

50.007 Machine Learning Project Report

Seah Qi Yan 1004628 | Ong Kah Yuan Joel 1004366

50.007 Machine Learning Project Report

Part 1: Labelling

Part 2: Emission

Results

Part 3: Transmission and Viterbi

Results

Part 4: Top 3 Sequences

Results

Part 5: Design Challenge

Part 1: Labelling

Omitted.

Part 2: Emission

The first order of business is to use maximum likelihood estimation to estimate the emission parameters from the training set, as per the equation below:

$$e(x|y) = \frac{\text{Count}(y \rightarrow x)}{\text{Count}(y)}$$

Our implementation of this can be found in the private method `__est_emission_params()` in the class `part2`. Here, we use a dictionary `count_y`, `count_y_to_x`, and `e_x_given_y` to keep track of the total number of occurrences of `y`, total number of emissions from `y` to `x`, and the probability of an emission of an `x` for a given `y`. We first populate `count_y_to_x` by iterating through the dataset, then iterating through that to populate `count_y`. Finally, `e_x_given_y` is generated as follows:

```
#If the word token x:
for entry, count in count_y_to_x.items(): #where entry is a tuple (x,y)
    self.e_x_given_y[entry] = count/count_y[entry[1]]

#If the word token x is the special token #UNK#:
for entry, count in count_y.items(): #where entry is y
    self.e_x_given_y[("#UNK#",entry)] = 0.5/(count+0.5)
```

There is also the need to produce the tag:

$$y^* = \arg \max_y e(x|y)$$

for each word x in the sequence. This is done in the method `find_y_max_given_x`, which simply iterates through `e_x_given_y`, getting the value x that is associated with the highest probability for a given y , and associating the respective x and y as a key-value pair in the dictionary `x_max_prob`. Specifically, the code does the following:

```
for entry, prob in self.e_x_given_y.items(): #where entry is a tuple (x,y)
    if entry[0] not in x_max_prob:
        x_max_prob[entry[0]] = prob
        self.y_max_given_x[entry[0]] = entry[1]
    else:
        if x_max_prob[entry[0]] < prob:
            x_max_prob[entry[0]] = prob
            self.y_max_given_x[entry[0]] = entry[1]
```

Results

Language	Entity	Sentiment
English	Precision: 0.5996 Recall: 0.7240 F: 0.5996	Precision: 0.4461 Recall: 0.6312 Sentiment F: 0.5227
Chinese	Precision: 0.0805 Recall: 0.4886 F: 0.1383	Precision: 0.0381 Recall: 0.2314 F: 0.0655
Singapore	Precision: 0.1926 Recall: 0.5457 F: 0.2847	Precision: 0.1204 Recall: 0.3413 F: 0.1780

Full results can be found in `part_2_results.txt`

Part 3: Transmission and Viterbi

We first need to estimate the transition parameters from the training set according to the following equation:

$$q(y_i|y_{i-1}) = \frac{\text{Count}(y_{i-1}, y_i)}{\text{Count}(y_{i-1})}$$

Our implementation of this can be found in the private method `__est_transition_params()` in the class `part3`. Here, similar to `part2`, we use a dictionary `count_y`, `count_y0_to_y1`, and `p_y1_given_y0` to keep track of the total number of occurrences of state y , total number of transitions from y_0 to y_1 for state transitions in the training data, and the probability of an transition to state y_1 given a previous state y_0 . We first populate `count_y0_to_y1` by iterating through lines in the dataset, accounting for the transitions involving the "START" and "STOP" states:

```
for line in self.train_data:
    if previous_state == None:
        previous_state = "START"
        state = line[1]
        transition = (previous_state, state)
        count_y0_to_y1[(transition)] = count_y0_to_y1[(transition)] + 1
    if transition in count_y0_to_y1:
        count_y["START"] = count_y["START"] + 1
    if "START" in count_y:
        count_y["START"] = count_y["START"] + 1
```

```

elif line[1] == "":
    state = "STOP"
    transition = (previous_state, state)
    count_y0_to_y1[(transition)] = count_y0_to_y1[(transition)] + 1 if
transition in count_y0_to_y1 else 1
    previous_state = None

else:
    state = line[1]
    transition = (previous_state, state)
    count_y0_to_y1[(transition)] = count_y0_to_y1[(transition)] + 1 if
transition in count_y0_to_y1 else 1
    count_y[state] = count_y[state] + 1 if state in count_y else 1
    previous_state = state

```

We also simultaneously iterate through the lines of the training data to populate count_y. Finally, p_y1_given_y0 is generated as follows:

```

for entry, count in count_y0_to_y1.items():
    try:
        self.p_y1_given_y0[entry] = count/count_y[entry[0]]
    except:
        self.p_y1_given_y0[entry] = count/count_y[entry[0]]

```

We then need to use the Viterbi algorithm to compute the following:

$$y_1^*, \dots, y_n^* = \arg \max_{y_1, \dots, y_n} p(x_1, \dots, x_n, y_1, \dots, y_n)$$

In particular, we use the dynamic programming sequence:

$$\pi(j+1, u) = \max_v \{ \pi(j, v) \times b_u(x_{j+1}) \times a_{v,u} \}$$

where:
 π = a 2-D array
 u = current node
 v = previous node
 a = transition probability
 b = emission probability

The first step to running the Viterbi algorithm was to format the input test data such that it could be iterated through easily. Hence, we implemented this through __format_testdata() in class part3, which returned a nested list of words/sequences, with each nested list containing the words forming a sequence in the input data, as follows:

```

def __format_testdata(self):
    """makes test data iterable for viterbi
    returns input test_data as a nested list of words/sentences,
    each nested list is 1 sentence in the input data
    """
    test_sequences = [[]]
    test_data = self.test_data
    i = 0
    for line in test_data:
        if line == "": # indicates a new sequence
            test_sequences.append([])
            i += 1
        else:
            test_sequences[i].append(line.strip('\n'))
    test_sequences.pop() # remove space in last line

```

```
self.input_sequences = test_sequences
```

Our data is now ready to be input to the Viterbi algorithm. In implementing the Viterbi algorithm, we considered how Viterbi would be run for each sequence/sentence in the input test data. Hence, we decided to modularise our implementation through the methods `_mini_viterbi(input_sequence, emission_dict, transition_dict)` and `viterbi()`. `_mini_viterbi(input_sequence, emission_dict, transition_dict)` would be the method we call for each sequence to predict the most probable state sequence for the given observations based on our transition and emission parameters calculated previously on the training data. Our implementation involves storing and updating the most probable transition and emission sequence for the given observation in the nested dictionary `sequence_prob`, updating it based on the following conditions:

```
for current_state in sequence_prob[layer]:
    max_p = 0
    max_prob_prev_state = "NA"
    for previous_state in sequence_prob[layer - 1]:
        transition = (previous_state, current_state)

        if (input_sequence[layer - 1], current_state) in emission_dict.keys() and
transition in transition_dict.keys():
            p = sequence_prob[layer - 1][previous_state]["p"] * \
                transition_dict[(transition)] * \
                emission_dict[(input_sequence[layer - 1], current_state)]
        elif transition in transition_dict.keys():
            p = sequence_prob[layer - 1][previous_state]["p"] * \
                transition_dict[(transition)] * \
                0.000000000000001 # to allow initial state "NA" to be updated
        else:
            p = sequence_prob[layer - 1][previous_state]["p"] * \
                0.000000000000001 * \
                0.000000000000001 # to allow initial state "NA" to be updated

        if p > max_p:
            max_p = p
            max_prob_prev_state = previous_state

    sequence_prob[layer][current_state] = {"p": max_p, "previous":
max_prob_prev_state}
```

We first initialise the most probable `previous_state` for each layer to be "NA", after which we would update the values according to the combined emission and transition probabilities. Afterwards, we then perform backtracking to find `argmax` to output the `predicted_sequence` of state transitions based on the observation, as follows:

```

# backtracking to find argmax
current_layer = n
reverse_path = ["STOP"]
while current_layer >= 0:
    reverse_path.append(sequence_prob[current_layer + 1]
[reverse_path[len(reverse_path) - 1]]["previous"])
    # just means taking the current state being backtracked, find its most probable
    previous state as argmax
    current_layer -= 1

predicted_sequence = reverse_path[::-1][1:len(reverse_path)-1]

return predicted_sequence

```

We then define the method `viterbi()` to iteratively run the Viterbi algorithm for all input sequences in the training date (from `input_sequences` in `__format_testdata()`), and return a nested list of predicted state sequence for all sentences within the input test data dev-in.

```

pred_state_sequences = [[]]
i = 0
for input_sequence in input_sequences:
    for state in self.__mini_viterbi(input_sequence, emission_dict,
transition_dict):
        pred_state_sequences[i].append(state)
    pred_state_sequences.append([]) # to store state sequence for next sentence
    i += 1
pred_state_sequences.pop() # remove last []
return pred_state_sequences

```

Results

Language	Entity	Sentiment
English	Precision: 0.8318 Recall: 0.8389 F: 0.8354	Precision: 0.7987 Recall: 0.8055 F: 0.8021
Chinese		
Singapore	Precision: 0.6232 Recall: 0.5092 F: 0.5605	Precision: 0.5356 Recall: 0.4376 F: 0.4816

Full results can be found in `part_3_results.txt`

Part 4: Top 3 Sequences

In order to get the 3rd best sequence, there is a need to modify the original Viterbi algorithm. In particular, for the equation:

$$\pi(j+1, u) = \max_v \{ \pi(j, v) \times b_u(x_{j+1}) \times a_{v,u} \}$$

There is a need to store the top 3 probabilities in $\pi(i, j)$ instead of just the highest one. Here, a 2D array fails to meet our requirements - we somehow need to maintain a sequence of probabilities and the previous state that led to it - and so in `part4.py` we use an object of our own implementation.

The class `queue` is essentially some modified priority queue concept. It maintains a series of dictionaries in an array, each of these dictionaries containing a "p" and "previous" keys to represent the probability and the previous state associated with it respectively. This array is sorted in descending order according to the value associated with the "p" key.

The method `try_add(prob, state)` is the primary means of updating the queue. It first checks if `prob` is at least higher than the lowest probability currently in the queue. If it is, it deletes the dictionary associated with the lowest probability and adds the incoming `(prob, state)` as a dictionary. Upon which, the array holding the dictionaries is sorted. The implementation is as follows:

```
class queue:
    ...
    def try_add(self, prob, state):
        if prob > self.min_prob or len(self.queue) < self.k:
            if len(self.queue) == self.k:
                del self.queue[-1]
            self.queue.append({"p": prob, "previous": state})
            self.queue = sorted(self.queue, key=lambda x: x["p"], reverse=True)
        ...
```

The full implementation can be found in `part4.py`

Viterbi then proceeds as per normal. After the full π dictionary is built however, the back propagation sequence then extracts the state associated with the third best probability (instead of the first). The results of this implementation are as follows:

Results

Language	Entity	Sentiment
English	Precision: 0.2904 Recall: 0.4209 F: 0.3437	Precision: 0.0682 Recall: 0.0989 F: 0.0807

Full results can be found in `part_4_results.txt`

Part 5: Design Challenge