Lecture 6: OpenFST

Lecturer: Mark Hasegawa-Johnson (jhasegaw@uiuc.edu)
TA: Sarah Borys (sborys@uiuc.edu)

January 14, 2009

Contents

1 Introduction: Finite State Transducers

A hidden Markov model can be viewed as either a finite state machine, or a dynamic Bayesian network. This lecture will discuss finite state machines.

The previous lectures described how we can model each context-dependent phone with a mixture Gaussian hidden Markov model. Typically, the triphone states are clustered until we have perhaps 1000-2000 different states. The possible transitions among states are specified by the transition probability matrices of the triphone models. The possible transitions among triphones are specified by the dictionary. The possible transitions among words are specified by the language model. During recognition ("search"), the recognizer must compute, at each time step, the probabilities of the N highest paths leading up to time t, $p(Q_1(t)), \ldots, p(Q_N(t))$, where $Q_i(t)$ is the path up to time t that has the ith highest probability:

$$Q_i(t) = \begin{bmatrix} \text{state}(1) & \text{state}(2) & \ldots & \text{state}(t) \\ \text{phone}(1) & \text{phone}(2) & \ldots & \text{phone}(t) \\ \text{word}(1) & \text{word}(2) & \ldots & \text{word}(t) \end{bmatrix} \quad (1)$$

Dynamic search algorithms, like HVite, keep a vector of state information at each time step, as suggested by Eq. 1. Dynamic Bayesian networks provide a formal representation of dynamic search, which we will discuss in lecture 7 (GMTK).

Static search algorithms, like HDecode and julius, pre-compile all of the information in Eq. 1 into a single integer state index at each time step. This is done by (1) creating search graphs (finite state transducers or FSTs) that describe the sequencing information at each level of abstraction, e.g., one FST for the language model, one for the dictionary, and one for each phone model, and (2) composing together the search graphs.

OpenFST (http://www.openfst.org/, [1]) is a tool for creating, composing, and optimizing finite state transducers. OpenFST started as an open-source re-implementation of the AT&T finite state machine library (http://www.research.att.com/ fsmtools/fsm/). OpenFST and FSMlib use the same text file format to specify finite state machines, so it's easy to mix and match the tools provided the two toolkits, in case you

find a tool that works better in one toolkit than the other. In particular, AT&T has also released their finite-state transducer based speech recognition decoder, http://www.research.att.com/ fsmtools/dcd/, which uses FSM networks specified in the same format as FSMlib. Mohri's tutorial, based on the AT&T decoder, is still the best available introduction to the use of FSTs in speech recognition [3]. It's probably also possible to read OpenFST-optimized search graphs into Julius [2], but I haven't done that yet.

2 Tutorial

This entire section is copied almost verbatim from an FSMlib tutorial presented in 2004 by Eric Fosler-Lussier (http://www.cse.ohio-state.edu/ fosler/) at the NAACL summer course at Johns Hopkins University (http://www.cs.cornell.edu/home/llee/naacl/summer-school/04/; http://www.clsp.jhu.edu/workshops/). I have changed program names and command line options to their OpenFST equivalents, and checked to make sure that the commands run.

2.1 Shake-and-bake language generation

To get acquainted with OpenFST, well make a little language generator. Well generate "sentences" based on parts of speech and fill in the lexical items randomly.

Open up your favorite editor, and type in the following:

```
0   1   DET
1   2   N
2   3   V
3   4   DET
4   5   N
5
```

This means from state 0 to state 1, we have a DET(a determiner) and so forth; the 5 by itself indicates that it's a final state. The first state mentioned in the file (0) is the initial state. DET stands for determiner, N stands for noun, and V stands for verb. Save the file as sent.fsa.txt. Now, open another file (pos.voc) which well put the part of speech vocabulary into. This is a file that gives mappings from symbols to integers. The epsilon symbol (-) should always be symbol 0.

```
- 0
DET 1
N 2
V 3
```

Save this file (pos.voc).

The first file gives the textual representation of the fsa. To compile it into binary form, use the following command:

```
fstcompile --acceptor -isymbols=pos.voc sent.fsa.txt > sent.fsa;
```

The `--acceptor` option tells it that this is a finite state acceptor (only one symbol per edge) rather than a finite state transducer (two symbols per edge: one input, one output).

You can print it out into text form using

```
fstprint -isymbols=pos.voc sent.fsa
```

or you can draw it using the following commands:

```
fstdraw i pos.voc sent.fsa | dot Tps > sent.ps
```

then you can view it using any postscript viewer, e.g., evince, gv, ghostview, or Acrobat. `dot` is part of the graphviz package, which can be installed under ubuntu (for example) by typing

```
sudo apt-get install graphviz
```

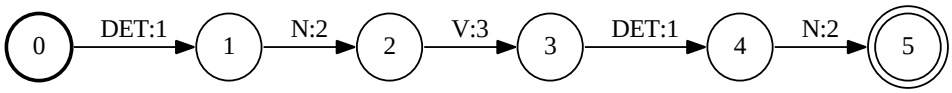

Figure 1: Finite state acceptor: a regular grammar specifying the parts of speech (POS) that compose a sentence.

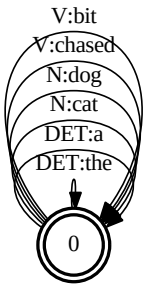

Figure 2: Finite state transducer: a dictionary mapping POS to words.

The resulting plot of the finite state transducer should look like Fig. 1.

Now, create a second file that maps parts of speech to words. For example:

```
0  0  DET the
0  0  DET a
0  0  N cat
0  0  N dog
0  0  V chased
0  0  V bit
0
```

Notice that this file specifies a finite state transducer (each edge has two symbols: an input part of speech symbol, and an output word symbol). Notice also that there is only one state in this FST: state 0. Every edge is a self-loop from state 0 to itself; they differ only in the edge labels (Fig. 2).

Save this file as dict.fst.txt. Youll need to create a second vocabulary file for the words:

```
- 0
a 1
the 2
cat 3
dog 4
chased 5
bit 6
```

save this as word.voc.

Now you can compile the transducer. This uses fstcompile with the t option:

```
fstcompile -isymbols=pos.voc -osymbols=word.voc dict.fst.txt > dict.fst
```

Compose the two together and you get all possible strings in the language.

```
fstcompose sent.fsa dict.fst > strings.fst
fstdraw i pos.voc o word.voc strings.fst | ./dot Tps > strings.ps
ghostview strings.ps
```

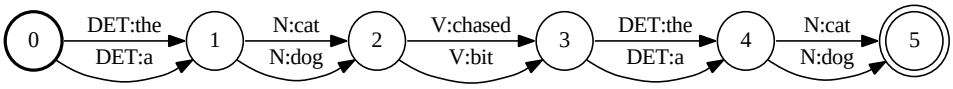

Figure 3: Finite state transducer: a regular grammar specifying a language in which there are 32 possible sentences.

To get a random string from this language, use

```
fstrandgen strings.fst | fstproject --project_output |
  fstprint --acceptor --isymbols=word.voc |
  awk 'BEGIN{printf("\n")}{printf("%s ",$3)}END{printf("\n")}'
```

To do:

1. Add more vocabulary to the POS/word map. What is the silliest sentence you can create?
2. How would you handle "The dog barked"?

2.2 Cryptography

The idea of this task is to try and decipher some encrypted text. You have intercepted five messages that you believe are using the same (simple) substitution cipher. Each letter is transformed into one and only one other letter. (You may see this type of puzzle in the newspaper as the "Cryptoquip"). For example, if you believe that "B" is "E" in one instance, it is "E" in all other instances in the message, and there is no other letter that can be changed to "E". To figure out how to break this code, we will use some known information about the statistics of letter frequencies in English. Then we will build a decoder (similar to a decoder in a speech recognition system) to try to figure out the code. The models we will use will include

- A finite state transducer to map cryptletters to letter pairs
- A finite state transducer to map letter pairs to regular (unencrypted) letters
- A finite state transducer to map letters to words
- A finite state automaton describing the input text
- A finite state automaton describing the output text

2.2.1 Unpack the distribution

Download crypto.tar.gz from the web page, then execute the following command:

```
tar xzf crypto.tar.gz
```

This will create a directory crypto. In the directory "data" you will see the file crypttext, which contains the text you need to break.

2.2.2 Gather statistics

The main thing that makes this system run is that the statistics of the letters in the encrypted message are assumed to be similar to that of general English. If you look in the file data/nyt-letterstats.txt, you will see statistics of letters from the New York Times (in June 2002), where the text was split into roughly 200-character chunks from which the mean and standard deviation of the frequency of each letter were computed. We now need to do something similar for the crypttext. Run the following script

```
scripts/get-letterstats.pl data/crypttext > data/cryptstats
```

You should get a file out with the probability of each letter in the crypttext. You should satisfy yourself that the result is somewhat close to right.

2.2.3 Build a letter-to-word FST

This is where scripting skills will come in handy. First, lets make a transducer that can convert the letters B O X into the word BOX. Open up a text editor and enter the following transducer (BOX.fst.txt)

```
0 1 B -
1 2 O
2 3 X BOX
3
```

Save the file and compile it:

```
fstcompile -isymbols=data/letter.voc -osymbols=data/word.voc BOX.fst.txt > BOX.fst
```

You can test to see if its working by composing it with test1.fsa (which contains B O X).

```
fstcompile --isymbols=data/letter.voc data/test1.fsa.txt |
fstcompose - BOX.fst |
fstprint -isymbols=data/letter.voc -osymbols=data/word.voc
```

If you get nothing back, then theres a bug somewhere. Now, for the scripting part: write a little script that takes an argument and creates the appropriate FST for that word. For example, if you called the script my_word_generator, by calling

```
my_word_generator BOX
```

you should get out exactly the fst above. If you need help with this, contact me or the TA.

2.2.4 Build a dictionary

Make a directory dict. Remember that a dictionary is just the union of a whole bunch of individual words. Write another script that takes every word in data/wordlist and creates an fst for each word in the wordlist, and puts it in the dict directory. You may want to look at scripts/gen-allwords.sh for an example. Gather all of the fsts into one big dictionary by taking the union:

```
for txt in ‘ls dict/*.fst.txt‘; do
  fst=‘echo $txt | sed ’s/\.txt//’‘;
  fstcompile --isymbols=data/letter.voc --osymbols=data/word.voc $txt >
$fst;
done
cp dict/ABLE.fst dict.fst;
for fst in ‘ls dict/*.fst‘; do
  fstunion dict.fst $fst > tmp.fst
  mv tmp.fst dict.fst;
done
```

Take a look to see how big the resulting fst is by using fstinfo dict.fst. We can compact it by determinizing the fst:

```
fstdeterminize dict.fst > dict-det.fst
```

You now have an fst which can convert one set of letters into one word. To get word sequences, you need to concatenate that with a word boundary marker and then take the closure. First, create a file pound.fst.txt which has the following contents:

```
0 1 # #
1
```

Compile this into a transducer:

```
fstcompile -isymbols=data/letter.voc -osymbols=data/word.voc pound.fst.txt >
pound.fst
```

Now do the concatenation and closure. Here, Ive also included some determinization and minimization. It turns out that the transducer itself is not determinizable, so there is a way to encode the transducer as an automaton, do the determinization/minimization, and then turn it back into a transducer. Note how you can pipe all of these FSTs into each program, and do a seqence of operations

```
rm f x.fst
fstconcat dict-det.fst pound.fst | fstclosure |
fstrmepsilon | fstencode -encode_labels  x.fst | fstdeterminize |
fstminimize | fstencode -decode -encode_labels - x.fst > dictstar.fst
rm f x.fst
```

2.2.5 Generate the cryptletter-to-real letter mapping

Running the following script will compute P(cryptletter|actualletter). What happens is that we take the frequency of each cryptletter and compare it against the Gaussian distribution for each actual letter. For example, the letter "E" (in real text) occurs roughly 12% of the time. The cryptletters "A" "X" and "U" might be good candidates for "E".

The script `generate-likelihoods.pl` will take two statistics files, plus an optional hypothesis file. The hypothesis file gives a guess as to a cryptletter/real letter combination. Originally, you dont have a hypothesis, so there is no hypothesis file.

```
scripts/generate-likelihoods.pl data/nyt-letterstats.txt
data/cryptstats > subs1.fst.txt
fstcompile -isymbols=data/letter.voc -osymbols=data/pairs.voc
subs1.fst.txt > subs1.fst
```

2.2.6 Generate a crypttext word sequence for input

You now need to generate, for each sentence, a fsa that represents the sentence. You can extend your my_word_generator script to do this; make sure that a "#" sign appears after each word. You also dont need to put out words (i.e. this doesnt need to be a transducer). However, if you're lazy (like me) you can reuse your word generator (see below).

You should create 5 files with one line each in them:

```
split -1 data/crypttext crypt_
```

This will create crypt_aa through crypt_ae.

Now, generate your fsa heres my script of laziness in action:

```
scripts/gen-sent.sh crypt_aa > crypt_aa.fsa.txt
fstcompile --isymbols=data/letter.voc crypt_aa.fsa.txt > crypt_aa.fsa
```

Now, if you compose these together, youll get the weighted graph of all possible words. Its easier to look at if you just project to the output words (i.e., get rid of the input letters).

```
fstcompose crypt_aa.fsa subs1.fst data/pair2real.fst
dictstar.fst | fstproject --project_output | fstrmepsilon |
fstdeterminize | fstminimize | fstprint --isymbols=data/word.voc | less
```

Or, if you want to look at it graphically

```
fstcompose crypt_aa.fsa subs1.fst data/pair2real.fst
dictstar.fst | fstproject --project_output | fstrmepsilon |
  fstdeterminize | fstminimize | fstdraw --isymbols=data/word.voc |
  dot -Tps > crypt_aa.ps
evince crypt_aa.ps
```

Yikes! Its huge!

Try this out with the other sentences (crypt_ab..crypt_ae).

You can also put some pruning into the loop... this means that youre not guaranteed to get the right answer, but it can help you guess. Try pruning paths that have costs ¿ 1:

```
fstcompose crypt_aa.fsa subs1.fst data/pair2real.fst
dictstar.fst | fstproject --project_output | fstrmepsilon |
  fstdeterminize | fstminimize | fstprune -weight=1 |
  fstprint --isymbols=data/word.voc | less
```

Youll notice that in one of the files you only get one possible answer for a word mapping somewhere in the file. This gives you some constraints that you can use as a first guess (see below).

You can also see the bestpath by doing

```
fstcompose crypt_aa.fsa subs1.fst data/pair2real.fst
dictstar.fst | fstbestpath | fstdraw --isymbols=data/letter.voc
  --osymbols=word.voc | dot Tps > crypt_aa_bp.ps
```

Notice that this doesnt end up making much sense. Why? All of the letter decisions are made independently at this point we dont have a constraint that says if you choose T for E, then you always choose T for E. Well deal with that a bit later in step 7. When you figure out the one word that has no alternative, youll want to put the corresponding letters into a hypothesis file as a guess. To get the corresponding letters out, take the file you found, and run (making sure to replace XX with the appropriate file)

```
fstcompose crypt_XX.fsa subs1.fst | fstproject --project_output > tmp.fsa
fstcompose tmp.fsa data/pair2real.fst dictstar.fst |
fstbestpath | fstprint --isymbols=data/pairs.voc --osymbols=data/word.voc | less
```

Extract the letter pairs and put it into a file guess1. For example, if you believe XEF means CAT, put into the file

```
XC
EA
FT
```

Now, generate a second substitution fst, with your new guesses:

```
scripts/generate-likelihoods.pl data/nyt-letterstats.txt
data/cryptstats guess1 > subs2.fst.txt
fstcompile -isymbols=data/letter.voc -osymbols=data/pairs.voc
  subs2.fst.txt > subs2.fst
```

Repeat (with subs2.fst instead of subs1.fst) until you have a complete mapping. The second round should be much better than the first, and you should have most of it by the third round.

2.2.7 Path constraints

OK, youve solved it, but now what? Well, it turns out that we could have made the problem easier to solve by adding more constraints. One constraint is that the letter substitutions have to apply to the whole string. Theoretically, you can do this with an automaton on the pairs alphabet. For example, if X is really E, let PAIR be all of the pairs that either do not start with X. You can write this language constraint as:

$$\text{PAIR}^*\text{XE}(\text{PAIR} \bigcap \text{XE})^* \tag{2}$$

Which says that XE can only occur in the string (and must occur once), but XA (for example) cant. Of course, we can write a similar constraint for XA, XB, etc. If you take the union of all 26 of these constraints, then you end up with the language that says X must pair with one and only one of these 26 letters. Of

course, this doesnt say that only one cryptletter can be converted into E. For that constraint, let PAIR2 be all of the pairs that do not end with E.

$$\text{PAIR2}^*\text{XE}(\text{PAIR2}\bigcap\text{XE})^* \tag{3}$$

says that only X (and no other letter) can be converted into E. These are the "backward" constraints, whereas the previous constraints are the "forward" constraints.

By intersecting all 26 forward and 26 backward constraints, in theory you can provide the complete constraints on the language. However, the state space becomes huge if you try this. On the plus side, though, we can intersect each one of the constraints in turn, followed by determinization and minimization, which often doesnt make the resulting FSTs grow too large. To try this out, first find the valid pairs of letters according to the word models, and then apply the constraints to these pairs.

```
# link to the forward/backward constraints
ln s /export/fosler/crypto/fwdconstr .
ln s /export/fosler/crypto/revconstr .
# determine the pairs that can possibly arise
fstcompose crypt_aa.fsa subs1.fst | fstproject -project_output > tmp1.fsa
# eliminate the pairs that dont make valid words
fstcompose tmp1.fsa data/pair2real.fst dictstar.fsa |
  fstproject > tmp2.fsa
# apply the constraints by intersecting one at a time
scripts/do-constraints.pl tmp2.fsa > tmp3.fsa
```

You will find if you use fstinfo n on tmp2.fsa and tmp3.fsa that the latter is much smaller. Now, if you compose with the dictionary again, youll see that there are many fewer hypotheses:

```
fstcompose tmp3.fsa data/pair2real.fst dictstar.fsa |
fstprint -isymbols=data/pairs.voc -osymbols=data/word.voc | less
```

Try this with some of the other sample sentences. What happens if you concatenate all five sentences together? To think about: what other types of constraints could you put into the system to make the decoding process faster (in terms of the number of iterations)?

References

[1] Cyril Allauzen, Michael Riley, Johan Schalwyk, Wojciech Skut, and Mehryar Mohri. OpenFst: A general and efficient weighted finite-state transducer library. *Lecture Notes in Computer Science CIAA*, 4783:11–23, 2007.

[2] Tatsuya Kawahara, Akinobu Lee, Tetsunori Kobayashi, Kazuya Takeda, Nobuaki Minematsu, Shigeki Sagayama, Katsunobu Itou, Akinori Ito, Mikio Yamamoto, Atsushi Yamada, Takehito Utsuru, and Kiyohito Shikano. Free software toolkit for japanese large vocabulary continuous speech recognition. In *Proc. Internat. Conf. Spoken Language Processing*, pages 476–9, Beijing, 2000.

[3] Mehryar Mohri, Fernando Pereira, and Michael Riley. Weighted finite-state transducers in speech recognition. *Computer Speech and Language*, 16:69–88, 2002.