Natural Language Processing HW 3

Christopher Buss, Sanad Saha

1 Part-of-Speech Tagging

We implemented the viterbi algorithm in *tagging.py*. Below you can see comparisons between carmel's output (taken from our previous report) and *tagging.py's* output. We updated our *tagging.py*'s probabilities in a way that made more sense to us, so there are differences in probabilities between the outputs.

(a) They can can a can

Carmel:

```
(base) pelican04 ~/539-NLP/homework/hw2-data 1038$ echo "THEY CAN CAN A CAN" | carmel -sliOEQk 5 lexicon.wfst bigram.wfsa
Input line 1: THEY CAN CAN A CAN
        (6 states / 11 arcs)
        (9 states / 10 arcs reduce-> 7/7)
PRO AUX V DT N 0.0002
PRO V V DT N 5.00000000000001e-06
0
0
0
Derivations found for all 1 inputs
Viterbi (best path) product of probs=0.0002, probability=2^-12.2877 per-input-symbol-perplexity(N=5)=2^2.45754 per-line-perplexity(N=1)=2^12.2877
```

tagging.py:

```
(python3.6) vm-maple ~/539-NLP/homework/hw3-data 1045$ python3 tagging.py 'THEY CAN CAN A CAN'
[('THEY', 'PRO'), ('CAN', 'AUX'), ('CAN', 'V'), ('A', 'DT'), ('CAN', 'N')] # 9.449999999999998e-05
```

(b) Time flies like an arrow

Carmel:

```
(base) pelican01 ~/539-NLP/homework/hw2-data 1008$ echo "TIME FLIES LIKE AN ARROW" | carmel -sliOEQk 5 lexicon.wfst bigram.wfsa
Input line 1: TIME FLIES LIKE AN ARROW
        (6 states / 7 arcs)
        (7 states / 7 arcs)
N V PREP DT N 0.0027
N V V DT N 0.00054
0
0
0
Derivations found for all 1 inputs
Viterbi (best path) product of probs=0.0027, probability=2^-8.53282 per-input-symbol-perplexity(N=5)=2^1.70656 per-line-perplexity(N=1)=2^8.53282
```

tagging.py:

```
(python3.6) vm-maple ~/539-NLP/homework/hw3-data 1046$ python3 tagging.py 'TIME FLIES LIKE AN ARROW'
[('TIME', 'N'), ('FLIES', 'V'), ('LIKE', 'PREP'), ('AN', 'DT'), ('ARROW', 'N')] # 5.250000000000002e-07
```

(c) I hope that this works

Carmel:

```
(base) pelican04 ~/539-NLP/homework/hw2-data 1019$ echo "I HOPE THAT THIS WORKS" | carmel -sliOEQk 5 lexicon.wfst bigram.wfsa
Input line 1: I HOPE THAT THIS WORKS
        (6 states / 9 arcs)
        (8 states / 8 arcs reduce-> 7/7)
PRO V CONJ PRO V 0.002
PRO V CONJ DT V 0.002
0
0
0
Derivations found for all 1 inputs
Viterbi (best path) product of probs=0.002, probability=2^-8.96578 per-input-symbol-perplexity(N=5)=2^1.79316 per-line-perplexity(N=1)=2^8.96578
```

tagging.py:

```
(python3.6) vm-maple ~/539-NLP/homework/hw3-data 1047$ python3 tagging.py 'I HOPE THAT THIS WORKS'
[('I', 'PRO'), ('HOPE', 'V'), ('THAT', 'CONJ'), ('THIS', 'PRO'), ('WORKS', 'V')] # 9.375000000000002e-08
```

(d) They can fish

Carmel:

```
Viterbi (best path) product of probs=0.002, probability=2^-8.96578 per-input-symbol-perplexity(N=5)=2^1.79316 per-line-perplexity(N=1)=2^8.96578
(base) pelican04 ~/539-NLP/homework/hw2-data 1046$ echo "THEY CAN FISH" | carmel -sliOEQk 5 lexicon.wfst bigram.wfsa
Input line 1: THEY CAN FISH
        (4 states / 6 arcs)
        (7 states / 8 arcs reduce-> 5/5)
PRO AUX V 0.01
PRO V N 0.0045
0
0
0
Derivations found for all 1 inputs
Viterbi (best path) product of probs=0.01, probability=2^-6.64386 per-input-symbol-perplexity(N=3)=2^2.21462 per-line-perplexity(N=1)=2^6.64386
```

tagging.py:

```
(python3.6) vm-maple ~/539-NLP/homework/hw3-data 1048$ python3 tagging.py 'THEY CAN FISH'
[('THEY', 'PRO'), ('CAN', 'AUX'), ('FISH', 'V')] # 0.00045
```

2 Decoding Katakana to English Phonemes

1. **Describe your algorithm in English first**

Before talking about our algorithm we would like to focus on the Recurrence relation of our trigram based Viterbi algorithm, on which our algorithm is based on.

The recurrence relation is:

$$opt(i,\ e,\ e') = max\ (for\ all\ e)\ max\ (k = 1,\ 2,\ 3)\ \{opt(i - k,\ e',\ e'') * P(e \mid e'\ e'') * P\ (J_{i-k+1} \dots J_i \mid e)\}$$

Where Opt(i, e, e') is our memoization which stores probability of best English phoneme upto *i*, given current English phoneme is *e* and its previous phoneme *e'*. The base case for our recurrence relation is

$$opt\ (0,\ <s>,\ <s>) = 1$$

Steps:

1. We created 2 python dictionaries: peprob (from epron.probs) [which is a trigram model] and pwords (from epron-jpron.probs) so that we can get our $P(e \mid e'\ e'')\ and\ P(J_i \mid e)\ in$ O(1) from the python dictionary. And we used them in our recurrence relation.

In our case P(e | e’ e”) is stored: pepron[e][eprev][eprevprev] = Probability from epron.probs file.
P(Ji | e) is stored: pwords[Ji][e] = probability from epron-jpron.probs file

2. Now, in our algorithm we edit the given text by adding <s> at the start and </s> at the end of the text.

3. We added P[</s>][</s>] = 1 to our pwords dictionary as it was not in the epron-jpron.probs file.

4. We intitialized our base case as opt[0][‘<s>’][‘<s>’] = 1

5. Then our algorithm is mostly the modified Viterbi Algorithm (Dynamic Programming). We needed to modify the viterbi for Trigram model. We are given a trigram English phoneme model, and also we know that 1 Japanese phoneme can map upto 3 English phonemes.
   So at each step i, We tried to map Japanese phoneme Ji, Ji-1 Ji, Ji-2 Ji-1 Ji each with the English phoneme e. We used our dictionary pword to get $P(J_i \mid e),\ P(J_{i-1},\ J_i \mid e),\ P(J_{i-2},\ J_{i-1},\ J_i \mid e)$
   As these J were gotten from the input it was easy to calculate.

6. Then we generated opt (i, e, e’) for all possible e, e’ and e”. We used English phonemes generated from our previous calculation to make our program run faster. Rather than using all possible English Phoneme tags.

7. At each DP state, we also stored information about e” and k (in beste(i, e, e’) = e”, bestk(i, e, e’) = k) so that we can backtrack our result and print the English phoneme sequence for the given japanese phonemes.

8. After generating the DP table. We did a brute force on the table to find out which is our best-1 result. And our last 2 tag (e’, e”).

9. We used the last 2 tag to recursively find out our best path with the help of beste and bestk.

2.Define the subproblem and recurrence relations

The subproblem of finding the most likely path to any vertex is finding the most likely path to the vertex’s predecessors. After finding the best path to a vertex’s predecessors, we can simply choose the predecessor with the best path and take the edge connecting the vertex and its best predecessor, defining the most probable path to the vertex.

The recurrence relation is:

$$opt(i,\ e,\ e') = max\ (for\ all\ e)\ max\ (k = 1,\ 2,\ 3)\ \{opt(i - k,\ e',\ e'') * P(e \mid e'\, e'') * P\ (J_{i-k+1} \dots J_i \mid e)\}$$

Where Opt(i, e, e’) is our memoization which stores probability of best English phoneme upto *i*, given current English phoneme is *e* and its previous phoneme *e’*. The base case for our recurrence relation is

$$opt\,(0,\ <s>,\ <s>) = 1$$

3. Analyse its complexity

Time complexity of our algorithm is $O(k * w * t ^ 3)$.
Where w is the length of given japanese phoneme sequence and t is the length of possible english phonemes. As k is set to 3 in our program and it is a constant and Big-O is asymptotic we can say the time complexity to be $O(w * t ^ 3)$

The space complexity of our algorithm is $O(w * t ^ 2)$, because we store the opt(i, e, e').

3 K-Best Output

We modified *decode.py* to create *kbest.py* which generates the *k* most probable encodings with the help of a heap. *kbest.py* is called using the following syntax,

kbest.py <epron_prob_file> <epron_jpron_prob_file> <k_best>

We compared our output for *k=10* to that of carmel's and found no difference except for our probabilities being more precise (described with more decimal places).