CSE 5525: Foundations of Speech & Language Processing

FST Tutorial

This is based on a tutorial I gave at an NSF summer school. Because pipes don't work under windows, I write out the individual commands and save a lot of intermediate files; you can feel free to pipe things, however. Later on in the tutorial, I use pipes as shorthand, but you can save each intermediate step as a separate file.

The OpenFST tools should be installed in your VM.

Part 1: Shake-and-bake language generation

To get acquainted with the OpenFST toolkit, we'll make a little language generator. We'll generate "sentences" based on parts of speech and fill in the lexical items randomly.

Step 0: Make a directory

```
mkdir part1
cd part1
```

Step 1: Create a sentence

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" 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 we'll 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=true --isymbols=pos.voc sent.fsa.txt sent.fsa
```

The acceptor flag is necessary since the toolkit will otherwise think it's a transducer. You can also save the symbols inside the FST now using the "--keep_isymbols" flag.

You can print the fsa out into text form using

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

or you can draw it if you have graphviz installed using:

```
fstdraw --acceptor=true --isymbols=pos.voc sent.fsa sent.dot

xdot sent.dot
```

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
```

Save this file as dict.fst.txt

You'll 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 without the --acceptor option (or fsmcompile 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 --isymbols=pos.voc --osymbols=word.voc strings.fst strings.dot

xdot strings.dot
```

To get a random string from this language, use

```
fstrandgen strings.fst rand.fst
fstdraw --isymbols=pos.voc --osymbols=word.voc rand.fst rand.dot


xdot rand.dot
```

Now try to see if sentences are in your language:

1) Create a new automaton that should have a sentence in your language in it.
    a. Call this senta.fsa
2) See if this is actually in your language by composing with STRINGS.fst
    a. `fstcompose strings.fst senta.fsa out.fst`
    b. `fstinfo out.fst`
    c. `fstprint --isymbols=pos.voc --osymbols=word.voc out.fst`

What's going on here? To see if a string is accepted by an acceptor/transducer, then you intersect (acceptor) or compose (transducer) an acceptor containing the string and see if the intersection/composition is empty (not accepted) or has more than zero states (accepted).

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"?

Part 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

Step 1: Check out the directory cryptography. In the directory "data" you will see the file crypttext, which contains the text you need to break.

Step 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

```
perl 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.

Step 3: Build a letter-to-word FST

This is where scripting skills will come in handy. First, let’s 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
    --keep_isymbols --keep_osymbols BOX.fst.txt > BOX.fst
```

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

```
fstcompose data/test1.fsa BOX.fst > test.fst
fstprint --isymbols=data/letter.voc --osymbols=data/word.voc test.fst
```

If you get nothing back, then there’s 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.

Step 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 a fsm 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. The AT&T toolkit used to let you do it all at once, but OpenFST only does two at a time, so I’ve written a script that generates dict.fst:

```
# build dict.fst by composing many fsts in dict/*.fst
./scripts/gen-union.sh
```

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 a 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, I’ve also included some determinization and mimimization. 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 sequence 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
```

Step 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 don’t 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
```

Step 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 don't need to put out words (i.e. this doesn't 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 (one after the dash below):

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

This will create crypt_aa through crypt_ae.

Now, generate your fsa – here's my script of laziness in action:

```
scripts/gen-sent.sh crypt_aa > crypt_aa.fsa.txt

fstcompile --acceptor --isymbols=data/letter.voc crypt_aa.fsa.txt >
crypt_aa.fsa
```

Step 6a: Now, if you compose these together, you'll get the weighted graph of all possible words. It's 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 | fstcompose - data/pair2real.fst |
fstcompose - dictstar.fst | fstproject --project_output | fstrmepsilon
| fstdeterminize | fstminimize  | fstprint --isymbols=data/word.voc --
acceptor | less
```

Or, if you want to look at it graphically, replace fstprint with fstdraw and check it out using graphviz/dot:

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

Yikes! It's huge!

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

You can also put some pruning into the loop… this means that you're 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 | fstcompose - data/pair2real.fst |
fstcompose - dictstar.fst | fstproject --project_output | fstrmepsilon
| fstdeterminize | fstminimize | fstprune --weight=1  | fstprint --
isymbols=data/word.voc --acceptor | less
```

You'll 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 | fstcompose -
data/pair2real.fst | fstcompose - dictstar.fst | fstproject
--project_output | fstrmepsilon | fstdeterminize |
fstminimize | fstshortestpath | fsttopsort  | fstprint --
isymbols=data/word.voc --acceptor | less
```

Notice that this doesn't end up making much sense. Why? All of the letter decisions are made independently at this point – we don't have a constraint that says "if you choose T for E, then you always choose T for E". We'll deal with that a bit later in step 7.
When you figure out the one word that has no alternative, you'll 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 | fstcompose - dictstar.fst |
fstshortestpath | fsttopsort | fstprint --isymbols=data/pairs.voc --
osymbols=data/word.voc  | less
```

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

X C
E A
F T

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 from 6a (with subs2.fst instead of subs1.fst) until you have a complete mapping. You may want to look at different sentences each time. The second round should be much better than the first, and you should have most of it by the third round.

Step 7: Path constraints (advanced topic)
NOTE: THIS WON'T BE DOABLE IN CLASS BECAUSE OF SOME MISSING DATA, BUT I'VE LEFT THE DISCUSSION INTACT BECAUSE IT GIVES YOU AN

IDEA ABOUT HOW TO USE CONSTRAINTS. OK, you've 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:

PAIR* XE (PAIR ∪ XE)*

Which says that XE can only occur in the string (and must occur once), but XA (for example) can't. 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 doesn't 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.

PAIR2* XE (PAIR2 ∪ XE)*

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 doesn't 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.

[Note: the VM does not currently have the forward and backward contstraints, so you can't do this yet.]

```
# 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
fsmcompose crypt_aa.fsa subs1.fst | fsmproject –2 >
tmp1.fsa

# eliminate the pairs that don't make valid words
fsmcompose tmp1.fsa data/pair2real.fst dictstar.fsa |
fsmproject –1 > tmp2.fsa

# apply the constraints by intersecting one at a time
```

```
scripts/do-constraints.pl tmp2.fsa > tmp3.fsa
```

If you use “fsminfo –n” on tmp2.fsa and tmp3.fsa, you’ll see the latter is much smaller. If you compose with the dictionary again, you’ll see there are many fewer hypotheses:

```
fsmcompose tmp3.fsa data/pair2real.fst dictstar.fsa |
fsmprint –i data/pairs.voc –o 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)?