave*.pl** note the contents again. Several new directories will have been created. These directories contain files which are being generated in the course of your training. At this point you need not know about the contents of these directories, though some of the directory names may be self explanatory and you may explore them if you are curious.

One of the files that appears in your current directory is an **.html** file, named **an4.html** or **rm1.html**, depending on which database you are using. This file will contain a status report of jobs already executed. Verify that the job you launched completed successfully. Only then launch the next **slave*.pl** in the specified sequence. Repeat this process until you have run the **slave*.pl** in all directories.

Note that in the process of going through the scripts in **00*** through **99***, you will have generated several sets of acoustic models, each of which could be used for recognition. Notice also that some of the steps are required only for the creation of semi-continuous models, such as those used by SPHINX-2. If you execute these steps while creating continuous models, the scripts will benignly do nothing. Once the jobs launched from **20.ci_hmm** have run to completion, you will have trained the Context-Independent (CI) models for the sub-word units in your dictionary. When the jobs launched from the**30.cd_hmm_untied** directory run to completion, you will have trained the models for Context-Dependent sub-word units (triphones) with untied states. These are called CD-untied models and are necessary for building decision trees in order to tie states. The jobs in **40.buildtrees** will build decision trees for each state of each sub-word unit. The jobs in **45.prunetree** will prune the decision trees and tie the states. Following this, the jobs in **50.cd-hmm_tied** will train the final models for the triphones in your training corpus. These are called CD-tied models. The CD-tied models are trained in many stages. We begin with 1 Gaussian per state HMMs, following which we train 2 Gaussian per state HMMs and so on till the desired number of Gaussians per State have been trained. The jobs in **50.cd-hmm_tied** will automatically train all these intermediate CD-tied models. Stages **90.deleted-interpolation** and **99.make_s2_models** are meaningful only if you are training models for SPHINX-2. Deleted interpolation smooths the HMMs, which are then converted to the format used by SPHINX-2. At the end of *any* stage you may use the models for recognition. Remember that you may decode even while the training is in progress, provided you are certain that you have crossed the stage which generates the models you want to decode with. Before you decode, however, read the section called How to decode, and key decoding issues to learn a little more about decoding. This section also provides all the commands needed for decoding with each of these models.

You have now completed your training. The final models and location will depend on the database and the model type that you are using. If you are using RM1 to train continuous models, you will find the parameters of the final 8 Gaussian/state 3-state CD-tied acoustic models (HMMs) with 1000 tied states in a directory called **./model_parameters/rm1.cd_cont_1000_8/**. You will also find a model-index file for these models called **rm1.1000.mdef** in **./model_architecture/** . This file, as mentioned before, is used by the system to associate the appropriate set of HMM parameters with the HMM for each sound unit you are modeling. The training process will be explained in greater detail later in this document. If, however, you trained semi-continuous models with AN4, the final models will be located at **./model_parameters/an4.1000.s2models**, where you will find all files need to decode with SPHINX-2.

How to perform a preliminary decode

Decoding is relatively simple to perform. First, compute MFCC features for all of the test utterances in the test set. If you downloaded **rm1**, the files are already provided in cepstra format, so you do not need, and in fact, cannot, follow this step. To compute MFCCs from the wave files, from the top level directory, namely **an4**, type the following from the command line:

```
perl scripts_pl/make_feats.pl  -ctl etc/an4_test.fileids
```

This will take approximately 10 minutes to run.

You are now ready to decode. Type the command below.

```
perl scripts_pl/decode/slave.pl
```

This uses all of the components provided to you for decoding, *including* the acoustic models and model-index file that you have generated in your preliminary training run, to perform recognition on your test data. When the recognition job is complete, the script computes the recognition Word Error Rate (WER) or Sentence Error Rate (SER). Notice that the script comes with a very simple built-in function that computes the SER. Unless you are using CMU machines, if you want to compute the WER you will have to download and compile code to do so. A popular one, used as a standard in the research community, is available from NIST. Check the section on Word Alignment.

If you provide a program that does alignment, you can change the file **etc/sphinx_decode.cfg** to use it. You have to change the following line:

```
$DEC_CFG_ALIGN = "builtin";
```

If you are running the scripts at CMU, the line above will default to:

```
$DEC_CFG_ALIGN = \\
"/afs/cs.cmu.edu/user/robust/archive/third_party_packages/NIST_scor
ing_tools/sctk/linux/bin/sclite";
```

When you run the decode script, it will print information about the accuracy in the top level **.html** page for your experiment. It will also create two sets of files. One of these sets, with extension .match, contains the hypothesis as output by the decoder. The other set, with extension .align, contains the alignment generated by your alignment program, or by the built-in script, with the result of the comparison between the decoder hypothesis and the provided transcriptions. If you used the NIST tool, the **.html** file will contain a line such as the following if you used **an4**:

```
SENTENCE ERROR: 56.154% (73/130)   WORD ERROR RATE: 16.429%
(127/773)
```

Miscellaneous tools

Three tools are provided that can help you find problems with your setup. You will find two of these executables in the directory **bin**. You can download and install the third as indicated below.

1. **mk_mdef_gen**: Phone and triphone frequency analysis tool. You can use this to count the relative frequencies of occurrence of your basic sound units (phones and triphones) in the training database. Since HMMs are statistical models, what

you are aiming for is to design your basic units such that they occur frequently enough for their models to be well estimated, while maintaining enough information to minimize confusions between words. This issue is explained in greater detail in Appendix 1.

2. **printp**: Tool for viewing the model parameters being estimated.
3. **cepview**: Tool for viewing the MFCC files. Available as a tarball

How to train, and key training issues

You are now ready to begin your own exercises. For every training and decoding run, you will need to first give it a name. We will refer to the experiment name of your choice by **$taskname**. For example, the names given to the experiments using the two available databases are *an4*, and *rm1*. Your choice of **$taskname** will be used automatically in all the files for that training and recognition run for easy identification. All directories and files needed for this experiment will be copied to a directory named **$taskname**. Some of these files, such as data, will be provided by you (maybe copied from either**tutorial/an4** or **tutorial/rm1**). Other files will be automatically copied from the trainer or decoder installations.

A new task is created from an existing one in a directory named **$taskname** in parallel to the existing one. Assuming that you are copying a setup from the existing setup named **tutorial/an4**, the new task will be located at **tutorial/$taskname**. Remember to replace **$taskname** with the name of your choice.

In the following example, we do just that: we copy a setup from the **an4** setup. Notice that your current working directory is the existing setup. The new one will be created by the script.

```
cd an4
perl scripts_pl/copy_setup.pl -task $taskname
```

This will create a new setup by rerunning the **SphinxTrain** setup, then rerunning the decoder setup using the same decoder as used by the originating setup (in this case, **an4**), and then copying the configuration files, located under **etc**, to the new setup, with the file names matching the new task's.

Be warned that the **copy_setup.pl** script also copies the data, located under **feat** and **wav**, to the new location. If your dataset is large, this duplication may be wasting disk space. A great option would be to just link the data directories. The script, as is, does not support this because not all operating systems can create symbolic links.

After this you will work entirely within this **$taskname** directory.

Your tutorial exercise begins with training the system using the MFCC feature files that you have already computed during your preliminary run. However, when you train this time, you will be required to take certain decisions based on what you know and the information that is provided to you in this document. The decisions that you take will affect the quality of the models that you train, and thereby the recognition performance of the system.

You must now go through the following steps in sequence.

1. Parameterize the training database, if you used the **an4** database or are using your own data. If you used **an4**, you have already done this for every training utterance during your preliminary run. If you used **rm1**, the data were provided already parameterized. At this point you do not have to do anything further except to note that in the speech recognition field it is common practice to call each file in a database an "utterance". The signal in an "utterance" may not necessarily be a full sentence. You can view the cepstra in any file by using the tool **cepview**.
2. Decide what sound units you are going to ask the system to train. To do this, look at the language dictionary **$taskname/etc/$taskname.dic** and the filler dictionary**$taskname/etc/$taskname.filler**, and note the sound units in these. A list of all sound units in these dictionaries is also written in the file **$taskname/etc/$taskname.phone**. Study the dictionaries and decide if the sound units are adequate for recognition. In order to be able to perform good recognition, sound units must not be confusable, and must be consistently used in the dictionary. Look at Appendix 1 for an explanation.

   Also check whether these units, and the triphones they can form (for which you will be building models ultimately), are well represented in the training data. It is important that the sound units being modeled be well represented in the training data in order to estimate the statistical parameters of their HMMs reliably. To study their occurrence frequencies in the data, you may use the tool**mk_mdef_gen**. Based on your study, see if you can come up with a better set of sound units to train.

You can restructure the set of sound units given in the dictionaries by merging or splitting existing sound units in them. By merging of sound units we mean the clustering of two or more different sound units into a single entity. For example, you may want to model the sounds "Z" and "S" as a single unit (instead of maintaining them as separate units). To merge these units, which are represented by the symbols Z and S in the language dictionary given, simply replace all instances of Z and S in the dictionary by a common symbol (which could be Z_S, or an entirely new symbol). By splitting of sound units we mean the introduction of multiple new sound units in place of a single sound unit. This is the inverse process of merging. For example, if you found a language dictionary where all instances of the sounds Z and S were represented by the same symbol, you might want to replace this symbol by Z for some words and S for others. Sound units can also be restructured by grouping specific sequences of sound into a single sound. For example, you could change all instances of the sequence "IX D" into a single sound IX_D. This would introduce a new symbol in the dictionary while maintaining all previously existing ones. The number of sound units is effectively increased by one in this case. There are other techniques used for redefining sound units for a given task. If you can think of any other way of redefining dictionaries or sound units that you can properly justify, we encourage you to try it.

Once you re-design your units, alter the file **$taskname/etc/$taskname.phone** accordingly. Make sure you do not have spurious empty spaces or lines in this file.

Alternatively, you may bypass this design procedure and use the phone list and dictionaries as they have been provided to you. You will have occasion to change other things in the training later.

3. Once you have fixed your dictionaries and the phone list file, edit the file **etc/sphinx_train.cfg** in **tutorial/$taskname/** to change the following training parameters.

    - **$CFG_DICTIONARY** = your training dictionary with full path (do not change if you have decided not to change the dictionary)
    - **$CFG_FILLERDICT** = your filler dictionary with full path (do not change if you have decided not to change the dictionary)
    - **$CFG_RAWPHONEFILE** = your phone list with full path (do not change if you have decided not to change the dictionary)
    - **$CFG_HMM_TYPE** = this variable could have the values **.semi.** or **.cont.**. Notice the dots "." surrounding the string.

Use **.semi.** if you are training semi-continuous HMMs (required for SPHINX-2), or **.cont.** if you are training continuous HMMs (required for SPHINX-4, and the most common choice for SPHINX-3 and SPHINX-3 Flat decoder)

- **$CFG_STATESPERHMM** = if you are using SPHINX-2, this variable has to be 5. If you are using any other decoder, it could be any integer, but we recommend 3 or 5. The number of states in an HMMs is related to the time-varying characteristics of the sound units. Sound units which are highly time-varying need more states to represent them. The time-varying nature of the sounds is also partly captured by the **$CFG_SKIPSTATE** variable that is described below.

- **$CFG_SKIPSTATE** =set this to **no** or **yes**. This variable controls the topology of your HMMs. When set to **yes**, it allows the HMMs to skip states. However, note that the HMM topology used in this system is a strict left-to-right Bakis topology. If you set this variable to **no**, any given state can only transition to the next state. In all cases, self transitions are allowed. See the figures inAppendix 2 for further reference. You will find the HMM topology file, conveniently named **$taskname.topology**, in the directory called **model_architecture/** in your current base directory (**$taskname**).

- **$CFG_FINAL_NUM_DENSITIES** = if you are using sphinx-2, set this number, as well as **$CFG_INITIAL_NUM_DENSITIES**, to 256. If you are using other decoders, set**$CFG_INITIAL_NUM_DENSITIES** to 1 and **$CFG_FINAL_NUM_DENSITIES** to any number from 1 to 8. Going beyond 8 is not advised because of the small training data set you have been provided with. The distribution of each state of each HMM is modeled by a mixture of Gaussians. This variable determines the number of Gaussians in this mixture. The number of HMM parameters to be estimated increases as the number of Gaussians in the mixture increases. Therefore, increasing the value of this variable may result in less data being available to estimate the parameters of every Gaussian. However, increasing its value also results in finer models, which can lead to better recognition. Therefore, it is necessary at this point to think judiciously about the value of this variable, keeping both these issues in mind. Remember that it is possible to overcome data insufficiency problems by sharing the Gaussian mixtures amongst many HMM states. When multiple HMM states share the same Gaussian mixture, they are said to be shared or tied. These shared states are called tied states (also referred to as senones). The number of mixtures you train will ultimately be exactly equal to the number of tied states you specify, which in turn can be controlled by the **$CFG_N_TIED_STATES** parameter described below. SPHINX-2 internally requires you to set the variables to 256, since it uses semi-continuous HMMs.

- **$CFG_N_TIED_STATES** = set this number to any value between 500 and 2500. This variable allows you to specify the total number of shared state distributions in your final set of trained HMMs (your acoustic models). States are shared to overcome problems of data insufficiency for any state of any HMM. The sharing is done in such a way as to preserve the "individuality" of each HMM, in that only the states with the most similar distributions are tied. The **$CFG_N_TIED_STATES** parameter controls the degree of tying. If it is small, a larger number of possibly dissimilar states may be tied, causing reduction in recognition performance. On the other hand, if this parameter is too large, there may be insufficient data to learn the parameters of the Gaussian mixtures for all tied states. (An explanation of state tying is provided in Appendix 3). If you are curious, you can see which states the system has tied for you by looking at the ASCII file**$taskname/model_architecture/$taskname.$CFG_N_TIED_STATES.mdef** and comparing it with the file **$taskname/model_architecture/$taskname.untied.mdef**. These files list the phones and triphones for which you are training models, and assign numerical identifiers to each state of their HMMs.

- **$CFG_CONVERGENCE_RATIO** = set this to a number between 0.1 to 0.001. This number is the ratio of the difference in likelihood between the current and the previous iteration of Baum-Welch to the total likelihood in the previous iteration. Note here that the rate of convergence is dependent on several factors such as initialization, the total number of parameters being estimated, the total amount of training data, and the inherent variability in the characteristics of the training data. The more iterations of Baum-Welch you run, the better you will learn the distributions of your data. However, the minor changes that are obtained at higher iterations of the Baum-Welch algorithm may not affect the performance of the system. Keeping this in mind, decide on how many iterations you want your Baum-Welch training to run in each stage. This is a subjective decision which has to be made based on the first convergence ratio which you will find written at the end of the log file for the second iteration of your Baum-Welch training (**$taskname/logdir/0*/$taskname.*.2.norm.log**. Usually, 5-15 iterations are enough, depending on the amount of data you have. Do not train beyond 15 iterations. Since the amount of training data is not large you will over-train the models to the training data.

- **$CFG_NITER** = set this to an integer number between 5 to 15. This limits the number of iterations of Baum-Welch to the value of **$CFG_NITER**.

Once you have made all the changes desired, you must train a new set of models. You can accomplish this by re-running all the **slave*.pl** scripts from the directories **$taskname/scripts_pl/00***through **$taskname/scripts_pl/09***, or simply by running **perl scripts_pl/RunAll.pl**.

How to decode, and key decoding issues

1. The first step in decoding is to compute the MFCC features for your test utterances. Since you have already done this in the preliminary run, you do not have to repeat the process here.
2. You may change decoder parameters, affecting the recognition results, by editing the file **etc/sphinx_decode.cfg** in **tutorial/$taskname/**. Some of the interesting parameters follow.

- **$DEC_CFG_DICTIONARY** = the dictionary used by the decoder. It may or may not be the same as the one used for training. The set of phones has be be contained in the set of phones from the trainer dictionary. The set of words can be larger. Normally, though, the decoder dictionary is the same as the trainer one, especially for small databases.

- **$DEC_CFG_FILLERDICT** = the filler dictionary.
- **$DEC_CFG_GAUSSIANS** = the number of densities in the model used by the decoder. If you trained continuous models, the process of training creates intermediate models where the number of Gaussians is 1, 2, 4, 8, etc, up to the total number you chose. You can use any of those in the decoder. In fact, you are encouraged to do so, so you get a sense of how this affects the recognition accuracy. You are encouraged to find the best number of densities for databases with different complexities.
- **$DEC_CFG_MODEL_NAME** = the model name. Unless you are using SPHINX-2, it defaults to using the context dependent (CD) tied state models with the number of senones and number of densities specified in the training step. You are encouraged to also use the CD untied and also the context independent (CI) models to get a sense to how accuracy changes.
- **$DEC_CFG_LANGUAGEWEIGHT** the language weight. A value between 6 and 13 is recommended. The default depends on the database that you downloaded. The language model and the language weight are described in Appendix 4. Remember that the language weight decides how much relative importance you will give to the actual acoustic probabilities of the words in the hypothesis. A low language weight gives more leeway for words with high acoustic probabilities to be hypothesized, at the risk of hypothesizing spurious words.
- **$DEC_CFG_ALIGN** = the path to the program that performs word alignment, or **builtin**, if you do not have one.

You may decode several times with changing the variables above without re-training the acoustic models, to decide what is best for you.

3. The script **scripts_pl/decode/slave.pl** already computes the word or sentence accuracy when it finishes decoding. It will add a line to the top level **.html** page that looks like the following if you are using NIST's **sclite**.

```
4. SENTENCE ERROR: 38.833% (233/600)   WORD ERROR RATE: 7.640%
(434/5681)
```

In this line the first percentage indicates the percentage of words in the test set that were correctly recognized. However, this is not a sufficient metric - it is possible to correctly hypothesize all the words in the test utterances merely by hypothesizing a large number of words for each word in the test set. The spurious words, called insertions, must also be penalized when measuring the performance of the system. The second percentage indicates the number of hypothesized words that were erroneous as a percentage of the actual number of words in the test set. This includes both words that were wrongly hypothesized (or deleted) and words that were spuriously inserted. Since the recognizer can, in principle, hypothesize many more spurious words than there are words in the test set, the percentage of errors can actually be greater than 100.

In the example above, using **rm1**, of the 5681 words in the reference test transcripts 5247 words (92.36%) were correctly hypothesized. In the process the recognizer hypothesized 434 spurious words (these include insertions, deletions and substitutions). You will find your recognition hypotheses in files called ***.match** in the directory **$taskname/result/**.

In the same directory, you will also generate files named **$taskname/result/*.align** in which your hypotheses are aligned against the reference sentences. You can study this file to examine the errors that were made. The list of confusions at the end of this file allows you to subjectively determine why particular errors were made by the recognizer. For example, if the word "FOR" has been hypothesized as the word "FOUR" almost all the time, perhaps you need to correct the pronunciation for the word FOR in your decoding dictionary and include a pronunciation that maps the word FOR to the units used in the mapping of the word FOUR. Once you make these corrections, you must re-decode.

If you are using the built-in method, the line reporting accuracy will look like the following if you used **an4**.

```
SENTENCE ERROR: 56.154% (73/130)
```

The meaning of numbers is parallel to the description above, but in this case, the numbers refer to sentences, not to words.

Appendix IV: Moses tutorial

This manual has been performed at the University of Edinburgh [31].

Moses Installation and Training Run-Through

The purpose of this guide is to offer a step-by-step example of downloading, compiling, and runing the Moses decoder and related support tools. We make no claims that all of the steps here will work perfectly on every machine you try it on, or that things will stay the same as the software changes. Please remember that Moses is research software under active development.

PART I - Download and Configure Tools and Data

Support Tools Background

Moses has a number of scripts designed to aid training, and they rely on GIZA++ and mkcls to function. More information on the origins of these tools is available at:

- http://www.fjoch.com/GIZA++.html
- http://www.fjoch.com/mkcls.html

A Google Code project has been set up, and the code is being maintained:

- http://giza-pp.googlecode.com/

Moses uses SRILM-style language models. SRILM is available from:

- http://www.speech.sri.com/projects/srilm/download.html

(Optional) The IRSTLM tools provide the ability to use quantized and disk memory-mapped language models. It's optional, but we'll be using it in this tutorial:

- http://sourceforge.net/projects/irstlm

Support Tools Installation

Before we start building and using the Moses codebase, we have to download and compile all of these tools. See the list of versions to double-check that you are using the same code.

I'll be working under /home/jschroe1/demo in these examples. I assume you've set up some appropriately named directory in your own system. I'm installing these tools under an FC6 distro.

```
mkdir tools
cd tools
```

- Download and compile `GIZA++` and `mkcls`

```
•   wget http://giza-pp.googlecode.com/files/giza-pp-v1.0.2.tar.gz
•   tar -xzvf giza-pp-v1.0.2.tar.gz
•   cd giza-pp
```

```
make
```

- Copy compiled executables to `bin/` folder

```
•   cd ../
•   mkdir bin
•   cp giza-pp/GIZA++-v2/GIZA++ bin/
•   cp giza-pp/mkcls-v2/mkcls bin/
•   cp giza-pp/GIZA++-v2/snt2cooc.out bin/
```

- Download and compile SRILM

SRILM has a lot of dependencies. These instructions work on `bash`.

```
mkdir srilm
cd srilm
```

(get srilm download 1.5.7, requires web registration, you'll end up with a .tgz file to copy to this directory)

```
tar -xzvf srilm.tgz
```

(SRILM expands in the current directory, not in a sub-directory).

READ THE INSTALL FILE - there are a lot of tips in there.

```
chmod +w Makefile
```

edit Makefile to point to your directory. Here's my diff:

```
7c7
< # SRILM = /home/speech/stolcke/project/srilm/devel
---
> SRILM = /home/jschroe1/demo/tools/srilm

make World
```

If you want to test that this worked, you'll need to add SRILM to your path and run their test suite. You don't need these in your path for normal training and decoding with Moses.

```
export
PATH=/home/jschroe1/demo/tools/srilm/bin/i686:/home/jschroe1/dem
o/tools/srilm/bin:$PATH
make all
```

Check output, look for `IDENTICAL` and `DIFFERS`. I still see the occasional difference, but it's pretty easy to tell when the tools are working and when they're dying instantly.

- Download and compile IRSTLM

You can either download a release or check out the latest files from svn.

```
cd /home/jschroe1/demo/tools
wget http://downloads.sourceforge.net/irstlm/irstlm-5.20.00.tgz
tar -xzvf irstlm-5.20.00.tgz
```

Or get it from sourceforge:

```
mkdir irstlm
svn co https://irstlm.svn.sourceforge.net/svnroot/irstlm irstlm

cd irstlm
./install
```

On my system, Moses looks in `irstlm/bin/i686`, and IRST compiles to `irstlm/bin/i686-redhat-linux-gnu`. Symlink to fix.

```
cd bin
ln -s i686-redhat-linux-gnu i686
cd ../../
```

Get The Latest Moses Version

Moses is available via Subversion from Sourceforge. See the list of versions to double-check that you are using the same code as this example. From the `tools/` directory:

```
mkdir moses
svn co
https://mosesdecoder.svn.sourceforge.net/svnroot/mosesdecoder/trunk
moses
```

This will copy all of the Moses source code to your local machine.

Compile Moses

Within the Moses folder structure are projects for Eclipse, Xcode, and Visual Studio -- though these are not well maintained and may not be up to date. I'll focus on the linux command-line method, which is the preferred way to compile.

```
cd moses
./regenerate-makefiles.sh
./configure --with-srilm=/home/jschroe1/demo/tools/srilm --with-
irstlm=/home/jschroe1/demo/tools/irstlm
make -j 2
```

(The `-j 2` is optional. `make -j X` where X is number of simultaneous tasks is a speedier option for machines with multiple processors)

This creates several files we will be using:

- `misc/processPhraseTable` - Used to binarize phrase tables
- `misc/processLexicalTable` - Used to binarize reordering tables
- `moses-cmd/src/moses` - The actual decoder

Confirm Setup Success

A sample model capable of translating one sentence is available on the Moses website. Download it and translate the sample input file.

```
cd /home/jschroe1/demo/
mkdir data
cd data
wget http://www.statmt.org/moses/download/sample-models.tgz
tar -xzvf sample-models.tgz
cd sample-models/phrase-model/
../../../tools/moses/moses-cmd/src/moses -f moses.ini < in > out
```

The input has "das ist ein kleines haus" listed twice, so the output file (`out`) should contain "this is a small house" twice.

At this point, it might be wise for you to experiment with the command line options of the Moses decoder. A tutoral using this example model is available at http://www.statmt.org/moses/?n=Moses.Tutorial.

Compile Moses Support Scripts

Moses uses a set of scripts to support training, tuning, and other tasks. The support scripts used by Moses are "released" by a Makefile which edits their paths to match your local environment. First, make a place for the scripts to live:

```
cd ../../../tools/
mkdir moses-scripts
cd moses/scripts
```

edit Makefile as needed. Here's my diff:

```
13,14c13,14
< TARGETDIR?=/home/s0565741/terabyte/bin
< BINDIR?=/home/s0565741/terabyte/bin
---
> TARGETDIR?=/home/jschroe1/demo/tools/moses-scripts
> BINDIR?=/home/jschroe1/demo/tools/bin

make release
```

This will create a time-stamped folder named `/home/jschroe1/demo/moses-scripts/scripts-`*`YYYYMMDD-HHMM`* with released versions of all the scripts. You will call these versions when training and tuning Moses. Some Moses training scripts also require a `SCRIPTS_ROOTDIR` environment variable to be set. The output of `make release` should indicate this. Most scripts allow you to override this by setting a `-scripts-root-dir` flag or something similar.

```
export SCRIPTS_ROOTDIR=/home/username/lab4/moses-scripts/scripts-
YYYYMMDD-HHMM
```

Additional Scripts

There are few scripts not included with moses which are useful for preparing data. These were originally made available as part of the WMT08 Shared Task and Europarl v3 releases, I've consolidated some of them into one set.

```
cd ../../
```

```
wget http://homepages.inf.ed.ac.uk/jschroe1/how-to/scripts.tgz
tar -xzvf scripts.tgz
```

We'll also get a NIST scoring tool.

```
wget ftp://jaguar.ncsl.nist.gov/mt/resources/mteval-v11b.pl
chmod +x mteval-v11b.pl
```

PART II - Build a Model

We'll used the WMT08 News Commentary data set, about 55k sentences. This should be good enough for moderate quality but still be doable in a reasonable amount of time on most machines. For this example we'll use FR-EN.

```
cd ../data
wget http://www.statmt.org/wmt08/training-parallel.tar
tar -xvf training-parallel.tar --wildcards training/news-
commentary08.fr-en.*
```

If you're low on disk space, remove the full tar.

```
rm training-parallel.tar

cd ../
```

Prepare Data

First we'll set up a working directory where we'll store all the data we prepare.

```
mkdir work
```

- ***Tokenize training data***

We'll keep the initial versions in zipped format. Note that Mac uses `gzcat` instead of `zcat`, so we'll just use `gzip -cd` for both.

```
mkdir work/corpus
gzip -cd data/training/news-commentary08.fr-en.fr.gz |
tools/scripts/tokenizer.perl -l fr > work/corpus/news-
commentary.tok.fr
gzip -cd data/training/news-commentary08.fr-en.en.gz |
tools/scripts/tokenizer.perl -l en > work/corpus/news-
commentary.tok.en
```

- ***Filter out long sentences***

```
•   tools/moses-scripts/scripts-YYYYMMDD-HHMM/training/clean-
    corpus-n.perl work/corpus/news-commentary.tok fr en
    work/corpus/news-commentary.clean 1 40
```

This ensures that only sentences of length 1-40 are selected for training. In this case, we lose almost 11,000 sentences:

```
Input sentences: 55030  Output sentences:  44219
```

We do this because GIZA++ takes a very long time to train on long sentences. This isn't much of an issue with a 55,000-sentence corpus, but it can be a limitation when dealing with corpora of millions of sentences. Of course, the more data you throw out to improve training times, the less examples Moses can choose from when building translations.

- ***Lowercase training data***

```
•   tools/scripts/lowercase.perl < work/corpus/news-
    commentary.clean.fr > work/corpus/news-commentary.lowercased.fr
•   tools/scripts/lowercase.perl < work/corpus/news-
    commentary.clean.en > work/corpus/news-commentary.lowercased.en
```

Build Language Model

Language models are concerned only with n-grams in the data, so sentence length doesn't impact training times as it does in GIZA++. So, we'll lowercase the full 55,030 tokenized sentences to use for language modeling. Many people incorporate extra target language monolingual data into their language models.

```
mkdir work/lm
tools/scripts/lowercase.perl < work/corpus/news-commentary.tok.en >
work/lm/news-commentary.lowercased.en
```

We will use SRILM to build a tri-gram language model.

```
tools/srilm/bin/i686/ngram-count -order 3 -interpolate -kndiscount -
unk -text work/lm/news-commentary.lowercased.en -lm work/lm/news-
commentary.lm
```

We can see how many n-grams were created

```
head -n 5 work/lm/news-commentary.lm


\data\
ngram 1=36035
ngram 2=411595
ngram 3=118368
```

Train Phrase Model

Moses' toolkit does a great job of wrapping up calls to `mkcls` and `GIZA++` inside a training script, and outputting the phrase and reordering tables needed for decoding. The script that does this is called train-factored-phrase-model.perl

We'll run this in the background and nice it since it'll peg the CPU while it runs. It may take up to an hour, so this might be a good time to run through the tutorial page mentioned earlier using the `sample-models` data.

```
nohup nice tools/moses-scripts/scripts-YYYYMMDD-HHMM/training/train-
factored-phrase-model.perl -scripts-root-dir tools/moses-
scripts/scripts-YYYYMMDD-HHMM/ -root-dir work -corpus
work/corpus/news-commentary.lowercased -f fr -e en -alignment grow-
diag-final-and -reordering msd-bidirectional-fe -lm
0:3:/home/jschroe1/demo/work/lm/news-commentary.lm >&
work/training.out &
```

You can `tail -f work/training.out` file to watch the progress of the tuning script. The last step will say something like:

```
(9) create moses.ini @ Tue Jan 27 19:40:46 CET 2009
```

Now would be a good time to look at what we've done.

```
cd work
ls
corpus  giza.en-fr  giza.fr-en  lm  model
```

We'll look in the model directory. The three files we really care about are in bold.

```
cd model
ls -l
total 192554
-rw-r--r-- 1 jschroe1 people  5021309 Jan 27 19:23 aligned.grow-diag-
final-and
-rw-r--r-- 1 jschroe1 people 27310991 Jan 27 19:24 extract.gz
-rw-r--r-- 1 jschroe1 people 27043024 Jan 27 19:25 extract.inv.gz
-rw-r--r-- 1 jschroe1 people 21069284 Jan 27 19:25 extract.o.gz
-rw-r--r-- 1 jschroe1 people  6061767 Jan 27 19:23 lex.e2f
-rw-r--r-- 1 jschroe1 people  6061767 Jan 27 19:23 lex.f2e
-rw-r--r-- 1 jschroe1 people     1032 Jan 27 19:40 moses.ini
-rw-r--r-- 1 jschroe1 people 67333222 Jan 27 19:40 phrase-table.gz
-rw-r--r-- 1 jschroe1 people 26144298 Jan 27 19:40 reordering-table.gz
```

Memory-Map LM and Phrase Table (Recommended for large data sets or computers with minimal RAM)

The language model and phrase table can be memory-mapped on disk to minimize the amount of RAM they consume. This isn't really necessary for this size of model, but we'll do it just for the experience.

If Moses segfaults when you try using a larger model than the one in this example, then you should try this step for sure.

More information is available on the Moses' web site at: http://www.statmt.org/moses/?n=Moses.AdvancedFeatures and http://www.statmt.org/moses/?n=FactoredTraining.BuildingLanguageModel.

Performing these steps can lead to heavy disk use during decoding - you're basically using your hard drive as RAM. Proceed at your own risk, especially if you're using a (slow) networked drive.

- ***IRSTLM Binary Language Model***

Produces a compact file on disk

```
cd ../../
tools/irstlm/bin/i686/compile-lm work/lm/news-commentary.lm
work/lm/news-commentary.blm
```

- ***IRSTLM Memory Mapping***

Changing the suffix of this file to `.mm` forces the decoder to leave the file on disk instead of loading it into memory. We'll just make a symlink.

```
cd work/lm
ln -s news-commentary.blm news-commentary.blm.mm
cd ../../
```

A note on memory mapping: IRSTLM makes use of a temp directory during decoding. Version 5.20.00 has this hard-coded to `/tmp`, but the trunk on svn has been updated to allow you to set it using the `TMP` environment variable. If this is important to your setup, be sure to set this variable, or check that it is already set appropriately.

- ***Binary Phrase Table***

As with the LM, the phrase table can be processed and read from disk on-demand instead of being loaded in its entirety into memory.

Note that if your phrase table was not sorted, you would need to pipe the zcat through a `sort`, and use the `LC_ALL=C` flag. Depending on the size of your temp directory, you may have to have sort use a different directory using the `-T` flag. `man sort` for more info.

```
gzip -cd work/model/phrase-table.gz | LC_ALL=C sort |
tools/moses/misc/processPhraseTable -ttable 0 0 - -nscores 5 -
out work/model/phrase-table
```

- ***Binary Reordering Table***

  Similar to the phrase table, including optional sorting.

```
gzip -cd work/model/reordering-table.gz | LC_ALL=C sort |
tools/moses/misc/processLexicalTable -out work/model/reordering-
table
```

- ***Edit Config File***

  We'll make a copy of `work/model/moses.ini` and set it to use these files. Moses will automatically use binary phrase and reordering tables if they are present with the correct naming stem, and since we used the same stem for output as for our input tables, we just need to remove the `.gz` suffix. For LM information, we need to set the type to be IRSTLM (1) instead of SRILM (0) and change the LM file.

```
cp work/model/moses.ini work/model/moses-bin.ini
```

Here's my diff:

```
15c15
< 0 0 5 /home/jschroe1/demo/work/model/phrase-table.gz
---
> 0 0 5 /home/jschroe1/demo/work/model/phrase-table
21c21
< 0 0 3 /home/jschroe1/demo/work/lm/news-commentary.lm
---
> 1 0 3 /home/jschroe1/demo/work/lm/news-commentary.blm.mm
31c31
< 0-0 msd-bidirectional-fe 6
/home/jschroe1/demo/work/model/reordering-table.gz
---
> 0-0 msd-bidirectional-fe 6
/home/jschroe1/demo/work/model/reordering-table
```

Sanity Check Trained Model

We haven't tuned yet, but let's just check that the decoder works, and output a lot of logging data with `-v 2`.

Here's an excerpt of moses initializing with binary files in place (note bold lines, and recall the IRSTLM `TMP` issue):

```
echo "c' est une petite maison ." | TMP=/tmp tools/moses/moses-
cmd/src/moses -f work/model/moses-bin.ini
Loading lexical distortion models...
have 1 models
Creating lexical reordering...
weights: 0.300 0.300 0.300 0.300 0.300 0.300
binary file loaded, default OFF_T: -1
Created lexical orientation reordering
Start loading LanguageModel /home/jschroe1/demo/work/lm/news-
commentary.blm.mm : [0.000] seconds
In LanguageModelIRST::Load: nGramOrder = 3
Loading LM file (no MAP)
blmt
loadbin()
mapping 36035 1-grams
mapping 411595 2-grams
mapping 118368 3-grams
done
OOV code is 1468
IRST: m_unknownId=1468
Finished loading LanguageModels : [0.000] seconds
Start loading PhraseTable
/amd/nethome/jschroe1/demo/work/model/phrase-table.0-0 : [0.000]
seconds
using binary phrase tables for idx 0
reading bin ttable
size of OFF_T 8
binary phrasefile loaded, default OFF_T: -1
Finished loading phrase tables : [1.000] seconds
IO from STDOUT/STDIN
```

And here's one if you skipped the memory mapping steps:

```
echo "c' est une petite maison ." | tools/moses/moses-cmd/src/moses -f
work/model/moses.ini
Loading lexical distortion models...
have 1 models
Creating lexical reordering...
weights: 0.300 0.300 0.300 0.300 0.300 0.300
Loading table into memory...done.
Created lexical orientation reordering
Start loading LanguageModel /home/jschroe1/demo/work/lm/news-
commentary.lm : [47.000] seconds
/home/jschroe1/demo/work/lm/news-commentary.lm: line 1476: warning:
non-zero probability for <unk> in closed-vocabulary LM
Finished loading LanguageModels : [49.000] seconds
Start loading PhraseTable
/amd/nethome/jschroe1/demo/work/model/phrase-table.0-0.gz : [49.000]
seconds
```

```
Finished loading phrase tables : [259.000] seconds
IO from STDOUT/STDIN
```

Again, while these short load times and small memory footprint are nice, decoding times will be slower with memory-mapped models due to disk access.

PART III - Prepare Tuning and Test Sets

Prepare Data

We'll use some of the dev and devtest data from WMT08. We'll stick with news-commentary data and use dev2007 and test2007. We only need to look at the input (FR) side of our testing data.

- ***Download tuning and test sets***

```
•   cd data/
•   wget http://www.statmt.org/wmt08/devsets.tgz
•   tar -xzvf devsets.tgz
•   cd ../
```

- ***Tokenize sets***

```
•   mkdir work/tuning
•   tools/scripts/tokenizer.perl -l fr < data/dev/nc-dev2007.fr >
  work/tuning/nc-dev2007.tok.fr
•   tools/scripts/tokenizer.perl -l en < data/dev/nc-dev2007.en >
  work/tuning/nc-dev2007.tok.en
•   mkdir work/evaluation
•   tools/scripts/tokenizer.perl -l fr < data/devtest/nc-
  test2007.fr > work/evaluation/nc-test2007.tok.fr
```

- ***Lowercase sets***

```
•   tools/scripts/lowercase.perl < work/tuning/nc-dev2007.tok.fr >
  work/tuning/nc-dev2007.lowercased.fr
•   tools/scripts/lowercase.perl < work/tuning/nc-dev2007.tok.en >
  work/tuning/nc-dev2007.lowercased.en
•   tools/scripts/lowercase.perl < work/evaluation/nc-
  test2007.tok.fr > work/evaluation/nc-test2007.lowercased.fr
```

PART IV - Tuning

Note that this step can take many hours, even days, to run on large phrase tables and tuning sets. We'll use the non-memory-mapped versions for decoding speed. The training script controls for large phrase and reordering tables by filtering them to include only data relevant to the tuning set (we'll do this ourselves for the test data later).

```
nohup nice tools/moses-scripts/scripts-YYYYMMDD-HHMM/training/mert-
moses.pl work/tuning/nc-dev2007.lowercased.fr work/tuning/nc-
dev2007.lowercased.en tools/moses/moses-cmd/src/moses
work/model/moses.ini --working-dir work/tuning/mert --rootdir
/home/jschroe1/demo/tools/moses-scripts/scripts-YYYYMMDD-HHMM/ --
decoder-flags "-v 0" >& work/tuning/mert.out &
```

Since this can take so long, we can instead make a small, 100 sentence tuning set just to see if the tuning process works. This won't generate very good weights, but it will let us confirm that our tools work.

```
head -n 100 work/tuning/nc-dev2007.lowercased.fr > work/tuning/nc-
dev2007.lowercased.100.fr
head -n 100 work/tuning/nc-dev2007.lowercased.en > work/tuning/nc-
dev2007.lowercased.100.en
nohup nice tools/moses-scripts/scripts-YYYYMMDD-HHMM/training/mert-
moses.pl work/tuning/nc-dev2007.lowercased.100.fr work/tuning/nc-
dev2007.lowercased.100.en tools/moses/moses-cmd/src/moses
work/model/moses.ini --working-dir work/tuning/mert --rootdir
/home/jschroe1/demo/tools/moses-scripts/scripts-YYYYMMDD-HHMM/ --
decoder-flags "-v 0" >& work/tuning/mert.out &
```

(Note that the scripts rootdir path needs to be absolute).

While this runs, check out the contents of `work/tuning/mert`. You'll see a set of runs, n-best lists for each, and `run*.moses.ini` files showing the weights used for each file. You can see the score each run is getting by looking at the last line of each `run*.cmert.log` file

```
cd work/tuning/mert
tail -n 1 run*.cmert.log

==> run1.cmert.log <==
Best point: 0.028996 0.035146 -0.661477 -0.051250 0.001667 0.056762
0.009458 0.005504 -0.006458 0.029992 0.009502 0.012555 0.000000 -
0.091232 => 0.282865

==> run2.cmert.log <==
Best point: 0.056874 0.039994 0.046105 -0.075984 0.032895 0.020815 -
0.412496 0.018823 -0.019820 0.038267 0.046375 0.011876 -0.012047 -
0.167628 => 0.281207

==> run3.cmert.log <==
Best point: 0.041904 0.030602 -0.252096 -0.071206 0.012997 0.516962
0.001084 0.010466 0.001683 0.008451 0.001386 0.007512 -0.014841 -
0.028811 => 0.280953

==> run4.cmert.log <==
Best point: 0.088423 0.118561 0.073049 0.060186 0.043942 0.293692 -
0.147511 0.037605 0.008851 0.019371 0.015986 0.018539 0.001918 -
0.072367 => 0.280063

==> run5.cmert.log <==
Best point: 0.059100 0.049655 0.187688 0.010163 0.054140 0.077241
0.000584 0.101203 0.014712 0.144193 0.219264 -0.005517 -0.047385 -
0.029156 => 0.280930
```

This gives you an idea if the system is improving or not. You can see that in this case it isn't, because we don't have enough data in our system and we haven't let tuning run for enough iterations. Kill `mert-moses.pl` after a few iterations just to get some weights to use.

If mert were to finish successfully, it would create a file named `work/tuning/mert/moses.ini` containing all the weights we needed. Since we killed mert, copy the best moses.ini config to be the one we'll use. Note that the weights calculated in `run1.cmert.log` were used to make the config file for run2, so we want `run2.moses.ini`

```
cp run2.moses.ini moses.ini
```

Insert weights into configuration file

```
cd ../../../
tools/scripts/reuse-weights.perl work/tuning/mert/moses.ini <
work/model/moses.ini > work/tuning/moses-tuned.ini
tools/scripts/reuse-weights.perl work/tuning/mert/moses.ini <
work/model/moses-bin.ini > work/tuning/moses-tuned-bin.ini
```

PART V - Filtering Test Data

Filtering is another way, like binarizing, to help reduce memory requirements. It makes smaller phrase and reordering tables that contain only entries that will be used for a particular test set. Binarized models don't need to be filtered since they don't take up RAM when used. Moses has a script that does this for us, which we'll apply to the evaluation test set we prepared earlier:

```
tools/moses-scripts/scripts-YYYYMMDD-HHMM/training/filter-model-given-
input.pl  work/evaluation/filtered.nc-test2007 work/tuning/moses-
tuned.ini work/evaluation/nc-test2007.lowercased.fr
```

There is also a `filter-and-binarize-model-given-input.pl` script if your filtered table would still be too large to load into memory.

PART VI - Run Tuned Decoder on Development Test Set

We'll try this a few ways.

- First, reusing the weights from tuning, without filtering:

```
•   nohup nice tools/moses/moses-cmd/src/moses -config
    work/tuning/moses-tuned.ini -input-file work/evaluation/nc-
    test2007.lowercased.fr 1> work/evaluation/nc-
    test2007.tuned.output 2> work/evaluation/tuned.decode.out &
```

- Next, with the filtered phrase table from the output of the filtering step:

```
•   nohup nice tools/moses/moses-cmd/src/moses -config
    work/evaluation/filtered.nc-test2007/moses.ini -input-file
    work/evaluation/nc-test2007.lowercased.fr 1> work/evaluation/nc-
    test2007.tuned-filtered.output 2> work/evaluation/tuned-
    filtered.decode.out &
```

- Finally, if you performed binarizing, you can try that too:

```
•   TMP=/tmp nohup nice tools/moses/moses-cmd/src/moses -config
    work/tuning/moses-tuned-bin.ini -input-file work/evaluation/nc-
    test2007.lowercased.fr 1> work/evaluation/nc-test2007.tuned-
    bin.output 2> work/evaluation/tuned-bin.decode.out &
```

All three of these outputs should be identical, but they will take different amounts of time and memory to compute.

PART VII - Evaluation

Train Recaser

Now we'll train a recaser. It uses a statistical model to "translate" between lowercased and cased data.

```
mkdir work/recaser
tools/moses-scripts/scripts-YYYYMMDD-HHMM/recaser/train-recaser.perl -
train-script tools/moses-scripts/scripts-YYYYMMDD-HHMM/training/train-
factored-phrase-model.perl -ngram-count tools/srilm/bin/i686/ngram-
count -corpus work/corpus/news-commentary.tok.en -dir
/home/jschroe1/demo/work/recaser -scripts-root-dir tools/moses-
scripts/scripts-YYYYMMDD-HHMM/
```

This goes through a whole GIZA and LM training run to go from lowercase sentences to cased sentences. Note that the `-dir` flag needs to be absolute.

Recase the output

```
tools/moses-scripts/scripts-YYYYMMDD-HHMM/recaser/recase.perl -model
work/recaser/moses.ini -in work/evaluation/nc-test2007.tuned-
filtered.output -moses tools/moses/moses-cmd/src/moses >
work/evaluation/nc-test2007.tuned-filtered.output.recased
```

Detokenize the output

```
tools/scripts/detokenizer.perl -l en < work/evaluation/nc-
test2007.tuned-filtered.output.recased > work/evaluation/nc-
test2007.tuned-filtered.output.detokenized
```

Appendix V: Sclite manual

This manual has been performed at the International Computer Science Institute[5]

NAME

sclite - score speech recognition system output

SYNOPSIS

sclite -r reffile [fmt] -h hypfile [fmt [title]] OPTIONS

DESCRIPTION

The program sclite is a tool for scoring and evaluating the output of speech recognition systems. Sclite is part of the NIST SCTK Scoring Tookit. The program compares the hypothesized text (HYP) output by the speech recognizer to the correct, or reference (REF) text. After comparing REF to HYP, (a process called alignment), statistics are gathered during the scoring process and a variety of reports can be produced to summarize the performance of the recognition system.

THE ALIGNMENT PROCESS

The Alignment process consists of two steps: 1) selecting matching REF and HYP texts, and 2) performing an alignment of the reference and hypothesis texts.

Step 1: Selection of matching REF and HYP texts

Sclite accepts as input a wide variety of file formats. The type of input formats define the algorithm for selecting matching REF and HYP texts. Currently sclite uses four algorithms:

Utterance ID Matching:

Input reference and hypothesis files in "trn" transcript format can be aligned by either dynamic programming (DP) or GNU's "diff".

When alignments are performed via DP, corresponding REF and HYP records with the same utterance id's are located in the REF and HYP files. DP Alignment and scoring are then performed on each pair of records. Only the utterance ID's present in the HYP file are aligned and scored. This means the REF file may contain more utterance records than the HYP.

When "diff" is used for alignment, corresponding REF and HYP records with the same utterance id's are located in the REF and HYP files. Rather than execute "diff" for each pair of records, all matching REF and HYP pairs are re-formatted to be newline separated words and written to a temporary files. Using the two temporary files, "diff" is then called to perform a global alignment. The output of "diff" is re-chunking into REF/HYP records by applying the rule: include all words in the output stream up to and including the last word in the reference record.

The reference file can contain extra transcripts, only needed transcripts are loaded.

Word Time Mark Matching:

When both the REF and HYP files are in the "ctm" format, The first step in the alignment process is to segment both the reference and hypothesis word lists by locating common areas of silence, (i.e. the absence of a word time mark). Once completed, the resulting "segments" are aligned via dynamic programming and scored as usual.

By default, the DP alignment is performed using word-to-word distances measures of: 0, 3, 3, 4 for correct, insertions, deletions and substitutions respectively.

Optionally, the command line flag '-T' forces the alignments to be performed using time-mediated alignments.

Reference Segment Time Mark to Hypothesis Word Time Mark

When the reference file format is "stm" and the hypothesis file format is "ctm", sclite chops up the hypothesis file into regions matching the reference segments. Currently, there a two methods of chopping the hypothesis file. The method is dependent on the text alignment algorithm.

When DP alignments are performed, the hypothesis file is segmented to match the reference segments by selecting the string of hypothesized words whose times occur before the end of each reference segment. The midpoint time of a word is used to determine if the word falls within a segment. DP alignments are then performed on the selected hypothesis words and the reference segment.

If the alignments are performed via "diff", pre-process the input reference and hypothesis texts, creating temporary reference and hypothesis files with one word per line. Then use GNU's "diff" program to perform a global alignment on the word lists. The output of "diff" is re-chunked into segments for scoring. Alternate reference transcripts can not be used with "diff" alignments.

Reference Segment Time Mark to Hypothesis Text file

When the reference file format "stm" and the hypothesis file format "txt" are used as inputs, the same alignment and scoring algorithm is used as describe above under the label "Reference Segment Time Mark to Hypothesis Word Time Mark" by GNU diff alignments.

Step 2: Text Alignments

Sclite can use either of two algorithms for finding alignments between reference and hypothesis word strings. The first, and most widely accepted, uses dynamic programming (DP) and the second uses GNU's "diff", a FSF (Free Software Foundation) program for comparing text files.

Dynamic Programming string alignment:

The DP string alignment algorithm performs a global minimization of a Levenshtein distance function which weights the cost of correct words, insertions, deletions and substitutions as 0, 3, 3 and 4 respectively. The computational complexity of DP is 0(NN).

When evaluating the output of speech recognition systems, the precision of generated statistics is directly correlated to the reference text accuracy. But uttered words can be coarticulated or mumbled to where they have ambiguous transcriptions, (e.i., "what are" or "what're"). In order to more accurately represent ambiguous transcriptions, and not penalize recognition systems, the ARPA community agreed upon a format for specifying alternative reference transcriptions. The convention, when used on the case above, allows the recognition system to output either transcripts, "what are" or "what're", and still be correct.

The case above handles ambiguously spoken words which are loud enough for the transcriber to think something should be recognized. For mumbled or quietly spoken words, the ARPA community agreed to neither penalize systems which correctly recognized the word, nor penalize systems which did not. To accommodate this, a NULL word, "@", can be added to an alternative reference transcript. For example, "the" is often spoken quickly with little acoustic evidence. If "the" and "@" are alternates, the recognition system will be given credit for outputting "the" but not penalized if it does not.

The presence of alternate transcriptions represents added computational complexity to the DP algorithm. Rather than align all alternate reference texts to the hypothesis text, then choose the lowest error rate alignment, this implementation of DP aligns two word networks, thus reducing the computational complexity from 2^(ref_alts + hyp_alts) * O(N_ref * N_hyp) to O((N_ref+ref_alts) * (N_hyp+hyp_alts)).

For a detailed explanation of DP alignment, see TIME WARPS, STRING EDITS, AND MACROMOLECULES: THE THEORY AND PRACTICE OF SEQUENCE COMPARISON, by Sankoff and Kruskal, ISBN 0-201-07809-0.

As noted above, DP alignment minimizes a distance function that is applied to word pairs. In addition to the "word" alignments which uses a distance function defined by static weights, the sclite DP alignment module can use two other distance functions. The first, called **Time-Mediated** alignment and the second called **Word-Weight-Mediated** alignment.

Time-Mediated Alignment

Time-Mediated alignment is a variation of DP alignment where word-to-word distances are based on the time of occurence for individual words. Time-mediated alignments are performed when the '-T' option is exercised and the input formats for both the reference and hypothesis files are in "ctm" format.

Time-mediated alignments are computed by replacing the standard word-to-word distance weights of 0, 3, 3, and 4 with measures based on beginning and ending word times. The formulas for time-mediated word-to-word distances are:

D(correct) = | T1(ref) - T1(hyp) | + | T2(ref) - T2(hyp) |
D(insertion) = T2(hyp) - T1(hyp)

D(deletion) = T2(ref) - T1(ref)
D(substitution) = | T1(ref) - T1(hyp) | + | T2(ref) - T2(hyp) | + 0.001
Distance for an Insertion or Deletion of the NULL Token '@' = 0.001

Where,

T1(x) is the beginning time mark of word x

T2(x) is the ending time mark of word x

Word-Weight-Mediated Alignment

Word-weight-mediated alignment is a variation of DP alignments where word-to-word distances are based on pre-defined word-weights. Each word has a unique weight assigned to it, via either a word-weight-list file, using the -w option, or through a language model file, using the -L option. The formulas for word-weight-mediated word-to-word distances are:

D(correct) = 0.0
D(insertion) = W(hyp)
D(deletion) = W(ref)
D(substitution) = W(hyp) + W(ref)
Distance for and Insertion or Deletion of the NULL Token '@' = 0.001

Where W(x) is the weight assigned to word 'x'.

String alignments via GNU's "diff":

While the DP algorithm has the advantage of flexibility, it is slow for aligning large chunks of text. To address the speed concerns, an alternative string alignment module, which utilizes GNU's "diff", has been added to sclite. The sclite program pre-processes the input reference and hypothesis texts, creating temporary reference and hypothesis files with one word per line. Then GNU's "diff" program is used to perform a global alignment on the word lists and the output is re-chunked into utterances or text segments for scoring.

Alignments can be performed with "diff" in about half the time taken for DP alignments on the standard 300 Utterance ARPA CSRNAB test set. However, in the opinion of the author, "diff" has the following bad effects:

1. it can not accommodate transcription alternations,
2. "diff" does not produce the same alignments as the DP alignments,

3. there is an increase measured error rates.

THE SCORING PROCESS

After reference and hypothesis texts have been aligned, scores are tallied for each speaker and each ref/hyp pair. After the tallies are made, a variety if output reports are generated by using the '-o' option. Here is a set of examples.

The categories tallied are:

$$\text{Percent of correct words} = \frac{\text{\# Correct words}}{\text{\# Reference words}} * 100$$

$$\text{Percent of substituted words} = \frac{\text{\# Substituted words}}{\text{\# Reference words}} * 100$$

$$\text{Percent of inserted words} = \frac{\text{\# Inserted words}}{\text{\# Reference words}} * 100$$

$$\text{Percent of deleted words} = \frac{\text{\# Deleted words}}{\text{\# Reference words}} * 100$$

$$\text{Percent of sentence errors} = \frac{\text{\# incorrect ref and hyp pairs}}{\text{\# ref and hyp pairs}} * 100$$

A variation in scoring called **Weighted-Word Scoring** can also be implemented by sclite. After Word-Weight-Mediated Alignment, the word weights can be tallied to produce weighted-word scores. The formulas for weighted-word scoring are very simliar to word scoring described above. The difference is rather than assume each word has the same weight, 1 in the case of word scoring, each individual word has a different weight. The word scoring formulas become:

$$\text{Weighted Percent of correct words} = \frac{\text{Sum of W(hyp) if correct}}{\text{Sum of W(ref)}} * 100$$

$$\text{Weighted Percent of substituted words} = \frac{\text{Sum of W(hyp) + W(ref) if substituted}}{\text{Sum of W(ref)}} * 100$$

$$\text{Weighted Percent of inserted words} = \frac{\text{Sum of W(hyp) if inserted}}{\text{Sum of W(ref)}} * 100$$

$$\text{Weighted Percent of deleted words} = \frac{\text{Sum of W(ref) if deleted}}{\text{Sum of W(ref)}} * 100$$

W(hyp) is the weight assigned to a hypothesis word, and W(ref) is the weight assigned to a reference word. Optionally deletable words have the default weight of 0.0.

WORD CONFIDENCE MEASURE EVALUATION

Confidence scores for each hypothesized word were requested of the LVCSR (Large Vocabulary Speech Recognition) participants beginning with the April 1996 evaluation. Each site was asked to do its analysis of these scores which were not processed by NIST. A review meeting was held at NIST in August 1996 which resulted in a decision to apply an agreed upon standard metric.

Confidence scores as they have been implemented are associated with each hypothesized word. (The issue has been raised whether for languages such as Mandarin, where character error rate is considered the primary measure of performance, the confidence ought to be associated with characters.) The confidence score p_c, associated with a word must be in the closed interval [0,1] and presumably, given the entropy related metric defined below, in the open interval (0,1). It should represent the system's best estimate of the a posterior probability that the hypothesized word is correct. (Correct here necessarily is with respect to an alignment procedure of the reference and hypothesis word strings.)

A single metric to use in the evaluation of confidence scores was adopted at the August meeting. This is a normalized version of the cross entropy or mutual information. Specifically, the metric os defined as:

$$\text{Confidence\_Score} = \left\{ H_{\max} + \sum_{\text{correct}\,w} \log_2\left(\hat{p}(w)\right) + \sum_{\text{incorrect}\,w} \log_2\left(1 - \hat{p}(w)\right) \right\} / H_{\max},$$

$$\text{where } H_{\max} = -n\log_2\left(p_c\right) - \left(N - n\right)\log_2\left(1 - p_c\right),$$

$$n = \text{the number of correct HYP words,}$$

$$N = \text{the total number of HYP words,}$$

$$p_c = \text{the average probability that an output word is correct} = n / N, \text{ ar}$$

$$\hat{p}(w) = \text{the confidence measure output, as a function of output word.}$$

Sclite will automatically detect the presence of confidence measures when reading in a hypothesis "ctm" file. When sclite detects the confidence scores, the report genererated by the options "-o sum" has an additional column containing the Normalized Cross Entropy (NCE).

Output graphs concerning confidence estimates are generated by using the '-C' option. A variety of graphs can be created:

DET Curve Example
Binned Histogram Example
Word Confidence Score Histogram Example

REVISION HISTORY

See revision.txt in the main directory of the sclite source code directory package.

EXAMPLE USES OF SCLITE

The sclite scoring utility was written to be used as a standard scoring tool for the ARPA speech recognition benchmark tests. Since evaluation paradigms have changed over the past several years, file formats and scoring proceedures have changed as well. This utility supports the following speech recognition benchmark tests:

Utterance based evaluations:

Resource Management
ATIS (Airline Travel Information Systems):

Found speech evaluations:

Hub 4 - Marketplace and Broadcast News
Hub 5 - LVCSR Switchboard