50.007
Machine Learning

Project Report

Group: 42

Group members:
Loh De Rong (1003557)
Koh Ting Yew (1003339)
Tiong Shan Kai (1003469)

Table of Contents

1. Introduction **1**

2. Part 2: Maximum Likelihood **1**

2.1 Emission Parameters Estimation 1

2.2 Fixed Emission Parameters Estimation 2

2.3 Sequence Labeling 2

2.4 Results 3

3. Part 3: HMM **4**

3.1 Transition Parameters Estimation 4

3.2 Taking log-likelihood 4

3.3 Viterbi Algorithm 5

3.3.1 Initialisation 6

3.3.2 Forward Recursion 7

3.3.3 Termination 7

3.3.4 Backtracking 8

3.4 Results 8

4. Part 4: Third Best Output Sequence **10**

4.1 Modified Viterbi Algorithm 10

4.1.1 Initialization 11

4.1.2 Forward Recursion 11

4.1.3 Termination 12

4.1.4 Backtracking 13

4.2 Results 14

5. Part 5: Second Order HMM **14**

5.1 Modified Transition Parameters Estimation 15

5.2 Modified Emission Parameters Estimation 16

5.3 Modified Viterbi Algorithm 17

5.3.1 Initialisation 17

5.3.2 Forward Recursion 17

5.3.3 Termination 18

5.3.4 Backtracking 18

5.4 Results 18

References **19**

1. Introduction

This report documents our code and results for the 50.007 project. For the design challenge, we implemented the second-order HMM.

2. Part 2: Maximum Likelihood

Our code for part 2 of the project is stored in part2/hmm_part2.ipynb.

2.1 Emission Parameters Estimation

$$b(v, x) = \frac{count(v \rightarrow x)}{count(v)} \quad - \quad (1)$$

To derive the emission parameters from the training data, the file is processed line by line. Each line of the file is an observation followed by its label.

We created a function which takes in the input line by line and outputs a nested dictionary such that it looks like the following, `{{state: {'obs': count, ...}, ...}`:

Example dictionary:

```
{'B-NP': {'Municipal': 1, ... , 'Benson': 0},
'O': {',': 8705, ... ,'Huy': 0},
 ... ,
'B-LST': {'2': 2, ... , 'assumed': 0}}
```

For example, if the dictionary is called `emission_dict`, and we want to compute count(u → x) where u is a state that leads to observation x, then we would evaluate like so:

```
emission_dict['B-NP']['Municipal']
```

To compute count(u), where u is a state, it would be:

```
sum(emission_dict[state].values())
```

This allows us to compute the emission parameter b(v, x) easily, from equation (1) as seen in our code below:

```python
def get_emission_params(emission_dict, state, obs):

    if state not in emission_dict:
        raise Exception("State not in emission dict")

    state_data = emission_dict[state]

    if obs not in state_data:
        raise Exception("Word did not appear in training data")

    count_y_to_x = state_data[obs] # count(y -> x)
    count_y = sum(state_data.values()) # count(y)

    return count_y_to_x / count_y
```

2.2 Fixed Emission Parameters Estimation

This is to account for words that appear in the test set but do not appear in the train set. In this case, we return the smoothed version of the emission parameter, where we set k = 0.5.

$$b(v,\ x = '\#UNK\#') = \frac{k}{count(v) + k} \quad - \quad (2)$$

Before running the following function, such words should be first replaced by the #UNK# token.

```python
def get_emission_params_fixed(emission_dict, state, obs, k=0.5):

    if state not in emission_dict:
        raise Exception("State not in emission dict")

    state_data = emission_dict[state]
    count_y = sum(state_data.values()) # count(y)

    if obs == "#UNK#":
        count_y_to_x = k
    else:
        count_y_to_x = state_data[obs] # count(y -> x)

    return count_y_to_x / (count_y + k)
```

2.3 Sequence Labeling

For each word, this simple sequence labeling assigns it the tag with the highest probability over all tags.

```python
def label_sequence(sentence, emission_dict):
    """
    sentence - a list of Strings (words or observations).
    emission_dict - a dictionary containing emission parameters
    Returns - list of Strings (corresponding highest prob state for each word)
    """

    all_states = list(emission_dict.keys()) # all distinct states

    sequence = [] # aka tags

    for word in sentence:
        emission_state = { state: get_emission_params_fixed(emission_dict, state, word) for state in all_states }
        sequence.append(max(emission_state, key=lambda state: emission_state[state]))

    return sequence
```

2.4 Results

The precision, recall and F-scores are reported for each dataset as follows.

Entity	Dataset		
	EN	**SG**	**CN**
Precision	0.5116	0.1950	0.0812
Recall	0.7240	0.5548	0.4929
F	0.5996	0.2885	0.1395

Sentiment	Dataset		
	EN	**SG**	**CN**
Precision	0.4534	0.1251	0.0393
Recall	0.6416	0.3560	0.2386
F	0.5313	0.1851	0.0675

3. Part 3: HMM

Our code for part 3 of the project is stored in part3/hmm_part3.ipynb.

Fixed parameter estimation is identical as described in section 2.2.

3.1 Transition Parameters Estimation

$$a(v, u) = \frac{count(u,v)}{count(u)} - (3)$$

To derive the transition parameters from the training data, the file is processed line by line. Each line of the file is an observation followed by its label.

We created a function which takes in the input line by line and outputs nested dictionary such that it looks like the following, which takes the form `{{state: {'state': count, ...}, ...}`:

```
{'START': {'B-NP': 4966, ... , 'I-INTJ': 0},
'B-NP': {'I-NP': 32390, ... ,'B-VP': 6164,},
 ... ,
'I-NP': {'B-VP': 7365, ... , 'B-PP': 8544}}
```

For example, if the dictionary is called `transition_dict`, and we want to compute count(u, v) where u and v are states, then we would evaluate like so:

```
transition_dict[‘B-NP’][‘I-NP’]
```

To compute count(u), where u is a state, it would be:

```
sum(transition_dict[v].values())
```

This allows us to compute the transition parameter a(v, u) easily, from equation (3) as seen in our code below:

3.2 Taking log-likelihood

At each timestep in the original Viterbi algorithm, the new score becomes smaller as we successively multiply each probability score by the emission parameter and transition parameter. This is because the probability values and the model parameters are values ranging from 0 to 1. Therefore, if the original algorithm takes in a slightly longer sentence

with more words, the resulting score is likely to be very close to 0. This is potentially a numerical underflow issue, and the system might not be able to differentiate the argmax correctly for extremely small scores that are close to 0.

To resolve this problem, we implemented our custom log function. If any value (can be an integer, float or numpy array), it will be converted to negative infinity. Other positive values will be transformed by the numpy log function.

```
def log(m):
    if isinstance(m, float) or isinstance(m, int):
        return -np.inf if m == 0 else np.log(m)

    m = np.clip(m, 1e-32, None)
    x = np.log(m)

    x[x <= np.log(1e-32)] = -np.inf

    return x
```

Taking log-likelihood does not change the original solution because log is a monotonically increasing function. The argmax of the total score will still produce the same most optimal sequence.

For example, in terms of implementation, instead of $P_{j+1} = P_j * a(u, v) * b(u \rightarrow o)$ in the original Viterbi algorithm, we will change it to: $P_{j+1} = P_j + \log(a(u, v)) + \log(b(u \rightarrow o))$ in our Viterbi algorithm in Section 3.3.

3.3 Viterbi Algorithm

By bringing in emissions parameters from Part 2 and transition parameters from Part 3, we can construct our Viterbi algorithm.

The function signature of our Viterbi is as follows:

```
viterbi(emission_dict, transition_dict, sentence, is_preprocessed)
```

The parameter `sentence` is the input sequence of words as a List.
The parameter `is_preprocessed` is a boolean value such that if it was `False`, then it would convert the unknown words in sentence into **"#UNK#"**.

For each input sentence, or sequence, a `numpy.array` designated as `P` is created. `P` is used to store the value of the best scoring path ending with a particular state of length j. `P` is of size (n+2, |T|), where:

- n is the length of the input sequence
- |T| is the number of distinct states from the training set.

In addition, a `numpy.array` designated as `B` is created too. `B` is a backpointer table to facilitate the backtracking algorithm. `B` has the same shape as `P`.

There are n time steps. At each step, we have to compute |T| values. Each of these values is calculated from the previous layer of |T| states. With n time steps, the total time complexity is O(n|T|^2).

3.3.1 Initialisation

We initialise as follows:

```
# Pi np
P = np.ones( (len(proc_sent), len(all_states))) * -np.inf
# Backtrace np
B = [ [ None for x in all_states ] for y in proc_sent ]
```

Our P table is initialised to store negative infinity for subproblems that the Viterbi algorithm has yet to calculate. We cannot initialize the values to be 0 because in each timestep, we are always logging values less than 1 which will give a negative value. Therefore, doing so will affect our forward recursion when we take the argmax in Section 3.3.2.

The B table is initialised to store None.

```
# Base Case at j=1
t = np.array([ a('START', v) for v in all_states ])
e = np.array([ b(v, proc_sent[1]) for v in all_states ])
P[1, :] = log(t) + log(e)
B[1] = [ "START" for row in B[1] ]
```

For the B table, we have all entries in its j=1 column store ‘START’, to indicate that the preceding node of the ‘j=1’ node must be the ‘START’ node.

3.3.2 Forward Recursion

```
# Recursive Forward Step
for j in range(2, n-1): # Going right the columns (obs)
    x = proc_sent[j]  # Obtain j'th word in the (processed) sentence

    for row_no, v in enumerate(all_states): # Going down the rows (states)
        transitions = np.array([ a(u, v) for u in all_states ])
        prev_scores = P[j-1, :] + log(transitions)
        top = prev_scores.argmax()
        P[j,row_no] = prev_scores[top] + log(b(v,x))
        B[j][row_no] = all_states[top]
        if P[j,row_no] == -np.inf:
            B[j][row_no] = None
```

In the P table, for j = 2 to n-2, for each node-state we have to compute the score for the best path ending in that state. This requires retrieving the previous layer's scores, and multiplied by the corresponding transition and emission scores to obtain a vector of scores. Then find the maximum and argmax to get the best score and the best parent (i.e. preceding state) to be stored in P and B respectively.

In some cases, there can be situations where after computing the transition and emission parameters for a given path from layer j-1 to j, the resultant score becomes negative infinity. This edge case happens when the multiplied transition or the emission parameter is 0. As such, the B table would store None to indicate that no parent/preceding state is eligible.

3.3.3 Termination

```
# Termination: j=n-1. Note that proc_sent[n-1] give us the last word in sentence.
j = n-1
transitions = np.array([ a(u, "STOP") for u in all_states ])
previous_scores = P[j-1] + log(transitions)
last_state = all_states[previous_scores.argmax()]
```

The forward recursive algorithm is terminated at j=n-1. At this stage, we obtain the path with the highest score.

We then compute `last_state`, to obtain the final state in the sequence. This will be used as the starting point of our backtracking component of the algorithm subsequently.

3.3.4 Backtracking

```
# Backtrace
state_seq = ['STOP'] + [last_state]
for j in range(n-2, 0, -1):
    curr_state = state_seq[-1]
    curr_state_row_no = all_states.index(curr_state)
    prev_state = B[j][curr_state_row_no]

    if prev_state == None: # edge case
        return ['O'] * (n-2)
        break

    state_seq.append(prev_state)

state_seq = state_seq[::-1][1:-1]  # reverse and drop START, STOP
```

Working backwards from j=n-2 to 1, we have to determine the correct parent state that immediately precedes the current state. Our backtrace table B allows us to identify the preceding state that contributes to the best scoring path, i.e. the optimal state sequence. This allows us to trace back the proper sequence of states back to 'START'.

At the end of the iterations, we reverse the `state_seq` list and remove START and STOP states.

3.4 Results

The precision, recall and F-scores are reported for each dataset as follows.

Entity	Dataset		
	EN	**SG**	**CN**
Precision	0.8116	0.5212	0.2561
Recall	0.7310	0.5092	0.2843
F	0.7692	0.5151	0.2695

Sentiment	Dataset		
	EN	**SG**	**CN**
Precision	0.7742	0.4296	0.1544
Recall	0.6974	0.4197	0.1714
F	0.7338	0.4246	0.1625

4. Part 4: Third Best Output Sequence

Our code for part 4 of the project is stored in part4/hmm_part4.ipynb.

Our emissions and transition matrix training process is identical to that described in Section 2.2 and Section 3.1 respectively.

In the modified Viterbi algorithm in Section 4.1, we will also be using the log-likelihood function that is described in Section 3.2.

4.1 Modified Viterbi Algorithm

To take into account the top k sequences (in our case k=3), we made two key changes to our original Viterbi algorithm in Part 3.

1. In the forward step, for each node in the hidden layers of the HMM network, we store a sorted array containing top k scores as well as the corresponding parent nodes.
2. In the backtrace step, we recursively find the parent node u and iteratively loop through its sorted k scores to find out which one when multiplied by the parameters would give rise to the current score at node v. This chosen score will tell us the identity of the grandparent node.

The modified Viterbi algorithm now includes sorting in its computation. To determine its time complexity, we consider the sorting and non-sorting components independently first.

- Sorting complexity:
  There are n time steps. At each step, we have to sort the scores of the previous layer. In the previous layer, there are |T| nodes, with k scores already sorted in each node—for a total k|T| scores. Sorting k|T| scores using merge sort algorithm costs O(k|T|log(k|T|) time. Doing this n times, it results in a total sorting time complexity of O(nk|T|log(k|T|).

- Non-sorting complexity:
  There are n time steps. At each step, we have to compute |T| values. Each of these values are calculated from the previous layer of |T| states, each with k scores. With n steps, total non-sorting time complexity is O(nk|T|^2).

Total time complexity: O(nk|T|log(k|T|) + nk|T|^2) = O(nk|T|^2).

4.1.1 Initialization

We begin by initializing our probability table P and our backtrace table B. In terms of space complexity, they are both O(nTk).

```
# Pi Table
P = np.ones( (len(proc_sent), len(all_states), k) ) * -np.inf
# Backtrace Table
B = [ [ [None] * k for x in all_states ] for y in proc_sent ]
```

Our P table is initialised to store negative infinity for subproblems that the Viterbi algorithm has yet to calculate. As previously explained in Section 3.1.1, we cannot initialize the values to be 0 because in each timestep, we are always logging values less than 1 which will give a negative value. Therefore, doing so will affect our forward recursion when we take the argmax in Section 3.3.2.

The B table is initialised to store None.

```
# Base Case at j=1
t = np.array([ a('START', v) for v in all_states ])
e = np.array([ b(v, proc_sent[1]) for v in all_states ])
P[1, :, 0] = log(t) + log(e)
B[1] = [ ["START"] + [None] * (k-1) for row in B[1] ]
```

For the B table, we have all entries in its j=1 column store [‘START’, None, None], to indicate that for each node in j=1, the only best possible path must have the ‘START’ node at j=0. Consequently, the second and third best path for j=1 would have None state at j=0, because no other paths exist.

4.1.2 Forward Recursion

```
for j in range(2, n-1): # Going right the columns (obs)
    x = proc_sent[j]  # Obtain j'th word in the (processed) sentence

    for row_no, v in enumerate(all_states): # Going down the rows (states)
        transitions = np.array([ a(u, v) for u in all_states ])
        previous_all_scores = (P[j-1, :] + log(transitions[:, None])).flatten()
        topk = previous_all_scores.argsort()[::-1][:k]
        P[j,row_no] = previous_all_scores[topk] + log(b(v,x))
        B[j][row_no] = [ all_states[pos // k] for pos in topk ]

        for i, sub_k in enumerate(P[j,row_no]):
            if sub_k == -np.inf:
                B[j][row_no][i] = None
```

In the P table, for j = 2 to n-2, for each node-state we have to compute the best k path scores ending at its respective state in descending order. This requires retrieving the previous layer's scores, and multiplied by the corresponding transition and emission scores to obtain a matrix of scores. Then we flatten these scores, sorted them, and obtained the best k scores which we store in the layer j node. The best k scores' index would be stored in the B table.

Note that everytime we obtain the best k scores from the previous layer given a layer j node, not all k scores would have well-defined results. An undefined score would mean that its value is negative infinity, which arose when either the transition and emission parameters responsible is 0. In such a case, for the k scores that hold values of negative infinity, the corresponding B table's position would store Nonen as the parent node. This indicates that the particular top k scoring path is improbable, and no valid sequence of states is possible that ends in that layer j's node's state.

4.1.3 Termination

```
# Termination: j=n-1. Note that proc_sent[n-1] give us the last word in sentence.
j = n-1
transitions = np.array([ a(u, "STOP") for u in all_states ])
previous_all_scores = (P[j-1] + log(transitions[:, None])).flatten()
final_topk = previous_all_scores.argsort()[::-1][:k]
final_scores = previous_all_scores[final_topk]

# top k parent STATES preceding the STOP state. By top k, means top k best scores.
final_topk_pos = [ all_states[pos // k] for pos in final_topk ]
```

The forward recursive algorithm is terminated at j=n-1. At this stage, the different scores of the full length path is computed, and we picked the top k scores.

We also obtain the top k terminal states resulting from the top k scores. Since we want the k-th best path, then we would only retrieve the k'th element in this top k terminal states, and use that as the starting point of the backtracking component of the algorithm.

4.1.4 Backtracking

```
# Backtrace
state_seq = ['STOP']

prev_states = final_topk_pos
prev_state = prev_states[-1] # you already know you want the kth best, which is the last
prev_row_no = all_states.index(prev_state)
curr_score = final_scores[-1]

# from n-2 to n-1 (STOP)
j = n-1
for i in range(k):
    prev_score = P[j-1, prev_row_no][i]
    if prev_score + log(a(prev_state, "STOP")) == curr_score:
        curr_score = prev_score
        curr_idx = i
        state_seq.append(prev_state)
        break

for j in range(n-2, 1, -1):
    x = proc_sent[j]
    curr_state = state_seq[-1]
    curr_row_no = all_states.index(curr_state)
    prev_state = B[j][curr_row_no][curr_idx]

    if prev_state == None: # No possible transition to STOP. Edge case.
        state_seq = ['O'] * (n-2)
        return P, B, state_seq

    prev_row_no = all_states.index(prev_state)

    for i in range(k):
        prev_score = P[j-1, prev_row_no][i]
        if prev_score + log(a(prev_state, curr_state)) + log(b(curr_state, x)) == curr_score:
            curr_score = prev_score
            curr_idx = i
            state_seq.append(prev_state)
            break

state_seq.append("START") # START will throw an error because it has no index in all_states
state_seq = state_seq[::-1][1:-1]  # reverse and drop START, STOP
```

As mentioned, since we want the k-th best path for Part 4, we would then only retrieve the k'th element in this top k terminal states, and use that as the starting point of the backtracking component of the algorithm.

Working backwards from j=n-2 to 1, we have to determine the correct parent state that immediately precedes the current state. Our backtrace table B allows us to locate the preceding layer's node's k scores. We then check which of these k scores, when multiplied with the corresponding transition and emission values result in an identical score as the current layer. This allows us to trace back the proper sequence of states back to 'START'.

At the end of the iterations, we reverse the `state_seq` list and remove START and STOP states.

4.2 Results

The precision, recall and F-scores are reported for EN dataset as follows.

Entity	EN
Precision	**0.7229**
Recall	**0.7082**
F	**0.7391**

Sentiment	EN
Precision	**0.7328**
Recall	**0.6714**
F	**0.7008**

5. Part 5: Second Order HMM

For part-of-speech tagging design challenge, we were told that the dataset will be EN. Hence, it makes sense to look at the grammar structure of the English language.

In the English language, there are numerous examples where the state of a given word has some level of dependency with a word two positions before. Guiding the training process with extra knowledge is useful. For example, before tagging the word 'flies' as either a verb or a noun in the sentence 'Time flies like an arrow', having its semantic meaning would make a correct tagging straightforward (Mahtab A. et al., 2018). Hence, we believe that a second order HMM will provide an improvement in prediction accuracy.

For the design challenge, we implemented second order HMM. It extends the idea of the first-order HMM by storing additional information about previous activity that can lead to better accuracy results. Second-order HMM depends on an observation from the current state as well as the transition probability function from the previous two states.

Our code for part 5 of the project is stored in part5/hmm_part5.ipynb.

5.1 Modified Transition Parameters Estimation

$$a(t,u,v) = \frac{count(t,u,v)}{count(t,u)} \quad - \quad (4)$$

Unlike the first order HMM, we need to keep track of the previous two states since the probability of transitions is dependent on both the previous and preceding states, given by $a(t,u,v)$. We introduced an additional 'PREVSTART' state due to this additional dependency. For ease of explanation, let t be the grandparent state, u be the parent state and v be the current state. These three terms will be used extensively in this section.

The parameters of our model were estimated using equation (4) .

To speed up the calculation time, we stored the transition probabilities in a pandas dataframe, with the columns being v and the index being (t, u). This will allow us to perform batch calculations in our Viterbi decoding algorithm.

```
for line in lines:
    split_line = line.split()

    # Start new sequence
    if len(split_line) < 2:
        if (prev_prev_state, prev_state) not in transition_dict.keys():
            transition_dict[(prev_prev_state, prev_state)] = defaultdict(int)
        transition_dict[(prev_prev_state, prev_state)]['STOP'] += 1
        prev_prev_state = 'PREVSTART'
        prev_state = 'START'

    # Processing the current sequence
    elif len(split_line) == 2:
        curr_state = split_line[1]
        states.add(curr_state)
        if (prev_prev_state, prev_state) not in transition_dict.keys():
            transition_dict[(prev_prev_state, prev_state)] = defaultdict(int)
        transition_dict[(prev_prev_state, prev_state)][curr_state] += 1
        prev_prev_state = prev_state
        prev_state = curr_state

# Convert each count to a probability
for tu, vs in transition_dict.items():
    count_tu = sum(vs.values())
    for v in vs:
        transition_dict[tu][v] = transition_dict[tu][v]/count_tu
```

We used a dictionary to store the counts of the state transitions initially and used the counts to calculate the probabilities.

```
# Create a numpy matrix to store all the transition probabilities
np_transition_matrix = np.zeros((len(all_state_pairs), len(states)))

for i in range(len(all_state_pairs)):
    for j in range(len(states)):
        tu = all_state_pairs[i]
        v = states[j]
        if tu in transition_dict.keys():
            np_transition_matrix[i][j] = transition_dict[tu[0], tu[1]][v]
        else:
            np_transition_matrix[i][j] = 0

# Convert into DataFrame for easy indexing
df_transition_matrix = pd.DataFrame(np_transition_matrix, index = all_state_pairs, columns = states)
```

We first instantiated a 2D numpy array of zeros and added the computed probability values from the dictionary we created above into the numpy array. We then converted the numpy array into a pandas DataFrame for easy indexing later on.

5.2 Modified Emission Parameters Estimation

```
# Convert state and observation set to list
states = list(states)
observations = list(observations) + ['#UNK#'] # Add the #UNK# word into observations at the end

# Create a numpy matrix to store all the emission probabilities
np_emission_matrix = np.zeros((len(states_w_start_stop), len(observations)))
for i in range(len(states_w_start_stop)):
    state = states_w_start_stop[i]

    if state == 'PREVSTART' or state == 'START' or state == 'STOP': # These states don't have emissions
        continue

    v_count = sum(emission_dict[state].values())
    for j in range(len(observations) - 1):
        state = states_w_start_stop[i]
        obs = observations[j]
        np_emission_matrix[i][j] = emission_dict[state][obs]/v_count # count(u -> v) / v_count

    # Add Laplace smoothing of k = 0.5 for #UNK# words
    np_emission_matrix[i][len(observations) - 1] = k/(v_count + k)


# Convert to df for easy indexing
df_emission_matrix = pd.DataFrame(np_emission_matrix, index = states_w_start_stop, columns = observations)
```

The method used for estimating the emission parameter is described in equation (1). To account for unknown words in the matrix, '#UNK#' was appended to the observations list, which contains all the unique observed words in the training set. This allows us to generate values for Laplace smoothing following equation (2). Similar to the transition parameters estimation, we used a pandas DataFrame to store the values for easier manipulation later on.

5.3 Modified Viterbi Algorithm

5.3.1 Initialisation

```
# Pi Table
P = pd.DataFrame(index = all_state_pairs, columns = range(n)).fillna(-np.inf)

# Backtrace Table
B = pd.DataFrame(index = all_state_pairs, columns = range(n))
```

To take advantage of parallel computation, we used pandas DataFrames to store the P values and to store the backtracking states. We decided not to use numpy as we faced issues regarding the indexing of the states and found it easier to index with tuple pairs of states.

```
# Initialisation
P.at[("PREVSTART", "START"), 0] = 1
```

We initialize the score of the starting states to be 1. So at j = 1, we only look at state transitions where the previous two states are PREVSTART and START.

5.3.2 Forward Recursion

```
# Forward Recursive Step
for j in range(1, n-1):
    x = proc_sent[j]

    a_b = log(df_transition_matrix.multiply(df_emission_matrix[x], axis = 'columns')) # a(t,u,v) * b(v,x)
    pis = a_b.add(P[j-1], axis = 'index').astype('float64') # a(t,u,v) * b(v,x) + pi at j-1

    for curr_state in states:
        if curr_state == 'STOP':
            continue
        tu = all_state_pairs[np.argmax(pis[curr_state].values)] # Get the highest arg for t, u -> v
        score = np.max(pis[curr_state].values) # Get highest score
        P.loc[[(tu[1], curr_state)], j] = score # Store score in Pi table at (u, v)
        B.loc[[(tu[1], curr_state)], j] = tu[0] # Store t in Backtrace table at (u, v)
```

To prevent the issue of underflow, the scores are calculated using the log-likelihood function, which has been described in Section 3.2. The helper function a(t, u, v) will give the probability of transitioning to state v where the previous 2 states are t and u. We add the product of the transition and emission parameters with the previous P values and use argmax to get the best (t, u) state pair, we store the t state in the B table. We also get the highest score by using max and store it in the P table.

5.3.3 Termination

```
# Termination
j = n-1
a_b = log(df_transition_matrix['STOP']).add(P[j-1],axis = 'index').astype('float64')
tu = a_b.idxmax() # t, u -> STOP
score = a_b.max()
P.loc[[(tu[1], 'STOP')], j] = score
B.loc[[(tu[1], 'STOP')], j] = tu[0]
```

At j = n - 1, we calculate the probabilities of the last state pair (t, u) transitioning to the STOP state and get the highest score and corresponding state pair. We proceed to store it in the P table and B table respectively, indexed by state u and STOP.

5.3.4 Backtracking

```
# Backtrace
u, v = P[n-1].astype('float64').idxmax()
state_seq = []
for j in range(n-1,0,-1):
    t = B.loc[[(u, v)],j][0]
    state_seq.append(v)
    u, v = t, u
state_seq = state_seq[::-1][:-1] # Reverse and remove #STOP#
return state_seq
```

To backtrack, we first get the argmax of the P table at j = n - 1 which will give us the state pair (u, v). We iterate from j = n - 1 down to 1 and obtain the next state pair by getting the t state stored in the B table. We append v to the `state_seq` list which holds our final tag sequence. Since u has to be in the state sequence, we set state v to be u and state u to be t. For the next iteration, we get state t again using the new (u, v) pair and so on so forth.

At the end of the iterations, we reverse the `state_seq` list and remove the STOP state.

5.4 Results

The precision, recall and F-scores are reported for EN development dataset as follows. Clearly, second-order HMM (part 5) performs much better than first-order HMM (part 3).

Entity	**EN Dataset**	
	Part 3	**Part 5**
Precision	0.8116	0.8116
Recall	0.7310	0.8234
F	0.7692	0.8175

Sentiment	**EN Dataset**	
	Part 3	**Part 5**
Precision	0.7742	0.7784
Recall	0.6974	0.7897
F	0.7338	0.7840

References

Mahtab A. et al. (2018). Improving Neural Sequence Labelling using Additional Linguistic Information. Obtained from: https://arxiv.org/pdf/1807.10805.pdf