Hidden Markov Models and Perceptrons for Natural Language Processing

01.112 Machine Learning

Aravind S. Kandiah [1], Reuben R. W. Wang [1]
Singapore University of Technology and Design

Contents

[1]

[1]Both authors contributed equally to this work.

§1 Part 4 (Second-Order Hidden Markov Model)

In this section, we will present the adapted algorithm for use in the second-order hidden Markov model, and give an explanation on the modifications.

§1.1 Model Modifications

In the first order hidden Markov model, we had that the joint probability of a given sentence of words and labels with the Markov assumption is given as:

$$\mathbb{P}(x,y) \approx \left[\prod_{j=1}^{n+1} \mathbb{P}(y_j|y_{j-1})\right] \cdot \left[\prod_{j=1}^{n} \mathbb{P}(x_j|y_j)\right] \approx \left[\prod_{j=1}^{n+1} a_{y_{j-1},y_j}\right] \cdot \left[\prod_{j=1}^{n} b_{y_j(x_j)}\right] \tag{1}$$

Extending this to a (strictly speaking) non-Markovian case where we instead 'remember' 2 of the previous states, our joint probability would now be:

$$\mathbb{P}(x,y) \approx \left[\prod_{j=2}^{n+2} \mathbb{P}(y_j|y_{j-1},y_{j-2})\right] \cdot \left[\prod_{j=1}^{n} \mathbb{P}(x_j|y_j)\right] \approx \left[\prod_{j=2}^{n+2} a_{(y_{j-1},y_{j-2}),y_j}\right] \cdot \left[\prod_{j=1}^{n} b_{y_j(x_j)}\right] \tag{2}$$

In order to account for 2 previous states in practice, what we did was to add 2 additional start and stop labels which we called:

$$\{\text{“}start0\text{”}, \text{“}start1\text{”}, \text{“}stop0\text{”}, \text{“}stop1\text{”}\} \tag{3}$$

As such, the visualization that we adopt would be modified to the following representation:

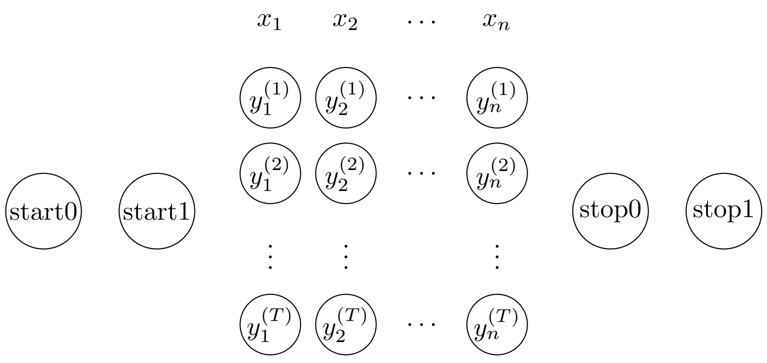


Figure 1: 2nd-Order HMM Visualization

Furthermore, we notice that the transition probabilities will now have to take into consideration transitions from both the $j-1$ **and** $j-2$ word layers. Because of this, finding the score for some tag u in word layer j will require taking the max over both v_1 and v_0. So, we can write our score update equation for iteration as:

$$\pi(j,u) = \max_{v_0,v_1} \left\{ [\pi(j-1,v_1) \cdot a_{(v_0,v_1),u} \cdot b_{x_j(u)} \right\} \tag{4}$$

Since there are several terms in the max over v_0 and v_1 that do not depend on v_0, we can move the max over v_0 into the sum over v_1. With this, our equation becomes:

$$\boxed{\pi(j,u) = \max_{v_1} \left\{ \pi(j-1,v_1) \cdot b_{x_j(u)} \cdot \max_{v_0} \left\{ a_{(v_0,v_1),u} \right\} \right\}} \tag{5}$$

The 2 equations above (4 and 5) are mathematically equivalent but correspond to 2 different means of implementation when utilized in the Viterbi algorithm. For clarity, our second-order HMM Viterbi algorithm (using equation 5) is as follows:

Second-Order HMM Viterbi algorithm:

1. We first initialize the start score with a base case:

$$\begin{aligned}\pi(0, \text{start0}) &= 1 \\ \pi(1, \text{start1}) &= 1\end{aligned} \tag{6}$$

2. At the j-th word (x_j), we assign scores to each possible label u using the following equation:

$$\pi(j, u) = \max_{v_0, v_1} \left\{ \pi(j-1, v_1) \cdot a_{(v_0, v_1), u} \cdot b_{x_{j-1}(u)} \right\} \tag{7}$$

where v_0 and v_1 are the possible labels for the $j-1$-th and $j-2$-th words.

3. We repeat step 2 for $j = 2, ..., n+2$, where the final cases at the '*stop0*' ($j = n+2$) and '*stop1*' layers are:

$$\begin{aligned}\pi(n+1, stop0) &= \max_{v_0, v_1} \left\{ \pi(n, v_1) \cdot a_{(v_0, v_1), stop0} \right\} \\ \pi(n+2, stop1) &= \max_{v_0} \left\{ \pi(n+1, stop0) \cdot a_{(v_0, stop0), stop1} \right\}\end{aligned} \tag{8}$$

4. Finally, we recover the optimal path by backtracking through the word layers and looking for the argmax values for each *pi* scoring (or alternatively, we store the argmax simultaneous to finding the max score in step 2).

§1.2 Results of Implementation

Here, we present the numerical results of our implementations of part 2, 3 and 4 for comparison.

PART 2

Entities in gold data : 802 **Entities in prediction**: 1044			
Entities		**Entity Type**	
Correct	577	**Correct**	448
Precision	0.5527	**Precision**	0.4291
Recall	0.7195	**Recall**	0.5586
F score	0.6251	**F score**	0.4854

Table 1: Part 2 Results (EN)

Entities in gold data : 238 **Entities in prediction**: 1114			
Entities		**Entity Type**	
Correct	186	**Correct**	79
Precision	0.1670	**Precision**	0.0709
Recall	0.7815	**Recall**	0.3319
F score	0.2751	**F score**	0.1169

Table 2: Part 2 Results (FR)

PART 3

Entities in gold data : 802 **Entities in prediction**: 839			
Entities		**Entity Type**	
Correct	513	**Correct**	421
Precision	0.6114	**Precision**	0.5018
Recall	0.6397	**Recall**	0.5249
F score	0.6252	**F score**	0.5131

Table 3: Part 3 Results (EN)

Entities in gold data : 238 **Entities in prediction**: 451			
Entities		**Entity Type**	
Correct	134	**Correct**	71
Precision	0.2971	**Precision**	0.1574
Recall	0.5630	**Recall**	0.2983
F score	0.3890	**F score**	0.2061

Table 4: Part 3 Results (FR)

PART 4

Entities in gold data : 802 **Entities in prediction**: 859			
Entities		**Entity Type**	
Correct	489	**Correct**	371
Precision	0.5693	**Precision**	0.4319
Recall	0.6097	**Recall**	0.4626
F score	0.5888	**F score**	0.4467

Table 5: Part 4 Results (EN)

Entities in gold data : 238 **Entities in prediction**: 507			
Entities		**Entity Type**	
Correct	164	**Correct**	61
Precision	0.3235	**Precision**	0.1203
Recall	0.6891	**Recall**	0.2563
F score	0.4403	**F score**	0.1638

Table 6: Part 4 Results (FR)

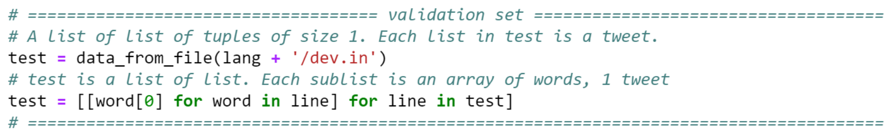
Since we are only tasked to run our second-order hidden Markov model on the EN and FR training sets, we will only show the results for those 2 languages here. The results for part 2 and 3 for the other languages are put in appendix A.1 and A.2.

§1.3 README (Part 4)

In this section, we will give a description on how to run our code for part 4 (*part4.py*). We have separated our code into 2 files, namely *part4_fun.py* and *part4.py*. These files are:

- *part4_fun.py*: This *.py* file contains the functions written for and utilized in *part4.py*. This file is imported in *part4.py*.
- *part4.py*: This *.py* contains the 'main' code block used to run the transition and emission parameter generation, and second-order Viterbi algorithm to predict the tags for the test data.

To change the test data, you can simply edit the code block entitled 'validation set' (2) and run the code, which will also print the results/scores.



```
# ================================== validation set ==================================
# A list of list of tuples of size 1. Each list in test is a tweet.
test = data_from_file(lang + '/dev.in')
# test is a list of list. Each sublist is an array of words, 1 tweet
test = [[word[0] for word in line] for line in test]
# ====================================================================================
```

Figure 2: Validation Set Code

part4.py can also be run on the python notebook entitled *test.ipynb*.

§2 Part 5 (Design Challenge)

For the design challenge, we have adopted and implemented the use of 2 different methods of improvement to the hidden Markov model. The first addresses the sparsity of the transition parameters when utilizing a second-order Hidden Markov assumption, whereas the second utilizes a perceptron to train the transition and emission parameters for a first-order Viterbi implementation to tagging.

§2.1 Second-Order HMM Improvements

We looked into 3 methods to improve our second-order HMM model performance. These were namely *shrinkage*, *Laplace smoothing* and *hyperparameter* tuning. The implementations and results of these are presented below.

2.1.1 Shrinkage

In order to compensate for transition parameter sparsity, we propose the use of not just the trigrams for the transition matrix entries, but also the bigrams and unigrams. This is more commonly known as *shrinkage*. What this essentially does is incorporate more count information from the same training data set so that we reduce trivial entries. For clarity, we define the n-grams as follows:

n-gram	Implementation
Unigram $= \mathbb{P}(u)$	$\mathbb{P}(u) = \frac{1}{\text{count}(u)}$
Bigram $= \mathbb{P}(u\|v_1)$	$\mathbb{P}(u\|v_1) = \frac{\text{count}(v_1,u)}{\text{count}(v_1)}$
Trigram $= \mathbb{P}(u\|v_0, v_1)$	$\mathbb{P}(u\|v_0, v_1) = \frac{\text{count}(v_0,v_1,u)}{\text{count}(v_0,v_1)}$

Table 7: n-Grams

With this, we then redefine our transmission parameters as follows:

$$a_{(v_0,v_1),u} = \lambda_1\mathbb{P}(u|v_0, v_1) + \lambda_2\mathbb{P}(u|v_1) + \lambda_3\mathbb{P}(u) \tag{9}$$

where λ_j are the associated weights assigned to each n-gram. Because these weights are unique to the transmission parameters for every set of $\{v_0, v_1, u\}$, they constitute new model parameters and must also be trained with the training data. To do this, we have employed the *deleted interpolation algorithm*. The deleted interpolation algorithm also utilizes count information to estimate the optimal parameters for use in this modification. The closed forms are as follows:

$$\begin{aligned} \lambda_1 &= k_3 \\ \lambda_2 &= (1 - k_3) \cdot k_2 \\ \lambda_3 &= (1 - k_3) \cdot (1 - k_2) \end{aligned} \tag{10}$$

where the values of the k_j terms are defined by:

$$\begin{aligned} k_2 &= \frac{\log(\text{count}(v_1, u) + 1) + 1}{\log(\text{count}(v_1, u) + 1) + 2} \\ k_3 &= \frac{\log(\text{count}(v_0, v_1, u) + 1) + 1}{\log(\text{count}(v_0, v_1, u) + 1) + 2} \end{aligned} \tag{11}$$

These improvements were based on an academic reference by Jungyeul Park, Mouna Chebbah, Siwar Jendoubi and Arnaud Martin [1]. With our implementation of shrinkage, there was a noticeable increase in the F-scores for both entity and entity type predictions in both languages (EN and FR). The raw scores are given in appendix B.1 while just the entity type F-score comparisons are given in table 8 below:

	w/o Shrinkage	w/ Shrinkage	% Change
EN	0.4467	0.4547	1.7909
FR	0.1638	0.1632	-0.3676

Table 8: F-Score Comparisons

where the percentage change is computed as follows:

$$\begin{aligned} F_s &= \text{F-score w/ shrinkage} \\ F_{ns} &= \text{F-score w/o shrinkage} \\ \%\ \text{Change} &= \frac{F_s - F_{ns}}{F_{ns}} \times 100\% \end{aligned} \tag{12}$$

2.1.2 Laplace Smoothing

In an attempt to further improve these results, we also attempted to implement Laplace smoothing [2] for unseen words in the training data instead of the smoothing suggested in the brief. Laplace smoothing is done as follows:

$$\mathbb{P}(x|y) = \frac{\text{count}(y \to x) + 1}{\text{count}(y) + |V|} \tag{13}$$

where $|V|$ is the size of the vocabulary (number of unique words in the training data). As such, we eliminate the use of word replacement by '*#UNK#*' and simply initialized the probability of unseen words to $\frac{1}{|V|}$. The emission parameters for existing words were then modified to equation 13. This method of smoothing gave us a minor increase in performance for EN, but unfortunately caused a significant decrease in our F-scores for FR. The raw results of using Laplace smoothing are given in the appendix B.2 but just a comparison of the entity type F-scores in given in table 9 below:

	k Smoothing	Laplace Smoothing	% Change
EN	0.4540	0.4851	6.8502%
FR	0.2161	0.0977	−54.7894%

Table 9: Entity Type F-Score Comparison

where percentage change here is given by:

$$\begin{aligned} F_k &= \text{F-score with } k \text{ smoothing} \\ F_L &= \text{F-score with Laplace smoothing} \\ \%\text{ Change} &= \frac{F_L - F_k}{F_k} \times 100\% \end{aligned} \tag{14}$$

From these results, it is inconclusive as to which method of smoothing is more effective because even though Laplace smoothing gave an increase in the tagging performance for EN, there was a large dip in performance for FR. As such, we stick to using k smoothing for the remainder of our analysis.

2.1.3 Hyperparameter Tuning

We also attempted to tune the hyperparameter k used in our k-smoothing as given in the design brief. In order to get a idea of what the optimal value of k would be, we generated a plot of entity and entity type F-scores against k values for both the EN and FR data sets. The results are presented below:

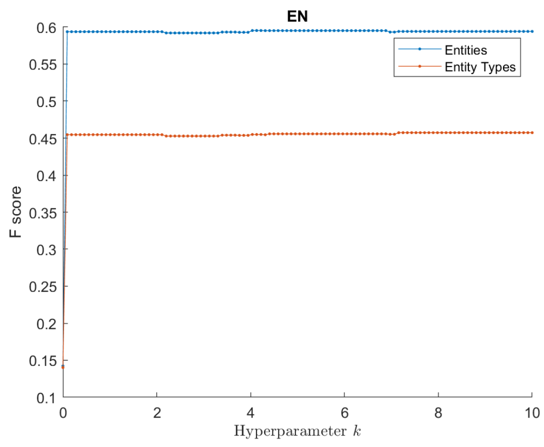


Figure 3: F-Scores vs k (EN)

Entities in gold data : 802 **Entities in prediction**: 872			
Entities		**Entity Type**	
Correct	500	**Correct**	406
Precision	0.5734	**Precision**	0.4656
Recall	0.6234	**Recall**	0.5062
F score	0.5974	**F score**	0.4851

Table 10: k Tuned Results (EN)

For EN, we found that the optimal hyperparameters should be set as $k_{max} = 4.1284$. Picking this parameter, we ran our second-order Viterbi with shrinkage to get the results seen in table 10 above. We can see that these results are a significant improvement from those in appendix B.1, where we employed shrinkage with parameter value $k = 1$.

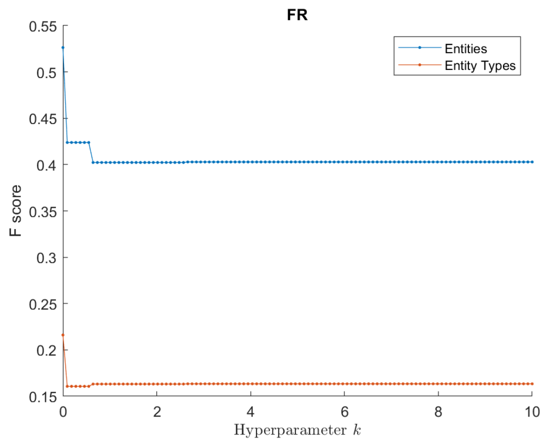


Figure 4: F-Scores vs k (FR)

Entities in gold data : 238 **Entities in prediction**: 123			
Entities		**Entity Type**	
Correct	95	**Correct**	39
Precision	0.7724	**Precision**	0.3171
Recall	0.3992	**Recall**	0.1639
F score	0.5263	**F score**	0.2161

Table 11: k Tuned Results (FR)

As for FR, we see that the optimal value of k is actually $0 (k_{max} = 0)$. As such set this value and run the algorithm to get the results in table 11. Similarly to EN, we see that these results are significant better from those in appendix B.1 with shrinkage and parameter value $k = 1$.

2.1.4 README 1 (Part 5)

In this section, we will give a description on how to run our code for (*part5.py*) modifications discussed thus far for part 5. Similar to part 4, We separated our code into 2 files, namely *part5_fun.py* and *part5.py*. These files are:

- *part5_fun.py*: This *.py* file contains the functions written for and utilized in *part5.py*. This file is imported in *part5.py*.
- *part5.py*: This *.py* contains the 'main' code block used to run the transition and emission parameter generation, and second-order Viterbi algorithm to predict the tags for the test data.

To change the test data, you can simply edit the code block entitled 'validation set' (2) written up the same way as in part 4. Proceed to run the code, which will also print the results/scores. *part5.py* can also be run on the python notebook entitled *test.ipynb*.

§2.2 Perceptron for Parameter Training

To better improve the performance of tagging for the design challenge, we have decided to also implement a simple *perceptron*. The approach was inspired by Michael Collins's paper on "*Discriminative Training Methods for Hidden Markov Models: Theory and Experiments with Perceptron Algorithms*" [3]. It is an error-driven (hinge-loss) algorithm that updates the weights for wrong predictions. We will explore how this algorithm performs by using the metrics of Precision, Recall, and F-scores as a measure of performance. The explanations and examples given below are referenced from the paper.

2.2.1 Implemented Algorithm

For clarity, we present the algorithm adapted from the paper below:

Algorithm:

1. We initialize a vector of weights $\bar{\alpha} = 0$.
2. For the specified number of epochs (T):
   - For all all sentences in the training data set:

$$\bar{y}_i = \arg\max_{y \in \mathrm{Y}} = \Phi(x_i, z) \cdot \bar{\alpha} \tag{15}$$

$$\bar{\alpha} = \bar{\alpha} + [\Phi(x_i, y_i) - \Phi(x_i, \bar{y}_i)] \cdot [[\bar{y}_i \neq y_i]] \tag{16}$$

   where $\bar{y}$ is the predicted tag, Y is the set of all possible tags and Φ is the count information from the training data.
3. Repeat step 2 until convergence and output $\bar{\alpha}$.
4. Average (normalize) the parameters via the formula:

$$\gamma_j = \sum_{t=1}^{T} \sum_{i=1}^{N} \left(\frac{\alpha_j^{t,i}}{NT} \right) \tag{17}$$

   where N is the number of tweets in the training set and j indicates the j-th parameter in the vector.

For a more high-level summary of this algorithm, refer to the write-up below:

```
* Preprocess the words by converting them to lowercase using lower.().
* Replace low frequency word, if a word occurs
    less than k number of times remove it.

From our list of observations

* Initialise Parameters
* For the specified number of epochs
    * for sentence
        * Based on the current weights,
            run the viterbi algorithm to generate predictions
        * Check if the predictions are correct
            * if the prediction is incorrect we increase
                the weights associated with the correct tag,
                and we reduce the weight associated with the
                incorrectly guessed tag
* Average Parameters
* Predict using averaged parameters.
```

For this implementation, there are 3 hyperparameters we explored. We wrote a script along with functions to analyze how varying each parameter changes the accuracy of the model. For every epoch t, of the training, we would assess the accuracy of the model using the validation set so that we could see how the weight updates from each cycle is changes the performance. The python notebook included in our submission visualizes some of these results and plots their performance over time. The 3 hyperparameters we experimented with are:

1. Number of epochs, T.
2. k value.
3. Learning rate, lr (the amount by which we change the weights).

From our experiments, we found that these combinations of values produced the best results (significant improvements) on the validation set with just the first-order implementation of the Viterbi algorithm.

2.2.2 Results from Perceptron

For the EN data set, we ran $T = 53$ epochs with associated hyperparameters $k = 1$ and $lr = 1$ to get a F-scoring of '*Entity*': 0.707347 and 'Entity Type': 0.642590. A more thorough breakdown of the scoring is given in table 12 below. We also generated a plot of F-scoring against epoch number to get a visualization of the our perceptron on the training data. This is given in figure 5 below.

Entities in gold data : 802 **Entities in prediction**: 807			
Entities		**Entity Type**	
Correct	567	**Correct**	516
Precision	0.7026	**Precision**	0.6394
Recall	0.7070	**Recall**	0.6434
F score	0.7048	**F score**	0.6414

Table 12: Perceptron Results (EN)

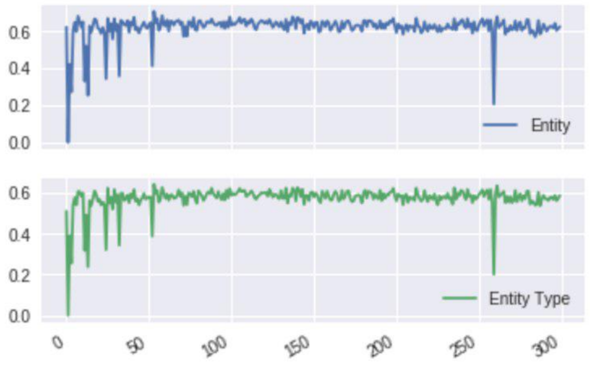


Figure 5: F-Score vs Epoch (EN)

For the FR data set, we ran $T = 16$ epochs with associated hyperparameters $k = 2$ and $lr = 1$ to get a F-scoring of '*Entity*': 0.687631 and 'Entity Type': 0.410901. A more thorough breakdown of the scoring is given in table 13 below. We similarly generated a plot of F-scoring against epochs to get a visualization of the our perceptron on the training data. This is given in figure 6 below.

Entities in gold data : 238			
Entities in prediction: 240			
Entities		**Entity Type**	
Correct	164	**Correct**	96
Precision	0.6833	**Precision**	0.4000
Recall	0.6891	**Recall**	0.40304
F score	0.6862	**F score**	0.4017

Table 13: Perceptron Results (FR)

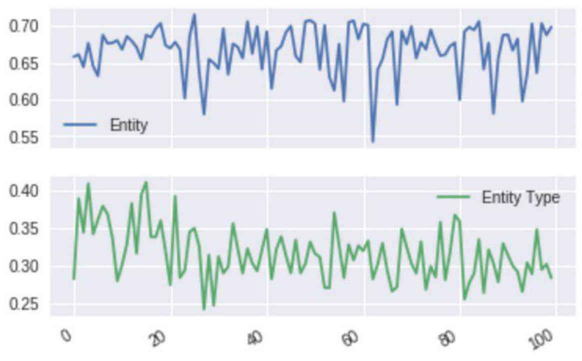


Figure 6: F-Score vs Epoch (FR)

Thus, we see that the results of the perceptron on just a first-order belief model out performed the second-order hidden Markov model with shrinkage and parameter tuning improvements.

2.2.3 README 2 (Part 5)

To run this code and get the same results we did, please use Python 2.7.9.

Below is an example of how to run the script on the command line:

```
python perceptron.py -traindata ../EN/train -testdata ../EN/dev.in -output
    ../EN/test.p5.out -k 1 -epochs 54
```

- *traindata*: Location of language file to train on eg ../EN/train
- *testdata*: Location of language test file to train on eg ../EN/dev.in
- *output*: Location of language test file to train on eg ../EN/test.p5.out
- *k*: The parameter is used here to remove words that occur less the k times.
- *epochs*: This is the number of epochs to train the perceptron.

A rough overview of the algorithm's class architecture and how to run complete training and prediction is given below:

```
model = Model(training_data_path, k=2)
model.readData()
model.replaceWord()
model.createEParams()
model.createTParams()
model.trainPerceptron(epochs=200, lr=1.0)
model.averageWeights()
model.predict(validation_data_path,output_data_path)
```

APPENDIX

§A Raw Results

§A.1 Part 2 Results (CN, SG)

Entities in gold data : 1081 **Entities in prediction**: 5161			
Entities		**Entity Type**	
Correct	602	**Correct**	354
Precision	0.1166	**Precision**	0.0686
Recall	0.5569	**Recall**	0.3275
F score	0.1929	**F score**	0.1134

Table 14: Part 2 Results (CN)

Entities in gold data : 4092 **Entities in prediction**: 12125			
Entities		**Entity Type**	
Correct	2394	**Correct**	1217
Precision	0.1974	**Precision**	0.1004
Recall	0.5850	**Recall**	0.2974
F score	0.2952	**F score**	0.1501

Table 15: Part 2 Results (SG)

§A.2 Part 3 Results (CN, SG)

Entities in gold data : 1081 **Entities in prediction**: 1892			
Entities		**Entity Type**	
Correct	502	**Correct**	295
Precision	0.2653	**Precision**	0.1559
Recall	0.4644	**Recall**	0.2729
F score	0.3377	**F score**	0.1985

Table 16: Part 3 Results (CN)

Entities in gold data : 4092 **Entities in prediction**: 5301			
Entities		**Entity Type**	
Correct	1946	**Correct**	1088
Precision	0.3671	**Precision**	0.2052
Recall	0.4756	**Recall**	0.2659
F score	0.4144	**F score**	0.2317

Table 17: Part 3 Results (SG)

§B Part 5 Results (EN, FR)

§B.1 Shrinkage Results (2nd Order HMM)

Entities in gold data : 802 **Entities in prediction**: 852			
Entities		**Entity Type**	
Correct	491	**Correct**	376
Precision	0.5763	**Precision**	0.4413
Recall	0.6122	**Recall**	0.4688
F score	0.5937	**F score**	0.4547

Table 18: Shrinkage Results (EN)

Entities in gold data : 238 **Entities in prediction**: 473			
Entities		**Entity Type**	
Correct	143	**Correct**	58
Precision	0.3023	**Precision**	0.1226
Recall	0.6008	**Recall**	0.2437
F score	0.4023	**F score**	0.1632

Table 19: Shrinkage Results (FR)

§B.2 Laplace Smoothing Results

Entities in gold data : 802 **Entities in prediction**: 872			
Entities		**Entity Type**	
Correct	500	**Correct**	406
Precision	0.5734	**Precision**	0.4656
Recall	0.6234	**Recall**	0.5062
F score	0.5974	**F score**	0.4851

Table 20: Laplace Smoothing Results (EN)

Entities in gold data : 238 **Entities in prediction**: 28			
Entities		**Entity Type**	
Correct	26	**Correct**	13
Precision	0.9286	**Precision**	0.4643
Recall	0.1092	**Recall**	0.0546
F score	0.1955	**F score**	0.0977

Table 21: Laplace Smoothing Results (FR)

References

[1] Park, J., Chebbah, M., Jendoubi, S., Martin, A. (2014). *Second-Order Belief Hidden Markov Models*. Belief Functions: Theory and Applications Lecture Notes in Computer Science, 284-293. doi:10.1007/978-3-319-11191-9_31

[2] Boodidhi, S. (2011). *Using smoothing techniques to improve the performance of Hidden Markov's Model*. Retrieved from UNLV Theses, Dissertations, Professional Papers, and Capstones.

[3] Collins, M. (2002). *Discriminative training methods for hidden Markov models*. Proceedings of the ACL-02 Conference on Empirical Methods in Natural Language Processing - EMNLP 02. doi:10.3115/1118693.1118694