Problem Set 4: FSTs, LMs, and Kaldi

The University of Texas at Austin
CS395T/EE381V: Spoken Language Technologies
Fall 2022
Instructor: David Harwath

Assignment Date: October 4, 2022

Due Date: October 13, 2022

This problem set consists of 5 problems, totaling 100 points.

You will explore:

1. Working through several written problem on FSTs and LMs **(50 points)**
2. Implementing and evaluating a speech recognition system with Kaldi **(50 points)**

To turn in: You should submit a single .pdf writeup to Canvas, containing all of your deliverables for all parts of the problem set. For the written parts of the assignment, you must show your work. For the coding parts of the assignment, you must turn in all of the plots and code fragments that are specified as deliverables in the corresponding exercises.

Collaboration Policy: You may collaborate in groups of no larger than 3 students, and each student must turn in their individual writeup written in their own words. **You must clearly indicate all of your collaborators in the title block of your writeup.**

Late policy: Sometimes we have bad days, bad weeks, and bad semesters. In an effort to accommodate any unexpected, unfortunate personal crisis, I have built "time banks" into our course. You do not have to utilize this policy, but if you find yourself struggling with unexpected personal events, I encourage you to e-mail me as soon as possible to notify me that you are using our grace policy. You may use this policy one of two ways (please choose, and let me know): You may have a two-day grace period for one assignment, OR You may have 2 one-day extensions for two different assignments. Late assignments will otherwise not be accepted.

Exercise 1 (10 pts)

Draw the result of determinizing and minimizing FSA A.

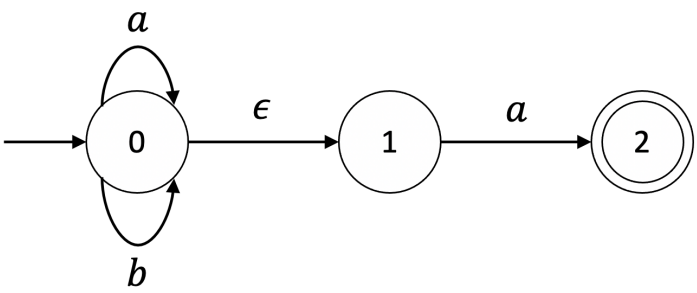

Figure 1: FSA A

Exercise 2 (10 pts)

Indicate whether FSA B is equivalent to FSA C. Recall that two FSAs are equivalent if they recognize the same language.

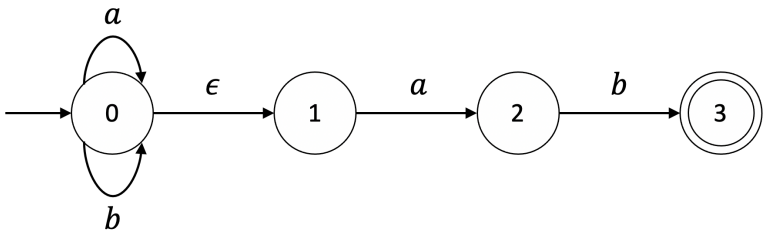

Figure 2: FSA B

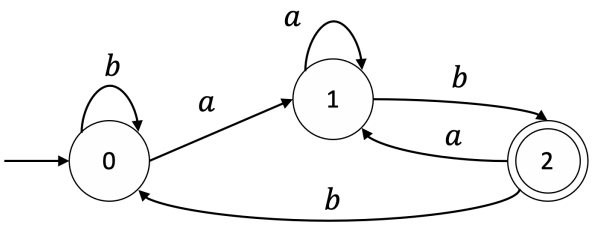

Figure 3: FSA C

Exercise 3 (10 pts)

Draw the FST that is the result of composing FST P with FST L, i.e. $P \circ L$.

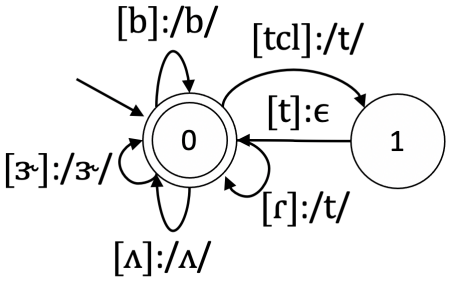

Figure 4: FST P

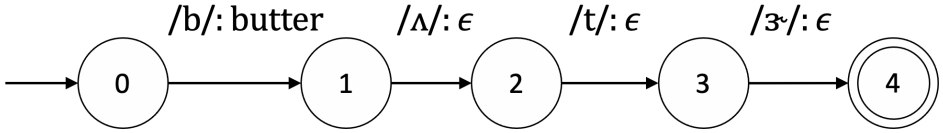

Figure 5: FST L

Exercise 4 (20 pts)

Consider the following 3-sentence "corpus" of text data:

```
navigate to san Francisco by car
directions to san diego by bike
san diego directions by car
```

Draw a weighted FSA that expresses a bigram language model estimated over this text "corpus" using maximum likelihood. Your FSA should use the Probability Semiring, and does not need to include transitions that have a weight of 0. You can leave all transition weights in fractional form.

Exercise 5 (50 pts)

This exercise will walk you through building a basic speech recognition system using the Kaldi toolkit (`www.kaldi-asr.org`). Kaldi is probably the most widely-used speech recognition software toolkit in the world today, both in industry and in research. Several of Kaldi's biggest strengths are:

1. Flexibility. Kaldi is highly modular, with most of its core functionality implemented by a large number of software binaries executable from the command line. Each of these binaries does a specific job, such as extracting MFCCs, performing E-M training of a GMM model, or decoding through an FST. These tools are typically combined in shell scripts to form "recipes" that specify how to train a fully functional ASR system start to finish.
2. Performance. Kaldi's implementations of all of the core algorithms used in ASR are highly accurate, generally free of bugs, and support all of the state-of-the-art tricks necessary to get good performance. Furthermore, Kaldi also ships with recipes for all of the major speech recognition datasets. These recipes typically work right out of the box, and achieve close to state-of-the-art performance on many tasks.
3. Speed. Speech recognition is computationally intensive. The majority of the Kaldi code is written in C++ that has been highly optimized, and Kaldi was written from the ground-up with parallelism in mind (both over a compute cluster as well as GPUs).

We have set up a Kaldi environment for you to use on the UTCS lab machines; the list of lab machines can be found at `https://apps.cs.utexas.edu/unixlabstatus/`, and you will need to use your CS computing account in order to ssh into them.

Note: you will need approximately 2 GB of free disk space in your working directory in order to complete this lab. CS home directories by default have a 10 GB quota, but if you have used all of that space already you might want to clean your directory or move data up elsewhere, such as UT Box.

First, you should download the `lab4_kaldi.tar.gz` file from Canvas and uncompress it wherever you would like to work on the lab (by default, this should probably be in your CS home directory). Next, `cd` into the `lab4_kaldi` directory and print its contents with `ls`. You will see several files and directories:

- `cmd.sh` : A file that specifies how Kaldi should run compute-heavy jobs. By default this file will specify `run.pl` to be used for all steps, which just indicates that the job should run on the local host machine. If you were on a compute cluster using a workload manager such as Slurm, you'd change this script to use slurm.pl, which would automatically dispatch all compute jobs to the Slurm cluster instead of running them locally.
- `path.sh` : This file sets all of the necessary environment paths, and specifies where the Kaldi binaries can be found.
- `run.sh` : This is the main "recipe" script that will construct the speech recognizer from start to finish.
- `conf` : This directory contains configuration files for things like feature extracting or decoding. If you wanted to change the number of MFCC features to extract or the Viterbi

beam width to use during decoding, you'd do it within the configuration files in this directory.

- `local` : A directory of scripts that are specific (local) to this particular recognizer recipe. Typically these will deal with things like processing the input data into Kaldi's standard format.
- `steps` : A symlink to a directory of scripts that execute commonly used steps in building a speech recognizer, such as extracting features, training acoustic models, and decoding. These scripts are often shared across different recognizer "recipes" for different datasets.
- `utils` : A symlink to a directory of utility scripts that often function as "helpers" to the scripts in `steps`

During the course of building the speech recognition system, several other directories will be created by the `run.sh` script:

- `data` : This directory will hold processed data. Sub-directories within here typically correspond to different data splits (for example, train, dev, and test), but language models and lexicons will also reside in here (typically these directories are prefixed with 'lang').
- `exp` : This directory will hold the outputs generated by the various steps invoked by the `run.sh` script. Generally, the sub-directories in here correspond to trained ASR models.

We have provided you with an initial `run.sh` script, and the remainder of this exercise will walk you through completing it. **Important note:** it can easily take 20-30 minutes for the completed `run.sh` script to run start to finish on the CS lab machines. You will probably find decoding to be the most time-intensive step.

The initial script: Open up `run.sh`. Notice that the control flow of the script is organized in "stages," allowing you to set the `stage` variable at the top of the script to determine where the script will start executing. You can use this feature to run the initial script, fill in the next stage block and update the stage variable, run the script again, and so on without having to re-run all of the stages that have already been run. The following stages are used in `run.sh`, with the ones that you will fill in marked "(TODO)":

- Stage 0: Copy the pre-trained language models over to `data/local/lm`
- Stage 1: Prepare the source data by putting it in the Kaldi format
- Stage 2: (TODO) Compute MFCCs and CMVN statistics for the acoustic data
- Stage 3: (TODO) Train a monophone HMM-GMM recognizer
- Stage 4: (TODO) Train a triphone HMM-GMM recognizer
- Stage 5: (TODO) Test the triphone recognizer

MFCC Extraction : You will be working with a small subset of the LibriSpeech corpus, which is a very popular dataset for speech recognition research comprised of public domain audiobooks. We will only be using a training set (called `train_clean_5`) and a testing set (called `dev_clean_2`)

For both of these datasets splits, we will need to 1) compute MFCCs and then 2) compute Cepstral Mean and Variance Normalization (CMVN) statistics. You can use the `steps/make_mfcc.sh` and `steps/compute_cmvn_stats.sh` scripts to do this. Try running these files without

arguments (or open up them up and examine the usage string) to get an idea of what command line arguments they are expecting. For our purposes, the default settings of the optional arguments will be fine, but you'll need to specify the `data-dir` (the input source data directory, for example `data/train_clean_5`), `log-dir` (where logfiles are stored (`exp/make_mfcc` is a good place to specify), and the output directory where the computed MFCCs will be stored (you can just use the value of the `$mfccdir` variable that is specified in the `run.sh` script, which will store the MFCCs in a directory named `mfcc`).

Training the monophone HMM-GMM system : Next, we'll train a monophone ASR system from a "flat start." This is a common approach for initializing the frame-to-HMM-state alignments for E-M training when all we have to start with are the word-level transcriptions for each utterance, and a pronunciation lexicon. The word-level transcriptions are converted into phonetic transcriptions (via the lexicon), and then the frames of the utterance are equally divided among the phones (and further equally divided among the states of each phone HMM) For example, if an utterance is 0.6 seconds long and contains the single word "bot," the first 0.2 seconds are assigned to the [b] phone, the middle 0.2 seconds are assigned to the [a] phone, and the final 0.2 seconds are aligned to the [t] phone. These alignments are used to train the very first E-M iteration of the HMMs and GMMs. We can then can go back and re-compute the alignments, re-estimate the HMM and GMM parameters, and continue in this fashion until convergence.

The flat-start initialization is most effective when performed with short utterances, as it minimizes the amount of mis-alignment that is bound to occur compared with using longer utterances. For this reason, we will use the `utils/subset_data_dir.sh` script to select the 500 shortest utterances in the training set, and create a new data split containing them in the `data/train_500short` folder. Because we're taking a subset of an existing data folder (namely `data/train_clean_5`) that we've already extracted MFCCs for, you don't need to re-extract MFCCs for the 500 utterance subset.

The only other step we need to take in this stage is to use the `steps/train_mono.sh` script, which actually does the training of the monophone HMM-GMM system. Examine the input arguments for that script:

- <data-dir> is the directory of the source data we want to train on. For this case, you'l want to use `data/train_500short`
- <lang-dir> is the directory containing the pronunciation lexicon and corresponding FST symbol tables that we need to use in order to build the HMMs that will be used for training (which correspond to the ground-truth transcripts). You'll want to use `data/lang_nosp` (the "nosp" means there are no pronunciation probabilities in Kaldi-jargon) directory for this argument.
- <exp-dir> is the actual output directory that the final monophone model (and all of the associated log files produced during training) will be written. `exp/mono` is a good place to put this.

You should also specify the `–boost-silence 1.25` optional argument to `steps/train_mono.sh`, which boosts the score of the silence phone during alignment. Because most utterances begin and end with a small amount of silence, this helps the silence phone capture these silences rather than having them become aligned with another phone model.

Training the triphone HMM-GMM system : This stage will require two steps that need

to be done in sequence. First, you will need to use the monophone model you just trained on the 500-utterance subset of the training data to force-align the entire training dataset. This will effectively do a Viterbi alignment of the acoustic frames of each training utterance to the states of the HMM formed by concatenating the word-level HMMs (which are themselves comprised of the phonetic HMMs) that make up the transcript of each utterance. This will provide the initial alignment for the triphone HMM-GMM training which will be done on the full training dataset. To do this, you should use the `steps/align_si.sh` script. You'll again want to specify the `–boost-silence 1.25` optional argument, but this time you should specify the full training dataset (`data/train_clean_5`) as the `<data-dir>`. You can use the same `<lang-dir>` as you did during monophone model training. The `<src-dir>` argument to the `steps/align_si.sh` script should be the directory that the model used to perform the alignment is stored. In the case that you stored your monophone model in `exp/mono`, you should use that directory. Finally, the `<align-dir>` argument specifies the directory that the output alignments will be written to. A good place for this would be `exp/mono_ali_train_clean_5`.

Next, you'll want to do the actual training of the triphone system. The `steps/train_deltas.sh` will take care of the entire triphone model training process, including constructing the context-dependent phonetic decision tree. The usage of this script is very similar to that of `steps/train_mono.sh`, but with a few additional arguments that specify the parameters for the context dependency tree. The `<num-leaves>` argument specifies the maximum allowed number of leaf nodes for the phonetic context dependency tree (i.e. the total number of different GMMs we will allow), and the `<tot-gauss>` parameter specifies the sum total number of Gaussians to be used across all of the GMMs associated with each leaf node. The script will intelligently allocate these Gaussians across the different leaf nodes in proportion to the number of acoustic frames assigned to each leaf node. In other words, leaf nodes representing extremely rare triphone states might only get assigned 1 or 2 Gaussians for their GMM, while very commonly observed states might get 10 or 20 Gaussians for their GMM. Try using 2000 leaves and 10000 Gaussians. For the `<data-dir>` argument, you should specify the directory of the full training dataset, but you can use the same `<lang-dir>` you used for monophone training. The `<alignment-dir>` should be the directory that you just created when calling `steps/align_si.sh`, and the `<exp-dir>` should be the directory you want to store the final triphone HMM-GMM model in; `exp/tri1` is a good place for this.

Decoding with the triphone system : Now we're ready to evaluate our triphone system. We will be decoding with a small language model, and then rescoring the lattices with a larger language model.

The first step is to construct the HCLG graph that we learned about in lecture. This is handled by the `utils/mkgraph.sh` script. It takes as input a language model directory and an acoustic model directory: the `<lang-dir>` should be `data/lang_nosp_test_tgsmall` (which was created during the data preparation stages and contains the small language model), and the `<model-dir>` should be the triphone acoustic model directory `exp/tri1`. The output argument is the `<graph-dir>`, which is where the HCLG graph will be written. `exp/tri1/graph_tgsmall` is a good place to put it.

Next, we will run the actual decoding using `steps/decode.sh`. This script takes three arguments: the graph directory you just created, the dataset directory, and the output directory. Instead of the training set, we'll want to specify the test set data directory `data/dev_clean_2`. Finally, you should write the decoding output to `exp/tri1/decode_tgsmall_dev_clean_2`.

The very last step is to perform rescoring with a larger language model. The `steps/lmrescore.sh` script will take care of this. The script will need to know the language model originally used in the first pass of decoding (`<old-lm-dir>` which should be `data/lang_nosp_test_tgsmall`), the new language model you want to use (`<new-lm-dir>`), the data directory you want to decode, the directory that contains the first pass recognition lattices (`<input-decode-dir>`, which should have been `exp/tri1/decode_tgsmall_dev_clean_2`), and where you want to write the second pass recognition lattices (`<output-decode-dir>`). You should specify `data/lang_nosp_test_tgmed` as the new language model directory, and `exp/tri1/decode_tgmed_dev_clean_2` as the new output directory.

Printing Word Error Rates : Recall that in theory, the decoding operation finds the word sequence that maximizes $P(A|W)P(W)$. In practice, however, the acoustic model likelihoods $P(A|W)$ tend to have a much larger dynamic range than the language model likelihoods. When using inadmissible search algorithms (as we nearly always are forced to do), this can result in the language model not having much influence over the recognition results. To combat this, we often apply a language model weight factor, and instead try to find the word sequence that maximizes $P(A|W)P(W)^{\beta}$. To choose what β should be, we usually use two test datasets: a development testing set ("dev" set) and an evaluation testing set ("eval" or "test" set). We decode the dev set with a range of β values, choose the one that achieves the lowest WER on the dev set, and then use that value to decode the final test set. For simplicity, in this lab we're only working with a single test set, so you can just report the best WER you achieved on that set across all values of β. There is a script that can do this for you, `utils/best_wer.sh`. For example, to print the word error rate after rescoring with the larger language model, from your working directory simply type

```
cat exp/tri1/decode_tgmed_dev_clean_2/wer* | utils/best_wer.sh
```

which will print a word error rate, as well as a breakdown of the types of errors made (insertions, deletions, and substitutions).

To turn in: You should turn a copy of your `run.sh` script, and report the best word error rates (WER) achieved with your triphone system when decoding both with the small LM, and then rescoring with the medium LM.