Lab 1: Data Preparation and Feature Extraction

University of Edinburgh

January 23, 2019

The main goal of this lab is to get acquainted with Kaldi[1], a state-of-the-art speech recognition toolkit. We will begin by creating and exploring a data directory for the TIMIT dataset. Then we will extract features for TIMIT upon which we can train a complete speech recognition system. Working with Kaldi often means spending a lot of time in the shell. Notes on UNIX commands are included in blue boxes; feel free to skip them if you're already familiar. *Most importantly, don't be afraid to ask questions when you get stuck.*

The work in this and future labs will use TIMIT. This corpus is interesting because it is *phonetically* labelled with 60 phone labels (and one end of sentence silence marker). TIMIT is often used as a benchmark for performance with new techniques, although results on TIMIT are not always transferable to other corpora. Some historical results are shown in Figure 1.

To be clear about what exact commands need to be run or written, commands that you should run are shown in a box with a red border, and notes in a box with a blue border:

```
Code to execute will appear in red boxes.
```

Notes will appear in blue boxes.

1 Kaldi setup

First, lets set up a local directory where we can run experiments. Open a terminal window on DICE. Change directory to a directory we've set up for you, inserting your UUN in place of <UUN>:

```
cd /afs/inf.ed.ac.uk/group/teaching/asr/Work/<UUN>
```

`cd dir` changes the directory to `dir`, and `ls` lists its contents. `cd ..` moves up one directory, `cd -` moves to the previous directory,
and `cd ~` moves to your home directory.

This is your work directory. It is tedious to remember that long path, so lets go back to your home directory, create a soft link called `asrworkdir`, and `cd` back into it:

[1] `www.kaldi-asr.org`

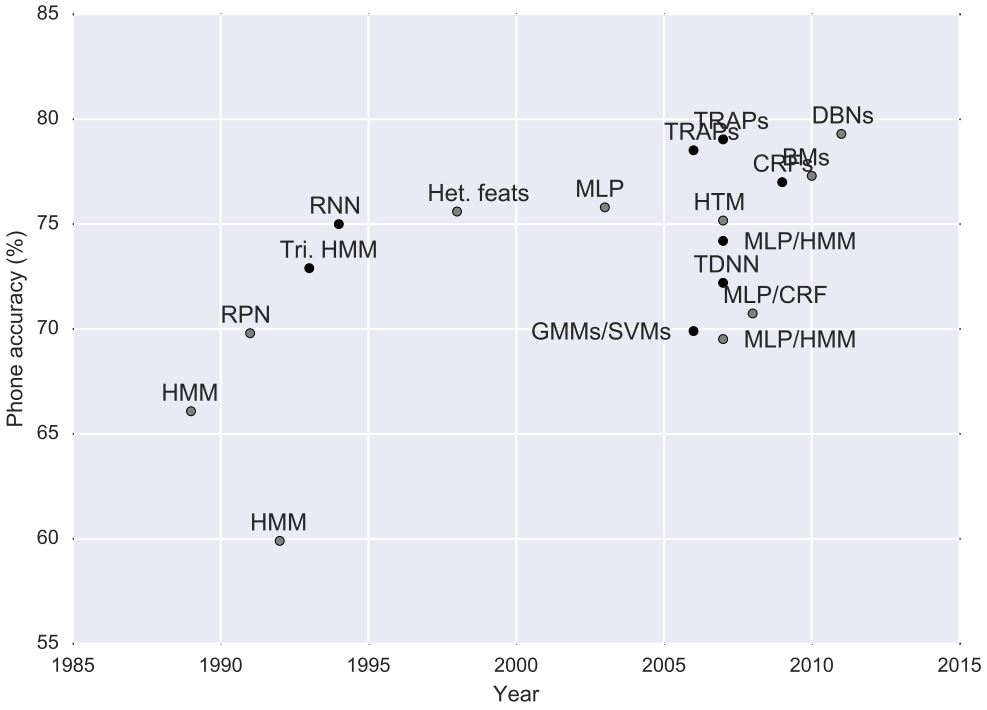


Figure 1: Results on TIMIT with various techniques. Most numbers from [1].

```
cd ~
ln -s /afs/inf.ed.ac.uk/group/teaching/asr/Work/<UUN> asrworkdir
cd asrworkdir
```

We'll now set up some files we need to run experiments in Kaldi. Run the following commands to create softlinks to your directory and to create a few empty directories. The dot means to create the links or copies in the current directory.

```
# First, we will create softlinks of some directories
ln -s /afs/inf.ed.ac.uk/group/teaching/asr/tools/labs/steps .
ln -s /afs/inf.ed.ac.uk/group/teaching/asr/tools/labs/utils .
ln -s /afs/inf.ed.ac.uk/group/teaching/asr/tools/labs/local .
ln -s /afs/inf.ed.ac.uk/group/teaching/asr/tools/labs/path.sh .

# Second, we will copy/make some directories so we can modify them
cp -r /afs/inf.ed.ac.uk/group/teaching/asr/tools/labs/conf .
mkdir data
mkdir exp
```

`ln -s f1 f2` creates a soft link from `f1` to `f2`, so that any changes made to one will affect the other. When you want to copy instead, use `cp`: `cp -r dir1 dir2` copies the directory `dir1` to `dir2`, the `-r` (recursive) flag is required for directories.

Your work dir now has a typical directory structure for Kaldi. Type the following command to list its contents

```
ls
```

You should see the following files and folders:

conf contains configurations for certain scripts that may read them. More on this later.

data will contain any data directories, such as a `train` and `test` directory for TIMIT. We will create these below.

exp contains the actual experiments and models, as well as logs.

local this directory typically contains scripts that relate only to the corpus we're working on (e.g. TIMIT). In this case it also may contain files we have provided for you.

path.sh contains the path to the Kaldi source directory

steps contains scripts for creating an ASR system

utils contains scripts to modify Kaldi files in certain ways, for example to subset data directories into smaller pieces

Kaldi is composed of a set of binaries (programmes) that perform particular tasks. These binaries can be strung together, and this is what provides Kaldi with much of its flexibility and is what most of the scripts do. The binaries are stored some place else than the work

directory. To access them from anywhere, we set (inside the file `path.sh`) an environment variable `KALDI_ROOT` to point to the Kaldi installation and add this to the system path (`PATH`). To set this variable type

```
source path.sh
```

or equivalently

```
. ./path.sh
```

To see whether it is set and where it points to run

```
echo $KALDI_ROOT
```

The command `echo` prints any string to the terminal along with any variables (e.g. `$KALDI_ROOT`). To omit newlines use the flag `-n`.

The `path.sh` file is called at the beginning of all Kaldi scripts, e.g. look at the first 18 lines of the script that computes MFCCs:

```
head -18 steps/make_mfcc.sh
```

`head -l file` and `tail -l file` prints the first and last `l` lines of `file`. A useful variation is `tail -n +l` which prints from line `l` to the end.

You should see a line at the end which sets the environment variables, *if path.sh exists*. It's a good idea to run this at the beginning of any Kaldi scripts.

`&&` will execute the next command if the previous succeeded (a typical Kaldi convention is using the opposite, `||` (double pipe), in its scripts, ending lines with `command || exit 1`, which means to exit the script with status 1 (error) if the preceding commands did *not* succeed).

Now that the environment variables are set, try to run a typical Kaldi binary *without any arguments*, e.g.

```
feat-to-dim
```

It should provide an explanation of its purpose and usage instructions. This is common to all Kaldi binaries and scripts.

Forgetting to run `source path.sh` if one of the most common mistakes. If you are getting errors like:

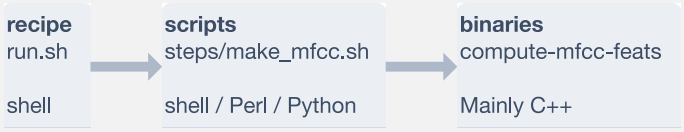
```
bash: feat-to-dim: command not found
```

Try to source path.sh and rerun the previous command.

Another common mistake is running commands from other directories than `~/asrworkdir`, if you get an error, make sure that you are in `~/asrworkdir` with the `pwd` command.

Other common errors can be found in Appendix 2.3.

Kaldi comes with recipes for various corpora. These are typically embodied in a `run.sh` script in the main directory, with supporting files in `local`. This script will call high level scripts in `steps` and `utils`, which in turn call binaries which perform the actual computation.



2 TIMIT

We will create a data directory for TIMIT and extract features. We will write the commands one-by-one into the shell.

If you're new to Bash scripting or need a refresher, here's a few resources you may find useful. The first three expect no previous knowledge of Bash, the last is good to get to know many useful commands.

- BASH Programming - Introduction How-To: `http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html`
- Advanced Bash-Scripting Guide: `http://www.tldp.org/LDP/abs/html/index.html`
- Bash Guide for Beginners: `http://www.tldp.org/LDP/Bash-Beginners-Guide/html/index.html`
- UNIX for Poets: `http://www.cs.upc.edu/~padro/Unixforpoets.pdf`

2.1 Data preparation

In the data preparation step we will create directories in `data` which will store any training and test sets, features and eventually a language model.

We create data directories for TIMIT by running the following two lines. Don't worry about warnings of nonzero return status. This might take a minute or two.

```
timit=/group/corporapublic/timit/original
local/timit_create_data.sh $timit
```

The data we just created is in the `data` directory. To appreciate better what this script does, navigate to the original TIMIT corpus training data directory and list its contents:

```
cd /group/corporapublic/timit/original/train
ls
```

It's split into multiple folders. Look at the first folder:

```
cd dr1
ls
```

Each of these directories represent a speaker. Move into the first speaker's directory and list the contents

```
cd fcjf0
ls
```

For each utterance there are four files: `.phn`, `.txt`, `.wav` and `.wrd`.

Look at each file in turn to figure out what they represent using the command `less`. You can use the up and down arrows to navigate. Hit `q` to exit.

```
less sa1.phn
less sa1.txt
less sa1.wav
less sa1.wrd
```

> `less` is useful when you only want to view a file and not edit it. It is also smart in that it doesn't read the entire file into memory at once, so files like `sa1.wav` which are not normally something you would look at, are handled neatly.

It's probably more interesting to listen to `sa1.wav`. On DICE you can run (but your computer might not have loudspeakers...)

```
play sa1.wav
```

Let's go back to our Kaldi work directory and see what we created with the command above:

```
cd ~/asrworkdir
```

Navigate to one of the created subdirs and look at the contents:

```
cd data/train
ls
```

The following files should be present. Have a look at each:

```
less text
less spk2utt
less wav.scp
less spk2gender
```

The script we ran has combined all the information from the TIMIT directory we just looked at into files that neatly contain the information in a way that Kaldi can work with it.

The files are closely related by *utterance* and *speaker* ids, abbreviated to *utt_id* and *spk_id* in Figure 2. The speaker information is used to pool statistics across utterances for speaker adaptation and for speaker specific scoring (if you have no speaker information, is it best to create one large "global" speaker for all the utterances, or to consider each utterance as from a different speaker?). There can also be recording ids, but in the absence of a `segments` file, which sets out what portions of each audio file should be used for an utterance id, the recording ids are equivalent to the utterance ids. In this case we use the entire length of each audio file set out in `wav.scp`.

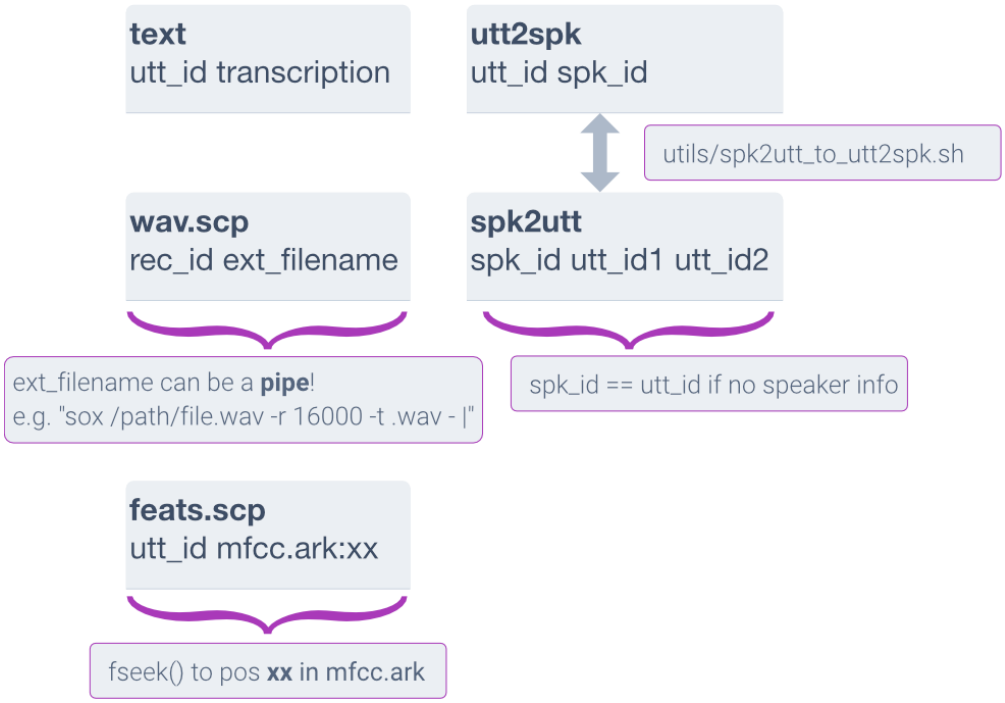


Figure 2: Illustration of a Kaldi data directory structure.

Change directory back to the main workdir:

```
cd ~/asrworkdir
```

To check that the data directories conform to Kaldi specifications, validate them by running the following two lines:

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

Uh oh. We're missing `utt2spk`, but we have `spk2utt`. These two files contain the same information, just with the mapping reversed. So we can easily convert one into the other. In the `utils` directory there is a file called `spk2utt_to_utt2spk.pl`. This is a Perl script which reads from `stdin` and writes to `stdout`. To pipe into the script we use `<`, to pipe out and into a file (instead of `stdout`) we use `>`. Run the following commands:

```
utils/spk2utt_to_utt2spk.pl < data/train/spk2utt > data/train/utt2spk
utils/spk2utt_to_utt2spk.pl < data/test/spk2utt > data/test/utt2spk
```

Run the validation scripts again. There should only be a `feats.scp` file missing, which we'll create next.

Have a look at the file you just created. How does it relate to `spk2utt`?

```
less data/train/utt2spk
less data/train/spk2utt
```

To see how many utterances there are in the training directory, we can use the command `wc`:

```
wc -l < data/train/utt2spk
```

- How many *speakers* are there in the training data?

2.2 Features

We'll now generate the features and the corresponding `feats.scp` script file, that will map utterance ids to positions in an archive, e.g. `feats.ark`.

For GMM-HMM systems we typically use MFCC or PLP features, and then apply cepstral mean and variance normalisation.

For the next step it can be handy to use a `for` loop, to loop over directory names. In Bash the syntax is:

```
for var in item1 item2 item3; do
  echo $var;
done
```

This will print:

```
item1
item2
item3
```

We will create MFCCs for our data. Run the following lines, which loops over the data directories and extracts features for each.

```
for dir in train test; do
  steps/make_mfcc.sh data/$dir data/$dir/log data/$dir/data
done
```

This will have created `feats.scp` with corresponding archives in a folder called `data/$dir/data` and written log files to `data/$dir/log`. (These are actually default directories and could have been omitted from the above command).

- You will now compute cepstral mean and variance normalisation statistics for the data. Find the appropriate script in the `steps` folder - perhaps using `ls steps/*cmvn*`. The run the script as above:

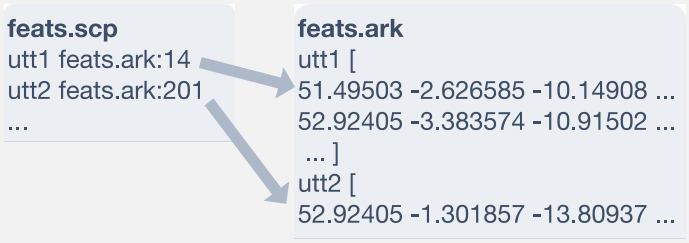
  ```
  for dir in train test; do
    steps/<insert-script-here> data/$dir
  done
  ```

  This will create `cmvn.scp` in each data directory.

- Validate the data directory again.

2.2.1 Script and archives (*.scp, *.ark)

`scp` files map utterance ids to positions in `ark` files. The latter contain the actual data. Kaldi binaries generally read and write script and archives interchangeably, as long as the filename is prepended with the type of file you wish to read or write, e.g. `scp:feats.scp` or `ark:mfcc.ark` or `ark:-` to write to stdout. Archives will be written in binary, unless you append the `,t` modifier: `ark,t:mfcc.ark`.



For more see the documentation on Kaldi I/O mechanisms, see: `http://kaldi-asr.org/doc/io.html#io_sec_tables`

Kaldi binaries typically read and/or write script and archive files. When this is the case, the usage message will show `rspecifier` or `wspecifier`. Scripts and archives represent the same data, so passing either to a program yields the same results.

Let's try using the programme `feat-to-dim` to find the dimensions of the features we just created:

```
feat-to-dim scp:data/train/feats.scp -
feat-to-dim ark:data/train/data/raw_mfcc_train.1.ark -
```

Are they the same?

Let's have a look at the actual features too. The archives are by default written in binary, but we can make a readable copy using the program `copy-feats` and a suitable write specifier (see box above). We pipe it into `head` to avoid overflowing the terminal window:

```
copy-feats scp:data/train/feats.scp ark,t:- | head
```

Do the features match what you got from `feat-to-dim`?

Read specifiers can take bash commands ending with a pipe (`|`) as arguments. This can be handy if you only want to look at the features for a particular utterance.

- Try replacing the read specifier `scp:data/train/feats.scp` in your previous solution with the following (what does this command do?). The single quotes are required because the entire string is now a read specifier.

  ```
  scp:'grep fdfb0_sx58 data/train/feats.scp |'
  ```

  > `grep` will search for a string in a file and output that entire line by default: `grep string filename`. The string could be a regex query and there are a lot of options. See `man grep` for more.

- Try the same trick as above, but find how many frames that utterance has using the program `feat-to-len`.

While *write specifiers* can write to `stdout` (e.g. `ark:-`), *read specifiers* can read from `stdin`. What does the following command do? This syntax is crucial to piping Kaldi programmes together.

```
head -10 data/train/feats.scp | tail -1 | copy-feats scp:- ark,t:- | head
```

`steps/make_mfcc.sh`, which you ran above, used the programme `compute-mfcc-feats` to extract features. This programme looks for `conf/mfcc.conf` in the `conf` folder for any non-default parameters. These are passed to the corresponding binaries.

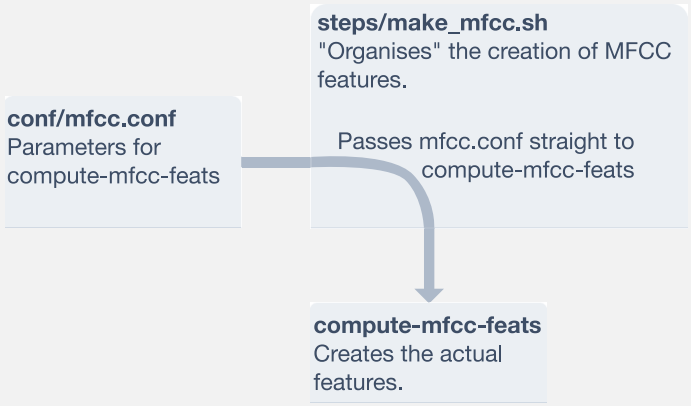
Look at that file

```
less conf/mfcc.conf
```

and compare it to the options for the program by running it without arguments:

```
compute-mfcc-feats
```

Configuration files in the `conf` folder are read by scripts from the `steps` folder. The contents of a configuration file is passed to a binary inside the script.
For example, `steps/make_mfcc.sh` will look for a file called `conf/mfcc.conf`. It will then read that file and pass the arguments to the binary `compute-mfcc-feats`. Hence, the arguments in `conf/mfcc.conf` should correspond to the arguments for `compute-mfcc-feats`.



If you have time, let's combine what we've learned and create filterbank features.

- Create copies of your data directories and generate filterbank and pitch features for each (look in the `steps` folder for a suitable script). However, first create a `conf/fbank.conf` file (using some text editor or see box below). Include an argument to set the dimension of the filterbank features to 40. (Hint: Look at `compute-fbank-feats` for arguments). You will also need to create a `conf/pitch.conf` file, but this can be empty. Finally, check the feature dimension and make sure it is 43 (there are three pitch features).

To open or create a file in `nano`, type

```
nano conf/fbank.conf
```

Inside `nano`, use the arrow keys to move around the text file. To exit, hit `ctrl+X` and hit `Y` or `N` to the question of whether to save any changes or not. Other commands are listed at the bottom of the window.

We're done! Next time we'll build a GMM-HMM system.

2.3 Appendix: Common errors

- Forgot to source `path.sh`, check current path with `echo $PATH`
- No space left on disk: check `df -h`
- No memory left: check `top` or `htop`

- Lost permissions reading or writing from/to AFS: run `kinit && aklog`. To avoid this, run long jobs with the `longjob` command.
- Syntax error: check syntax of a Bash script without running it using `bash -n scriptname`
- Avoid spaces after \when splitting Bash commands over multiple lines
- Optional params:
- command line utilities: `--param=value`
- shell scripts: `--param value`
- Most file paths are absolute: make sure to update the paths if moving data directories
- Search the forums: `http://kaldi-asr.org/forums.html`
- Search the old forums: `https://sourceforge.net/p/kaldi/discussion`

2.4 Appendix: UNIX

- `cd dir` - change directory to `dir`, or the enclosing directory by `..`
- `cd -` - change to previous directory
- `ls -l` - see directory contents
- `less script.sh` - view the contents of `script.sh`
- `head -l` and `tail -l` - show first or last `l` lines of a file
- `grep text file` - search for `text` in `file`
- `wc -l file` - compute number of lines in `file`

References

[1] Carla Lopes and Fernando Perdigao. Phone recognition on the timit database. *Speech Technologies/Book*, 1:285–302, 2011.