A Toolbox for a Complete ASR System

Overview:

This toolbox constructs a complete ASR system. This toolbox is coupled with the forced alignment tool package, which has a separate manual. The forced alignment tool package is used to find the correct phonetic pronunciation transcriptions for a large dataset. To use the ASR toolbox, the basic assumption is that the correct phonetic transcriptions have been made available, either by running the forced alignment toolbox, or by manual labeling. The examples and steps in this tutorial assume that the phonetic transcriptions are created by the forced alignment toolbox. So, please run the forced alignment toolbox for the training database before implementing the examples in this tutorial. However, for other database where the transcriptions (in HTK MLF format) are already available, such as the Youtube database, it is also very easy to use this ASR toolbox without any need to do forced alignment beforehand. We will also briefly talk about this point in the training step. In this tutorial, our task is to build a system for Mandarin Chinese character level ASR using the toolbox. The steps involved in this task are:

1. Feature extraction for both training and test data.

2. Training monophones.

3. Training triphones.

4. Training a language model.

5. Decoding

The feature extraction step is exactly the same as in the forced alignment step, and the monophone training step has only slight difference. Therefore, these two steps will be briefly explained. We will mainly focus on steps 3 to 5. In the end, various experimental results will be given.

Along with this manual, two folders are provided. In the folder “Tools,” there are all the tool matlab files and their setup files, and in the folder “files needed”, there are all other files needed. These files are for all steps. As we go along steps 1-2-3-4-5, different files will be copied to our experiment folder.

In consistent with the tool format in the forced alignment package, each tool in this ASR toolbox is a matlab m file, and each m name begins with “Tool\_...” A setup file is needed for each tool, and is the only argument that can be passed when the tool is called. The extension of all the setup files is “.dcf,” and for clarity, the setup file names also begin with “Tool\_...” For example, the setup file for the tool “Tool\_Decode.m” is “Tool\_Decode.dcf,” and when this tool is called, the format will be Tool\_Decode(‘Tool\_Decode.dcf’). The setup file for a tool contains all the control options and parameters for that tool. The setup file and the matlab file should be placed in the same experiment folder. Next, steps 1 to 5 will be illustrated.

1. Feature extraction (Tool\_ComputeFeat.m)

1.1. Data preparation

As in the forced alignment step, before we run this feature extraction tool, some data preparation needs to be done. First, create a folder called “data.” Inside this folder, create a subfolder called “train\_wave.” This is the folder where all the training wave files are to be placed. Copy all the training wave files into this folder. Similarly, create another folder called “test\_wave,” and copy all the test wave files into this folder. Next, go outside “data” folder, and create a folder called “exp.” This will be our experiment folder. Then, copy "Tool\_ComputeFeat.m" from “Tools” folder into “exp” folder. This matlab file is the feature extraction tool. Since we want to extract features for both training and test data, we need to call this tool twice, and each time pass a different setup file. Please copy “Tool\_ComputeFeat\_train.dcf” and “Tool\_ComputeFeat\_test.dcf” from “Tools” folder into “exp” folder. These are the setup files for extracting training and test features respectively. In addition, also copy "readhtk.m" file from "files needed" folder into "exp."

A list of all the wave files in “train\_wave” and “test\_wave” folders are needed for the feature extraction tool to read in wave files. Since we have already run the forced alignment step, which also requires the training wave file list in the feature extraction processing, we can simply create a folder “lists” inside “exp” folder, and copy the training wave file list into “lists” folder. Then, rename this list to “train\_wavefile.lst.” The same list used in the forced alignment step is called “wavefile.lst,” but here, we need to distinguish the training list from the test list. However, if we do not have this list from previous steps, we can simply create it using a short program provided in the “files needed” folder called “makelist.m.” Please copy this file into “exp” folder. The description of how to use this short program was explained in details in the forced alignment manual, feature extraction step. After running this program, the folder “lists”, and the wave file list will be generated. The folder name “lists” and the wave file list name “train\_wavefile.lst” can be set in the program. Similarly, since we do not have the test wave file list yet, we can use “makelist.m” to create one and put it in “lists” folder also.

For simplicity, these two lists are also provided in “files needed” folder. We can simply copy them into “lists” folder if we do not want to manually create them.

1.2. Run tool

The detailed explanations of each option in the setup file is provided in the forced alignment manual. First, please refer to the feature extraction step in that manual. The only difference is that we need to call the tool (Tool\_ComputeFeat.m) twice since we want to extract features for both training and test data. Assume that you have already read the feature extraction step in the forced alignment manual, then, please open a new matlab file in your experiment directory (“exp” folder), and write:

copyfile ('cp\_MFCC.ini', 'tfront\cp\_fea13.ini');

copyfile ('snr\_801.trn', 'tfront\tfrontm.dat');

copyfile ('v7\tfrontm.exe', 'tfront\tfrontm.exe');

Tool\_ComputeFeat(‘Tool\_ComputeFeat\_train.dcf’);

Tool\_ComputeFeat(‘Tool\_ComputeFeat\_test.dcf’);

Then, save this new file as “do\_main.m.” This is similar to the main function in a C program, which is responsible for calling different tools in sequence. In our example, we choose to use the standard MFCC method provided in the tfrontm frontend. A brief description on how to setup tfrontm frontend is also given in the forced alignment manual, feature extraction step. Please read the corresponding steps first. The “do\_main” function here assumes that you have already compiled the “tfrontm.m” file, thus, a “tfrontm.exe” was generated. In the forced alignment step, we used 42 features (39 MFCC features plus pitch), which was specified by “cp\_42.ini,” but in this example, we will use the standard 39 MFCC features without pitch first. So, the corresponding configuration file changed to “cp\_MFCC.ini”, which can be found in “files needed” folder.

Later on, we will also extract features with pitch (MFCC+pitch) in another experiment, and compare the performance without pitch features. In that experiment, we will use “cp\_42.ini”, which is also provided in “files needed” folder. We need to choose which pitch tracker to use. The option “Tracker\_type” in “cp\_42.ini” gives 3 methods: Yaapt, Yin, Praat, and “All\_part\_voice\_yappt” controls all voiced mode or partially voiced mode in Yaapt. If you want to use Praat pitch tracker, then, in addition to the steps illustrated in the forced alignment manual, tfrontm frontend steps, you also need to copy “praatcon” and “pitch.praat” files from “files needed” folder to “exp” folder. You can choose all voiced mode or voiced/unvoiced mode in “pitch.praat” file by removing or adding the # sign (which means to comment out the corresponding row).

After the feature extraction step, a “train\_feat” and a “test\_feat” folders will be generated inside “data” folder, and inside these two folders are the training and test feature files. In addition, a “train\_wavefile.lst” and a “test\_wavefile.lst” file will be generated inside “lists” folder. These are the lists of feature files for training and test data.

2. Training monophones (Tool\_trainMono2.m)

This tool is for monophone training. Its setup file is “Tool\_trainMono2.dcf.” The training stage is almost the same as the same tool in the forced alignment step (Tool\_trainFA.m). However, there are two major differences. One is the transcription preparation step; the other one is that there is an additional option called “Triphone\_later” in the initialization step. We will mainly focus on these two differences in this section.

First, please copy the setup file “Tool\_trainMono2.dcf” from “Tools” folder to “exp” folder. Next, create a subfolder “labs” in the “exp” folder. Then, go to the same “labs” folder in your forced alignment experiment folder, and copy the outcome of the last round forced alignment to “labs” folder of our current task example. If you have run the forced alignment step using the provided setup file (Tool\_FA.dcf) in the forced alignment package, the last round outcome should be “aligned\_6.mlf.” This is the “perfect” version of the phonetic transcriptions of the training data. Rename this file to “trainphone\_sp.mlf.” This file is also available in “files needed” folder, and you can just copy it from there to “labs” folder for convenience. But it is strongly recommended that you first run the forced alignment package by yourself to get familiar with similar tools.

A byproduct of the forced alignment step is a list of all the monophones in the training data, including the short pause “sp.” We will use this list in our ASR task. Please go to “lists” folder in your forced alignment experiment folder, and copy the file “monophone\_sp” to the “lists” folder of our ASR task. For convenience, this list can also be found in “files needed” folder.

Then, let’s open “Tool\_trainMono2.dcf” file and focus on the transcription preparation step. The purpose of this step is to delete the short pause (sp) between words from the transcription “trainphone\_sp.mlf.” The output is simply the transcription without “sp.” The reason why we need both the transcription with and without “sp” is that the “sp” model is a short pause between every two words, so, this model is a one-state model, whose parameters are copied from and tied to the central state of the silence (sil) model, which is a 3-state model. To do this, at the beginning of training, a phone set of low mixture models (usually 1 mixture) will be trained using the transcription without “sp,” and then, the “sp” model will be introduced by copying the central state of the silence model. After this point, we will use the transcription with “sp” to keep training the models. The forced alignment transcription contains the “sp” model. So, at the beginning of training, we need to delete it from the transcription.

The followings are the options to delete “sp”

Trans\_prep: y means to turn on transcription preparation; n means to turn it off.

PhoneMLF\_sp: this is the path of the transcription with “sp.” It is the input of this step. In our setup file, the path is labs\trainphone\_sp.mlf.

Conf\_deleteSP: this is the path of the configuration file for the underlying HTK tool (HLEd) to conduct this deletion. In our example, please copy the file “deleteSP.led” from “files needed” folder to “toolconfs” folder. You should have already created the “toolconfs” folder inside “exp” folder in the feature extraction step if you choose to use HTK\_MFCC or HTK\_PLP frontend, as explained in details in the forced alignment manual, feature extraction section. “toolconfs” folder is where we put all the configuration files of the underlying HTK tools. However, since in our example, we’ve chosen to use the tfrontm MFCC frontend, which is not controlled by HTK, you may have not created this folder. So, please create “toolconfs” folder first if that’s the case.

PhoneMLF\_nosp: this is the path for the output transcription without “sp.” In our example, it is set to “labs\trainphone\_nosp.mlf.” So, a MLF file “trainphone\_nosp.mlf” will be generated in the folder “labs.”

PhoneList\_nosp: this is the path for the output phone list without “sp.” In our example, it is set to “lists\monophone\_nosp.” This list, along with the transcription, will be used to train the phone set without “sp.”

Notice that the operations in the transcription preparation step are based on the assumption that the phonetic transcription with “sp” is generated by the forced alignment package. However, in some cases, the transcription is already available in MLF format, such as the Youtube database, and it is possible that there is even no “sp” in the transcription. In this case, there is no need to delete any “sp.” However, we still need a list of all the phones encountered in the transcription. So, we can follow the following steps:

a. Set “Trans\_prep” to y, since we still need to generate the phone list.

b. Set “PhoneMLF\_sp” to the path of your phonetic MLF file. Note that do not try to change the parameter name “PhoneMLF\_sp,” because otherwise, the code of the tool will also need to be changed. Though the parameter name is “PhoneMLF\_sp,” that is only for clarity. You need to be aware that in this case, this is just your phonetic transcription MLF path.

c. For “Conf\_deleteSP,” you can still leave the configuration file path “toolconfs\deleteSP.led” here. Since there is actually no “sp” in the transcription, the underlying HTK tool (HLEd) won’t do anything though the command in the file “deleteSP.led” (DE sp) means to delete “sp” from the transcription. The other way is to make the file “deleteSP.led” an empty file.

d. For the output file parameter “PhoneMLF\_nosp,” this is exactly the same as the input file parameter “PhoneMLF\_sp” because there is no “sp” involved. Suppose your input MLF file name is “trainphone.mlf,” you can make the output file name “trainphone1.mlf,” but they are exactly the same files.

e. “PhoneList\_nosp,” in this case, is the list of monophones in the transcription, and is what we really want. You can specify the file path as you want here.

This is how the transcription preparation step generalizes to the case where the MLF is not obtained from the forced alignment. In the training stage, there are a couple of other places that require inputs of the MLF transcription or phone list, both with and without “sp.” So, we can simply provide the same file for the “sp” version and “non-sp” version. For example, in the embedded training stage, we can set the entries “hmmList\_nosp” and “hmmList\_sp” both to the path of our sole HMM list. In addition, remember to set the entry “fix\_sil” to n, because we do not introduce the “sp” model in this case. The meaning of “fix\_sil”, as well as how the “sp” model is created, and how the silence model is fixed, are described in details in the forced alignment manual. Please refer to “Tool\_trainFA” section for details.

The transcription preparation in this training tool assumes that a MLF format transcription is already available, such as the “trainphone\_sp.mlf” file in our example, and any editing should be made based on this MLF file. However, it is often the case that the initial transcription is in its “raw” format, not MLF format. The “raw” format may take on many possible pattern. For example, in TIMIT database, there is one phonetic transcription for each wave file, and the transcription has time markers for each phone. The reason why we start from a MLF is that it is very difficult to use the same piece of code to convert different raw formats into a MLF format. The code to make this conversion may vary, depending on the specific raw format, and it also makes the tool setup file look too messy with so many options for different variations. So, we have to assume that a user has already got this MLF format of transcription. It is recommended that you go through the tool “HLEd” in HTKbook, section 17.10 to learn how to make this MLF file. An example for TIMIT database is as follows:

In the matlab command window, type in:

arg = sprintf('-G TIMIT -i %s -l \* -n %s -S %s %s', 'trainphone.mlf', 'hmmList','trnp\_exsa.lst','phn2cmu39.led' );

system(sprintf('HLEd -A -T 1 %s', arg));

Then, a MLF file “trainphone.mlf,” and a HMM list “hmmList” will be generated in your current matlab directory. One input is a list “trnp\_exsa.lst,” which is a list of all the raw phonetic transcriptions with one transcription for one wave file. Each file path in this list is with respect to your current matlab directory. Another input is a configuration file “phn2cmu39.led,” which is provided in the “files needed” folder. It converts the TIMIT original phone set to the CMU phone set. A “phn2lab48.led” configuration file is also provided, which converts the TIMIT original phone set to the 48 phone set. A user can choose which phone set to be trained. These input files should also be placed in your current matlab directory. “-G TIMIT” means that the time markers in the raw transcription files are in TIMIT format, whose unit is sample. After conversion, the unit will change to HTK format, which is in 100ns.

With this MLF file as well as the HMM list, there is no need to make any further editing in the transcription preparation step of the monophone training tool, since the original phone set has already been converted to the 39 phone set by the commands above. So, you can simply set “Trans\_prep” to n. Whenever the tool requires the MLF or HMM list in the setup file, either with “sp” or without “sp,” simply pass the same MLF and the HMM list path to both places.

This ASR toolbox is designed for large database. The training method for a large database includes two steps: flat start initialization and embedded training. The concepts of flat start and embedded training are illustrated in the forced alignment manual. Both steps do not require time stamps in the phonetic transcriptions. For some small database, such as TIMIT, the phonetic transcriptions have time stamps for each phone. For such a database, the training method also consists of two steps: boost initialization and embedded training. The boost initialization uses the time markers to cut out feature segments for each phone, and each phone model is trained by its own feature segments individually. The underlying HTK tools for the boost initialization are HInit and HRest. An overview of the boost initialization algorithm can be found in HTKbook, section 2.3.2. At this point, this ASR toolbox only supports the flat start+embedded training mode, and this is also why the monophone training tool is called “Tool\_trainMono2.” A “Tool\_trainMono1” will be developed, which only supports the boost+embedded training mode, which is the typical training method for small database whose phonetic transcriptions contain time markers.

A new option “Triphone\_later” is added to the initialization step, compared with the training tool in the forced alignment package. If we want to use monophones to form triphones in the next step, we should set “Triphone\_later” to y. Triphones can only be formed by 1-mixture monophones. In other words, we can not first train a set of 16-mixture monophones, and use them to form a set of 16-mixture triphones. We can only start from 1-mixture triphones, which are built from 1-mixture monophones, and then, the mixture splitting will be conducted to these 1-mixture triphones to split the mixtures till the desired order. The triphone mixture splitting sequence will be specified in the triphone training tool (Tool\_trainTri.m). So, if we set “Triphone\_later” to y, the monophone mixture splitting sequence will lose effect. The number of mixtures given by “numMixture” will be forced to 1, though it is specified as 1;2;4;6;8;12;16 in the setup file. Also, only the first number of the iteration sequence will be preserved, though it is set as 3;5;5;6;6;7;7. So, all the monophones will be 1-mixture phones, and 3 iterations will be conducted to train these monophones. Thus, in a word, setting “Triphone\_later” to y in the initialization step guarantees that all monophone models are 1-mixture models. Later on, these 1-mixture models will form triphones. If triphones are not wanted, we can simply set “Triphone\_later” to n. In this case, the mixture splitting sequence will be performed, and multiple mixture monophones will be generated.

All the other options and parameters in the setup file are the same as those in the monophone training tool of the forced alignment package. Please read through and implement section 2, Tool\_trainFA in the forced alignment manual. In our task example, a set of 1-mixuture monophones will be generated in the folder “hmms\fhmm\_mono.”

3. Training triphones (Tool\_trainTri.m)

After we’ve got a set of 1-mixture monophones, triphones will be made from these monophones. There are two types of triphones: internal word triphones and cross word triphones. Before we get into the details of the tool, let’s first get to know what they are.

The formation of internal word triphones is bounded by the inter-word short pause “sp” and the silence (sil). Here is an example of how to convert monophone transcription into internal word triphone transcription:

Original sentence: sil this sp man sp…

Monophone sequence: sil th ih s sp m ae n sp…

Internal word triphone sequence: sil th+ih th-ih+s ih-s sp m+ae m-ae+n ae-n sp…

In the above example, the monophone sequence is converted to the internal word triphone sequence. A triphone consists of a central phone, a left phone, and a right phone. For example, in the triphone “th-ih+s”, “ih” is the central phone, “th” the left phone, and “s” the right phone. In HTK, we use a “-“to represent the left context, and a “+” to represent the right context. Note that the word boundary markers “sil” and “sp” are not used to form triphones. They block the addition of context at word boundaries, such that some biphones (or monophones) will also be generated. For this reason, when we talk about internal word triphones from now on, we will simply call them triphones, but reader should be aware that the concept of “triphones”, in the internal word case, also includes biphones or monophones.

The formation of cross word triphones is not subject to the word boundaries. This is why it is called cross word triphones. Next, let’s look at the same example:

Original sentence: sil this sp man sp…

Monophone sequence: sil th ih s sp m ae n sp…

Cross word triphone sequence: sil sil-th+ih th-ih+s ih-s+m sp s-m+ae m-ae+n ae-n+…

As can be seen from this example, the formation of cross word triphones is not restricted to “sil” and “sp.” “sil” is regarded as the context of the center phone, whereas in the internal word style, “sil” can not be a part of any triphones. When building triphones, the “sp” is “jumped over”, so that monophones from adjacent words are combined into a triphone, such as “ih-s+m.” However, the “sp” is not totally ignored. It is actually shifted to the right by one phone. There are different ways to deal with “sil” and “sp” when making cross word triphones. The method in the above example is adopted by the RMHTK, which is an example of how to build HTK based systems for the ARPA RM task. It can be downloaded from <http://htk.eng.cam.ac.uk/download.shtml>. Another way of making cross word triphones can be found from [www.keithv.com/software/htk/](http://www.keithv.com/software/htk/). In our example, we will use the method in the above example.

After knowing what triphones are, we will get into the procedures of triphone training. We will start from internal word triphones.

3.1. Internal word triphones

Copy “Tool\_trainTri.m” from “Tools” folder to “exp” folder, and copy the setup file “Tool\_trainTri\_inword.dcf” from “Tools” folder to “exp” folder. You may notice that there is another setup file “Tool\_trainTri\_xwd.dcf.” That is the setup file for cross word triphones. Training internal word and cross word triphones follows the same steps. So, only one tool (Tool\_trainTri.m) can cover both. Different setup files will be passed to the tool respectively.

Next, please open “Tool\_trainTri\_inword.dcf.” We will explain tutorially the theory behind each step as we go through the setup file.

Trace\_on, Clean\_up, LogDir: these three terms are the same as in the monophone training. “Trace\_on” enables the progress to be displayed on the screen. “Clean\_up” deletes the old triphone models in each step before new models are generated. “LogDir” specifies a log directory inside which a progress report “progress\_trainTri.log” will be generated.

To train internal word triphones, we must first have internal word triphone transcriptions. In the first example above, we have learnt how to convert monophone transcriptions to the internal word triphone version in theory. This conversion is implemented by the transcription preparation step. This step has two inputs and two outputs.

Trans\_prep: set this to y enables the conversion, and n turns it off.

PhoneMLF: this is the path of the monophone transcription MLF file, which is an input. Note that in the setup file, this is set to “labs\trainphone\_sp.mlf.” This is the transcription with “sp” between each two words. To make internal word triphones, there must be a word boundary marker, such as “sp.”

Conf\_mon2tri: this is the path of the configuration file for the underlying HTK tool (HLEd) to convert monophones to internal word triphones. Please copy the file “mktri\_inword.led” from “files needed” folder to “toolconfs” folder. In “mktri\_inword.led” file, there are two commands: each WB specifies a word boundary marker, and TC means to expand the monophone transcription to triphone transcription.

TriMLF: this is the output triphone MLF path. We set it to be “labs\traintri\_inword.mlf.”

Trilist\_ini: this is another output, which is a list of all the triphones in the transcription TriMLF. Note that this list only covers the triphones in the training data. But many triphones, which are not in this list, may appear in the test data. To solve this problem, we will have another full list in a later step. So, at this point, this list will only be used for triphone initialization, as implied by the name of the term “Trilist\_ini.”

After we obtain the internal word triphone transcription, we can begin the training processing. There are two main steps: initialization and making tied state triphones. We will describe both steps later. But before we get into the details, there are some global settings and files needed by both steps.

Train\_on: y turns on training; n turns off training.

Feat\_List: this is the training feature file list. Both steps will need this list.

Tri\_MLF: this is the triphone transcription MLF file generated in the transcription preparation step. Both steps will need this transcription.

Src\_hmmfolder: this is the monophone HMM folder, which is the starting point of making triphones. Our monophones are stored in “hmms\fhmm\_mono.”

Final\_hmmfolder: this is the folder to store the final triphone models after initialization and making tied state triphones. This folder is the end point of training. We will place the final triphone models in “hmms\fhmmtri\_inword.” This directory will be automatically created.

embdOptStr: this is the pruning threshold sequence of the embedded training, which has the same meaning as in monophone training.

Conf\_embd: this is the configuration file path for the underlying HTK tool (HERest) of embedded training. There should be a “herest.conf” file already copied to “toolconfs” folder in the monophone training step.

Next, we come to the initialization step. The initialization of triphones is somewhat analogous to the flat start initialization of monophones. First, HTK will read in a list of triphones to be initialized. Then, for all the triphones in this list that have the same central phone, HTK will make a copy of the parameters of the central phone (which is a monophone), and use it as the initial parameters of all these triphones. For example, for all the triphones of the format \*-b+\* (this includes \*-b+\*, \*-b, b+\*, b), the parameters of monophone b will be used as the initialization of all triphones of \*-b+\*. Then, embedded training will be conducted for a couple of iterations, and these initial parameters will change correspondingly.

A problem is obvious: the number of triphones is huge. If each triphone requires its own samples to get trained, then, the parameter estimates will be very poor since many triphones only appear once or twice. So, in the initialization processing, in addition to the “clone” operation described above, the transition matrices of all the triphones in the class \*-b+\* will be tied together. Generally speaking, tying means that one or more HMMs share the same set of parameters. When reestimating tied parameters, the data which would have been used for each of the original untied parameters is pooled so that a much more reliable estimate can be obtained.

Of course, tying could affect performance if performed indiscriminately. Hence, it is important to only tie parameters which have little effect on discrimination. This is the case in the initialization where the transition parameters do not vary significantly with acoustic context but nevertheless need to be estimated accurately. Some triphones will occur only once or twice and so very poor estimates would be obtained if tying was not done.

With this background, let’s go into the setup file to see the initialization part.

Init: y turns on the initialization; n turns it off.

Iteration\_init: this is how many iteration of embedded training to be conducted after initialization.

There are three inputs for initialization.

hmmlist\_mono: this is the monophone HMM list path. Since the initial parameters of each triphone is copied from its central phone, we need to pass this monophone list to HTK to specify which monophones to copy from. Note that in the setup file, this list includes the “sp” model, because “sp” is also a member in the triphone list, and so it needs to be carried through all stages.

Trilist\_init: this is the triphone HMM list path. This list is generated in the transcription preparation stage. The triphones in this list are to be initialized.

Conf\_init: this is the configuration file path for the underlying HTK tool (HHEd) to copy parameters as well as tie transition matrices. Please copy the file “tieTrans.hed” from “files needed” folder to “toolconfs” folder. Let’s look at a command in this configuration file: TI T\_zh {(\*-zh+\*,zh+\*,\*-zh).transP}. The command “TI” means to tie a parameter; {(\*-zh+\*,zh+\*,\*-zh).transP} means that the parameter to be tied is the transition matrices (specified by transP) of all the triphones whose central phone is zh. The “T\_zh” is the macro name of the tied matrix, which means for all the triphones in this class, they will have an identical transition matrix called “T\_zh.” Note that there is no “sil” and “sp” in these commands, since “sil” and “sp” have no left or right contexts according to the internal word rule of forming triphones. But it does not matter if they are in this file; for example, there is a command TI T\_sp {(\*-sp+\*, sp+\*,\*-sp).transP}. In this case, a warning will show up, saying there is nothing to tie for this class, but it won’t hang up the program.

It is very easy to write a short program to create this configuration file by passing it a monophone list (without sp and sil), and each monophone will be expanded into a command of the form “TI…” The content of this configuration file changes with the monophone list. So, if you have your own phone set in your own task, please write a program to make this configuration file by yourself.

However, at this point, the configuration file only does the tying job by executing each “TI” command. It does not include the cloning job, as described above. Actually, what the tool does is that it will add a row before all the “TI” commands, which is “CL lists/trilist\_ini\_inword,” and rename this complete configuration file as “mktri.hed,” and save it to our experiment folder “exp.” The command “CL” simply means clone. The initial triphone list “lists\trilist\_ini\_inword” is specified by the “Trilist\_init” entry in the setup file. Then, the tool will use “mktri.hed” as the complete configuration file to conduct both cloning and tying.

There are two output entries in the initialization step:

TgtDir\_init: this is the directory to save the initialized triphones. A folder “hmms\hmm3\_init\_inword” will be automatically created as specified in the setup file.

Stat\_embd: this is a statistical file generated by the embedded training. As indicated in the setup file, 3 iterations of embedded training will be performed after initialization. After each iteration, a statistical file “toolconfs\hstats” will be generated. The newer one will overwrite the old one. The last round statistical file will be used in the next step, which is making tied state triphones.

Next, before we go into the tool implementation of the last step, a theoretical overview is very helpful for us to understand what this step does.

The outcome of the initialization step is a set of triphones with all triphones with the same central phone sharing the same transition matrix. However, there are two problems remaining. First, the data insufficiency problem is still prominent, since only the transition matrix in each triphone class is tied. When estimating these models, many of the variances in the output distributions will be floored. Second, the triphones that have been trained so far are all from training data; but many possible triphones that might appear in the test data are missing from the training data. To solve these two problems, the last step in building triphones is to tie states within each triphone set (class) that has the same central phone, so that data will be shared when estimating the parameters of each tied state.

Unlike the initialization step, which ties the transition matrices of all the triphones in the same class, how to partition the states of all the triphones with the same central phone into different clusters requires phonetic knowledge. In simple words, states of triphones with similar co-articulation effect is most likely to be tied together. A common characteristic of the triphones in the same cluster is that the co-articulatory impact of their left and right contexts to the central phone is similar across the cluster.

A decision tree based method is used to decide which triphones are to be put in a cluster. First, a question set is carefully designed according to similar co-articulatory impact of the left context and right context. Each question is a binary question. Let’s give two examples of English question set. Here is one example of a left context question:

QS “L\_Class-Stop” {p-\*,b-\*,t-\*,d-\*,k-\*,g-\*}

“QS” means that this is a question, and “L\_Class-Stop” is the question name. This question simply asks: is the left phone of the central phone a stop consonant, namely, one of p, b, t, d, k, g? Recall that HTK uses “-“ to represent the left phone of a central phone. Here is another example of a right context question:

QS “R\_Nasal” {\*+m,\*+n,\*+ng}

This question asks: is the right phone of the central phone a nasal consonant, or namely, one of m, n, ng? It can be seen that each question represents a group of phones that have similar impact on the pronunciation of the central phone.

The decision tree is constructed based on these questions. For example, suppose we want to make clusters of state 2 of all the triphones with a central phone “aw.” Before clustering, there might be hundreds of triphones in this set. Initially, state 2’s of all the triphones in this set are pooled at the root of a tree (the tree has not been built yet). Then, the question set is loaded. Since the answer to each question is binary, by answering a question, the initial pool will be split into two pools. Splitting any pool into two will increase the log likelihood of the training data for that state, since it provides twice as many parameters as the original pool (because each sub-pool has its own Gaussian output density) to model the same amount of data. After all the questions have been scanned through, the one question that provides the biggest improvement of the log likelihood of the training data will be selected as the first branch of the decision tree. This branch gives the best split of the root node, and two descendent nodes are generated by this splitting. The first part of the original pool is placed at one descendent node, and the second part is at the other descendent node. Then, the pool at each descendent node is split again by the locally optimized question (locally means the question maximizes the improvement of the likelihood of training data at that local node). Thus, this processing repeats and a decision tree is built by this top-down sequential optimization process. As the tree keeps growing, the maximal improvement of the log likelihood of the training data brought by a splitting at a descendent node gets smaller. When the maximal improvement at any node falls below a user defined threshold, the construction of the tree stops. It is easy to see that a smaller threshold makes the splitting process last longer, and thus, makes the tree bigger. When the process stops, the nodes at the very bottom of the tree, who do not have any descendants, are called leaf nodes, or senones. The states at each senone are tied together. From now on, these states will share the same training data, and have identical output density parameters. A more detailed explanation of the decision tree based method which includes a pictorial example can be found in the HTKbook, section 10.5. A basic paper on this tree-based tying can be found in [1].

There are a couple of points revealed from the descriptions above. First, the design of the question set is crucial for the quality of the tying. Ideally, the question set would include every possible context which can influence the acoustic realization of the central phone, and can include any linguistic or phonetic classification which may be relevant. There is no harm in creating unnecessary or “meaningless” questions, since only the questions that give the maximal increase in the log likelihood of the training data will be selected in the tree construction processing. Second, the user defined stop criteria determines the number of tied states (leaf nodes). The smaller it is, the more splitting will be conducted, and therefore, more tied states will be generated. In a triphone system, the number of triphones is not that important. What matters is the number of tied states, because they are what get trained. So, this user defined threshold needs to be tuned according to the amount of data we have.

Finally, another advantage of this decision tree based clustering is that it’s able to synthesize the triphones which never appear in the training data. In other words, even if a triphone does not have any samples in the training data, it can still be trained. This is because after the state tying, there is actually no longer concept of any individual triphone. The states of each triphone are categorized into different clusters. So, for any triphone which is not in the training data, each of its state will first find the corresponding tree, and then descends that tree by answering the questions at each node until it gets to one of the leaf node. Then, it will use the parameters of this leaf node as its own estimates for that state.

With this background, let’s focus on the tool to implement these algorithms. Please focus on the “Tied state triphones” part of the setup file.

Tie: y turns on the state tying; n turns it off. Once it is turned on, the models will be loaded from “TgtDir\_init,” which is the target folder of the initialization step, and the tied models will be saved in “Final\_hmmfolder,” which is the folder to store final HMM models.

Iteration\_tie: this is how many iterations of embedded training will be performed after making tied state triphones.

Let’s skip the two configuration files specified by “Question” and “TB” for the time being, and focus on the “Full\_list” entry as one of the inputs. As stated in the background, after the decision trees have been built, unseen triphones in the training data will be synthesized using the trees. To do this, we need to manually create a “full list” which contains all possible internal word triphones for our ASR system, so that HTK will collect those unseen triphones with respect to the training data from this full list. Since the vocabulary of an ASR system consists of all possible words that can be decoded in theory, we can easily find all possible internal word triphones by converting the monophone pronunciation of each word in the dictionary to its internal word triphone form, and make a list of it. To make this conversion, please follow the following steps:

a. Create a folder called “dicts” inside “exp” folder, and copy the dictionary file “dict863\_tone\_sp1” from “files needed” folder to “dicts” folder. Note that a “sp” is appended after each pronunciation entry.

b. Copy the configuration file “global.ded” from “files needed” folder to “exp” folder.

c. Make sure that your matlab directory is “exp.” Then, in your command window, type in this command:

system(sprintf(‘HDMan –b sp –g %s %s %s’, ‘global.ded’,’tridict’,’dicts\dict863\_tone\_sp1’));

The command “HDMan” is the HTK tool to manipulate a dictionary, and the option “-b sp” specifies “sp” as the word boundary. Note that the original dictionary to be converted must be in sorted order; otherwise, “HDMan” won’t work. The dictionary provided is already in sorted order. Then, a file “tridict” will be generated inside “exp” folder. This is the internal word triphone dictionary. The monophone pronunciations have been converted to its internal word triphone format. You may find that the Chinese words all become numbers. Those numbers are the encoding of the Chinese characters. But this does not matter because what we want is the triphone pronunciations, not the Chinese words. Then, you can write a program to get rid of those Chinese words and only preserve the triphones. Such a matlab program called “remove.m” is provided in “files needed” folder. Copy this file to “exp” folder, and the words will be eliminated after running this program. Then, use a text editor to open the output file “tlist,” and replace all spaces by line breakers (\n) and remove empty lines. However, this list contains duplicated entries. Please copy the file “tlist” to a Linux system, and open a terminal. Then, type in the following two commands:

dos2unix tlist

awk ‘!a[$0]++’ tlist>fulllist\_inword

Then, a “fulllist\_inword” will be generated. Only one entry of each duplicated entries is preserved. Copy this file back to the “lists” folder inside the “exp” folder. Note that in HTKbook section 3.3, the tutorial example, there is a “-n fulllist” term in HDMan command. A “fulllist” will be generated after the conversion of the dictionary. However, please do not use this method, since this “fulllist” misses some triphones, which will result in errors in following steps. For your convenience, a “fulllist\_inword” is provided in the “files needed” folder using the recommended method above. This full list comes from the dictionary “dict863\_tone\_sp1.” In your own project, you need to make your own full list if a different dictionary is used.

Full\_list: make a full list of all possible internal word triphones from the dictionary, and specify its file path here.

Now, let’s go back to the two configuration files Question and TB.

Question: this is the question set for building the decision tree. An example question set formed by Chinese tonal phones is provided. Please copy the file “Quest.hed” from “files needed” folder to “toolconfs” folder, and specify the file path here. There is an English question set. It can be found in HTK samples\RMHTK\lib\quests.hed. The HTK samples can be downloaded from the official website of HTK.

TB: please copy “TB.hed” file from “files needed” folder to “exp” folder. Let’s look at one of the TB commands in this file:

TB 2000.0 "zh\_s2" {("zh","\*-zh+\*","zh+\*","\*-zh").state[2]}

Each TB command constructs a decision tree for a state of a triphone set. In this command, a decision tree will be built for state 2 of the triphone set with the central phone “zh.” The procedures of building this tree is described in the background section. The number 2000 is the user defined stop criteria for the growth of the tree. This number needs to be tuned according to the amount of training data. In our example, 2000 is the optimal setting for tonal internal word triphones.

A Perl script “mkclscript” can be found in HTK samples\RMHTK\perl\_scripts to generate this TB file. It is run under Linux. The threshold number as well as a monophone list excluding “sil” and “sp” are needed as two arguments of the program.

In addition to the full list of all the possible triphones, there are other two inputs:

Stat\_embd: this is the statistical file generated by the last round of the embedded training after the initialization step. This file needs to be loaded at the beginning of the clustering processing. After the state tying, it becomes useless, and will be overwritten by embedded training.

Trilist\_init: this is the initial triphone list file path. The clustering will be conducted to each triphone in this list.

In addition to a new set of tied state (also tied transition matrices) triphones stored in the “Final\_hmmfolder,” another two outputs will also be produced:

Trilist\_tied & Tree: “Trilist\_tied” is the output file path for a list of all compact tied-state triphones. In our example, it is set to “lists\tiedlist\_inword.” “Tree” the path where the decision trees get saved to. To be specific, before making decision trees, the tool will first merge the question set “Quest.hed” and the TB file “TB.hed” together, and a new file “tree.hed” will be generated inside “exp” folder. Then, it writes in the following 3 lines at the end of “tree.hed.” Then, “tree.hed” will be used as the complete configuration file for the decision tree construction as well as state clustering.

AU "lists/fulllist\_inword"

CO "lists/tiedlist\_inword"

ST "toolconfs/trees\_inword"

“AU” means to synthesize all the unseen triphones in the full list after trees have been built. After state tying, it is possible that for some triphones with the same central phone, their three emitting states all fall into the same leaf node in corresponding trees. Thus, these triphones become exactly identical (recall that their transition matrices are already tied together, thus also the same). The “CO” command finds such triphones and tie them together, producing a new list of models whose path is designated by “Trilist\_tied.” This final list will be used to load HMMs in the following embedded training and decoding steps. Finally, the generated decision trees will be saved to the path “toolconfs\trees\_inword.”

Till now, we have completed making 1-mixture triphones. These 1-mixture triphones are saved in the directory “hmms\fhmmtri\_inword” specified by “Final\_hmmfolder” entry. However, if we want to get multiple mixture triphones, we can carry out the mixture splitting processing, controlled by the last 4 entries in the setup file. The splitting is directly conducted to the models in the folder “Final\_hmmfolder.”

Split: y enables mixture splitting; n disables it.

numMixture & Iteration: these are the mixture splitting sequence as well as the number of iteration sequence.

numState: this is the number of emitting states.

At this point, the models in the “hmms\fhmmtri\_inword” folder become 16-mixture models, as specified by the mixture splitting sequence. We need to emphasize again that multiple mixture triphones can ONLY be obtained by splitting mixtures from 1-mixture triphones. Directly making multiple mixture triphoens from multiple mixture monophones is not permitted. In fact, when the tool loads monophones before triphone initialization step, it will first check whether the monophones are 1-mixture. If not, the tool will give out an error and stop proceeding.

3.2. Cross word triphones

The theory of making cross word triphones is the same as that of internal word triphones. There are two differences. First, the conversion from monophones to cross word triphones is not subject to the word boundary marker “sil” and “sp.” Second, due to the first property, the full list in making tied state triphones can not be obtained from the dictionary. We will only focus on these two differences. All other steps are the same as in the internal word triphone case.

Please copy the setup file “Tool\_trainTri\_xwd.dcf” from “Tools” folder to “exp” folder. This setup has exactly the same format and entries as the setup file for the internal word triphones. So, it can be called by the tool in the same way:

Tool\_trainTri(‘Tool\_trainTri\_xwd.dcf’);

The first different place is in the transcription preparation step:

Conf\_mon2tri: this is the configuration file path for the conversion between monophones to cross word triphones. Please copy the file “mktri\_xwd.led” from “files needed” to “toolconfs” folder, and specify the file path here. “mktri\_xwd.led” makes cross word triphones from monophones in the same way as in the example above.

The second different place is in the tied state triphone step:

Full\_list: since the cross word triphones can be formed by monophones across adjacent words. The full list of the system cannot be extracted from the dictionary. We need to find all possible cross word triphones in the task language in a brute force way, and put them in the list. Making this list requires linguistic knowledge. Given a list of all monophones, randomly combining three of them into a triphone is not a good way, because certain combinations are illegal in a language. These illegal triphones will become burden of the system since they require more training data, and will never be used in a recognition network. In a Chinese system, all possible triphones can be found by the rules of how Initials and Finals form a syllable.

Specifically, the first rule is that a syllable in Chinese is formed by an Initial and a Final. The Final always goes after the Initial. The pronunciation of a character is specified by a syllable, and a word is formed by characters. So, when creating triphones, if the middle phone is an Initial, then, its right context can only be a Final, and its left context can be either “sil” (if this character is the first character of a sentence) or a Final. Similarly, if the middle phone is a Final, then, its right context can be any Initial or “sil” (if this character is the last character of a sentence), and its left context can only be an Initial. Another rule is that there are only certain Finals which are allowed to append an Initial when forming a syllable. For example, for the Initial “j,” the Final “ong” is invalid to form a syllable, but “iong” is a valid Final for “j.” So, knowing the valid Final sets for each Initial, we can further rule out a lot of invalid triphones.

A full list called “fulllist\_xwd” made by these rules is provided in “files needed” folder. Please copy this file to “lists” folder. This list contains all valid triphones in Chinese made from the monophones covered by our task. There are one or two monophones not covered by our task database, but are valid Chinese monophones. So, this full list is not really 100% complete, but is appropriate for our task.

For a language who does not have too many monophones, such as English, it is fine (though not recommended) to brute force all possible combinations, regardless of any linguistic rules. A Perl script can be found in HTK samples\RMHTK\perl\_scripts\full\_list.prl, which generates all possible monophones, biphones, and triphones for a cross word system from a monophone list (the monophone list should remove sil and sp first). Minor change to the program can be made to output only those cross word triphones.

All the other entries in the cross word triphone setup file are the same as the internal word one, except some file or folder names are appended by “\_xwd” to distinguish from the internal word ones. So far, all the acoustic aspects of our ASR system have been completed.

4. Language modelling (Tool\_trainLM.m)

In the previous steps, we have generated a set of acoustic models. In this step, we will focus on language modelling. The tool for language modelling is “Tool\_trainLM.m,” and its setup file is “Tool\_trainLM.dcf.” Please copy these two files from “Tools” folder to “exp” folder.

Though, in this manual, language modelling is placed in the fourth step, it’s recommended that you run this tool apart from other steps. Do not connect this step with others. First, language model is not likely to change with other steps. We could have different features, or different acoustic models, but these changes won’t lead to any changes for the language model. So, it is usually the case in a large vocabulary ASR system that a language model should be prepared before other steps. Once it has been created, it does not change with other components. Second, a language model is a complete system by itself. It can be trained and evaluated isolated from any acoustic factors. It has its own training data and test data, both in text format. In other words, we do not need a complete ASR system to either train or tune a language model. So, people always train and tune up a language model towards different tasks without any acoustic data, and then put it in the whole system. Third, all language models for an ASR system comply with a set of standard format (ARPA format). There are various software specialized on language model analysis, such as SRILM and CMU\_Cam\_Toolkit. They provide much more powerful and flexible choices for making a language model than HTK. So, people always use such language-model-specialized software to generate better language models, and then connect them with other components of the ASR system in HTK environment. This connection is enabled by the standard format of language models.

No matter which software is selected, a background of language modelling is important. It is too long to give a detailed background in a manual for a complete system. Please refer to Chapter 14 of HTKbook for fundamental knowledge on language modelling. We will get into the task description and tool description directly.

In our example, the training data of the language model is a subset of the transcriptions of the acoustic wave files. The sentences in the wave files are divided into 4 groups (A,B,C,D). We use all speakers who speak A, B, C, and some speakers who speak D as the training data for the acoustic model, and use the rest of speakers from D as the test data. The language model training data is the transcriptions of A, B and C. But we also add the words in D into the vocabulary of our language model. There are many words in D that are unseen in A, B, C. These unseen words will still be assigned unigram probabilities according to the smoothing algorithm, which means in theory, it is at least possible for them to be correctly decoded.

This tool supports two methods. The first one uses the transcription of the acoustic data to build a simple bigram model. The second method uses plain text from any sources to create any n-grams specified by user. Let’s first look at method 1.

Please open “Tool\_trainLM.dcf.” Before getting into method 1, there are 4 global settings for the tool:

Trace\_on: y enables envision of progress on the screen; n turns it off.

Train\_on: y turns on LM training; n turns it off.

Convert\_on: this step is to convert an ARPA format LM to a lattice format. In the decoding step, the decoder Hvite works with the lattice format of a language model, not its ARPA format. For a detailed description what a lattice network is, please refer to HTKbook section 12.2. The other decoder HDecode works directly with the ARPA format. So, there is no need to make this conversion is HDecode is to be used in the next step.

Log: as in other tools, a progress report “progress\_trainLM.log” will be generated inside the folder specified here.

4.1. Method 1

Method 1 is a relatively simple method to create a LM. The source data usually comes from the transcriptions of the wave files, one transcription in one separate file, with one word in one row. The tool will first convert these “raw” transcriptions into a MLF file, and then, the LM is built from the MLF file.

One major restriction for this method is that we cannot pass it a predefined vocabulary. By default, this method uses all the words encountered in the transcriptions. The vocabulary is the first important step in building a LM. In theory, for those out-of-vocabulary (OOV) words in the source data, they should be either mapped to an unknown class, or simply thrown away. For those words in the vocabulary, but not in the source data, they will also be assigned probability based on various smoothing algorithm. Due to the restriction of not accepting the vocabulary, we will not use this method in our task. We will only show the meaning of the setting parameters of this method in the setup file.

Another major restriction for this method is that it only creates bigram models, and the smoothing algorithm is crude (a simple absolute discounting). Thus, this method is often used in phoneme recognition experiments, such as TIMIT. However, we still provide it in the tool as a convenient and quick method. Its settings are as follows. Some parameters are appended by a “1”. This is only to distinguish this method with method 2.

Method1\_on: y turns on this method; n turns it off.

Startword1 & Endword1: internally, when processing each sentence, a startword as specified here will be prefixed to the sentence. Similarly, an endword will be appended after each sentence. The startword and endword avoid confusion of counting the last word a sentence and the first word of the next sentence as a bigram. In our example, we use “SENT\_START” and “SENT\_END” as the startword and endword. Note that these words must be in the dictionary. Make sure they have the following entries:

SENT\_START [] sil

SENT\_END [] sil

OptString: this string specifies the unigram floor count and the bigram count threshold. A straightforward mathematical form of these parameters can be found in HTKbook, section 17.14.2.

Discount: this is the discount factor in the bigram. Each bigram count will subtract this factor to make room for the unseen events. Its mathematical form can be found in HTKbook, section 17.14.2.

datalist1: this is the only input for the LM, which is a list of all the transcription files. Please copy “LMdata\_trs.lst” from “files needed” folder to “lists” folder, and specify the file path here. Then, create a folder called “train\_word\_trs” inside “data” folder, and copy all the transcription files in the list into this folder. The list “LMdata\_trs.lst” contains all transcription files of A, B, C. Since this method cannot account for unseen words, these words will be totally missing from the bigram generated. So, many words in D (which is our test data) cannot be decoded correctly.

There are three output settings:

LM\_folder1: specify the folder to store the bigram model generated. This folder will be automatically created.

LM\_name1: specify the output bigram model file name (only the file name, not the full path).

wordlist: as a byproduct, a word list consisting of all the words in the LM training data will be output. In our example, the file path is “lists\wordlist\_abc.” This word list only contains words from A, B, C. Note that the startword and endword are also in the list.

4.2. Method 2

Now, let’s focus on method2. This method is much more sophisticated than method 1. It supports any n-gram specified by user, and also supports a user-defined vocabulary. All the OOV words are mapped to a unknown class called !!UNK. Any unseen words (which means the word is in the vocabulary but not in the training data) will also be assigned probabilities. So, at least, they get chances to be correctly decoded. From the setup file, it looks that this method is pretty simple. However, there are many intermediate steps in the method, which are hidden from the user. It’s recommended that a user should read through HTKbook section 15.1 to 15.3 to learn these intermediate steps. Now, let’s look at the setting parameters of this method. For those entries with a “2” appended, they are only to distinguish from method 1.

First, there are some global settings for this method:

method2\_on: y turns on method2; n turns it off.

Startword2 & Endword2: these are the start word and end word of each sentence. In this method, the source data format is different from that of method 1. Each row has one sentence. Each sentence starts with the startword, and ends with the endword, as specified by Startword2 and Endword2. These data can come from anywhere, not necessarily from the transcriptions. There is actually an important step called “data preparation” skipped in this manual. A huge amount of cleaning-up needs to be done in this step. For example, punctuations need to be removed, and digital numbers need to be converted to words. If the source data is downloaded from websites, the headers and hyperlinks are to be removed. In a word, only the pure “language” will be preserved. For Chinese, the character encoding scheme needs to be unified to GB2312, and sentences need to be segmented into words.

This tool assumes that this work has been done. In our task, the LM training data is provided in the file “863data\_abc” in the “files needed” folder. Please open it to see the “clean” format of the data. In our example, the startword and endword are still “SENT\_START” and “SENT\_END.” The training data is the transcriptions of A, B and C. In a large vocabulary task, the LM training data usually comes from various sources of human life, depending on the recognition task. The size of the training data easily achieves tens of Gigabytes. For our example task, we will use the transcription data for simplicity.

LM\_order: specify the order of the language model. For example, 2 means a bigram; 3 means a trigram. We will first build a bigram model.

DCtype: this is the type of discounting algorithm. HTK provides two types: TG for Good Turing method, and ABS for absolute discounting. Please refer to HTKbook Chapter 14 for details of these two algorithms.

cutoffs: this is the cutoff sequence for the language model. Cut-off is used to throw away those grams who appear infrequently enough. With cut-off, the model size can be greatly reduced, and more frequently observed shorter-context estimates can be made more robust. The cutoff sequence is used to specify a sequence of thresholds. A n-gram that appear more often than its threshold will be preserved when computing the probability estimate; otherwise, this n-gram entry will be thrown away. For example, if the language model order is 3 (a trigram), and the cutoff sequence is 1;1, this means all the bigrams who appear at least two times (greater than 1 time) in the training data will be preserved, so are the trigrams. The rest of them will be discarded. Note that there is no cutoff for a 1-gram (unigram). All the unigrams will be kept. So, the cutoff sequence is only effective when LM\_order is at least 2, and when that is the case, the length of the cutoff sequence must equal to LM\_order-1, and it is obvious that setting a cutoff factor to 0 means that all of the corresponding n-grams will be preserved. If the LM\_order is set to 1, then, the cutoff sequence can be set to any length with any number (since it loses effect for a unigram). In fact, when running this tool, a table will show up on the screen. This table counts how many n-grams will be left when different cutoffs are set. For example, in the following table, it says that a cutoff factor 0 for bigram will make 11181 entries, and a cutoff 1 will have only 1045 entries left. In our example, we set the cutoff to 0, which means we will preserve all the bigram counts.

cutoff 1-g 2-g

0 4488 11181

1 1610 1045

2 936 351

LM\_format: specify the format of the language model. The available choices are text format and binary format. In a large vocabulary task, the language model is usually very large. Setting it to “binary” will convert the bigram and any higher order grams to its binary format, and only leave the unigram part in its text format. Thus, the size of the LM will be much smaller. Setting this term to “text” will make the whole file in its text format.

Max\_vocab: this term specifies the maximum number of unique words the training data is allowed to have. If the training text contains more unique words than this number,an error will be given.

Next, there are two input files:

datalist2: this is the list of all the training data for the language model. Each file in this list complies with the same format: one sentence in one row, and each sentence starts with the specified startword, and ends with the endword. Please copy the file “LMdata\_text.lst” from “files needed” folder to “lists” folder, and put the file path here. Then, create a folder called “LMtext” inside the folder “data,” and copy the training data file “863data\_abc” from “files needed” folder to “LMtext” folder.

vocabulary: this is the vocabulary of the LM. Please copy the file “wordlist\_abcd” from “files needed” folder to “lists” folder. This list is manually extracted from all the sentences in A, B, C, D. There are many words in group D (which is our recognition task), which are unseen in the training data (A, B, C). But due to the smoothing algorithm, these words will be assigned equal probabilities as unigrams, so that they are still likely to be decoded correctly by the decoder. In many cases, the vocabulary is a subset of the training data. For example, it might be the top 20,000 most frequent words in the training data. In this case, those OOV words (out-of-vocabulary words) will be mapped to a unknown class called !!UNK in the generated LM. If “vocabulary” is set to “none” (case insensitive), this simply means that no vocabulary is provided. In this case, all the words encountered in the training data will be counted as in-vocabulary words, and are used to generated the LM.

Note that the vocabulary provided MUST include the startword and the endword as two entries. In our example, please notice that SENT\_START and SENT\_END are in the list “wordlist\_abcd” in addition to those “real” words in the text. However, the vocabulary must NOT include the entry !!UNK. This is because !!UNK is the unknown class marker to the system, which will be automatically generated, not an input word, since logically, a user cannot input an “unknown” word to the system.

There are two output settings:

LM\_folder2: this is the folder to store the output LM, which will be automatically generated if not existed.

LM\_name2: this is the output LM file name.

After running this tool, a bigram LM model “bigram2” will be generated inside the “LMs” folder. Please open this file. It’s interesting to notice that the !!UNK entry appears in the unigram. Theoretically, it shouldn’t be there, since all the words in the training data set (A, B, C) are covered by the vocabulary, which is formed by words from A, B, C, D. So, there should not be any unknown words. In fact, internally, when creating a LM, HTK will first assign each new word in the training data a number. All the following operations are based on this mapping. When a vocabulary is passed to HTK, the unknown class !!UNK will be added to this mapping forcefully, no matter there are any OOVs or not (this mapping file=vocabulary+!!UNK). So, since there is actually no !!UNK in the training data, but !!UNK is in the mapping file, it will be regarded as an unseen word, just as those “real” unseen words, which are in set D, but never appear in the training data composed of A, B, C. Thus, !!UNK becomes a unigram and is assigned a probability by the smoothing algorithm.

So, due to this issue, when you know ahead of time that there is no OOV words in the training data, just set “vocabulary” to “none” rather than leaving it there. Apparently, setting “vocabulary” to “none” in this case will result in a more accurate LM, since there will be no nuisance !!UNK, which will affect the estimates of other grams.

As a practice, you can change the LM\_order and cutoff sequence to create a trigram LM model, or even higher order.

4.3. Converting ARPA bigram model to a lattice network

The final step is to convert the ARPA format LM to the lattice format. In the decoding step, we will use two decoders respectively. One is Hvite, and the other one is HDecode. Hvite only works with a word lattice network, not the language model directly. But HDecode works directly with ARPA LMs (both bigram and trigram). So, this step is to prepare for Hvite. There is no need to make this conversion if HDecode is to be used.

For a detailed description on what a lattice network is, please refer to HTKbook section 12.2. The conversion from the ARPA format to the lattice format only works for bigram models. It does not work for any higher order models. Then, the question is: how does the decoder Hvite work with a trigram model, since trigram models can not be converted to the lattice format? Later on, you will know that Hvite will first use the bigram lattice network to do the first round of decoding. Then, another HTK tool HLrescore will be called to expand the output lattice for each sentence to incorporate the trigram and rescore each path in this network. In this processing, the ARPA format trigram will be passed to do the expansion. So, only the bigram model needs to be converted to the lattice format.

With this background, let’s look at how the tool implements this conversion. A point will be made ahead. In our example, as well as the results reported in a later section, we will not use the LMs generated by HTK. Instead, those results are based on LMs generated by SRILM (Standford Research Institute Language Modelling Toolkit), which is a dedicated toolkit for language modelling. This toolkit provides much more powerful and flexible options in making LMs than HTK. As mentioned before, because of the standard ARPA format, we can easily use LMs generated by other tools other than HTK in the decoder of HTK. Our task is an appropriate example of this kind. The only thing to do is to convert the ARPA bigram model into its lattice format if Hvite is to be used in the next step.

So, first turn off “Train\_on” and turn on “Convert\_on.” Then, please copy the two LM files “bigram\_abc” and “trigram\_abc” from “files needed” folder to “LMs” folder. These are the bigram and trigram models made by SRILM. Its training data is also the transcription sets A, B, C, and the vocabulary is also all the words in A, B, C, D. The smoothing algorithm is Witten Bell, which is not an option in HTK LM tools. Please open either of these two LMs. It might be noticed that there is no !!UNK in the LM, whereas in the HTK-based LM, there is. SRILM provides an option to turn off !!UNK. Let’s look at the setting parameters of the conversion step:

Startword & Endword: again, please specify the startword and endword in the LM to be converted.

Two input files are needed:

Bigram: this is the bigram file path. In our example, it is set to “LMs\bigram\_abc.” Recall that the conversion only works for bigram models.

wrdlist: a word list is needed. This word list must cover all the words in the LM, including the startword, endword, and !!UNK (if !!UNK appears in the LM). Please copy the file “wordlist\_abcd\_unk” from “files needed” folder to “lists” folder, and specify the file path here. Compared with the vocabulary file “wordlist\_abcd”, a !!UNK entry is added in the last row. In our task, the LM “bigram\_abc” does not have a !!UNK term. However, it does not matter that the word list has this extra term in this case, because the only requirement is that the word list needs to fully cover all the words in the LM. Extra words do not affect anything. However, if the LM “bigram2”, which is the one generated using method 2 in this tool were to be converted, then, the word list MUST contain !!UNK since this entry did appear in this LM.

A side point needs to be illustrated. As mentioned in method 2, sometimes, we know ahead of time that there are no OOV words in the training data. So, in order to avoid a nuisance !!UNK, we set “vocabulary” to “none.” Since we do not need to provide any word list in that step, we might forget to make one for the conversion step. Unlike method 1, method 2 won’t automatically generate a word list for us to use in the conversion step. So, when method 2 is selected, a warning will be given, reminding the user that a word list needs to be manually prepared no matter a vocabulary is necessary or not.

There are two output settings:

Network\_folder: specify the folder to store the output lattice file.

Network\_name: specify the lattice file name.

As specified in the setup file, a folder “Networks” will be automatically generated, inside which the lattice file “network” will be stored. This file will be used in the Hvite decoding step.

As a practice, please convert “bigram2,” which is the bigram generated by method 2 into its lattice format. If you do so, you might notice that in the output lattice file, there will be no !!UNK term, though !!UNK indeed is in the bigram model, as well as in the word list of the conversion step. This is because the decoder Hvite requires that all the words in the lattice network must have at least a pronunciation entry in the dictionary. However, what should be the pronunciation of the !!UNK? Some people use silence (sil) as its pronunciation. This solution does make the decoder work. But it does not make sense. Logically, !!UNK means a class of words not known to the system. They are out of the vocabulary, which defines all the words capable of being recognized by the system. Then, what should be the pronunciation of a word unknown to the system? No one actually knows. So, in this tool, we do not have an entry !!UNK in the pronunciation dictionary, since the system does not know its pronunciation. But to make Hvite work, we have to delete !!UNK from the lattice during the conversion. The other decoder HDecode does not have this problem. It is able to deal with !!UNK in a LM, and it works directly with the ARPA LM format. Also, there is no need to have the !!UNK entry in the pronunciation dictionary, which makes logical sense.

5. Decoding (Tool\_Decode.m)

With the acoustic and language models at hand, we are ready for the final step: decoding. The tool for the decoding step is “Tool\_Decode.m,” and its setup file is “Tool\_Decode.dcf.” Please copy these two files from “Tools” folder to “exp” folder.

This tool provides two decoders: Hvite and HDecode. Hvite is suitable for small and medium size vocabulary systems, and works better for monophone and internal word triphones. It becomes progressively inefficient as the size of the vocabulary grows and cross word triphones are used. HDecode is a dedicated decoder for large vocabulary systems. It only works with cross word triphones, which is the typical acoustic model type for any large vocabulary system. HDecode is much more efficient ( much lower real time factor ) than Hvite when cross word triphones are used in both decoders.

Please open the setup file "Tool\_Decode.dcf." Let's first look at the global settings for both decoders.

Trace\_on: 'y' displays the progress on the screen; 'n' turns it off.

Clean\_up: 'y' cleans up the old output MLF file (for both decoders), as well as the output lattice file (for Hvite) before new ones are generated. 'n' turns it off.

LogDir: a progress report "progress\_decode.log" will be generated in this directory. Note that the percentage accuracy is also written to this report.

Decode\_on: 'y' turns on decoding. 'n' turns it off.

Feat\_list: this is the feature file list path. It contains all the feature files to be decoded. Both decoders need this list.

Feat\_folder: this is the feature file folder. This entry is only for the use of Hvite. Along with the decoded MLF file, a lattice network for each feature file will also be generated. These lattice files will be used in decoding with a trigram ( Hvite works directly with a bigram only. To use a trigram, a lattice network based on the bigram decoding must be generated first ). They are placed in the same folder as the features. So, the tool will copy these lattices from the feature file folder specified by "Feat\_folder" to another folder called "Lattice\_folder" (specified later) . When the trigram decoding is turned on, lattice files will be loaded from "Lattice\_folder" ( the original ones in "Feat\_folder" will be deleted ). The other decoder HDecode works directly with a trigram model. So, there is no need to specify the "Feat\_folder" for copying lattices.

Startword & Endword: these are the startword and endword of each sentence in the language model. They are exactly the same as those specified in the LM generation step. These sentence start and end tokens need to be specified for both decoders. Note that in the dictionary, there must be two entries for these words. In our example, they are:

SENT\_START [] sil

SENT\_END [] sil

Result\_folder: this is the folder to store the output MLF file. Note that the percentage accuracy will be written the progress report "progress\_decode.log." The folder specified in the entry only stores the output MLF file.

Test\_trslist: this is a list of all the ground truth transcriptions of the test data for the computation of the recognition accuracy. Please copy the file "test\_wordtrs.lst" from "files needed" folder to "lists" folder and specify the path here. Then, create a folder "test\_word\_trs" inside "data" folder, and copy all the transcriptions of the test data into this folder. The transcription must be in its raw format: one transcription for one sentence, and one word in each row. The tool will first convert the raw format to its MLF format. In the conversion, each Chinese word will be split into characters, since people compute the character level accuracy in a Chinese ASR system. So, in the MLF file, there is one Chinese character in one row. If the task language is English, this "splitting" will lose effect. So, an English word is still an English word. There is no need to specify which language we are recognizing.

Next, let's look at the first decoder: Hvite

5.1. Hvite

Before we get into the settings, there are some general properties and restrictions of this decoder:

a. Hvite supports decoding with monophone, internal word triphone, and cross word triphone.

b. Hvite supports decoding with bigram LM directly. The LM must be in its lattice format.

c. To use a trigram, the output lattice for each feature file is expanded by the trigram, and each path in the expanded network is rescored. The path with the highest score is selected as the output. So, the trigram decoding is based on the result of the bigram decoding. The trigram is in its original ARPA format.

d. As mentioned in the language model generation part, Hvite does not work with the unknown word class !!UNK. So, when converting the bigram model to its lattice format in the language model tool, the !!UNK is deleted.

Now, let's look at the settings for Hvite (in addition to the global settings for both decoders).

Hvite\_on: 'y' turns on Hvite. 'n' turns it off.

HMM\_type: this is the underlying acoustic model type. 'iwd' for internal word triphones; 'xwd' for cross word triphones; 'mono' for monophones. In our example, this is set to 'iwd'.

Dict\_hvite: specify the dictionary file path. The word lattice network will be expanded into the underlying phoneme network by looking up this dictionary. So, the dictionary must contain all the words in the language model, including the sentence startword and endword. Please create a folder "dicts" inside "exp" folder, and copy the file "dict863\_tone\_sp1" from "files needed" into "dicts" if you have not done so in the triphone training step, and specify the file path here.

HMM\_folder\_hvite: this is the directory to load all HMM models. In our example, the HMMs are stored in "hmms\fhmmtri\_inword."

HMM\_list\_hvite: this is the list of all the HMMs. In our example, it is set to "lists\tiedlist\_inword."

Conf\_iwd, Conf\_xwd & Conf\_mono: these are the configuration files for decoding with internal word triphones, cross word triphones, and monophones. Please copy the files "hvite\_iwd.conf," "hvite\_xwd.conf" and "hvite\_mono.conf" from "files needed" folder to "toolconfs" folder, and specify the paths in the corresponding places. The tool will locate which file to use according to the "HMM\_type" specified above. Note that you do not need to specify all three files at the same time, because only one HMM\_type is used in each decoding processing. You can leave the other two places anything but empty, such as 'none.'

HviteOptstring: this string controls related parameters for Hvite. Frequently used ones are -t, -s, -p, -u, -v. In our example, '-t' is the pruning factor. It greatly affects the speed and accuracy of the decoder. Larger -t value leads to higher accuracy, but lower speed. Typical values for -t are between 200 to 250. '-s' is the language model scale factor, which also has significant impact on the accuracy of the decoder. Normally speaking, its value is affected by the size of the vocabulary as well as the size of the HMM set. Typical values for a large vocabulary system is between 12-15. Larger size of the vocabulary and HMM set leads to large value of -s. '-p' is the word insertion penalty factor. Normally, its value is fixed at 0. You can add -u and -v in the string to see what impact they will have on the accuracy. For the detailed meaning of these options, please refer to HTKbook section 17.23.

The followings are for bigram decoding.

Bigram\_on: 'y' enables decoding with a bigram. 'n' turns it off.

Network: specify the bigram lattice network file path as an input. As described above, Hvite only works with bigram lattice format, not ARPA format.

Rec\_output\_bg: specify the file name for the decoded MLF using a bigram. Note that only specify the file name here, not the full path. This file will be generated in the directory specified in "Result\_folder."

Lattice\_folder: specify the folder to store the output lattice for each feature file. These lattices will be used in the trigram decoding. In our example, it is set to be "..\data\Lattice." A folder "Lattice" will be automatically created inside "data" folder.

Lattice\_list: specify the file path for a list of all the lattices. This is an output. This list will be used in the trigram decoding.

The followings are trigram decoding settings:

Trigram\_on: 'y' turns on trigram decoding; 'n' turns it off.

Trigram: specify the trigram file path. In our example, it is set to "LMs\trigram\_abc." As described in the language model task, this trigram is built from training data A, B, C. But all the words in A, B, C, D are put in the vocabulary. Note that this trigram is in ARPA format, not in lattice format.

Latlist: this is the file path for the list of all the lattices generated in the bigram decoding step. Each lattice network in this list will be expanded by the trigram, and each path will be rescored ( only the language model score, the acoustic score remain unchanged). The path with the highest score in a network will be selected as the output sequence for the trigram decoding.

Conf\_rescore: this is the configuration file path for the trigram decoding. This configuration file will be automatically generated in the path specified.

HLrescore\_Optstring: this string has the same meaning as the "HviteOptstring." For a detailed list of options, please refer to HTKbook section 17.13.

Rec\_output\_tg: specify the output MLF file name for the trigram decoding. Again, this is only the file name. The file will be generated inside the folder specified by "Result\_folder."

Next, let's look at the other decoder: HDecode. This decoder is not included in the regular HTK package. An additional, more restrictive license must be agreed in order to download HDecode. HDecode can be downloaded from the official website of HTK. This manual assumes that it has been correctly installed.

Again, let's first look at the general properties and restrictions for this decoder.

a. HDecode is designed for large vocabulary task. It ONLY works with cross word triphones.

b. HDecode works directly with bigram and trigram models. The LMs are in the ARPA format.

c. sil and sp models are reserved as silence models. sil must be used as the pronunciation for the sentence start and sentence end tokens in the dictionary. sp is the short pause between words. sp is automatically added to the end of all pronunciation variants of each word in the recognition dictionary. So, each word in the dictionary MUST NOT have a sp appended.

d. Only the sentence start and end tokens (SENT\_START, SENT\_END in our example) are allowed to have sil as their pronunciations. sil can NOT appear anywhere else.

e. In the HMM set, only sil and sp are allowed to be monophones. Others must be cross word triphones.

f. HDecode works with unknown class !!UNK. So, the LMs are allowed to have !!UNK entries. !!UNK should not appear in the dictionary.

As stated in the Hvite section, Hvite also works with cross word triphones. However, Hvite is much less efficient than HDecode, especially for a large vocabulary task. Though Hvite has very close accuracy as HDecode, HDecode has a much lower real time factor. So, HDecode is strongly recommended for large vocabulary tasks with cross word triphone models.

Now, let's look at the settings for HDecode in the tool (in addition to the global settings for both decoders).

HDecode\_on: 'y' turns on HDecode. 'n' turns it off. Note that HDecode and Hvite can not be turned on at the same time. Only one decoder is allowed in one decoding processing.

Conf\_hdecode: this is the configuration file path for HDecode. This file will be automatically generated.

HdecodeOptstring: this is the operation string of HDecode. The options in this string have the same meaning as those in Hvite. Please refer to HTKbook, section 17.6 for details of all the available options.

Use\_bigram: 'y' turns on bigram decoding. 'n' turns it off. Note that HDecode works directly with bigram and trigram models. So, there is no need to run bigram decoding first in order to use a trigram model.

Use\_trigram: 'y' turns on trigram decoding. 'n' turns it off.

HMM\_folder\_hd: this is the folder to load HMMs. In our example, all the cross word triphones are stored in "hmms\fhmmtri\_xwd."

HMM\_list\_hd: this is the list of all HMMs to be loaded. In our example, the file path is set to "lists\tiedlist\_xwd."

LM\_bigram & LM\_trigram: these are the file paths for LMs. There should be two files "bigram\_abc" and "trigram\_abc" in the folder "LMs." HDecode works directly with the ARPA format, not the lattice format. Note that there is no need to specify the paths for both LMs. Depending on which order of LM is to be used (set by Use\_bigram and Use\_trigram), only the corresponding LM file path needs to be specified. The other place can be set to anything but empty, such as 'none' or '\'.

Dict\_hd: this is the dictionary file path for HDecode. Please copy the file "dict863\_tone\_nosp" from "files needed" folder to "dicts" folder and specify the path here. In this dictionary, all the sp's have been removed, and silence pronunciation (sil) only occurs in the sentence start and end tokens. Note that if your dictionary is copied from the forced alignment step, there should be an entry "SENT\_Boundary [] sil." Please remove this entry to make sure that sil only occurs in the sentence start and end entries.

Rec\_hd\_bg: this is the output MLF file name for bigram decoding. Again, this is only the file name. This file will be stored in the folder specified by "Result\_folder" in the global settings.

Rec\_hd\_tg: similarly, this is the output MLF file name for trigram decoding.

At this point, we've completed all the steps in this ASR system. The percentage accuracy for the test data can be found in the progress report "progress\_decode.log."

6. Experiment results

In this section, a series of experiments are conducted using this ASR toolbox. The database is the 863 Mandarin Chinese database. 78 women speakers from this database are used to train acoustic models. There are 2185 training sentences, which are divided into 4 groups: A, B, C, D, and each group of sentences are spoken by multiple speakers, resulting in 37116 utterances. 4 speakers from group D are included in the training data. The test data is formed by another 5 speakers from group D, resulting in 3125 utterances. For the language model, the training data consists of the transcriptions of A, B, C. But we put all the words in A, B, C, D into the vocabulary. There are many words in D that are unseen in A, B, C. So, according to the smoothing algorithm, these words are assigned equal probabilities as unigrams.

In Table 1, we present the Chinese character level percentage accuracy results using tonal phone acoustic models. All the models are 16-mixture models. Two feature types are used: baseline MFCC method, and the spectral/temporal method. The baseline MFCC uses frame length 25ms, and frame space 10ms, and has 39 features (12 DCTC coefficients+log energy+delta and acceleration terms). The spectral/temporal method uses 13 DCTC and 6 DCS (78 features), frame length 25ms, frame space 2ms, and block length 102ms (51 frames), block space 12ms (6 frames). As stated before, in forming triphones, the TB factor controls the degree of state tying, thus has great influence on the accuracy. The other factor used in state tying called 'RO' is fixed at 100. Also, in the decoding stage, the language model scale factor '-s' can affect the accuracy significantly. So, in Table 1, we also list the optimal values for TB and s wherever applicable. A bigram model is used for all the results in Table 1.

Table 1. Results for tonal phone acoustic models, LM=bigram

|  |  |  |  |
| --- | --- | --- | --- |
|  | Monophone | Internal word triphone | Cross word triphone |
| MFCC | 80.3% | 84.7% | 85.5% |
| FFT+DCTC/DCS | 82.4% | 86.1% | 87.5% |
| TB value | None | 2000 for both methods | 2000 for both methods |
| -s value | 10 for both methods | 12 for both methods | 12 for MFCC, 15 for FFT+DCTC/DCS |

In Table 2, we present the same results using base phone acoustic models. The models are 16-mixture models. The feature types are the same as in the previous experiments. The baseline MFCC still uses 25ms frame length, and 10ms frame space, 39 features. The spectral/temporal method uses 78 features (13 DCTC/DCS ) as before. The frame length is still 25ms, frame space 2ms, black length 102ms (51 frames); but the optimal block space changes to 14ms (7 frames). Again, the same bigram LM is used in this group of experiments.

Table 2. Results for base acoustic models, LM=bigram

|  |  |  |  |
| --- | --- | --- | --- |
|  | Monophone | Internal word triphone | Cross word triphone |
| MFCC | 75.9% | 81.5% | 82.5% |
| FFT+DCTC/DCS | 77.6% | 82.7% | 84.2% |
| TB value | None | 2500 for both methods | 2500 for both methods |
| -s value | 7 for both methods | 10 for both methods | 10 for both methods |

In Table 3, we compare three different pitch trackers: Yaapt, Yin and Praat. The acoustic models are 16-mixture tonal monophones. The baseline is still MFCC method (39 features without pitch). Pitch features are used in the comparison, resulting in 42 features (39 MFCC features+3 pitch features). Pitch is normalized by the mean and standard deviation of the whole sentence, and its delta and acceleration terms are incorporated. All voiced mode is used in the computation of pitch. The language model is still the same bigram model. The optimal '-s' value is 10 both all of them.

Table 3. Results using different pitch trackers, LM=bigram

|  |  |
| --- | --- |
|  | Tonal monophone |
| MFCC | 80.3% |
| MFCC+Yaapt | 83.1% |
| MFCC+Yin | 82.5% |
| MFCC+Praat | 82.7% |

Finally, in Table 4, we compare the results using bigram and trigram models. Similar to the bigram model, the trigram is built out of the training data A, B, C, and has all the words in A, B, C, D in the vocabulary. The acoustic models used are tonal internal word triphones and cross word triphones. The feature type used is 78 spectral/temporal features as in Table 1. The optimal TB and -s values are also listed in Table 1. The trigram -s value is the same as that of the bigram.

Table 4. Accuracy using bigram and trigram models for tonal triphones

|  |  |  |
| --- | --- | --- |
|  | Bigram | Trigram |
| Internal word triphone | 86.1% | 86.3% |
| Cross word triphone | 87.5% | 87.6% |

As expected, the trigram model does not help much to improve the overall accuracy. This is because many words in the test data are missing in the training data. So, these unseen words only exist in unigram format, not trigram. Perplexity can be used to evaluate two language models with the same vocabulary on the same test data set. Lower perplexity means a better language model. The LM evaluation is not included in the ASR toolbox. Please refer to HTKbook section 15.4 for details. As a practice, you can use the HTK tool LPlex to compute the perplexity of the bigram and trigram models used in our task example. It will be found that their perplexity on the same test data set is almost the same.

References

[1] S.J.Young, J.J.Odell,P.C.Woodland, "Tree-Based State Tying for High Accuracy Acoustic Modelling," Cambridge University, Engineering Department.