Data preparation

Introduction

After running the example scripts (see **Kaldi tutorial**), you may want to set up Kaldi to run with your own data. This section explains how to prepare the data. This page will assume that you are using the latest version of the example scripts (typically named "s5" in the example directories, e.g. egs/rm/s5/). In addition to this page, you can refer to the data preparation scripts in those directories. The top-level run.sh scripts (e.g. egs/rm/s5/run.sh) have a few commands at the top of them that relate to various phases of data preparation. The parts in the sub-directory named local/ are always specific to the database. For example, in the Resource Management (RM) setup it is local/rm_data_prep.sh. In the case of RM these commands are:

```
local/rm_data_prep.sh /export/corpora5/LDC/LDC93S3A/rm_comp || exit 1;

utils/prepare_lang.sh data/local/dict '!SIL' data/local/lang data/lang || exi

local/rm_prepare_grammar.sh || exit 1;
```

In the WSJ case the commands are:

```
wsj0=/export/corpora5/LDC/LDC93S6B
wsj1=/export/corpora5/LDC/LDC94S13B

local/wsj_data_prep.sh $wsj0/??-{?,??}.? $wsj1/??-{?,??}.?  || exit 1;

local/wsj_prepare_dict.sh || exit 1;

utils/prepare_lang.sh data/local/dict "<SPOKEN_NOISE>" data/local/lang_tmp da



local/wsj_format_data.sh || exit 1;
```

There are more commands after these in the WSJ script that relate to training language models locally (rather than using the ones supplied by LDC), but the ones above are the most important ones.

The output of the data preparation stage consists of two sets of things. One relates to "the data" (directories like data/train/) and one relates to "the language" (directories like data/lang/). The "data" part relates to the specific recordings you have, and the "lang" part contains things that relate more to the language itself, such as the lexicon, the phone set, and various extra information about the phone set that Kaldi needs. If you want to prepare data which you will decode with an already existing system and an already existing language model, the "data" part is all you need to touch.

Data preparation-- the "data" part.

As an example of the "data" part of the data preparation, look at the directory "data/train" in one of the example directories (assuming you have already run the scripts there). Note: there is nothing special about the directory name "data/train". There are other directories such as "data/eval2000" (for a test set) that have essentially the same format ("essentially" because we may have an "sgm" and "glm" file in the test directory, to enable sclite scoring). The specific example we'll look at the Switchboard recipe in egs/swbd/s5.

```
s5# ls data/train
cmvn.scp  feats.scp  reco2file_and_channel  segments  spk2utt  text  utt2spk
```

Not all of the files are equally important. For a simple setup where there is no "segmentation" information (i.e. each utterance corresponds to a single file), the only files you have to create yourself are "utt2spk", "text" and "wav.scp" and possibly "segments" and "reco2file_and_channel", and the rest will be created by standard scripts.

We will describe the files in this directory, starting with the files you need to create yourself.

Files you need to create yourself

The file "text" contains the transcriptions of each utterance.

```
s5# head -3 data/train/text
sw02001-A_000098-001156 HI UM YEAH I'D LIKE TO TALK ABOUT HOW YOU DRESS FOR W
sw02001-A_001980-002131 UM-HUM
sw02001-A_002736-002893 AND IS
```

The first element on each line is the utterance-id, which is an arbitrary text string, but if you have speaker information in your setup, you should make the speaker-id a prefix of the utterance id; this is important for reasons relating to the sorting of these files. The rest of the line is the transcription of each sentence. You don't have to make sure that all words in this file are in your vocabulary; out of vocabulary words will get mapped to a word specified in the file data/lang/oov.txt.

It needs to be the case that when you sort both the utt2spk and spk2utt files, the orders "agree", e.g. the list of speaker-ids extracted from the utt2spk file is the same as the string sorted order. The easiest way to make this happen is to make the speaker-ids a prefix of the utter Although, in this particular example we have used an underscore to separate the "speaker" and "utterance" parts of the utterance-id, in general it is probably safer to use a dash ("-"). This is because it has a lower ASCII value; if the speaker-ids vary in length, in certain cases the speaker-ids and their corresponding utterance ids can end up being sorted in different orders when using the standard "C"-style ordering on strings, which will lead to a crash. Another important file is `wav.scp`. In the Switchboard example,

```
s5# head -3 data/train/wav.scp
sw02001-A /home/dpovey/kaldi-trunk/tools/sph2pipe_v2.5/sph2pipe -f wav -p -c
sw02001-B /home/dpovey/kaldi-trunk/tools/sph2pipe_v2.5/sph2pipe -f wav -p -c
```

The format of this file is

```
<recording-id> <extended-filename>
```

where the "extended-filename" may be an actual filename, or as in this case, a command that extracts a wav-format file. The pipe symbol on the end of the extended-filename specifies that it is to be interpreted as a pipe. We will explain what "recording-id" is below, but we would first like to point out that if the "segments" file does not exist, the first token on each line of "wav.scp" file is just the utterance id. The files in wav.scp must be single-channel (mono); if the underlying wav files have multiple channels, then a sox command must be used in the wav.scp to extract a particular channel.

In the Switchboard setup we have the "segments" file, so we'll discuss this next.

```
s5# head -3 data/train/segments
sw02001-A_000098-001156 sw02001-A 0.98 11.56
sw02001-A_001980-002131 sw02001-A 19.8 21.31
sw02001-A_002736-002893 sw02001-A 27.36 28.93
```

The format of the "segments" file is:

```
<utterance-id> <recording-id> <segment-begin> <segment-end>
```

where the segment-begin and segment-end are measured in seconds. These specify time offsets into a recording. The "recording-id" is the same identifier as is used in the "wav.scp" file– again, this is an arbitrary identifier that you can choose. The file "reco2file_and_channel" is only used when scoring (measuring error rates) with NIST's "sclite" tool:

```
s5# head -3 data/train/reco2file_and_channel
sw02001-A sw02001 A
sw02001-B sw02001 B
sw02005-A sw02005 A
```

The format is:

```
<recording-id> <filename> <recording-side (A or B)>
```

The filename is typically the name of the .sph file, without the suffix, but in general it's whatever identifier you have in your "stm" file. The recording side is a concept that relates to telephone conversations where there are two channels, and if not, it's probably safe to use "A". If you don't have an "stm" file or you have no idea what this is all about, then you don't need the "reco2file_and_channel" file.

The last file you need to create yourself is the "utt2spk" file. This says, for each utterance, which speaker spoke it.

```
s5# head -3 data/train/utt2spk
sw02001-A_000098-001156 2001-A
sw02001-A_001980-002131 2001-A
sw02001-A_002736-002893 2001-A
```

The format is

```
<utterance-id> <speaker-id>
```

Note that the speaker-ids don't need to correspond in any very accurate sense to the names of actual speakers– they simply need to represent a reasonable guess. In this case we assume each conversation side (each side of the telephone conversation) corresponds to a single speaker. This is not entirely true – sometimes one person may hand the phone to another person, or the same person may be speaking in multiple calls – but it's good enough for our purposes. **If you have no information at all about the speaker identities, you can just make the speaker-ids the same as the utterance-ids** , so the format of the file would be just `<utterance-id> <utterance-id>`. We have made the previous sentence bold because we have encountered people creating a "global" speaker-id. This is a bad idea because it makes cepstral mean normalization ineffective in traning (since it's applied globally), and because it will create problems when you use utils/split_data_dir.sh to split your data into pieces.

There is another file that exists in some setups; it is used only occasionally and not in the Kaldi system build. We show what it looks like in the Resource Management (RM) setup:

```
s5# head -3 ../../rm/s5/data/train/spk2gender
adg0 f
ahh0 m
ajp0 m
```

This file maps from speaker-id to either "m" or "f" depending on the speaker gender.

All of these files should be sorted. If they are not sorted, you will get errors when you run the scripts. In **The Table concept** we explain why this is needed. It has to do with the I/O framework; the ultimate reason for the sorting is to enable something equivalent to random-access lookup on a stream that doesn't support fseek(), such as a piped command. Many Kaldi programs are reading multiple pipes from other Kaldi commands, reading different types of object, and are doing something roughly comparable to merge-sort on the different inputs; merge-sort, of course, requires that the inputs be sorted. Be careful when you sort that you have the shell variable LC_ALL defined as "C", for example (in bash),

```
export LC_ALL=C
```

If you don't do this, the files will be sorted in an order that's different from how C++ sorts strings, and Kaldi will crash. You have been warned!

If your data consists of a test set from NIST that has an "stm" and a "glm" file provided so that you can measure WER, then you can put these files in the data directory with the names "stm" and "glm". Note that we put the scoring script (which measures WER) in `local/score.sh`, which means it is specific to the setup; not all of the scoring scripts in all of the setups will recognize the stm and glm file. An example of a scoring script that uses those files is the one the Switchboard setup, i.e. `egs/swbd/s5/local/score_sclite.sh`, which is invoked by the top-level scoring script `egs/swbd/s5/local/score.sh` if it notices that your test set has the stm and glm files.

Files you don't need to create yourself

The other files in this directory can be generated from the files you provide. You can create the "spk2utt" file by a command like the following (this one is extracted from egs/rm/s5/local/rm_data_prep.sh)

```
utils/utt2spk_to_spk2utt.pl data/train/utt2spk > data/train/spk2utt
```

This is possible because the utt2spk and spk2utt files contain exactly the same information; the format of the spk2utt file is `<speaker-id> <utterance-id1> <utterance-id2> ....`

Next we come to the `feats.scp` file.

```
s5# head -3 data/train/feats.scp
sw02001-A_000098-001156 /home/dpovey/kaldi-trunk/egs/swbd/s5/mfcc/raw_mfcc_tr
sw02001-A_001980-002131 /home/dpovey/kaldi-trunk/egs/swbd/s5/mfcc/raw_mfcc_tr
sw02001-A_002736-002893 /home/dpovey/kaldi-trunk/egs/swbd/s5/mfcc/raw_mfcc_tr
```

This points to the extracted features– MFCC features in this case, because that is what we use in this particular script. The format is:

```
<utterance-id> <extended-filename-of-features>
```

Each of the feature files contains a matrix, in Kaldi format. In this case the dimension of the matrix would be (the length of the file in 10ms intervals) by 13. The "extended filename" `/home/dpovey/kaldi-trunk/egs/swbd/s5/mfcc/raw_mfcc_train.1.ark:24` means, open the "archive" file /home/dpovey/kaldi-trunk/egs/swbd/s5/mfcc/raw_mfcc_train.1.ark, fseek() to position 24, and read the data that's there.

This feats.scp file is created by the command

```
steps/make_mfcc.sh --nj 20 --cmd "$train_cmd" data/train exp/make_mfcc/train
```

which is invoked by the top-level "run.sh" script. For the definitions of the shell variables, see that script. `$mfccdir` is a user-specified directory where the .ark files will be written.

The last file in the directory data/train is "cmvn.scp". This contains statistics for cepstral mean and variance normalization, indexed by speaker. Each set of statistics is a matrix, of dimension 2 by 14 in this case. In our example, we have:

```
s5# head -3 data/train/cmvn.scp
2001-A /home/dpovey/kaldi-trunk/egs/swbd/s5/mfcc/cmvn_train.ark:7
2001-B /home/dpovey/kaldi-trunk/egs/swbd/s5/mfcc/cmvn_train.ark:253
2005-A /home/dpovey/kaldi-trunk/egs/swbd/s5/mfcc/cmvn_train.ark:499
```

Unlike feats.scp, this scp file is indexed by speaker-id, not utterance-id. This file is created by a command such as this:

```
steps/compute_cmvn_stats.sh data/train exp/make_mfcc/train $mfccdir
```

(this example is from `egs/swbd/s5/run.sh`).

Because errors in data preparation can cause problems later on, we have a script to check that the data directory is correctly formatted. Run e.g.

```
utils/validate_data_dir.sh data/train
```

You may also find the following command useful:

```
utils/fix_data_dir.sh data/train
```

(of course the command will work for any data directory, not just data/train). This script will fix sorting errors and will remove any utterances for which some required data, such as feature data or transcripts, is missing.

Data preparation-- the "lang" directory.

Now we turn our attention to the "lang" directory.

```
s5# ls data/lang
L.fst  L_disambig.fst  oov.int   oov.txt  phones  phones.txt  topo  words.txt
```

There may other directories with a very similar format: in the case we have a directory "data/lang_test" that contains the same information but also a file G.fst that is a Finite State Transducer form of the language model:

```
s5# ls data/lang_test
G.fst  L.fst  L_disambig.fst  oov.int  oov.txt   phones  phones.txt  topo  wo
```

Note that lang_test/ was created by copying lang/ and adding G.fst. Each of these directories seems to contain only a few files. It's not quite as simple as this though, because "phones" is a directory:

```
s5# ls data/lang/phones
context_indep.csl  disambig.txt         nonsilence.txt        roots.txt    si
context_indep.int  extra_questions.int  optional_silence.csl  sets.int     wo
context_indep.txt  extra_questions.txt  optional_silence.int  sets.txt     wo
disambig.csl       nonsilence.csl       optional_silence.txt  silence.csl
```

The phones directory contains various bits of information about the phone set; there are three versions of some of the files, with extensions .csl, .int and .txt, that contain the same information in three formats. Fortunately you, as a Kaldi user, don't have to create all of these files because we have a script "utils/prepare_lang.sh" that creates it all for you based on simpler inputs. Before we describe that script and the simpler inputs it takes, we feel obligated to explain what is in the "lang" directory. After that we will explain the easy way to create it. The user who is simply aiming to quickly build a system without needing to understand how Kaldi works may skip to **Creating the "lang" directory** below.

Contents of the "lang" directory

First there are the files `phones.txt` and `words.txt`. These are both symbol-table files, in the OpenFst format, where each line is the text form and then the integer form:

```
s5# head -3 data/lang/phones.txt
<eps> 0
SIL 1
SIL_B 2
s5# head -3 data/lang/words.txt
<eps> 0
!SIL 1
-'S 2
```

These files are used by Kaldi to map back and forth between the integer and text forms of these symbols. They are mostly only accessed by the scripts utils/int2sym.pl and utils/sym2int.pl, and by the OpenFst programs fstcompile and fstprint.

The file `L.fst` is the Finite State Transducer form of the lexicon (L, see "Speech Recognition with Weighted Finite-State Transducers" by Mohri, Pereira and Riley, in Springer Handbook on SpeechProcessing and

Speech Communication, 2008). with phone symbols on the input and word symbols on the output. The file `L_disambig.fst` is the lexicon, as above but including the disambiguation symbols #1, #2, and so on, as well as the self-loop with #0 on it to "pass through" the disambiguation symbol from the grammar. See **Disambiguation symbols** for more explanation. Anyway, you won't have to deal with this directly.

The file `data/lang/oov.txt` contains just a single line:

```
s5# cat data/lang/oov.txt
<UNK>
```

This is the word that we will map all out-of-vocabulary words to during training. There is nothing special about "<UNK>" here, and it does not have to be this particular word; what is important is that this word should have a pronunciation containing just a phone that we designate as a "garbage phone"; this phone will align with various kinds of spoken noise. In our particular setup, this phone is called <SPN> (short for "spoken noise"):

```
s5# grep -w UNK data/local/dict/lexicon.txt
<UNK> SPN
```

The file `oov.int` contains the integer form of this (extracted from `words.txt`), which happens to be 221 in this setup. You might notice that in the Resource Management setup, oov.txt contains the silence word, which in that setup happens to be called "!SIL". In that case we simply chose an arbitrary word from the vocabulary– there are no out of vocabulary words in the training set, so the word we choose has no effect.

The file data/lang/topo contains the following data:

```
s5# cat data/lang/topo
<Topology>
<TopologyEntry>

<ForPhones>
21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
</ForPhones>
<State> 0 <PdfClass> 0 <Transition> 0 0.75 <Transition> 1 0.25 </State>
<State> 1 <PdfClass> 1 <Transition> 1 0.75 <Transition> 2 0.25 </State>
<State> 2 <PdfClass> 2 <Transition> 2 0.75 <Transition> 3 0.25 </State>
<State> 3 </State>
</TopologyEntry>
<TopologyEntry>
<ForPhones>
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
</ForPhones>
<State> 0 <PdfClass> 0 <Transition> 0 0.25 <Transition> 1 0.25 <Transition> 2
<State> 1 <PdfClass> 1 <Transition> 1 0.25 <Transition> 2 0.25 <Transition> 3
<State> 2 <PdfClass> 2 <Transition> 1 0.25 <Transition> 2 0.25 <Transition> 3
<State> 3 <PdfClass> 3 <Transition> 1 0.25 <Transition> 2 0.25 <Transition> 3
<State> 4 <PdfClass> 4 <Transition> 4 0.75 <Transition> 5 0.25 </State>
<State> 5 </State>
</TopologyEntry>
</Topology>
```

This specifies the topology of the HMMs we use. In this case, the "real" phones contain three emitting states with the standard 3-state left-to-right topology– the "Bakis model". (Emitting states are states that "emit" feature vectors, as distinct from the "fake" non-emitting states that are just used to glue other states together). Phones 1 to 20 are various kinds of silence and noise; we have a lot because of word-position-dependency, and in fact most of these will never be used; the real number excluding word position dependency is more like five. The "silence phones" have a more complex topology with an initial emitting state and an end emitting state, but then three emitting states in the middle. You don't have to create this file by hand.

There are a number of files in `data/lang/phones/` that specify various things about the phone set. Most of these files exist in three separate versions: a ".txt" form, e.g.:

```
s5# head -3 data/lang/phones/context_indep.txt
SIL
SIL_B
SIL_E
```

a ".int" form, e.g:

```
s5# head -3 data/lang/phones/context_indep.int
1
2
3
```

and a ".csl" form, which in a slight abuse of notation, denotes a colon-separated list, not a comma-separated list:

```
s5# cat data/lang/phones/context_indep.csl
1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16:17:18:19:20
```

These files always contain the same information, so let's focus on the ".txt" form which is more human-readable. The file "context_indep.txt" contains a list of those phones for which we build context-independent models: that is, for those phones, we do not build a decision tree that gets to ask questions about the left and right phonetic context. In fact, we do build smaller trees where we get to ask questions about the central phone and the HMM-state; this depends on the "roots.txt" file which we'll describe below. See **How decision trees are used in Kaldi** for more in-depth discussion of tree issues.

The file `context_indep.txt` contains all the phones which are not "real phones": i.e. silence (SIL), spoken noise (SPN), non-spoken noise (NSN), and laughter (LAU):

```
# cat data/lang/phones/context_indep.txt
SIL
SIL_B
SIL_E
SIL_I
SIL_S
SPN
SPN_B
SPN_E
SPN_I
SPN_S
NSN
NSN_B
NSN_E
NSN_I
```

```
NSN_S
LAU
LAU_B
LAU_E
LAU_I
LAU_S
```

There are a lot of variants of these phones because of word-position dependency; not all of these variants will ever be used. Here, `SIL` would be the silence that gets optionally inserted by the lexicon (not part of a word), `SIL_B` would be a silence phone at the beginning of a word (which should never exist), `SIL_I` word-internal silence (unlikely to exist), `SIL_E` word-ending silence (should never exist), and `SIL_S` would be silence as a "singleton word", i.e. a phone with only one word-- this might be used if you had a "silence word" in your lexicon and explicit silences appear in your transcriptions.

The files `silence.txt` and `nonsilence.txt` contains lists of the silence phones and nonsilence phones respectively. These should be mutually exclusive and together, should contain all the phones. In this particular setup, `silence.txt` is identical to `context_indep.txt`. What we mean by "nonsilence" phones is, phones that we intend to estimate various kinds of linear transforms on: that is, global transforms such as LDA and MLLT, and speaker adaptation transforms such as fMLLR. Our belief based on prior experience is that it does not pay to include silence in the estimation of such transforms. Our practice is to designate all silence, noise and vocalized-noise phones as "silence" phones, and all phones representing traditional phonemes as "nonsilence" phones. We haven't experimented in Kaldi with the best way to do this.

```
s5# head -3 data/lang/phones/silence.txt
SIL
SIL_B
SIL_E
s5# head -3 data/lang/phones/nonsilence.txt
IY_B
IY_E
IY_I
```

The file `disambig.txt` contains a list of the "disambiguation symbols" (see **Disambiguation symbols**):

```
s5# head -3 data/lang/phones/disambig.txt
#0
#1
#2
```

These symbols appear in the file `phones.txt` as if they were phones.

The file `optional_silence.txt` contains a single phone which can optionally appear between words:

```
s5# cat data/lang/phones/optional_silence.txt
SIL
```

The mechanism by which it appears optionally between words is that it appears optionally in the lexicon FST at the end of every word (and also the beginning of the utterance). The reason it has to be specified in `phones/` instead of just appearing in `L.fst` is obscure and we won't go into it here.

The file `sets.txt` contains sets of phones that we group together (consider as the same phone) while clustering the phones in order to create the context-dependency questions (in Kaldi we use automatically generated questions when building decision trees, rather than linguistically meaningful ones). In this particular setup, `sets.txt` groups together all the word-position-dependent versions of each phone:

```
s5# head -3 data/lang/phones/sets.txt
SIL SIL_B SIL_E SIL_I SIL_S
SPN SPN_B SPN_E SPN_I SPN_S
NSN NSN_B NSN_E NSN_I NSN_S
```

The file `extra_questions.txt` contains some extra questions which we'll include in addition to the automatically generated questions:

```
s5# cat data/lang/phones/extra_questions.txt
IY_B B_B D_B F_B G_B K_B SH_B L_B M_B N_B OW_B AA_B TH_B P_B OY_B R_B UH_B AE
IY_E B_E D_E F_E G_E K_E SH_E L_E M_E N_E OW_E AA_E TH_E P_E OY_E R_E UH_E AE
IY_I B_I D_I F_I G_I K_I SH_I L_I M_I N_I OW_I AA_I TH_I P_I OY_I R_I UH_I AE
IY_S B_S D_S F_S G_S K_S SH_S L_S M_S N_S OW_S AA_S TH_S P_S OY_S R_S UH_S AE
SIL SPN NSN LAU
SIL_B SPN_B NSN_B LAU_B
SIL_E SPN_E NSN_E LAU_E
SIL_I SPN_I NSN_I LAU_I
SIL_S SPN_S NSN_S LAU_S
```

You will observe that a question is simply a set of phones. The first four questions are asking about the word-position, for regular phones; and the last five do the same for the "silence phones". The "silence" phones also come in a variety without a suffix like _B, for example `SIL`. These may appear as optional silence in the lexicon, i.e. not inside an actual word. In setups with things like tone dependency or stress markings, `extra_questions.txt` may contain questions that relate to those features.

The file `word_boundary.txt` explains how the phones relate to word positions:

```
s5# head  data/lang/phones/word_boundary.txt
SIL nonword
SIL_B begin
SIL_E end
SIL_I internal
SIL_S singleton
SPN nonword
SPN_B begin
```

This is the same information as is in the suffixes of the phones (_B and so on), but we don't like to hardcode this in the text form of the phones– for one thing, Kaldi executables never see the text form of the phones, but only an integerized form. So it is specified by this file `word_boundary.txt`. The main reason we need this information is in order to recover the word boundaries within lattices (for example, the program lattice-align-words reads the integer versin of this file, `word_boundaray.int`). Finding the word boundaries is useful for reasons including NIST sclite scoring, which requires the time markings for words, and for other downstream processing.

The file `roots.txt` contains information that relates to how we build the phonetic-context decision tree:

```
head data/lang/phones/roots.txt
shared split SIL SIL_B SIL_E SIL_I SIL_S
shared split SPN SPN_B SPN_E SPN_I SPN_S
shared split NSN NSN_B NSN_E NSN_I NSN_S
shared split LAU LAU_B LAU_E LAU_I LAU_S
...
shared split B_B B_E B_I B_S
```

For now you can ignore the words "shared" and "split"– these relate to certain options in how we build the decision tree (see **How decision trees are used in Kaldi** for more information). The significance of having a number of phones on a single line, for example `SIL SIL_B SIL_E SIL_I SIL_S`, is that all of these phones have a single "shared root" in the decision tree, so states may be shared between those phones. For stress and tone-dependent systems, typically all the stress or tone-dependent versions of a particular phone will appear on the same line. In addition, all three states of a HMM (or all five states, for silences) share the root, and the decision-tree building process gets to ask about the state. This sharing of the decision-tree root between the HMM-states is what we mean by "shared" in the roots file.

Creating the "lang" directory

The `data/lang/` directory contains a lot of different files, so we have provided a script that creates it for you starting from a relatively simple input:

```
utils/prepare_lang.sh data/local/dict "<UNK>" data/local/lang data/lang
```

Here, the inputs are the directory `data/local/dict/`, and the label <UNK> which is the dictionary word we will map OOV words to when appear in transcripts (this becomes data/lang/oov.txt). The location `data/local/lang/` is simply a temporary directory which the script will use; `data/lang/` is where it actually puts its output.

The thing which you, as the data-preparer, need to create, is the directory `data/local/dict/`. The directory contains the following contents:

```
s5# ls data/local/dict
extra_questions.txt  lexicon.txt nonsilence_phones.txt  optional_silence.txt
```

(in fact there are a few more files there which we haven't listed, but they are just temporary files that were put there while creating that directory, and we can ignore them). The commands below give you an idea what is in these files:

```
s5# head -3 data/local/dict/nonsilence_phones.txt
IY
B
D
s5# cat data/local/dict/silence_phones.txt
SIL
SPN
NSN
LAU
s5# cat data/local/dict/extra_questions.txt
s5# head -5 data/local/dict/lexicon.txt
!SIL SIL
-'S S
```

```
-'S Z
-'T K UH D EN T
-1K W AH N K EY
```

As you can see, the contents of this directory are very simple in this setup (the Switchboard setup). We just have lists of the "real" phones and of the "silence" phones respectively, an empty file called `extra_questions.txt`, and a file called `lexicon.txt` which has the format

```
<word> <phone1> <phone2> ...
```

Note: `lexicon.txt` will contain repeated entries for the same word, on separate lines, if we have multiple pronunciations for it. If you want to use pronunciation probabilities, instead of creating the file `lexicon.txt`, create a file called `lexiconp.txt` that has the probability as the second field. Note that it is a common practice to normalize the pronunciations probabilities so that instead of summing to one, the most probable pronunciation for each word is one. This tends to give better results. For a top-level script that runs with pronunciation probabilities, search for pp in `egs/wsj/s5/run.sh`.

Notice that in this input there is no notion of word-position dependency, i.e. no suffixes like _B and _E. This is because it is the scripts `prepare_lang.sh` that adds those suffixes.

You can see from the empty `extra_questions.txt` file that there is some kind of potential here that is not being fully exploited. This relates to things like stress markings or tone markings. You may want to have different versions of a particular phone that have different stress or tone. In order to demonstrate what this looks like, we'll view the same files as above, but in the `egs/wsj/s5/` setup. The result is below:

```
s5# cat data/local/dict/silence_phones.txt
SIL
SPN
NSN
s5# head data/local/dict/nonsilence_phones.txt

S
UW UW0 UW1 UW2
T
N
K
Y
Z
AO AO0 AO1 AO2
AY AY0 AY1 AY2
SH
s5# head -6 data/local/dict/lexicon.txt
!SIL SIL
<SPOKEN_NOISE> SPN
<UNK> SPN
<NOISE> NSN
!EXCLAMATION-POINT  EH2 K S K L AH0 M EY1 SH AH0 N P OY2 N T
"CLOSE-QUOTE  K L OW1 Z K W OW1 T
s5# cat data/local/dict/extra_questions.txt
SIL SPN NSN
S UW T N K Y Z AO AY SH W NG EY B CH OY JH D ZH G UH F V ER AA IH M DH L AH P
UW1 AO1 AY1 EY1 OY1 UH1 ER1 AA1 IH1 AH1 OW1 AW1 AE1 IY1 EH1
UW0 AO0 AY0 EY0 OY0 UH0 ER0 AA0 IH0 AH0 OW0 AW0 AE0 IY0 EH0
UW2 AO2 AY2 EY2 OY2 UH2 ER2 AA2 IH2 AH2 OW2 AW2 AE2 IY2 EH2
s5#
```

You may notice that some of the lines in `nonsilence_phones.txt` contain multiple phones on a single line. These are the different stress-dependent versions of the vowels. Note that four different versions of each phone appear in the CMU dictionary: for example, `UW UW0 UW1 UW2`; for some reason, one of these versions does not have a numeric suffix. The order of the phones on the line does not matter, but the grouping into different lines does matter; in general, we advise users to group all forms of each "real phone" on a separate line. We use the stress markings present in the CMU dictionary. The file extra_questions.txt contains a single question containing all of the "silence" phones (in fact this is unnecessary as it appears that the script `prepare_lang.sh` adds such a question anyway), and also a question corresponding to each of the different stress markers. These questions are necessary in order to get any benefit from the stress markers, because the fact that the different stress-dependent versions of each phone are together on the lines of `nonsilence_phones.txt`, ensures that they stay together in `data/lang/phones/roots.txt` and `data/lang/phones/sets.txt`, which in turn ensures that they share the same tree root and can never be distinguished by a question. Thus, we have to provide a special question that affords the decision-tree building process a way to distinguish between the phones. Note: the reason we put the phones together in the `sets.txt` and `roots.txt` is that some of the stress-dependent versions of phones may have too little data to robustly estimate either a separate decision tree or the phone clustering information that's used in producing the questions. By grouping them together like this, we ensure that in the absence of enough data to estimate them separately, these different versions of the phone all "stay together" throughout the decision-tree building process.

We should mention at this point that the script `utils/prepare_lang.sh` supports a number of options. To give you an idea of what they are, here is the usage messages of that script:

```
usage: utils/prepare_lang.sh <dict-src-dir> <oov-dict-entry> <tmp-dir> <lang-
e.g.: utils/prepare_lang.sh data/local/dict <SPOKEN_NOISE> data/local/lang da
options:
     --num-sil-states <number of states>             # default: 5, #states in
     --num-nonsil-states <number of states>          # default: 3, #states in
     --position-dependent-phones (true|false)        # default: true; if true
                                                     # markers on phones to i
     --share-silence-phones (true|false)             # default: false; if tru
                                                     # all non-silence phones
     --sil-prob <probability of silence>             # default: 0.5 [must hav
```

A potentially important option is the `–share-silence-phones` option. The default is false. If this option is true, all the pdf's (the Gaussian mixture models) of all the silence phones such as silence, vocalized-noise, noise and laughter, will be shared and only the transition probabilities will differ between those models. It's not clear why this should help, but we found that it was extremely helpful for the Cantonese data of IARPA's BABEL project. That data is very messy and has long untranscribed portions that we try to align to a special phone which we designate for that purpose. We suspect that the training data was somehow failing to align correctly, and for some reason setting this option to true changed that.

Another potentially important option is the "--sil-prob" option. In general, we have not experimented much with any of these options so we cannot give very detailed advice.

Creating the language model or grammar

Our tutorial above on how to create the `lang/` directory did not address how to create the file `G.fst`, which is the finite state transducer form of the language model or grammar that we'll decode with. In fact, in some setups we may have many "lang" directories for testing purposes, with different language models and dictionaries. The Wall Street Journal (WSJ) setup is an example:

```
s5# echo data/lang*
data/lang data/lang_test_bd_fg data/lang_test_bd_tg data/lang_test_bd_tgpr da
 data/lang_test_bg_5k data/lang_test_tg data/lang_test_tg_5k data/lang_test_t
```

The process for creating `G.fst` is different depending on whether we're using a statistical language model or some kind of grammar. In the RM setup there is a bigram grammar, which only allows certain pairs of words. We make this sum to one within each grammar state by assigning a probability of 1 over the number of outgoing arcs. There is a statement in `local/rm_data_prep.sh` that does:

```
local/make_rm_lm.pl $RMROOT/rm1_audio1/rm1/doc/wp_gram.txt  > $tmpdir/G.txt |
```

This script `local/make_rm_lm.pl` creates a grammar in FST format (text format, not binary format). It contains lines like the following:

```
s5# head data/local/tmp/G.txt
0    1    ADD    ADD    5.19849703126583
0    2    AJAX+S    AJAX+S    5.19849703126583
0    3    APALACHICOLA+S    APALACHICOLA+S    5.19849703126583
```

See www.openfst.org for more information on OpenFst (they have a useful tutorial). The script `local/rm_prepare_grammar.sh` will turn this into the binary-format file `G.fst` using the following statement:

```
fstcompile --isymbols=data/lang/words.txt --osymbols=data/lang/words.txt --ke
    --keep_osymbols=false $tmpdir/G.txt > data/lang/G.fst
```

If you want to create your own grammar, you will probably want to do something similar. Note: this type of procedure only applies to grammars of a certain class: it won't allow you to compile a complete Context Free Grammar, because it can't be represented in OpenFst format. There are ways to do this in the WFST framework (e.g. see recent work by Mike Riley with push down transducers), but we have not yet worked with those ideas in Kaldi.

Please, before asking any questions on the list about language models or about making grammar FSTs, read "A Bit of Progress in Language Modeling" by Joshua Goodman; and go to www.openfst.org and do the FST tutorial so that you understand the basics of finite state transducers. (Note that language models would be represented as finite state acceptors, or FSAs, which can be considered as a special case of finite state transducers).

The script `utils/format_lm.sh` deals with converting the ARPA-format language models into an OpenFst format. Here is the usage messages of that script:

```
Usage: utils/format_lm.sh <lang_dir> <arpa-LM> <lexicon> <out_dir>
E.g.: utils/format_lm.sh data/lang data/local/lm/foo.kn.gz data/local/dict/le
Convert ARPA-format language models to FSTs.
```

Some of the key commands from that script are:

```
gunzip -c $lm \
  | arpa2fst --disambig-symbol=#0 \
             --read-symbol-table=$out_dir/words.txt - $out_dir/G.fst
```

This Kaldi program, `arpa2fst`, turns the ARPA-format language model into a Weight Finite State Transducer (actually, an acceptor).

A popular toolkit for building language models is SRILM. Various language modeling toolkits are used in the Kaldi example scripts. SRILM is the best documented and most fully featured, and we generally recommend it (its only drawback is that it don't have the most free licence). Here is the usage messages of `utils/format_lm_sri.sh`

```
Usage: utils/format_lm_sri.sh [options] <lang-dir> <arpa-LM> <out-dir>
E.g.: utils/format_lm_sri.sh data/lang data/local/lm/foo.kn.gz data/lang_test
Converts ARPA-format language models to FSTs. Change the LM vocabulary using
```