CSE 5525: Foundations of Speech and Language Processing
Tutorial: Building a word prediction / autocorrect system

This tutorial assumes that you have at least completed the Shake-and-Bake part of the Cryptography tutorial, so you are familiar with fstcompile, fstcompose, fstintersect, fstbestpath, fstrmepsilon, and so forth.

Today's tutorial is a bit more open ended: we are going to construct a word prediction module. If a user types the first few characters of a word, can we guess some words that can complete the word? If we have time, you can also see about putting in a confusion model to predict typos.

Part 1:

General instructions (see if you can follow these first):

The first idea is that we will need to build a dictionary of words that transforms letters to words, and then compose this with a constraint that indicates the letters that have been typed up to this point.

Let sigma be an acceptor that accepts any one letter.
Let L be an acceptor that represents the letters that have been typed already.
Let D be a dictionary (converts letters into known words)

The result of part 1 should be:

bestpath (compose (concat (L, closure(sigma)), D))

sigma and L should be relatively straightforward to create.

To create D, you first need to create for every word a single transducer converting the letters to the word. The dictionary is the union of all of these transducers (although you might want to remove epsilons and then determinize the transducers after taking the union).

Next page: a python-based solution (don't peek until you've thought this through)

Part 1: one solution

One of the students in the class introduced me to pyfst, which gives python bindings to the FST libraries (which the command line version is built on). I've installed it in the revised VMs. You can also use the ipython interactive notebook to display finite state machines (it calls graphviz/xdot).

You can start up the notebook using

```
ipython notebook
```

You can see examples of how pyfst (the python bindings for OpenFST) work at

http://pyfst.github.io

I created a file, gen_word.py, that can be read into the notebook using

```
%run gen_word.py
```

The first thing to work on is collecting a set of words for the dictionary. I created a function get the words from a file:

```
def create_count_array (file):
      "This opens a file, reads the contents, and then stores the
contents as a dictionary, indexed by word, with the count of the
word"
      count={}
      with open(file) as f:
            for line in f:
                  words = line.split()
                  for word in words:
                        if word in count:
                              count[word]=count[word]+1
                        else:
                              count[word]=1
      return count
```

This creates a dictionary that maps words to the count of the number of times that word was seen.

The next step was to create a transducer to go from the spelling of a word to that word.

```
def gen_word_fst (word,isyms=None,osyms=None):
      "Takes a word and creates a transducer from the letters to
the word, introducing symbols into the symbol table as needed"
      wordfst=fst.Transducer(isyms,osyms)
      state=0
      for char in word:
            wordfst.add_arc(state,state+1,char,'ε')
            state=state+1
      wordfst[state].final=True
      for arc in wordfst[state-1].arcs:
            arc.olabel=wordfst.osyms[word]
      return wordfst
```

Take a look at what happens when you call gen_word_fst('hello').

You can then create the dictionary by taking the union of all of the word fsts.

```
def create_wordlist_fst (words):
      "This takes a list of words and creates a letter-to-word
transducer for all of the words (unioned together)."
      wordset=fst.Transducer();
      for word in words:
            wordfst=gen_word_fst(word,isyms=wordset.isyms,osyms=wo
      rdset.osyms)
            wordset=wordset|wordfst
      return wordset
```

The other side of the equation is the letter constraints. Remember that this is the first few letters, followed by the closure of sigma (to absorb all of the other letters). The first part can use a built in function to pyfst. You need to make sure that the symbol files remain constant, so pass the letter symbol table from the dictionary to the function.

```
thisfst=fst.linear_chain(letterstring,syms)
```

The sigma acceptor can be constructed from the symbol table:

```
def sigma (syms):
      "This creates a two-state acceptor that accepts any one
letter in isyms"
      thisfst=fst.Acceptor(syms=syms);
      for sym,val in syms.items():
            if (val > 0):
                  thisfst.add_arc(0,1,sym)
      thisfst[1].final=True;
return thisfst
```

Putting this all together:

```
def letter_constraint (letterstring,syms):
     "Create an FSA that has the already typed letters followed
by sigma*"
     thisfst=fst.linear_chain(letterstring,syms);
     sigmafst=sigma(syms).closure()
     thisfst.concatenate(sigmafst)
     thisfst.remove_epsilon()
     return thisfst.determinize()
```

At this point, we can now create a function to build the dictionary, create the letter constraints, and search for the n best alternatives for word completion. I have various corpora to build the end system from: "test" which has some silly sentences, "brown" which is the entire brown corpus, "brown1000" and "brown100" which are the first 1000 or 100 sentences of brown. Note that "a>>b" is shorthand for compose(a,b).

```
def find_matches1 (letterstring,n=3):
     "Run the whole thing: build the fsts, run the match, return
the results"
     counts=create_count_array('brown100')
     dict=create_wordlist_fst(counts.keys())
     let=letter_constraint(letterstring,dict.isyms)
     out=(let>>dict).shortest_path(n)
     out.project_output()
     out.remove_epsilon()
     return out
```

Part 2: Integrating unigram constraints

The unigram probability of a word is just the probability that it occurs in a test file. We can get this probability from counting and dividing (which is why we have the count function). The negative log of the probability can be used as a weight on the output. In order to reuse the previous work, I integrated this constraint by creating a single state weighted FSA, where each word was on an arc with its negative log probability as the weight. This can be composed with the previous fst. The python file above has my solution (defined as find_matches2) but you might want to explore this yourself.

Part 3: autocorrect

As a last idea, how can you build an autocorrect function? Hint: you need to think about the relationship between typed letters and the letters that were intended.