ASR Lab 1
Building HMMs with OpenFst

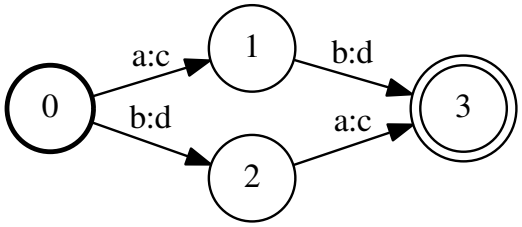
Peter Bell and Andrea Carmantini

January 21, 2020

1 Introduction

In this practical we will use OpenFst (http://openfst.org) to build and manipulate finite-state transducer based representations of HMMs. Weighted finite state transducers (WFSTs) are used as the basis of many modern ASR systems, including by the state-the-art Kaldi toolkit.

WFST-based ASR will be covered in detail in later lectures, but for now, all that you need to know is that WFSTs are a graph-based formalism consisting of states and (directed) arcs between them. Arcs have an input label, an output label and optionally, a weight (or cost). The WFST structure *transduces* a sequence of input labels to a sequence of output labels (and optionally, applies a cost to this).

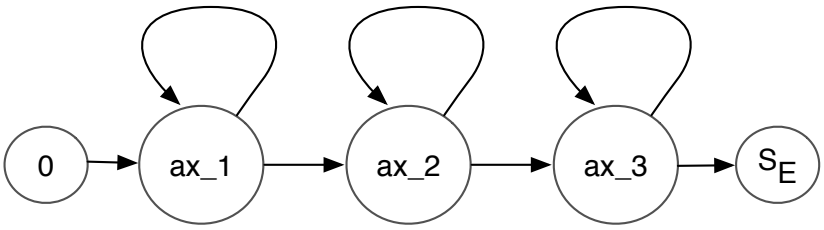


A very simple transducer mapping the string "ab" to "cd" and the string "ba" to "dc". The initial state is shown in bold; the final state is shown with a double circle.

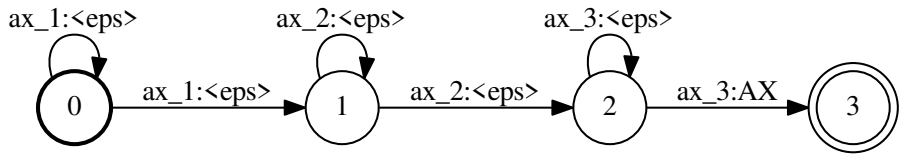
2 The HMM in WFST form

Some small changes are needed to represent the HMM as an FST. It turns out to be easier to think of the HMM emitting states as being on the arcs. The input labels can be used to denote the state ID, whilst the output labels can be used to encode labels to be output by the recogniser, such as words or phones. Note that the epsilon symbol, ε (or `<eps>`), is conventionally used to indicate an empty input/output symbol, meaning that no symbol was consumed/produced by that arc.

This example shows how the 3-state HMM for the phone "AX", containing emitting states that we will label `ax_1`, `ax_2` and `ax_3` can be represented in WFST form. (The non-emitting initial and final states are denoted by 0 and S_E).



Conventional HMM for phone "AX" with three emitting states



WFST representation of the HMM above. Note the output label "AX"

3 Getting started

We will use the Python wrapper to OpenFst. Documentation can be found at `http://www.openfst.org/twiki/bin/view/FST/PythonExtension` and also by using Python's `help()` function.

On a DICE machine, run:

```
virtualenv -p python3 asr_labs/asr_env
source asr_labs/asr_env/bin/activate
pip install openfst-python jupyter
cd asr_labs
git clone https://github.com/Ore-an/asr_lab1.git
cd asr_lab1
jupyter notebook asr_lab1.ipynb
```

If you're using your own machine, ensure you have Python 3 and virtualenv installed, then run those same commands.

4 Creating FSTs in Python

We start by importing the OpenFst Python wrapper

```
1 import openfst_python as fst
```

In this example we'll build the simple WFST above. Internally, OpenFst stores symbols in integer form. We need to create symbol tables that associate each possible input and output symbol with an integer.

```
1
2 input_sym = fst.SymbolTable()
3 output_sym = fst.SymbolTable()
4
5 input_sym.add_symbol('<eps>') # by convention, <eps> always
6                                # has symbol zero
7
8 input_sym.add_symbol('a')     # input symbols
9 input_sym.add_symbol('b')
10
11 output_sym.add_symbol('<eps>')  # output symbols
12 output_sym.add_symbol('d')
13 output_sym.add_symbol('c')
```

A `SymbolTable()` is simply a table associating symbols and indexes. We add symbols to the table with the method `add_symbol()`.

Now that we've got our symbol tables, we will build the FST itself:

```
1 f = fst.Fst()
2 f.set_input_symbols(input_sym)
3 f.set_output_symbols(output_sym)
```

Our FST transduces the input to the outputs, so we set the symbol tables as such. Now, we want to add a number of states:

```
1 s0 = f.add_state()
2 s1 = f.add_state()
3 s2 = f.add_state()
4 s3 = f.add_state()
```

The output of the `add_state()` method is just the index assigned to the state, but it can be useful to assign that index to a variable to give it a more meaningful label.

To create arcs between states, we do:

```
1 a = input_sym.find('a')
2 b = input_sym.find('b')
3 c = output_sym.find('c')
4 d = output_sym.find('d')
5
6 f.add_arc(s0, fst.Arc(a, c, None, s1))
7 f.add_arc(s0, fst.Arc(b, d, None, s2))
8 f.add_arc(s1, fst.Arc(a, c, None, s3))
9 f.add_arc(s2, fst.Arc(b, d, None, s3))
```

The syntax for the method `add_arc` is:

`add_arc(<source state>, <arc to add>)`

while the syntax for initializing a class `Arc()` is:

`Arc(<input symbol index>, <output symbol index>, <weight>, <destination state>)`

We use the `find()` method of the symbol tables to get the index of a certain label.

Now we just add start and end states:

```
f.set_start(s0)
f.set_final(s3)
```

And our first WFST, shown in the example above, is done! It can be seen visually in the Jupyter notebook.

5 Exercises

1. Create WFSTs representing the following HMMs:
   (a) A 3-state left-to-right phone HMM with self-loops, for an arbitrary phone of your choice
   (b) The parallel-path left-to-right model shown on Slide 10 of Lecture 2
   (c) An ergodic HMM with n states (you will need to think about how to handle final states)
2. Using the provided phonetic dictionary in `lexicon_lab1.txt`, generate an HMM in WFST form for the word "peppers", by concatenating 3-state phone HMMs. You can use `p_1`, `p_2`, ..., `eh_1`, ... to denote the phone states. The word should appear as an output label.
3. If you haven't already, make your solution to 2. more general by writing a function to generate an HMM for any word in the lexicon, using n states per phone:

```
def generate_hmm_wfst(word, n):

    f = fst.Fst()
    ...
    return f
```

   You might want to also write functions to retrieve word pronunciations and a list of phones from the lexicon
4. Generate an HMM that can recognise:
   (a) any single phone contained in the lexicon

(b) any sequence of phones contained in the lexicon

5. You have decided that the best way to start teaching a friend English is to have them learn the notorious tongue-twister *"peter piper picked a peck of pickled peppers"*. Using the provided dictionary, create an HMM that can:

   (a) recognise any one of these words

   (b) any sequence of these words

If you have more time

Probabilities in WFSTs are traditionally expressed in negative log format, that is, the weight w on an arc transitioning between states i and j is given by $w = -\log a_{ij}$, where a_{ij} is the HMM transition probability.

5. Add weights to your WFSTs corresponding to transition probabilities. Assume that the probability of a self-loop is 0.1, and that when transitioning *between* separate phones or words, the probabilities are uniform over all transitions.