# Language Technology, Spring 2013, HW 2

Prof. Liang Huang

Due at 11:59pm on Friday April 5 on Blackboard; late submissions by Sunday April 7

In Japanese text, foreign words borrowed in the modern era (such as Western names and technical terms) are *transliterated* into special symbols called *katakana*. Katakana is a syllabary, in which most symbols stand for syllable sounds (such as *ko* or *ra*). Because spoken Japanese consists largely of consonant-vowel syllables, most katakana symbols stand for two Japanese phonemes. In this assignment, among other things, we'll decode katakana words of English origin back into English. Refer to the slides for this section and the tutorial on writing systems and transliteration for more information (available from the course website).

You will get from `hw2/hw2-data.tgz` which includes:

| | |
|---|---|
| `eword.wfsa` | a unigram WFSA of English word sequences |
| `epron.wfsa` | a trigram WFSA of English phoneme sequences |
| `eword-epron.data` | an online dictionary of English words and their phoneme sequences |
| `eword-epron.wfst` | a WFST from English words to English phoneme sequences |
| `epron-eword.wfst` | inverse transducer; the result of `carmel -v eword-epron.wfst` |
| `epron-espell.wfst` | a WFST from English phoneme sequences to English letter sequences |
| `epron-jpron.data` | a database of aligned English/Japanese phoneme sequence pairs |
| `jprons.txt` | a short list of Japanese Katakana sounds to decode |
| `epron.probs` | a human-readable version of `epron.wfsa`. |

## 1 Pronouncing and Spelling English (15 pts)

1. Pick some English words and pronounce them with `eword-epron.wfst`. This transducer is essentially a giant lookup table built from `eword-epron.data`. Turn in five (5) examples. Here is one:

   ```
   echo HELLO | carmel -sliOEQk 5 eword-epron.wfst
   ```

2. Now let's try to pronounce character sequences without whole-word lookup (since we may face unknown words, for example). You can send a character string backwards through `epron-espell.wfst`. Try several words and turn in your examples; here is a suggested command:

   ```
   echo 'H E L L O' | carmel -sriIEQk 5 epron-espell.wfst
   ```

   Did you get some non-sense output? Why? Explain it in terms of probabilitic modeling.

3. Let's fix the nonsense by adding a language model of likely pronunciations, `epron.wfsa`. Try several and turn in your examples.

   ```
   echo 'H E L L O' | carmel -sriIEQk 5 epron.wfsa epron-espell.wfst
   ```

4. Now spell out some phoneme sequences by using `epron-espell.wfst` in the forward direction. Try several and turn in your examples.

   ```
   echo 'HH EH L OW' | carmel -sliOEQk 50 epron-espell.wfst
   ```

5. How to improve the above results? Well, you can try to take advantage of the language model `eword.wfsa` (as a filter). But you need a "bridge" from espell to eword. That's trivial, isn't it? Write a simple Python program `gen_espell_eword.py` to generate `espell-eword.wfst` from the word list in `eword-epron.data`. And now you can:

   ```
   echo 'HH EH L OW' | carmel -sliOEQk 50 epron-espell.wfst espell-eword.wfst eword.wfsa
   ```

   Try several examples and turn them in. Include `gen_espell_eword.py` and `espell-eword.wfst` in the submission also.

6. Well, alternatively, you can go directly from epron to eword:

   ```
   echo 'HH EH L OW' | carmel -sliOEQk 50 epron-eword.wfst eword.wfsa
   ```

   Is this better than 5)? Try the same set of examples can compare the results with those from 5).

7. Try to pronounce some words that are not in the data file `eword-espell.data`. Try several examples and turn them in.

   ```
   echo WHALEBONES | carmel -sliOEQk 5 eword-epron.wfst
   ```

8. Now try pronouncing letter sequences of those same words. Try several examples and turn them in.

   ```
   echo 'W H A L E B O N E S' | carmel -sriIEQk 5 epron.wfsa epron-espell.wfst
   ```

9. How about phoneme sequences of new words? What happens when you use `eword-epron.wfst` versus `epron-espell.wfst`? Try several examples and turn them in.

   ```
   echo 'W EY L B OW N Z' | carmel -sriIEQk 5 eword-epron.wfst
   echo 'W EY L B OW N Z' | carmel -sliOEQk 5 epron-espell.wfst
   ```

10. You can hook up these transducers in unexpected ways. What is the following command trying to accomplish?

    ```
    echo 'BEAR' | carmel -sliOEQk 10 eword-epron.wfst epron-eword.wfst eword.wfsa
    ```

11. Write up a half-page or so of observations from your experiments. What did you learn about these automata and what they can (or can't) do? Please take time to express your thoughts clearly using complete sentences.

## 2 Decoding English Words from Japanese Katakana (50 pts)

1. Look at `epron-jpron.data`. Each pair in this file consists of an English phoneme sequence paired with a Japanese phoneme sequence. For each pair, Japanese phonemes are assigned integers telling which English phonemes map to them. For example:

   ```
   AE K T ER          ;; English phoneme sequence for 'actor'
   A K U T A A        ;; Same word, loaned into Japanese
   1 2 2 3 4 4        ;; e.g., Japanese T maps to the 3rd English sound
   ```

   From the data, we can see that each English phoneme maps to *one or more* Japanese phonemes. Estimate the channel probabilities $p(\mathbf{j} \mid s)$ where **j** represents one or more Japanese phoneme(s), and $s$ an English phoneme, using the simplest maximum likelihood estimation (MLE). Write a Python program `estimate.py` that generates the file `epron-jpron.probs` like this (each line represents one $p(\mathbf{j} \mid s)$, and you can omit those with probability $< 0.01$ and you can omit those where one English phoneme maps to more than three (3) Japanese phonemes since they are mostly noise):

   ```
   AE : A # 0.95
   AE : Y A # 0.05
   T : T # 0.4
   T : T O # 0.4
   T : TT O # 0.1
   T : TT # 0.1
   R : A A # 0.8
   R : E R # 0.1
   R : E R U # 0.1
   ...
   ```

   Transitions for each English phoneme should be consecutive (see above). Include `estimate.py` and `epron-jpron.probs`.

2. Now, based on this, you can create a WFST called `epron-jpron.wfst`. It should probabilistically map English phoneme sequences onto Japanese ones, behaving something like this in the forward direction:

   ```
   echo 'L AE M P' | carmel -sliOEQk 5 epron-jpron.wfst
   ```

   and you'll get a list like

   ```
   R A M P
   R U A M P
   R A M U P
   R A M P U
   R A M PP U
   ...
   ```

   Most transducers will consist of 300 transitions or so.

3. Using your knowledge about Japanese syllable structure (see slides), do these results (including their rankings) make sense? If not, why (briefly explain)?

4. Now consider the following Japanese katakana sequences, from a Japanese newspaper (actually these are the phoneme sequences that are easily read off from the katakana characters):

```
H I R A R I K U R I N T O N
H O M A A SH I N P U S O N
R A PP U T O PP U
SH E E B I N G U K U R I I M U
CH A I R U D O SH I I T O
SH I I T O B E R U T O
SH I N G U R U R U U M U
G A A R U H U R E N D O
T O R A B E R A A Z U CH E KK U
B E B I I SH I T A A
S U K O TT O R A N D O
B A I A R I N K O N TCH E R U T O
```

(For your convenience I have included a `jprons.txt` for the above sequences.)

How would you decode them into English *phoneme sequences* using carmel (with the WFST you just built and with help of `epron.wfsa`)? Show and command-line and results (with probs, with `-k 5` option). Do they make sense (as English words)?

5. Now redo the above exercise, but this time into English *words* by assembling `eword.wfsa`, `eword-epron.wfst`, and `epron-jpron.wfst` with carmel. Show the command-line and results (with probs, with `-k 5`). Do you think the results make more sense this time? Briefly explain.

6. Try search for at least three (3) other Japanese katakana examples not covered in this HW or in slides, and try to decode them back into English using the above approach. They should preferably be a two word phrase or a compound word, like "shopping center" or "piano sonata".

7. Are there other ways to decode into English words given all these existing WFSAs and WFSTs (with some tweaking)? What are the differences of these methods (speed, accuracy, etc.)?

## 3 Viterbi Decoding Implementation (35 pts)

1. (25 pts) Now implement your own Viterbi decoding in Python for Question 2.4 (jprons to eprons). Your command-line should be:

```
echo -e 'P I A N O\nN A I T O' | ./decode.py epron.probs epron-jpron.probs
```

with the result like: (note the input represents the Japanese katakana `PI A NO`, not the English word!)

```
P IY AA N OW # 1.489806e-08
N AY T # 8.824983e-06
```

For your convenience, we have converted `epron.wfsa` into a human-readable `epron.probs`, in the same format as `epron-jpron.probs` above, e.g.:

```
T S : </s> # 0.774355
T S : AH # 0.0333625
T S : Z # 2.92651e-07
```

which correspond to our intuition that sounds `T S` (words with `-ts`) are very likely to end a sentence or a word, and sounds `T S AH` are much less likely (but still noticeable in words like `WATSON` and `BOTSON`), but `T S Z` is (almost) completely unheard of (with a tiny prob here due to smoothing).

**You only need to print the 1-best solution and its probability.** Please

(a) describe your algorithm in pseudocode first,

(b) analyse its complexity, and

(c) implement it in Python. Your grade will depend on both correctness and efficiency.

Note that each English phoneme corresponds to **one to three** Japanese phonemes (the PIANO example is 1-1 and is too simple, and the NIGHT/KNIGHT example is more interesting), so the Viterbi algorithm from the slides needs to be extended a little bit.

Compare your 1-best results with carmel's (on the data in Q2.4). Try your best to match them. Include a side-by-side comparison (hint: `diff -y`).

Hint: if you want to make sure your program is 100% correct with respect to Carmel, you can compare their results by:

```
cat jprons.txt | ./viterbi.py epron.probs epron-jpron.probs > results.my
cat jprons.txt | carmel -bsriIEQk 1 epron.wfsa epron-jpron.wfst 2>/dev/null \
               | awk '{for (i=1;i<NF;i++) printf("%s ", $i); printf("# %e\n", $NF)}' \
               >  results.carmel
diff -b results.my results.carmel
```

You should see no output if you did everything correctly. To generate a side-by-side comparison, replace `diff -b` by `diff -by`.

FYI, my not-very-optimized implementation is 60 lines of Python, and runs about 4 times slower than Carmel. Your implementation will be graded in terms of both correctness and efficiency.

## 4 Extra Credit

1. Decoding with Word LM (20 pts)

   (Do NOT attempt this problem until you finish everything else. This problem is *extremely* hard.) Now if you are to decode with the approach in Question 2.5 (using `eword.wfsa`), how would you do it? Describe your algorithm in pseudocode and analyse its complexity, but don't implement it!

## 5 Debrief

Please include a separate `debrief.txt` for these questions:

1. Did you work alone, or did you discuss with other students? If the latter please write down their names. Note: in general, only high-level discussions are allowed; each student still has to write the code, perform the experiments, and write the report all by him/herself.

2. How many hours did you spend on this assignment?

3. Would you rate it as easy, moderate, or difficult?

4. Are the lectures too fast, too slow, or just in the right pace?

5. Any other comments?

**Note:** You'll get 5 pts off for not including the debrief. If you have submitted but forgot to do include it before the deadline, email `debrief.txt` alone to TA Jie (`jchu1@gc.cuny.edu`) and the professor.