50.007 Machine Learning

Fall 2020

Design Project

Report

Implementation Approach & Results

1003516 Lee Jia Le

1003331 Wong Tin Kit

1003835 Yeo Ying Xuan

Part 2

Emission parameter estimation using maximum likelihood estimation (MLE)

Looking at the training data provided, each line in the file consists of a token (ie. word), followed by a space, then finally its tag. A single empty line serves to separate sentences in the training data.

To learn the emission parameter from training data for first-order Hidden Markov Model (HMM), first, we wrote a function `count_emission` to tabularise the number of occurrences of each word emitted by a specific state.

For each line in the file containing a word and its tag, we store the number of occurrences a specific observation (ie. word) in a dictionary as a key-value pair, with key as the word, and value as the number of occurrences. This forms the a dictionary for a specific state (ie. tag).

```
{key = observation: value = count}
```

We then store the dictionary of each state as the value of another dictionary, where the key represents the tag. As a result, `count_emission` function returns `emission_tracker`, which is nested dictionary in this format:

```
{key = state:
      {key = observation: value = count}
}
```

This would hence allow us to easily retrieve values required to calculate the emission parameters, using the function `emission_para`:

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

Based on the dictionary obtained that tabularises the emission counts above, we can easily obtain $Count(y \to x)$ using `emission_tracker[y][x]`, and also obtaining $Count(y)$ using `sum(emission_tracker[y].values())`.

Emission parameter estimation with special token

In a modified version of computing emission parameters, the function `emission_para_token` takes into account words that appear in the test set, but do not appear in the training set. Similarly, this function returns the emission parameters.

$$e(x|y) = \begin{cases} \frac{\text{Count}(y \to x)}{\text{Count}(y)+k} & \text{If the word token } x \text{ appears in the training set} \\ \frac{k}{\text{Count}(y)+k} & \text{If word token } x \text{ is the special token } \texttt{\#UNK\#} \end{cases}$$

First, we do a conditional `if-else` check on the observation - whether the observation exists in the training data. If the observation did not appear in the training data, a special token, `#UNK#`, is assigned to the observation. The remaining steps remain the same as MLE estimation of emission parameter without special token above. With $k = 0.5$, we can easily obtain emission parameters.

Simple system that produces tag y

A function `tag_producer` was implemented to take in a sentence as a list object, with each word being a `string` type element of the list, and return a list of predicted tags y for the sentence.

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

For each word in a sentence, we loop through all the states that the word can take and calculate the respective emission probabilities. We then return the state that gives the highest emission probability as the predicted tag. The individual predicted tags for each word are then stored in a list containing all predicted tags of a sentence, which is returned by the function.

Results

Using the evaluation script given, our results are as follow:

```
EN

#Entity in gold data: 13179
#Entity in prediction: 18650

#Correct Entity : 9542
Entity  precision: 0.5116
Entity  recall: 0.7240
Entity  F: 0.5996

#Correct Sentiment : 8456
Sentiment  precision: 0.4534
Sentiment  recall: 0.6416
Sentiment  F: 0.5313
====
CN

#Entity in gold data: 700
#Entity in prediction: 4248

#Correct Entity : 345
Entity  precision: 0.0812
Entity  recall: 0.4929
Entity  F: 0.1395

#Correct Sentiment : 167
Sentiment  precision: 0.0393
Sentiment  recall: 0.2386
Sentiment  F: 0.0675
====
SG

#Entity in gold data: 4301
#Entity in prediction: 12237

#Correct Entity : 2386
Entity  precision: 0.1950
Entity  recall: 0.5548
Entity  F: 0.2885

#Correct Sentiment : 1531
Sentiment  precision: 0.1251
Sentiment  recall: 0.3560
Sentiment  F: 0.1851
====
```

Part 3

Transition parameter estimation using maximum likelihood estimation (MLE)

To learn the transition parameter from training data for first-order Hidden Markov Model (HMM), first, we wrote a function `count_transition` to tabularise the number of occurrences of transiting from a particulate state to another particular state. Here, we define our transition from $state\ u$ to $state\ v$.

For each line in the file containing a word and its tag, we store the number of occurrences of transitioning from one state to another in a nested dictionary format - the key represents $state\ u$, and the value is a dictionary where key represents $state\ v$ and the value is the number of occurrences.

As a result, `count_transition` function returns `transition_tracker`, which is a nested dictionary in this format:

```
{key = state_u:
      {key = state_v: value = count}
}
```

This would hence allow us to easily retrieve values required to calculate the transition parameters, using the function `transition_para`:

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

Based on the dictionary obtained that tabularises the transition counts above, we can easily obtain $Count(y_{i-1}, y_i)$ using `transition_tracker[state_u][state_v]`, and also obtaining $Count(y_{i-1})$ using `sum(transition_tracker[state_u].values())`.

Viterbi Algorithm

Using the functions we have implemented in Parts 2 and 3, we implement the viterbi algorithm to return the best path.

$a_{u,v} = q(y_v|y_u)$
$b_v(x_j) = e(x_j|y_j)$

Keeping in mind that the natural logarithm function is strictly increasing, all values are calculated in natural logarithmic form in our implementation, replacing multiplication by addition. This is based on the logarithmic property $ln(ab) = ln(a) + ln(b)$. The purpose of taking the natural logarithm of probabilities is to reduce the risk of potential numerical underflow. However, one new issue with natural logarithmic calculations is that $ln\,0 = -\infty$. To deal with probability values that are equal to 0, we simply set $ln\,0 = C$, where C is a large negative number.

In addition, in view of the fact that the training data is split into sentences as indicated by a single empty line, the base case of $\pi(0, v) = 1\ if\ v = START,\ 0\ otherwise$ is already implicitly accounted for. Furthermore, $ln\,1 = 0$. Hence, our implementation's base case is to determine the score at the position of the first word in a sentence, which is not dependent on the $START$ state position.

The score of the best path at each position/time step are stored in a dictionary in such format:

```
{key = position:
      {key = state_v: value = (state_u, score)}
}
```

We store the previous state, $state\ u$, as a tuple with the highest score of the path from $START$ to $state\ v$ is for the purpose of backtracking.

At every position/time-step, we calculate the transition and emission probabilities, then take their natural logarithmic values as the score that we are tracking on. Prior to calculating the emission probabilities, we check whether the observation has appeared in the training set or not. If the observation was not seen in the training set, we assign the special token `#UNK#`.

Moving forward recursively:

We begin by initialising the first word of a sentence, ie. calculating $\pi(1, v)$:

$$\begin{aligned}\pi(1, v) &= ln[a_{START,v} \bullet b_v(x_1)] \\ &= ln(a_{START,v}) + ln([b_v(x_1)] \\ &= ln(1) + ln[b_v(x_1)] \\ &= ln[b_v(x_1)]\end{aligned}$$

From the second word to the last word of a sentence, we calculate all the possible paths and keep track of the state that gives the maximum score:

$$\pi(k, v) = max_v\,\{\pi(k-1, u) + ln(a_{u,v}) + ln[b_v(x_k)]\}$$

In the final step, we calculate the score of the overall path from $START$ state to $STOP$ state.

ie. calculate $\pi(k+1, STOP) = max_v\{\pi(k,u) + ln(a_{u,STOP})\}$.

After calculating the best path from $START$ state to $STOP$ state, ie. the best path for the entire sentence, we then do backtracking to retrieve the states (ie. tags) at each position/time step. During the forward recursive process, we stored the state of position $k-1$ that gave the maximum score as a tuple along with the score at position k, hence we now retrieve all the states from each tuple to produce the full tag sequence for a sentence in the backtracking process.

Here are our results from our viterbi algorithm:

```
EN

#Entity in gold data: 13179
#Entity in prediction: 13375

#Correct Entity : 10791
Entity  precision: 0.8068
Entity  recall: 0.8188
Entity  F: 0.8128

#Correct Sentiment : 10268
Sentiment  precision: 0.7677
Sentiment  recall: 0.7791
Sentiment  F: 0.7734
====
CN

#Entity in gold data: 700
#Entity in prediction: 850

#Correct Entity : 208
Entity  precision: 0.2447
Entity  recall: 0.2971
Entity  F: 0.2684

#Correct Sentiment : 122
Sentiment  precision: 0.1435
Sentiment  recall: 0.1743
Sentiment  F: 0.1574
====
SG

#Entity in gold data: 4301
#Entity in prediction: 4274

#Correct Entity : 2212
Entity  precision: 0.5175
Entity  recall: 0.5143
Entity  F: 0.5159

#Correct Sentiment : 1821
Sentiment  precision: 0.4261
Sentiment  recall: 0.4234
Sentiment  F: 0.4247
====
```

Part 4

Modified viterbi algorithm to obtain the third best output sequence

To obtain the third best output sequence, we find the top three best paths and store these values at each position/time-step. For node i at position k, we obtain all the possible paths from all nodes in position $k-1$ and calculate the scores of these paths. We then keep track of the scores of the best three paths, along with the path from $START$ up till position k, passing through the node that gives one of the best three scores at position $k-1$. This process is repeated for all nodes at position k to obtain the best three scores at each node. One numerical example would be - given position k has j nodes, we will keep track of $k \bullet j$ number of best paths and scores.

Similar to the Viterbi algorithm to get the best output sequence that was implemented in part 3, the base case of $\pi(0,\ v)\ =\ 1\ if\ v\ =\ START,\ 0\ otherwise$ is already implicitly accounted for, as $ln\ 1 = 0$. Hence, our implementation's base case is to determine the score at the position of the first word in a sentence, which is not dependent on the $START$ state position.

At each position/time step, the scores of the best three paths for each node are stored. All the scores across all positions are stored in the `scores` dictionary in such format:

```
scores =
      {key = position:
            {key = state_v: value = (state_u1, score),
                                    (state_u2, score),
                                    (state_u3, score)
            }
      }
```

We store the previous state, $state\ u$, as a tuple with the highest score of the path from $START$ to $state\ v$ is for the purpose of backtracking.

At every position/time-step, we calculate the transition and emission probabilities, then take their natural logarithmic values as the score that we are tracking on. Prior to calculating the emission probabilities, we check whether the observation has appeared in the training set or not. If the observation was not seen in the training set, we assign the special token `#UNK#`.

The forward process to calculate and determine the best three paths described above is split into three portions:

First, we initialise the first word of a sentence, ie. calculating $\pi(1, v)$. There is no dependent score from the previous position, which is the $START$ state, because of $ln(1) = 0$.

$$\begin{aligned}\pi(1, v) &= ln[a_{START,v} \bullet b_v(x_1)] \\ &= ln(a_{START,v}) + ln([b_v(x_1)] \\ &= ln(1) + ln[b_v(x_1)] \\ &= ln[b_v(x_1)]\end{aligned}$$

From the second word to the last word of a sentence, we calculate all possible paths and keep track of the three states that give the top three best paths.

ie. for each node v at position k, we calculate all possible values of $\pi(k, v)$.

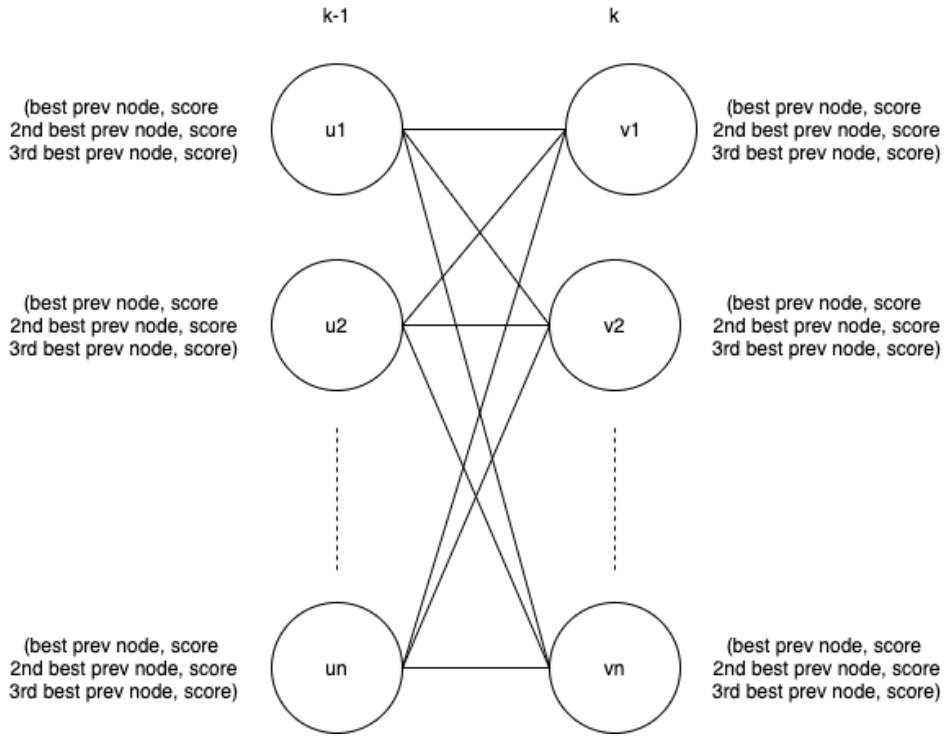
The calculation of $\pi(k, v)$ is based on $\pi(k-1, u) + ln(a_{u,v}) + ln[b_v(x_k)]\}$, where node u is all the possible states in position $k-1$. We then obtain the top three best scores for each node v. We store these three scores and its corresponding preceding node u as a tuple in the `scores` dictionary as described above.

In the final step, we calculate the score of the best 3 paths for each node at position n, where n is the length of a sentence, to the $STOP$ state.

ie. calculate $\pi(n+1,\ STOP)$ based on $\pi(n,\ u)\ +\ ln(a_{u,STOP})$ for the best three paths across all node u at position n.

After obtaining all the scores, we then filtered out the best three paths. The scores and its corresponding preceding node are then stored in a tuple.

In the backtracking process, we hold N best path sequences in the list $N\_bestPaths$ in decreasing order:
$N\_bestPaths\ =\ [[1st\ best\ path\ sequence], [2nd\ best\ path\ sequence], ..., [N\ best\ path\ sequence]]$.
During the forward algorithm, in each node at position k, we kept track of the 3 best previous nodes (at position k-1) with their respective scores as illustrated in the figure below.



For each path, we need to keep track of the corresponding previous node's 3 best paths. Therefore at each current node at position k, we consult the scores dictionary at the current position k with the key of the best previous node (at position $k-1$) for each N best paths. Now, we have the best previous node to the current node and we add it to the corresponding path in the list $N\_bestPaths$. We iterate through this process for all positions until position=1.

We have to modify the backtracking algorithm when picking the best path from position k=1 as nodes at this position only have the a singular score (ie. there is only 1 way to traverse from $START$ state to nodes at this position).

With $N\_bestPaths$, we can return the Nth best path.

This is our result for the third best output sequence, based on our modified viterbi algorithm:

```
EN

#Entity in gold data: 13179
#Entity in prediction: 13556

#Correct Entity : 10788
Entity  precision: 0.7958
Entity  recall: 0.8186
Entity  F: 0.8070

#Correct Sentiment : 10100
Sentiment  precision: 0.7451
Sentiment  recall: 0.7664
Sentiment  F: 0.7556
```

We ran the Nviterbi algorithm with N=1 and we got the same result as in Part 3. This proves that the algorithm is consistent regardless of what N is provided, and will work for any specified N.

Part 5

Second-Order Hidden Markov Model

The second-order hidden Markov model works in a similar fashion compared to the first order HMM. The only difference now is that we consider second-order dependencies, in a trigram format - take into account second-order dependencies where the prediction of a tag is dependent on not just one, but two preceding tags. Here, we define the order of states that we calculate at each position/time step to be $state\ u\ \rightarrow state\ v\ \rightarrow state\ w$.

Mathematical Formulation

$$a_{u,v,w} = q(w|u,v) = \frac{Count(u,v,w)}{Count(u,v)}$$
$$b_w(x_j) = e(x|y) = \frac{Count(w \rightarrow x_j)}{Count(w)}$$

Base case:

$$\pi(0,\ v,\ w) = 1\ if\ v,w = START,\ 0\ otherwise$$

Moving forward recursively:

For any $k \in \{1, \ldots, n\}$, for any $v,\ w \in \tau$,

$$\pi(k,v,w) = max_{v \in \tau}\{\pi(k-1),u,v) \cdot a_{u,v,w} \cdot b_w(x_k)\}$$

Transition from y_n to STOP:

$$max_{y_1, \ldots, y_n}\, p(x_1, \ldots, x_n, y_0 = START,\ y_1, \ldots y_n, y_{n+1} = STOP)$$
$$= max_{v,w \in \tau}\{\pi(n,v,w) \cdot a_{v,w,STOP}\}$$

Backtracking

$$y_{n-1}^*,\ y_n^* = argmax_{v,w}\{\pi(n,v,w) \cdot a_{v,w,STOP}\}$$

For $k = n-2$ to $k = 1$:

$$y_k^* = argmax_u\{\pi(k+1,\ u,\ y_{k+1}^*) \cdot a_{u,\ y_{k+1}^*, y_{k+2}^*} \cdot b_{y_{k+2}^*}(x_{k+2})\}$$
$$= argmax_u\{\pi(k+1,\ u,\ y_{k+1}^*) \cdot a_{u, y_{k+1}^*, y_{k+2}^*}\}$$

Code Implementation

Similarly, we also assign the special token `#UNK#`, to words that did not appear in the training set, but yet appearing in the test set. This smoothing parameter helps to optimise our predictions further, especially for unseen words.

We implemented two new functions, `HMM2_count_transition` and `HMM2_transition_para` to tabularise transition counts and to calculate the transition probabilities respectively.

`HMM2_count_transition` function returns `transition_tracker`, which is a nested dictionary in this format:

```
{key = state_u:
      {key = state_v:
            {key = state_w: value = count_u_v_w}
      }
}
```

This would hence allow us to easily retrieve numerator and denominator values required to calculate the transition parameters, using the function `HMM2_transition_para`.

Viterbi algorithm for second-order HMM

Keeping in mind that the natural logarithm function is strictly increasing, all values are calculated in natural logarithmic form in our implementation, replacing multiplication by addition. This is based on the logarithmic property $ln(ab) = ln(a) + ln(b)$. The purpose of taking the natural logarithm of probabilities is to reduce the risk of potential numerical underflow. However, one new issue with natural logarithmic calculations is that $ln\,0 = -\infty$. To deal with probability values that are equal to 0, we simply set $ln\,0 = C$, where C is a large negative number.

Similar to the viterbi-related algorithms implemented in Parts 3 and 4 above, the base case of $\pi(0, START, START) = 1\ if\ v, w = START, 0\ otherwise$ is already accounted for, due to the structure of the datasets and that $ln\,1 = 0$. Hence, our implementation's base case is to determine the score at $\pi(1, START, w)$, ie. the score at the position of the first word in a sentence.

At each position/time step, the score of the best path are stored in a dictionary, `scores`, in such format:

```
{key = state_w:
      {key = state_u:
            {key = state_v: score}
}
```

We store the highest score of previous 2 states, *state u and state v*, as the value of the nested dictionary from $START$ to *state w* is for the purpose of backtracking.

At every position/time-step, we calculate the transition and emission probabilities, then take their natural logarithmic values as the score that we are tracking on. Prior to calculating the emission probabilities, we check whether the observation has appeared in the training set or not. If the observation was not seen in the training set, we assign the special token `#UNK#`.

Moving forward recursively:

1. We begin by initialising the first word of a sentence, ie. calculating $\pi(1,\ v,\ w)$:

$$\begin{aligned}\pi(1,v,w) &= ln[a_{START,\,START,\,w} \bullet b_w(x_1)] \\ &= ln(a_{START,\,START,\,w}) + ln([b_w(x_1)] \\ &= ln(1) + ln[b_w(x_1)] \\ &= ln[b_w(x_1)]\end{aligned}$$

2. Case: 1 Word Sentence and 2 Word Sentences
   a. In both cases, we calculate the possible paths and keep track of the state that yields the maximum score. It is important to look up the `scores` dictionary for the correct previous states $\pi(k-1,u,v)$.

$$\pi(k,v,w) = max_v\{\pi(k-1,u,v) + ln(a_{u,v,w}) + ln[b_w(x_k)]\}$$

   An example for 1 word sentences is as follows:

$$\pi(0,v,w) = max_v\{\pi(-1,START,START) + ln(a_{START,START,START})\}$$
$$\pi(1,v,w) = max_v\{\pi(1,START,START) + ln(a_{START,START,w}) + ln[b_w(x_1)]\}$$
$$\pi(2,v,w) = max_v\{\pi(1,START,v) + ln(a_{START,v,w})\}$$

3. Case: >2 Word Sentences
   a. From the second word to the last word of a sentence, we calculate all the possible paths and keep track of the state that gives the maximum score:

$$\pi(k,v,w) = max_v\{\pi(k-1,u,v) + ln(a_{u,v,w}) + ln[b_w(x_k)]\}$$

4. In the final step, we calculate the score of the overall path from $START$ state to $STOP$ state.
   ie. calculate $\pi(k+1,w,STOP) = max_v\{\pi(k,v,w) + ln(a_{v,w,STOP})\}$.

After calculating the best path from $START$ state to $STOP$ state, ie. the best path for the entire sentence, we then do backtracking to retrieve the states (ie. tags) at each position/time step. During the forward recursive process, we stored the state of position $k-1$ that gave the maximum score as a tuple along with the score at position k, hence we now retrieve all the states from each tuple to produce the full tag sequence for a sentence in the backtracking process.

This is our result for the second order hidden markov model, based on our modified viterbi algorithm:

```
EN

#Entity in gold data: 13179
#Entity in prediction: 13374

#Correct Entity : 10835
Entity  precision: 0.8102
Entity  recall: 0.8221
Entity  F: 0.8161

#Correct Sentiment : 10416
Sentiment  precision: 0.7788
Sentiment  recall: 0.7903
Sentiment  F: 0.7845
```

Tuning k value (Optional)

We tune the value of k to select the ideal balance between replacing unseen words with the `#UNK#` token and reducing the amount of vocabulary that our HMM model can capture in its parameters. In this case, we simply select the value of k that produces the best results on the development set.

We tested a range of k values from 2 to 9 and found that the higher the k, the better the **Part 3 Viterbi Algorithm** model performed in terms of F scores.

```
EN

#Entity in gold data: 13179
#Entity in prediction: 13344

#Correct Entity : 10813
Entity  precision: 0.8103
Entity  recall: 0.8205
Entity  F: 0.8154

#Correct Sentiment : 10290
Sentiment  precision: 0.7711
Sentiment  recall: 0.7808
Sentiment  F: 0.7759
====
```

k = 2

```
EN

#Entity in gold data: 13179
#Entity in prediction: 13341

#Correct Entity : 10814
Entity  precision: 0.8106
Entity  recall: 0.8205
Entity  F: 0.8155

#Correct Sentiment : 10292
Sentiment  precision: 0.7715
Sentiment  recall: 0.7809
Sentiment  F: 0.7762
====
```

k = 3

```
EN

#Entity in gold data: 13179
#Entity in prediction: 13341

#Correct Entity : 10816
Entity  precision: 0.8107
Entity  recall: 0.8207
Entity  F: 0.8157

#Correct Sentiment : 10294
Sentiment  precision: 0.7716
Sentiment  recall: 0.7811
Sentiment  F: 0.7763
====
```

k = 4

```
EN

#Entity in gold data: 13179
#Entity in prediction: 13338

#Correct Entity : 10817
Entity  precision: 0.8110
Entity  recall: 0.8208
Entity  F: 0.8159

#Correct Sentiment : 10294
Sentiment  precision: 0.7718
Sentiment  recall: 0.7811
Sentiment  F: 0.7764
====
```

k = 5

```
EN

#Entity in gold data: 13179
#Entity in prediction: 13329

#Correct Entity : 10827
Entity  precision: 0.8123
Entity  recall: 0.8215
Entity  F: 0.8169

#Correct Sentiment : 10304
Sentiment  precision: 0.7731
Sentiment  recall: 0.7818
Sentiment  F: 0.7774
====
```

k = 6

```
EN

#Entity in gold data: 13179
#Entity in prediction: 13330

#Correct Entity : 10830
Entity  precision: 0.8125
Entity  recall: 0.8218
Entity  F: 0.8171

#Correct Sentiment : 10307
Sentiment  precision: 0.7732
Sentiment  recall: 0.7821
Sentiment  F: 0.7776
====
```

k = 7

```
EN

#Entity in gold data: 13179
#Entity in prediction: 13327

#Correct Entity : 10833
Entity  precision: 0.8129
Entity  recall: 0.8220
Entity  F: 0.8174

#Correct Sentiment : 10311
Sentiment  precision: 0.7737
Sentiment  recall: 0.7824
Sentiment  F: 0.7780
====
```

k = 8

```
EN

#Entity in gold data: 13179
#Entity in prediction: 13326

#Correct Entity : 10834
Entity  precision: 0.8130
Entity  recall: 0.8221
Entity  F: 0.8175

#Correct Sentiment : 10312
Sentiment  precision: 0.7738
Sentiment  recall: 0.7825
Sentiment  F: 0.7781
====
```

k = 9

We then tested a few k values from range of 2 to 9 and found that the higher the k, the better the **Part 5 Second Order HMM** model seems to perform in terms of F scores.

```
EN

#Entity in gold data: 13179
#Entity in prediction: 13376

#Correct Entity : 10835
Entity  precision: 0.8100
Entity  recall: 0.8221
Entity  F: 0.8160

#Correct Sentiment : 10414
Sentiment  precision: 0.7786
Sentiment  recall: 0.7902
Sentiment  F: 0.7843
====
```

k = 2

```
EN

#Entity in gold data: 13179
#Entity in prediction: 13377

#Correct Entity : 10836
Entity  precision: 0.8100
Entity  recall: 0.8222
Entity  F: 0.8161

#Correct Sentiment : 10420
Sentiment  precision: 0.7789
Sentiment  recall: 0.7907
Sentiment  F: 0.7848
====
```

k = 6

```
EN

#Entity in gold data: 13179
#Entity in prediction: 13377

#Correct Entity : 10836
Entity  precision: 0.8100
Entity  recall: 0.8222
Entity  F: 0.8161

#Correct Sentiment : 10420
Sentiment  precision: 0.7789
Sentiment  recall: 0.7907
Sentiment  F: 0.7848
====
```

k = 7

```
EN

#Entity in gold data: 13179
#Entity in prediction: 13381

#Correct Entity : 10840
Entity  precision: 0.8101
Entity  recall: 0.8225
Entity  F: 0.8163

#Correct Sentiment : 10422
Sentiment  precision: 0.7789
Sentiment  recall: 0.7908
Sentiment  F: 0.7848
====
```

k = 8

```
EN

#Entity in gold data: 13179
#Entity in prediction: 13381

#Correct Entity : 10840
Entity  precision: 0.8101
Entity  recall: 0.8225
Entity  F: 0.8163

#Correct Sentiment : 10422
Sentiment  precision: 0.7789
Sentiment  recall: 0.7908
Sentiment  F: 0.7848
====
```

k = 9

To prevent overfitting of the model, k= 8 will be preferred instead of k=9. In addition, this model is for reference and its output is not considered for the evaluation on the test sets provided in the zip file.