Context Sensitive Hidden Markov Models

Tiffany Ding

CS181 Fall 2019 Final Project

Summary

Context sensitive hidden Markov models (csHMMs) can be used to capture long-range dependencies between distant symbols, thus addressing one of the major limitations of traditional hidden Markov models. For the first part of my project, I explore how csHMMs can be used to generate palindromes of different types. For the second part of my project, I implement a solution to the optimal alignment problem for csHMMs and demonstrate its validity by running it on several examples. Finally, I discuss the limitations of this algorithm and potential directions for future work.

Introduction

Hidden Markov models (HMMs) are useful for modeling Markov processes with unobservable states, and they have been applied to many problems, such as speech recognition. However, a weakness of traditional HMMs is that they cannot capture long-range interactions between observed symbols. In 2006, Byung-Yun Yoon and P. P. Vaidyanathan proposed a modified version of a traditional HMM, called a *context-sensitive HMM* (csHMM), that is capable of capturing long-range pairwise correlations between symbols [1].

Traditional HMMs have unobservable hidden states that each have a probability distribution for emitting symbols [2]. State transitions in HMMs have the Markov property, which means that the probability of transitioning from one hidden state to another only depends on the current state. csHMMs build off the framework of traditional HMMs, but they include additional memory to allow for the modeling of more complex relationships.

csHMMs have three types of states:

- $\boldsymbol{S_n}$: A *single emission state*. This is the same as a normal state in a traditional HMM.

- ***P_n***: A *pairwise-emission state*. Symbols emitted in P_n are pushed onto a stack called Z_n.
- ***C_n***: A *context-sensitive state* that is paired with P_n. The symbol emission probabilities of C_n and the transition probabilities from C_n to other states is *dependent on the contents of the stack* Z_n (usually only dependent on the top element). Each time we enter C_n, we pop the first element off the stack Z_n.

By allowing the probabilities associated with C_n to depend on the symbols emitted by a previous state, P_n, we are able to enforce relationships between non-adjacent symbols. One use case of csHMMs is generating palindromes, which cannot be done using traditional HMMs. Figure 1 depicts a palindrome generating csHMM. The symbol emission probabilities of C_1 are defined so that C_1 emits the symbol at the top of Stack 1 with probability 1 (the symbol is then removed from the stack after it is emitted). The transition probabilities of C_1 are defined so that C_1 transitions to itself as long as Stack 1 is not empty and C_1 transitions to the end state once Stack 1 is empty. In the following sections, I present a Python implementation of a palindrome-generating csHMM and investigate how changing the symbol emission probabilities and transition probabilities affects the characteristics of the palindromes that are generated.

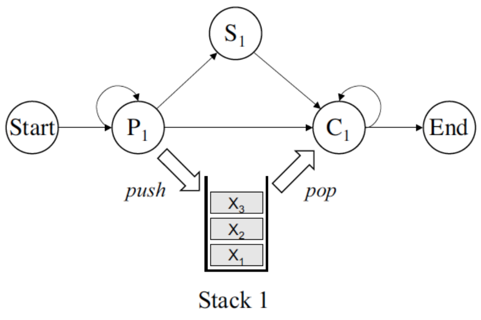

Figure 1. A csHMM that generates palindromes [1].

More generally, csHMMs can be used to model context sensitive grammars (hence the name *context sensitive* hidden Markov model) [3]. Traditional HMMs can only represent regular grammars, which have less representational power than context sensitive grammars. As described above, context sensitive grammars can be used to describe the language consisting of only palindromes. Context sensitive grammars can also be used to describe the “copy language,” in which each string consists of a string of a’s and b’s followed by a pair of c’s followed by a copy of the first string of a’s and b’s. Examples of strings in the “copy language” are *cc, acca, abaccaba,* and *bbabccbbab*. Regular grammars are incapable of representing the copy language.

Now that we have an understanding of how csHMMs can be used, an interesting question that arises is: Given a csHMM and a sequence of symbols emitted by the csHMM, can we determine what sequence of states most likely generated this symbol sequence? We call this problem the *optimal alignment problem*, because we are trying to determine the best alignment between the symbol sequence and the csHMM. With traditional HMMs, we solve the optimal alignment problem using the Viterbi algorithm. To solve the optimal alignment problem for csHMMs, we draw inspiration from the dynamic programming approach taken in the Viterbi algorithm. One difference is that we build our optimal alignment by starting from the middle of the sequence and moving out instead of by starting from the left side of the sequence and moving right like we do with traditional HMMs. However, there are many special cases to consider due to the added complexity of csHMMs. Following my description of my palindrome-generating csHMM, I will present a Python implementation of the optimal alignment problem for csHMMs

Methods

Palindrome-generating csHMM

To implement a palindrome-generating csHMM, I implemented a function called `generate_palindrome` which calls a helper function called `gen_next_state_and_symbol` that takes in the current state and Z1 (the stack associated with P1 and C1) as arguments. `gen_next_state_and_symbol` then generates the symbol emitted by the current state and the next state according to the symbol emission probabilities and transition probabilities, respectively. When a symbol is generated in state P1, the symbol is pushed onto the stack Z1. The symbol emission probabilities and transition probabilities of C1 are dependent on the contents of Z1. We continue generating symbols and next states until we reach the 'end' state. The symbol emission probabilities are defined within `gen_next_state_and_symbol`, but they can be changed to produce different types of palindromes (e.g. longer palindromes or palindromes with predominantly one character). The only requirement is that the symbol emission probability of S1 must be defined so that S1 transitions to C1 with probability 1 (otherwise, the output may not be a palindrome).

Optimal alignment algorithm

My implementation of the optimal alignment algorithm for csHMMs closely follows the algorithm presented in Part III of Yoon and Vaidyanathan's paper [1]. Each state (e.g. P1, S1, C1) is a string, but for convenience, I assign each state an integer index to make iteration easier. I implement a function called `most_likely_state_sequence`, which takes the following arguments: (1) A string containing a sequence of emitted symbols. (2) A map from integer indices to states. (3) Symbol emission probabilities for each state, represented as nested dictionaries. The symbol emission probabilities for C states are a special case because we must use an additional level of nested dictionaries in order to capture how the symbol emission probabilities depend on the symbol emitted by the corresponding P state. (4) A list of indices that correspond to S states. (5) A list of indices that correspond to P states. (6) A list of indices that correspond to C states. (7) Transition probabilities for each state, which are represented using nested dictionaries. For additional explanation, see the comments in my code.

Results

Palindrome-generating csHMM

In my code, I define a csHMM with the following symbol emission probabilities for P1 and S1:

P(a | P1) = 0.5
P(b | P1) = 0.5

P(a | S1) = 0.5
P(b | S1) = 0.5

And the following state transition probabilities for P1:

P(P1→P1) = 0.5
P(P1→S1) = 0.25
P(P1→C1) = 0.25

We get the following output after running this csHMM a few times:

bb	baaaab	bab	bab	bb	aaa	aba	baab

To create longer palindromes, we can increase the probability that P1 will transition back to itself and decrease the probability that P1 will transition to S1 or C1. I define a second csHMM with the same symbol emission probabilities as before but update the transition probabilities for P1 to be:

P(P1→P1) = 0.9
P(P1→S1) = 0.05

P(P1→C1) = 0.05

Here is the output of running this csHMM a few times:

bbbaabbaabbb	bababbabbaababbaabbaabbabaabbabbaba	abaabbabbbabbaaba

These palindromes are clearly longer than the palindromes we generated with the original csHMM.

To create palindromes that contain more a's than b's, we can adjust the symbol emission probabilities. I define a third csHMM with the transition probabilities that we used originally but with the following symbol emission probabilities:

P(a | P1) = 0.9
P(b | P1) = 0.1

P(a | S1) = 0.8
P(b | S1) = 0.2

Here is the output of running this csHMM a few times:

aaaa	aaa	aaa	bb	aa	aa	aa	aaaaa

As you can see, these strings now contain predominantly a's.

Optimal alignment algorithm

Below, I present the results of running my implementation of the optimal alignment algorithm on several different csHMMs and compare these results to my manual calculations.

Example 1:

I implemented the csHMM presented in Figure 5 of Yoon and Vaidyanathan's paper [1]. The transition diagram from their paper is included below for reference (See Figure 2). Given the symbol sequence *abbba*, there are only two possible state sequences that could have emitted this symbol sequence. I call these state sequences s_1 and s_2, where s_1 = P1, S1, S1, S1, C1 and s_2 = P1, P1, S1, C1, C1. I performed the manual calculation of P(s_1 | *abbba*) = P(Start→P1)P(a | P1)P(P1→S1)P(b | S1)P(S1→S1)P(b | S1)P(S1→S1) P(b | S1)P(S1→C1)P(a | C1, P1 emitted a)P(C1→End) = 1(0.5)(0.5)(.75)(.5)(.75)(.5)(.75)(.5)(1)(1) = .01318. Similarly, I manually calculated P(s_2 | *abbba*) = .023438. Since P(s_2 | *abbba*) > P(s_1 | *abbba*), we conclude that s_2 = P1, P1, S1, C1, C1 is the sequence of states that most likely produced the symbol sequence *abbba*.

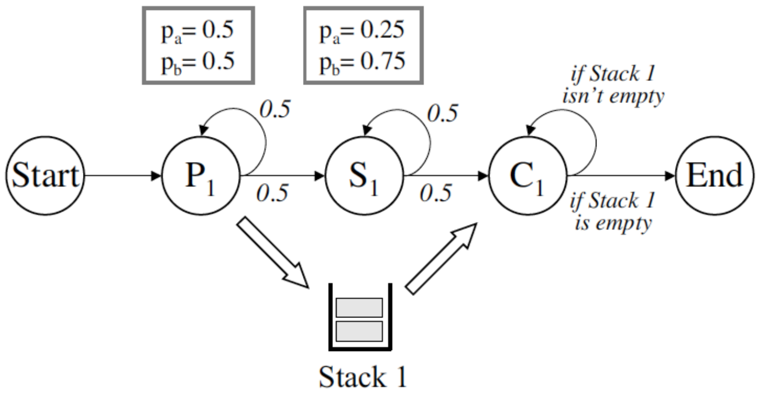

Figure 2. The csHMM I implemented for Example 1.

Here is the output of my program for Example 1:

```
    Log probability of most likely state sequence: -3.7534179752515078
    Most likely state sequence: ['P1', 'P1', 'S1', 'C1', 'C1']
```

Note that ln(.023438)= -3.754, so the probability and state sequence outputted by my code are consistent with my manual calculations.

Example 2:

In this example, I wanted to test how my code would handle multiple (P_n, C_n) pairs and nested pairwise interactions, so I designed the csHMM shown in Figure 3. I defined the symbol emission probabilities to be:

P(a | P1) = .8
P(b | P1) = .2

P(a | P2) = .5
P(b | P2) = .5

P(a | S1) = .75
P(b | S1) = .25

C1 emits the symbol emitted by P1 with probability 1.
C2 emits the symbol emitted by P2 with probability 1.

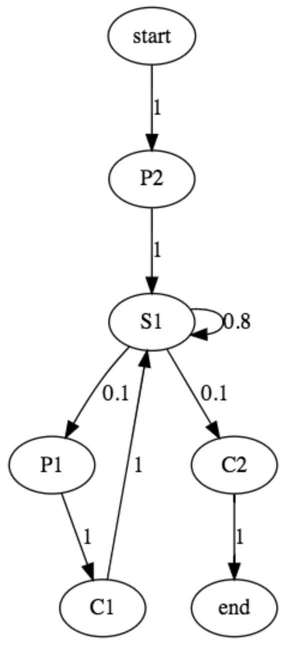

Figure 3. The csHMM I implemented for Example 2.

The symbol sequence I tested was *abaaaa*. There are only two possible state sequences that could produce this symbol sequence: s_1 = P2, S1, S1, S1, S1, C2 and s_2 = P2, S1, P1, C1, S1, C2. Through manual calculation, I determined that P(s_1 | *abaaaa*) = .0027 and P(s_2 | *abaaaa*) = .00075. Therefore, s_1 is more likely.

Here is the output of my program for Example 2:

```
Log probability of most likely state sequence: −5.914503505971853
Most likely state sequence: ['P2', 'S1', 'S1', 'S1', 'S1', 'C2']
```

Note that ln(.0027) = -5.9125, so the output of my program matches my manual calculations.

Example 3:

For Example 3, I use the csHMM from Example 2 but update the transition probabilities for S1 so that it is now less likely for S1 to transition to itself and more likely for S1 to transition to P1. Figure 4 shows this csHMM, which is identical to Figure 3 except for the probabilities on the edges coming out of S1.

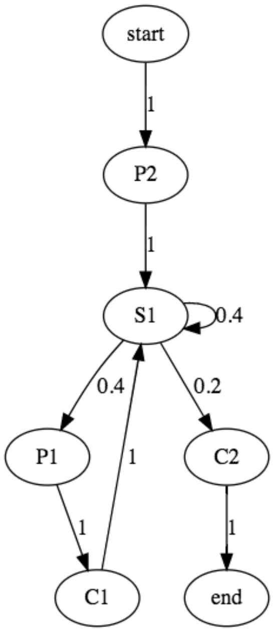

Figure 4. The csHMM I implemented for Example 3.

As in Example 2, I test the symbol sequence *abaaaa*. There are only two possible state sequences that could produce this symbol sequence: s_1 = P2, S1, S1, S1, S1, C2 and s_2 = P2, S1, P1, C1, S1, C2. Through manual calculation, I determined that P(s_1 | *abaaaa*) = .000675 and P(s_2 | *abaaaa*) = .006. Therefore, s_2 is more likely.

Here is the output of my program for Example 2:

```
Log probability of most likely state sequence: -5.115995809754081
Most likely state sequence: ['P2', 'S1', 'P1', 'C1', 'S1', 'C2']
```

Note that ln(.006) = -5.1160, so the output of my program matches my manual calculations.

Discussion

Palindrome generation is an interesting application of csHMMs, and I found that varying the symbol emission probabilities and transition probabilities of my csHMM could change the characteristics of the palindromes that were produced.

By implementing Yoon and Vaidyanathan's algorithm for solving the optimal alignment problem, I was able to verify that their algorithm works and produces outputs that are consistent with manual calculations. Overall, this algorithm provides a good solution to the optimal alignment problem for csHMMs. However, there are two major limitations to Yoon and Vaidyanathan's algorithm. The first limitation is that the algorithm only considers state sequences in which all P_n's and C_n's are matched. The second limitation is that it cannot handle pairwise interactions that cross each other (e.g., the sequence P1, P2, C1, C2 is considered impossible). There may be cases in which the true optimal alignment contains unmatched P_n's or C_n's or contains intersecting pairwise interactions. In these cases, Yoon and Vaidyanathan's algorithm will not produce the true optimal alignment. Figuring out how to overcome these two limitations is a good direction for future work.

References

[1] B. Yoon and P.P. Vaidyanathan, *Context-Sensitive Hidden Markov Models for Modeling Long-Range Dependencies in Symbol Sequences*, IEEE Transactions on Signal Processing, vol 54 (11), Nov. 2006.

[2] L. R. Rabiner, *A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition, Proceedings of the IEEE*, vol. 77 (2), Feb 1989.

[3] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison, *Biological sequence analysis: Probabilistic models of proteins and nucleic acids*. Cambridge, United Kingdom, Cambridge University Press, 1998.

Appendix

I. Reflection

Throughout the course of this project, I faced several challenges.

First, there was the challenge of how to represent symbol emission probabilities and transition probabilities. This was especially hard for C states, because the probabilities associated with C states are dependent on the symbols emitted by the corresponding P states. After trying a few different approaches, I decided to use nested dictionaries, which ended up working quite well.

Second, I had to deal with the challenge of understanding the algorithm. Even though I was given the pseudocode for the algorithm, understanding the pseudocode was not an easy task. I initially tried to implement the algorithm without really understanding how the algorithm worked, which resulted in hours of debugging. There are certain parts of the pseudocode that are unclear or lack detail, and I had to run through the algorithm by hand a few times in order to make an educated guess as to how to fill in the missing gaps in the pseudocode. One example is the notation used to describe the iteration step of the algorithm. In the paper, it is written as "For i = 1, . . . , L − 1, j = i + 1, . . . , L and v = 2, . . . , M − 1, w = 2, . . . , M – 1." Originally, I directly translated this into four nested for loops: `for i in range(0,L)`, `for j in range(i+1,L+1)` , `for v in range(2, M)` , `for w in range(2, M)`. However, this did not work. After spending some more time working through the algorithm, I realized that the actual intent of the iteration is to first iterate through all subsequences of length one, then iterate through all subsequences of length two, then length three, and so on. With this understanding, I replaced my outer two for loops with

```
for subseq_length in range(2, L+1):
```

```
for i in range(1, L-subseq_length+2):
    j = i + subseq_length-1
```

This fixed many of the problems with my code.

Overall, I learned a lot by doing this final project and I am glad to have had the opportunity to do an in-depth exploration of context-sensitive hidden Markov models.

II. Instructions to run code

Palindrome-generating csHMM

To generate a palindrome, open a Terminal window and type:

```
python gen_palindrome.py
```

You can update the symbol emission probabilities and transmission probabilities by changing the lines marked with the comment #CAN CHANGE.

Optimal alignment algorithm

To see the algorithm's output of Examples 1-3 presented above, open a Terminal window and type:

```
python csHMM.py
```

III. Code

Palindrome-generating csHMM

```
import math
import numpy as np


def gen_next_state_and_symbol(curr_state, Z1):
    '''
    Inputs:
        curr_state = string corresponding to current state
        Z1 = stack corresponding to S1 and C1
    Outputs:
        (symbol generated in current state, next state, updated Z1)
    '''
    # List of hidden states that the csHMM can be in
    hidden_states = ['P1', 'S1', 'C1', 'end']

    # List of symbols that can be emitted by this csHMM
    symbols = ['a', 'b']

    # Symbol emission probabilities for P1: 1st entry is P(a|P1), 2nd entry is P(b|P1)
    P1_emission_probs = [0.5, 0.5] # CAN CHANGE
    # Transition probabilities for P1: Entry i = P(P1 -> Entry i of hidden_states)
    P1_transition_probs = [.5, .25, .25, 0] # CAN CHANGE
    # P1_transition_probs = [.9, .05, .05, 0]

    # Symbol emission probabilities for S1: 1st entry is P(a|S1), 2nd entry is P(b|S1)
    S1_emission_probs = [0.5, 0.5] # CAN CHANGE
    # Transition probabilities for S1: Entry i = P(S1 -> Entry i of hidden_states)
    # We must always transition from S1 to C1 to ensure that the output is a
palindrome
    S1_transition_probs = [0, 0, 1, 0]

    if curr_state == 'P1':
        # generate symbol
        symbol = np.random.choice(symbols, p=P1_emission_probs)
        Z1.append(symbol) # store symbol in Z1

        # get next state
        next_state = np.random.choice(hidden_states, p=P1_transition_probs)

        return symbol, next_state, Z1

    elif curr_state == 'S1':
        # generate symbol
        symbol = np.random.choice(symbols, p=S1_emission_probs)

        # get next state
        next_state = np.random.choice(hidden_states, p=S1_transition_probs)

        return symbol, next_state, Z1

    elif curr_state == 'C1':
        # generate symbol
        symbol = Z1.pop()

        # get next state
        if Z1 != []: # stay in C1 as long as Z1 is not empty
            next_state = 'C1'
        else:
            next_state = 'end'
        return symbol, next_state, Z1


def generate_palindrome():

    observed_symbols = []
    visited_states = ['start']

    # Z1 is the stack associated with P1 and C1
    Z1 = []
    # Always start transition from start to P1
    curr_state = 'P1'
    visited_states.append(curr_state)

    while curr_state != 'end':
        emitted_symbol, curr_state, Z1 = gen_next_state_and_symbol(curr_state, Z1)
        observed_symbols.append(emitted_symbol)
        visited_states.append(curr_state)

    print('Symbols emitted:', ''.join(observed_symbols))
    print('States visited:', visited_states)

if __name__ == '__main__':
    generate_palindrome()
```

Optimal alignment algorithm

```
import math
import numpy as np

# Elements of a csHMM:
# S = single-emission states (a normal hidden state)
# P = pairwise-emission states. Symbol emitted at Pn is stored in Zn
# C = context-sensitive states
# Note: |P| = |C|

# Pn and Cn are associated with stack Zn

def log(x):
    if x > 0:
        return math.log(x)
    else:
        return float("-inf")

# Solution to the optimal alignment problem (Finding the most probable sequence of
states)
# Assumption 1: Pn and Cn always come in pairs
# Assumption 2: Pn and Cn interactions are nested (do not cross each other)


def most_likely_state_sequence(seq, index_to_state, e, S, P, C, t):
    '''
    Inputs:
    seq: a string containing a sequence of emitted symbols
    index_to_state: dict from index (integer) to state (string)
    e = emission probabilities (nested dicts). e[v][s_i] = P(emit symbol s_i in state
v)
    S = list of keys in e that correspond to S states
    P = list of keys in e that correspond to P states. ith element of list = P_i
    C = list of keys in e that correspond to C states. ith element of list = C_i
    t = transition probabilities (nested dicts). t[x][y] = P(x -> y)
    '''
    L = len(seq)
    M = len(e)
    s = " " + seq # add space to allow for easy 1-indexing
    # 1) Initialization
    # log_prob(i,j,v,w) = log probability of the optimal (most likely) path with s_i=v
and s_j=w
    # backtraceL and backtraceR are used to back trace the optimal path
    # + 1 to allow for easy 1-indexing (row 0 and col 0 will be unused)
    log_prob = np.full((L+1, L+1, M+1, M+1), float('-inf'))
    backtraceL = np.full((L+1, L+1, M+1, M+1), None)
    backtraceR = np.full((L+1, L+1, M+1, M+1), None)

    # Probability of sequence of length 1:
    for i in range(1,L+1): # for i = 1, ..., L
        for v in range(2, M): # for v = 2, ..., M-1
            if v in S:
                log_prob[i,i,v,v] = log(e[v][s[i]])
            else:
                log_prob[i,i,v,v] = float("-inf")
            backtraceL[i,i,v,v] = (0,0,0,0)
            backtraceR[i,i,v,v] = (0,0,0,0)

    # 2) Iteration
    # Intuition: First iterate through all subsequences of length 2
    #   and try all combinations of states. Then iterate through all
    #   subsequences of length 3, building off the optimal subsequences
    #   of length 3. And so on.
    for subseq_length in range(2, L+1):
        # i = start of subsequence
        # j = end of subsequence
        for i in range(1, L-subseq_length+2):
            j = i + subseq_length-1
            for v in range(2, M): # for v = 2, ..., M-1
```

```
                for w in range(2, M): # for w = 2, ..., M-1

                    # Case (i)
                    if v in C or w in P:
                        log_prob[i,j,v,w] = float("-inf")
                        backtraceL[i,i,v,v] =  (0,0,0,0)
                        backtraceR[i,i,v,v] = (0,0,0,0)

                    # Case (ii) or Case (viii)
                    elif (v in P and w in S) or (v in S and w in S):
                        temp_list = []
                        for u in range(2, M): # for u = 2, ..., M-1
                            temp_list.append(log_prob[i,j-1,v,u] + log(t[u][w]) +
log(e[w][s[j]]))
                        log_prob[i,j,v,w] = np.max(temp_list)
                        u_star = np.argmax(temp_list) + 2 # + 2 to get proper index
                        backtraceL[i,j,v,w] =  (i,j-1,v,u_star)
                        backtraceR[i,j,v,w] =  (j,j,w,w)

                    # Case (iii)
                    elif v in S and w in C:
                        temp_list = []
                        for u in range(2, M): # for u = 2, ..., M-1
                            temp_list.append(log(e[v][s[i]]) + log(t[v][u]) +
log_prob[i+1,j,u,w])
                        log_prob[i,j,v,w] = np.max(temp_list)
                        u_star = np.argmax(temp_list) + 2 # + 2 to get index
                        backtraceL[i,j,v,w] =  (i,i,v,v)
                        backtraceL[i,j,v,w] =  (i+1,j,u_star,w)

                    # Case (iv)
                    elif v in P and w in C and P.index(v) != C.index(w) and j<i+3:
                        log_prob[i,j,v,w] = float("-inf")
                        backtraceL[i,i,v,v] =  (0,0,0,0)
                        backtraceR[i,i,v,v] = (0,0,0,0)

                    # Case (v)
                    elif v in P and w in C and P.index(v) != C.index(w) and j>=i+3:
                        # v_bar = the C state that is paired with the P state that v
corresponds with
                        v_bar = C[P.index(v)]
                        temp_matrix = []
                        for u in range(2, M): # for u = 2, ..., M-1
                            temp_list = []
                            for k in range(i+1,j-1): # for k = 1+1, ..., j-2
                                temp_list.append(log_prob[i,k,v,v_bar] +
log(t[v_bar][u]) + log_prob[k+1,j,u,w])
                            temp_matrix.append(temp_list)
                        temp_matrix = np.array(temp_matrix)
                        log_prob[i,j,v,w] = np.max(temp_matrix)
                        k_star, u_star = np.argwhere(temp_matrix ==
np.max(temp_matrix))[0] + (2, i+1) # argmax. Add (2, i+1) to get argmax
                        backtraceL[i,j,v,w] = (i,k_star, v, v_bar)
                        backtraceR[i,j,v,w] = (k_star + 1, j, u_star, w)

                    # Case (vi)
                    elif v in P and w in C and P.index(v) == C.index(w) and j == i+1:
                        log_prob[i,j,v,w] = log(e[v][s[i]]) + log(t[v][w]) +
log(e[w][s[i]][s[j]])
                        backtraceL[i,i,v,v] =  (0,0,0,0)
                        backtraceR[i,i,v,v] = (0,0,0,0)

                    # Case (vii)
                    elif v in P and w in C and P.index(v) == C.index(w) and j > i+1:
                        v_bar = C[P.index(v)]
                        temp_matrix1 = []
                        for u in range(2, M): # for u = 2, ..., M-1
                            temp_list = []
                            for k in range(i+1,j-1): # for k = i+1, ..., j-2
                                temp_list.append(log_prob[i,k,v,v_bar] +
log(t[v_bar][u]) + log_prob[k+1,j,u,w])
                            temp_matrix1.append(temp_list)
                        temp_matrix1 = np.array(temp_matrix1)
                        if temp_matrix1.size > 0:
                            gamma_1 = np.max(temp_matrix1)
                            k_star, u_star = np.argwhere(temp_matrix1 ==
np.max(temp_matrix1))[0] + (i+1,2)
                        else:
                            gamma_1 = float("-inf")

                        temp_matrix2 = []
                        for u1 in range(2, M): # for u1 = 2, ..., M-1
                            temp_list = []
                            for u2 in range(2, M): # for u2 = 2, ..., M-1
                                temp_list.append(log(e[v][s[i]]) + log(t[v][u1]) +
                                    log_prob[i+1,j-1,u1,u2] + log(t[u2][w]) +
log(e[w][s[i]][s[j]]))
                            temp_matrix2.append(temp_list)
                        temp_matrix2 = np.array(temp_matrix2)
                        gamma_2 = np.max(temp_matrix2)
                        u1_star, u2_star = np.argwhere(temp_matrix2 ==
np.max(temp_matrix2))[0] + (2,2)

                        if gamma_1 > gamma_2:
                            log_prob[i,j,v,w] = gamma_1
                            backtraceL[i,j,v,w] = (i, k_star, v, w)
                            backtraceR[i,j,v,w] = (k_star + 1, j, u_star, w)
                        else:
                            log_prob[i,j,v,w] = gamma_2
                            backtraceL[i,j,v,w] = (i+1, j-1, u1_star, u2_star)
                            backtraceR[i,j,v,w] = (0,0,0,0)

    # 3) Termination
    temp_matrix = []
    for v in range(2,M): # for v = 2, ..., M-1
        temp_list = []
        for w in range(2,M): # for w = 2, ..., M-1
            temp_list.append(log(t[1][v]) + log_prob[1,L,v,w] + log(t[w][M]))
        temp_matrix.append(temp_list)
    temp_matrix = np.array(temp_matrix)
    highest_prob = np.max(temp_matrix)
    v_star, w_star = np.argwhere(temp_matrix == highest_prob)[0] + (2,2) # ??? to 1-
index
    lambda_star = (1, L, v_star, w_star)
    print("Log probability of most likely state sequence:", highest_prob)
    if highest_prob == float('-inf'):
        print("It is impossible for the input symbol sequence to be generated by this
csHMM")
        return

    # 4) Traceback

    # state_seq = most likely sequence of states
    state_seq = np.zeros((L+1),dtype='uint8') # Use L+1 to allow for easy 1-indexing
    T = []
    T.append(lambda_star)
    while T != []:
        i,j,v,w = T.pop()
        if (i,j,v,w) != (0,0,0,0):
            if state_seq[i] == 0:
                state_seq[i] = v
            if state_seq[j] == 0:
                state_seq[j] = w
            if backtraceL[i,j,v,w] != None:
                T.append(backtraceL[i,j,v,w])
            if backtraceR[i,j,v,w] != None:
                T.append(backtraceR[i,j,v,w])

    # convert from indices to state names
    state_seq = [index_to_state[x] for x in  state_seq[1:]] # Do [1:] to skip over
first state that was added for 1-indexing purposes
    print("Most likely state sequence:", state_seq)
```

```
# Example 1 (example from Figure 5 of Yoon's paper)
def example1():

    seq = "abbba"

    # Assign an index to each state
    index_to_state = {1: 'start', 2: 'P1', 3: 'S1', 4: 'C1', 5: 'end'}

    # symbol emission probabilities
    e = {1: {}, # start
        2: {'a':.5, 'b':.5}, # P1
        3: {'a':.25,'b':.75}, # S1
        4: {'a':{'a':1, 'b':0}, 'b':{'a':0, 'b':1}}, # C1
        5: {}} # end
    S = [3]
    P = [2]
    C = [4]
    t = {1: {1:0, 2:1, 3:0, 4:0, 5:0},
        2: {1:0, 2:.5, 3:.5, 4:0, 5:0},
        3: {1:0, 2:0, 3:.5, 4:.5, 5:0},
        4: {1:0, 2:0, 3:0, 4:1, 5:1}}
    most_likely_state_sequence(seq, index_to_state, e, S, P, C, t)

# Example 2 - more complicated, has multiple Pn, Cn pairs
def example2():
    seq = "abaaaa"

    # Assign an index to each state
    index_to_state = {1: 'start', 2: 'P1', 3: 'P2', 4: 'C1', 5: 'C2', 6: 'S1', 7:
'end'}
    # symbol emission probabilities
    e = {1: {}, # start
        2: {'a':.8, 'b':.2}, # P1
        3: {'a':.5,'b':.5}, # P2
        4: {'a':{'a':1, 'b':0}, 'b':{'a':0, 'b':1}}, # C1 : e[4][x][y] = P(C1 emits y
given that P1 emitted x)
        5: {'a':{'a':1, 'b':0}, 'b':{'a':0, 'b':1}}, # C2
        6: {'a':.75,'b':.25}, # S1
        7: {}} # end
    # Define what state type each state is
    S = [6]
    P = [2, 3]
    C = [4, 5]
    t = {1: {1:0, 2:0, 3:1, 4:0, 5:0, 6:0, 7:0}, # Always transition from start to P2
        2: {1:0, 2:0, 3:0, 4:1, 5:0, 6:.8, 7:0}, # P1 always transitions to C1
        3: {1:0, 2:0, 3:0, 4:0, 5:0, 6:1, 7:0}, # P2 always transitions to S1
        4: {1:0, 2:0, 3:0, 4:0, 5:0, 6:1, 7:0}, # C1 always transitions to S1
        5: {1:0, 2:0, 3:0, 4:0, 5:0, 6:0, 7:1}, # C2 always transitions to the end
        6: {1:0, 2:.1, 3:0, 4:0, 5:.1, 6:.8, 7:0}} # S1 transitions to itself with 
high probability but it can also transition to P1 and C2
    most_likely_state_sequence(seq, index_to_state, e, S, P, C, t)


# Example 3 - Same as Example 2 except with different transition probabilities for S1
def example3():
    seq = "abaaaa"

    # Assign an index to each state
    index_to_state = {1: 'start', 2: 'P1', 3: 'P2', 4: 'C1', 5: 'C2', 6: 'S1', 7: 
'end'}
    # symbol emission probabilities
    e = {1: {}, # start
        2: {'a':.8, 'b':.2}, # P1
        3: {'a':.5,'b':.5}, # P2
        4: {'a':{'a':1, 'b':0}, 'b':{'a':0, 'b':1}}, # C1 : e[4][x][y] = P(C1 emits y 
given that P1 emitted x)
        5: {'a':{'a':1, 'b':0}, 'b':{'a':0, 'b':1}}, # C2
        6: {'a':.75,'b':.25}, # S1
        7: {}} # end
    # Define what state type each state is
    S = [6]
    P = [2, 3]
    C = [4, 5]
    t = {1: {1:0, 2:0, 3:1, 4:0, 5:0, 6:0, 7:0}, # Always transition from start to P2
        2: {1:0, 2:0, 3:0, 4:1, 5:0, 6:.8, 7:0}, # P1 always transitions to C1
        3: {1:0, 2:0, 3:0, 4:0, 5:0, 6:1, 7:0}, # P2 always transitions to S1
        4: {1:0, 2:0, 3:0, 4:0, 5:0, 6:1, 7:0}, # C1 always transitions to S1
        5: {1:0, 2:0, 3:0, 4:0, 5:0, 6:0, 7:1}, # C2 always transitions to the end
        6: {1:0, 2:.4, 3:0, 4:0, 5:.2, 6:.4, 7:0}} # S1 transitions to itself with 
high probability but it can also transition to P1 and C2
    most_likely_state_sequence(seq, index_to_state, e, S, P, C, t)

def main():
    print()
    print("----- Example 1 -----")
    example1()
    print()
    print("----- Example 2 -----")
    example2()
    print()
    print("----- Example 3 -----")
    example3()
    print()
```

```python
if __name__ == "__main__":
    main()
```